Hardware Modeling [VU] (191.011)
– WS24 –
Introduction to Hardware Design

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

Welcome to the very first lecture of this year's hardware modeling course! I will be the voice guiding you through the lecture part and in this first lecture we will introduce you to hardware design in general. In this lecture we will discuss what hardware modeling actually is, what makes it stand apart from writing software and why you need to care about it. After you have finished this video, you can explain the differences between writing software and programmatically describing hardware and motivate why it is necessary to design hardware in some cases. Furthermore, you can discuss why general purpose hardware designs are not suitable for some applications.

HW Design
Motivation
SW Comparison
Hardware Design

Hardware Modeling [VU] (191.011) – WS24 –

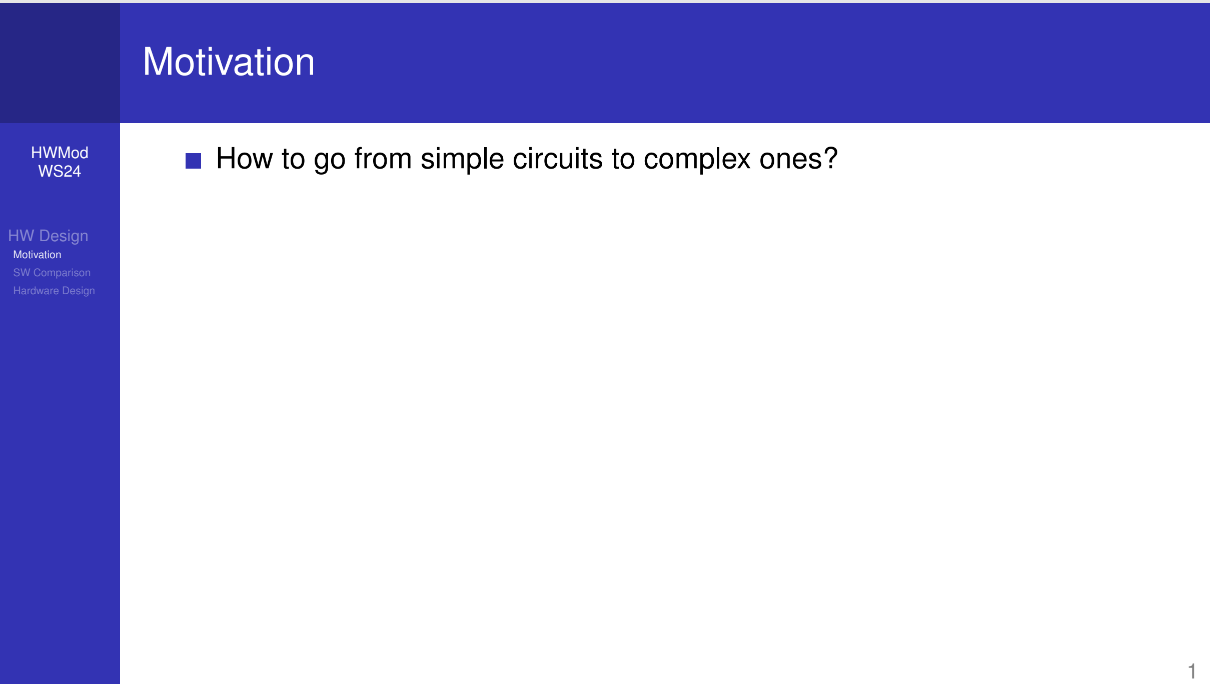
Introduction to Hardware Design

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

- How to go from simple circuits to complex ones?

So far in your studies, you have learned how to write programs that make general purpose computing devices perform desired tasks. If we simplify things, such devices consist of a central processing unit and a memory, which are very complex circuits oftentimes comprising billions of distinct transistors.

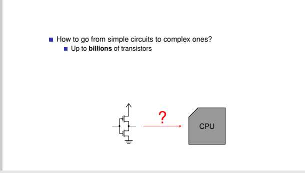


Motivation

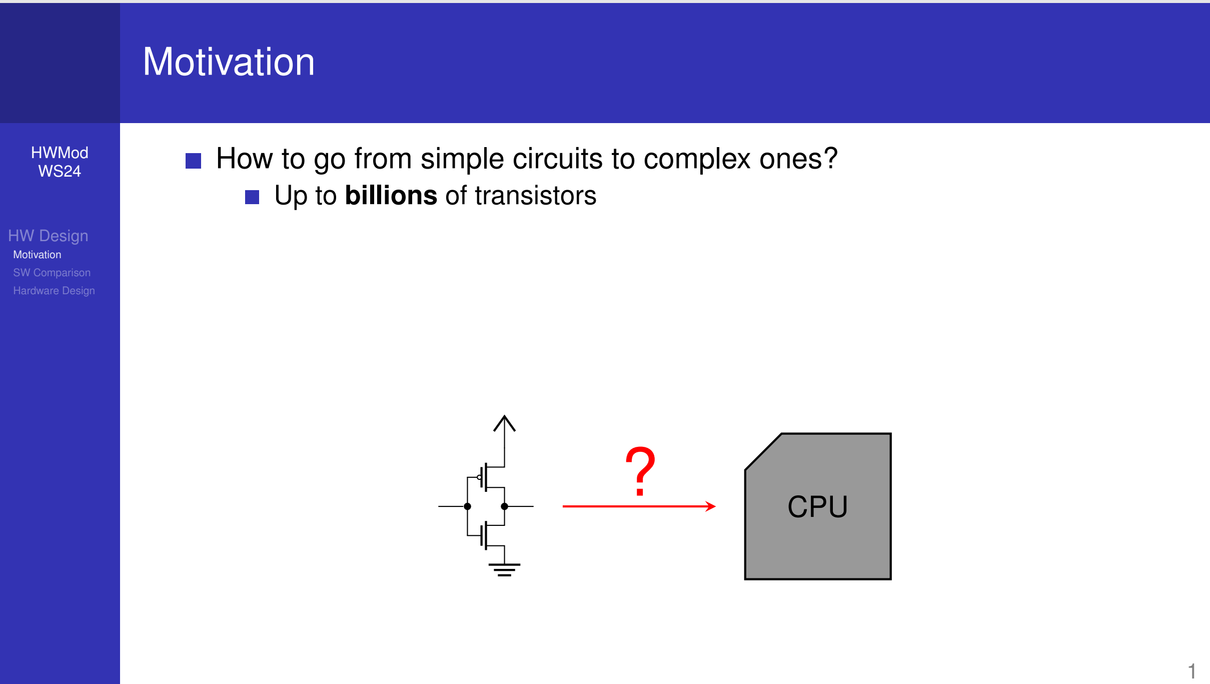
HWMod WS24

HW Design
Motivation
SW Comparison
Hardware Design

- How to go from simple circuits to complex ones?

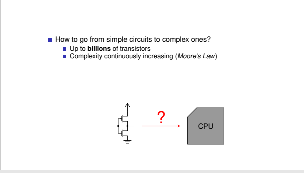


In previous courses, you have then been taught how these CPUs, memories and other components operate and how transistors can be used to build simple digital circuits. However, have you ever wondered how the gap between simple and complex circuits can be bridged? In particular, how a complex system like a CPU can be designed? How do hardware designers handle the billions of transistors and make them do something meaningful?



Motivation

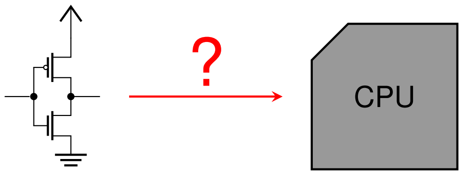
- How to go from simple circuits to complex ones?
  - Up to **billions** of transistors

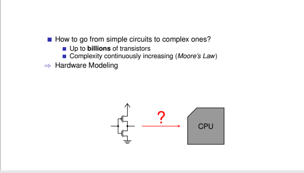


Are these questions important? Certainly! The transistor is easily the most-fabricated thing since the dawn of humankind. While you were listening to the previous sentence, multiple *trillion* transistors got manufactured, with hundreds of quintillions of transistors per year. Furthermore, if we consider the path and current increase in the complexity of digital circuits, it is not far-fetched to assume that complexity will further grow in the years to come.

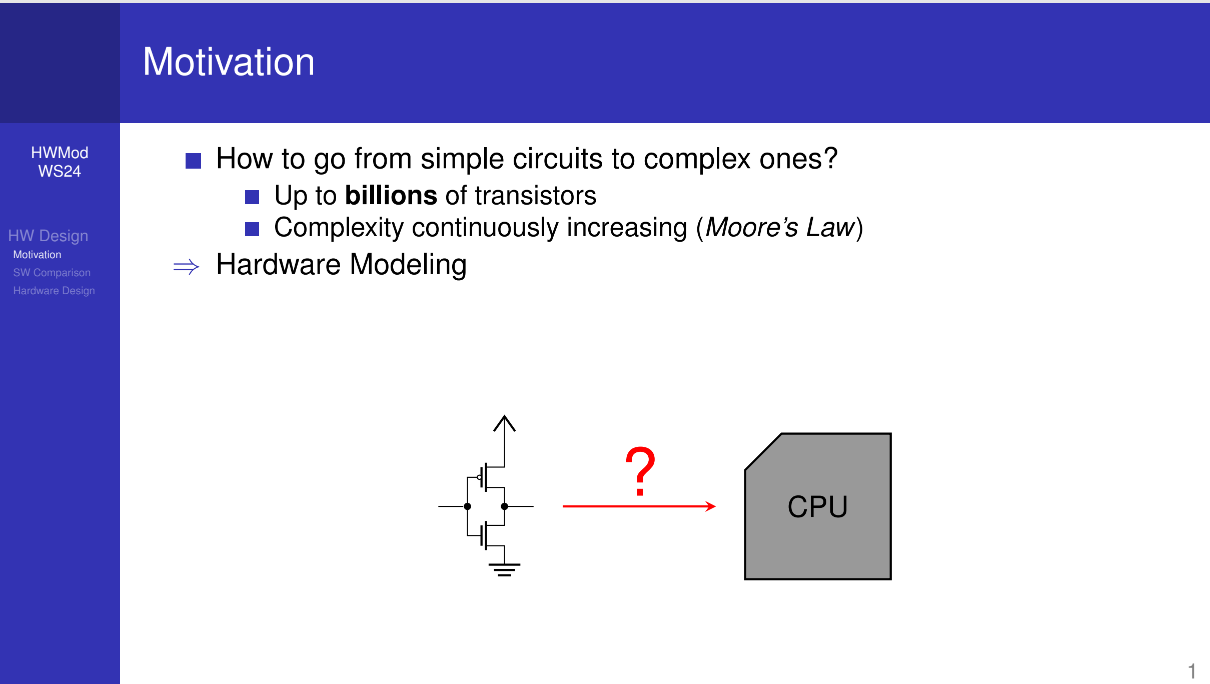
Motivation

- How to go from simple circuits to complex ones?
  - Up to **billions** of transistors
  - Complexity continuously increasing (*Moore's Law*)





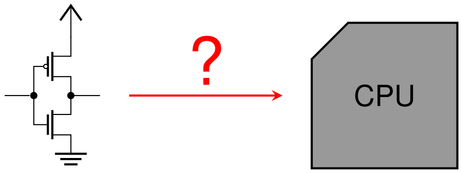
These dimensions are so far from what we encounter on a daily basis that our brains can hardly comprehend them. And yet, apparently humans are able to tame these immense quantities, putting the billions of transistors per single chip to use, driving the technological advancements of humankind for decades. This is *exactly*, what hardware modeling is about and what we will be concerned with in this course.

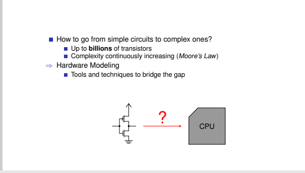


Motivation

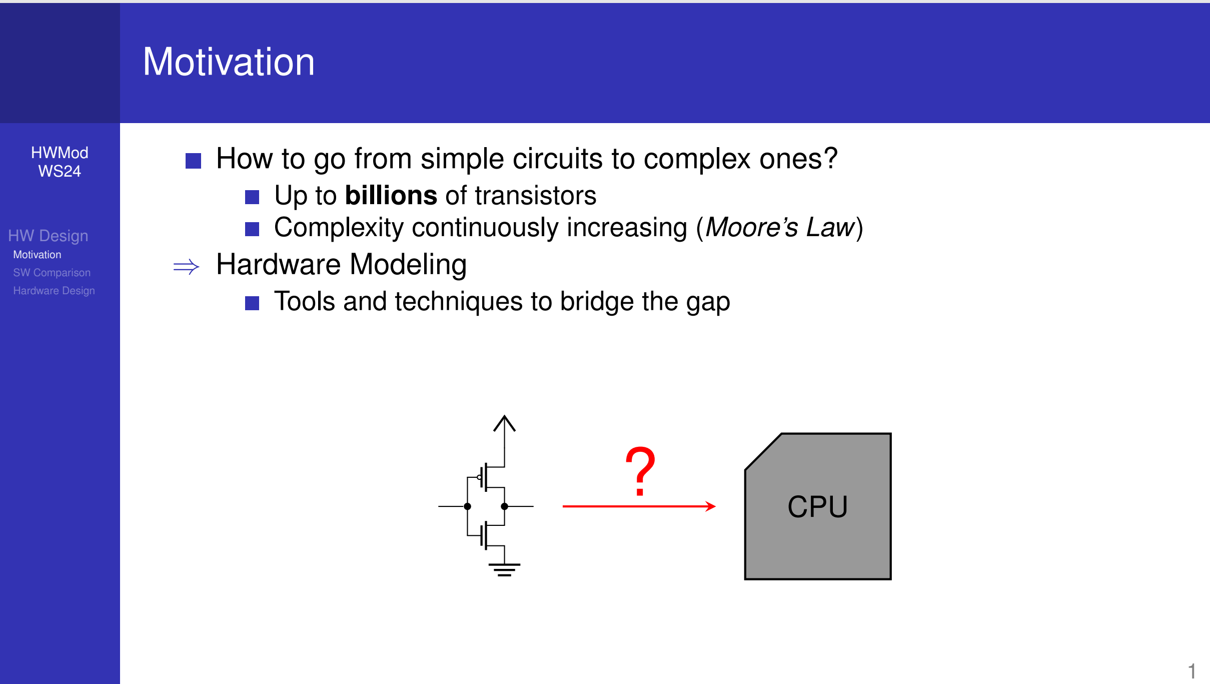
- How to go from simple circuits to complex ones?
  - Up to **billions** of transistors
  - Complexity continuously increasing (*Moore's Law*)

⇒ Hardware Modeling





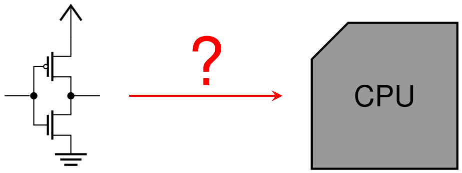
We will teach you tools and techniques to bridge the gap between simple digital circuits, that can easily be designed per hand, and complex circuits consisting of up to billions of transistors. And after successful completion of this course, you will be able to tame the myriads of transistors of modern technologies yourself.



Motivation

- How to go from simple circuits to complex ones?
  - Up to **billions** of transistors
  - Complexity continuously increasing (*Moore's Law*)

⇒ Hardware Modeling
  - Tools and techniques to bridge the gap

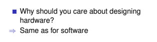


Why should you care about designing hardware?

Although we hope our little motivation sparked your interest in hardware modeling, you still might ask yourself why you should bother to learn this. After all, there is a plethora of readily-available chips. You do not need to develop your own CPU. You can simply go out and by the one you fancy. And while that is true, there is a catch.

Why bother?

- Why should you care about designing hardware?


■ Why should you care about designing hardware?
⇒ Same as for software

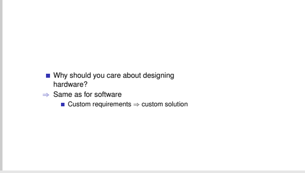
In general, the reason for caring about the design of hardware are the same for caring about writing software programs.

Why bother?

HW Design
Motivation
SW Comparison
Hardware Design

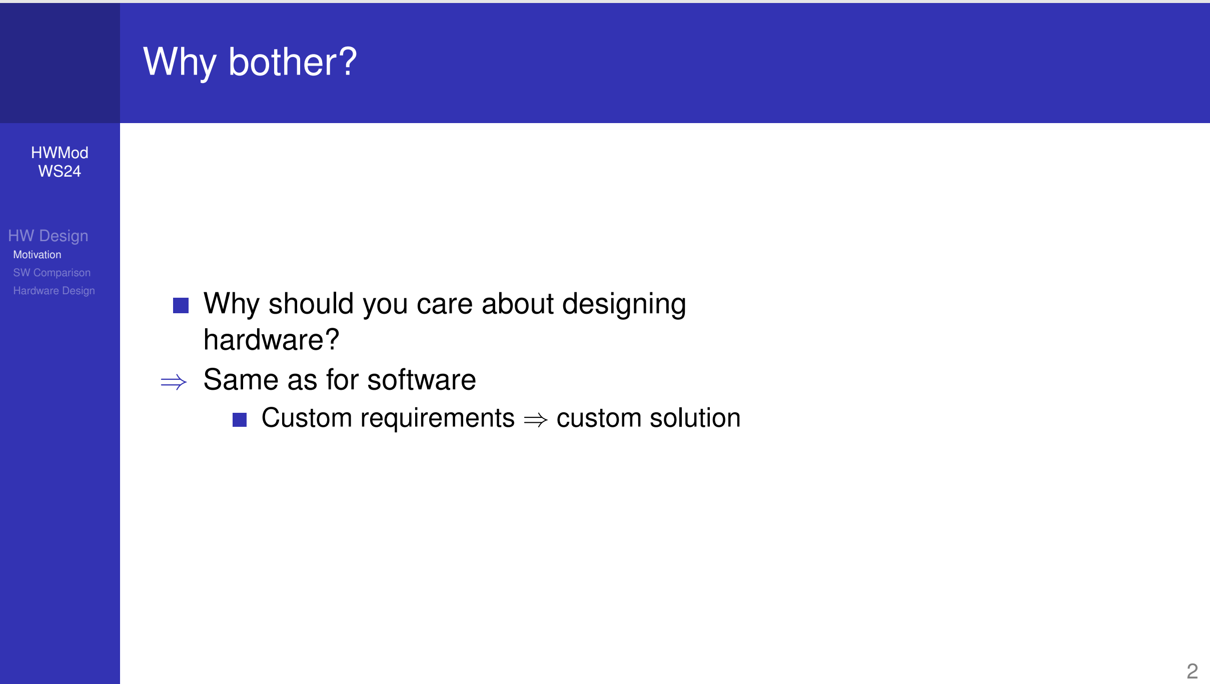
- Why should you care about designing hardware?

⇒ Same as for software


- Why should you care about designing hardware?

⇒ Same as for software
  - Custom requirements ⇒ custom solution

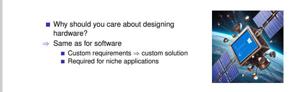
In general, the reason for caring about the design of hardware are the same for caring about writing software programs. When you download a program or piece of software, the developer will have decided about its features and properties. Usually this means that the program either excels at envisioned average use cases in order to target a broad audience, or that the program is specifically tailored to the needs of the developer. If the task for which you want to use this software strongly deviates from the applications for which it was originally created, the readily-available programs might not be suitable for you.



Why bother?

- Why should you care about designing hardware?

⇒ Same as for software
  - Custom requirements ⇒ custom solution



For hardware it is essentially the same. If you want to build a satellite you cannot simply buy the next-best consumer-grade CPU. The requirements of these two applications are completely different. For example, while power and reliability are paramount for a satellite, they are often just an afterthought for consumer-grade CPUs. Just as with software, if the available programs do not suit your needs, you either have to create it yourself or hire someone to do that for you. Typically, this is required for low-volume niche applications.

Why bother?

- Why should you care about designing hardware?

$\Rightarrow$ Same as for software
  - Custom requirements $\Rightarrow$ custom solution
  - Required for niche applications

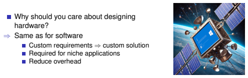
Why should you care about designing hardware?
⇒ Same as for software
Custom requirements ⇒ custom solution
Required for niche applications
Reduce overhead

Another thing to consider is overhead. While a software that has features you do not need will in its majority only require more disk space, a chip that has features you do not need draws more power than necessary. In some applications such behavior is highly undesired.

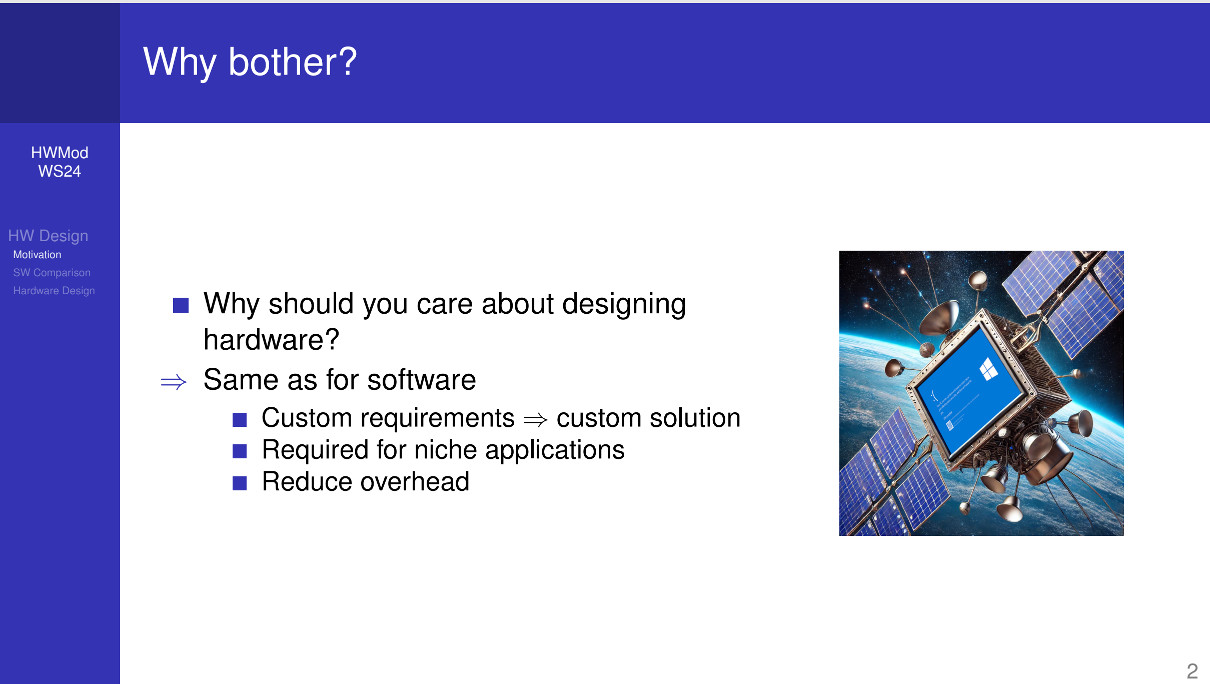
Why bother?

- Why should you care about designing hardware?

⇒ Same as for software
  - Custom requirements ⇒ custom solution
  - Required for niche applications
  - Reduce overhead



In conclusion, while hardware design might sound very niche at the beginning, the reasons to do it are actually not very different from the ubiquitous software design.



Why bother?

- Why should you care about designing hardware?

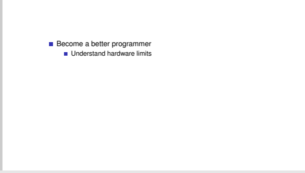
⇒ Same as for software
  - Custom requirements ⇒ custom solution
  - Required for niche applications
  - Reduce overhead

■ Become a better programmer

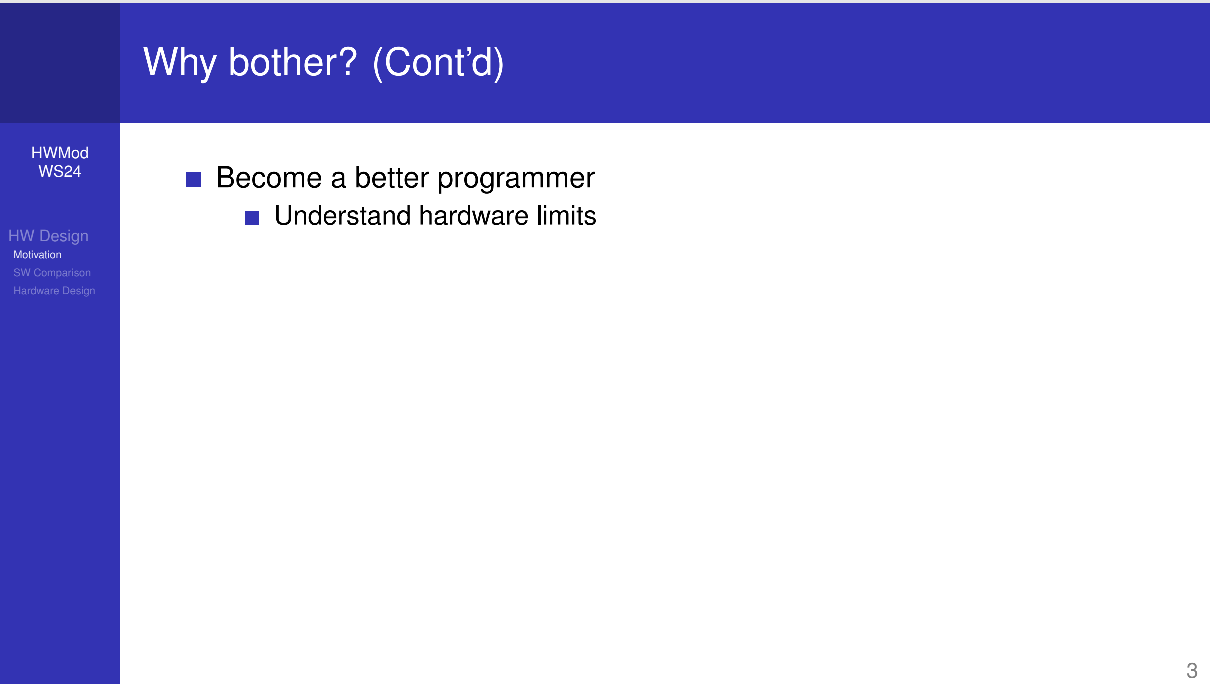
Another good reason for caring about the design of hardware is that you can become a better programmer. Regardless of how good you are at programming, the hardware on which your software runs ultimately dictates some of its key properties.

Why bother? (Cont'd)

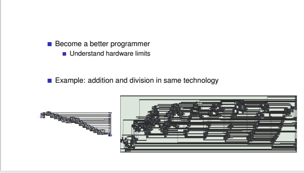
- Become a better programmer


- Become a better programmer
  - Understand hardware limits

Knowing how hardware is designed allows you to comprehend its limits, which will allow you to understand your computing platform more intimately. For example, you might already know that some operations in hardware are more expensive in terms of execution time than others. But do you also know why?


Why bother? (Cont'd)

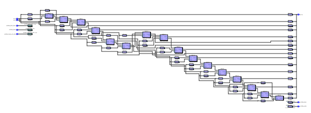
- Become a better programmer
  - Understand hardware limits

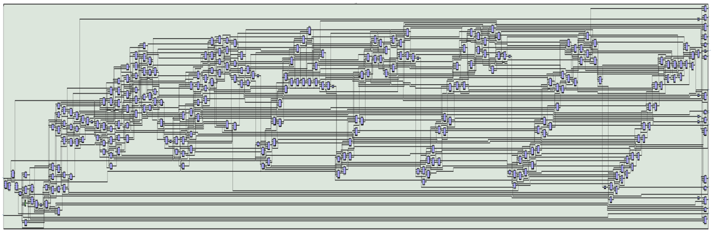


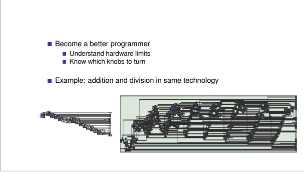
Consider the two images shown on the slide. The left one shows an addition circuit for an FPGA, whereas the right one shows a division circuit for the same types of operands and the same FPGA. You can easily see that the division operation involves significantly more hardware, leading to a longer execution time than for the addition operation.

Why bother? (Cont'd)

- Become a better programmer
  - Understand hardware limits
- Example: addition and division in same technology



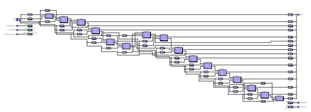


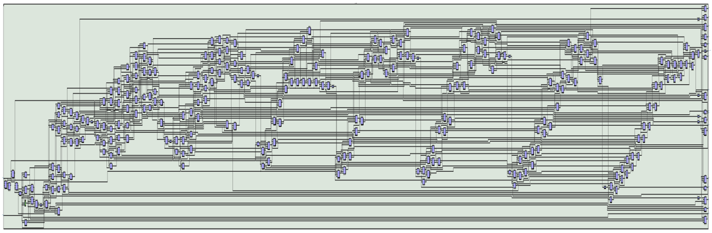


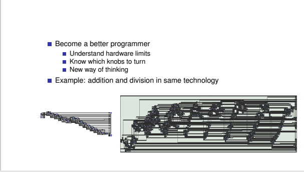
In addition to that, you will become aware of the knobs you can turn in your programming when optimizing it for certain properties.

Why bother? (Cont'd)

- Become a better programmer
  - Understand hardware limits
  - Know which knobs to turn
- Example: addition and division in same technology



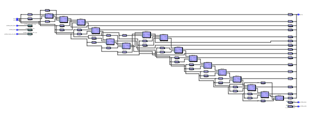


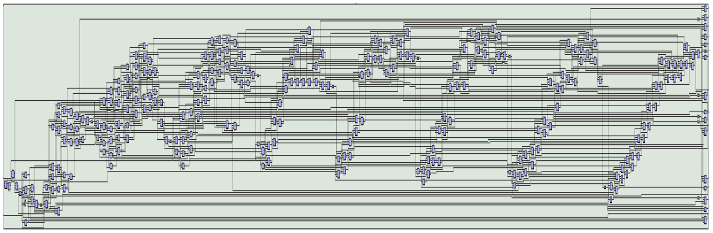


Furthermore, as we will discuss on the next slide, designing hardware requires a completely different mindset when compared to writing sequential software. This new mindset will help you to program concurrent software, which is a useful skill to have with the increasing concurrency of modern general purpose devices.

Why bother? (Cont'd)

- Become a better programmer
  - Understand hardware limits
  - Know which knobs to turn
  - New way of thinking
- Example: addition and division in same technology





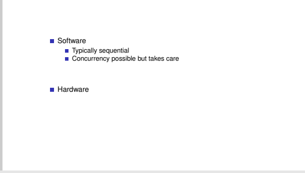
- Software
- Hardware

On the previous slide we already hinted that designing hardware requires a different way of thinking than designing software. The reason for that lies in the completely different nature of how software and hardware operate. In particular there is, to some extent, a certain duality.

Differences to Software Design

- Software

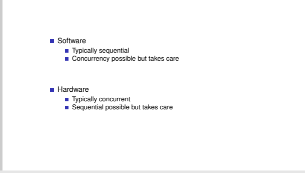
- Hardware



They way we write software is that we consider it to be a sequential problem, with one operation a time being performed. While there can be concurrency, this is rather the special than the average case and requires particular care and thought.

Differences to Software Design

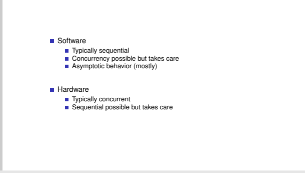
- Software
  - Typically sequential
  - Concurrency possible but takes care
- Hardware


- Software
  - Typically sequential
  - Concurrency possible but takes care
- Hardware
  - Typically concurrent
  - Sequential possible but takes care

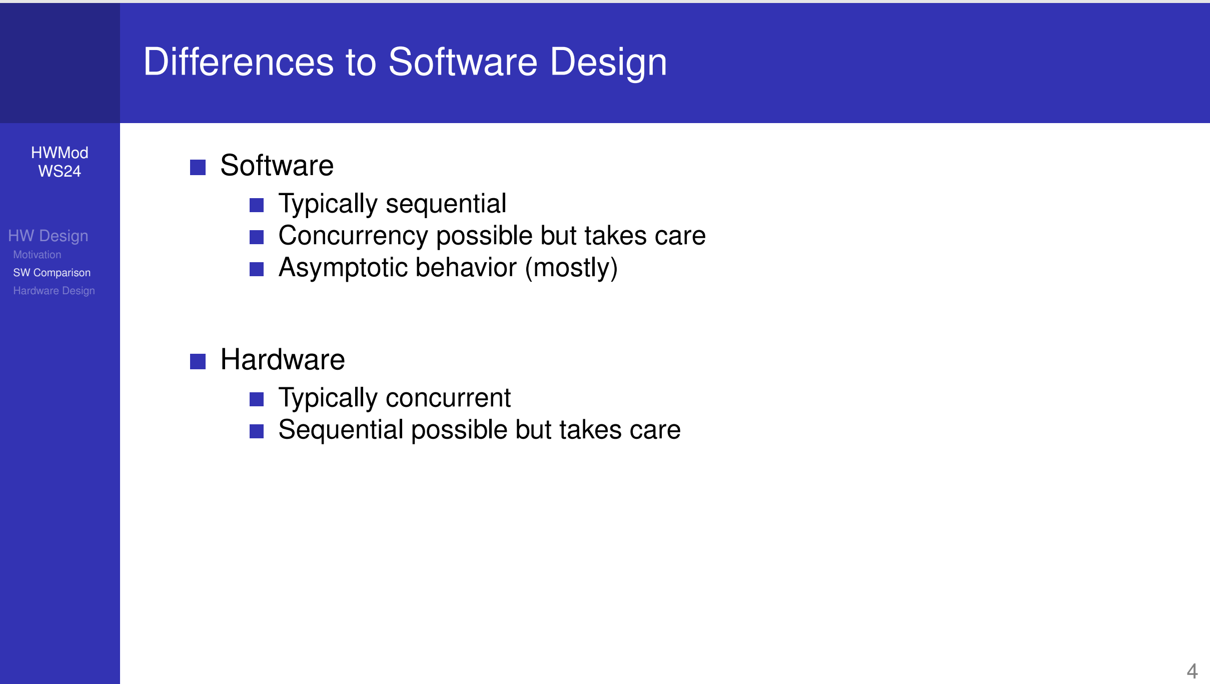
In hardware design, everything is concurrent by default, with operations typically happening in parallel. While sequential operations and thinking in hardware are not only possible but also vital, this requires more care and thought, somewhat akin to concurrent software. It is this duality that makes hardware design initially so alien but also rewarding in the end. Mastering the tools and techniques involved will ultimately teach you how to think concurrently.

Differences to Software Design

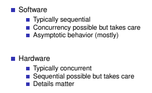
- Software
  - Typically sequential
  - Concurrency possible but takes care
- Hardware
  - Typically concurrent
  - Sequential possible but takes care


- Software
  - Typically sequential
  - Concurrency possible but takes care
  - Asymptotic behavior (mostly)
- Hardware
  - Typically concurrent
  - Sequential possible but takes care

Another difference between designing software and hardware are the considerations involved. For software, with exceptions, you are often mostly concerned with the asymptotic execution time or memory requirements. You usually do not care about single instructions or cycle times.


Differences to Software Design

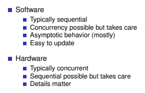
- Software
  - Typically sequential
  - Concurrency possible but takes care
  - Asymptotic behavior (mostly)
- Hardware
  - Typically concurrent
  - Sequential possible but takes care


- Software
  - Typically sequential
  - Concurrency possible but takes care
  - Asymptotic behavior (mostly)
- Hardware
  - Typically concurrent
  - Sequential possible but takes care
  - Details matter

For hardware however, attention to detail is crucial. After all, even marginal imperfections leading to a single clock cycle difference can severely impede the performance of executed programs.

Differences to Software Design

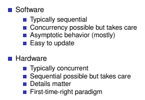
- Software
  - Typically sequential
  - Concurrency possible but takes care
  - Asymptotic behavior (mostly)
- Hardware
  - Typically concurrent
  - Sequential possible but takes care
  - Details matter


- Software
  - Typically sequential
  - Concurrency possible but takes care
  - Asymptotic behavior (mostly)
  - Easy to update
- Hardware
  - Typically concurrent
  - Sequential possible but takes care
  - Details matter

Software and hardware also significantly differ in the design flow. Where software often grows very dynamically, up to the extent where it only matures at the customer by deploying patches over time, a hardware design is eventually poured in silicon. Updating such chips is, in general, not possible, meaning that errors that happen during the development are irreversible and tremendously expensive.

Differences to Software Design

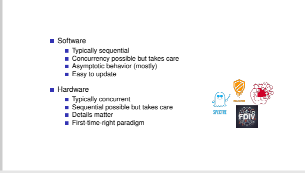
- Software
  - Typically sequential
  - Concurrency possible but takes care
  - Asymptotic behavior (mostly)
  - Easy to update
- Hardware
  - Typically concurrent
  - Sequential possible but takes care
  - Details matter


- Software
  - Typically sequential
  - Concurrency possible but takes care
  - Asymptotic behavior (mostly)
  - Easy to update
- Hardware
  - Typically concurrent
  - Sequential possible but takes care
  - Details matter
  - First-time-right paradigm

Therefore, hardware is designed with the, so-called, *first-time-right* paradigm in mind, where a design needs to be correct when it is manufactured.

Differences to Software Design

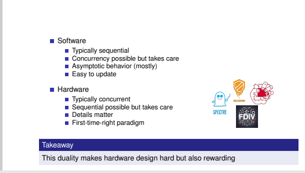
- Software
  - Typically sequential
  - Concurrency possible but takes care
  - Asymptotic behavior (mostly)
  - Easy to update
- Hardware
  - Typically concurrent
  - Sequential possible but takes care
  - Details matter
  - First-time-right paradigm



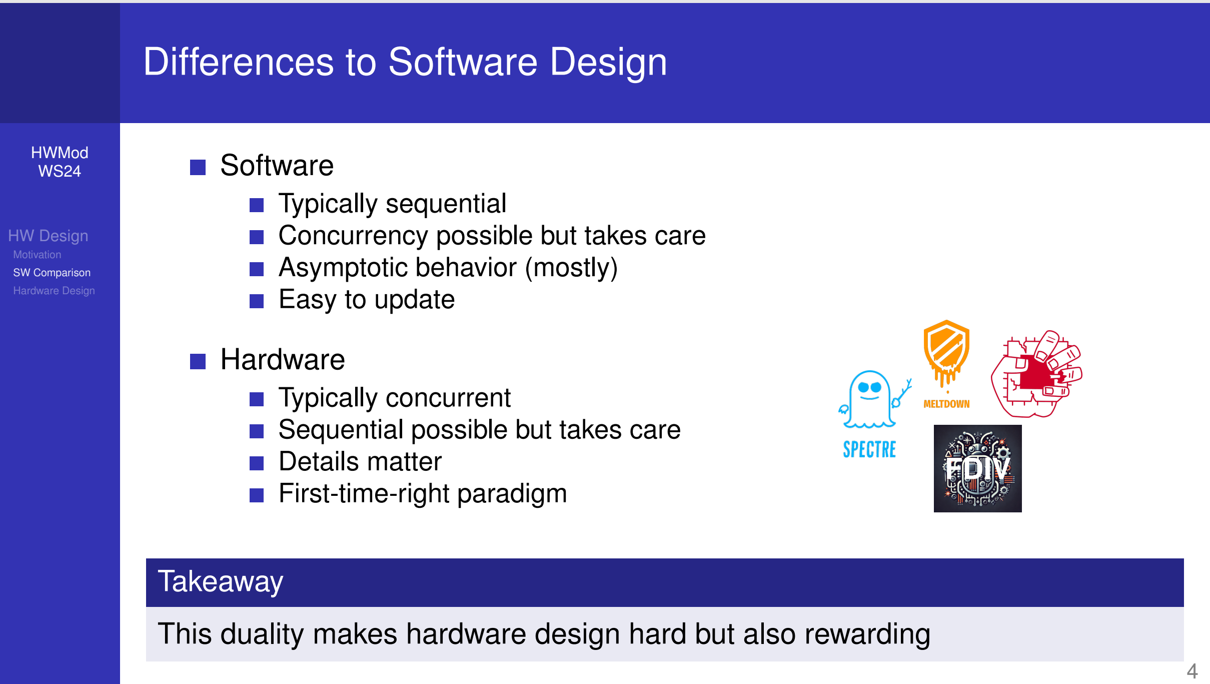
However, this is far from trivial and in the recent years we were able to witness several cases where even leading companies failed to do so. Examples range from the well-know f-diff bug in the 1990s to more recent cases such as spectre or meltdown.

Differences to Software Design

- Software
  - Typically sequential
  - Concurrency possible but takes care
  - Asymptotic behavior (mostly)
  - Easy to update
- Hardware
  - Typically concurrent
  - Sequential possible but takes care
  - Details matter
  - First-time-right paradigm



In conclusion, what you should take away from this slide is the duality between software and hardware design and that it is often the cause why skilled programmers have a hard time to initially get the grip on hardware design. However, this duality also makes designing hardware so rewarding, as it means that you can use the mindset you will develop in this course also when writing software.

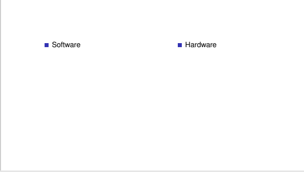


Differences to Software Design

- Software
  - Typically sequential
  - Concurrency possible but takes care
  - Asymptotic behavior (mostly)
  - Easy to update
- Hardware
  - Typically concurrent
  - Sequential possible but takes care
  - Details matter
  - First-time-right paradigm

Takeaway

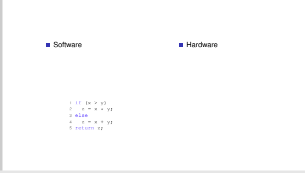
This duality makes hardware design hard but also rewarding


■ Software ■ Hardware

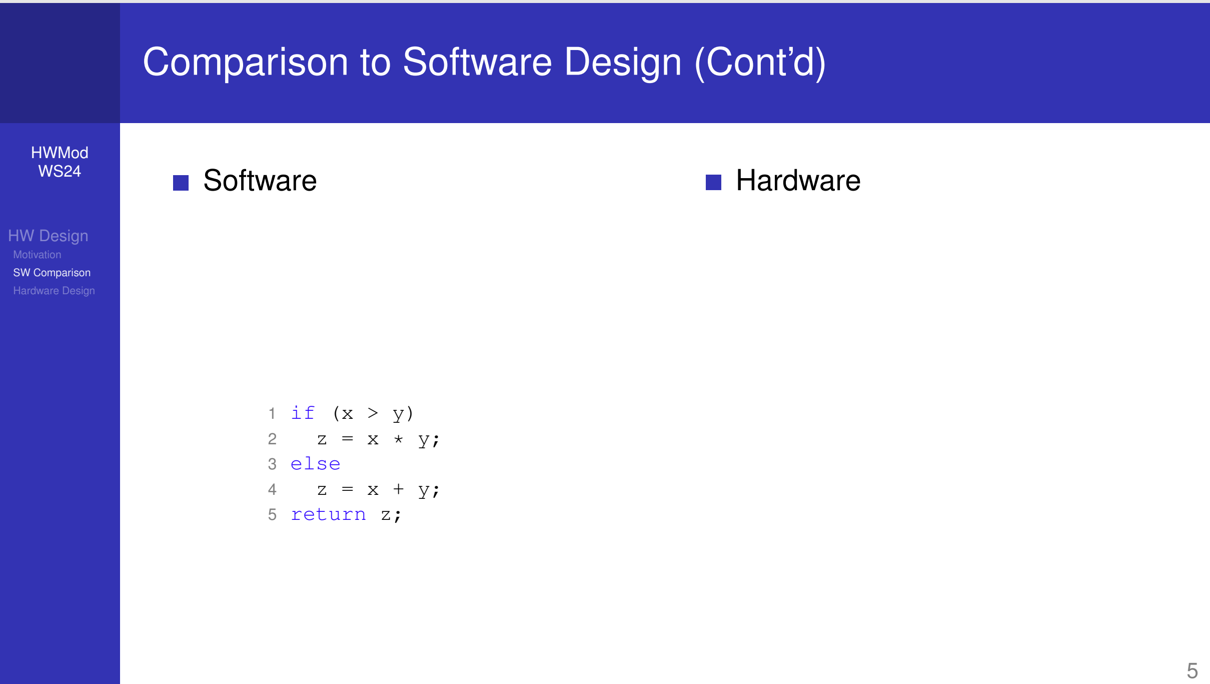
Finally, let us finish this comparison by illustrating the different nature of software and hardware.

Comparison to Software Design (Cont’d)

- Software
- Hardware



Consider the code shown on the bottom left. It consists of a simple branch that determines whether the resulting value is the sum or the product of x and y.

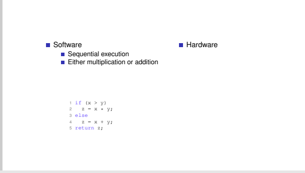


Comparison to Software Design (Cont’d)

- Software

```
if (x > y)
  z = x * y;
else
  z = x + y;
return z;
```

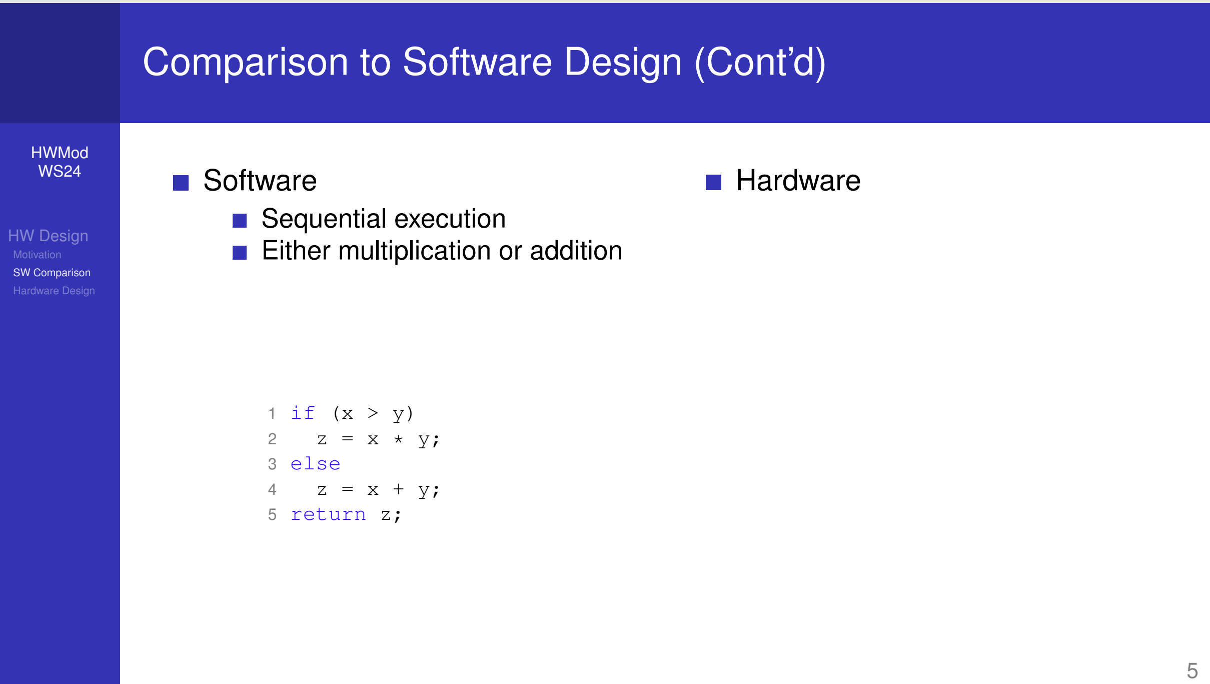
- Hardware



- Software
  - Sequential execution
  - Either multiplication or addition
- Hardware

```
if (x > y)
  z = x * y;
else
  z = x + y;
return z;
```

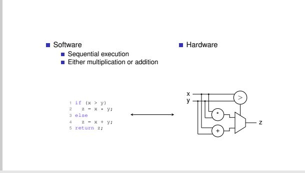
If we look at this program, we typically think about it being executed sequentially. As a result, the CPU will either compute the addition OR the product, but definitely not both.



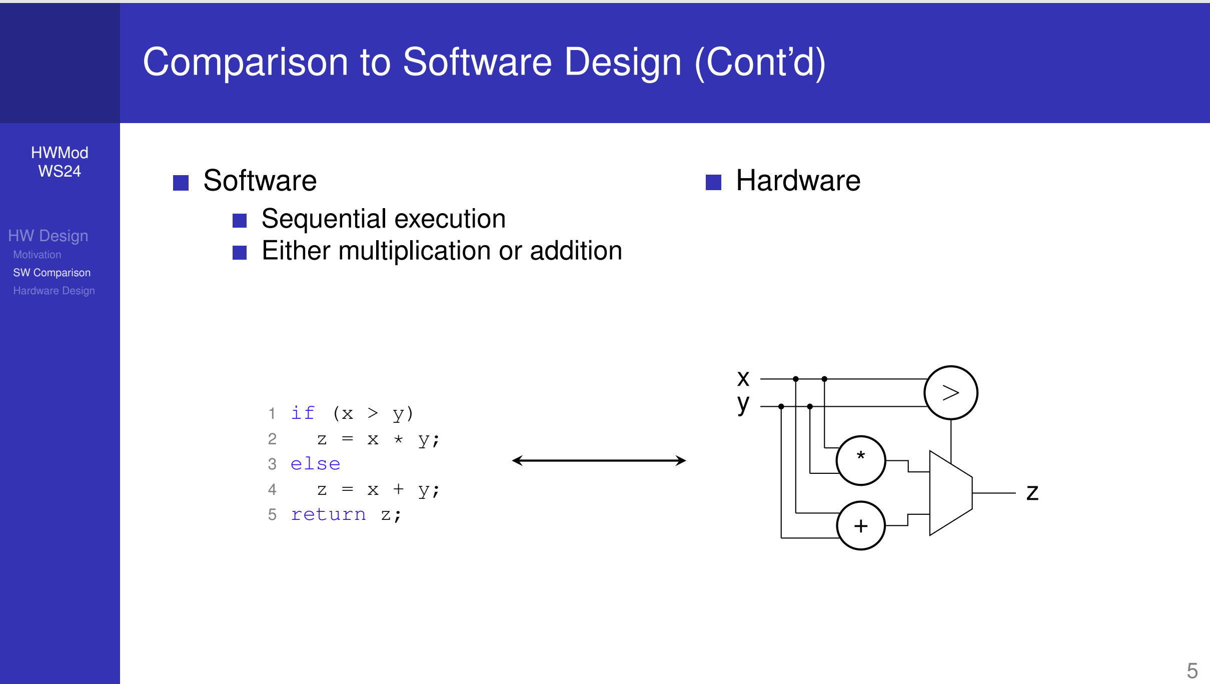
Comparison to Software Design (Cont'd)

- Software
  - Sequential execution
  - Either multiplication or addition
- Hardware

```
if (x > y)
  z = x * y;
else
  z = x + y;
return z;
```



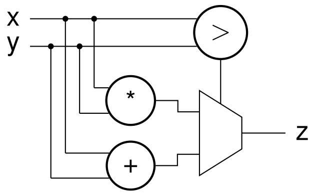
Now consider a corresponding circuit on the bottom right. It features an adder and a multiplier feeding a multiplexer and a comparator that selects the required value. The two arithmetic operations are performed by distinct circuits.

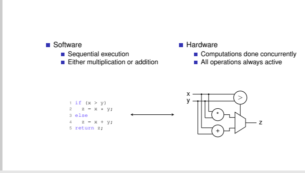


Comparison to Software Design (Cont'd)

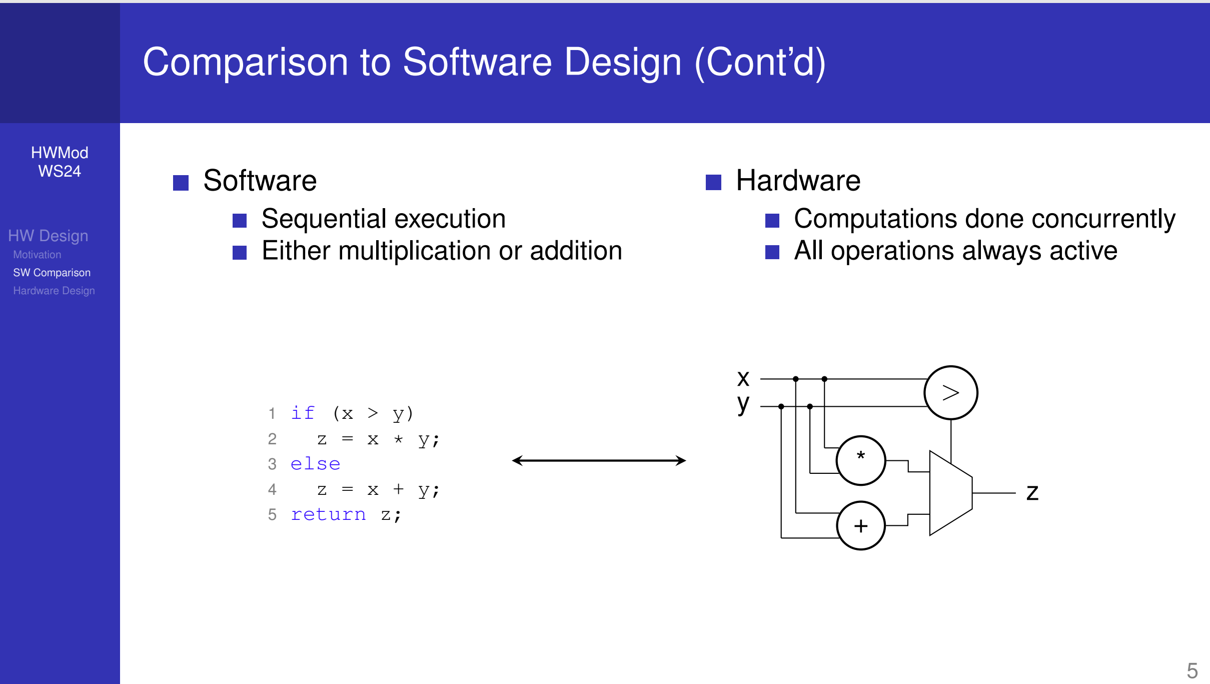
- Software
  - Sequential execution
  - Either multiplication or addition
- Hardware

```
if (x > y)
  z = x * y;
else
  z = x + y;
return z;
```





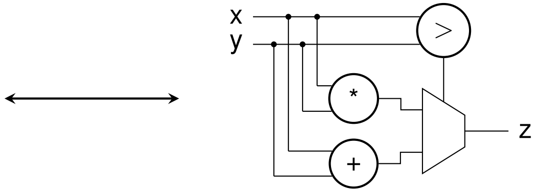
Therefore, the two values are always computed, regardless which value we actually require. This is obviously in stark contrast to the way the software performs these computations.

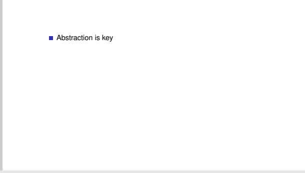


Comparison to Software Design (Cont'd)

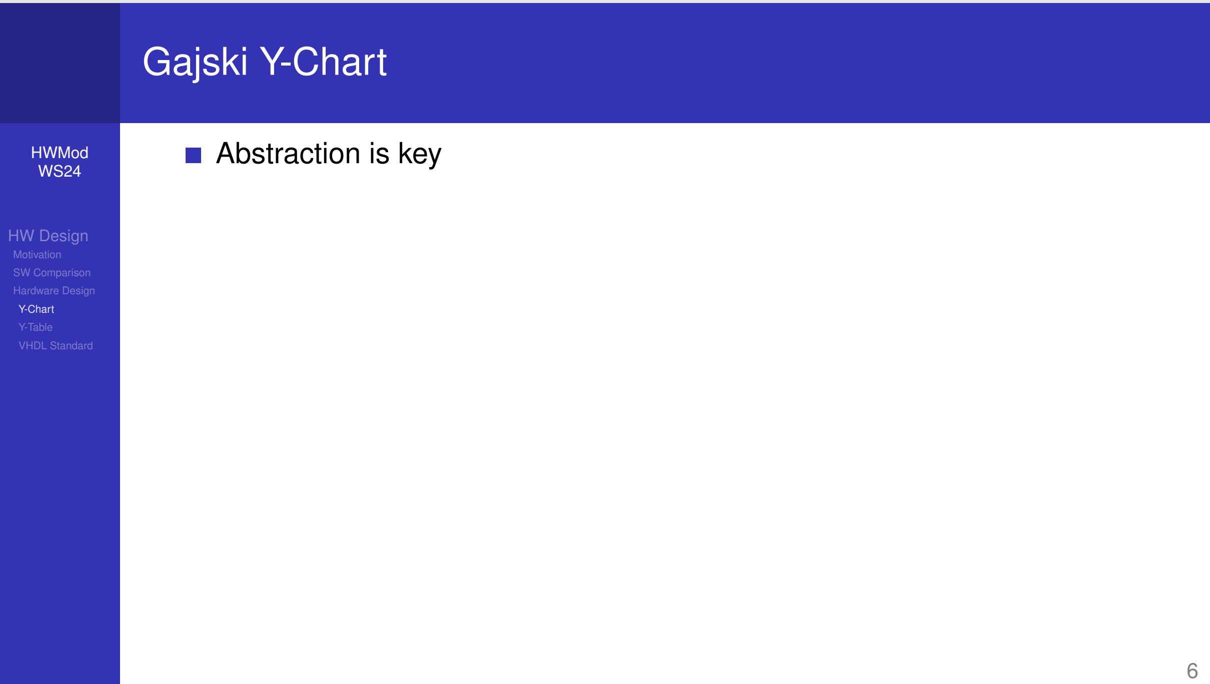
- Software
  - Sequential execution
  - Either multiplication or addition
- Hardware
  - Computations done concurrently
  - All operations always active

```
if (x > y)
  z = x * y;
else
  z = x + y;
return z;
```





Now that we convinced you that hardware design is interesting, important and fundamentally distinct from software design, let us discuss how complex circuits can actually be created systematically. As with programming complex software, abstraction is vital. We cannot constantly think about every single transistor.



Gajski Y-Chart

- Abstraction is key

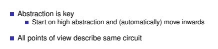
HWMod WS24
HW Design
Motivation
SW Comparison
Hardware Design
Y-Chart
Y-Table
VHDL Standard

- Abstraction is key
  - Start on high abstraction and (automatically) move inwards

Therefore, we need a means to work on a higher level of abstraction than on the one of singular circuit elements. Ideally, we would than move to lower levels of abstraction automatically using tools. However, more on that later.

Gajski Y-Chart

- Abstraction is key
  - Start on high abstraction and (automatically) move inwards



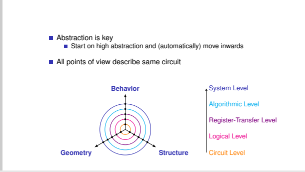
- Abstraction is key
  - Start on high abstraction and (automatically) move inwards
- All points of view describe same circuit

In addition to different levels of abstraction, a circuit can be described from different points of view. We can think about our envisioned circuit's behavior, about its structure in terms of distinct components, and its physical geometry.

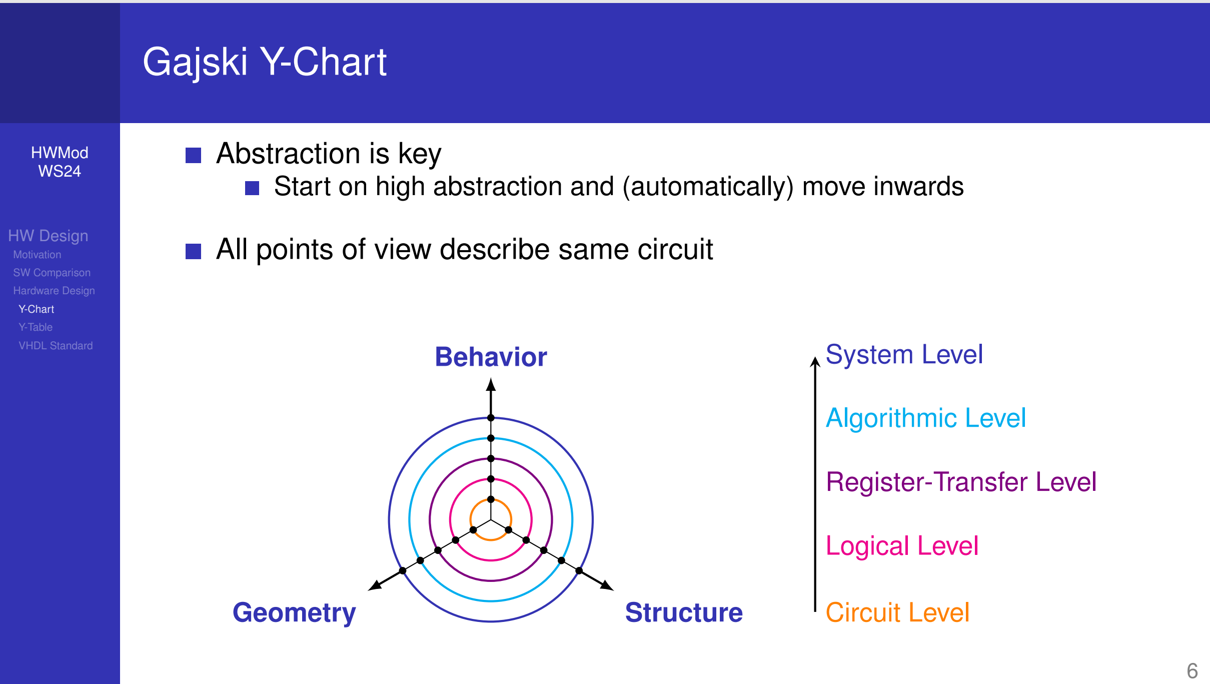
Gajski Y-Chart

HW Design
Motivation
SW Comparison
Hardware Design
Y-Chart
Y-Table
VHDL Standard

- Abstraction is key
  - Start on high abstraction and (automatically) move inwards
- All points of view describe same circuit



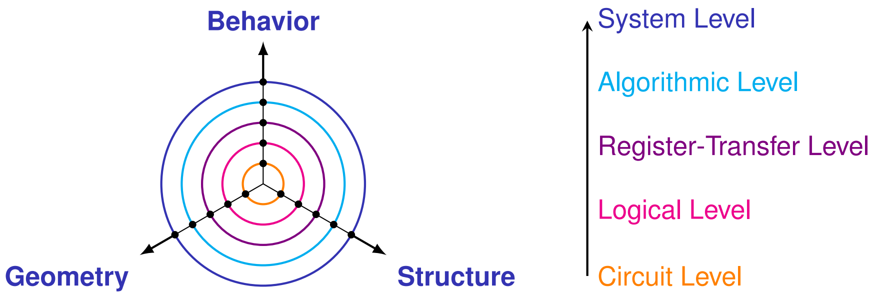
A popular illustration of these views and the different levels of abstractions is the Gajski Y-chart as depicted on the slide. The different rings of the chart correspond to different levels of abstraction, with the outermost level, called the system level, abstracting the most and the innermost level, referred to as circuit level, the least.

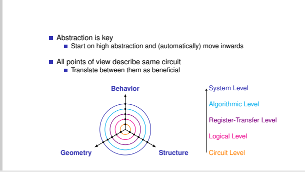


Gajski Y-Chart

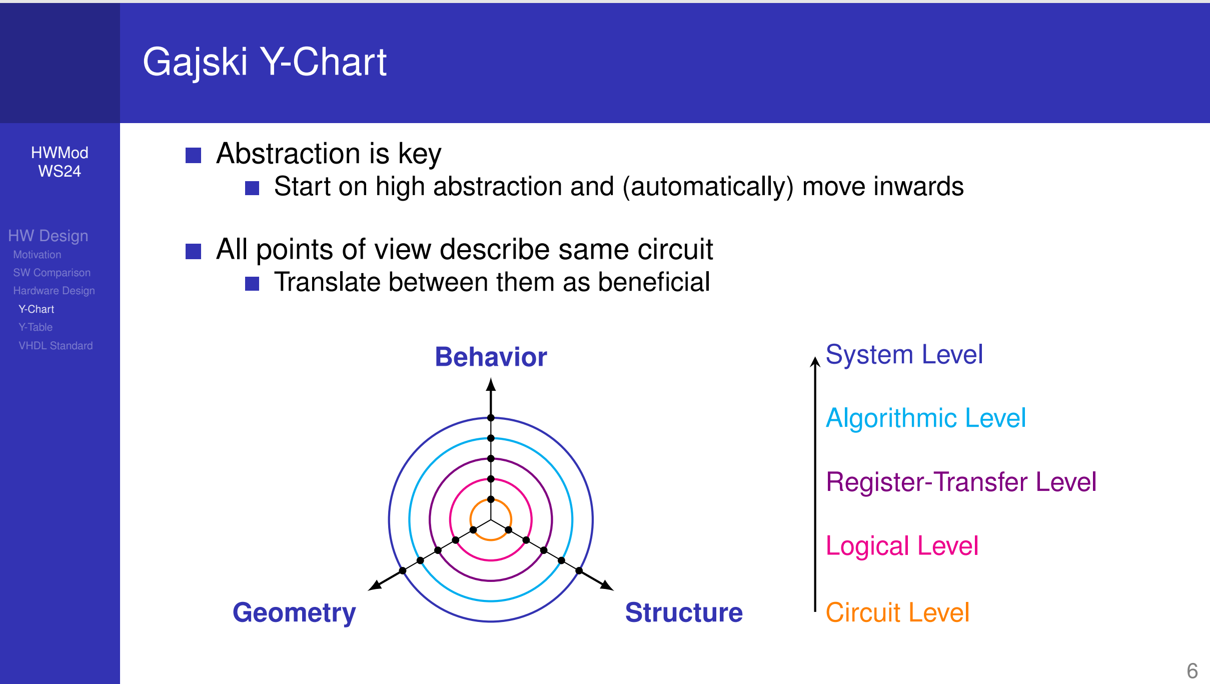
HW Design
Motivation
SW Comparison
Hardware Design
Y-Chart
Y-Table
VHDL Standard

- Abstraction is key
  - Start on high abstraction and (automatically) move inwards
- All points of view describe same circuit



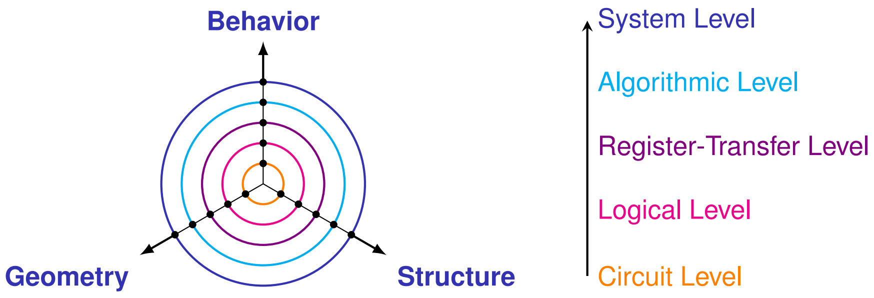


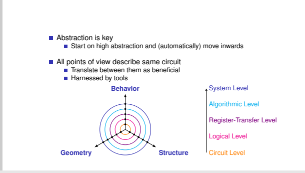
An interesting observation we can make is that all three points of view ultimately describe the same circuit on each level of abstraction. This fact can be harnessed when designing, since some viewpoints lend themselves to different levels of abstraction. In particular, this allows describing a circuit at the highest possible level of abstraction via its desired behavior and then converting it to a structural description.



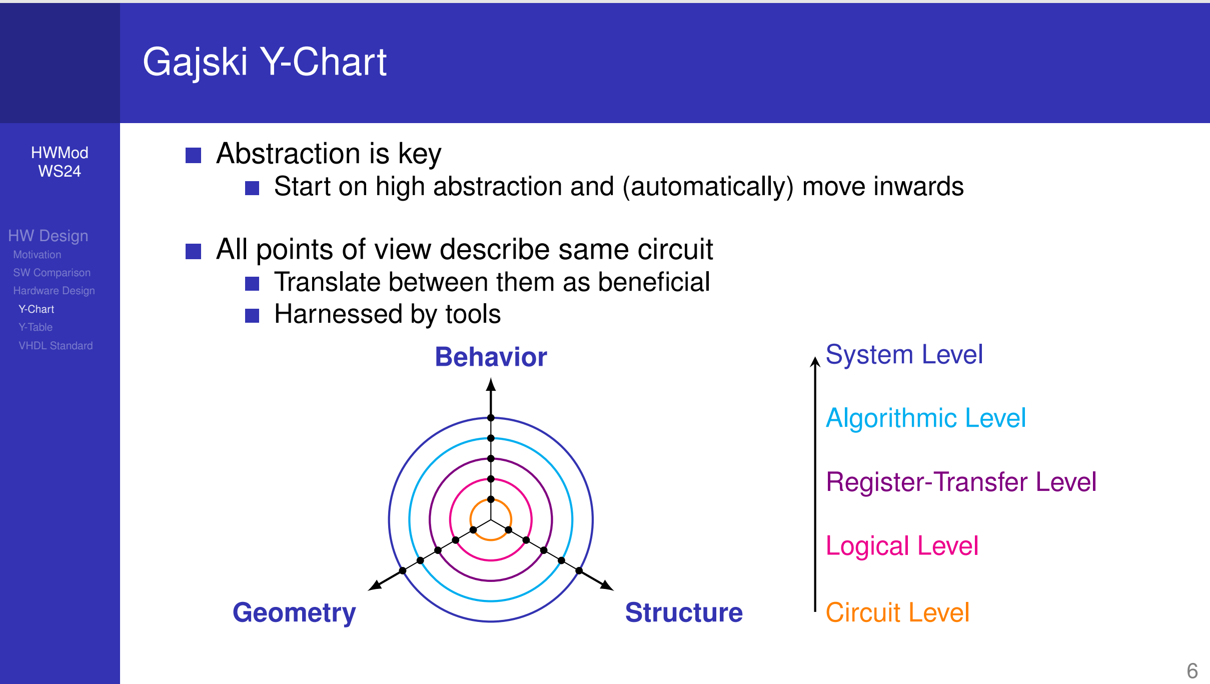
Gajski Y-Chart

- Abstraction is key
  - Start on high abstraction and (automatically) move inwards
- All points of view describe same circuit
  - Translate between them as beneficial



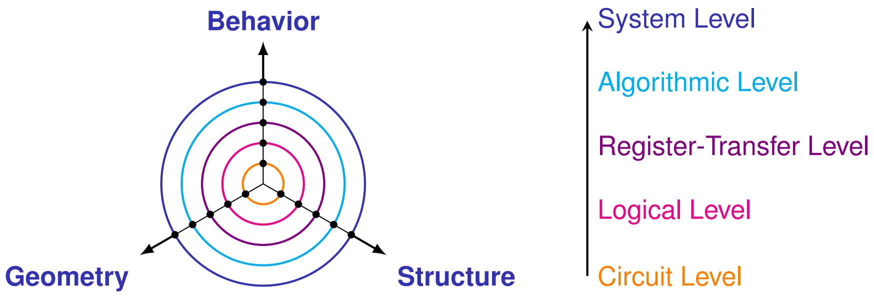


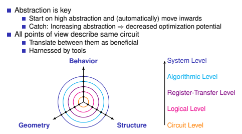
To translate between these different points of views and levels of abstraction, hardware designers use tools like the ones we will discuss in later lectures of this course.



Gajski Y-Chart

- Abstraction is key
  - Start on high abstraction and (automatically) move inwards
- All points of view describe same circuit
  - Translate between them as beneficial
  - Harnessed by tools



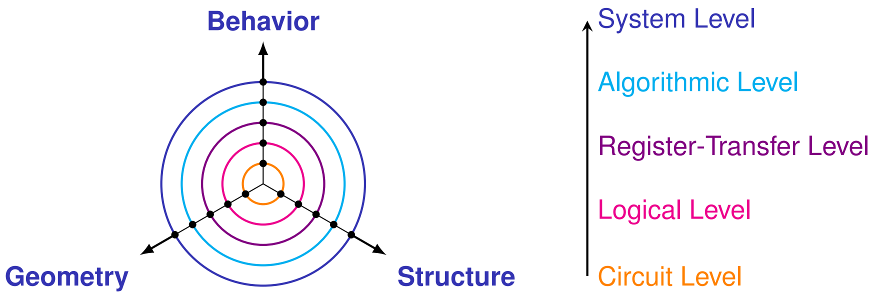


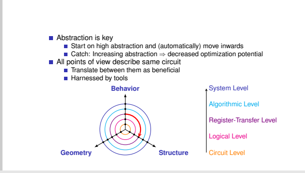
Be aware though that the higher the abstraction, the harder it is to perform optimizations. As a result it can sometimes be necessary that we work on a lower level of abstraction than we would like. For example, this is the case when we need to match strict requirements regarding timing, power, area or other properties.

Gajski Y-Chart

HW Design
Motivation
SW Comparison
Hardware Design
Y-Chart
Y-Table
VHDL Standard

- Abstraction is key
  - Start on high abstraction and (automatically) move inwards
  - Catch: Increasing abstraction ⇒ decreased optimization potential
- All points of view describe same circuit
  - Translate between them as beneficial
  - Harnessed by tools



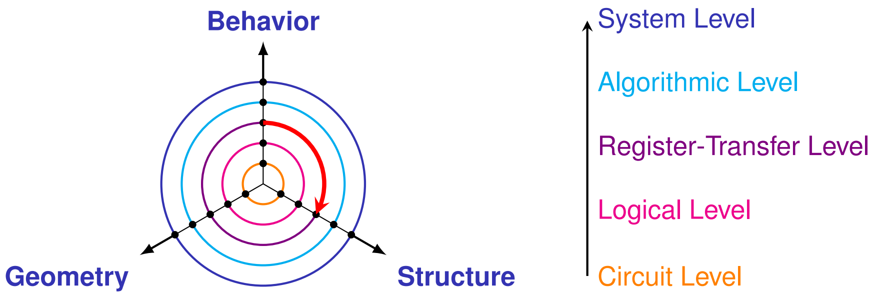


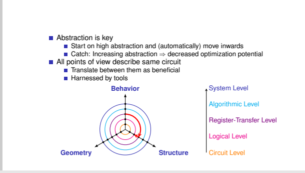
Note how the Y-chart shows that there are multiple ways to reach a certain circuit description. For example, let us consider the usual design flow for FPGAs, highlighted in red. It starts with a behavioral description at the register-transfer level and translate this into a structural description in a step called synthesis.

Gajski Y-Chart

HW Design
Motivation
SW Comparison
Hardware Design
Y-Chart
Y-Table
VHDL Standard

- Abstraction is key
  - Start on high abstraction and (automatically) move inwards
  - Catch: Increasing abstraction ⇒ decreased optimization potential
- All points of view describe same circuit
  - Translate between them as beneficial
  - Harnessed by tools



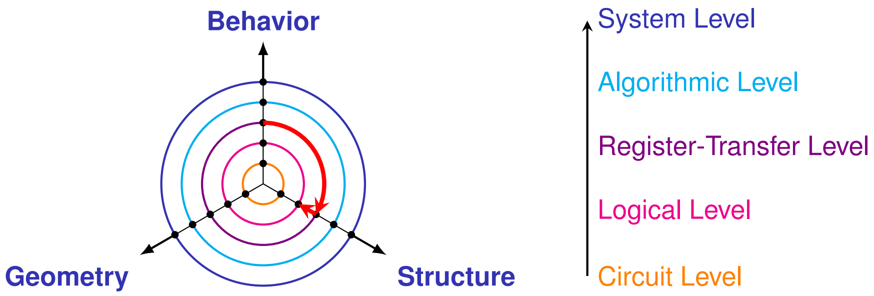


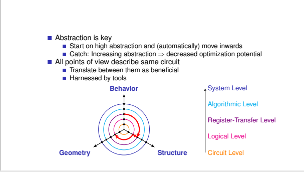
Next, details are introduced, leading to a decrease of the level of abstraction. The respective step is called technology mapping.

Gajski Y-Chart

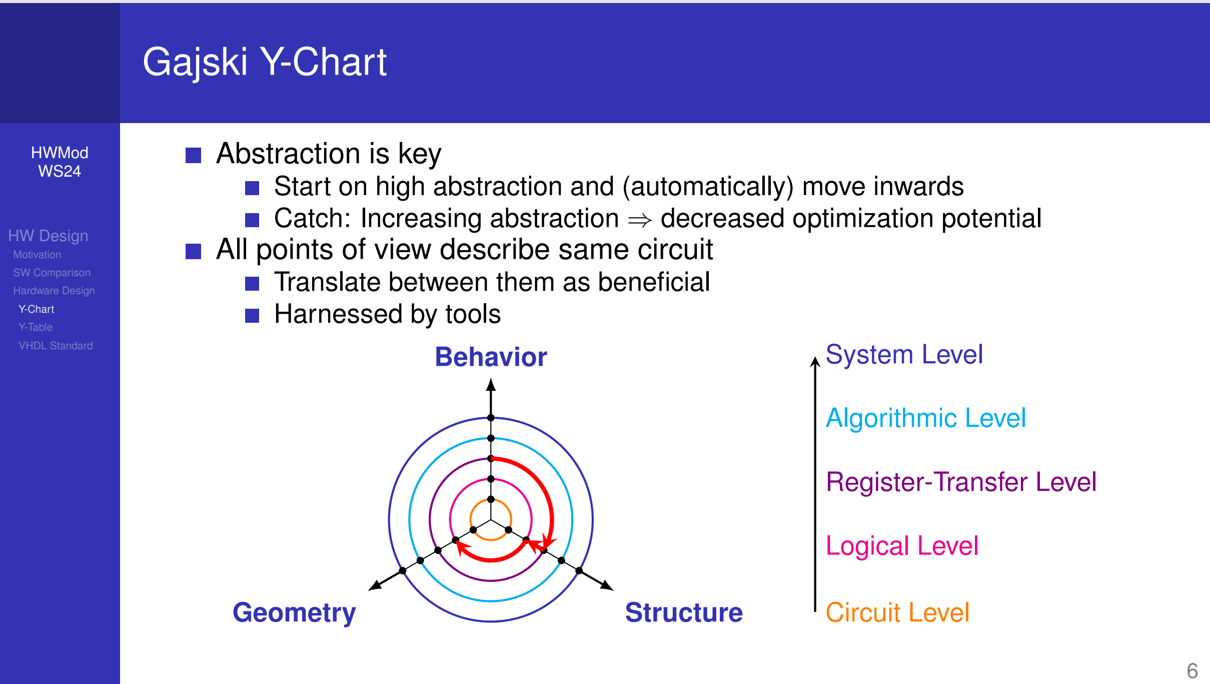
HW Design
Motivation
SW Comparison
Hardware Design
Y-Chart
Y-Table
VHDL Standard

- Abstraction is key
  - Start on high abstraction and (automatically) move inwards
  - Catch: Increasing abstraction ⇒ decreased optimization potential
- All points of view describe same circuit
  - Translate between them as beneficial
  - Harnessed by tools





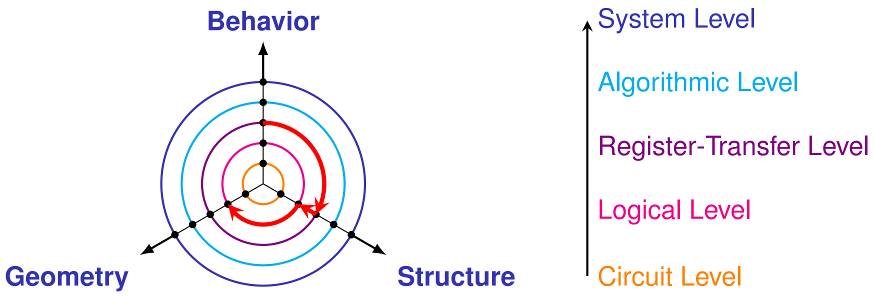
Finally, the place and route step converts the structural description into a geometric description on the logic level which we require as input for an FPGA design.

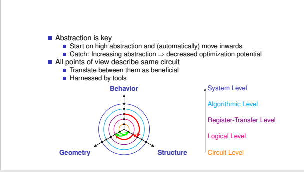


Gajski Y-Chart

HWMod WS24
HW Design
Motivation
SW Comparison
Hardware Design
Y-Chart
Y-Table
VHDL Standard

- Abstraction is key
  - Start on high abstraction and (automatically) move inwards
  - Catch: Increasing abstraction ⇒ decreased optimization potential
- All points of view describe same circuit
  - Translate between them as beneficial
  - Harnessed by tools

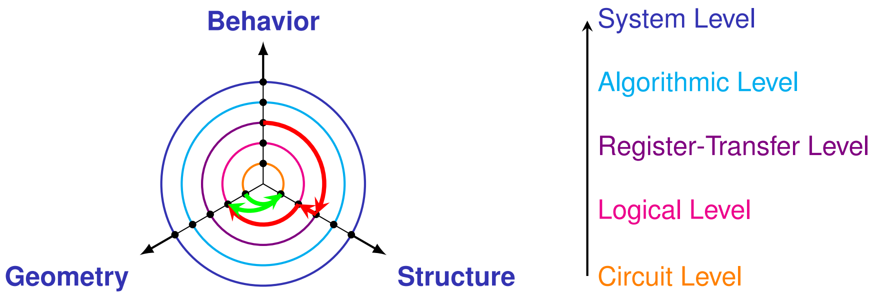


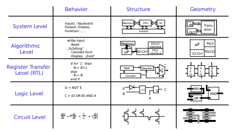


To simply the design flow, making it faster and cheaper, the steps from the circuit level to the logic level are already done by the manufacturer. We highlighted them in green.

Gajski Y-Chart

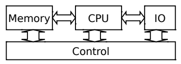
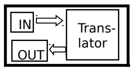
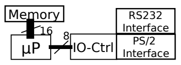
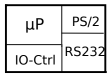
- Abstraction is key
  - Start on high abstraction and (automatically) move inwards
  - Catch: Increasing abstraction ⇒ decreased optimization potential
- All points of view describe same circuit
  - Translate between them as beneficial
  - Harnessed by tools

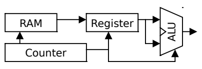
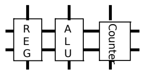


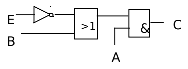
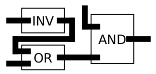


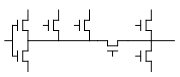
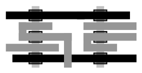
On the previous slide we have talked about the Y-chart and how we can use it to describe the hardware design flow for different target technologies. However, so far the distinct levels of abstraction and points of view were hardly tangible. Therefore, we present to you a tabular version of this chart on this slide. This table contains the same three viewing points as before as columns and the levels of abstraction as rows.

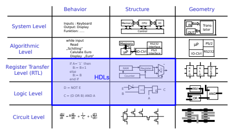
Y-Table

	Behavior	Structure	Geometry
System Level	Inputs : Keyboard Output: Display Funktion:		
Algorithmic Level	while input Read „Schilling“ Calulate Euro Display „Euro“		

Register Transfer Level (RTL)	if A=`1` then
	B:= B+1
	else
	B:= B
	end if
			

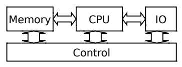
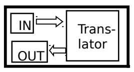
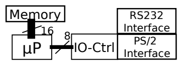
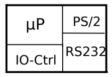
Logic Level	D = NOT E
	
	C = (D OR B) AND A
			

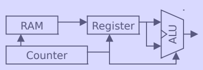
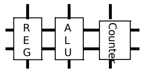
Circuit Level	$\frac{dU}{dt} = R\frac{dI}{dt} + \frac{I}{C} + L\frac{d^2I}{dt^2}$		

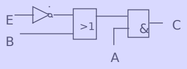
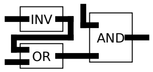


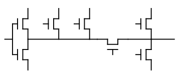
Nowadays, hardware design typically starts with a behavioral description on the register-transfer level, abbreviated as *RTL*. This programmatic description is done using hardware description languages, or *HDL* for short. Be noted though that such HDLs usually support a mixture of behavioral descriptions and structural descriptions on the RTL and logic level, as this allows for efficient designing. We have highlighted the different circuit description they contain on the slide.

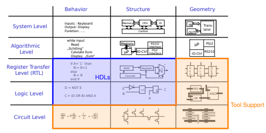
Y-Table

	Behavior	Structure	Geometry
System Level	Inputs : Keyboard Output: Display Funktion:		
Algorithmic Level	while input Read „Schilling“ Calulate Euro Display „Euro“	 Memory 16 8 µP IO-Ctrl RS232 Interface PS/2 Interface	

Register Transfer Level (RTL)	if A=`1` then
	B:= B+1
	else
	B:= B
	end if
	HDLs
			

Logic Level	D = NOT E
	
	C = (D OR B) AND A
			

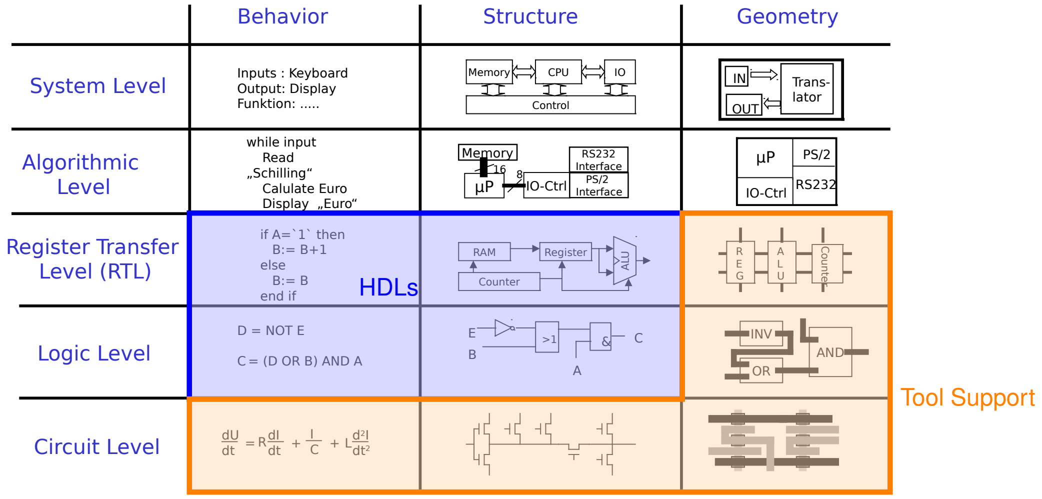
Circuit Level	$\frac{dU}{dt} = R\frac{dI}{dt} + \frac{I}{C} + L\frac{d^2I}{dt^2}$		

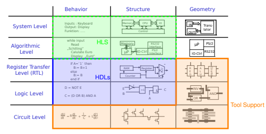


After a designer has created a circuit description in such an HDL, tools then take this description and convert it to the desired target description. This typically involves the incorporation of information about the specific target technology in order to decrease the level of abstraction.

Y-Table

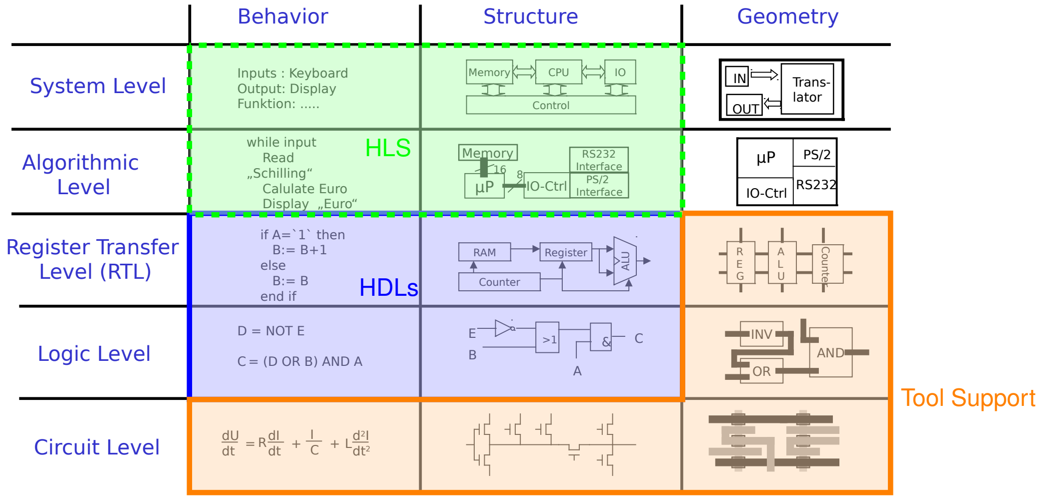
HW Design
Motivation
SW Comparison
Hardware Design
Y-Chart
Y-Table
VHDL Standard

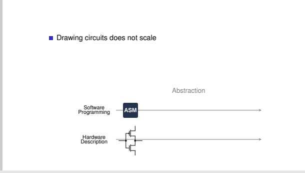




Due to the ever-increasing complexity of circuits, the RTL abstraction that served us well for the past few decades is slowly but steadily reaching its limit. As a remedy researches are currently working on so-called high-level synthesis or for short *HLS*. As the name suggests, the goal is to provide circuit descriptions on a higher behavioral or structural level like the algorithmic or system one. However, this will be the topic of other courses.

Y-Table

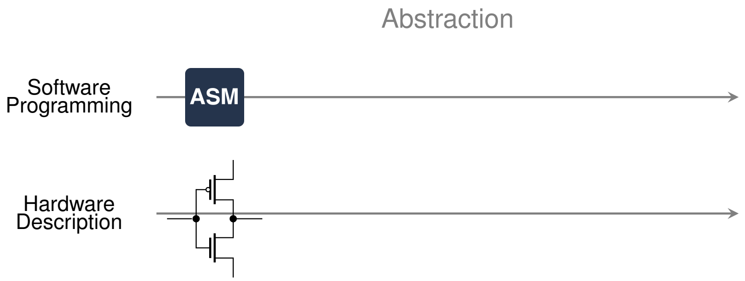


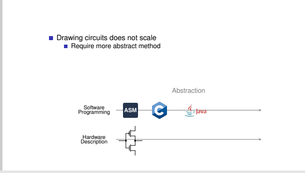


As mentioned before, hardware description languages are typically used to describe circuits. As the name already suggests, these are languages created for describing hardware. But why do we need specific languages for that? Let us again draw a comparison to software development. Ff you are writing object-oriented code, you prefer an object-oriented language like Java or C# that has the notion of objects built in. Nevertheless, you still *could* write your code also in C or even assembly. You will mostly combat the language instead of creating the desired program, impeding your efficiency and reducing the possible complexity of your programs.   When designing hardware things are essentially the same. We could simply draw circuits by hand and that's it. However, much as with writing assembly code you would be limited to rather small circuits. Just imagine drawing a modern CPU by hand.

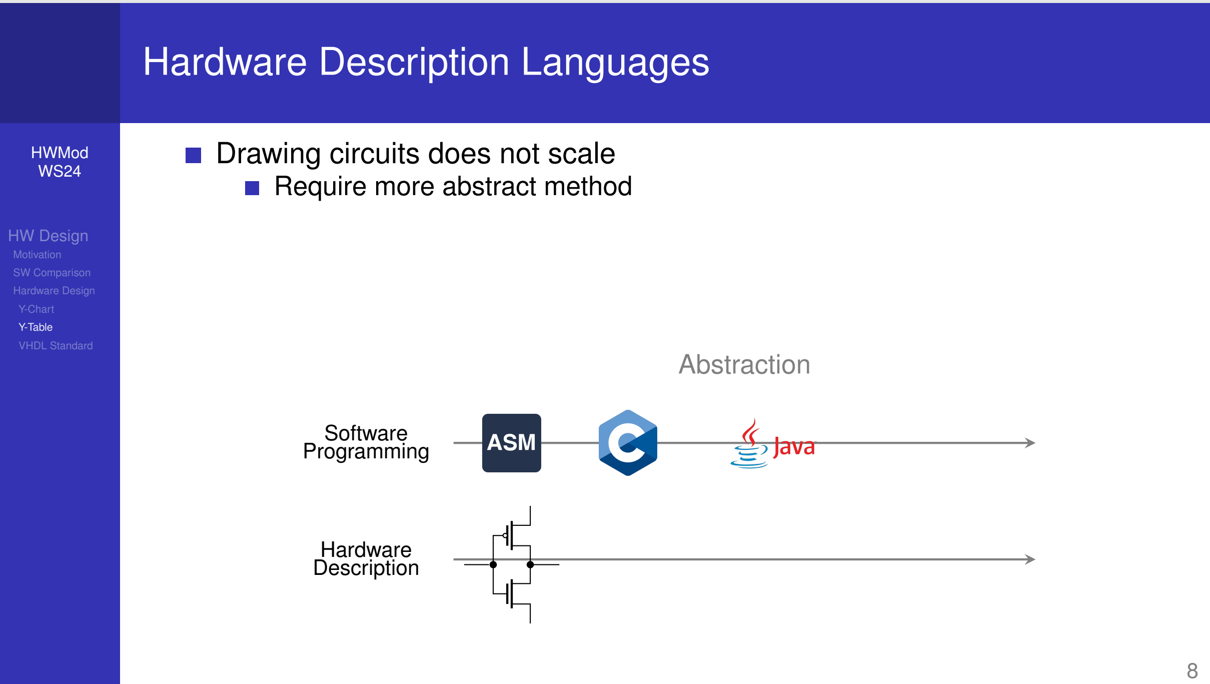
Hardware Description Languages

- Drawing circuits does not scale



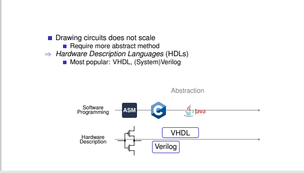


As a remedy, we need to increase the level of abstraction. Software engineers also discovered the need for more abstraction some time ago, and ended up with languages like Pascal or C. Over time, with computers becoming more powerful, more specialized and even more abstract programming languages like Java or Haskell came up.

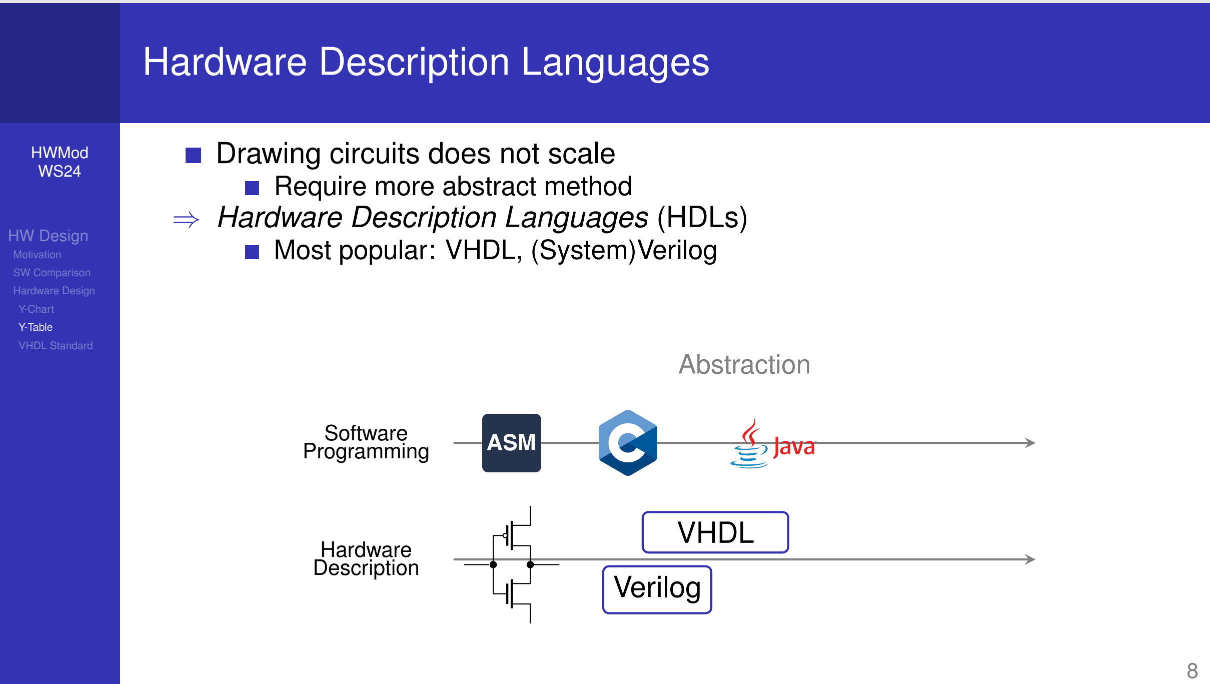


Hardware Description Languages

- Drawing circuits does not scale
  - Require more abstract method



Hardware designers had similar problems and desires and ended up in a similar spot. From drawing circuit schematics by hand, they moved to drawing them using computer assistance and then to describing them programmatically. This is also where we are currently at with hardware description languages, with VHDL and System-Verilog arguably being the most popular ones.

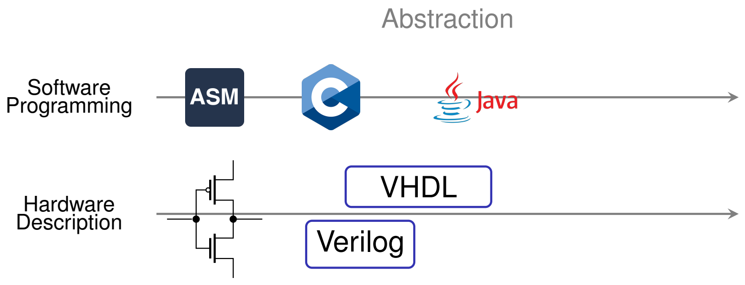


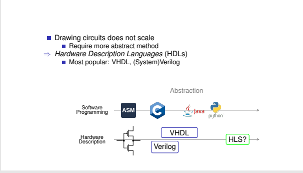
Hardware Description Languages

HW Design
Motivation
SW Comparison
Hardware Design
Y-Chart
Y-Table
VHDL Standard

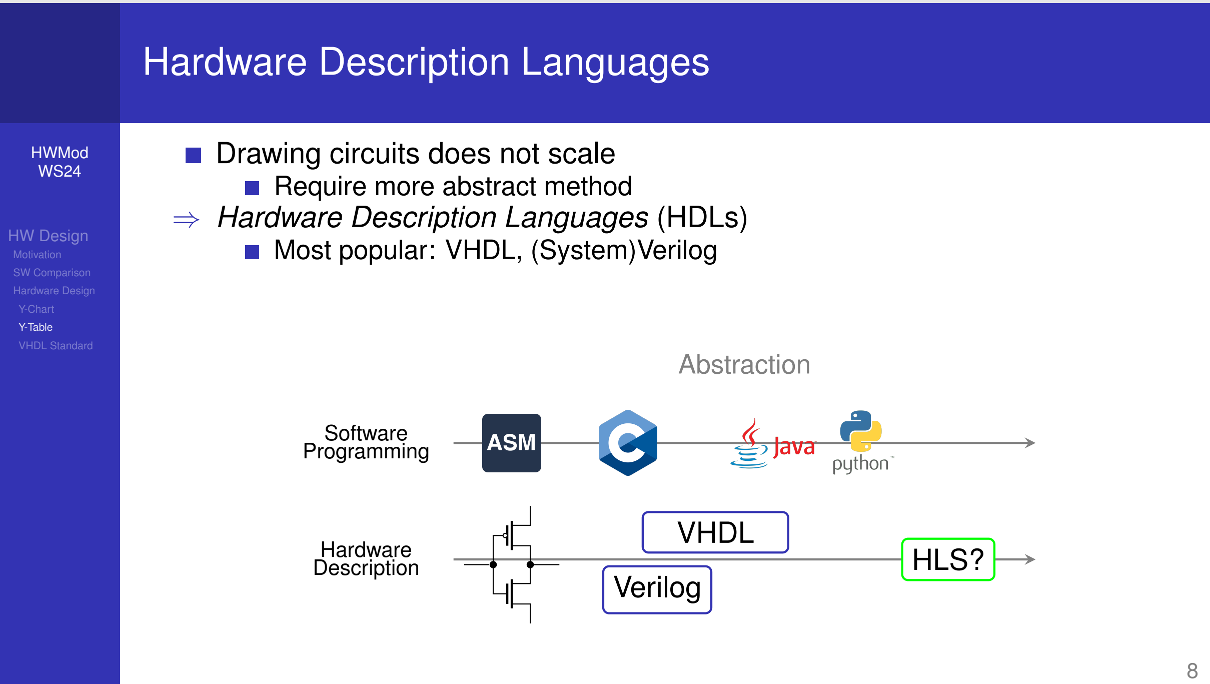
- Drawing circuits does not scale
  - Require more abstract method

⇒ *Hardware Description Languages* (HDLs)
  - Most popular: VHDL, (System)Verilog





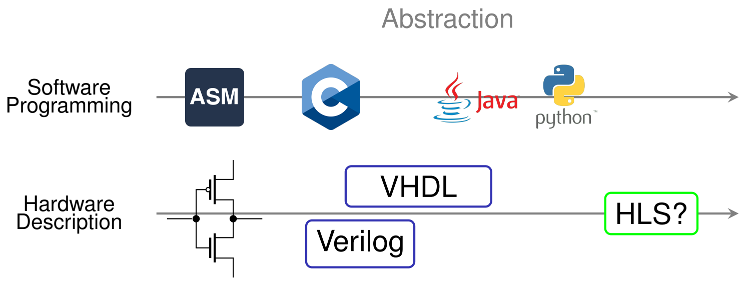
In the past few years we could observe the growing popularity of languages that are even more abstract than Java, C#, Haskell and consorts. A prominent example is certainly Python, allowing to easily harness the increasing available power of computers and top deal with the complexity of modern computing devices. Hardware designers are still working on something similar in the form of the mentioned high level synthesis.

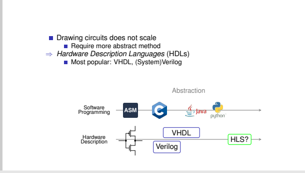


Hardware Description Languages

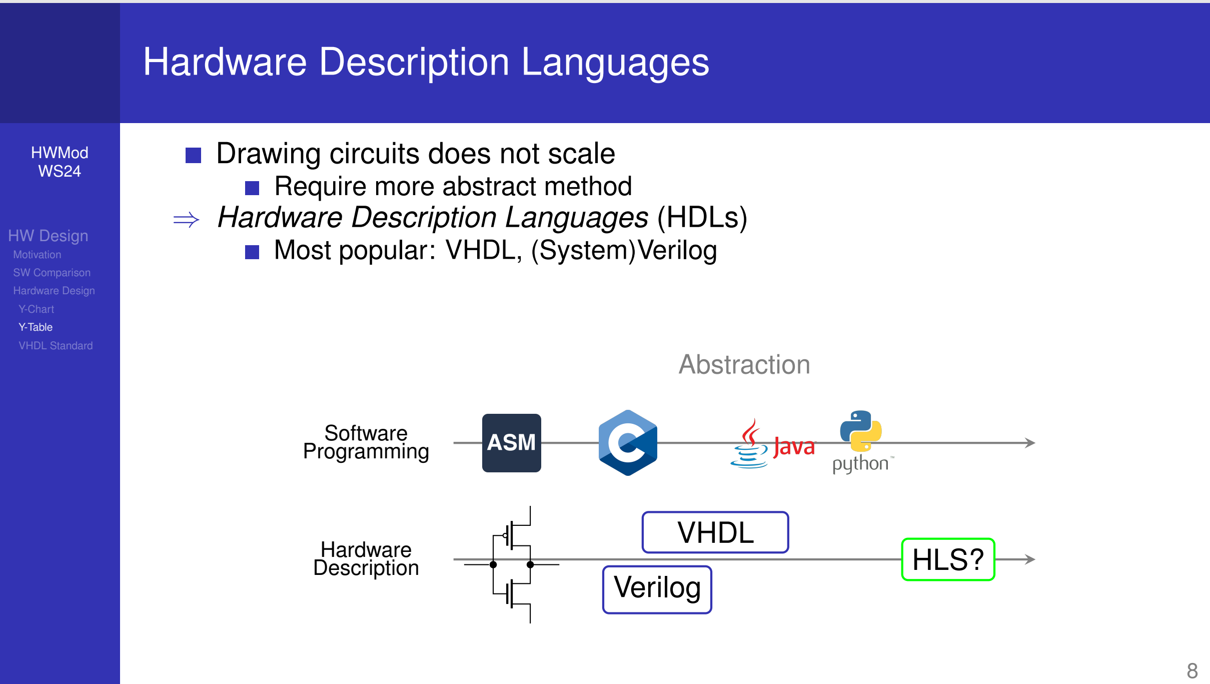
- Drawing circuits does not scale
  - Require more abstract method

⇒ *Hardware Description Languages* (HDLs)
  - Most popular: VHDL, (System)Verilog





Nevertheless, as stated before moving up on the abstraction ladder comes at a cost as the loss of information and detail prohibits intricate optimizations. Therefore, low-level programming languages and hardware description languages will likely never stop to be of use.



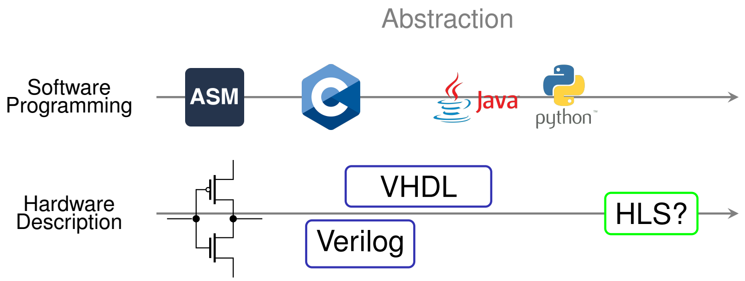
Hardware Description Languages

HWMod
WS24

HW Design
Motivation
SW Comparison
Hardware Design
Y-Chart
Y-Table
VHDL Standard

- Drawing circuits does not scale
  - Require more abstract method

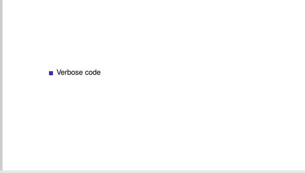
⇒ *Hardware Description Languages* (HDLs)
  - Most popular: VHDL, (System)Verilog



In this lecture we will use the hardware description language VHDL. However, as we just heard there is also Verilog as a notable alternative. So why did we choose VHDL? In general the reason is that we believe that VHDL is a great language for beginners.

We will use VHDL! But why?

HW Design
Motivation
SW Comparison
Hardware Design
Y-Chart
Y-Table
VHDL Standard



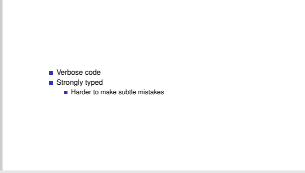
First, VHDL is very verbose. While this makes you write a bit more code than you would need in other languages, VHDL is easily comprehensible once you got the hang of it.

We will use VHDL! But why?

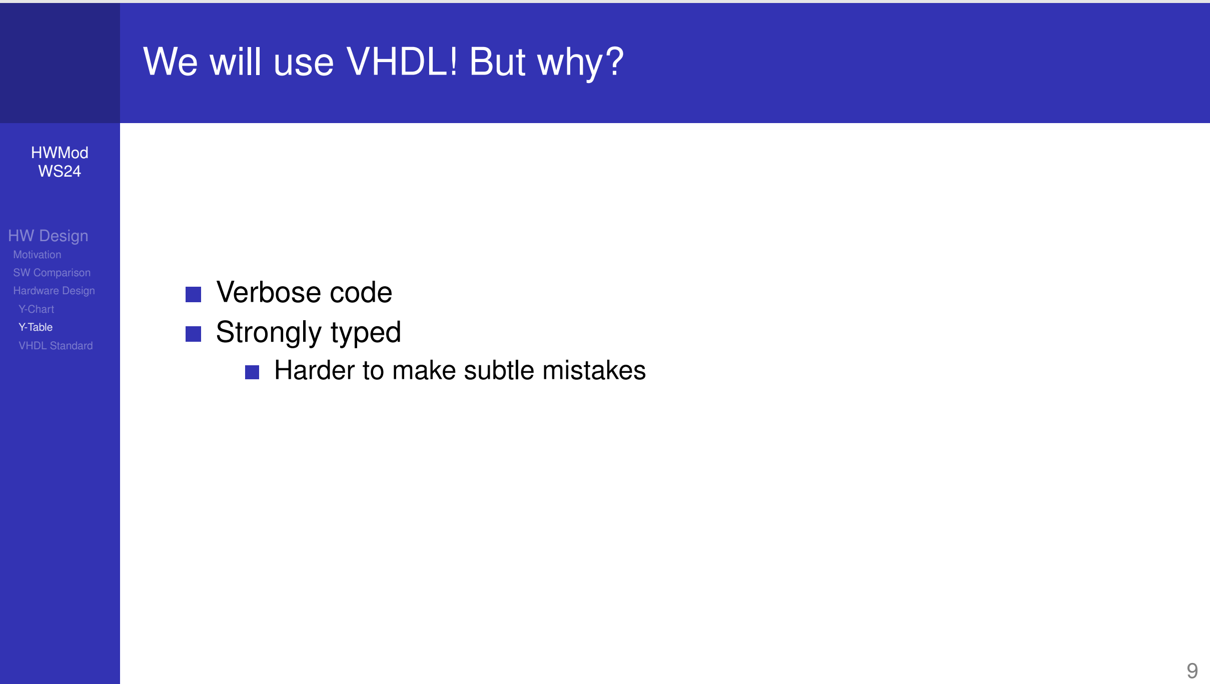
HWMod
WS24

HW Design
Motivation
SW Comparison
Hardware Design
Y-Chart
Y-Table
VHDL Standard

- Verbose code

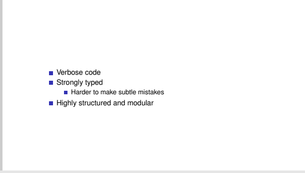


Furthermore, VHDL is a strongly typed language. Getting it to compile can sometimes be a bit more demanding than for example Verilog code, but on the other hand you get much better feedback from the tools about the issues with your code. Furthermore, this strong typing makes it harder to introduce subtle mistakes.



We will use VHDL! But why?

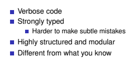
- Verbose code
- Strongly typed
  - Harder to make subtle mistakes



Due to its similarity to the Ada programming language VHDL is highly structural and modular, which also contributes to VHDL programs being easily comprehensible.

We will use VHDL! But why?

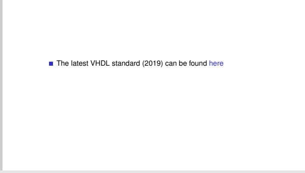
- Verbose code
- Strongly typed
  - Harder to make subtle mistakes
- Highly structured and modular


- Verbose code
- Strongly typed
  - Harder to make subtle mistakes
- Highly structured and modular
- Different from what you know

Finally, we believe VHDL is a good starting language for you in particular, as it does not resemble any programming language you have been taught in university so far. This explicitly emphasizes that hardware design is **not** just software design.

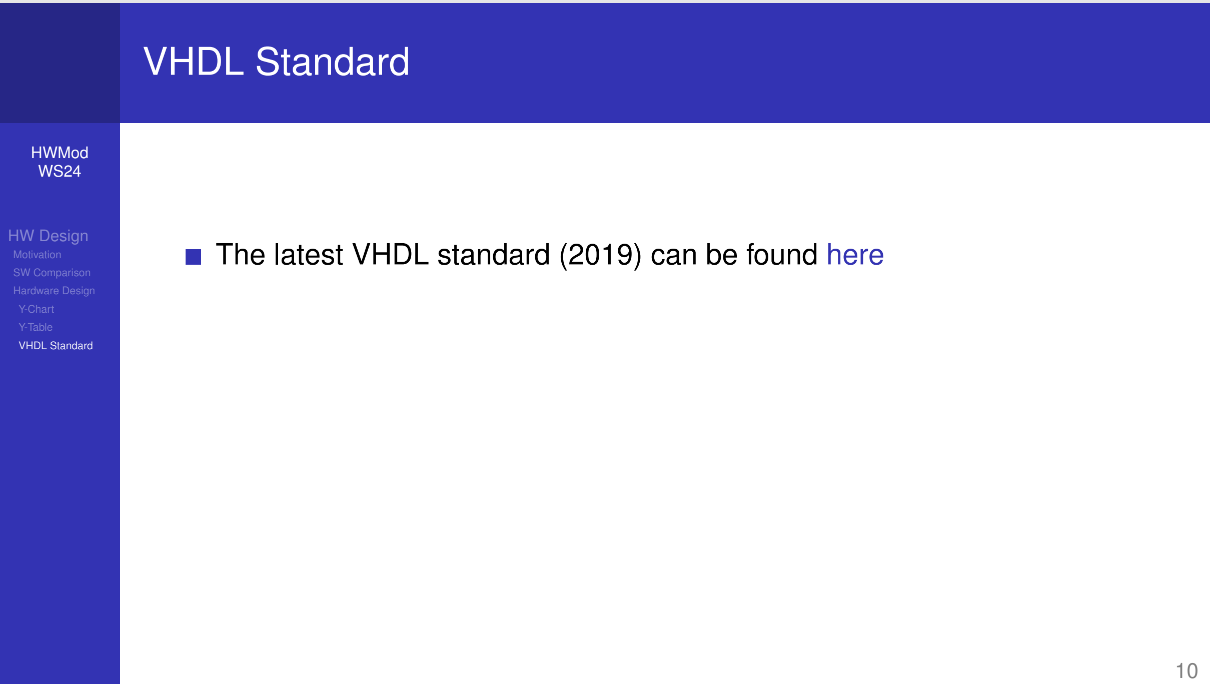
We will use VHDL! But why?

- Verbose code
- Strongly typed
  - Harder to make subtle mistakes
- Highly structured and modular
- Different from what you know



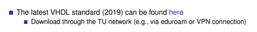
- The latest VHDL standard (2019) can be found here

Finally, let us end this lecture by pointing you towards the VHDL standard, to which we will often refer in the lectures. The standard is maintained and distributed by the IEEE. You can find it using the clickable link on the slide.

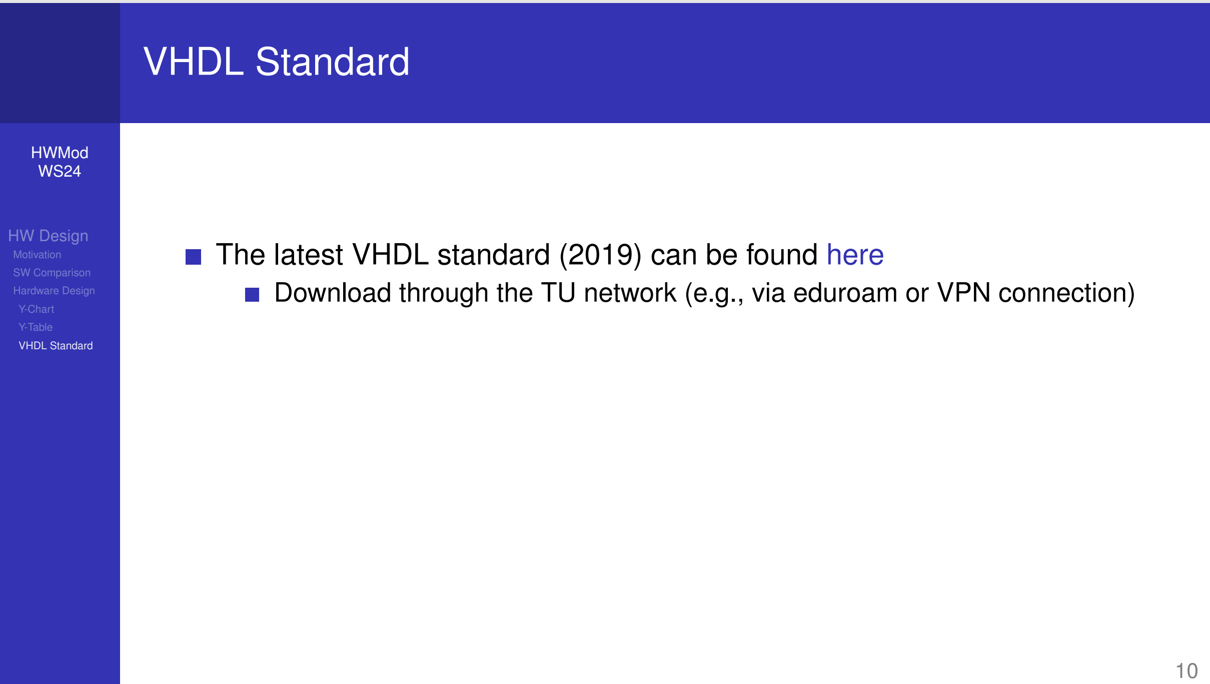


VHDL Standard

- The latest VHDL standard (2019) can be found here



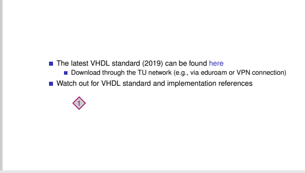
However, be aware that the standard is not openly available for everyone, but as a student of TU Vienna you can access it free of charge. To do so, you have to access the provided link from within the TU network. Hence, either connect to, for example, the *Eduroam* wireless network or use the TU VPN service.



VHDL Standard

HW Design
Motivation
SW Comparison
Hardware Design
Y-Chart
Y-Table
VHDL Standard

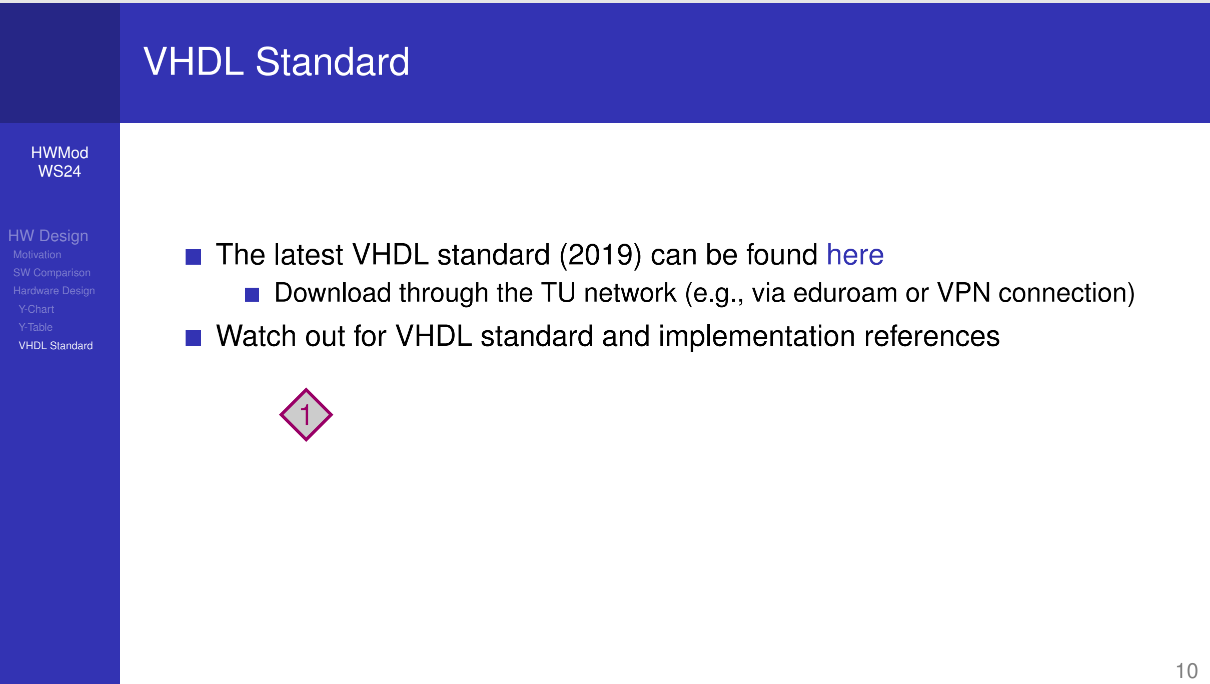
- The latest VHDL standard (2019) can be found here
  - Download through the TU network (e.g., via eduroam or VPN connection)



- The latest VHDL standard (2019) can be found here
  - Download through the TU network (e.g., via eduroam or VPN connection)
- Watch out for VHDL standard and implementation references

1

In this lecture videos, we refer to the standard in two ways. One is the diamond shape you can see on the slide. This refers to the depicted page in the VHDL standard, in this case page one is referenced. If you click this shape, and the VHDL standard is in the same directory as the slides with the exact same name as when you downloaded it, the standard will be opened at the respective page.

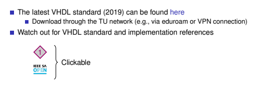


VHDL Standard

HW Design
Motivation
SW Comparison
Hardware Design
Y-Chart
Y-Table
VHDL Standard

- The latest VHDL standard (2019) can be found here
  - Download through the TU network (e.g., via eduroam or VPN connection)
- Watch out for VHDL standard and implementation references

1


- The latest VHDL standard (2019) can be found here
  - Download through the TU network (e.g., via eduroam or VPN connection)
- Watch out for VHDL standard and implementation references

1 IEEE SA OPEN } Clickable

The other type of reference is to the openly available implementation of the standard libraries by the IEEE. Clicking the respective logo will open up the referenced part of the implementation in your browser.

VHDL Standard

- The latest VHDL standard (2019) can be found here
  - Download through the TU network (e.g., via eduroam or VPN connection)
- Watch out for VHDL standard and implementation references

1
IEEE SA OPEN
} Clickable

Lecture Complete!

Thank you for listening! We recommend you to immediately take the self-check test in TUWEL, to see if you understood the material presented in this lecture.

HW Design
Motivation
SW Comparison
Hardware Design
Y-Chart
Y-Table
VHDL Standard

Lecture Complete!