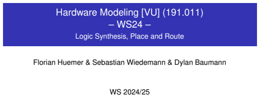


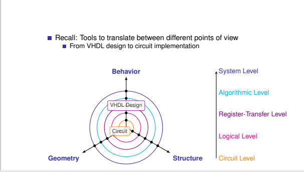
In this lecture we will discuss how VHDL designs become actual hardware implementations of the circuits they model. We will discuss logic synthesis, placement and routing of VHDL code to circuits. We will briefly talk about the design flow in general, and the compilation, technology mapping, placement and routing steps in particular and illustrate them at the hand of an example. We will also introduce FPGAs as a target technology for our designs.

Hardware Modeling [VU] (191.011) – WS24 – Logic Synthesis, Place and Route

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

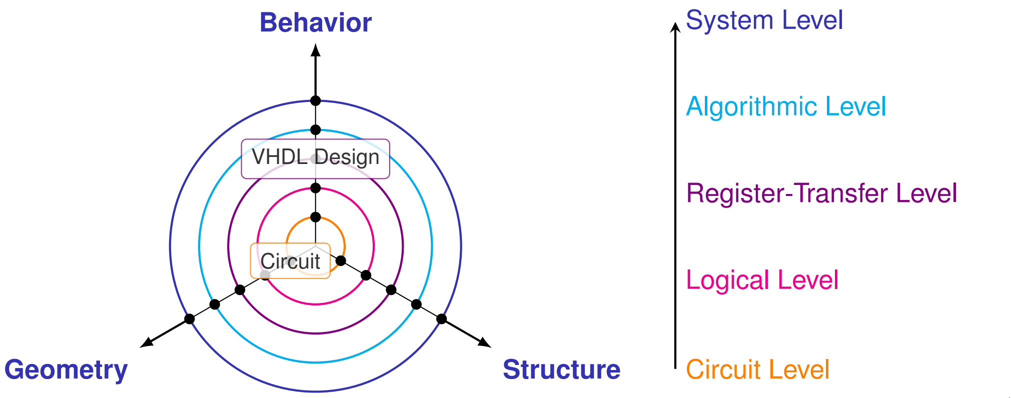
Modified: 2025-03-12, 16:32 (21636bb)

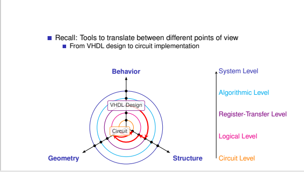


Recall the Y-diagram of previous lectures. We introduced it as a useful tool for capturing and illustrating overall flow from a model of a circuit to a proper implementation. In essence, the Y-diagram expresses that a model must be equipped with an increasing amount of detail in order to implement it. However, we also learned that there might exist multiple paths when descending from the model's level of abstraction to that of the final circuit.

Recall: Y-Diagram

- Recall: Tools to translate between different points of view
  - From VHDL design to circuit implementation

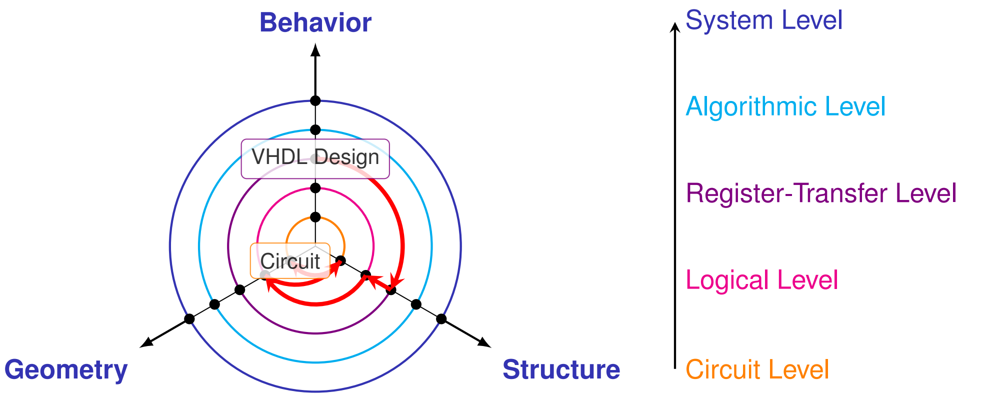


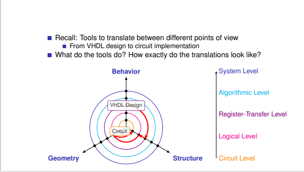


On the slide we can see an example path, drawn in red, that translates between the different points of view while descending on the level of abstraction. While we already saw this in a previous lecture, we only mentioned that these translations are done by tools, harnessing the advantages of different viewpoints at the various levels of abstraction. We have not yet talked about these tools, which do much of the heavy lifting involved in creating complex circuits, though.

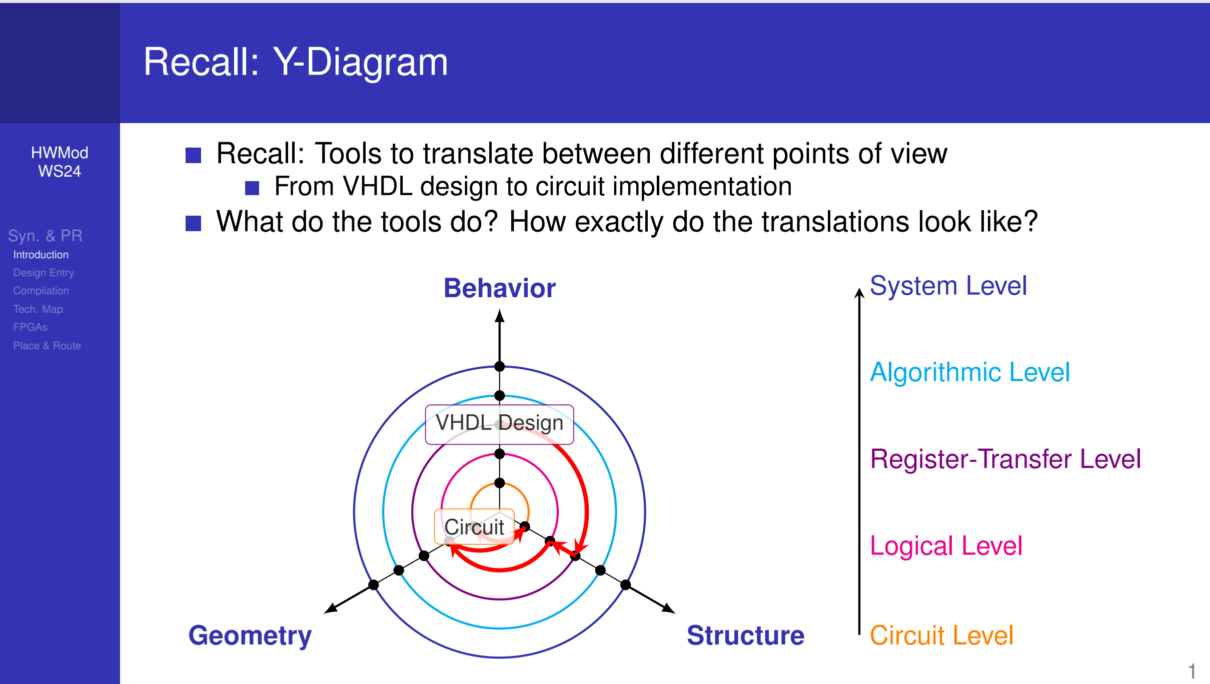
Recall: Y-Diagram

- Recall: Tools to translate between different points of view
  - From VHDL design to circuit implementation





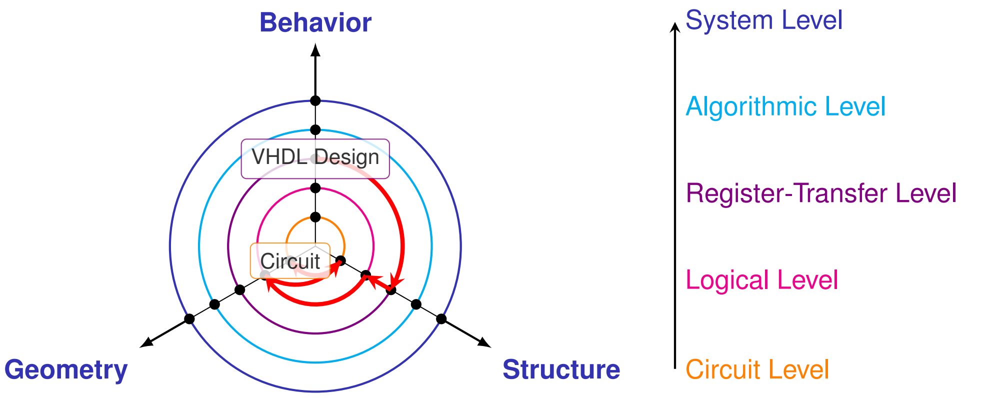
However, just as good software developers should be aware of the steps involved in compilation or interpretation of their programs, good hardware designers should be aware of the tasks handled by the tools that convert their models into hardware. The overarching goal of this lecture will be to discuss this steps briefly. First, let us have a first look at the specific steps involved in the hardware design flow before we discuss each one in a bit more detail.

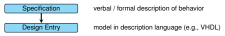


Recall: Y-Diagram

HWMod
WS24

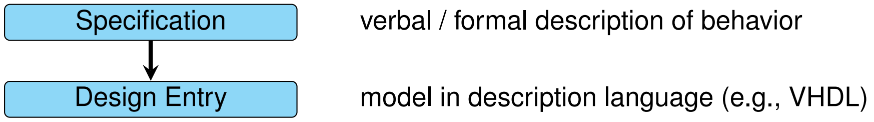
- Recall: Tools to translate between different points of view
  - From VHDL design to circuit implementation
- What do the tools do? How exactly do the translations look like?

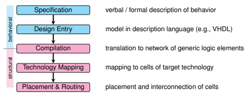




Each hardware design starts with a specification. This specification should clearly define the desired behavior, interface and environment of a design. It is typically plain text or a formal description like a timing diagram of the desired behavior. Note that you already used such specifications yourselves during this course. After all, the homework task descriptions are mostly nothing else but a specification of a desired design. This description is the major reference during the development and validation of the target circuit. As the specification should be easily understood by all parties involved in the overall design process, the first task for a designer is to convert the specification into a design entry. The design entry is a description of the design in a hardware description language, based on the specification. In our course a design entry is always VHDL code describing the desired hardware.

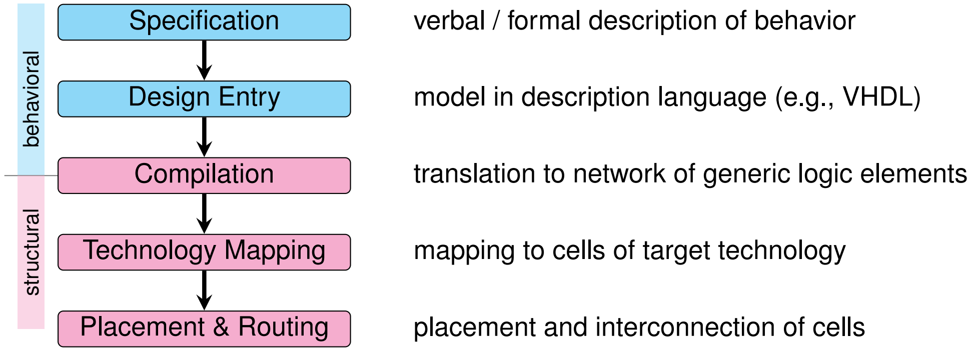
Hardware Design Flow

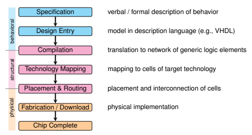

Specification
Design Entry
verbal / formal description of behavior
model in description language (e.g., VHDL)



Given a design entry, meaning some VHDL code, the next step is to convert it to data structures that can be used by tools in subsequent steps. This is the task of the compilation step. Among others, the code is checked for syntactical validity, type safety and the violation of any interfaces. The result of the compilation is a network of generic logic elements, meaning we can view the compilation as translation from a behavioral to a structural description. After the compilation, the technology mapping step takes place. In this step the network produced by the compilation step is mapped to a target technology. Finally, the process of creating an implementation description finishes with the placement and routing step. Here the logic cells are placed in physical space and connected to another.

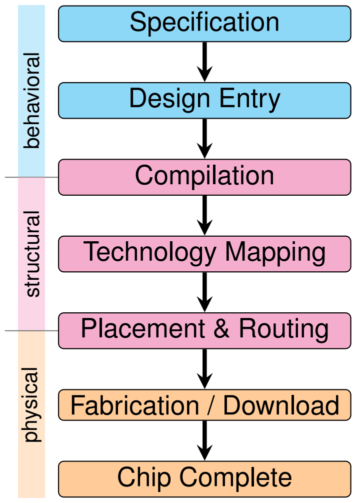
Hardware Design Flow


behavioral
structural
Specification
Design Entry
Compilation
Technology Mapping
Placement & Routing
verbal / formal description of behavior
model in description language (e.g., VHDL)
translation to network of generic logic elements
mapping to cells of target technology
placement and interconnection of cells



After the previous steps the intermediate structural description was converted to a physical one. This description is sufficiently detailed to either manufacture a chip from scratch, or to download it to a generic hardware platform. We will discuss this in more detail later. The final result of this design flow is a chip implementing the initial specification. Finally, it should be noted that, depending on the target technology, the distinct steps can become quite elaborate themselves, consisting of multiple steps and iterations.

Hardware Design Flow



verbal / formal description of behavior

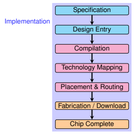
model in description language (e.g., VHDL)

translation to network of generic logic elements

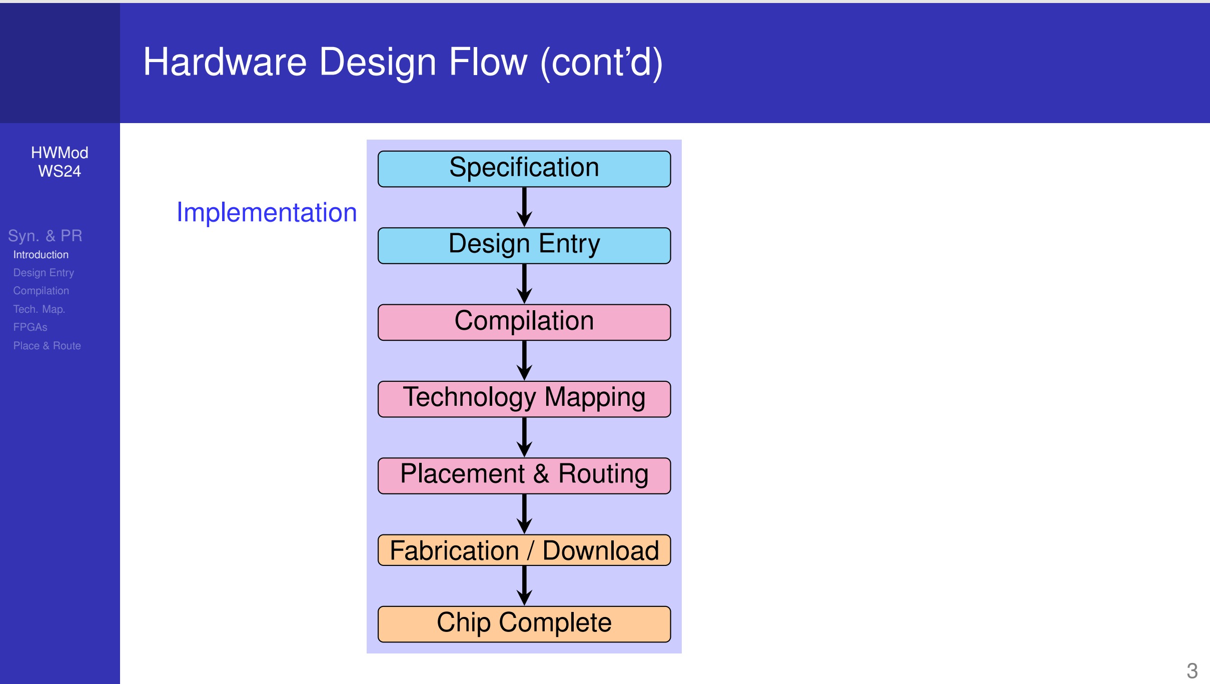
mapping to cells of target technology

placement and interconnection of cells

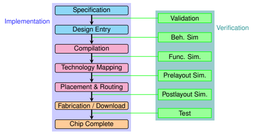
physical implementation



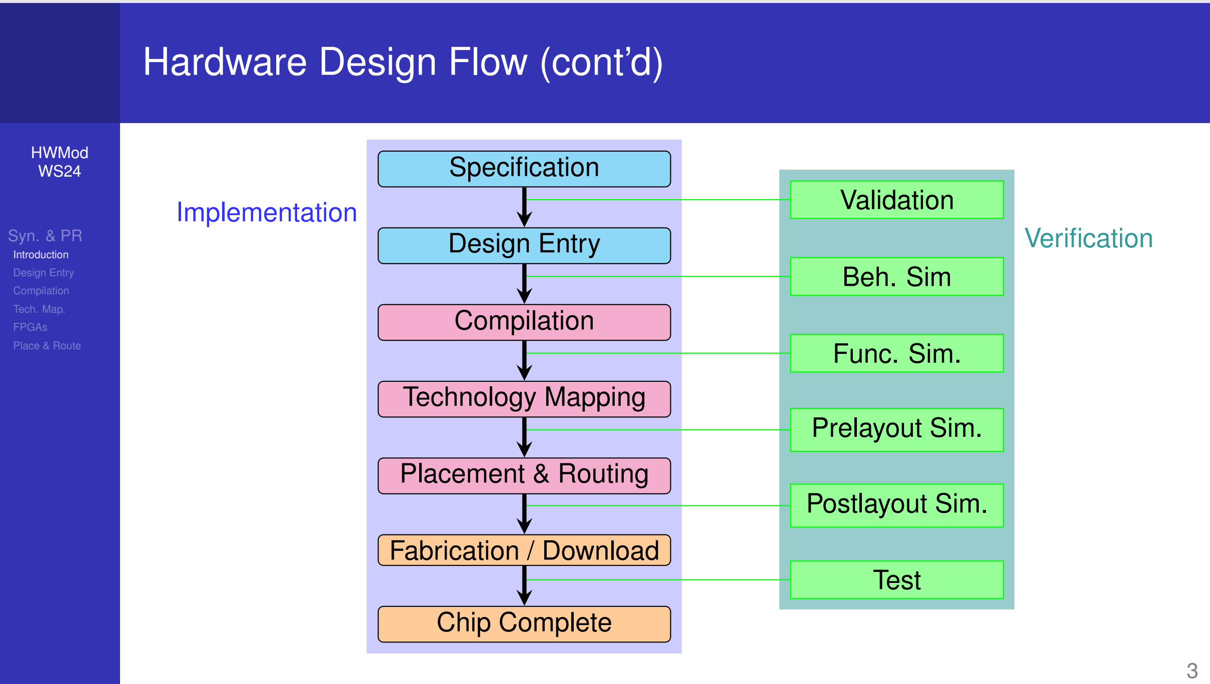
The hardware design flow discussed so far captures all the steps involved in implementing a circuit. However, this is an incomplete view. As we learned in a previous lecture, hardware development is driven by the first-time-right paradigm, where designs have to be thoroughly validated to decrease the probability of severe bugs when deploying a circuit. For this reason it is important to perform verification after each implementation step.



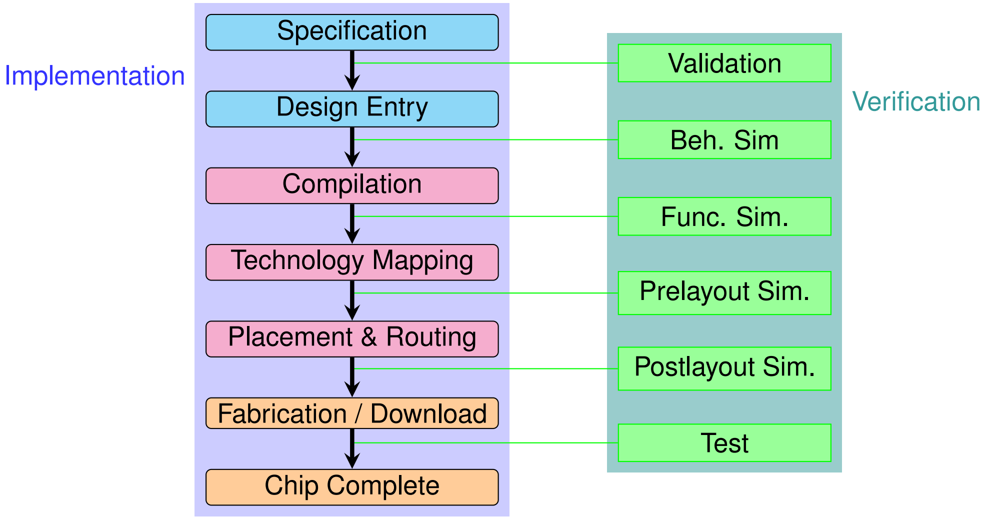
Hardware Design Flow (cont'd)

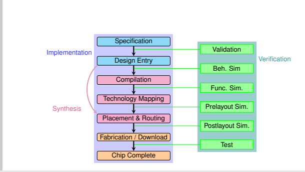


You can find the respective verification step between two implementation steps on the right of the slide. While all steps are paramount in practice, we will restrict ourselves to behavioral, and the occasional postlayout, simulations during this course. Note that the simulations that you performed so far are behavioral simulations.



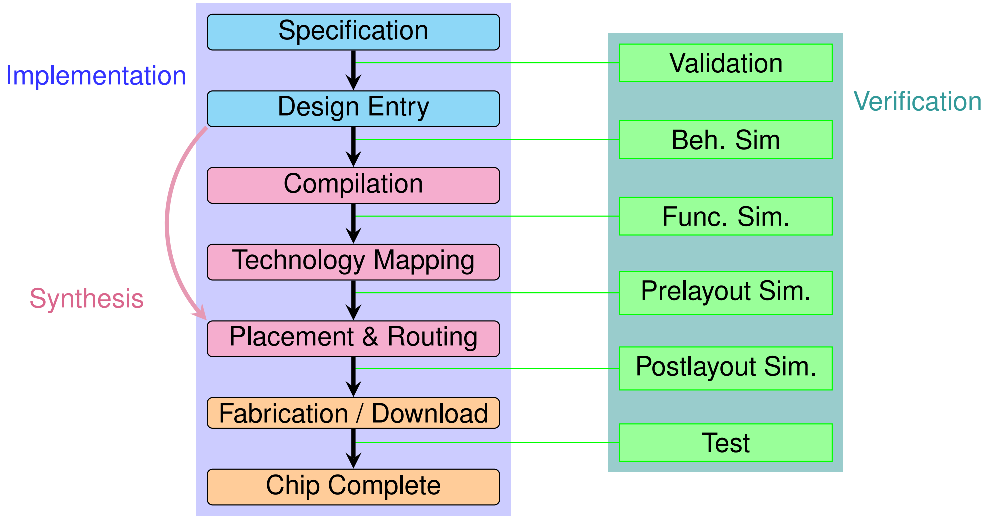
Hardware Design Flow (cont'd)

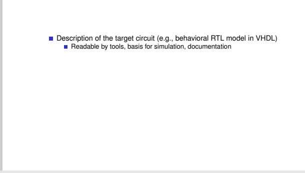




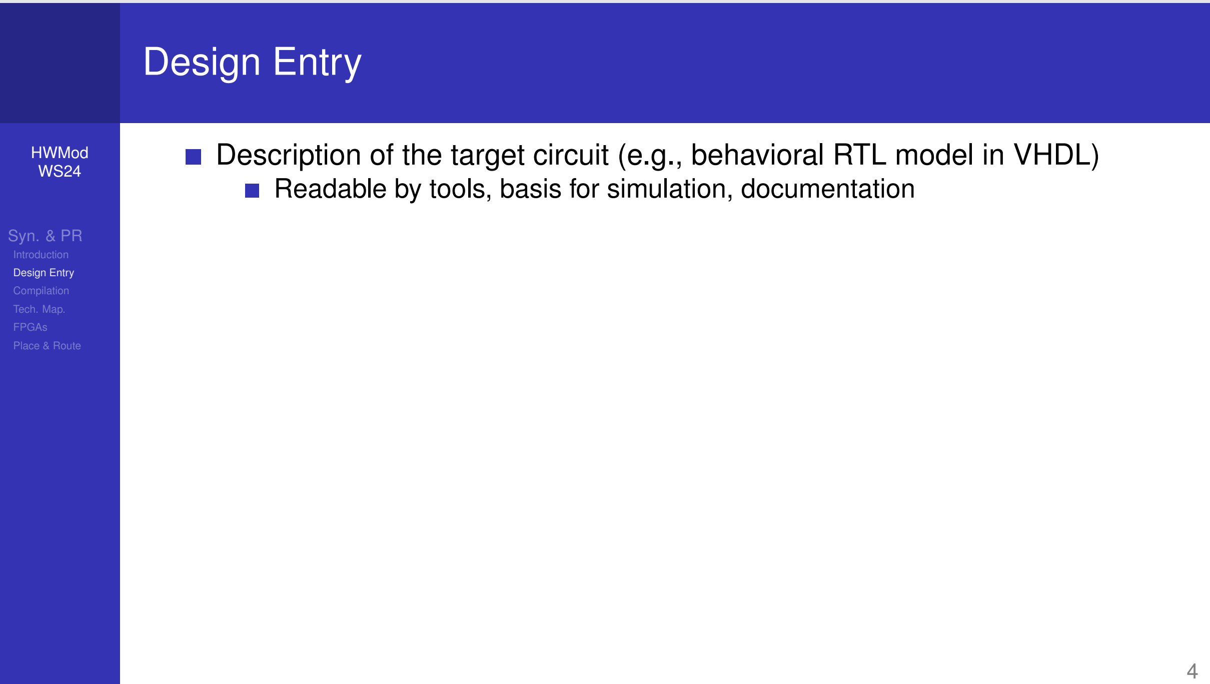
However, as verification, and the steps involved, is a deep topic on its own, there will be a dedicated lecture on it at a later point in this course. In this lecture we will focus on the synthesis, involving the compilation and technology mapping steps, and the placement and routing step. Let us now continue by discussing these steps in more detail.

Hardware Design Flow (cont'd)



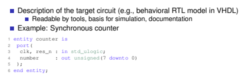

- Description of the target circuit (e.g., behavioral RTL model in VHDL)
  - Readable by tools, basis for simulation, documentation

As already mentioned, the design entry is a description of the target circuit in a hardware description language like VHDL. This makes it a formal model of the target circuit. Often, especially for complex designs, this will be a behavioral RTL model. While a specification might also be a formal model, a major difference between it and the design entry is that the design entry is of a form that is easy to read for tools. This makes it a suitable basis for simulations and further implementation steps. Furthermore, as the name suggests, a model in a description language also serves as documentation of the desired behavior. If you recall the introduction to VHDL, this was initially actually the very purpose of hardware description languages.


Design Entry

HWMod
WS24

- Description of the target circuit (e.g., behavioral RTL model in VHDL)
  - Readable by tools, basis for simulation, documentation


- Description of the target circuit (e.g., behavioral RTL model in VHDL)
  - Readable by tools, basis for simulation, documentation
- Example: Synchronous counter

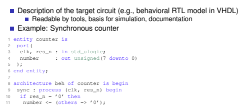
```vhdl
entity counter is
 port(
  clk, res_n : in std_ulogic;
  number     : out unsigned(7 downto 0)
 );
end entity;
```

Although you already saw some design entries, such as the various full adder VHDL descriptions, let us consider another example which we will use throughout this lecture to illustrate the implementation steps. Our example will be a synchronous eight bit counter. The counter shall have an active-low reset, a clock input and an output for the counter value.

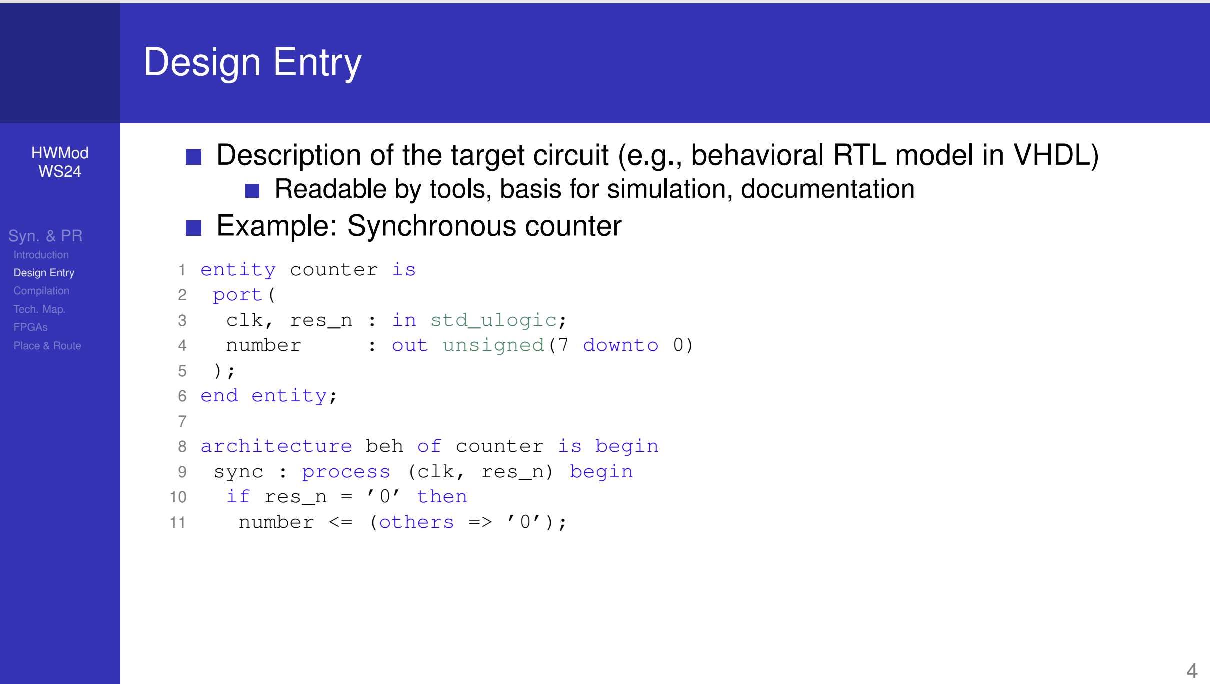
Design Entry

- Description of the target circuit (e.g., behavioral RTL model in VHDL)
  - Readable by tools, basis for simulation, documentation
- Example: Synchronous counter

```vhdl
entity counter is
 port(
  clk, res_n : in std_ulogic;
  number     : out unsigned(7 downto 0)
 );
end entity;
```


Description of the target circuit (e.g., behavioral RTL model in VHDL)
Readable by tools, basis for simulation, documentation
Example: Synchronous counter

Recall that since we want to model a synchronous circuit with asynchronous reset, our sensitivity list only contains the clock and the reset. The counter is initially reset to the value 0 using an asynchronous reset. Note that it is checked if the reset is low, because we want an active-low reset.

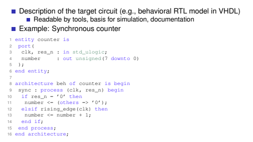


Design Entry

- Description of the target circuit (e.g., behavioral RTL model in VHDL)
  - Readable by tools, basis for simulation, documentation
- Example: Synchronous counter

```vhdl
entity counter is
 port(
  clk, res_n : in std_ulogic;
  number     : out unsigned(7 downto 0)
 );
end entity;

architecture beh of counter is begin
 sync : process (clk, res_n) begin
  if res_n = '0' then
   number <= (others => '0');
```



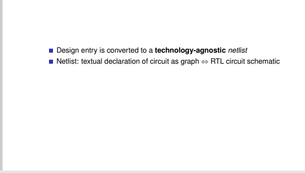
At each rising edge of the clock input the counter value shall increase by one. Can you recall what happens when the counter reaches its maximum value, meaning all eight bits are set to one, and it is incremented further? You might want to pause the video and try to recall. As the type of the counter is unsigned, it will simply wrap-around on an overflow.

Design Entry

- Description of the target circuit (e.g., behavioral RTL model in VHDL)
  - Readable by tools, basis for simulation, documentation
- Example: Synchronous counter

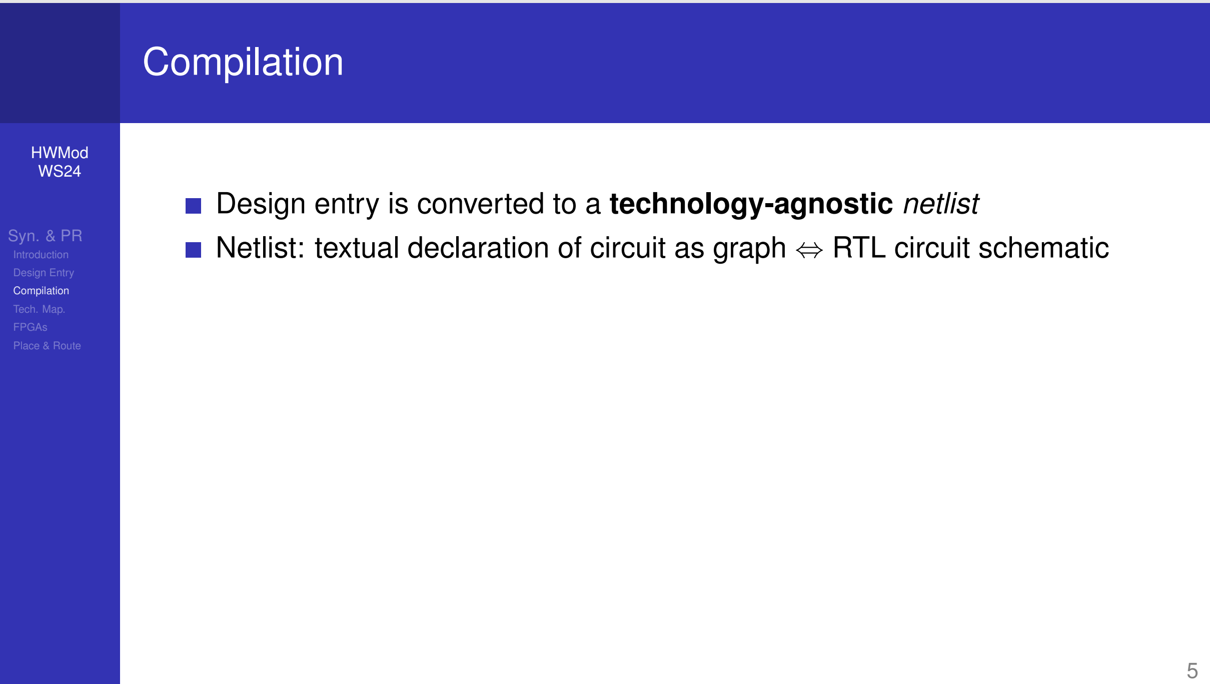
```vhdl
entity counter is
 port(
  clk, res_n : in std_ulogic;
  number     : out unsigned(7 downto 0)
 );
end entity;

architecture beh of counter is begin
 sync : process (clk, res_n) begin
  if res_n = '0' then
   number <= (others => '0');
  elsif rising_edge(clk) then
   number <= number + 1;
  end if;
 end process;
end architecture;
```



- Design entry is converted to a **technology-agnostic** *netlist*
- Netlist: textual declaration of circuit as graph ⇔ RTL circuit schematic

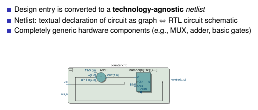
Next, after we have a design entry, the compilation step takes place. As already mentioned, this step converts the design entry into a network of generic logic elements. Okay, but what does this mean? Let us first discuss the network part. With circuits being just a set of components connected via wires, they can easily be described as graphs. The circuit components become nodes and the wires edges. The compilation step will result in a data structure containing a graph corresponding to the circuit modeled by our design entry. This data structure is typically called netlist.



Compilation

HWMod
WS24

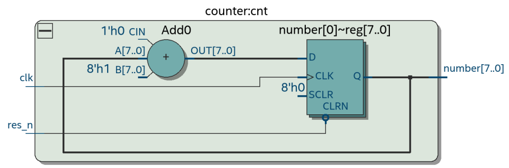
- Design entry is converted to a **technology-agnostic** *netlist*
- Netlist: textual declaration of circuit as graph ⇔ RTL circuit schematic

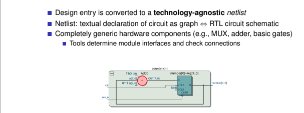


On the slide you can see the resulting network when compiling our counter example. Not too surprising, it consists of an eight bit adder and eight flip-flops with feedback to the adder. What about the second part of the compilation step, the "generic" logic elements? Well, in a nutshell, the compilation simply converts a design entry, which is on the behavioral axis of the Y-diagram, to a structural description. It is completely oblivious to any specific technology and uses basic circuit elements that do not depend on a specific technology as well. Examples for such generic circuit elements are multiplexers, basic gates but also adders and flip-flops. These elements are so fundamental that it is safe to assume that any target technology must be able to create them by some means.

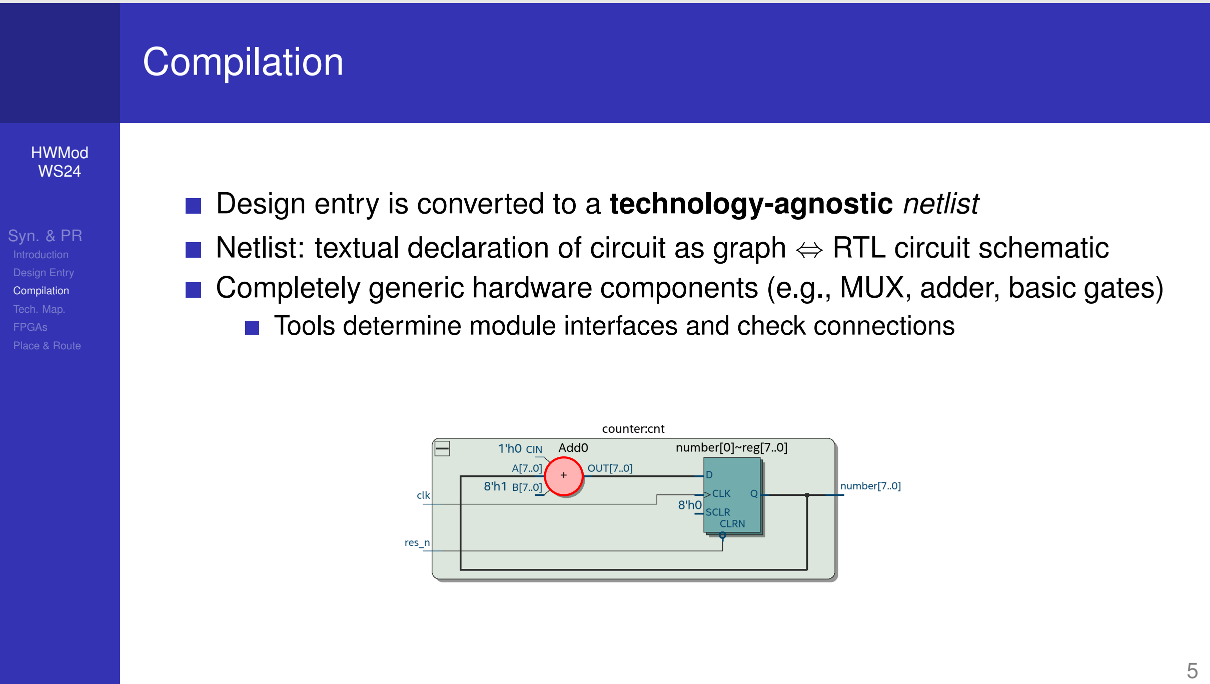
Compilation

- Design entry is converted to a **technology-agnostic** *netlist*
- Netlist: textual declaration of circuit as graph ⇔ RTL circuit schematic
- Completely generic hardware components (e.g., MUX, adder, basic gates)



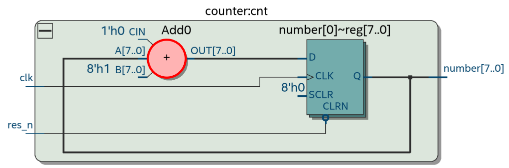


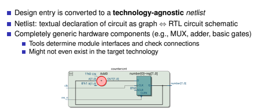
For example, the adder in our example is simply shown as generic addition block. Naturally, this is not a valid description of an adder circuit. However, the tool knows how the interface of an adder looks like and that the target technology will be able to provide a circuit that can perform addition using this interface. A more elaborate description is not necessary at this step.



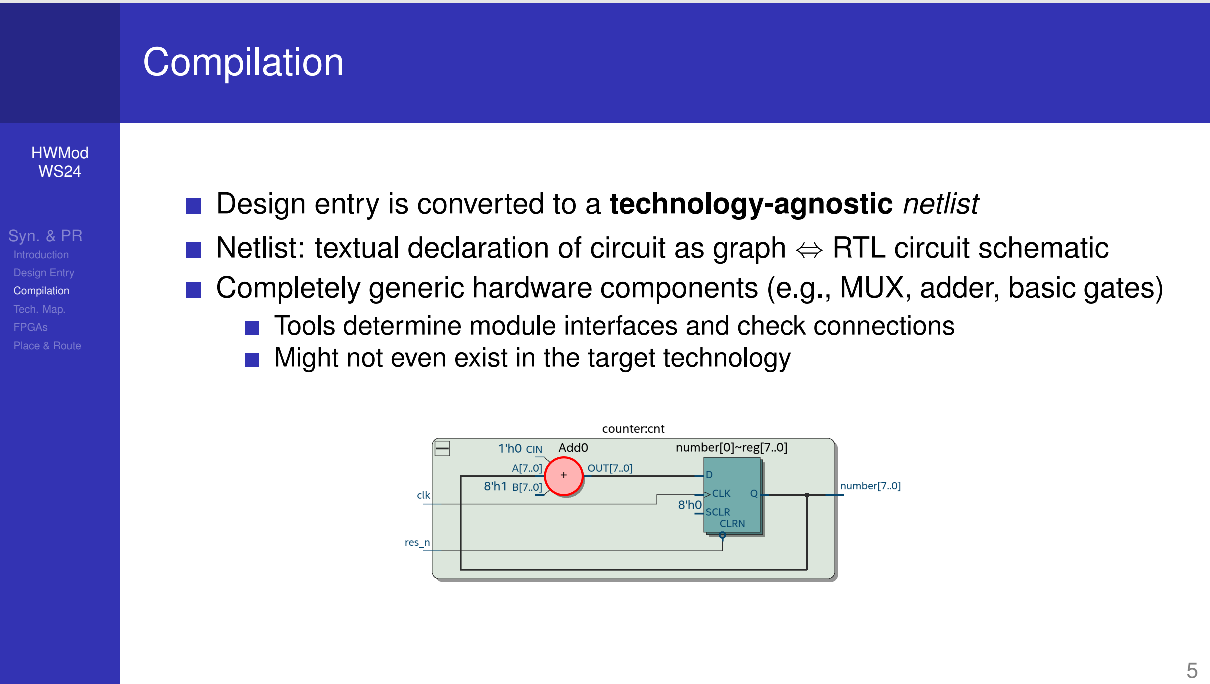
Compilation

- Design entry is converted to a **technology-agnostic** *netlist*
- Netlist: textual declaration of circuit as graph ⇔ RTL circuit schematic
- Completely generic hardware components (e.g., MUX, adder, basic gates)
  - Tools determine module interfaces and check connections



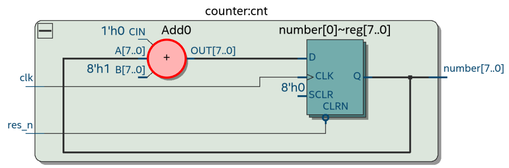


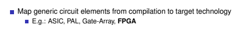
In particular, even rather basic elements like the particular flip-flops used during the compilation might not even exist in the target technology. For example, it could be that no flip-flops with asynchronous reset are available, although the model requires this functionality. However, dealing with issues like this is the job of the technology mapping step.



Compilation

- Design entry is converted to a **technology-agnostic** *netlist*
- Netlist: textual declaration of circuit as graph ⇔ RTL circuit schematic
- Completely generic hardware components (e.g., MUX, adder, basic gates)
  - Tools determine module interfaces and check connections
  - Might not even exist in the target technology

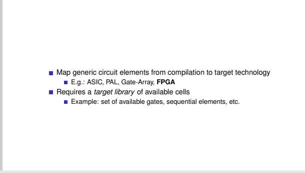



- Map generic circuit elements from compilation to target technology
  - E.g.: ASIC, PAL, Gate-Array, **FPGA**

The next step after the compilation is the technology mapping. Here the generic circuit elements used by the compilation are mapped to the target technology. Such target technologies can, for example, be application-specific-integrated-circuits, programmable-array-logic, gate-arrays or FPGAs. Do not worry if you did not here about these technologies before, they will be covered in another course. In this course we will only consider FPGAs, which we will therefore introduce in this lecture.

Technology Mapping

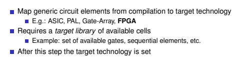
- Map generic circuit elements from compilation to target technology
  - E.g.: ASIC, PAL, Gate-Array, **FPGA**


■ Map generic circuit elements from compilation to target technology
■ E.g.: ASIC, PAL, Gate-Array, **FPGA**
■ Requires a *target library* of available cells
■ Example: set of available gates, sequential elements, etc.

The way this mapping works is that each technology comes with a library of available cells that can be used. Such a library can, for example, consist of a set of gates with which arbitrary logic functions can be implemented, but also of sequential elements like flip-flops. However, also more specialized or less frequently used elements like particular memories or latches can be part of the library. The tool responsible for mapping the generic elements to the target technology then basically tries to implement each generic logic element of the compilation using the available ones.

Technology Mapping

- Map generic circuit elements from compilation to target technology
  - E.g.: ASIC, PAL, Gate-Array, **FPGA**
- Requires a *target library* of available cells
  - Example: set of available gates, sequential elements, etc.

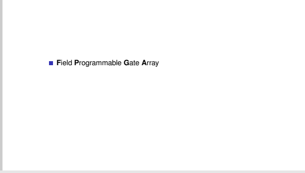


- Map generic circuit elements from compilation to target technology
  - E.g.: ASIC, PAL, Gate-Array, **FPGA**
- Requires a *target library* of available cells
  - Example: set of available gates, sequential elements, etc.
- After this step the target technology is set

Note that after this step the target technology is set and can no longer be changed in subsequent steps.

Technology Mapping

- Map generic circuit elements from compilation to target technology
  - E.g.: ASIC, PAL, Gate-Array, **FPGA**
- Requires a *target library* of available cells
  - Example: set of available gates, sequential elements, etc.
- After this step the target technology is set

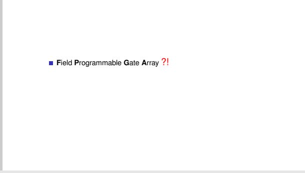


- **F**ield **P**rogrammable **G**ate **A**rray

Let us now introduce FPGAs, the target technology we use in this and further courses. This is an acronym for field-programmable-gate-array.

Introduction to FPGAs

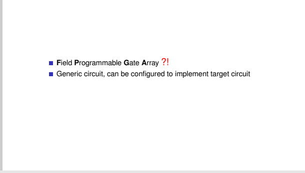
- **F**ield **P**rogrammable **G**ate **A**rray


▪ **F**ield **P**rogrammable **G**ate **A**rray ?!

So far so good, but what does this actually mean?

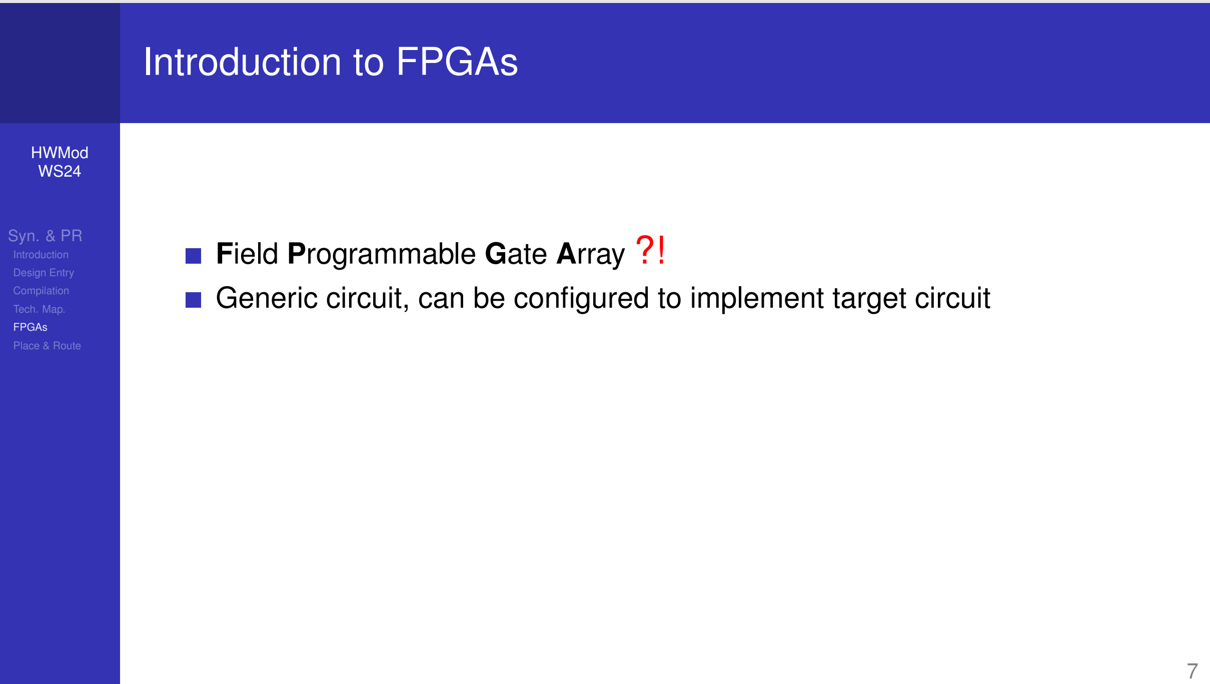
Introduction to FPGAs

- **F**ield **P**rogrammable **G**ate **A**rray ?!



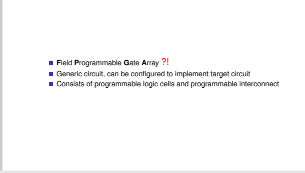
- **F**ield **P**rogrammable **G**ate **A**rray ?!
- Generic circuit, can be configured to implement target circuit

FPGAs are one instance of a class of generic circuits that can be configured to implement a wide range of digital logic circuits. To some limited extent, you can view such generic circuits like a box filled with Lego blocks. As long as you do not run out of blocks, you can build whatever you want assuming it can be built using Lego. Just like this box of toy bricks, generic circuits provide you with a set of different building blocks you can connect to compose a complex system. But how can you actually compose different circuits out of a static chip?



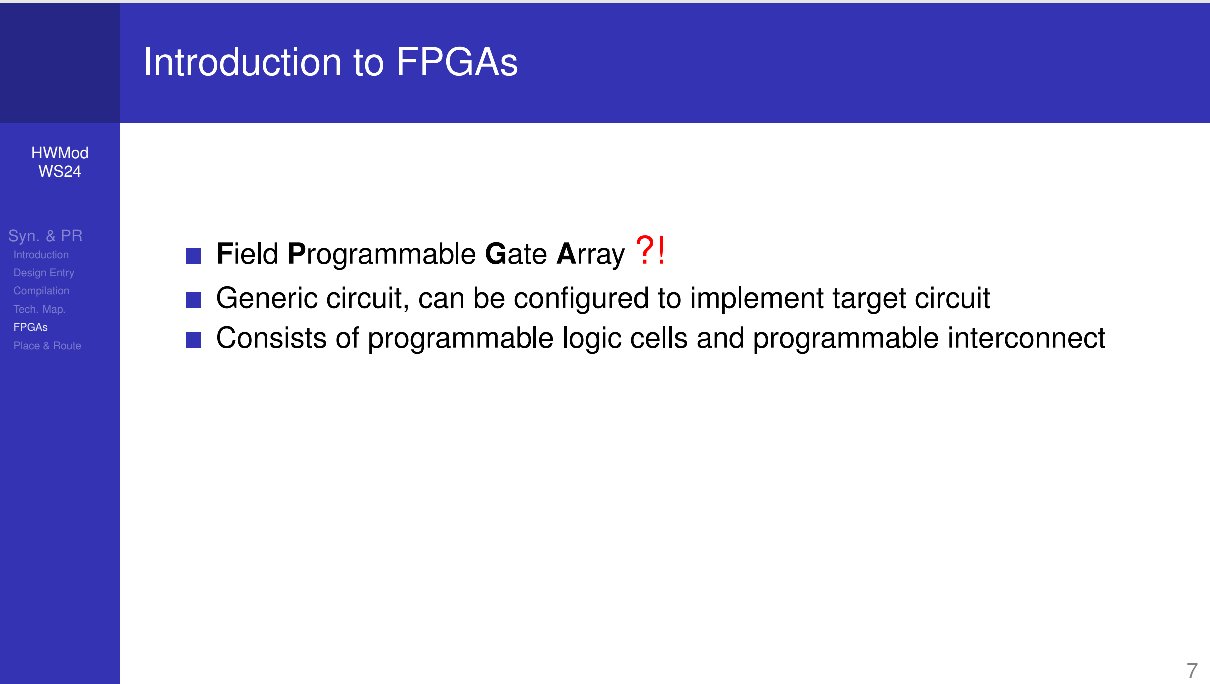
Introduction to FPGAs

- **F**ield **P**rogrammable **G**ate **A**rray ?!
- Generic circuit, can be configured to implement target circuit



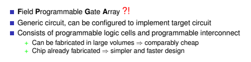
- **F**ield **P**rogrammable **G**ate **A**rray ?!
- Generic circuit, can be configured to implement target circuit
- Consists of programmable logic cells and programmable interconnect

The answer to this question are programmable logic cells and a programmable interconnect. We will view both in detail over the next few slides. However, first let us address why one would use such a generic circuit at all, and why they are not used all the time.

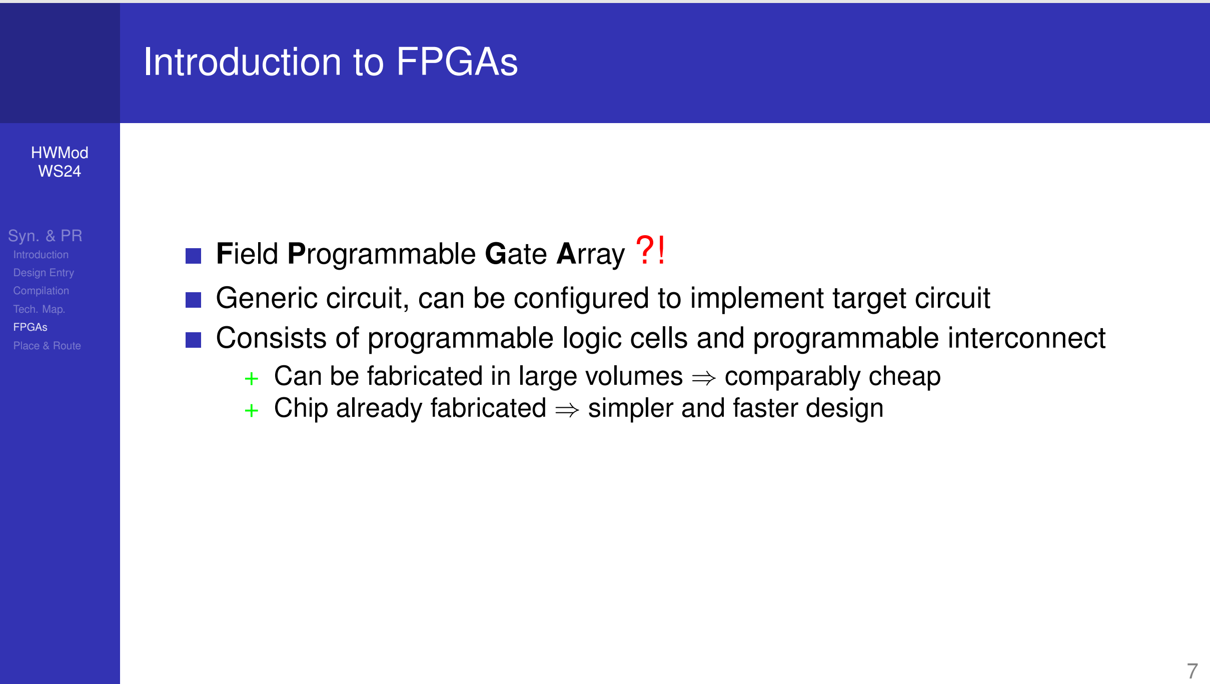


Introduction to FPGAs

- **F**ield **P**rogrammable **G**ate **A**rray ?!
- Generic circuit, can be configured to implement target circuit
- Consists of programmable logic cells and programmable interconnect

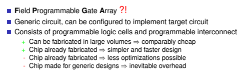


Let us start with some advantages of generic over full custom chips. First, generic chips can be manufactured in larger volumes as they address a larger audience than most application-specific designs do. This makes them, for a hardware design, comparably cheap. Furthermore, if we recall the Y-diagram, some steps like the manufacturing were already done. This makes designing circuits for FPGAs simpler, faster and reduces the entry barrier to getting developing hardware.



Introduction to FPGAs

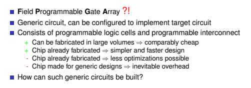
- **F**ield **P**rogrammable **G**ate **A**rray ?!
- Generic circuit, can be configured to implement target circuit
- Consists of programmable logic cells and programmable interconnect
  - + Can be fabricated in large volumes ⇒ comparably cheap
  - + Chip already fabricated ⇒ simpler and faster design


- **F**ield **P**rogrammable **G**ate **A**rray ?!
- Generic circuit, can be configured to implement target circuit
- Consists of programmable logic cells and programmable interconnect
  - + Can be fabricated in large volumes ⇒ comparably cheap
  - + Chip already fabricated ⇒ simpler and faster design
  - - Chip already fabricated ⇒ less optimizations possible
  - - Chip made for generic designs ⇒ inevitable overhead

So far so good, but what about disadvantages? While having an already manufactured chip comes with some benefits, it also has some drawbacks when compared to a full custom design. In particular, the amount of possible optimizations is limited. Furthermore, a chip that is made for arbitrary designs is inevitably subject to overhead. To revisit the Lego example: Assume you are provided with a box of an equal amount of red and blue blocks because most people need a combination of both. If you only happen to require red blocks, the blue blocks are for naught, only consuming space you could use for red blocks.

Introduction to FPGAs

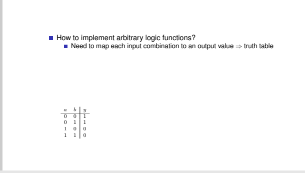
- **F**ield **P**rogrammable **G**ate **A**rray ?!
- Generic circuit, can be configured to implement target circuit
- Consists of programmable logic cells and programmable interconnect
  - + Can be fabricated in large volumes ⇒ comparably cheap
  - + Chip already fabricated ⇒ simpler and faster design
  - - Chip already fabricated ⇒ less optimizations possible
  - - Chip made for generic designs ⇒ inevitable overhead


- **F**ield **P**rogrammable **G**ate **A**rray ?!
- Generic circuit, can be configured to implement target circuit
- Consists of programmable logic cells and programmable interconnect
  - + Can be fabricated in large volumes ⇒ comparably cheap
  - + Chip already fabricated ⇒ simpler and faster design
  - - Chip already fabricated ⇒ less optimizations possible
  - - Chip made for generic designs ⇒ inevitable overhead
- How can such generic circuits be built?

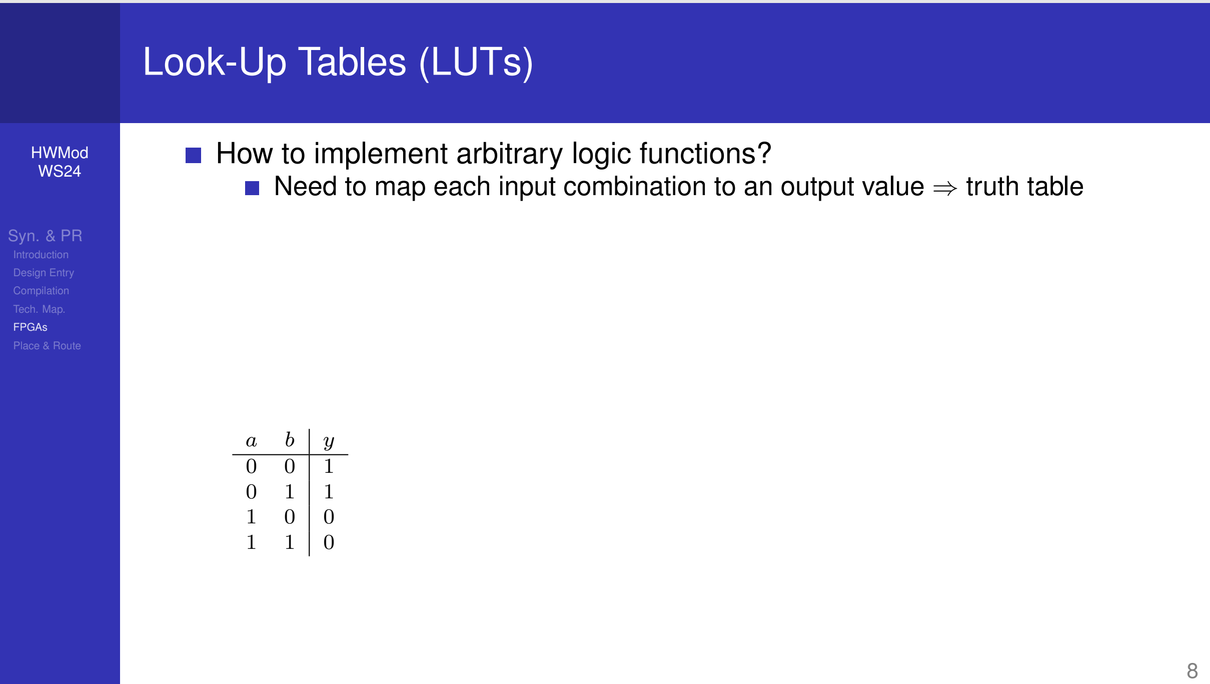
At this point this generic circuits likely still appear quite magical. Therefore, we will break down how a simple generic circuit could be implemented on the next few slides.

Introduction to FPGAs

- **F**ield **P**rogrammable **G**ate **A**rray ?!
- Generic circuit, can be configured to implement target circuit
- Consists of programmable logic cells and programmable interconnect
  - + Can be fabricated in large volumes ⇒ comparably cheap
  - + Chip already fabricated ⇒ simpler and faster design
  - - Chip already fabricated ⇒ less optimizations possible
  - - Chip made for generic designs ⇒ inevitable overhead
- How can such generic circuits be built?



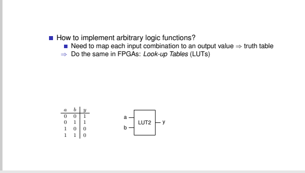
Let us first address the issue that we must be able to implement arbitrary digital logic in our FPGAs. As we all know, defining an arbitrary digital logic function requires us to map each possible input to an output. We do this using a truth table, just as the one shown on the slide for the function not "A".



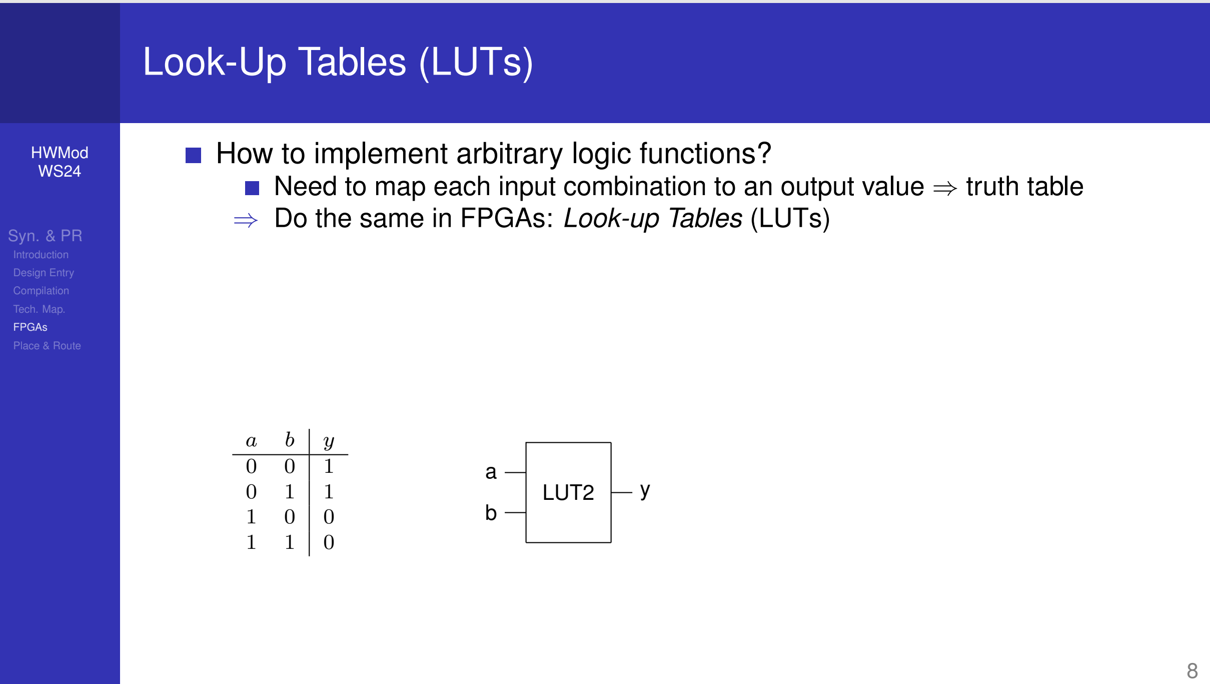
Look-Up Tables (LUTs)

- How to implement arbitrary logic functions?
  - Need to map each input combination to an output value ⇒ truth table

a	b	y
0	0	1
0	1	1
1	0	0
1	1	0



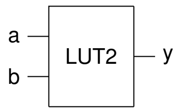
One way to build programmable logic is doing exactly the same. So-called look-up tables are used, which take arbitrary input vectors and return the respective output. The slide shows a two-input look-up table as blackbox. This element can implement all sixteen possible two-input functions. Let us now look how this abstract look-up table can be implemented in hardware.

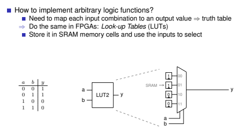


Look-Up Tables (LUTs)

- How to implement arbitrary logic functions?
  - Need to map each input combination to an output value ⇒ truth table
  - ⇒ Do the same in FPGAs: *Look-up Tables* (LUTs)

a	b	y
0	0	1
0	1	1
1	0	0
1	1	0



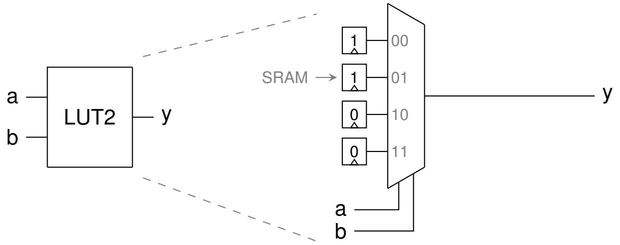


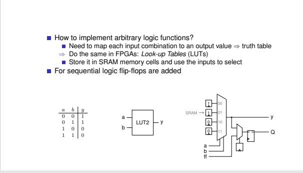
A straightforward implementation is based on SRAM. For our purpose it suffices to consider this kind of memory as being programmed once after starting our circuit, and then providing the programmed values until the power supply is cut. We can use such memory cells to store the truth table of our desired logic function. These memory cells are then connected to a multiplexer controlled by the function's input signals. Depending on their value the respective truth table value is forwarded.

Look-Up Tables (LUTs)

- How to implement arbitrary logic functions?
  - Need to map each input combination to an output value ⇒ truth table
  - ⇒ Do the same in FPGAs: *Look-up Tables* (LUTs)
  - Store it in SRAM memory cells and use the inputs to select

a	b	y
0	0	1
0	1	1
1	0	0
1	1	0



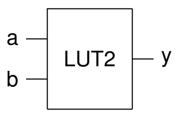

- How to implement arbitrary logic functions?
  - Need to map each input combination to an output value ⇒ truth table
  - ⇒ Do the same in FPGAs: *Look-up Tables* (LUTs)
  - Store it in SRAM memory cells and use the inputs to select
- For sequential logic flip-flops are added

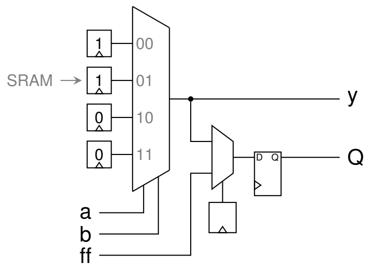
Since we do not only want to build combinational, but also sequential logic, we equip our programmable logic block also with an optional flip-flop. Depending on another SRAM cell we can either use the flip-flop on its own or to sample the truth table output.

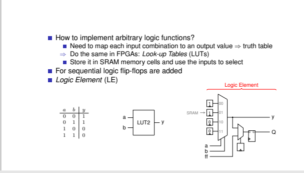
Look-Up Tables (LUTs)

- How to implement arbitrary logic functions?
  - Need to map each input combination to an output value ⇒ truth table
  - ⇒ Do the same in FPGAs: *Look-up Tables* (LUTs)
  - Store it in SRAM memory cells and use the inputs to select
- For sequential logic flip-flops are added

a	b	y
0	0	1
0	1	1
1	0	0
1	1	0





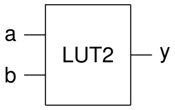


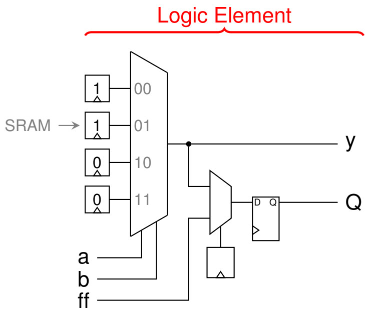
And with this we have our simple programmable logic circuit. All we have to do is to program the SRAM cells and we get an arbitrary two-input logic function with optional flip-flop. We will call blocks like this logic elements from now on. Please pause the video for a moment, study the circuit in detail and make sure that you understand why and how this is programmable.

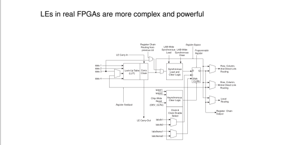
Look-Up Tables (LUTs)

- How to implement arbitrary logic functions?
  - Need to map each input combination to an output value ⇒ truth table
  - ⇒ Do the same in FPGAs: *Look-up Tables* (LUTs)
  - Store it in SRAM memory cells and use the inputs to select
- For sequential logic flip-flops are added
- *Logic Element* (LE)

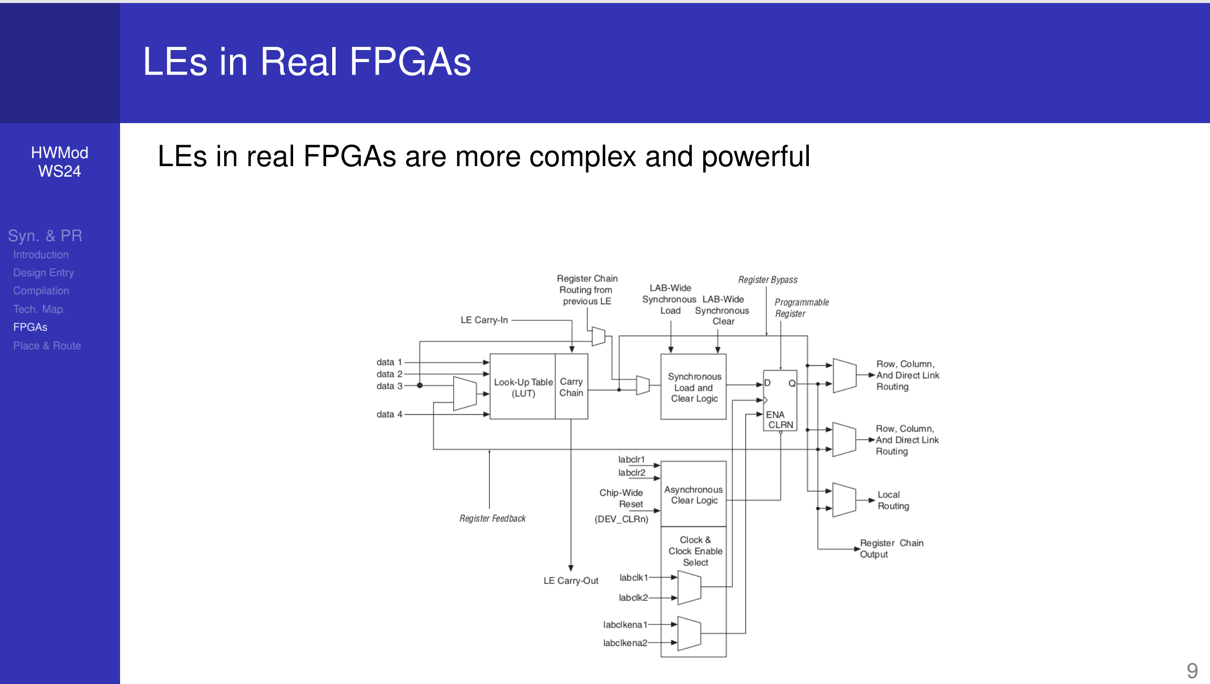
a	b	y
0	0	1
0	1	1
1	0	0
1	1	0







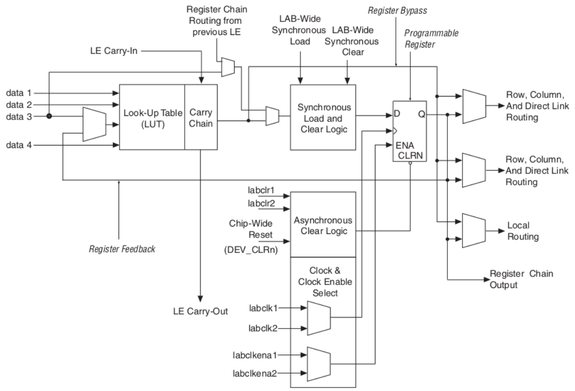
While our simple logic element is illustrative and generally not too far from reality, the logic elements in real FPGAs are usually more complex and also powerful. The slide shows the schematic of a logic element used in an actual FPGA.

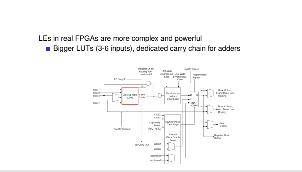


LEs in Real FPGAs

HWMod
WS24

LEs in real FPGAs are more complex and powerful

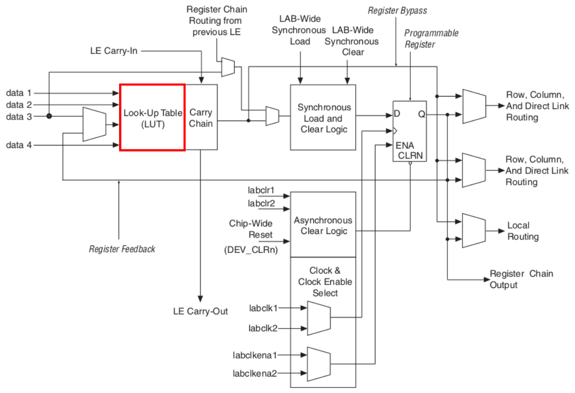

Register Chain Routing from previous LE
LAB-Wide Synchronous Load
LAB-Wide Synchronous Clear
Register Bypass
Programmable Register
LE Carry-In
data 1
data 2
data 3
data 4
Look-Up Table (LUT)
Carry Chain
Synchronous Load and Clear Logic
D
Q
ENA
CLRN
Row, Column, And Direct Link Routing
Row, Column, And Direct Link Routing
Local Routing
Register Chain Output
labclr1
labclr2
Chip-Wide Reset (DEV_CLRn)
Asynchronous Clear Logic
Register Feedback
Clock & Clock Enable Select
LE Carry-Out
labclk1
labclk2
labclkena1
labclkena2

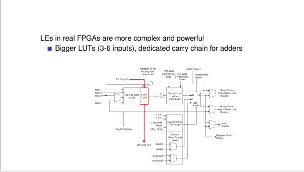

LEs in real FPGAs are more complex and powerful
- Bigger LUTs (3-6 inputs), dedicated carry chain for adders

Among other things, we can observe that the look-up table has four inputs instead of two, allowing it to implement significantly more logic functions.

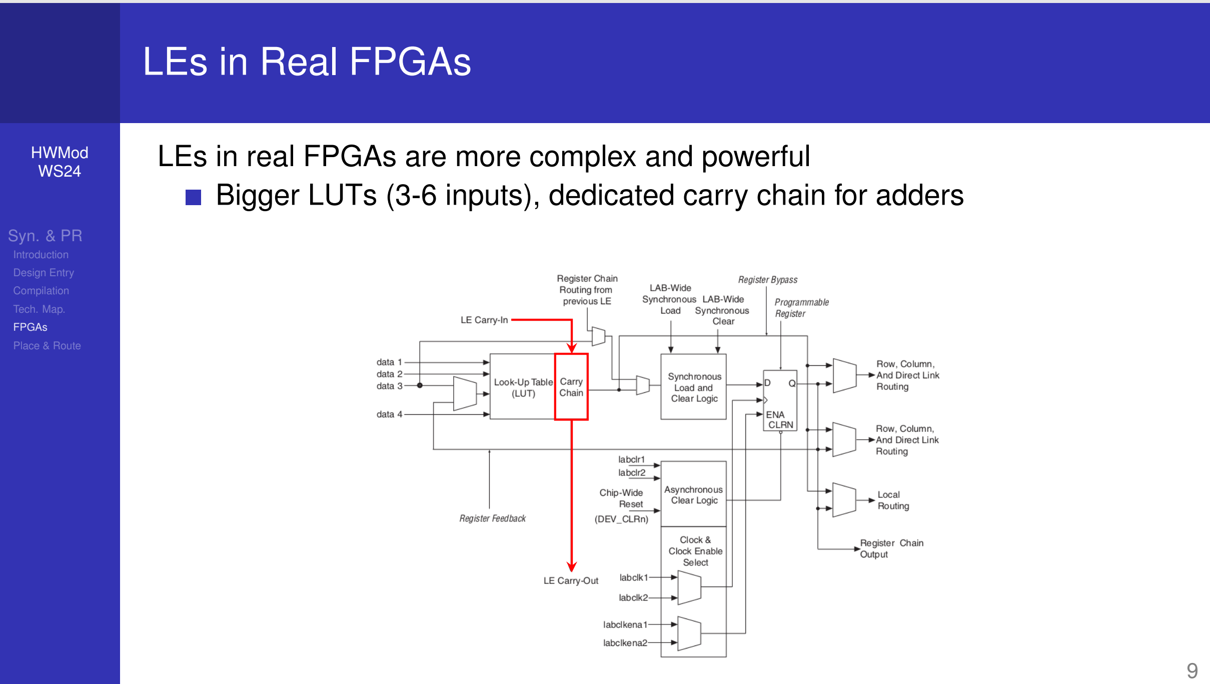
LEs in Real FPGAs

LEs in real FPGAs are more complex and powerful
- Bigger LUTs (3-6 inputs), dedicated carry chain for adders


Register Chain Routing from previous LE
LAB-Wide Synchronous Load
LAB-Wide Synchronous Clear
Register Bypass
Programmable Register
LE Carry-In
data 1
data 2
data 3
data 4
Look-Up Table (LUT)
Carry Chain
Synchronous Load and Clear Logic
D
Q
ENA
CLRN
Row, Column, And Direct Link Routing
Row, Column, And Direct Link Routing
Local Routing
Register Chain Output
labclr1
labclr2
Chip-Wide Reset (DEV_CLRn)
Asynchronous Clear Logic
Register Feedback
Clock & Clock Enable Select
LE Carry-Out
labclk1
labclk2
labclkena1
labclkena2



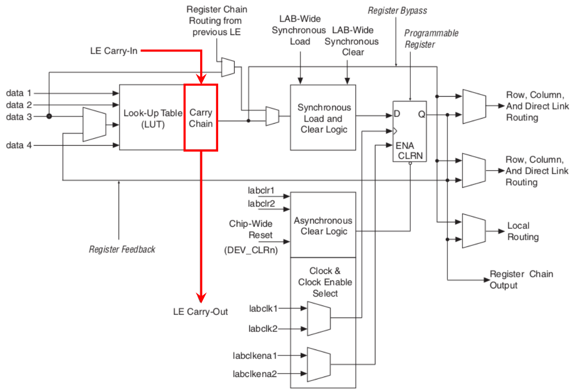
Furthermore, the table is equipped with a dedicated logic for carry chains which allows efficient adder implementations. The reason for that is that addition is required often, and it is typical practice to make the common case fast.

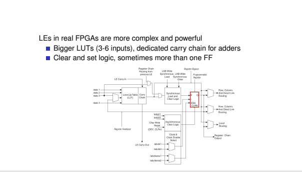


LEs in Real FPGAs

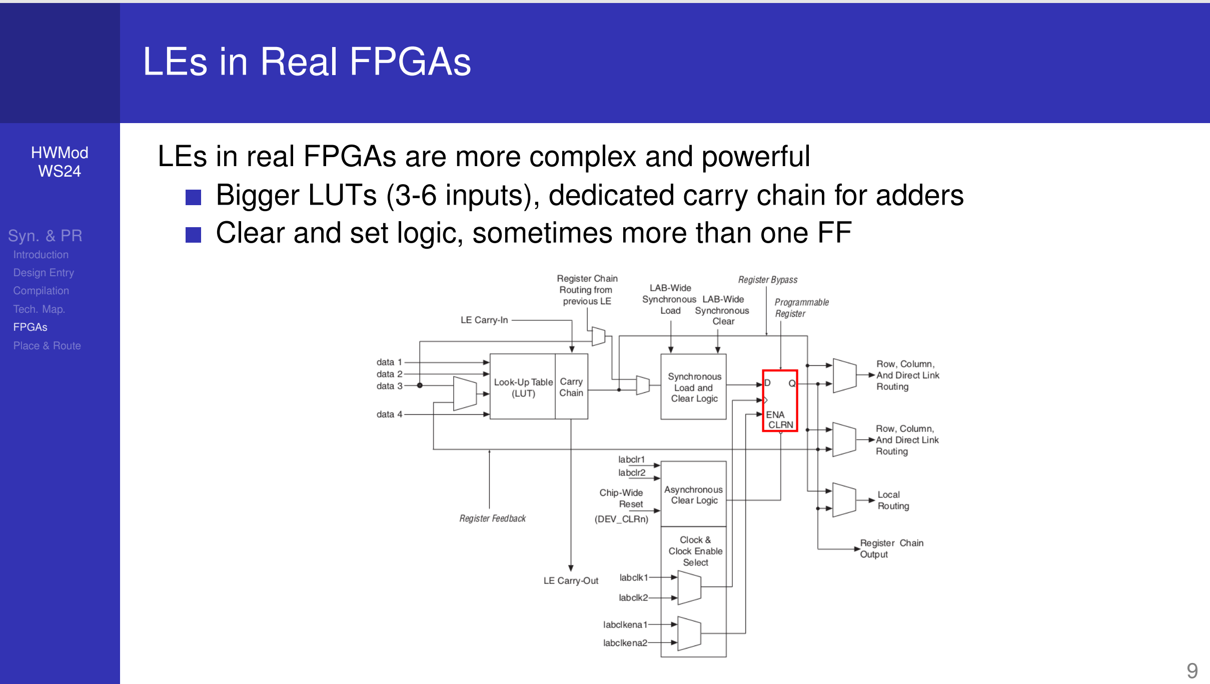
LEs in real FPGAs are more complex and powerful

- Bigger LUTs (3-6 inputs), dedicated carry chain for adders


Register Chain Routing from previous LE
LAB-Wide Synchronous Load
LAB-Wide Synchronous Clear
Register Bypass
Programmable Register
LE Carry-In
data 1
data 2
data 3
data 4
Look-Up Table (LUT)
Carry Chain
Synchronous Load and Clear Logic
D
Q
ENA
CLRN
Row, Column, And Direct Link Routing
Row, Column, And Direct Link Routing
Local Routing
Register Chain Output
labclr1
labclr2
Chip-Wide Reset (DEV_CLRn)
Asynchronous Clear Logic
Clock & Clock Enable Select
Register Feedback
LE Carry-Out
labclk1
labclk2
labclkena1
labclkena2

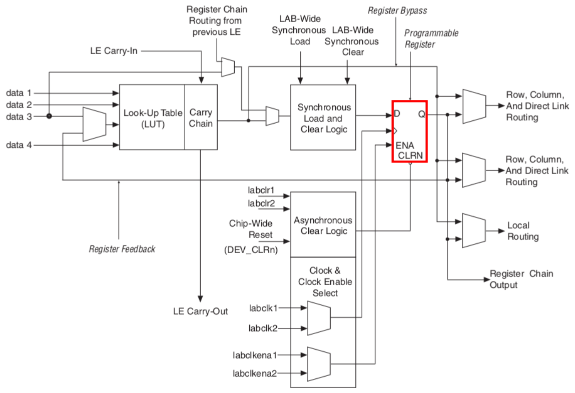


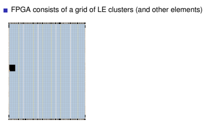
The flip-flop comes with some additional logic attached to it, allowing to feed it from more sources, setting or clearing its content and even disabling it. Furthermore, some FPGAs feature logic elements with more than a single flip flop. However, this is not the case for the one shown on the slide.



LEs in Real FPGAs

LEs in real FPGAs are more complex and powerful
- Bigger LUTs (3-6 inputs), dedicated carry chain for adders
- Clear and set logic, sometimes more than one FF

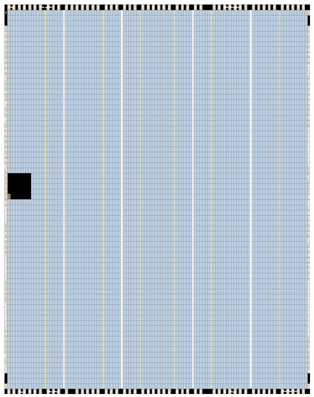

Register Chain Routing from previous LE
LAB-Wide Synchronous Load
LAB-Wide Synchronous Clear
Register Bypass
Programmable Register
LE Carry-In
data 1
data 2
data 3
data 4
Look-Up Table (LUT)
Carry Chain
Synchronous Load and Clear Logic
D
Q
ENA
CLRN
Row, Column, And Direct Link Routing
Row, Column, And Direct Link Routing
Local Routing
Register Chain Output
labclr1
labclr2
Chip-Wide Reset (DEV_CLRn)
Asynchronous Clear Logic
Clock & Clock Enable Select
Register Feedback
LE Carry-Out
labclk1
labclk2
labclkena1
labclkena2

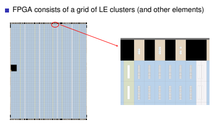


In its majority, an FPGA consists of such logic elements. Multiple logic elements are gathered in a cluster and such clusters are then physically arranged in a grid. The image on the slide shows the overview of an actual FPGA chip, where everything shaded blue consists of logic elements. Let us now zoom into this grid.

FPGA Structure

- FPGA consists of a grid of LE clusters (and other elements)

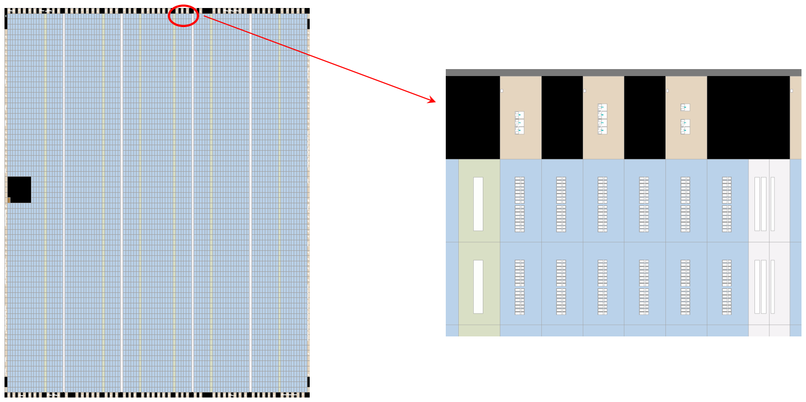


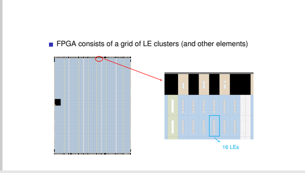


This view allows us to clearly observe the grid of rectangular cells. Note how the chip does not only consist of a single type of cell, but rather of multiple different ones, highlighted by the different colors.

FPGA Structure

- FPGA consists of a grid of LE clusters (and other elements)

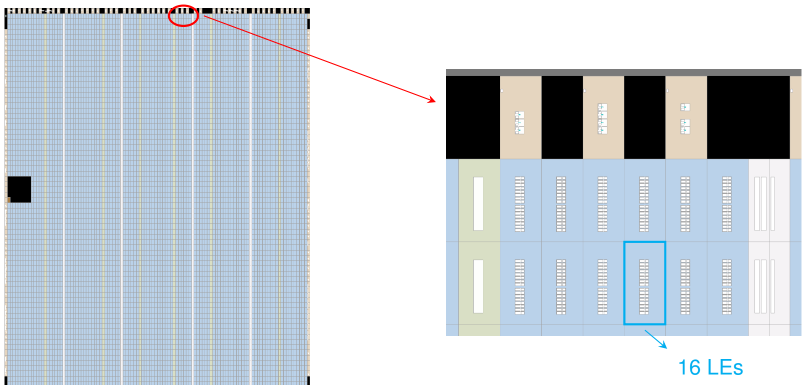


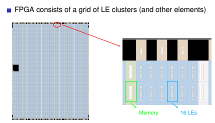


The majority of cells are the clusters of logic elements. In this up-close view we can clearly observe that for this particular FPGA logic elements form clusters of size sixteen.

FPGA Structure

- FPGA consists of a grid of LE clusters (and other elements)

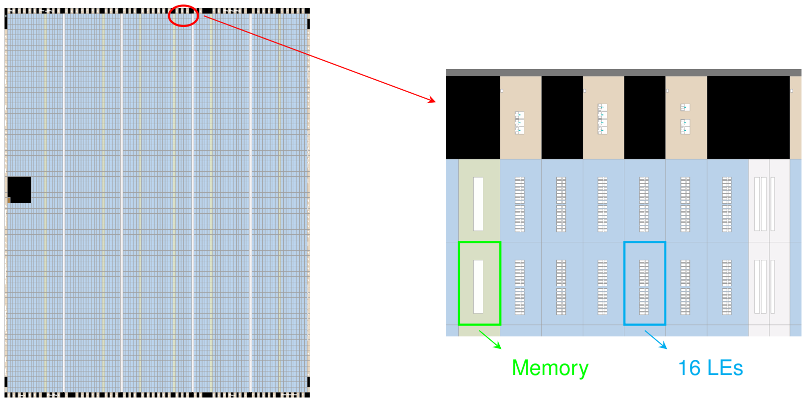


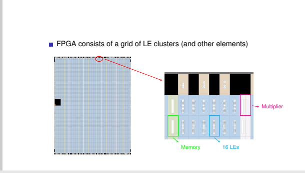


Next, we also have some dedicated memory cells. The reason for that is that many designs require memory to store data. And while it is possible to build memory out of logic elements, this is very costly in terms of chip area and also performance. To combat this FPGA vendors tend to equip their chips with such embedded memory blocks that offer a lot denser and faster memory.

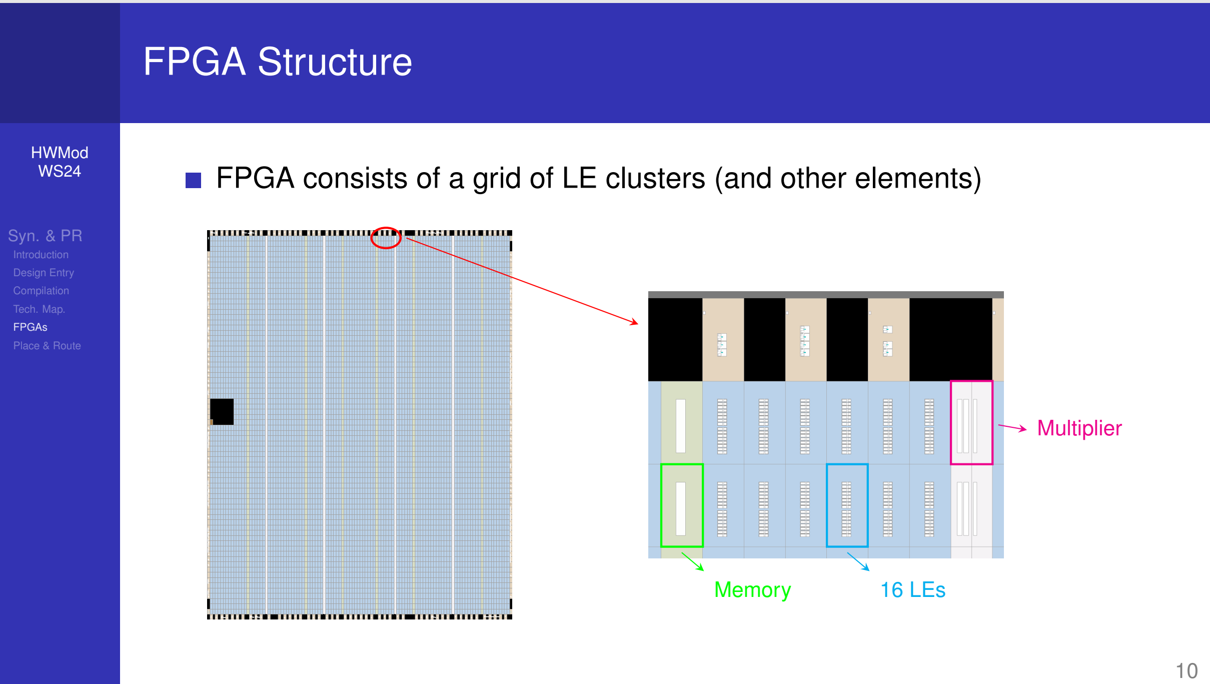
FPGA Structure

- FPGA consists of a grid of LE clusters (and other elements)



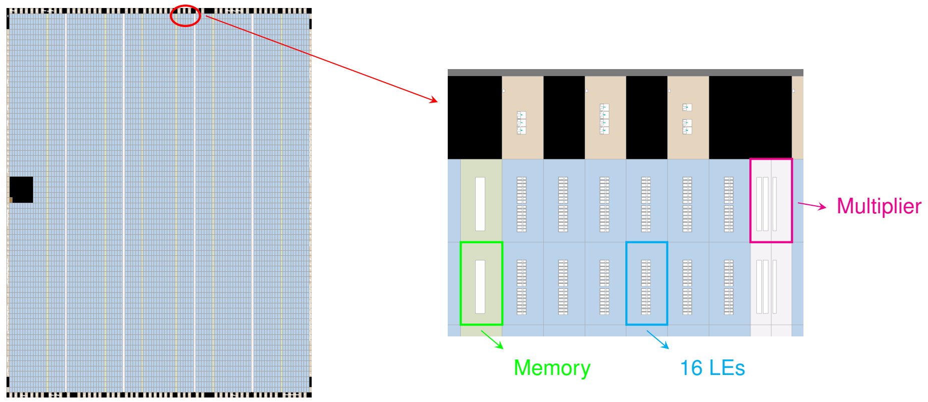


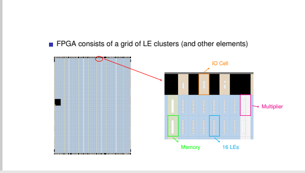
Furthermore, just as the dedicated adder logic we could observe in the logic element on the previous slide, many FPGAs come with dedicated multipliers. Just as with memories and adders, this is a lot more area efficient and faster than building multipliers out of logic elements considering that multipliers are used often.



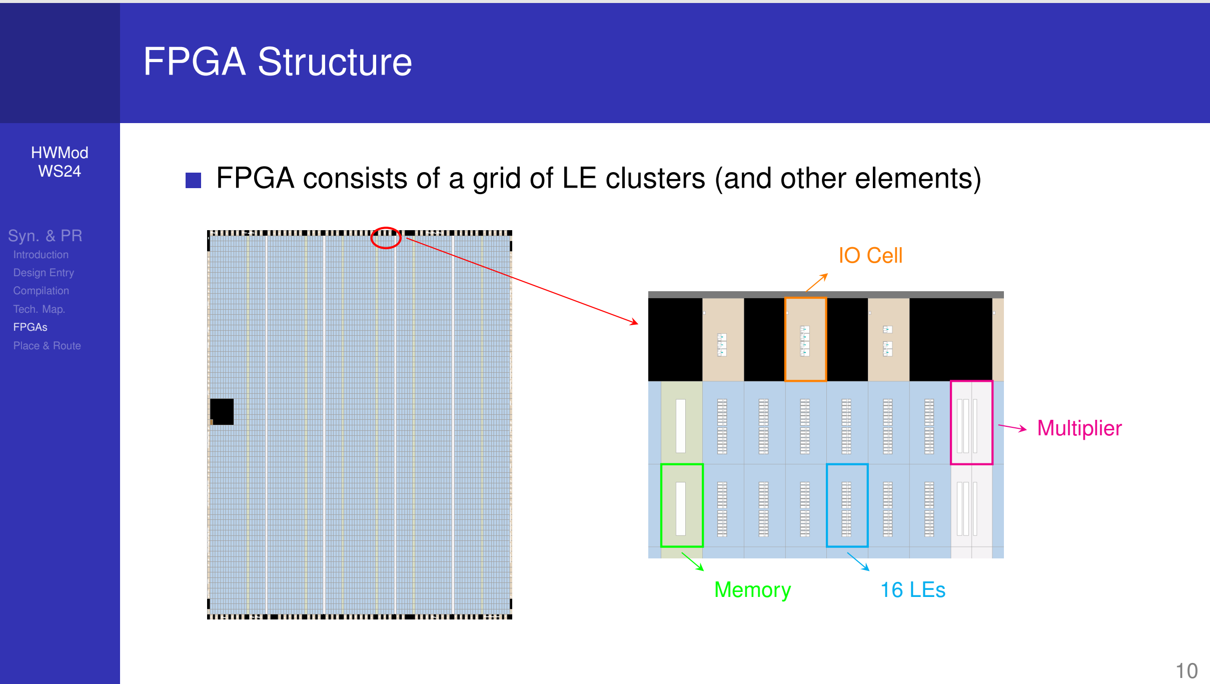
FPGA Structure

- FPGA consists of a grid of LE clusters (and other elements)



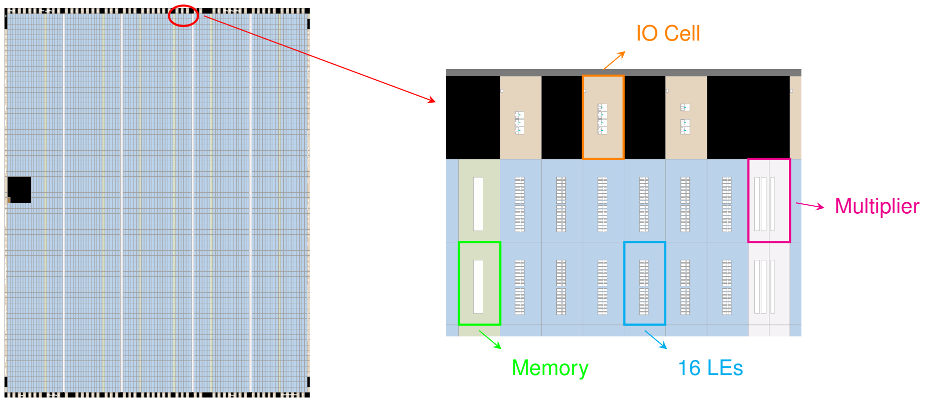


In order to interface with the outside world, FPGA chips are equipped with IO-cells that provide a plethora of different input and output functionality. Finally, the black areas mark parts of the chip that are not equipped with any usable logic.



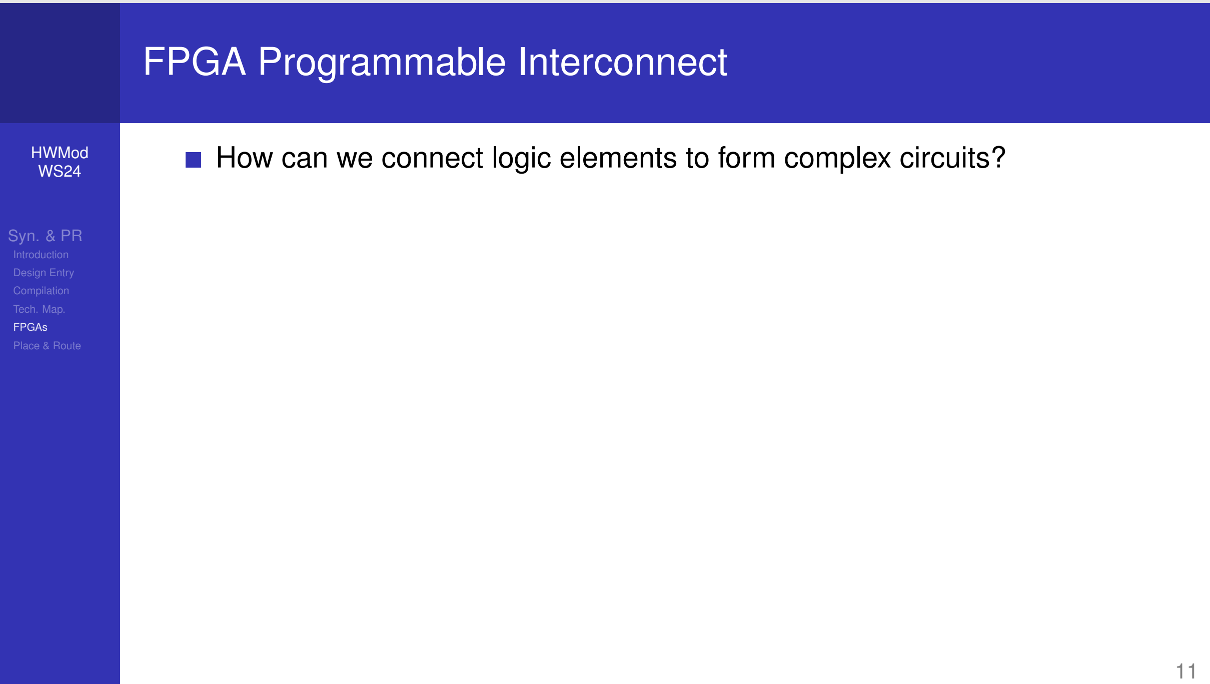
FPGA Structure

- FPGA consists of a grid of LE clusters (and other elements)



■ How can we connect logic elements to form complex circuits?

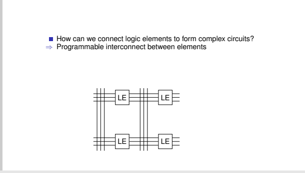
While programmable logic cells are paramount for creating a generic circuit, they are only one part of it. We also need a means to compose networks of this simple cells in order to implement our target circuits.



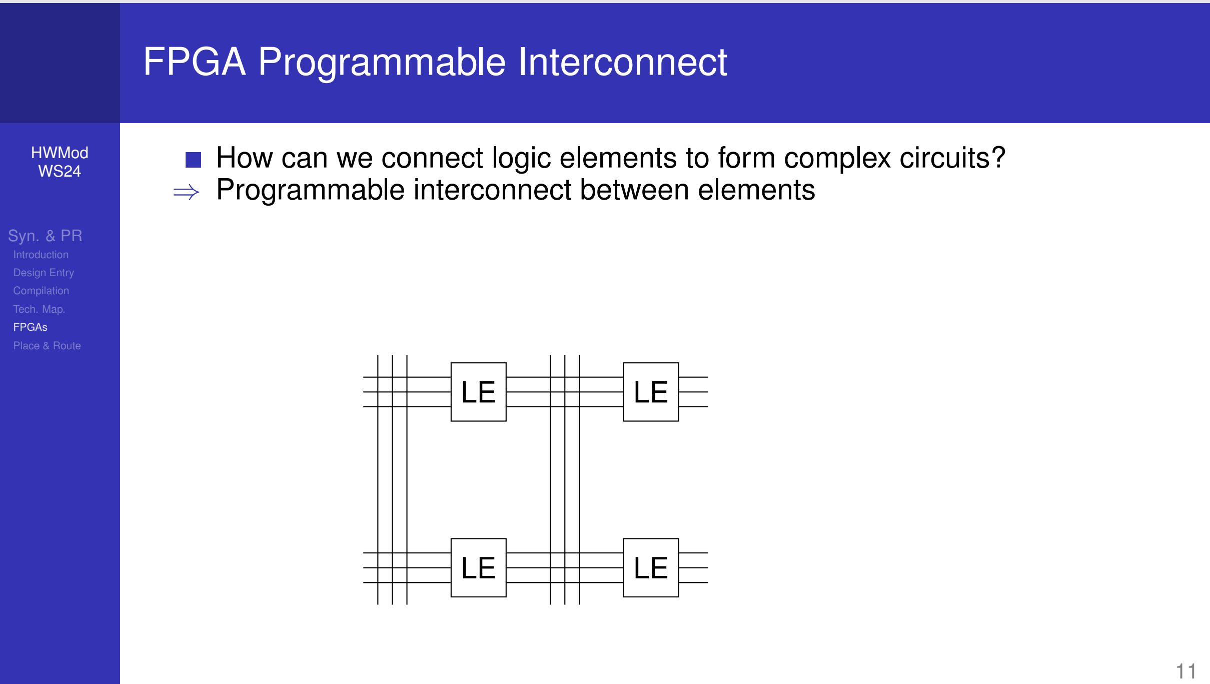
FPGA Programmable Interconnect

HWMod
WS24

- How can we connect logic elements to form complex circuits?



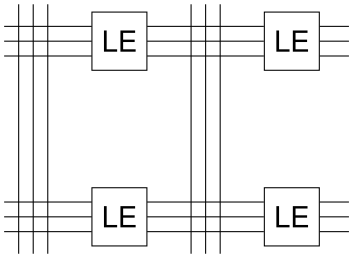
FPGAs can connect logic elements by using programmable interconnect. In a nutshell, all logic elements are connected to a switching matrix where multiple wires of adjacent logic elements and dedicated transport wires intersect. By programming which of these intersections are tapped, we can configure where the inputs of logic elements come from and where their outputs are forwarded to.

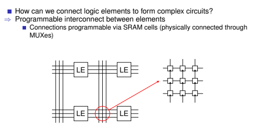


FPGA Programmable Interconnect

- How can we connect logic elements to form complex circuits?

⇒ Programmable interconnect between elements



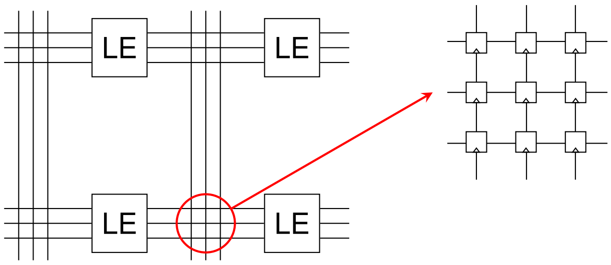


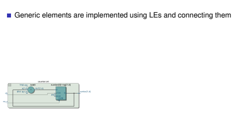
If we zoom in into such a switching matrix, we can see that the programmable intersections are again implemented using SRAM cells. The respective switching itself happens using multiplexers. We will not delve into this in more detail in this lecture.

FPGA Programmable Interconnect

- How can we connect logic elements to form complex circuits?

⇒ Programmable interconnect between elements
  - Connections programmable via SRAM cells (physically connected through MUXes)

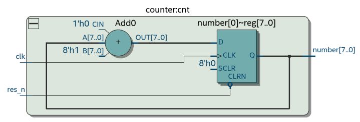


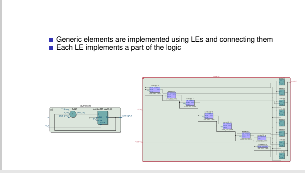


Let us now return to the technology mapping step of a synchronous counter to an SRAM-based FPGA. Recall that the overall goal was to map the generic circuit shown on the slides to the target technology. For an FPGA this means that we must map the circuit to a network of logic elements.

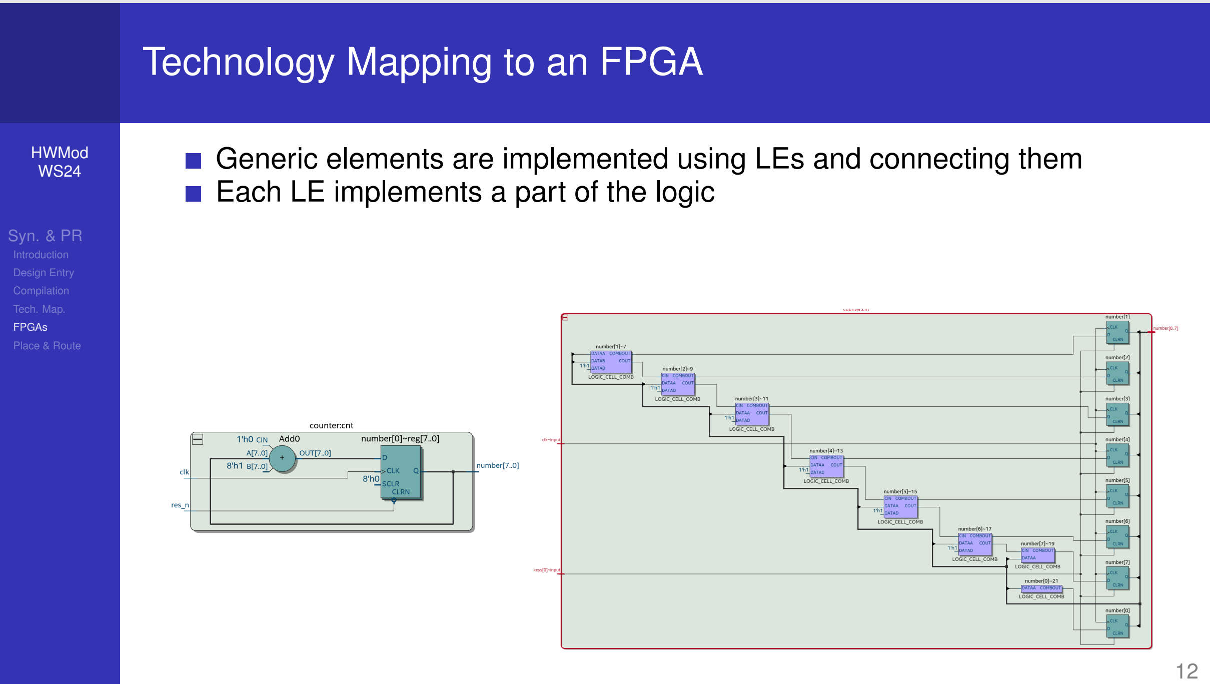
Technology Mapping to an FPGA

- Generic elements are implemented using LEs and connecting them



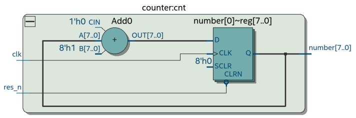


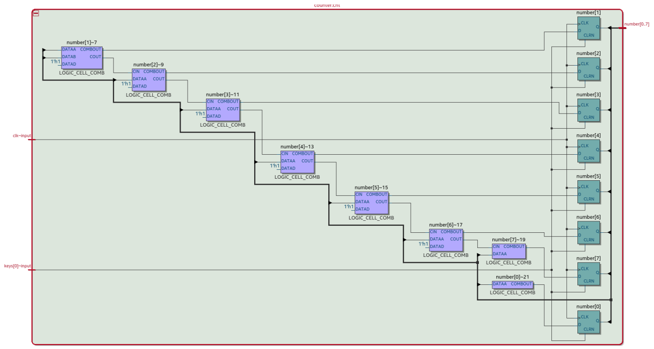
On the right of the slide the result of such a mapping, performed by a tool, is shown. The teal boxes right are the eight flip-flops our eight bit counter requires. These flip-flops are fed by a chain of logic. Let us have a closer look at one of these logic blocks.

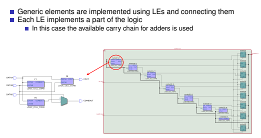


Technology Mapping to an FPGA

- Generic elements are implemented using LEs and connecting them
- Each LE implements a part of the logic



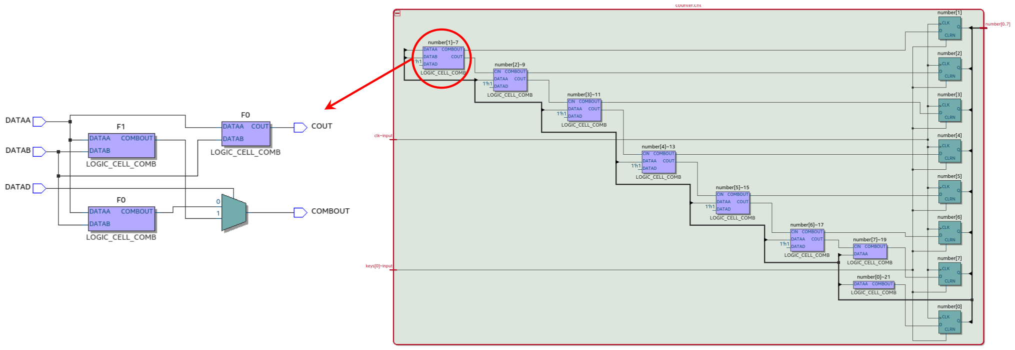


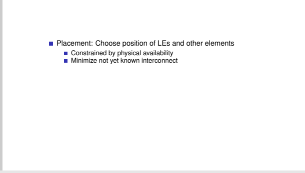

- Generic elements are implemented using LEs and connecting them
- Each LE implements a part of the logic
  - In this case the available carry chain for adders is used

Zooming into such a block reveals that in our particular example the dedicated carry chain for adders is used instead of the lookup table, clearly shown by use of the carry out, ”C-OUT” output. This makes sense considering that we modelled a counter. The tool thus picked the most efficient implementation available in the target technology.

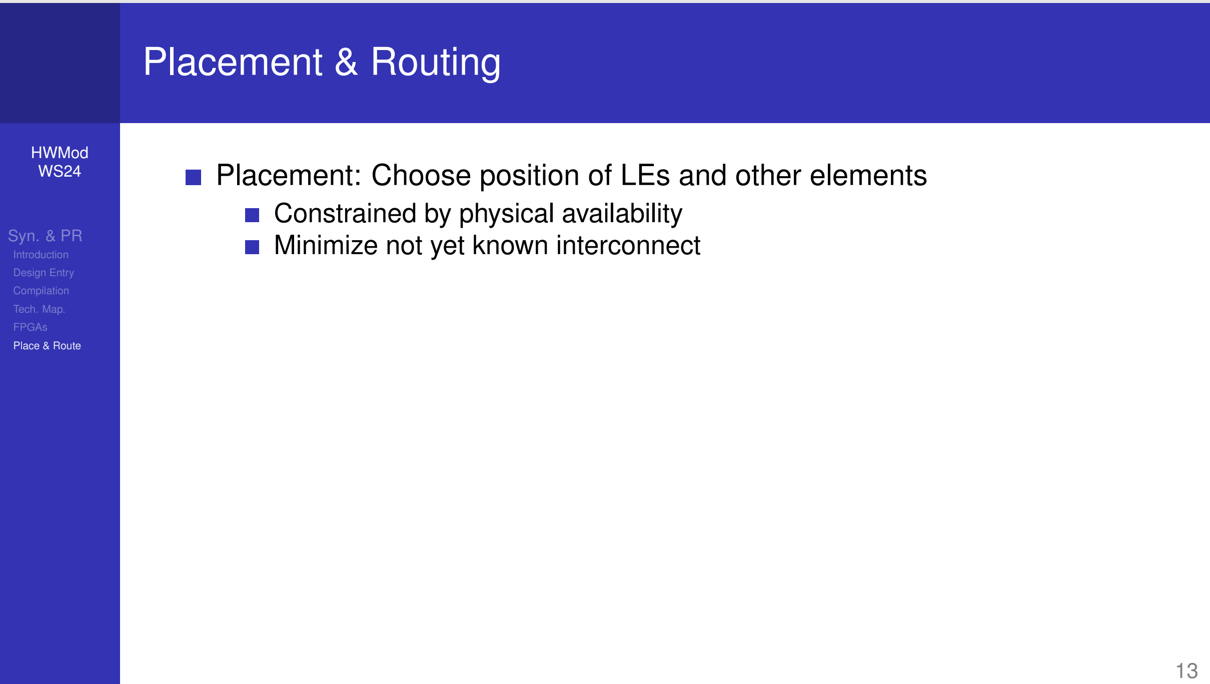
Technology Mapping to an FPGA

- Generic elements are implemented using LEs and connecting them
- Each LE implements a part of the logic
  - In this case the available carry chain for adders is used

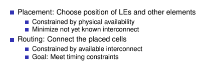



- Placement: Choose position of LEs and other elements
  - Constrained by physical availability
  - Minimize not yet known interconnect

The final step in obtaining a physical description of our design is the placement and routing one. As the name suggests, these are actually two steps. However, due to their close interplay we will cover them together. First, let's consider the placement step. As mentioned at the beginning of this lecture, the task of this step is to place the elements used in the network produced by the technology mapping step to the available physical area. For an FPGA this means that we must assign each logic element, memory cell, multiplier and IO cell to a cell of the grid of the respective type. While this might sound easy, it is actually highly non-trivial. First, for very complex designs that fill most of the available area the physical constraints of possible cell locations can make it hard to place connected cells next to another. Furthermore, naturally the placement has a big impact on the wires used to connect cells and thus on the overall timing. In order to meet timing constraints the placer thus has to minimize delay due to the wiring without knowing the wiring yet.


Placement & Routing

- Placement: Choose position of LEs and other elements
  - Constrained by physical availability
  - Minimize not yet known interconnect



- Placement: Choose position of LEs and other elements
  - Constrained by physical availability
  - Minimize not yet known interconnect
- Routing: Connect the placed cells
  - Constrained by available interconnect
  - Goal: Meet timing constraints

Computing this wiring is the job of the routing step. It takes the cell locations generated by the placer and tries to connect them via the available interconnect resources. As mentioned, the wiring between cells must satisfy the required timing constraints.

Placement & Routing

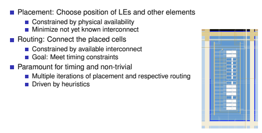
- Placement: Choose position of LEs and other elements
  - Constrained by physical availability
  - Minimize not yet known interconnect
- Routing: Connect the placed cells
  - Constrained by available interconnect
  - Goal: Meet timing constraints

- Placement: Choose position of LEs and other elements
  - Constrained by physical availability
  - Minimize not yet known interconnect
- Routing: Connect the placed cells
  - Constrained by available interconnect
  - Goal: Meet timing constraints
- Paramount for timing and non-trivial
  - Multiple iterations of placement and respective routing
  - Driven by heuristics

In general, placing and routing cells is highly non-trivial. To combat this the placing and routing steps are usually executed in an iterative manner where the placer gets timing information about previous tries and thus is able to incrementally improve its output. Furthermore, both steps are typically guided by heuristics.

Placement & Routing

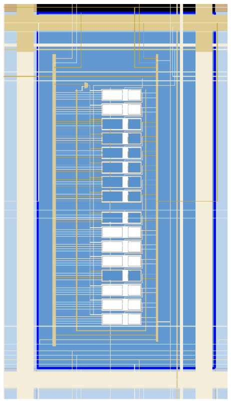
- Placement: Choose position of LEs and other elements
  - Constrained by physical availability
  - Minimize not yet known interconnect
- Routing: Connect the placed cells
  - Constrained by available interconnect
  - Goal: Meet timing constraints
- Paramount for timing and non-trivial
  - Multiple iterations of placement and respective routing
  - Driven by heuristics

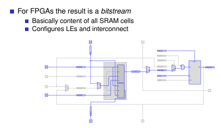


On the right side of the slide the final result of placing and routing our simple example circuit on an FPGA is shown. We can observe that the eight logic elements used in the network produced by the technology mapping step are placed within a single cluster. The brown lines show how the cells are connected to another and to the chip's IO cells.

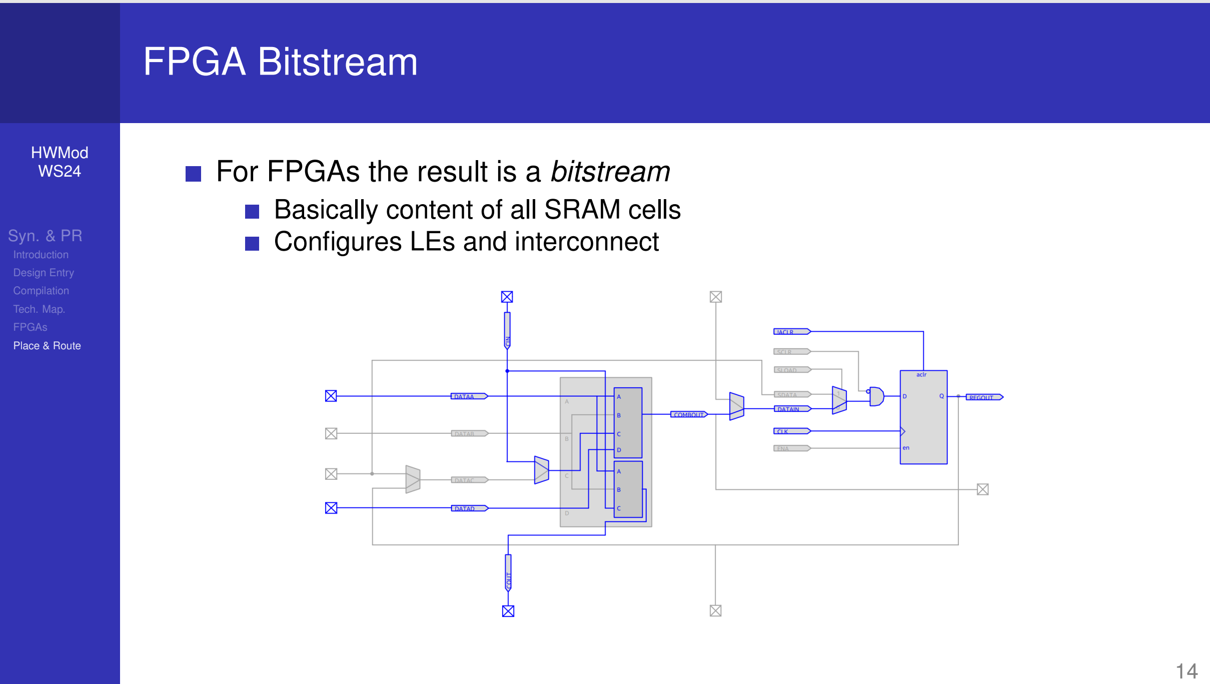
Placement & Routing

- Placement: Choose position of LEs and other elements
  - Constrained by physical availability
  - Minimize not yet known interconnect
- Routing: Connect the placed cells
  - Constrained by available interconnect
  - Goal: Meet timing constraints
- Paramount for timing and non-trivial
  - Multiple iterations of placement and respective routing
  - Driven by heuristics



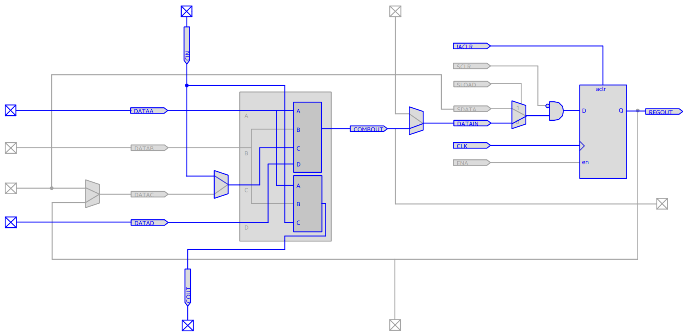


In case of an FPGA, the final result of the design flow is a so-called bitstream. This basically encodes the content of all the chip's SRAM cells. By programming the SRAM cells using this bitstream, all logic elements and the interconnect are configured as required to implement the target circuit. The figure at the bottom of the slide illustrates how such a configuration looks like for a single logic element of our counter implementation. We can see the dedicated carry chain being used and fed into the flip-flop. This flip-flop is connected to an asynchronous reset and clock, just as in our VHDL code.



FPGA Bitstream

- For FPGAs the result is a *bitstream*
  - Basically content of all SRAM cells
  - Configures LEs and interconnect



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.

Lecture Complete!