

How does VHDL code become actual hardware? In this lecture you will learn about the essential steps that transform your behavioral description into a physical circuit implementation. We will explore logic synthesis, technology mapping, and placement and routing, using FPGAs as our target technology.

HWMMod
WS25

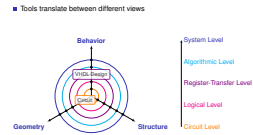
Syn. & PR

Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

Hardware Modeling [VU] (191.011) – WS25 – Logic Synthesis, Place and Route

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

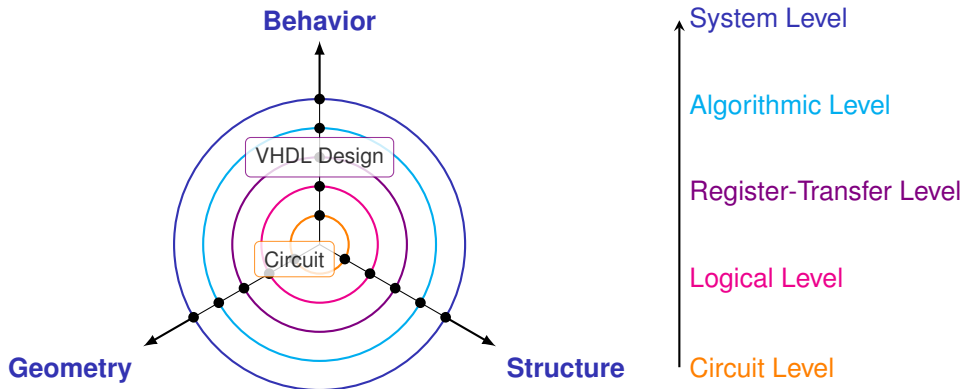
WS 2025/26

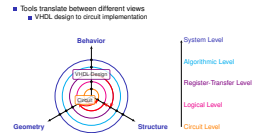


Recall the Y-diagram from previous lectures. We introduced it as a useful tool for capturing and illustrating overall flow from a model of a circuit to a proper implementation. In essence, the Y-diagram expresses that a model must be equipped with increasing detail to implement it. However, you also learned that there might exist multiple paths when descending from the model's abstraction level to the final circuit. These paths form due to translations between different viewpoints, which are done by tools that harness their respective advantages.

Recall: Y-Diagram

- Tools translate between different views

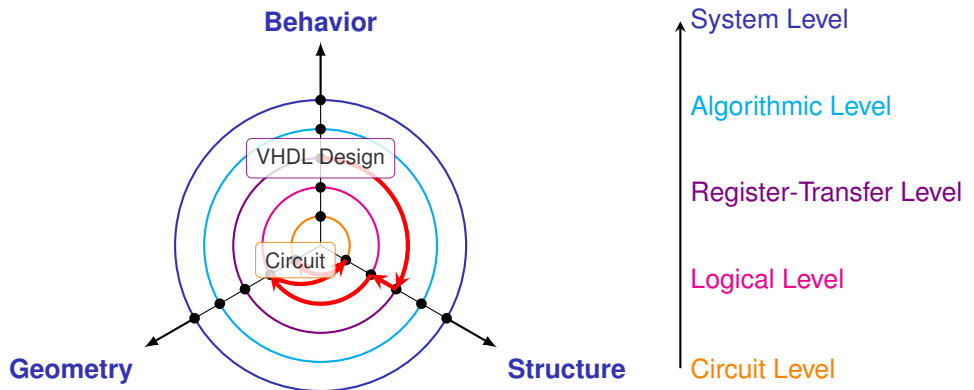


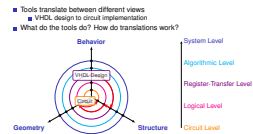


On the slide you can find an example path, drawn in red, that translates between different views while descending from a VHDL design to a respective circuit.

Recall: Y-Diagram

- Tools translate between different views
 - VHDL design to circuit implementation

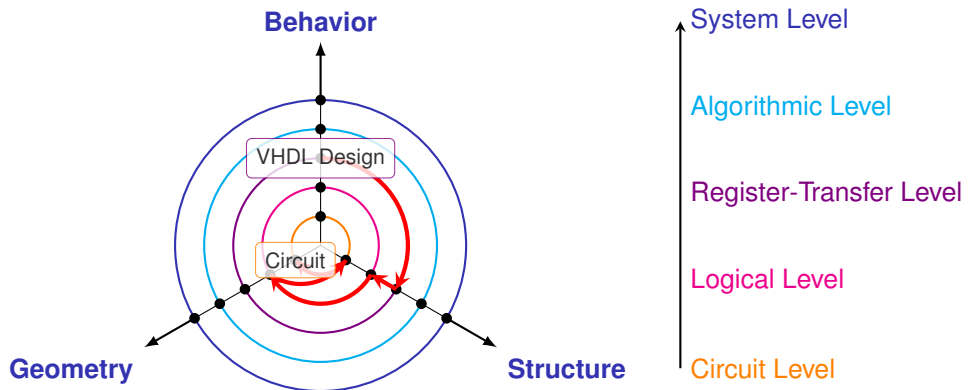




In this lecture we will talk about these tools that do much of the heavy lifting in creating complex circuits. But why should you care about the tools? Well, just as good software developers should understand compilation, good hardware designers should understand the tools that convert models into hardware. Let us now look at the specific steps in the hardware design flow before discussing each in more detail.

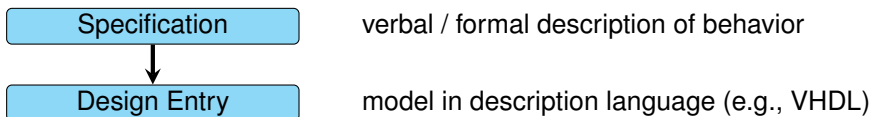
Recall: Y-Diagram

- Tools translate between different views
 - VHDL design to circuit implementation
- What do the tools do? How do translations work?



Each hardware design starts with a specification. This should clearly define the desired behavior, interface and environment. Typically this is plain text or a formal description like a timing diagram. You already used such specifications in this course. The homework task descriptions are specifications of desired designs. This description is the major reference during development and validation. The first task for a designer is to convert the specification into a design entry. The design entry is a description in a hardware description language, based on the specification. In our course this is always VHDL code.

Hardware Design Flow



Logic Synthesis, Place and Route

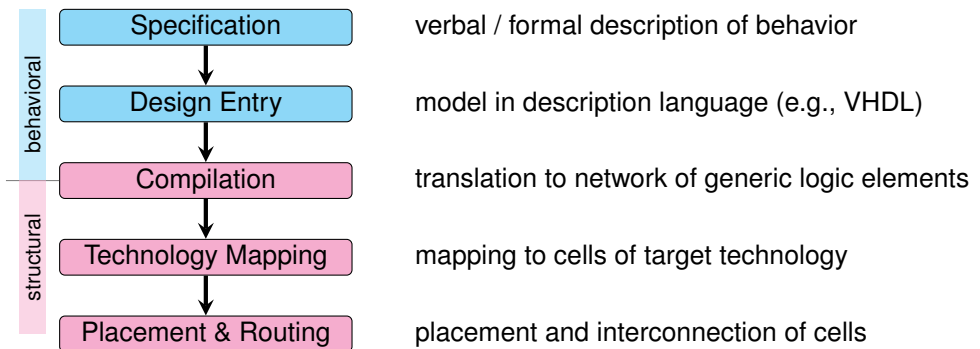
Introduction

Hardware Design Flow



Given a design entry, the next step is to convert it to data structures for subsequent tools. This is the task of the compilation step. The code is checked for syntactical validity, type safety and interface violations. The result is a network of generic logic elements - a translation from behavioral to structural description. Next, after compilation, the so-called technology mapping takes place. Here the network is mapped to a target technology. Finally, the placement and routing step finishes the process. Here logic cells are placed in physical space and connected to each other.

Hardware Design Flow



HWMMod
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

Logic Synthesis, Place and Route

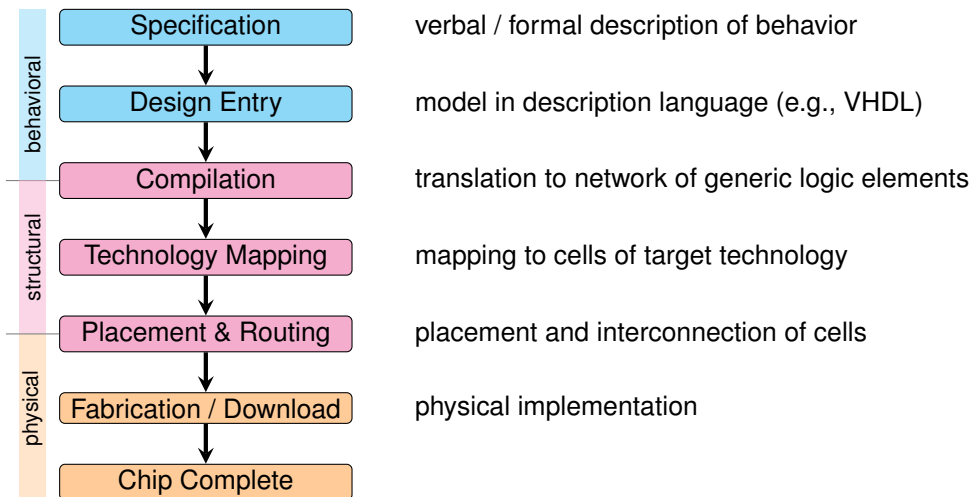
Introduction

Hardware Design Flow



After these steps, the structural description is converted to a physical one. This description is then sufficiently detailed to manufacture a chip or download it to a generic hardware platform. The final result is a chip implementing the initial specification. Note that depending on the target technology, these steps can become quite elaborate with multiple sub-steps and iterations.

Hardware Design Flow



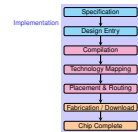
HWMoD
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

Logic Synthesis, Place and Route

Introduction

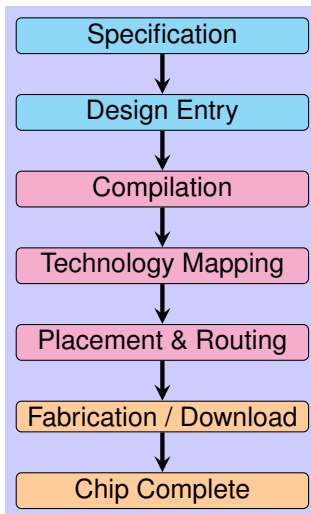
Hardware Design Flow (cont'd)



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)

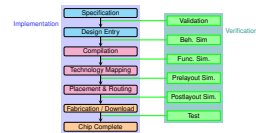
Implementation



HWMoD
WS25

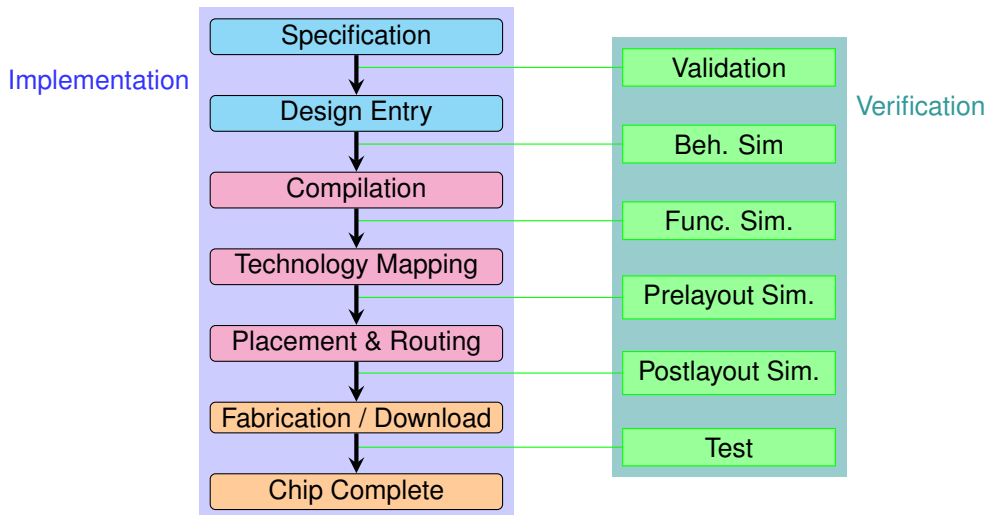
Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

- └ Logic Synthesis, Place and Route
- └ Introduction
- └ **Hardware Design Flow (cont'd)**



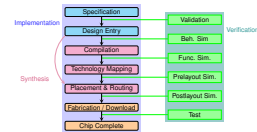
You can find the respective verification steps between consecutive implementation steps on the right of the slide. While all are paramount in practice, we restrict ourselves to behavioral and occasional post-layout simulations in this course. Note that the simulations you performed so far all were behavioral simulations. As verification is a deep topic on its own, there will be a dedicated lecture on it later.

Hardware Design Flow (cont'd)



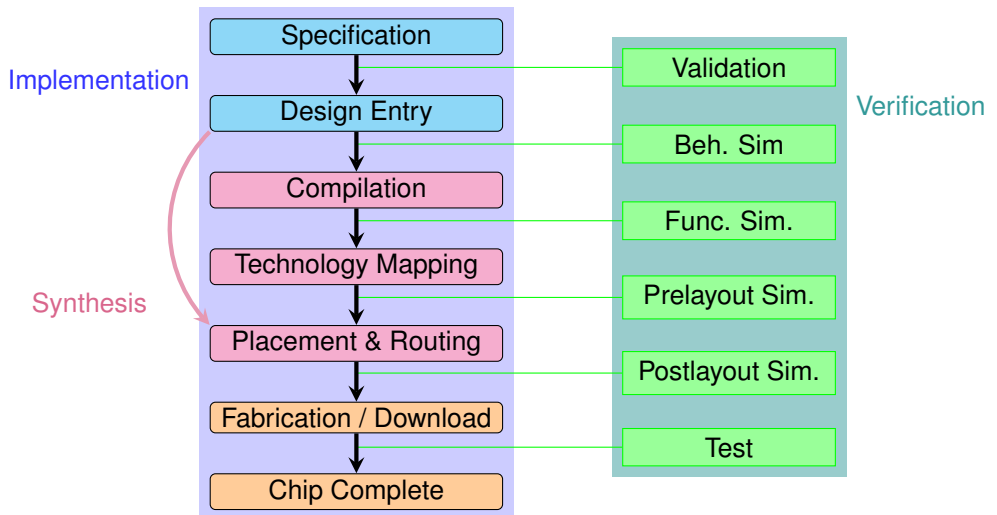
HWMMod
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route



In this lecture we focus on synthesis, which consists of compilation and technology mapping, and placement and routing. Let us now discuss these steps in more detail.

Hardware Design Flow (cont'd)



As 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 a behavioral RTL model for complex designs.

Design Entry

- Description of target circuit (behavioral RTL in VHDL)

While a specification might also be formal, a major difference is that the design entry is easy to read for tools. This makes it suitable for simulations and further implementation steps. Furthermore, a model in a description language serves as documentation of the desired behavior. Recall that this was initially the very purpose of hardware description languages.

Design Entry

- Description of target circuit (behavioral RTL in VHDL)
 - Readable by tools, basis for simulation and documentation

Although you already saw some design entries, let us consider an example we will use throughout this lecture. Our example is a synchronous 8-bit counter with active-low reset, a clock input and an output for the counter value.

Design Entry

- Description of target circuit (behavioral RTL in VHDL)
 - Readable by tools, basis for simulation and documentation
- Example: Synchronous counter

```
1 entity counter is
2   port (
3     clk, res_n : in std_ulogic;
4     number     : out unsigned(7 downto 0)
5   );
6 end entity;
7
8 architecture beh of counter is begin
9   sync : process (clk, res_n) begin
10     if res_n = '0' then
11       number <= (others => '0');
12     elsif rising_edge(clk) then
13       number <= number + 1;
14     end if;
15   end process;
16 end architecture;
```

Recall that for a synchronous circuit with asynchronous reset, our sensitivity list only contains clock and reset. The counter is initially reset to zero using an asynchronous reset.

Design Entry

- Description of target circuit (behavioral RTL in VHDL)
 - Readable by tools, basis for simulation and documentation
- Example: Synchronous counter

```
1 entity counter is
2   port(
3     clk, res_n : in std_ulogic;
4     number    : out unsigned(7 downto 0)
5   );
6 end entity;
7
8 architecture beh of counter is begin
9   sync : process (clk, res_n) begin
10     if res_n = '0' then
11       number <= (others => '0');
12     elsif rising_edge(clk) then
13       number <= number + 1;
14     end if;
15   end process;
16 end architecture;
```

```
■ Description of target circuit (behavioral RTL in VHDL)
■ Readable by tools, basis for simulation and documentation
■ Example: Synchronous counter

entity counter is
  port (
    clk, res_n : in std_logic;
    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;
```

At each rising clock edge, the counter value increases by 1. What happens when the counter reaches its maximum value and is incremented further? As the counter type is `unsigned`, it will simply wrap-around on overflow.

Design Entry

- Description of target circuit (behavioral RTL in VHDL)
 - Readable by tools, basis for simulation and documentation
- Example: Synchronous counter

```
1 entity counter is
2   port (
3     clk, res_n : in std_logic;
4     number     : out unsigned(7 downto 0)
5   );
6 end entity;
7
8 architecture beh of counter is begin
9   sync : process (clk, res_n) begin
10     if res_n = '0' then
11       number <= (others => '0');
12     elsif rising_edge(clk) then
13       number <= number + 1;
14     end if;
15   end process;
16 end architecture;
```

After we have a design entry, the compilation step takes place. This step converts the design entry into a network of generic logic elements. But what does this mean? Let us first discuss the *network* part. Circuits are just components connected via wires, so they can be described as graphs. Components become nodes, wires become edges. Hence, the compilation step results in a data structure containing a graph corresponding to our circuit.

Compilation

- Design entry converted to **technology-agnostic netlist**

HWMoD
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

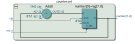
This data structure is typically called a *netlist*.

Compilation

- Design entry converted to **technology-agnostic netlist**
- Netlist: circuit as graph \leftrightarrow RTL schematic

HWMoD
WS25

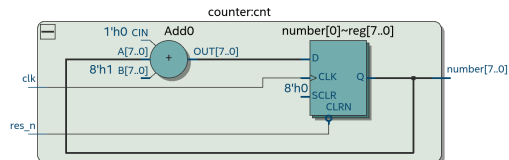
Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

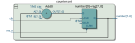


On the slide you see the resulting network when compiling our counter example. Not surprising: it consists of an eight-bit adder and eight flip-flops with feedback to the adder.

Compilation

- Design entry converted to **technology-agnostic netlist**
- Netlist: circuit as graph \leftrightarrow RTL schematic

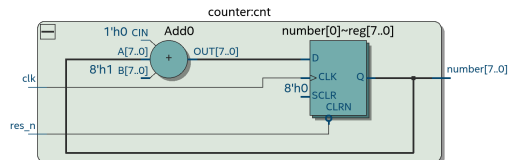


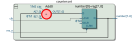


What about the *generic* logic elements? Well, the compilation converts a behavioral description to a structural one. It is completely oblivious to any specific technology and uses basic circuit elements that do not depend on a technology. Examples are multiplexers, basic gates, adders and flip-flops - so fundamental that any target technology must be able to create them.

Compilation

- Design entry converted to **technology-agnostic netlist**
- Netlist: circuit as graph \Leftrightarrow RTL schematic
- Generic components (MUX, adder, basic gates)

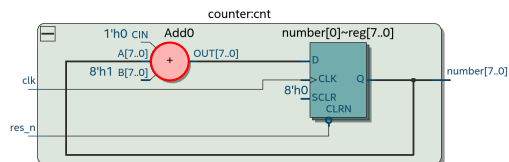


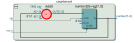


For example, the adder in our example is simply shown as a generic addition block. This is not a valid circuit description, but the tool knows the adder interface and that the target technology will provide it.

Compilation

- Design entry converted to **technology-agnostic netlist**
- Netlist: circuit as graph \leftrightarrow RTL schematic
- Generic components (MUX, adder, basic gates)
 - Tools determine interfaces and check connections

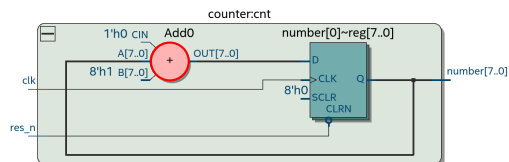




Even basic elements like the flip-flops might not exist in the target technology. For example, flip-flops with asynchronous reset might not be available, although our model requires this. Dealing with such issues is the job of technology mapping.

Compilation

- Design entry converted to **technology-agnostic netlist**
- Netlist: circuit as graph \leftrightarrow RTL schematic
- Generic components (MUX, adder, basic gates)
 - Tools determine interfaces and check connections
 - Might not exist in target technology



The next step after compilation is technology mapping. Here the generic circuit elements are mapped to the target technology.

Technology Mapping

HWMoD
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

- Map generic elements to target technology

Such target technologies can be application-specific integrated circuits, programmable array logic, gate arrays or FPGAs. Do not worry if you have not heard about these - they will be covered in another course. In this course we only consider FPGAs, which we will introduce in this lecture.

Technology Mapping

HWMoD
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

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

The way mapping works is that each technology comes with a library of available cells. Such a library can consist of gates, sequential elements like flip-flops, or more specialized elements like memories or latches. The tool responsible for mapping then tries to implement each generic logic element using the available ones. Note that after this step the target technology is set and cannot be changed in subsequent steps.

Technology Mapping

HWMoD
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

- Map generic elements to target technology
 - E.g.: ASIC, PAL, Gate-Array, **FPGA**
- Requires *target library* of available cells
 - Gates, sequential elements, etc.
- Target technology is now fixed

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

Introduction to FPGAs

HWMoD
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map
FPGAs
Place & Route

■ **Field Programmable Gate Array**

But what does this actually mean?

Introduction to FPGAs

HWMoD
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

- **Field Programmable Gate Array ?!**

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

HWMMod
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

- **Field Programmable Gate Array ?!**
- Generic circuit, configurable for target circuit

The answer: programmable logic cells and 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

HWMoD
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

- **Field Programmable Gate Array ?!**
- Generic circuit, configurable for target circuit
- Programmable logic cells and interconnect

Let us start with advantages of generic over full custom chips. First, generic chips can be manufactured in larger volumes, addressing a larger audience. This makes them comparably cheap for hardware designs.

Introduction to FPGAs

HWMoD
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

- **Field Programmable Gate Array ?!**
- Generic circuit, configurable for target circuit
- Programmable logic cells and interconnect
 - + Large volumes ⇒ comparably cheap

Furthermore, if we recall the Y-diagram, some steps like manufacturing are already done. This makes designing circuits for FPGAs simpler, faster and reduces the entry barrier. While having a manufactured chip has benefits, it also has drawbacks compared to full custom designs.

Introduction to FPGAs

HWMoD
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

- **Field Programmable Gate Array ?!**
- Generic circuit, configurable for target circuit
- Programmable logic cells and interconnect
 - + Large volumes ⇒ comparably cheap
 - + Already fabricated ⇒ simpler, faster design

In particular, the amount of possible optimizations is limited.

Introduction to FPGAs

HWMod
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

- **Field Programmable Gate Array ?!**
- Generic circuit, configurable for target circuit
- Programmable logic cells and interconnect
 - + Large volumes ⇒ comparably cheap
 - + Already fabricated ⇒ simpler, faster design
 - Already fabricated ⇒ less optimizations

Furthermore, a chip made for arbitrary designs is inevitably subject to overhead. To revisit the Lego example: Assume you get equal red and blue blocks because most people need both. However, if you only need red blocks, the blue ones are wasted, consuming space and costing money you could use and spend for red blocks.

Introduction to FPGAs

HWMod
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map
FPGAs
Place & Route

- **Field Programmable Gate Array ?!**
- Generic circuit, configurable for target circuit
- Programmable logic cells and interconnect
 - + Large volumes ⇒ comparably cheap
 - + Already fabricated ⇒ simpler, faster design
 - Already fabricated ⇒ less optimizations
 - Generic design ⇒ inevitable overhead

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

Introduction to FPGAs

HWMoD
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

- **Field Programmable Gate Array ?!**
- Generic circuit, configurable for target circuit
- Programmable logic cells and interconnect
 - + Large volumes ⇒ comparably cheap
 - + Already fabricated ⇒ simpler, faster design
 - Already fabricated ⇒ less optimizations
 - Generic design ⇒ inevitable overhead
- How can such circuits be built?

0	0	1
0	1	1
1	0	0
1	1	0

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 $\neg a$.

Look-Up Tables (LUTs)

- How to implement arbitrary logic functions?
 - Map inputs to outputs \Rightarrow truth table

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

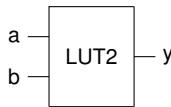


One way to build programmable logic is doing exactly the same. So-called *look-up tables* take arbitrary input vectors and return the respective output. The slide shows a two-input look-up table as a blackbox. This element can implement all 16 possible two-input functions. Let us look at how this can be implemented in hardware.

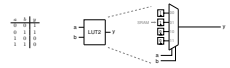
Look-Up Tables (LUTs)

- How to implement arbitrary logic functions?
 - Map inputs to outputs ⇒ truth table
 - ⇒ Same in FPGAs: *Look-up Tables* (LUTs)

<i>a</i>	<i>b</i>	<i>y</i>
0	0	1
0	1	1
1	0	0
1	1	0



- How to implement arbitrary logic functions?
 - Map inputs to outputs ⇒ truth table
 - Store in FPGAs: Look-up Tables (LUTs)
 - Store in SRAM cells, use inputs to select

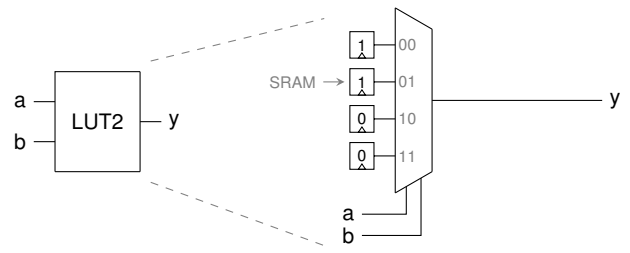


A straightforward implementation is based on SRAM. For our purpose, consider this memory as consisting of single bit cells that are being programmed once after powering on, and then provide values until their power is cut. We can use such memory cells to store the truth table of our desired logic function. The outputs of these cells are then connected to a multiplexer controlled by the function's input signals, which forwards the respective truth table value.

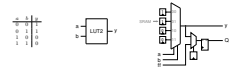
Look-Up Tables (LUTs)

- How to implement arbitrary logic functions?
 - Map inputs to outputs ⇒ truth table
 - ⇒ Same in FPGAs: *Look-up Tables (LUTs)*
 - Store in SRAM cells, use inputs to select

<i>a</i>	<i>b</i>	<i>y</i>
0	0	1
0	1	1
1	0	0
1	1	0



- How to implement arbitrary logic functions?
 - Map inputs to outputs \Rightarrow truth table
 - Store in FPGAs: Look-up Tables (LUTs)
 - Store in SRAM cells, use inputs to select
- For sequential logic, add flip-flops

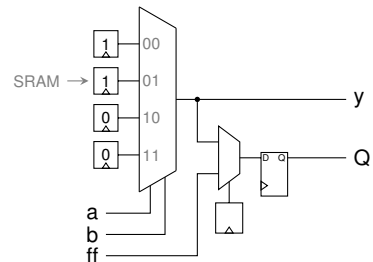
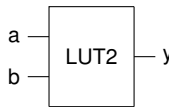


Since we want both combinational and sequential logic, we equip our block with an optional flip-flop. Depending on another SRAM cell, we can use the flip-flop on its own, or to sample the truth table output.

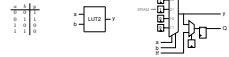
Look-Up Tables (LUTs)

- How to implement arbitrary logic functions?
 - Map inputs to outputs \Rightarrow truth table
 - \Rightarrow Same in FPGAs: *Look-up Tables* (LUTs)
 - Store in SRAM cells, use inputs to select
- For sequential logic, add flip-flops

<i>a</i>	<i>b</i>	<i>y</i>
0	0	1
0	1	1
1	0	0
1	1	0



- How to implement arbitrary logic functions?
 - Map inputs to outputs ⇒ truth table
 - Store in FPGAs: Look-up Tables (LUTs)
 - Store in SRAM cells, use inputs to select
- For sequential logic, add flip-flops
- Logic Element (LE)

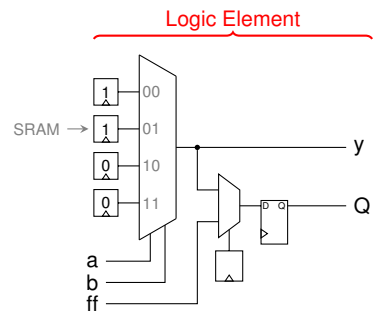
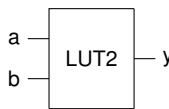


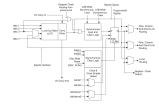
And with this we have our simple programmable logic circuit. All we need to do is programming the SRAM cells to get an arbitrary two-input logic function with an optional flip-flop. We call blocks like this logic elements.

Look-Up Tables (LUTs)

- How to implement arbitrary logic functions?
 - Map inputs to outputs ⇒ truth table
 - ⇒ Same in FPGAs: *Look-up Tables* (LUTs)
 - Store in SRAM cells, use inputs to select
- For sequential logic, add flip-flops
- Logic Element* (LE)

<i>a</i>	<i>b</i>	<i>y</i>
0	0	1
0	1	1
1	0	0
1	1	0

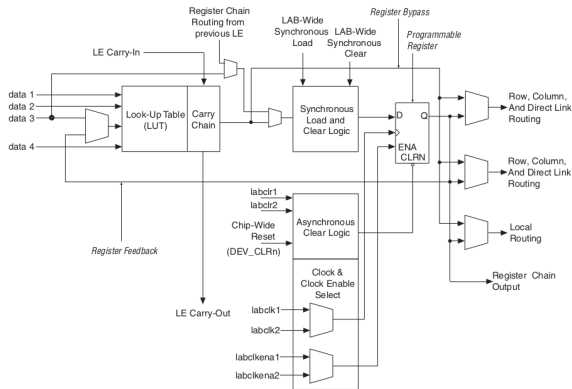


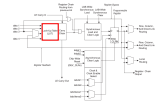


While our simple logic element is illustrative and generally not too far from reality, the logic elements in real FPGAs are usually more complex and also powerful. The slide shows a logic element schematic from an actual FPGA.

LEs in Real FPGAs

■ Real LEs are more complex and powerful

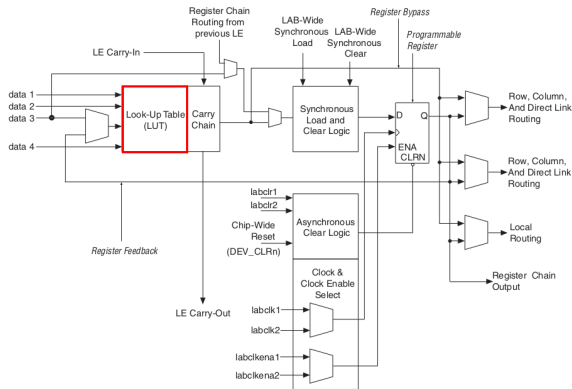


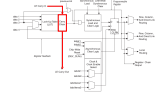


We can observe that the look-up table has four inputs instead of two, allowing significantly more logic functions to be implemented.

LEs in Real FPGAs

- Real LEs are more complex and powerful
- Bigger LUTs (3-6 inputs)

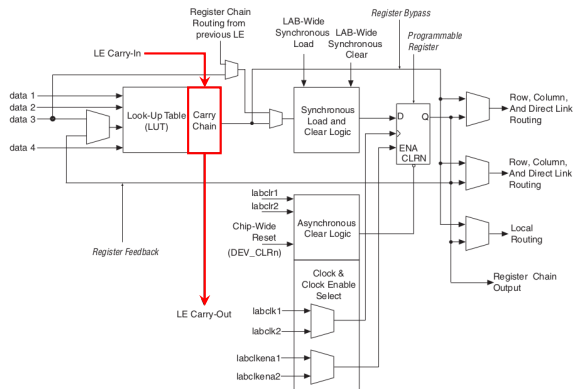




Furthermore, it is equipped with dedicated logic for carry chains, allowing efficient adder implementations. Addition is required often, and it is typical practice to make the common case fast.

LEs in Real FPGAs

- Real LEs are more complex and powerful
- Bigger LUTs (3-6 inputs), dedicated carry chains

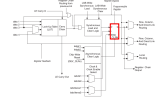


Logic Synthesis, Place and Route

FPGAs

LEs in Real FPGAs

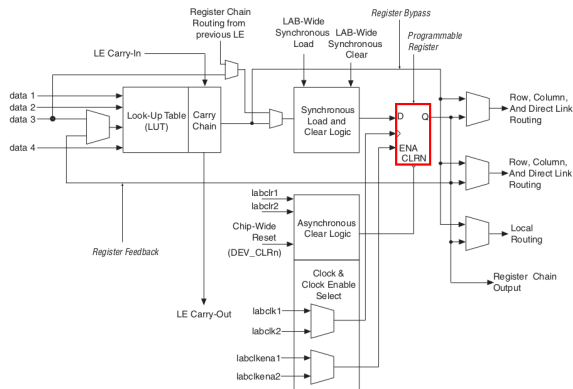
- Real LEs are more complex and powerful
- Bigger LUTs (3-6 inputs), dedicated carry chains
- Clear/set logic, sometimes multiple FFs



The flip-flop has additional logic, allowing more input sources, setting or clearing content, and even disabling it. Some FPGAs feature logic elements with more than one flip-flop, though not the one shown here.

LEs in Real FPGAs

- Real LEs are more complex and powerful
- Bigger LUTs (3-6 inputs), dedicated carry chains
- Clear/set logic, sometimes multiple FFs



HWMoD
WS25

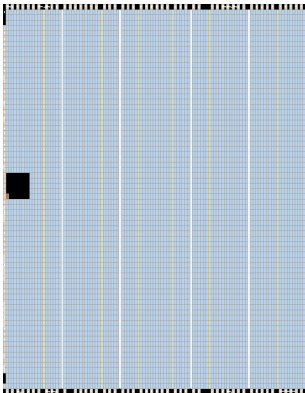
Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

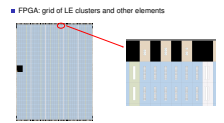


In its majority, an FPGA consists of such logic elements. Multiple logic elements are gathered in clusters that are arranged in a grid. The image shows an actual FPGA chip, where everything blue consists of logic elements. Let us zoom into this grid.

FPGA Structure

- FPGA: grid of LE clusters and other elements

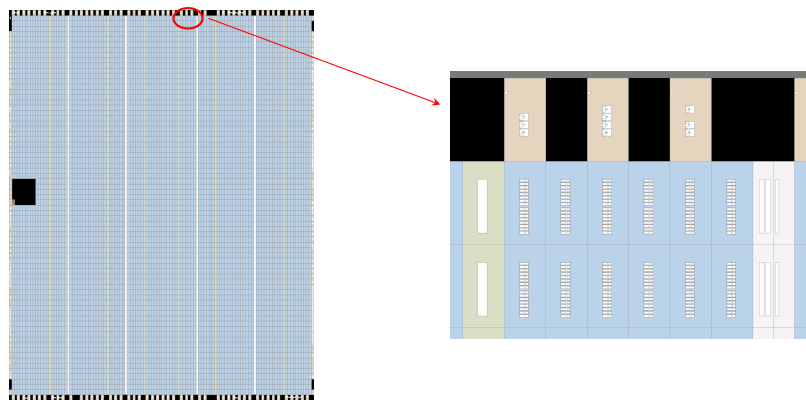


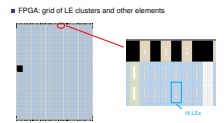


This view shows the grid of rectangular cells. Note how the chip has multiple cell types, highlighted by different colors.

FPGA Structure

FPGA: grid of LE clusters and other elements

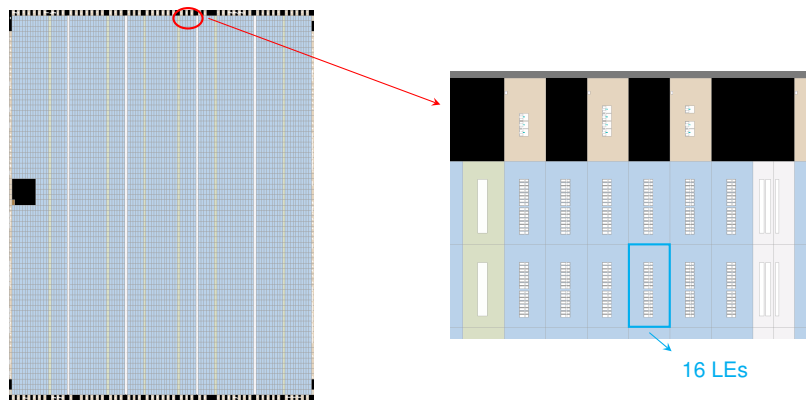


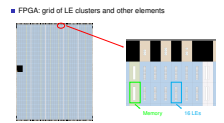


The majority are clusters of logic elements. In this up-close view we see that for this FPGA, logic elements form clusters of size sixteen.

FPGA Structure

FPGA: grid of LE clusters and other elements

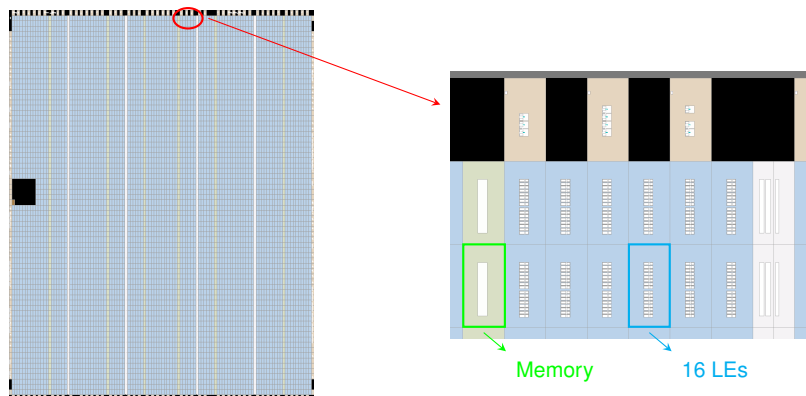


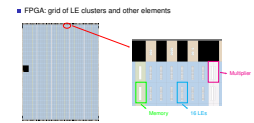


Next, we have dedicated memory cells. Many designs require memory to store data. While building memory from logic elements is possible, it is very costly in chip area and performance. FPGA vendors equip chips with embedded memory blocks that offer denser and faster memory.

FPGA Structure

■ FPGA: grid of LE clusters and other elements

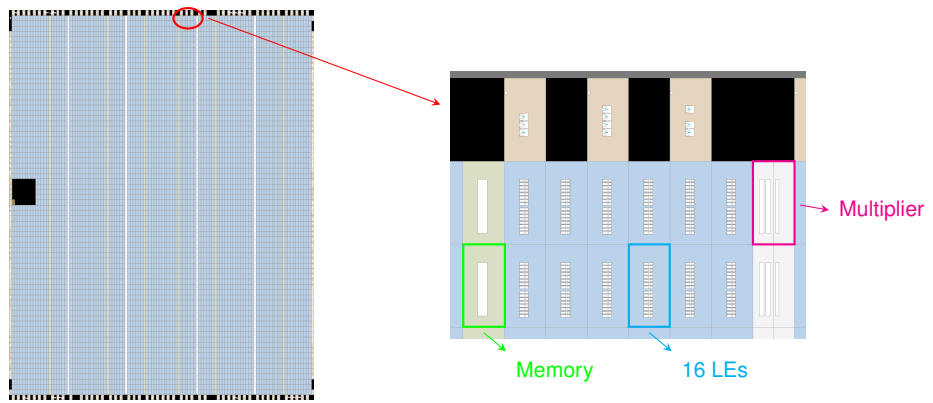


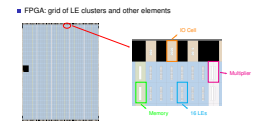


Furthermore, just like dedicated adder logic, many FPGAs have dedicated multipliers. This is more area efficient and faster than building multipliers from logic elements.

FPGA Structure

FPGA: grid of LE clusters and other elements

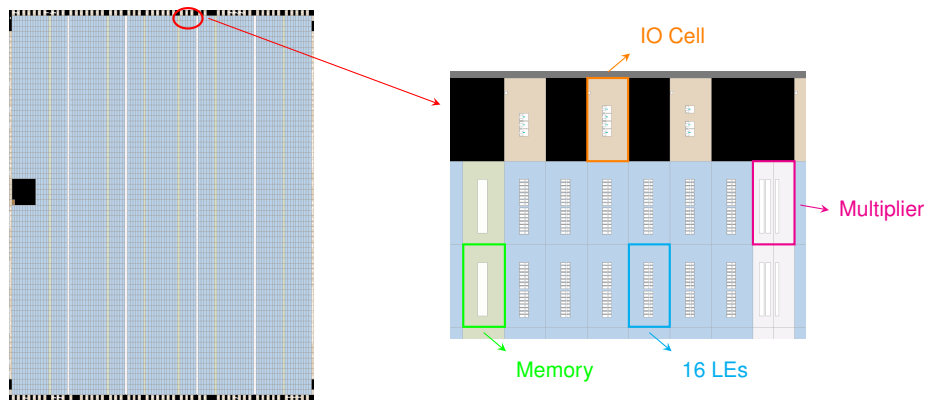




To interface with the outside world, FPGAs have IO cells providing different input and output functionality. Finally, black areas mark parts without usable logic.

FPGA Structure

■ FPGA: grid of LE clusters and other elements



While programmable logic cells are paramount, they are only one part. We also need a means to compose networks of these cells to implement our target circuits.

FPGA Programmable Interconnect

- How to connect LEs to form complex circuits?

HWMoD
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route



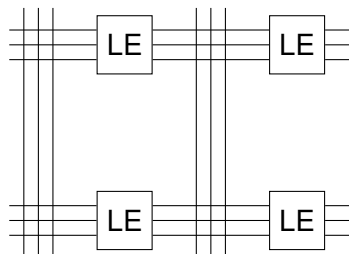
FPGAs connect logic elements using programmable interconnect. In a nutshell, all logic elements connect to a switching matrix where wires of adjacent elements and transport wires intersect. By programming which intersections are tapped, we configure where inputs come from and where outputs go.

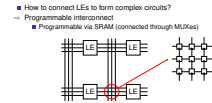
FPGA Programmable Interconnect

HWMoD
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

- How to connect LEs to form complex circuits?
- ⇒ Programmable interconnect

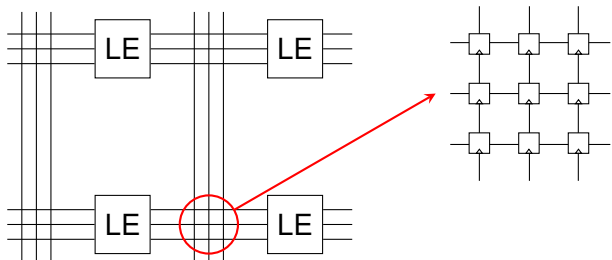




If we zoom into such a switching matrix, we see that programmable intersections are implemented using SRAM cells. The switching itself uses multiplexers. However, we will not delve deeper into this in this lecture.

FPGA Programmable Interconnect

- How to connect LEs to form complex circuits?
⇒ Programmable interconnect
 - Programmable via SRAM (connected through MUXes)





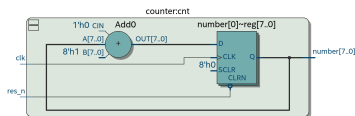
Let us return to the technology mapping step of our synchronous counter to an SRAM-based FPGA. Recall the goal: map the generic circuit to the target technology. For an FPGA this means mapping to a network of logic elements.

Technology Mapping to an FPGA

- Generic elements implemented using LEs

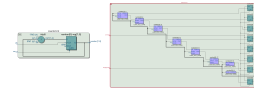
HWMoD
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route



- └ Logic Synthesis, Place and Route
- └└ FPGAs
- └└└ Technology Mapping to an FPGA

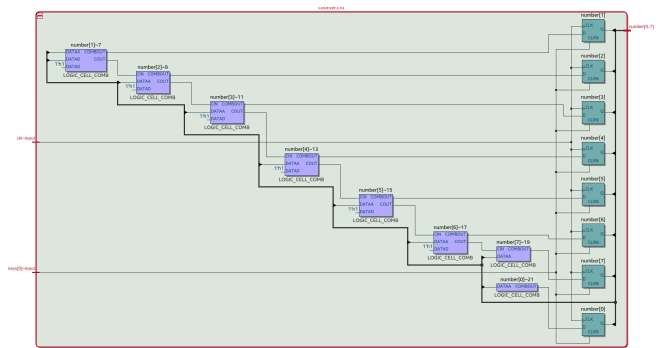
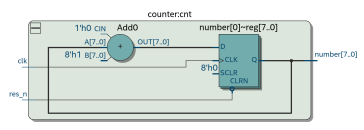
- Generic elements implemented using LEs
- Each LE implements part of the logic



On the right, the result of such a mapping is shown. The teal boxes are the eight flip-flops our 8-bit counter requires. These flip-flops are fed by a chain of logic. Let us look closer at one of these logic blocks.

Technology Mapping to an FPGA

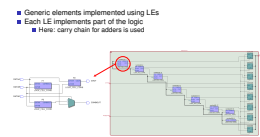
- Generic elements implemented using LEs
- Each LE implements part of the logic



HWMoD
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

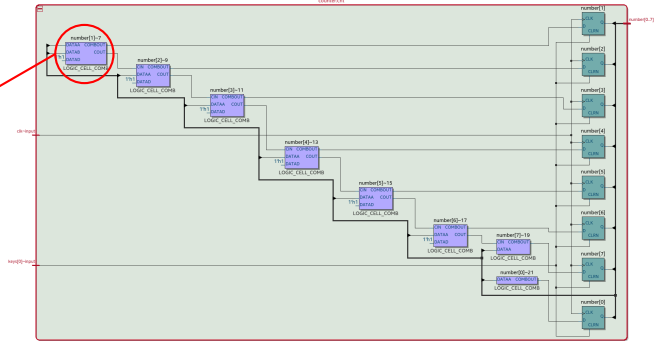
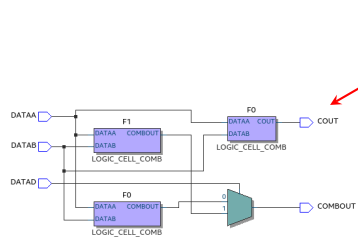
Logic Synthesis, Place and Route
 └─ FPGAs
 └─ Technology Mapping to an FPGA



Zooming in reveals that the dedicated carry chain for adders is used instead of the lookup table. This is clearly shown by the carry out output named COUT. This makes sense for a counter - the tool picked the most efficient implementation available.

Technology Mapping to an FPGA

- Generic elements implemented using LEs
- Each LE implements part of the logic
 - Here: carry chain for adders is used



HWMMod
 WS25
 Syn. & PR
 Introduction
 Design Entry
 Compilation
 Tech. Map.
 FPGAs
 Place & Route

- └ Logic Synthesis, Place and Route
 - └ Placement & Routing
 - └ **Placement & Routing**

The final step in obtaining a physical description is placement and routing. These are actually two steps, but due to their close interplay we cover them together.

Placement & Routing

HWMoD
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

First, let us consider placement. Its task is placing elements from the previously obtained network to the available physical area. For an FPGA this means assigning each logic element, memory cell, multiplier and IO cell to a grid cell. While this might sound easy, it is highly non-trivial.

Placement & Routing

- Placement: Choose position of LEs

HWMoD
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

For complex designs, physical constraints can make it hard to place connected cells next to each other.

Placement & Routing

- Placement: Choose position of LEs
 - Constrained by physical availability

HWMoD
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

Furthermore, placement has big impact on wires and thus overall timing. To meet timing constraints, the placer must minimize delay without knowing the wiring yet.

Placement & Routing

- Placement: Choose position of LEs
 - Constrained by physical availability
 - Minimize unknown interconnect

Determining this wiring is the job of routing. 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.

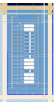
Placement & Routing

- Placement: Choose position of LEs
 - Constrained by physical availability
 - Minimize unknown interconnect
- Routing: Connect placed cells
 - Constrained by available interconnect
 - Goal: Meet timing constraints

In general, placing and routing is highly non-trivial. The steps are usually executed iteratively, where the placer gets timing information and incrementally improves. Both steps are typically guided by heuristics.

Placement & Routing

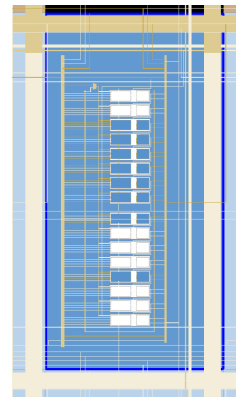
- Placement: Choose position of LEs
 - Constrained by physical availability
 - Minimize unknown interconnect
- Routing: Connect placed cells
 - Constrained by available interconnect
 - Goal: Meet timing constraints
- Paramount for timing, non-trivial
 - Multiple iterations
 - Driven by heuristics



On the right, the final result of placing and routing the example counter is shown. The eight logic elements are placed within a single cluster. Brown lines show how cells are connected to each other and to IO cells.

Placement & Routing

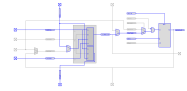
- Placement: Choose position of LEs
 - Constrained by physical availability
 - Minimize unknown interconnect
- Routing: Connect placed cells
 - Constrained by available interconnect
 - Goal: Meet timing constraints
- Paramount for timing, non-trivial
 - Multiple iterations
 - Driven by heuristics



For an FPGA, the final result of the design flow is a so-called *bitstream*. This encodes the content of all the chip's SRAM cells. By programming the SRAM cells using this bitstream, all logic elements and interconnect are configured to implement the target circuit.

FPGA Bitstream

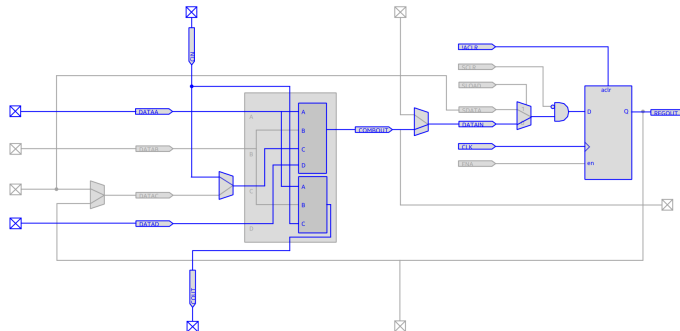
- For FPGAs: result is a *bitstream*
 - Content of all SRAM cells
 - Configures LEs and interconnect



The figure at the bottom illustrates such a configuration for a single logic element of our counter. We see the dedicated carry chain being used and fed into the flip-flop. This flip-flop is connected to asynchronous reset and clock, just as in our VHDL code.

FPGA Bitstream

- For FPGAs: result is a *bitstream*
 - 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.

HWMoD
WS25

Syn. & PR
Introduction
Design Entry
Compilation
Tech. Map.
FPGAs
Place & Route

Lecture Complete!