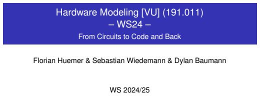


Hardware Modeling [VU] (191.011)
– WS24 –
From Circuits to Code and Back

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

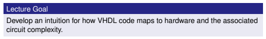
This lecture aims at giving you a more thorough understanding of how certain VHDL code structures map to hardware. We will not introduce any new language concepts and instead focus on practically applying what we already know.

Hardware Modeling [VU] (191.011) – WS24 –

From Circuits to Code and Back

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

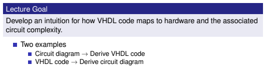

Lecture Goal
Develop an intuition for how VHDL code maps to hardware and the associated circuit complexity.

Similar to how a system-level programmer must have a good understanding of how the written program maps to the available machine instructions on a given architecture, in order to produce efficient software, you – as a hardware developer – should have a similar intuition about the mapping between VHDL code and circuits. Hence, with this lecture we want to help you to develop such an intuition, as we believe that this is a key skill in becoming a great hardware developer!

Introduction

Lecture Goal

Develop an intuition for how VHDL code maps to hardware and the associated circuit complexity.

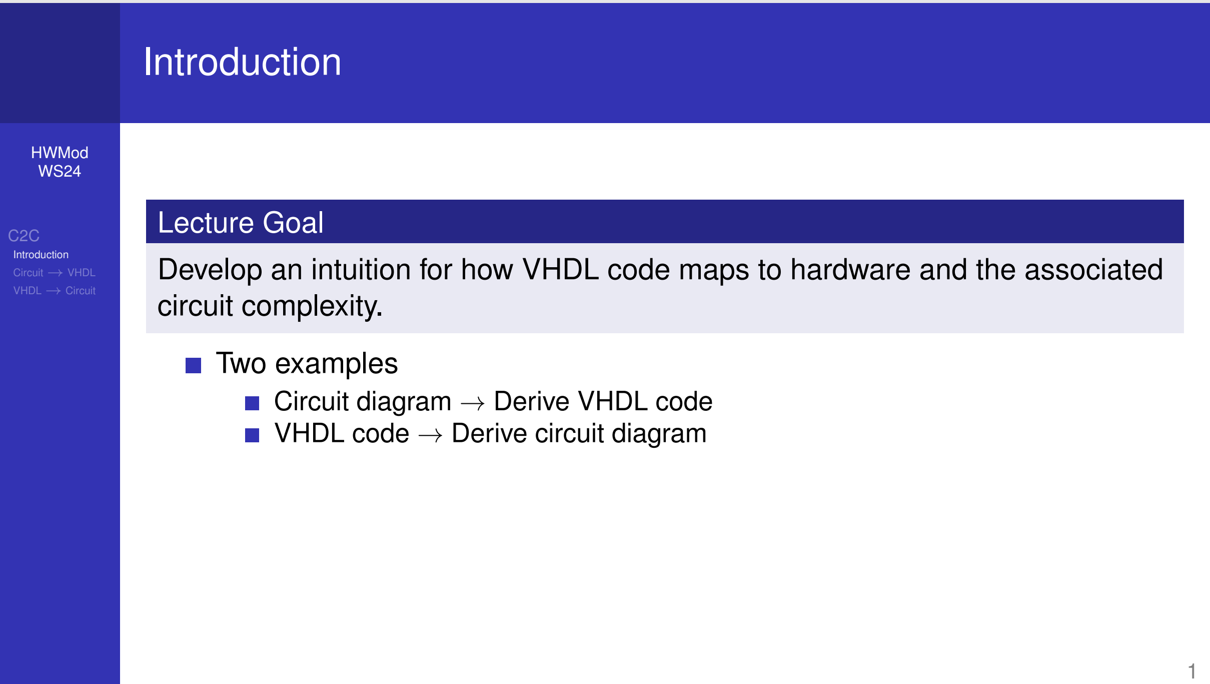


Lecture Goal

Develop an intuition for how VHDL code maps to hardware and the associated circuit complexity.

- Two examples
  - Circuit diagram → Derive VHDL code
  - VHDL code → Derive circuit diagram

To achieve this goal we present and work through two circuit modeling examples. In the first example we take a circuit diagram and derive a VHDL entity/architecture pair that models the behavior of the given circuit. For the second example, we perform the reverse operation, taking a simple VHDL design and deriving a circuit diagram from it. We will thus effectively perform the task of the synthesis tool ourselves.

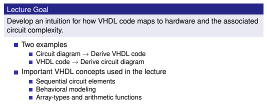


Introduction

Lecture Goal

Develop an intuition for how VHDL code maps to hardware and the associated circuit complexity.

- Two examples
  - Circuit diagram → Derive VHDL code
  - VHDL code → Derive circuit diagram


Lecture Goal
Develop an intuition for how VHDL code maps to hardware and the associated circuit complexity.
- Two examples
  - Circuit diagram → Derive VHDL code
  - VHDL code → Derive circuit diagram
- Important VHDL concepts used in the lecture
  - Sequential circuit elements
  - Behavioral modeling
  - Array-types and arithmetic functions

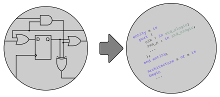
Note that, besides other topics, this lecture heavily builds upon the videos about sequential circuit elements and behavioral modeling.

Introduction

Lecture Goal

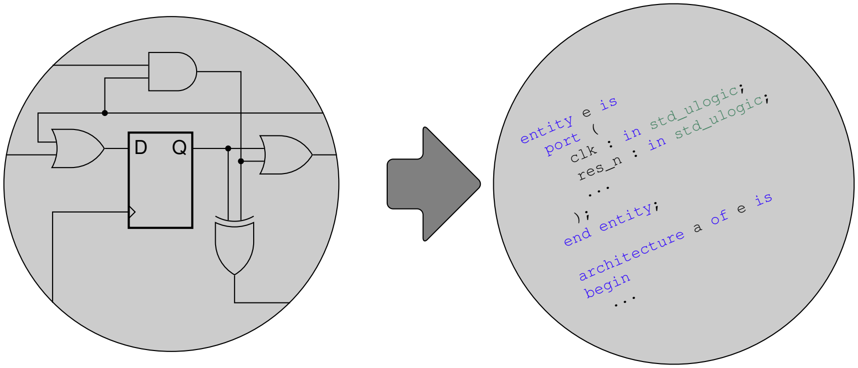
Develop an intuition for how VHDL code maps to hardware and the associated circuit complexity.

- Two examples
  - Circuit diagram → Derive VHDL code
  - VHDL code → Derive circuit diagram
- Important VHDL concepts used in the lecture
  - Sequential circuit elements
  - Behavioral modeling
  - Array-types and arithmetic functions

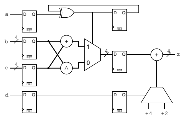


Let's start with the first example, where we take a circuit diagram and derive an appropriate VHDL design from it.

Example 1: Circuit → VHDL Code

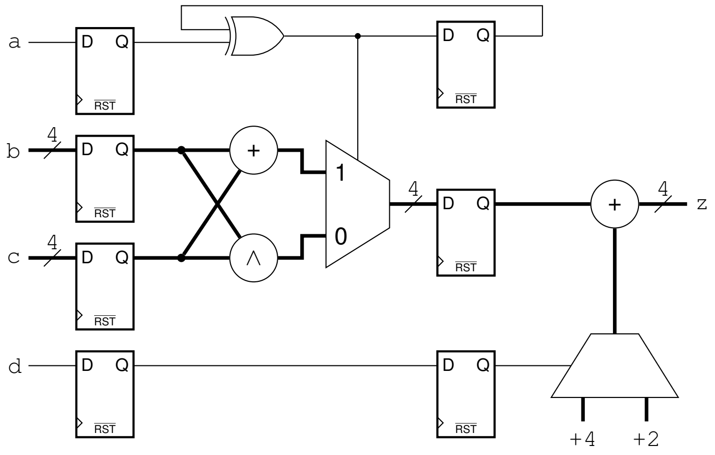


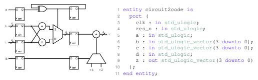
```
entity e is
  port (
    clk : in std_ulogic;
    res_n : in std_ulogic;
    ...
  );
end entity;

architecture a of e is
begin
  ...
```



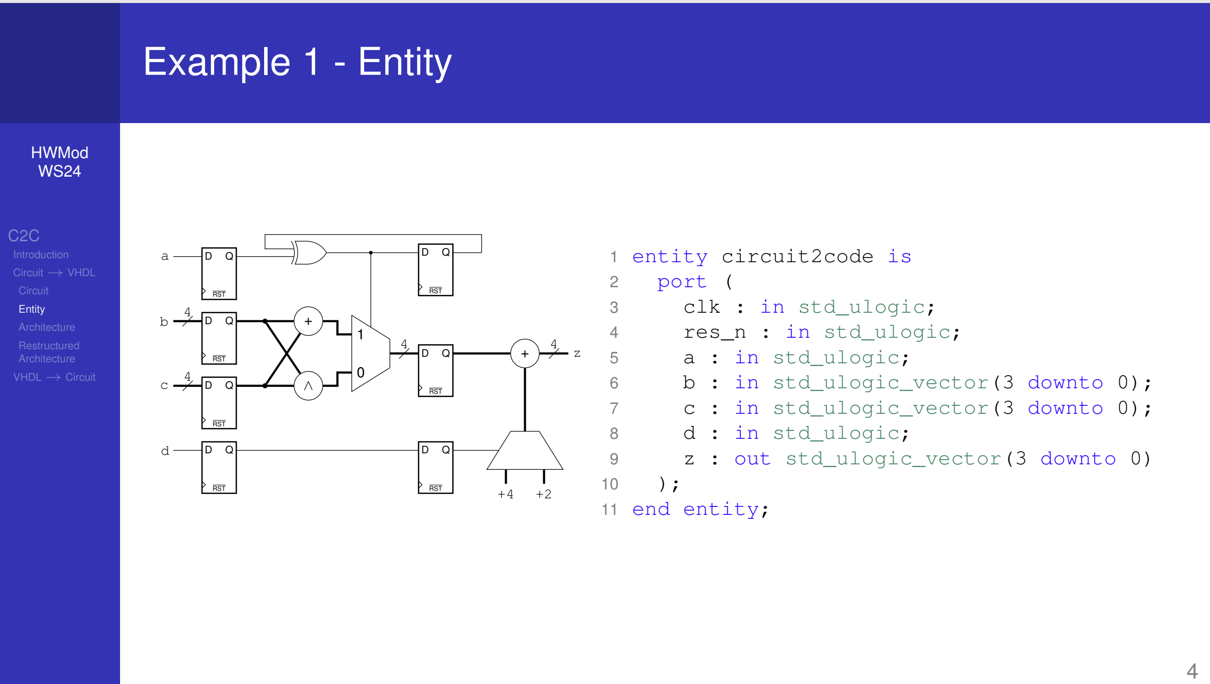
Consider the circuit shown on the slide. Before we start, note that its actual function or purpose is not important for this lecture – in fact we did not have any particular application in mind when designing this circuit. You should already know most of the used circuit symbols. The circle-shaped component with the plus sign represents an adder, while the symbol with the logical AND denotes a bit-wise AND operation. The inputs `a` and `d` are single-bit signals while `b` and `c` are four bit wide as is the single output `z`. Since it contains registers the circuit also has a clock and reset input. However, these signals have been omitted from the drawing, as not to clutter the figure. Thus, for the sake of completeness, imagine all registers being connected to a common clock and an active-low reset signal.

Example 1: Circuit Diagram

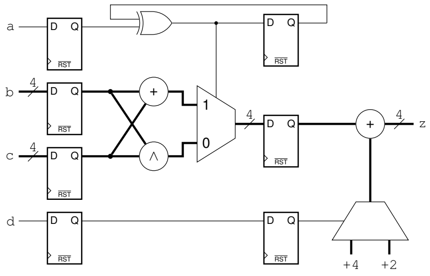




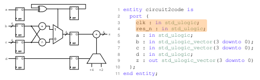
The first step in deriving a suitable VHDL design is to derive its interface, that is, its entity declaration.



Example 1 - Entity

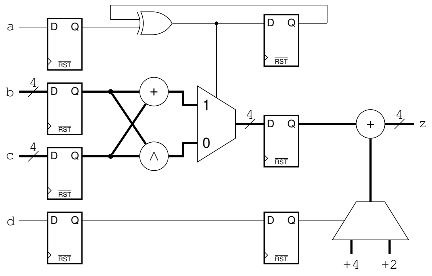


```
entity circuit2code is
  port (
    clk : in std_ulogic;
    res_n : in std_ulogic;
    a : in std_ulogic;
    b : in std_ulogic_vector(3 downto 0);
    c : in std_ulogic_vector(3 downto 0);
    d : in std_ulogic;
    z : out std_ulogic_vector(3 downto 0)
  );
end entity;
```

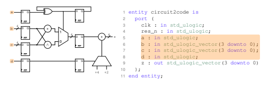


First, we add a clock and reset input, as we will need them to implement the registers. As always, we use the `std_ulogic` type for both of these signals. Note that the symbols used for the flip-flops in the circuit indicate an active-low reset.

Example 1 - Entity

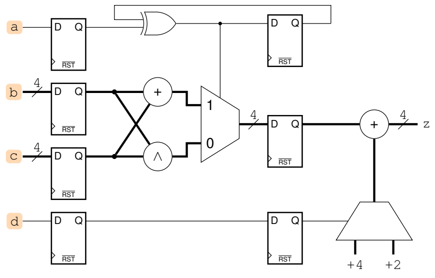


```vhdl
entity circuit2code is
  port (
    clk : in std_ulogic;
    res_n : in std_ulogic;
    a : in std_ulogic;
    b : in std_ulogic_vector(3 downto 0);
    c : in std_ulogic_vector(3 downto 0);
    d : in std_ulogic;
    z : out std_ulogic_vector(3 downto 0)
  );
end entity;
```

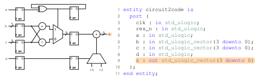


Next, we list all the input signals in the order of their appearance. For the signals `b` and `c` we could also have gone with the signed or the unsigned data-type. However, as the signedness of these input values does not change the way they need to be handled by the circuit, we simply settled on the `std_ulogic_vector` type.

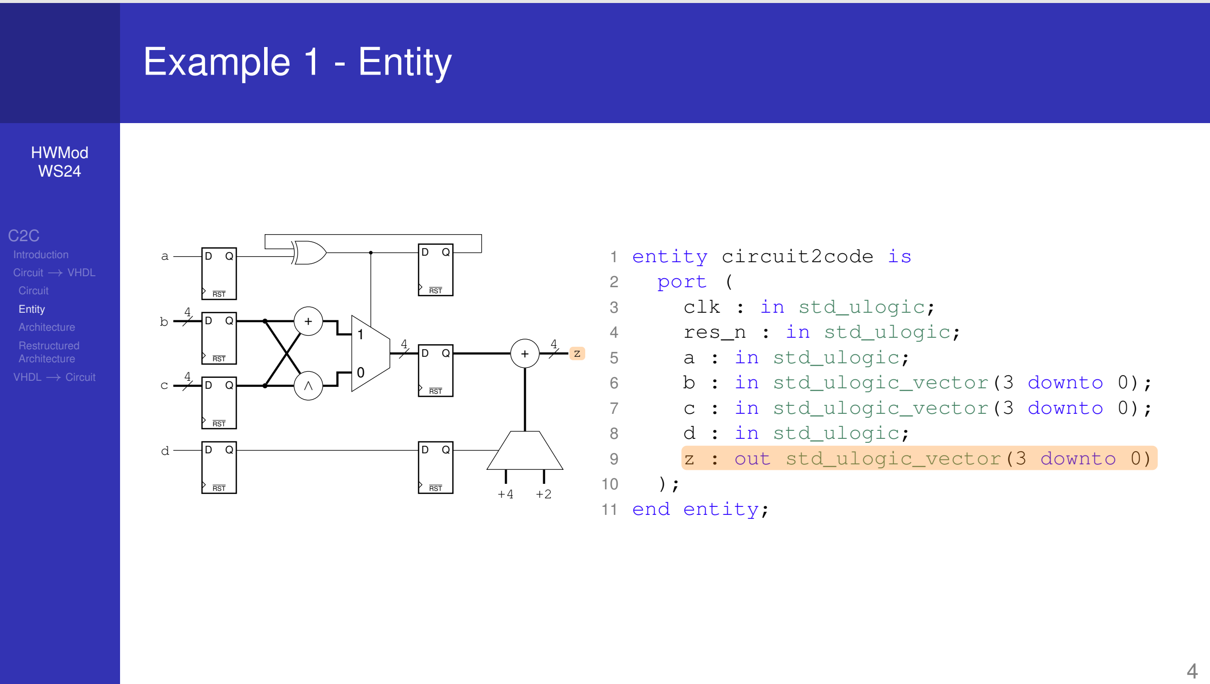
Example 1 - Entity



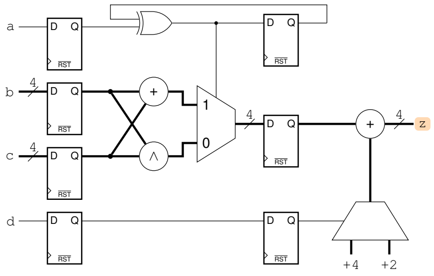
```
entity circuit2code is
  port (
    clk : in std_ulogic;
    res_n : in std_ulogic;
    a : in std_ulogic;
    b : in std_ulogic_vector(3 downto 0);
    c : in std_ulogic_vector(3 downto 0);
    d : in std_ulogic;
    z : out std_ulogic_vector(3 downto 0)
  );
end entity;
```



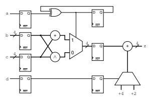
Finally, we add the single, four-bit-wide, output signal `z`.



Example 1 - Entity

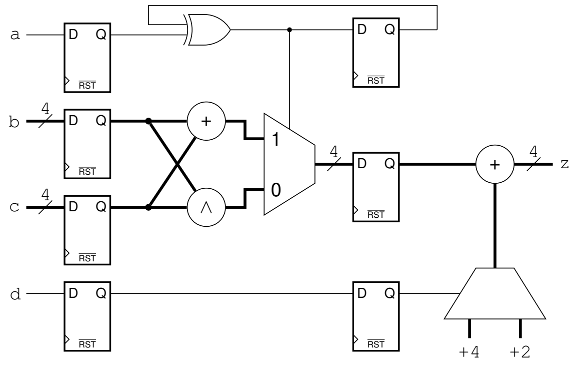


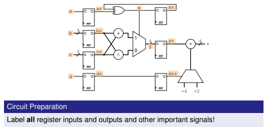
```
entity circuit2code is
  port (
    clk : in std_ulogic;
    res_n : in std_ulogic;
    a : in std_ulogic;
    b : in std_ulogic_vector(3 downto 0);
    c : in std_ulogic_vector(3 downto 0);
    d : in std_ulogic;
    z : out std_ulogic_vector(3 downto 0)
  );
end entity;
```



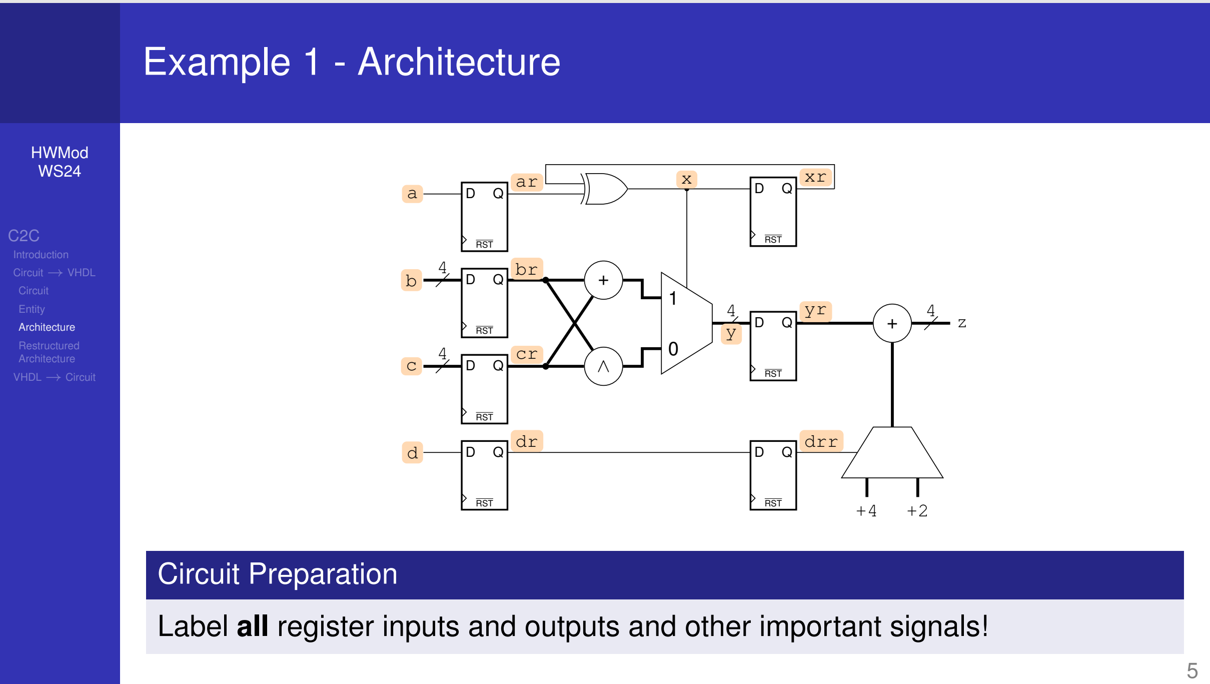
With the entity in place, we can now turn our attention to the architecture, which contains the description of the actual circuit behavior. Before we can start coding, we have to label some of the internal signals of our circuit, such that we can then refer to these signals using appropriate identifiers in the VHDL design. Let us therefore again look at the circuit.

Example 1 - Architecture

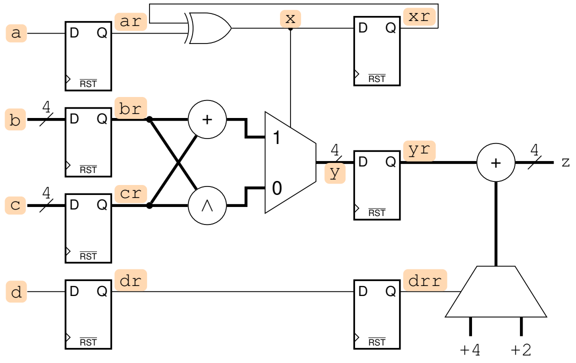




In particular, we make sure that the inputs and outputs of all sequential elements have proper names. Therefore, for all inputs and outputs of such elements, which are not directly connected to an entity port signal, we need to introduce a proper signal name. For this example we use the following naming scheme: The output of a sequential element gets the name of its input extended by the letter `r`, referring to the signal being "registered". Furthermore, depending on the exact way we express the combinational parts of the circuit as VHDL code, it can also make sense to label further signals.

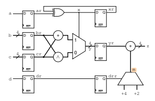


Example 1 - Architecture



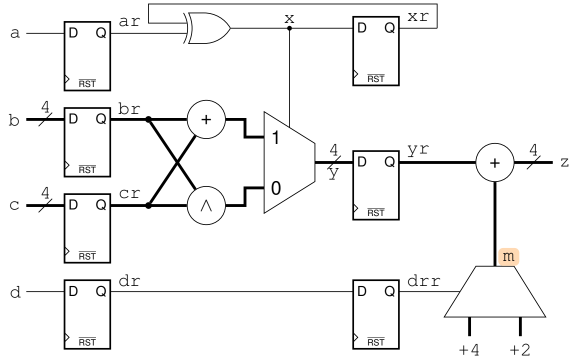
Circuit Preparation

Label **all** register inputs and outputs and other important signals!



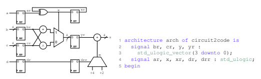
One candidate for such a signal is the output of the multiplexer in the lower right corner, which we simply label `m`.

Example 1 - Architecture



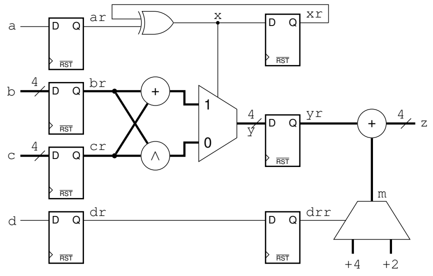
Circuit Preparation

Label **all** register inputs and outputs and other important signals!

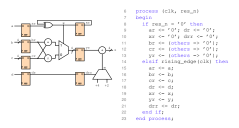


Now we have everything we need to start implementing the architecture. We begin with the declarations for the internal signals, that we have identified and labeled in the previous step. Note that we don't declare a signal for `m` here. We will see why we don't do this shortly.

Architecture - Signal Declarations

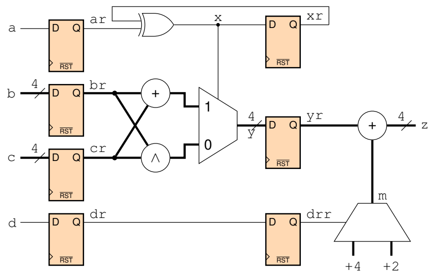


```
architecture arch of circuit2code is
  signal br, cr, y, yr :
    std_ulogic_vector(3 downto 0);
  signal ar, x, xr, dr, drr : std_ulogic;
begin
```

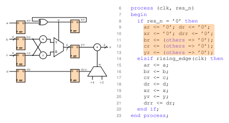


After we declared the required internal signals, we can start to implement the behavior of our circuit within the architecture body. A good place to start are the sequential elements. Hence, we create a synchronous process by using the appropriate pattern presented in the sequential circuit elements lecture. Inside this process we write to all signals in our design that represent the output of a register. For our example these are the signals whose names end with an `r`.

Architecture - Registers

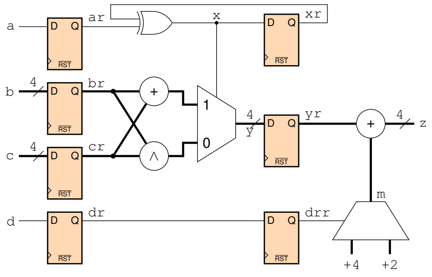


```
process (clk, res_n)
begin
  if res_n = '0' then
    ar <= '0'; dr <= '0';
    xr <= '0'; drr <= '0';
    br <= (others => '0');
    cr <= (others => '0');
    yr <= (others => '0');
  elsif rising_edge(clk) then
    ar <= a;
    br <= b;
    cr <= c;
    dr <= d;
    xr <= x;
    yr <= y;
    drr <= dr;
  end if;
end process;
```

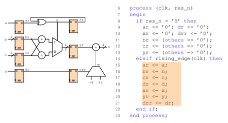


As always in our course, we use an asynchronous reset. Furthermore, as already noted, the circuit diagram indicates that the reset is active-low. In addition to that, because the flip-flops in the circuit feature a reset rather than a set input, we initialize all of them to zero.

Architecture - Registers

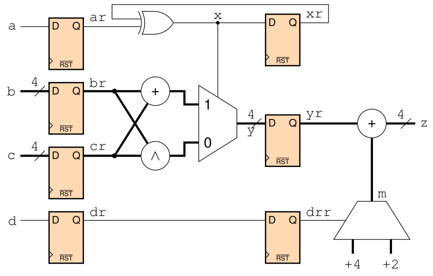


```
process (clk, res_n)
begin
  if res_n = '0' then
    ar <= '0'; dr <= '0';
    xr <= '0'; drr <= '0';
    br <= (others => '0');
    cr <= (others => '0');
    yr <= (others => '0');
  elsif rising_edge(clk) then
    ar <= a;
    br <= b;
    cr <= c;
    dr <= d;
    xr <= x;
    yr <= y;
    drr <= dr;
  end if;
end process;
```

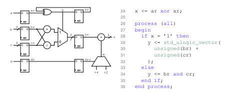


On rising clock edges we simply assign each input signal of a sequential element to the appropriate output signal. For example, the signal `a` is assigned to `ar`. This process is a quite straight forward step that doesn't demand much thought and is more or less independent of the actual circuit.

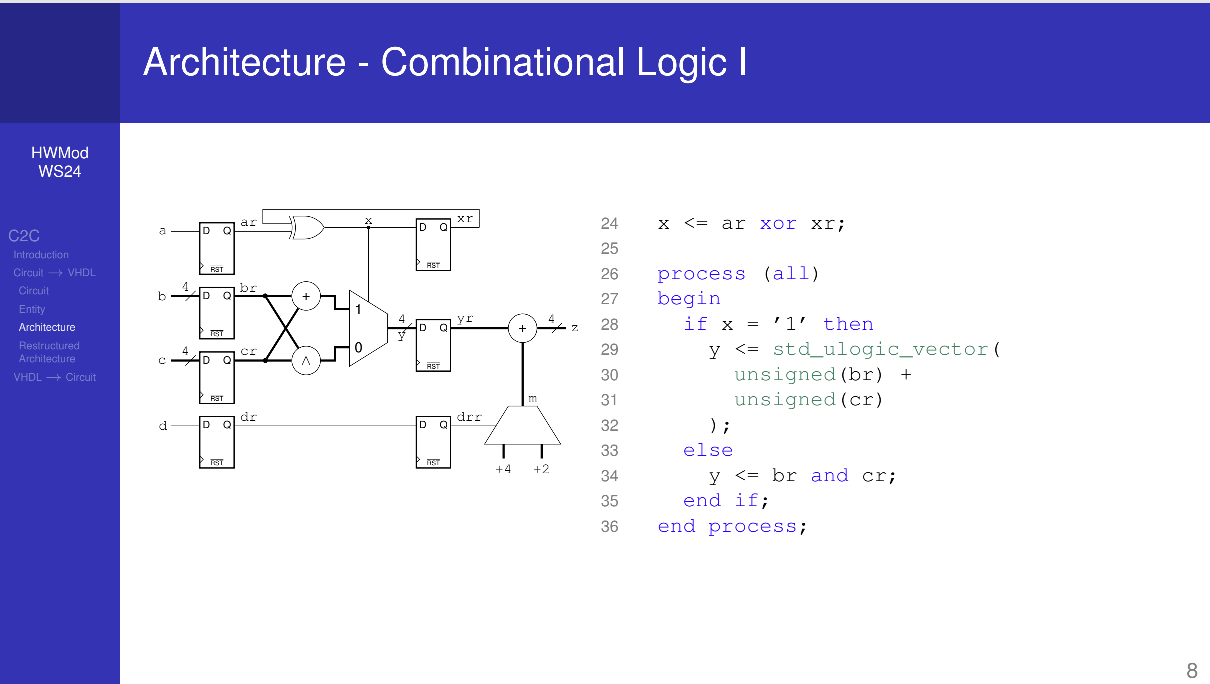
Architecture - Registers



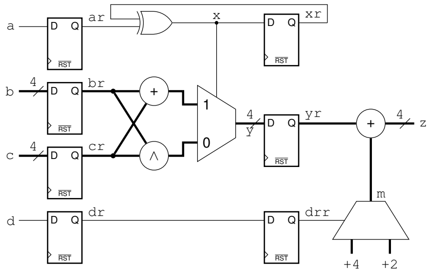
```
process (clk, res_n)
begin
  if res_n = '0' then
    ar <= '0'; dr <= '0';
    xr <= '0'; drr <= '0';
    br <= (others => '0');
    cr <= (others => '0');
    yr <= (others => '0');
  elsif rising_edge(clk) then
    ar <= a;
    br <= b;
    cr <= c;
    dr <= d;
    xr <= x;
    yr <= y;
    drr <= dr;
  end if;
end process;
```



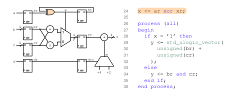
With the sequential circuit elements in place, we can now add the combinational logic between them that defines most of our circuits specific behavior.



Architecture - Combinational Logic I

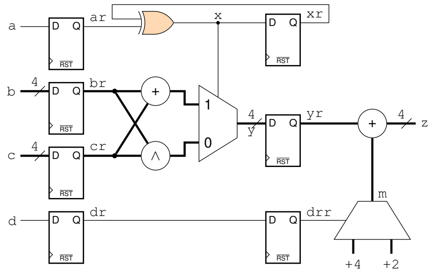


```vhdl
x <= ar xor xr;

process (all)
begin
  if x = '1' then
    y <= std_ulogic_vector(
      unsigned(br) +
      unsigned(cr)
    );
  else
    y <= br and cr;
  end if;
end process;
```

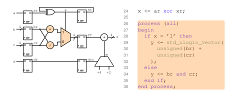


Let's start with the XOR gate in the upper part of the circuit that produces the x signal. While there exist multiple ways to describe this logic, we can very easily express such simple gates using concurrent signal assignments. An assignment like the one highlighted on the slide typically directly maps to a single gate of the respective type for single bit signals, and to multiple such gates for multi-bit signals.

Architecture - Combinational Logic I

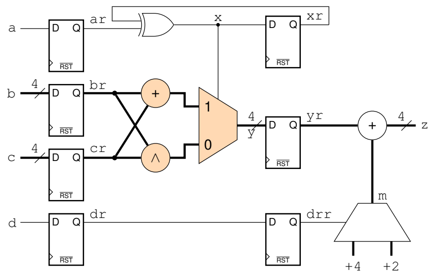


```
x <= ar xor xr;

process (all)
begin
  if x = '1' then
    y <= std_ulogic_vector(
      unsigned(br) +
      unsigned(cr)
    );
  else
    y <= br and cr;
  end if;
end process;
```

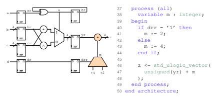


For the logic that decides whether to forward the "bitwise-AND" or the sum of the signals `br` and `cr` to the signal `y` we use a separate process. As you already know, multiplexers can be expressed using if-else statements. Hence, we test the signal `x` and either assign the result of the "bitwise-AND" or the addition to `y`. Of course this is not the only possibility how we can describe this sub-circuit in VHDL. One alternative would be to use a concurrent signal assignment with an appropriate when/else expression. However, operations comprising multiple circuit elements quickly become quite confusing using such assignments, and we therefore usually recommend the use of a process in such cases. Please note that, we would get the exact same circuit, if we would put the concurrent signal assignment for `x` into the process for `y`.

Architecture - Combinational Logic I

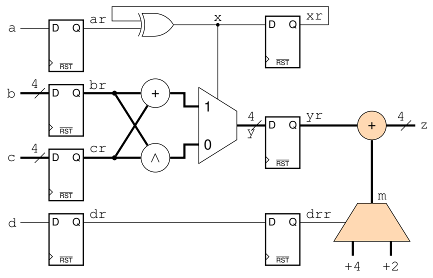


```
x <= ar xor xr;

process (all)
begin
  if x = '1' then
    y <= std_ulogic_vector(
      unsigned(br) +
      unsigned(cr)
    );
  else
    y <= br and cr;
  end if;
end process;
```

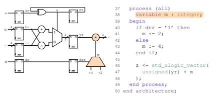


The last thing we need to do is to implement the logic that produces the output signal z. For that we use a separate process, although this is not strictly required. It would be semantically equivalent to simply add this code to the other combinational process that produces the y signal. However, it is often preferable to restrict processes to related parts of a circuit in order to give your code more structure and to improve its readability and maintainability.

Architecture - Combinational Logic II

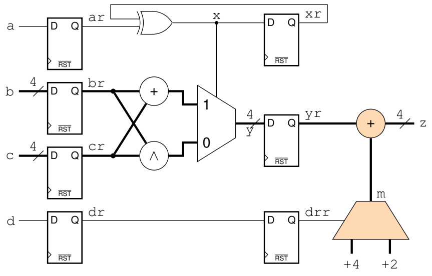


```vhdl
  process (all)
    variable m : integer;
  begin
    if drr = '1' then
      m := 2;
    else
      m := 4;
    end if;

    z <= std_ulogic_vector(
      unsigned(yr) + m
    );
  end process;
end architecture;
```

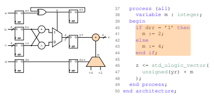


We start with the declaration of the intermediate `m` as a variable in the process. Since the output of the second multiplexer is only required within this process, it makes sense to restrict its scope in order to reduce needless cluttering of the architecture's name space. We select the `integer` data type for `m`. However, `std_ulogic_vector`, or `unsigned` would also be valid options.

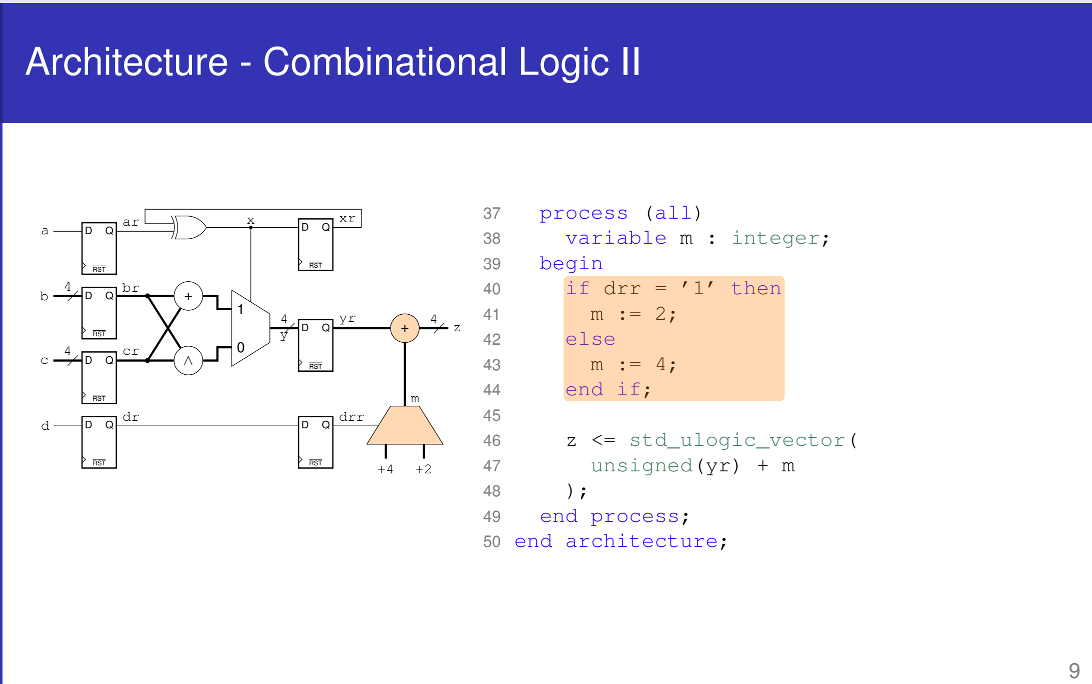
Architecture - Combinational Logic II



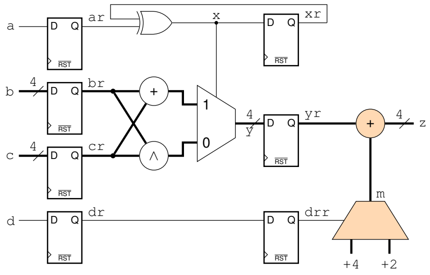
```
  process (all)
    variable m : integer;
  begin
    if drr = '1' then
      m := 2;
    else
      m := 4;
    end if;

    z <= std_ulogic_vector(
      unsigned(yr) + m
    );
  end process;
end architecture;
```



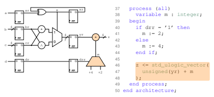
Next we implement the multiplexer that selects between the constants two and four based on the value of `drr` and writes to the variable `m`.



Architecture - Combinational Logic II

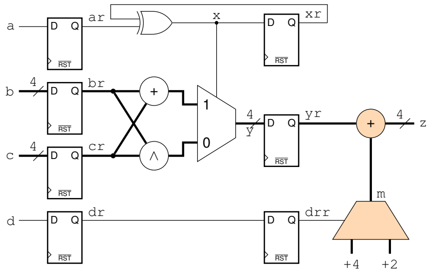


```
  process (all)
    variable m : integer;
  begin
    if drr = '1' then
      m := 2;
    else
      m := 4;
    end if;

    z <= std_ulogic_vector(
      unsigned(yr) + m
    );
  end process;
end architecture;
```

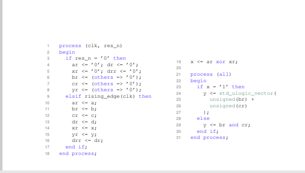


Finally, we produce the output z by adding yr to m. This completes our architecture and thus also our VHDL design.

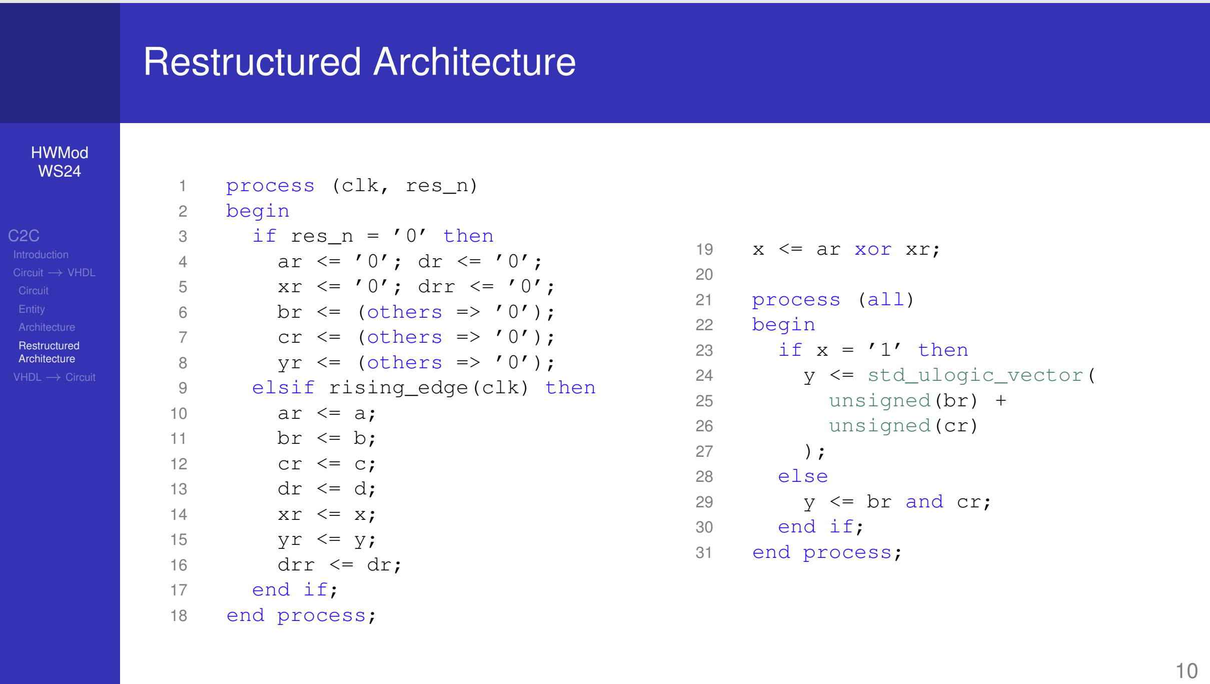
Architecture - Combinational Logic II



```
  process (all)
    variable m : integer;
  begin
    if drr = '1' then
      m := 2;
    else
      m := 4;
    end if;

    z <= std_ulogic_vector(
      unsigned(yr) + m
    );
  end process;
end architecture;
```



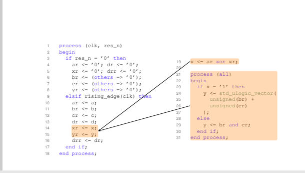
Before we continue with the next example, we want to show you that there usually exist multiple different VHDL descriptions of the exact same circuit. Therefore, let us quickly discuss some example modifications to our previous VHDL code that do not change the resulting circuit. The reason why we discuss that, is to really show you the different ways to describe the exact same circuit.



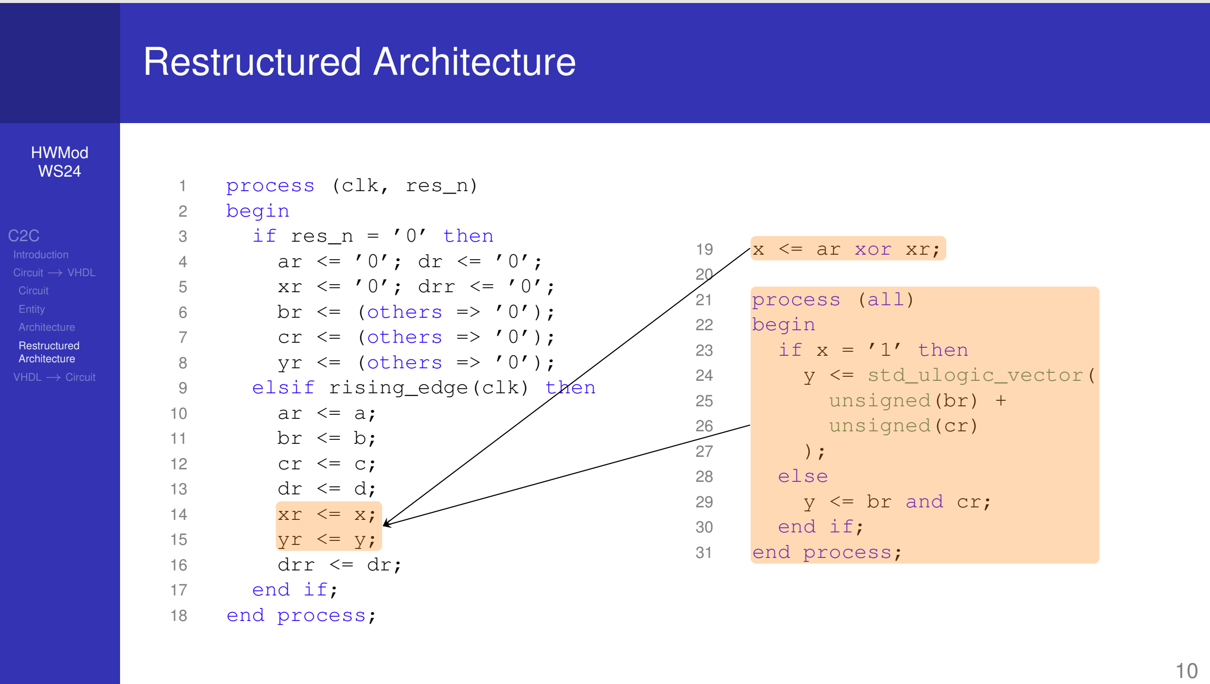
Restructured Architecture

```vhdl
process (clk, res_n)
begin
  if res_n = '0' then
    ar <= '0'; dr <= '0';
    xr <= '0'; drr <= '0';
    br <= (others => '0');
    cr <= (others => '0');
    yr <= (others => '0');
  elsif rising_edge(clk) then
    ar <= a;
    br <= b;
    cr <= c;
    dr <= d;
    xr <= x;
    yr <= y;
    drr <= dr;
  end if;
end process;
```

```vhdl
x <= ar xor xr;

process (all)
begin
  if x = '1' then
    y <= std_ulogic_vector(
      unsigned(br) +
      unsigned(cr)
    );
  else
    y <= br and cr;
  end if;
end process;
```



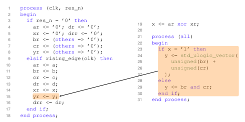
One thing we could do is merge the concurrent signal assignment and the process that produces the y signal into the synchronous process. This change would reduce the overall length of our architecture. However, it arguably also makes the VHDL code less structured, as we no longer have dedicated processes for synchronous and combinational parts of the circuit.



Restructured Architecture

```vhdl
process (clk, res_n)
begin
  if res_n = '0' then
    ar <= '0'; dr <= '0';
    xr <= '0'; drr <= '0';
    br <= (others => '0');
    cr <= (others => '0');
    yr <= (others => '0');
  elsif rising_edge(clk) then
    ar <= a;
    br <= b;
    cr <= c;
    dr <= d;
    xr <= x;
    yr <= y;
    drr <= dr;
  end if;
end process;
```

```vhdl
x <= ar xor xr;

process (all)
begin
  if x = '1' then
    y <= std_ulogic_vector(
      unsigned(br) +
      unsigned(cr)
    );
  else
    y <= br and cr;
  end if;
end process;
```

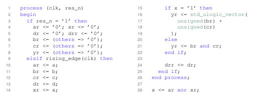


Let's start with the process. The goal is to take the shown sequence of code and replace the assignment in the synchronous process with it. However, care must be taken because the combinational process writes to the signal `y`, while the synchronous process writes to `yr`. To get semantically equivalent code, we therefore need to replace all occurrences of `y` by `yr` in the part we take from the combinational process.

Restructured Architecture

```
process (clk, res_n)
begin
  if res_n = '0' then
    ar <= '0'; dr <= '0';
    xr <= '0'; drr <= '0';
    br <= (others => '0');
    cr <= (others => '0');
    yr <= (others => '0');
  elsif rising_edge(clk) then
    ar <= a;
    br <= b;
    cr <= c;
    dr <= d;
    xr <= x;
    yr <= y;
    drr <= dr;
  end if;
end process;
```

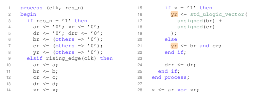
```
x <= ar xor xr;

process (all)
begin
  if x = '1' then
    y <= std_ulogic_vector(
      unsigned(br) +
      unsigned(cr)
    );
  else
    y <= br and cr;
  end if;
end process;
```



This slide shows the result of this modification.

Restructured Architecture

```
process (clk, res_n)
begin
  if res_n = '1' then
    ar <= '0'; xr <= '0';
    dr <= '0'; drr <= '0';
    br <= (others => '0');
    cr <= (others => '0');
    yr <= (others => '0');
  elsif rising_edge(clk) then
    ar <= a;
    br <= b;
    cr <= c;
    dr <= d;
    xr <= x;
    if x = '1' then
      yr <= std_ulogic_vector(
        unsigned(br) +
        unsigned(cr)
      );
    else
      yr <= br and cr;
    end if;

    drr <= dr;
  end if;
end process;

x <= ar xor xr;
```

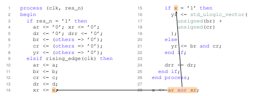


```
process (clk, res_n)
begin
  if res_n = '1' then
    ar <= '0'; xr <= '0';
    dr <= '0'; drr <= '0';
    br <= (others => '0');
    cr <= (others => '0');
    yr <= (others => '0');
  elsif rising_edge(clk) then
    ar <= a;
    br <= b;
    cr <= c;
    dr <= d;
    xr <= x;
    if x = '1' then
      yr <= std_ulogic_vector(
        unsigned(br) +
        unsigned(cr)
      );
    else
      yr <= br and cr;
    end if;

    drr <= dr;
  end if;
end process;

x <= ar xor xr;
```

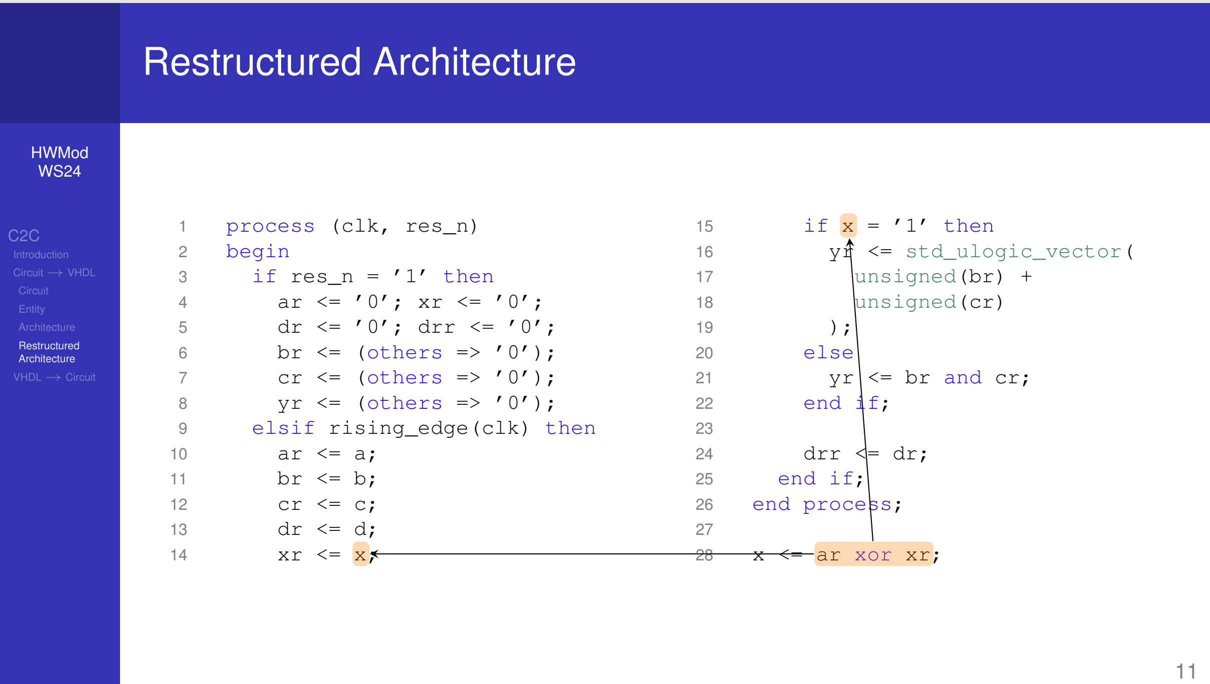
Notice that the synchronous process only writes to the signal yr and that the signal y is no longer needed at all. We can therefore remove its declaration altogether.

Restructured Architecture

```
process (clk, res_n)
begin
  if res_n = '1' then
    ar <= '0'; xr <= '0';
    dr <= '0'; drr <= '0';
    br <= (others => '0');
    cr <= (others => '0');
    yr <= (others => '0');
  elsif rising_edge(clk) then
    ar <= a;
    br <= b;
    cr <= c;
    dr <= d;
    xr <= x;
    if x = '1' then
      yr <= std_ulogic_vector(
        unsigned(br) +
        unsigned(cr)
      );
    else
      yr <= br and cr;
    end if;

    drr <= dr;
  end if;
end process;

x <= ar xor xr;
```

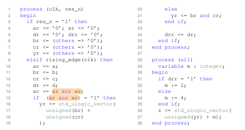


Finally, we can also remove the signal x from our design, by directly calculating the XOR function in the synchronous process as well.

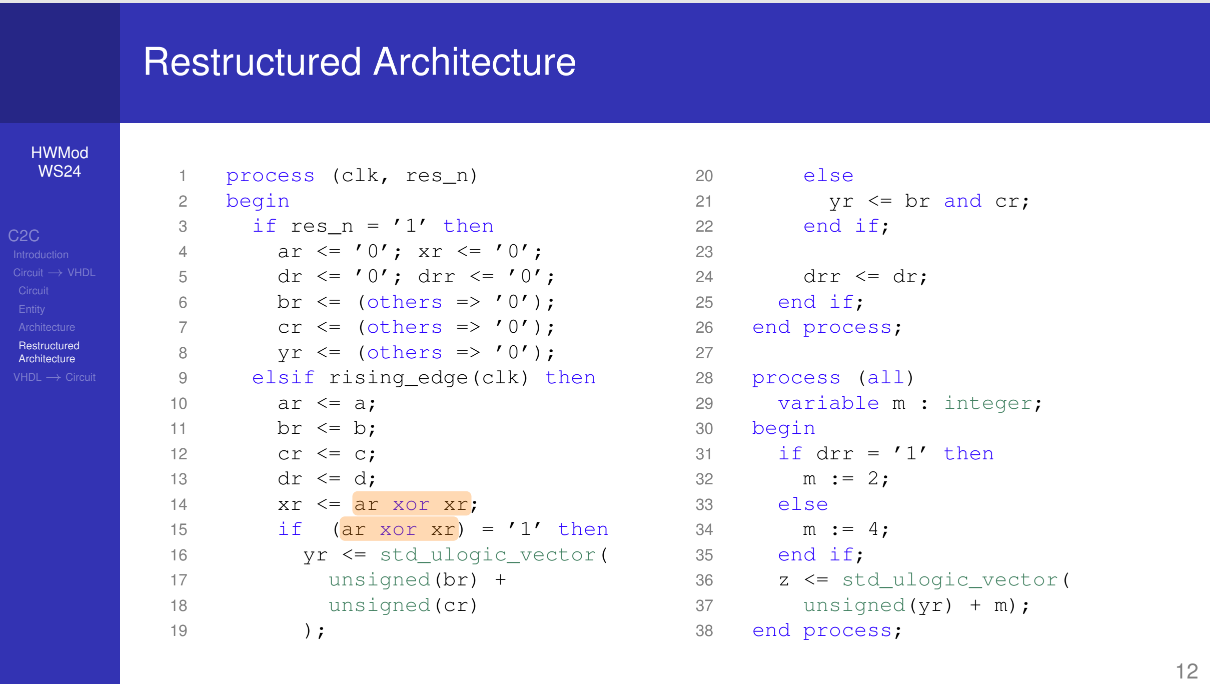


Restructured Architecture

```
process (clk, res_n)
begin
  if res_n = ’1’ then
    ar <= ’0’; xr <= ’0’;
    dr <= ’0’; drr <= ’0’;
    br <= (others => ’0’);
    cr <= (others => ’0’);
    yr <= (others => ’0’);
  elsif rising_edge(clk) then
    ar <= a;
    br <= b;
    cr <= c;
    dr <= d;
    xr <= x;
    if x = ’1’ then
      yr <= std_ulogic_vector(
        unsigned(br) +
        unsigned(cr)
      );
    else
      yr <= br and cr;
    end if;

    drr <= dr;
  end if;
end process;

x <= ar xor xr;
```

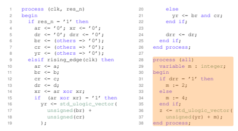


Here, we see the complete final resulting statement part of our architecture.



Restructured Architecture

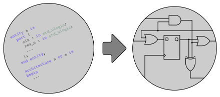
```
process (clk, res_n)
begin
  if res_n = '1' then
    ar <= '0'; xr <= '0';
    dr <= '0'; drr <= '0';
    br <= (others => '0');
    cr <= (others => '0');
    yr <= (others => '0');
  elsif rising_edge(clk) then
    ar <= a;
    br <= b;
    cr <= c;
    dr <= d;
    xr <= ar xor xr;
    if  (ar xor xr) = '1' then
      yr <= std_ulogic_vector(
        unsigned(br) +
        unsigned(cr)
      );
    else
      yr <= br and cr;
    end if;

    drr <= dr;
  end if;
end process;

process (all)
  variable m : integer;
begin
  if drr = '1' then
    m := 2;
  else
    m := 4;
  end if;
  z <= std_ulogic_vector(
    unsigned(yr) + m);
end process;
```



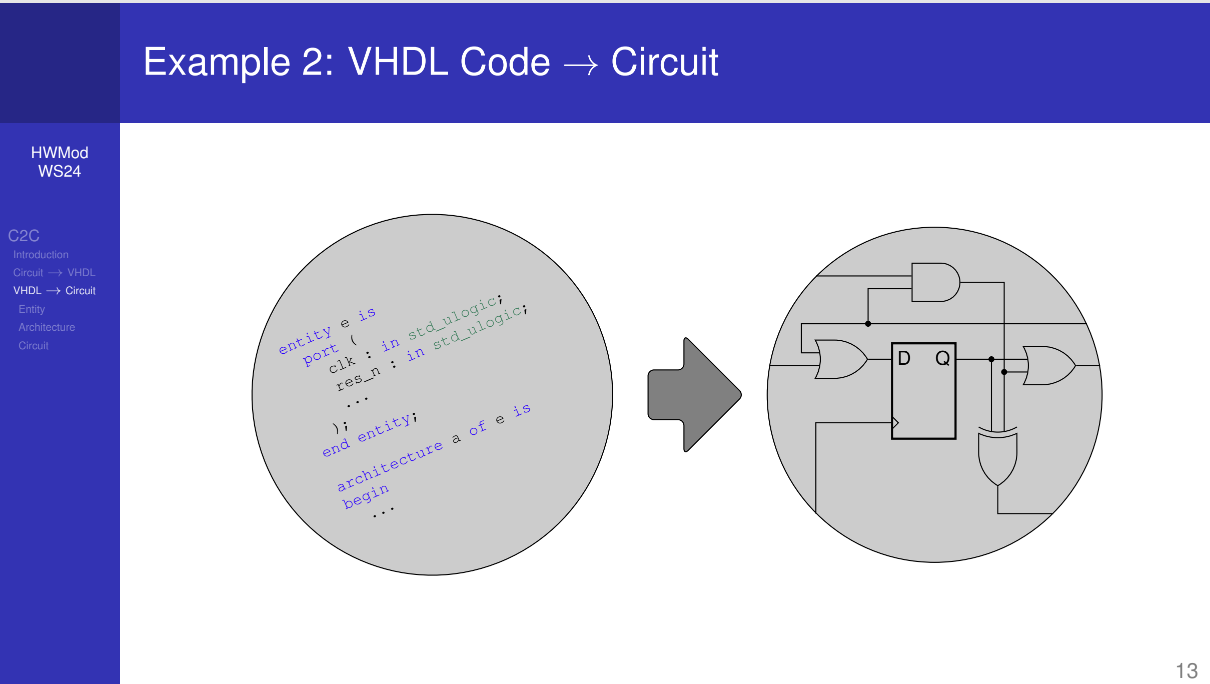
Please note that it is NOT possible to also move the last remaining combinational process generating `z` into the synchronous process as well. This is because the output `z` is not directly provided by a register, but generated combinationally out of the signals `yr` and `drr`.

Restructured Architecture

```
process (clk, res_n)
begin
  if res_n = '1' then
    ar <= '0'; xr <= '0';
    dr <= '0'; drr <= '0';
    br <= (others => '0');
    cr <= (others => '0');
    yr <= (others => '0');
  elsif rising_edge(clk) then
    ar <= a;
    br <= b;
    cr <= c;
    dr <= d;
    xr <= ar xor xr;
    if  (ar xor xr) = '1' then
      yr <= std_ulogic_vector(
        unsigned(br) +
        unsigned(cr)
      );
    else
      yr <= br and cr;
    end if;

    drr <= dr;
  end if;
end process;

process (all)
  variable m : integer;
begin
  if drr = '1' then
    m := 2;
  else
    m := 4;
  end if;
  z <= std_ulogic_vector(
    unsigned(yr) + m);
end process;
```



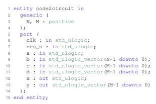
Let us now move to the second example, where we take a VHDL design and derive a circuit diagram from it.



Example 2: VHDL Code → Circuit

HWMod
WS24

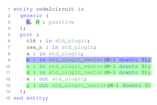
```
entity e is
  port (
    clk : in std_ulogic;
    res_n : in std_ulogic;
    ...
  );
end entity;

architecture a of e is
begin
  ...
```



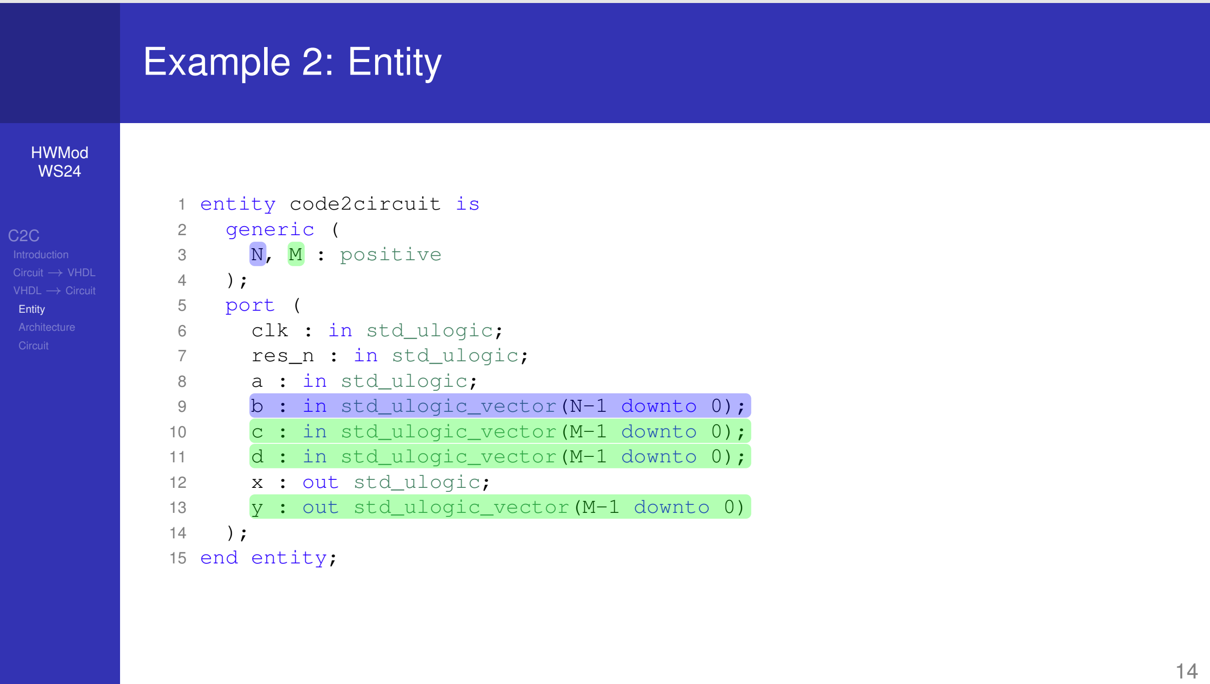
This slide shows the entity declaration of our example circuit. As with the previous example the design does not serve any particular purpose, other than being an example.

Example 2: Entity

```vhdl
entity code2circuit is
  generic (
    N, M : positive
  );
  port (
    clk : in std_ulogic;
    res_n : in std_ulogic;
    a : in std_ulogic;
    b : in std_ulogic_vector(N-1 downto 0);
    c : in std_ulogic_vector(M-1 downto 0);
    d : in std_ulogic_vector(M-1 downto 0);
    x : out std_ulogic;
    y : out std_ulogic_vector(M-1 downto 0)
  );
end entity;
```



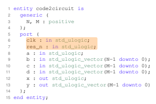
The entity has two generics `M` and `N` that define the widths of the inputs `b`, `c` and `d` as well as of the output `y`.



Example 2: Entity

HWMod
WS24

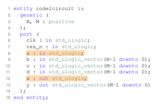
```
entity code2circuit is
  generic (
    N, M : positive
  );
  port (
    clk : in std_ulogic;
    res_n : in std_ulogic;
    a : in std_ulogic;
    b : in std_ulogic_vector(N-1 downto 0);
    c : in std_ulogic_vector(M-1 downto 0);
    d : in std_ulogic_vector(M-1 downto 0);
    x : out std_ulogic;
    y : out std_ulogic_vector(M-1 downto 0)
  );
end entity;
```



It also has a clock and reset input, indicating that it contains some sequential circuit elements.

Example 2: Entity

```
entity code2circuit is
  generic (
    N, M : positive
  );
  port (
    clk : in std_ulogic;
    res_n : in std_ulogic;
    a : in std_ulogic;
    b : in std_ulogic_vector(N-1 downto 0);
    c : in std_ulogic_vector(M-1 downto 0);
    d : in std_ulogic_vector(M-1 downto 0);
    x : out std_ulogic;
    y : out std_ulogic_vector(M-1 downto 0)
  );
end entity;
```

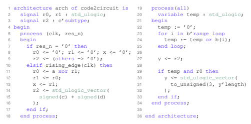


```vhdl
entity code2circuit is
  generic (
    N, M : positive
  );
  port (
    clk : in std_ulogic;
    res_n : in std_ulogic;
    a : in std_ulogic;
    b : in std_ulogic_vector(N-1 downto 0);
    c : in std_ulogic_vector(M-1 downto 0);
    d : in std_ulogic_vector(M-1 downto 0);
    x : out std_ulogic;
    y : out std_ulogic_vector(M-1 downto 0)
  );
end entity;
```

Finally, the entity has an input `a` and an output `x`, both being single-bit signals.

Example 2: Entity

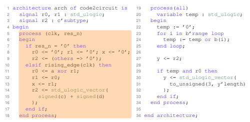
```vhdl
entity code2circuit is
  generic (
    N, M : positive
  );
  port (
    clk : in std_ulogic;
    res_n : in std_ulogic;
    a : in std_ulogic;
    b : in std_ulogic_vector(N-1 downto 0);
    c : in std_ulogic_vector(M-1 downto 0);
    d : in std_ulogic_vector(M-1 downto 0);
    x : out std_ulogic;
    y : out std_ulogic_vector(M-1 downto 0)
  );
end entity;
```



Let us now look at a given architecture for this entity. We can see that it consists of two processes.

Example 2: Architecture

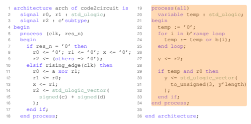
```
architecture arch of code2circuit is
  signal r0, r1 : std_ulogic;
  signal r2 : c'subtype;
begin
  process (clk, res_n)
  begin
    if res_n = '0' then
      r0 <= '0'; r1 <= '0'; x <= '0';
      r2 <= (others => '0');
    elsif rising_edge(clk) then
      r0 <= a xor r1;
      r1 <= r0;
      x <= r1;
      r2 <= std_ulogic_vector(
        signed(c) + signed(d)
      );
    end if;
  end process;

  process(all)
    variable temp : std_ulogic;
  begin
    temp := '0';
    for i in b'range loop
      temp := temp or b(i);
    end loop;

    y <= r2;

    if temp and r0 then
      y <= std_ulogic_vector(
        to_unsigned(3, y'length)
      );
    end if;
  end process;

end architecture;
```



The process on the left describes sequential circuit elements, as indicated by the usual code pattern for D flip-flops.

Example 2: Architecture

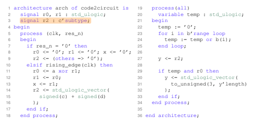
```
architecture arch of code2circuit is
  signal r0, r1 : std_ulogic;
  signal r2 : c'subtype;
begin
  process (clk, res_n)
  begin
    if res_n = '0' then
      r0 <= '0'; r1 <= '0'; x <= '0';
      r2 <= (others => '0');
    elsif rising_edge(clk) then
      r0 <= a xor r1;
      r1 <= r0;
      x <= r1;
      r2 <= std_ulogic_vector(
        signed(c) + signed(d)
      );
    end if;
  end process;

  process(all)
    variable temp : std_ulogic;
  begin
    temp := '0';
    for i in b'range loop
      temp := temp or b(i);
    end loop;

    y <= r2;

    if temp and r0 then
      y <= std_ulogic_vector(
        to_unsigned(3, y'length)
      );
    end if;
  end process;

end architecture;
```



The other one only contains combinational logic, as is also suggested by the by now familiar `all` keyword in its sensitivity list. You should be familiar with all VHDL features used in this design.

Example 2: Architecture

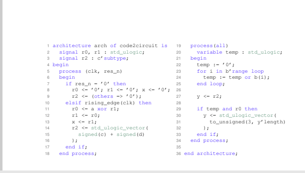
```vhdl
architecture arch of code2circuit is
  signal r0, r1 : std_ulogic;
  signal r2 : c'subtype;
begin
  process (clk, res_n)
  begin
    if res_n = '0' then
      r0 <= '0'; r1 <= '0'; x <= '0';
      r2 <= (others => '0');
    elsif rising_edge(clk) then
      r0 <= a xor r1;
      r1 <= r0;
      x <= r1;
      r2 <= std_ulogic_vector(
        signed(c) + signed(d)
      );
    end if;
  end process;

  process(all)
    variable temp : std_ulogic;
  begin
    temp := '0';
    for i in b'range loop
      temp := temp or b(i);
    end loop;

    y <= r2;

    if temp and r0 then
      y <= std_ulogic_vector(
        to_unsigned(3, y'length)
      );
    end if;
  end process;

end architecture;
```



The only really noteworthy thing is the declaration of the `r2` signal. Here the `subtype` attribute is used on the input `c` to extract its fully-constrained type information. This is a neat way to declare a signal, variable or constant with the same type as an already declared object and saves us from re-specifying the whole type information. Unfortunately this construct is not supported by all synthesis tools, but works fine in simulation. We still wanted to include it, such that you have seen this feature once.

Example 2: Architecture

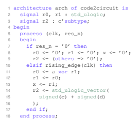
```
architecture arch of code2circuit is
  signal r0, r1 : std_ulogic;
  signal r2 : c'subtype;
begin
  process (clk, res_n)
  begin
    if res_n = '0' then
      r0 <= '0'; r1 <= '0'; x <= '0';
      r2 <= (others => '0');
    elsif rising_edge(clk) then
      r0 <= a xor r1;
      r1 <= r0;
      x <= r1;
      r2 <= std_ulogic_vector(
        signed(c) + signed(d)
      );
    end if;
  end process;

  process(all)
    variable temp : std_ulogic;
  begin
    temp := '0';
    for i in b'range loop
      temp := temp or b(i);
    end loop;

    y <= r2;

    if temp and r0 then
      y <= std_ulogic_vector(
        to_unsigned(3, y'length)
      );
    end if;
  end process;

end architecture;
```



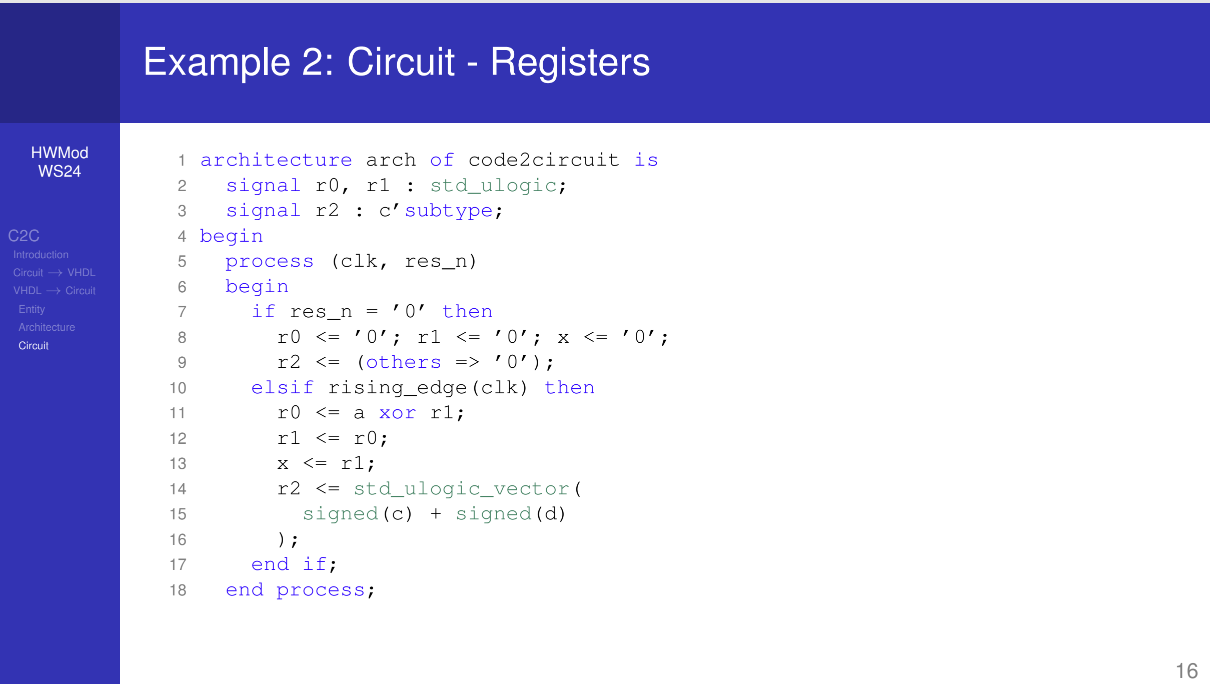
Pause the video at this point and think about how the circuit can look like. You can also make a quick paper sketch.

Example 2: Architecture

```
architecture arch of code2circuit is
  signal r0, r1 : std_ulogic;
  signal r2 : c'subtype;
begin
  process (clk, res_n)
  begin
    if res_n = '0' then
      r0 <= '0'; r1 <= '0'; x <= '0';
      r2 <= (others => '0');
    elsif rising_edge(clk) then
      r0 <= a xor r1;
      r1 <= r0;
      x <= r1;
      r2 <= std_ulogic_vector(
        signed(c) + signed(d)
      );
    end if;
  end process;

  process(all)
    variable temp : std_ulogic;
  begin
    temp := '0';
    for i in b'range loop
      temp := temp or b(i);
    end loop;

    y <= r2;

    if temp and r0 then
      y <= std_ulogic_vector(
        to_unsigned(3, y'length)
      );
    end if;
  end process;

end architecture;
```

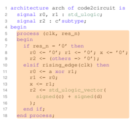


Recall that when going from a circuit diagram to VHDL code, we started by implementing the sequential circuit elements. This is also our starting point for deriving a circuit from VHDL code.

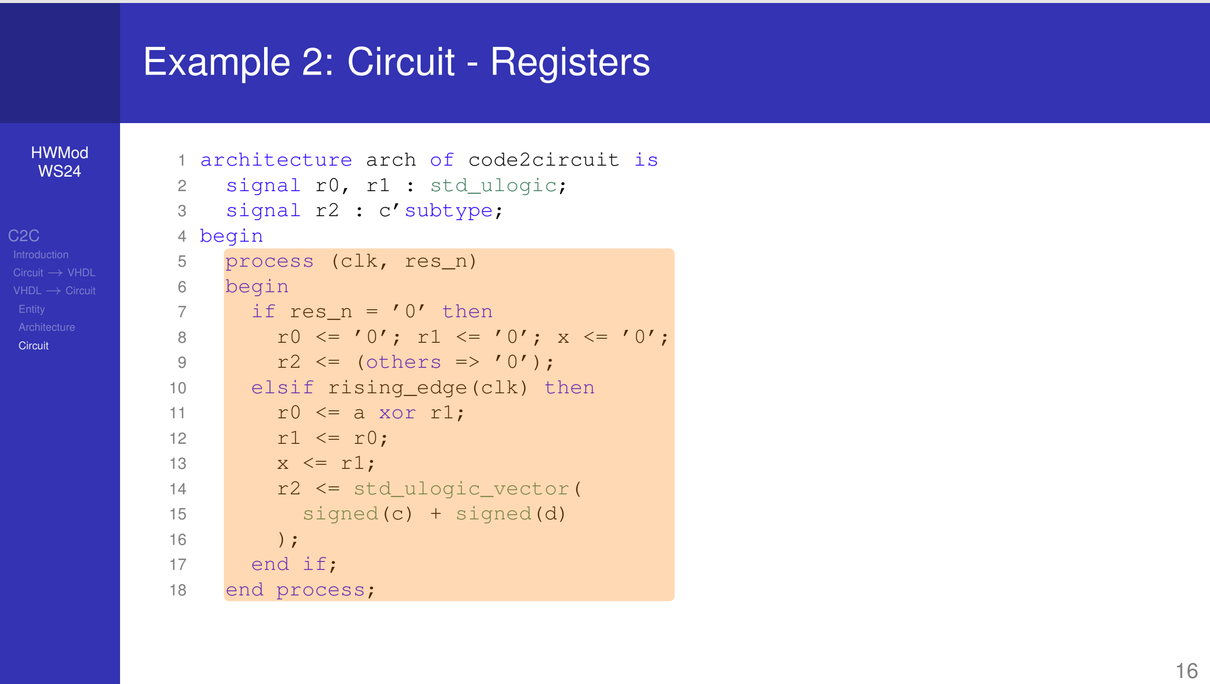


Example 2: Circuit - Registers

```
architecture arch of code2circuit is
  signal r0, r1 : std_ulogic;
  signal r2 : c'subtype;
begin
  process (clk, res_n)
  begin
    if res_n = '0' then
      r0 <= '0'; r1 <= '0'; x <= '0';
      r2 <= (others => '0');
    elsif rising_edge(clk) then
      r0 <= a xor r1;
      r1 <= r0;
      x <= r1;
      r2 <= std_ulogic_vector(
        signed(c) + signed(d)
      );
    end if;
  end process;
```

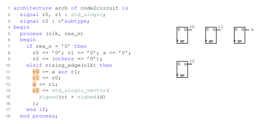


Hence, we first identify all processes that describe sequential circuit elements. As already mentioned, in our example there is only one such process.



Example 2: Circuit - Registers

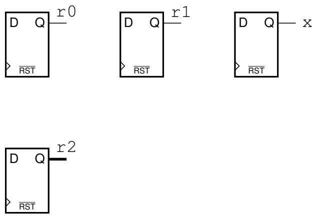
```vhdl
architecture arch of code2circuit is
  signal r0, r1 : std_ulogic;
  signal r2 : c'subtype;
begin
  process (clk, res_n)
  begin
    if res_n = '0' then
      r0 <= '0'; r1 <= '0'; x <= '0';
      r2 <= (others => '0');
    elsif rising_edge(clk) then
      r0 <= a xor r1;
      r1 <= r0;
      x <= r1;
      r2 <= std_ulogic_vector(
        signed(c) + signed(d)
      );
    end if;
  end process;
```

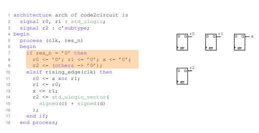


Looking at this process we first identify all the signals the process writes to, as these signals represent the outputs of flip-flops. We add registers with these signals as their outputs to our circuit diagram. If you perform such a conversion on a piece of paper, be sure to leave enough space between the elements, such that you can add in the combinational logic in the next step.

Example 2: Circuit - Registers

```vhdl
architecture arch of code2circuit is
  signal r0, r1 : std_ulogic;
  signal r2 : c'subtype;
begin
  process (clk, res_n)
  begin
    if res_n = '0' then
      r0 <= '0'; r1 <= '0'; x <= '0';
      r2 <= (others => '0');
    elsif rising_edge(clk) then
      r0 <= a xor r1;
      r1 <= r0;
      x <= r1;
      r2 <= std_ulogic_vector(
        signed(c) + signed(d)
      );
    end if;
  end process;
```

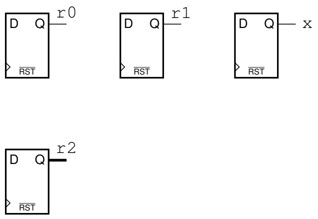


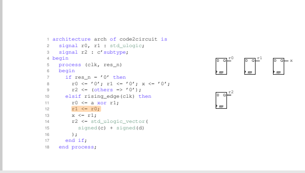


The reset code shows that the reset is active-low and that everything must be initialized to zero. As with the other example we don't draw the clock nor the reset signal, as not to clutter the figure with to many connections. With the flip-flops in place, we can now connect their inputs to the appropriate sources.

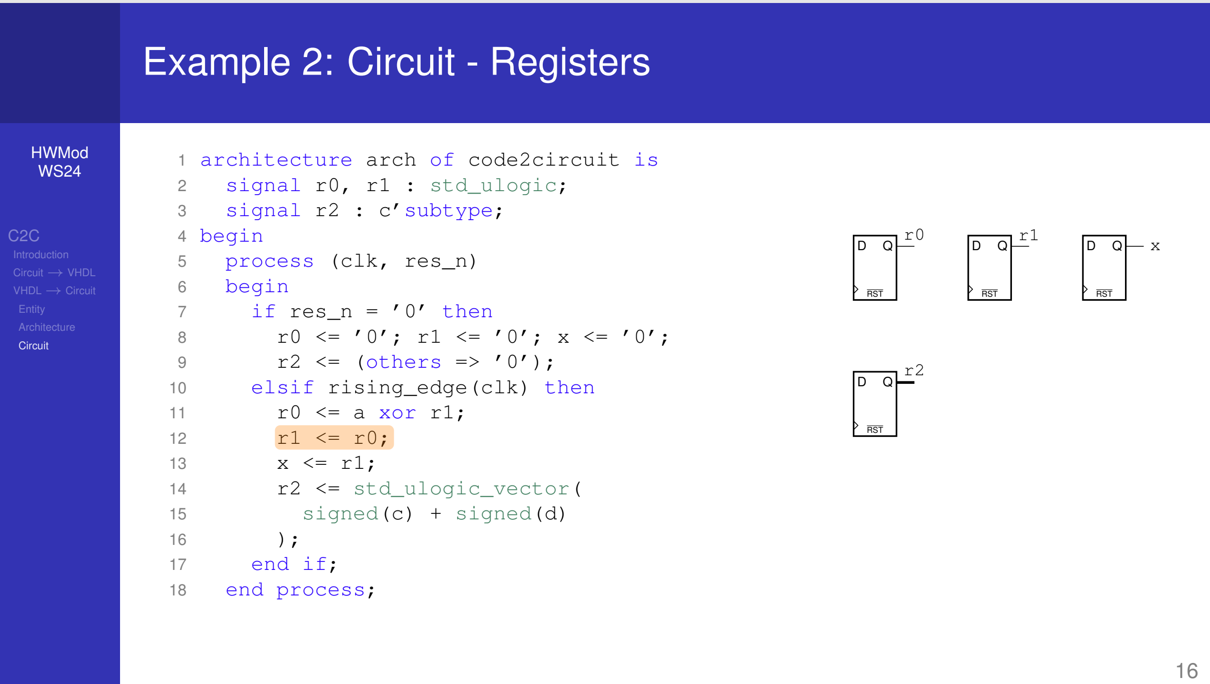
Example 2: Circuit - Registers

```vhdl
architecture arch of code2circuit is
  signal r0, r1 : std_ulogic;
  signal r2 : c'subtype;
begin
  process (clk, res_n)
  begin
    if res_n = '0' then
      r0 <= '0'; r1 <= '0'; x <= '0';
      r2 <= (others => '0');
    elsif rising_edge(clk) then
      r0 <= a xor r1;
      r1 <= r0;
      x <= r1;
      r2 <= std_ulogic_vector(
        signed(c) + signed(d)
      );
    end if;
  end process;
```



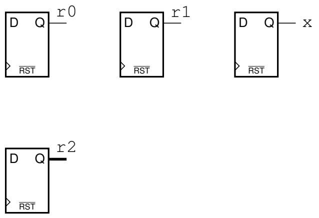


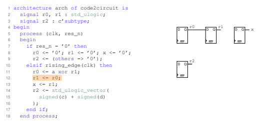
Let's start with `r1`. The respective assignment in the synchronous process specifies that this signal gets assigned the value of `r0`.



Example 2: Circuit - Registers

```
architecture arch of code2circuit is
  signal r0, r1 : std_ulogic;
  signal r2 : c'subtype;
begin
  process (clk, res_n)
  begin
    if res_n = '0' then
      r0 <= '0'; r1 <= '0'; x <= '0';
      r2 <= (others => '0');
    elsif rising_edge(clk) then
      r0 <= a xor r1;
      r1 <= r0;
      x <= r1;
      r2 <= std_ulogic_vector(
        signed(c) + signed(d)
      );
    end if;
  end process;
```

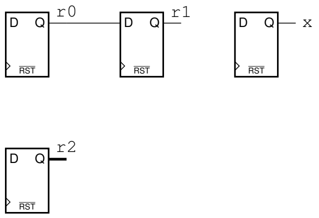


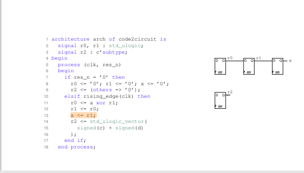


Hence, we simply add an appropriate connection to our circuit diagram.

Example 2: Circuit - Registers

```vhdl
architecture arch of code2circuit is
  signal r0, r1 : std_ulogic;
  signal r2 : c'subtype;
begin
  process (clk, res_n)
  begin
    if res_n = '0' then
      r0 <= '0'; r1 <= '0'; x <= '0';
      r2 <= (others => '0');
    elsif rising_edge(clk) then
      r0 <= a xor r1;
      r1 <= r0;
      x <= r1;
      r2 <= std_ulogic_vector(
        signed(c) + signed(d)
      );
    end if;
  end process;
```

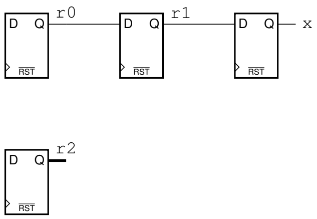


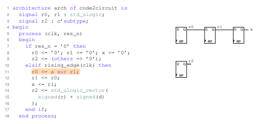


The signal x is handled in the exact same way.

Example 2: Circuit - Registers

```vhdl
architecture arch of code2circuit is
  signal r0, r1 : std_ulogic;
  signal r2 : c'subtype;
begin
  process (clk, res_n)
  begin
    if res_n = '0' then
      r0 <= '0'; r1 <= '0'; x <= '0';
      r2 <= (others => '0');
    elsif rising_edge(clk) then
      r0 <= a xor r1;
      r1 <= r0;
      x <= r1;
      r2 <= std_ulogic_vector(
        signed(c) + signed(d)
      );
    end if;
  end process;
```

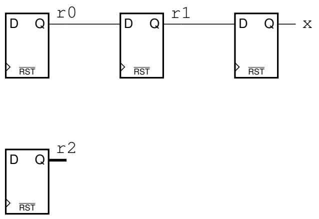


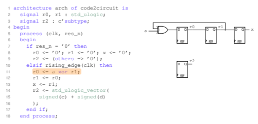


For `r0` the situation is slightly different, as here the right-hand side of the assignment contains an XOR-operation of two signals. However, the respective part of the circuit is still quite straight-forward to implement.

Example 2: Circuit - Registers

```vhdl
architecture arch of code2circuit is
  signal r0, r1 : std_ulogic;
  signal r2 : c'subtype;
begin
  process (clk, res_n)
  begin
    if res_n = '0' then
      r0 <= '0'; r1 <= '0'; x <= '0';
      r2 <= (others => '0');
    elsif rising_edge(clk) then
      r0 <= a xor r1;
      r1 <= r0;
      x <= r1;
      r2 <= std_ulogic_vector(
        signed(c) + signed(d)
      );
    end if;
  end process;
```

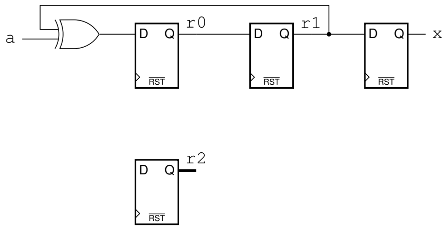


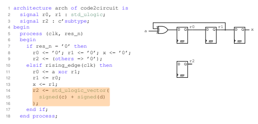


We simply add an XOR gate with the appropriate input signals to the input of the flip-flop that drives `r0`.

Example 2: Circuit - Registers

```
architecture arch of code2circuit is
  signal r0, r1 : std_ulogic;
  signal r2 : c'subtype;
begin
  process (clk, res_n)
  begin
    if res_n = '0' then
      r0 <= '0'; r1 <= '0'; x <= '0';
      r2 <= (others => '0');
    elsif rising_edge(clk) then
      r0 <= a xor r1;
      r1 <= r0;
      x <= r1;
      r2 <= std_ulogic_vector(
        signed(c) + signed(d)
      );
    end if;
  end process;
```

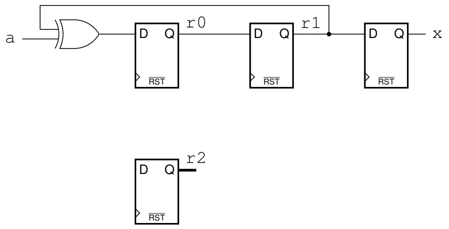


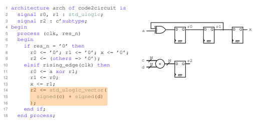


Finally, we have to deal with `r2`. However, this is essentially handled in the exact same way as the XOR gate.

Example 2: Circuit - Registers

```
architecture arch of code2circuit is
  signal r0, r1 : std_ulogic;
  signal r2 : c'subtype;
begin
  process (clk, res_n)
  begin
    if res_n = '0' then
      r0 <= '0'; r1 <= '0'; x <= '0';
      r2 <= (others => '0');
    elsif rising_edge(clk) then
      r0 <= a xor r1;
      r1 <= r0;
      x <= r1;
      r2 <= std_ulogic_vector(
        signed(c) + signed(d)
      );
    end if;
  end process;
```

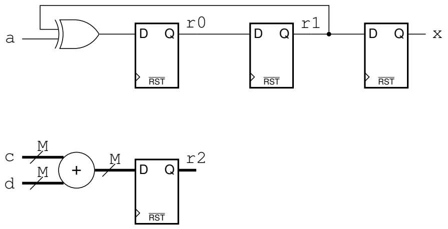


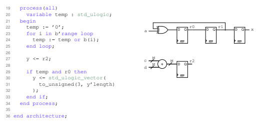


We simply add an adder with the appropriate inputs to the register that drives `r2`. Notice that the adder produces an M-bit wide signal.

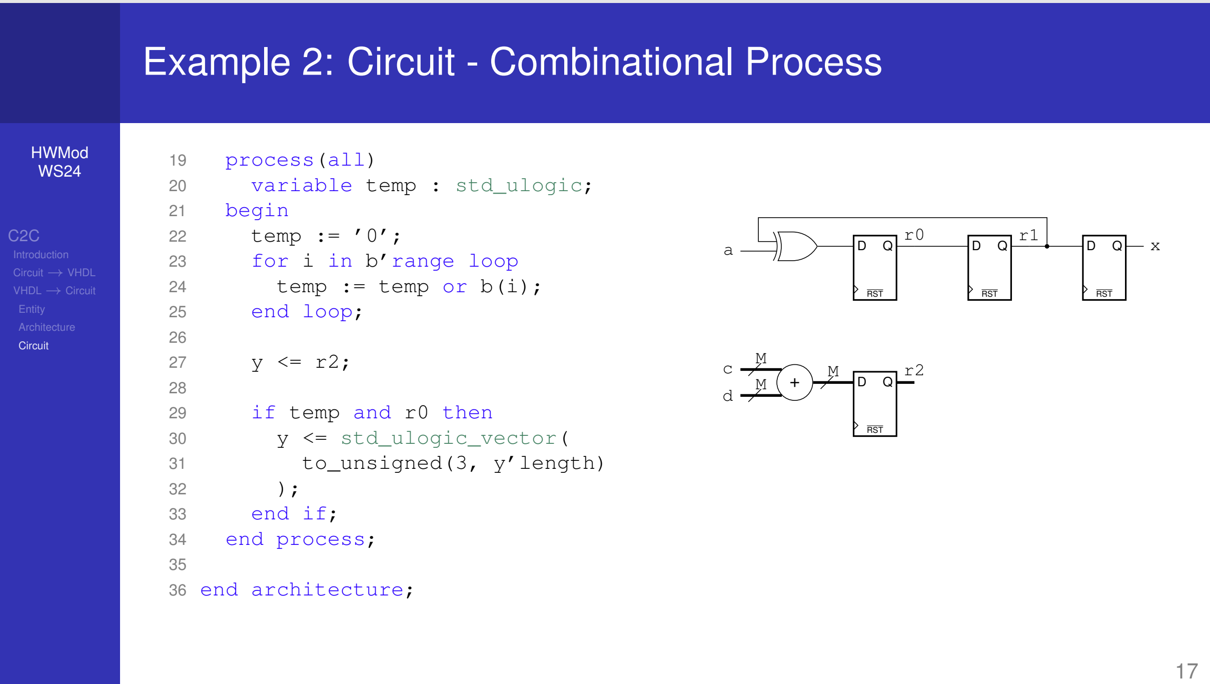
Example 2: Circuit - Registers

```
architecture arch of code2circuit is
  signal r0, r1 : std_ulogic;
  signal r2 : c'subtype;
begin
  process (clk, res_n)
  begin
    if res_n = '0' then
      r0 <= '0'; r1 <= '0'; x <= '0';
      r2 <= (others => '0');
    elsif rising_edge(clk) then
      r0 <= a xor r1;
      r1 <= r0;
      x <= r1;
      r2 <= std_ulogic_vector(
        signed(c) + signed(d)
      );
    end if;
  end process;
```



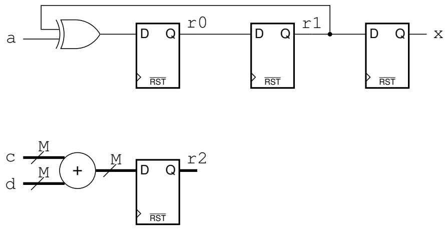


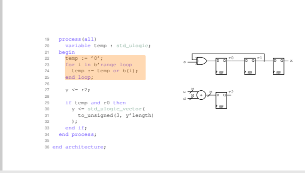
Now let's turn to the second process, which – as already mentioned – only contains combinational logic.



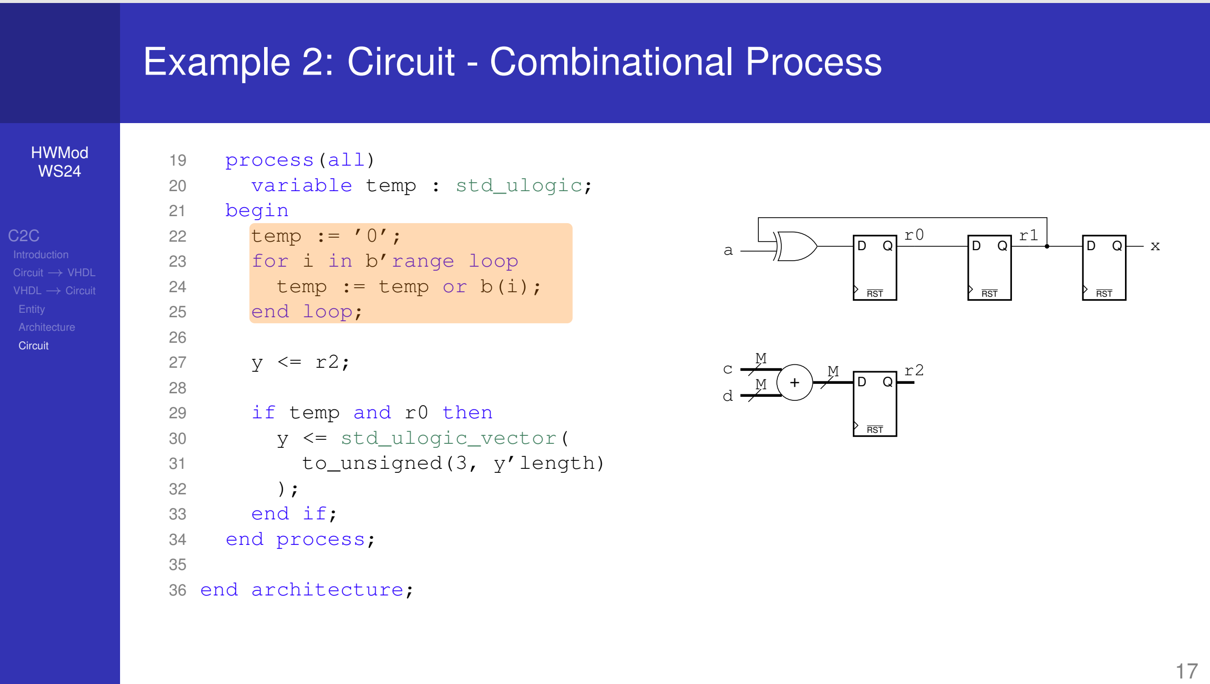
Example 2: Circuit - Combinational Process

```
  process(all)
    variable temp : std_ulogic;
  begin
    temp := '0';
    for i in b'range loop
      temp := temp or b(i);
    end loop;

    y <= r2;

    if temp and r0 then
      y <= std_ulogic_vector(
        to_unsigned(3, y'length)
      );
    end if;
  end process;

end architecture;
```



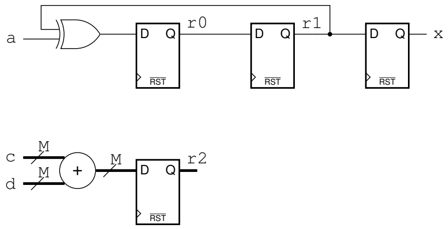


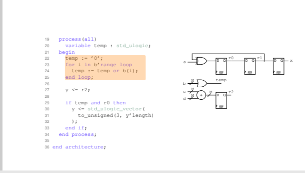
The process initially sets the temporary single-bit variable `temp` to zero. It then uses a for-loop to go over all elements of the input `b` and calculates the disjunction with `temp`, which is then again assigned to the `temp` variable. Hence, if any of the elements of `b` is one, `temp` is set to one as well and will stay at that value until the end of the loop.



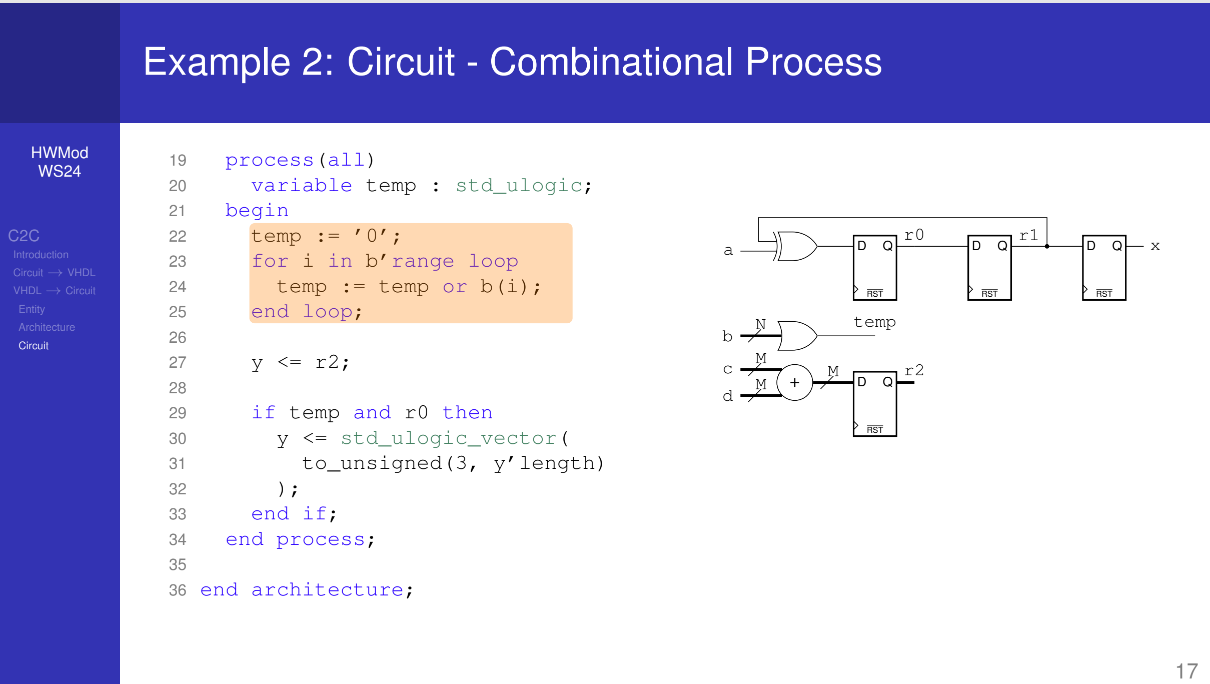
Example 2: Circuit - Combinational Process

```
  process(all)
    variable temp : std_ulogic;
  begin
    temp := '0';
    for i in b'range loop
      temp := temp or b(i);
    end loop;

    y <= r2;

    if temp and r0 then
      y <= std_ulogic_vector(
        to_unsigned(3, y'length)
      );
    end if;
  end process;

end architecture;
```



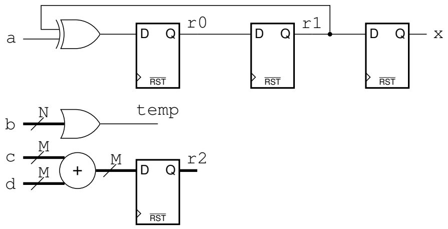


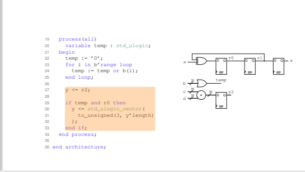
Notice how this effectively describes an N-input OR gate with `temp` as its output signal.



Example 2: Circuit - Combinational Process

```vhdl
  process(all)
    variable temp : std_ulogic;
  begin
    temp := '0';
    for i in b'range loop
      temp := temp or b(i);
    end loop;

    y <= r2;

    if temp and r0 then
      y <= std_ulogic_vector(
        to_unsigned(3, y'length)
      );
    end if;
  end process;

end architecture;
```

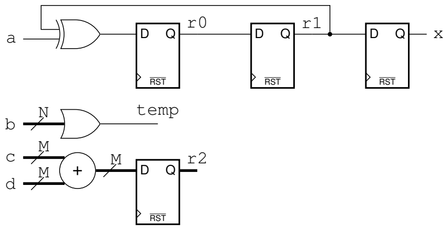


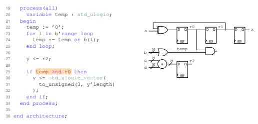


The next part of the process contains an if-statement. First `y` is assigned the value of `r2`. If the if-condition is true this value is then overridden by the constant three. Notice that it would be semantically equivalent to perform the `r2` assignment in the else-branch of the if statement. We already know that if-statements can be expressed using multiplexers. However, before we can do that we have to generate a signal that we can feed into the control input of this multiplexer.

Example 2: Circuit - Combinational Process

```
  process(all)
    variable temp : std_ulogic;
  begin
    temp := '0';
    for i in b'range loop
      temp := temp or b(i);
    end loop;

    y <= r2;

    if temp and r0 then
      y <= std_ulogic_vector(
        to_unsigned(3, y'length)
      );
    end if;
  end process;

end architecture;
```

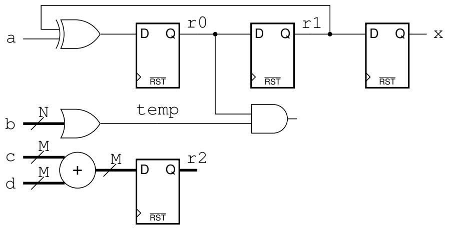


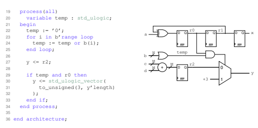


For that purpose we add an AND gate, that evaluates the conjunction used by the expression in the if-condition.

Example 2: Circuit - Combinational Process

```
  process(all)
    variable temp : std_ulogic;
  begin
    temp := '0';
    for i in b'range loop
      temp := temp or b(i);
    end loop;

    y <= r2;

    if temp and r0 then
      y <= std_ulogic_vector(
        to_unsigned(3, y'length)
      );
    end if;
  end process;

end architecture;
```

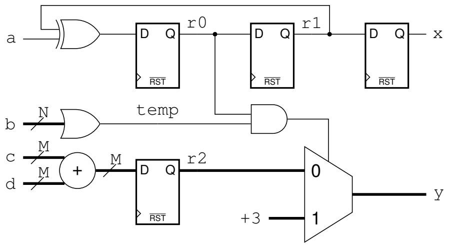




Finally we can use the output signal of this "AND" gate to control the multiplexer that produces the output y of our circuit. And with that our circuit diagram is complete.

Example 2: Circuit - Combinational Process

```
  process(all)
    variable temp : std_ulogic;
  begin
    temp := '0';
    for i in b'range loop
      temp := temp or b(i);
    end loop;

    y <= r2;

    if temp and r0 then
      y <= std_ulogic_vector(
        to_unsigned(3, y'length)
      );
    end if;
  end process;

end architecture;
```



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!