**RTL Design**

- **Gate-level design is now rare!**
  - design automation is necessary to manage the complexity of modern circuits
  - only library designers use gates
  - automated RTL synthesis is now almost universal

- **RTL = Register Transfer Level**
  - The design is perceived as a number of registers with transfer functions which transform data as it passes from one register to another
  - this is a synchronous methodology
  - chosen as the methodology for input to synthesis

---

**RTL Overview**

![RTL Diagram](image)

- **Controller**
- **Register**

---

**RTL Design Steps**

- **There are typically 8 steps in the RTL design process**
  1. Create a Dependency Graph for the Data Path
  2. Determine the widths of the data paths
  3. Decide what resources to provide
  4. Allocate operations to resources and schedule them
  5. Allocate registers to intermediate results
  6. Share registers
  7. Design the controller
  8. Design the reset/initialisation mechanism

- **The order of these steps may vary and may be iterated**

---

**Example**

- **Scalar-product calculator**
- **Not a realistic design, but shows the elements of RTL design**
- **The example will work on 8 element vectors.**
Step 1: Create Dependency Graph for Data Path

- So, the data operations are:
  - 8 multiplications
  - one 8-way addition

Step 1b: 8-way Addition?

Balanced binary tree

Skewed binary tree

Step 1c: Identify and Make Unique Data Paths

- Create what is known as the Single Assignment Form

\[
\begin{align*}
p_0 & := a_0 \times b_0; \\
p_1 & := a_1 \times b_1; \\
p_2 & := a_2 \times b_2; \\
... & \ \\
z_1 & := p_0 + p_1; \\
z_2 & := z_1 + p_2; \\
z_3 & := z_2 + p_3; \\
... & \ \\
z & := z_6 + p_7; \\
\end{align*}
\]

- Each variable assigned only once
- Each statement uses just one operator
- Create new intermediate variables to achieve this

Step 2: Determine Data Path Widths

- In most cases, this is defined for I/O as part of the specification.
- The designer typically then has to decide on internal data path widths.
- In this case we'll assume the specification:
  - Input is 8-bit 2’s-complement
  - Output is 16-bit 2’s-complement
- To achieve this output precision, internal paths must be 16-bit 2’s-complement
- Precision of Operations:
  - all multiplications are 8-bit input, 16-bit output
  - the addition is 16-bit input, 16-bit output
Step 3: Choose Resources to Provide

- As a first solution, target a minimum-area implementation
  - one multiplier, 8-bit inputs, 16-bit output
  - one adder, 16-bit inputs, 16-bit output
- These resources will be shared – remember that the original algorithm requires the following operations:
  - eight multiplications
  - seven additions

Step 4: Allocate/Schedule Operations

- This stage of the design determines which operations are to be performed by which resources at which time
  - Allocation – which resource
  - Scheduling – what time (i.e. which clock cycle)
- Allocation for this example is trivial
  - all additions are allocated to the adder
  - all multiplications are allocated to the multiplier
- Scheduling has many permutations
  - it doesn’t matter what order multiplications are performed
  - it doesn’t matter what order additions are performed
  - addition is associative and commutative

Step 4b: Allocate/Schedule Operations

<table>
<thead>
<tr>
<th>Cycle</th>
<th>mult1</th>
<th>add1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a0 * b0 ⇒ p0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>a1 * b1 ⇒ p1</td>
<td>p0 + p1 ⇒ z1</td>
</tr>
<tr>
<td>3</td>
<td>a2 * b2 ⇒ p2</td>
<td>z1 + p2 ⇒ z2</td>
</tr>
<tr>
<td>4</td>
<td>a3 * b3 ⇒ p3</td>
<td>z2 + p3 ⇒ z3</td>
</tr>
<tr>
<td>5</td>
<td>a4 * b4 ⇒ p4</td>
<td>z3 + p4 ⇒ z4</td>
</tr>
<tr>
<td>6</td>
<td>a5 * b5 ⇒ p5</td>
<td>z4 + p5 ⇒ z5</td>
</tr>
<tr>
<td>7</td>
<td>a6 * b6 ⇒ p6</td>
<td>z5 + p6 ⇒ z6</td>
</tr>
<tr>
<td>8</td>
<td>a7 * b7 ⇒ p7</td>
<td>z6 + p7 ⇒ z</td>
</tr>
</tbody>
</table>

Step 4c: Simplify Schedule

<table>
<thead>
<tr>
<th>Cycle</th>
<th>mult1</th>
<th>add1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a0 * b0 ⇒ p0</td>
<td>0 + p0 ⇒ z0</td>
</tr>
<tr>
<td>2</td>
<td>a1 * b1 ⇒ p1</td>
<td>z0 + p1 ⇒ z1</td>
</tr>
<tr>
<td>3</td>
<td>a2 * b2 ⇒ p2</td>
<td>z1 + p2 ⇒ z2</td>
</tr>
<tr>
<td>4</td>
<td>a3 * b3 ⇒ p3</td>
<td>z2 + p3 ⇒ z3</td>
</tr>
<tr>
<td>5</td>
<td>a4 * b4 ⇒ p4</td>
<td>z3 + p4 ⇒ z4</td>
</tr>
<tr>
<td>6</td>
<td>a5 * b5 ⇒ p5</td>
<td>z4 + p5 ⇒ z5</td>
</tr>
<tr>
<td>7</td>
<td>a6 * b6 ⇒ p6</td>
<td>z5 + p6 ⇒ z6</td>
</tr>
<tr>
<td>8</td>
<td>a7 * b7 ⇒ p7</td>
<td>z6 + p7 ⇒ z</td>
</tr>
</tbody>
</table>
Step 5: Allocate registers to intermediate results

- Every variable in the scheduling that is generated in one cycle and used in another must be registered.
- The schedule proposed only requires outputs $z_0...z$ to be registered, not the products:

$$
\begin{align*}
  & z_0 + p_1 \\
  & z_1 \\
  & z_0 + p_0 \\
  & z_1 \quad \text{(products)}
\end{align*}
$$

- However, the schedule does assume that a multiply-accumulate can be done in one clock cycle

Step 6: Share Registers

- A register can be shared between variables if their lifetimes do not intersect.

Step 6b: Simplify Registers

<table>
<thead>
<tr>
<th>Cycle</th>
<th>mult1</th>
<th>add1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$a_0 \cdot b_0 \Rightarrow p_0$</td>
<td>$0 + p_0 = z_0$</td>
</tr>
<tr>
<td>2</td>
<td>$a_1 \cdot b_1 \Rightarrow p_1$</td>
<td>$z_0 + p_1 = z_1$</td>
</tr>
<tr>
<td>3</td>
<td>$a_2 \cdot b_2 \Rightarrow p_2$</td>
<td>$z + p_1 = z_2$</td>
</tr>
<tr>
<td>4</td>
<td>$a_3 \cdot b_3 \Rightarrow p_3$</td>
<td>$z + p_3 = z_3$</td>
</tr>
<tr>
<td>5</td>
<td>$a_4 \cdot b_4 \Rightarrow p_4$</td>
<td>$z + p_4 = z_4$</td>
</tr>
<tr>
<td>6</td>
<td>$a_5 \cdot b_5 \Rightarrow p_5$</td>
<td>$z + p_5 = z_5$</td>
</tr>
<tr>
<td>7</td>
<td>$a_6 \cdot b_6 \Rightarrow p_6$</td>
<td>$z + p_6 = z_6$</td>
</tr>
<tr>
<td>8</td>
<td>$a_7 \cdot b_7 \Rightarrow p_7$</td>
<td>$z + p_7 = z_7$</td>
</tr>
</tbody>
</table>

Design so Far

- The data path design is now complete:
Step 7: Design the controller

- The controller is responsible for:
  - routing operation inputs to resource inputs at the scheduled cycle
  - enabling registers to store intermediate results

<table>
<thead>
<tr>
<th>Cycle</th>
<th>a mux</th>
<th>b mux</th>
<th>zero mux</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>select a0</td>
<td>select b0</td>
<td>select 0</td>
<td>load add1</td>
</tr>
<tr>
<td>2</td>
<td>select a1</td>
<td>select b1</td>
<td>select z</td>
<td>load add1</td>
</tr>
<tr>
<td>3</td>
<td>select a2</td>
<td>select b2</td>
<td>select z</td>
<td>load add1</td>
</tr>
<tr>
<td>4</td>
<td>select a3</td>
<td>select b3</td>
<td>select z</td>
<td>load add1</td>
</tr>
<tr>
<td>5</td>
<td>select a4</td>
<td>select b4</td>
<td>select z</td>
<td>load add1</td>
</tr>
<tr>
<td>6</td>
<td>select a5</td>
<td>select b5</td>
<td>select z</td>
<td>load add1</td>
</tr>
<tr>
<td>7</td>
<td>select a6</td>
<td>select b6</td>
<td>select z</td>
<td>load add1</td>
</tr>
<tr>
<td>8</td>
<td>select a7</td>
<td>select b7</td>
<td>select z</td>
<td>load add1</td>
</tr>
</tbody>
</table>

Step 7b: Design the Controller

- In this case the controller becomes a simple 3-bit counter to control the muxes
- The register is always enabled

Step 8: Design Reset Mechanism

- Most systems require a reset mechanism
- This may be synchronous or asynchronous – this will be in the specification
- Most RTL designs are reset by resetting the controller
- The controller then resets the data path (as in this example)
- In this example, the reset simply puts the counter in its start state (count = 0)

VHDL Description

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package cross_product_types is
    type sig8_vector is array (natural range <>) of sig8;
signal i : unsigned(2 downto 0);
signal ai, bi : sig8;
signal product, add_in, sum, accumulator : sig16;
end package cross_product_types;

architecture RTL of cross_product is
    begin
        if reset = '1' then
            i <= "000";
        else
            i <= i + 1;
        end if;
end synthesis;
```

library ieee;
use ieee.std_logic_1164.all;
use work.cross_product_types.all;

entity cross_product is
    port (a, b : in sig8_vector(7 downto 0);
            ck, reset : in std_logic;
            z : out sig16);
end entity cross_product;

architecture RTL of cross_product is
    begin
        process (ck)
        begin
            if ck'event and ck = '1' then
                if i = 0 then
                    z <= X"0000";
                else
                    multiplier <= ai * bi;
                    add <= add_in + multiplier;
                    accumulator <= add;
                end if;
            end if;
        end process;
    end architecture;
```
More Complex Schedules

- e.g. Zwolinski p197 (simplified)

```vhdl
input_sum := input;
for j in 0 to order-1 loop
    input_sum := input_sum + delay(j) * ceoffb(j);
end loop;
output_sum := input_sum * coeffa(order);
for k in 0 to order loop
    output_sum := output_sum + delay(k) * coeffa(k);
end loop;
output := output_sum;
```

- for order = 1

```vhdl
input_sum := input;
input_sum := input_sum + delay(0) * ceoffb(0);
output_sum := input_sum * coeffa(1);
output_sum := output_sum + delay(0) * coeffa(0);
output_sum := output_sum + delay(1) * coeffa(1);
output := output_sum;
```

Single Assignment Form

```vhdl
input_sum := input;
input_sum := input_sum + delay(0) * ceoffb(0);
output_sum := input_sum * coeffa(1);
output_sum := output_sum + delay(0) * coeffa(0);
output_sum := output_sum + delay(1) * coeffa(1);
output := output_sum;
```

- 3 adds
- 4 multiplies

Data Dependency Graph

ASAP Schedule

- 1 adder
- 3 multipliers
**ALAP Schedule**

- Cycle 1: *  
- Cycle 2: +  
- Cycle 3: *  
- Cycle 4: *  
- Cycle 5: +  

- 1 adder  
- 2 multipliers

**Resource Constrained Schedule**

- Cycle 1: *  
- Cycle 2: +  
- Cycle 3: *  
- Cycle 4: *  
- Cycle 5: +  

- 1 adder  
- 1 multiplier

**Allocation**

- Cycle 1: *  
- Cycle 2: +  
- Cycle 3: *  
- Cycle 4: *  
- Cycle 5: +  

- 1 adder  
- 2 multipliers

**Allocate Registers**

- Cycle 1: *  
- Cycle 2: +  
- Cycle 3: *  
- Cycle 4: *  
- Cycle 5: +  

- 1 adder  
- 2 multipliers
### Share Registers

- 5 Registers

### Data Path

### Controller State Machine

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>mux1 = 0</td>
<td>mux5 = 0</td>
<td>mux1 = 1</td>
<td>mux5 = 1</td>
<td>mux5 = 2</td>
</tr>
<tr>
<td>mux2 = 0</td>
<td>mux6 = 0</td>
<td>mux2 = 1</td>
<td>mux6 = 1</td>
<td>mux6 = 2</td>
</tr>
<tr>
<td>mux3 = 0</td>
<td>mux3 = 1</td>
<td>enable R6</td>
<td>enable R9</td>
<td>enable R11</td>
</tr>
<tr>
<td>mux4 = 0</td>
<td>mux4 = 1</td>
<td>enable R3</td>
<td>enable R7</td>
<td>enable R10</td>
</tr>
<tr>
<td>enable R1</td>
<td>enable R4</td>
<td>enable R8</td>
<td>enable R5</td>
<td></td>
</tr>
</tbody>
</table>