EE-334
Digital System Design

Custom Digital Circuits

From Algorithms to Architectures

Andreas Burg
(Alain Vachoux)

Starting from Specifications as Sequential Code

- **Algorithms are** usually **specified** as **mathematical expressions** or **as sequential programs**
- Mathematical representation must be written with care to avoid ambiguities
- Representation as sequential programs
  - usually remove already significant amounts of overhead (re-use)
  - often imply a step by step execution on few shared computing resources
    - Longer programs generally take only more time on a sequential processor
    - Sequential execution may resolve ambiguities (imply priority)
- Example algorithm

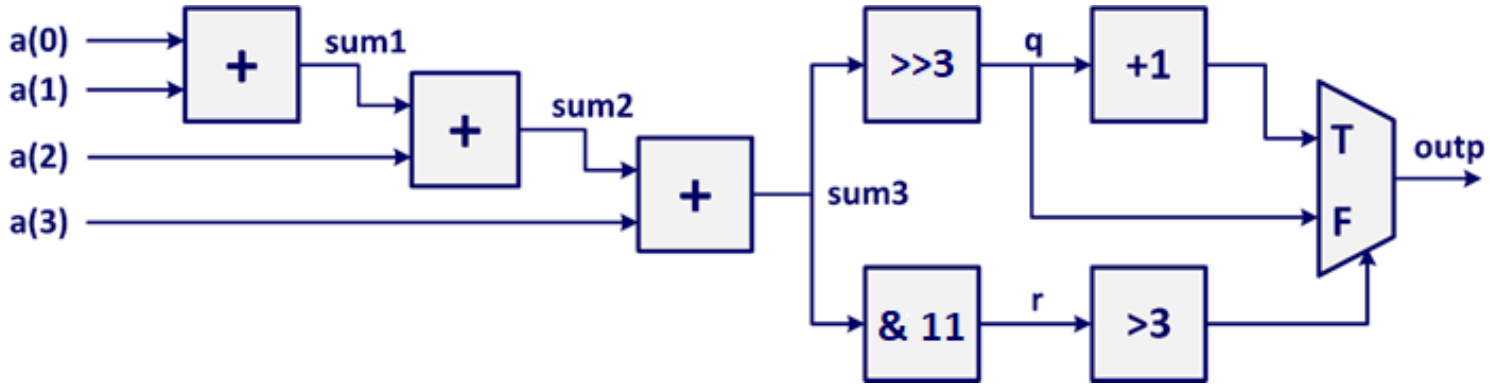
$$q = \begin{cases} \frac{1}{8}\sum_{i=0}^{3} a[i] + 1 & \mod_8\left(\sum_{i=0}^{3} a[i]\right) > 3 \\ \frac{1}{8}\sum_{i=0}^{3} a[i] & else \end{cases}$$

```
int sigproc(int a[]) {
   int q, r;
   int sum = a[0];
   for (int i=1; i<=3;i++) {
      sum += a[i];
   }
   q = sum/8;
   r = sum % 8;
   if (r > 3) q += 1;
   return q;
}
```

Isomorphic Architecture

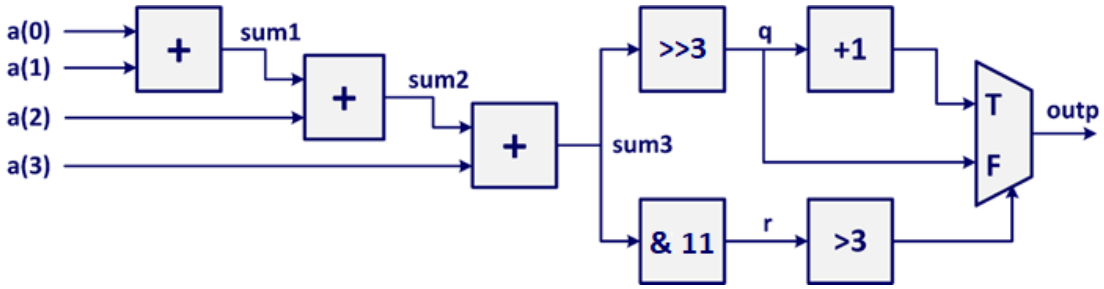
- **Isomorphic architecture:** straightforward mapping of an algorithm to hardware
  - Corresponds to the data flow graph of the algorithm
    - Vertices correspond to operations, edges route intermediate results to other vertices
  - **Every operation** is **mapped to a dedicated** combinational hardware **unit**
    - **Loops are unrolled**: loop counter must be computable (known) at design time
    - **Conditional statements become multiplexers**

```
int sigproc(int a[]) {
   int q, r;
   int sum = a[0];
   for (int i=1; i<=3;i++) {
      sum += a[i];
   }
   q = sum/8;
   r = sum % 8;
   if (r > 3) q += 1;
   return q;
}
```



Isomorphic Architecture in VHDL

- VHDL offers multiple **options to describe an isomorphic architecture**:



Sequential code in process statements

```
process (a) is
      variable acc, q, r : unsigned(7 downto 0);
   begin
      acc := unsigned(a(0));
      for i in 1 to 3 loop
         acc := acc + unsigned(a(i));
      end loop;
      q := "000" & acc(7 downto 3);  -- /8
      r := "00000" & acc(2 downto 0);  -- rem 8
      if r > 3 then
         q := q + 1;
      end if;
      outp <= std_logic_vector(q);
   end process;
```

Explicit modelling with concurrent statements

```
architecture dfl of sigproc is
   signal sum0, sum1  : unsigned(7 downto 0);
   signal sum2, sum3, q, r : unsigned(7 downto 0);
begin
   sum0 <= unsigned(a(0));
   sum1 <= sum0 + unsigned(a(1));
   sum2 <= sum1 + unsigned(a(2));
   sum3 <= sum2 + unsigned(a(3));
   q <= "000" & sum3(7 downto 3);  -- /8
   r <= "00000" & sum3(2 downto 0);  -- rem 8
   outp <= std_logic_vector(q + 1) when r > 3 else
           std_logic_vector(q);
end architecture dfl;
```

Isomorphic Architecture in VHDL

- VHDL offers multiple **options to describe an isomorphic architecture**:

- **Sequential code in process statements**
  - **Compact code** is interpreted and **expanded automatically** in the elaboration phase of the hardware synthesis
    - Loops are unrolled automatically
    - Hardware resources are automatically allocated and connected
  - Variables are used instead of signals to pass results between sequential statements
    - **Variables** are similar to signals, but are assigned immediately and **can be re-used**. They only exist until the end of the process.

```
process (a) is
      variable acc, q, r : unsigned(7 downto 0);
   begin
      acc := unsigned(a(0));
      for i in 1 to 3 loop
         acc := acc + unsigned(a(i));
      end loop;
      q := "000" & acc(7 downto 3);  -- /8
      r := "00000" & acc(2 downto 0);  -- rem 8
      if r > 3 then
         q := q + 1;
      end if;
      outp <= std_logic_vector(q);
   end process;
```

- **Convenient, but dangerous approach : IDEALLY DO NOT USE!**
  - Many pitfalls for code that can not be synthesized (e.g., non-constant loop counters)
  - Often hides complexity: loops do not imply reuse of resources!!!

Isomorphic Architecture in VHDL

- VHDL offers multiple **options to describe an isomorphic architecture**:

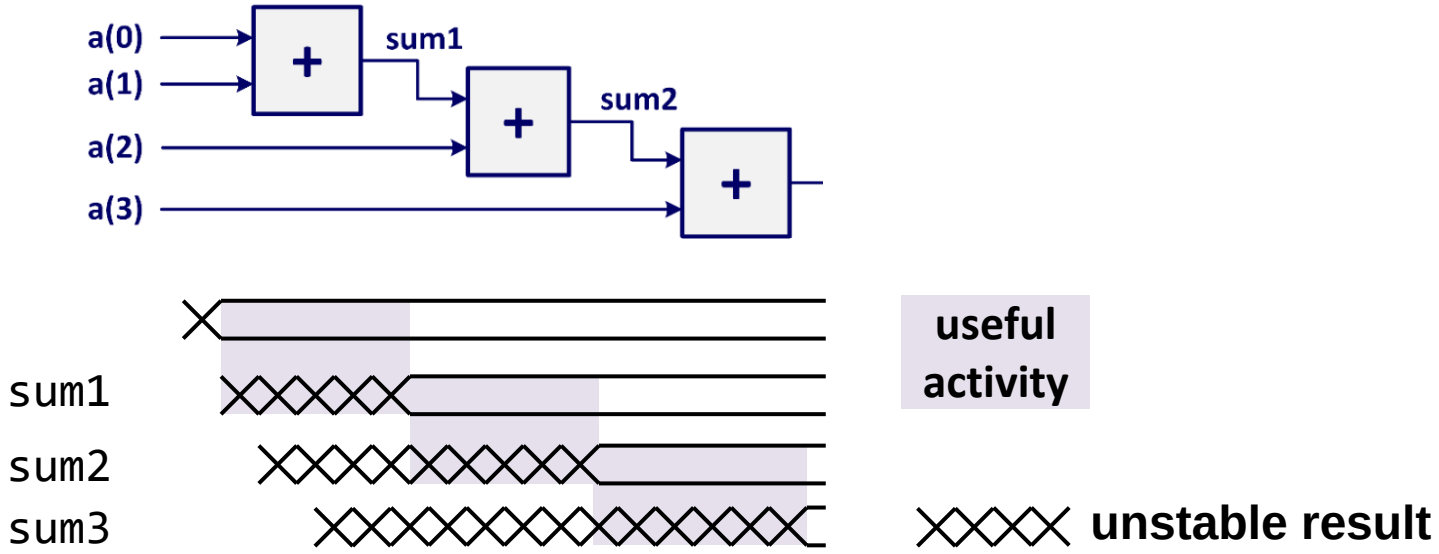
- **Explicit modelling with concurrent statements**
  - Sequential code must be expanded manually
  - Components connected with signals
    - Need for many signals that must be defined explicitly
  - Does not use many of the features that render sequential code compact

```
architecture dfl of sigproc is
   signal sum0, sum1  : unsigned(7 downto 0);
   signal sum2, sum3, q, r : unsigned(7 downto 0);
begin
   sum0 <= unsigned(a(0));
   sum1 <= sum0 + unsigned(a(1));
   sum2 <= sum1 + unsigned(a(2));
   sum3 <= sum2 + unsigned(a(3));
   q <= "000" & sum3(7 downto 3);  -- /8
   r <= "00000" & sum3(2 downto 0);  -- rem 8
   outp <= std_logic_vector(q + 1) when r > 3 else
           std_logic_vector(q);
end architecture dfl;
```

- **A bit more tedious to write, but much more close to the architecture**
  - All statements map directly to corresponding hardware

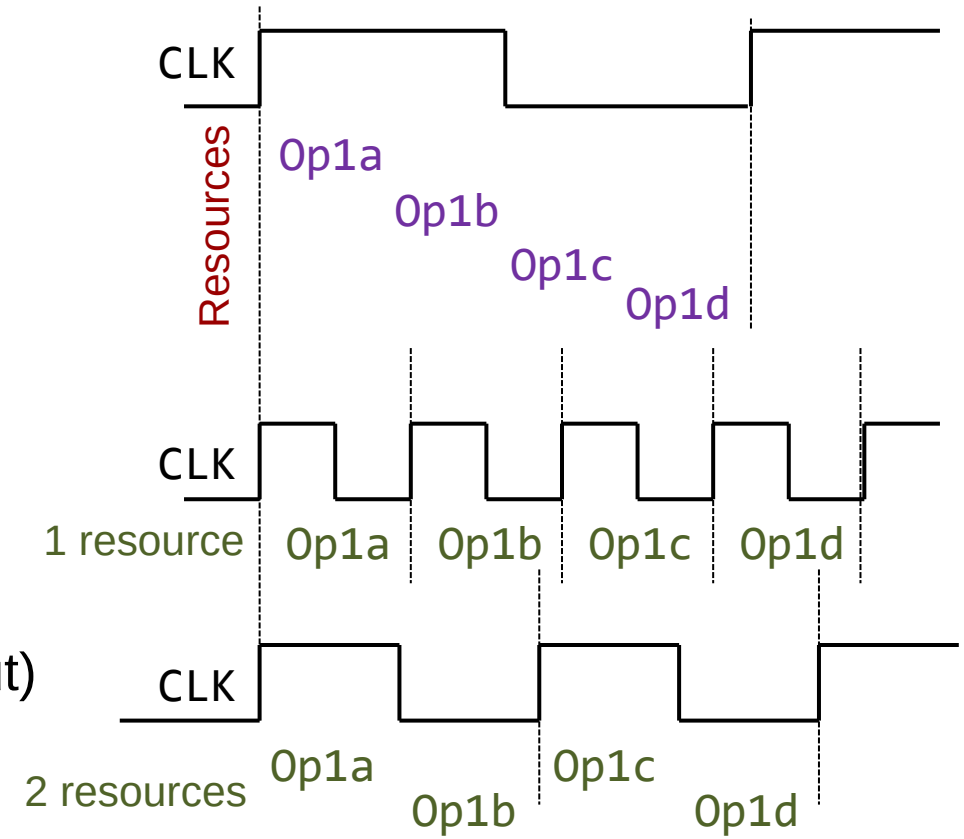
Efficiency of Combinational Circuits

- **Combinational circuits are** actually **very inefficient:**
  - **Every element has** only **a very small** delay (**time in which it performs** useful **computations**)
  - Synchronous paradigm requires **every gate** to **wait almost 1 complete cycle** for the next input
- Useful activity propagates like a wave through the combinational circuit



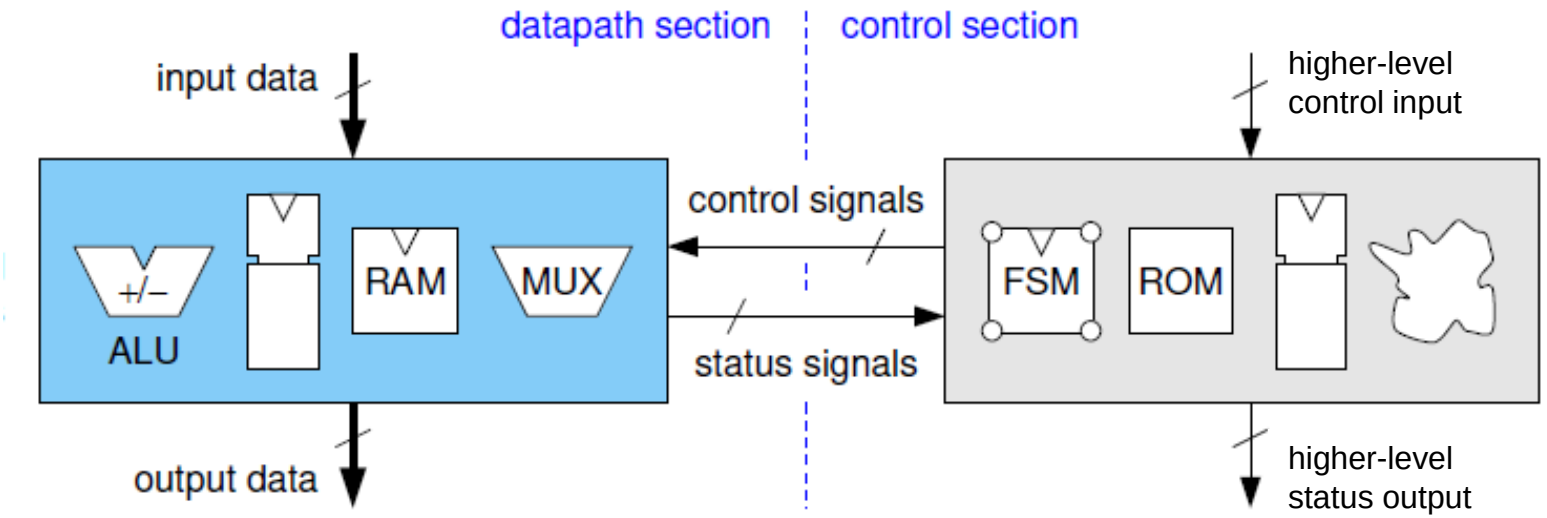
Sequential Processing in Custom Hardware

- Single-cycle (**combinational**) **processing** is **not efficient and not flexible**
  - Algorithm defines both circuit complexity and computation time (throughput)
- **Multi-cycle** computation **architecture** enables
  - Decomposition of computations into multiple steps
  - Re-use of computation resources between the steps
  - Flexibility in scheduling operations in time and binding them to different resources
- **Distribution of computations** over multiple cycles (time steps) **provides more flexibility** to trade time for complexity (area)
  - Ability to design for given requirements (area or throughput)
  - Often better overall efficiency (AT-product)



Sequential Processing in Custom Hardware

- **Separation between computations and control**
  - **Datapath** : performs (arithmetic or other) operations on data and keeps intermediate resultsin memory elements. Control inputs (from control logic) define what should be done in each cycle
  - **Control** : controls datapath and manages the sequence of operations (potentially based on feedback from the datapath), but does not manipulate data itself

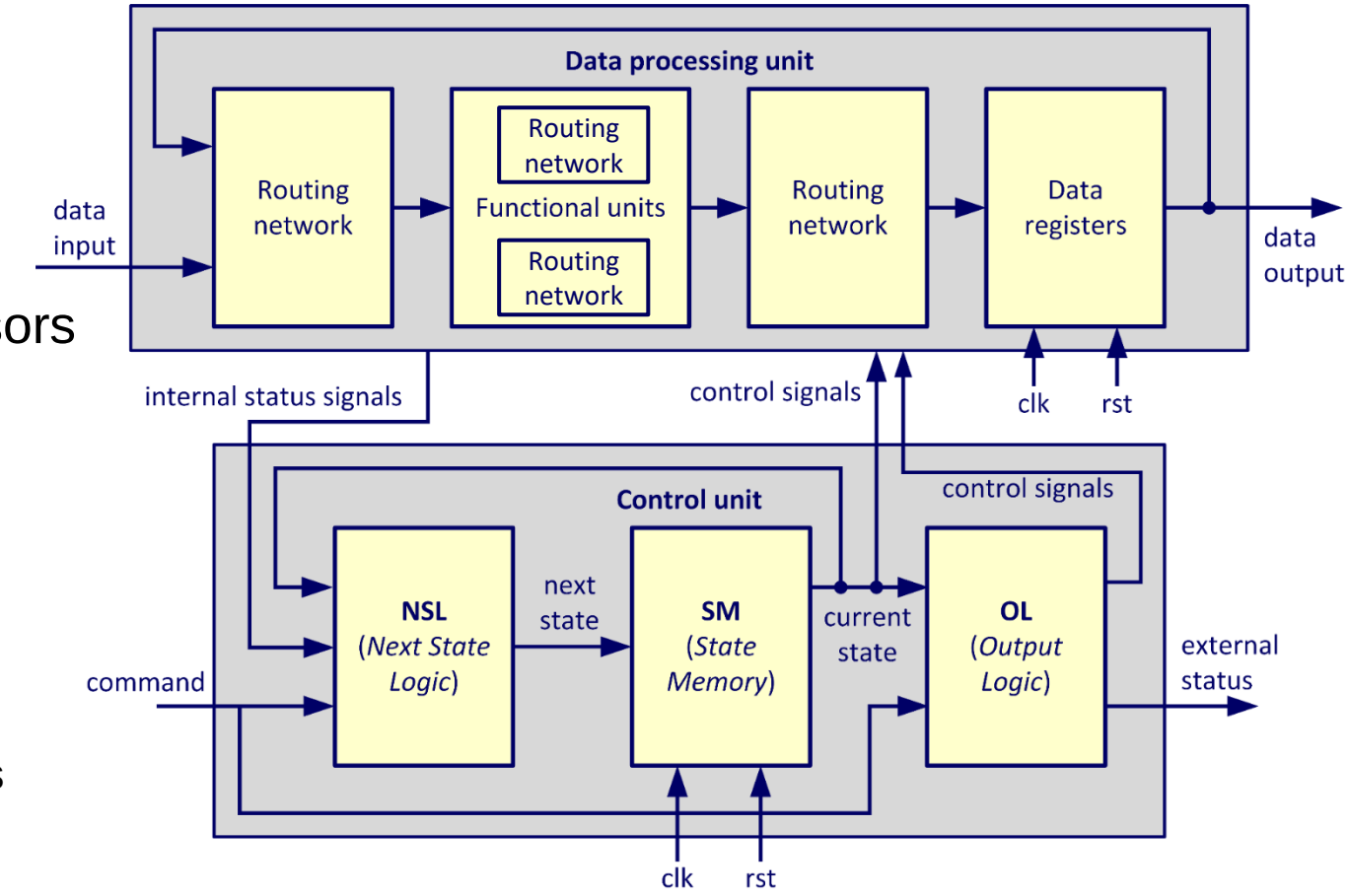


data processing (arithmetic) units, data switches, random logic for data manipulation, data storage

Finite state machines, (programmable or hardwired) instruction sequences, counters, hardwired control logic

Sequential Processing in Custom Hardware

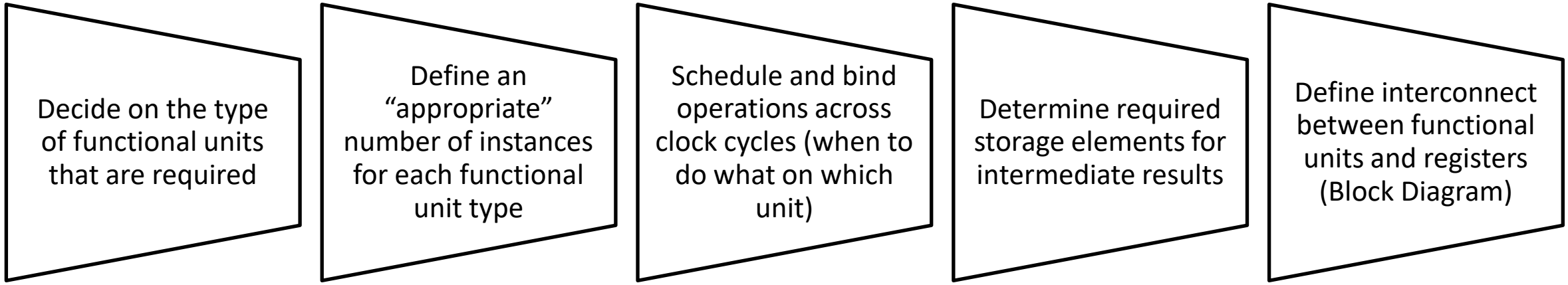
- Architectural template follows the RTL principle
  - **FSM** for control with connections to and from the datapath
  - **Datapath**: organized similar to processors
    - Storage elements keep partial results for subsequent clock cycle
    - Functional units perform computations
      - Reconfigurable with control signals
    - Routing networks connect
      - Functional units and storage elements
      - Functional units with other functional units



Datapath Design Procedure

- **Datapath determines complexity and throughput**
  - **Many degrees of freedom** render finding optimal the design difficult

- **Datapath design in five steps**



  - **Leads to an RTL diagram** of the datapath with various control inputs

- **Control path:** follows from the design of the datapath

Design Procedure Example

- **Step-1**: identify **functional units**
  - Adder, Div-by-8 (shift-right-3), Modulo-8 (extract 3 LSBs) Compare to 3, Add-1

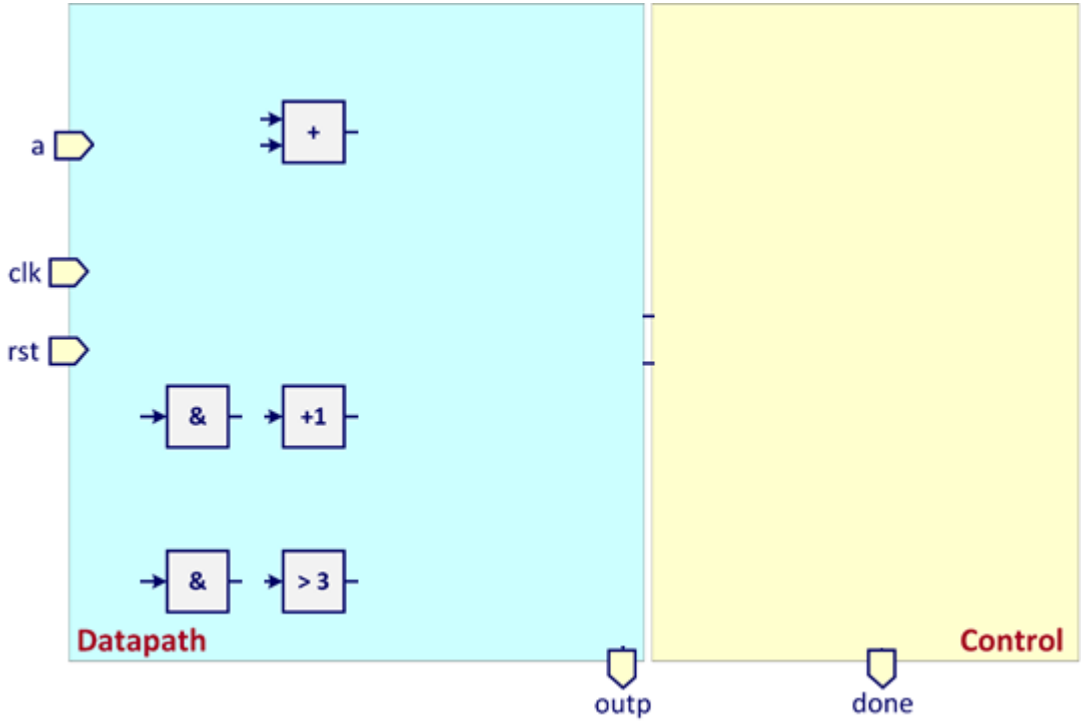
```
int sigproc(int a[]) {
   int q, r;
   int sum = a[0];
   for (int i=1; i<=3;i++) {
      sum += a[i];
   }
   q = sum/8;
   r = sum % 8;
   if (r > 3) q += 1;
   return q;
}
```

Design Procedure Example

- **Step-1**: identify **functional units**
  - Adder, Div-by-8 (shift-right-3), Modulo-8 (extract 3 LSBs)
    Compare to 3, Add-1

- **Step-2**: define **hardware resources**
  - **One** instance **for each type** of resource (max reuse)

```
int sigproc(int a[]) {
   int q, r;
   int sum = a[0];
   for (int i=1; i<=3;i++) {
      sum += a[i];
   }
   q = sum/8;
   r = sum % 8;
   if (r > 3) q += 1;
   return q;
}
```



Design Procedure Example

- **Step-1**: identify **functional units**
  - Adder, Div-by-8 (shift-right-3), Modulo-8 (extract 3 LSBs) Compare to 3, Add-1
- **Step-2**: define **hardware resources**
  - **One** instance **for each type** of resource (max reuse)
    - **Can be used only once in each cycle**
- **Step-3**: schedule operations across cycles and bind to hardware resources
  - For-loop in cycles 0-3, update of q in cycle 4
  - Binding with 1 operation
- **Step-4**: determine storage elements
  - Registers (2) for interediate sum and redult q

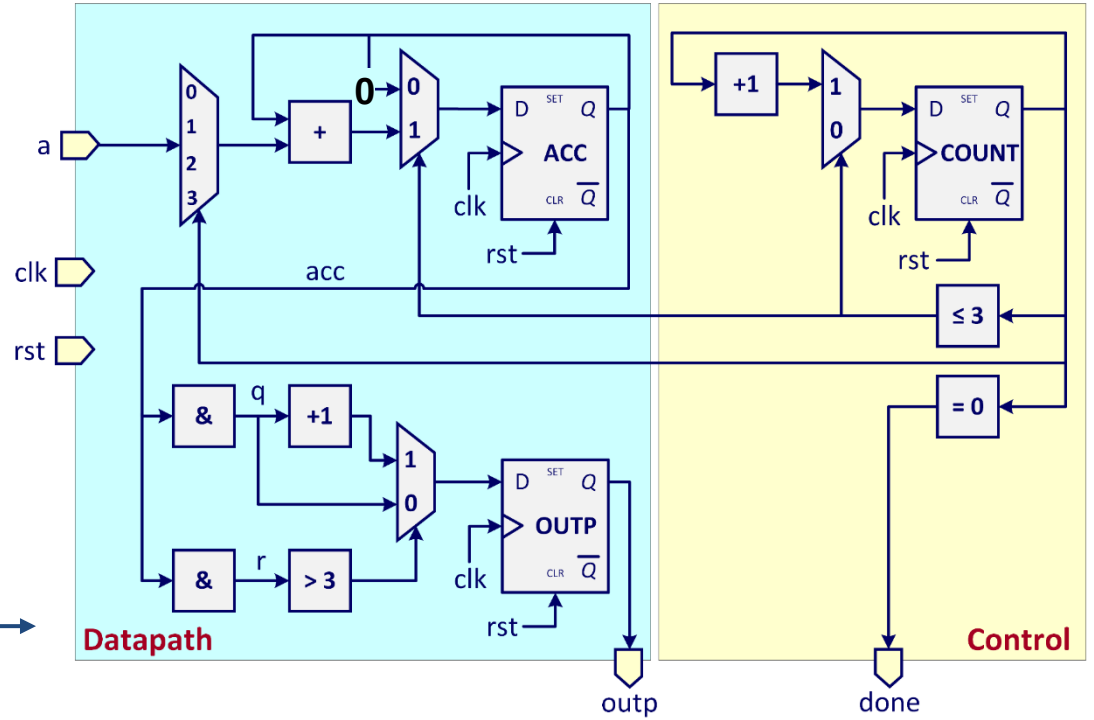
```
int sigproc(int a[]) {
   int q, r;
   int sum = a[0];
   for (int i=1; i<=3;i++) {
      sum += a[i];
   }
   q = sum/8;
   r = sum % 8;
   if (r > 3) q += 1;
   return q;
}
```

#	Operations	REG
0	sum_nxt=0+a[0]	sum
1	sum_nxt=sum+a[1]	sum
2	sum_nxt=sum+a[2]	sum
3	sum_nxt=sum+a[3]	sum
4	q=sum/8; q'=q+1; r=sum%8; cond=r>3 out_nxt=sel(q,q';cond)	out

Datapath Design Procedure Example

```
int sigproc(int a[]) {
   int q, r;
   int sum = a[0];
   for (int i=1; i<=3;i++) {
      sum += a[i];
   }
   q = sum/8;
   r = sum % 8;
   if (r > 3) q += 1;
   return q;
}
```

- **Step-1**: identify **functional units**
  - Adder, Div-by-8 (shift-right-3), Modulo-8 (extract 3 LSBs) Compare to 3, Add-1
- **Step-2**: define **hardware resources**
  - **One** instance **for each type** of resource (max reuse)
- **Step-3**: schedule operations across cycles and bind to hardware resources
  - For-loop in cycles 0-3, update of q in cycle 4
  - Binding with 1 operation
- **Step-4**: determine storage elements
  - Registers (2) for interediate sum and redult q
- **Step-5**: define interconnect (MUXes)



Example Implementation in VHDL

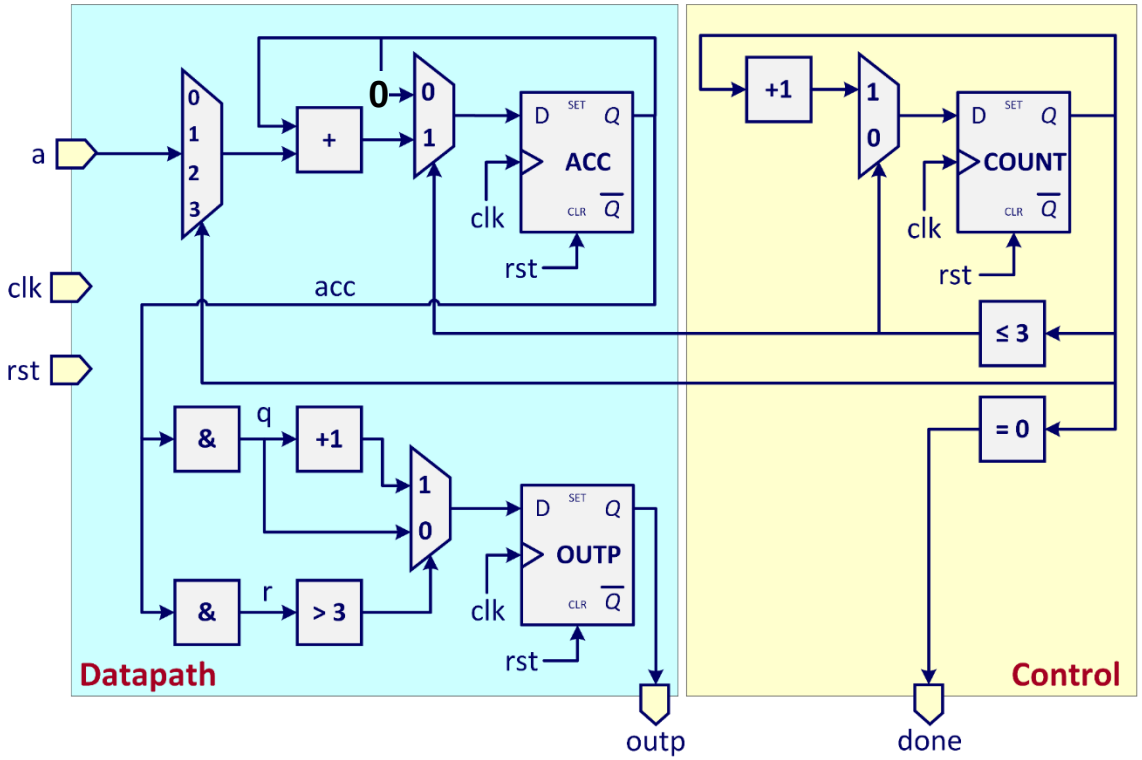
```
entity sigproc is
   port (
      signal clk  : in  std_logic;
      signal rst  : in  std_logic;
      signal a    : in  std_logic_vector(31 downto 0);
      signal outp : out std_logic_vector(7 downto 0)
      signal done : out std_logic);
end entity sigproc;
```

```
architecture rtl of sigproc is

   -- for structuring the input a
   type au_vector is array (0 to 3) of unsigned(7 downto 0);

   signal acc_next, acc_reg     : unsigned(7 downto 0);
   signal outp_next, outp_reg   : unsigned(7 downto 0);
   signal count_next, count_reg : unsigned(2 downto 0);
   signal q, r                  : unsigned(7 downto 0);
   signal au                    : au_vector;

...continued next slide...
```



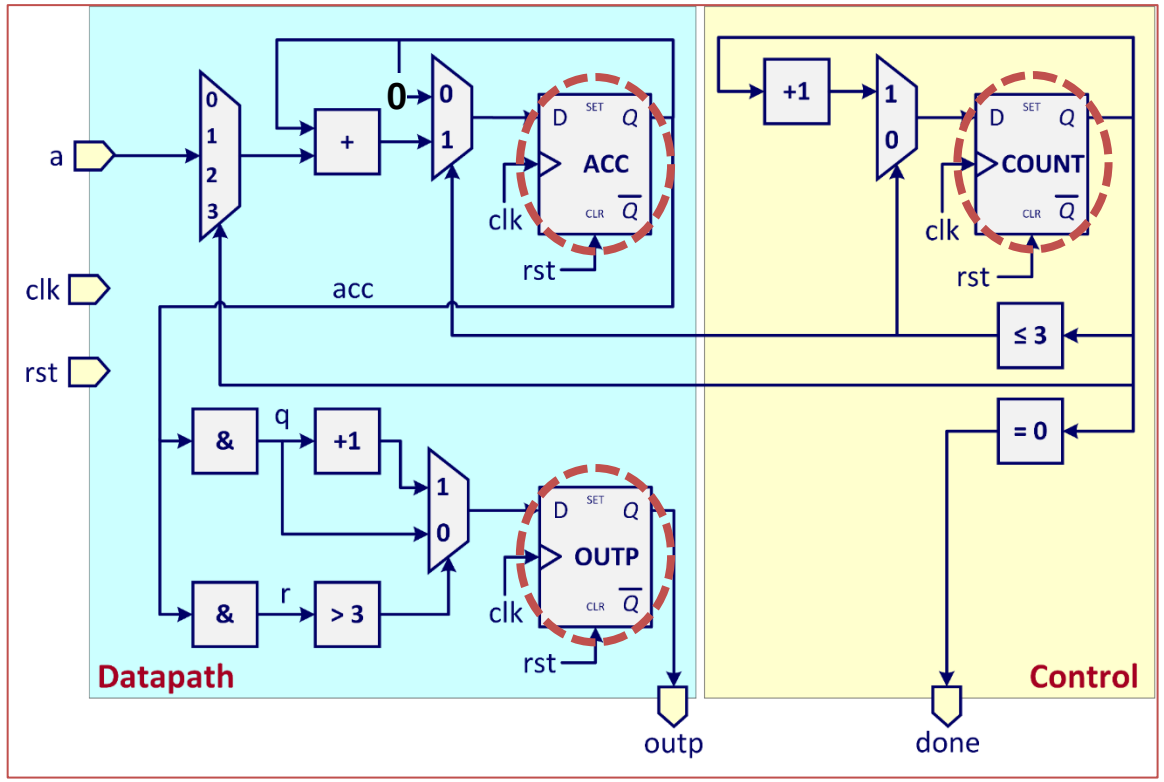
Example Implementation in VHDL

```
...continued from previous slide...

begin  -- architecture rtl

   REG : process (clk, rst) is
   begin
      if rst = '1' then
         acc_reg   <= (others => '0');
         outp_reg  <= (others => '0');
         count_reg <= (others => '0');
      elsif rising_edge(clk) then
         acc_reg   <= acc_next;
         outp_reg  <= outp_next;
         count_reg <= count_next;
      end if;
   end process REG;

...continued next slide ...
```



Example Implementation in VHDL

```
...continued from previous slide...

   count_next <= count_reg + 1 when count_reg <= 3 else
                 (others => '0');
   done <= '1' when count_reg = 0 else
           '0';

   A_SPLIT : for i in 0 to 3 generate
      au(i) <= unsigned(a((i+1)*8-1 downto i*8));
   end generate A_SPLIT;

   acc_next <= acc_reg + au(to_integer(count_reg))
                              when count_reg <= 3 else
               (others => '0');

   q <= "000" & acc_reg(7 downto 3);  -- /8
   r <= "00000" & acc_reg(2 downto 0);  -- % 8
   outp_next <= q + 1 when r > 3 else
                q;

   outp <= std_logic_vector(outp_reg);
end architecture rtl;
```