VHDL
Miscellaneous

EL 310

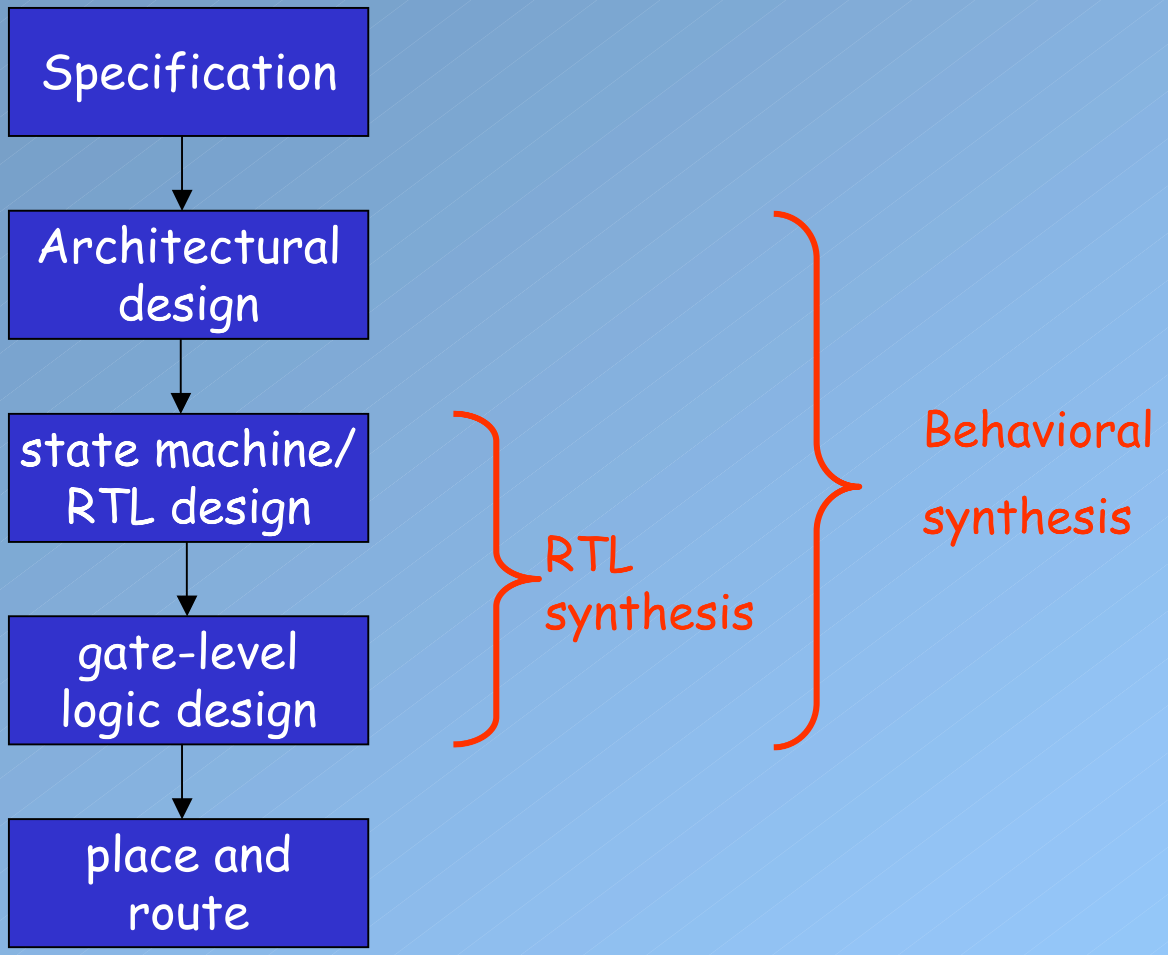
Erkay Savaş

Sabancı University

VHDL as a High-Level Synthesis Tool

- VHDL is proposed to model the behavior of the existing hardware
- VHDL was not originally intended for automatic synthesis of hardware.
- In 1987, when the VHDL standard was written, there were no automatic synthesis tools in widespread use.
- The meanings of some VHDL constructs in hardware are derived later.
- As a consequence, some parts of VHDL are not suitable for synthesis.
- Aim is to fully automate the design process using high-level behavioral model.

VHDL as a High-Level Synthesis Tool



RTL Synthesis vs. Behavioral Synthesis

- RTL synthesis tools take a VHDL description of a design in terms of registers, state machines and combinational logic functions and generate a netlist of gates and library cells.
- Behavioral synthesis tools take algorithmic VHDL models and transform them to gates and cells.
  - For example,there is no need to specify clock inputs
  - one may simply specify a certain time interval in which a particular operation is completed.
- RTL synthesis tools are in widespread use while behavioral synthesis tools are rare.

RTL Synthesis

- Starting point is a model described in terms of combinational and sequential building blocks and state machines.
- We have to know all the inputs and outputs of the system, including the clock and resets.
- We may also have to know the number of states in state machines
  - RTL synthesis tools, in general, do not perform state minimization.
- Constraints can be specified
  - particular form of state encoding, area and speed constraints, etc.
  - Constraints are not part of VHDL and unique to particular tools, but some of them may be included in the VHDL description.

RTL Synthesis

- Three styles of VHDL

1. Structural
2. Dataflow (CSA)
   - was originally intended for RTL modeling,
   - most of dataflow constructs are synthesizeable.
3. Behavioral
   - intended for high-level, algorithmic modeling

IEEE 1076.6-1999 Standard

- It defines a subset of VHDL for synthesis
- The purpose is a minimum subset that can be accepted to <u>any</u> synthesis tool.
- Certain constructs, for example delays and floating-point operators are not synthesizeable.
  - Floating-point data types will be rejected by synthesis tools because they require at least 32 bits and the hardware required for many operations is too large for most ASICs and FPGAs.

Non-synthesizeable VHDL

- The following VHDL constructs are either ignored or rejected by RTL synthesis tools
- **`after`**, **`transport`**, **`inertial`** keywords.
  - A model can be synthesized to meet various constraints, but cannot be synthesized to meet some exact timing model.
  - For example, it is not possible to specify that a gate will have a delay of exactly 5 ns.
  - But it is reasonable to require synthesis tool to generate a block of combinational logic such that its total delays is less than, say 20 ns.

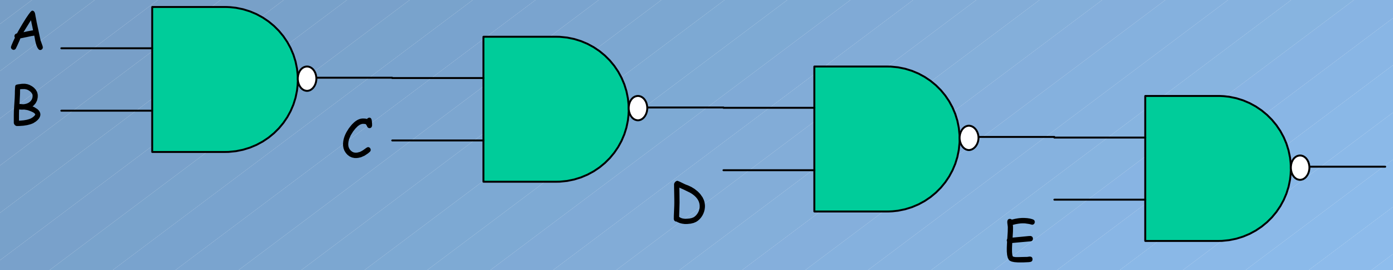
Non-synthesizeable VHDL

- **`wait for`** construct is also ignored
  - we may not be able to built a piece of hardware whose delay is exactly specified.
- **`File`** operations
- default values must be specified for parameterized models.
- pointers, specified by the **access** keyword are not recognized by synthesis tools.
  - high-level data structures, such as linked lists and trees, that is possible with pointers are not, therefore, supported.
- Initial values of signals and variables are ignored.

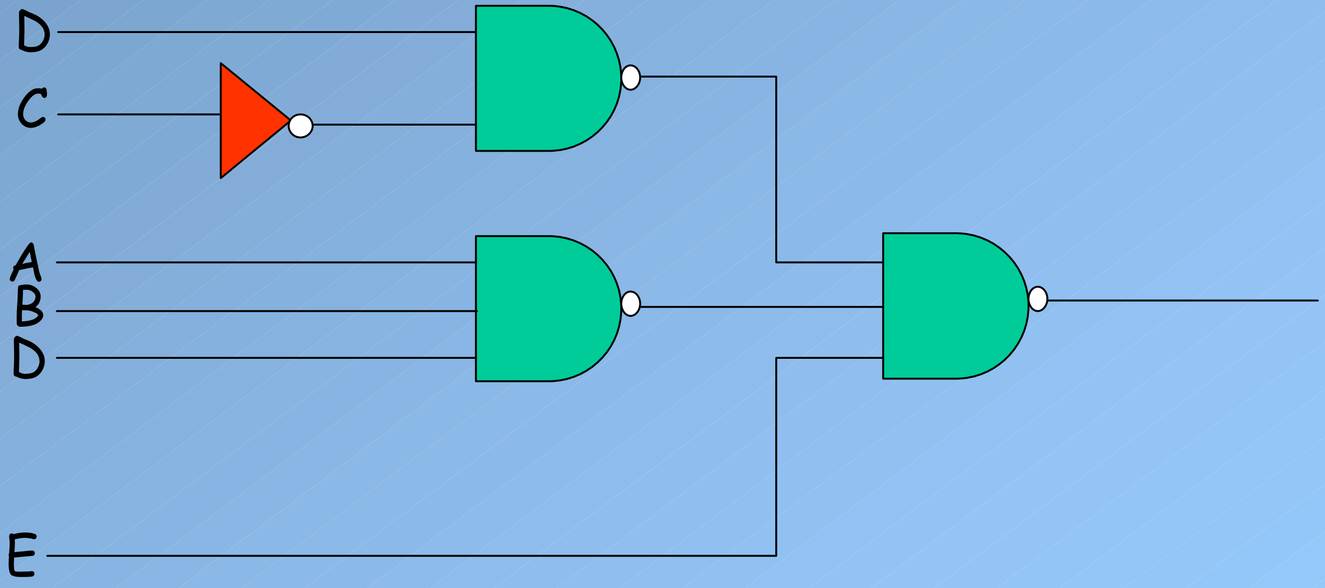
Constraints

- For any non-trivial digital function, there exist a number of alternative implementations.
- What we want is a infinitely fast, infinitesimally small, and totally testable system that consumes no power.
- The designer should decide the objectives
- These objectives are expressed to synthesis tools as <u>constraints</u>.
  - a design must fit on a particular FPGA, has to operate at a particular clock frequency, etc.
  - But sometimes, these objectives may not be met.

Constraints



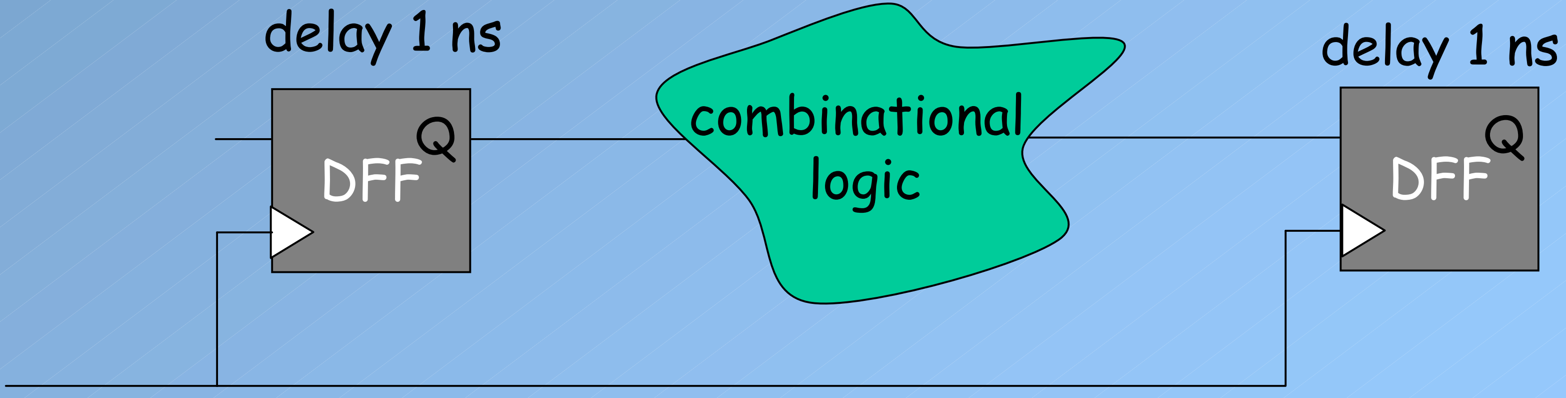
Implementation 1: 16 transistor, 4 gate delays



Implementation 2: 18 transistor, 3 gate delays

Timing Constraints

- Assume we want a circuit to operate synchronously with a clock at a particular frequency, say 20 MHz
  - Then, the maximum delay through the state registers and the next state logic must be at most 50 ns.
  - A constraint can be expressed as the clock frequency or maximum delay through the combinational logic.



clk (clock frequency is 50 MHz, maximum delay through combinational logic is 48 ns)

Timing Constraints

- What synthesis tool does is
  - delay through combinational logic can only be estimated.
  - exact delay depends on how the combinational logic is laid out.
  - Synthesis is performed using an estimate of the likely delays
  - In the end, the design objectives may not be met.
  - Synthesis must be restarted.
  - to speedup the hardware more operations are performed concurrently, which means that the design is larger.

Synthesis for FPGAs

- Xilinx and Actel FPGAs
- Xilinx is based on Static RAM technology,
  - volatile
  - it has tri-state buffers
- Actel is based on antifuse technology.
  - A connection is normally open circuit, but the application of a suitably large voltage causes a short circuit to be formed.
  - not reversible.
  - no tri-state buffers

Synthesis for FPGAs

- Both FPGA technologies feature high ratio of flip-flops to combinational logic.
  - emphasis is not on minimizing the number of flip-flops, but the combinational logic surrounding them.
  - One-hot state encoding is popular, particularly for relatively small state machines.
- A single asynchronous set or reset is the most efficient way of initializing all the flip-flops in an FPGA device
  - Each flip-flop has an asynchronous set and reset, but only only one of these may be used at one time.

Synthesis for FPGAs

- In both technologies, the flip-flops are edge-sensitive
  - level-sensitive latches have to be synthesized from combinational logic.
  - This can waste flip-flops and they are best avoided.
- It is desirable to instantiate predefined library components for certain functions.
  - It is safer (talk about it later)
  - Not only the logic is defined, but the configuration of logic blocks is already known
  - This potentially simplifies both RTL synthesis and place and route tasks.

Synthesis for FPGAs

- ACTEL technology does not support tri-state buffers,
  - language constructs that will result in tri-state buffer usage must be avoided.
- The two technologies have different limitations with respect to fan-outs.
  - In antifuse technology, one output can drive up to 16 inputs without degradation of the signal.
  - CMOS SRAM technology has higher fan-out limit.
  - A design that can easily be synthesized to a Xilinx technology may not be synthesized to an ACTEL FPGA without rewriting.

Fan-Out Limitations: Example

```
signal a, b: std_ulogic_vector(31 downto 0);
begin
  p0: process (enable, b) is
  begin
    if enable = ‘1’ then
      a <= b;
    else
      a <= (others => ‘0’);
    end if;
  end process
end;
```

- this simple VHDL code fragment may not be synthesized since enable signal is controlling 32 multiplexers.
- Instead, it must be split into two using buffers, and each buffered signal then controls half the bus.

Fan-Out Limitations: Example

```
signal a, b: std_ulogic_vector(31 downto 0);
signal en0, en1: std_ulogic;
begin
  b0: buf port map (enable, en0);
  b1: buf port map (enable, en1);
  p0: process (en0, en1, b) is
  begin
    if en0 = ‘1’ then a(15 downto 0) <= b (15 downto 0);
    else a (15 downto 0) <= (others => ‘0’);
    end if;
    if en1 = ‘1’ then a(31 downto 16) <= b (31 downto 16);
    else a (31 downto 16) <= (others => ‘0’);
    end if;
  end process
end;
```

- *A good synthesis tool should recognize the fan-out limits, and automatically insert buffers*

Asynchronous Sequential Design

- Why bother with asynchronous logic design?
  - power considerations
  - Due to ultra fast clocks (as a result high clock skews) systems may consist of synchronous islands that communicate asynchronously.
- However, it is better to avoid asynchronous structures
  - synthesis tools are intended for the design of synchronous systems, normally with a single clock.
- For example, the following concurrent VHDL construct

```
q <= d when c = '1' else q;
```

would be synthesized to an asynchronous circuit structure.

Asynchronous Sequential Design

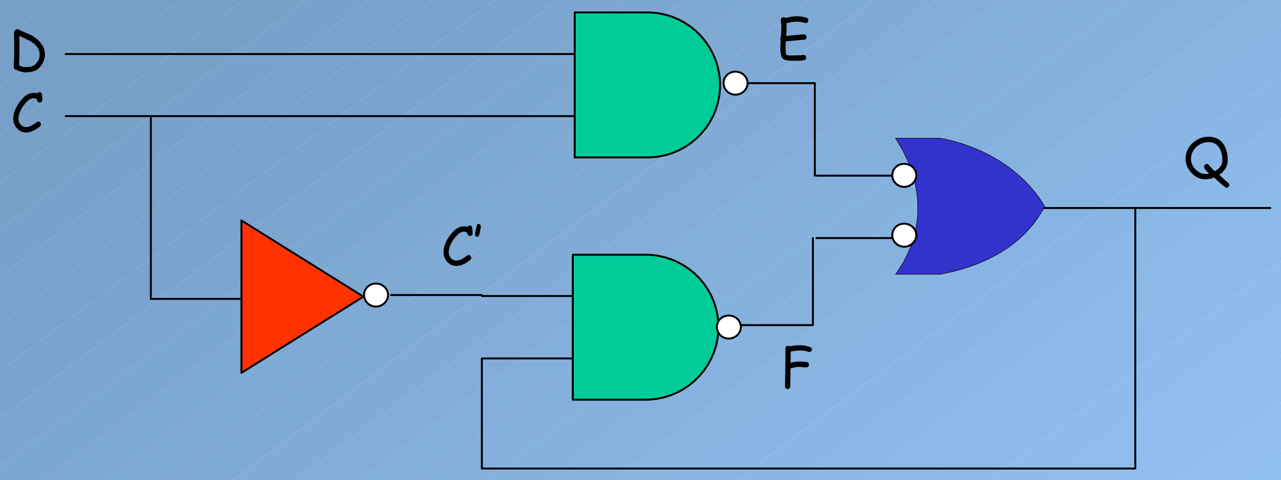
```
process (d, c) is
begin
  if c = ‘1’ then q <= d;
  end if;
end process;
```

- Similarly the above sequential block would also be synthesized to an asynchronous latch.
- But the circuit structures for these two constructs are not identical.
- In fact, a IEEE 1076.6 compliant synthesis tool would infer a latch from the incomplete **`if`** statement and use a latch from library.

Asynchronous Latch Inference

- The latch created by Boolean minimization (the first case) and the library latch is not the same.
- Indeed, the RTL synthesis standard, IEEE 1076.6, explicitly forbids the use of such concurrent statements
- It only permits the use of incomplete **`if`** and **`case`** statements.
- To see why, we have to look at a basic D latch and its timing characteristics.

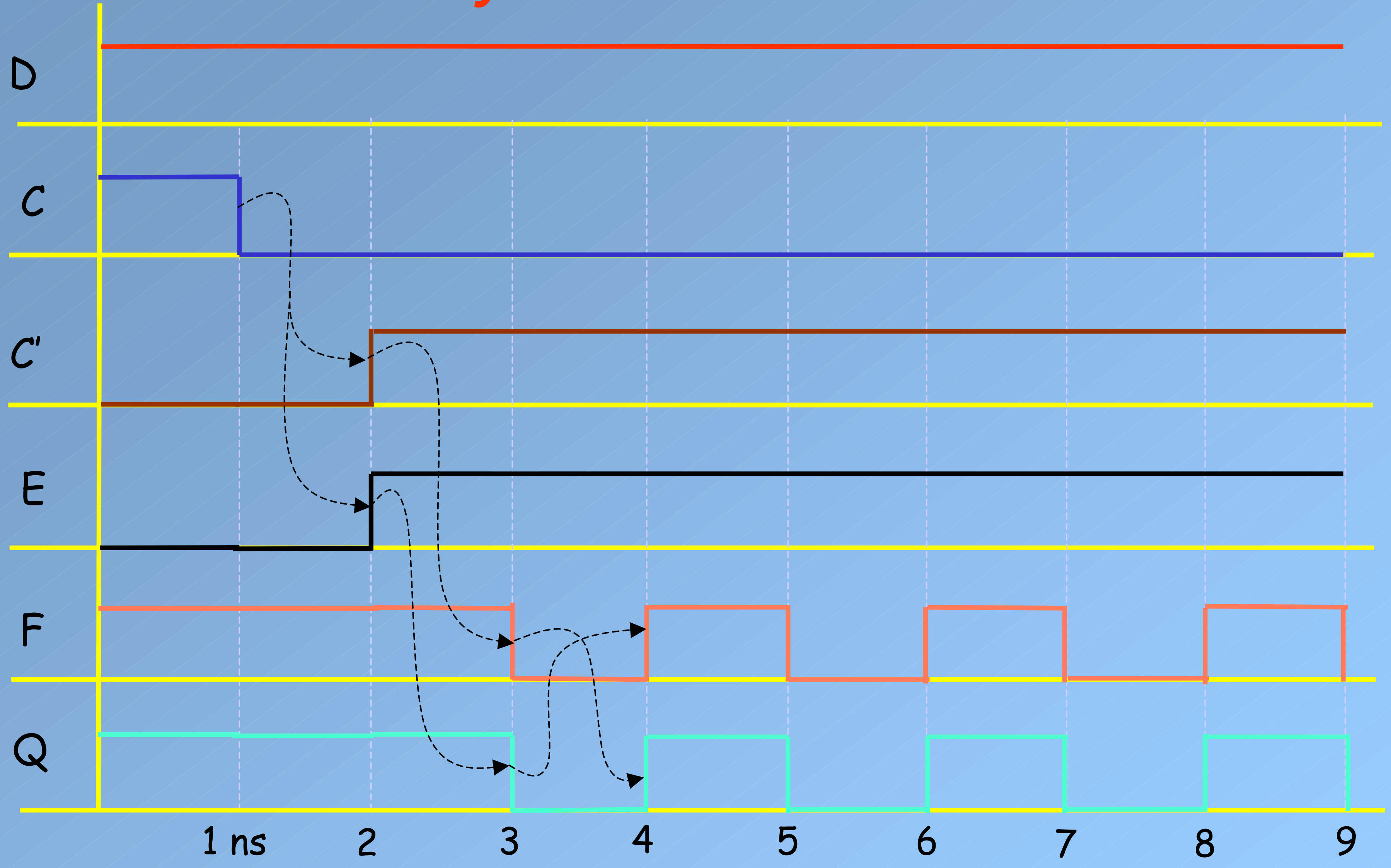
D-Latch



```
q <= (d and c) or (q and not c);
```

- Assume each gate has a delay of 1 unit time, say 1 ns.
- We will show that oscillatory behavior will occur.

Oscillatory Behavior of D-Latch



Gate Level Implementation of D-Latch

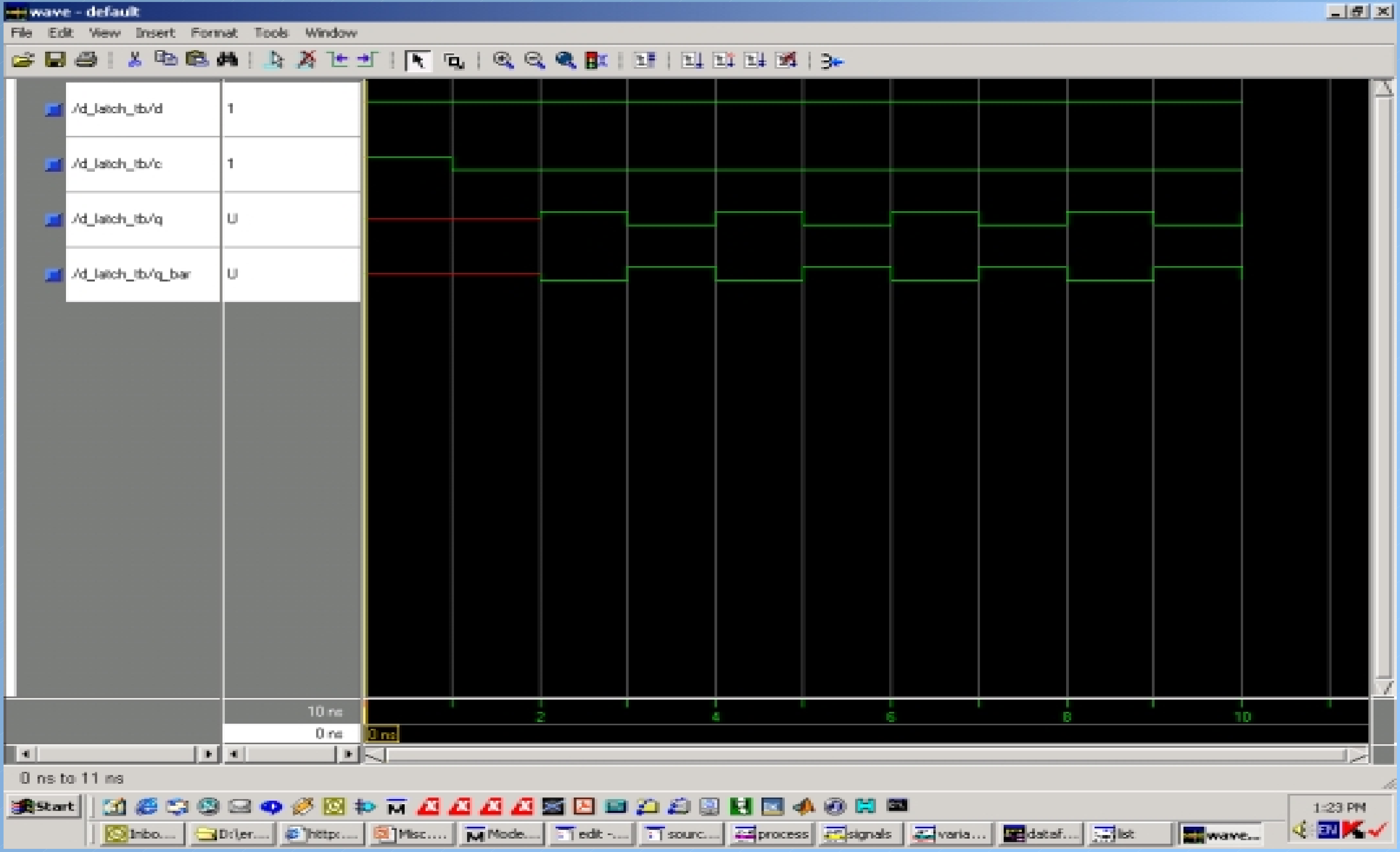
```
library ieee;
use ieee.std_logic_1164.all;
entity d_latch is
  generic (delay: Time := 1 ns);
  port(q: inout std_logic;
       q_bar: out std_logic;
       d, c: in std_logic);
end entity d_latch;
architecture concurrent of d_latch is
  signal e, f, c_bar: std_logic;
begin
  e <= d nand c after delay;
  c_bar <= not c after delay;
  f <= c_bar nand q after delay;
  q <= e nand f after delay;
  q_bar <= not q;
end architecture concurrent;
```

Test Bench for D-Latch

```
library ieee;
use ieee.std_logic_1164.all;
entity d_latch_tb is
end entity d_latch_tb;
architecture concurrent of d_latch_tb is
  component d_latch is
  generic (delay: Time := 1 ns);
  port(q: inout std_logic; q_bar: out std_ulogic;
       d, c: in std_ulogic);
end component d_latch;
  signal D, C, Q: std_ulogic := '1';
  signal Q_bar: std_ulogic := '0';
begin
  d0: d_latch port map(q => Q, q_bar => Q_bar, d => D, c => C);
  C <= '0' after 1 ns;
  Q_bar <= not Q;

end architecture concurrent;
```

Simulation Results



More Realistic Model for Timing

```
library ieee;
use ieee.vital_primitives.all;
...
architecture vtl of d_latch is
  signal e, f, c_bar: std_logic;
begin
  VitalNAND2(e, d, c, (delay, delay), (delay, delay));
  VitalINV(c_bar, c, (delay, delay));
  VitalNAND2(f, c_bar, q, (delay, delay), (delay, delay));
  VitalNAND2(q, e, f, (delay, delay), (delay, delay));
  q_bar <= not q;
end architecture vtl;
```

Behavioral Synthesis

- RTL synthesis requires that the design be specified in terms of register operations.
- Behavioral synthesis takes the process one stage further
  - the hardware to be synthesized in terms of an algorithm. from which, the registers and logic are derived.
- Main obstacle is the inefficient or infeasible hardware generated by the behavioral synthesis tools
  - with the decreasing cost of silicon, it seems safe to predict that behavioral synthesis will become in widespread usage in the future.

Behavioral Synthesis: Example

5th order infinite impulse response (IIR) filter

```
package iir_defs is
  constant precision: positive := 16;
  subtype int is integer range –2**(precision – 1) to
                              2**(precision –1) – 1;
  type integer_array is array (natural range <>) of int;
  constant order: positive := 5;
end package iir_defs;
```

IIR Filter Design

```
use work.iir_defs.all;
entity iir is
  generic(coeffa: integer_array(0 to order);
          coeffb: integer_array(0 to order-1));
  port(input: in int;
       strobe: in bit;
       output: out int);
end entity iir;
```

IIR Filter Design

```
architecture beh of iir is
begin
  process is
    variable input_sum, output_sum: int;
    variable delay:integer_array(0 to order):=(others => 0);
  begin
    input_sum := input;
    for j in 0 to order-1 loop
      input_sum := input_sum + (delay(j)*coeffb(j))/1024;
    end loop;
    output_sum := (input_sum*coeffa(order))/1024;
    for k in 0 to order-1 loop
      input_sum := input_sum + (delay(k)*coeffa(k))/1024;
    end loop;
...
```

IIR Filter Design

```
    ...
    for m in 0 to order-1 loop
      delay(m) := delay(m+1);
    end loop;
    delay(order) := input_sum;
    output <= output_sum;
    wait on strobe;
  end process;
end architecture beh;
```

- A C version of the algorithm would look very similar
- It has neither a clock nor a reset.
- If this code is given to RTL synthesis tool, the resulting hardware would have 12 16-bit combinational multipliers 11 16-bit adders.
- The same operation can be done by using one adder and one multiplier.

Principles of Behavioral Design

- For sake of simplicity, consider a first-order filter.

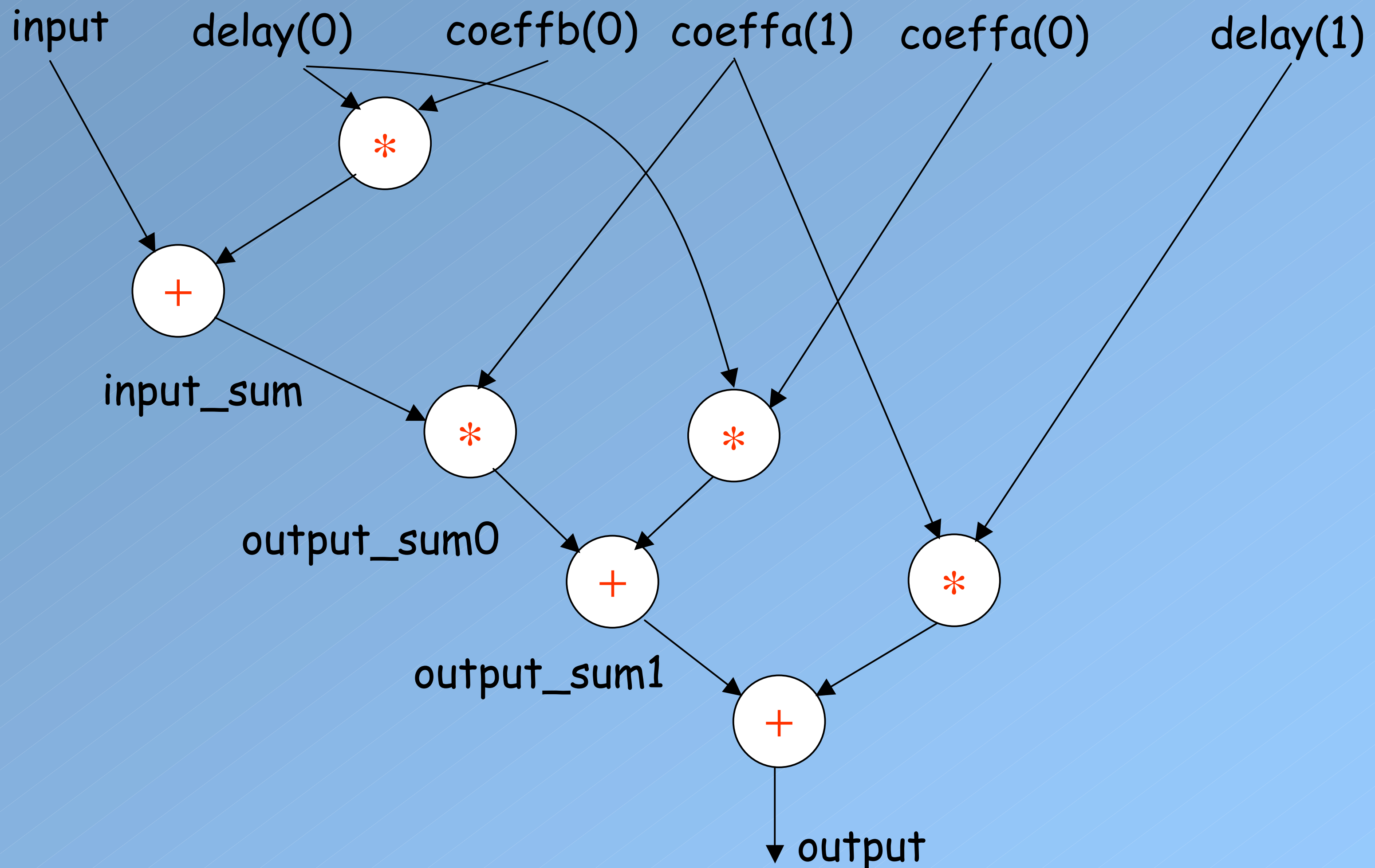
```
input_sum := input + delay(0)*coeffb(0);
output_sum := input_sum*coeffa(1);
output_sum := output_sum + delay(0)*coeffa(0);
output := output_sum + delay(1)*coeffa(1);
```

- To figure out the data dependency, rewrite the code fragment as follows

```
input_sum := input + delay(0)*coeffb(0);
output_sum0 := input_sum*coeffa(1);
output_sum1 := output_sum0 + delay(0)*coeffa(0);
output := output_sum1 + delay(1)*coeffa(1);
```

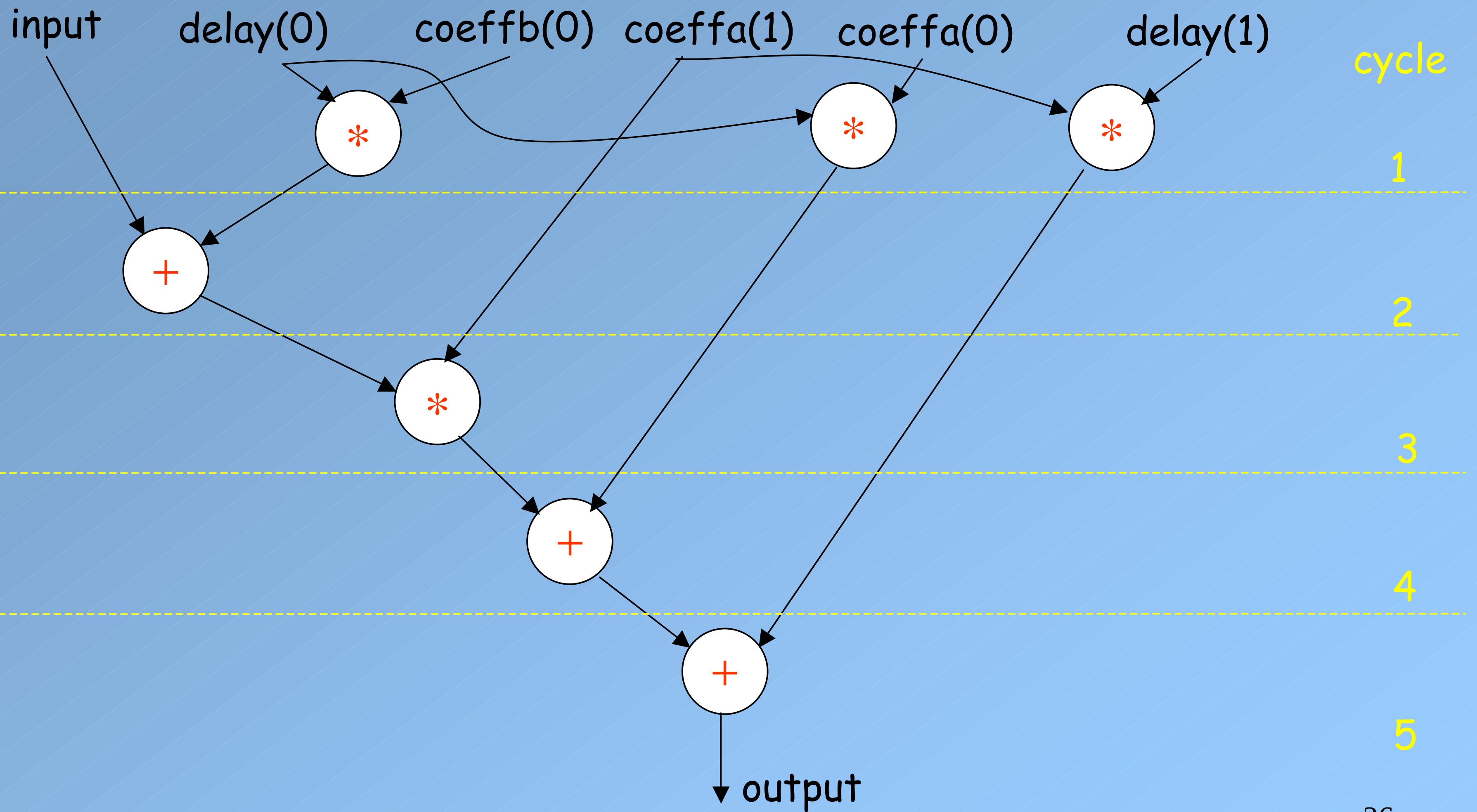
Principles of Behavioral Design

- Data dependency graph of first-order IIR filter



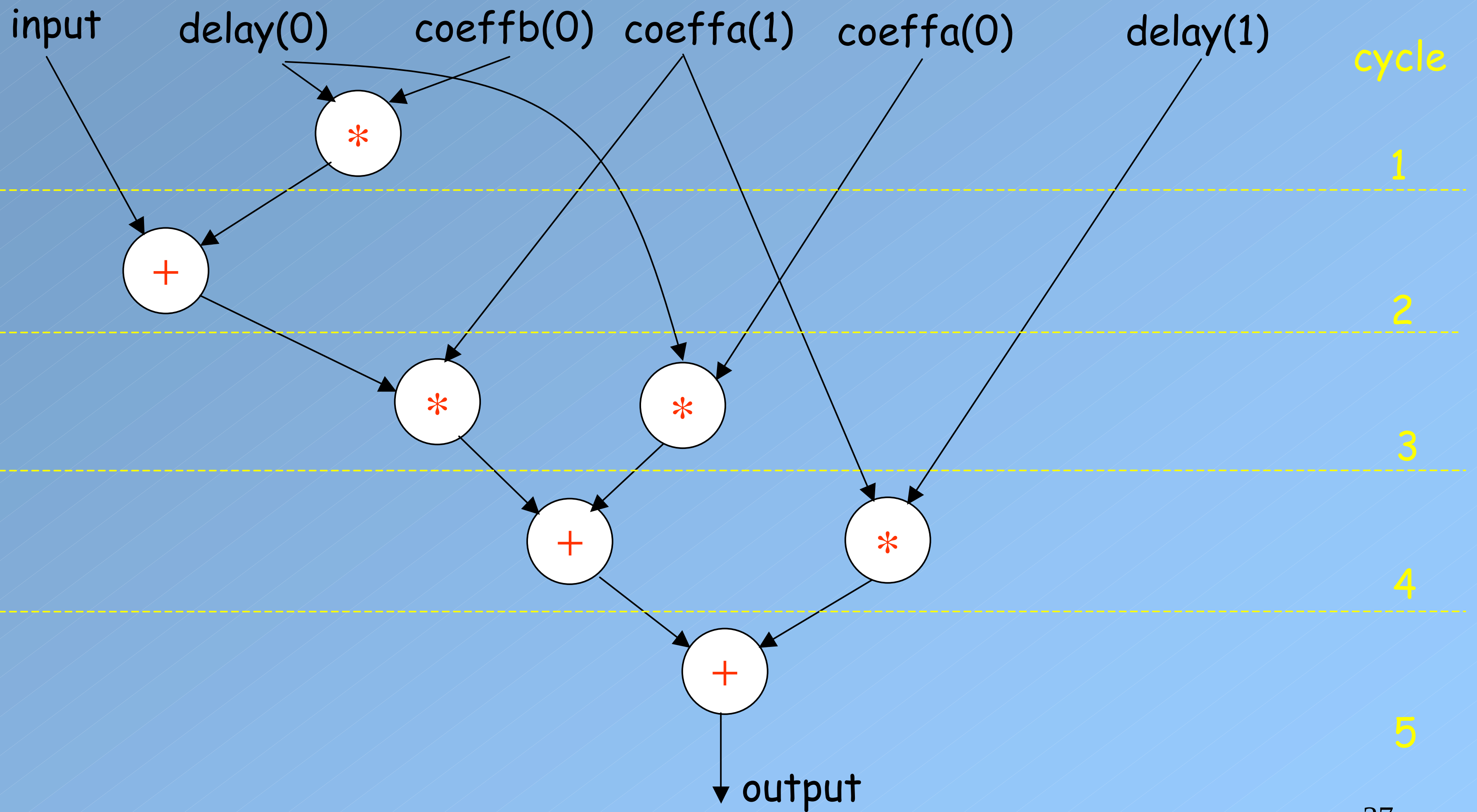
Principles of Behavioral Design

- ASAP schedule



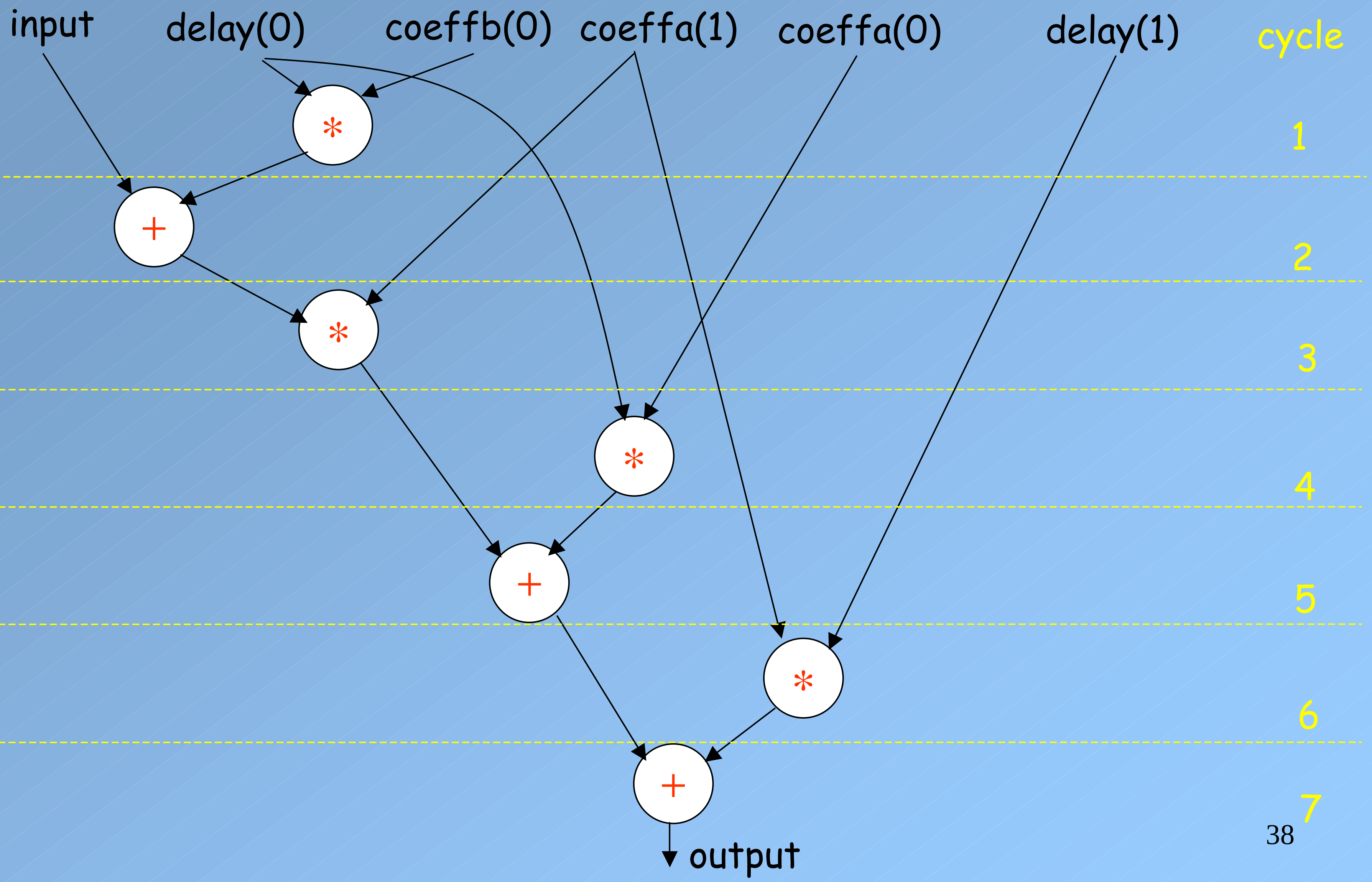
Principles of Behavioral Design

- Unconstrained ALAP Schedule

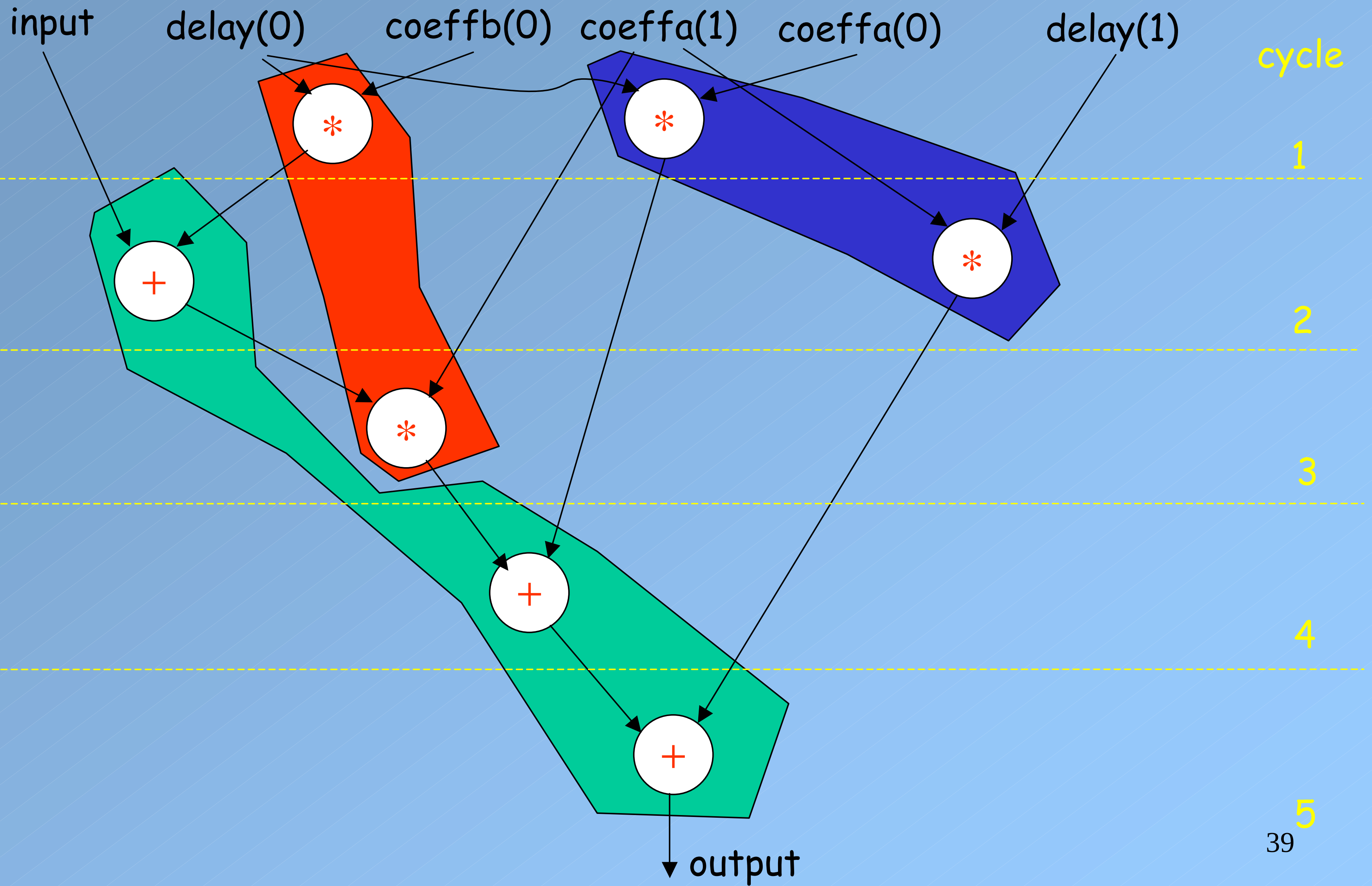


Principles of Behavioral Design

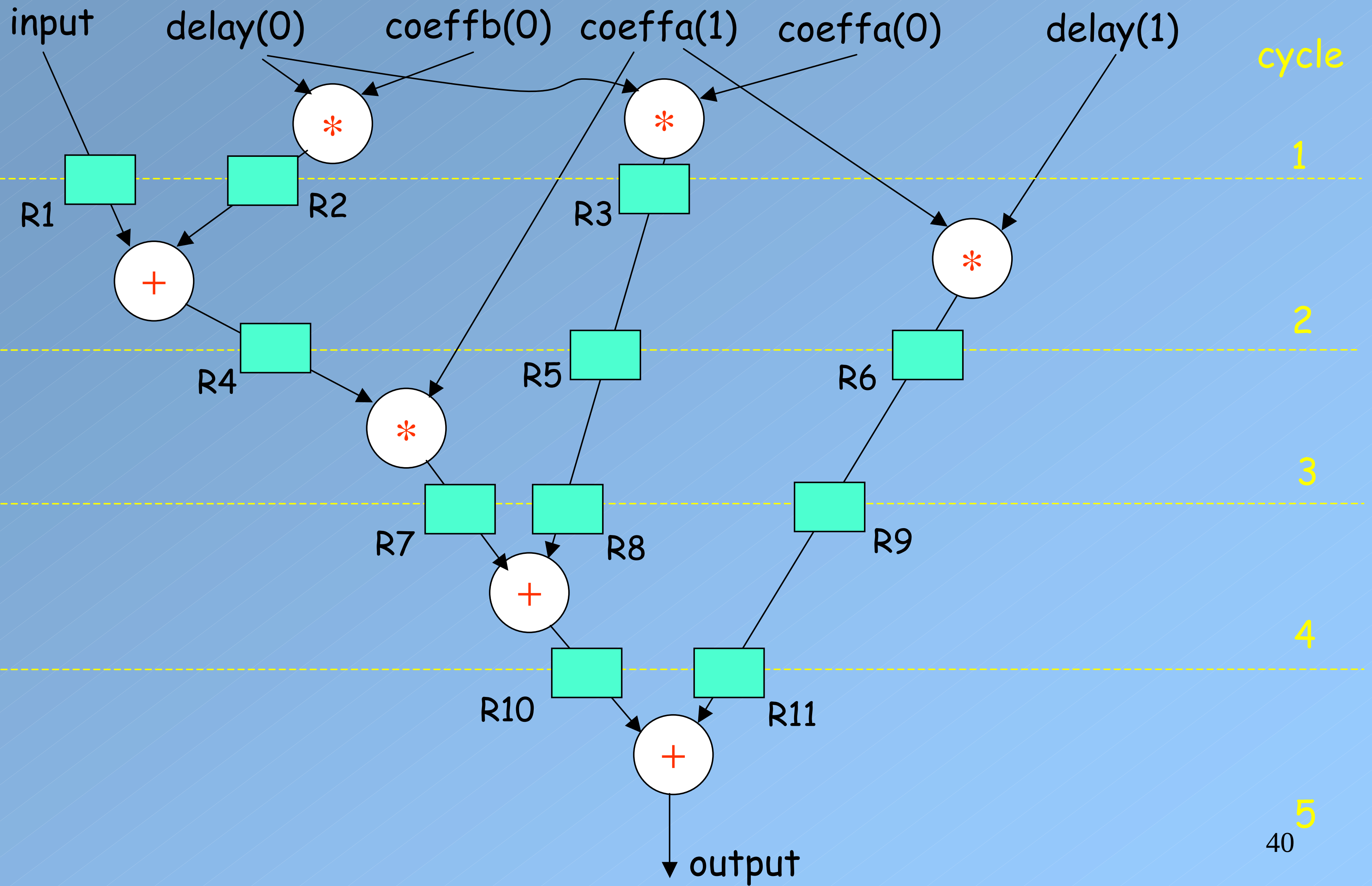
- Resource constrained schedule



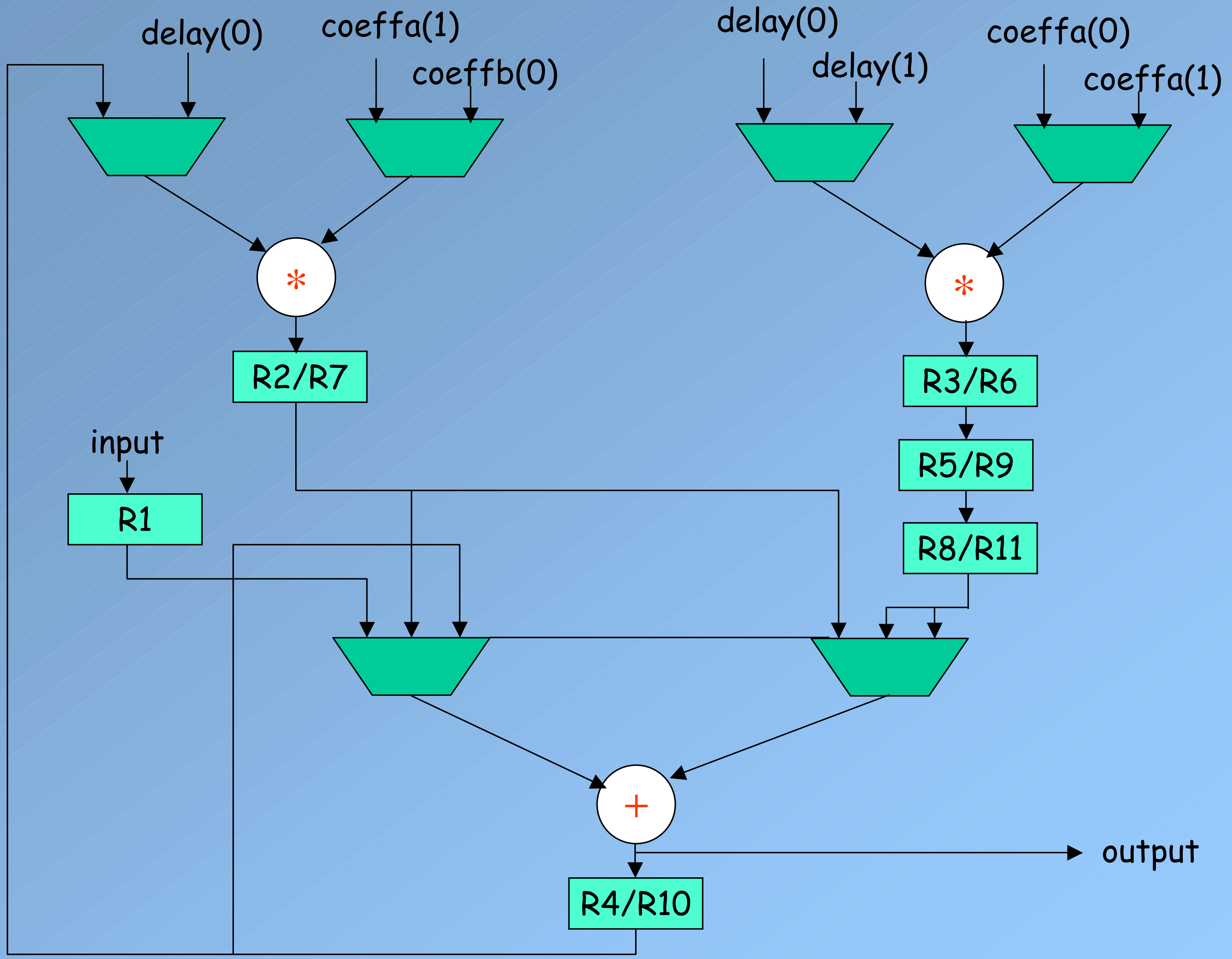
Mapping of Operations onto Resources



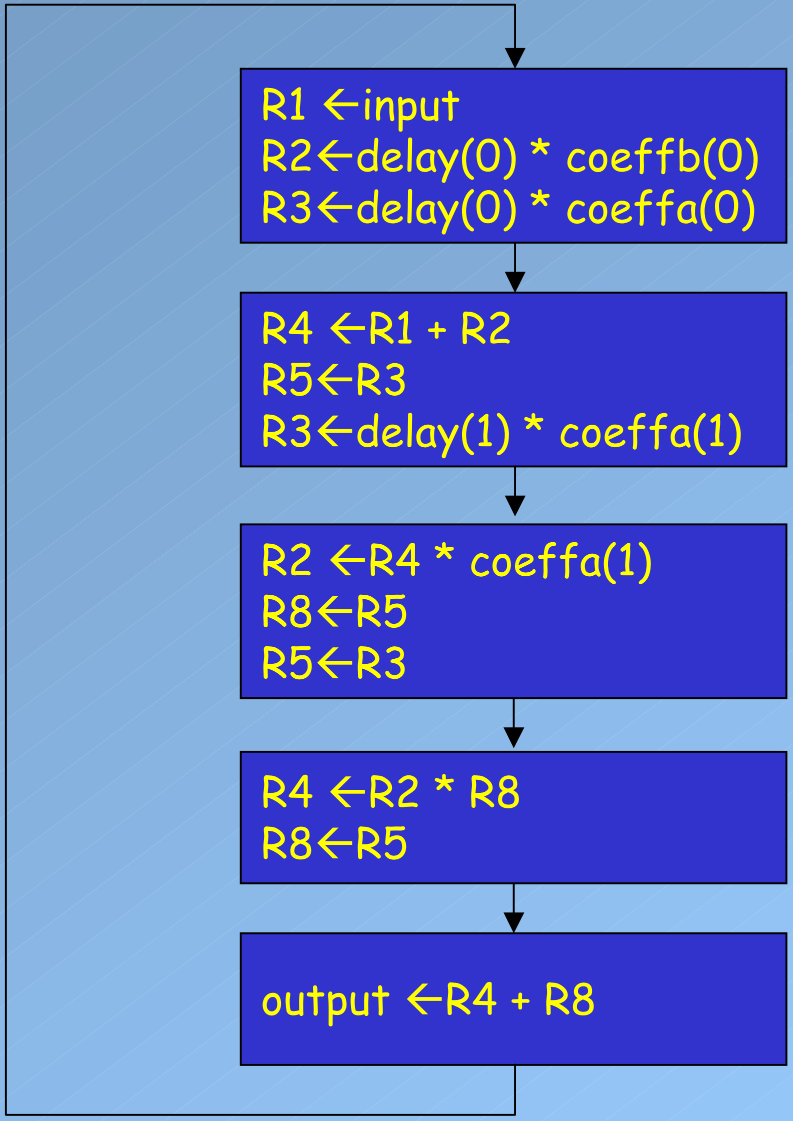
Schedule Showing Registers



Hardware Implementation of First-Order Filter



ASM Chart of Controller



VHDL Standards

- IEEE Std 1076-1993
  - Formal definition of VHDL
- IEEE Std 1076.1-1999 (VHDL-AMS)
  - Analog and mixed extensions to IEEE Standard VHDL
- IEEE Std 1076.2-1996
  - IEEE standard VHDL mathematical packages
  - real and complex functions for VHDL
- IEEE Std 1076.3-1997
  - IEEE standard VHDL synthesis packages
  - define `signed` and `unsigned` types and arithmetic functions, for use with synthesis tools

VHDL Standards

- IEEE Std 1076.4-1995
  - IEEE standard VITAL ASIC modeling
  - VITAL (VHDL Initiative Towards ASIC Libraries) is a set of low-level primitives for accurate timing simulations of gate-level models.
- IEEE Std 1076.6-1999
  - IEEE standard for VHDL register transfer level synthesis
  - defines a subset of VHDL appropriate to RTL synthesis.
- IEEE Std 1164-1993
  - IEEE standard multivalue logic system for VHDL model interoperability.

VHDL Standards

- IEEE Std 1029.1-1998
  - IEEE standard for waveform and vector exchanges (WAVES)
  - WAVES (Waveform and Vector Exchange to Support Design and Test Verification) is a set of VHDL methods to assist in verifying and testing hardware.
  - The motivations is to allow test vector files to be shared between VHDL simulators and hardware testers.

Verilog

- Seen as an alternative to VHDL
- developed in early 1980s by Gateway Design Automation, which was later taken over by Cadence.
- It is an IEEE standard (1364).
- Resembles the C programming language while VHDL is closer to Ada.
- Said to be simpler and closer to hardware than VHDL.
- It can be used to model logic circuits at the transistor or switch level.

Verilog

- Fault simulators have been developed to use Verilog, while such tools are almost non-existent for VHDL.
- On the other hand, VHDL has high-level constructs and abstract data types that Verilog does not have
  - This feature makes VHDL much more suitable to behavioral modeling.
- Many simulation and RTL synthesis tools accept both.
  - use VHDL for high-level design and Verilog for low-level post-synthesis timing and fault simulation

Basics of Verilog

- A Verilog Model of two-input NAND gate

```
// comment
module NAND (in1, in2, out);
  input in1, in2;
  output out;
  assign out = ~(in1 & in2);
endmodule // Note no semicolon
```

- Verilog is case-sensitive
- It does not have separate interface and implementation section – everything in a **module**.
- Signals do not have types.
- A signal can take the values: 0, 1, X, Z.
- **assign** introduces a assignment statement analogous to CSA.

Basics of Verilog

- The bitwise operators: ~, &, |, ^.
- To instantiate a module, it simply has to be invoked.

```
module simple (a, b, c, d);
  input a, b, c;
  output d;
  wire p, q; // can be omitted
  NAND g1(a, b, p);
  NAND g2(a, c, q);
  NAND g3(p, q, d);
endmodule
```

Procedures with Verilog

```
module MUX (SEL, A, B, C, D, Y);
  input [1:0] SEL;  // two bit vector
  input A, B, C, D;
  output Y;
  reg Y;            // needed for procedural assignment
  always @(SEL or A or B or C or D)
    case (SEL)
      2'b00 : Y = A;
      2'b01 : Y = B;
      2'b10 : Y = C;
      2'b11 : Y = D;
      default : Y = A;
    endcase
endmodule
```

Basics of Verilog

```
module LATCH (enable, D, Q);
  input enable, D;
  output Q;
  reg Y;                // needed for procedural assignment
  always @(enable or D)
    if (enable)
      Q = D;
endmodule
```

- Level-sensitive latch can be modeled using an incomplete if statement (as in VHDL)

Non-Blocking Assignment Statements

```
module LATCH (enable, D, Q);
  input enable, D;
  output Q;
  reg Y;
  reg M;
  always @(enable or D)
    if (enable)
    begin
      M <= D
      Q <= M;
    end
endmodule
```

- Like VHDL signal assignments, all non-blocking assignments are completed at the end of the current time period.
- No concept of delta delays
- This example will synthesize to two latches.

Blocking Assignment Statements

```
module LATCH (enable, D, Q);
  input enable, D;
  output Q;
  reg Y;
  reg M;
  always @(enable or D)
    if (enable)
    begin
      M = D;
      Q = M;
    end
endmodule
```

- A blocking statement must be completed before control passes to the next statement.
- This is similar to variable assignment in VHDL.
- One latch will be inferred.

Edge-Triggered Flip-Flop

```
module DFF (clock, D, Q);
  input clock, D;
  output Q;
  reg Q;
  always @(posedge clock)
    Q = D;
endmodule
```

A negative edge would be detected using **negedge**

Flip-Flop with Asynchronous Reset

```
module DFF (clock, reset, D, Q);
  input clock, reset, D;
  output Q;
  reg Q;
  always @(posedge clock or negedge reset)
    if (!reset)
      Q = 0;
    else
      Q = D;
endmodule
```

- Verilog does not have enumerated types
- State machines require that the state assignment to be explicitly stated.

State Machines with Verilog

```
module vending (clock, reset, twenty, ten, ready, dispense,
                ret, coin);
  input clock, reset, twenty, ten;
  output ready, dispense, ret, coin;
  reg ready, dispense, ret, coin;
  parameter A = 0, B = 1, C = 2, D = 3, F = 4, I = 5;
  reg [0:2] present_state, next_state;
  always @(posedge clock or posedge reset)
    if (reset)
      present_state = A;
    else
      present_state = next_state;
...
endmodule
```

State Machines with Verilog

```
module vending (clock, reset, twenty, ten, ready, dispense,
                ret, coin);
...
  always @(twenty or ten or present_state)
  begin
    ready = 0;
    dispense = 0;
    ret = 0;
    coin = 0;
    case (present_state)
      A: begin
         ready = 1;
         if (twenty) next_state = D;
         else if (ten) next_state = C;
         else next_state = A;
         end
...
endmodule
```

State Machines with Verilog

```
    case (present_state)
      ...
      B: begin
         dispense = 1; next_state = A;
         end
      C: begin
         coin = 1;
         if (twenty) next_state = F;
         else if (ten) next_state = D;
         else next_state = C;
         end
      ...
endmodule
```

State Machines with Verilog

```
     case (present_state)
      ...
      D: begin
         coin = 1;
         if (twenty) next_state = B;
         else if (ten) next_state = F;
         else next_state = D;
         end
      F: begin
         coin = 1;
         if (twenty) next_state = I;
         else if (ten) next_state = B;
         else next_state = F;
         end
...
endmodule
```

State Machines with Verilog

```
...
    case (present_state)
      ...
      I: begin
         ret = 1;
         next_state = A;
      end
      default next_state = A;
    endcase
  end
endmodule
```

Testbench with Verilog

```
‘timescale 1ns / 100 ps
module testbench;
  reg clock, reset, twenty, ten;
  wire ready, dispense, ret, coin;
  vending vm (clock, reset, twenty, ten, ready, dispense, ret, coin);
  initial clock = 0;
  always #10 clock = !clock;
  initial
  begin
    reset = 1; twenty, = 0; ten = 0;
    #1  reset = 0;
    #64 twenty = 1;
    #80 twenty = 0;
    #20 ten = 1;
    #20 ten = 0;
    #20 twenty = 1;
    #20 twenty = 0;
  end
endmodule
```

Switch-Level Modeling with Verilog HDL

- Two types of MOS switches are specified in Verilog HDL
  - **nmos** (drain, source, gate);
  - **pmos** (drain, source, gate);
- Switch level implementation of an inverter
  - module inverter(Y, A);

```
module inverter(Y, A);
   input A;
   output Y;
   supply1 PWR;
   supply0 GRD;
   pmos(Y, PWR, A);
   nmos(Y, GRD, A);
endmodule
```

Switch-Level Modeling with Verilog HDL

- Switch level implementation of 2-input NAND gate

  -
    ```
    module NAND2(Y, A, B);
      input A, B;
      output Y;
      supply1 PWR;
      supply0 GRD;
      wire W1;
      pmos(Y, PWR, A);
      pmos(Y, PWR, B);
      nmos(Y, W1, A);
      nmos(W1, GRD, B);
    endmodule
    ```