TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Computer Engineering, Chair of VLSI Design, Diagnostics & Architecture

VHDL – SEQUENTIAL LOGIC DESCRIPTION

Dr. Thomas B. Preußer

thomas.preusser@tu-dresden.de

Dresden, July 4, 2014

Goals of this Lecture

- Show how sequential logic is modeled.
- Give examples for standard structures:
  - Plain registers,
  - Counters, and
  - State machines.

Covered Scope

- Sequential logic has *state*.
- We address *synchronous* sequential logic where state changes occur upon active clock edges.
- An *asynchronous* reset or set signal may be present (but its use is discouraged).

Basic Approach

Use a **process**, which is triggered by an event on:

- the clock input, or
- one of the asynchronous controls.

```
process(clk, rst) -- !!!: ONLY inputs that may
begin             --      induce a state transition
  if rst = '1' then                    -- asynchronous reset
    Q <= '0';
  elsif clk'event and clk = '1' then -- positive clock edge
    Q <= D;
  end if;
end process;
```

Fully Synchronous Design

```
process(clk)          -- !!!: ONLY inputs that may
begin                 --       induce a state transition
  if clk'event and clk = '1' then -- positive clock edge
    if rst = '1' then             -- synchronous reset
      Q <= '0';
    else
      Q <= D;
    end if;
  end if;
end process;
```

The formula `clk'event and clk = '1'` is often abbreviated by the more sophisticated:

```
function rising_edge(signal s : std_ulogic) return boolean is
begin
  return (s'event and (to_X01(s) = '1') AND (to_X01(s'last_value) = '0'));
end;
```

which can cope with any transition from `'0'|'L' => '1'|'H'`.

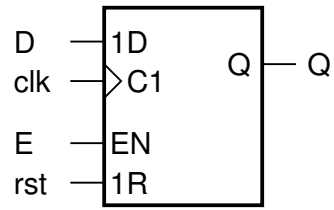
Synthesis tools accept both.

Conditional State Transitions

State transitions may be guarded, as by enable logic, or derived from more complex computations.

Example: D-FF with an enable and a synchronous reset:

```
process(clk) -- !!!: ONLY inputs that may
begin        --      induce a state transition
  if rising_edge(clk) then          -- positive clock edge
    if rst = '1' then               -- synchronous reset
      Q <= '0';
    elsif E = '1' then              -- enable
      Q <= D;
    end if;
  end if;
end process;
```



Beware: you *cannot* model combinatorics deriving from the state within the same process.

Permissible Clock

$$f = \frac{1}{T}$$

Proper operation requires:

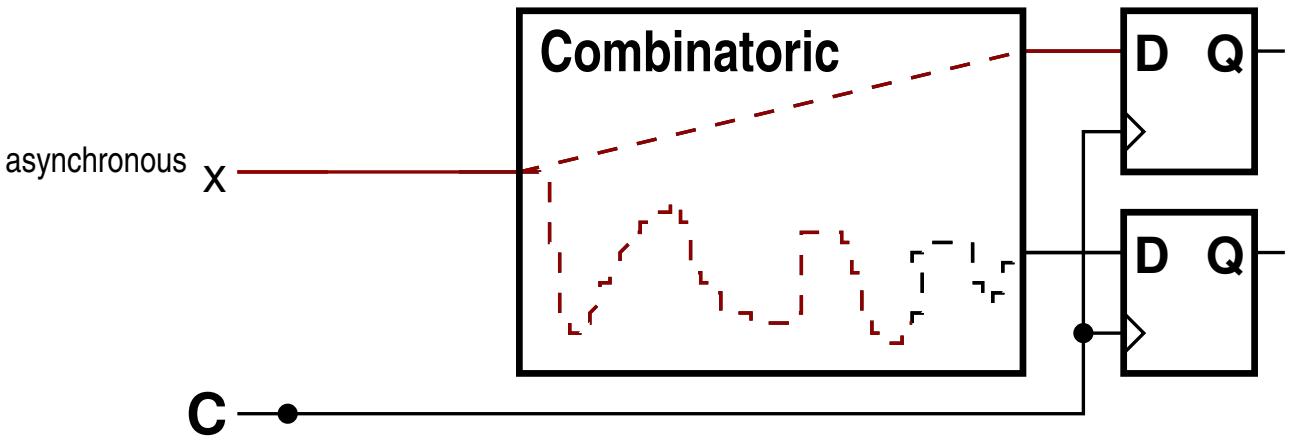
$$\forall t_{\text{comp}} . t_{\text{clock2output}} + t_{\text{comp}} + t_{\text{setup}} \leq T$$

Synthesis ensures this property or reports an error if it is unable to achieve the specified timing.

This constraint is checked for all FF-FF paths (within the same clock domain). The longest such paths are called the *critical* paths.

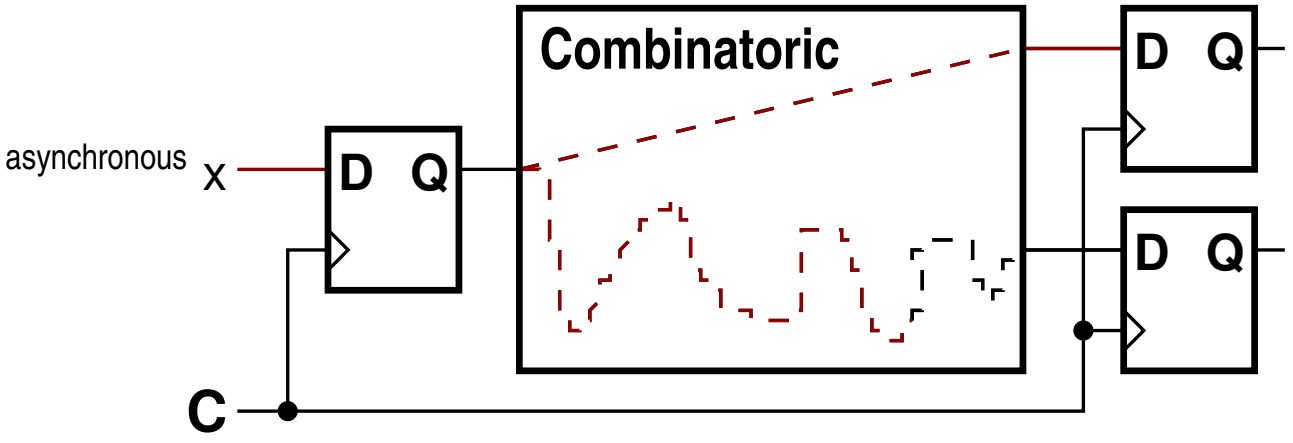
What about asynchronous inputs?

Asynchronous Inputs



- Input switches at an *arbitrary* time (with respect to the device clock).
- Propagation through the combinatorics by the next active edge is *not* assured.
- Inconsistencies in the system state may arise from different path lengths.

Asynchronous Inputs



- Register-register delays are determined by the synthesis tool.
  → A warning is generated if the clock constraint cannot be satisfied.
- Often, there is a second FF-stage for masking meta-stabilities.
- Note that also Reset buttons are asynchronous inputs.

Standard Pattern: Shift Register

For data serialization (here: MSB first):

```
signal Buf          : std_logic_vector(7 downto 0);
signal Load, Shift  : std_logic;
...
process(clk)
begin
  if rising_edge(clk) then
    if Load = '1' then
      Buf <= Dat;
    elsif Shift = '1' then
      Buf <= Buf(Buf'left-1 downto 0) & '-';
    end if;
  end if;
end process;
```

Standard Pattern: Shift Register

As ring counter:

```
signal Buf   : std_logic_vector(7 downto 0);
signal Shift : std_logic;
...
process(clk)
begin
  if rising_edge(clk) then
    if rst = '1' then
      Buf    <= (others => '0'); -- default assignment
      Buf(0) <= '1';             -- override wins
    elsif Shift = '1' then
      Buf <= Buf(Buf'left-1 downto 0) & Buf(Buf'left);
    end if;
  end if;
end process;
```

Array Construction

- by aggregate composition: ('0', '0', '0', '1')
  - with positional association: (0 => '0', 1 => '0', 2 => '0', 3 => '1')
  - using ranges: (0 **to** 2 => '0', 3 => '1')
  - simplified (using character literals only): "0001", x"1", 4x"1"
- as slice: a(5 **downto** 2)
- from concatenation: a(7 **downto** 6) & a(3 **downto** 2) & '–'

For assignments:

- only length matters,
- left goes left and right goes right,
- exact index range and direction are irrelevant,
- **others** range may be used on right hand side.

Standard Pattern: Counter

For event counting:

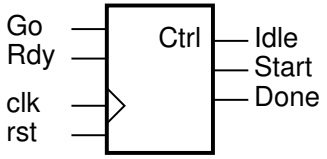
```
signal Cnt   : unsigned(17 downto 0) := (others => '0');
signal Event : std_logic;
...
process(clk)
begin
  if rising_edge(clk) then
    if rst = '1' then
      Cnt <= (others => '0');
    elsif Event = '1' then
      Cnt <= Cnt + 1;
    end if;
  end if;
end process;
```

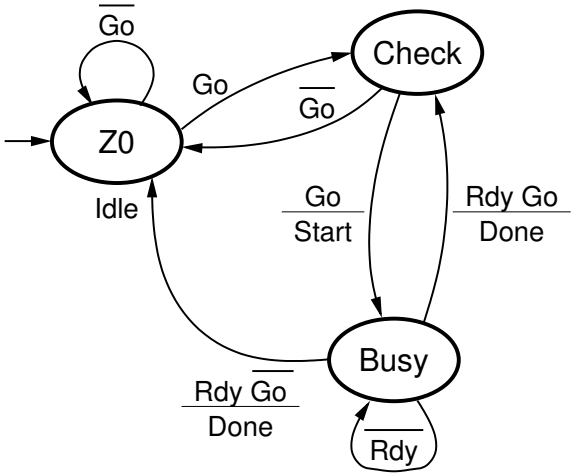
Standard Pattern: Counter

As delay counter:

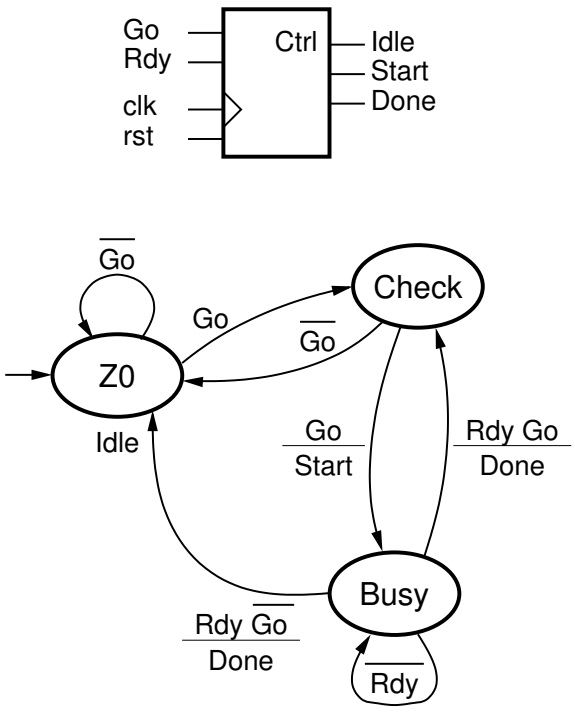
```
signal Cnt : signed(log2ceil(CNT_DELAY-1) downto 0)
                := ('0', others => '-');
signal Init, Done : std_logic;
...
process(clk)
begin
  if rising_edge(clk) then
    if Init = '1' then
      Cnt <= to_signed(CNT_DELAY-2, Cnt'length);
    elsif Shift = '1' then
      Cnt <= Cnt - 1; -- Advantage of counting down?
    end if;
  end if;
end process;
Done <= Cnt(Cnt'left);
```

Standard Pattern: Finite State Machine





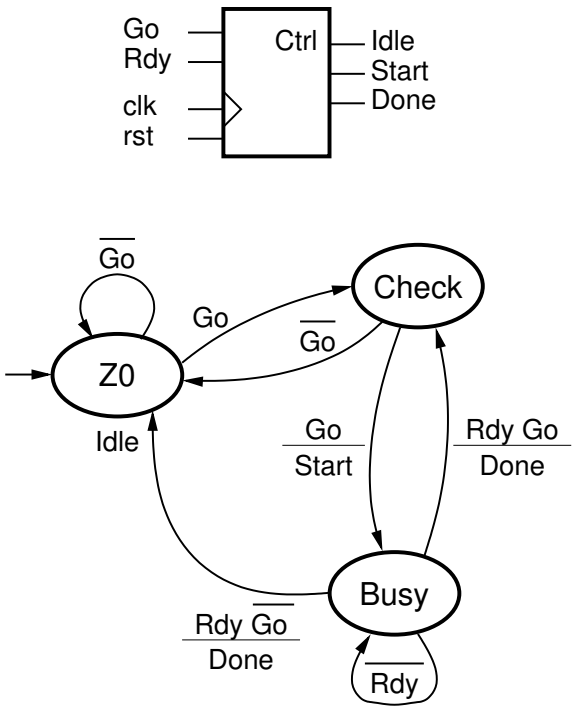
Interface Definition

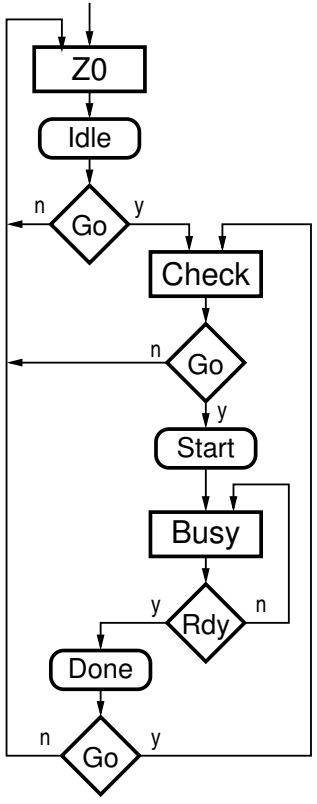



```
entity ctrl is
 port(
   rst, clk : in  bit; -- Reset, Clock
   Go       : in  bit; -- Start cycle
   Rdy      : in  bit; -- Stop cycle

   Idle     : out bit; -- Idle
   Start    : out bit; -- Start of cycle
   Done     : out bit  -- End of cycle
 );
end ctrl;
```

SM-Chart





- equivalent to a state diagram,
- precise modeling of hierarchical decisions, and
- easy to translate into an HDL.

Types and Signals

```
architecture rtl of ctrl is
  type tState is (Z0, Check, Busy);
  signal State     : tState := Z0; -- state register
  signal NextState : tState;       -- combinatorial computation
begin                              --   of successor state
  ...
end rtl;
```

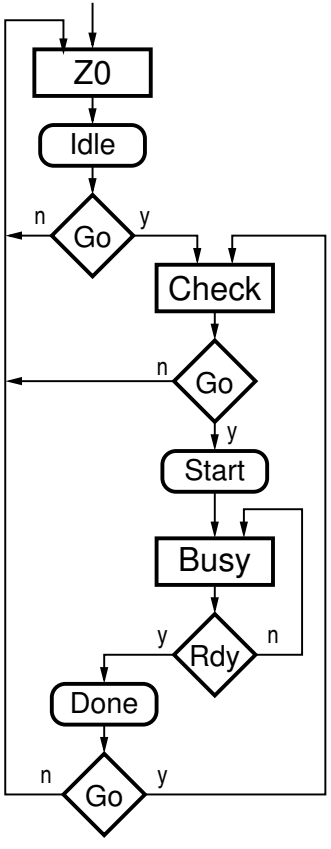
Common Problem:

Initialization of register signals should be equivalent to their reset assignment! Otherwise inconsistencies between the system states after an initialization (e.g. FPGA-programming) and after a reset are possible. → *Happy Debugging!*
If there is no initialization, synthesis tools try to deduce the initialization state from the reset description – simulators do not!

Registers

```
architecture rtl of strg is
  type tState is (Z0, Check, Busy);
  signal State     : tState := Z0; -- state register
  signal NextState : tState;       -- combinatorial computation
begin                              --   of successor state
  -- Sequential Process
  process(clk)
  begin
    if clk'event and clk = '1' then -- rising edge
      if rst = '1' then             -- synchronous reset
        State <= Z0;
      else
        State <= NextState;
      end if;
    end if;
  end process;
  ...
end rtl;
```

Combinatorics




```
-- Combinatorial Process
process(State, Go, Rdy)
begin
  -- Default Assignments
  NextState <= State;
  Idle      <= '0';
  Start     <= '0';
  Done      <= '0';

  case State is
    when Z0 =>
      Idle <= '1';
      if Go = '1' then
        NextState <= Check;
      end if;

    when Check =>
      if Go = '0' then
        NextState <= Z0;
      else
        Start     <= '1';
        NextState <= Busy;
      end if;

    when Busy =>
      if Rdy = '1' then
        Done <= '1';
        if Go = '0' then
          NextState <= Z0;
        else
          NextState <= Check;
        end if;
      end if;

    -- not required here
    -- completes state enumeration
    when others =>
      null;       -- does nothing
  end case;
end process;
```