UiO : Department of Informatics
University of Oslo

IN3160 IN4160
Diagrams,   Reset circuits,   ASMD/ desing example

Yngve Hafting

Course Goals and Learning Outcome

https://www.uio.no/studier/emner/matnat/ifi/IN3160/index-eng.html

In this course you will learn about the **design of advanced digital systems**. This includes programmable logic circuits, a hardware design language and system-on-chip design (processor, memory and logic on a chip). Lab assignments provide practical experience in how real design can be made.

After completion of the course you will:

- understand important **principles for design** and testing of digital systems
- understand the relationship between behaviour and different construction criteria
- **be able to describe advanced digital systems at different levels of detail**
- be able to perform simulation and synthesis of digital systems.

Goals for this lesson:

- Be able to read, use and create -diagrams
    - Timing-/ waveform-
    - Datapath-
    - Block-
    - State-
    - ASM-,
    - ASMD
- Know the purpose of reset circuits
    - Why do we reset at all?
    - What are potential pitfalls for reset handling?

- Practice in reading code
    - Design considerations

Next lesson: Microcoded FSMs

Wave diagrams

- When do we read clocked signals?
- When does assignment occur?

- Does phase matter?

Basic wave diagram layout

- Undefined values are usually hatch-patterned

- Single bit signals
  - Are usually displayed as high ('1') or low ('0')
  - Defined signals of arbitrary values are usually shown as an area without hatching
    - Can be named (here a, b)

- Multi-bit is usually shown as an area
  - Values or names are often used
  - Multi-bit vector vs single-bit signal
    - Sometimes you must use context to understand which is which.

- Edges
  - Both vertical and slanted edges mean the same.
    - *Hand drawing mostly vertical…*
  - Normally sequential logic relate to the rising_edge
  - RTL-simulation of sequential logic:
    - Signals are read when edge occurs
    - Signals are assigned immediately after the edge
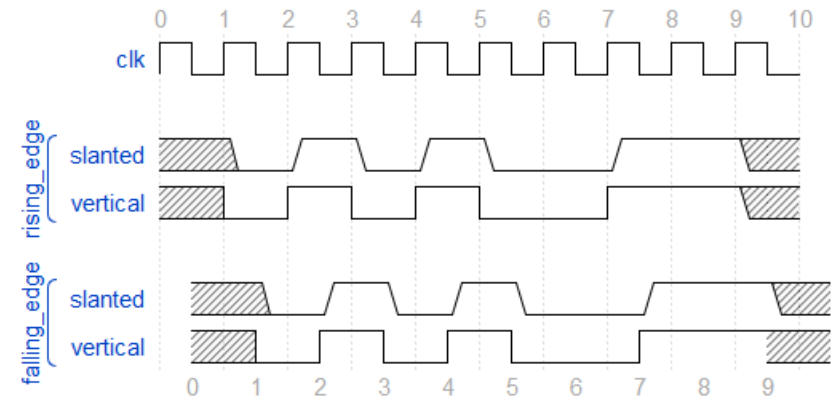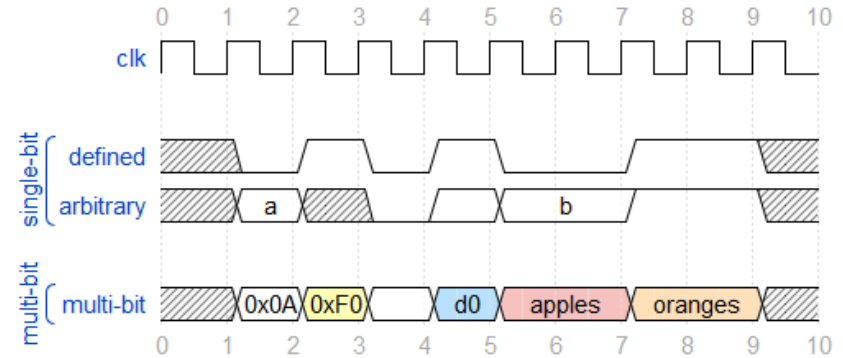
Basic wave diagram layout

- Combinational logic (CL) responds immediately when input changes
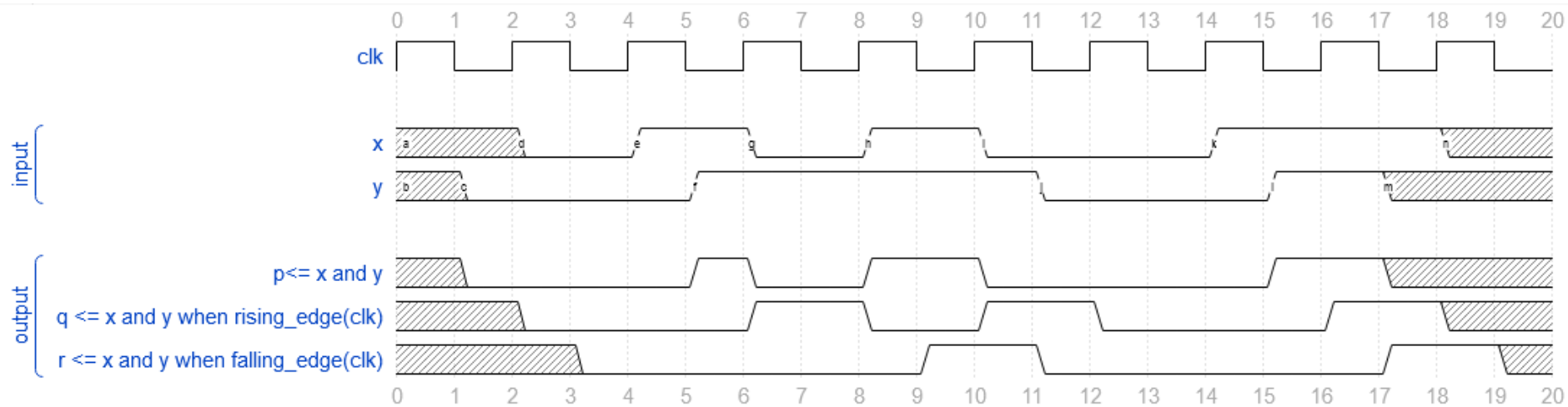
- Sequential logic (clocked circuits)
  - responds to the **input status** present *when the clock edge occurs*.
  - **output** is changed *immediately after* the clock edge occurs.
    - *Unless specific timing information is given*

6

Waveform combinational

- Combinational response is immediate in a waveform.
  - *Timing diagrams* will show gate delays
  - Clock *does not matter*
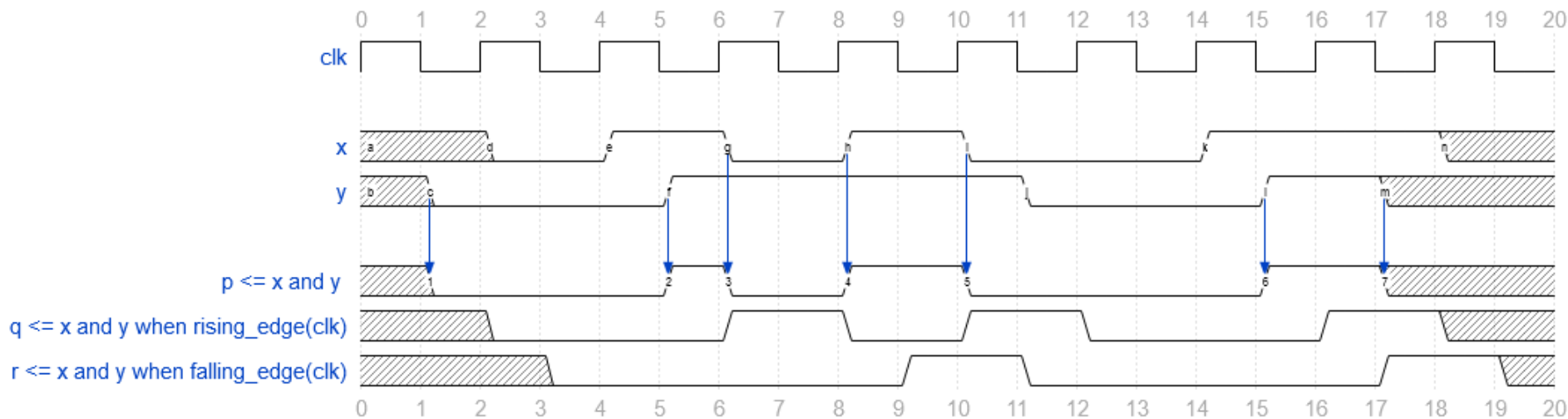
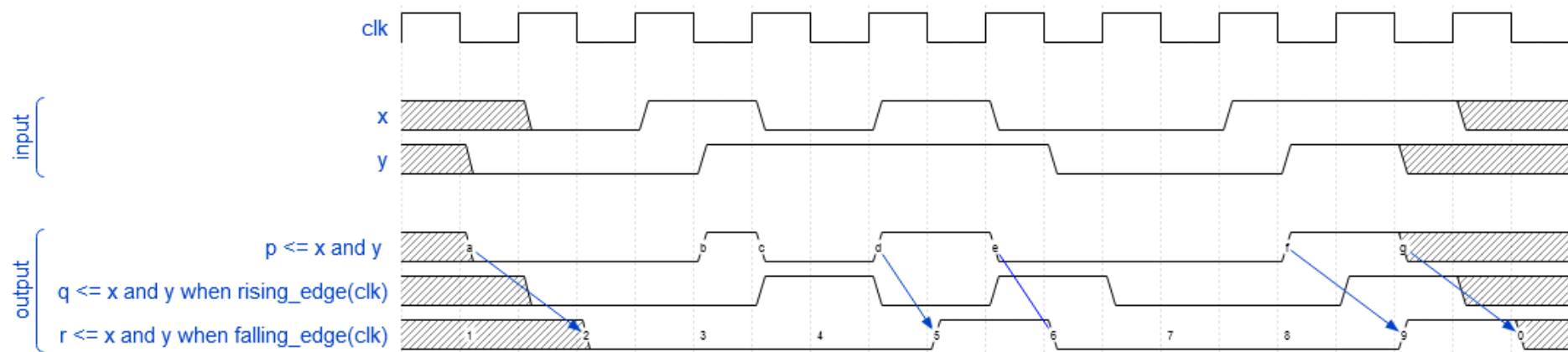# **Sequential logic, rising_edge**

<div style="border: 3px solid red; background: yellow;">
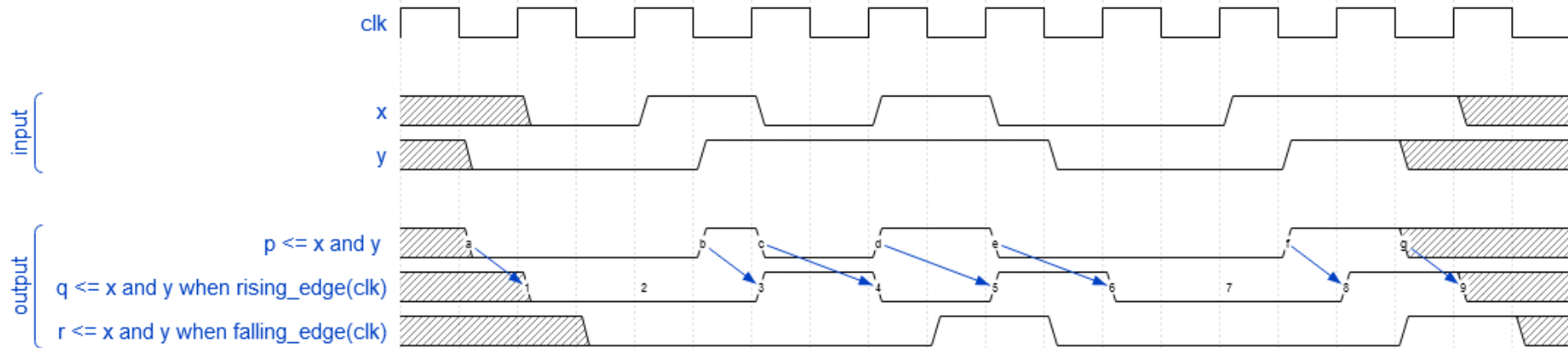
– Look at what the input status **was** when the clock edge occurs.

</div>

- Here: The b-c pulse activates q for one whole clock cycle
  - even though (a and b) *is high for a shorter duration*
    - *Only what is present when the clock edge occurs matters*

Sequential logic, falling_edge

- Here: The b-c pulse is too short to activate r
 - (a and b) *is low on the next falling edge*

Timing diagrams

- = Waveforms with timing information for a specified circuit.
  - Typically found in a data sheets
    - Printed circuit boards (PCBs),
    - components (chips)
    - communication protocol standards
  - Can be generated when doing post-synthesis simulation.
  - Useful…
    - .. when considering *setup and hold timing requirements* in a system
      - selecting circuits for a PCB (Printed Circuit Board)
    - *.. when optimizing data-flow circuitry..*
      - High performance ASIC design
- *So far today: no timing information given (only RTL sim)*

Block diagrams

- Shows how modules are connected *or* communicate
  - Level of detail may vary
  - May include
    - digital and analog components
- Suited for system level overview
  - Or partial overview
 - *May* be used for connection
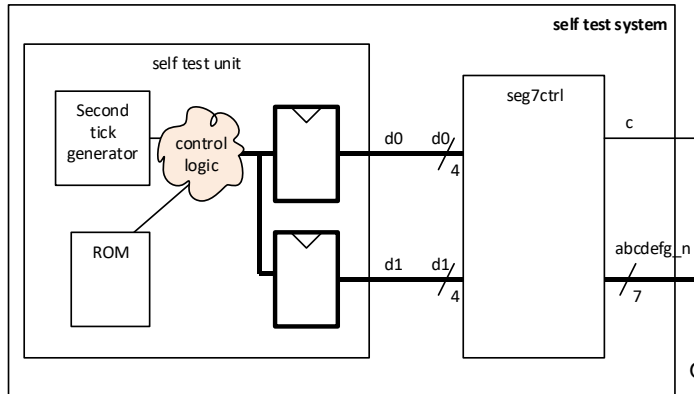
Oblig 6 diagram
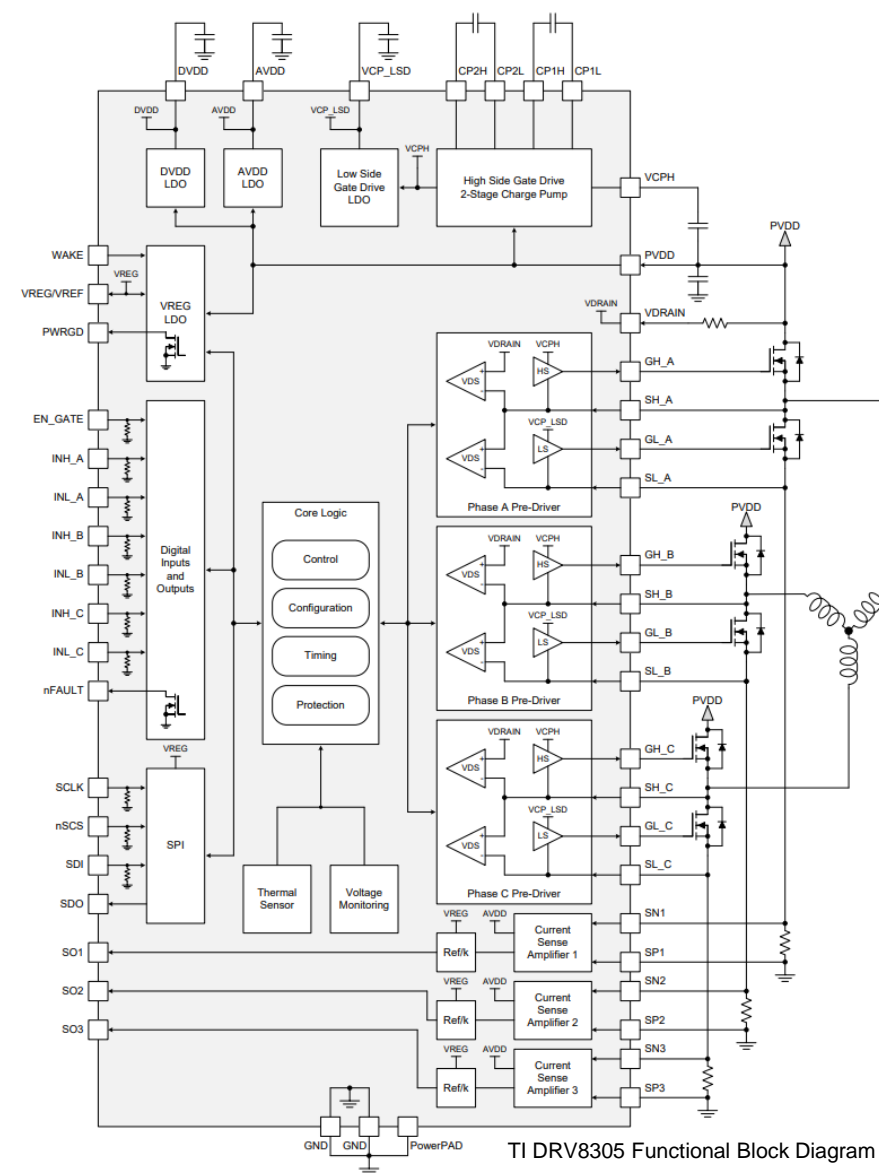
TI DRV8305 Functional Block Diagram

Block diagrams

- Many variants…
  - Entire- or parts of systems

  - *HW block diagrams tend to be more detailed than those used for other purposes (SW, business, …)*

- *Usually the first diagrams drawn in a design process*
  - Several may be added and edited later
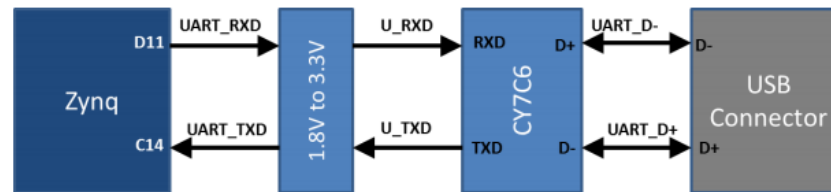
- Almost always present in design documentation.
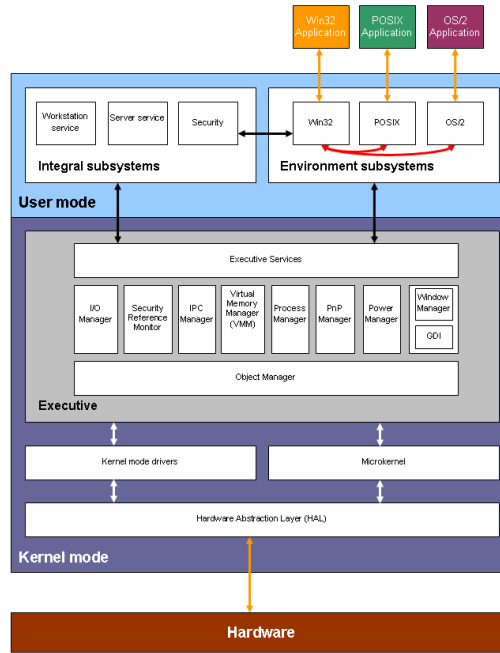
https://commons.wikimedia.org/wiki/File:Windows_2000_architecture.png
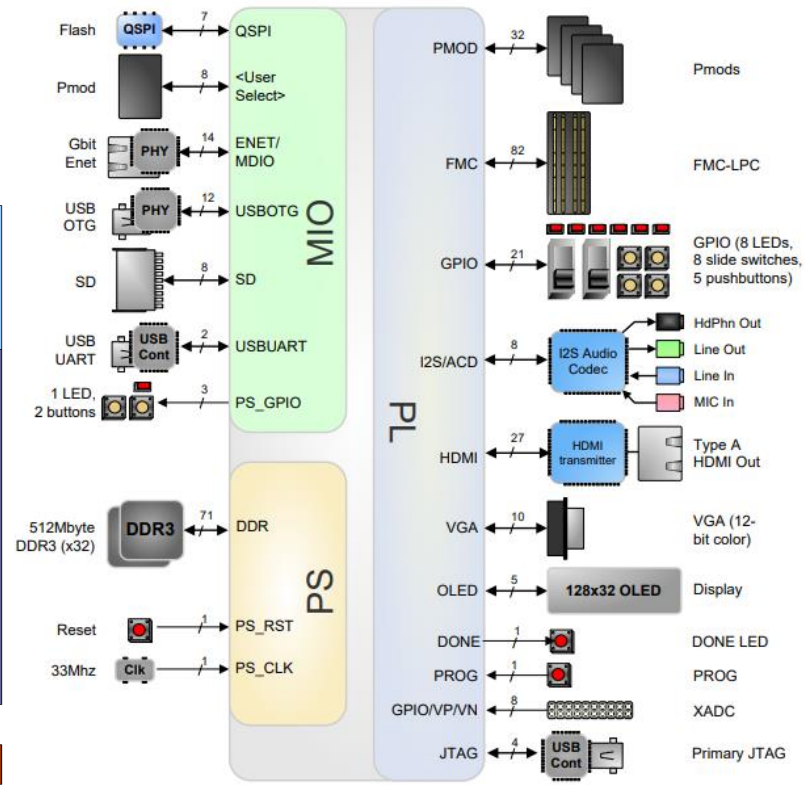
ZYNQ XC7Z020-CSG484

ZedBoard Block Diagram

← ↑ Zedboard Users guide

USB-UART Bridge Interface

12

Datapath Diagrams
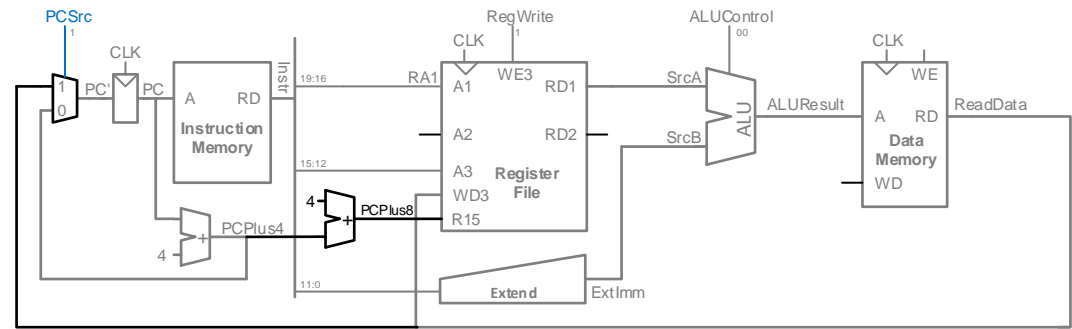
- A specialised block diagram that **show how data moves within a system or module**.

- Should normally contain
  - How data travels between modules or functional units
  - Registers (flipflops) used for storing data (not FSM - registers)
  - Bit widths for each path

- Usually direction of travel is from left to right

- Modules, such as **FSMs** will contain **registers,** however these registers **are considered control**, and *not a part of the datapath*.
  - Ie. A processor may have many states but data still moves through the pipeline one stage at a time.
  - (Thus we have non datapath modules using clk.)
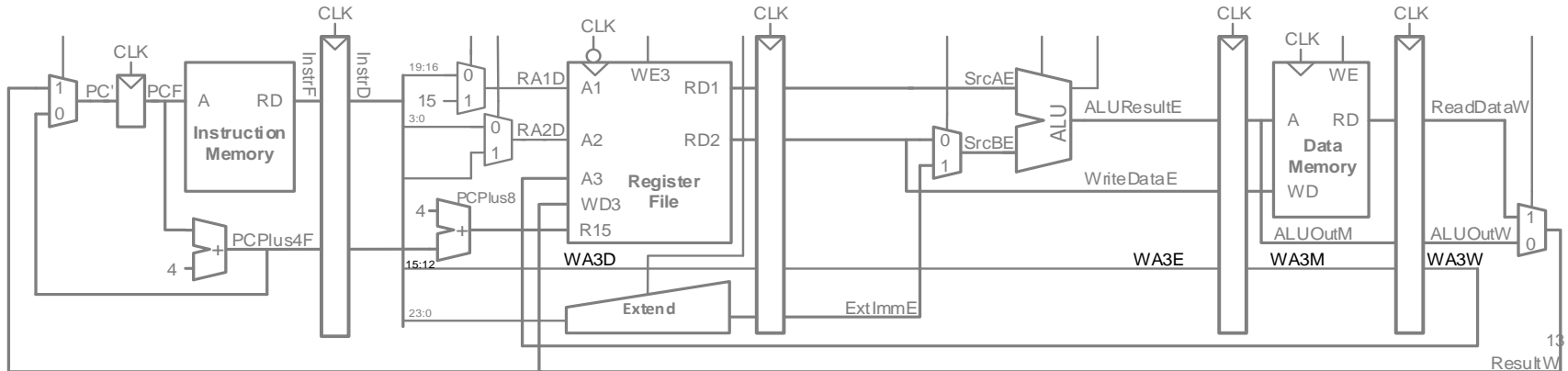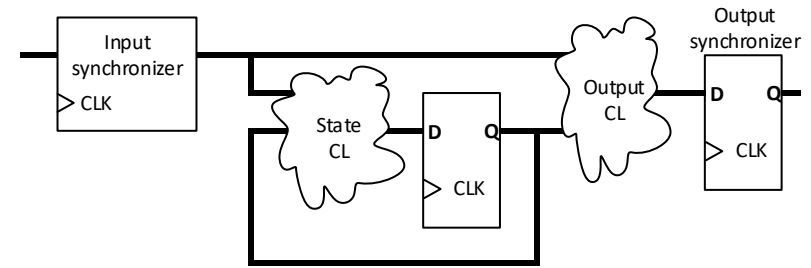
Datapath diagrams

- Often used to describe parts of an architecture
- RTL logic *can* be represented using datapath diagrams
  - It is not necessarily the best representation…
    - FSMs are better described using ASM diagrams…
    - Counters..
    - LFSR..

- Drawing a datapath diagram will in some cases be a very efficient way to gain understanding..
  - => Pipelining
    - **Example:** Exam 2021, Assignment 6 (next page)-

14

Example: Exam 2021
Assignment 6, «Pipelining»

- The pipelined module entity is described above.

- In this assignment a module which **calculates result = a+b+c** shall be implemented.

- In order to meet timing closure at the required frequency**, pipelining shall be used**.

- All input are synchronized to the clock signal clk.

- Reset can be either synchronous or asynchronous.

- The implementation shall be synchronous to the clock signal clk.

- **All output should be driven by registers** to avoid propagating hazards.

- The computation shall use unsigned arithmetic operation on the operands a, b and c.

- *When the start signal is high the computation shall start, and the result_valid signal shall be high when valid data is present on the result signal.*

- Implement the architecture for the pipelined module.

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

entity pipelined is
 port (
  clk : in std_logic;
  rst : in std_logic;
  a : in std_logic_vector(7 downto 0);
  b : in std_logic_vector(7 downto 0);
  c : in std_logic_vector(7 downto 0);
  result : out std_logic_vector(9 downto 0);
  start : in std_logic;
  result_valid : out std_logic
  );
end entity pipelined;
```

- We read this as the control signal shall be pipelined along with the data
  - It is not necessary to block computation when there is no start signal.

- NOTE: It does not make sense to pipeline unless a, b, c and the control signal follows each other.
  - Ie: expect new data each clock cycle

- Make your own datapath diagram for this task (2 min)!

Ex2021-pipelining

UiO : Department of Informatics
University of Oslo

- VHDL code may come in many flavours:

Datapath Diagram..
to be revealed during lecture

```vhdl
architecture RTL of pipelined is
begin
  pipeline: process(rst, clk) is
    variable r_start : std_logic;
    variable r_c : unsigned(7 downto 0);
    variable r_ab : unsigned(8 downto 0);
  begin
    if rst then
      r_start := '0';
      r_c := (others => '0');
      r_ab := (others => '0');
      result <= (others => '0');
      result_valid <= '0';
    elsif rising_edge(clk) then
      -- Note: Variables must be used _before_ being
      -- assigned to form registers...
      result <= std_logic_vector("00" & r_c + "0" & r_ab);
      result_valid <= r_start;
      r_start := start;
      r_ab := ("0" & unsigned(a) + "0" & unsigned(b));
      r_c := unsigned(c);
    end if;
  end process;
end architecture RTL;
```

```vhdl
architecture RAW of pipelined is
  signal r_c : unsigned(7 downto 0);
  signal r_ab : unsigned(8 downto 0);
  signal r_start : std_logic;
begin
  r_start        <= '0'  when rst else start    when rising_edge(clk);
  result_valid   <= '0'  when rst else r_start  when rising_edge(clk);
  r_ab <=(others => '0') when rst else ("0" & unsigned(a) + "0" & unsigned(b))     when rising_edge(clk);
  r_c <= (others => '0') when rst else (unsigned(c))                               when rising_edge(clk);
  result <= (others => '0') when rst else std_logic_vector("00" & r_c + "0" & r_ab) when rising_edge(clk);
end architecture RAW;
```

- Getting to this code should be doable when having the diagram…
- Common mistakes:
  – Forgetting to put c in pipeline before calculating step 2.
  – Believing start shall be valid for 3 clock cycles…
  – Attempting SW-style loops(!)..
    • Think of VHDL as a circuit description language…

16

FSMs and their diagrams

- Datapath shows
  - Moore vs Mealy machine
  - Synchronizers
  - …
- State- and ASM diagrams
  - Shows state transitions
    - conditions (and priority)
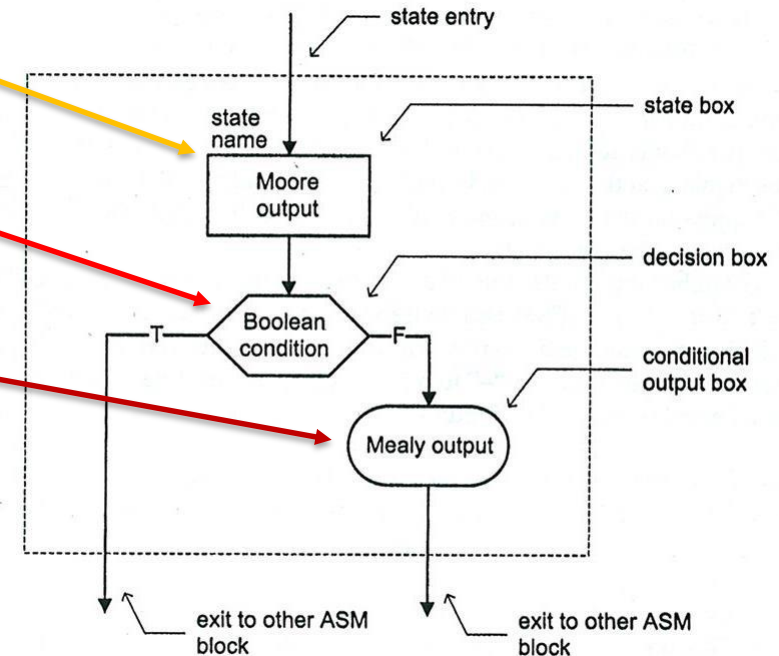 - Output
  - Register operations (ASMD)
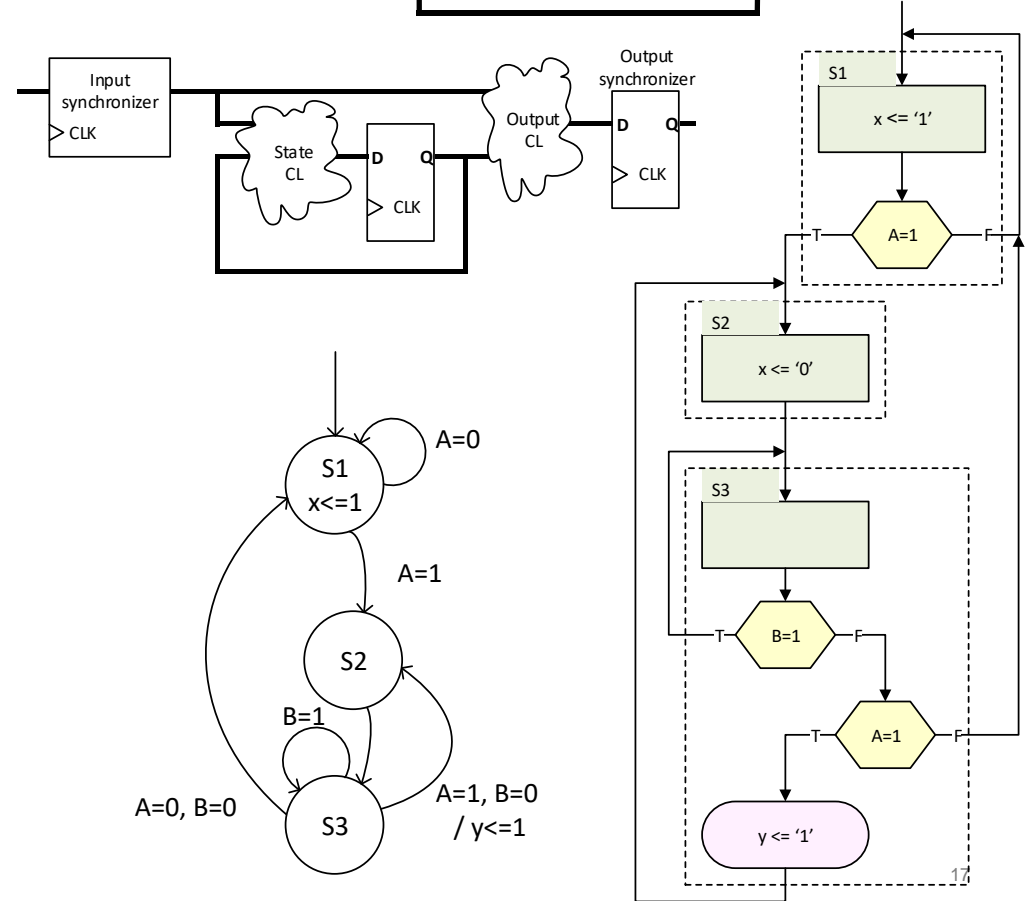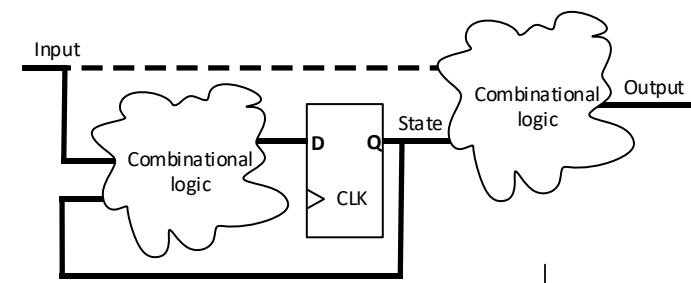
State diagram

- States
- Transitions between states
- Beside transition arc:
  - Descision parameter
  - / Mealy output
- Inside bubble:
  - Moore output

- Frequently used, but not always with all parameters.
- Note: *Default values often omitted*
  - Here: default: x, y = 0 (boolean false)

S1
x<=1

A=0

A=1

S2

B=1

A=0, B=0

S3

A=1, B=0
/ y<=1
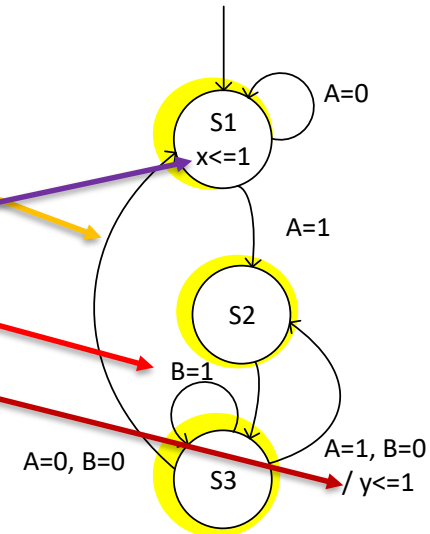
ASM (Algorithmic State Machine) block

- The **state box** represents a state in the FSM,
  - State based output is shown inside (i.e. the **Moore outputs**).

- The **decision box** tests an input condition to determine the exit path of the current ASM block.

- A **conditional output box ("Mealy box")**
  - lists conditionally asserted signals.
  - Can only be placed after an exit path of a decision box
  - (i.e. the **Mealy outputs** that depends on the state and input values).

state entry

state box

state name

Moore output

decision box

T — Boolean condition — F

conditional output box

Mealy output

exit to other ASM block

exit to other ASM block

«Datapath» FSM

- Datapath is described by a function rather than a table
    - Counters
    - Mathematical operations
    - Shift registers
    - Etc.
- We usually divide into control FSM and Datapath
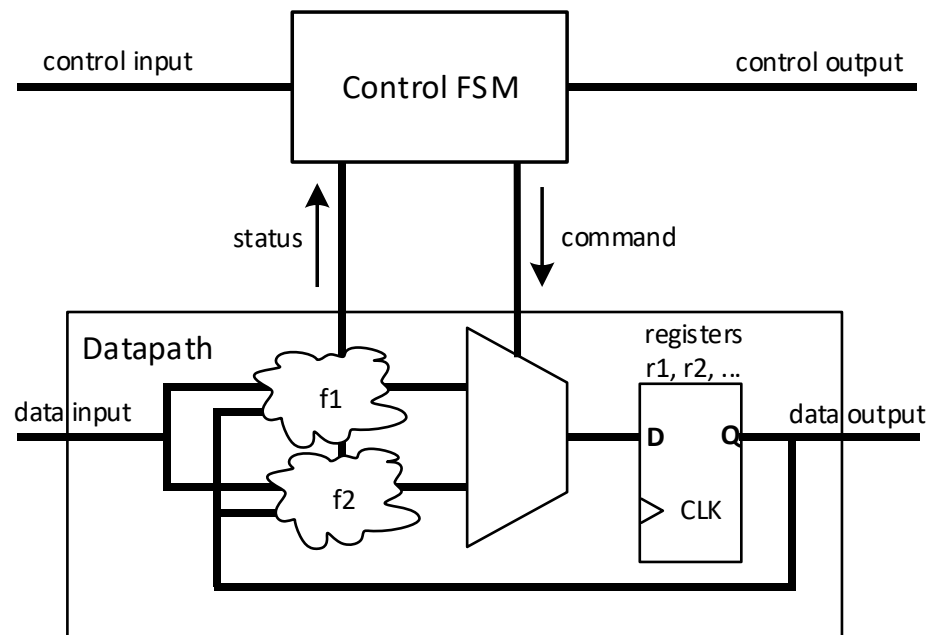
«Register operations» in data-path FSM (FSMD) -and how to deal with it

- Common notations for register operations:
  - on clock edge we increment r1 ──────────────► $r1 \leftarrow r1 + 1$
  - on clock edge we update r1 based on a function of register outputs ──► $r1 \leftarrow f(r1,r2)$
  - on clock edge, set r1 to r2+r3 ──────────────► $r1 \leftarrow r2 + r3$

This notation can be confusing, as it implies one clock delay if it is put into an ASM chart.

Solution:
   Use '←' for datapath only (not for FSM)
   Know that '←' implies the use of registers that
   are not a part of the state machine
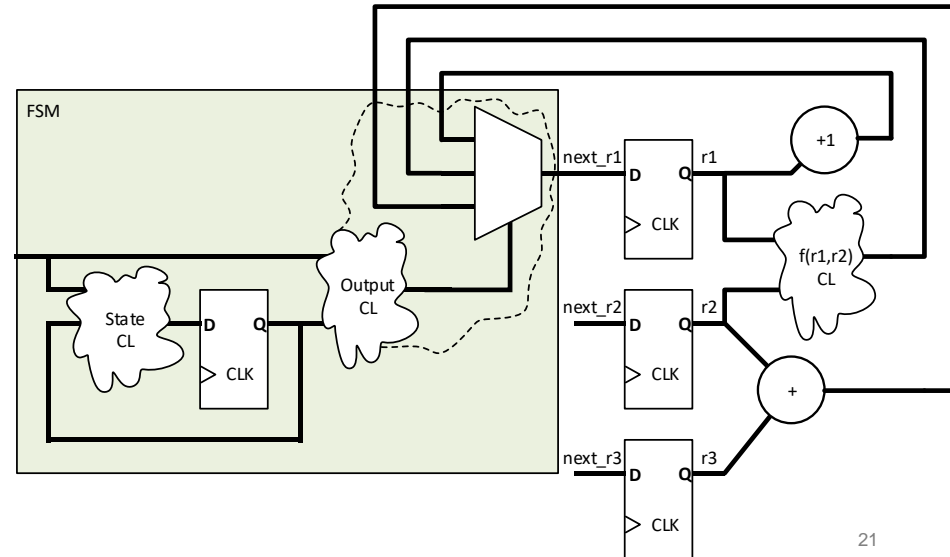
Other drawings or schematics

- System sketches, drawings
  - Usually used to display a concept or an idea
  - Can be anything (*vague block diagram..?*)

- Circuit diagrams
  - Show how the current flows in a circuit
  - Netlists can be made from VHDL
    - (and then turned into circuit diagrams)
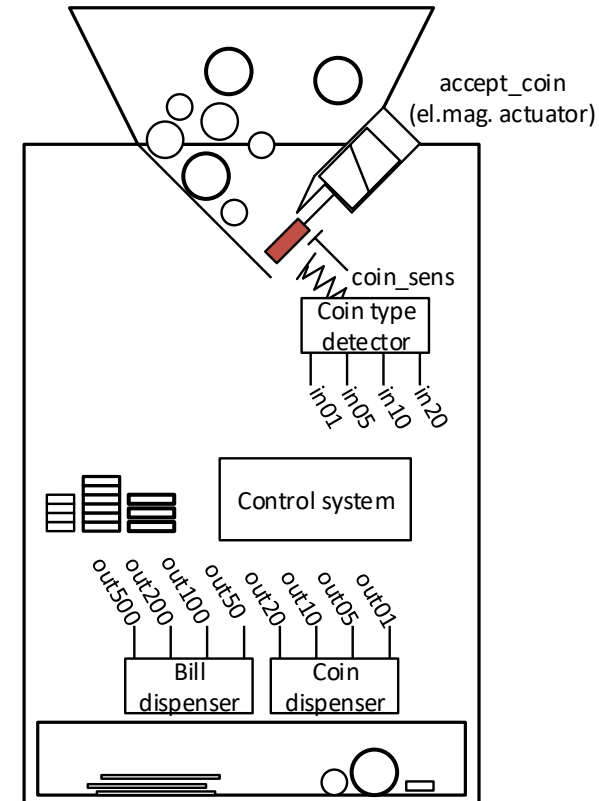
IN3160

Reset circuits

Synchronous or Asynchronous reset?

Source: Steve Kilts: Advanced FPGA Design. 2007

UiO **:** **Department of Informatics**
University of Oslo

Outline

- Combinational logic and floating pins

- Why reset?

- Asynchronous reset

- Synchronous reset

- Reset circuits

Combinational logic and I/O pins

- No need for reset circuitry for CL
  - No values are stored in CL
  - Setting the input will give the desired output

- However... avoid floating gates
  - Floating gates may cause power surges and noise
  - All input pins should be driven
    - Potentially unconnected inputs should be pulled high or low
      - Pull-up or pull-down on PCB or
      - FPGAs may have internal pull up/down circuitry for IO-pins.
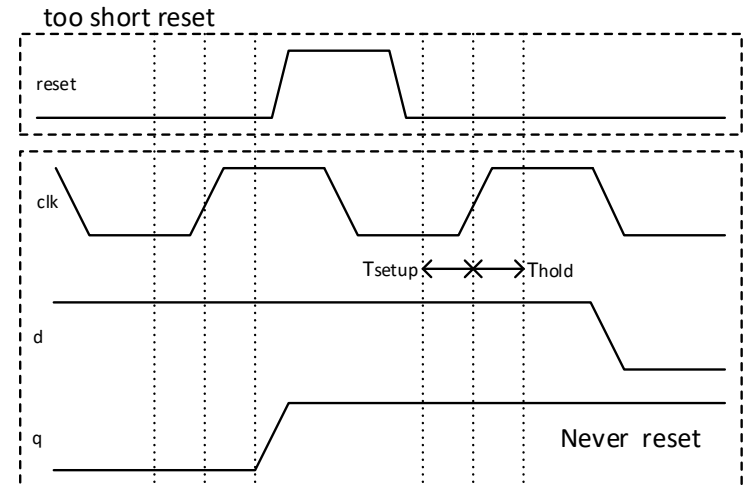   » *Can be* a life saver… (Product/ project / etc).
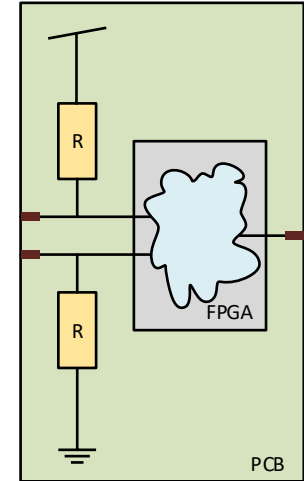
Why reset?

- Avoid unpredictable behavior during startup
  - Metastability creating unpredictable results
    - Both in our system and surrounding systems
  - Random register values may lead to undesired or illegal states
    - Lockup – states with no exit
    - undesired output can have unpleasant consequenses

- To get out of illegal states
  - Unpredicted behaviour may lead to illegal states
  - Noise
    - Crosstalk / EMP
    - Radiation - both thermal and radioactive
    - Floating gates

- ... To ensure verifiable predictability ...

Synchronous reset

```
next_q <= '0' when reset else d;
q <= next_q when rising_edge(clk);
```

- Externally activated resets are per definition asynchronous
  - Synchronization is needed.
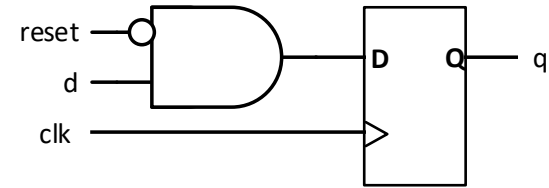
- Synchronous reset 'and flip-flop input
  - Added logic can add to critical path
  - FPGA primitives may have this option built in.

- Reset pulses coming from faster clock domains may be missed entirely.

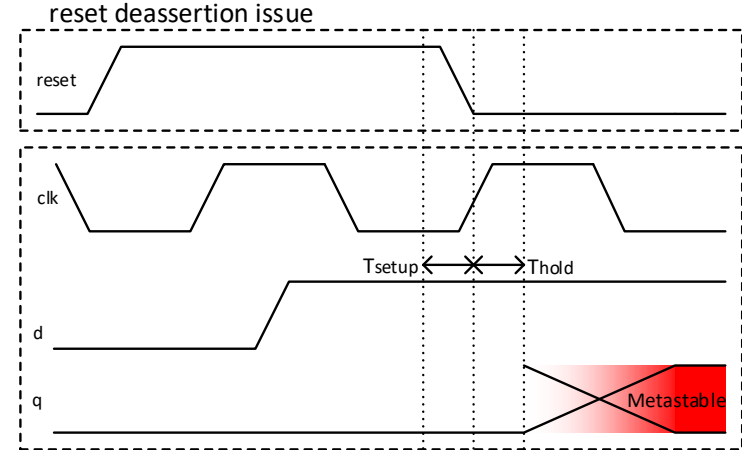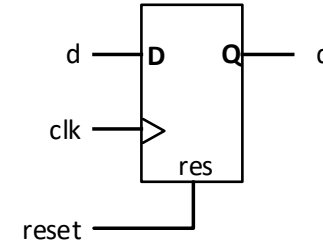too short reset

Asynchronous reset

```
q <= '0' when reset else d when rising_edge(clk);
```


- Asynchronous assertion will always trigger
  – Reset duration must be longer than setup+hold...

- Asynchronous deassertion may cause metastability
  – Deassertion during setup/hold period

reset deassertion issue

reset

clk

Tsetup ✕ Thold

d

q   Metastable

Reset circuit(s)

– Asynchronous assertion,
Synchronous deassertion

- Short reset pulses will trigger
- 2FF mitigates metastability
  – *More in clock domain crossing lecture…*

– Pitfalls?

- Hazards => random reset
- This circuit should not be used
  unless external reset is hazard-free.

– Multiple sources for reset?

- Ensure resetpulses are long enough
- Use Synchronous reset
  – 2FF when crossing domains
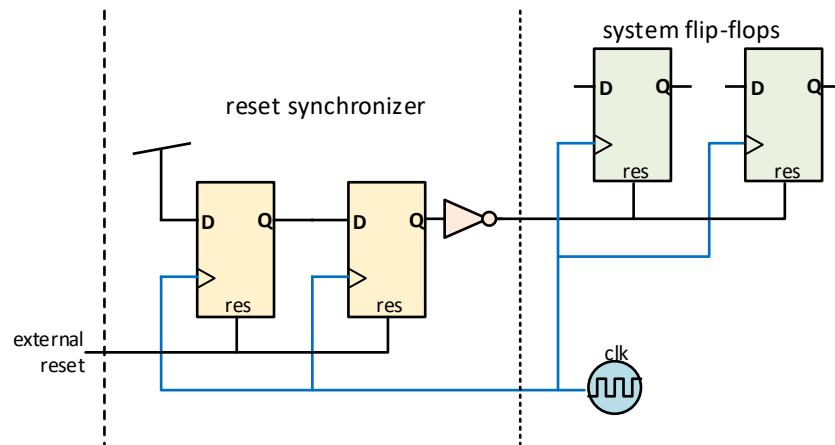

```
ff1 <= '0' when ext_reset else '1' when rising_edge(clk);
ff2 <= '0' when ext_reset else ff1 when rising_edge(clk);
reset <= not ff2;
```

29
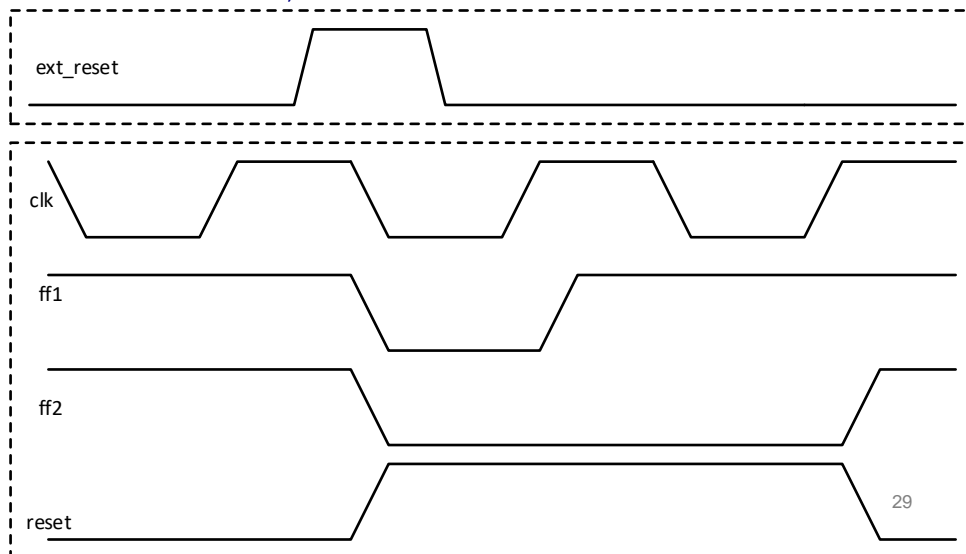
Reset summary

- External reset is asynchronous
  - Should be synchronized to avoid causing metastability.

- It is OK to use asynchronous reset once synchronized…
  - Once synchronized, synchronous reset is perfect…
    - Some FPGA primitives prefer synch reset only.

- Reset pulse must be long enough for reset circuitry
  (depends on technology / logic family, not clk frequency..)

Non-FSM example with ASMD diagram

(Theory covered earlier)

Example: Decade counter

```vhdl
entity dec_count is
  port (
  Clock,
  Reset,
  Enable,
  Load,
  Mode  : in  Std_logic;
  Data  : in  Std_logic_vector(7 downto 0);
  Count : out Std_logic_vector(7 downto 0));
end;
```

Enable	Load	Mode	Next Count
0	0	X	Data
0	1	0	Count+1 (bin)
0	1	1	Count+1 (dec)
1	X	X	Count

```vhdl
architecture doulos_model_solution of dec_count is
  constant nibble_max  : Unsigned(3 downto 0) := "1111";
  constant decade_max  : Unsigned(3 downto 0) := "1001";
  constant zero_nibble : Unsigned(3 downto 0) := "0000";
  constant zero_byte   : Unsigned(7 downto 0) := "00000000";
  signal Q             : Unsigned(7 downto 0);
begin
  process (Clock, Reset)
  begin
    if Reset = '0' then
      Q <= zero_byte;
    elsif RISING_EDGE(Clock) then
      if Enable = '0' then
        if Load = '0' then
          Q <= Unsigned(Data);
        elsif (Mode = '0' and Q(3 downto 0) /= nibble_max) or
              (Mode = '1' and Q(3 downto 0) /= decade_max) then
          Q(3 downto 0) <= Q(3 downto 0) + 1;
        else
          Q(3 downto 0) <= zero_nibble;
          if (Mode = '0' and Q(7 downto 4) /= nibble_max) or
             (Mode = '1' and Q(7 downto 4) /= decade_max) then
            Q(7 downto 4) <= Q(7 downto 4) + 1;
          else
            Q(7 downto 4) <= zero_nibble;
          end if;
        end if;
      end if;
    end if;
  end process;
  count <= Std_logic_vector(Q);
end;
```

Critique design
- Readability
  - Uses negative logic
  - «Mode» requires explanation
- Portability
 - Asynchronous reset

Critique implementation
- Readability
  - Mix of CL and register storage
  - If-based…
  - CL style assignment in clocked process
- Maintainability
  - Not scaleable
  - Duplicate code
    - For each nibble

Decade counter

- A new slightly different design:
  - Synchronous reset
  - High and low indicators
  - Hazard free output
    - **all output in registers**
  - Positive logic
    - enable
    - load
    - decade (nbinary)
    - up (ndown)
  - *Generic ?*
    - *Will be bound by 4bit nibble size*

- New implementation
  - Separate CL and register assignment

- But first:
  - Diagrams & tables
    - Entity
    - ASMD
      - *Algorithmic state machine with data path*
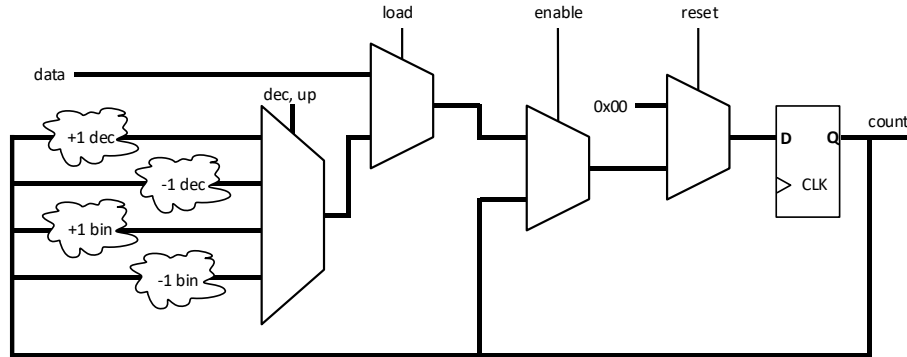
8 bit decade counter
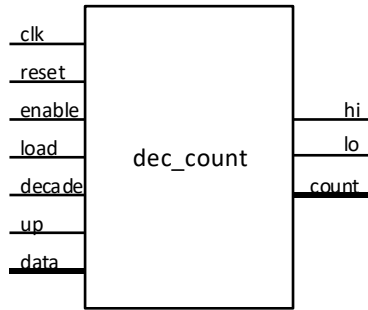
- load, up/down, dec/bin
  - Indicates function
- High/low signal

- Counter
  - not considered FSM
    - State is only by counter value
    - Input and counter value decides output
      - No real state storage
  - More than just counting
    - Complex decision tree
      - Suitable for *data path diagram*
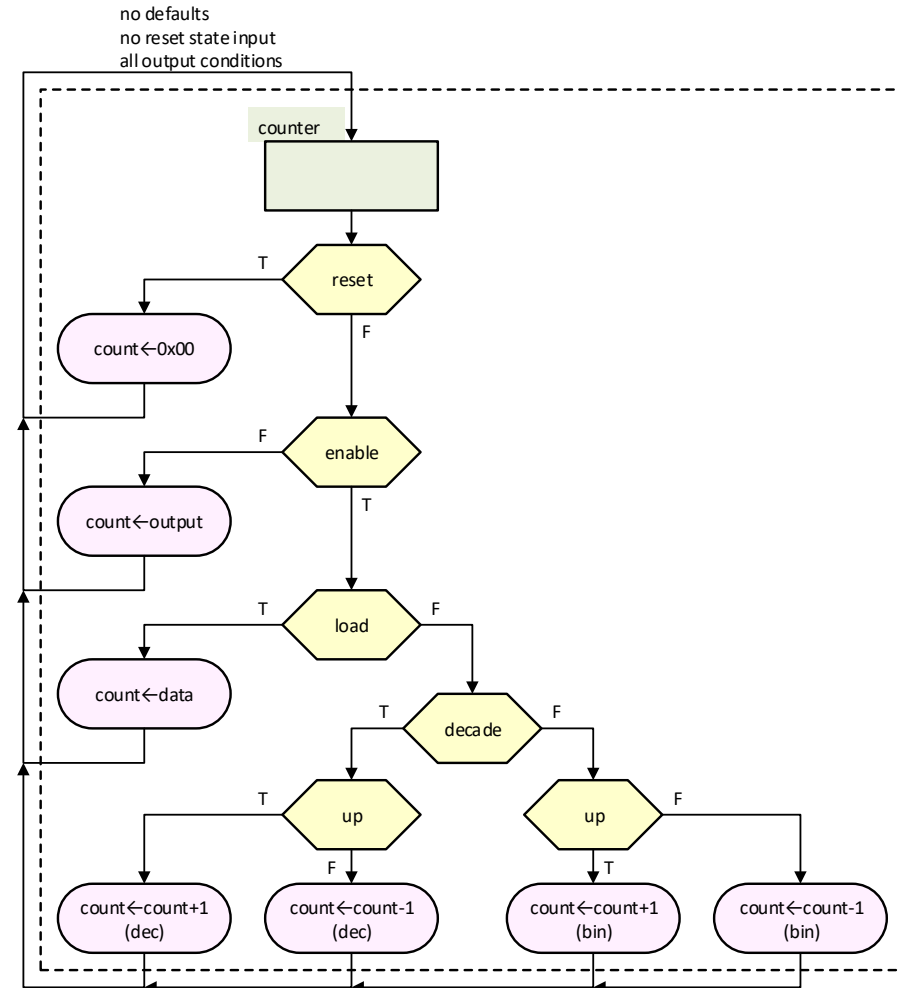   - *ASMD for the sake of instruction*

enable	load	decade	up	next_count
0	X	X	X	count
1	1	X	X	data
1	0	0	1	count+1 (bin)
1	0	0	0	count-1 (bin)
1	0	1	1	count+1 (dec)
1	0	1	0	count-1 (dec)

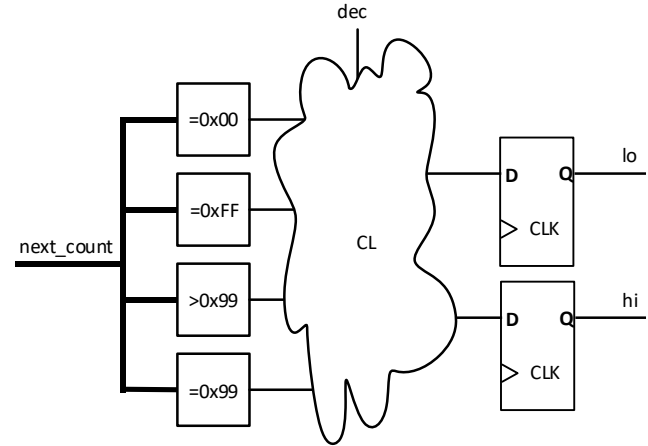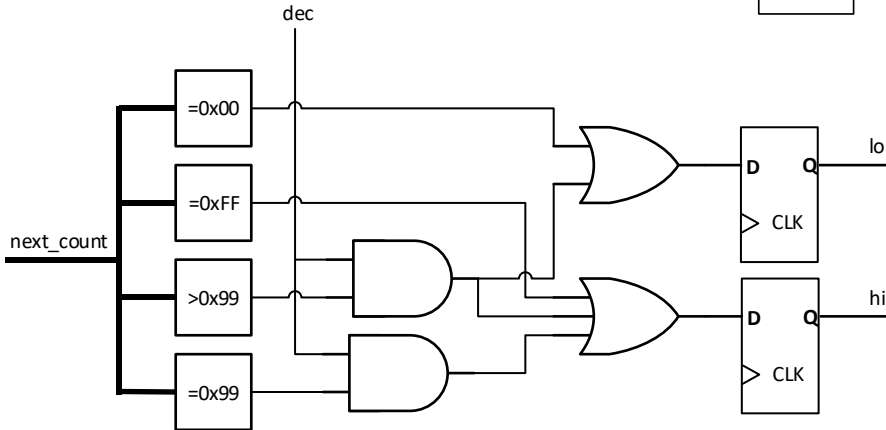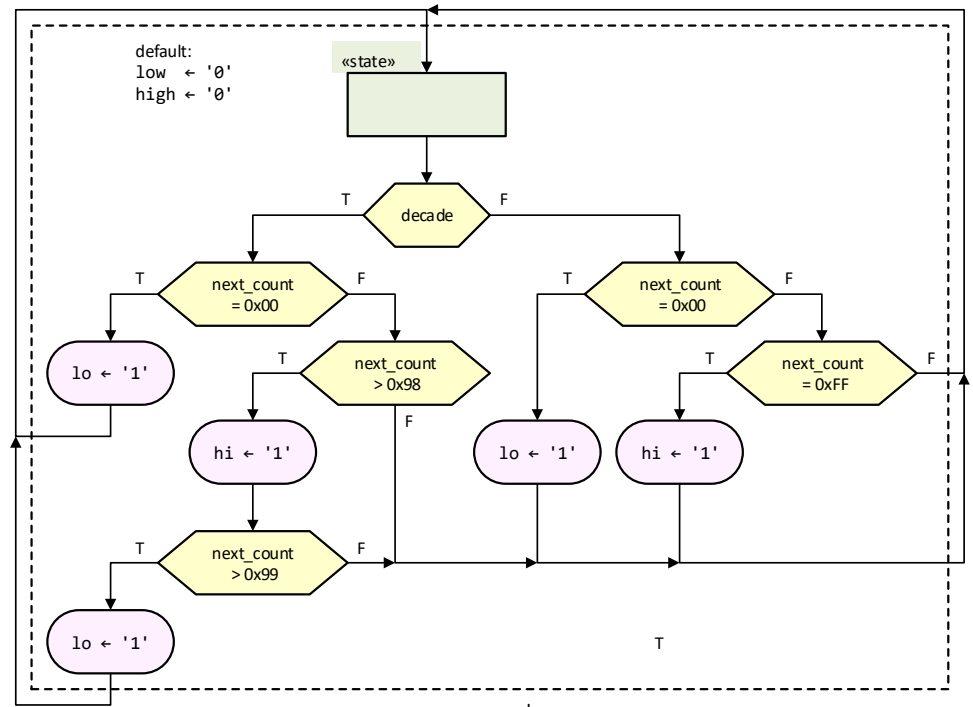| decade	count	high	low
X	0x00	0	1
0	0x01-0xFE	0	0
0	0xFF	1	0
1	0x01-0x98	0	0
1	0x99	1	0
1	0xA0-0xFF	1	1

34

Datapath and ASMD

- What type of conditional statement is most suited?
  - Count conditions and outputs..

Decade counter : hi/ lo signals

- Various level of detail
  – *Do we need unwrap all details?*

36

Decade counter example: entity + constants and register storage

```vhdl
library IEEE;
  use IEEE.STD_LOGIC_1164.all;
  use IEEE.numeric_std.all;

entity dec_count is
generic(
   COUNT_WIDTH : natural := 8);
 port(
   clk, reset, enable, load, decade, up  : in std_logic;
   data    : in std_logic_vector(COUNT_WIDTH-1 downto 0);
   hi, lo  : out std_logic;
   count   : out std_logic_vector(COUNT_WIDTH-1 downto 0)
 );
end entity dec_count;

architecture RTL of dec_count is
  constant DEC_MAX : unsigned(3 downto 0) := "1001";
  constant ZERO_NIBBLE : unsigned(3 downto 0) := "0000";
  constant NIBBLES : integer := COUNT_WIDTH/4;

  signal next_count : unsigned(count'range);
  signal next_hi, next_lo : std_logic;

  alias upper : unsigned(3 downto 0) is next_count(next_count'high downto next_count'high - 3);
  alias lower : unsigned(3 downto 0) is next_count(3 downto 0);

begin
 -- registry update
 count <= std_logic_vector(next_count) when rising_edge(clk);
 hi    <= std_logic(next_hi) when rising_edge(clk);
 lo    <= std_logic(next_lo) when rising_edge(clk);

 -- non generic part -- -- can also be processed in a loop --
 next_hi <= '1' when (and next_count = '1') or ( (decade = '1') and ((lower>8) and (upper >8)) ) else '0';
 next_lo <= '1' when (nor next_count = '1') or ( (decade = '1') and ((lower>9) or (upper >9)) ) else '0';
```
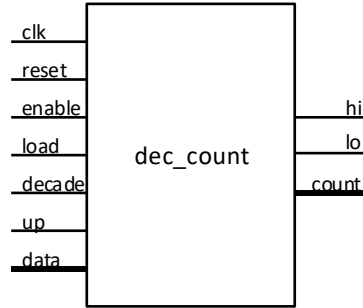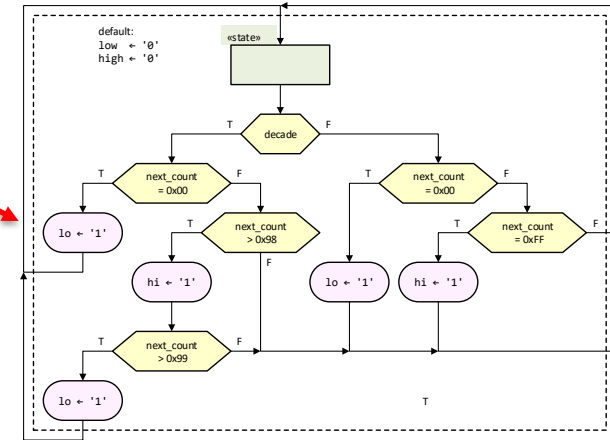
decade	count	high	low
X	0x00	0	1
0	0x01-0xFE	0	0
0	0xFF	1	0
1	0x01-0x98	0	0
1	0x99	1	0
1	0xA0-0xFF	1	1

- Do we need both?

- Reset not part of register storage when synchronous.
- When checking multiple arguments
  - => break down to boolean clauses with >, <, = or /=

no defaults
no reset state input
all output conditions

counter

T — reset — F

count←0x00

F — enable — T

count←output

T — load — F

count←data

T — decade — F

T — up — F

T — up — F

count←count+1 (dec)   count←count-1 (dec)   count←count+1 (bin)   count←count-1 (bin)
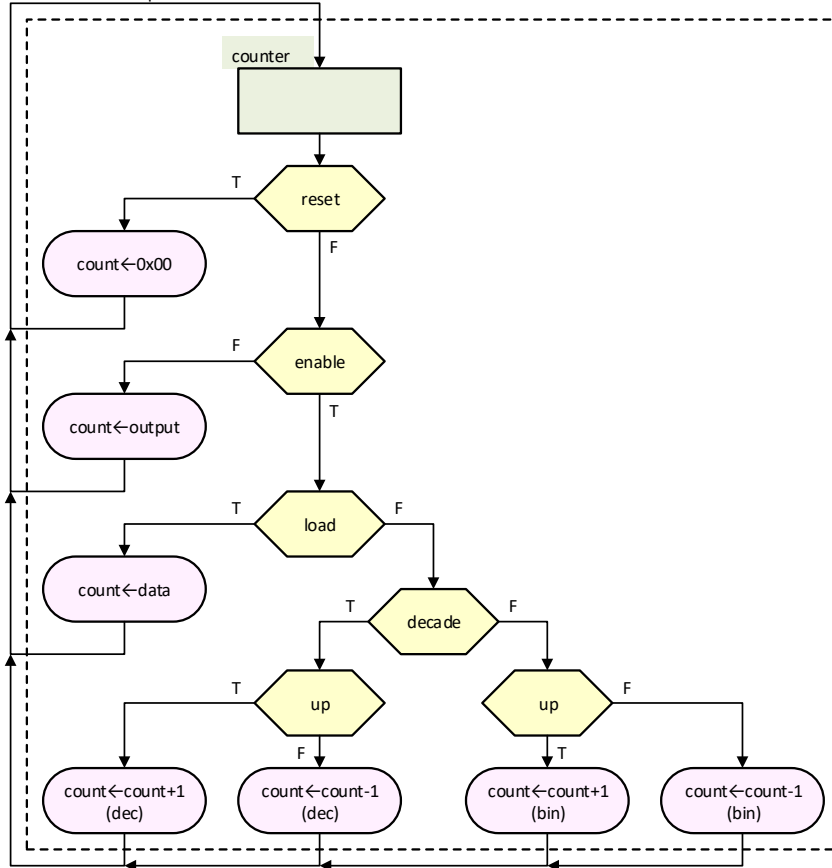
- Decision tree: one output, several conditions => when else
- *There are many ways of performing decade counting*
  - *Indexing requirements = obstacle when reading.*
  - *Use mod or rem will be OK but still 4 bit nibble*

```vhdl
COUNTING: process(all) is
  function count_dec(up: std_logic; dec : unsigned(count'range)) return unsigned is
    variable i_count : unsigned(count'range);
    variable acc : integer;
    variable criteria, nibble : unsigned(3 downto 0);
  begin
    acc := 1                when up else -1;
    criteria := DEC_MAX     when up else ZERO_NIBBLE;
    nibble := ZERO_NIBBLE when up else DEC_MAX;
    i_count := dec; -- by default use the old value.
    for i in 0 to NIBBLES-1 loop
      if dec(i*4+3 downto i*4) = criteria then
        i_count(i*4+3 downto i*4) := nibble;
        next;
      else
        i_count(i*4+3 downto i*4) := dec(i*4+3 downto i*4) + acc;
        exit;
      end if;
    end loop;
    return i_count;
  end function;
begin
  next_count <=
    (others => '0')                          when reset       else
    unsigned(count)                          when not enable else
    unsigned(data)                           when load        else
    count_dec(up, unsigned(count))  when decade      else
    unsigned(count) + 1                      when up          else
    unsigned(count) - 1;
end process;
end architecture RTL;
```

Note: SW-like trick to create priority.. (Works in synthesis)

Before going into function details:

ASM style decision tree is normally easy to implement

Here: up/down used as parameter in function

Suggested reading

- Diagrams
  - These and earlier lecture notes.

- Reset circuits
  - Steve Kilts: Advanced FPGA Design: Architecture, Implementation and Optimization, 2007, chapter 10.
    - download from university library
      Semesterside for IN3160->"Pensum/litteratur i Leganto"