Digital Design, Group Project EENG 28010

Course Overview - Introduce Assignment 2

Dr Faezeh Arab Hassani and Dr Shuangyi Yan

Dept of Electrical and Electronic Engineering

University of BRISTOL

Unit Goals

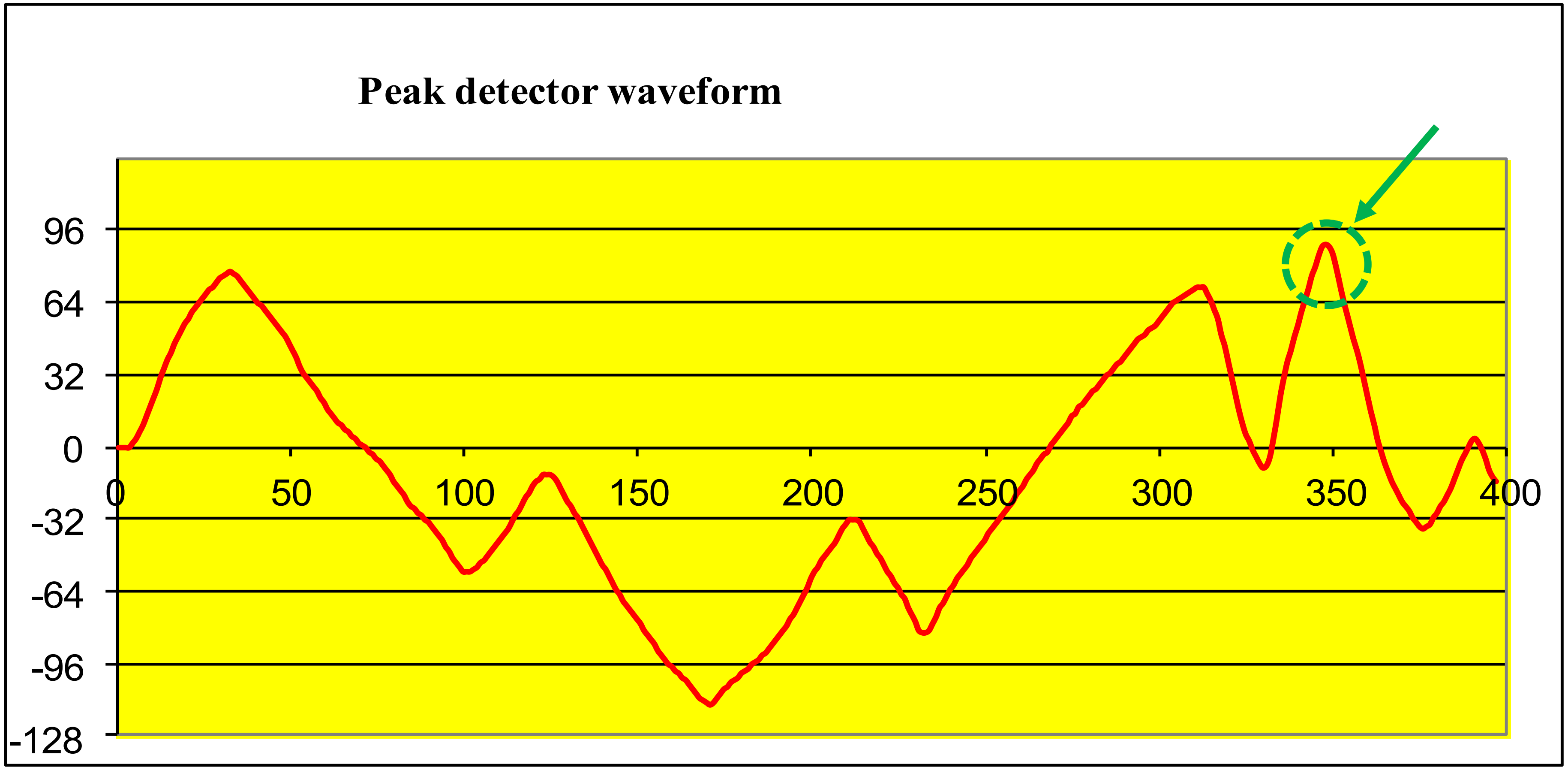
- To gain experience in digital design using VHDL and FPGA prototyping
- To familiarize with related industry standard tools
- To work in a team to design a digital system and implement it in an FPGA

Outline

- Introduction of Assignment 2
- Writing Synthesisable Code

Assignment 2: Peak Detector

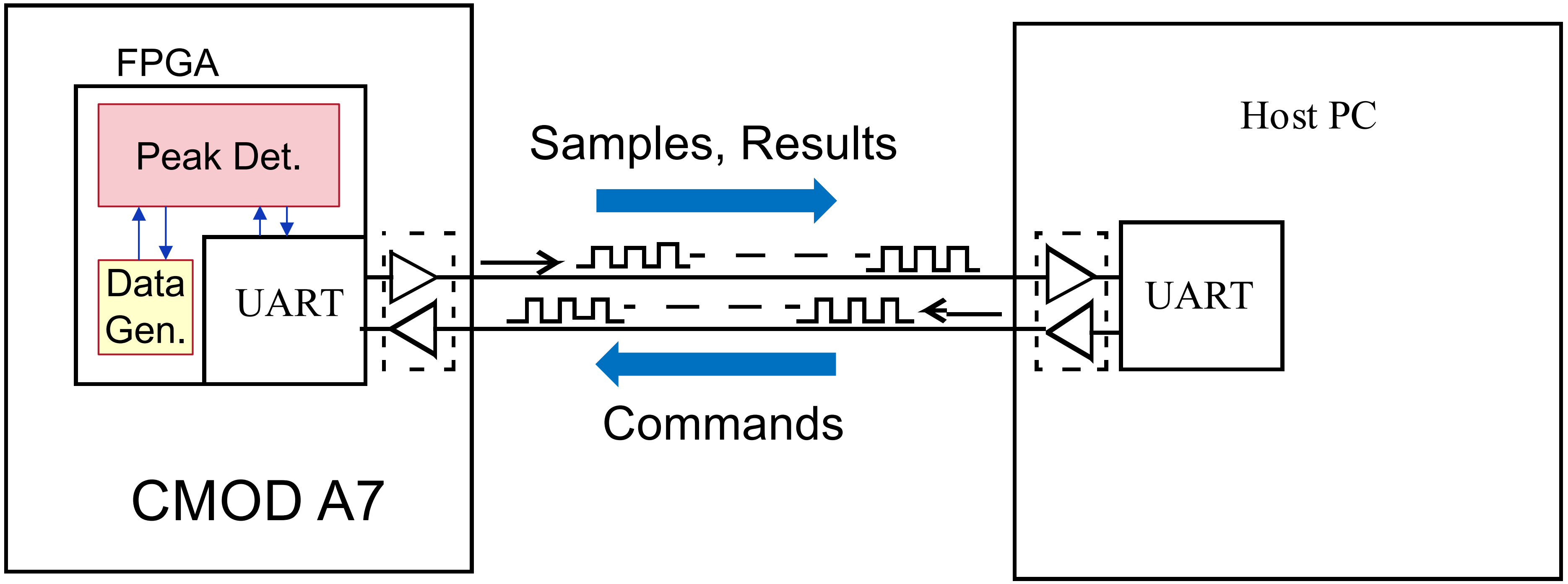
Specification: To design, build and test a peak detector which is under the control of a computer. The system captures up to 500 8-bit digital samples from a source.



University of BRISTOL

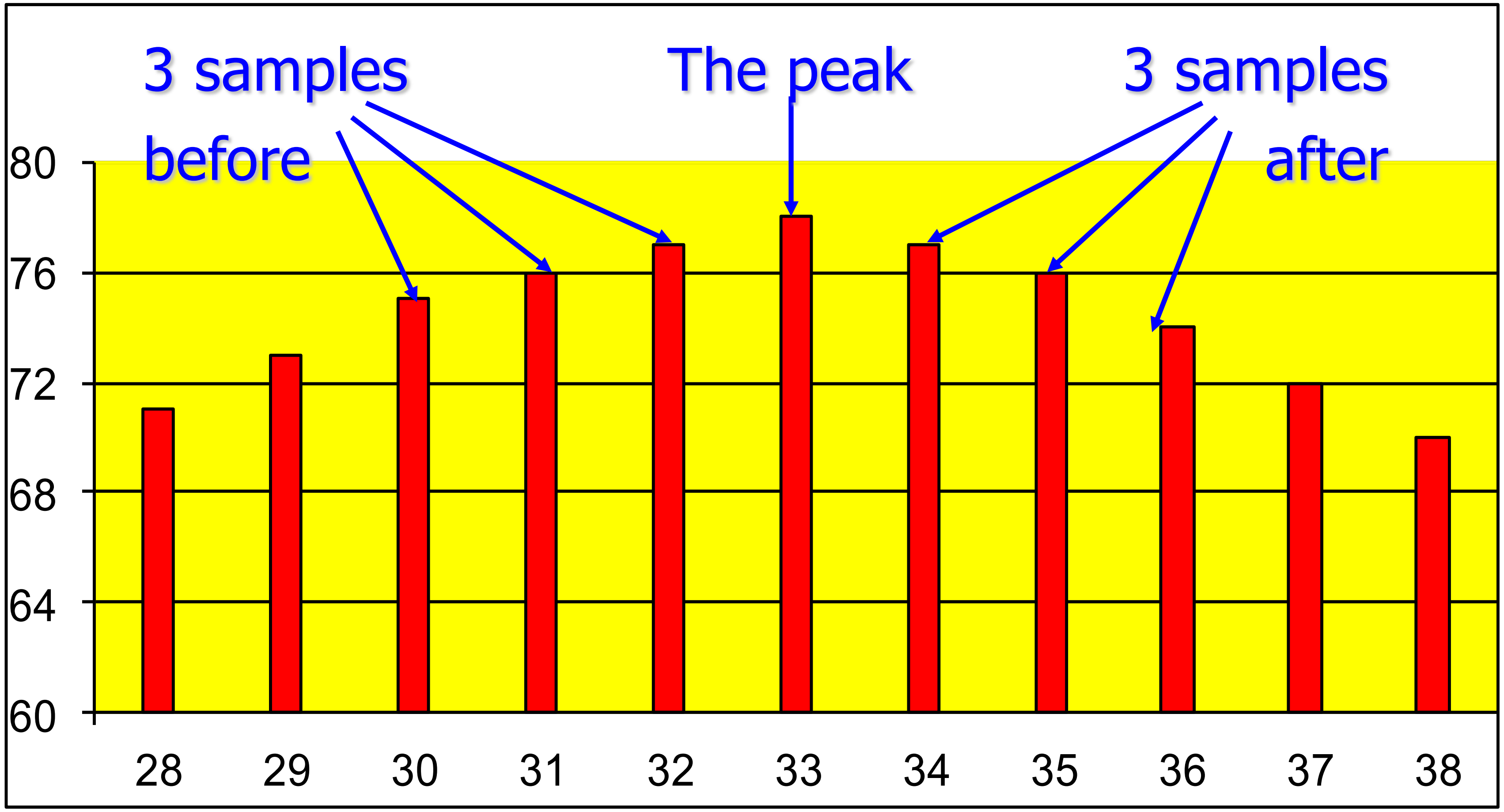
Peak Detector – Serial Communication with PC via UART

- User commands accepted from PC and outputs printed to the PC
- Serial communication implemented by Universal Asynchronous Receiver Transmitter (UART)



Capturing samples

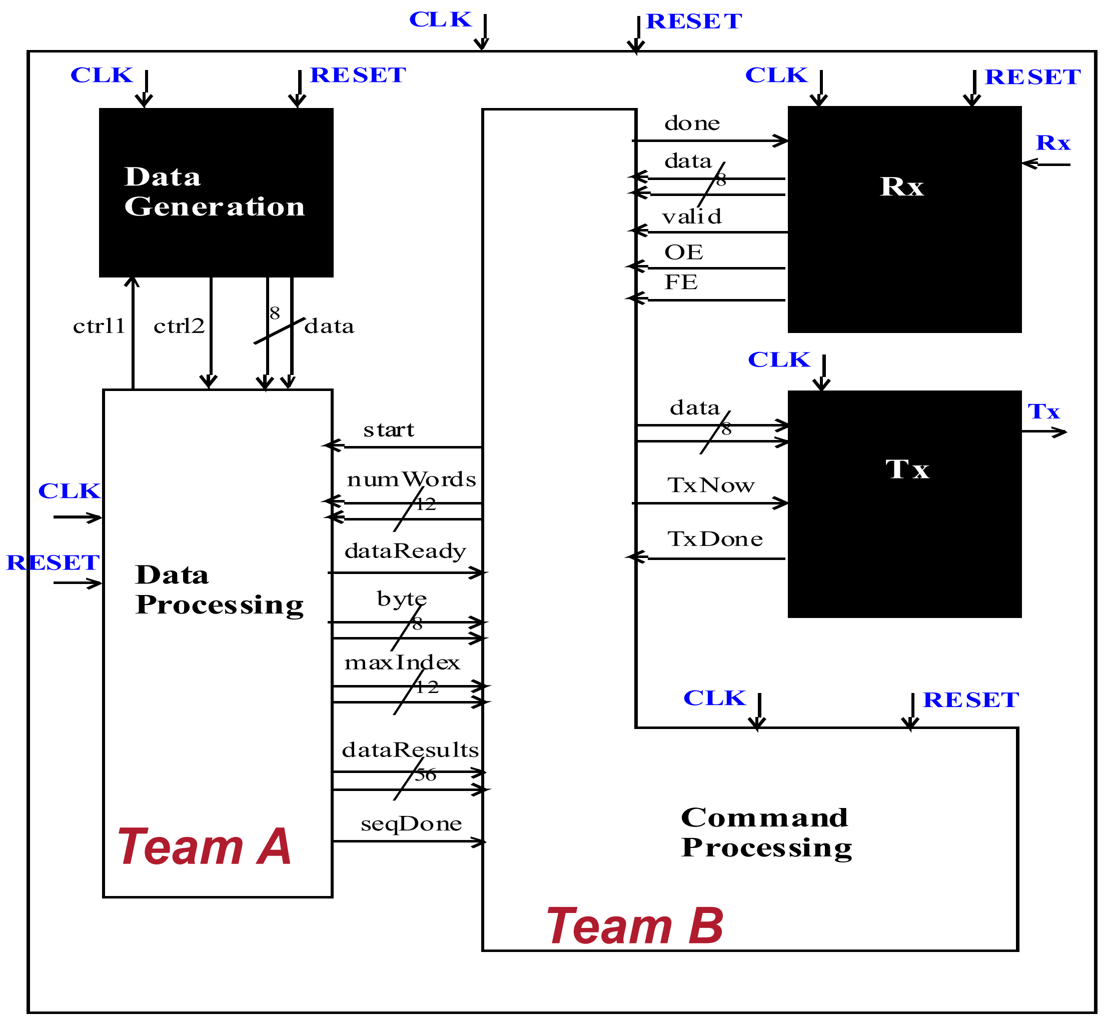
Capture seven points around the highest peak; 3 each side.



Main Requirements

- Receive commands through UART from a computer terminal
  - UART demo is provided
  - Ascii code for printing text in the terminal
- Request amount of data based on commands
  - “aNNN” or “ANNN”
  - Display/print all data
- Analysis peak data and capture the adjacent samples
- Send the data and results back to the computer terminal through UART
  - “P” or ”p”
  - “L” or ”l”

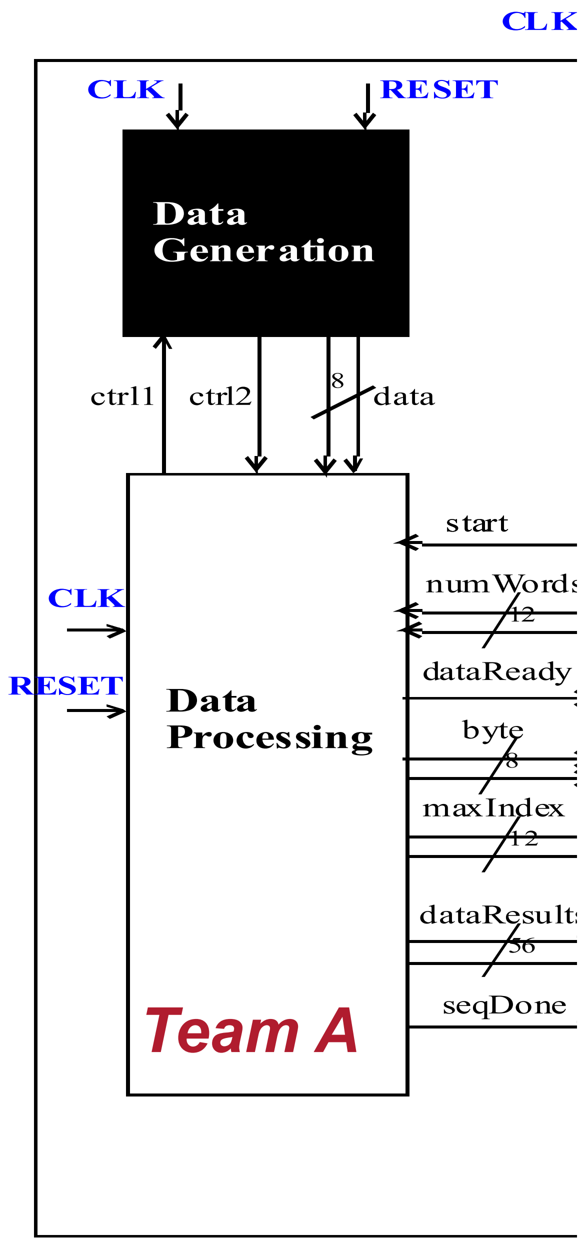
Function requirements



- Data Generator, Tx and Rx has been provided
- Encrypted solutions for individual testing
  - Data Processor
  - Command Processor
- Data Processor
- Command Processor
- Integration simulation
- Synthesis and Implementation

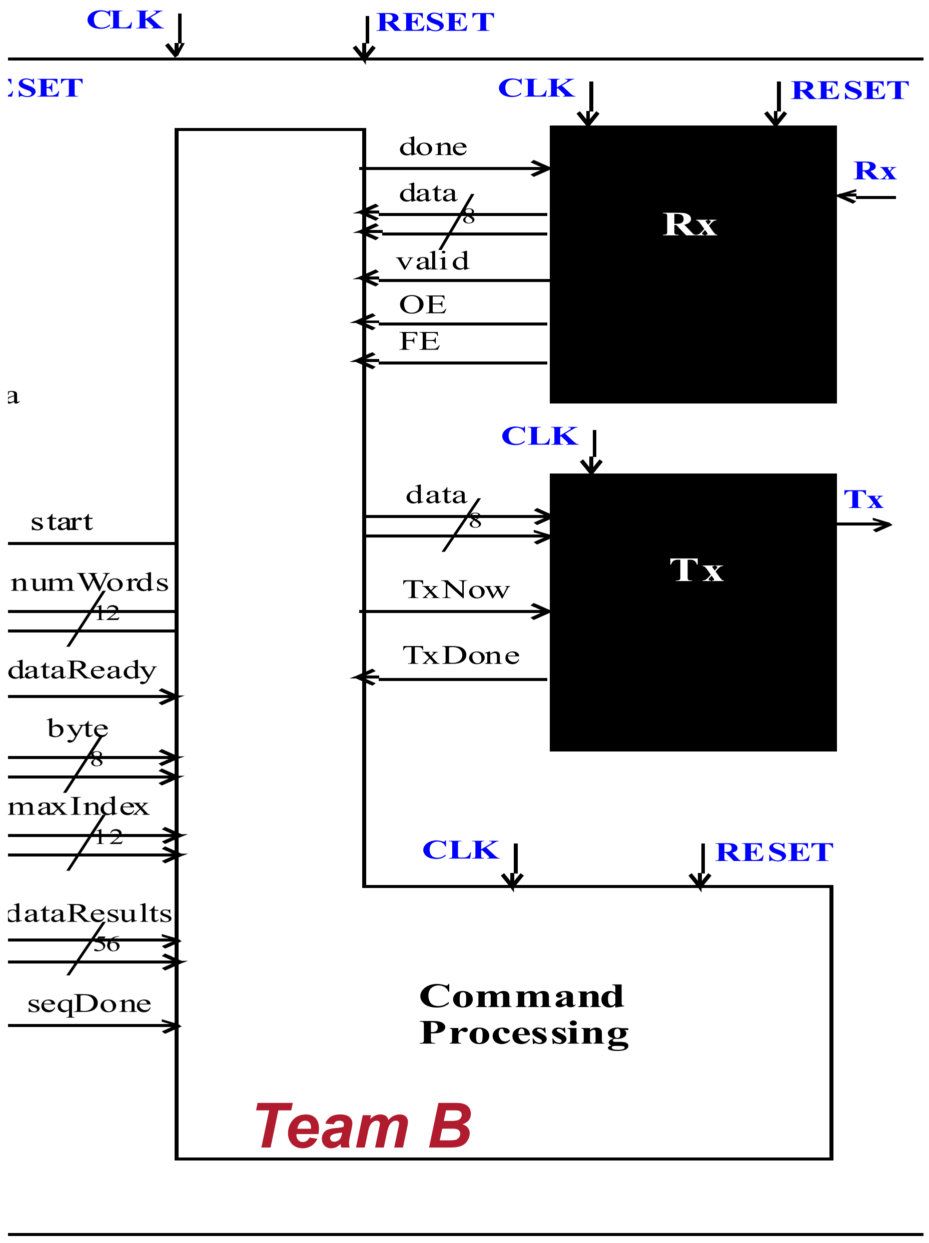
University of BRISTOL

Data Processor



- ”start”: sync data processor and command processor
- Read requirements from CMD processor
- “ctrl 1” and “ctrl 2”: request data
- “dataReady”: inform CMD processor to read data
- Analysis peak and store samples:
  - Peak and other 6 samples
  - Provide results

Command Processor

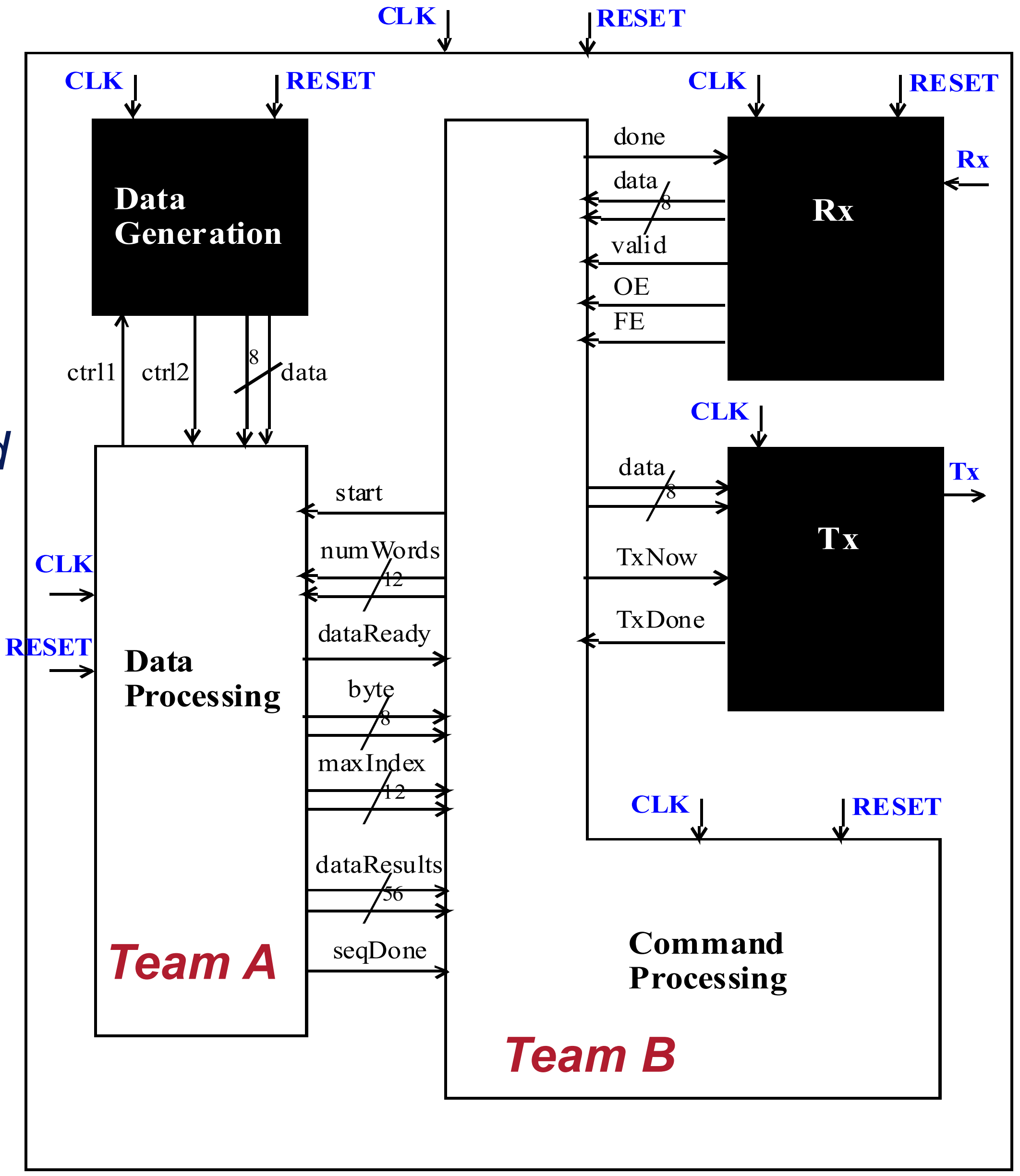


- ❑ Manage UART links
- ❑ CMD validation
- ❑ Request data from Data Processor
- ❑ Print all input commands, results for different commands
  - – Send the ASCII code to Tx

University of BRISTOL

Peak Detector – Work Plan

- Divide group into two sub teams
- *Team A* works on the *Data Processing* block (two members)
- *Team B* works on the *Command Processing* block (three members)
- Clearly defined interface and black box implementations of Data Processor / Command Processor allow independent development



University of BRISTOL

Signed vs. Unsigned implementation

Odd team number: unsigned implementation

Even team number: signed implementation

- ❑ If your group number is odd, you should treat the bytes delivered by the data source as unsigned.
- ❑ If your group number is even, you should treat the bytes delivered by the data source as signed.

- ❑ Affect data processor design
  - comparison between two values

Live Demo

University of BRISTOL

Use knowledge learnt in Assignment 1

- Building FSM diagram
- Using D Flip-Flops
- Synchronous system
  - using x-reg to hold its value
  - detecting edges and applying it in assignment 2
  - Synchronizing two modules with “start” and “dataReady”
- Understanding how to design a system comprising a datapath and controller

University of BRISTOL

Suggested Approach

- Try both demo and simulation
- Start from the FSM diagram
- Separation of the combinatorial logic with state machine
- List all the functions need to by synchronized to the clock
- Manage handshaking protocol between all units
- Use the testbench efficiently
  - You can customize it for your testing.
- Get rid of all the latches we are encountered while synthesizing the code.

Odd group number: unsigned implementation

Even group number: signed implementation

Week 16 - 17

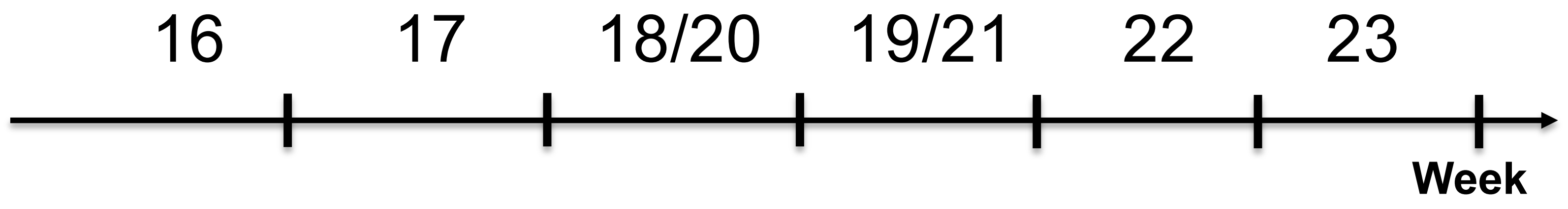
- Start with UART demo
  - Understand the counters used in both Tx and Rx
  - Watch video explanation about TX/RX implementation

- Understand the whole system
  - Read lab note for the system requirements
  - Analysis simulation results based on the provided codes
  - Watch the two-phase signaling protocol

- Work on FSM charts
  - Feedback on charts will be provided on week 19 (Group A) and week 21 (Group B) per group
  - Team meeting with all team members (required full attendance)

Peak Detector – Work Plan *(cont.)*

- Task 1: Command Processing and Data Processing modules should work independently and satisfy specifications according to a software testbench
- Task 2: Integrate Cmd Proc and Data Proc to realise the full working design and validate by full system simulation and FPGA implementation



Week 16 – 21 (4 Weeks): Complete design of data processor and command processor. Prepare drafts for your individual chapter.

Week 22: Integration and testing individual components

Week 23: System integration, simulation and report

Deadline and Marking

Assignment 2: Design of a Peak Detector	Final Design	Group Assignment	Submit code and group report	13:00 Thursday 28 April	65%

Assignment 2: (In total: 65%) Including below sections:

- Report: 20% for all members
- Peer assessment: 5%
- Code:
  - Individual marking of Data processing and Command processing (30%), Team A and B get individual marks
  - Data processing and Command processing integration marking (10%) for all members

Submission Package

- VHDL code for the whole system (40%)
  - Data Processor
  - Command Processor
  - Any developed modules

- Team report (20%)

- Peer assessment (5%), to be included in your team report

Report Contents

- Around 10-15 pages
- Chapter 1: Description of the group composition and task division between group members.
  - Each student needs to summarize their contributions.
- Chapter 2: Design and simulation of Data Processing module including a block-level sketch of the main logic components and a state diagram (FSM or ASM) (Sub Team A)
- Chapter 3: Design and Simulation of Command Processing module including a block-level sketch of the main logic components and a state diagram (FSM or ASM) (Sub Team B)
- Chapter 4: Description of the system integration and test of your final implementation on FPGA.

  Readable screenshots with description should be provided to demonstrate the required functions.
- Peer Assessments

FPGA Collection and Return

- One FPGA per group
  - collect today or after class at MVB 4.23

- Return deadline: May 5th, 2023 to
  - MVB 4.23

Group Work

- Sharing Work
  - Two teams in each group for cmd and data procs
  - Within each team, have tasks defined along structural lines
  - Don’t..
    - Allocate one person for simulation and one for synthesis
    - Allocate one person for debugging only or writing TBs
    - Have one person loosely associated with both teams for e.g., trouble-shooting
    - Above are all excuses for not doing fair share
    - Change roles and responsibilities of a group member without consultation within group

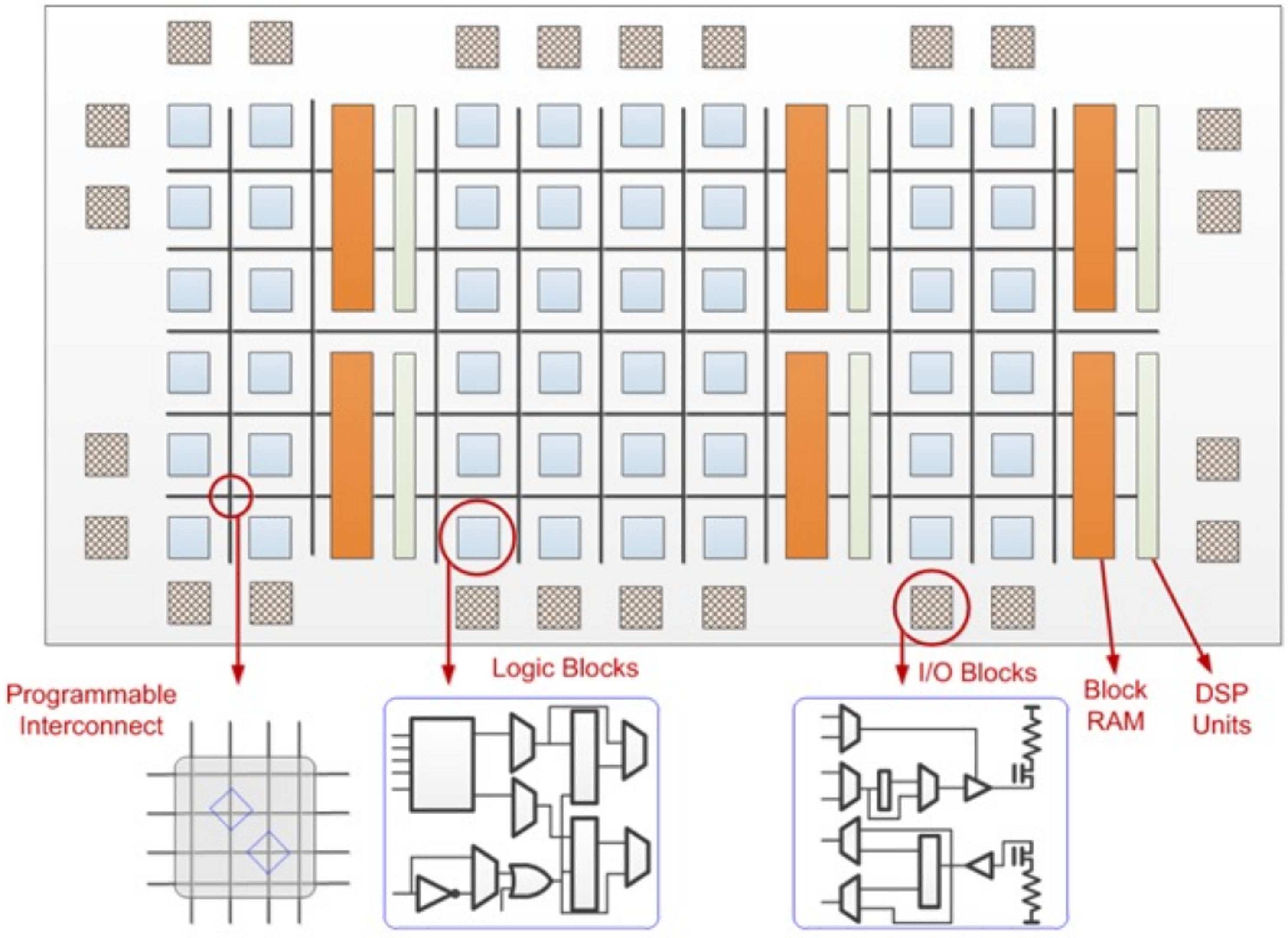
Group Work

- Planning
  - Do…
    - Have a project plan with timelines, milestones and regular meetings
    - Have a record of who does what and minutes of group meetings
    - These feed into interim report
    - Use official e-mail and agree on platform for information exchange
- Difficulties with group members
  - Try to resolve through a conversation
  - If it doesn’t work, talk to Shuangyi or Faezeh.
  - Don’t…
    - Ignore issues until last moment

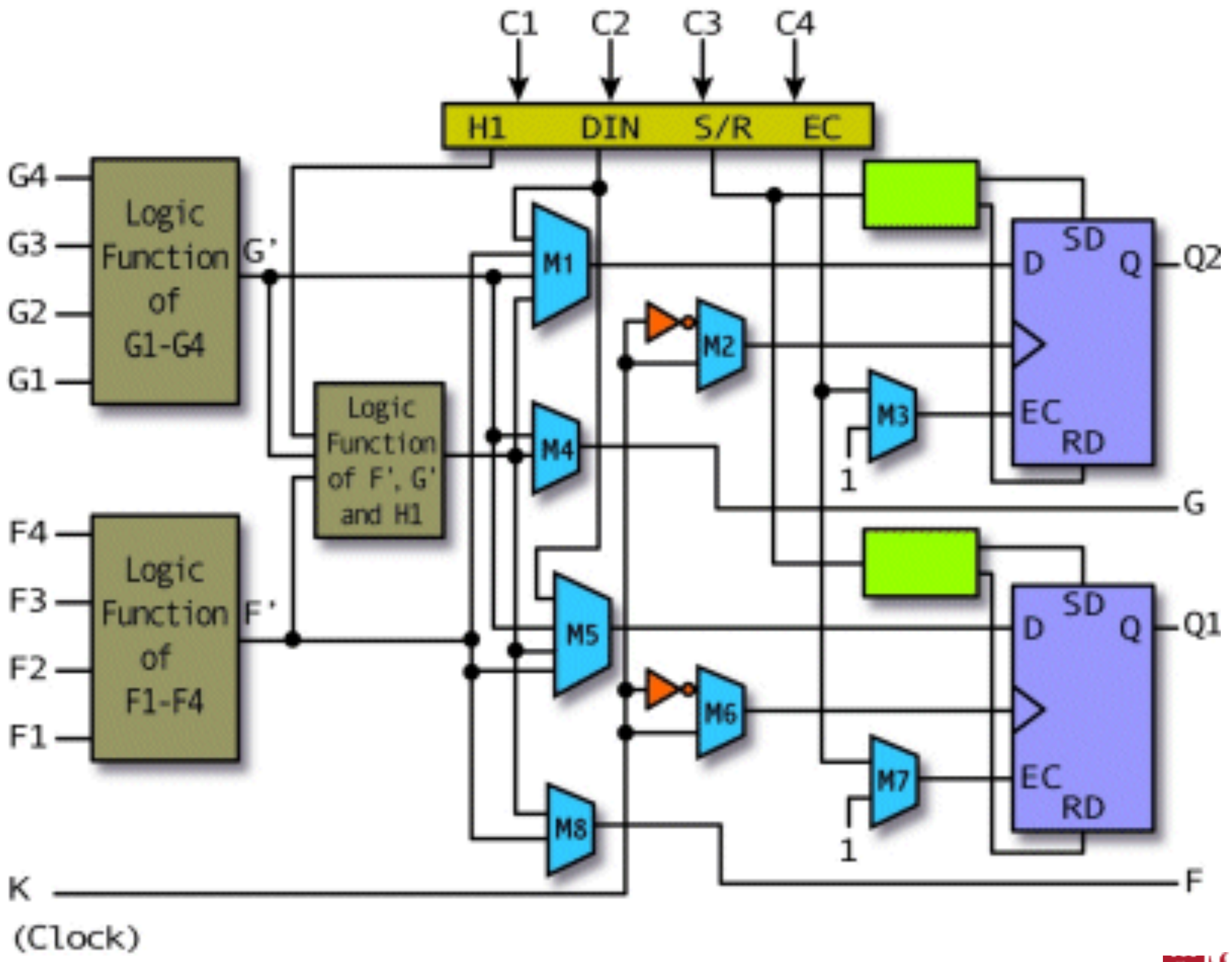
Outline

- ❑ Assignment 2
- ❑ Writing Synthesisable Code
  - – Generating hardware from code constructs
  - – Type conversion
  - – Avoiding unwanted latches

FPGA Structure



Configurable logic block

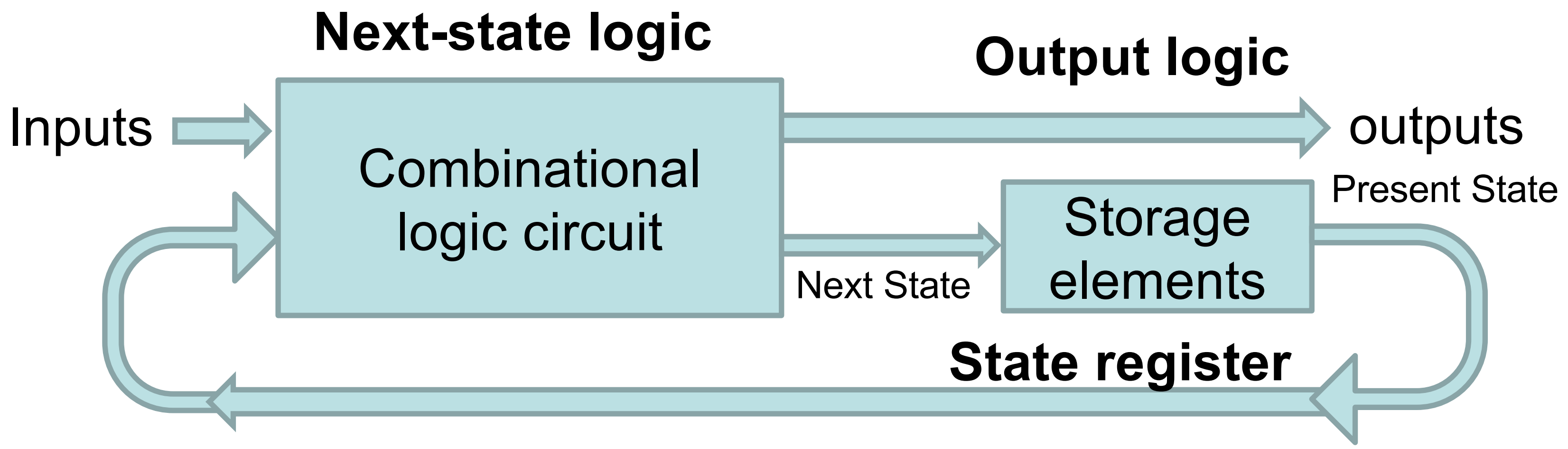


University of BRISTOL

Basic architecture of sequential logics

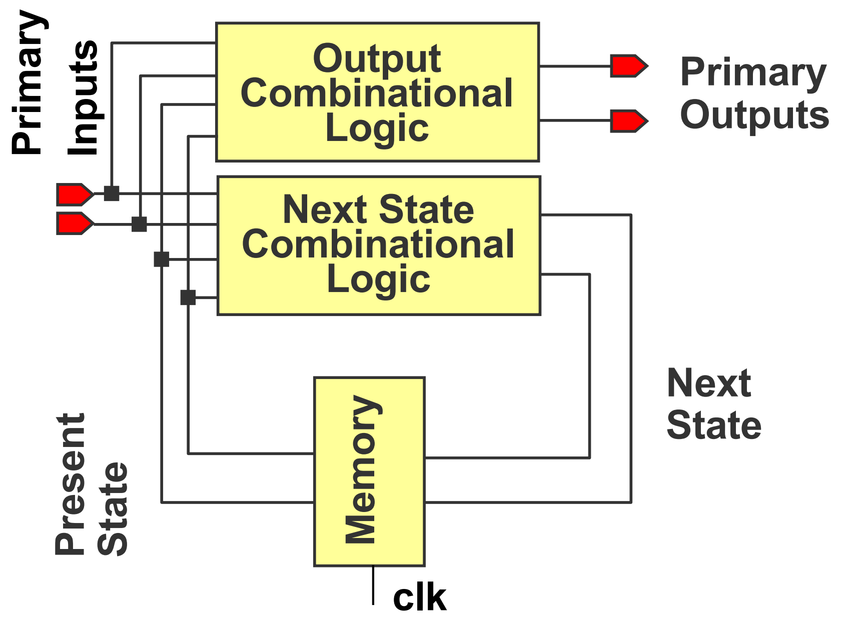
Three blocks:

- **State register**: a collection of D FFs controlled by the same clock signal
- **Next-state logic**: Combinational logic determine new value of the registers
- **Output logic**: combinational logic to generate output signals

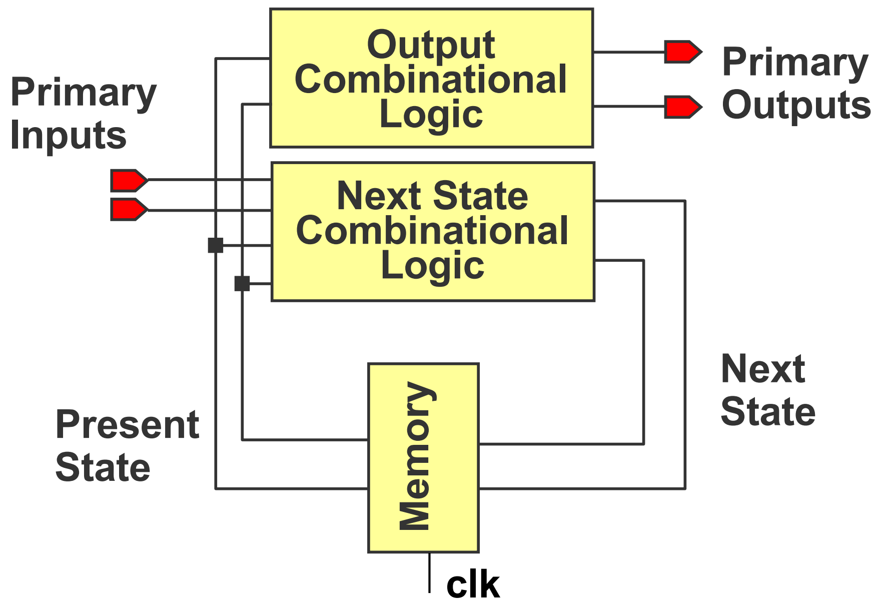


University of BRISTOL

Mealy and Moore Machines



Mealy Machine



Moore Machine

- Most straightforward way to synthesise a state machine is to use a process for each function
  - next state (combinational)
  - output (combinational)
  - memory (sequential)

Modelling a Mealy Machine

```
entity pattern_recog is
port ( X       : in     bit;
       CLK     : in     bit;
       RESET      : in     bit;
       Y          : out   bit );
end;
```

State register

```
seq: process (clk, reset)
  begin
    if reset = '0' then
            CURRENT_STATE <= S0;
    elsif clk'event AND clk='1' then
            CURRENT_STATE <= NEXT_STATE;
    end if;
  end process; -- seq
```

Output logic

```
combi_output: process(CURRENT_STATE, X)
  begin
              case CURRENT_STATE is
     when S0 =>
              Y <= 0;      ....
                           ....
```

Next-state logic

```
combi_nextState: process(CURRENT_STATE, X)
  begin
    case CURRENT_STATE is
      when S0 =>
                    if X='0' then
               NEXT_STATE <= S1;
                    else
               NEXT_STATE <= S0;
                    end if;
      when S1 =>
                    if X='0' then
               NEXT_STATE <= ?;
                    else
                    ....
      when Sn =>
                    if X='0' then
               NEXT_STATE <= ?;
                    else
               NEXT_STATE <= ?;
                    end if;
    end case;
  end process; -- combi_nextState
```

Unsigned and Signed Types

Source: ref [4]

- Used to represent numeric values:

TYPE	Value	Interpretation
unsigned	0 to $2^N - 1$	
signed	$- 2^{(N-1)}$ to $2^{(N-1)} - 1$	2's Complement

- Usage similar to std_logic_vector:

```
signal A      : unsigned (3 down to 0);
signal B      : signed (3 downto 0);
signal C      : std_logic_vector (3 downto 0);
....
A <= “1111”;     --  decimal 15
B <= “1111”;     --  decimal –1
C <= “1111”;     --  decimal 15 if using std_logic_unsigned
                 --  (Synopsys library which extends std_logic_arith)
```

Synthesis Support for Numeric Operations

Source: ref [4]

- Generally, if the numeric_bit and numeric_std packages are supported, the operators within them are supported
- Arithmetic operators - “abs”, “+”, “-”, “*”, “/”, “rem”, “mod”
- Comparison operators - “>”, “<”, “<=”, “>=”, “=”, “/=”
- Shift functions: “shift_left”, “shift_right”, “rotate_left”, “rotate_right”, “resize”
  - Use these library defined functions rather than the built-in VHDL constructs “sla”, “sra”, “sll” etc for shifting and concatenation
- Conversion functions: “to_integer”, “to_unsigned”, “to_signed”
- use ieee.numeric_std.all;

University of BRISTOL

Packages for Numeric Operations

Source: ref [4]

- All synthesis tools support some type of arithmetic packages
- Synopsys developed packages based on std_logic_1164 package - supported by many other synthesis tools
  - std_logic_arith
  - std_logic_signed
  - std_logic_unsigned

use IEEE.STD_LOGIC_ARITH.all

use one or the other, never both

- Actel synthesis tools support their own package
  - asyl.arith
- IEEE has developed standard packages for synthesis IEEE Std. 1076.3
  - Numeric_Bit
  - Numeric_Std

use IEEE.NUMERIC_STD.all

Recommendation:
For new design, use numeric_std

University of BRISTOL

Overloading Examples

Source: ref [4]

```
signal A_uv, B_uv, C_uv, D_uv, E_uv : unsigned(7 downto 0) ;
signal R_sv, S_sv, T_sv, U_sv, V_sv : signed(7 downto 0) ;
signal J_slv, K_slv, L_slv : std_logic_vector(7 downto 0) ;
signal Y_sv : signed(8 downto 0) ;
. . .
-- Permitted
A_uv <= B_uv + C_uv ;  -- Unsigned + Unsigned = Unsigned
D_uv <= B_uv + 1 ;     -- Unsigned + Integer = Unsigned
E_uv <= 1 + C_uv;      -- Integer + Unsigned = Unsigned
R_sv <= S_sv + T_sv ;  -- Signed + Signed = Signed
U_sv <= S_sv + 1 ;     -- Signed + Integer = Signed
V_sv <= 1 + T_sv;      -- Integer + Signed = Signed
-- only legal if using std_logic_unsigned (which you should not!)
J_slv <= K_slv + L_slv ;

-- Illegal Cannot mix different array sizes
Y_sv <= A_uv - B_uv; -- want signed result
-- Solution is to carry out type conversions
```

University of BRISTOL

Automatic Type Conversion: Unsigned, Signed <=> std_logic

Source: ref [4]

- Two types convert automatically when both are subtypes of the same type

```
SUBTYPE std_logic IS resolved std_ulogic;
```

  - Converting between std_ulogic and std_logic is automatic

```
-- all legal
B_sul <= K_sv(7) ;
L_uv(0) <= C_sl ;
M_slv(2) <= N_sv(2);
Y_sl <= A_sl and B_sul and J_uv(2) and K_sv(7) and M_slv(2);
```

Type Casting: Unsigned, Signed <=> Std_Logic_Vector

Source: ref [4]

- Use type casting to convert equal sized arrays when:
  - Elements have a common base type (i.e., std_logic)
  - Indices have a common base type (i.e., Integer)

```
A_slv <= std_logic_vector( B_uv ) ;
C_slv <= std_logic_vector( D_sv ) ;
G_uv <= unsigned( H_slv ) ;
J_sv <= signed( K_slv ) ;
```

```
-- Eg: unsigned – unsigned = signed
signal X_uv, Y_uv : unsigned (6 downto 0) ;
signal Z_sv : signed (7 downto 0) ;
. . .
Z_sv <= signed('0' & X_uv) - signed('0' & Y_uv) ;
```

University of BRISTOL

Numeric_Std Function Conversions: Unsigned, Signed <=> Integer

Source: ref [4]

- Converting to and from integer requires a conversion function

```
signal A_uv, C_uv : unsigned (7 downto 0) ;
signal Unsigned_int : integer range 0 to 255 ;
signal B_sv, D_sv : signed( 7 downto 0) ;
signal Signed_int : integer range -128 to 127

-- unsigned, signed => integer
Unsigned_int <= TO_INTEGER ( A_uv ) ;
Signed_int <= TO_INTEGER ( B_sv ) ;

-- interger => unsigned, Signed
C_uv <= TO_UNSIGNED ( unsigned_int, 8 ) ;
D_sv <= TO_SIGNED ( signed_int, 8 ) ;

-- Eg: indexing an array
Data_slv <= ROM(  TO_INTEGER( Addr_uv)  ) ;
```

Width of array

Avoid using latches

- Latches are generally "a bad thing" - unless you *really* know what you are doing.
- In synthesis, latches are not an error - they will be reported as either information or warning
- In mapping and place and route, you will not get any warnings - the mapping and place and route tools just do what the synthesis tool tells them to do!
- But you can often spot latches in the place and route summary, either as elements containing strings such as LD or DL; or explicitly listed as latches.

Avoiding Latches

- Where do these evil latches come from?
  - You directly instantiated them or inferred them in your code on purpose
    - Valid technique for meeting timing requirements in aggressive designs (e.g., borrowing time from the next cycle)
    - DO NOT PRACTICE IN THIS UNIT!
  - You inferred them in your code by accident
    - In processes modelling combinational logic, all the signals on Right hand side (RHS) of assignment must appear in sensitivity list.
    - In processes modelling combinational logic, you have incompletely specified if-else or case statements, or read before assigning.
    - In processes modeling sequential logic, you have more than a single clock and asynchronous control in the sensitivity list.

Inferring a Latch

```
ENTITY what IS
PORT (clk, d: IN std_logic;
    q: out std_logic);
END what;

ARCHITECTURE behav OF what IS
BEGIN
    PROCESS(clk, d)
    BEGIN
        IF clk = '1' THEN
            q <= d;
        END IF;
    END PROCESS;
END behav;
```

- VHDL semantics require that q retains its value over the entire simulation run
- Not assigned in each branch
- Latch inferred as assignment is under the control of a level-sensitive condition

Do not use latches!

University of BRISTOL

Mistakenly Inferring Latches with CASE

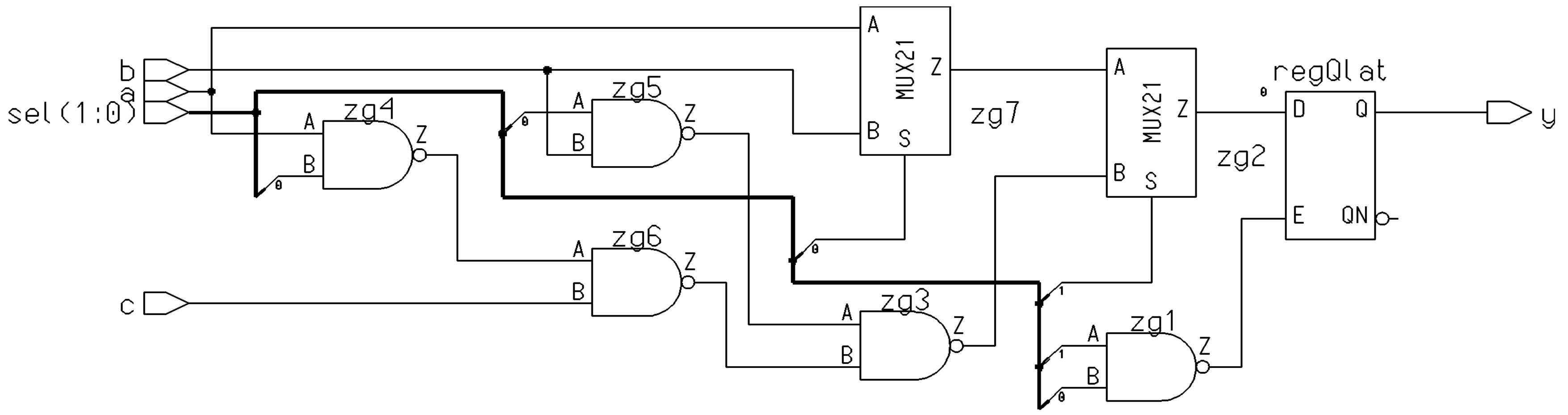
Latch Creation via Incomplete Assignment in Conditional Assignment:

- Not assigning all conditional expressions in a CASE structure

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY mux3_seq is
  PORT(a   : IN  std_logic;
       b   : IN  std_logic;
       c   : IN  std_logic;
       sel : IN std_logic_vector(1 DOWNTO 0);
       y   : OUT std_logic);
END mux3_seq;
```

```
ARCHITECTURE behavior OF mux3_seq IS
  BEGIN
    comb : PROCESS(a,b,c,sel)
      BEGIN
        CASE sel IS
          WHEN "00" => y <= a;
          WHEN "01" => y <= b;
          WHEN "10" => y <= c;
          WHEN OTHERS => --empty
        END CASE;
    END PROCESS comb;
END behavior;
```



Avoiding Unwanted Latches

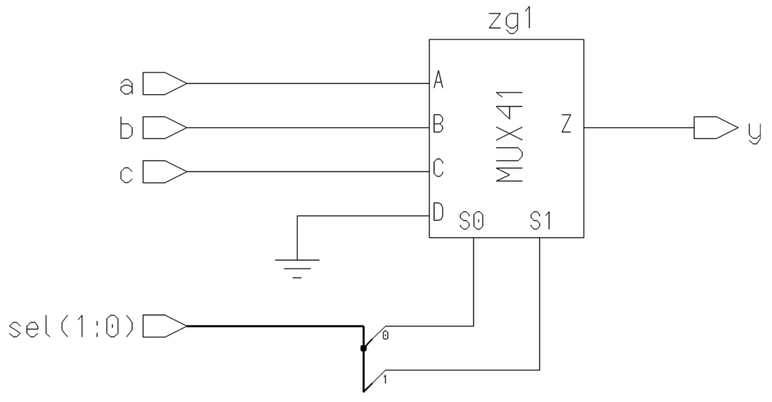
- Assigning values of “don’t care” (‘-’) in these cases can avoid unwanted latch inference

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY mux3 is
  PORT(a    : IN  std_logic;
       b    : IN  std_logic;
       c    : IN  std_logic;
       sel  : IN std_logic_vector(1 DOWNTO 0);
       y    : OUT std_logic);
END mux3;

ARCHITECTURE behavior OF mux3 IS

  BEGIN
    comb : PROCESS(a,b,c,sel)
      BEGIN
        CASE sel IS
          WHEN "00" => y <= a;
          WHEN "01" => y <= b;
          WHEN "10" => y <= c;
          WHEN OTHERS => y <= '-';
        END CASE;
    END PROCESS comb;
END behavior;
```



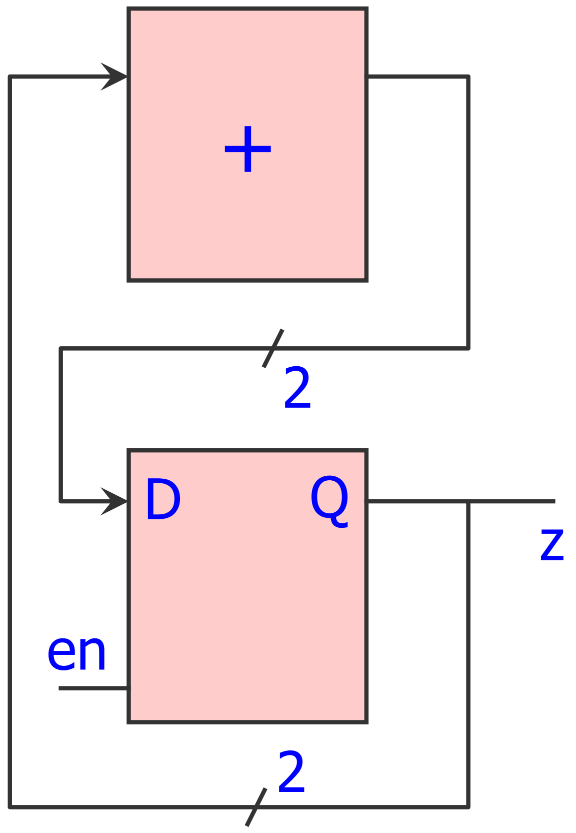
Latches are generated mostly by combinational logic!

Be careful when you are writing combinational code!

Wrong Implementation of a Counter

```
ENTITY incr IS
PORT (en: IN std_logic;
      z: out unsigned(0 to 1));
END incr;

ARCHITECTURE behav OF incr IS
BEGIN
      PROCESS(en)
            VARIABLE count: unsigned(0 to 1);
      BEGIN
            IF en = '1' THEN
                  count := count +1;
            END IF;
            z <= count;
      END PROCESS;
END behav;
```

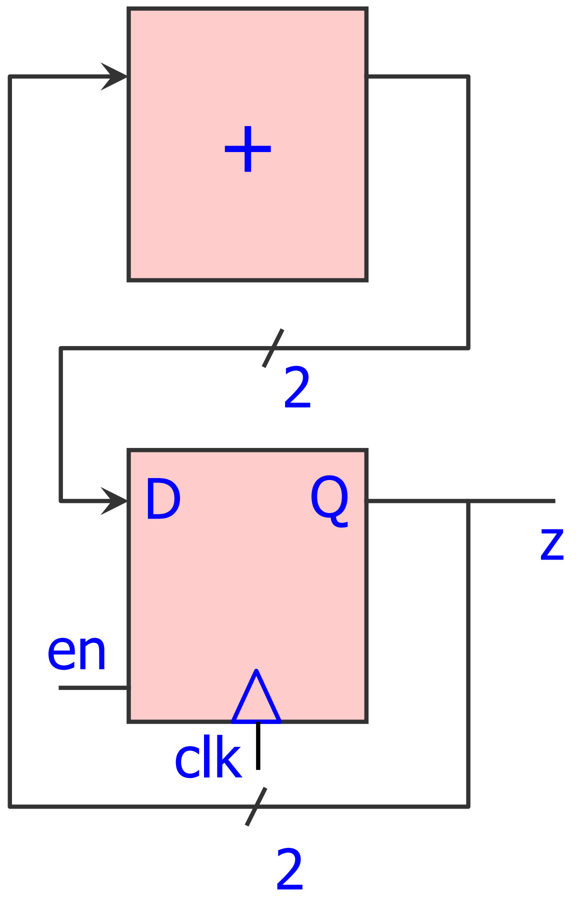


Do not use latches!

Correct Implementation of a Counter

```
ENTITY incr IS
PORT (en: IN std_logic;
      z: out unsigned(0 to 1));
END incr;

ARCHITECTURE behav OF incr IS
BEGIN
      PROCESS(clk)
            VARIABLE count: unsigned(0 to 1);
      BEGIN
            IF rising_edge(clk) THEN
                  IF en = '1' THEN
                        count := count +1;
                  END IF;
            END IF;
            z <= count;
      END PROCESS;
END behav;
```



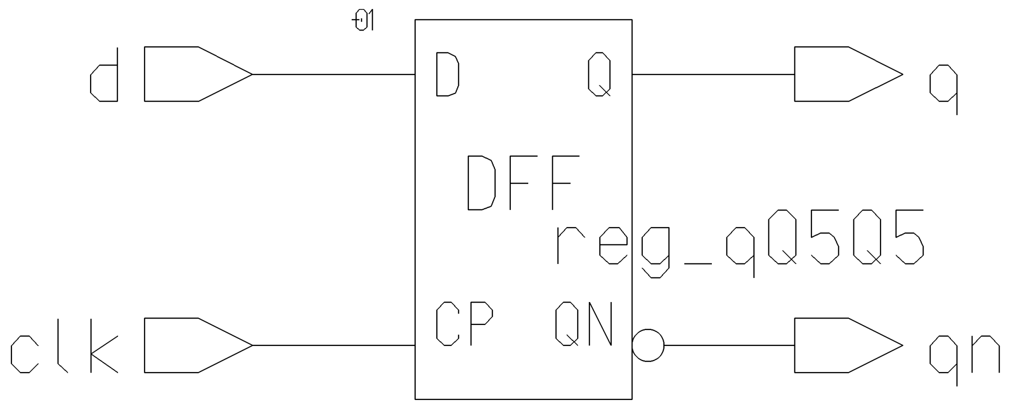
Edge Sensitive D Flip-Flop

- Assignment on rising or falling edge of control signal (Clock)
  - **rising_edge()** and f**alling_edge()** functions can be used to specify clock edge
- Memory inferred because “q” is not assigned for all cases

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY d_ff is
  PORT(d   : IN  std_logic;
       clk : IN  std_logic;
       q   : OUT std_logic;
       qn  : OUT std_logic);
END d_ff;

ARCHITECTURE behavior OF d_ff IS
  BEGIN
    seq : PROCESS(clk)
      BEGIN
        IF(rising_edge(clk)) THEN
          q <= d;
        END IF;
    END PROCESS seq;
    qn <= not q;
END behavior;
```



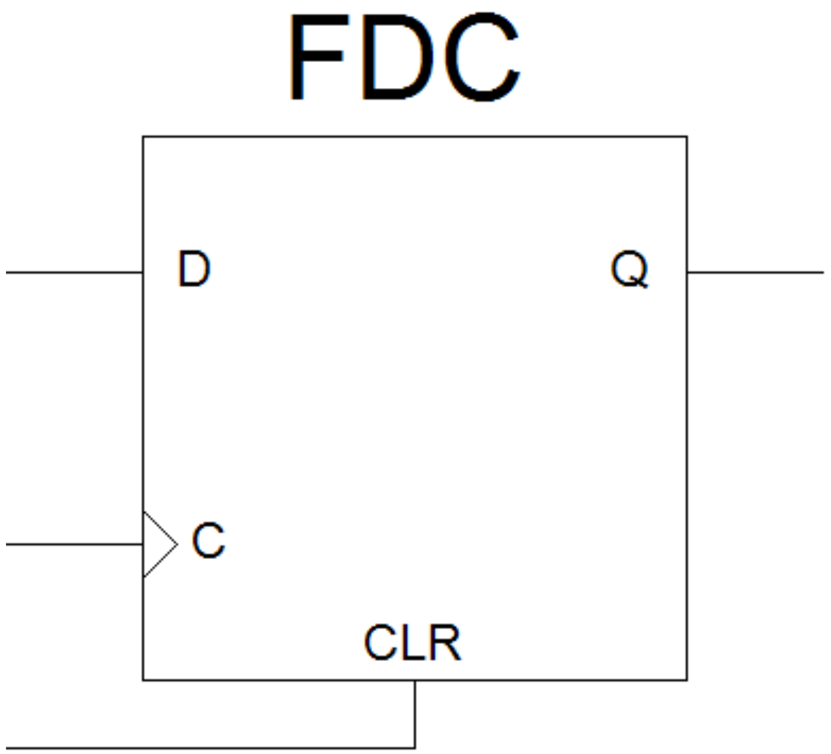
Edge Sensitive D Flip-Flop with Synchronous Reset

- Synchronous reset, so only clk in sensitivity list
- Memory inferred because “q” is not assigned for all cases

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY d_ff is
  PORT(d   : IN  std_logic;
       clk : IN  std_logic;
       q   : OUT std_logic);
END d_ff;

ARCHITECTURE behav OF FlipFlop IS
BEGIN
    PROCESS(clk)
    BEGIN
        IF clk‘event AND clk= ‘1’ THEN
           IF reset = ‘0’ THEN
                q <= ‘0’;
           ELSE
                q <= d;
           END IF;
        END IF;
    END PROCESS;
END behav;
```



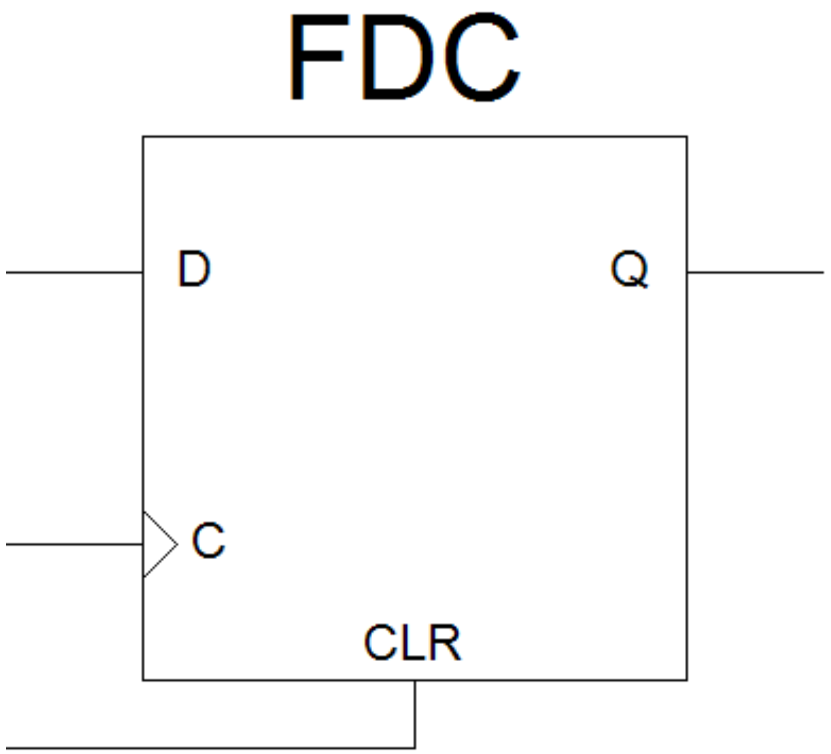
Edge Sensitive D Flip-Flop with Asynchronous Reset

- The only signals in the process sensitivity list other than CLOCK should be asynchronous signals
- Memory inferred because “d” is not assigned for all cases

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY d_ff is
  PORT(d   : IN  std_logic;
       clk : IN  std_logic;
       q   : OUT std_logic);
END d_ff;

ARCHITECTURE behav OF FlipFlop IS
BEGIN
    PROCESS(clk, reset)
    BEGIN
        IF reset = ‘0’ THEN
           q <= ‘0’;
        ELSIF clk‘event AND clk= ‘1’ THEN
           q <= d;
        END IF;
    END PROCESS;
END behav;
```



A Few Coding Tips

- Modularity
  - Write one module per file and name the file the same as the module.
  - Break larger designs into modules on meaningful boundaries.
  - Always use formal port mapping of sub-modules (named instantiation)
  - Be careful to create correct sensitivity lists.
- Is your code for synthesis or simulation only?
  - VHDL provides many ways of doing the same thing
  - Some modules such as test benches, are never intended to be synthesized into actual hardware –in these types of modules, feel free to use the full monster that is VHDL

Synthesis Considerations

- For modules you intend on synthesizing, you should apply a coding style to realize:
  - Efficiency
  - Predictability
  - Synthesizability
- Efficiency is important, for both performance and cost concerns
  - Use multiplication, division, and modulus operators sparingly
  - Use vectors / arrays to create “memories” only if you are sure the synthesis tool does it properly
  - CASE may be better than large IF-ELSE trees
  - Keep your mind focused on what hardware you are implying by your description

Synthesis Considerations (cont.)

- Predictability is important
  - Predictability of what hardware may be created by what you describe
  - Predictability of hardware behavior
    - Hardware behavior will match your description.
    - No dependency on event ordering in code.
  - Use supplied templates!
- Realize that synthesis is automated mapping of your functionality based on various algorithms
  - Invariably limited capability
  - Garbage in, garbage (or errors) out
- Carefully read synthesis warnings and errors to identify problems you may be unaware of

Questions?

University of BRISTOL