Digital Design, Group Project
EENG 28010

Course Overview - Introduce Assignment 2

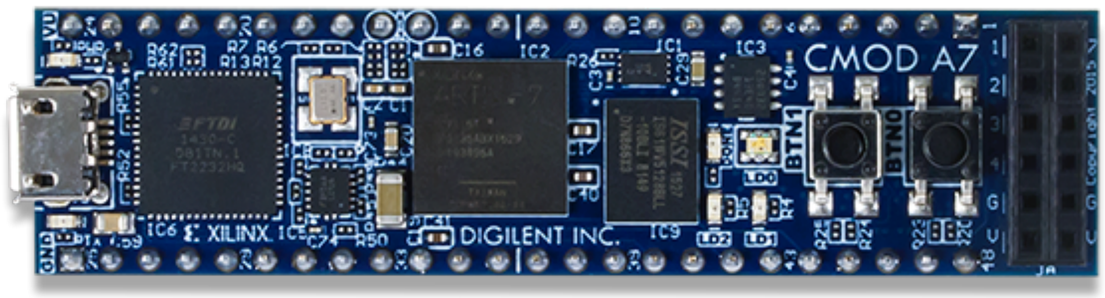
Dr Faezeh Arab Hassani and Dr Shuangyi Yan

Dept of Electrical and Electronic Engineering

University of BRISTOL

Unit Goals

- To give you 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

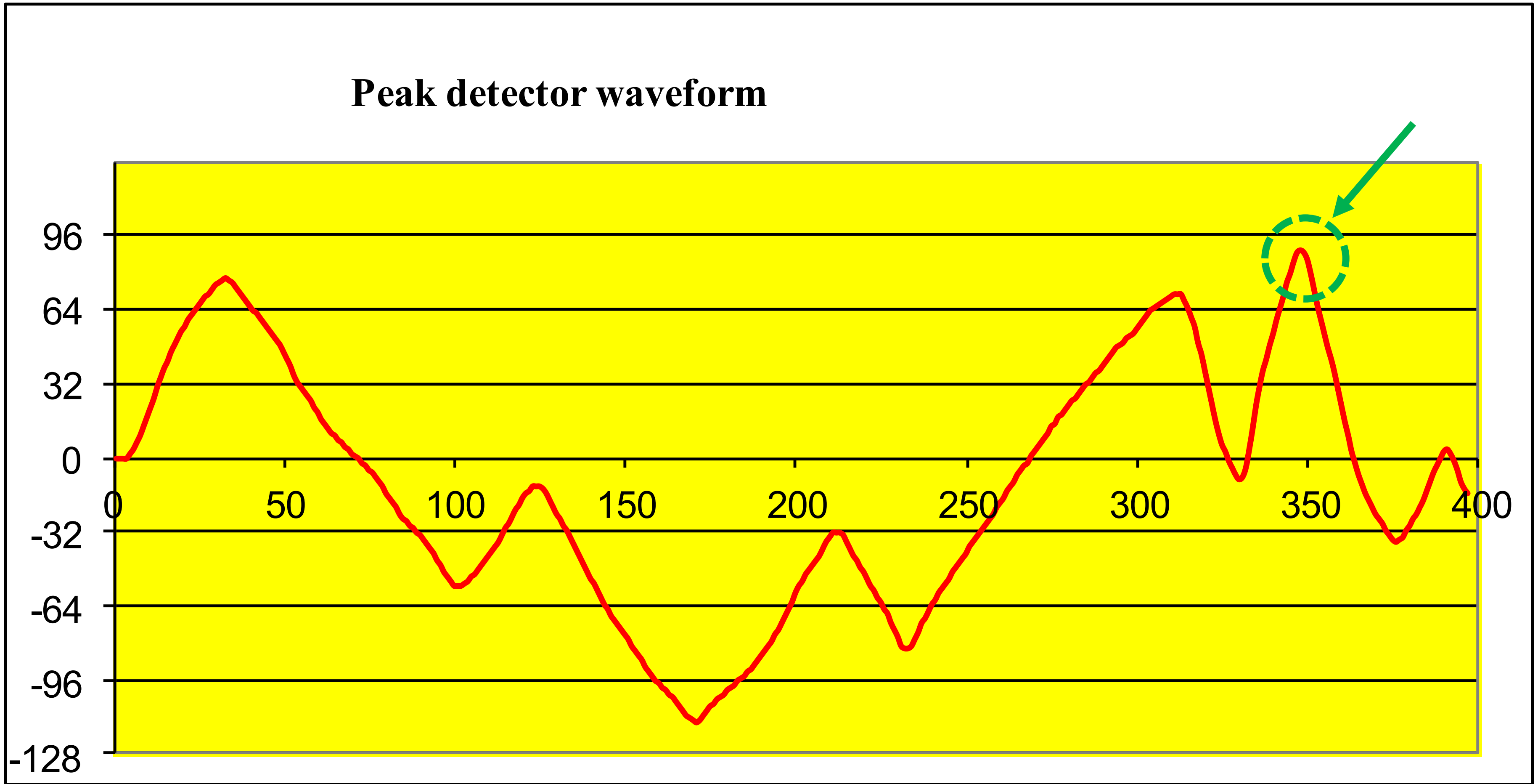


Outline

- Introduction of Assignment 2
- Writing Synthesisable Code

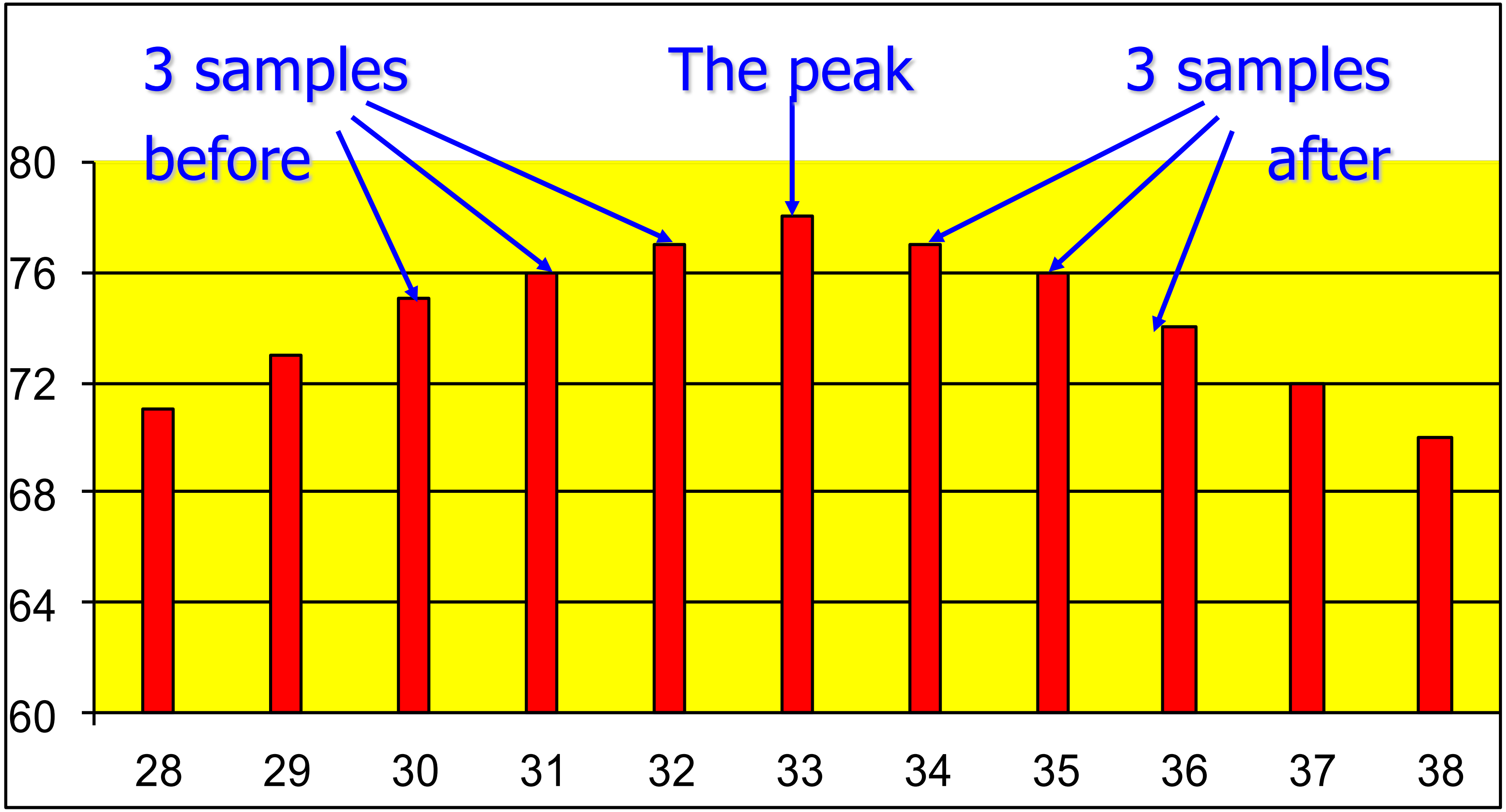
Assignment 2: Peak Detector

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.



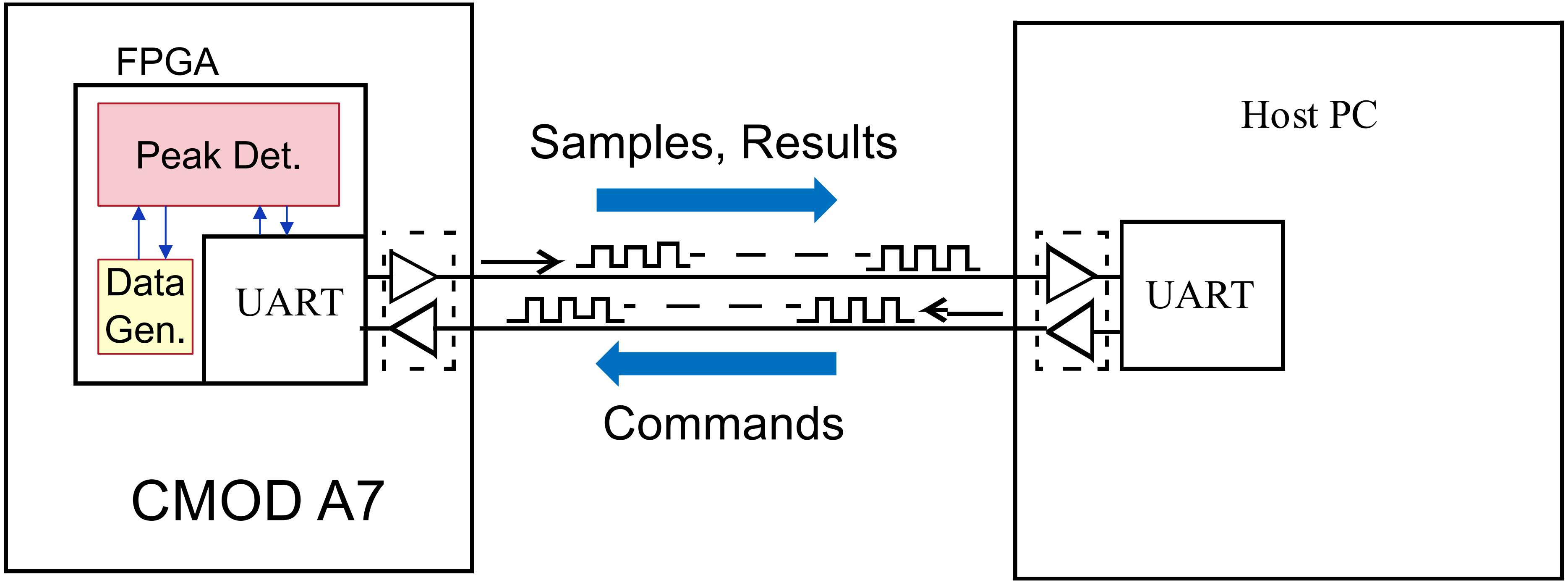
Capturing Samples

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



Peak Detector – Serial Communication with PC via UART

- User commands accepted from PC and outputs printed to PC
- Uses the RS232 serial protocol
- Serial communication Implemented by Universal Asynchronous Receiver Transmitter (UART)



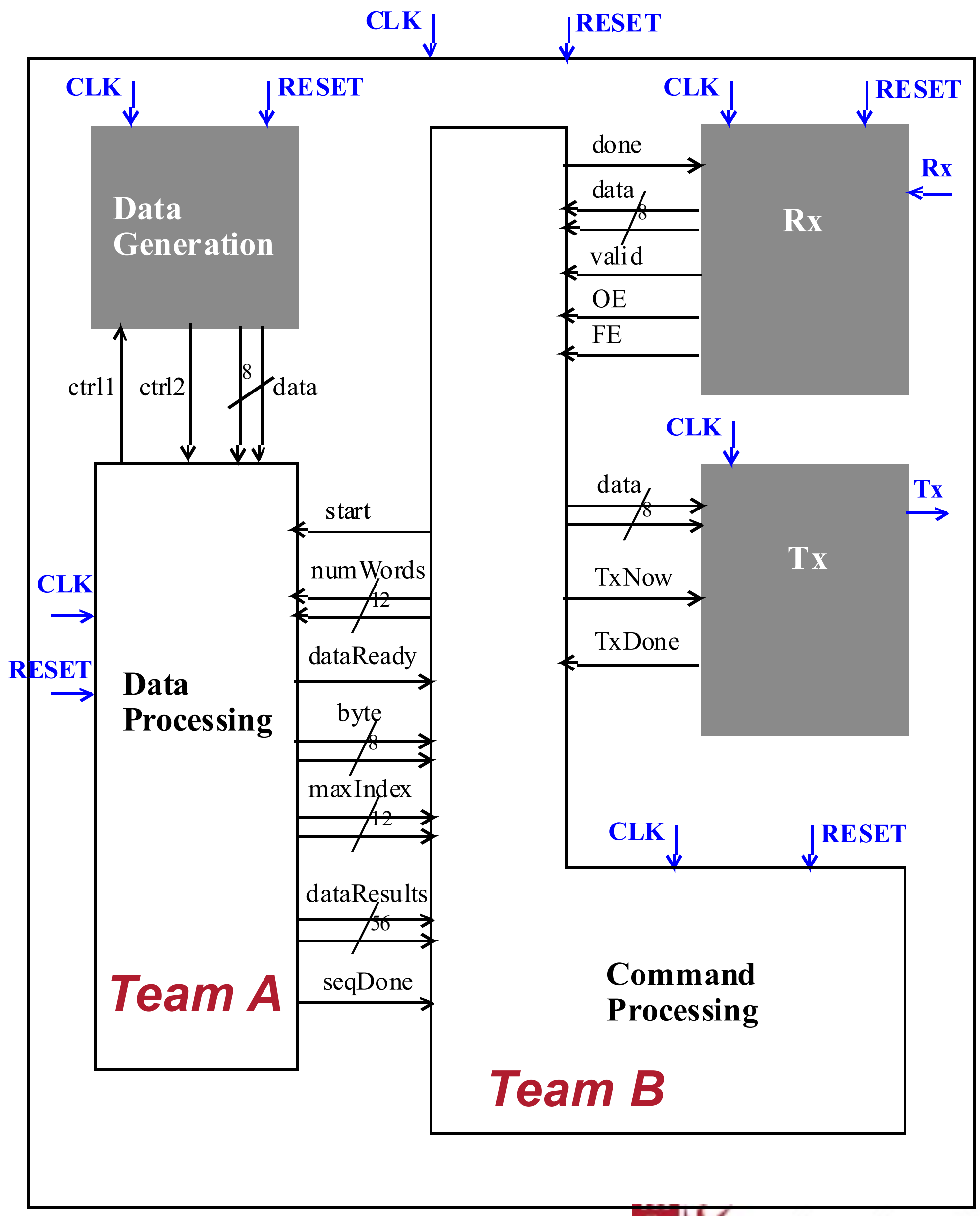
University of BRISTOL

Main Requirements

- Receive commands through UART from a computer terminal
  - UART demo is provided
  - Ascii code
- Request a mount of data based on commands
- Analysis peak data and capture the adjacent samples
- Send the data and results back to the computer terminal through UART

Peak Detector – Work Plan

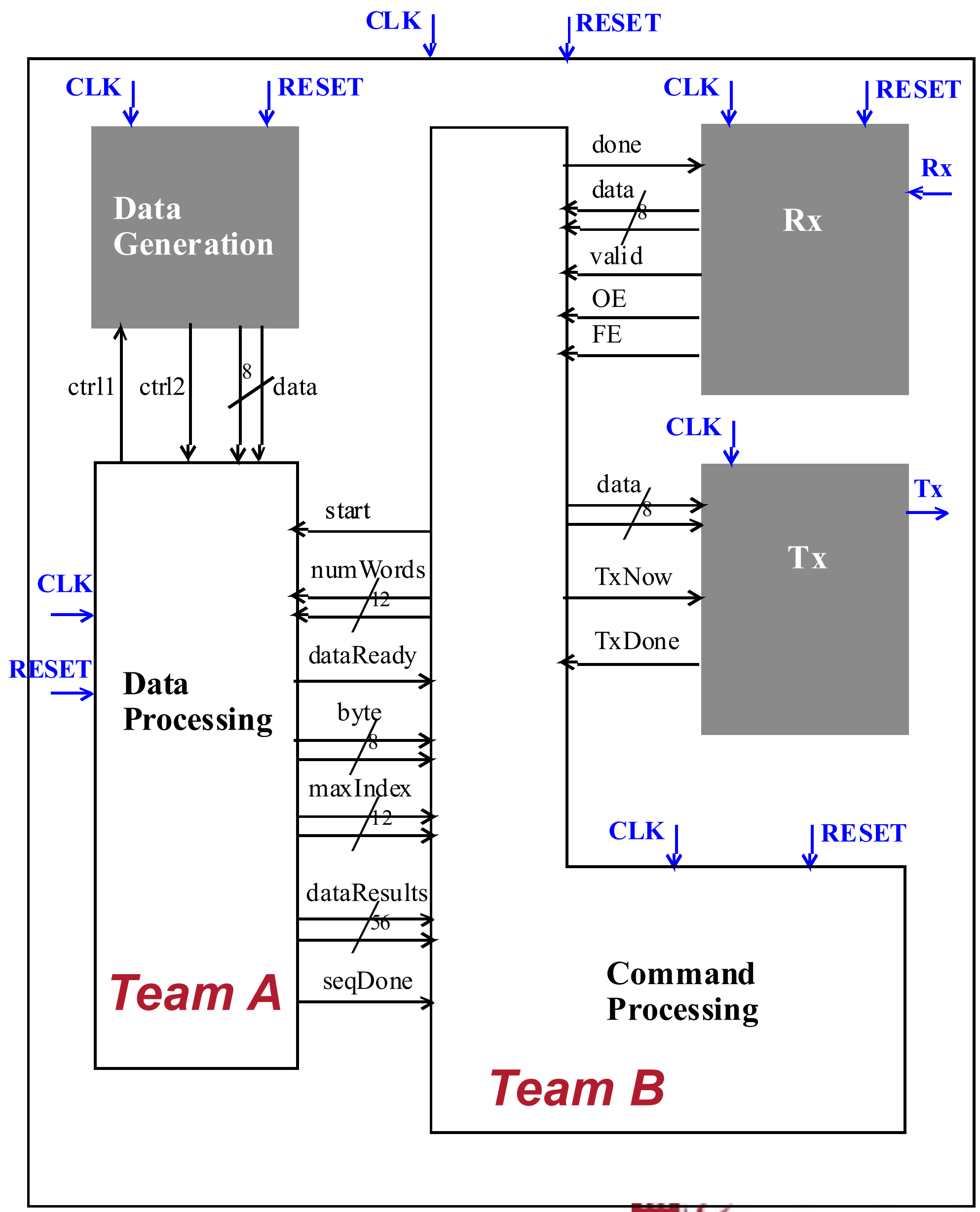
- *Data Generator*, *Rx* and *Tx* are supplied
- Divide group into two teams
- ***Team A*** works on the *Data Processing* block (two students)
- ***Team B*** works on the *Command Processing* block (three students)
- Clearly defined interface and blackbox implementations of Data Processor / Command Processor allow independent development



University of BRISTOL

Function requirements – Simulation (ModelSim)

- Data Processor
  - All required functions
- Command Processor
  - ***Successful response on ”ANNN" or "aNNN”***
  - No need to implement the ”L” and “P” commands
- Integration
  - Just “ANNN" or "aNNN”



Hardware and Vivado-based implementation

- Optional, not contributing to the final marks
- Require a team member to borrow and return after the submission
- Web-based guidance is available.
- Check the different code achieve for implementation.

Suggested Approach

- Start from the FSM diagram and then turn it into Arithmetic State Machine (ASM) chart
- Separation of the combinatorial logic with state machine. Get rid of all the latches we are encountered while synthesizing the code.
- 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.

Odd group number: unsigned implementation.

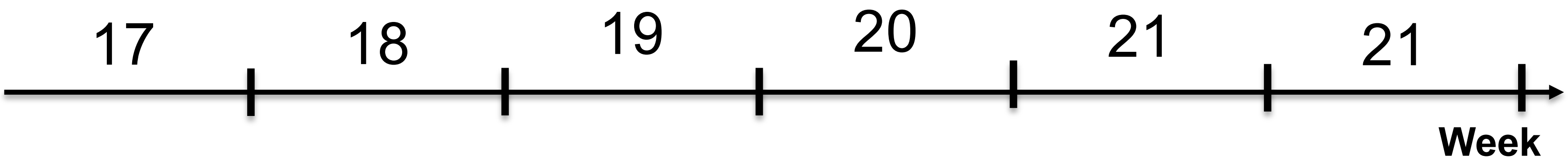
Even group number: signed implementation.

Week 17 - 19

- 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
    - Trace key signals
  - Watch the two-phase signaling protocol
- Work on FSM charts
  - Provide feedbacks on week 18 and week 19

Peak Detector – Work Plan *(cont.)*

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



Week 17 – 19: Complete design of data processor and command processor

Prepare drafts for your individual chapter.

Week 20: Integration and Testing individual components

Week 21: System integration, simulation and report

Report Contents

- Around 10 pages
- Chapter 1: Description of the group composition and task division between group members.
  - Each student needs to summarize your 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) (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) (Team B)
- Chapter 4: Description of the simulation results and test of your final implementation.

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

Deadline and Marking

Assignment 2: Design of a Peak Detector	Final Design	Group Assignment	Submit code and group report	13:00 Thursday 28 April	60%
	Peer Assessment	Individual	Include in your report	13:00 Thursday 28 April	5%

Assignment 2: (In total: 65%, including Design and Report: 60%, Peer Assessment: 5%)

- Report: 30% for all members
- Individual marking of Data processing and Command processing (50%)
  - Team A and B get individual marks
- Integrating marking (20%) for all members

Pass/fail unit: decide based on your performance in three assignments (over 40% for the three assignments)

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 eg: 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 exchg
- 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

University of BRISTOL

Signed vs. Unsigned

Odd group number: unsigned implementation.

Even group 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

Unsigned and Signed Types

Source: ref [3]

- 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 [3]

- 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;

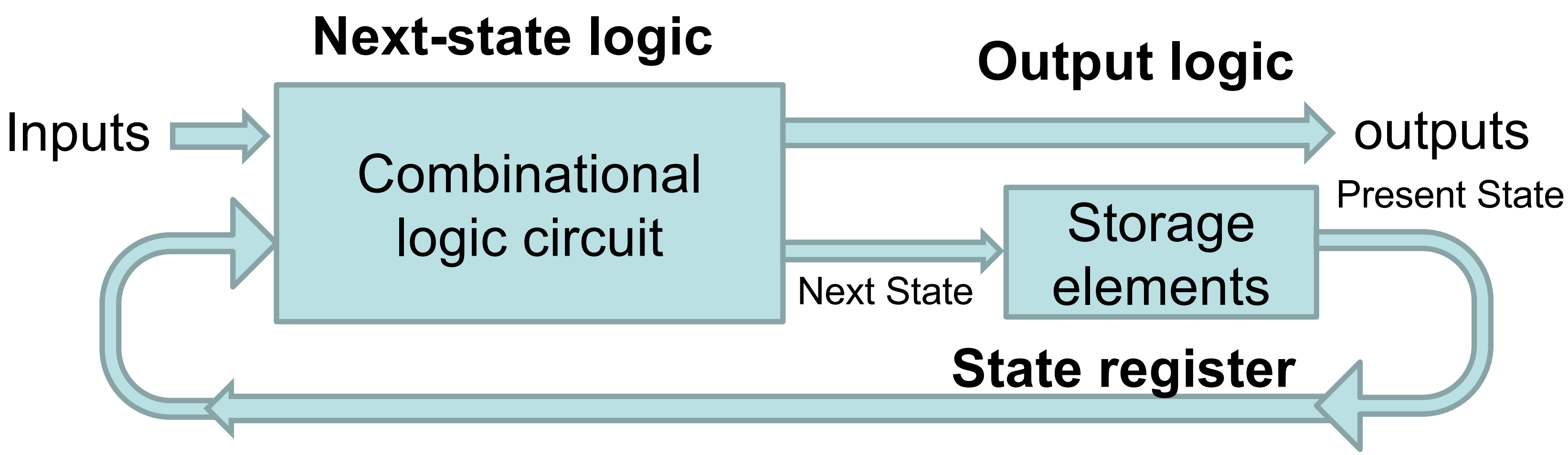
Outline

- Assignment 2
- Writing Synthesisable Code
  - Generating Hardware from Code Constructs

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



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
```

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 (eg: 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!

Mistakenly Inferring Latches with CASE

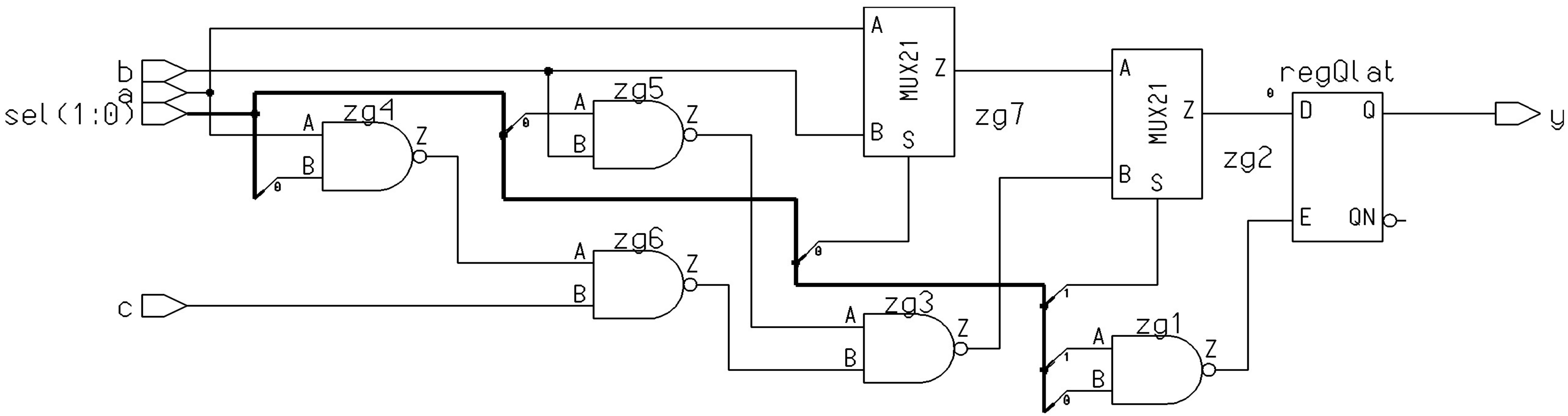
Latch Creation via Incomplete Assignment in Conditional Assignment:

- 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

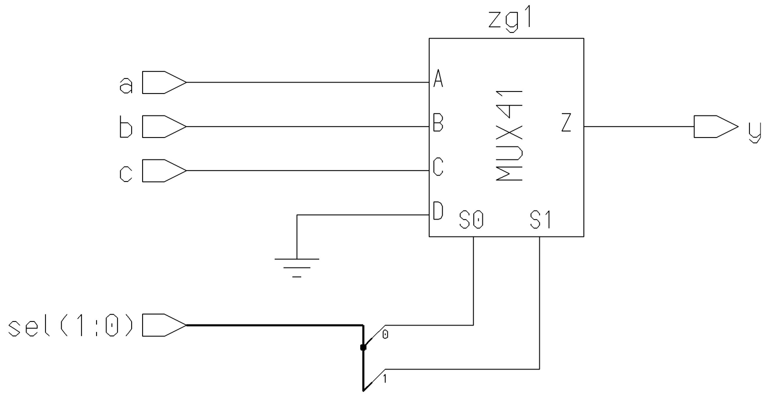
- Assigning values of “don’t care” (‘-’) in these cases can avoid unwanted latch inference

```
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;
```



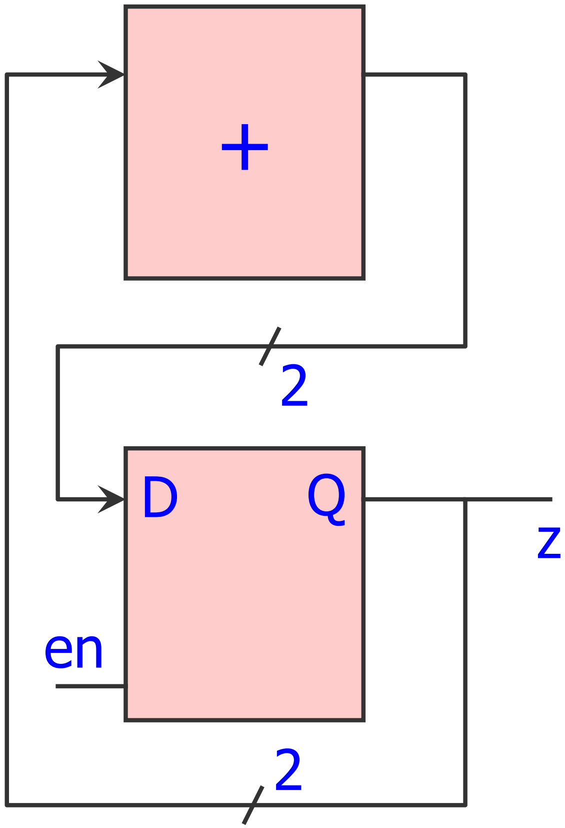
Latches are generated mostly by combinational logic!

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;
```



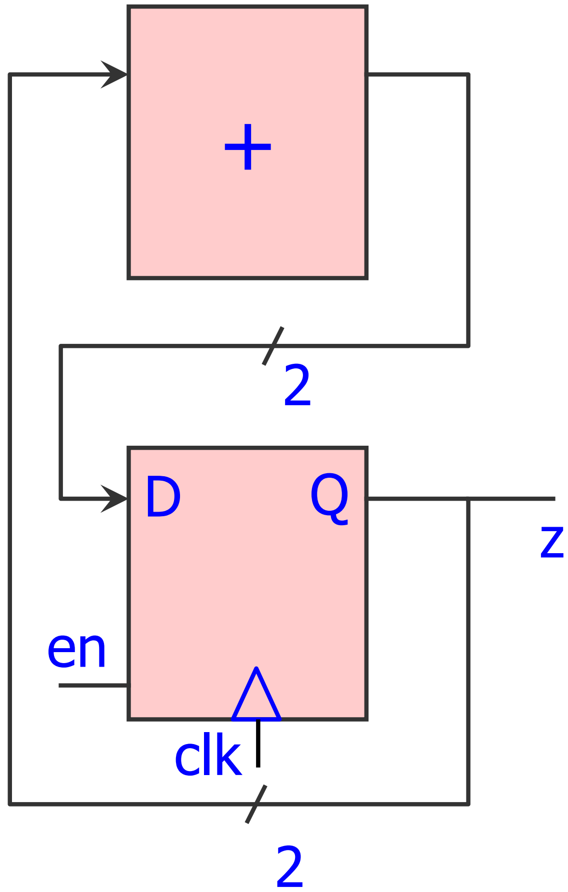
Do not use latches!

University of BRISTOL

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;
```



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

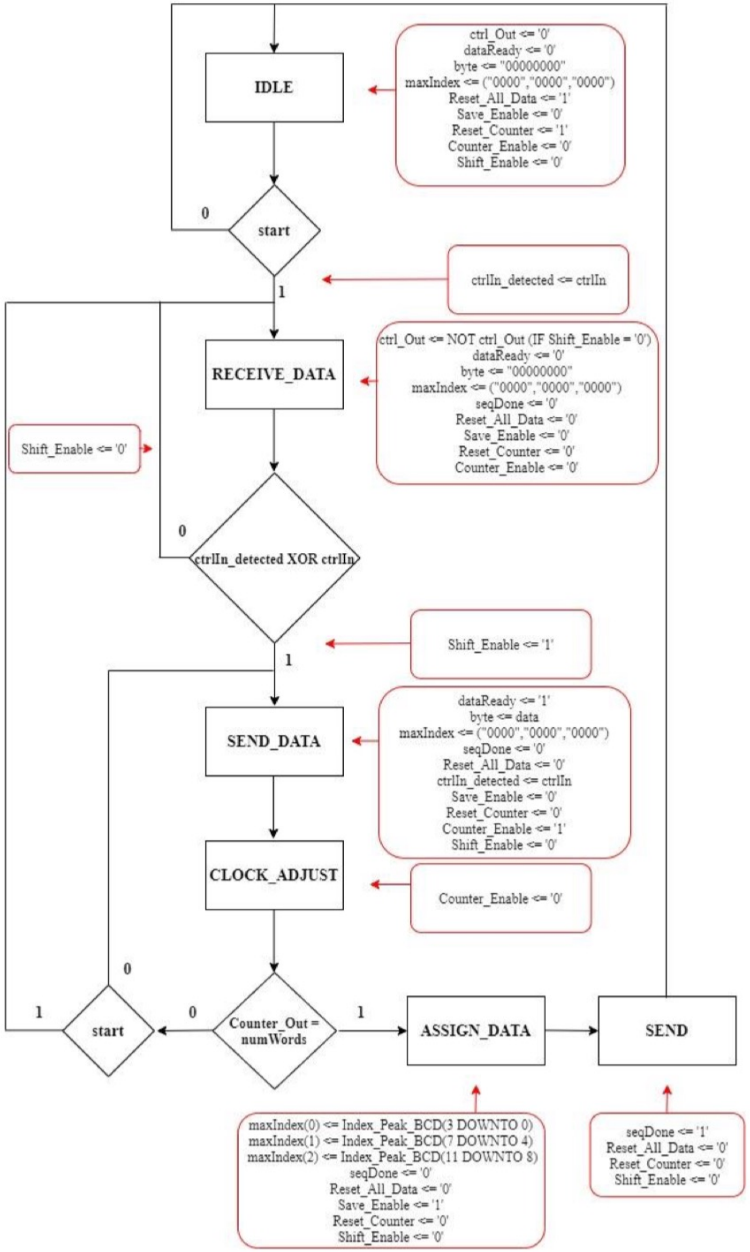
Synthesis Considerations

- 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

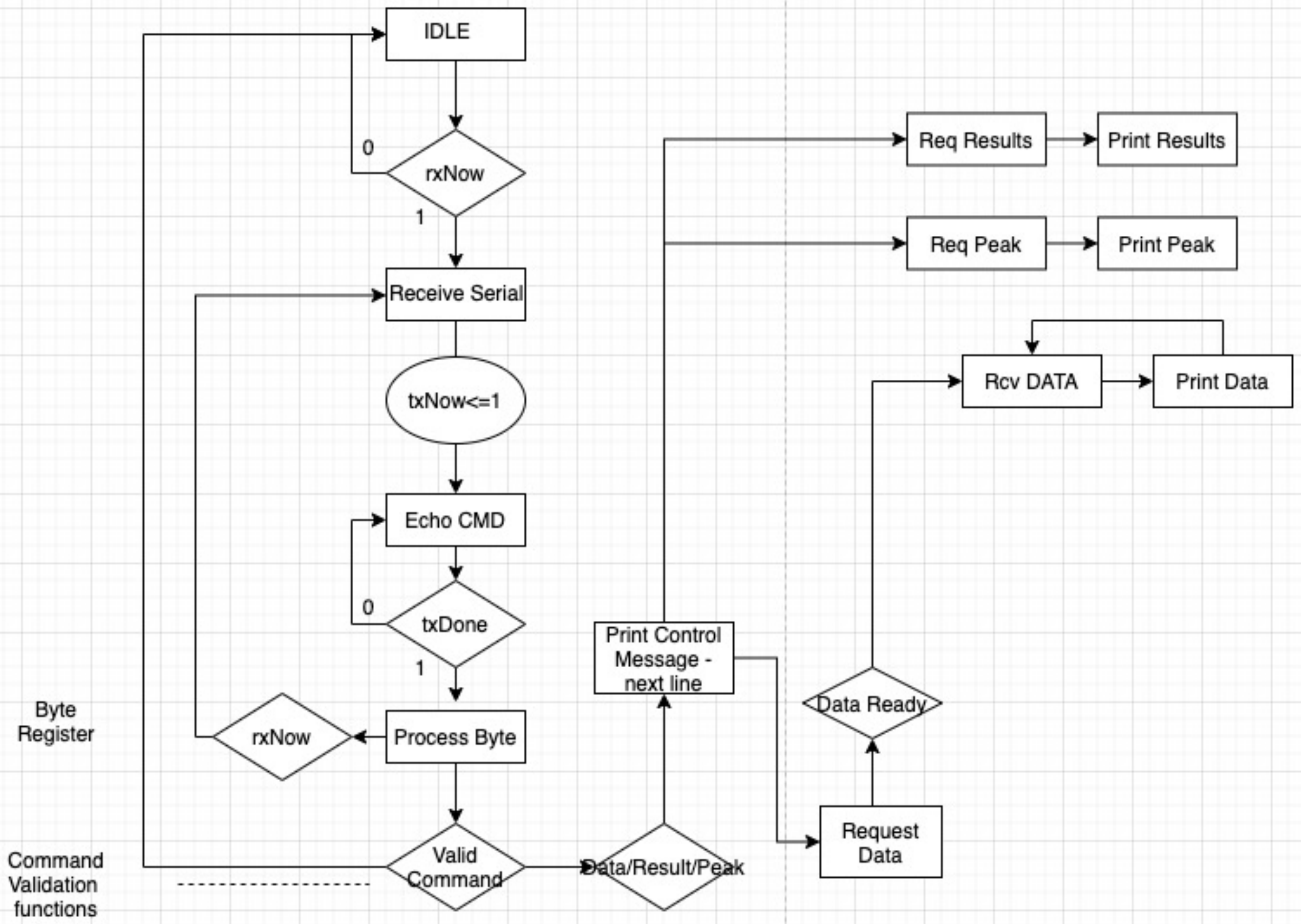
Synthesis Considerations (cont.)

- 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

Data Processor



Command Processor



Synchronous Functions:

- State register
- nibble counter
- Results counter
- Bit shifter
- BCD number register

Unsynchronous Functions:

- Command Validation
- Ascii code convertion
- Tx-data for printing/echo
- Bit shifter
- BCD number register