# ELEC2202
# Digital Systems and Communications

Introduction to design experiment D1
CPLD implementation of a SHIFT-ADD multiplier

University of Southampton
School of Electronics and Computer Science

## D1

- Mark contributes 15% to the module's assessment
- You work and are assessed individually
- Time in lab is one full day, i.e. both lab sessions, am and pm.
- Formal report required for assessment
  - log book is not marked
  - progress is not marked in lab
  - preparation is your responsibility and is not marked in lab
  - However:
- In lab you need to fill in the Design Completion Form, get it signed by your supervisor and attach to your formal report
  - The Design Completion Form will be taken into account during assessment, so preparation and progress demonstrated in lab are important!



## Preparation for D1

- You need to develop, test, and demonstrate a PLD implementation of a fairly complex sequential digital system, (a 4-bit sequential multiplier) using a MachXO2 CPLD.
- To assure success, before the lab:
  - develop your SystemVerilog code and testbenches for each module,
  - develop a testbench for the whole design
  - verify individual modules and whole design in Modelsim,
  - synthesise modules in Synplify.
- Devise a scheme to debounce the START button
  - Important as the internal clock of the CPLD is high, approx 25MHz.



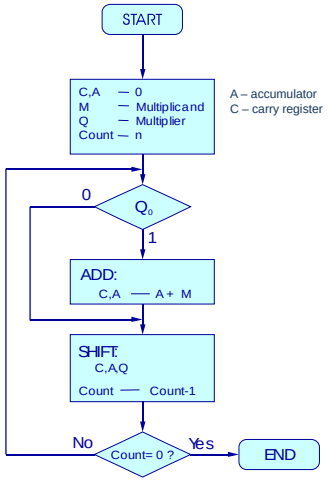
## Shift-and-add multiplication example

Multiply 7 x 5 in unsigned binary using the ADD & SHIFT algorithm:

```
   111  M muliplicand
x  101  Q multiplier
   111  (ADD 7,  Q[0]= 1)
  0000  (don't ADD, Q[1] = 0)
 111    (ADD 7 shifted by 2 bits, Q[2]=1)
100011  (result = 35)
```

## Shift-and-add algorithm for n bits

```
      111  M
    x 101  Q
      111000  A after ADD 1
      011100  A after SHIFT 1
      001110  A after SHIFT 2
carry 1000110  A after ADD 3
      100011  A after SHIFT 3
```





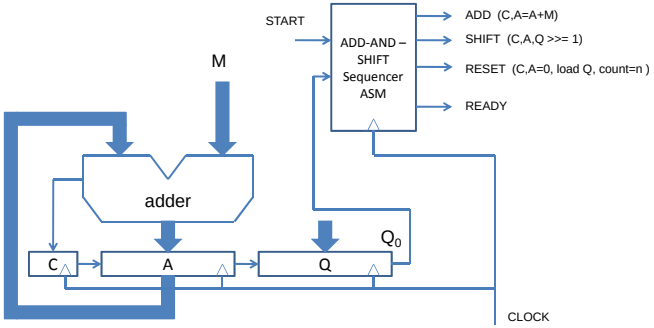

## Same example cycle by cycle

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| M= | | 1 | 1 | 1 | (M=7) | | | |
| | C | A | | | Q = 5 | | | |
| | 0 | 0 | 0 | 0 | 1 | 0 | 1 | Initial Values |
| | 0 | 1 | 1 | 1 | 1 | 0 | 1 | ADD: A := A+M |
| | 0 | 0 | 1 | 1 | 1 | 1 | 0 | SHIFT (cycle 1) |
| | 0 | 0 | 0 | 1 | 1 | 1 | 1 | SHIFT (cycle 2) |
| | 1 | 0 | 0 | 0 | 1 | 1 | 1 | ADD: A:=A+M |
| | 0 | 1 | 0 | 0 | 0 | 1 | 1 | SHIFT (cycle 3) |
| A,Q= | 0 | **1** | **0** | **0** | **0** | **1** | **1** | **= 35** |

Note: here the double length accumulator AQ shares storage with multiplier Q



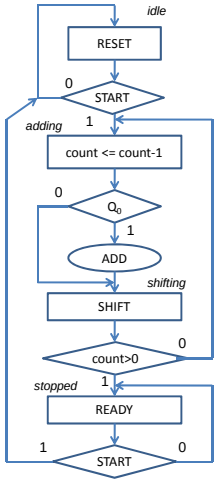
## Hardware implementation




C,A and Q can right shift by one bit

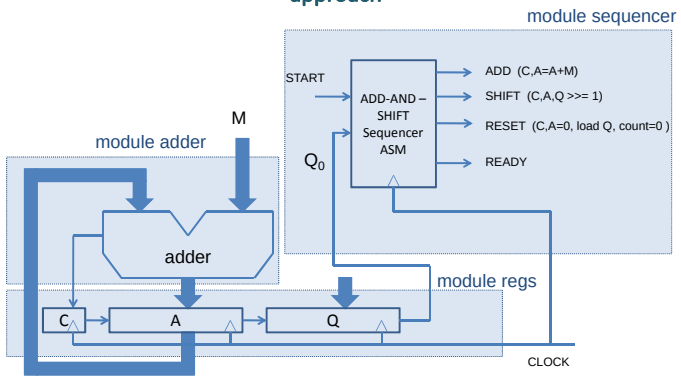


## Sequencer ASM chart




- 4 states: idle, adding, shifting, stopped
- modulo 4 cycle counter (2-bit counter) for 4-bit multiplier

## CPLD implementation of an n-bit multiplier using hierarchical approach



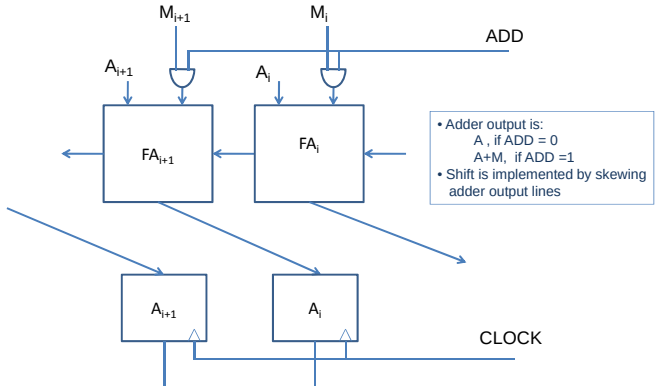



## Suggestions for further work

- Combine ADD and SHIFT in a single clock cycle
- Extend your multiplier design to more than 4 bits
  - MachXO will have plenty of room to spare
- Implement combinational array multiplier for comparison
  - Array multiplication is much faster than sequential shift-and-add
  - With only 4-bits an array multiplier might even be smaller than your sequential design!



## How to combine ADD and SHIFT in single clock cycle

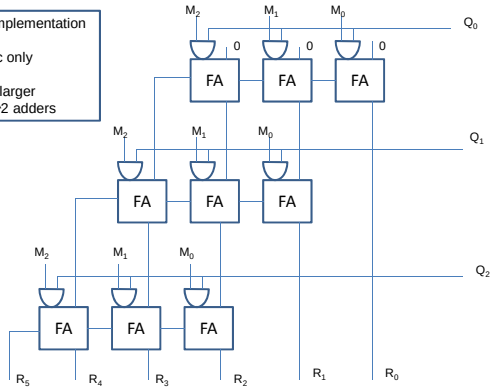



- Adder output is:
  A , if ADD = 0
  A+M, if ADD =1
- Shift is implemented by skewing adder output lines



## Array multiplier
## Implements SHIFT and ADD in combinational logic

- Fastest multiplier implementation
- Combinational logic only
- Very expensive for larger numbers of bits: n^2 adders



## Coding in SystemVerilog

- SHIFT-AND-ADD sequencer is a standard state machine plus a cycle counter
- Adder is standard combinational logic
- C,A,Q registers are shift registers with parallel load
- Read D1 notes on the ELEC2014 notes page and PLD area on the labs pages for details specific to MachXO.



## C,A,Q registers in SystemVerilog

```
module Regs(input logic clock, reset, add, shift, C,
                input logic[3:0] Qin, Sum, output logic[7:0] AQ);

logic Creg; // MSB carry bit storage

always_ff @ (posedge clock)
if(reset) // clear C,A and load Q
begin
  Creg <= 0;
  AQ[7:4] <= 0;
  AQ[3:0]  <= Qin; // load multiplier into Q
else if (add) // store Sum in C,A
begin
    Creg <= C;
    AQ[7:4] <= Sum;
end
else if (shift) // shift A,Q
begin
    // use concatenation to implement shift: Carry into MSB of A
   // and AQ by one bit to right: AQ[0] <= AQ[1], AQ[1] <= AQ[2]; …
   // note also that Creg must be cleared
    {Creg,AQ} <= {1'b0,Creg,AQ[7:1]};
end

endmodule
```



## 4-bit adder in SystemVerilog

```
module Adder (input logic[3:0] A,M, output logic C, output logic [3:0] Sum);
// Synplify can synthesise A+M, but output carry needs to be derived

//  9-bit arithmetic addition allows to extract carry
assign  {C,Sum} =  {1'b0,A} + {1'b0,M};


endmodule
```



## General advice on coding and testing

- Write a separate testbench for each PLD
- Also write a testbench for whole design
- Synthesise each PLD and test separately using Digital Testbed
- Use a slow clock for testing but also demonstrate your design using a fast clock

## Encapsulating module for whole design

```
module multiplier // for MachXO implementation – read MachXO lab notes
    #(parameter n=4)
    (input logic nPB, // active low push button, e.g. S1
     input logic nreset,       output logic [7:0] leds);

  // Instantiate the MachXO internal 25MHz oscillator
  OSCC OSCC_INST (.OSC(fastclk));

  logic shift,reset, C, start,pulse,clk;
  logic [n-1:0] M,Q,sum;
  logic[2*n-1:0] AQ;
  //internal signals
 logic C, reset, shift, add;

 assign leds = ~AQ; // MachXO Mini Kit leds are active low

// module instances, note: cannot always use implicit mapping syntax here
adder A(.A(AQ[7:4], .M(M), .C(C), .Sum(Sum)); // must use named mapping here
regs R(.*); // named mapping equivalent: (.clock(clock), .reset(reset), … etc.)
sequencer S(.clock(clock), .Q0(AQ[0]), … etc. ) // use named mapping

endmodule
```



## Testbench for whole design – simulation only

```
module TestMultiplier;

logic clock, C, start, ready, reset, shift, add;
logic [2:0] M,Qin, Sum;
logic [5:0] AQ;

// multiplier instance - can use implicit mapping syntax
multiplier m(.*);

initial ... Specify your test waveforms here

….
endmodule
```

Notes:
1) Test whole design in Modelsim,
2) Also test each module separately in Modelsim and synthesise modules in Synplify Pro.