University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Science

EECS150, Spring 2010

Homework 4 Solutions: Testbenches, Transistors, and Implementation Technologies

1. DDCA 4.44

   (a) When a signal is declared as `reg` it means that its value is set in an always statement, i.e. it is on the left hand side of a <= or = in an `always` block. A signal declared as `reg` is not necessarily the output of a register or flip-flop, but can be inferred to be.

   (b) In the context of a simulation testbench, a signal declared as a `reg` can also be used on the left hand side of a = in an `initial` block. In short, there are only two places you can ever give a `reg` a value: inside of an `initial` block or inside an `always` block.

2. DDCA 4.34.

   There are different ways of implementing this state machine. In some cases, you will want to draw out a bubble-arc diagram and implement the FSM using the framework we have described in class. In other cases, when the FSM has predictable state transitions, you will want to implement using specialized hardware blocks. For example, an FSM that unconditionally transitions to a new unique state at each transition (i.e. the bubble-arc diagram looks like a chain) would probably be implemented with a counter and not the traditional "next state/current state" methodology.

   The FSM in 4.34 is such a circuit. We will implement this FSM with an adder/accumulator and a subtractor for the output logic. We will encode the current coin total in 5 cent denominations. This allows us to implement the adder/accumulator version in the same number of flip-flops as the conventional "next state/current state" implementation. See the next page for the HDL.

```
module CoinDispenser (
    input      Clock,
    input      Reset,
    input      Nickel,
    input      Dime,
    input      Quarter,
    output     Dispense,
    output     ReturnNickel,
    output     ReturnDime,
    output     ReturnTwoDimes
);

wire [3:0] Next, Remainder;
reg [3:0] Count;

// Output logic
assign Remainder =      Count - 4'd5;
assign Dispense =       ~Remainder[3];
assign ReturnNickel =   Count == 4'd6 || Count == 4'd8;
assign ReturnDime =     Count == 4'd7 || Count == 4'd8;
assign ReturnTwoDimes = Count == 4'd9;

// ``Next state'' logic
assign Next = (Nickel)  ? 4'd1 :
              (Dime)    ? 4'd2 :
              (Quarter) ? 4'd5 : 4'd0;

// ``Current state'' logic
always @(posedge Clock) begin
    if (Reset) Count <= 4'd0;
    else       Count <= Next + Count;
end

endmodule
```

When designing FSMs, look for when the FSM can be implemented using a standard block like a counter. It simplifies the implementation. As a rule of thumb, when the FSM bubble-arc diagram is very branchy, it is more difficult to do this. *Note:* Hardware firms love interview questions like this! They will give you a design and say to build the FSM in HDL or gates/registers. Pay very close attention to what they give you! If you take 30 minutes mapping out the state encoding and minimizing the states when the FSM could have been implemented in 5 minutes using an adder, they won't be impressed.

3. The FSM (with corrections indicated through a '*' and with comments indicated with //) is shown below. If a line of code is added to the design, it is marked with "ADDED."

```
  module TastyFSM (
      input reg   Clock,
      input reg   Reset,
*     input reg   a,
  // Inputs cannot be 'reg's
*     input reg   b,
  // Same issue as above
      output reg  Out1,
      output wire Out2
  );

  localparam  APPLES = 2'd0,
              ARE = 2'd1,
              TASTY = 2'd2;

  reg CurrentState, NextState;

  assign Out2 = (CurrentState == APPLES);

* assign Out1 = (NextState == ARE);
  // FSM outputs are derived from either the FSM's inputs or the
  // CurrentState, but not from the NextState logic.  Also, based
  // on how ``Out1'' has been defined as a reg, this assignment
  // cannot be made with an 'assign' statement.

  always@(posedge Clock) begin

*     if (Reset) CurrentState = NextState;
  // This block should be using only non-blocking statements
  // (<=).  Furthermore, the reset behavior shouldn't transition
  // the CurrentState to the NextState.

*     else CurrentState = APPLES;
  // It is in this block where the CurrentState should transition
  // to the NextState.  In any case, it should not transition to
  // a fixed state.

  end

  always@( * ) begin

*     NextState = CurrentState;
```

```
  // ADDDED line.  The NextState should be assigned the
  // CurrentState default value.

*     case(NextState)
  // Change ``NextState'' to ``CurrentState.''  This is a
  // combinational loop.  Instead, the output of the NextState
  // decision block should be determined by the ``CurrentState''

          APPLES : begin

*             if (a & !b) NextState <= ARE;
  // NextState logic should be assigned using blocking (=)
  // statements.

              else if (!a & b) NextState = TASTY;
          end
          ARE : begin

*             NextState <= TASTY;
  // Change '=' to '<=' (same problem as above).

          end
          TASTY : begin

*              if (b) NextState <= ARE;
  // Change '=' to '<='.  Also, the transition based on the 'b'
  // signal should be to the 'Apples' state.

          end
      endcase
  end

  endmodule
```

4. DDCA 1.63

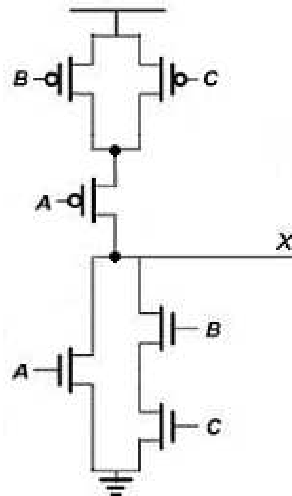
   Here is the truth table:

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

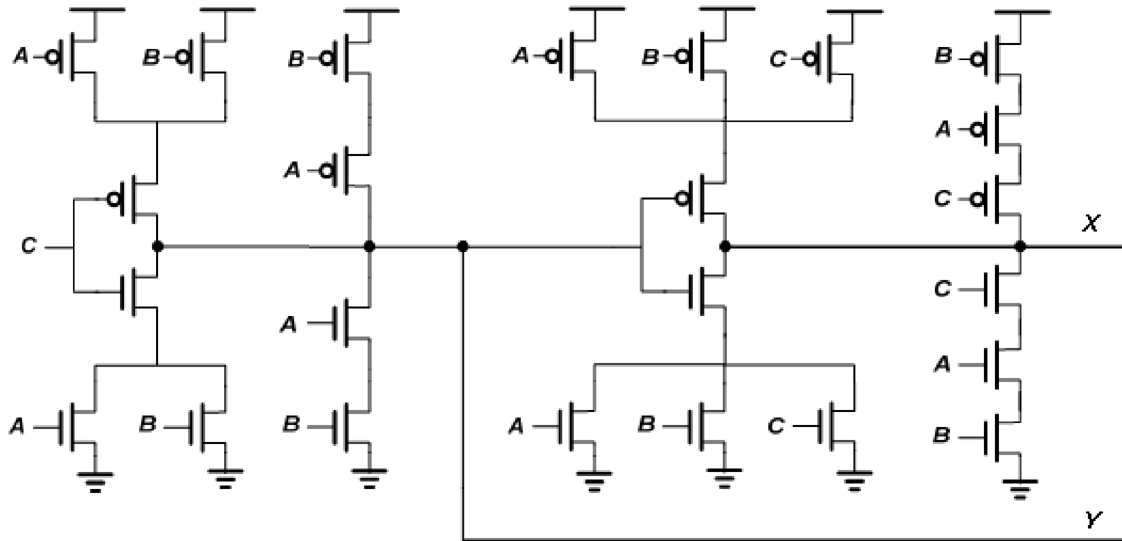
5. (a) The easiest way to do this problem is to just try out all the input combinations. See the solution truth table:

A	**B**	**C**	**X**
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

   (b) Yes. See here:

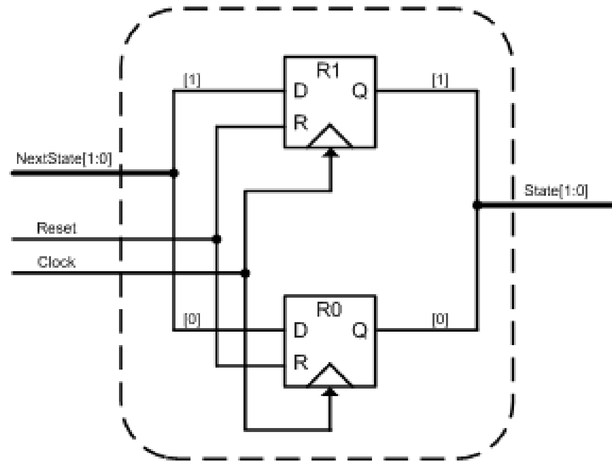


6. Again, the easiest way to do this problem is to just try out all the input combinations. This is circuit is actually just a full-adder, where **Y** and **X** are just the inverted C_{out} and S outputs, respectively. Shown here is the circuit and truth table:

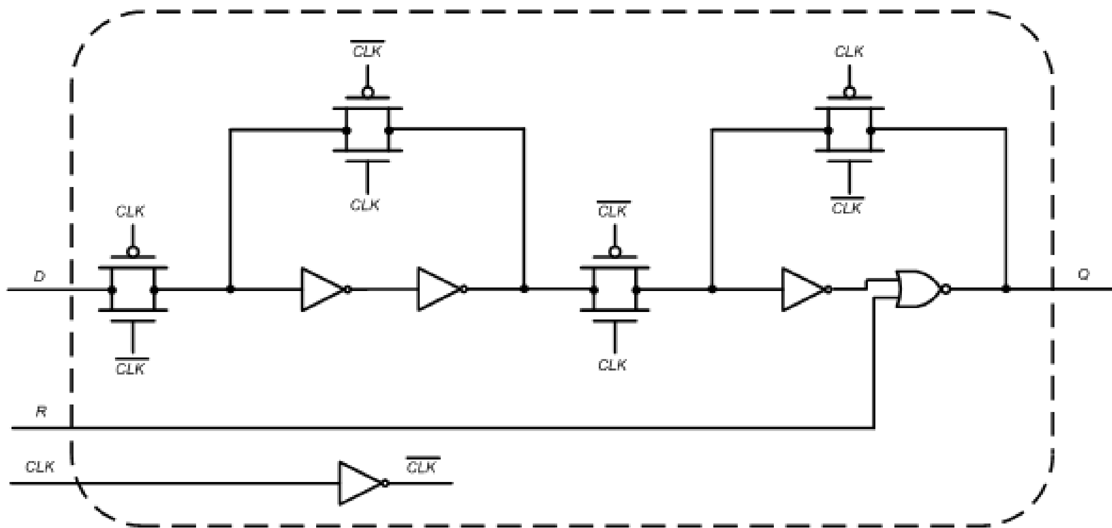


A	B	C	Y	X
0	0	0	1	1
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	0	0

7. We will begin thinking about how to build this FSM by first splitting it into three different parts: the State registers, the output logic, and the NextState logic. The State registers, shown here:

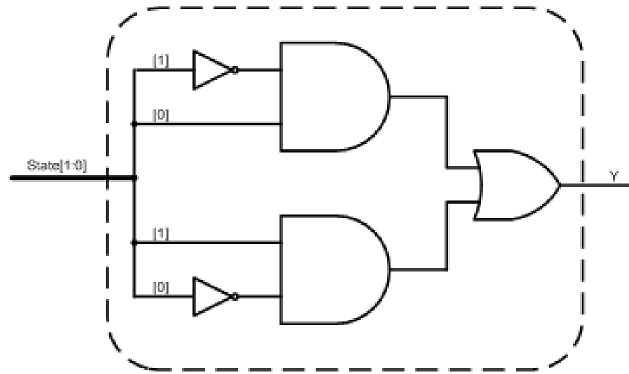


is easy to build - it is just 2 D-flip flops in parallel, each responsible for holding one bit of the state. Each flip flop is implemented here:



and note also that they have an **asynchronous reset**.

The output logic is relatively straightforward - just see whether `State` equals the encoding for `S1` or `S2`, namely whether `State == 2'b01` or `State == 2'b10`. This is implemented here:

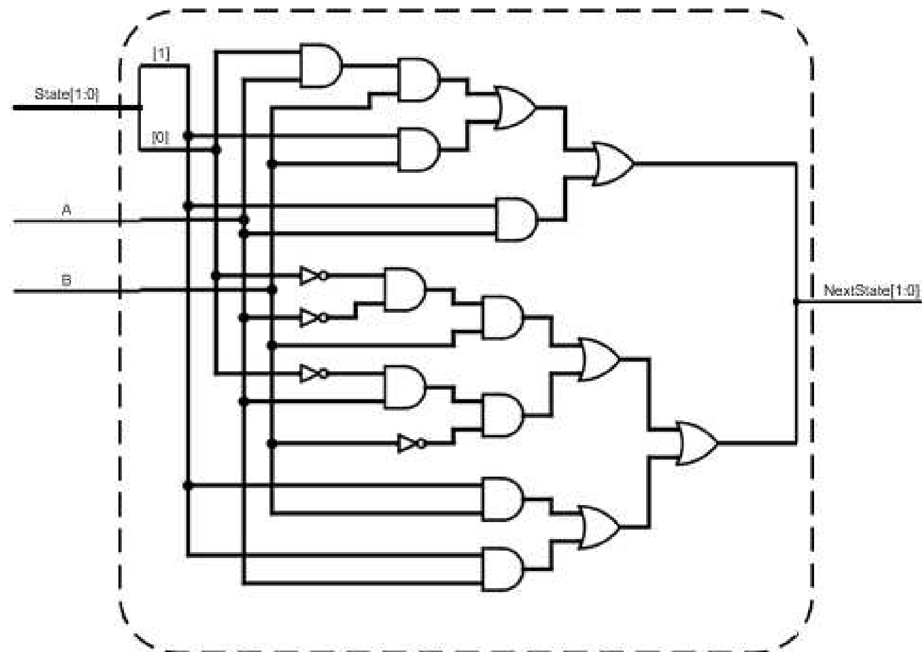


The next state logic is a bit tricky. It is probably easiest to begin with a truth table, such as the one in the below table:

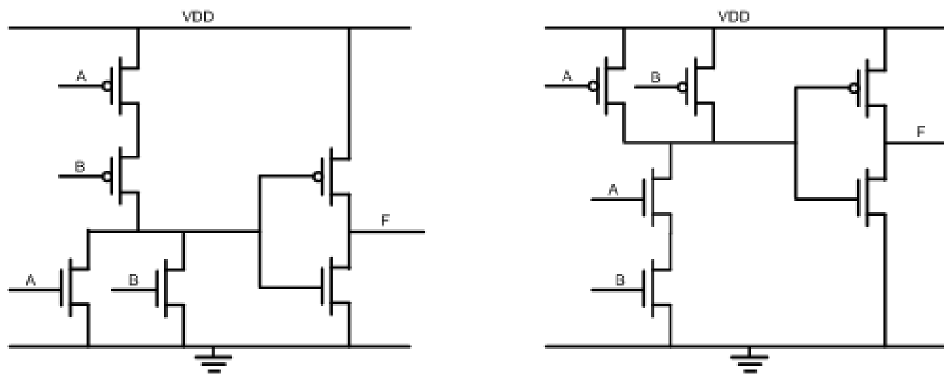
ST_1	ST_0	**A**	**B**	NS_1	NS_0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	0	1
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	0	0
0	1	1	0	0	0
0	1	1	1	1	0
1	0	0	0	0	0
1	0	0	1	1	1
1	0	1	0	1	1
1	0	1	1	1	1
1	1	0	0	0	0
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1

Note that the NextState function has four inputs (`State[1:0]`, `A`, and `B`) and two outputs (`NextState[1:0]`).

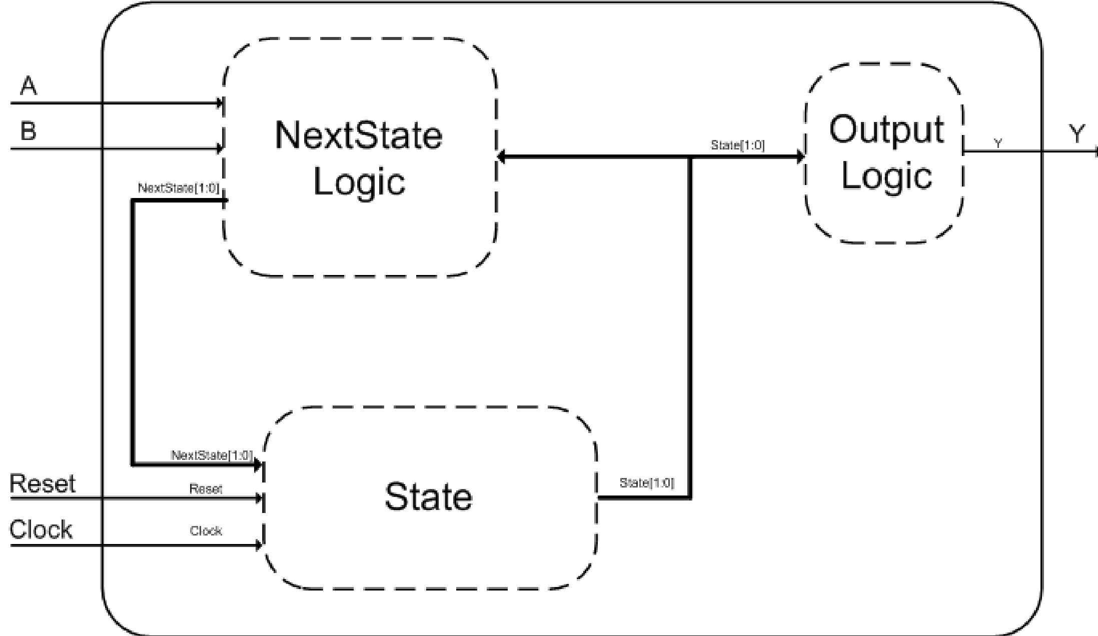
With the truth table in hand, we can now use *Logisim* to help us quickly draw the equivalent circuit, shown here:



We can implement two-input OR and AND gates by just using NOR or NAND gates followed by an inverter, shown here:



How all three pieces connect together to form the FSM is shown below:



8. Each metric is listed below with the various implementation alternatives listed from *greatest* (first) to *least* (last).

9. Each metric is listed below with the various implementation alternatives listed from *greatest* (first) to *least* (last).

NRE cost: Full-custom, standard-cell, structured-ASIC, FPGA, GPP.

NRE refers to the "non-recurring expenditures" associated with an implementation. NRE cost then comes down primarily to how much work is involved in customizing or programming the device for an application. NRE costs also include any one-time costs associated with customizing the device, such as CAD tool costs or IC mask generation. Our assumption (in the following paragraphs) is that you start by choosing one of these implementation technologies and then proceed in a way that will minimize your NRE costs. So for instance, if you decide to use an FPGA in your application, you will not first design the actual FPGA fabric then proceed to configure it. Instead, you will choose an existing FPGA part from one of the FPGA vendors and your work will only be to "program" it (configure it in this case).

Full-custom has the highest NRE cost, because designing custom IC circuits and masks is very labor intensive and involves the purchase of expensive design tools. Standard-cell is better in this regard, because it is based on libraries of predesigned and layed-out blocks (such as 2-input *nand* gates) and more of the layout details is left to the CAD tools and is therefore less labor intensive. Both full-custom and standard-cell, however, require a full mask set to be produced for every design—an expense on the order of $1M (in 2010). Structured-ASICs are based on prefabricated standard chips with customization occurring only a few interconnection layers. Therefore only a fraction of the labor and mask costs

are incurred. FPGAs have a lower cost than all of the above, because no chip fabrication is involved. Furthermore, FPGA design tools do a good job of automatically mapping HDL based designs. These two factors combine to result in relatively low NRE costs. The story is similar with GPPs and software, but software is generally thought of as being easier to write and optimize than hardware (it is typically more sequential in nature, which is easier for humans to reason about). Furthermore, the process of mapping software to a processor generally goes quicker than mapping an HDL to an FPGA, as FPGA designers typically will need to iterate to achieve the designed timing behavior.

Per-unit cost: FPGA, structured-ASIC, standard-cell, full-custom

Except for the GPP case, per-unit cost is straightforward. Generally, chip cost is a strong function of die area and therefore per-unit cost is directly proportional to die area. Also, similarly to with NRE costs above, here we will assume that per-unit cost is the only important metric and therefore the designer chooses to minimize area. Therefore a full-custom design, where the designer can devise a microarchitecture and transistor circuit implementation optimized for a particular function, will have the lowest per-unit cost. Next lowest would be standard-cell; it can be customized by choosing the optimal set of cells and microarchitecture, but the designer has no control of the contents of the cells. Similarly, with structure-ASIC, where the designer has even less control over the area used by the design. With FPGAs, the device can be customized for a particular function, but the resulting circuit will be larger in area than the above devices. This is because FPGAs include extra circuitry to provide the high degree of flexibility in the wiring and logic blocks.

Comparing the GPP is more complex, because the answer really depends on the desired functionality and one's view of what a GPP actually is. A simple function, for instance stripping the payload out of a Ethernet packet, can easily be implemented with a cheap hardware circuit. A software implementation of this function is also simple, but the implementation of a processor capable of executing such a program is in itself relatively complex (compared to an implementation of the function directly in hardware). However, if our desired application of the device needs the flexibility and reprogrammability that a processor provides, then a processor might be the inevitable solution. However, as we are learning, a processor can be implemented in any of the above implementation technologies, and therefore the resulting cost will be reflective of the underlying technology. Generally, it is better to think of a GPP as an *abstraction layer* and not as an implementation technology.

Time-to-market: Full-custom, standard-cell, structured-ASIC, FPGA, GPP.

Time-to-market is just that: how quickly can you get a working solution out to the market/consumer? In general, fabricating a custom chip takes the most time (full-custom/standard-cell/structured-ASIC). Also, for full-custom designs it takes the designer longer to fully designed and verified than a standard-cell designs, and similarly longer for standard-cell, than structured-ASIC, etc. FPGAs (as we have seen) just need an HDL design and the CAD tools (which take less time than fabrication). GPPs are software-based solutions and take only software compilation time (which is faster than FPGA PAR time). Also, as we noted above, software is traditionally thought of as simpler to write relative to HDL (which means that it takes less time to develop).

Application performance: Full-custom, standard-cell, structured-ASIC, FPGA, GPP.

In general, the more customized the hardware to the problem, the better performance possible. As we move from left to right, we change from full custom solutions to gate-array like implementations (structured-ASIC, FPGA), and finally software solutions (GPP). Despite the different hardware implementations, they always have the potential to beat out software implementations in terms of performance.

Chip power consumption: FPGA, structured-ASIC, standard-cell, full-custom

Power consumption is a complex story and depends on a variety of factors from design microarchitecture, to logic organization, to transistor level circuits. Roughly speaking, implementation technologies that give us better control over the device will give us a better handle on controlling energy per operation. For instance, at one end of the spectrum, with full-custom we can use a variety of transistor level circuit tricks not possible with standard-cell. However, both choices will allow us to choose logic expressions that optimize for power consumption. At the other extreme, FPGAs give us control over the application microarchitecture, but little control over choice of logic implementation details, and no control at the transistor level. The GPP choice is difficult to compare, and comparing it follows the same line of argument as we used above for per-unit cost.

10. The best way to approach this problem is not to make up random numbers (we can tell), but to sit down in front of a computer and learn how to use the Digi-Key catalogue. This company is one of the leading US suppliers of various electronic and other components for large-scale manufacture. Unlike most other suppliers, however, you can also buy things in very small quantities from this store at nearly reasonable prices. Besides showing you where hardware comes from, this problem is meant to give you a real estimate of how expensive/inexpensive hardware can be. We encourage you to shuffle through the catalogue, find some of the parts used in the Calinx or your computer's motherboard. Understanding which design choices are expensive and which aren't is an important skill to an engineer.

    That said, if you took this problem seriously, you probably found it more challenging than expected: Digi-Key sells thousands upon thousands of similar components, yet does not allow sorting search results by price. If you do not know exactly what you are searching for, you may have a hard time finding the cheapest part. Also important is the fact that Digi-Key lists parts it does not actually have. Out-of stock parts may be difficult (or impossible) to purchase. When designing a system, you may want to make sure you can actually buy your components and that the parts you chose have not been obsolete for a year.

    The following are essential when looking for a component:

    Availability: is the item in stock? Being able to actually buy the item you want is helpful.

    Unit price: how much is one of these going to cost you?

    Minimum quantity: unless you're prepared to pay 1000x for a crate of these components, you should pay attention to this number. For this problem, the minimum order quantity should be 1.

    Note: it is not a problem if you have no found the absolute cheapest part. We're looking for a sign of effort, and will accept a broad range of prices found.

(a) What is the cost of each flip flop?

Below we show the flip-flop that we found (Link):

74ACT74SC-ND	74ACT74SC	IC F/F DL D-TYPE POS EDGE 14SOIC	74ACT	Fairchild Semiconductor	Surface Mount	14-SOIC	Set and Reset	1	2 - Dual	24mA, 24mA	Differential	Positive Edge	D-Type	Tube	-40°C ~ 85°C	7.5ns	210MHz	4.5 V ~ 5.5 V	1,858	1	0.42000	

This is one of the cheapest D-flip flops that Digi-Key has in stock, and allows you to buy individually. Some characteristics: Fairchild Semiconductor 74ACT74SC dual posedge-triggered D flip-flop. \$0.42 each. \$520 for 5000 of them. The word "dual" means there are 2 D-flip flops in each chip.

(b) The Virtex-5 LX110T FPGA used in lab has approximately 70,000 flip flops. Look up the cost for this part (you are searching for a XC5VLX110T part in a FF1136 package). How much would it cost to build the same number of flip flops as an LX110T using the cheapest dual D flip flop part that you found?

To get 70,000 flip flops, you would need to purchase 35,000 of the 74ACT74SC's. It is easy enough to realize that if you were to buy the 35,000 dual D flip-flops one by one, it would set you back no less than \$14,700.00. Fortunately, due to volume pricing, the actual cost is much lower, at about \$3640. You may have noticed, however, that the store currently stocks less than 2,000 of these dual flip flops.

Note: this is only about twice the cost of your FPGA (\$1750 on Digi-key, but they have been out of stock of a long time). Don't get any ideas about FPGAs being outrageously overpriced, as it is important to consider how much space 70,000 discrete flip-flops would take, their performance characteristics, not to mention the vast amounts of other resources the FPGA provides.

(c) What is the cheapest Xilinx FPGA (not CPLD!) device that you can find on DigiKey? How many flip flops does it have? To make sure the part is not obsolete, please find one that is still stocked! Hint: Consider looking up the "Spartan" family of FPGA devices.

The cheapest Xilinx FPGA we found on Digi-Key is the Spartan-II XC2S15 speed grade 5, commercial temperature range, VQFP100 (VQG100) package. The cost of this part is \$7.10. Link here. Package Drawing here.

We would like to emphasize that this is a very, very small FPGA. This chip has 60 I/O pins and only 96 CLBs (and as few registers), whereas the XC5VLX110T has 640 I/O and 8640 CLBs.

We hope this problem has shown you that the resource you have at your disposal (the ML505 board with the XC5VLX110T FPGA) is an incredibly versatile platform. Using this board you can build anything from Ethernet switches to GPUs and full-featured CPUs to oscilloscopes. We encourage you to be courageous and try new things with the ML505 after developing the necessary skills.

(d) Suppose you need a really cheap fix to a digital problem on a board that you are designing. For a very simple design with just a few flip flops, discrete D flip flop packages will be cheaper. What is the crossover point for which a Xilinx part will be cheaper?

To find the crossover point, find how many D flip-flops will cost as much as the cheap Xilinx FPGA. At \$0.42 per dual flip-flop, only 20 parts are needed to exceed the very low \$7.10. This amounts to a total of 40 D flip-flops (two per package).