Lecture 10:  Programmable Logic

• We’ve spent the past couple of lectures going over some of the applications of digital logic
• And we can easily think of more useful things to do
  – like having a 7-segment LED display all the numbers between 0 and 9, not just 0 and 3
• But the more interesting the task, the more gates (in general) it will take to build the required circuit
  – we’ll run out of space on our breadboards before too long!
• An integrated circuit (with a large number of gates built on a single silicon chip) provides a way around the space limitations
  – The CPU in a PC is a (highly complex) example of such a circuit
• But there are drawbacks:
  – takes a lot of engineering effort to design an IC
  – expensive to set up for producing one
• And what happens if we want to change the circuit design?
  – that means designing and producing a whole new IC!
• For small and medium-sized projects, there’s a better idea: programmable logic
• To understand how programmable logic works, we return to the concept of memory in electronics
  – we’ve already seen that an S-R latch (and its variants, such as J-K, D, and T) can have memory states
  – i.e., the output depends upon the history of the circuit
  – The RAM is your computer is based on these
  – This memory is volatile – we lose all the information the instant the power is removed from the circuit
• Computers need some memory that can survive loss of power
  – i.e., to store the instructions for booting the computer!
• How do we make memory that behaves this way?
  – consider the 7-segment decoder you built
  – for a given two bits of input, the 7 output bits have a definite value (even after the power is turned off and back on again)
  – we can think of our two input bits as a memory address, and the 7 output bits as the value stored at that address
  – clearly, we could add more logic to handle larger numbers of address bits (and thus have more possible outputs stored in the memory)
• The point is, a fixed set of logic gates can act as a permanent memory
• In the simplest form of such a device, the gates can’t be changed once the IC is built
  – so we have read-only memory (ROM)
PROM

• What if you need ROM for a small project?
  – so that it’s not cost-effective to develop an IC to store the memory you want
• You can get a PROM (Programmable ROM), which is a ROM that initially has all the output bits set to 1, no matter what the input is
• Not so useful, except that the output values are controlled by fuses
• By sending a large current, the fuses can be broken, which sets some of the outputs to 0
  – this is called “burning” the PROM
• By burning the right set of fuses, you can make the PROM store whatever values you want
  – note that you can only do this once (the fuses can’t be reconnected after being broken)
If you’re worried that you might some day change your mind about the values to store in the ROM, an EPROM (Erasable PROM) might be a better choice.

In this case, rather than fuses, charge (permanently) stored in MOSFET gates determines whether each output bit is 0 or 1.

Charge can be moved one way (1 → 0, for example) by electrical burning, as with a PROM.

Can go the other way with exposure to high-intensity UV light:

- this erases the EPROM – i.e. it puts all the charge back to where it was when the EPROM was new.
EEPROM

- Even better is the EEPROM (Electrically Erasable PROM)
- With this technology, both the $1 \rightarrow 0$ and $0 \rightarrow 1$ changes can be accomplished electrically
  - we can reprogram our EEPROM without even removing it from the circuit board!
Programmable Logic

• So far, the memory devices we’ve talked about provide a single well-defined output for a fixed input
  – i.e., they do combinatorial logic
• But if one builds an IC with both gates and flip-flops (with the connections between them programmable) one can program in sequential logic as well
• This means that you don’t just program memory values, you program circuit behavior!
• There are two main classes of programmable logic
  – Programmable array logic (PAL)
  – Field programmable gate array (FPGA)
• FPGA’s have many more gates per IC, but PAL’s have better-understood signal timing
  – we’ll use the following PAL in this lab:
Altera EPM7128

- The programmable logic chip we’ll be using is the Altera EPM7128
  - This is an EEPROM using CMOS logic
  - Contains 2500 logic gates
    - plenty for us this semester…
  - 128 macrocells
  - 100 I/O pins
  - Will run at up to 150 MHz
    - Note, though, that chip clock speed is not the whole story
    - Need to account also for propagation delays through the logic
      (the ripple counter, for example, wouldn’t work at the maximum clock speed)
    - Should fully simulate the circuit to make sure your design behaves the way you expect
Figure 4. MAX 7000E & MAX 7000S Device Macrocell
Example: Divide-by-8 counter

- In some of the homework and lab exercises, you’ll want to be able to watch the output of your circuit change as time goes by.
- But the clock that comes on the board we’ll be using runs at 25 MHz
  - much too fast to see!
- The solution is to slow the clock down by a large factor.
- Slowing down by a factor of 2 is a start. The following circuit will do it:

- Let’s see how we can build one of these on the Altera chip.
1. Start the Max+Plus II software

2. From the Max+plusII menu, select “Grapic editor”
   - Use “Save as” to give it a name, such as “Divideby2”. If you’re using the computers in 272, save your files in the “Lock” folder

3. To set this for the particular chip we’ll be using, select “Assign → Device”
   - For device family, select MAX7000S
   - Uncheck the box that says “Show Only Fastest Speed Grades”
   - Select EPM7128SLC84-7

4. Start building the circuit using Symbol → Enter Symbol
   - Type “tff” (for T Flip-flop) in the Symbol Name box
• The following symbol appears:

![Symbol Image]

• We want to make sure there are no set or resets coming in, so we want to connect PRN and CLRN (as well as T!) to $+V_{CC}$
  – insert a symbol of type “vcc” and click and drag from the output to T, PRN and CLRN
  – circuit should now look like:
Inputs/Outputs

• The circuit has a single input (the clock signal) and a single output (Q). To tell the software that it needs to bring these signals to pins on the Altera chip, add symbols of type “input” and “output”, and connect them appropriately:

• Double-click where it says “PIN_NAME” to assign unique names to each pin
  – You have to do this!
  – I’ll use CLK_IN and CLK_OUT as the names
Compiling

- To see if the Altera can really implement the circuit we’ve built, we need to compile our design
- Select File → Project → Set project to current file
- Select Max+plus II → Compiler, and click Start
- Should see a box like:

  ![Compilation Success Message](image)

  - Clicking on the “rpt” symbol from the compiler window brings up a text message about the compilation
    - including what pin numbers correspond to CLK_IN and CLK_OUT
Simulating

- First we need to simulate the inputs
- Do Max+plus II → Waveform editor
- Do File → Save as. The default choice should be “divideby2.scf”. That’s fine, so click OK.
- Now right-click in the “Name” area of the waveform editor

![Waveform Editor Screenshot]

Right-click somewhere in here
• Select “Enter nodes from SNF”
• The following box appears:

![Enter Nodes from SNF]

- Click on the “List” button, then the “=>” arrow, then OK
• The waveform editor now looks like:

![Waveform Editor](image)

• It’s up to us to define how CLK_IN behaves
  - Since it’s a clock, we can just click on CLK_IN, and then select the box on the left
  - The default clock values are fine for now
• The waveform editor now looks like:

• To see a longer time sample, do View → Fit in window
  – You get:
• To see what the output looks like, selected Max+plusII → Simulator, and click Start

• The waveform editor box now looks like:

![Waveform Editor Screenshot]

• CLK.OUT is in fact a clock with half the frequency of CLK.IN. Success!
  – But dividing our 25 MHz clock by a factor of 2 doesn’t help much
• How do we get a larger division factor?
  – could start over, and build a circuit with more T flip-flops chained together
  – but that’s too much work!
• For an easier way, click on the graphic editor window, and select File → Create default symbol
• Now close the existing graphic editor window, and open a new (Max+plus II → Graphic editor)
  – and enter a symbol of type “divby2”
  – i.e., the circuit we just built
  – the symbol looks like:
- We can copy any device by pressing CTRL, then clicking on it
- Doing this twice, it’s easy to get a circuit like:

  ![Diagram of a divide-by-8 counter]

  This is a divide-by-8 counter!
  Add input and output pins, and we’re done…
  Now you should be able to see how to make a divide by $2^{24}$ (or so) counter