EEM232 Digital Systems I

Counters and Registers

Overview

• Counters
• Registers
Introducing counters

- Counters are a specific type of sequential circuit
- The state serves as the “output” (Moore)
- A counter that follows the binary number sequence is called a binary counter
  - n-bit binary counter: n flip-flops, count in binary from 0 to 2^n - 1
- Counters are available in two types:
  - Synchronous Counters
  - Ripple Counters
- Synchronous Counters:
  - A common clock signal is connected to the C input of each flip-flop

Synchronous Binary Up Counter

- The output value increases by one on each clock cycle
- After the largest value, the output “wraps around” back to 0
- Using two bits, we’d get something like this:

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Synchronous Binary Up Counter

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
<th>( D_1 = A'B + AB' )</th>
<th>( D_0 = B' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>( B )</td>
<td>( A )</td>
<td>( B )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

What good are counters?

- Counters can act as simple clocks to keep track of “time”
- You may need to record how many times something has happened
  - How many bits have been sent or received?
  - How many steps have been performed in some computation?
- All processors contain a program counter, or PC
  - Programs consist of a list of instructions that are to be executed one after another (for the most part)
  - The PC keeps track of the instruction currently being executed
  - The PC increments once on each clock cycle, and the next program instruction is then executed.
**Synch Binary Up/Down Counter**

- 2-bit Up/Down counter
  - Counter outputs will be 00, 01, 10 and 11
  - There is a single input, X.
    - \( X = 0 \), the counter counts up
    - \( X = 1 \), the counter counts down
- We’ll need two flip-flops again. Here are the four possible states:

```
00   01
11   10
```

---

**The complete state diagram and table**

- Here’s the complete state diagram and state table for this circuit

```
<table>
<thead>
<tr>
<th>Present State</th>
<th>Inputs</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1 Q0</td>
<td>X</td>
<td>Q1 Q0</td>
</tr>
<tr>
<td>00 01</td>
<td>0</td>
<td>00 01</td>
</tr>
<tr>
<td>00 01</td>
<td>1</td>
<td>00 01</td>
</tr>
<tr>
<td>00 01</td>
<td>0</td>
<td>00 01</td>
</tr>
<tr>
<td>00 01</td>
<td>1</td>
<td>00 01</td>
</tr>
</tbody>
</table>
```

---

EEM232 Digital Systems I  Counters and Registers
D flip-flop inputs

- If we use D flip-flops, then the D inputs will just be the same as the desired next states.
- Equations for the D flip-flop inputs are shown at the right.
- Why does $D_0 = Q_0'$ make sense?

<table>
<thead>
<tr>
<th>Present State</th>
<th>Inputs</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q_1$</td>
<td>$Q_0$</td>
<td>$X$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Synchronous Binary Up/Down Counter

![Synchronous Binary Up/Down Counter Diagram]
Unused states

- The examples shown so far have all had $2^n$ states, and used $n$ flip-flops. But sometimes you may have unused, leftover states.
- For example, here is a state table and diagram for a counter that repeatedly counts from 0 (000) to 5 (101).
- What should we put in the table for the two unused states?

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q_2$ $Q_1$ $Q_0$</td>
<td>$Q_2$ $Q_1$ $Q_0$</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 1 0</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1 1</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1 0 0</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1 0 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>0 0 0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>?? ?? ?</td>
</tr>
<tr>
<td>1 1 1</td>
<td>?? ?? ?</td>
</tr>
</tbody>
</table>

Unused states can be don’t cares...

- To get the simplest possible circuit, you can fill in don’t cares for the next states. This will also result in don’t cares for the flip-flop inputs, which can simplify the hardware.
- If the circuit somehow ends up in one of the unused states (110 or 111), its behavior will depend on exactly what the don’t cares were filled in with.

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q_2$ $Q_1$ $Q_0$</td>
<td>$Q_2$ $Q_1$ $Q_0$</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 1 0</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1 1</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1 0 0</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1 0 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>0 0 0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>?? ?? ??</td>
</tr>
<tr>
<td>1 1 1</td>
<td>?? ?? ??</td>
</tr>
</tbody>
</table>
...or maybe you do care

- To get the safest possible circuit, you can explicitly fill in next states for the unused states 110 and 111
- This guarantees that even if the circuit somehow enters an unused state, it will eventually end up in a valid state
- This is called a self-starting counter

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2 Q1 Q0</td>
<td>Q2 Q1 Q0</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 1 0</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1 1</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1 0 0</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1 0 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>0 0 0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>1 1 1</td>
<td>0 0 0</td>
</tr>
</tbody>
</table>

More complex counters

- More complex counters are also possible:
  - It can increment or decrement, by setting the UP input to 1 or 0
  - You can immediately (asynchronously) clear the counter to 0000 by setting CLR = 1
  - You can specify the counter’s next output by setting D3-D0 to any four-bit value and clearing LD
  - The active-low EN input enables or disables the counter
  - When the counter is disabled, it continues to output the same value without incrementing, decrementing, loading, or clearing
  - The “counter out” CO is normally 1, but becomes 0 when the counter reaches its maximum value, 1111
An 8-bit counter

- As you might expect by now, we can use these general counters to build other counters.
- Here is an 8-bit counter made from two 4-bit counters:
  - The bottom device represents the least significant four bits, while the top counter represents the most significant four bits.
  - When the bottom counter reaches 1111 (i.e., when CO = 0), it enables the top counter for one cycle.
- Other implementation notes:
  - The counters share clock and clear signals.
  - Hex displays are used here.

A restricted 4-bit counter

- We can also make a counter that “starts” at some value besides 0000.
- In the diagram below, when CO=0 the LD signal forces the next state to be loaded from D3-D0.
- The result is this counter wraps from 1111 to 0110 (instead of 0000).
**Another restricted counter**

- We can also make a circuit that counts up to only 1100, instead of 1111
- Here, when the counter value reaches 1100, the NAND gate forces
  the counter to load, so the next state becomes 0000

**Counters - Summary**

- Counters serve many purposes in sequential logic design
- There are lots of variations on the basic counter
  - Some can increment or decrement
  - An enable signal can be added
  - The counter’s value may be explicitly set
- There are also several ways to make counters
  - You can follow the sequential design principles to build counters
    from scratch
  - You could also modify or combine existing counter devices
Registers

- A common sequential device: Registers
  - They’re a good example of sequential analysis and design
  - They are also frequently used in building larger sequential circuits
- Registers hold larger quantities of data than individual flip-flops
  - Registers are central to the design of modern processors
  - There are many different kinds of registers
  - We'll show some applications of these special registers

What good are registers?

- Flip-flops are limited because they can store only one bit
  - We had to use two flip-flops for our two-bit counter examples
  - Most computers work with integers and single-precision floating-point numbers that are 32-bits long
- A register is an extension of a flip-flop that can store multiple bits
- Registers are commonly used as temporary storage in a processor
  - They are faster and more convenient than main memory
  - More registers can help speed up complex calculations
A basic register

- Basic registers are easy to build. We can store multiple bits just by putting a bunch of flip-flops together!

- A 4-bit register is given below
  - This register uses D flip-flops, so it’s easy to store data without worrying about flip-flop input equations
  - All the flip-flops share a common CLK and CLR signal

74x175 – 4-bit register
A shift register “shifts” its output once every clock cycle.

- SI is an input that supplies a new bit to shift “into” the register.
- For example, if on some positive clock edge we have:
  
  \[
  \begin{align*}
  SI &= 1 \\
  Q_0-Q_3 &= 0110
  \end{align*}
  \]

  then the next state will be:
  
  \[
  Q_0-Q_3 &= 1011
  \]

- The current \( Q_3 \) (0 in this example) will be lost on the next cycle.

**Shift direction**

- The circuit and example make it look like the register shifts “right.”
- But it really depends on your interpretation of the bits. If you consider \( Q_3 \) to be the most significant bit instead, then the register is shifting in the opposite direction!
Serial data transfer

- One application of shift registers is converting between “serial data” and “parallel data”
- Computers typically work with multiple-bit quantities
  - ASCII text characters are 8 bits long
  - Integers, single-precision floating-point numbers, and screen pixels are up to 32 bits long
- But sometimes it’s necessary to send or receive data serially, or one bit at a time. Some examples include:
  - Input devices such as keyboards and mice
  - Output devices like printers
  - Any serial port, USB or Firewire device transfers data serially
  - Recent switch from Parallel ATA to Serial ATA in hard drives

Receiving serial data

- To receive serial data using a shift register:
  - The serial device is connected to the register’s SI input
  - The shift register outputs Q3-Q0 are connected to the computer
- The serial device transmits one bit of data per clock cycle
  - These bits go into the SI input of the shift register
  - After four clock cycles, the shift register will hold a four-bit word
- The computer then reads all four bits at once from the Q3-Q0 outputs.
Sending data serially

- To *send* data serially with a shift register, you do the opposite:
  - The CPU is connected to the register’s D inputs
  - The shift output (Q3 in this case) is connected to the serial device

- The computer first stores a four-bit word in the register, in one cycle

- The serial device can then read the shift output
  - One bit appears on Q3 on each clock cycle
  - After four cycles, the entire four-bit word will have been sent

Registers in Modern Hardware

- Registers store data in the CPU
  - Used to supply values to the ALU
  - Used to store the results

- If we can use registers, why bother with RAM?

<table>
<thead>
<tr>
<th>CPU</th>
<th>Size</th>
<th>L1 Cache</th>
<th>L2 Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium 4</td>
<td>32 bits</td>
<td>8 KB</td>
<td>512 KB</td>
</tr>
<tr>
<td>Athlon XP</td>
<td>32 bits</td>
<td>64 KB</td>
<td>512 KB</td>
</tr>
<tr>
<td>Athlon 64</td>
<td>64 bits</td>
<td>64 KB</td>
<td>1024 KB</td>
</tr>
<tr>
<td>PowerPC 970 (G5)</td>
<td>64 bits</td>
<td>64 KB</td>
<td>512 KB</td>
</tr>
<tr>
<td>Itanium 2</td>
<td>64 bits</td>
<td>16 KB</td>
<td>256 KB</td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>64 bits</td>
<td>up to 4 MB</td>
<td></td>
</tr>
<tr>
<td>MIPS R14000</td>
<td>64 bits</td>
<td>32 KB</td>
<td>16 MB</td>
</tr>
</tbody>
</table>

Answer: Registers are expensive!

- Registers occupy the most expensive space on a chip – the core
- L1 and L2 are very fast RAM – but not as fast as registers.
Registers - Summary

- A register is a special state machine that stores multiple bits of data.
- Several variations are possible:
  - Parallel loading to store data into the register
  - Shifting the register contents either left or right
  - Counters are considered a type of register too!
- One application of shift registers is converting between serial and parallel data.
- Most programs need more storage space than registers provide
  - RAM is used to address this problem
- Registers are a central part of modern microprocessors.