Datapath Components

Overview

In this lesson we will

- ✓ Examine the system datapath and common components
- ✓ Introduce and model several commonly used combinational circuits
- ✓ Introduce and model several commonly used sequential circuits
- ✓ Explore implementations for the four fundamental arithmetic operations
- ✓ Introduce and model the basic types of register type storage elements

Introduction

Have determined that can decompose system into

Datapath and control

Identified

Datapath as

Route(s) data takes through system

Components through which data passes

As it moves through system

Control elements

Mechanisms or means for orchestrating /managing flow along datapath

Observed that datapath components comprise myriad of collection of

Analog, digital, mechanical, etc. pieces

Key point with each

- ✓ Each encapsulates a specific high-level function

  Piece of functionality
- ✓ Each contributes that functionality

  To support intended function of the design

Want to now briefly touch on some of more common digital ones

Earlier identified number of common operations or functions to be performed

- ✓ Store
- ✓ Transfer
  - Select
  - Direct or route
- ✓ Modify
  - Arithmetic
  - Logical
  - Encode / decode

Let's examine components that affect those operations

Selection – Multiplexing

In context of datapath

Selection entails choosing data from any of number of different sources

- ✓ Potential sources include
  - Memory
  - Supporting or companion processor
  - PLD
  - Port
- ✓ Potential targets include same set

From abstract perspective

Selection is simply multiplexing in time or frequency

Following diagram presents high-level model of

Selection function

The illustrated module

Inputs

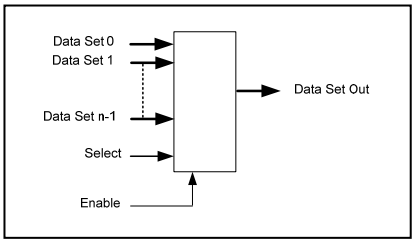
n data sets with m signals each

$\log_2 n$ selector lines

Output enable control

Enables or disables output

Either to set state or tristate



Outputs

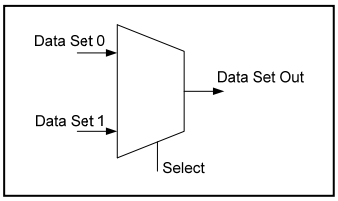
One data set with m signals

Selected from the n input sets

Simple selector / multiplexer

Choosing between two alternate data sets

Given in accompanying figure



The enable control may or may not be included

Design can be extended

Increasing

Number of inputs

Corresponding number of selector lines

In Verilog it can be modeled as

Data flow or behavioural level as

If-else construct

Switch or case statement

Gate level as

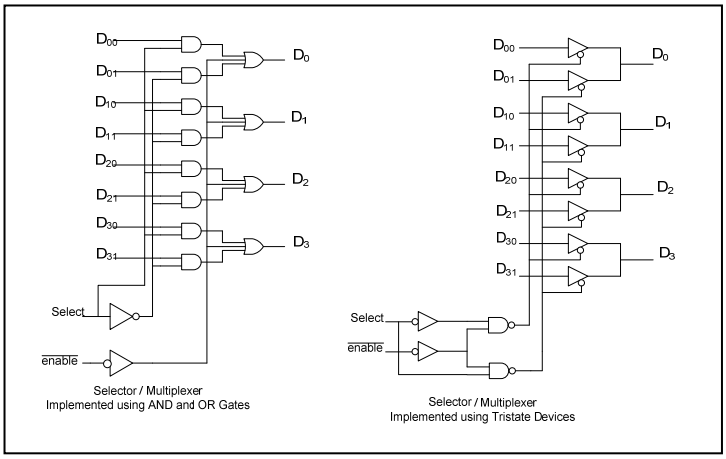
Collection of tristate gates

Combination of AND and OR gates

Implemented in hardware

As expressed in gate level model

Two hardware implementations illustrated in following figure



Routing – Demultiplexing

Routing or demultiplexing is reverse of selection or multiplexing

Whereas the latter

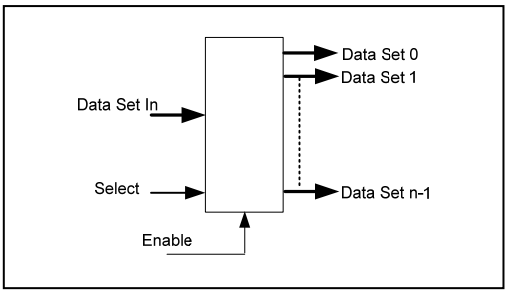
Accepts signals from multiple sources

Directs to single target

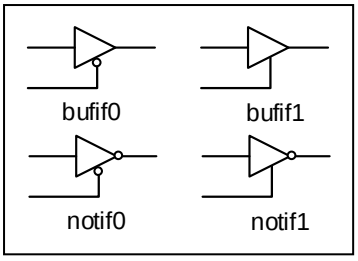
Former

Accept signals from single source

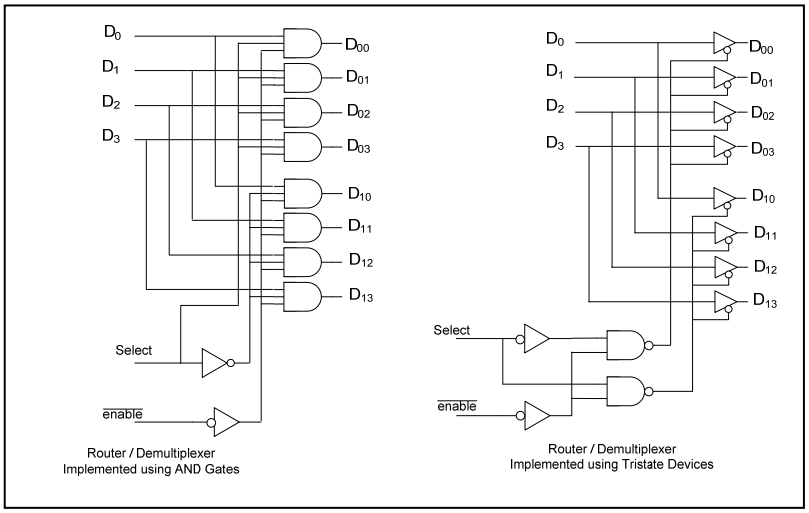
Direct to multiple targets
From abstract perspective
    Routing is simply demultiplexing in time
Following diagram presents high-level model of
    Routing - demultiplexing function



The illustrated module
    Inputs
        One data set with m signals
        $\log_2$ n selector lines
            Selected from the n input sets
        Output enable control
    Outputs
        n data sets with m signals each
As see with multiplexer in Verilog at can be modeled at
    Data flow or behavioural level as
        If-else construct
        Switch or case statement
    Gate level as
        Collection of tristate gates
        Combination of AND and OR gates
Implemented in hardware
    As expressed in gate level model



Two hardware implementations illustrated in following figure



Encoding – Decoding

Encoding and decoding are fancy names

For collections of combinational logic that

Map one set of bit patterns to another

Simple examples include

BCD to one of 10

Octal to one of 8

Hexadecimal – hex to one of 16

BCD to 7 segment

Used as mapping

From

BCD digit

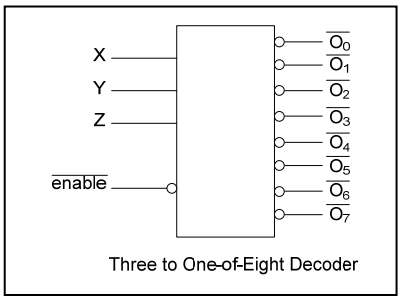
To

Anode or cathode drive signals on 7 segment display

Modifications to these basic operations

Enable / disable Input or output

The following graphic gives high level view of



- Octal (3 line) to one of eight decoder
- Inputs
  - 3 bit octal value ranging from 0..7
  - Active low enable signal
    - When active
      - Selected output is asserted low
    - When inactive
      - All outputs asserted high
- Outputs
  - 8 active low outputs
    - Active output corresponds to octal value on input

- Such a device may be used in conjunction with
  - Selector or router to choose
    - Source or destination data set
  - Memory array
    - Control chip select signals

Arithmetic Devices

- Arithmetic devices comprise the four major functions
  - Addition, subtraction, multiplication, division
  - Also form building blocks for more complex
    - Arithmetic and logical units – ALU

- Each of the blocks supporting binary operations
  - Accepts two sets of inputs
    - The operands
  - Performs an operation on inputs
    - To yield required output

Addition and Subtraction

Addition and subtraction relatively straight forward operations

Executed much as one would expect

Addition

The basic building block for addition

Denoted *full adder*

Accepts three inputs

2 data bits to be added

1 bit denoted carry in

Permits devices to be cascaded

To produce multibit adder

Produces two outputs

- ✓ Sum

  Result of summing two data bits and carry in

- ✓ Carry out

  Carry that may be generated from summing operation

  Permits devices to be cascaded

  To produce multibit adder

  Must recognize and address

  - Carry out
  - Overflow

More primitive building block

Denoted *half adder*

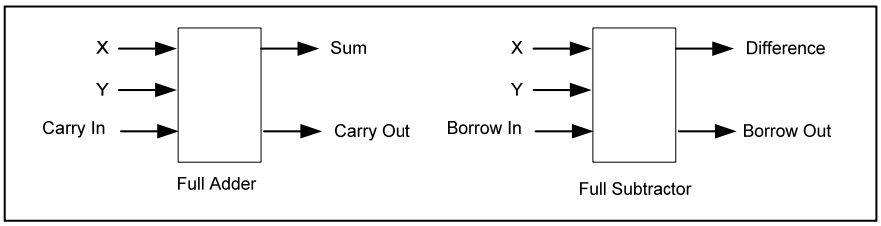
Identical to full adder

Except does not support carry in signal

Subtraction

- Basic subtraction function almost identical to addition
- The basic building block for addition
  - Denoted *full subtractor*
    - Accepts three inputs
      - 2 data bits to be subtracted
      - 1 bit denoted borrow in
        - Permits devices to be cascaded
          - To produce multibit subtractor
    - Produces two outputs
      - Difference
        - Result of subtracting two data bits and borrow in
      - Borrow out
        - Borrow that may be generated from subtraction operation
          - Permits devices to be cascaded
            - To produce multibit subtractor
- Full subtractor
  - Rarely used in contemporary designs
  - Subtraction typically implemented using
    - 2’s complement addition

Following figures give high level block depiction of both functions



Multibit Addition / Subtraction

Variety of techniques for implementing

Multibit addition and subtraction functions

Ranking designs from slowest and cheapest to fastest and most expensive

We find

- Carry save addition

  Utilizes

  Single full adder

  To perform addition

  Storage device

  Save carry out from i^{th} addition

  Feedback to input as carry in to $(i + 1)^{th}$ addition

  Operation

  Addition performed one bit at time

  Starting with LSB

- Ripple carry

  Utilizes

  N full adders connected in cascade

  To perform N bit addition

  Sum bits

  Appear in parallel out of the adder complex

  Carry out from ith stage becomes

  Becomes input as carry in to $(i + 1)^{th}$ stage

  Hence the name ripple carry

  Zeroth carry in set to 0

- Anticipated carry or Carry Look Ahead – Parallel adder

  Utilizes

  N full adders

  To perform N bit addition

  Sum bits

  Appear in parallel out of the adder complex

  Carries in to all stages

  Computed at same time as all sums

  Based upon

  Basic carry out equation

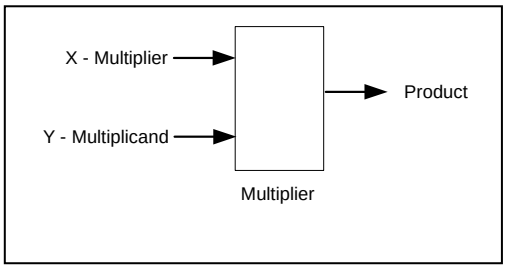
  Knowledge that carry out of i^{th} stage

  Is carry in to $(i+1)^{th}$ stage

- Look-up Table
  Utilizes
    Memory
    Sums or differences of two n bit words - w0 and w1
      Precomputed and stored in memory
    W0 interpreted as
      Lower half of address to memory
    W1 interpreted
      Upper half of address to memory

Multiplication

Like addition and subtraction
  Variety of different ways to implement
As with others
  Multiplication is binary operation
  Consequently from diagrammatic point of view
    Represented similarly as following block figure illustrates



Multiplication function found frequently in
  Digital signal processing applications
  Computations of FFTs and similar functions
Demand in such cases
  Emphasizes number of computations per second
To accommodate diversity of applications
  Implementations have same range as found in adders

Can classify into two general categories

- ✓ Serial

  Most familiar technique here called shift and add

  Replicates in hardware

  Familiar pencil and paper algorithm

  Some speed up can be incorporated by recognizing

  Multiplication by zero does not contribute final result

- ✓ Parallel

  Parallel approaches can be similarly decomposed

  Here we find

  - Algorithmic
  - Memory based

  Algorithmic approaches

  Among more common

  - Booth algorithm
  - Wallace tree
  - Dada tree

  Recognizing that arithmetic is

  Basically combinational operation

  Multiplication is repeated addition

  Common thread with each is

  Combinational logic array

  Interconnected in such a way so as to

  Implement shift and add algorithm in parallel

  Rather than in serial

  Multiplier and multiplicand entered in parallel

  Partial product array produced and reduced

  To yield two numbers

  That can be added using high-speed adder

  To give final product

Algorithmic approach illustrated using Wallace tree

Wallace Tree

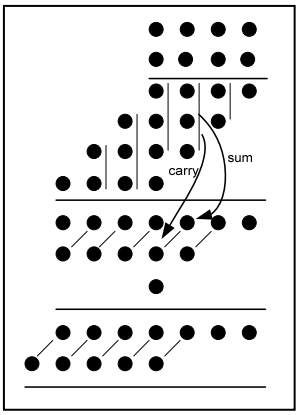
A convenient scheme for fast multiplication uses parallel approach

Easily implemented in combinational logic

Algorithm illustrated in following figure

Will illustrate for 4 x 4 multiplier

Use dot to indicate either 0 or 1 bit



The first step

Compute the partial product array

Implement using AND gate array

Multiply same as logical AND

Next reduce partial product array

Using collection of full adders

3 input 2 output

Within each column

Add bits in groups of 3

Indicated by vertical lines in each column

Sum and carry bits propagate as illustrated

To next level of reduction

Sum bit into same column

Next level

Carry bit one column to left

Next level

Continue to reduce until two rows remain

Add remaining two rows using parallel adder

Memory Based

A memory based scheme similar to that for addition

Multiplier and multiplicand

Serve as addresses into memory in hardware model

Precomputed product stored at addressed location

Concept implemented in software

As look-up table – LUT

Division

Division more of challenge

Several reasons

Must deal with

Quotient and remainder

Computing inverse which would reduce division to multiplication

Not simple task

Not as easy to implement

Parallel schemes

Recognize that multiplication basically repeated addition

Appropriately managed

Division can be implemented as repeated subtraction

Also appropriately managed

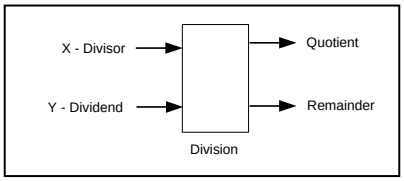
Basic schemes variants on

Shift and add multiplication scheme

Two most common

Restoring and nonrestoring division

High level diagram given in accompanying figure



Arithmetic Logic Unit – Comparison

Extending discussion of arithmetic blocks we have

- Arithmetic Logic Unit – ALU

  Many basic arithmetic and logical functions

  Into single device

- Comparison operation

  Important for testing two entities for

  Equality

  Inequality

  Is one entity greater or less than another

ALU

Common building block in most CPU type functions

Device is able to perform variety of arithmetic and logical operations

On two N bit numbers

Generate N bit output

Control inputs specify operation to be performed

- Simple operations

Basic addition, subtraction, multiplication, division

In very simple devices operations limited to

Addition and subtraction

Bit wise logical operations

AND, OR, NOR, XOR

Bit shift operations

Shifting or rotating word

Left or right

Sign extension

- Complex operations

As complexity of supported operations increases

Cost, size, and power all increase

Complexity can branch in different directions

Speed

Perform elementary arithmetic or logical operations

One or several clock cycles

Barrel shifter is good example

Can shift data word specified number of bits

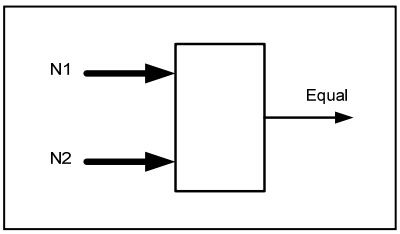
In single clock cycle

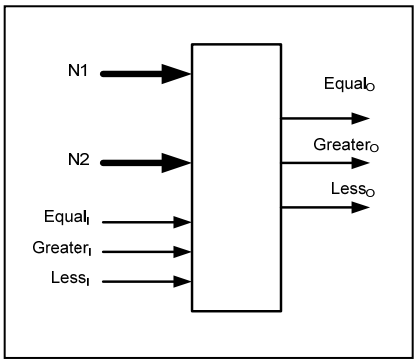
Functionality

Implement operations such as floating point math

In hardware

Comparison





Device will accept two N bit binary numbers
Depending upon design
Will produce

- Single output
  Indicating that two input numbers are equal
  High level diagram given in accompanying figure
- Two outputs
  N1 is larger than N2
  N2 is larger than N1
  If both are true
  Numbers are equal
- Three outputs
  N1 is larger than N2
  N2 is larger than N1
  N1 is equal to N2

More complex designs
Include inputs signaling
Less than
Greater than
Equal
Using such inputs
Can cascade series of comparators
To compare two M digit numbers
High level diagram given in accompanying figure

Designing with Storage Elements – Registers

Among the more common and useful datapath blocks

Registers and latches

A single latch or flip-flop can store a single bit of information

Single logical 1 or logical 0

Collection of such devices treated as a single entity

Called a *register* or *latch*

These are different and a bad choice of names

Depending upon mechanism by which data

Enters and is stored in device

- Register utilizes

  Strobe or clock to enter data

- Latch utilizes

  Gate

Storage Registers

Registers used to hold data

Form one small component of the memory system

In CPLD, FPGA, microprocessor

Often they are used for temporary storage

On software side

Frequently used values

Control variable in a *for* or *while* loop

On hardware side

Data written to or read from I/O port

No restrictions placed on the size of a register

Number of flip-flops or latches

However common practice to design binary sized groups

4, 8, 16, or 32 bit registers

Size of a register is often more appropriately called its *width*

Devices comprising a register all have

- Common clock or gate
- May have common reset (and preset)
- Work as a single unit

As noted earlier common parlance refers to the device as

- A *latch*
  If comprised of gated devices
    Typically such devices are single bit latches themselves
- A *register*
  If the member devices are clocked or strobed devices
    Typically such devices are flip-flops

The accompanying logic diagram illustrates

Four bit latch and a similar sized register

Any values placed on inputs to device

Clocked or gated to the outputs

With a simple inversion

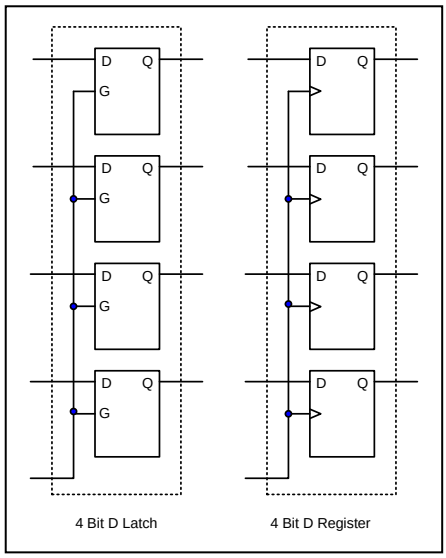
Sense of gate or clock can be modified

Register is sometimes implemented with

Common reset signal as well

Important point

Set of devices comprising the part are treated as a group



Shift Registers

Like the basic register

Shift register is a collection of flip-flops that can store data

Shift register has the additional capability

Shift the stored data to the left or to the right

As with the basic register

Common practice to design the shift register in binary sized groups

Devices comprising a shift register all have,

- Common clock to effect the shift operation
- May have common reset
- Work as a single unit

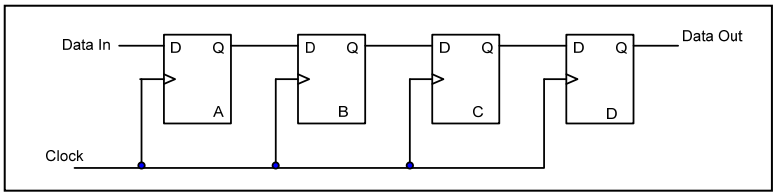
Why not use gated latches as the building block for a shift register?

Shift Right Shift Register

Four D flip flops in the configuration shown

Implement a *four bit shift right - shift register*

Illustrate the basic architecture for the family of devices



Assume that the initial state of all devices is logical 0

At time t_0 we put a logical 1 on the data input and issue one clock pulse

State of the flip-flops is given

Time	*Data*	Q_A	Q_B	Q_C	Q_D
t_0	1	0	0	0	0
t_1	0	1	0	0	0

After the clock pulse

Input data bit has been stored into the first flip flop.

At time t_1

We put a 0 on the data input and issue another clock pulse

State of the flip-flops now appears as

Time	*Data*	Q_A	Q_B	Q_C	Q_D
t_0	1	0	0	0	0
t_1	0	1	0	0	0
t_2	0	0	1	0	0

After the clock pulse

Logical 0 has been stored into the first flip flop

Logical 1 from the first flip flop has moved to the second

At time t_2

We leave the 0 on the data input and issue another clock pulse

State of the flip-flops now given as

Time	*Data*	Q_A	Q_B	Q_C	Q_D
t_0	1	0	0	0	0
t_1	0	1	0	0	0
t_2	0	0	1	0	0
t_3	0	0	0	1	0

After the clock pulse logical 0 has

Propagated to the second flip flop

Logical 1 from the second flip flop has moved to the third

At time t_3

We leave the 0 on the data input and issue another clock pulse

State of the flip-flops appears as

Time	*Data*	Q_A	Q_B	Q_C	Q_D
t_0	1	0	0	0	0
t_1	0	1	0	0	0
t_2	0	0	1	0	0
t_3	0	0	0	1	0
t_4	0	0	0	0	1

After the clock pulse

Logical 0 has propagated to the third flip flop

Logical 1 from the third flip-flop has now moved to the fourth

With each clock pulse

Stored data is shifted one position to the right

New data bit is entered into the first flip-flop

Data entered into the device will appear on the output

After N clock pulses

Output stream is thus

Delayed version of the input stream

Making it an effective tool when ever

Embedded application requires a well controlled delay

Based upon these observations

Two equations formalizing behavior of the device can be written

For the 0^{th} flip flop,

D_0 = data

For the i^{th} D flip flop

$D_i = Q_{i-1}$

Design implements a *right shift by one four-bit shift register.*

Verilog model for such a device is given

We begin with the RTL model

```
//  RTL Model - Four Bit Shift Right Shift Register

module ShiftRegister4(dataOut, dataIn, clk, por);
//  declare the inputs and outputs
   input   dataIn, clk, por;
   output  dataOut;

   reg [3:0] data;         //  implements the shift register
   reg dataOut;

   //  build the shift register
   always@ (negedge por or posedge clk)
   begin
      //  reset the register
      if(por==0)
      begin
         data<= 4'b0;
         assign dataOut = 0;
      end

   //  implement shift operation
      else
      begin
         assign dataOut = data[3];
         data <= {data[2:0], dataIn};
      end
    end
endmodule
```

Design implements a master reset called *POR – Power On Reset*
Such a reset is essential in any embedded application
    Ensure that the system will always start in a known state
        Typically that state is the all zeros condition
Most significant bit, D3 or data[3], is on the right
    Data enters on the left

RTL implementation utilizes a Verilog array type data structure
    To express the functionality

Structural or gate level implementation for the design

Given in the code module in

```
module DFF(q, qBar, D, clk, rst);
  input D, clk, rst;
  output q, qBar;
  parameter delay0 = 2;     //  delay reset to q
  parameter delay1 = 3;     //  delay clock to q
  parameter delay2 = 2;     //  delay for qBar with respect to q

  reg q;

  not #delay2 n1 (qBar, q);

  always@ (negedge rst or posedge clk)
  begin
    if(rst==0)
      #delay0 q = 0;
    else
      #delay1 q = D;
  end
endmodule

//  Structural Model - Four Bit Shift Right Shift Register

module ShiftRegister4(dataOut, dataIn, clk, por);
//  declare the inputs and outputs
    input   dataIn, clk, por;
    output  dataOut;

//  bulid the shift register
    DFF ff3(dataOut, q3Bar, q2, clk, por);
    DFF ff2(q2, q2Bar, q1, clk, por);
    DFF ff1(q1, q1Bar, q0, clk, por);
    DFF ff0(q0, q0Bar, dataIn, clk, por);

endmodule
```

D flip-flop model is included

Has been simplified to only support a reset signal

Propagation delay parameters

Implemented through the underlying flip-flop implementation

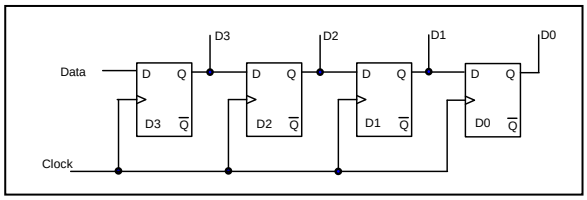
Rather than in the shift register itself

Parallel In / Serial Out – Serial In / Parallel Out Left Shift Registers

Shift register is also a convenient means

Converting between serial and parallel data

First diagram implements a simple four-bit serial to parallel converter



Four bit word is entered into the shift register

In serial through the input labeled *Data*

After four clock pulses

Word appears on the four output lines labeled *D3..D0.*

One common extension to the basic design

Use a tristate buffer on the outputs

Permit the outputs of several such devices to be multiplexed

Onto a common bus

As implemented one must count the number of bits entered

Ensure that the register is not overrun

Addition of one flip-flop to incorporate a marker bit into the design

Can provide an alternate approach

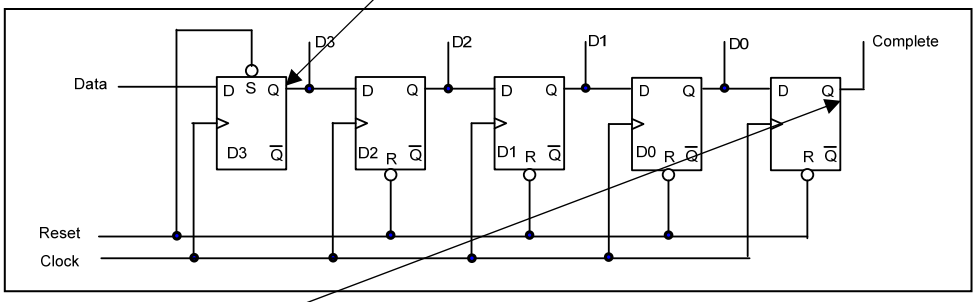
Prior to entering the data word

Reset input is asserted

Following the reset

0^{th} stage is in the logical 1 state

All others are at logical 0's - a marker bit



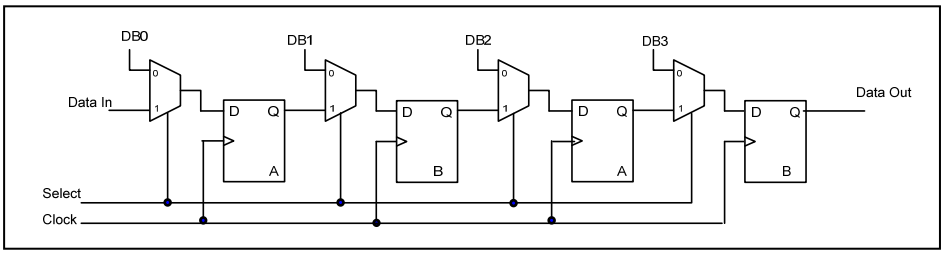
After four shifts

Logical 1 is in the last state

Complete signal is now asserted

Implementing a parallel in / serial out shift register
    Entails adding a two-to-one multiplexer on the input of each stage
    Plus selector control input
        Select between loading and shifting
Such a design is presented in the diagram below

When the *Select* input is in the logical 1 state
    Circuit acts like a four bit shift right shift register
When it is in the logical 0 state
    Parallel input data can be stored on the next clock rising edge



Linear Feedback Shift Registers

Linear feedback shift register (LFSR)
    Finds wide application in any embedded applications
        That utilize pseudo-random sequences
Such applications include
    Random noise generation
    Development of ‘random’ vectors in test systems
    Encoding and encryption,
    Wireless telecommunication systems utilizing
        CDMA or spread spectrum techniques

One cannot generate truly random numbers
    Using a finite state machine
Finite number of states
    Ensures that any path through the sequence of states
        Must repeat eventually
Best that can be expected is
    Period of the machine is ‘very long’

Thus, such a machine is called *pseudo-random*

Upper limit to the length of any such sequence is given by $2^n - 1$

Where n is the number of flip-flops in the shift register

Such a sequence is called a *maximal length sequence*

Shift register configuration

Described as a *maximal length shift register*

Upper bound is not 2^n as one might expect with n stages

Because the all zero state is not permitted

Once the generator enters the all zero state

Will not be able to exit

Because of their application to noise generation

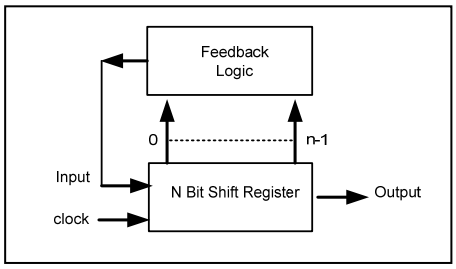
Maximal length LFSRs sequences are often termed

Pseudo noise sequences or *PN sequences*

High level block diagram for such a design

Called a *Linear Feedback Shift Register, LFSR*

Given in accompanying figure



Subset of the outputs from the shift register

Fed back as input data according to the polynomial given in

$$input = v_0 + v_1 X + v_2 X^2 + v_3 X^3 + ... + v_{n-1} X^{n-1}$$

$v_i = 0$ *or* 1

$+$ *implemented as an exclusive OR*

X_i *represent the flip – flop outputs*

Length of the generated sequence

Appearing on the output as a series of 0's and 1's

Determined by the starting value in the shift register

By which outputs are fed back

Specified by the values for the v_i

Generator will produce a maximal length sequence

If connection polynomial is irreducible

Can't be factored

Such a polynomial is called a *primitive polynomial*

$$input = 1 + X + X^4$$

Gate level Verilog model for the design of the LFSR

Given in the code module

```
module LFShiftRegister4(q3, q2, q1, q0, feedBack, clk, por);

//  declare the inputs and outputs
    input     clk, por;
    output    q3, q2, q1, q0, feedBack;
    reg       pullUp;

    initial
      pullUp = 1;

    xor       xr0 (feedBack, q2, q3);
//  bulid the shift register
    DFF       ff3(q3, q3Bar, q2, clk,  por, pullUp);
    DFF       ff2(q2, q2Bar, q1, clk, por, pullUp);
    DFF       ff1(q1, q1Bar, q0, clk, por, pullUp);
    DFF       ff0(q0, q0Bar, feedBack, clk, por, pullUp);

endmodule
```

D flip-flop used in the implementation

Has both an asynchronous *set* and an asynchronous *reset* input

Since only the *reset* input is used

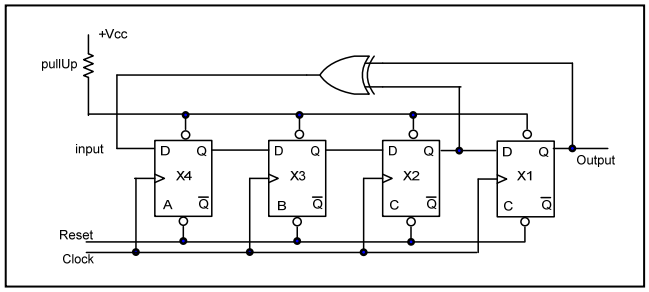
Set input must be defined

Cannot be left floating

Variable *pullUp* serves that purpose

An LFSR configured according to the given polynomial

Illustrated in the following diagram



Designing with Storage Elements – Counting and Dividing

Sequential machines and finite state automata

Form the theoretical models of computation

Upon which we base most of the computation and control capability

Found in modern digital systems

Counting and dividing

Essential tasks in a wide variety of contemporary embedded applications

Designs implementing such capability

Represent some of the simpler sequential machines

We find such capability supported

Inside of a

Microprocessor

Through a number of user programmable counters / timers

Programmable logic devices

Through user designed counters / timers

Outside of either

With the implementation of specialized MSI or LSI

Timing and counting functions.

We employ counters to

- Accumulate events
- Count bits
- Determine when or if a specified number of events have occurred

We use timers (a simple variant on a counter)

To measure elapsed time between events in an application

To delay an operation for a specified time after an event

Dividers are primarily used to develop a lower frequency from a higher

In the ensuing discussions will base most designs on the D flip-flop

Because it finds common application in most of the implementation mediums

VLSI, FPGAs, CPLDs

It is attractive because

It is easy to implement

It presents a very small footprint in integrated implementations

Dividers

Dividers find frequent application in designs where

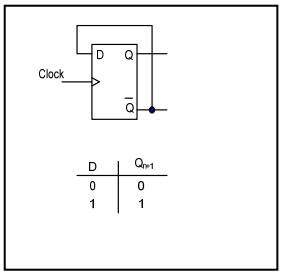
Must produce a lower frequency signal from higher one.

Divide by Two

Simplest such circuit

Accepts an input frequency

Produces one of half the frequency as output



Implementation of a divide by two circuit

Rather straight forward

Begin with a D flip flop

Connect the Q output back to the D input

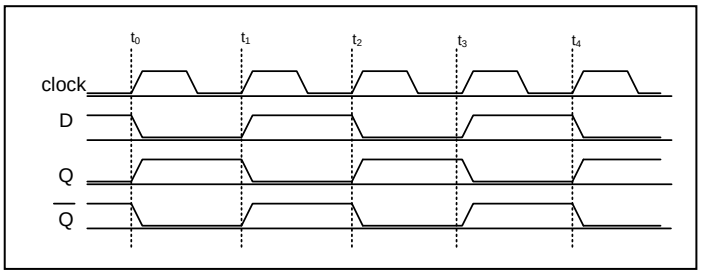
From the truth table for the flip-flop

Will alternate between states if

Configured as shown then clocked

Neglecting delays

Output of the device will appear as in following the timing diagram



On each rising edge of the clock

Flip-flop changes state

Each such occurrence

New state of the Q output is fed back into the input of the flip-flop

Will thus affect the value of the next state via the D input

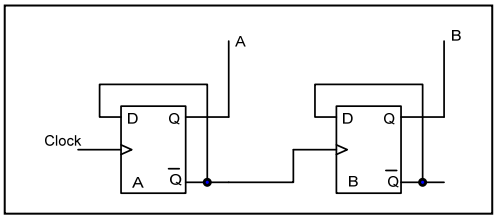
After several cycles of the clock

Clearly evident that the frequency of the signal at the Q output

One half of that of the clock

Input frequency has been divided by two

Asynchronous Dividers and Counters



Can extend the circuit as shown

Second flip-flop, B, is clocked by the Q output of the first flip-flop, A

When the Q output of A changes state from logical 1 to logical 0

Q output will change state from logical 0 to logical 1

On such a transition

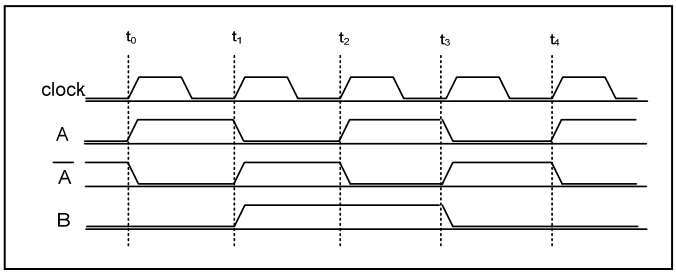
Flip-flop B will change state

B will be clocked every other time A changes state

At one fourth the frequency of the clock

Observe how we label the output signals on the flip-flops

Timing diagram is now given



The circuit is called by several names

A *divide by four* circuit

Because the output is one fourth of the input frequency

Qualified as an *asynchronous divide by four* circuit

Because the two flip-flops are not clocked by the same signal

Based upon the sequence of states through which the circuit transitions

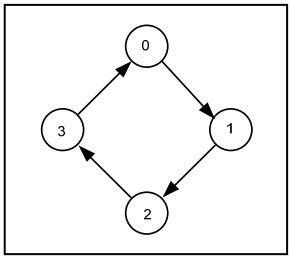
{B,A = 00, 01, 10, 11}

Circuit is also referred to as an *asynchronous, 2 bit, binary up counter*

It is counting up from the initial state of 00

The counting sequence is in binary

The state diagram and state table for the circuit is given



Present State $t = t_n$	Next State $t=t_{n+1}$
BA	BA
0 00	1 01
1 01	2 10
2 10	3 11
3 11	0 00

Edges in the state diagram are not labeled

Because there is no input signal causing the state change

Other than the clock which is not shown

Nodes or states labeled to reflect the binary value of the two state variables A and B.

Left hand column labeled *Present State*

Illustrates successive current states

Right hand column labeled *Next State*

Identifies the successor or next states

State the system will be in at the next time tick

Observe that because of the way logic drawings are commonly presented signal flow from left to right and top to bottom

Least significant bit of the counter appears on the left hand side

Little endian notation

Configured as it is

Flip-flop B cannot change state until after flip-flop A changes

As long as all we are doing is dividing we have no problem

If many such stages are cascaded

Will encounter significant delay as each stage changes state

Last stage cannot change state

Until all preceding stages have changed

Such a design is called

Asynchronous since the clocking of successive stages

Not synchronized to a master clock

Also called a *ripple counter*

Because a change in the first stage

Ripples through the intermediate stages

Eventually reaching the last

We cannot decode any of the state variable patterns

Without running the serious risk

Static and dynamic hazards

To see the significance of the effects of the delay

Assume that each device has a clock to Q propagation delay

m time units

Let the first flip-flop be clocked at time t_0

- The first stage output will appear at time $t = t_0+m$.
- The second stage output will appear m time units after the output of the first or at $t = t_0+2m$.
- For n stages, the final output will appear at $t = t_0+2mn$ worst case.
- Let m have a value of 10ns
- The output of the last stage of a 10 stage ripple counter will change states 200ns after the initial clock edge.
- If the input clock has a frequency of 1 MHz, the delay is 20% of the clock's period.

Previous analysis illustrates why ripple counters

Typically don't find wide application as general purpose counters or timers

They can be very effective

Dividing a higher frequency signal down to a lower one

Synchronous Dividers and Counters

Synchronous design is the preferred choice for a counter or timer
All stages are synchronized to a common clock
Each flip-flop output signal changes at approximately the same time
The state diagram and state tables will remain unchanged

Working with
- Characteristic equation and truth table for the D flip-flop
- State table for the counter

Can develop the D input equations for the two flip-flops

From the definition of the D flip-flop
- As expressed by either the truth table or characteristic equation
  - Conclude that for the state of the device to be a logical 1 at time t_{n+1}
    - D input must be a logical 1 at time t_n
      - Otherwise the state will be a logical 0

From state table we determine that
- From state 0
  - Counter must transition to state 1
  - In doing so flip-flop A
    - Must change state from logical 0 to logical 1
  - Flip-flop B must remain unchanged
  - Therefore D_A must be a logical 1
- From state 1
  - Counter must transition to state 2
  - In doing so flip-flop A
    - Must change state from logical 1 to logical 0
  - Flip-flop B must change state from logical 0 to logical 1
  - Thus, D_A must be a logical 0 and D_B must be a logical 1
- From state 2,
  - Counter must transition to state 3
  - In doing so flip-flop A
    - Must change state from logical 0 to logical 1
  - Flip-flop B must not change state
  - Thus, D_A and D_B must both be a logical 1
- From state 3
  - Counter must transition to state 0
  - Both flip-flops must transition to logical 0
  - Both D inputs must be logical 0

We conclude

D_A must then be a logical 1

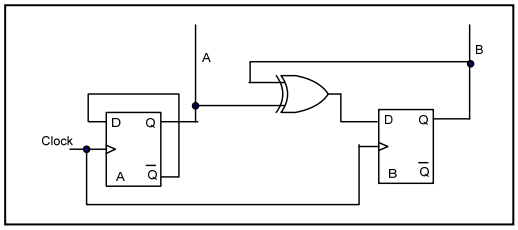
In states 0 and 2

D_B must be a logical 1 in states 1 and 2

Following D input equations result

$$D_A = \overline{A} \bullet \overline{B} + \overline{A} \bullet B$$
$$= \overline{A}$$
$$D_B = \overline{A} \bullet B + A \bullet \overline{B}$$
$$= A \oplus B$$

Logic diagram accompanies



Johnson Counters

Johnson counters are an interesting and useful subset of counters

Find significant utility in designing time bases

For embedded applications as well as for other digital systems

Their design is based upon a classic shift register

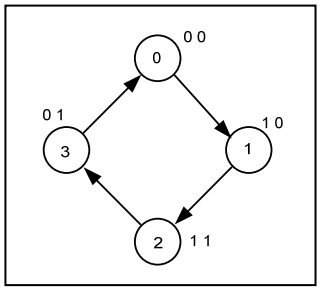
With the Q output of the last stage fed back

As the data input to the first stage

Two Stage Johnson Counter

The two stage Johnson counter

Has the following state diagram and executes the given state table

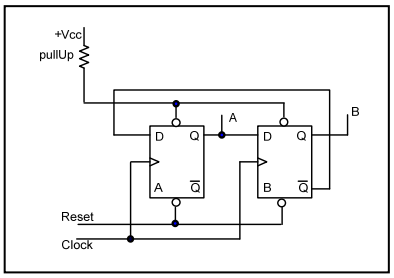


Present State $t = t_n$	Next State $t=t_{n+1}$
A B	A B
0 0 0	1 1 0
1 1 0	2 1 1
2 1 1	3 0 1
3 0 1	0 0 0

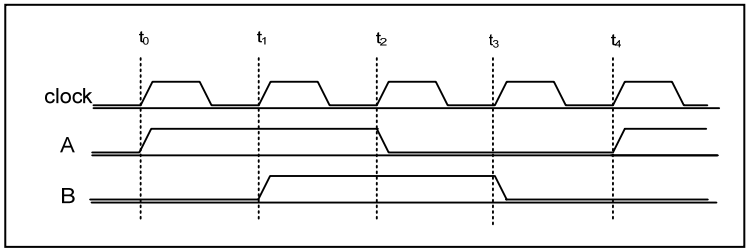
The structural Verilog model for the design

```
module JohnsonCounter(qF1, qF0, clk, por);
  input   clk, por;
  output  qF1, qF0;
  reg     pullUp;

  initial
    pullUp = 1;

//  Build the counter
  DFF f1(qF1, qBarF1, qF0, clk, pullUp, por);
  DFF f0(qF0, qBarF0, qBarF1, clk, pullUp, por);

endmodule
```

Based upon the Verilog code
Logic diagram is given as



Timing diagram given as



Observe the following key points about the two stage Johnson counter

- The states change in a Gray sequence – there is only a single variable change between successive states.
- Because the count sequence is Gray, any state can be decoded, using combinational logic and there will never be any race conditions or hazards (decoding spikes).
- With two state variables, there are 2^2 combinations; all are used in the count sequence.
- The period of the counter is 2^2.

Three or Greater Stage Johnson Counter

Three stage Johnson counter

Has the state diagram and executes following state table

Those with more than three stages simply extend the pattern

State table for the three stage Johnson counter

Has two distinct components

Also seen in the state diagram or graph

Made up of two disconnected subgraphs

Desired state diagram subgraph is given on the left

However if the counter ever enters the second state

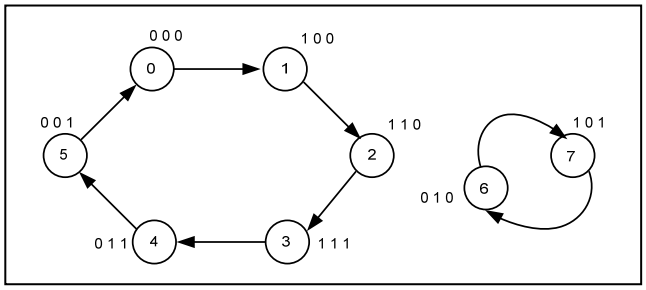
Subgraph shown on the right

Because of noise in the system or some other external causes

It cannot exit

Such a situation is not acceptable

From either a reliability or safety point of view



Present State $t = t_n$	Next State $t=t_{n+1}$
A B C	A B C
0 0 0	1 1 0 0
1 1 0 0	2 1 1 0
2 1 10	3 1 1 1
3 1 1 1	4 0 1 1
4 0 1 1	5 0 0 1
5 0 0 1	0 0 0
6 0 1 0	7 1 0 1
7 1 0 1	6 0 1 0

Problem must be corrected

Such a correction can be implemented

By specifying the inputs to each of the D flip-flops

So as to ensure that the system returns to a valid state

Within the count sequence

Observe the key points about Johnson counters with more than two stages

- The states change in a Gray sequence – there is only a single variable change between successive states.
- Because the count sequence is Gray, any state can be decoded, using combinational logic and there will never be any hazards (decoding spikes).
- With n stages or state variables, there are 2^n combinations; however, not all are used in the count sequence.
- The period of any Johnson counter is 2n; the remaining 2^n -2n states form a disconnected subgraph of illegal states. These must be identified and managed.

Summary

In this lesson we

- ✓ Examined the system datapath and common components
- ✓ Introduced and modeled several commonly used combinational circuits
- ✓ Introduced and modeled several commonly used sequential circuits
- ✓ Explored implementations for the four fundamental arithmetic operations
- ✓ Introduced and modeled the basic types of register type storage elements