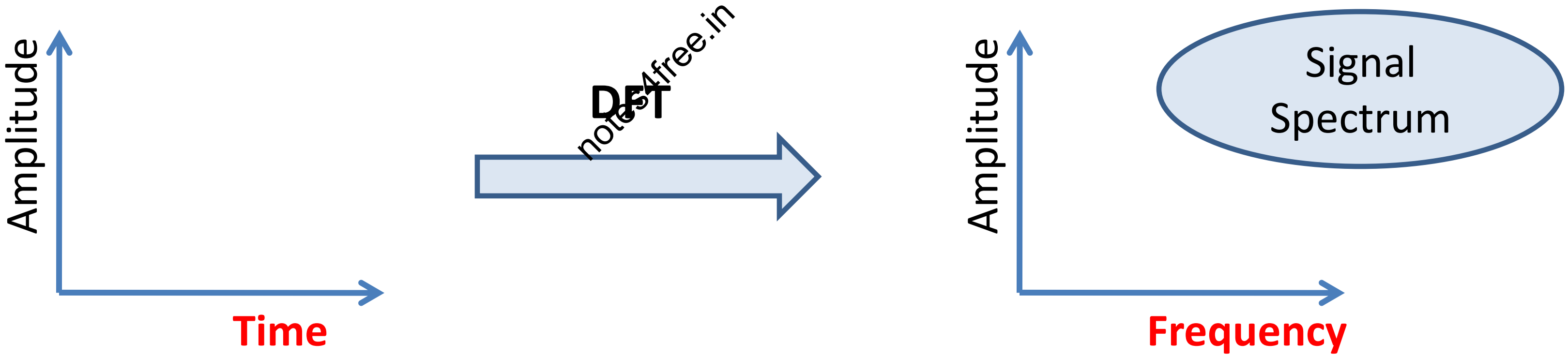
Discrete Fourier Transform (DFT)

DFT transforms the time domain signal samples to the frequency domain components.



Amplitude

Time

DFT

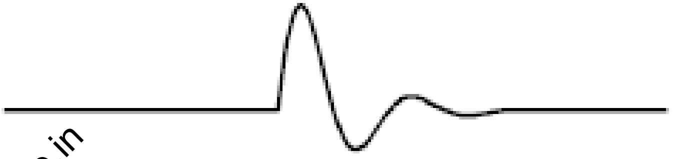
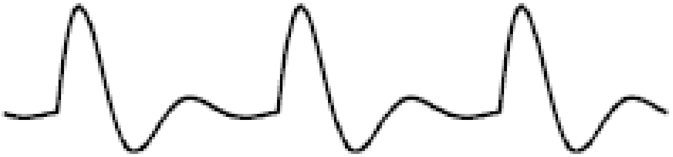
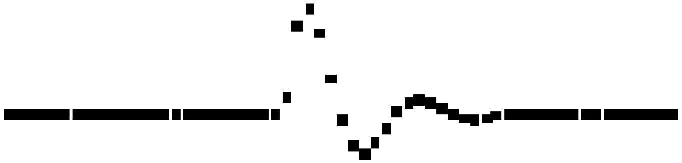
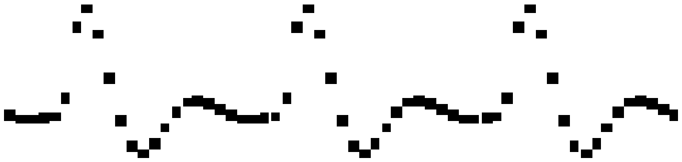
Amplitude

Frequency

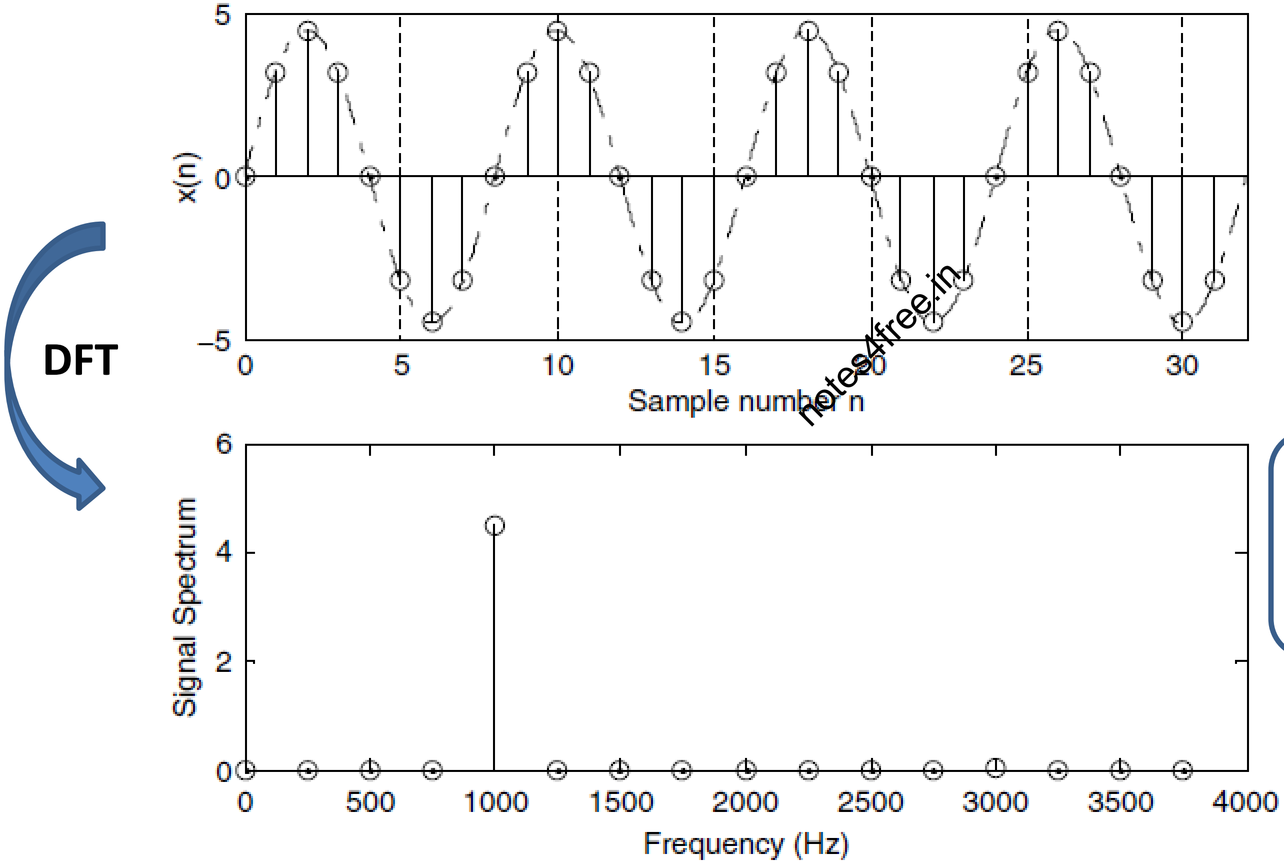
Signal Spectrum

DFT is often used to do frequency analysis of a time domain signal.

Four Types of Fourier Transform

Type of Transform	Example Signal
Fourier Transform *signals that are continuous and aperiodic*	
Fourier Series *signals that are continuous and periodic*	
Discrete Time Fourier Transform *signals that are discrete and aperiodic*	
Discrete Fourier Transform *signals that are discrete and periodic*	

DFT: Graphical Example



1000 Hz sinusoid with 32 samples at 8000 Hz sampling rate.

Sampling rate

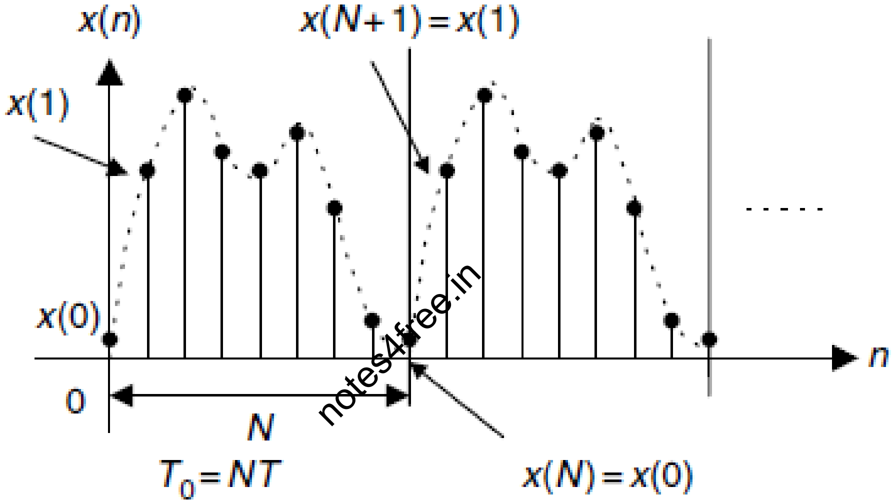
8000 samples = 1 second
32 samples = 32/8000 sec
= 4 millisecond

Frequency

1 second = 1000 cycles
32/8000 sec =
(1000*32/8000=) 4 cycles

DFT Coefficients of Periodic Signals

Periodic
Digital
Signal



Equation of DFT coefficients: $c_k = \frac{1}{N}\sum_{n=0}^{N-1} x(n)e^{-j\frac{2\pi kn}{N}}, \quad -\infty < k < \infty$

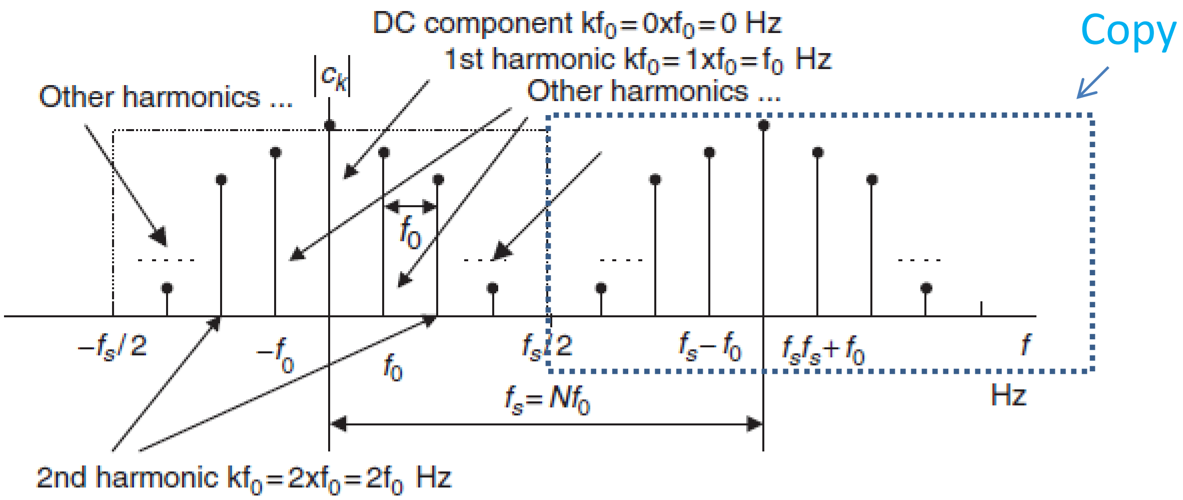
DFT Coefficients of Periodic Signals

Fourier series coefficient c_k is periodic of N

$$c_{k+N} = \frac{1}{N}\sum_{n=0}^{N-1} x(n)e^{-j\frac{2\pi(k+N)n}{N}} = \frac{1}{N}\sum_{n=0}^{N-1} x(n)e^{-j\frac{2\pi kn}{N}}e^{-j2\pi n}$$

Since $e^{-j2\pi n} = \cos(2\pi n) - j\sin(2\pi n) = 1,$ ⇨ $c_{k+N} = c_k.$

Amplitude spectrum of the periodic digital signal ⇨



Example 1

The periodic signal: $\boxed{x(t) = \sin(2\pi t)}$ is sampled at $f_s = 4\,\text{Hz}$

a. Compute the spectrum c_k using the samples in one period.

b. Plot the two-sided amplitude spectrum $|c_k|$ over the range from -2 to 2 Hz.

Solution:

Fundamental frequency

a. We match $x(t) = \sin(2\pi t)$ with $x(t) = \sin(2\pi f t)$ and get ***f* = 1 Hz**.

$\Downarrow$

Therefore the signal has 1 cycle or 1 period in 1 second.

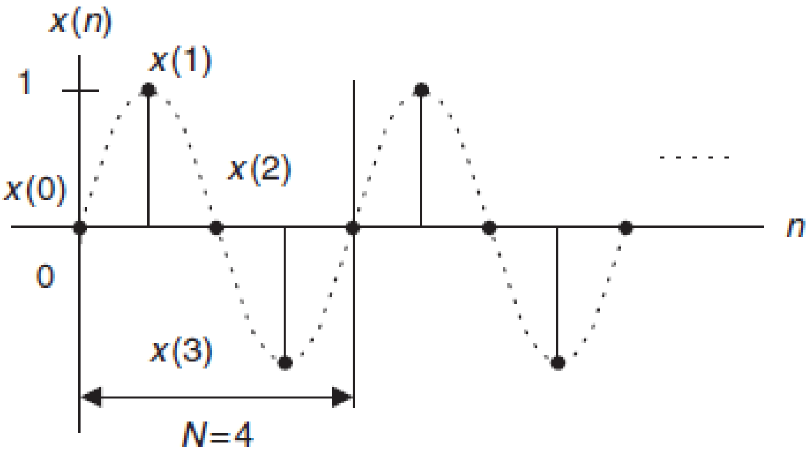
Sampling rate f_s = 4 Hz $\Rightarrow$ 1 second has 4 samples.

Hence, there are 4 samples in 1 period for this particular signal.

$$T = 1/f_s = 0.25 \xrightarrow{\text{Sampled signal}} x(n) = x(nT) = \sin(2\pi nT) = \sin(0.5\pi n)$$

Example 1 – contd. (1)

$$x(0) = 0;\ x(1) = 1;\ x(2) = 0;\ \text{and}\ x(3) = -1$$



b.

$$c_k = \frac{1}{N}\sum_{n=0}^{N-1} x(n)e^{-j\frac{2\pi kn}{N}}, \qquad -\infty < k < \infty$$

$$c_0 = \frac{1}{4}\sum_{n=0}^{3} x(n) = \frac{1}{4}(x(0) + x(1) + x(2) + x(3)) = \frac{1}{4}(0 + 1 + 0 - 1) = 0$$

$$c_1 = \frac{1}{4}\sum_{n=0}^{3} x(n)e^{-j2\pi\times 1n/4} = \frac{1}{4}\left(x(0) + x(1)e^{-j\pi/2} + x(2)e^{-j\pi} + x(3)e^{-j3\pi/2}\right)$$

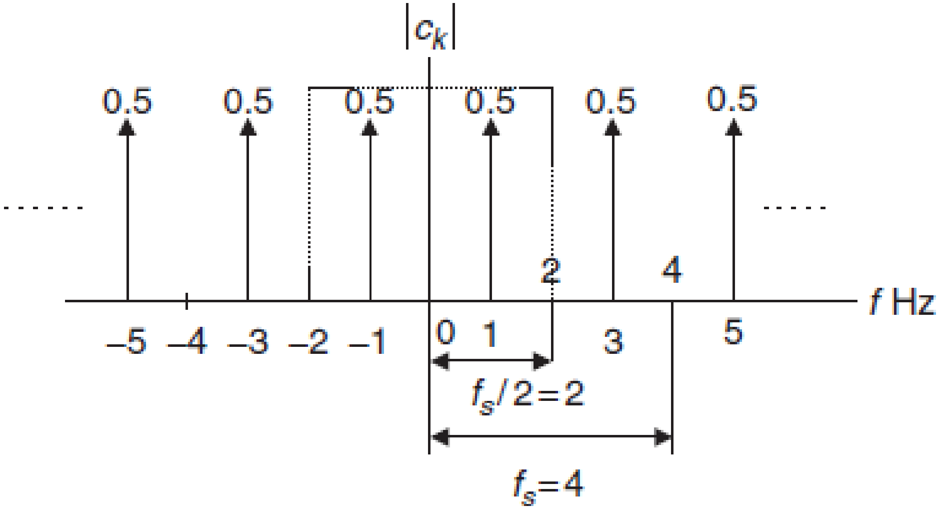
$$= \frac{1}{4}(x(0) - jx(1) - x(2) + jx(3) = 0 - j(1) - 0 + j(-1)) = -j0.5.$$

Example 1 - contd. (2)

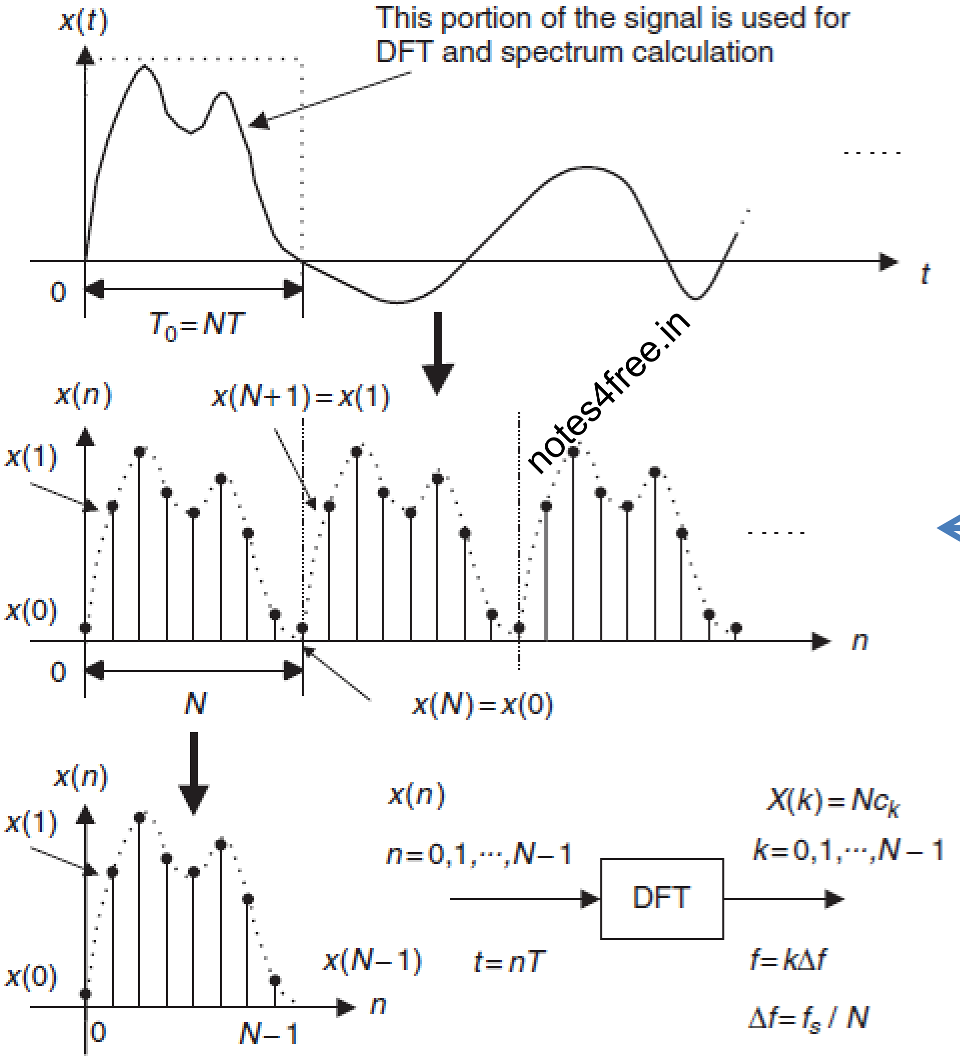
$$c_2 = \frac{1}{4}\sum_{k=0}^{3} x(n)e^{-j2\pi\times 2n/4} = 0, \text{ and } c_3 = \frac{1}{4}\sum_{n=0}^{3} x(k)e^{-j2\pi\times 3n/4} = j0.5.$$

Using periodicity, it follows that

$$c_{-1} = c_3 = j0.5, \text{ and } c_{-2} = c_2 = 0.$$



On the Way to DFT Formulas



Imagine periodicity of *N* samples.

Take first *N* samples (index 0 to *N* -1) as the input to DFT.

DFT Formulas

$$\boxed{X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi kn/N}}$$

$$= \sum_{n=0}^{N-1} x(n) W_N^{kn}, \text{ for } k = 0, 1, \ldots, N-1.$$

$$X(k) = x(0)\, W_N^{k0} + x(1)\, W_N^{k1} + x(2)\, W_N^{k2} + \ldots + x(N-1)\, W_N^{k(N-1)}$$

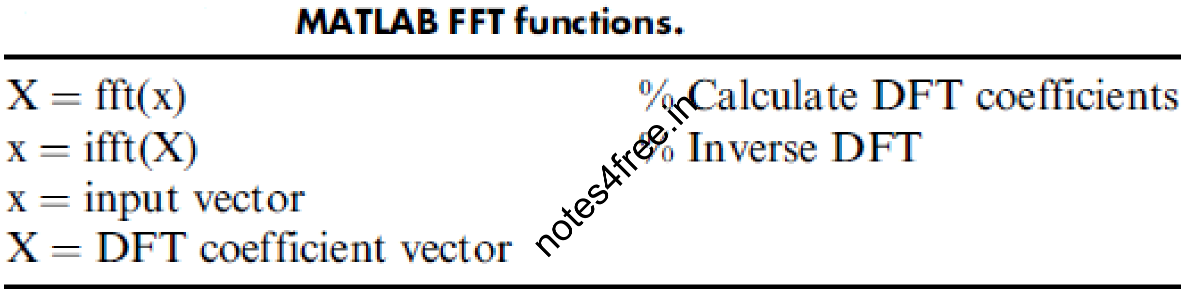
Where, $W_N = e^{-j2\pi/N} = \cos\left(\frac{2\pi}{N}\right) - j\sin\left(\frac{2\pi}{N}\right).$

Inverse DFT:

$$\boxed{x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{j2\pi kn/N}} = \frac{1}{N} \sum_{k=0}^{N-1} X(k) W_N^{-kn}, \text{ for } n = 0,1, \ldots, N-1$$

MATLAB Functions

FFT: Fast Fourier Transform



MATLAB FFT functions.

X = fft(x)	% Calculate DFT coefficients
x = ifft(X)	% Inverse DFT
x = input vector	
X = DFT coefficient vector	

Example 2

Given a sequence $x(n)$ for $0 \le n \le 3$, where $x(0) = 1$, $x(1) = 2$, $x(2) = 3$, and $x(3) = 4$,

a. Evaluate its DFT $X(k)$.

Solution:

$N = 4$ and $W_4 = e^{-j\frac{\pi}{2}}$ ⇒ $X(k) = \sum_{n=0}^{3} x(n) W_4^{kn} = \sum_{n=0}^{3} x(n) e^{-j\frac{\pi k n}{2}}$

Thus, for $k = 0$

$$X(0) = \sum_{n=0}^{3} x(n)e^{-j0} = x(0)e^{-j0} + x(1)e^{-j0} + x(2)e^{-j0} + x(3)e^{-j0}$$

$$= x(0) + x(1) + x(2) + x(3)$$

$$= 1 + 2 + 3 + 4 = 10$$

$$X(1) = \sum_{n=0}^{3} x(n)e^{-j\frac{\pi n}{2}} = x(0)e^{-j0} + x(1)e^{-j\frac{\pi}{2}} + x(2)e^{-j\pi} + x(3)e^{-j\frac{3\pi}{2}}$$

$$= x(0) - jx(1) - x(2) + jx(3)$$

$$= 1 - j2 - 3 + j4 = -2 + j2$$

Example 2 – contd.

$$
\begin{aligned}
X(2) &= \sum_{n=0}^{3} x(n)e^{-j\pi n} = x(0)e^{-j0} + x(1)e^{-j\pi} + x(2)e^{-j2\pi} + x(3)e^{-j3\pi} \\
&= x(0) - x(1) + x(2) - x(3) \\
&= 1 - 2 + 3 - 4 = -2
\end{aligned}
$$

$$
\begin{aligned}
X(3) &= \sum_{n=0}^{3} x(n)e^{-j\frac{3\pi n}{2}} = x(0)e^{-j0} + x(1)e^{-j\frac{3\pi}{2}} + x(2)e^{-j3\pi} + x(3)e^{-j\frac{9\pi}{2}} \\
&= x(0) + jx(1) - x(2) - jx(3) \\
&= 1 + j2 - 3 - j4 = -2 - j2
\end{aligned}
$$

Using MATLAB,

```
≫ X = fft([1  2  3  4])
  X = 10.0000    − 2.0000 + 2.0000i    − 2.0000    − 2.0000 − 2.0000i
```

Example 3

Inverse DFT of the previous example.

$$N = 4 \text{ and } W_4^{-1} = e^{j\frac{\pi}{2}}, \longrightarrow x(n) = \frac{1}{4}\sum_{k=0}^{3} X(k)W_4^{-nk} = \frac{1}{4}\sum_{k=0}^{3} X(k)e^{j\frac{\pi kn}{2}}.$$

$$x(0) = \frac{1}{4}\sum_{k=0}^{3} X(k)e^{j0} = \frac{1}{4}\left(X(0)e^{j0} + X(1)e^{j0} + X(2)e^{j0} + X(3)e^{j0}\right)$$

$$= \frac{1}{4}(10 + (-2 + j2) - 2 + (-2 - j2)) = 1$$

$$x(1) = \frac{1}{4}\sum_{k=0}^{3} X(k)e^{j\frac{k\pi}{2}} = \frac{1}{4}\left(X(0)e^{j0} + X(1)e^{j\frac{\pi}{2}} + X(2)e^{j\pi} + X(3)e^{j\frac{3\pi}{2}}\right)$$

$$= \frac{1}{4}(X(0) + jX(1) - X(2) - jX(3))$$

$$= \frac{1}{4}(10 + j(-2 + j2) - (-2) - j(-2 - j2)) = 2$$

Example 3 – contd.

$$x(2) = \frac{1}{4}\sum_{k=0}^{3} X(k)e^{jk\pi} = \frac{1}{4}\left(X(0)e^{j0} + X(1)e^{j\pi} + X(2)e^{j2\pi} + X(3)e^{j3\pi}\right)$$

$$= \frac{1}{4}(X(0) - X(1) + X(2) - X(3))$$

$$= \frac{1}{4}(10 - (- 2 + j2) + (- 2) - (- 2 - j2)) = 3$$

$$x(3) = \frac{1}{4}\sum_{k=0}^{3} X(k)e^{j\frac{k\pi 3}{2}} = \frac{1}{4}\left(X(0)e^{j0} + X(1)e^{j\frac{3\pi}{2}} + X(2)e^{j3\pi} + X(3)e^{j\frac{9\pi}{2}}\right)$$

$$= \frac{1}{4}(X(0) - jX(1) - X(2) + jX(3))$$

$$= \frac{1}{4}(10 - j(- 2 + j2) - (- 2) + j(- 2 - j2)) = 4$$

Using MATLAB,

$$\gg \mathrm{x} = \mathrm{ifft}([10 \quad - 2 + 2j \quad - 2 \quad - 2 - 2j])$$

$$\mathrm{x} = 1 \qquad 2 \qquad 3 \qquad 4.$$

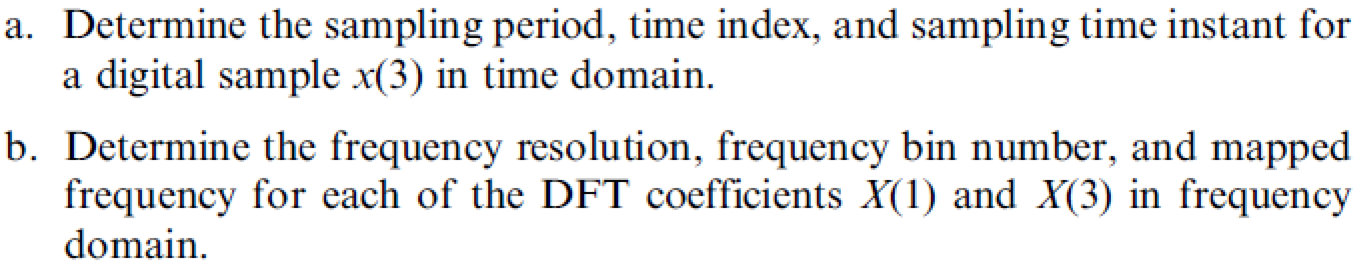
Relationship Between Frequency Bin *k* and Its Associated Frequency in Hz

$$\boxed{f = \frac{kf_s}{N} \ (\text{Hz})}$$

Frequency step or frequency resolution: $\Delta f = \frac{f_s}{N} \ (\text{Hz})$

Example 4

In the previous example, if the sampling rate is 10 Hz,



a. Determine the sampling period, time index, and sampling time instant for a digital sample $x(3)$ in time domain.

b. Determine the frequency resolution, frequency bin number, and mapped frequency for each of the DFT coefficients $X(1)$ and $X(3)$ in frequency domain.

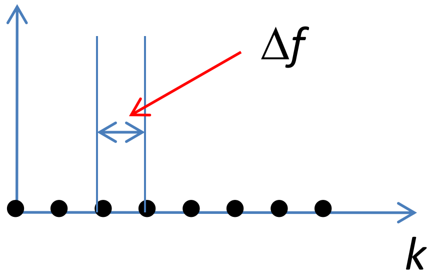
Example 4 - contd.

a.

Sampling period: $T = 1/f_s = 1/10 = 0.1$ second

For x(3), time index is n = 3, and sampling time instant is $t = nT = 3 \cdot 0.1 = 0.3$ second.

b.



Frequency resolution: $\Delta f = \frac{f_s}{N} = \frac{10}{4} = 2.5$ Hz.

Frequency bin number for X(1) is k = 1, and its corresponding frequency is $f = \frac{kf_s}{N} = \frac{1 \times 10}{4} = 2.5$ Hz.

Similarly, for X(3) is k = 3, and its corresponding frequency is $f = \frac{kf_s}{N} = \frac{3 \times 10}{4} = 7.5$ Hz.

Amplitude and Power Spectrum

Since each calculated DFT coefficient is a complex number, it is not convenient to plot it versus its frequency index

Amplitude Spectrum:

$$A_k = \frac{1}{N}|X(k)| = \frac{1}{N}\sqrt{(\text{Real}[X(k)])^2 + (\text{Imag}[X(k)])^2},$$
$$k = 0, 1, 2, \ldots, N-1.$$

To find one-sided amplitude spectrum, we double the amplitude.

$$\bar{A}_k = \begin{cases} \frac{1}{N}|X(0)|, & k = 0 \\ \frac{2}{N}|X(k)|, & k = 1, \ldots, N/2 \end{cases}.$$

Amplitude and Power Spectrum –contd.

Power Spectrum:

$$P_k = \frac{1}{N^2}|X(k)|^2 = \frac{1}{N^2}\left\{(\mathrm{Real}[X(k)])^2 + (\mathrm{Imag}[X(k)])^2\right\},$$
$$k = 0,\ 1,\ 2, \ldots,\ N-1.$$

For, one-sided power spectrum:

$$\bar{P}_k = \begin{cases} \frac{1}{N^2}|X(0)|^2 & k = 0 \\ \frac{2}{N^2}|X(k)|^2 & k = 0,\ 1, \ldots,\ N/2 \end{cases}$$

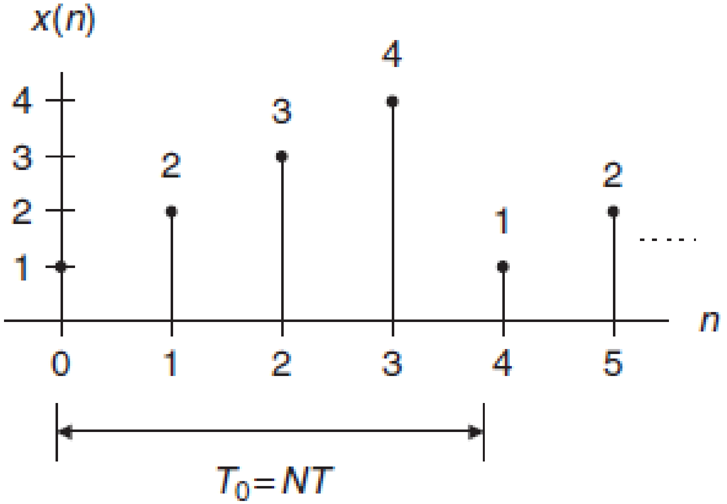
Phase Spectrum:

$$\varphi_k = \tan^{-1}\left(\frac{\mathrm{Imag}[X(k)]}{\mathrm{Real}[X(k)]}\right),\ k = 0,\ 1,\ 2, \ldots,\ N-1.$$

Example 5

Assuming that $f_s = 100\,\text{Hz}$,

a. Compute the amplitude spectrum, phase spectrum, and power spectrum.



Solution:

$$X(0) = 10$$
$$X(1) = -2 + j2$$
$$X(2) = -2$$
$$X(3) = -2 - j2.$$

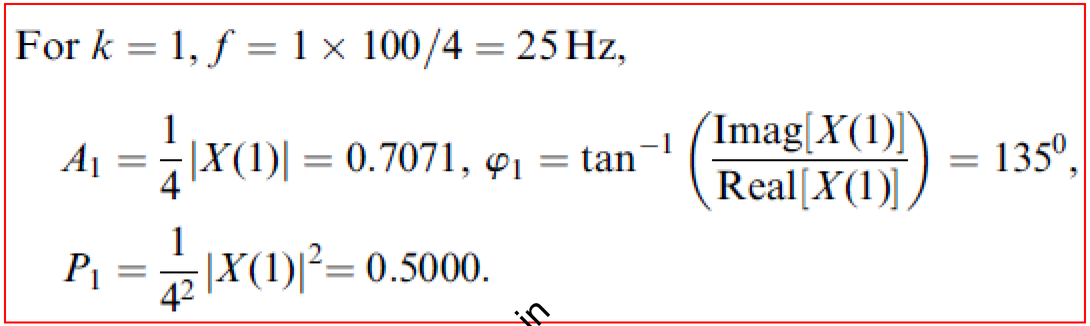
See Example 2.

For $k = 0$, $f = k \cdot f_s/N = 0 \times 100/4 = 0\,\text{Hz}$,

$$A_0 = \frac{1}{4}|X(0)| = 2.5,\ \varphi_0 = \tan^{-1}\left(\frac{\text{Imag}[X(0)]}{\text{Real}([X(0)]}\right) = 0^0,$$

$$P_0 = \frac{1}{4^2}|X(0)|^2 = 6.25.$$

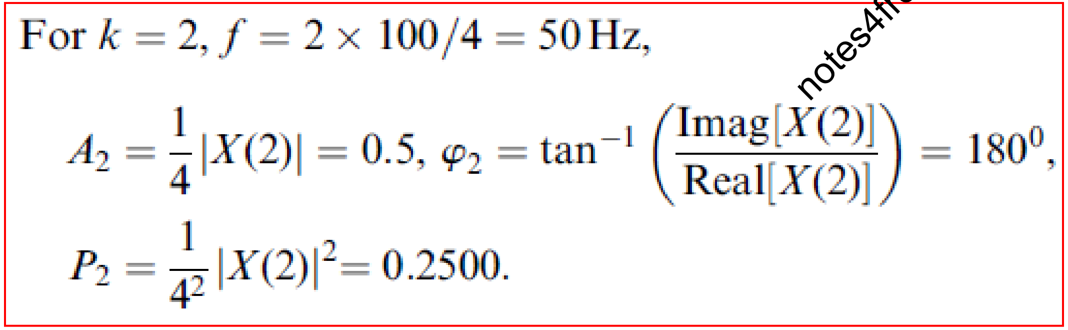
Example 5 - contd. (1)



For $k = 1$, $f = 1 \times 100/4 = 25\,\text{Hz}$,

$$A_1 = \frac{1}{4}|X(1)| = 0.7071,\ \varphi_1 = \tan^{-1}\left(\frac{\text{Imag}[X(1)]}{\text{Real}[X(1)]}\right) = 135^0,$$

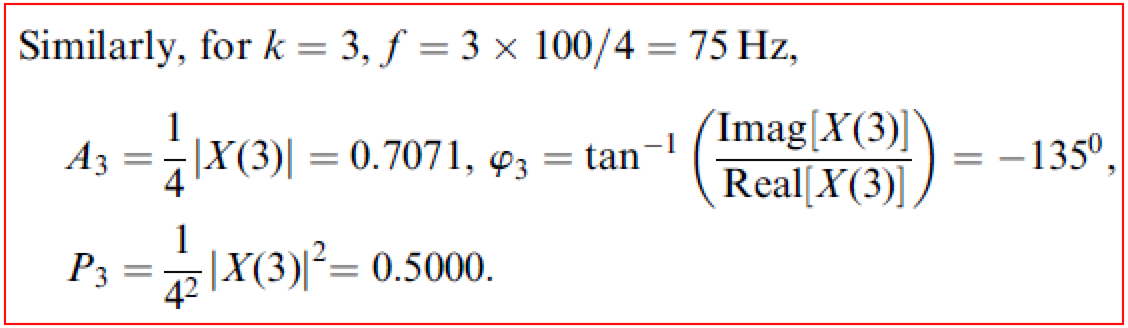
$$P_1 = \frac{1}{4^2}|X(1)|^2 = 0.5000.$$



For $k = 2$, $f = 2 \times 100/4 = 50\,\text{Hz}$,

$$A_2 = \frac{1}{4}|X(2)| = 0.5,\ \varphi_2 = \tan^{-1}\left(\frac{\text{Imag}[X(2)]}{\text{Real}[X(2)]}\right) = 180^0,$$

$$P_2 = \frac{1}{4^2}|X(2)|^2 = 0.2500.$$

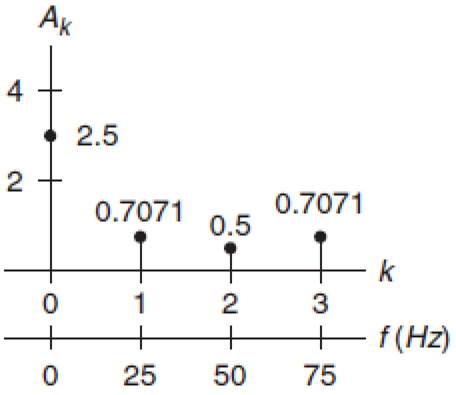


Similarly, for $k = 3$, $f = 3 \times 100/4 = 75\,\text{Hz}$,

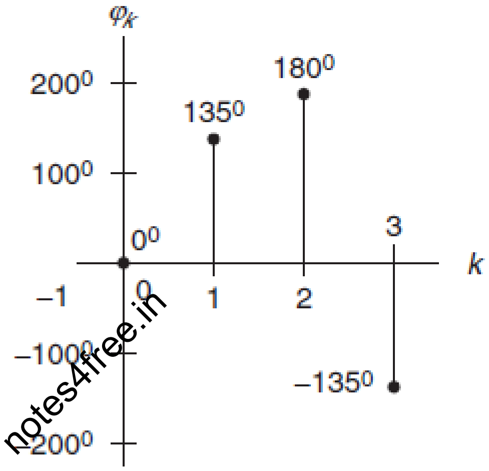
$$A_3 = \frac{1}{4}|X(3)| = 0.7071,\ \varphi_3 = \tan^{-1}\left(\frac{\text{Imag}[X(3)]}{\text{Real}[X(3)]}\right) = -135^0,$$

$$P_3 = \frac{1}{4^2}|X(3)|^2 = 0.5000.$$

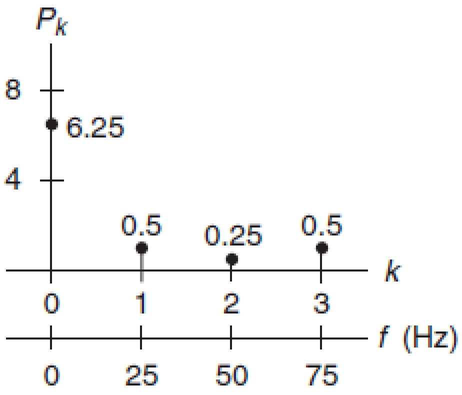
Example 5 – contd. (2)



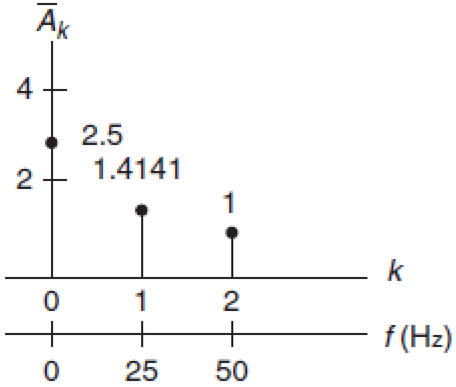
Amplitude Spectrum



Phase Spectrum



Power Spectrum



One sided Amplitude Spectrum

Example 6

Consider a digital sequence sampled at the rate of 10 kHz. If we use a size of 1,024 data points and apply the 1,024-point DFT to compute the spectrum,

a. Determine the frequency resolution.

b. Determine the highest frequency in the spectrum.

Solution:

a. $\Delta f = \frac{f_s}{N} = \frac{10000}{1024} = 9.776 \text{ Hz}.$

b. The highest frequency is the folding frequency, given by

$$f_{\max} = \frac{N}{2}\Delta f = \frac{f_s}{2}$$

$$= 512 \cdot 9.776 = 5000 \text{ Hz}$$

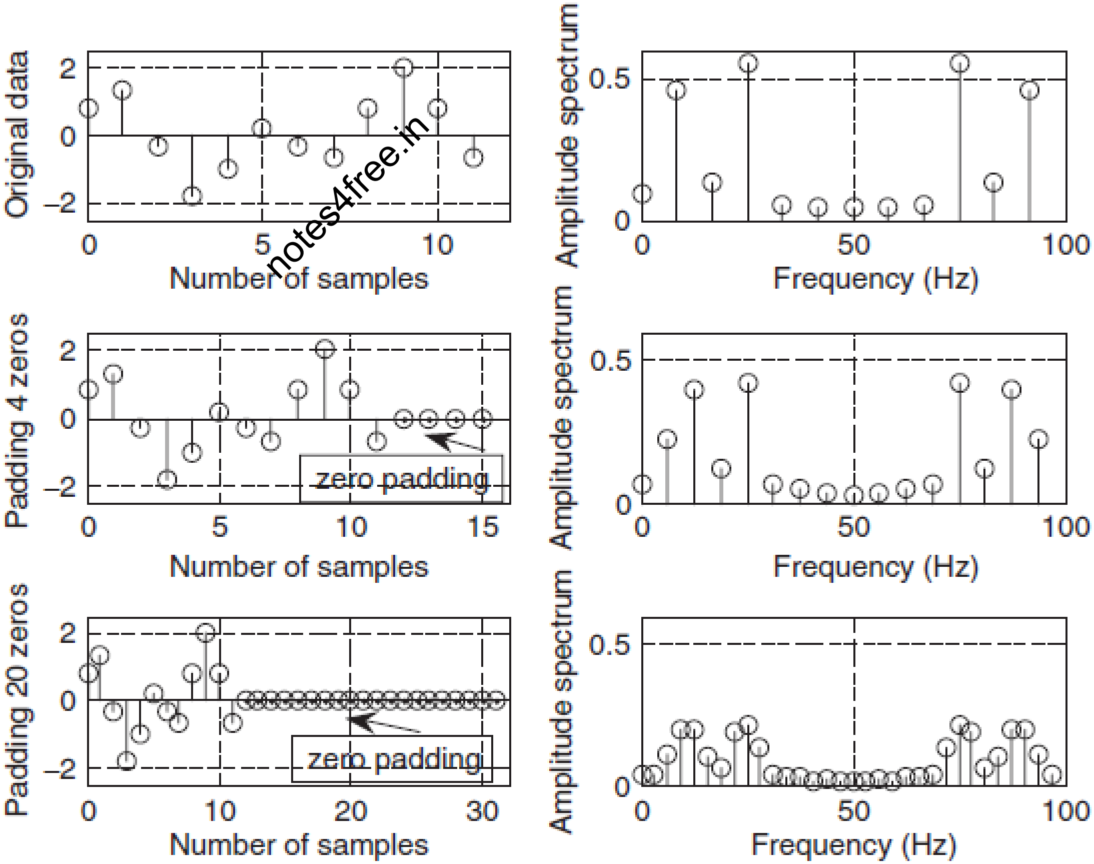
Zero Padding for FFT

FFT: Fast Fourier Transform.

→ A fast version of DFT; It requires signal length to be power of 2.

Therefore, we need to pad zero at the end of the signal.

However, it does not add any new information.



Example 7

Consider a digital signal has sampling rate = 10 kHz. For amplitude spectrum we need frequency resolution of less than 0.5 Hz. For FFT how many data points are needed?

Solution:

$$\Delta f = 0.5\,\text{Hz}$$

$$N = \frac{f_s}{\Delta f} = \frac{10000}{0.5} = 20000$$

For FFT, we need *N* to be power of 2.

$2^{14} = 16384 < 20000$ And $\boxed{2^{15} = 32768 > 20000}$

Recalculated frequency resolution,

$$\Delta f = \frac{f_s}{N} = \frac{10000}{32768\cdot} = 0.31\,\text{Hz}.$$

MATLAB Example - 1

$$x(n) = 2 \cdot \sin\left(2000\pi \frac{n}{8000}\right)$$

f_s

Use the MATLAB DFT to compute the signal spectrum with the frequency resolution to be equal to or less than 8 Hz.

$$N = \frac{f_s}{\Delta f} = \frac{8000}{8} = 1000$$

```
% Generate the sine wave sequence
fs = 8000;              %Sampling rate
N = 1000;               % Number of data points
x = 2* sin (2000* pi* [0:1:N − 1]/fs);
```

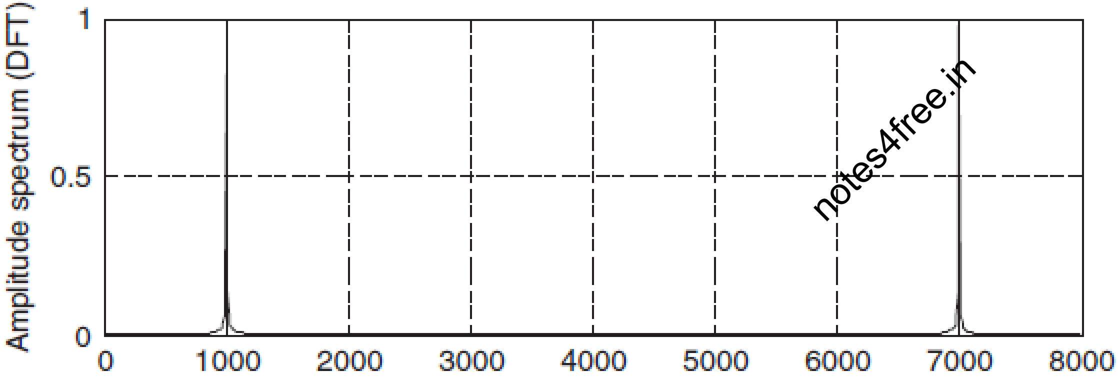
```
figure(1), plot(x);
```

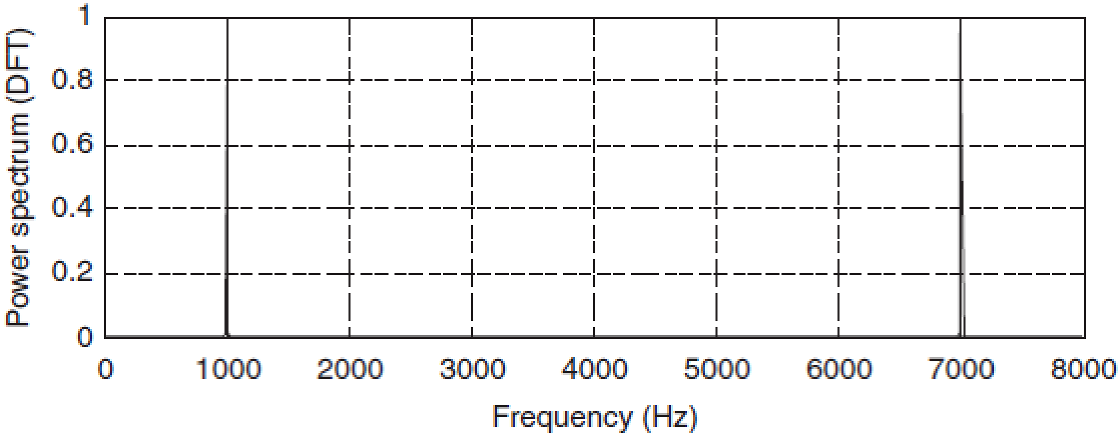
```
xf = abs(fft(x))/N;    %Compute the amplitude spectrum
```

```
P = xf.*xf;             %Compute the power spectrum
f = [0:1:N − 1]*fs/N;           %Map the frequency bin to the frequency (Hz)
```

MATLAB Example – contd. (1)

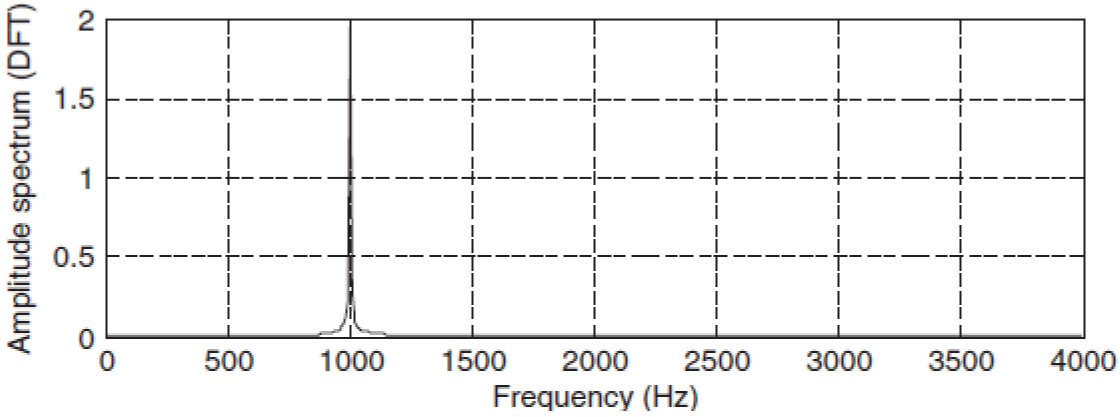
```
subplot(2,1,1); plot(f,xf);grid
xlabel('Frequency (Hz)'); ylabel('Amplitude spectrum (DFT)');
subplot(2,1,2);plot(f,P);grid
xlabel('Frequency (Hz)'); ylabel('Power spectrum (DFT)');
```



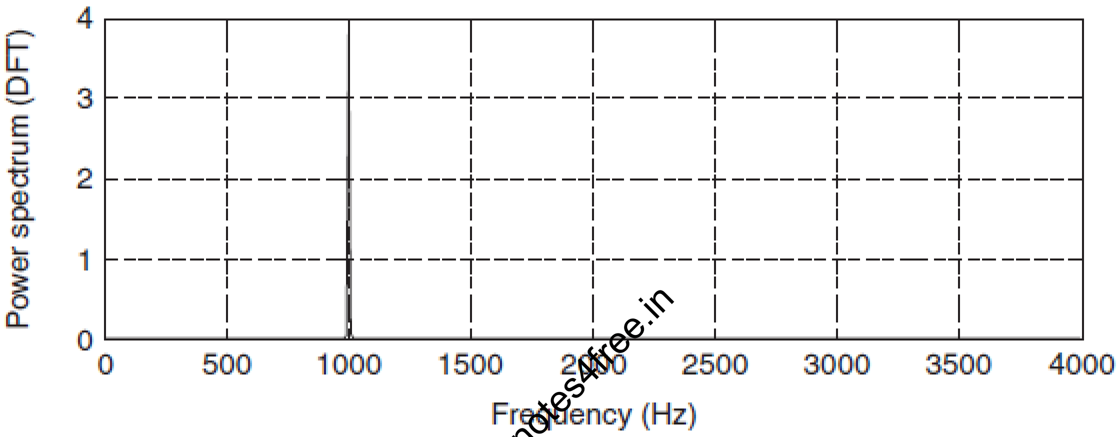


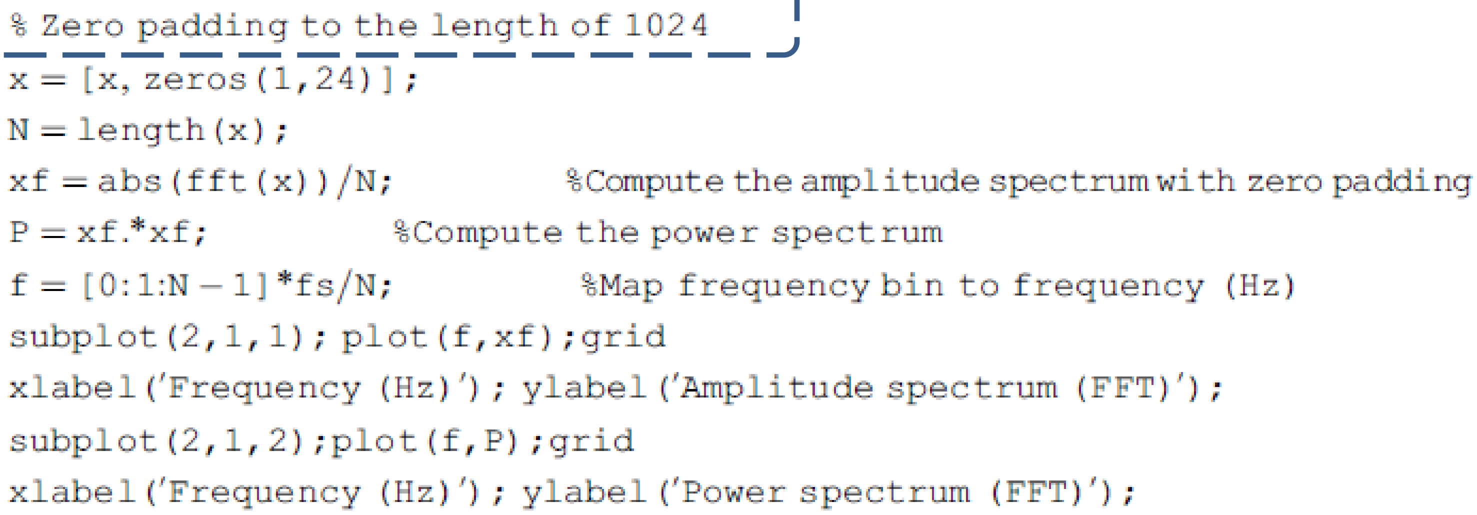
MATLAB Example – contd. (2)

```
% Convert it to one-sided spectrum
xf(2:N) = 2*xf(2:N);          % Get the single-sided spectrum
P = xf.*xf; % Calculate the power spectrum
f = [0:1:N/2]*fs/N % Frequencies up to the folding frequency
subplot(2,1,1); plot(f,xf(1:N/2 + 1));grid
xlabel('Frequency (Hz)'); ylabel('Amplitude spectrum (DFT)');
subplot(2,1,2);plot(f,P(1:N/2 + 1));grid
xlabel('Frequency (Hz)'); ylabel('Power spectrum (DFT)');
```



MATLAB Example – contd. (3)





```
% Zero padding to the length of 1024
x = [x, zeros(1,24)];
N = length(x);
xf = abs(fft(x))/N;          %Compute the amplitude spectrum with zero padding
P = xf.*xf;          %Compute the power spectrum
f = [0:1:N – 1]*fs/N;          %Map frequency bin to frequency (Hz)
subplot(2,1,1); plot(f,xf);grid
xlabel('Frequency (Hz)'); ylabel('Amplitude spectrum (FFT)');
subplot(2,1,2);plot(f,P);grid
xlabel('Frequency (Hz)'); ylabel('Power spectrum (FFT)');
```

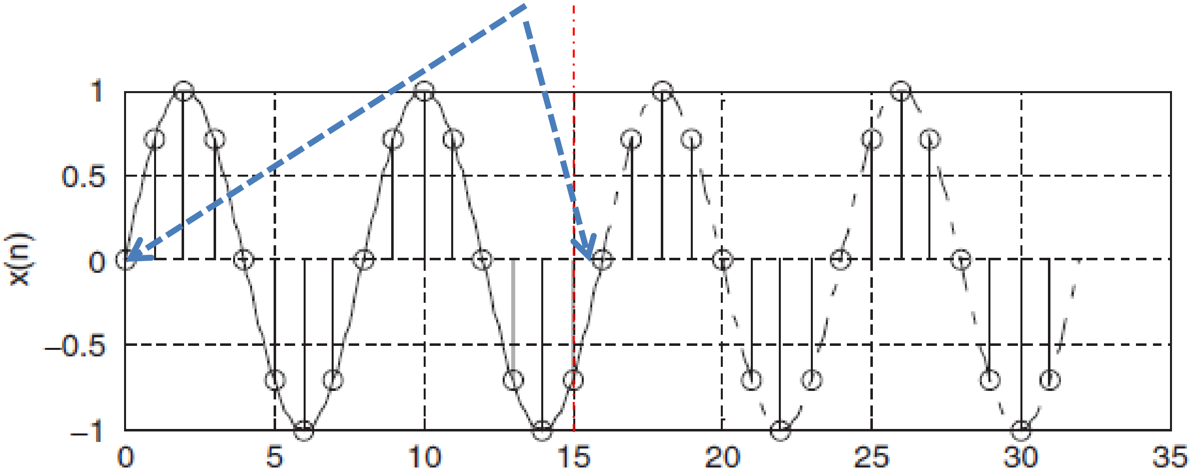
Effect of Window Size

When applying DFT, we assume the following:

1. Sampled data are periodic to themselves (repeat).

2. Sampled data are continuous to themselves and band limited to the folding frequency.

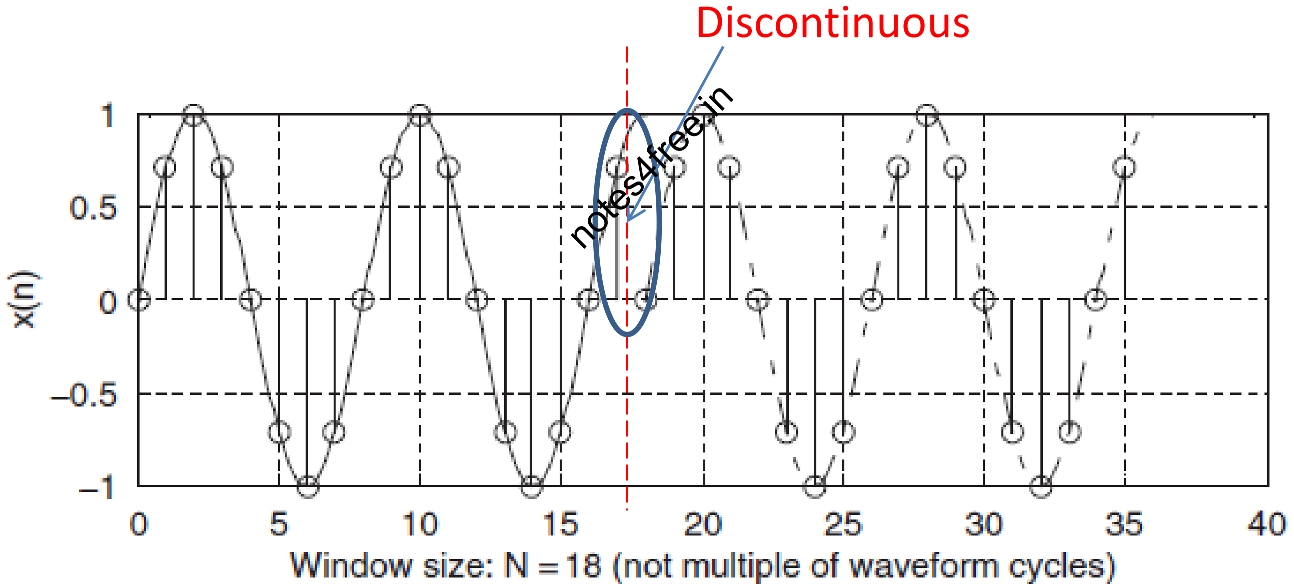
1 Hz sinusoid, with 32 samples



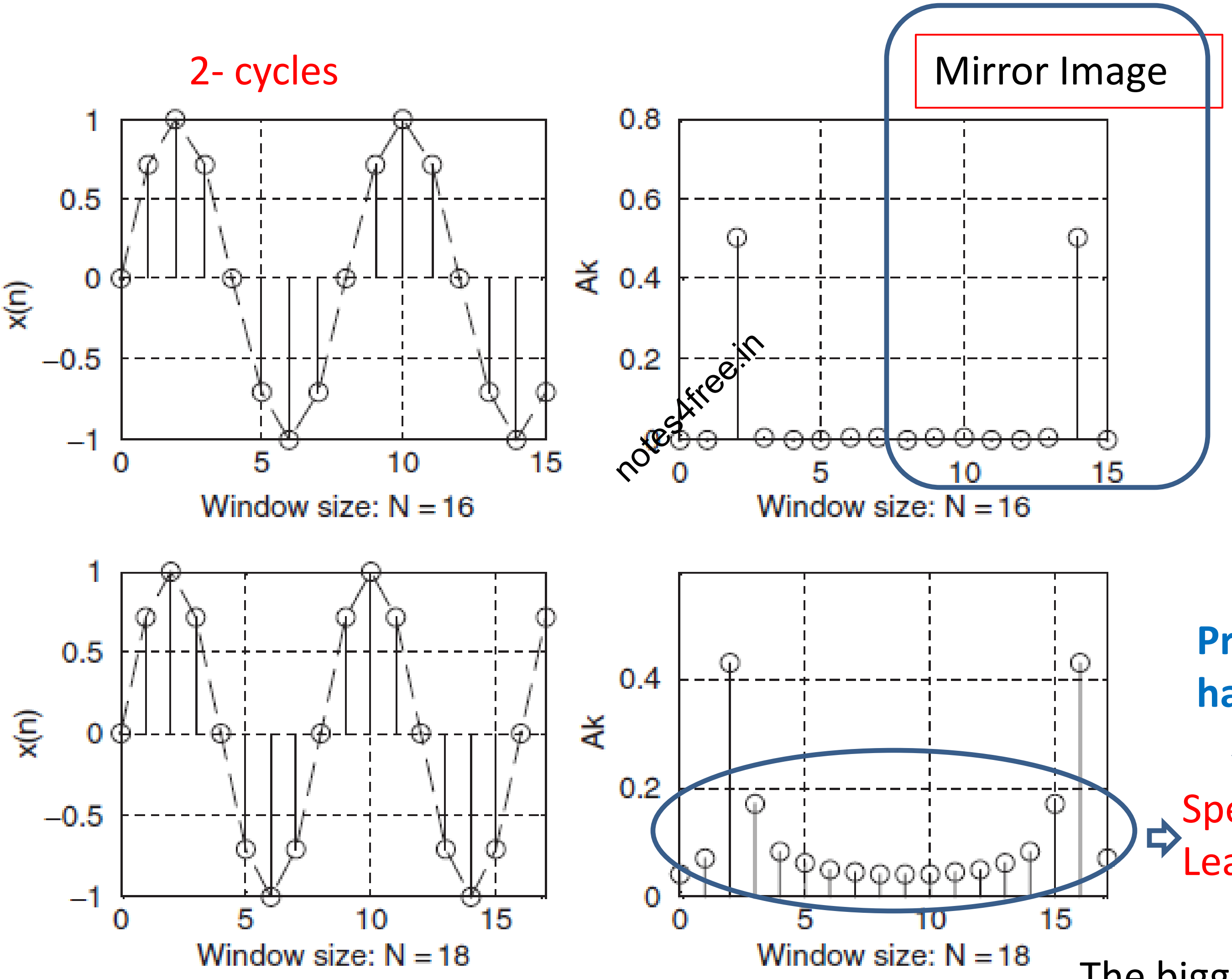
Window size: N = 16 (multiple of waveform cycles)

Effect of Window Size –contd. (1)

If the window size is not multiple of waveform cycles:



Effect of Window Size -contd. (2)



2- cycles

Mirror Image

Produces single frequency

Produces many harmonics as well.

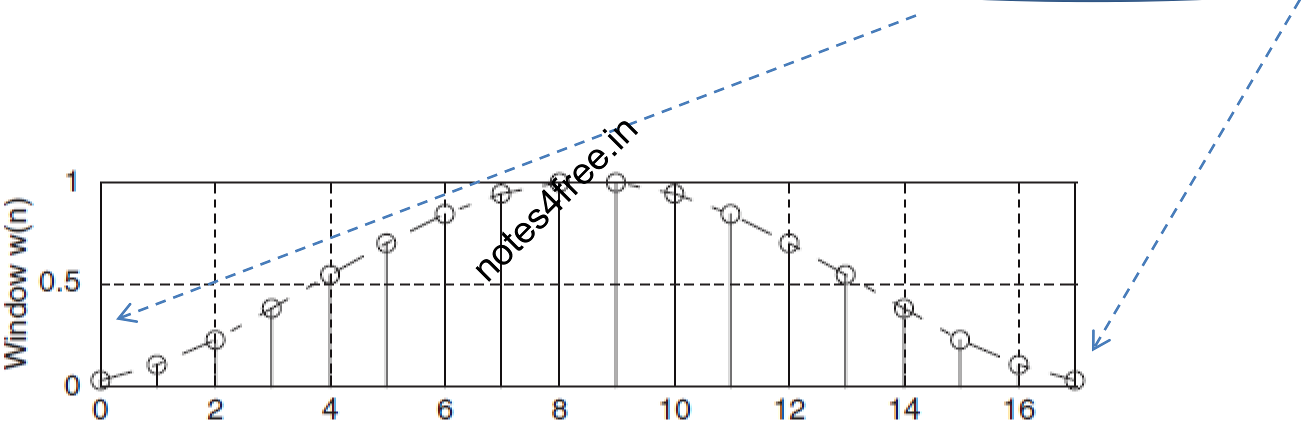
⇩

⇨ Spectral Leakage

The bigger the discontinuity, the more the leakage

Reducing Leakage Using Window

To reduce the effect of spectral leakage, a window function can be used whose amplitude tapers smoothly and gradually toward zero at both ends.



$$x_w(n) = x(n)w(n), \text{ for } n = 0, 1, \ldots, N - 1.$$

Window function, *w*(*n*)
Data sequence, *x*(*n*)
Obtained windowed sequence, $x_w(n)$

Example 8

Given,

$x(2) = 1$ and $w(2) = 0.2265;$

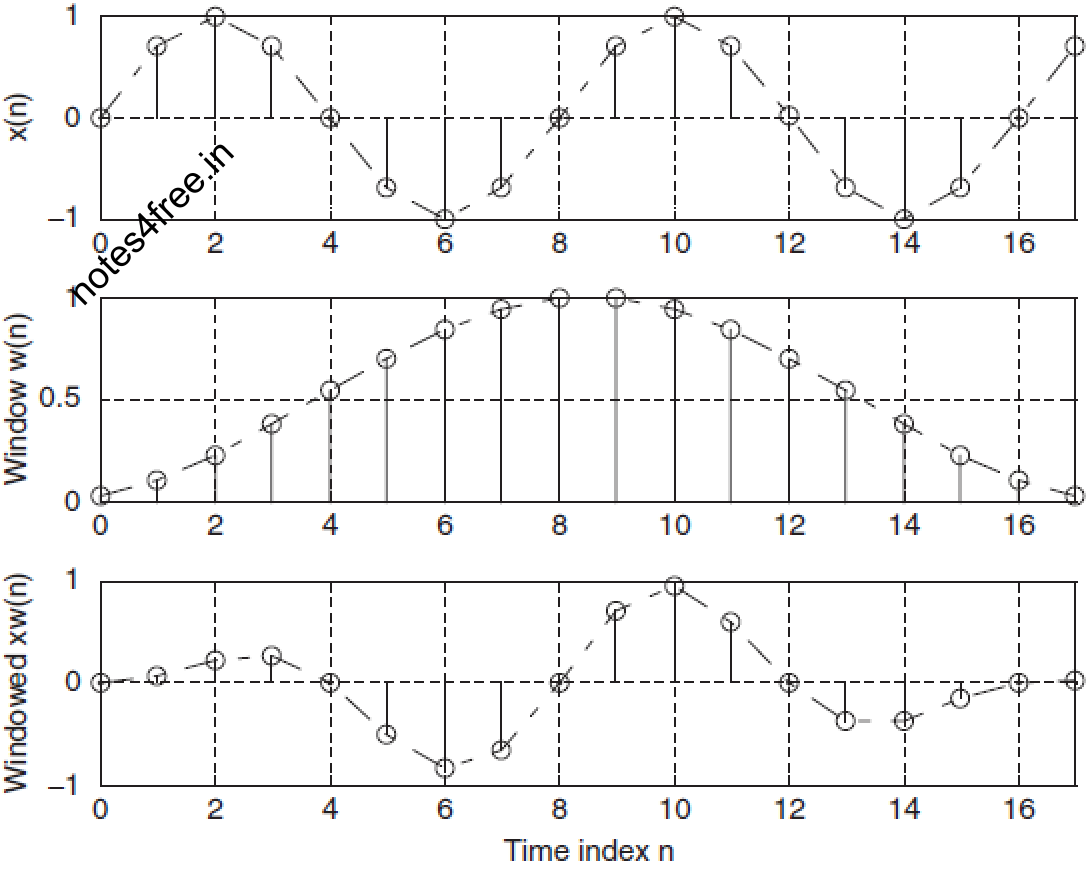
$x(5) = -0.7071$ and $w(5) = 0.7008,$

Calculate,

$x_w(2)$ and $x_w(5)$.

$$x_w(2) = x(2) \times w(2)$$
$$= 1 \times 0.2265 = 0.2265$$

$$x_w(5) = x(5) \times w(5)$$
$$= -0.7071 \times 0.7008 = -0.4956$$



Different Types of Windows

Rectangular Window (no window): $w_R(n) = 1 \qquad 0 \le n \le N-1$

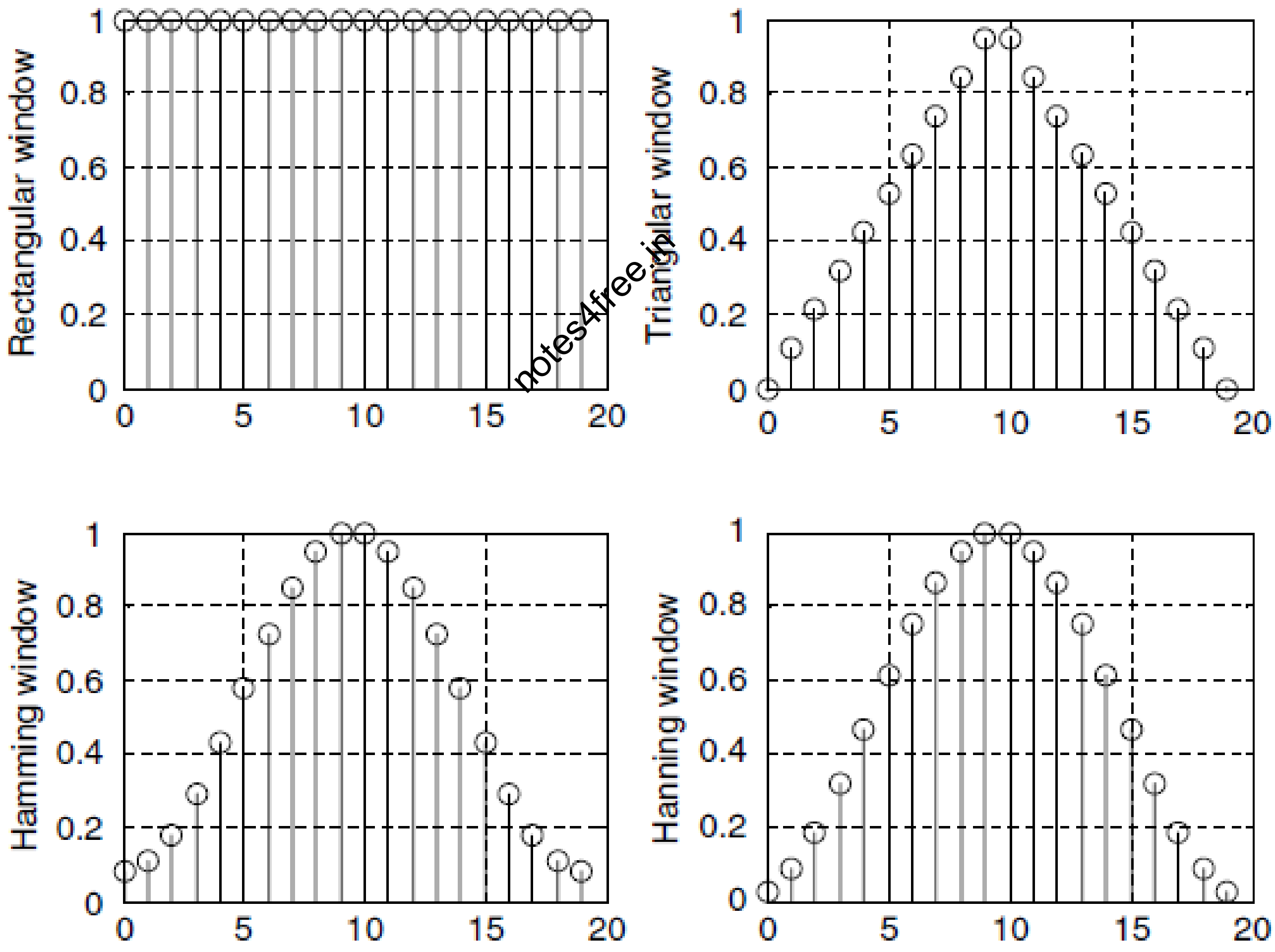
Triangular Window: $w_{tri}(n) = 1 - \dfrac{|2n - N + 1|}{N-1},\ 0 \le n \le N-1$

Hamming Window: $w_{hm}(n) = 0.54 - 0.46\cos\left(\dfrac{2\pi n}{N-1}\right),\ 0 \le n \le N-1$

Hanning Window: $w_{hn}(n) = 0.5 - 0.5\cos\left(\dfrac{2\pi n}{N-1}\right),\ 0 \le n \le N-1$

Different Types of Windows –contd.

Window size of 20 samples



Example 9

Problem:

Considering the sequence $x(0) = 1$, $x(1) = 2$, $x(2) = 3$, and $x(3) = 4$, and given $f_s = 100$ Hz, $T = 0.01$ seconds, compute the amplitude spectrum, phase spectrum, and power spectrum

Using the Hamming window function.

Solution:

Since N = 4, Hamming window function can be found as:

$$w_{hm}(0) = 0.54 - 0.46\cos\left(\frac{2\pi \times 0}{4-1}\right) = 0.08$$

$$w_{hm}(1) = 0.54 - 0.46\cos\left(\frac{2\pi \times 1}{4-1}\right) = 0.77.$$

Similarly, $w_{hm}(2) = 0.77$, $w_{hm}(3) = 0.08$.

Example 9 - contd. (1)

Windowed sequence:

$$x_w(0) = x(0) \times w_{hm}(0) = 1 \times 0.08 = 0.08$$

$$x_w(1) = x(1) \times w_{hm}(1) = 2 \times 0.77 = 1.54$$

$$x_w(2) = x(2) \times w_{hm}(2) = 3 \times 0.77 = 2.31$$

$$x_w(0) = x(3) \times w_{hm}(3) = 4 \times 0.08 = 0.32.$$

DFT Sequence:

$$X(k) = x(0)\, W_N^{k0} + x(1)\, W_N^{k1} + x(2) W_N^{k2} + \ldots + x(N-1) W_N^{k(N-1)}$$

$$\Rightarrow \quad X(k) = x_w(0) W_4^{k\times 0} + x(1) W_4^{k\times 1} + x(2) W_4^{k\times 2} + x(3) W_4^{k\times 3}.$$

$$\Rightarrow \quad \begin{cases} X(0) = 4.25 \\ X(1) = -2.23 - j1.22 \\ X(2) = 0.53 \\ X(3) = -2.23 + j1.22 \end{cases}$$

$$\Delta f = \frac{1}{NT} = \frac{1}{4 \cdot 0.01} = 25\,\text{Hz}$$

Example 9 – contd. (2)

$$A_0 = \frac{1}{4}|X(0)| = 1.0625,\ \varphi_0 = \tan^{-1}\left(\frac{0}{4.25}\right) = 0^0,$$

$$P_0 = \frac{1}{4^2}|X(0)|^2 = 1.1289$$

$$A_1 = \frac{1}{4}|X(1)| = 0.6355,\ \varphi_1 = \tan^{-1}\left(\frac{-1.22}{-2.23}\right) = -151.32^0,$$

$$P_1 = \frac{1}{4^2}|X(1)|^2 = 0.4308$$

$$A_2 = \frac{1}{4}|X(2)| = 0.1325,\ \varphi_2 = \tan^{-1}\left(\frac{0}{0.53}\right) = 0^0,$$

$$P_2 = \frac{1}{4^2}|X(2)|^2 = 0.0176.$$

$$A_3 = \frac{1}{4}|X(3)| = 0.6355,\ \varphi_3 = \tan^{-1}\left(\frac{1.22}{-2.23}\right) = 151.32^0,$$

$$P_3 = \frac{1}{4^2}|X(3)|^2 = 0.4308.$$

MATLAB Example - 2

$$x(n) = 2 \cdot \sin\left(2000\pi \frac{n}{8000}\right)$$

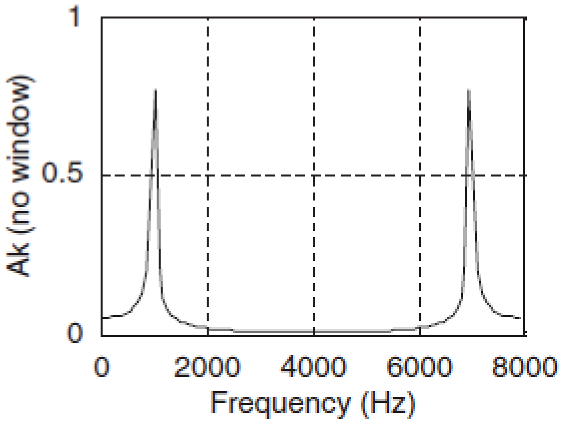
Compute the spectrum of a Hamming window function with a window size $= 100$.

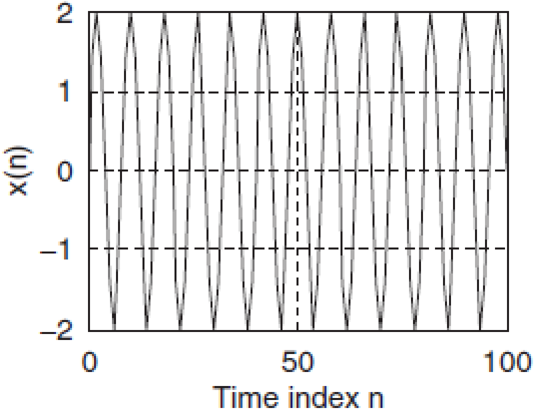
```
% Generate the sine wave sequence
fs = 8000; T = 1/fs;            % Sampling rate and sampling period


% Generate the sine wave sequence
x = 2* sin (2000*pi*[0:1:100]*T);
% Apply the FFT algorithm
N=length(x);
index_t = [0:1:N − 1];
f = [0:1:N − 1]*fs/N;
xf = abs(fft(x))/N;
```

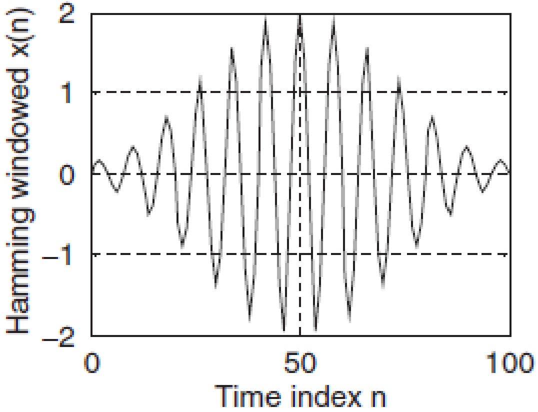
```
%Using the Hamming window
x_hm = x.*hamming(N)';            %Apply the Hamming window function
xf_hm=abs(fft(x_hm))/N;              %Calculate the amplitude spectrum
```

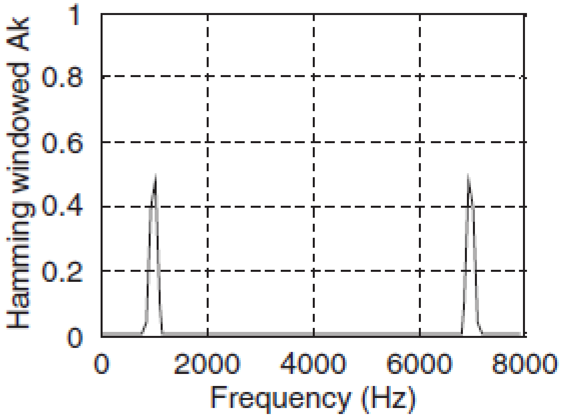
MATLAB Example – 2 contd.

```
subplot(2,2,1);plot(index_t,x);grid
xlabel('Time index n'); ylabel('x(n)');
subplot(2,2,3); plot(index_t,x_hm);grid
xlabel('Time index n'); ylabel('Hamming windowed x(n)');
subplot(2,2,2);plot(f,xf);grid;axis([0 fs 0 1]);
xlabel('Frequency (Hz)'); ylabel('Ak (no window)');
subplot(2,2,4); plot(f,xf_hm);grid;axis([0 fs 0 1]);
xlabel('Frequency (Hz)'); ylabel('Hamming windowed Ak');
```









DFT Matrix

Frequency Spectrum

Multiplication Matrix

Time-Domain samples

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ \vdots \\ X(N-2) \\ X(N-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & & 1 & 1 \\ 1 & e^{-j\frac{2\pi}{N}} & e^{-j\frac{4\pi}{N}} & \cdots & e^{-j\frac{2(N-2)\pi}{N}} & e^{-j\frac{2(N-1)\pi}{N}} \\ 1 & e^{-j\frac{4\pi}{N}} & e^{-j\frac{8\pi}{N}} & \cdots & e^{-j\frac{4(N-2)\pi}{N}} & e^{-j\frac{4(N-2)\pi}{N}} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & e^{-j\frac{2(N-2)\pi}{N}} & e^{-j\frac{4(N-2)\pi}{N}} & \cdots & e^{-j\frac{2(N-2)^2\pi}{N}} & e^{-j\frac{2(N-2)(N-1)\pi}{N}} \\ 1 & e^{-j\frac{2(N-1)\pi}{N}} & e^{-j\frac{4(N-1)\pi}{N}} & \cdots & e^{-j\frac{2(N-1)(N-2)\pi}{N}} & e^{-j\frac{(N-1)^2\pi}{N}} \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ \vdots \\ x(N-2) \\ x(N-1) \end{bmatrix}$$

DFT Matrix

Let, $w_N = e^{-j2\pi/N}$

Then

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ \vdots \\ X(N-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & w & w^2 & \cdots & w^{(N-1)} \\ 1 & w^2 & w^4 & \cdots & w^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w^{(N-1)} & w^{2(N-1)} & \cdots & w^{(N-1)^2} \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ \vdots \\ x(N-1) \end{bmatrix}$$

DFT equation: $$X(k) = \sum_{m=0}^{N-1} x(m) w_N^{mk} \qquad k = 0, \ldots, N-1$$

DFT requires N^2 complex multiplications.

FFT

FFT: Fast Fourier Transform

A very efficient algorithm to compute DFT; it requires less multiplication.

The length of input signal, x(*n*) must be 2^m samples, where *m* is an integer.

⇩

Samples *N* = 2, 4, 8, 16 or so.

If the input length is not 2^m, append (pad) zeros to make it 2^m.

4	5	1	7	1

N = 5

⇨

4	5	1	7	1	0	0	0

N = 8, power of 2

DFT to FFT: Decimation in Frequency

DFT: $$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} \text{ for } k = 0, 1, \ldots, N-1,$$

$$X(k) = x(0) + x(1)W_N^k + \ldots + x(N-1)W_N^{k(N-1)}$$

$$X(k) = \boxed{x(0) + x(1)W_N^k + \ldots + x\left(\frac{N}{2}-1\right)W_N^{k(N/2-1)}} + \boxed{x\left(\frac{N}{2}\right)W^{kN/2} + \ldots + x(N-1)W_N^{k(N-1)}}$$

$$X(k) = \sum_{n=0}^{(N/2)-1} x(n)W_N^{kn} + \boxed{\sum_{n=N/2}^{N-1} x(n)W_N^{kn}}$$

$$\Downarrow$$

$$X(k) = \sum_{n=0}^{(N/2)-1} x(n)W_N^{kn} + W_N^{(N/2)k} \sum_{n=0}^{(N/2)-1} x\left(n+\frac{N}{2}\right)W_N^{kn}.$$

$$W_N^{N/2} = e^{-j\frac{2\pi(N/2)}{N}} = e^{-j\pi} = -1;$$

$$X(k) = \sum_{n=0}^{(N/2)-1} \left(x(n) + (-1)^k x\left(n+\frac{N}{2}\right)\right)W_N^{kn}$$

DFT to FFT: Decimation in Frequency

Now decompose into even (k = 2m) and odd (k = 2m+1) sequences.

$$X(2m) = \sum_{n=0}^{(N/2)-1} \left(x(n) + x\left(n + \frac{N}{2} \right) \right) W_N^{2mn}$$

$$X(2m+1) = \sum_{n=0}^{(N/2)-1} \left(x(n) - x\left(n + \frac{N}{2} \right) \right) W_N^{n} W_N^{2mn}$$

$$\boxed{W_N^2 = e^{-j\frac{2\pi \times 2}{N}} = e^{-j\frac{2\pi}{(N/2)}} = W_{N/2}}$$

$$X(2m) = \sum_{n=0}^{(N/2)-1} a(n) W_{N/2}^{mn} = DFT\{a(n) \text{ with } (N/2) \text{ points}\}$$

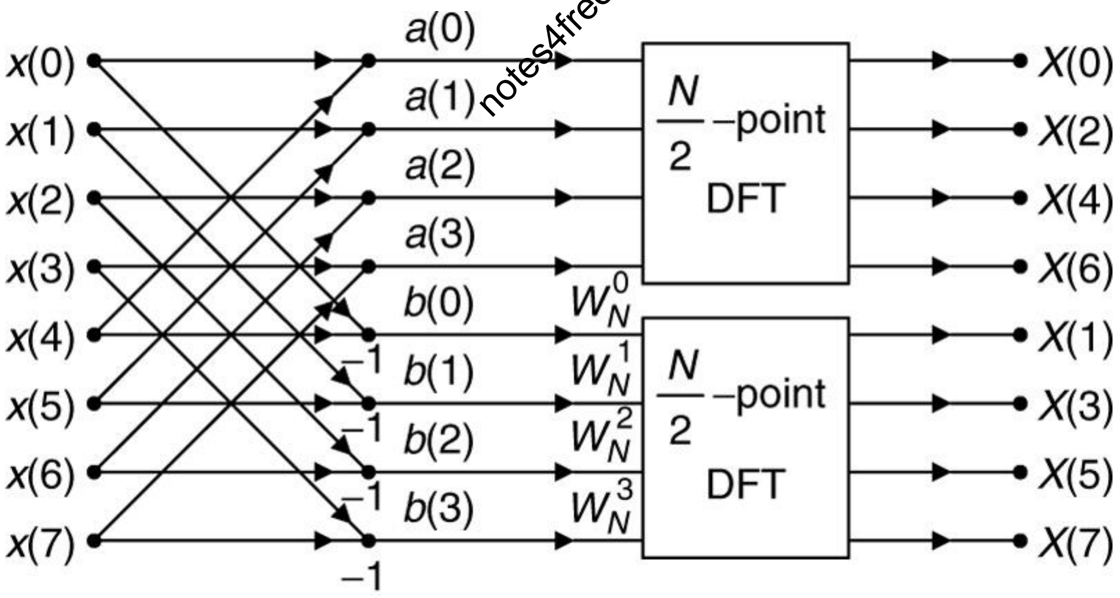
$$X(2m+1) = \sum_{n=0}^{(N/2)-1} b(n) W_N^{n} W_{N/2}^{mn} = DFT\{b(n) W_N^{n} \text{ with } (N/2) \text{ points}\}$$

$$a(n) = x(n) + x\left(n + \frac{N}{2} \right), \text{ for } n = 0,1 \ldots, \frac{N}{2} - 1$$

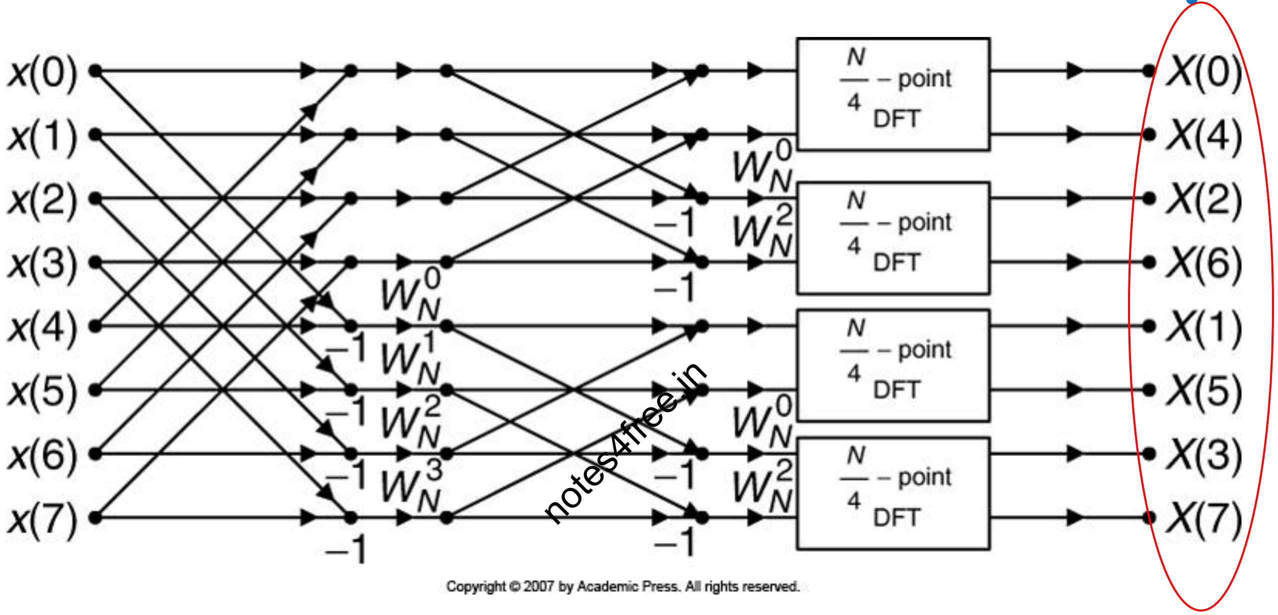
$$b(n) = x(n) - x\left(n + \frac{N}{2} \right), \text{ for } n = 0,1, \ldots, \frac{N}{2} - 1.$$

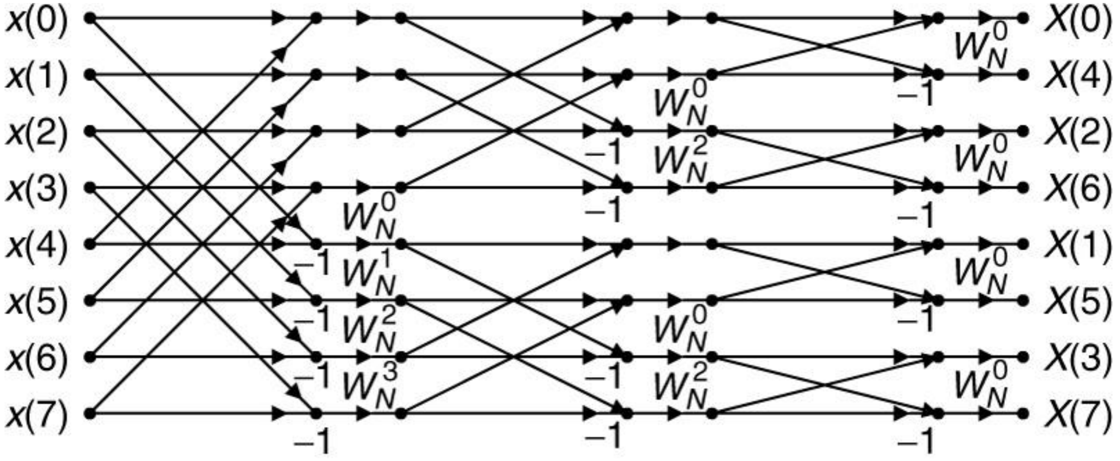
DFT to FFT: Decimation in Frequency

$$DFT\{x(n)\ with\ N\ \text{points}\} = \begin{cases} DFT\{a(n)\ with\ (N/2)\ \text{points}\} \\ DFT\{b(n)W_N^n\ with\ (N/2)\ \text{points}\} \end{cases}$$



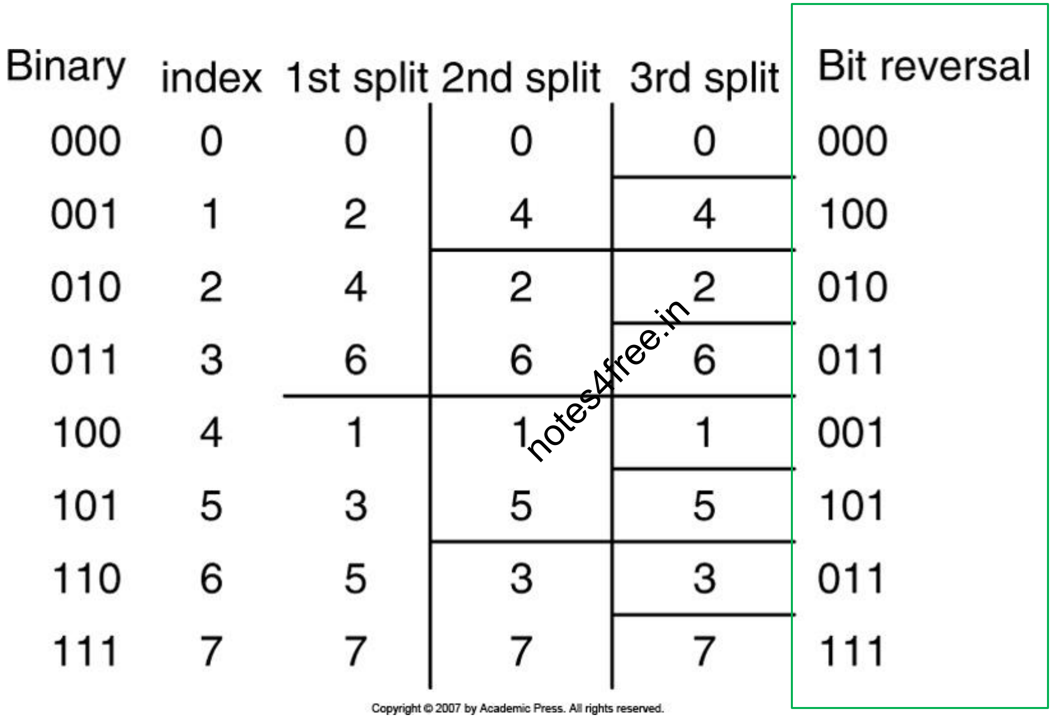
DFT to FFT: Decimation in Frequency


x(0)
x(1)
x(2)
x(3)
x(4)
x(5)
x(6)
x(7)
W_N^0
W_N^1
W_N^2
W_N^3
−1
$\frac{N}{4}$ − point DFT
$\frac{N}{4}$ − point DFT
$\frac{N}{4}$ − point DFT
$\frac{N}{4}$ − point DFT
X(0)
X(4)
X(2)
X(6)
X(1)
X(5)
X(3)
X(7)



12 complex multiplication

DFT to FFT: Decimation in Frequency



Binary	index	1st split	2nd split	3rd split	Bit reversal
000	0	0	0	0	000
001	1	2	4	4	100
010	2	4	2	2	010
011	3	6	6	6	011
100	4	1	1	1	001
101	5	3	5	5	101
110	6	5	3	3	011
111	7	7	7	7	111

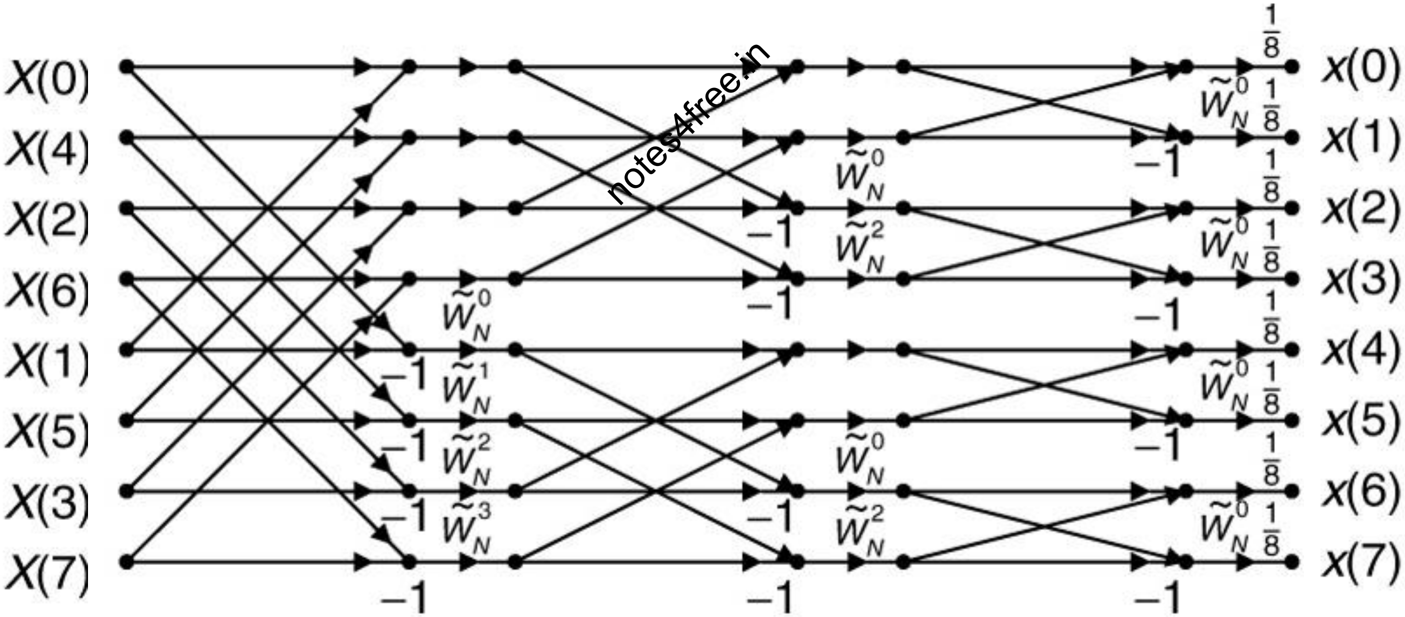
Complex multiplications of DFT $= N^2$, and

Complex multiplications of FFT $= \frac{N}{2}\log_2(N)$

For 1024 samples data sequence, DFT requires 1024×1024 = 1048576 complex multiplications. FFT requires (1024/2)log(1024) = 5120 complex multiplications.

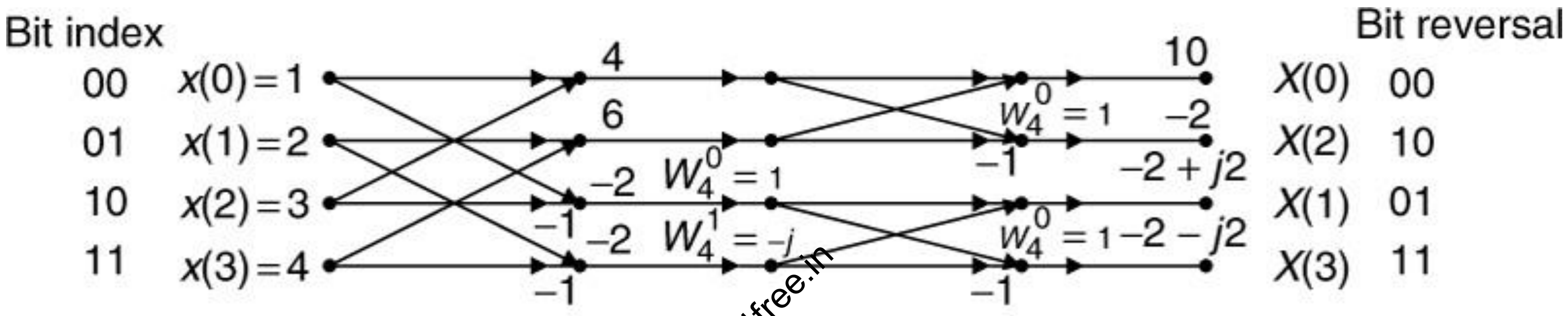
IFFT: Inverse FFT

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k)W_N^{-kn} = \frac{1}{N}\sum_{k=0}^{N-1} X(k)\tilde{W}_N^{kn}, \text{ for } k = 0, 1, \ldots, N-1.$$



FFT and IFFT Examples

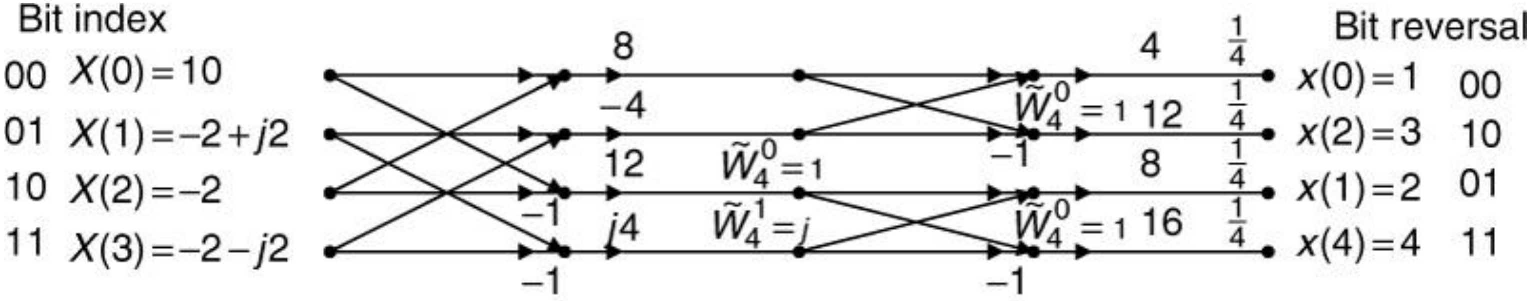
FFT



Copyright © 2007 by Academic Press. All rights reserved.

Number of complex multiplication = $\frac{N}{2}\log_2(N) = \frac{4}{2}\log_2(4) = 4.$

IFFT



Copyright © 2007 by Academic Press. All rights reserved.

DFT to FFT: Decimation in Time

Split the input sequence x(n) into the even indexed x(2m) and x(2m + 1), each with N/2 data points.

$$X(k) = \sum_{m=0}^{(N/2)-1} x(2m)W_N^{2mk} + \sum_{m=0}^{(N/2)-1} x(2m+1)W_N^k W_N^{2mk},$$

for $k = 0, 1, \ldots, N - 1$

Using

$$w_N^2 = \left(e^{-j2\pi/N}\right)^2 = e^{-j2\pi/(N/2)} = w_{N/2}$$

$$X(k) = \sum_{m=0}^{(N/2)-1} x(2m)W_{N/2}^{mk} + W_N^k \sum_{m=0}^{(N/2)-1} x(2m+1)W_{N/2}^{mk},$$

for $k = 0, 1, \ldots, N - 1.$

DFT to FFT: Decimation in Time

Define new functions as

$$G(k) = \sum_{m=0}^{(N/2)-1} x(2m) W_{N/2}^{mk} = DFT\{x(2m)\ with\ (N/2) \text{ points}\}$$

$$H(k) = \sum_{m=0}^{(N/2)-1} x(2m+1) W_{N/2}^{mk} = DFT\{x(2m+1)\ with\ (N/2) \text{ points}\}.$$

As,

$$G(k) = G\left(k + \frac{N}{2}\right), \text{ for } k = 0, 1, \ldots, \frac{N}{2} - 1$$

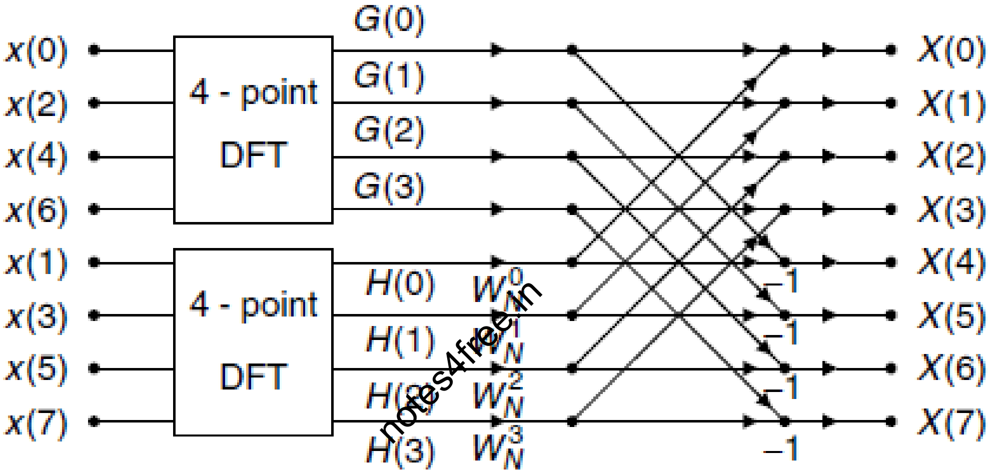
$$H(k) = H\left(k + \frac{N}{2}\right), \text{ for } k = 0, 1, \ldots, \frac{N}{2} - 1.$$

$$X(k) = G(k) + W_N^k H(k), \text{ for } k = 0, 1, \ldots, \frac{N}{2} - 1.$$

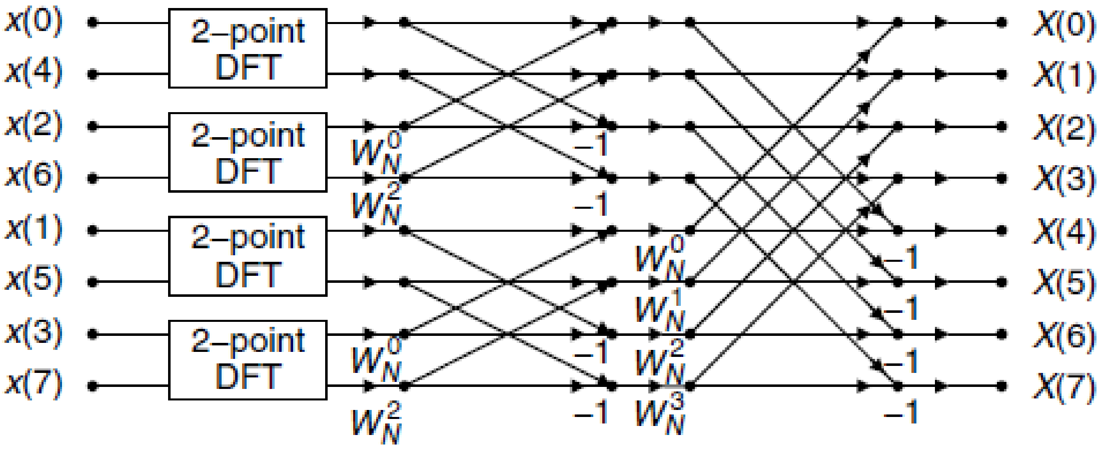
$$X\left(\frac{N}{2} + k\right) = G(k) - W_N^k H(k), \text{ for } k = 0, 1, \ldots, \frac{N}{2} - 1. \leftarrow \quad W_N^{(N/2+k)} = -W_N^k.$$

DFT to FFT: Decimation in Time

First iteration:

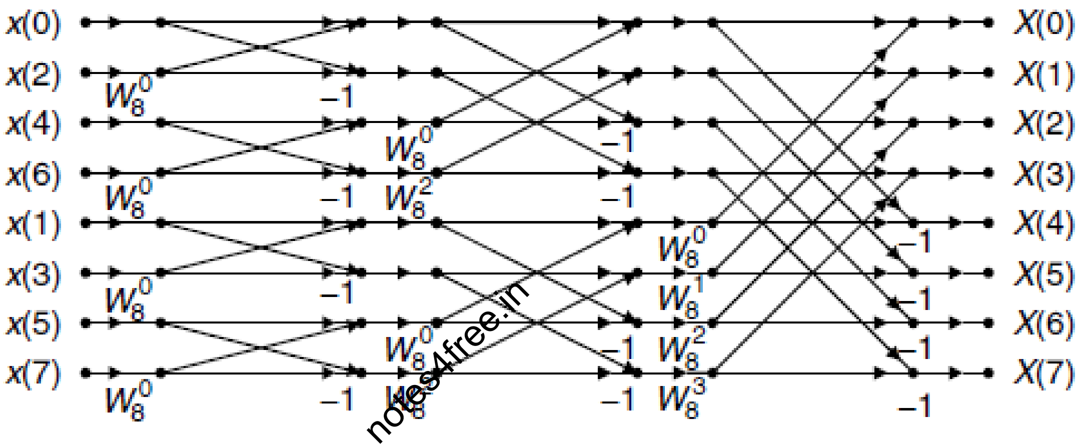


Second iteration:



DFT to FFT: Decimation in Time

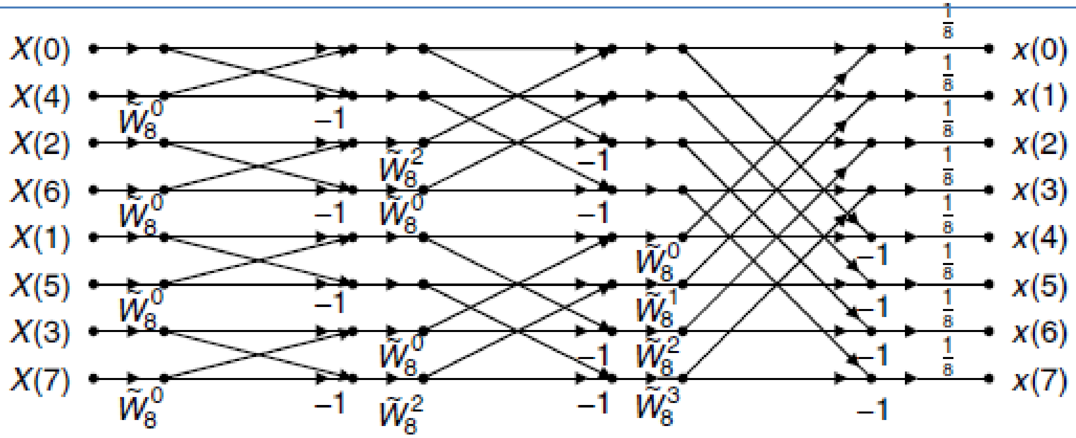
Third iteration:



$$W_N = e^{-\frac{2\pi}{N}} = \cos\left(\frac{2\pi}{N}\right) - j\sin\left(\frac{2\pi}{N}\right)$$

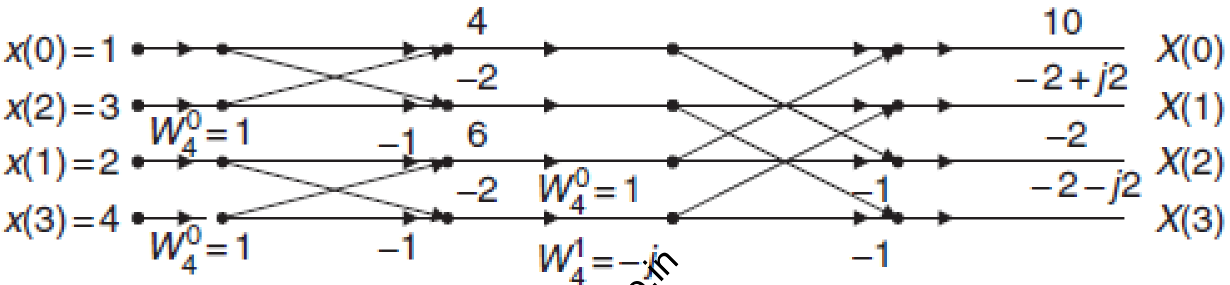
$$W_8^2 = e^{-\frac{2\pi\times 2}{8}} = e^{-\frac{\pi}{2}} = \cos(\pi/2) - j\sin(\pi/2) = -j$$

IFFT

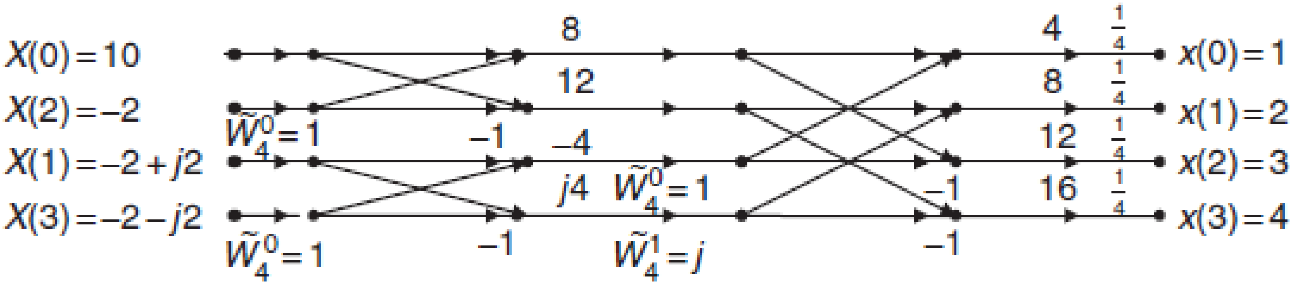


FFT and IFFT Examples

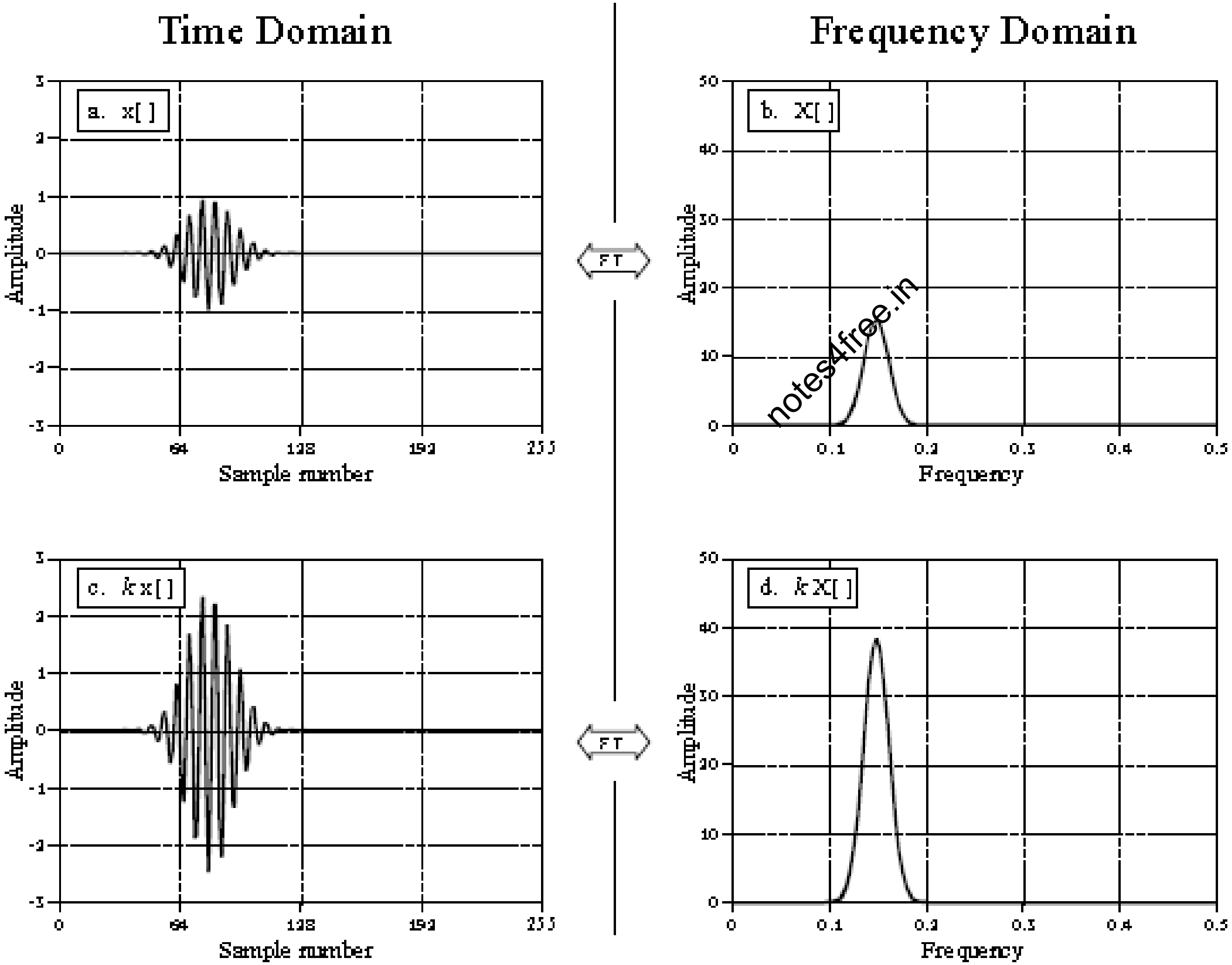
FFT



IFFT



Fourier Transform Properties (1)



FT is linear:

- **Homogeneity**
- **Additivity**

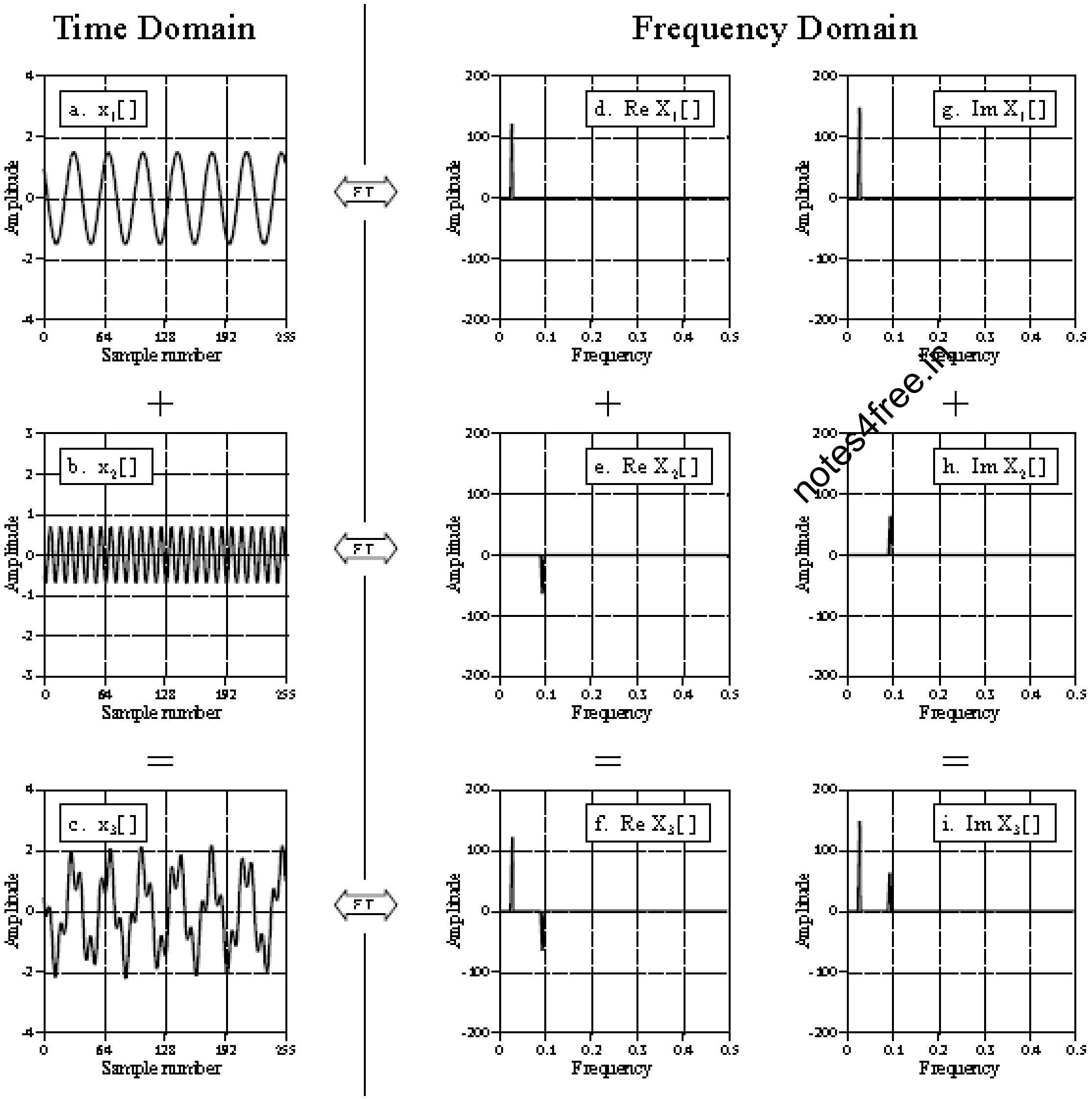
Homogeneity:

$x[] \xrightarrow{DFT} X[]$

$kx[] \xrightarrow{DFT} kX[]$

Frequency is not changed.

Fourier Transform Properties (2)



Additivity

$$If: x_1[n]+x_2[n]=x_3[n]$$

$$Then: \mathrm{Re}\, X_1[f]+\mathrm{Re}\, X_2[f]=\mathrm{Re}\, X_3[f]$$

$$and \quad \mathrm{Im}\, X_1[f]+\mathrm{Im}\, X_2[f]=\mathrm{Im}\, X_3[f]$$

Fourier Transform Pairs

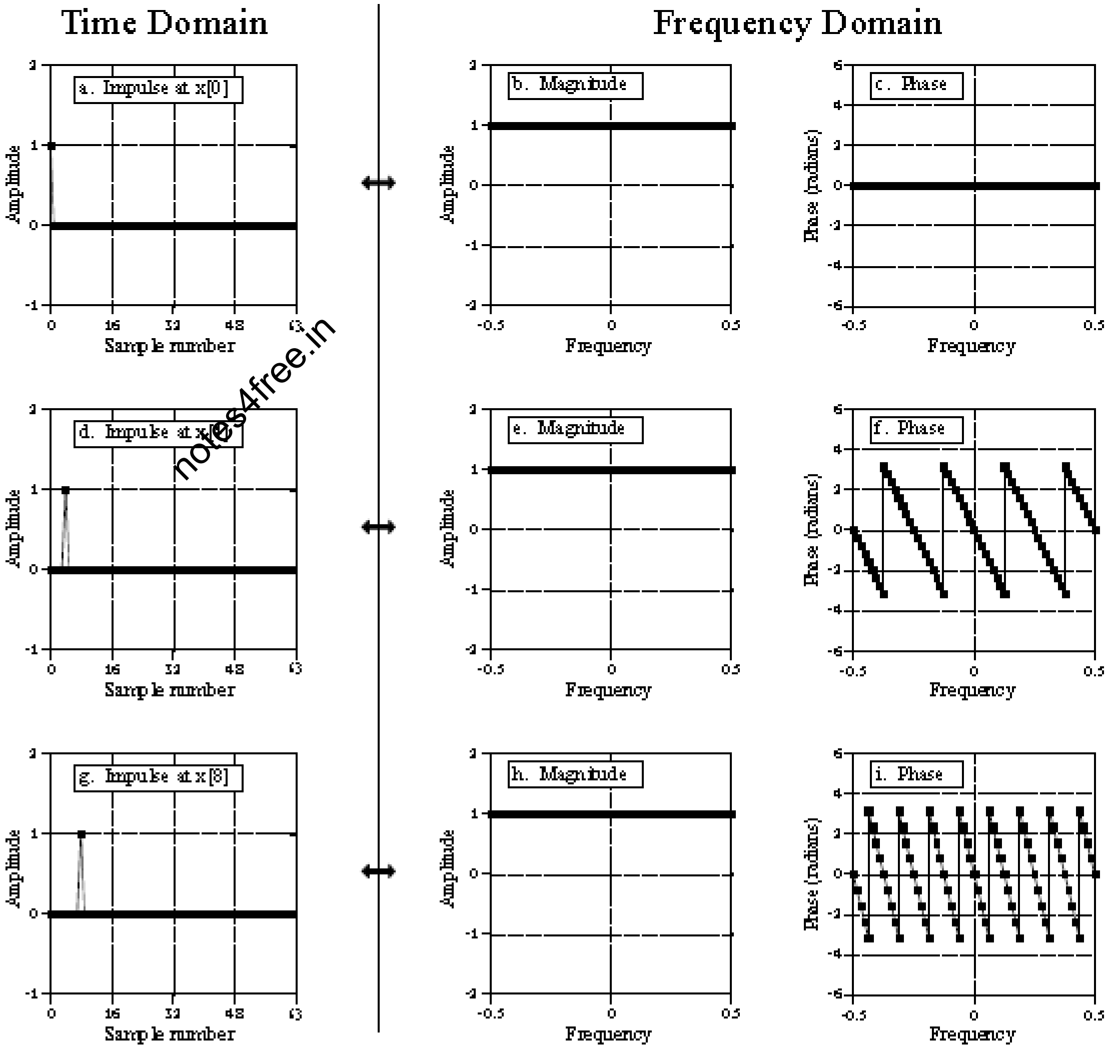
Delta Function Pairs in Polar Form

Delta Function

Shifted Delta Function

Same Magnitude, Different Phase

Shifted Delta Function



MODULE-2

BASIC CONCEPTS AND MODULES AND PORTS

2.1: Objectives

- Understand the lexical conventions and define the logic value set and data type.
- Identify useful system tasks and basic compiler directives.
- Identify and understanding of components of a Verilog module definition.
- Understand the port connection rules and connection to external signals by ordered list and by name.

2.2 Lexical conventions

The basic lexical conventions used by Verilog HDL are similar to those in the C programming language. Verilog contains a stream of tokens. Tokens can be comments, delimiters, numbers, strings, identifiers, and keywords. Verilog HDL is a case-sensitive language. **All keywords are in lowercase.**

2.2.1 Whitespace

Blank spaces (\b), tabs (\t) and newlines (\n) comprise the whitespace. Whitespace is ignored by Verilog except when it separates tokens. Whitespace is not ignored in strings.

2.2.2 Comments

Comments can be inserted in the code for readability and documentation. There are two ways to write comments. A one-line comment starts with "//". Verilog skips from that point to the end of line. A multiple-line comment starts with "/*" and ends with "*/". Multiple-line comments cannot be nested. However, one-line comments can be embedded in multiple-line comments.

```
a = b && c; // This is a one-line comment

/* This is a multiple line comment
   */
/* This is /* an illegal */ comment */

/* This is //a legal comment */
```

2.2.3 Operators

Operators are of three types: unary, binary, and ternary. Unary operators precede the operand. Binary operators appear between two operands. Ternary operators have two separate operators that separate three operands.

a = ~ b; // ~ is a unary operator. b is the operand

a = b && c; // && is a binary operator. b and c are operands

a = b ? c : d; // ?: is a ternary operator. b, c and d are operands

2.2.4 Number Specification

There are two types of number specification in Verilog: sized and unsized.

Sized numbers

Sized numbers are represented as <size> '<base format> <number>.

<size> is written only in decimal and specifies the number of bits in the number. Legal base formats are decimal ('d or 'D), hexadecimal ('h or 'H), binary ('b or 'B) and octal ('o or 'O). The number is specified as consecutive digits from 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f. Only a subset of these digits is legal for a particular base. Uppercase letters are legal for number specification.

4'b1111 // This is a 4-bit binary number

12'habc // This is a 12-bit hexadecimal number

16'd255 // This is a 16-bit decimal number

Unsized numbers

Numbers that are specified without a <base format> specification are decimal numbers by default. Numbers that are written without a <size> specification have a default number of bits that is simulator- and machine-specific (must be at least 32).

23456 // This is a 32-bit decimal number by default

'hc3 // This is a 32-bit hexadecimal number

'o21 // This is a 32-bit octal number

X or Z values

Verilog has two symbols for unknown and high impedance values. These values are very important for modeling real circuits. An unknown value is denoted by an x. A high impedance value is denoted by z.

12'h13x // This is a 12-bit hex number; 4 least significant bits unknown

6'hx // This is a 6-bit hex number

32'bz // This is a 32-bit high impedance number

An x or z sets four bits for a number in the hexadecimal base, three bits for a number in the octal base and one bit for a number in the binary base. If the most significant bit of a number is 0, x, or z, the number is automatically extended to fill the most significant bits, respectively, with 0, x, or z.

This makes it easy to assign x or z to whole vector. If the most significant digit is 1, then it is also zero extended.

Negative numbers

Negative numbers can be specified by putting a minus sign before the size for a constant number. Size constants are always positive. It is illegal to have a minus sign between <base format> and <number>. An optional signed specifier can be added for signed arithmetic.

6'd3 // 8-bit negative number stored as 2's complement of 3

-6'sd3 // Used for performing signed integer math

4'd-2 // Illegal specification

Underscore characters and question marks

An underscore character "_" is allowed anywhere in a number except the first character. Underscore characters are allowed only to improve readability of numbers and are ignored by Verilog. A question mark "?" is the Verilog HDL alternative for z in the context of numbers. The ? is used to enhance readability in the casex and casez statements.

2.2.5 Strings

A string is a sequence of characters that are enclosed by double quotes. The restriction on a string is that it must be contained on a single line, that is, without a carriage return. It cannot be on multiple lines. Strings are treated as a sequence of one-byte ASCII values.

"Hello Verilog World" // is a string

"a / b" // is a string

2.2.6 Identifiers and Keywords

Keywords are special identifiers reserved to define the language constructs. Keywords are in lowercase. Identifiers are names given to objects so that they can be referenced in the design. Identifiers are made up of alphanumeric characters, the underscore (_), or the dollar sign ($). Identifiers are case sensitive. Identifiers start with an alphabetic character or an underscore. They cannot start with a digit or a $ sign (The $ sign as the first character is reserved for system tasks)

```
 reg value; // reg is a keyword; value is an identifier
 input clk; // input is a keyword, clk is an identifier
```

2.2.7 Escaped Identifiers

Escaped identifiers begin with the backslash (\ ) character and end with whitespace (space, tab, or newline). All characters between backslash and whitespace are processed literally. Any printable ASCII character can be included in escaped identifiers.

Neither the backslash nor the terminating whitespace is considered to be a part of the identifier.

```
  \a+b-c
  \**my_name**
```

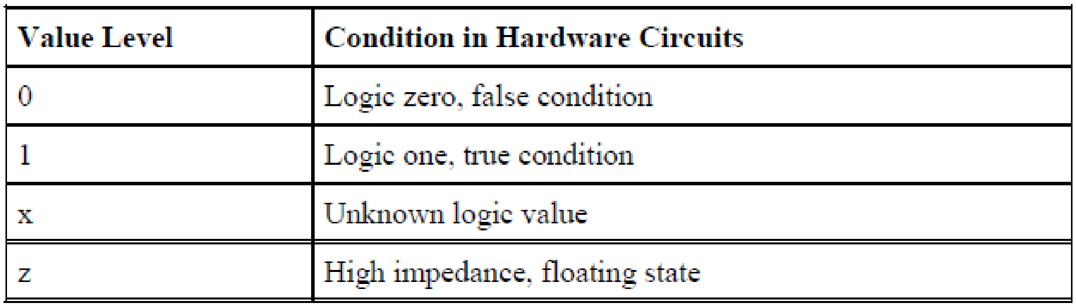
2.3 Data Types

This section discusses the data types used in Verilog.

2.3.1 Value Set

Verilog supports four values and eight strengths to model the functionality of real hardware. The four value levels are listed in Table 2-1.

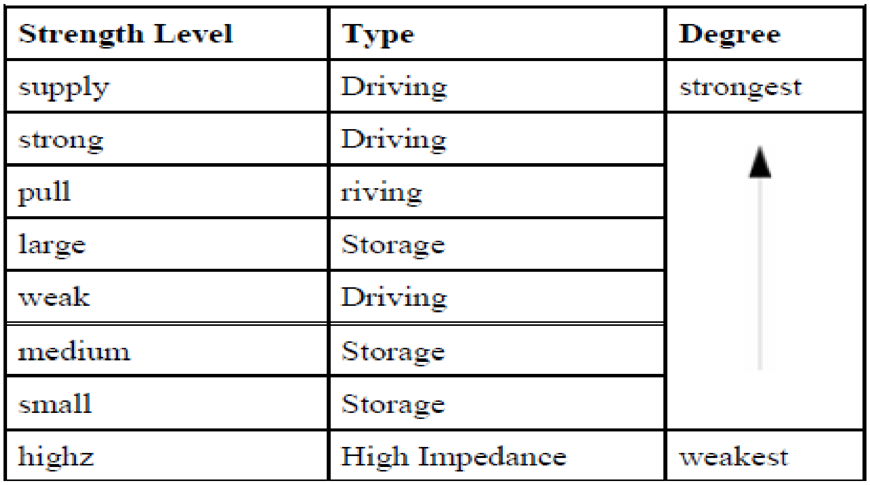
Table 2-1. Value Levels



Value Level	Condition in Hardware Circuits
0	Logic zero, false condition
1	Logic one, true condition
x	Unknown logic value
z	High impedance, floating state

In addition to logic values, strength levels are often used to resolve conflicts between drivers of different strengths in digital circuits. Value levels 0 and 1 can have the strength levels listed in Table2-2.

Table 2-2. Strength Levels



Strength Level	Type	Degree
supply	Driving	strongest
strong	Driving	↑
pull	riving	
large	Storage	
weak	Driving	
medium	Storage	
small	Storage	
highz	High Impedance	weakest

If two signals of unequal strengths are driven on a wire, the stronger signal prevails. For example, if two signals of strength strong1 and weak0 contend, the result is resolved as a strong1. If two signals of equal strengths are driven on a wire, the result is unknown. If two signals of strength strong1 and strong0 conflict, the result is an x.

2.3.2 Nets

Nets represent connections between hardware elements. Just as in real circuits, nets have values continuously driven on them by the outputs of devices that they are connected to. In Figure 2.1 net a is connected to the output of and gate g1. Net a will continuously assume the value computed at the output of gate g1, which is b & c.

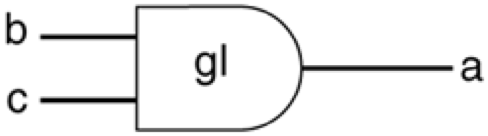


Figure 2.1. Example of Nets

Nets are declared primarily with the keyword wire. Nets are one-bit values by default unless they are declared explicitly as vectors. The terms wire and net are often used interchangeably. The default value of a net is z (except the trireg net, which defaults to x). Nets get the output value of their drivers.

If a net has no driver, it gets the value z.

wire a; // Declare net a for the above circuit

wire b,c; // Declare two wires b,c for the above circuit

wire d = 1'b0; // Net d is fixed to logic value 0 at declaration.

2.3.3 Registers

Registers represent data storage elements. Registers retain value until another value is placed onto them. In Verilog, the term register merely means a variable that can hold a value. Unlike a net, a register does not need a driver. Verilog registers do not need a clock as hardware registers do. Values of registers can be changed anytime in a simulation by assigning a new value to the register.

Register data types are commonly declared by the keyword reg.

Example 3-1 Example of Register

```
reg reset; // declare a variable reset that can hold its value
initial // keyword to specify the initial  value of reg.
reset = 1'b1; //initialize reset to 1 to reset the digital circuit.
#100 reset = 1'b0; // after 100 time units reset is deasserted.
end
```

Example 2-2 Signed Register Declaration

```
reg signed [63:0] m; // 64 bit signed value
integer i; // 32 bit signed value
```

2.3.4 Vectors

Nets or reg data types can be declared as vectors (multiple bit widths). If bit width is not specified, the default is scalar (1-bit).

```
wire a; // scalar net variable, default
wire [7:0] bus; // 8-bit   bus
wire [31:0] busA,busB,busC; // 3 buses of 32-bit width.
reg clock; // scalar register, default
reg [0:40] virtual_addr; // Vector register, virtual address 41 bits wide
```

Vectors can be declared at [high# : low#] or [low# : high#], but the left number in the squared brackets is always the most significant bit of the vector. In the example shown above, bit 0 is the most significant bit of vector virtual_addr.

Vector Part Select

For the vector declarations shown above, it is possible to address bits or parts of vectors.

```
busA[7] // bit # 7 of vector busA
bus[2:0] // Three least significant bits of vector bus,
// using bus[0:2] is illegal because the significant bit shouldalways be on the left of a range specification
virtual_addr[0:1] // Two most significant bits of vector virtual_addr
```

Variable Vector Part Select

Another ability provided in Verilog HDL is to have variable part selects of a vector. This allows part selects to be put in for loops to select various parts of the vector. There are two special part-select operators:

[<*starting_bit*>+:*width*] - part-select increments from starting bit.

[<*starting_bit*>-:*width*] - part-select decrements from starting bit.

The starting bit of the part select can be varied, but the width has to be constant. The following example shows the use of variable vector part select:

```
reg [255:0] data1; //Little endian notation
reg [0:255] data2; //Big endian notation
reg [7:0] byte;
//Using a variable part select, one can choose parts
byte = data1[31-:8]; //starting bit = 31, width =8 => data[31:24]
byte = data1[24+:8]; //starting bit = 24, width =8 => data[31:24]
byte = data2[31-:8]; //starting bit = 31, width =8 => data[24:31]
byte = data2[24+:8]; //starting bit = 24, width =8 => data[24:31]

//The starting bit can also be a variable. The width has to be constant.
//Therefore, one can use the variable part select
//in a loop to select all bytes of the vector.
 for (j=0; j<=31; j=j+1)
byte = data1[(j*8)+:8]; //Sequence is [7:0], [15:8]... [255:248]
//Can initialize a part of the vector
data1[(byteNum*8)+:8] = 8'b0; //If byteNum = 1, clear 8 bits [15:8]
```

2.3.5 Integer , Real, and Time Register Data Types

Integer, real, and time register data types are supported in Verilog.

Integer

An integer is a general purpose register data type used for manipulating quantities. Integers are declared by the keyword integer. Although it is possible to use reg as a general-purpose variable, it is more convenient to declare an integer variable for purposes such as counting. The default width for an integer is the host-machine word size, which is implementation-specific but is at least 32 bits. Registers declared as data type reg store values as unsigned quantities, whereas integers store values as signed quantities.

```
integer counter; // general purpose variable used as a counter.
initial
counter = -1; // A negative one is stored in the counter
```

Real

Real number constants and real register data types are declared with the keyword real. They can be specified in decimal notation (e.g., 3.14) or in scientific notation (e.g., 3e6, which is 3 x 10^6). Real numbers cannot have a range declaration, and their default value is 0. When a real value is assigned to an integer, the real number is rounded off to the nearest integer.

```
real delta; // Define a real variable called delta initial
begin
delta = 4e10; // delta is assigned in scientific notation

delta = 2.13; // delta is assigned a value 2.13 end
integer i; // Define an integer i
initial
i = delta; // i gets the value 2 (rounded value of 2.13)
```

Time

Verilog simulation is done with respect to simulation time. A special time register data type is used in Verilog to store simulation time. A time variable is declared with the keyword time. The width for time register data types is implementation-specific but is at least 64 bits.The system function $time is invoked to get the current simulation time.

```
time save_sim_time; // Define a time variable save_sim_time

initial
```

save_sim_time = $time; // Save the current simulation time

Arrays

Arrays are allowed in Verilog for reg, integer, time, real, realtime and vector register data types. Multi-dimensional arrays can also be declared with any number of dimensions. Arrays of nets can also be used to connect ports of generated instances. Each element of the array can be used in the same fashion as a scalar or vector net. Arrays are accessed by <array_name>[<subscript>]. For multi- dimensional arrays, indexes need to be provided for each dimension.

integer count[0:7]; // An array of 8 count variables

reg bool[31:0]; // Array of 32 one-bit boolean register variables time

chk_point[1:100]; // Array of 100 time checkpoint variables reg [4:0]

port_id[0:7]; // Array of 8 port_ids; each port_id is 5 bits wide

integer matrix[4:0][0:255]; // Two dimensional array of integers

reg [63:0] array_4d [15:0][7:0][7:0][255:0]; //Four dimensional array

wire [7:0] w_array2 [5:0]; // Declare an array of 8 bit vector wire

wire w_array1[7:0][5:0]; // Declare an array of single bit wires.

It is important not to confuse arrays with net or register vectors. A vector is a single element that is n-bits wide. On the other hand, arrays are multiple elements that are 1-bit or n-bits wide.

Examples of assignments to elements of arrays discussed above are shown below:

count[5] = 0; // Reset 5th element of array of count variables

chk_point[100] = 0; // Reset 100th time check point value

port_id[3] = 0; // Reset 3rd element (a 5-bit value) of port_id array.

matrix[1][0] = 33559; // Set value of element indexed by [1][0] to 33559

port_id = 0; // Illegal syntax - Attempt to write the entire array

matrix [1] = 0; // Illegal syntax - Attempt to write [1][0]..[1][255]

2.3.6 Memories

In digital simulation, one often needs to model register files, RAMs, and ROMs. Memories are modeled in Verilog simply as a one-dimensional array of registers. Each element of the array is known as an element or word and is addressed by a single array index. Each word can be one or more bits. It is important to differentiate between n 1-bit registers and one n-bit register. A particular word in memory is obtained by using the address as a memory array subscript.

reg mem1bit[0:1023]; // Memory mem1bit with 1K 1-bit words

reg [7:0] membyte[0:1023]; // Memory membyte with 1K 8-bit words(bytes)

membyte[511] // Fetches 1 byte word whose address is 511.

2.3.7 Parameters

Verilog allows constants to be defined in a module by the keyword parameter. Parameters cannot be used as variables. Parameter values for each module instance can be overridden individually at compile time. This allows the module instances to be customized. This aspect is discussed later. Parameter types and sizes can also be defined.

parameter port_id = 5; // Defines a constant port_id

parameter cache_line_width = 256; // Constant defines width of cache line

parameter signed [15:0] WIDTH; // Fixed sign and range for parameter WIDTH

2.3.8 Strings

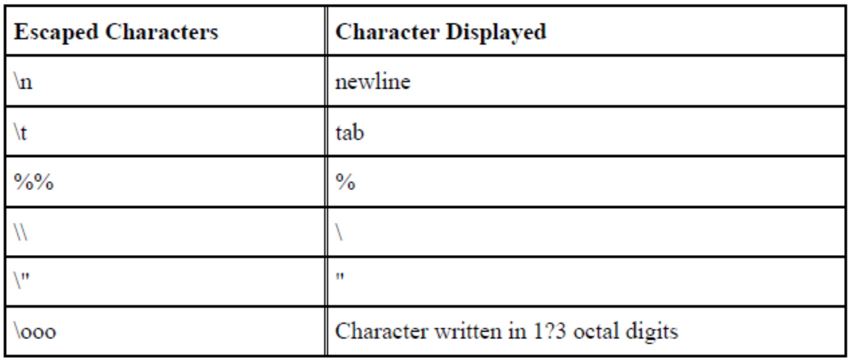
Strings can be stored in reg. The width of the register variables must be large enough to hold the string. Each character in the string takes up 8 bits (1 byte). If the width of the register is greater than the size of the string, Verilog fills bits to the left of the string with zeros. If the register width is smaller than the string width, Verilog truncates the leftmost bits of the string. It is always safe to declare a string that is slightly wider than necessary.

reg [8*18:1] string_value; // Declare a variable that is 18 bytes wide initial

string_value = "Hello Verilog World"; // String can be stored in variable

Special characters serve a special purpose in displaying strings, such as newline, tabs, and displaying argument values. Special characters can be displayed in strings only when they are preceded by escape characters, as shown in Table 2-3

Table 2-3. Special Characters



Escaped Characters	Character Displayed
\n	newline
\t	tab
%%	%
\\	\
\"	"
\ooo	Character written in 1?3 octal digits

2.4 System Tasks and Compiler Directives

In this section, we introduce two special concepts used in Verilog: system tasks and compiler directives.

2.4.1 System Tasks

Verilog provides standard system tasks for certain routine operations. All system tasks appear in the form $<keyword>. Operations such as displaying on the screen, monitoring values of nets, stopping, and finishing are done by system tasks.

Displaying information

$display is the main system task for displaying values of variables or strings or expressions. This is one of the most useful tasks in Verilog.

Usage: $display(p1, p2, p3,....., pn);

p1, p2, p3,..., pn can be quoted strings or variables or expressions. The format of $display is very similar to printf in C. A $display inserts a newline at the end of the string by default. A $display without any arguments produces a newline.

Monitoring information

Verilog provides a mechanism to monitor a signal when its value changes. This facility is provided by the $monitor task.

Usage: $monitor(p1,p2,p3,....,pn);

The parameters p1, p2, ... , pn can be variables, signal names, or quoted strings. A format similar to the $display task is used in the $monitor task. $monitor continuously monitors the values of the variables or signals specified in the parameter list and displays all parameters in the list whenever the value of any one variable or signal changes. Unlike $display, $monitor needs to be invoked only once. Only one monitoring list can be active at a time.

If there is more than one $monitor statement in your simulation, the last $monitor statement will be the active statement. The earlier $monitor statements will be overridden.

Two tasks are used to switch monitoring on and off.

Usage:

$monitoron;

$monitoroff;

The $monitoron tasks enables monitoring, and the $monitoroff task disables monitoring during a simulation.

Example of Monitor Statement

```
//Monitor time and value of the signals clock and reset
//Clock toggles every 5 time units and reset goes down at 10 time units
initial
begin
$monitor ($time," Value of signals clock = %b reset = %b", clock,reset);
end
```

Partial output of the monitor statement:

```
-- 0 Value of signals clock = 0 reset = 1
-- 5 Value of signals clock = 1 reset = 1
-- 10 Value of signals clock = 0 reset = 0
```

Stopping and finishing in a simulation

The task $stop is provided to stop during a simulation.

Usage: $stop;

The $stop task puts the simulation in an interactive mode. The designer can then debug the design from the interactive mode. The $stop task is used whenever the designer wants to suspend the simulation and examine the values of signals in the design.

The $finish task terminates the simulation.

Usage: $finish;

Examples of $stop and $finish are given below

Example of Stop and Finish Tasks

```
// Stop at time 100 in the simulation and examine the results
// Finish the simulation at time 1000.
initial
begin
clock = 0;
reset = 1;
#100 $stop; // This will suspend the simulation at time = 100
#900 $finish; // This will terminate the simulation at time = 1000
end
```

2.4.2 Compiler Directives

Compiler directives are provided in Verilog. All compiler directives are defined by using the `<keyword> construct. The two most useful compiler directives are

`define

The `define directive is used to define text macros in Verilog .The Verilog compiler substitutes the text of the macro wherever it encounters a `<macro_name>. This is similar to the #define construct in C. The defined constants or text macros are used in the Verilog code by preceding them with a ` (back tick).

Example for `define Directive

```
//define a text macro that defines default word size
//Used as 'WORD_SIZE in the code
'define WORD_SIZE 32
//define an alias. A $stop will be substituted wherever 'S appears
'define S $stop;
//define a frequently used text string
'define WORD_REG reg [31:0]
```

`include

The `include directive allows you to include entire contents of a Verilog source file in another Verilog file during compilation. This works similarly to the #include in the C programming language.

Example for `include Directive

```
// Include the file header.v, which contains declarations in themain verilog file design.v.
'include header.v
...
...
<Verilog code in file design.v>
...
...
```

2.5 Modules

Module is a basic building block in Verilog. A module definition always begins with the keyword module. The module name, port list, port declarations, and optional parameters must come first in a module definition. Port list and port declarations are present only if the module has any ports to interact with the external environment.

The five components within a module are: variable declarations, dataflow statements, instantiation of lower modules, behavioral blocks, and tasks or functions. These components can be in any order and at any place in the module definition.

The endmodule statement must always come last in a module definition. All components except module, module name, and endmodule are optional and can be mixed and matched as per design needs. Verilog allows multiple modules to be defined in a single file. The modules can be defined in any order in the file.

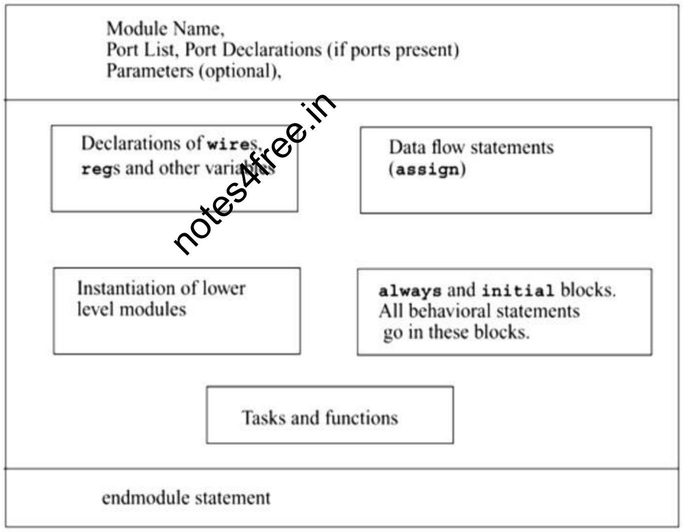

Module Name,
Port List, Port Declarations (if ports present)
Parameters (optional),
Declarations of wires, regs and other variables
Data flow statements (assign)
Instantiation of lower level modules
always and initial blocks. All behavioral statements go in these blocks.
Tasks and functions
endmodule statement

Figure 2.2.:Components of a Verilog Module

Consider a simple example of an SR latch, as shown in Figure 2.3

Figure 2-3. SR Latch

The SR latch has S and R as the input ports and Q and Qbar as the output ports. The SR latch and its stimulus can be modeled as shown in Example.

Example of Components of SR Latch

```
// This example illustrates the different components of a module
// Module name and port list
// SR_latch module
module SR_latch(Q, Qbar, Sbar, Rbar);
//Port declarations
output Q, Qbar;
input Sbar, Rbar;
// Instantiate lower-level modules
// In this case, instantiate Verilog primitive nand gates
// Note how the wires are connected in a cross-coupled fashion. nand n1(Q, Sbar, Qbar);
nand n2(Qbar, Rbar, Q);
// endmodule statement
endmodule

// Module name and port list
// Stimulus module
module Top;
// Declarations of wire, reg, and other variables
reg set, reset;
// Instantiate lower-level modules
// In this case, instantiate SR_latch Feed inverted set and reset signals to the SR latch
SR_latch m1(q, qbar, ~set, ~reset);
// Behavioral block, initial
initial
begin
$monitor($time, " set = %b, reset= %b, q= %b\n",set,reset,q);
set = 0; reset = 0;
#5 reset = 1;
```

```
#5 reset = 0;
#5 set = 1;
end
// endmodule statement
endmodule
```

From the above example following characteristics are noticed:

- In the SR latch definition above ,all components described in Figure 2-2 need not be present in a module. We do not find variable declarations, dataflow (assign) statements, or behavioral blocks (always or initial).
- However, the stimulus block for the SR latch contains module name, wire, reg, and variable declarations, instantiation of lower level modules, behavioral block (initial), and endmodule statement but does not contain port list, port declarations, and data flow (assign) statements.
- Thus, all parts except module, module name, and endmodule are optional and can be mixed and matched as per design needs.

2.6 Ports

Ports provide the interface by which a module can communicate with its environment. For example, the input/output pins of an IC chip are its ports. The environment can interact with the module only through its ports. The internals of the module are not visible to the environment. This provides a very powerful flexibility to the designer. The internals of the module can be changed without affecting the environment as long as the interface is not modified. Ports are also referred to as terminals.

2.6.1 List of Ports

A module definition contains an optional list of ports. If the module does not exchange any signals with the environment, there are no ports in the list. Consider a 4-bit full adder that is instantiated inside a top-level module Top. The diagram for the input/output ports is shown in Figure 2-4.

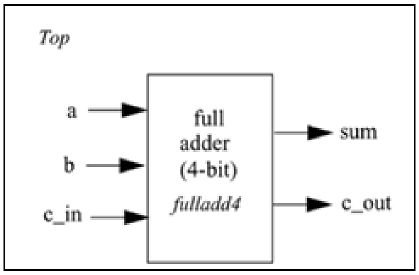


Figure 2-4. I/O Ports for Top and Full Adder

From the above figure, the module Top is a top-level module. The module fulladd4 is instantiated below Top. The module fulladd4 takes input on ports a, b, and c_in and produces an output on ports sum and c_out. Thus, module fulladd4 performs an addition for its environment. The module Top is a top-level module in the simulation and does not need to pass signals to or receive signals from the environment. Thus, it does not have a list of ports. The module names and port lists for both module declarations in Verilog are as shown in below example.

Example of List of Ports

```
module fulladd4(sum, c_out, a, b, c_in); //Module with a list of ports
module Top; // No list of ports, top-level module in simulation
```

2.6.2 Port Declaration

All ports in the list of ports must be declared in the module. Ports can be declared as follows:

input -Input port

output- Output port

inout- Bidirectional port

Each port in the port list is defined as input, output, or inout, based on the direction of the port signal. Thus, for the example of the the port declarations will be as shown in example below.

Example for Port Declarations

```
module fulladd4(sum, c_out, a, b, c_in);
//Begin port declarations section
output[3:0] sum;
output c_cout;
 input [3:0] a, b;
input c_in;
//End port declarations section
...
<module internals>
... endmodule
```

All port declarations are implicitly declared as wire in Verilog. Thus, if a port is intended to be a wire, it is sufficient to declare it as output, input, or inout. Input or inout ports are normally declared as wires.

However, if output ports hold their value, they must be declared as reg. Ports of the type input and inout cannot be declared as reg because reg variables store values and input ports should not store values but simply reflect the changes in the external signals they are connected to.

Alternate syntax for port declaration is shown in below example. This syntax avoids the duplication of naming the ports in both the module definition statement and the module port list definitions. If a port is declared but no data type is specified, then, under specific circumstances, the signal will default to a wire data type.

Example for ANSI C Style Port Declaration Syntax

```
module fulladd4(output reg [3:0] sum,
output reg c_out,
input [3:0] a, b, //wire by default
input c_in); //wire by default
...
<module internals>
...
endmodule
```

2.6.3 Port Connection Rules

A port as consisting of two units, one unit that is internal to the module and another that is external to the module. The internal and external units are connected. There are rules governing port connections when modules are instantiated within other modules. The Verilog simulator complains if any port connection rules are violated. These rules are summarized in Figure2.5

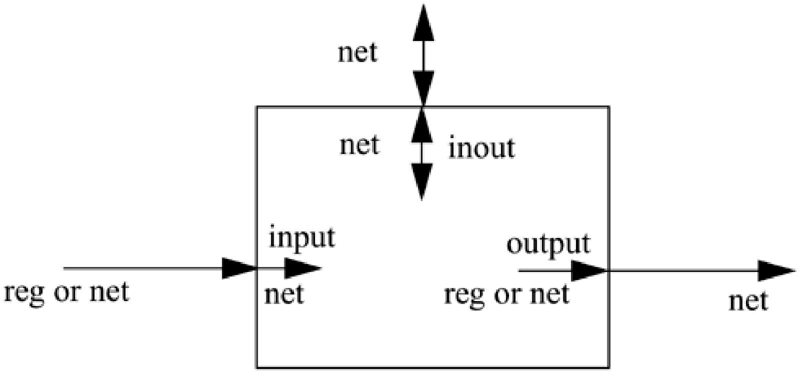


Figure 2-5. Port Connection Rules

Inputs

Internally, input ports must always be of the type net. Externally, the inputs can be connected to a variable which is a reg or a net.

Outputs

Internally, outputs ports can be of the type reg or net. Externally, outputs must always be connected to a net. They cannot be connected to a reg.

Inouts

Internally, inout ports must always be of the type net. Externally, inout ports must always be connected to a net.

Width matching

It is legal to connect internal and external items of different sizes when making intermodule port connections. However, a warning is typically issued that the widths do not match.

Unconnected ports

Verilog allows ports to remain unconnected. For example, certain output ports might be simply for debugging, and you might not be interested in connecting them to the external signals. You can let a port remain unconnected by instantiating a module as shown below

```
                    fulladd4  fa0 (SUM,    , A, B, C_IN); // Output port c_out is unconnected
```

Example of illegal port connection

To illustrate port connection rules, assume that the module fulladd4 Example is instantiated in the stimulus block Top. Below example shows an illegal port connection

Example 2-14 Illegal Port Connection

```
module Top;
//Declare  connection  variables  reg
[3:0]A,B;
reg C_IN;
reg [3:0] SUM;
wire C_OUT;
//Instantiate fulladd4, call it fa0
fulladd4 fa0(SUM, C_OUT, A, B, C_IN);
//Illegal connection because output port sum in module fulladd4
//is connected to a register variable SUM in module Top.
```

```
.
.
<stimulus>
.
. endmodule
```

This problem is rectified if the variable SUM is declared as a net (wire).

2.7 Connecting Ports to External Signals

There are two methods of making connections between signals specified in the module instantiation and the ports in a module definition. These two methods cannot be mixed. These methods are

Connecting by ordered list

The signals to be connected must appear in the module instantiation in the same order as the ports in the port list in the module definition. Consider the module fulladd4.To connect signals in module Top by ordered list, the Verilog code is shown in below example. Notice that the external signals SUM, C_OUT, A, B, and C_IN appear in exactly the same order as the ports sum, c_out, a, b, and c_in in module definition of fulladd4.

Example 2-15 Connection by Ordered List

```
module Top;
//Declare connection variables
reg [3:0]A,B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;
//Instantiate fulladd4, call it fa_ordered.
//Signals are connected to ports in order (by position)
fulladd4 fa_ordered (SUM, C_OUT, A, B, C_IN);
...
<stimulus>
... endmodule
```

```
module fulladd4(sum, c_out, a, b, c_in);
output[3:0] sum; output c_cout; input [3:0] a, b; input c_in;
...
<module internals>
... endmodule
```

Connecting ports by name

For large designs where modules have, say, 50 ports, remembering the order of the ports in the module definition is impractical and error-prone. Verilog provides the capability to connect external signals to ports by the port names, rather than by position. We could connect the ports by name in above example by instantiating the module fulladd4, as follows. Note that you can specify the port connections in any order as long as the port name in the module definition correctly matches the external signal.

```
// Instantiate module fa_byname and connect signals to ports by name
fulladd4 fa_byname(.c_out(C_OUT), .sum(SUM), .b(B), .c_in(C_IN), .a(A),);
```

Note that only those ports that are to be connected to external signals must be specified in port connection by name. Unconnected ports can be dropped. For example, if the port c_out were to be kept unconnected, the instantiation of fulladd4 would look as follows. The port c_out is simply dropped from the port list.

```
// Instantiate module fa_byname and connect signals to ports by
name fulladd4 fa_byname(.sum(SUM), .b(B), .c_in(C_IN), .a(A),);
```

Another advantage of connecting ports by name is that as long as the port name is not changed, the order of ports in the port list of a module can be rearranged without changing the port connections in module instantiations.

2.8 Hierarchical Names

Every module instance, signal, or variable is defined with an identifier. A particular identifier has a unique place in the design hierarchy. Hierarchical name referencing allows us to denote every identifier in the design hierarchy with a unique name. A hierarchical name is a list of identifiers separated by dots (".") for each level of hierarchy. Thus, any identifier can be addressed from any place in the design by simply specifying the complete hierarchical name of that identifier. The top-level module is called the root module because it is not instantiated anywhere. It is the starting point.

To assign a unique name to an identifier, start from the top-level module and trace the path along the design hierarchy to the desired identifier.

Consider the simulation of SR latch Example. The design hierarchy is shown in Figure 2.6.

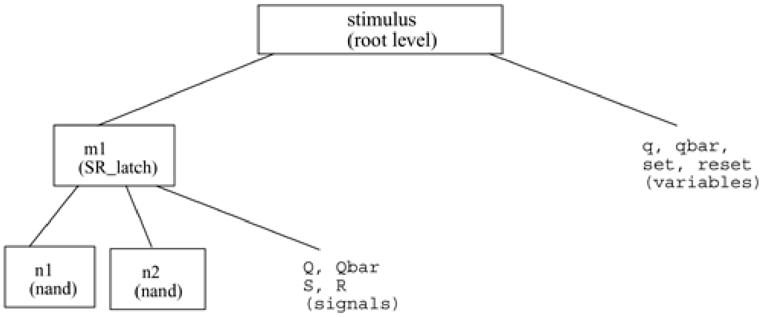


Figure 2-6. Design Hierarchy for SR Latch Simulation

For this simulation, stimulus is the top-level module. Since the top-level module is not instantiated anywhere, it is called the root module. The identifiers defined in this module are q, qbar, set, and reset. The root module instantiates m1, which is a module of type SR_latch. The module m1 instantiates nand gates n1 and n2. Q, Qbar, S, and R are port signals in instance m1. Hierarchical name referencing assigns a unique name to each identifier. To assign hierarchical names, use the module name for root module and instance names for all module instances below the root module.

Example

stimulus
stimulus.q
stimulus.qbar
timulus.set
stimulus.reset
stimulus.m1
stimulus.m1.Q
stimulus.m1.Qbar
stimulus m1.S
stimulus.m1.R
stimulus.n1
stimulus.n2

Each identifier in the design is uniquely specified by its hierarchical path name. To display the level of hierarchy, use the special character %m in the $display task.

2.9: Outcomes

After completion of the module the students are able to:

- Understand the lexical conventions and different data types of verilog.
- Identify useful system tasks such as $display and $monitor and basic compiler directives.
- Understand different components of a Verilog module definition
- Understand the port connection rules and connection to external signals by ordered list and by name

2.10: Recommended questions

1. Describe the lexical conventions used in Verilog HDL with examples.
2. Explain different data types of Verilog HDL with examples
3. What are system tasks and compiler directives?
4. What are the uses of $monitor, $display and $finish system tasks? Explain with examples.
5. Explain `define and `include compiler directives.
6. Explain the components of Verilog HDL module.
7. What are the components of SR latch? Write Verilog HDL module of SR latch.
8. Explain the different types of ports supported by Verilog HDL with examples.
9. Explain the port connection rules of Verilog HDL with examples.
10. How hierarchical names helps in addressing any identifier used in the design from any other level of hierarchy? Explain with examples.
11. What are the basic components of a module? Which components are mandatory?

MODULE -3

GATE LEVEL MODELING AND DATA FLOW MODELING

3.1: Objectives

- Identify logic gate primitives provided in Verilog.
- Understand instantiation of gates, gate symbols, and truth tables for and/or and buf/not type gates.
- Understand how to construct a Verilog description from the logic diagram of the circuit.
- Describe rise, fall, and turn-off delays in the gate-level design and Explain min, max, and typ delays in the gate-level design
- Describe the continuous assignment (assign) statement, restrictions on the assign statement, and the implicit continuous assignment statement.
- Explain assignment delay, implicit assignment delay, and net declaration delay for continuous assignment statements and Define expressions, operators, and operands.
- Use dataflow constructs to model practical digital circuits in Verilog

3.2 Gate Types

A logic circuit can be designed by use of logic gates. Verilog supports basic logic gates as predefined primitives. These primitives are instantiated like modules except that they are predefined in Verilog and do not need a module definition. All logic circuits can be designed by using basic gates. There are two classes of basic gates: **and/or gates and buf/not gates.**

3.2.1 And/Or Gates

And/or gates have one scalar output and multiple scalar inputs. The first terminal in the list of gate terminals is an output and the other terminals are inputs. The output of a gate is evaluated as soon as one of the inputs changes. The and/or gates available in Verilog are: **and, or, xor, nand, nor, xnor.**

The corresponding logic symbols for these gates are shown in Figure 3-1. Consider the gates with two inputs. The output terminal is denoted by out. Input terminals are denoted by i1 and i2.

These gates are instantiated to build logic circuits in Verilog. Examples of gate instantiations are shown below. In Example 3-1, for all instances, OUT is connected to the output out, and IN1 and IN2 are connected to the two inputs i1 and i2 of the gate primitives. Note that the instance name does not need to be specified for primitives. This lets the designer instantiate hundreds of gates without giving them a name. More than two inputs can be specified in a gate instantiation. Gates with more than two inputs are instantiated by simply adding more input ports in the gate instantiation. Verilog automatically instantiates the appropriate gate.

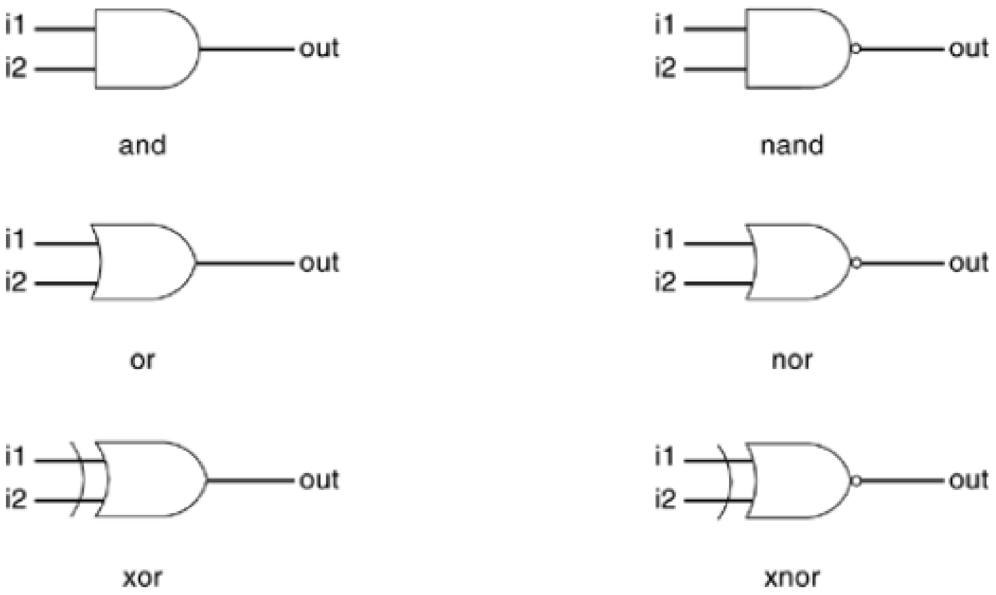


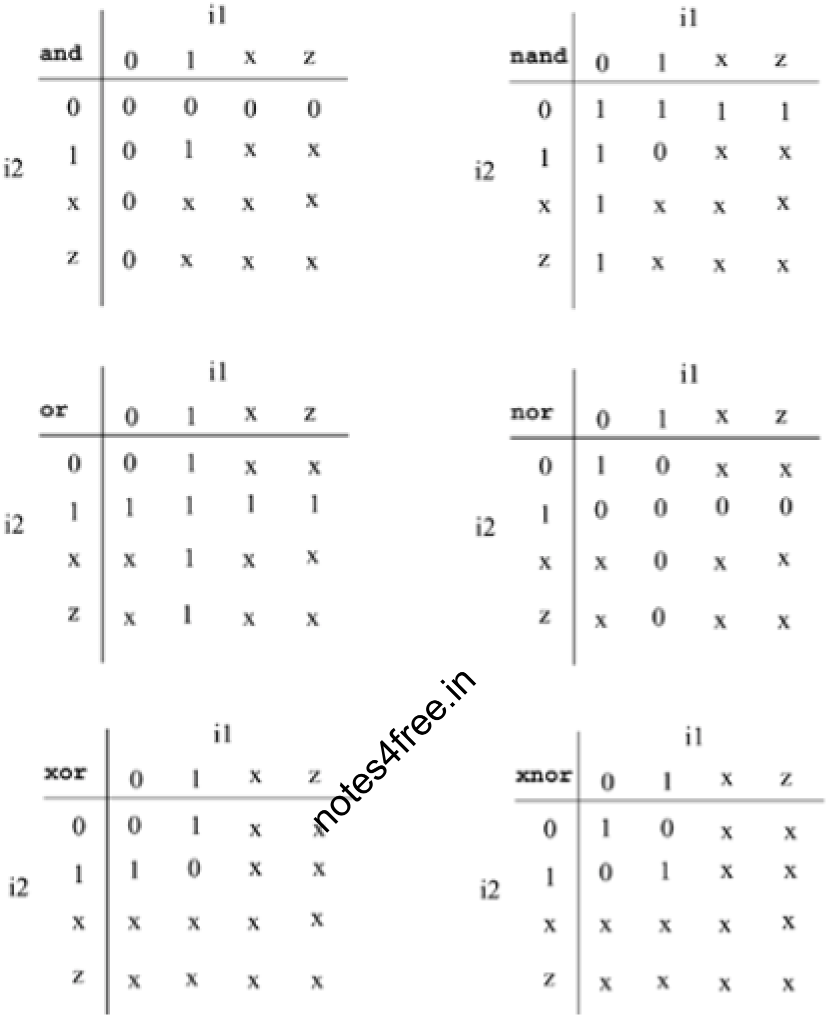
Figure 3-1. Basic Gates

Example 3-1 Gate Instantiation of And/Or Gates

```
wire OUT, IN1, IN2;

// basic gate instantiations.

and a1(OUT, IN1, IN2);

nand na1(OUT, IN1, IN2);

or or1(OUT, IN1, IN2);

nor nor1(OUT, IN1, IN2);

xor x1(OUT, IN1, IN2);

xnor nx1(OUT, IN1, IN2);

// More than two inputs; 3 input nand gate

nand na1_3inp(OUT, IN1, IN2, IN3);

// gate instantiation without instance name

and (OUT, IN1, IN2); // legal gate instantiation
```

The truth tables for these gates define how outputs for the gates are computed from the inputs. Truth tables are defined assuming two inputs. The truth tables for these gates are shown in Table 3-1. Outputs of gates with more than two inputs are computed by applying the truth table iteratively.

Table 3-1. Truth Tables for And/Or



		i1			
	and	0	1	x	z
i2	0	0	0	0	0
	1	0	1	x	x
	x	0	x	x	x
	z	0	x	x	x

		i1			
	nand	0	1	x	z
i2	0	1	1	1	1
	1	1	0	x	x
	x	1	x	x	x
	z	1	x	x	x

		i1			
	or	0	1	x	z
i2	0	0	1	x	x
	1	1	1	1	1
	x	x	1	x	x
	z	x	1	x	x

		i1			
	nor	0	1	x	z
i2	0	1	0	x	x
	1	0	0	0	0
	x	x	0	x	x
	z	x	0	x	x

		i1			
	xor	0	1	x	z
i2	0	0	1	x	x
	1	1	0	x	x
	x	x	x	x	x
	z	x	x	x	x

		i1			
	xnor	0	1	x	z
i2	0	1	0	x	x
	1	0	1	x	x
	x	x	x	x	x
	z	x	x	x	x

3.2.2 Buf/Not Gates

Buf/not gates have one scalar input and one or more scalar outputs. The last terminal in the port list is connected to the input. Other terminals are connected to the outputs. We will discuss gates that have one input and one output. Two basic buf/not gate primitives are provided in Verilog

buf not

The symbols for these logic gates are shown in Figure 3-2.

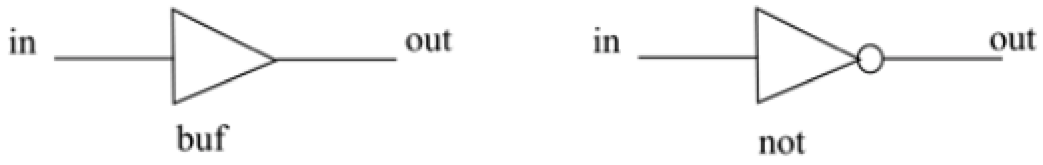


Figure 3-2. Buf/not Gates

These gates are instantiated in Verilog as shown Example 3-2. Notice that these gates can have multiple outputs but exactly one input, which is the last terminal in the port list.

Example 3-2 Gate Instantiations of Buf/Not Gates

```
// basic gate instantiations.

buf b1(OUT1, IN);

not n1(OUT1, IN);

// More than two outputs

buf b1_2out(OUT1, OUT2, IN);

// gate instantiation without instance name

not (OUT1, IN); // legal gate instantiation
```

Truth tables for gates with one input and one output are shown in Table 3-2.

Table 3-2. Truth Tables for Buf/Not Gates

buf	in	out
	0	0
	1	1
	x	x
	z	x

not	in	out
	0	1
	1	0
	x	x
	z	x

Bufif/notif

Gates with an additional control signal on buf and not gates are also available.

bufif1 notif1

bufif0 notif0

These gates propagate only if their control signal is asserted. They propagate z if their control signal is deasserted. Symbols for bufif/notif are shown in Figure 3-3.

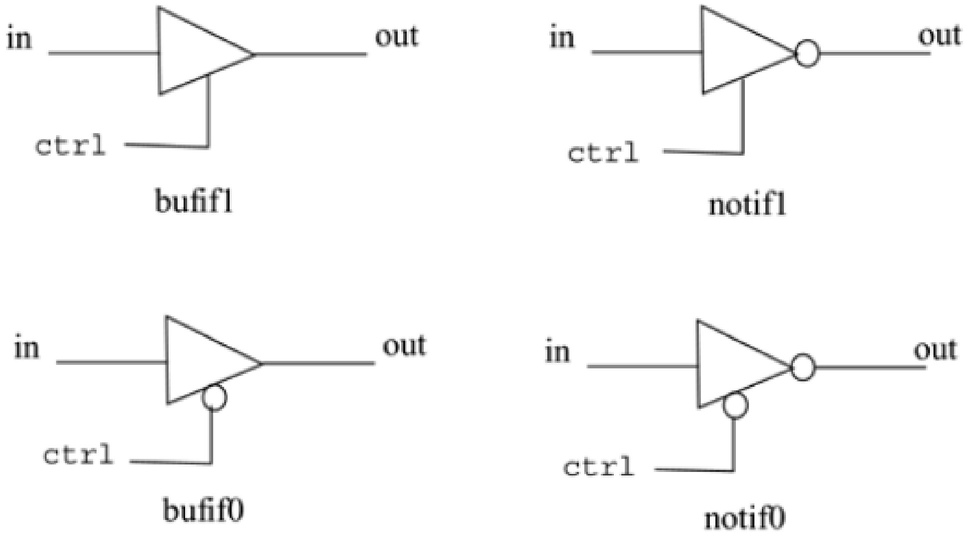
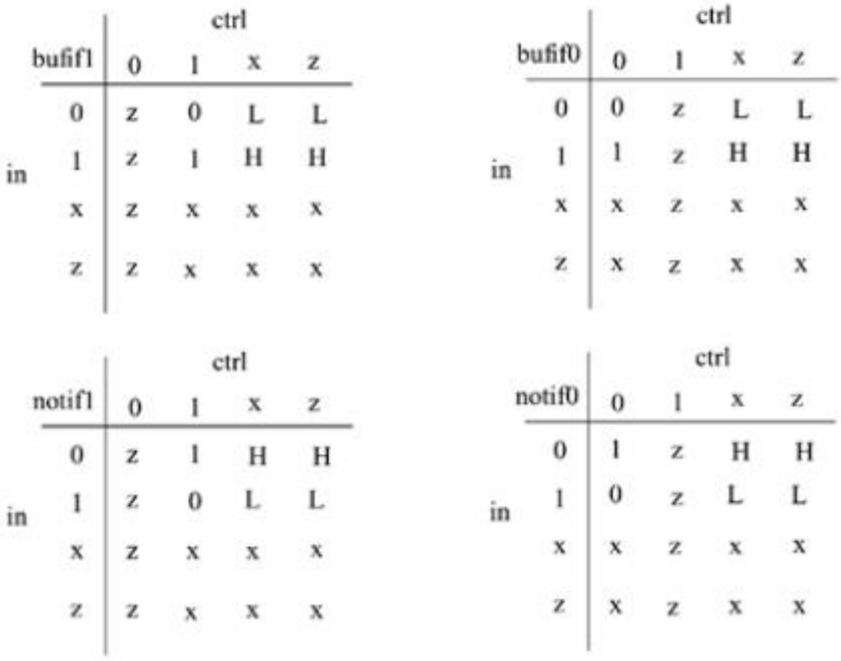


Figure 3-3 Bufif/notif Gates

The truth tables for these gates are shown in Table 3-3

Table 3-3. Truth Tables for Bufif/Notif Gates



		ctrl			
	bufif1	0	1	x	z
in	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	x	x	x

		ctrl			
	bufif0	0	1	x	z
in	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	x	z	x	x

		ctrl			
	notif1	0	1	x	z
in	0	z	1	H	H
	1	z	0	L	L
	x	z	x	x	x
	z	z	x	x	x

		ctrl			
	notif0	0	1	x	z
in	0	1	z	H	H
	1	0	z	L	L
	x	x	z	x	x
	z	x	z	x	x

These gates are used when a signal is to be driven only when the control signal is asserted. Such a situation is applicable when multiple drivers drive the signal. These drivers are designed to drive the signal on mutually exclusive control signals. Example 3-3 shows examples of instantiation of bufif and notif gates.

Example 3-3 Gate Instantiations of Bufif/Notif Gates

```
//Instantiation of bufif gates.
bufif1 b1 (out, in, ctrl);
bufif0 b0 (out, in, ctrl);
//Instantiation of notif gates
notif1 n1 (out, in, ctrl);
notif0 n0 (out, in, ctrl);
```

3.2.3 Array of Instances

There are many situations when repetitive instances are required. These instances differ from each other only by the index of the vector to which they are connected. To simplify specification of such instances, Verilog HDL allows an array of primitive instances to be defined. Example3-4 shows an example of an array of instances.

Example 3-4 Simple Array of Primitive Instances

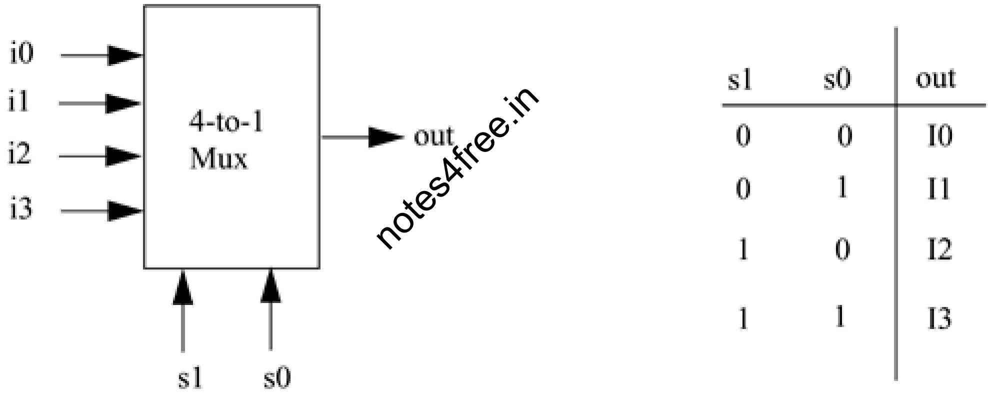
```
wire [7:0] OUT, IN1, IN2;
// basic gate instantiations.
nand n_gate[7:0](OUT, IN1, IN2);
// This is equivalent to the following 8 instantiations
nand n_gate0(OUT[0], IN1[0], IN2[0]);
nand n_gate1(OUT[1], IN1[1], IN2[1]);
nand n_gate2(OUT[2], IN1[2], IN2[2]);
nand n_gate3(OUT[3], IN1[3], IN2[3]);
nand n_gate4(OUT[4], IN1[4], IN2[4]);
nand n_gate5(OUT[5], IN1[5], IN2[5]);
nand n_gate6(OUT[6], IN1[6], IN2[6]);
nand n_gate7(OUT[7], IN1[7], IN2[7]);
```

3.1.4 Examples

Having understood the various types of gates available in Verilog, consider the real examples that illustrates design of gate-level digital circuits.

Gate-level multiplexer

Consider the design of 4-to-1 multiplexer with 2 select signals. Multiplexers serve a useful purpose in logic design. They can connect two or more sources to a single destination. They can also be used to implement Boolean functions. We will assume for this example that signals s1 and s0 do not get the value x or z. The I/O diagram and the truth table for the multiplexer are shown in Figure 3-4. The I/O diagram will be useful in setting up the port list for the multiplexer.



s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3

Figure 3-4. 4-to-1 Multiplexer

Implement the logic for the multiplexer using basic logic gates. The logic diagram for the multiplexer is shown in Figure 3-5.

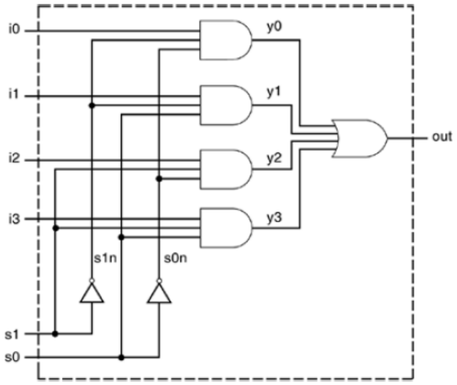


Figure 3-5. Logic Diagram for Multiplexer

The logic diagram has a one-to-one correspondence with the Verilog description. The Verilog description for the multiplexer is shown in Example 3-5. Two intermediate nets, s0n and s1n, are created; they are complements of input signals s1 and s0. Internal nets y0, y1, y2, y3 are also required. Note that instance names are not specified for primitive gates, not, and, and or. Instance names are optional for Verilog primitives but are mandatory for instances of user-defined modules.

Example 3-5 Verilog Description of Multiplexer

```
// Module 4-to-1 multiplexer. Port list is taken exactly from

// the I/O diagram.

module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram

output out;

input i0, i1, i2, i3;

input s1, s0;

// Internal wire declarations

wire s1n, s0n;

wire y0, y1, y2, y3;

// Gate instantiations

// Create s1n and s0n signals.

not (s1n, s1);

not (s0n, s0);

// 3-input and gates instantiated

and (y0, i0, s1n, s0n);

and (y1, i1, s1n, s0);

and (y2, i2, s1, s0n);

and (y3, i3, s1, s0);

// 4-input or gate instantiated

or (out, y0, y1, y2, y3);
```

```
endmodule
```

This multiplexer can be tested with the stimulus shown in Example 3-6. The stimulus checks that each combination of select signals connects the appropriate input to the output. The signal OUTPUT is displayed one time unit after it changes. System task $monitor could also be used to display the signals when they change values.

Example 3-6 Stimulus for Multiplexer

```
// Define the stimulus module (no ports)
module stimulus;
// Declare variables to be connected
// to inputs
reg IN0, IN1, IN2, IN3;
reg S1, S0;
// Declare output wire
wire OUTPUT;
// Instantiate the multiplexer
mux4_to_1 mymux(OUTPUT, IN0, IN1, IN2, IN3, S1, S0);
// Stimulate the inputs
// Define the stimulus module (no ports)
initial
begin
// set input lines
IN0 = 1; IN1 = 0; IN2 = 1; IN3 = 0;
#1 $display("IN0= %b, IN1= %b, IN2= %b, IN3= %b\n",IN0,IN1,IN2,IN3);
// choose IN0
S1 = 0; S0 = 0;
#1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
// choose IN1
```

```
S1 = 0; S0 = 1;

#1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);

// choose IN2

S1 = 1; S0 = 0;

#1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);

// choose IN3

S1 = 1; S0 = 1;

#1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);

end

endmodule
```

The output of the simulation is shown below. Each combination of the select signals is tested.

```
IN0= 1, IN1= 0, IN2= 1, IN3= 0

S1 = 0, S0 = 0, OUTPUT = 1

S1 = 0, S0 = 1, OUTPUT = 0

S1 = 1, S0 = 0, OUTPUT = 1

S1 = 1, S0 = 1, OUTPUT = 0
```

4-bit Ripple Carry Full Adder

Consider the design of a 4-bit full adder whose port list was defined in, List of Ports. We use primitive logic gates, and we apply stimulus to the 4-bit full adder to check functionality. For the sake of simplicity, we will implement a ripple carry adder. The basic building block is a 1-bit full adder. The mathematical equations for a 1-bit full adder are shown below.

sum = (a b cin)

cout = (a b) + cin (a b)

The logic diagram for a 1-bit full adder is shown in Figure 3-6.

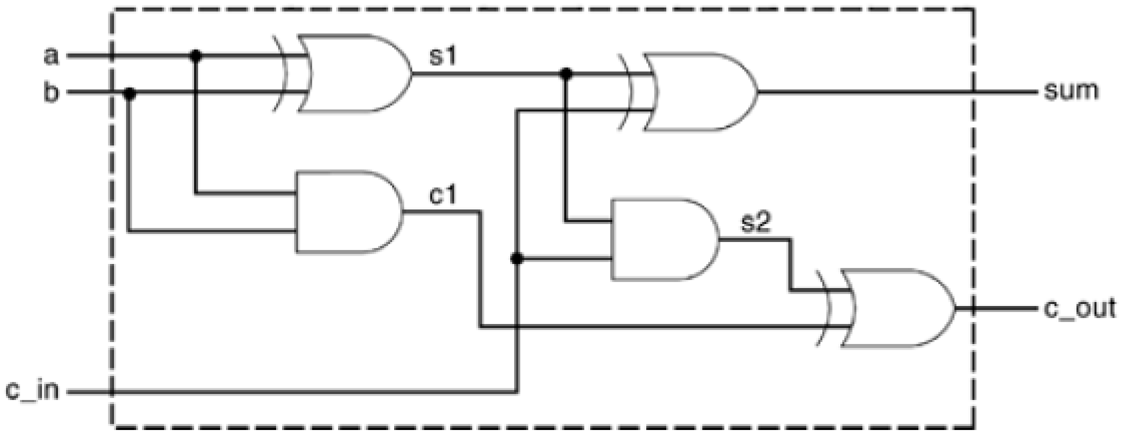


Figure 3-6. 1-bit Full Adder

This logic diagram for the 1-bit full adder is converted to a Verilog description, shown in Example 3-7.

Example 3-7 Verilog Description for 1-bit Full Adder

```
// Define a 1-bit full adder

module fulladd(sum, c_out, a, b, c_in);

// I/O port declarations

output sum, c_out;

input a, b, c_in;

// Internal nets

wire s1, c1, c2;

// Instantiate logic gate primitives

xor (s1, a, b);

and (c1, a, b);

xor (sum, s1, c_in);

and (c2, s1, c_in);

xor (c_out, c2, c1);

endmodule
```

A 4-bit ripple carry full adder can be constructed from four 1-bit full adders, as shown in Figure 3-7. Notice that fa0, fa1, fa2, and fa3 are instances of the module fulladd (1-bit full adder).

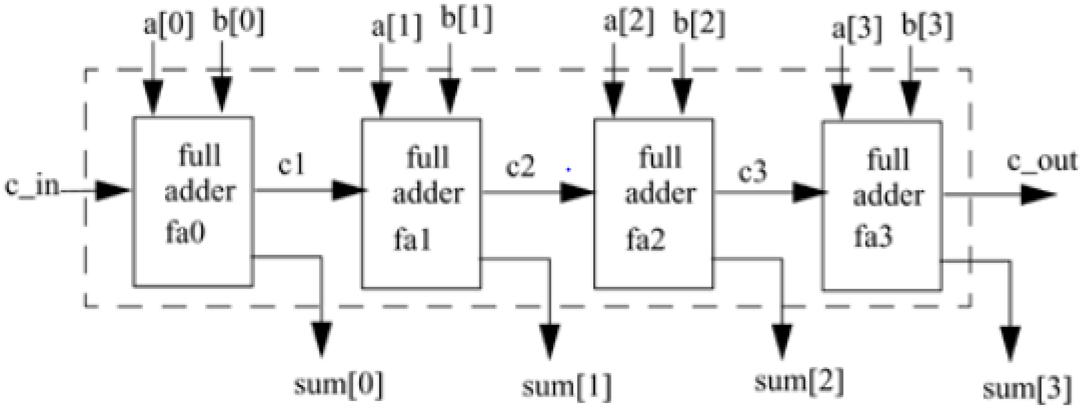


Figure 3-7. 4-bit Ripple Carry Full Adder

This structure can be translated to Verilog as shown in Example 3-8. Note that the port names used in a 1-bit full adder and a 4-bit full adder are the same but they represent different elements. The element sum in a 1-bit adder is a scalar quantity and the element sum in the 4-bit full adder is a 4-bit vector quantity. Verilog keeps names local to a module.

Names are not visible outside the module unless hierarchical name referencing is used. Also note that instance names must be specified when defined modules are instantiated, but when instantiating Verilog primitives, the instance names are optional.

Example 3-8 Verilog Description for 4-bit Ripple Carry Full Adder

```
// Define a 4-bit full adder

module fulladd4(sum, c_out, a, b, c_in);

// I/O port declarations

output [3:0] sum;

output c_out;

input[3:0] a, b;

input c_in;

// Internal nets

wire c1, c2, c3;

// Instantiate four 1-bit full adders.

fulladd fa0(sum[0], c1, a[0], b[0], c_in);
fulladd fa1(sum[1], c2, a[1], b[1], c1);
fulladd fa2(sum[2], c3, a[2], b[2], c2);
fulladd fa3(sum[3], c_out, a[3], b[3], c3);
endmodule
```

Finally, the design must be checked by applying stimulus, as shown in Example 3-9. The module stimulus stimulates the 4-bit full adder by applying a few input combinations and monitors the results.

Example 3-9 Stimulus for 4-bit Ripple Carry Full Adder

```
// Define the stimulus (top level module)
module stimulus;
// Set up variables
reg [3:0] A, B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;
// Instantiate the 4-bit full adder. call it FA1_4
fulladd4 FA1_4(SUM, C_OUT, A, B, C_IN);
// Set up the monitoring for the signal values
initial
begin
$monitor($time," A= %b, B=%b, C_IN= %b, --- C_OUT= %b, SUM= %b\n",
A, B, C_IN, C_OUT, SUM);
end
// Stimulate inputs
initial
begin
A = 4'd0; B = 4'd0; C_IN = 1'b0;
#5 A = 4'd3; B = 4'd4;
```

```
#5 A = 4'd2; B = 4'd5;

#5 A = 4'd9; B = 4'd9;

#5 A = 4'd10; B = 4'd15;

#5 A = 4'd10; B = 4'd5; C_IN = 1'b1;

end

endmodule
```

The output of the simulation is shown below.

```
0 A= 0000, B=0000, C_IN= 0, --- C_OUT= 0, SUM= 0000
5 A= 0011, B=0100, C_IN= 0, --- C_OUT= 0, SUM= 0111
10 A= 0010, B=0101, C_IN= 0, --- C_OUT= 0, SUM= 0111
15 A= 1001, B=1001, C_IN= 0, --- C_OUT= 1, SUM= 0010
20 A= 1010, B=1111, C_IN= 0, --- C_OUT= 1, SUM= 1001
25 A= 1010, B=0101, C_IN= 1,--- C_OUT= 1, SUM= 0000
```

3.3 Gate Delays

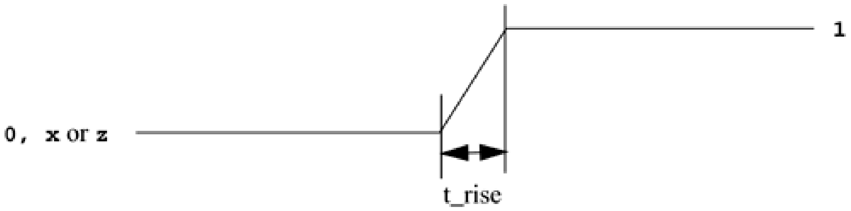
Until now, circuits are described without any delays (i.e., zero delay). In real circuits, logic gates have delays associated with them. Gate delays allow the Verilog user to specify delays through the logic circuits. Pin-to-pin delays can also be specified in Verilog.

3.3.1 Rise, Fall, and Turn-off Delays

There are three types of delays from the inputs to the output of a primitive gate.

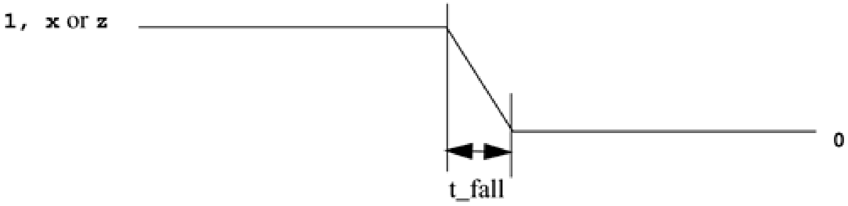
Rise delay

The rise delay is associated with a gate output transition to a 1 from another value.



Fall delay

The fall delay is associated with a gate output transition to a 0 from another value.



Turn-off delay

The turn-off delay is associated with a gate output transition to the high impedance value (z) from another value. If the value changes to x, the minimum of the three delays is considered.

Three types of delay specifications are allowed. If only one delay is specified, this value is used for all transitions. If two delays are specified, they refer to the rise and fall delay values. The turn-off delay is the minimum of the two delays. If all three delays are specified, they refer to rise, fall, and turn-off delay values. If no delays are specified, the default value is zero. Examples of delay specification are shown in Example 3-10.

Example 3-10 Types of Delay Specification

```
// Delay of delay_time for all transitions

and #(delay_time) a1(out, i1, i2);

// Rise and Fall Delay Specification.

and #(rise_val, fall_val) a2(out, i1, i2);

// Rise, Fall, and Turn-off Delay Specification

bufif0 #(rise_val, fall_val, turnoff_val) b1 (out, in, control);
```

Examples of delay specification are shown below.

```
and #(5) a1(out, i1, i2); //Delay of 5 for all transitions

and #(4,6) a2(out, i1, i2); // Rise = 4, Fall = 6

bufif0 #(3,4,5) b1 (out, in, control); // Rise = 3, Fall = 4, Turn-off= 5
```

3.3.2 Min/Typ/Max Values

Verilog provides an additional level of control for each type of delay mentioned above. For each type of delay?rise, fall, and turn-off?three values, min, typ, and max, can be specified. Any one value can be chosen at the start of the simulation. Min/typ/max values are used to model devices whose delays vary within a minimum and maximum range because of the IC fabrication process variations.

Min value

The min value is the minimum delay value that the designer expects the gate to have.

Typ val

The typ value is the typical delay value that the designer expects the gate to have.

Max value

The max value is the maximum delay value that the designer expects the gate to have. Min, typ, or max values can be chosen at Verilog run time. Method of choosing a min/typ/max value may vary for different simulators or operating systems. (For Verilog- XL , the values are chosen by specifying options +maxdelays, +typdelays, and +mindelays at run time. If no option is specified, the typical delay value is the default).

This allows the designers the flexibility of building three delay values for each transition into their design. The designer can experiment with delay values without modifying the design.

Examples of min, typ, and max value specification for Verilog-XL are shown in Example3-11.

Example 3-11 Min, Max, and Typical Delay Values

```
// One delay
// if +mindelays, delay= 4
// if +typdelays, delay= 5
// if +maxdelays, delay= 6
and #(4:5:6) a1(out, i1, i2);
// Two delays
// if +mindelays, rise= 3, fall= 5, turn-off = min(3,5)
// if +typdelays, rise= 4, fall= 6, turn-off = min(4,6)
// if +maxdelays, rise= 5, fall= 7, turn-off = min(5,7)
and #(3:4:5, 5:6:7) a2(out, i1, i2);
// Three delays
// if +mindelays, rise= 2 fall= 3 turn-off = 4
// if +typdelays, rise= 3 fall= 4 turn-off = 5
// if +maxdelays, rise= 4 fall= 5 turn-off = 6

and #(2:3:4, 3:4:5, 4:5:6) a3(out, i1,i2);
```

Examples of invoking the Verilog-XL simulator with the command-line options are shown below. Assume that the module with delays is declared in the file test.v.

```
//invoke simulation with maximum delay

> verilog test.v +maxdelays

//invoke simulation with minimum delay

> verilog test.v +mindelays

//invoke simulation with typical delay

> verilog test.v +typdelays
```

3.3.3 Delay Example

Let us consider a simple example to illustrate the use of gate delays to model timing in the logic circuits. A simple module called D implements the following logic equations:

out = (a b) + c

The gate-level implementation is shown in Module D (Figure 3-8). The module contains two gates with delays of 5 and 4 time units.

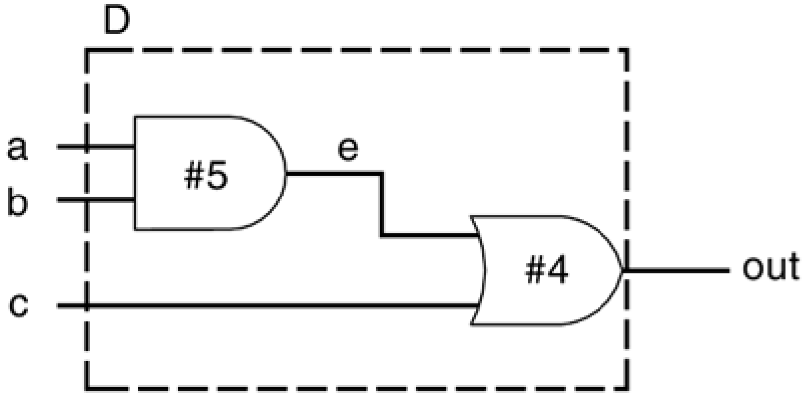


Figure 3-8. Module D

The module D is defined in Verilog as shown in Example 3-12.

Example 3-12 Verilog Definition for Module D with Delay

```
// Define a simple combination module called D
module D (out, a, b, c);
// I/O port declarations
output out;
input a,b,c;
// Internal nets
wire e;
// Instantiate primitive gates to build the circuit
and #(5) a1(e, a, b); //Delay of 5 on gate a1
or #(4) o1(out, e,c); //Delay of 4 on gate o1
endmodule
```

This module is tested by the stimulus file shown in Example 3-13.

Example 3-13 Stimulus for Module D with Delay

```
// Stimulus (top-level module)
module stimulus;
// Declare variables
reg A, B, C;
wire OUT;
// Instantiate the module D
D d1( OUT, A, B, C);
// Stimulate the inputs. Finish the simulation at 40 time units.
initial
begin
A= 1'b0; B= 1'b0; C= 1'b0;
#10 A= 1'b1; B= 1'b1; C= 1'b1;
```

```
#10 A= 1'b1; B= 1'b0; C= 1'b0;

#20 $finish;

end

endmodule
```

The waveforms from the simulation are shown in Figure 3-9 to illustrate the effect of specifying delays on gates. The waveforms are not drawn to scale. However, simulation time at each transition is specified below the transition.

1. The outputs E and OUT are initially unknown.

2. At time 10, after A, B, and C all transition to 1, OUT transitions to 1 after a delay of 4 time units and E changes value to 1 after 5 time units.

3. At time 20, B and C transition to 0. E changes value to 0 after 5 time units, and OUT transitions to 0, 4 time units after E changes.

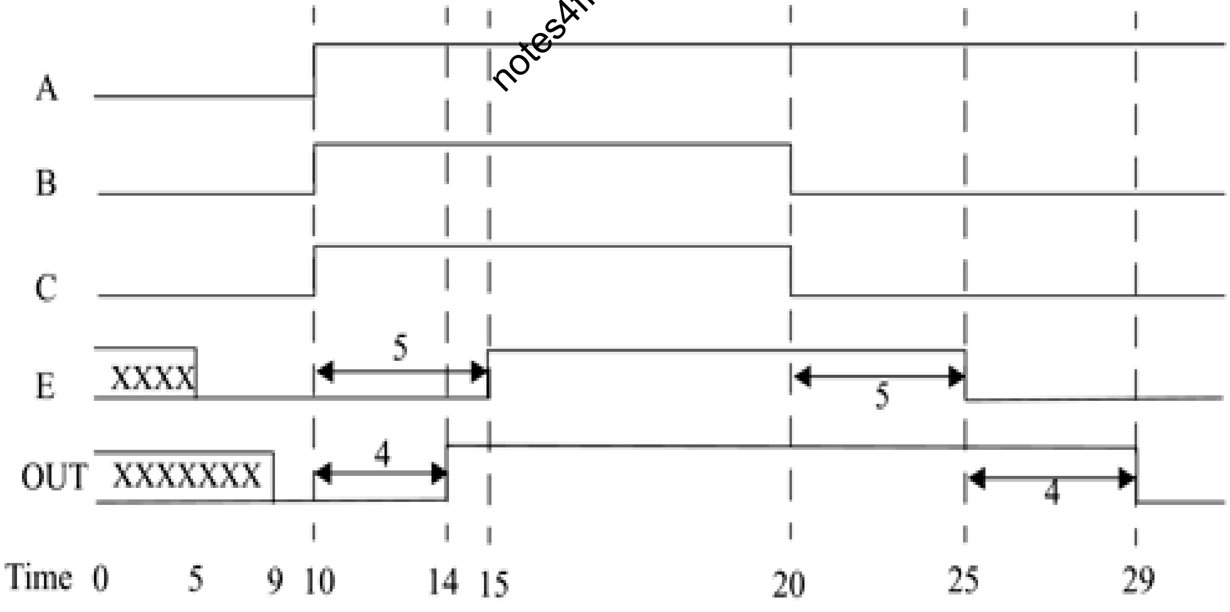


Figure 3-9. Waveforms for Delay Simulation of module D

It is a useful exercise to understand how the timing for each transition in the above waveform corresponds to the gate delays shown in Module D.

3.4 Dataflow Modeling

For small circuits, the gate-level modeling approach works very well because the number of gates is limited and the designer can instantiate and connects every gate individually. Also, gate-level modeling is very intuitive to a designer with a basic knowledge of digital logic design. However, in complex designs the number of gates is very large. Thus, designers can design more effectively if they concentrate on implementing the function at a level of abstraction higher than gate level. Dataflow modeling provides a powerful way to implement a design. Verilog allows a circuit to be designed in terms of the data flow between registers and how a design processes data rather than instantiation of individual gates.

3.4.1 Continuous Assignments

A continuous assignment is the most basic statement in dataflow modeling, used to drive a value onto a net. This assignment replaces gates in the description of the circuit and describes the circuit at a higher level of abstraction. The assignment statement starts with the keyword assign. The syntax of an assign statement is as follows.

continuous_assign ::= assign [drive_strength] [delay3] list_of_net_assignments ;

list_of_net_assignments ::= net_assignment { , net_assignment }

net_assignment ::= net_lvalue = expression

The default value for drive strength is strong1 and strong0. The delay value is also optional and can be used to specify delay on the assign statement. This is like specifying delays for gates. Continuous assignments have the following characteristics:

1. The left hand side of an assignment must always be a scalar or vector net or a concatenation of scalar and vector nets. It cannot be a scalar or vector register.

2. Continuous assignments are always active. The assignment expression is evaluated as soon as one of the right-hand-side operands changes and the value is assigned to the left-hand-side net.

3. The operands on the right-hand side can be registers or nets or function calls. Registers or nets can be scalars or vectors.

4. Delay values can be specified for assignments in terms of time units. Delay values are used to control the time when a net is assigned the evaluated value. This feature is similar to specifying delays for gates. It is very useful in modeling timing behavior in real circuits.

Examples of continuous assignments are shown below. Operators such as &, ^, |, {, } and + used in the examples, At this point, concentrate on how the assign statements are specified.

Example 3-14 Examples of Continuous Assignment

```
// Continuous assign. out is a net. i1 and i2 are nets.

assign out = i1 & i2;

// Continuous assign for vector nets. addr is a 16-bit vector net

// addr1 and addr2 are 16-bit vector registers.

assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];

// Concatenation. Left-hand side is a concatenation of a scalar

// net and a vector net.

assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
```

3.4.2 Implicit Continuous Assignment

Instead of declaring a net and then writing a continuous assignment on the net, Verilog provides a shortcut by which a continuous assignment can be placed on a net when it is declared. There can be only one implicit declaration assignment per net because a net is declared only once.

In the example below, an implicit continuous assignment is contrasted with a regular continuous assignment.

```
//Regular continuous assignment

wire out;

assign out = in1 & in2;

//Same effect is achieved by an implicit continuous assignment

wire out = in1 & in2;
```

Implicit Net Declaration

If a signal name is used to the left of the continuous assignment, an implicit net declaration will be inferred for that signal name. If the net is connected to a module port, the width of the inferred net is equal to the width of the module port.

```
// Continuous assign. out is a net.

wire i1, i2;

assign out = i1 & i2; //Note that out was not declared as a wire

//but an implicit wire declaration for out

//is done by the simulator
```

3.5 Delays

Delay values control the time between the change in a right-hand-side operand and when the new value is assigned to the left-hand side. Three ways of specifying delays in continuous assignment statements are regular assignment delay, implicit continuous assignment delay, and net declaration delay.

3.5.1 Regular Assignment Delay

The first method is to assign a delay value in a continuous assignment statement. The delay value is specified after the keyword assign. Any change in values of in1 or in2 will result in a delay of 10 time units before re-computation of the expression in1 & in2, and the result will be assigned to out. If in1 or in2 changes value again before 10 time units when the result propagates to out, the values of in1 and in2 at the time of re-computation are considered. This property is called inertial delay. An input pulse that is shorter than the delay of the assignment statement does not propagate to the output.

```
assign #10 out = in1 & in2; // Delay in a continuous assign
```

1. When signals in1 and in2 go high at time 20, out goes to a high 10 time units later (time = 30).

2. When in1 goes low at 60, out changes to low at 70.

3. However, in1 changes to high at 80, but it goes down to low before 10 time units have elapsed.

4. Hence, at the time of re-computation, 10 units after time 80, in1 is 0. Thus, out gets the value 0. A pulse of width less than the specified assignment delay is no propagated to the output.

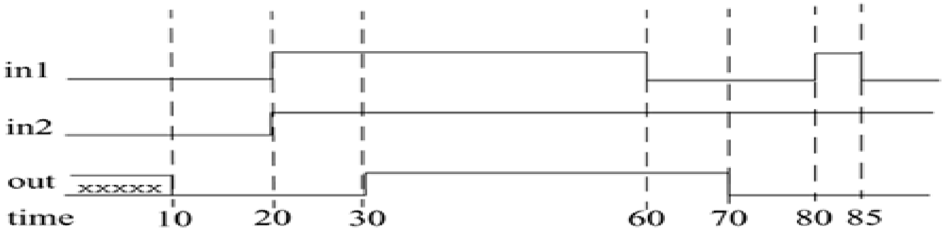


Figure 3-10. Waveforms for Delay Simulation

Inertial delays also apply to gate delays,

Implicit Continuous Assignment Delay

An equivalent method is to use an implicit continuous assignment to specify both a delay and an assignment on the net.

```
//implicit continuous assignment delay

wire #10 out = in1 & in2;

//same as

wire out;

assign #10 out = in1 & in2;
```

The declaration above has the same effect as defining a wire out and declaring a continuous assignment on out.

Net Declaration Delay

A delay can be specified on a net when it is declared without putting a continuous assignment on the net. If a delay is specified on a net out, then any value change applied to the net out is delayed accordingly. Net declaration delays can also be used in gate-level modeling.

```
//Net Delays

wire # 10 out;

assign out = in1 & in2;

//The above statement has the same effect as the following.

wire out;

assign #10 out = in1 & in2;
```

3.5 Expressions, Operators, and Operands

Dataflow modeling describes the design in terms of expressions instead of primitive gates. Expressions, operators, and operands form the basis of dataflow modeling.

Expressions are constructs that combine operators and operands to produce a result.

```
// Examples of expressions. Combines operands and operators

a ^ b
```

```
addr1[20:17] + addr2[20:17]

in1 | in2
```

Operands can be any one of the data types defined, Data Types. Some constructs will take only certain types of operands. Operands can be constants, integers, real numbers, nets, registers, times, bit-select (one bit of vector net or a vector register), part-select (selected bits of the vector net or register vector), and memories or function calls

```
integer count, final_count;

final_count = count + 1;//count is an integer operand

real a, b, c;

c = a - b; //a and b are real operands

reg [15:0] reg1, reg2;

reg [3:0] reg_out;

reg_out = reg1[3:0] ^ reg2[3:0];//reg1[3:0] and reg2[3:0] are

//part-select register operands

reg ret_value;

ret_value = calculate_parity(A, B);//calculate_parity is a

//function type operand
```

Operators

Operators act on the operands to produce desired results. Verilog provides various types of operators. Operator Types d1 && d2 // && is an operator on operands d1 and d2.

!a[0] // ! is an operator on operand a[0]

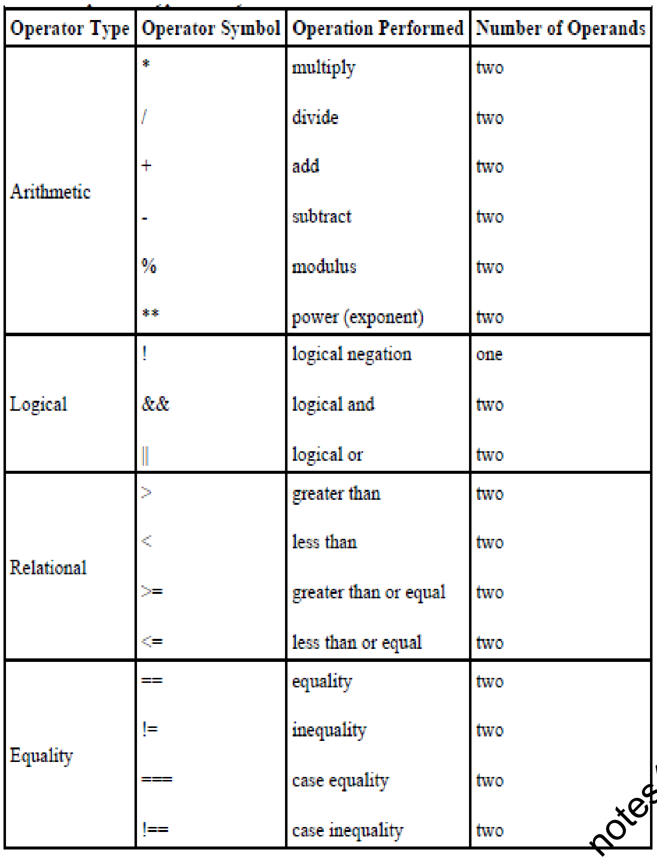
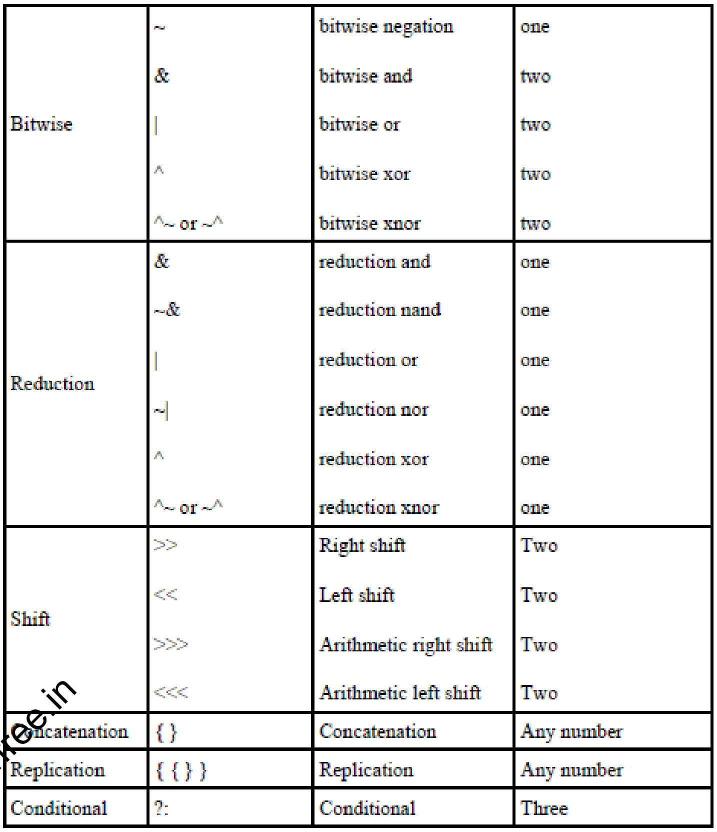
B >> 1 // >> is an operator on operands B and 1

Operator Types

Verilog provides many different operator types. Operators can be arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, or conditional. Some of these operators are similar to the operators used in the C programming language. Each operator type is denoted by a symbol. Table shows the complete listing of operator symbols classified by category.

.

Table 3-4 Operator Types and Symbols

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
	**	power (exponent)	two
Logical	!	logical negation	one
	&&	logical and	two
	||	logical or	two
Relational	>	greater than	two
	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two
Equality	==	equality	two
	!=	inequality	two
	===	case equality	two
	!==	case inequality	two
Bitwise	~	bitwise negation	one
	&	bitwise and	two
	|	bitwise or	two
	^	bitwise xor	two
	^~ or ~^	bitwise xnor	two
Reduction	&	reduction and	one
	~&	reduction nand	one
	|	reduction or	one
	~|	reduction nor	one
	^	reduction xor	one
	^~ or ~^	reduction xnor	one
Shift	>>	Right shift	Two
	<<	Left shift	Two
	>>>	Arithmetic right shift	Two
	<<<	Arithmetic left shift	Two
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number
Conditional	?:	Conditional	Three

Examples

A design can be represented in terms of gates, data flow, or a behavioral description. Consider the 4-to-1 multiplexer and 4-bit full adder described earlier. Previously, these designs were directly translated from the logic diagram into a gate-level Verilog description. Here, we describe the same designs in terms of data flow. We also discuss two additional examples: a 4-bit full adder using carry look ahead and a 4-bit counter using negative edge-triggered D-flip-flops.

4-to-1 Multiplexer

Gate-level modeling of a 4-to-1 multiplexer, Example. The logic diagram for the multiplexer is given in Figure 3.4 and the gate-level Verilog description is shown in Example. We describe the multiplexer, using dataflow statements. Compare it with the gate-level description. We show two methods to model the multiplexer by using dataflow statements.

Method 1: logic equation

We can use assignment statements instead of gates to model the logic equations of the multiplexer. Notice that everything is same as the gate-level Verilog description except that computation of out is done by specifying one logic equation by using operators instead of individual gate instantiations. I/O ports remain the same. This important so that the interface with the environment does not change. Only the internals of the module change.

Example 4-to-1 Multiplexer, Using Logic Equations

```
// Module 4-to-1 multiplexer using data flow. logic equation

// Compare to gate-level model

module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram

output out;

input i0, i1, i2, i3;

input s1, s0;

//Logic equation for out

assign out = (~s1 & ~s0 & i0)|

(~s1 & s0 & i1) |

(s1 & ~s0 & i2) |

(s1 & s0 & i3) ;

endmodule
```

Method 2: conditional operator

There is a more concise way to specify the 4-to-1 multiplexers.

Example of 4-to-1 Multiplexer, Using Conditional Operators

```
// Module 4-to-1 multiplexer using data flow. Conditional operator.

// Compare to gate-level model

module multiplexer4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram

output out;

input i0, i1, i2, i3;

input s1, s0;
```

```
// Use nested conditional operator

assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ;

endmodule
```

In the simulation of the multiplexer, the gate-level module can be substituted with the dataflow multiplexer modules described above. The stimulus module will not change. The simulation results will be identical. By encapsulating functionality inside a module, we can replace the gate-level module with a dataflow module without affecting the other modules in the simulation. This is a very powerful feature of Verilog.

4 bit Full Adder

The 4-bit full adder in, Examples, was designed by using gates; the logic diagram is shown in Figure 3.7. In this section, we write the dataflow description for the 4-bit adder. In gates, we had to first describe a 1-bit full adder. Then we built a 4-bit full ripple carry adder. We again illustrate two methods to describe a 4-bit full adder by means of dataflow statements.

Method 1: dataflow operators

A concise description of the adder is defined with the + and { } operators.

Example 4-bit Full Adder, Using Dataflow Operators

```
// Define a 4-bit full adder by using dataflow statements.

module fulladd4(sum, c_out, a, b, c_in);

// I/O port declarations

output [3:0] sum;

output c_out;

input[3:0] a, b;

input c_in;

// Specify the function of a full adder

assign {c_out, sum} = a + b + c_in;

endmodule
```

If we substitute the gate-level 4-bit full adder with the dataflow 4-bit full adder, the rest of the modules will not change. The simulation results will be identical.

Method 2: full adder with carry lookahead

In ripple carry adders, the carry must propagate through the gate levels before the sum is available at the output terminals. An n-bit ripple carry adder will have 2n gate levels. The propagation time can be a limiting factor on the speed of the circuit. One of the most popular methods to reduce delay is to use a carry lookahead mechanism. Logic equations for implementing the carry lookahead mechanism can be found in any logic design book. The propagation delay is reduced to four gate levels, irrespective of the number of bits in the adder. The Verilog description for a carry lookahead adder. This module can be substituted in place of the full adder modules described before without changing any other component of the simulation. The simulation results will be unchanged.

Example 4-bit Full Adder with Carry Lookahead

```
module fulladd4(sum, c_out, a, b, c_in);

// Inputs and outputs

output [3:0] sum;

output c_out;

input [3:0] a,b;

input c_in;

// Internal wires

wire p0,g0, p1,g1, p2,g2, p3,g3;

wire c4, c3, c2, c1;

// compute the p for each stage

assign p0 = a[0] ^ b[0],

p1 = a[1] ^ b[1],

p2 = a[2] ^ b[2],

p3 = a[3] ^ b[3];

// compute the g for each stage

assign g0 = a[0] & b[0],

g1 = a[1] & b[1],

g2 = a[2] & b[2],

g3 = a[3] & b[3];

// compute the carry for each stage

// Note that c_in is equivalent c0 in the arithmetic equation for
// carry lookahead computation
assign c1 = g0 | (p0 & c_in),
c2 = g1 | (p1 & g0) | (p1 & p0 & c_in),
c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & c_in),
c4 = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0) |
(p3 & p2 & p1 & p0 & c_in);
// Compute Sum
assign sum[0] = p0 ^ c_in,
sum[1] = p1 ^ c1,
sum[2] = p2 ^ c2,
sum[3] = p3 ^ c3;
// Assign carry output
assign c_out = c4;
endmodule
```

Ripple Counter

Consider the design of a 4-bit ripple counter by using negative edge-triggered flipflops. This example was discussed at a very abstract level, Hierarchical Modeling Concepts. We design it using Verilog dataflow statements and test it with a stimulus module. The diagrams for the 4-bit ripple carry counter modules are show the counter being built with four T-flipflops.

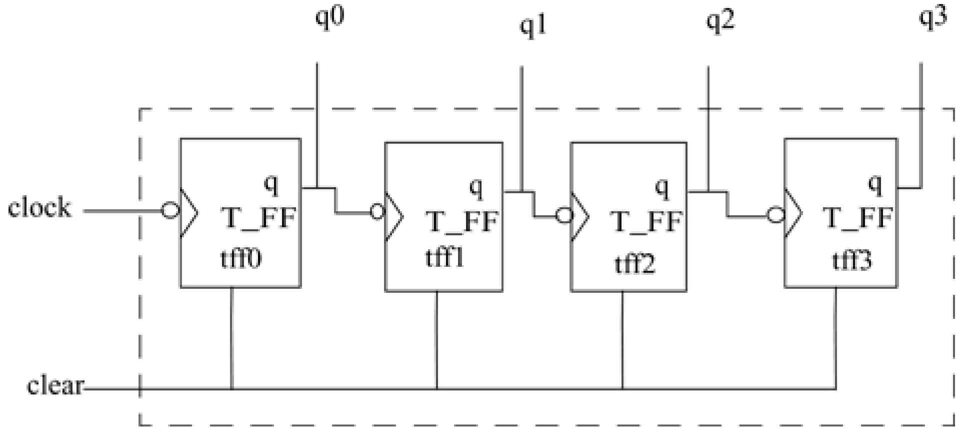


Figure 3.11 4 bit ripple counter

.

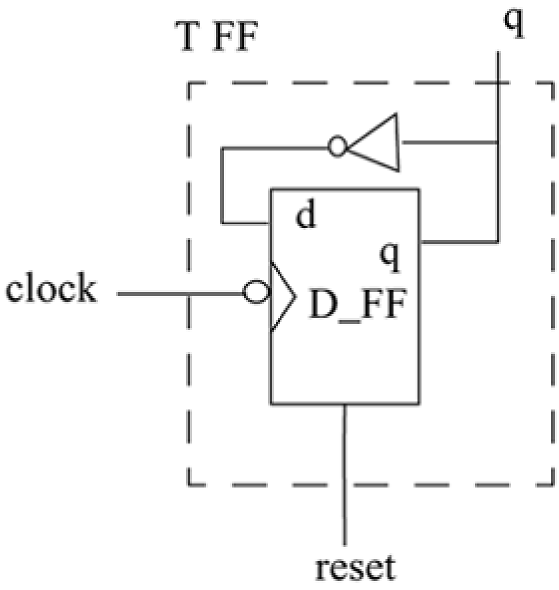


Figure 3.12 T-flipflop is built with one D-flipflop and an inverter gate

Figure 3.13 shows the D-flipflop constructed from basic logic gates.

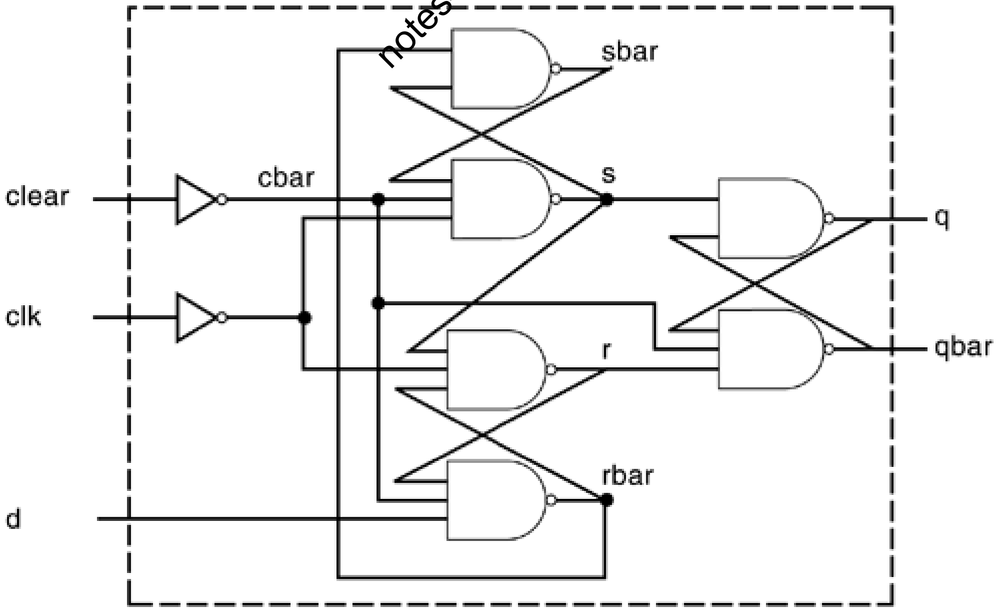


Figure 3.13 Negative Edge-Triggered D-flipflop with Clear

Given the above diagrams, we write the corresponding Verilog, using dataflow statements in a top-down fashion. First we design the module counter. The code is shown in. The code contains instantiation of four T_FF modules.

Example: Verilog Code for Ripple Counter

```
// Ripple counter
```

```
module counter(Q , clock, clear);
// I/O ports
output [3:0] Q;
input clock, clear;
// Instantiate the T flipflops
T_FF tff0(Q[0], clock, clear);
T_FF tff1(Q[1], Q[0], clear);
T_FF tff2(Q[2], Q[1], clear);
T_FF tff3(Q[3], Q[2], clear);
endmodule
```

Example :Verilog Code for T-flipflop

```
// Edge-triggered T-flipflop. Toggles every clock
// cycle.
module T_FF(q, clk, clear);
// I/O ports
output q;
input clk, clear;
// Instantiate the edge-triggered DFF
// Complement of output q is fed back.
// Notice qbar not needed. Unconnected port.
edge_dff ff1(q, ,~q, clk, clear);
endmodule
```

Verilog Code for Edge-Triggered D-flipflop

```
// Edge-triggered D flipflop
module edge_dff(q, qbar, d, clk, clear);
// Inputs and outputs
output q,qbar;
input d, clk, clear;
// Internal variables
wire s, sbar, r, rbar,cbar;
```

```
// dataflow statements

//Create a complement of signal clear

assign cbar = ~clear;

// Input latches; A latch is level sensitive. An edge-sensitive

// flip-flop is implemented by using 3 SR latches.

assign sbar = ~(rbar & s),

s = ~(sbar & cbar & ~clk),

r = ~(rbar & ~clk & s),

rbar = ~(r & cbar & d);

// Output latch

assign q = ~(s & qbar),

qbar = ~(q & r & cbar);

endmodule
```

Stimulus Module for Ripple Counter

```
// Top level stimulus module

module stimulus;

// Declare variables for stimulating input

reg CLOCK, CLEAR;

wire [3:0] Q;

initial

$monitor($time, " Count Q = %b Clear= %b", Q[3:0],CLEAR);

// Instantiate the design block counter

counter c1(Q, CLOCK, CLEAR);

// Stimulate the Clear Signal

initial

begin

CLEAR = 1'b1;

#34 CLEAR = 1'b0;

#200 CLEAR = 1'b1;

#50 CLEAR = 1'b0;
```

```
end

// Set up the clock to toggle every 10 time units

initial

begin

CLOCK = 1'b0;

forever #10 CLOCK = ~CLOCK;

end

// Finish the simulation at time 400

initial

begin

#400 $finish;

end

endmodule
```

The output of the simulation is shown below. Note that the clear signal resets the count to zero.

```
0 Count Q = 0000 Clear= 1

34 Count Q = 0000 Clear= 0

40 Count Q = 0001 Clear= 0

60 Count Q = 0010 Clear= 0

80 Count Q = 0011 Clear= 0

100 Count Q = 0100 Clear= 0

120 Count Q = 0101 Clear= 0

140 Count Q = 0110 Clear= 0

160 Count Q = 0111 Clear= 0

180 Count Q = 1000 Clear= 0

200 Count Q = 1001 Clear= 0

220 Count Q = 1010 Clear= 0

234 Count Q = 0000 Clear= 1

284 Count Q = 0000 Clear= 0

300 Count Q = 0001 Clear= 0

320 Count Q = 0010 Clear= 0
```

```
340 Count Q = 0011 Clear= 0

360 Count Q = 0100 Clear= 0

380 Count Q = 0101 Clear= 0
```

3.6: Outcomes

After completion of the module the students are able to:

- Identify logic gate primitives provided in Verilog and Understand instantiation of gates, gate symbols, and truth tables for and/or and buf/not type gates.
- Understand how to construct a Verilog description from the logic diagram of the circuit.
- Describe rise, fall, and turn-off delays in the gate-level design and Explain min, max, and typ delays in the gate-level design
- Describe the continuous assignment (assign) statement, restrictions on the assign statement, and the implicit continuous assignment statement.
- Explain assignment delay, implicit assignment delay, and net declaration delay for continuous assignment statements and Define expressions, operators, and operands.
- Use dataflow constructs to model practical digital circuits in Verilog

3.7: Recommended questions

1. Write the truth table of all the basic gates. Input values consisting of ‘0’, ‘1’, ‘x’, ‘z’.
2. What are the primitive gates supported by Verilog HDL? Write the Verilog HDL statements to instantiate all the primitive gates.
3. Use gate level description of Verilog HDL to design 4 to 1 multiplexer. Write truth table, top-level block, logic expression and logic diagram. Also write the stimulus block for the same.
4. Explain the different types of buffers and not gates with the help of truth table, logic symbol, logic expression
5. Use gate level description of Verilog HDL to describe the 4-bit ripple carry counter. Also write a stimulus block for 4-bit ripple carry adder.
6. How to model the delays of a logic gate using Verilog HDL? Give examples. Also explain the different delays associated with digital circuits.
7. Write gate level description to implement function y = a.b + c, with 5 and 4 time units of gate delay for AND and OR gate respectively. Also write the stimulus block and simulation waveform.
8. With syntax describe the continuous assignment statement.

9. Show how different delays associated with logic circuit are modelled using dataflow description.

10. Explain different operators supported by Verilog HDL.

11. What is an expression associated with dataflow description? What are the different types of operands in an expression?

12. Discuss the precedence of operators.

13. Use dataflow description style of Verilog HDL to design 4:1 multiplexer with and without using conditional operator.

14. Use dataflow description style of Verilog HDL to design 4-bitadder
using i. Ripple carry logic.
ii. Carry look ahead logic.

15. Use dataflow description style, gate level description of Verilog HDL to design 4-bit ripple carry counter. Also write the stimulus block to verify the same.

MODULE -4

BEHAVIORAL MODELING

4.1 Objectives

- To Explain the significance of structured procedures always and initial in behavioral modeling.
- To Define blocking and nonblocking procedural assignments.
- To Understand delay-based timing control mechanism in behavioral modeling. Use regular delays, intra-assignment delays, and zero delays.
- To Describe event-based timing control mechanism in behavioral modeling. Use regular event control, named event control, and event OR control.
- To Use level-sensitive timing control mechanism in behavioral modeling.
- To Explain conditional statements using if and else.
- To Describe multiway branching, using case, casex, and casez statements.
- To Understand looping statements such as while, for, repeat, and forever.
- To Define sequential and parallel blocks.

4.2 Structured Procedures

There are two structured procedure statements in Verilog: always and initial. These statements are the two most basic statements in behavioral modeling. All other behavioral statements can appear only inside these structured procedure statements. Verilog is a concurrent programming language unlike the C programming language, which is sequential in nature.

Activity flows in Verilog run in parallel rather than in sequence. Each always and initial statement represents a separate activity flow in Verilog. Each activity flow starts at simulation time 0. The statements always and initial cannot be nested. The fundamental difference between the two statements is explained in the following sections

4.2.1 Initial Statement

All statements inside an initial statement constitute an initial block. An initial block starts at time 0, executes exactly once during a simulation, and then does not execute again. If there are multiple initial blocks, each block starts to execute concurrently at time 0. Each block finishes execution independently of other blocks.

Multiple behavioral statements must be grouped, typically using the keywords begin and end. If there is only one behavioral statement, grouping is not necessary. This is similar to the begin-end blocks in Pascal programming language or the { } grouping in the C programming language. Example 4.1 illustrates the use of the initial statement.

Example 4.1:Initial Statement

```
module stimulus;

reg x,y, a,b, m;

initial

m = 1'b0; //single statement; does not need to be grouped

initial

begin

#5 a = 1'b1; //multiple statements; need to be grouped

#25 b = 1'b0;

end

initial

begin

#10 x = 1'b0;

#25 y = 1'b1;

end

initial

128

#50 $finish;

endmodule
```

In the above example, the three initial statements start to execute in parallel at time 0. If a delay #<delay> is seen before a statement, the statement is executed <delay> time units after the current simulation time. Thus, the execution sequence of the statements inside the initial blocks will be as follows.

```
time statement executed

0 m = 1'b0;

5 a = 1'b1;

10 x = 1'b0;
```

```
30 b = 1'b0;

35 y = 1'b1;

50 $finish;
```

The initial blocks are typically used for initialization, monitoring, waveforms and other processes that must be executed only once during the entire simulation run. The following subsections discussion how to initialize values using alternate shorthand syntax. The use of such shorthand syntax has the same effect as an initial block combined with a variable declaration.

Combined Variable Declaration and Initialization

Variables can be initialized when they are declared. Example 4-2 shows such a declaration.

Example 4-2 Initial Value Assignment

```
//The clock variable is defined first

reg clock;

//The value of clock is set to 0

initial clock = 0;

//Instead of the above method, clock variable

//can be initialized at the time of declaration

//This is allowed only for variables declared

//at module level.

reg clock = 0;
```

Combined Port/Data Declaration and Initialization

The combined port/data declaration can also be combined with an initialization. Example 4-3 shows such a declaration.

Example 4-3 Combined Port/Data Declaration and Variable Initialization

```
module adder (sum, co, a, b, ci);

output reg [7:0] sum = 0; //Initialize 8 bit output sum

output reg co = 0; //Initialize 1 bit output co

input [7:0] a, b;

input ci;
```

```
--

--

endmodule
```

Combined ANSI C Style Port Declaration and Initialization

ANSI C style port declaration can also be combined with an initialization. Example 4-4 shows such a declaration.

Example 4-4 Combined ANSI C Port Declaration and Variable Initialization

```
module adder (output reg [7:0] sum = 0, //Initialize 8 bit output

output reg co = 0, //Initialize 1 bit output co

input [7:0] a, b,

input ci

);

--

--

endmodule
```

4.2.2 Always Statement

All behavioral statements inside an always statement constitute an always block. The always statement starts at time 0 and executes the statements in the always block continuously in a looping fashion. This statement is used to model a block of activity that is repeated continuously in a digital circuit. An example is a clock generator module that toggles the clock signal every half cycle. In real circuits, the clock generator is active from time 0 to as long as the circuit is powered on. Example 4-5 illustrates one method to model a clock generator in Verilog.

Example 4-5 always Statement

```
module clock_gen (output reg clock);

//Initialize clock at time zero

initial

clock = 1'b0;

//Toggle clock every half-cycle (time period = 20)

always
```

```
#10 clock = ~clock;
initial
#1000 $finish;
endmodule
```

In Example 4-5, the always statement starts at time 0 and executes the statement clock = ~clock every 10 time units. Notice that the initialization of clock has to be done inside a separate initial statement. If we put the initialization of clock inside the always block, clock will be initialized every time the always is entered. Also, the simulation must be halted inside an initial statement. If there is no $stop or $finish statement to halt the simulation, the clock generator will run forever. C programmers might draw an analogy between the always block and an infinite loop.

But hardware designers tend to view it as a continuously repeated activity in a digital circuit starting from power on. The activity is stopped only by power off ($finish) or by an interrupt ($stop).

4.3 Procedural Assignments

Procedural assignments update values of reg, integer, real, or time variables. The value placed on a variable will remain unchanged until another procedural assignment updates the variable with a different value. These are unlike continuous assignments, Dataflow Modeling, where one assignment statement can cause the value of

the right-hand-side expression to be continuously placed onto the left-hand-side net. The

syntax for the simplest form of procedural assignment is shown below.

assignment ::= variable_lvalue = [delay_or_event_control] expression

The left-hand side of a procedural assignment <lvalue> can be one of the following:

- A reg, integer, real, or time register variable or a memory element

- A bit select of these variables (e.g., addr[0])

- A part select of these variables (e.g., addr[31:16])

- A concatenation of any of the above

The right-hand side can be any expression that evaluates to a value. In behavioral modeling, all operators can be used in behavioral expressions.

There are two types of procedural assignment statements: blocking and nonblocking.

4.3.1 Blocking Assignments

Blocking assignment statements are executed in the order they are specified in a sequential block. A blocking assignment will not block execution of statements that follow in a parallel block. The = operator is used to specify blocking assignments.

Example 4-6 Blocking Statements

```
reg x, y, z;

reg [15:0] reg_a, reg_b;

integer count;

//All behavioral statements must be inside an initial or always block

initial

begin

x = 0; y = 1; z = 1; //Scalar assignments

count = 0; //Assignment to integer variables

reg_a = 16'b0; reg_b = reg_a; //initialize vectors

#15 reg_a[2] = 1'b1; //Bit select assignment with delay

#10 reg_b[15:13] = {x, y, z} //Assign result of concatenation to  part select of a vector

count = count + 1; //Assignment to an integer (increment)

end
```

In Example 4-6, the statement y = 1 is executed only after x = 0 is executed. The behavior in a particular block is sequential in a begin-end block if blocking statements are used, because the statements can execute only in sequence. The statement count = count + 1 is executed last. The simulation times at which the statements are executed are as follows:

- All statements x = 0 through reg_b = reg_a are executed at time 0

- Statement reg_a[2] = 0 at time = 15

- Statement reg_b[15:13] = {x, y, z} at time = 25

- Statement count = count + 1 at time = 25

- Since there is a delay of 15 and 10 in the preceding statements, count = count + 1 will be executed at time = 25 units

Note that for procedural assignments to registers, if the right-hand side has more bits than the register variable, the right-hand side is truncated to match the width of the register variable. The least significant bits are selected and the most significant bits are discarded. If the right-hand side has fewer bits, zeros are filled in the most significant bits of the register variable.

4.3.2 Nonblocking Assignments

Nonblocking assignments allow scheduling of assignments without blocking execution of the statements that follow in a sequential block. A <= operator is used to specify nonblocking assignments. Note that this operator has the same symbol as a relational operator, less_than_equal_to. The operator <= is interpreted as a relational operator in an expression and as an assignment operator in the context of a nonblocking assignment. To illustrate the behavior of nonblocking statements and its difference from blocking statements, let us consider Example 4-7, where we convert some blocking assignments to nonblocking assignments, and observe the behavior.

Example 4-7 Nonblocking Assignments

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
//All behavioral statements must be inside an initial or always block
initial
begin
x = 0; y = 1; z = 1; //Scalar assignments
count = 0; //Assignment to integer variables
reg_a = 16'b0; reg_b = reg_a; //Initialize vectors
reg_a[2] <= #15 1'b1; //Bit select assignment with delay
reg_b[15:13] <= #10 {x, y, z}; //Assign result of concatenation
//to part select of a vector
count <= count + 1; //Assignment to an integer (increment)
end
```

In this example, the statements x = 0 through reg_b = reg_a are executed sequentially at time 0. Then the three nonblocking assignments are processed at the same simulation time.

1. reg_a[2] = 0 is scheduled to execute after 15 units (i.e., time = 15)

2. reg_b[15:13] = {x, y, z} is scheduled to execute after 10 time units (i.e., time = 10)

3. count = count + 1 is scheduled to be executed without any delay (i.e., time = 0) Thus, the simulator schedules a nonblocking assignment statement to execute and continues to the next statement in the block without waiting for the nonblocking statement to complete execution. Typically, nonblocking assignment statements are

executed last in the time step in which they are scheduled, that is, after all the blocking assignments in that time step are executed.

In the example above, we mixed blocking and nonblocking assignments to illustrate their behavior. However, it is recommended that blocking and nonblocking assignments not be mixed in the same always block.

Application of nonblocking assignments

Having described the behavior of nonblocking assignments, it is important to understand why they are used in digital design. They are used as a method to model several concurrent data transfers that take place after a common event. Consider the following example where three concurrent data transfers take place at the positive edge of clock.

```
always @(posedge clock)

begin

reg1 <= #1 in1;

reg2 <= @(negedge clock) in2 ^ in3;

reg3 <= #1 reg1; //The old value of reg1

end
```

At each positive edge of clock, the following sequence takes place for the nonblocking assignments.

1. A read operation is performed on each right-hand-side variable, in1, in2, in3, and reg1, at the positive edge of clock. The right-hand-side expressions are evaluated, and the results are stored internally in the simulator.

2. The write operations to the left-hand-side variables are scheduled to be executed at the time specified by the intra-assignment delay in each assignment, that is, schedule "write" to reg1 after 1 time unit, to reg2 at the next negative edge of clock, and to reg3 after 1 time unit.

3. The write operations are executed at the scheduled time steps. The order in which the write operations are executed is not important because the internally stored right-hand-side expression values are used to assign to the left-hand-side values. For example, note that reg3 is assigned the old value of reg1 that was stored after the read operation, even if the write operation wrote a new value to reg1 before the write operation to reg3 was executed.

Thus, the final values of reg1, reg2, and reg3 are not dependent on the order in which the assignments are processed.

To understand the read and write operations further, consider Example 4-8, which is intended to swap the values of registers a and b at each positive edge of clock, using two concurrent always blocks.

Example 4-8 Nonblocking Statements to Eliminate Race Conditions

```
//Illustration 1: Two concurrent always blocks with blocking

//statements

always @(posedge clock)

a = b;

always @(posedge clock)

b = a;

135

//Illustration 2: Two concurrent always blocks with nonblocking

//statements

always @(posedge clock)

a <= b;

always @(posedge clock)

b <= a;
```

In Example 4-8, in Illustration 1, there is a race condition when blocking statements are used. Either a = b would be executed before b = a, or vice versa, depending on the simulator implementation. Thus, values of registers a and b will not be swapped. Instead, both registers will get the same value (previous value of a or b), based on the Verilog simulator implementation.

However, nonblocking statements used in Illustration 2 eliminate the race condition. At the positive edge of clock, the values of all right-hand-side variables are "read," and the right-hand-side expressions are evaluated and stored in temporary variables. During the write operation, the values stored in the temporary variables are assigned to the left-handside variables. Separating the read and write operations ensures that the values of registers a and b are swapped correctly, regardless of the order in which the write operations are performed. Example 4-9 shows how nonblocking assignments shown in Illustration 2 could be emulated using blocking assignments.

Example 4-9 Implementing Nonblocking Assignments using Blocking Assignments

```
//Emulate the behavior of nonblocking assignments by

//using temporary variables and blocking assignments

always @(posedge clock)

begin

//Read operation

//store values of right-hand-side expressions in temporary variables

temp_a = a;

temp_b = b;

//Write operation

//Assign values of temporary variables to left-hand-side variables

a = temp_b;

b = temp_a;

end
```

For digital design, use of nonblocking assignments in place of blocking assignments is highly recommended in places where concurrent data transfers take place after a common event. In such cases, blocking assignments can potentially cause race conditions because the final result depends on the order in which the assignments are evaluated. Nonblocking assignments can be used effectively to model concurrent data transfers because the final result is not dependent on the order in which the assignments are evaluated. Typical applications of nonblocking assignments include pipeline modeling and modeling of several mutually exclusive data transfers. On the downside, nonblocking assignments can potentially cause degradation in the simulator performance and increase in memory usage.

4.4 Timing Controls

Various behavioral timing control constructs are available in Verilog. In Verilog, if there are no timing control statements, the simulation time does not advance. Timing controls provide a way to specify the simulation time at which procedural statements will execute.

There are three methods of timing control: delay-based timing control, event-based timing control, and level-sensitive timing control.

4.4.1 Delay-Based Timing Control

Delay-based timing control in an expression specifies the time duration between when the statement is encountered and when it is executed. We used delay-based timing control statements when writing few modules in the preceding chapters but did not explain them in detail. In this section, we will discuss delay-based timing control statements. Delays are specified by the symbol #. Syntax for the delay-based timing control statement is shown below.

delay3 ::= # delay_value | # (delay_value [, delay_value [,

delay_value]])

delay2 ::= # delay_value | # (delay_value [, delay_value])

delay_value ::=

unsigned_number

| parameter_identifier

| specparam_identifier

| mintypmax_expression

Delay-based timing control can be specified by a number, identifier, or a mintypmax_expression. There are three types of delay control for procedural assignments: regular delay control, intra-assignment delay control, and zero delay control.

Regular delay control

Regular delay control is used when a non-zero delay is specified to the left of a procedural assignment. Usage of regular delay control is shown in Example 4-10.

Example 4-10 Regular Delay Control

```
//define parameters

parameter latency = 20;

parameter delta = 2;
```

```
//define register variables
reg x, y, z, p, q;
initial
begin
x = 0; // no delay control
#10 y = 1; // delay control with a number. Delay execution of
// y = 1 by 10 units
#latency z = 0; // Delay control with identifier. Delay of 20
units
#(latency + delta) p = 1; // Delay control with expression
#y x = x + 1; // Delay control with identifier. Take value of y.
#(4:5:6) q = 0; // Minimum, typical and maximum delay values.
//Discussed in gate-level modeling chapter.
end
```

In Example 4-10, the execution of a procedural assignment is delayed by the number specified by the delay control. For begin-end groups, delay is always relative to time when the statement is encountered. Thus, y =1 is executed 10 units after it is encountered in the activity flow.

Intra-assignment delay control

Instead of specifying delay control to the left of the assignment, it is possible to assign a delay to the right of the assignment operator. Such delay specification alters the flow of activity in a different manner. Example 4-11 shows the contrast between intra-assignment delays and regular delays.

Example 4-11 Intra-assignment Delays

```
//define register variables
reg x, y, z;
//intra assignment delays
initial
begin
x = 0; z = 0;
y = #5 x + z; //Take value of x and z at the time=0, evaluate
```

```
//x + z and then wait 5 time units to assign value to y.
end
//Equivalent method with temporary variables and regular delay control
initial
begin
x = 0; z = 0;
temp_xz = x + z;
#5 y = temp_xz; //Take value of x + z at the current time and
//store it in a temporary variable. Even though x and z might change between 0 and 5,
//the value assigned to y at time 5 is unaffected.
end
```

Note the difference between intra-assignment delays and regular delays. Regular delays defer the execution of the entire assignment. Intra-assignment delays compute the righthand- side expression at the current time and defer the assignment of the computed value to the left-hand-side variable. Intra-assignment delays are like using regular delays with a temporary variable to store the current value of a right-hand-side expression.

Zero delay control

Procedural statements in different always-initial blocks may be evaluated at the same simulation time. The order of execution of these statements in different always-initial blocks is nondeterministic. Zero delay control is a method to ensure that a statement is executed last, after all other statements in that simulation time are executed. This is used to eliminate race conditions. However, if there are multiple zero delay statements, the order between them is nondeterministic. Example 4-12 illustrates zero delay control.

Example 4-12 Zero Delay Control

```
initial
begin
x = 0;
y = 0;
end
initial
begin
#0 x = 1; //zero delay control
```

```
#0 y = 1;

end
```

In Example 4-12, four statements?x = 0, y = 0, x = 1, y = 1?are to be executed at simulation time 0. However, since x = 1 and y = 1 have #0, they will be executed last. Thus, at the end of time 0, x will have value 1 and y will have value 1. The order in which x = 1 and y = 1 are executed is not deterministic. The above example was used as an illustration. However, using #0 is not a recommended practice.

4.4.2 Event-Based Timing Control

An event is the change in the value on a register or a net. Events can be utilized to trigger execution of a statement or a block of statements. There are four types of event-based timing control: regular event control, named event control, event OR control, and level sensitive timing control.

Regular event control

The @ symbol is used to specify an event control. Statements can be executed on changes in signal value or at a positive or negative transition of the signal value. The keyword posedge is used for a positive transition, as shown in Example 4-13.

Example 4-13 Regular Event Control

```
@(clock) q = d; //q = d is executed whenever signal clock changes value

@(posedge clock) q = d; //q = d is executed whenever signal clock does

//a positive transition ( 0 to 1,x or z,

// x to 1, z to 1 )

@(negedge clock) q = d; //q = d is executed whenever signal clock does

//a negative transition ( 1 to 0,x or z,

//x to 0, z to 0)

q = @(posedge clock) d; //d is evaluated immediately and assigned

//to q at the positive edge of clock
```

Named event control

Verilog provides the capability to declare an event and then trigger and recognize the occurrence of that event (see Example 4-14). The event does not hold any data. A named event is declared by the keyword event. An event is triggered by the symbol ->. The triggering of the event is recognized by the symbol @.

Example 4-14 Named Event Control

```
//This is an example of a data buffer storing data after the
//last packet of data has arrived.
event received_data; //Define an event called received_data
always @(posedge clock) //check at each positive clock edge
begin
if(last_data_packet) //If this is the last data packet
->received_data; //trigger the event received_data
end
always @(received_data) //Await triggering of event received_data
//When event is triggered, store all four
//packets of received data in data buffer
//use concatenation operator { }
data_buf = {data_pkt[0], data_pkt[1], data_pkt[2],
data_pkt[3]};
```

Event OR Control

Sometimes a transition on any one of multiple signals or events can trigger the execution of a statement or a block of statements. This is expressed as an OR of events or signals. The list of events or signals expressed as an OR is also known as a sensitivity list. The keyword or is used to specify multiple triggers, as shown in Example 4-15.

Example 4-15 Event OR Control (Sensitivity List)

```
//A level-sensitive latch with asynchronous reset
always @( reset or clock or d)
//Wait for reset or clock or d to
change
```

```
begin
if (reset) //if reset signal is high, set q to 0.
q = 1'b0;
else if(clock) //if clock is high, latch input
q = d;
end
```

Sensitivity lists can also be specified using the "," (comma) operator instead of the or operator. Example 4-16 shows how the above example can be rewritten using the comma operator. Comma operators can also be applied to sensitivity lists that have edge-sensitive triggers.

Example 4-16 Sensitivity List with Comma Operator

```
//A level-sensitive latch with asynchronous reset
always @( reset, clock, d)
//Wait for reset or clock or d to
change
begin
if (reset) //if reset signal is high, set q to 0.
q = 1'b0;
else if(clock) //if clock is high, latch input
q = d;
end
//A positive edge triggered D flipflop with asynchronous falling
//reset can be modeled as shown below
always @(posedge clk, negedge reset) //Note use of comma operator
if(!reset)
q <=0;
else
```

```
q <=d;
```

When the number of input variables to a combination logic block are very large, sensitivity lists can become very cumbersome to write. Moreover, if an input variable is missed from the sensitivity list, the block will not behave like a combinational logic block. To solve this problem, Verilog HDL contains two special symbols: @* and @(*). Both symbols exhibit identical behavior. These special symbols are sensitive to a change on any signal that may be read by the statement group that follows this symbol

Example 4-17 shows an example of this special symbol for combinational logic sensitivity lists.

IEEE Standard Verilog Hardware Description Language document for details and restrictions on the @* and @(*) symbols.

Example 4-17 Use of @* Operator

```
//Combination logic block using the or operator
//Cumbersome to write and it is easy to miss one input to the block
always @(a or b or c or d or e or f or g or h or p or m)
begin
out1 = a ? b+c : d+e;
out2 = f ? g+h : p+m;
end
//Instead of the above method, use @(*) symbol
//Alternately, the @* symbol can be used
//All input variables are automatically included in the
//sensitivity list.
always @(*)
begin
out1 = a ? b+c : d+e;
out2 = f ? g+h : p+m;
end
```

4.4.3 Level-Sensitive Timing Control

Event control discussed earlier waited for the change of a signal value or the triggering of an event. The symbol @ provided edge-sensitive control. Verilog also allows level sensitive timing control, that is, the ability to wait for a certain condition to be true before a statement or a block of statements is executed. The keyword wait is used for level sensitive constructs.

```
always

wait (count_enable) #20 count = count + 1;
```

In the above example, the value of count_enable is monitored continuously. If count_enable is 0, the statement is not entered. If it is logical 1, the statement count = count + 1 is executed after 20 time units. If count_enable stays at 1, count will be incremented every 20 time units.

4.5 Conditional Statements

Conditional statements are used for making decisions based upon certain conditions. These conditions are used to decide whether or not a statement should be executed. Keywords if and else are used for conditional statements. There are three types of conditional statements. Usage of conditional statements is shown below.

```
//Type 1 conditional statement. No else statement.

//Statement executes or does not execute.

if (<expression>) true_statement ;

//Type 2 conditional statement. One else statement

//Either true_statement or false_statement is evaluated

if (<expression>) true_statement ; else false_statement ;

//Type 3 conditional statement. Nested if-else-if.

//Choice of multiple statements. Only one is executed.

if (<expression1>) true_statement1 ;

else if (<expression2>) true_statement2 ;

else if (<expression3>) true_statement3 ;
```

else default_statement ;

The <expression> is evaluated. If it is true (1 or a non-zero value), the true_statement is executed. However, if it is false (zero) or ambiguous (x), the false_statement is executed. The <expression> can contain any operators. Each true_statement or false_statement can be a single statement or a block of multiple statements. A block must be grouped, typically by using keywords begin and end. A single statement need not be grouped.

Example 4-18 Conditional Statement Examples

```
//Type 1 statements
if(!lock) buffer = data;
if(enable) out = in;
//Type 2 statements
if (number_queued < MAX_Q_DEPTH)
begin
data_queue = data;
number_queued = number_queued + 1;
end
else
$display("Queue Full. Try again");
//Type 3 statements
//Execute statements based on ALU control signal.
if (alu_control == 0)
y = x + z;
else if(alu_control == 1)
y = x - z;
else if(alu_control == 2)
y = x * z;
else
$display("Invalid ALU control signal");
```

4.6 Multiway Branching

Conditional Statements, there were many alternatives, from which one was chosen. The nested if-else-if can become unwieldy if there are too many alternatives. A shortcut to achieve the same result is to use the case statement.

4.6.1 case Statement

The keywords case, endcase, and default are used in the case statement..

```
case (expression)

alternative1: statement1;

alternative2: statement2;

alternative3: statement3;
...
...
default: default_statement;

endcase
```

Each of statement1, statement2 , default_statement can be a single statement or a block of multiple statements. A block of multiple statements must be grouped by keywords begin and end. The expression is compared to the alternatives in the order they are written. For the first alternative that matches, the corresponding statement or block is executed. If none of the alternatives matches, the default_statement is executed. The default_statement is optional. Placing of multiple default statements in one case statement is not allowed. The case statements can be nested. The following Verilog code implements the type 3 conditional statement in Example 4-18.

```
//Execute statements based on the ALU control signal

reg [1:0] alu_control;

...

...

case (alu_control)

2'd0 : y = x + z;

2'd1 : y = x - z;
```

```
2'd2 : y = x * z;
default : $display("Invalid ALU control signal");
endcase
```

The case statement can also act like a many-to-one multiplexer. To understand this, let us model the 4-to-1 multiplexer, using case statements. The I/O ports are unchanged. Notice that an 8-to-1 or 16-to-1 multiplexer can also be easily implemented by case statements.

Example 4-19 4-to-1 Multiplexer with Case Statement

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
reg out;
always @(s1 or s0 or i0 or i1 or i2 or i3)
case ({s1, s0}) //Switch based on concatenation of control signals
2'd0 : out = i0;
2'd1 : out = i1;
2'd2 : out = i2;
2'd3 : out = i3;
default: $display("Invalid control signals");
endcase
endmodule
```

The case statement compares 0, 1, x, and z values in the expression and the alternative bit for bit. If the expression and the alternative are of unequal bit width, they are zero filled to match the bit width of the widest of the expression and the alternative. In Example 4- 20, we will define a 1-to-4 demultiplexer for which outputs are completely specified, that is, definitive results are provided even for x and z values on the select signal.

Example 4-20 Case Statement with x and z

```
module demultiplexer1_to_4 (out0, out1, out2, out3, in, s1, s0);
// Port declarations from the I/O diagram
output out0, out1, out2, out3;
reg out0, out1, out2, out3;
input in;
input s1, s0;
always @(s1 or s0 or in)
case ({s1, s0}) //Switch based on control signals
2'b00 : begin out0 = in; out1 = 1'bz; out2 = 1'bz; out3 =
1'bz; end
2'b01 : begin out0 = 1'bz; out1 = in; out2 = 1'bz; out3 =
1'bz; end
2'b10 : begin out0 = 1'bz; out1 = 1'bz; out2 = in; out3 =
1'bz; end
2'b11 : begin out0 = 1'bz; out1 = 1'bz; out2 = 1'bz; out3 =
in; end
//Account for unknown signals on select. If any select signal is x
//then outputs are x. If any select signal is z, outputs are z.
//If one is x and the other is z, x gets higher priority.
2'bx0, 2'bx1, 2'bxz, 2'bxx, 2'b0x, 2'b1x, 2'bzx :
begin
out0 = 1'bx; out1 = 1'bx; out2 = 1'bx; out3 = 1'bx;
end
2'bz0, 2'bz1, 2'bzz, 2'b0z, 2'b1z :
begin
```

```
out0 = 1'bz; out1 = 1'bz; out2 = 1'bz; out3 = 1'bz;

end

default: $display("Unspecified control signals");

endcase

endmodule
```

In the demultiplexer shown above, multiple input signal combinations such as 2'bz0, 2'bz1, 2,bzz, 2'b0z, and 2'b1z that cause the same block to be executed are put together with a comma (,) symbol.

4.6.2 casex, casez Keywords

There are two variations of the case statement. They are denoted by keywords, casex and casez.

• casez treats all z values in the case alternatives or the case expression as don't cares. All bit positions with z can also represented by ? in that position.

• casex treats all x and z values in the case item or the case expression as don't cares.

The use of casex and casez allows comparison of only non-x or -z positions in the case expression and the case alternatives. Example 4-21 illustrates the decoding of state bits in a finite state machine using a casex statement. The use of casez is similar. Only one bit is considered to determine the next state and the other bits are ignored.

Example 4-21 casex Use

```
reg [3:0] encoding;

integer state;

casex (encoding) //logic value x represents a don't care bit.

4'b1xxx : next_state = 3;

4'bx1xx : next_state = 2;

4'bxx1x : next_state = 1;

4'bxxx1 : next_state = 0;

default : next_state = 0;

endcase
```

Thus, an input encoding = 4'b10xz would cause next_state = 3 to be executed.

4.7 Loops

There are four types of looping statements in Verilog: while, for, repeat, and forever. The syntax of these loops is very similar to the syntax of loops in the C programming language. All looping statements can appear only inside an initial or always block. Loops may contain delay expressions.

4.7.1 While Loop

The keyword while is used to specify this loop. The while loop executes until the while expression is not true. If the loop is entered when the while-expression is not true, the loop is not executed at all. Each expression can contain the operators. Any logical expression can be specified with these operators. If multiple statements are to be executed in the loop, they must be grouped typically using keywords begin and end. Example 4-22 illustrates the use of the while loop.

Example 4-22 While Loop

```
//Illustration 1: Increment count from 0 to 127. Exit at count 128.
//Display the count variable.
integer count;
initial
begin
count = 0;
while (count < 128) //Execute loop till count is 127.
//exit at count 128
begin
$display("Count = %d", count);
count = count + 1;
end
end
//Illustration 2: Find the first bit with a value 1 in flag (vector
variable)
```

```
'define TRUE 1'b1';

'define FALSE 1'b0;

reg [15:0] flag;

integer i; //integer to keep count

reg continue;

initial

begin

flag = 16'b 0010_0000_0000_0000;

i = 0;

continue = 'TRUE;

148

while((i < 16) && continue ) //Multiple conditions using operators.

begin

if (flag[i])

begin

$display("Encountered a TRUE bit at element number %d", i);

continue = 'FALSE;

end
i = i + 1;
end
end
```

4.7.2 for Loop

The keyword for is used to specify this loop. The for loop contains three parts:

- An initial condition

- A check to see if the terminating condition is true

- A procedural assignment to change value of the control variable

The counter described in Example 4-22 can be coded as a for loop (Example 4-23). The initialization condition and the incrementing procedural assignment are included in the for loop and do not need to be specified separately. Thus, the for loop provides a more compact loop structure than the while loop. Note, however, that the while loop is more general-purpose than the for loop. The for loop cannot be used in place of the while loop in all situations.

Example 4-23 For Loop

```
integer count;
initial
for ( count=0; count < 128; count = count + 1)
$display("Count = %d", count);
for loops can also be used to initialize an array or memory, as shown below.
//Initialize array elements
'define MAX_STATES 32
integer state [0: 'MAX_STATES-1]; //Integer array state with elements
0:31
integer i;
initial
begin
for(i = 0; i < 32; i = i + 2) //initialize all even locations with 0
state[i] = 0;
for(i = 1; i < 32; i = i + 2) //initialize all odd locations with 1
state[i] = 1;
end
```

for loops are generally used when there is a fixed beginning and end to the loop. If the loop is simply looping on a certain condition, it is better to use the while loop.

4.7.3 Repeat Loop

The keyword repeat is used for this loop. The repeat construct executes the loop a fixed number of times. A repeat construct cannot be used to loop on a general logical expression. A while loop is used for that purpose. A repeat construct must contain a number, which can be a constant, a variable or a signal value. However, if the number is a variable or signal value, it is evaluated only when the loop starts and not during the loop execution.

The counter in Example 4-22 can be expressed with the repeat loop, as shown in

Illustration 1 in Example 4-24. Illustration 2 shows how to model a data buffer that latches data at the positive edge of clock for the next eight cycles after it receives a data start signal.

Example 4-24 Repeat Loop

```
//Illustration 1 : increment and display count from 0 to 127

integer count;

initial

begin

count = 0;

repeat(128)

begin

$display("Count = %d", count);

count = count + 1;

end

end

//Illustration 2 : Data buffer module example

//After it receives a data_start signal.

//Reads data for next 8 cycles.

module data_buffer(data_start, data, clock);

parameter cycles = 8;

input data_start;
input [15:0] data;

input clock;

reg [15:0] buffer [0:7];

integer i;

150

always @(posedge clock)

begin

if(data_start) //data start signal is true

begin

i = 0;

repeat(cycles) //Store data at the posedge of next 8 clock

//cycles

begin

@(posedge clock) buffer[i] = data; //waits till next

// posedge to latch data

i = i + 1;

end

end

end

endmodule
```

4.7.4 Forever loop

The keyword forever is used to express this loop. The loop does not contain any expression and executes forever until the $finish task is encountered. The loop is equivalent to a while loop with an expression that always evaluates to true, e.g., while (1). A forever loop can be exited by use of the disable statement.

A forever loop is typically used in conjunction with timing control constructs. If timing control constructs are not used, the Verilog simulator would execute this statement infinitely without advancing simulation time and the rest of the design would never be executed. Example 4-25 explains the use of the forever statement.

Example 4-25 Forever Loop

```
//Example 1: Clock generation

//Use forever loop instead of always block

reg clock;

initial

begin

clock = 1'b0;

forever #10 clock = ~clock; //Clock with period of 20 units

end

//Example 2: Synchronize two register values at every positive edge of

//clock

reg clock;

reg x, y;

initial

forever @(posedge clock) x = y;
```

4.8 Sequential and Parallel Blocks

Block statements are used to group multiple statements to act together as one. In previous examples, we used keywords begin and end to group multiple statements. Thus, we used sequential blocks where the statements in the block execute one after another. In this section we discuss the block types: sequential blocks and parallel blocks. We also discuss three special features of blocks: named blocks, disabling named blocks, and nested blocks.

4.8.1 Block Types

There are two types of blocks: sequential blocks and parallel blocks.

Sequential blocks

The keywords begin and end are used to group statements into sequential blocks.

Sequential blocks have the following characteristics:

- The statements in a sequential block are processed in the order they are specified. A statement is executed only after its preceding statement completes execution (except for nonblocking assignments with intra-assignment timing control).

- If delay or event control is specified, it is relative to the simulation time when the previous statement in the block completed execution.

We have used numerous examples of sequential blocks in this book. Two more examples of sequential blocks are given in Example 4-26. Statements in the sequential block execute in order. In Illustration 1, the final values are x = 0, y= 1, z = 1, w = 2 at simulation time 0. In Illustration 2, the final values are the same except that the simulation time is 35 at the end of the block.

Example 4-26 Sequential Blocks

```
//Illustration 1: Sequential block without delay
reg x, y;

reg [1:0] z, w;

initial

begin

x = 1'b0;

y = 1'b1;

z = {x, y};

w = {y, x};

end
//Illustration 2: Sequential blocks with delay.
reg x, y;
reg [1:0] z, w;

initial

begin
```

```
x = 1'b0; //completes at simulation time 0

#5 y = 1'b1; //completes at simulation time 5

#10 z = {x, y}; //completes at simulation time 15

#20 w = {y, x}; //completes at simulation time 35

end
```

Parallel blocks

Parallel blocks, specified by keywords fork and join, provide interesting simulation features. Parallel blocks have the following characteristics:

- Statements in a parallel block are executed concurrently.

- Ordering of statements is controlled by the delay or event control assigned to each statement.

- If delay or event control is specified, it is relative to the time the block was entered.

Notice the fundamental difference between sequential and parallel blocks. All statements in a parallel block start at the time when the block was entered. Thus, the order in which the statements are written in the block is not important.

Let us consider the sequential block with delay in Example 4-26 and convert it to a parallel block. The converted Verilog code is shown in Example 4-27. The result of simulation remains the same except that all statements start in parallel at time 0. Hence, the block finishes at time 20 instead of time 35.

Example 4-27 Parallel Blocks

```
//Example 1: Parallel blocks with delay.

reg x, y;

reg [1:0] z, w;

initial

fork

x = 1'b0; //completes at simulation time 0

#5 y = 1'b1; //completes at simulation time 5

#10 z = {x, y}; //completes at simulation time 10
```

```
#20 w = {y, x}; //completes at simulation time 20

join
```

Parallel blocks provide a mechanism to execute statements in parallel. However, it is important to be careful with parallel blocks because of implicit race conditions that might arise if two statements that affect the same variable complete at the same time. Shown below is the parallel version of Illustration 1 from Example 4-26. Race conditions have been deliberately introduced in this example. All statements start at simulation time 0.

The order in which the statements will execute is not known. Variables z and w will get values 1 and 2 if x = 1'b0 and y = 1'b1 execute first. Variables z and w will get values 2'bxx and 2'bxx if x = 1'b0 and y = 1'b1 execute last. Thus, the result of z and w is nondeterministic and dependent on the simulator implementation. In simulation time, all statements in the fork-join block are executed at once. However, in reality, CPUs running simulations can execute only one statement at a time. Different simulators execute statements in different order. Thus, the race condition is a limitation of today's simulators, not of the fork-join block.

```
//Parallel blocks with deliberate race condition

reg x, y;

reg [1:0] z, w;

initial

fork

x = 1'b0;

y = 1'b1;

z = {x, y};

w = {y, x};

join
```

The keyword fork can be viewed as splitting a single flow into independent flows. The keyword join can be seen as joining the independent flows back into a single flow. Independent flows operate concurrently.

4.8.2 Special Features of Blocks

We discuss three special features available with block statements: nested blocks, named blocks, and disabling of named blocks.

Nested blocks

Blocks can be nested. Sequential and parallel blocks can be mixed, as shown in Example 4-28.

Example 4-28 Nested Blocks

```
//Nested blocks

initial

begin

x = 1'b0;

154

fork

#5 y = 1'b1;

#10 z = {x, y};

join

#20 w = {y, x};

end
```

Named blocks

Blocks can be given names.

• Local variables can be declared for the named block.

• Named blocks are a part of the design hierarchy. Variables in a named block can be accessed by using hierarchical name referencing.

• Named blocks can be disabled, i.e., their execution can be stopped.

Example 4-29 shows naming of blocks and hierarchical naming of blocks.

Example 4-29 Named Blocks

```
//Named blocks

module top;

initial

begin: block1 //sequential block named block1

integer i; //integer i is static and local to block1

// can be accessed by hierarchical name, top.block1.i

...

...

end

initial

fork: block2 //parallel block named block2

reg i; // register i is static and local to block2

// can be accessed by hierarchical name, top.block2.i
...
...
join
```

Disabling named blocks

The keyword disable provides a way to terminate the execution of a named block. Disable can be used to get out of loops, handle error conditions, or control execution of pieces of code, based on a control signal. Disabling a block causes the execution control to be passed to the statement immediately succeeding the block. For C programmers, this is very similar to the break statement used to exit a loop.

4.9: Outcomes

After completion of the module the students are able to:

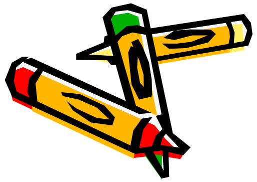
- Explain the significance of structured procedures always and initial in behavioral modeling.
- Define blocking and nonblocking procedural assignments.
- Understand delay-based timing control mechanism in behavioral modeling. Use regular delays, intra-assignment delays, and zero delays.
- Describe event-based timing control mechanism in behavioral modeling. Use regular event control, named event control, and event OR control.
- Use level-sensitive timing control mechanism in behavioral modeling.
- Explain conditional statements using if and else.
- Describe multiway branching, using case, casex, and casez statements.
- Understand looping statements such as while, for, repeat, and forever.
- Define sequential and parallel blocks.

4.10: Recommended Questions

1. Describe the following statements with an example: initial and always
2. What are blocking and non-blocking assignment statements? Explain with examples.
3. With syntax explain conditional, branching and loop statements available in Verilog HDL behavioural description.
4. Describe sequential and parallel blocks of Verilog HDL.
5. Write Verilog HDL program of 4:1 mux using CASE statement.
6. Write Verilog HDL program of 4:1 mux using If-else statement.
7. Write Verilog HDL program of 4-bit synchronous up counter.
8. Write Verilog HDL program of 4-bit asynchronous down counter.
9. Write Verilog HDL program to simulate traffic signal controller

Filter Design Techniques

- Filter
  - Filter is a system that passes certain frequency components and totally rejects all others
- Stages of the design filter
  - Specification of the desired properties of the system
  - Approximation of the specification using a causal discrete-time system
  - Realization of the system



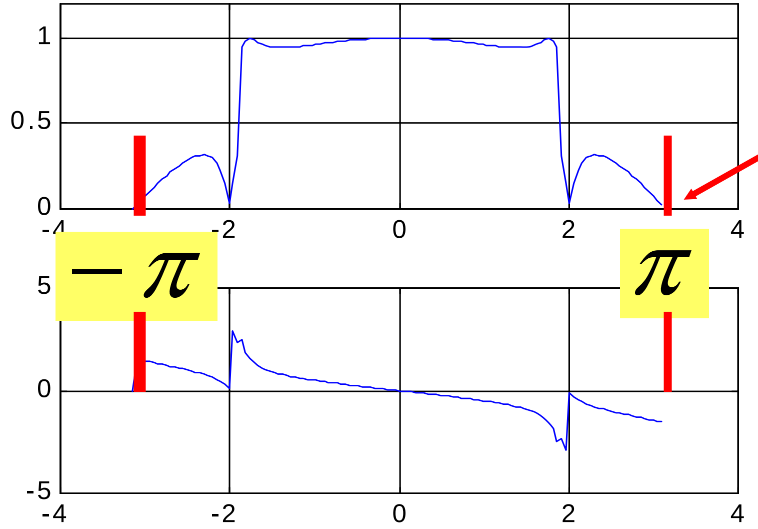
Review of discrete-time systems

Frequency response :

- periodic : period = 2π
- for a real impulse response h[k]

  Magnitude response $\left|H(e^{j\omega})\right|$ is even function

  Phase response $\angle H(e^{j\omega})$ is odd function
- example :



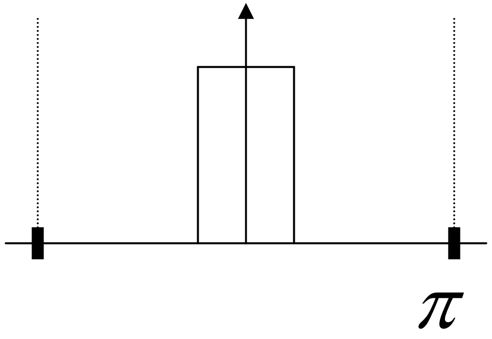
Nyquist frequency

$$e^{j\pi k} = \ldots,1,-1,1,-1,1,\ldots$$

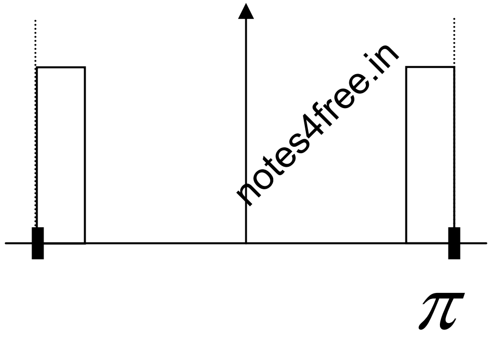
Review of discrete-time systems

`Popular' frequency responses for filter design :

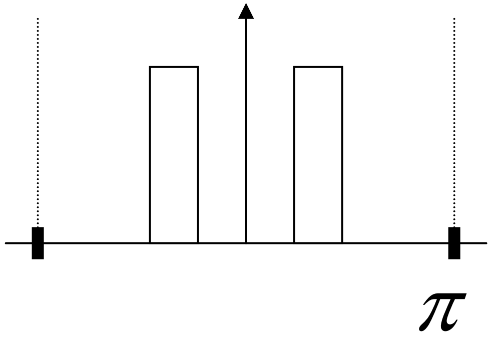
low-pass (LP)



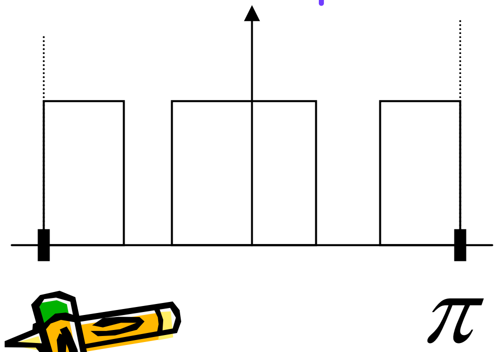
high-pass (HP)



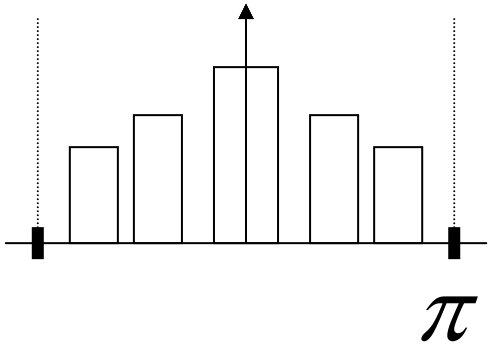
band-pass (BP)



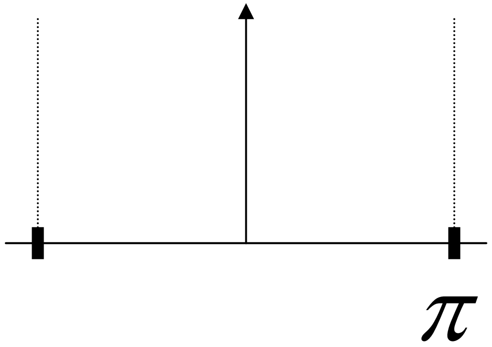
band-stop



multi-band



…



Review of discrete-time systems

"FIR filters" (finite impulse response):

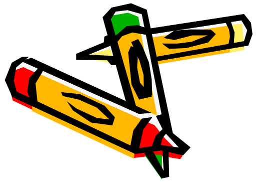
$$H(z) = \frac{B(z)}{z^N} = b_0 + b_1 z^{-1} + \ldots + b_N z^{-N}$$

- "Moving average filters" (MA filters)
- N poles at the origin z=0 (hence guaranteed stability)
- N zeros (zeros of B(z)), "all zero" filters
- corresponds to difference equation

$$y[k] = b_0 u[k] + b_1 u[k-1] + \ldots + b_N u[k-N]$$

- impulse response

$$h[0] = b_0, h[1] = b_1, \ldots, h[N] = b_N, h[N+1] = 0, \ldots$$



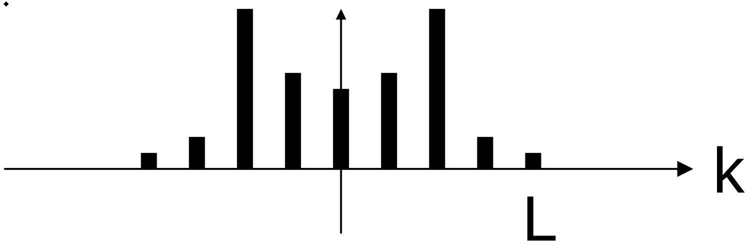
Linear Phase FIR Filters

Non-causal zero-phase filters :

example: symmetric impulse response

h[-L],….h[-1],h[0],h[1],...,h[L]

h[k]=h[-k], k=1..L



frequency response is

$$H(e^{j\omega}) = \sum_{k=-L}^{+L} h[k]e^{-j\omega.k} = ... = \sum_{k=0}^{L} a_k \cos(\omega k)$$

- i.e. real-valued (=zero-phase) transfer function
- causal implementation by introducing (group) delay

Linear Phase FIR Filters

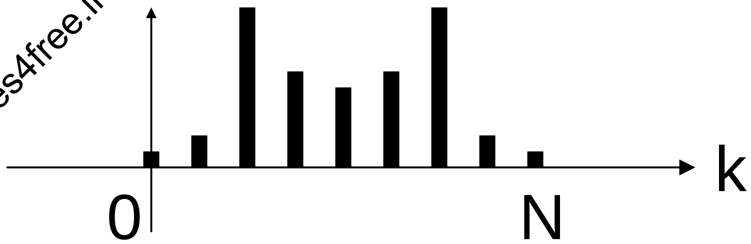
- Causal linear-phase filters = non-causal zero-phase + delay

example: symmetric impulse response & N even

h[0],h[1],....,h[N]

N=2L (even)

h[k]=h[N-k], k=0..L



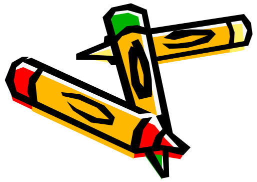
frequency response is

$$H(e^{j\omega}) = \sum_{k=0}^{N} h[k]e^{-j\omega.k} = \ldots = e^{-j\omega L}\sum_{k=0}^{L} a_k \cos(\omega k)$$

= i.e. causal implementation of zero-phase filter, by introducing (group) delay $z^{-L}\Big|_{z=e^{j\omega}} = e^{-j\omega L}$

Linear Phase FIR Filters

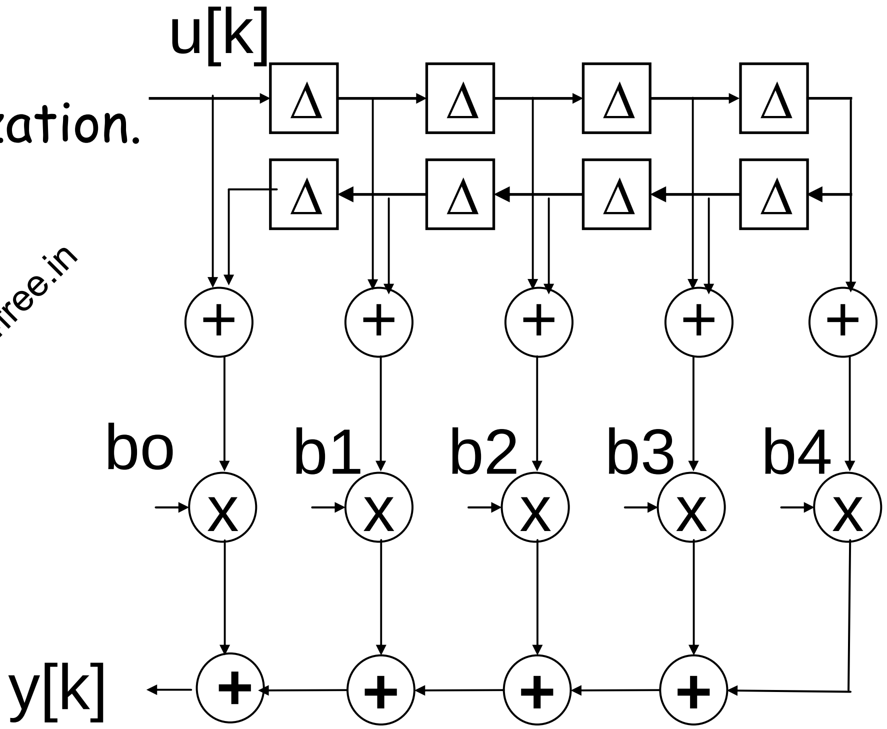
Type-1	Type-2	Type-3	Type-4
N=2L=even	N=2L+1=odd	N=2L=even	N=2L+1=odd
symmetric	symmetric	anti-symmetric	anti-symmetric
h[k]=h[N-k]	h[k]=h[N-k]	h[k]=-h[N-k]	h[k]=-h[N-k]
$e^{-j\omega N/2}\sum_{k=0}^{L} a_k \cos(\omega k)$	$e^{-j\omega N/2}\cos(\frac{\omega}{2})\sum_{k=0}^{L} a_k \cos(\omega k)$	$je^{-j\omega N/2}\sin(\omega)\sum_{k=0}^{L-1} a_k \cos(\omega k)$	$j.e^{-j\omega N/2}\sin(\frac{\omega}{2})\sum_{k=0}^{L} a_k \cos(\omega k)$
	zero at $\omega = \pi$	zero at $\omega = 0, \pi$	zero at $\omega = 0$
LP/HP/BP	LP/BP		HP



Linear Phase FIR Filters

- efficient direct-form realization.

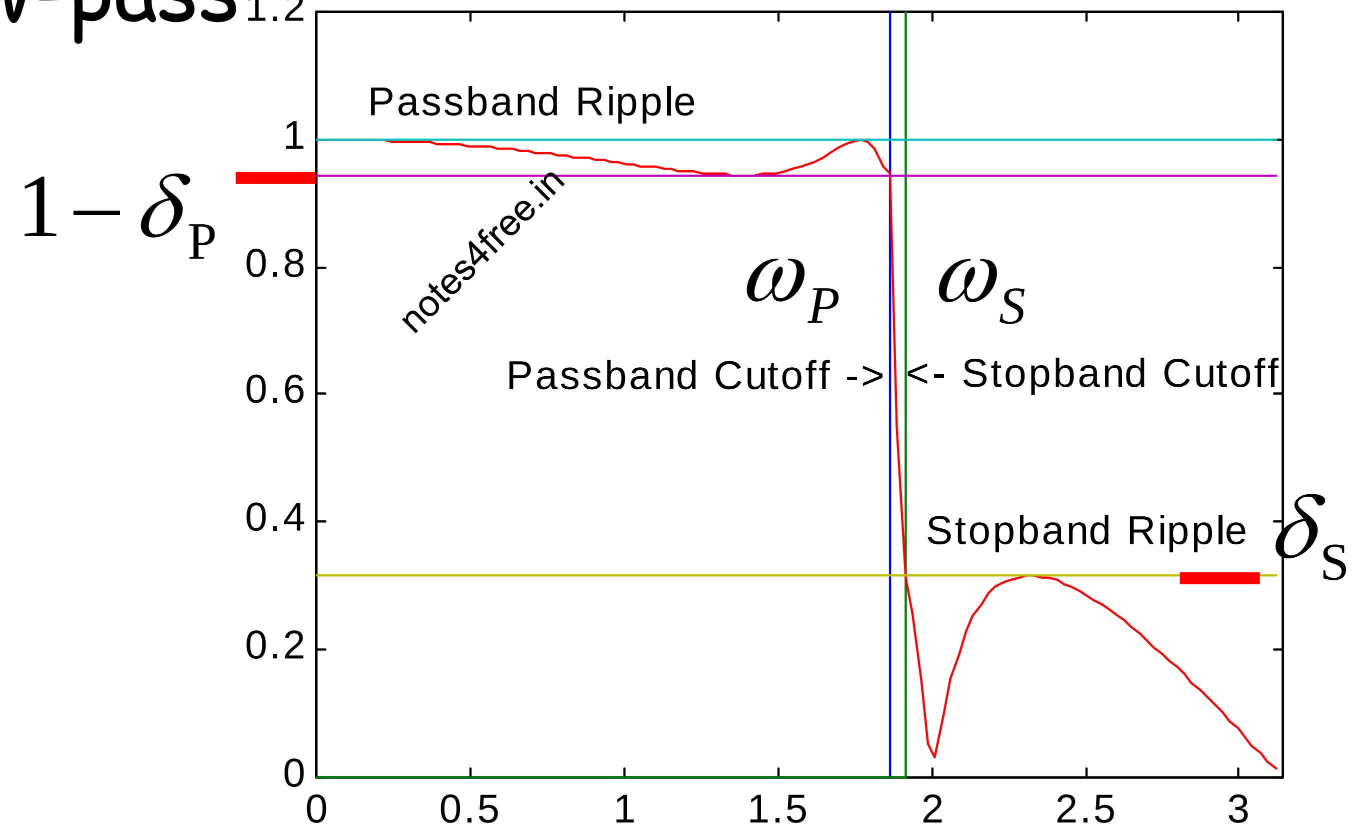
  example:



- PS: IIR filters can NEVER have linear-phase property !

Filter Specification

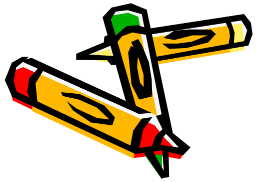
Ex: Low-pass



Filter Design Problem

- Design of filters is a problem of function approximation

- For FIR filter, it implies polynomial approximation
- For IIR filter, it implies approximation by a rational function of z



Filter Design by Optimization

(I) Weighted Least Squares Design :

- select one of the basic forms that yield linear phase

e.g. Type-1 $$H(e^{j\omega}) = e^{-j\omega N/2} \sum_{k=0}^{L} a_k \cos(\omega k) = e^{-j\omega N/2} A(\omega)$$

- specify desired frequency response (LP,HP,BP,…)

$$H_d(\omega) = e^{-j\omega N/2} A_d(\omega)$$

- optimization criterion is

$$\min_{a_0,\ldots,a_L} \int_{-\pi}^{+\pi} W(\omega) \left| H(e^{j\omega}) - H_d(\omega) \right|^2 d\omega = \min_{a_0,\ldots,a_L} \underbrace{\int_{-\pi}^{+\pi} W(\omega) \left| A(\omega) - A_d(\omega) \right|^2 d\omega}_{F(a_0,\ldots,a_L)}$$

where $W(\omega) \geq 0$ is a weighting function

Filter Design by Optimization

- …this is equivalent to

$$\min_x \{ \overbrace{x^T Q x - 2x^T p + \mu}^{F(a_0,\dots,a_L)} \}$$

$$x^T = [a_0 \quad a_1 \quad \dots \quad a_L]$$

$$Q = \int_0^\pi W(\omega)c(\omega)c^T(\omega)d\omega$$

$$p = \int_0^\pi W(\omega)A_d(\omega)c(\omega)d\omega$$

$$c^T(\omega) = [1 \quad \cos(\omega) \quad \dots \quad \cos(L\omega)]$$

$$\mu = \dots$$

=standard ‘Quadratic Optimization’ problem

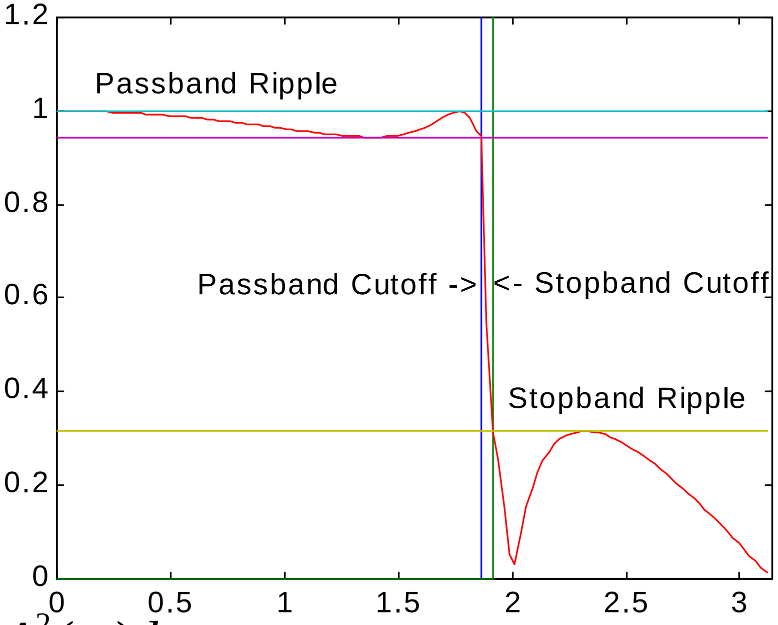
$$x_{OPT} = Q^{-1}p$$

Filter Design by Optimization

- Example: Low-pass design

$$A_d(\omega) = 1, |\omega| < \omega_P \qquad \text{(pass - band)}$$

$$A_d(\omega) = 0, \omega_S \le |\omega| \le \pi \qquad \text{(stop - band)}$$



optimization function is

$$F(a_0,\dots,a_L) = \underbrace{\int_0^{\omega_P} |A(\omega)-1|^2 d\omega}_{\text{pass - band}} + \gamma . \underbrace{\int_{\omega_S}^{+\pi} A^2(\omega) d\omega}_{\text{stop - band}} = \dots$$

i.e.

$$W(\omega) = \dots$$

Filter Design by Optimization

- a simpler problem is obtained by replacing the F(..) by…

$$\underline{F}(a_0,\dots,a_L) = \sum_i W(\omega_i)\left|A(\omega_i) - A_d(\omega_i)\right|^2 = \sum_i W(\omega_i)\left\{c^T(\omega_i)\begin{bmatrix}a_0\\ :\\ a_L\end{bmatrix} - A_d(\omega_i)\right\}^2$$

where the w_i's are a set of n sample frequencies

The quadratic optimization problem is then equivalent to a least-squares problem

$$\min_x \left\|\underline{A}x - \underline{b}\right\|_2^2 = \min_x \{x^T \underbrace{\underline{A}^T\underline{A}}_{\sum_i W(\omega_i)c(\omega_i)c^T(\omega_i)} x - 2x^T \underbrace{\underline{A}^T\underline{b}}_{\sum_i \dots} + \underbrace{\underline{b}^T\underline{b}}_{\sum_i \dots}\}$$

$$x_{LS} = (\underline{A}^T\underline{A})^{-1}\underline{A}^T\underline{b}$$

Compare to p.12

+++ : simple

--- : unpredictable behavior in between sample frequencies.

Filter Design by Optimization

- …then all this is often supplemented with
  additional constraints

Example: Low-pass (LP) design (continued)

pass-band ripple control :

$$\left|A(\omega)-1\right| \le \delta_{\mathrm{P}}, \left|\omega\right| < \omega_{P} \quad (\delta_{\mathrm{P}} \text{ is pass - band ripple})$$

stop-band ripple control :

$$\left|A(\omega)\right| \le \delta_{\mathrm{S}}, \omega_{\mathrm{S}} \le \left|\omega\right| \le \pi \quad (\delta_{\mathrm{S}} \text{ is stop - band ripple})$$

Filter Design by Optimization

Example: Low-pass (LP) design (continued)

a realistic way to implement these constraints, is to impose the constraints (only) on a set of sample frequencies

$\omega_{P1}, \omega_{P2}, \ldots, \omega_{Pm}$ in the pass-band

and $\omega_{S1}, \omega_{S2}, \ldots, \omega_{Sn}$ in the stop-band

The resulting optimization problem is :

minimize : $F(a_0, \ldots, a_L) = \ldots$

$$x^T = \begin{bmatrix} a_0 & a_1 & \ldots & a_L \end{bmatrix}$$

subject to $A_P x \le b_P$ (pass-band constraints)

$A_S x \le b_S$ (stop-band constraints)

= `Quadratic Linear Programming' problem

Filter Design by Optimization

(II) `Minimax' Design :

- select one of the basic forms that yield linear phase

  e.g. Type-1 $H(e^{j\omega}) = e^{-j\omega N/2}\sum_{k=0}^{L} a[k]\cos(\omega k) = e^{-j\omega N/2}A(\omega)$

- specify desired frequency response (LP,HP,BP,…)

$$H_d(\omega) = e^{-j\omega N/2}A_d(\omega)$$

- optimization criterion is

$$\min_{a_0,\ldots,a_L}\ \max_{0\le\omega\le\pi} W(\omega)\left|H(e^{j\omega}) - H_d(\omega)\right| = \min_{a_0,\ldots,a_L}\ \max_{0\le\omega\le\pi} W(\omega)\left|A(\omega) - A_d(\omega)\right|$$

where $W(\omega) \ge 0$ is a weighting function

Filter Design by Optimization

- Conclusion:

  (I) weighted least squares design

  (II) minimax design

  provide general `framework’, procedures to translate filter design problems into standard optimization problems

- In practice (and in textbooks):

  emphasis on specific (ad-hoc) procedures :

  - filter design based on ‘windows’

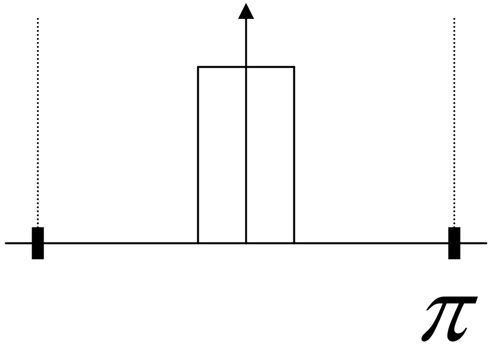
  - equi-ripple design

Filter Design using ‘Windows’

Example : Low-pass filter design

- ideal low-pass filter is

$$H_d(\omega) = \begin{cases} 1 & |\omega| < \omega_C \\ 0 & \omega_C < |\omega| < \pi \end{cases}$$



- hence ideal time-domain impulse response is

$$h_d[k] = \frac{1}{2\pi}\int_{-\pi}^{+\pi} H_d(e^{j\omega})e^{j\omega.k}\,d\omega = \ldots = \alpha\,\frac{\sin(\omega_c k)}{\omega_c k}$$

Non-causal and infinitely long

- truncate $h_d[k]$ to N+1 samples :

$$h[k] = \begin{cases} h_d[k] & -N/2 < k < N/2 \\ 0 & \text{otherwise} \end{cases}$$

- add (group) delay to turn into causal filter

Filter Design using 'Windows'

Example : Low-pass filter design (continued)

- note : it can be shown that time-domain truncation corresponds to solving a weighted least-squares optimization problem with the given H$_d$, and weighting function $W(\omega)=1$

- truncation corresponds to applying a 'rectangular window' :

$$h[k]=h_d[k]w[k]$$

$$w[k]=\begin{cases}1 & -N/2<k<N/2\\ 0 & \text{otherwise}\end{cases}$$

- +++: simple procedure (also for HP,BP,...)
- --- : truncation in the time-domain results in 'Gibbs effect' in the frequency domain, i.e. large ripple in pass-band and stop-band, which cannot be reduced by increasing the filter order N.

Filter Design using ‘Windows’

Remedy : apply windows other than rectangular window:

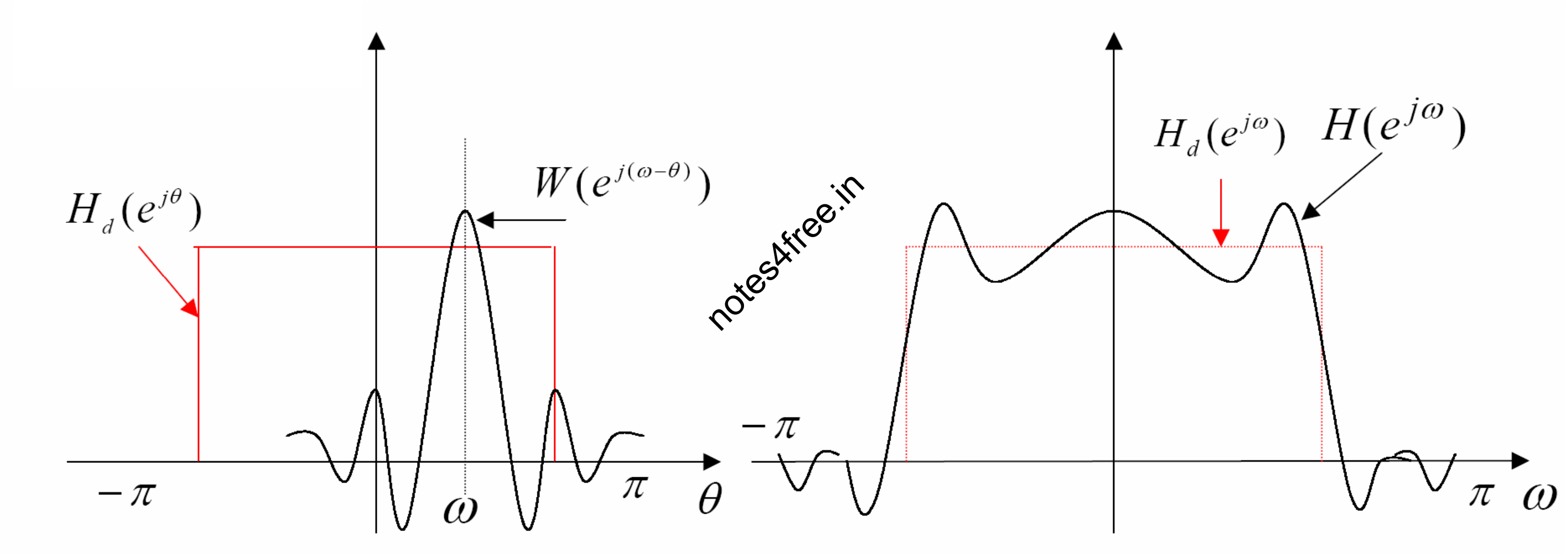
- time-domain multiplication with a window function w[k] corresponds to frequency domain convolution with W(z) :

$$h[k] = h_d[k]w[k]$$

$$H(z) = H_d(z) * W(z)$$

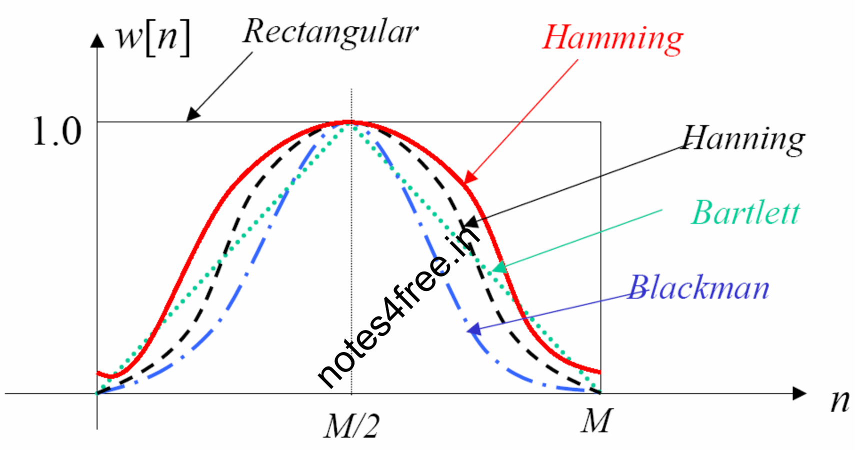
- candidate windows : Han, Hamming, Blackman, Kaiser,…. (see textbooks)
- window choice/design = trade-off between side-lobe levels (define peak pass-/stop-band ripple) and width main-lobe (defines transition bandwidth)

Windowing Effect

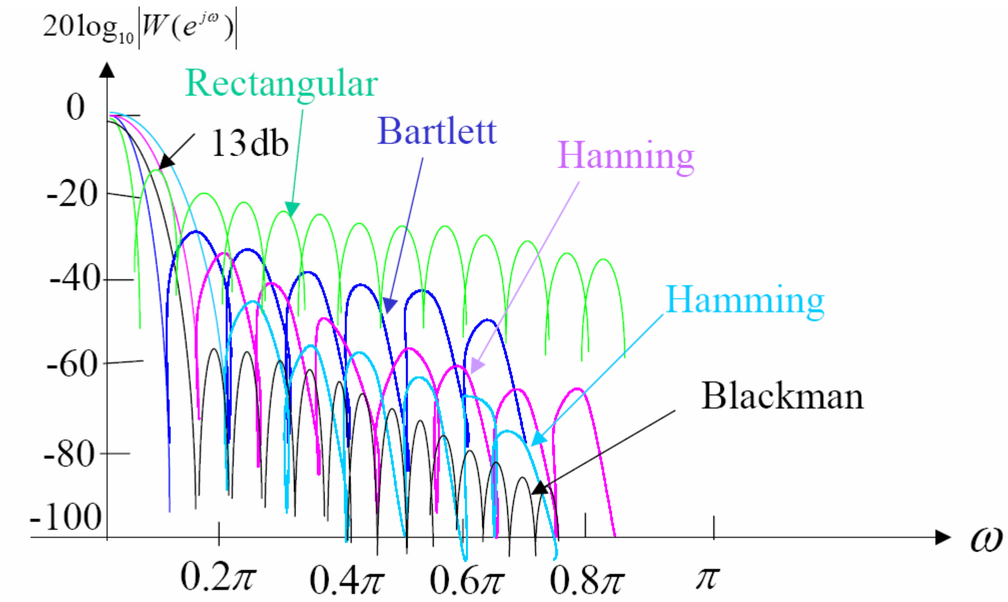


Gibbs phenomenon

Windowing



$w[n]$
Rectangular
Hamming
1.0
Hanning
Bartlett
Blackman
n
$M/2$
M



$20\log_{10}|W(e^{j\omega})|$
Rectangular
0
13db
Bartlett
Hanning
-20
-40
Hamming
-60
Blackman
-80
-100
0.2π
0.4π
0.6π
0.8π
π
ω

Equiripple Design

- Starting point is minimax criterion, e.g.

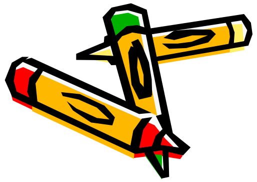
$$\min_{a_0,\dots,a_L}\ \max_{0\le\omega\le\pi} W(\omega)\left|A(\omega)-A_d(\omega)\right| = \min_{a_0,\dots,a_L}\ \max_{0\le\omega\le\pi}\left|E(\omega)\right|$$

- Based on theory of Chebyshev approximation and the 'alternation theorem', which (roughly) states that the optimal a_i's are such that the 'max' (maximum weighted approximation error) is obtained at L+2 extremal frequencies…

$$\max_{0\le\omega\le\pi}\left|E(\omega)\right| = \left|E(\omega_i)\right| \quad \text{for } i=1,..,L+2$$

…that hence will exhibit the same maximum ripple ('equiripple')

- Iterative procedure for computing extremal frequencies, etc. (Remez exchange algorithm, Parks-McClellan algorithm)
- Very flexible, etc., available in many software packages
- Details omitted here (see textbooks)



Software

- FIR Filter design abundantly available in commercial software
- Matlab:

b=fir1(n,Wn,type,window), windowed linear-phase FIR design, n is filter order, Wn defines band-edges, type is `high',`stop',…

b=fir2(n,f,m,window), windowed FIR design based on inverse Fourier transform with frequency points f and corresponding magnitude response m

b=remez(n,f,m), equiripple linear-phase FIR design with Parks-McClellan (Remez exchange) algorithm