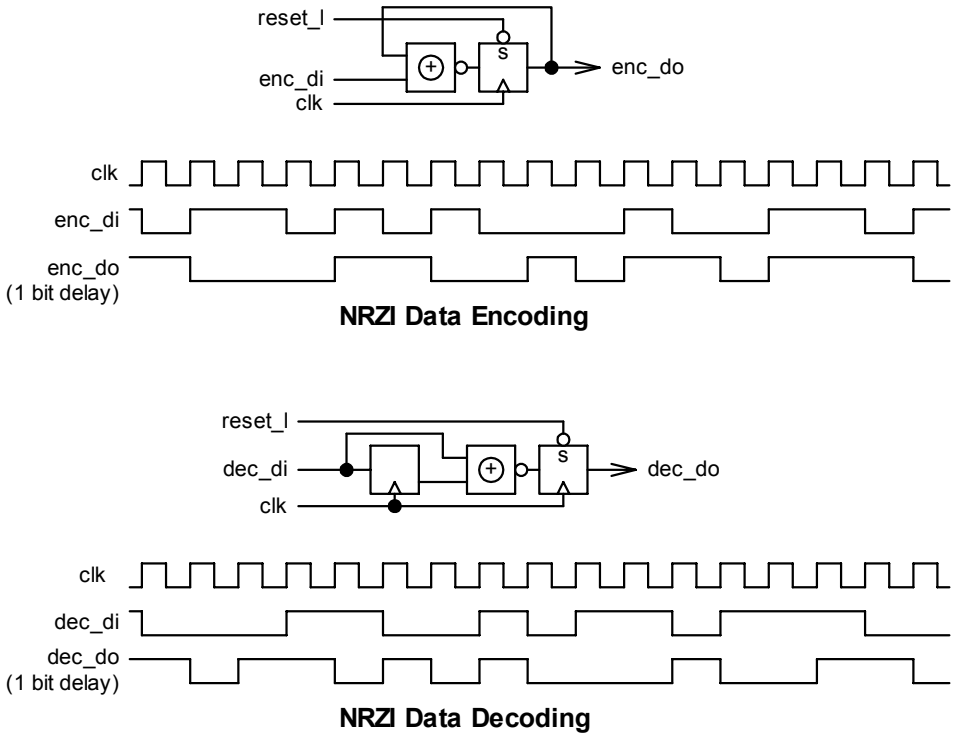
NRZI (Non Return to Zero Invert) Encoding & Decoding

(A) General concept



NRZI Data Encoding

NRZI Data Decoding

(B) Emulation by C language

A utility called “nrzi.c” emulates NRZI encoder & decoder.

```
/************************************************
Program name:   nrzi
Module name:    nrzi.c
Description:    Emulates NRZI encoder and decoder.
************************************************/

#include     <stdio.h>
#include     <process.h>

static int   i;
static int   enc_do;
static int   enc_di[17] = {
             0x00, 0x01, 0x01, 0x00, 0x01, 0x00, 0x01, 0x00,
             0x00, 0x00, 0x01, 0x00, 0x00, 0x01, 0x01, 0x00,
             0x01};
static int   dec_do;
static int   dec_di[18] = {
             0x01, 0x00, 0x00, 0x00, 0x01, 0x01, 0x00, 0x00,
             0x01, 0x00, 0x01, 0x01, 0x00, 0x01, 0x01, 0x01,
             0x00, 0x00};

FILE         *outfp, *fopen();

main()
{
if((outfp = fopen("o.o", "w")) == NULL)
  {
  printf("Output file %s open error...\n", "o.o");
  exit(0);
```

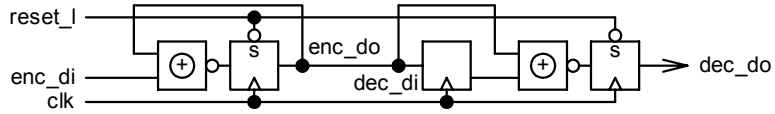
```
  }

/* NRZI encoder */
fprintf(outfp, "e e\n");
fprintf(outfp, "n n\n");
fprintf(outfp, "c c\n");
fprintf(outfp, "_ _\n");
fprintf(outfp, "d d\n");
fprintf(outfp, "i o\n");
fprintf(outfp, "---\n");

enc_do = 1;
fprintf(outfp, "%1.1x %1.1x\n", enc_di[0], enc_do);

for(i = 0; i < 16; i++)
  {
  if(enc_di[i] == 0x00)  enc_do = !enc_do;
  fprintf(outfp, "%1.1x %1.1x\n", enc_di[i + 1], enc_do);
  }
fprintf(outfp, "\n\n");

/* NRZI decoder */
fprintf(outfp, "d d\n");
fprintf(outfp, "e e\n");
fprintf(outfp, "c c\n");
fprintf(outfp, "_ _\n");
fprintf(outfp, "d d\n");
fprintf(outfp, "i o\n");
fprintf(outfp, "---\n");

dec_do = 1;
fprintf(outfp, "%1.1x %1.1x\n", dec_di[0], dec_do);
fprintf(outfp, "%1.1x %1.1x\n", dec_di[1], dec_do);

for(i = 0; i < 17; i++)
  {
  if((dec_di[i] ^ dec_di[i + 1]) == 1)  dec_do = 0;
  else                                  dec_do = 1;
  fprintf(outfp, "%1.1x %1.1x\n", dec_di[i + 2], dec_do);
  }

fclose(outfp);
}
```

Emulation result

```
e e
n n
c c
_ _
d d
i o
---
0 1
1 0
1 0
0 0
1 1
0 1
1 0
0 0
0 1
0 0
1 1
0 1
0 0
1 1
1 1
0 1
1 0
```

```
d d
e e
c c
_ _
d d
i o
---
1 1
0 1
0 0
0 1
1 1
1 0
0 1
0 0
1 1
0 0
1 0
1 0
0 1
1 0
1 0
1 1
0 1
0 0
0 1
```

(C) Computer simulation by Verilog HDL



Behavioral or state machine description frequently causes the timing and speed problems after logic synthesis due to lack of detailed timing consideration reflecting the actual functional cells like DFF and combinational logic gates. From the original logic design stage, the timing must be considered. It must not be resolved by putting complicated timing constraints when the logic synthesis is done.
Software people who do not know what kind of physical gates and flip-flops are made after logic synthesis can make big logic easily by Verilog HDL using behavioral or state machine description as well. However, they may possibly make dangerous and unstable logic due to such unskilled technical background. Debugging effort never ends in such case.
RTL description considering the timing and actual functional cells generated is more realistic practical logic design approach. Take simpler and straight-forward approach.

(a) Verilog source code
(1) Test bench (nrzi_sys.v)
Test bench generates a simulation vector that should be flexible. To make good test bench, some sort of programming skill backed up by polished sense is required.

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Test bench for NRZI encoding & decoding
//  nrzi_sys.v
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
`timescale     1ns / 1ns

module         nrzi_sys;

// Regs/wires/integers
reg            clk, reset_l, enc_di;
wire           dec_do;
integer        i, mask;
```

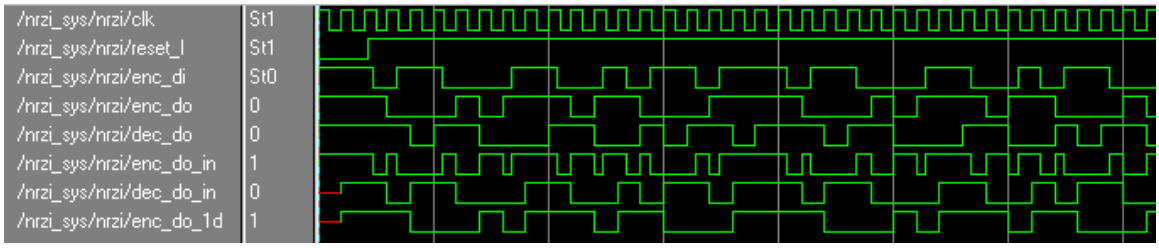
```
reg [15:0]     enc_di_16[1:0];

// Simulation target
nrzi    nrzi(.enc_di(enc_di), .dec_do(dec_do), .reset_l(reset_l), .clk(clk));

//--------------------
// Simulation vector
//--------------------
initial
begin
  $readmemh("enc_di.dat", enc_di_16);

     clk     <= 1'b1;
     reset_l <= 1'b0;
     enc_di  <= 1'b1;
#110 reset_l <= 1'b1;
#10  enc_di  <= 1'b0;

  for(i = 0;  i < 2;  i = i + 1)
  begin
    for(mask = 16'h8000;  mask > 0;  mask = (mask >> 1))
    begin
      enc_di_gen;
    end
  end

#100  $finish;
end

// 20MHz clock generator
always  #25     clk <= ~clk;

//--------
// Tasks
//--------
task  enc_di_gen;
begin
  if((enc_di_16[i] & mask) == 0)  #50  enc_di <= 1'b0;
  else                            #50  enc_di <= 1'b1;
end
endtask

endmodule
```

(2) Target module (nrzi.v)
Making a simple block diagram is recommended.

```
//~~~~~~~~~~~~~~~~~~~~~~~
// NRZI encoder/decoder
//  nrzi.v
//~~~~~~~~~~~~~~~~~~~~~~~
module nrzi(enc_di, dec_do, reset_l, clk);

// I/O definition
input   enc_di;        // NRZI encoder input  (H)
input   reset_l;       // Reset               (L)
input   clk;           // Clock               (X)

output  dec_do;        // NRZI decoder output (H)

// Regs for random logic
reg     enc_do_in, dec_do_in;

// Regs for DFFs
reg     enc_do, enc_do_1d, dec_do;

//---------------
// Random logic
//---------------
```

```
always @(enc_di or enc_do or enc_do_1d)
begin
  enc_do_in <= ~(enc_di    ^ enc_do);
  dec_do_in <= ~(enc_do_1d ^ enc_do);
end

//-------
// DFFs
//-------
always @(posedge(clk) or negedge(reset_l))
begin
  if(reset_l == 1'b0)
  begin
    enc_do <= 1'b1;
    dec_do <= 1'b1;
  end
  else
  begin
    enc_do <= enc_do_in;
    dec_do <= dec_do_in;
  end
end

always @(posedge(clk))
begin
  enc_do_1d <= enc_do;
end

endmodule
```

(b) Verilog simulation result



“enc_di” is an input to the NRZI encoder. This is an original serial data to be encoded.

The NRZI encoder outputs “enc_do”. This serial data encoded is transmitted on serial transmission line and becomes an input of NRZI decoder.

“dec_do” is an output of the NRZI decoder. The serial data exactly same as the original serial data, “enc_di”, must be reproduced after the decoding.

2 bit delay happens between “enc_di” and “dec_do” to achieve stable digital encoding and decoding.