# Synchronization

Alex Yang
Engineering School, NPU

## 1. Single bit synchronizer

### a) TasyncIn > Tclock

```
module sync(
                clk,
                rst,
                asyncIn_i,
                syncOut_o
);
    input       clk;
    input       rst;
    input       asyncIn_i;
    output      syncOut_o;

    reg         syncOut_o;
    reg         oneDelayIn_r;

    always@(posedge clk or negedge rst)begin        //Async. DFF
        if(!rst)begin
            oneDelayIn_r <= 0;
            syncOut_o <= 0;
        else    {syncOut_o, oneDelayIn_r } <= {oneDelayIn_r, asyncIn_i};
    end
endmodule
```

### b) TasyncIn < Tclock

```
module sync(
                clk,
                rst,
                asyncIn_i,
                syncOut_o
);
    input       clk;
    input       rst;
    input       asyncIn_i;
    output      syncOut_o;

    reg         syncOut_o;
    reg         StageOneOut_r;
    reg         StageTwoOut_r;

    wire        StageOneRst;
```

```
    assign          firstStageRst = syncOut_o && ~asyncIn_i;

    always@(posedge asyncIn_i or negedge firstStageRst )begin
        if(!firstStageRst)      StageOneOut_r <= 0;
        else                    StageOneOut_r <= 1;
    end

    always@(posedge clk or posedge rst)begin        //Async. DFF
        if(!rst)begin
            StageTwoOut_r <= 0;
            syncOut_o <= 0;
        else      {syncOut_o, StageTwoOut_r} <= {StageTwoOut_r ,StageOneOut_r};

    end
endmodule
```

## 2. More bits handshaking protocol(4-phase rule)

```
`define   kDataW                    8
`define   kDataB                    `kDataW-1:0

module handShakeSender(
        clk1_i,
        rst1_i,
        dataIn_i,
        sendReady_i,
        ack_i,
        req_o,
        dataOut_o
);
        input               clk1_i;
        input               rst1_i;
        input[`kDataB]      dataIn_i;
        input               sendReady_i;
        input               ack_i;
        output              req_o;
        output[`kDataB]     dataOut_o;

        reg[`kDataB]        dataOut_o;
        reg                 req_o;

        reg                 ack_r;
        reg                 synAck_r;

        reg                 curSt_r;
        reg                 nxtSt_r;
```

```
  parameter             pStart = 1'b0;
  parameter             pWait = 1'b1;

  //Sending synchronized data from input data
  always@(posedge clk1_i)begin
              if(rst1_i)                    dataOut_o <= 0;
              else if(sendReady_i)          dataOut_o <= dataIn_i;
              else                          dataOut_o <= dataOut_o;
  end

  //Synchronizing ack signal from receiver
  always@(posedge clk1_i)begin
              if(rst1_i)begin
                        ack_r <= 0;
                        synAck_r <= 0;
              end
              else      {synAck_r, ack_r} <= {ack_r, ack_i};
end

  //FSM sequential logic
  always@(posedge clk1_i)begin
            if(rst1_i)          curSt_r <= pStart;
            else                curSt_r <= nxtSt_r;
  end
  //FSM combinational logic
  always@(*)begin
            req_o = 0;
            case(curSt_r)
                pStart: begin
                    if(sendReady_i) begin
                        req_o = 1;
                        nxtSt_r = pWait;
                    end
                    else nxtSt_r = pStart;
                end
                pWait: begin
                    if(synAck_r)begin
                        req_o = 0;
                        nxtSt_r = pStart;
                    end
                    else begin
                        req_o = 1;
                        nxtSt_r = pWait;
                    end
                end
                default:begin
```

```
                        nxtSt_r = pStart;
                    end
                endcase
        end
endmodule

`define   kDataW                    8
`define   kDataB                    `kDataW-1:0

module handShakeRec(
        clk2_i,
        rst2_i,
        dataIn_i,
        req_i,
        ack_o
);
        input               clk2_i;
        input               rst2_i;
        input[`kDataB]      dataIn_i;
        input               req_i;
        output              ack_o;

        reg                 ack_o;

        reg                 req_r;
        reg                 synReq_r;

        reg                 load_r;

        reg[`kDataB]        data_r;

        reg[1:0]            curSt_r;
        reg[1:0]            nxtSt_r;

        parameter           pStart = 2'b00;
        parameter           pAck = 2'b01;
        parameter           pDone = 2'b10;

        //Receiving data from input
        always@(posedge clk2_i)begin
                if(rst2_i)          data_r <= 0;
                else if(load_r)         data_r <= dataIn_i;
                else                data_r <= data_r;
        end

        //Synchronizing req signal from sender
```

```
    always@(posedge clk2_i)begin
            if(rst2_i)begin
                    req_r <= 0;
                    synReq_r <= 0;
            end
            else    {synReq_r, req_r} <= {req_r, req_i};
    end

   //FSM sequential logic
    always@(posedge clk2_i)begin
       if(rst2_i)   curSt_r <= pStart;
       else         curSt_r <= nxtSt_r;
    end
    //FSM combinational logic
    always@(*)begin
            ack_o = 0;
            load_r = 0;
          case(curSt_r)
          pStart: begin
            if(synReq_r) begin
                    load_r = 1;
                    ack_o = 1;
                    nxtSt_r = pAck;
            end
            else nxtSt_r = pStart;
          end
          pAck: begin
            ack_o = 1;
            nxtSt_r = pDone;
          end
          pDone: begin
            ack_o = 0;
            nxtSt_r = pStart;
          end
          default:begin
            nxtSt_r = pStart;
          end
       endcase
    end
endmodule


`include "handShakeSender.v"
`include "handShakeRec.v"

module handShakeTop(
                    clk1_i,
```

```
                                clk2_i,
                                rst1_i,
                                rst2_i,
                                dataIn_i,
                                sendReady_i

);
        input           clk1_i;
        input           clk2_i;
        input           rst1_i;
        input           rst2_i;
        input[`kDataB]  dataIn_i;
        input           sendReady_i;

        wire            ack_w;
        wire            req_w;
        wire[`kDataB]   dataOut_w;

        handShakeSender uHandShakeSender(
                                                .clk1_i         (clk1_i),
                                                .rst1_i         (rst1_i),
                                                .dataIn_i       (dataIn_i),
                                                .sendReady_i    (sendReady_i),
                                                .ack_i          (ack_w),
                                                .req_o          (req_w),
                                                .dataOut_o      (dataOut_w)
        );
        handShakeRec    uHandShakeRec(
                                                .clk2_i         (clk2_i),
                                                .rst2_i         (rst2_i),
                                                .dataIn_i       (dataOut_w),
                                                .req_i          (req_w),
                                                .ack_o          (ack_w)
        );

endmodule
```

# 3. FIFO

## A. Sync. FIFO

1. Pointer Moving:
    a) wrPt +1 (wr_en & !full)
    b) rdPt +1 (rd_en & !empty)
2. Pointer Counter:
    a) Write Only (!full && wr_en)

ptCnt = ptCnt +1

b) Read Only (!empty && rd_en)

ptCnt = ptCnt -1

c) Write & Read Simultaneously ((!full && wr_en) &&(!empty && rd_en))

ptCnt = ptCnt

d) Don't write & read

ptCnt = ptCnt

3. One clock:
   a) Write data in (!full && wr_en)
   b) Read data out (!empty && rd_en)
4. Full & Empty
   a) If ptCnt = 0, it is empty
   b) If ptCnt = FIFO depth, it is full

```
`define  kFIFOBitW      8
`define  kFIFOBitS      `kFIFOBitW-1 : 0
`define  kFIFOAdrW      4                       // Bit of address
`define  kFIFODepth     ( 1 << `kFIFOAdrW )     // Depth = 2^4

module syncFIFO(
                    clk,
                    rst,
                    wrEn_i,
                    rdEn_i,
                    dIn_i,
                    dOut_o,
                    empty_o,
                    full_o
);
    input                   clk;
    input                   rst;
    input                   wrEn_i;
    input                   rdEn_i;
    input[`kFIFOBitS]       dIn_i;
    output[`kFIFOBitS]      dOut_o;
    output                  empty_o;
    output                  full_o;

    reg[`kFIFOBitS]         dOut_o;
    reg[`kFIFOBitS]         FIFOMem[`kFIFODepth-1 : 0];

    reg[`kFIFOAdrW:0]       ptCnt;
    reg[`kFIFOAdrW-1:0]     wrPt;
    reg[`kFIFOAdrW-1:0]     rdPt;
```

```
    //Pointer moving
    always@(posedge clk)begin
        if(rst)begin
            wrPt <= 0;
            rdPt <= 0;
        end
        else if(!full_o && wrEn_i)      wrPt <= wrPt + 1;
        else if(!empty_o && rdEn_i)  rdPt <= rdPt +1;
    end
    //Pointer counter
    always@(posedge clk)begin
        if(rst)          ptCnt <= 0;
        else if(!full_o && wrEn_i)      ptCnt <= ptCnt + 1;
        else if(!empty_o && rdEn_i)  ptCnt <= ptCnt - 1;
        else if((!full_o && wrEn_i) &&(!empty_o && rdEn_i))       ptCnt <= ptCnt;
        else             ptCnt <= ptCnt;
    end
    //Write data in
    always@(posedge clk)begin
        if(!full_o && wrEn_i)           FIFOMem[wrPt] <= dIn_i;
    end
    //Read data out
    always@(posedge clk)begin
        if(!empty_o && rdEn_i)        dOut_o <= FIFOMem[rdPt];
    end
    //Full & empty outputs
    assign          full_o = (ptCnt == `kFIFODepth);
    assign          empty_o  = (ptCnt == 0);
endmodule
```

## B. Async. FIFO

1. Port list: wClk, wRst, wEn, wData, wFull / rClk, rRst, rdEn, rData, rEmpty
2. Pointer moving in binary:
   a) wrPt +1 (wr_en & !full)
   b) rdPt +1 (rd_en & !empty)

3. Pointer conversion from binary to Gray code
   wrPt => grayWrPt
   rdPr => grayRdPt

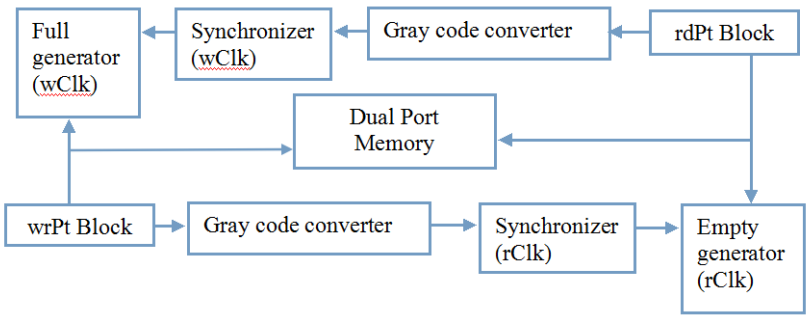
4. Pointer Synchronization:
   grayWrPt => syncGrayWrPt
   grayRdPt => syncGrayRdPt

5. Pointer from Gray code to binary
   syncGrayWrPt => syncBinWrPt
   syncGrayRdPt => syncBin Rdpt

6. Full & Empty
   a) Empty = 1
      i. when syncBinWrPt= = rdPt + 1 if (!empty && rdEn)
      ii. when syncBinWrPt= = rdPt else
   b) Full = 1
      iii. when syncBinRdPt= = wrPt + 2 if (!full && wrEn)
      iv. when syncBinRdPt = = wrPt + 1 else

7. Dual port memory(w/o read-clk) for wrPt / rdPt not syncWrPt / syncRdPt

8. Async. FIFO block diagram




```
`define  kFIFOBitW      8
`define  kFIFOBitS      `kFIFOBitW-1 : 0
`define  kFIFOAdrW      4                       // Bit of address
`define  kFIFODepth     ( 1 << `kFIFOAdrW )     // Depth = 2^4

module asyncFIFO(
                        wClk,
                        wRst,
                        wrEn_i,
                        wData_i,
                        wFull_o,

                        rClk,
                        rRst,
                        rdEn_i,
                        rData_o,
                        rEmpty_o
);
```

```
input                   wClk;
input                   wRst;
input                   wrEn_i;
input[`kFIFOBitS]       wData_i;
output                  wFull_o;
reg                     wFull_o;

input                       rClk;
input                       rRst;
input                       rdEn_i;
output[`kFIFOBitS]          rData_o;
reg[`kFIFOBitS]             rData_o;
output                      rEmpty_o;
reg                         rEmpty_o;

reg[`kFIFOAdrW-1:0]     wrPt;
wire[`kFIFOAdrW-1:0]    grayWrPt;
reg[`kFIFOAdrW-1:0]     delayGrayWrPt;
reg[`kFIFOAdrW-1:0]     syncGrayWrPt;
wire[`kFIFOAdrW-1:0]    syncBinWrPt;

reg[`kFIFOAdrW-1:0]     rdPt;
wire[`kFIFOAdrW-1:0]    grayRdPt;
reg[`kFIFOAdrW-1:0]     delayGrayRdPt;
reg[`kFIFOAdrW-1:0]     syncGrayRdPt;
wire[`kFIFOAdrW-1:0]    syncBinRdPt;

reg[`kFIFOBitS]             FIFOMem[`kFIFODepth-1 : 0];

//Pointer moving in wClk/rClk
always@(posedge wClk)begin
    if(wRst)            wrPt <= 0;
    else if(!wFull_o && wrEn_i)       wrPt <= wrPt + 1;
end
always@(posedge rClk)begin
    if(wRst)            rdPt <= 0;
    else if(!rEmpty_o && rdEn_i)      rdPt <= rdPt - 1;
end

//Binary to Gray converter
function [`kFIFOBitS]     bin2Gray;
    input[`kFIFOBitS]     binDin;
        bin2Gray = binDin ^ (binDin>>1);
endfunction

assign          grayWrPt = bin2Gray(wrPt);
```

```
assign          grayRdPt = bin2Gray(rdPt);

//Write/Read pointer synchronizer
always@(posedge wClk)begin
    if(wRst)begin
        delayGrayWrPt <= 0;
        syncGrayWrPt <=0;
    end
    else    {syncGrayWrPt, delayGrayWrPt}<={delayGrayWrPt, grayWrPt };
end
always@(posedge rClk)begin
    if(rRst)begin
        delayGrayRdPt <= 0;
        syncGrayRdPt <=0;
    end
    else    {syncGrayRdPt, delayGrayRdPt}<={delayGrayRdPt, grayRdPt };
end

//Gray to binary converter
function [`kFIFOBitS]   gray2Bin;
    input[`kFIFOBitS]   grayDin;
    integer             i;
    begin
        for(i = 0; i < `kFIFOBitW; i = i+1)
            gray2Bin[i] = ^(grayDin >> i);
    end
endfunction

assign          syncBinWrPt = gray2Bin(syncGrayWrPt );
assign          syncBinRdPt = gray2Bin(syncGrayRdPt );

//Full & empty outputs
always@(posedge rClk)begin
    if(rRst)        rEmpty_o <= 0;
    else if(!rEmpty_o && rdEn_i)        rEmpty_o <= (syncBinWrPt == rdPt + 1);
    else        rEmpty_o <= (syncBinWrPt == rdPt);
end
always@(posedge wClk)begin
    if(wRst)        wFull_o <= 0;
    else if(!wFull_o && wrEn_i)         wFull_o <= (syncBinRdPt == wrPt + 2);
    else        wFull_o <= (syncBinRdPt == wrPt + 1);
end

//Write-in/Read-out from memory
always@(posedge wClk)begin
    if(!wFull_o && wrEn_i)          FIFOMem[wrPt] <= wData_i;
```

```
    end
    always@(posedge rClk)begin
        if(!rEmpty_o && rdEn_i)       rData_o <= FIFOMem[rdPt];
    end
endmodule
```

# 4. FIFO Depth Calculation

a) Equation:

WritingTime = number of data(needed for write-in) * ($1/f_{write}$)

ReadingTime = number of data(same as above) * ($1/f_{read}$)

TimeDifference = | WritingTime - ReadingTime |

Depth = TimeDifference * bigger($f_{write}$, $f_{read}$)

b) Examples:

i. $f_{write}$=30MHz; $f_{read}$=40MHz; 10 data should be transferred by FIFO

WritingTime = 10 * (1/30MHz); ReadingTime = 10 * (1/40MHz)

Depth = [10 * (1/30MHz) -10 * (1/40MHz)] * 40MHz

= 3.333 ≈ 4

ii. $f_{write}$=100MHz, 80data/100cycles

$f_{read}$=80MHz, 80data/80cycles

Depth =| [80 * (1/100MHz) -80 * (1/80MHz)] |* 100MHz

= 20

iii. Continuous write-in: 80 data/80 cycles

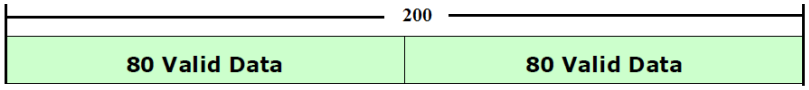
Burst write-in(20 data every time): 80 data/100 cycles, no data between two burst slots.

Read out continuously: 8data/10cycles

$f_{burstWrite}$ = $f_{read}$ = 8 data/100 cycles

But $f_{continuous\ Write}$ > $f_{read}$

Considering worse case in 200 cycles , there are 160 data. But it may be 160 data in 160 cycles if continuously written in.




Depth = 160 * (80/80cycles) -160 * (8/10cycles) = 32