

# DIGITAL DESIGN WITH FPGA CAMP

DAY 1 BASIC FPGA AND VHDL

# DIGITAL LOGIC OVERVIEW

# DIGITAL LOGIC OVERVIEW (1)

- เลขฐานสอง (Binary) และเลขฐาน 16 (Hexadecimal) ซึ่งเป็นเลขฐานหลักที่ใช้ในงานออกแบบวงจร digital
- การบวก ลบ คูณ หารบนเลขฐานสอง (unsigned, signed, 2's complement) ซึ่งเป็นเบื้องหลังของการออกแบบตัวประมวลผล (CPU) สำหรับงานที่ต้องอาศัยการคำนวณต่าง ๆ
- Logic Gate พื้นฐาน เช่น AND OR NOT และการออกแบบวงจร โดยใช้ AND/OR/NOT ตาม truth table หรือ equation ที่กำหนดไว้ เริ่มจากการใช้ Karnaugh map
- ตัวอย่าง Application ของการออกแบบวงจรด้วย logic gate ต่าง ๆ นี้ เช่น สร้างวงจรสำหรับการบวก ลบ คูณ หาร หรือวงจรใด ๆ ได้ทั้งหมด และเริ่มแนะนำชิพสำเร็จรูปที่ซับซ้อนกว่า logic gate เช่น Full Adder/Subtractor, Multiplexer, Decoder, Encoder

## DIGITAL LOGIC OVERVIEW (2)

- วงจรที่ออกแบบ เริ่มมีสัญญาณ clock มา เพื่อกำกับการทำงาน ให้ทำงานหรือมีการเปลี่ยนแปลงสัญญาณตามขอบขาขึ้น/ขาลงของสัญญาณ clock และแนะนำวงจรพื้นฐานเพิ่มเติม เช่น Flip-Flop ชนิดต่าง ๆ (D Flip-Flop/S-R Flip-Flop), Shift Register, Counter
- การมี Flip Flop ที่ทำงานตามสัญญาณ clock ก่อให้ความทรงจำขึ้นมา ว่าสัญญาณปัจจุบันมีค่าเท่าไร และเราต้องการออกแบบให้ค่าต่อไปเท่ากับเท่าไร เราสามารถกำหนดได้หมด และออกแบบวงจรให้แสดงค่าตามที่เราต้องการได้ โดยการใช้ Flip Flop ทำงานคู่กับ Logic พื้นฐานอื่น ๆ
- ขยาย Application ของการออกแบบนี้ ไปเป็นระบบต่าง ๆ ที่สมบูรณ์ได้ โดยความคุ้นเคยของเรา มักจะต้องการทำงานตามลำดับ ๆ จึงโยงไปสู่การออกแบบ State machine

## DIGITAL LOGIC OVERVIEW (3)

- เมื่อวงจรของเราซับซ้อนมาก ๆ การใช้ชิพ logic สำเร็จรูปมานั้น ไม่สะดวกเลย และแก้ไขวงจรได้ยาก เสียเวลานาน จึงเป็นที่มาของชิพ ที่สามารถโปรแกรม logic ต่าง ๆ เข้าไปได้สะดวก ใช้เวลาน้อย แก้ไขได้ง่าย ผ่าน tool ที่กำหนด เรียกชิพเหล่านี้ว่า Programmable Logic Device
- FPGA (Field Programmable Gate Array) เป็นชิพที่มีความสามารถสูง ประกอบด้วย logic ภายในจำนวนมาก (หมื่น/แสน/ล้าน gate ตามขนาดของ device) แต่มีราคาสูง
- การออกแบบวงจร digital บน FPGA นั้น สามารถทำได้ง่าย โดยใช้การเขียนเป็น CODE เพื่อโปรแกรมให้ชิพ กลายเป็น logic ต่าง ๆ ตามที่เราต้องการ แทนการต่อวงจรภายนอกด้วยชิพสำเร็จรูปมากมาย
- ภาษาที่ใช้ในการออกแบบที่นิยมกัน คือ VHDL และ Verilog

## DIGITAL LOGIC OVERVIEW (4)

- VHDL/Verilog ใช้สำหรับเขียนเพื่อโปรแกรมให้ชิพกลายเป็นวงจร digital ต่าง ๆ ที่เรารู้จักกัน เช่น Multiplexer, RAM, ROM, Counter ดังนั้นทุก ๆ บรรทัดของการเขียน code คือ การสร้าง hardware เพิ่มขึ้นภายใน FPGA
- การเขียน VHDL/Verilog จึงต้องอาศัยความรู้ของการออกแบบวงจร digital อยู่ด้วย เพื่อให้ทำงานได้ตาม timing diagram ที่เราต้องการ และใช้ทรัพยากรใน FPGA ให้คุ้มค่าที่สุด

# FPGA STRUCTURE

4-input LUT

1) 4-input LUT นี้สามารถทำได้ เช่น ANR โดย LUT 1 ตัวนี้จะสามารถ output ออกมาจำนวน



1) 4-input LUT นี้สามารถโปรแกรมให้เป็น logic พื้นฐานต่าง ๆ ได้ เช่น ANR, OR, NOT, Comparator, Multiplexer โดย LUT 1 ตัวนี้จะรับ input ได้สูงสุด 4 bit เพื่อสร้าง output ออกมาจำนวน 1 bit



# 4-INPUT LUT

Signal Name	xx	xx	xx	xx	xx
	data1	data2	data3	data4	Out
	0	0	0	0	x
	0	0	0	1	x
	0	0	1	0	x
	0	0	1	1	x
	0	1	0	0	x
	0	1	0	1	x
	0	1	1	0	x
	0	1	1	1	x
	1	0	0	0	x
	1	0	0	1	x
	1	0	1	0	x
	1	0	1	1	x
	1	1	0	0	x
	1	1	0	1	x
	1	1	1	0	x
	1	1	1	1	x

การเขียน code ลงบน FPGA เพื่อโปรแกรม 4-input LUT ให้ทำงานตามที่เรากออกแบบนั้น ตัว FPGA จะสังเคราะห์วงจร โดยมีการทำงานทั้งหมด 2 ขั้นตอน ได้แก่

1. Route สัญญาณ input ของ code เราไปที่สัญญาณ data1 data2 data3 data4 และให้ output ของ code เราไปเชื่อมต่อกับสัญญาณ out
2. กำหนดค่าของ out ให้มีค่าเป็น 1 หรือ 0 ตาม code ที่เราเขียน

# EXAMPLE1: 2-BIT COMPARATOR

Signal Name	A(0)	A(1)	B(0)	B(1)	C
	data1	data2	data3	data4	Out
	0	0	0	0	1
	0	0	0	1	0
	0	0	1	0	0
	0	0	1	1	0
	0	1	0	0	0
	0	1	0	1	1
	0	1	1	0	0
	0	1	1	1	0
	1	0	0	0	0
	1	0	0	1	0
	1	0	1	0	1
	1	0	1	1	0
	1	1	0	0	0
	1	1	0	1	0
	1	1	1	0	0
	1	1	1	1	1

เมื่อเราเขียน code ดังต่อไปนี้

$C = '1' \text{ when } A(1 \text{ downto } 0) = B(1 \text{ downto } 0) \text{ else } '0';$

ซึ่งคือการสร้างวงจร comparator เพื่อเปรียบเทียบสัญญาณ A และ B ที่มีขนาด 2 bit นี้ ว่ามีค่าเท่ากันหรือไม่ ถ้าเท่ากัน ให้ค่า C='1' แต่ถ้าไม่เท่ากันให้ค่า C='0' ตัว FPGA มาสามารถสังเคราะห์วงจรนี้ได้ โดยมีขั้นตอนดังนี้

1. Route สัญญาณ ให้ data1=A(0), data2=A(1), data3=B(0), data4=B(1) และ out=C

2. กำหนดค่าของ out ให้มีค่าเป็น '1' เมื่อ A=B และให้ช่องที่เหลือมีค่าเป็น '0'

## EXAMPLE2: 2-BIT ADDER

เมื่อเราเขียน code ดังต่อไปนี้

$C(1 \text{ downto } 0) = A(1 \text{ downto } 0) + B(1 \text{ downto } 0);$

จะสังเคราะห์วงจรด้วย 4-input LUT ทั้งหมด 2 ตัวได้ดังนี้

Signal Name	A(0)	xx	B(0)	xx	C(0)
	data1	data2	data3	data4	Out
	0	0	0	0	0
	0	0	0	1	0
	0	0	1	0	1
	0	0	1	1	1
	0	1	0	0	0
	0	1	0	1	0
	0	1	1	0	1
	0	1	1	1	1
	1	0	0	0	1
	1	0	0	1	1
	1	0	1	0	0
	1	0	1	1	0
	1	1	0	0	1
	1	1	0	1	1
	1	1	1	0	0
	1	1	1	1	0

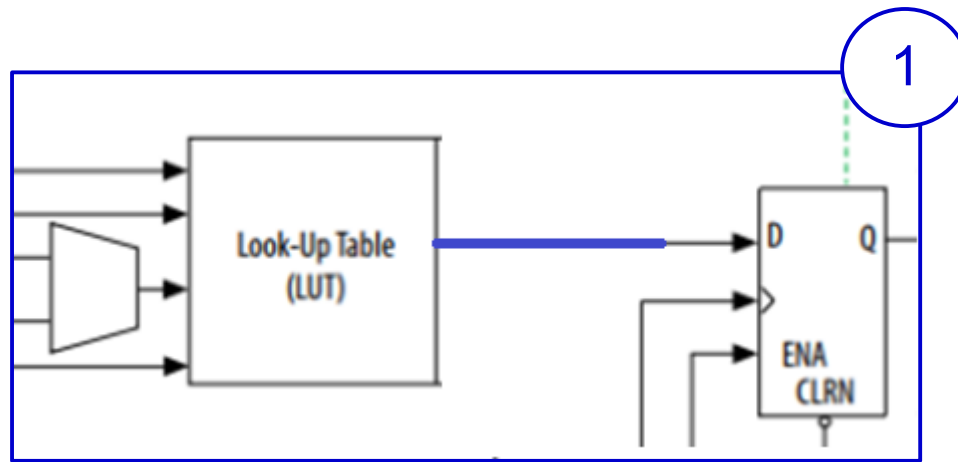
Signal Name	C(0)	A(1)	xx	B(1)	C(1)
	data1	data2	data3	data4	Out
	0	0	0	0	0
	0	0	0	1	1
	0	0	1	0	0
	0	0	1	1	1
	0	1	0	0	1
	0	1	0	1	0
	0	1	1	0	1
	0	1	1	1	0
	1	0	0	0	1
	1	0	0	1	0
	1	0	1	0	1
	1	0	1	1	0
	1	1	0	0	0
	1	1	0	1	1
	1	1	1	0	0
	1	1	1	1	1

(วงจรมี 3 bit ไม่สมบูรณ์  
เพราะ output ต้องมี 3 bit  
ถึงจะได้ output ครบ  
โจทย์นี้ใช้แสดงตัวอย่าง  
การสังเคราะห์เท่านั้น)

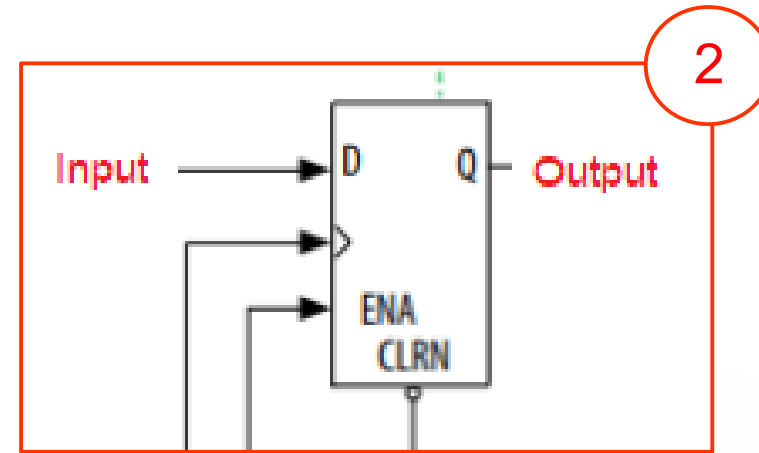
# FLIP FLOP

ตัว FLIP FLOP ที่อยู่ใน LE จะถูกใช้งานได้ 2 กรณี คือ


1. ใช้เชื่อมต่อกับสัญญาณ output ที่ออกมาจาก 4-input LUT เพื่อให้ output ที่ออกมาจาก LE นี้ เปลี่ยนค่าตามจังหวะขอบขาขึ้น/ขาลงของ Clock
2. เมื่อ code ของเราออกแบบเป็น Flip Flop เปล่า ๆ อยู่แล้ว เช่น D Flip Flop หรือ shift register



Input --> LUT --> FF --> Output



Input --> FF --> Output



# VHDL LANGUAGE (FPGA) VS C LANGUAGE (PROCESSOR)

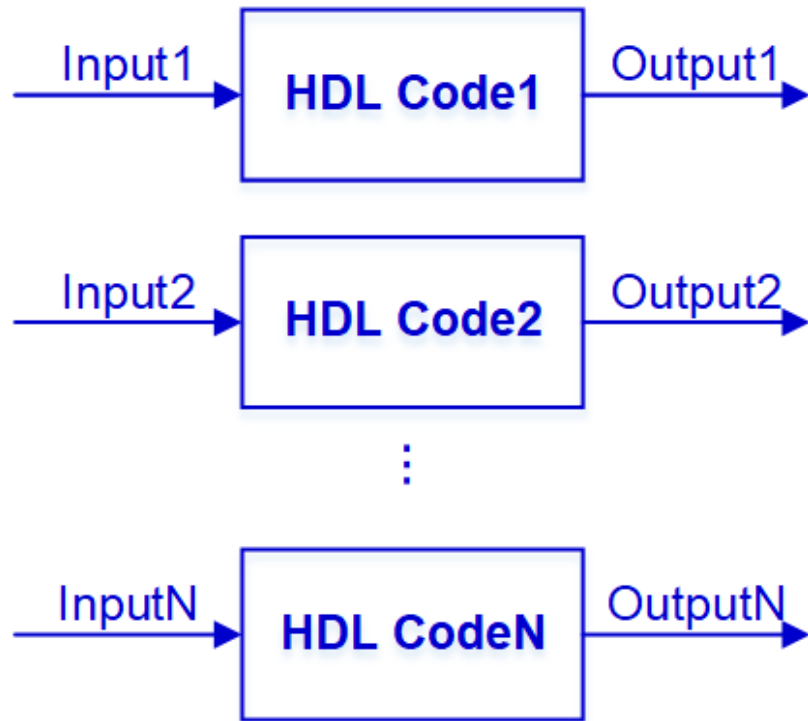
## VHDL ON FPGA

- เป็นการเขียน code เพื่อสร้าง hardware ตามที่เราต้องการ
- Hardware แต่ละตัวจะทำงานอิสระต่อกัน จึงประมวลผลได้พร้อมกัน
- ไม่ได้ทำงานจากบนลงล่างหากไม่ได้อยู่ภายใต้ Process เดียวกัน แต่จะทำงานพร้อม ๆ กัน ดังนั้นสามารถเขียน code สลับบรรทัดกันได้ในบางกรณี (Concurrent)
- ถูกจำกัดด้วยจำนวน LE ที่มีใน FPGA หากต้องใช้ hardware มากขึ้น ต้องใช้ชิพใหญ่ขึ้น
- เหมาะสมกับงานที่ต้องการความเร็วสูง ที่สามารถแยกงานออกมาประมวลผลพร้อม ๆ กันได้หลาย ๆ งาน

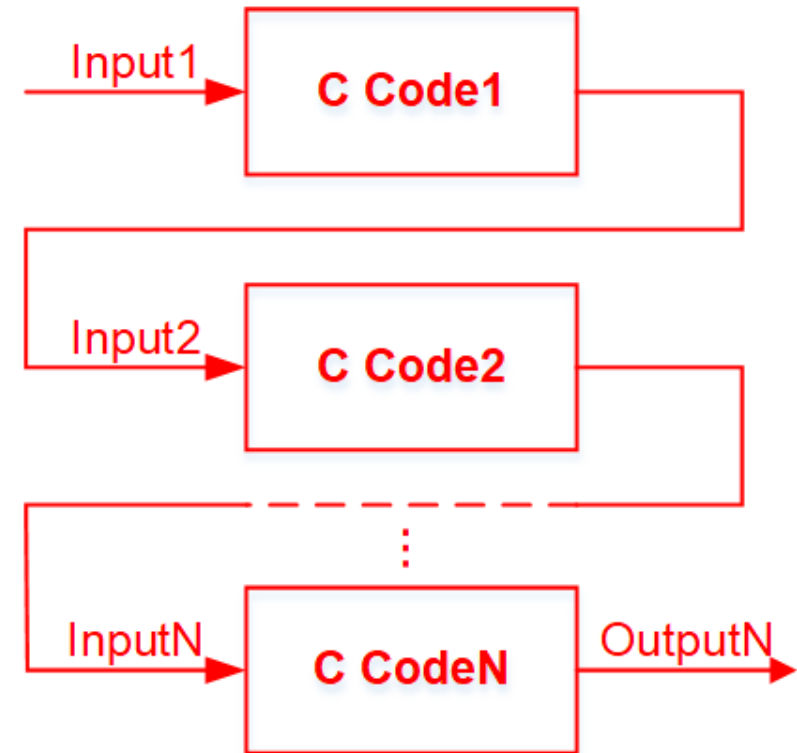
## C ON PROCESSOR

- เป็นการเขียน software ภายใต้ hardware ที่ถูกกำหนดไว้แล้วว่าสามารถทำคำสั่งอะไรได้บ้างในแต่ละช่วงเวลา
- ประมวลผลได้เพียงทีละคำสั่งใน 1 ช่วงเวลา
- Software ที่เขียนคือการป้อนคำสั่งลงไปทีละคำสั่ง ดังนั้นจะทำงานตามลำดับคือ จากบนลงล่าง (Sequential)
- ถูกจำกัดด้วยขนาดของหน่วยความจำที่จะเก็บลำดับของคำสั่ง หากหน่วยความจำไม่พอ ต้องใช้ชิพใหญ่ขึ้น
- เหมาะสมกับงานที่มีความซับซ้อนสูง และคำสั่งหรือข้อมูลที่ต้องประมวลผลในลำดับต่อไปจะขึ้นอยู่กับผลลัพธ์ที่กำลังคำนวณอยู่ในปัจจุบัน ไม่สามารถทำก่อนล่วงหน้าได้

## VHDL ON FPGA



## C ON PROCESSOR





VHDL



# CODE STRUCTURE BY VHDL

ประกอบด้วย 3 ส่วน คือ

- 1) Package: Library สำเร็จต่าง ๆ ที่เราจะใช้งาน
- 2) Entity: กำหนด input/output ของ code
- 3) Architecture: Logic design ของ code

```
library IEEE;  
use IEEE.std_logic_1164.ALL;  
use IEEE.STD_LOGIC_ARITH.all;  
use IEEE.STD_LOGIC_UNSIGNED.all;
```

Package

```
Entity AdderTree Is  
Port  
(  
    A      : in    std_logic_vector( 7 downto 0 );  
    B      : in    std_logic_vector( 7 downto 0 );  
    C      : in    std_logic_vector( 7 downto 0 );  
    D      : in    std_logic_vector( 7 downto 0 );  
    Z      : out   std_logic_vector( 7 downto 0 );  
);  
End Entity AdderTree;
```

Entity

```
Architecture rtl Of AdderTree Is  
  
    signal AB      : std_logic_vector( 7 downto 0 );  
    signal CD      : std_logic_vector( 7 downto 0 );  
  
Begin  
  
    AB <= A + B;  
    CD <= C + D;  
  
    Z  <= AB + CD;  
  
End Architecture rtl;
```

Architecture

# 1. PACKAGE

```
library IEEE;  
use IEEE.std_logic_1164.ALL;  
use IEEE.STD_LOGIC_ARITH.all;  
use IEEE.STD_LOGIC_UNSIGNED.all;
```

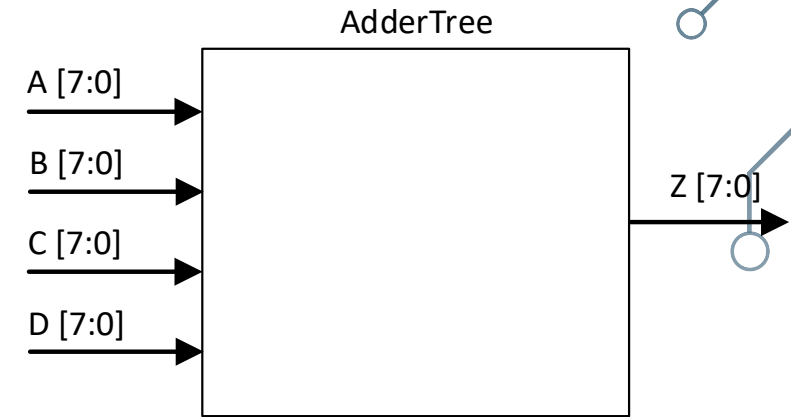
## Package

- ประกาศการเรียกใช้ library จากภายนอก เพื่อใช้ HDL code ที่มีอยู่แล้ว (เหมือน #include ในภาษา C)
- พื้นฐานที่ใช้งานในเกือบทุก code ได้แก่
  - 1) IEEE.std\_logic\_1164.all : กำหนดค่า operation ต่าง ๆ ของสัญญาณ std\_logic, std\_logic\_vector ให้มีค่า '0', '1', 'Z', 'X' และค่า output ต่าง ๆ ที่ควรจะได้ หลังทำ logical operation เช่น AND OR NOT XOR XNOR
  - 2) IEEE.STD\_LOGIC\_ARITH.all: Operation ทางคณิตศาสตร์ เช่น บวก ลบ คูณ หาร รวมถึง function ชื่อ conv\_std\_logic\_vector สำหรับแปลงสัญญาณชนิด integer ให้เป็น std\_logic\_vector
  - 3) IEEE.STD\_LOGIC\_UNSIGNED.all: มี function ชื่อ conv\_integer สำหรับแปลงสัญญาณชนิด std\_logic\_vector ให้เป็น integer

## 2. ENTITY

```
Entity AdderTree Is
Port
(
    A      : in    std_logic_vector( 7 downto 0 );
    B      : in    std_logic_vector( 7 downto 0 );
    C      : in    std_logic_vector( 7 downto 0 );
    D      : in    std_logic_vector( 7 downto 0 );
    Z      : out   std_logic_vector( 7 downto 0 );
);
End Entity AdderTree;
```

Entity



- ประกาศ Port ทั้งหมดที่เชื่อมต่อกับวงจรภายนอก
- ประเภทของสัญญาณมีใช้งานอยู่ 3 แบบ คือ in (สำหรับรับเข้า) out (สำหรับส่งออก) และ inout (สำหรับรับและส่งในเวลาต่างกัน)
- Inout ใช้ประกาศกับสัญญาณที่เชื่อมต่อกับภายนอกชิพเท่านั้น ไม่เชื่อมต่อกับ code อื่น ๆ ที่อยู่ในชิพเดียวกัน

ถ้าจะเชื่อมต่อกันภายในชิพ ให้ใช้สัญญาณ in และ out เท่านั้น

- สัญญาณ out จะไม่สามารถอ่านกลับมาประมวลผลภายใน code เดียวกันได้ ต้องสร้าง signal มารองรับก่อน ใช้เป็นตัวกลางที่สามารถส่งออกไปเป็น output ได้และอ่านกลับมาเพื่อประมวลผลต่อได้
- หากต้องการรับ parameter จากภายนอกมาเป็นค่าคงที่ ให้ประกาศ generic เพิ่มเติมเพื่อรับค่าคงที่เข้ามา (ไม่มีแสดงในตัวอย่าง)

# การใช้ SIGNAL มาเป็นตัวกลางเพื่อสร้าง OUTPUT และอ่านค่ากลับมาได้

```
Entity IncCnt Is
Port
(
    Clk      : in    std_logic;
    RstB     : in    std_logic;
    En       : in    std_logic;
    Dout     : out   std_logic_vector(3 downto 0)
);
End Entity IncCnt;
```

```
Architecture rtl Of IncCnt Is
Begin
    u_Output : Process (Clk) is
    begin
        if ( rising_edge(Clk) ) then
            if ( RstB = '0' ) then
                Dout      <= (others=>'0');
            else
                if ( En = '1' ) then
                    Dout  <= Dout+1;
                else
                    Dout  <= Dout;
                end if;
            end if;
        end if;
    end Process u_Output;
End Architecture rtl;
```

Dout คือ output ที่ไม่สามารถ  
อ่านกลับมาเพื่อ +1 ได้

ERROR

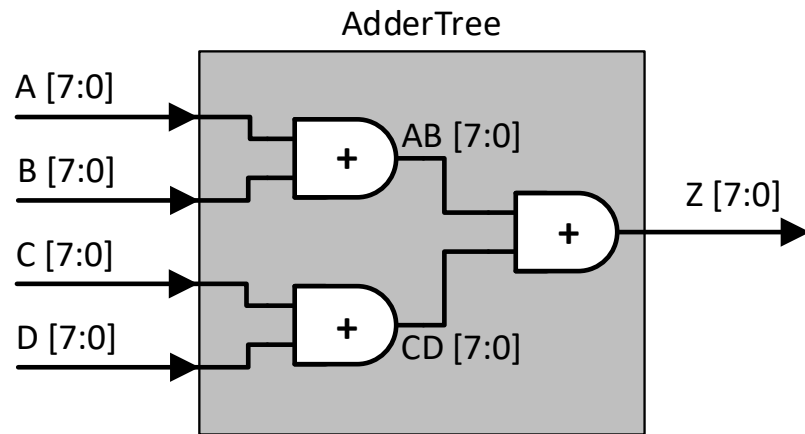
```
Architecture rtl Of IncCnt Is
    signal rDout      : std_logic_vector(3 downto 0);
Begin
    u_Output : Process (Clk) is
    begin
        if ( rising_edge(Clk) ) then
            if ( RstB = '0' ) then
                rDout      <= (others=>'0');
            else
                if ( En = '1' ) then
                    rDout  <= rDout+1;
                else
                    rDout  <= rDout;
                end if;
            end if;
        end if;
    end Process u_Output;

    Dout  <= rDout;
End Architecture rtl;
```

OK

สร้างสัญญาณชื่อ rDout เพื่อเชื่อม  
ต่อไปที่ output และเราสามารถอ่าน  
ค่าที่เราสร้างกลับมาได้

### 3. ARCHITECTURE



Architecture rtl Of AdderTree Is

```
signal AB      : std_logic_vector( 7 downto 0 );  
signal CD      : std_logic_vector( 7 downto 0 );
```

Begin

```
AB <= A + B;  
CD <= C + D;  
Z  <= AB + CD;
```

End Architecture rtl;

1

Architecture

2

ทั้ง 3 บรรทัดทำงานพร้อมกัน และ  
สามารถเขียนสลับบรรทัดกันได้

แบ่งออกเป็น 2 ส่วน

1) ส่วนที่ประกาศสัญญาณที่จะใช้งาน โดยจะประกาศสิ่งที่จะใช้งาน ได้แก่ signal (สัญญาณที่จะสร้าง) constant (ค่าคงที่) type (ชนิดของสัญญาณที่จะใช้เพิ่มเติม) รวมถึง component (เรียกใช้ entity ที่เขียนด้วย HDL ไฟล์อื่น ๆ) ทั้งหมดนี้ จะประกาศอยู่ก่อน Begin

2) ส่วนที่เป็น logic design ทั้งหมด ซึ่งจะเป็น HDL เพื่อสร้าง signal ที่ประกาศไว้ด้วย operation ต่าง ๆ รวมถึงการเชื่อมต่อ signal ที่ประกาศไว้เข้ากับ component ที่เรียกใช้ เพื่อให้ component สร้างสัญญาณตามที่เราต้องการ ทั้งหมดนี้จะออกแบบหลัง Begin



# PART1 OF ARCHITECTURE

CONSTANT DECLARATION  
COMPONENT DECLARATION  
SIGNAL DECLARATION

# DECLARATION

Architecture HTWTestBench Of TbTxSerial Is

-- Constant Declaration

```
constant    tClk      : time := 20 ns;
```

-- Component Declaration

```
Component TxSerial Is
Port
(
  RstB      : in    std_logic;
  Clk       : in    std_logic;

  TxFfEmpty : in    std_logic;
  TxFfRdData : in   std_logic_vector( 7 downto 0 );
  TxFfRdEn  : out   std_logic;

  SerDataOut : out   std_logic
);
End Component TxSerial;
```

-- Signal Declaration

```
signal TM      : integer range 0 to 65535;
```

Begin

ประกาศค่าคงที่ที่ใช้ในวงจร

(time เป็นชนิดตัวแปรที่ใช้ใน testbench สำหรับ simulation เท่านั้น  
ไม่สามารถสังเคราะห์เป็นวงจรจริงได้)

ประกาศวงจรย่อย ๆ (module) ที่เราเคยออกแบบไว้  
แล้ว และจะเรียกใช้งานซ้ำใน code นี้ โดยการต่อเชื่อม  
signal เข้ากับ input/output ของ module นี้

ประกาศชื่อสัญญาณภายในทั้งหมดที่จะสร้างและใช้งาน  
ใน code นี้ (ไม่แนะนำให้ใช้ variable ในการออกแบบวงจร  
ที่จะสังเคราะห์จริง แนะนำให้ใช้ variable เฉพาะ  
testbench สำหรับ simulation เท่านั้น)



# STANDARD DATA TYPE

integer: ต้องกำหนด range ของ integer

แนะนำให้กำหนดเป็นค่า 0 ถึง  $(2^n) - 1$  เพื่อรองรับค่าทั้งหมดที่เป็นไปได้ตามจำนวน bit ที่จะใช้งาน

```
signal TM : integer range 0 to 65535;
```

จะได้สัญญาณขนาด 16-bit เพื่อรองรับค่า 0-65535

std\_logic: สัญญาณขนาด 1 bit ซึ่งมีค่าได้หลากหลาย ที่นิยมใช้งานจริง ๆ คือ '0', '1', 'Z'  
(High impedance สำหรับ inout port)

```
signal rBuadEnd : std_logic;
```

std\_logic\_vector: สัญญาณเมื่อมีขนาดหลาย bit  
สามารถประกาศขนาด bit ด้วย downto หรือ to

```
signal rBuadCnt0 : std_logic_vector( 9 downto 0 );  
signal rBuadCnt1 : std_logic_vector( 0 to 9 );
```

9 downto 0 : Bit 0 เป็น LSB และ bit 9 เป็น MSB

0 to 9 : Bit 9 เป็น LSB และ bit 0 เป็น MSB



# NEW DATA TYPE

type: ประกาศชนิดสัญญาณใหม่ เช่น การประกาศชื่อ state ที่จะใช้งานทั้งหมด

```
type      SerStateType is
(
    stIdle   ,
    stStart  ,
    stData   ,
    stStop
);
signal    rState : SerStateType;
```

array: เมื่อต้องการใช้สัญญาณชนิดเดียวกันหลาย ๆ ตัว ใช้ในการสร้าง RAM หรือ simulation

```
type      RamType is array (0 to 15) of std_logic_vector( 7 downto 0 );
signal    Ram16x8      : RamType;
```

```
Ram16x8 (0)  <= x"A5";
```

# PART2 OF ARCHITECTURE

## LOGIC DESIGN

## การออกแบบวงจรดิจิทัลด้วยภาษา VHDL

- การออกแบบวงจร digital คือ การออกแบบแต่ละ signal ที่ได้ประกาศไว้ว่าจะมีค่าอะไรได้บ้างตามเงื่อนไขต่าง ๆ ซึ่งข้อกำหนดและเงื่อนไขเหล่านั้น จะทำให้เกิดการสร้าง logic ชนิดต่าง ๆ ขึ้นมา
- สัญญาณ 1 สัญญาณจึงถูกสร้างด้วยชุด logic 1 ชุด
- เมื่อมีการสร้างสัญญาณจำนวน N สัญญาณ ก็จะมี logic ที่ถูกออกแบบขึ้นมาในรูปแบบเงื่อนไขที่ต่างกันทั้งหมด N logic โดยแต่ละ logic จะทำงานอิสระ ทำงานตลอดเวลา ไม่รอกัน
- ข้อแนะนำ เราจะแยก code ให้ออกแบบเพื่อสร้างสัญญาณทีละสัญญาณ แต่ละสัญญาณจะมี code แยกกันเด็ดขาด เพื่อให้เห็นการทำงานของสัญญาณแต่ละสัญญาณได้ชัดเจน และสื่อถึงการทำงานที่อิสระต่อกัน

# OPERATOR

# RELATIONAL OPERATOR

## Relational Operators

=	equal to:
/=	not equal to
<	less than
<=	less than equal
>	greater than
>=	greater than equal:

```
-- Example1 (Equal + Signal assignment)
wEqualTo    <=  '1' when (DataInA=DataInB)
            else '0';

-- Example2 (Less than equal + Signal assignment)
wLessEq     <=  '1' when (DataInA<=DataInB)
            else '0';
```

## Assignment

<=	signal assignment
:=	variable assignment, signal initialization

Reference: [http://atlas.physics.arizona.edu/~kjohns/downloads/vhdl/VHDL\\_Lang.pdf](http://atlas.physics.arizona.edu/~kjohns/downloads/vhdl/VHDL_Lang.pdf)

# LOGICAL OPERATOR

## Logical Operators

**not**  
**and**  
**or**  
**nand**  
**nor**  
**xor**  
**xnor**

**Concatenation, &**

```
-- Example3 (AND)
wAND(0)          <= DataInA(0) and DataInB(0);
wAND(1)          <= DataInA(1) and DataInB(1);
wAND(3 downto 2) <= DataInA(3 downto 2) and DataInB(3 downto 2);

-- Example4 (OR)
wOR(0)           <= DataInA(1) or DataInB(1);
wOR(1)           <= DataInA(1) or DataInB(1);
wOR(3 downto 2)  <= DataInA(3 downto 2) or DataInB(3 downto 2);

-- Example5 (Concat)
wConcat(3 downto 0) <= DataInA(1 downto 0) & DataInB(1 downto 0);
```

Reference: [http://atlas.physics.arizona.edu/~kjohns/downloads/vhdl/VHDL\\_Lang.pdf](http://atlas.physics.arizona.edu/~kjohns/downloads/vhdl/VHDL_Lang.pdf)

# STATEMENT

# 1. CONCURRENT STATEMENT

การเขียน code จะสามารถเขียนได้ 2 แบบ

## 1) Concurrent Statement

- จะเป็น code ที่ใช้สร้างสัญญาณโดยไม่ภายใต้ process
- กำหนดค่า โดยใช้ operator ต่าง ๆ ได้โดยตรงเลย
- สามารถสร้างสัญญาณภายใต้เงื่อนไขได้ โดยใช้ชุดคำสั่ง when ... else
- แต่ละ statement ที่ออกแบบสำหรับแต่ละสัญญาณ จะมีการทำงานพร้อม ๆ กัน  
ไม่มีลำดับก่อน/หลัง

ASumB และ ASubB จะทำงานพร้อมกัน และ  
สามารถเขียนสลับบรรทัดกันได้ ไม่มีลำดับก่อน/หลัง

```
ASumB(3 downto 0)  <= ('0'&A(2 downto 0)) + ('0'&B(2 downto 0));  
ASubB(3 downto 0)  <= ('0'&A(2 downto 0)) - ('0'&B(2 downto 0));
```

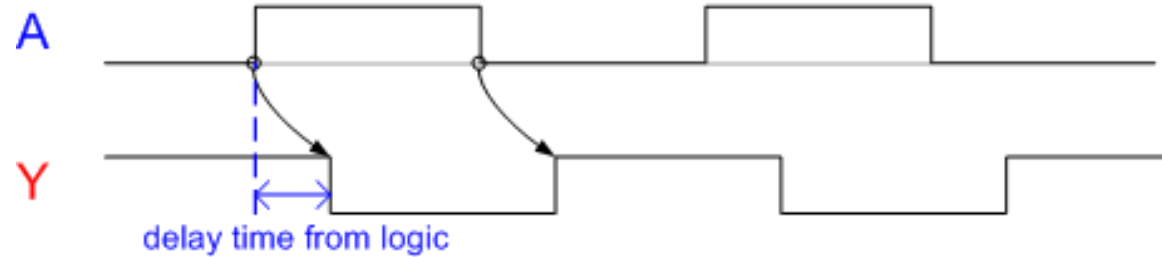
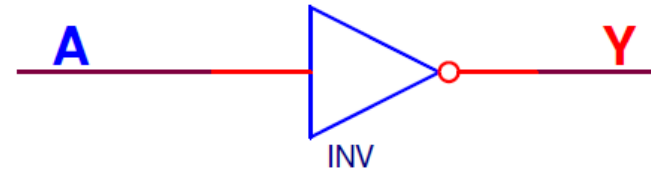


# NOT LOGIC

Truth table

Input	Output
A	Y
0	1
1	0

Logic diagram



Timing diagram ของสัญญาณ Y ที่วาด จะแสดง delay time ไว้หนึ่ง เพื่อเตือนให้ทราบว่า สัญญาณที่สร้างออกมา จะมี delay ที่เกิดจาก hardware ภายใน จะไม่เปลี่ยนแปลงทันที ดังนั้นหากเราสร้างสัญญาณ โดยใช้ concurrent statement เพื่อทำ operator ต่าง ๆ จำนวนมาก สัญญาณ output สุดท้าย จะสะสม delay ที่เกิดจาก hardware แต่ละตัวไว้ และเป็นการยากที่จะทราบว่า สุดท้ายแล้ว delay ทั้งหมดมีค่าเท่าไรกันแน่ และ delay เหล่านี้อาจจะเปลี่ยนไป เมื่อเราสังเคราะห์วงจรบน FPGA ใหม่อีกครั้ง

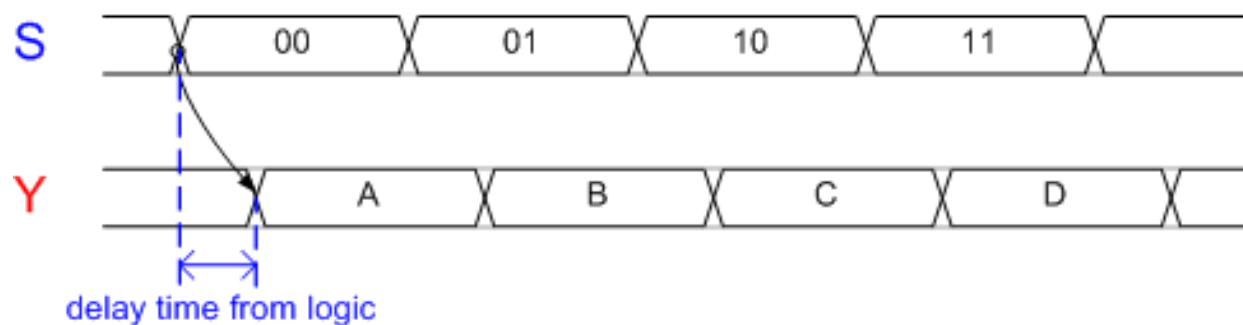
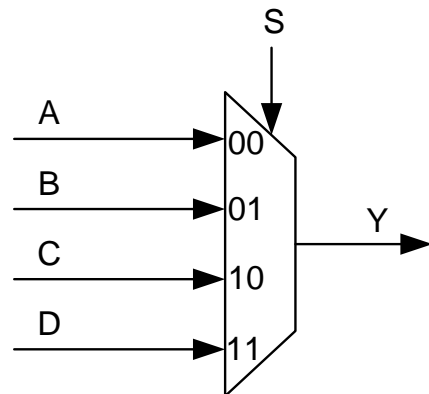
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

Entity NotLogic Is
    Port
    (
        A      : in    std_logic;
        Y      : out   std_logic
    );
End Entity NotLogic;

Architecture rtl Of NotLogic Is
Begin
    Y <= not (A);
End Architecture rtl;
```

<https://www.slideshare.net/ricardo.castro/experiment-writevhdlcodeforrealizealllogicgates>

## WHEN .. ELSE STATEMENT (MUX 4 TO 1)



ใช้ when ... else เมื่อต้องการสร้างสัญญาณแบบมีเงื่อนไข โดยไม่ต้องการทำงานภายใต้ clock ดังนั้น output จะสามารถเปลี่ยนแปลงได้ทันทีตามค่า S แต่จะมี delay time ที่เกิดจาก hardware ที่ใช้สร้าง logic นี้เอง

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

Entity Mux4t1Comb Is
    Port
    (
        A      : in    std_logic;
        B      : in    std_logic;
        C      : in    std_logic;
        D      : in    std_logic;
        S      : in    std_logic_vector(1 downto 0);
        Y      : out   std_logic
    );
End Entity Mux4t1Comb;

Architecture rtl Of Mux4t1Comb Is
Begin
    Y <= A when (S = "00") else
        B when (S = "01") else
        C when (S = "10") else
        D;
End Architecture rtl;
```

## WHY CLOCK?

**ปัญหา** เมื่อวงจรเป็น Concurrent statement ยากที่จะควบคุม delay time ของสัญญาณ output และไม่สามารถคาดเดาได้ว่า จะต้องรอ output เป็นเวลาเท่าไร ถึงจะได้ค่าที่ถูกต้อง

### แก้ไข

- ให้วงจรทำงานตามขอบขาขึ้น/ขาลงของ Clock และการสร้างสัญญาณแต่ละสัญญาณ จะต้องเสร็จสิ้นภายใน 1 คาบเวลาของ Clock หากเสร็จไม่ทัน ต้องแบ่งการทำงานการสร้างวงจรออกเป็นหลาย ๆ ขั้นตอน เพื่อลดทอนขนาดของวงจร และทำให้ delay ในการสร้างสัญญาณมีขนาดน้อยกว่า 1 คาบของ Clock
- สามารถทราบเวลาที่แน่นอนได้ว่า เราจะได้ output หลังจากทำงานไปถึง clock cycle ตามการออกแบบ
- ตัว tool ที่ใช้สังเคราะห์วงจรบน FPGA จะตรวจสอบให้ว่า สัญญาณที่สร้างนั้น จะมี delay น้อยกว่าคาบเวลาไหม ถ้าไม่ จะฟ้อง error ดังนั้น project ที่เราจะสร้าง ต้องระบุคาบเวลาของสัญญาณนาฬิกาที่เราจะใช้ด้วย เพื่อให้ tool ช่วยตรวจสอบว่า วงจรเราสามารถทำงานภายใต้ clock ความถี่นี้ได้ไหม
- การเขียน code ที่ทำงานภายใต้ clock จะเขียนอยู่ใต้ Process ซึ่งเป็น Sequential Statement

<https://www.slideshare.net/ricardo.castro/experiment-writevhdlcodeforrealizealllogicgates>

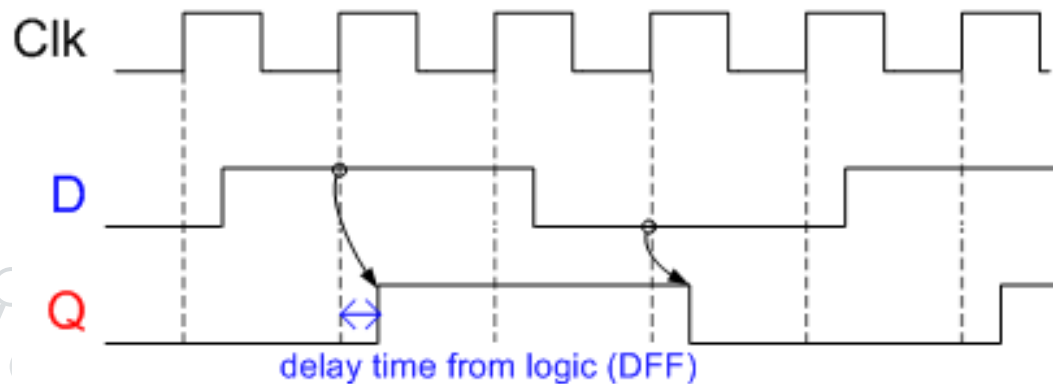
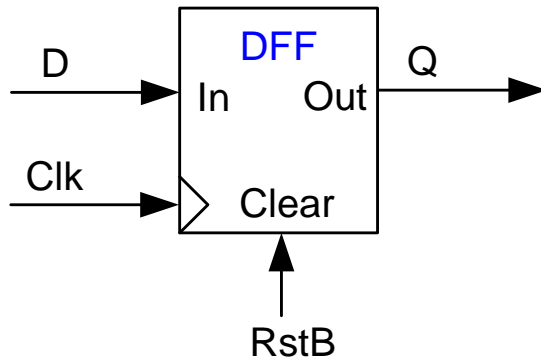
## 2. SEQUENTIAL STATEMENT

### 2) Sequential Statement

- ทำงานเป็นลำดับจากบรรทัดบนไปบรรทัดล่าง ภายใต้ Process
- ด้านหลัง Process จะเป็น sensitivity list ซึ่งใช้ระบุว่า Process จะทำงานเมื่อสัญญาณใน sensitivity list มีการเปลี่ยนแปลง ในตัวอย่างต่อ ๆ ไป จะแสดงเพียงสัญญาณ Clk เท่านั้น เพราะจะสร้างสัญญาณที่เปลี่ยนแปลงตามขอบของ Clock เท่านั้น หรือเป็นสัญญาณที่ผ่าน Flip Flop แล้วนั่นเอง

Note: หากจะสร้างสัญญาณที่ไม่ทำงานตาม Clock แนะนำให้ใช้การสร้างสัญญาณแบบ Concurrent Statement แทนการใช้ Process ซึ่งจะต้องระบุ Sensitivity list ให้ครบ

# PROCESS AND SENSITIVITY LIST



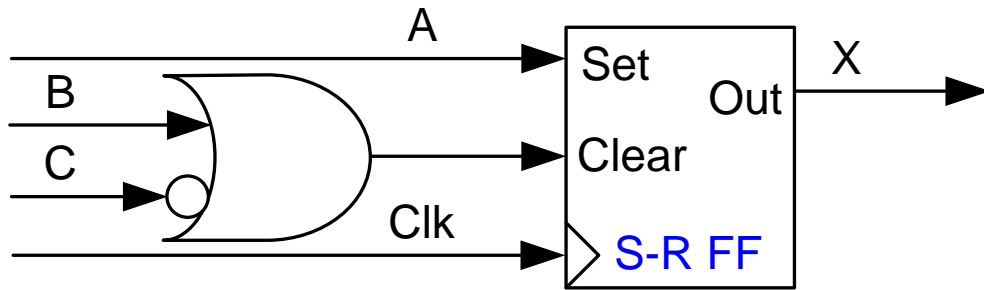
```
u_DFF : Process (Clk) Is
Begin
    if ( rising edge(Clk) ) then
        if ( RstB='0' ) then
            Q <= '0';
        else
            Q <= D;
        end if;
    end if;
End Process u_DFF;
```

Clk: Sensitivity list ดังนั้น Process จะทำงานเมื่อเจอขอบขาขึ้นของ Clk ก่อน

RstB: Synchronous reset (active low) จะทำงาน reset ก็ต่อเมื่อเจอขอบขาขึ้นของ Clk ก่อน

Q จะเปลี่ยนเมื่อเจอขอบขาขึ้นของ Clock และในความเป็นจริงสัญญาณ Q ที่สร้างออกมา จะมี delay time นิดหนึ่งเมื่อเทียบกับขอบของ Clock

# IF .. ELSE .. STATEMENT

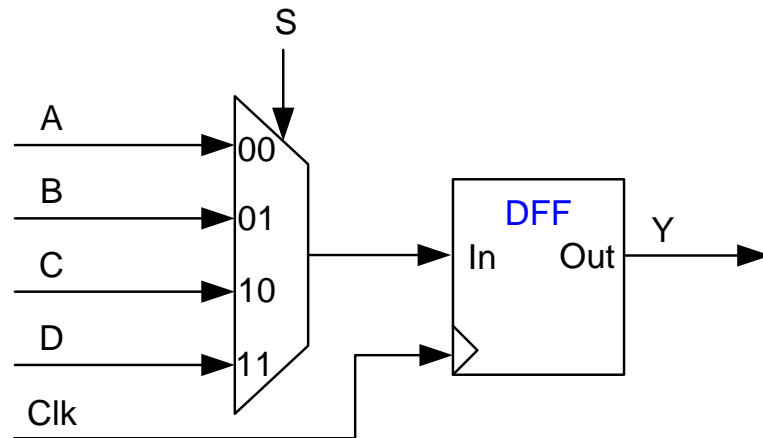


```
u_SRFF : Procss (Clk) Is
Begin
  if ( rising_edge(Clk) ) then
    if ( A='1' ) then
      X <= '1';
    elsif ( B='1' or C='0' ) then
      X <= '0';
    else
      X <= X;
    end if;
  end if;
End Process u_SRFF;
```

ใส่หรือไม่ใส่บรรทัดนี้  
ได้ logic เหมือนกัน

- If else statement จะทำงานเงื่อนไขแบบมี priority โดย if แรกจะมี priority สูงสุด และ elsif ถัด ๆ ไปจะมี priority รองลงมา ส่วน else สุดท้ายจะมี priority ต่ำสุด
- else ในคำสั่งสุดท้าย หากไม่ใส่ลงไปเลย จะแปลความได้ว่า ให้สัญญาณคงค่าเดิมไว้ (latch ค่าไว้) แนะนำให้ระบุเป็น code ลงไปว่า ต้องการเก็บค่าเดิม ดังแสดงในรูปด้านขวา เพื่อจะได้ชัดเจนว่าต้องการเก็บค่าเดิม ไม่ใช่ลืมเขียน code

## CASE STATEMENT (MUX 4 TO 1)



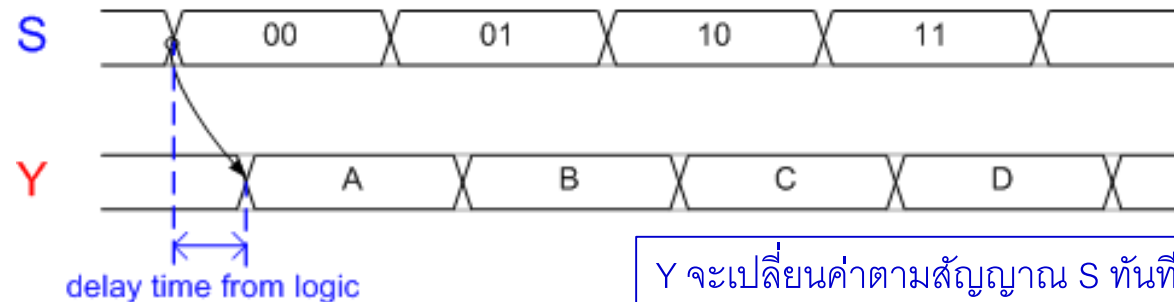
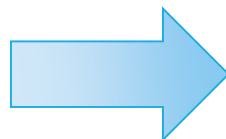
```
u_Mux4to1 : Procss (Clk) Is
Begin
    if ( rising_edge(Clk) ) then
        case (S) is
            when "00"    => Y    <= A;
            when "01"    => Y    <= B;
            when "10"    => Y    <= C;
            when others => Y    <= D;
        end case;
    end if;
End Process u_Mux4to1;
```

- Case statement จะต่างจาก if ตรงที่ไม่มี priority (มี priority เท่ากันหมด)
- เนื่องจากสัญญาณ std\_logic นั้นไม่ได้มีแค่ 1 กับ 0 แต่มี X, Z และค่าอื่นๆ ได้อีกด้วย เช่น ถ้า input มี 2 bit จะมีเงื่อนไขมากกว่า 4 แบบ (00 01 10 11) ดังนั้นบรรทัดสุดท้ายของ case จะตามด้วย others เสมอเพื่อให้ครอบคลุมเงื่อนไขอื่นๆ เช่น XX ZZ การออกแบบจริง เรามักจะรวมเงื่อนไขสุดท้าย เช่น 11 เป็น others เลย
- MUX นี้จะมี timing diagram ต่างจากการสร้างด้วย when ... else ตรงที่จะมี D Flip Flop ไล่ตามมาด้วยหลังจาก MUX 4 to 1 เพราะทำงานภายใต้ rising\_edge(Clk) ดังนั้น output ที่ได้ จะต้องรอขอบขาขึ้นของ clock ถัดไปก่อนถึงจะเปลี่ยนค่าได้

# CONCURRENT VS SEQUENTIAL

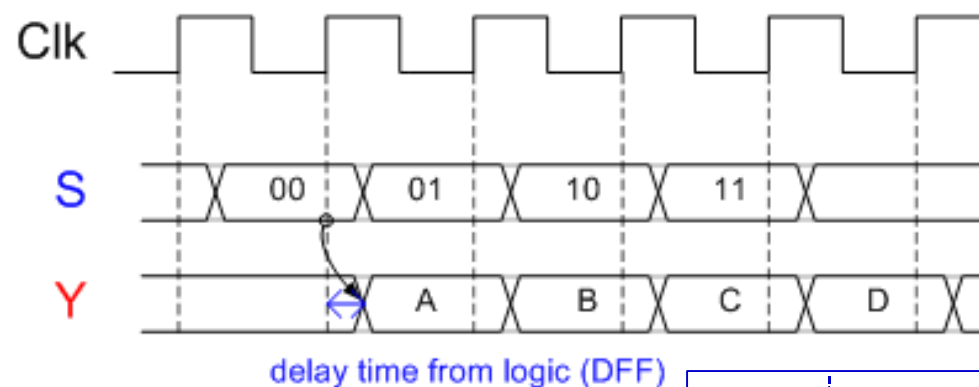
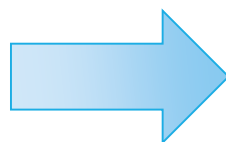
```

Architecture rtl Of Mux4t1Comb Is
Begin
    Y <= A when (S = "00") else
        B when (S = "01") else
        C when (S = "10") else
        D;
End Architecture rtl;
    
```



```

u_Mux4t1 : Procss (Clk) Is
Begin
    if ( rising_edge(Clk) ) then
        case (S) is
            when "00" => Y <= A;
            when "01" => Y <= B;
            when "10" => Y <= C;
            when others => Y <= D;
        end case;
    end if;
End Process u_Mux4t1;
    
```





# EXAMPLE DIGITAL LOGIC

# 4-BIT COUNTER

```
-- 4-bit increment counter
u_rCnt : Process (Clk) Is
Begin
    if ( rising_edge(Clk) ) then
        if ( RstB='0' ) then
            rCnt(3 downto 0)    <= (others=>'0');
        else
            -- Load start value
            if ( CntLd='1' ) then
                rCnt(3 downto 0)    <= DataIn(3 downto 0);
            -- Counter enable
            elsif ( CntEn='1' ) then
                rCnt(3 downto 0)    <= rCnt(3 downto 0) + 1;
            -- Hold same value
            else
                rCnt(3 downto 0)    <= rCnt(3 downto 0);
            end if;
        end if;
    end if;
End Process u_rCnt;
```

1

1. โหลดค่า start value มาจาก DataIn เมื่อสัญญาณ CntLd='1'

2

2. นับขึ้นเมื่อมีสัญญาณ CntEn='1'

3

3. คงค่าเดิมเมื่อไม่มีสัญญาณอะไรเข้ามา

# 4-BIT SERIAL TO PARALLEL

```
-- 4-bit Serial to Parallel
u_rSerData : Process (Clk) Is
Begin
    if ( rising_edge(Clk) ) then
        if ( RstB='0' ) then
            rSerData(3 downto 0)    <= (others=>'0');
            -- Shift register (Style 1)
        else
            rSerData(3) <= rSerData(2);
            rSerData(2) <= rSerData(1);
            rSerData(1) <= rSerData(0);
            rSerData(0) <= SerDataIn;
        end if;
    end if;
End Process u_rSerData;
```

สัญญาณ rSerData จะเป็นการรวมค่าของสัญญาณ SerDataIn ทั้งหมด 4 clock เข้าด้วยกัน ซึ่งตรงกับการทำงานของวงจรแปลงข้อมูลจาก Serial (1-bit) เป็นแบบ Parallel (4-bit) โดยในที่นี้ จะรับค่า SerDataIn มาทาง LSB แล้ว shift ซ้ายไปเรื่อย ๆ เพื่อให้ข้อมูลที่เข้ามาก่อนหน้านี้ จะเลื่อนไปทาง MSB

ในตัวอย่างนี้ แสดงการเขียน code สำหรับการ shift ซ้ายทั้งหมด 3 แบบ ซึ่งทุก ๆ แบบจะได้ผลลัพธ์เดียวกัน

```
-- Shift register (Style 2)
else
    rSerData(3 downto 1) <= rSerData(2 downto 0);
    rSerData(0) <= SerDataIn;
end if;
```

```
-- Shift register (Style 3)
else
    rSerData(3 downto 0) <= rSerData(2 downto 0) & SerDataIn;
end if;
```

# 4-BIT PARALLEL TO SERIAL

```
-- 4-bit Parallel to Serial
u_rParData : Process (Clk) Is
Begin
    if ( rising_edge(Clk) ) then
        if ( RstB='0' ) then
            rParData(3 downto 0)    <= (others=>'0');
        else
            -- Load 4-bit data input
            if ( ParLoad='1' ) then
                rParData(3 downto 0)    <= ParDataIn;
            -- Shift data out, starting from LSB to MSB (right shift)
            -- Fill LSB by 0
            else
                rParData(2 downto 0)    <= rParData(3 downto 1);
                rParData(3)             <= '0';
            end if;
        end if;
    end if;
End Process u_rParData;

DataOut    <= rParData(0);
```

สัญญาณ rParData รับค่า input ขนาด 4 bit มาจาก ParDataIn เมื่อสัญญาณ ParLoad='1' หลังจากนั้นจะทยอยส่งสัญญาณออกไปทีละ bit ตามจังหวะของ clock โดยเริ่มจาก LSB ก่อน

ดังนั้นการทำงานจะเป็นการ shift ขวา (การทำงานต่างจากข้อที่แล้ว) และให้สัญญาณขาออกมีค่าเท่ากับ rParData bit ที่ 0 เพื่อจะส่งข้อมูลเริ่มจาก LSB ไปจบที่ MSB

และเมื่อส่งเสร็จ ข้อมูลจะกลายเป็น '0' หมดเพราะเราเติมค่า '0' ลงไปแทนที่ค่าเดิมระหว่าง shift ข้อมูล

## Q & A

<https://www.facebook.com/DigitalDesignThailand/>



<https://forfpgadesign.wordpress.com/>

