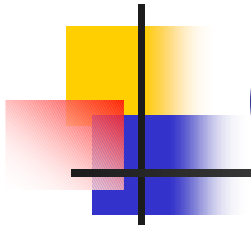
第２６回九州工業大学情報技術セミナー

SOC設計演習

VHDLによるRAIDシステムの設計

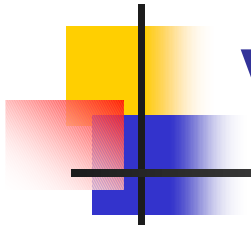
琉球大学工学部情報工学科
和田　知久　wada@ie.u-ryukyu.ac.jp
http://bw-www.ie.u-ryukyu.ac.jp/~wada/



(1章) VHDL関連ツールの復習

- VHDLというハードウエア記述言語を用いるデジタル設計の復習

1. ハーフアダーを例にVHDL言語の説明
2. ハーフアダーの動作をシミュレーション
3. ハーフアダーの回路を合成する
4. ALUの記述の紹介
5. ALU回路の合成
6. ALUの動作をシミュレーション



VHDLの歴史

- 1970年代に米国国防省にてVery High Speed ICプロジェクト発足
- 1981年にVHSICプロジェクトの一環として機能記述言語VHDLが提案
- 1986年にIEEEにてVHDLバージョン7.2を基本言語として採択し、標準化スタート
- 1987年にLanguage Reference Manualをリリースして、IEEE-1076Bとして正式に標準化

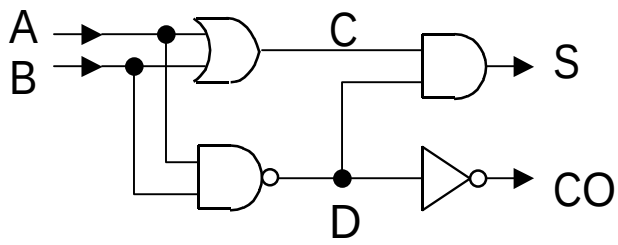
VHDLの適用分野

設計レベル（例：パソコン）	VHDLの記述レベル
マザーボードと周辺機器	アーキテクチャレベル
マザーボード	ビヘビアレベル
Integrated Circuit	RTL（レジスタトランスファレベル）
論理ゲート	ストラクチャレベル

- 本実習では論理合成プログラムによる回路生成が可能なRTLの記述を学ぶ。
- RTLの記述で詳細なデジタル回路の記述ができる。

VHDLの簡単な例題

```
library IEEE;
use IEEE.std_logic_1164.all;
entity HALF_ADDER is
   port ( A, B : in std_logic;
          S, CO : out std_logic);
end HALF_ADDER;
architecture DATAFLOW of HALF_ADDER is
   signal C, D : std_logic;
   begin
   C <= A or B;
   D <= A nand B;
   CO <= not D;
   S <= C and D;
end DATAFLOW;
```



VHDLを簡単に見てゆくと

library IEEE; use IEEE.std_logic_1164.all;	ライブラリ宣言、通常STD_LOGICを使用するので、必ず書く
entity HALF_ADDER is	エンティティ宣言
port (A, B : IN std_logic; S, CO : OUT std_logic);	ポート宣言（入出力信号と信号のデータタイプを定義）
end HALF_ADDER;	エンティティ終了。";"忘れるな！
architecture DATA_FLOW of HALF ADDER is	アーキテクチャ宣言
signal C, D : std_logic;	内部信号を定義
begin C <= A or B ; D <= A nand B ; CO <= not D; S <= C and D; end DATAFLOW;	同時処理文！ コンピュータプログラムとは違う、この場合は4つの同時動作するゲートに対応

std_logicとは

- デジタルの信号を示すデータタイプで、通常‘0’と‘1’を示すが、それ以外に‘X’など、9種類の状態を示す

'U'	Uninitialized	'W'	Weak Unknown
'X'	Unknown	'L'	Weal Low
'0'	Low	'H'	Weak High
'1'	High	'-'	Don't care
'Z'	Hi-impedance		

実習1：ハーフアダーのシミュレーション (1)

- **ツールのセットアップ**

1. PCにloginする。
2. そこから nirai にrloginする。

```
% rlogin nirai
```

3. テキストエディターを使って、自分のホーム・ディレクトリにある .cshrc ファイルの最後の行に以下の1行を付け加える。

```
source     /usr/synopsys/setup_2000.05/init/synopsys_setup_2000.05
```

4. 変更した設定を有効にするために、niraiにrloginし直す。

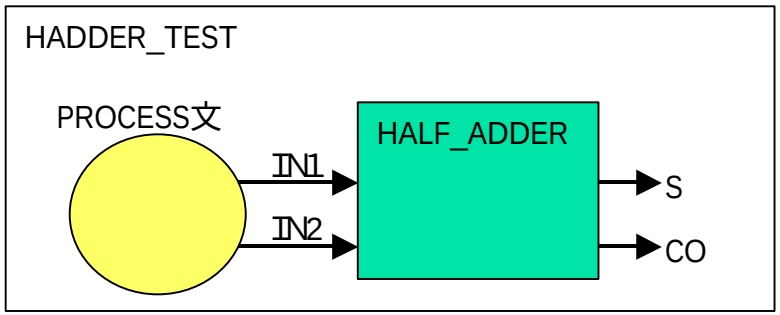
実習1：ハーフアダーのシミュレーション (2)

- ディスプレイの設定とファイルのコピー

1. PCのローカルなウインドウで、以下のコマンドを実行する
   % xhost nirai
   % xhost nirai-1
2. nirai のウインドウで、以下のコマンドを実行する
   *hostname*はPW01のような自分の使用中のマシン名
   % setenv DISPLAY *hostname*:0.0
3. nirai のウインドウで、作業ディレクトリ (たとえば、vhdl) を作成し、必要なファイルをコピーする
   % mkdir　vhdl
   % cd　vhdl
   % cp　/home/teacher/wada/pub/*.*　.

hadder_test.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

entity HADDER_TEST  is
end HADDER_TEST;

architecture TESTBENCH of HADDER_TEST  is
component HALF_ADDER
      port ( A, B         : in  std_logic;
             S, CO        : out std_logic);
end component;
signal  IN1, IN2,  S, CO  :  std_logic
```




```
begin
   U0: HALF_ADDER port map (IN1, IN2,  S, CO);
   process begin
      IN1<='0' ; IN2<='0' ;
      wait for 10 ns;
      IN1<='0' ; IN2<='1' ;
      wait for 10 ns;
      IN1<='1' ; IN2<='0' ;
      wait for 10 ns;
      IN1<='1' ; IN2<='1' ;
      wait for 10 ns;
      wait;
   end process;
end TESTBENCH;

configuration  CFG_ADDER  of HADDER_TEST is
     for TESTBENCH
     end for;
end CFG_ADDER;
```

実習1：ハーフアダーのシミュレーション (3)

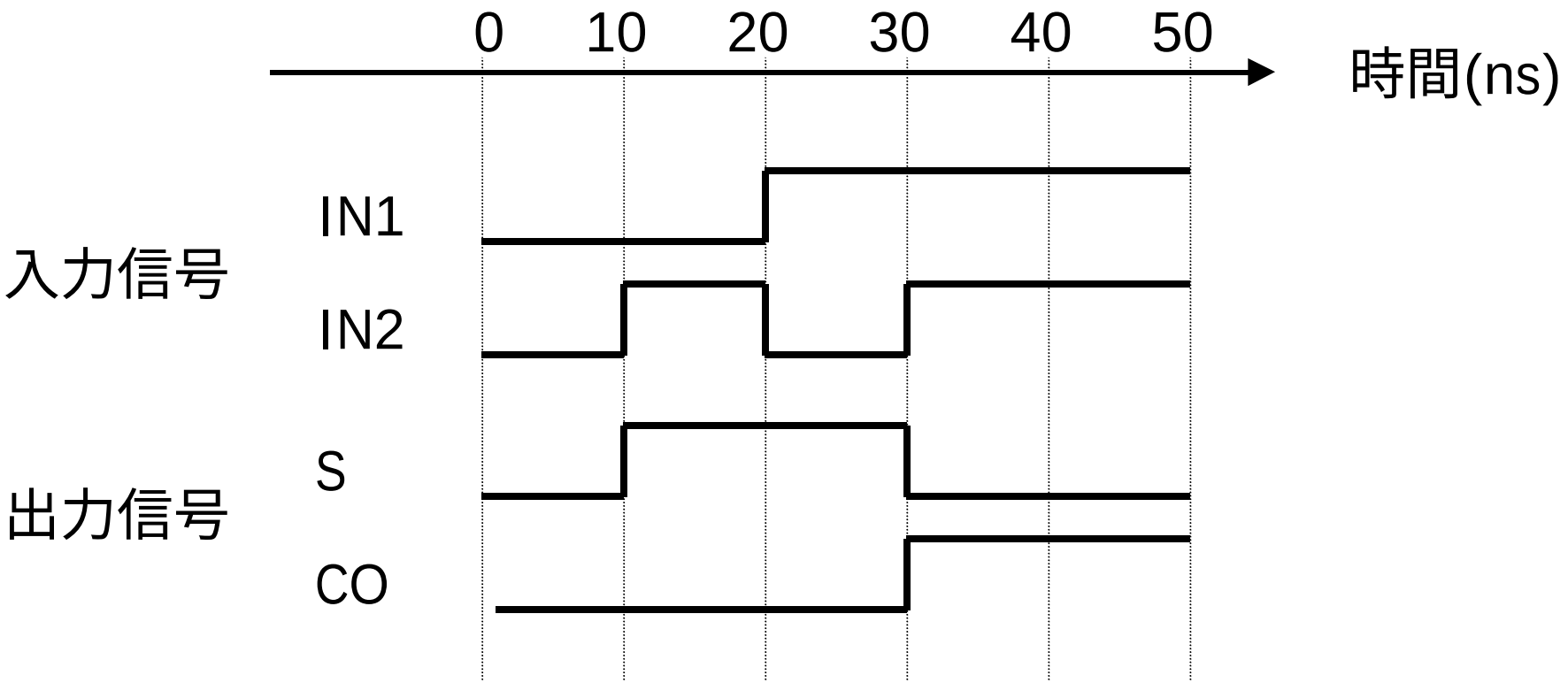
- VHDLファイルの構文解析

1. 2つのVHDL記述の内容を確かめる
   half_adder.vhd　：ハーフアダーの記述
   hadder_test.vhd　：ハーフアダーをテストするための記述：テストベンチ
2. 構文解析する　(VHDLアナライザ)
   % vhdlan　half_adder.vhd
   % vhdlan　hadder_test.vhd

- VHDL Debugger で動作を確認する

1. VHDL Debuggerを起動する
   % vhdldbx　CFG_ADDER &
2. 観測したい信号を指定する
   （VHDL Debugger ウインドウの最下位のＢOXで　　trace *’ signal
3. シミュレーションの時間を進める
   （VHDL Debugger ウインドウの最下位のＢOXで　　run　50
4. 次ページのような波形になればＯＫ！

ハーフアダーの動作波形



実習2: ハーフアダーの回路合成

- design_analyzerの起動とファイルの読み込み

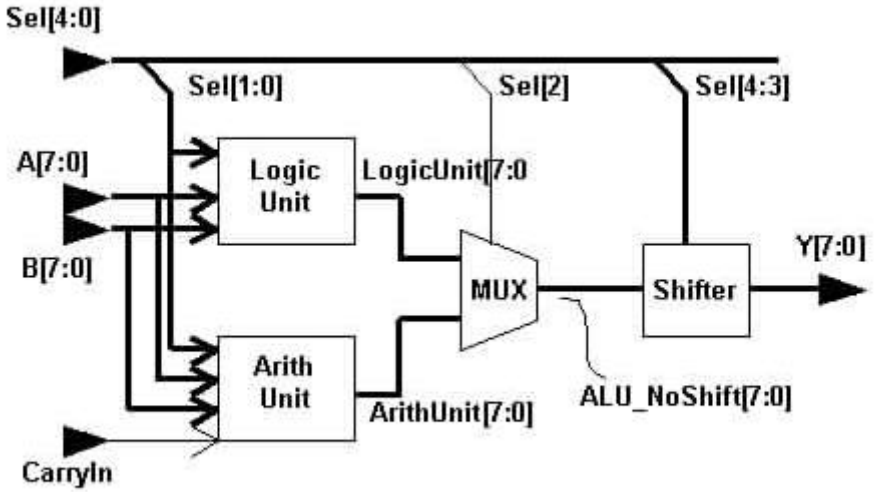
1. デザインアナライザを起動する
   % design_analyzer &
2. メニューからFile->Read…を選択し、
   half_adder.vhd を読み込む
3. 画面中央のHALF_ADDERシンボルをダブルクリックし、1階層下のレベルへ入ると、入出力ピンがわかる
4. さらに中央の箱をダブルクリックし、1階層下のレベルへ入ると、記述どおりのハーフアダー回路が確認できる

- 回路の最適化

1. メニューからTool->DesignOptimizationを選び、OKをクリックする
2. 回路が最適化されたことを確認する

ALUのVHDL記述の紹介

- ALUとは算術（加算や減算）や論理演算を行う回路である
- 今回2つの8ビットの数AとBの入力に対してYを出力する、１４種類の演算を行うＡＬＵの記述を紹介する



ALUの動作

S4	S3	S2	S1	S0	Cin	動作	説明	実行ブロック
0	0	0	0	0	0	Y <= A	Aを転送	Arithmetic Unit
0	0	0	0	0	1	Y <= A+1	Aをインクリメント	Arithmetic Unit
0	0	0	0	1	0	Y <= A + B	加算	Arithmetic Unit
0	0	0	0	1	1	Y <= A + B + 1	キャリー付加算	Arithmetic Unit
0	0	0	1	0	0	Y <= A + Bbar	AとBの１の補数をたす	Arithmetic Unit
0	0	0	1	0	1	Y <= A + Bbar + 1	減算	Arithmetic Unit
0	0	0	1	1	0	Y <= A - 1	デクリメント	Arithmetic Unit
0	0	0	1	1	1	Y <= A	Aを転送	Arithmetic Unit
0	0	1	0	0	0	Y <= A and B	論理積	Logic Unit
0	0	1	0	1	0	Y <= A or B	論理和	Logic Unit
0	0	1	1	0	0	Y <= A xor B	排他的論理和	Logic Unit
0	0	1	1	1	0	Y <= Abar	１の補数	Logic Unit
0	0	0	0	0	0	Y <= A	Aを転送	Shifter Unit
0	1	0	0	0	0	Y <= shl A	Aを左シフト	Shifter Unit
1	0	0	0	0	0	Y <= shr A	Aを右シフト	Shifter Unit
1	1	0	0	0	0	Y <= 0	0を転送	Shifter Unit

alu.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;

entity ALU is
   port(Sel    :  in  unsigned(4 downto 0);
                  :  in  std_logic
                                 downto 0);
                                 downto 0) );



begin

   ALU_AND_SHIFT:
   process (Sel, A, B, CarryIn)
        variable Sel0_1_CarryIn  : unsigned(2            0);
        variable LogicUnit  ArithUnit,
                    ALU_NoShift     : unsigned(7 downto 0);
   begin
        --------------------------
        --

        LOGIC_UNIT: case Sel(1 downto 0) is
           when "00"    =>              := A and B;
                             LogicUnit := A or B;
           when "10"    => LogicUnit       xor B;
                             LogicUnit := not A;
           when others  => LogicUnit := (others => 'X');
      end case LOGIC_UNIT;
```

```
        --------------------------
        --

        Sel0_1_CarryIN := Sel(1 downto 0) & CarryIN;
        ARITH_UNIT: case Sel0_1_CarryIn is
           when "000"   => ArithUnit
                           ArithUnit := A+1;
           when "010"   => ArithUnit := A+B;
           when "011"   =>              := A+B+1;
                           ArithUnit := A + not B;
           when "101"   => ArithUnit := A-B;
           when "110"   =>              := A-
           when "111"   => ArithUnit := A;
           when others  =>              := (others => 'X');



           Mutiplex

        LA_MUX: if (Sel
           ALU_NoShift := LogicUnit;
      else
           ALU_NoSHift :=             ;


        -- Shift operation
        --------------------------
                      Sel(4            3) is
           when "00"    =>  Y <= ALU_NoShift;
           when "01"    =>  Y <=             (ALU_NoShift
           when "10"    =>  Y <= Shift_right  ALU_NoShift, 1);

           when others  =>  Y <= (others => 'X');
      end case SHIFT;
    end process ALU_AND_SHIFT;
  end architecture COND_DATA_FLOW;
```

実習: ALUの回路合成

design_analyzer

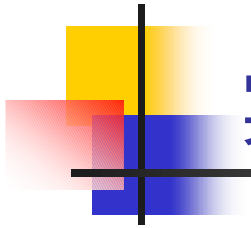
1. デザインアナライザを起動する
   %　　&
2. メニューからFile->Read　を選択し、
   alu.vhd　を読み込む
3. 画面中央のALUシンボルをダブルクリックし、1階層下のレベルへ入る
4. さらに中央の箱をダブルクリックし、1階層下のレベルへ入ると、記述回路が確認できる

- 回路の最適化

1. メニューからTool->DesignOptimizationを選び、OKをクリックする
2. 回路が生成されたことを確認する

test_alu.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;

entity TEST_ALU is
end TEST_ALU;

architecture TESTBENCH of TEST_ALU is
component ALU
   port ( Sel     : in   unsigned(4 downto 0);
          CarryIn  : in   std_logic;
          A, B     : in   unsigned(7 downto 0);
          Y       : out  unsigned(7 downto 0));
end component;
signal   SIN         : unsigned(4 downto 0);
signal   CIN         : std_logic;
signal   AIN,BIN,YIN   : unsigned(7 downto 0);
begin
        U0: ALU port map (SIN,CIN,AIN,BIN,YIN);
        process begin
         AIN <= "00001111";  BIN <= "11110000";
           wait for 10 ns;
           SIN <= "00000"; CIN <= '0';
           wait for 10 ns;
           SIN <= "00000"; CIN <= '1';
           wait for 10 ns;
           SIN <= "00001"; CIN <= '0';
           wait for 10 ns;
           SIN <= "00001"; CIN <= '1';
           wait for 10 ns;
           SIN <= "00010"; CIN <= '0';
           wait for 10 ns;
           SIN <= "00010"; CIN <= '1';
           wait for 10 ns;
           SIN <= "00011"; CIN <= '0';
           wait for 10 ns;
           SIN <= "00011"; CIN <= '1';
           wait for 10 ns;
           SIN <= "00100"; CIN <= '0';
           wait for 10 ns;
           SIN <= "00101"; CIN <= '0';
           wait for 10 ns;
           SIN <= "00110"; CIN <= '0';
           wait for 10 ns;
           SIN <= "00111"; CIN <= '0';
           wait for 10 ns;
           SIN <= "01000"; CIN <= '0';
           wait for 10 ns;
           SIN <= "10000"; CIN <= '0';
           wait for 10 ns;
           SIN <= "11000"; CIN <= '0';
           wait for 10 ns;
           wait;

        end process;
end TESTBENCH;

configuration CFG_ALU of TEST_ALU is
   for TESTBENCH
   end for;
end CFG_ALU;
```



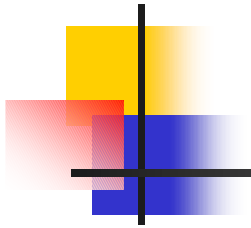
実習4：ALUのシミュレーション

- VHDLファイルの構文解析

1. ２つのVHDL記述の内容を確かめる
   alu.vhd　：ALUの記述
   test_alu.vhd　：ALUをテストするための記述：テストベンチ
2. 構文解析する　(VHDLアナライザ)
   % vhdlan alu.vhd
   % vhdlan test_alu.vhd

- VHDL Debugger で動作を確認する

1. VHDL Debuggerを起動する
   % vhdldbx　CFG_ALU &
2. 観測したい信号を指定する
   （VHDL Debugger ウインドウの最下位のＢOXで　　trace *’signal
3. シミュレーションの時間を進める
   RUNボタンの右横のＢOXに、２０　と記入し、RUNボタンを数回押す

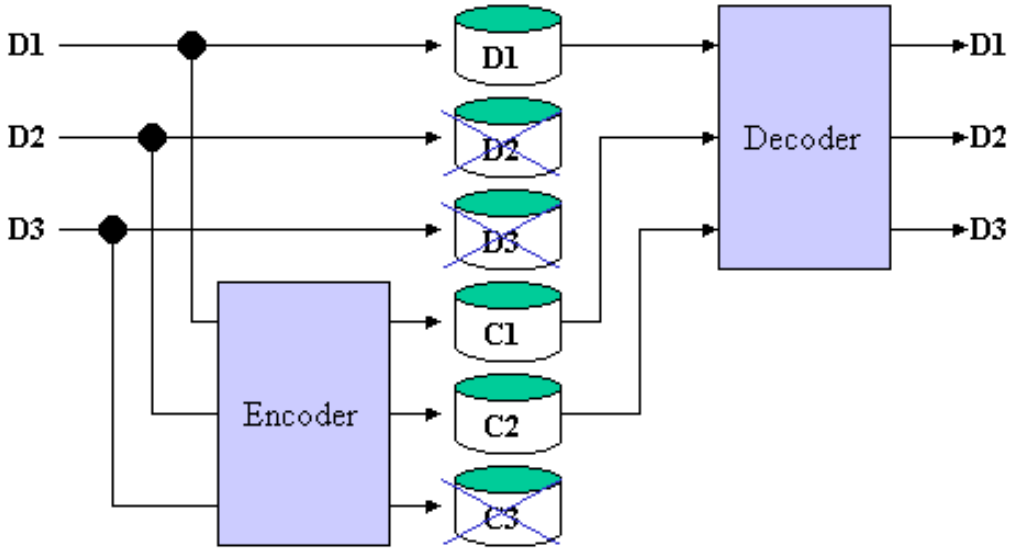


(2章) RAIDシステムとガロア体

- RAIDシステムとは
- Encoder / Decoder
- ガロア体の説明
- ガロア演算の応用例
- 具体的なガロア体での乗算の例

RAIDシステム

- Redundant Array of Inexpensive Disks
  - データのHDD保存時に、通常データのD1/D2/D3以外にチェックサムC1/C2/C3に冗長データを保存することで、HDD故障時に元のデータを復元する。
- 今回は簡単なRAIDシステムに必要なEncoder/Decoderを設計する。



Encoder / Decoderの計算式

- 式 (1) がチックサムの生成式
- 式 (2) がD 2D3/C3故障時のデータ再生式
-
  - ここでは4ビットのガロア体を取り扱います。

$$\begin{bmatrix} c1 \\ c2 \\ c3 \end{bmatrix} = \begin{bmatrix} 1^0 & 2^0 & 3^0 \\ 1^1 & 2^1 & 3^1 \\ 1^2 & 2^2 & 3^2 \end{bmatrix} \begin{bmatrix} d1 \\ d2 \\ d3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 4 & 5 \end{bmatrix} \begin{bmatrix} d1 \\ d2 \\ d3 \end{bmatrix} \quad \cdots \quad (1)$$

$$\begin{bmatrix} d1 \\ d2 \\ d3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 3 & 1 \\ 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} d1 \\ c1 \\ c2 \end{bmatrix} \quad \cdots \quad (2)$$

デジタル設計者へのガロア体の説明

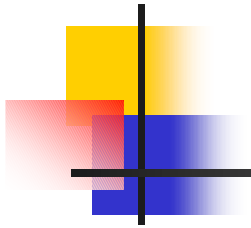
- ‘0’と‘1’だけが基本

  実際は4ビット、8ビットの塊で数値や文字を表すので、その塊を１文字で表す方が便利

- 今回取り扱う4ビットのガロア体は‘0’、‘1’という2種類に数字を以下の１６種類の数字に拡張したものである

１ビットのガロア体シンボル：　0、１

４ビットのガロア体シンボル：　0、１、α^1　、α^2、…、α



ガロア体演算はエラー訂正符号（リードソロモン符号）計算での一般的手法であり、以下で応用されている。

- コンパクトディスクのエラー訂正
- デジタルTV放送でのMPEGストリームのエラー訂正
- 地上波デジタルTV放送

- 1ビットのガロア体：GF(2)

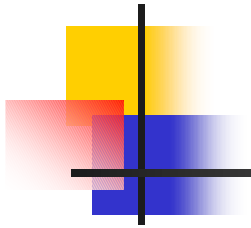
ガロア体のシンボル	対応する整数	対応する整数

整数とガロア体シンボルの対応 (2)
4ビットのガロア体：GF(2^4)

ガロア体シンボル	対応する2進数	対応する整数
0	0000	0
1	0001	1
α^1	0010	2
α^2	0100	4
α^3	1000	8
α^4	0011	3
α^5	0110	6
α^6	1100	12
α^7	1011	11
α^8	0101	5
α^9	1010	10
α^{10}	0111	7
α^{11}	1110	14
α^{12}	1111	15
α^{13}	1101	13
α^{14}	1001	9

ガロア体での乗算の例

- 3 x 3 をガロア体GF(2^4)上でやると
  - 整数の3は　＝＞　ガロア体シンボルで α^4
  - 3 X 3 => α^4 X α^4 = α^8
  - ガロア体シンボルで α^8は => 整数の5
- 3 x 3 = 5　となる
- 0011 X 0011 => 0101　となる乗算器が必要



(3章) ガロア体GF(2^4)乗算器の設計

- 今回２種類のアルゴリズムで乗算器をVHDLで設計します！

1. gflog / gfilog 関数を用いる方法 (一般的)
2. 乗算ブール式から直接設計する方法

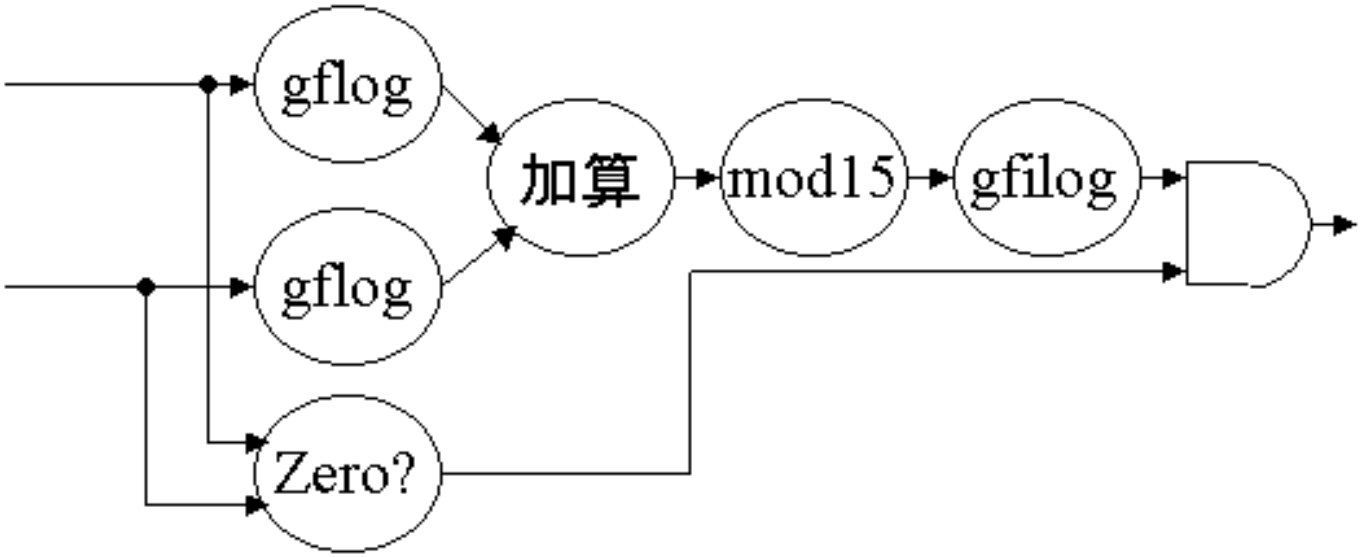
1) / gfilog 関数を用いる方法

- 再度、 算例 示す。
- 8x10 > α α^{9} => α^{12} > 15
  1000X1010 => 0011+1001 => 1100 =>1111
- 指数がオーバフローする場合
  α^{15}=1 を使用する。
  x9 => α^{5}X α^{14} => α^{15}• α^{4} > ^{4} =>3

整数/指数変換器、加算器、MOD15器、指数/整数変換器で実現できる

乗算器のアーキテクチャ

- 以下のような回路構成で乗算器を実現できる
-



整数/指数変換機gflogの設計

入力：4ビット	出力：4ビット
0000	XXXX (DON T CARE)
0001	0000
0010	0001
0011	0100
0100	0010
0101	1000
0110	0101
0111	1010
1000	0011
1001	1110
1010	1001
1011	0111
1100	0110
1101	1101
1110	1011
1111	1100

gflog.vhdの例

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;
-- 2^4Galois Field Log conversion
-- assuming gflog(0) = 15;

                                    downto 0);
        y    : out unsigned(3 downto 0));
end entity GFLOG;
```

```
architecture COMBof GFLOG is
begin
    process(x)

                                    end case;
```

(実習) gflog.vhdのＶＳＳシミュ＆回路合成

- VHDL Debugger で動作を確認してください！

  gflog.vhd
  test_gflog.vhd

- design_analyzerを用いて回路を合成してください！

(実習) 指数/整数変換機gfilogの設計

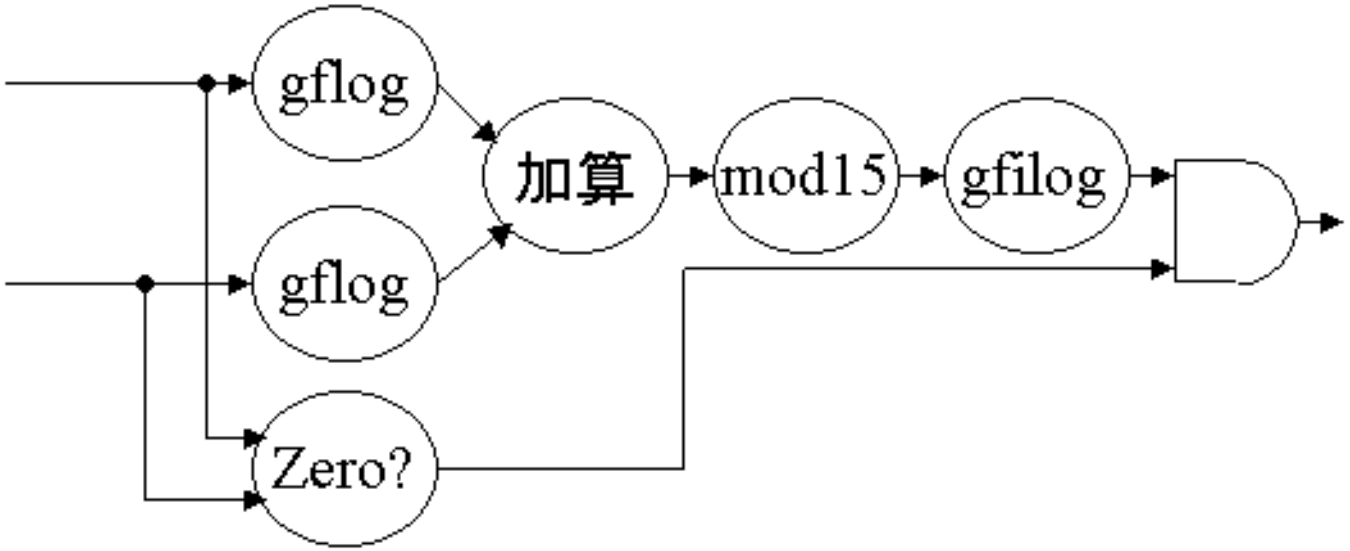
- 以下の仕様にしたがって、gfilogを設計せよ！

入力：4ビット	出力：4ビット
0000	0001
0001	0010
0010	0100
0011	1000
0100	0011
0101	0110
0110	1100
0111	1011
1000	0101
1001	1010
1010	0111
1011	1110
1100	1111
1101	1101
1110	1001
1111	XXXX (DON'T CARE)

1. エンティティ名：GFILOG
2. ファイル名：
   gfilog.vhd
   test_gfilog.vhd

(実習) ガロア乗算器gfmultの設計(1)

- 以下のアーキテクチャにしたがって、乗算器をVHDLで設計してください。



乗算器VHDLコード例gfmult.vhd (見ないでやってほしいですが)

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;
-- 2^4 Galoi Fileld Multiplier
-- Tom Wada

entity GFMULT is
    port ( ina      : in  unsigned(3 downto 0);
           inb      : in  unsigned(3 downto 0);
           product  : out unsigned(3 downto 0));
end entity GFMULT;

architecture RTL of GFMULT is
component GFLOG
    port ( x      : in  unsigned(3 downto 0);
           y      : out unsigned(3 downto 0));
end component;
component GFILOG
    port ( x      : in  unsigned(3 downto 0);
           y      : out unsigned(3 downto 0));
end component;

signal   logina,loginb   : unsigned(3 downto 0);
signal   sum             : unsigned(4 downto 0);
signal   mod15           : unsigned(4 downto 0);
signal  ilogmod          : unsigned(3 downto 0);

begin

    U0: GFLOG port map (ina, logina);
    U1: GFLOG port map (inb, loginb);

    sum <= ('0' & logina) + ('0' & loginb);

    MOD_CAL: process(sum) begin
        if (sum >= 15) then
            mod15 <= sum - 15;
        else
            mod15 <= sum;
        end if;
    end process MOD_CAL;

    U2: GFILOG port map (mod15(3 downto 0), ilogmod);

    MUX: process(ina, inb, ilogmod) begin
        if ( ina = 0 or inb = 0) then
            product <= "0000";
        else
            product <= ilogmod;
        end if;
    end process MUX;

end architecture RTL;
```

(実習) ガロア乗算器 gfmult の設計 (2)

- 作成したgfmult.vhd, gflog.vhd, gfilog.vhdをアナライズして

上記３

1. READウインドウでSHIFTしながら、３つのファイルが選べる
2. GFMULTを選択して回路合成→GFLOGを2回使用しているのでエラー
3. GFMULT選択しEdit>Uniquify>Hierarchyでコピーを作成
4. GFMULTを選択して回路合成→GFMULTの階層化をチェック
5. 階層展開するブロックを選択し、Attributes>Coptimization Directives>Cellにて、dgroup Cell s Hierarchyをチェック
6. GFMULTを選択して再回路合成
7. ReportでAreaをチェックして面積と速度は？

２乗算テーブルから直接設計する方法

-
- 4ビット $A=a_3 a_2 a_1 a_0$は以下の多項式である

$$a_3 \cdot \alpha^3 + a_2 \cdot \alpha^2 + a_1 \cdot \alpha^1 + a_0$$

係数同士の乗算はAND、加算はXORとする

- 乗算等の処理で4次以上の項の次数を下げるために、以下の特性方程式を使用する。

$$\alpha^4 = \alpha + 1$$

- 4ビット A=a_3,a_2,a_1,a_0 とB=b_3,b_2,b_1,b_0の加 算

$$(a_3\cdot a^3+a_2\cdot a^2+a_1\cdot a^1+a_0)+(b_3\cdot a^3+b_2\cdot a^2+b_1\cdot a^1+b_0)$$
$$=(a_3\oplus b_3)a^3+(a_2\oplus b_2)a^2+(a_1\oplus b_1)a^1+(a_0\oplus b_0)$$

- となる

$$11+7=1011\oplus 0111=1100=12$$

4ビットの乗算は実は…

$a^4 = a + 1$、$a^5 = a^2 + a$、$a^6 = a^3 + a^2$であるので

$$(a_3 \cdot a^3 + a_2 \cdot a^2 + a_1 \cdot a^1 + a_0) \cdot (b_3 \cdot a^3 + b_2 \cdot a^2 + b_1 \cdot a^1 + b_0)$$

$$= (a_3 \cdot b_3)a^6 + (a_3 \cdot b_2 \oplus a_2 \cdot b_3)a^5 + (a_3 \cdot b_1 \oplus a_2 \cdot b_2 \oplus a_1 \cdot b_3)a^4$$

$$+ (a_3 \cdot b_0 \oplus a_2 \cdot b_1 \oplus a_1 \cdot b_2 \oplus a_0 \cdot b_3)a^3 + (a_2 \cdot b_0 \oplus a_1 \cdot b_1 \oplus a_0 \cdot b_2)a^2$$

$$+ (a_1 \cdot b_0 \oplus a_0 \cdot b_1)a + a_0 \cdot b_0$$

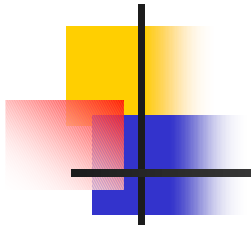
$$= (a_3 \cdot b_0 \oplus a_2 \cdot b_1 \oplus a_1 \cdot b_2 \oplus a_0 \cdot b_3 \oplus a_3 \cdot b_3)a^3$$

$$+ (a_2 \cdot b_0 \oplus a_1 \cdot b_1 \oplus a_0 \cdot b_2 \oplus a_3 \cdot b_3 \oplus a_3 \cdot b_2 \oplus a_2 \cdot b_3)a^2$$

$$+ (a_1 \cdot b_0 \oplus a_0 \cdot b_1 \oplus a_3 \cdot b_2 \oplus a_2 \cdot b_3 \oplus a_3 \cdot b_1 \oplus a_2 \cdot b_2 \oplus a_1 \cdot b_3)a$$

$$+ a_0 \cdot b_0 \oplus a_3 \cdot b_1 \oplus a_2 \cdot b_2 \oplus a_1 \cdot b_3$$

- ANDゲート16個とXORゲート１８個で構成可能



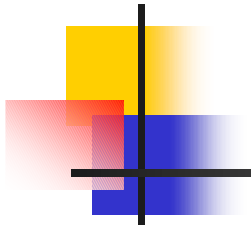
(実習) ガロア乗算器gfmult2の設計

- 前ページの仕様にしたがって、gfmult2を設計せよ！
- gfmult2.vhd(entity GFMULT2)の動作をＶＳＳで確認せよ！
- デザインアナライザーで合成し、面積と速度を以前の設計と比べよ！

乗算器VHDLコード例gfmult2.vhd (見ないでやってほしいですが)

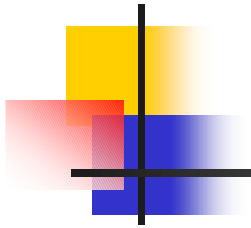
```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;
-- 2^4 Galoi Fileld Multiplier easy version
-- Tom Wada

entity GFMULT2 is
    port ( ina      : in  unsigned(3 downto 0);
           inb      : in  unsigned(3 downto 0);
           product  : out unsigned(3 downto 0));
end entity GFMULT2;
```

```
architecture RTL of GFMULT2 is
begin
    product(3)<= (ina(3) and inb(0)) xor
                 (ina(2) and inb(1)) xor
                 (ina(1) and inb(2)) xor
                 (ina(0) and inb(3)) xor
                 (ina(3) and inb(3));
    product(2)<= (ina(2) and inb(0)) xor
                 (ina(1) and inb(1)) xor
                 (ina(0) and inb(2)) xor
                 (ina(3) and inb(3)) xor
                 (ina(3) and inb(2)) xor
                 (ina(2) and inb(3));
    product(1)<= (ina(1) and inb(0)) xor
                 (ina(0) and inb(1)) xor
                 (ina(3) and inb(2)) xor
                 (ina(2) and inb(3)) xor
                 (ina(3) and inb(1)) xor
                 (ina(2) and inb(2)) xor
                 (ina(1) and inb(3));
    product(0)<= (ina(0) and inb(0)) xor
                 (ina(3) and inb(1)) xor
                 (ina(2) and inb(2)) xor
                 (ina(1) and inb(3));
end architecture RTL;
```



(4章) レジスタファイルの設計

- 和田担当分では初めての順序回路である
  レジスタファイルの設計をVHDLで行います

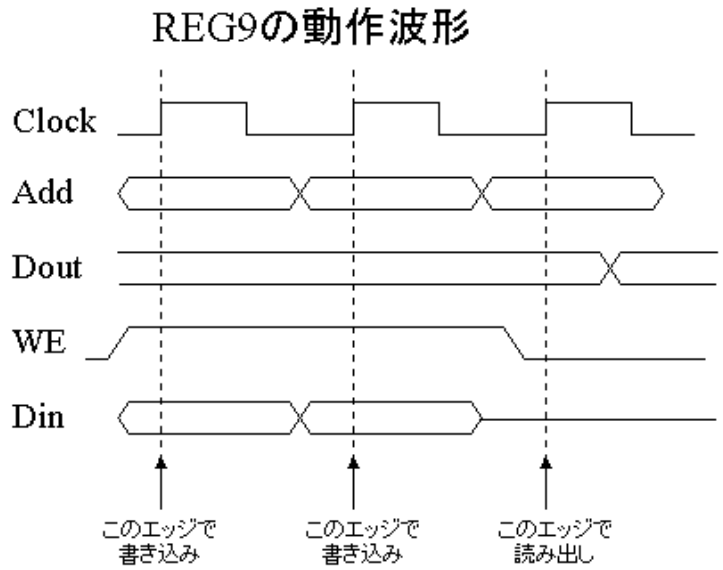


レジスタファイルとは

- 多数のフリップフロップの集まりで、
  あるビット幅のデータをアドレスを指定した場所のフリップフロップに書き込み・読み出しを行うことができる
- 今回は後に使うRAIDシステムに合うように、
  以下の仕様とする
  - アドレス番地0から8
  - 各番地のデータ幅は4ビット
  - 同期動作

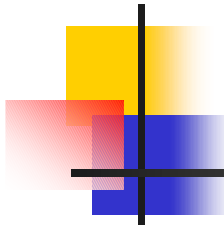
レジスタファイルの動作タイミング

- Entity名：REG9
- Clockの立ち上がりエッジでWEが‘0’であれば、その時与えられたアドレスに対応して、Doutにデータが読み出されるようにする。
  図で示すと以下のようになる。



REG9のコード例

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;

entity REG9 is
    port ( ADD    : in  unsigned(3 downto 0);
           DIN    : in  unsigned(3 downto 0);
           WE     : in  std_logic;
           CLK    : in  std_logic;
           DOUT   : out unsigned(3 downto 0));
end entity REG9;

architecture RTL of REG9 is
    ---- 4bit x 9 register type ----
    type RegType is array (0 to 8) of unsigned(3 downto 0);
    ---- 4bit x 9 register declaration ----
    signal elereg  : RegType;
begin
    process (CLK)
    begin
        if (rising_edge(CLK)) then
            if (WE = '1') then elereg ( TO_INTEGER(ADD) ) <= DIN;
             else
             DOUT <= elereg( TO_INTEGER(ADD) );
             end if;
         end if;
    end process ;

end architecture RTL;
```



(実習) REG9のVSSシミュ&回路合成

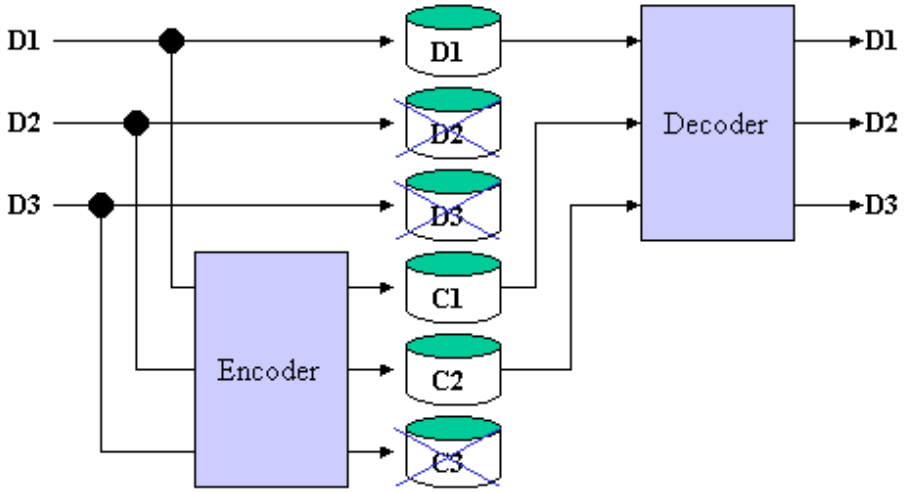
- 以下のファイルをアナライズし、VHDL Debugger で動作を確認してください！

  reg9.vhd
  test_reg9.vhd

- design_analyzerを用いて回路を合成してください！
- クロックのある回路ではクロック信号とその周期を指定する必要があります。
  - クロック入力端子をダブルクリックで選択し、
  - Attributes -> Clock -> Specify
  - Periodの欄に数値（たとえば５０）を入力し、Apply
  - この数値がサイクルタイムの合成時の目標値になるので、面積重視で、サイクルタイムが長くてよい場合は大きな値を入れる。
  - 逆に、サイクルタイムを小さくしたければ、小さな値をいれる。

(5章) RAID用Encoder/Decoderの設計

- さて、以下のEncoder/Decoderを設計するわけですが、Encoder：式（1）、Decoder:式（2）に示すように、どちらも同じ行列の乗算をガロア体で行います。
- したがって、今回は行列パラメータをレジスタにセットすることで、Encoder/Decoderのどちらにも使える、ガロア体の行列乗算器を設計します。

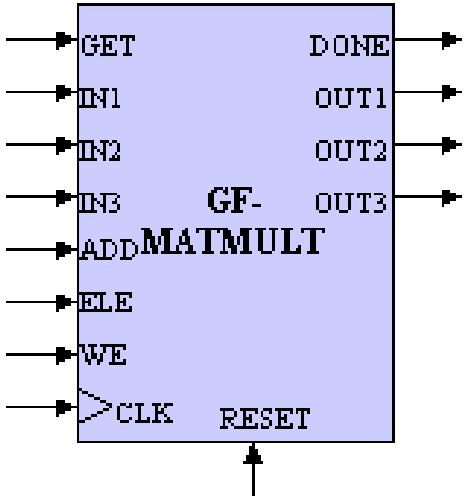


$$\begin{bmatrix} c1 \\ c2 \\ c3 \end{bmatrix} = \begin{bmatrix} 1^0 & 2^0 & 3^0 \\ 1^1 & 2^1 & 3^1 \\ 1^2 & 2^2 & 3^2 \end{bmatrix} \begin{bmatrix} d1 \\ d2 \\ d3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 4 & 5 \end{bmatrix} \begin{bmatrix} d1 \\ d2 \\ d3 \end{bmatrix} \quad \cdots \quad (1)$$

$$\begin{bmatrix} d1 \\ d2 \\ d3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 3 & 1 \\ 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} d1 \\ c1 \\ c2 \end{bmatrix} \quad \cdots \quad (2)$$

ガロア行列乗算器仕様（1）

- この乗算器をENCODERにもDECODERにも共用するために、各行列要素はアドレス信号(ADD)に指定された要素に係数(ELE)を書き込む。この書き込みはクロック信号(CLK)の立ち上がりエッジで書き込みコントロール信号(WE)をアサートすることで行われる。各要素とアドレスの関係は右下の図に示す。
- CLK立ち上がりエッジでGET=1の時のIN1/IN2/IN3を取り込んで、演算を行い、OUT1/OUT2/OUT3を計算します。この時、DONE＝1として、出力データができたことを外に知らせます。



$$\begin{bmatrix} OUT1 \\ OUT2 \\ OUT3 \end{bmatrix} = \begin{bmatrix} ELE[Add=0] & ELE[Add=1] & ELE[Add=2] \\ ELE[Add=3] & ELE[Add=4] & ELE[Add=5] \\ ELE[Add=6] & ELE[Add=7] & ELE[Add=8] \end{bmatrix} \begin{bmatrix} IN1 \\ IN2 \\ IN3 \end{bmatrix}$$

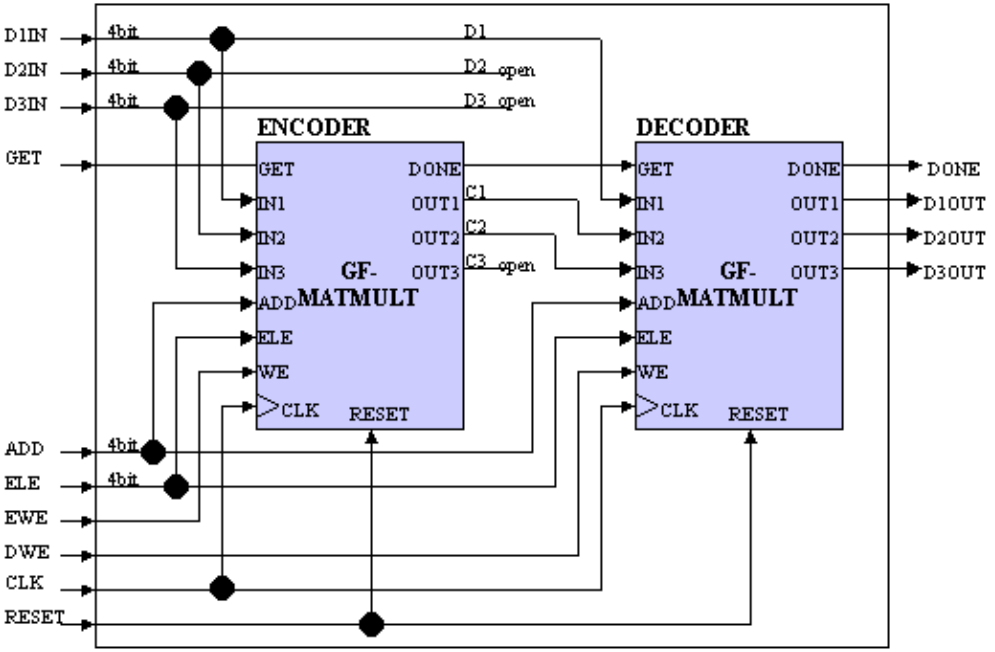
ガロア行列乗算器仕様 (2)

- 各ピンの信号の一覧です

信号名	入出力	ビット幅	説明	信号名	入出力	ビット幅	説明
GET	IN	1	Hで入力データを取込み計算開始	CLK	IN	1	クロック信号
IN1	IN	4	入力データ1	RESET	IN	1	Hでリセット
IN2	IN	4	入力データ2	DONE	OUT	1	Hで出力データ確定
IN3	IN	4	入力データ3	OUT1	OUT	4	出力データ1
ADD	IN	4	行列要素の位置を指定	OUT2	OUT	4	出力データ2
ELE	IN	4	行列要素入力	OUT3	OUT	4	出力データ3
WE	IN	1	Hで行列要素書き込み				

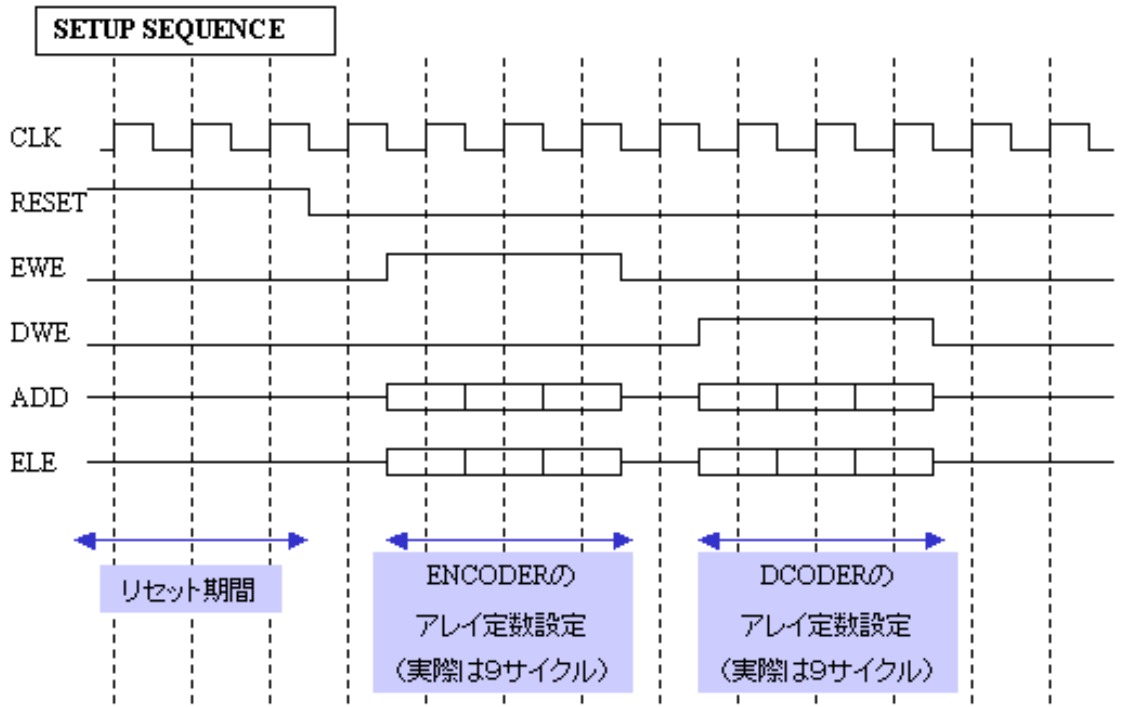
システム動作検証方法

- RS-RAIDシステムの動作検証のために以下図のシステムモデルを用いて、検証を行う。
- EWEはENCODER用のWE、DWEはDECODER用のWEであり、ADD/ELEは共有されている。
  EWE/DWEを用いてENCODER/DECODERに式 (1) 、 (2) に示される適切な行列要素を設定する。
- D1IN/D2IN/D3INに適当な入力を入れ、内部信号C1/C2/C3/DONE等をモニターし、最終的に
  D1OUT/D2OUT/D3OUTに再生された同じデータが出力されることを観測する



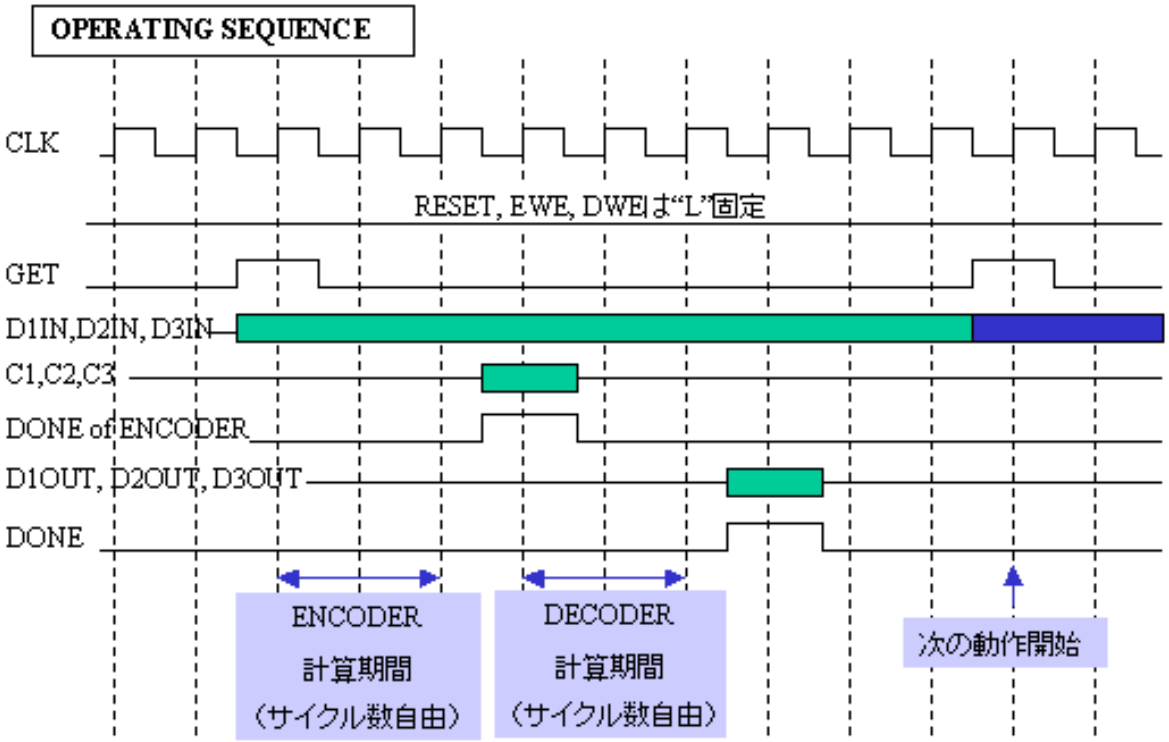
検証時の動作波形 (1)

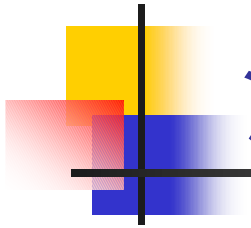
- Encoder / Decoderへの値の設定



検証時の動作波形 (2)

- RAID動作時の波形



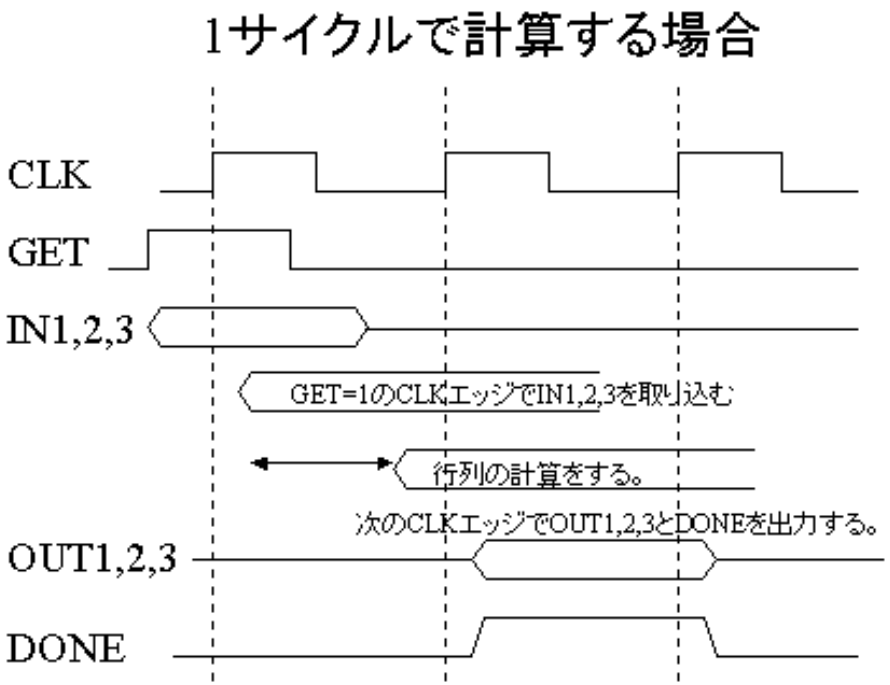


注意

- ガロア体行列乗算器のテンプレート gfmatmut.vhdを利用して設計してください。
- システム検証に対応する検証用コード test_rsraid.vhdを供給します。
- どうしてもできない人のために、例のコード示します。

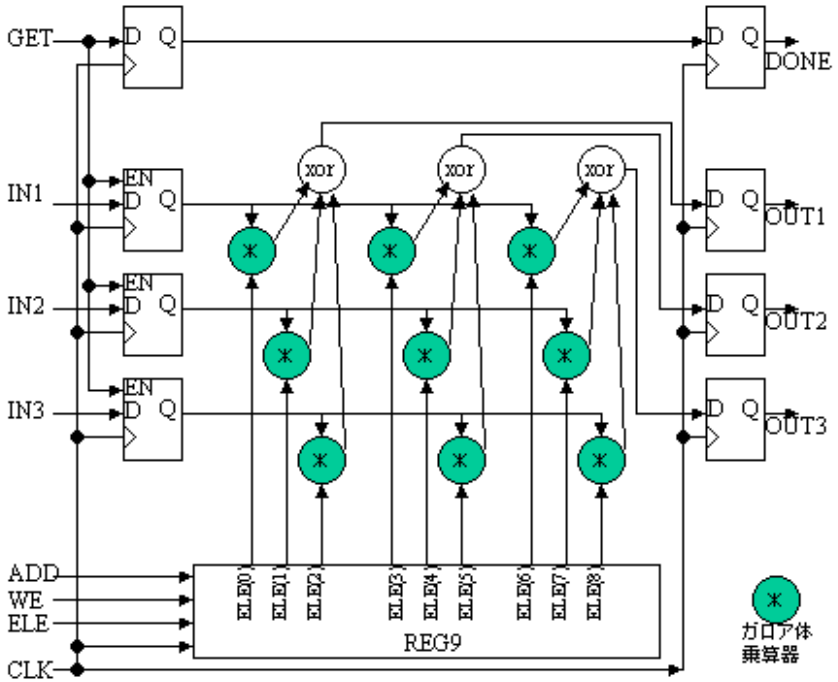
最も単純な構成 (1)

- 以下のような1サイクルでの計算が単純です！



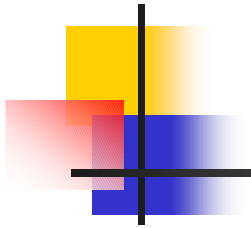
最も単純な構成 (2)

- 単純な構成はこのようになります。これを目指してつくりましょう！



gfmatmult.vhdの例

```
library IEEE;
use IEEE.STD_LOGIC_1164.all,
          IEEE.NUMERIC_STD.all;

entity GFMATMULT is
    port ( GET    : in  std_logic;
           IN1    : in  unsigned(3 downto 0);
           IN2    : in  unsigned(3 downto 0);
           IN3    : in  unsigned(3 downto 0);
           ADD    : in  unsigned(3 downto 0);
           ELE    : in  unsigned(3 downto 0);
           WE     : in  std_logic;
           CLK    : in  std_logic;
           RESET  : in  std_logic;
           DONE   : out std_logic;
           OUT1   : out unsigned(3 downto 0);
           OUT2   : out unsigned(3 downto 0);
           OUT3   : out unsigned(3 downto 0) );
end entity GFMATMULT;

architecture RTL of GFMATMULT is
    ---- GFMULT2 ----
    component GFMULT2
        port ( ina     : in  unsigned(3 downto 0);
               inb     : in  unsigned(3 downto 0);
               product : out unsigned(3 downto 0) );
    end component;
    ---- 4bit x 9 register type ----
    type RegType is array (0 to 8) of unsigned(3
        downto 0);
    ---- ELE register file ----
    signal elereg  : Regtype;
    ---- input registers ----
    signal din1, din2, din3 : unsigned(3 downto 0);
    ---- output registers ----
    signal dout1, dout2, dout3 : unsigned(3 downto 0);
    ---- done registers ----
    signal getd1, getd2  : std_logic;
    ---- partial products ----
    signal pd0,pd1,pd2,pd3,pd4,pd5,pd6,pd7,pd8 : unsigned(3 downto 0);

begin
    ---- REG9 write logic ----
    REG9_WT: process (CLK)
    begin
        if (rising_edge(CLK)) then
            if (WE='1') then elereg(TO_INTEGER(ADD)) <= ELE;
            end if;
        end if;
    end process REG9_WT;

    ---- input register logic ----
    INPUT_REG: process (CLK)
    begin
        if (rising_edge(CLK)) then
            if (GET='1') then
                din1 <= IN1;
                din2 <= IN2;
                din3 <= IN3;
            end if;
        end if;
    end process INPUT_REG;

    ---- gfmult2 array ---
    U0: GFMULT2 port map (elereg(0),din1, pd0);
    U1: GFMULT2 port map (elereg(1),din2, pd1);
    U2: GFMULT2 port map (elereg(2),din3, pd2);
    U3: GFMULT2 port map (elereg(3),din1, pd3);
    U4: GFMULT2 port map (elereg(4),din2, pd4);
    U5: GFMULT2 port map (elereg(5),din3, pd5);
    U6: GFMULT2 port map (elereg(6),din1, pd6);
    U7: GFMULT2 port map (elereg(7),din2, pd7);
    U8: GFMULT2 port map (elereg(8),din3, pd8);

    ---- output register logic ----
    OUTPUT_REG: process (CLK)
    begin
        if (rising_edge(CLK)) then
            dout1 <= pd0 xor pd1 xor pd2;
            dout2 <= pd3 xor pd4 xor pd5;
            dout3 <= pd6 xor pd7 xor pd8;
        end if;
    end process OUTPUT_REG;

    ---- done logic ----
    DONE_LOGIC: process(CLK)
    begin
        if (rising_edge(CLK)) then
            getd1 <= GET;
            getd2 <= getd1;
        end if;
    end process DONE_LOGIC;

    --- output assigns ---
    DONE <= getd2;
    OUT1 <= dout1;
    OUT2 <= dout2;
    OUT3 <= dout3;

end architecture RTL;
```



ここまで完了していたら！

- 単純な例ではGFMULT2を9個使用していました。GFMULT2を3個か、1個のみ使用するようにGFMATMULTを改造してください。