

2020年4月17日

ハードウェア設計論:3

ハードウェアにおける設計表現 ハードウェア設計記述言語VerilogHDL

状態遷移と順序機械&種々の記述

質問は随時はSLACK #2020s-ハードウェア設計論

verilogが実行できる状態にしておいてください。

<http://www.mos.t.u-tokyo.ac.jp/~ikeda/HWDesign/>

本日講義以降:

ユーザ名: HardwareDesign2020

パスワード: Makoto_Ikeda

本日の課題

課題2～課題4までを、次回(4月24日朝)までに提出を終えておくこと

次回は、課題5:乗算回路、課題6:FIFOを実装していただきます

テストベンチ

testcount4.v

システムタスク:
\$monitor: 変化ごとに表示

initial手続きブロック

システムタスク:
\$finish シミュレーション終了

always構文は条件が満たされるたびに実行される(つまり10単位時間ごとにckが~ck (ckの論理反転)になる→20周期のクロックが生成される)

```
module testcount4;
```

```
    wire [3:0] out;
```

```
    reg ck;
```

```
    initial begin
```

```
        $monitor( "%t %b %b", $time, ck, out);
```

```
        ck<=0;
```

```
        #350
```

```
        $finish;
```

```
    end
```

```
    always #10 ck <= ~ck;
```

```
    count4 cnt ( out, ck);
```

```
endmodule
```

initial構文はプログラム言語同様に逐次実行

```
% iverilog testcount4.v count4.v  
% ./a.out
```

```
0 0 xxxxx  
10 1 xxxxx  
20 0 xxxxx  
30 1 xxxxx
```

テスト対象のモジュールを呼び出す継続的代入同様に常に実行(変化が即時に伝搬する)

??????

リセットしなくてはレジスタ型変数の値が不定

テストベンチ

testcount4r.v

システムタスク:
\$monitor: 変化ごとに表示

initial手続きブロック

システムタスク:
\$finish シミュレーション終了

被検証用モジュールの
呼び出し(インスタンス化宣言)

モジュール名

```
module testcount4r;
    wire          [3:0] out;
    reg           ck, res;

    initial begin
        $monitor( "%t %b %b %b", $time, ck, res, out);
        ck<=0;
        res<=0;
        #40
        res <= 1;
        #350
        $finish;
    end

    always #10 ck <= ~ck;
    count4r cnt ( out, ck, res );

endmodule
```

入出力信号名

インスタンス名
(任意)

いざ実行

```
% iverilog testcount4r.v count4r.v  
% ./a.out
```

```
0 0 0 0000  
10 1 0 0000  
20 0 0 0000  
30 1 0 0000  
40 0 1 0000  
50 1 1 0001  
60 0 1 0001  
70 1 1 0010  
80 0 1 0010  
90 1 1 0011  
100 0 1 0011  
110 1 1 0100  
  
370 1 1 0001  
380 0 1 0001
```

GUI出力用のテストベンチ

```
module testcount4rgui;
```

```
    wire            [3:0] out;
```

GUI表示用データの出力制御

```
    reg            ck, res;
```

```
    initial begin
```

```
        $dumpfile( "count4.vcd" );
```

```
        $dumpvars;
```

システムタスク:

\$monitor: 変化ごとに表示

```
        $monitor( "%t %b %b %b", $time, ck, res,
```

```
        out);
```

```
        ck<=0;
```

```
        res<=0;
```

```
        #40
```

```
        res <= 1;
```

```
        #350
```

```
        $finish;
```

```
    end
```

システムタスク:

\$finish シミュレーション終了

被検証用モジュールの
呼び出し(インスタンス化宣言)

```
    always #10 ck <= ~ck;
```

```
    count4r cnt ( out, ck, res );
```

```
endmodule
```

モジュール名

入出力信号名

インスタンス名
(任意)

記述誤りとエラーの例

<http://www.mos.t.u-tokyo.ac.jp/~ikeda/HWDesign/>

から alu_e1.v .. alu_e5.v をダウンロードして iverilog でコンパイル

alu_e1.v

alu_e1.v:8: syntax error

alu_e1.v:7: error: syntax error in reg variable list.

きちんと記述しているはずなのにエラーが出る場合たいていは、前の行の最後の ; が不在
その結果、"syntax error in reg variable list" となる。

alu_e2.v

alu_e2.v:13: error: C is not a reg/integer/time in alu.

alu_e2.v:7: : C is declared here as wire.

Elaboration failed

wire に対して手続き代入をしようとしている。

alu_e3.v

alu_e3.v:16: error: reg OUT; cannot be driven by primitives or continuous assignment.

2 error(s) during elaboration.

reg に対して継続代入をしようとしている。

alu_e4.v

alu_e4.v:1: error: Port CTR (4) of module alu is not declared within module.

alu_e4.v:12: error: Unable to bind wire/reg/memory `CTR' in `alu'

Elaboration failed

入出力ポートの定義がない

alu_e5.v

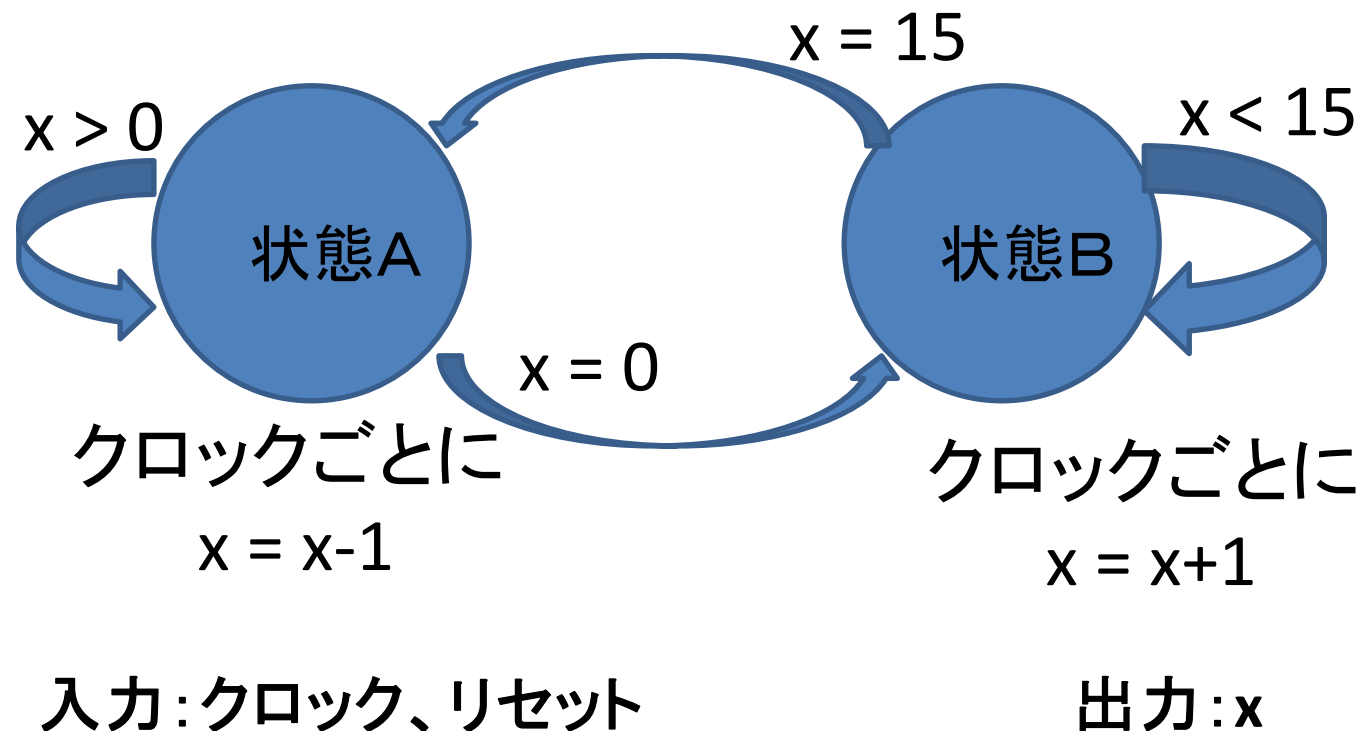
alu_e5.v:6: error: CTR in module alu declared as input and as a reg type.

1 error(s) during elaboration.

入力ポートに対して reg 定義をしようとしている

順序機械の実現

- 簡単な状態機械を実現してみよう



状態の定義

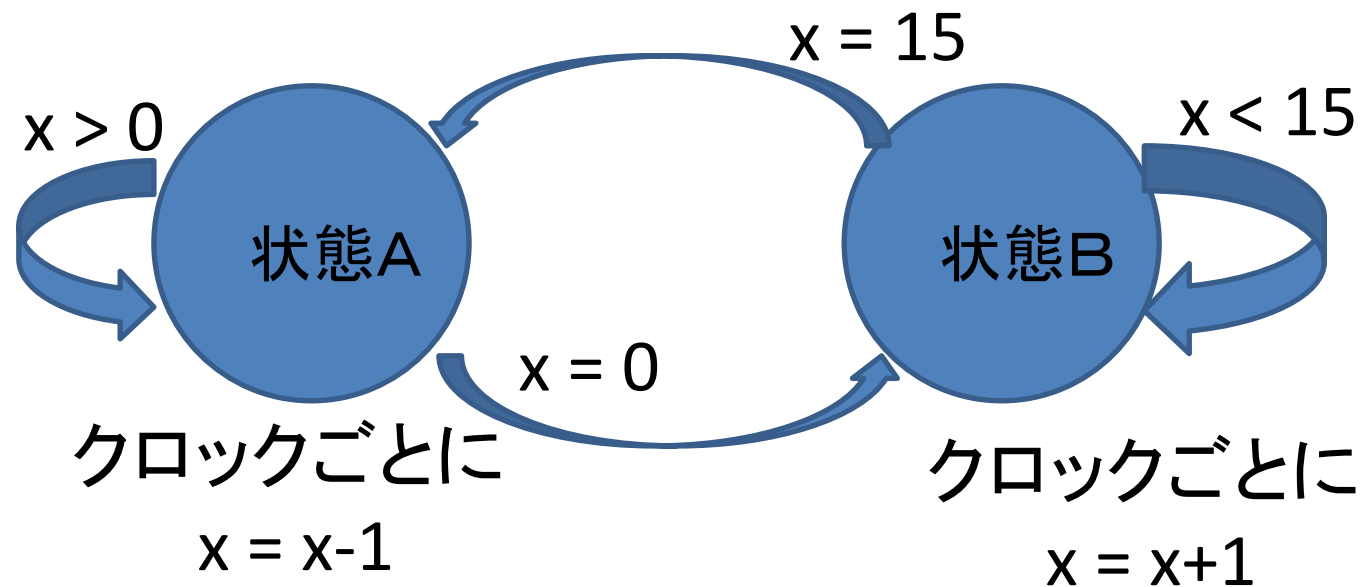
状態変数: st ,

状態A: $st=0$, 状態B: $st=1$,

リセット時状態: A

変数: x


リセット時: $x=0$



状態機械の構成

```
always @(posedge ck) begin
    if( st == 0 ) begin // State A
        if( x == 0 ) st <= 1;
        else      x<=x-1;
    end else begin // State B
        if( x == 15 ) st <= 0;
        else      x<=x+1;
    end
end
end
```

全体

```
module stm(ck,rst,x);
input  ck,rst;
output [3:0] x;
reg [3:0] x;
reg      st;
always @(posedge ck) begin
    if( rst == 1 ) begin
        st <= 0;
        x <= 0;
    end else begin

    end
end
endmodule
```

stm.v

全体

```
module stm(ck,rst,x);
input    ck,rst;
output   [3:0] x;
reg [3:0] x;
regst;
always @(posedge ck) begin
    if( rst == 1 ) begin
        st <= 0;
        x <= 0;
    end else begin
        if( st == 0 ) begin
            if( x == 0 ) st <= 1;
            else x <= x-1;
        end else begin
            if( x == 15 ) st <= 0;
            else x <= x+1;
        end
    end
end
endmodule
```

複数行にわたる場合には
必ず begin ~ endでくる

simstm.v

テストベンチ

```
module simstm;
reg ck, rst;
wire [3:0] x;
initial begin
    $dumpvars;
    $dumpfile("stm.vcd");
    $monitor( "st = %b: x=%x", s.st, x );
    ck=0; rst=0;
    #20 rst=1;
    #60 rst=0;
    #1000 $finish;
end
always #10      ck=~ck;

stm    s(ck,rst,x);

endmodule
```

↓
モジュール内
変数の参照

VerilogHDLの実行結果の確認

- 実行結果

```
% iverilog simstm.v stm.v
```

```
% ./a.out
```

- エラー例

```
% iverilog simstm.v stm.v
```

```
simstm.v:9: syntax error
```

```
simstm.v:8: error: malformed statement
```

ありがちなエラー

wire型に手続き代入 (`<=`)をしようとしている

reg型に継続代入(`assign` 文)をしようとしている

VerilogHDLの実行結果の確認

- 正常な場合

ck = 0, st = x: x=x

ck = 1, st = x: x=x

ck = 0, st = x: x=x

ck = 1, st = 0: x=0

.

ck = 0, st = 1: x=e

ck = 1, st = 1: x=f

ck = 0, st = 1: x=f

ck = 1, st = 0: x=f

ck = 0, st = 0: x=f

ck = 1, st = 0: x=e

.

ck = 0, st = 0: x=1

ck = 1, st = 0: x=0

ck = 0, st = 0: x=0

ck = 1, st = 1: x=0

ck = 0, st = 1: x=0

ck = 1, st = 1: x=1

.

初期化されるまでは値は x を持つ

x=f の時は状態のみを遷移させる

状態遷移後、次のクロックからxの減算が始まる

x=0 の時は状態のみを遷移させる

状態遷移後、次のクロックからxの加算が始まる

simstm.v

```
module simstm;  
reg ck, rst;  
wire [3:0] x;  
initial begin  
    $dumpvars;  
    $dumpfile("stm.vcd");  
    $monitor("st = %b: x=%x", s.st, x);  
    ck=0; rst=0;  
    #20 rst=1;  
    #60 rst=0;  
    #1000 $finish;  
end  
always #10 ck=~ck;  
  
stm s(ck,rst,x);  
  
endmodule
```

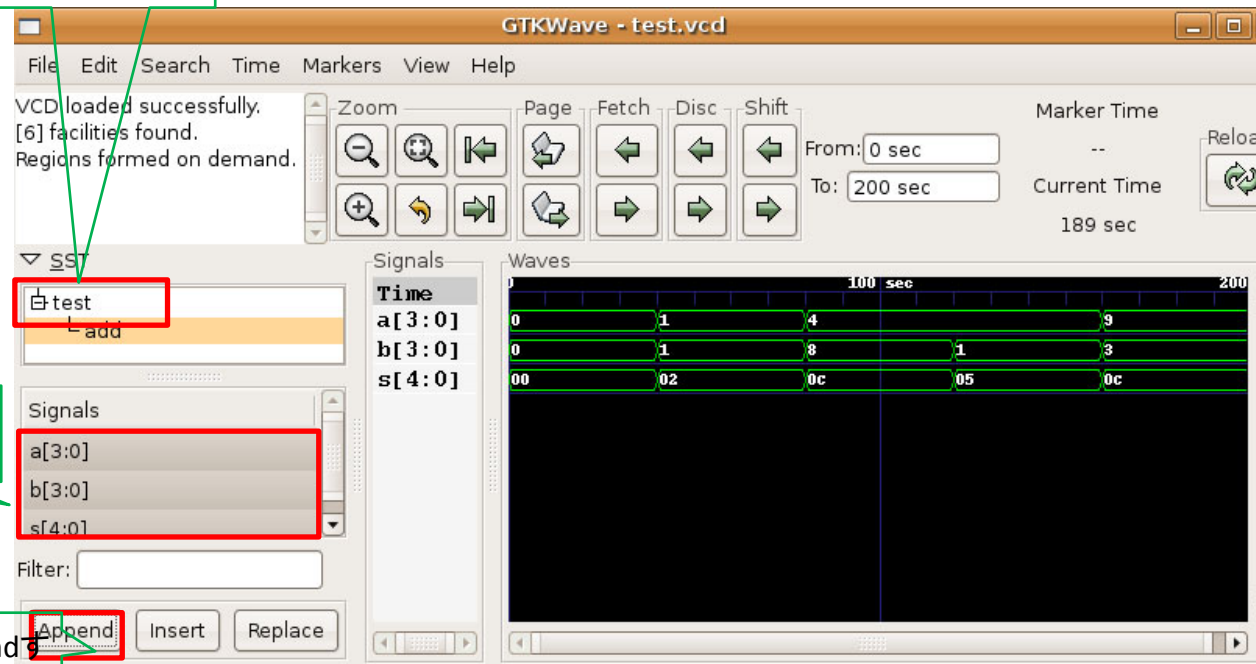
グラフィカルに見たい

```
% iverilog simstm.v stm.v  
% ./a.out  
% gtkwave stm.vcd
```

クリックすると下位モジュール名が表示される

選択したモジュール内の信号が表示される

信号を選択してAppendすると右画面に波形が表示



モジュール間の変数の参照

```
module A (*****)  
reg hoge1, hoge2, hoge3;  
endmodule
```

```
module TOP_A;
```

```
A InstanceNameA (**** );
```

```
endmodule
```

simstm.v

テストベンチ

```
module simstm;
```

```
stm s(ck,rst,x);
```

```
$monitor( "st = %b: x=%x", s.st, s.x );
```

ハードウェアとしては、module Aの内部の変数を参照するためには、ポートから出力する必要がある。

→ 不便なのでシミュレーションとしては、

TOP_Aモジュールから

InstanceNameA . hoge1

という形で InstanceNameAとして定義したモジュール内の変数を参照することができる

演習2

- add4.v, testadd41.vをダウンロードして実行
- add4.vを修正し減算をするsub4.v, 乗算をするmul4.vを作成し、それらに対応するテストベンチ testsub41.v, testmul41.vを作成して実行し結果を確認

課題2-1 sub4.v

課題2-2 mul4.v

演習3

- count4r.v, testcount4r.vをダウンロードして実行
- count4r.vを修正しデクリメント(クロックごとに-1)するcount4rs.v, 2つつ加算するcount4r2.v, 2倍つつ乗算するcount4r2m.vを作成し、それらに対応するテストベンチtestcount4rs.v, testcount4r2.v, testcount4r2m.v を作成して実行し結果を確認

課題3-1 count4rs.v

課題3-2 count4r2.v

課題3-3 count4r2m.v

演習4: 簡単な演算器

- 入力A: 8ビット、入力B: 8ビット、出力O: 8ビット
- 制御入力CTR: 4ビット
 - 0000: 加算、0001: 減算
 - 1000: 論理積、1001: 論理和、1010: 排他的論理和、1011: 反転、
 - 1100: 1ビット右シフト(0で埋める)、1101: 1ビット左シフト(0で埋める)、
 - 1110: 1ビット右ローテーション(MSBをLSBで埋める)、
 - 1111: 1ビット左ローテーション(LSBをMSBで埋める)
- 入力はクロックの立ち上がりで取り込み、1クロック後の立ち上がりで出力

暗黙の了解・・・1: 定義していない制御入力の場合の出力は？ ここでは0にする

暗黙の了解・・・2: タイミング: ここでは、

すべての入力はクロックの立ち上がりで内部の(入力)レジスタに取り込む
演算結果はクロックの立ち上がりで内部の(出力)レジスタに取り込む
(出力)レジスタの結果を出力として外部に出力する

演習2のヒント

```
module sub4(s,a,b);
```

```
  output  [4:0] s;
```

```
  input   [3:0] a,b;
```



```
endmodule
```

ここを埋めてください

```
module mul4(s,a,b);
```

```
  output  [7:0] s;
```

```
  input   [3:0] a,b;
```



```
endmodule
```

結果の例

0	0000	0	-	0000	0	=	00000	0
40	0001	1	-	0011	3	=	11110	-2
80	0100	4	-	1000	8	=	11100	-4
120	0100	4	-	0001	1	=	00011	3
160	1001	9	-	0011	3	=	00110	6

2の補数表現
1,0を反転させて1を足す

結果の例

0	0000	*	0000	=	00000000
40	0001	*	0011	=	00000011
80	0100	*	1000	=	00100000
120	0100	*	0001	=	00000100
160	1001	*	0011	=	00011011

演習2のテストベンチの例

```
module testsub4;
  wire [4:0] s;
  reg [3:0] a, b;
  initial begin
    $monitor( "%t %b - %b = %b", $time, a, b, s);
    a = 0; b = 0;
    #40 a = 1; b = 3;
    #40 a = 4; b = 8;
    #40 a = $random; b = $random;
    #40 a = $random; b = $random;
    #40
    $finish;
  end
  sub4 sub ( s,a,b );
endmodule
```

```
module testmul4;
  wire [7:0] s;
  reg [3:0] a, b;
  initial begin
    $monitor( "%t %b * %b = %b", $time, a, b, s);
    a = 0; b = 0;
    #40 a = 1; b = 3;
    #40 a = 4; b = 8;
    #40 a = $random; b = $random;
    #40 a = $random; b = $random;
    #40
    $finish;
  end
  mul4 mul ( s,a,b );
endmodule
```

注意: ハードウェア記述そのものでは「ノンブロッキング代入を用い
なくてはならないが、テストベンチ記述においては、「ノンブロッキング
」、「ブロッキング」どちらを用いても結果に差は生じない

testadd42.v もう少し洒落たテストベンチ

```
module testadd42;
  wire                [4:0] s;
  reg      [3:0] a, b;
  reg      ck;
  initial begin
    $monitor( "%t %b + %b = %b", $time, a, b, s);
    a = 0; b = 0;
    ck = 0;
    #400
    $finish;
  end
  always #10 ck = ~ck;
  always @(posedge ck) begin
    a = $random;
    b = $random;
  end
  add4 add ( s,a,b );
endmodule
```

```
% iverilog testadd42.v add4.v
% ./a.out
```

```
0 0000 + 0000 = 00000
10 0100 + 0001 = 00101
30 1001 + 0011 = 01100
50 1101 + 1101 = 11010
70 0101 + 0010 = 00111
90 0001 + 1101 = 01110
110 0110 + 1101 = 10011
    ...
350 1010 + 1101 = 10111
370 0110 + 0011 = 01001
390 1101 + 0011 = 10000
```

testadd43.v

全数をチェックしたければ

```
module testadd43;
  wire      [4:0] s;
  reg       [3:0] a, b;
  reg       ck;

  initial begin
    $monitor( "%t %b + %b = %b", $time, a, b, s);
    a = 0; b = 0;
    ck = 0;
  end
  always #10 ck = ~ck;
  always @(negedge ck) begin
    if( s != a + b ) begin
      $finish;
    end
    if( a == 'h f && b == 'h f ) begin
      $display( "OK¥n" );
      $finish;
    end
  end
  always @(posedge ck) begin
    {b,a} = {b,a} + 1;
  end
  add4 add ( s,a,b );
endmodule
```

接続

% iverilog testadd43.v add4.v

% ./a.out

```
0 0000 + 0000 = 00000
10 0001 + 0000 = 00001
30 0010 + 0000 = 00010
50 0011 + 0000 = 00011
70 0100 + 0000 = 00100
90 0101 + 0000 = 00101
...
5030 1100 + 1111 = 11011
5050 1101 + 1111 = 11100
5070 1110 + 1111 = 11101
5090 1111 + 1111 = 11110
```

OK

テストベンチ

```
simfulladd.v
module simfulladd;
  wire      s, cout;
  reg       x, y, cin, ck, flag;
  initial begin
    $monitor( "%t Input (x, y, cin) -> Output (s, cout): (%b, %b, %b) -> (%b, %b)", $time, x, y, cin, s, cout);
    x <= 0; y <= 0; cin <= 0; ck <= 0; flag <= 0;
  end
  always #10 ck <= ~ck;
  always @(negedge ck) begin
    if( s != (x ^ y ^ cin) || cout != (x & y | x & cin | y & cin) ) begin
      flag <= 1;
      $finish;
    end
    if( {cin,x,y} == 3'b 111 ) begin
      $display( "OK¥n" );
      $finish;
    end
  end
  always @(posedge ck) begin
    {cin,x,y} <= {cin,x,y} + 1;
  end
  FullAdderFunction add ( x, y, cin, cout, s );
endmodule
```

FullAdderStructureも同様

演習3

- count4r.v, testcount4r.vをダウンロードして実行
- count4r.vを修正しデクリメント(クロックごとに-1)するcount4rs.v, 2つつ加算するcount4r2.v, 2倍つつ乗算するcount4r2m.vを作成し、それらに対応するテストベンチtestcount4rs.v, testcount4r2.v, testcount4r2m.v を作成して実行し結果を確認

課題3-1 count4rs.v

課題3-2 count4r2.v

課題3-3 count4r2m.v

演習3の期待される結果

count4rs.v	count4r2.v	初期値を1(0以外)に しておく必要あり	count4r2m.v
0 0 0 0000	0 0 0 0000		0 0 0 0001
10 1 0 0000	10 1 0 0000		10 1 0 0001
20 0 0 0000	20 0 0 0000		20 0 0 0001
30 1 0 0000	30 1 0 0000		30 1 0 0001
40 0 1 0000	40 0 1 0000		40 0 1 0001
50 1 1 1111	50 1 1 0010		50 1 1 0010
60 0 1 1111	60 0 1 0010		60 0 1 0010
70 1 1 1110	70 1 1 0100	オーバーフローす ると以後0となる	70 1 1 0100
80 0 1 1110	80 0 1 0100		80 0 1 0100
90 1 1 1101	90 1 1 0110		90 1 1 1000
100 0 1 1101	100 0 1 0110		100 0 1 1000
			110 1 1 0000
			120 0 1 0000
370 1 1 1111	370 1 1 0010		
380 0 1 1111	380 0 1 0010		
390 1 1 1110	390 1 1 0100		390 1 1 0000

VerilogHDLの基本構文：関数・タスク

- function: 戻り値は関数名を左辺に指定した代入
function ビット幅 関数名;
 input ビット幅 変数名;
 宣言
 シーケンシャル文
endfunction

関数呼び出しは
a <= 関数名(引数)

- task: 戻り値はoutputもしくはinout変数として宣言
task タスク名;
 input ビット幅 変数名;
 output ビット幅 変数名;
 宣言
 シーケンシャル文
endtask

タスク呼び出しは
タスク名(引数1, 引数2・・・)

演習4: 簡単な演算器・・・1

- 入力A: 8ビット、入力B: 8ビット、出力O: 8ビット
- 制御入力CTR: 4ビット
 - 0000: 加算、0001: 減算
 - 1000: 論理積、1001: 論理和、1010: 排他的論理和、1011: 反転、
 - 1100: 1ビット右シフト(0で埋める)、1101: 1ビット左シフト(0で埋める)、
 - 1110: 1ビット右ローテーション(MSBをLSBで埋める)、
 - 1111: 1ビット左ローテーション(LSBをMSBで埋める)
- 入力はクロックの立ち上がりで取り込み、1クロック後の立ち上がりで出力

暗黙の了解・・・1: 定義していない制御入力の場合の出力は？ ここでは0にする

暗黙の了解・・・2: タイミング: ここでは、
すべての入力はクロックの立ち上がりで内部の(入力)レジスタに取り込む
演算結果はクロックの立ち上がりで内部の(出力)レジスタに取り込む
(出力)レジスタの結果を出力として外部に出力する

簡単な演算器とは

制御コード	機能	Verilog記述	ほかの記述
0000	加算	$A+B$	
0001	減算	$A-B$	
1000	論理積	$A \& B$	
1001	論理和	$A B$	
1010	排他的論理和	$A \wedge B$	
1011	反転	$\sim A$	
1100	1ビット右シフト	$A \gg 1$	$\{1'b0, A[7:1]\}$
1101	1ビット左シフト	$A \ll 1$	$\{A[6:0], 1'b0\}$
1110	1ビット右ローテーション		$\{A[0], A[7:1]\}$
1111	1ビット左ローテーション		$\{A[6:0], A[7]\}$
0010, 0011, 0100, 0101, 0110, 0111,	定義なし: 0を出力		

実装例1

alu.v

骨格

```
module alu(A,B,O,CTR,ck);  
input  [7:0]  A, B;  
input  [3:0]  CTR;  
input   ck;  
output [7:0] O;  
reg    [7:0] INA, INB, O;  
reg    [3:0] C;  
wire   [7:0] OUT;  
  
endmodule
```

alu.vとして作成しuploadすること

順序機械。。 1

```
always @(posedge ck) begin  
    INA <= A;  
    INB <= B;  
    C <= CTR;  
    O <= OUT;  
end
```

継続代入で実現

```
assign OUT=(C=='b0000 ? INA + INB :  
            (C=='b0001 ? INA - INB :
```

```
)))))))));
```

8'b0

実装例1

alu.v

骨格

```
module alu(A,B,O,CTR,ck);
input  [7:0]  A, B;
input  [3:0]  CTR;
input  ck;
output [7:0] O;
reg    [7:0] INA, INB, C;
reg    [3:0] C;
wire   [7:0] OUT;

endmodule
```

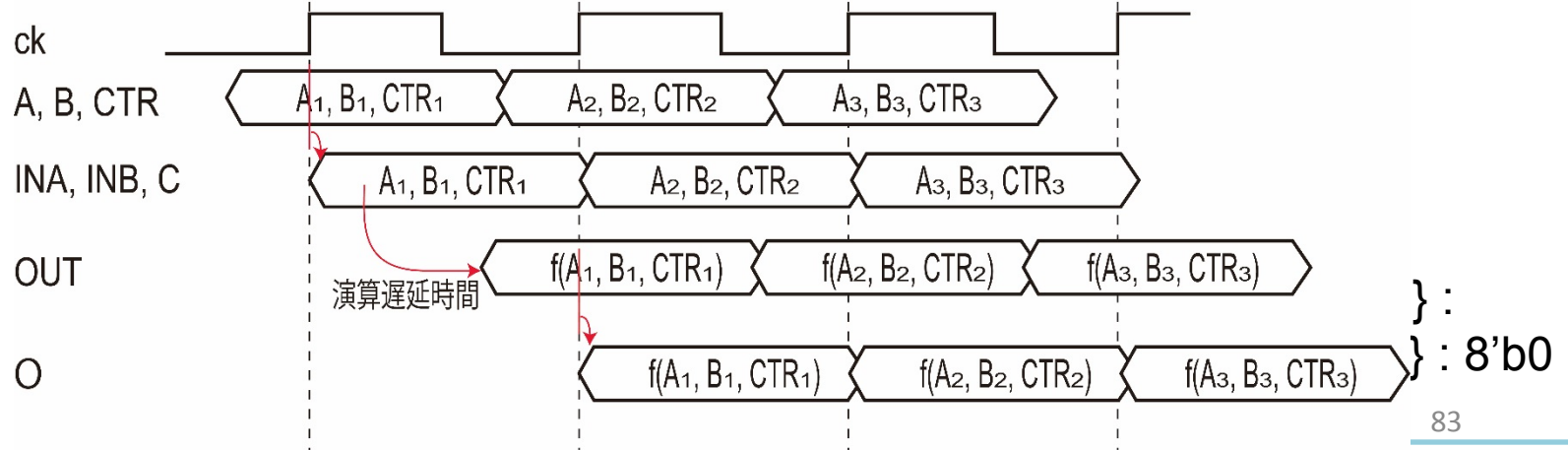
順序機械。。 1

```
always @(posedge ck) begin
    INA <= A;
    INB <= B;
    C <= CTR;
    O <= OUT;
end
```

継続代入で実現

```
assign OUT=(C=='b0000 ? INA + INB :
            (C=='b0001 ? INA - INB :
            (C=='b1000 ? INA & INB :
            8'b0
```

endmodule



実装例2

alu2.v

骨格

```
module alu(A,B,O,CTR,ck);  
input  [7:0]  A, B;  
input  [3:0]  CTR;  
input   ck;  
output [7:0] O;  
reg    [7:0]  INA, INB, OUT, O;  
reg    [3:0]  C;  
  
endmodule
```

順序機械

```
always @(posedge ck) begin  
    C <= CTR;  
    INA <= A;  
    INB <= B;  
    case (C)  
        'b0000 : O <= INA + INB;  
        'b0001 : O <= INA - INB;  
    endcase  
end
```

実装例2

alu2.v

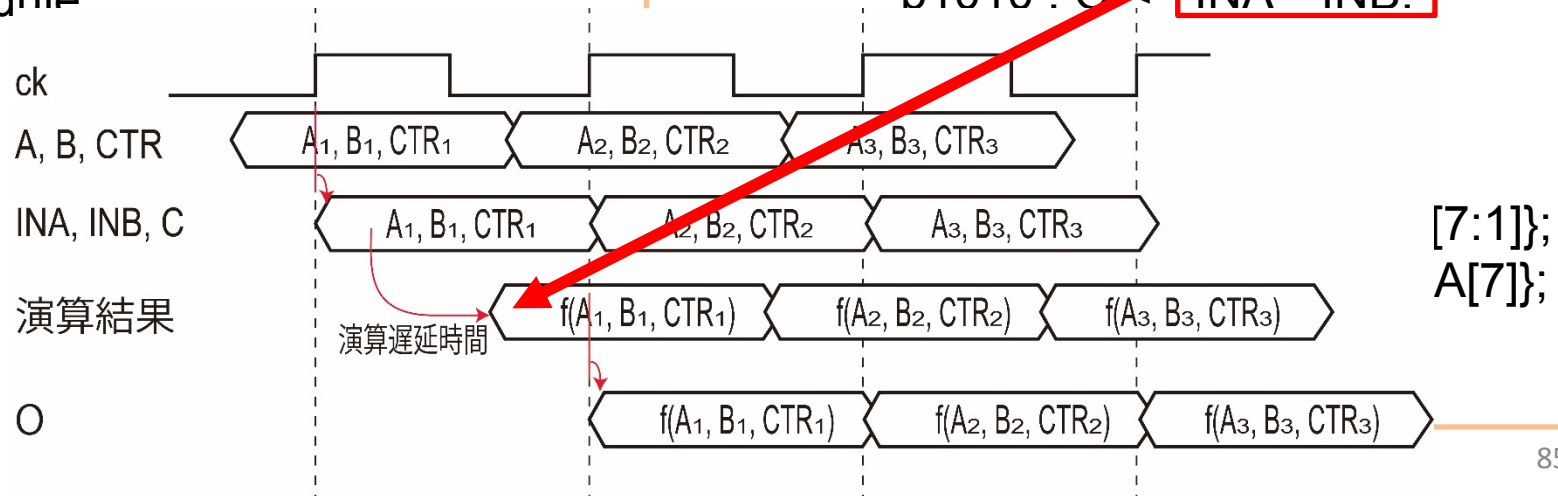
骨格

```
module alu(A,B,O,CTR,ck);
input  [7:0]  A, B;
input  [3:0]  CTR;
input  ck;
output [7:0] O;
reg    [7:0] INA, INB, OUT, O;
reg    [3:0] C;

endmodule
```

順序機械

```
always @(posedge ck) begin
    C <= CTR;
    INA <= A;
    INB <= B;
    case (C)
        'b0000 : O <= INA + INB;
        'b0001 : O <= INA - INB;
        'b1000 : O <= INA & INB;
        'b1001 : O <= INA | INB;
        'b1010 : O <= INA ^ INB;
```



実装例2.1

順序機械。。2

alu21.v

骨格

```
module alu(A,B,O,CTR,ck);
input  [7:0]  A, B;
input  [3:0]  CTR;
input  ck;
output [7:0] O;
reg    [7:0]  INA, INB, OUT, O;
reg    [3:0]  C;

endmodule
```

```
always @(posedge ck) begin
    C <= CTR;
    INA <= A;
    INB <= B;
    O <= OUT;
end
always @(C or INA or INB) begin
    case (C)
        'b0000 : OUT <= INA + INB;
        'b0001 : OUT <= INA - INB;
```

```
endcase
```

```
end
```

実装例2-1

順序機械。。2

alu21.v

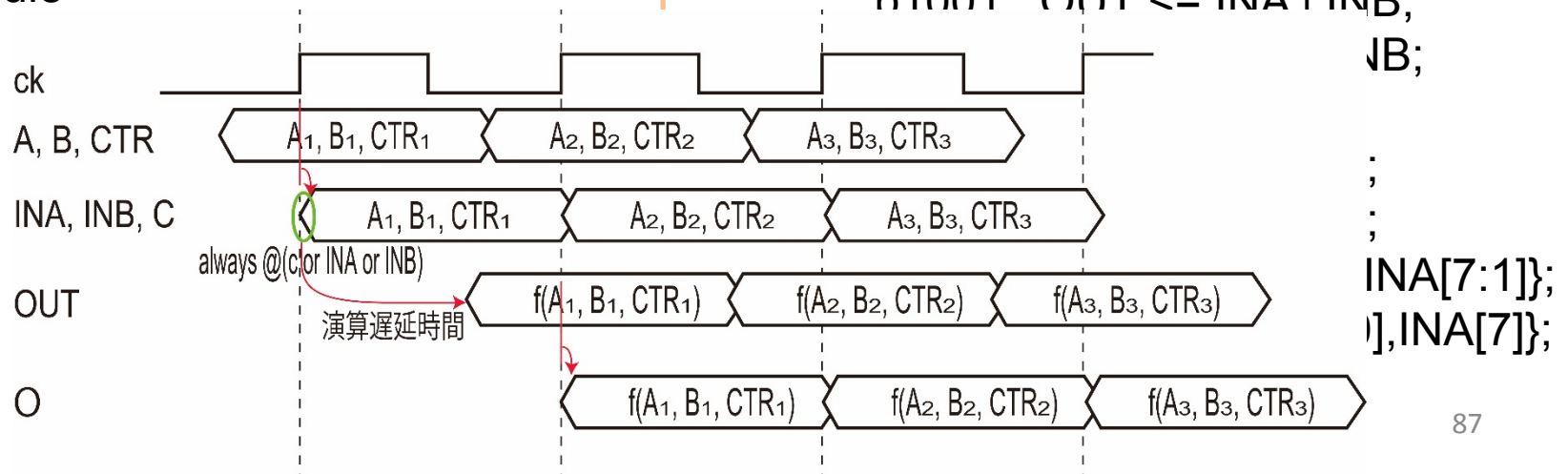
骨格

```
module alu(A,B,O,CTR,ck);
input  [7:0]  A, B;
input  [3:0]  CTR;
input  ck;
output [7:0] O;
reg    [7:0]  INA, INB, OUT, O;
reg    [3:0]  C;

endmodule
```

```
always @(posedge ck) begin
    C <= CTR;
    INA <= A;
    INB <= B;
    O <= OUT;
end

always @(C or INA or INB) begin
    case (C)
        'b0000 : OUT <= INA + INB;
        'b0001 : OUT <= INA - INB;
        'b1000 : OUT <= INA & INB;
        'b1001 : OUT <= INA | INB;
```



実装例2・・・2

だめな例:

ブロッキング代入をしてしまうと、タイミングがずれてしまう

alu22.v

骨格

```
module alu(A,B,O,CTR,ck);  
input  [7:0]  A, B;  
input  [3:0]  CTR;  
input  ck;  
output [7:0] O;  
reg    [7:0]  INA, INB, OUT, O;  
reg    [3:0]  C;  
  
endmodule
```

順序機械

```
always @(posedge ck) begin  
    C = CTR;  
    INA <= A;  
    INB <= B;  
    case (C)  
        'b0000 : O <= INA + INB;  
        'b0001 : O <= INA - INB;  
  
    endcase  
end
```

実装例3 : functionを使用

alu3.v

骨格

```
module alu(A,B,O,CTR,ck);  
input  [7:0]  A, B;  
input  [3:0]  CTR;  
input   ck;  
output [7:0] O;  
reg    [7:0]  INA, INB, O;  
reg    [3:0]  C;
```

endmodule

順序機械。。 2

```
always @(posedge ck) begin  
    C <= CTR;  
    INA <= A;  
    INB <= B;  
    O <= alufunc(INA,INB,C);  
end
```

function

```
function [7:0] alufunc;  
    input [7:0] A;  
    input [7:0] B;  
    input [3:0] C;  
  
    case (C)  
        'b0000 : alufunc = A + B;  
        'b0001 : alufunc = A - B;
```

endcase

endfunction

実装例3 : functionを使用

alu3.v

骨格

```
module alu(A,B,O,CTR,ck);
input  [7:0]  A, B;
input  [3:0]  CTR;
input  ck;
output [7:0] O;
reg    [7:0]  INA, INB, O;
reg    [3:0]  C;
```

endmodule

順序機械。 2

```
always @(posedge ck) begin
    C <= CTR;
    INA <= A;
    INB <= B;
    O <= alufunc(INA,INB,C);
end
```

function

```
function [7:0] alufunc;
input  [7:0] A;
input  [7:0] B;
input  [3:0] C;

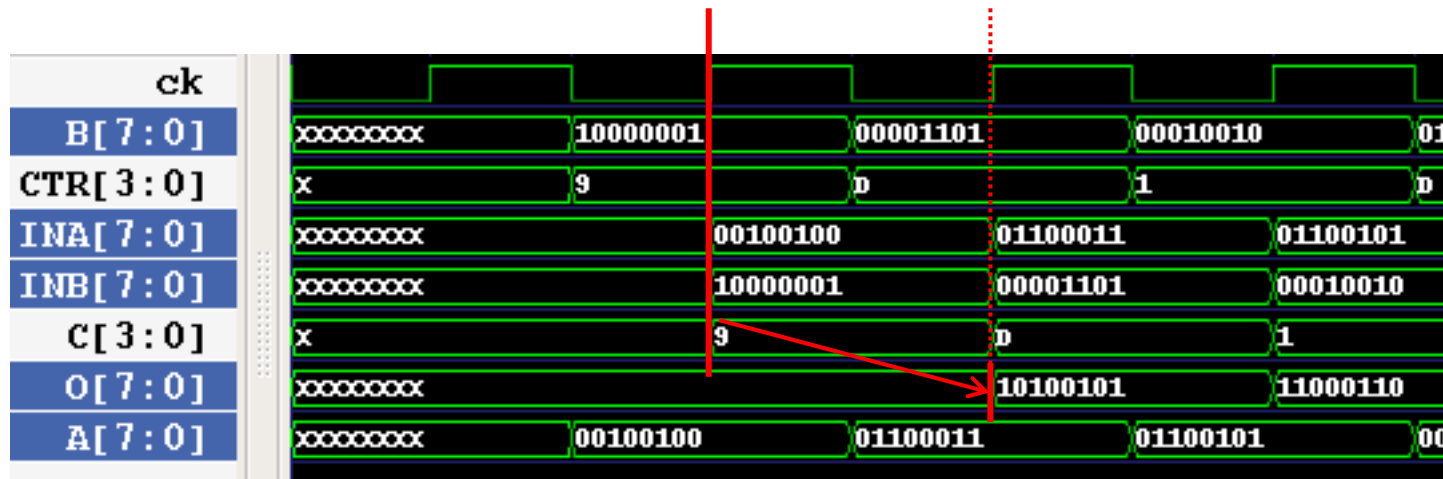
case (C)
'b0000 : alufunc = A + B;
'b0001 : alufunc = A - B;
'b1000 : alufunc = A & B;
'b1001 : alufunc = A | B;
'b1010 : alufunc = A ^ B;
'b1011 : alufunc = ~A;
'b1100 : alufunc = A>>1;
'b1101 : alufunc = A<<1;
'b1110 : alufunc = {A[0], A[7:1]};
'b1111 : alufunc = {A[6:0], A[7]};
endcase
endfunction
```

テストベンチ

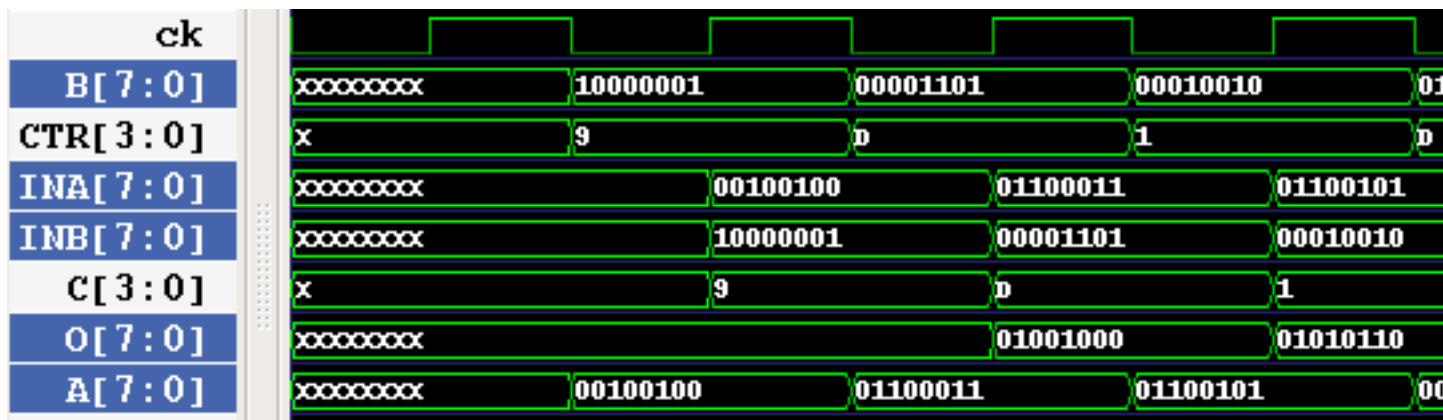
骨格 **alutest.v**

```
module alutest;
reg      [7:0]    A, B;
reg      [3:0]    CTR;
reg      ck;
wire     [7:0] O;
initial begin
    ck=0;
    $monitor( "%t\tA=%h, B=%h, CTR=%h, OUT=%h", $time, A, B, CTR, O );
    #1000 $finish;
end
alu      ALU(A , B , O , CTR , ck);
always   #10      ck = ~ck;
always   @(negedge ck) begin
    A = $random;
    B = $random;
    CTR = $random;
end
endmodule
```

実行結果を確認してみましょう



alu.v



alu22.v

課題5 mul.v (multest2.vを使用)

演習5: 乗算の実装

```

1101 被乗数 a
1011 乗数 b
-----
1101 部分積 a*b[0]
1101 部分積 a*b[1]
0000 部分積 a*b[2]
1101 部分積 a*b[3]
-----
10001111 積(部分積の総和)

```

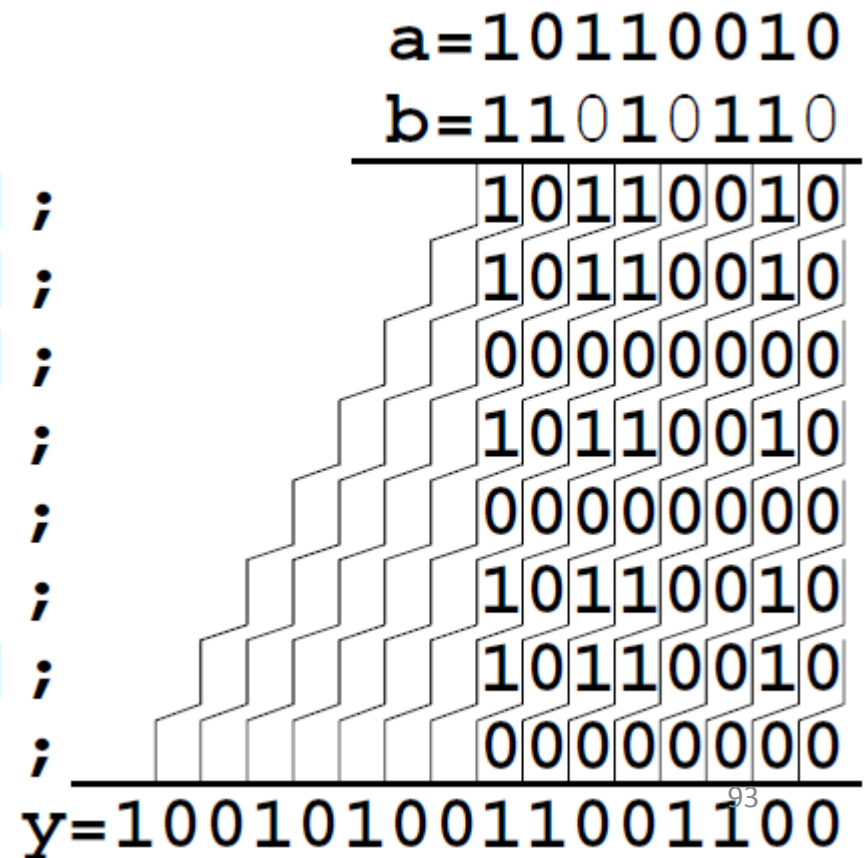
「*」を使わない乗算の記述

ソフトウェア的な記述＝ブロッキング代入(逐次実行)で記述

```

y = 0;
y = (y<<1) + a*b[7];
y = (y<<1) + a*b[6];
y = (y<<1) + a*b[5];
y = (y<<1) + a*b[4];
y = (y<<1) + a*b[3];
y = (y<<1) + a*b[2];
y = (y<<1) + a*b[1];
y = (y<<1) + a*b[0];

```



乗算の実装：複数サイクルで実行

```

1101 被乗数 a
1011 乗数 b
1101 部分積 a*b[0]
1101 部分積 a*b[1]
0000 部分積 a*b[2]
1101 部分積 a*b[3]
10001111 積(部分積の総和)
    
```

入力 A, B, ck, start

start=1で A, Bを内部レジスタ AIN, BINに取り込み

状態変数 stを0、終了フラグ finを0とする

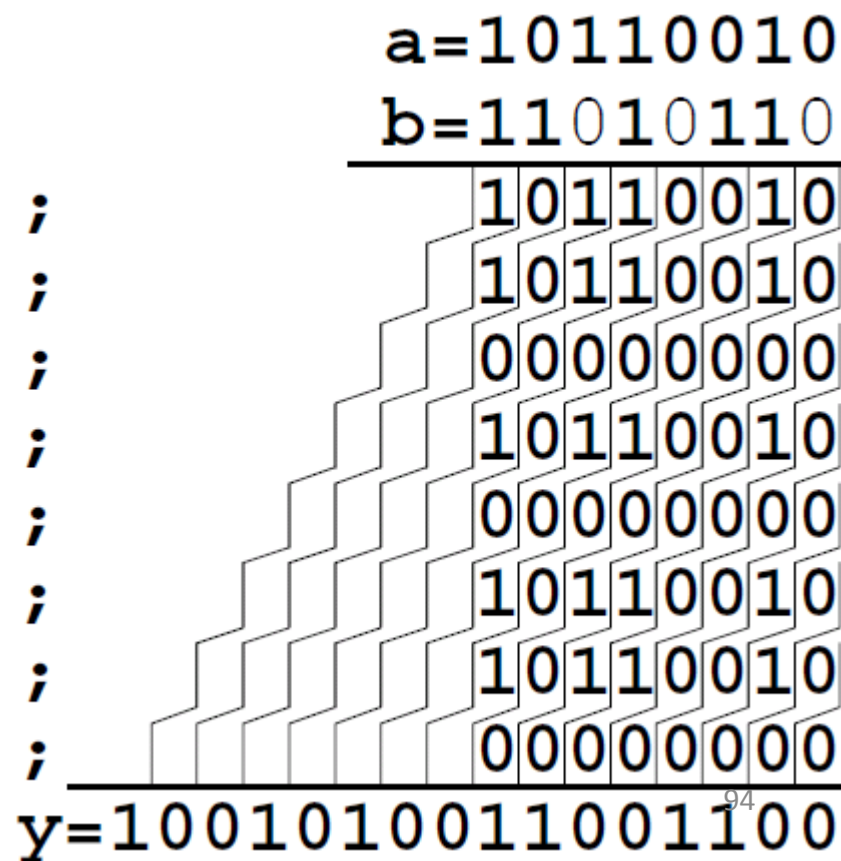
ck毎にstをインクリメント、以下のような演算を実行

st=7(演算終了)でfin=1とする

st=8でfin=0とする

```

start=1  y <= 0 ;
st=0     y <= (y<<1) + a*b[7] ;
st=1     y <= (y<<1) + a*b[6] ;
st=2     y <= (y<<1) + a*b[5] ;
st=3     y <= (y<<1) + a*b[4] ;
st=4     y <= (y<<1) + a*b[3] ;
st=5     y <= (y<<1) + a*b[2] ;
st=6     y <= (y<<1) + a*b[1] ;
st=7     y <= (y<<1) + a*b[0] ;
    
```



複数クロックでの実装

```
always @(posedge ck) begin  
    if( start == 1 ) begin
```

```
        end else begin
```

```
        end
```

```
    end
```

```
module mul(A,B,O,ck,start,fin);  
    input [7:0] A, B;  
    input ck,start;  
    output [16:0] O;  
    output fin;
```

変数(レジスタ等)の定義

実行

```
endmodule
```

multest.v

複数クロックでの実装(テストベンチ)

```
module multest;
    reg      [7:0]      A, B;
    reg      ck;
    reg      start;
    reg      [3:0] st;
    wire     [16:0] O;
    reg      [16:0] OR;
    initial begin
        ck=0;
        start=0;
        st=0;
        $monitor( "%t\tA=%h, B=%h, CTR=%h, (OUT=%h) OUT=%h", $time, A, B, CTR, O, OR );
        #1000 $finish;
    end

    mul      MUL(A , B , O , ck, start,fin);
    always   #10      ck = ~ck;
    always @(negedge ck) begin
        if( st == 0 ) start <= 1;
        else start <= 0;
        if( fin == 1 ) OR <= O;
        st <= st+1;
        A = $random;
        B = $random;
    end
endmodule
endmodule
```

全数チェックするにはmultest2.v (WEBから取得)

fifo.v 演習6

8ビット16段のFIFOを完成させよ(fifo.vをダウンロードして完成させ実行結果で確認)

モジュール名: fifo

入力: 8ビット入力データ: Din, クロック: ck, データ入力フラグ: Wen, データ出力フラグ: Ren
リセット: rst

出力: 8ビット出力データ: Dout, FIFOエンプティフラグ: Fempty, FIFOフルフラグ: Ffull

```
module fifo ( Din, Dout, Wen, Ren, rst, ck, Fempty, Ffull );
```

```
input [7:0] Din;
```

```
output [7:0] Dout;
```

```
input Wen, Ren, rst, ck;
```

```
output Fempty, Ffull;
```

```
reg [7:0] FMEM[0:15];
```

.....以下変更なし

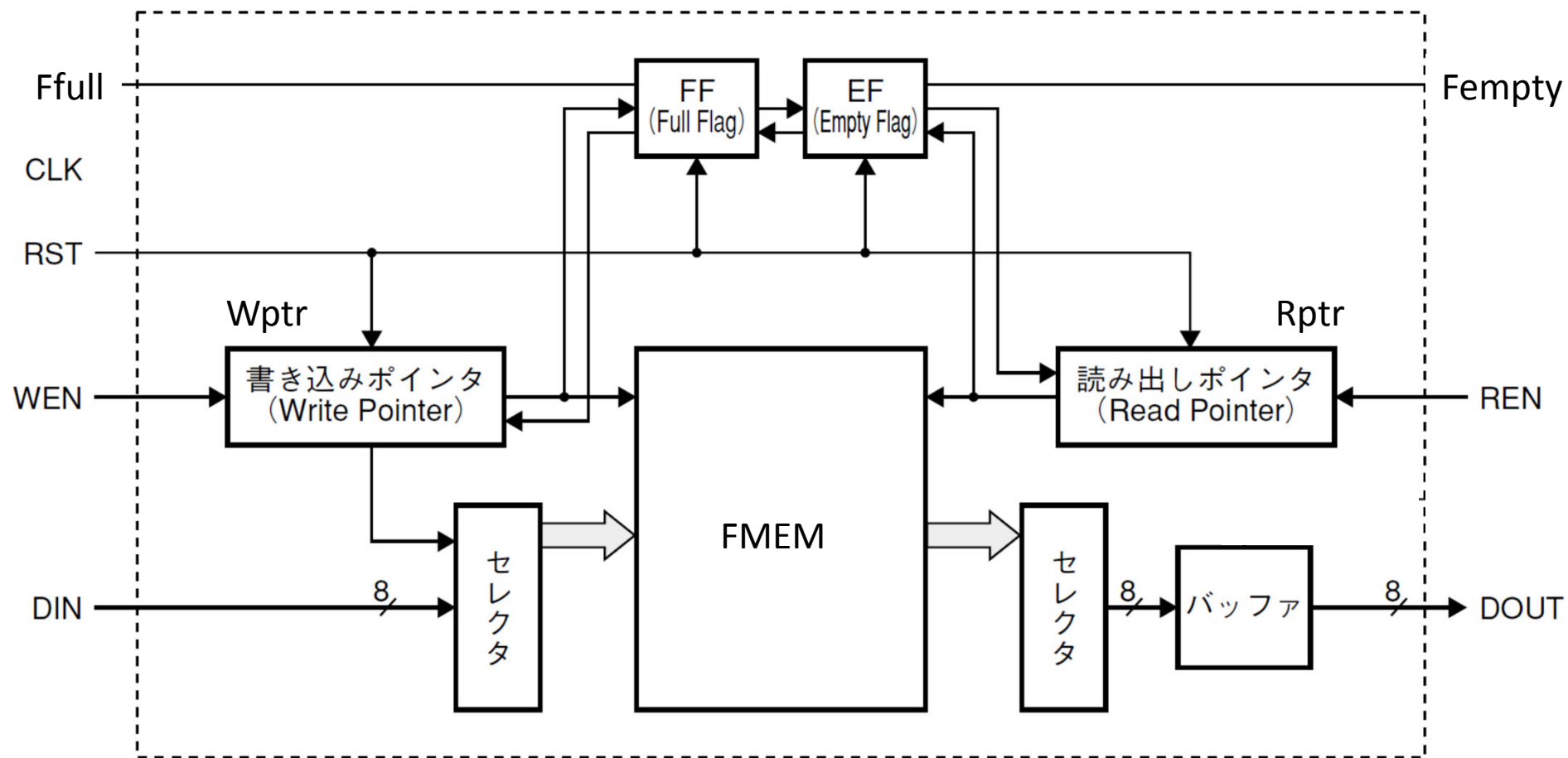
```
assign f0 = FMEM[0];
```

メモリーの内容は通常は参照することができない
→アドレス毎にassign文で切り分けてあげることシミュレーション中に参照(デバッグ)ができるようになる

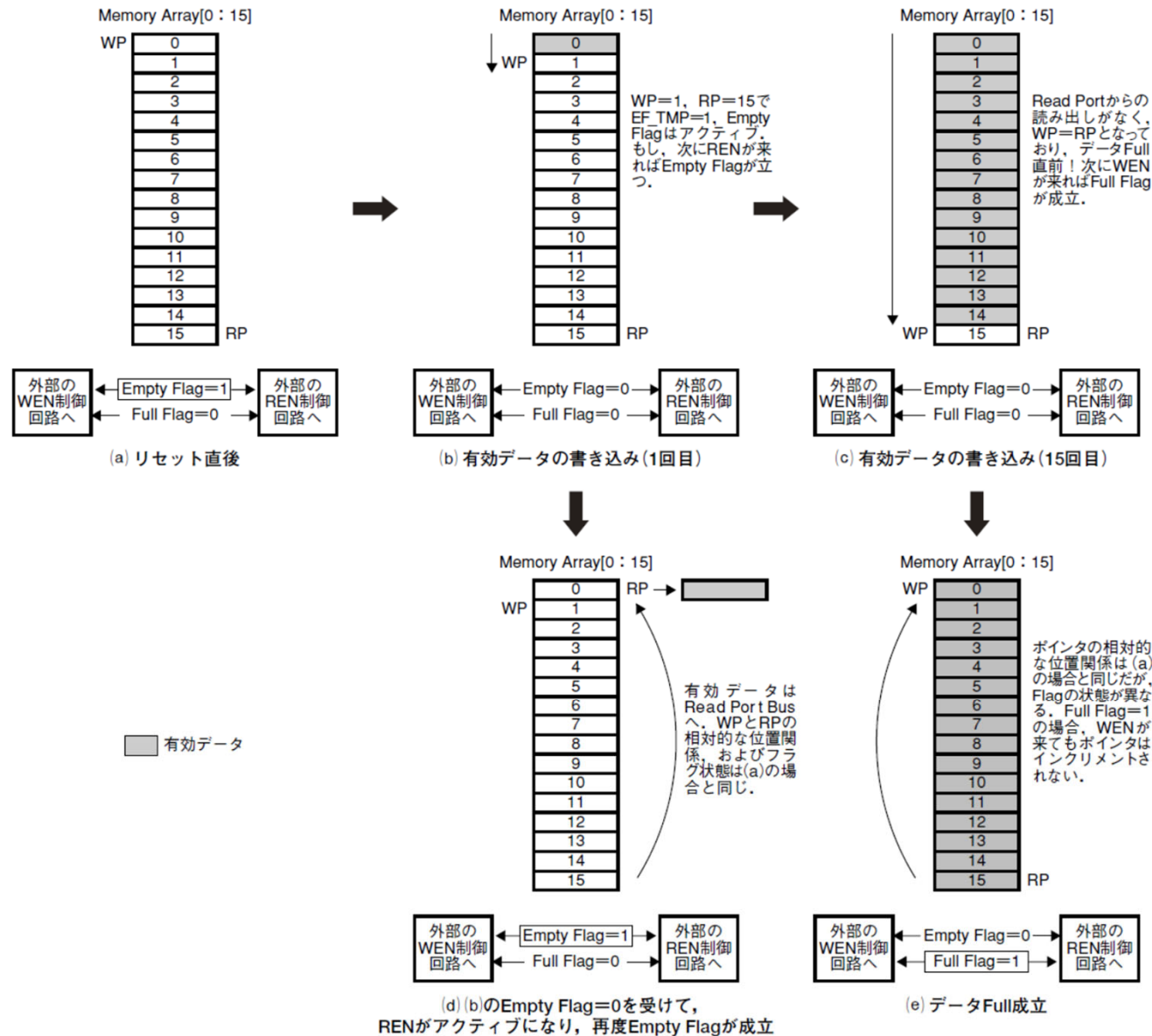
メモリの中身をシミュレーションで参照する仕組み

```
wire [7:0] f0, f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13, f14, f15;  
assign f0 = FMEM[0];  
assign f1 = FMEM[1];  
assign f2 = FMEM[2];  
assign f3 = FMEM[3];  
assign f4 = FMEM[4];  
assign f5 = FMEM[5];  
assign f6 = FMEM[6];  
assign f7 = FMEM[7];  
assign f8 = FMEM[8];  
assign f9 = FMEM[9];  
assign f10 = FMEM[10];  
assign f11 = FMEM[11];  
assign f12 = FMEM[12];  
assign f13 = FMEM[13];  
assign f14 = FMEM[14];  
assign f15 = FMEM[15];
```

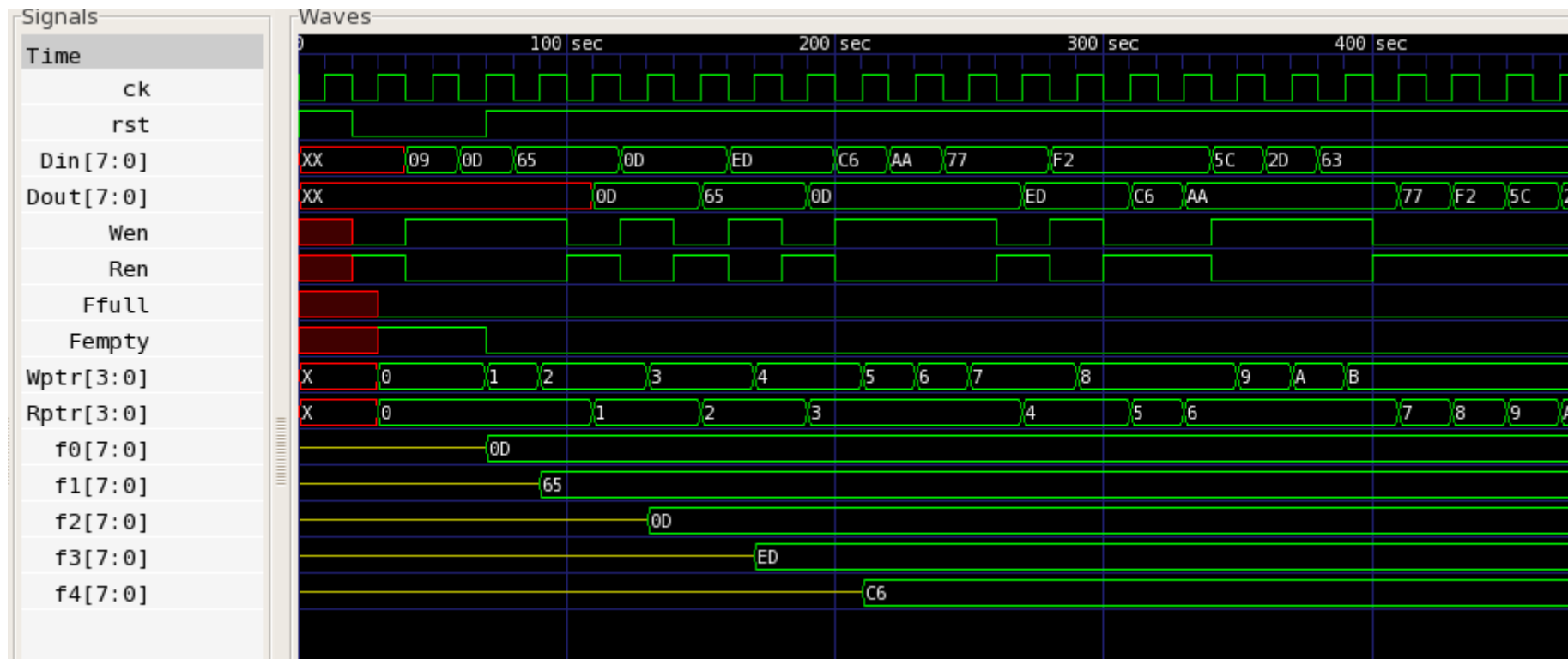
演習6:FIFO



FIFOの動作



演習6:FIFO



f0, f1,,,はFMEM[0..15]の内容を表す

Wen=1の時には、クロックごとにDinが内部に書き込まれる

同時に、Wptrがインクリメントされる

Wen=1の時には、クロックごとにFMEMの内容がDoutに出力される

同時に、Rptrがインクリメントされる

fifo.v 演習6 fifo.vの完成(simfifo.vを使用)

モジュール構成の理解

8ビット16段のFIFOを完成させよ(fifo.vをダウンロードして完成させ実行結果で確認)

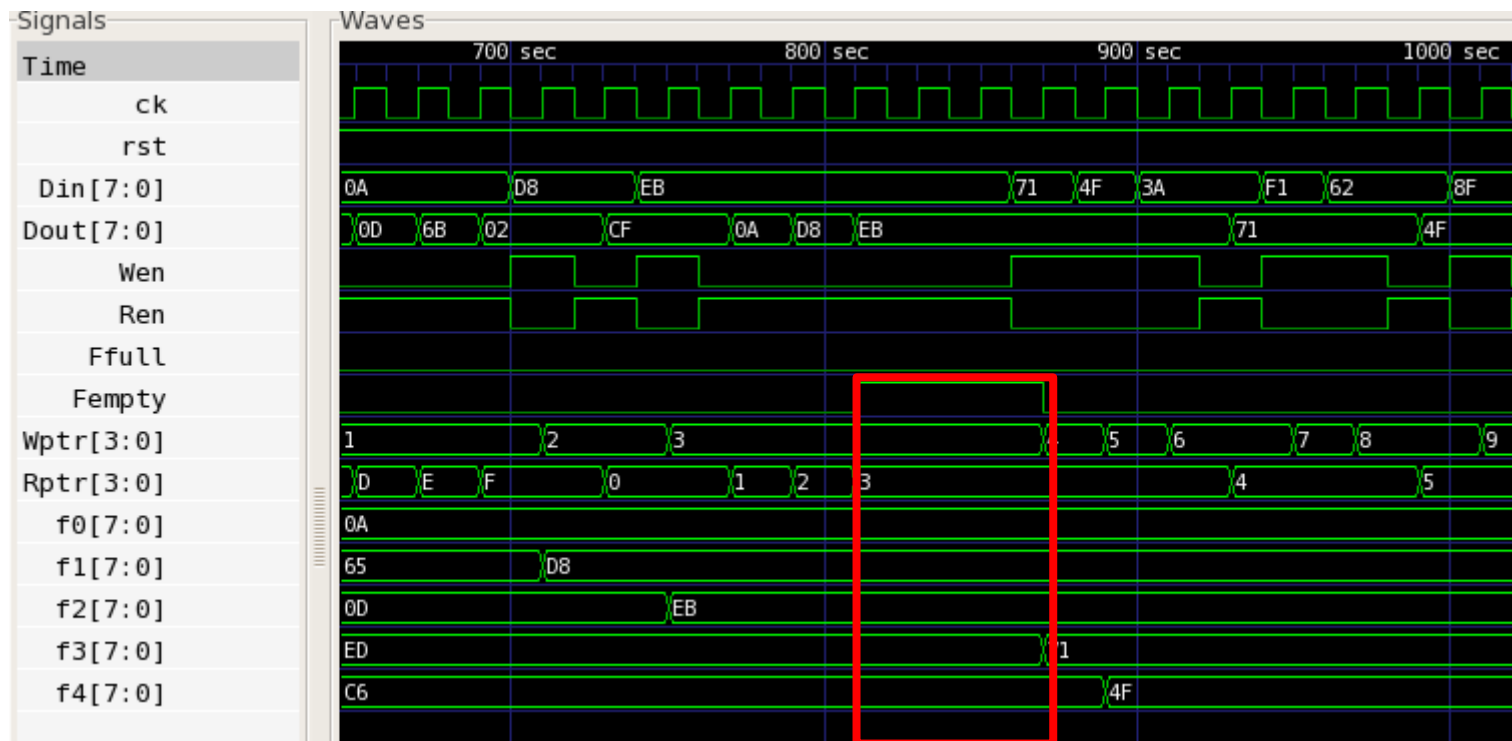
モジュール名: fifo

入力: 8ビット入力データ: Din, クロック: ck, データ入力フラグ: Wen, データ出力フラグ: Ren

リセット: rst

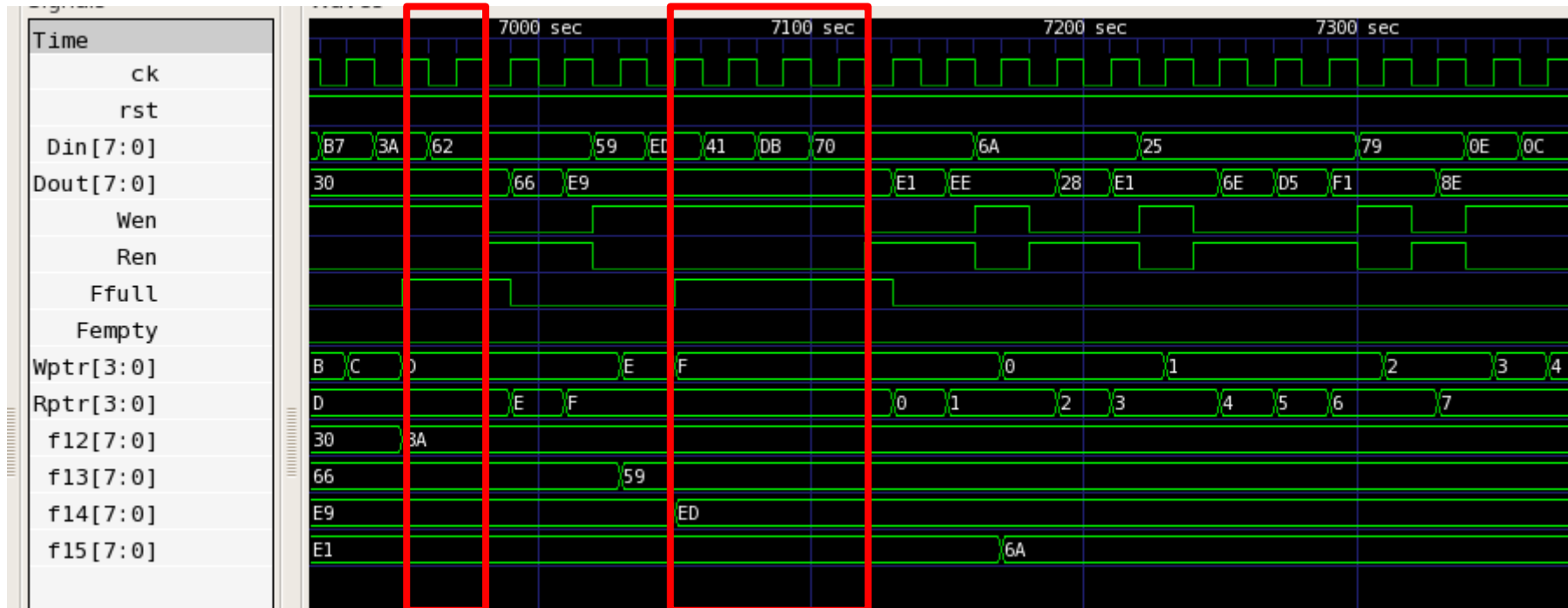
出力: 8ビット出力データ: Dout, FIFOエンプティフラグ: Fempty, FIFOフルフラグ: Ffull

FIFOの動作・・・1 (FIFO empty)



Ren=1の時に Wptr == Rptr (Fempty=1)だと、FIFOが空っぽであるため読み出しは行われない

FIFOの動作・・・1 (FIFO full)



Wen = 1 の時に Wptr == Rptr (Ffull=1)だと、FIFOがいっぱいであるため書き込みは行われない