

2020年5月1日

ハードウェア設計論:5

ハードウェアにおける設計表現・ ハードウェアの設計フロー

本日の課題

課題7－2～課題7－4までを、次回(5月8日朝)までに提出を終えておくこと

課題7－4 に関しては、
「cpu.v」と、「simcpu2.v以下の部分を切り出して」
hoge.vとして、**cpu.v, hoge.vの2ファイルをupload**

```
IMEM[0]='b  
IMEM[1]='b
```

.....
IMEM[**]='b

DMEM[0]='b 適当な値
DMEM[1]='b 適当な値

演習7－2：CPUの完成：cpu.v

```
module CPU(CK,RST,IA,ID,DA,DD,RW);
input CK,RST;
input [15:0] ID;
output RW;
output [15:0] IA,DA;
inout [15:0] DD;
reg [15:0] PC, INST, FUA,
    FUB, LSUA, LSUB, FUC, LSUC, PCC, PCI;
reg [15:0] RF[0:14];
reg [1:0] STAGE;
reg FLAG,RW;
wire [15:0] ABUS, BBUS, CBUS;
wire [3:0] OPCODE, OPR1, OPR2, OPR3;
wire [7:0] IMM;
wire [15:0] PCn;
assign PCn = PC + 1;
wire [15:0] RF01, RF05;
assign RF01 = RF[1];
assign RF05 = RF[5];
assign OPCODE = INST[15:12];
assign OPR1 = INST[11:8];
assign OPR2 = INST[7:4];
assign OPR3 = INST[3:0];
assign IMM = INST[7:0];
assign ABUS = (OPR2 == 0 ? 0 : RF[OPR2]);
assign BBUS = (OPR3 == 0 ? 0 : RF[OPR3]);
assign DA = LSUB;
assign IA = PC;
assign DD =((RW==0)? LSUA : 16'b Z);
assign CBUS = (OPCODE[3]==0 ? FUC :
(OPCODE[3:1]=='b 101 ? LSUC :
(OPCODE=='b 1100 ? {8'b 0,IMM} :
OPCODE=='b 1000 ? PCC : 'b z)));

```

```
always @(posedge CK) begin
if( RST == 1 ) begin
    PC <= 0;
    STAGE <= 0;
    RW<=1;
end else if( STAGE == 0 ) begin
    INST <= ID;
    STAGE <= 1;
end else if( STAGE == 1 ) begin
    if( (OPCODE[3:0] == 'b 1000) ||
        (OPCODE[3:0] == 'b 1001 && FLAG == 1 ) ) PCI <= BBUS;
    else PCI <= PCn;
    if( OPCODE[3] == 0 ) begin
    end else if( OPCODE[2:1] == 'b01) begin
    end
    STAGE <= 2;
end else if( STAGE == 2 ) begin
    if( OPCODE[3] == 0 ) begin
        case(OPCODE[2:0])
            'b 001: FUC<=FUA-FUB;
            'b 010: FUC<=FUA>>FUB;
            'b 011: FUC<=FUA<<FUB;
            'b 100: FUC<=FUA|FUB;
            'b 110: FUC<=~FUA;
            'b 111: FUC<=FUA^FUB;
        endcase
    end else if( OPCODE[3:1] == 'b 101 ) begin
        if( OPCODE[0] == 0 ) begin
        end else begin
        end
    end else if( OPCODE[3:0] == 'b 1000 ) PCC <= PCn;
    STAGE <= 3;
end
end else if( STAGE == 3 ) begin
    RW <= 1;
    if( OPCODE[3] == 0 ) begin
        if( CBUS == 0 ) FLAG <= 1;
        else FLAG <= 0;
    end
    STAGE <= 0;
end
end
endmodule
```

演習7-3：足し算のテストベンチ完成

```
module simcpu;
reg CK, RST;
wire RW;
wire [15:0] IA, DA, DD;
reg [15:0] ID, DDi;
reg [15:0] IMEM [0:127], DMEM[0:127];
CPU c(CK,RST,IA,ID,DA,DD,RW);
assign DD = ((RW == 1) ? DDi : 'b Z);
initial begin
    CK = 0;
    RST = 0;
#5  RST = 1;
#100 RST = 0;
#10000 $finish;
end
always @ (negedge CK) begin
    if( DA == 'b 0 && DD == 'b 0100 && RW == 0 ) begin
        $display( "OK" );
        $finish;
    end
end
always @ (negedge CK) begin
    ID = IMEM[IA];
end
always @ (negedge CK) begin
    if( RW == 1 ) DDi = DMEM[DA];
    else DMEM[DA] = DD;
end
```

```
initial begin
    IMEM[0]='b 1100_0001_0000_0000; // IMM R1, [0]
    IEM[1]='b 1100_0010_0000_0001; // IMM R2, [1]
    IMEM[2]='b 1100_0011_0000_0001; // IMM R3, [1]
    IMEM[3]='b 1100_0100_0000_1001; // IMM R4, [9]
    IMEM[4]='b 1100_0101_0000_1100; // IMM R5, [12]
    IMEM[5]='b 1100_0110_0000_0111; // IMM R6, [7]
    IMEM[6]='b 1100_0111_0000_0000; // IMM R7, [0]
    IMEM[7]='b 0000_0001_0001_0010; // ADD R1, R1, R2
    IMEM[8]='b 0000_0010_0010_0011; // ADD R2, R2, R3
    IMEM[9]='b 0001_0100_0100_0011; // SUB R0, R4, R3
    IMEM[10]='b 1001_0000_0000_0101; // BR f=0, R5
    IMEM[11]='b 1000_0000_0000_0110; // JMP R0, R6
    IMEM[12]='b 1010_0000_0001_0111; // ST R1, R7
end
always #10 CK = ~CK;
endmodule
```

演習7－4: 乗算のテストベンチ完成

```
initial begin
    DMEM[0]= 5;
    DMEM[1]= 50;
    //
    IMEM[0]='b 1100_0001_0000_0000; // IMM R1, [0]
    IMEM[1]='b 1100_0100_0000_1111; // IMM R4, [15]
    IMEM[2]='b 1100_0101_0000_0001; // IMM R5, [1]
    IMEM[3]='b 0011_0101_0101_0100; // SHL R5, R5,R4
    IMEM[4]='b 1100_0100_0000_0001; // IMM R4, [1]
    IMEM[5]='b 1100_1001_0000_0000; // IMM R9, [0]
    IMEM[6]='b 1100_1010_0000_0001; // IMM R10, [1]
    IMEM[7]='b 1100_1011_0000_0010; // IMM R11, [2]
    IMEM[8]='b 1100_0110_0001_0001; // IMM R6, [17]
    IMEM[9]='b 1100_0111_0001_0100; // IMM R7, [20]
    IMEM[10]='b 1100_1000_0000_1101; // IMM R8, [13]
    IMEM[11]='b 1011_0010_0000_1001; // LD R2, R9
    IMEM[12]='b 1011_0011_0000_1010; // LD R3, R10
    IMEM[13]='b 0011_0001_0001_0100; // SHL R1, R1,R4
    IMEM[14]='b 0101_0000_0101_0011; // AND R0,R5,R3
    IMEM[15]='b 1001_0000_0000_0110; // BR f=0, R6
    IMEM[16]='b 0000_0001_0001_0010; // ADD R1, R1, R2
    IMEM[17]='b 0010_0101_0101_0100; // SHR R5, R5, R4
    IMEM[18]='b 1001_0000_0000_0111; // BR f=0, R7
    IMEM[19]='b 1000_0000_0000_1000; // JMP R0, R8
    IMEM[20]='b 1010_0000_0001_1011; // ST R1, R11
end
always #10 CK = ~CK;
endmodule
```

演習7-4 : CPUで乗算の実行

- データメモリ0番地のデータと1番地のデータを掛け算して2番地に格納

- R1: アキュムレータ
- R2: 被乗数
- R3: 加数(1)
- R4: 乗数
- R5: ジャンプアドレス(13)
- R6: ジャンプアドレス(9)
- R7: 入力1メモリアドレス(0)
- R8: 入力2メモリアドレス(1)
- R9: 出力メモリアドレス(2)

```
R1=0;
for (R4; R4>=0; R4--){
    R1 += R2;
}
```

```
IMEM[0]='b 1100_0001_0000_0000; // IMM R1, [0]
IMEM[1]='b 1100_0111_0000_0000; // IMM R7, [0]
IMEM[2]='b 1100_1000_0000_0001; // IMM R8, [1]
IMEM[3]='b 1100_1001_0000_0010; // IMM R9, [2]
IMEM[4]='b 1100_0011_0000_0001; // IMM R3, [1]
IMEM[5]='b 1100_0101_0000_1101; // IMM R5, [13]
IMEM[6]='b 1100_0110_0000_1001; // IMM R6, [9]
IMEM[7]='b 1011_0010_0000_0111; // LD R2, R7
IMEM[8]='b 1011_0100_0000_1000; // LD R4, R8
IMEM[9]='b 0000_0001_0001_0010; // ADD R1, R1, R2
IMEM[10]='b 0001_0100_0100_0011; // SUB R4, R4, R3
IMEM[11]='b 1001_0000_0000_0101; // BR f=0, R5
IMEM[12]='b 1000_0000_0000_0110; // JMP R0, R6
IMEM[13]='b 1010_0000_0001_1001; // ST R1, R9
```

R2をR4回加算することで乗算を実現

演習7－4：CPUで乗算の実行‘

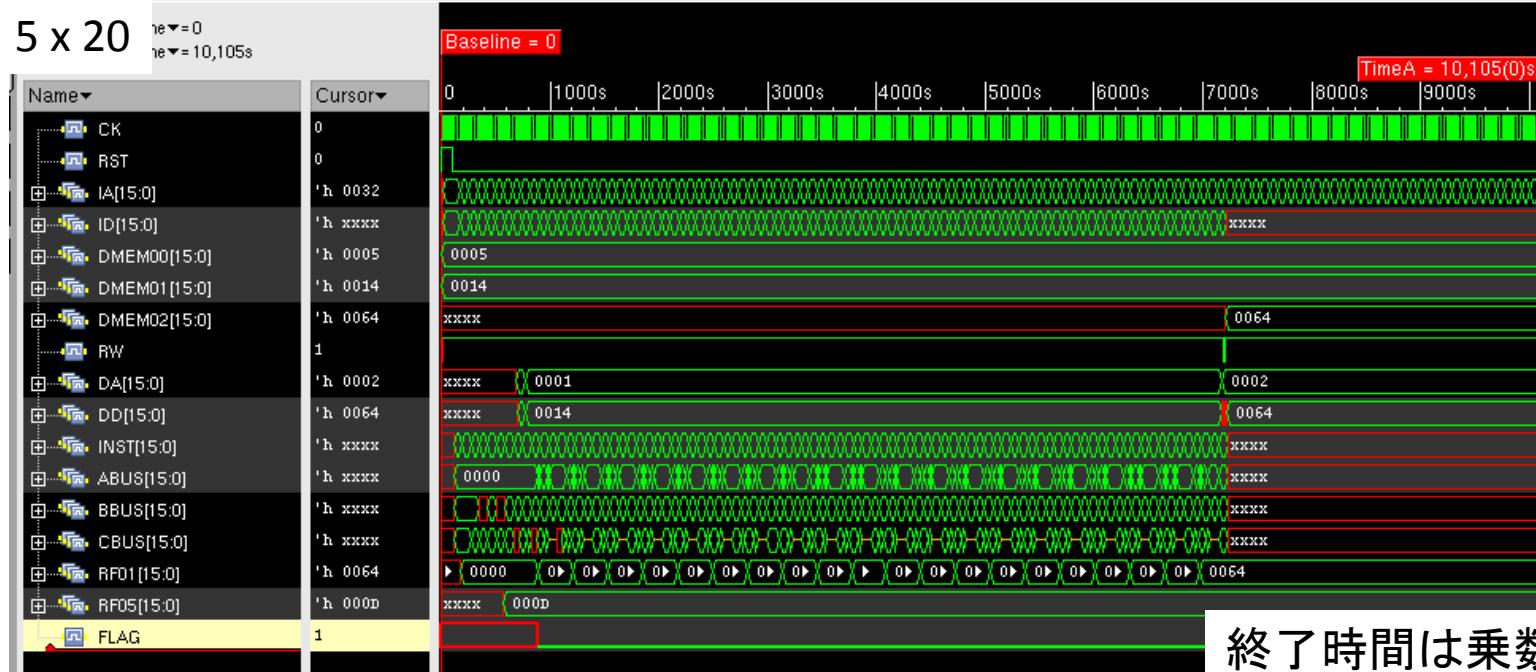
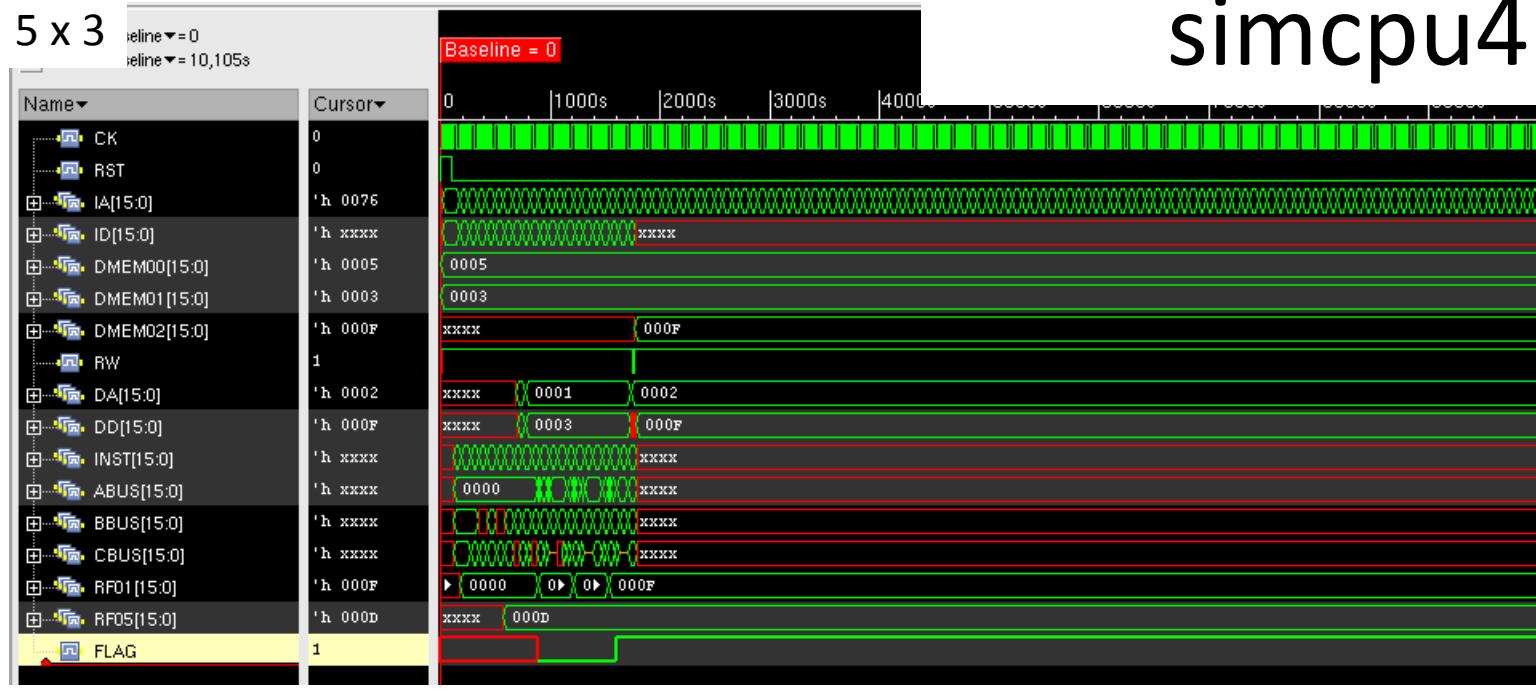
- R1: アキュムレータ
- R2: 被乗数
- R3: 乗数
- R4: 定数(1)
- R5: マスク('h 8000)
- R6: ジャンプアドレス(17)
- R7: ジャンプアドレス(20)
- R8: ジャンプアドレス(13)
- R9: 入力1メモリアドレス(0)
- R10: 入力2メモリアドレス(1)
- R11: 出力メモリアドレス(2)

```
R1=0;
for (R5='8000; R5>=0; R5>>=1){
    R1 <= 1;
    if( R3 & R5 ) R1 += R2;
}
```

```
IMEM[0]='b 1100_0001_0000_0000; // IMM R1, [0]
IMEM[1]='b 1100_0100_0000_1111; // IMM R4, [15]
IMEM[2]='b 1100_0101_0000_0001; // IMM R5, [1]
IMEM[3]='b 0011_0101_0101_0100; // SHL R5, R5,R4
IMEM[4]='b 1100_0100_0000_0001; // IMM R4, [1]
IMEM[5]='b 1100_1001_0000_0000; // IMM R9, [0]
IMEM[6]='b 1100_1010_0000_0001; // IMM R10, [1]
IMEM[7]='b 1100_1011_0000_0010; // IMM R11, [2]
IMEM[8]='b 1100_0110_0001_0001; // IMM R6, [17]
IMEM[9]='b 1100_0111_0001_0100; // IMM R7, [20]
IMEM[10]='b 1100_1000_0000_1101; // IMM R8, [13]
IMEM[11]='b 1011_0010_0000_1001; // LD R2, R9
IMEM[12]='b 1011_0011_0000_1010; // LD R3, R10
IMEM[13]='b 0011_0001_0001_0100; // SHL R1, R1,R4
IMEM[14]='b 0101_0000_0101_0011; // AND R0,R5,R3
IMEM[15]='b 1001_0000_0000_0110; // BR f=0, R6
IMEM[16]='b 0000_0001_0001_0010; // ADD R1, R1, R2
IMEM[17]='b 0010_0101_0101_0100; // SHR R5, R5, R4
IMEM[18]='b 1001_0000_0000_0111; // BR f=0, R7
IMEM[19]='b 1000_0000_0000_1000; // JMP R0, R8
IMEM[20]='b 1010_0000_0001_1011; // ST R1, R11
```

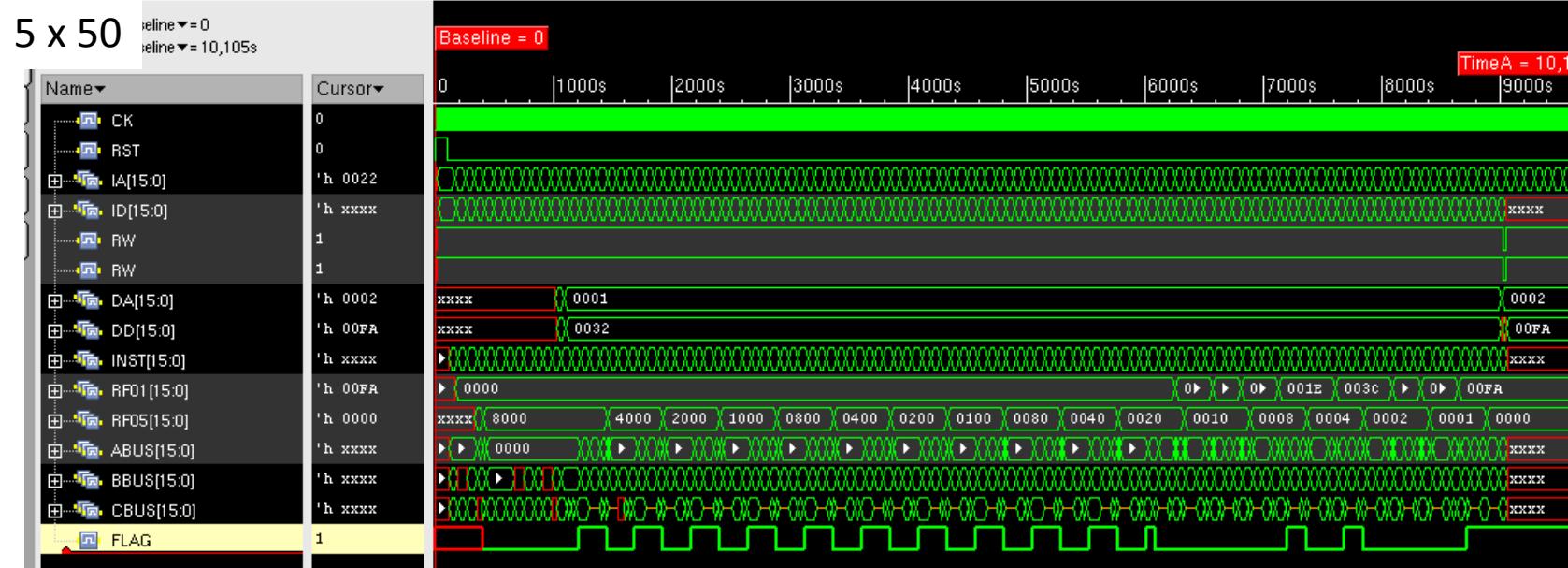
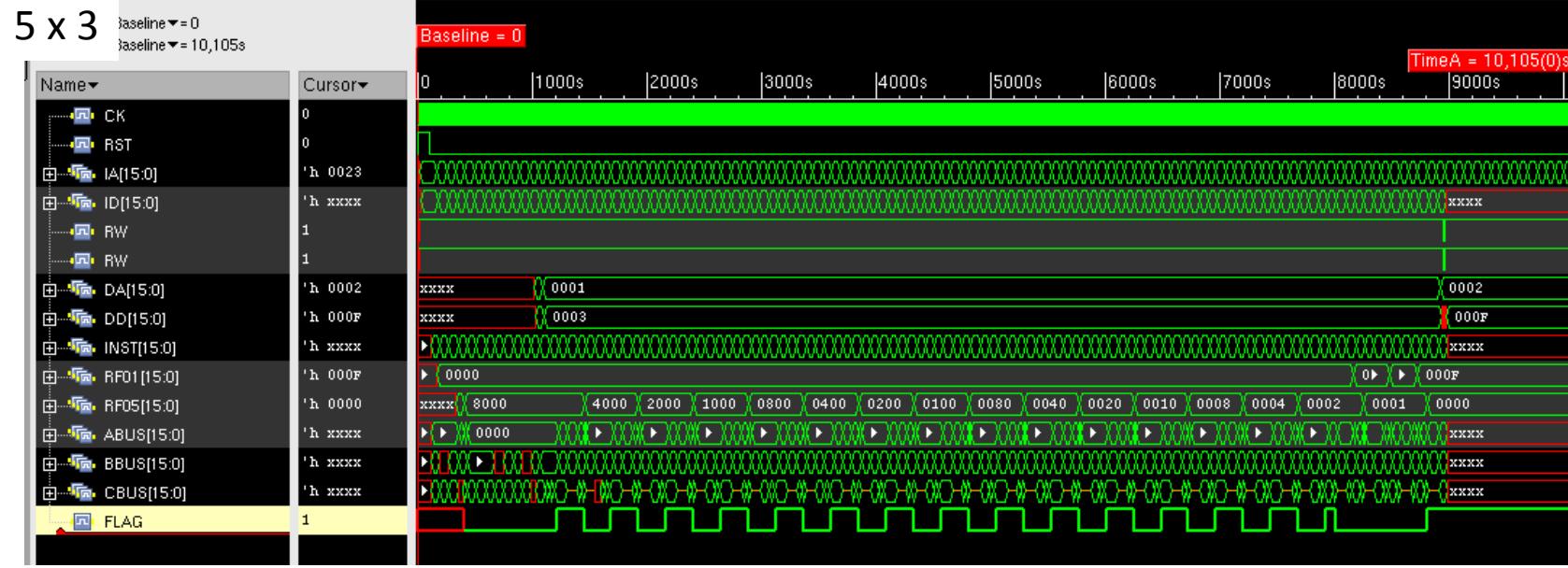
₁₃₆

simcpu4.v



終了時間は乗数により変わる 137

simcpu41.v



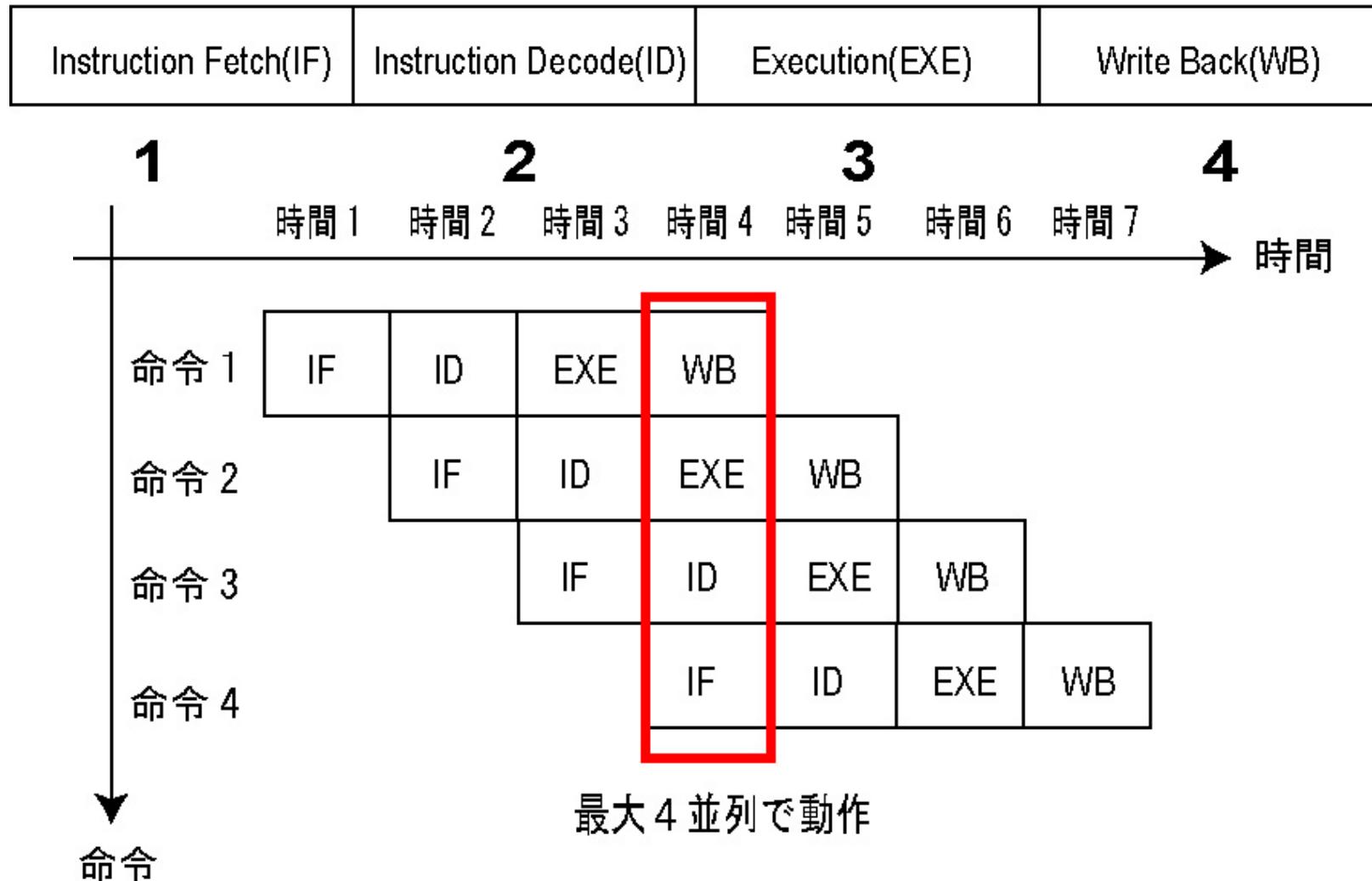
138 終了時間はほぼ同じ

演習7-5: 発展課題: CPUのパイプライン化

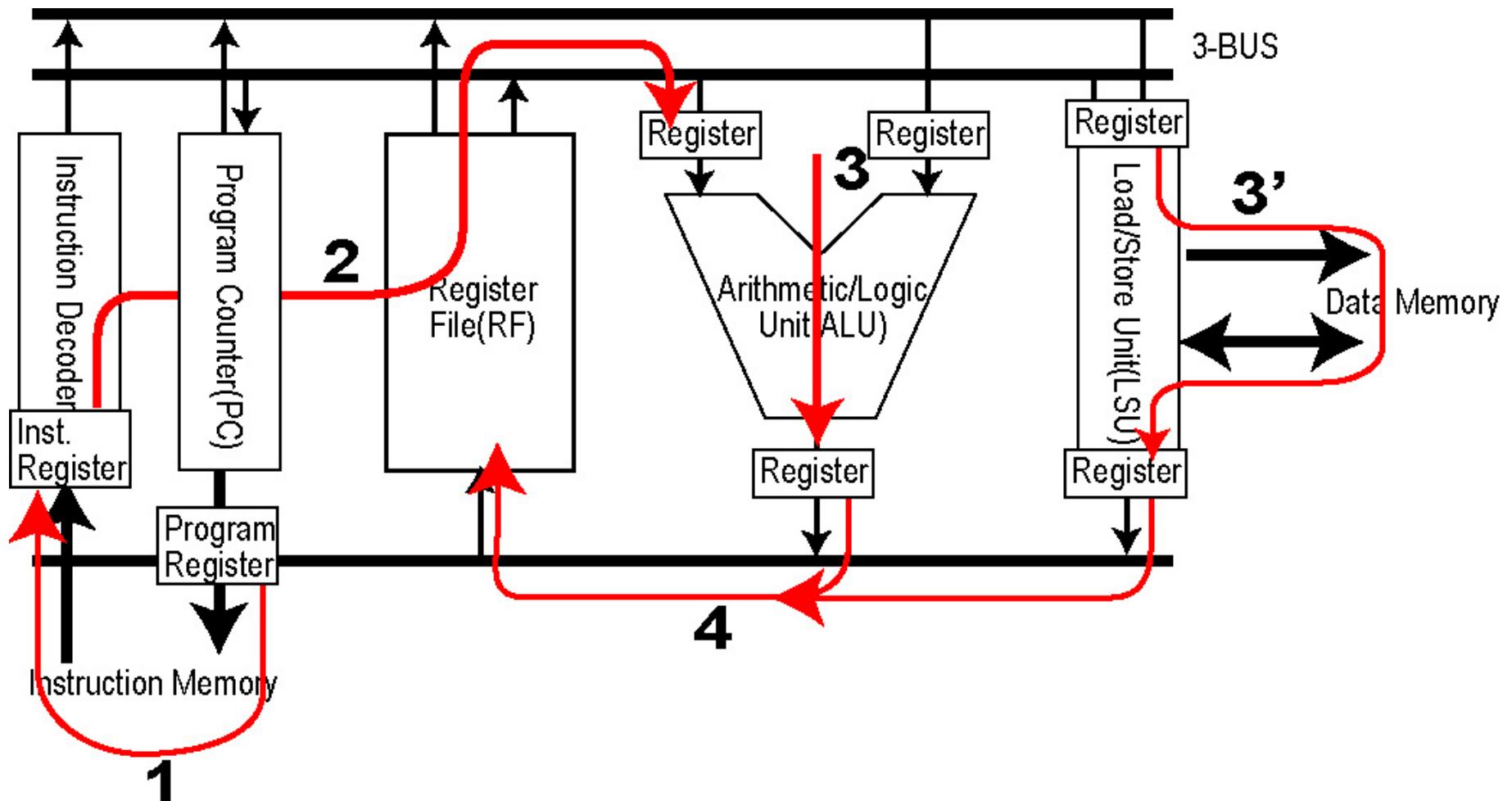
- ・パイプライン化について検討する

パイプラインとは

パイプラインステージ構成



単純パイプラインの動作



パイプライン動作の問題点

- パイプラインストール
 - データハザードによるストール
 - データフォワーディング、レジスタリネーミング、命令並び替え
 - ジャンプ・分岐によるストール(遅延分岐)
 - 命令並び替え、分岐予測
 - メモリ入出力時間遅延に伴うストール
 - キヤッシュ、演算終了時間制御

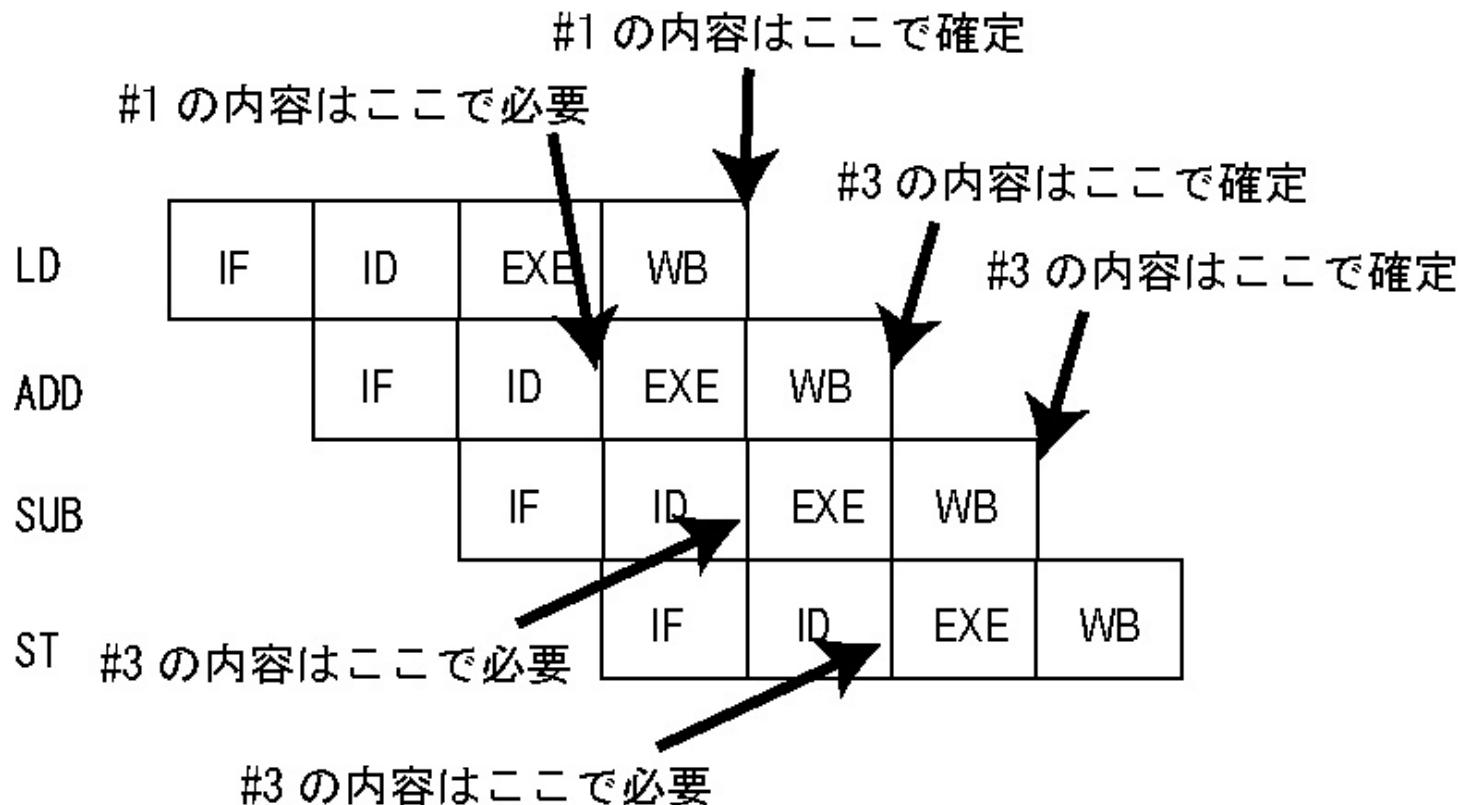
データハザードとは

LD #1, [10] : レジスタ1に10番地の内容を読み込み

ADD #3, #1, #3 : レジスタ1, 2を加算し3に書き込む

SUB #3, #3, #4 : レジスタ3, 4を減算し3に書き込む

ST #3, [11] : レジスター3を11番地に書き出す



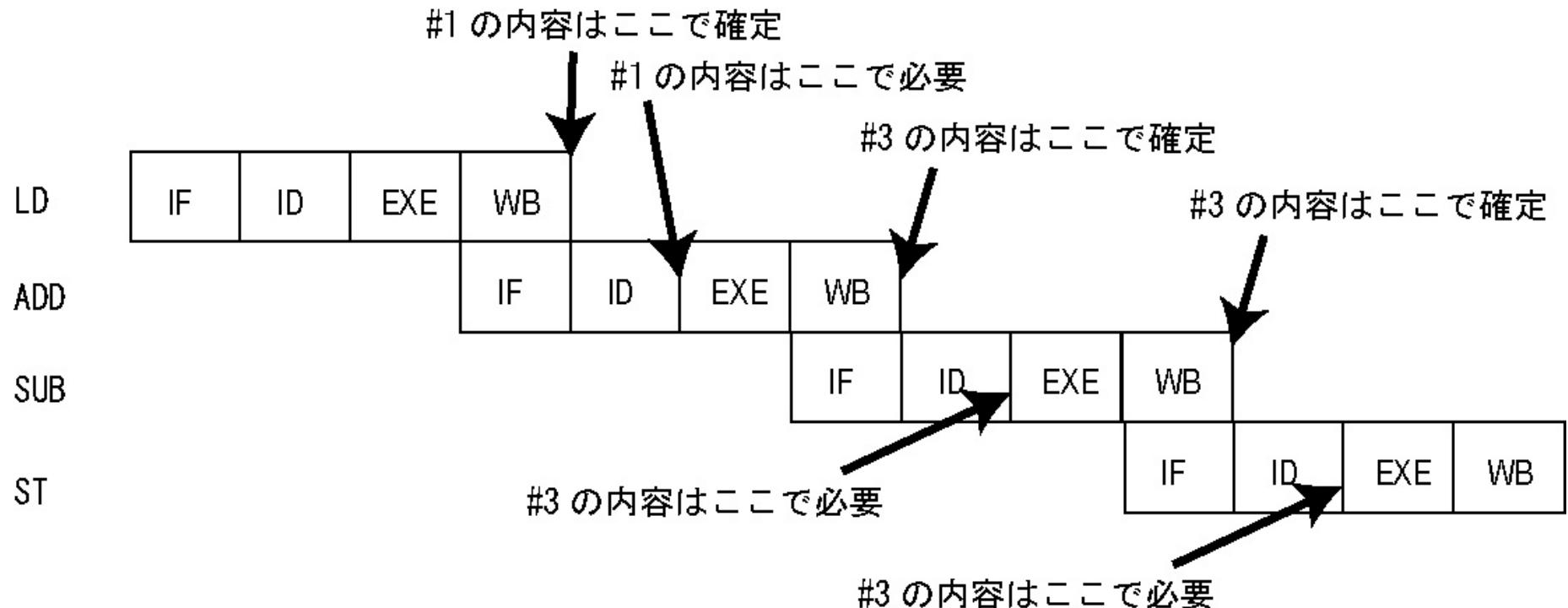
データハザードの解決

LD #1, [10] : レジスタ1に10番地の内容を読み込み

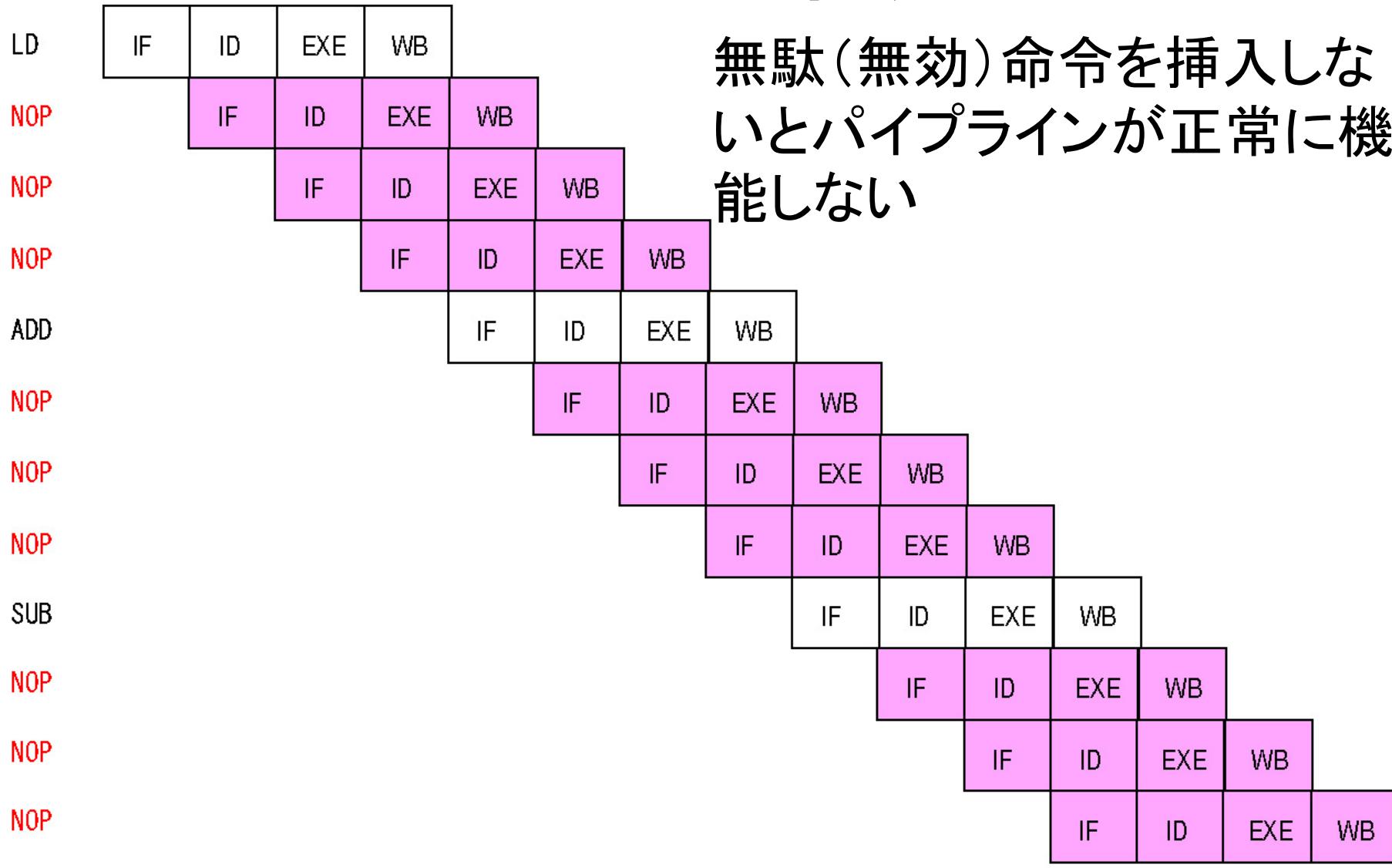
ADD #3, #1, #3 : レジスタ1, 2を加算し3に書き込む

SUB #3, #3, #4 : レジスタ3, 4を減算し3に書き込む

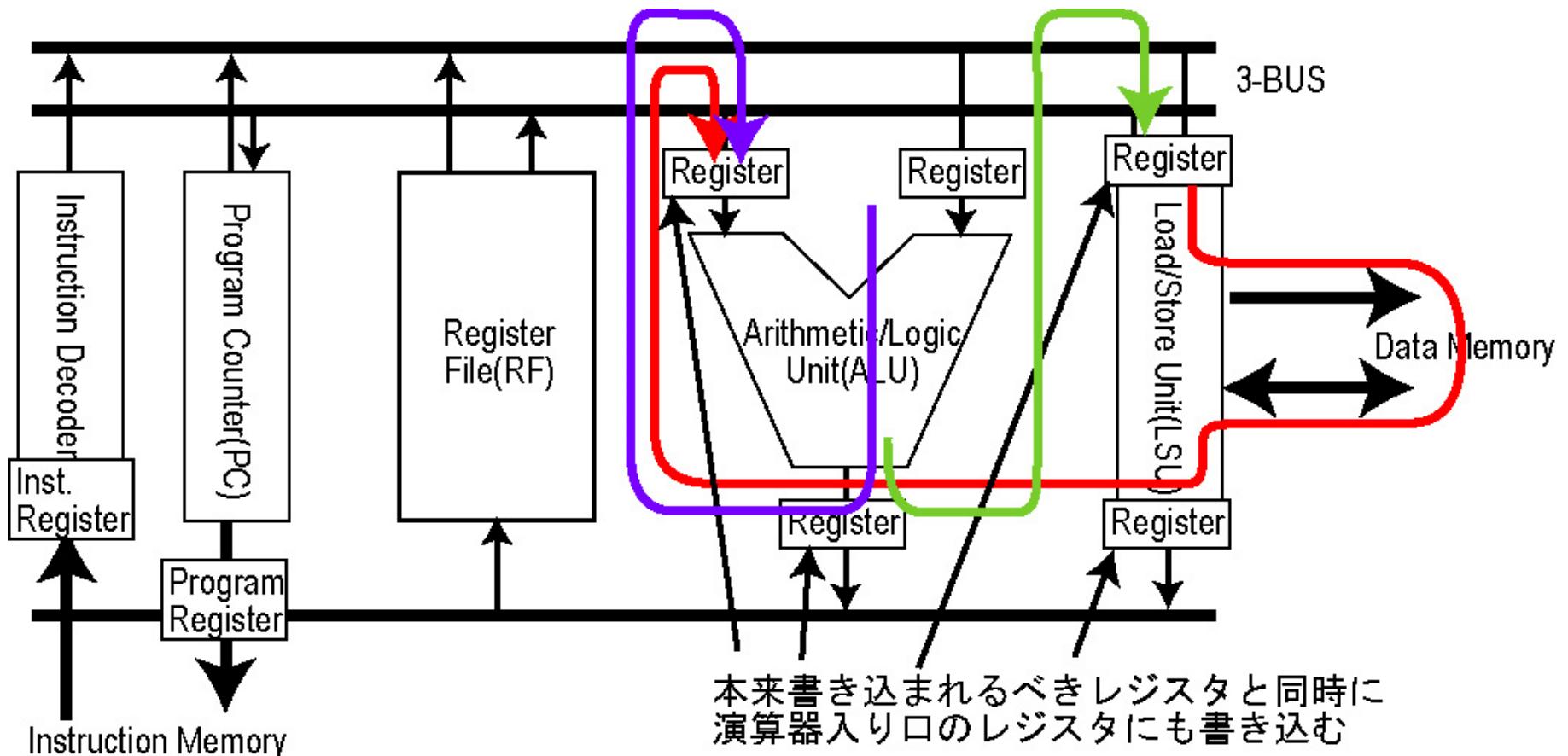
ST #3, [11] : レジス3を11番地に書き出す



データハザードの解決・・つまり



データフォワーディング



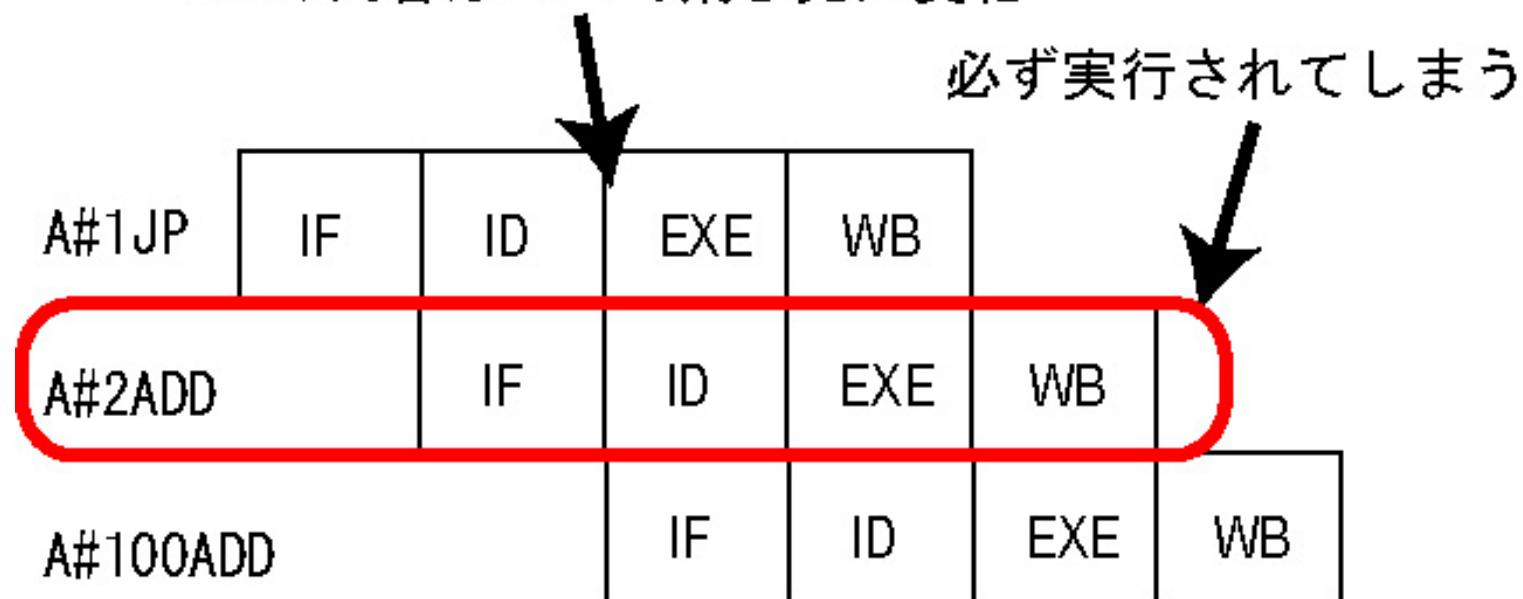
遅延分岐によるストール

A#1 JP A#100 : 100番地にジャンプ

A#2 ADD #3, #1, #2 : レジスタ1, 2を加算し3に書き込む

A#100 ADD #3, #2, #3 : レジスタ2, 3を加算し3に書き込む

PC の内容はここで飛び先に変化



パイプライン化

各状態間ですべての値を複
製し受け渡す

reg A;



reg A_st0, A_st1, A_st2, A_st3;

```
always @(posedge CK) begin
    if( RST == 1 ) begin
        ←
    end else begin
        if( STAGE == 0 )begin
            ←
            STAGE <= 1;           A_st0 <= *****
        end
        ←
        end else if( STAGE == 1 ) begin
            ←
            STAGE <= 2;           A_st1 <= A_st0
        end
        ←
        end else if( STAGE == 2) begin
            ←
            STAGE <= 3;           A_st2 <= A_st1
        end
        ←
        end else if( STAGE == 3 ) begin
            ←
            STAGE <= 0;           A_st3 <= A_st2
        end
    end
end
endmodule
```

パイプライン化したCPU・・・1

```

assign OPCODE = INST[15:12];           IFステージでのオペコード
assign OPCODE1 = INST_1[15:12];        IDステージでのオペコード
assign OPCODE2 = INST_2[15:12];        EXEステージでのオペコード
assign OPCODE3 = INST_3[15:12];        WBステージでのオペコード
assign OPR1 = INST_3[11:8]; // For Reg File Write Back at Stage==3
assign OPR12 = INST_2[11:8];// For Reg File Write Back at Stage==3
assign OPR2 = INST_1[7:4]; // For Reg File Read at Stage==1
assign OPR3 = INST_1[3:0]; // For Reg File Read at Stage==1
assign OPR3 = INST_1[3:0]; // For Reg File Read at Stage==1

```

```

assign ALU=      (OPCODE2[2:0] == 'b 000 ? FUA + FUB :
                  (OPCODE2[2:0] == 'b 001 ? FUA - FUB :
                  (OPCODE2[2:0] == 'b 010 ? FUA >> FUB :
                  (OPCODE2[2:0] == 'b 011 ? FUA << FUB :
                  (OPCODE2[2:0] == 'b 100 ? FUA | FUB :
                  (OPCODE2[2:0] == 'b 101 ? FUA & FUB :
                  (OPCODE2[2:0] == 'b 110 ? ~FUA :
                  (OPCODE2[2:0] == 'b 111 ? FUA ^ FUB : 'h z ))))))));
assign ALUz = (ALU == 0 ? 1 : 0 );

```

パイプライン化したCPU・・2

```

always @(posedge CK) begin
    if( RST == 1 ) begin
        PC <= 0;  RW<=1;
    end else begin
        INST <= ID;  INST_1 <= INST;  INST_2 <= INST_1;           INST_3 <= INST_2;
        if( (OPCODE1[3:0] == 'b 1000) || (OPCODE1[3:0] == 'b 1001 && FLAG == 1 ) ) PC <= BBUS;
        else PC <= PCn;
        if( OPCODE1[3] == 0 ) begin
            FUA <= FUAI; FUB <= FUBI;
        end else if( OPCODE1[2:1] == 'b01) begin
            LSUA <= FUAI; LSUB <= FUBI;
        end
        if( OPCODE2[3] == 0 ) begin
            FUC <= ALU;  FUCz <= ALUz;
        end else if( OPCODE2[3:1] == 'b 101 )
            if( OPCODE2[0] == 0 ) RW <= 0;
            else begin
                RW <= 1;  LSUC <= DD;
            end
        end
        RF[OPR1] <= CBUS;
        if( OPCODE3[3] == 0 ) FLAG <= FUCz;
    end
end

```

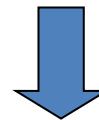
パイプライン化したCPU・・3

データフォワード

```
assign FUAin = ((OPCODE2[3]==0 && OPR12 == OPR2) ? ALU :  
                ( (OPCODE2 == 'b 1011 && OPR12 == OPR2) ? DD :  
                  ( ((OPCODE3[3]==0 || OPCODE3 == 'b 1011) && OPR1 == OPR2) ?  
                    CBUS : ABUS ) );  
assign FUBin = ((OPCODE2[3]==0 && OPR12 == OPR3) ? ALU :  
                ( (OPCODE2 == 'b 1011 && OPR12 == OPR3) ? DD :  
                  ( ((OPCODE3[3]==0 || OPCODE3 == 'b 1011) && OPR1 == OPR3) ?  
                    CBUS : BBUS ) );
```

LSIと設計

設計記述言語



論理式

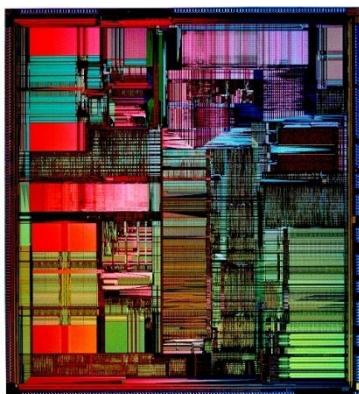
$$f = \overline{A \cdot B} + \overline{C} \otimes D$$

回路

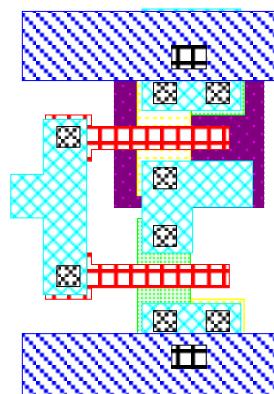
```
module add4(s,a,b);  
  output [4:0] s;  
  input  [3:0] a,b;  
  assign s=a+b;  
endmodule
```

ロジック

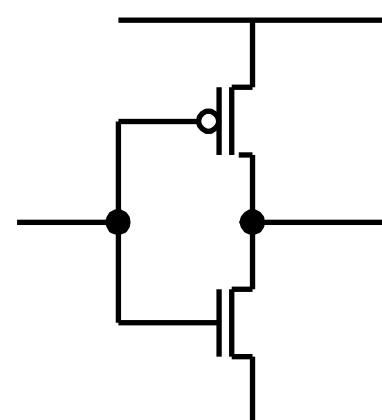
LSIチップ



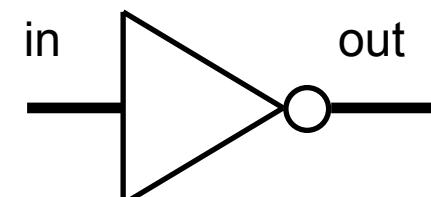
レイアウト



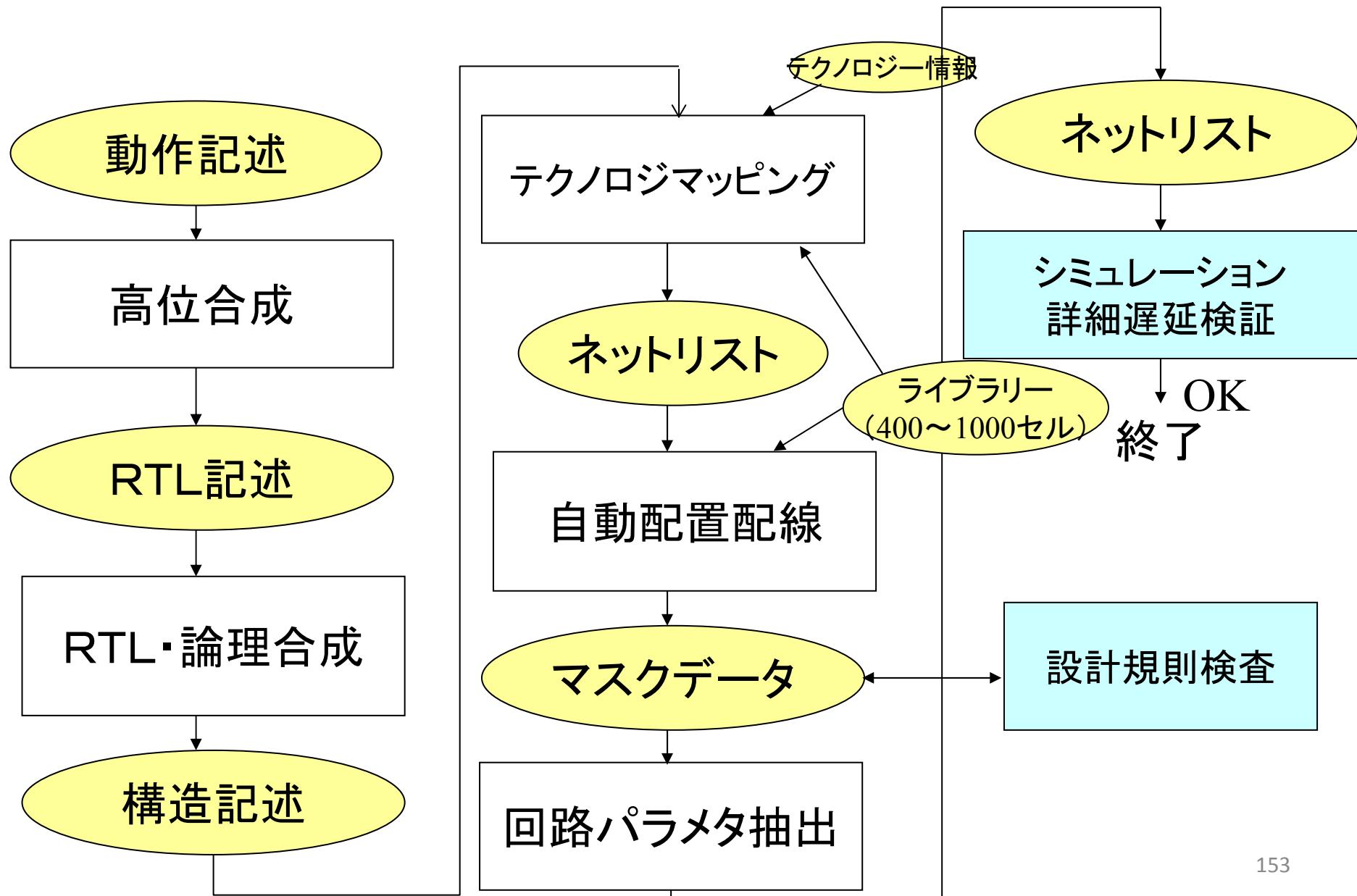
回路レベル
ネットリスト



論理レベル
ネットリスト



デジタルLSIの設計フロー



実際には ネットリスト

RTL

```
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
        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
end
endmodule
```



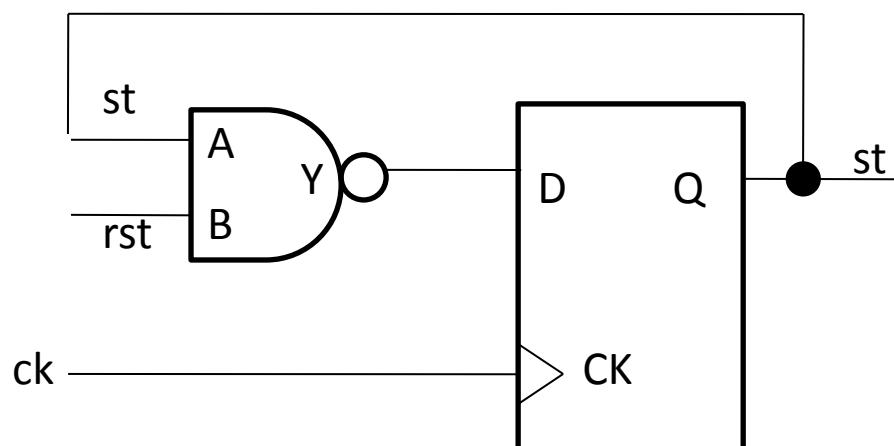
```
module stm ( ck, rst, x );
output [3:0] x;
input ck, rst;
wire n27, n28, n30, net1992,
WDFXP x_reg_3_ (.D(n76), .C(ck), .XQ(net2135));
DFXP x_reg_0_ (.D(n30), .C(ck), .XQ(net2120));
DFXP st_reg (.D(n77), .C(ck), .XQ(net2127));
DFXP x_reg_2_ (.D(n28), .C(ck), .XQ(net2139));
DFXP x_reg_1_ (.D(n27), .C(ck), .XQ(net2124));
INV2 U32 (.A(net2128), .Y(n31));
INV4 U33 (.A(net2127), .Y(net2128));
INV1 U34 (.A(n53), .Y(n32));
INV1 U35 (.A(x[0]), .Y(n33));
INV4 U36 (.A(net2120), .Y(n53));
BUF4 U37 (.A(n53), .Y(x[0]));
INV1 U38 (.A(net2139), .Y(net2140));
```

状態機械の合成

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

```
module stm ( ck, rst, st );
    input ck, rst;
    output st;
    wire N0;

    NOR2P U3 ( .A(st), .B(rst), .Y(N0) );
    DF st_reg ( .D(N0), .C(ck), .Q(st) );
endmodule
```



組み合わせ論理の簡単化(最適化)

$$\begin{aligned}T_1 &= \overline{S_1 S_2 X} \cup \overline{S_1} S_2 \overline{X} \cup S_1 \overline{S_2} \overline{X} \cup S_1 S_2 \overline{X} \cup S_1 S_2 X \\&= (\overline{S_1 S_2} \cup S_1 \overline{S_2} \cup \overline{S_1} S_2 \cup S_1 S_2) \overline{X} \cup S_1 S_2 X \\&= \overline{X} \cup S_1 S_2 X \\&= \boxed{\overline{X} \cup S_1 S_2}\end{aligned}$$

	S_1		\overline{S}_1
X	0	1	0
\overline{X}	1	1	1
	\overline{S}_2	S_2	\overline{S}_2

合成してみると…

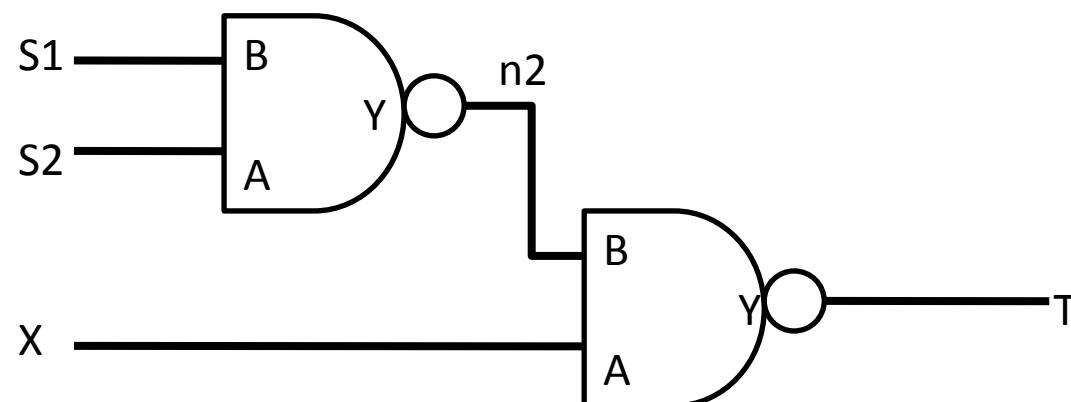
```
module logic ( T, S1, S2, X );
  input S1, S2, X;
  output T;
  assign T = ~X | S1&S2;
endmodule
```



論理合成

```
module logic ( T, S1, S2, X );
  input S1, S2, X;
  output T;
  wire n2;

  NAND2 U3 ( .A(X), .B(n2), .Y(T) );
  NAND2 U4 ( .A(S2), .B(S1), .Y(n2) );
endmodule
```

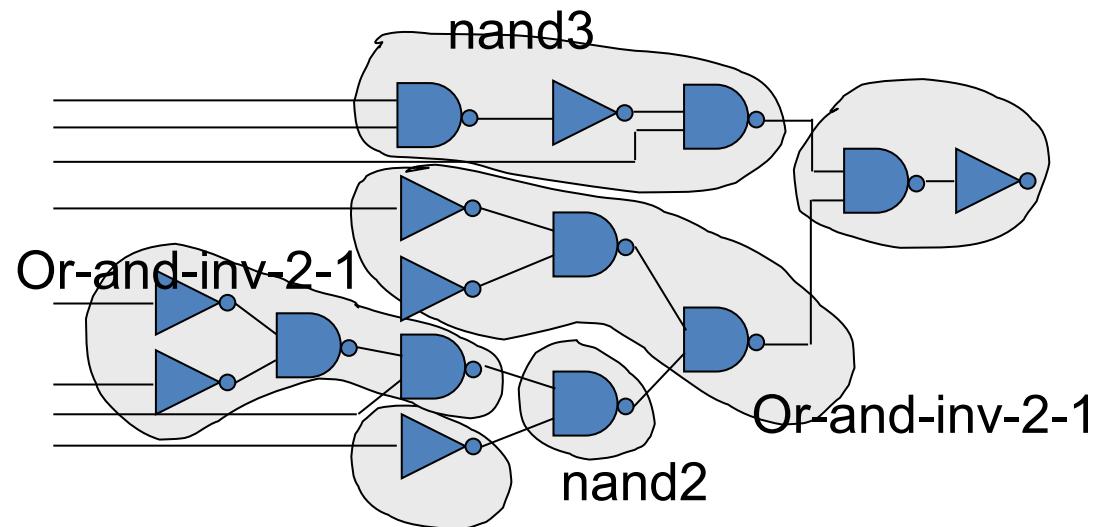


論理の最適化

- 状態遷移表の各状態を符号化すると状態遷移関数および出力関数をあらわす真理値表が得られる
- その関数はドントケアを含む場合もある
- 論理最適化とは与えられた論理関数に対する「コストの低い」実現を求める問題
- 実現方法：二段論理回路・多段論理回路

テクノロジ・マッピング

- 性能・面積・消費電力の目標を満たす最良の素子の組み合わせを見出す



具体的的には・・16ビット乗算器

```

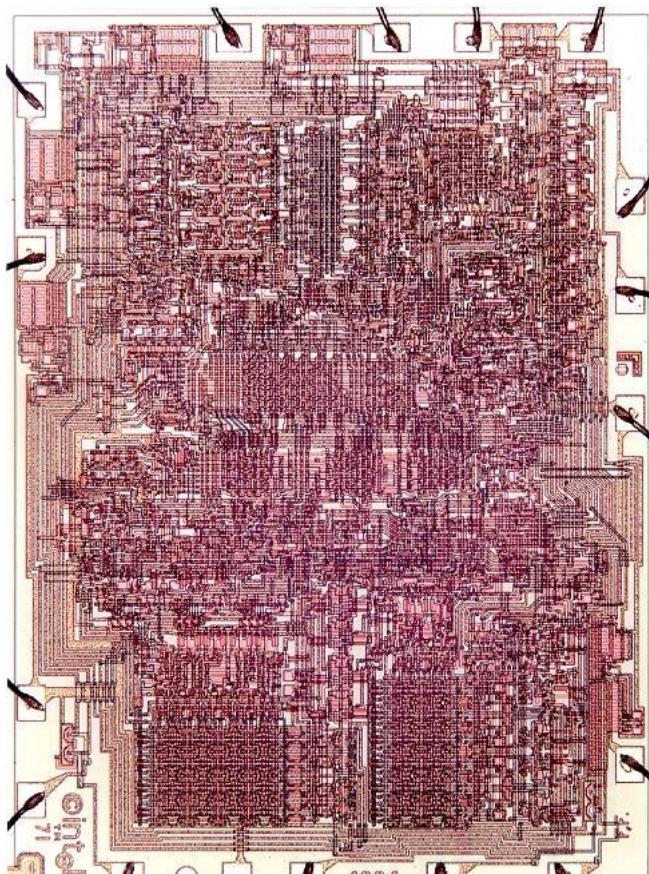
module multi( inA, inB, inTC, PROD, CLK, RST);
    parameter A_width = 16;
    parameter B_width = 16;
    input  RST,CLK, inTC;
    input [A_width-1 : 0] inA;
    input [B_width-1 : 0] inB;
    output [A_width+B_width-1 : 0] PROD;
    reg [A_width+B_width-1 : 0] PROD;
    reg [A_width-1 : 0] inst_A_FF;
    reg [B_width-1 : 0] inst_B_FF;
    reg inst_TC_FF;
    wire [A_width+B_width-1 : 0] PRODUCT_inst_IN;
    DW02_mult #(A_width, B_width)
    U1 ( .A(inst_A_FF), .B(inst_B_FF), .TC(inst_TC_FF), .PRODUCT(PRODUCT_inst_IN) );
    always @ (posedge CLK or negedge RST) begin
        if (!RST) begin
            PROD <= 0; inst_TC_FF <= 0;
            inst_A_FF <= 0; inst_B_FF <= 0;
        end else begin
            PROD <= PRODUCT_inst_IN;
            inst_A_FF <= inA; inst_B_FF <= inB;
            inst_TC_FF <= inTC;
        end
    end
end
endmodule

```

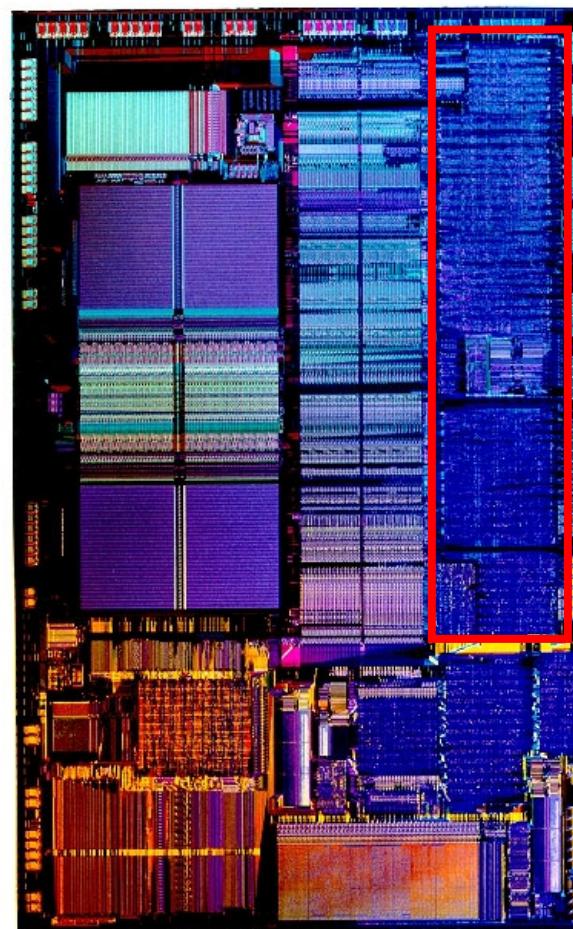
制約条件	最大遅延[ns]	面積[um ²](kG)	消費電力[mW]
20.0	10.54	29,062 (2.25)	2.20
10.0	9.8	29,169 (2.26)	2.35
5.5	5.5	33,211 (2.57)	4.66
5.0	5.1	37,456 (2.90)	5.48
4.5	5.2	37,420 (2.90)	6.04
4.0	5.2	37,278 (2.89)	6.75

0.18um CMOSの例

設計自動化と自動配置配線



Intel 4004(1971)



Intel 80486(1989)

配置配線領域

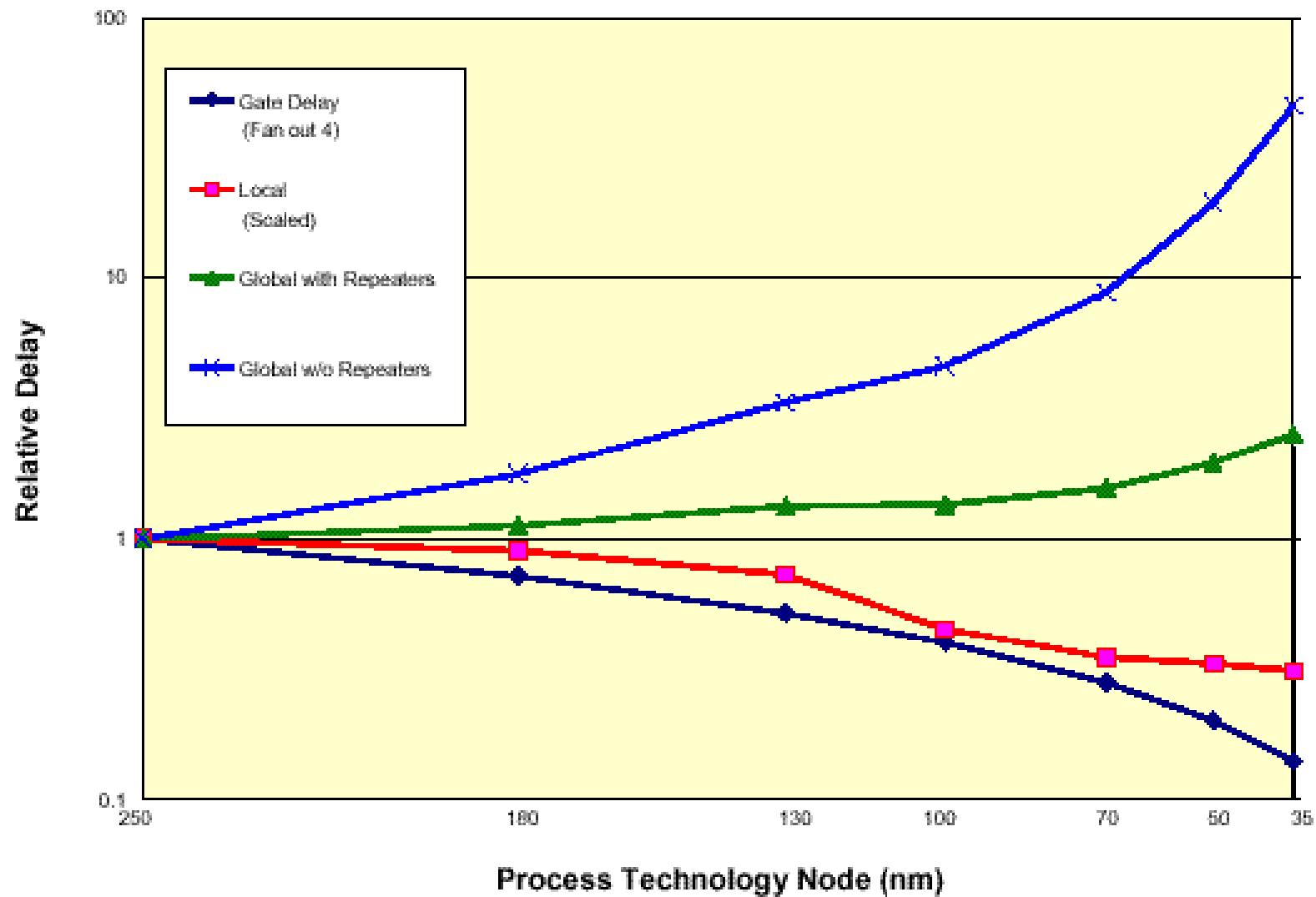
微細化と遅延時間

- トランジスターの寸法が $1/\alpha$ 倍になると・
 - 電源電圧: $1/\alpha$ 倍
 - ドレイン電流: $1/\alpha$ 倍 ($W:1/\alpha, L:1/\alpha, C:x\alpha$)
 - 容量: $1/\alpha$ 倍

→電流: $1/\alpha$ 倍
→トランジスタのゲート遅延時間: $(CV/I) \rightarrow 1/\alpha$ 倍
これをスケーリング則と呼ぶ
- 配線遅延は? CR時定数
 - 単位配線長あたりのC: $1/\alpha$
 - 単位配線長あたりのR: α 倍

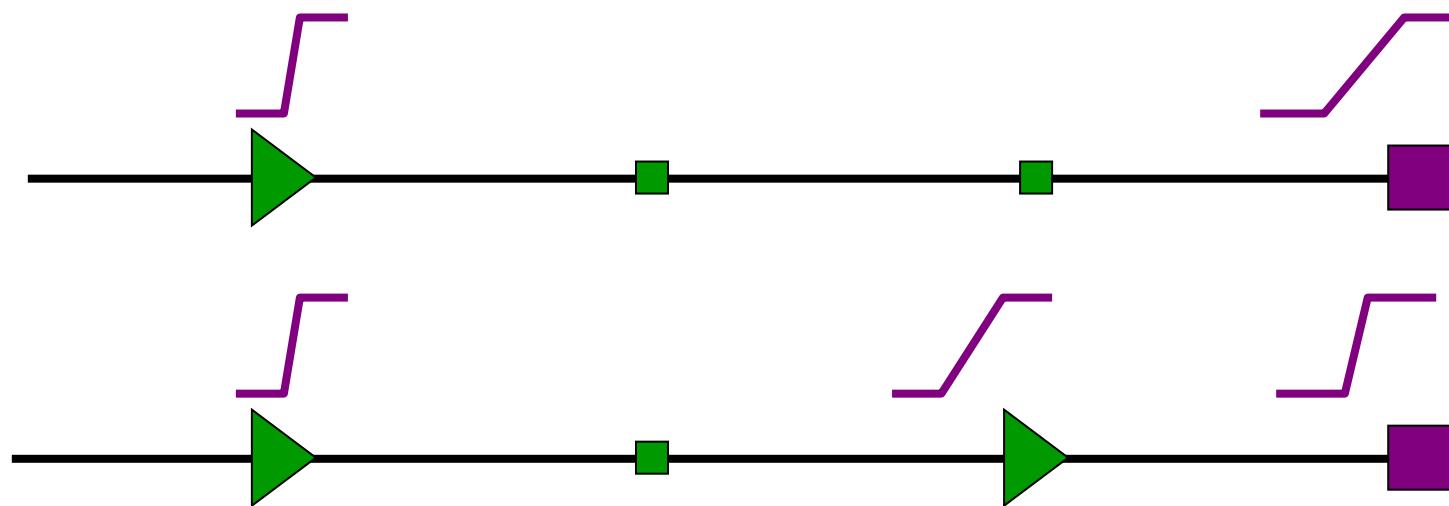
→ CR時定数: 1
→ トランジスタ遅延との比: α 倍
→ 一定長配線とトランジスタ遅延との比: α^3 倍

微細化と配線遅延



配線遅延改善のためのバッファ挿入

配置が決まると、各ネットの配線長がほぼ確定
その情報を基に、長距離配線にバッファを挿入することで遅延の改善と波形鈍りを改善



配線遅延に対する対応

- 配置(配線経路)が決まらないと配線長が決まらない
- レイアウトを意識した論理合成
 - 配置配線プログラムに対する制約付加
 - 配線長予測精度の向上
 - 配置ルーチンの共有化
- レイアウト時の配線長に基づく論理最適化
 - 重負荷ネットに対するバッファ挿入(ネットバッファリング)
 - 重負荷ネットの論理の再合成
- 論理合成と配置配線の同時実行

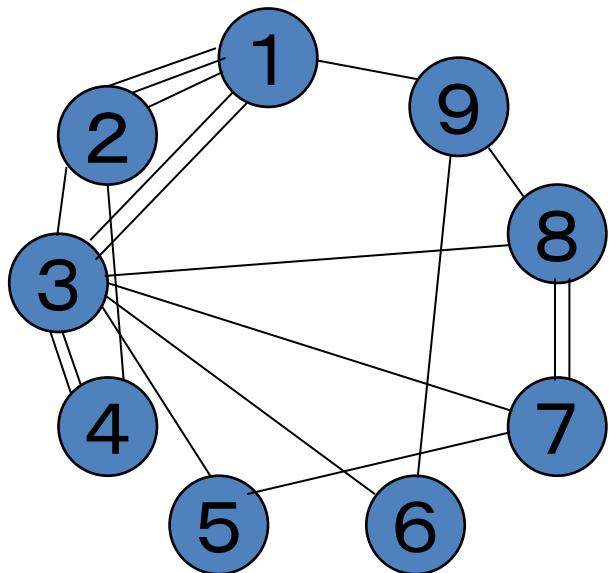
配置配線

- フロアープラン
 - セル配置領域の確保
- 配置
 - セルの配置(総配線長最小・配線混雑度緩和・特定の配線長最小化等)
- 概略配線
 - 配線経路の決定
- 詳細配線
 - 設計規則を満たすように配線、規則を満たさない部分の修正
- 設計規則検査・設計検証
 - 生成されたレイアウトの検証

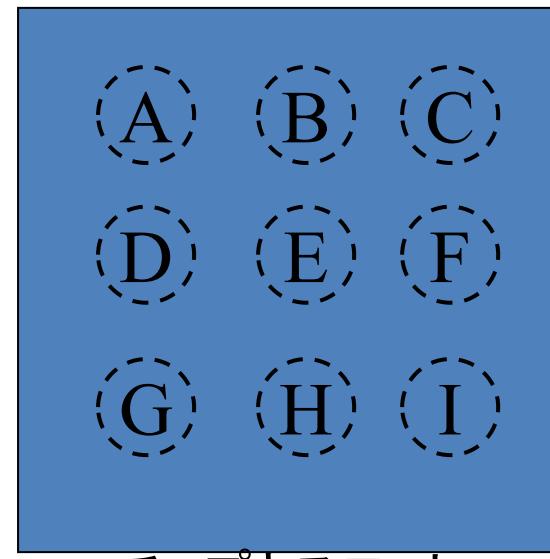
配置手法

- 目標→なるべく配線のしやすい配置
 - 仮想配線長
線長が短ければ配線の占める面積が少なく配線しやすい
 - カット数
局所的に配線が集中するのを避けたほうが配線しやすい

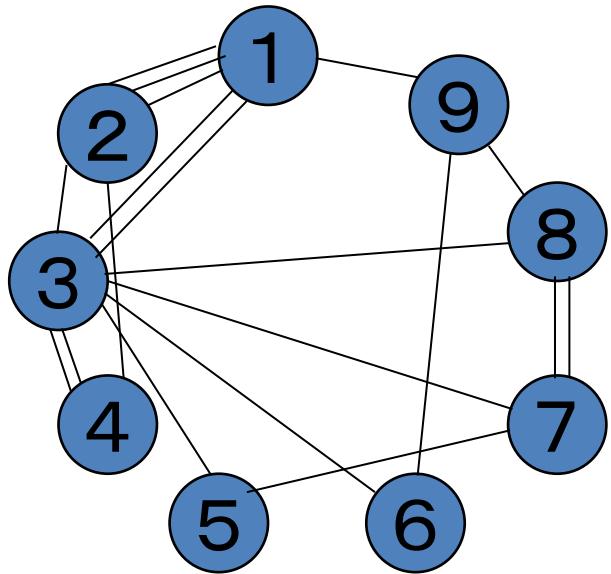
配置問題と基本モデル



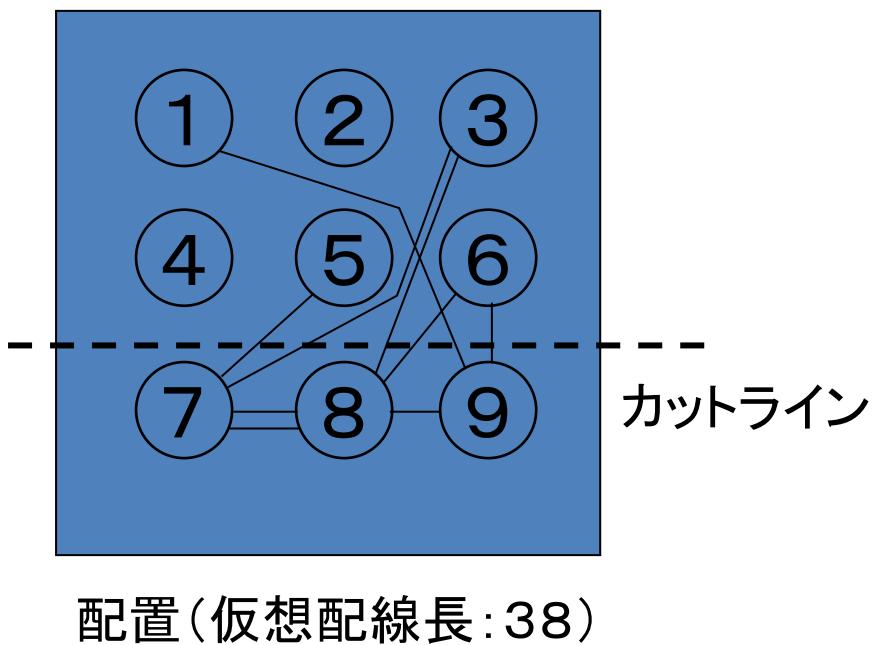
モジュールと接続関係



チップとスロット



モジュールと接続関係



配置(仮想配線長:38)

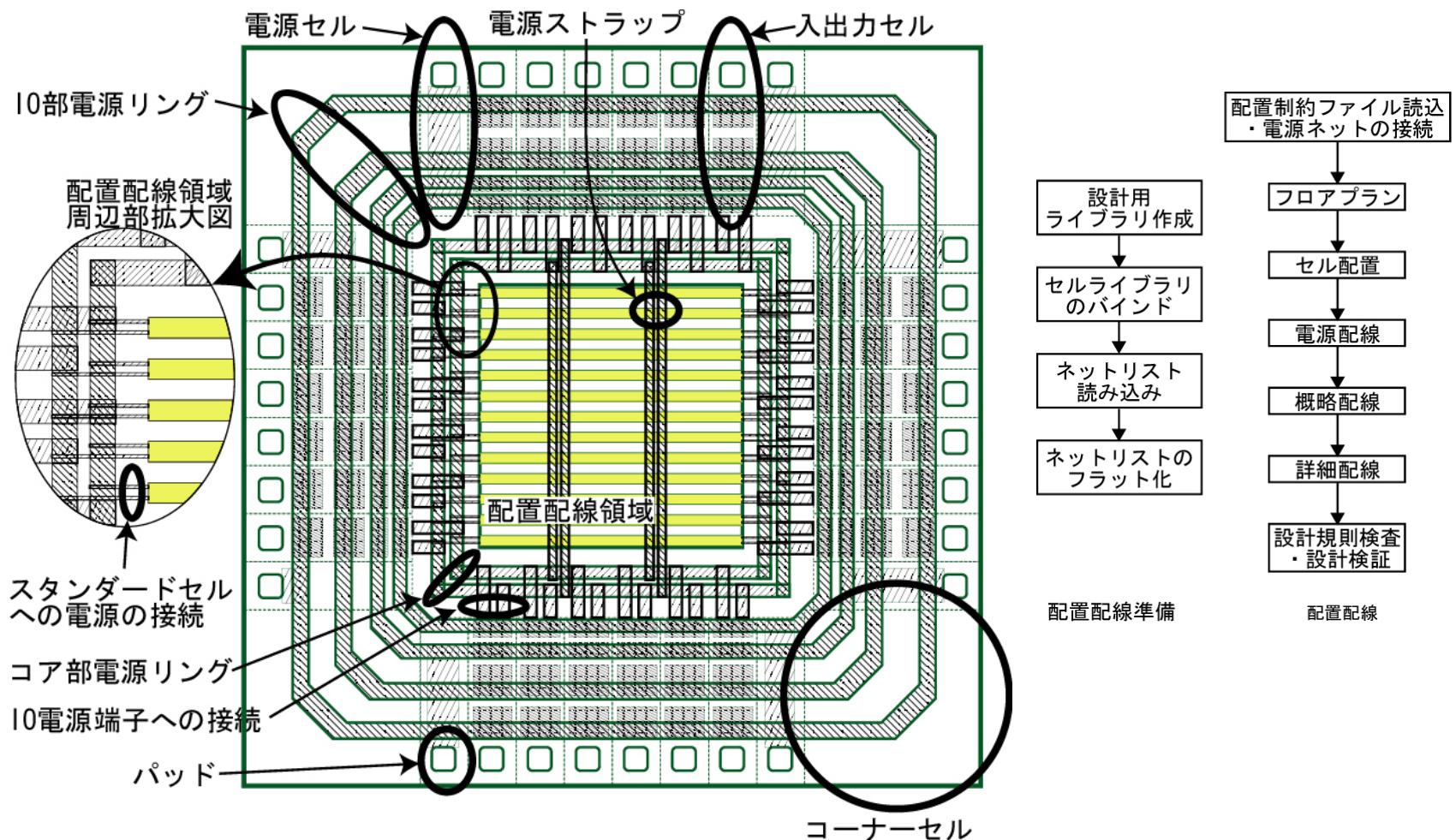
配線手法

- 迷路法
- 線分探索法
- チャネル配線法

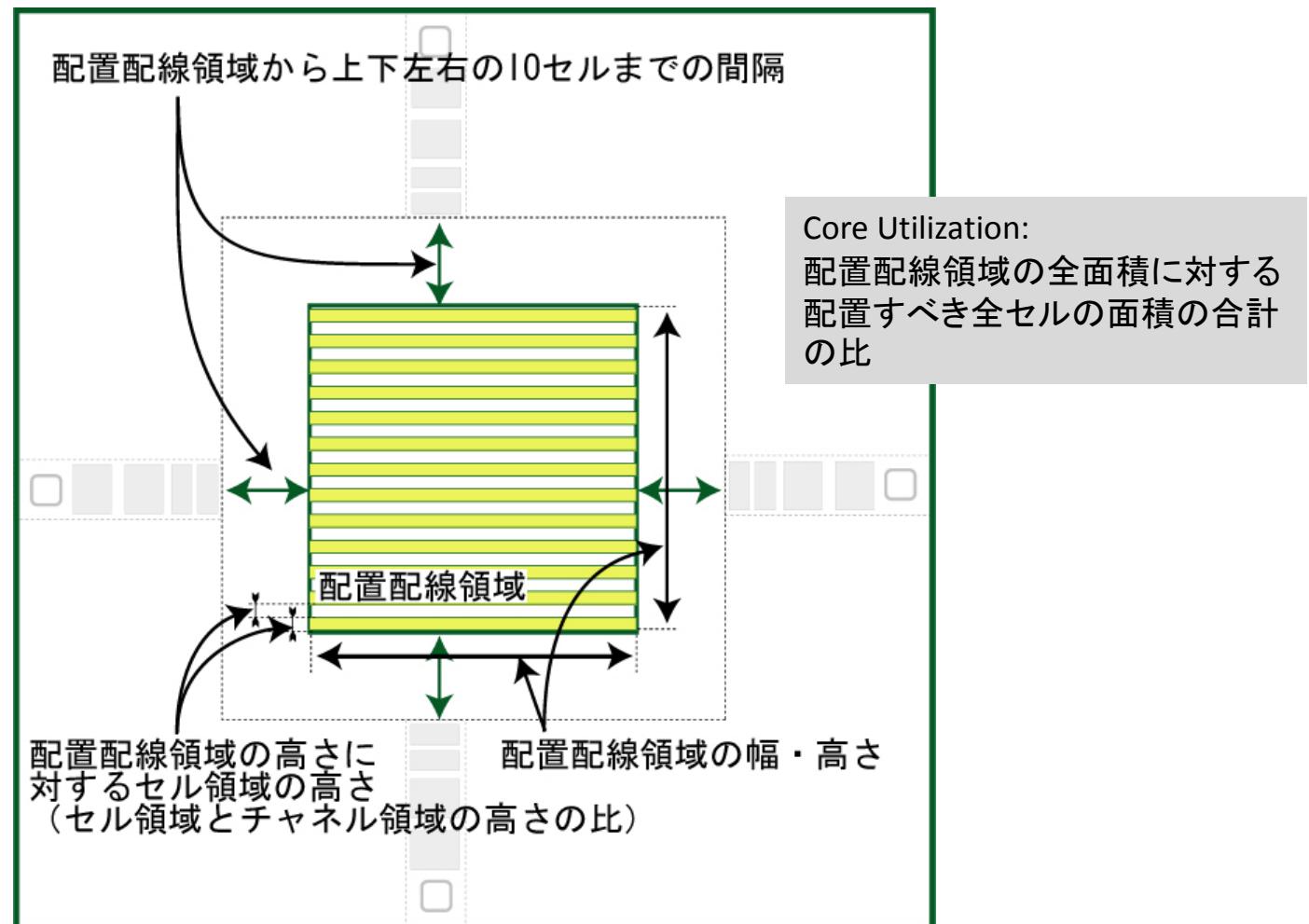
実際のLSIでは

- チャネルベース配置配線方式
 - 各チャネル領域を通過する配線本数はチャネル毎に可変
 - 100%の配線が保証される
- エリアベース配置配線方式
 - チップ面積・チャネル領域の大きさは当初の設定時に決定
 - 100%配線を保証できない

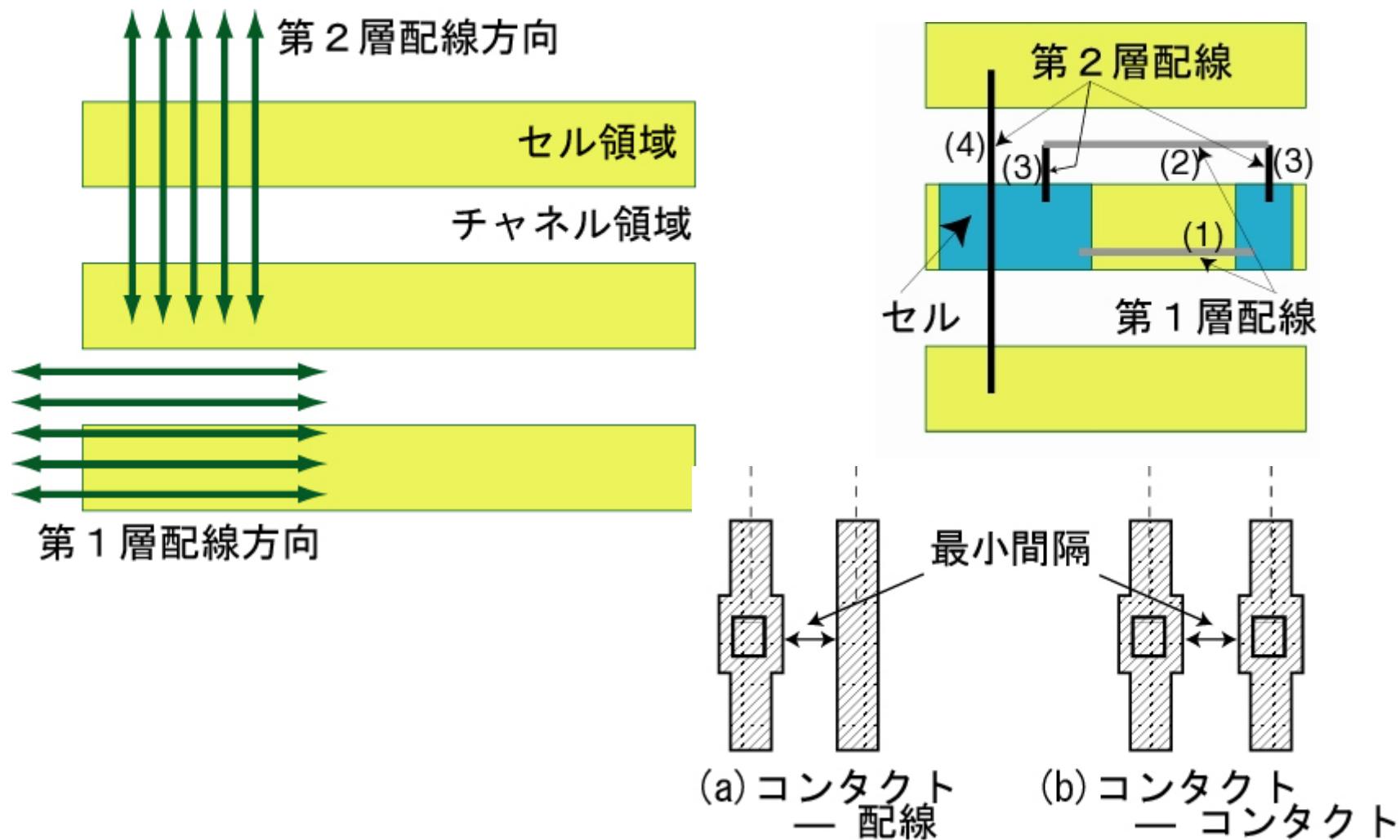
ASIC配置配線の流れ



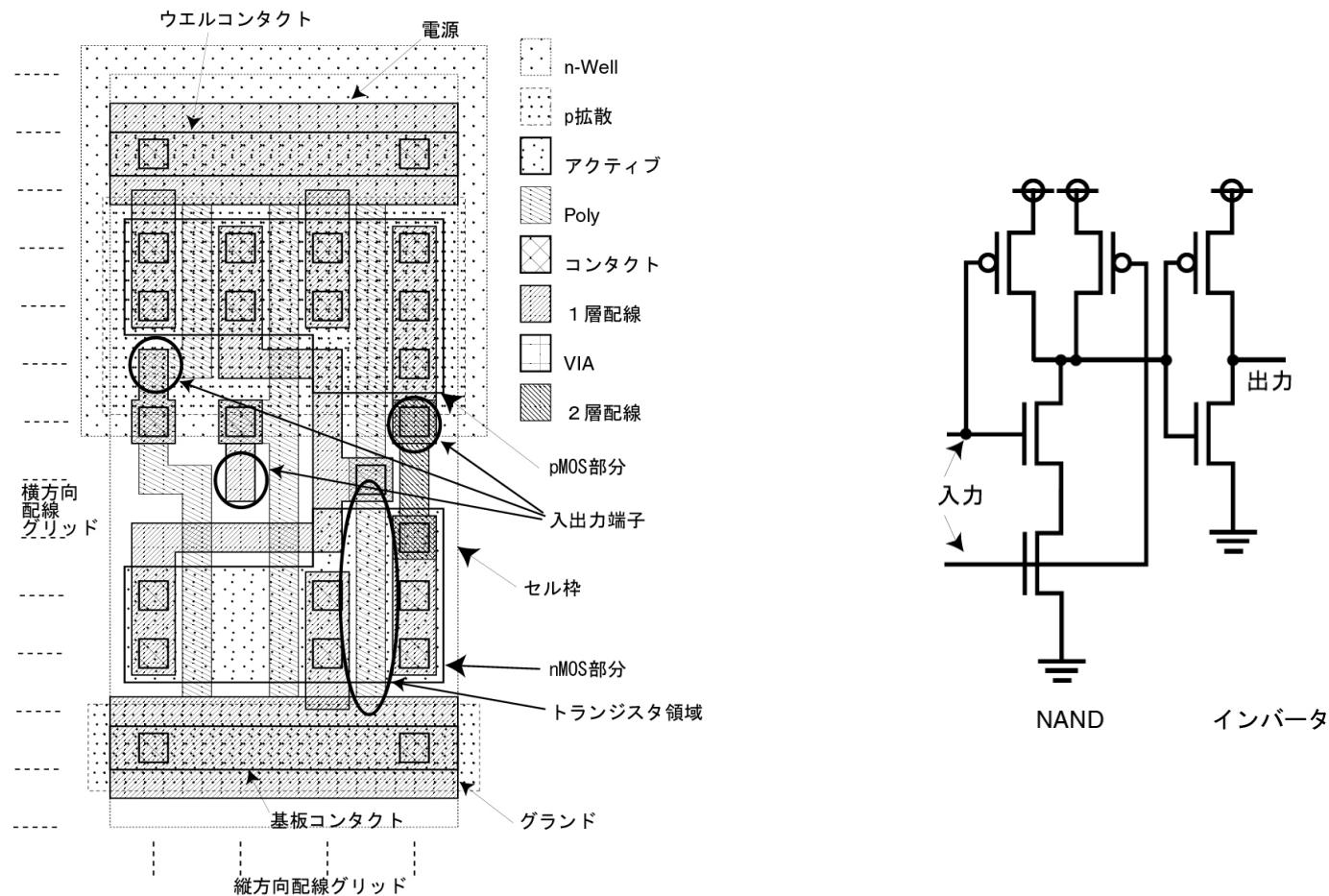
フロアープラン



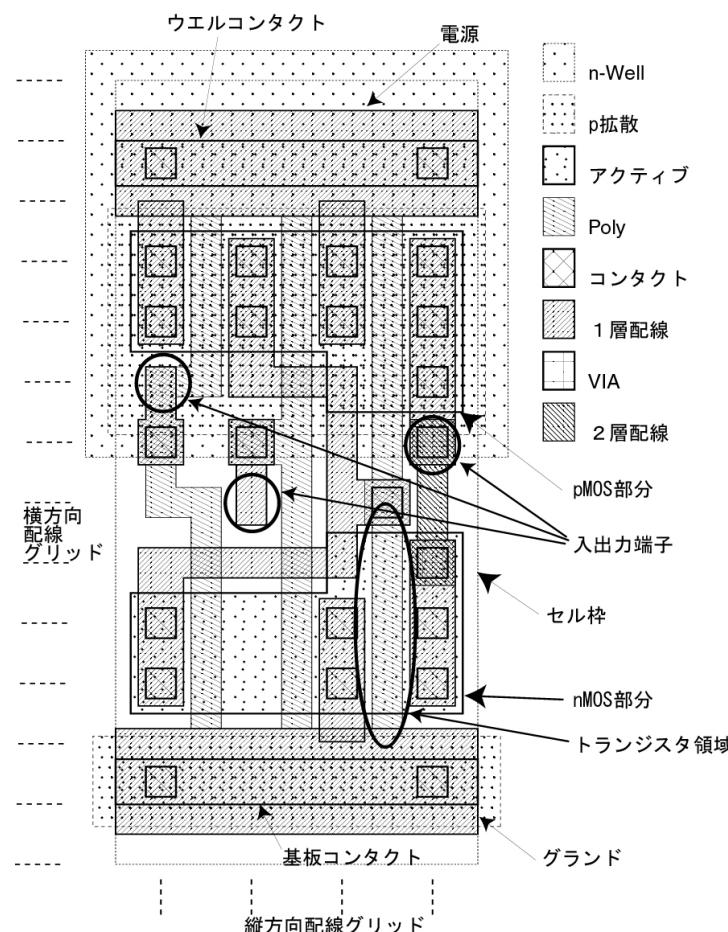
セルの配置と配線方向



セルベース配置配線向け セルのレイアウト例

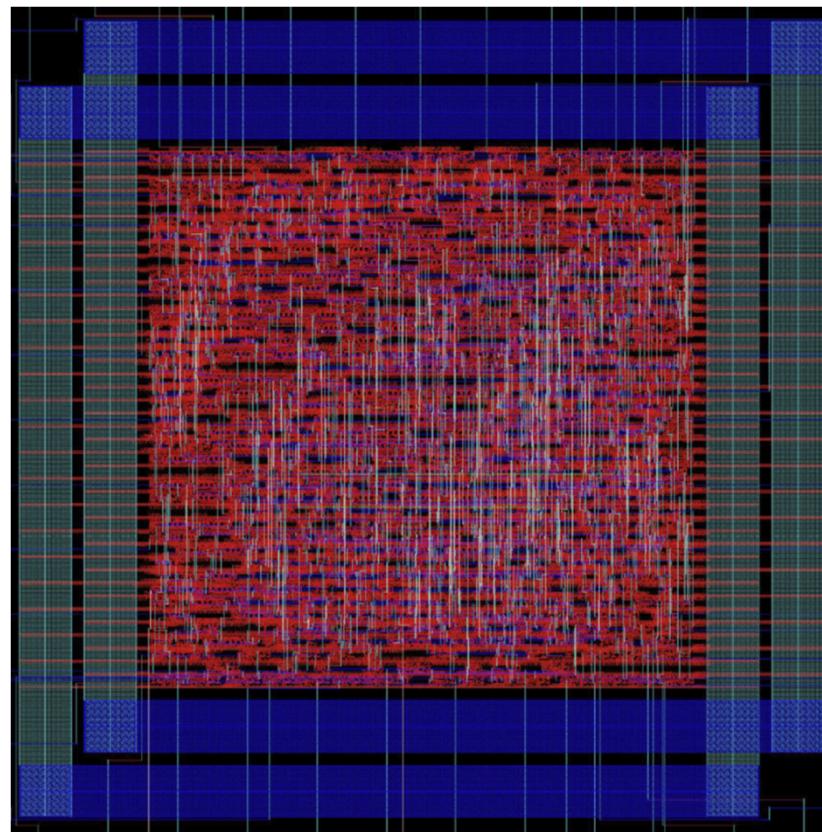


セルベース配置配線向け セルのレイアウト例

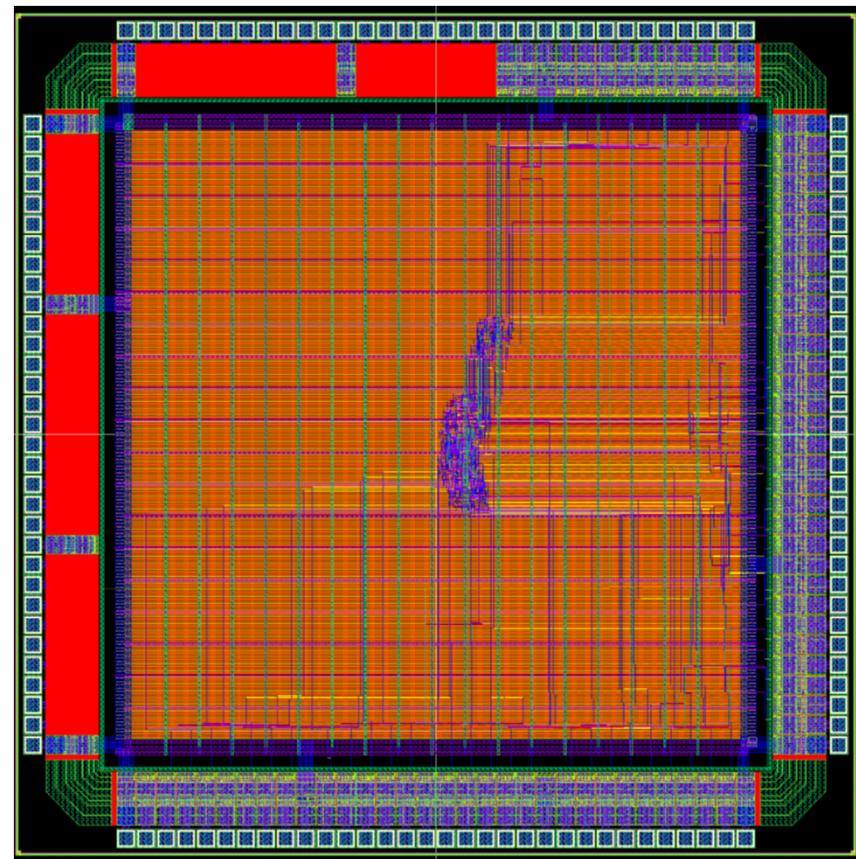


- ◎セルの高さは一定(もしくは2倍、3倍高)
- ◎セルの高さは横方向配線グリッドの整数倍
- セル幅は縦方向の整数倍
- 端子は配線グリッド上に配置
- 電源グランド配線をセルの上下に配置
- ◎配置した場合に隣接セルの間で設計規則違反を起こさないようにする

16ビット乗算器のレイアウト



乗算器のみ



IOを含めたチップ全体