

2020年4月24日

ハードウェア設計論:4

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 ～CPU:ハード&ソフト～

質問は随時はSLACK #2020s-ハードウェア設計論

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

本日の課題

課題5～課題6までを、次回(5月1日朝)までに提出を終えておくこと

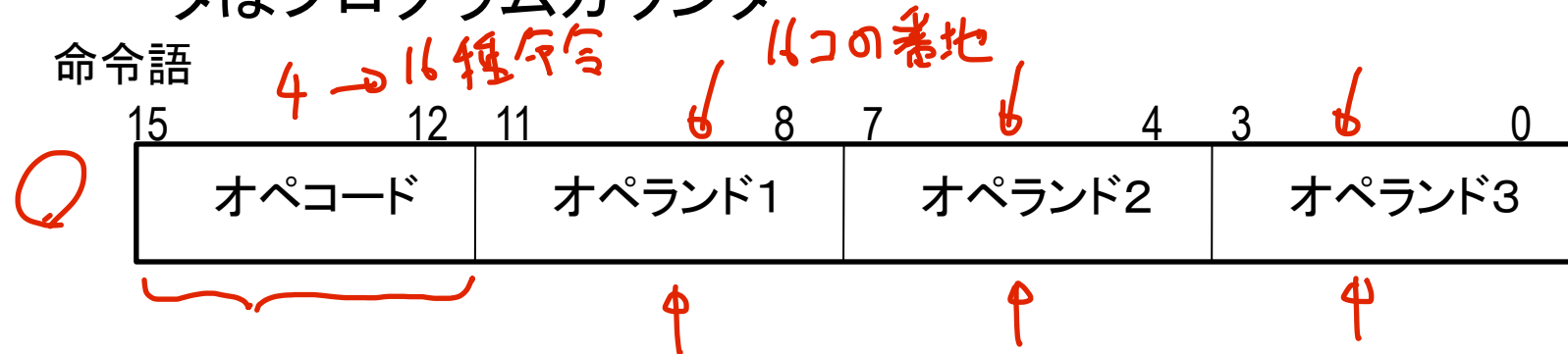
次回は、課題7-2:CPUとそのプログラム(7-4)を実装していただきます

12:00から追加の説明します

演習7

- 簡単なマイクロプロセッサを作ってみよう
 - 命令16ビット
 - 加算、減算、右シフト、左シフト、論理和、論理積、論理反転、排他的論理和
 - ジャンプ、条件分岐(ゼロ)
 - ロード、ストアー、下位ビットセット
 - データ16ビット、ゼロフラグ
 - レジスタ: 16本、ただし0番レジスタは常に0、15番レジスタはプログラムカウンタ

/dev/null と等価



演習7: 命令セット

書き込み先レジスタ番号

ISA

命令に利用する

読み出しレジスタ番号

命令	オペコード	オペランド1	オペランド2	オペランド3	命令の詳細
加算	0000	RC	RA	RB	$[RA] + [RB] \rightarrow [RC]$
減算	0001	RC	RA	RB	$[RA] - [RB] \rightarrow [RC]$
右シフト	0010	RC	RA	RB	$[RA] \gg [RB] \rightarrow [RC]$
左シフト	0011	RC	RA	RB	$[RA] \ll [RB] \rightarrow [RC]$
論理和	0100	RC	RA	RB	$[RA] [RB] \rightarrow [RC]$
論理積	0101	RC	RA	RB	$[RA] \& [RB] \rightarrow [RC]$
論理反転	0110	RC	RA	RB	$\sim [RA] \rightarrow [RC]$
排他的論理和	0111	RC	RA	RB	$[RA] \wedge [RB] \rightarrow [RC]$
下位ビットセット	1100	RC	即値データ		$\{8b0, IMM\} \rightarrow [RC]$
ジャンプ	1000	RC	0000	RB	$[RB] \rightarrow [PC], [PC] + 1 \rightarrow [RC]$
条件分岐(ゼロ)	1001	0000 or RC	0000	RB	$If(flag) [RB] \rightarrow [PC]$
ロード	1011	RC	0000	RB	$\#[RB] \rightarrow [RC]$
ストア	1010	0000	RA	RB	$[RA] \rightarrow \#[RB]$

ALU

↑

算術
論理
演算

↑

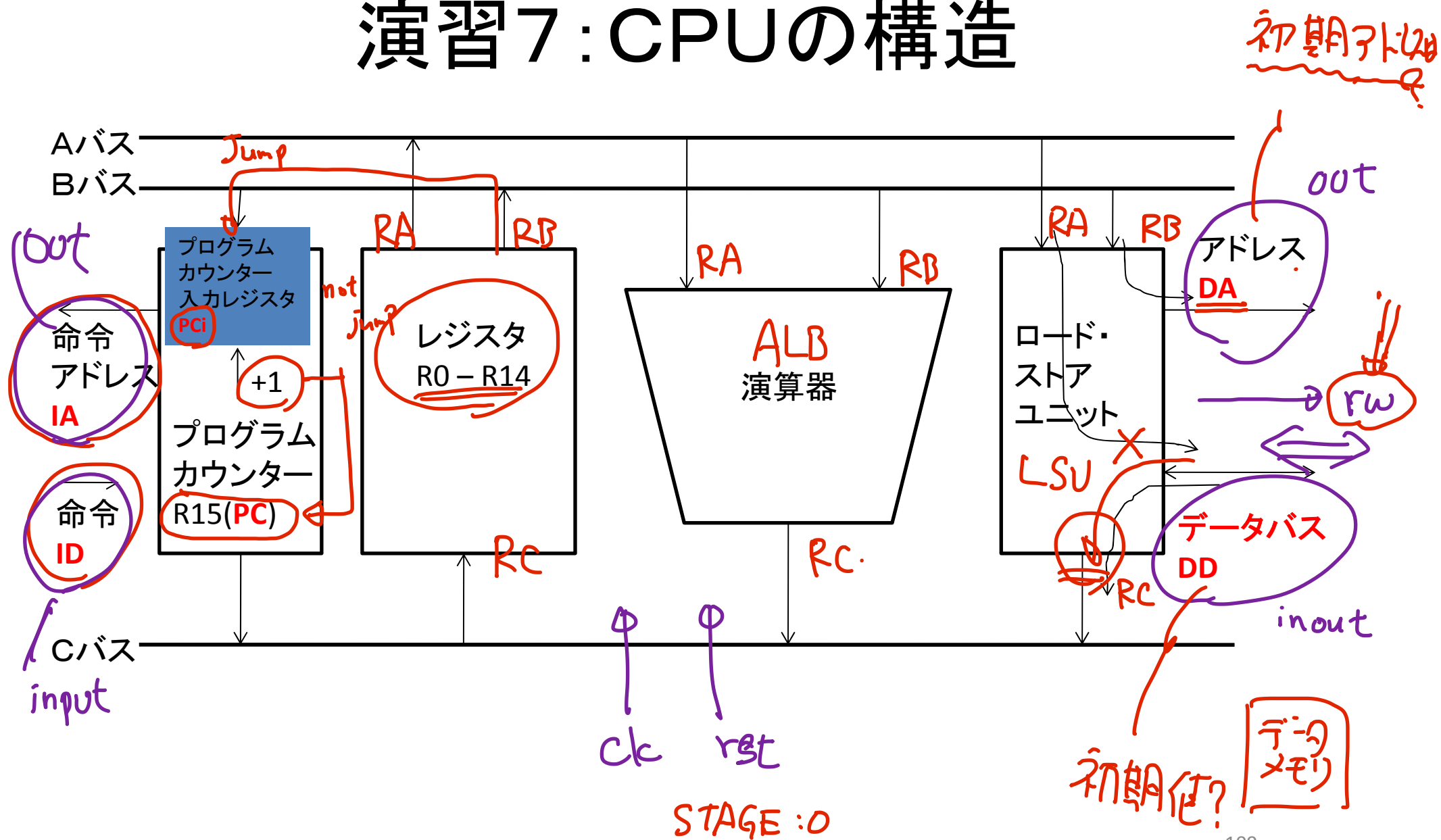
定数を
プログラム中に
実装したいとき

RA := 0000

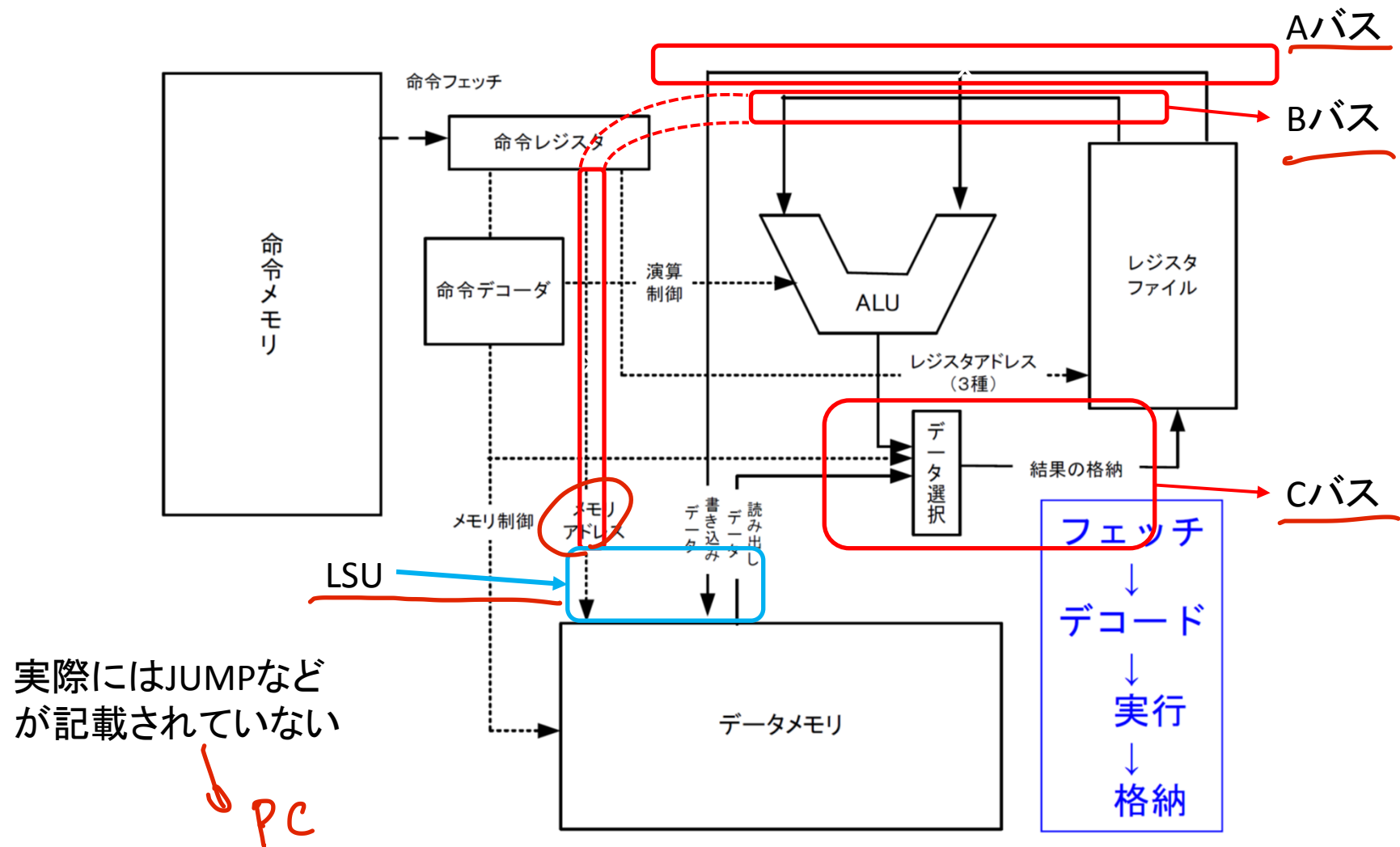
↑

レジスタ番号に書き込む

演習7: CPUの構造

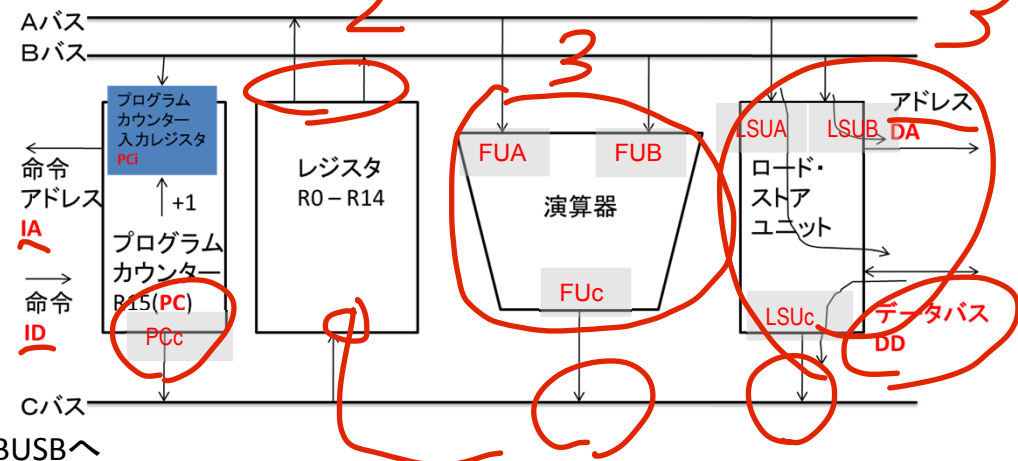


コンピュータアーキテクチャでは、、、



演習7: CPUの動作

rst : Active reset
rst = 1 とき
nrst : Negative Reset rst = 0..



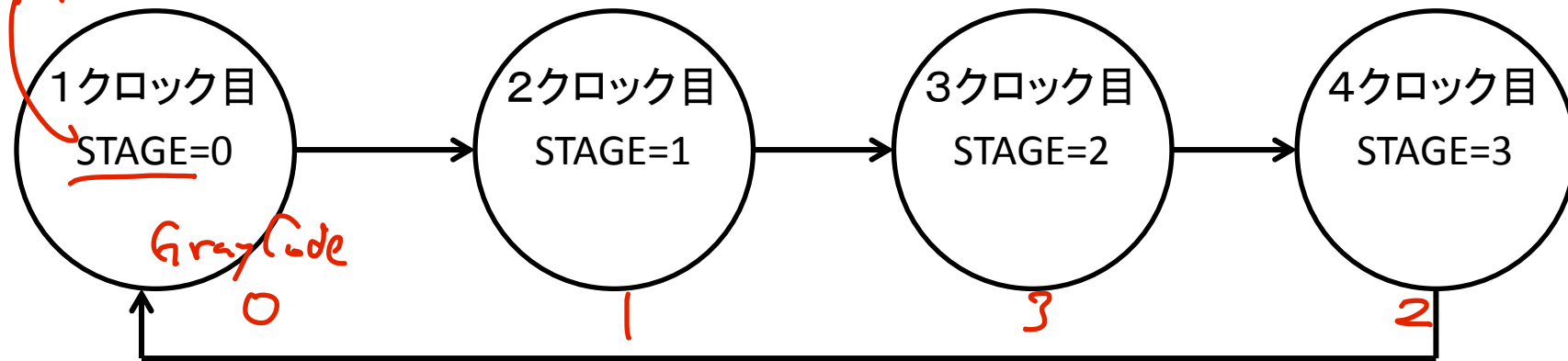
- 1クロック目: 命令フェッチ **Inst. Fetch**
 - 命令アドレスの番地から命令を取り込む
- 2クロック目: 命令デコード、レジスタ読み出し **IF DEC**
 - 命令のOP2, OP3のレジスタを読み出しBUSA, BUSBへ
 - オペコード0xxxの場合に演算器の入力レジスタA, BにBUSA, BUSBの内容を取り込む
 - オペコード101xの場合にロードストアユニットの入力レジスタA, BにBUSA, BUSBの内容を取り込む
 - オペコード1000の場合プログラムカウンタ入力レジスタPCiにBBUSの内容を取り込む
 - オペコード1001かつフラグレジスタが1の場合プログラムカウンタ入力レジスタPCiにBBUSの内容を取り込む
 - オペコードが100x以外の場合には、プログラムカウンタ入力レジスタPCiにPC + 1を取り込む
- 3クロック目: 演算実行 **EXE**
 - オペコード0xxxの場合に、xxxに応じた演算結果を演算器出力レジスタFUCに取り込む
 - オペコード101xの場合、LSUAをデータアドレスに、x=1のときRW=1とし、データバスの結果をLSUCに取り込む、x=0のとき、RW=0とし、データバスにLSUAを出力する
 - オペコード1000のときPC出力レジスタPCcにPC+1値を取り込む
- 4クロック目: 書き込み **WB write back**
 - オペコード0xxxの場合に、演算器出力レジスタ値FUCをCBUSに出力する
 - オペコード101xの場合、LSU出力レジスタ値LSUCをCBUSに出力する
 - オペコード1100の場合、即値データIMMをCBUSに出力する(ただし上位8ビットは0とする)
 - オペコード1000のときPC出力レジスタ値PCcをCBUSに出力する
 - CBUS値をOP1のレジスタに書き込む
 - PCにプログラムカウンタ入力レジスタ値を書き込む

演習7－1: CPUの動作の状態遷移 図を描いてみよう

状態遷移図からVerilogHDLのひな型ができるはず・・・

演習7-1: CPUの動作の状態遷移

reg [1:0] STAGE; 図を描いてみよう



状態遷移図からVerilogHDLのひな型ができるはず・・・

```

always @(posedge CK) begin
  if( RST == 1 ) begin
  end else begin
    if( STAGE == 0 )begin
      STAGE <= 1;
    end else if( STAGE == 1 ) begin
      end
      STAGE <= 2;
    end else if( STAGE == 2 ) begin
      STAGE <= 3;
    end else if( STAGE == 3 ) begin
      STAGE <= 0;
    end
  end
end
end
  
```

case (STAGE)

0 :






1 :

2 :






3 :

end case

STAGE <= STAGE + 1;

```
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 [1:0] STAGE;
```

```
  always @(posedge CK) begin  
    if( RST == 1 ) begin  
  
    end else begin  
      if( STAGE == 0 )begin  
  
        STAGE <= 1;  
      end else if( STAGE == 1 ) begin  
  
      end  
      STAGE <= 2;  
    end else if( STAGE == 2 ) begin  
  
      STAGE <= 3;  
    end else if( STAGE == 3 ) begin  
  
      STAGE <= 0;  
    end  
  end  
end  
endmodule
```

継続代入

プログラムカウンタの出力はIA(命令アドレス)として外部に出力

オペコードは命令語 (INST) の15-12ビット目

オペランド1は命令語の11-8ビット目

オペランド2は命令語の7-4ビット目

オペランド3は命令語の3-0ビット目

即値データは命令語の7-0ビット目

Aバスには常にOPR2で指示されるレジスタファイルの値を出力

Bバスには常にOPR3で指示されるレジスタファイルの値を出力

```
assign IA = PC;
```

```
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 = RF[OPR2];
```

```
assign BBUS = RF[OPR3];
```

通常レジスタ0は0を出力する

```
assign ABUS = (OPR2 == 0 ? 0 : RF[OPR2] );
```

```
assign BBUS = (OPR3 == 0 ? 0 : RF[OPR3] );
```

LSU・バス周りの制御

3クロック目

オペコード101xの場合、

x=1(Load)のとき RW=1とし、データバス(DD)の結果をLSUCに取り込む、
x=0(Store)のとき、RW=0とし、データバス(DD)にLSUAを出力する

4クロック目

オペコード0xxxの場合に、演算器出力レジスタ値をCBUSに出力する

オペコード101xの場合、LSU出力レジスタ値FUCをCBUSに出力する

オペコード1100の場合、即値データIMMをCBUSに出力する(ただし上位8ビットは0とする)

オペコード1000のときPC出力レジスタ値PCcをCBUSに出力する

```
assign DD =(RW==0? LSUA : 'b Z);
```

```
assign DA = LSUB;
```

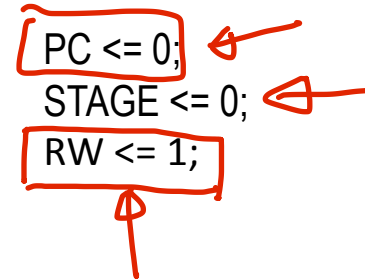
```
assign CBUS = (OPCODE[3]==0 ? FUC : (OPCODE[3:1]=='b 101 ? LSUC :  
      (OPCODE=='b 1100 ? {8'b 0,IMM} : OPCODE=='b 1000 ? PCC : 'b z)));
```

ALU

Selection

手続き代入：リセット時動作

プログラムカウンターの初期化
状態変数(ステージ)の初期化
ロードストアユニット出力を読み込みモードとし外部メモ
リへの不必要な書き込みをなくす



PC <= 0;
STAGE <= 0;
RW <= 1;

The code is annotated with hand-drawn red boxes and arrows. The first line 'PC <= 0;' is enclosed in a red box with a red arrow pointing to it from the right. The second line 'STAGE <= 0;' is also enclosed in a red box with a red arrow pointing to it from the right. The third line 'RW <= 1;' is enclosed in a red box with a red arrow pointing to it from below.

手続き代入：1, 2 クロック目

1クロック目

命令アドレスの番地から命令(ID)を取り込む(INSTに)

```
INST <= ID;
```

2クロック目

オペコード0xxxの場合に演算器の入力レジスタA, Bに
BUSA, BUSBの内容を取り込む

```
if( OP CODE[3] == 0 ) begin  
    FUA <= ABUS; FUB <= BBUS;  
end
```

オペコード101xの場合にロードストアユニットの入力レジ
スタA, Bに BUSA, BUSBの内容を取り込む

```
if( OP CODE[2:1] == 'b01) begin  
    LSUA <= ABUS; LSUB <= BBUS;  
end
```

オペコード1000の場合プログラムカウンタ入力レジスタ
PCiに BBUSの内容を取り込む

オペコード1001かつフラグレジスタが1の場合プログラム
カウンタ入力レジスタPCiに BBUSの内容を取り込む
オペコードが100x以外の場合には、プログラムカウンタ
入力レジスタPCiに PC + 1を取り込む

```
if( (OP CODE[3:0] == 'b 1000) ||  
    (OP CODE[3:0] == 'b 1001 && FLAG == 1 ) )  
    PCI <= BBUS;  
else    PCI <= PC + 1;
```

手続き代入：3・4クロック目

3クロック目

オペコード0xxxの場合に、xxxに応じた演算結果を演算器出力レジスタFUCに取り込む

```
if( OP CODE[3] == 0 ) begin
    case (OP CODE[2:0])
        'b 0000 : FUC <= FUA + FUB;
        'b 0001 : FUC <= FUA - FUB;
```

```
    endcase
end
```

オペコード101xの場合

x=1のとき **RW=1**,とし、**データバス**の結果をLSUCに取り込む

x=0のとき、**RW=0**とし、**データバス**にLSUAを出力する

オペコード1000のときPC出力レジスタPCCに**PC+1**値を取り込む

```
if( OP CODE[3:1] == 'b 101 ) begin
    if( OP CODE[0] == 0 ) begin
        RW <= 0;
    end else begin
        RW <= 1; LSUC <= DD;
    end
end
end
```

```
if( OP CODE[3:0] == 'b 1000 )
    PCC <= PC+1;
```

4クロック目

CBUS値をOP1のレジスタに書き込む

PCに**プログラムカウンタ**入力レジスタ値を書き込む

```
RF[OPR1] <= CBUS;
PC <= PCI;
```

そのほか。。分岐・フラグ(Zero)

2クロック目

- オペコード1000の場合プログラムカウンタ入力レジスタPCiに BBUSの内容を取り込む
- オペコード1001かつフラグレジスタが1の場合プログラムカウンタ入力レジスタPCiに BBUSの内容を取り込む
- オペコードが100x以外の場合には、プログラムカウンタ入力レジスタPCiに PC + 1を取り込む

```
if( (OPCODE[3:0] == 'b 1000) ||  
    (OPCODE[3:0] == 'b 1001 && FLAG == 1 ) )  
    PCI <= BBUS;  
else PCI <= PCn;
```

4クロック目

- FLAGの生成: 演算命令のとき、CBUSのデータが0であればFLAGを1にし、そうではない場合にはFLAGを0にする

```
if( OPCODE[3] == 0 ) begin  
    if( CBUS == 0 ) FLAG <= 1;  
    else FLAG <= 0;  
end
```


演習7-2: CPUの完成

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

\$dumpfile("cpu.vcd");

\$dumpvars;

CK = 0;

RST = 0;

#5 RST = 1;

#100 RST = 0;

#10000 \$finish;

end

always @(negedge CK) begin

ID = IMEM[IA];

end

cpu.vを完成させ simcpuで動作を確認する

always @(negedge CK) begin

if(RW==1)DDi = DMEM[DA]; else DMEM[DA]=DD;

end

initial begin

IMEM[0]='b 1100_0000_0000_0000; // IMM R0, [0]

IMEM[1]='b 1100_0001_0000_0001; // IMM R1, [1] R1 = 1

IMEM[2]='b 1100_0010_0000_0010; // IMM R2, [2] x

IMEM[3]='b 1100_0011_0000_0011; // IMM R3, [3] R3 = 3

IMEM[4]='b 1100_0100_0000_0100; // IMM R4, [4] x

IMEM[5]='b 0000_0101_0001_0011; // ADD R5, R1, R3 ⇒ R5 = 4

IMEM[6]='b 1010_0000_0101_0000; // ST R5, R0

end

always #10 CK = ~CK;

endmodule

[R5] ← # [R0] 0番地の
レジスタに、
R5の値を格納

演習7－3：CPUで計算させる

- 1～10までの整数を足し算し結果をデータメモリの0番地に出力するプログラム

- R1: アキュムレータ
- R2: 加算する数(初期値1)
- R3: 加数(1)
- R4: ループ回数(9)
- R5: ジャンプアドレス(**12**)
- R6: ジャンプアドレス(7)
- R7: 出力メモリアドレス(0)

演習7－3：CPUで計算させる


- 1～10までの整数を足し算し結果をデータメモリの0番地に出力するプログラム

- R1: アキュムレータ
- R2: 加算する数(初期値1) IMEM[0]='b 1100_0001_0000_0000; // IMM R1, [0]
- R3: 加数(1) IMEM[1]='b 1100_0010_0000_0001; // IMM R2, [1]
- R4: ループ回数(9) IMEM[2]='b 1100_0011_0000_0001; // IMM R3, [1]
- R5: ジャンプアドレス(12) IMEM[3]='b 1100_0100_0000_1001; // IMM R4, [9]
- R6: ジャンプアドレス(7) IMEM[4]='b 1100_0101_0000_1100; // IMM R5, [12]
- R7: 出力メモリアドレス(0) 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

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

下記のこと
ISA に `MUL Rc RA RB`
を追加して
ok -- `Rc = RA * RB`



課題5 mul.v (multest2.vを使用) 演習5:乗算の実装

```
module mul(A,B,O,ck,start,fin);
input [7:0] A, B;
input ck,start;
output [16:0] O;
output fin;
reg [3:0] st;
reg [7:0] AIN, BIN;
reg [16:0] O;
reg fin;
always @(posedge ck) begin
    if( start == 1 ) begin
        st <= 0;
        fin <= 0;
        AIN <= A;
        BIN <= B;
        O <= 0;
    end else begin
        case (st)
        0: O <= (O<<1) + AIN * BIN[7];
        1: O <= (O<<1) + AIN * BIN[6];
        2: O <= (O<<1) + AIN * BIN[5];
        3: O <= (O<<1) + AIN * BIN[4];
        4: O <= (O<<1) + AIN * BIN[3];
        5: O <= (O<<1) + AIN * BIN[2];
        6: O <= (O<<1) + AIN * BIN[1];
        7: begin O <= (O<<1) + AIN * BIN[0]; fin<= 1; end
        8: fin <= 0;
        endcase
        st <= st + 1;
    end
end
endmodule
```

```
module mul(A,B,O,ck,start,fin);
input [7:0] A, B;
input ck,start;
output [16:0] O;
output fin;
reg [3:0] st;
reg [7:0] AIN, BIN;
reg [16:0] O, Y;
reg fin;
always @(posedge ck) begin
    if( start == 1 ) begin
        st <= 0;
        fin <= 0;
        AIN <= A;
        BIN <= B;
        Y <= 0;
    end else begin
        case (st)
        0: Y <= (Y<<1) + (BIN[7]==1 ? AIN : 0);
        1: Y <= (Y<<1) + (BIN[6]==1 ? AIN : 0);
        2: Y <= (Y<<1) + (BIN[5]==1 ? AIN : 0);
        3: Y <= (Y<<1) + (BIN[4]==1 ? AIN : 0);
        4: Y <= (Y<<1) + (BIN[3]==1 ? AIN : 0);
        5: Y <= (Y<<1) + (BIN[2]==1 ? AIN : 0);
        6: Y <= (Y<<1) + (BIN[1]==1 ? AIN : 0);
        7: begin O <= (Y<<1) + (BIN[0]==1 ? AIN : 0); fin<= 1; end
        8: fin <= 0;
        endcase
        st <= st + 1;
    end
end
endmodule
```

```
module mul(A,B,O,ck,start,fin);
input [7:0] A, B;
input ck,start;
output [16:0] O;
output fin;
reg [3:0] st;
reg [7:0] AIN, BIN;
reg [16:0] Y;
reg fin;
assign O = (fin == 1 ? Y : 'b 0);
always @(posedge ck) begin
    if( start == 1 ) begin
        st <= 0;
        fin <= 0;
        AIN <= A;
        BIN <= B;
        Y <= 0;
    end else begin
        case (st)
        0: Y <= (Y<<1) + (BIN[7]==1 ? AIN : 0);
        1: Y <= (Y<<1) + (BIN[6]==1 ? AIN : 0);
        2: Y <= (Y<<1) + (BIN[5]==1 ? AIN : 0);
        3: Y <= (Y<<1) + (BIN[4]==1 ? AIN : 0);
        4: Y <= (Y<<1) + (BIN[3]==1 ? AIN : 0);
        5: Y <= (Y<<1) + (BIN[2]==1 ? AIN : 0);
        6: Y <= (Y<<1) + (BIN[1]==1 ? AIN : 0);
        7: begin Y <= (Y<<1) + (BIN[0]==1 ? AIN : 0); fin<= 1; end
        8: fin <= 0;
        endcase
        st <= st + 1;
    end
end
endmodule
```

計算の途中で出力が
遷移しないようにする

途中を出力
しない

finのタイミングとOへの代入のタイミングがずれないように注意

課題5 mul.v (multest2.vを使用) 演習5:乗算の実装・・・しつこい続

```
module mul(A,B,O,ck,start,fin);
input [7:0] A, B;
input ck,start;
output [16:0] O;
output fin;
reg [2:0] st;
reg [7:0] AIN, BIN;
reg [16:0] Y;
reg fin;
assign O = (fin == 1 ? Y : 'b 0);
always @(posedge ck) begin
    if (start == 1) begin
        st <= 7;
        fin <= 0;
        AIN <= A;
        BIN <= B;
        Y <= 0;
    end else begin
        if (st==0) fin<= 1;
        else fin <= 0;
        Y <= (Y<<1) + (BIN[7]==1 ? AIN : 0);
        BIN <= BIN<<1;
        st <= st - 1;
    end
end
endmodule
```

演習5: 蛇足: パラメータ化

mul3.v

```
module mul(A,B,O,ck,start,fin);
```

```
  parameter wA=16;  
  parameter wB=16;  
  parameter wS=4;
```

```
  input [wA-1:0] A;  
  input [wB-1:0] B;  
  output [wA+wB:0] O;  
  reg [wS-1:0] st;
```

```
  if( start == 1 ) begin
```

```
    st <= {wA {1'b1}};
```



multest3.v

```
`define wwA 8  
`define wwB 8  
`define wwS 3
```

```
module multest;
```

```
  reg [wwA-1:0] A;  
  reg [wwB-1:0] B;  
  reg [wwS-1:0] st;  
  wire [wwA+wwB:0] O;  
  reg [wwA+wwB:0] OR;
```

```
  defparam MUL.wA=`wwA;  
  defparam MUL.wB=`wwB;  
  defparam MUL.wS=`wwS;
```

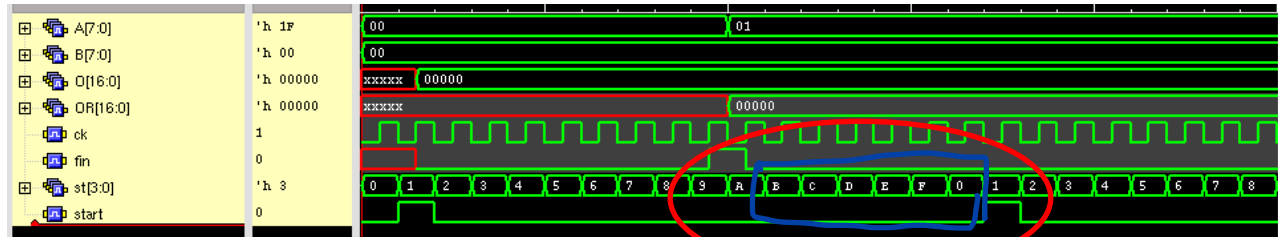
verilog + define + wwA =

バッチワートン

multest2.v

課題5のテストベンチ

```
module multest;
reg [7:0] A, B;
reg [3:0] st;
.....
initial begin
.....
end
```



```
mul MUL(A, B, O, ck, start, fin)
always @(negedge ck) begin
```

stが0になるまで待つて(stは4ビットなので乗算は16
クロック以内に終わることを前提としている、そうでな
いとテストベンチが誤動作)演算実施(=start→1)

```
if( st == 0 ) start <= 1;
```

```
else start <= 0;
```

```
if( fin == 1 ) begin
```

```
OR <= O;
```

```
st <= 0;
```

```
{B,A} <= {B,A} + 1;
```

```
if( O != A*B ) $finish;
```

```
if( A == 'h f && B == 'h f ) begin
```

```
$display( "OK¥n" ); $finish;
```

```
end
```

```
end else st <= st+1;
```

```
end
```



finが出力されると即次の演
算実施(=start→1)

fifo.v 課題6 fifo.vの完成(simfifo.vを使用)

```
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;
```

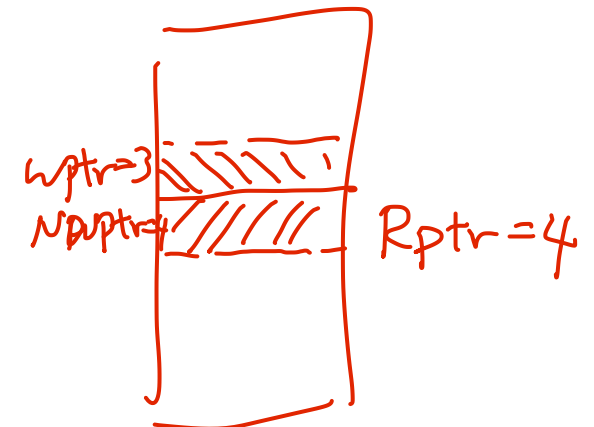
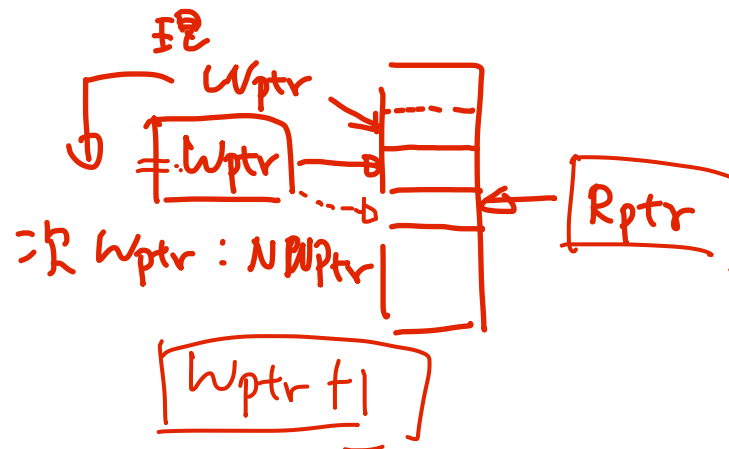
```
reg [7:0] FMEM[0:15];
reg [3:0] Wptr, Rptr;
reg Fempty, Ffull;
reg [7:0] obuf;
wire [3:0] NWptr, NRptr;
assign Dout = obuf;
assign NWptr = Wptr + 1;
assign NRptr = Rptr + 1;
```

```
always @(posedge ck) begin
  if( !rst ) begin
    Wptr <= 0;
    Rptr <= 0;
    Fempty <= 1;
    Ffull <= 0;
  end else begin
    if( Ren == 1 && Fempty != 1 ) begin
      obuf <= FMEM[Rptr];
      Rptr <= NRptr;
      Ffull <= 0;
      if( NRptr == Wptr ) Fempty <= 1;
      else Fempty <= 0;
    end
    if( Wen == 1 && Ffull != 1 ) begin
      FMEM[Wptr] <= Din;
      Wptr <= Wptr + 1;
      Fempty <= 0;
      if( NWptr == Rptr ) Ffull <= 1;
      else Ffull <= 0;
    end
  end
end
endmodule
```

メモリの中身をシミュレーションで参照する仕組み

output Dout;
reg Dout;
[7:0]

```
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];
```

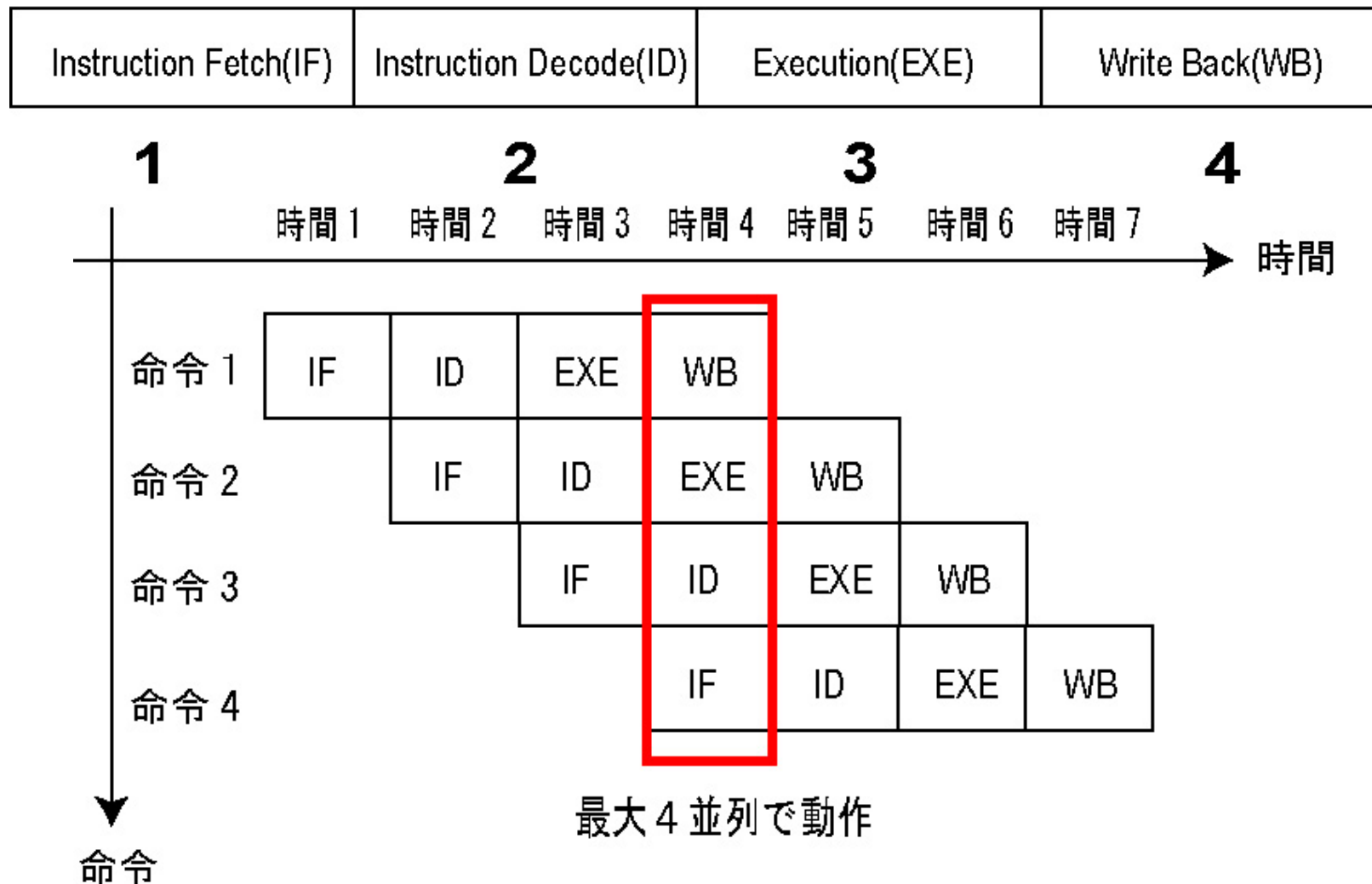


演習7－5：発展課題： CPUのパイプライン化

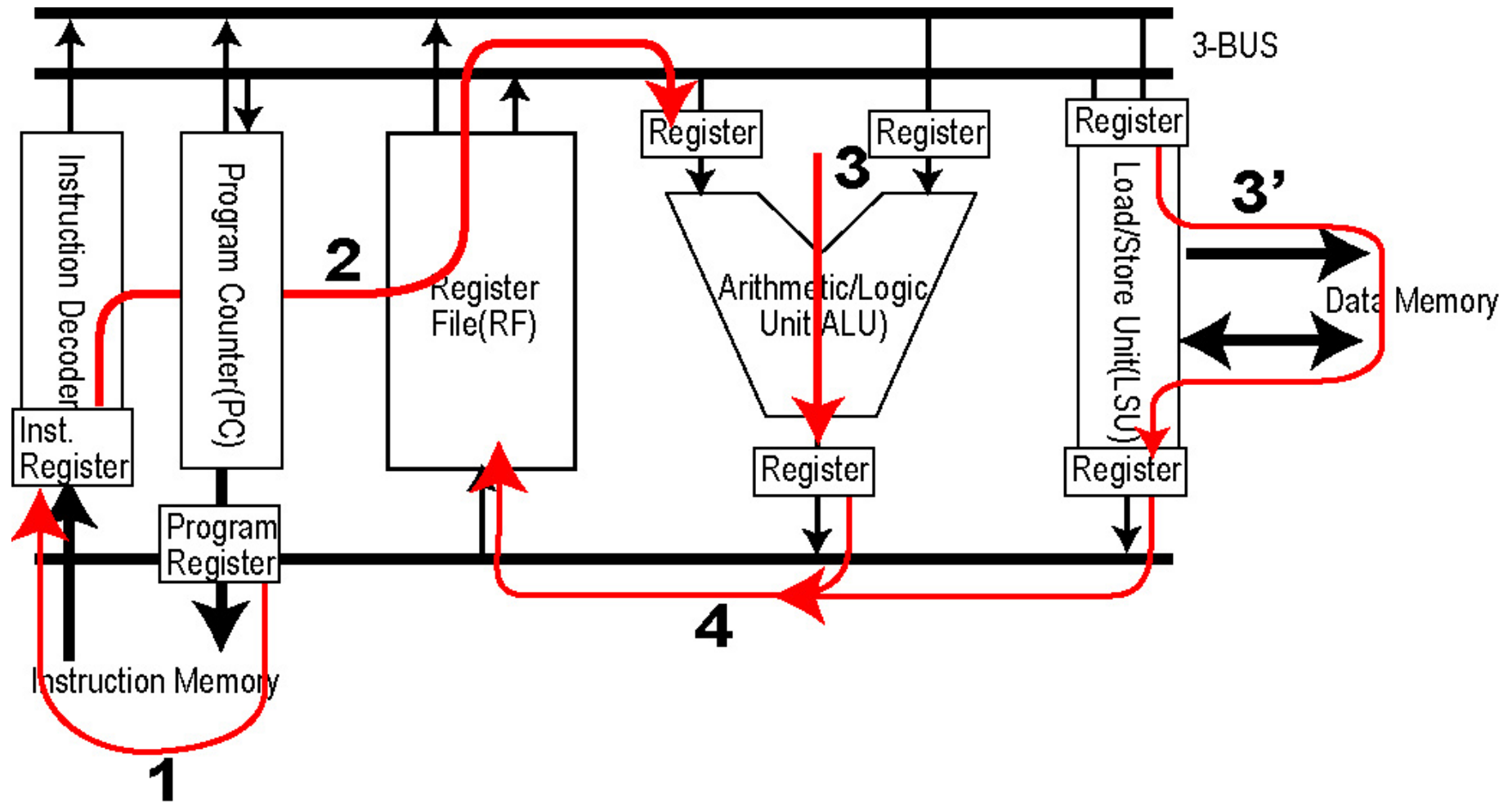
- パイプライン化について検討する

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
 - キャッシュ、演算終了時間制御

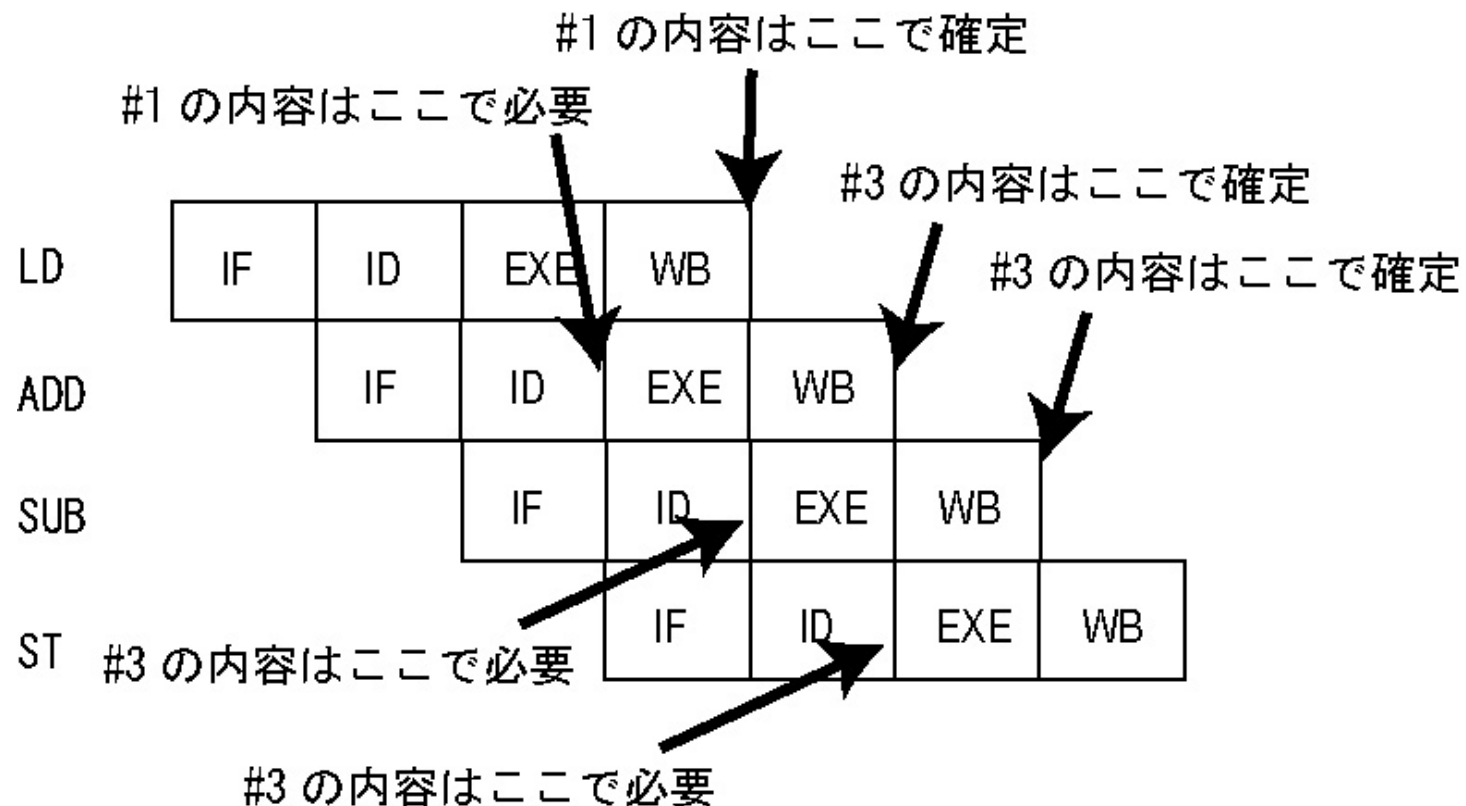
データハザードとは

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



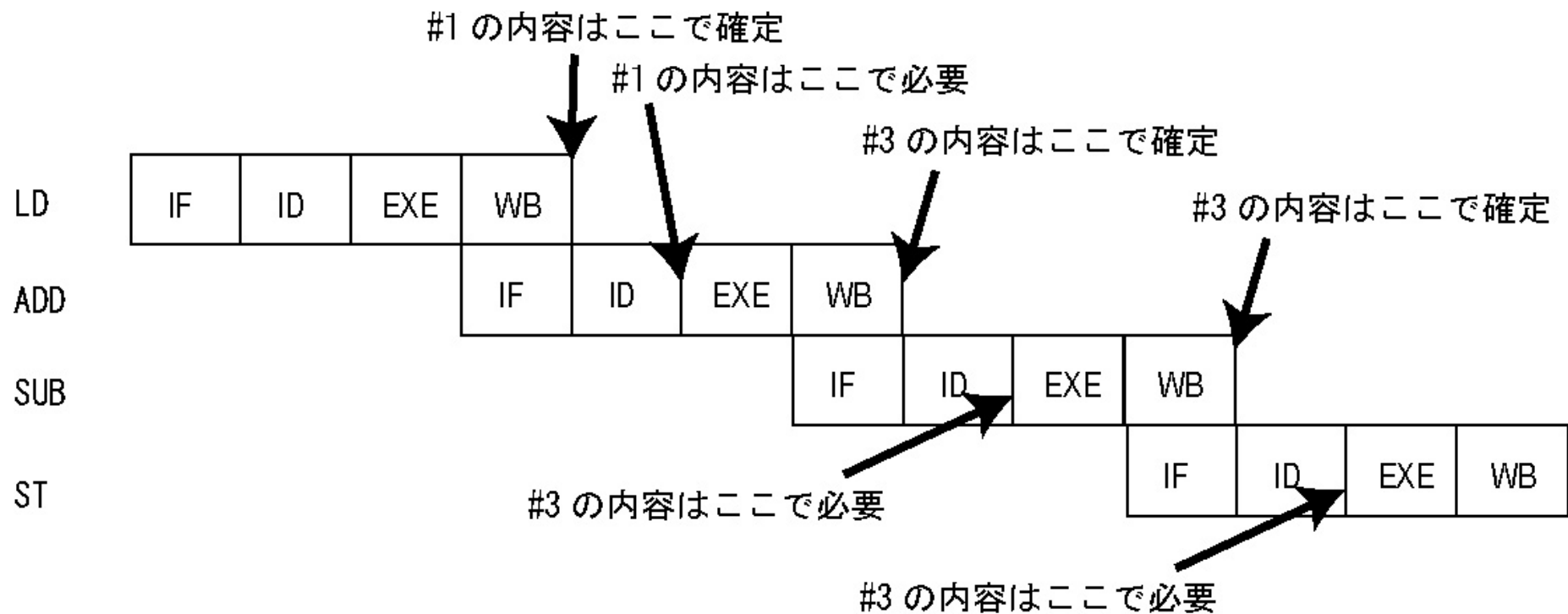
データハザードの解決

LD #1, [10] :レジスタ1に10番地の内容を読み込み

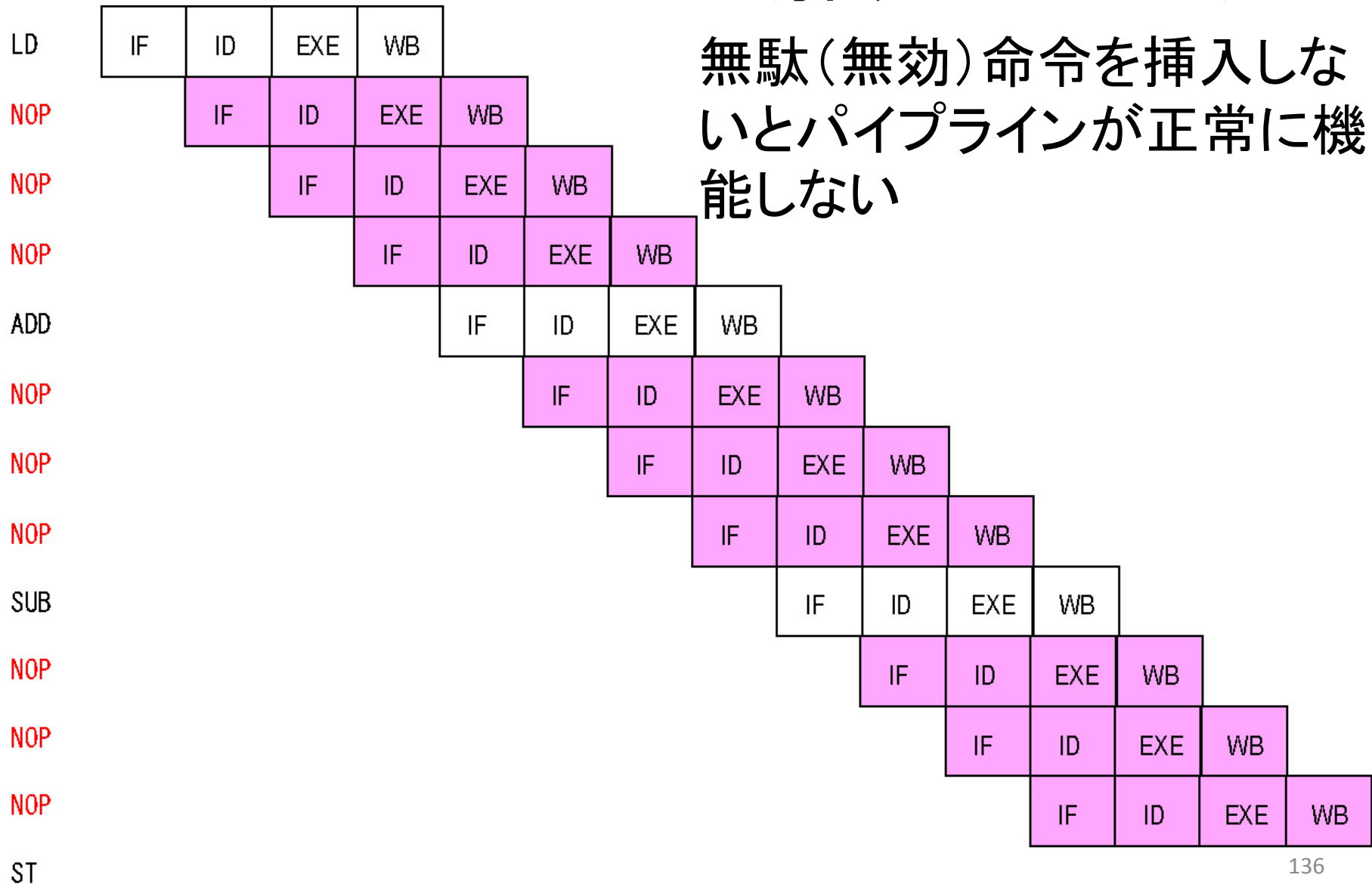
ADD #3, #1, #3 : レジスタ1, 2を加算し3に書き込む

SUB #3, #3, #4 : レジスタ3, 4を減算し3に書き込む

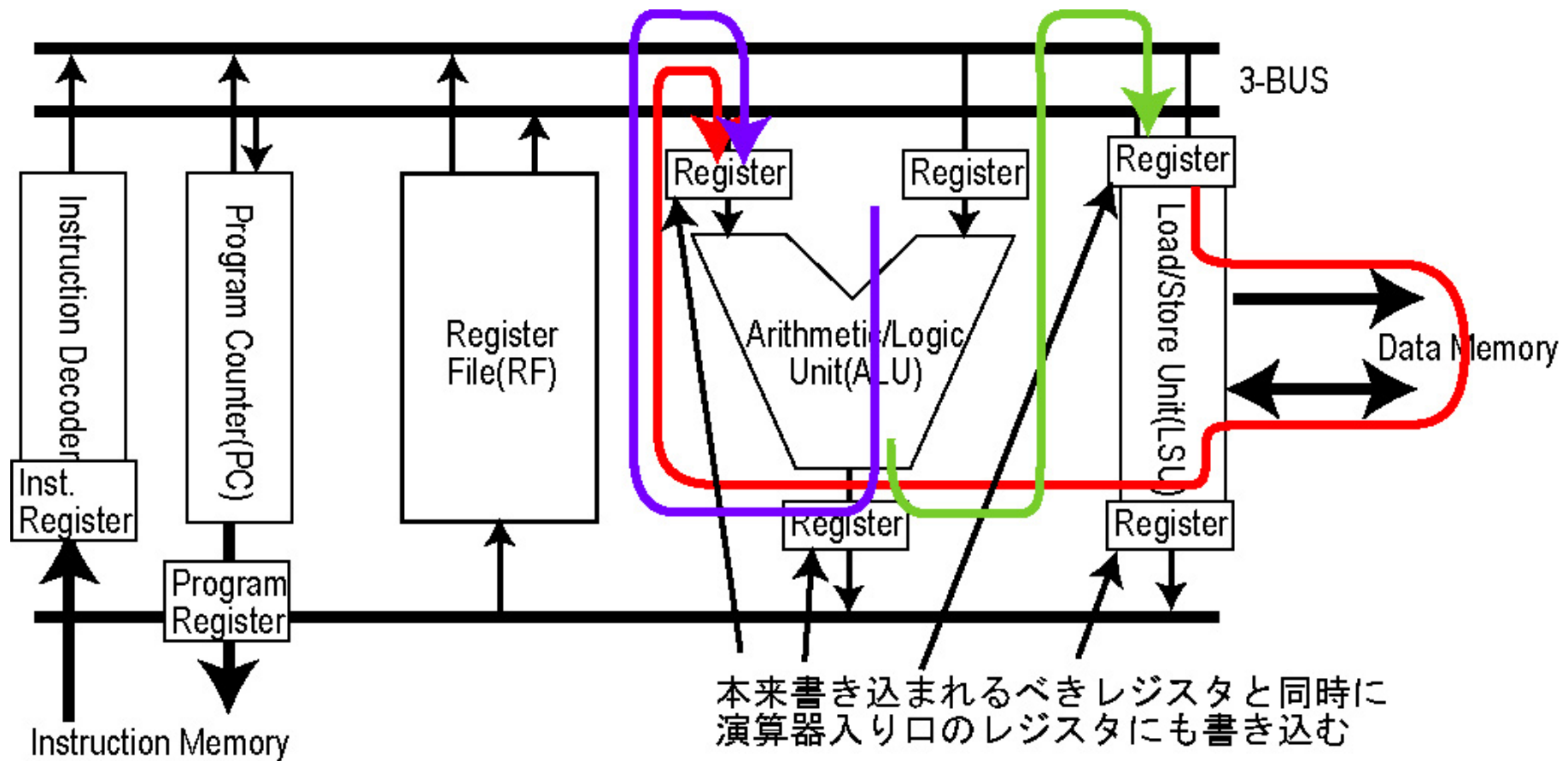
ST #3, [11] :レジスタ3を11番地に書き出す



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

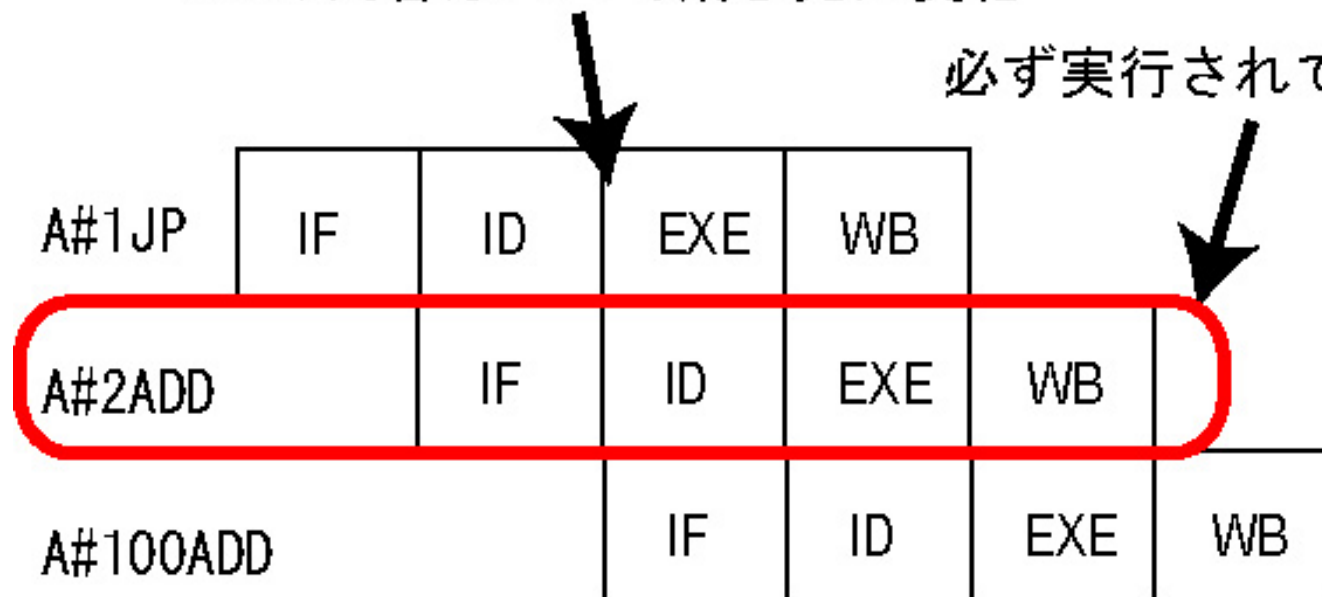
A#1 JP A#100 :100番地にジャンプ

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

必ず実行されてしまう



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 else if( STAGE == 1 ) begin
```



```
      end
```

A_st1 <= A_st0

```
      STAGE <= 2;
```

```
    end else if( STAGE == 2 ) begin
```



```
      STAGE <= 3;
```

A_st2 <= A_st1

```
    end else if( STAGE == 3 ) begin
```



```
      STAGE <= 0;
```

A_st3 <= A_st2

```
    end
```

```
  end
```

```
end
```

```
endmodule
```