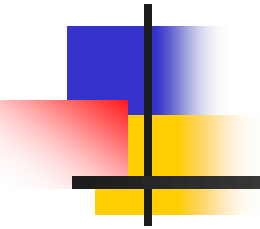
Department of Computer Science
Course number: CSC.T341



コンピュータ論理設計
Computer Logic Design

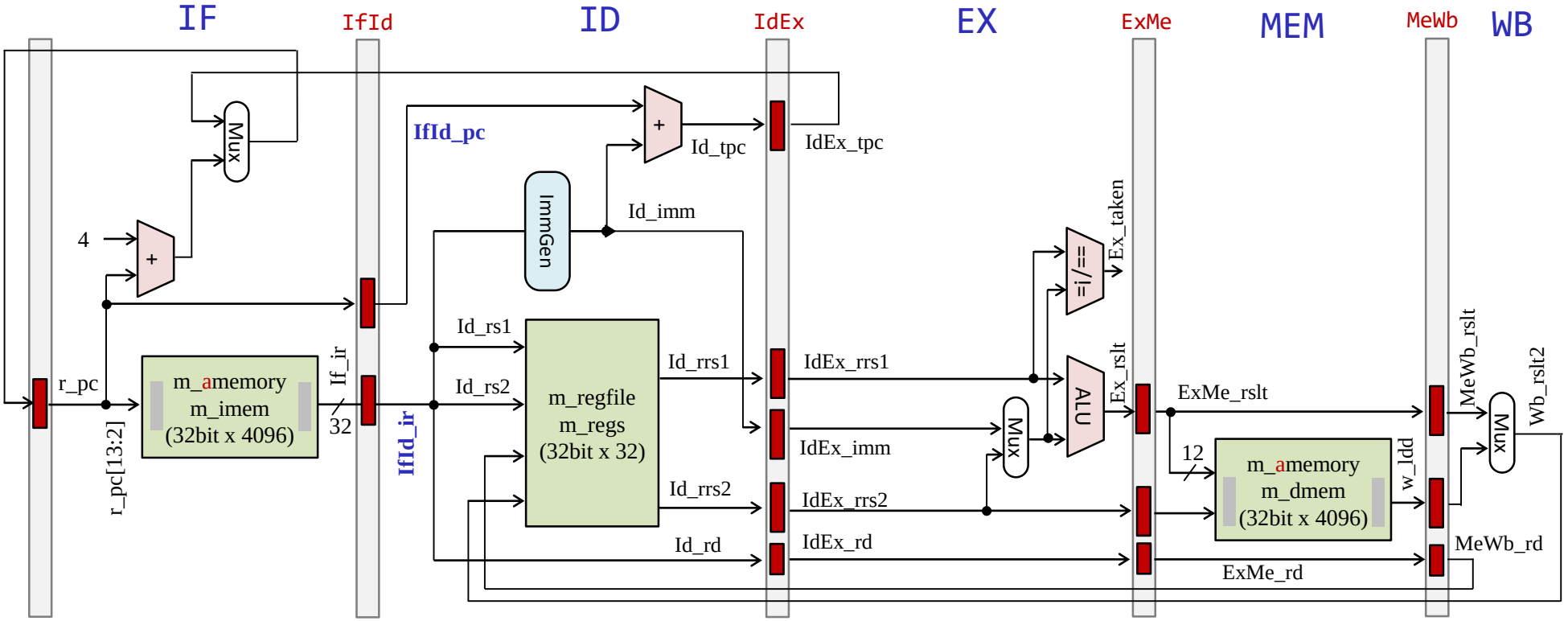
12. パイプラインプロセッサとハザード処理 (2)
Pipelining Processor and Hazards (2)

吉瀬 謙二　情報工学系
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp　www.arch.cs.titech.ac.jp/lecture/CLD/
講義：月曜日 10:45-12:25, 木曜日 10:45-12:25

Ver. 2023-05-22a

m_proc14 5段のパイプライン版

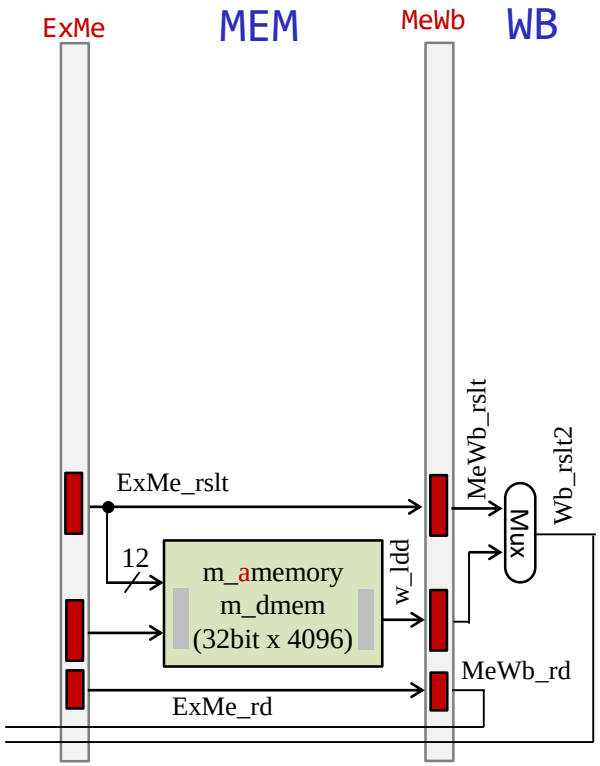
- add, addi, sll, srl, lw, sw, beq, bne命令に対応したプロセッサ（データフォワーディング無し）
- IF, ID, EX, MEM, WB のそれぞれをステージとする5段のパイプラインプロセッサ.
- m_amemory を利用しているが，その出力がレジスタに接続される dmem は同期メモリとして動作する. このため，BRAMが利用される.



code185.v

m_proc14 5段のパイプライン版

- m_amemory を利用しているが，その出力がレジスタに接続される dmem は同期メモリとして動作する．このため，BRAMが利用される．



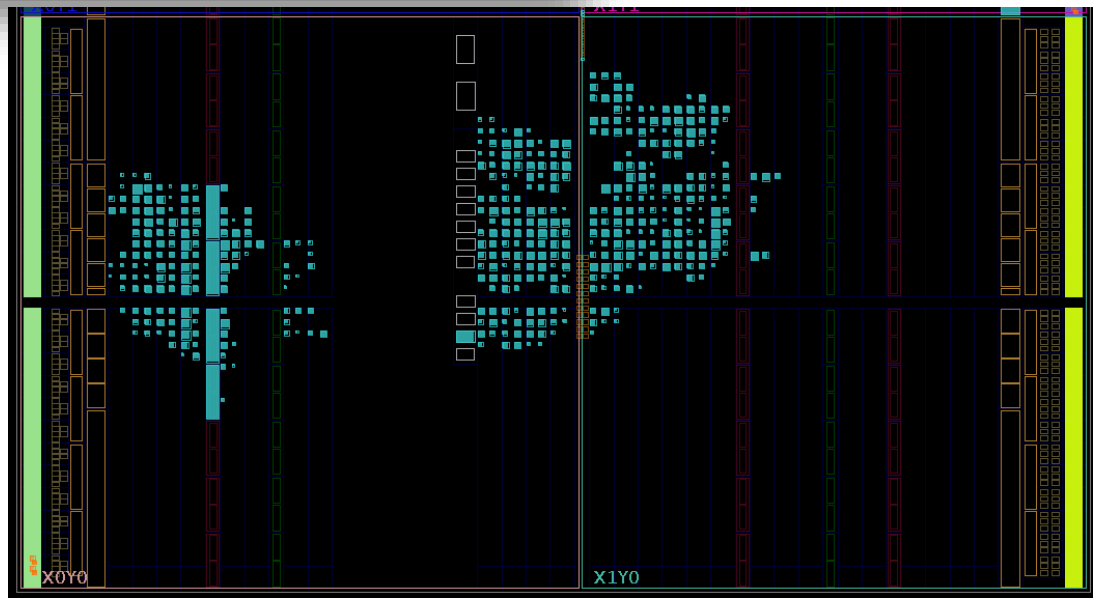

```
/********************** Me stage ********************************/
  wire [4:0] Me_op5 = ExMe_ir[6:2];
  wire       Me_we = (Me_op5==5'b01100 || Me_op5==5'b00100 || Me_op5==5'b00000);
  wire [31:0] Me_ldd;
  m_amemory m_dmem (w_clk, ExMe_rslt[13:2], (Me_op5==5'b01000), ExMe_rrs2, Me_ldd);
  always @(posedge w_clk) #5 if(w_ce) begin
     MeWb_pc   <= ExMe_pc;
     MeWb_ir   <= ExMe_ir;
     MeWb_rslt <= ExMe_rslt;
     MeWb_ldd  <= Me_ldd;
     MeWb_rd   <= ExMe_rd;
  end
  /********************** Wb stage ********************************/
  wire Wb_LW = (MeWb_ir[6:2]==5'b00000);
  wire [31:0] Wb_rslt2 = (Wb_LW) ? MeWb_ldd : MeWb_rslt;
  always @(posedge w_clk) #5
    if(w_ce && IfId_ir!=32'h000f0033) r_pc <= (Ex_taken) ? IdEx_tpc : r_pc+4;

  reg [31:0] r_led = 0;
  always @(posedge w_clk) if(w_ce & MeWb_rd==30) r_led <= Wb_rslt2;
  assign w_led = r_led;
endmodule
```

/home/tu_kise/cld/2023/code185.v

m_proc14 5段のパイプライン版

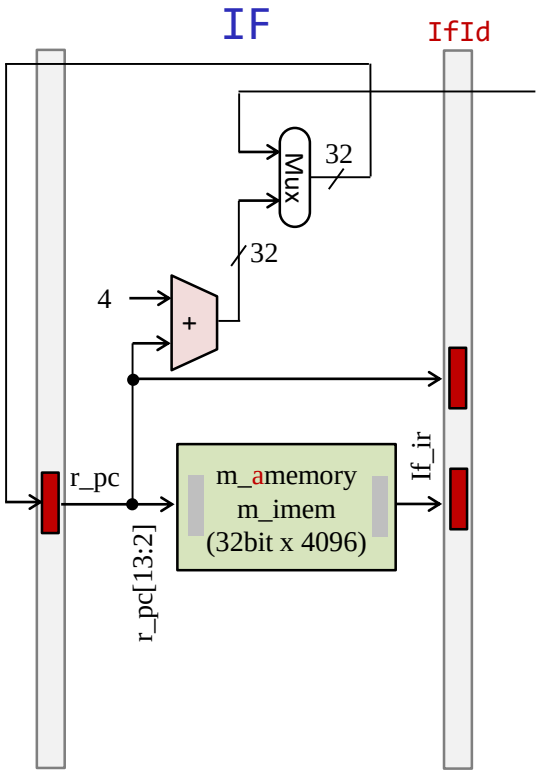
- add, addi, sll, srl, lw, sw, beq, bne命令に対応したプロセッサ（データフォワーディング無し）
- IF, ID, EX, MEM, WB のそれぞれをステージとする5段のパイプラインプロセッサ.
- m_amemory を利用しているが，その出力がレジスタに接続される dmem は同期メモリとして動作する．このため，BRAMが利用される.

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	Block RAM Tile (50)
N m_main	1042	1445	40	20	462	894	148	4
clk_w0 (clk_wiz_0)	0	0	0	0	0	0	0	0
dbg_hub (dbg_hub)	448	741	0	0	228	424	24	0
p (m_proc14)	381	292	40	20	138	257	124	4
m_dmem (m_amem	33	0	0	0	14	33	0	4
m_imem (m_amemc	100	0	40	20	29	20	80	0
m_regs (m_regfile)	44	0	0	0	11	0	44	0
vio_00 (vio_0)	205	408	0	0	108	205	0	0



m_proc14 5段のパイプライン版

- ステージ IF と ID の間のパイプラインレジスタには IfId_ から始まる名前を利用する.
- ステージ ID で生成される配線には Id_ から始まる名前を利用する.




```
/********************** IF stage *******************************/
 wire [31:0] If_ir;
 m_amemory m_imem (w_clk, r_pc[13:2], 1'd0, 32'd0, If_ir);
 always @(posedge w_clk) #5 if(w_ce) begin
   IfId_pc <= (Ex_taken) ? 0 : r_pc;
   IfId_ir <= (Ex_taken) ? {25'd0, 7'b0010011} : If_ir;
 end
 /********************** ID stage *******************************/
 wire [4:0] Id_op5 = IfId_ir[6:2];
 wire [4:0] Id_rs1 = IfId_ir[19:15];
 wire [4:0] Id_rs2 = IfId_ir[24:20];
 wire [4:0] Id_rd  = IfId_ir[11:7];
 wire Id_we = (Id_op5==5'b01100 || Id_op5==5'b00100 || Id_op5==5'b00000);
 wire [31:0] Id_imm, Id_rrs1, Id_rrs2;
 m_immgen m_immgen0 (IfId_ir, Id_imm);
 m_regfile m_regs (w_clk, Id_rs1, Id_rs2, MeWb_rd, 1'b1, Wb_rslt2,
    Id_rrs1, Id_rrs2);
 always @(posedge w_clk) #5 if(w_ce) begin
   IdEx_pc   <= (Ex_taken) ? 0 : IfId_pc;
   IdEx_ir   <= (Ex_taken) ? {25'd0, 7'b0010011} : IfId_ir;
   IdEx_tpc  <= IfId_pc + Id_imm;
   IdEx_imm  <= Id_imm;
   IdEx_rrs1 <= Id_rrs1;
   IdEx_rrs2 <= Id_rrs2;
   IdEx_rd   <= (Id_we==0 || Ex_taken) ? 0 : Id_rd; // Note
 end
```

/home/tu_kise/cld/2023/code185.v

Hazards make pipelining hard

- 命令を適切なサイクルで実行できないような状況が存在する．これをハザード(hazard)と呼ぶ．
  - 構造ハザード (structural hazard)
    - オーバラップ実行する命令の組み合わせをハードウェアがサポートしていない場合．資源不足により生じる．
  - データ・ハザード(data hazard)
    - データの受け渡しの制約によって生じるハザード

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //    add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd4, 7'b0010011}; //    addi x4, x0, 3   // x4 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //    addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd5, 5'd4, 3'b000, 5'd30,7'b0110011}; //    add  x30,x4, x5  // led = x4 + x5
```

  - 制御ハザード(control hazard)
    - 分岐命令，ジャンプ命令によって生じるハザード

```
cm_ram[1]={12'd5,      5'd0, 3'b000, 5'd4, 7'b0010011};     //    addi x4, x0, 5   // x4 = 5
cm_ram[2]={12'd1,      5'd0, 3'b000, 5'd5, 7'b0010011};     //    addi x5, x0, 1   // x5 = 1
cm_ram[3]={12'd1,      5'd5, 3'b000, 5'd5, 7'b0010011};     // L1:addi x5, x5, 1   // x5 = x5 + 1
cm_ram[4]={7'h7f,5'd4, 5'd5, 3'b001, 5'b11101, 7'b1100011}; //    bne  x5, x4, L1  // goto L1 if x5!=x4
cm_ram[5]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011};     //    add  x30,x5, x0  // led = x5
```

プロセッサが命令を処理するための基本的な5つのステップ

- IF (Instruction Fetch)
  メモリから命令をフェッチする.
- ID (Instruction Decode)
  命令をデコード(解読)しながら, レジスタファイルの値を読み出す (Operand Fetch)
- EX (Execution)
  命令操作の実行またはアドレスの生成を行う.
- MEM (Memory Access)
  必要であれば, メモリ(データ・メモリ)のオペランドにアクセスする.
- WB (Write Back)
  必要であれば, 結果をレジスタファイルに書き込む.

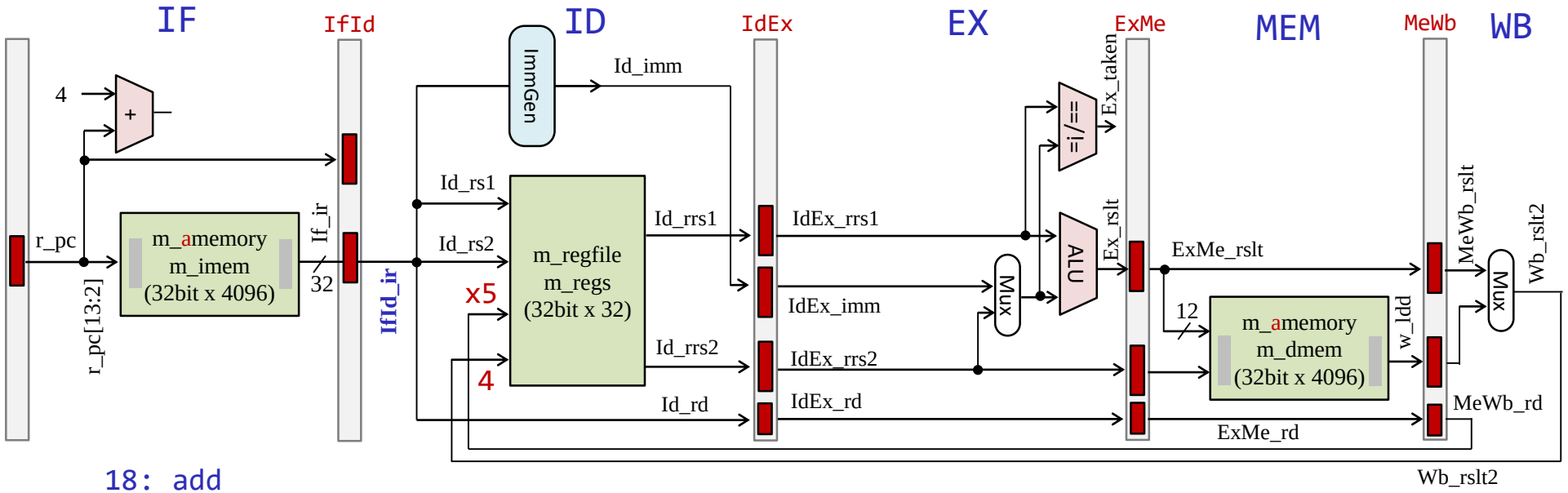
(1) m_proc14 5段のパイプライン版：データ依存の無い例

- 命令の間に3個の NOP 命令を挿入したプログラム.

program7.txt

CC7

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  00:   add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  04:   addi x5, x0, 3   // x5 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  08:   addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  0c:   add  x0, x0, x0  // NOP
cm_ram[4]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  10:   add  x0, x0, x0  // NOP
cm_ram[5]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  14:   add  x0, x0, x0  // NOP
cm_ram[6]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011}; //  18:   add  x30,x5, x0  // led = x5
cm_ram[7]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  1c:   add  x0, x0, x0  // NOP
```



18: add x30,x5,x0

14: NOP

10: NOP

0c: NOP

08: addi x5,x0,4

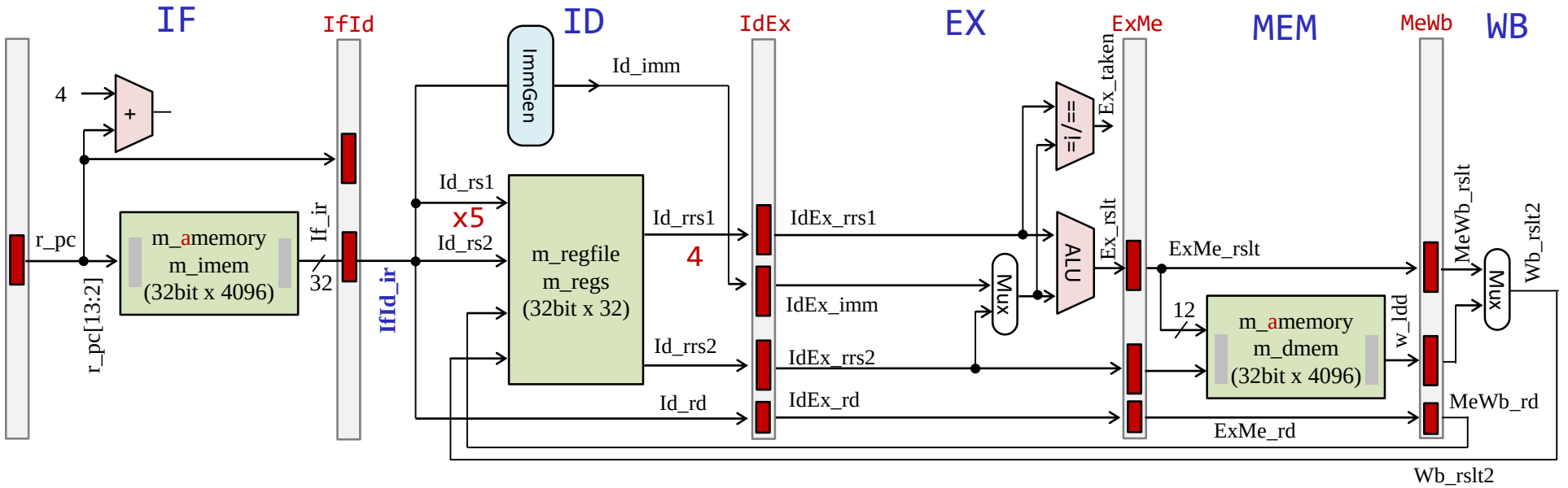
(1) m_proc14 5段のパイプライン版：データ依存の無い例

- IDステージの命令 18: add x30,x5,x0 は正しいレジスタの値4を読み出すことができる.

program7.txt

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  00:   add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  04:   addi x5, x0, 3   // x5 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  08:   addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  0c:   add  x0, x0, x0  // NOP
cm_ram[4]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  10:   add  x0, x0, x0  // NOP
cm_ram[5]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  14:   add  x0, x0, x0  // NOP
cm_ram[6]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011}; //  18:   add  x30,x5, x0  // led = x5
cm_ram[7]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  1c:   add  x0, x0, x0  // NOP
```

CC8



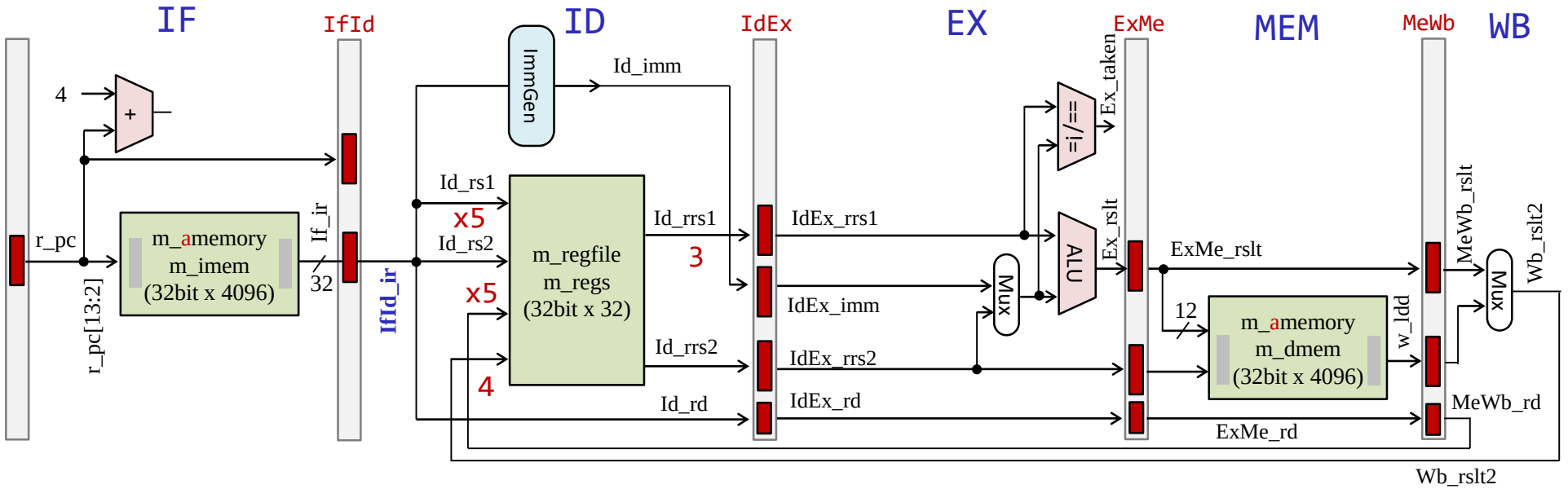
(2) m_proc14 5段のパイプライン版：レジスタバイパス

- 命令の間に2個の NOP 命令を挿入したプログラム.
- IDステージの命令 14: add x30,x5,x0 は正しくないレジスタの値3を読み出してしまう.

program8.txt

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  00:   add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  04:   addi x5, x0, 3   // x5 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  08:   addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  0c:   add  x0, x0, x0  // NOP
cm_ram[4]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  10:   add  x0, x0, x0  // NOP
cm_ram[5]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011}; //  14:   add  x30,x5, x0  // led = x5
cm_ram[6]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  18:   add  x0, x0, x0  // NOP
```

CC7



14: add x30,x5,x0

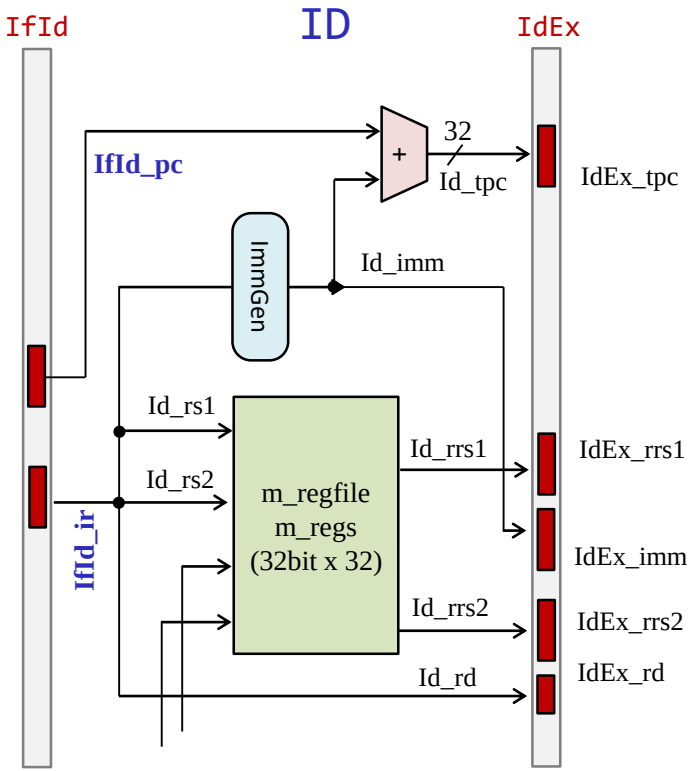
10: NOP

0c: NOP

08: addi x5,x0,4

m_proc14 5段のパイプライン版

- ステージ IF と ID の間のパイプラインレジスタには IfId_ から始まる名前を利用する.
- ステージ ID で生成される配線には Id_ から始まる名前を利用する.




```
/********************** IF stage *******************************/
 wire [31:0] If_ir;
 m_amemory m_imem (w_clk, r_pc[13:2], 1'd0, 32'd0, If_ir);
 always @(posedge w_clk) #5 if(w_ce) begin
   IfId_pc <= (Ex_taken) ? 0 : r_pc;
   IfId_ir <= (Ex_taken) ? {25'd0, 7'b0010011} : If_ir;
 end
 /********************** ID stage *******************************/
 wire [4:0] Id_op5 = IfId_ir[6:2];
 wire [4:0] Id_rs1 = IfId_ir[19:15];
 wire [4:0] Id_rs2 = IfId_ir[24:20];
 wire [4:0] Id_rd  = IfId_ir[11:7];
 wire Id_we = (Id_op5==5'b01100 || Id_op5==5'b00100 || Id_op5==5'b00000);
 wire [31:0] Id_imm, Id_rrs1, Id_rrs2;
 m_immgen m_immgen0 (IfId_ir, Id_imm);
 m_regfile m_regs (w_clk, Id_rs1, Id_rs2, MeWb_rd, 1'b1, Wb_rslt2,
                   Id_rrs1, Id_rrs2);
 always @(posedge w_clk) #5 if(w_ce) begin
   IdEx_pc   <= (Ex_taken) ? 0 : IfId_pc;
   IdEx_ir   <= (Ex_taken) ? {25'd0, 7'b0010011} : IfId_ir;
   IdEx_tpc  <= IfId_pc + Id_imm;
   IdEx_imm  <= Id_imm;
   IdEx_rrs1 <= Id_rrs1;
   IdEx_rrs2 <= Id_rrs2;
   IdEx_rd   <= (Id_we==0 || Ex_taken) ? 0 : Id_rd; // Note
 end
```

/home/tu_kise/cld/2023/code185.v

(2) レジスタファイルにバイパスの経路を追加

- 書き込みレジスタ番号と読み出しレジスタ番号が一致するときに，書き込む値を出力する．
  すなわち，書き込む値をバイパスするように修正したモジュール m_rf_bypass を用いる．

```
module m_regfile (w_clk, w_rr1, w_rr2, w_wr, w_we, w_wdata, w_rdata1, w_rdata2);
   input  wire        w_clk;
   input  wire [4:0]  w_rr1, w_rr2, w_wr;
   input  wire [31:0] w_wdata;
   input  wire        w_we;
   output wire [31:0] w_rdata1, w_rdata2;

   reg [31:0] r[0:31];
   assign #8 w_rdata1 = (w_rr1==0) ? 0 : r[w_rr1];
   assign #8 w_rdata2 = (w_rr2==0) ? 0 : r[w_rr2];
   always @(posedge w_clk) if(w_we) r[w_wr] <= w_wdata;
endmodule
```

```
module m_rf_byass (w_clk, w_rr1, w_rr2, w_wr, w_we, w_wdata, w_rdata1, w_rdata2);
   input  wire        w_clk;
   input  wire [4:0]  w_rr1, w_rr2, w_wr;
   input  wire [31:0] w_wdata;
   input  wire        w_we;
   output wire [31:0] w_rdata1, w_rdata2;

   reg [31:0] r[0:31];
   assign #8 w_rdata1 = (w_rr1==0) ? 0 : (w_rr1==w_wr & w_we) ? w_wdata : r[w_rr1];
   assign #8 w_rdata2 = (w_rr2==0) ? 0 : (w_rr2==w_wr & w_we) ? w_wdata : r[w_rr2];
   always @(posedge w_clk) if(w_we) r[w_wr] <= w_wdata;
endmodule
```

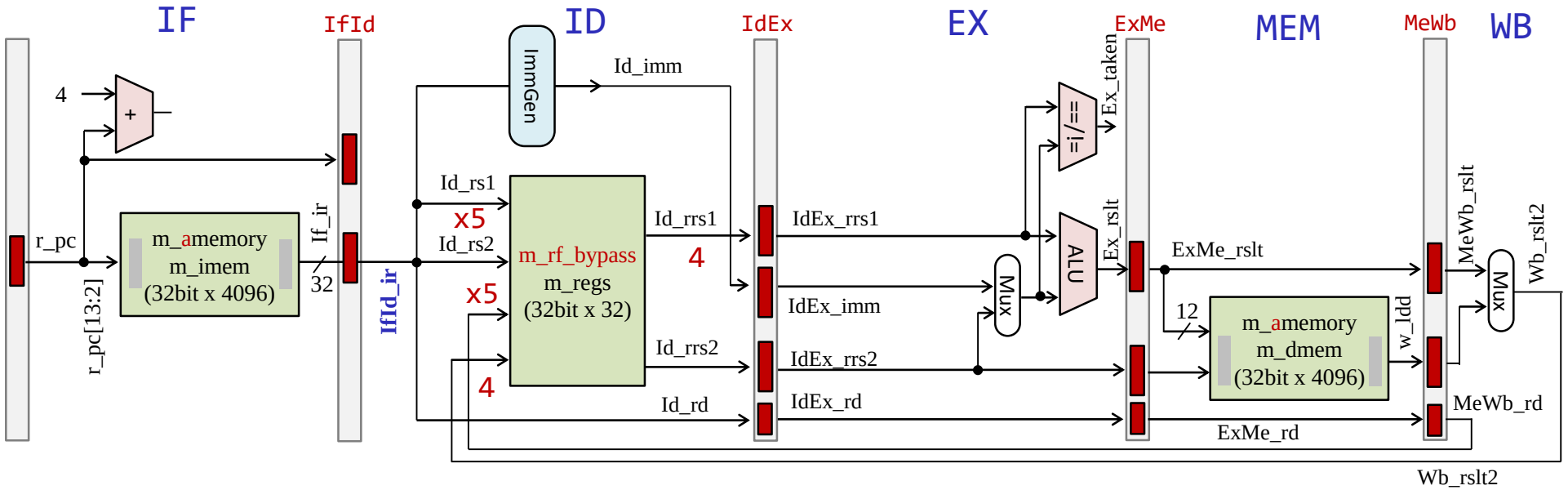
(2) m_proc14 5段のパイプライン版：レジスタバイパス

- 命令の間に2個の NOP 命令を挿入したプログラム．m_rf_bypass を用いる．
- IDステージの命令 14: add x30,x5,x0 は正しいレジスタの値4を読み出せる．

program8.txt

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  00:   add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd3, 7'b0010011}; //  04:   addi x5, x0, 3   // x5 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  08:   addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  0c:   add  x0, x0, x0  // NOP
cm_ram[4]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  10:   add  x0, x0, x0  // NOP
cm_ram[5]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011}; //  14:   add  x30,x5, x0  // led = x5
cm_ram[6]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  18:   add  x0, x0, x0  // NOP
```

CC7



14: add x30,x5,x0

10: NOP

0c: NOP

08: addi x5,x0,4

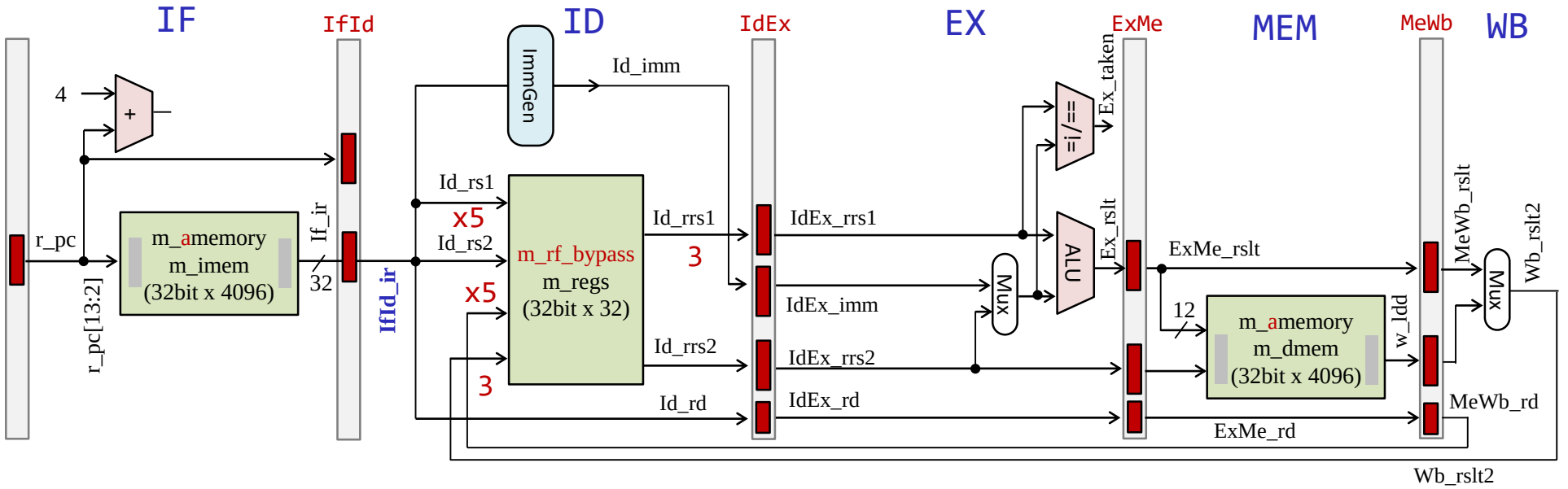
(3) m_proc14 5段のパイプライン版: WBフォワーディング

- 命令の間に1個の NOP 命令を挿入したプログラム.
- IDステージの命令 10: add x30,x5,x0 は正しくないレジスタの値3を読み出してしまう.

program9.txt

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  00:   add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  04:   addi x5, x0, 3   // x5 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  08:   addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  0c:   add  x0, x0, x0  // NOP
cm_ram[4]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011}; //  10:   add  x30,x5, x0  // led = x5
cm_ram[5]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  14:   add  x0, x0, x0  // NOP
```

CC6



10: add x30,x5,x0

0c: NOP

08: addi x5,x0,4

04: addi x5,x0,3

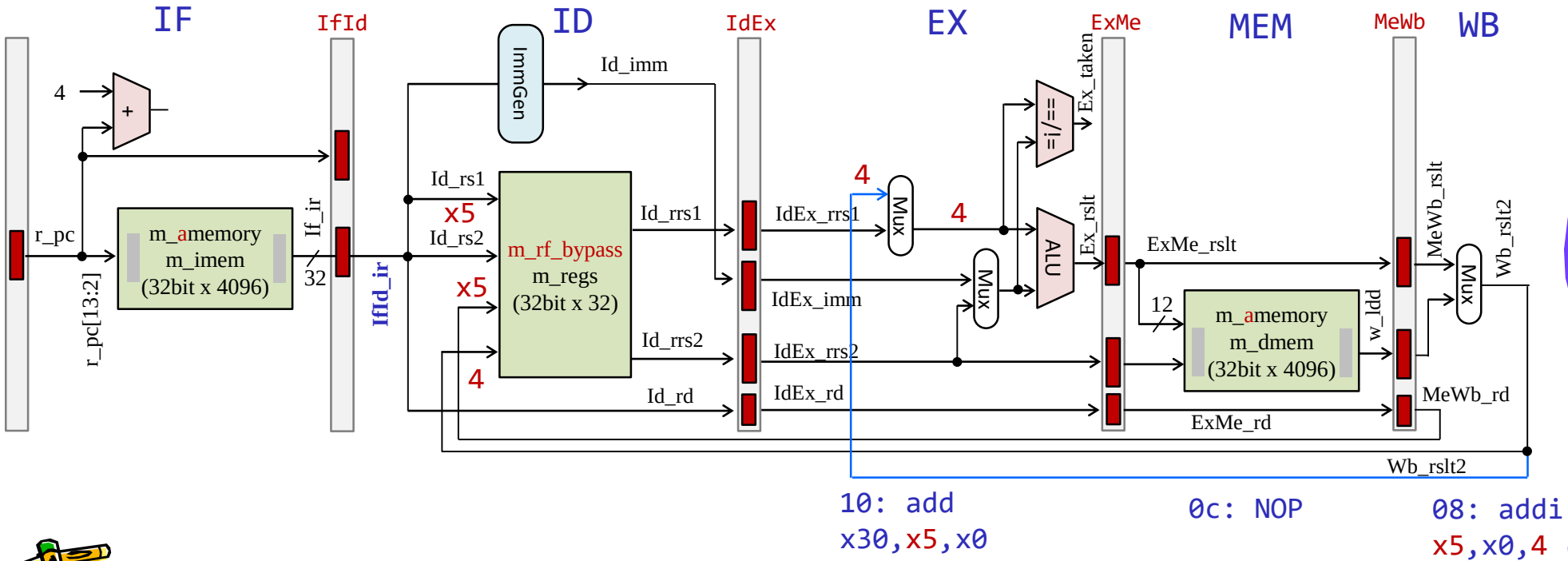
(3) m_proc14 5段のパイプライン版: WBフォワーディング

- 命令の間に1個の NOP 命令を挿入したプログラム.
- WBステージからオペランドをALUにフォワーディングする. 正しい値4を利用できる

program9.txt

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  00:   add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  04:   addi x5, x0, 3   // x5 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  08:   addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  0c:   add  x0, x0, x0  // NOP
cm_ram[4]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011}; //  10:   add  x30,x5, x0  // led = x5
cm_ram[5]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  14:   add  x0, x0, x0  // NOP
```

CC7



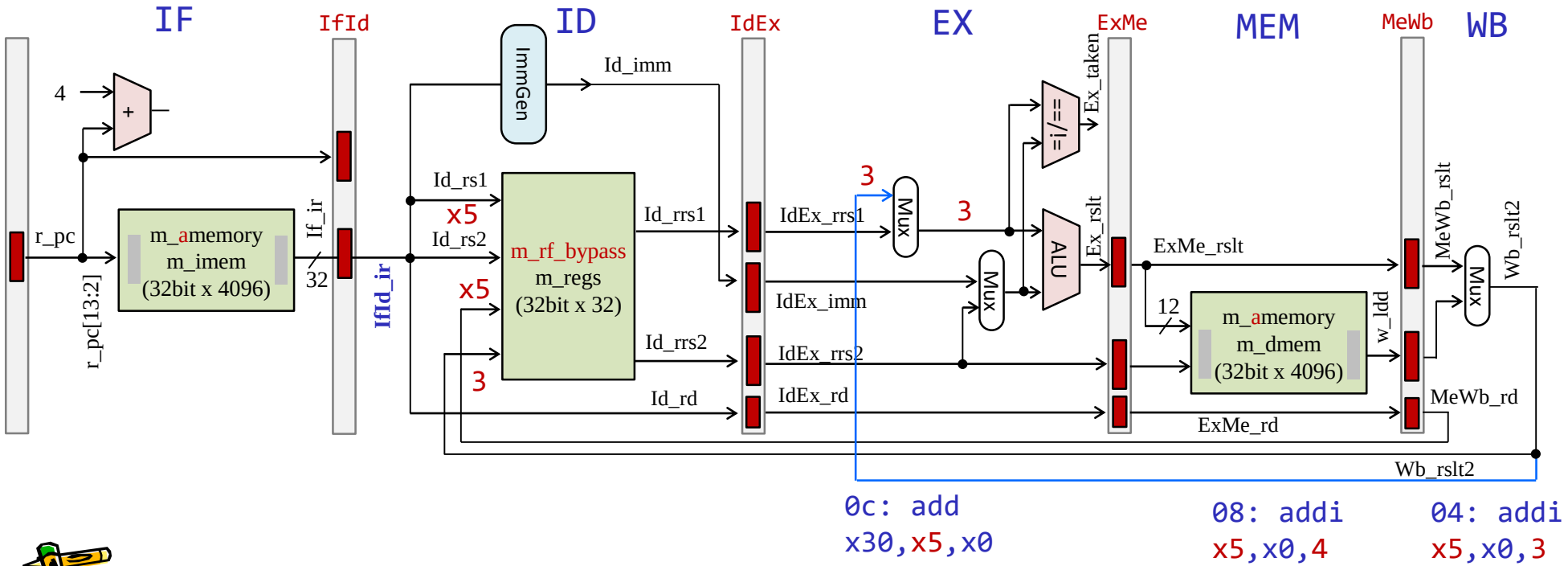
(4) m_proc14 5段のパイプライン版: MEMフォワーディング

- 命令の間に0個の NOP 命令を挿入したプログラム.
- Exステージの命令 0c: add x30,x5,x0 は正しくないレジスタの値3を読み出してしまう.

program10.txt

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  00:   add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  04:   addi x5, x0, 3   // x5 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  08:   addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011}; //  0c:   add  x30,x5, x0  // led = x5
cm_ram[4]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  10:   add  x0, x0, x0  // NOP
```

CC6



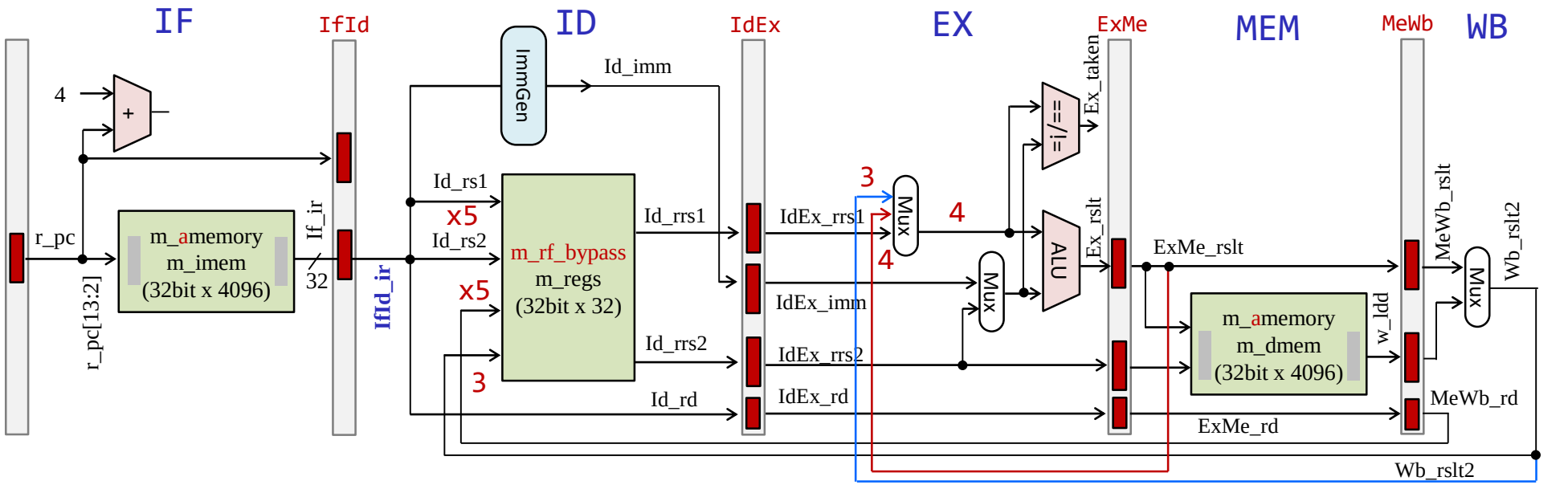
(4) m_proc14 5段のパイプライン版: MEMフォワーディング

- 命令の間に0個の NOP 命令を挿入したプログラム.
- MEMステージからオペランドをALUにフォワーディングする. 正しい値4を利用できる.

program10.txt

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  00:   add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  04:   addi x5, x0, 3   // x5 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  08:   addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011}; //  0c:   add  x30,x5, x0  // led = x5
cm_ram[4]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  10:   add  x0, x0, x0  // NOP
```

CC6



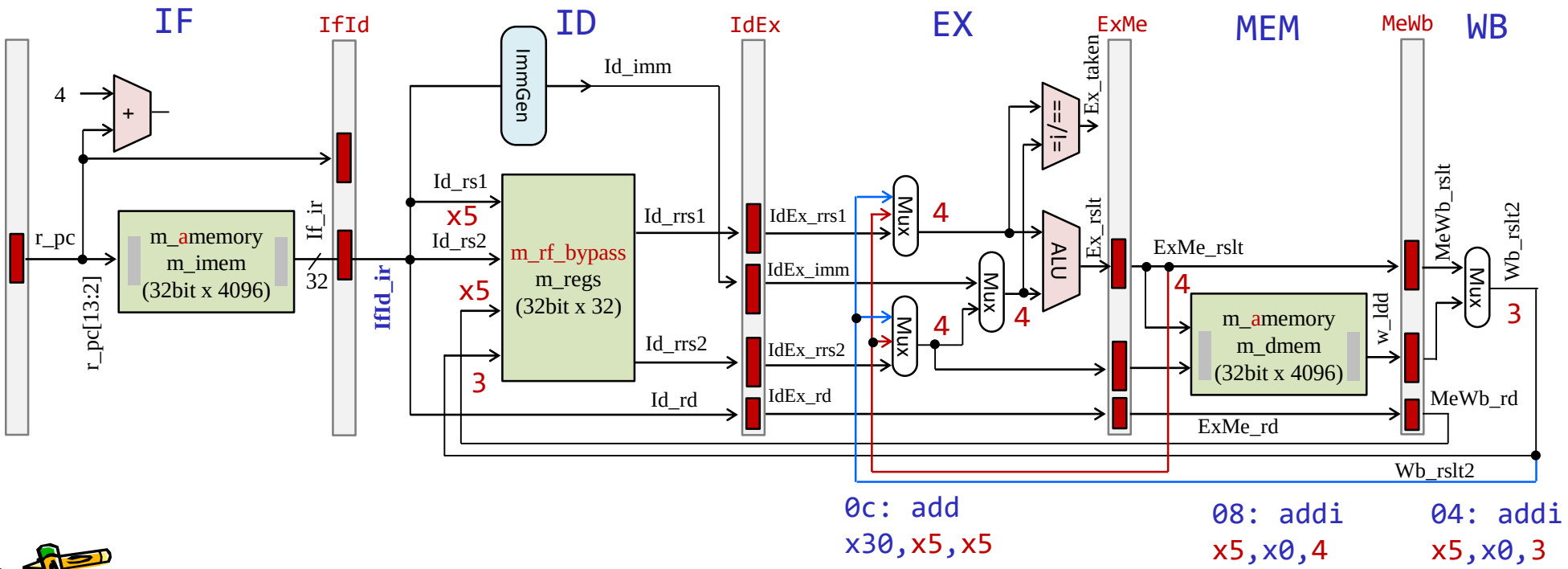
m_proc14 5段のパイプライン版

- rrs1 と同様に rrs2 のためにもデータをフォワーディングする.

program10.txt

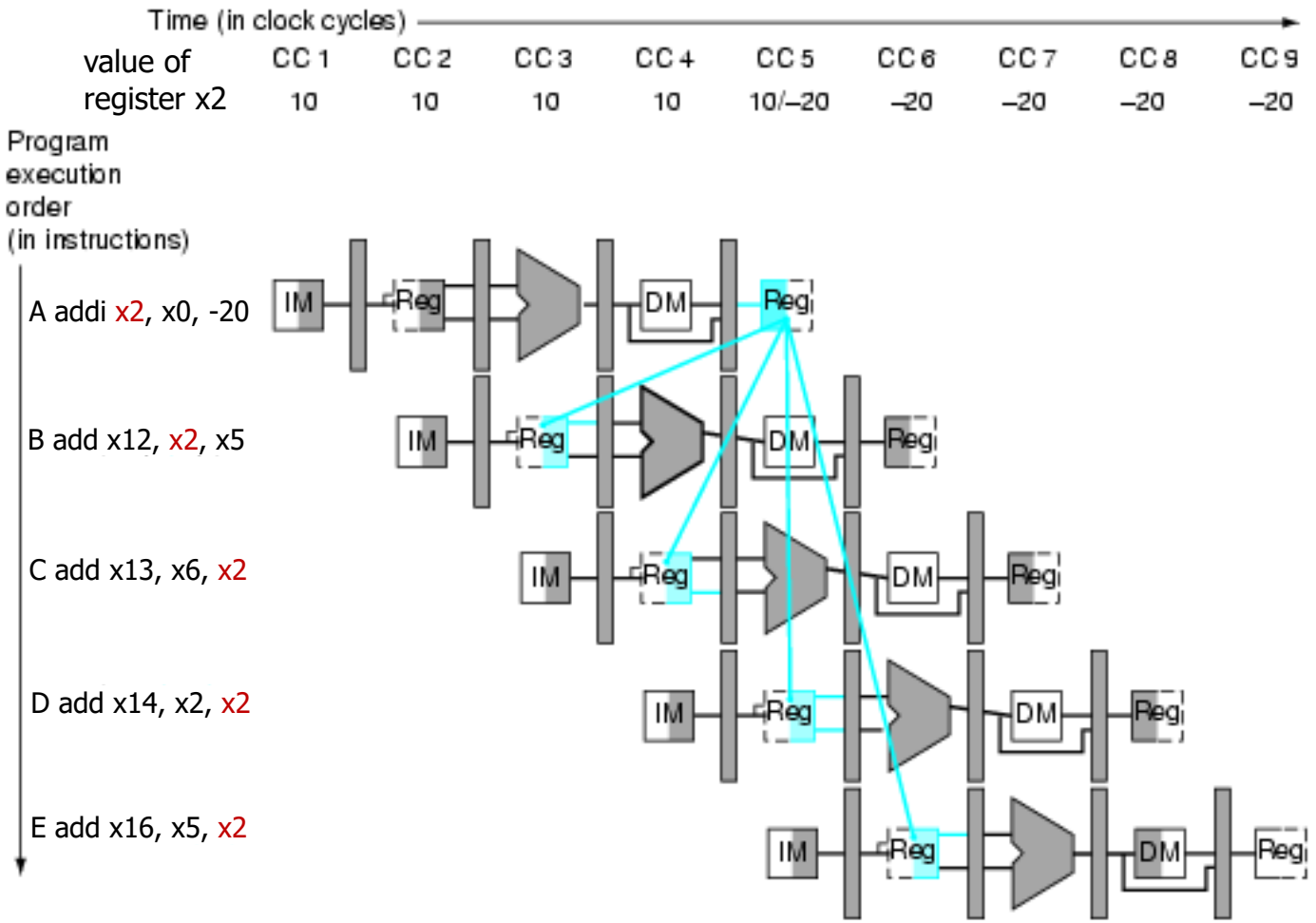
```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  00:   add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  04:   addi x5, x0, 3   // x5 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  08:   addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd5, 5'd5, 3'b000, 5'd30,7'b0110011}; //  0c:   add  x30,x5, X5  // led = x5 + x5
cm_ram[4]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  10:   add  x0, x0, x0  // NOP
```

CC6

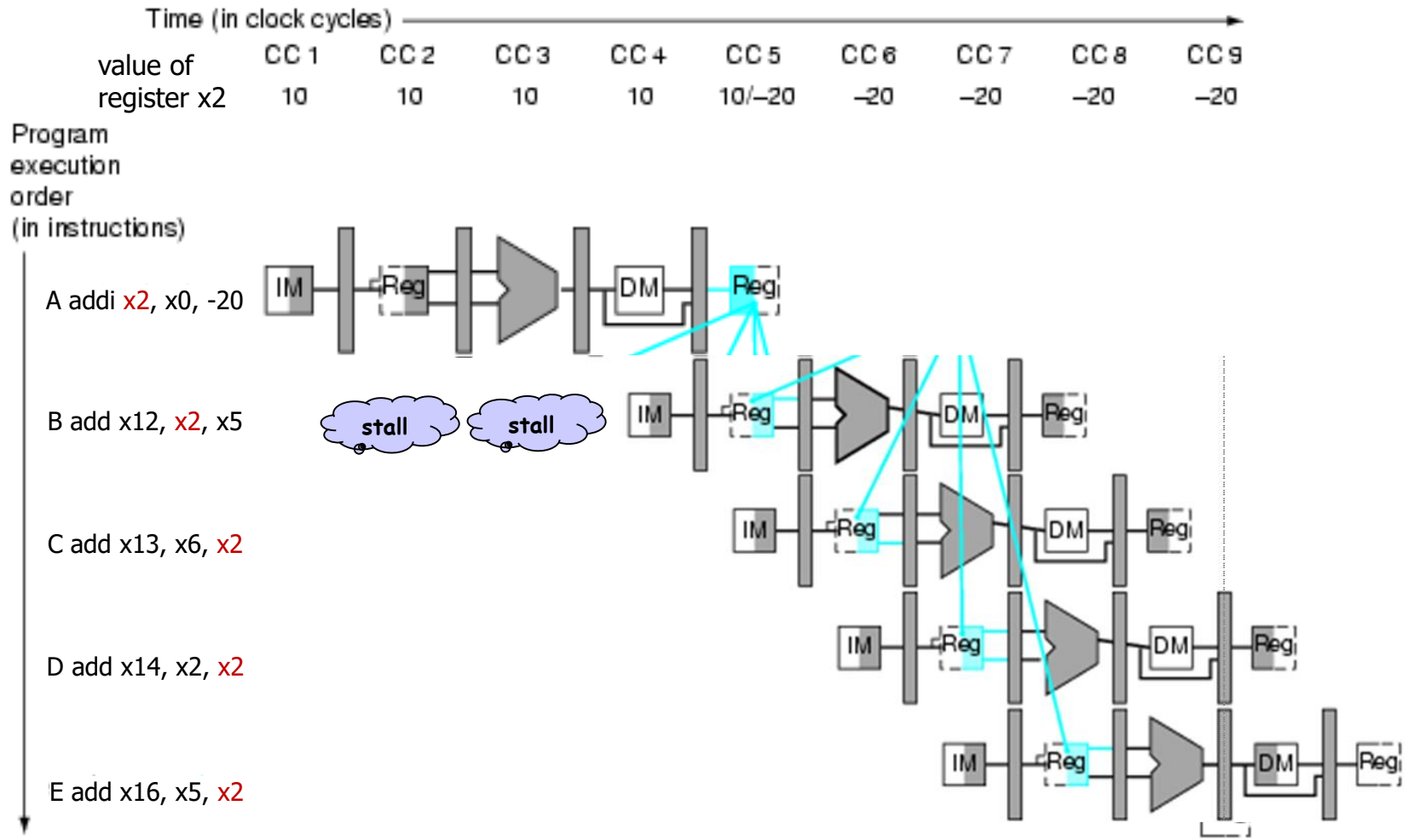


データ・ハザードとパイプラインチャート

- 命令Aが生成した x2 を後続の命令が利用する場合に，データの受け渡しの制約が生じる．



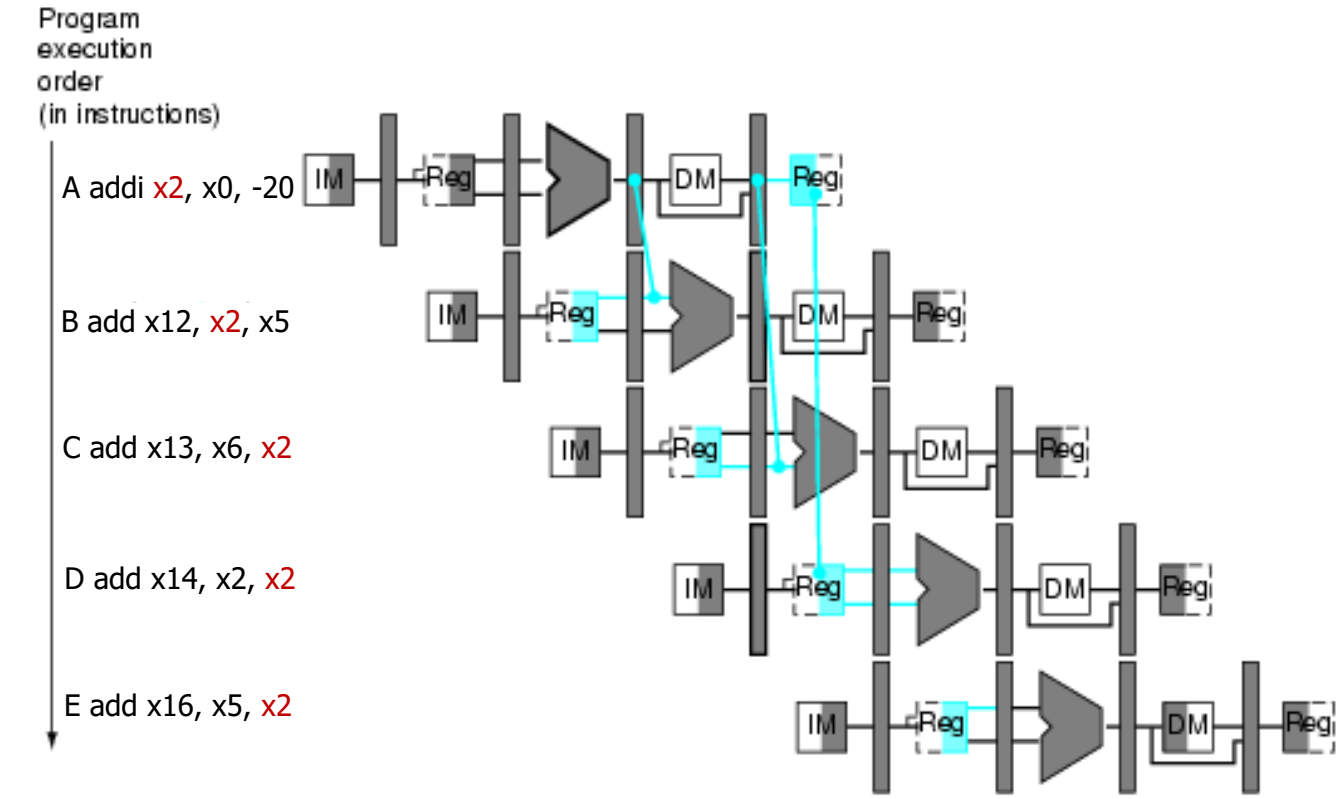
Data Hazard and Stall



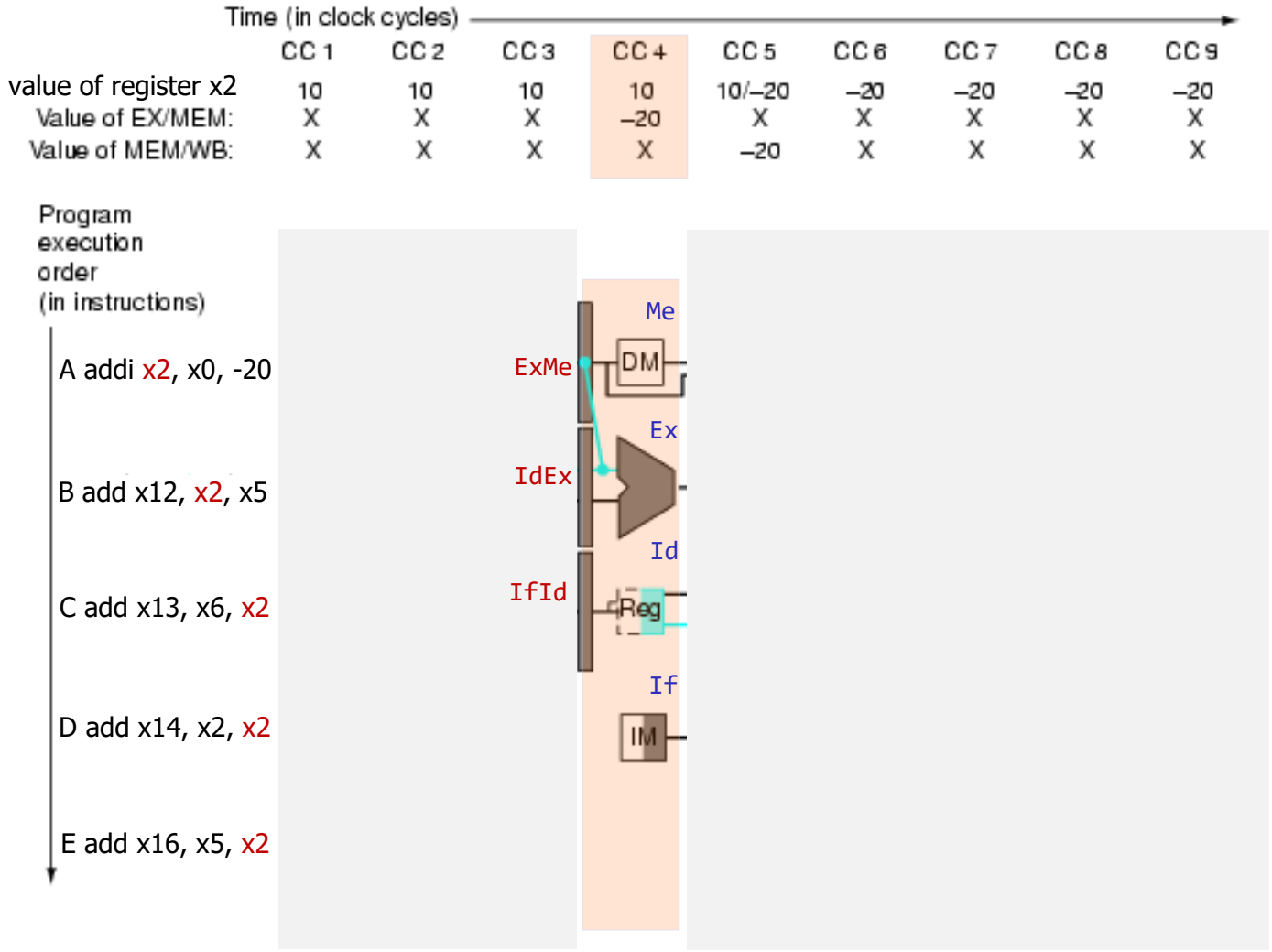
フォワーディングによるデータハザードの回避

Time (in clock cycles)

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
value of register x2	10	10	10	10	10/–20	–20	–20	–20	–20
Value of EX/MEM:	X	X	X	–20	X	X	X	X	X
Value of MEM/WB:	X	X	X	X	–20	X	X	X	X



フォワーディングによるデータハザードの回避



フォワーディングによるデータハザードの回避

Time (in clock cycles)

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
value of register x2	10	10	10	10	10/–20	–20	–20	–20	–20
Value of EX/MEM:	X	X	X	–20	X	X	X	X	X
Value of MEM/WB:	X	X	X	X	–20	X	X	X	X

Program execution order (in instructions)

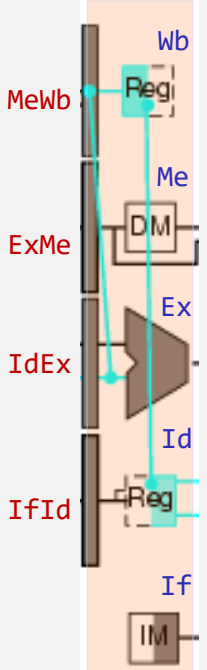
A addi x2, x0, -20

B add x12, x2, x5

C add x13, x6, x2

D add x14, x2, x2

E add x16, x5, x2



フォワーディングによるデータハザードの回避

Time (in clock cycles)

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
value of register x2	10	10	10	10	10/–20	–20	–20	–20	–20
Value of EX/MEM:	X	X	X	–20	X	X	X	X	X
Value of MEM/WB:	X	X	X	X	–20	X	X	X	X

Program execution order (in instructions)

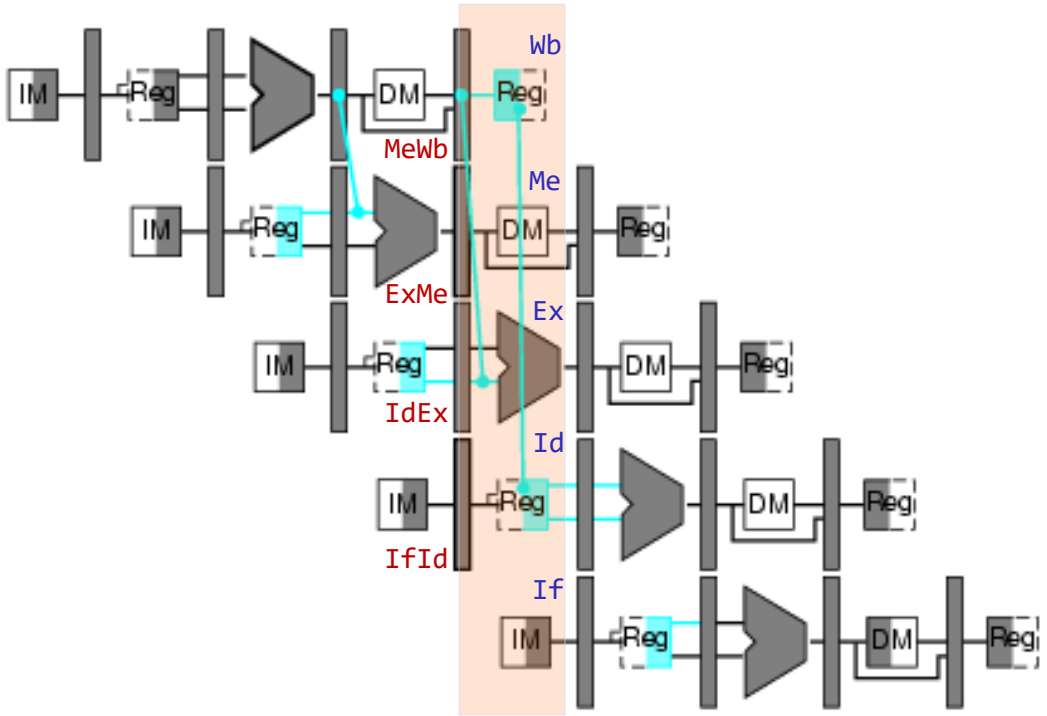
A addi x2, x0, -20

B add x12, x2, x5

C add x13, x6, x2

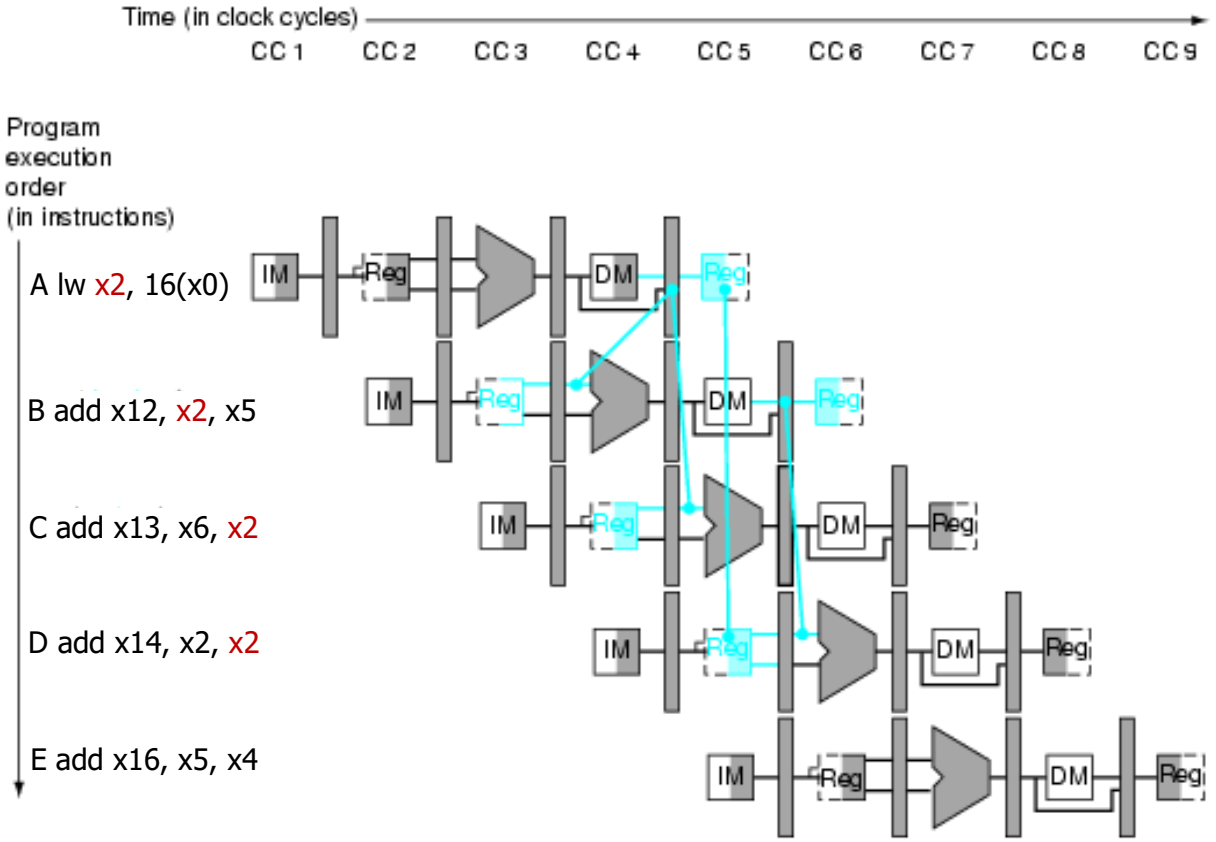
D add x14, x2, x2

E add x16, x5, x2



フォワーディングで解決できないデータハザード

- ロード命令がロードしたデータを次の命令が使う場合には，フォワーディングでもデータを供給できない．
- このような命令列が生成されないように，ソフトウェア（コンパイラ，アセンブラ）が工夫する．または，このような命令列が実行される時に，ハードウェアをストールさせる．
- 今回のコンテストでは，lw命令でロードした値を次の命令が使うことは無いので、このストールを実装する必要はない．



RISC-V Program for the contest

- Result : 0x017fd000

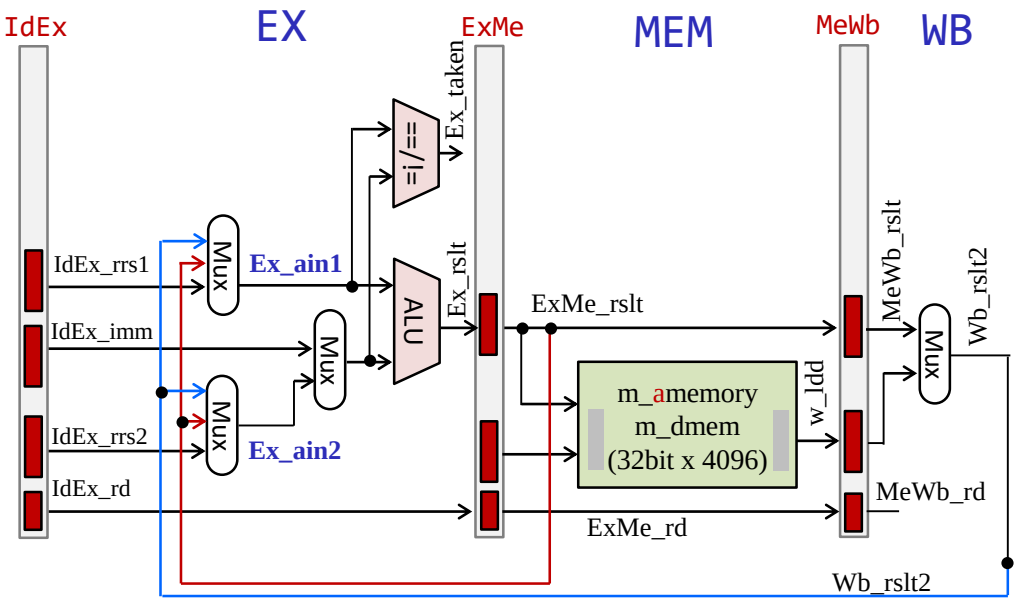
```
#include <stdio.h>

main()
{
    int mem[2048];
    int i=0, j=0, sum = 0;

    for(j=0; j<3; j++){
        for(i=0; i<2048; i++) {
            mem[i] = i*4;
        }
        for(i=0; i<2048; i++) {
            sum += mem[i];
        }
    }
    printf("%d %x¥n", sum, sum);
}
```

/home/tu_kise/cld/2023/baseline/program.txt

```
/*************************************************************/
/** program for CLD design contest 2023 (Version 2023-05-14a) **/
/** do not modify this code                                  **/
/*************************************************************/
initial begin
cm_ram[ 0] = 32'h00000033; //     add  x0,  x0,  x0
cm_ram[ 1] = 32'h00000033; //     add  x0,  x0,  x0
cm_ram[ 2] = 32'h00000a13; //     addi x20, x0,  0
cm_ram[ 3] = 32'h00300a93; //     addi x21, x0,  3
cm_ram[ 4] = 32'h00000633; //     add  x12, x0,  x0
cm_ram[ 5] = 32'h00000593; // L03:addi x11, x0,  0
cm_ram[ 6] = 32'h40000413; //     addi x8,  x0,  1024
cm_ram[ 7] = 32'h40040413; //     addi x8,  x8,  1024
cm_ram[ 8] = 32'h00000493; //     addi x9,  x0,  0
cm_ram[ 9] = 32'h00000513; //     addi x10, x0,  0
cm_ram[10] = 32'h00b52023; // L01:sw   x11, 0(x10)
cm_ram[11] = 32'h00148493; //     addi x9,  x9,  1
cm_ram[12] = 32'h00458593; //     addi x11, x11, 4
cm_ram[13] = 32'h00959cb3; //     sll  x25, x11, x9
cm_ram[14] = 32'h009cdd33; //     srl  x26, x25, x9
cm_ram[15] = 32'h00058593; //     addi x11, x11, 0
cm_ram[16] = 32'h00450513; //     addi x10, x10, 4
cm_ram[17] = 32'h00940463; //     beq  x8,  x9,  L04
cm_ram[18] = 32'hfe0410e3; //     bne  x8,  x0,  L01
cm_ram[19] = 32'h40000413; // L04:addi x8,  x0,  1024
cm_ram[20] = 32'h40040413; //     addi x8,  x8,  1024
cm_ram[21] = 32'h00000493; //     addi x9,  x0,  0
cm_ram[22] = 32'h00000513; //     addi x10, x0,  0
cm_ram[23] = 32'h00052583; // L02:lw   x11, 0(x10)
cm_ram[24] = 32'h00148493; //     addi x9,  x9,  1
cm_ram[25] = 32'h00450513; //     addi x10, x10, 4
cm_ram[26] = 32'h00b60633; //     add  x12, x12, x11
cm_ram[27] = 32'h00160613; //     addi x12, x12, 1
cm_ram[28] = 32'hfff60613; //     addi x12, x12, -1
cm_ram[29] = 32'h00160613; //     addi x12, x12, 1
cm_ram[30] = 32'h00160613; //     addi x12, x12, 1
cm_ram[31] = 32'hfff60613; //     addi x12, x12, -1
cm_ram[32] = 32'h00160613; //     addi x12, x12, 1
cm_ram[33] = 32'hffe60613; //     addi x12, x12, -2
cm_ram[34] = 32'hfc941ae3; //     bne  x8,  x9,  L02
cm_ram[35] = 32'h015d5d33; //     srl  x26, x26, x21
cm_ram[36] = 32'h001a0a13; //     addi x20, x20, 1
cm_ram[37] = 32'h01140413; //     addi x8,  x8,  0x11
cm_ram[38] = 32'h01240413; //     addi x8,  x8,  0x12
cm_ram[39] = 32'h01340413; //     addi x8,  x8,  0x13
cm_ram[40] = 32'h01440413; //     addi x8,  x8,  0x14
cm_ram[41] = 32'hf75a18e3; //     bne  x20, x21, L03
cm_ram[42] = 32'h00000033; //     add  x0,  x0,  x0
cm_ram[43] = 32'h00060f33; //     add  x30, x12, x0
cm_ram[44] = 32'h000f0033; //     add  x0,  x30, x0
cm_ram[45] = 32'h00000033; //     add  x0,  x0,  x0
cm_ram[46] = 32'h00000033; //     add  x0,  x0,  x0
cm_ram[47] = 32'h00000033; //     add  x0,  x0,  x0
cm_ram[48] = 32'h00000033; //     add  x0,  x0,  x0
cm_ram[49] = 32'h00000033; //     add  x0,  x0,  x0
end
```

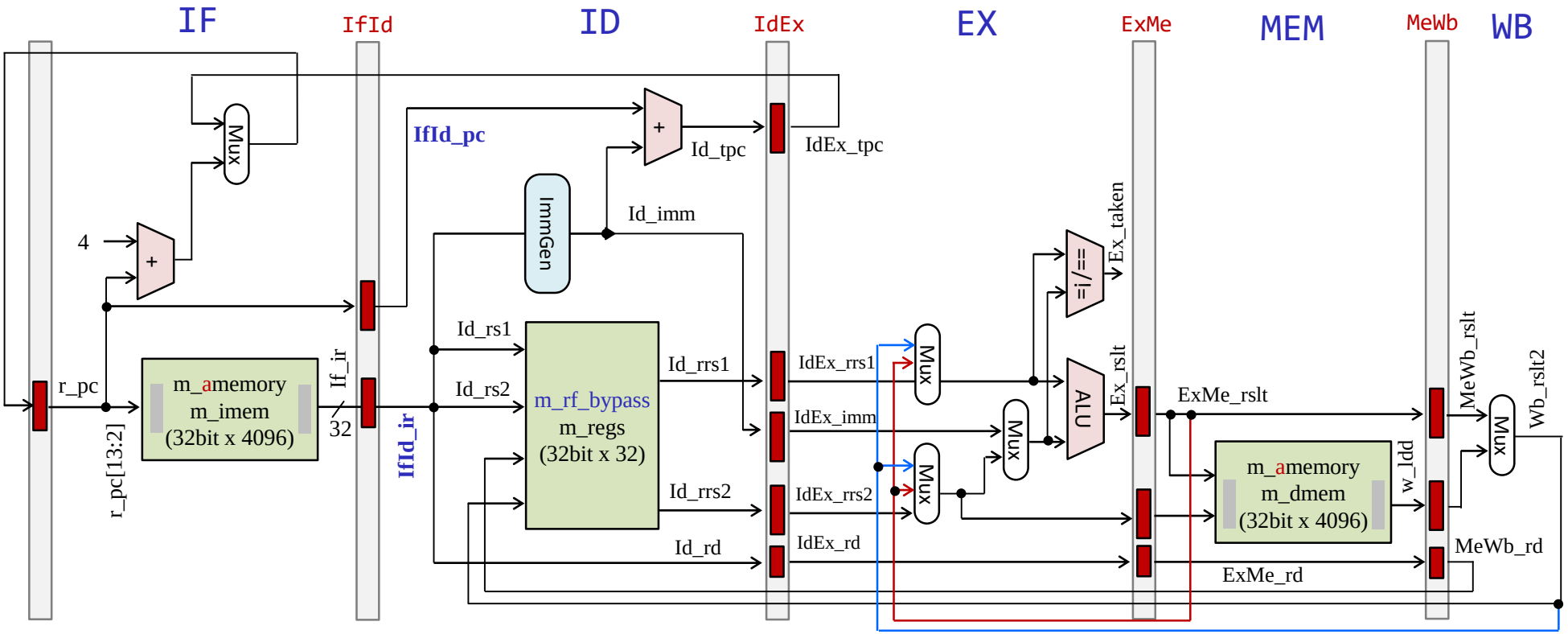
フォワーディングのための変更点

```
wire [31:0] Ex_ain1 = (Ex_rs1==0) ? 0 : (Ex_rs1==Me_rd & Me_we) ? ExMe_rslt : (Ex_rs1==Wb_rd & Wb_we) ? Wb_rslt2 : IdEx_rrs1;

wire [31:0] Ex_ain2 = (Ex_rs2==0) ? 0 : (Ex_rs2==Me_rd & Me_we) ? ExMe_rslt : (Ex_rs2==Wb_rd & Wb_we) ? Wb_rslt2 : IdEx_rrs2;
```



m_proc20 5段のパイプライン版

- add, addi, sll, srl, lw, sw, beq, bne命令に対応したプロセッサ（データフォワーディング有り）
- IF, ID, EX, MEM, WB のそれぞれをステージとする5段のパイプラインプロセッサ
- ステージ IF と ID の間のパイプラインレジスタには IfId_ から始まる名前を利用する.
- ステージ ID で生成される配線には Id_ から始まる名前を利用する.



code190.v（code185.v を参考に自分で実装する）

Recommended Reading

PAPER Special Section on Parallel, Distributed, and Reconfigurable Computing, and Networking

RVCoreP: An Optimized RISC-V Soft Processor of Five-Stage Pipelining

Hiromu MIYAZAKI[†a)], *Student Member*, Takuto KANAMORI[†b)], Md Ashraful ISLAM[†c)], *Nonmembers*, *and* Kenji KISE[†d)], *Member*

SUMMARY RISC-V is a RISC based open and loyalty free instruction set architecture which has been developed since 2010, and can be used for cost-effective soft processors on FPGAs. The basic 32-bit integer instruction set in RISC-V is defined as RV32I, which is sufficient to support the operating system environment and suits for embedded systems. In this paper, we propose an optimized RV32I soft processor named RVCoreP adopting five-stage pipelining. Three effective methods are applied to the processor to improve the operating frequency. These methods are instruction fetch unit optimization, ALU optimization, and data memory optimization. We implement RVCoreP in Verilog HDL and verify the behavior using Verilog simulation and an actual Xilinx Atrix-7 FPGA board. We evaluate IPC (instructions per cycle), operating frequency, hardware resource utilization, and processor performance. From the evaluation results, we show that RVCoreP achieves 30.0% performance improvement compared with VexRiscv, which is a high-performance and open source RV32I processor selected from some related works.

key words: *soft processor, FPGA, RISC-V, RV32I, Verilog HDL, five-stage pipelining*

1. Introduction

version of a general-purpose instruction set.

Among these instruction sets, we focus on RV32I in this paper because it is sufficient to support the operating system environment and suits for embedded systems. RV32I can emulate other extensions of M, F, and D, and can be configured with fewer hardware resources than processors supporting RV32G. Although several soft processors that support RV32I have been released [5], they are not highly optimized for FPGAs.

In this paper, we propose an optimized RV32I soft processor named **RVCoreP** of five-stage pipelining which is highly optimized for FPGAs. The main contributions of this paper are as follows.

- We propose an optimized RV32I soft processor of five-stage pipelining highly optimized for FPGAs. To improve the operating frequency, three optimization methods are applied to the processor. They are instruction fetch unit optimization, ALU optimization, and

RV-PC: RISC-Vパーソナルコンピュータ $500

- Stand-alone RISC-V computer
  - Digilent Nexys A7 FPGA Board
  - Pmod ps2 keyboard / PS2 keyboard
  - USB wireless mouse
  - Mobile battery
  - VGA display
  - Some cables (microUSB, VGA)
  - Two acrylic plates, spacers
  - Option, ethernet connection

https://youtu.be/Kt_iXVAjXcQ

Syllabus (1/3)

講義の概要とねらい

本講義では、「論理回路理論」の講義で習得した知識をベースに、より実用的なディジタル回路について学ぶ。また、簡単なコンピュータを例題として、コンピュータの基本原理とその論理設計の方法を学習する。
演習では、学んだ組合せ回路と順序回路をVerilog HDL等のハードウェア記述言語で記述し、シミュレーションによる回路の動作検証、FPGAが搭載されたハードウェアボード等に実装して動作確認をおこなう。

到達目標

本講義を履修することによって以下を習得する。
・コンピュータシステムの基本構成
・シングルサイクルプロセッサの論理設計に関する知識
・パイプライン処理をおこなうプロセッサの論理設計に関する知識
・ハードウェア記述言語を用いたシンプルなコンピュータシステムの設計能力

キーワード

コンピュータ，命令セットアーキテクチャ，プロセッサ、パイプライン処理，ハードウェア記述言語，Verilog HDL、FPGA

学生が身につける力

国際的教養力	コミュニケーション力	専門力	課題設定力	実践力または解決力
-	-	✓	-	✓

授業の進め方

原則として、90分×2コマの講義の後、90分×1コマのFPGAボードを用いた演習をおこないます。

Syllabus (2/3)

授業計画・課題		
	授業計画	課題
第1回	コンピュータシステムの基本構成	コンピュータシステムの基本構成について理解する。
第2回	論理設計演習(1)	論理設計演習(1)
第3回	ハードウェア記述言語：組合せ回路	組合せ回路の記述を理解する。
第4回	ハードウェア記述言語：順序回路	順序回路の記述を理解する。
第5回	論理設計演習(2)	論理設計演習(2)
第6回	ハードウェア記述言語：よく使われる回路	よく使われる回路の記述を理解する。
第7回	リコンフィギャラブルシステム	リコンフィギャラブルシステムとFPGAボードについて理解する。
第8回	論理設計演習(3)	論理設計演習(3)
第9回	命令セットアーキテクチャ：データ表現とアドレス指定形式	ISAにおけるデータ表現とアドレス指定形式について理解する。
第10回	命令セットアーキテクチャ：算術論理演算命令	ISAにおける算術論理演算命令について理解する。
第11回	論理設計演習(4)	論理設計演習(4)
第12回	命令セットアーキテクチャ：ロードストア命令と分岐命令	ISAにおけるロードストア命令と分岐命令について理解する。
第13回	プロセッサの基本構成要素：算術論理演算ユニット	算術論理演算ユニットについて理解する。
第14回	論理設計演習(5)	論理設計演習(5)
第15回	プロセッサの基本構成要素：レジスタファイルとメモリ	レジスタファイルとメモリについて理解する。
第16回	シングルサイクルプロセッサのデータパス	シングルサイクルプロセッサのデータパスについて理解する。
第17回	論理設計演習(6)	論理設計演習(6)
第18回	シングルサイクルプロセッサの制御	シングルサイクルプロセッサの制御について理解する。
第19回	パイプライン処理	パイプライン処理について理解する。
第20回	論理設計演習(7)	論理設計演習(7)
第21回	パイプラインハザードとデータフォワーディング	パイプラインハザードとデータフォワーディングについて理解する。
第22回	論理設計演習(8)	論理設計演習(8)

Syllabus (3/3)

教科書

デイビッド・A. パターソン、ジョン・L. ヘネシー (著)、成田光彰 (翻訳)『コンピュータの構成と設計 第5版 上/下』日経BP社

参考書、講義資料等

無し。

成績評価の基準及び方法

講義で扱うコンピュータ論理設計に関する理解、ハードウェア記述言語を用いたコンピュータシステム実装への応用力を評価する。演習（30%）と期末

演習 30%, 設計コンテスト 20%, 期末試験 50%

関連する科目

CSC.T252 ： 論理回路理論
CSC.T262 ： アセンブリ言語
CSC.T372 ： コンパイラ構成
CSC.T363 ： コンピュータアーキテクチャ
CSC.T433 ： 先端コンピュータアーキテクチャ

履修の条件(知識・技能・履修済科目等)

履修条件は特に設けないが、関連する科目の論理回路理論を履修していることが望ましい。

連絡先（メール、電話番号）　※"[at]"を"@"(半角)に変換してください。

吉瀬謙二: kise[at]c.titech.ac.jp

オフィスアワー

メールで事前予約すること。

References

- Computer Logic Design support page
  - https://www.arch.cs.titech.ac.jp/lecture/CLD/
- ACRi Room
  - https://gw.acri.c.titech.ac.jp
- ACRi Blog
  - https://www.acri.c.titech.ac.jp/wordpress/
- 情報工学系計算機室
  - http://www.csc.titech.ac.jp/
- Xilinx Vivado Design Suite
  - https://japan.xilinx.com/products/design-tools/vivado.html
- Digilent Arty A7-35T
  - https://reference.digilentinc.com/reference/programmable-logic/arty-a7/start
- Verilog HDL
  - https://ja.wikipedia.org/wiki/Verilog