Course number: CSC.T341

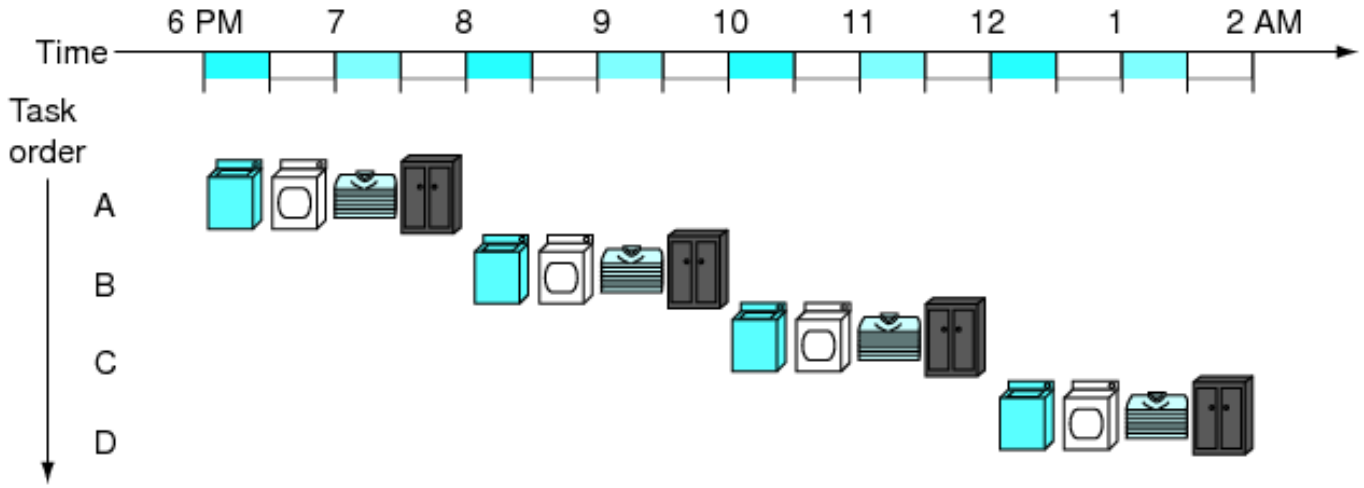
コンピュータ論理設計
Computer Logic Design

13. パイプライン処理と制御ハザード
Pipelining and Control Hazard

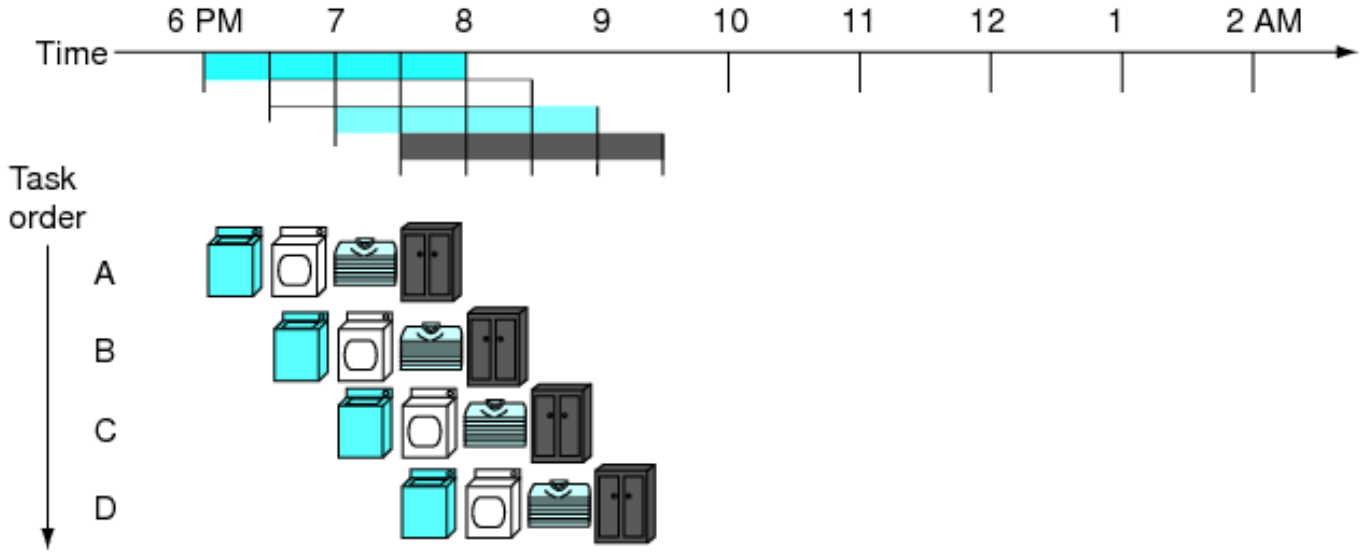
吉瀬 謙二　情報工学系
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp　www.arch.cs.titech.ac.jp/lecture/CLD/
W621 講義室　月 10:45-12:15, 木 9:00-12:15

Pipelined Processor

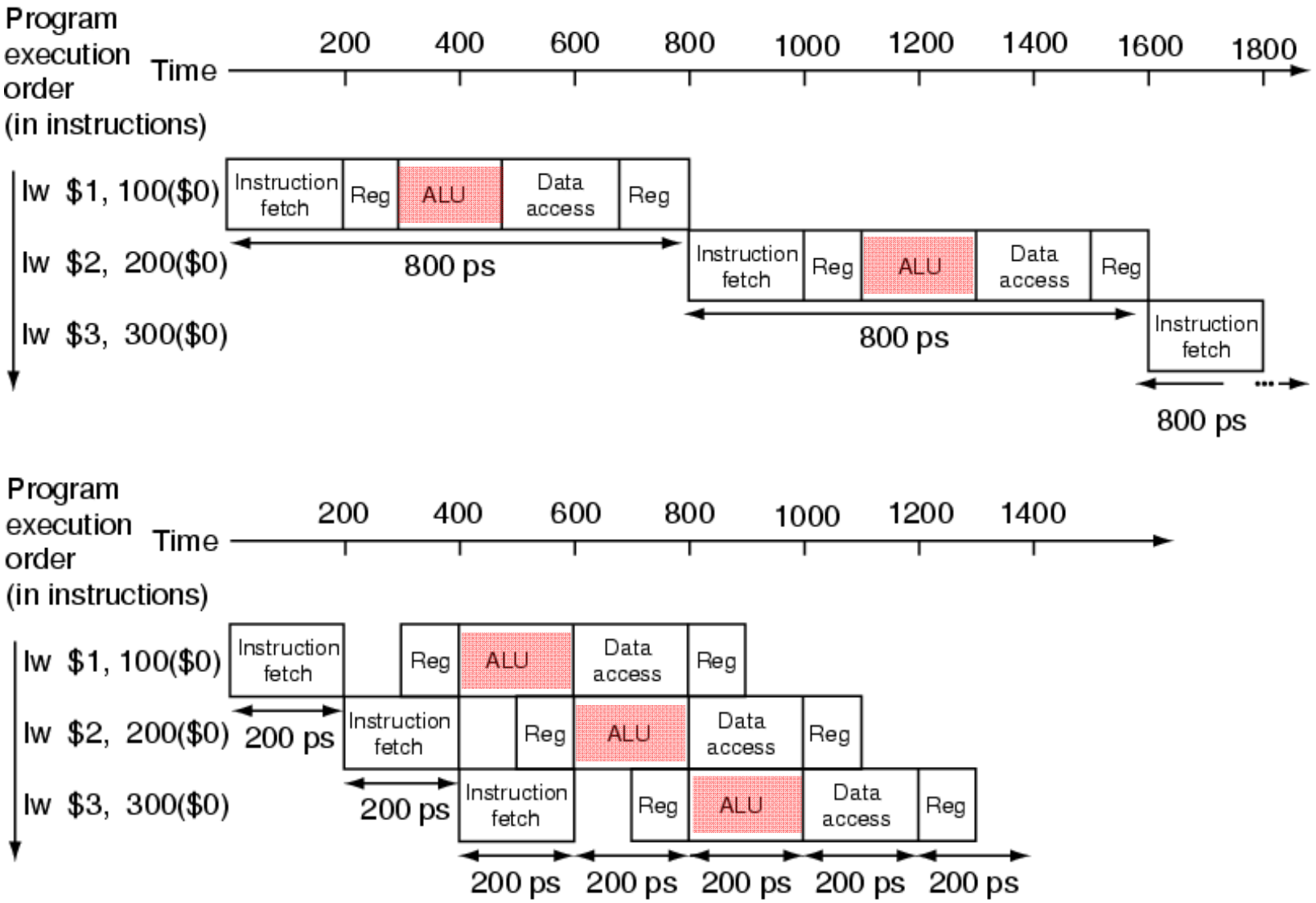
- Non pipelining (Multi-cycle)



- Pipelining



Pipelined Processor



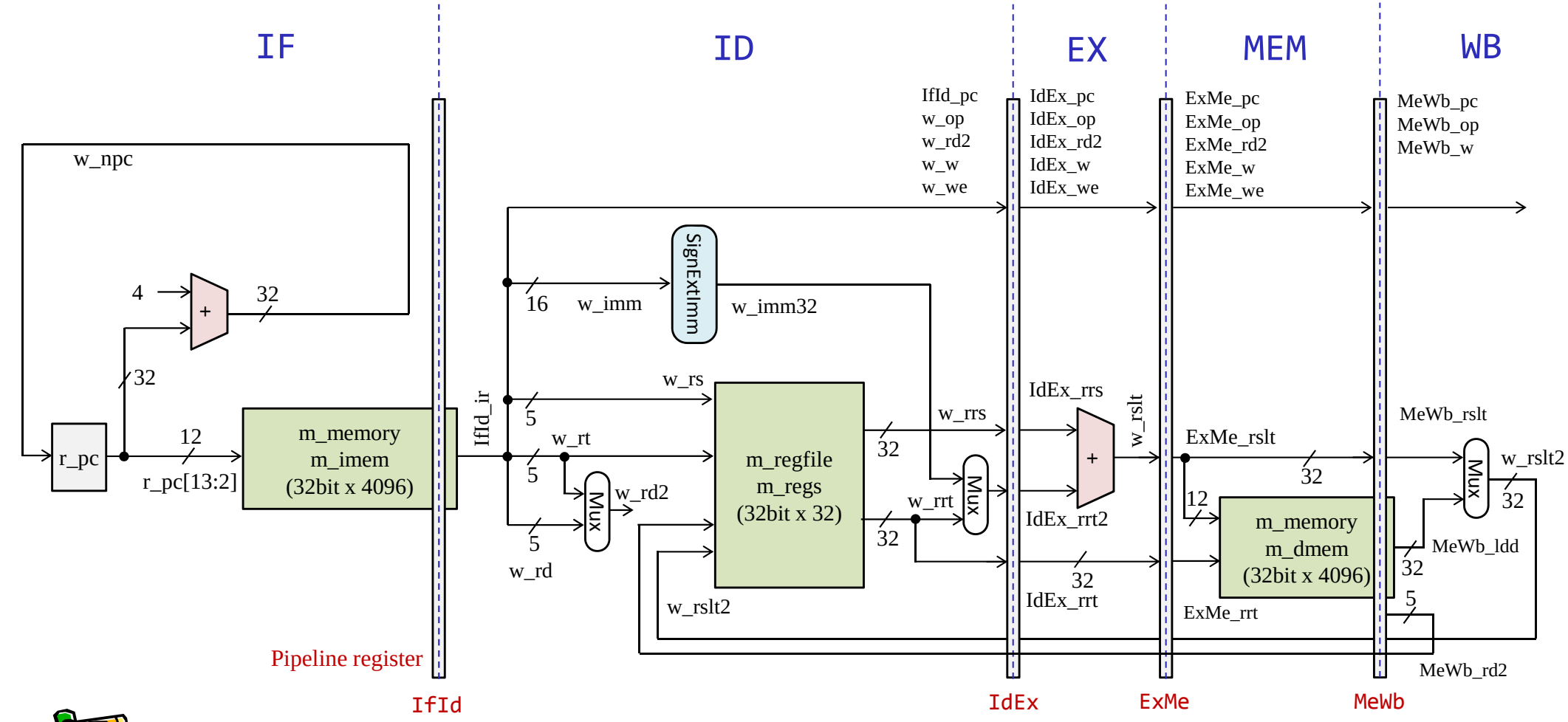
Hazards make pipelining hard

- 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
  - 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

- まずは，データ・ハザードと制御ハザードが無いものとしてパイプラインプロセッサを構築する．その後，これらに対応するための修正を加える．

Inside module m_proc10 (pipelined processor)

- add, addi, lw, sw, halt命令に対応したデータハザードを考慮しないパイプライン処理のプロセッサ
- 最も遅延の長い（長い時間を要する）ステージがプロセッサの動作周波数を決める.



Inside module m_proc10 (pipelined about 150MHz)

code130.v

```
module m_proc10 (w_clk, w_rst, r_rout, r_halt);
  input  wire w_clk, w_rst;
  output reg [31:0] r_rout;
  output reg        r_halt;

  reg  [31:0] IdEx_rrs=0, IdEx_rrt=0, IdEx_rrt2=0;           // pipe regs
  reg  [31:0] ExMe_rslt=0, ExMe_rrt=0;                       //
  reg  [31:0] MeWb_rslt=0;                                   //
  reg   [5:0]             IdEx_op=0,  ExMe_op=0,  MeWb_op=0; //
  reg  [31:0] IfId_pc=0,  IdEx_pc=0,  ExMe_pc=0,  MeWb_pc=0; //
  reg   [4:0] IfId_rd2=0, IdEx_rd2=0, ExMe_rd2=0, MeWb_rd2=0;//
  reg         IfId_w=0,   IdEx_w=0,   ExMe_w=0,   MeWb_w=0;  //
  reg         IfId_we=0,  IdEx_we=0,  ExMe_we=0;             //
  wire [31:0] IfId_ir, MeWb_ldd;                             // note

  /************************** IF stage ***************************/
  reg  [31:0] r_pc   = 0;
  wire [31:0] w_npc = r_pc + 4;
  always @(posedge w_clk) begin
    r_pc <= #3 (w_rst | r_halt) ? 0 : w_npc;
  end
  m_memory m_imem (w_clk, r_pc[13:2], 0, 0, IfId_ir);
  always @(posedge w_clk) begin
    IfId_pc <= #3 r_pc;
  end
  /************************** ID stage ***************************/
  wire  [5:0] w_op  = IfId_ir[31:26];
  wire  [4:0] w_rs  = IfId_ir[25:21];
  wire  [4:0] w_rt  = IfId_ir[20:16];
  wire  [4:0] w_rd  = IfId_ir[15:11];
  wire  [4:0] w_rd2 = (w_op!=0) ? w_rt : w_rd;
  wire [15:0] w_imm = IfId_ir[15:0];
  wire [31:0] w_rrs, w_rrt, w_rslt2;

  m_regfile m_regs (w_clk, w_rs, w_rt, MeWb_rd2, MeWb_w, w_rslt2, w_rrs,
      w_rrt);
  wire [31:0] w_imm32 = {{16{w_imm[15]}}, w_imm};
  wire [31:0] w_rrt2  = (w_op>6'h5) ? w_imm32 : w_rrt;
```

```
  always @(posedge w_clk) begin
    IdEx_pc   <= #3 IfId_pc;
    IdEx_op   <= #3 w_op;
    IdEx_rd2  <= #3 w_rd2;
    IdEx_w    <= #3 (w_op==0 || (w_op>6'h5 && w_op<6'h28));
    IdEx_we   <= #3 (w_op>6'h27);
    IdEx_rrs  <= #3 w_rrs;
    IdEx_rrt  <= #3 w_rrt;
    IdEx_rrt2 <= #3 w_rrt2;
  end
  /************************** EX stage ***************************/
  wire [31:0] #10 w_rslt = IdEx_rrs + IdEx_rrt2; // ALU
  always @(posedge w_clk) begin
    ExMe_pc   <= #3 IdEx_pc;
    ExMe_op   <= #3 IdEx_op;
    ExMe_rd2  <= #3 IdEx_rd2;
    ExMe_w    <= #3 IdEx_w;
    ExMe_we   <= #3 IdEx_we;
    ExMe_rslt <= #3 w_rslt;
    ExMe_rrt  <= #3 IdEx_rrt;
  end
  /************************** MEM stage ***************************/
  m_memory m_dmem (w_clk, ExMe_rslt[13:2], ExMe_we, ExMe_rrt, MeWb_ldd);
  always @(posedge w_clk) begin
    MeWb_pc   <= #3 ExMe_pc;
    MeWb_rslt <= #3 ExMe_rslt;
    MeWb_op   <= #3 ExMe_op;
    MeWb_rd2  <= #3 ExMe_rd2;
    MeWb_w    <= #3 ExMe_w;
  end
  /************************** WB stage ***************************/
  assign w_rslt2 = (MeWb_op>6'h19 && MeWb_op<6'h28) ? MeWb_ldd : MeWb_rslt;

  /***************************************************************/
  initial r_halt = 0;
  always @(posedge w_clk) if (MeWb_op==`HALT) r_halt <= 1;
  initial r_rout = 0;
  reg [31:0] r_tmp=0;
  always @(posedge w_clk) r_tmp <= (w_rst) ? 0 : (w_rs==30) ? w_rrs : r_tmp;
  always @(posedge w_clk) r_rout <= r_tmp;
endmodule
```

Inside module m_proc10 (pipelined about 150MHz)

code130.v

```
module m_top ();
  reg r_clk=0; initial forever #50 r_clk = ~r_clk;
  reg r_rst=0;

  wire w_halt;
  wire [31:0] w_rout;
  m_proc10 p (r_clk, r_rst, w_rout, w_halt);
  always@(posedge r_clk) if (w_halt) $finish;

  reg [31:0] r_cnt = 0;
  always@(posedge r_clk) r_cnt <= r_cnt + 1;
  always@(posedge r_clk) begin #90
    $write("%8d : %x %x[%x] %x %x %x | %d %x ¥n",
           r_cnt, p.r_pc, p.IfId_pc, p.w_op, p.IdEx_pc, p.ExMe_pc, p.MeWb_pc,
           p.MeWb_rd2, p.w_rslt2);
  end
endmodule
```

```
       1 : 00000004 00000000[00] 00000000 00000000 00000000 |  0 00000000
       2 : 00000008 00000004[08] 00000000 00000000 00000000 |  0 00000000
       3 : 0000000c 00000008[08] 00000004 00000000 00000000 |  0 00000000
       4 : 00000010 0000000c[08] 00000008 00000004 00000000 |  0 00000000
       5 : 00000014 00000010[08] 0000000c 00000008 00000004 |  1 00000020
       6 : 00000018 00000014[08] 00000010 0000000c 00000008 | 10 00000001
       7 : 0000001c 00000018[08] 00000014 00000010 0000000c | 11 00000002
       8 : 00000020 0000001c[08] 00000018 00000014 00000010 | 12 00000003
       9 : 00000024 00000020[08] 0000001c 00000018 00000014 | 13 00000004
      10 : 00000028 00000024[08] 00000020 0000001c 00000018 | 10 00000011
      11 : 0000002c 00000028[00] 00000024 00000020 0000001c | 11 00000012
      12 : 00000030 0000002c[00] 00000028 00000024 00000020 | 12 00000013
      13 : 00000034 00000030[00] 0000002c 00000028 00000024 | 13 00000014
      14 : 00000038 00000034[00] 00000030 0000002c 00000028 | 10 00000031
      15 : 0000003c 00000038[11] 00000034 00000030 0000002c | 11 00000032
      16 : 00000040 0000003c[00] 00000038 00000034 00000030 | 12 00000033
      17 : 00000044 00000040[00] 0000003c 00000038 00000034 | 13 00000034
      18 : 00000048 00000044[00] 00000040 0000003c 00000038 |  0 00000000
      19 : 0000004c 00000048[00] 00000044 00000040 0000003c |  0 00000000
```

```
module m_memory (w_clk, w_addr, w_we, w_din, r_dout);
  input  wire w_clk, w_we;
  input  wire [11:0] w_addr;
  input  wire [31:0] w_din;
  output reg  [31:0] r_dout;
  reg [31:0] cm_ram [0:4095]; // 4K word (4096 x 32bit) memory
  always @(posedge w_clk) begin
    r_dout <= cm_ram[w_addr];
    if (w_we) cm_ram[w_addr] <= w_din;
  end
  initial r_dout = 0;

  initial begin
    cm_ram[0] ={`NOP};                              //     nop
    cm_ram[1] ={`ADDI, 5'd0, 5'd1, 16'h20};         //     addi $1,  $0, 0x20
    cm_ram[2] ={`ADDI, 5'd0, 5'd10,16'd1};          //     addi $10, $0, 1
    cm_ram[3] ={`ADDI, 5'd0, 5'd11,16'd2};          //     addi $11, $0, 2
    cm_ram[4] ={`ADDI, 5'd0, 5'd12,16'd3};          //     addi $12, $0, 3
    cm_ram[5] ={`ADDI, 5'd0, 5'd13,16'd4};          //     addi $13, $0, 4
    cm_ram[6] ={`ADDI, 5'd10,5'd10,16'h10};         //     addi $10, $10,0x10
    cm_ram[7] ={`ADDI, 5'd11,5'd11,16'h10};         //     addi $11, $11,0x10
    cm_ram[8] ={`ADDI, 5'd12,5'd12,16'h10};         //     addi $12, $12,0x10
    cm_ram[9] ={`ADDI, 5'd13,5'd13,16'h10};         //     addi $13, $13,0x10
    cm_ram[10]={`ADD,  5'd10,5'd1, 5'd10,11'h20};   //     add  $10,$10,$1
    cm_ram[11]={`ADD,  5'd11,5'd1, 5'd11,11'h20};   //     add  $11,$11,$1
    cm_ram[12]={`ADD,  5'd12,5'd1, 5'd12,11'h20};   //     add  $12,$12,$1
    cm_ram[13]={`ADD,  5'd13,5'd1, 5'd13,11'h20};   //     add  $13,$13,$1
    cm_ram[14]={`HALT, 26'h0};                      //     halt
    cm_ram[15]={`NOP};                              //     nop
    cm_ram[16]={`NOP};                              //     nop
    cm_ram[17]={`NOP};                              //     nop
    cm_ram[18]={`NOP};                              //     nop
    cm_ram[19]={`NOP};                              //     nop
  end
endmodule
```

Hazards make pipelining hard

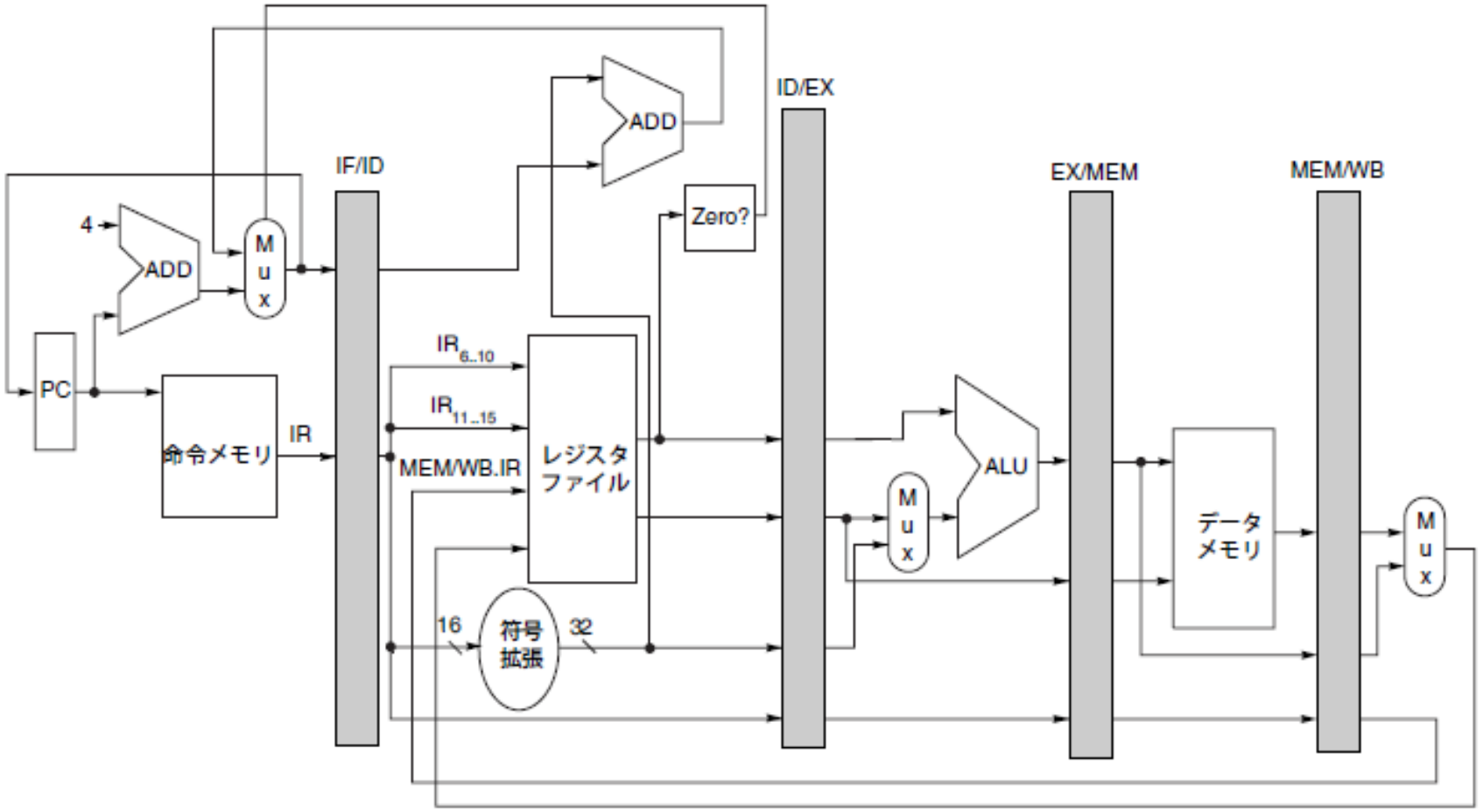
- 命令を適切なサイクルで実行できないような状況が存在する．これをハザード(hazard)と呼ぶ．
  - 構造ハザード (structural hazard)
    - オーバラップ実行する命令の組み合わせをハードウェアがサポートしていない場合．資源不足により生じる．
  - データ・ハザード(data hazard)
    - データの受け渡しの制約によって生じるハザード
  - 制御ハザード(control hazard)
    - 分岐命令，ジャンプ命令によって生じるハザード

- まずは，データ・ハザードと制御ハザードが無いものとしてパイプラインプロセッサを構築する．その後，これらに対応するための修正を加える．

プロセッサが命令を処理するための5つのステップの修正

- 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, レジスタの値を読み出す.
  分岐命令である可能性を考慮し, 読み出されたレジスタの間で一致比較を行う. 命令のオフセットフィールドを符号拡張し, インクリメントされたPCに符号拡張されたオフセットを足し合わせて分岐先のアドレスを計算する. 条件が成立した場合には分岐先アドレスをPCにセットして, このステージで分岐命令を完了させる.
- 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, データ・メモリ中のオペランドにアクセスする.
- WB (Write Back)
  必要であれば, 結果をレジスタに書き込む.

プロセッサのデータパス（パイプライン処理）



静的に採用できる制御ハザードの対処

- 戦略1
  - 分岐方向が判明するまで分岐命令の後続命令を止める.
  - IDステージで分岐命令が完了することに注意.
  - 分岐命令の出現毎に1サイクルのストールが発生する.

分岐命令	IF	ID	EX	MEM	WB		
分岐先命令			IF	ID	EX	MEM	WB
分岐先命令＋1				IF	ID	EX	MEM
分岐先命令＋2					IF	ID	EX

戦略2：predicted-not-taken方式 (Exercise)

- すべての分岐命令を not taken（不成立）として処理を進める.
  - 分岐結果が不成立であれば，ペナルティは生じない.
  - 分岐結果が成立であれば，1サイクルのペナルティ

Untaken分岐命令	IF	ID	EX	MEM	WB				
命令 $i+1$		IF	ID	EX	MEM	WB			
命令 $i+2$			IF	ID	EX	MEM	WB		
命令 $i+3$				IF	ID	EX	MEM	WB	
命令 $i+4$					IF	ID	EX	MEM	WB

Taken分岐命令	IF	ID	EX	MEM	WB				
命令 $i+1$		IF	idle	idle	idle	idle			
分岐先			IF	ID	EX	MEM	WB		
分岐先 + 1				IF	ID	EX	MEM	WB	
分岐先 + 2					IF	ID	EX	MEM	WB

戦略3：predicted-taken方式

- すべての分岐命令を taken（成立）として処理を進める.
- IDステージが終了して，分岐と判定するとすぐに分岐成立として処理を継続.
- 今考えている構成では，この方式の利点はない.

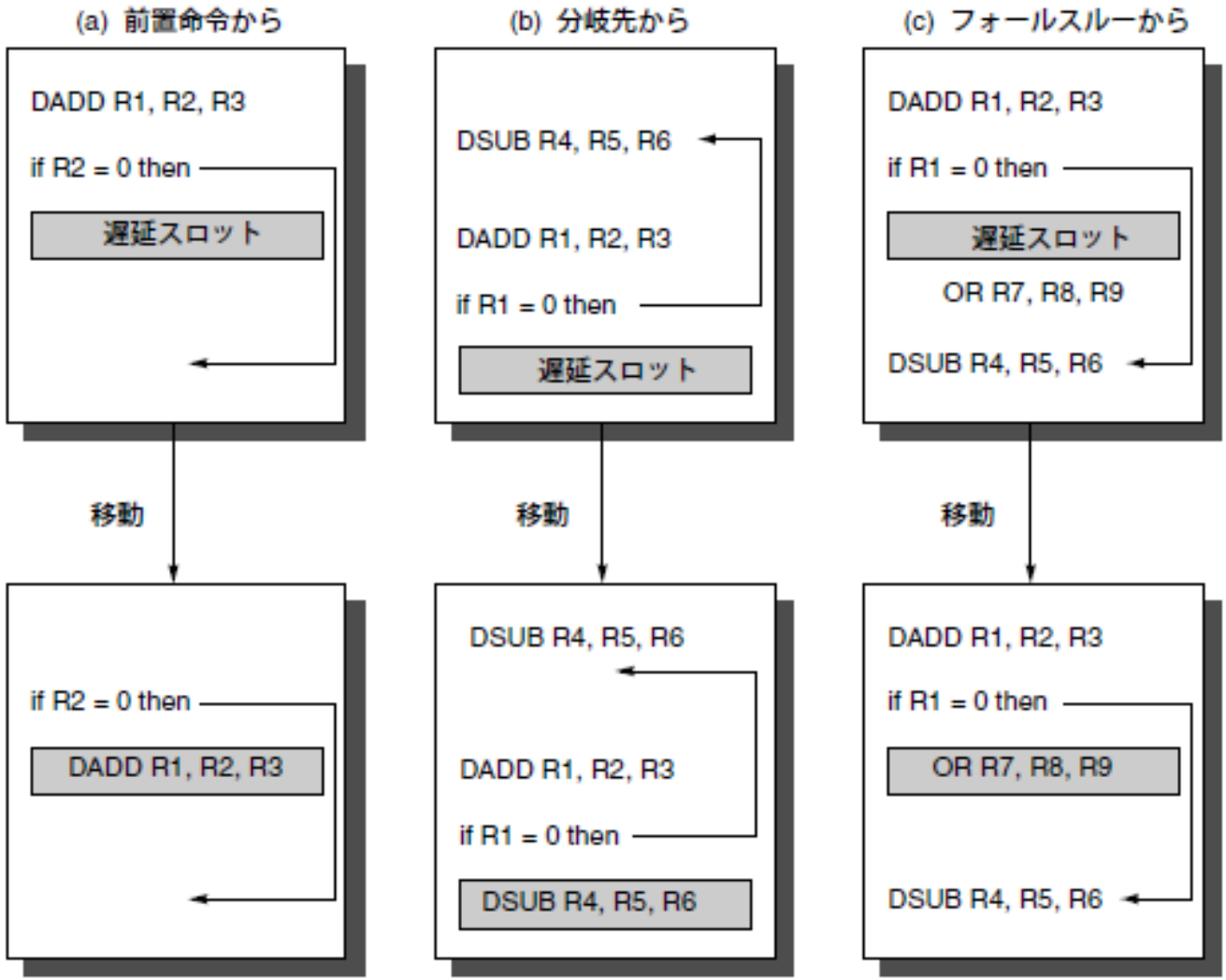
戦略4：遅延分岐 (delayed branch), MIPSが採用

- 分岐命令の後続の幾つかの命令を実行した後に，分岐する．1サイクルの遅延を持つ命令実行順は次の通り．
  - 分岐命令を実行
  - 分岐命令の次アドレスの命令を実行
  - 分岐成立では，飛び先アドレスの命令を実行
    （不成立では，分岐命令の次の次のアドレスの命令を実行）
- 分岐命令の後続の幾つかの命令を実行した後に分岐．分岐命令によるストールは生じない．

Untaken分岐命令	IF	ID	EX	MEM	WB				
分岐遅延命令 ($i+1$)		IF	ID	EX	MEM	WB			
命令 $i+2$			IF	ID	EX	MEM	WB		
命令 $i+3$				IF	ID	EX	MEM	WB	
命令 $i+4$					IF	ID	EX	MEM	WB

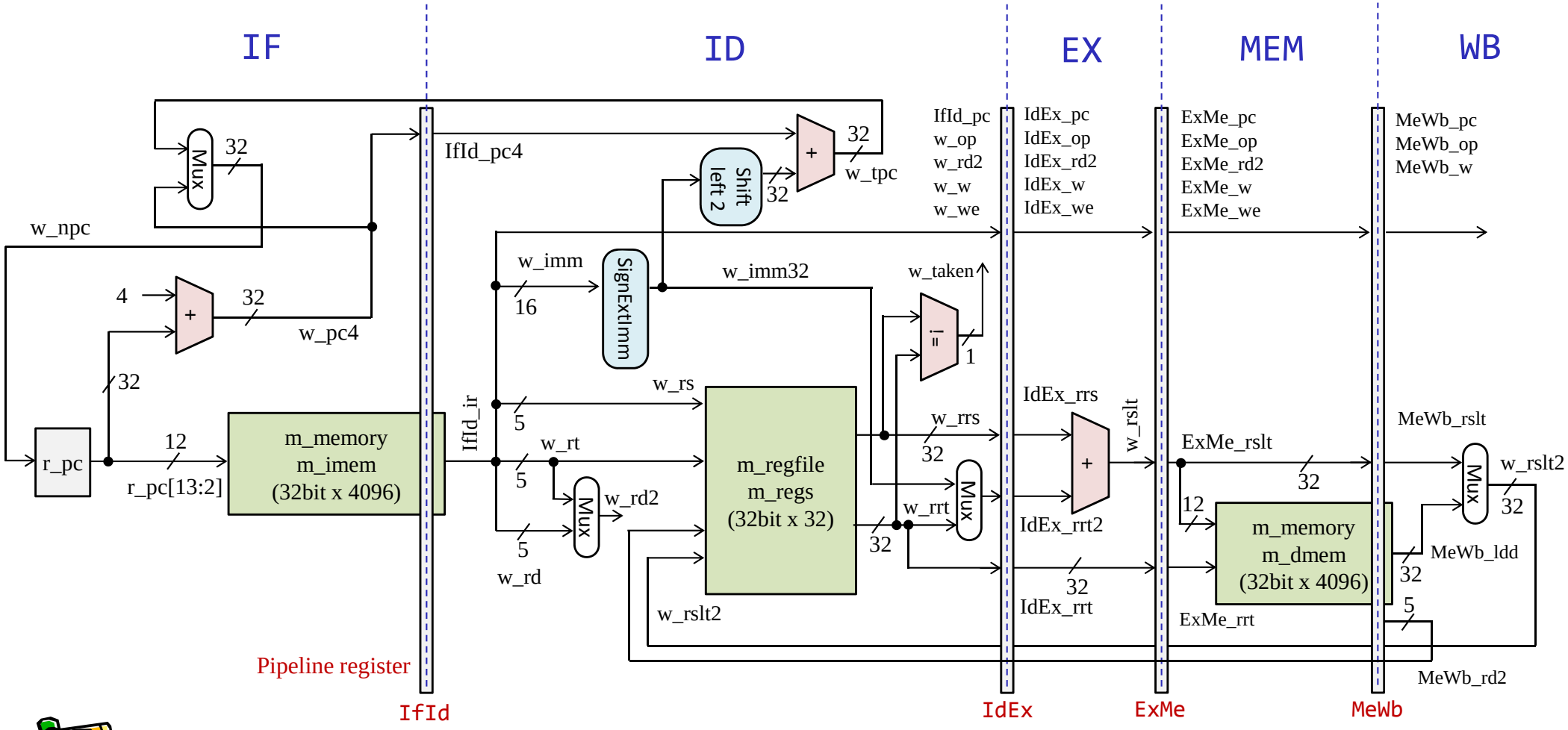
Taken分岐命令	IF	ID	EX	MEM	WB				
分岐遅延命令 ($i+1$)		IF	ID	EX	MEM	WB			
分岐先			IF	ID	EX	MEM	WB		
分岐先 + 1				IF	ID	EX	MEM	WB	
分岐先 + 2					IF	ID	EX	MEM	WB

遅延分岐スロットのスケジューリング



Inside module m_proc11 (pipelined processor)

- add, addi, lw, sw, bne, halt命令に対応したデータハザードを考慮しないパイプライン版
- 最も遅延の長い（長い時間を要する）ステージがプロセッサの動作周波数を決める.



Inside module m_proc11 (pipelined about 100MHz)

code140.v

```
module m_proc11 (w_clk, w_rst, r_rout, r_halt);
  input  wire w_clk, w_rst;
  output reg [31:0] r_rout;
  output reg        r_halt;

  reg  [31:0] IfId_pc4=0;                                    // pipe regs
  reg  [31:0] IdEx_rrs=0, IdEx_rrt=0, IdEx_rrt2=0;           //
  reg  [31:0] ExMe_rslt=0, ExMe_rrt=0;                       //
  reg  [31:0] MeWb_rslt=0;                                   //
  reg   [5:0]             IdEx_op=0,  ExMe_op=0,  MeWb_op=0; //
  reg  [31:0] IfId_pc=0,  IdEx_pc=0,  ExMe_pc=0,  MeWb_pc=0; //
  reg   [4:0] IfId_rd2=0, IdEx_rd2=0, ExMe_rd2=0, MeWb_rd2=0;//
  reg         IfId_w=0,   IdEx_w=0,   ExMe_w=0,   MeWb_w=0;  //
  reg         IfId_we=0,  IdEx_we=0,  ExMe_we=0;             //
  wire [31:0] IfId_ir, MeWb_ldd;                             // note
  /************************** IF stage ********************************/
  wire w_taken;
  wire [31:0] w_tpc;
  reg  [31:0] r_pc  = 0;
  wire [31:0] w_pc4 = r_pc + 4;
  m_memory m_imem (w_clk, r_pc[13:2], 0, 0, IfId_ir);
  always @(posedge w_clk) begin
    r_pc     <= #3 (w_rst | r_halt) ? 0 : (w_taken) ? w_tpc : w_pc4;
    IfId_pc  <= #3 r_pc;
    IfId_pc4 <= #3 w_pc4;
  end
  /************************** ID stage ********************************/
  wire [31:0] w_rrs, w_rrt, w_rslt2;
  wire  [5:0] w_op    = IfId_ir[31:26];
  wire  [4:0] w_rs    = IfId_ir[25:21];
  wire  [4:0] w_rt    = IfId_ir[20:16];
  wire  [4:0] w_rd    = IfId_ir[15:11];
  wire  [4:0] w_rd2   = (w_op!=0) ? w_rt : w_rd;
  wire [15:0] w_imm   = IfId_ir[15:0];
  wire [31:0] w_imm32 = {{16{w_imm[15]}}, w_imm};
  wire [31:0] w_rrt2  = (w_op>6'h5) ? w_imm32 : w_rrt;
  assign      w_tpc   = IfId_pc4 + {w_imm32[29:0], 2'h0};
  assign      w_taken = (w_op==`BNE && w_rrs!=w_rrt);
  m_regfile m_regs (w_clk, w_rs, w_rt, MeWb_rd2, MeWb_w, w_rslt2, w_rrs, w_rrt);
```

```
  always @(posedge w_clk) begin
    IdEx_pc   <= #3 IfId_pc;
    IdEx_op   <= #3 w_op;
    IdEx_rd2  <= #3 w_rd2;
    IdEx_w    <= #3 (w_op==0 || (w_op>6'h5 && w_op<6'h28));
    IdEx_we   <= #3 (w_op>6'h27);
    IdEx_rrs  <= #3 w_rrs;
    IdEx_rrt  <= #3 w_rrt;
    IdEx_rrt2 <= #3 w_rrt2;
  end
  /************************** EX stage ********************************/
  wire [31:0] #10 w_rslt = IdEx_rrs + IdEx_rrt2; // ALU
  always @(posedge w_clk) begin
    ExMe_pc   <= #3 IdEx_pc;
    ExMe_op   <= #3 IdEx_op;
    ExMe_rd2  <= #3 IdEx_rd2;
    ExMe_w    <= #3 IdEx_w;
    ExMe_we   <= #3 IdEx_we;
    ExMe_rslt <= #3 w_rslt;
    ExMe_rrt  <= #3 IdEx_rrt;
  end
  /************************** MEM stage ********************************/
  m_memory m_dmem (w_clk, ExMe_rslt[13:2], ExMe_we, ExMe_rrt, MeWb_ldd);
  always @(posedge w_clk) begin
    MeWb_pc   <= #3 ExMe_pc;
    MeWb_rslt <= #3 ExMe_rslt;
    MeWb_op   <= #3 ExMe_op;
    MeWb_rd2  <= #3 ExMe_rd2;
    MeWb_w    <= #3 ExMe_w;
  end
  /************************** WB stage ********************************/
  assign w_rslt2 = (MeWb_op>6'h19 && MeWb_op<6'h28) ? MeWb_ldd : MeWb_rslt;
  /********************************************************************/
  initial r_halt = 0;
  always @(posedge w_clk) if (MeWb_op==`HALT) r_halt <= 1;
  initial r_rout = 0;
  reg [31:0] r_tmp=0;
  always @(posedge w_clk) r_tmp <= (w_rst) ? 0 : (w_rs==30) ? w_rrs : r_tmp;
  always @(posedge w_clk) r_rout <= r_tmp;
endmodule
```