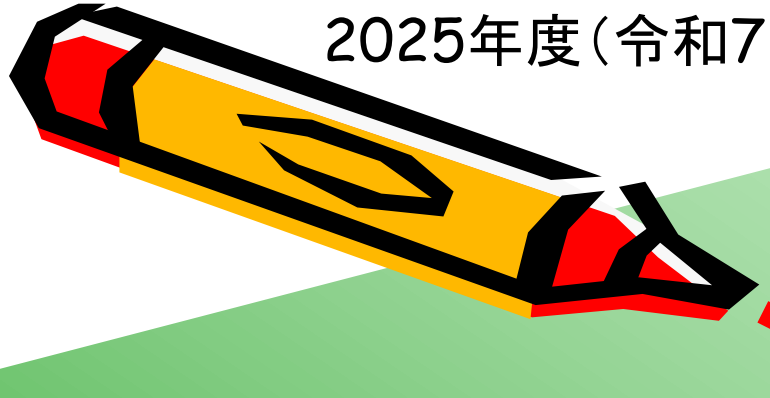


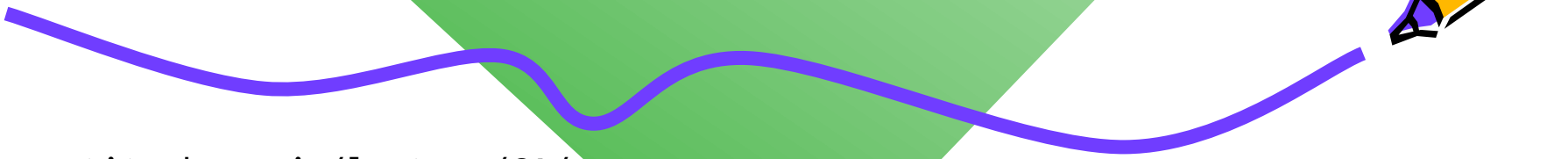
2025年度（令和7年）版

Ver. 2025-10-31a

Course number: CSC.T363

コンピュータアーキテクチャ
Computer Architecture

8.パイプラインプロセッサ
Pipelined Processor

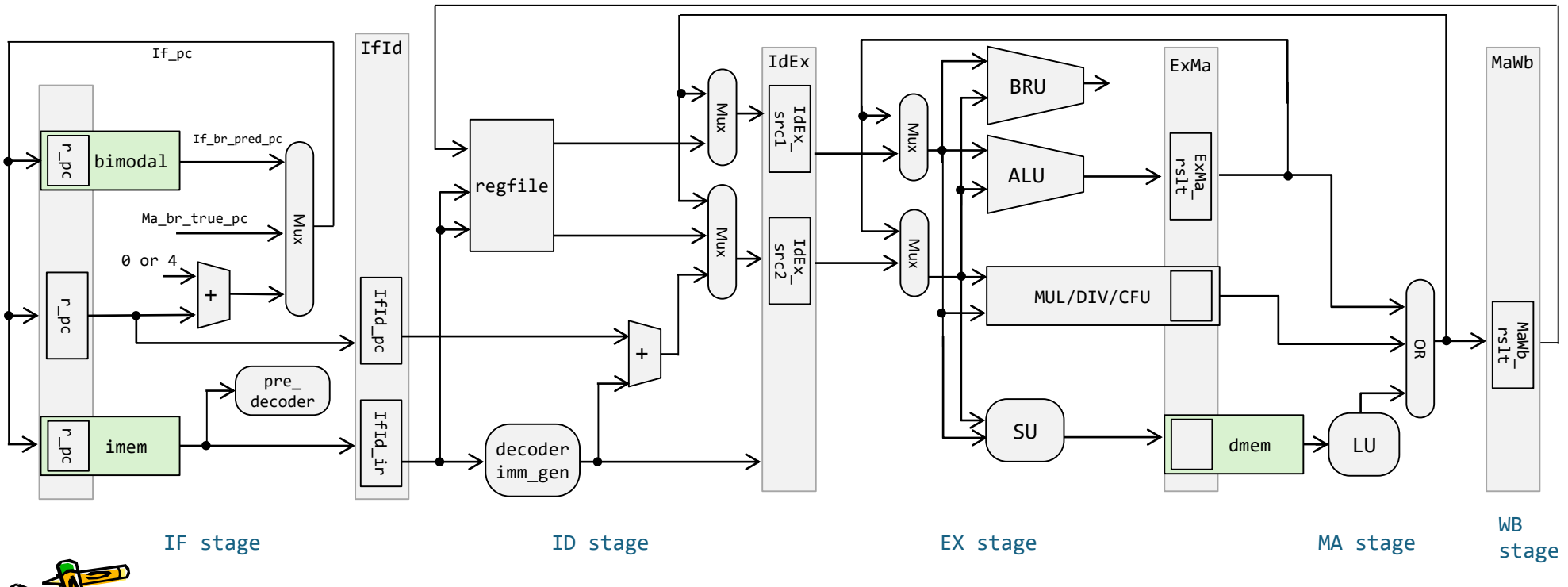


www.arch.cs.titech.ac.jp/lecture/CA/
Tue 13:30-15:10, 15:25-17:05
Fri 13:30-15:10

吉瀬 謙二　情報工学系
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp

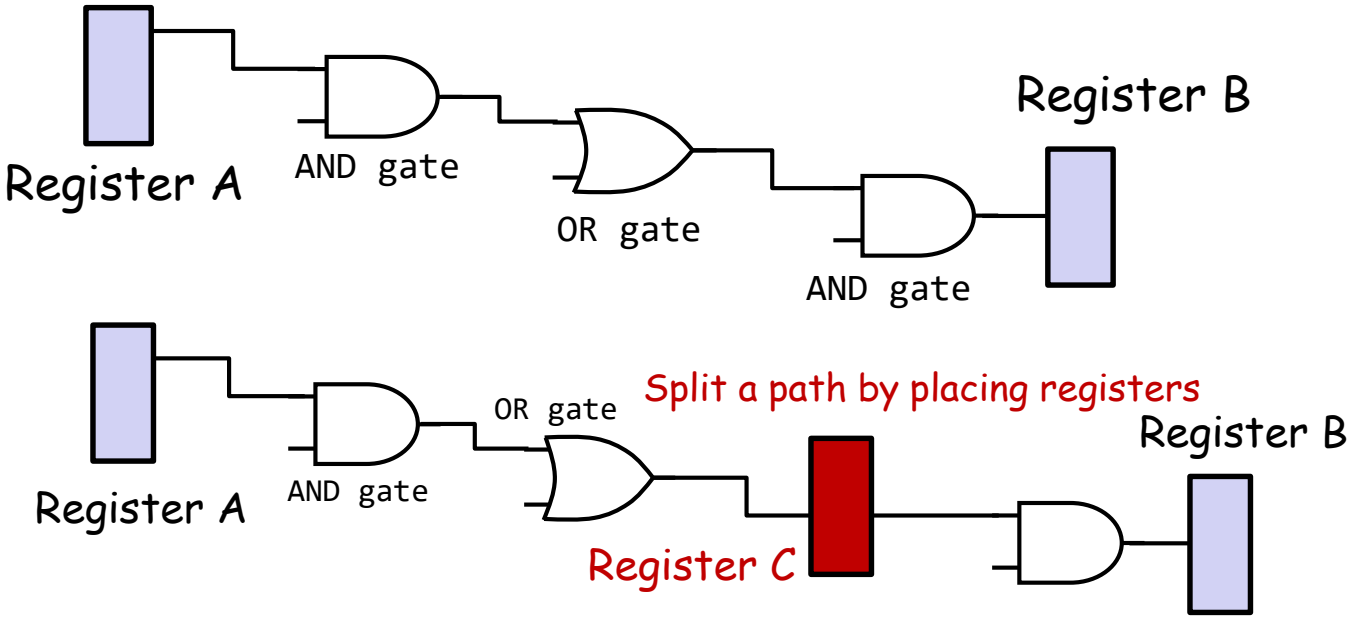
5-stage pipelining processor of CFU Proving Ground

- IF (Instruction Fetch) 命令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。
- EX (Execution) 命令操作の実行またはアドレスの生成を行う。
- MA (Memory Access) 必要であれば、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、結果をレジスタに書き込む。



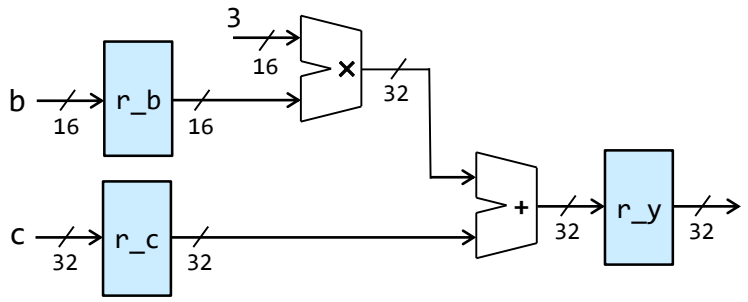
Clock rate is mainly determined by

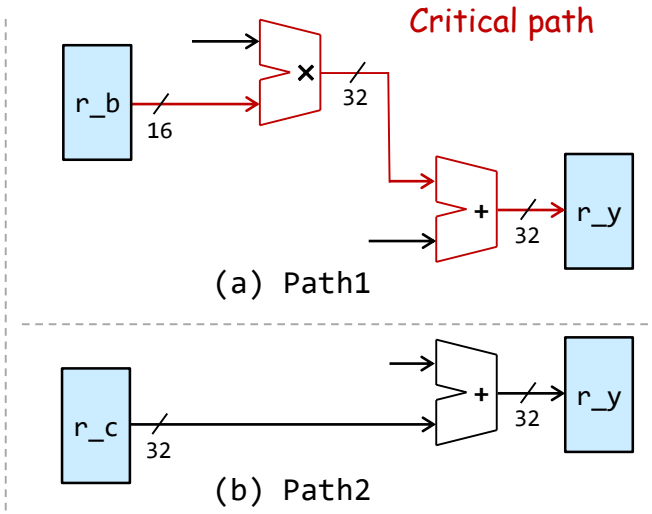
- Switching speed of gates (transistors)
- The number of levels of gates
  - The maximum number of gates cascaded in series in any combinational logics.
  - In this example, the number of levels of gates is 3.
- Wiring delay and fanout



Pipelining example: multiply-add operation (1)

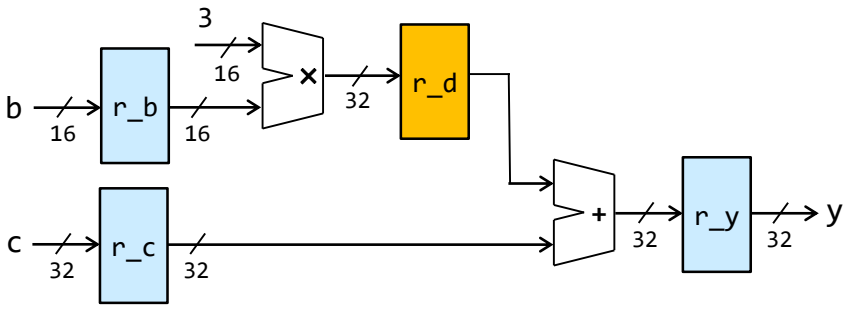
- As an example of pipelining, we will see a multiply-add circuit.
- r_b, r_c are input registers and r_y is output register of the circuit.
- This has two paths named path1 and path2, and path1 is the critical path to determine the maximum operating frequency.

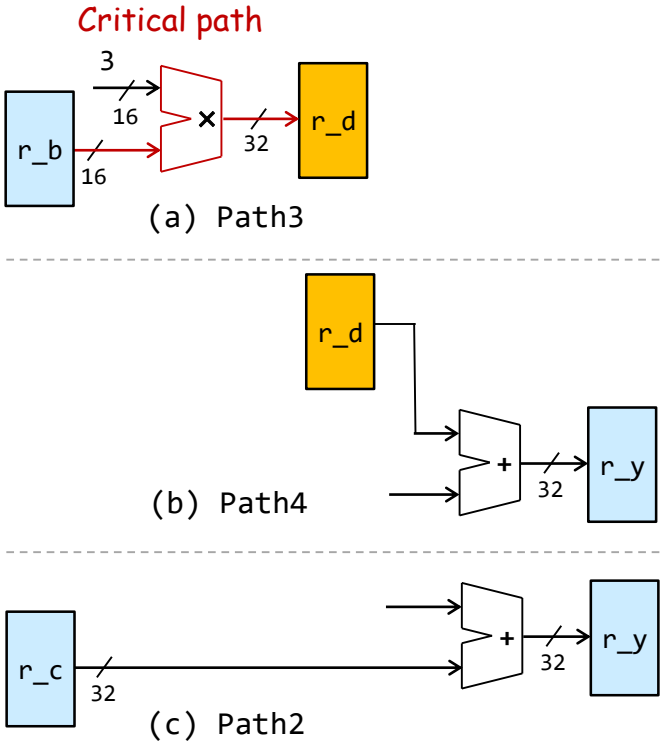




Pipelining example: multiply-add operation (2)

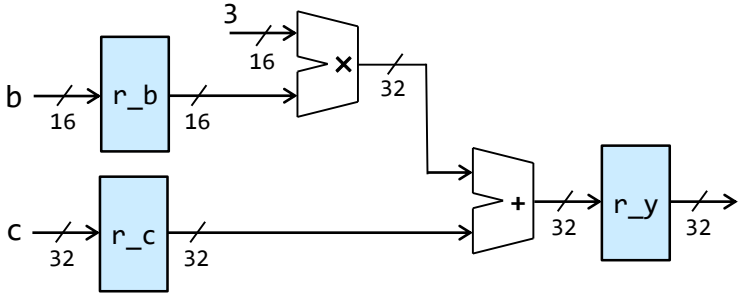
- By inserting register r_d, the critical path can be divided into Path3 and Path4.
- As a result, the new critical path becomes Path3.
- This has the disadvantage that input b and c in the same clock cycle cannot be processed.



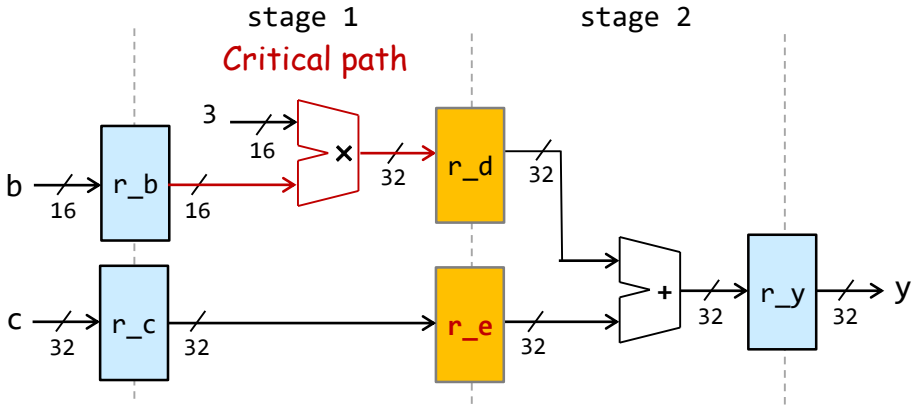


Pipelining example: multiply-add operation (3)

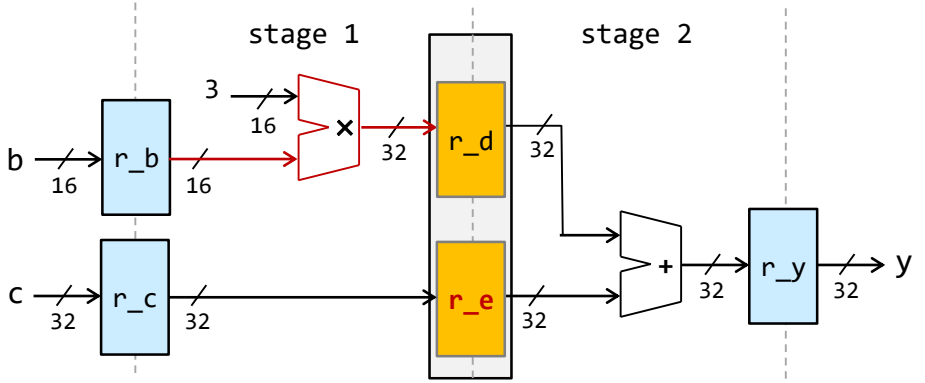
- To overcome this drawback, we insert register r_e.
- This realizes a pipeline with stages 1 and 2.
  A set of registers between two adjacent stages are called a pipeline register.



(a) original multiply-add circuit

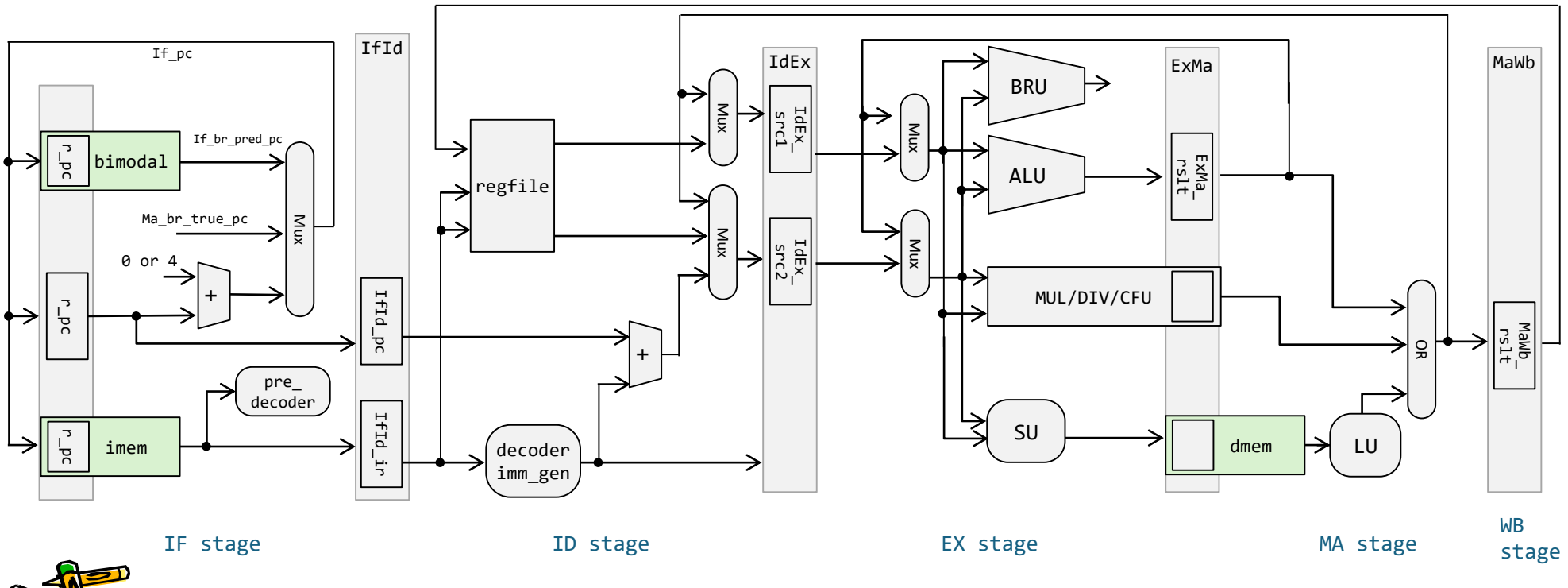


(b) two-stage pipelined circuit



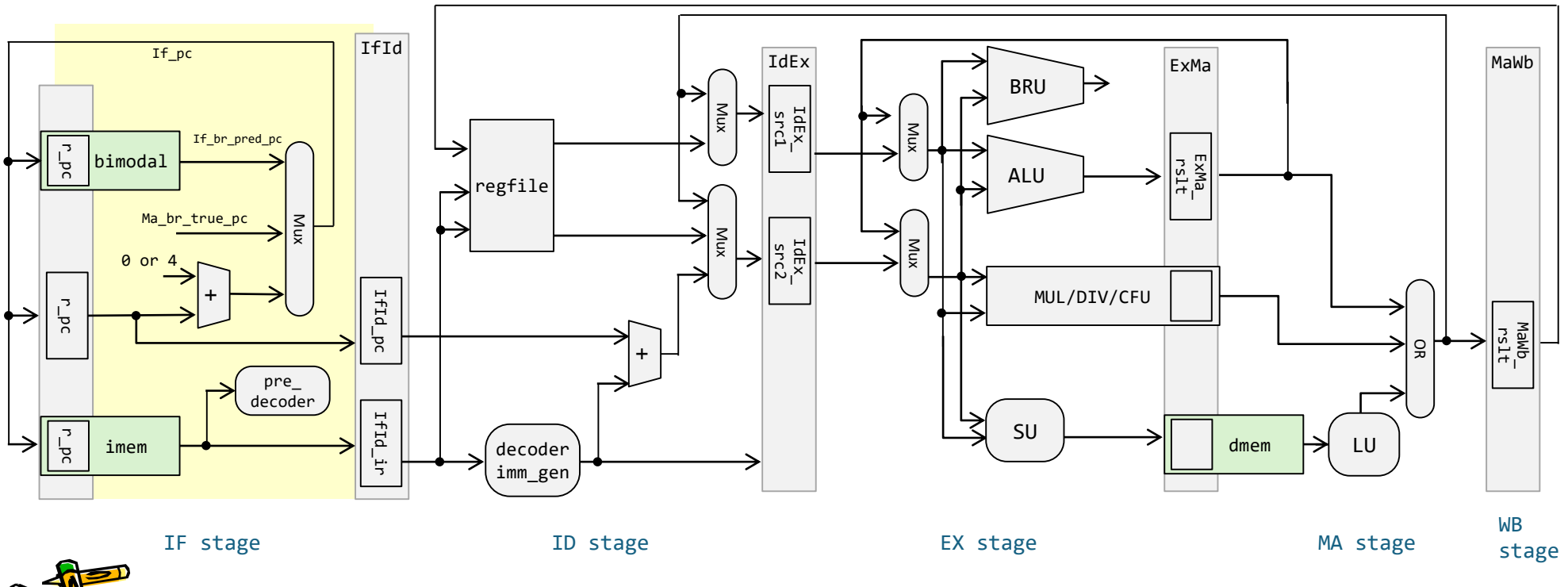
5-stage pipelining processor of CFU Proving Ground

- IF (Instruction Fetch) 命令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。
- EX (Execution) 命令操作の実行またはアドレスの生成を行う。
- MA (Memory Access) 必要であれば、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、結果をレジスタに書き込む。



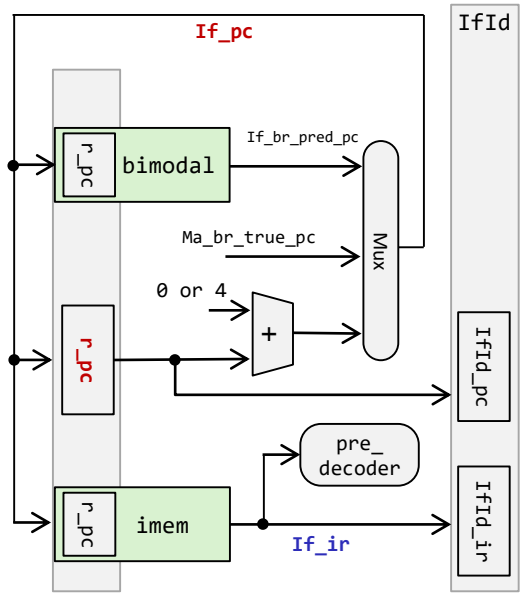
5-stage pipelining processor of CFU Proving Ground

- IF (Instruction Fetch) 命令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。
- EX (Execution) 命令操作の実行またはアドレスの生成を行う。
- MA (Memory Access) 必要であれば、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、結果をレジスタに書き込む。



IF (Instruction Fetch) stage

```
always @(posedge clk_i) if (!w_stall) begin
        r_pc <= If_pc;  // update pc
        if (rst) begin
            IfId_v  <= 0;
            IfId_pc <= 0;
            IfId_ir <= `NOP;
        end else if (!ExMa_stall) begin
            IfId_v               <= If_v;
            IfId_load_muldiv_use <= If_load_muldiv_use;
            if (!IfId_load_muldiv_use) begin
                IfId_pc          <= r_pc;
                IfId_ir          <= If_ir;
                IfId_br_pred_tkn <= If_br_pred_tkn;
                IfId_pat_hist    <= If_pat_hist;
                IfId_instr_type  <= If_instr_type;
                IfId_rf_we       <= If_rf_we;
                IfId_rd          <= If_rd;
                IfId_rs1         <= If_rs1;
                IfId_rs2         <= If_rs2;
            end
        end
    end
```



IF stage

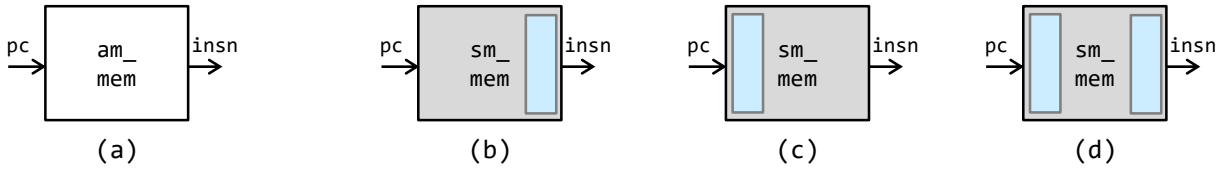
```
//------------------------------------------------------------------------------
// pipeline registers
//------------------------------------------------------------------------------
    // IF: Instruction Fetch
    reg [          `XLEN-1:0] r_pc;  // program counter
```

```
//------------------------------------------------------------------------------
// IF: Instruction Fetch
//------------------------------------------------------------------------------
    wire [`PC_W-1:0] If_pc;  // the program counter of the next clock cycle
    wire [`PC_W-1:0] If_pc_inc;  //
    wire If_pc_stall;
    wire [1:0] If_pat_hist;
    wire If_br_pred_tkn;
    wire [31:0] If_br_pred_pc;
    wire [`ITYPE_W-1:0] If_instr_type;
    wire If_rf_we;
    wire [4:0] If_rd;
    wire [4:0] If_rs1;
    wire [4:0] If_rs2;
    wire [31:0] If_ir;  // instruction from imem

    assign ibus_araddr_o = If_pc;  // read address of imem
    assign If_ir = ibus_rdata_i;  // instruction from imem

    bimodal bimodal (
        .clk_i        (clk_i),           // input  wire
        .rst_i        (rst),             // input  wire
        .stall_i      (w_stall),         // input  wire
        .raddr_i      (If_pc),           // input  wire [`XLEN-1:0]
        .pat_hist_o   (If_pat_hist),     // output reg        [1:0]
        .br_pred_tkn_o(If_br_pred_tkn),  // output wire
        .br_pred_pc_o (If_br_pred_pc),   // output reg  [`XLEN-1:0]
        .br_tkn_i     (Ma_br_tkn),       // input  wire
        .br_tsfr_i    (ExMa_j_b_insn),   // input  wire
        .waddr_i      (ExMa_pc),         // input  wire [`XLEN-1:0]
        .pat_hist_i   (ExMa_pat_hist),   // input  wire       [1:0]
        .br_tkn_pc_i  (ExMa_br_tkn_pc)   // input  wire [`XLEN-1:0]
    );

    assign If_pc_stall = ExMa_stall || IfId_load_muldiv_use;
    assign If_pc_inc = (If_pc_stall) ? 0 : 4;
    assign If_pc = (w_stall) ? r_pc :
                   (Ma_br_misp                   ) ? Ma_br_true_pc :
                   (!If_pc_stall & If_br_pred_tkn) ? If_br_pred_pc : r_pc+If_pc_inc;
```

同期メモリ、非同期メモリ

```
module am_mem (
    input  wire             clk,
    input  wire [`ADDRW+1:0] addr_i,
    output wire [31:0]       data_o
);
    reg [31:0] mem[0:`MEM_ENTRIES-1];

    assign data_o = mem[addr_i];
endmodule
```

(a) 非同期メモリ




```
module sm_mem (
    input  wire             clk,
    input  wire [`ADDRW+1:0] addr_i,
    output wire [31:0]       data_o
);
    reg [31:0] mem[0:`MEM_ENTRIES-1];

    reg [31:0] rdata;
    always @(posedge clk) begin
        r_data <= mem[addr_i];
    end
    assign data_o = r_data;
endmodule
```

(b) 同期メモリ

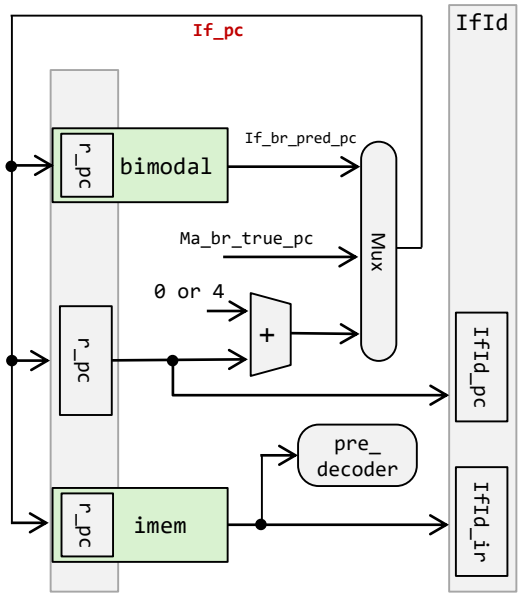
```
module sm_mem (
    input  wire             clk,
    input  wire [`ADDRW+1:0] addr_i,
    output wire [31:0]       data_o
);
    reg [31:0] mem[0:`MEM_ENTRIES-1];

    reg [`ADDRW+1:0] r_addr;
    always @(posedge clk) begin
        r_addr <= addr_i;
    end
    assign data_o = mem[r_addr];
endmodule
```

(c) 同期メモリ

```
module sm_mem (
    input  wire             clk,
    input  wire [`ADDRW+1:0] addr_i,
    output wire [31:0]       data_o
);
    reg [31:0] mem[0:`MEM_ENTRIES-1];

    reg [`ADDRW+1:0] r_addr;
    reg [31:0] r_data;
    always @(posedge clk) begin
        r_addr <= addr_i;
        r_data <= mem[r_addr];
    end
    assign data_o = r_data;
endmodule
```

(d) 同期メモリ

IF (Instruction Fetch) stage



IF stage

```
    cpu cpu (
        .clk_i         (clk),         // input  wire
        .rst_i         (rst),         // input  wire
        .stall_i       (0),           // input  wire
        .ibus_araddr_o (imem_raddr),  // output wire [`IBUS_ADDR_WIDTH-1:0]
        .ibus_rdata_i  (imem_rdata),  // input  wire [`IBUS_DATA_WIDTH-1:0]
        .dbus_addr_o   (dbus_addr),   // output wire [`DBUS_ADDR_WIDTH-1:0]
        .dbus_wvalid_o (dbus_we),     // output wire
        .dbus_wdata_o  (dbus_wdata),  // output wire [`DBUS_DATA_WIDTH-1:0]
        .dbus_wstrb_o  (dbus_wstrb),  // output wire [`DBUS_STRB_WIDTH-1:0]
        .dbus_rdata_i  (dbus_rdata)   // input  wire [`DBUS_DATA_WIDTH-1:0]
    );

    m_imem imem (
        .clk_i   (clk),         // input  wire
        .raddr_i (imem_raddr),  // input  wire [ADDR_WIDTH-1:0]
        .rdata_o (imem_rdata)   // output reg  [DATA_WIDTH-1:0]
    );
```

```
module m_imem (
    input  wire        clk_i,
    input  wire [31:0] raddr_i,
    output wire [31:0] rdata_o
);

    (* ram_style = "block" *) reg [31:0] imem[0:`IMEM_ENTRIES-1];
    `include "memi.txt"

    wire [`IMEM_ADDRW-1:0] valid_raddr = raddr_i[`IMEM_ADDRW+1:2];

    reg [31:0] rdata = 0;
    always @(posedge clk_i) begin
        rdata <= imem[valid_raddr];
    end
    assign rdata_o = rdata;
endmodule
```

5-stage pipelining processor of CFU Proving Ground

- 次のコマンドで、CFU Proving Ground をクローンする。
  - cd ~/ca
  - mkdir lec8
  - cd lec8
  - git clone https://github.com/archlab-sciencetokyo/CFU-Proving-Ground
  - cd CFU-Proving-Ground
- 次のコマンドで cp4 の Makefile と config.vh をコピーする。
  - cp ../../cp4/CFU-Proving-Ground/Makefile .
  - cp ../../cp4/CFU-Proving-Ground/config.vh .
- 次のコマンドで、CFU PG を初期化する。
  - make init
- 次のコマンドで、CFU PG をコンパイルする。
  - make
- 次のコマンドで、CFU PG の GUI シミュレーションを実行する。
  - make drun

シンプルなアセンブリ言語のコードのシミュレーション

```
reg r_rst = 0;
always @(posedge clk) r_rst <= !m0.rst;
always @(posedge clk) if (r_rst) begin
    $write(" %06d: %x ", minstret, m0.cpu.r_pc);
    if (!m0.cpu.IfId_v) $write("-------- "); else if(m0.cpu.stall) $write("ssssssss "); else $write("%x ", m0.cpu.IfId_pc);
    if (!m0.cpu.IdEx_v) $write("-------- "); else if(m0.cpu.stall) $write("ssssssss "); else $write("%x ", m0.cpu.IdEx_pc);
    if (!m0.cpu.ExMa_v) $write("-------- "); else if(m0.cpu.stall) $write("ssssssss "); else $write("%x ", m0.cpu.ExMa_pc);
    if (!m0.cpu.MaWb_v) $write("--------:"); else if(m0.cpu.stall) $write(«ssssssss:"); else $write("%x:", m0.cpu.MaWb_pc);
    if (m0.cpu.stall_i) $write(" stall");
    $write("\n");
end
```

- top.v を編集して、上図のコードを追加する。
- app/crt0.s を下図のコードに編集する。
- 次のコマンドで実行
  - make
  - make run

```
    .equ TO_HOST,    0x80000000
    .equ CMD_FINISH, 0x00020000
    .align 4
    .section .text.init
    .globl _start
_start:
    li      x1, 2
    li      x2, 0
    li      t0, TO_HOST
    li      t1, CMD_FINISH
    sw      t1, 0(t0)
L1: j       L1
```

```
./obj_dir/top
 000000: 00000000 -------- -------- -------- --------:
 000000: 00000004 00000000 -------- -------- --------:
 000000: 00000008 00000004 00000000 -------- --------:
 000000: 0000000c 00000008 00000004 00000000 --------:
 000001: 00000010 0000000c 00000008 00000004 00000000:
 000002: 00000014 00000010 0000000c 00000008 00000004:
 000003: 00000018 00000014 00000010 0000000c 00000008:
 000004: 0000001c 00000018 00000014 00000010 0000000c:
- top.v:47: Verilog $finish

===> mcycle                                :          8
===> minstret                              :          4
===> Total number of branch predictions    :          0
===> Total number of branch mispredictions :          0
===> simulation finish!!

- S i m u l a t i o n   R e p o r t: Verilator 5.026 2024-06-15
- Verilator: $finish at 105ps; walltime 0.000 s; speed 0.000 s/s
- Verilator: cpu 0.000 s on 1 threads; alloced 121 MB
```

シンプルなアセンブリ言語のコードのシミュレーション

- app/crt0.s を下図のコードに編集する。
- 次のコマンドで実行
  - make
  - make run

```
    .equ TO_HOST,    0x80000000
    .equ CMD_FINISH, 0x00020000
    .align 4
    .section .text.init
    .globl _start
_start:
    li      x1, 2
    li      x2, 0
L2: addi    x2, x2, 1
    bne     x1, x2, L2
    li      t0, TO_HOST
    li      t1, CMD_FINISH
    sw      t1, 0(t0)
L1: j       L1
```

```
./obj_dir/top
 000000: 00000000 -------- -------- -------- --------:
 000000: 00000004 00000000 -------- -------- --------:
 000000: 00000008 00000004 00000000 -------- --------:
 000000: 0000000c 00000008 00000004 00000000 --------:
 000001: 00000010 0000000c 00000008 00000004 00000000:
 000002: 00000014 00000010 0000000c 00000008 00000004:
 000003: 00000018 00000014 00000010 0000000c 00000008:
 000004: 00000008 -------- -------- -------- 0000000c:
 000004: 0000000c 00000008 -------- -------- --------:
 000004: 00000010 0000000c 00000008 -------- --------:
 000004: 00000014 00000010 0000000c 00000008 --------:
 000005: 00000018 00000014 00000010 0000000c 00000008:
 000006: 0000001c 00000018 00000014 00000010 0000000c:
 000007: 00000020 0000001c 00000018 00000014 00000010:
 000008: 00000024 00000020 0000001c 00000018 00000014:
- top.v:47: Verilog $finish

===> mcycle                                :         15
===> minstret                              :          8
===> Total number of branch predictions    :          2
===> Total number of branch mispredictions :          1
===> simulation finish!!
```

```
assign If_pc_stall = ExMa_stall || IfId_load_muldiv_use;
assign If_pc_inc = (If_pc_stall) ? 0 : 4;
assign If_pc = (w_stall) ? r_pc :
               (Ma_br_misp                    ) ? Ma_br_true_pc :
               (!If_pc_stall & If_br_pred_tkn) ? If_br_pred_pc :
 r_pc+If_pc_inc;
```

シンプルなアセンブリ言語のコードのシミュレーション

- app/crt0.s を下図のコードに編集し、実行する。
  - make; make run

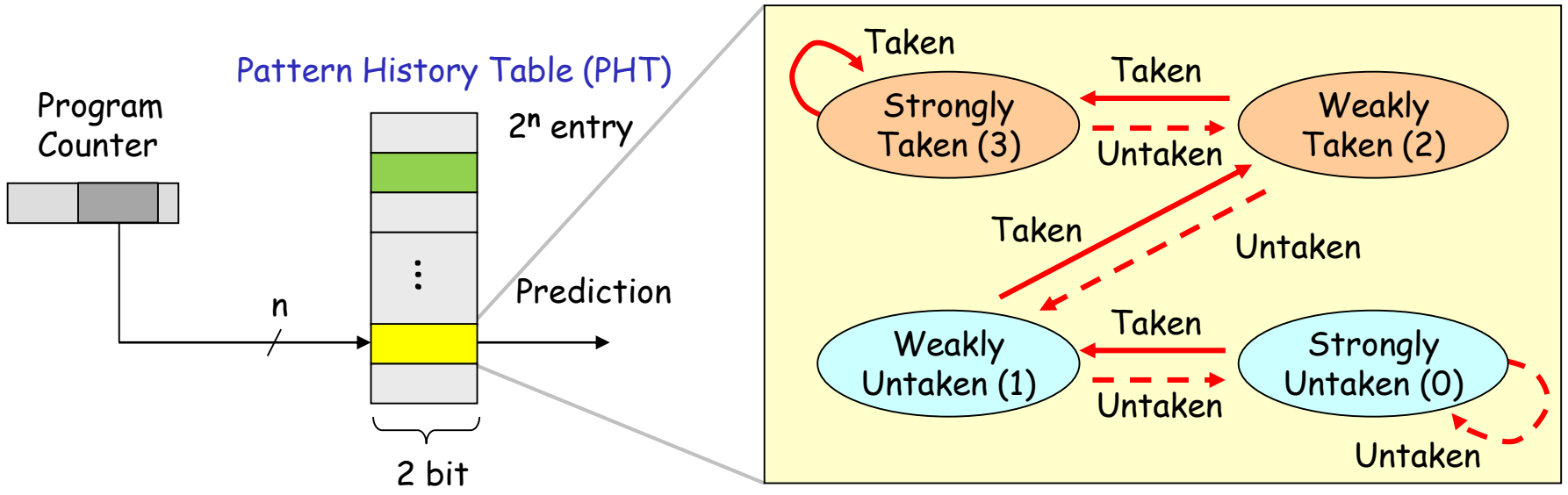
```
    .equ TO_HOST,    0x80000000
    .equ CMD_FINISH, 0x00020000
    .align 4
    .section .text.init
    .globl _start
_start:
    li      x1, 8
    li      x2, 0
L2: addi    x2, x2, 1
    bne     x1, x2, L2
    li      t0, TO_HOST
    li      t1, CMD_FINISH
    sw      t1, 0(t0)
L1: j       L1
```

```
    bimodal bimodal (
        .clk_i        (clk_i),          // input  wire
        .rst_i        (rst),            // input  wire
        .stall_i      (w_stall),        // input  wire
        .raddr_i      (If_pc),          // input  wire [`XLEN-1:0]
        .pat_hist_o   (If_pat_hist),    // output reg        [1:0]
        .br_pred_tkn_o(If_br_pred_tkn), // output wire
        .br_pred_pc_o (If_br_pred_pc),  // output reg  [`XLEN-1:0]
        .br_tkn_i     (Ma_br_tkn),      // input  wire
        .br_tsfr_i    (ExMa_j_b_insn),  // input  wire
        .waddr_i      (ExMa_pc),        // input  wire [`XLEN-1:0]
        .pat_hist_i   (ExMa_pat_hist),  // input  wire       [1:0]
        .br_tkn_pc_i  (ExMa_br_tkn_pc)  // input  wire [`XLEN-1:0]
    );
```

```
./obj_dir/top
000000: 00000000 -------- -------- -------- --------:
000000: 00000004 00000000 -------- -------- --------:
000000: 00000008 00000004 00000000 -------- --------:
000000: 0000000c 00000008 00000004 00000000 --------:
000001: 00000010 0000000c 00000008 00000004 00000000:
000002: 00000014 00000010 0000000c 00000008 00000004:
000003: 00000018 00000014 00000010 0000000c 00000008:
000004: 00000008 -------- -------- -------- 0000000c:
000004: 0000000c 00000008 -------- -------- --------:
000004: 00000010 0000000c 00000008 -------- --------:
000004: 00000014 00000010 0000000c 00000008 --------:
000005: 00000018 00000014 00000010 0000000c 00000008:
000006: 00000008 -------- -------- -------- 0000000c:
000006: 0000000c 00000008 -------- -------- --------:
000006: 00000008 0000000c 00000008 -------- --------:
000006: 0000000c 00000008 0000000c 00000008 --------:
000007: 00000008 0000000c 00000008 0000000c 00000008:
000008: 0000000c 00000008 0000000c 00000008 0000000c:
000009: 00000008 0000000c 00000008 0000000c 00000008:
000010: 0000000c 00000008 0000000c 00000008 0000000c:
000011: 00000008 0000000c 00000008 0000000c 00000008:
000012: 0000000c 00000008 0000000c 00000008 0000000c:
000013: 00000008 0000000c 00000008 0000000c 00000008:
000014: 0000000c 00000008 0000000c 00000008 0000000c:
000015: 00000008 0000000c 00000008 0000000c 00000008:
000016: 0000000c 00000008 0000000c 00000008 0000000c:
000017: 00000008 0000000c 00000008 0000000c 00000008:
000018: 00000010 -------- -------- -------- 0000000c:
000018: 00000014 00000010 -------- -------- --------:
000018: 00000018 00000014 00000010 -------- --------:
000018: 0000001c 00000018 00000014 00000010 --------:
000019: 00000020 0000001c 00000018 00000014 00000010:
000020: 00000024 00000020 0000001c 00000018 00000014:
- top.v:47: Verilog $finish

===> mcycle                                :         33
===> minstret                              :         20
===> Total number of branch predictions    :          8
===> Total number of branch mispredictions :          3
===> simulation finish!!
```

Simple branch predictor: bimodal

- Program has many static branch instructions. The behavior may depend on each branch. Use plenty of counters (PHT) and assign a counter for a branch instruction.
- How to predict
  - Select a 2-bit counter using PC, and it predicts 1 for taken if the MSB of the register is one; otherwise, it predicts 0 for untaken.
- How to update
  - Select a counter using PC, then update the counter in the same way as 2-bit counter.



CFU PG の bimodal branch predictor

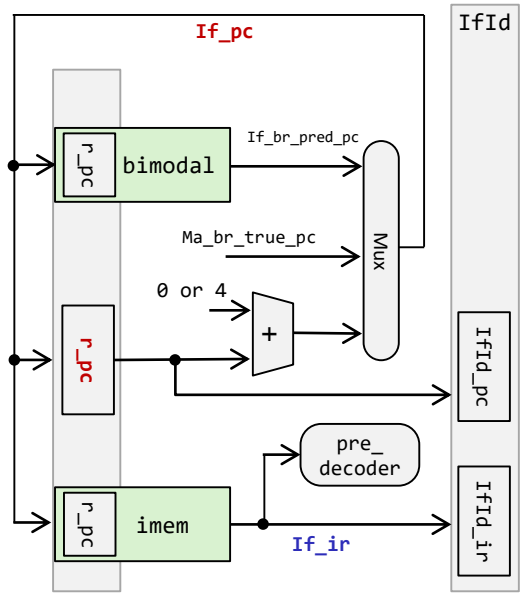
```
module bimodal (
    input  wire             clk_i,
    input  wire             rst_i,
    input  wire             stall_i,
    input  wire [     31:0] raddr_i,
    output wire [      1:0] pat_hist_o,
    output wire             br_pred_tkn_o,
    output wire [`PC_W-1:0] br_pred_pc_o,
    input  wire             br_tkn_i,
    input  wire             br_tsfr_i,
    input  wire [     31:0] waddr_i,
    input  wire [      1:0] pat_hist_i,
    input  wire [     31:0] br_tkn_pc_i
);

    integer i;
    (* ram_style = "block" *) reg [`PC_W-1:0] btb[0:`BTB_ENTRY-1];  // BTB, branch target buf
    initial for (i = 0; i < `BTB_ENTRY; i = i + 1) btb[i] = 0;  // init with weak untaken

    wire [1:0] w_cnt = (br_tkn_i) ? pat_hist_i + (pat_hist_i < 3) : pat_hist_i - (pat_hist_i > 0);
    wire [`BTB_IDXW-1:0] btb_ridx = raddr_i[`BTB_IDXW+`BTB_OSTW-1:`BTB_OSTW];
    wire [`BTB_IDXW-1:0] btb_widx = waddr_i[`BTB_IDXW+`BTB_OSTW-1:`BTB_OSTW];

    reg [31:0] r_btb_entry;
    always @(posedge clk_i) if (!stall_i) begin
        r_btb_entry <= btb[btb_ridx];
        if (br_tsfr_i) begin
            btb[btb_widx] <= {br_tkn_pc_i[`PC_W-1:2], w_cnt};  // lower tow bits for counter
        end
    end

    assign pat_hist_o    = r_btb_entry[1:0];
    assign br_pred_tkn_o = r_btb_entry[1];
    assign br_pred_pc_o  = {r_btb_entry[`PC_W-1:2], 2'b0};
endmodule
```

IF (Instruction Fetch) stage

```
always @(posedge clk_i) if (!w_stall) begin
        r_pc <= If_pc;  // update pc
        if (rst) begin
            IfId_v  <= 0;
            IfId_pc <= 0;
            IfId_ir <= `NOP;
        end else if (!ExMa_stall) begin
            IfId_v               <= If_v;
            IfId_load_muldiv_use <= If_load_muldiv_use;
            if (!IfId_load_muldiv_use) begin
                IfId_pc          <= r_pc;
                IfId_ir          <= If_ir;
                IfId_br_pred_tkn <= If_br_pred_tkn;
                IfId_pat_hist    <= If_pat_hist;
                IfId_instr_type  <= If_instr_type;
                IfId_rf_we       <= If_rf_we;
                IfId_rd          <= If_rd;
                IfId_rs1         <= If_rs1;
                IfId_rs2         <= If_rs2;
            end
        end
    end
```

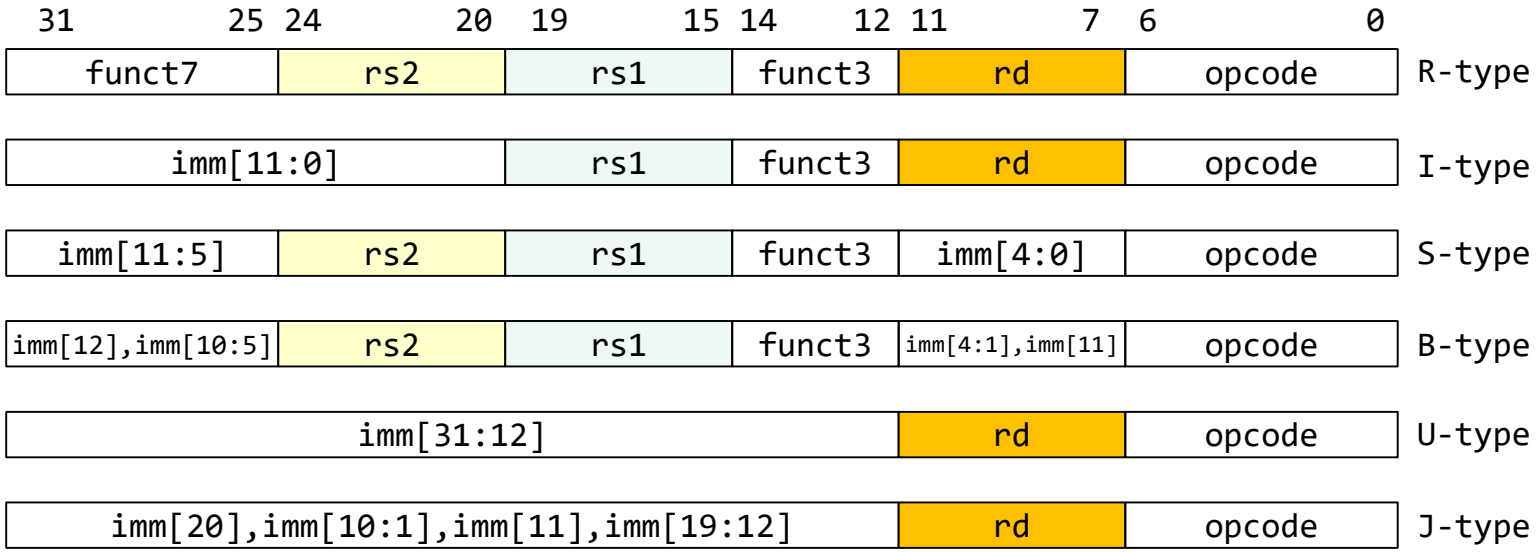
```
    pre_decoder pre_decoder (
        .ir_i        (If_ir),          // input  wire         [31:0]
        .instr_type_o(If_instr_type),  // output wire [`ITYPE_W-1:0]
        .rf_we_o     (If_rf_we),       // output wire
        .rd_o        (If_rd),          // output wire          [4:0]
        .rs1_o       (If_rs1),         // output wire          [4:0]
        .rs2_o       (If_rs2)          // output wire          [4:0]
    );

    wire If_load_muldiv_use = IfId_v && !Ma_br_misp && !IfId_load_muldiv_use
                              && (Id_lsu_ctrl[`LSU_CTRL_IS_LOAD] ||
                                  Id_mul_ctrl[`MUL_CTRL_IS_MUL] ||
                                  Id_div_ctrl[`DIV_CTRL_IS_DIV] ||
                                  Id_cfu_ctrl[`CFU_CTRL_IS_CFU]  )
                              && IfId_rf_we && ((IfId_rd==If_rs1) || (IfId_rd==If_rs2));
```

```
module pre_decoder (
    input  wire [31:0] ir_i,
    output wire [ 2:0] instr_type_o,
    output wire        rf_we_o,
    output wire [ 4:0] rd_o,
    output wire [ 4:0] rs1_o,
    output wire [ 4:0] rs2_o
);

    wire [4:0] opcode = ir_i[6:2];
    assign instr_type_o = (opcode == 5'b01101) ? `U_TYPE :  // LUI
        (opcode == 5'b00101) ? `U_TYPE :  // AUIPC
        (opcode == 5'b11011) ? `J_TYPE :  // JAL
        (opcode == 5'b11001) ? `I_TYPE :  // JALR
        (opcode == 5'b11000) ? `B_TYPE :  // BRANCH
        (opcode == 5'b00000) ? `I_TYPE :  // LOAD
        (opcode == 5'b01000) ? `S_TYPE :  // STORE
        (opcode == 5'b00100) ? `I_TYPE :  // OP-IMM
        (opcode == 5'b01100) ? `R_TYPE :  // OP
        (opcode == 5'b00010) ? `R_TYPE : `NONE_TYPE;  // CUSTOM-0 : NONE

    assign rd_o = ((instr_type_o == `S_TYPE) | (instr_type_o == `B_TYPE)) ? 0 : ir_i[11:7];
    assign rs1_o = ((instr_type_o == `U_TYPE) | (instr_type_o == `J_TYPE)) ? 0 : ir_i[19:15];
    assign rs2_o = ((instr_type_o==`I_TYPE) |
                    (instr_type_o==`U_TYPE) | (instr_type_o==`J_TYPE)) ? 0 : ir_i[24:20];
    assign rf_we_o = (rd_o != 0);
endmodule
```



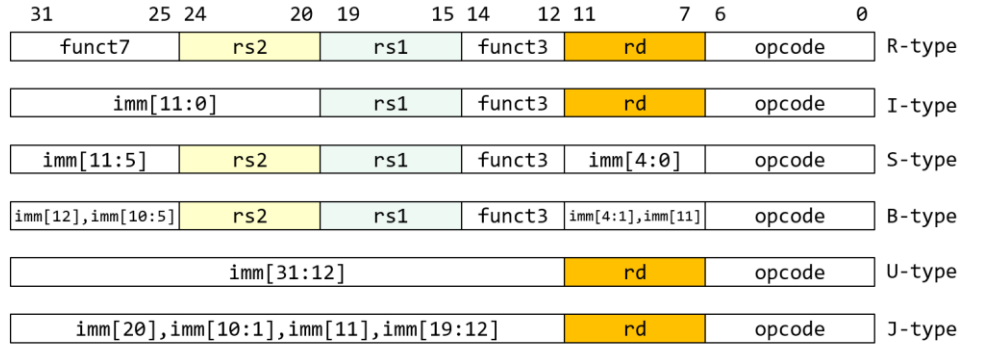
IF stage

RISC-V RV32I instruction format



opcode5[2:0] / opcode5[4:3]	000	001	010	011	100	101	110
00	LOAD (I-type)	LOAD-FP	custom-0	MISC-MEM (I-type)	OP-IMM (I-type)	AUIPC (U-type)	OP-IMM-32
01	STORE (S-type)	STORE-FP	custom-1	AMO	OP (R-type)	LUI (U-type)	OP-32
10	MADD	MSUB	NMSUB	NMADD	OP-FP	OP-V	custom-2/ rv128
11	BRANCH (B-type)	JALR (I-type)	reserved	JAL (J-type)	SYSTEM (I-type)	reserved	custom-3/ rv128

RISC-V RV32I instruction format

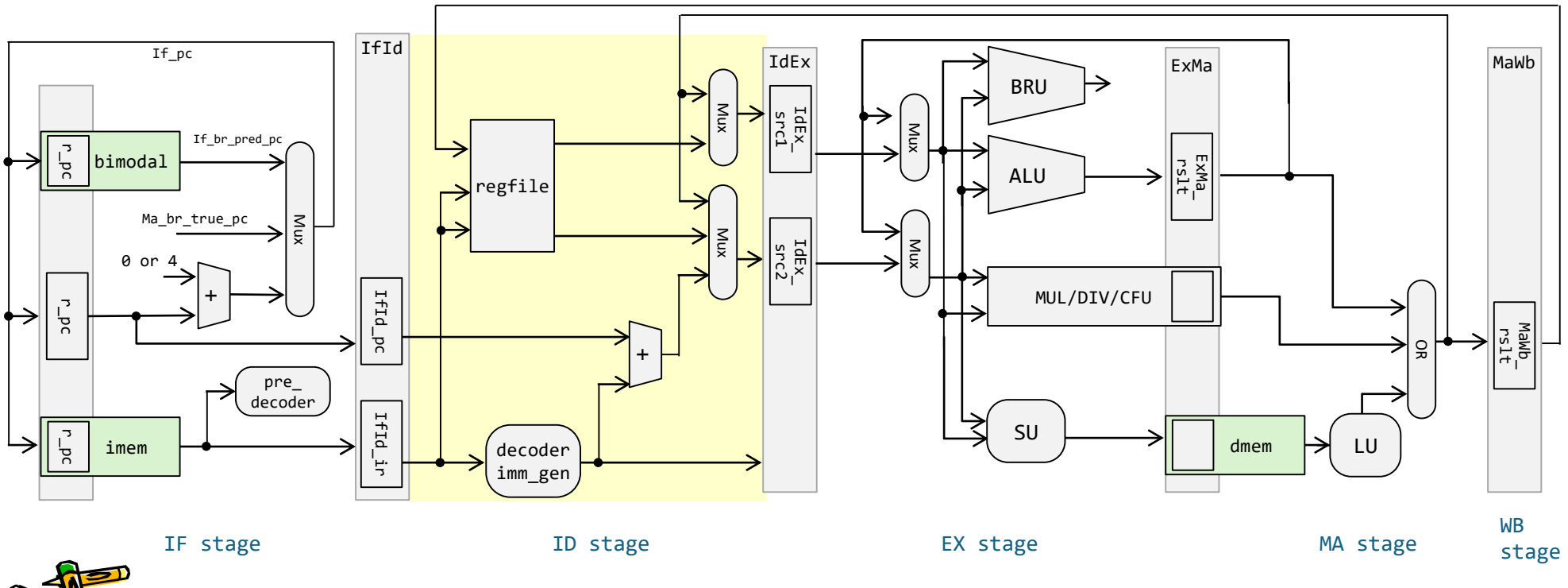



```
module pre_decoder (
    input  wire [31:0] ir_i,
    output wire [ 2:0] instr_type_o,
    output wire        rf_we_o,
    output wire [ 4:0] rd_o,
    output wire [ 4:0] rs1_o,
    output wire [ 4:0] rs2_o
);

    wire [4:0] opcode = ir_i[6:2];
    assign instr_type_o = (opcode == 5'b01101) ? `U_TYPE :  // LUI
        (opcode == 5'b00101) ? `U_TYPE :  // AUIPC
        (opcode == 5'b11011) ? `J_TYPE :  // JAL
        (opcode == 5'b11001) ? `I_TYPE :  // JALR
        (opcode == 5'b11000) ? `B_TYPE :  // BRANCH
        (opcode == 5'b00000) ? `I_TYPE :  // LOAD
        (opcode == 5'b01000) ? `S_TYPE :  // STORE
        (opcode == 5'b00100) ? `I_TYPE :  // OP-IMM
        (opcode == 5'b01100) ? `R_TYPE :  // OP
        (opcode == 5'b00010) ? `R_TYPE : `NONE_TYPE;  // CUSTOM-0 : NONE

    assign rd_o = ((instr_type_o == `S_TYPE) | (instr_type_o == `B_TYPE)) ? 0 : ir_i[11:7];
    assign rs1_o = ((instr_type_o == `U_TYPE) | (instr_type_o == `J_TYPE)) ? 0 : ir_i[19:15];
    assign rs2_o = ((instr_type_o==`I_TYPE) |
                    (instr_type_o==`U_TYPE) | (instr_type_o==`J_TYPE)) ? 0 : ir_i[24:20];
    assign rf_we_o = (rd_o != 0);
endmodule
```

opcode5[2:0] / opcode5[4:3]	000	001	010	011	100	101	110
00	LOAD (I-type)	LOAD-FP	custom-0	MISC-MEM (I-type)	OP-IMM (I-type)	AUIPC (U-type)	OP-IMM-32
01	STORE (S-type)	STORE-FP	custom-1	AMO	OP (R-type)	LUI (U-type)	OP-32
10	MADD	MSUB	NMSUB	NMADD	OP-FP	OP-V	custom-2/ rv128
11	BRANCH (B-type)	JALR (I-type)	reserved	JAL (J-type)	SYSTEM (I-type)	reserved	custom-3/ rv128

5-stage pipelining processor of CFU Proving Ground

- IF (Instruction Fetch) 命令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。
- EX (Execution) 命令操作の実行またはアドレスの生成を行う。
- MA (Memory Access) 必要であれば、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、結果をレジスタに書き込む。



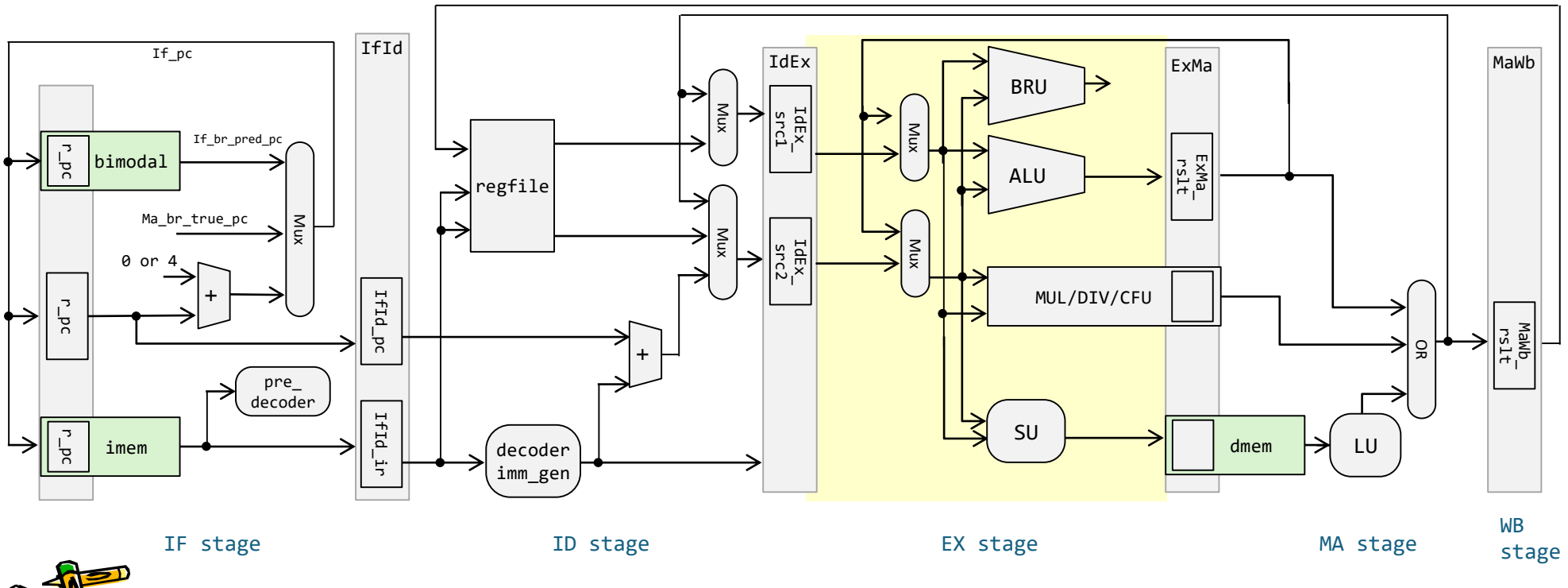
ID (Instruction Decode) stage

```
//------------------------------------------------------------------------------
// ID: Instruction Decode
//------------------------------------------------------------------------------
    // instruction decoder
    wire [`SRC2_CTRL_WIDTH-1:0] Id_src2_ctrl;
    wire [ `ALU_CTRL_WIDTH-1:0] Id_alu_ctrl;
    wire [ `BRU_CTRL_WIDTH-1:0] Id_bru_ctrl;
    wire [ `LSU_CTRL_WIDTH-1:0] Id_lsu_ctrl;
    wire [ `MUL_CTRL_WIDTH-1:0] Id_mul_ctrl;
    wire [ `DIV_CTRL_WIDTH-1:0] Id_div_ctrl;
    wire [ `CFU_CTRL_WIDTH-1:0] Id_cfu_ctrl;
    decoder decoder (
        .ir_i       (IfId_ir),       // input  wire                 [31:0]
        .src2_ctrl_o(Id_src2_ctrl),  // output wire [`SRC2_CTRL_WIDTH-1:0]
        .alu_ctrl_o (Id_alu_ctrl),   // output wire  [`ALU_CTRL_WIDTH-1:0]
        .bru_ctrl_o (Id_bru_ctrl),   // output wire  [`BRU_CTRL_WIDTH-1:0]
        .lsu_ctrl_o (Id_lsu_ctrl),   // output wire  [`LSU_CTRL_WIDTH-1:0]
        .mul_ctrl_o (Id_mul_ctrl),   // output wire  [`MUL_CTRL_WIDTH-1:0]
        .div_ctrl_o (Id_div_ctrl),   // output wire  [`DIV_CTRL_WIDTH-1:0]
        .cfu_ctrl_o (Id_cfu_ctrl)    // output wire  [`CFU_CTRL_WIDTH-1:0]
    );

    // immediate value generator
    wire [`XLEN-1:0] Id_imm;
    imm_gen imm_gen (
        .ir_i        (IfId_ir),          // input  wire         [31:0]
        .instr_type_i(IfId_instr_type),  // input  wire [`ITYPE_W-1;0]
        .imm_o       (Id_imm)            // output wire    [`XLEN-1:0]
    );

    // register file
    wire [`XLEN-1:0] Id_xrs1;
    wire [`XLEN-1:0] Id_xrs2;
    wire             Wb_xreg_we = MaWb_v && MaWb_rf_we && !ExMa_stall;
    regfile xreg (
        .clk_i  (clk_i),       // input  wire
        .rs1_i  (IfId_rs1),    // input  wire       [4:0]
        .rs2_i  (IfId_rs2),    // input  wire       [4:0]
        .xrs1_o (Id_xrs1),     // output wire [`XLEN-1:0]
        .xrs2_o (Id_xrs2),     // output wire [`XLEN-1:0]
        .we_i   (Wb_xreg_we && !w_stall),  // input  wire
        .rd_i   (MaWb_rd),     // input  wire       [4:0]
        .wdata_i(MaWb_rslt)    // input  wire [`XLEN-1:0]
    );
```

```
    // data forwarding
    wire Id_rs1_fwd_Ma_to_Ex = IdEx_v && IdEx_rf_we && (IdEx_rd == IfId_rs1);
    wire Id_rs2_fwd_Ma_to_Ex = IdEx_v && IdEx_rf_we && (IdEx_rd == IfId_rs2);
    wire Id_rs1_fwd_Wb_to_Ex = ExMa_v && ExMa_rf_we && (ExMa_rd == IfId_rs1);
    wire Id_rs2_fwd_Wb_to_Ex = ExMa_v && ExMa_rf_we && (ExMa_rd == IfId_rs2);

    wire [31:0] Id_pc_in = (Id_src2_ctrl[`SRC2_CTRL_USE_AUIPC]) ? IfId_pc : 0;
    wire Id_use_imm = Id_src2_ctrl[`SRC2_CTRL_USE_AUIPC] |
      Id_src2_ctrl[`SRC2_CTRL_USE_IMM];

    // source select
    wire [`XLEN-1:0] Id_src1 = (Id_rs1_fwd_Wb_to_Ex) ? Ma_rslt : Id_xrs1;
    wire [`XLEN-1:0] Id_src2 = (Id_rs2_fwd_Wb_to_Ex) ? Ma_rslt :
                               (Id_use_imm) ? Id_pc_in+Id_imm  : Id_xrs2 ;

    wire [31:0] Id_j_pc4 = (Id_bru_ctrl[`BRU_CTRL_IS_JAL_JALR]) ? IfId_pc + 4 : 0;

    always @(posedge clk_i) if (!w_stall) begin
        if (rst) begin
            IdEx_v  <= 0;
            IdEx_pc <= 0;
            IdEx_ir <= `NOP;
        end else if (!ExMa_stall) begin
            IdEx_v                <= Id_v;
            IdEx_pc               <= IfId_pc;
            IdEx_j_pc4            <= Id_j_pc4;
            IdEx_ir               <= IfId_ir;
            IdEx_br_pred_tkn      <= IfId_br_pred_tkn;
            IdEx_pat_hist         <= IfId_pat_hist;
            IdEx_alu_ctrl         <= Id_alu_ctrl;
            IdEx_bru_ctrl         <= Id_bru_ctrl;
            IdEx_lsu_ctrl         <= Id_lsu_ctrl;
            IdEx_mul_ctrl         <= Id_mul_ctrl;
            IdEx_div_ctrl         <= Id_div_ctrl;
            IdEx_rs1_fwd_Ma_to_Ex <= Id_rs1_fwd_Ma_to_Ex;
            IdEx_rs2_fwd_Ma_to_Ex <= Id_rs2_fwd_Ma_to_Ex;
            IdEx_src1             <= Id_src1;
            IdEx_src2             <= Id_src2;
            IdEx_imm              <= Id_imm;
            IdEx_rf_we            <= IfId_rf_we;
            IdEx_rd               <= IfId_rd;
            IdEx_cfu_ctrl         <= Id_cfu_ctrl;  // Note
        end
    end
```

5-stage pipelining processor of CFU Proving Ground

- IF (Instruction Fetch) 命令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。
- EX (Execution) 命令操作の実行またはアドレスの生成を行う。
- MA (Memory Access) 必要であれば、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、結果をレジスタに書き込む。



EX (Execution) stage

```
//------------------------------------------------------------------------------
// EX: Execution
//------------------------------------------------------------------------------
    wire Ex_valid = IdEx_v && !Ma_br_misp && !ExMa_stall;

    ///// data forwarding
    wire [`XLEN-1:0] Ex_src1 = (IdEx_rs1_fwd_Ma_to_Ex) ? ExMa_rslt : IdEx_src1;
    wire [`XLEN-1:0] Ex_src2 = (IdEx_rs2_fwd_Ma_to_Ex) ? ExMa_rslt : IdEx_src2;

    ///// arithmetic logic unit
    wire [`XLEN-1:0] Ex_alu_rslt;
    alu alu (
        .alu_ctrl_i(IdEx_alu_ctrl),  // input  wire [`ALU_CTRL_WIDTH-1:0]
        .src1_i    (Ex_src1),        // input  wire           [`XLEN-1:0]
        .src2_i    (Ex_src2),        // input  wire           [`XLEN-1:0]
        .j_pc4_i   (IdEx_j_pc4),     // input  wire           [`XLEN-1:0]
        .rslt_o    (Ex_alu_rslt)     // output wire           [`XLEN-1:0]
    );

    ///// branch resolution unit
    wire             Ex_is_ctrl_tsfr;
    wire             Ex_br_tkn;
    wire             Ex_br_misp_rslt1;
    wire             Ex_br_misp_rslt2;
    wire [`XLEN-1:0] Ex_br_tkn_pc;
    bru bru (
        .bru_ctrl_i     (IdEx_bru_ctrl),     // input  wire [`BRU_CTRL_WIDTH-1:0]
        .src1_i         (Ex_src1),           // input  wire           [`XLEN-1:0]
        .src2_i         (Ex_src2),           // input  wire           [`XLEN-1:0]
        .pc_i           (IdEx_pc),           // input  wire           [`XLEN-1:0]
        .imm_i          (IdEx_imm),          // input  wire           [`XLEN-1:0]
        .npc_i          (IfId_pc),           // input  wire           [`XLEN-1:0]
        .br_pred_tkn_i  (IdEx_br_pred_tkn),  // input  wire
        .is_ctrl_tsfr_o (Ex_is_ctrl_tsfr),   // output wire
        .br_tkn_o       (Ex_br_tkn),         // output wire
        .br_misp_rslt1_o(Ex_br_misp_rslt1),  // output wire
        .br_misp_rslt2_o(Ex_br_misp_rslt2),  // output wire
        .br_tkn_pc_o    (Ex_br_tkn_pc)       // output wire           [`XLEN-1:0]
    );

    ///// store unit
    wire [         `XLEN-1:0] dbus_addr = dbus_addr_o;  // for simulation
    wire [         `XLEN-1:0] dbus_wdata = dbus_wdata_o;  // for simulation
    wire [`DBUS_OFFSET_W-1:0] dbus_offset;  // Note
    store_unit store_unit (
        .valid_i      (Ex_valid && !w_stall), // input  wire
        .lsu_ctrl_i   (IdEx_lsu_ctrl),  // input  wire [`LSU_CTRL_WIDTH-1:0]
        .src1_i       (Ex_src1),        // input  wire           [`XLEN-1:0]
        .src2_i       (Ex_src2),        // input  wire           [`XLEN-1:0]
        .imm_i        (IdEx_imm),       // input  wire           [`XLEN-1:0]
        .dbus_addr_o  (dbus_addr_o),    // output wire           [`XLEN-1:0]
        .dbus_offset_o(dbus_offset),    // output wire    [OFFSET_WIDTH-1:0]
        .dbus_wvalid_o(dbus_wvalid_o),  // output wire
        .dbus_wdata_o (dbus_wdata_o),   // output wire           [`XLEN-1:0]
        .dbus_wstrb_o (dbus_wstrb_o)    // output wire         [`XBYTES-1:0]
    );
```

```
    ///// multiplier unit
    wire             Ex_mul_stall;
    wire [`XLEN-1:0] Ex_mul_rslt;
    multiplier multiplier (
        .clk_i     (clk_i),          // input  wire
        .rst_i     (rst),            // input  wire // Note
        .stall_i   (w_stall),        // input  wire
        .valid_i   (Ex_valid),       // input  wire
        .mul_ctrl_i(IdEx_mul_ctrl),  // input  wire [`MUL_CTRL_WIDTH-1:0]
        .src1_i    (Ex_src1),        // input  wire           [`XLEN-1:0]
        .src2_i    (Ex_src2),        // input  wire           [`XLEN-1:0]
        .stall_o   (Ex_mul_stall),   // output wire
        .rslt_o    (Ex_mul_rslt)     // output wire           [`XLEN-1:0]
    );

    ///// divider unit
    wire             Ex_div_stall;
    wire [`XLEN-1:0] Ex_div_rslt;
    divider divider (
        .clk_i     (clk_i),          // input  wire
        .rst_i     (rst),            // input  wire
        .stall_i   (w_stall),        // input  wire // Note
        .valid_i   (Ex_valid),       // input  wire
        .div_ctrl_i(IdEx_div_ctrl),  // input  wire [`DIV_CTRL_WIDTH-1:0]
        .src1_i    (Ex_src1),        // input  wire           [`XLEN-1:0]
        .src2_i    (Ex_src2),        // input  wire           [`XLEN-1:0]
        .stall_o   (Ex_div_stall),   // output wire
        .rslt_o    (Ex_div_rslt)     // output wire           [`XLEN-1:0]
    );



    always @(posedge clk_i) if (!w_stall) begin
        ExMa_mul_stall <= Ex_mul_stall;
        ExMa_div_stall <= Ex_div_stall;
        ExMa_stall     <= Ex_mul_stall | Ex_div_stall | Ex_cfu_stall;
        ExMa_mdc_rslt  <= Ex_mul_rslt | Ex_div_rslt | Ex_cfu_rslt;
        if (rst) begin
            ExMa_v  <= 0;
            ExMa_pc <= 0;
            ExMa_ir <= `NOP;
        end else if (!ExMa_stall) begin
            ExMa_v             <= Ex_v;
            ExMa_pc            <= IdEx_pc;
            ExMa_ir            <= IdEx_ir;
            ExMa_pat_hist      <= IdEx_pat_hist;
            ExMa_is_ctrl_tsfr  <= Ex_is_ctrl_tsfr;
            ExMa_br_tkn        <= Ex_br_tkn;
            ExMa_br_misp_rslt1 <= Ex_br_misp_rslt1;
            ExMa_br_misp_rslt2 <= Ex_br_misp_rslt2;
            ExMa_br_tkn_pc     <= Ex_br_tkn_pc;
            ExMa_lsu_ctrl      <= IdEx_lsu_ctrl;
            ExMa_dbus_offset   <= dbus_offset;
            ExMa_rf_we         <= IdEx_rf_we;
            ExMa_rd            <= IdEx_rd;
            ExMa_rslt          <= Ex_alu_rslt;
            ExMa_j_b_insn      <= IdEx_bru_ctrl[0] & Ex_v;
        end
    end
```

EX (Execution) stage

```
module alu (
    input  wire [`ALU_CTRL_WIDTH-1:0] alu_ctrl_i,
    input  wire               [31:0] src1_i    ,
    input  wire               [31:0] src2_i    ,
    input  wire               [31:0] j_pc4_i   ,
    output wire               [31:0] rslt_o
);

    wire w_signed = alu_ctrl_i[`ALU_CTRL_IS_SIGNED];
    wire w_neg    = alu_ctrl_i[`ALU_CTRL_IS_NEG];
    wire w_less   = alu_ctrl_i[`ALU_CTRL_IS_LESS];

    wire [33:0] adder_src1   = {w_signed && src1_i[31], src1_i, 1'b1};
    wire [33:0] adder_src2   = {w_signed && src2_i[31], src2_i, 1'b0} ^ {34{w_neg}};
    wire [33:0] adder_rslt_t = adder_src1+adder_src2;
    wire        less_rslt    = w_less && adder_rslt_t[33];
    wire [31:0] adder_rslt   = (alu_ctrl_i[`ALU_CTRL_IS_ADD]) ? adder_rslt_t[32:1] : 0;

    wire signed  [32:0] right_shifter_src1 = {w_signed && src1_i[31], src1_i};
    wire  [4:0] shamt              = src2_i[4:0];
    wire [31:0] left_shifter_rslt  = (alu_ctrl_i[`ALU_CTRL_IS_SHIFT_LEFT] ) ?
                                     src1_i <<  shamt : 0;
    wire [31:0] right_shifter_rslt = (alu_ctrl_i[`ALU_CTRL_IS_SHIFT_RIGHT]) ?
                                     right_shifter_src1 >>> shamt : 0;

    wire [31:0] bitwise_rslt       = ((alu_ctrl_i[`ALU_CTRL_IS_XOR_OR]) ?
                                     (src1_i ^ src2_i) : 0) |
                                     ((alu_ctrl_i[`ALU_CTRL_IS_OR_AND])
                                      ? (src1_i & src2_i) : 0);
    wire [31:0] lui_auipc_rslt     = (alu_ctrl_i[`ALU_CTRL_IS_SRC2]) ? src2_i : 0;

    assign rslt_o = less_rslt | adder_rslt | left_shifter_rslt | right_shifter_rslt |
                    bitwise_rslt | lui_auipc_rslt | j_pc4_i;
endmodule
```

```
module store_unit (
    input  wire        valid_i,
    input  wire [ 5:0] lsu_ctrl_i,
    input  wire [31:0] src1_i,
    input  wire [31:0] src2_i,
    input  wire [31:0] imm_i,
    output wire [31:0] dbus_addr_o,
    output wire [ 1:0] dbus_offset_o,
    output wire        dbus_wvalid_o,
    output wire [31:0] dbus_wdata_o,
    output wire [ 3:0] dbus_wstrb_o
);

    assign dbus_addr_o   = (valid_i && (lsu_ctrl_i[`LSU_CTRL_IS_STORE] ||
                                        lsu_ctrl_i[`LSU_CTRL_IS_LOAD])) ? src1_i + imm_i : 0;
    assign dbus_offset_o = dbus_addr_o[1:0];
    assign dbus_wvalid_o = valid_i && lsu_ctrl_i[`LSU_CTRL_IS_STORE];

    wire w_sb = lsu_ctrl_i[`LSU_CTRL_IS_BYTE];
    wire w_sh = lsu_ctrl_i[`LSU_CTRL_IS_HALFWORD];
    wire w_sw = lsu_ctrl_i[`LSU_CTRL_IS_WORD];

    assign dbus_wdata_o[7:0]   = src2_i[7:0];
    assign dbus_wdata_o[15:8]  = (w_sb) ? src2_i[7:0] : src2_i[15:8];
    assign dbus_wdata_o[23:16] = (w_sw) ? src2_i[23:16] : src2_i[7:0];
    assign dbus_wdata_o[31:24] = (w_sb) ? src2_i[7:0] : (w_sh) ? src2_i[15:8] : src2_i[31:24];

    assign dbus_wstrb_o[0] = (w_sb && dbus_offset_o==0) || (w_sh && dbus_offset_o[1]==0) || w_sw;
    assign dbus_wstrb_o[1] = (w_sb && dbus_offset_o==1) || (w_sh && dbus_offset_o[1]==0) || w_sw;
    assign dbus_wstrb_o[2] = (w_sb && dbus_offset_o==2) || (w_sh && dbus_offset_o[1]==1) || w_sw;
    assign dbus_wstrb_o[3] = (w_sb && dbus_offset_o==3) || (w_sh && dbus_offset_o[1]==1) || w_sw;
endmodule
```

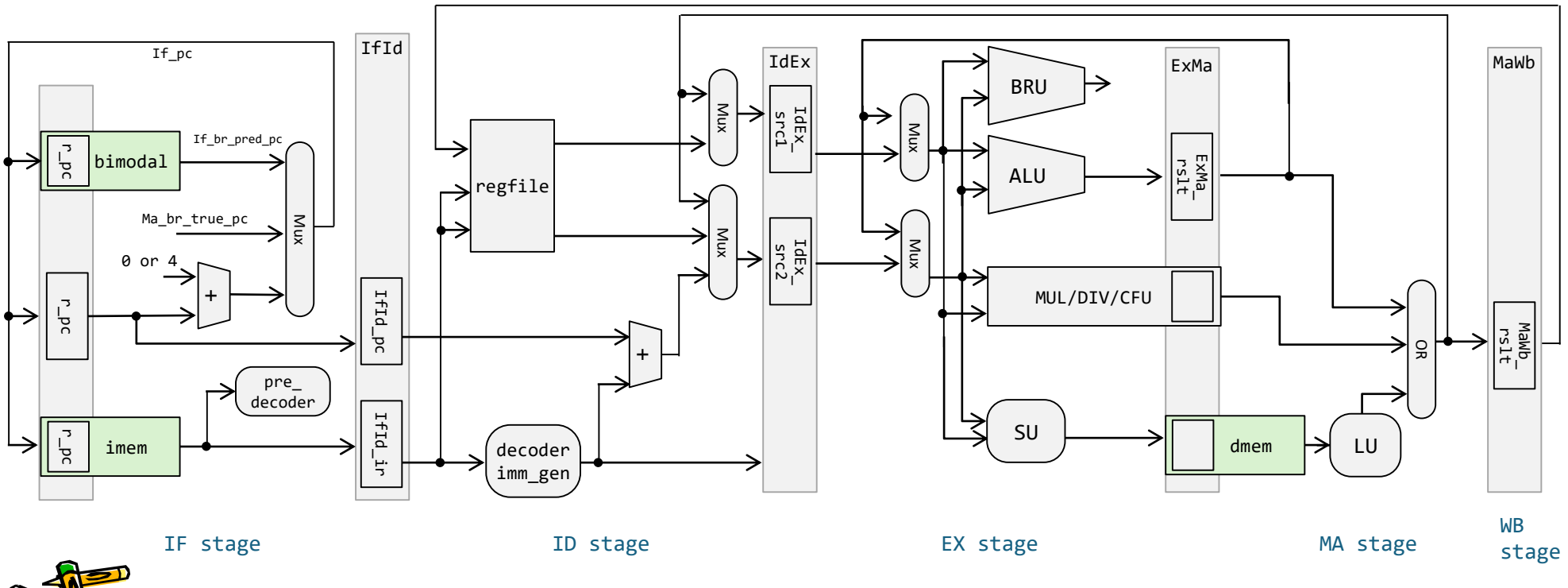
EX (Execution) stage

```
module bru (
    input  wire [`BRU_CTRL_WIDTH-1:0] bru_ctrl_i,
    input  wire [               31:0] src1_i,
    input  wire [               31:0] src2_i,
    input  wire [               31:0] pc_i,
    input  wire [               31:0] imm_i,
    input  wire [               31:0] npc_i,
    input  wire                       br_pred_tkn_i,
    output wire                       is_ctrl_tsfr_o,
    output wire                       br_tkn_o,
    output wire                       br_misp_rslt1_o,
    output wire                       br_misp_rslt2_o,
    output wire [               31:0] br_tkn_pc_o
);

    wire signed [32:0] sext_src1 = {bru_ctrl_i[`BRU_CTRL_IS_SIGNED] && src1_i[31], src1_i};
    wire signed [32:0] sext_src2 = {bru_ctrl_i[`BRU_CTRL_IS_SIGNED] && src2_i[31], src2_i};

    wire               w_eq = (src1_i == src2_i);  // equal
    wire               w_lt = (sext_src1 < sext_src2);  // less than

    assign br_tkn_o = bru_ctrl_i[`BRU_CTRL_IS_JAL_JALR] |
                     (bru_ctrl_i[`BRU_CTRL_IS_BEQ] &  w_eq) |
                     (bru_ctrl_i[`BRU_CTRL_IS_BNE] & !w_eq) |
                     (bru_ctrl_i[`BRU_CTRL_IS_BLT] &  w_lt) |
                     (bru_ctrl_i[`BRU_CTRL_IS_BGE] & !w_lt);

    wire [31:0] br_tkn_pc_t;
    assign br_tkn_pc_t     = ((bru_ctrl_i[`BRU_CTRL_IS_JALR]) ? src1_i : pc_i) + imm_i;
    assign br_tkn_pc_o     = {br_tkn_pc_t[31:1], 1'b0};

    assign is_ctrl_tsfr_o  = (bru_ctrl_i[`BRU_CTRL_IS_CTRL_TSFR] || br_pred_tkn_i);

    assign br_misp_rslt1_o = (npc_i != br_tkn_pc_o);
    assign br_misp_rslt2_o = (npc_i != (pc_i + 'h4));
endmodule
```

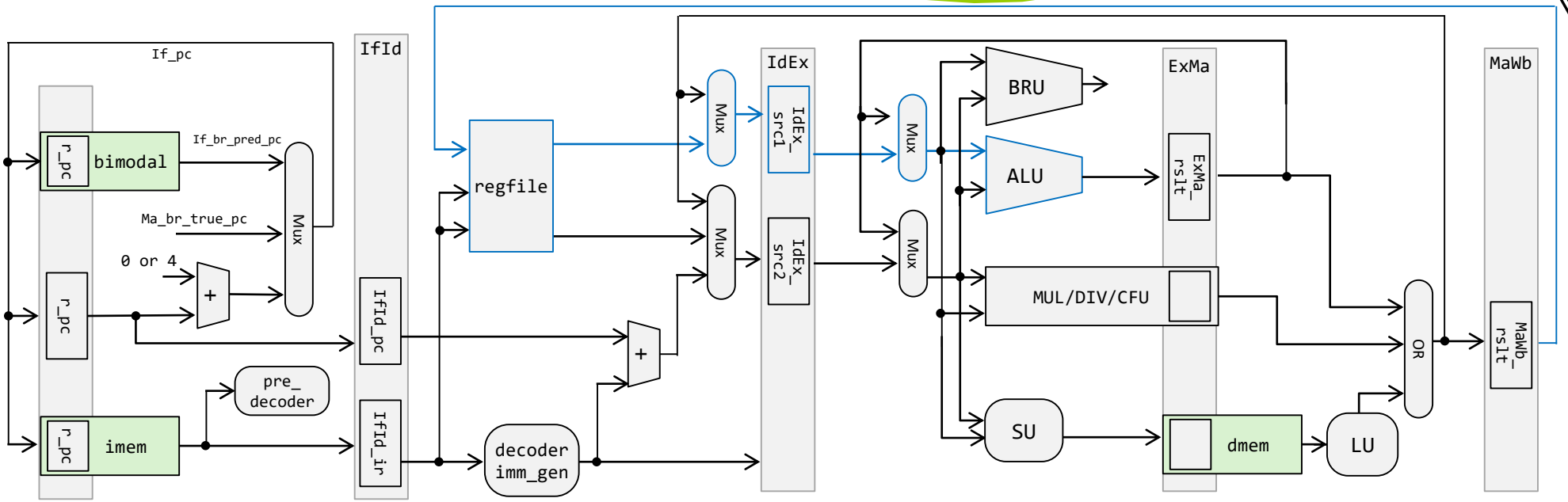
```
module multiplier (
    input  wire        clk_i,
    input  wire        rst_i,
    input  wire        stall_i,
    input  wire        valid_i,
    input  wire [ 3:0] mul_ctrl_i,
    input  wire [31:0] src1_i,
    input  wire [31:0] src2_i,
    output wire        stall_o,
    output wire [31:0] rslt_o
);

    reg        [ 1:0] state = `MUL_IDLE;
    reg signed [32:0] r_multiplicand;  // 33bit
    reg signed [32:0] r_multiplier;  // 33bit
    reg        [63:0] product;  // 64bit
    reg               is_high;  //

    assign rslt_o = (state != `MUL_RET) ? 0 : (is_high) ? product[63:32] : product[31:0];

    wire w_mul = mul_ctrl_i[`MUL_CTRL_IS_MUL];
    wire w_src1_signed = mul_ctrl_i[`MUL_CTRL_IS_SRC1_SIGNED];
    wire w_src2_signed = mul_ctrl_i[`MUL_CTRL_IS_SRC2_SIGNED];
    wire w_is_high = mul_ctrl_i[`MUL_CTRL_IS_HIGH];
    wire [1:0] w_state = (state==`MUL_IDLE && valid_i && w_mul) ? `MUL_EXEC :
                         (state==`MUL_EXEC) ? `MUL_RET : `MUL_IDLE;

    always @(posedge clk_i) if (!stall_i) begin
        if (rst_i) begin
            state <= `MUL_IDLE;
        end else if (!stall_i) begin
            if (state == `MUL_IDLE) r_multiplicand <= {w_src1_signed && src1_i[31], src1_i};
            if (state == `MUL_IDLE) r_multiplier <= {w_src2_signed && src2_i[31], src2_i};
            if (state == `MUL_IDLE) is_high <= w_is_high;
            if (state == `MUL_EXEC) product <= r_multiplicand * r_multiplier;
            state <= w_state;
        end
    end
    assign stall_o = (w_state != `MUL_IDLE);
endmodule
```

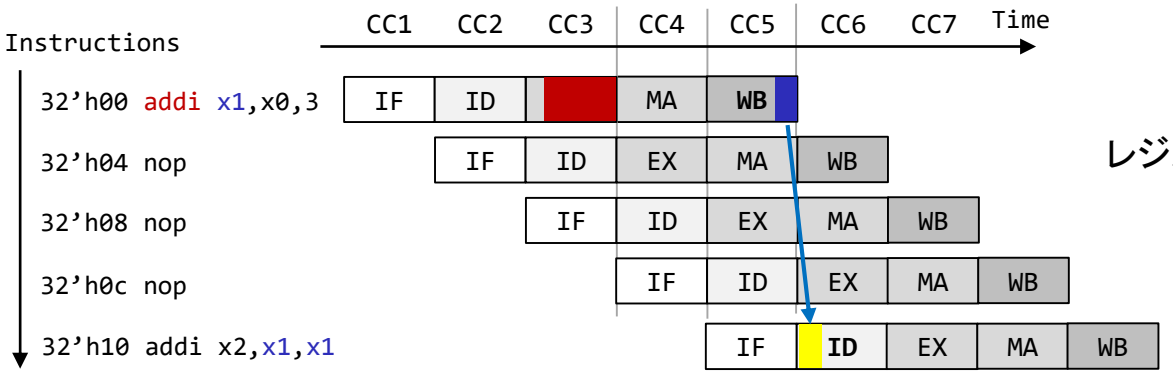
5-stage pipelining processor of CFU Proving Ground

- IF (Instruction Fetch) 命令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。
- EX (Execution) 命令操作の実行またはアドレスの生成を行う。
- MA (Memory Access) 必要であれば、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、結果をレジスタに書き込む。



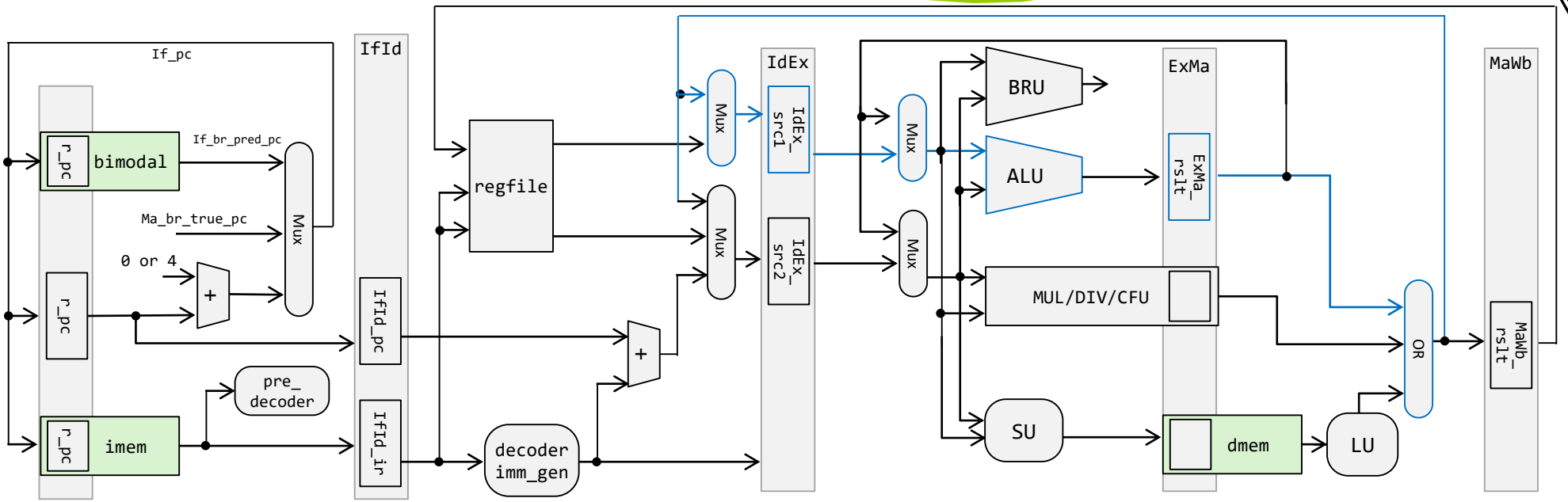
パイプラインプロセッサでのデータ供給 (1)

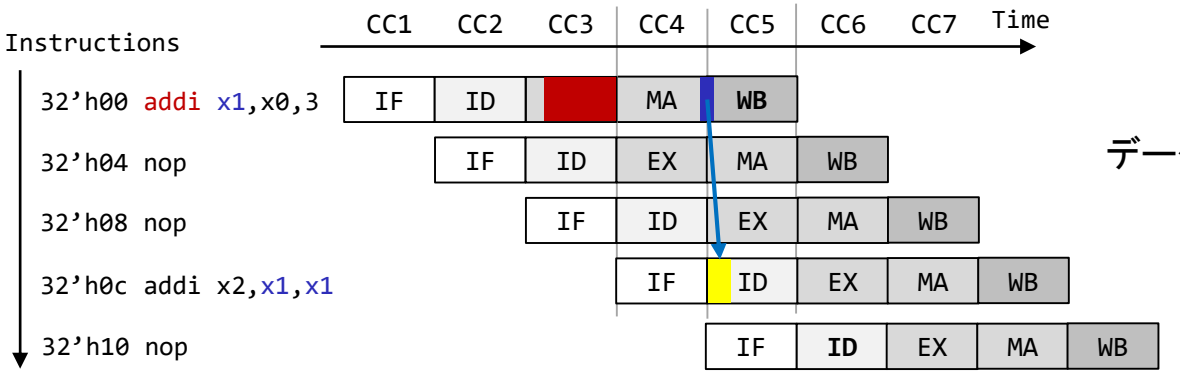




レジスタファイルから

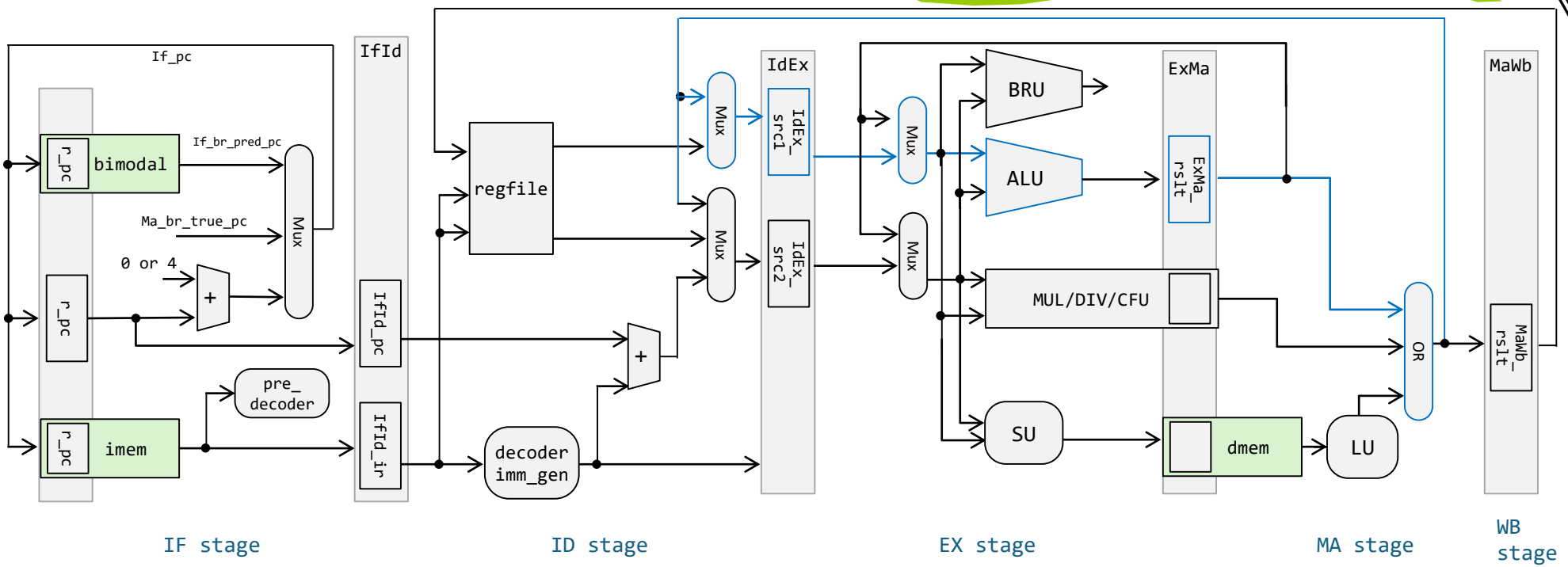
パイプラインプロセッサでのデータ供給 (2)

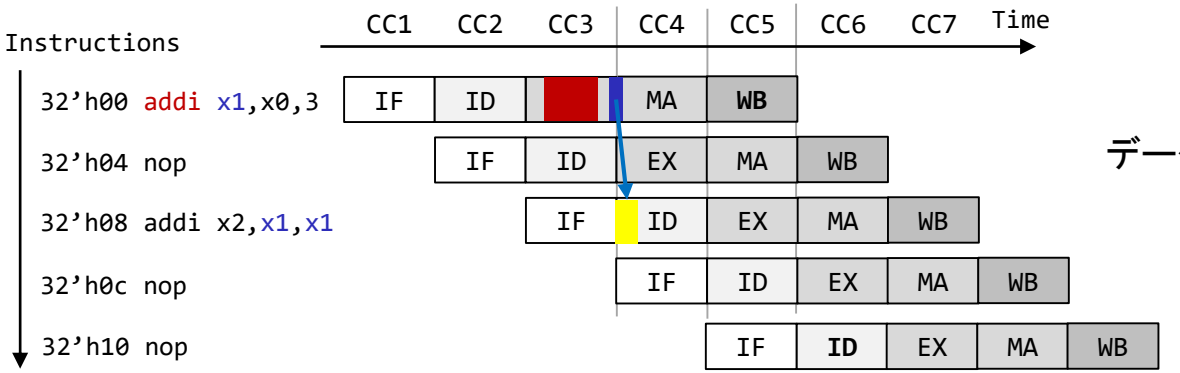




データフォワーディング

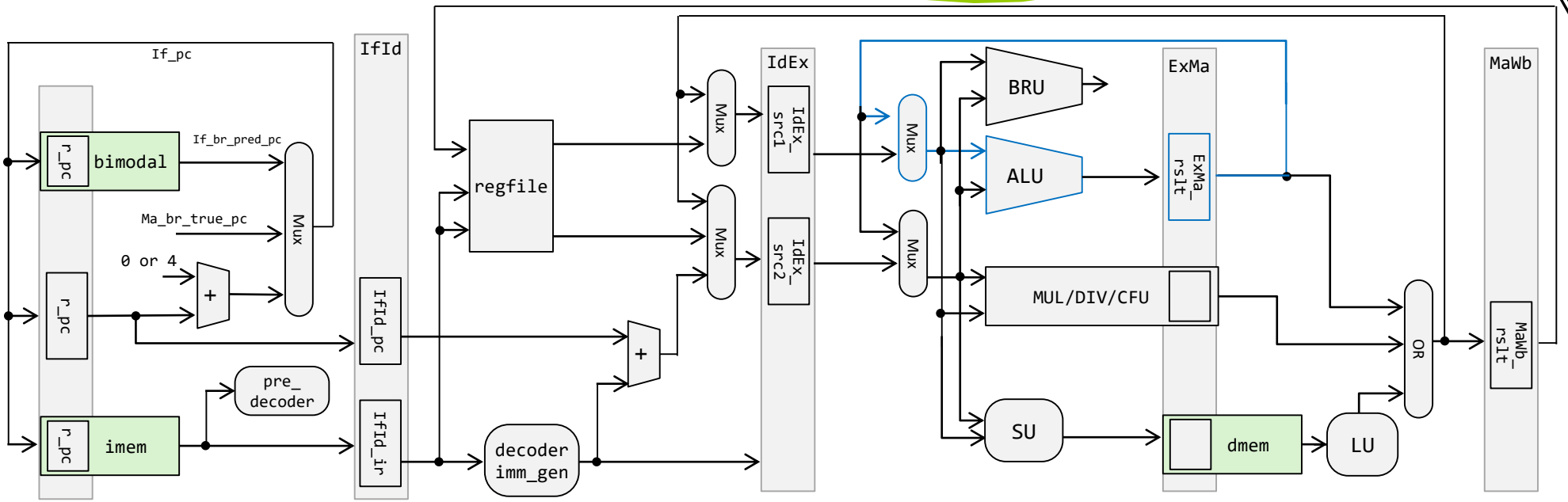
パイプラインプロセッサでのデータ供給 (3)

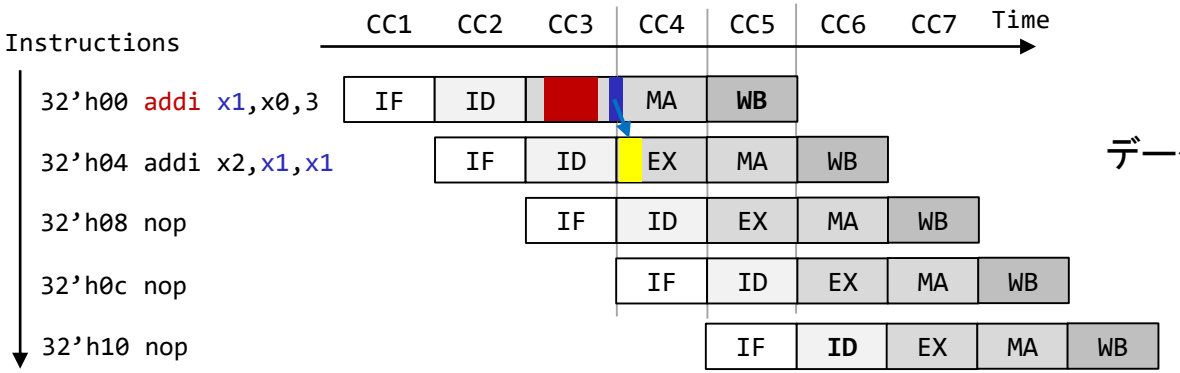




データフォワーディング

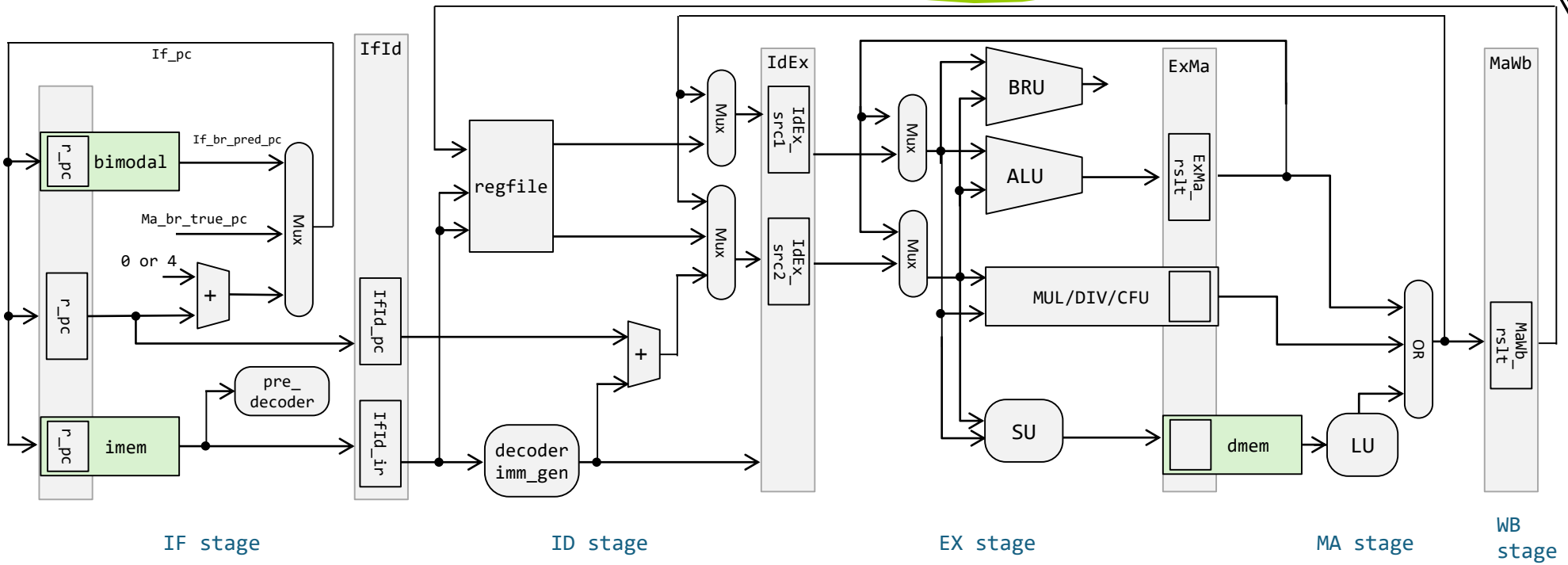
パイプラインプロセッサでのデータ供給 (4)

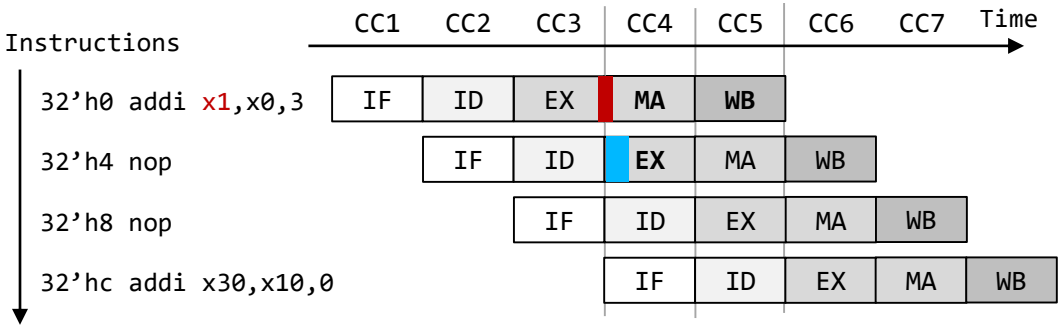




データフォワーディング

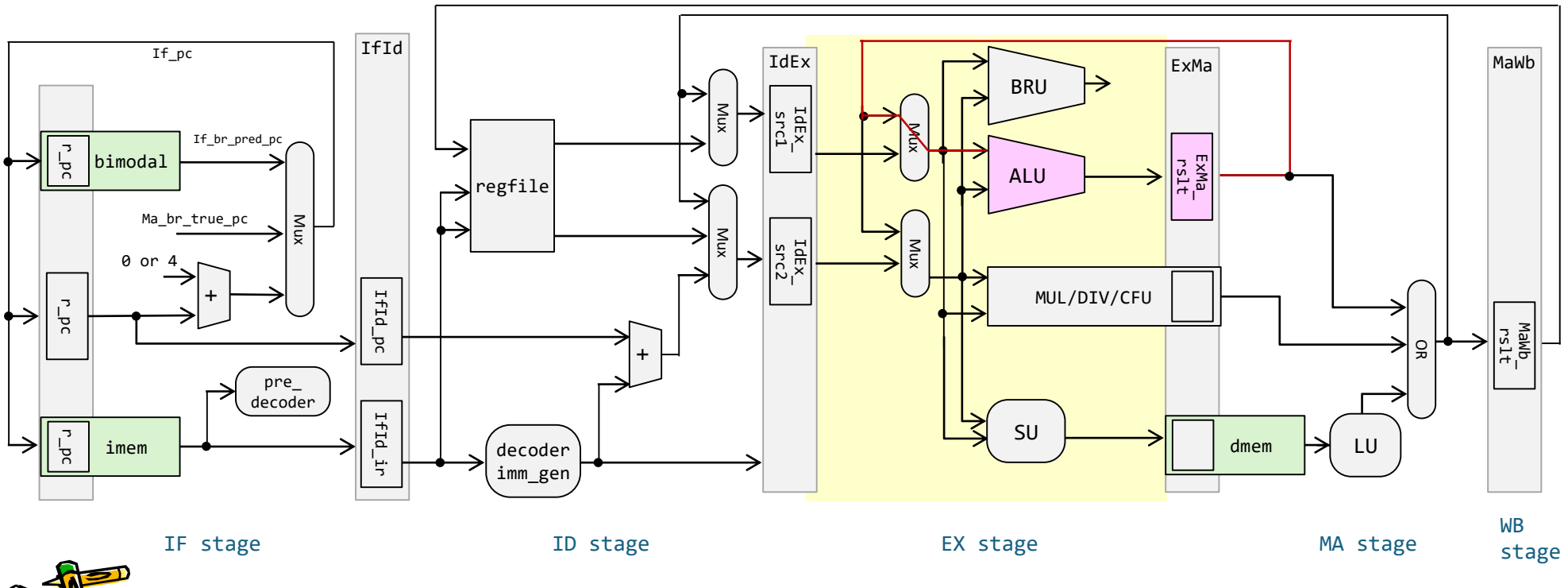
5-stage pipelining processor of CFU Proving Ground





データフォワーディング

- IF (Instruction Fetch) 命令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。
- EX (Execution) 命令操作の実行またはアドレスの生成を行う。
- MA (Memory Access) 必要であれば、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、結果をレジスタに書き込む。



データフォワーディング

- app/crt0.s を下図のコードに編集する。
- 次のコマンドで実行
  - make
  - make run

```
    .equ TO_HOST,    0x80000000
    .equ CMD_FINISH, 0x00020000
    .align 4
    .section .text.init
    .globl _start
_start:
    li      x1, 1
    li      x2, 2
    add     x3, x2, x2
    li      t0, TO_HOST
    li      t1, CMD_FINISH
    sw      t1, 0(t0)
L1: j       L1
```

```
./obj_dir/top
 000000: 00000000 -------- -------- -------- --------:
 000000: 00000004 00000000 -------- -------- --------:
 000000: 00000008 00000004 00000000 -------- --------:
 000000: 0000000c 00000008 00000004 00000000 --------:
 000001: 00000010 0000000c 00000008 00000004 00000000:
 000002: 00000014 00000010 0000000c 00000008 00000004:
 000003: 00000018 00000014 00000010 0000000c 00000008:
 000004: 0000001c 00000018 00000014 00000010 0000000c:
 000005: 00000020 0000001c 00000018 00000014 00000010:
- top.v:47: Verilog $finish

===> mcycle                                :         9
===> minstret                              :         5
===> Total number of branch predictions    :         0
===> Total number of branch mispredictions :         0
===> simulation finish!!
```

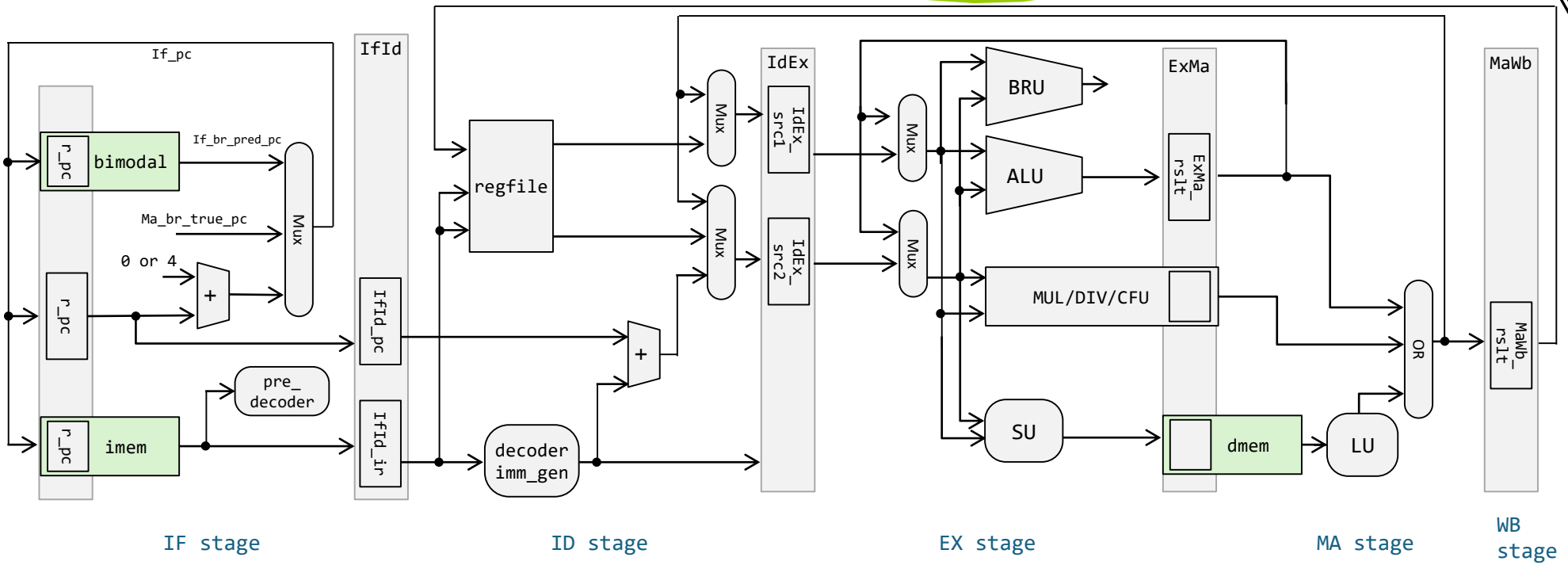
データフォワーディング

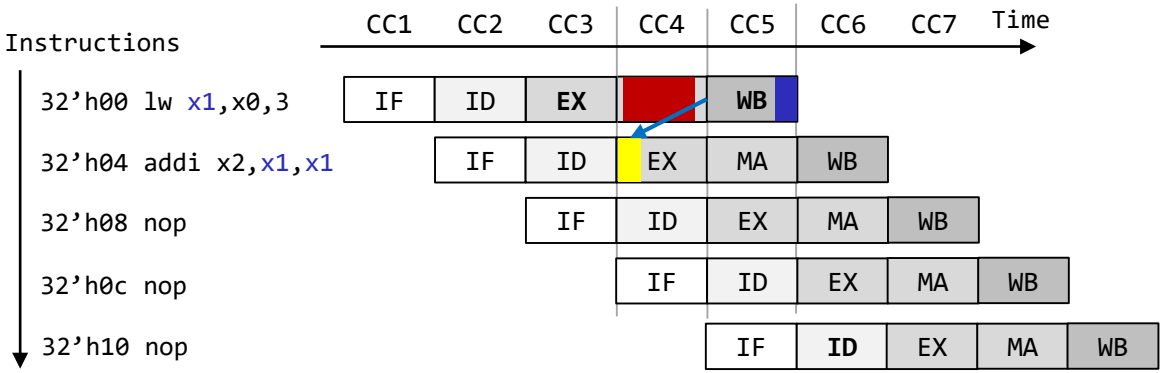
- ロード命令がロードしたデータを直後の命令が利用する場合には、データフォワーディングでオペランドを供給できない。そのため、直後の命令のデコードステージから実行ステージへの移動を1サイクル遅らせる（バブルをひとつ挿入する）。
- app/crt0.s を下図のコードに編集する。
- 次のコマンドで実行
  - make
  - make run

```
    .equ TO_HOST,    0x80000000
    .equ CMD_FINISH, 0x00020000
    .equ DMEM_ADDR,  0x10000000
    .align 4
    .section .text.init
    .globl _start
_start:
    li      x1, 1
    li      x2, DMEM_ADDR
    lw      x2, 0(x2)
    add     x3, x2, x2
    li      t0, TO_HOST
    li      t1, CMD_FINISH
    sw      t1, 0(t0)
L1: j       L1
```

```
./obj_dir/top
 000000: 00000000 -------- -------- -------- --------:
 000000: 00000004 00000000 -------- -------- --------:
 000000: 00000008 00000004 00000000 -------- --------:
 000000: 0000000c 00000008 00000004 00000000 --------:
 000001: 00000010 0000000c 00000008 00000004 00000000:
 000002: 00000010 0000000c -------- 00000008 00000004:
 000003: 00000014 00000010 0000000c -------- 00000008:
 000003: 00000018 00000014 00000010 0000000c --------:
 000004: 0000001c 00000018 00000014 00000010 0000000c:
 000005: 00000020 0000001c 00000018 00000014 00000010:
 000006: 00000024 00000020 0000001c 00000018 00000014:
- top.v:47: Verilog $finish

===> mcycle                                :         11
===> minstret                              :          6
===> Total number of branch predictions    :          0
===> Total number of branch mispredictions :          0
===> simulation finish!!
```

パイプラインプロセッサでのデータ供給





マルチサイクルを必要とする命令の処理

- 除算の実行には35サイクルを要する。そのため、34サイクルでストールさせる。
- app/crt0.s を下図のコードに編集する。
- 次のコマンドで実行
  - make
  - make run

```
    .equ TO_HOST,    0x80000000
    .equ CMD_FINISH, 0x00020000
    .align 4
    .section .text.init
    .globl _start
_start:
    li      x1, 1
    li      x2, 2
    div     x3, x2, x2
    li      t0, TO_HOST
    li      t1, CMD_FINISH
    sw      t1, 0(t0)
L1: j       L1
```

```
./obj_dir/top
 000000: 00000000 -------- -------- -------- --------:
 000000: 00000004 00000000 -------- -------- --------:
 000000: 00000008 00000004 00000000 -------- --------:
 000000: 0000000c 00000008 00000004 00000000 --------:
 000001: 00000010 0000000c 00000008 00000004 00000000:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 00000010 0000000c 00000008 00000004:
 000003: 00000018 00000014 00000010 0000000c 00000008:
 000004: 0000001c 00000018 00000014 00000010 0000000c:
 000005: 00000020 0000001c 00000018 00000014 00000010:
- top.v:47: Verilog $finish

===> mcycle                                :          43
===> minstret                              :           5
===> Total number of branch predictions    :           0
===> Total number of branch mispredictions :           0
===> simulation finish!!
```

マルチサイクルを必要とする命令の処理

- 乗算の実行には3サイクルを要する。そのため、2サイクルでストールさせる。
- app/crt0.s を下図のコードに編集する。
- 次のコマンドで実行
  - make
  - make run

```
    .equ TO_HOST,    0x80000000
    .equ CMD_FINISH, 0x00020000
    .align 4
    .section .text.init
    .globl _start
_start:
    li      x1, 1
    li      x2, 2
    mul     x3, x2, x2
    li      t0, TO_HOST
    li      t1, CMD_FINISH
    sw      t1, 0(t0)
L1: j       L1
```

```
./obj_dir/top
 000000: 00000000 -------- -------- -------- --------:
 000000: 00000004 00000000 -------- -------- --------:
 000000: 00000008 00000004 00000000 -------- --------:
 000000: 0000000c 00000008 00000004 00000000 --------:
 000001: 00000010 0000000c 00000008 00000004 00000000:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
 000002: 00000014 00000010 0000000c 00000008 00000004:
 000003: 00000018 00000014 00000010 0000000c 00000008:
 000004: 0000001c 00000018 00000014 00000010 0000000c:
 000005: 00000020 0000001c 00000018 00000014 00000010:
- top.v:47: Verilog $finish

===> mcycle                                :         11
===> minstret                              :          5
===> Total number of branch predictions    :          0
===> Total number of branch mispredictions :          0
===> simulation finish!!
```

マルチサイクルを必要とする命令の処理

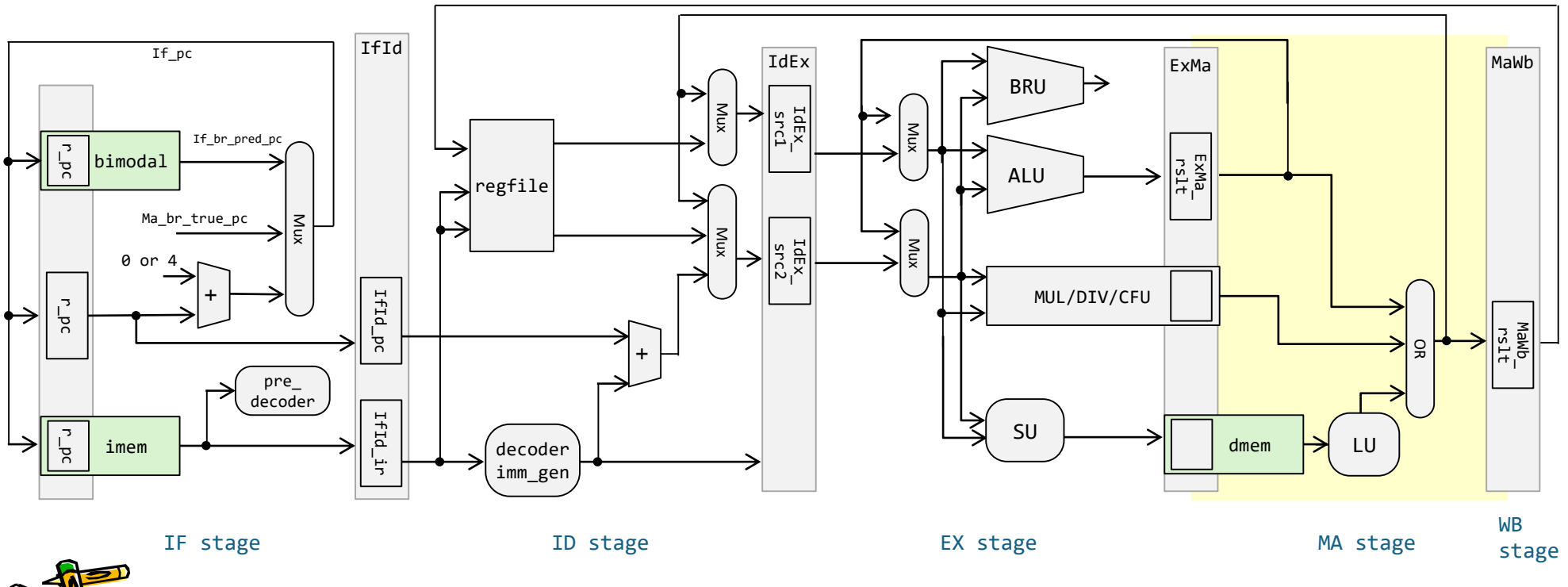
- 乗算の実行には3サイクルを要する。そのため、2サイクルでストールさせる。
- ロード命令と同様に、直後の命令が除算の結果を利用する場合には、データフォワーディングでオペランドを供給できない。そのため、バブルをひとつ挿入する。
- app/crt0.s を下図のコードに編集する。
- 次のコマンドで実行
  - make
  - make run

```
    .equ TO_HOST,    0x80000000
    .equ CMD_FINISH, 0x00020000
    .align 4
    .section .text.init
    .globl _start
_start:
    li      x1, 1
    li      x2, 2
    mul     x3, x2, x2
    add     x4, x3, x3
    li      t0, TO_HOST
    li      t1, CMD_FINISH
    sw      t1, 0(t0)
L1: j       L1
```

```
./obj_dir/top
 000000: 00000000 -------- -------- -------- --------:
 000000: 00000004 00000000 -------- -------- --------:
 000000: 00000008 00000004 00000000 -------- --------:
 000000: 0000000c 00000008 00000004 00000000 --------:
 000001: 00000010 0000000c 00000008 00000004 00000000:
 000002: 00000010 ssss000c -------- ssss0008 ssss0004:
 000002: 00000010 ssss000c -------- ssss0008 ssss0004:
 000002: 00000010 0000000c -------- 00000008 00000004:
 000003: 00000014 00000010 0000000c -------- 00000008:
 000003: 00000018 00000014 00000010 0000000c --------:
 000004: 0000001c 00000018 00000014 00000010 0000000c:
 000005: 00000020 0000001c 00000018 00000014 00000010:
 000006: 00000024 00000020 0000001c 00000018 00000014:
- top.v:47: Verilog $finish

===> mcycle                                :         13
===> minstret                              :          6
===> Total number of branch predictions    :          0
===> Total number of branch mispredictions :          0
===> simulation finish!!
```

5-stage pipelining processor of CFU Proving Ground

- IF (Instruction Fetch) 命令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。
- EX (Execution) 命令操作の実行またはアドレスの生成を行う。
- MA (Memory Access) 必要であれば、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、結果をレジスタに書き込む。



MA (Memory Access) stage

```
//------------------------------------------------------------------------------
// MA: Memory Access
//------------------------------------------------------------------------------
    // load unit
    wire [`XLEN-1:0] Ma_load_rslt;
    load_unit load_unit (
        .lsu_ctrl_i   (ExMa_lsu_ctrl),     // input  wire [`LSU_CTRL_WIDTH-1:0]
        .dbus_offset_i(ExMa_dbus_offset),  // input  wire    [OFFSET_WIDTH-1:0]
        .dbus_rdata_i (dbus_rdata_i),      // input  wire           [`XLEN-1:0]
        .rslt_o       (Ma_load_rslt)       // output wire           [`XLEN-1:0]
    );

    wire [`XLEN-1:0] Ma_rslt = ExMa_rslt | ExMa_mdc_rslt | Ma_load_rslt;

    always @(posedge clk_i) if (!w_stall) begin
        if (rst) begin
            MaWb_v  <= 0;
            MaWb_pc <= 0;
            MaWb_ir <= `NOP;
        end else if (!ExMa_stall) begin
            MaWb_v     <= Ma_v;
            MaWb_pc    <= ExMa_pc;
            MaWb_ir    <= ExMa_ir;
            MaWb_rf_we <= ExMa_rf_we;
            MaWb_rd    <= ExMa_rd;
            MaWb_rslt  <= Ma_rslt;
        end
    end
```

```
module load_unit (
    input  wire [ 5:0] lsu_ctrl_i,
    input  wire [ 1:0] dbus_offset_i,
    input  wire [31:0] dbus_rdata_i,
    output wire [31:0] rslt_o
);

    wire w_lb = lsu_ctrl_i[`LSU_CTRL_IS_BYTE];
    wire w_lh = lsu_ctrl_i[`LSU_CTRL_IS_HALFWORD];
    wire w_lw = lsu_ctrl_i[`LSU_CTRL_IS_WORD];
    wire w_signed = lsu_ctrl_i[`LSU_CTRL_IS_SIGNED];
    wire w_load = lsu_ctrl_i[`LSU_CTRL_IS_LOAD];
    wire [1:0] ost = dbus_offset_i;  // offset
    wire [31:0] d = dbus_rdata_i;  // data

    wire w_lb_sign = w_lb & ((ost==0) ? d[7] : (ost==1) ? d[15] :(ost==2) ? d[23] : d[31]) & w_signed;
    wire w_lh_sign = w_lh & ((ost[1] == 0) ? d[15] : d[31]) & w_signed;

    wire [7:0] w1, w2, w3, w4;
    assign w1   = (w_load==0) ? 0 : (w_lw || (w_lh & ost[1]==0) || (w_lb & ost==0)) ? d[7:0] :
                          (w_lb && ost==1) ? d[15:8] : ((w_lb && ost==2) ||
                          (w_lh && ost[1]==1)) ? d[23:16] : d[31:24];
    assign w2  = (w_load==0) ? 0 : (w_lw || (w_lh && ost[1]==0)) ? d[15:8] :
                          (w_lh && ost[1]==1) ? d[31:24] : (w_lb_sign) ? 8'hff : 0;
    assign w3 = (w_load == 0) ? 0 : (w_lw) ? d[23:16] : ((w_lb_sign) || (w_lh_sign)) ? 8'hff : 0;
    assign w4 = (w_load == 0) ? 0 : (w_lw) ? d[31:24] : ((w_lb_sign) || (w_lh_sign)) ? 8'hff : 0;
    assign rslt_o = {w4, w3, w2, w1};
endmodule
```

MA (Memory Access) stage

- ロード命令がロードしたデータを直後の命令が利用する場合には、データフォワーディングでオペランドを供給できない。そのため、直後の命令のデコードステージから実行ステージへの移動を1サイクル遅らせる（バブルをひとつ挿入する）。
- app/crt0.s を下図のコードに編集する。
- 次のコマンドで実行
  - make
  - make run

```
    .equ TO_HOST,    0x80000000
    .equ CMD_FINISH, 0x00020000
    .equ DMEM_ADDR,  0x10000000
    .align 4
    .section .text.init
    .globl _start
_start:
    li      x1, 1
    li      x2, DMEM_ADDR
    lw      x2, 0(x2)
    add     x3, x2, x2
    li      t0, TO_HOST
    li      t1, CMD_FINISH
    sw      t1, 0(t0)
L1: j       L1
```

```
./obj_dir/top
 000000: 00000000 -------- -------- -------- --------:
 000000: 00000004 00000000 -------- -------- --------:
 000000: 00000008 00000004 00000000 -------- --------:
 000000: 0000000c 00000008 00000004 00000000 --------:
 000001: 00000010 0000000c 00000008 00000004 00000000:
 000002: 00000010 0000000c -------- 00000008 00000004:
 000003: 00000014 00000010 0000000c -------- 00000008:
 000003: 00000018 00000014 00000010 0000000c --------:
 000004: 0000001c 00000018 00000014 00000010 0000000c:
 000005: 00000020 0000001c 00000018 00000014 00000010:
 000006: 00000024 00000020 0000001c 00000018 00000014:
- top.v:47: Verilog $finish

===> mcycle                                :          11
===> minstret                              :           6
===> Total number of branch predictions    :           0
===> Total number of branch mispredictions :           0
===> simulation finish!!
```

e3-cp9: 遅いデータメモリとキャッシュメモリ

- main.v を編集して、m_dmem を遅いデータメモリに変更する。(オリジナルの m_dmem の記述をコメントアウトして、右のコードを追加する。)
- 読み出しには MEM_LATENCY で定義するサイクル数の遅延が必要。
- re_i が1になると、読み出し処理を開始し、読み出しが完了したら、レジスタ r_rdata に読み出した値を保存する。
- 読み出しが完了してから、次の読み出し要求を受け付ける点に注意。
- 書き込みは、遅延なく実行される。

```
`define MEM_LATENCY 5  // a number greater than zero
module m_dmem (
    input  wire        clk_i,   //
    input  wire        re_i,    //
    input  wire        we_i,    //
    input  wire [31:0] addr_i,  //
    input  wire [31:0] wdata_i, //
    input  wire  [3:0] wstrb_i, //
    output wire [31:0] rdata_o, //
    output wire        oe       // output enable
);
    (* ram_style = "block" *) reg [31:0] dmem[0:`DMEM_ENTRIES-1];
    `include "memd.txt"

    wire [`DMEM_ADDRW-1:0] valid_addr = addr_i[`DMEM_ADDRW+1:2];

    reg [31:0] r_load_adr = 0;
    reg [31:0] rdata = 0;
    always @(posedge clk_i) begin
        if (we_i) begin
            if (wstrb_i[0]) dmem[valid_addr][7:0]   <= wdata_i[7:0];
            if (wstrb_i[1]) dmem[valid_addr][15:8]  <= wdata_i[15:8];
            if (wstrb_i[2]) dmem[valid_addr][23:16] <= wdata_i[23:16];
            if (wstrb_i[3]) dmem[valid_addr][31:24] <= wdata_i[31:24];
        end
        if (re_i) r_load_adr <= addr_i;
        rdata <= dmem[r_load_adr[`DMEM_ADDRW+1:2]];
    end

    reg [31:0] r_dm_wait = 0;
    reg [31:0] r_rdata = 0;
    reg [31:0] r_oe = 0;
    always @(posedge clk_i) begin
        r_dm_wait <= (r_dm_wait==0 && re_i) ? `MEM_LATENCY :
                     (r_dm_wait!=0) ? r_dm_wait - 1 : 0;
        r_rdata   <= (r_dm_wait==1) ? rdata : r_rdata;
        r_oe      <= (r_dm_wait==1);
    end

    assign rdata_o = r_rdata;
    assign oe = r_oe;
endmodule
```

遅いデータメモリ

e3-cp9: 遅いデータメモリとキャッシュメモリ

- top.v を編集して、赤色の部分を追加する。
  - ロード命令とストア命令の実行履歴を保存する mtrace.txt というファイルを生成する。

```
integer fp;
initial begin
    fp=$fopen("mtrace.txt","w");
    if(fp == 0)begin
        $display("File_Open_Error_!!!!!");
        $finish();
    end
end
reg [31:0] r_load_adr;
wire w_valid = !m0.rst && !cpu_sim_fini && !m0.cpu.stall_i && !m0.cpu.stall;
always @(posedge clk) if (w_valid) begin
    if (m0.dmem_re) r_load_adr <= m0.dbus_addr;
    if (m0.cpu.ExMa_v && m0.cpu.ExMa_lsu_ctrl[0])
      $fwrite(fp, "%07d: ld %08x %08x\n", minstret, r_load_adr, m0.dbus_rdata);
    if (m0.dmem_we)
      $fwrite(fp, "%07d: st %08x %08x %b\n",
              minstret, m0.dbus_addr, m0.dbus_wdata, m0.dbus_wstrb);
end
```

遅いデータメモリとキャッシュメモリ

- 右図を参考に、main.v を編集する。
- 正しいトレース mtrace_good.txt と、生成されるトレース mtrace.txt の比較により、間違っている箇所を特定する。

```
cpu cpu (
    .clk_i         (clk),         // input  wire
    .rst_i         (rst),         // input  wire
    .stall_i       (r_cpu_stall), // input  wire // Note
    .ibus_araddr_o (imem_raddr),  // output wire [`IBUS_ADDR_WIDTH-1:0]
    .ibus_rdata_i  (imem_rdata),  // input  wire [`IBUS_DATA_WIDTH-1:0]
    .dbus_addr_o   (dbus_addr),   // output wire [`DBUS_ADDR_WIDTH-1:0]
    .dbus_wvalid_o (dbus_we),     // output wire
    .dbus_wdata_o  (dbus_wdata),  // output wire [`DBUS_DATA_WIDTH-1:0]
    .dbus_wstrb_o  (dbus_wstrb),  // output wire [`DBUS_STRB_WIDTH-1:0]
    .dbus_rdata_i  (dbus_rdata)   // input  wire [`DBUS_DATA_WIDTH-1:0]
);


reg r_cpu_stall = 0;
always @(posedge clk) r_cpu_stall <= (dmem_re) ? 1 : (dmem_oe) ? 0 : r_cpu_stall;

wire [31:0] dmem_addr  = dbus_addr;
wire [31:0] dmem_wdata = dbus_wdata;
wire  [3:0] dmem_wstrb = dbus_wstrb;
wire        dmem_re    = !dbus_we & (dbus_addr[28]);
wire        dmem_we    =  dbus_we & (dbus_addr[28]);
wire [31:0] dmem_rdata;
wire        dmem_oe;
m_dmem dmem (
    .clk_i   (clk),          // input  wire
    .we_i    (dmem_we),      // input  wire
    .re_i    (dmem_re),      // input  wire // Note
    .addr_i  (dmem_addr),    // input  wire [ADDR_WIDTH-1:0]
    .wdata_i (dmem_wdata),   // input  wire [DATA_WIDTH-1:0]
    .wstrb_i (dmem_wstrb),   // input  wire [STRB_WIDTH-1:0]
    .rdata_o (dmem_rdata),   // output reg  [DATA_WIDTH-1:0]
    .oe      (dmem_oe)       //
);
```

MA (Memory Access) stage

- ロード命令がロードしたデータを直後の命令が利用する場合には、データフォワーディングでオペランドを供給できない。そのため、直後の命令のデコードステージから実行ステージへの移動を1サイクル遅らせる（バブルをひとつ挿入する）。
- app/crt0.s を下図のコードに編集する。
- 次のコマンドで実行
  - make
  - make run
  - cat mtrace.txt

```
    .equ TO_HOST,    0x80000000
    .equ CMD_FINISH, 0x00020000
    .equ DMEM_ADDR,  0x10000000
    .align 4
    .section .text.init
    .globl _start
_start:
    li      x1, 9
    li      x2, DMEM_ADDR
    sw      x1, 0(x2)
    lw      x2, 0(x2)
    add     x3, x2, x2
    li      t0, TO_HOST
    li      t1, CMD_FINISH
    sw      t1, 0(t0)
L1: j       L1
```

```
./obj_dir/top
 000000: 00000000 -------- -------- -------- --------: 00000000:
 000000: 00000004 00000000 -------- -------- --------: 00000000:
 000000: 00000008 00000004 00000000 -------- --------: 00000000:
 000000: 0000000c 00000008 00000004 00000000 --------: 00000000:
 000001: 00000010 0000000c 00000008 00000004 00000000: 00000000:
 000002: 00000014 00000010 0000000c 00000008 00000004: 00000000:
 000003: 00000014 00000010 -------- 0000000c 00000008: 00000000:  stall
 000003: 00000014 00000010 -------- 0000000c 00000008: 00000000:  stall
 000003: 00000014 00000010 -------- 0000000c 00000008: 00000000:  stall
 000003: 00000014 00000010 -------- 0000000c 00000008: 00000000:  stall
 000003: 00000014 00000010 -------- 0000000c 00000008: 00000000:  stall
 000003: 00000014 00000010 -------- 0000000c 00000008: 00000009:  stall
 000003: 00000014 00000010 -------- 0000000c 00000008: 00000009:
 000004: 00000018 00000014 00000010 -------- 0000000c: 00000009:
 000004: 0000001c 00000018 00000014 00000010 --------: 00000009:
 000005: 00000020 0000001c 00000018 00000014 00000010: 00000009:
 000006: 00000024 00000020 0000001c 00000018 00000014: 00000009:
 000007: 00000028 00000024 00000020 0000001c 00000018: 00000009:
- top.v:47: Verilog $finish

===> mcycle                                :         18
===> minstret                              :          7
===> Total number of branch predictions    :          0
===> Total number of branch mispredictions :          0
===> simulation finish!!
```

```
0000001: st 10000000 00000009 1111
0000003: ld 10000000 00000009
```

シンプルなアセンブリ言語のコードのシミュレーション

- top.v を編集して、赤色のコードを追加する。

```
    reg r_rst = 0;
    always @(posedge clk) r_rst <= !m0.rst;
    always @(posedge clk) if (r_rst) begin
        $write(" %06d: %x ", minstret, m0.cpu.r_pc);
        if (!m0.cpu.IfId_v) $write("-------- "); else if(m0.cpu.stall) $write("ssssssss "); else $write("%x ", m0.cpu.IfId_pc);
        if (!m0.cpu.IdEx_v) $write("-------- "); else if(m0.cpu.stall) $write("ssssssss "); else $write("%x ", m0.cpu.IdEx_pc);
        if (!m0.cpu.ExMa_v) $write("-------- "); else if(m0.cpu.stall) $write("ssssssss "); else $write("%x ", m0.cpu.ExMa_pc);
        if (!m0.cpu.MaWb_v) $write("--------:"); else if(m0.cpu.stall) $write(«ssssssss:"); else $write("%x:", m0.cpu.MaWb_pc);
        $write(" %d %d %08x: ", m0.dmem_we, m0.dmem_re, m0.cpu.dbus_rdata_i);
        if (m0.cpu.stall_i) $write(" stall");
        $write("\n");
    end
```

MA (Memory Access) stage

- ロード命令がロードしたデータを直後の命令が利用する場合には、データフォワーディングでオペランドを供給できない。そのため、直後の命令のデコードステージから実行ステージへの移動を1サイクル遅らせる（バブルをひとつ挿入する）。
- app/crt0.s を下図のコードに編集する。
- 次のコマンドで実行
  - make
  - make run
  - cat mtrace.txt

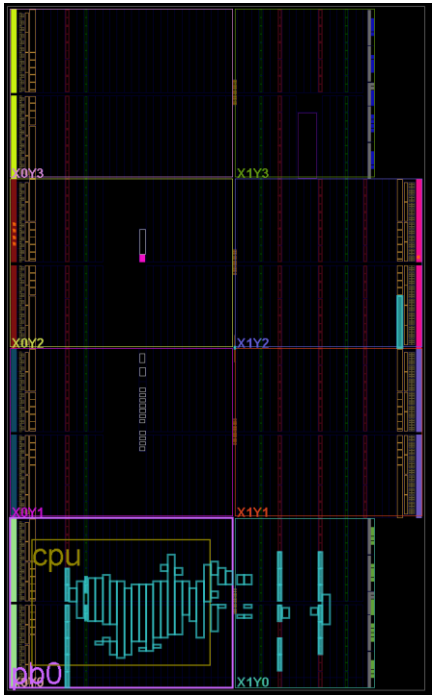
```
    .equ TO_HOST,    0x80000000
    .equ CMD_FINISH, 0x00020000
    .equ DMEM_ADDR,  0x10000000
    .align 4
    .section .text.init
    .globl _start
_start:
    li      x1, 9
    li      x2, DMEM_ADDR
    sw      x1, 0(x2)
    lw      x2, 0(x2)
    add     x3, x2, x2
    li      t0, TO_HOST
    li      t1, CMD_FINISH
    sw      t1, 0(t0)
L1: j       L1
```

```
./obj_dir/top
 000000: 00000000 -------- -------- -------- --------: 0 0 00000000:
 000000: 00000004 00000000 -------- -------- --------: 0 0 00000000:
 000000: 00000008 00000004 00000000 -------- --------: 0 0 00000000:
 000000: 0000000c 00000008 00000004 00000000 --------: 0 0 00000000:
 000001: 00000010 0000000c 00000008 00000004 00000000: 1 0 00000000:
 000002: 00000014 00000010 0000000c 00000008 00000004: 0 1 00000000:
 000003: 00000014 00000010 -------- 0000000c 00000008: 0 0 00000000:  stall
 000003: 00000014 00000010 -------- 0000000c 00000008: 0 0 00000000:  stall
 000003: 00000014 00000010 -------- 0000000c 00000008: 0 0 00000000:  stall
 000003: 00000014 00000010 -------- 0000000c 00000008: 0 0 00000000:  stall
 000003: 00000014 00000010 -------- 0000000c 00000008: 0 0 00000000:  stall
 000003: 00000014 00000010 -------- 0000000c 00000008: 0 0 00000009:  stall
 000003: 00000014 00000010 -------- 0000000c 00000008: 0 0 00000009:
 000004: 00000018 00000014 00000010 -------- 0000000c: 0 0 00000009:
 000004: 0000001c 00000018 00000014 00000010 --------: 0 0 00000009:
 000005: 00000020 0000001c 00000018 00000014 00000010: 0 0 00000009:
 000006: 00000024 00000020 0000001c 00000018 00000014: 0 0 00000009:
 000007: 00000028 00000024 00000020 0000001c 00000018: 0 0 00000009:
- top.v:47: Verilog $finish

===> mcycle                                :         18
===> minstret                              :          7
===> Total number of branch predictions    :          0
===> Total number of branch mispredictions :          0
===> simulation finish!!
```

```
0000001: st 10000000 00000009 1111
0000003: ld 10000000 00000009
```

論理合成、配置・配線の結果

- Vivado 2024.1, xc7a100tcsg324-1
- `define CLK_FREQ_MHZ 120 // operating clock frequency in MHz
- `define IMEM_SIZE (32*1024) // instruction memory size in byte
- `define DMEM_SIZE (16*1024) // data memory size in byte

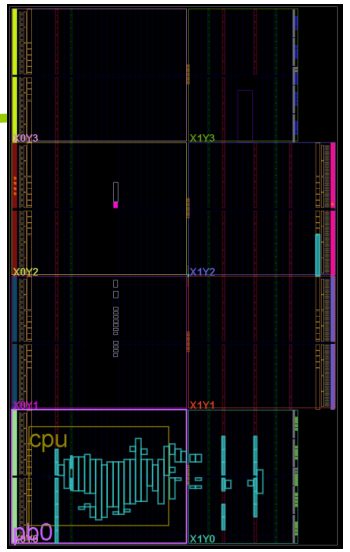


Resource	Utilization	Available	Utilization %
LUT	1652	63400	2.61
LUTRAM	44	19000	0.23
FF	912	126800	0.72
BRAM	19	135	14.07
DSP	4	240	1.67
IO	5	210	2.38
BUFG	2	32	6.25
MMCM	1	6	16.67

Name	Slack ^1	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement
Path 1	0.661	3	10	cpu/bimodal/btb_reg/CLKBWRCLK	imem/rdata_reg_3/ADDRARDADDR[14]	6.993	3.054	3.939	8.3
Path 2	0.677	3	10	cpu/bimodal/btb_reg/CLKBWRCLK	imem/rdata_reg_7/ADDRARDADDR[14]	6.967	3.054	3.913	8.3
Path 3	0.696	3	10	cpu/bimodal/btb_reg/CLKBWRCLK	imem/rdata_reg_5/ADDRARDADDR[9]	6.876	3.054	3.822	8.3
Path 4	0.696	3	10	cpu/bimodal/btb_reg/CLKBWRCLK	imem/rdata_reg_7/ADDRARDADDR[12]	6.947	3.054	3.893	8.3
Path 5	0.706	3	10	cpu/bimodal/btb_reg/CLKBWRCLK	imem/rdata_reg_5/ADDRARDADDR[12]	6.866	3.054	3.812	8.3
Path 6	0.722	3	10	cpu/bimodal/btb_reg/CLKBWRCLK	imem/rdata_reg_3/ADDRARDADDR[12]	6.932	3.054	3.878	8.3
Path 7	0.745	3	10	cpu/bimodal/btb_reg/CLKBWRCLK	imem/rdata_reg_3/ADDRARDADDR[6]	6.908	3.054	3.854	8.3
Path 8	0.763	3	10	cpu/bimodal/btb_reg/CLKBWRCLK	imem/rdata_reg_4/ADDRARDADDR[12]	6.805	3.054	3.751	8.3
Path 9	0.781	3	10	cpu/bimodal/btb_reg/CLKBWRCLK	imem/rdata_reg_5/ADDRARDADDR[6]	6.791	3.054	3.737	8.3
Path 10	0.784	3	10	cpu/bimodal/btb_reg/CLKBWRCLK	imem/rdata_reg_7/ADDRARDADDR[9]	6.859	3.054	3.805	8.3

論理合成、配置・配線の結果

- Vivado 2024.1, xc7a100tcsg324-1
- `define CLK_FREQ_MHZ 120 // operating clock frequency in MHz
- `define IMEM_SIZE (32*1024) // instruction memory size in byte
- `define DMEM_SIZE (16*1024) // data memory size in byte



Name	Slice LUTs (63400)	Slice Registers (126800)	Slice (15850)	LUT as Logic (63400)	LUT as Memory (19000)	Block RAM Tile (135)	DSPs (240)	Bonded IOB (210)	BUFGCTRL (32)	MMCME2_ADV (6)
N main	1652	912	572	1608	44	19	4	5	2	1
clk_wiz_0 (clk_wiz_0)	0	0	0	0	0	0	0	0	2	1
cpu (cpu)	1383	550	449	1383	0	1	4	0	0	0
alu (alu)	122	0	54	122	0	0	0	0	0	0
bimodal (bimodal)	28	0	19	28	0	1	0	0	0	0
bru (bru)	117	0	51	117	0	0	0	0	0	0
divider (divider)	379	108	112	379	0	0	0	0	0	0
multiplier (multiplier)	95	38	48	95	0	0	4	0	0	0
store_unit (store_unit)	79	0	35	79	0	0	0	0	0	0
dmem (m_dmem)	43	45	19	43	0	4	0	0	0	0
imem (m_imem)	27	0	10	27	0	8	0	0	0	0
perf (perf_cntr)	31	98	40	31	0	0	0	0	0	0
st7789_disp (m_st7789_disp)	123	165	68	123	0	0	0	0	0	0
vmem (vmem)	2	52	15	2	0	6	0	0	0	0