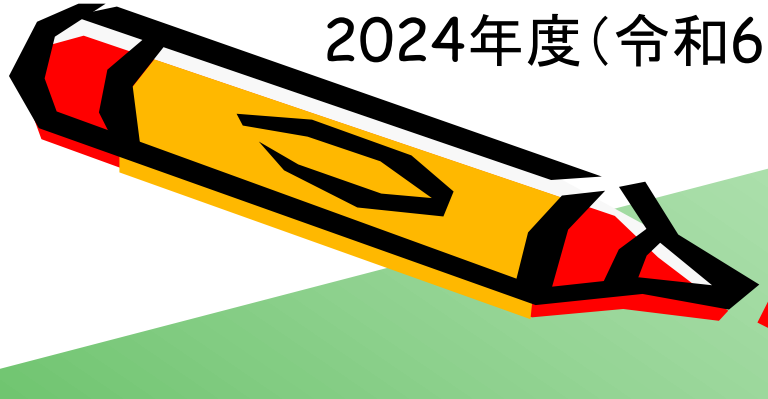


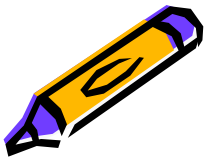
2024年度（令和6年）版

Ver. 2024-10-25a

Course number: CSC.T363

コンピュータアーキテクチャ
Computer Architecture

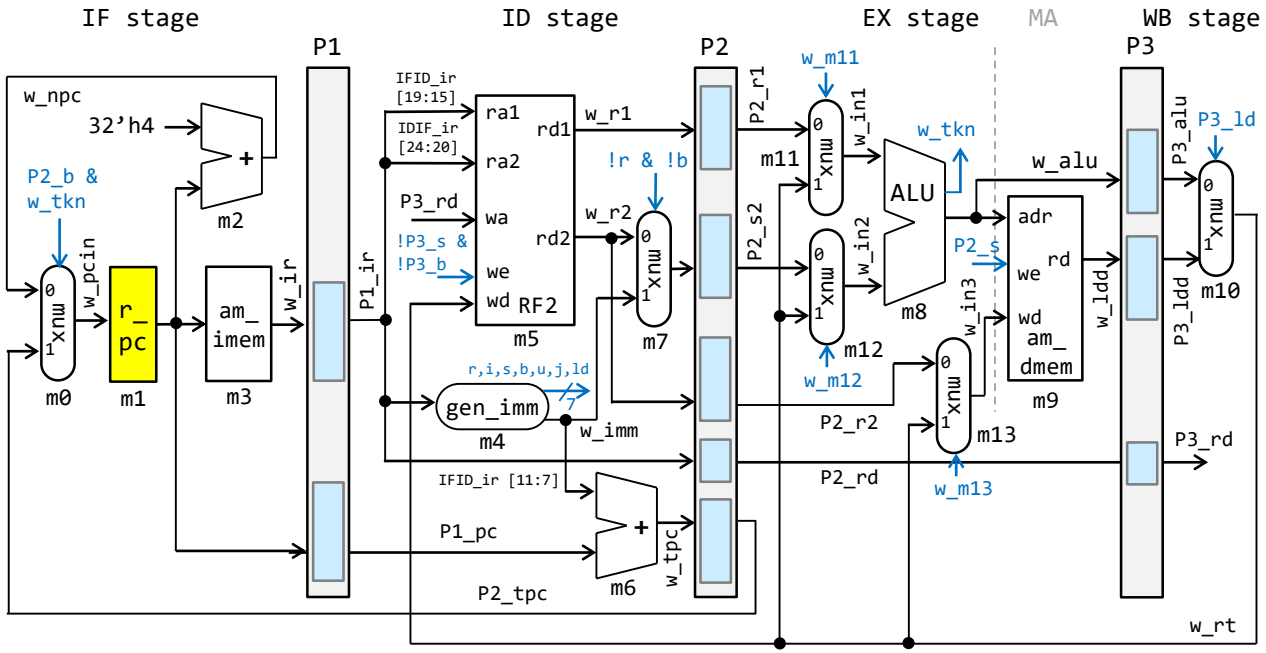
8. 分岐予測
Branch Prediction

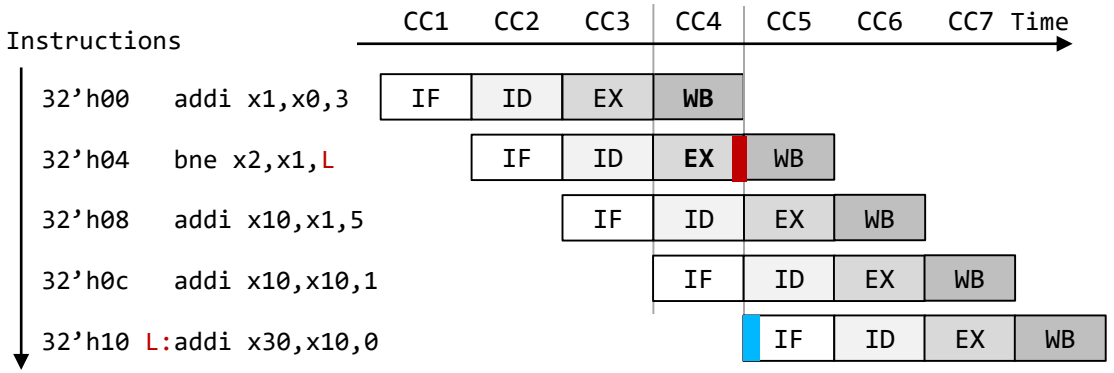


www.arch.cs.titech.ac.jp/lecture/CA/
Tue 13:30-15:10, 15:25-17:05
Fri 13:30-15:10

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

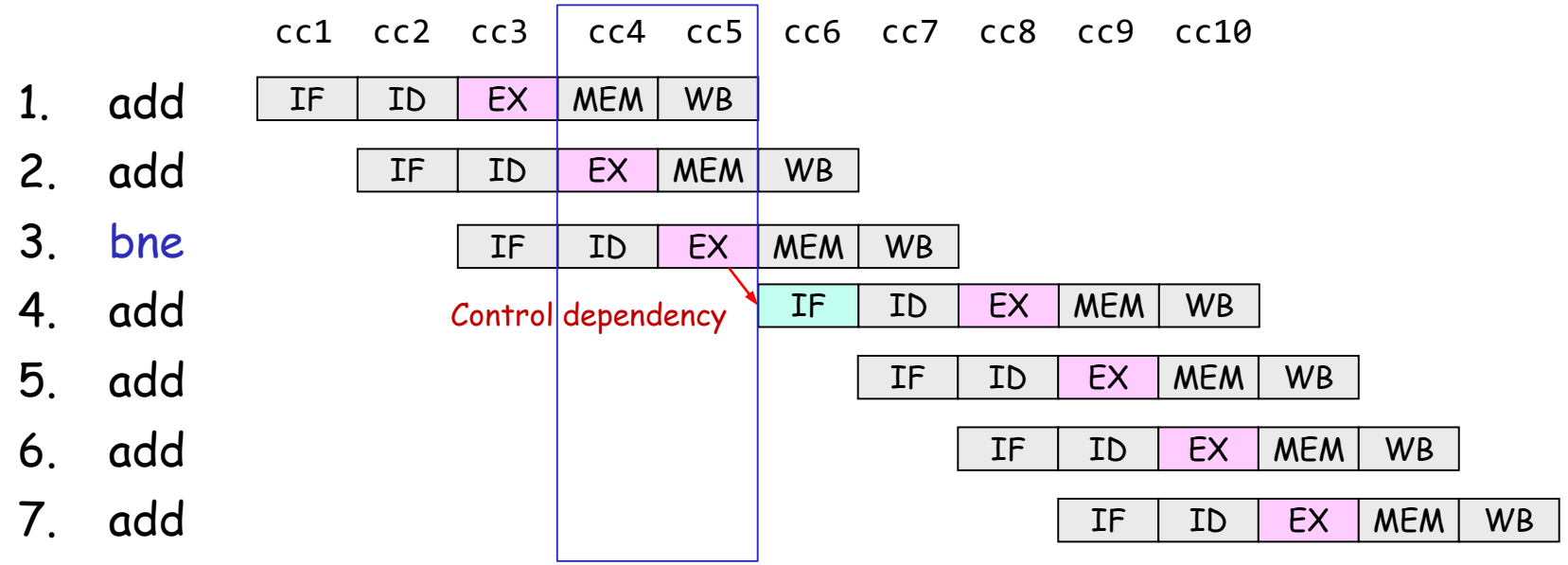
proc8: 4-stage pipelining processor





Why do branch instructions degrade IPC?

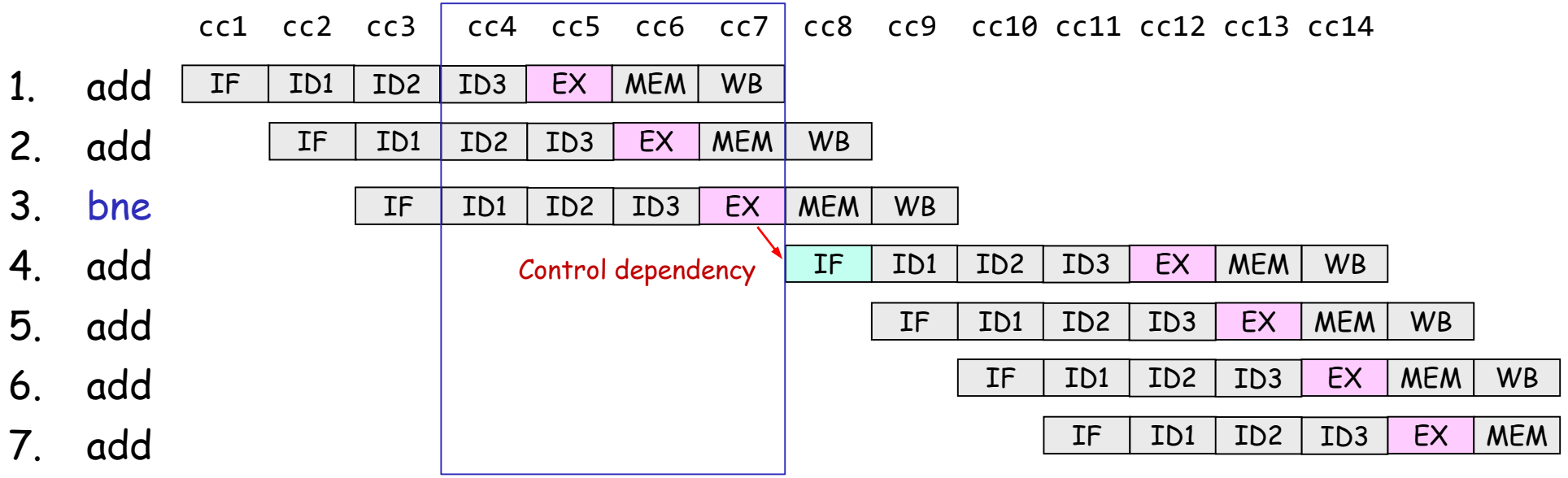
- The branch taken / untaken is determined in the execution stage of the branch.
- The conservative approach of stalling instruction fetch until the branch direction is determined.



five stages pipelined processor executing instruction sequence with a branch

Deeper pipeline

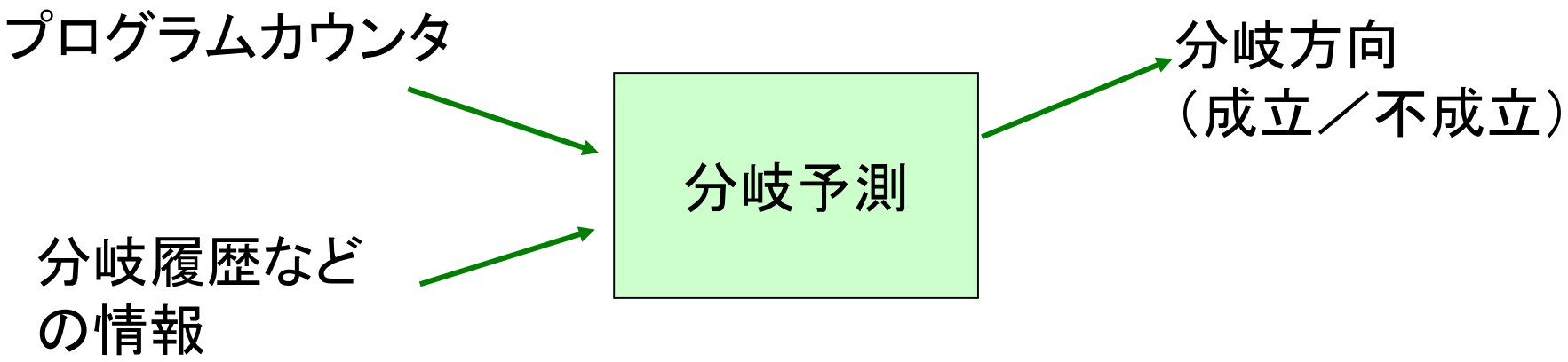
- In conservative approach, IPC degradation will be significant by deeper pipeline



高いバンド幅の命令フェッチ

- パイプラインにバブルを生じさせないために，条件分岐命令をフェッチした時に次の3つを予測 (prediction) する.
  - フェッチしている命令が分岐命令か？
  - 分岐先アドレス(32bit)
  - 分岐方向
    - 分岐成立(Taken)か分岐不成立(Not taken, Untaken)か？
- Speculation assuming the prediction is true
  - No penalty by a branch instruction when the prediction hits
  - Recovery when the miss is detected

分岐方向の予測(分岐予測)



bne命令と即値

31 25	24 20	19 15	14 12	11 7	6 0	
imm[12],imm[10:5]	rs2	rs1	funct3	imm[4:1],imm[11]	opcode	B-type

imm[12],imm[10:5]	rs2	rs1	000	imm[4:1],imm[11]	1100011	B-type	**beq**
imm[12],imm[10:5]	rs2	rs1	001	imm[4:1],imm[11]	1100011	B-type	**bne**
imm[12],imm[10:5]	rs2	rs1	100	imm[4:1],imm[11]	1100011	B-type	**blt**
imm[12],imm[10:5]	rs2	rs1	101	imm[4:1],imm[11]	1100011	B-type	**bge**
imm[12],imm[10:5]	rs2	rs1	110	imm[4:1],imm[11]	1100011	B-type	**bltu**
imm[12],imm[10:5]	rs2	rs1	111	imm[4:1],imm[11]	1100011	B-type	**bgeu**

imm[12:2]	imm[12:1]	imm[12:1]	{imm[12],imm[10:5],imm[4:1],imm[11]}
-10	-20	12'b111111101100	12'b1111110_11001
-9	-18	12'b111111101110	12'b1111110_11101
-8	-16	12'b111111110000	12'b1111111_00001
-7	-14	12'b111111110010	12'b1111111_00101
-6	-12	12'b111111110100	12'b1111111_01001
-5	-10	12'b111111110110	12'b1111111_01101
-4	-8	12'b111111111000	12'b1111111_10001
-3	-6	12'b111111111010	12'b1111111_10101
-2	-4	12'b111111111100	12'b1111111_11001
-1	-2	12'b111111111110	12'b1111111_11101
0	0	12'b000000000000	12'b0000000_00000
1	2	12'b000000000010	12'b0000000_00100
2	4	12'b000000000100	12'b0000000_01000
3	6	12'b000000000110	12'b0000000_01100
4	8	12'b000000001000	12'b0000000_10000

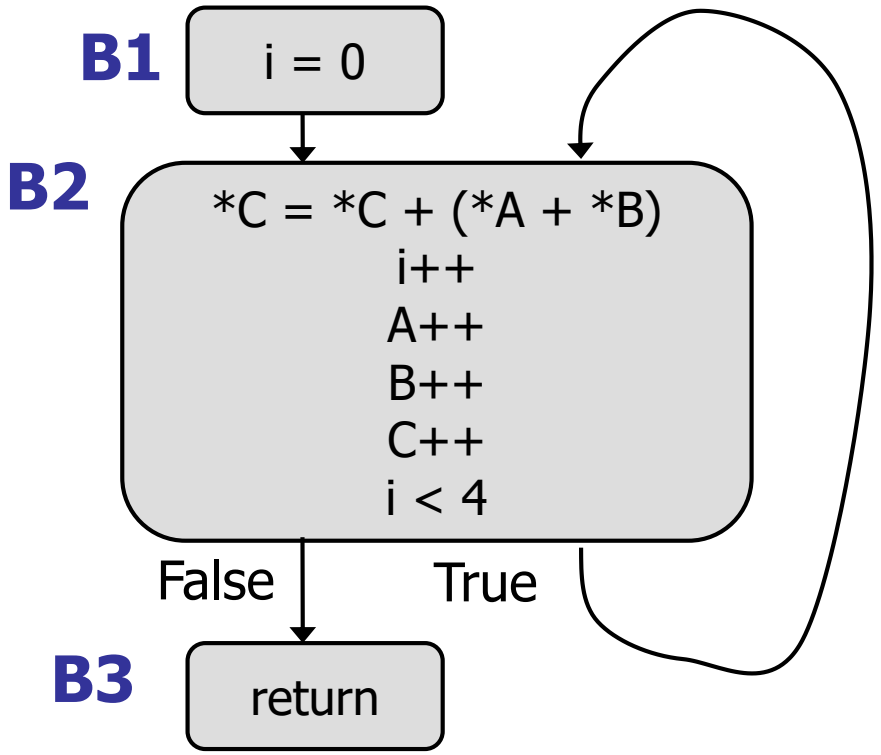
サンプルプログラム

```
1  `MM[0]={12'd0,5'd0,3'h0,5'd10,7'h13};          // 00   addi x10,x0,0
2  `MM[1]={12'd0,5'd0,3'h0,5'd3,7'h13};           // 04   addi x3,x0,0
3  `MM[2]={12'd101,5'd0,3'h0,5'd1,7'h13};         // 08   addi x1,x0,101
4  `MM[3]={7'd0,5'd3,5'd10,3'h0,5'd10,7'h33};     // 0c L:add  x10,x10,x3
5  `MM[4]={12'd1,5'd3,3'h0,5'd3,7'h13};           // 10   addi x3,x3,1
6  `MM[5]={~12'd0,5'd1,5'd3,3'h1,5'b11001,7'h63};// 14   bne  x3,x1,L
7  `MM[6]=32'h00050f13;                           // 18   HALT
```

```
1  `MM[0]={12'd0,5'd0,3'h0,5'd10,7'h13};          // 00   addi x10,x0,0
2  `MM[1]={12'd0,5'd0,3'h0,5'd3,7'h13};           // 04   addi x3,x0,0
3  `MM[2]={12'd101,5'd0,3'h0,5'd1,7'h13};         // 08   addi x1,x0,101
4  `MM[3]={7'd0,5'd3,5'd10,3'h0,5'd10,7'h33};     // 0c L:add  x10,x10,x3
5  `MM[4]={~12'd0,5'd0,5'd0,3'h1,5'b11101,7'h63};// 10   bne  x0,x0,L
6  `MM[5]={12'd1,5'd3,3'h0,5'd3,7'h13};           // 14   addi x3,x3,1
7  `MM[6]={~12'd0,5'd1,5'd3,3'h1,5'b10001,7'h63};// 18   bne  x3,x1,L
8  `MM[7]=32'h00050f13;                           // 1c   HALT
```

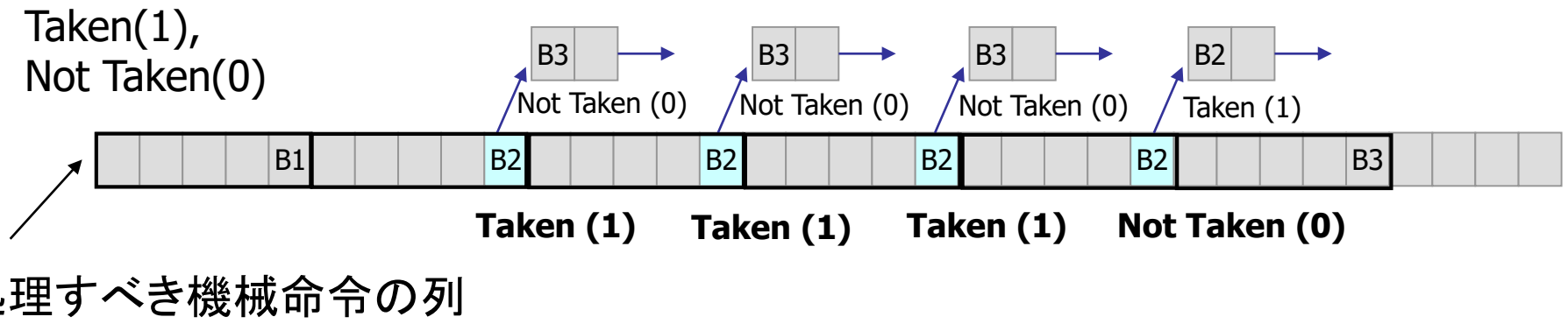
サンプルプログラム Vector Add

```
#define VSIZE 4
void vadd(long *A, long *B, long *C){
  for(i=0; i<VSIZE; i++)
    C[i] += (A[i] + B[i]);
}
```



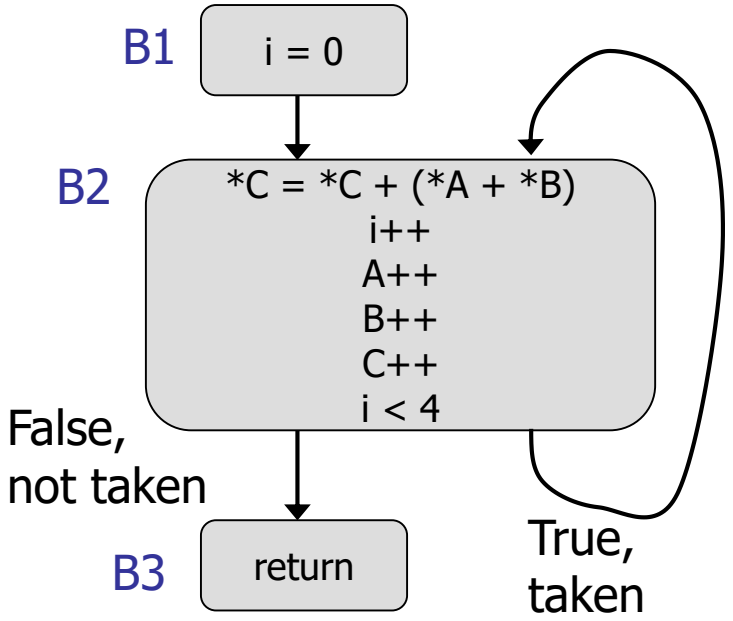
制御フローグラフ

シンプルな分岐予測: Branch Always



処理すべき機械命令の列

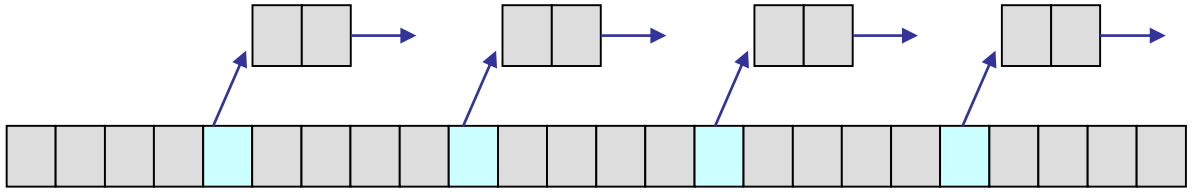
- Branch Always: 常に分岐が成立すると予測する.
  - 上の例では，予測成功率は75%，ミス率25%
  - 予測のためのメモリを必要としない.
  - 予測とよぶほどのものではない.



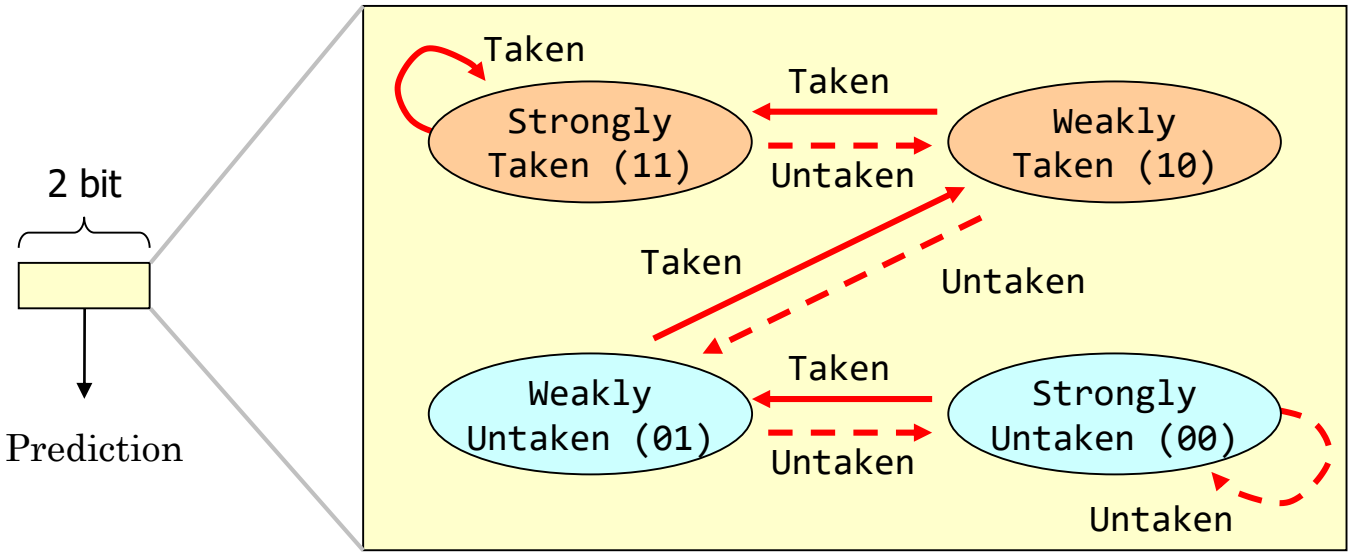
シンプルな分岐予測: 2ビットカウンタ方式

トレースデータ
（分岐アドレス，分岐結果）

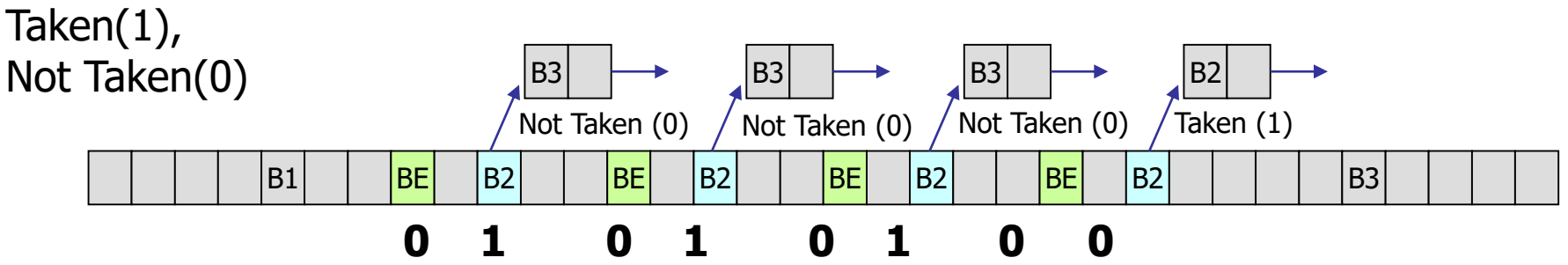
0040d6c5 1	0040d89c 1
0040d6b8 1	0040d89c 1
0040d6bc 0	0040d89c 1
0040d6c5 0	0040d89c 1
0040d6df 0	0040d89c 1
0040d71f 0	0040d89c 1
0040d736 0	0040d89c 0
0040d7ab 0	0040d8a2 0
0040d7cd 0	0040d8c0 1
0040d7f9 0	0040d8c4 0
0040d81e 1	0040d8cd 1
0040d7f9 1	0040d8c0 0
0040d81e 1	0040d8c4 1
0040d7f9 0	0040d8cd 1
0040d81e 0	0040d8c0 1
0040d83d 0	0040d8c4 0
0040d86d 1	0040d8cd 0
0040d86d 1	0040d8e7 0
0040d86d 1	0040d923 1
0040d86d 1	0040d7ab 0
0040d86d 1	0040d7cd 0
0040d86d 1	0040d7f9 0



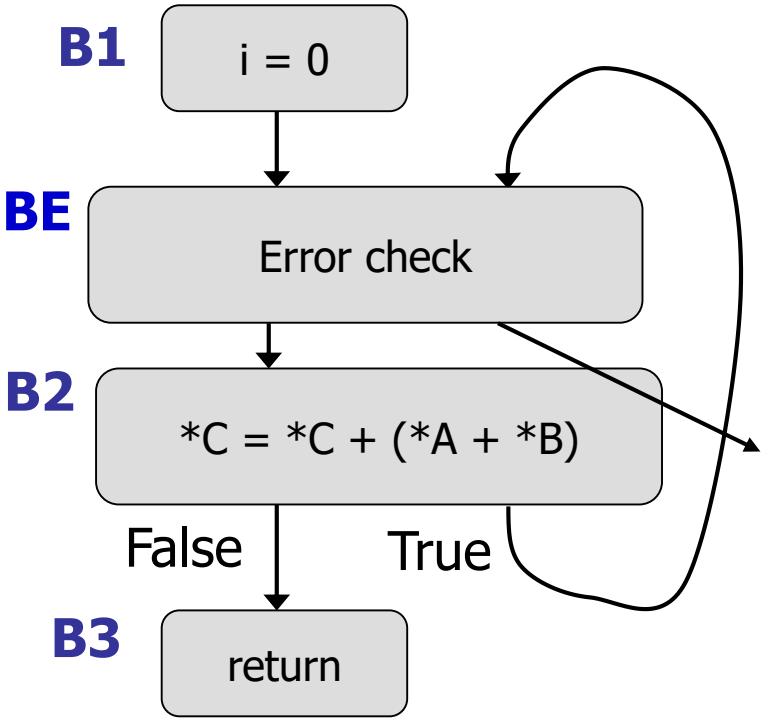
- 2ビットカウンタ方式
  - 大域的な偏り，局所性の利用
  - 2ビットカウンタの状態に応じて予測
  - 予測のためのメモリは2ビット
  - 状態の更新



サンプルプログラム Vector Add

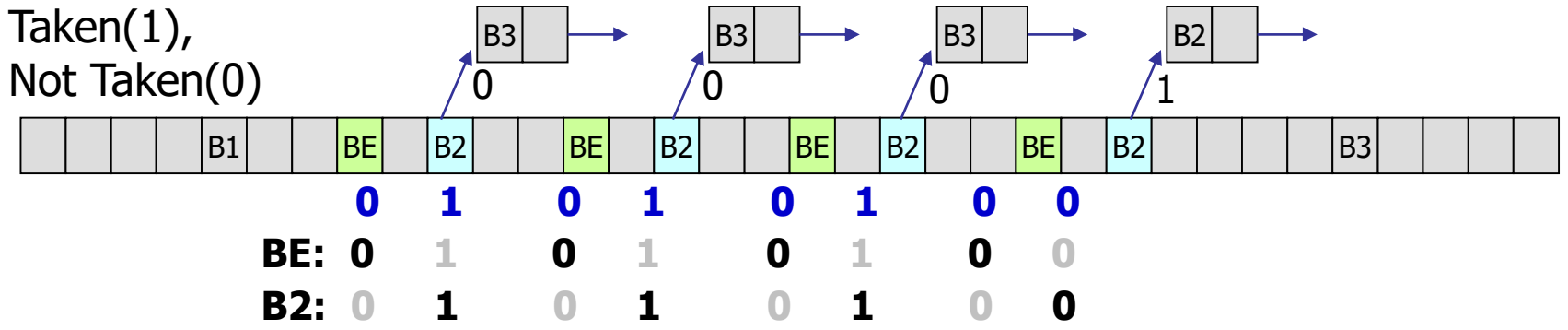



```
#define VSIZE 4
void vadd(long *A, long *B, long *C){
  for(i=0; i<VSIZE; i++) {
    if(A[i]<0) error_routine();
    C[i] += (A[i] + B[i]);
  }
}
```

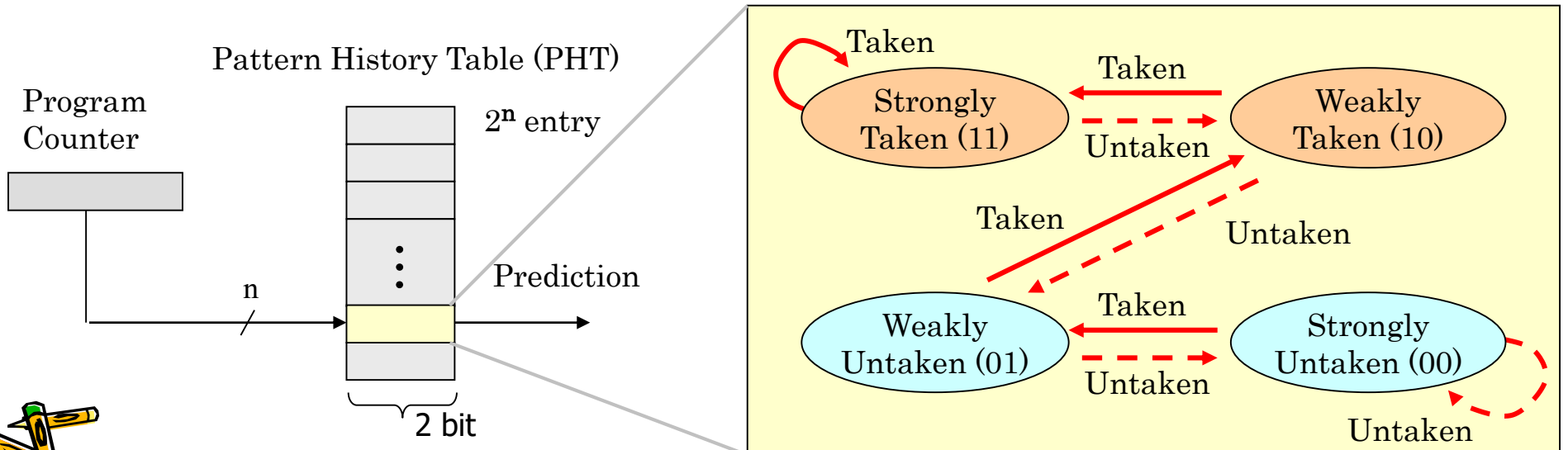


制御フローグラフ

Bimodal branch predictor (ISCA 1981)

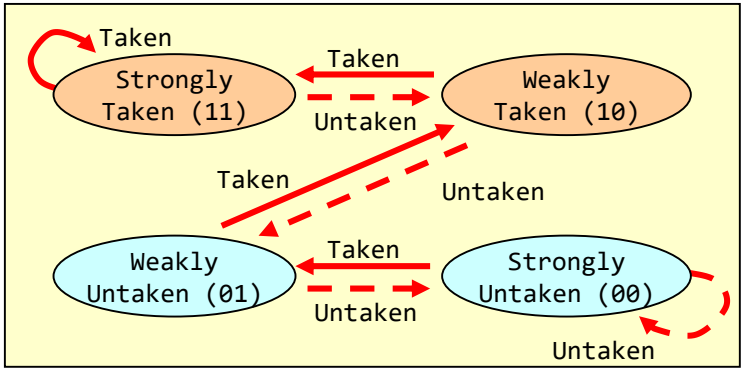


- 分岐アドレス（プログラムカウンタ）毎に履歴を切り替える
  - 分岐アドレスによりパターン履歴表（PHT）のインデックスを作成
- パターン履歴表は2ビットカウンタの配列.



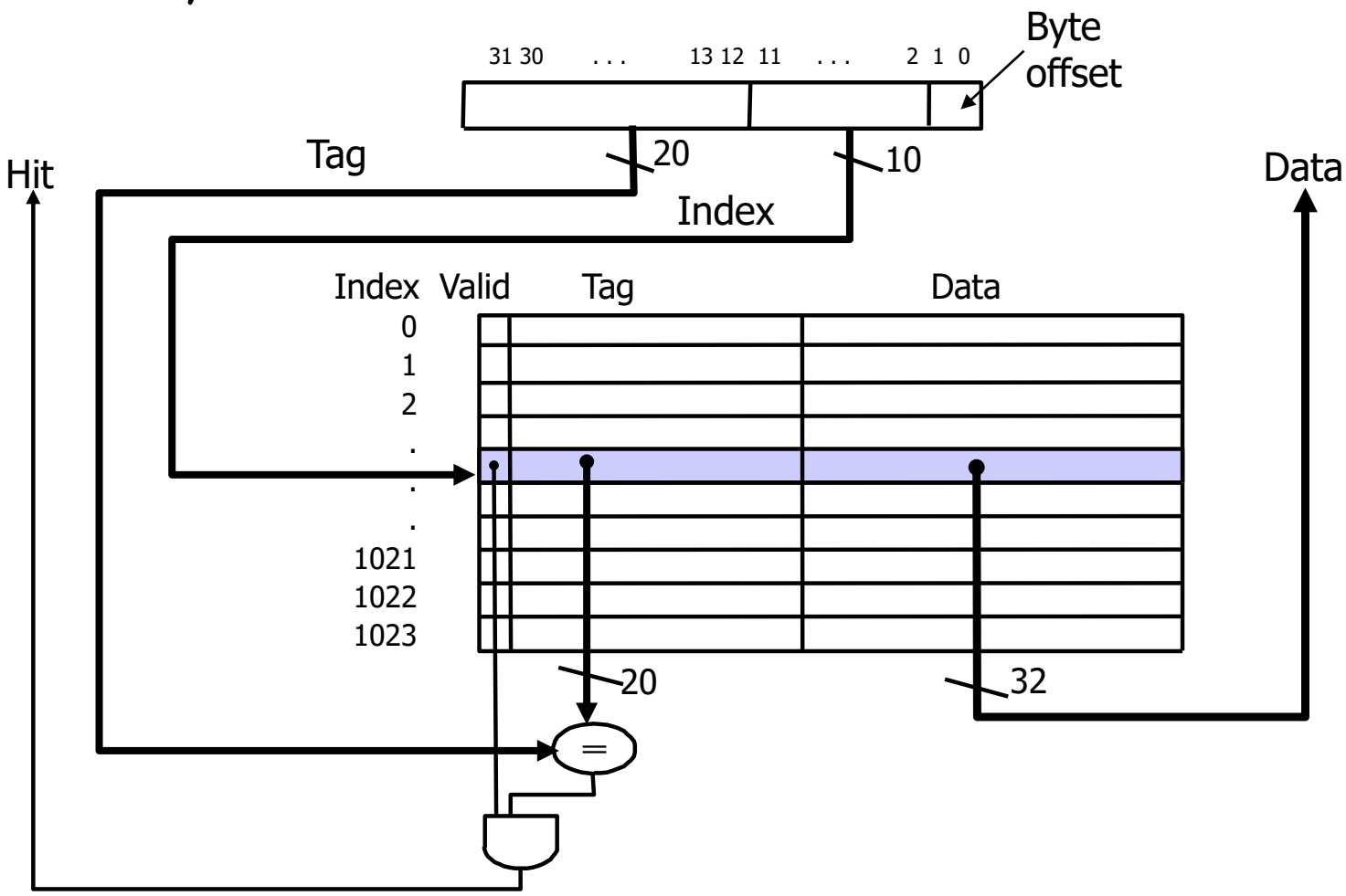
Bimodal branch predictor (ISCA 1981)

```
 1  module m_bimodal(w_clk, w_radr, w_pred, w_wadr, w_we, w_tkn);
 2    input  wire w_clk, w_we;
 3    input  wire [4:0] w_wadr, w_radr;
 4    input  wire w_tkn;
 5    output wire w_pred;
 6    reg [1:0] mem [0:31];
 7    wire [1:0] w_data = mem[w_radr];
 8    assign w_pred = w_data[1];
 9    wire [1:0] w_cnt  = mem[w_wadr];
10    always @(posedge w_clk) if (w_we)
11      mem[w_wadr] <= (w_cnt < 3 &  w_tkn) ? w_cnt + 1 :
12                     (w_cnt > 0 & !w_tkn) ? w_cnt - 1 : w_cnt;
13    integer i; initial for (i=0; i<32; i=i+1) mem[i] = 1;
14  endmodule
```



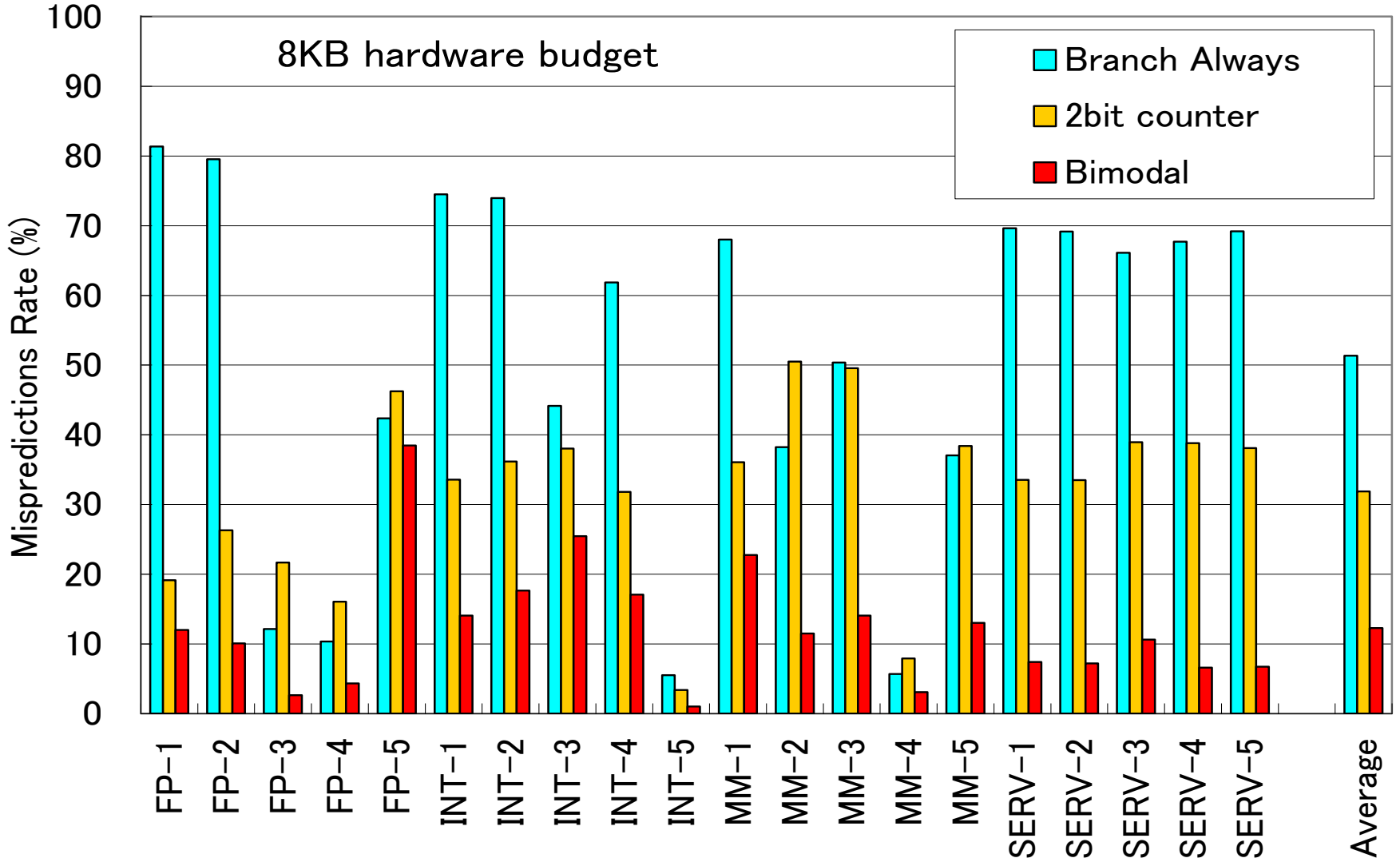
Direct Mapped Cache Example

- One word/block, cache size = 1K words



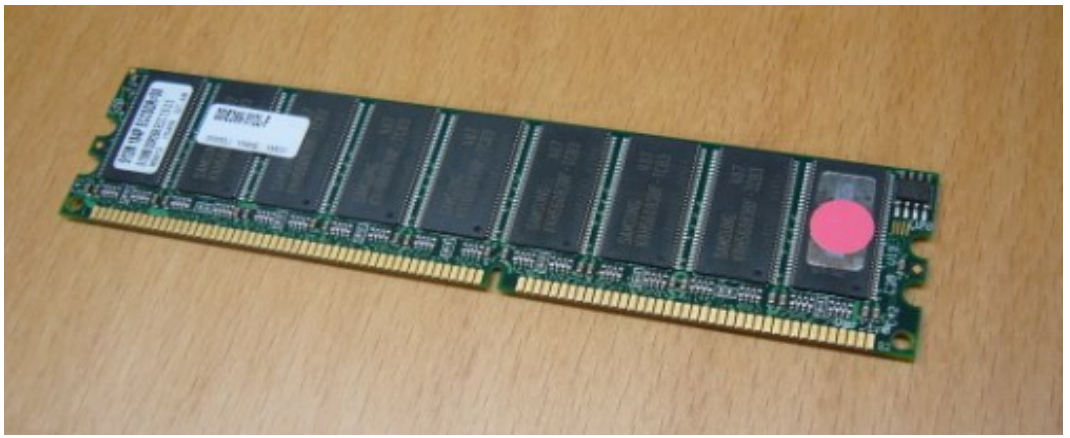
What kind of locality are we taking advantage of?

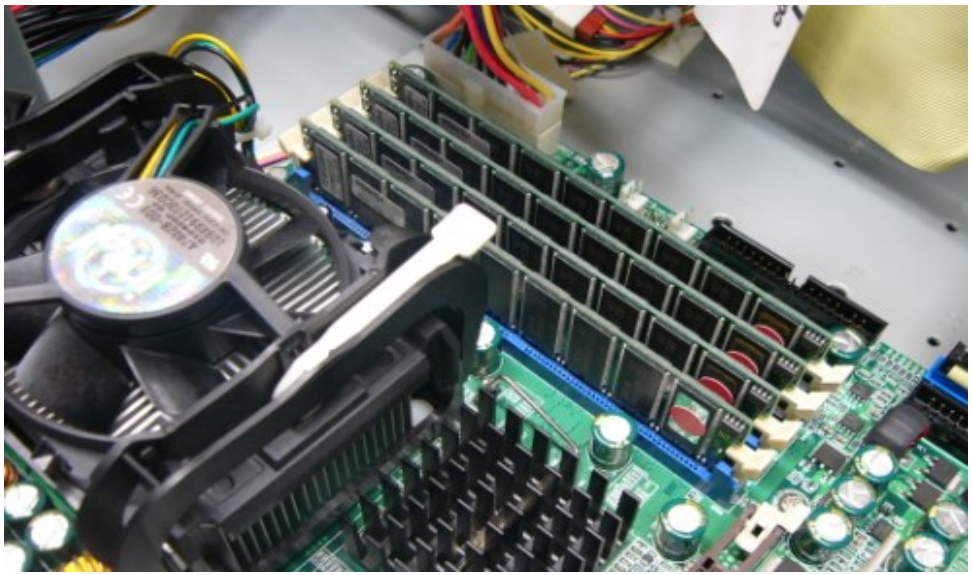
ここまでの分岐予測の精度（予測ミス率）



Benchmark for CBP(2004) by Intel MRL and IEEE TC uARCH.

DRAM (dynamic random access memory)





https://github.com/kisek/fpga_arty_a7_dram

```
/** main.v for Arty A7-35T (xc7a35ticsg324-1L)          ArchLab, Institute of Science Tokyo **/
/** DDR3 SDRAM: MT41K128M16XX-15E for MT41K128M16JT-125 of Arty A7-35T                      **/
/*****************************************************************************************/
`default_nettype none

`define DDR3_DQ_WIDTH   16
`define DDR3_DQS_WIDTH  2
`define DDR3_ADR_WIDTH  14
`define DDR3_BA_WIDTH   3
`define DDR3_DM_WIDTH   2
`define APP_ADDR_WIDTH  28
`define APP_CMD_WIDTH   3
`define APP_DATA_WIDTH  128
`define APP_MASK_WIDTH  16
`define CMD_READ        3'b001
`define CMD_WRITE       3'b000

module m_main (
    input  wire w_clk,                          // 100MHz clock signal
    output wire [3:0] w_led,                    // LED
    inout  wire [`DDR3_DQ_WIDTH-1 : 0]  ddr3_dq,
    inout  wire [`DDR3_DQS_WIDTH-1 : 0] ddr3_dqs_n,
    inout  wire [`DDR3_DQS_WIDTH-1 : 0] ddr3_dqs_p,
    output wire [`DDR3_ADR_WIDTH-1 : 0] ddr3_addr,
    output wire [`DDR3_BA_WIDTH-1 : 0]  ddr3_ba,
    output wire                         ddr3_ras_n,
    output wire                         ddr3_cas_n,
    output wire                         ddr3_we_n,
    output wire                         ddr3_reset_n,
    output wire [0:0]                   ddr3_ck_p,
    output wire [0:0]                   ddr3_ck_n,
    output wire [0:0]                   ddr3_cke,
    output wire [0:0]                   ddr3_cs_n,
    output wire [`DDR3_DM_WIDTH-1 : 0]  ddr3_dm,
    output wire [0:0]                   ddr3_odt
);

    wire                        sys_clk;     // input clock (166.67MHz),
    wire                        ref_clk;     // reference clock (200MHz),
    wire                        sys_rst = 0; // reset (active-high)
    clk_wiz_0 m0 (sys_clk, ref_clk, w_clk);

    reg  [`APP_ADDR_WIDTH-1 : 0] r_app_addr = 0;
    reg  [`APP_CMD_WIDTH-1 : 0] r_app_cmd  = 0;
    reg                         r_app_en = 0;
    reg                         r_app_wdf_wren = 0;
    reg  [`APP_DATA_WIDTH-1 : 0] r_app_wdf_data = {32'h1, 32'h1, 32'h1, 32'h1};
    reg  [`APP_MASK_WIDTH-1 : 0] r_app_wdf_mask = 0;

    wire [`APP_DATA_WIDTH-1 : 0] app_rd_data;
    wire                        app_rd_data_valid;
    wire                        app_rdy;
    wire                        app_wdf_rdy;

    wire                        w_ui_clk; // 333.33MHz / 4 = 83.33MHz
    wire                        init_calib_complete;
```

```
#include <stdio.h>
#define N  (1 << 25)
int dram[N];
int main() {
    for (int addr = 0; addr < N; addr++) dram[addr] = 1; /// WRITE

    int sum = 0;
    for (int addr = 0; addr < N; addr++) sum += dram[addr]; /// READ

    printf("%x %d\n", sum, sum);
    return 0;
}
```

The final sum will be 33,554,432.

```
reg [3:0] r_state = 0;
reg [31:0] r_sum = 0;
always @(posedge w_ui_clk) if (init_calib_complete) begin
   if (r_state==0 && app_rdy && app_wdf_rdy) begin  ///// WRITE_1
       r_app_en       <= 1;
       r_app_wdf_wren <= 1;
       r_app_cmd      <= `CMD_WRITE;
       r_state        <= 1;
   end
   else if (r_state==1) begin ///// WRITE_2
       if (app_rdy && app_wdf_rdy && r_app_en) begin
           r_app_en <= 0;
           r_app_wdf_wren <= 0;
       end
       if (r_app_en==0 && r_app_wdf_wren==0) begin
           r_app_addr <= r_app_addr + 8;
           r_state <= (r_app_addr[27:3]==25'h1ffffff) ? 2 : 0;
       end
   end
   else if (r_state==2) begin ///// INIT_FOR_READ
       r_app_addr <= 0;
       r_state    <= 3;
   end
   else if (r_state==3 && app_rdy) begin ///// READ_1
       r_app_en       <= 1;
       r_app_wdf_wren <= 0;
       r_app_cmd      <= `CMD_READ;
       r_state        <= 4;
   end
   else if (r_state==4) begin  ///// READ_2
       if (app_rdy && r_app_en) r_app_en <= 0;
       if (app_rd_data_valid) begin
           r_app_addr <= r_app_addr + 8;
           r_sum <= r_sum + app_rd_data[31:0];
           r_state <= (r_app_addr[27:3]==25'h1ffffff) ? 5 : 3;
       end
   end
end
assign w_led = {init_calib_complete, r_state[2:0]};

vio_0 vio0 (w_ui_clk, r_app_addr, r_sum);
```