Fiscal Year 2025

Ver. 2025-12-11a

Course number: CSC.T440
School of Computing,
Graduate major in Computer Science

Computer Organization and Architecture

2. Instruction Level Parallelism: Pipelining Processor and Branch Prediction

www.arch.cs.titech.ac.jp/lecture/coa/
Room No. M-112(H117), Lecture (Face-to-face)
Thr 13:30-15:10

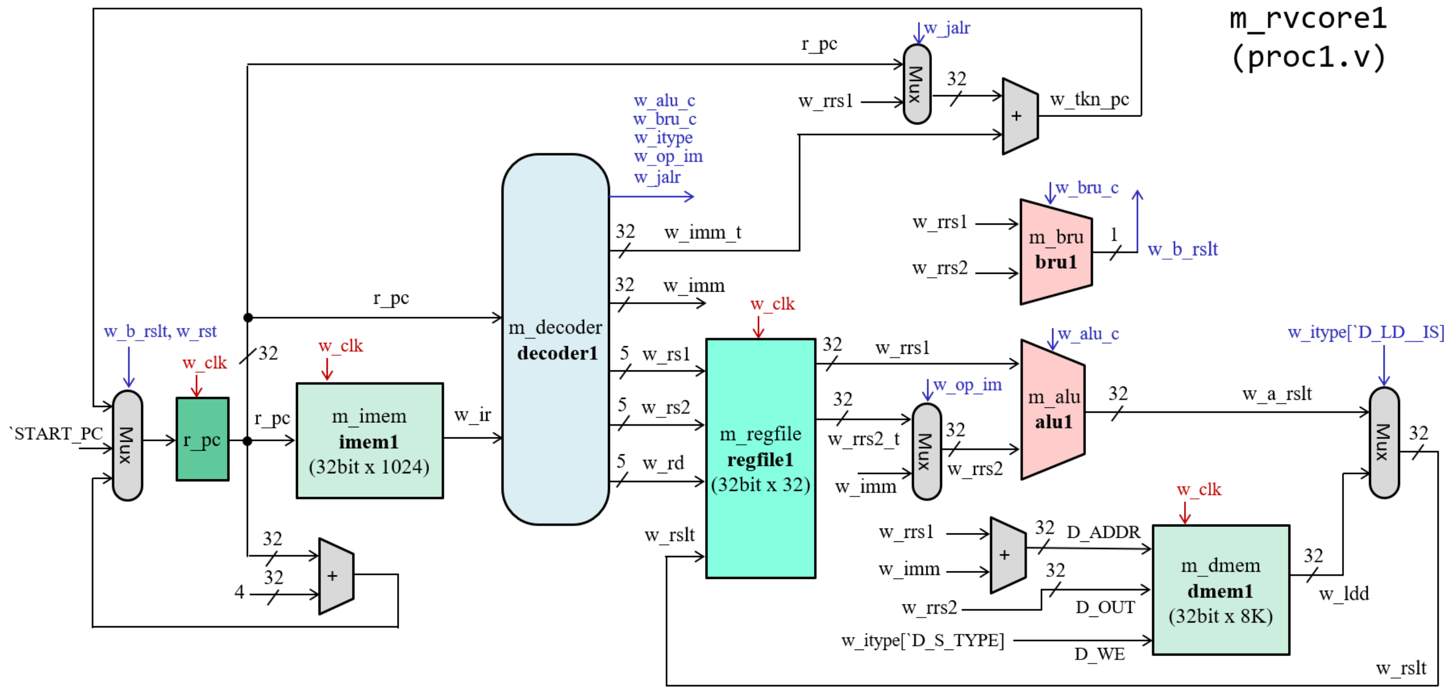
Kenji Kise, Department of Computer Science
kise[at]comp.isct.ac.jp

Typical five steps in processing an instruction

- IF: Instruction Fetch
  fetch an instruction from instruction memory or instruction cache
- ID: Instruction Decode
  decode an instruction and read input operands from register file
- EX: Execution
  perform operation, calculate an address of lw/sw
- MEM: Memory Access
  access data memory or data cache for lw/sw
- WB: Write Back
  write operation result and loaded data to register file

Single-cycle implementation of processors

- Single-cycle implementation also called single clock cycle implementation is the implementation in which an instruction is executed in one clock cycle.
  While easy to understand, it is too slow to be practical.
  It is useful as a baseline for lectures.



Exercise 1

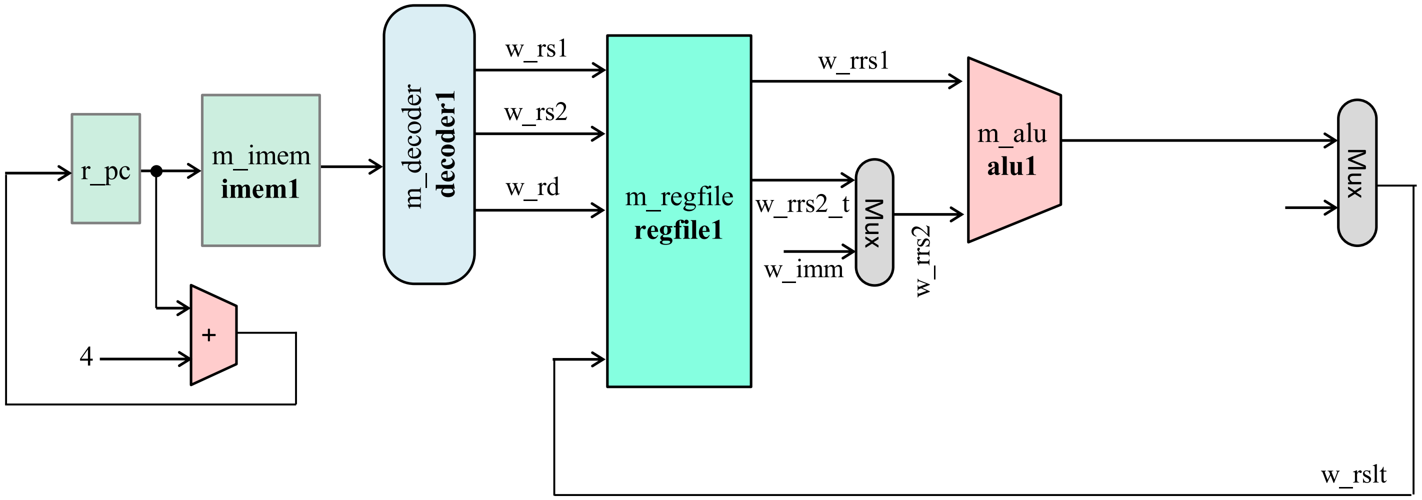
- Draw the main **datapath** of the processor m_rvcore and write the bit-width and valid values on wires when the processor is executing the third instruction

```
0x00  addi x1, x0, 3     # x1 = 3
0x04  addi x2, x1, 4     # x2 = 3 + 4 = 7
0x08  add  x5, x1, x2    # x5 = 3 + 7 = 10
```

add x5, x1, x2

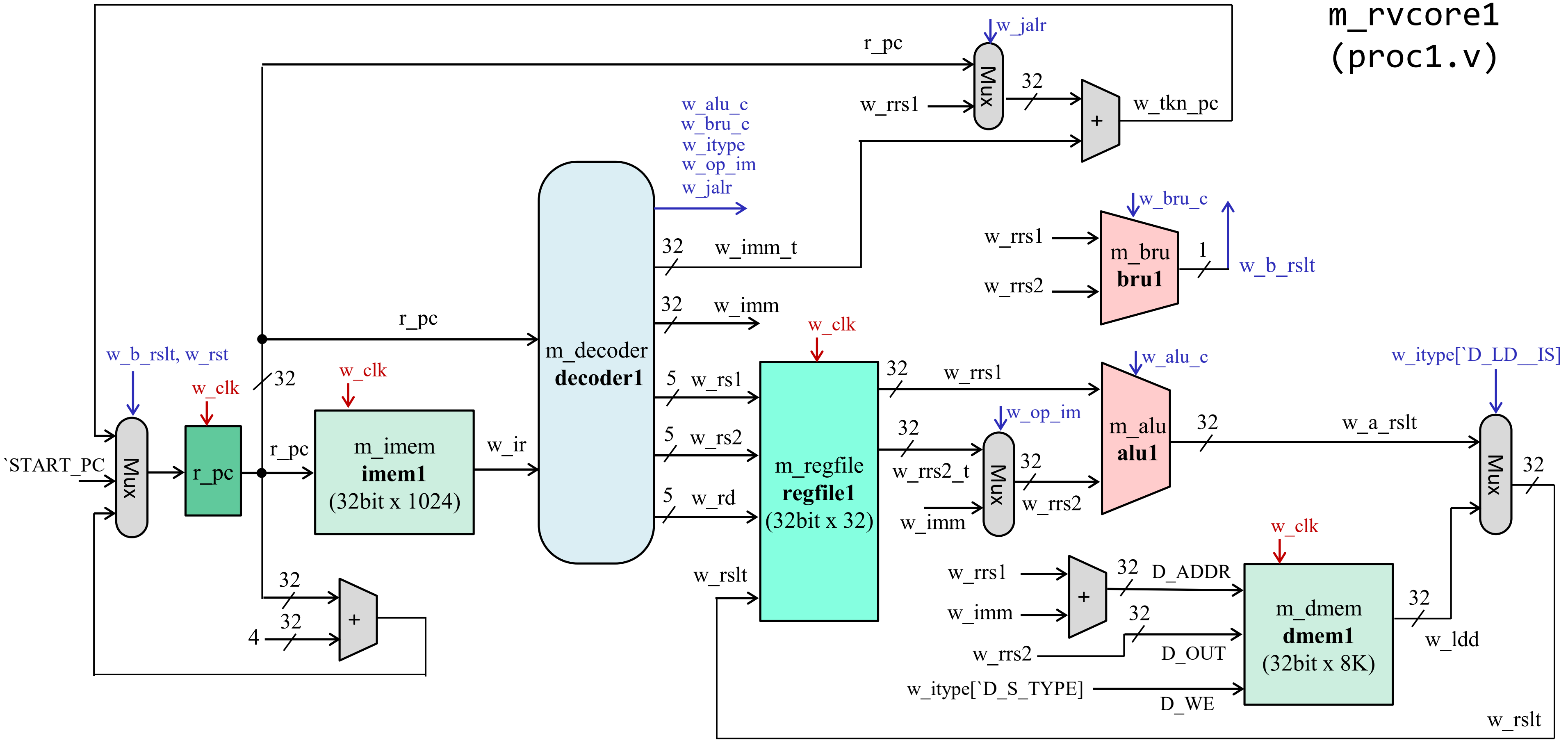
funct7	rs2	rs1	funct3	rd	opcode

R-type



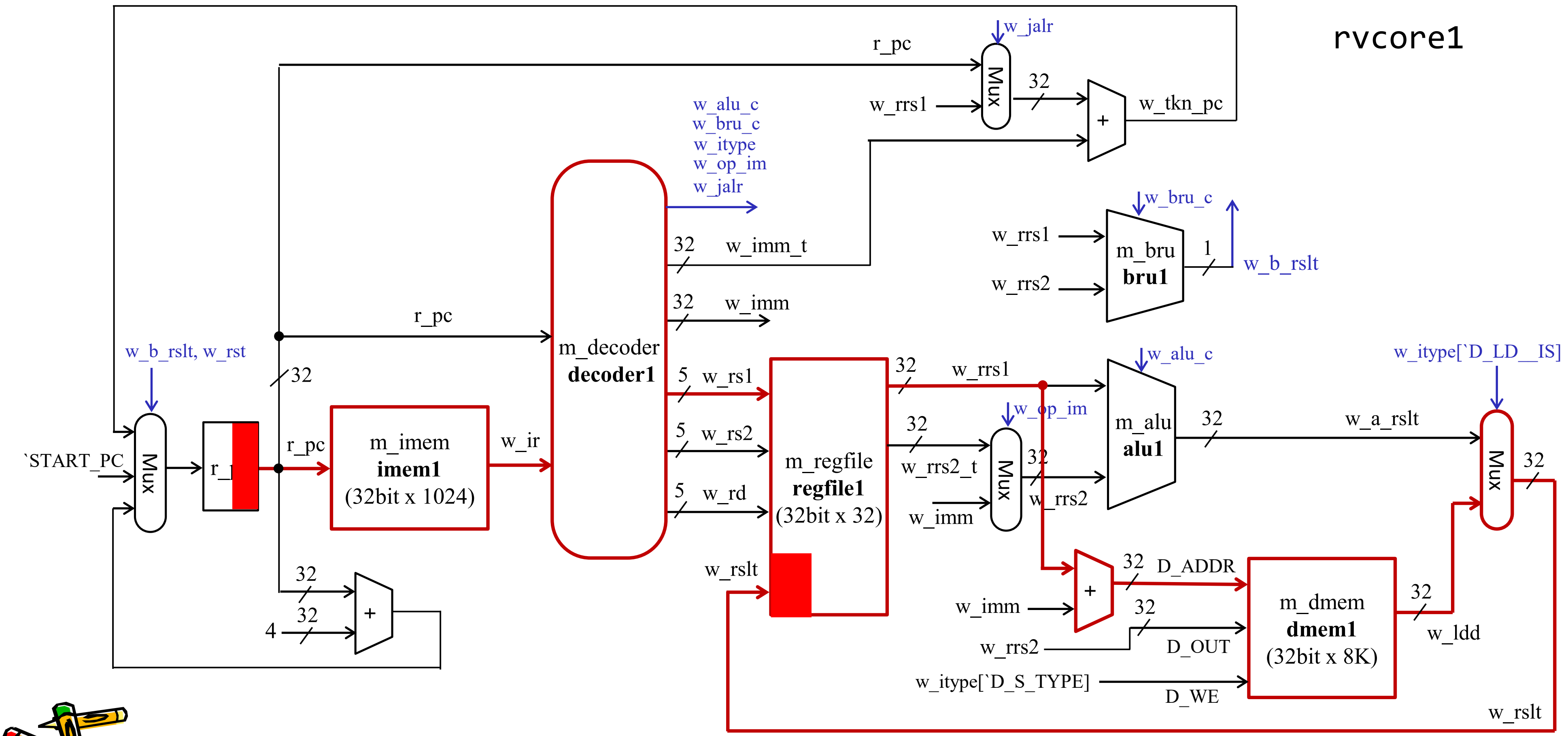
m_rvcore (RV32I, single-cycle processor)

- around 40MHz operating frequency for Arty A7 FPGA board
- lb, lbu, lh, lhu, sb, sh are not supported



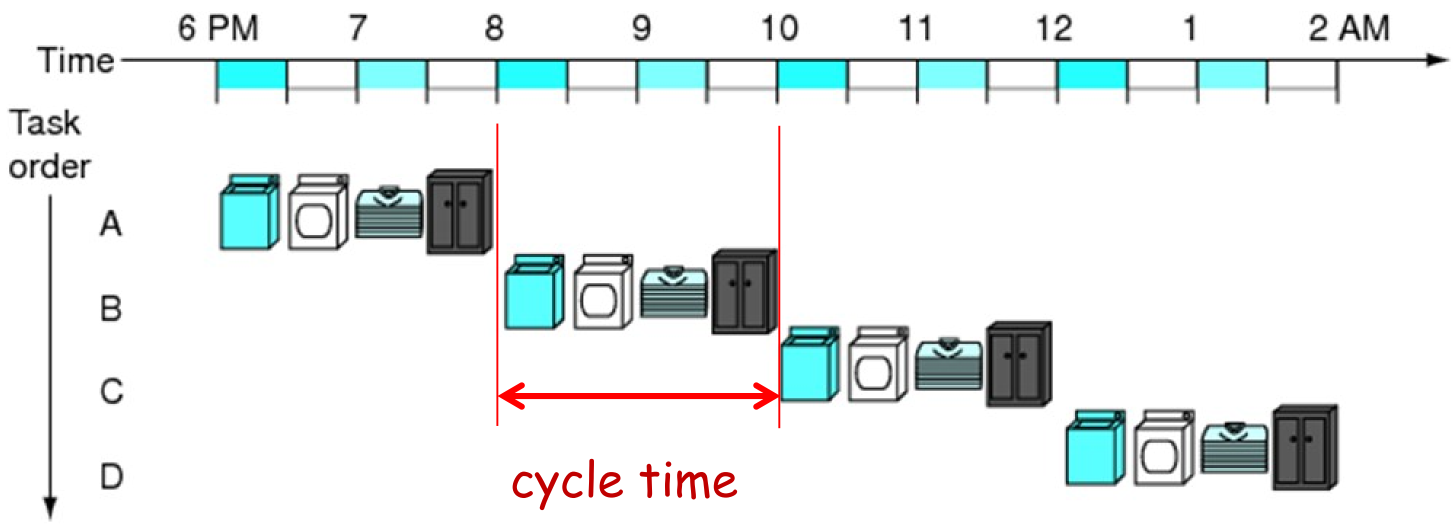
Critical path of rvcore1 (single-cycle version)

- The critical path is defined as the path between a source register (or memory) and a destination register with the maximum delay.
- The path for Load Word instruction like `lw x5, 8(x7)`



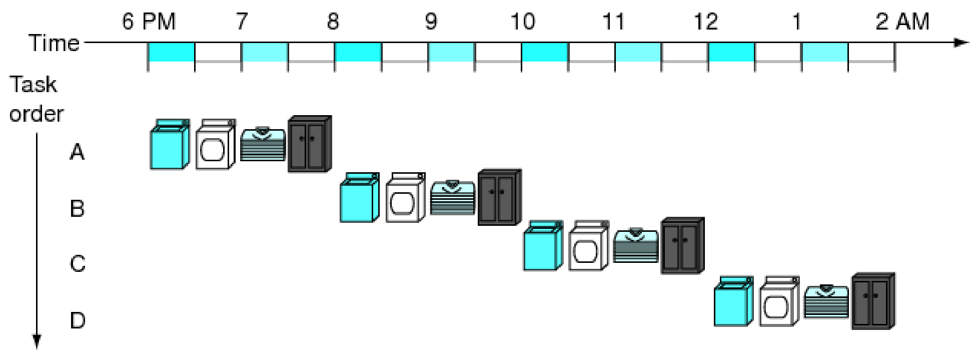
Single-cycle implementation of laundry

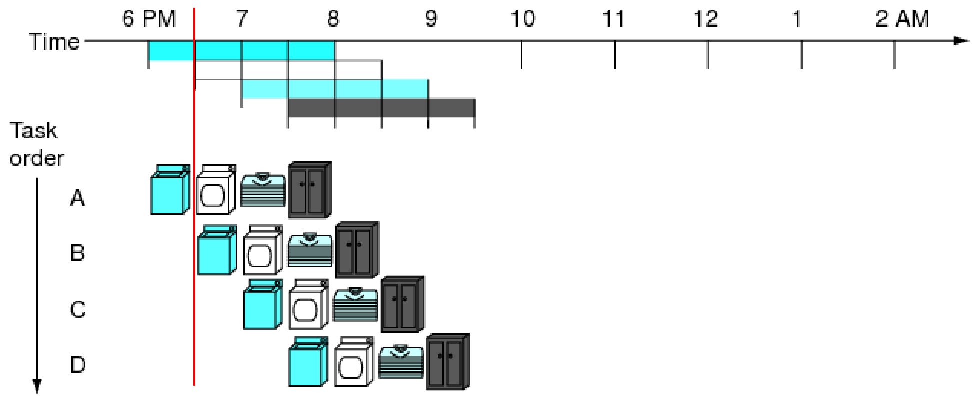
- (A) Ann, (B) Brian, (C) Cathy, and (D) Don each have dirty clothes to be *washed*, *dried*, *folded*, and *put away*, each taking 30 minutes.
- The cycle time (the time from the end of one load to the end of the next one) is 2 hours.
- For four loads, the sequential laundry takes 8 hours.



Single-cycle implementation and pipelining

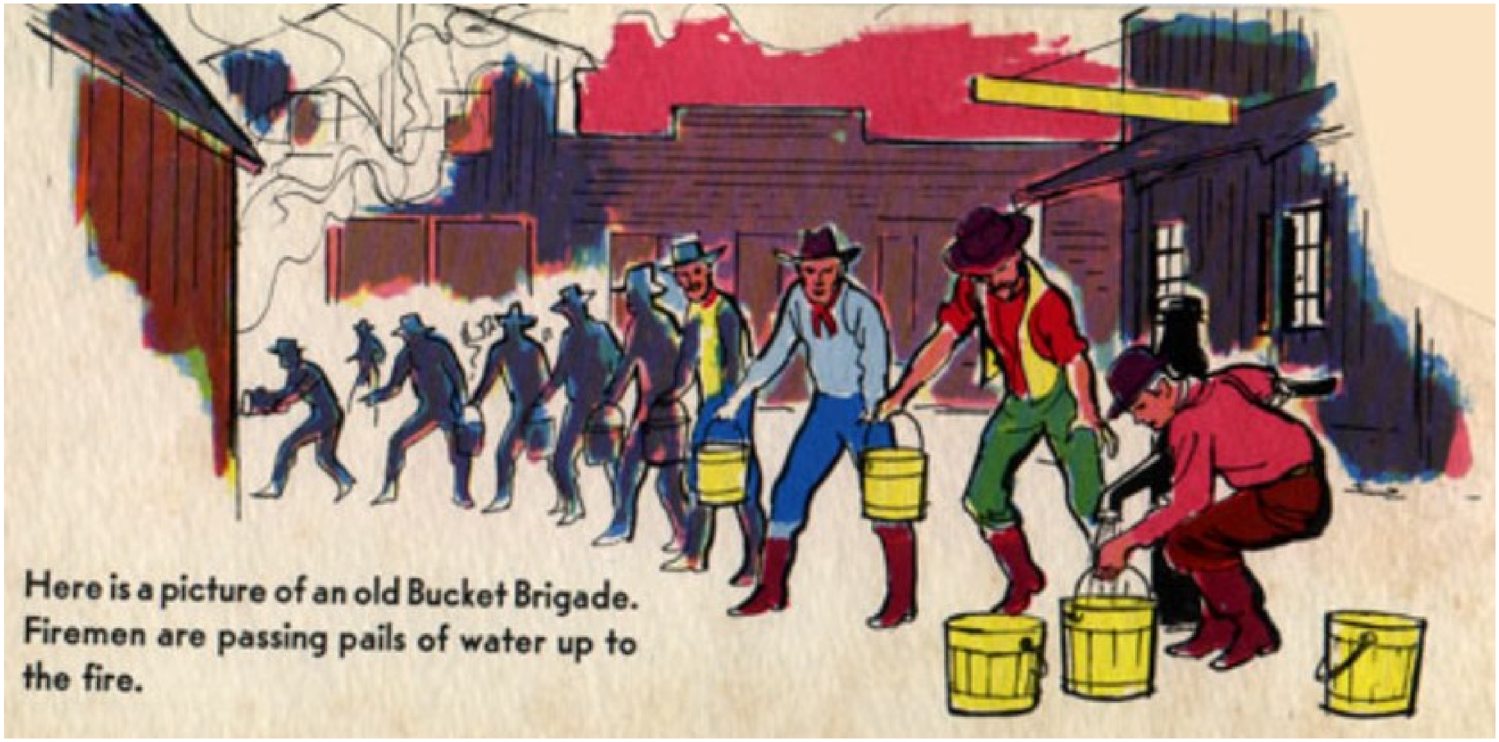
- When the washing of load A is finished at 6:30 p.m., another washing of load B starts.
- Pipelined laundry takes 3.5 hours just using the same hardware resources. The cycle time is 30 minutes.
- What is the cycle time (latency) of each load?





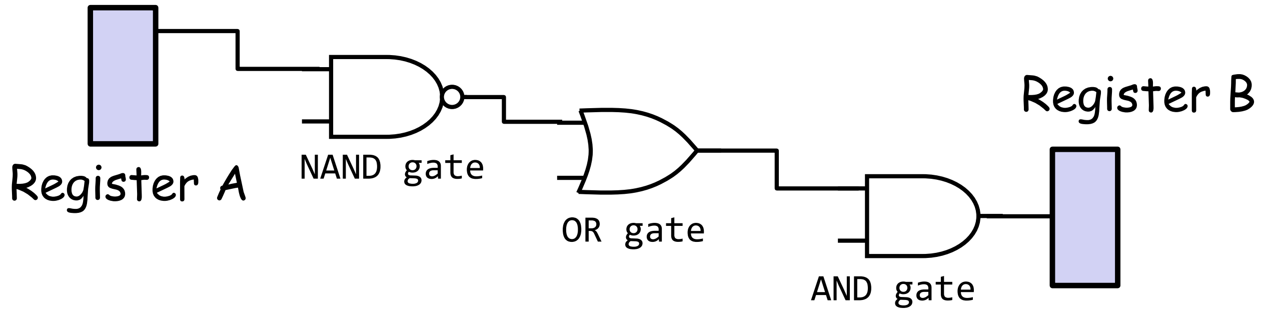
pipeline diagram

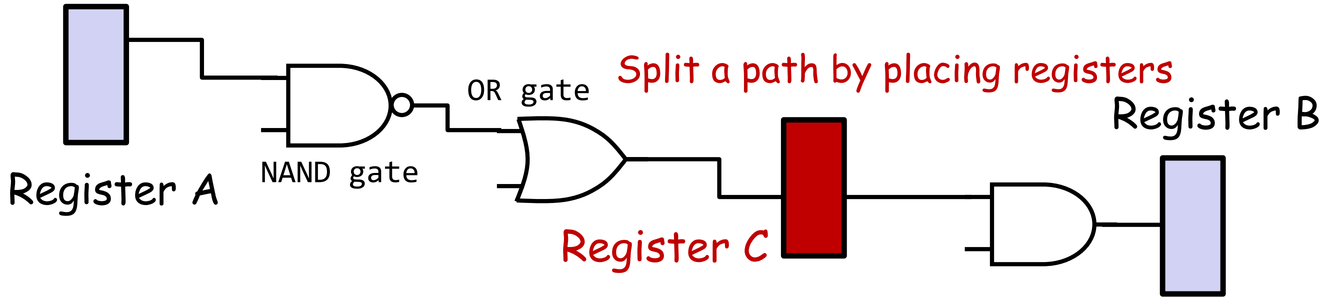
Bucket brigade


Here is a picture of an old Bucket Brigade.
Firemen are passing pails of water up to the fire.

Clock rate is mainly determined by

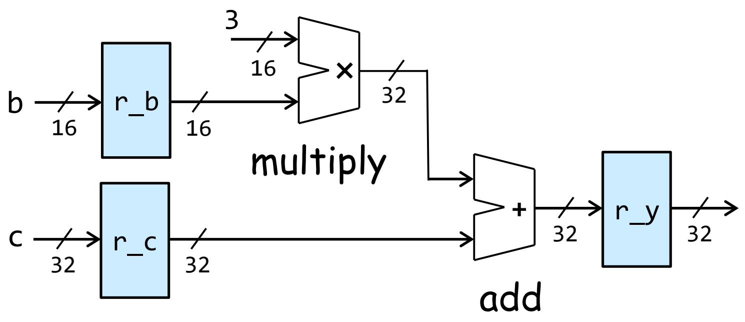
- Switching speed of gates (transistors)
- The number of levels of gates
  - The maximum number of gates cascaded in series in any combinational logics.
  - In this example, the number of levels of gates is 3.
- Wiring delay and fanout

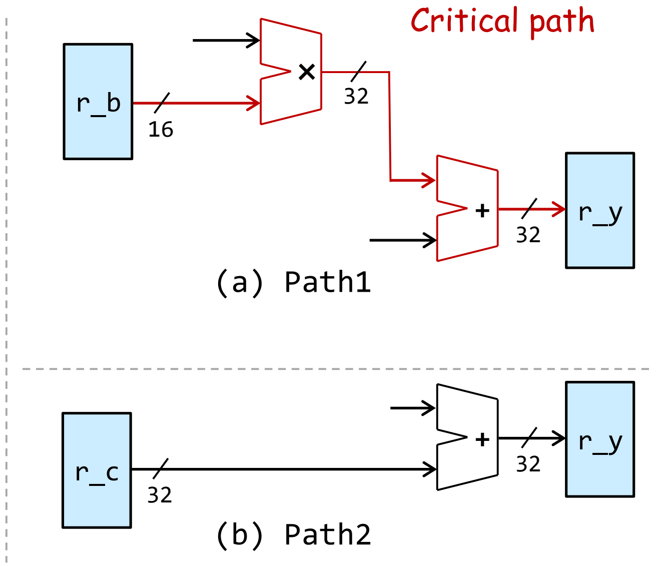




Pipelining example: multiply-add operation (1)

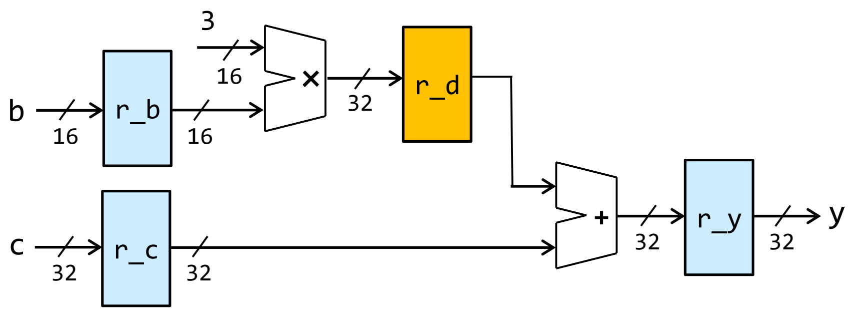
- As an example of pipelining, we will see a multiply-add circuit.
- r_b, r_c are input registers and r_y is output register of the circuit.
- This has two paths named path1 and path2, and path1 is the critical path to determine the maximum operating frequency.

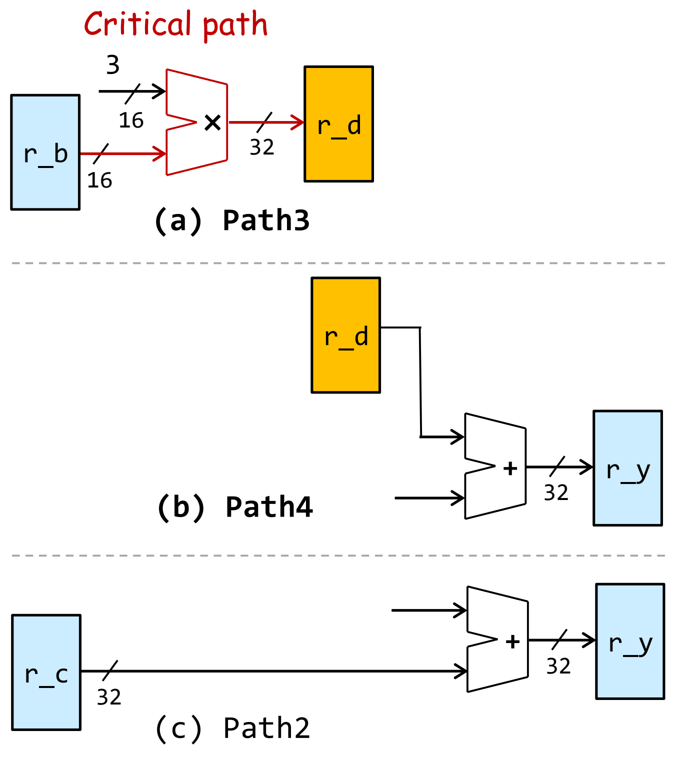




Pipelining example: multiply-add operation (2)

- 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.





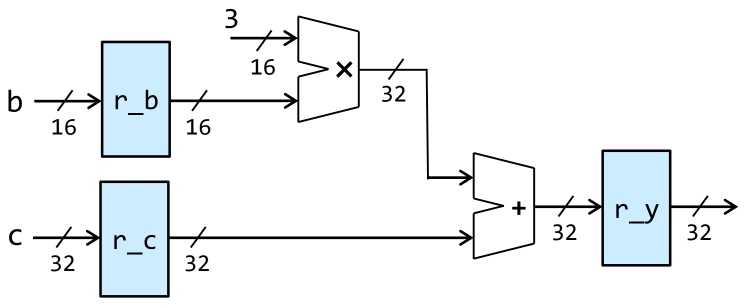
(a) Path3

(b) Path4

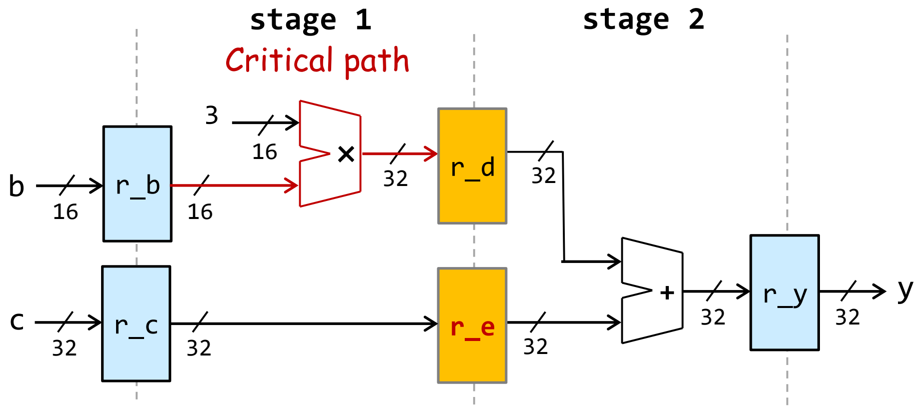
(c) Path2

Pipelining example: multiply-add operation (3)

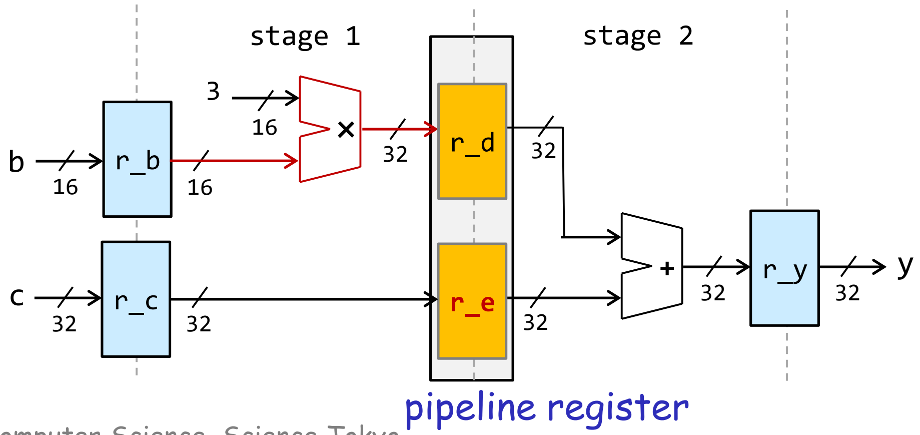
- To overcome this drawback, we insert register r_e.
- This realizes a pipeline with stages 1 and 2. A set of registers between two adjacent stages are called a pipeline register.



(a) original multiply-add circuit

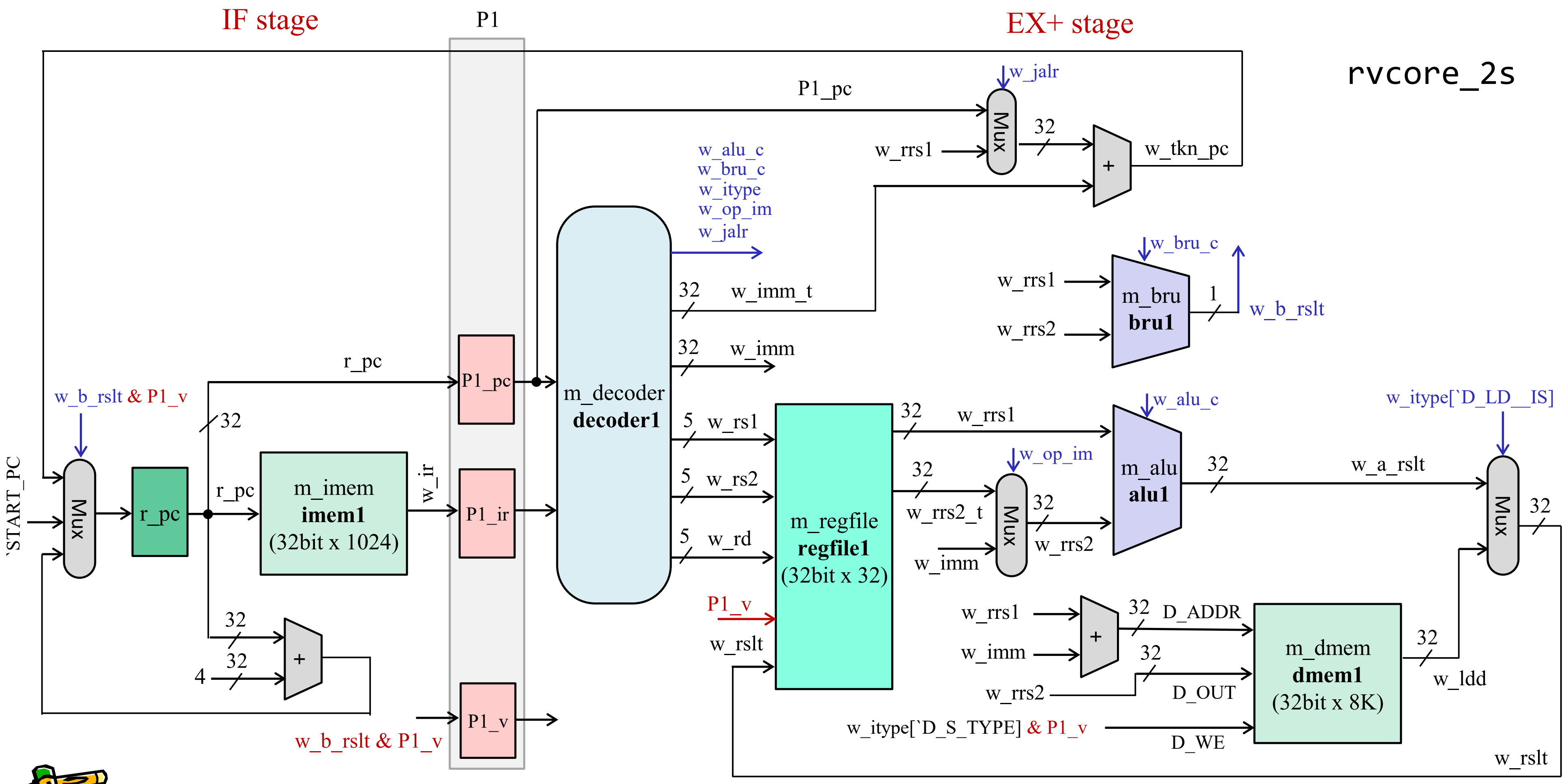


(b) two-stage pipelined circuit

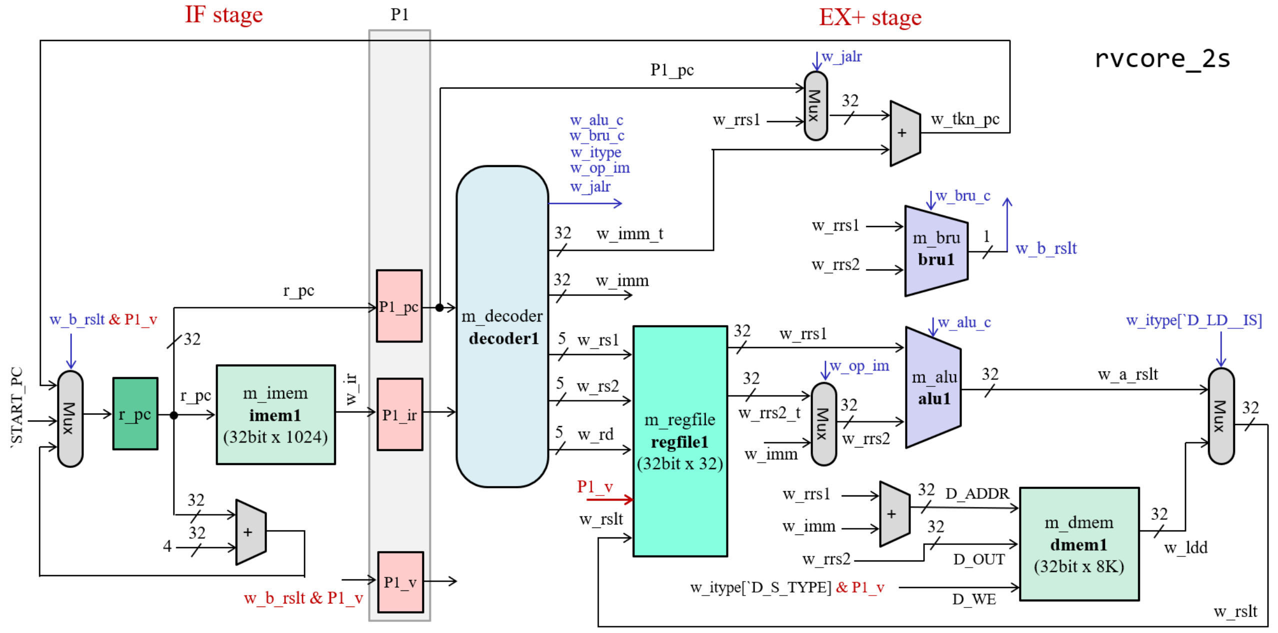


rvcore_2s : 2-stage pipelining processor

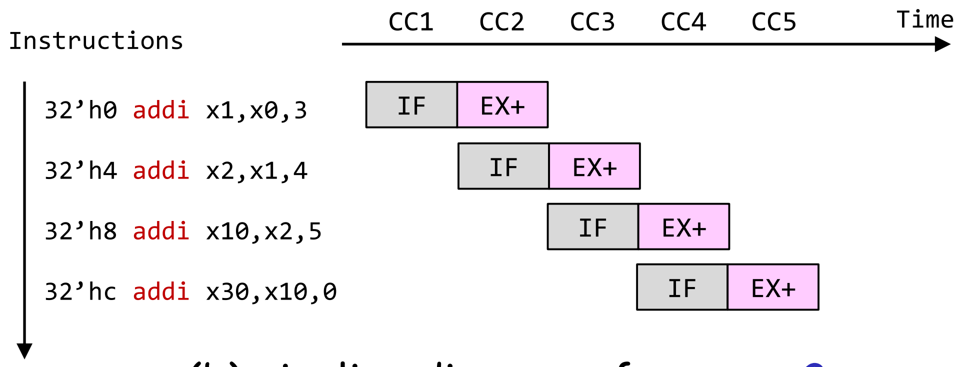
- The strategy is to separate the instruction fetch (IF) step and other (ID, EX, MA, WB) steps. The first stage is named IF. The other stage is named EX+.



rvcore_2s : 2-stage pipelining processor



(a) rvcore_2s: 2-stage pipelining processor

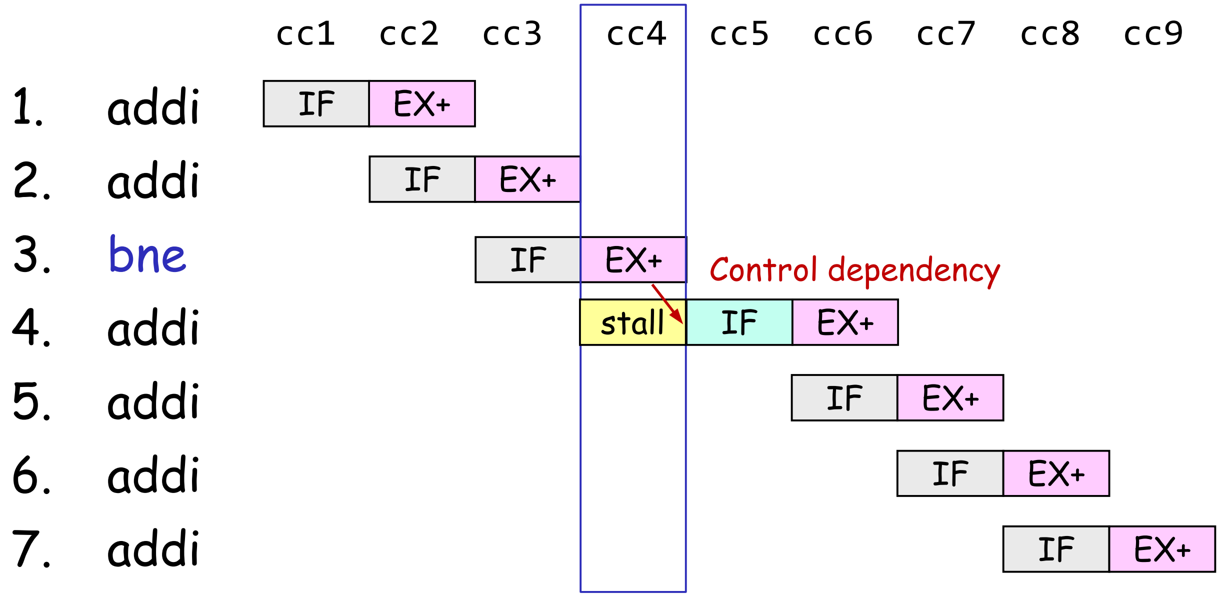


```
32’h0 addi x1,x0,3
32’h4 addi x2,x1,4
32’h8 addi x10,x2,5
32’hc addi x30,x10,0
```

(b) pipeline diagram of rvcore_2s

Why do branch instructions degrade IPC?

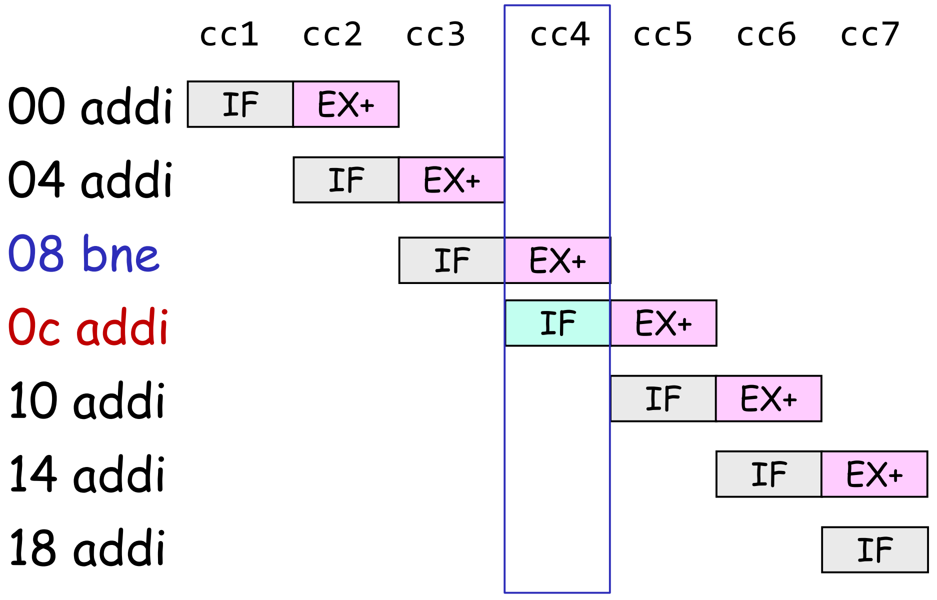
- The branch taken / untaken (branch result) is determined in the EX+ stage of the branch.
- The conservative approach is stalling instruction fetch until the branch direction is determined.



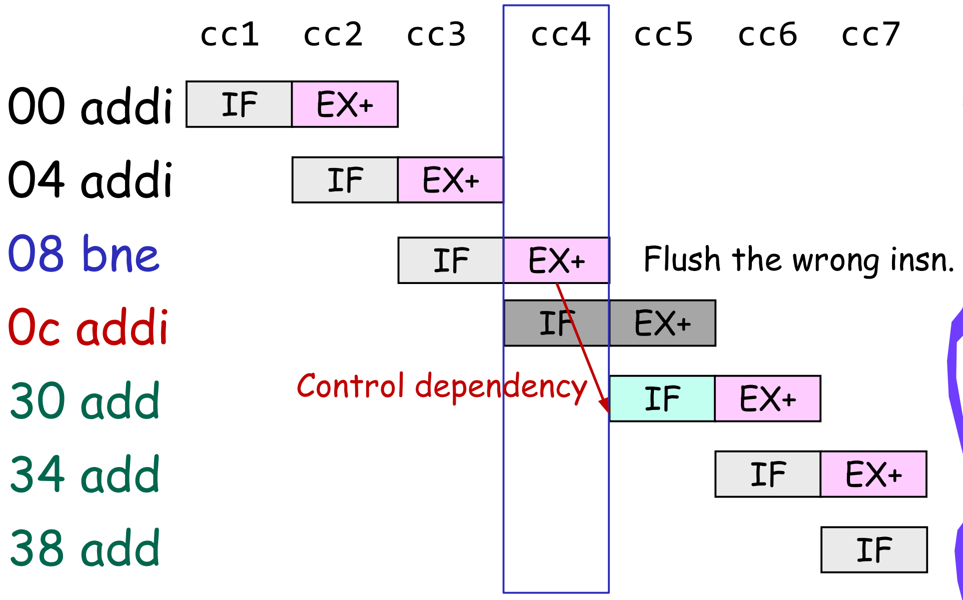
two-stage pipelining processor executing instruction sequence with a branch (bne)

Why do branch instructions degrade IPC?

- Another approach is fetching the following instruction (an instruction at the next address) when a branch (bne) is fetched.
- When a branch (08 bne) is taken to address 0x30, the wrong instruction fetched (0c addi) must be flushed.



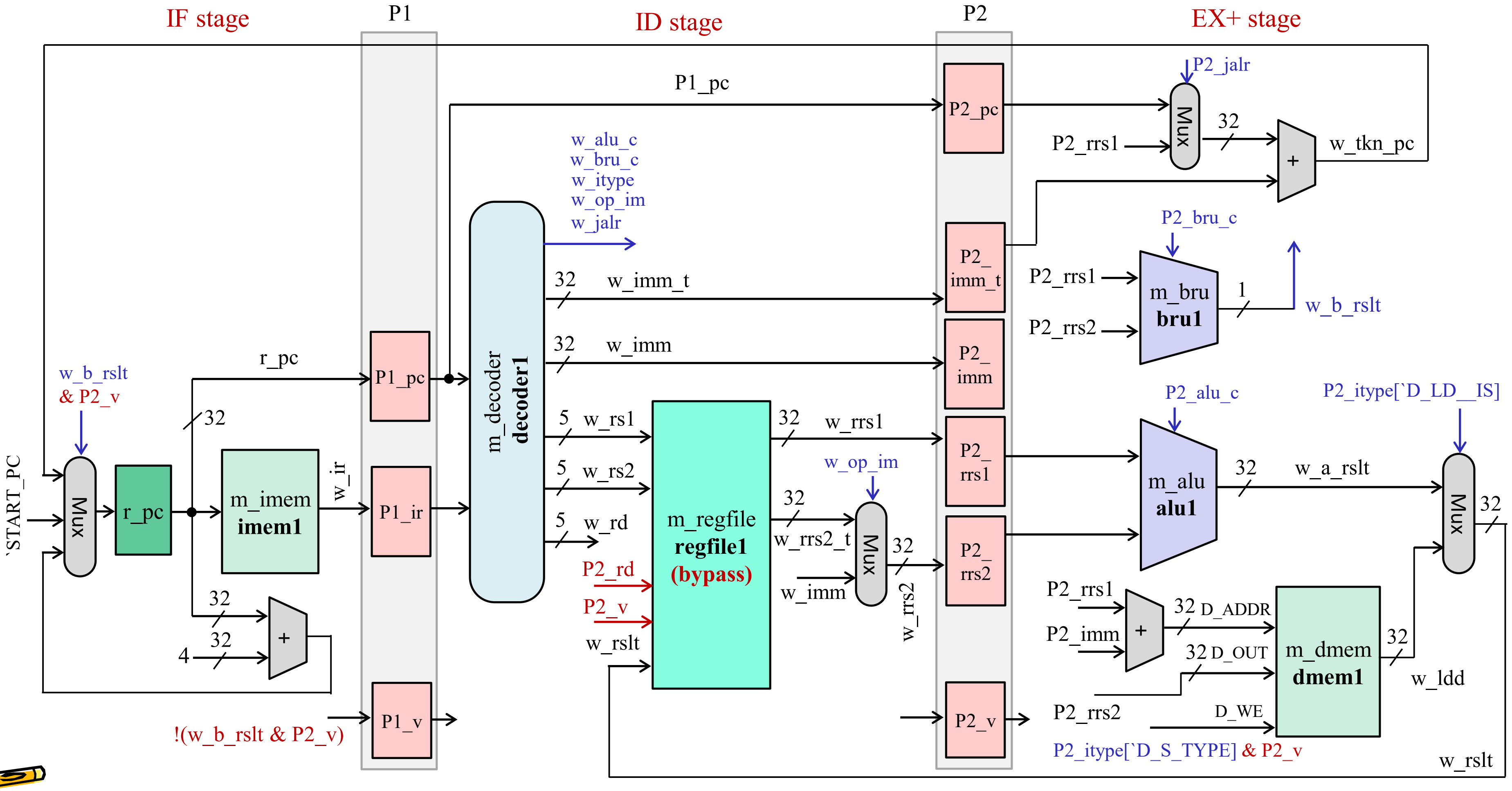
(a) branch untaken case



(b) branch taken case

rvcore_3s : 3-stage pipelining processor

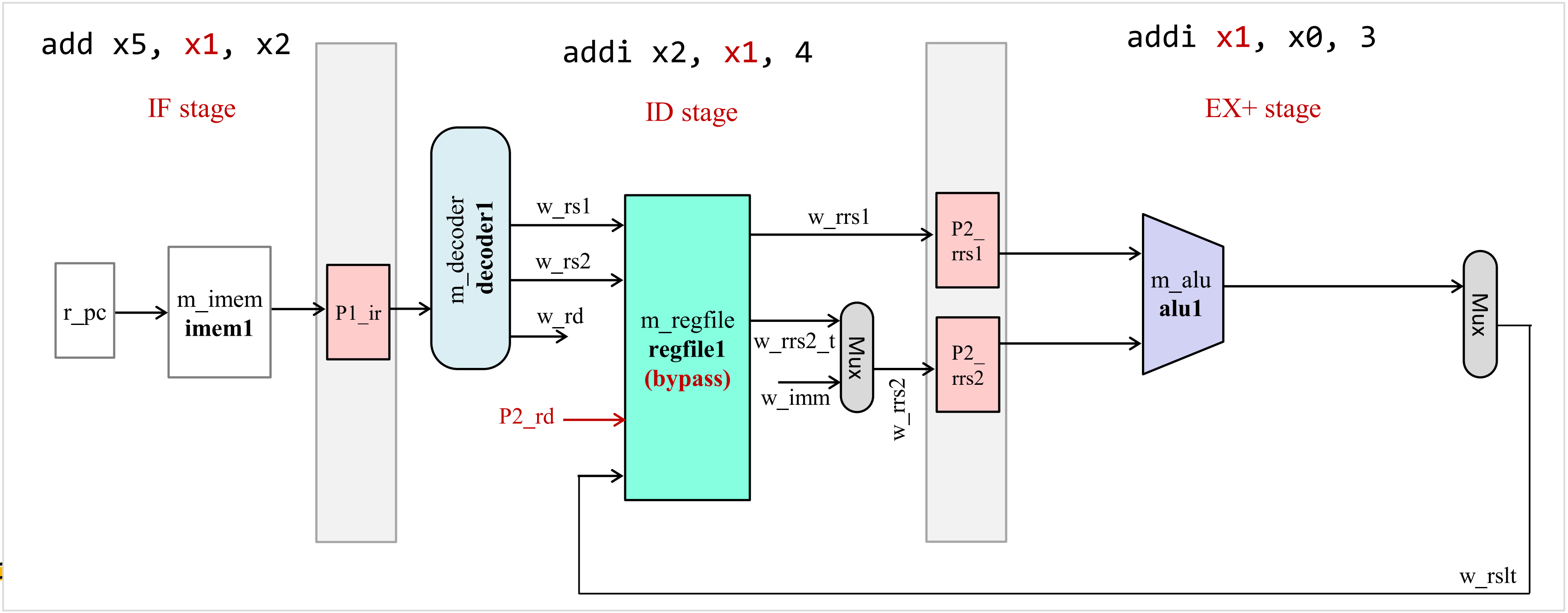
- The strategy is to separate the instruction fetch (IF) step, instruction decode (ID) step, and other (EX, MA, WB) steps. The first stage is named IF. The second stage is named ID. The last stage is named EX+.



Exercise 2

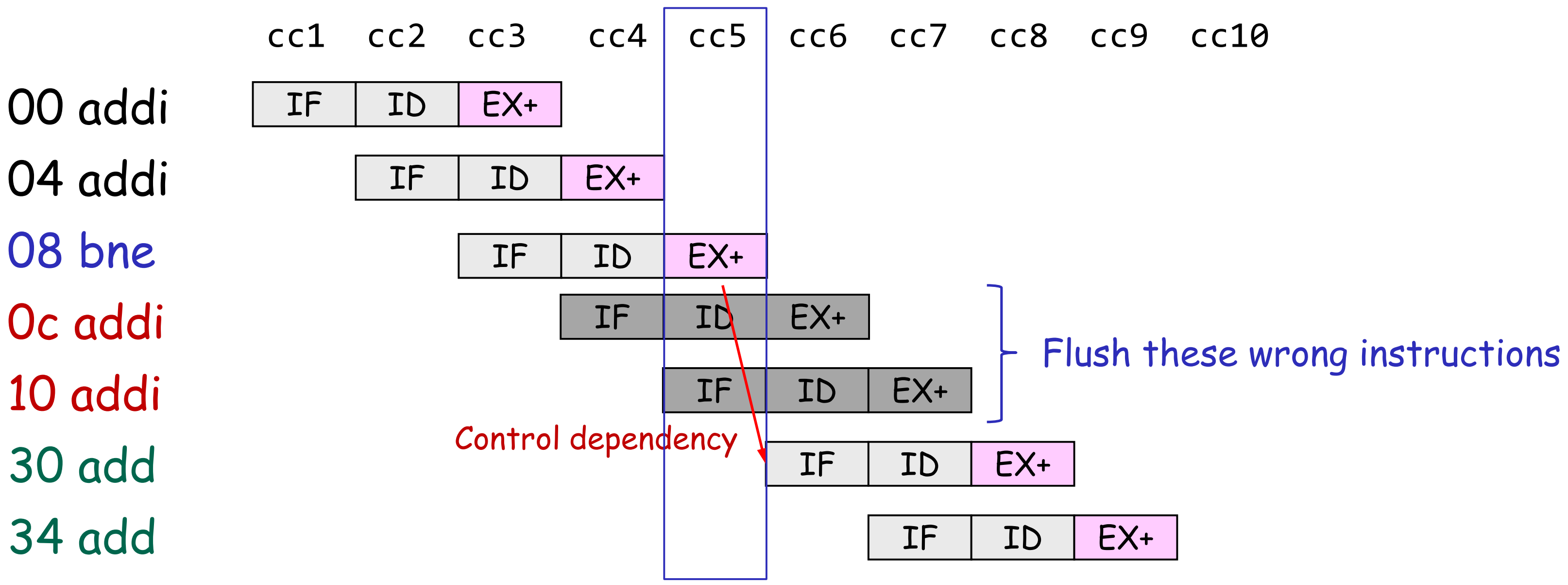
- Draw the main datapath of the processor rvcore_3s and write the valid values on wires when the processor is executing these three instructions

```
0x00  addi x1, x0, 3      # x1 = 3
0x04  addi x2, x1, 4      # x2 = 3 + 4 = 7
0x08  add  x5, x1, x2     # x5 = 3 + 7 = 10
```



Why do branch instructions degrade IPC?

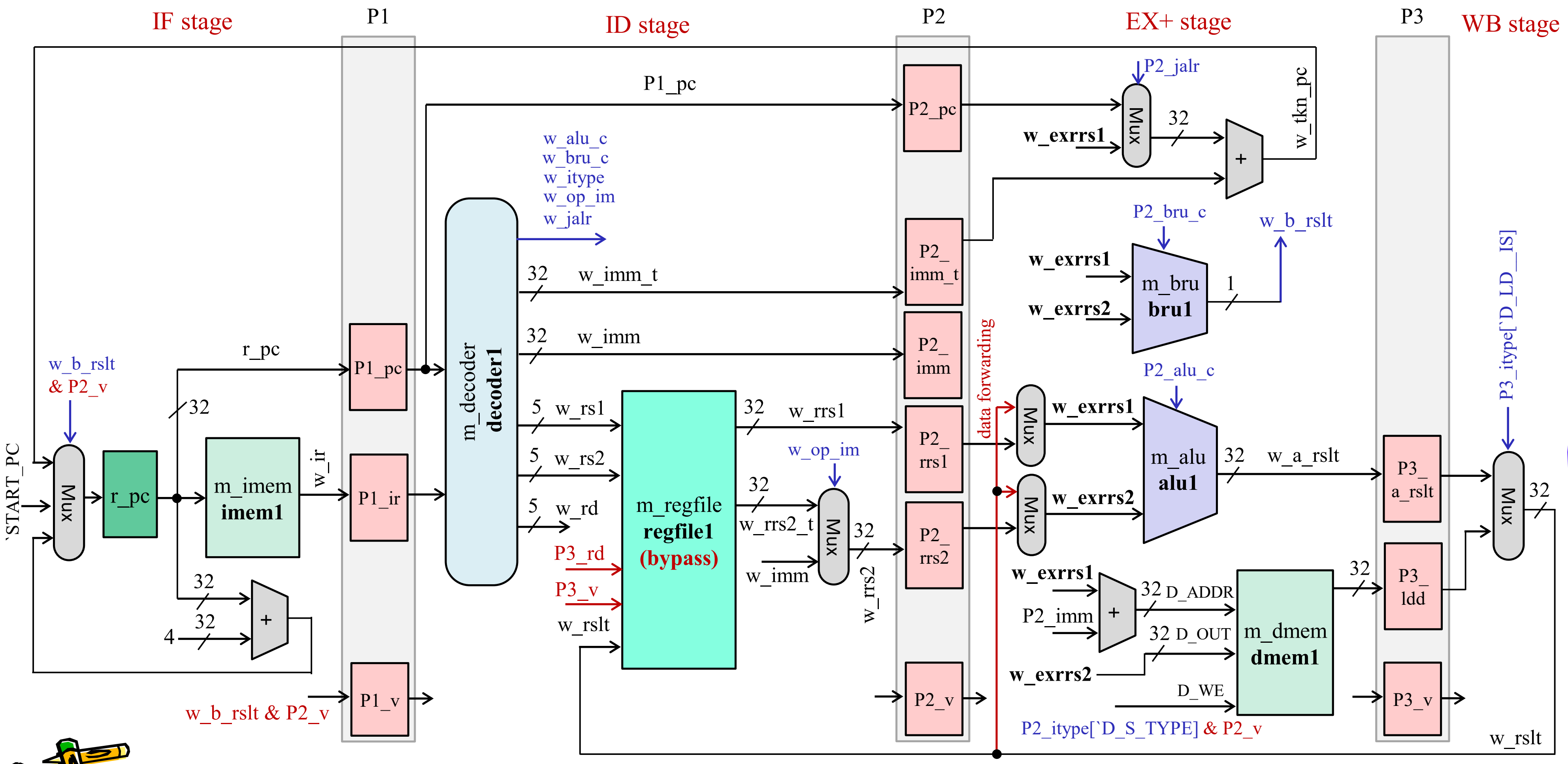
- Another approach is fetching the following instructions (0c addi, 10 addi) after a branch (bne) is fetched.
- When a branch (08 bne) is taken, the wrong instructions fetched (0c addi, 10 addi) must be flushed.



three-stage pipelining processor executing instruction sequence with a taken branch

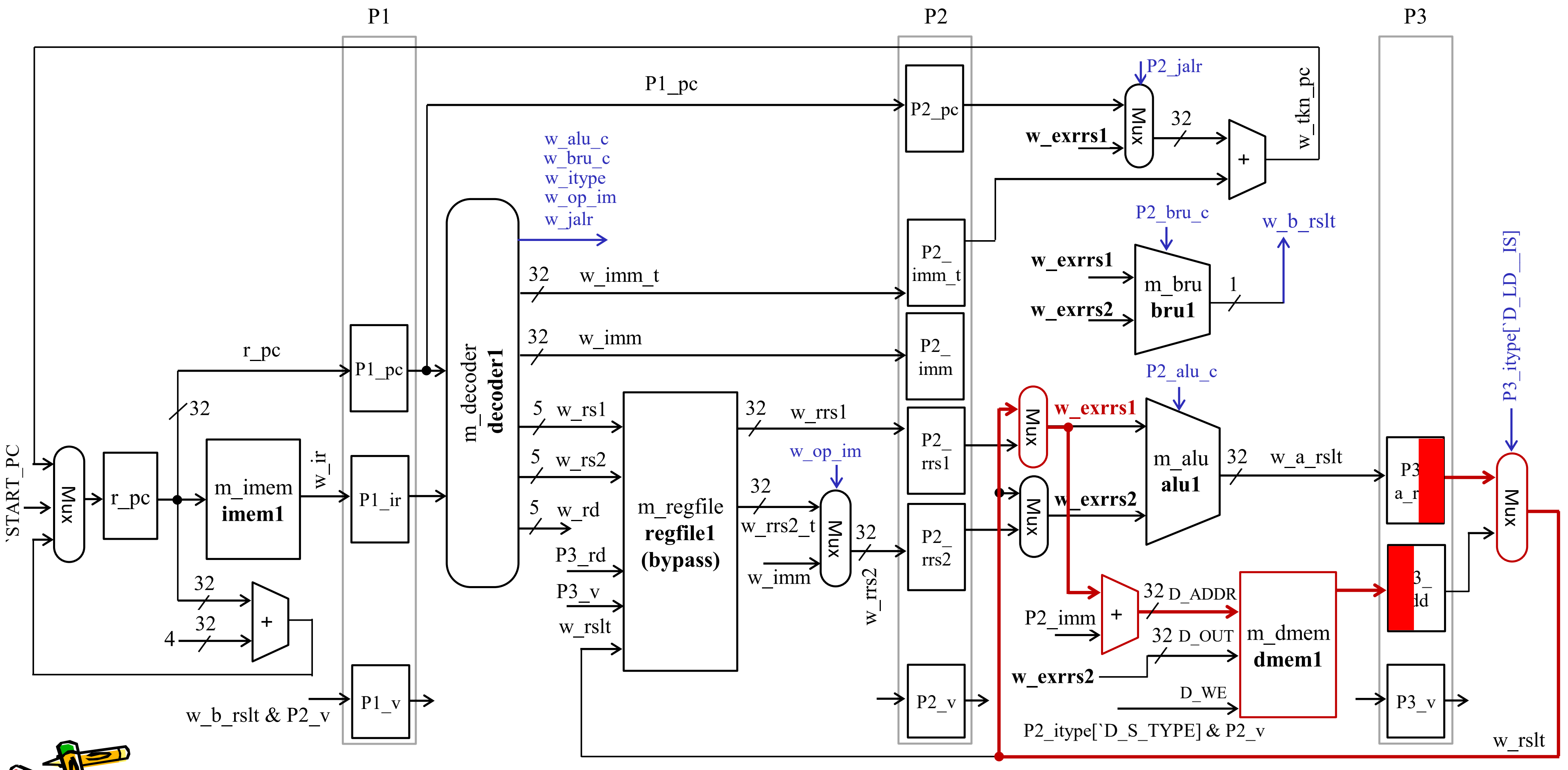
rvcore_4s : 4-stage pipelining processor

- The strategy is to separate the instruction fetch (IF) step, instruction decode (ID) step, and write back(WB) step, and other (EX, MA) steps. The first stage is named IF. The second stage is named ID. The third stage is named EX+. The last stage is named WB.



rvcore_4s : 4-stage pipelining processor

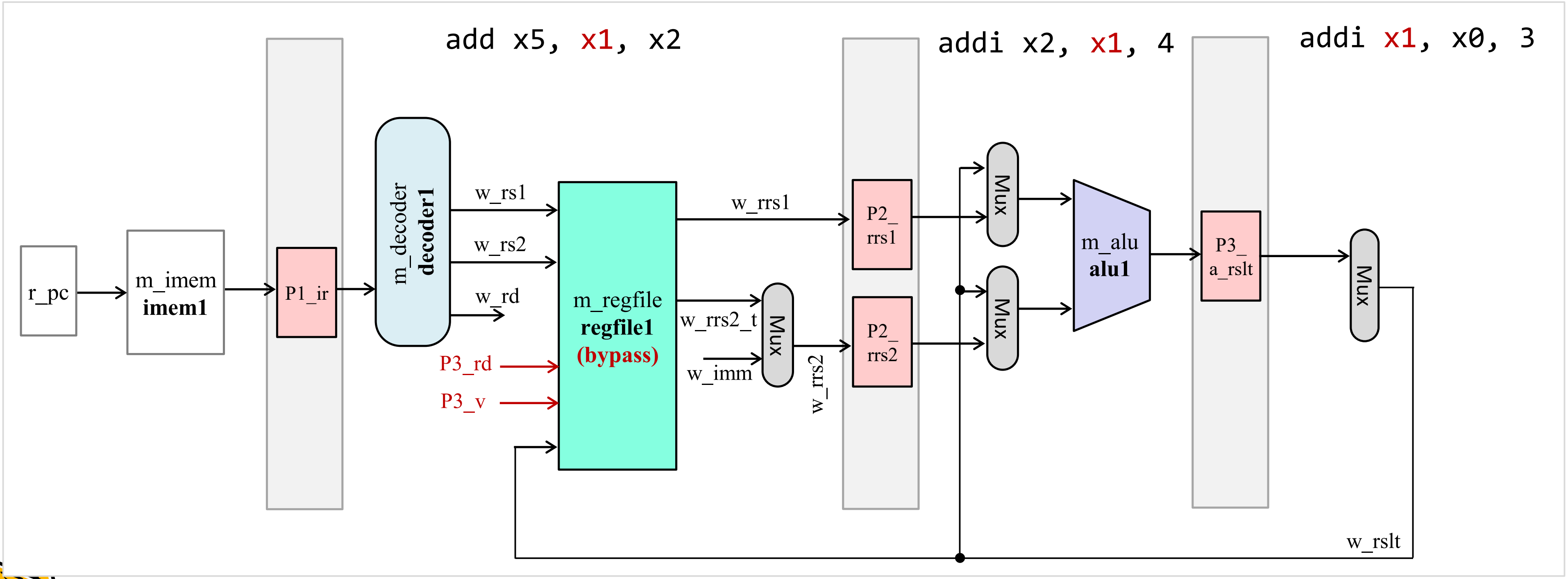
- Critical path
  - deley(register read) + delay(mux) + delay(mux) + delay(adder) + delay(dmem read)



Exercise 3

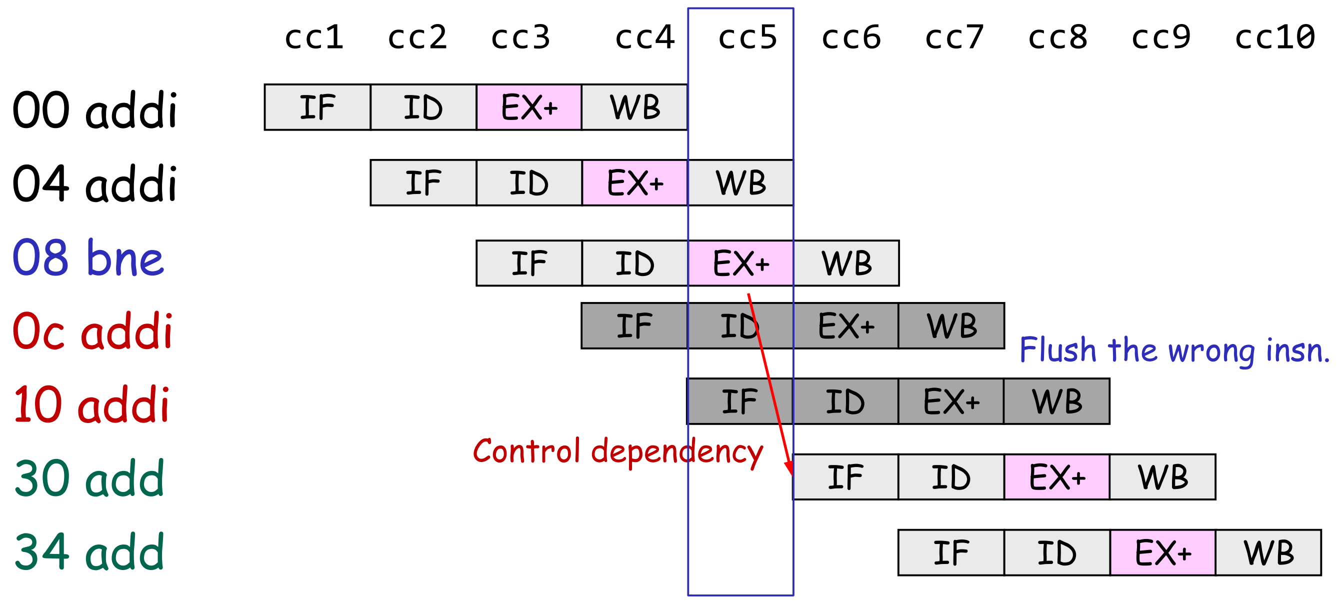
- Draw the main datapath of the processor rvcore_4s and write the valid values on wires when the processor is executing these three instructions in ID, EX+, and WB stages

```
0x00  addi x1, x0, 3     # x1 = 3
0x04  addi x2, x1, 4     # x2 = 3 + 4 = 7
0x08  add  x5, x1, x2    # x5 = 3 + 7 = 10
```



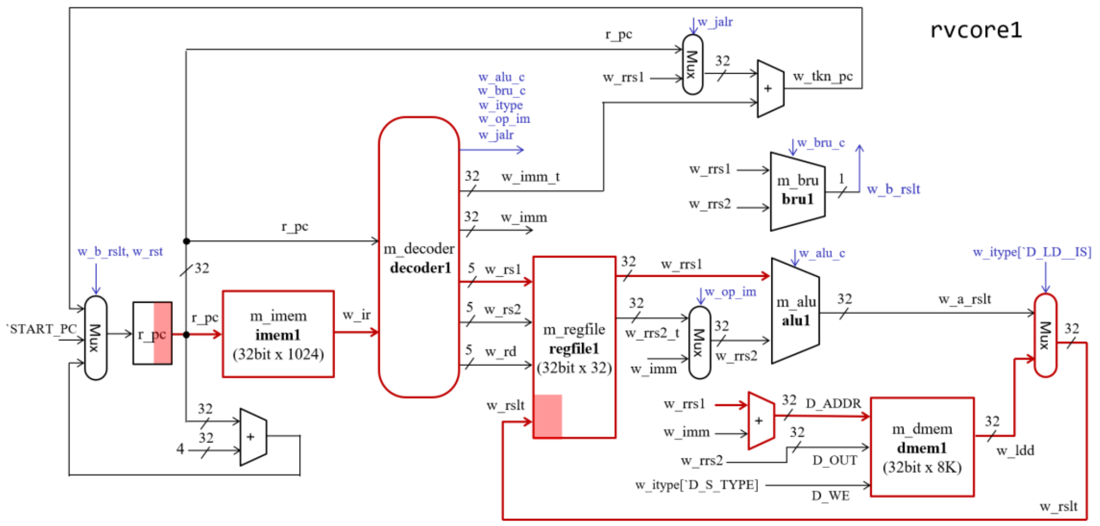
Why do branch instructions degrade IPC?

- Another approach is fetching the following instructions (0c addi, 10 addi) after a branch (bne) is fetched.
- When a branch (08 bne) is taken, the wrong instructions fetched (0c addi, 10 addi) are flushed.



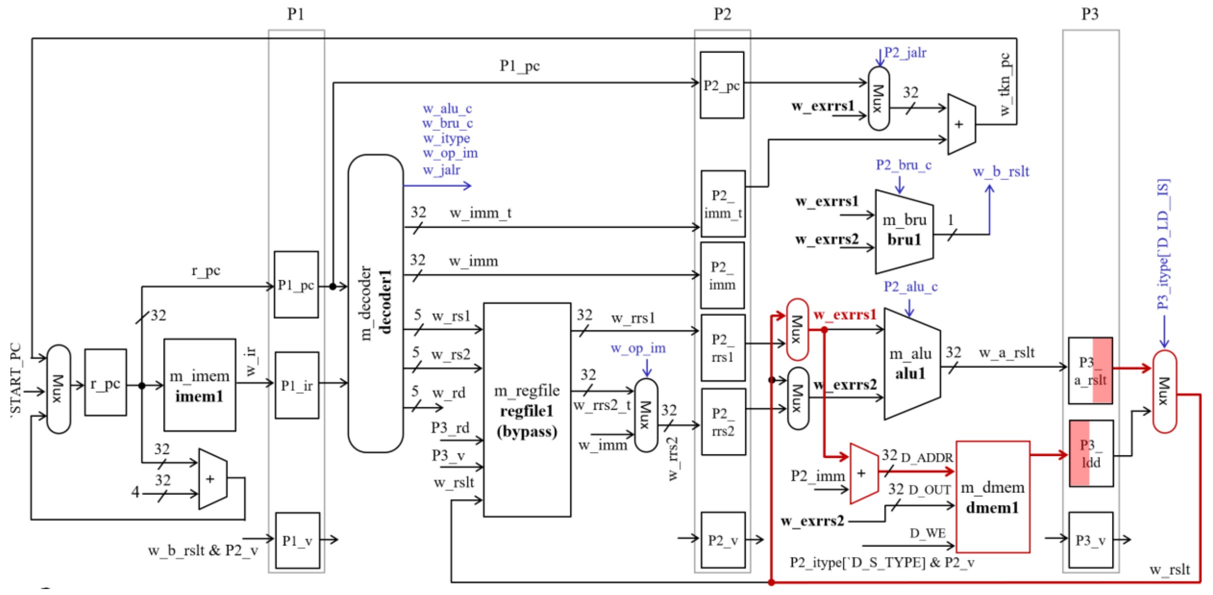
four-stage pipelining processor executing instruction sequence with a taken branch

Comparison of critical path between rvcore1 and rvcore_4s



(a) the critical path of rvcore_1s

deley(register read) + delay(imem read) + delay(decode) + delay(regfile read) + delay(adder) + delay(dmem read) + delay(mux)

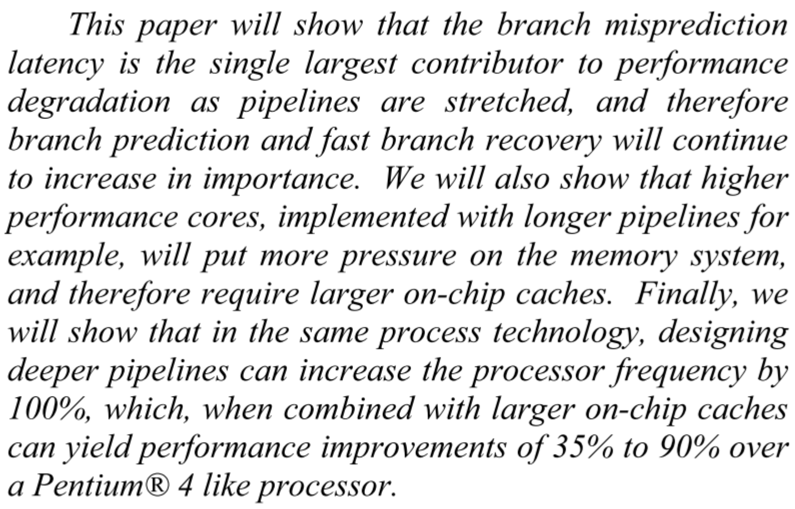


(b) the critical path of rvcore_4s

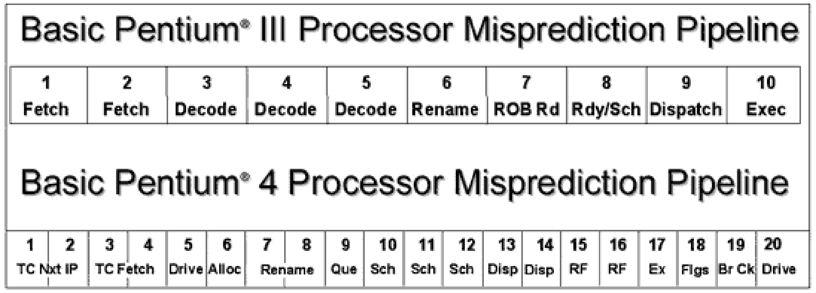
deley(register read) + delay(mux) + delay(mux) + delay(adder) + delay(dmem read)

Recommended Reading

- Increasing Processor Performance by Implementing Deeper Pipelines
  - Eric Sprangle , Doug Carmean (Intel Corporation)
  - ISCA-2002 pp. 25-34 (2002)



This paper will show that the branch misprediction latency is the single largest contributor to performance degradation as pipelines are stretched, and therefore branch prediction and fast branch recovery will continue to increase in importance. We will also show that higher performance cores, implemented with longer pipelines for example, will put more pressure on the memory system, and therefore require larger on-chip caches. Finally, we will show that in the same process technology, designing deeper pipelines can increase the processor frequency by 100%, which, when combined with larger on-chip caches can yield performance improvements of 35% to 90% over a Pentium® 4 like processor.

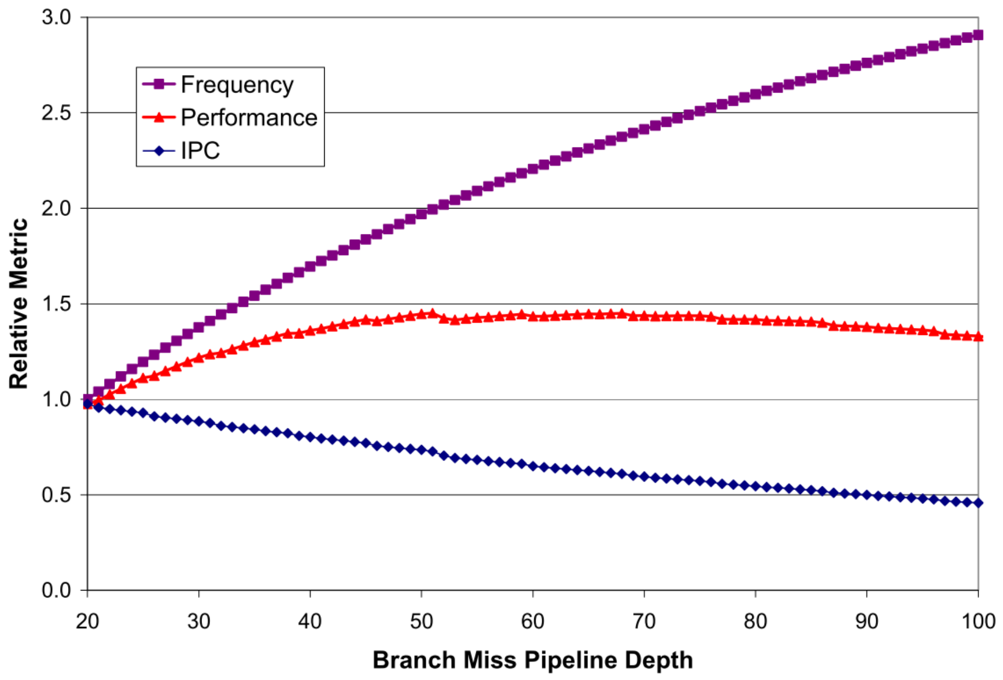


Basic Pentium® III Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

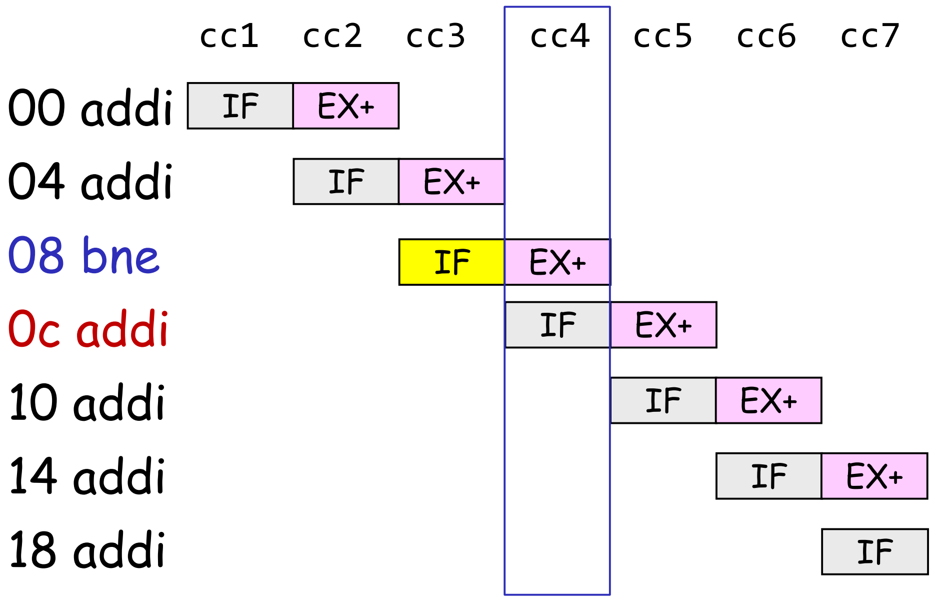
Basic Pentium® 4 Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

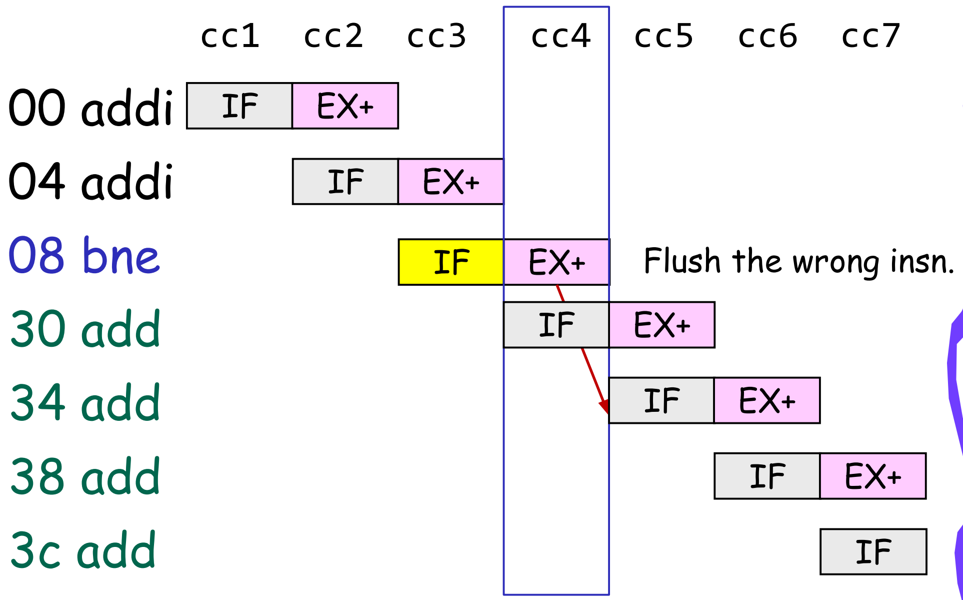


Why do branch instructions degrade IPC?

- Another approach is fetching the instruction with branch prediction when a branch (bne) is fetched.
- Predict the branch outcome (taken / untaken), and taken PC.
- When a preciction is miss, the wrong instructions fetched are flushed.



(a) branch prediction as untaken and hit



(b) branch prediction as taken and hit

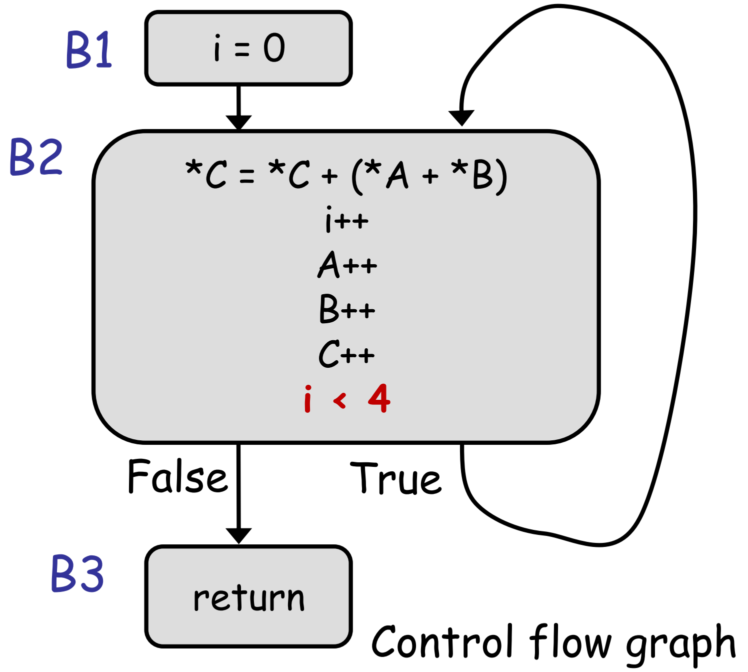
Hardware branch predictor

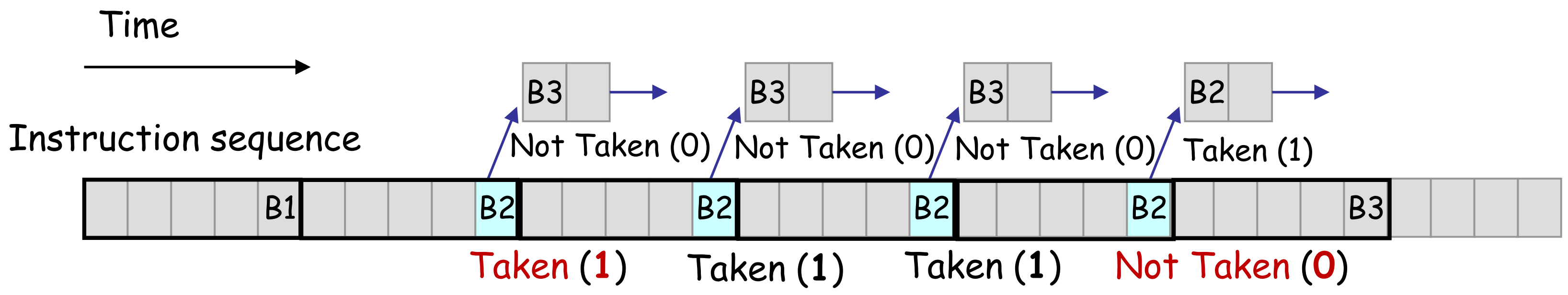
- A branch predictor is a digital circuit that tries to guess or predict which way (taken or untaken) a branch will go before this is known definitively.
  - A random predictor will achieve about a 50% hit rate because the prediction output is 1 (taken) or 0 (untaken).
  - Let's guess the accuracy.
    What is the accuracy of typical branch predictors for high-performance commercial processors?

Sample program: vector add (function v_add)

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

int main(){
  for(int i=0; i<N; i++) v_add(A, B, C);
}
```

Basic block contains a sequence of statement.
The flow of control enters at the beginning of the statement and leave at the end.

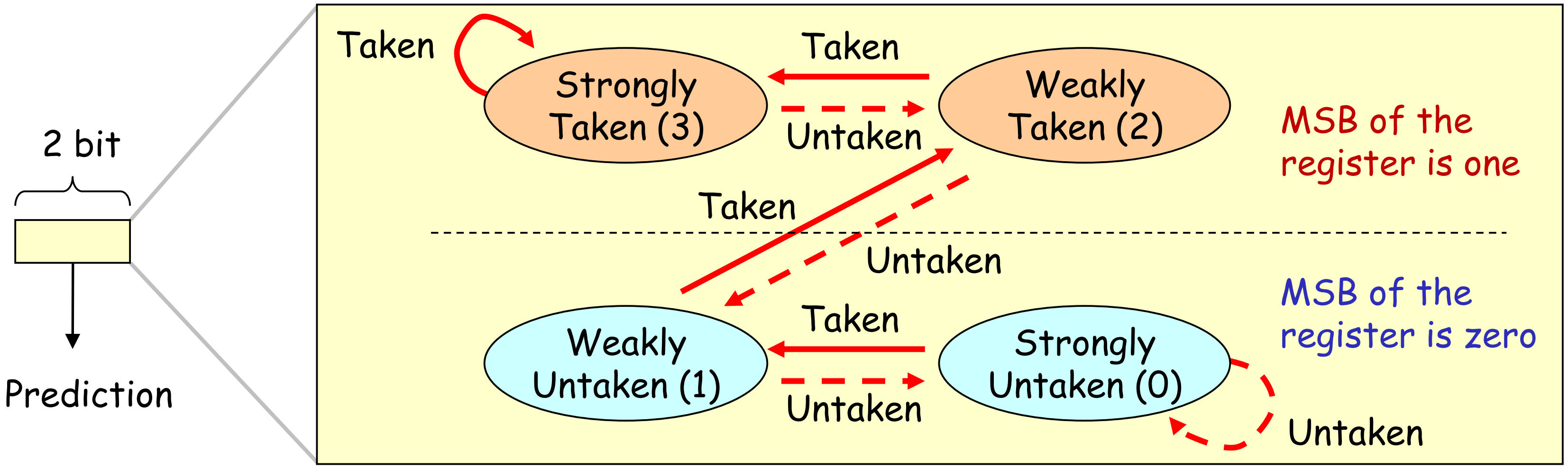




Predicting the branch outcome sequence of **1110 1110 1110 1110 1110** …

Simple branch predictor: 2-bit counter (2BC)

- It uses two bit register as a saturating counter.
- How to update the register
  - If the branch outcome is taken and the value is not 3, then increment the register.
  - If the branch outcome is untaken and the value is not 0, then decrement the register.
- Hot to predict
  - It predicts as 1 if the MSB of the register is one, otherwise predicts as 0.

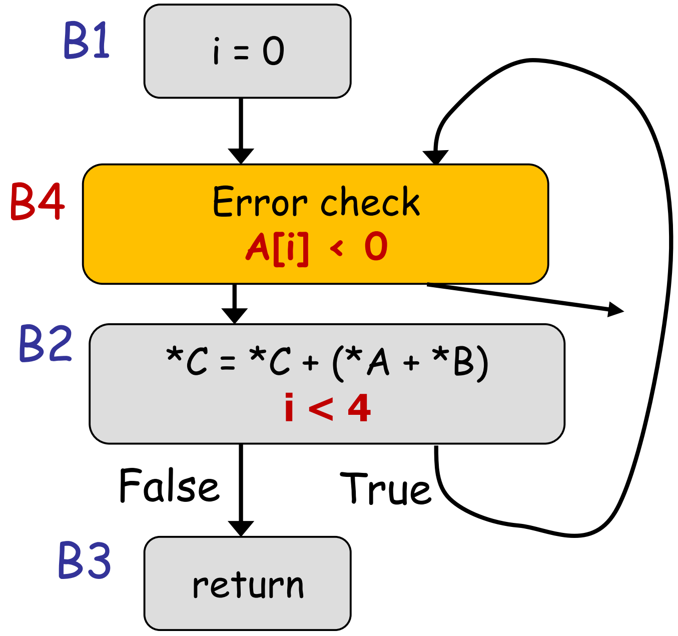


```
Predicting the sequence of 1110 1110 1110 1110 1110 ...
State of the counter       2333 2333 2333 2333 2333 ...
Prediction                 1111 1111 1111 1111 1111 ...
Hit/Miss of the pred.      HHHM HHHM HHHM HHHM HHHM
```

Sample program: vector add with two branches

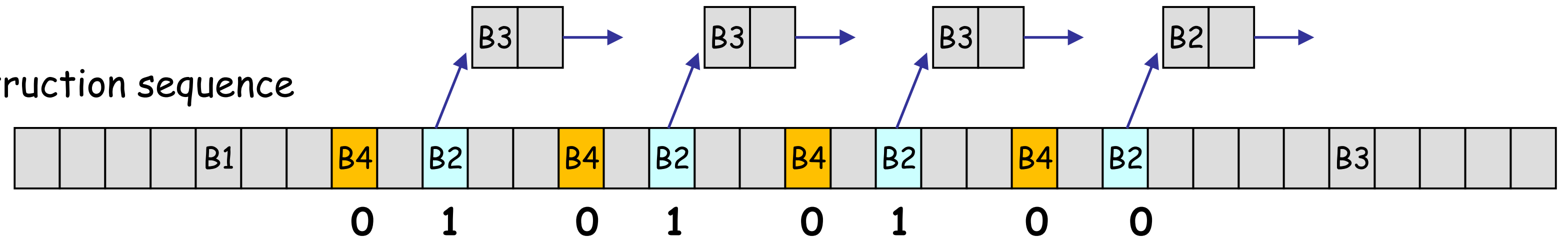
```
#define VSIZE 4
void v_add(int *A, int *B, int *C){
  for(i=0; i<VSIZE; i++) {
    if(A[i]<0) error_routine();
    C[i] += (A[i] + B[i]);
  }
}
```

We add a branch for error checking.
We assume that this error rarely occurs.



Control flow graph

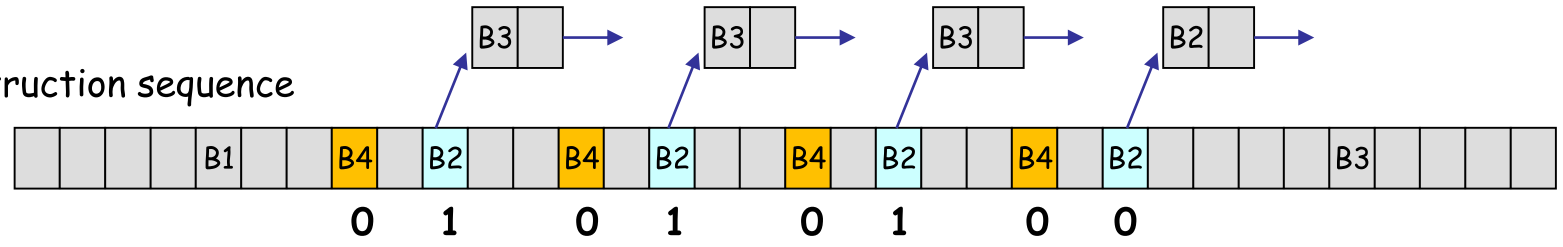
Executed instruction sequence



Predicting the sequence of **01010100** 01010100 01010100 ...

Sample program: vector add with two branches

Executed instruction sequence



Predicting the branch outcome sequence

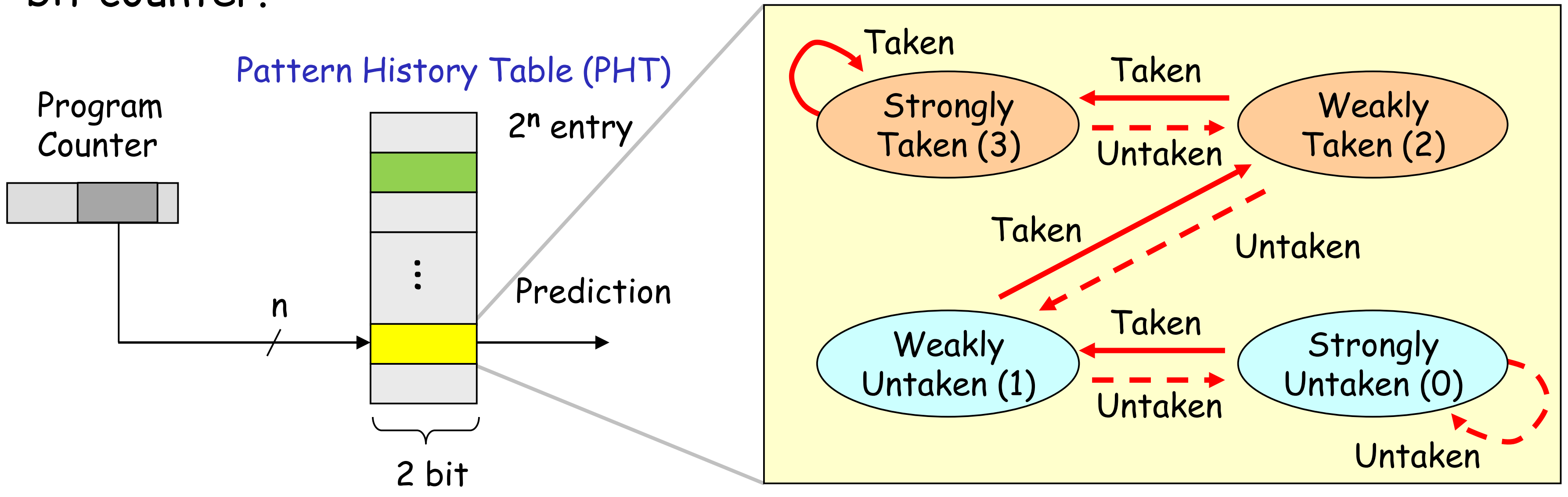
01010100 01010100 01010100 ...

The B4's sequence 01010100 01010100 01010100 ...

The B2's sequence 01010100 01010100 01010100 ...

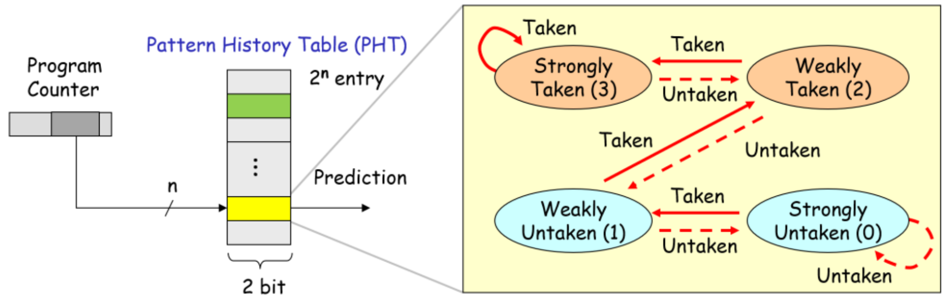
Simple branch predictor: bimodal

- 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.



Simple branch predictor: bimodal

```c
#define N 1024    // Number of PHT entries
int pht[N];       // pattern history table
int idx;          // index of PHT
/********************************************************************************/
void init_predictor()
{
    for(int i=0; i<N; i++) pht[i] = 2;
}

/********************************************************************************/
int make_prediction(unsigned int pc)
{
    idx = (pc>>2) % N;
    return (pht[idx] & 0x2) ? 1 : 0;
}

/********************************************************************************/
void train_predictor(unsigned int pc, int outcome)
{
    if(outcome==1 && pht[idx]<3) pht[idx]++;
    if(outcome==0 && pht[idx]>0) pht[idx]--;
}

/********************************************************************************/
int main()
{
    int pred;     // branch prediction
    int outcome;  // branch outcome (taken/untaken)
    init_predictor();

    int pc = 0x20;
    for(int i=1; i<25; i++) {
        pred = make_prediction(pc); /***** prediction *****/

        outcome = (i % 4) ? 1 : 0; /***** branch outcome: 111011101110... *****/

        printf("%4d: pc=%3x, idx=%d, cnt=%d, pred=%d, outcome=%d ",
               i, pc, idx, pht[idx], pred, outcome);

        train_predictor(pc, outcome); /***** training *****/

        if(pred==outcome) printf("hit\n"); else printf("miss\n");
    }
    return 0;
}
```

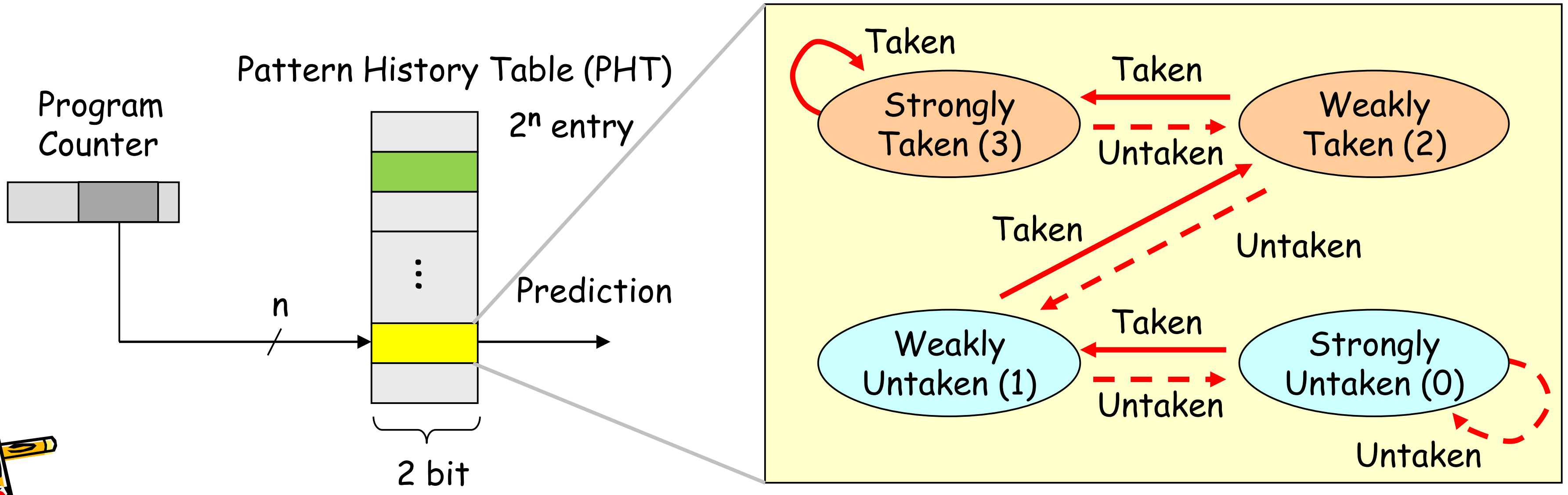


Predicting the branch outcome sequence
1110 1110 1110 1110 1110 …

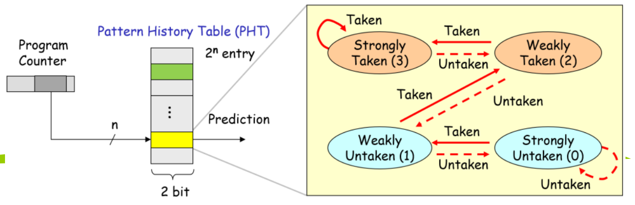
```
   1: pc= 20, idx=8, cnt=2, pred=1, outcome=1 hit
   2: pc= 20, idx=8, cnt=3, pred=1, outcome=1 hit
   3: pc= 20, idx=8, cnt=3, pred=1, outcome=1 hit
   4: pc= 20, idx=8, cnt=3, pred=1, outcome=0 miss
   5: pc= 20, idx=8, cnt=2, pred=1, outcome=1 hit
   6: pc= 20, idx=8, cnt=3, pred=1, outcome=1 hit
   7: pc= 20, idx=8, cnt=3, pred=1, outcome=1 hit
   8: pc= 20, idx=8, cnt=3, pred=1, outcome=0 miss
   9: pc= 20, idx=8, cnt=2, pred=1, outcome=1 hit
  10: pc= 20, idx=8, cnt=3, pred=1, outcome=1 hit
  11: pc= 20, idx=8, cnt=3, pred=1, outcome=1 hit
  12: pc= 20, idx=8, cnt=3, pred=1, outcome=0 miss
  13: pc= 20, idx=8, cnt=2, pred=1, outcome=1 hit
  14: pc= 20, idx=8, cnt=3, pred=1, outcome=1 hit
  15: pc= 20, idx=8, cnt=3, pred=1, outcome=1 hit
  16: pc= 20, idx=8, cnt=3, pred=1, outcome=0 miss
  17: pc= 20, idx=8, cnt=2, pred=1, outcome=1 hit
  18: pc= 20, idx=8, cnt=3, pred=1, outcome=1 hit
  19: pc= 20, idx=8, cnt=3, pred=1, outcome=1 hit
  20: pc= 20, idx=8, cnt=3, pred=1, outcome=0 miss
  21: pc= 20, idx=8, cnt=2, pred=1, outcome=1 hit
  22: pc= 20, idx=8, cnt=3, pred=1, outcome=1 hit
  23: pc= 20, idx=8, cnt=3, pred=1, outcome=1 hit
  24: pc= 20, idx=8, cnt=3, pred=1, outcome=0 miss
```

Simple branch predictor: bimodal

Predicting the sequence	01010100	01010100	01010100	...
The B4's sequence	01010100	01010100	01010100	...
State of the counter	2 1 0 0	0 0 0 0	0 0 0 0	...
Prediction	1 0 0 0	0 0 0 0	0 0 0 0	...
Hit/Miss or the pred.	M H H H	H H H H	H H H H	...
The B2's sequence	01010100	01010100	01010100	...
State of the counter	2 3 3 3	2 3 3 3	2 3 3 3	...
Prediction	1 1 1 1	1 1 1 1	1 1 1 1	...
Hit/Miss or the pred.	H H H M	H H H M	H H H M	...



Simple branch predictor: bimodal

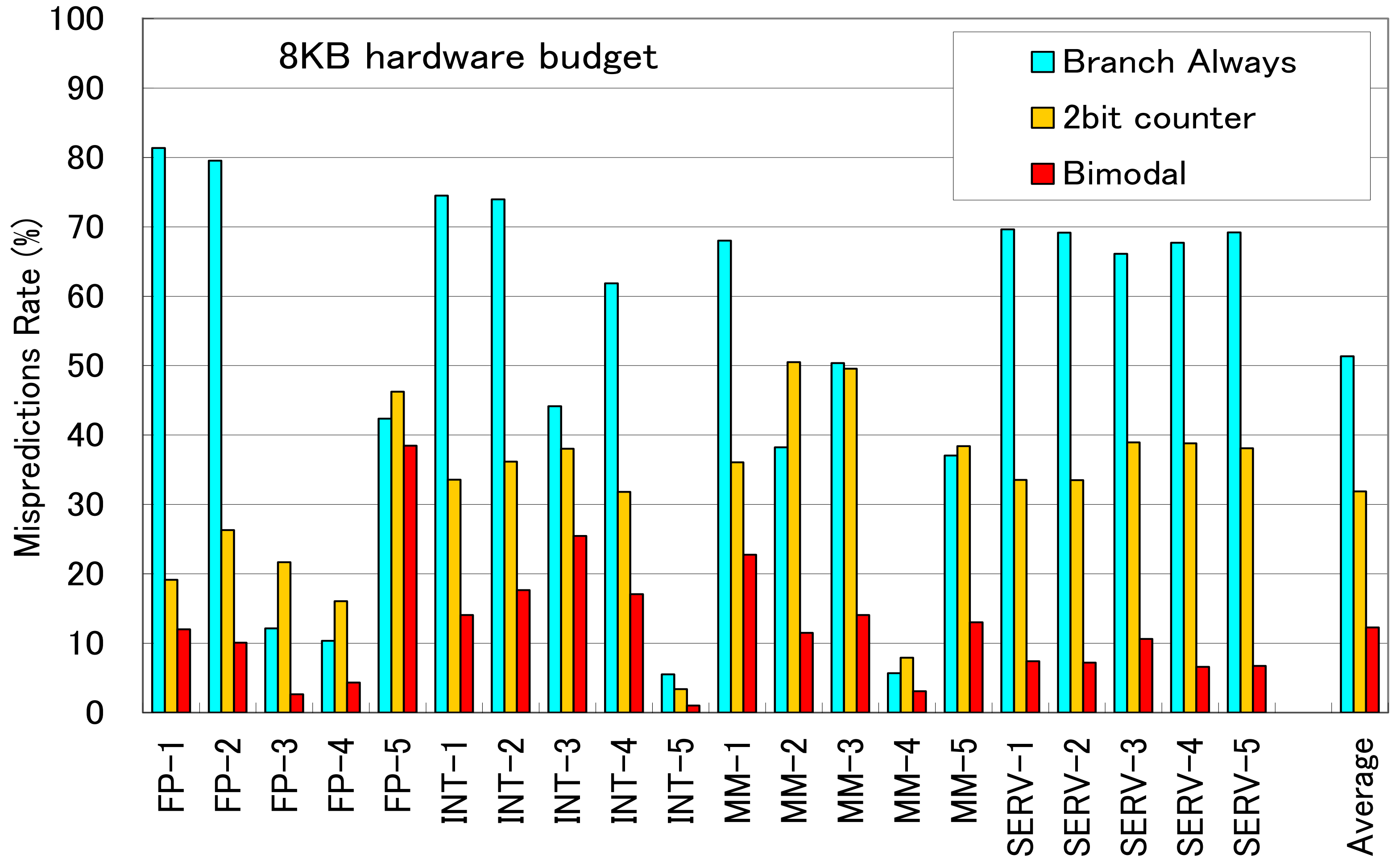


```
/*********************************************************************************/
int make_prediction(unsigned int pc)
{
    idx = (pc>>2) % N;
    return (pht[idx] & 0x2) ? 1 : 0;
}

/*********************************************************************************/
void train_predictor(unsigned int pc, int outcome)
{
    if(outcome==1 && pht[idx]<3) pht[idx]++;
    if(outcome==0 && pht[idx]>0) pht[idx]--;
}

/*********************************************************************************/
int main()
{
    int pred;     // branch prediction
    int outcome;  // branch outcome (taken/untaken)
    init_predictor();

    int pc;
    for(int i=1; i<25; i++) {
        if(i&1) { pc = 0x10; } else { pc = 0x20;}

        pred = make_prediction(pc); /***** prediction *****/

        if(pc==0x10) {
            outcome = 0;
        }
        else {
            outcome = (i/2 % 4) ? 1 : 0; /***** outcome: 111011101110... *****/
        }

        printf("%4d: pc=%3x, idx=%d, cnt=%d, pred=%d, outcome=%d ",
               i, pc, idx, pht[idx], pred, outcome);

        train_predictor(pc, outcome); /***** training *****/

        if(pred==outcome) printf("hit\n"); else printf("miss\n");
    }
    return 0;
}
```

```
Predicting the sequence      01010100 01010100 01010100 ...
   The B4's sequence         01010100 01010100 01010100 ...
   State of the counter      2 1 0 0  0 0 0 0  0 0 0 0  ...
   Prediction                1 0 0 0  0 0 0 0  0 0 0 0  ...
   Hit/Miss or the pred.     M H H H  H H H H  H H H H  ...

   The B2's sequence         01010100 01010100 01010100 ...
   State of the counter       2 3 3 3  2 3 3 3  2 3 3 3 ...
   Prediction                 1 1 1 1  1 1 1 1  1 1 1 1 ...
   Hit/Miss or the pred.      H H H M  H H H M  H H H M ...
```

```
   1: pc= 10, idx=4, cnt=2, pred=1, outcome=0 miss
   2: pc= 20, idx=8, cnt=2, pred=1, outcome=1 hit
   3: pc= 10, idx=4, cnt=1, pred=0, outcome=0 hit
   4: pc= 20, idx=8, cnt=3, pred=1, outcome=1 hit
   5: pc= 10, idx=4, cnt=0, pred=0, outcome=0 hit
   6: pc= 20, idx=8, cnt=3, pred=1, outcome=1 hit
   7: pc= 10, idx=4, cnt=0, pred=0, outcome=0 hit
   8: pc= 20, idx=8, cnt=3, pred=1, outcome=0 miss
   9: pc= 10, idx=4, cnt=0, pred=0, outcome=0 hit
  10: pc= 20, idx=8, cnt=2, pred=1, outcome=1 hit
  11: pc= 10, idx=4, cnt=0, pred=0, outcome=0 hit
  12: pc= 20, idx=8, cnt=3, pred=1, outcome=1 hit
  13: pc= 10, idx=4, cnt=0, pred=0, outcome=0 hit
  14: pc= 20, idx=8, cnt=3, pred=1, outcome=1 hit
  15: pc= 10, idx=4, cnt=0, pred=0, outcome=0 hit
  16: pc= 20, idx=8, cnt=3, pred=1, outcome=0 miss
  17: pc= 10, idx=4, cnt=0, pred=0, outcome=0 hit
  18: pc= 20, idx=8, cnt=2, pred=1, outcome=1 hit
  19: pc= 10, idx=4, cnt=0, pred=0, outcome=0 hit
  20: pc= 20, idx=8, cnt=3, pred=1, outcome=1 hit
  21: pc= 10, idx=4, cnt=0, pred=0, outcome=0 hit
  22: pc= 20, idx=8, cnt=3, pred=1, outcome=1 hit
  23: pc= 10, idx=4, cnt=0, pred=0, outcome=0 hit
  24: pc= 20, idx=8, cnt=3, pred=1, outcome=0 miss
```

Accuracy of simple predictors with 8KB HW budget



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

An innovation in branch predictors in 1993

- Using branch history
  - global branch history
  - local branch history
- 2-level branch predictor and gshare
- Assume predicting the sequence 1110 1110 1110 1110 1110 ...

```
1110111 0
11101110 ?
111011101 ?
1110111011 ?
11101110111 ?
111011101110 ?
```

adr	pred
000	
001	
010	
011	1
100	
101	1
110	1
111	0

Use the recent branch history as an address of a table.

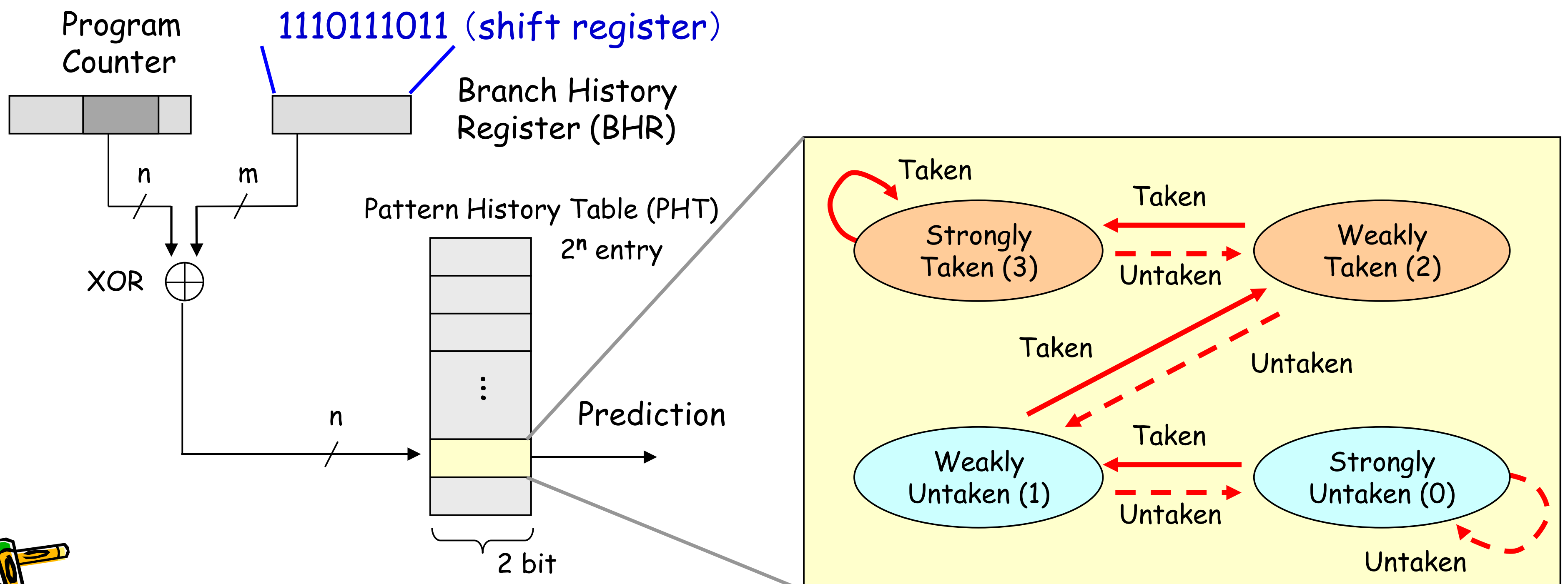
Recommended Reading

- Combining Branch Predictors
  - Scott McFarling, Digital Western Research Laboratory
  - WRL Technical Note TN-36, 1993

- A quote:
  "In this paper, we have presented two new methods for improving branch prediction performance. First, we showed that using the bit-wise exclusive OR of the global branch history and the branch address to access predictor counters results in better performance for a given counter array size."

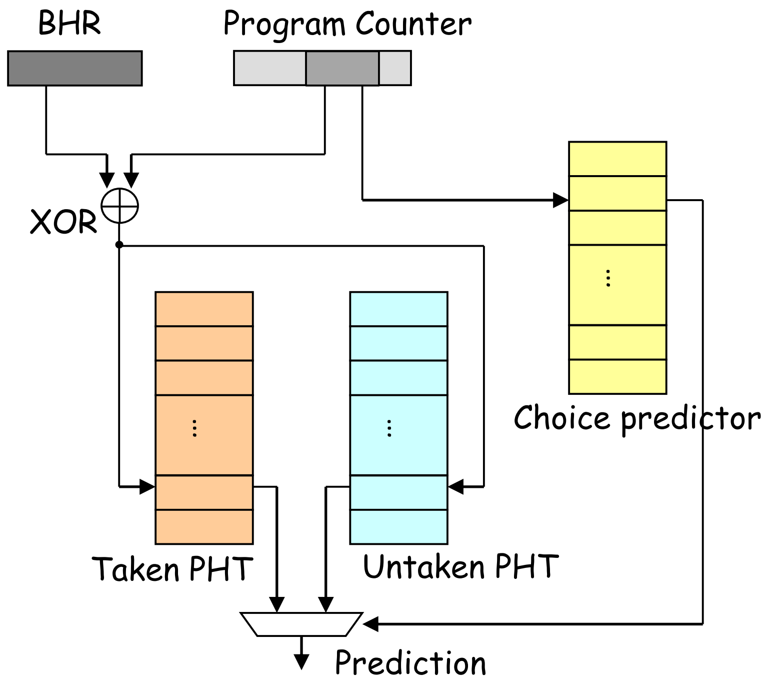
Gshare (TR-DEC 1993)

- How to predict
  - Using the exclusive OR of the global branch history and PC to access PHT, then MSB of the selected counter is the prediction.
- How to update
  - Shifting BHR one bit left and update LSB by branch outcome in IF stage.
  - Update the used counter in the same way as 2BC in WB stage.



Bi-Mode (MICRO 1997)

- A choice predictor (bimodal) is used as a meta-predictor
- How to predict
  - Like gshare, both of Taken PHT and Untaken PHT make two predictions.
  - Select one among them by the choice predictor which tracks the global bias of a branch.
- How to update
  - The used PHT is updated in the same way as 2BC.
  - Choice predictor is updated in the same way as bimodal.



To go beyond gshare

- Using branch history
  - global branch history
  - local branch history
- 2-level branch predictor and gshare
- Assume predicting the sequence 1110 1110 1110 1110 1110 ...

1110111 0
11101110 ?
111011101 ?
1110111011 ?
11101110111 ?
111011101110 ?

adr	pred
000	
001	
010	
011	1
100	
101	1
110	1
111	0

Gshare: use the recent branch history as an address of a table.

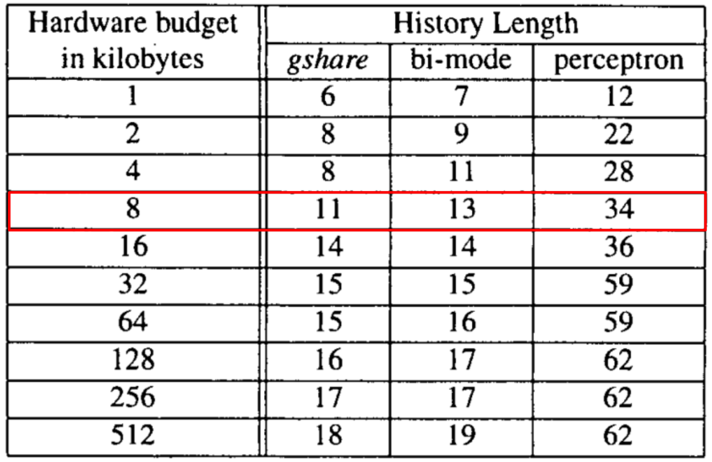
11101110 ?
111011101 ?
1110111011 ?
11101110111 ?
111011101110 ?

11101110 ?
111011101 ?
1110111011 ?
11101110111 ?
111011101110 ?

Use long branch history and weights (importance) of each history bit

Recommended Reading

- Dynamic branch prediction with perceptrons
  - Daniel A. Jimenez, Calvin Lin (The University of Texas at Austin)
  - HPCA-7, pp. 197-206 (2001)



Hardware budget in kilobytes	History Length		
	gshare	bi-mode	perceptron
1	6	7	12
2	8	9	22
4	8	11	28
8	11	13	34
16	14	14	36
32	15	15	59
64	15	16	59
128	16	17	62
256	17	17	62
512	18	19	62

Table 1: Best History Lengths. This table shows the best amount of global history to keep for each of the branch prediction schemes.

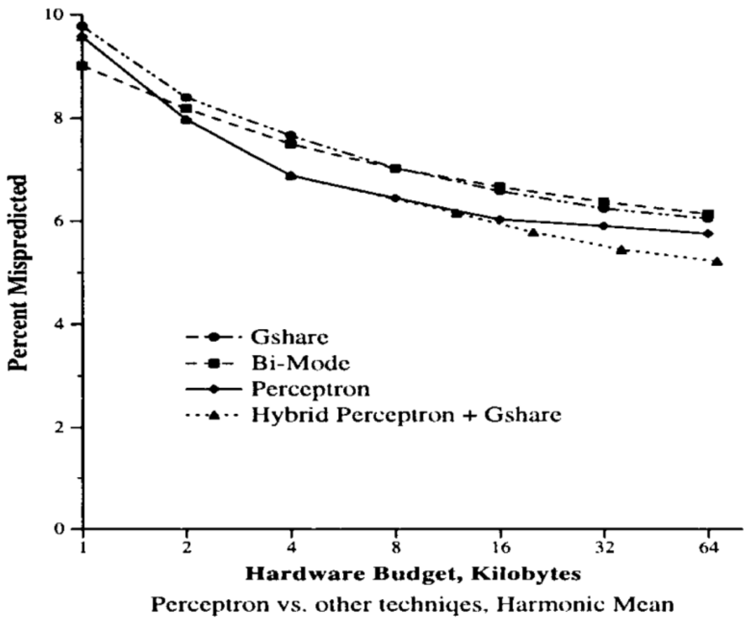


Figure 3: Hardware Budget vs. Prediction Rate on SPEC 2000. The perceptron predictor is more accurate than the two PHT methods at all hardware budgets over one kilobyte.

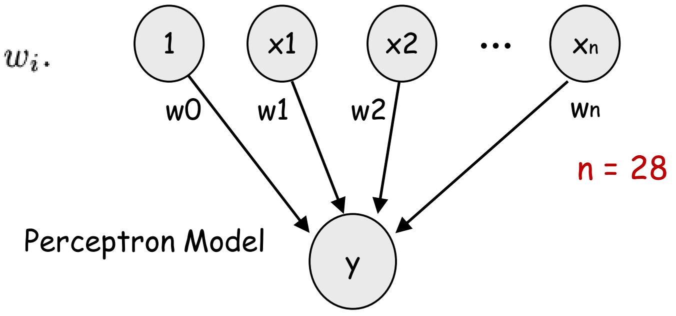
Perceptron (HPCA 2001)

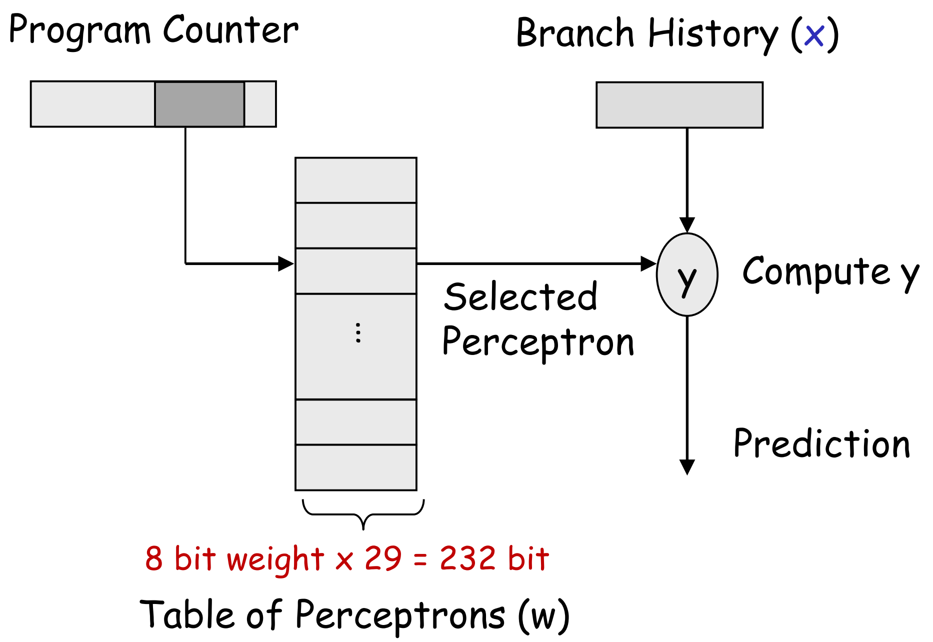
- How to predict
  - Select one perceptron by PC
  - Compute y using the equation. It predicts 1 if y>=0, predicts 0 if y<0
  - x is branch history. xi is either -1, meaning not taken or 1, meaning taken
- How to update
  - Train the weights of used perceptron when the prediction miss or |y| < T (Threshold)

```
if sign($y_{out}$) $\neq t$ or $|y_{out}| \leq \theta$ then
        for $i := 0$ to $n$ do
                $w_i := w_i + t x_i$
        end for
end if
```

T = 1.93n + 14

$$y = w_0 + \sum_{i=1}^{n} x_i w_i.$$





Perceptron (HPCA 2001)

- How to predict
  - Select one perceptron by PC
  - Compute y using the equation. It predicts 1 if y>=0, predicts 0 if y<0
  - x is branch history. xi is either -1, meaning not taken or 1, meaning taken
- How to update
  - Train the weights of used perceptron when the prediction miss or |y| < T (Threshold)

$$y = w_0 + \sum_{i=1}^{n} x_i w_i.$$

```
if $\mathrm{sign}(y_{out}) \neq t$ or $|y_{out}| \leq \theta$ then
    for $i := 0$ to $n$ do
        $w_i := w_i + t x_i$
    end for
end if
```

T = 1.93n + 14

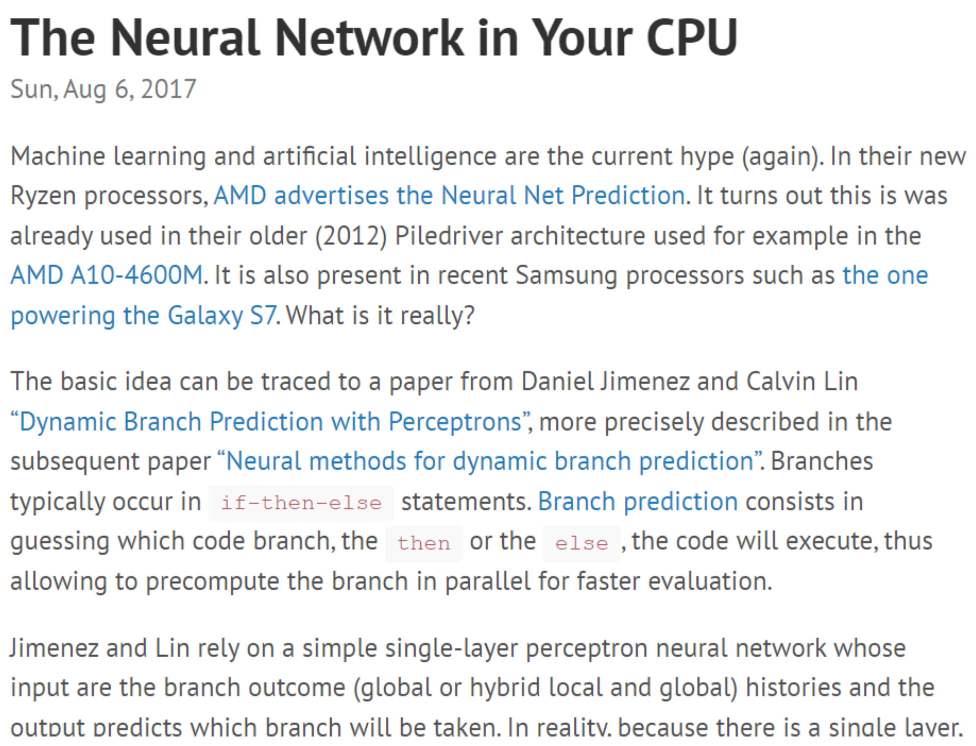
```
Number of weights (without bias) of perceptron: 4
Theta:  21.720
   1: Wn-W0 =   0   0   0   0   0 : bhr=0000: y=  0, p=1 : out=1 : hit
   2: Wn-W0 =  -1  -1  -1  -1   1 : bhr=0001: y=  3, p=1 : out=1 : hit
   3: Wn-W0 =  -2  -2  -2   0   2 : bhr=0011: y=  4, p=1 : out=1 : hit
   4: Wn-W0 =  -3  -3  -1   1   3 : bhr=0111: y=  3, p=1 : out=0 : miss
   5: Wn-W0 =  -2  -4  -2   0   2 : bhr=1110: y= -6, p=0 : out=1 : miss
   6: Wn-W0 =  -1  -3  -1  -1   3 : bhr=1101: y= -1, p=0 : out=1 : miss
   7: Wn-W0 =   0  -2  -2   0   4 : bhr=1011: y=  4, p=1 : out=1 : hit
   8: Wn-W0 =   1  -3  -1   1   5 : bhr=0111: y=  1, p=1 : out=0 : miss
   9: Wn-W0 =   2  -4  -2   0   4 : bhr=1110: y=  0, p=1 : out=1 : hit
  10: Wn-W0 =   3  -3  -1  -1   5 : bhr=1101: y=  5, p=1 : out=1 : hit
  11: Wn-W0 =   4  -2  -2   0   6 : bhr=1011: y= 10, p=1 : out=1 : hit
  12: Wn-W0 =   5  -3  -1   1   7 : bhr=0111: y= -1, p=0 : out=0 : hit
  13: Wn-W0 =   6  -4  -2   0   6 : bhr=1110: y=  6, p=1 : out=1 : hit
  14: Wn-W0 =   7  -3  -1  -1   7 : bhr=1101: y= 11, p=1 : out=1 : hit
  15: Wn-W0 =   8  -2  -2   0   8 : bhr=1011: y= 16, p=1 : out=1 : hit
  16: Wn-W0 =   9  -3  -1   1   9 : bhr=0111: y= -3, p=0 : out=0 : hit
  17: Wn-W0 =  10  -4  -2   0   8 : bhr=1110: y= 12, p=1 : out=1 : hit
  18: Wn-W0 =  11  -3  -1  -1   9 : bhr=1101: y= 17, p=1 : out=1 : hit
  19: Wn-W0 =  12  -2  -2   0  10 : bhr=1011: y= 22, p=1 : out=1 : hit
  20: Wn-W0 =  12  -2  -2   0  10 : bhr=0111: y= -6, p=0 : out=0 : hit
  21: Wn-W0 =  13  -3  -3  -1   9 : bhr=1110: y= 17, p=1 : out=1 : hit
  22: Wn-W0 =  14  -2  -2  -2  10 : bhr=1101: y= 22, p=1 : out=1 : hit
  23: Wn-W0 =  14  -2  -2  -2  10 : bhr=1011: y= 22, p=1 : out=1 : hit
  24: Wn-W0 =  14  -2  -2  -2  10 : bhr=0111: y=-10, p=0 : out=0 : hit
  25: Wn-W0 =  15  -3  -3  -3   9 : bhr=1110: y= 21, p=1 : out=1 : hit
  26: Wn-W0 =  16  -2  -2  -4  10 : bhr=1101: y= 22, p=1 : out=1 : hit
  27: Wn-W0 =  16  -2  -2  -4  10 : bhr=1011: y= 22, p=1 : out=1 : hit
  28: Wn-W0 =  16  -2  -2  -4  10 : bhr=0111: y=-14, p=0 : out=0 : hit
  29: Wn-W0 =  17  -3  -3  -5   9 : bhr=1110: y= 25, p=1 : out=1 : hit
```

Perceptron (HPCA 2001)

```c
/********************************************************************************/
/* perceptron based branch predictor Version v2024-12-26a                       */
/*  Copyright (c) 2024 Archlab. Science Tokyo                                   */
/*  Released under the MIT license https://opensource.org/licenses/mit          */
/********************************************************************************/
#include <stdio.h>

#define N 4                     // Number of weights of perceptron, default 28
#define BitsInWeight 8          // Number of bits in a weight
#define MAXVAL  127             // max value of a weight
#define MINVAL -128             // min value of a weight
#define NPerceptron (1024)      // the number of perceptrons
#define ThetaMax (N * 1.93 + 14)  // Threshold max value
#define ThetaMin (-1 * ThetaMax)  // Threshold min value

int perceptron[NPerceptron][N+1]; // perceptron table
int bhr;                          // global branch history register
int idx;                          // index of perceptron table
int y;                            // weighted sum with bias
int prediction;                   // prediction of taken/untaken

/********************************************************************************/
void init_predictor()
{
    for(int i=0; i<NPerceptron; i++){
        for(int j=0 ; j<=N ; j++){
            perceptron[i][j] = 0;
        }
    }
    bhr = 0;
}

/********************************************************************************/
int make_prediction(unsigned int pc)
{
    idx = (pc>>2) % NPerceptron;

    y = perceptron[idx][0];
    for(int i=1; i<=N; i++){
        if((bhr >> (i-1)) & 1) y += perceptron[idx][i];
        else                   y -= perceptron[idx][i];
    }

    prediction = (y >= 0) ? 1 : 0;
    return prediction;
}
```

```c
void train_predictor(unsigned int pc, int outcome)
{
    if(outcome != prediction || ((y < ThetaMax) && (y > ThetaMin))){

        int *bias = &perceptron[idx][0];
        if(outcome==1 && (*bias < MAXVAL)) *bias = *bias + 1;
        if(outcome==0 && (*bias > MINVAL)) *bias = *bias - 1;

        for(int i=1; i <=N; i++){
            if(((bhr >> (i-1)) & 1)==outcome){
                if (perceptron[idx][i] < MAXVAL) perceptron[idx][i]++;
            }
            else{
                if (perceptron[idx][i] > MINVAL) perceptron[idx][i]--;
            }
        }
    }
    bhr = (bhr << 1) | outcome;
}

/********************************************************************************/
int main()
{
    int pred;     // branch prediction
    int outcome;  // branch outcome (taken/untaken)
    init_predictor();
    printf("Number of weights (without bias) of perceptron: %d\n", N);
    printf("Theta: %7.3f\n", ThetaMax);

    int pc = 0x2000;
    for(int i=1; i<30; i++) {
        pred = make_prediction(pc); /***** prediction *****/

        printf("%4d: Wn-W0 = ", i);
        for(int i=N; i>=0; i--) printf("%3d ", perceptron[idx][i]);

        outcome = (i % 4) ? 1 : 0; /***** branch outcome: 111011101110... *****/

        printf(": bhr=");
        for(int j=N-1; j>=0; j--){
            printf("%d", ((bhr>>j) & 1));
        }
        printf(": y=%3d, p=%d : out=%d : ", y, pred, outcome);

        train_predictor(pc, outcome); /***** training *****/


        if(pred==outcome) printf("hit\n"); else printf("miss\n");
    }
    return 0;
}
```

Perceptron (HPCA 2001)



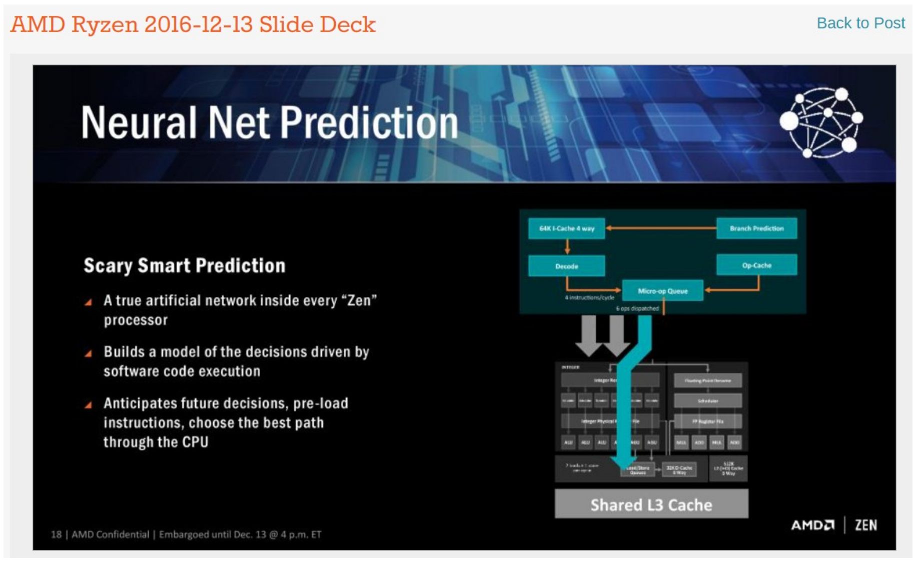
The Neural Network in Your CPU

Sun, Aug 6, 2017

Machine learning and artificial intelligence are the current hype (again). In their new Ryzen processors, AMD advertises the Neural Net Prediction. It turns out this is was already used in their older (2012) Piledriver architecture used for example in the AMD A10-4600M. It is also present in recent Samsung processors such as the one powering the Galaxy S7. What is it really?

The basic idea can be traced to a paper from Daniel Jimenez and Calvin Lin "Dynamic Branch Prediction with Perceptrons", more precisely described in the subsequent paper "Neural methods for dynamic branch prediction". Branches typically occur in `if-then-else` statements. Branch prediction consists in guessing which code branch, the `then` or the `else`, the code will execute, thus allowing to precompute the branch in parallel for faster evaluation.

Jimenez and Lin rely on a simple single-layer perceptron neural network whose input are the branch outcome (global or hybrid local and global) histories and the output predicts which branch will be taken. In reality, because there is a single layer,



AMD Ryzen 2016-12-13 Slide Deck

Back to Post

Neural Net Prediction

Scary Smart Prediction

- A true artificial network inside every "Zen" processor
- Builds a model of the decisions driven by software code execution
- Anticipates future decisions, pre-load instructions, choose the best path through the CPU

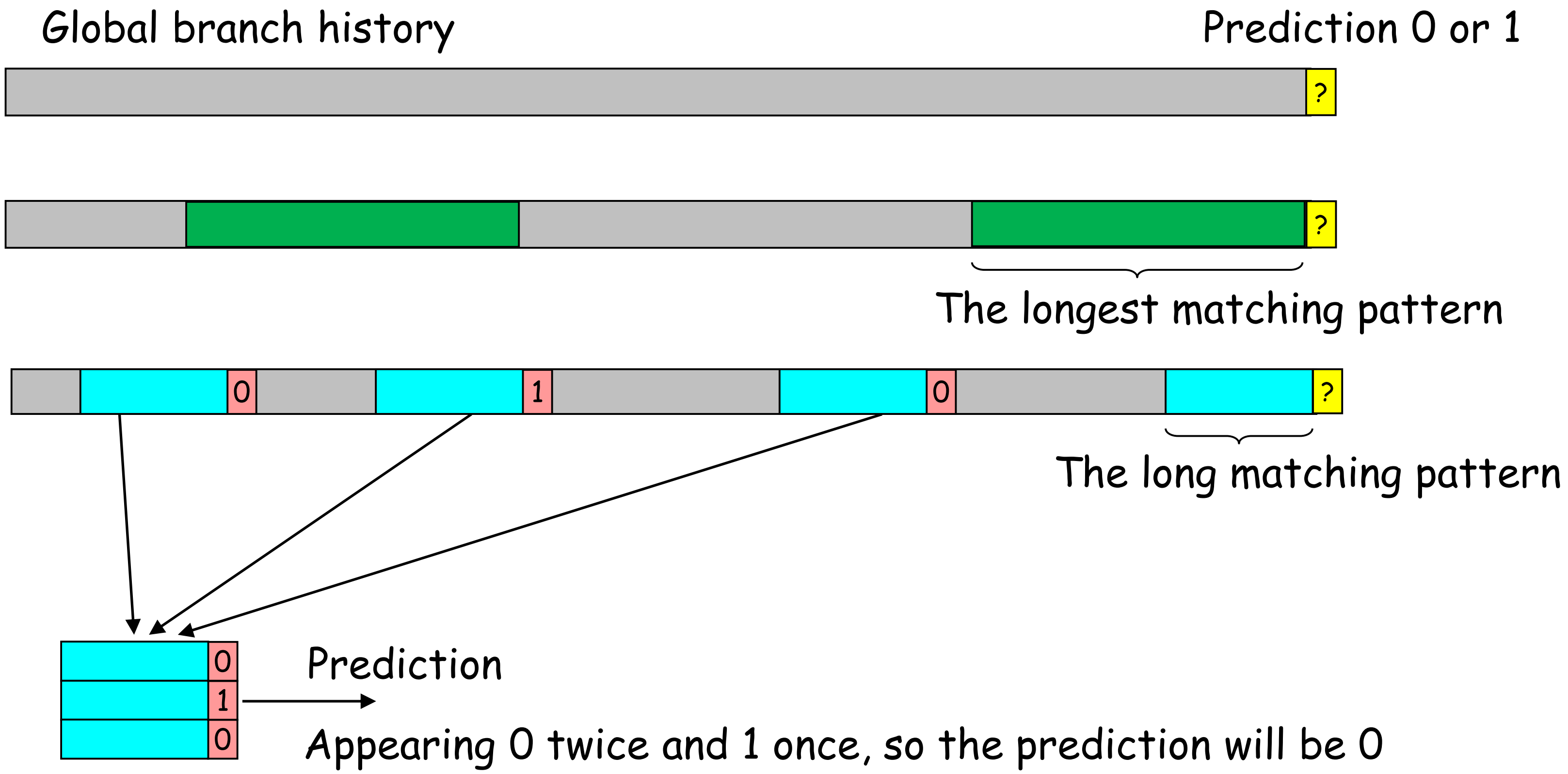
18 | AMD Confidential | Embargoed until Dec. 13 @ 4 p.m. ET

https://www.anandtech.com/Gallery/Album/5197#18

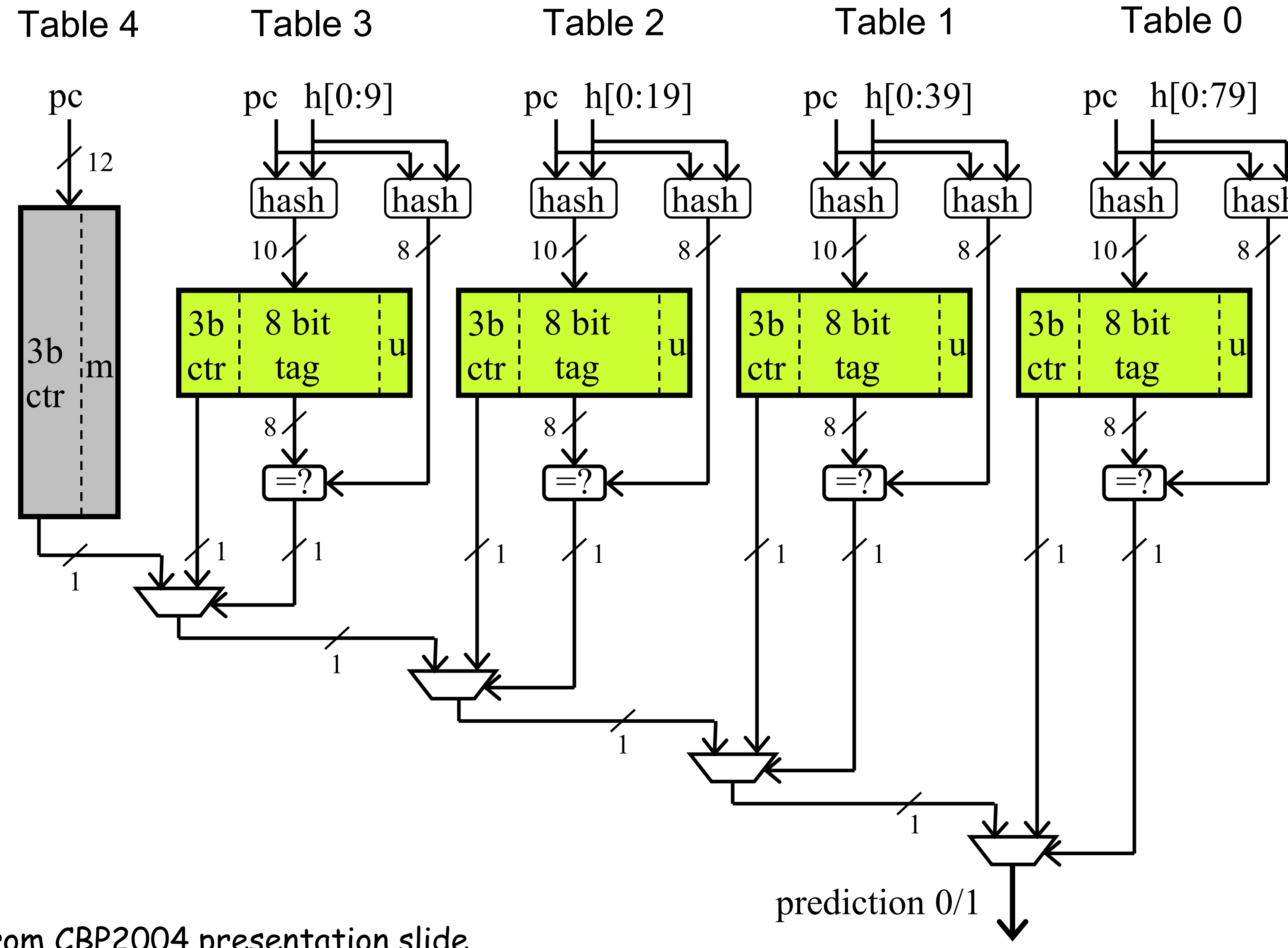
https://chasethedevil.github.io/post/the_neural_network_in_your_cpu/

Branch predictors based on pattern matching

- Find the longest matching pattern (green rectangle)
- Select the proper matching length or long matching pattern (blue rectangle)
- Count the number of 0 and the number of 1 after the long matting patterns (red rectangle), then predict by majority vote.

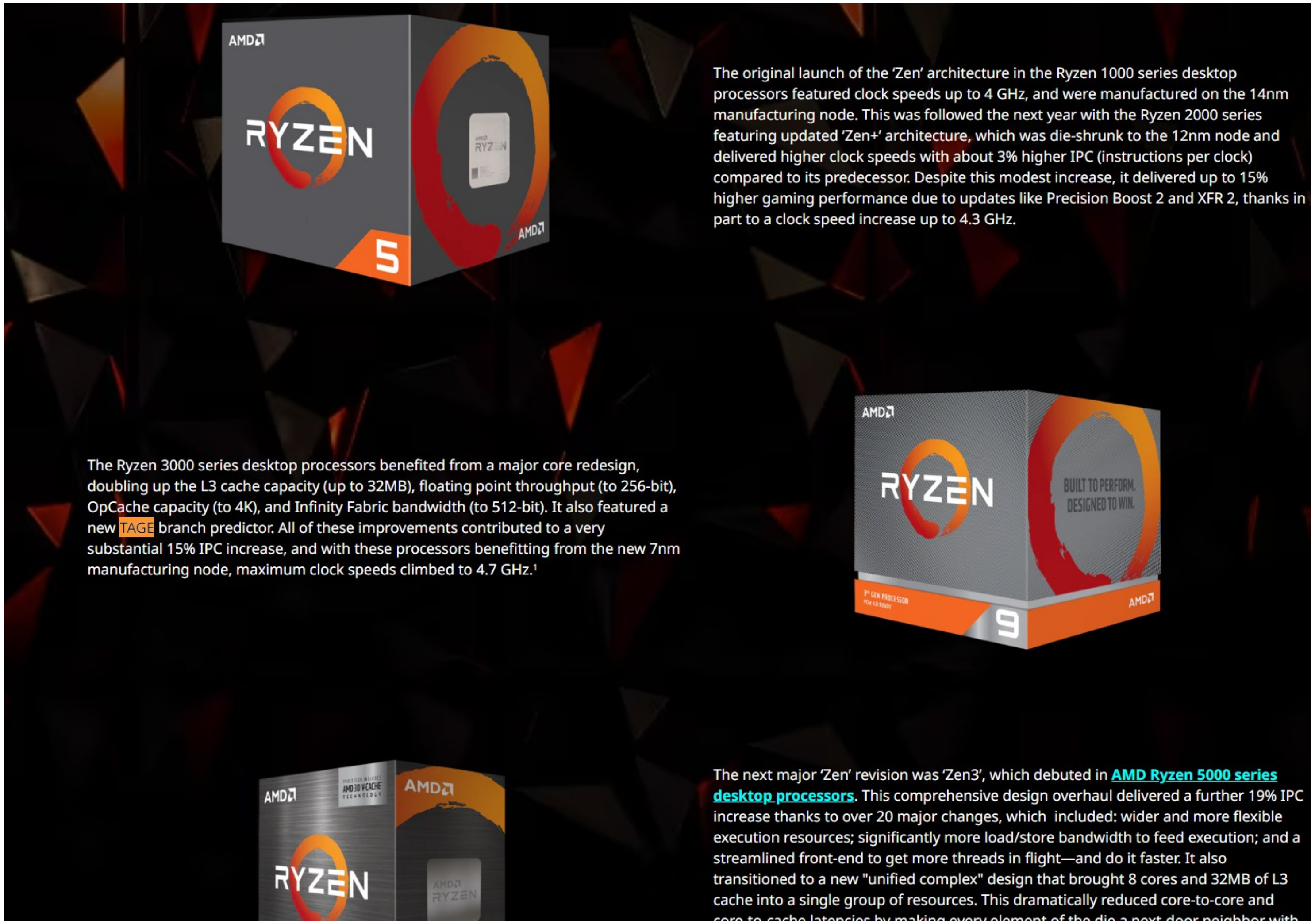

Global branch history
Prediction 0 or 1
?
?
The longest matching pattern
0
1
0
?
The long matching pattern
0
1
0
Prediction
Appearing 0 twice and 1 once, so the prediction will be 0

Partial Pattern Matching, PPM or TAGE (CBP 2004)



From CBP2004 presentation slide

Partial Pattern Matching, PPM or TAGE (CBP 2004)



The original launch of the 'Zen' architecture in the Ryzen 1000 series desktop processors featured clock speeds up to 4 GHz, and were manufactured on the 14nm manufacturing node. This was followed the next year with the Ryzen 2000 series featuring updated 'Zen+' architecture, which was die-shrunk to the 12nm node and delivered higher clock speeds with about 3% higher IPC (instructions per clock) compared to its predecessor. Despite this modest increase, it delivered up to 15% higher gaming performance due to updates like Precision Boost 2 and XFR 2, thanks in part to a clock speed increase up to 4.3 GHz.

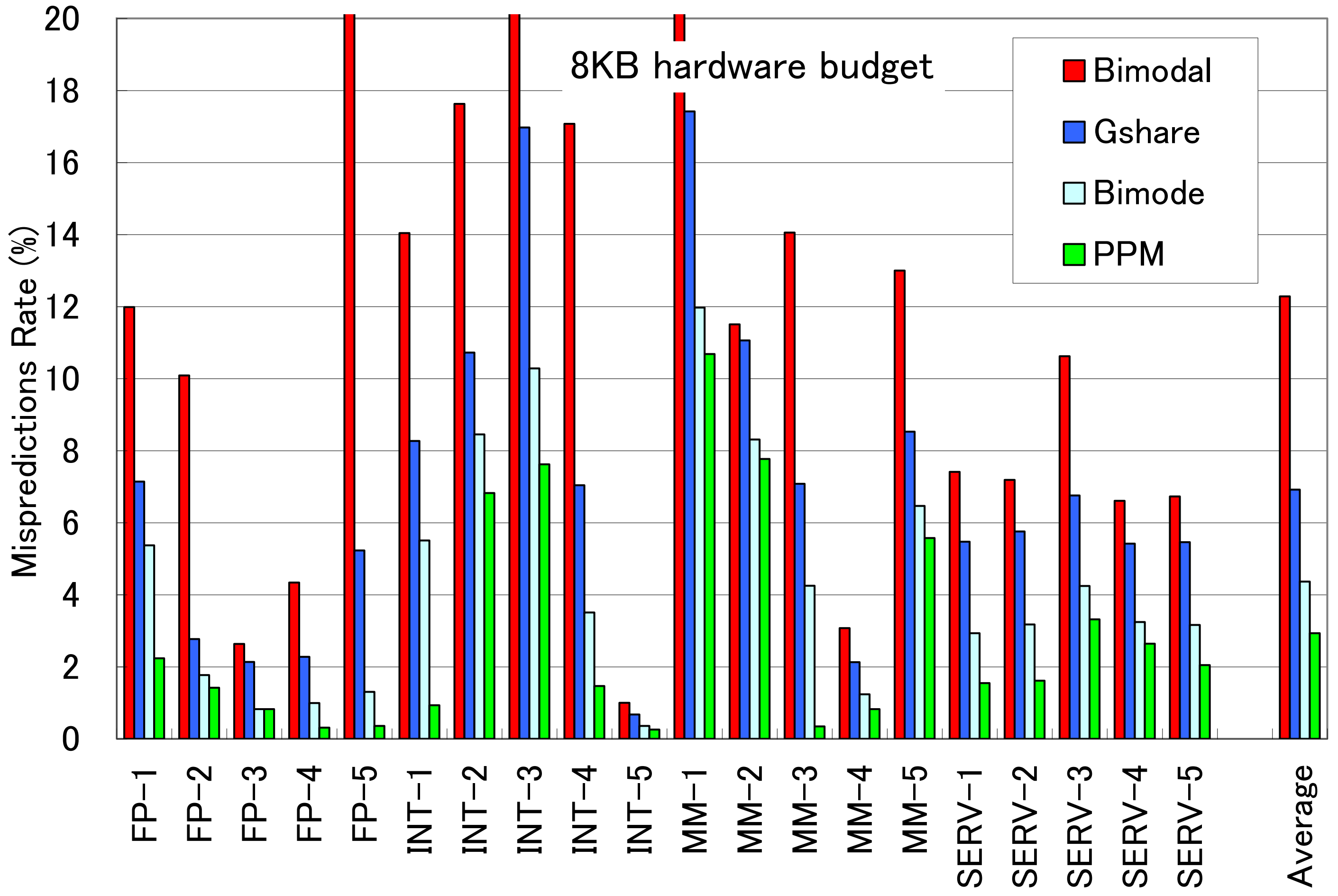
The Ryzen 3000 series desktop processors benefited from a major core redesign, doubling up the L3 cache capacity (up to 32MB), floating point throughput (to 256-bit), OpCache capacity (to 4K), and Infinity Fabric bandwidth (to 512-bit). It also featured a new TAGE branch predictor. All of these improvements contributed to a very substantial 15% IPC increase, and with these processors benefitting from the new 7nm manufacturing node, maximum clock speeds climbed to 4.7 GHz.[1]

The next major 'Zen' revision was 'Zen3', which debuted in **AMD Ryzen 5000 series desktop processors**. This comprehensive design overhaul delivered a further 19% IPC increase thanks to over 20 major changes, which included: wider and more flexible execution resources; significantly more load/store bandwidth to feed execution; and a streamlined front-end to get more threads in flight—and do it faster. It also transitioned to a new "unified complex" design that brought 8 cores and 32MB of L3 cache into a single group of resources. This dramatically reduced core-to-core and core-to-cache latencies by making every element of the die a next-door neighbor with

https://www.amd.com/en/technologies/zen-core

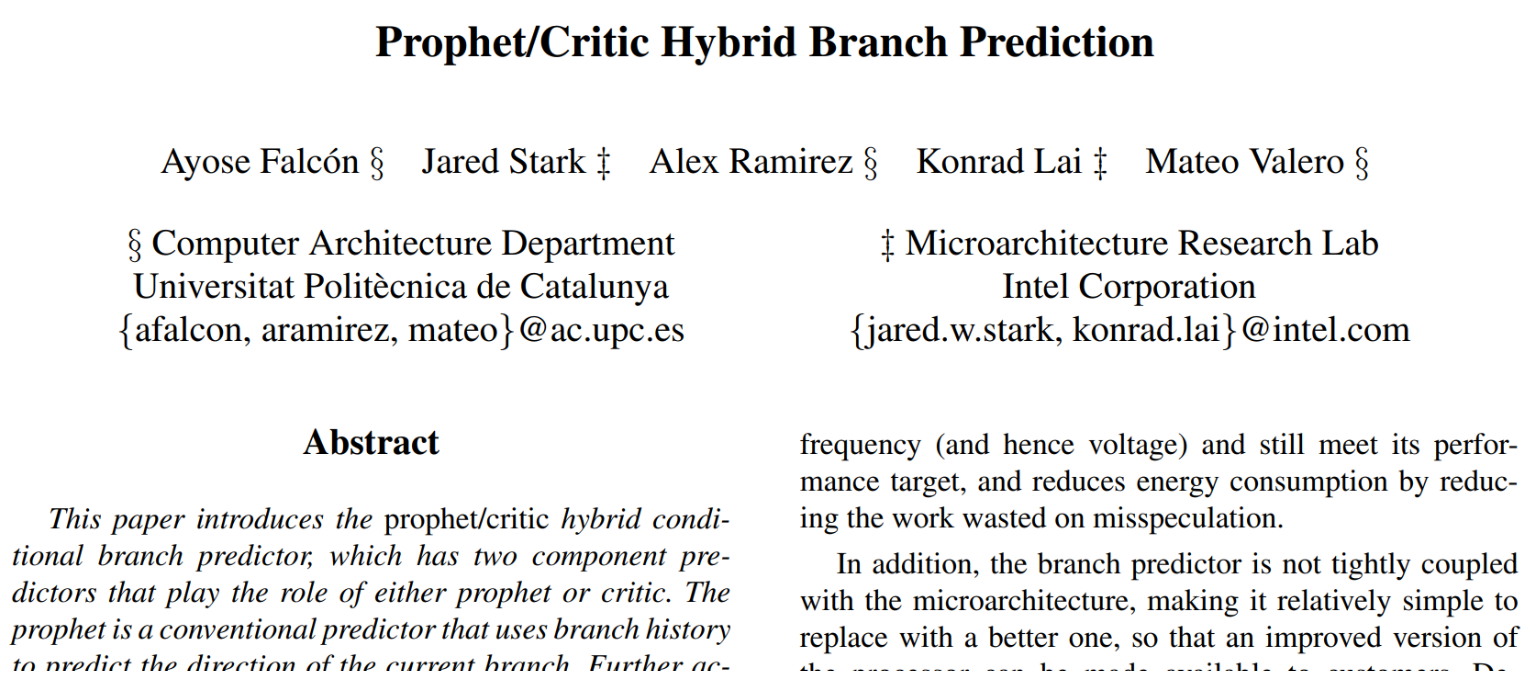
Prediction accuracy

- The accuracy of 4KB Gshare is about 93%.
- The accuracy of 4KB PPM is about 97%.



Recommended Reading

- Prophet-Critic Hybrid Branch Prediction
  - Ayose Falcon, UPC, Jared Stark, Intel, Alex Ramirez, UPC, Konrad Lai, Intel, Mateo Valero
  - ISCA-31 pp. 250-261 (2004)



Prophet/Critic Hybrid Branch Prediction

Ayose Falcón § Jared Stark ‡ Alex Ramirez § Konrad Lai ‡ Mateo Valero §

§ Computer Architecture Department
Universitat Politècnica de Catalunya
{afalcon, aramirez, mateo}@ac.upc.es

‡ Microarchitecture Research Lab
Intel Corporation
{jared.w.stark, konrad.lai}@intel.com

Abstract

This paper introduces the prophet/critic *hybrid conditional branch predictor, which has two component predictors that play the role of either prophet or critic. The prophet is a conventional predictor that uses branch history to predict the direction of the current branch. Further ac-*

frequency (and hence voltage) and still meet its performance target, and reduces energy consumption by reducing the work wasted on misspeculation.

In addition, the branch predictor is not tightly coupled with the microarchitecture, making it relatively simple to replace with a better one, so that an improved version of

A quote from Introduction (1/2)

Conventional predictors are analogous to a taxi with just one driver.

He gets the passenger to the destination using knowledge of the roads acquired from previous trips; i. e., using history information stored in the predictor's memory structures.

When he reaches an intersection, he uses this knowledge to decide which way to turn.

The driver accesses this knowledge in the context of his current location.

Modern branch predictors access it in the context of the current location (the program counter) plus a history of the most recent decisions that led to the current location.

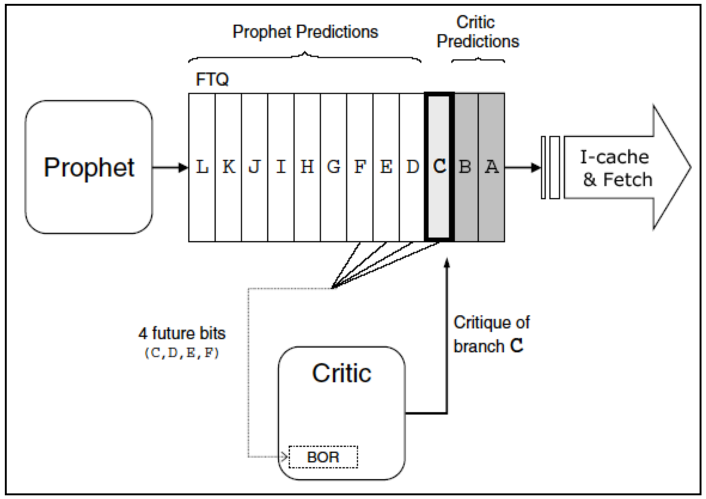
A quote from Introduction (2/2)

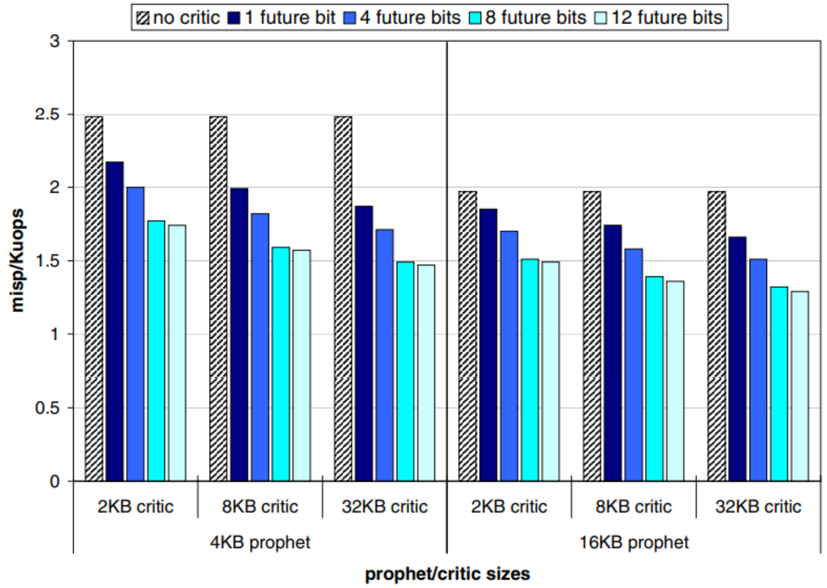
Prophet/critic hybrids are analogous to a taxi with two drivers: the front-seat and the back-seat. The front-seat driver has the same role as the driver in the single-driver taxi. This role is called the prophet.

The back-seat driver has the role of critic. She watches the turns the prophet makes at intersections. She doesn't say anything unless she thinks he's made a wrong turn. When she thinks he's made a wrong turn, she waits until he's made a few more turns to be certain they are lost. (Sometimes the prophet makes turns that initially look questionable, but, after he makes a few more turns, in hindsight appear to be correct.) Only when she's certain does she point out the mistake.

To recover, they backtrack to the intersection where she believes the wrong-turn was made and try a different direction.

Prophet-Critic Hybrid Branch Prediction





(c) Prophet: perceptron; Critic: tagged gshare

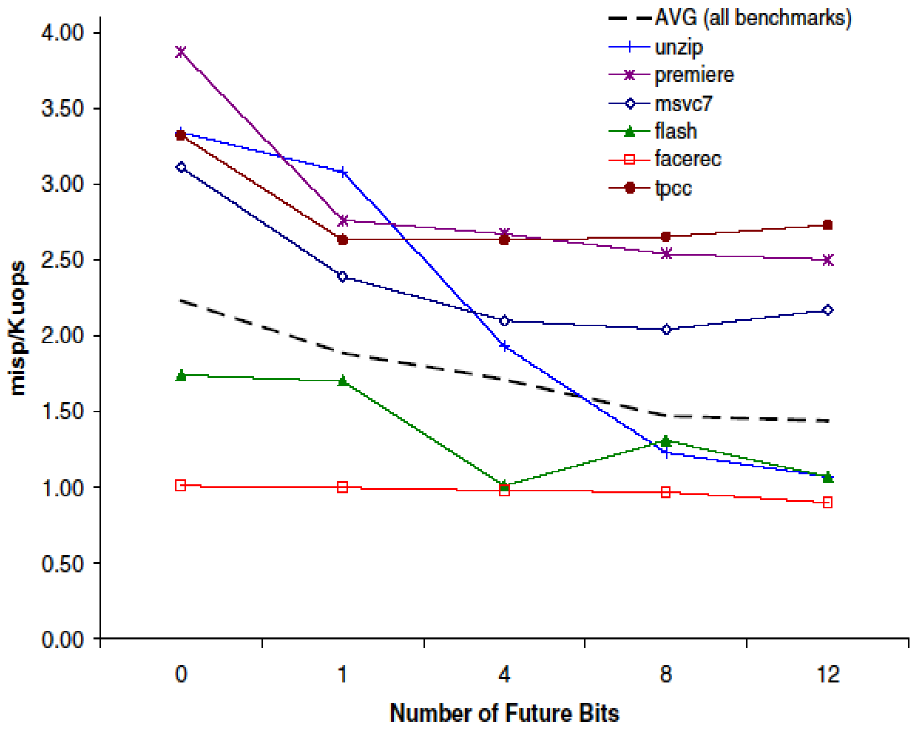
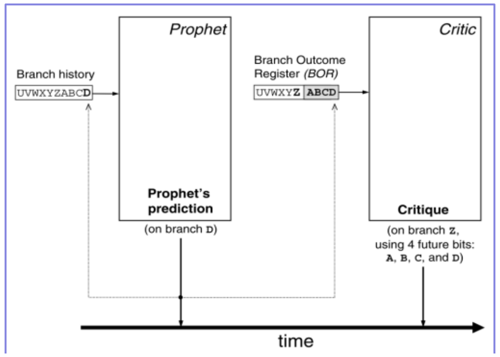


Figure 5. Effect of varying the number of future bits used by the critic on prediction accuracy for selected benchmarks. (prophet: 8KB perceptron; critic: 8KB tagged gshare)

Prophet-Critic Hybrid Branch Prediction

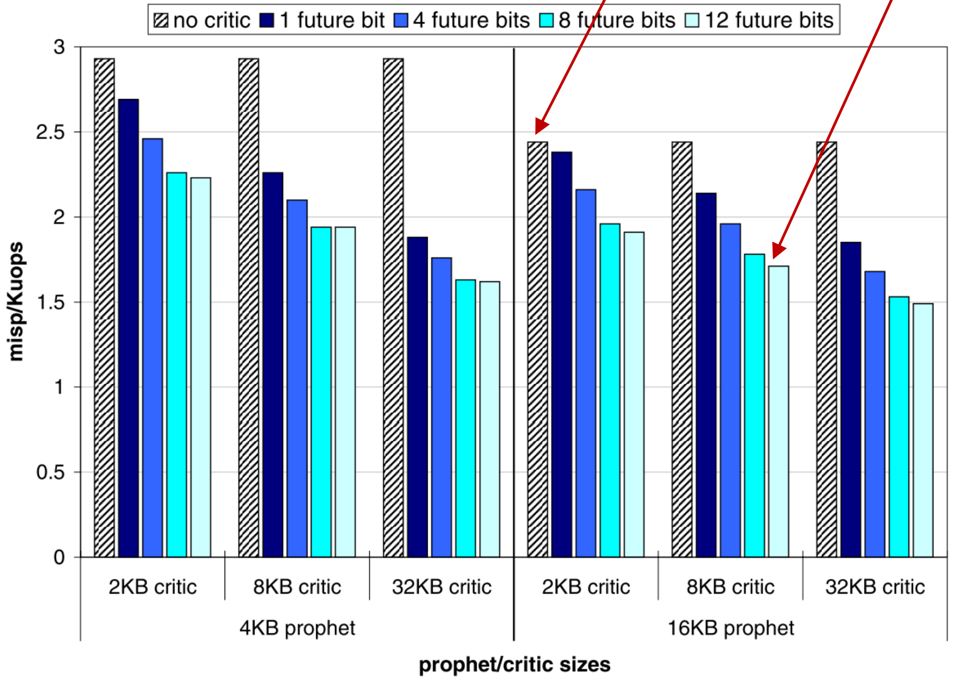


16KB gshare

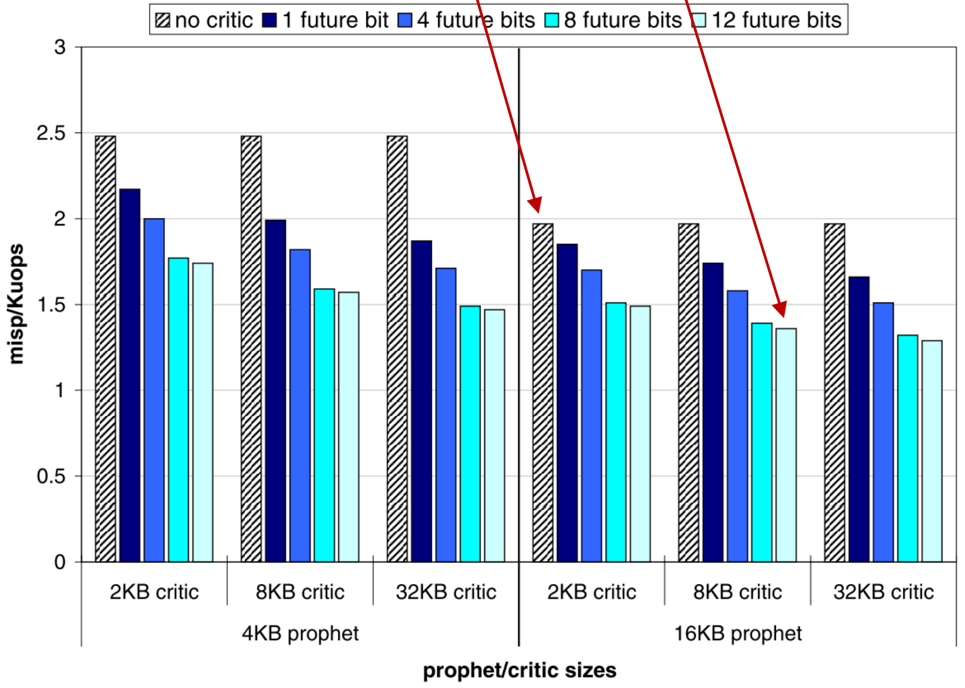
16KB gshare (prophet) +
8KB perceptron (critic)

16KB perceptron

16KB perceptron (prophet) +
8KB gshare (critic)



(b) Prophet: gshare; Critic: filtered perceptron



(c) Prophet: perceptron; Critic: tagged gshare