# ECE 4750 Computer Architecture

# Topic 10: Advanced Processors – Branch Prediction

`http://www.csl.cornell.edu/courses/ece4750`
School of Electrical and Computer Engineering
Cornell University

revision: 2025-12-01-14-37

## List of Problems

## Problem 1. Short Answer

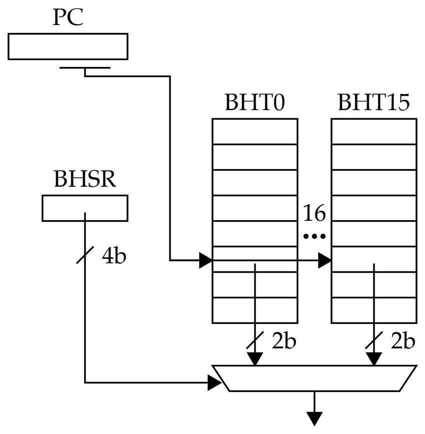
### Part 1.A Branch Prediction for Count Nonzeros

Consider the C code shown below which counts the number of non-zero elements in an array. This code will generate two branches: one forward branch to check if the element is non-zero (let's call this branch B1) and one backward branch for the loop (let's call this branch B2). *Assume that branch B1 is taken when aptr[i] is zero.* Part of the input array data is shown on the right and the rest of the array continues in a similar pattern.

```
int count_nonzeros( int* aptr, int size )
{
  int count = 0;
  for ( int i = 0; i < size; i++ )
    if ( aptr[i] != 0 )
      count++;
  return count;
}
```

```
int A[1000]
  = { 0, 1, 0, 2, 0, 3,
      0, 4, 0, 5, 0, 6,
      ... };
```

Consider the branch predictor shown below. This predictor is the same as one of the predictors we described in lecture; there is one predictor in the system that is used for all branches. Assume the entries in all BHTs contain two-bit saturating counter finite state machines and that they are initialized to the weakly taken state. Assume that there are enough entries in the BHT to avoid aliasing. **Fill in the table on the next page to illustrate how the branch predictor performs for branch B1.**



| | | **Branch B1** | | | | |
|---|---|---|---|---|---|---|
| `i` | `aptr[i]` | BHSR | BHT | Predicted | Actual | Correct? |
| 0 | 0 | 0000 | WT | T | T | y |
| 1 | 1 | | | | | |
| 2 | 0 | | | | | |
| 3 | 2 | | | | | |
| 4 | 0 | | | | | |
| 5 | 3 | | | | | |
| 6 | 0 | | | | | |
| 7 | 4 | | | | | |
| 8 | 0 | | | | | |
| 9 | 5 | | | | | |

**Estimate the misprediction rate for branch B1 over the entire loop:** ______________

**Answer these two multiple-choice questions and justify your answers below.**

| | | |
|---|---|---|
| Branch B1 exhibits: | temporal correlation | spatial correlation |
| This predictor is generally capable (i.e., not just for branch B1 but for any arbitrary branch) of exploiting: | temporal correlation | spatial correlation |

## Problem 2. Branch Prediction

Consider the C code shown in Figure 1. This function processes an input signal (stored in array `src`), which for example, might represent a sampled audio signal. The function performs two operations on the input signal to produce a processed output signal (stored in array `dest`). First, the function "saturates" the signal such that any input sample greater than the given `limit` is set to the limit value. Second, the function counts the number of non-zero input samples and returns this count.

The corresponding assembly code is shown in Figure 2. Assume that when the assembly code begins, the destination array pointer is stored in `x1`, the source array pointer is stored in `x2`, the size of the two arrays is stored in `x3`, the limit value is stored in `x4`, and the return value should ultimately be written to `x5`. Note that this function requires the following three branches:

- **Branch B0 –** Used to saturate the input signal. Branch is taken if `src[i]` $\leq$ `limit`.
- **Branch B1 –** Used to count number of non-zero input samples. Branch is taken if `src[i]` $= 0$.
- **Branch B2 –** Used to implement the loop.

In this problem, we will explore how three different branch predictors perform on this assembly code. For all parts, assume that there is no unwanted aliasing and that the branch predictors are updated immediately after the branch is predicted. This is obviously unrealistic, since we need to first resolve the branch before updating the predictor, but it simplifies our analysis. For all parts, `limit` is 10 and the input signal includes the following 20 samples:

```
0, 0, 12, 15, 0, 0, 11, 17, 0, 0, 11, 13, 9, 0, 12, 15, 0, 8, 12, 18
```

This means the `size` input argument is 20.

```
int process_signal( int dest[], int src[], int size, int limit )
{
  int num_non_zeros = 0;
  for ( int i = 0; i < size; i++ ) {
    dest[i] = src[i];      // Ensure samples in destination array
    if ( src[i] > limit )  // do not exceed limit
      dest[i] = limit;     //

    if ( src[i] > 0 )      // Count number of non-zero samples
      num_non_zeros++;     //
  }
  return num_non_zeros;
}
```

**Figure 1: C Code for Signal Processing Function**

```
  # x1 = dest_ptr, x2 = src_ptr, x3 = size, x4 = limit
  # x5 = return value

  addi x5, x0, 0        # num_non_zeros = 0
loop:

  # Ensure samples in destination array do not exceed limit

  lw   x6, 0(x2)        #  temp_a = *src_ptr
  addi x7, x6, 0        #  temp_b = temp_a
  slt  x8, x4, x6       #  if ( limit >= temp_a )
  beq  x8, x0, L1       #    goto L1                  Branch B0
  addi x7, x4, 0        #  temp_b = limit
L1:                     # L1:
  sw   x7, 0(x1)        #  *dest_ptr = temp_b

  # Count number of non-zero samples

  beq  x6, x0, L2       #  if ( temp_a == 0 ) goto L2   Branch B1
  addi x5, x5, 1        #  num_non_zeros++
L2:                     # L2:

  # Increment pointers and iterate

  addi x1, x1, 4        #  dest_ptr++
  addi x2, x2, 4        #  src_ptr++
  addi x3, x3, -1       #  size--
  blt  x0, x3, loop     #  if ( size > 0 ) goto loop    Branch B2
```

**Figure 2: Assembly Code for Signal Processing Function**

**Part 2.A Two-Bit Saturating Counter Branch History Table**

We begin with a basic two-bit saturating counter finite-state machine shown in Figure 3 implemented with the simple one-level branch history table (BHT) shown in Figure 4. There are four states in the finite-state machine:

- **Strongly Taken –** (ST) predict taken
- **Weakly Taken –** (WT) predict taken
- **Weakly Not-Taken –** (WNT) predict non-taken
- **Strongly Not-Taken –** (SNT) predict not-taken

Assume that all entries in the BHT are initialized to the weakly taken (WT) state.

**Create a table similar to the one shown in Figure 5.** There should be twenty rows, one for each iteration of the loop. There are columns for the branch predictor state, prediction (column labelled P), and actual resolution (column labelled A) for each of the three branches. In this table, the branch predictor state should always reflect the state of the predictor that is used to make the prediction (i.e., before we update the state for that branch resolution). So for example, the BHT entry for branch B0 is initially WT and thus we predict taken for the first iteration of the loop when `src[0]` is zero. The branch is indeed taken, and thus we update the BHT entry for branch B0 to strongly taken (ST) as indicated by the BHT entry for the second iteration. **Calculate the branch predictor accuracy for each of the three branches individually, and then also for the entire function.** *Hint: Fill in the actual resolution column for all rows and all branches first, then fill in the BHT state, and finally fill in the prediction column.*

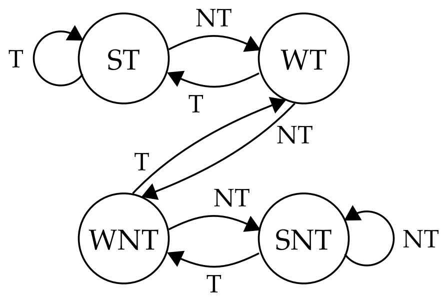


**Figure 3: Two-Bit Saturating Counter FSM**

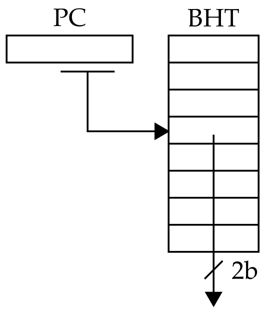


**Figure 4: BHT Implementation**

| | | Branch B0 | | | Branch B1 | | | Branch B2 | | |
|---|---|---|---|---|---|---|---|---|---|---|
| `i` | `src[i]` | **BHT** | **P** | **A** | **BHT** | **P** | **A** | **BHT** | **P** | **A** |
| 0 | 0 | WT | T | T | | | | | | |
| 1 | 0 | ST | T | T | | | | | | |
| 2 | 12 | ... | | | | | | | | |
| ... | | | | | | | | | | |

**Figure 5: Two-Bit Saturating Counter BHT Execution**

## Part 2.B Two-Level Adaptive Branch Predictor to Exploit Temporal Correlation

The input signal has interesting temporal correlation that we may be able to capture with a more sophisticated two-level adaptive branch predictor. Consider the two-level BHT shown in Figure 6. In this predictor, we use a branch history shift register table (BHSRT) to capture a local branch history pattern for each branch. The patterns are used to choose between one of several BHTs. Each entry in the BHT is a two-bit saturating counter initialized to the weakly taken (WT) state. For this problem, assume that each entry in the BHSRT is three bits. This means that we need eight BHTs. Assume that all entries in the BHSRT are initialized to zero.

**Create a table similar to the one shown in Figure 7. Calculate the branch predictor accuracy for each of the three branches individually, and then also for the entire function.** *Hint: Fill in the actual resolution column for all rows and all branches first (should be the same as in the previous part), then fill in the BHSRT state, then fill in the BHT state, and finally fill in the prediction column.*

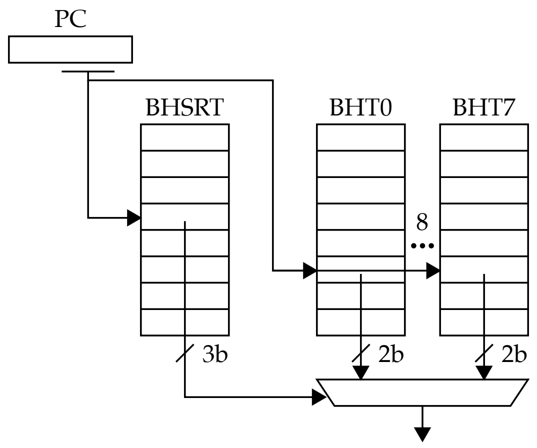


**Figure 6: Two-Level BHT for Temporal Correlation**

| | | Branch B0 | | | | Branch B1 | | | | Branch B2 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `i` | `src[i]` | **BHSRT** | **BHT** | **P** | **A** | **BHSRT** | **BHT** | **P** | **A** | **BHSRT** | **BHT** | **P** | **A** |
| 0 | 0 | 000 | WT | T | T | | | | | | | | |
| 1 | 0 | 001 | WT | T | T | | | | | | | | |
| 2 | 12 | ... | | | | | | | | | | | |
| ... | | | | | | | | | | | | | |

**Figure 7: Two-Level BHT for Temporal Correlation Execution**

**Part 2.C Two-Level Adaptive Branch Predictor to Exploit Spatial Correlation**

The branches in the assembly sequence have spatial correlation that we may be able to capture with a more sophisticated two-level adaptive branch predictor. Consider the two-level BHT shown in Figure 8. In this predictor, we use a single branch history shift register (BHSR) to capture a global branch history pattern across all branches. This pattern is used to choose between one of several BHTs. Each entry in the BHT is a two-bit saturating counter initialized to the weakly taken (WT) state. For this problem, assume that the BHSR is only one bit. This means that we need two BHTs. Assume that the BHSR is initially zero.

**Create a table similar to the one shown in Figure 9. Calculate the branch predictor accuracy for each of the three branches individually, and then also for the entire function.** *Hint: Fill in the actual resolution column for all rows and all branches first (should be the same as in the previous parts), then fill in the BHSR state, then fill in the BHT state, and finally fill in the prediction column.*

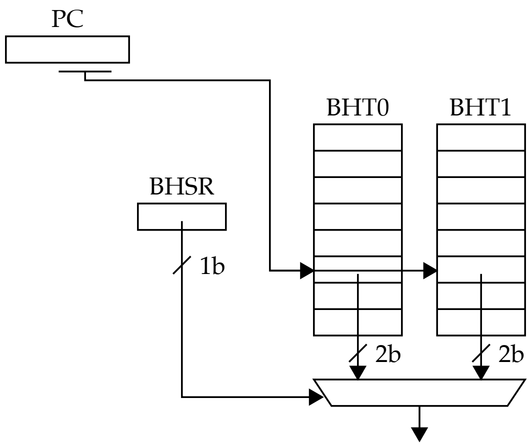


**Figure 8: Two-Level BHT for Spatial Correlation**

| | | **Branch B0** | | | | **Branch B1** | | | | **Branch B2** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `i` | `src[i]` | **BHSR** | **BHT** | **P** | **A** | **BHSR** | **BHT** | **P** | **A** | **BHSR** | **BHT** | **P** | **A** |
| 0 | 0 | 0 | WT | T | T | 1 | WT | T | T | 1 | WT | T | T |
| 1 | 0 | 1 | WT | T | T | 1 | ST | T | T | | | | |
| 2 | 12 | ... | | | | | | | | | | | |
| ... | | | | | | | | | | | | | |

**Figure 9: Two-Level BHT for Spatial Correlation Execution**

**Part 2.D Branch Predictor Comparison**

Create a table similar to the one in Figure 10 to record the branch prediction accuracies for all three branch predictors. For each predictor and each branch, discuss why the accuracy is better or worse than the other predictors on the same branch. Your answer should reflect your understanding of the temporal and spatial correlation in this example and how it impacts the various prediction accuracies.

| | **Two-Bit FSM Accuracy** | **Two-Level Temporal Accuracy** | **Two-Level Spatial Accuracy** |
|---|---|---|---|
| **Branch B0** | | | |
| **Branch B1** | | | |
| **Branch B2** | | | |
| **All Branches** | | | |

**Figure 10: Summary of Branch Predictor Accuracies**

## Appendix A: TinyRV1 Canonical Microarchitectures

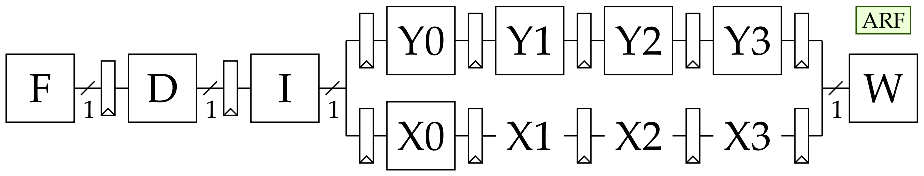


**Figure A.1: I3L Microarchitecture for MUL, ADDU, ADDIU**

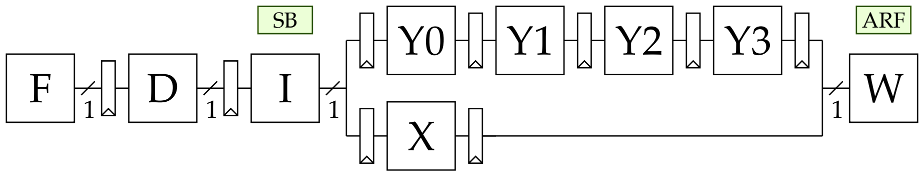


**Figure A.2: I2OE Microarchitecture for MUL, ADDU, ADDIU**

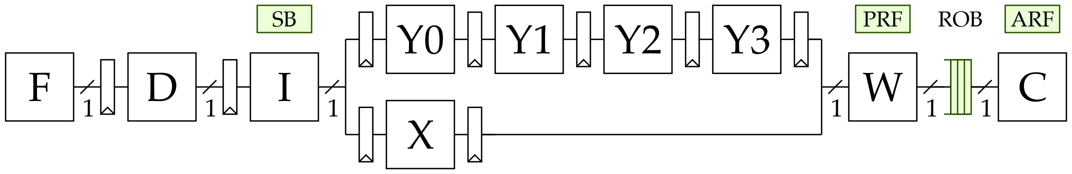


**Figure A.3: I2OL Microarchitecture for MUL, ADDU, ADDIU**

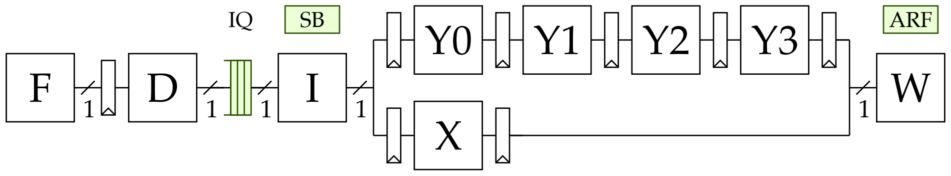


**Figure A.4: IO2E Microarchitecture for MUL, ADDU, ADDIU**

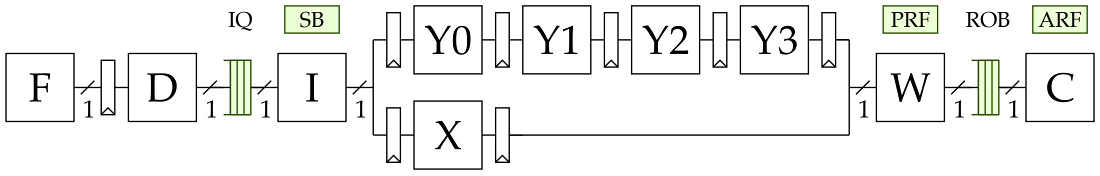


**Figure A.5: IO2L Microarchitecture for MUL, ADDU, ADDIU**

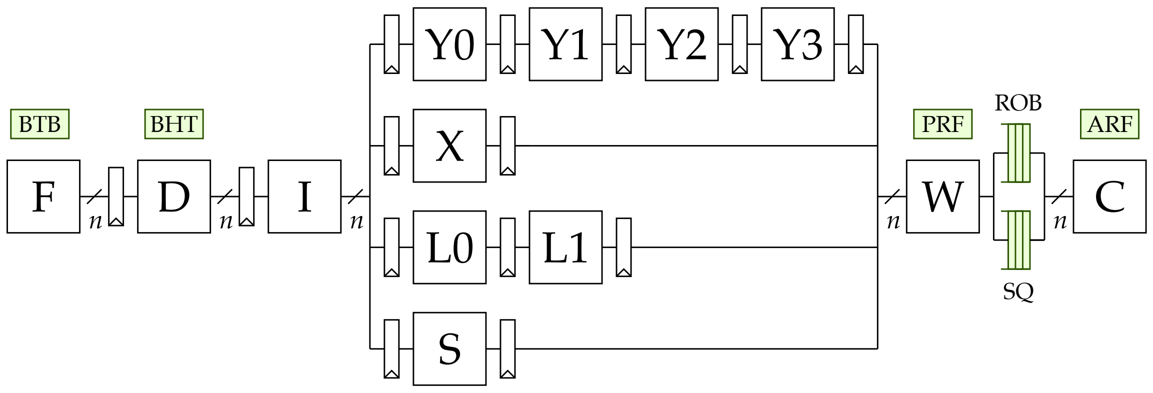


**Figure A.6: Complete I2OL Microarchitecture (single issue: *n* = 1; quad issue: *n* = 4)**

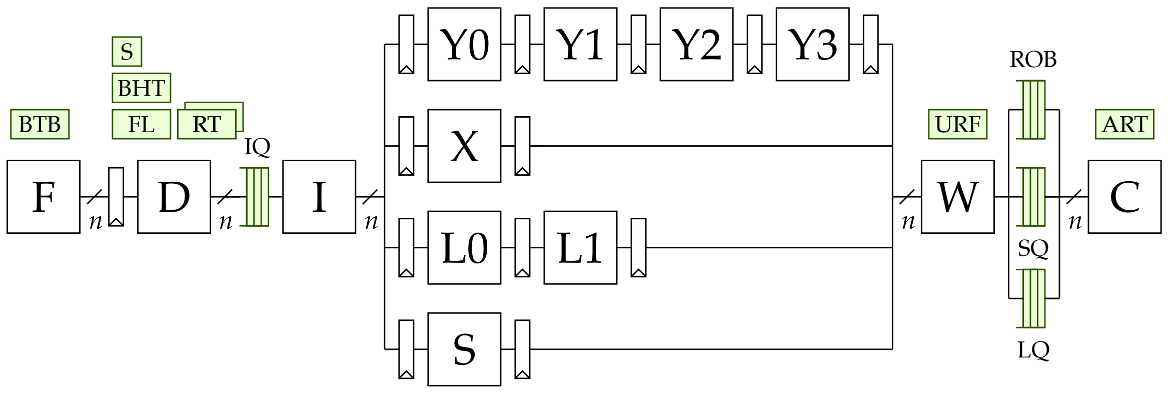


**Figure A.7: Complete IO2L Microarchitecture (single issue: *n* = 1; quad issue: *n* = 4)**