

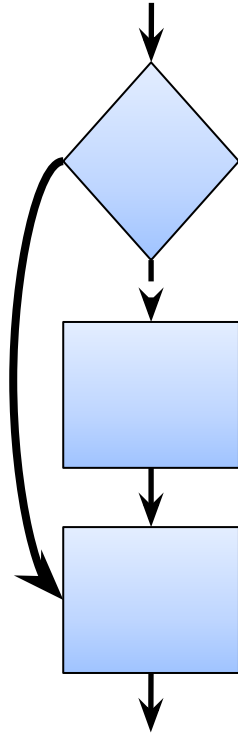
Perceptron Branch Predictor Simulator

CMPUT 229 lab 6

Background:

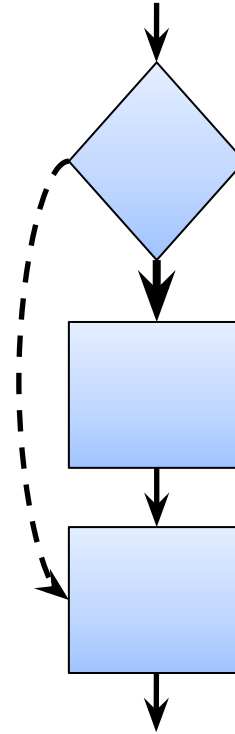
Static Predictions VS Dynamic
Predictions

Static Predictions



Always
Taken

OR



Always
Not Taken

Static Predictors

- Simple design
- Not very accurate
- Cannot learn and adapt predictions based on program execution behavior



Dynamic Predictors

- Prediction of a given branch changes with the execution of the program.
 - **Simple:** a finite-state machine encodes the outcome of a few recent executions of the branch.
 - **Elaborate:** Not only early branch outcomes, but other correlated parts of the programs are considered.

Predicting Direction

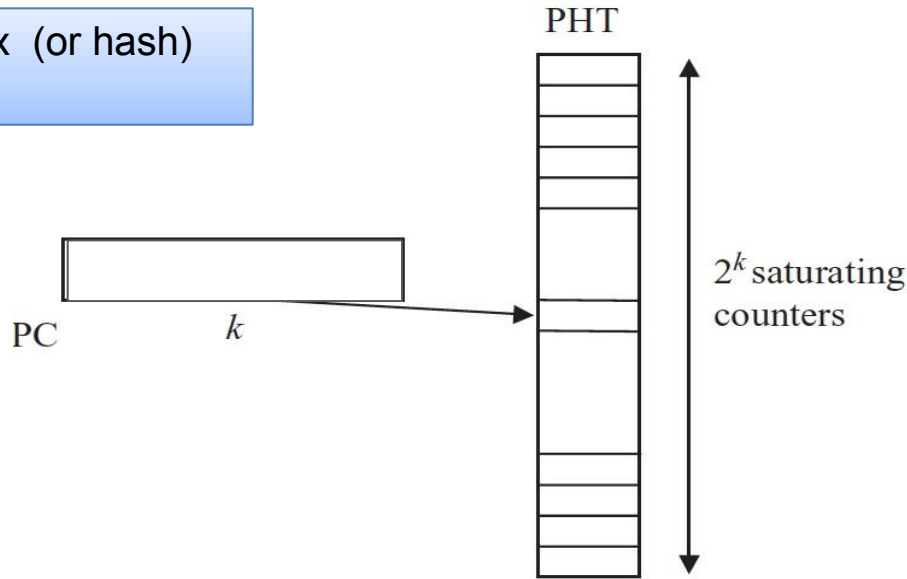
- Where we find the prediction?
Look at the recent past: What was the direction the last time this same branch was executed?
- How to encode the prediction?
A single bit encodes the prediction:
Prediction bit is set at prediction time.

1-Bit counter Dynamic Predictor

Table Entry	Branch Address (or tag)	Predictor State
0	0xFF80 0004	0
1	0xC340 00F8	1
	...	
n	0x0004 0000	1

Counters Stored in Pattern History Table (PHT)


Use PC to index (or hash) the PHT.



Each entry of the PHT stores the state of the counter associated with a branch.

1-Bit counter Dynamic Predictor Example

PC	1-bit counter	Prev. Outcome	Prediction	Actual Outcome
0x00400030	1	Taken	Taken	Taken ✓
0x00400080	0	Not Taken	Not Taken	Not Taken ✓
0x0040008c	0	Not Taken	Not Taken	Taken ✗ (mispred.)



After the actual result of the branch is determined, update the saturating counter.

1-Bit counter Dynamic Predictor Example

PC	1-bit counter	Prev. Outcome	Prediction	Actual Outcome
0x00400030	1	Taken	Taken	Taken ✓
0x00400080	0	Not Taken	Not Taken	Not Taken ✓
0x0040008c	1	Taken		

↑
This branch will be predicted taken the next time it is fetched.

1-Bit counter Predictor Summary

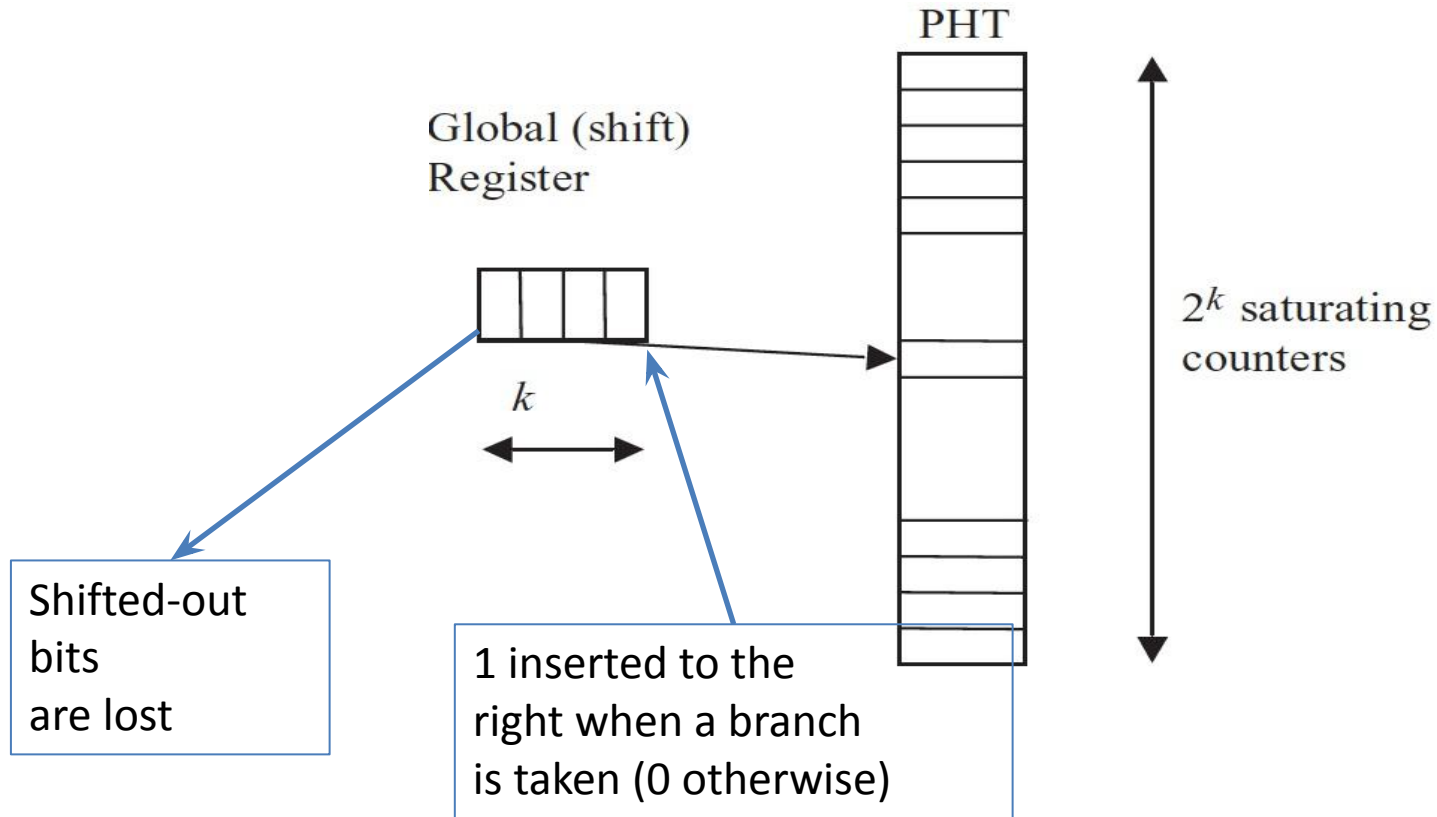
- The PC of a branch is used to index into a table of counters (PHT)
- Each counter indicates whether that branch should be predicted Taken or Not Taken
- The outcome of the branch depends on its previous outcome
- This is known as a **Local** dynamic predictor

Local vs Global Predictors

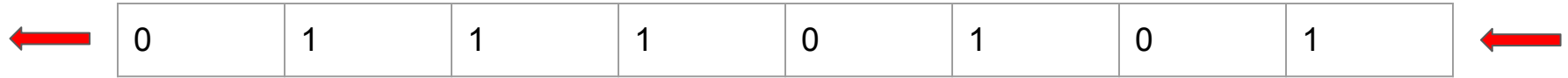
Local: Predictions for a branch depend only on the past behaviour of that branch.

Global: Predictions for a branch depend on the behaviour of other branches.

Global Predictor



8-Bit Global Shift Register (GSR)

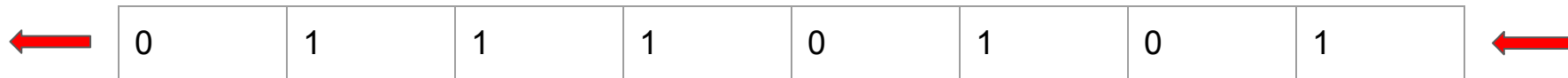


Oldest branch
outcome shifted
out

1 = Taken
0 = Not Taken

Most recent
branch outcome
shifted in

8-Bit Global Shift Register (GSR)



Tracks the history of the 8 most recent branches

Global Predictor Summary

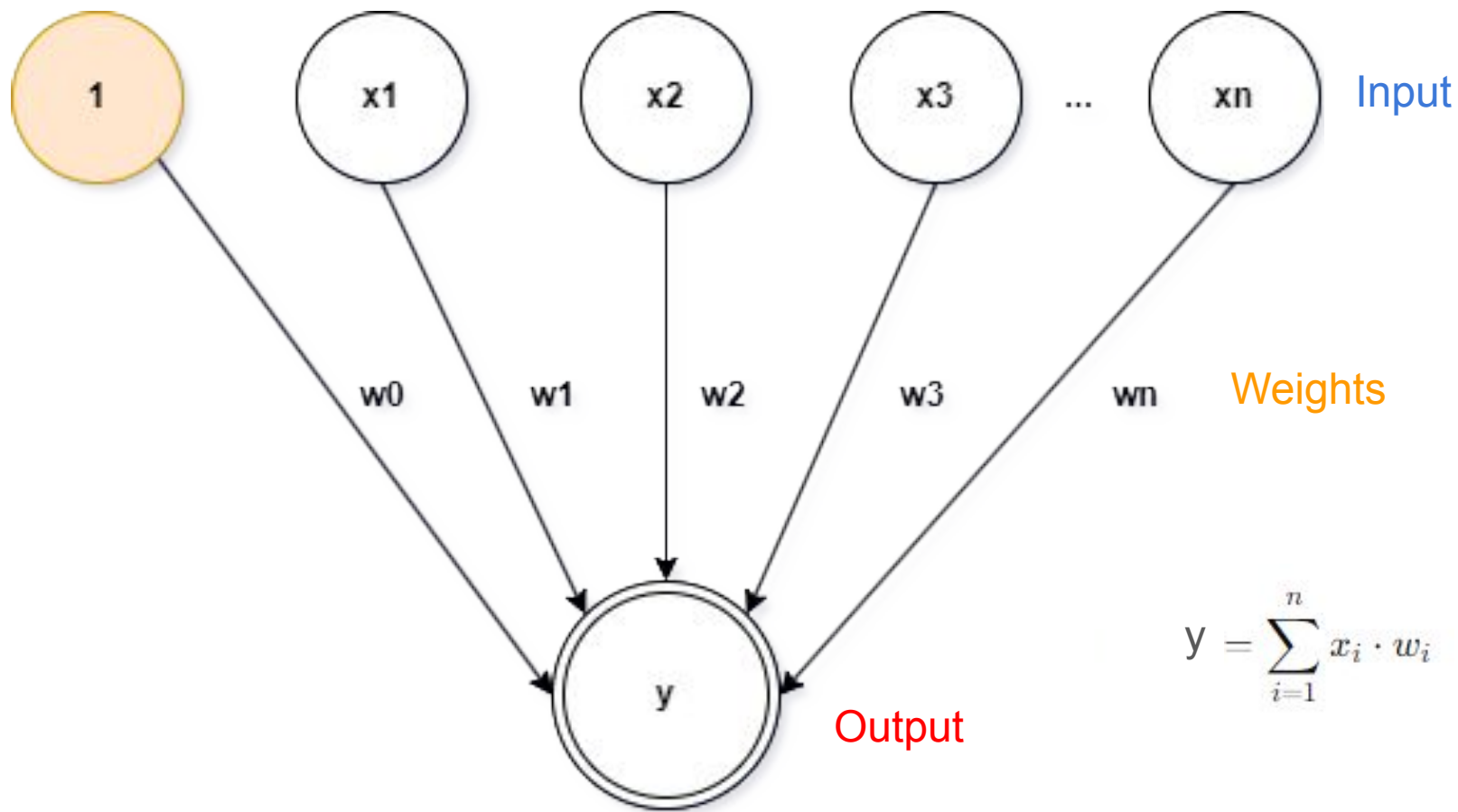
- The bits of the global shift register are used to index into a table of counters (PHT)
- Each counter indicates whether the current branch should be predicted Taken or Not Taken
- The outcome of the branch depends on global branch history
- Predictions for current branch are independent of its PC

Perceptron Branch Predictor

- Uses both local and global branch history information to make predictions
- Uses simple binary classification to learn complex branch patterns and correlations
- Increased accuracy in many applications

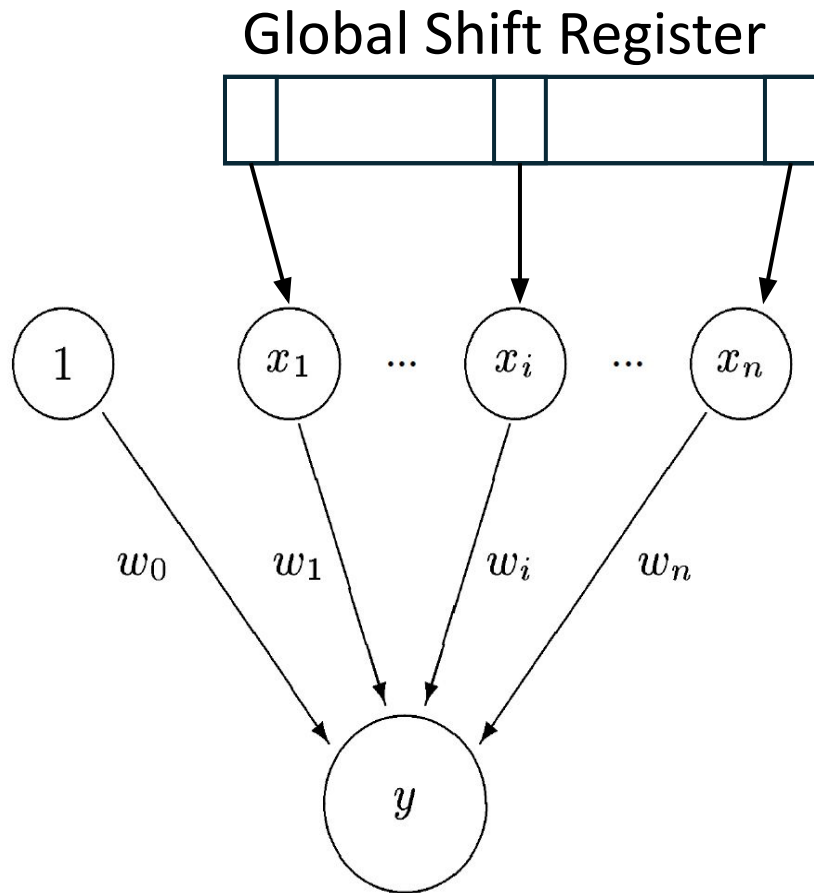
What is a Perceptron?

- One of the simplests forms of neural networks
- Binary classifier
- A set of weights, where each weight is a measure of importance that each input value has to the output

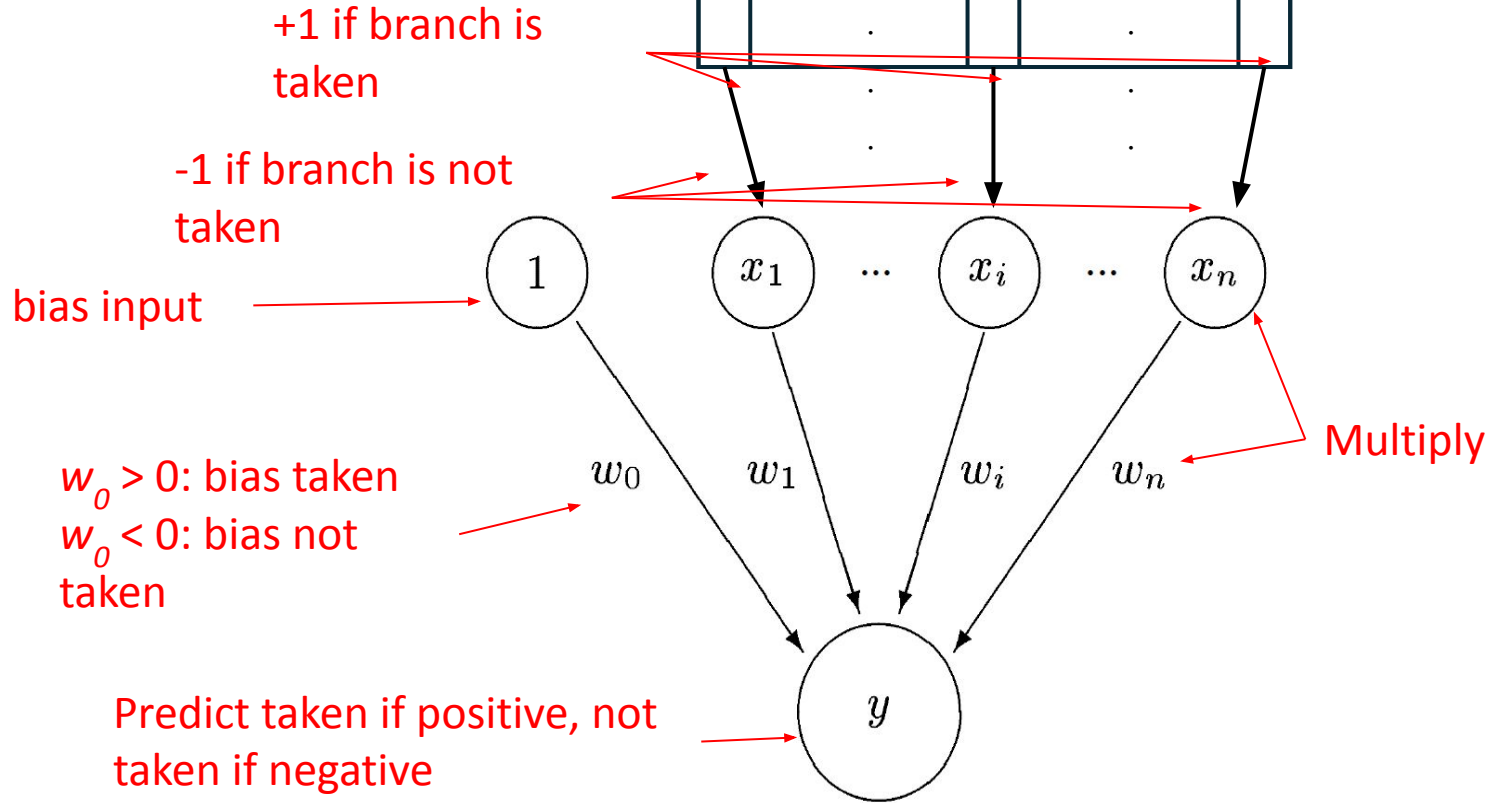


GSR Used for Perceptron Predictor

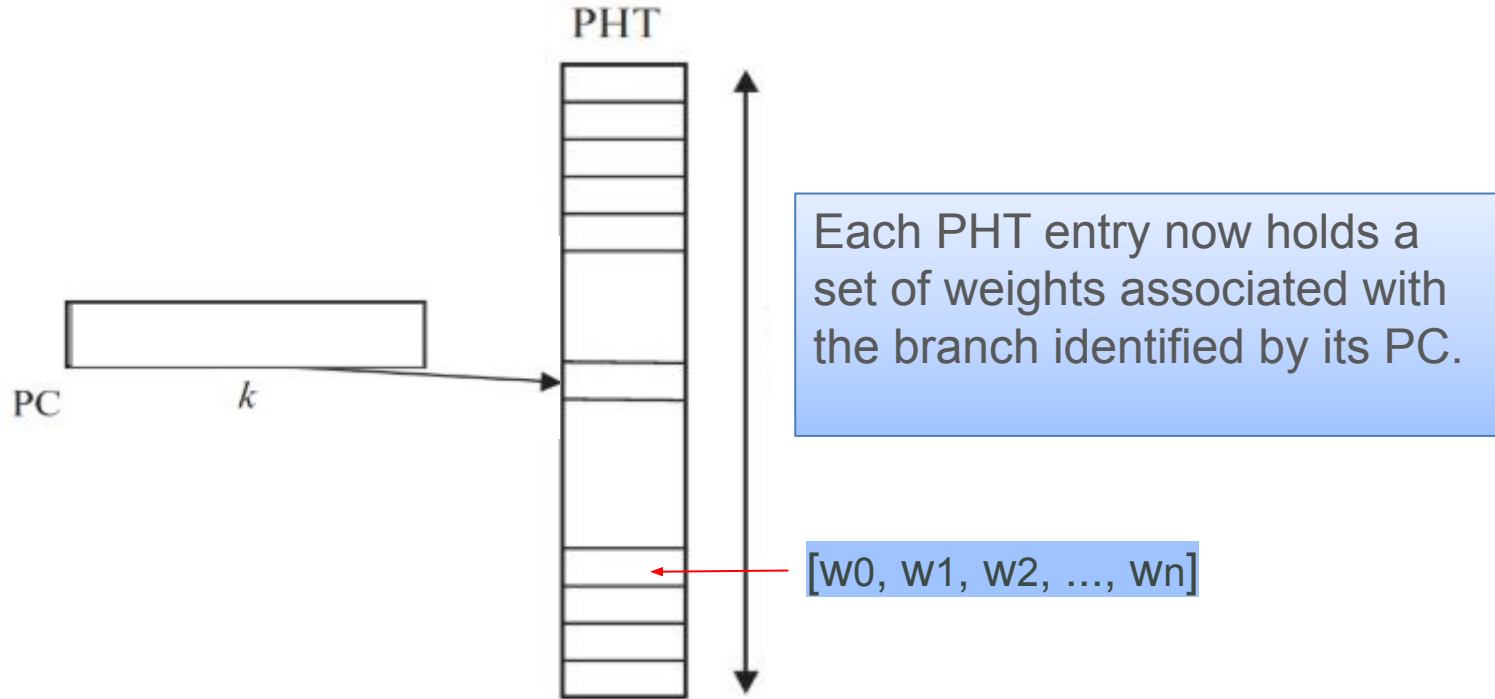
Global shift register
bits are the input
values



Global Shift Register

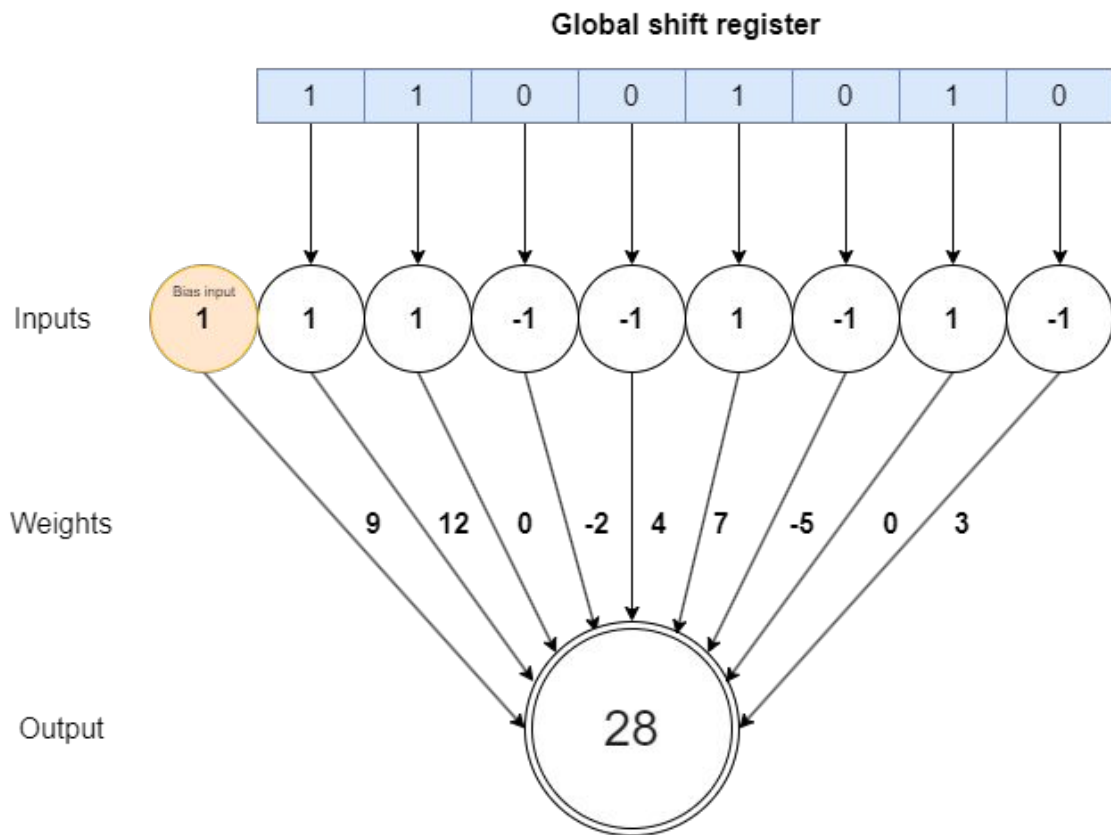


PHT for Perceptron Predictor



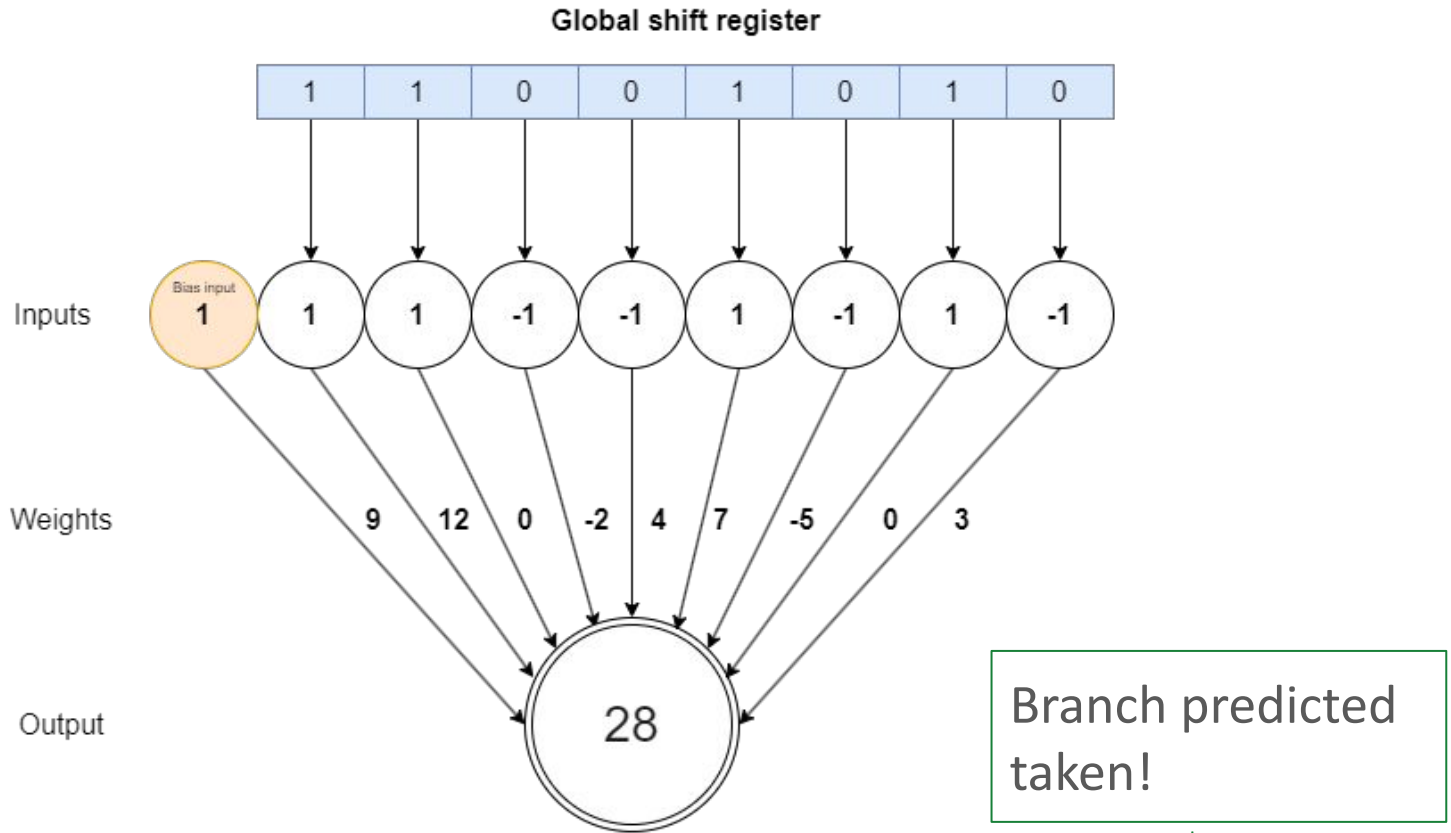
Making a Prediction

Dot product: $\mathbf{x} \cdot \mathbf{w} = \sum_{i=1}^n x_i \cdot w_i$



$$y = x \cdot w = x_1 \cdot w_1 + x_2 \cdot w_2 + \dots + x_n \cdot w_n$$

$$y = (1)(9) + (1)(12) + (1)(0) + (-1)(-2) + (-1)(4) + (1)(7) + (-1)(-5) + (1)(0) + (-1)(3) = 28$$



$$y = x \cdot w = x_1 \cdot w_1 + x_2 \cdot w_2 + \dots + x_n \cdot w_n$$

$$y = (1)(9) + (1)(12) + (1)(0) + (-1)(-2) + (-1)(4) + (1)(7) + (-1)(-5) + (1)(0) + (-1)(3) = 28$$

Training a Perceptron

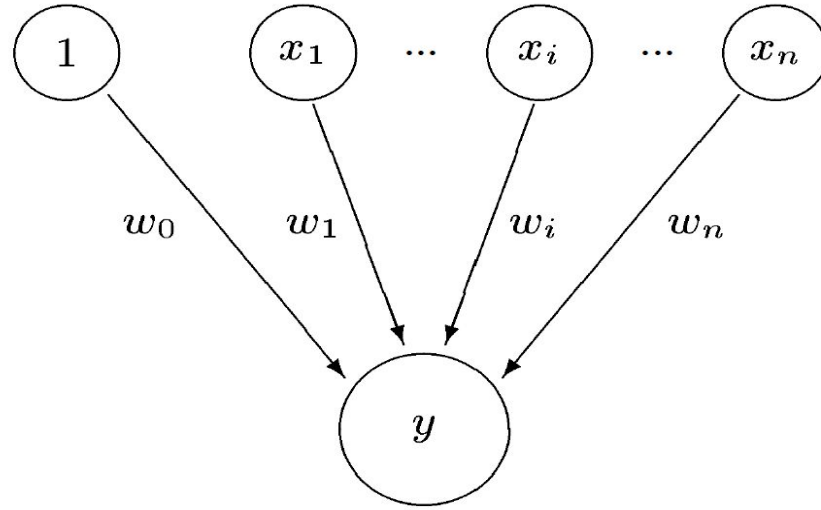
```
if  $\text{sign}(y) \neq t$  or  $|y| \leq \theta$  then  
    for  $i := 0$  to  $n$  do  
         $w_i := w_i + tx_i$   
    end for  
end if
```

Update weights after
actual outcome of
branch is determined

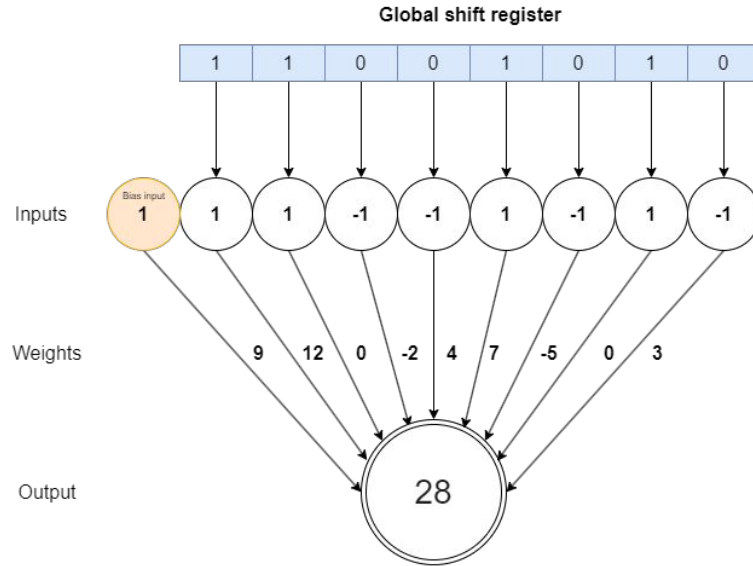
$$w_i = w_i + tx_i$$

Actual Outcome of the
Branch:

$$t \equiv \begin{cases} 1 & \text{if branch was taken} \\ -1 & \text{if branch was not taken} \end{cases}$$



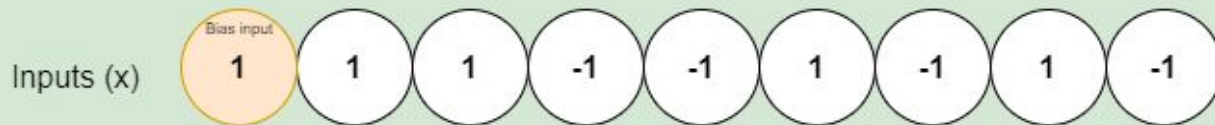
The perceptron weights are updated based on the actual branch outcome



$$y = x \cdot w = x_1 \cdot w_1 + x_2 \cdot w_2 + \dots + x_n \cdot w_n$$

$$y = (1)(9) + (1)(12) + (1)(0) + (-1)(-2) + (-1)(4) + (1)(7) + (-1)(-5) + (1)(0) + (-1)(3) = 28$$

Case 1: Branch is actually taken ✓



(t)(xi)

1	1	1	-1	-1	1	-1	1	-1
---	---	---	----	----	---	----	---	----

Original Weights

9	12	0	-2	4	7	-5	0	3
---	----	---	----	---	---	----	---	---

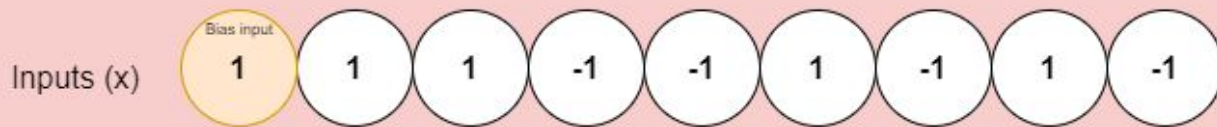
Updated Weights

10	13	1	-3	3	8	-6	1	2
----	----	---	----	---	---	----	---	---

$$w_i = w_i + tx_i$$

Branch actually taken, $t=1$

Case 2: Branch is actually not taken ❌



(t)(xi)

-1	-1	-1	1	1	-1	1	-1	1
----	----	----	---	---	----	---	----	---

Original Weights

9	12	0	-2	4	7	-5	0	3
---	----	---	----	---	---	----	---	---

Updated Weights

8	11	-1	-1	5	6	-4	-1	4
---	----	----	----	---	---	----	----	---

$$w_i = w_i + tx_i$$

Branch actually not taken, $t = -1$

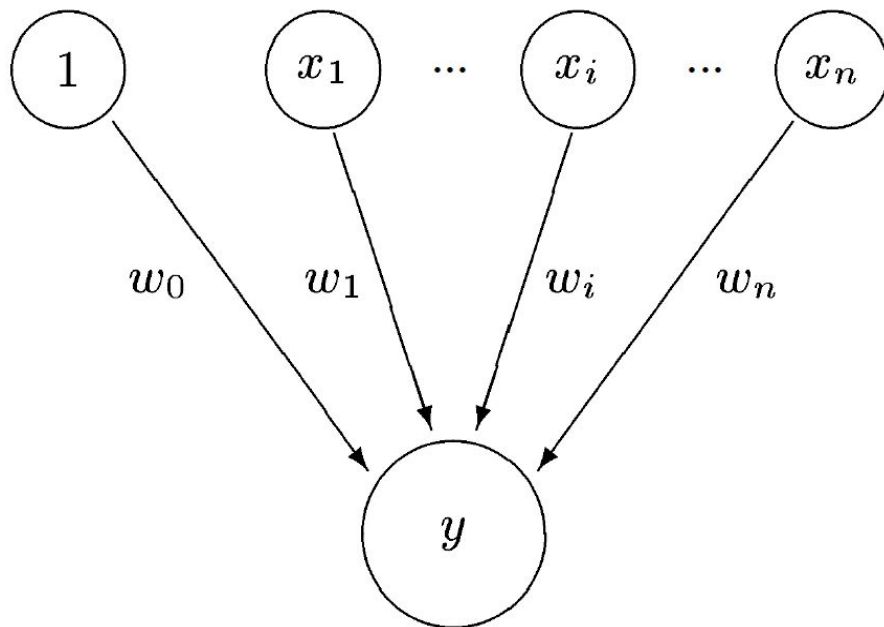
When to Stop Training?

Intuition

When the value y becomes high,
the branch has a strong bias.
No need for further training.

Stop training is $|y| > \Theta$.

Threshold Θ



Perceptron branch predictor in RARS

Similar to the the coding question on the midterm,
perceptron_predictor.s reads and parses the binary of
another program provided as an argument.

perceptron_predictor.s will predict the branches in the input program, and train its perceptrons on the actual branch outcomes.

Unlike an actual perceptron branch predictor, the one in this lab will not use the PC of a branch to access the PHT. Instead each branch from the input program will be assigned a static branch Id.

demo:

addi sp, sp, -20

sw s0, 0(sp)

sw s1, 4(sp)

sw s2, 8(sp)

sw s3, 12(sp)

sw ra, 16(sp)

addi s0, zero, 25

addi s1, zero, 0

addi s2, zero, 0

bge zero s0, done

loop:

andi s3, s1, 1

bne s3, s1, label1

add s2, s2, s1

j label2

label1:

beq s0, s2, label2

addi s2, s2, 1

label2:

addi s1, s1, 1

blt s1, s0, loop

Input Program

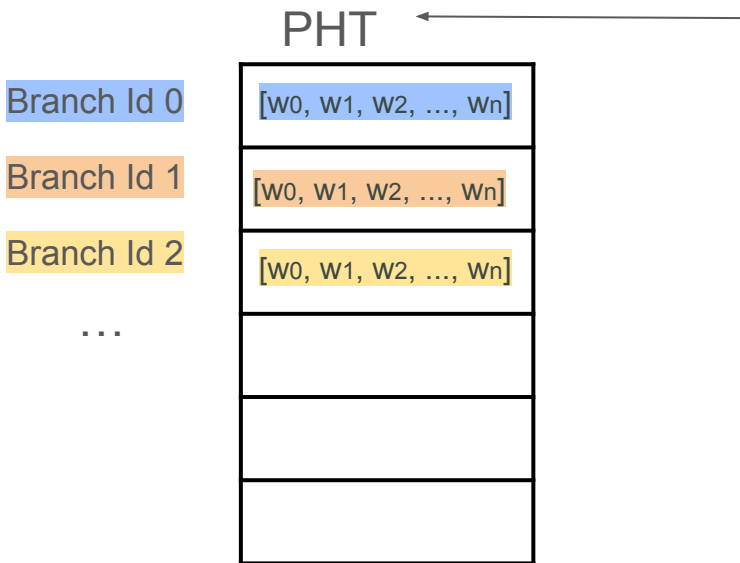
Branch Id 0

Branch Id 1

Branch Id 2

Branch Id 3

PHT for the Lab



The PHT is represented as an array of words in this lab, indexed by branch Id.

makePrediction

Description:

Makes a prediction on whether a particular branch will be taken or not taken, given the program state and branch Id

Arguments:

a0: Branch id

Returns:

None

trainPredictor

Description:

Trains the perceptron corresponding to the branch id stored in **activeBranch**.

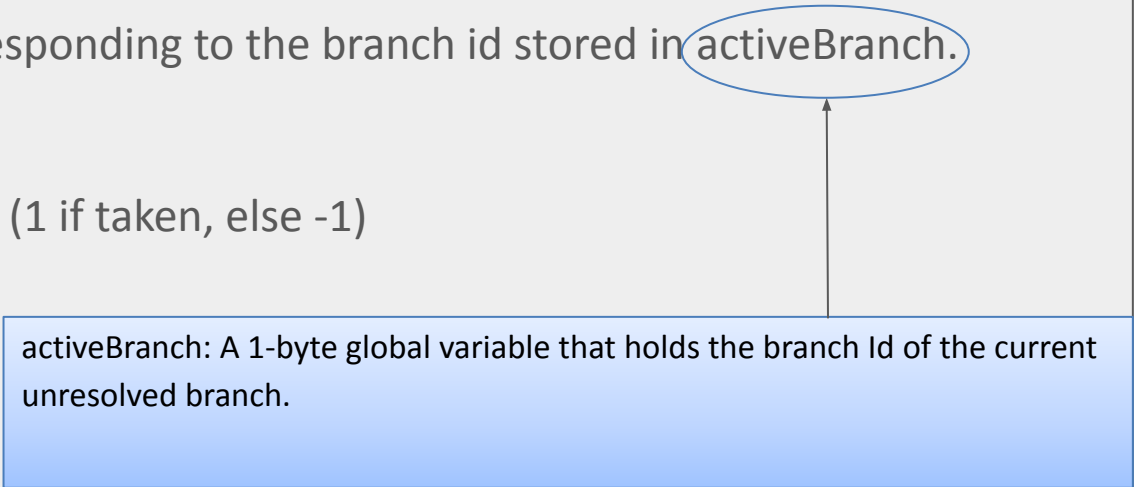
Arguments:

a0: Actual branch outcome (1 if taken, else -1)

Returns:

None

activeBranch: A 1-byte global variable that holds the branch Id of the current unresolved branch.

A blue rectangular callout box with a thin blue border is positioned at the bottom right of the slide. It contains the text 'activeBranch: A 1-byte global variable that holds the branch Id of the current unresolved branch.' A thin black arrow points vertically upwards from the top center of this box to the word 'activeBranch' in the description text above.

Instrumentation

Unlike the midterm question,
perceptron_predictor.s **must modify the binary of the input program, then begin executing the instructions of the input program.**

Why modify the input file's binary?

The modifications to the input binary will include inserting jumps to the functions makePrediction and trainPredictor, before/after each branch.

demo:

```
addi sp, sp, -20
sw s0, 0(sp)
sw s1, 4(sp)
sw s2, 8(sp)
sw s3, 12(sp)
sw ra, 16(sp)
```

```
addi s0, zero, 25
addi s1, zero, 0
addi s2, zero, 0
```

```
[setup(0)]
bge zero s0, done
[resolve(-1)]
```

The setup and resolve instructions have been inserted into the binary by `perceptron_predictor.s`

setup for branch Id 0

resolve fallthrough

loop:

```
[resolve(1)]
andi s3, s1, 1
[setup(1)]
bne s3, s1, label1
[resolve(-1)]
add s2, s2, s1
j label2
```

resolve target

setup for branch Id 1

resolve fallthrough

label1:

```
[resolve(1)]
[setup(2)]
beq s0, s2, label2
[resolve(-1)]
addi s2, s2, 1
```

resolve target

setup for branch Id 2

resolve fallthrough

demo:

```
addi sp, sp, -20
sw s0, 0(sp)
sw s1, 4(sp)
sw s2, 8(sp)
sw s3, 12(sp)
sw ra, 16(sp)
```

```
addi s0, zero, 25
addi s1, zero, 0
addi s2, zero, 0
[setup(0)]
bge zero s0, done
[resolve(-1)]
```

setup for branch Id 0

resolve fallthrough

loop:

```
[resolve(1)]
andi s3, s1, 1
[setup(1)]
bne s3, s1, label1
[resolve(-1)]
add s2, s2, s1
j label2
```

resolve target

setup for branch Id 1

resolve fallthrough

label1:

```
[resolve(1)]
[setup(2)]
beq s0, s2, label2
[resolve(-1)]
addi s2, s2, 1
```

resolve target

setup for branch Id 2

resolve fallthrough

setup(branch id):

- Calls makePrediction

resolve(-1):

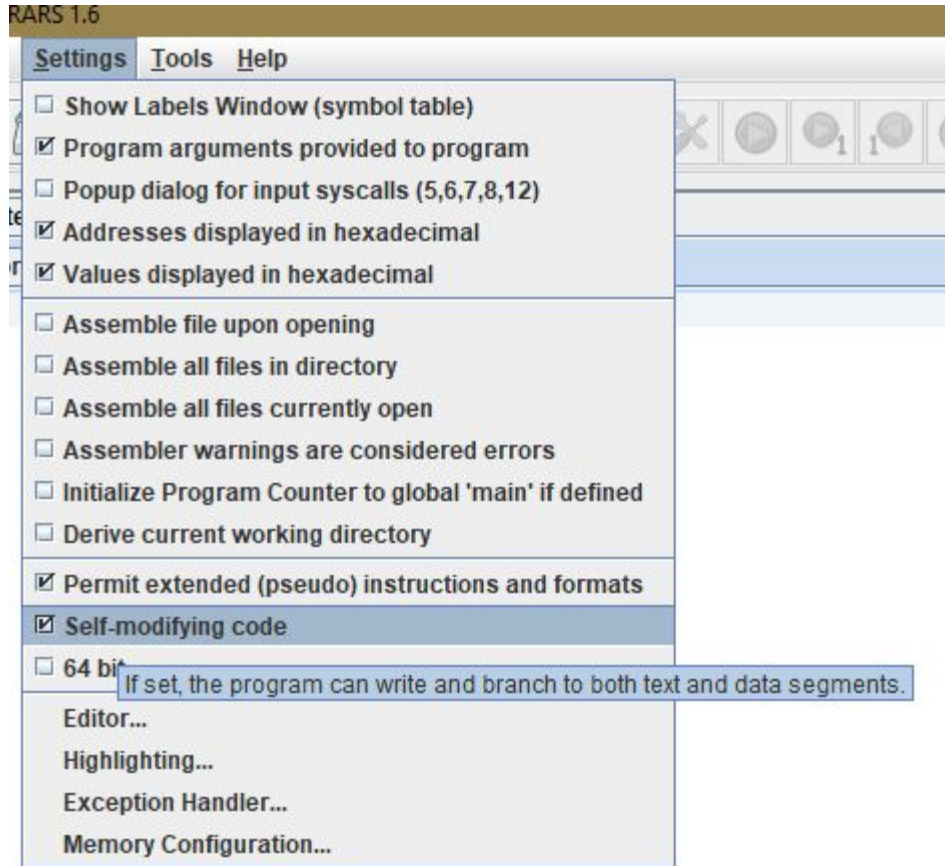
- Calls trainPredictor to indicate that the setup branch was not taken.

resolve(1):

- Calls trainPredictor to indicate the setup branch was actually taken.

How to modify the input binary
and execute it within
perceptron_predictor.s?

Self Modifying Code



With the self-modifying code feature enabled, your program can jump to the starting address of the input program's modified binary, and begin executing those instructions.

➡ `modifiedInstructionsArray: .space 6000`

Required Data Structures

globalShiftRegister: This is a 1-byte global variable. Each bit in globalShiftRegister corresponds to a previously taken branch.

patternHistoryTable: This global variable is an array of words that simulates a pattern history table. Each element is a pointer to the 9-byte long array of perceptron weights for the corresponding branch id.

instructionIndicatorsArray: A byte array where each byte acts as an indicator that a specific sequence of instructions must be inserted into modifiedInstructionsArray.

numPriorInsertionsArray: An array of 32-bit words, where each word specifies the total number of instructions to insert **before** the corresponding instruction in the originalInstructionsArray.

`originalInstructionsArray`: An array that contains every instruction from the input function.

`modifiedInstructionsArray`: An array that contains the input function after the insertion of the instrumentation function calls.

Functions to Complete

perceptronPredictor

Description:

The primary function that initiates the instrumentation and execution stages. This function must modify the input binary, train and run the predictor, then print the results.

Arguments:

a0: Pointer to originalInstructionsArray
a1: Pointer to modifiedInstructionsArray
a2: Pointer to instructionIndicatorsArray
a3: Pointer to numPriorInsertionsArray

Returns:

None

fill_instructionIndicatorsArray

Description:

This function is responsible for filling instructionIndicatorsArray.

Arguments:

a0: Pointer to originalInstructionsArray

a1: Pointer to instructionIndicatorsArray

Returns:

None

fill_modifiedInstructionsArray

Description:

This function is responsible for filling numPriorInsertionsArray.

Arguments:

a0: Pointer to instructionIndicatorsArray

a1: Pointer to numPriorInsertionsArray

Returns:

None

makePrediction

Description:

Makes a prediction on whether a particular branch will be taken or not taken, given the program state and branch Id

Arguments:

a0: Branch id

Returns:

None

trainPredictor

Description:

Trains the perceptron corresponding to the branch id stored in activeBranch.

Arguments:

a0: Actual branch outcome (1 if taken, else -1)

Returns:

None