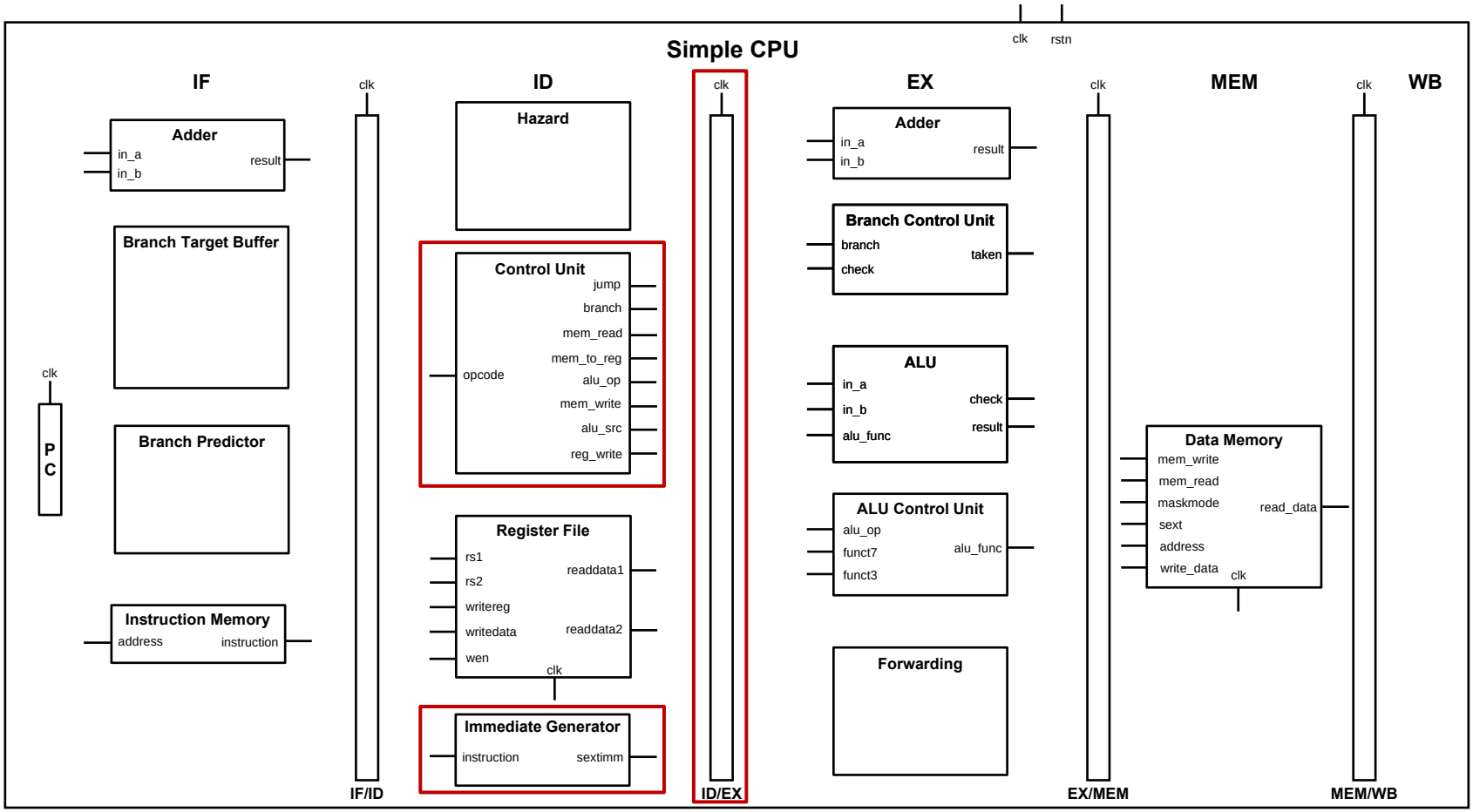
Computer Architecture

Lab 3: Branch Prediction Hardware Implementation

Jaewoong Sim
Electrical and Computer Engineering
Seoul National University

Lab Overview

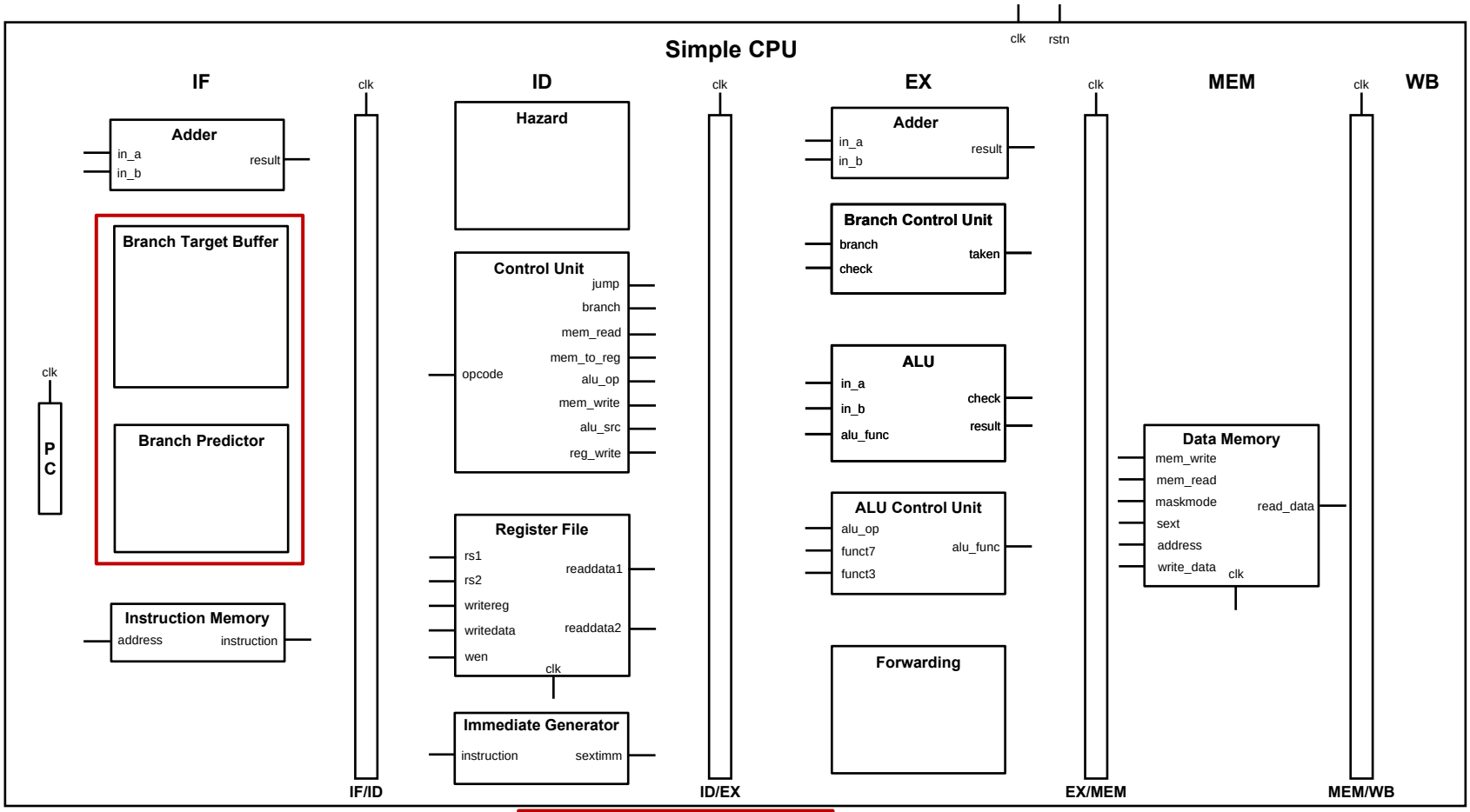
- **Goal**: Run **real** RV32I programs on your **optimized** CPU design
  - Implement **U-type instructions** (*lui* and *auipc*)



You should add ports, wires, and MUXs to complete the diagram

Lab Overview

- **Goal**: Run **real** RV32I programs on your **optimized** CPU design
  - Implement **Branch Hardware** (*branch predictor* and *branch target buffer*)



You should add ports, wires, and MUXs to complete the diagram

Lab Overview

- Workloads
  - **Unit Tests**: **Synthetic instructions** to test the CPU design
    - Task 1: Arithmetic/Logical Operations
    - Task 2: Arithmetic/Logical Operations with Immediate
    - Task 3: Load/Store Operations
    - Task 4: Branch Instructions
    - Task 5: Jump Instructions
    - Task 6: U-type Instructions
  - **Benchmarks**: **Real RV32I programs** to test the branch hardware performance
    - bst_array, fibo, matmul, quicksort, spmv, spconv
- Do not modify ***inst.mem***
  - If you want to, keep in mind that…
    - Each line of *inst.mem* consists of 32-bit instruction + newline character (33 characters)
    - *inst.mem* begins with NOP
    - *inst.mem* ends with five NOPs & Jump

Lab Overview

- Follow the instructions and improve your CPU design step by step
  - **Part 0**: Lab 3 Set Up
  - **Part 1**: Enable Full RV32I Support
  - **Part 2**: Measuring Baseline CPU Performance
  - **Part 3**: Add Branch Hardware to CPU
  - **Part 4**: Implement a Modern Branch Predictor

- Refer to **README.md** for **the details**
  - Today, we will mainly discuss **Part 3** & **Part 4**

Part 3: Add Branch Hardware to CPU

- Branch Hardware consists of …
  - **Branch Predictor**: Predict **the direction** of **conditional branches**
  - **Branch Target Buffer**: Predict **the target address** of **taken branches**

- **Accessing** Branch Hardware
  - The branch hardware is accessed in the **instruction fetch (IF) stage**,
    if the instruction is a **conditional branch** or a **(direct/indirect) jump**
    - Peek the instruction opcode in the IF stage (sort of pre-decoding)

- **Updating** Branch Hardware
  - The branch hardware is updated with **actual direction and target address**
  - It is updated in the **memory access (MEM) stage**
    (i.e., The branch is resolved in the MEM stage)
    - The branch target address is computed in the EX stage,
      but **is latched to the EX/MEM register** to shorten the critical path

Part 3: Add Branch Hardware to CPU

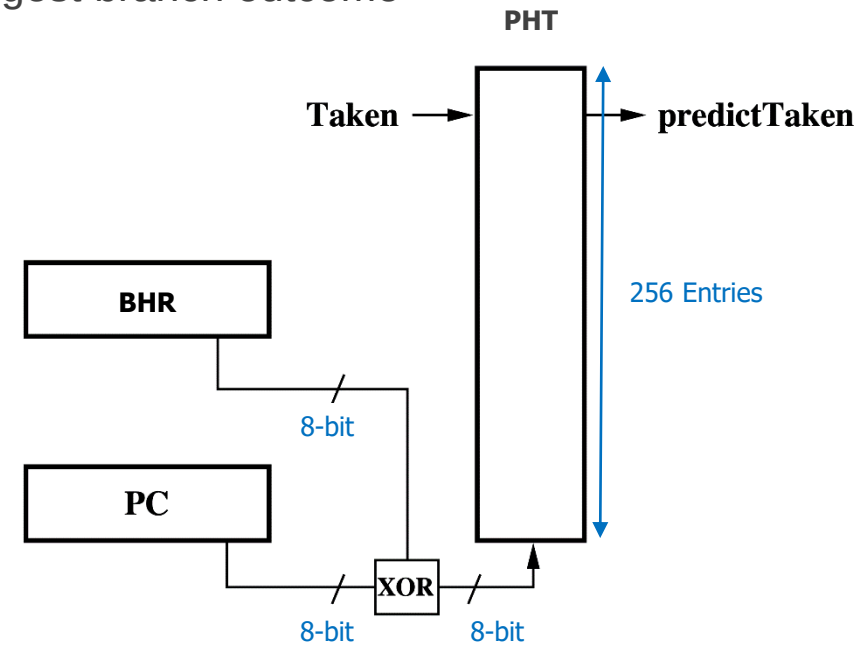
Branch Target Buffer (BTB)

- **Configurations**
  - Direct-mapped Cache
  - Consists of 256 entries
  - Each entry consists of a *valid bit*, *tag bits*, and a *32-bit branch target address*
- Initialization
  - For an **active low reset**, all the entries in the BTB must be **invalid**
- Accessing BTB
  - Index BTB **using the lower bits of the PC** (excluding PC[1:0])
  - If a BTB miss happens, use PC + 4 as the target address
- Updating BTB
  - Update BTB with the **actual branch target address** for **all types of taken branches** (taken conditional, jumps, etc)
    (i.e., Do not update BTB if the conditional branch is actually not taken)

Part 3: Add Branch Hardware to CPU

Gshare Branch Predictor

- **Configurations**
  - Branch History Register (BHR) + Pattern History Table (PHT)
  - BHR
    - 8-bit register that stores actual branch outcomes
    - The right-most bit indicates the youngest branch outcome
    - 1: taken, 0: not taken
  - PHT
    - Consists of 256 entries
    - Each entry consists of
      a **2-bit saturating counter**



Part 3: Add Branch Hardware to CPU

Gshare Branch Predictor

- Initialization
  - For an **active low reset,**
    - BHR: 0
    - PHT: 01 (weakly NT)
- Accessing Gshare Branch Predictor
  - Index Perceptron Table **using the PC XOR BHR** (PC[1:0] is ignored)
- Updating Gshare Branch Predictor
  - The branch predictor is updated **only** for the **conditional branch instructions**
  - It is updated in the **MEM stage**

Part 3: Add Branch Hardware to CPU

Next PC Selection Logic

- With branch hardware,

  now there will be **four possible next PC values**

  - PC
  - PC + 4
  - Predicted Taken PC (predicted target address from BTB)
  - Misprediction recovery PC (actual branch target address)

- Your next PC selection logic should be revised accordingly

Part 4: Implement a Modern Branch Predictor

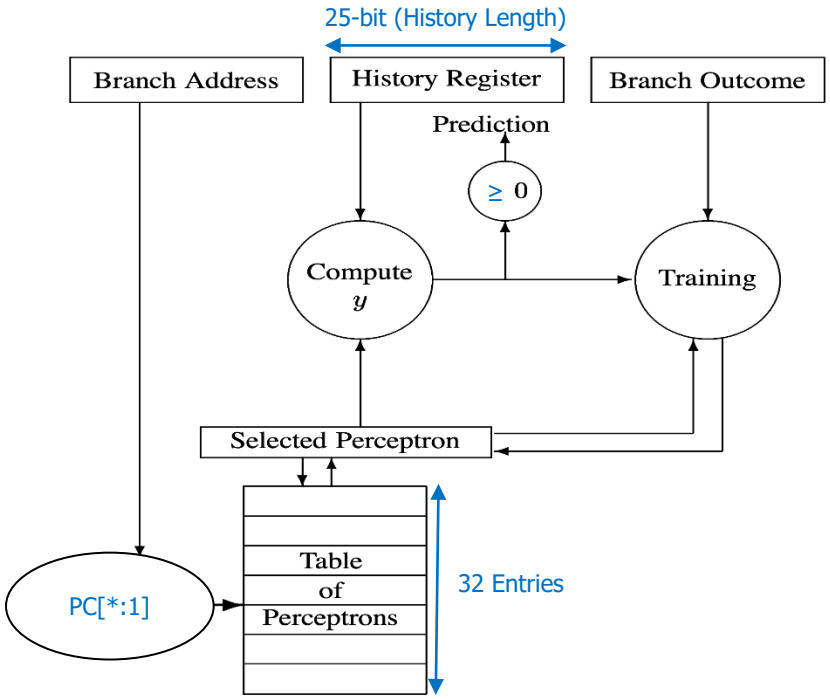
- In Part 4, you will implement
  one of the state-of-the-art branch predictors
- Implementation of **BTB and Next PC Selection Logic**
  can be reused
  - All you need to do is to replace
    the gshare predictor with the perceptron predictor
  - Every branch predictors are **functionally the same**;
    they **implement different policies** to improve the prediction accuracy

Part 4: Implement a Modern Branch Predictor

Perceptron Branch Predictor

- **Configuration**
  - Branch History Register (BHR) + Perceptron Table
  - BHR
    - 25-bit register that stores actual branch outcomes
    - The right-most bit indicates the youngest branch outcome
    - 1: taken, 0: not taken
  - Perceptron Table
    - Consists of 32 entries
    - Each entry consists of

      25 perceptron weights + 1 bias
      - Weight: 7-bit 2's complement
      - Output: 12-bit 2's complement
    - Training Threshold: 62



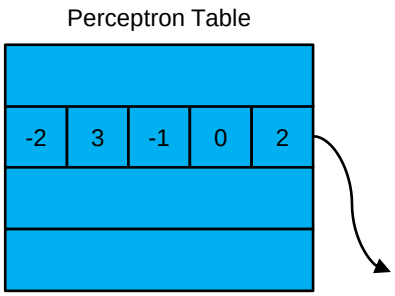
Part 4: Implement a Modern Branch Predictor

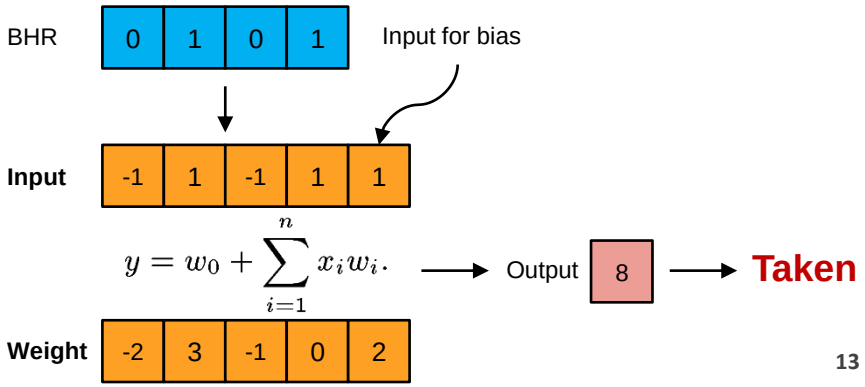
Perceptron Branch Predictor

- Initialization
  - For an **active low reset**
    - BHR: 0
    - Perceptron Table: 0
- Accessing Perceptron Branch Predictor
  - Index Perceptron Table **using the lower bits of the PC** (excluding PC[1:0])
  - Make a branch prediction by

    performing **the dot product of the weights and the inputs**
    - Inputs are the same as the BHR, except that…
      - **The 0 in BHR is considered -1 in the inputs**
      - **Input to the bias is always set to 1**
    - **Output >= 0: Taken**

*** Toy Example**
History Length: 4
Perceptron Table Entries: 4





Part 4: Implement a Modern Branch Predictor

Perceptron Branch Predictor

- Updating Perceptron Branch Predictor
  - The branch predictor is updated **only** for the **conditional branch instructions**
  - It is updated in the **MEM stage**
  - Updating Algorithm

```
if sign(y_out) ≠ t or |y_out| ≤ θ then
        for i := 0 to n do
                w_i := w_i + t x_i
        end for
end if
```

Θ: Training Threshold
t: Actual branch direction
x: Perceptron inputs
w: Perceptron weights

- **Perceptron weights are saturated at MIN/MAX value**

Tips

- Before you dive into the codes, complete the diagram
  - You should add ports, wires, and MUXs

- This lab requires **a lot of debugging**
  - Even if you pass all the unit tests, your CPU design may have bugs
  - **Do not hesitate to go over the assembly files**
    - We provide assembly files for each workload
      - Unit Tests: *inst.txt*
      - Benchmarks: **.riscv.dump*
    - The assembly looks complex at first glance,
      but if you know where to focus, **you can easily understand it**
      - Instructions are in the **.text** section
      - <_start>: 1. Initialize the registers
        2. Jump to the **<main>**
        3. Once the main function returns, jump to the **<end>**
      - **<main>: Several important functions are called**
      - <end>: Execute NOPs to wait until all the instructions retire from the five-stage pipelined CPU

Tips

- How to debug your CPU Design
  - Take advantage of **GTKWave**
    - It is a very powerful debugging tool for Verilog codes
    - `Linux> ./simple_cpu`
      - *sim.vcd* (value change dump file) will be generated
    - `Linux> gtkwave sim.vcd`
      - *gtkwave* will be launched, loading the *sim.vcd* file
  - Debugging from the command line
    - Take advantage of **Built-in Verilog System Tasks**
    - Use the "`$monitor`" or "`$display`" task to inspect the variables you want