ADS I Class Project **WS 2025/26**
Hardware Design with Chisel
Prof. Dr.-Ing. Wolfgang Kunz RPTU Kaiserslautern-Landau
M.Sc. Tobias Jauch Fachbereich Elektrotechnik und Informationstechnik
Andro Mazmishvili Entwurf informationstechnischer Systeme

Assignment 4 Pipelined RISC-V Core with Forwarding Unit and Branch Target Buffer

The fourth assignment extends the pipelined processor from task 3 with a forwarding unit, and a branch target buffer. The implemented instruction set is extended to include the B-type and J-type instructions from the RV32I ISA.

Structure of the Project

This task uses your implementation from task 3 as a starting point. Make sure that your pipelined core is working correctly before starting with this assignment.

Task 4.1 Forwarding Unit

In a pipelined processor, subsequent instructions might have data dependencies. For instance, one instruction writes to a register that the following instruction reads from. In your design from task 3, without a way to detect and resolve such a hazard, the core would produce a wrong result. A forwarding unit detects and resolves data hazards in pipelined processor cores by forwarding values from later pipeline stages to the execute stage, allowing instructions to continue without stalling.

a) Preparation Before you start with the implementation, please answer the following questions:

1. What types of hazards can be solved by forwarding?
2. Which pipeline stages are connected to the forwarding unit, and how can potential hazards be detected?
3. Are there any other potential hazards left in your specific implementation that would require stalling?

b) Implementation Add a forwarding unit to your processor design from task 3, and stick to the design presented in the lecture (slide 6-24ff, schematic on slide 6-26)

c) Test your Implementation Check whether your design runs and produces correct results by adapting the test cases you created in the previous assignments to also include dependencies that create data hazards. The test bench is located in PipelinedRISCV32I_tb.scala, test cases need to be added in HEX assembly to the "Binary File" in the "programs" folder.
A RISC-V encoder and a RISC-V interpreter might be helpful for you to achieve this task.

Task 4.2 Branch and Jump Instructions

a) Implementation Add all branch and jump instructions from the RV32I ISA subset to your pipelined 5-stage 32-bit RISC-V processor core. This includes the following instructions: beq, bne, blt, bge, bltu, bgeu, jal, and jalr.
Pay attention to enabling the core to handle control hazards that arise from branch and jump instructions correctly. For the beginning, you can use a simple static branch prediction scheme that always assumes conditional branches as not taken. Unconditional jumps (jal, jalr) should always be taken. Implement the necessary control logic to flush the pipeline when a conditional branch turns out to be taken.
Structure your core in a modular way, so that you can easily switch between the static prediction scheme and the BTB-based dynamic prediction scheme which will be implemented in the next task.

b) Test your Implementation Check whether your design runs and produces correct results by adapting the test cases you created in the previous assignments. Make sure to add test cases that specifically test different interactions of branch and jump instructions.
A RISC-V encoder and a RISC-V interpreter might be helpful for you to achieve this task.

Task 4.3 Branch Target Buffer

Your task is to implement a Branch Target Buffer (BTB), as presented in the ADS I lecture (slide 6-48), in combination with a dynamic 2-bit branch predictor (slide 6-47).
A Branch Target Buffer (BTB) is a hardware component used in modern processors to improve the efficiency of branch instructions. The BTB acts as a lookup table for target addresses. When the control flow encounters a branch instruction, the BTB checks if it contains an entry for this branch and what outcome the prediction scheme used by the BTB predicts. This allows the processor to quickly fetch the next instruction without waiting for a full evaluation of the branch. If the prediction is correct, the processor continues execution without delay. If it is wrong, the processor rolls back and resumes execution at the correct target PC. In both cases, the prediction is updated to reflect the current behavior of the program. This way, a BTB helps to reduce pipeline stalls and improves overall performance.
The BTB serves as a cache for target addresses. Therefore, we can utilize strategies known from data caches, such as associativity (cf. slide 7-23), to make the BTB more efficient.
The BTB should only be applied to conditional branch instructions (beq, bne, blt, bge, bltu, bgeu). Unconditional jumps (jal, jalr) should always be handled as taken branches without consulting the BTB.

a) Specification Enhance your RISC-V core with a 2-way set-associative Branch Target Buffer (BTB) with 8 sets. Your design should operate as an independent module, interfacing with the rest of the processor through a predefined interface.

The BTB consists of 8 sets, each containing 2 ways, making it a 2-way set-associative structure. Each entry in the BTB should include:

- A **valid bit** to indicate whether the entry contains usable data.
- A **tag** to identify the branch instruction associated with the entry.
- A **branch target address**, which provides the predicted next program counter (PC) when the branch is taken.
- A **2-bit predictor state** to indicate whether the branch is predicted to be taken or not.

Each entry in the BTB contains a 2-bit saturating counter (similar to the one shown on slide 6-47 in the ADS I lecture slides) to predict whether the branch is likely to be taken or not taken. In a 2-bit counter, a prediction must be wrong twice before it is changed. This enhancement allows the BTB not only to predict the target address of a branch but also to predict whether the branch will be taken or not. The branch prediction mechanism implements a 2-bit finite state machine (FSM) with four states:

- strongTaken
- weakTaken
- weakNotTaken
- strongNotTaken

The BTB uses bits [4:2] of the input program counter to index into one of the 8 sets (bits [1:0] are always zero, due to RV32I instruction alignment). For each set, the two ways should be checked for a tag match. If a matching tag is found and the corresponding entry is valid, the BTB should output the predicted branch target address. If no valid match is found, the BTB should indicate that no prediction is available (*valid := false.B*). In this case, a new entry will be written after the branch target has been calculated in EX stage (*update* signals). Do not forget to also initialize the 2-bit predictor FSM for a new entry.
Since each set has two ways, the design must implement a least recently used (LRU) replacement policy. This policy ensures that when a new entry is added to a full set, the entry that has not been used for the longest time will be evicted.

The BTB module uses the following I/O signals:

Inputs:

- **PC:** A 32-bit program counter representing the address of the branch instruction being fetched or executed.
- **update**: A 1-bit signal indicating whether the BTB should be updated with new information.
- **updatePC:** A 32-bit program counter associated with the branch instruction being updated.
- **updateTarget:** A 32-bit branch target address to be stored in the BTB.
- **mispredicted:** A 1-bit signal indicating whether the prediction turned out to be incorrect during execution (used to update the predictor).

Outputs:

- **valid:** A 1-bit signal indicating whether the BTB has a valid prediction for the provided program counter.
- **target:** A 32-bit signal representing the predicted branch target address when a valid prediction exists.
- **predictTaken:** A 1-bit signal indicating whether the branch is predicted to be taken or not.

When the update signal is asserted, the BTB should update the appropriate set with the updatePC and updateTarget values. In addition to updating the target address and tag, the predictor's state must also be initialized. If both ways in the set are already in use, the LRU replacement policy should be used to determine which entry to evict. An update is sent to the BTB two clock cycles as soon as the actual

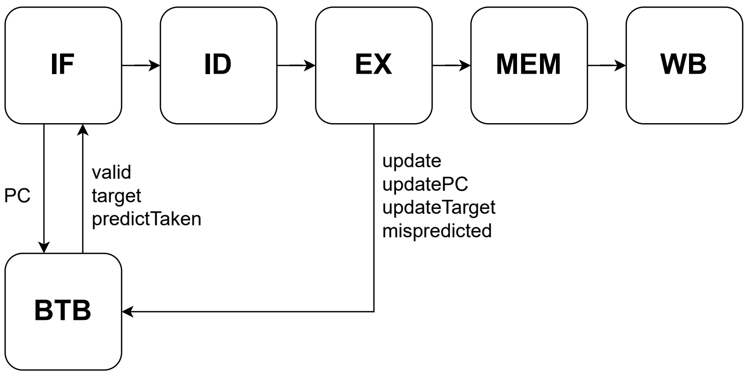

Figure 1: I/O Signals

branch target is calculated in the execute phase. Therefore, it is important to ensure that the update mechanism still selects the correct entry when a new target is written and the 2-bit predictor FSM is updated.

b) Preparation Before you start with the implementation, please answer the following questions:

1. How does a higher associativity affect the performance of a cache?
2. What's the best initial state for the 2-bit predictor FSM? What effects would different initial states have in regular program patterns (e.g., loops)?

It might also help you to think about the following details before starting with your implementation.

1. Find a way to implement a structure of registers with 8 sets and 2 ways. Standard memory classes might not be a good option here.
2. How can the number of index bits be determined?
3. How many bits does the tag of each BTB entry need in a configuration with 8 sets and 2 ways?

c) Implementation Implement a 2-way set-associative BTB according to the specification above and integrate it into your core design. Structure your core in a modular way, so that you can easily switch between the static prediction scheme from the previous task and the BTB-based dynamic prediction scheme.

d) Test your Implementation Create test cases to verify your BTB's functionality. These tests should demonstrate the following:

- Correct predictions for program counters with valid entries.
- Handling updates to the BTB properly.
- Correct eviction behavior based on the LRU replacement policy.
- Demonstrate accurate state transitions of the FSM and correct predictions based on the current state.

e) Performance Evaluation Evaluate the performance of your BTB implementation by measuring the branch prediction accuracy. Create a set of benchmark programs that include various branching patterns (e.g., loops, conditional branches) and run them on your processor core with the BTB enabled. Calculate the prediction accuracy as the ratio of correctly predicted branches to the total number of branches executed. Compare the performance of your processor core with and without the BTB to highlight the improvements achieved through branch prediction.