Programming Assignment 2 - Pipelining and Forwarding

Design Document

Group 55 - Maxwell Bruce, Samuel Woolledge, Jiapei Kuang

Pipelining

Each cycle, the five pipelined stages - Fetch, Decode, Execute, Memory, and Writeback - are ran, in reverse order. With the exception of Fetch, each stage recieves pertinent information required for running operations from a pipeline register corresponding to that instruction. Fetch recieves its instruction from the input RISC-V machine code at the appropriate PC value. With the exception of Writeback, at the completion of each stage, values necessary to the operation of the next cycle are passed along into the next cycle's pipeline register.

Pipelining Diagram (flows downwards)

Left path is the flow of information as it appears to the user, right path is the actual flow of information within the program:

Machine Code @ PC

| | *(Will be interpreted by Fetch)*

Fetch (Branch Prediction)

| \ *(Sends values to pipeline register)*

| Decode Pipeline Register

| / *(Supplies its values to Decode)*

Decode

| \ *(Sends values to pipeline register)*

| Execute Pipeline Register

| / *(Supplies its values to Execute)*

Execute (ALU Operations, BTB Updates)

| \ *(Sends values to pipeline register)*

| Memory Pipeline Register

| / *(Supplies its values to Memory)*

Memory (Load/Store Operations)

| \ *(Sends values to pipeline register)*

| Writeback Pipeline Register

| / *(Supplies its values to Writeback)*

Writeback (Writes to Registers)

Forwarding and Stalling

Forwarding is implemented via two functions: forwarded_register_read_single and forwarded_register_read. Functions are checked for dependencies with the later cycles - if such a dependency is found, the program will forward the results from the pipeline registers to the stage that requires these values with these functions, or in the event of a necessary stall, will stall until the value is ready for forwarding, in which case it will occur as normal.

Because the program has full forwarding, stalls will only occur in the event of a load followed by an instruction where a dependcy exists between the load's destination register and the operation register(s) instruction, or a branch misprediction. To handle stall, a variable "execute_flush_cycles" is incremented by the necessary amount, which then stalls the instruction flow until dependencies are resolved. To handle mispredictions, Fetch and Decode are flushed and the PC is set according to the correct target address. The stall and flush and implemented via an early return in the execute stage.

Branch Prediction - Overview

The functions and values governing branch prediction are within the branch_predictor.* files. These functions and values are implemented within the program as needed. They must be included with the riscv_virtualizer for the program to be able to function properly.

Branch Prediction - Predicting the Address

Branches predictions are made in the Fetch stage. Values within the BTB are stored in an array of length 32, which is an array of a struct that contains the address of the instruction it corresponds to, the target address of the instruction, and the BHT (which is vestigial for the BTFNT predictor). The lower 5-bits of the address of an instruction are used as a key to access an index of the BTB, and an address comparison is made to ascertain whether or not the corresponding BTB entry matches with the instruction's full address. If so, an address prediction is made. If not, PC+4 is returned. The PC returned by the prediction function is always used as the PC going forward, until a misprediction is caught, in which case the pipeline is flushed.

There are two functions for address prediction within this program: one that uses a 2-bit BHT for prediction, and one that uses the BTFNT principle, named, respectively, predict_address and predict_address_BTFNT. The functions receive both take one parameter, which is the PC of the current instruction. predict_address* checks the BTB, and if a corresponding entry is found, it looks at the BHT. For predict_address, if the BHT is greater than 1, the target address from the BTB is returned. If not, PC+4 is returned. predict_address_BTFNT, instead of checking the BHT, compares PC+4 with the target, and returns the lesser value as the predicted target address.

Branch Prediction - Predicting the Address Diagram

```
Fetch

| (Sends address to predict_address*)

predict_address*

|        -predict_address

|        --- if entry in BTB found for given address && BHT > 1, return target address

|        --- else, return PC+4

|        -predict_address_BTFNT

|        --- if entry in BTB found for given address && target address < PC+4, return

|            target address

|        --- else, return PC+4

| (Sends predicted address back to Fetch to be used as the PC)

Fetch
```

Branch Prediction - Updating the BTB

Mispredicts will be caught in the execute stage. If a mispredict is found, the Fetch and Decode stages will be flushed, and the PC will be set to the proper value, so that the next Fetch cycle will fetch the proper instruction. Additionally, whether or not the branch was mispredicted, if a branch instruction occurred, then the BTB will be updated. The BTB will be passed the address of the instruction, the target of the branch, and a value corresponding to whether or not the branch was taken (hereafter referred to as branch_taken, and refferred to internally in the program as t_bht) to the function update_entry.

update_entry is unchanged whether the BTFNT or the 2-bit BHT implementation is used. branch_taken is passed along as well. Because the calculation for whether or not a branch is taken ignores the BHT in predict_address_BTFNT, the BHT is irrelevant in that regard and can be updated as normal in order to simplify implementation.

If the index corresponding the the address of the branch matches with the branch, the BHT is updated. The BHT variable is [0-3], and is decremented if the branch was not taken, and incremented if it was.If the entry at the index does NOT correspond to the address of the branch, it is overwritten with the address and target of the new branch. The BHT is also set based on the BTFNT principle - in other words, to (branch address+4 < target address) * 3.

Branch Prediction - Updating the BTB Diagram

Execute *(Mispredictions are caught, and PC is updated/Pipeline flushed accordingly)*

| *(If a branch occurred, the BTB is updated)*

update_entry

```
|        - BTB entry corresponds to this branch instruction
|        --- update the BHT
|        - BTB entry does NOT correspond to this branch instruction
|        --- update the branch address to the new branch address
|        --- update the target address to the new target address
|        --- set the BHT based on BTFNT principle [(branch address+4 < target address)
|             * 3]
```

Execute