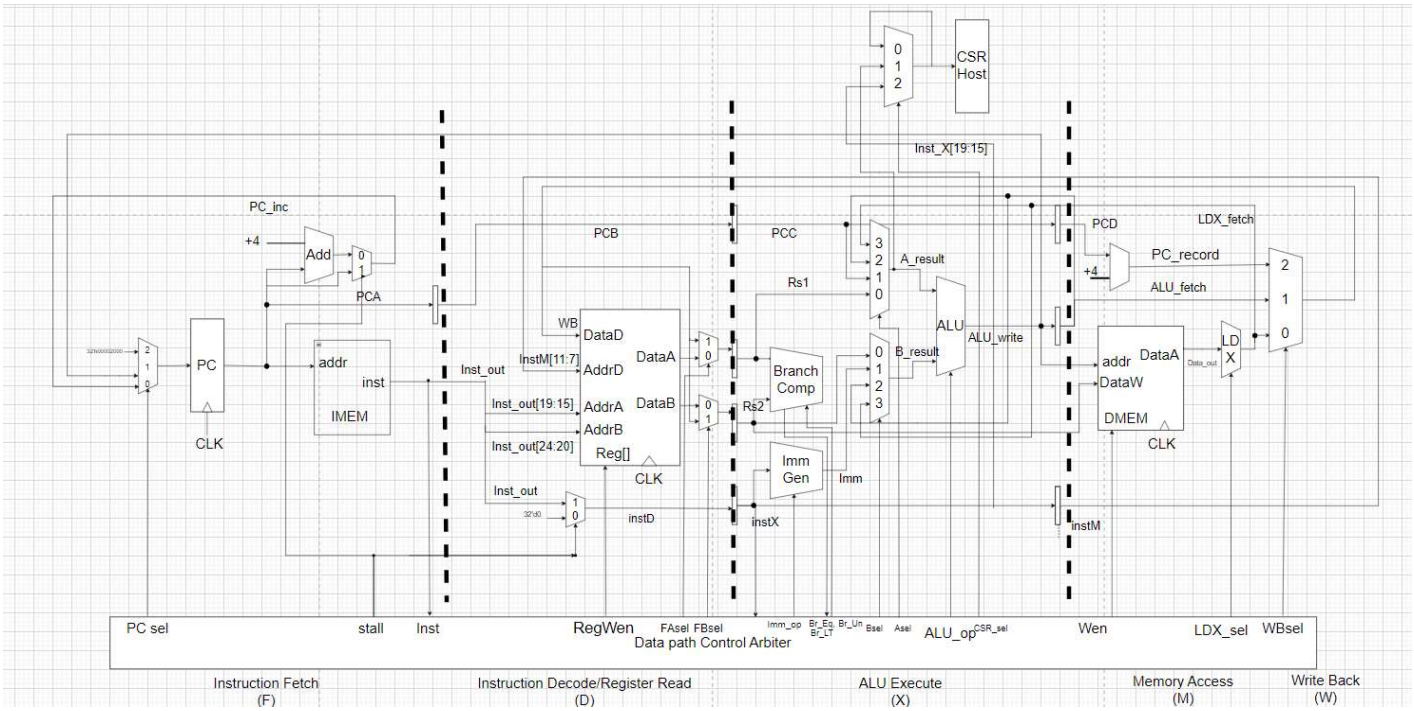
EECS151 final project report

Marco Chen, Shengxi Liang



1. Design strategy

1.1 Datapath

We are using a pretty ordinary 4 stage pipeline.

The first stage is Instruction Fetch (F), in which it reads instructions from memory. The address of this instruction is in the Program Counter (PC). The address in the program counter is transferred to the memory address register. IMEM stores instructions that the program operates on.

The second stage is instruction decoding (D), in which it takes uncompressed instruction data and issues appropriate control signals to the other blocks to execute the instruction. Register x0 is statically bound to 0 and can only be read, it does not contain any sequential logic. Register file data is available the same cycle a read is requested. The register file allows for significant area savings compared to an implementation using regular SRAM, but consumes more power.

The third stage is Execute (X), in which it performs the operation defined in the instruction. The Arithmetic and Logic Unit (ALU) is a purely combinational block that implements operations required for the integer computational instructions and the comparison operations required for the control transfer instructions in the RISC-V specification. The Control and Status Register Block (CSR) stores state independent of the register file and memory. The register is monitored by the test harness, and simulation ends when a value is written to this register. A value of 1 indicates success, and a value greater than 1 gives clues as to the location of the failure.

The fourth stage is memory access (M) and write-back (W). If necessary, it reads memory locations and writes data to memory. We are combining the write-back and memory stages because we can think of the memory system as an isolating register, and when the DMEM receives a signal from the execution stage, it will begin fetching and will generate results after passing LDX, which aids in load masking, and then go to write-back.

To solve errors regarding operations, we implement various forwarding mechanics. The first case where forwarding needs to happen is when an operation would change the register value after the memory stage but happens to have an execute state instruction that requires such a new value for operations. We solve this by adding more options to the Asel and Bsel, new 2 operations indicate forwarding on the ALU result, 3 three indicate forwarding on the load result. Similar to ALU modules, the branch comparator also needs a forwarding logic to compare new register values instead of the old one, thus having similar but separate forwarding signals to help switch between result coming from the register, the AIU result of the previous stage, and the result of the previous stage loading. There is also another type of forwarding we took a caution on, in case a decode stage operation would require a new wb information, if with forwarding, the new wb result will not be pumped out of the register on time. Thus, we add a forwarding logic inside the register file, which helps to check the case when rd of the previous operation is in fact the needed rs1 or rs2 for the new operation.

In the current stage, due to the limit of time and our failure to get a fully functional core, we haven’t considered any branch and jump predictions, and instead we use stall signals to help clean up the PCs and instructions when needed.

To minimize the clock period and increase the frequency of our design, we took advantage of many clock based components, viewing them as a pipeline register themselves. Inside each stage we wish all operations can be performed instantly to help reduce time, and using that idea we designed a complicated control arbiter to send signals to different devices based on separate instruction signals of different time.

1.2 Control arbiter

The control arbiter contains the state machine that directs and orchestrates execution of the instructions and program, including data flows.

In the fetch state, the control arbiter issues a read command on the control bus, and the result is displayed on the data bus before being copied into the register file. The program counter is increased by one to prepare for the next instruction.

In the execute state and memory state, we decode instX to get instructions. Based on instructions, we assign our control signal to implement the CPU properly.

To manage the pipeline errors, we have implemented several types of forwarding.

The first step forward occurs when we attempt to match the calculated result of the previous execution, which we accomplish by using AB mux option number 2.

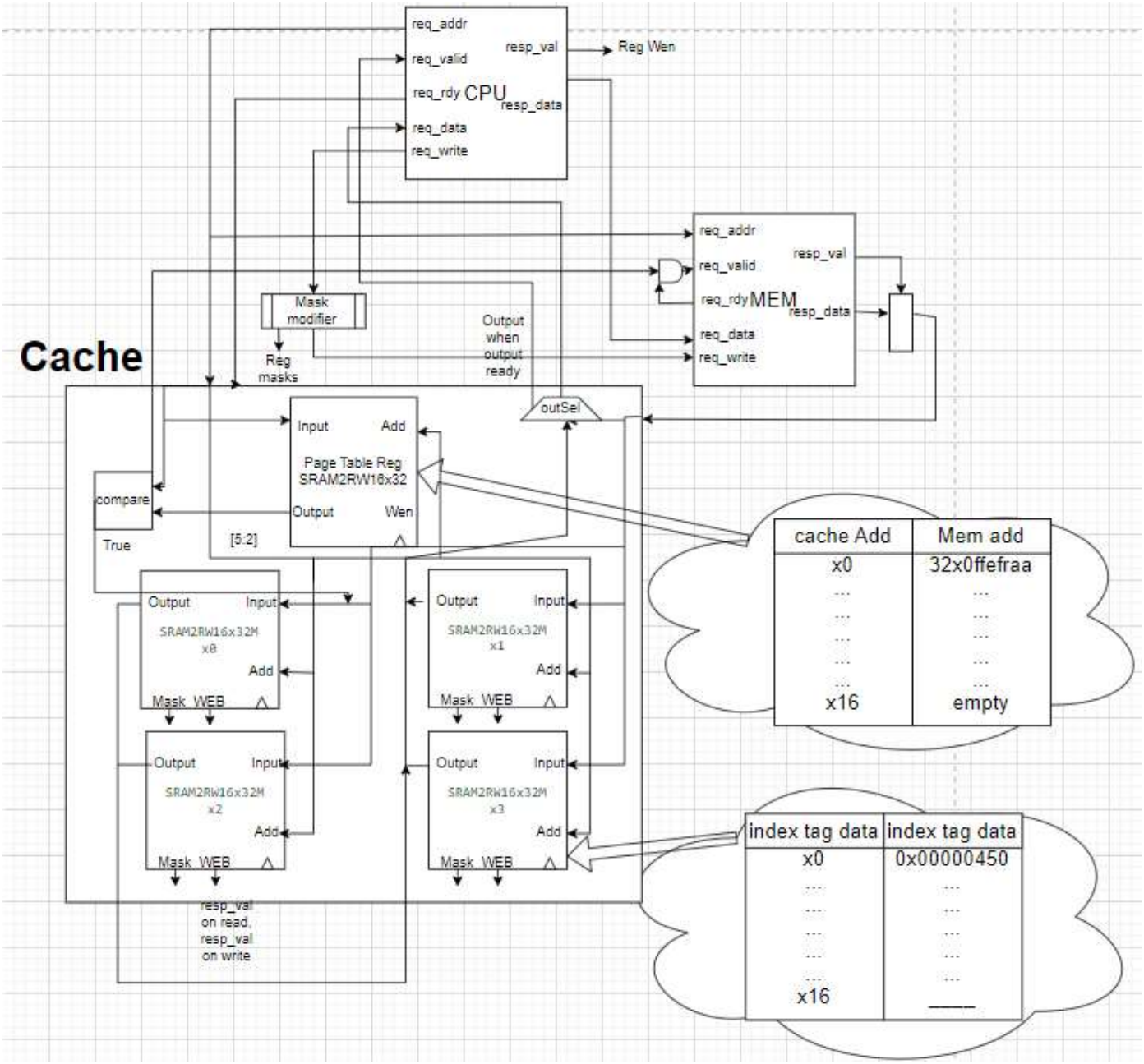
The second forwarding happens after the load, and we straight up connect the result from LDX to AB mux and get option number 3.
Forwarding type 3 is a bit special because when the two interacting instructions are only two instructions apart, after the result is just written to the register, the later instruction could potentially pull the old result out instead of the new one.

To fix this special issue, we made some small changes in our register and gave the register output an option that depends on the signal RegWen. Once detected, the control arbiter would send such a signal and tell the register to output a new value instead of giving out the old one.

Branching and jumping change our program counter. As we do branch prediction and jump prediction, when it happens, we set stall to be 1. This has helped us to get a stable running system to some extent.

Our CPU can be configured to use static branch prediction by setting the stall to 1. When successful, the prediction removes a stall cycle from a taken branch. However, there is a mis-predicted penalty if a branch is wrongly predicted to be taken. This penalty is at least one cycle, or at least two cycles if the instruction following the branch is uncompressed and not aligned.

1.3 Cache system



Though we never actually reach the cache section due to the hindrance in the data path, we have come up with a good idea of what our register would be like and even write a prototype cache for it.

The cache we're currently using is 16 * 4 = 64 words long. We are using a 16 associative LRU cache. The writing rules are written through so we plan to write to cache and to the memory at the same time to solve potential synchronization issues.

We also have a register act as a page table to help check if each set of cache represents the real address of the memory system. The order of execution is as follows:
1) The CPU calls for a write or read when the cache is idle.
2) The cache first searches on the table using the address given (the first 2 digits of the address are the tag value since a memory has 4 CPUs and the next 4 digits correspond to 16 different cache sets; the memory I am using here is the first of the tag set), and if the cache already exists in the cache system, it will move to the read stage next; if it does not, it will move to the write stage (for a write hit, it also moves to the write stage).
3) For the read stage, the cache will simply fetch the result based on the cache sets and tag number (mod 4 result) and give that value to the output.
4) For the write stage, if it is a read miss or write miss, both would give a signal to the memory system with a read mask to read and then store the memory set by separating them into 4 different memory sectors to be stored in different cache units based on the same cache set address. If it is a write, we will modify the mask based on the mask given by the cpu to only modify the given cpu address and change the memory result, and then for all write action will change the particular cache data with the given mask. After writing cache data to the table and memory, the cache system enters an idle state and waits for new CPU calls.

During the whole process, the cache system will not call for any new CPU actions, thus needing at least 3 cycles to finish (check table, read from memory, write to cache), much less than the suggested 4 cycles.

This cache is far from perfect, first we have tested it before and it may have many unexpected errors. Second, our current cache is still relatively too small with only 4 sets of memory for a memory set (1/16 of all mem addresses), it would be much better to have a bigger cache and better mapping technique).