Progress Report

For the first checkpoint, the contributions were as follows:

Arjun: Added pipeline control. Implemented the control word using a struct and created the ROM that generates the control signals. Completed the forwarding logic, hazard detection, and arbiter design.

Gally: Worked alongside John to implement the datapath and created the modules required for each of the stages and the pipeline registers. Played a major role in debugging the code by looking at signals.

John: Worked alongside Gally to implement the datapath and created the modules required for each of the stages and the pipeline registers. Worked on fixing errors and making was responsible for code changes during the debug process.

Roadmap

Arjun: Arjun plans on finishing the arbiter relatively quickly and then work on implementing the cache.

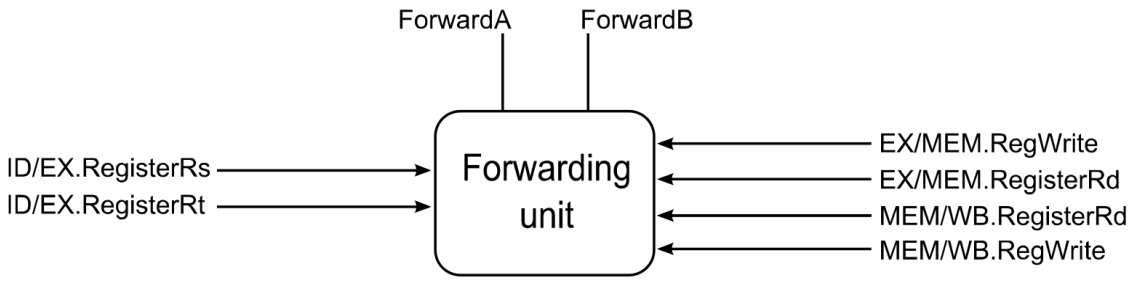
Gally: Gally will be responsible for doing the branch prediction which we foresee to be difficult.

John: The plan is for John to focus on the forwarding unit and then move on to the hazard detection which Arjun should be able to help with.

After the checkpoint is completed we plan on working together to get the competition code working and make optimizations.

Forwarding Logic

Our forwarding unit takes in input from the ID/EX stage, EX/MEM stage, and the MEM/WB stage to produce two outputs, ForwardA and ForwardB, that are the mux selects that determine if forwarding is required. This also assumes that we have implemented a read after write register file so that the ID and WB stages can be overlapped



The logic for the forwarding unit must do the following:
(1) Forward from the EX stage to the input of the ALU for dependent instruction*
(2) Forward from the MEM stage to the input of the ALU for dependent instructions*
(3) Forwarding from the EX stage takes precedence over forwarding from the MEM stage
(4) If there are no dependencies on any of the two previous instructions, use the register values read in the ID stage

*Dependent instructions are ones where the source register of a later instruction is the destination register of an earlier (potentially still executing) instruction

NOTE: Forwarding alone cannot handle a load instruction followed by a dependent ALUop instruction as the output of the MEM stage for the load instruction is not ready for the EX stage of the ALUop instruction. We will have to implement a one-cycle stall along with forwarding.

Hazard Detection

Data Hazards and Stalls
WAW and WAR data hazards will not occur in our implementation of a RISC-V pipelined processor because writes always occur in the same pipeline stage and reads of an earlier instruction will always happen before writes of a later instruction.

A RAW hazard could still occur when we have a load instruction followed by a dependent ALUop instruction, in which case we need to implement a one cycle stall. The pseudocode for determining this stall would look like:

```
if (ID/EX.MemRead and

   ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or

    (ID/EX.RegisterRd = IF/ID.RegisterRs2)))

          stall the pipeline
```

The pipeline also needs to stall while waiting for memory accesses, either in the IF or MEM stages. The pipeline can continue once the corresponding mem_response signals are asserted.

Control Hazards
For our basic implementation (and to keep the forwarding logic as above), we plan to have the branch resolution occur in MEM stage. Eventually we will have some structure that uses the PC to determine the next instruction. We maintain a record of this choice and when the branch is resolved, we determine if this was the correct decision. With the branch resolution occurring in the MEM stage, potentially three incorrect instructions were fetched. We have to clear the instruction fields of the IF/ID, ID/EX, and EX/MEM stages, essentially turning these instructions into nops.

Arbiter

Signals out of I-Cache to arbiter:
instruction_pmem_address;
instruction_pmem_wdata;
instruction_pmem_read;
instruction_pmem_write;

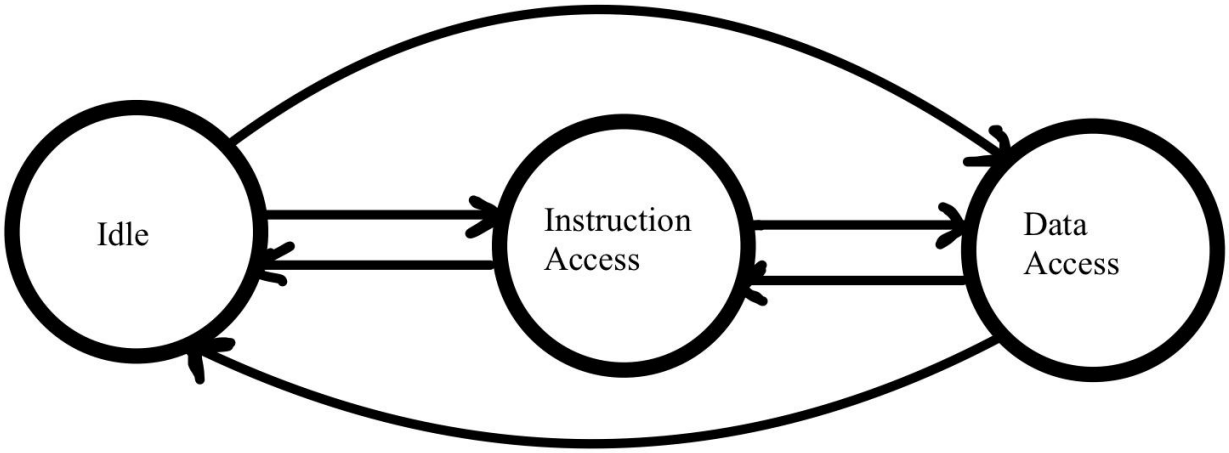
Signals into I-Cache from arbiter:
instruction_pmem_resp;
instruction_pmem_rdata;

Signals out of D-Cache to arbiter:
data_pmem_address;
data_pmem_wdata;
data_pmem_read;
data_pmem_write;

Signals into D-Cache from arbiter:
data_pmem_resp;
data_pmem_rdata;

Signal out of arbiter to memory:
pmem_address;
pmem_wdata;
pmem_read;
pmem_write;

Signal into arbiter from memory:
pmem_resp;
pmem_rdata;



State Definitions:
Idle - No current memory request
Instruction Access - Ongoing instruction memory request
Data Access - Ongoing data memory request

Transitions:
From Idle - If there is an instruction memory request go to instruction access. If there is a data memory request go to data access. (priority to instructions over data request in the same cycle)
From Instruction Access - Upon memory access completion, if there is a data memory request go to data access. Else, return to idle.
From Data Access - Upon memory access completion, if there is an instruction memory request go to instruction access. Else, return to idle.