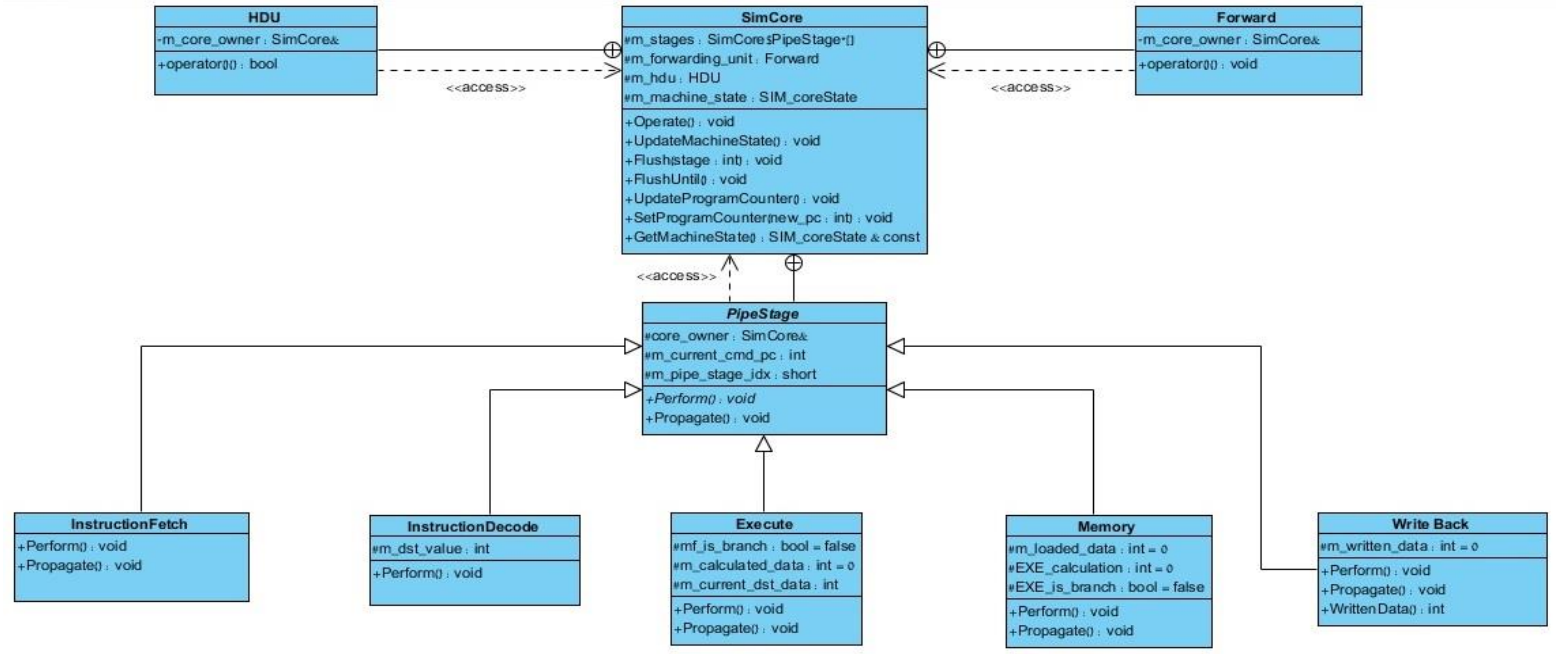
Comupter Structure 1

Partners:

1. Yaroslav Funt – 317582518
2. Dror Kabely – 311177109

Class Hierarchy



Implementation classes:

1. **<u>PipeStage</u>** – An abstract class representing a single pipe stage. Contains a reference to a SimCore instance (the owner of the stage), the pipe index inside the SimCore owner's stage container and the PC of the current command inside the stage.
   Abstraction is caused by the method ***"Perform"*** that is implemented only by the derived classes.
   The second method inside this base class is ***"Propagate"*** which pulls the previous stage's values at the beginning of every clock tick.
   Derived classes are:
   1. **<u>InstructionFetch</u>** – overrides "***Perform***" and essentially does nothing in it.
      Overrides ***"Propagate"*** which reads another instruction from the instruction memory.

2. **<u>InstructionDecode</u>** – overrides ***"Perform"*** which reads the register file for the sources' values (including the destination register value).
Doesn't override ***"Propagate"*** method. Also invokes PipeStage::Perform().
3. **<u>Execute</u>** – overrides ***"Perform"*** which forwards the values from WB and MEM in need and then operates on provided the data according to the current command opcode in the EXE stage.
Overrides ***"Propagate"*** method which pulls the destination value from ID stage. Also invokes PipeStage::Perform().
4. <u>Memory</u> - overrides ***"Perform"*** which stalls the pipe in case of memory stall, writes to memory in case of STORE instruction, and flushes the pipe in case of a taken branch.
Overrides ***"Propagate"*** method which pulls the calculations from EXE stage, including branch flag for branch resolution. Also invokes PipeStage::Perform().
5. <u>WriteBack</u> - overrides ***"Perform"*** which writes to the register file in case of a LOAD, ADD or SUB instruction.
Overrides ***"Propagate"*** method which pulls the appropriate value to be written to the register file.
Doesn't invoke PipeStage::Perform().

2. **<u>Forward</u>** – The forwarding unit.
   Implements the forwarding algorithm via operator() overloading.
3. **<u>HDU</u>** – The hazard detection unit.
   Implements the hazard detection algorithm via operator() overloading.
4. **<u>SimCore</u>** – The main class.
   Contains all the declarations and definitions of the rest of the classes.
   Member variables:

1. m_stages – an std::vector of type PipeStage* (exploiting the virtual methods ***"Perform"*** and ***"Propagate"***.
2. m_hazard_detection_unit – an HDU object representing the hazard detection unit of this core.
3. m_forwarding_unit – a Forward object representing the forwarding unit of this core.
4. m_machine_state – a struct of type Sim_coreState provided by the API.

Pipe Structure:

As described above, the 5 pipe stages are implemented by dynamically allocating 5 objects of "PipeStage" derived classes: InstructionFetch, InstructionDecode, Execute, Memory and WriteBack that interact with each other. Every pipe stage derived class implements its own unique behavior via overloading ***"Perform"*** and ***"Propagate"*** methods, and doing changes in the Sim_coreState member struct according to its behavior.

Invoke Order:

At every clock tick we do 2 basic operations:

1. Update the machine state (a.k.a propagate the values computed in the previous clock tick)
2. Operate (compute values) at every stage of the pipe.

So essentially we, every call to SIM_clkTick we invoke the two member functions of the static object "machine_core":

```
void SIM_CoreClkTick(void)
{
	machine_core.UpdateMachineState();
	machine_core.Operate();
}
```

Forwarding

Forwarding is done in the beginning of Execute::Perform.

Forwarding is done as follows:

1. If MEM destination register index matches EXE src1 register index and MEM opcode is SUB or ADD then assign EXE src1 value with the value of in MEM stage that was propagated from EXE stage the previous clock tick.

   Else if WB destination register index matches EXE src1 register index and WB opcode is SUB or ADD or LOAD then assign EXE src1 value with the value held by WB (the one that is supposed to be written back to the register file)

2. If EXE src2 is not immediate than do (1) for EXE src2

3. If EXE opcode is BR or BREQ or BRNEQ or STORE and MEM opcode is ADD or SUB and EXE destination register index matches MEM destination register index then assign the value that was propagated in the previous clock cycle from EXE to MEM, to EXE current destination value.

   Else if EXE opcode is BR or BREQ or BRNEQ or STORE and WB opcode is ADD or SUB or LOAD and EXE destination register index matches WB destination register index then assign the value held by WB to EXE destination value

Memory stall:

Memory read is done in SimCore::Memory::Perform in case MEM opcode is LOAD. In case the call to Sim_MemDataRead fails, Memory stage sets a flag that belongs to SimCore (SimCore::Memory has access to SimCore internals via friend declaration), called "m_update_flag".

The flag controls the flow of the method that updates the machine state, 'SimCore::UpdateMachineState' – if 'm_update_flag' is false, don't propagate values and return. The next call to SimCore::Operate() will only execute WB stage, try to read again from the memory via Memory::Perform until it succeeds and when it does, set the flag to true. Of course, in case 'm_update_flag' is false, the program counter is not updated.

Two-phase write-read register file:

In every call to SimCore::Operate we invoke the stages "Perform" methods in reverse order, first the WriteBack, then Memory, then Execute, then InstructionDecode and InstructionFetch at last. Because WriteBack operates before InstructionDecode tries to read the register file for value, we succeed in out goal.

Hazard Detection:

Hazard detection is done at the end of each call to SimCore::UpdateMachineState, as follows:

1. Get a reference to EXE and ID command struct
2. If EXE command opcode is LOAD

    If ID command opcode is not NOP nor BR and EXE destination register equals ID src1 register or ID src2 register then return true

    Else If ID command opcode is BR or BREQ or BRNEQ and ID destination regsiter equals EXE destination register then reutrn true

    Else return false

    Else return false (EXE command is not LOAD)