Using the code

1. I need to start by analyzing all the files

```
ghdl -a archer_pkg.vhd
ghdl -a add4.vhd alu.vhd branch_cmp.vhd control.vhd immgen.vhd lmb.vhd mux2to1.vhd
pc.vhd regfile.vhd rom.vhd sram.vhd CSRfile.vhd branch_addr_alu.vhd IF_ID.vhd
ID_EX.vhd EX_MEM.vhd MEM_WB.vhd dataForwarding.vhd mux3to1.vhd mux2to1_5b.vhd
multALU.vhd multALUbus.vhd archer_rv32i_pipelined.vhd archer_rv32i_pipelined_tb.vhd
```

Note that I could analyze them all in one line but I prefer this for neatness
Also note that if new components are added they need to be analyzed as well

2. I need to elaborate by entering the top-level-entity which is the test bench

```
ghdl -e archer_rv32i_single_cycle_tb
```

3. I can finally run the code

```
ghdl -r archer_rv32i_single_cycle_tb --wave=waveName.ghw
```

4. To observe the wave

```
gtkwave wave.ghw
```

Observing the fibonacci series

Note that there is a file containing the ROM content for the fibonacci series

To observe the series I need to look into the data memory (dmem), into the ram. There I can see everything stored in memory after the code ran.

For the purposes of our application, we will have the first 11 terms of the fibonacci series stored

- In Decimal: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
- In Hexadecimal: 0, 1, 1, 2, 3, 5, 8, D, 15, 22, 37

Since this is a 32bit processor, every number will take 32bits so 4 bytes, meaning every 4bytes (4 locations in memory) we can find the number.

Note that the numbers are small enough that they will only be 1 byte meaning the other 3 will be empty (0s)
So ram[0]->ram[3] will store the first number which is 0, ram[4]->ram[7] will store the second number which is 1 and so on
So at ram[0] I observe 0, at ram[4] I observe 1, at ram[8] I observe 1, at ram[12] I observe 2 and so on

Reading instructions in ROM

- To understand how instructions are read, each instruction is 32bits, meaning it is stored in 4 consecutive places in memory (each address holds 8bits only). So for example the first add t0,x0,0 instruction has the first 4 X"" in ROM corresponding to it so the instruction is (X"00 00 02 B3")
  (Note rd=5 which corresponds to the first temporary register t0)
- I noticed that using the initial code, the last instruction which is ecall is not fetched, the code ends before it at the addi a0,x0,10. Essentially ecall is not supported (not yet at least) so it wouldn't make much of a difference.
  But in case we change the first instruction and initialize the counter t0 to a bigger value (5 for example) instead of 0, we can observe that ecall gets called at some point , and after it the instruction that is located right after it in memory gets fetched. This is because ecall is not supported hence calling it will do nothing and pc will continue incrementing normally.
  ↑ To do this we used replaced the first instruction by X"00500293"

Stack Pointer (out of scope)

As a side note or observation: The register x2 which is the sp (stack pointer) in the register file is being initialize to 1024(the last memory address in RAM).

Accordingly we can observe that values are being sent to memory throughout the code other than just the ones that are the fibonacci numbers, and those are being stored in the higher registers of the RAM.

I believe these have to do with the stack pointer and jump instructions.

Adding CSRs to the Datapath

- I am going to create a new component, a register file, to store the control and status registers CSRs.
  - To follow the convention used in the book for the addresses I will create a register file of fictive size 4096.

    - Effectively the register file will only have 4 registers of 32bits, but I will use the conventional memory addresses for the CSRs and map them to the 4 registers I have
- I will assume we do not need to keep track of when the code stops running, ie I will assume the clock cycles are just enough. (I could potentially tell when the code ends by keeping track of the ecall, but so far ecall is not supported)
- The counting is being done inside the CSRfile to allow storing the values during the same clock cycle.
- CSR_cycles: I need to keep track of the clk and rst signal
- CSR_instret: I will keep track of the instruction coming back from the WB stage

Code Modifications

→ In the processor file: Instantiated 2 new components

- Imported the component CSRfile and maped the signals accordingly.
- Imported a new mux (CSRMux) between *alu_src2_mux* and *alu_inst*, for the purpose of choosing the output of the CSRfile to go into inputB of the ALU in case it is a CSR instruction. It is driven by the control *CSR*. Its result is connected to inputB of the ALU

→ In the control signal: Created 2 new signals

- CSR: To indicate whether this is a CSR instruction or not
- CSRWen: To allow the instruction to write back to the CSRfile after checking that rs1 /= x0

→ ALU: Created a new ALUOp

- For the purpose of having a case to invert inputA and then "AND" it with inputB.
- This is needed for the correct implementation of CSRRC

CSR_cycles

- The addresses will be X"C00" for the lower 32bits, X"C80" for the upper 32 bits
- To observe the values go into archer->csrfile_inst->storage and then select the desired address

CSR_instret

- I am getting the instruction from the WB stage. This is to ensure that if instructions end up being flushed they don't get counted
- I counted by just saying if the instr_word I have is not a NOP or a zero instruction then increment the counter

→ I first implemented each CSR in a process, but noticed that VHDL does not like when 2 processes "potentially" write to the same location in storage. So I ended up implementing them n the same process

CSRRS/CSRRC

For the purpose of supporting those two instructions below are the modifications implemented:

- We added a new mux between *alu_src2_mux* and *alu_inst*. Its first input is the output of the previous mux (d_alu_src2), its second input is the result coming from the CSRfile. It is driven by the control *CSR*. Its result is connected to inputB of the ALU
- We added a new function in the ALU (and hence a new ALU_OP in archer_pkg) for the purpose of inverting rs1 (using xor with ones) and doing AND with the result coming from CSRfile. (This is only needed for the CSRRC because in CSRRS we don't need to invert the mask, we only need to do AND which is already implemented).

Pipelining

New Components, and New Command

→ New Components

- *branch_addr_alu* to calculate the address of the branch in ID stage
- *IF_ID* register bus
- *ID_EX* register bus
- *EX_MEM* register bus
- *MEM_WB* register bus
- *multALU* as the multiply unit
- *multALUbus* as the register bus inside the multALU unit pipelined

New Control signals

- Input:
  - MStall coming from multALU unit
  - MNop coming from multALU unit
- Output:
  - Stall: This is gonna be used for stalling in general. For our current purposes, MStall will tell me there is an instruction in the multiply unit. I check if the current instruction in ID is a multiply instruction we let it pass (because multiply is pipelined) so "Stall<=0", else I stall it so "Stall<=1"

    - Nop: This is gonna be used for sending NOP to EX/MEM in general. For our current purposes, Nop<=MNop
    - MExt: This is '1' when the instruction i have is a M extension instruction

Modifications in CSRfile

- The input instruction is used for both reading and writing to the CSRfile. But now each will be done in a different stage
- Reading is done in the EX stage
- Writing is done in the WB stage
- To solve this issue I added an additional instruction_WB input port. This will take the instruction from the WB register, and the initial instruction will therefore only be used for reading
- I also modified the instruction counter to read from the instruction_WB not EX
→ Side Note: The condition of instr_word_WB(0)=0 or 1 is used to make sure I don't count if the signal is undefined

M extension instruction unit

- I was asked to do a multiply unit that simulates 5 clock cycles to give a result
- I did a new multALUbus as well to simulate the pipelining inside the multiply unit alu
- When it comes to pipelining:
    - It was made in a way to support multiply pipelining
    - The only signals that I care to preserve from one multiply to the next are the result and the instruction. So we inputed the instruction in the mult unit, made it go throught the five stages and at the entrance of the EX_MEM bus there are 2 muxes that ensure we take the instruction and the result from the multiply unit
- When it comes to stalling:
    - MStall will go to 10 when MExt_EX1 becomes 1, if the instruction is a divide it will go to 11 (because divide instructions do not support pipelining)
    - MStall will go back to zero as soon as the instruction reaches the last execution stage (EX5) in the mult unit at the condition that the instruction in the EX1 and the instruction in EX5 are the same one to account for pipelined multiplication
- When it comes to NOP:
    - I only sent NOP to EX/MEM because the MEM/WB will inherit the NOP from it
    - As soon as the first multiply that entered the pipeline arrive to EX5, NOP turns back to 0

Implementing Data Forwarding

New Components and Elements modifications

- *dataForwarding* used as the data forwarding unit
- *bNop* signal for the ID/EX stage
- *bStall* and *dfStall* inputs for the control signal

Data Forwarding unit functioning

- For regular forwarding
    - It takes in rs1 and rs2 from the EX stage to compare them with rd from MEM and WB
    - rd coming from EX is used to make sure it is not zero, in case it does it would be useless to check for equality
    - The instruction coming from EX is used to check that rs1 and rs2 are used and aren't just buffer numbers (some instructions don't have rs1 and rs2)
    - RegWrite coming from MEM and WB are used to check that the instructions there will actually write back to there respective rd, otherwise the values coming from them are insignificant and shouldn't be forwarded
- For M extension forwarding
    - The concept goes mainly the same as for the normal forwarding, the difference is that when the mult(or div) is in the last EX stage inside the multiply unit, it will check for data forwarding there
- For branch forwarding
    - rs1 and rs2 are taken from ID to check for dependencies while calculating in the branch comparator
    - In case rs1 and rs2 are dependent on rd of EX I will assert bStall, which will cause a stall in IF and ID and will send a NOP through the ID/EX bus
    - In case rs1 and rs2 depend on rd of MEM or WB, I will implement normal data forwarding
      → I had to create the *regularStall* signal for the ID/EX bus. In case a branch in ID depends on an instruction in EX, and this instruction in turn depends on a load in MEM, we need to give the priority for the stall coming from the EX instuction, and would want to avoid sending a nop in the EX_MEM bus until the next cc

Implementing Hazard Detection

Data forwarding with load

- I implemented this feature while working on the normal data forwarding as it is only a matter of checing the below

- MEMtoReg_MEM is used to check that the instruction in the memory stage isn't a load. In case it is the stall signal will be asserted
- Stall is sent to the control unit which will in turn assert the Stall signal coming out of it. It will stall the instructions in IF, ID and EX and will send a NOP to the MEM stage
- On the next clock cycle the instruction in the EX will be able to take the result of the load from the WB stage normally

Handling Control Hazards

Flushing

→ I need to flush the instruction fetched after a taken branch or a jump instruction

- I can do this relying on the c_PCSrc, if it is '1' it means I want to choose the addres coming from the branch_addr alu so it means I shoudn't have taken the pcplus4 hence I need to flush.
- To flush all i need to do is just inside the *IF_ID* register I put an if condition and set every output port to 0 in case c_PCSrc ='1'

```
ghdl -a archer_pkg.vhd
ghdl -a add4.vhd alu.vhd branch_cmp.vhd control.vhd immgen.vhd lmb.vhd mux2to1.vhd
pc.vhd regfile.vhd rom.vhd sram.vhd CSRfile.vhd branch_addr_alu.vhd IF_ID.vhd
ID_EX.vhd EX_MEM.vhd MEM_WB.vhd dataForwarding.vhd mux3to1.vhd
archer_rv32i_pipelined.vhd archer_rv32i_pipelined_tb.vhd
ghdl -e archer_rv32i_pipelined_tb
ghdl -r archer_rv32i_pipelined_tb --wave=wave.ghw
```

Useful information

- if signal(4 downto 0) ='11111' and you do +1 you will get signal="00000"