## Preliminairies

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
ghdl -a archer_rv32i_single_cycle.vhd archer_rv32i_single_cycle_tb.vhd
```

Note that I could analyze them all in one line but I prefer this for neatness
Also note that new components may be added through the project, they will also need to be analyzed

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

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 )

- We 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 haliyan)

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

- We are 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 we will create a register file of size 4096.
- We will assume we do not need to keep track of when the code stops running, ie when the last instruction is being executed. We will just assume the clock cycles are just enough. (We could potentially tell when the coded ends by keeping track of the ecall)
- The counting is being done inside the CSRfile to allow storing the values during the same clock cycle.
- CSR_cycles: We need to keep track of the clk and rst signal
- CSR_instret: We will assume we need to keep track of when an instruction got fetched
  We thought of doing it in a naive way to keep track of when an instruction finishes in case there may be cases where instructions are fetched then flushed (not executed). We would do it through checking what type of instruction it is and whether it completed it's purpose or not (example if it's a store instruction, we will check if it finished accessing memory) But it is very time consuming and not efficient

## Code Modifications

→ In the processor file: Instantiated 2 new components

- We imported the component CSRfile and maped the signals accordingly.
- We also imported a new mux (CSRMux), for the purpose of choosing the output of the CSRfile to go into inputB of the ALU in case it is a CSR instruction
  → In the control signal: Created 2 new signals
- CSR: To indicate whether this is an 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 would allow 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

- We will assume that once the instruction is fetched it will be executed , i.e. even with pipelining instructions will be executed non speculatively, and none will be flushed
- We counted by just saying if the instr_word I have is not a NOP then increment the counter

→ We first implemented each CSR in a process, but we noticed that VHDL does not like when 2 processes "potentially" write to the same location in storage. This is a problem we were facing when we started implementing the CSRRS and CSRRC instructions (which needed to write back to the CSRfile)

## CSRRS/CSRRC

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

- We added 2 new control signals *CSR* which tells me this is a CSR instruction and *CSRWen* which in case it is a CSR instruction will also tell me if rs1 is not x0 to enable me to write back to CSRfile (because if rs1 is x0, it should not be allowed to write back to CSRfile)
- 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).

## Adding the instruction manually inside the ROM

→ We started by adding the instruction randomly inside ROM. Turns out we added it after *addi a0,t0,0* (inside loop, before jal fib_rec)
→ While reading the code we noticed that once it reaches the *j loop* instruction, it should go back to loop (meaning the first instruction in loop which is *bge t0,t1,end*) but instead it goes back to the second instruction (*addi a0,t0,0*)
→ Turns out the jump instruction takes the value of an offset (we see the word loop but actually it is a negative offset), so it's going up 6 instructions. But since we added an extra instruction we would want it to go up 7 instructions.
→ A fix could be changing the offset of j instruction and adding to it *-4*
→ We decided to add the instruction right before the ecall (after *addi a0,x0,10*).
We had to change the offset of the jal intruction that was inside the loop, it was 28 we made it 36 to account for the two insrtcutions we added (CSRRS and CSRRC)
This way each instruction would run only once.
NB: The number of clock cycles is just enough to finish the code (as mentioned earlier the ecall doesn't even get fetched). So we may need to add the number of clock cycles in the test bench.

→ We used the following 2 instructions code:

- CSRRS: X"C002AFF3" --Note that rs1=5=t0 and we know its value is 11 at the end
- CSRRC: X"C022BFF3" --Note that CSRRS changes cc counter, CSRRC changes the instruction counter

# 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 Command

```
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 multALU.vhd multALUbus.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

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 archer_rv32i_pipelined.vhd archer_rv32i_pipelined_tb.vhd && ghdl -e
archer_rv32i_pipelined_tb && ghdl -r archer_rv32i_pipelined_tb --wave=wave.ghw
```

## 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 us there is an instruction in the multiply unit. We check if the current instruction in ID is a M extension instruction we let it pass so "Stall<=0", else we stall it so "Stall<=1"
  - Nop: This is gonna be used for sendind 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 of the WB register, and the initial instruction will therefore only be used for reading
- We also modified the instruction counter to read from the instruction_WB not EX
- The condition of instr_word_WB(0)=0 or 1 is used to make sure we don't count if the signal is undefined

## 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'

## M extension instructions

- He asked us 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:
  - We made it in a way to support multiply pipelining
  - The only signals that we 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 1 when MExt_EX1 becomes 1
  - 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 EX4 and the instruction in EX5 are the same one.
  - This is 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

# Useful information

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

# Questions for Mazen