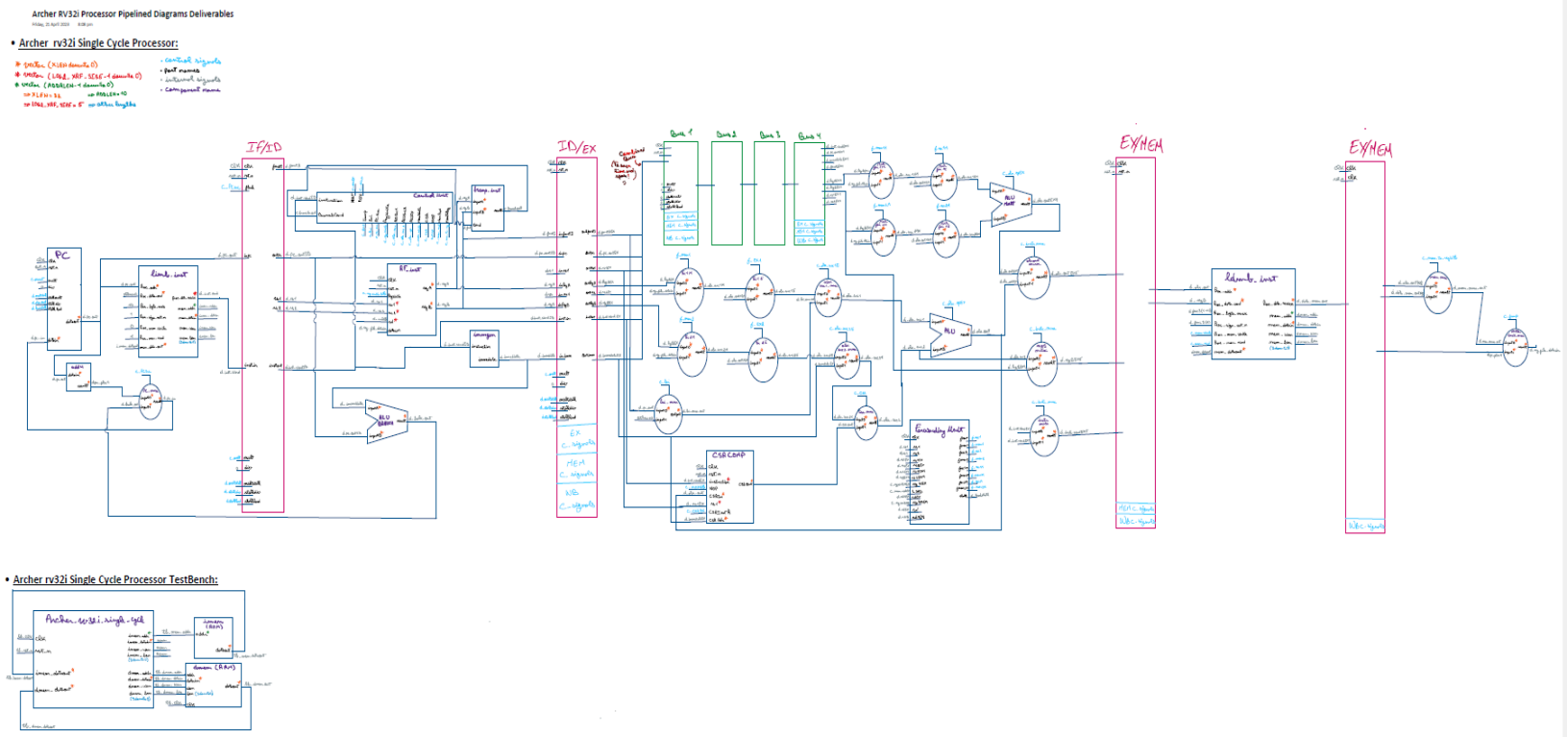
Nadim Dib Christian Abou Mrad

EECE 421 – Archer rv32i Project (Part 2) Report

Datapath (with pipeline, forwarding and hazard detection):



Disclaimer: The resolution is very bad in this picture due to its large size. Please find the picture in the PDF that is in the same folder as this document.

Pipelining, Hazard Detection and Forwarding in the Datapath:

I. IF/ID Bus (and changes to IF):
   a. Passes the data from the instruction fetch phase to the instruction decode. Notably, the d_inst_word and the d_pc_out.
   b. Notice that we did not pass d_pc_plus4 since we can manually add 4 to d_pc_out when we need it later.
   c. A bunch of signals were also created and passed such as:
      i. mult - indicates if multiply is fetched.
      ii. div - indicated if divide is being fetched.

iii. stallmult - indicated in case we need to stall for a multiply.
iv. stalldiv - indicated in case we need to stall for a divide.
v. stallload - indicated in case we need to stall for a load.

d. It outputs the registers that are needed for the instructions for them to be decoded, funct3 which will determine if there is a branch instruction and the rest of the mentioned signals from the fetch stage.

II. ID/EX Bus (and changes to ID):

a. Passes the data from the instruction decode phase to the instruction execute.
b. Added 2 control signals: MULT and DIV to indicate when a multiply or division are taking place.
c. Added **Branch ALU** which, in the case of a jump, jump and link or branch, will calculate the address to jump/branch to.
    i. It takes as inputs: the offset (immediate) and the current PC value (d_pc_outID) and outputs the address to go to.
d. The signals computed (as well as the ones unchanged from the IF stage) are passed and renamed through this bus such as:
    i. d_rs1, d_rs2, d_rd (the register addresses needed)
    ii. d_regA, d_regB, (the value inside of the previous registers)
    iii. d_immediate (immediate generated by **immgen**)
e. All the control signals needed for the instructions are issued in the ID stage and are also passed through the bus in bundles divided into: Execute control signals, Memory control signals and Write Back control signals.

III. EX/MEM Bus (and changes to EX):

a. Passes the data from the execute stage to the memory stage.
b. **Four buses** were added to this stage to simulate the multiply and divide latency.
    i. Bus 1 takes all the signals (including the control signals) from the previous stage bus (ID/EX) and passes them onto Bus 2 and so on until bus 4. No calculation is taking place. (The purpose of these buses is to waste clock cycles since the multiply executes instantly in VHDL.)
    ii. Bus 4 passes the same signals that were imputed to a **series of muxes** (which we will discuss later in this same section)
    iii. After the selection process the source registers are sent to the **ALU Mult** where the multiplication takes place.
    iv. It is then passed into the **alu_out mux** which decides which ALU output will be chosen (based on the type of the instruction).
c. A **Forwarding Unit** was also added in order to pass data from one stage to the other.
    i. It takes as input the registers rs1, rs2 and rd from multiple stages which and internally decides which ones are going to be forwarded based on the instruction type being issued.
    ii. It outputs a multitude of control signals, each one selecting the origin of the forwarding and its destination since we can forward either from the EX/MEM stage or MEM/WB stage.
    iii. All those generated signals are directed towards muxes to pick the right register to operate on.

d. A multitude of **muxes** were added, they may look overkill but there is a reason.
   i. *Reason:* We did not have a three-input mux ready, so we decided to use two muxes to simulate one three-input mux. So technically each two muxes with a "for" in the name (for=forwarding) represent one "decision" all in all.
   ii. The **forwarding muxes after the 4 buses** are dedicated for the forwarding of values for multiplication/division instructions (rs1 and rs2) or just pass the normal value in case forwarding is not needed to the **ALU Mult**. (Two for rs1 decision and two for rs2 decision)
   iii. The **forwarding muxes after the EX/MEM bus** are for the same purpose as the ones mentioned previously but for the other operations that go to the normal **ALU**.
   iv. We then added an **aluout mux** that picks the appropriate value between the two ALUs based on the instruction being executed.
   v. Added a **regB mux** and an **instr mux** which are both used in forwarding. They are respectively responsible for the regB and the instruction passing.

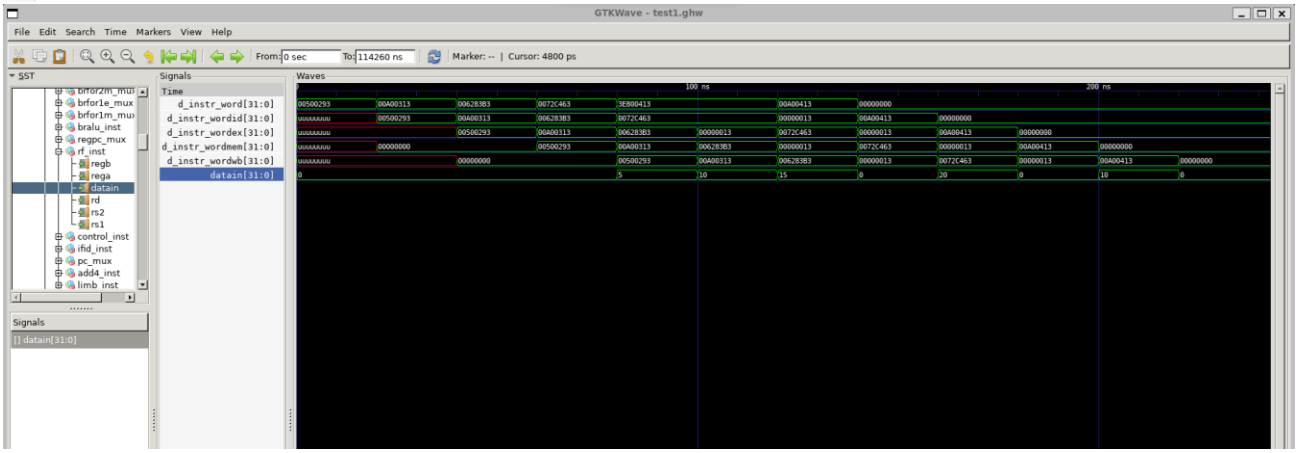
IV. EX/MEM Bus (and changes to MEM and WB):

a. Passes the all the signals from the execution stage to the memory stage.
b. At this point the EX-control signals have been used so we only pass the MEM and WB control signals.
c. Nothing changed in the memory stage 😊.
d. Nothing changed in the write-back stage 😊.

Tests and Results:

I. Test I:

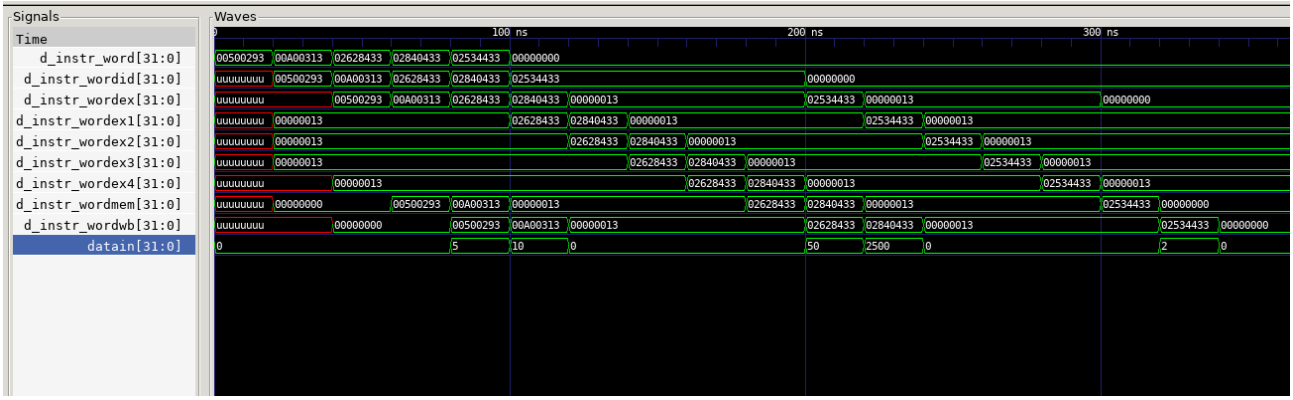
```
addi x5 x0 5
addi x6 x0 10
add x7 x5 x6
bgt x7 x5 branch
addi x8 x0 1000
branch:
addi x8 x0 10
```



In this test, we can see the functionality of multiple things: first, the pipeline working, second, data forwarding between instruction 3 and instructions 1 and 2, third, data forwarding between the branch and instruction 3 where the branch had to stall first because instruction 3 was still in EX stage, and finally, we can see the flushing of instruction 5 after the branch was taken and instruction 6 should be executed.

II. Test II:

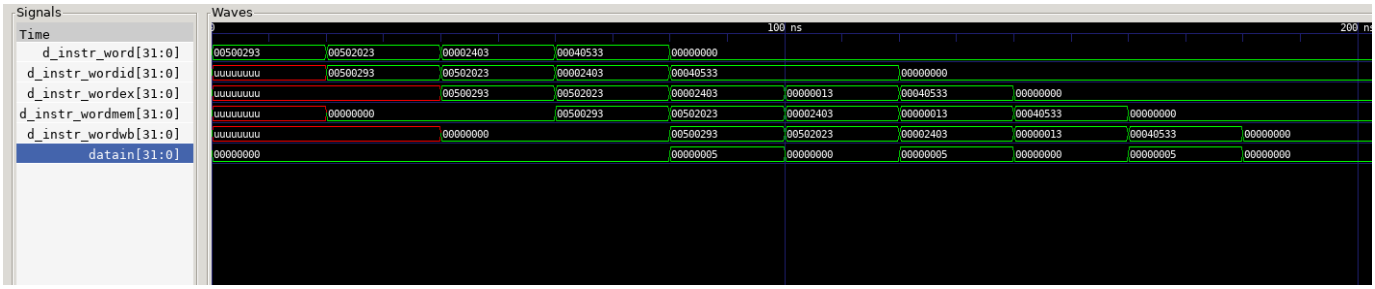
```
1 addi x5 x0 5
2 addi x6 x0 10
3 mul x8 x5 x6
4 mul x8 x8 x8
5 div x8 x6 x5
```



In this test, we see the pipeline for the mult and div instructions where we can see that mult are being pipelined but div is not being pipelined with them, and the following instructions are being stalled until mult and div are done executing. We can also see data forwarding between the mult instructions.

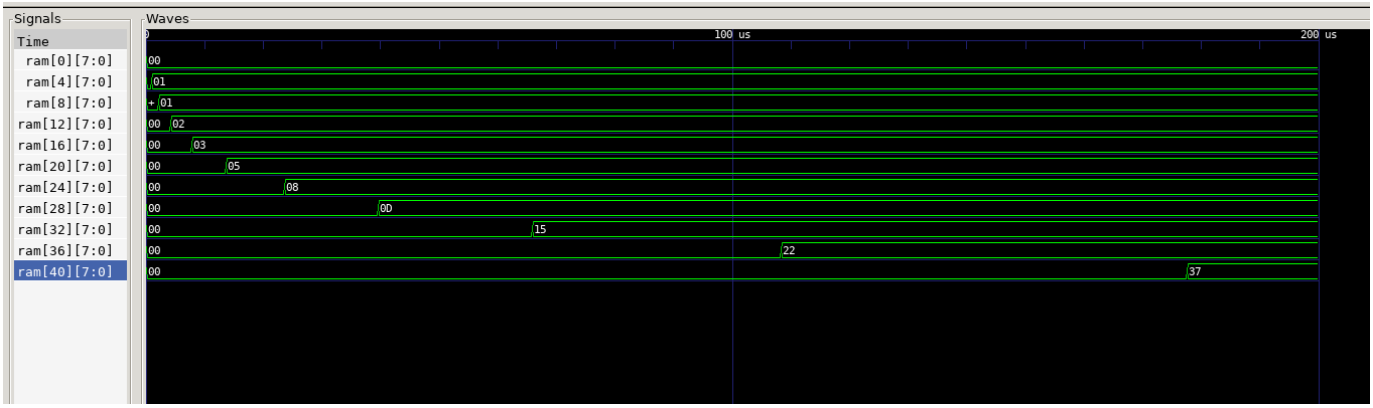
III. Test III:

```
1 addi x5 x0 5
2 sw x5 0(x0)
3 lw x8 0(x0)
4 add x10 x8 x0
5
```



In this test we can see the load instruction caused a stall because it had to do the MEM stage before it could do data forwarding to the following instruction dependent on it.

Fibonacci code:



Still working as intended but needed to add a lot of extra clocks.

Contributions:

The coding part was done in parallel and together as we gave ideas to each other and coded on one machine. We drew the Datapath together and reviewed it together to catch any mistakes. Chris provided the tests and results for the report and Nadim wrote the description for the changes.