Tomasulo’s with Speculation

Chris Ban and Dalin Williams

CPHE-533

5/11/15

# 

# Work breakdown

**Regfile**:

code: DJ + Chris + Class

simulation: Chris

**ROB**:

code: DJ

simulation: DJ

**Load/Store Buffer**:

code: DJ

simulation: Chris

**Reservation Station**:

code: DJ

simulation: Chris + DJ

**CDB**:

code: DJ

simulation: Chris

**Issuer**:

code: Class + DJ + Chris

simulation: DJ + Chris

**Memory Unit**:

code: Chris

simulation: Chris

**Address Unit**:

code: Chris

simulation: Chris

# Regfile

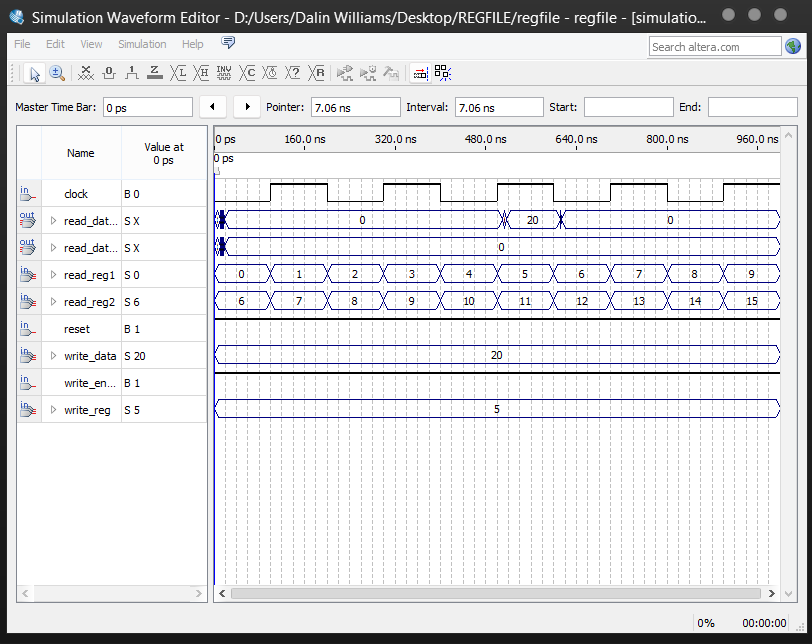
The regfile records the current values of each register. Once an instruction’s result has been committed the result will be updated in the regfile.

IN:

* clock/reset - clock, reset signals
* write\_enable - signals whether it is valid/acceptable to write
* read\_reg1, read\_reg2 - signals designating register number to access
* write\_reg - the register to write to
* write\_data - the data to write

OUT:

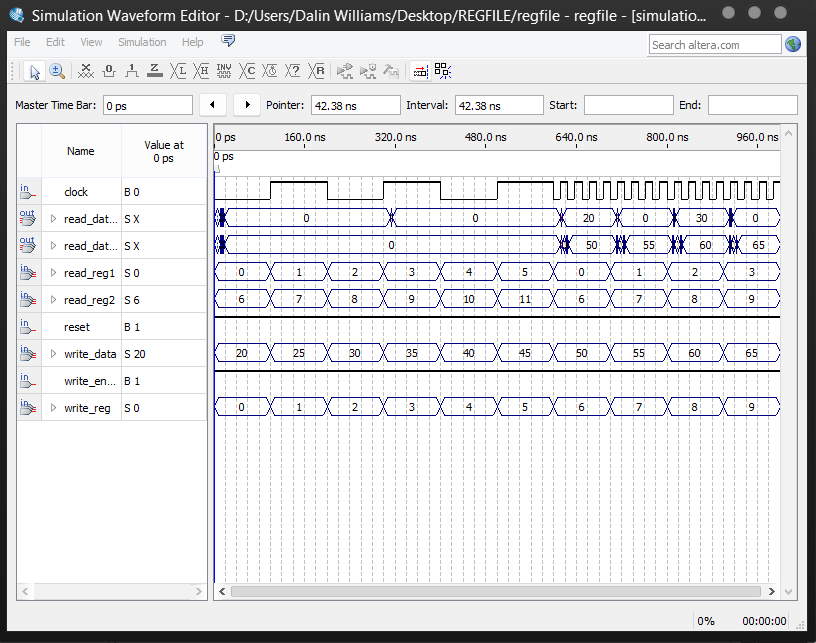
* read\_data1 - signal used to retrieve reg data based on read\_reg1 input
* read\_data2 - signal used to retrieve reg data based on read\_reg2 input



Regfile showing multiple entries being updated.

In this simulation, we set write enable to high, which allows for writing data. We set the write data to 20 here and the reg number to 5, and then using a test signals read\_data and read\_reg, we look at the value of the register to verify that the data was written into the register array and persists across cycles.

The value 20 is written to reg 5, and then we look through each register. Once read\_reg1 hits 5, our test signal read\_data1 looks at reg 5 and spits out the value 20 for that clock cycle confirming expected behavior.



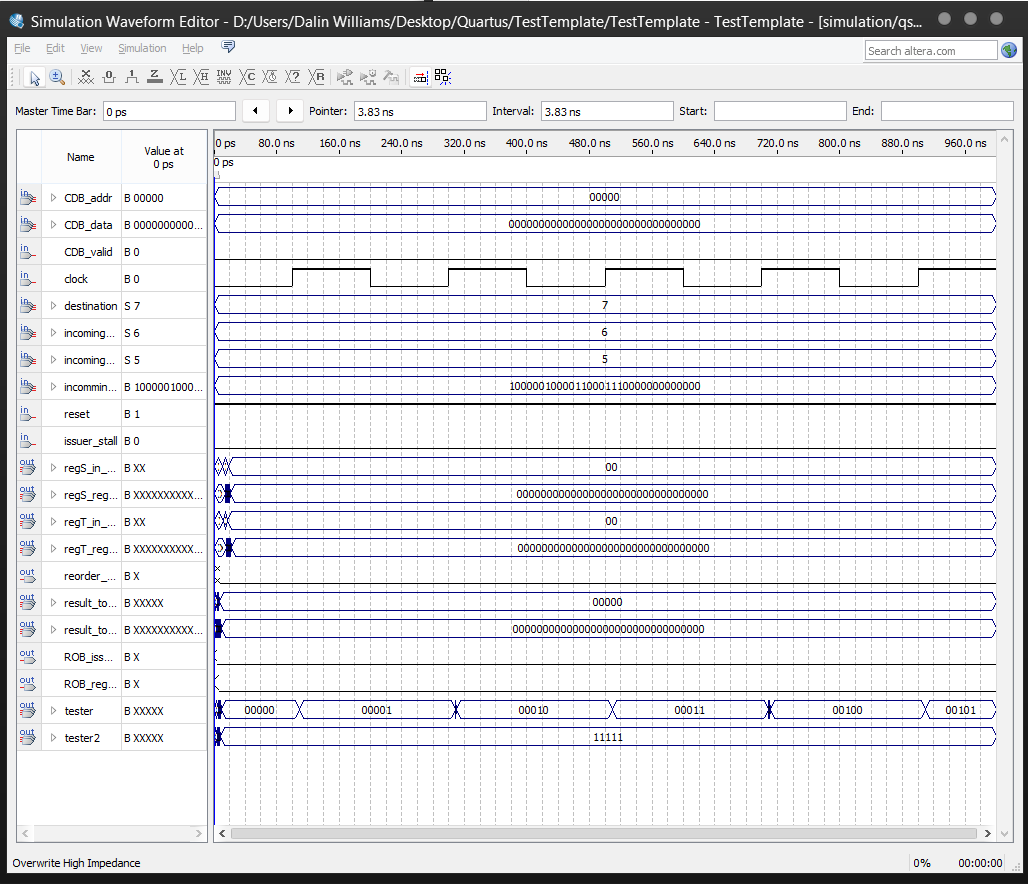
In this simulation we write a bunch of data to registers then, similar to the previous simulation, we check the registers to confirm expected behavior. As you can see, read\_data2 reports 50, 55, 60, etc. corresponding to the values write\_data and write\_reg were passed.

# ROB

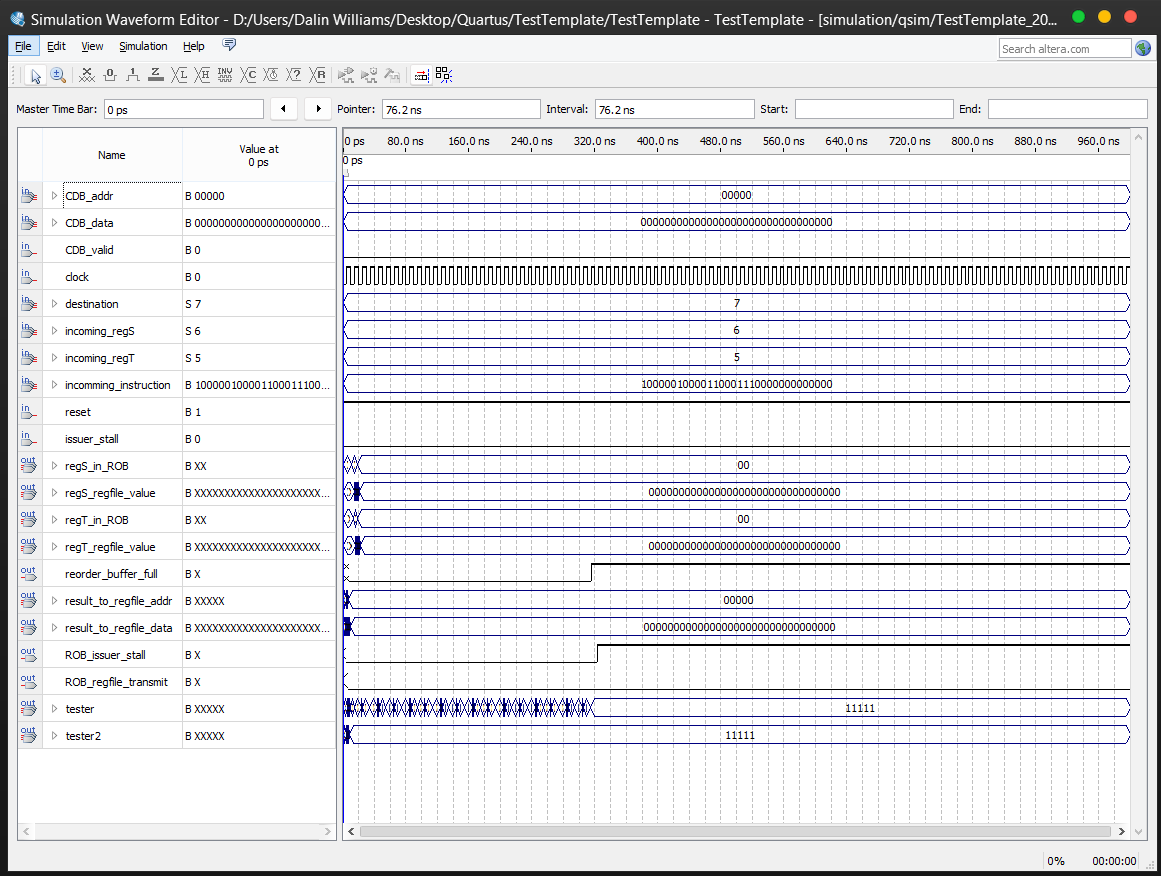
The reorder buffer is a circular holding and reordering station/buffer for instructions issued from the issuer. Once an instruction has been entered via the issuer, the ROB will seek a value for its registers and register t entries, and returns the index of the entered instruction if it does not exist, or the value from the CDB if it does. The ROB will not accept any incoming instructions if it is full. This is determined via the head and tail pointers of the buffer. If these are equal, the ROB is full. Finally, a ROB entry will update as the CDB broadcasts the value of said entries r or t entries as well as the computed result of the instruction. Once an instruction has completed, it is marked as done via the ready bit. Finally, an instruction in the CDB will not commit unless it is in order. This is to say that the entry which will commit will be the oldest in the ROB. Once this instruction commits, the ROB entry is committed to the register file, and the ROB entry is cleared.

The I/O for this module is as follows:

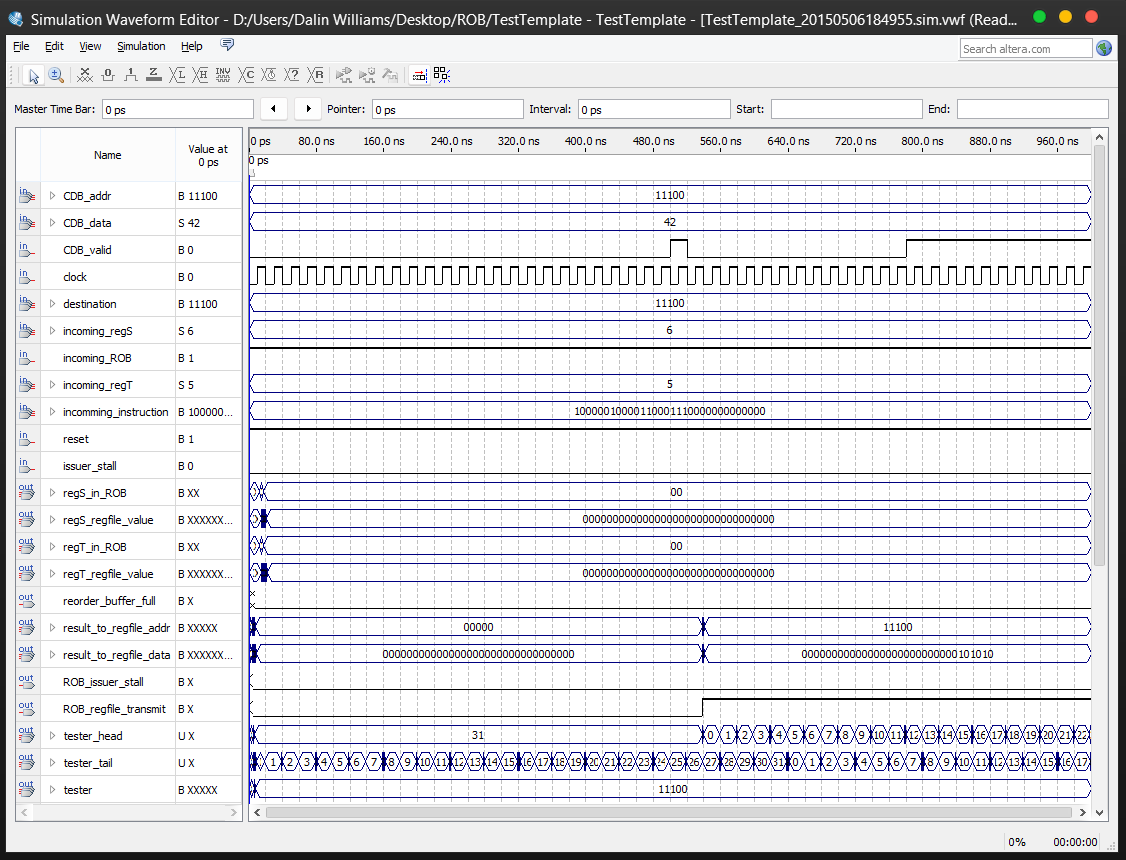
* IN
  + Clock – The clock for the selected module. This clock would normally be set to the global clock
  + Reset – A signal which signifies the processor is being reset
  + Incoming\_instruction – The incoming instruction from the issuer
  + Destination – The register address at which the value of the operation will be stored
  + Incoming\_regS – The register address which holds the operation’s s register value
  + Incoming\_regT – The register address which holds the operation’s t register value
  + CDB\_valud – A single bit which signifies the incoming of data across the CDB
  + CDB\_addr – The address correlating to a register which the data on the CDB represents
  + CDB\_data – The data which is broadcasted across the CDB
* OUT
  + ROB\_issuer\_stall/Reorder\_buffer\_full – A signal that is sent when the ROB is full
  + ROB\_regfile\_transmit – A signal that is sent when the register file is being written to
  + regS\_in\_ROB – A signal sent to the issuer giving information if the s register is in the ROB
  + regT\_in\_ROB – A signal sent to the issuer giving information if the t register is in the ROB
  + regS\_regfile\_value – The value returned from the ROB upon query for a validly stored register s is the ROB – this is the data of register S
  + regT\_regfile\_value – The value returned from the ROB upon query for a validly stored register t in the ROB – this is the data of register T
  + result\_to\_regfile\_addr – The address of the register returned to the register file
  + result \_to\_regfile\_data – The data returned to the register file



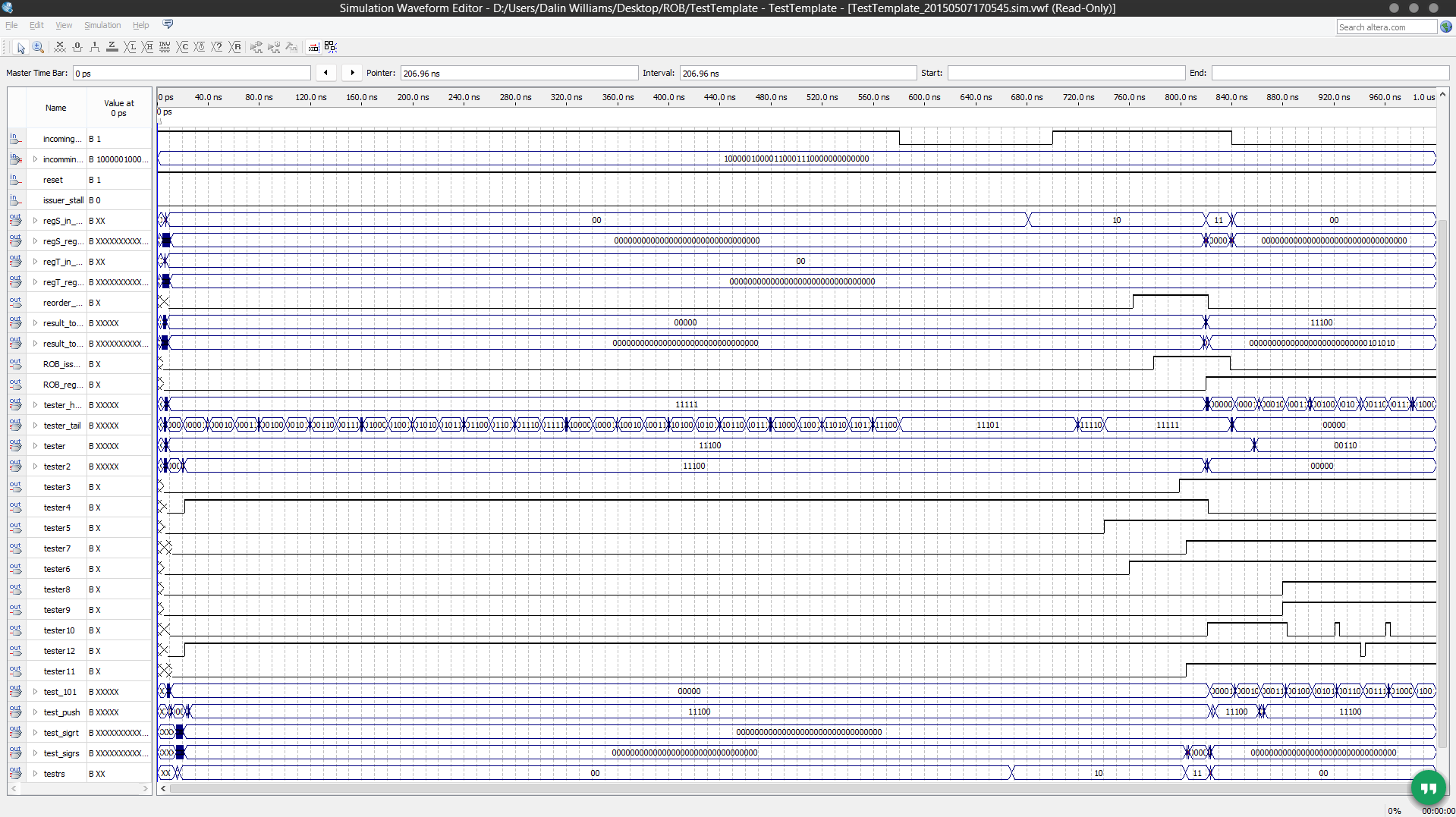
In this example, we can see the ROB having items being added to it. This can be observed by the values of tester and tester2 (pointing to the values of tail and head respectively). In this example, head remains constant while tails increments on each rising edge, a result of increasing number of entries in the ROB.



In this example, the ROB is filled with data. We can see the ROB being filed by again observing output tester (tail) and output tester2 (head). Once the two are equal, ROB\_issuer\_stall/reorder\_buffer\_full go high, preventing the insertion of any further data into the ROB.



As we see in this example, the CDB assists in emptying the ROB. Note that all incoming instructions have the same destination address. This being the case, when the CDB gets an entry with the same address as the destination, it populates all 32 of the ROB’s entries, and proceeds to empty at a rate of one per clock cycle. This is accompanied by the ROB gaining and setting new entries which will not be done until all old entries are emptied and written to the register\_file.



This diagram illustrates all interactions on the ROB. This includes interactions with the issuer, CDB, and register file. The issuer requests data corresponding with data that does and does not exist within the ROB, and the ROB responds correctly as defined in our code. This response, in return, generates either the instruction’s index within the ROB, or the data stored within the register referenced at the location.

# Load/Store Buffer

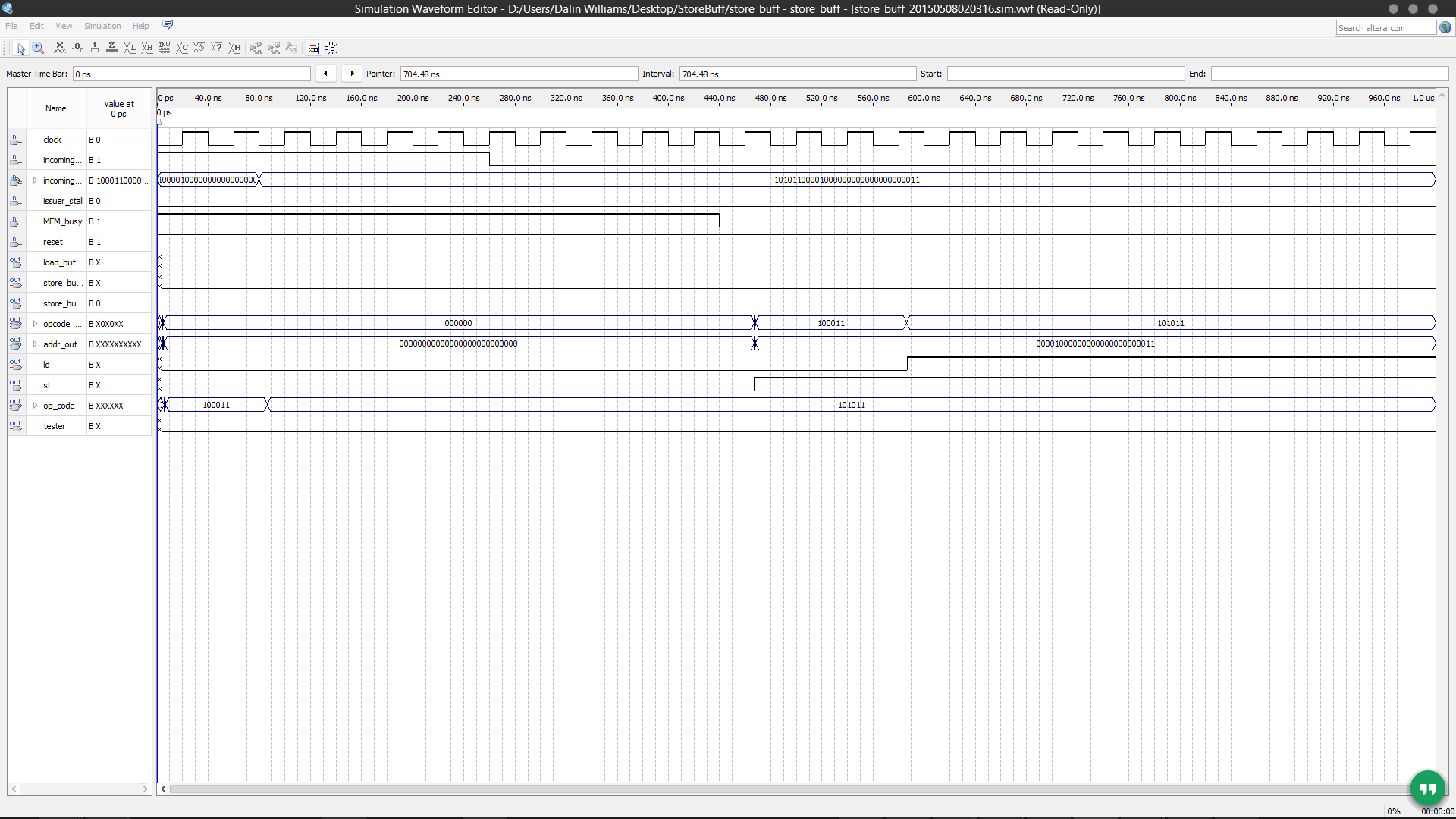
description

**IN:**

* Clock – clock
* incoming\_indicator – signals new incoming
* incoming\_instruction – new instruction
* incoming\_address – new address
* MEM\_busy – specifies if mem is available

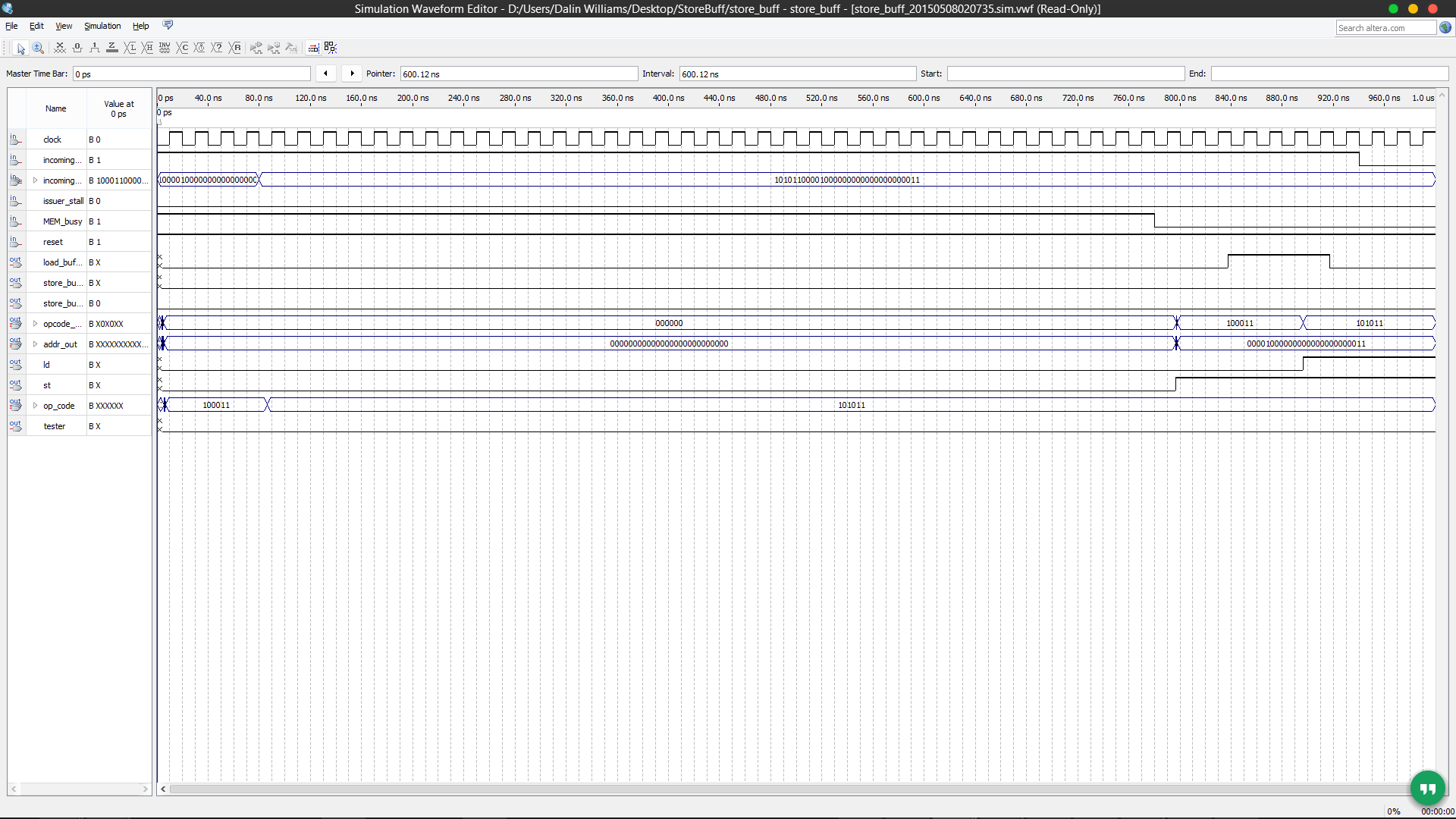
**OUT:**

* opcode\_out – specifies opcode of ins
* addr\_out – specifies address
* store\_buff\_full, store\_buff\_empty – specifies whether buffer is full/empty
* load\_buff\_full – specifies whether load buff is full
* ld, st
* op\_code



After speaking with Dr. Hall, we came to the conclusion that we would assume that the code being run will not create hazards if all writes are completed first, then reads are done in order to prevent hazards.

In this simulation, we show the load/store buffer managing load and store instructions. Once MEM\_busy goes low we perform all stores, then all loads due to the aforementioned assumption.

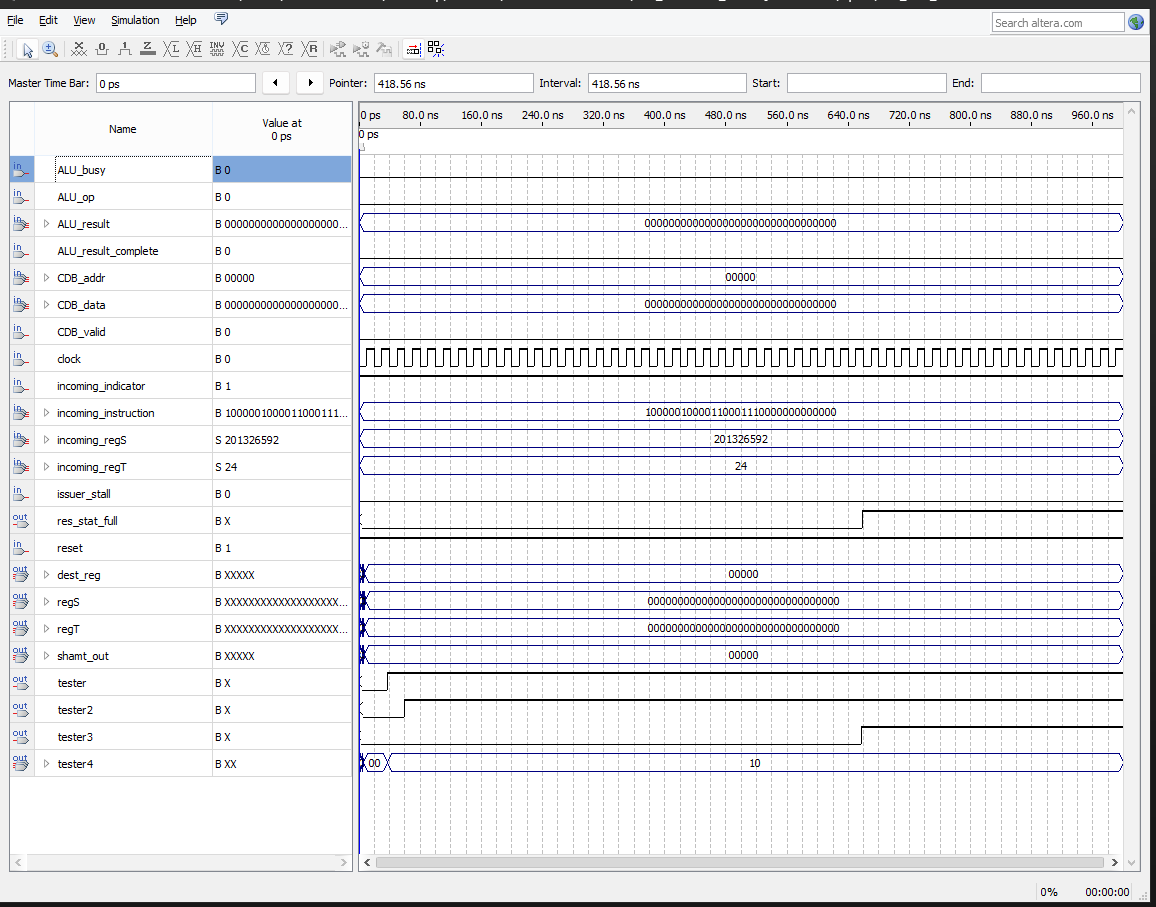


This shows the load buffer being filled first (notice load\_buffer\_full high) and then empties after stores are cleared.

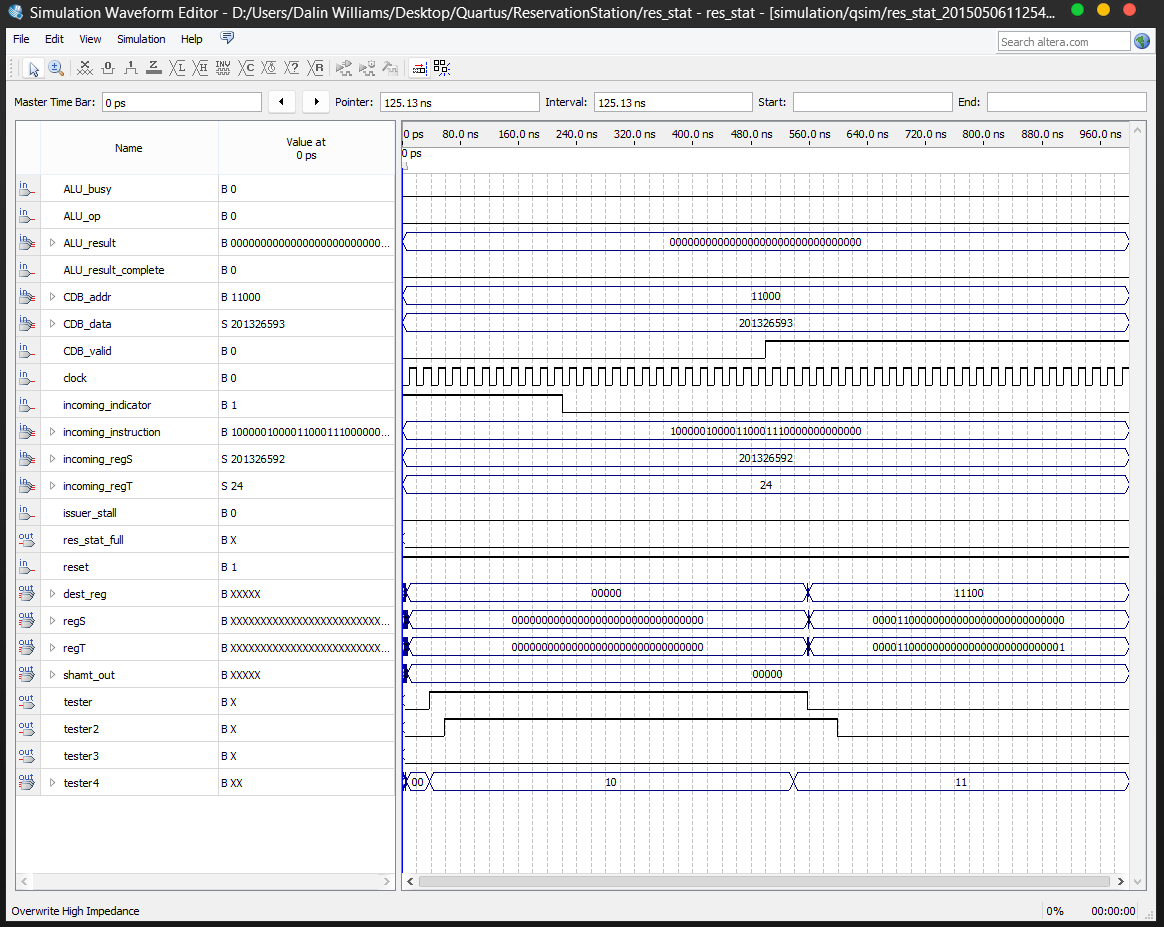
# Reservation Station

The reservation station stores and holds the instructions issued by the issuer. Once instructions are issued and the ROB is not full, an entry is placed is the reservation station. This entry may have a ROB entry in the data local for the data of the s and t registers. If there is a ROB entry, then the entry must wait until the value is flashed across the CDB. This CDB flash must contain the address of the entries corresponding s or t registers. If it does, the entry is updated. If the entry had all valid data for its instruction registers, the entry is then placed within a pipeline for execution in the respective ALU unit. The pipeline serves as a secondary staging ground for completed instructions. This staging ground is only for complete and ready to run instructions. Instructions within the pipeline have a timer which, when equal to zero, with proceed to check if the cost is clear to run an operation on an ALU/logical unit. If the logical unit is not full, the pipeline entry is emptied into its respective unit and computed. Upon the data being transmitted across the CDB, the pipelining segment holding this value will then delete it from the pipeline.

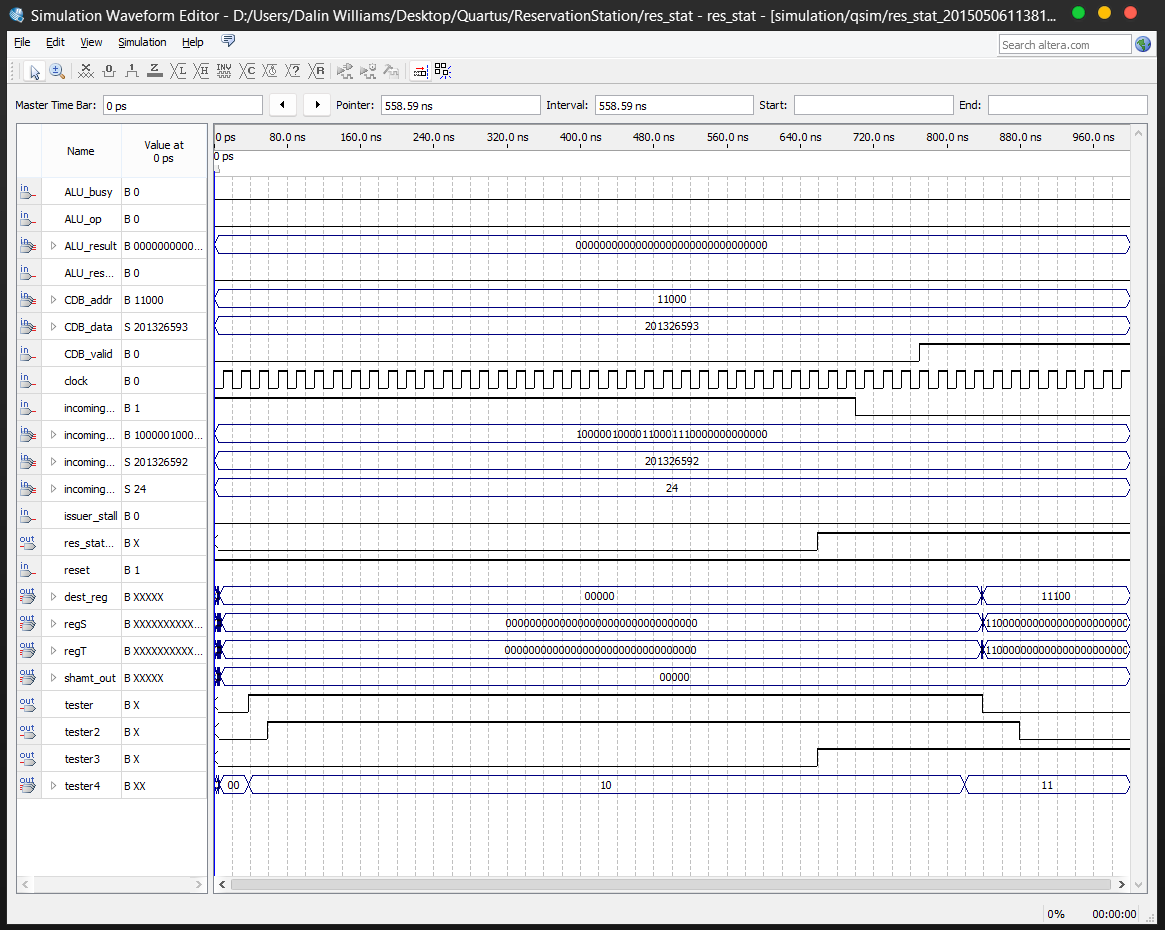
* IN
  + Clock – The clock for the selected module. This clock would normally be set to the global clock
  + Reset – A signal which signifies the processor is being reset
  + Incoming\_indicator – A single bit which specifies that an instruction is inbound from the issuer
  + Incoming\_instruction – The incoming instruction from the issuer
  + Incoming\_regS – The register address which holds the operation’s s register value
  + Incoming\_regT – The register address which holds the operation’s t register value
  + CDB\_valud – A single bit which signifies the incoming of data across the CDB
  + CDB\_addr – The address correlating to a register which the data on the CDB represents
  + CDB\_data – The data which is broadcasted across the CDB
* OUT
  + Res\_stat\_full – A signal that is sent when the reservation station is full
  + regS – This is the data of register S
  + regT – This is the data of register T
  + shamt\_out/fmt\_out – The data stored within the fmt/shamt portion of the instruction
  + funct\_out – The data stored within the funct portion of the instruction
  + dest\_reg – The address at which to store the result of the operation



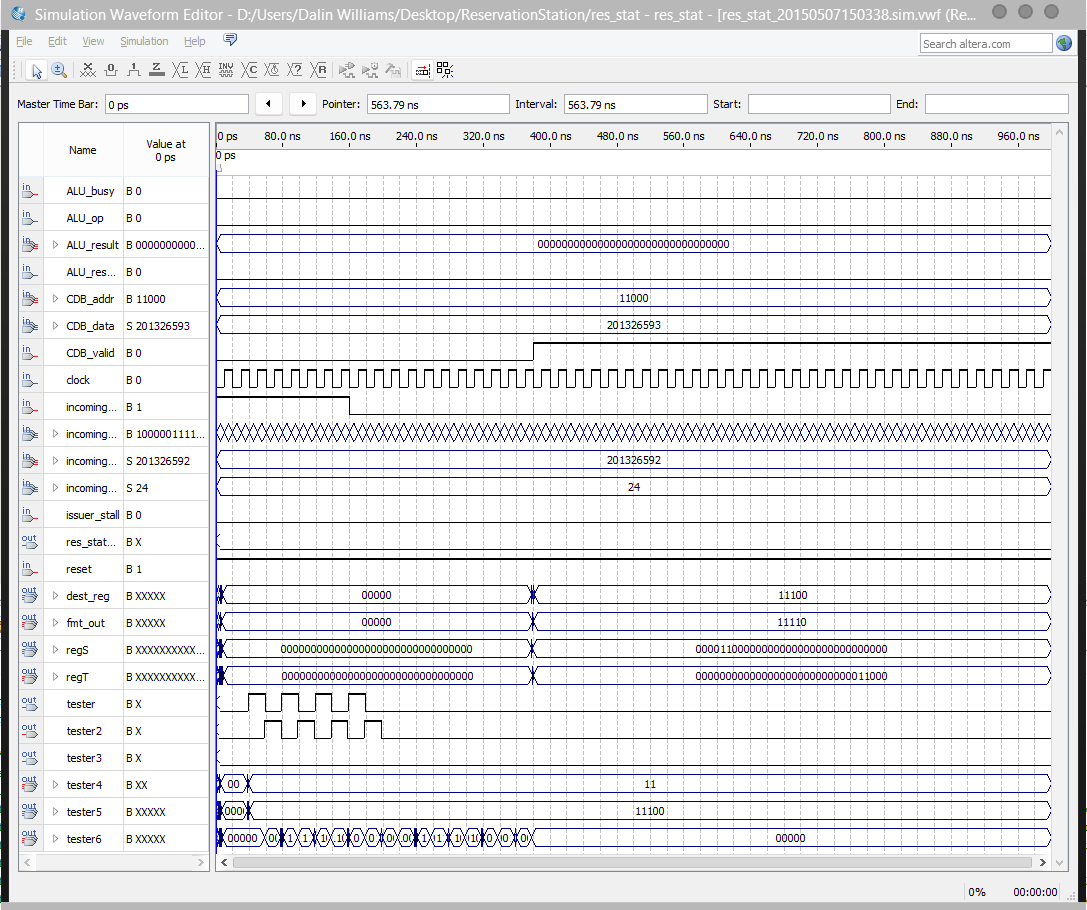
Here we can observe the reservation station being filed. Once the number of instructions currently in the buffer is reached, res\_stat\_full goes high, signifying the maximal number of instructions currently being stored is reached.



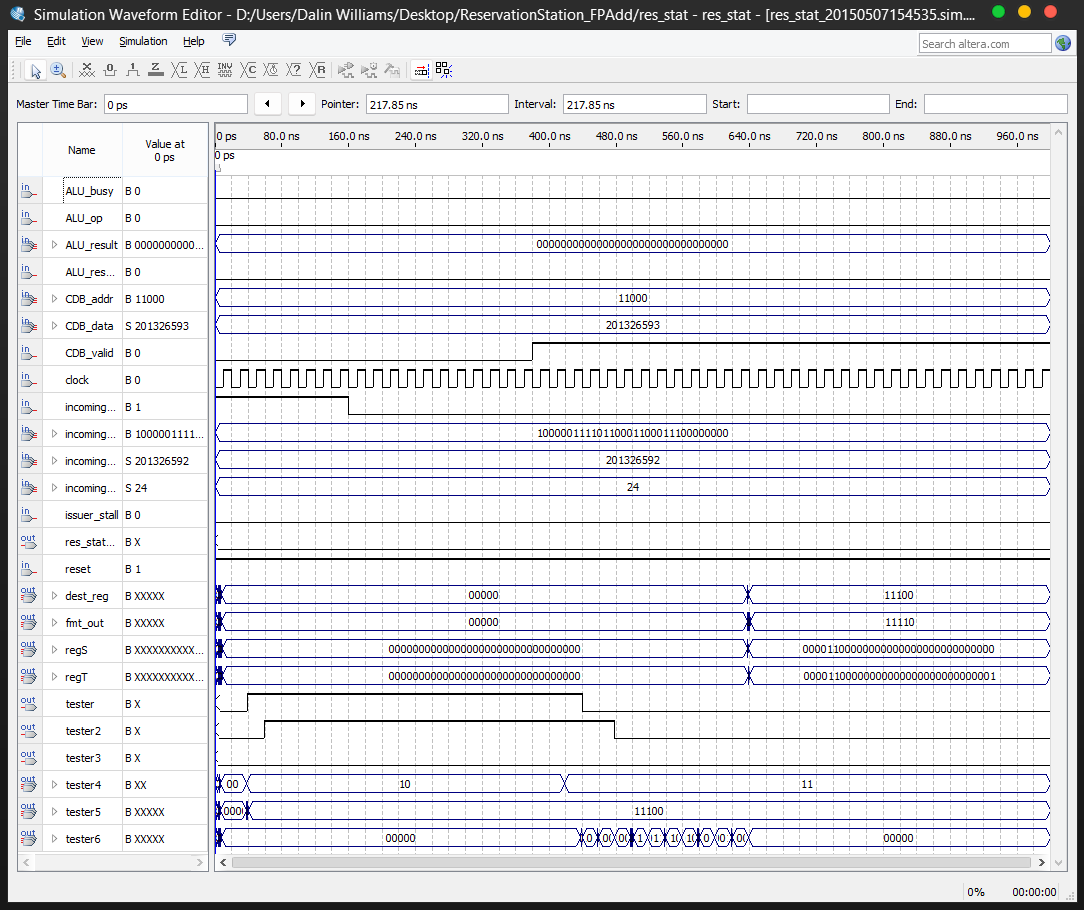
Building off the previous example, after the reservation station is full, it can only be emptied by 1. Having all of the instructions set to complete, that is they all have valid data and not index numbers, and 2. The operator unit is not busy/pipeline is not full. In this example, we see tester (representing the reservation station’s first entry’s valid bit) being set to high, and then set to low. This is due to the fact that CDB\_valid is set to high for a portion, setting the values within the reservation station to ready. These values are then instated into the pipelined queue (since this is an ALU pipeline, there is not penalty), and finally into the ALU unit itself.



Carrying off of the pervious example, only where the extraction happens after the unit is full.



FP multiply reservation station: we do not need to show other cases because the architecture is exactly the same, all we change is the time increment and the rest is handled by the code. As one may see, there is 16 cycles of waiting until the pipe is done and the data is exported.

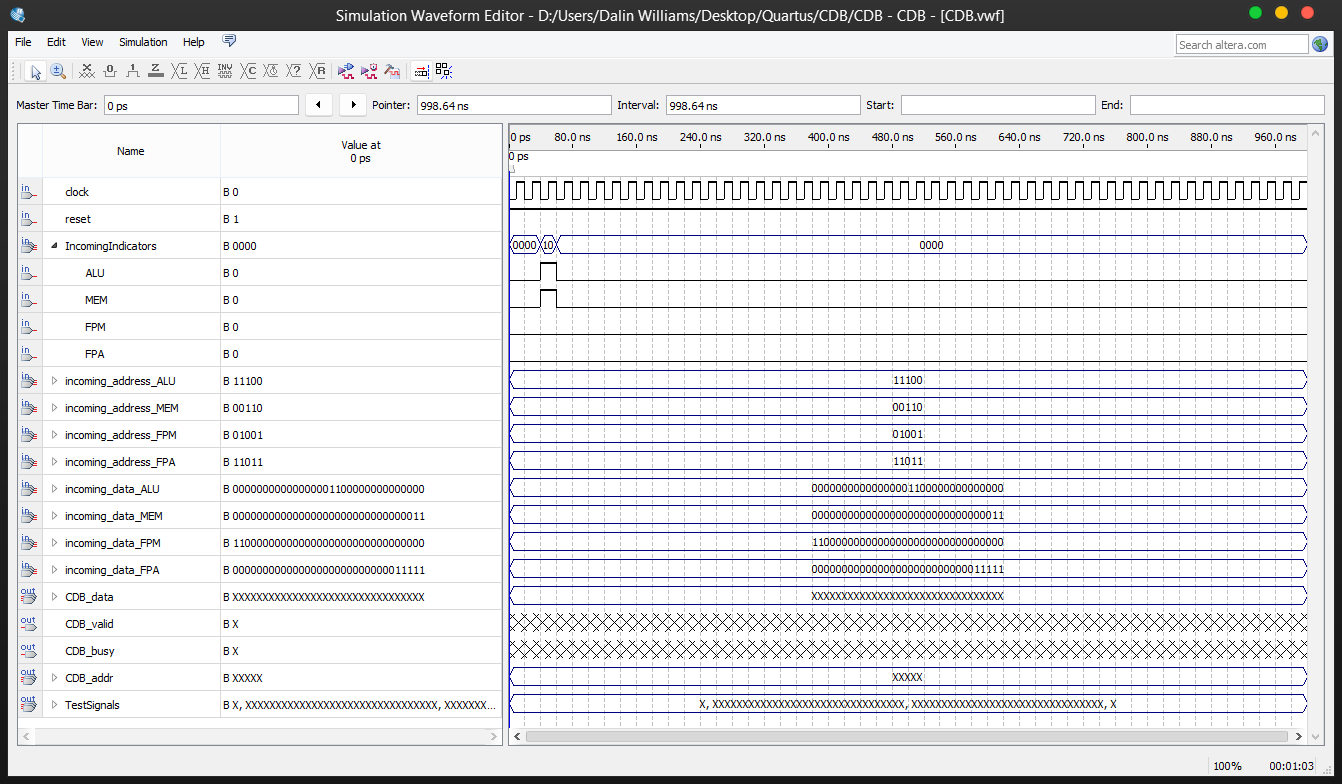


FPAdd reservation station. Again, we can see the clock (10 cycles) at tester6 after the data is made ready by the CDB.

# CDB

The CDB is the broadcasting component of Tomasulo’s Algorithm. The CDB is an interconnected bus that allows all components of the component to communicate. This is achieved via the broadcasting of results across the CDB. Whenever an operation unit completes an operation, the CDB waits a cycle, then broadcasts the data. However, the CDB must also handle multiple broadcasts in parallel, as this would cause a conflict on write, and data to be lost.

* IN
  + Clock – The clock for the selected module. This clock would normally be set to the global clock
  + Reset – A signal which signifies the processor is being reset
  + Incoming\_data\_ALU – The data incoming from a completed ALU operation
  + Incoming\_data\_MEM – The data incoming from a completed MEM operation
  + Incoming\_data\_FPA – The data incoming from a completed FPA operation
  + Incoming\_data\_FPM – The data incoming from a completed FPM operation
  + Incoming\_address\_ALU – The address incoming from a completed ALU operation
  + Incoming\_address\_MEM – The address incoming from a completed MEM operation
  + Incoming\_address\_FPA – The address incoming from a completed FPA operation
  + Incoming\_address\_FPM – The address incoming from a completed FPM operation
  + ALU – A bit specifying if an ALU operation is incoming
  + MEM – A bit specifying if an MEM operation is incoming
  + FPM – A bit specifying if an FPM operation is incoming
  + FPA – A bit specifying if an FPA operation is incoming
* OUT
  + CDB\_busy – A bit specifying if the CDB is busy or not
  + CDB\_valid – A bit specifying if the CDB has a valid value permuting across it
  + CDB\_addr – The address broadcast across the CDB
  + CDB\_data – The data broadcast across the CDB



As we can see, here is an example of the CDB handling multiple inputs attempting to write to the CDB at the same time. After the initial one clock wait, we pipe out the items in some arbitrary order as defined in the VHDL. This prevents the value within the CDB from overwriting from new items.

# Issuer

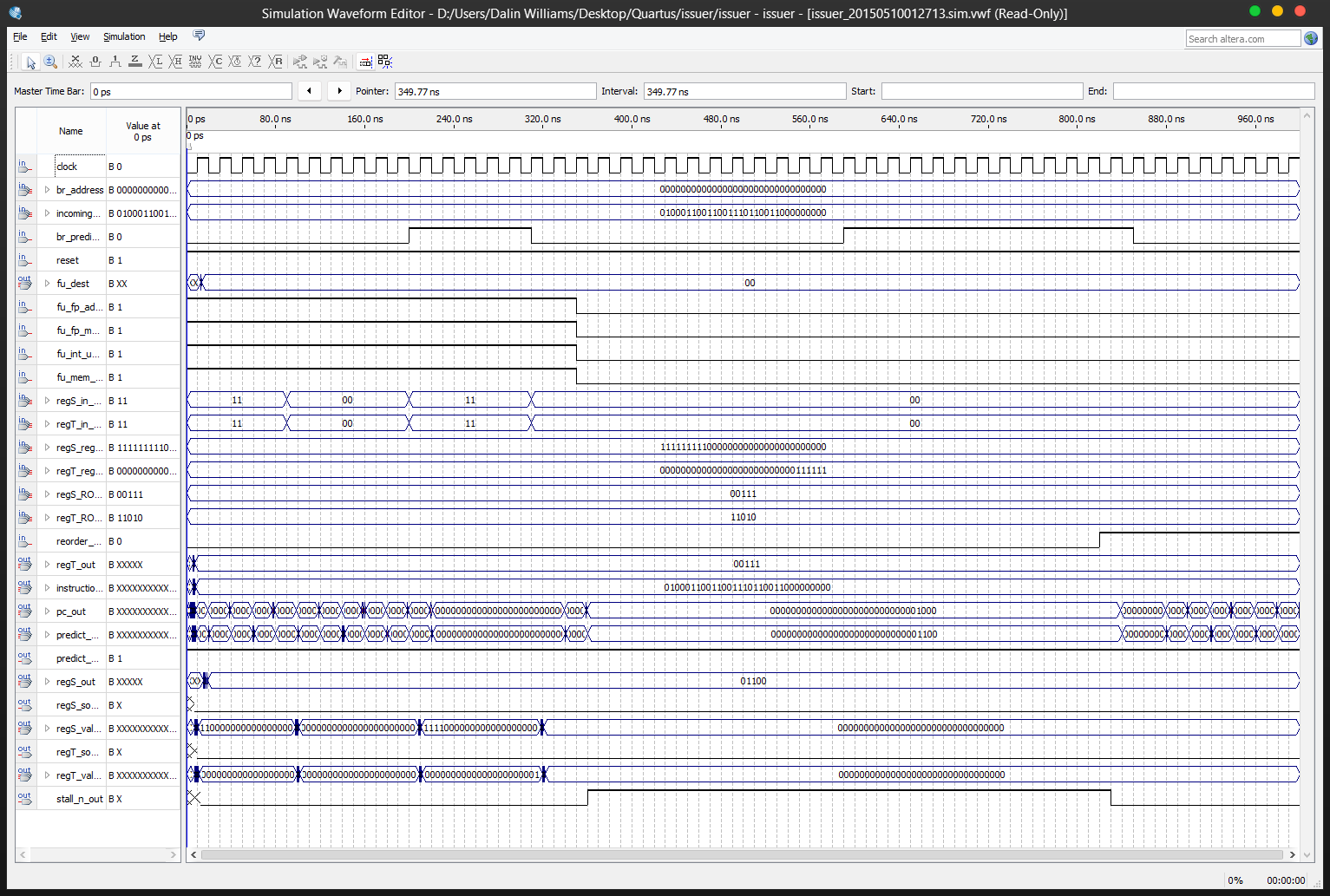
Issues instructions to the relevant unit depending on instruction and ROB status.

**IN:**

* clock/rest – clock
* fu\_mem\_unit\_fulll, fu\_fp\_add\_unit\_full, fu\_fp\_mult\_unit\_full, fu\_int\_unit\_full, reorder\_buffer\_full – do we need to stall?
* br\_predict\_fail – the flag sent from the branch predictor
* br\_address – where in the instruction queue are we
* regS\_in\_ROB – rs in rob
* regT\_in\_ROB - rt in rob
* regS\_ROB\_entry - rob entry
* regT\_ROB\_entry - rob entry
* regS\_regfile\_value – rs regfile val
* regT\_regfile\_value - rt regfile val

**OUT:**

* stall\_n\_out – pass the stall info onto the ROB or other components like CDB
* pc\_out - next pc addr
* instruction\_out - specifies instruction
* predict\_br\_taken\_out
* predict\_br\_addr\_out
* regS\_value\_out - rs
* regT\_value\_out - rt
* regS\_source\_out, '0' = regfile, '1' = ROB
* regT\_source\_out, '0' = regfile, '1' = ROB
* regS\_out, regT\_out, goes to the ROB and ROB will send a signal into reg\_in vals indicating vals are in ROB
* fu\_dest – defines the intended functional unit: 00 = adder, 01 = mult, 10 = mem, 11 = integer



This simulation shows the issuer gathering data from the ROB, and returning the relevant data to the respective reservation stations via the ROB. Also shows the stalling of the issuer upon any full signals. For example, you can see that when stall\_n\_out goes high, pc\_out does not change (stalls).

# Memory Unit

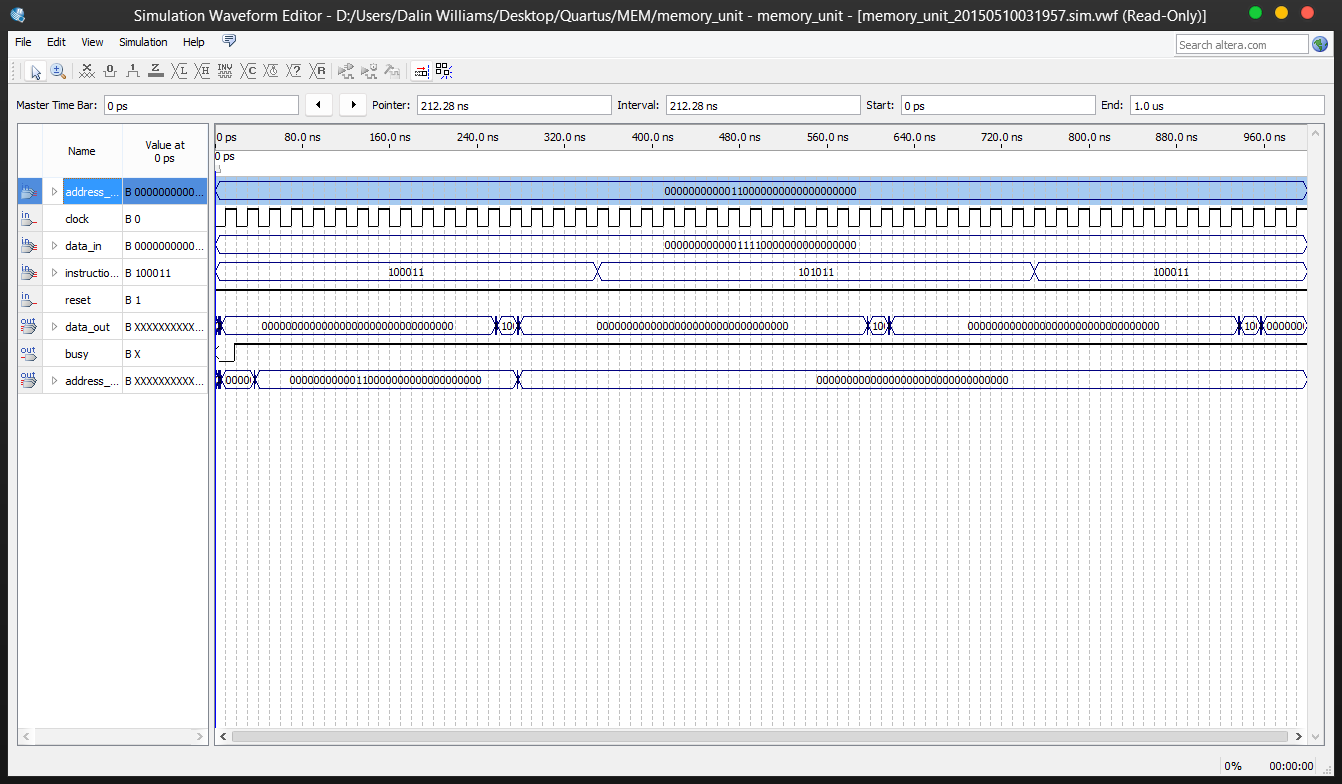
The memory unit acts as the bridge between the load/store instructions and memory. In this component we handle when a load instruction is pulling from memory and when a store instruction is pushing to memory. We simulate the interaction this component would have with the instructions and memory.

**IN**:

* clock/reset - clock
* instruction\_op - specifies type of instruction
* address\_dest – memory destination
* data\_in – memory data in (for stores)

**OUT**:

* data\_out: memory from data (for loads)
* address\_out\_to\_mem: send data to mem for fetch (loads)
* busy: busy



Shows the memory unit interacting with both loads and stores. Holds values and stays busy for a clock duration defined per-type.

In this simulation we specify a instruction\_op of 100011 and 101011 which signifies a LW and SW.

We start the latency counter (11 seconds, not shown) in order to simulate realistic timings. While busy, signals backup in/outs.

For a LW, on the first cycle we simulate memory responding to the request and set address\_out\_to\_mem to our address\_dest. Then we wait for the latency and then simulate the response from memory and set data\_out to the received data. Once busy is set low we flush outputs.

For the SW, when the latency counter reaches 0, we set our output to the passed data simulating the latency of memory for storing. Once outputted we zero out our outputs. These values are protected by the busy output which specifies whether we are waiting for values or not.

# Address Unit

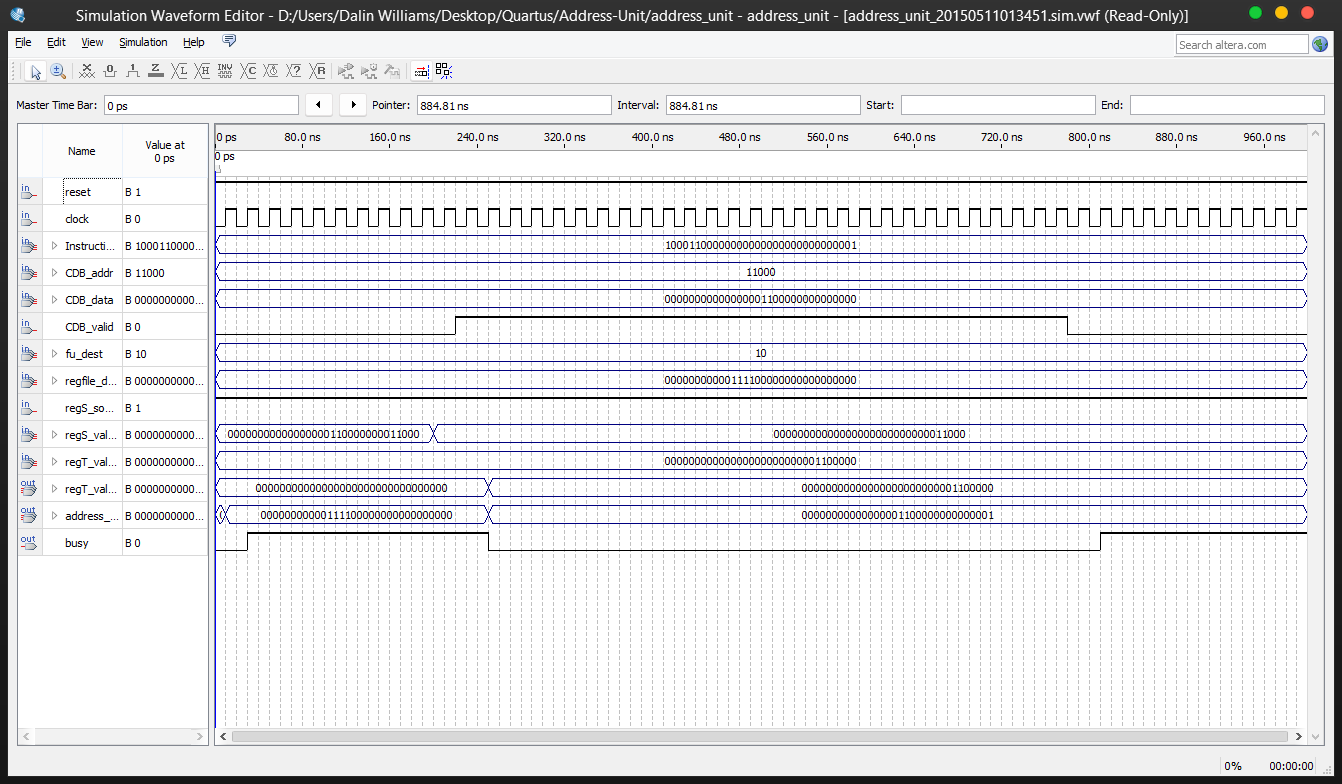
The address unit is used to calculate the address for load words.

**IN:**

* clock: clock
* Instruction\_IN – the load instruction
* CDB\_data – data field from cdb
* CDB\_addr – addr field from cdb
* CDB\_valid – valid field from cdb
* regS\_value\_in – base address
* regS\_source\_in – specefies source, 0 = regfile 1 = rob/issuer
* regfile\_data – if from reg, will hold valid data to add to immediate
* regT\_value\_in – rob entry for LW to write to

**OUT:**

* address\_OUT – calculated address meant for load buffer
* regT\_value\_out – rob entry id for lw to write to
* busy - busy



In the above example, the address unit (if not busy) takes incoming data from the CDB(depends on the source specified by regS\_source\_in) and places the calculated address into the address\_out field. Once data is output, busy is set to 0.

In this simulation we set cdb\_data and instruction arbitrary addresses. Since fu\_dest is 10 (aka meant for AUnit) and the opcode is 100011 (which specifies a LW), we wait for the relevant data to appear on the cdb. Once CDB\_addr is equal to regS\_value’s bottom 5 bits and the CDB\_valid signal is high, we calculate the address by adding cdb\_data to the instruction's immediate field (15 down to 0), assign it to address\_out and set busy low.

We can then see the expected behavior of 1100000000000000 and 0000000000000001 summed creates the output 1100000000000001. The same process is repeated if it were from the regfile as the inputs still fall into the same signals. We differentiate simply because the ROB/Issuer requires us to wait for values from the CDB.

# Other Modules/Components

Please note that we include the semi-related ALU, FP Adders/Mults/etc. components along with the aforementioned components in the resources provided.