Robert W. Sanchez

ECE438 Advanced Logic Design

Final Project

*MIPS Microprocessor*

December 10th, 2011

**Introduction:**

The MIPs microprocessor is one of the main topics that we have been focusing on this semester in class. The task at hand for our final is project is to implement the pipelined architecture of the processor using VHDL. In class we learned about the separation of data and control paths and the ways that they coincide in order to process the work that is requested by the user’s code. The pipelined structure of the system is what gives the processor the ability to handle synchronous instructions while at the same time completing these instructions in a timely manner. As a reference guide our textbook provided the base outline of the structure of the system as well useful information and hints as in how to place control and data lines. Our instructor also provided us with registers, instruction memory, and data memory VHDL segments to be used for the project.

**Final Project: MIPS Processor**

In the beginning stages of project I contemplated the different paths that could be taken in order to construct the architecture in the most organized fashion. I decided that because of the large number of signals as well as the three major components of the system already created for us (Instruction Memory, Registers, Data Memory) that I would combine all the necessary work into one top level file. I felt like this would be the most efficient way to see all the signals and registers and be able to change them if need be rather quickly. The major registers as well as the program counter were the only clock controlled devices that I made within the project; everything else was asynchronous in order to meet the timing sequence of the pipeline.

* *Top Level Registers*: In order to meet the timing constrains of the pipline “top level registers” were needed to separate the different blocks of work. In this report when I refer to top level registers I mean the registers which differentiate between different blocks of the pipeline. In other words the top level register pass values from one block to the next subsequent block on the rising edge of the clock to ensure that all the work can complete within the timeframe of one clock cycle.

I broke the code up into 5 separate blocks: Instruction Memory, Registers, ALU, Data Memory, and Write back.

*Instruction Memory Block:* The instruction memory block was my base line starting point as all instruction data had to originate there. The block was pretty straight forward as it only contained an adder/program counter register, the branch program counter mux, and the provided instruction memory. On system reset the counter starts at a value of zero and increments its way on thorough the provided input instruction memory. The only major exception that had to be handled in this stage was branches to other portions of the code. How branches were handled will be more intensely analyzed further in the report. The instruction data feed as well as the current value of the program counter were registered and passed onto the register stage of the architecture. Captured below is the basic outline of the Instruction Memory.

*Register Block:* The register block handled several functions within the pipeline including decoding, sign extension, system control, and of course reading and writing of the 32 registers of the system. The incoming instruction data is broken down and decoded into it constituent components based on the instruction format. In this project we mostly focused on “i” type formatting which contained an opcode(6 bits), source register(5 bits), transfer register(5 bits), and a 16 bit immediate field. These fields were broken up into their respective signals to be used within the register block itself. The RS and RT were used as inputs to the Register block in order to define the necessary register to handle the instructions. The controls of the register block (write address, & write data) are passed in from the subsequent blocks down the pipeline within the system. The sign extension of the immediate field is used to define the addition or subtraction of the ALU block as well as for the branch command. One major responsibility of the Register block was defining the control lines necessary to run the machine.

* *Control:* In order to describe the control of the system I felt it would be best to start by inspecting the opcode of every incoming instruction within the register block. Because we know each opcode is unique in defining the functionality of the instruction we can then trigger signals to pass on down the pipeline to perform the necessary work to carry this out. This is easier said than done. I created four major control lines to pass on down the pipeline: memory read, memory write, write back enable, and execute. Memory read/write were pretty straight forward as they are directly linked to the read/write control of the data memory unit. The signals were toggled quite frequently as many instructions including lui, ori, add, addi, lw, sw all required the use of the memory feature. The Execute signal was a little more difficult for me to define. In the end I feel like the execute signal was replaced by more by me looking at the opcode more frequently than described in the textbook. The write back signal was used to signal at appropriate times when the mux in the write back stage lets certain data pass through. During lw instructions when data is read from the memory this mux passes data from the read data output of the memory and passes it to the register block. On most other instructions the write back line is strobed high to let the ALU result bypass the memory and be passed back to the register block.

Other small scale signals were created in this stage that weren’t described in the book to make life a little easier. The LUI\_Value signal was created and concatenated with zeroes as a forwarding method to directly load the constant in the immediate field without the use of the ALU. Other branching signals were handled here as well.

*ALU Block:* The main work load of the MIPS processor is handled here in the ALU Block. The control and data path of this section was a little more complicated than I was expecting. Due to forwarding and different signals getting passed into the ALU at different times more muxes than shown on the textbook diagram were required. Input1 to the ALU required a feedback line from its output for situations such that and lui and ori instructions happened back to back otherwise the value of the RS format register was passed straight through. The second input to the ALU was a little more cumbersome. The ALU input2 had to handle lui values, sign extended values, as well a determining the correct timing of everything. I used the ALU unit that I constructed from our other project within this design. For the most part the ALU was able to handle all the functions that it was asked to do. I did have to make a few altercations though in order for system to run smoothly. I added a “B\_Value” ALU Control input in order to let ALU input2 pass through in certain situations when work on the data was not necessary. The other ability that my ALU did not possess was the zero flag necessary for branch operations. I made my way around this by letting VHDL do the work for me; I used a compare outside the ALU using a “when” statement to determine if the two incoming values are equal or not in order to set the flag high. One more “instruction mux” was also added in this portion of pipeline. The purpose of this instruction mux was to select which value of “RT” or “RD” to pass on through the pipeline to be eventually used as the write address in the register block. The value to be passed through was determined by looking at the ALU opcode; for the majority the RT register was passed through with the exception of the “addi” field with required the RD register instead.

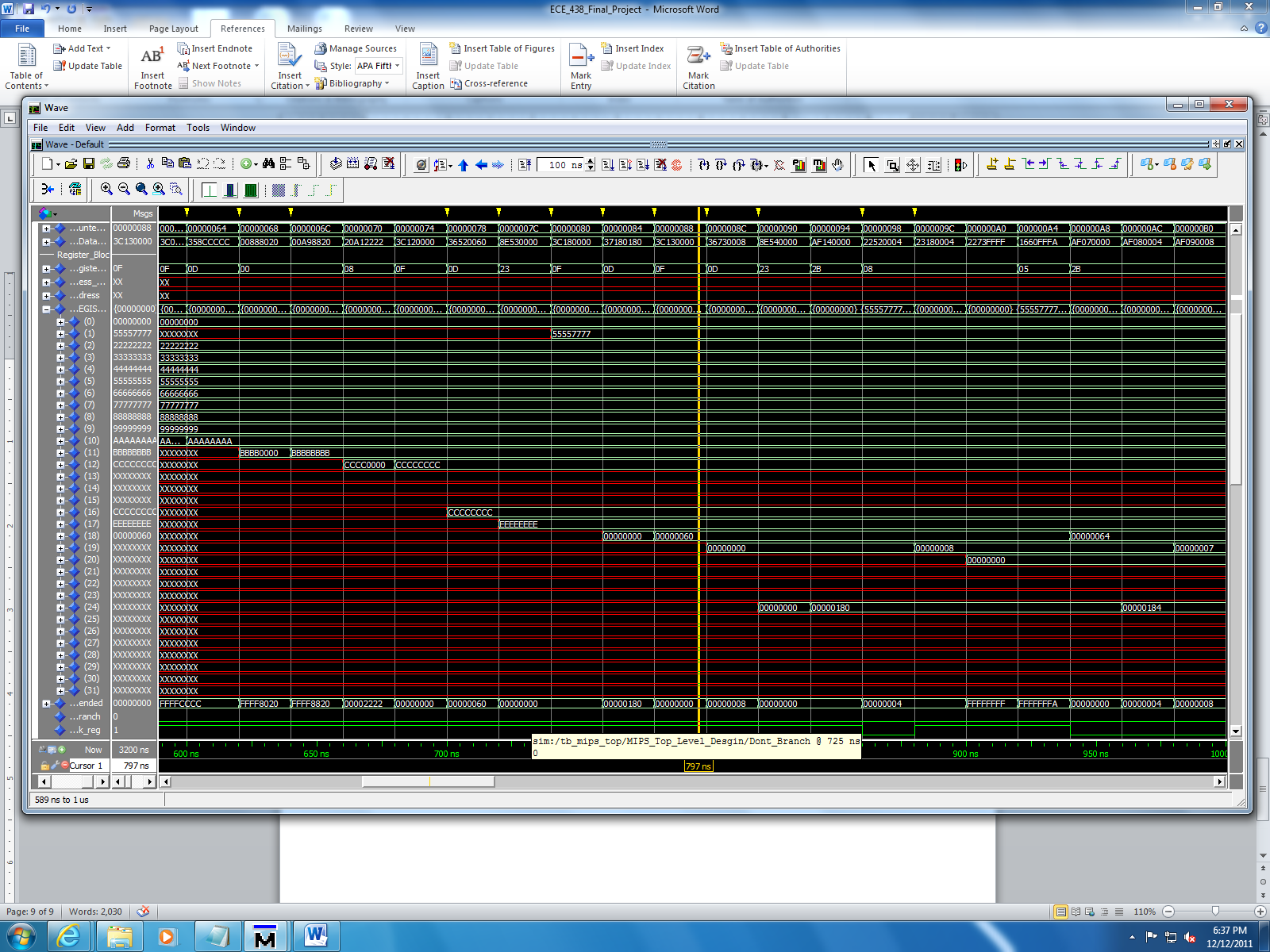
* *ALU Control:* In order to control the functionality of the ALU unit an ALU control bubble was also initialized within the ALU Block. Again the way I handled the operations of knowing when to assert ALU Control lines I looked at the opcode in the ALU Block to know when to place certain combinations into the ALU Control input.
* *Branch Functions:* Here within ALU unit the shift and add functions of the branching were handled. The program counter was added with a shifted sign extended version of the immediate field during branch instructions. The result is then registered and passed back to the instruction memory to be loaded into the program counter to branch back to required location in memory. Although this sounds pretty straight forward the operation of branching was more also more complex than expected. Because of the combined time delay of waiting for the two registers to be compared and moving the appropriate address from the branch adder back to the program counter the whole system as a whole had to be stalled. The system stall has to be implemented in order to wait for the correct program counter address to reach the instruction memory block. I created a stall signal in which monitored the opcode to know when to freeze the program counter to allow the correct address reach the counter. The branch signal was also registered and passed on through the pipeline in order to properly hold memory write read signals low as to stall their functionality.

*Data Memory Block:* The data memory block contained relatively simpler logic compared to that of the other blocks. The read write control lines of the memory were assigned earlier in the register block of the pipeline so there were no worries as to if the needed to be assigned in this block. There was one forwarding situation that had to be taken care of in the case when a load instruction (lw) required the value of a register from the prior instruction. The value actually had to be picked off from the write back stage register in order to be used immediately in the memory stage. Other than this forwarding issue the value from the ALU result register was either passed into address of the memory or passed on the data memory top level register.

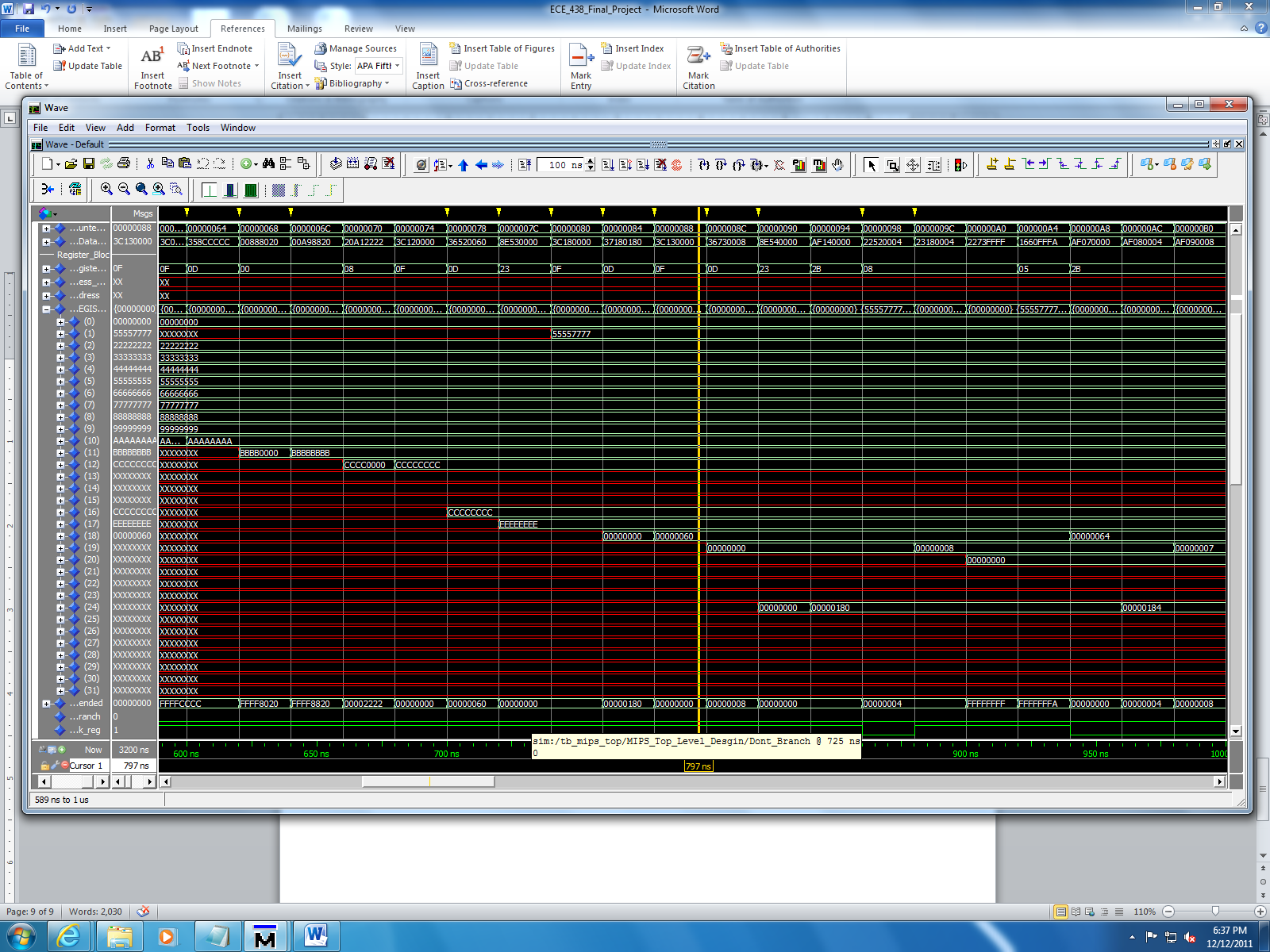
*Write Back:* The sole purpose of the write back stage is to mux the values coming out of the data memory register. This was accomplished by looking at the “write back” signal that was assigned in the register block. For the majority of instructions used in the scope of this project the value from the ALU result which bypassed the data memory input was used to pass directly back to the register write data input. Only during ‘lw’ instructions was the read output value of the data memory used to load into the register block. Along with the write value the write address was also passed from the top level registers back to the register block to write to the appropriate location in the register block.

**Conclusion:**

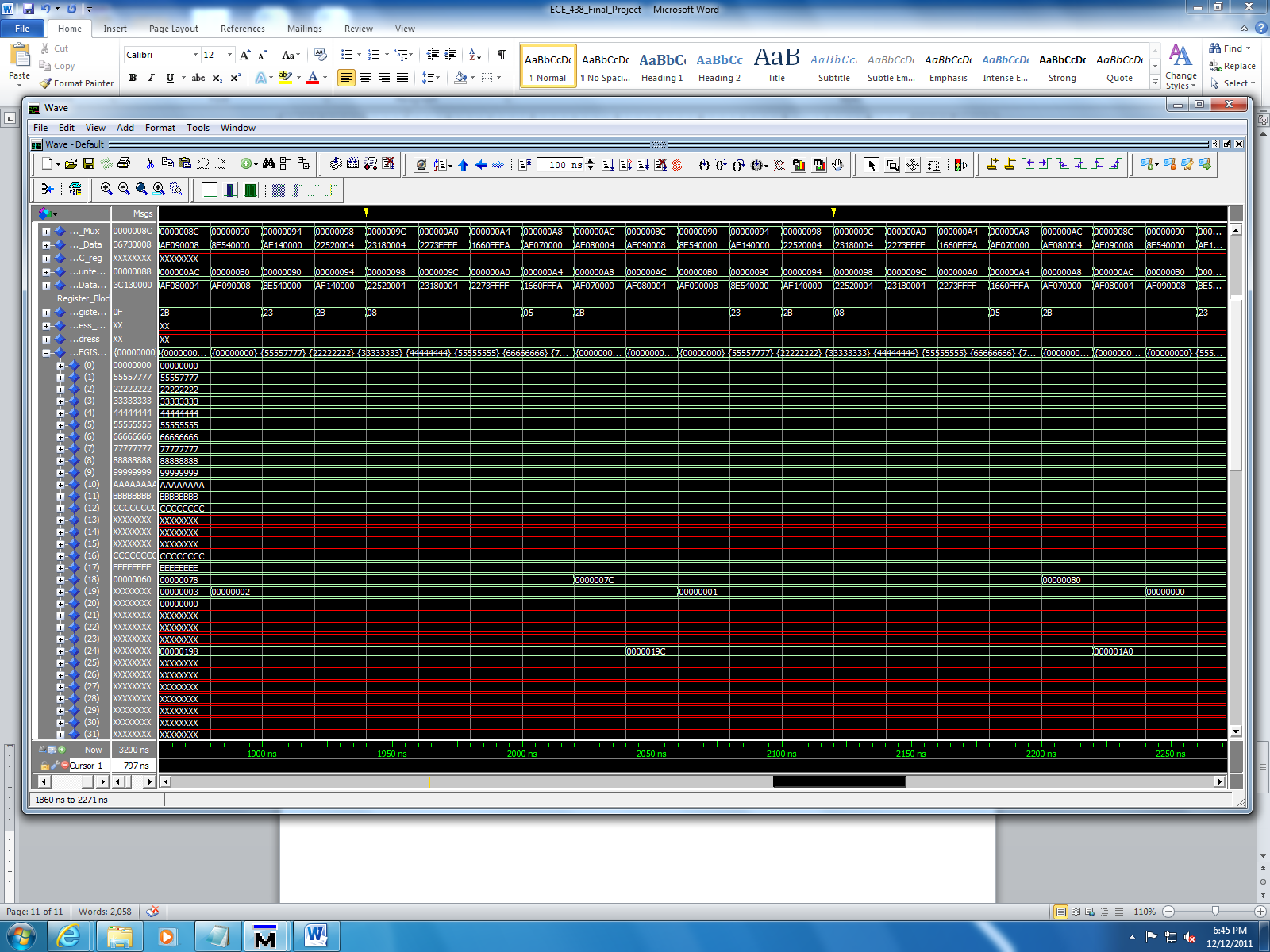
The project gave me good insight onto how complicated the MIPS architecture really is. This is obvious to me especially when realizing the small instruction set we were asked to implement. There is a lot of control that is necessary in order to properly perform all the instructions within the system. One major problem that this version of my design lacks is the ability to correctly mux information in certain instructions as well as disorganization in branching function. I realize that the numbers within my data memory are far from being exact in the sense that they do not correctly match up with the requested assembly language program. Given more time and less pressure from outside classes I feel like I could have more efficiently performed the task at hand. The waveforms below show the register functionality and control signals within the system.



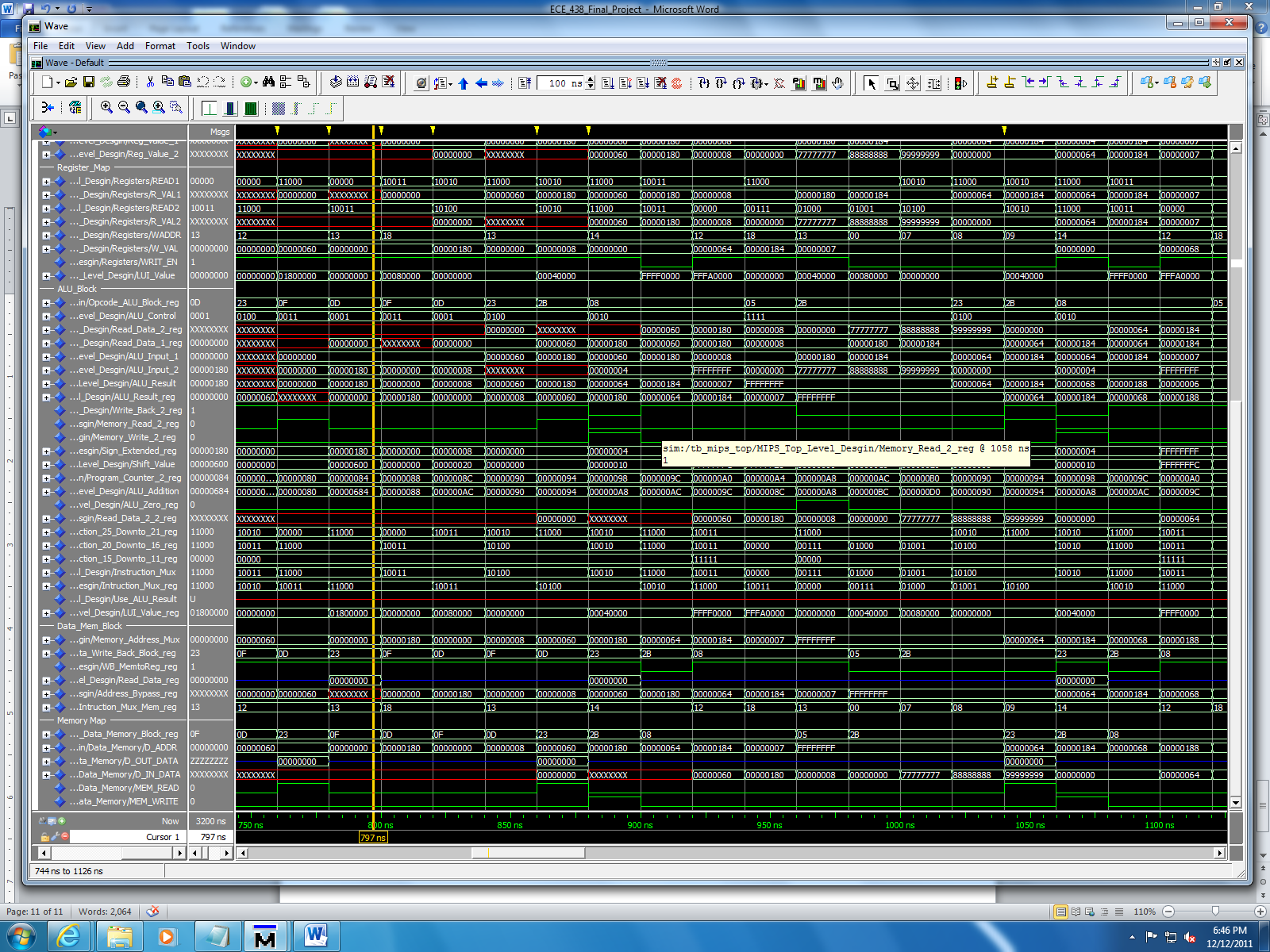
Above is a screen shot showing the functionality of the lui, ori, add, addi, and lw instructions.



This screen shot demonstrates the beginning stages of the “branch” instruction.



The end of the branch instruction.



]\

Above is a just a cool picture of the all the data being processed through the pipeline.