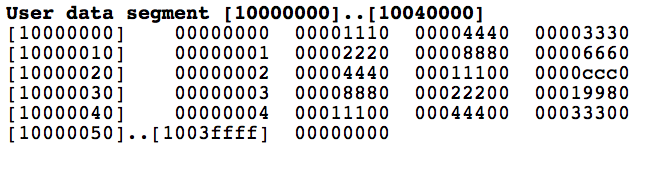
Ariel Harris

June 6, 2017

Lab 6

In this lab, we’ve created the code for a single cycle microprocessor out of codes from previous labs. Those codes include registers, which creates the component for an add/subtract and shift register. We also use the ALU component which does the arithmetic within the processor. The memory component is also included to provide RAM and the S register bank. The main task to create the processor was to put all the components together and create the control and muxs within the processor. The muxs were simply created by following the provided map of the microprocessor and understanding which instructions required certain outcomes from the muxs. For example, the shift commands (sll and srl) are the only two instructions which require the first mux, which decides the source register, to choose the output at 1 allowing rt to be the source register. This mean the control for that mux, RegSrc, would be ‘1’ when the shift instructions were given from instruction RAM.

Instruction RAM is an array of instructions corresponding to an address. So, when the program counter send an address out, the instruction memory takes the bottom four bits and converts them to integers to find the corresponding instruction code. As the program counter increments it should go through each instruction in order, unless there is a branch instruction that sends it to a different address. At the final coded instruction, the code breaks to a -1 instruction and the code stops cycling. At this point the RAM of the microprocessor should look like this: 

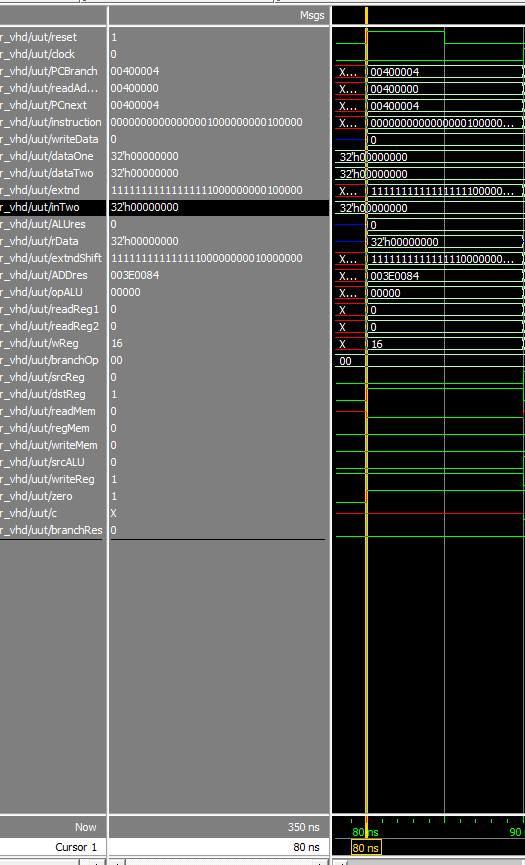
To create the control from the microprocessor it was simply coding multiple truth tables. I first created the ALUOp code since this was the largest and would require the most logic. Since the ALUOp depended on either the opcode or the function depending on the instruction type I concatenated the opcode and function inputs into one logic vector of 12 bits called opFunct.

|  |  |  |
| --- | --- | --- |
| ALUOp | opFunct | Corresponding instruction |
| 00001 | (11 downto 6) = 001000 | Opcode for addi |
| 00101 | (11 downto 6) = 001101 | Opcode for ori |
| 01000 | (11 downto 6) = 100011 | Opcode for lw |
| 01001 | (11 downto 6) = 101011 | Opcode for sw |
| 01010 | (11 downto 6) = 000100 | Opcode for beq |
| 01011 | (11 downto 6) = 000101 | Opcode for bne |
| 00000 | (5 downto 0) = 100000 | Function for add |
| 00010 | (5 downto 0) = 100010 | Function for sub |
| 00011 | (5 downto 0) = 100100 | Function for and |
| 00100 | (5 downto 0) = 100101 | Function for or |
| 00110 | (5 downto 0) = 000000 | Function for sll |
| 00111 | (5 downto 0) = 000010 | Function for srl |

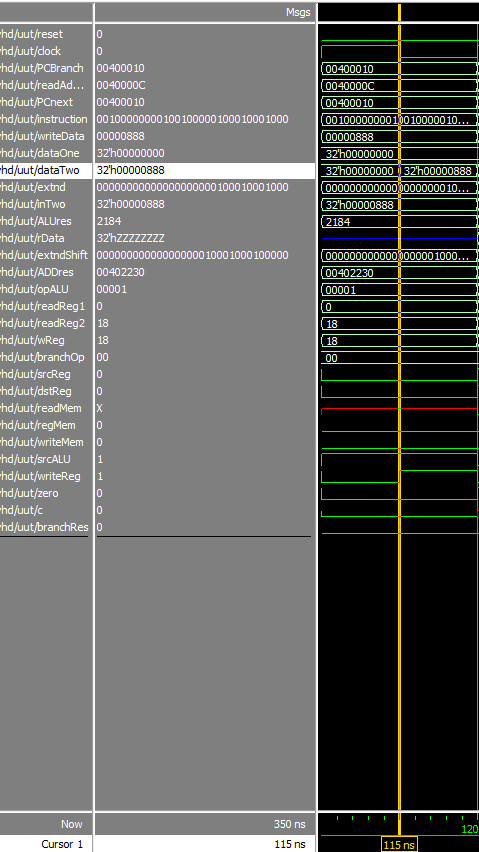
Aside from branch, the rest of the truth tables were simple ‘1’s or ‘0’s for the control outputs. Some of these were only the difference between being and I or R type instruction. For example, RegDst will always be rd for R type instructions, so RegDst <= ‘1’ for any opcode “000000” since the R types all have a 0 opcode. MemRead and MemtoReg are only ‘1’, meaning we can access and read from memory, when the opcode for the lw instruction and MemWrite is ‘1’ only for the sw instruction. RegWrite is ‘1’, allowing data to be written to the register corresponding to RegDst, for most of the instructions aside from sw and branching. ALUSrc, which determines if the ALU gets the value in the second register or the sign extended immediate, required a less straight forward truth table since it was not simply I or R types, or one instruction determining the outcome.

|  |  |
| --- | --- |
| ALUSrc | opFunct |
| ‘1’ | (11 downto 6) = 001000 |
| ‘1’ | (11 downto 6) = 001101 |
| ‘1’ | (11 downto 6) = 100011 |
| ‘1’ | (11 downto 6) = 101011 |
| ‘1’ | (5 downto 0) = 000000 |
| ‘1’ | (5 downto 0) = 000010 |
| ‘0’ | others |

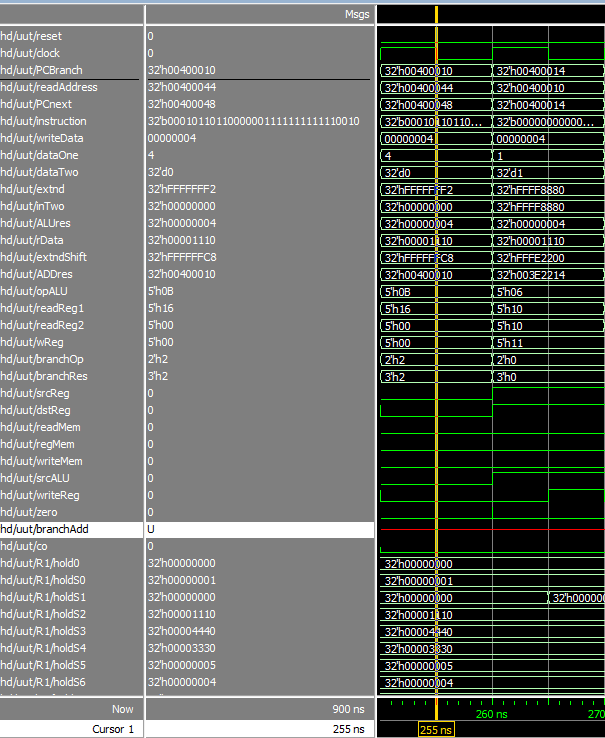
ALUSrc allows the sign extended immediate for the I type instructions which take an immediate value instead of a register value, but also two R type instructions for the shifts. This is because the shamt value that tells the code how far over to shift the value is stored in instruction (10 downto 6) which is inside the immediate value of instruction (15 downto 0). Lastly, the branch truth table didn’t follow an set guidelines as long as we could understand the values we entered. The branch variable was 2 bits wide since there were three possible options: beq, bne, and no branch. So, I chose that branch <= “11” with the opcode for beq and <= “10” with the opcode for bne.



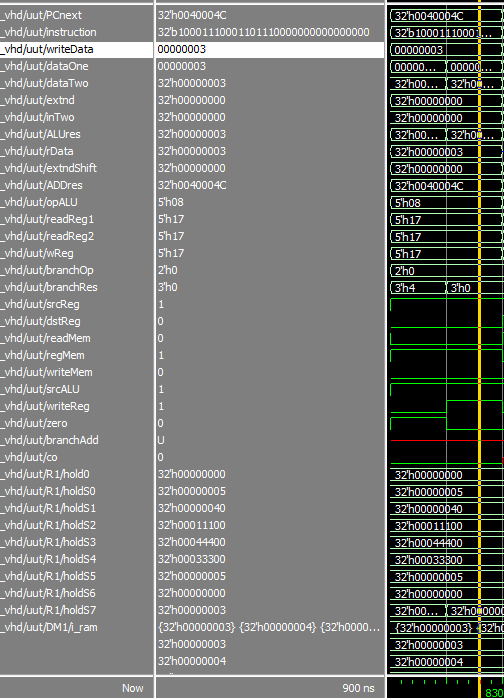
Here we see the reset working and starting the program counter at 0x00400000. This initiates the 0th instruction in i\_ram: 000000\_00000\_00000\_10000\_00000\_100000, this adds the $0 register to itself and stores the result in $s0, which is 16 in wReg on the wave.



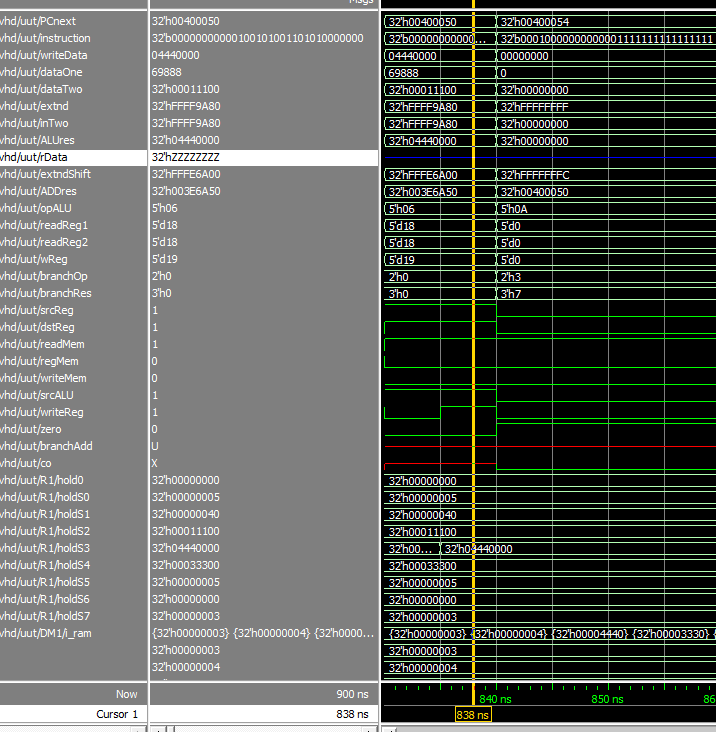
Here the address is 0x0040000C and the wave is at the third instruction. This shows the program counter working correctly as it should increment by 4 each cycle. The instruction 001000\_00000\_10010\_0000100010001000 is adding the immediate value 0x0888 to $0 register and stores it in $s2.



This wave is at the 17th instruction which is the bne, for the first five times the code encounters this instruction it will branch back to a line earlier in the code at the address 0x00400010, which is stored in PCBranch. We see in the next cycle that instead of going to the PCnext address, the microprocessor has broken and gone to the branch address. This part of the wave also shows how previous parts of the code have operated correctly. For example, the 7th instruction shifts the value in $s2 to the left 1 space and stores it back into $s2. In the holdS2 value, which represents the value in each register, there is 0x00001110 which is the shifted result of 0x00000888. The value of $s2 is shifted again, but by 2, and stored in $s3 in the 10th instruction. We can see this value in holdS3. $s1 is set to 1 and $s5 is set to 5 and $s6 is the result of $s5 - $s1, which is 4.

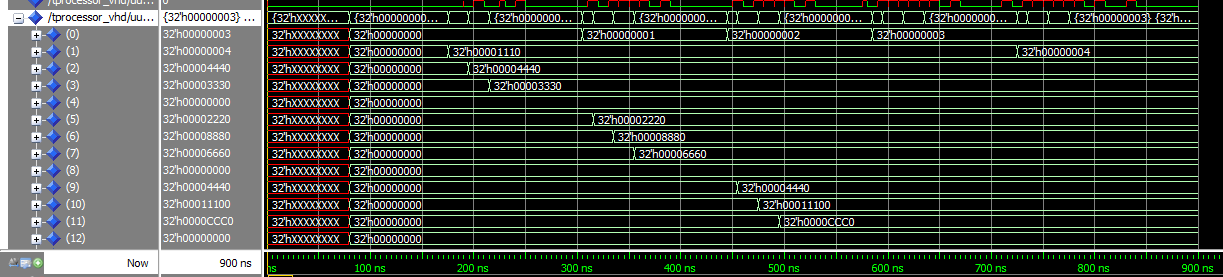


Once the code finally continues past the bne instruction is will load a word from memory. This shows that instruction working properly as it loads the final value stored in 0($s0) and writes it to $s7. The final RAM in my code didn’t function correctly and began re-writing certain values rather than shifting over to look like the memory expected from QTSpim, but the final value that does end up getting stored is the value loaded into $s7.



This final wave shot shows the code breaking at the last beq instruction when it is to branch to a -1 address. This also shows an added instruction for one of the extra credit options. I expanded my shift register component to include all possible values from shamt. This instruction, 000000\_00000\_10010\_10011\_01010\_000000 shifts the final value stored in $s2, which has the decimal value 69888 and hex value of 0x00011100, and shifts it left by 10 storing the final value, 0x04440000 into $s3.

Like I mentioned above my data memory doesn’t act exactly as it should. Instead of continuing to go down the addresses of the memory array, it rewrites existing memory. But, as you go along you can see that each value is the correct one to be stored, just not necessarily in the correct space. Most of the data is going to the correct spot, but the incrementing value in $s0 is being rewritten into the first slot of memory each time rather than into every 4th slot. I’m not sure what would cause the issue since all the other sw instructions are working correctly.



For extra credit, I chose to implement shift using the full shamt value. This meant extending the portion of shamt that was concatenated with the direction control for determining the shift value. Using more of the shamt value simply meant including more ‘0’s to replace the moved values in the input data. I added a test instruction for this before the final beq instruction.