**CPE380 Assignment 2: Multicycle Team Project**

# Implementor’s Notes

Ephraim Morgan, Dalton Grega, and Jacob Dayley

Department of Electrical and Computer Engineering

University of Kentucky, Lexington, KY USA

[ecmo237@uky.edu](mailto:ecmo237@uky.edu), [joda226@uky.edu](mailto:joda226@uky.edu), [dcgr235@uky.edu](mailto:dcgr235@uky.edu)

## ABSTRACT

The goal of this project is to modify a Verilog implementation of a multi-cycle design by adding a MIPS instruction and three non-MIPS instructions as a team.

## 1. GENERAL APPROACH

To complete this assignment, we had to take the shell of a processor provided to us and implement 4 new instructions. Since there were 3 of us and 4 instructions, we had one person do 2 instructions and the other 2 people do 1. Each instruction has one word in memory to be read and executed.

## Memory and Decoding

The decoding logic was quite simple once we figured out how to use the macros provided. For each instruction we were provided with a specific OP code and or told what would be in the FUNCT field. The macros provided allow us to easily set the mask for the decode which was all 1s in either the OP field and FUNCT field or in the case of instructions that use immediate just the OP field. We then assign the provided OP + FUNCT values and set “noop” to the state where our instruction implementation begins.

In our memory we initialize 7 words. These are m[0]-m[5] as well as m[128]. The words at m[0-5] are each instructions while m[128] is a random piece of memory initialized for the purpose of testing our test and set implementation. Each piece of memory that holds an instruction holds its OP code as well as what registers it will be using. It also either holds a FUNCT field or an IMMED field depending on if an immediate is needed for the instruction.

## Or Immediate Implementation

The OR Immediate instruction is and instruction that takes the values from the register assigned rs and bitwise ORs it with the value in the immediate field of the instruction. It then deposits the result into the register assigned rt.

My implementation takes 3 cycles and starts at state 8 of the state machine and goes until state 10. It takes the bits from the immediate field and puts them into the Yin register. Then it puts the value of rs onto the bus and triggers ALUor. At the end of the cycle it calls Zin to lock the result of the ALUor into the Z register.

My test case uses r[1] as an input and has 9 in the immediate field. Register r[1] holds the number 6 because the add instruction put the sum of r[2] and r[3] there. 6 in binary is 0110 and 9 in binary is 1001. When you or this you get 1111 which in hex is 0xf. The output is then stored in the rt field which in this case is assigned to r[4]. You can see the correct result if you run the simulation and go to time 31. 0xf is stored in r[4] and the state goes back to the beginning of the fetch cycle.

## Average Round Up

For the average rounding up function, we were expected to take two registers values (rs & rt) and compute the rounded-up average between the two that will be placed in the rd register. This function was built around the provided equation rd = ((rs^rt)>>1) + (rs&rt) + (rs^rt)&1), which ensured that the computed average returns the rounded-up value. This is possible by taking (rs^rt)&1, where the low bit of rs^rt is taken into consideration when the average computation is done. The result is then placed into the register rd and jumps to the initial state.

To achieve this, we first needed to initialize our ONOP, memory, and registers the function needed. Our ONOP statement required the decimal value 0 to be placed in the OP field and the decimal 1 to be placed in the FUNCT field. This resulted in our ONOP:

`DECODE(`OP(-1)+`FUNCT(-1), `FUNCT(-1), 57).

This function begins in state 57, hence 57 being called in the ONOP statement. Next, we initialized the function in memory by using:  
 m[4] = `OP(0) + `RD(9) + `RS(10) + `RT(11) + `FUNCT(1);

In our memory, registers 9, 10, and 11 are called where 9 is the result register (RD) and 10 and 11 are the computation registers (RS) & (RT).

After everything is prepared, we began implementing the function in state 57. To simplify the process, we focused on computing each of the three terms of the provided rd equation. These three terms were: 1. (rs^rt)>>1, 2. (rs&rt), 3. (rs^rt)&1. Y and MAR were used to store these temporary values. The first term computed was rs^rt, which is repeatedly recalculated throughout the function because of temporary values issues explained later. An example of this calculation can be seen in states 57-59. The next calculation made is rs&rt, which can be seen in states 60-62. In states 63-67, the computed rs^rt is shifted right by a constant 1 and is latched into the temporary y. In states 68-69 (rs^rt) >> 1 + rs&rt is calculated and placed into the temporary MAR. In states 70-74, rs^rt & 1 is calculated and latched into temporary y. In states 75-77, the final calculation is made and moved into the rd register. Finally, the function jumps to the initial state.

Once completed, multiple values as test cases were used to confirm success. The first test case was setting register 10 to a decimal value 7 and register 11 to a decimal value of 4. When computed with a calculator, the average is returned as 5.5, so this function is expected to return the value 6. In the simulation, we can see in times 183-195, which is after all of the states are finished, register 9 has the correct value of 6. For another test case, I used values 12 and 8 to make sure it returns the correct value using two even numbers. After the simulation, the hex value of 10 (a) was returned, ensuring success.

## Test Odd Bit Parity

With an odd bit parity, the goal is to have a “1” in the least significant bit when there is an odd number of “1” bits inside the 32\_bit word. For a case when the number of “1” bits are even, a “0” is shown in the least significant bit. To achieve this, we will split the 32-bit word into two 16-bit half-words. Then we will XOR the two half words to compress the 32-bit word into a single 16-bit half-word. We continue doing this by a factor of two until we reach a single bit which will store our “1” or “0”.

To achieve this, we implemented our own hardware instructions inside of the provided processor outline from “<https://aggregate.org/CPE380/multiF25.html>”. The first part that needs to be implemented for this function is the registers. For this function, we used R[7] and R[8]. For testing purposes, we assigned the decimal value 1,390 in R[7] and 0 in R[8]. Next, we initialized the function in memory by using “m[3] = `OP(32) + `RT(8) + `RS(7);”. After this, we can assign the ONOP with “`DECODE(`OP(-1), `OP(32), 17)”.

After the base outline for our function has been made, we can write the instructions starting at state 17. To start, we need to shift our 32-bit word 16-bits to the right. By using “17: begin `CONST(16) `Yin `NEXT end”, we can assign the number of bit shifts needed inside of the ALU. By using “18: begin `SELrs `REGout `ALUsrl `ALUZin `NEXT end”, we are taking the 32-bit word from register rs(R[7]), shifting it 16-bits to the right and storing it into Z. To save this value stored into Z onto rt(R[8]), we use “19: begin `ALUZout `SELrt `REGin `NEXT end”. To use our XOR function, we now need to store the value in rt into Y using, “20: begin `SELrt `REGout `Yin `NEXT end”. With our new register to compare to, we can implement “21: begin `SELrs `REGout `ALUxor `ALUZin `NEXT end”. Now that we have finished the XOR process, we will store the output into rs again with “22: begin `ALUZout `SELrs `REGin `NEXT end”.

The rest of the XOR functions follow the same structure but with half the shifts of the previous. This means that each implementation of the XOR, we will step the steps down by a factor of 2. This makes each blocks shift 16, 8, 4, 2, and 1. After the last XOR is implemented, we don’t implement “22: begin `ALUZout `SELrs `REGin `NEXT end”. In this instance we need to and the XOR output with the decimal value “1”. This will get rid of all the junk bits and replace them with “0”. To do this we will load Z into Y with “46: begin `ALUZout `Yin `NEXT end” and implementing “47: begin `CONST(1) `ALUand `ALUZin `NEXT end”. Now that we have finished the parity, we can store the AND output into rt with “48: begin `ALUZout `SELrt `REGin `JUMP(0) end”. With our current implementation of these systems, rt(R[8]) will have the “1” or “0”, in this case “1”, at: TIME 133 and STATE 0.

## Atomic Test and Set

The Atomic Test and Set instruction reads a word of memory into the register assigned to rt. Them memory location is determine by adding the value stored in the register assigned rs with the value stored in the immediate field. It then writes a 1 to the memory location found after it has been read.

To implement this I first took the values from rs and the immediate field and added them so that the result was stored into the Z register. From there I put the found address into the MAR register using MARin and used the macro MEMread then untilMFC. This stores the data at the address in MAR into the MDR register. From there I store the data in MDR into the register assigned rt and put a CONST(1) into MDR. My last cycle writes that 1 to memory then returns the the beginning of the fetch cycle.

The instruction that holds this instruction for testing is m[2] and it holds the OP code provided 34 as well as the identification for registers rt and rs as well as a 16-bit immediate field. I set rt to be r[6] and rs to be r[5]. My immediate field holds 256 while r[5] also holds 256. The instruction adds these to get 512 then shifts right by 2 when calling MEMread. This means that it reads from m[128] then writes the value 1 back into m[128]. I set m[128] to hold 0x000abcdf and wired it through the memory module so that it can be read in place of r[31] in the debugger. This allows the user to see what is in m[128] at all times and when it changes. The instruction begins at state 11 and goes until state 16. At time 55 in the simulation you can see m[128] being read to r[6] and at time 56 you can see where CONST[1] gets written back to m[128].

## 2. ISSUES

Some of the issues experienced with the odd parity were from having specific states out of order. This caused the register values to change to unrelated values that in turn broke the whole operation. We tried many things to fix these issues and in the end added an AND operation after each shift. Later we came back and removed some of these unnecessary ALUands which is why the average up starts at state 57 not state 49.

Another issue we ran into was that the srl macro was slightly different than the one given in the reference document. The Yin and Bus were reversed. Since I used the reference when designing the odd parity I changed the srl macro provided in the Verilog to match the one given in the reference document.

While working on the average up function, we ran into the issue of using MDR as a temporary register to store values in. In our original implementation using MAR and MDR, the output of multiple test cases resulted in garbage values. In this original implementation, MDR was used to store rs^rt, so it could be called multiple times for future computations, rather than having to recalculate it. However, due to this issue with MDR, our working implementation computes rs^rt multiple times, resulting in 5-6 more states than before.

## 3. REFERENCES

[1] The reference simple multicycle implementation system provided can be found at <https://aggregate.org/CPE380/multiF25.html>

[2] The Simple Processor Architecture reference used can be found at <https://aggregate.org/EE380/refsp.html>

[3] The reference material used for understanding the processor design can be found at <https://aggregate.org/CPE380/slidesF25simple.pdf>