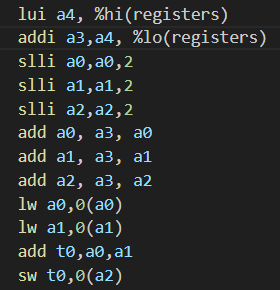
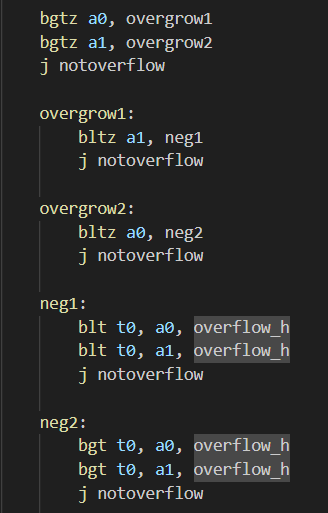
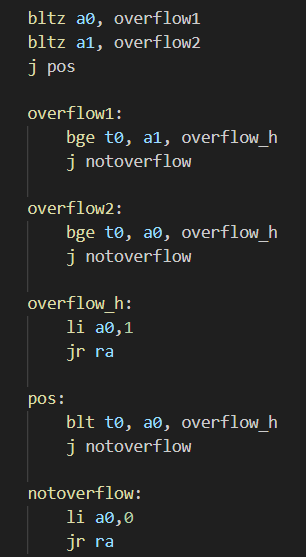
CSC 3050 Assignment 1 Report

Keven Pratama Hendrata 122040031

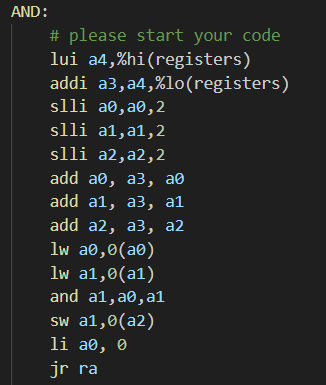
1. ALU

* AND and SUB function

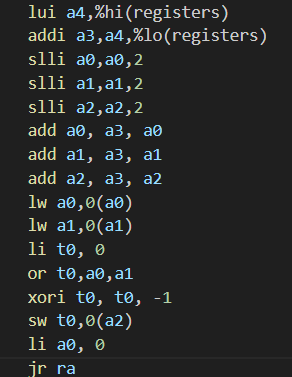
As stated in the Assignment PDF, for these 2 functions, the first and the second registers store the value of the operands, while the third register stores the result. Before starting to implement the function, firstly, we need to load the base address, scale the value of the base address so that 3 integers can be stored, and load the value of the first and second registers’ value so that it can be used to perform some operations. This is done by the first 10 lines of codes presented below. After all of that is done, then we can use the add or sub function provided by RISCV to perform the operation. The code on the left is the implementation for the ADD function. Here I use register t0 to temporarily store the value of the operation before storing the value of t0 to a2. This is done to preserve the memory address of a2. For the SUB function, we must change the add t0, a0, a1 to sub t0, a0, a1.

To ensure the function is robust, overflow within the program must be checked. If there are no overflow, the program will return the value of 0 to register a0, while if there is overflow, it will return the value of 1 to register a1. For the add function, overflow can only happen when the addition is between 2 positives or 2 negatives. Hence, the code on the right will check whether the values a0 and a1 are both positive or negative. This is done by checking the sign of the output. 2 positives will result in a positive value, while both negatives will result in a negative value. Other than that, it means an overflow occurred. The same logic also applies with the SUB function. However, the overflow occurs when the operand is positive and negative or negative and positive. The code on the left is for the ADD function, while the code on the right is for the SUB function.

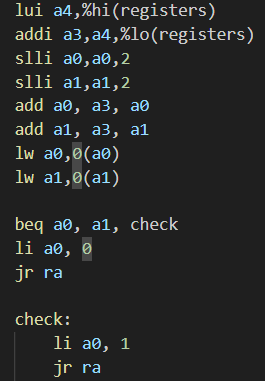
* AND, OR, XOR, SLT

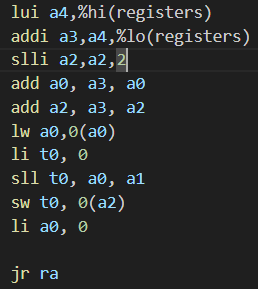
Compared to the previous ADD and SUB functions, these 4 functions are much simpler. Since these 4 functions do not need to handle any overflow and changes in the a0 register, the implementation is simply the loading of the memory address, allocation of 3 integers, and the load of the operands, which is the same as the initial operations in the ADD and SUB functions. The result will be stored at the memory address of a2, while the value of a0 will be 0 at the end of the function. The code on the left is for the AND function. For the other 3 functions, we just need to change the add a1, a0, and a1 functions into their respective functions. This function is the same as the code provided in the assignment PDF.

* NOR

Since there is no nor function within the RISCV architecture, there needs to be some adjustment to be made to complete this function. Similar to previous functions, the first step is to load the memory address and operands. Since the NOR function is the negation of the OR function, we first need to apply the or function to the 2 operands, and then we need to apply a negation function. To do negation, we can utilize the xor default function. We can apply the xor function to the result and -1. Since the xor function will return 1 if the value is different, and 0 if the value is the same, and -1 in binary is consecutive 1s, if the result has 1 it will be turned to 0, and if the value is 0 it will be turned to 1, effectively negating the result. The result of the xor operation will then be stored in a2 register and the value of a0 will be set to 0.

* BEQ and BNE

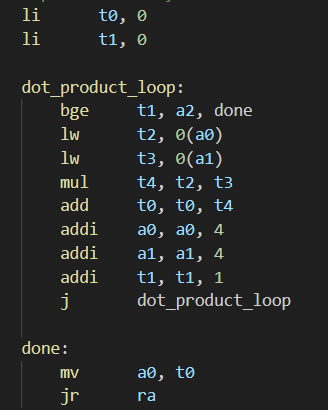
The beginning of this code is the same as previous codes. However, we do not need to reserve some space for a2 since there is no return value. Instead, we only need to reserve some addresses for a0 and a1. The function will compare the values of a0 and a1. If the function returns true, then the value of a0 will be turned to 1, and if it's false, the value of a0 will be turned to 0. This operation will return the value of a0. This can be done by creating a new branch using the default beq or bne function. The following code is for the BEQ function. For the BNE function, we must change the beq a0, a1 check to bne a0, a1, check1. The function is different since there cannot be a function with the same name.

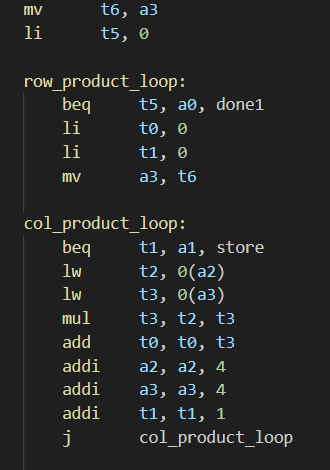
* SLL, SRL, SRA

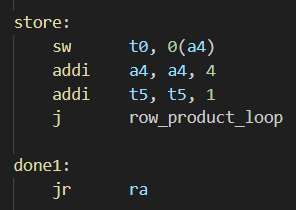
Similar to BEQ and BNE, we only need to reserve memory addresses for some of the registers. In this case, we do not need to reserve some memory for register a1, since the only memory required is for the number we want to shift which is a0, and the register to store the result, which is a2. The register a1 will be use to store the value of bit we want to shift. Other than that, the implementation for this function is similar to the previous functions. The result of the operation will be stored in t0 before being moved to a2. The following code is for the SLL function. For the remaining functions, we only need to alter the function sll t0, a0, a1 to their respective requirements.

1. DOT\_PROD

* vector\_dot\_product\_vector

For the first function, several registers has been set to contain the information about the vectors, and where the result should be stored. According to the assignment PDF, the register a0 is reserved for storing the first vector, register a1 is for the second vector, a3 is for the size of the vector, and finally a4 is reserved to store the result of the operations. To create this function, firstly we need to initialize 2 registers, t0 and t1. T0 is used to store the value of the dot product, while t1 is used as a counter to store the value of the row the operation is currently in. Aftrer the 2 registers have been initialized, we will then create a function that will loop itself indefinitely until some criteria has been fulfilled. If the value of t1 is equal to the value of a3, then the function will jump to another function where the value of t0 will be stored at a4 and return the value. Otherwise if the value of t1 is not equal to a3, we first load the value of integer stored within a0 and a1 to register t2 and t3. The value of t3 and t3 will then be multiplied and added to t0. The address of a0 and a1 will then be incremented by 4, indicating the bit size required to store an integer. Finally, at the end of the function, t1 will be incremented by 1, and the function will call itself. By calling this function, the value of t1 will keep adding until the value reaches a2 and the function will end. If t1 is still less than a2, then a0 and a1 will keep incrementing by 4 and the value will be stored into t2 and t3 to be processed and accumulated in t0. The following code is my implementation of this program.

* matrix\_dot\_vector:

For this task, I used a similar approach to the dot product with several additional functions. The designated registers for storing data in this function is different compared with the previous function. A0 is used to store the number of rows, a1 is used to store the number of columns, a2 for the beginning of the matrix, a3 for the beginning of the vector, and a4 for the final vector result. To start this function, firstly we need to store the memory address of a3 so that we can access it later when we change row. I store the memory address of a3 at t6. Additionally, we also need several counters to keep track of our calculations. T0 is used to store the value of each row operation, t1 is used to keep track of the columns that have been calculated, and t5 is used to keep track the value of row that has been calculated. Firstly, I used the same method as the previous function to calculate the first element of the result array. After the calculation of dot product has been done, a new function called store will store the newly calculated element into an increment of a4. For each element, the memory address of a4 will be incremented by 4 to accommodate the size of 1 integer. Then, in this function the value of t5 will also be incremented by 1. Finally, this function will call another function to handle the amount of vector dot operation. The function will increment the value of t5 by 1, reset t0 and t1 to 0, indicating the start of a new sum and column. If the value of t5 equals a0, then the function will return the value. Additionally, the previously stored address of the original a3 stored in t6 will be returned to a3 so that the multiplication and addition will begin back at the beginning of the vector. This process is repeated until the whole matrix in a2 in finished. The following code is my implementation of this program.