Computer Architecture and Assembly

INTRODUCTION:

For my coursework on computer architecture and assembly I had to create an assembly code for a simple calculator for multiplication. I have decided to chose the calculator task in which I have to develop assembly code for. After some deliberation I believed that from the format in which assembly code is delivered in it quickly became the most attractive to get started on.

For the task of creating a calculator I initially had chosen the CPU X with the most registers. This was because through the process of looking through the instruction sets, thinking about what I will need for the calculator, I realised there was no multiplication instruction in which you can just load the contents of 2 registers, multiply them, and store the answer in another. Therefore, I required more registers for taking on more arguments and for the method of developing a loop. This is because for multiplication sum you need to have a multiplier and a multiplicand. This is because you have one number to add to itself and one number to say how many times to do so.

However, after beginning to work on the calculator there became a sudden necessity to create more instructions and replace specific instructions that we unused. For example, there was the removal of the 13th instruction "output value stored in $X" and it was replaced by the more appropriate and utilized instruction to the task – Mov $x <value>. Furthermore, in the belief of making the task easier to work on there was the modification of the aesthetics of the registers as shown in the table below.

Register Labels		
Return Registers	$2-$3	$r1-$r2
Argument Registers	$4-$7	$a3, $a4, $a5, $a6
Index/Pointer Registers	$8-$11	$ip8, $ip9, $ip10, $ip11

Instructions	
Instruction 13	Mov $x <value>
Instruction 14	La <name>
Instruction 15	Je <name> $X > 0

Function Order	Functions	
1	MSG1	Message "Multiplier $r1"
2	MSG2	Message "Multiplicand $r2"
3	Loop	Reoccur addition and subtraction
4	Loop	Reoccur addition and subtraction
5	Done	When Boolean is true (multiplication is achieved)
6	Done	When Boolean is true (multiplication is achieved)
7	MSG3	Message "Answer MSG1 ASCII 42 MSG2 ASCII 61 $a4"
8	MSG3	Message "Answer MSG1 ASCII 42 MSG2 ASCII 61 $a4"
9	End	End program

FIGURE 1.

```
Start Program

Input AC                (Input keypad into AC)
Load $r1                (Load contents of AC into $r1)
Input AC                (Input keypad into AC)

Load $r2                (Load contents of AC into $r2)

MSG1    db              "Multiplier $r1"
MSG2    db              "Multiplicand $r2"

Mov $a3 <9>             9 in $a3
Cmp $a3 $r1 $a3         (if true) 1 in $a3
Je $a3 <end>            if 1 in $a3 end program
Cmp $a3 $r2 $a3         (if true) 1 in $a3
Je $a3 <end>            if 1 in $a3 end program

Mov $a3 <0>             (Move value of 0 into $a3)
Mov $a4 <0>             (Move value of 0 into $a4)
Mov $a6 <0>             (Move value of 0 into $a6)
Mov $a5 <1>             (Move value of 1 into $a5)
Subt $a5 $a3 $a3        (Subtract $a5 from $a3 and store in $a3)

La $ip8 <loop>          (Load Address $ip8 <loop>)
Add $r2 $a4 $a4         (Add $r2 to $a4 and store in $a4)
Add $r1 $r1 $a3         (Add $a3 to $r1 and store in $r1)

Cmp $r1 $a6 $a6         (Compare $r1 to $a6 & store 1 in $a6 if true)
Je $a6 <done>           (Jump to function <done> if $a6 is true (1))
Je $ip8 <loop> $r1>0)   (Jump back to <loop>)
La $ip9 <done>          (Load Address $ip1 <done>)

MSG3 db                 "Answer MSG1 ASCII 42 MSG2 ASCII 61"
Lea <MSG3> $AC          (Load Effective Address <MSG3> to $AC)
Output                  (Output AC to display)

La $ip10 <end>          End Program
```

Simplistic example of binary being successfully made:

Here is the main logic of the assembly with the inputs being 5 and 5 into the return registers from the keypad.

```
La $ip2 <loop>              (Load Address $ip2 <loop>)
Add $r2 $a4 $a4             (Add $r2 to $a4 and store in $a4)
5 in $a4 / 10 in $a4 / 15 in $a4 / 20 in $a4 / 25 in $a4
Add $r1 $r1 $a3             (Add $a3 to $r1 and store in $r1)
4 in $r1 / 3 in $r1 / 2 in $r1 / 1 in $r1 / 0 in $r1
Cmp $r1 $a6 $a6             (Compare $r1 to $a6 & store 1 in $a6 if true)
Je $a6 <done>               (Jump to function <done> if $a6 is true (1))
false / false / false / false / true (done)
Je $ip2 <loop> $r1>0)       (Jump back to <loop>)
loop / loop / loop / loop
La $ip1 <done>              (Load Address $ip1 <done>)

MSG3 db “Answer MSG1 (ASCII 42) MSG2 (ASCII 61)”
Lea <MSG3> $AC
Load $a4                    (Load contents of $a4 into AC)
Output AC                   (Output a number from AC to display)
```

Display: Answer 5 * 5 = 25

```
La $ip3 <end>               End Program
```

Breakdown of assembly written to perform task:

For the creation of the assembly the recommended process of thinking up some initial pseudocode was in order. This greatly to helped to come up with the assembly that could come close to achieving demands of the task at hand. From that, I learned that I was going to have to go without a specific multiplication instruction from which to carry out the task. I later realised it would take a loop to run a calculation enough times until the conditions were met, the multiplication was achieved and the answer was outputted to the display.

I began with loading the two individual inputs from the keypad from the AC into CPU X’s two return registers. Then I would create two messages to be used later freezing the initial inputs from the keypad to later be displayed in the output ahead of time. After that I created a check to see if the inputs were 9 or higher. If either was, it would jump to the end program function.

Then the setup of the calculation follows where it relied on registers 3, 4 and 6 to have a value of 0 and 5 to have a value 1. The reason we subtract register 5 away from 3, to get minus 1, is to set up a decrement for the multiplier. Whilst that happens register 4 collects the additions and will harbour the final answer. Furthermore, after each time the multiplicand is added to itself the multiplier is decreased by 1.

We then achieve, after several loops, the multiplication of the two values which where input into the key pad. This is because after the multiplier is decremented down to 0 it is compared to the register 6 which is already zero and causes a jump the done function. Then register 4 be ready to store the answer into AC to be outputted to the display, along with MSG3 being constructed with the results from MSG1 and MSG2.

Explain clearly why checking the compiled binary is useful:
Checking compiled binary is useful as it is a method of data validation and permits the program to continue on with the correct information. From the *Methodology for data validation 1.0* we learn "Data validation focuses on the quality dimensions related to the 'structure of the data', that are accuracy, comparability, coherence" (Foundation, 2016). This agrees with the fact that for the continuation of each step, each input needs to be checked. An internationally agreed upon definition of data validation is "[an] activity aimed at verifying whether the value of a data item comes from the given (finite or infinite) set of acceptable values" (COMMISSION, 2000). This helped formulate my decision to check the data as soon as it is inputted and if incorrect to end the program. Furthermore, for the world of software and hardware development, it is important to have common standards and covenants on is legally required from the technology industry and also to ensure security of internationally supplied goods and services.

Perform and write about any checks you performed on your binary to make sure it was created properly:
From the scenario we told that the input numbers must not be higher than the integer 9. I presumed that as the input device, a keypad, only has numerical integers for button numbered 1 to 9. Moreover, to check if the inputs met the requirement or not, I started with moving 9 into register 3. Then comparing the input registers against register 3 to see if they were 9. This affected a Boolean's result. If it was 9 it would add a 1 into register 3 and then based of register 3's value it would jump to end program function and the process would have to be restarted. The limitation of this process would be that there is no error message so it is a human they would no be able to see that their input of 9 was not permitted as it is not stated in the beginning and does not tell them that the inputs are not 9 even though there is a keypad presumably containing the number 9 as it is not allowed. The assembly code for the check that is performed can be seen in figure 2 below.

Figure 2.

```
Mov $a3 <9>              9 in $a3
Cmp $a3 $r1 $a3          (if true) 1 in $a3
Je $a3 <end>             if 1 in $a3 end program
Cmp $a3 $r2 $a3          (if true) 1 in $a3
Je $a3 <end>             if 1 in $a3 end program
```

Conclusion:

Through further deliberation of the choice of hardware to work with, after completing the work, it confirmed my choice and became quite necessary to use the more expensive CPU X dude to it's larger number of registers than the other CPUs. Toward the end of this task I believe that with the hardware and instruction set limitations I was able to create a convincing, simple calculator. I was content with the creation of the logic in which the multiplication is carried out. What I learned from the task is the logic behind how software interacts with the hardware and the importance of maintaining valid data. Unsecure data validation is a major factor in vulnerabilities, and therefore exploits, in software and hardware.

Bibliography

COMMISSION, U. S. (2000). Data Validation. In UN, *GLOSSARY OF TERMS ON STATISTICAL DATA EDITING.* UN Geneva.

Foundation, E. V. (2016). In E. V. Foundation. Essnet Validat Foundation.