EECS 2021 Lab 2 Report

By : Rayhaan Yaser Mohammed
Student ID: 219739887

Introduction

This Lab dives into learning more about constant when writing the assembly language programs. We use more stuff like EQU, DD, and operators to perform calculations. We also learnt more about the memory instructions and assembly language commands to instruct and store values and results.

B1: Assembly operations and Commands(>>, &, EQU, DD)

⇒ **b1a05.asm**

```
b20 : EQU      6146 >> 12
b12 : EQU      6146 & 0xfff
```

This code first extracts the most significant 20 bits of value from 6146 by the shift right(>>) command and is then stored and labelled as b20 by using the EQU assembler command. Similarly the next part of the code stores the least significant 12 bits by using the and(&) operation and then is labelled for reference as b12.

⇒ **b1b05.asm**

```
lui     x6, 1
addi    x6, x6, 0x802
addi    x7, x0, 3
add     x5, x6, x7
```

This code is an example code for calculating 6146 + 3. But the output for this code seen in register x5 is 2050 which is different from the value 6146 this is due to the the assembly architecture interpreting it as a signed integer and thus the input 0x802 is treated as -2046 thus obtaining the value 2050 in register x6 which is 4096-2046 = 2050.

⇒ **b1c05.asm**

```
lui  x6, 2
addi x6, x6, 0x802
addi x7, x0, 3
add x5, x6 ,x7
```

This code solves the error of the previous code by loading value 2 into the upper 20 bits of the register which solves the issue and outputs the expected result. This code then functions as normal and uses the least 12 bits to calculate the expected result 6149 as the answer which is to be stored in register x5.

⇒ **b1d05.asm**

```
b20: EQU 6146 >> 12
b12: EQU 6146 & 0xfff
lui  x6, b20 + 1
addi x6, x6, b12
addi x7, x0, 3
add x5, x6 ,x7
```

This code does the same operation as the code before but this directly references values in the source code to perform the assembly operations.

⇒ **b1e05.asm**

```
lui  x6, (6146 >> 12) +1
addi x6, x6, 6146 & 0xfff
addi x5, x6 ,3
```

This code is an even shorter version of the code in b1f05.asm.

⇒**b1f05.asm**

```
C:        EQU   0x1234567811223344
           lui    x6, (c & 0xffffffff) >> 12
          addi    x6, x6, c & 0xfff
          lui     x7, c >> 44
          addi    x7, x7, (c & 0xfff00000000) >> 32
          slli    x7, x7, 32
          or      x5, x6, x7
```

This code helps us avoid manual calculations in the source code. It automatically adds 1 to the argument of the lui instruction based on the sign of the least significant 12 bits of the given constant. When dealing with constants larger than 32 bits, we can split a 64-bit constant into two 32-bit values. Using the lui and addi methods, we load these values into registers and then combine them into a single register.

⇒ **b1g05.asm**

```
c1:     EQU 0x123
c2:     DD 0x123
```

The EQU command sets the label c1 to represent the value 0x123. This means we can use c1 instead of typing 0x123 in instructions like addi, ori, lui, etc. Meanwhile, the DD command links the label c2 to the memory address where 0x123 is stored. So, instead of dealing with the address directly, we can use c2 to refer to it.

B2: Memory Instructions and Assembly language Commands(ld, sd, DM, ORG)

⇒b2a05.asm

```
C:      DD      0x1234567811223344
        ld      x5, c(x0)
```

This code loads the Double value stored by the DD command into the specified x5 register. In the example, the label "c" tells us where the computer stores a certain number during its operation, which is 0 in this case. This location is then used as a quick reference in the ld instruction. To find the actual spot, we just add the starting point (x0, always set to 0) to the immediate value, which is the offset. This makes it really easy to get to values stored at the beginning of the memory, like addresses ranging from 0 to 4095, using only the 12 bits of the offset.

⇒b2b05.asm

```
sd      x0,     0(x0)
```

This code is intrcuted to write a value of 0 in the memory at address 0.

⇒ b2c05.asm

```
sd      x0,     4(x0)
```

This code is instructed to write aan 8-byte word at memory address 4 which is not divisible by 8 which cause an alignment error.

⇒b2d05.asm

```
sd      x0,     8(x0)
```

This code does the same but instead of memory address 4 it stores it at memory address 8 at the 8 byte word boundary.

⇒b2e05.asm

```
c: DM 1
addi x5, x0, 0x123
sd x5, c(x0)
```

This code uses the "DM" command, a novel addition facilitating memory reservation in programming. With a simple syntax, it allows developers to allocate memory space without immediate value assignment, using a user-defined label as a pointer.

⇒ b2f05.asm

```
c: ORG 0x10000000
DD 0x1234567811223344
ld x5, c(x0)
```

The ORG command shifts the memory address during compilation, allowing data storage at higher locations. This code is made to store and retrieve a substantial constant at the memory address 0x10000000. This command facilitates flexibility in memory allocation for efficient programming.

⇒b2g05.asm

```
ORG 0x10000000
c: DD 0x1234567811223344
lui x6, c >> 12
addi x6, x6, c & 0xfff
ld x5, 0(x6)
```

This code uses the Lui and addi commands to load larger variable to avoid boundary errors by which we can load larger bit address like 32 bit and 12 bit.

⇒b2h05.asm

```
a: DD c
ORG 0x1000000000000000
c: DD 0x1234567811223344
ld x6, a(x0)
ld x5, 0(x6)
```

This code loads larger addresses (larger than 12 bits) into registers, using the same code as b1b05.asm. The 32 bit address loaded into x6 then is used to load the actual stored double.

B3: Branches (bge,beq,bne,jal,jalr,bltu,blt,slt,slti)

⇒b3a05.asm

```
src: DD -3
ld x5, src(x0)
bge x5, x0,skip
sub x5, x0, x5; negate
skip: sd x5, dst(x0)
ebreak x0, x0, 0
dst: DM 1
```

Introducing the "bge" instruction, a critical component that compares values in two registers. This instruction facilitates a conditional jump to a different memory address, initiating code execution from that point. Despite the assembly code specifying an absolute address for this jump, it's important to note that the compiler intelligently translates it into a value relative to the address of the "bge" instruction. The subsequent sections of the code adhere to the procedural guidelines outlined in the lab manual, employing commands characterised by syntax elucidated in the preceding lab report.

⇒ b3b05.asm

```
src: DD -1, 5, -3, 7, 0
add x6, x0, x0
loop: ld x5, src(x6)
beq x5, x0, end ;0 marks the end
sd x5, dst(x6)
addi x6, x6, 8
beq x0, x0, loop
end: ebreak x0, x0, 0
dst: DM 1
```

A new instruction called "beq" is introduced. It works like "bge" but only jumps if two registers have the same values. In a simple program, it checks if a number is zero by comparing it to x0. The program stores a list of numbers, copies them one by one using a loop, and stops when it finds a zero, ending the process with an "ebreak." "Beq" is also used to make the program loop again.

⇒ **b3c05.asm**

```
src: DD -1, 5, -3, 7, 0

add x6, x0, x0

loop: ld x5, src(x6)

beq x5, x0, end ;0 marks the end
bge x5, x0, skip
sub x5, x0,x5;negate
skip: sd x5,dst(x6)
addi x6,x6, 8
beq x0, x0, loop
end: ebreak x0,x0,0
dst: DM 1
```

This example is a combination of the previous 2, copying a sequence of integers to new memory addresses, while also converting them to their absolute values. No new instructions are used, and the branch instructions are used in the same way as above.

Conclusion:

This lab was a lot of information at once but the examples and topics explained are necessary and required to more forward to doing harder and more complex assembly instructions and loops. This lab covered a lot of stuff like branches, memory instructions and also some assembly instructions and commands. The lab also showed new topics like jumps and breaks as well