SLXS Language Manual

Vasilis Sikkis
Student ID:1594115
Supervisor: Dr. David Oswald

Submitted in conformity with the requirements
for the degree of MSc Cyber Security
School of Computer Science
University of Birmingham

Contents

List of Abbreviations

OISC	One Instruction Set Computer
subleq	Subtract Less Equal
SLXS	SubLeqXorShr

Chapter 1

Introduction

1.1 SLXS Architecture Definition

The SubLeqXorShr (SLXS) [UNP] architecture was developed as an extension of the Subtract Less Equal (subleq) [Sub16], as a general-purpose processor for cryptographic and security applications. The SLXS is a One Instruction Set Computer (OISC) Turing-complete machine and supports only a single instruction. It has four address memory operands; *slxs(a,b,c,d)*. The definition of the architecture follows:

$$slxs(a, b, c, d)$$

$$D = *b - *a,\ C = D \veebar c,\ T = C >> 1$$

$$If\ MSB(d) = 0 : *b = C\ else\ if\ MSB(d) = 1 : *b = T$$

$$If\ D <= 0 : goto\ d\ else : goto\ PC + 1$$

Instead of storing the subtraction to the address memory b, it has a temporary variable D that stores the subtraction. Then the address memory c is xored with the result that was previously calculated and stored in the C variable. Another variable, the variable T contains the left shift of the variable *C*. If the most significant bit of the forth address memory d is zero then in the memory address b it stores the variable *C*, else it stores the variable *T*. The execution jumps to to the memory address d if the variable D is less or equal than zero else it continues to the next instruction.

1.2 Outline

This document contains the definition of the SLXS language on an assembly like format and the SLXS tool which is used as complete tool to compile and run programs on the SLXS CPU. It also contains the macro assembler that translates the language to memory addresses, an intel hex converter and the SLXS simulator with some additional features.

Chapter 2

SLXS Language Definition

The SLXS language is an assembly-like language that was created as an intermediate language to facilitate communication with the SLXS simulator. In the subsections, the three types of operations are introduced, variables, instructions, and comments. Some of the key features of the language are: It is a case sensitive programming language, with each line representing one operation (variable/ instruction/comment). Additionally, indentation is not used though it is recommended for more readable code. Empty code lines can be utilised and are discarded as comments. The length of each variable is 17 bit long.

2.1 Variables

Variables are the memory addresses that contain the values of the program. The language recognizes decimal and hexadecimal numbers. The decimal system is the default and the hexadecimal numbers must start with the prefix "0x" or "0X".

The variables stored have a length of 17 bits, with the 17th bit used to differentiate between positive and negative values, with wider numbers capped to 17 bits. Examples of positive numbers are: 1, 0, 0x0ffff and examples of negative numbers are: -1, 0X1ffff.

There are four ways to initialize a variable. The first is the *simple*, that initializes one single variable. Secondly, the *lists* that initializes an array of variables in one line and third the *assignments* that initializes the variable with the value of another variable. The final is the *pointer* which stores, as a variable, the address of another variable.

2.1.1 Simple

The regular expression of the simple variable is:

$$([A..Z][a..z]) + ([A..Z][a..z][0..9]) * \{:\}([0..9] + |(\{0x\}|\{0X\})[0..9] * [A..F] * [a..f]*)\{\$\}$$

The language recognizes system variables that start with the underscore symbol and they cannot be defined by the programmer, only by the language itself.

$$\{\_\}([A..Z][a..z]) + ([A..Z][a..z][0..9]) * \{:\}([0..9] + |(\{0x\}|\{0X\})[0..9] * [A..F] * [a..f]*)\{\$\}$$

Examples:

```
Z:0
temp1:0xaB
```

2.1.2 Lists

The regular expression of the list is:
$([A..Z][a..z]) + ([A..Z][a..z][0..9]) * \{:\ [\}([0..9] + |(\{0x\}|\{0X\})[0..9] * [A..F] * [a..f]*)(\{,\}([0..9] + |(\{0x\}|\{0X\})[0..9] * [A..F] * [a..f]*)) * \{]\$\}$
Example:

```
temp: [0,1,2]
```

The list will be stored in the memory with a pointer at the start of the list and then the values will follow. Each list variable can be accessed with the following pattern:

1. The values of the list can be accessed with the name of the list and the "(" +*number*+")". In the previous list the variable *temp(1)* equals with 1.

2. The list address is stored with the name of the variable given by the programmer. In the example the variable *temp* is equal with the address of *temp(0)*.

An example of the previous list stored in the memory is:

```
Address Values
0004:   00005   00000   00001    00002
address temp    temp(0) temp(1)  temp(2)
```

2.1.3 Assignment

The regular expression of the assignment is:

$$([A..Z][a..z]) + ([A..Z][a..z][0..9]) * \{:\}([A..Z][a..z]) + ([A..Z][a..z][0..9]) * \{\$\}$$

If a variable was previously declared it can be used to initialize another variable. In the example that follows the $var1$ holds the value 0. The *var1* can then be used to initialize the *var2* with the same value.

```
var1 : 0
var2 : var1 //The var2  will hold the value 0.
```

2.1.4 Pointers

The regular expression of the pointer is:

$$\{*\}([A..Z][a..z]) + ([A..Z][a..z][0..9]) * \{:\}([A..Z][a..z]) + ([A..Z][a..z][0..9]) * \{\$\}$$

If a variable was previously declared it can be used to store its address as another variable. The *var1* is stored in the address 5 and holds the value 10. By using the star symbol "*", the **var2* will be initialized with the address of *var1* (5).

```
var1 : 10   // stored in address 5
*var2: var1 // var2 = 5
var3 :var1  // var3 = 10
```

2.2 Instructions

Each instruction has four parameters *slxs(a,b,c,d)* in the SLXS language and are separated by "," with the end of the instruction denoted by ";". Each instruction uses the variables that have been previously declared. The parameters a and c can alternatively use decimal numbers or hexadecimal numbers instead of variables. An example is given in the 2.2.1 simple instruction. The regular expressions that follow will minimise the length of the different types of instructions.

$$number = ([0..9] + |(\{0x\}|\{0X\})[0..9] * [A..F] * [a..f]*)$$

$$var = \{\_\} * ([A..Z][a..z]) + ([A..Z][a..z][0..9])*$$

$$label = \{\_\} * ([A..Z][a..z]) + ([A..Z][a..z][0..9])*$$

$$list\_value = ([A..Z][a..z]) + ([A..Z][a..z][0..9]) * \{(\}number\{)\}$$

2.2.1 Simple

The regular expression of a simple instruction is:

$$var|number|list\_value\{,\}var|list\_value\{,\}var|number|list\_value\{;\}$$

A simple instruction has three variables. If the programmer wants the program to continue the regular flow to the next instruction they can leave the fourth parameter blank and the micro-assembler will add the address of the next instruction to the fourth parameter. A code example of a simple command that moves x to y is:

```
Z:0
x:16
y:0
Z,y,x;
```

The previous operator example can be written with undefined variables. The parameter b (in the example this is denoted as the variable y) must be a variable that was declared by the programmer because the result of the instruction will be stored in the memory address of the variable.

```
y:0
0,y,0x10;
```

The current version of the tool can store in the memory a fixed number of undefined variables. It can be enabled by using the $[-u]$ option from the tool that will be introduced in chapter 3. The same example that saves the result of the instruction on a list is:

```
Z:0
x:16
temp: [0,1,2]
Z,temp(0),x;
```

2.2.2 Functions

The regular expression of a function is:

$$label\{:\}var|number|list\_value\{,\}var|list\_value\{,\}var|number|list\_value\{;\}$$

The use of a label at the start of the instruction can be used to create functions and loops. The address of the specific instruction is saved with the label name so it can be accessible. An example on how a function is used is:

```
foo:Z,x,y
```

2.2.3 Jumps

The regular expression of a jump is:

$$var|number|list\_value\{,\}var|list\_value\{,\}var|number|list\_value\{,\}label\{;\}$$

The jump instruction uses all four parameters that were introduced by the SLXS architecture. The fourth parameter is a label that contains the address of the function we want to jump to. The example uses the previous function *foo*.

```
Z,x,y,foo;
```

2.2.4 Functions with Jump

The regular expression of a function with a jump is:

$$label\{:\}var|number\{,\}var\{,\}var|number\{,\}label\{;\}$$

This type of instruction combines the two previous types. An example is:

```
foo:Z,x,y,foo1;
```

2.2.5 Labels as Pointers

A label at the start of the instruction can be used to define the address of a current memory position. This is useful when the code must change dynamically in a loop. In the example that follows the *foo(0)* holds the address of x.

```
foo:Z,x,y;
Z,foo(0),y; //foo(0) holds the address of x.
```

2.2.6 Shifts

For each of the previous types the string "_SH" can be appended as a last parameter to create the shift command. An example of the simple instruction is the following:
Regular Expression: $var\{,\}var\{,\}var\{,\}\{\_SH;\}$

```
z,x,y,_SH;
/*With function and a jump:*/
foo:x,y,z,foo1,_SH;
```

2.3 Comments

The language recognizes two types of comments that are both in-line. The first is the double dash "//" and the language ignores everything until it finds a newline. The second type is the boxed comments that start with "/*" and end with "*/" and the language ignores everything inside them.

1. **Whole line comments**
   Regular Expression:{//}[^] * $
   Example:

   ```
   //This is a comment
   ```

2. **Boxed comments**
   Regular Expression: {/*}[^] * {*/}
   Example:

   ```
   S :/*This is a comment */ 10
   x,y,temp(0)/*first variable inside the list temp*/,_SH;
   ```

2.4 Predefined Variables/Labels/Instructions

The SLXS language has some predefined variables, labels and instructions that are listed below. Additional material can upgrade/update the language in the future. Variables and labels used by the system, start with underscore for consistency.

Variables:
_zero: 0
The *_zero* variable is initialized by the micro-assembler. Additionally, it can be used by the programmer without implicitly declaring it.

Labels:
_main:
The main function of the program. Each program must have one otherwise, it cannot run.
_SH:
The *_SH* appended at the end of the instruction ensures that the assembler knows how to execute the shift command.

Instructions:
_zero,_zero,_zero,_main;

This instruction is the first line of code in every program. It is used as guidance for the simulator to know where the *_main* function is. The assembler always adds it to the start of the code, so the programmer does not have to.
loop:_zero,_zero,_zero,loop;
This instruction indicates the end of the program. It is needed so that the SLXS simulator knows when to stop. It has to be added by the programmer.

The tool has additional variables and labels that are used to implement the macros that are going to be introduced in the section 2.6. The variables that can be used as constants are: *_two=2, _minus=-1, _one=1*. The rest of the variables are not recommended to be used if the programmer is not familiar with the language because it might disrupt the correct execution of the program.

2.5 Operators

To define the basic operators we need the following variables to be declared.

```
/*Variables:*/
  Z:  0
  M:  1FFFF //(-1)
  U:  1
  Label:  01234 // A 16 bit address (the 17 bit is for the shift)
  var:10
  var_r: -10
```

Commands:

```
 CLEAR : var   , var, Z;          // var = 0
 JUMP  : Z     , Z  , Z, Label;  // Goto Label
 MOV   : var   , var, var1;       // var = var1
 LOOP  : Z     , Z  , Z  ,Label; // current address| EOP
 RSHIFT: Z     , var, Z  ,_SH;    // var >> 1
 LSHIFT: var_r, var, Z;           // var = var + var = var << 1
 XOR   : Z     , var, var1;       // var = var  var1
 ADD   : M     , var, Z;          // var = var + 1
 SUB   : U     , var, Z;          // var = var - 1
```

The multiplication operator can be implemented by a sequence of additions and the division by a sequence of subtractions. The most common multiplications and divisions are computed with the number two as a multiplier/divisor and they can be implemented with the use of two shift operators.

The most complex operators are the AND/OR, due to the need of a sequence of instructions to implement them. Based on the formula $x + y = (x \veebar y) + 2(x\&y)$ and $x + y = 2(x|y) - (x \veebar y)$, we can extract the two operators: $x\&y = (x + y + (x \veebar y))/2$ for AND, $x|y = (x + y - (x \veebar y))/2$ for OR. The two operators can be translated to the SLXS language with the code listed below. All the operators are included in appendix C.

```
/*Extra Variables:*/
res = 00000
xor = 00000
y,x
AND: res, res, x; // res = x
y , my , Z; // my = -y
my , res, Z; // res = x + y
xor, xor, x; // xor = x
Z , xor, y; // xor = x  y
xor, res, Z,SH; res = (res - xor ) >> 1

OR : res, res, x; //res = x
y , my , Z; // my = -y
my , res, Z; // res = x + y
xor, xor, x; // xor = x
Z , xor, y; // xor = x  y
xor, mxr, Z; // mxr = -xor
mxr, res, Z,SH; // res = (res + xor) >> 1
```

2.6 Macros

For additional usability of the language, the operators have been imported into the language. The programmer can use these specified macros to the language. The only drawback of using the macros is that they can not be written with labels as functions.

2.6.1 Clear

The clear macro sets the value of the variable to zero. In the example that follows the *val* variable is set to zero after the execution.

```
    _CLR,val;
```

2.6.2 Move

The *move* macro sets the value of the first variable with the value of the second. In the example that follows the *move1* variable has the value of *move2* after the execution.

```
_MOV,move1,move2;
```

2.6.3 Multiplication

The *multiplication* macro calculates the result of the multiplication end stores it to the first variable. In the example that follows the *mul1* variable is equal to the result of the multiplication.

```
_MUL,mul1,mul2;
```

2.6.4 Division/Modulo

The *division/modulo* macro calculates the result of the division end stores it to the first variable. The second variable has the reminder which equals to the modulo operator. In the example that follows the *div1* variable is equal to the result of the division and the *div2* variable is equal to the result of the modulo.

```
_DIV,div1,div2;
```

2.6.5 Addition

The *addition* macro calculates the result of the addition end stores it to the first variable. In the example that follows the *add1* variable is equal to the result of the addition.

```
_ADD,add1,add2;
```

2.6.6 Subtraction

The *subtraction* macro calculates the result of the subtraction end stores it to the first variable. In the example that follows the *sub1* variable is equal to the result of the subtraction.

```
_SUB,sub1,sub2;
```

2.6.7 And

The *and* macro calculates the result of the *and* operator end stores it to the first variable. In the example that follows the *and1* variable is equal to the result of the *and* operator.

```
_AND,and1,and2;
```

2.6.8 Or

The *or* macro calculates the result of the *or* operator end stores it to the first variable. In the example that follows the *or1* variable is equal to the result of the *or* operator.

```
_OR,or1,or2;
```

2.6.9 Xor

The *xor* macro calculates the result of the *xor* operator end stores it to the first variable. In the example that follows the *xor1* variable is equal to the result of the *xor* operator.

```
_XOR,xor1,xor2;
```

2.6.10 Left Shift

The *left shift* macro calculates the result of the *left shift* operator end stores it to the first variable. In the example that follows the *shift* variable is equal to the result of the *left shift* operator.

```
_LSHIFT,shift,ls;
```

2.6.11 Right Shift

The *right shift* macro calculates the result of the *right shift* operator end stores it to the first variable. In the example that follows the *shift* is equal to the result of the *right shift* operator.

```
_RSHIFT,shift,rs;
```

Chapter 3

SLXS Toolchain

Chapter 3 introduce the SLXS toolchain that contains the macro-assembler, the intel-hex converter and the simulator. It explains how each stage functions and examples follows that show the results generated for each stage.

3.1 Introduction to SLXS Toolchain

The SLXS code must go through three stages to provide the end result as it is showed in Figure 3.1. At first, the code is translated into memory addresses. Next, those memory addresses must be converted to an Intel hex file format, one for each of the four parameters of the SLXS language. The final stage is the simulator that takes the four files and outputs the result of the code.

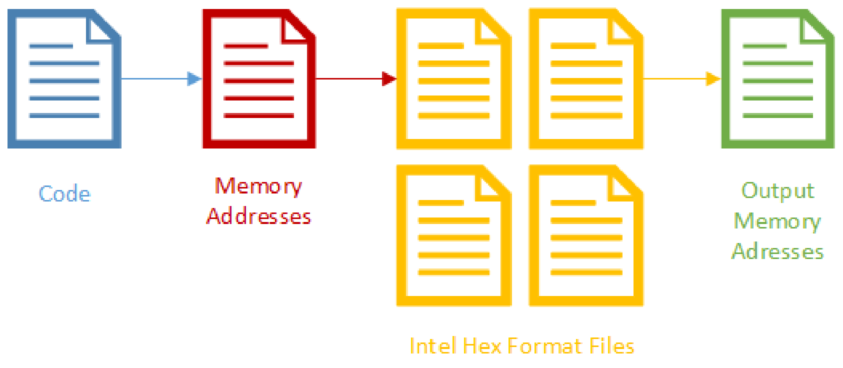


Figure 3.1: SLXS Language Stages

The SLXS toolchain automatizes this process with a set of tools that produce output files for each part of the three stages. The tools can be used separately if the user needs to use only one of them. The five options that the toolchain provides are:

1. $[-i|--input]$ Takes a file with SLXS code as an argument and creates a file with the name *output.txt* stored inside the *output_files* directory, with the memory addresses. It is essentially a translator and macro-assembler that provides to the user feedback if something is going wrong in case he made a mistake while writing the code.

2. $[-f|--intel\_hex]$ It creates four files in the *output_files* directory in hex_intel format that can be used in the simulator.

3. $[-s|--simulator]$ It simulates an SLXS processor. The four hex files created by the $[-f]$ option are used. If the program can not be executed due to memory corruption, it will terminate. The result is stored in the *simulator_output.txt* inside the output file directory.

4. $[-p]$ Prints the variables before and after the simulator runs. All the previous options must be enabled. If the variable does not exist, nothing is displayed.

5. $[-u]$ Allows the programmer to use undefined variables inside the code.

6. $[-h|--help]$ Prints the help menu.

While using the option $[-i|--input]$ the tool outputs on the command line the size in bits of the variables, the size of the instructions and the overall size of the code that will be needed to be stored in the memory.

3.2 Code Example for OR Operator

This example shows how the OR operator is implemented in the SLXS language and how it is compiled and executes. The tool can be used and the result will be displayed in the command line with the command:

```
python slxs.py -i or.txt -f -s -p res
```

That will execute all three stages and print the variable res before the execution and after.

```
/*Variables*/
Z   : 0         // zero value
mxr : 0         //-xor
One : 1
xor : 0
```

```
res : 0
x   : 0x11
y   : 0x10
my  : 0          // -y

/*main function*/
_main:res, res, x;
    y   , my , Z;
    my , res, Z;
    xor, xor, x;
    Z   , xor, y;
    xor, mxr,Z;
    mxr, res,Z,_SH;
    loop,Z,Z,Z,loop;
```

An example of undefined variables on the previous code is and the option $[-u]$ of the tool enabled:

```
    0x10 , my , 0;
    my , res, 0;
    xor, xor, 0x11;
    0   , xor, y;
    xor, mxr,Z;
    mxr, res,Z,SH;
```

With the option $[-i]$ the output file with the memory addresses is created. The first column indicates the memory address, the other four are the values stored in the addresses in the block.

```
0000 00004 00004 00004 00010 | _zero,_zero ,_zero,_main | Jump to main
0004 00000 00000 00000 00001 | 0 , 0 , 0 ,1             | _zero:0 Z:0   mxr:0   One:1
0008 00000 00000 00011 00010 | 0 , 0 , 17,16            | xor:0   res:0 x:17  y:16
000c 00000 00000 00000 00000 | 0 , 0 , 0 ,0             | my:0    empty slots
0010 00009 00009 0000a 00014 | res,res,x;               | main function res=x
0014 0000b 0000c 00005 00018 | y   , my , Z;            | my   = - y
0018 0000c 00009 00005 0001c | my , res, Z;             | res = x + y
001c 00008 00008 0000a 00020 | xor, xor, x;             | xor = x
0020 00005 00008 0000b 00024 | Z   , xor, y;            | xor = x ^ y
0024 00008 00006 00005 00028 | xor, mxr,Z;              | mxr = -xor
0028 00006 00009 00005 1002c | mxr, res,Z,SH;           | res = (res + xor ) >> 1
002c 00004 00004 00004 0002c | loop                     | End of program
```

Intel Hex Format Files for simulator:
Each Intel Hex [INT16] block in the SLXS simulator contains the following data:

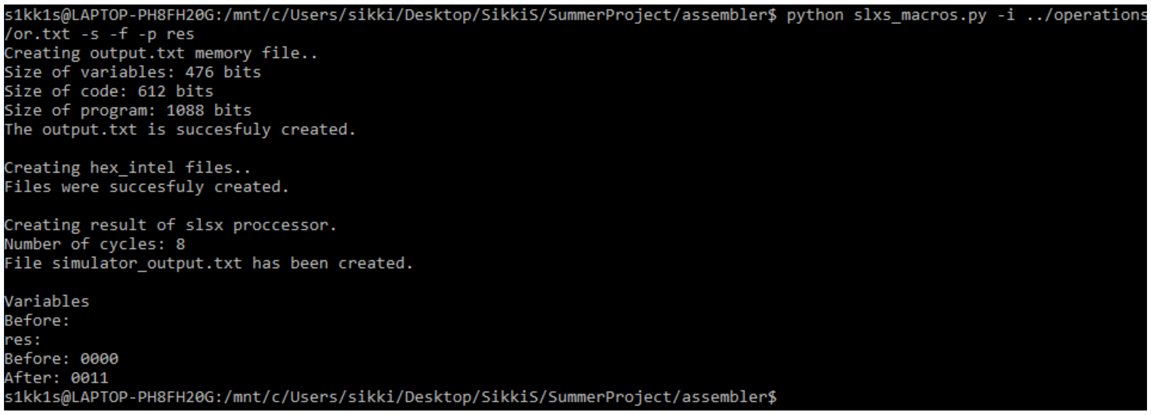
1. **Start code**, one character, an ASCII colon ':'.
2. **Byte count**, two hex digits, indicating the number of bytes (hex digit pairs) in the data field.
3. **Address**, four hex digits, representing the 16-bit beginning memory address offset of the data.
4. **Record type**, two hex digits, 00 to 05,in the implementation the 01 and 03 are the only digits used.
5. **Data**, six hex digits.
6. **Checksum**, two hex digits, a computed value that can be used to verify the record has no errors.

The output from the assembler has to be split in four files of intel hex format. An example that was created with the $[-f]$ option for the OR operator is the following:

```
    mem0.hex             mem1.hex            mem2.hex             mem3.hex
:030000000000004f9 |:030000000000004f9 |:030000000000004f9 |:03000000000010ed
:03000100000000fc |:03000100000000fc |:03000100000000fc |:03000100000001fb
:03000200000000fb |:03000200000000fb |:03000200000011ea |:03000200000010eb
:03000300000000fa |:03000300000000fa |:03000300000000fa |:03000300000000fa
:03000400000009f0 |:03000400000009f0 |:0300040000000aef |:03000400000014e5
:0300050000000bed |:0300050000000cec |:03000500000005f3 |:03000500000018e0
:0300060000000ceb |:03000600000009ee |:03000600000005f2 |:0300060000001cdb
:03000700000008ee |:03000700000008ee |:0300070000000aec |:03000700000020d6
:03000800000005f0 |:03000800000008ed |:0300080000000bea |:03000800000024d1
:03000900000008ec |:03000900000006ee |:03000900000005ef |:03000900000028cc
:03000a00000006ed |:03000a00000009ea |:03000a00000005ee |:03000a0001002cc6
:03000b00000004ee |:03000b00000004ee |:03000b00000004ee |:03000b0000002cc6
:00000001FF        |:00000001FF        |:00000001FF        |:00000001FF
```

The four files can be now imported to the simulator. The option $[-s| - -simulator]$ takes the four intel_hex files and creates the final state of the memory in an output file as shown in the code below. Only the addresses are stored in the output file, the two other columns are for explanation. In the position the *res* variable was stored the value has changed to 0x10 which is the result of *xy*.

```
0000 00004 00004 00004 00010 | _zero,_zero ,_zero,_main | Jump to main
0004 00000 00000 00000 00001 | 0 , 0 , 0 ,1             | _zero:0 Z:0   mxr:0   One:1
0008 00000 00010 00011 00010 | 0 , 16 , 17,16           | xor:0   res:16 x:17  y:16
000c 00000 00000 00000 00000 | 0 , 0 , 0 ,0             | my:0    empty slots
0010 00009 00009 0000a 00014 | res,res,x;               | main function res=x
0014 0000b 0000c 00005 00018 | y  , my , Z;             | my  = - y
0018 0000c 00009 00005 0001c | my , res, Z;             | res = x + y
001c 00008 00008 0000a 00020 | xor, xor, x;             | xor = x
0020 00005 00008 0000b 00024 | Z  , xor, y;             | xor = x ^ y
0024 00008 00006 00005 00028 | xor, mxr,Z;              | mxr = -xor
0028 00006 00009 00005 1002c | mxr, res,Z,SH;           | res = (res + xor ) >> 1
002c 00004 00004 00004 0002c | loop                     | End of program
```



```
s1kk1s@LAPTOP-PH8FH20G:/mnt/c/Users/sikki/Desktop/SikkiS/SummerProject/assembler$ python slxs_macros.py -i ../operations
/or.txt -s -f -p res
Creating output.txt memory file..
Size of variables: 476 bits
Size of code: 612 bits
Size of program: 1088 bits
The output.txt is succesfuly created.

Creating hex_intel files..
Files were succesfuly created.

Creating result of slsx proccessor.
Number of cycles: 8
File simulator_output.txt has been created.

Variables
Before:
res:
Before: 0000
After: 0011
s1kk1s@LAPTOP-PH8FH20G:/mnt/c/Users/sikki/Desktop/SikkiS/SummerProject/assembler$
```

Figure 3.2: OR operator line result

In the Figure 3.2, the example analyzed is executed. In each step of the process messages are printed to indicate in which option the tool currently is. The size of the variables, the code, and the overall size is printed to show to the programmer the minimum requirements the program needs to be stored. While the simulator is running the number of SLXS clock cycles is calculated. Lastly the result of the variable *res* is printed with the value before and after the execution.

3.3 List Example With Pointers

The following code takes two lists xor them and saves the result in the first list. This example shows the use of pointers and how the code can dynamically change.

```
/*Variables*/
/* List */
List :  [0x1,0x2,3,4,5,6,7,8,9,10]
List2 : [0x1,0x2,3,4,5,6,7,8,9,10]

Z   : 0
One : 1
M   : -1
counter :10

list_add : List // The value of the variable List not used.
*list_add_10: List(9) //The address of List(9) not used.

/*Code*/
/*main function*/

_main: Z,List(0),List2(0);     //List(n)=List(n) xor List2(n)
       M,_main(1),Z;           // List(n+1)
       M,_main(2),Z;           // List2(n+2)
       One,counter,Z;          // counter--
       0,counter,Z,loop;       // LEQ goto loop
       Z,Z,Z,_main;            // goto _main
       loop: Z,Z,Z,loop;       // End of program
```

The following code translates to:

```
0000 00004 00004 00004 00038    |_zero,_zero,_zero,_main;
0004 00000 00006 00001 00002    |_zero,List_address,List(0),List(1)
0008 00003 00004 00005 00006    |List(2),List(3),List(4),List(5)
000c 00007 00008 00009 0000a    |List(6),List(7),List(8),List(9)
0010 00011 00001 00002 00003    |List2_address,List2(0),List2(1),List2(2)
0014 00004 00005 00006 00007    |List2(3),List2(4),List2(5),List2(6)
0018 00008 00009 0000a 00000    |List2(7),List2(8),List2(9),Z
001c 00001 1ffff 0000a 00006    |One,M,counter,list_add
0020 0000f 00000 00000 00000    |list_add_10
0024 00000 00000 00000 00000
0028 00000 00000 00000 00000    |This empty slots are reserved to hold the values
002c 00000 00000 00000 00000    |of undefined variables with the option [-u] enabled.
```

```
0030 00000 00000 00000 00000
0034 00000 00000 00000 00000
0038 0001b 00006 00011 0003c    |_main: Z,List(0),List2(0);
003c 0001d 00039 0001b 00040    |       M,_main(1),Z;
0040 0001d 0003a 0001b 00044    |       M,_main(2),Z;
0044 0001c 0001e 0001b 00048    |       One,counter,Z;
0048 00024 0001e 0001b 00050    |       0,counter,Z,loop;
004c 0001b 0001b 0001b 00038    |       Z,Z,Z,_main;
0050 0001b 0001b 0001b 00050    |       loop: Z,Z,Z,loop;
```

The result of the slxs simulator is:

```
0000 00004 00004 00004 00038    | The List has been xored with he List2 and
0004 00000 00006 00000 00000    | the results where saved in the addresses
0008 00000 00000 00000 00000    | of the List.
000c 00000 00000 00000 00000
0010 00011 00001 00002 00003
0014 00004 00005 00006 00007
0018 00008 00009 0000a 00000
001c 00001 1ffff 00000 00006
0020 0000f 00000 00000 00000
0024 00000 00000 00000 00000
0028 00000 00000 00000 00000
002c 00000 00000 00000 00000
0030 00000 00000 00000 00000
0034 00000 00000 00000 00000
0038 0001b 00010 0001b 0003c
003c 0001d 00039 0001b 00040
0040 0001d 0003a 0001b 00044
0044 0001c 0001e 0001b 00048
0048 00024 0001e 0001b 00050
004c 0001b 0001b 0001b 00038
0050 0001b 0001b 0001b 00050
```

Bibliography

[CS16] Antony Charalambous and Vasilis Sikkis. Msp430 present impemenation. https://github.com/Sikkis/hardware_exercise1_present, 2016.

[INT16] Intel hex. https://en.wikipedia.org/wiki/Intel_HEX, 2016. Online; accessed: 11-June-2016.

[Sub16] Subleq-esolang. https://esolangs.org/wiki/Subleq, 2016. Online; accessed: 18-August-2016.

[UNP] UNPUBLISHED. One instruction set computers as cryptographic accelerators. Provided by David Oswald.

Appendices

Appendix A

Code Sources

This appendix contains how the project that was submitted is structured. The SLXS toolchain is stored into the *tool* folder with the *algorithms* and *operators* containing samples of SLXS code. The *extras* folder contains small applications that were used in the early stages of the project but may be useful in future stages of the project.

```
-algorithms
    -AES
        aes_slxs.txt                | The AES algorithm implementation on SLXS
        aes_reference.txt           | The reference Aes algorithm.
    -PRESENT
        present.c                   | The Present algorithm implementation on C
        present.txt                 | The Present algorithm implementation on SLXS
        present_precomputedval.txt  | The Present algorithm implementation on SLXS
                                    | with precomputed pbox
    -extras
        -intelhex_converter
            intelhex_converter.c    | Intel hex converter implementation on C
        -mem_template
            template_creator.c      | Memory template implementation on C
        -simulator
            main.c                  | SLXS Simulator on C
            Makefile
    -operators
        and.txt                     | The and operator on SLXS
        div_mod.txt                 | The division, modulo operator on SLXS
        lshift.txt                    | The left shift operator on SLXS
        macros.txt                    | Macros examples on SLXS
        mul.txt                       | The multiplication operator on SLXS
        or.txt                        | The or operator on SLXS
        rshift.txt                    | The right shift operator on SLXS
    -tool
        -output_files                 | Output files created SLXS tool
            mem0.hex                  | First intel hex format file
            mem1.hex                  | Second intel hex format file
            mem2.hex                  | Third intel hex format file
            mem3.hex                  | Forth intel hex format file
        slxs.py                       | The SLXS tool
        slxs_macros.py                | The SLXS tool with macros
    README.MD                         | Readme file with how to use tutorial
    results.txt                       | Results of operators and algorithms
```

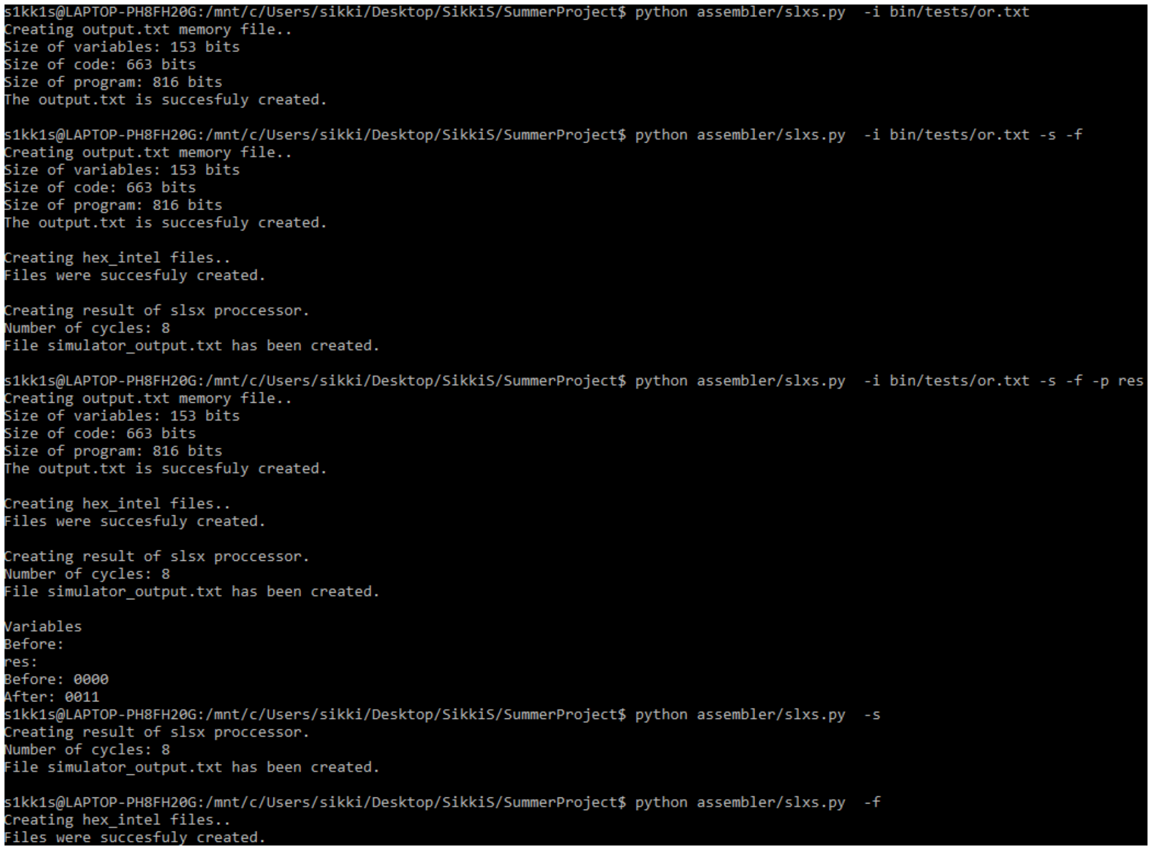
The *aes_slxs.txt* was created and adapted to run in the SLXS toolchain from the use of the provide material *aes_reference.txt*. The *present.c* file was a part of an exercise of the Hardware and Embedded Systems that we were instructed to implement [CS16]. A variation of the simulator, inside the extras directory, was aslo provided, the version that is submitted has minor changes and it was translated from C to Python and used in the SLXS toolchain. The content of the folder *output_files* are automatically generated from the tool to produce the end result *simulator_output.txt*.

Appendix B

Manual

The SLXS tool can start from any of the first three stages introduced in 3.1. The three stages are: first the SLXS code which corresponds to the $[-ifilename]$ option, the memory address which is the option $[-f]$, and the Intel hex format files which is the $[-s]$ option. It is recommended to use the three options together $[-i\ filename,\ -s,\ -f]$ so the $[-p]$ option is available, because the program needs to know the name of the variable that is initialized in the code.

The SLXS tool is located into the *tool* directory. With the command *python tool/slxs.py -i input_file -f -s* the program will execute and produce the result. The *output_files* directory will be created with the results in the current directory. While the macro-assembler is running, programming errors can be found. For additional help the command *python tool/slxs.py –help* can be used.



```
s1kk1s@LAPTOP-PH8FH20G:/mnt/c/Users/sikki/Desktop/SikkiS/SummerProject$ python assembler/slxs.py  -i bin/tests/or.txt
Creating output.txt memory file..
Size of variables: 153 bits
Size of code: 663 bits
Size of program: 816 bits
The output.txt is succesfuly created.

s1kk1s@LAPTOP-PH8FH20G:/mnt/c/Users/sikki/Desktop/SikkiS/SummerProject$ python assembler/slxs.py  -i bin/tests/or.txt -s -f
Creating output.txt memory file..
Size of variables: 153 bits
Size of code: 663 bits
Size of program: 816 bits
The output.txt is succesfuly created.

Creating hex_intel files..
Files were succesfuly created.

Creating result of slsx proccessor.
Number of cycles: 8
File simulator_output.txt has been created.

s1kk1s@LAPTOP-PH8FH20G:/mnt/c/Users/sikki/Desktop/SikkiS/SummerProject$ python assembler/slxs.py  -i bin/tests/or.txt -s -f -p res
Creating output.txt memory file..
Size of variables: 153 bits
Size of code: 663 bits
Size of program: 816 bits
The output.txt is succesfuly created.

Creating hex_intel files..
Files were succesfuly created.

Creating result of slsx proccessor.
Number of cycles: 8
File simulator_output.txt has been created.

Variables
Before:
res:
Before: 0000
After: 0011
s1kk1s@LAPTOP-PH8FH20G:/mnt/c/Users/sikki/Desktop/SikkiS/SummerProject$ python assembler/slxs.py  -s
Creating result of slsx proccessor.
Number of cycles: 8
File simulator_output.txt has been created.

s1kk1s@LAPTOP-PH8FH20G:/mnt/c/Users/sikki/Desktop/SikkiS/SummerProject$ python assembler/slxs.py  -f
Creating hex_intel files..
Files were succesfuly created.
```

Figure B.1: Execution Examples

Figure B.1 shows some examples how to run the SLXS toolchain. The first execution runs only the macro-assembler and the second in addition to that, it creates the hex files and runs the simulator. The only difference the third execution haves is that it prints the res variable in the command line. The two last examples show the two options $[-s, -f]$ that can be used separately if the files are provided in the output directory.

Appendix C

Operators

The Operators Appendix contains the implementation of the behavior for each of the operators in the SLXS language:

C.1 And

```
/*This program calculates the AND operator.*/
/* Variables */
Z   : 0
mxr : 0
One : 1
xor : 0
res : 0
x : 0x11
y: 0x10
my: 0

/*main function*/
_main:res, res, x;
y  , my , Z;
my , res, Z;
xor, xor, x;
Z  , xor, y;
xor, res, Z,_SH;
loop: Z,Z,Z,loop;
```

C.2 Or

```
/*This program calculates the OR operator.*/
/* Variables */
Z   : 0
mxr : 0
One : 1
xor : 0
res : 0
x : 0x11
y: 0x10
my: 0

/*main function*/
_main:res, res, x;
      y  , my , Z;
      my , res, Z;
      xor, xor, x;
      Z  , xor, y;
      xor, mxr, Z;
      mxr, res, Z,_SH;
      loop: Z,Z,Z,loop;
```

C.3 Multiplication

```
/*This program calculates the multiply operator.*/
/* Variables */
Z   : 0
One : 1
x :-8
y: 2
counter:0
x_n:0
y_n:0
until:0
M:-1
flx:0
```

```
flx_n:0
fly:0
val2:2
flags:0

/*main function*/
_main:Z,x,Z,neg;  //x<=0 jump to neg
      Z,y,Z,neg;  //y<=0 jump to neg
      One,y,Z,loop; //if y=1 end
      M,y,Z;
      /*If both numbers are possitive*/
      counter,counter,One;
      until,until,y;
      x_n,x_n,Z;
      x,x_n,Z;
      start:x_n,x,Z;
        M,counter,Z;
        counter,until,Z,loop;
        until,until,y,start;

      neg:x,x_n,Z,flx_end;      //x=0 jump to flx_end
      M,flx,Z;
      flx,flx_n,Z;
      flx_end:y,y_n,Z,fly_end;  //y=0 jump to flx_end
      M,fly,Z;
      flags,flags,fly;
      flx_n,flags,Z;
      M,flags,Z;                  //add another one so you can compare correctly


      /*IF both numbers are negative:*/
      fly_end:val2,flags,Z,continue; // flags==2
      counter,counter,One;
      One,y_n,Z,end_two_neg;
      M,y_n,Z;
      until,until,y_n;
      startneg:x,x_n,Z;
```

```
    M,counter,Z;
    counter,until,Z,end_two_neg;
    until,until,y_n,startneg;
    end_two_neg:x,x,x_n,loop;

  continue:Z,flx,Z,continue1;
  x,x,x_n,oneneg;

  continue1:Z,fly,Z,end;
  y,y,y_n,oneneg;

  /*If one of the two numbers is negative*/
  oneneg:One,y,Z,loop; //if y=1 end
        M,y,Z;
        counter,counter,One;
        until,until,y;
        x_n,x_n,Z;
        x,x_n,Z;
        startn1:x_n,x,Z;
          M,counter,Z;
          counter,until,Z,endn1;
          until,until,y,startn1;
        endn1:x_n,x_n,Z;
              x,x_n,Z;
              x,x,x_n,loop;

  /*If one of the two numbers is 0*/
  end:x,x,Z;
  loop: Z,Z,Z,loop;
```

C.4 Division/Modulo

```
/*This program calculates the divisor and the modulo of two numbers*/
/*Variables*/
x:20
y:0
cony:8
```

```
d:0
mod:0
Z:0
M:-1
prev_st:0
U:1

/*Main*/
_main:M,y,Z,neg; //First check if y = 0 if it is got to yZero
      U,y,Z,yZero;
      Z,Z,Z,st;
      neg:U,y,Z;

      st:d,d,Z;
      cony,cony,y;
      start:y,y,cony;
      prev_st,prev_st,x;
      M,x,Z; //to make if statement less from lessequal
      y,x,Z,negative;
      U,x,Z;
      mod,mod,x;
      M,d,Z;
      mod,y,Z,start;
      Z,Z,Z,loop;
      negative:mod,mod,prev_st;
      d,d,Z,loop;
      yZero:mod,mod,M;  //If y=Z d=-1 mod =-1 (cant be calculated)
      d,d,M;
      loop:Z,Z,Z,loop;
```

C.5 Left Shift

```
/*This program calculates the LEFT SHIFT operator.*/
/*Variables*/
x:10
y:5
t_y:0
```

```
shtemp:0
t:0
N:0
Z:0
M:-1

/*Main*/
_main:t_y,t_y,y;
      shtemp,shtemp,Z;
      t,t,Z;
      x,t,Z;
      Z,t_y,Z,loop;                //skip if shift equals to Zero
      ls_loop:t,x,Z;
              t,t,Z;
              x,t,Z;
              M,shtemp,Z;
              shtemp,t_y,Z,loop;
              t_y,t_y,y,ls_loop;
      loop:Z,Z,Z,loop;
```

C.6 Right Shift

```
/*This program calculates the RIGHT SHIFT operator.*/
/*Variables*/
x:100
y:4
t_y:0
Z:0
shtemp:0
M:-1

/*Main*/
_main:t_y,t_y,y;
      Z,t_y,Z,loop; //shift is 0
      shtemp,shtemp,Z;
      rs_loop:Z,x,Z,_SH;
             M,shtemp,Z;
```

```
        shtemp,t_y,Z,loop;
        t_y,t_y,y,rs_loop;
loop:Z,Z,Z,loop;
```