RSSB SIMULATOR DOCUMENTATION

RYLEY JEWSBURY

1. Instruction Set

All the available scripts.
Movement:

- INIT A
- MOV A, B
- MOVN A, B
- NEG A
- SWAP A, B
- LOAD A, [B]
- LOAD A, [B,C]
- STR A, [B]
- STR A, [B,C]
- PUSH A
- POP A

Arithmetic:

- ADD A, B, C
- ADD A, B
- SUB A, B, C
- SUB A, B
- SUBP A, B
- SUBN A, B

Control:

- IFLT A, B
- IFGT A, B
- ELSE
- END
- B label
- BL label
- BX A
- BXL A
- HALT
- NOP

2. Syntax

Code is written in plaintext, and stored in .src files. Each line of code consists of:

- A label, ending in a colon (optional)
- an operation
- 0 to 3 operands, depending on the instruction. Separated by commas.
- comments, starting with a semicolon (optional)

e.g.

```
code: ADD R0, R1 ;one-line example
```

1. **sections.** :

Every script must have 3 sections: text, data, stack. These sections tell the assembler where to place code, constants, and stack space, respectively. To start a section, use the operation `.section` with the argument `.text`,`.stack`, or `.data`. Once a section has been started, it needs to be given an origin memory address using `.origin`. Even if these sections are left empty, their origins must be specified.

note: The first 18 addresses are reserved for pre-defined names. The lowest address a section can start at is `0x12`

e.g.

```
.section .text
.origin 0x0020
```

2. **constants.** :

Constants can be written in either decimal, or hexadecimal. Hexadecimal constants are identified by the prefix `0x`. Addresses of labelled data or instructions can also be treated as constants. To refer to the address of a label, prefix it with =. The RSSB simulator uses a 16 bit architecture, so when treating constants as signed, the maximum constant that can be stored is 0x7FFF, and the minimum is -0x8000. unsigned constants can go up to 0xFFFF

e.g.

```
0020> list: .word 15
0021>       .word 0x2F
0022>       LOAD list, [=list, 1] ;stores 0x2F in list, overwriting 15
```

3. **registers.** :

The first 18 addresses are reserved to act as "registers", although they are effectively just labelled. when writing source code, the only important registers are thegeneral purpose registers, `R0` to `R11`, and the link register, `LR`. The rest are used by the asembler to write RSSB instructions.

3. Instruction Descriptions

1. **Assembler Instructions.**

.section [.text|.data|.stack]. :
Tells the assembler which section to place the following code in

.origin [constant]. :
Tells the assembler what memory address to begin a section at
note: The assembler will not realize if sections overlap. It is up to the programmer to make sure they do not overwrite one section with another.

.word [constant]. :
Stores a value in memory. If a label is used on this line, the address can be treated as a register elsewhere in the code.

2. **Data Movement Scripts.**

INIT A. :
Sets A and ACC to 0

MOV A, B. :
moves the value from B into A
if &A = &B, replace with NOP

MOVN A, B. :
moves the negative value -B into A
note: A cannot be the same register as B. use NEG instead

NEG A. :
negates the value in A

SWAP A, B. :
moves the value from A into B, and the value from B into A
if &A = &B, replace with NOP

LOAD A, [B]. :
Loads the data from the address stored in B into A

LOAD A, [B,C]. :
Loads the data from the address stored in B+C into A
assumes that B is positive and B+C is positive

STR A, [B]. :
Stores the data from A into the address stored in B

STR A, [B,C]. :
Stores the data from A into the address stored in B+C

PUSH A. :
Stores the data from A on the stack

POP A. :
Stores the data from the top of the stack in A

3. **Arithmetic Scripts.**

ADD A, B, C. :

Stores the result B+C in A

ADD A, B. :

Stores the result A+B in A

SUB A, B, C. :

Stores the result B-C in A

SUB A, B. :

Stores the result A-B in A. **note:** you would think this script would be really short, but the skipping of RSSB ruins it, so some shorter scripts are provided by SUBP and SUBN

SUBP A, B. :

Stores the result A-B in A
requires that B is positive or zero
if B is negative, NOP

SUBN A, B. :

Stores the result A-B in A
requires that B is negative
if B is positive or zero, NOP

4. **Flow Control Scripts.**

IFLT A, B. :

continues execution until ELSE if $A < B$
jumps to ELSE otherwise (when $A \geq B$)

IFGT A, B. :

continues execution until ELSE if $A > B$
jumps to ELSE otherwise (when $A \leq B$)
note: testing if two numbers are equal can be done by using both IFLT and IFGT

ELSE. :

separates conditional blocks
note: must always be included, even if the else block is empty

END. :

ends a conditional block
note: must always be included after ELSE

B label. :

Jumps a number of steps. distance to labels must be pre-computed by the compiler

BL label. :

Jumps a number of steps. distance to labels must be pre-computed by the compiler
Updates the Link Register (LR)

BX A. :

Jumps to the address stored in A
note: Typically used as (BX LR) to return from a function

BXL A. :

Jumps to the address stored in A
Updates the Link Register

HALT. :

Stops execution
note: due to the simple hardware, stopping just busy-loops on addresses 0 to 2