Team members: Yuyong Chen, Jie Wang

Introduction:

About Lambo architecture:

The Lambo architecture is a simple and easy to code, highly efficient architecture that uses 9-bit encoded instructions. It supports 6 general purpose registers, 2 less general registers for specialized usage in encryption software, and a few internal registers for flagging. 8 addressable registers allow for reduced memory interactions for performance. There are 12 different instructions specified with also 3 bits, fully capable of building small programs.

Instruction Formats:

RR-type:

Instructions that process two registers, each expressed in 3 bits.
Example: <ADD r1, r2>, <STR r1, [r3]>

RC-type:

Instructions that process one register and one constant, each expressed in 3 bits
Example: <LSL r3, #5>

R-type:

Instructions that process one register, expressed in 3 bits
Example: <RDX r1>

B-type:

Branch instructions, with target within [-15, 15] from current PC
Example: <BEQ 'label'>

S-type:

Instructions that does not take arguments.
Example: <HALT>, <SHIFT>

Operations:

Instruction - Opcode
Format Type
Usage
Description

Compare - 000
RR
CMP ra, rb
Compare register a and b and set the "is >=" condition flag

Branch - 0010/0011
B
B/BGE #offset
Branch to address with offset [-15, 15], option of looking at condition flag

Load registr - 010
RR
LDR ra, [rb]
Load value from memory address *rb into ra and increment rb by 1

Store register - 011
  RR
  STR ra, [rb]
  Store value from ra into memory address *rb and increment rb by 1

Xor registers - 100
  RR
  XOR ra, rb
  Xor register a and b and store the result to a

Add registers - 101
  RR
  ADD ra, rb
  Add value of register a and b and store the result to a

Subtract registers - 110
  RR
  SUB ra, rb
  Subtract value of register b from a and store the result to a

Logical left shift - 111
  RC
  LSL ra, #shift_amount
  Logical left shift register a by shift_amount

Halting - 001110000
  S
  HALT
  Halts the Machine

LFSR Shift - 001010000
  S
  SHIFT
  Shift lfsr register (r7) with tap in tap register (r6)

Reduction XOR - 111110
  R
  RO ra
  Xor every bit of register a and store the result to least significant bit of a

Set register - 111111
  R
  MOV ra, #value
  Set the value of the specified register with value placed the next machine instruction

Internal Operands:
  8 registers are supported (r1-r8)
  special register r7 stores LFSR tap pattern, LSL not supported

special register r8 stores LFSR, LSL not supported

Control Flow:
  2 types of branches are supported: branch always & branch greater than or equal to
  target address calculated by adding offset value (2's complement) to the current address
  Branch range [-15, 15]

Addressing Mode:
  register stores pointer to memory address, e.g. [ra] = memory[ra]

Machine Classification:
  reg-reg & load-store

Assembly Example:

```
 Assembly:
  d:
    LDR r4, [r1]
    MOV r5, #4
    LSL r5, #3
    SUB r4, r5
    XOR r4, r7
    RDX r5
    SHIFT!
    CMP r3, r1
    BGE d
    HALT

 Machine code:
  010100001
  111111101
  000000100
  111101011
  110100101
  100100111
  111110101
  001010000
  000011001
  001111000
  001110000
```

```
|------------Quick Reference-------------|
|Regular 3-bit map:                      |
|   000 CMP reg, reg                     |
|   0010 B                               |
|   0011 BGE                             |
|   010 LDR reg, [reg]                   |
|   011 STR reg, [reg]                   |
|   100 XOR reg, reg                     |
|    101 ADD reg, reg                   |
|    110 SUB reg, reg                  |
|    111 LSL reg, #constant             |
|                                |
|Special case:                        |
|    (001)010000  SHIFT                |
|    (001)110000  HALT                 |
|    (111)110    RDX reg               |
|                                |
|Special operation:                   |
|    (111)111 followed by XXXXXXXXX      |
|    MOV reg                          |
|------------Quick Reference-------------|
```

```
Program 1:
   MOV r1, #0                   // r1: msgPt = 0
   MOV r2, #61                  // r2: encPt = 61
   LDR r3, [r2]                 // r3: numSpace = data[encPt++]

   // If (numSpace > 15) numSpace = 15;
   MOV r4, #10                  // r4 = 10
   CMP r3, r4
   BGE a                       // if numSpace >= 10 go a

   MOV r3, #10                  // numSpace = 10
   B b

  a:
   MOV r4, #15                  // r4 = 15
   CMP r4, r3
   BGE b                          // if 15 >= numSpace go b
   MOV r3, #15                  // r3 = 15

  b:
   LDR r7, [r2] //r7: tap         // r2: 63
   LDR r8, [r2] //r8: lfsr        // r2: 64
   MOV r5, #1

  c:                          // Prepend spaces
   MOV r4, #0
   XOR r4, r8                  // r4 = lfsr
   STR r4, [r2]                // data[encPt++] = r4

   SHIFT!                      // Call LFSR hardware module

   SUB r3, r5                  // numSpace--
   CMP r3, r1                   // while numSpace >= 0
   BGE c
   MOV r3, #60                  // r3 = 60
   MOV r6, #32

  d:                          // Encrypt message
   LDR r4, [r1]                 // r4 = data[msgPt++]
   SUB r4, r6                   // r4 -= 0x20
   XOR r4, r8                   // r4 ^= lfsr

   // Parity bit
   SUB r5, r5
   ADD r5, r4
   RDX r5                       // r5 = ^(r4)
   LSL r5, #7                   // r5 = r5 << 7
   ADD r4, r5                   // r4 += r5

   STR r4, [r2]                 // data[encPt++] = r4

   SHIFT!                       // Call LFSR hardware module

   CMP r3, r1                   // while 60 >= msgPt
   BGE d

   HALT!
```

Program 2&3:

```
  MOV r1, #0                   // r1: msgPt = 0
  LDR r3, #128                 // r3: lfsr_ptrnPt

  // Load each pattern
  MOV r4, 0x60
  STR r4, [r3]
  MOV r4, 0x48
  STR r4, [r3]
  MOV r4, 0x78
  STR r4, [r3]
  MOV r4, 0x72
  STR r4, [r3]
  MOV r4, 0x6A
  STR r4, [r3]
  MOV r4, 0x69
  STR r4, [r3]
  MOV r4, 0x5C
  STR r4, [r3]
  MOV r4, 0x7E
  STR r4, [r3]
  MOV r4, 0x7B
  STR r4, [r3]

  LDR r3, #128                 // r3: lfsr_ptrnPt
  MOV r4, #1

a:                          // |-----------LOOP a------------|
  MOV r2, #64                    // encPt = 64
  LDR r8, [r2]                 // rLFSR = start_state = data[encPt++]

  LDR r5, #8                   // r4: numShift - 1 = 8
  LDR r7, [r3]                 // rTAP = data[lfsr_ptrnPt]

  B b
h:
  B a                        // Branch pitstop

b:                          // |---------LOOP b---------|
  SHIFT
  LDR r6, [r2]                 // r6 = data[encPt++]

  CMP r6, r8
  BGE c
  B f                       // pattern match failed, break
c:
  CMP r8, r6
  BGE d
  B f                       // pattern match failed, break
d:
  SUB r5, r4
  CMP r5, r0                    // while numShift >= 0
  BGE b                       // |---------LOOP b---------|

  B g                        // pattern match, break
e:
  B h                        // Branch pitstop

f:
  MOV r6, #136
  CMP r6, r3                    // while 136 >= lfsr_ptrn
  BGE e                       // |-----------LOOP a------------|

  MOV r0, #127                    // SHOULD NOT GET HERE!!
  MOV r1, #127                    // SHOULD NOT GET HERE!!
  HALT                        // First 10 bytes def have corruption

  MOV r3, 0x20
  MOV r4, #127

g:
  SHIFT

  LDR r6, [r2]
  CMP r6, r8
  BGE i
  B l
i:
  CMP r8, r6
  BGE k
  B l
k:
  CMP r4, r2                    // while encPt <= 127
  BGE g

l:
  SHIFT
  LDR r6, [r2]

  SUB r5, r5
  ADD r5, r6
  RDX r5                      // r5 = ^(r6)
  SUB r3, r3
  CMP r3, r5
  BGE o:                      // go o if r5 is 0

  MOV r6, 0x80                  // Error char
  B p

q:
  B l                        // Branch pitstop

o:
  XOR r6, r8
  MOV r3, 0x20
  ADD r6, r3                    // byte = enc ^ lfsr + 0x20
p:
  STR r6, [r1]                 // data[msgPt++] = byte
  CMP r4, r2                    // while encPt <= 127
  BGE q

  MOV r4, #63
m:
  SHIFT
  SUB r6, r6
  XOR r6, r8
  ADD r6, r3                    // byte = enc ^ lfsr + 0x20
  STR r6, [r1]                 // data[msgPt++] = byte
  CMP r4, r1                    // while msgPt <= 63
  BGE m

  HALT
```