```
/* Binary Encodings */

/* STACK */
/*
 * END  00000000 | 4-bit base address | 4-bit multiplier | 16-bit offset field
 * PUSH 00000001 | 4-bit base address | 4-bit multiplier | 16-bit offset field
 * POP  00000010 | 4-bit base address | 4-bit multiplier | 16-bit offset field
 * ADD  00000011 | 4-bit base address | 4-bit multiplier | 16-bit offset field
 * MUL  00000100 | 4-bit base address | 4-bit multiplier | 16-bit offset field
 * PIMM 00000101 | 24-bit immediate field
 */


/* ACCUMULATOR */
/*
 * END  00000000 | 4-bit base address | 4-bit multiplier | 16-bit offset field
 * LOAD 00000001 | 4-bit base address | 4-bit multiplier | 16-bit offset field
 * STO  00000010 | 4-bit base address | 4-bit multiplier | 16-bit offset field
 * ADD  00000011 | 4-bit base address | 4-bit multiplier | 16-bit offset field
 * MUL  00000100 | 4-bit base address | 4-bit multiplier | 16-bit offset field
 * LIMM 00000101 | 24-bit immediate field
 */


/* ANALYSIS */
/*
 * To accomodate 140 instructions we need 8 bits for the opcode
 * For the address, since we only have 32 bits, we'll have to come up with some
kind of way to construct a target address
 * Here we have a 4 bit field to choose between 16 different base addresses,
which will be registers
 * In our case we can speciify the different areas of memory here (for example,
0000 will be address of user text, 0001 user data, etc...)
 * We can use a multiplier which will multiply the base address by 2 to the
power of the value we load into this 4-bit field, considerably extending the
range of addresses we can reach
 * From there we can specify a simple 16-bit offset immediate field
 * We have two new instructions, PUSH IMMEDIATE (PIMM) and LOAD IMMEDIATE (LIMM)
which have a 8 bit opcode and 24-bit immediate field
 */
```

```
/* Example of how accumulator code would break down to binary

      .text

    main:                  OPCODE    Base   MULT    Address Offset      Hex
        LOAD A          # 00000001 | 0001 | 0000 | 0000000000000001 | 01100001
        MUL  X          # 00000100 | 0001 | 0000 | 0000000000000000 | 04100000
        MUL  X          # 00000100 | 0001 | 0000 | 0000000000000000 | 04100000
        STO  A          # 00000010 | 0001 | 0000 | 0000000000000001 | 02100001
        LOAD B          # 00000001 | 0001 | 0000 | 0000000000000002 | 01100002
        MUL  X          # 00000100 | 0001 | 0000 | 0000000000000000 | 04100000
        STO  B          # 00000010 | 0001 | 0000 | 0000000000000002 | 02100002
        LOAD C          # 00000001 | 0001 | 0000 | 0000000000000003 | 01100003
        ADD  A          # 00000011 | 0001 | 0000 | 0000000000000001 | 03100001
        ADD  B          # 00000011 | 0001 | 0000 | 0000000000000002 | 03100002
        STO  ANS        # 00000010 | 0001 | 0000 | 0000000000000004 | 02100004
        END             # 00000000 | 0000 | 0000 | 0000000000000000 | 00000000

        .data

                                       Binary                     Hex
    A: 7                # 00000000000000000000000000000111 | 00000007
    B: 5                # 00000000000000000000000000000101 | 00000005
    X: 3                # 00000000000000000000000000000011 | 00000003
    C: 4                # 00000000000000000000000000000100 | 00000004
    ANS: 0              # 00000000000000000000000000000000 | 00000000

    The base for most of the commands corresponds to the data section where the
operands are held. The multipler is simply zero.
    Then we find the address offset by looking at the order of the data section
variables.
 */

/*
 * In the hex we could give our file a simple format to know when the different
sections are.
 * We'll denote .text with T and .data with D. We will also specify how many
bytes are in each section
 *
 * T|30|01100001|04100000|04100000|02100001|01100002|04100000|02100002|01100003|
03100001|03100002|02100004|00000000|
 * D|14|00000003|00000007|00000005|000000004|00000000|
 *
 * Here we have T and D to denote our sections. Afterwards, we have the number
of bytes in hex (30 would be 48 bytes).
 * Then we have our stream of instructions.
 * Here in total we have 68 (decimal) bytes
 */




/* Example of how stack code would break down to binary
```

```
    .text

  main:                    OPCODE    Base   MULT    Address Offset      Hex
      PUSH C           # 00000001 | 0001 | 0000 | 0000000000000003 | 01100003
      PUSH B           # 00000001 | 0001 | 0000 | 0000000000000002 | 01100002
      PUSH X           # 00000001 | 0001 | 0000 | 0000000000000000 | 01100000
      MUL              # 00000100 | 0000 | 0000 | 0000000000000000 | 04000000
      PUSH X           # 00000001 | 0001 | 0000 | 0000000000000000 | 01100000
      PUSH X           # 00000001 | 0001 | 0000 | 0000000000000000 | 01100000
      MUL              # 00000100 | 0000 | 0000 | 0000000000000000 | 04000000
      PUSH A           # 00000001 | 0001 | 0000 | 0000000000000001 | 01100001
      MUL              # 00000100 | 0000 | 0000 | 0000000000000000 | 04000000
      ADD              # 00000011 | 0000 | 0000 | 0000000000000000 | 03000000
      ADD              # 00000011 | 0000 | 0000 | 0000000000000000 | 03000000
      POP  ANS         # 00000010 | 0001 | 0000 | 0000000000000004 | 02100004
      END              # 00000000 | 0000 | 0000 | 0000000000000000 | 00000000

      .data
                                       Binary                       Hex
  X: 3                 # 00000000000000000000000000000011 | 00000003
  A: 7                 # 00000000000000000000000000000111 | 00000007
  B: 5                 # 00000000000000000000000000000101 | 00000005
  C: 4                 # 00000000000000000000000000000100 | 00000004
  ANS: 0               # 00000000000000000000000000000000 | 00000000

    The base for many commands corresponds to the data section where the
operands are held. The multipler is simply zero.
    Then we find the address offset by looking at the order of the data section
variables. Commands like MUL, ADD, and END
    only need the opcode as they will use the stack pointer register to perform
the arithmetic.
 */

/*
 * In the hex we could give our file a simple format to know when the different
sections are.
 * We'll denote .text with T and .data with D. This will be followed by a byte
to specify how many bytes are in each section.
 *
 * T|34|01100003|01100002|01100000|04000000|01100000|01100000|04000000|01100001|
04000000|03000000|03000000|02100004|00000000|
 * D|14|00000003|00000007|00000005|00000004|00000000|
 *
 * Here we have T and D to denote our sections. Afterwards, we have the number
of bytes in hex (34 would be 52 bytes).
 * Then we have our stream of instructions.
 * Here in total we have 72 (decimal) bytes
 */
```