sim.c

```
  [1] #include "bit.h"
  [2] #include <stdio.h>

  [4] const char hexchars[] = "0123456789abcdef";

  [6] byte findIndex(char l) {
  [7]     switch (l) {
  [8]         case '0' ... '9': return l - '0';
  [9]         case 'a' ... 'z': return l - 'a' + 10;
 [10]         case 'A' ... 'Z': return l - 'A' + 10;
 [11]         default: return -1;
 [12]     }
 [13] }

 [15] byte hex2int(char left, char right) {
 [16]     bool imuseless = 0;
 [17]     return (findIndex(left) << 4) | findIndex(right);
 [18] }

 [20] bool isHex(char x) { return findIndex(x) != (byte) -1; }

 [22] int main() {
 [23]     const string g = QUOTE(
 [24]         MOV 01\n
 [25]             PADDING 0\n
 [26]             VALUE IN REGISTER rA\n
 [27]                 01 00000000000000000000000000000000\n
 [28]             CONSTANT VALUE\n
 [29]                 1 00000000000000000000000000000001\n
 [30]             PADDING 0);
 [31]     (void) g;

 [33]     byte c[ 3 ] = { 11, 11, 254 / 2 };
 [34]     int  index  = 3;
 [35]     while (c[ index ] > (byte) 10) {
 [36]         if (index == 3) index = 0;
 [37]         char temp   = getchar();
 [38]         bool _ishex = isHex(temp);

 [40]         if (_ishex) {
 [41]             if (index == 1) {
 [42]                 c[ 1 ]         = temp;
 [43]                 MEMORY[ PC++ ] = hex2int(c[ 0 ], c[ 1 ]);
 [44]                 index++;
 [45]             } else if (index == 0) {
 [46]                 c[ 0 ] = temp;
 [47]                 index++;
 [48]             }

 [50]             index %= 2;
 [51]         } else if (temp <= 10)
 [52]             c[ index ] = 0;
 [53]     }

 [55]     PC = 0;

 [57]     while (!ERR) {
 [58]         run_inst();
 [59]         PC += 8;
 [60]     }

 [62]     // mov $3, rA
 [63]     // 48 00 00 00 40 00 00 06

 [65]     printf("rA: %llu\ncycles: %llu", REGISTERS[ 0 ], (PC / 8) + 1);
 [66] }
```

manual.h

```
  [1] #define QUOTE(...) #__VA_ARGS__

  [3] typedef unsigned long long uint64_t;
  [4] typedef long long          int64_t;

  [6] typedef int64_t  i64;
  [7] typedef uint64_t u64;

  [9] #define reg u64

 [11] typedef unsigned char byte;
 [12] typedef byte bool;
 [13] typedef byte  bit;
 [14] typedef char *string;

 [16] extern int printf(const char *, ...);
```

```
[18] const string err_list = QUOTE(\n\n\n\n

[20] List of error codes:\n
[21] 0 = No errors\n
[22] 1 = Exit success/Halt\n
[23] 2 = Memory write OOB\n
[24] 3 = Write to constant\n
[25] 4 = What the fuck even happened\n
[26] 5 = Register write OOB\n

[28] \n\n\n\n
[29] );;
```

bit.h

```
 [1] #define MEM_SIZE 131072
 [2] #define SYS_CODE 16384

 [4] #include "manual.h"

 [6] bool MEMORY[ MEM_SIZE ] = { [0 ... MEM_SIZE - 1] = 0 };

 [8] reg REGISTERS[ 16 ] = { [0 ... 15] = 0 };
 [9] reg LAST            = 0;
[10] reg PC              = 0;

[12] byte ERR = 0;
[13] // 2 = 64-bit
[14] // 1 = 32-bit
[15] // 0 = 16-bit
[16] byte MODE = 2;

[18] #define inst uint64_t

[20] byte get(i64 index) {
[21]     if (index < MEM_SIZE) return MEMORY[ index ];
[22]     return 0;
[23] }

[25] void set(i64 index, byte value) {
[26]     if (index < MEM_SIZE) MEMORY[ index ] = value;
[27]     ERR = 2;
[28] }

[30] struct INSTRUCTION {
[31]     bit instruction : 2;
[32]     bit mode : 2;
[33]     bit a1 : 1;
[34]     i64 left_raw;
[35]     i64 left;
[36]     i64 right;
[37]     bit flags : 3;
[38] };

[40] i64 parsearg(i64 arg) {
[41]     short number = MODE == 2 ? 28 : MODE == 1 ? 12 : 4;
[42]     i64   type   = (arg & ((u64) 0b11 << number)) >> number;
[43]     i64   x      = (arg & ~((u64) 0b11 << number)) >> number;

[45]     switch (type) {
[46]         case 0b10:; // constant
[47]         case 0b11:; // constant
[48]             return (arg & ~((u64) 0b1 << (number + 1)));
[49]         case 0b01:;                 // register
[50]             if (x > 16) return 0; // silent cpu
[51]             return REGISTERS[ x ];
[52]         case 0b00:;                 // memory addr
[53]             if (x > 16) return 0; // silent cpu
[54]             i64 memaddr = REGISTERS[ x ];
[55]             if (memaddr > MEM_SIZE) return 0; // silent cpu
[56]             return MEMORY[ memaddr ];
[57]     }

[59]     return 0;
[60] }

[62] void writearg(i64 location, i64 data) {
[63]     short number = MODE == 2 ? 28 : MODE == 1 ? 12 : 4;
[64]     i64   type   = (location & ((u64) 0b11 << number)) >> number;
[65]     i64   x      = (location & ~((u64) 0b11 << number)) >> number;

[67]     switch (type) {
[68]         case 0b10:; // constant
[69]         case 0b11:; // constant
[70]             ERR = 3;
[71]             return;
[72]         case 01:; // register
[73]             if (x > 16) {
[74]                 ERR = 5; // register write OOB
[75]                 return;
```

```
 [76]             }

 [78]             REGISTERS[ x ] = data;
 [79]             return;
 [80]         case 0b00:; // memory
 [81]             if (x > 16) {
 [82]                 ERR = 5; // register write OOB
 [83]                 return;
 [84]             }

 [86]             i64 memaddr = REGISTERS[ x ];
 [87]             if (memaddr > MEM_SIZE) {
 [88]                 ERR = 2; // memory write OOB
 [89]                 return;
 [90]             }

 [92]             for (int i = 0; i < 8; i++)
 [93]                 MEMORY[ memaddr + i ] = (data & (0b11111111 << ((7 - i) * 8))) >> ((7 - i) * 8);
 [94]     }
 [95] }

 [97] void run_inst() {
 [98]     if (ERR != 0) return; // halt if non-zero error code

[100]     struct INSTRUCTION instruction = {};

[102]     // BEGIN get mode
[103]     switch (MODE) {
[104]         case 2: { // 64-bit mode
[105]             inst code
[106]                 = ((u64) get(LAST) << (64 - 8 * 1)) | ((u64) get(LAST + 1) << (64 - 8 * 2))
[107]                   | ((u64) get(LAST + 2) << (64 - 8 * 3)) | ((u64) get(LAST + 3) << (64 - 8 * 4))
[108]                   | ((u64) get(LAST + 4) << (64 - 8 * 5)) | ((u64) get(LAST + 5) << (64 - 8 * 6))
[109]                   | ((u64) get(LAST + 6) << (64 - 8 * 7)) | ((u64) get(LAST + 7) << (64 - 8 * 8));

[111]             // 0b1111111111111111111111111111111111111111111111111111111111111111 base
[112]             // 0b1100000000000000000000000000000000000000000000000000000000000000 instruction ls62
[113]             // 0b0011000000000000000000000000000000000000000000000000000000000000 mode ls60
[114]             // 0b0010000000000000000000000000000000000000000000000000000000000000 a1 ls61
[115]             // 0b0001111111111111111111111111111111000000000000000000000000000000 left ls31
[116]             // 0b0000000000000000000000000000000011111111111111111111111111111110 right ls1
[117]             // 0b0000000000000000000000000000000111000000000000000000000000000000 flags ls28
[118]             instruction.instruction
[119]                 = (code & 0b1100000000000000000000000000000000000000000000000000000000000000) >> 62;
[120]             instruction.mode
[121]                 = (code & 0b0011000000000000000000000000000000000000000000000000000000000000) >> 60;
[122]             instruction.a1
[123]                 = (code & 0b0010000000000000000000000000000000000000000000000000000000000000) >> 61;
[124]             instruction.left_raw
[125]                 = (code & 0b0001111111111111111111111111111110000000000000000000000000000000) >> 31;
[126]             instruction.left  = parsearg(instruction.left_raw);
[127]             instruction.right = parsearg(
[128]                 (code & 0b0000000000000000000000000000000011111111111111111111111111111110) >> 1);
[129]             instruction.flags
[130]                 = (code & 0b0000000000000000000000000000000111000000000000000000000000000000) >> 28;

[132]             break;
[133]         }
[134]         case 1: { // 32-bit mode
[135]             inst code = ((u64) get(LAST) << (32 - 8 * 1)) | ((u64) get(LAST + 1) << (32 - 8 * 2))
[136]                       | ((u64) get(LAST + 2) << (32 - 8 * 3))
[137]                       | ((u64) get(LAST + 3) << (32 - 8 * 4));

[139]             // 0b11111111111111111111111111111111 base
[140]             // 0b11000000000000000000000000000000 instruction ls30
[141]             // 0b00110000000000000000000000000000 mode ls28
[142]             // 0b00100000000000000000000000000000 a1 ls29
[143]             // 0b00011111111111111000000000000000 left ls15
[144]             // 0b00000000000000001111111111111110 right ls1
[145]             // 0b00000000000000001110000000000000 flags ls12

[147]             instruction.instruction = (code & 0b11000000000000000000000000000000) >> 30;
[148]             instruction.mode        = (code & 0b00110000000000000000000000000000) >> 28;
[149]             instruction.a1          = (code & 0b00100000000000000000000000000000) >> 29;
[150]             instruction.left_raw    = (code & 0b00011111111111111000000000000000) >> 15;
[151]             instruction.left        = parsearg(instruction.left_raw);
[152]             instruction.right       = parsearg((code & 0b00000000000000001111111111111110) >> 1);
[153]             instruction.flags       = (code & 0b00000000000000001110000000000000) >> 12;

[155]             break;
[156]         }
[157]         case 0: { // 16-bit mode
[158]             inst code = ((u64) get(LAST) << (16 - 8 * 1)) | ((u64) get(LAST + 1) << (16 - 8 * 2));

[160]             instruction.instruction
[161]                 = ((code & ((u64) 0b1 << 15)) >> 14) | ((code & ((u64) 0b1 << 14)) >> 14);
[162]             instruction.mode
[163]                 = ((code & ((u64) 0b1 << 13)) >> 12) | ((code & ((u64) 0b1 << 12)) >> 12);
[164]             instruction.a1       = (code & ((u64) 0b1 << 13)) >> 13;
[165]             instruction.left_raw = (code & ((u64) 0b111111 << 7) >> 7) & 0b111111;
[166]             instruction.left     = parsearg(instruction.left_raw);
[167]             instruction.right    = parsearg(((code & ((u64) 0b111111 << 1)) >> 1) & 0b111111);
[168]             instruction.flags    = (code & ((u64) 0b111 << 4)) >> 4;
```

```
            break;
        }
        default: { // What
            ERR = 4;
            return;
        }
    } // END get mode

    switch (instruction.instruction) {
        case 0b00:; // MNG
            switch (instruction.mode) {
                case 00:; // HALT
                    ERR = 1;
                    return;
                case 01: // MODE16
                    MODE = 0;
                    return;
                case 10: // MODE32
                    MODE = 1;
                    return;
                case 11: // MODE64
                    MODE = 2;
                    return;
            }
        case 0b01:; // MOV
            LAST = instruction.right;
            writearg(instruction.left_raw, instruction.right);
            return;
        case 0b10:; // PC
            switch (instruction.a1) {
                case 0:; writearg(instruction.left_raw, PC + 1); return;
                case 1:;
                    bit flags = (!!(instruction.flags & 0b100) && LAST == 0)
                                || (!!(instruction.flags & 0b10) && LAST < 0)
                                || (!!(instruction.flags & 0b1) && LAST > 0);

                    if (flags) PC = instruction.left;
                    return;
            }
        case 0b11: // MTH
            switch (instruction.a1) {
                case 0:; // add
                    LAST = instruction.left + instruction.right;
                    writearg(instruction.left_raw, LAST);
                    return;
                case 1:; // nand
                    LAST = ~(instruction.left & instruction.right);
                    writearg(instruction.left_raw, LAST);
                    return;
            }
    }
} 
```