```
/* Instruction set simulator for Y86 Architecture */

#include <stdio.h>
#include <stdlib.h>

#include "y86sim.h"

#define err_print(_s, _a ...) \
        fprintf(stdout, _s"\n", _a);


typedef enum {STAT_AOK, STAT_HLT, STAT_ADR, STAT_INS} stat_t;

char *stat_names[] = { "AOK", "HLT", "ADR", "INS" };

char *stat_name(stat_t e)
{
        if (e < STAT_AOK || e > STAT_INS)
                return "Invalid Status";
        return stat_names[e];
}

char *cc_names[8] = {
        "Z=0 S=0 O=0",
        "Z=0 S=0 O=1",
        "Z=0 S=1 O=0",
        "Z=0 S=1 O=1",
        "Z=1 S=0 O=0",
        "Z=1 S=0 O=1",
        "Z=1 S=1 O=0",
        "Z=1 S=1 O=1" };

char *cc_name(cc_t c)
{
        int ci = c;
        if (ci < 0 || ci > 7)
                return "???????????";
        else
                return cc_names[c];
}

bool_t get_byte_val(mem_t *m, long_t addr, byte_t *dest)
{
        if (addr < 0 || addr >= m->len)
                return FALSE;
        *dest = m->data[addr];
        return TRUE;
}

bool_t get_long_val(mem_t *m, long_t addr, long_t *dest)
{
        int i;
        long_t val;
        if (addr < 0 || addr + 4 > m->len)
                return FALSE;
        val = 0;
        for (i = 0; i < 4; i++)
                val = val | m->data[addr+i]<<(8*i);
        *dest = val;
        return TRUE;
}

bool_t set_byte_val(mem_t *m, long_t addr, byte_t val)
{
        if (addr < 0 || addr >= m->len)
                return FALSE;
        m->data[addr] = val;
        return TRUE;
}

bool_t set_long_val(mem_t *m, long_t addr, long_t val)
{
        int i;
        if (addr < 0 || addr + 4 > m->len)
                return FALSE;
        for (i = 0; i < 4; i++) {
           m->data[addr+i] = val & 0xFF;
           val >>= 8;
        }
        return TRUE;
}

bool_t push_long_val(y86sim_t *sim, long_t val)
{
        long_t reg_ESP_val = get_reg_val(sim->r,REG_ESP) - 0x4;
        set_reg_val(sim->r,REG_ESP ,reg_ESP_val);
        if(!set_long_val(sim->m,reg_ESP_val,val))
                return FALSE;
```

```
	return TRUE;
}

bool_t pop_long_val(y86sim_t *sim,long_t *dest)
{
	long_t reg_ESP_val = get_reg_val(sim->r,REG_ESP);
	if(!get_long_val(sim->m,reg_ESP_val,dest))
		return FALSE;
	set_reg_val(sim->r,REG_ESP,reg_ESP_val+0x4);
	return TRUE;
}

mem_t *init_mem(int len)
{
	mem_t *m = (mem_t *)malloc(sizeof(mem_t));
	len = ((len+BLK_SIZE-1)/BLK_SIZE)*BLK_SIZE;
	m->len = len;
	m->data = (byte_t *)calloc(len, 1);

	return m;
}

void free_mem(mem_t *m)
{
	free((void *) m->data);
	free((void *) m);
}

mem_t *dup_mem(mem_t *oldm)
{
	mem_t *newm = init_mem(oldm->len);
	memcpy(newm->data, oldm->data, oldm->len);
	return newm;
}

bool_t diff_mem(mem_t *oldm, mem_t *newm, FILE *outfile)
{
	long_t pos;
	int len = oldm->len;
	bool_t diff = FALSE;

	if (newm->len < len)
		len = newm->len;
```

```
    for (pos = 0; (!diff || outfile) && pos < len; pos += 4) {
        long_t ov = 0;    long_t nv = 0;
        get_long_val(oldm, pos, &ov);
        get_long_val(newm, pos, &nv);
        if (nv != ov) {
            diff = TRUE;
            if (outfile)
                fprintf(outfile, "0x%.4x:\t0x%.8x\t0x%.8x\n", pos, ov, nv);
        }
    }
    return diff;
}


reg_t reg_table[REG_CNT] = {
    {"%eax", REG_EAX},
    {"%ecx", REG_ECX},
    {"%edx", REG_EDX},
    {"%ebx", REG_EBX},
    {"%esp", REG_ESP},
    {"%ebp", REG_EBP},
    {"%esi", REG_ESI},
    {"%edi", REG_EDI},
};

long_t get_reg_val(mem_t *r, regid_t id)
{
    long_t val = 0;
    if (id >= REG_NONE)
        return 0;
    get_long_val(r, id*4, &val);
    return val;
}

void set_reg_val(mem_t *r, regid_t id, long_t val)
{
    if (id < REG_NONE)
        set_long_val(r, id*4, val);
}

mem_t *init_reg()
{
    return init_mem(REG_SIZE);
}
```

```
void free_reg(mem_t *r)
{
    free_mem(r);
}

mem_t *dup_reg(mem_t *oldr)
{
    return dup_mem(oldr);
}

bool_t diff_reg(mem_t *oldr, mem_t *newr, FILE *outfile)
{
    long_t pos;
    int len = oldr->len;
    bool_t diff = FALSE;

    if (newr->len < len)
        len = newr->len;

    for (pos = 0; (!diff || outfile) && pos < len; pos += 4) {
        long_t ov = 0;
        long_t nv = 0;
        get_long_val(oldr, pos, &ov);
        get_long_val(newr, pos, &nv);
        if (nv != ov) {
            diff = TRUE;
            if (outfile)
                fprintf(outfile, "%s:\t0x%.8x\t0x%.8x\n",
                        reg_table[pos/4].name, ov, nv);
        }
    }
    return diff;
}

/* create an y86 image with registers and memory */
y86sim_t *new_y86sim(int slen)
{
    y86sim_t *sim = (y86sim_t*)malloc(sizeof(y86sim_t));
    sim->pc = 0;
    sim->r = init_reg();
    sim->m = init_mem(slen);
    sim->cc = DEFAULT_CC;
    return sim;
```

```
}

void free_y86sim(y86sim_t *sim)
{
        free_reg(sim->r);
        free_mem(sim->m);
        free((void *) sim);
}

/* load binary code and data from file to memory image */
int load_binfile(mem_t *m, FILE *f)
{
        int flen;

        clearerr(f);
        flen = fread(m->data, sizeof(byte_t), m->len, f);
        if (ferror(f)) {
                err_print("fread() failed (0x%x)", flen);
                return -1;
        }
        if (!feof(f)) {
                err_print("too large memory footprint (0x%x)", flen);
                return -1;
        }
        return 0;
}

/*
 * compute_alu: do ALU operations
 * args
 *         op: operations (A_ADD, A_SUB, A_AND, A_XOR)
 *         argA: the first argument
 *         argB: the second argument
 *
 * return
 *         val: the result of operation on argA and argB
 */
long_t compute_alu(alu_t op, long_t argA, long_t argB)
{
        long_t val = 0;
        switch(op){
        case A_ADD:
       val = argA + argB;
       break;
```

```
        case A_SUB:
       val = argB - argA;
       break;
        case A_AND:
       val = argA & argB;
       break;
        case A_XOR:
       val = argA ^ argB;
       break;
        case A_NONE:
       val = 0;
       break;
        }
        return val;
}

/*
  * compute_cc: modify condition codes according to operations
  * args
  *          op: operations (A_ADD, A_SUB, A_AND, A_XOR)
  *          argA: the first argument
  *          argB: the second argument
  *          val: the result of operation on argA and argB
  *
  * return
  *          PACK_CC: the final condition codes
  */
cc_t compute_cc(alu_t op, long_t argA, long_t argB, long_t val)
{
        bool_t zero = FALSE;
        bool_t sign = FALSE;
        bool_t ovf = FALSE;
        val = compute_alu(op,argA,argB);
        if(val == 0)
              zero = TRUE;
        if(val < 0)
              sign = TRUE;
        switch(op){
        case A_ADD:
        {
        if((argA > 0) && (argB > 0) && (val < 0))
       ovf = TRUE;
        if((argA < 0) && (argB < 0) && (val > 0))
              ovf = TRUE;
        else
              ovf = FALSE;
        break;
          }
          case A_SUB:
        {
        if((-argA > 0) && (argB > 0) && (val < 0))
       ovf = TRUE;
        if((-argA < 0) && (argB < 0) && (val > 0))
              ovf = TRUE;
        else
              ovf = FALSE;
        break;
          }
        case A_AND: case A_XOR:
                 break;
        case A_NONE:
              zero = FALSE;
              sign = FALSE;
        break;
        }
        return PACK_CC(zero,sign,ovf);
}


/*
  * cond_doit: whether do (mov or jmp) it?
  * args
  *         PACK_CC: the current condition codes
  *         cond: conditions (C_YES, C_LE, C_L, C_E, C_NE, C_GE, C_G)
  *
  * return
  *         TRUE: do it
  *         FALSE: not do it
  */
bool_t cond_doit(cc_t cc, cond_t cond)
{
        bool_t doit = FALSE;
        switch(cond){
        case C_YES:
       doit = TRUE;
       break;
        case C_LE:
       doit = (GET_SF(cc) ^ GET_OF(cc))| GET_ZF(cc);
```

```
        break;
         case C_L:
        doit = GET_SF(cc) ^ GET_OF(cc);
        break;
         case C_E:
        doit = GET_ZF(cc);
        break;
         case C_NE:
        doit = !GET_ZF(cc);
        break;
         case C_GE:
        doit = !(GET_SF(cc) ^ GET_OF(cc));
        break;
         case C_G:
        doit = !(GET_SF(cc) ^ GET_OF(cc)) & !GET_ZF(cc);
        break;
          }
         return doit;
}

/*
 * nexti: execute single instruction and return status.
 * args
 *       sim: the y86 image with PC, register and memory
 *
 * return
 *       STAT_AOK: continue
 *       STAT_HLT: halt
 *       STAT_ADR: invalid instruction address, data address, stack address, ...
 *       STAT_INS: invalid instruction, register id, ...
 */
stat_t nexti(y86sim_t *sim)
{
        byte_t codefun = 0;
        itype_t icode;
        alu_t ifun;
        long_t next_pc = sim->pc;
        regid_t regA = REG_NONE, regB = REG_NONE;
        long_t imm;

        /* get code and function (1 byte) */
        if (!get_byte_val(sim->m, next_pc, &codefun)) {
                err_print("PC = 0x%x, Invalid instruction address", sim->pc);
                return STAT_ADR;
 }
 icode = GET_ICODE(codefun);
 ifun = GET_FUN(codefun);
 next_pc++;

 /* get registers if needed (1 byte) */
 switch(icode){
 case I_RRMOVL: case I_IRMOVL: case I_RMMOVL:
 case I_MRMOVL: case I_ALU: case I_POPL: case I_PUSHL:
if (!get_byte_val(sim->m, next_pc, &codefun)) {
            err_print("PC = 0x%x, Invalid instruction address", sim->pc);
      return STAT_ADR;
}
regA = GET_REGA(codefun);
regB = GET_REGB(codefun);
next_pc++;
break;
 default:
break;
 }
 /* get immediate if needed (4 bytes) */
 switch(icode){
 case I_IRMOVL: case I_RMMOVL:
 case I_MRMOVL: case I_JMP: case I_CALL:
if (!get_long_val(sim->m, next_pc, &imm)) {
            err_print("PC = 0x%x, Invalid instruction address", sim->pc);
      return STAT_ADR;
}
next_pc += 0x4;
break;
 default:
break;
 }

 /* execute the instruction */
 switch (icode) {
    case I_HALT: /* 0:0 */
        return STAT_HLT;
        break;
    case I_NOP: /* 1:0 */
        sim->pc = next_pc;
        break;

    case I_RRMOVL:    /* 2:x regA:regB */
```

```
          if(!cond_doit(sim->cc,ifun)){
            sim->pc = next_pc;
          break;
           }
          set_reg_val(sim->r,regB,get_reg_val(sim->r,regA));
          sim->pc = next_pc;
          break;

        case I_IRMOVL: /* 3:0 F:regB imm */
      if (regA != 0xF) {
              err_print("PC = 0x%x, Invalid instruction address", sim->pc);
              return STAT_ADR;
           }
          set_reg_val(sim->r,regB,imm);
          sim->pc = next_pc;
          break;

        case I_RMMOVL: /* 4:0 regA:regB imm */
         {
          long_t regB_val = get_reg_val(sim->r,regB);
          if(!set_long_val(sim->m, imm+regB_val, get_reg_val(sim->r,regA))) {
              err_print("PC = 0x%x, Invalid data address
0x%x",sim->pc,regB_val+imm);
          return STAT_ADR;
          }
          sim->pc = next_pc;
          break;
         }

        case I_MRMOVL: /* 5:0 regB:regA imm */
         {
           long_t mval;
           if(!get_long_val(sim->m,get_reg_val(sim->r,regB)+imm,&mval)) {
               err_print("PC = 0x%x, Invalid data address 0x%x",
sim->pc,get_reg_val(sim->r,regB)+imm);
           return STAT_ADR;
            }
           set_reg_val(sim->r,regA,mval);
           sim->pc = next_pc;
           break;
         }

        case I_ALU: /* 6:x regA:regB */
         {
```

```
         long_t regA_val = get_reg_val(sim->r,regA),
                regB_val = get_reg_val(sim->r,regB);
         long_t res = compute_alu(ifun,regA_val,regB_val);
         sim->cc = compute_cc(ifun,regA_val,regB_val,res);
         set_reg_val(sim->r,regB,res);
         sim->pc = next_pc;
         break;
       }

      case I_JMP: /* 7:x imm */
      {
      if(cond_doit(sim->cc,ifun))
          sim->pc = imm;
      else
          sim->pc = next_pc;
      break;
       }

      case I_CALL: /* 8:x imm */
       {
       if(imm < 0 || imm > sim->m->len) {
            err_print("WTFatCALL %x",sim->pc);
       return STAT_ADR;
        }
       else{
        sim->pc = imm;
        if(!push_long_val(sim,next_pc)) {
            err_print("PC = 0x%x, Invalid stack address
0x%x",sim->pc,get_reg_val(sim->r,REG_ESP));
        return STAT_ADR;
         }
        }
        break;
       }

      case I_RET: /* 9:0 */
      {
      if(!pop_long_val(sim,&(sim->pc))) {
         err_print("PC = 0x%x, Invalid stack address 0x%x",
sim->pc,get_reg_val(sim->r,REG_ESP));
      return STAT_ADR;
       }
      break;
      }
      case I_PUSHL: /* A:0 regA:F */
      {
      if(!push_long_val(sim,get_reg_val(sim->r,regA))) {
            err_print("PC = 0x%x, Invalid stack address 0x%x",
sim->pc,get_reg_val(sim->r,REG_ESP));
      return STAT_ADR;
       }
      sim->pc = next_pc;
      break;
      }

      case I_POPL: /* B:0 regA:F */
       {
          sim->pc = next_pc;
          if(regB != 0xF) {
              err_print("PC = 0x%x, Invalid instruction %.2x", sim->pc, codefun);
              return STAT_INS;
          }
          long_t val;
          if(!pop_long_val(sim,&val)) {
              err_print("PC = 0x%x, Invalid stack address 0x%x",
                  sim->pc,get_reg_val(sim->r,REG_ESP));
              return STAT_ADR;
          }
          set_reg_val(sim->r,regA,val);
          break;
      }
      default:
        err_print("PC = 0x%x, Invalid instruction %.2x", sim->pc, codefun);
        return STAT_INS;
    }

    return STAT_AOK;
}

void usage(char *pname)
{
    printf("Usage: %s file.bin [max_steps]\n", pname);
    exit(0);
}

int main(int argc, char *argv[])
{
```

```
FILE *binfile;
int max_steps = MAX_STEP;
y86sim_t *sim;
mem_t *saver, *savem;
int step = 0;
stat_t e = STAT_AOK;

if (argc < 2 || argc > 3)
    usage(argv[0]);

/* set max steps */
if (argc > 2)
    max_steps = atoi(argv[2]);

/* load binary file to memory */
if (strcmp(argv[1]+(strlen(argv[1])-4), ".bin"))
    usage(argv[0]); /* only support *.bin file */

binfile = fopen(argv[1], "rb");
if (!binfile) {
    err_print("Can't open binary file '%s'", argv[1]);
    exit(1);
}

sim = new_y86sim(MEM_SIZE);
if (load_binfile(sim->m, binfile) < 0) {
    err_print("Failed to load binary file '%s'", argv[1]);
    free_y86sim(sim);
    exit(1);
}
fclose(binfile);

/* save initial register and memory stat */
saver = dup_reg(sim->r);
savem = dup_mem(sim->m);

/* execute binary code step-by-step */
for (step = 0; step < max_steps && e == STAT_AOK; step++)
    e = nexti(sim);

/* print final stat of y86sim */
printf("Stopped in %d steps at PC = 0x%x.   Status '%s', CC %s\n",
          step, sim->pc, stat_name(e), cc_name(sim->cc));
    printf("Changes to registers:\n");
    diff_reg(saver, sim->r, stdout);

    printf("\nChanges to memory:\n");
    diff_mem(savem, sim->m, stdout);

    free_y86sim(sim);
    free_reg(saver);
    free_mem(savem);

    return 0;
}
```