```c
long_t compute_alu(alu_t op, long_t argA, long_t argB)
{
    switch (op) {
        case A_ADD:
          return argB + argA;
        case A_SUB:
          return argB - argA;
        case A_AND:
          return argB & argA;
        case A_XOR:
          return argB ^ argA;
        case A_NONE: /* act as default */
        default:
          return 0;
    }
}

/*
 * 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;

    zero = val == 0;
    sign = val < 0;
    switch (op) {
        case A_ADD:
            if (argA > 0 && argB > 0)
                ovf = val < 0;
            else if (argA < 0 && argB < 0)
                ovf = val >= 0;
            else
                ovf = 0;
            break;
        case A_SUB:
            if (-argA > 0 && argB > 0)
                ovf = val < 0;
            else if (-argA < 0 && argB < 0)
                ovf = val >= 0;
            else
                ovf = 0;
            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)
{
    switch (cond) {
        case C_YES:
          return TRUE;
        case C_LE:
          return (GET_SF(cc) ^ GET_OF(cc))| GET_ZF(cc);
        case C_L:
          return GET_SF(cc) ^ GET_OF(cc);
          break;
        case C_E:
          return GET_ZF(cc);
        case C_NE:
          return !GET_ZF(cc);
        case C_GE:
          return !(GET_SF(cc) ^ GET_OF(cc));
        case C_G:
          return !(GET_SF(cc) ^ GET_OF(cc)) & !GET_ZF(cc);
        default:
          return FALSE;
    }
}

/*
 * 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
 *     STAT_INS: invalid instruction, register id, data address, stack address, ...
 */
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 */
        sim->pc = next_pc;
        if(!cond_doit(sim->cc,ifun))
            break;
        set_reg_val(sim->r,regB,get_reg_val(sim->r,regA));
        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_cont = get_reg_val(sim->r,regB);
          if(!set_long_val(sim->m, imm+regB_cont, get_reg_val(sim->r,regA))) {
              err_print("PC = 0x%x, Invalid data address 0x%x",
                      sim->pc,regB_cont+imm);
          return STAT_ADR;
          }
          sim->pc = next_pc;
          break;
      }

      case I_MRMOVL: /* 5:0 regB:regA imm */
      {
          long_t cont;
          if(!get_long_val(sim->m,get_reg_val(sim->r,regB)+imm,&cont)) {
              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,cont);
          sim->pc = next_pc;
          break;
      }

      case I_ALU: /* 6:x regA:regB */
      {
          long_t regA_cont = get_reg_val(sim->r,regA),
                 regB_cont = get_reg_val(sim->r,regB);
          long_t res = compute_alu(ifun,regA_cont,regB_cont);
          sim->cc = compute_cc(ifun,regA_cont,regB_cont,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;
        }
```

```c
            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;
}
```