```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#include "y86asm.h"

line_t *y86bin_listhead = NULL;      /* the head of y86 binary code line list*/
line_t *y86bin_listtail = NULL;      /* the tail of y86 binary code line list*/
int y86asm_lineno = 0; /* the current line number of y86 assemble code */

#define err_print(_s, _a ...) do { \
   if (y86asm_lineno < 0) \
      fprintf(stderr, "[--]: "_s"\n", ## _a); \
   else \
      fprintf(stderr, "[L%d]: "_s"\n", y86asm_lineno, ## _a); \
} while (0);

int vmaddr = 0;        /* vm addr */

/* register table */
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},
};

regid_t find_register(char *name)
{
       int i;
       for (i = 0; i < REG_CNT; i++)
              if (!strncmp(name, reg_table[i].name, 4))
                     return reg_table[i].id;
       return REG_ERR;
}

/* instruction set */
instr_t instr_set[] = {
       {"nop", 3,      HPACK(I_NOP, F_NONE), 1 },
```

```
        {"halt", 4,    HPACK(I_HALT, F_NONE), 1 },
        {"rrmovl", 6,HPACK(I_RRMOVL, F_NONE), 2 },
        {"cmovle", 6,HPACK(I_RRMOVL, C_LE), 2 },
        {"cmovl", 5, HPACK(I_RRMOVL, C_L), 2 },
        {"cmove", 5, HPACK(I_RRMOVL, C_E), 2 },
        {"cmovne", 6,HPACK(I_RRMOVL, C_NE), 2 },
        {"cmovge", 6,HPACK(I_RRMOVL, C_GE), 2 },
        {"cmovg", 5, HPACK(I_RRMOVL, C_G), 2 },
        {"irmovl", 6,HPACK(I_IRMOVL, F_NONE), 6 },
        {"rmmovl", 6,HPACK(I_RMMOVL, F_NONE), 6 },
        {"mrmovl", 6,HPACK(I_MRMOVL, F_NONE), 6 },
        {"addl", 4,    HPACK(I_ALU, A_ADD), 2 },
        {"subl", 4,    HPACK(I_ALU, A_SUB), 2 },
        {"andl", 4,    HPACK(I_ALU, A_AND), 2 },
        {"xorl", 4,    HPACK(I_ALU, A_XOR), 2 },
        {"jmp", 3,      HPACK(I_JMP, C_YES), 5 },
        {"jle", 3,      HPACK(I_JMP, C_LE), 5 },
        {"jl", 2,       HPACK(I_JMP, C_L), 5 },
        {"je", 2,       HPACK(I_JMP, C_E), 5 },
        {"jne", 3,      HPACK(I_JMP, C_NE), 5 },
        {"jge", 3,      HPACK(I_JMP, C_GE), 5 },
        {"jg", 2,       HPACK(I_JMP, C_G), 5 },
        {"call", 4,    HPACK(I_CALL, F_NONE), 5 },
        {"ret", 3,      HPACK(I_RET, F_NONE), 1 },
        {"pushl", 5, HPACK(I_PUSHL, F_NONE), 2 },
        {"popl", 4,    HPACK(I_POPL, F_NONE),    2 },
        {".byte", 5, HPACK(I_DIRECTIVE, D_DATA), 1 },
        {".word", 5, HPACK(I_DIRECTIVE, D_DATA), 2 },
        {".long", 5, HPACK(I_DIRECTIVE, D_DATA), 4 },
        {".pos", 4,    HPACK(I_DIRECTIVE, D_POS), 0 },
        {".align", 6,HPACK(I_DIRECTIVE, D_ALIGN), 0 },
        {NULL, 1,       0      , 0 } //end
};

instr_t *find_instr(char *name)
{
       int i;
       for (i = 0; instr_set[i].name; i++)
      if (strncmp(instr_set[i].name, name, instr_set[i].len) == 0)
            return &instr_set[i];
       return NULL;
}

/* symbol table (don't forget to init and finit it) */
```

```
symbol_t *symtab = NULL;

/*
 * find_symbol: scan table to find the symbol
 * args
 *          name: the name of symbol
 *
 * return
 *          symbol_t: the 'name' symbol
 *          NULL: not exist
 */
symbol_t *find_symbol(char *name)
{
        return NULL;
}

/*
 * add_symbol: add a new symbol to the symbol table
 * args
 *          name: the name of symbol
 *
 * return
 *          0: success
 *          -1: error, the symbol has exist
 */
int add_symbol(char *name)
{
        /* check duplicate */

        /* create new symbol_t (don't forget to free it)*/

        /* add the new symbol_t to symbol table */

        return 0;
}

/* relocation table (don't forget to init and finit it) */
reloc_t *reltab = NULL;

/*
 * add_reloc: add a new relocation to the relocation table
 * args
 *          name: the name of symbol
 *
```

```
 * return
 *         0: success
 *         -1: error, the symbol has exist
 */
void add_reloc(char *name, bin_t *bin)
{
      /* create new reloc_t (don't forget to free it)*/

      /* add the new reloc_t to relocation table */
}


/* macro for parsing y86 assembly code */
#define IS_DIGIT(s) ((*(s)>='0' && *(s)<='9') || *(s)=='-' || *(s)=='+')
#define IS_LETTER(s) ((*(s)>='a' && *(s)<='z') || (*(s)>='A' && *(s)<='Z'))
#define IS_COMMENT(s) (*(s)=='#')
#define IS_REG(s) (*(s)=='%')
#define IS_IMM(s) (*(s)=='$')

#define IS_BLANK(s) (*(s)==' ' || *(s)=='\t')
#define IS_END(s) (*(s)=='\0')

#define SKIP_BLANK(s) do {    \
   while(!IS_END(s) && IS_BLANK(s))    \
      (s)++;       \
} while(0);

/* return value from different parse_xxx function */
typedef enum { PARSE_ERR=-1, PARSE_REG, PARSE_DIGIT, PARSE_SYMBOL,
      PARSE_MEM, PARSE_DELIM, PARSE_INSTR, PARSE_LABEL} parse_t;

/*
 * parse_instr: parse an expected data token (e.g., 'rrmovl')
 * args
 *         ptr: point to the start of string
 *         inst: point to the inst_t within instr_set
 *
 * return
 *         PARSE_INSTR: success, move 'ptr' to the first char after token,
 *                                         and store the pointer of the instruction to 'inst'
 *         PARSE_ERR: error, the value of 'ptr' and 'inst' are undefined
 */
parse_t parse_instr(char **ptr, instr_t **inst)
{
```

```
    char *cur = *ptr;
    instr_t *tmp;

    /* skip the blank */
    SKIP_BLANK(cur);
    if (IS_END(cur))
        return PARSE_ERR;

    /* find_instr and check end */
    tmp = find_instr(cur);
    if (tmp == NULL)
        return PARSE_ERR;

    cur += tmp->len;
    if (!IS_END(cur) && !IS_BLANK(cur))
        return PARSE_ERR;

    /* set 'ptr' and 'inst' */
    *inst = tmp;
    *ptr = cur;
    return PARSE_INSTR;
}

/*
 * parse_delim: parse an expected delimiter token (e.g., ',')
 * args
 *      ptr: point to the start of string
 *
 * return
 *      PARSE_DELIM: success, move 'ptr' to the first char after token
 *      PARSE_ERR: error, the value of 'ptr' and 'delim' are undefined
 */
parse_t parse_delim(char **ptr, char delim)
{
    /* skip the blank and check */

    /* set 'ptr' */

    return PARSE_ERR;
}

/*
 * parse_reg: parse an expected register token (e.g., '%eax')
 * args
```

```
 *          ptr: point to the start of string
 *          regid: point to the regid of register
 *
 * return
 *          PARSE_REG: success, move 'ptr' to the first char after token,
 *                                          and store the regid to 'regid'
 *          PARSE_ERR: error, the value of 'ptr' and 'regid' are undefined
 */
parse_t parse_reg(char **ptr, regid_t *regid)
{
        /* skip the blank and check */

        /* find register */

        /* set 'ptr' and 'regid' */

        return PARSE_ERR;
}

/*
 * parse_symbol: parse an expected symbol token (e.g., 'Main')
 * args
 *          ptr: point to the start of string
 *          name: point to the name of symbol (should be allocated in this function)
 *
 * return
 *          PARSE_SYMBOL: success, move 'ptr' to the first char after token,
 *                                                  and allocate and store name to 'name'
 *          PARSE_ERR: error, the value of 'ptr' and 'name' are undefined
 */
parse_t parse_symbol(char **ptr, char **name)
{
        /* skip the blank and check */

        /* allocate name and copy to it */

        /* set 'ptr' and 'name' */

        return PARSE_ERR;
}

/*
 * parse_digit: parse an expected digit token (e.g., '0x100')
 * args
```

```
 *          ptr: point to the start of string
 *          value: point to the value of digit
 *
 * return
 *          PARSE_DIGIT: success, move 'ptr' to the first char after token
 *                                          and store the value of digit to 'value'
 *          PARSE_ERR: error, the value of 'ptr' and 'value' are undefined
 */
parse_t parse_digit(char **ptr, long *value)
{
        /* skip the blank and check */

        /* calculate the digit, (NOTE: see strtoll()) */

        /* set 'ptr' and 'value' */

        return PARSE_ERR;
}

/*
 * parse_imm: parse an expected immediate token (e.g., '$0x100' or 'STACK')
 * args
 *          ptr: point to the start of string
 *          name: point to the name of symbol (should be allocated in this function)
 *          value: point to the value of digit
 *
 * return
 *          PARSE_DIGIT: success, the immediate token is a digit,
 *                                          move 'ptr' to the first char after token,
 *                                          and store the value of digit to 'value'
 *          PARSE_SYMBOL: success, the immediate token is a symbol,
 *                                          move 'ptr' to the first char after token,
 *                                          and allocate and store name to 'name'
 *          PARSE_ERR: error, the value of 'ptr', 'name' and 'value' are undefined
 */
parse_t parse_imm(char **ptr, char **name, long *value)
{
        /* skip the blank and check */

        /* if IS_IMM, then parse the digit */

        /* if IS_LETTER, then parse the symbol */

        /* set 'ptr' and 'name' or 'value' */
```

```
    return PARSE_ERR;
}

/*
 * parse_mem: parse an expected memory token (e.g., '8(%ebp)')
 * args
 *      ptr: point to the start of string
 *      value: point to the value of digit
 *      regid: point to the regid of register
 *
 * return
 *      PARSE_MEM: success, move 'ptr' to the first char after token,
 *                                  and store the value of digit to 'value',
 *                                  and store the regid to 'regid'
 *      PARSE_ERR: error, the value of 'ptr', 'value' and 'regid' are undefined
 */
parse_t parse_mem(char **ptr, long *value, regid_t *regid)
{
    /* skip the blank and check */

    /* calculate the digit and register, (ex: (%ebp) or 8(%ebp)) */

    /* set 'ptr', 'value' and 'regid' */

    return PARSE_ERR;
}

/*
 * parse_data: parse an expected data token (e.g., '0x100' or 'array')
 * args
 *      ptr: point to the start of string
 *      name: point to the name of symbol (should be allocated in this function)
 *      value: point to the value of digit
 *
 * return
 *      PARSE_DIGIT: success, data token is a digit,
 *                                  and move 'ptr' to the first char after token,
 *                                  and store the value of digit to 'value'
 *      PARSE_SYMBOL: success, data token is a symbol,
 *                                  and move 'ptr' to the first char after token,
 *                                  and allocate and store name to 'name'
 *      PARSE_ERR: error, the value of 'ptr', 'name' and 'value' are undefined
 */
```

```
parse_t parse_data(char **ptr, char **name, long *value)
{
    /* skip the blank and check */

    /* if IS_DIGIT, then parse the digit */

    /* if IS_LETTER, then parse the symbol */

    /* set 'ptr', 'name' and 'value' */

    return PARSE_ERR;
}

/*
 * parse_label: parse an expected label token (e.g., 'Loop:')
 * args
 *      ptr: point to the start of string
 *      name: point to the name of symbol (should be allocated in this function)
 *
 * return
 *      PARSE_LABEL: success, move 'ptr' to the first char after token
 *                                      and allocate and store name to 'name'
 *      PARSE_ERR: error, the value of 'ptr' is undefined
 */
parse_t parse_label(char **ptr, char **name)
{
    /* skip the blank and check */

    /* allocate name and copy to it */

    /* set 'ptr' and 'name' */

    return PARSE_ERR;
}

/*
 * parse_line: parse a line of y86 code (e.g., 'Loop: mrmovl (%ecx), %esi')
 * (you could combine above parse_xxx functions to do it)
 * args
 *      line: point to a line_t data with a line of y86 assembly code
 *
 * return
 *      PARSE_XXX: success, fill line_t with assembled y86 code
 *        PARSE_ERR: error, try to print err information (e.g., instr type and line
```

```
number)
 */
type_t parse_line(line_t *line)
{
	bin_t *y86bin;
	char * y86asm;
	char *label = NULL;
	instr_t *inst = NULL;

	char *cur;
	int ret;

	y86bin = &line->y86bin;
	y86asm = (char *)
		malloc(sizeof(char) * (strlen(line->y86asm) + 1));
	strcpy(y86asm, line->y86asm);
	cur = y86asm;

/* when finish parse an instruction or lable, we still need to continue check
* e.g.,
*    Loop: mrmovl (%ebp), %ecx
*                  call SUM    #invoke SUM function */
cont:

	/* skip blank and check IS_END */
	SKIP_BLANK(cur);
	if (IS_END(cur))
		goto out; /* done */

	/* is a comment ? */
	if (IS_COMMENT(cur)) {
		goto out; /* skip rest */
	}


	/* is a label ? */
	ret = parse_label(&cur, &label);
	if (ret == PARSE_LABEL) {
		/* add new symbol */
		if (add_symbol(label) < 0) {
			line->type = TYPE_ERR;
			err_print("Dup symbol:%s", label);
			goto out;
		}
```

```
        /* set type and y86bin */
        line->type = TYPE_INS;
        line->y86bin.addr = vmaddr;

        /* continue */
        goto cont;
}

/* is an instruction ? */
ret = parse_instr(&cur, &inst);
if (ret == PARSE_ERR) {
        line->type = TYPE_ERR;
        err_print("Invalid instr");
        goto out;
}

/* set type and y86bin */
line->type = TYPE_INS;
y86bin->addr = vmaddr;
y86bin->codes[0] = inst->code;
y86bin->bytes = inst->bytes;

/* update vmaddr */
vmaddr += inst->bytes;

/* parse the rest of instruction according to the itype */
switch (HIGH(inst->code)) {
   /* further partition the y86 instructions according to the format */
   case I_HALT:    /* 0:0 - e.g., halt */
   case I_NOP:     /* 1:0 - e.g., nop */
   case I_RET: { /* 9:0 - e.g., ret" */
      goto cont;
   }

   case I_PUSHL: /* A:0 regA:F - e.g., pushl %esp */
   case I_POPL: {/* B:0 regA:F - e.g., popl %ebp */
      /* parse register */

      /* set y86bin codes */

      goto cont;
   }
```

```
        case I_RRMOVL:/* 2:x regA,regB - e.g., rrmovl %esp, %ebp */
        case I_ALU: { /* 6:x regA,regB - e.g., xorl %eax, %eax */
           goto cont;
        }

        case I_IRMOVL: {    /* 3:0 Imm, regB - e.g., irmovl $-1, %ebx */
           goto cont;
        }

        case I_RMMOVL: {     /* 4:0 regA, D(regB) - e.g., rmmovl %eax, 8(%esp)
*/
           goto cont;
        }

        case I_MRMOVL: {    /* 5:0 D(regB), regA - e.g., mrmovl 8(%ebp), %ecx */
           goto cont;
        }

        case I_JMP:      /* 7:x dest - e.g., je End */
        case I_CALL: {/* 8:x dest - e.g., call Main */
           goto cont;
        }

        case I_DIRECTIVE: {
           /* further partition directive according to dtv_t */
           switch (LOW(inst->code)) {
              case D_DATA: {       /* .long data - e.g., .long 0xC0 */
                 goto cont;
              }

              case D_POS: {      /* .pos D - e.g., .pos 0x100 */
                 goto cont;
              }

              case D_ALIGN: {      /* .align D - e.g., .align 4 */
                 goto cont;
              }
              default:
                 line->type = TYPE_ERR;
                 err_print("Unknown directive");
                 goto out;
           }
           break;
        }
```

```
            default:
                line->type = TYPE_ERR;
                err_print("Unknown instr");
                goto out;
        }

out:
        free(y86asm);
        return line->type;
}

/*
 * assemble: assemble an y86 file (e.g., 'asum.ys')
 * args
 *          in: point to input file (an y86 assembly file)
 *
 * return
 *          0: success, assmble the y86 file to a list of line_t
 *          -1: error, try to print err information (e.g., instr type and line number)
 */
int assemble(FILE *in)
{
        static char asm_buf[MAX_INSLEN]; /* the current line of asm code */
        line_t *line;
        int slen;
        char *y86asm;

        /* read y86 code line-by-line, and parse them to generate raw y86 binary code
list */
        while (fgets(asm_buf, MAX_INSLEN, in) != NULL) {
                slen    = strlen(asm_buf);
                if ((asm_buf[slen-1] == '\n') || (asm_buf[slen-1] == '\r')) {
                        asm_buf[--slen] = '\0'; /* replace terminator */
                }

                /* store y86 assembly code */
                y86asm = (char *)malloc(sizeof(char) * (slen + 1)); // free in finit
                strcpy(y86asm, asm_buf);

                line = (line_t *)malloc(sizeof(line_t)); // free in finit
                memset(line, '\0', sizeof(line_t));

                /* set defualt */
                line->type = TYPE_COMM;
```

```
            line->y86asm = y86asm;
            line->next = NULL;

            /* add to y86 binary code list */
            y86bin_listtail->next = line;
            y86bin_listtail = line;
            y86asm_lineno ++;

            /* parse */
            if (parse_line(line) == TYPE_ERR)
                  return -1;
      }

      /* skip line number information in err_print() */
      y86asm_lineno = -1;
      return 0;
}

/*
 * relocate: relocate the raw y86 binary code with symbol address
 *
 * return
 *       0: success
 *       -1: error, try to print err information (e.g., addr and symbol)
 */
int relocate(void)
{
      reloc_t *rtmp = NULL;

      rtmp = reltab->next;
      while (rtmp) {
            /* find symbol */

            /* relocate y86bin according itype */

            /* next */
            rtmp = rtmp->next;
      }
      return 0;
}

/*
 * binfile: generate the y86 binary file
 * args
```

```
 *          out: point to output file (an y86 binary file)
 *
 * return
 *          0: success
 *          -1: error
 */
int binfile(FILE *out)
{
        /* prepare image with y86 binary code */

        /* binary write y86 code to output file (NOTE: see fwrite()) */

        return 0;
}


/* whether print the readable output to screen or not ? */
bool_t screen = FALSE;

static void hexstuff(char *dest, int value, int len)
{
        int i;
        for (i = 0; i < len; i++) {
                char c;
                int h = (value >> 4*i) & 0xF;
                c = h < 10 ? h + '0' : h - 10 + 'a';
                dest[len-i-1] = c;
        }
}

void print_line(line_t *line)
{
        char buf[26];

        /* line format: 0xHHH: cccccccccccc | <line> */
        if (line->type == TYPE_INS) {
                bin_t *y86bin = &line->y86bin;
                int i;

                strcpy(buf, "    0x000:                          | ");

                hexstuff(buf+4, y86bin->addr, 3);
                if (y86bin->bytes > 0)
                        for (i = 0; i < y86bin->bytes; i++)
```

```
                        hexstuff(buf+9+2*i, y86bin->codes[i]&0xFF, 2);
        } else {
                strcpy(buf, "                                    | ");
        }

        printf("%s%s\n", buf, line->y86asm);
}

/*
 * print_screen: dump readable binary and assembly code to screen
 * (e.g., Figure 4.8 in ICS book)
 */
void print_screen(void)
{
        line_t *tmp = y86bin_listhead->next;

        /* line by line */
        while (tmp != NULL) {
                print_line(tmp);
                tmp = tmp->next;
        }
}

/* init and finit */
void init(void)
{
        reltab = (reloc_t *)malloc(sizeof(reloc_t)); // free in finit
        memset(reltab, 0, sizeof(reloc_t));

        symtab = (symbol_t *)malloc(sizeof(symbol_t)); // free in finit
        memset(symtab, 0, sizeof(symbol_t));

        y86bin_listhead = (line_t *)malloc(sizeof(line_t)); // free in finit
        memset(y86bin_listhead, 0, sizeof(line_t));
        y86bin_listtail = y86bin_listhead;
        y86asm_lineno = 0;
}

void finit(void)
{
        reloc_t *rtmp = NULL;
        do {
                rtmp = reltab->next;
                if (reltab->name)
```

```
                free(reltab->name);
            free(reltab);
            reltab = rtmp;
        } while (reltab);

        symbol_t *stmp = NULL;
        do {
            stmp = symtab->next;
            if (symtab->name)
                free(symtab->name);
            free(symtab);
            symtab = stmp;
        } while (symtab);

        line_t *ltmp = NULL;
        do {
            ltmp = y86bin_listhead->next;
            if (y86bin_listhead->y86asm)
                free(y86bin_listhead->y86asm);
            free(y86bin_listhead);
            y86bin_listhead = ltmp;
        } while (y86bin_listhead);
}

static void usage(char *pname)
{
        printf("Usage: %s [-v] file.ys\n", pname);
        printf("      -v print the readable output to screen\n");
        exit(0);
}

int main(int argc, char *argv[])
{
        int rootlen;
        char infname[512];
        char outfname[512];
        int nextarg = 1;
        FILE *in = NULL, *out = NULL;

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

        if (argv[nextarg][0] == '-') {
            char flag = argv[nextarg][1];
```

```
        switch (flag) {
            case 'v':
                screen = TRUE;
                nextarg++;
                break;
            default:
                usage(argv[0]);
        }
}

/* parse input file name */
rootlen = strlen(argv[nextarg])-3;
/* only support the .ys file */
if (strcmp(argv[nextarg]+rootlen, ".ys"))
        usage(argv[0]);

if (rootlen > 500) {
        err_print("File name too long");
        exit(1);
}


/* init */
init();


/* assemble .ys file */
strncpy(infname, argv[nextarg], rootlen);
strcpy(infname+rootlen, ".ys");
in = fopen(infname, "r");
if (!in) {
        err_print("Can't open input file '%s'", infname);
        exit(1);
}

if (assemble(in) < 0) {
        err_print("Assemble y86 code error");
        fclose(in);
        exit(1);
}
fclose(in);


/* relocate binary code */
```

```
    if (relocate() < 0) {
        err_print("Relocate binary code error");
        exit(1);
    }


    /* generate .bin file */
    strncpy(outfname, argv[nextarg], rootlen);
    strcpy(outfname+rootlen, ".bin");
    out = fopen(outfname, "wb");
    if (!out) {
        err_print("Can't open output file '%s'", outfname);
        exit(1);
    }

    if (binfile(out) < 0) {
        err_print("Generate binary file error");
        fclose(out);
        exit(1);
    }
    fclose(out);

    /* print to screen (.yo file) */
    if (screen)
        print_screen();

    /* finit */
    finit();
    return 0;
}
```