/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* \*/

/\* RISCV-LC Assembler \*/

/\* \*/

/\* CEG3420 Lab2 \*/

/\* cbai@cse.cuhk.edu.hk \*/

/\* The Chinese University of Hong Kong \*/

/\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "asm.h"

int reg\_to\_num(char \*reg, int line\_no) {

if (is\_reg(reg) == MISMATCH)

handle\_err(7, line\_no);

size\_t len\_of\_regs\_code = sizeof(arch\_regs\_encode) / sizeof(arch\_regs\_encode[0]);

for (int i = 0; i < len\_of\_regs\_code; i++) {

if (cmp\_str(arch\_regs[i], reg))

return arch\_regs\_encode[i];

}

}

int is\_imm\_in\_range(int val, int width) {

/\*

\* given a bit width, check whether it can hold the signed value

\*/

// return (val <= ((1 << width) - 1) && val >= -(1 << width));

return val <= ((1 << (width - 1)) - 1) && (val >= (-(1 << (width - 1))));

}

int handle\_label\_or\_imm(int line\_no, char \*val, struct\_label\_table \*label\_table, int number\_of\_labels) {

int \_val, ret;

if ((ret = is\_imm(val)) == MISMATCH) {

// label

return label\_to\_num(line\_no, val, 20, label\_table, number\_of\_labels);

} else {

// immediate value

\_val = (int)strtol(val, NULL, (ret == HEX) ? 16 : 10);

if (is\_imm\_in\_range(\_val, 20))

return \_val;

else

handle\_err(5, line\_no);

}

}

int validate\_imm(char \*imm, int width, int line\_no) {

/\*

\* do the base conversion or given a bit width,

\* validate whether it can hold the immediate value

\*/

int val, ret;

if ((ret = is\_imm(imm)) == MISMATCH)

handle\_err(7, line\_no);

else

val = (int)strtol(imm, NULL, (ret == HEX) ? 16 : 10);

if (width != -1) {

if (is\_imm\_in\_range(val, width))

return val;

else

handle\_err(3, line\_no);

} else

return val;

}

int lower5bit(char \*imm, int line\_no) {

int val = 0, ret;

if ((ret = is\_imm(imm)) == MISMATCH)

handle\_err(7, line\_no);

else

val = ((int)strtol(imm, NULL, (ret == HEX) ? 16 : 10) & 0x1F);

if (is\_imm\_in\_range(val, 5))

return val;

else

handle\_err(5, line\_no);

}

int label\_to\_num(

int line\_no,

char \*label,

int width,

struct\_label\_table \*label\_table,

int number\_of\_labels

) {

if (is\_label(label) != VALID\_LABEL)

handle\_err(10, line\_no);

for (int i = 0; i < number\_of\_labels; i++) {

if(cmp\_str(label\_table[i].label, label)) {

int address = label\_table[i].address;

if (!is\_imm\_in\_range(address, width))

handle\_err(8, line\_no);

return address;

}

}

handle\_err(1, line\_no);

}

struct\_regs\_indirect\_addr\* parse\_regs\_indirect\_addr(char \*arg2, int line\_no) {

struct\_regs\_indirect\_addr\* ret;

char \*reg, \*imm;

char temp[MAX\_LINE\_LENGTH + 1];

int i = 0;

if ((ret = (struct\_regs\_indirect\_addr \*)malloc(sizeof(ret))) == NULL)

handle\_err(11, line\_no);

while (arg2[i] != '\0') {

temp[i] = arg2[i];

i++;

}

temp[i] = '\0';

imm = strtok(temp, "( ");

ret->imm = validate\_imm(imm, 12, line\_no);

reg = strtok(NULL, ")");

copy\_str(ret->reg, reg);

return ret;

}

void fill\_to\_binary(FILE\* output\_file, char \*arg1, int line\_no) {

fprintf(output\_file, "0x%08x\n", validate\_imm(arg1, 31, line\_no));

}

void halt\_to\_binary(FILE \*output\_file) {

/\*

\* HALT is mapped to the fixed binary

\*/

fprintf(output\_file, "0x%08x\n", 0x0000707f);

}

void la\_to\_binary(

FILE \*output\_file,

int line\_no,

char \*opcode,

char \*arg1,

char \*arg2,

char \*arg3,

struct\_label\_table \*label\_table,

int number\_of\_labels,

int addr

) {

/\*

\* LA is translated to lui and addi.

\*/

char temp[MAX\_LINE\_LENGTH + 1];

fprintf(

output\_file,

"0x%08x\n",

inst\_to\_binary(

line\_no, "lui", arg1, arg2, arg3, label\_table, number\_of\_labels, addr

)

);

sprintf(

temp,

"%d",

(handle\_label\_or\_imm(

line\_no,

arg2,

label\_table,

number\_of\_labels

)) & 0xFFF

);

fprintf(

output\_file,

"0x%08x\n",

inst\_to\_binary(

line\_no, "addi", arg1, arg1, temp, label\_table, number\_of\_labels, addr

)

);

}

int inst\_to\_binary(

int line\_no,

char \*opcode,

char \*arg1,

char \*arg2,

char \*arg3,

struct\_label\_table \*label\_table,

int number\_of\_labels,

int addr

) {

/\* save the translated machine binary \*/

int binary = 0;

// Integer Register-Immediate Instructions

if (is\_opcode(opcode) == ADDI) {

binary = (0x04 << 2) + 0x03;

binary += (reg\_to\_num(arg1, line\_no) << 7);

binary += (reg\_to\_num(arg2, line\_no) << 15);

binary += (MASK11\_0(validate\_imm(arg3, 12, line\_no)) << 20);

} else if (is\_opcode(opcode) == SLLI) {

/\* Lab2-1 assignment \*/

warn("Lab2-1 assignment: SLLI instruction\n");

exit(EXIT\_FAILURE);

} else if (is\_opcode(opcode) == XORI) {

/\* Lab2-1 assignment \*/

warn("Lab2-1 assignment: XORI instruction\n");

exit(EXIT\_FAILURE);

} else if (is\_opcode(opcode) == SRLI) {

/\*

\* Lab2-1 assignment

\* tip: you may need the function `lower5bit`

\*/

warn("Lab2-1 assignment: SRLI instruction\n");

exit(EXIT\_FAILURE);

} else if (is\_opcode(opcode) == SRAI) {

/\*

\* Lab2-1 assignment

\* tip: you may need the function `lower5bit`

\*/

warn("Lab2-1 assignment: SRAI instruction\n");

exit(EXIT\_FAILURE);

} else if (is\_opcode(opcode) == ORI) {

/\* Lab2-1 assignment \*/

warn("Lab2-1 assignment: ORI instruction\n");

exit(EXIT\_FAILURE);

} else if (is\_opcode(opcode) == ANDI) {

/\* Lab2-1 assignment \*/

warn("Lab2-1 assignment: ADDI instruction\n");

exit(EXIT\_FAILURE);

} else if (is\_opcode(opcode) == LUI) {

binary = (0x0D << 2) + 0x03;

binary += (reg\_to\_num(arg1, line\_no) << 7);

binary += handle\_label\_or\_imm(

line\_no,

arg2,

label\_table,

number\_of\_labels

) & 0xFFFFF000;

}

// Integer Register-Register Operations

else if (is\_opcode(opcode) == ADD) {

binary = (0x0C << 2) + 0x03;

binary += (reg\_to\_num(arg1, line\_no) << 7);

binary += (reg\_to\_num(arg2, line\_no) << 15);

binary += (reg\_to\_num(arg3, line\_no) << 20);

binary += (0x0 << 25);

} else if (is\_opcode(opcode) == SUB) {

/\* Lab2-1 assignment \*/

warn("Lab2-1 assignment: SUB instruction\n");

exit(EXIT\_FAILURE);

} else if (is\_opcode(opcode) == SLL) {

/\* Lab2-1 assignment \*/

warn("Lab2-1 assignment: SLL instruction\n");

exit(EXIT\_FAILURE);

} else if (is\_opcode(opcode) == XOR) {

/\* Lab2-1 assignment \*/

warn("Lab2-1 assignment: XOR instruction\n");

exit(EXIT\_FAILURE);

} else if (is\_opcode(opcode) == SRL) {

/\* Lab2-1 assignment \*/

warn("Lab2-1 assignment: SRL instruction\n");

exit(EXIT\_FAILURE);

} else if (is\_opcode(opcode) == SRA) {

/\* Lab2-1 assignment \*/

warn("Lab2-1 assignment: SRA instruction\n");

exit(EXIT\_FAILURE);

} else if (is\_opcode(opcode) == OR) {

/\* Lab2-1 assignment \*/

warn("Lab2-1 assignment: OR instruction\n");

exit(EXIT\_FAILURE);

} else if (is\_opcode(opcode) == AND) {

/\* Lab2-1 assignment \*/

warn("Lab2-1 assignment: AND instruction\n");

exit(EXIT\_FAILURE);

}

// Unconditional Jumps

else if (is\_opcode(opcode) == JALR) {

/\*

\* Lab2-1 assignment

\* tip: you may need the function `parse\_regs\_indirect\_addr`

\* e.g., parse\_regs\_indirect\_addr(arg2, line\_no)

\*/

warn("Lab2-1 assignment: JALR instruction\n");

exit(EXIT\_FAILURE);

} else if (is\_opcode(opcode) == JAL) {

/\*

\* Lab2-1 assignment

\* tip: you may need the function `handle\_label\_or\_imm`

\* e.g., handle\_label\_or\_imm(arg2, label\_table, cmd\_no, line\_no)

\*/

warn("Lab2-1 assignment: JAL instruction\n");

exit(EXIT\_FAILURE);

}

// Conditional Branches

else if (is\_opcode(opcode) == BEQ) {

binary = (0x18 << 2) + 0x03;

binary += (reg\_to\_num(arg1, line\_no) << 15);

binary += (reg\_to\_num(arg2, line\_no) << 20);

int val = label\_to\_num(

line\_no, arg3, 12, label\_table, number\_of\_labels

), offset = val - addr;

// imm[11]

binary += ((offset & 0x800) >> 4);

// imm[4:1]

binary += ((offset & 0x1E) << 7);

// imm[10:5]

binary += ((offset & 0x7E0) << 20);

// imm[12]

binary += ((offset & 0x1000) << 19);

} else if (is\_opcode(opcode) == BNE) {

/\* Lab2-1 assignment \*/

warn("Lab2-1 assignment: BNE instruction\n");

exit(EXIT\_FAILURE);

} else if (is\_opcode(opcode) == BLT) {

/\* Lab2-1 assignment \*/

warn("Lab2-1 assignment: BLT instruction\n");

exit(EXIT\_FAILURE);

} else if (is\_opcode(opcode) == BGE) {

/\* Lab2-1 assignment \*/

warn("Lab2-1 assignment: BGE instruction\n");

exit(EXIT\_FAILURE);

}

// Load and Store Instructions

else if (is\_opcode(opcode) == LB) {

binary = 0x03;

binary += (reg\_to\_num(arg1, line\_no) << 7);

struct\_regs\_indirect\_addr\* ret = parse\_regs\_indirect\_addr(arg2, line\_no);

binary += (reg\_to\_num(ret->reg, line\_no) << 15);

binary += (MASK11\_0(ret->imm) << 20);

} else if (is\_opcode(opcode) == LH) {

/\* Lab2-1 assignment \*/

warn("Lab2-1 assignment: LH instruction\n");

exit(EXIT\_FAILURE);

} else if (is\_opcode(opcode) == LW) {

/\* Lab2-1 assignment \*/

warn("Lab2-1 assignment: LW instruction\n");

exit(EXIT\_FAILURE);

} else if (is\_opcode(opcode) == SB) {

/\* Lab2-1 assignment \*/

warn("Lab2-1 assignment: SB instruction\n");

exit(EXIT\_FAILURE);

} else if (is\_opcode(opcode) == SH) {

/\* Lab2-1 assignment \*/

warn("Lab2-1 assignment: SH instruction\n");

exit(EXIT\_FAILURE);

} else if (is\_opcode(opcode) == SW) {

/\* Lab2-1 assignment \*/

warn("Lab2-1 assignment: SW instruction\n");

exit(EXIT\_FAILURE);

}

return binary;

}

int is\_label(char \*ptr) {

if (\*ptr == '\0')

return MISMATCH;

char \*label = ptr;

if (

is\_opcode(label) == MISMATCH && \

is\_reg(ptr) == MISMATCH && \

is\_imm(ptr) == MISMATCH && \

is\_regs\_indirect\_addr(ptr) == MISMATCH

) {

if (isdigit(\*label))

return INVALID\_LABEL;

else {

label++;

while (label != NULL && \*label != '\0') {

/\* all characters have been converted to lower-case \*/

if (isalnum(\*label) || \*label == '\_') {

label++;

continue;

}

else

return INVALID\_LABEL;

}

}

return VALID\_LABEL;

} else

return INVALID\_LABEL;

}

int is\_opcode(char \*ptr) {

if (\*ptr == '\0')

return MISMATCH;

size\_t len\_of\_opcode\_table = sizeof(opcode\_table) / sizeof(opcode\_table[0]);

for (int i = 0; i < (int)len\_of\_opcode\_table; i++) {

if (cmp\_str(opcode\_table[i], ptr)) {

/\* i is equal to the enumeration of opcodes \*/

return i;

}

}

/\* if the opcode is invalid \*/

return MISMATCH;

}

int is\_reg(char \*ptr) {

if (\*ptr == '\0')

return MISMATCH;

size\_t len\_of\_arch\_regs = sizeof(arch\_regs) / sizeof(arch\_regs[0]);

for (int i = 0; i < (int)len\_of\_arch\_regs; i++) {

if (cmp\_str(arch\_regs[i], ptr)) {

/\* it is an architecture register \*/

return ARCH\_REGS;

}

}

return MISMATCH;

}

int is\_imm(char \*ptr) {

/\* -0x7, 0x7, -7, 7 \*/

if (\*ptr == '\0')

return MISMATCH;

int i = 0, l = strlen(ptr);

bool hex = false;

if (l > 3 && ptr[0] == '-' && ptr[1] == '0' && (tolower(ptr[2]) == 'x')) {

// negative hexadecimal

i += 3;

hex = true;

} else if (l > 2 && ptr[0] == '0' && (tolower(ptr[1]) == 'x')) {

// positive hexadecimal

i += 2;

hex = true;

} else if (l > 1 && ptr[0] == '-') {

// negative number

i += 1;

}

while (i < l && isxdigit(ptr[i]))

i++;

if (i == l) {

if (hex)

return HEX;

else

return DEC;

} else

return MISMATCH;

}

int is\_regs\_indirect\_addr(char \*ptr) {

if (\*ptr == '\0')

return MISMATCH;

char \*imm = NULL, \*reg = NULL, \*\_ptr = NULL;

int i = 0;

if (!(imm = (char \*)malloc((strlen(ptr) + 1) \* sizeof(char))))

error("malloc is failed in 'is\_regs\_indirect\_addr'.\n");

if (!(reg = (char \*)malloc((strlen(ptr) + 1) \* sizeof(char))))

error("malloc is failed in 'is\_regs\_indirect\_addr'.\n");

memset(imm, '\0', strlen(ptr));

memset(reg, '\0', strlen(ptr));

\_ptr = ptr;

/\* parse imm. \*/

while (\*\_ptr != '\0' && \*\_ptr != '(') {

if (\*\_ptr != ' ' && \*\_ptr != '\t')

imm[i++] = \*\_ptr;

\_ptr++;

}

imm[i] = '\0';

if (\*\_ptr == '\0') {

free(imm);

free(reg);

return MISMATCH;

}

\_ptr++;

i = 0;

/\* parse reg. \*/

while (\*\_ptr != '\0' && \*\_ptr != ')') {

if (\*\_ptr != ' ' && \*\_ptr != '\t')

reg[i++] = \*\_ptr;

\_ptr++;

}

reg[i] = '\0';

if (is\_imm(imm) != MISMATCH && is\_reg(reg) != MISMATCH) {

free(imm);

free(reg);

return REGS\_INDIRECT\_ADDR;

}

else {

free(imm);

free(reg);

return MISMATCH;

}

}

int is\_arg(char \*ptr) {

if (\*ptr == '\0')

return MISMATCH;

if (is\_label(ptr) == VALID\_LABEL)

return VALID\_LABEL;

if (is\_reg(ptr) != MISMATCH)

return ARCH\_REGS;

else if (is\_imm(ptr) != MISMATCH)

return IMM;

else if (is\_regs\_indirect\_addr(ptr) != MISMATCH)

return REGS\_INDIRECT\_ADDR;

else

return REGS\_INDIRECT\_ADDR;

}

void calc\_la(FILE \*input\_file) {

/\*

\* each pseudo-instruction LA is translated to Lui and addi,

\* i.e., two instructions, so we should count them first to

\* correctly know the offset of other instructions.

\*/

char line[MAX\_LINE\_LENGTH + 1];

char \*ptr;

int line\_cnt = 0, k = 0;

/\* re-read the input file \*/

rewind(input\_file);

while (fgets(line, MAX\_LINE\_LENGTH, input\_file) != NULL) {

for (int i = 0; i < strlen(line); i++)

line[i] = tolower(line[i]);

ptr = line;

while (\*ptr != '#' && \*ptr != '\0' && \*ptr != '\n' && \*ptr != '\r')

ptr++;

\*ptr = '\0';

if (!(ptr = strtok(line, "\t\n ,")))

continue;

if (is\_opcode(ptr) == LA)

la\_inst[line\_cnt] = ++k;

else {

if (!(ptr = strtok(NULL, "\t\n ,")))

continue;

if (is\_opcode(ptr) == LA)

la\_inst[line\_cnt] = ++k;

else

la\_inst[line\_cnt] = k;

}

line\_cnt++;

}

la\_cnt = k;

}

int parse\_inst(

FILE \*input\_file,

char \*\*label,

char \*\*opcode,

char \*\*arg1,

char \*\*arg2,

char \*\*arg3

) {

char \*ptr;

int i;

char \*line, \*\_line;

if (!(line = malloc(sizeof(char) \* (MAX\_LINE\_LENGTH + 1))))

error("malloc is failed in 'parse\_inst'.\n");

if (!fgets(line, MAX\_LINE\_LENGTH, input\_file))

return(END\_OF\_FILE);

/\* convert the entire line to lowercase characters \*/

for (i = 0; i < strlen(line); i++)

line[i] = tolower(line[i]);

/\* make the pointers point to the address of the end, i.e., '\0' \*/

\*label = \*opcode = \*arg1 = \*arg2 = \*arg3 = line + strlen(line);

/\* ignore the comments \*/

ptr = strstr(line, "#");

if (ptr)

\*ptr = '\0';

ptr = line;

while (\*ptr == ' ' || \*ptr == '\n' || \*ptr == '\r')

ptr++;

if (!strlen(ptr))

return(COMMENT\_LINE);

if (!(\_line = (char \*)malloc((strlen(line) + 1) \* sizeof(char))))

error("malloc is failed in 'parse\_inst'.\n");

copy\_str(\_line, line);

/\* skip blank spaces at the beginning of the line \*/

if (!(ptr = strtok(\_line, "\t\n ,")))

return(DONE);

i = 1;

while (strtok(NULL, delimiters))

i++;

switch(i) {

case 1:

/\*

\* 1. halt

\*/

ptr = strtok(line, "\t\n ,");

assert(

cmp\_str(opcode\_table[30], ptr),

"halt is expected in the input asm source code.\n"

);

\*opcode = ptr;

return TYPE1;

case 2:

/\*

\* 1. a label with a halt, e.g., End HALT

\*/

ptr = strtok(line, delimiters);

assert(

is\_label(ptr) == VALID\_LABEL,

"a valid label is expected in the input asm source code.\n"

);

\*label = ptr;

ptr = strtok(NULL, delimiters);

assert(

cmp\_str(opcode\_table[30], ptr),

"halt is expected in the input asm source code.\n"

);

\*opcode = ptr;

return TYPE2;

case 3:

/\*

\* 1. label opcode argument, e.g., a .fill 10

\* 2. opcode, arg1, arg2, e.g., li t0, 1

\*/

ptr = strtok(line, delimiters);

if (is\_label(ptr) == VALID\_LABEL) {

\*label = ptr;

ptr = strtok(NULL, delimiters);

assert(

is\_opcode(ptr) != MISMATCH,

"a valid label is expected in the input asm source code.\n"

);

\*opcode = ptr;

ptr = strtok(NULL, delimiters);

assert(

is\_arg(ptr) != MISMATCH,

"opcode is expected in the input asm source code.\n"

);

\*arg1 = ptr;

return TYPE3;

} else {

assert(

is\_opcode(ptr) != MISMATCH,

"opcode is expected in the input asm source code.\n"

);

\*opcode = ptr;

ptr = strtok(NULL, delimiters);

assert(

is\_arg(ptr) != MISMATCH,

"argument is expected in the input asm source code.\n"

);

\*arg1 = ptr;

ptr = strtok(NULL, delimiters);

assert(

is\_arg(ptr) != MISMATCH,

"argument is expected in the input asm source code.\n"

);

\*arg2 = ptr;

return TYPE4;

}

case 4:

/\*

\* 1. label opcode, arg1, arg2, e.g., label li t0, 1

\* 2. opcode, arg1, arg2, arg3, e.g., addi t0, t0, 1

\*/

ptr = strtok(line, delimiters);

if (is\_label(ptr) == VALID\_LABEL) {

\*label = ptr;

ptr = strtok(NULL, delimiters);

assert(

is\_opcode(ptr) != MISMATCH,

"a valid label is expected in the input asm source code.\n"

);

\*opcode = ptr;

ptr = strtok(NULL, delimiters);

assert(

is\_arg(ptr) != MISMATCH,

"opcode is expected in the input asm source code.\n"

);

\*arg1 = ptr;

ptr = strtok(NULL, delimiters);

assert(

is\_arg(ptr) != MISMATCH,

"argument is expected in the input asm source code.\n"

);

\*arg2 = ptr;

return TYPE5;

} else {

assert(

is\_opcode(ptr) != MISMATCH,

"opcode is expected in the input asm source code.\n"

);

\*opcode = ptr;

ptr = strtok(NULL, delimiters);

assert(

is\_arg(ptr) != MISMATCH,

"opcode is expected in the input asm source code.\n"

);

\*arg1 = ptr;

ptr = strtok(NULL, delimiters);

assert(

is\_arg(ptr) != MISMATCH,

"argument is expected in the input asm source code.\n"

);

\*arg2 = ptr;

ptr = strtok(NULL, delimiters);

assert(

is\_arg(ptr) != MISMATCH,

"argument is expected in the input asm source code.\n"

);

\*arg3 = ptr;

return TYPE6;

}

case 5:

/\*

\* 1. label opcode, arg1, arg2, arg3, e.g., label addi, t0, t0, 1

\*/

ptr = strtok(line, delimiters);

assert(

is\_label(ptr) == VALID\_LABEL,

"a valid label is expected in the input asm source code.\n"

);

\*label = ptr;

ptr = strtok(NULL, delimiters);

assert(

is\_opcode(ptr) != MISMATCH,

"a valid label is expected in the input asm source code.\n"

);

\*opcode = ptr;

ptr = strtok(NULL, delimiters);

assert(

is\_arg(ptr) != MISMATCH,

"opcode is expected in the input asm source code.\n"

);

\*arg1 = ptr;

ptr = strtok(NULL, delimiters);

assert(

is\_arg(ptr) != MISMATCH,

"argument is expected in the input asm source code.\n"

);

\*arg2 = ptr;

ptr = strtok(NULL, delimiters);

assert(

is\_arg(ptr) != MISMATCH,

"argument is expected in the input asm source code.\n"

);

\*arg3 = ptr;

return TYPE7;

default:

error("source codes contain errors.\n");

}

return(DONE);

}

void handle\_err(int err\_no, int line\_no) {

switch (err\_no) {

case 1:

error("<error no: 1> undefeind label in the line %d.\n", line\_no + 1);

exit(1);

case 2:

error("<error no: 2> invalid opcode in the line %d.\n", line\_no + 1);

exit(2);

case 3:

error("<error no: 3> invalid constant in the line %d.\n", line\_no + 1);

exit(3);

case 4:

error("<error no: 4> invalid register operand in the line %d.\n", line\_no + 1);

exit(4);

case 5:

error("<error no: 5> wrong number of operands in the line %d.\n", line\_no + 1);

exit(5);

case 6:

error("<error no: 6> unexpected error occurs in the line %d.\n", line\_no + 1);

exit(6);

case 7:

error("<error no: 7> unexpected operand in the line %d.\n", line\_no + 1);

exit(7);

case 8:

error("<error no: 8> the address of the label in the line %d cannot be fetched.\n", line\_no + 1);

exit(8);

case 9:

error("<error no: 9> the memory address in the line %d is invalid to load the program.\n", line\_no + 1);

exit(9);

case 10:

error("<error no: 10> the label name in line %d is invalid.\n", line\_no + 1);

exit(10);

/\* reserved for other Error Code \*/

case 11:

error("<error no: 11> malloc is failed in the line %d.\n", line\_no + 1);

exit(11);

default:

printf("<error no: 12> unexpected error cannot be handled.\n");

exit(12);

}

}

void initialize\_label\_table(struct\_label\_table\* head) {

for (int i = 0; i < MAX\_LABELS; i++, head++) {

head->label[0] = '\0';

}

}

bool if\_no\_duplicated\_label(struct\_label\_table\* label\_table, char\* label, int line\_no) {

for (int i = 0; i < MAX\_LABELS; i++) {

if (cmp\_str(label\_table[i].label, label)) {

/\* a conflict occurs with duplicated labels \*/

error(

"line: %d, label '%s' has a conflict with other labels in line %d.\n",

line\_no, label, label\_table[i].line\_no

);

}

}

return true;

}

void insert\_a\_label(

struct\_label\_table\* label\_table,

struct\_label\_table\*\* head\_of\_label\_table,

char\* label,

int line\_no,

int addr

) {

if (if\_no\_duplicated\_label(label\_table, label, line\_no)) {

copy\_str((\*head\_of\_label\_table)->label, label);

(\*head\_of\_label\_table)->address = addr;

(\*head\_of\_label\_table)->line\_no = line\_no;

(\*head\_of\_label\_table)++;

}

}

void assemble(char \*input\_file\_name, char \*output\_file\_name) {

FILE \*input\_file = open(input\_file\_name, "r");

FILE \*output\_file = open(output\_file\_name, "w");

char \*label, \*opcode, \*arg1, \*arg2, \*arg3;

struct\_label\_table label\_table[MAX\_LABELS], \*head\_of\_label\_table = label\_table;

int addr = 0x0, line\_no = 0, number\_of\_labels = 0;

int ret;

initialize\_label\_table(head\_of\_label\_table);

info("Processing input file: %s\n", input\_file\_name);

info("Writing result to output file: %s\n", output\_file\_name);

// handle labels

while ((ret = parse\_inst(input\_file, &label, &opcode, &arg1, &arg2, &arg3)) \

!= END\_OF\_FILE

) {

line\_no++;

// `addr += 0x4` is not applied to `COMMENT\_LINE`

if (ret == TYPE1 || ret == TYPE4 || ret == TYPE6) {

// instructions wo. a label

addr += 0x4;

} else if (ret == TYPE2 || ret == TYPE3 || ret == TYPE5 || ret == TYPE7) {

// instructions w. a label

insert\_a\_label(label\_table, &head\_of\_label\_table, label, line\_no, addr);

number\_of\_labels++;

addr += 0x4;

}

if (is\_opcode(opcode) == LA) {

// `la` pseudo instruciton

addr += 0x4;

}

}

#ifdef DEBUG

for (int i = 0; i < number\_of\_labels; i++) {

test(

"label: %s, address: %d, line number: %d\n",

label\_table[i].label,

label\_table[i].address,

label\_table[i].line\_no

);

}

#endif

// assemble the instruction

rewind(input\_file);

addr = line\_no = 0x0;

while ((ret = parse\_inst(input\_file, &label, &opcode, &arg1, &arg2, &arg3)) \

!= END\_OF\_FILE

) {

line\_no++;

if (ret == TYPE1 || ret == TYPE2) {

halt\_to\_binary(output\_file);

addr += 0x4;

} else if (ret == TYPE3) {

fill\_to\_binary(output\_file, arg1, line\_no);

addr += 0x4;

} else if (ret == TYPE4 || ret == TYPE5) {

if (is\_opcode(opcode) != LA) {

fprintf(

output\_file,

"0x%08x\n",

inst\_to\_binary(

line\_no,

opcode,

arg1,

arg2,

arg3,

label\_table,

number\_of\_labels,

addr

)

);

} else {

la\_to\_binary(

output\_file,

line\_no,

opcode,

arg1,

arg2,

arg3,

label\_table,

number\_of\_labels,

addr

);

}

addr += 0x4;

} else if (ret == TYPE6 || ret == TYPE7) {

fprintf(

output\_file,

"0x%08x\n",

inst\_to\_binary(

line\_no,

opcode,

arg1,

arg2,

arg3,

label\_table,

number\_of\_labels,

addr

)

);

addr += 0x4;

}

if (is\_opcode(opcode) == LA) {

// `la` pseudo instruciton

addr += 0x4;

}

}

}

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

if (argc != 3) {

warn("Usage: asm <path to \*.asm> <path to \*.bin>\n");

error("%s: error: lack of input file or output file.\n", argv[0]);

}

assemble(argv[1], argv[2]);

return 0;

}