# Emulator

# Prevodjenje

Koristi se operativni sistem linux, gnu alat I g++ kompajler . Potrebno je u komandnoj linija napisati sledeću komandu:

g++ Main.cpp Emulator.cpp -o emulator

Napomena: Pošto I assembler I emulator imaju fajl Exception.h, prevodjenje je neophpdno raditi u odvojenim direktorijuma.

# Pokretanje

Pre svega je potrebno napraviti objektne fajlove od asemblerskih fajlova. Sledeći proces je potrebno uraditi za svaki asemblerski fajl. Ovo se radi pomoću programa asembler.

Kada smo dobili objektne fajlove, treba pokrenuti emulator. Kako bismo to uradili potrebno je napisati sledeću komandu.

emulator [-place=imeSekcije@adresaNaKojojTrebaDaSeNadjeSekcija]\* [imeUlaznogFajla]+

Prethodnom komandom se pokreće simulacija.

## Opis rešenja

### Emulator

Prvo je odrađeno učitavanje ulaznih fajlova i pravljenje odgovarajućih struktura(Tabele simbola, tabele sekcija sa kodom i relokacionim zapisima, i tabela equ relokacionih zapisa).

Kada su se napravile odgovarajuće strukture, proverava se da li imamo višestruko definisane simbole. Takođe se određuje ukupna veličina svake sekcije kao i početna adresa. Zatim se proverava da li se sekcije preklapaju. Dodeljivanje početnoj adresi za svaku sekciju se radi prvo po adresama koje je korisnik zadao za date sekcije a zatim po slobodnim adresama.

Posle se svakom delu sekcije dodeljue počtna adresa. Onda se svaki simbol pomera za odgovarajuću novu početnu vrednost sekcije. Pomeraju se simboli koji pripadaju nekoj sekciji. UNKNOWN i UNDEFINED simboli se ne diraju.

Zatim se za svaki fajl razrešavaju undefined simboli i equ undefined simboli, ovde se koriste equ relokacioni zapisi.

Zatim se razrešavaju relokacioni zapisi za svaki fajl

Onda se kod svake sekcije upisuju u memoriju.

Zatim se prvo resetuje procesor, odnosno ucitava se adresa 0 u PC i resetuje se PSW i SP.

Onda se kreće u simulaciju. Prvo se dohvata instrukcija, zatim se ide u fazu izvršavanja instrukcije. Kada se izvrši instruckija, proverava se da li treba prestati sa simulacijom(HALT instrukcija), zatim se proverava da li treba da se ispise nesto u DATA\_OUT registar, onda se proverava da li je prosla perioda i da li treba pozvati timer prekidnu rutinu, samim tim se resetuje proteklo vreme kada se završi prekidna rutina timer. Onda se proverava da li je korisnik nesto uneo u terminal i poziva se prekidna rutina ze terminal.

Dohvatanje instrukcije se sastoji od dohvatanja operacionog koda i dohvatanje operanada. Svaka instrukcija ima svoju funkciju za izvršavanje.

## Емулатор

#ifndef emulator\_h

#define emulator\_h

//Načini adresiranja

enum class AdressMode { IMD, REGDIR, REGIND, REGINDPOM, MEM };

//Oznake za odogvarajuce registre

enum Registers {r0,r1,r2,r3,r4,r5,r6,r7,sp=6,pc=7,psw=0xf};

//Struktura koja služi za predstavljanje jednog operanda instrukcije

struct Operand {

AdressMode addressMode;

std::string symbol;

char usedRegister;

char operandSize;

char usedPart;

int operandAditionalInfo;

int operandValue;

Operand() {

this->addressMode = AdressMode::IMD;

usedRegister = 0; operandSize = 0; usedPart = 0; usedRegister = 0;

operandAditionalInfo = 0; operandValue = 0;

}

};

//Obeležja za kodove instrukcije

enum InstructionName {

UNKNOWN = -1,HALT, IRET, RET, INT, CALL, JMP, JEQ, JNE, JGT, PUSH,

POP, XCHG, MOV, ADD, SUB, MUL, DIV, CMP, NOT, AND,

OR, XOR, TEST, SHL, SHR, MINOPCODE = 0, MAXOPCODE = 24

};

//Služi da predstavi jednu instrukciju

struct Instruction {

InstructionName instructionName;

char operandNumber;

char operandSize;

Operand operand1, operand2;

unsigned char instructionDescriptionCode;

Instruction() {

instructionName = UNKNOWN; operandNumber = 0; operandSize = 0; instructionDescriptionCode = 0;

}

};

//Tipovi relokacionih zhteva

enum class TypeOfUse { PC\_REL, SYMBOL, SYMBOL\_ONE\_BYTE, EQU\_SYMBOL, EQU\_SYMBOL\_SUBSTRACT, DECIDE\_LATER };

//Služi da se razreše equ simboli koji zavise od extern simbola

struct SymbolValue {

std::string symbol;

char sign;

bool isDefined;

int sectionNumber;

SymbolValue(std::string symbol, char sign) {

this->symbol = symbol;

this->sign = sign;

isDefined = false;

}

SymbolValue() {

this->symbol = "";

this->sign = '+';

isDefined = false;

};

};

//Red u tabeli realokcionih zapisa

struct RealocEntry {

TypeOfUse realocType;

long location;

long symbolTableReference;

bool negativeSymbol;

RealocEntry() {

realocType = TypeOfUse::DECIDE\_LATER; negativeSymbol = false; location = 0; symbolTableReference = 0;

}

};

//Red u tabeli simbola

struct SymbolEntry {

static long nextSymbolNumber;

std::string name;

long sectionNumber;

long value;

long size;

bool isGlobal;

bool isEqueUndefined;

long number;

SymbolEntry() {

name = ""; sectionNumber = 0; value = 0; isGlobal = false; number = 0; size = -1; isEqueUndefined = false;

}

};

//Red u tabeli sekcija

struct SectionEntry {

std::string name;

long startAdress;

long size;

std::vector<char> code;

std::vector<RealocEntry>realocTable;

long number;

SectionEntry() {

name = ""; startAdress = -1; size = -1; number = 0;

}

};

//Red u tabeli fajlova

struct FileEntry {

std::map<std::string,SectionEntry> sectionTable;

std::map<int, SectionEntry> sectionTableByNumber;

std::map<std::string, SymbolEntry> symbolTable;

std::map<int, SymbolEntry> symbolTableByNumber;

std::vector<RealocEntry> equReallocation;

};

//Sluzi kako bi se na osnovu sekcija u fajlovima, dobile početne adrese svake sekcije u svakom fajlu, kao i da bi se izračunala veličina sekcije i dodeli početna adresa nezavisno od toga u kom fajlu se nalazi.

struct SectionInfo {

int startAddress;

int size;

int offset;

SectionInfo(){

startAddress = 0; size = 0; offset = 0;

}

};

//Sluzi kao hardver

struct Machine {

char memory[65535] = {0};

int pc;//Broj instrukcije

char instructionPart[7] = { 0 };

int registers[16];

bool stop = false;

bool writeDataOutRegister = false;

bool timeForInterrupt = false;

};

//Pokazivač na funkciju koja izvršava instrukciju

typedef void (\*InstructionFunction) (Instruction&);

class Emulator {

private:

/Perioda između prekida

long timeBetweenInterrupt[8] = {500, 1000,1500,2000,5000,10 \* 1000,30 \* 1000,60 \* 1000};

//Range za word i byte

enum WORD\_RANGE {LOW\_WORD = −32,768, HIGH\_WORD = 65555};

enum BYTE\_RANGE { LOW\_BYTE = 0, HIGH\_BYTE = 255 };

//Specijalne vrednosti za sekcije simbola

enum symbols {UNKNOWN = -1, UNDEFINED};

//Oznake za memorijski mapirane registre

enum MEMORY\_MAPED\_REGISTERS{DATA\_OUT = 0xFF00, DATA\_IN = 0xFF02, TIMER\_CFG = 0xFF10};

//Ulazi u tabeli prekidnih rutina

enum INTERRUPT\_ENTRY{RESTART = 0, ERROR = 1, TIMER = 2, TERMINAL = 3};

//Veličina operanda

enum OPERAND\_SIZE {ONE\_BYTE = 1, TWO\_BYTE = 2};

//Range za adresu u memoriji

enum MEMORY\_ADDRESS { MAX\_ADDRESS = 0xFFFF, MIN\_ADDRESS = 0x00 };

static Emulator\* emulator;

static Machine machine;

/Koliko koja funckija ima operanada

static std::map<InstructionName, int> instructionNumberOfOperands;

//Koja funkcija izvršava koju instrukciju

static std::map<InstructionName,InstructionFunction>instructionFunction;

//Tabela sa infomracijama o sekcijo

static std::map<std::string, SectionInfo> sectionInfoTable;

//Tabela sa informacijama o fajlovima

static std::map<std::string, FileEntry> fileTable;

static Emulator\* makeEmulator();

static std::vector<std::string> inputFileNames;

static std::map<std::string, unsigned int> sectionStartAdress;

//Čitanje jednog reda tabele simbola iz ulaznog fajla

static void processSymbolTableLine(std::string fileName, std::string oneLine);

//Čitanje jednog reda equ relokacionih zapisa iz ulaznog fajla

static void processEquReallocTableLine(std::string fileName, std::string oneLine);

//Čitanje jednog reda relokacionih zapisa za sekciju iz ulaznog fajla

static void processSectionReallocTableLine(std::string fileName, std::string sectionName, std::string oneLine);

//Čitanje jednog reda koda za odgovarajuću sekciju iz ulaznog fajla

static void processSectionCodeLine(std::string fileName, std::string sectionName, std::string oneLine);

//Provera da li imamo višestruko definisane simbole

static bool checkIFHaveMultipleDefinedSymbols();

//Određivanje ukupne veličine sekcija ukoliko se ponavljaju u vise fajlova

static void decideSectionsSize();

//Određvanje početne adrese za svaku sekciju

void decideSectionsStartAddress();

//Provera da li se za dodeljene početne adrese sekcije preklapaju

void checkIfSectionsOverlap();

//Dodeljivanje početne adrese svakom delu sekcije

void assignSectionsPartStartAddress();

//Pomeranje simbola za početnu adresu njihove sekcije

void moveSymbolsBySectionOffset();

//Razrešavanje equ simbola koji zavise od extern simbola kao i svih extern simbola u fajlovima.

void resolveUndefinedEQUSymbols();

//Obrada relokacionih zapisa za svaku sekciju u svakom fajlu

void processReallocEntries();

//Ažuriranje tabela sekcija i simbola po broju odnosno po imenu

void updateTablesByNumber();

void updateTablesByName();

//Određivanje o kojoj instrukciji se radi i koja su veličine operandi

void extractInstructionCode(Instruction&);

void extractInstructionSize(Instruction&);

//Određivanje načina adresiranja i dodatnih informacija koje idu uz to

void getOpernadDescription(Operand&,int, int);

//Ukoliko se radi o neposrednom, memorijskom ili regindpom, dohvatanje dodatnih informacija

void getAdditionalOperandInfo(Operand&,int,int&, int size);

//Upis i citanje operanda u memoriju

int readOperandFromMemmory(int fromWhere, int size);

int writeOperandToMemmory(int where, Operand &operand);

//Izvršavanje instrukcije

void executeInstruction(Instruction&);

//Dohvatanje instrukcije

Instruction fetchInstruction();

//Dohvatanje operanada

void fetchOperands(Instruction&);

//Dohvatanje operanda

void fetchOperand(Operand&);

//Upis operanda

void writeOperand(Operand&);

//Inkrementiranje PC-ija

static void incPC();

//Obezbedjuje da adresa sigurno bude 16 bita

static int getAddress(int address);

//Funkcije koe su zadužene za izvršavanje određene instrukcije

static void instructionHALT(Instruction& instruction);

static void instructionIRET(Instruction& instruction);

static void instructionRET(Instruction& instruction);

static void instructionINT(Instruction& instruction);

static void instructionCALL(Instruction& instruction);

static void instructionJMP(Instruction& instructon);

static void instructionJEQ(Instruction& instructon);

static void instructionJNE(Instruction& instructon);

static void instructionJGT(Instruction& instructon);

static void instructionPUSH(Instruction& instructon);

static void instructionPOP(Instruction& instructon);

static void instructionXCHG(Instruction& instructon);

static void instructionMOV(Instruction& instructon);

static void instructionADD(Instruction& instructon);

static void instructionSUB(Instruction& instructon);

static void instructionMUL(Instruction& instructon);

static void instructionDIV(Instruction& instructon);

static void instructionCMP(Instruction& instructon);

static void instructionNOT(Instruction& instructon);

static void instructionAND(Instruction& instruction);

static void instructionOR(Instruction& instruction);

static void instructionXOR(Instruction& instructon);

static void instructionTEST(Instruction& instructon);

static void instructionSHL(Instruction& instructon);

static void instructionSHR(Instruction& instructon);

//Pomočne funkcije koje stavljaju i skidaju sa steka

static void push(int whatToPush);

static int pop();

//Upis podatka odgovarajuće veličine u memoriju

static void writeToMemmoryOneByte(int where, int whatToWrite);

static void writeToMemmoryTwoByte(int where, int whatToWrite);

static char readFromMemmoryOneByte(int fromWhere);

static int readFromMemmoryTwoByte(int fromWhere);

//Resetovanje procesora

void resetProcessor();

//Ažuriranje odgovarajučih flegova

static void updateZeroFlag(int result, int size);

static void updateNegativeFlag(int result, int size);

static void updateOverFlowFlag(int old1, int old2, int result, int size);

static void updateCarryFlagShiftRight(int old1, int old2, int result, int size);

static void updateCarryFlagShiftLeft(int old1, int old2, int result, int size);

static void updateZNFlags(int result, int size);

static void updateOCFlagsSub(int old1, int old2, int result, int size);

static void updateOCFlagsAdd(int old1, int old2, int result, int size);

//Ažuriranje znaka na osnovu 16. bita broja

static void updateSign(int& result);

//Dohvatanje odgovarajućeg flega

static bool getZ();

static bool getO();

static bool getC();

static bool getN();

static bool getTr();

static bool getTl();

static bool getI();

//Postavljanje odgovarjućeg flega

static bool setZ();

static bool setO();

static bool setC();

static bool setN();

static bool setTr();

static bool setTl();

static bool setI();

//Uklanjanje odgovarajućeg flega

static bool unsetZ();

static bool unsetO();

static bool unsetC();

static bool unsetN();

static bool unsetTr();

static bool unsetTl();

static bool unsetI();

//Prekidne rutine

//Poziv prekidne rutine koja se nalazi u ulazu broj number

static void callInterruptRoutine(char number);

//Periferije

//Terminal

//Postavljanje i restauriranje atribute terminala

void setTerminalAtrubutes();

void returnToOldTerminanlAtributes();

//Ispis data out registra

void writeDataOutRegister();

//Upis procitanog podatka u data in registar

void writeToDataInRegister(int whatTyped);

//Provera da li je korisnik nešto pritisnuo

bool checkIfUserTypedSomething();

//Tajmer

//Provera da li je prosla perioda od poslednjeg poziva tajmera

bool checkIfTimerElapsed(std::chrono::steady\_clock::time\_point &, long long& elapsedTime);

public:

//Provera da li su ulazni argumenti programa dobri

static bool checkIfArgumentsRight(int argc, char\* argv[]);

static Emulator \*getEmulator();

//Procitaj ulazne fajlove i formiraj odgovaraju će strukture

void processInputFiles();

//Linkuje fajlove

void linkFiles();

//Učitava fajlove u memoriju

void writeFilesToMemmory();

//Simulira izvršavanje programa

void simulate();

~Emulator();};

#endif // !emulator\_h