마컴설 설계 과제

1. **프로그램 전체 구조** 
   1. 프로그램 특징 요약

이 프로그램은 cpp파일인 main.cpp과 datatypes.h 파일로 구성된 프로그램으로 오브젝트 파일과 심볼테이블인 .sym, .obj 파일을 읽어와서 disassemble을 한 뒤 .asm 파일을 생성해서 결과물을 출력한다.

각 파일은 적절한 구조체에 저장되어 활용된다. symbol테이블 먼저 읽어서 오브젝트 파일의 라벨을 붙일 수 있도록 한다. 오브젝트 파일을 opcode와 바이트 유무를 따져서 instruction 구조체에 저장한다. 오브젝트 코드의 명령어를 포맷과 addressing방식에 따라 적절한 처리를 거친 후 .asm 파일로 출력한다. Disassemble이 완료된 파일인 .asm을 cmd로 확인할 수 있다.

* 1. 동작 순서
     1. .sym파일로부터 symbol table을 읽어와서 symtabData라는 구조체에 저장.
     2. .obj파일로부터 오브젝트 프로그램을 읽어와서 objectFildData라는 구조체에 저장
     3. 오브젝트 프로그램의 헤더 출력
     4. 텍스트 명령어 별로 적절한 처리를 거친 후 텍스트 출력
     5. 엔드 출력
     6. Disassemble한 .asm파일을 cmd에 출력
  2. **datatypes.h**

|  |  |
| --- | --- |
| 함수: 설명 | 코드 |
| InstructNode: 명령어 구조체 | class InstructNode {  public:  string command;  string format;  InstructNode(string command, string format) {  this->command = command;  this->format = format;  }  }; |
| headerRecord: 헤더 구조체 | **typedef** **struct** {  string programName;  string startAddress;  string programLength;  } headerRecord; |
| endRecord: 엔드 구조체 | **typedef** **struct** {  string firstExecAddress;  } endRecord; |
| Instruction: 명령어 구조체 | **typedef** **struct** {  string rawString;  **int** format;  **int** length;  string opCode;  string mnemonic;  **int** n;  **int** i;  **int** x;  **int** b;  **int** p;  **int** e;  string register1;  string register2;  string disp;  string byte;  } instruction; |
| textRecord: 텍스트 구조체 | **typedef** **struct** {  string startAddress;  string recordLength;  vector<instruction> instructions;  } textRecord; |
| objectFileData: object파일 구조체 | **typedef** **struct** {  ::headerRecord headerRecord;  vector<textRecord> textRecords;  ::endRecord endRecord;  } objectFileData; |
| symbolData : symbol구조체 | **typedef** **struct** {  string symbol;  string value;  string flags;  string convertedValue;  } symbolData; |
| symtabData: symtab 구조체 | **typedef** **struct** {  vector<symbolData> symbols;  } symtabData; |

* 1. main.cpp

|  |  |
| --- | --- |
| 함수, 설명 | 코드 |
| Printing\*\*: 헤더, 텍스트, 엔드 출력 | **void** printingHeader(headerRecord headerRecord, string filename) {  string file = filename + ".asm";  ofstream myFile(file, ios\_base::app);  …  }  **void** printingText(string memory, string label, string instruction, string operand, string rawString, **bool** format4, string filename) {  …  }  **void** printingEnd(string endAddress, string filename) {  …  } |
| Binarytohex, hextoBinary: 바이너리를 헥스로, 헥스를 바이너리로 바꿈 | string binaryToHex(string binary) {  vector<string> bytes;  string hexVal = "";  …  }  string hexToBinary(string hexValue) {  **long** **int** i = 0;  string bin = "";  …  } |
| build\_list: 전체 명령어 리스트 1,2,3/4(X)라 표현 | **void** build\_list(map<string, InstructNode>\* inList) {  insertToMap(inList, "18", InstructNode("ADD", "X"));  insertToMap(inList, "58", InstructNode("ADDF", "X"));  …  } |
| getMnemonicByOpCode:  opcode를 입력받아 명령어를 반환 | string getMnemonicByOpCode(string opCode) {  map<string, InstructNode> inList;  build\_list(&inList);  map<string, InstructNode>::iterator it = inList.find(opCode.c\_str());  **if** (it != inList.end()) {  **return** it->second.command;  }  } |
| getFormatByOpCode:  opcode를 입력 받아 포맷을 반환 | string getFormatByOpCode(string opCode) {  map<string, InstructNode> inList;  build\_list(&inList);  map<string, InstructNode>::iterator it = inList.find(opCode.c\_str());  **if** (it != inList.end()) {  **return** it->second.format;  }  } |
| getOpCode:  포맷 3,4일 경우 opcode 6자리 만들어서 반환 | string getOpCode(string opCodeByte) {  string binValue = hexToBinary(opCodeByte);  string opCodeBinary = binValue.substr(0, binValue.length() - 2);  opCodeBinary.push\_back('0');  opCodeBinary.push\_back('0');  **return** binaryToHex(opCodeBinary);  } |
| splitString:  symtab을 입력 받을 때 빈 공간을 구분함 | vector<string> splitString(string str) {  vector<string> strings;  string s = "";  **for** (**int** i = 0; i < str.length(); ++i) {  **if** (isspace(str.at(i))) {  **if** (!s.empty()) {  strings.push\_back(s);  s = "";  }  }  **else** {  s += str.at(i);  }  }  strings.push\_back(s);  **return** strings;  } |
| getFileLines:  파일을 읽어오는 함수 | vector<string> getFileLines(string fileName) {  vector<string> lines;  ifstream in(fileName.c\_str());  string line;  **while** (getline(in, line)) {  lines.push\_back(line);  }  in.close();  **return** lines;  } |
| getInstructions:  명령어를 분석해서 addressing모드와 opcode를 instruct 구조체에 넣어주는 함수.  텍스트라인 한 줄을 받아와서 처리함. 먼저 LOC와 심볼테이블을 확인해서 BYTE일 경우를 처리하고, 명령어에서 포맷 2는 레지스터 분류, 포맷 3,4는 nixbpe와 operand를 분류해서 instruct 구조체에 넣어줌. | vector<instruction> getInstructions(string rawString, symtabData symtabData, string startAddress) {  vector<instruction> instructions;  **int** LOC = (**int**)strtol(startAddress.c\_str(), NULL, 16);  **while** (rawString.length()) {  …  }  …  instruct.opCode = opCode;  instruct.format = numFormat;  instruct.mnemonic = mnemonic;  instruct.length = length;  instruct.rawString = rawInstructString;  instructions.push\_back(instruct);  rawString = rawString.substr(instruct.length);  LOC += instruct.format;  }  **return** instructions;  } |
| getObjectFileData:  오브젝트파일을 읽고 헤더, 텍스트, 엔드에 저장 | objectFileData getObjectFileData(string fileName, symtabData symtab) {  vector<string> objectFileLines = getFileLines(fileName);//오브젝트 파일의 모든 내용 읽기  //헤더 포맷 저장  headerRecord.programName = headerLine.substr(1, 6);  headerRecord.startAddress = headerLine.substr(7, 6);  headerRecord.programLength = headerLine.substr(13, 6);  //엔드 포맷 저장  string endLine = objectFileLines.back  endRecord.firstExecAddress = endLine.substr(1, 6);  //텍스트 포맷 저장  for (auto const& line : objectFileLines) {  if (line.front() == 'T') {  textRecord textRecord;  textRecord.startAddress = line.substr(1, 6);  textRecord.recordLength = line.substr(7, 2);  vector<instruction> instructions;  string instructsRawString = line.substr(9);  instructions = getInstructions(instructsRawString, symtab, textRecord.startAddress);  textRecord.instructions = instructions;  textRecords.push\_back(textRecord);  }  }  return objectFileData; |
| getSymtabData:  심볼테이블을 읽는 함수 | symtabData getSymtabData(string fileName) {  vector<string> symtabLines = getFileLines(fileName);  symtabData symtabData;  vector<symbolData> symbols;  symtabData.symbols = symbols;  **return** symtabData;  } |
| makeUpper:  LOC의 소문자를 대문자로 변경하는 함수 | string makeUpper(string LOC) {  **for** (**int** i = 0; i < LOC.length(); i++) {  **if** (isalpha(LOC.at(i))) {  LOC.at(i) = toupper(LOC.at(i));  }  }  **return** LOC;  } |
| makeString:  int LOC를 string LOC로 변환하는 함수 | string makeString(**int** targetLOCint) {  stringstream sstream;  sstream << setfill('0') << setw(4) << hex << targetLOCint;  string targetLOC = sstream.str();  targetLOC = makeUpper(targetLOC);  **return** targetLOC;  } |
| findSymbol:  operand의 symbol을 찾는 함수 | string findSymbol(string targetLOC, symtabData symtabData) {  string operand;  **for** (**int** i = 0; i < symtabData.symbols.size(); i++) {  string targetSymbolValue = symtabData.symbols.at(i).value.substr(2, 4);    **if** (targetLOC.compare(targetSymbolValue) == 0) {  operand = symtabData.symbols.at(i).symbol;  **break**;  }  **else** {  operand = targetLOC;  }  }  **return** operand;  } |
| getNextLOC:  현재 LOC와 포맷을 받아서 다음 LOC를 반환하는 함수 | string getNextLOC(string addr, **int** format) {  **int** decValue;  stringstream stream;  stream << addr;  stream >> hex >> decValue;  **int** decAddress = decValue + format;  stringstream stream2;  stream2 << setfill('0') << setw(4) << hex << decAddress;  string hexAddress(stream2.str());  hexAddress = makeUpper(hexAddress);  **return** hexAddress;  } |

Main 함수

|  |  |
| --- | --- |
| Obj, sym파일을 읽고,  Objectfiledata와 symtabdata에 저장 |  |
| 텍스트 라인별로, 명령어 별로 포맷 2,3,4일 경우에 적절한 처리를 해서 instructions 구조체에 넣어줌 |  |
| 텍스트로 출력하고, opcode가 LDB일 경우 따로 출력을 하고 basevalue 저장 |  |
| RESW, RESB를 출력.  symtab에서 워드 또는 바이트 간의 LOC차이를 구해서 그 크기를 구하고 텍스트로 출력함. |  |
| 엔드 부분을 출력하고 완성된 asm파일을 층에 출력함. |  |

1. **입출력 결과**

요구사항에 만족하는 결과를 얻을 수 있다.

입력에 사용한 sym파일과 obj파일

|  |  |
| --- | --- |
|  |  |
| Sample.sym | Sample.obj |

결과 파일인 sample.asm 파일

|  |
| --- |
|  |

1. **Discussion**

본 프로그램은 obj파일과 sym파일을 입력으로 받아 disassemble하여 asm파일을 출력하는 프로그램이다. 처음 프로그램을 설계할 때 사용할 구조체들을 잘 설계하는 것이 중요했다. 구조체를 불안정하게 짜면 나중에 힘들기 때문에 처음 구조체를 설계할 때 최대한 꼼꼼하게 설계한 뒤 불필요한 부분들을 제거했다.

disassemble과정에서 사전에 필요한 정보를 정리하는 것이 중요했다. 예를 들어 obj파일을 파싱할 때 명령어의 길이가 포맷별로, 바이트, 워드별로 다르기 때문에 symtab을 먼저 정리해야 했다. 이후에 명령어들을 벡터 배열에 넣어주고 이를 적절한 처리를 해서 텍스트 구조체에 넣었다.

바이너리, 헥스, 인트, 스트링 자료형들을 서로 변환하는 함수를 만든 것이 큰 도움이 됐다.

처음 오브젝트 파일을 읽고 이후에 main함수에서 포맷별로 다시한번 작업을 해줬다. 한번만 읽도록 프로그램을 짤 수 있었을 것 같다. 그렇게 짰다면 실행시간을 절반으로 단축할 수 있었을 것이다.