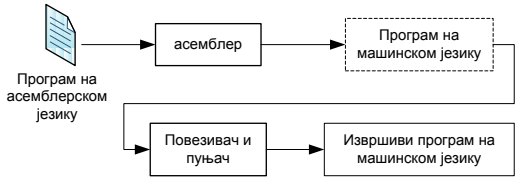
АСЕМБЛЕР

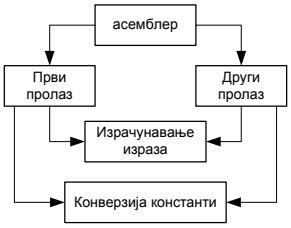
Теоријске основе

Асемблер је програм који преводи изворни програм написан на асемблерском језику у програм на машинском језику и извршиви програм на машинском језику.



Слика 1 Процес превођења са асемблерског језика

Због потребе за познавањем вредности симболичких израза у инструкцијама који не морају бити дефинисани пре њиховог коришћења, процес асемблирања се реализује у два пролаза.



Слика 2 Однос између модула асемблера

Оба пролаза користе потпрограме за израчунавање израза и конверзију константи (Слика 2), који су заједно са табелама, како дефинише [1], објашњени у даљем тексту, али прилагођени примеру малог асемблера који се користи на овом курсу, а који је близак данашњем MIPS асемблеру.

Први пролаз

Основни задатак првог пролаза асемблера је дефинисање свих симбола и литерала (константи смештених у меморијске локације) који се у програму користе. Због тога се мора доделити меморија за:

- сваку инструкцију која се генерише из симболичке машинске инструкције (нпр. sub $7, $8, 1),
- сваки литерал који се генерише псеудооперацијом α: .word β (α – симбол, β - литерал),
- сваки меморијски блок резервисан исказом .space P,
- сваку секцију која се дефинише псеудооперацијом .org S.

Када асемблер наиђе на машинску инструкцију у улазној датотеци, он мора одредити колико меморијских локација треба да заузме за ту инструкцију. Код неких циљних платформи (најчешће MISC типа) величина инструкције није увек иста, те се она може одредити тек након утврђивања типа инструкције; док је код неких других платформи (RISC типа) величина инструкције константна. У нашем примеру ради се о платформи са константном величином инструкције, која износи 32 бита, то јест 4 бајта.

Међутим, податак о величини инструкције није довољан да би асемблер знао колико меморијских локација јој мора доделити. Потребно је знати и организацију меморије у циљној платформи. Основне

карактеристике меморијског подсистема су адресни простор (са колико бита је дефинисана меморијска адреса), величина (адресни простор дефинише максималну величину меморије, али она у стварности може бити мања), и ширина (количина података која се налази на једној меморијској адреси, тј. локацији). Додатан податак који је потребан је управо ширина меморије. У нашем примеру ширина меморије је 8 бита, тако да асемблер једној инструкцији мора доделити четири (32/8=4) узастопне меморијске локације.

У платформи у овом примеру величина података над којима се обављају операције је исто 32 бита. Зато се каже да је платформа 32битна, то јест да је њена једна рачунарска реч (енг. word) 32 бита. Из тога следи да се и за сваки литерал заузимају 4 меморијске локације.

Унутар додељеног меморијског простора не сме да се појави ни једна недодељена (прескочена) меморијска локација. Ради остваривања оваквог редоследа додељивања меморијских локација, потребно је управљати **бројачем меморијских локација, који указује на прву недодељену реч** (која је у овом случају величине 4 бајта) у меморији. Након обраде сваког исказа бројач се увећава за одрђену вредност која зависи од појединачног исказа:

- у случају извршне инструкције, бројач се увећава за 4 (то јест, за једну реч),
- у случају литерала, бројач се увећава за 4,
- у случају резервисања блока, бројач се увећава за вредност дефинисану параметром P,
- у случају симбола, бројач се не мења,
- у случају псеудооперације .org S, бројач се поставља на адресу почетка секције S и евидентира се тренутна секција (*text* или *data*) уписом у табелу симбола.

За сваки пронађени симбол проверава се да ли већ постоји исти симбол у *табели симбола*. У случају да не постоји, симбол се додаје у табелу, а у случају да постоји, пријављује се грешка (јер симбол не сме бити редефинисан). Специјалне врсте симбола, глобални (енг. global) и спољашњи (енг. extern), смештају се у посебне табеле глобалних и спољашњих симбола, редом.

За сваки пронађени литерал формира се нова врста у *табели литерала* у коју се он додаје. Вредност литерала је константа и она се израчунава помоћу потпрограма за *конверзију константи*.

Приликом преузимања параметара операција и псеудооперација користи се потпрограм за *израчунавање израза*.

<u>Други пролаз</u>

Задатак другог пролаза је асемблирање и формирање машинских инструкција на основу симболичких инструкција. У те сврхе, неопходно је одређивање сваког поља у инструкцији. Вредности кодова операције се одређују на основу вредности које се налазе у *табели кода операције* у пољу одговарајуће врсте дефинисане симболом који се налази у првом пољу симболичке инструкције. Поред тога, у табели кода операције са наводи и број и тип операнда, што се приликом преузимања симболичког кода и параметара проверава и по потреби се пријављује грешка. Константе које су већ конвертоване у бинарне вредности такође се асемблирају и смештају у одговарајуће бајтове.

Главни задатак у асемблирању машинске инструкције је израчунавање израза који дефинише вредност различитих поља у инструкцији. Ови изрази су различито организовани у пољу операнда. Тип операнда се преузима из табеле кода операције ради одређивања начина обраде поља операнда.

Табеле у асемблеру

Табела 1 Табела симбола

Симбол	Вредност (меморијска локација)
N	0x00000018

Табела 2 Табела литерала

Константа (бројна вредност)	Меморијска локација
0x40	0x00001048

Табела 3 Табела кода операције

Симбол	Тип операције	Формат[1]	Нумерички код	Функција	Број параметара
add	машинска	R	0x0	0x20	3

[1] Поред стандардних формата инструкција (**R**egister, **I**mmediate и **J**ump), користе се и изведени типови (B, BZ, MD, JR, LUI, MF, N, S, SV и SC) ради лакше реализације малог асемблера.

Реализација модула и функција од интереса у библиотеци *assemblerLib*

Први пролаз асемблера

Први пролаз асемблера је, ради једноставности, подељен у два дела, односно учитавање линија кода у листу и замена псеудо инструкција је извучена у посебан модул *sourceLoader*. Модул је већ реализован у библиотеци и његова једина функција обавља наведене операције, односно попуњава листу (глобалну променљиву) *sourceList*:

```
bool loadSourceRepPseudo(std::string input_file)
```

Главни део првог пролаза остављен је за модул *passI* и његова једина функција је **нереализована**:

```
bool passI()
```

Табела симбола

Табела симбола је реализована као мапа са симболом као кључем и меморијском локацијом као придруженим податком, у модулу *assemblerLib*:

```
typedef std::map<std::string, SYMBOL> TABLE_SYMBOL          // мапа
void pushSymbol(std::string symbol, long location, SECTION_T section)
                                                            // додавање елемента
bool readSymbol(std::string symbol, long &val)              // преузимање и брисање ел.
bool symbolExists(std::string symbol)                       // провера постојања ел.
void pushGlobalSymbol(std::string symbol, long location)
                                                    // додавање глобалног симбола
bool globalSymbolExists(std::string symbol)         // провера постојања глобаланог сим.
void pushExternSymbol(std::string symbol, long location)
                                                    // додавање спољашњег симбола
bool externSymbolExists(std::string symbol)         // провера постојања спољашњег сим.
```

Табела литерала

Табела литерала је реализована као мапа са вредношћу литерала као кључем и меморијском локацијом као придруженим податком, у модулу *assemblerLib*:

```
typedef std::map<long, LITERAL> TABLE_LITERAL                       // мапа
void pushLiteral(long location, long value)  // додавање ел.
void readLiteral(long &location, long &value)               // читање и брисање ел.
```

Листе линија кода и бројева линија

Свака линија изворног кода предстаљена је структуром која садржи саму линију кода и редни број те линије у улазној датотеци:

```
struct SOURCE_LINE {
    long  lineNumber;
    std::string sourceLine;
};
```

Листа линија кода и бројева линија (структура *SOURCE_LINE*), као и функција добављања показивача на глобалну листу наведене су испод, а реализоване су у модулу *assemblerLib*:

```
typedef std::list<SOURCE_LINE> SOURCE_LIST;  // листа
```

```
SOURCE_LIST* getSourceList();                // добављање показивача на листу са изворним
                                             // кодом (глобална променљива у библиотеци
                                             // assemblerLib)
```

Листа параметара

Листа параметара псеудооперације или инструкције је у библиотеци реализована као стек.

```
typedef std::stack<std::string> PARAM_STACK          // стек
```

Помоћне функције при првом пролазу

Конверзија константи:

```
long constConv(std::string constant)
```

Ишчитавање дела линије кода са извршивом инструкцијом:

```
bool getExecutable(std::string &executable, std::string line, long lineNumber,
                   bool checkForErrors)
```

Ажурирање тренутне секције `section` и постављање бројача меморијских локација `location` на последњу адресу секције `sectionID`:

```
bool changeSectionAndLocation(std::string sectionID, SECTION_T &section, long &location)
```

Ишчитавање директиве из линије кода:

```
DIRECTIVE_T getDirective(std::string line, long lineNumber)
```

Ишчитавање симбола из линије кода:

```
bool getSymbol(std::string &symbol, std::string line)
```

Ишчитавање свих параметара из линије кода:

```
bool getParams(PARAM_STACK &params, std::string line, long lineNumber)
```

Провера једнакости актуелног броја параметара и очекиваног броја параметара:

```
bool checkEnoughParams(long lineNumber, std::string line, PARAM_STACK params, int
                       nExpected)
```

Пријављивање грешке додавањем поруке у листу грешака:

```
void addError(long lineNumber, std::string line, std::string errorText)
```

Провера постојања пријављених грешака и испис порука о грешкама, уколико постоје:

```
bool errorsFound()
```

Детаљнији опис ових и других функција можете прочитати у *html* документацији библиотеке *assemblerLib – doc/html/index.html*.

ЗАДАТАК

1. Шта је улаз, а шта излаз из првог пролаза асемблера?
2. Шта је улаз, а шта излаз из другог пролаза асемблера?
3. Шта подразумева конверзија константи?
4. Реализовати употребом програмског језика C/C++ и већ постојеће библиотеке асемблера

AssemblerLib, први пролаз асеблера у датотеци ***passI.cpp***. Искористити постојећи пројекат *small_assebmler*. Потребно је попунити табелу симбола, док је табела литерала већ попуњена. За њихово попуњавање потребно је водити рачуна о бројачу меморијских локација. У пројекат је већ укључена библиотека *AssemblerLib.lib* и дефинисане су путање до додатних датотека (заглавља). Са циљем једноставније реализације користити постојеће функције за конверзију константи, издвајање директива, издвајање симбола, проверу да ли је линија извршна, проверу броја параметара, као и све функције за приступ табелама које су потребне у првом пролазу.