

Лекция 8

Програмиране на Асемблер

1. Компилятор на Асемблер TASM

За създаване на програма на Асемблер с използване на TASM са необходими някои допълнителни инструкции (директиви), за разлика от използването на *Debug*:

.MODEL SMALL – директива на Асемблер, която дефинира модела на паметта, използван в програмата;

.CODE – директива, която дефинира програмните инструкции;

.STACK – директива, която резервира пространство от паметта за стека;

END – директива, която задава края на програмния код.

Пример за програма на Асемблер:

```
; use ; to put comments in the assembler program
.MODEL SMALL; модел на паметта
.STACK ; заделяне на памет за стека с инструкции
.CODE ; следващите редове са програмни инструкции
mov ah,1h ; прехвърля стойност 1h в регистър ah
mov cx,07h ; прехвърля стойност 07h в регистър cx
int 10h ; 10h прекъсване
mov ah,4ch ; прехвърля стойност 4ch в регистър ah
int 21h ; 21h прекъсване
END ; край на програмния код
```

Тази асемблерска програма променя размера на показалеца (курсора) на компютъра. Текстът на програмата (source code) трябва да се съхрани като файл – например: **exam1.asm**. Файлът трябва да съдържа текста на програмата в ASCII формат. Използва се TASM за да се създаде обектен код. Например:

```
C:\>tasm exam1.asm
```

Следното съобщение е показател за правилна трансляция на програмата:

```
Turbo Assembler Version 2.0 Copyright (c) 1988, 1990 Borland International
Assembling file: exam1.asm
Error messages: None
Warning messages: None
Passes: 1
Remaining memory: 471k
```

Резултат от изпълнение на трансляцията е създаване на файл **exam1.obj**.

Програмата TLINK се използва за създаване на изпълним файл:

```
C:\>tlink exam1.obj
```

Създава се изпълним файл **exam1.exe**, който директно може да се изпълни от командния интерпретатор на DOS:

```
C:\>exam1[enter]
```

1. Сегментиране на паметта

Архитектурата на x86 процесорите използва оперативната памет на компютъра, като я разделя на сегменти, размера на които е 64kb. Причината за това разделяне на паметта е, че максималният адрес (двоично число), което може да се съхрани в регистрите на микропроцесора има дължина 16 бита (**дума**). Това означава, че не може да се адресира клетка с номер по-голям от 65536 (2^{16}). Поради тази причина паметта на компютъра се раздели в групи (сегменти), всеки с по 65536 клетки (байта),

и се използва специален регистър, чрез който се задава номер на сегмента. Действителният адрес на клетка от паметта в този случай се задава с два регистъра – един за номер на сегмента и друг за адреса вътре в сегмента – отместване (offset). По този начин може да се адресира памет от 4294967296 байта.

За да може чрез Асемблерска програма да се управляват данните е необходимо всяка част информацията или инструкциите да се разполагат в съответен сегмент от паметта. Асемблер осъществява достъп до тази информация или инструкции, използвайки адреса на сегмента зададен с регистри DS, ES, SS и CS и адрес вътре в сегмента – адреса на клетка от паметта в този сегмент.

Ако се използва Асемблер с помощта на програма Debug, мястото където се разполага дадена команда може да се зададе по следния начин:

1CB0:0102 MOV AX,BX

Тук първия адрес 1CB0, съответства на номер на сегмент от паметта, а втория адрес 0102 съответства на адрес вътре в сегмента. Инструкцията ще се разположи от този адрес нататък. Посредством директивите **.CODE**, **.DATA** и **.STACK** може да се укаже какви сегменти от паметта ще се използват в програмата на Асемблер.

Асемблер може да задава размера на сегментите в зависимост от дължината на програмите за да се оползотворява ефективно паметта. Например, ако една програма използва не повече от 10 kb памет за съхранение на информацията, сегментът за данни може да бъде зададен с размер 10 kb, а не 64 kB.

2. Типове инструкции в език Асемблер.

Обикновено всеки микропроцесор има групи команди за прехвърляне на данни, аритметични и логически операции и други. Понякога те имат различни мнемонически означение, в зависимост от типа на микропроцесора. Въпреки това, основните команди за всеки процесор са близки в тяхната същност и групирането им помага за тяхното разбиране и изучаване.

2.1 Трансфер (прехвърляне) на данни

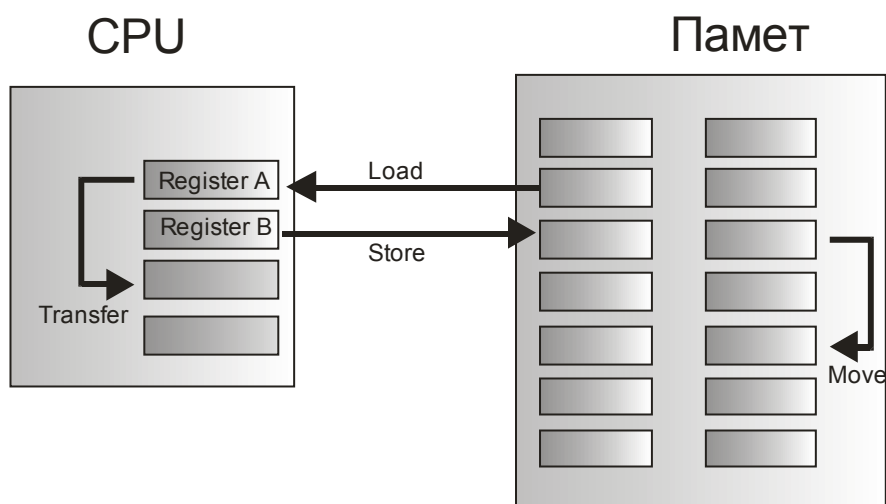
В програмите често се налага да се прехвърлят данни (информация) между адреси (клетки) от паметта, регистри на CPU и външни устройства – фиг. 2. Съществуват различни начини да извърши това. Може например да се копира съдържанието на клетка от паметта в някой регистър; да се копира информация от регистър в друг регистър; да се прехвърли информация от регистър към стека, от стека към регистър; да се прехвърля информация към външни устройства и други.

Пренасянето на данни е регламентирано с помощта на няколко правила и ограничения. Ето някои от тях:

- Не е възможно да се пренасят данни от клетка в паметта към друга клетка от паметта директно. Необходимо е първо данните да се прехвърлят в регистър на CPU и след това от регистъра да се пренесат към другата клетка от паметта.

- Не възможно да се прехвърли константа директно към сегментен регистър. Трябва първо константата да се прехвърли в регистър на CPU и след това в сегментен регистър.

- Възможно е прехвърлянето на блок от данни с помощта на **mov** инструкции, които копират последователност от битове или думи. Такива инструкции са **movsb**, която копира **n** байта от един адрес на друг и **movsw**, която копира **n** думи от един адрес на друг. Тези две команди използват стойности на адрес, зададен в регистри **DS:SI** като група от данни (байтове или думи) и ги пренася на друг адрес зададен с **ES:DI**.



Фиг. 2. Прехвърляне на данни в компютърна система

- **Команди за прехвърляне (трансфер) на данни**

Тези команди се използват за прехвърляне на данни. Всяка инструкция може да се използва с различни начини на адресиране. Типове команди:

MOV

MOVS (MOVSB) (MOVSW)

Команда MOV

Функцията на тези команди е да се прехвърлят данни между клетки от паметта, регистри и акумулатора.

Синтаксисът на тази команда е:

MOV Destiny, Source

Тук **Destiny** е адрес (място), където данните ще се прехвърлят, а **Source** е мястото (адрес), където се намират данните.

Различните трансфери на данни, които се допускат с тази команда са:

- **Destiny:** memory. **Source:** accumulator
- **Destiny:** accumulator. **Source:** memory
- **Destiny:** segment register. **Source:** memory/register
- **Destiny:** memory/register. **Source:** segment register
- **Destiny:** register. **Source:** register
- **Destiny:** register. **Source:** memory
- **Destiny:** memory. **Source:** register
- **Destiny:** register. **Source:** константа (стойности)
- **Destiny:** memory. **Source:** константа (стойности)

Примерна програма:

MOV AX,0006h

MOV BX,AX

MOV AX,4C00h

INT 21H

Тази малка програма прехвърля стойност **0006H** в регистър **AX**, след което прехвърля съдържанието на регистър **AX** (0006h) в регистър **BX**, и накрая прехвърля стойност **4C00h** в регистър **AX**. Програмата завършва с изпълнението на функция **4C** за **21h** прекъсване.

- **MOV** инструкцията копира **source** операнда в **destination** операнд без промяна на **source**.

LDS Инструкция

Функцията на тази инструкция е да се зареди регистъра за сегмента за данни

LDS *destiny, source*

Source операнда трябва да бъде двойна дума в паметта (32 бита). Думата (16 бита), свързана с по-високия адрес се записва в **DS** регистъра, а другата дума се използва като адрес на сегмента. Думата, свързана с по-малкия адрес е изместването в сегмента и се разглежда като **destiny** (адрес за изпращане).

LEA Инструкция

Функцията зарежда адреса на **source** операнда

LEA *destiny, source*

Source оператора трябва да бъде разположен в паметта, а изместването в сегмента е разположено в индексния регистър или специален указател за адрес на изпращане (дестинация).

За илюстрация на тези възможности може да се запише следния еквивалент от операции:

MOV SI,OFFSET VAR1

е равнозначно на:

LEA SI,VAR1

2.2. Стекови инструкции

Стекът е пространство от паметта, в което се съхранява информация, достъпът до която се осъществява по правилото „последен въведен – първи достъпен“. Това означава, че в тази памет се натрупва информация, при която всеки нов запис се позиционира на върха на стека и във всеки момент е достъпен последният запис. След като се прочете той, става достъпен следващият и т.н.

POP Инструкция

Функцията на тази инструкция е да се прочете (възвърне) информация от стека.

POP *destiny*

Инструкцията прочита последната записана в стека информация, записва я в място указано с операнда **destiny** и след това увеличава с 2 стойността на SP регистъра. Това нарастване се използва за да се позиционира SP регистъра на следващата записана информация. Стекът работи с думи, което означава, че записаната информация е 2 байта.

За да се възстановят коректно стойностите на регистри чрез съхранение им стойности в стека се използват други команди в следната последователност:

POP CX, POP BX, POP AX

POPF Инструкция

Функцията на тази команда е да възвърне стойностите на флаговете във флаговия регистър, записани в стека.

POPF

Тази команда прехвърля битовте от думата, записана в най-горния участък от стека във флаговия регистър.

След прехвърлянето на битовете във флаговия регистър стойността в SP регистъра се увеличава с 2, намалявайки по този начин размера на стека .

PUSH Инструкция

Функцията на тази инструкция е да запише информация (дума) в стека

PUSH source

PUSH инструкцията намалява с 2 стойността в **SP** регистъра и прехвърля съдържанието от адреса, зададен със **source** операнда в стека. Съдържанието на стека нараства с 2 байта и това е информацията, която той може да обработва при следваща операция.

За да се съхрани съдържанието на регистри в стека се използват следните инструкции: **PUSH AX, PUSH BX, PUSH CX**

PUSHF Инструкция

Функцията на тази инструкция е да се прехвърли (запише) съдържанието на флаговия регистър в стека.

PUSHF

Тази инструкция намалява с 2 стойността на SP регистъра и прехвърля съдържанието на флаговия регистър в стека.

2.3. Инструкции за прехвърляне на данни към външни устройства

За прехвърляне на данни към външни устройства (портове) се използва команда **out**, а за прочитане (приемане) на данни от порт се използва команда **in**.

OUT DX,AX

Тук **DX** съдържа адреса на порт, който се използва за изпращане на данните и **AX** съдържа информацията, която се изпраща..

IN AX,DX

Тук **AX** е регистър, където получаваната информация ще бъде съхранена и **DX** е регистър, който съдържа адреса на порт, чрез който се прехвърля необходимата информация.

2.4. Логически и аритметически операции

Инструкциите за логически операции са: **and, not, or** и **xor**. Тези операции действат върху битовете, зададени като операнди за операциите.

Алгебричните операции са: **add** (събиране), **sub** (изваждане), **mul** (умножение) и **div** (деление).

Почти всички операции за сравнение са свързани с информация, съдържаща се във флаговите регистри. Един от флаговете, които могат директно да се задават от програмистите са флага за направлението на данните **DF**, използван за задаване на операциите за последователност (списък от данни). Друг флаг, който също може да се управлява от програмистите е **IF** флага с помощта на **sti** и **cli** instructions, за активиране и деактивиране на прекъсванията.

- Логически инструкции

Те се използват за извършване на логически операции върху операндите.

AND, NEG, NOT, OR, TEST, XOR

AND Инструкция

Функцията на тази команда е да се изпълни логическата операция 'конюнкция' върху операндите бит по бит.

AND destiny, source

Source Destiny | Destiny

1	1	1
1	0	0

0	1		0
0	0		0

Резултатът от тази операция се записва в адрес зададен с **destiny** операнда.

NEG Инструкция

Функцията на тази инструкция е допълване до 2.

NEG destiny

Тази инструкция генерира допълнение до 2 на съдържанието на адреса в destiny операнда и записва резултата на същото място

Например ако в AX регистъра има стойност 1234H, тогава:

NEG AX

Ще генерира стойност EDCCH, която ще се съхрани в AX регистъра.

NOT Инструкция

Функцията на тази инструкция е да генерира отрицание на стойността съхранявана чрез destiny операнда бит по бит. .

NOT destiny

Резултатът се съхранява на същото място, където се намира дестини стойността.

OR Инструкция

Функцията на тази команда е да се изпълни логическата операция, 'дезюнкция' върху операндите бит по бит.

OR destiny, source

Резултатът от тази операция се записва в адрес зададен с **destiny** операнда.

TEST Инструкция

Функцията на тази инструкция е да сравни стойностите съхранявани в source и destiny операндите бит по бит. .

TEST destiny, source

Тази операция извършва логическата операция 'конюнкция' бит по бит, но за разлика от операция **and** резултатът не се записва в **destiny**. Той има ефект само за състоянието на флага.

XOR Инструкция

Функцията на тази команда е да се изпълни логическата операция, 'изключваща дезюнкция' върху операндите бит по бит.

XOR destiny, source

Source	Destiny		Destiny
1	1		0
1	0		1
0	1		1
0	0		0

- Аритметични инструкции

Те се използват за извършване на аритметични операции върху операндите.

ADC, ADD, DIV, IDIV, MUL, IMUL, SBB, SUB

ADC Инструкция

Функцията на тази операция е да се реализира аритметичната операция събиране.

ADC destiny, source

Инструкцията извършва събиране на стойностите зададени с двата операнда и се добавя 1 към резултата ако CF флага е активиран.
Резултатът се записва в адрес зададен с **destiny** операнд.

ADD Инструкция

Функцията на тази операция е да се реализира аритметичната операция събиране.

ADD destiny, source

Инструкцията извършва събиране на стойностите, зададени с двата операнда и резултатът се записва в адрес, зададен с **destiny** операнда.

DIV Инструкция

Функцията на тази инструкция е деление без знак.

DIV source

Делителят може да бъде байт или дума и е зададен със **source** операнда. Ако делителят е 8 бита (1 байт) 16-те бита на регистър AX се разглеждат като делимо, а ако делителят е 16 бита (дума) – комбинацията DX:AX регистри се разглежда като делимо, като DX регистъра се разглежда като старша дума и AX като младша дума. Ако делителят е байт частното се записва в регистър AL и остатъка – в регистър AH. Ако делителят е дума, частното се записва в регистър AX и остатъка - в регистър DX.

IDIV Инструкция

Функцията на тази инструкция е деление.

IDIV source

Тази инструкция е както горната, като единствената разлика е, че резултатът е със знак.

MUL Инструкция

Функцията на тази инструкция е умножение без знак.

MUL source

В езика Асемблер се приема, че множимото е със същия размер както множителя. Това позволява множимото се съхранява в AH регистър ако множителя е 8 бита или в AX регистъра – ако множителя е 16 бита. Когато операндите за умножението са 8 битови, резултатът се съхранява в AX регистъра, а когато операндите при умножението са 16 битови – резултатът се записва в DX:AX регистри.

IMUL Инструкция

Функцията на тази инструкция е умножение на две числа със знак.

IMUL source

Тази инструкция изпълнява същите действия както горната, с тази разлика, че се взмат пред вид и знаците на числата. Резултатът се съхранява в същия регистър, който се използва от операцията MOV за зареждане на множимото.

SBB Инструкция

Функцията на тази инструкция е изваждане с отстъпка.

SBB destiny, source

Тази инструкция изважда **source** стойност от **destiny** и се изважда единица от резултата ако флаг CF е активиран. Този тип изваждане се използва когато се работи с 32 битови стойности.

SUB Инструкция

Функцията на тази инструкция е изваждане.

SUB destiny, source

Тази инструкция изважда **source** стойността от **destiny** стойността.

2.5. Преходи, цикли и процедури

Безусловните преходи в асемблерските програми се задават с **jmp** команда. Командата за преход се използва за прехвърляне на изпълнението на потока от инструкции на определено място в програмата, зададено с определен етикет (адрес). Цикълът е програмна конструкция за организиране на повторение на процеса определен брой пъти, докато се изпълни определено условие.

Jump инструкции

Тези инструкции се използват за промяна на последователността (потока) на изпълнение операциите (командите).

JMP, JA (JNBE), JAE (JNBE), JB (JNAE), JBE (JNA), JE (JZ), JNE (JNZ), JG (JNLE), JGE (JNL), JL (JNGE), JLE (JNG), JC, JNC, JNO, JNP (JPO), JNS, JO, JP (JPE), JS

JMP инструкция

Функцията на тази инструкция е безусловен преход.

JMP destiny

Тази инструкция се използва за разделяне на потока на изпълнение на командите в програмата без да се отчита състоянието на флаговете или данните. Прехода към друга инструкция се извършва безусловно. Тя съответства на оператора Goto в езиците от високо ниво.

Пример:

```
•           ; Handle one case
•           label1: .
•           .
•           jmp done
•           ; Handle second case
•           label2: .
•           .
•           jmp done
•           .
done:
```

JA (JNBE) Инструкция

Функцията на тази инструкция е условен преход.

JA Label

След сравнение тази команда прави преход ако сравнението е истина или не прави преход ако сравнението не е истина.

Това означава, че преходът се прави ако CF флаг е активиран или ZF флаг е деактивиран.

Другите команди за преход реализират условен преход при различни условия и състояния на флаговете.

Инструкции за цикъл

Тези инструкции осигуряват изпълнение на група операции многократно условно или безусловно, докато определен брояч има стойност нула.

LOOP, LOOPE, LOOPNE

LOOP Инструкция

Функцията на тази инструкция е да организира циклично изпълнение на група операции.

LOOP label

Loop инструкцията намалява стойността в CX регистъра с 1 и изпълнява потока от команди до етикета ***label*** докато стойността в CX е различна от 1.

LOOPE Инструкция

Функцията на тази инструкция е да организира циклично изпълнение на група операции в зависимост от състоянието на флаг ZF.

LOOPE label

Тази инструкция намалява стойността в CX регистъра с 1. Ако CX е различен от нула и ZF е равен на 1 се изпълнява потока от команди до етикета ***label***.

LOOPNE Инструкция

Функцията на тази инструкция е да организира циклично изпълнение на група операции в зависимост от състоянието на флаг ZF.

LOOPNE label

Тази инструкция намалява стойността в CX регистъра с 1. Ако ZF е различен от нула се изпълнява потока от команди до етикета ***label***.

2.6. Инструкции за управление на броячи

Тези инструкции се използват за увеличаване и намаляване на стойността на броячи.

DEC

INC

DEC Инструкция

Функцията на тази инструкция е да намали с 1 стойността на брояч.

DEC destiny

Тази операция изважда 1 от стойността зададена с ***destiny*** операнда и съхранява новата стойност на същия адрес.

INC Инструкция

Функцията на тази инструкция е да увеличи с 1 стойността на брояч.

INC destiny

Тази операция добавя 1 към стойността зададена с ***destiny*** операнда и съхранява новата стойност на същия адрес

2.7. Инструкции за сравнение

Тези инструкции се използват за сравнение на стойности и действат върху състоянието на флаговете.

CMP

CMPS (CMPSB) (CMPSW)

CMP Инструкция

Функцията на тази инструкция е да сравнява стойностите зададени с операндите.

CMP destiny, source

Тази инструкция изважда стойността зададена със ***source*** операнда от стойността зададена с ***destiny*** операнда без да съхранява резултата. Тя действа само върху състоянието на флаговете.

CMPS (CMPSB) (CMPSW) инструкции

Функцията на тази инструкция е да сравнява последователност от байтове или думи.

CMP destiny, source

С тази инструкция последователността от символи, зададена с операнда ***source*** се изважда от последователността, зададена с ***destiny***.

Регистър ***DI*** се използва като индекс за допълнителен сегмент за последователността на операнда ***source***, а регистър ***SI*** като индекс за последователността на операнда ***destiny***.

Инструкцията действа само върху съдържанието на флаговете и регистри DI и SI.

2.8. Инструкции за управление на флаговете

Тези инструкции действат директно върху съдържанието на флаговете.

CLC, CLD, CLI, CMC, STC, STD, STI

CLC Инструкция

Функцията на тази инструкция е да нулира флага за пренос.

CLC

Инструкцията изключва флага за пренос (бит във флаговия регистър) – записва се стойност нула.

CLD Инструкция

Функцията на тази инструкция е да нулира флага за адрес.

CLD

Инструкцията изключва флага за адрес – записва се стойност нула.

CLI Инструкция

Функцията на тази инструкция е да нулира флага за прекъсване.

CLI

Инструкцията изключва флага за прекъсване, като по този начин деактивира така наречените маскируеми прекъсвания. Маскируеми прекъсвания са тези прекъсвания, които се деактивират когато флага $IF=0$.

CMC Инструкция

Функцията на тази инструкция е да допълни (увеличи) флага за пренос.

CMC

Инструкцията променя състоянието на флага за пренос. Ако $CF = 0$ инструкцията променя тази стойност на 1, а ако $CF = 1$ инструкцията променя стойността 0. Може да се каже, че инструкцията инвертира стойността на флага.

STC Инструкция

Функцията на тази инструкция е да активира флага за пренос.

STC

Инструкцията записва стойност 1 във флага CF.

STD Инструкция

Функцията на тази инструкция е да активира флага за адрес.

STD

Инструкцията активира флага за адрес – записва се стойност 1.

STI Инструкция

Функцията на тази инструкция е да активира флага за прекъсване.

STI

Инструкцията активира флага за прекъсване, като по този начин прави възможни маскируемите прекъсвания. Маскируеми прекъсвания са тези прекъсвания, които се активират когато флага IF=1.