КАТЕДРА: КОМПЮТЪРНИ СИСТЕМИ И ТЕХНОЛОГИИ
ДИСЦИПЛИНА: КОМПЮТЪРНИ АРХИТЕКТУРИ

ЛАБОРАТОРНО УПРАЖНЕНИЕ № 3

ТЕМА: СИСТЕМА ИНСТРУКЦИИ НА ПРОЦЕСОРА (ISA)

Цел: Да запознае студентите със системата инструкции MIPS. Студентите ще научат как да пишат програми за MIPS архитектура, как да ги тестват и изпълняват.

I. Теоретична част:

Множеството от всички инструкции на процесора, които са видими за програмиста се нарича система инструкции на процесора, на англ. Instruction Set Architecture или ISA. ISA служи като граница между софтуера и хардуера.

Инструкциите имат различни формати. Форматът определя:

1) броя на битовете от които е съставена една инструкция, или (с други думи) нейната дължина и
2) кои битове за какво се използват, или полетата в инструкцията.

На фиг. 1 е представен един от възможните формати.

КОП	АОП1	АОП2	АРЕЗ

Фиг. 1. Общ формат на инструкция с три адресни полета

Показаната инструкцията се състои от 4 полета:

1) код на операцията (КОП) – указва на процесора какво точно да извърши с операндите, напр. да ги събере, извади, умножи и т.н.;
2) адрес на операнд 1 (АОП1) – посочва къде се намира първата стойност, участваща в операцията;
3) адрес на операнд 2 (АОП2) – посочва къде се намира втората стойност, участваща в операцията;
4) адрес на резултат (АРЕЗ) – посочва къде да се запише резултата от операцията.

Съществуват инструкции при които, вместо адрес на операнд, полето съдържа направо неговата стойност. Това се нарича непосредствено задаване на операнда.

Друг вариант за формат на инструкциите е показан на фиг. 2. В този случай адресът на първия операнд се използва и като адрес, в който да се запише резултатът. Разбира се стойността на резултата се записва на мястото на първия операнд, подменяйки деструктивно съдържанието му.

КОП	АОП1	АОП2

Фиг. 3. Общ формат на инструкция с две адресни полета

Представените общи формати на инструкцията определят броя и вида на полетата, но не фиксират дължините им. Има три възможни варианта според които да се определи каква да бъде дължината на инструкциите:

1. Всички инструкции да се изберат с еднаква дължина (например 32 бита).
2. Да може да се избира между няколко дължини, например между четири: 16, 32, 64 и 128 бита. В този случай е необходимо да се заделят 2 бита от кода на операцията, чрез които да се посочва колко е дълга инструкцията. Така инструкциите ще се разделят на четири групи, според дължините си.
3. Дължините на инструкциите да не са строго установени, а да могат да варират. В този

случай кодът на операцията трябва да определя броя на операндите, а всеки операнд да има своя спецификация, която да определя дължината му. Така дължината ще се определя по време на четенето на инструкцията. Когато се стигне до определен байт, ще се разбере, че инструкцията е свършила.

Форматът и дължината са два от аспектите на системата инструкции. Третият аспект касае видовете операции, които системата инструкции ще поддържа. Операциите са свързани с аритметическите блокове и процесора. След като се прочете инструкцията, се определя кои блокове ще са нужни на процесора за нейното изпълнение.

При проектиране на нов процесор, първичното е наборът от инструкции. Това ще рече, че първо се решава какви инструкции ще бъдат включени в набора, какви и колко операции ще се изпълняват, върху какви данни ще се изпълняват операциите, а след това се проектират процесорните блокове, необходими за тях.

Групи инструкции

Въпреки всички различия в операционните кодове за различните процесори, съществуват следните групи операции (респ. инструкции), които се поддържат от всички процесори:

работа с паметта – във всяка система инструкции са необходими операции за прехвърляне на данни между паметта и процесора. Във всяка инструкция за трансфер трябва да са посочени източника и дестинацията, както и дължината на данните.

аритметична обработка – в тази група попадат операциите за работа с цели числа и числа с плаваща запетая: събиране, изваждане, умножение, деление, унарните операции (abs, ++, --, -).

логическа обработка – булеви операции над битове, байтове, думи и др.: not, and, or, xor, shiftL, shiftR, rotateL, rotateR.

конвертиране на данните – форматират данните или променят техния формат (напр. от десетична към двоична бройна система).

управление на системата – привилегировани инструкции, които се използват в режим на супервайзор на ЦП и ОП. Такива са четене или зареждане на управляващите регистри, промяна на ключ за защита на паметта и други.

предаване на управлението – те променят естествения ред на изпълнение на инструкциите в компютърната програма, като указват адреса на инструкцията, подлежаща на изпълнение. Включва инструкции за преход, извикване и връщане на процедура, примитиви за цикъл, прекъсване.

безусловен преход – съдържат, в адресното си поле, адреса на следващата инструкция, която трябва да бъде изпълнена. Предизвикват процесора да продължи изпълнението на програмата от указания адрес.

условен преход – тестват определено условие. Когато е изпълнено условието, процесорът продължава изпълнението на програмата от адреса, указан в адресното поле. В противен случай се продължава с поредната инструкцията, така както е заложено в реда на изпълнение в програмата.

символна обработка – преместване на символен низ; сравняване два низа; търсене на шаблон; търсене на шаблон с канонично заместване; конкатенация; въвеждане на низ; извеждане на низ; повтаря указаната операция последователно за всеки елемент на низа.

вход и изход – поддържат вход и изход на данни.

Представените категории – формат и дължина на инструкциите и видове инструкции не са достатъчни, за да се дефинира напълно една система инструкции. За пълното дефиниране на системата инструкции на процесора е необходимо да се определят:

1) регистрите в процесора – т.е. количеството и вида на вътрешната памет в процесора, в която ще се съхраняват операндите;

2) адресирането на паметта – четенето и записът в основната памет изисква процесорът да подаде адрес, с който да се посочи с коя клетка от паметта ще се работи;
3) режимите на адресиране – начините по-които може да се определи къде се намират данните или как може да се формира адресът за достъп до паметта;
4) типът и размерът на операндите.

RISC (Reduced Instruction Set Computer)

Първите компютри са имали малък брой инструкции. С времето той се увеличава, респективно се увеличава сложността на процесора. Стига се до машини с няколкостотин инструкции (дори и такива с над 400 – например архитектурата на компютрите VAX). Най-късите инструкции са дълги един байт, най-дългите – над 160 байта.

Наборът инструкции оказва влияние върху сложността на компилатора. При повече инструкции е по-лесно за компилаторите да превеждат код на машинен език.

В началото на 1980-те в два университета – в Калифорнийския университет, Бъркли и в Станфордския университет се правят изследвания върху наборите от инструкции – проследяват се възможностите им и ефективността и се стига до извода, че съществува минимален набор от инструкции, чрез които могат да се изразят всички останали. Този набор се нарича ортогонален. Такива компютри биват наречени RISC (Reduced Instruction Set Computer) – компютър с процесор, имащ един ортогонален набор от инструкции. След като в Бъркли намират първия ортогонален набор, в Станфордския университет също откриват такъв и наричат тази система инструкции MIPS.

Ортогоналните инструкции в едно множество са около 50. При такъв набор програмирането на машинен език става бавно и изисква повече писане, тъй като често са необходими няколко RISC инструкции за работата, която извършва една единствена инструкция от по-голям набор. RISC обаче не са предвиждани за машинно програмиране, тъй като навлизането им е обвързано с навлизането на езиците от високо ниво.

Предимството е, че такъв набор от инструкции се изпълнява ефективно от процесора, понеже са малко на брой, не са толкова дълги и се изпълняват за малък брой тактове. Така тежестта на програмирането се прехвърля от сложността на процесора към сложността на компилатора, който трябва да превежда от езици от високо ниво на машинен език и да оптимизира кода. При RISC наборите има правила – всички инструкции се изпълняват за фиксиран брой тактове (например 4), и са с точно определена дължина, което ускорява допълнително процесора. Всички инструкции за аритметична и логическа обработка имат 3 операнда, представляващи регистри. Достъпът до паметта е явен и се осъществява единствено, чрез две инструкции за работа с паметта.

II. Практическа част:

В практическата част студентите ще се запознаят със системата инструкции MIPS, за да напишат програми на асемблер за класически RISC процесор в учебната среда Mipsit и да анализират тяхното изпълнение.

Система инструкции MIPS

MIPS (Microprocessor without Interlocked Pipeline Stages) е система инструкции от типа RISC. Тя има 32 и 64 битови варианти и е реализирана в множество процесори, включително за вградени системи. Други популярни системи инструкции са: Intel 80x86, Motorola 88000, SPARC, ARM.

Системата инструкции MIPS 32 работи с 32-битови данни. Първата нейна версия е създадена през 1985 г., а последната е от 2014 г.

Всички инструкции са 32 битови, операндите са с размер 1 байт, 2 байта (полудума) или 4 байта (дума). Един символ заема 1 байт. За съхранение на целочислените данни е необходима 1 дума (4 байта).

При задаване на непосредствени стойности (наричани още литерали) в програмата на асемблер се използват следните правила: десетичните числа се задават направо, напр. 4,

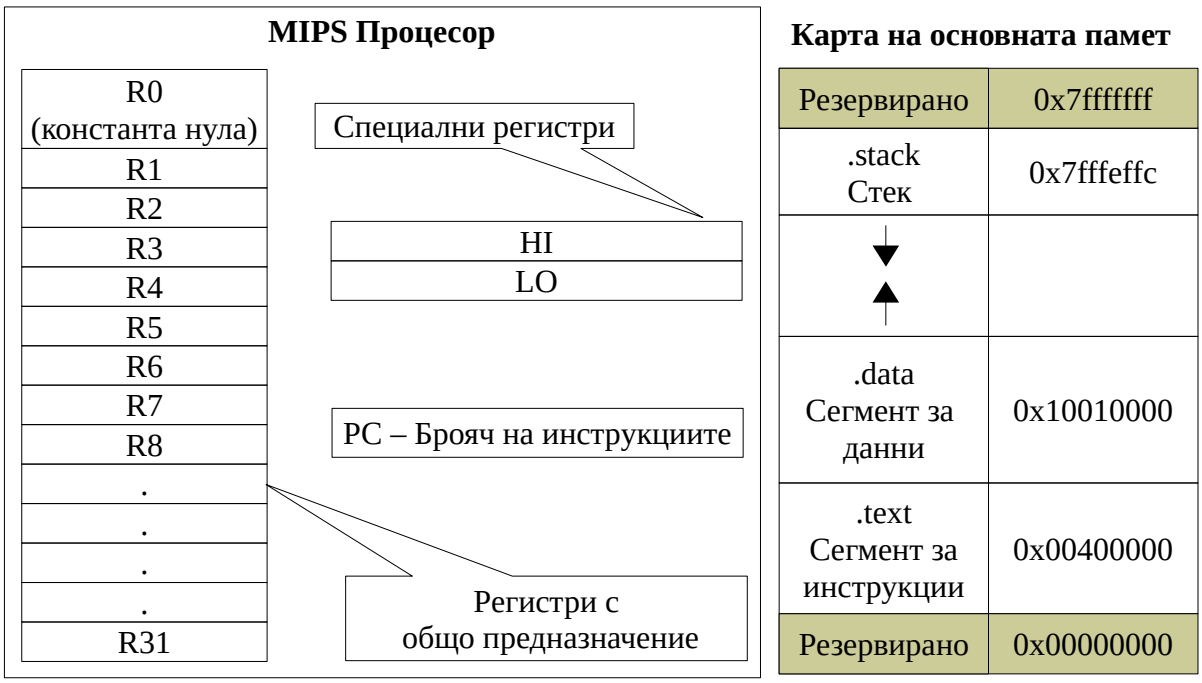
шестнадесетичните числа се задава с префикс 0x, напр. 0x1ac5d68e, единичен символ се огражда с единични кавички, напр. 'b', низовете се ограждат с двойни кавички, напр. "A string".

Регистри в MIPS

Регистрите в MIPS, достъпни за програмиста, са два вида с общо предназначение и специални. Те са представени на фиг. 3. Фигурата представя също паметта в система с MIPS процесор и разпределението на адресното пространство в нея.

В таблица 1 са описани регистрите с общо предназначение, достъпни в процесор, реализиращ системата инструкции MIPS.

При писане на асемблер номерът на регистъра се предшества от знака '$'. Например регистър *t1* в инструкциите на асемблер трябва да се изпише като *$9.*



Фиг. 3. Разпределение на регистрите и паметта в MIPS

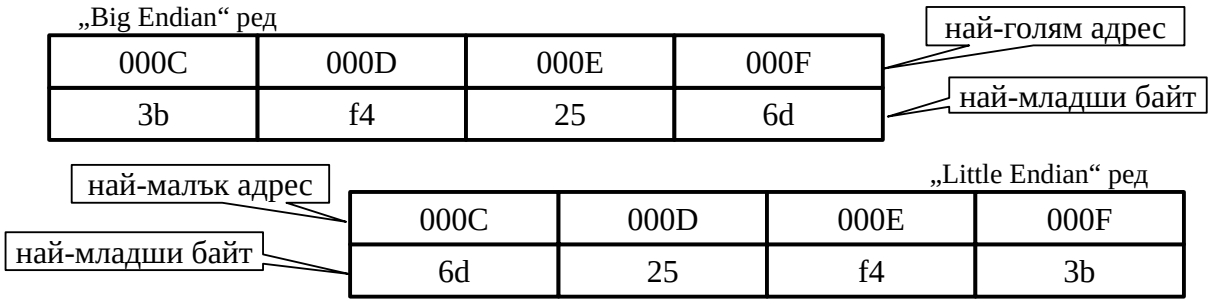
Табл. 1. Регистри с общо предназначение достъпни в MIPS

Регистър №	***Име***	***Употреба***
0	zero	константа 0
1	at	запазен за програмата за асемблиране
2-3	v0-v1	върнати стойности от извикани процедури
4-7	a0-a3	аргументи предавани към извикани процедури
8-15	t0-t7	временни стойности
16-23	s0-s7	съхранени стойности
24-25	t8-t9	временни стойности
26-27	k0-k1	запазени за ядрото на операционната система
28	gp	указател за адресиране на глобална памет
29	sp	указател за адресиране на стек
30	fp	указател за адресиране на кадър
31	ra	адрес на връщане

Ред на байтовете в паметта

Клетките в паметта са еднобайтови. По-големите формати полудума, дума и двойна дума могат да се записват в клетките от паметта или в намаляващ (от старши към младши байт) – Big Endian ред или в нарастващ (от младши към старши байт) – Little Endian ред. На

фиг. 4 е показано шестнайсетчното число 3bf4256d, записано в нарастващ и намаляващ ред в 4 клетки от паметта с адреси 000C, 000D, 000E и 000F.



Фиг. 4. Подреждане на данните в паметта

Групи инструкции в системата MIPS

Инструкциите са организирани в следните функционални групи:

1. Работа с паметта (load и store)
2. Аритметични и логически операции (ALU)
3. Предаване на управлението и преходи (jump и branch)
4. Други

Достъпът до паметта се осъществява само с инструкции load и store. Всички останали инструкции работят с операнди записани в регистрите на процесора.

З*ареждане на данни от паметта в регистрите на процесора: lb, lw*

lw -- load word – зарежда 4 байта от паметта в един от регистрите на процесора.

lb -- load byte – зарежда 1 байт от паметта в най-младшите 8 бита в регистър на процесора.

Запис на данни от регистрите на процесора в паметта: sb, sw

sw -- store word – записва съдържанието на регистър (една дума) в паметта.

sb -- store byte – записва най-младшия байт от регистър в паметта.

З*ареждане на литерал (непосредствена стойност) в регистър: li, lui*

li -- load immediate – зарежда 32-битова непосредствена стойност в регистър

lui – load upper immediate – зарежда 16-битова непосредствена стойност в старшата (горна) половина на регистър

Пример:

```
.data               # начало на данните в паметта
var1:  .word   23   # заделя 4 байта от паметта за var1 и задава начална стойност 23
.text               # начало на инструкциите в паметта
start:              # указва, че следва първата инструкция в програмата
       lw $8, var1  # зарежда заделените 4 байта с име var1 в регистър t0, t0 ← var1
       li $9, 5     # зарежда непосредствената стойност 5 в регистър t1, t1 ← 5
       sw $9, var1  # съхранява съдържанието на регистър t1 в заделените 4 байта от
                    # паметта с име var1, var1 ← t1
```

Индиректно и индексно адресиране

Използва се само с инструкции load и store.

Зареждане на адрес: la $10, var1 – копира адреса на var1 (която предварително е дефинирана в програмата) в регистър $10 (t2).

Индиректно адресиране: lw $8, ($10) – зарежда дума от адрес в RAM паметта, записан в регистър *$10*, в регистър *$8.*

sw $9, ($10) – записва стойността на регистъра $9 в RAM паметта на адреса, съдържащ се в регистър $10

С отместване или индексно адресиране: l*w $8, 4($10)* – зарежда дума от адрес *($10+4)* от RAM паметта в регистър *$8*. „*4“* е отместването от адреса в регистъра *$10.*

sw $t2, -12($t0) – Съхранява думата, съдържаща се в регистър $9 в RAM паметта на адрес ($10 – 12). Отрицателното отместване също е разрешено.

Индексното адресиране е полезно при: работа с масиви – достъпът до елементите е отместването от базовия адрес; използване на стек – лесен достъп до елементите в стека като отместване от указателя, сочещ началото.

Пример:

```
.data
 array1: .space 12          # заделя 12 байта от паметта за масив от 3 цели числа
.text
start: la $8, array1        # зарежда базовия адрес на масива в регистър $8
       li $9, 5             # $9 ← 5 (зарежда „непосредствено“ 5 в регистър 9)
       sw $9, ($8)          # в първия елемент на масива „индиректно“ се записва 5
       li $9, 13            # $9 ← 13
       sw $9, 4($8)         # във втория елемент на масива „индексно“ се записва 13
       li $9, -7            # $9 ← -7
       sw $9, 8($8)         # в третия елемент на масива „индексно“ се записва -7
```

За по-лесното им запомняне инструкциите за аритметични и логически операции могат да се разделят на следните групи:

Инструкции с 16-битов „непосредствен“ операнд		***Инструкции с три операнда***	
ADDI	Събиране с литерал	ADD	Събиране
ADDIU	Събиране с литерал без знак	ADDU	Събиране на стойности без знак
ANDI	Логическо „И“ с литерал	AND	Логическо „И“
XORI	Изключващо „ИЛИ“ с литерал	NOR	Логическо „ИЛИ-НЕ“
ORI	Логическо „ИЛИ“ с литерал	OR	Логическо „ИЛИ“
SLTI	Сравнение (по-малко от) с литерал	SLT	Сравнение (по-малко от)
SLTIU	Сравнение с литерал без знак	SLTU	Сравнение на стойности без знак
Инструкции за логическо изместване		SUB	Изваждане
SLL, SLLV	Логическо изместване на ляво	SUBU	Изваждане на стойности без знак
SLA	Аритметично изместване на ляво	XOR	Изключващо „ИЛИ“
SRL, SRLV	Логическо изместване на дясно	***Инструкции с два операнда***	
SRA	Аритметично изместване на дясно	MULT, MULTU	Умножение, Умножение без знак
		DIV, DIVU	Деление, Деление без знак

В следващите таблици са описани различните групи инструкции.

Аритметични инструкции

Инструкция	**Пример**	**Значение**	**Пояснение**
Събиране	add $1,$2,$3	$1 ← $2+$3	
Изваждане	sub $1,$2,$3	$1 ← $2-$3	
Събиране с литерал	addi $1,$2,100	$1 ← $2+100	Литерал означава твърдо зададено, константно число в кода.
Събиране на числа без знак	addu $1,$2,$3	$1 ← $2+$3	Стойностите се интерпретират като цели числа без знак. За разлика от целите числа със знак, които ако са отрицателни са в допълнителен код.
Изваждане на числа без знак	subu $1,$2,$3	$1 ← $2-$3	Стойностите се интерпретират като цели числа без знак. За разлика от целите числа със знак.
Събиране с литерал без знак	addiu $1,$2,100	$1 ← $2+100	Стойностите се интерпретират като цели числа без знак. За разлика от целите числа със знак.
Умножение	mult $2,$3	$hi,$low ← $2*$3	Старшите 32 бита се съхраняват в специалния регистър *hi*, а младшите 32 бита в – *lo*.
Деление	div $2,$3	$hi,$low ← $2/$3	Остатъкът се съхраняват в специалния

			регистър *hi*, а частното в – *lo*.

Логически инструкции

Инструкция	Пример	Значение	Пояснение
И	and $1,$2,$3	$1 ← $2&$3	Побитово И
ИЛИ	or $1,$2,$3	$1 ← $2|$3	Побитово ИЛИ
И с литерал	andi $1,$2,100	$1 ← $2&100	Побитово И с непосредствена стойност.
ИЛИ с литерал	or $1,$2,100	$1 ← $2|100	Побитово ИЛИ с непосредствена стойност.
Логическо преместване наляво	sll $1,$2,10	$1 ← $2<<10	Преместване наляво със зададен константен брой битове.
Логическо преместване надясно	srl $1,$2,10	$1 ← $2>>10	Преместване надясно със зададен константен брой битове.

Работа с паметта

Инструкция	Пример	Значение	Пояснение
Зареждне на дума	lw $1,100($2)	$1 ← Memory[$2+100]	Копира от паметта в регистър.
Съхранение на дума	sw $1,100($2)	Memory[$2+100] ← $1	Копира от регистър в паметта.
Зареждане в старшата половина на литерал	lui $1,100	$1 ← 100x$2^{16}$	Зарежда константа в старшите 16 бита. В младшите 16 бита се записват нули.
Зареждане на адрес	la $1,label	$1 ← адрес на label	Псевдо инструкция. Изчислява адреса на label и го зарежда в регистър (адресът на label, не стойността).
Зареждане на литерал	li $1,100	$1 ← 100	Псевдо инструкция. (Осигурява се от програмата за асемблиране не от процесора.) Зарежда константна стойност в регистър.
Преместване от hi	mfhi $2	$2 ← hi	Копира от специалния регистър hi в регистър с общо предназначение
Преместване	mflo $2	$2 ← lo	Копира от специалния регистър lo в регистър с общо предназначение
Пеместване	move $1,$2	$1 ← $2	Псевдо инструкция. (Осигурява се от програмата за асемблиране не от процесора.) Копира от регистър в регистър.

Съществуват варианти на load и store с по-малки размери на данните: *lh u sh* – за 16-битова полудума и *lb u sb*– за байт (8 бита).

Условни преходи

Инструкция	Пример	Значение	Пояснение
Преход при равен резултат	beq $1,$2,100	if($1==$2) go to PC+4+100	Проверява дали стойностите в два регистъра са равни
Преход при не равен резултат	bne $1,$2,100	if($1!=$2) go to PC+4+100	Проверява дали стойностите в два регистъра не са равни
Преход при по-голямо от	bgt $1,$2,100	if($1>$2) go to PC+4+100	Псевдо инструкция.
Преход при по-голямо или равно на	bge $1,$2,100	if($1>=$2) go to PC+4+100	Псевдо инструкция.
Преход при по-малко от	blt $1,$2,100	if($1<$2) go to PC+4+100	Псевдо инструкция.

Преход при по-малко или равно на	ble $1,$2,100	if($1<=$2) go to PC+4+100	Псевдо инструкция.

Всички условни преходи сравняват стойностите на два регистъра. Ако резултатът от проверката е истина, преходът се осъществява (т.е. процесорът продължава изпълнението от указания в инструкцията за преход адрес). В противен случай процесорът продължава със поредната инструкция в програмата.

Пояснение: По-лесно е да се използва етикет в инструкциите за преход, вместо числов адрес. Например: *beq $t0, $t1, equal.* Етикетът *„equal“* трябва да бъде деклариран някъде на друго място в кода.

Сравнение

Инструкция	Пример	Значение	Пояснение
Установяване при по-малко от	slt $1,$2,$3	if($2<$3)$1 ← 1; else $1 ← 0	Проверява дали даден регистър има по-малка стойност от друг. Ако е вярно установява $1 в 1. В противен случай, установява$1 в 0.
Установяване при по-малко от литерал	slti $1,$2,100	if($2<100)$1 ← 1; else $1 ← 0	Проверява дали даден регистър има по-малка стойност от константа. Ако е вярно установява $1 в 1. В противен случай, установява $1 в 0.

Безусловни преходи

Инструкция	Пример	Значение	Пояснение
Преход към адрес	j 1000	PC ← 1000	Изпълнението а програмата продължава от посочения адрес.
Преход към адрес в регистър	jr $1	PC ← $1	Изпълнението а програмата продължава от адреса, съдържащ се в посочения регистър.
Преход към адрес и запазване на адреса за връщане	jal 1000	$ra ← PC+4; PC ← 1000	Използва се когато се прави извикване на процедура. Адреса за връщане от процедурата се съхранява в $ra.

Директиви на асемблер

Директивите позволяват да се задава какво да направи програмата за асемблиране когато конвертира сорс кода в двоичен код.

Директиви на асемблер

Директива	Резултат
.word w1, ..., wn	Съхранява n на брой 32-битови стойности в последователни клетки от паметта. За всяка стойност се заделят 4 байта.
.half h1, ..., hn	Съхранява n на брой 16-битови стойности в последователни клетки от паметта. За всяка стойност се заделят 2 байта.
.byte b1, ..., bn	Съхранява n на брой 8-битови стойности в последователни клетки от паметта. За всяка стойност се заделя по 1 байт.
.ascii str	Съхранява ASCII стринга *str* в паметта. Стринговете се ограждат в двойни кавички, напр. „*Computer Science*“
.asciiz str	Съхранява ASCII стринга *str* в паметта, като в края му се добавя *null*.
.space n	Оставя празна *n*-байтова област в паметта, която да се използва в последствие.
.align n	Подравнява следващата стойност в граници от 2^n байта. Например, *.align 2* подравнява следващата стойност в граници от 4 байта т.е. на машинната дума.

Формат на MIPS инструкциите

Процесорната инструкция е 32-битова дума подравнена в граници от 32-бита. Полетата използвани в процесорната инструкция са показани в следващата таблица.

Поле	Описание
КОП	Код на операцията. Най-старшите 6 бита от 32 битовия код на инструкцията.
ФОП	Функционален операционен код. Ако КОП=0, ФОП определя конкретната функция, изпълнявана от инструкцията. Най-младшите 6 бита от кода на инструкцията.
ЦР	Целеви регистър. Пет битово поле, което определя номера на регистъра, в който ще се запише резултатът от операцията.
ИР	Изходен регистър. Пет битово поле, което определя номер на регистър, от който ще се вземе стойността на входен операнд.
ИЗМ	Изместване. Използва се при инструкциите за побитово изместване. Пет битово поле. Показва с колко бита да се измести стойността е ИР.
НС	Непосредствена стойност. Константа записана вътре в кода на самата инструкция (за разлика от константите в паметта, за достъпа до които е необходимо използването на инструкция load).
ОТМ	Отместване. Непосредствена константа в инструкцията, която се използва за да се формира адрес в паметта, спрямо базов адрес или спрямо програмния брояч (PC). Най-често полето е 16-битово, но в някой инструкции е 9-, 18-, 19-, 21- и 26-битово.
ИНД	Индекс. Използва се при инструкциите за безусловен преход. Стойността в това поле е 26-битова. Тя се измества наляво с 2 бита, за да се осигурят младшите 28-бита на адреса от който трябва да продължи програмата.

Налични са три различни формата:

- формат от тип *R;*
- формат от тип *I;*
- формат от тип *J.*

Тип R – Операндите са три регистъра

КОП	ИР1	ИР2	ЦР	ИЗМ	ФОП
6 бита	5 бита	5 бита	5 бита	5 бита	6 бита

Пример: ***add t0, s1, s2*** ***# t0 ← s1+s2***

0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

Тип I – Операндите са три – два регистъра и непосредствен операнд – стойността му е в самата инструкция. Аритметични инструкции и инструкции load/store

КОП	ИР	ЦР	НС или ОТМ
6 бита	5 бита	5 бита	16 бита

Пример: ***addi s0, s1, 2*** ***# s0 ← s1+2***

8	17	16	2
001000	10001	10000	0000 0000 0000 0010

Тип J – Операндът е един, съдържа адрес за преход. Инструкции j, jal

КОП	ИНД
6 бита	26 бита

Пример: ***j 0x020004 # pc ← 0x020004<<2***

2	131076
000010	0000 0010 0000 0000 0000 0100

Програма за събиране на числа за учебен RISC компютър със система инструкции MIPS

```
.set noreorder                # Предотвратява пренареждането на инструкциите
.text                         # Начало на сегмента в паметта с инструкции
.globl start                  # Етикетът start трябва да е глобален
.ent start                    # Маркира входната точка в програмата

start:      li $8, 0x1        # Зарежда непосредствената стойност 1
            li $9, 0x1        # Зарежда непосредствената стойност 1
            add $10, $8, $9   # Събира стойностите
.end start                    # Маркира края на програмата
```

Първите четири реда от програмата са директиви, които казват на програмата за асемблиране как трябва да бъде транслиран кода. Следващите три реда, започвайки от етикета *start* са машинни инструкции, които зареждат две числа (в случая две единици) в 8-ми и 9-ти регистър (t0 и t1). След което стойностите в двата регистъра се събират и резултата се записва в 10-ти регистър (t2).

Софтуер за симулация на RISC компютър със система инструкции MIPS

MipsIt е софтуер за симулация на работата на компютър с класически RISC процесор със система инструкции MIPS-32. Той позволява да се създават програми на асемблер или C, които да се компилират до машинен код, да се зареждат в паметта на MIPS симулатора и да се изпълняват от него. MipsIt се състои от: MipsIt Studio – основната програма, в която програмите се въвеждат, компилират и подготвят за симулация; Mips Simulator – симулира работата на система с MIPS процесор. Двоичният код на програмата се зарежда в симулатора и се изпълнява. Mips симулатора е много по-бавен от реален компютър, но позволява създаването и тестването на програми на асемблер да се извършва без да се закупува реален хардуер.

Ако MipsIt не е инсталиран може да се изтегли, за да се инсталира от MipsICT.exe.zip.

Пояснение: Работи с Windows 7 или по-ниска версия.

Друг софтуер за симулация на работата на процесор с MIPS инструкции, който може да се използва за реализация на задачите в това упражнение е MARS.

Важно! Програмата DrMIPS е специално създадена за подпомагане на обучението по компютърни архитектури. В сегашната ситуация на отдалечено онлайн обучение с предимство може да се използва DrMIPS.

Програмиране на асемблер с използване на MipsIt

Стартирайте MipsIt от C:\MipsIt\Bin\MipsIt.exe. Създайте нов проект (File->New->Project ->Assembler) и нов файл в проекта (File->New->File->Assembler), въведете кода и компилирайте (Build->Compile или Ctrl+F7). Отстранете грешките, ако се появят такива и направете програмата готова за изпълнение (Build->Build или F7).

Стартирайте симулатора Mips.exe от директорията C:\MipsIT\bin\. Върнете се в прозореца на MipsIt и заредете подготвената за изпълнение програма в симулатора (Build -> Upoad -> To Simulator или F5). Отворете прозореца на регистрите (щракнете върху блока CPU), за да виждате стойностите в тях. Отворете прозореца на RAM паметта (щракнете върху блока RAM), за да видите адресите, на които е заредена програмата. В прозореца на паметта щракнете двукратно върху адрес, които е след последната инструкция в програмата. Изпълнете програмата (Cpu->Run). За да проследите как се изпълнява програмата и как се променят стойностите в регистрите, изпълнявате програмата постъпково (Cpu->Step).

III. Задачи за изпълнение:

Задача 1: Напишете програма на асемблер, която изчислява резултата от израза: Z=X+Y. Нека X се съхранява в паметта, докато Y се съхранява в регистър.

Задача 2: Напишете програма на асемблер, която реализира следното:

```
if X>Y then
   Z=X
else
   Z=Y
end
```

Нека X, Y и Z се съхраняват в паметта.

Задача 3: Напишете програма на асемблер, която реализира следното:

```
Summa=0;
For I=1 to 10 begin
    Summa=Summa+2;
End
```

Задача 4: Напишете програма на асемблер, която изчислява сумата на числата до N, т.е. за дадено N изчислява 1+2+3+4+.....+N.

Задача 5: Напишете програма на асемблер, която изчислява колко единици има в даден байт.

Задача 6: Напишете програма на асемблер, която изчислява най-големия общ делител (НОД). НОД е най-голямото цяло число, което дели две цели числа по-големи от нула. Например, НОД(12,18)=6. За повече подробности прочетете в Wikipedia.

Забележка: Изпратете написаните от вас програми на имейл pminev@tugab.bg, като посочите две имена, курс и подгрупа.