Тема 15. Архитектуры вычислительных машин и языки ассемблера, часть 4

15.1 Общие сведения

Мы переходим к изучению вычислительной архитектуры MIPS32, которая относится к семейству RISC. Далее представлен частичный список ассемблерных команд. За полным описанием рекомендуется обратиться к англоязычной официальной документации.

Структура программы на MIPS-ассемблере и многие обозначения аналогичны использовавшимся при изучении GNU/Assembler, поэтому далее мы, главным образом, сосредотачиваемся на отличительных особенностях.

Далее символами $1, $2, $3 обозначены «номера» регистров. Очевидно, при написании программ следует использовать названия регистров, а не их номера (эта проблема обсуждается далее в пп.15.6). Также встречается термин «псевдоинструкция», т.е. такая команда, которая есть в ассемблере, но нет в "железе".

15.2 Арифметические операции

Некоторые команды для выполнения арифметических операций представлены в следующей таблице (СЛАЙДЫ 2 и 3).

Инструкция	Пример	Семантика	Комментарии
add	add $1, $2, $3	$1 = $2 + $3	
subtract	sub $1, $2, $3	$1 = $2 - $3	
add immediate	addi $1, $2, 100	$1 = $2 + 100	*immediate* - константа
add unsigned	addu $1, $2, $3	$1 = $2 + $3	Значения рассматриваются как беззнаковые, без дополнительного кода
subtract unsigned	subu $1, $2, $3	$1 = $2 - $3	Значения рассматриваются как беззнаковые, без дополнительного кода
add immediate unsigned	addiu $1, $2, 100	$1 = $2 + 100	Значения рассматриваются как беззнаковые, без дополнительного кода
multiply (without overflow)	mul $1, $2, $3	$1 = $2 * $3	Результат занимает 32 бита
multiply	mult $2, $3	{$hi, $low} = = $2 * $3	Верхние 32 бита размещаются в специальном регистре $hi, нижние - в специальном регистре $low
divide	div $2, $3	{$hi, $low) = = $2 / $3	Остаток будет сохранен в специальном регистре $hi, а частное - в $low
and	and $1, $2, $3	$1 = $2 & $3	Побитовое И
or	or $1, $2, $3	$1 = $2 | $3	Побитовое ИЛИ
and immediate	andi $1, $2, 200	$1 = $2 & 200	Побитовое И с непосредственным значением

or immediate	ori $1, $2, 200	$1 = $2 | 200	Побитовое ИЛИ с непосредственным значением
shift left logical	sll $1, $2, 10	$1 = $2 << 10	Сдвиг влево на заданное число битов
shift right logical	srl $1, $2, 10	$1 = $2 >> 10	Сдвиг вправо на заданное число битов

15.3 Копирование

Инструкция	Пример	Семантика	Комментарии
load word	lw $1, 100($2)	$1 = Memory[$2 + 100]	Копирование из памяти в регистр
store word	sw $1, 100($2)	Memory[$2 + 100] = $1	Копирование из регистра в память
load upper immediate	lui $1, 200	$1 = 200 * 2 ^ 16	Загрузить константу в верхние 16 битов. Нижние 16 битов устанавливаются в 0.
load address	la $1, label	$1 = Address of label	Псевдоинструкция для загрузки адреса метки в регистр
load immediate	li $1, 300	$1 = 300	Псевдоинструкция для загрузки непосредственного значения в регистр
move from hi	mfhi $2	$2 = $hi	Копировать из специального регистра $hi в регистр общего назначения
move from low	mflo $3	$3 = $low	Копировать из специального регистра $low в регистр общего назначения
move	move $1, $2	$1 = $2	Псевдоинструкция для копирования из регистра в регистр

См. СЛАЙД 4. Имеются вариации для загрузки и сохранения данных меньшего размера: *lh* и *sh* для 16-битных слов, *lb* и *sb* — для 8-битных байтов.

15.4 Сравнения и переходы

Все инструкции сравнивают значения из двух регистров. Если условие проверки истинно, то выполняется переход, в противном случае - выполняется следующая инструкция (СЛАЙД 5).

Инструкция	Пример	Семантика	Комментарии
branch on equal	beq $1, $2, 100	if ($1 == $2) goto PC + 4 + 100	Переход, если значения в регистрах равны
branch on not equal	bne $1, $2, 100	if ($1 != $2) goto PC + 4 + 100	Переход, если значения в регистрах не равны
branch on greater than	bgt $1, $2, 100	if ($1 > $2) goto PC + 4 + 100	Псевдоинструкция
branch on greater than or equal	bge $1, $2, 100	if ($1 >= $2) goto PC + 4 + 100	Псевдоинструкция
branch on less than	blt $1, $2, 100	if ($1 < $2) goto PC + 4 + 100	Псевдоинструкция
branch on less than or equal	ble $1, $2, 100	if ($1 <= $2) goto PC + 4 + 100	Псевдоинструкция

Версия 0.9pre-release от 28.04.2014. Возможны незначительные изменения.

Вместо абсолютного значения метки рекомендуется указывать символьное. Например: *beq* $*t*0, $*t*1, *lab_eq*. Очевидно, метка *lab_eq* должна быть установлена в другом месте кода.

Существует большое количество вариаций команд сравнения и установки значения регистра из приведенной далее таблицы, которые иногда упрощают код. Рекомендуется ознакомиться с ними в официальной англоязычной документации (СЛАЙД 6).

Инструкция	Пример	Семантика	Комментарии
set on less than	slt $1, $2, $3	if ($2 < $3) $1 = 1; else $1 = 0;	Сравнить. Если меньше, установить $1 в 1, в противном случае $1 устанавливается в 0
set on less than immediate	slti $1, $2, 300	if ($2 < 300) $1 = 1; else $1 = 0;	Сравнить. Если меньше, установить $1 в 1, в противном случае $1 устанавливается в 0

Существует большое количество вариаций команд переходов из приведенной далее таблицы, которые иногда упрощают код. Рекомендуется ознакомиться с ними в официальной англоязычной документации.

Инструкция	Пример	Семантика	Комментарии
jump	j 1000	goto 1000;	Переход на целевой адрес
jump register	jr $1	goto *($r1);	Переход на адрес в регистре $1. Используется для операторов множественного выбора или возврата значений из подпрограмм
jump and link	jal 1000	$ra = PC + 4; goto 1000;	Используется при вызове подпрограмм, в $ra сохраняется адрес возврата

Вместо абсолютного значения метки рекомендуется указывать символьное. Например: *j loop*_1. Очевидно, метка *loop*_1 должна быть установлена в другом месте кода.

15.5 Ассемблерные директивы

Директивы ассемблеру предназначены для выполнения некоторых действий при «ассемблировании».

Некоторые директивы (СЛАЙД 7):

Директива	Результат
.word w1, ..., wn	Зарезервировать непрерывный блок памяти для *n* 32-битных целых чисел
.half h1, ..., hn	Зарезервировать непрерывный блок памяти для *n* 16-битных целых чисел
.byte b1, ..., bn	Зарезервировать непрерывный блок памяти для *n* 8-битных целых чисел
.ascii str	Зарезервировать блок памяти для ASCII-строки, которая должна заключаться в кавычки. Пример — "System Software"
.asciiz str	Зарезервировать блок памяти для ASCII-строки, заканчивающейся символом с кодом 0. Эти строки заключаются в кавычки
.space n	Зарезервировать пустой регион памяти из *n* байтов для последующего использования
.align n	Выровнять следующие данные по границе 2^*n*. Например, .align 2 выровняет следующее значение по границе 32-битного слова

15.6 Регистры

В архитектуре MIPS-32 предусмотрены регистры общего назначения в количестве 32 штук. Технически они могут использоваться для любых нужд программиста. Тем не менее, по соглашению регистры делятся на несколько групп и используются для разных задач. У регистров есть аппаратный «номер» и ассемблерное «имя».

В следующей таблице дано только краткое описание регистров общего назначения. За подробным описанием рекомендуется обратиться к официальной документации (СЛАЙД 8).

Номер регистра	Имя регистра	Описание
0	$zero	Значение 0
2-3	$v0 - $v1	Используются для вычисления выражений и возврата значений из подпрограмм
4-7	$a0 - $a3	Первые четыре аргумента подпрограммы
8-15, 24-25	$t0 - $t9	Временные переменные
16-23	$s0 - $s7	Значения, сохраняемые вызываемой подпрограммой
31	$ra	Адрес возврата из подпрограммы
1	$at	Зарезервирован для ассемблера
26-27	$k0 - $k1	Зарезервированы для обработчиков прерываний и ловушек
28	$gp	Глобальный указатель на середину 64-килобайтного блока в сегменте статической памяти
29	$sp	Указатель стека
30	$s8 / $fp	Сохраненное значение / указатель стекового фрейма

15.7 Системные вызовы

Для выполнения программы мы используем симулятор SPIM. Он допускает несколько полезных системных вызовов. Все они симулируются и не входят в систему команд MIPS. Системные вызовы выполняются операционным окружением непосредственно либо через стандартную библиотеку функций.

Системные вызовы используются для задач ввода-вывода, а также завершения выполнения. Все они инициируются командой *syscall* и требуют параметров, передаваемых через регистры $*v*0, $*a*0-$*a*1, $*f*12. Это зависит от конкретного системного вызова. Иными словами, системные вызовы используют не все регистры. Результат системного вызова возвращается через регистр $*v*0 для целых чисел и регистр $*f*0 для чисел с плавающей точкой.

В SPIM доступны следующие системные вызовы (СЛАЙДЫ 9-11):

Сервис	Операция	Код (в регистре $v0)	Аргументы	Результаты
print_int	Вывести 32-битное целое	1	В $a0 целое число	Нет
print_float	Вывести 32-битное «плавающее»	2	В $f12 число с плавающей точкой	Нет
print_double	Вывести 64-битное «плавающее»	3	В $f12 число с плавающей точкой	Нет
print_string	Вывести строку, концом которой является символ с кодом 0	4	В $a0 адрес строки в памяти	Нет
read_int	Получить от пользователя 32-битное целое	5	Нет	Целое число в $v0
read_float	Получить	6	Нет	Число в

	от пользователя 32-битное число с плавающей точкой			регистре $f0
read_double	Получить от пользователя 64-битное число с плавающей точкой	7	Нет	Число в $f0
read_string	Аналогично стандартной C-функции fgets	8	$a0 - адрес буфера в памяти для строки; $a1 - размер буфера	Нет
sbrk	Аллокация и деаллокация памяти из n байтов	9	$a0 = n, где n - количество байтов. Отрицательное число - освобождение памяти, неотрицательное - распределение памяти.	Адрес участка памяти в $v0
exit	Снятие программы с выполнения	10	Нет	Нет
print_char	Вывод символа	11	В $a0 символ, который нужно вывести	Нет
read_char	Получение символа от пользователя	12	Нет	В $v0 - введенный символ
open	Открыть файл	13	В $a0 - адрес буфера с именем файла, в $a1 - флаги, в $a2 — режим	В $v0 - дескриптор открытого файла
read	Чтение из файла	14	В $a0 — дескриптор файла, в $a1 — адрес буфера, в $a2 — размер буфера	В $v0 - количество прочитанных байтов
write	Запись в файл	15	В $a0 — дескриптор файла, в $a1 — адрес буфера, в $a2 — размер буфера	В $v0 - количество записанных байтов
close	Закрыть файл	16	В $a0 — дескриптор файла	Нет
exit2	Прерывание программы с возвратом значения	17	В $a0 - целое число	Нет

Вызовы с 13 по 16 возвращают -1 в случае ошибки.

15.8 Примеры кода

Пример 1. Базовая арифметика (СЛАЙД 12)

```
# A demonstration of some simple MIPS instructions
# used to test SPIM

      # Declare main as a global function
      .globl main

      # All program code is placed after the
      # .text assembler directive
      .text

# The label 'main' represents the starting point
main:
      li $t2, 25        # Load immediate value (25)
      lw $t3, value           # Load the word stored at label 'value'
      add $t4, $t2, $t3 # Add
```

```
      sub $t5, $t2, $t3 # Subtract

      # Exit the program by means of a syscall.
      # There are many syscalls - pick the desired one
      # by placing its code in $v0. The code for exit is "10"
      li $v0, 10 # Sets $v0 to "10" to select exit syscall
      syscall # Exit

      # All memory structures are placed after the
      # .data assembler directive
      .data

      # The .word assembler directive reserves space
      # in memory for a single 4-byte word (or multiple 4-byte words)
      # and assigns that memory location an initial value
      # (or a comma separated list of initial values)
value:      .word 12
```

Пример 2. Выдача сообщения (СЛАЙД 13)

```
# "Hello World" in MIPS assembly
# From: http://labs.cs.upt.ro/labs/so2/html/resources/nachos-doc/mipsf.html

      # All program code is placed after the
      # .text assembler directive
      .text

      # Declare main as a global function
      .globl      main

# The label 'main' represents the starting point
main:
      # Run the print_string syscall which has code 4
      li    $v0,4       # Code for syscall: print_string
      la    $a0, msg    # Pointer to string (load the address of msg)
      syscall
      li    $v0,10            # Code for syscall: exit
      syscall

      # All memory structures are placed after the
      # .data assembler directive
      .data

      # The .asciiz assembler directive creates
      # an ASCII string in memory terminated by
      # the null character. Note that strings are
      # surrounded by double-quotes
msg:  .asciiz     "Hello World!\n"
```

Пример 3. Ввод-вывод и арифметика (СЛАЙДЫ 14-15)

```
# Simple input/output in MIPS assembly
# From: http://labs.cs.upt.ro/labs/so2/html/resources/nachos-doc/mipsf.html

      # Start .text segment (program code)
      .text

      .globl      main
main:
      # Print string msg1
      li    $v0,4       # print_string syscall code = 4
      la    $a0, msg1   # load the address of msg
      syscall

      # Get input A from user and save
```

```
      li    $v0,5       # read_int syscall code = 5
      syscall
      move  $t0,$v0           # syscall results returned in $v0

      # Print string msg2
      li    $v0,4       # print_string syscall code = 4
      la    $a0, msg2   # load the address of msg2
      syscall

      # Get input B from user and save
      li    $v0,5       # read_int syscall code = 5
      syscall
      move  $t1,$v0           # syscall results returned in $v0

      # Math!
      add   $t0, $t0, $t1     # A = A + B

      # Print string msg3
      li    $v0, 4
      la    $a0, msg3
      syscall

      # Print sum
      li    $v0,1       # print_int syscall code = 1
      move  $a0, $t0    # int to print must be loaded into $a0
      syscall

      # Print \n
      li    $v0,4       # print_string syscall code = 4
      la    $a0, newline
      syscall

      li    $v0,10            # exit
      syscall

      # Start .data segment (data!)
      .data
msg1: .asciiz     "Enter A:   "
msg2: .asciiz     "Enter B:   "
msg3: .asciiz     "A + B = "
newline:   .asciiz      "\n"
```

Пример 4. Циклы (СЛАЙДЫ 16-17)

```
# Simple routine to demo a loop
# Compute the sum of N integers: 1 + 2 + 3 + ... + N
# From: http://labs.cs.upt.ro/labs/so2/html/resources/nachos-doc/mipsf.html

      .text

      .globl      main
main:
      # Print msg1
      li    $v0,4       # print_string syscall code = 4
      la    $a0, msg1
      syscall

      # Get N from user and save
      li    $v0,5       # read_int syscall code = 5
      syscall
      move  $t0,$v0           # syscall results returned in $v0

      # Initialize registers
      li    $t1, 0            # initialize counter (i)
```

```
      li    $t2, 0            # initialize sum

      # Main loop body
loop: addi  $t1, $t1, 1 # i = i + 1
      add   $t2, $t2, $t1     # sum = sum + i
      beq   $t0, $t1, exit    # if i = N, continue
      j     loop

      # Exit routine - print msg2
exit: li    $v0, 4            # print_string syscall code = 4
      la    $a0, msg2
      syscall

      # Print sum
      li    $v0,1       # print_string syscall code = 4
      move  $a0, $t2
      syscall

      # Print newline
      li    $v0,4       # print_string syscall code = 4
      la    $a0, lf
      syscall
      li    $v0,10            # exit
      syscall

      # Start .data segment (data!)
      .data
msg1: .asciiz     "Number of integers (N)?  "
msg2: .asciiz     "Sum = "
lf:     .asciiz   "\n"
```

Пример 5. Подпрограммы без использования стека (СЛАЙДЫ 18-20)

```
# Simple routine to demo functions
# NOT using a stack in this example.
# Thus, the function does not preserve values
# of calling function!

# ---------------------------------------------------------------

      .text

      .globl      main
main:
      # Register assignments
      # $s0 = x
      # $s1 = y

      # Initialize registers
      lw    $s0, x            # Reg $s0 = x
      lw    $s1, y            # Reg $s1 = y

      # Call function
      move  $a0, $s0    # Argument 1: x ($s0)
      jal   fun         # Save current PC in $ra, and jump to fun
      move  $s1,$v0           # Return value saved in $v0. This is y ($s1)

      # Print msg1
      li    $v0, 4            # print_string syscall code = 4
      la    $a0, msg1
      syscall

      # Print result (y)
      li    $v0,1       # print_int syscall code = 1
```

```
      move  $a0, $s1    # Load integer to print in $a0
      syscall

      # Print newline
      li    $v0,4       # print_string syscall code = 4
      la    $a0, lf
      syscall

      # Exit
      li    $v0,10            # exit
      syscall

# ---------------------------------------------------------------

      # FUNCTION: int fun(int a)
      # Arguments are stored in $a0
      # Return value is stored in $v0
      # Return address is stored in $ra (put there by jal instruction)
      # Typical function operation is:

fun:  # Do the function math
      li $s0, 3
      mul $s1,$s0,$a0         # s1 = 3*$a0  (i.e. 3*a)
      addi $s1,$s1,5          # 3*a+5

      # Save the return value in $v0
      move $v0,$s1

      # Return from function
      jr $ra                  # Jump to addr stored in $ra

# ---------------------------------------------------------------

      # Start .data segment (data!)
      .data
x:    .word 5
y:    .word 0
msg1: .asciiz     "y="
lf:     .asciiz   "\n"
```

Пример 6. Подпрограммы с использованием стека (СЛАЙДЫ 21-24)

Это рекомендованный способ.

```
# Simple routine to demo functions
# USING a stack in this example to preserve
# values of calling function

# ---------------------------------------------------------------

      .text

      .globl      main
main:
      # Register assignments
      # $s0 = x
      # $s1 = y

      # Initialize registers
      lw    $s0, x            # Reg $s0 = x
      lw    $s1, y            # Reg $s1 = y

      # Call function
      move  $a0, $s0    # Argument 1: x ($s0)
      jal   fun         # Save current PC in $ra, and jump to fun
```

```
      move  $s1,$v0           # Return value saved in $v0. This is y ($s1)

      # Print msg1
      li    $v0, 4            # print_string syscall code = 4
      la    $a0, msg1
      syscall

      # Print result (y)
      li    $v0,1       # print_int syscall code = 1
      move  $a0, $s1    # Load integer to print in $a0
      syscall

      # Print newline
      li    $v0,4       # print_string syscall code = 4
      la    $a0, lf
      syscall

      # Exit
      li    $v0,10            # exit
      syscall

# ---------------------------------------------------------------

      # FUNCTION: int fun(int a)
      # Arguments are stored in $a0
      # Return value is stored in $v0
      # Return address is stored in $ra (put there by jal instruction)
      # Typical function operation is:

fun:  # This function overwrites $s0 and $s1
      # We should save those on the stack
      # This is PUSH'ing onto the stack
      addi $sp,$sp,-4         # Adjust stack pointer
      sw $s0,0($sp)           # Save $s0
      addi $sp,$sp,-4         # Adjust stack pointer
      sw $s1,0($sp)           # Save $s1

      # Do the function math
      li $s0, 3
      mul $s1,$s0,$a0         # s1 = 3*$a0  (i.e. 3*a)
      addi $s1,$s1,5          # 3*a+5

      # Save the return value in $v0
      move $v0,$s1

      # Restore saved register values from stack in opposite order
      # This is POP'ing from the stack
      lw $s1,0($sp)           # Restore $s1
      addi $sp,$sp,4          # Adjust stack pointer
      lw $s0,0($sp)           # Restore $s0
      addi $sp,$sp,4          # Adjust stack pointer

      # Return from function
      jr $ra                  # Jump to addr stored in $ra

# ---------------------------------------------------------------

      # Start .data segment (data!)
      .data
x:    .word 5
y:    .word 0
msg1: .asciiz     "y="
lf:     .asciiz   "\n"
```

Эквивалентный код на языке C (СЛАЙД 25):

```
#include <stdio.h>

int function(int a);

int main()
{
  int x = 5;
  int y;
  y = function(x);
  printf("y=%i\n", y);
  return 0;
}

int function(int a)
{
  return 3 * a + 5;
}
```

Пример 7. Циклы с использованием подпрограмм и стека (СЛАЙДЫ 26-29)

Аналогично примеру 4, но с использованием подпрограмм:

```
# Simple routine to demo a loop
# Compute the sum of N integers: 1 + 2 + 3 + ... + N
# Same result as example4, but here a function performs the
# addition operation:  int add(int num1, int num2)

# ------------------------------------------------------------------

      .text

      .globl      main
main:
      # Register assignments
      # $s0 = N
      # $s1 = counter (i)
      # $s2 = sum

      # Print msg1
      li    $v0,4       # print_string syscall code = 4
      la    $a0, msg1
      syscall

      # Get N from user and save
      li    $v0,5       # read_int syscall code = 5
      syscall
      move  $s0,$v0           # syscall results returned in $v0

      # Initialize registers
      li    $s1, 0            # Reg $s1 = counter (i)
      li    $s2, 0            # Reg $s2 = sum

      # Main loop body
loop: addi  $s1, $s1, 1 # i = i + 1

      # Call add function
      move  $a0, $s2    # Argument 1: sum ($s2)
      move  $a1, $s1    # Argument 2: i ($s1)
      jal   add2        # Save current PC in $ra, and jump to add2
      move  $s2,$v0           # Return value saved in $v0. This is sum ($s2)
      beq   $s0, $s1, exit    # if i = N, continue
      j     loop

      # Exit routine - print msg2
exit: li    $v0, 4            # print_string syscall code = 4
```

```
        la    $a0, msg2
        syscall

        # Print sum
        li    $v0,1       # print_string syscall code = 4
        move  $a0, $s2
        syscall

        # Print newline
        li    $v0,4       # print_string syscall code = 4
        la    $a0, lf
        syscall
        li    $v0,10            # exit
        syscall

# -----------------------------------------------------------------

        # FUNCTION: int add(int num1, int num2)
        # Arguments are stored in $a0 and $a1
        # Return value is stored in $v0
        # Return address is stored in $ra (put there by jal instruction)
        # Typical function operation is:
        #  1.) Store registers on the stack that we will overwrite
        #  2.) Run the function
        #  3.) Save the return value
        #  4.) Restore registers from the stack
        #  5.) Return (jump) to previous location
        # Note: This function is longer than it needs to be,
        # in order to demonstrate the usual 5 step function process...

add2:   # Store registers on the stack that we will overwrite (just $s0)
        addi $sp,$sp, -4  # Adjust stack pointer
        sw $s0,0($sp)           # Save $s0 on the stack

        # Run the function
        add $s0,$a0,$a1         # Sum = sum + i

        # Save the return value in $v0
        move $v0,$s0

        # Restore overwritten registers from the stack
        lw $s0,0($sp)
        addi $sp,$sp,4          # Adjust stack pointer

        # Return from function
        jr $ra                  # Jump to addr stored in $ra

# -----------------------------------------------------------------

        # Start .data segment (data!)
        .data
msg1:   .asciiz     "Number of integers (N)?  "
msg2:   .asciiz     "Sum = "
lf:     .asciiz     "\n"
```

Литература и дополнительные источники к Теме 15

1. MIPS32 Architecture - https://imgtec.com/mips/architectures/mips32/
2. http://labs.cs.upt.ro/labs/so2/html/resources/nachos-doc/mipsf.html
3. http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm
4. http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html