Архитектура вычислительных систем

Лекция 6.

Яревский Е.А.

Кафедра вычислительной физики

Архитектура – язык ассемблера

Архитектура определяется
– **набором команд** (языком) и
– **местом нахождения операндов** (регистры и память).

Существует некоторое количество различных архитектур
(x86, ARM, MIPS, SPARC, PowerPC, …)

Архитектура **НЕ** определяет однозначно аппаратное обеспечение
(например, Intel / AMD).

Рассмотрим архитектуру MIPS (с некоторыми упрощениями).
Начальная версия разработана в Стэнфорде, в 1980е.
На 2014 год – более 3.5 млрд процессоров.
Использовалась, в частности, Silicon Graphics, Nintendo, Cisco …

Четыре принципа (Паттерсоном, Хеннесси):
(1) для простоты придерживайтесь единообразия;
(2) типичный сценарий должен быть быстрым;
(3) чем меньше, тем быстрее;
(4) хорошая разработка требует хороших компромиссов.

ЯЗЫК АССЕМБЛЕРА **MIPS**

Код на языке высокого уровня	**Код на языке ассемблера MIPS**
a = b + c;	*add a, b, c*

add – мнемоника, определяет операцию. Операция осуществляется над *b* и *c* (операндами-источниками, или просто операнды), а результат записывается в *a* (операнд-назначение, или результат).

a = b - c;	*sub a, b, c*

1й принцип – единообразие.

a = b + c - d;	*sub t, c, d # t = c − d* *add a, b, t # a = b + t*

2й принцип – типичный сценарий должен быть быстрым.
Небольшое число быстрых команд, пример reduced instruction set computer (RISC) архитектуры.

Операнды: регистры, память и константы

Рассматриваем 32-битную версию (существует и 64 бит вариант).
Чтение из памяти – медленное.
Архитектура MIPS использует 32 регистра, которые называют *набором регистров* или *регистровым файлом*.

3й принцип – чем меньше, тем быстрее.

Код на языке высокого уровня

a = b + c;

Код на языке ассемблера MIPS

```
# $s0 = a, $s1 = b, $s2 = c
add $s0, $s1, $s2   # a = b + c
```

MIPS обычно хранит переменные в 18 из 32 регистров: $s0–$s7 и $t0–$t9.
Регистры, имена которых начинаются на $s, называют *сохраняемыми (saved)* регистрами. В соответствии с соглашением об использовании регистров MIPS используются для размещения в них переменных. Имеют особое значение при вызове процедур.
Регистры, имена которых начинаются с $t, называют *временными (temporary)* регистрами. Они используются для хранения временных переменных.

a = b + c - d;

```
# $s0 = a, $s1 = b, $s2 = c, $s3 = d
sub $t0, $s2, $s3    # t = c − d
add $s0, $s1, $t0    # a = b + t
```

Табл. 6.1 **Набор регистров MIPS**

Название	Номер	Назначение
$0	0	Константный нуль
$at	1	Временный регистр для нужд ассемблера
$v0–$v1	2–3	Возвращаемые функциями значения
$a0–$a3	4–7	Аргументы функций
$t0–$t7	8–15	Временные переменные
$s0–$s7	16–23	Сохраняемые переменные
$t8–$t9	24–25	Временные переменные
$k0–$k1	26–27	Временные переменные операционной системы (ОС)
$gp	28	Глобальный указатель (англ.: global pointer)
$sp	29	Указатель стека (англ.: stack pointer)
$fp	30	Указатель кадра стека (англ.: frame pointer)
$ra	31	Регистр адреса возврата из функции

Память: 32 битовые слова,
32 битовые адреса.

Рис. 6.1 **Память с пословной адресацией**

ЗАГРУЗКА СЛОВА ИЗ ПАМЯТИ С ПОСЛОВНОЙ АДРЕСАЦИЕЙ
Код на языке ассемблера (**НЕ MIPS!**)

lw $s3, 1($0) # read memory word 1 into $s

ЗАПИСЬ СЛОВА В ПАМЯТЬ С ПОСЛОВНОЙ АДРЕСАЦИЕЙ
Код на языке ассемблера (**НЕ MIPS!**)

sw $s7, 5($0) # write $s7 to memory word 5

($0) – базовый адрес, можно использовать другие регистры.

MIPS использует **побайтовую** адресацию.

ДОСТУП К ПАМЯТИ С ПОБАЙТОВОЙ АДРЕСАЦИЕЙ
Код на языке ассемблера MIPS

```
lw $s0, 0($0)     # read data word 0 (0xABCDEF78) into $s0
lw $s1, 8($0)     # read data word 2 (0x01EE2842) into $s1
lw $s2, 0xC($0) # read data word 3 (0x40F30788) into $s2
sw $s3, 4($0)     # write $s3 to data word 1
sw $s4, 0x20($0) # write $s4 to data word 8
sw $s5, 400($0)   # write $s5 to data word 100
```

lb, sb – загрузка/запись байт

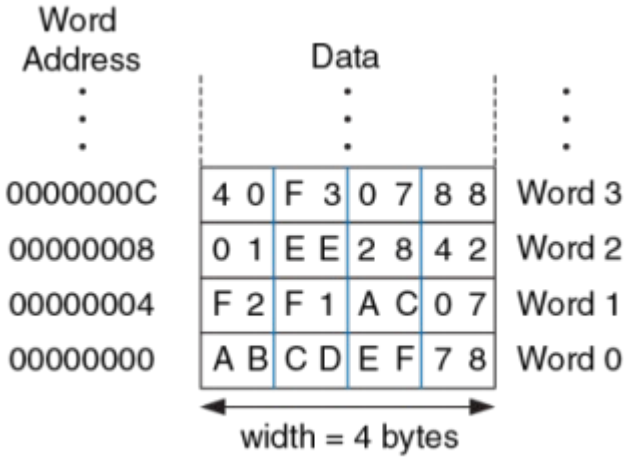


Рис. 6.2 Память с побайтовой адресацией

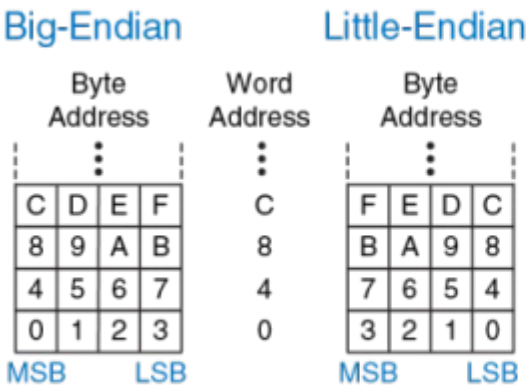


Рис. 6.3 **Адресация данных с прямым и обратным порядком байтов**

Разные процессоры MIPS используют разный порядок байт.

В архитектуре MIPS адреса слов для команд *lw* и *sw* должны быть *выровнены по словам* (*word aligned*), то есть адреса должны делиться на 4 без остатка.

Непосредственные операнды:

Код на языке высокого уровня

```
a = b + c;
a = a + 4;
b = a − 12;
```

Код на языке ассемблера MIPS

```
# $s0 = a, $s1 = b
addi $s0, $s0, 4      # a = a + 4
addi $s1, $s0, −12    # b = a – 12
```

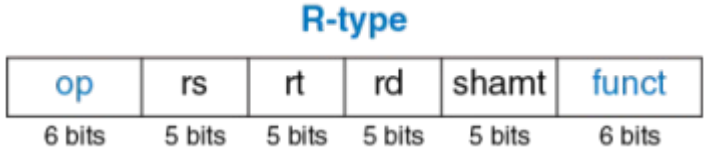
Константа, находящаяся внутри команды, является 16-битным числом, представленным в дополнительном коде, и может принимать значения из диапазона [–32,768; 32,767]. Команда *subi* отсутствует.

Форматы команд

Длина всех команд – 32 бита, некоторые используют не все биты.
Можно бы было сделать единый формат для **ВСЕХ** команд, но …
4й принцип – нужны хорошие компромиссы :)

В MIPS используются три формата команд:
1) **тип R.** Используют три регистровых операнда.
2) **тип I.** Используют два регистровых операнда и 16-битную константу.
3) **тип J** (от jump). Используют 26-битную константу.

Инструкции типа R



Операция закодирована двумя полями (синими):
полем *op* (*opcode,* код операции) и полем *funct* (функция).
У всех команд типа R поле *opcode* равно нулю.
Операция определяется исключительно полем *funct*.

Форматы команд

Операнды закодированы тремя полями: *rs*, *rt* и *rd*.
Поля содержат номера регистров (см. слайд 5).
Регистры *rs* и *rt* являются регистрами-источниками,
а *rd* – регистром-назначением (или регистром результата).

Поле *shamt* используется только для операций сдвига.
В таких командах значение, хранимое в 5-битном поле *shamt*, задаёт величину сдвига (shift amount).
У всех остальных команд типа R поле *shamt* равно 0.

Пример кодировки команд:

Assembly Code	Field Values						Machine Code						
	op	rs	rt	rd	shamt	funct	op	rs	rt	rd	shamt	funct	
`add $s0, $s1, $s2`	0	17	18	16	0	32	000000	10001	10010	10000	00000	100000	(0x02328020)
`sub $t0, $t3, $t5`	0	11	13	8	0	34	000000	01011	01101	01000	00000	100010	(0x016D4022)
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Табл. B.2 Инструкции типа R, отсортированные по полю funct

Funct	Имя	Описание	Операция
000000 (0)	`sll rd, rt, shamt`	логический сдвиг влево (shift left logical)	`[rd] = [rt] << shamt`
000010 (2)	`srl rd, rt, shamt`	логический сдвиг вправо (shift right logical)	`[rd] = [rt] >> shamt`
000011 (3)	`sra rd, rt, shamt`	арифметический сдвиг вправо (shift right arithmetic)	`[rd] = [rt] >>> shamt`
000100 (4)	`sllv rd, rt, rs`	логический переменный сдвиг влево (shift left logical variable)	`[rd] = [rt] << [rs]`$_{4:0}$
000110 (6)	`srlv rd, rt, rs`	логический переменный сдвиг вправо (shift right logical variable)	`[rd] = [rt] >> [rs]`$_{4:0}$
000111 (7)	`srav rd, rt, rs`	арифметический переменный сдвиг вправо (shift right arithmetic variable)	`[rd] = [rt] >>> [rs]`$_{4:0}$
001000 (8)	`jr rs`	безусловный переход по регистру (jump register)	`PC = [rs]`
001001 (9)	`jalr rs`	безусловный переход по регистру с возвратом (jump and link register)	`$ra = PC + 4, PC = [rs]`

Funct	Имя	Описание	Операция
001100 (12)	`syscall`	системный вызов (system call)	`исключение типа «системный вызов»`
001101 (13)	`break`	точка останова (break)	`исключение типа «останов в контрольной точке»`
010000 (16)	`mfhi rd`	пересылка из регистра hi (move from hi)	`[rd] = [hi]`
010001 (17)	`mthi rs`	пересылка в регистр hi (move to hi)	`[hi] = [rs]`
010010 (18)	`mflo rd`	пересылка из регистра lo (move from lo)	`[rd] = [lo]`
010011 (19)	`mtlo rs`	пересылка в регистр lo (move to lo)	`[lo] = [rs]`
011000 (24)	`mult rs, rt`	умножение 32-битных операндов со знаком и 64-битным результатом (multiply)	`{[hi], [lo]} = [rs] ×[rt]`
011001 (25)	`multu rs, rt`	умножение 32-битных операндов без знака и 64-битным результатом (multiply unsigned)	`{[hi], [lo]} = [rs] ×[rt]`
011010 (26)	`div rs, rt`	деление со знаком (divide)	`[lo] = [rs]/[rt], [hi] = [rs]%[rt]`

Funct	Имя	Описание	Операция
011011 (27)	`divu rs, rt`	деление без знака (Divide unsigned)	`[lo] = [rs]/[rt], [hi] = [rs]%[rt]`
100000 (32)	`add rd, rs, rt`	сложение со знаком, арифметическое переполнение вызвает исключение (add)	`[rd] = [rs] + [rt]`
100001 (33)	`addu rd, rs, rt`	сложение без знака, арифметическое переполнение не вызвает исключение (add unsigned)	`[rd] = [rs] + [rt]`
100010 (34)	`sub rd, rs, rt`	Вычитание со знаком, арифметическое переполнение вызвает исключение (subtract)	`[rd] = [rs] - [rt]`
100011 (35)	`subu rd, rs, rt`	вычитание без знака, арифметическое переполнение не вызвает исключение (subtract unsigned)	`[rd] = [rs] - [rt]`
100100 (36)	`and rd, rs, rt`	побитовое логическое «И» (and)	`[rd] = [rs] & [rt]`
100101 (37)	`or rd, rs, rt`	побитовое логическое «ИЛИ» (or)	`[rd] = [rs] | [rt]`

Funct	Имя	Описание	Операция
100110 (38)	`xor rd, rs, rt`	побитовое логическое «исключающее ИЛИ» (xor)	`[rd] = [rs] ^ [rt]`
100111 (39)	`nor rd, rs, rt`	побитовое отрицание логического «ИЛИ» (nor)	`[rd] = ~([rs] | [rt])`
101010 (42)	`slt rd, rs, rt`	установить, если меньше, сравнение выполняется со знаком (set less than)	`[rs] <[rt] ? [rd] = 1 : [rd] = 0`
101011 (43)	`sltu rd, rs, rt`	установить, если меньше, сравнение выполняется без знака (set less than unsigned)	`[rs] <[rt] ? [rd] = 1 : [rd] = 0`