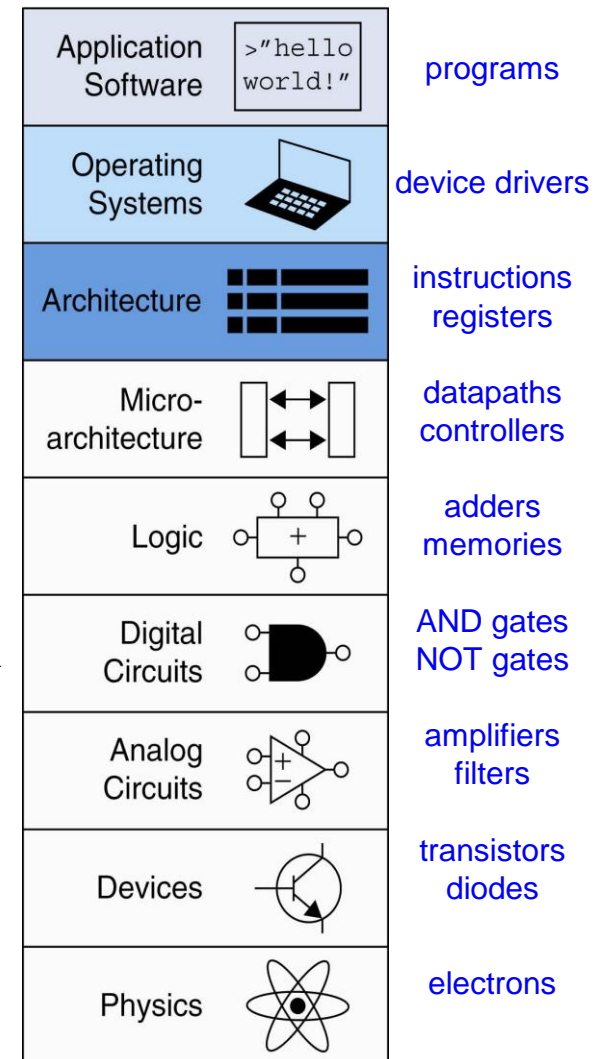


# КАРХ: Тема\_8: MIPS архитектура

## Въведение.

- Прескачаме няколко нива на абстракция.
- **Архитектура на компютъра** – това е погледа (визията) на програмиста за компютъра.
- Дефинира се чрез набора от инструкции (езика) и мястото на операндите (регистри и памет).
- Съществуват множество различни архитектури – x86, MIPS, SPARC, Power PC.
- Първа стъпка към разбирането на архитектурата на компютъра е научаването на **неговия език**.
- Думите на комп. език – **инструкции**, речникът му – набор от инструкции (**instruction set**).
- Инструкцията представлява операция, която се извършва върху дадени обекти – **операнди**.
- Операнди могат да бъдат регистри на процесора, адреси на клетки от паметта, константи (числа).



# КАРХ: Тема\_8: MIPS архитектура

## Въведение.

- Инструкциите се кодират с двоични числа – **машинен език**.
- Представянето на инструкциите в символен формат – **асемблерен език** (асемблер) на процесора.
- Наборът от инструкции на различните архитектури е по-скоро различен диалект, отколкото различен език.
- Компютърната архитектура не определя какъв точно да е хардуерът за дадена реализация, т.е. възможни са много хардуерни реализации, описващи дадена архитектура.
- **Микроархитектура:** специфичният набор от регистри, памети, АЛУ и други изграждащи блокове при направата на даден микропроцесор.
- В дадена архитектура могат да съществуват множество различни микроархитектури.
- Запознаване с **MIPS** архитектурата:
  - Разработена от John Hennessy и негови колеги в Stanford през 80-те години на миналия век.
  - **MIPS** процесори използват много фирми в продуктите си, като Silicon Graphics, Nintendo и Cisco. Продажбите на този тип процесори надхвърля стотици милиони.
  - Представяне на основните инструкции, локациите на операндите и форматът им на машинен език.

# КАРХ: Тема\_8: MIPS архитектура

Създател на MIPS архитектурата.

## **John Hennessy**

- President of Stanford University
- Professor of Electrical Engineering and Computer Science at Stanford since 1977
- Coinvented the Reduced Instruction Set Computer (RISC) with David Patterson
- Developed the MIPS architecture at Stanford in 1984 and cofounded MIPS Computer Systems
- As of 2004, over 300 million MIPS microprocessors have been sold

*John Hennessy*



# КАРХ: Тема\_8: MIPS архитектура

Основополагащи принципи при дизайна (Underlying design principles).

Въведени от авторите Hennessy и Patterson:

1. **Simplicity favors regularity** (Простотата подкрепя, улеснява обичайното).
2. **Make the common case fast** (Прави общото бързо).
3. **Smaller is faster** (По-малкото е по-бързо).
4. **Good design demands good compromises** (Добрият дизайн изисква добър компромис).

# КАРХ: Тема\_8: MIPS архитектура

## Общи принципи при MIPS архитектурата.

- Всяка инструкция специфицира както операцията, която се извършва, така и операндите, върху които се извършва.
- Наборът от инструкции в MIPS архитектурата съдържа само прости, често използващи се инструкции, като броят им се поддържа малък.
- Така хардуерът, който ги изпълнява, да е прост, малък и бърз.
- Всички по-сложни операции се представят чрез поредица от прости инструкции. За това MIPS архитектурата спада към групата на т. нар. RISC (Reduced Instruction Set Computers) архитектури. (Има и CISC архитектури).
- Малкият набор инструкции позволява лесното им кодиране и декодиране, например за 64 инструкции са необходими  $\log_2 64 = 6$  bit за кодиране.
- MIPS е 32 bit архитектура, защото оперира с 32 bit данни ( има и 64 bit версия).
- Операндите са регистри, памет и константи.
- MIPS архитектурата използва 32 регистъра (набор регистри, регистър файл).
- Регистър файлът най-често е изграден от малка SRAM памет с декодер, адресиращ всяка клетка от паметта.

# КАРХ: Тема\_8: MIPS архитектура

## Регистри в MIPS архитектурата.

- MIPS регистрите се означават с \$-знак, напр. \$s1 означава регистър s1.
- При MIPS данни могат да се запазват в 18 от 32-та регистъра, а именно \$s0 \$s7 и \$t0 \$t9 ( s – saved register, t – temporary register), t-регистрите съхраняват временни променливи.
- Предполага се познаване на някои от езиците от високо ниво като C, C++ или Java.

- Примери – събиране и изваждане:

### C Code

```
a = b + c;
```

### MIPS assembly code

```
add a, b, c
```

- **add:** мнемоничен код на операцията;
- **b, c:** **операнди източници** (source operands);
- **a:** **операнд получател** (destination operand) (където се записва резултата).
- В езика C инструкциите завършват с (;), // е коментар в една линия, а /\* е коментар в много линии \*/
- При MIPS коментарът е само в една линия и се бележи с (#).

### C Code

```
a = b - c;
```

### MIPS assembly code

```
sub a, b, c
```

# КАРХ: Тема\_8: MIPS архитектура

## Регистри в MIPS архитектурата.

- По-сложните операции изискват няколко MIPS инструкции.

### C Code

```
a = b + c - d;
```

### MIPS assembly code

```
add t, b, c    # t = b + c
sub a, t, d     # a = t - d
```

- В действителност MIPS кодът трябва да изглежда така:

### C Code

```
a = b + c
```

```
a = b + c - d;
```

### MIPS assembly code

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

# КАРХ: Тема\_8: MIPS архитектура

## Набор регистри в MIPS архитектурата.

Name	Register Number	Usage
\$0	0	the constant value 0
\$at	1	assembler temporary
\$v0-\$v1	2-3	Function return values
\$a0-\$a3	4-7	Function arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved variables
\$t8-\$t9	24-25	more temporaries
\$k0-\$k1	26-27	OS temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	Function return address



# КАРХ: Тема\_8: MIPS архитектура

## Памет в MIPS архитектурата.

- Данните в компютъра са твърде много за да се поберат само в 32 регистъра.
- Повечето данни се пазят в паметта.
- Паметта е голяма, но бавна. (Memory is large, but slow.)
- За това най-често използваните данни се съхраняват в регистрите.
- MIPS архитектурата използва 32 bit адреси за 32 bit думи, като адресирането е на байтове, т.е. всеки байт има уникален адрес.
- За начало разглеждаме адресирането на цели думи (4 байта).

	Word Address	Data	
	⋮	⋮	⋮
	00000003	4 0 F 3 0 7 8 8	Word 3
	00000002	0 1 E E 2 8 4 2	Word 2
Например, адрес 1	00000001	F 2 F 1 A C 0 7	Word 1
съдържа данните	00000000	A B C D E F 7 8	Word 0
0xF2F1AC07			

# КАРХ: Тема\_8: MIPS архитектура

## Четене на данни от паметта.

- Четенето на данни от паметта се нарича *зареждане* (*load*).
- **Мнемоничен код** : *load word* (lw)
- **Формат**:  
*lw \$s0, 5(\$t1)*
- **Изчисляване на адреса**:
  - Събира се базовия адрес (*base address*) (\$t1) с отместването (*offset* )(5)
  - Реален адрес (address) = (\$t1 + 5)
- **Резултат**:
  - Регистърът \$s0 съдържа стойността (данните) записани на адрес (\$t1 + 5)
- **Всеки регистър** може да се използва като базов адрес.

# КАРХ: Тема\_8: MIPS архитектура

## Четене на данни от паметта.

- **Пример:** зареждане на регистъра \$s3 със съдържанието на думата записана на адрес 1
  - $\text{address} = (\$0 + 1) = 1$
  - $\$s3 = 0xF2F1AC07$  след зареждането

### Assembly code

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

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

# КАРХ: Тема\_8: MIPS архитектура

## Запис на данни в паметта.

- Записът в паметта се нарича съхраняване (*store*).
- **Мнемоничен код** : *store word* (sw)
- **Пример**: Запис (store) на стойността на регистъра \$t4 в паметта на адрес 7.
  - Събира се базовия адрес (*base address*)( $\$0$ ) с отместването (*offset* )(0x7)
  - Реален адрес (address) :  $(\$0 + 0x7) = 7$
  - Отместването (Offset) може да е десетично число /по подразбиране (default)/ или шестнадесетично число (hexadecimal)

## Assembly code

```
sw $t4, 0x7($0)  # write the value in $t4
                  # to memory word 7
```

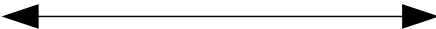
Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

# КАРХ: Тема\_8: MIPS архитектура

## Адресиране на данните в паметта.

- **Паметта в MIPS архитектурата е байтово адресируема (не на цели думи!)**, за това адресът на всяка дума е кратен на 4, т.е. адресът на думата е 4 пъти по-голям от номера на съответната дума.
- Всеки байт данни има уникален адрес.
- Могат да се четат/записват думи или единични байтове с инструкциите *load byte (lb)* и *store byte (sb)*.
- 32-bit дума = 4 bytes (байта), следователно адресът на думите нараства с 4.

Word Address	Data								
⋮	⋮								⋮
0000000C	4	0	F	3	0	7	8	8	Word 3
00000008	0	1	E	E	2	8	4	2	Word 2
00000004	F	2	F	1	A	C	0	7	Word 1
00000000	A	B	C	D	E	F	7	8	Word 0

  
width = 4 bytes

# КАРХ: Тема\_8: MIPS архитектура

## Адресиране на данните в паметта.

- Примери:

**Пример 1:** зареждане на регистъра \$s3 със съдържанието на думата записана на адрес 4.

– \$s3 съдържа числото 0xF2F1AC07 след зареждането.

### MIPS assembly code

```
lw $s3, 4($0) # read word at address 4 into $s3
```

Word Address	Data					
⋮	⋮					
0000000C	4	0	F	3	0	7 8 8 Word 3
00000008	0	1	E	E	2	8 4 2 Word 2
00000004	F	2	F	1	A	C 0 7 Word 1
00000000	A	B	C	D	E	F 7 8 Word 0

← width = 4 bytes →

**Пример 2:** Запис на стойността на регистъра \$t7 в паметта на адрес 0x2C (44).

### MIPS assembly code

```
sw $t7, 44($0) # write $t7 into address 44
```

# КАРХ: Тема\_8: MIPS архитектура

## Организация на адресиране на данните в паметта.

- Как се номерират байтовете в една дума?
- **Little-endian:** номерирането на байтовете започва от малкия (least significant) край.
- **Big-endian:** номерирането на байтовете започва от големия (most significant) край.
- **Адресът на думата е един и същ за big- или little-endian.**

### Big-Endian

Byte Address			
⋮			
C	D	E	F
8	9	A	B
4	5	6	7
0	1	2	3
MSB		LSB	

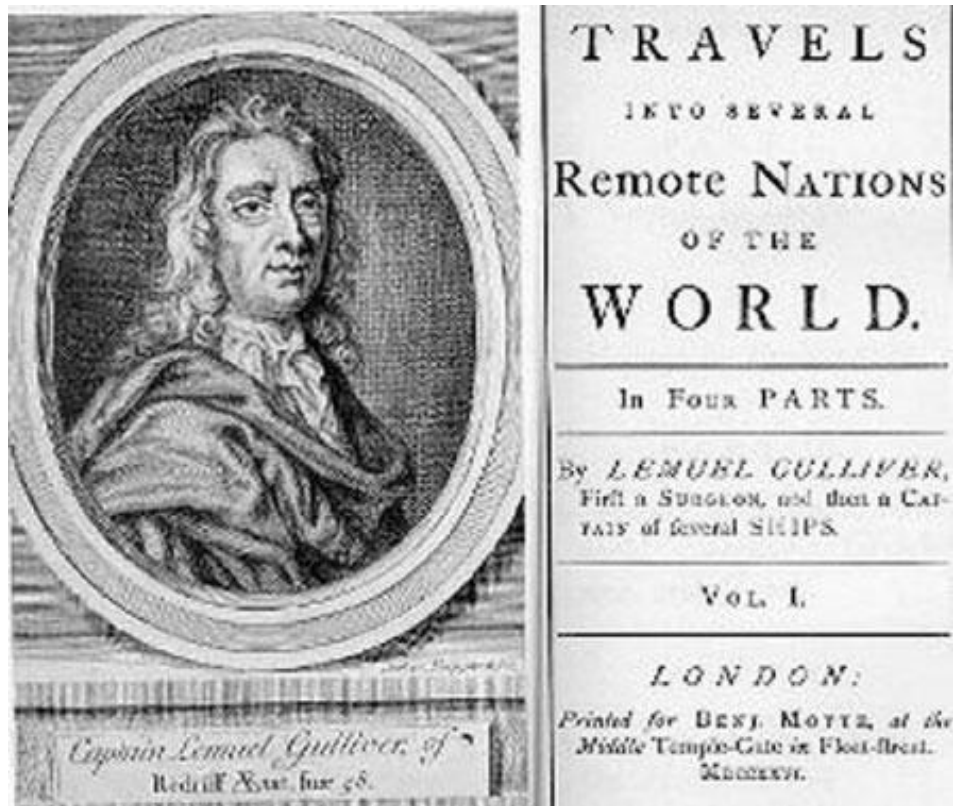
### Little-Endian

Byte Address			
⋮			
F	E	D	C
B	A	9	8
7	6	5	4
3	2	1	0
MSB		LSB	

# КАРХ: Тема\_8: MIPS архитектура

## Организация на адресиране на данните в паметта.

- В романа на Джонатан Суифт (Jonathan Swift) *Пътешествията на Гъливер* (*Gulliver's Travels*) (1726): подданиците на царя на Лилипутите (the Little-Endians) чупели техните яйца откъм острия край, а (the Big-Endians) били бунтовници и чупели яйцата откъм тъпия край.
- Няма значение кой от двата формата се използва, освен в случаите, когато две системи трябва да обменят данни!





# КАРХ: Тема\_8: MIPS архитектура

## Организация на адресиране на данните в паметта.

- **Задача.** Да предположим, че регистъра `$t0` съдържа числото `0x23456789`.
- След като дадения код се изпълни на `big-endian` система, каква стойност ще е записана в регистъра `$s0`?
- А при `little-endian` система?

```
sw $t0, 0($0)
```

```
lb $s0, 1($0)
```

# КАРХ: Тема\_8: MIPS архитектура

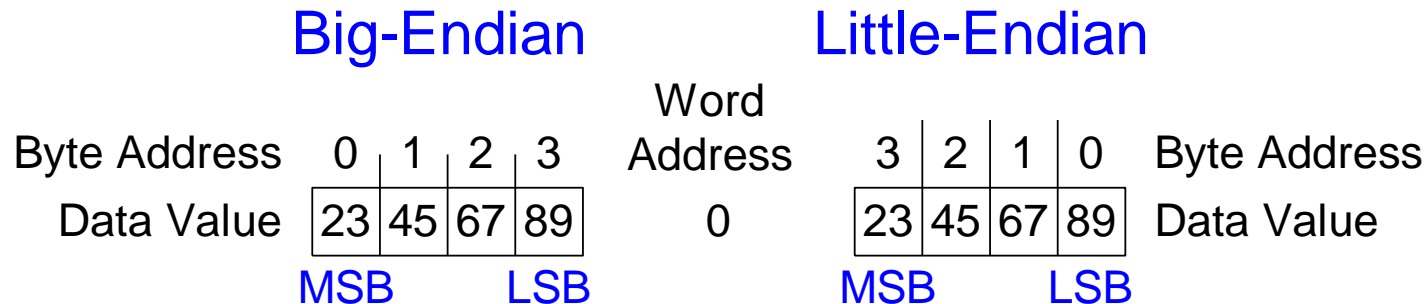
## Организация на адресиране на данните в паметта.

- **Задача.** Да предположим, че регистъра `$t0` съдържа числото `0x23456789`.
- След като дадения код се изпълни на `big-endian` система, каква стойност ще е записана в регистъра `$s0`?
- А при `little-endian` система?

```
sw $t0, 0($0)
```

```
lb $s0, 1($0)
```

- **Big-endian:** `0x00000045`
- **Little-endian:** `0x00000067`



# КАРХ: Тема\_8: MIPS архитектура

## Организация на адресиране на данните в паметта.

- Има два формата на организация на данните в една дума – **Big-endian** и **Little-endian**. И при двата формата старшият байт (MSB) е отляво, а младшият байт (LSB) – отдясно.
- При **Big-endian машините** номерацията на байтовете започва от MSB, а при **Little-endian машините** – от LSB.
- Адресирането на цели думи е еднакво и при двата формата, но номерът на отделните байтове в думата е различен.
- При *Intel\_x86* – архитектурата се използва Little-endian формата.
- При MIPS архитектурата адресите на думите използвани с инструкциите `lw` и `sw` трябва да бъдат **подравнени (word aligned)**, т.е. Адресът трябва да е делим на 4 (`lw $s3, 7($0)` е **недопустима** инструкция!). Някои архитектури (напр. *Intel\_x86*) допускат нарушаване на това правило.

# КАРХ: Тема\_8: MIPS архитектура

## Константи/непосредствени операнди (*immediates*).

- Стойностите им се задават непосредствено, без да е необходим достъп до регистър или памет.
- Използване – напр. В инструкцията **addi** ;

### C Code

```
a = a + 4;  
b = a - 12;
```

### MIPS assembly code

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

- Няма нужда от **subi** инструкция в MIPS архитектурата!
- Константата е 16-bit число в двоично-допълнителен код [-32768 32767].