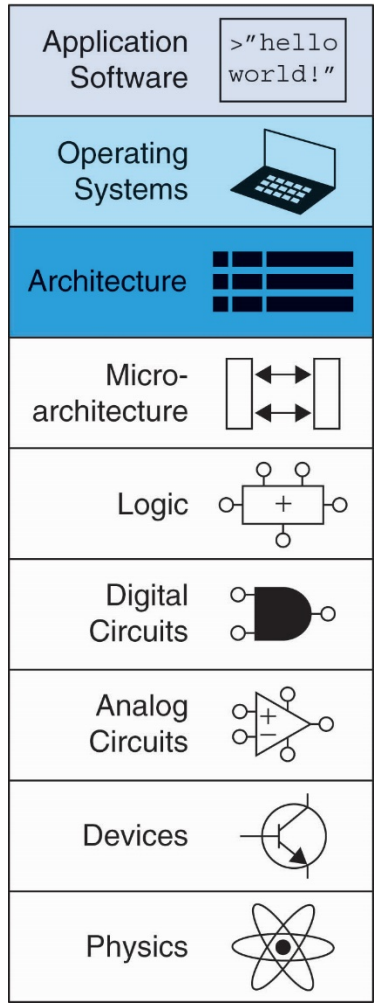
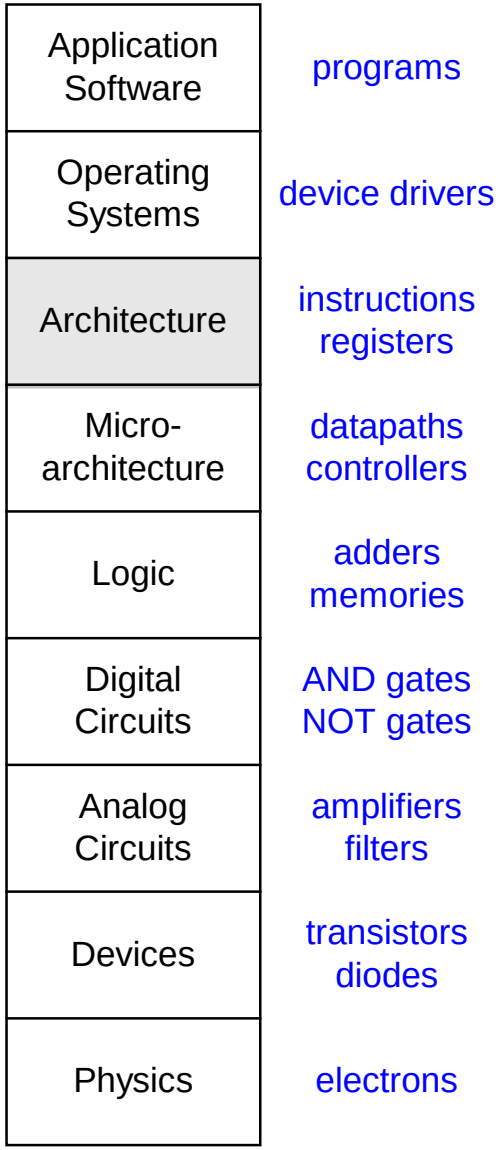
Розділ 6. Теми

- **Вступ**
- **Мова асемблера**
- **Машина мова**
- **Програмування**
- **Режими адресації**
- **Компіляція, асемблювання і завантаження**
- **Додаткова інформація**



Вступ

- Наступним рівнем абстракції над цифровими схемами є *архітектура і мікроархітектура*.
- **Архітектура:** те, як бачить комп'ютер програміст.
  - Архітектура визначається набором команд процесора (інструкціями) і місцем знаходження їх операндів (регістри і пам'ять), а також способами адресації операндів.
  - **Мікроархітектура:** спосіб апаратної реалізації конкретної архітектури як схеми (розглядується у розділі 7).



Мова асемблера

- **Інструкції:** команди, які виконує процесор
  - **Мова асемблера:** зручний для сприйняття людиною формат інструкцій процесора (інструкції і їх операнди задані у текстовому вигляді)
  - **Машинна мова:** формат інструкцій зрозумілий для процесора (двійкові числа, які складаються з 0 і 1)
- **MIPS** архітектура:
  - Розроблена Джоном Хеннесі і його колегами у Стенфорді, в 1980-х.
  - Використовується у багатьох комерційних проектах, включно з продуктами Silicon Graphics, Nintendo і Cisco

Після вивчення однієї архітектури, стає набагато простіше вивчити любу іншу, оскільки різні архітектури використовують деякі загальні підходи.

Принципи побудови архітектури

Основні принципи побудови архітектури:

1. **Одноманітність – запорука простоти**
2. **Типовий сценарій має бути швидким**
3. **Чем менше, тем швидше**
4. **Добра розробка потребує добрих компромісів**

Приклад інструкції. Додавання

Код на C

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

Код на асемблері MIPS

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

- **add:** мнемоніка інструкції, вказує, яку операцію необхідно виконати
- **a:** операнд-приймач (в який записується результат операції)
- **b, c:** операнди-джерела (над якими виконується операція, задана мнемонікою)

Приклад інструкції. Віднімання

- Формат дуже подібний до інструкції додавання, відрізняється тільки кодом мнемоніки sub

Код на C

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

Код на асемблері MIPS

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

- **sub:** мнемоніка
- **a:** операнд-призначення, операнд-приймач
- **b, c:** операнди-джерела

Одноманітність – запорука простоти

- Одноманітний формат інструкцій є прикладом першого принципу доброї розробки
- Однакова кількість операндів (два-операнди джерела, один операнд-призначення)
- Такий підхід дозволяє значно спростити апаратну реалізацію – інструкції однакового формату простіше декодувати і виконувати

Багато простих інструкцій

- Більш складний високорівневий код перетворюється в декілька простих інструкцій MIPS з однаковим форматом

Код на C

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

Код на асемблері MIPS

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

Принцип побудови архітектури. 2

Типовий сценарій має бути простим

- Архітектура MIPS включає тільки прості, часто використовувані інструкції
- Апаратна реалізація схем декодування і виконання таких інструкцій проста, швидка і займає мало місця на кристалі
- Більш складні інструкції (і менш поширені) перетворюються компілятором в декілька простих інструкцій
- MIPS є RISC процесором із скороченим набором команд – невеликою кількістю простих інструкцій однакового формату (**RISC – *Reduced Instruction Set Computer***)
- Інші архітектури, такі як Intel x86, є прикладом CISC обчислювачів з повним набором команд (***CISC – Complex Instruction Set Computers***)

Операнди

- Фізичне розміщення операндів:
  - регістри
  - пам’ять
  - константи (операнди безпосереднього типу – *immediate operands*)

Регістри

- MIPS має 32 32-бітних регістри загального призначення
- Регістри швидші ніж пам’ять (менший час доступу)
- Операнди арифметичних і логічних інструкцій MIPS або регістри, або константи (для підвищення швидкодії і спрощення апаратної реалізації)
- MIPS називають “32-бітною архітектурою” оскільки він оперує 32-бітними даними і має 32-бітні інструкції

Принцип побудови архітектури. 3

Чем менше, тим швидше. MIPS має незначну кількість регістрів загального призначення

Ім’я	Номер	Використання
$0	0	Регістр нуля (завжди повертає 0)
$at	1	Тимчасовий регістр для потреб асемблера
$v0-$v1	2-3	Значення, які повертає функція
$a0-$a3	4-7	Аргументи функцій
$t0-$t7	8-15	Тимчасові змінні
$s0-$s7	16-23	Змінні (локальні), які зберігаються
$t8-$t9	24-25	Тимчасові змінні
$k0-$k1	26-27	Тимчасові змінні ОС
$gp	28	Глобальний вказівник
$sp	29	Вказівник стеку
$fp	30	Вказівник кадра стеку
$ra	31	Адреса повернення з функції

Операнди: Регістри

- Регістри:
  - Знак $ перед іменем в програмах асемблера
  - Наприклад: $0, “регістр нуля”
- Регістри використовуються з певною метою:
  - $0 завжди зберігає константу 0
  - *регістри для зберігання*, $s0-$s7, використовуються для зберігання змінних
  - *тимчасові регістри*, $t0 - $t9, використовуються для зберігання проміжних значень при обчисленнях складних виразів
  - решта регістрів будуть розглянуті пізніше

Інструкції з регістрами

Код на C

```
a = b + c
```

Код на асемблері MIPS

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

Операнди: Пам’ять. Адресація

- Все дані не помістяться в 32 регістри загального призначення
- Більшість даних зберігається у пам’яті
- Обсяг пам’яті більший, ніж у регістрів (можна зберігати більше даних), але швидкодія пам’яті нижча (доступ до пам’яті займає більше часу, ніж доступ до регістрів)
- В регістрах зберігаються найбільш часто використовувані змінні
- Кажне 32-розрядне слово в пам’яті має унікальну адресу. Сусідні адреси відрізняються на 1. В пам’яті адресуються слова.

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

Читання із пам’яті, яка адресує слова

- Для читання 32-розрядного слова із пам’яті в регістр загального призначення використовується інструкція ***load***
- **Мнемоніка:** *load word* (***lw***)
- **Формат:**

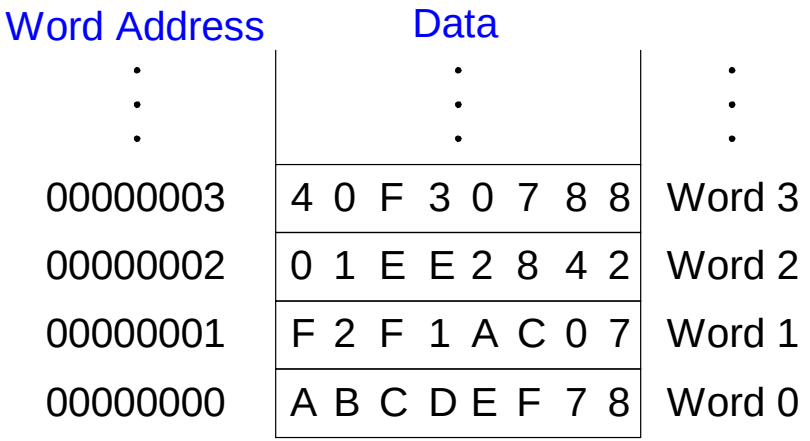
  lw $s0, 5($t1)
- **Розрахунок адреси слова у пам’яті:**
  - Необходимо додати *базову адресу, яка зберігається у регістрі* ($t1) *з константним зміщенням* (5)
  - Адреса = ($t1 + 5)
  - Для зберігання базової адреси може бути використаний **любий регістр**
- **Результат:**
  - В регістр $s0 зчитується значення, яке зберігається в пам’яті за адресою ($t1 + 5)

Читання із пам’яті, яка адресує слова

- **Приклад:** необхідно прочитати слово з пам’яті (яка адресує слова) за адресою 1 в регістр $s3
  - Адреса = ($0 + 1) = 1
  - Після виконання інструкції *load*, в регістрі $s3 буде значення 0xF2F1AC07

Код на асемблері

```
#зчитати слово за адресою 1 в $s3
lw $s3, 1($0)
```



Зауваження: інструкція lw працює в MIPS трохи інакше, оскільки пам’ять в MIPS адресується побайтно. Детальніше про відмінності – далі в презентації.

Записування у пам’ять, яка адресує слова

- Записування в комірку пам’яті вмісту регістра загального призначення виконується за допомогою інструкції ***store***
- **Мнемоніка:** *store word* (***sw***)
- **Формат** має багато спільного з інструкцією lw
- **Приклад:** Необхідно записати (зберегти) значення регістру $t4 у пам’ять (яка адресує слова) за адресою 7
  - адреса комірки пам’яті отримується додаванням *базової адреси* ($0) і *зміщення* (0x7)
  - отримується адреса: ($0 + 0x7) = 7

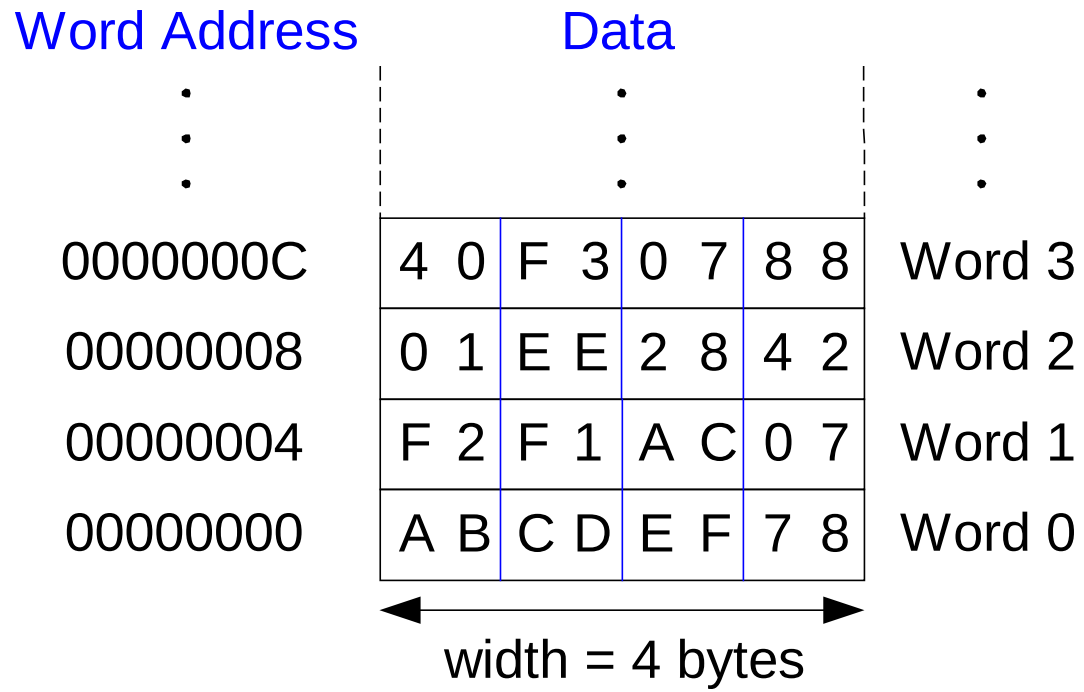
  Зміщення можна записувати як у десятковій системі, так і у шістнадцятковій

Код на асемблері

```
sw $t4, 0x7($0)  # збереження значення регістру $t4
                 # у комірку пам’яті за адресою 7
```

Пам’ять, яка адресує слова

- Кожний байт в пам’яті має унікальну адресу
- Адреси сусідніх байтів відрізняються на 1
- Можна зчитувати/записувати (load/store), як слова, так і окремі байти: load byte (lb) і store byte (sb)
- 32-бітне слово = 4 байти, тому адреса кажного наступного слова збільшується на 4



Читання/записування пам’яті, яка адресується побайтно

- Для переходу від адреси в словах до адреси в байтах, необхідно адресу в словах помножити на 4 (кожне слово містить 4 байти). Наприклад:
  - адреса в байтах 2-го слова в пам’ятs: 2 × 4 = 8
  - адреса в байтах 10-го слова в пам’яті: 10 × 4 = 40 (0x28)
- **В архітектурі MIPS пам’ять адресується за байтами, а не за словами**
- **Приклад:** необхідно зчитати слово з пам’яті за адресою 4 (в байтах) в регістр $s3.
- Після операції load регістр $s3 буде містити значення 0xF2F1AC07.

Код на асемблері MIPS

```
lw $s3, 4($0)  #зчитування слова за адресою 4 в $s3
```

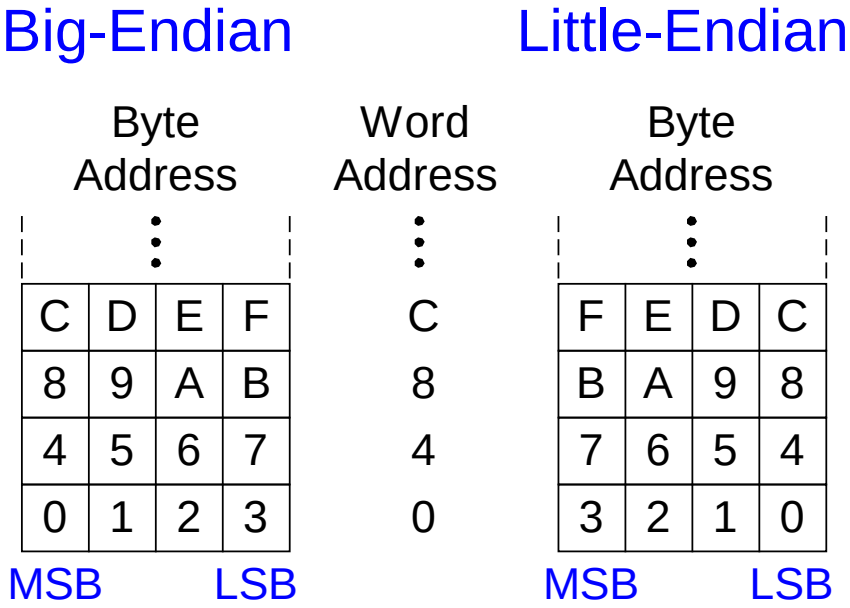
Приклад: записати значення регістра $t7 в пам’ять за десятковою адресою 44 (в байтах).

Код на ассемблері MIPS

```
sw $t7, 44($0)  #збереження вмісту $t7 за адресою 44
```

Порядок байтів в слові пам’яті

- В якому порядку байти слова розміщуються у пам’яті з байтовою адресацією?
- **Little-endian порядок:** байти в слові розміщені від молодшого до старшого (за меншою адресою знаходиться молодший байт слова)
- **Big-endian порядок:** байти в слові розміщені від старшого до молодшого (за меншою адресою знаходиться старший байт слова)
- **Адреса слова** одна і та ж для big- і little- порядків (відрізняється тільки порядок байтів всередині слова)

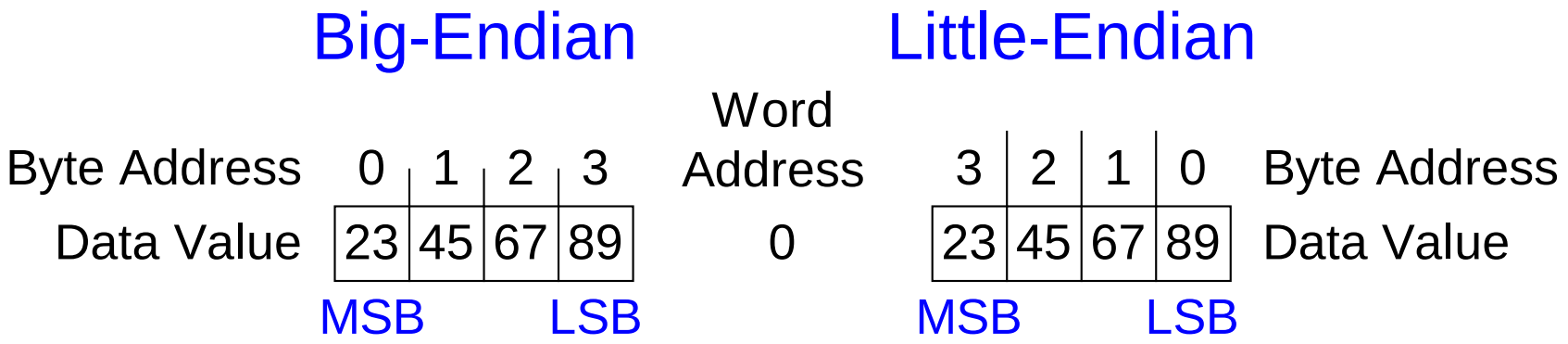


Приклад Big-Endian і Little-Endian

- Припустимо, що $t0 спочатку містить 0x23456789
- Яке значення буде в $s0 після виконання наступного коду у big-endian і little-endian системах?

```
sw $t0, 0($0)
lb $s0, 1($0)
```

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



Принцип побудови архітектури 4.

Добра розробка вимагає добрих компромісів

- Декілька форматів інструкцій дають гнучкість і більше можливостей
  - add, sub: використовують 3 операнди-регістри
  - lw, sw: використовують 2 операнди-регістри і константу
- Кількість форматів інструкцій має бути мінімальною
  - щоб забезпечити попередні 3 принципи побудови архітектури.

Операнди: Константи (immediates)

- **lw** і **sw** використовують константу-операнд (безпосередні дані – *immediate)*
- константа-операнд зберігається безпосередньо (*immediate*ly) в інструкції
- ця константа – 16-бітне знаковое число
- инструкция **addi**: дадає знакову константу до вмісту регістра
- Чи є необхідність в інструкції віднімання такої знакової константи (**subi**)?

Код на C

```
a = a + 4;
b = a – 12;
```

Код на асемблері MIPS

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

Машинна мова

- Двійкове подання інструкцій
- Процесор розуміє тільки нулі і одиниці
- 32-бітні інструкції
  - Одноманітність запорука простоти: 32-бітні дані і інструкції
- 3 формати інструкцій:
  - **R-Тип:** 3 операнди-регістри
  - **I-Тип:** 2 операнди-регістри і один операнд-константа (Immediate)
  - **J-Тип:** для безумовних переходів типу jump (будуть розглянуті пізніше)

R-тип

- *Регістровий-тип*
- 3 регістри-операнди:
  - rs, rt: регістри-джерела
  - rd: регістр-призначення (регістр-приймач)
  - поля rs, rt, rd містять номери регістрів (від 0 до 31)
- Інші поля:
  - op: *код операції*, *opcode* (0 для всіх інструкцій R-типу)
  - funct: *функція* (разом з *opcode*, вказує процесору яку операцію необхідно виконати)
  - shamt: *shift amount*. Кількість бітів зсуву для інструкцій зсуву (для решти інструкцій це поле 0)

R-Type

Приклади інструкцій R-типу

Assembly Code

```
add $s0, $s1, $s2

sub $t0, $t3, $t5
```

Field Values

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

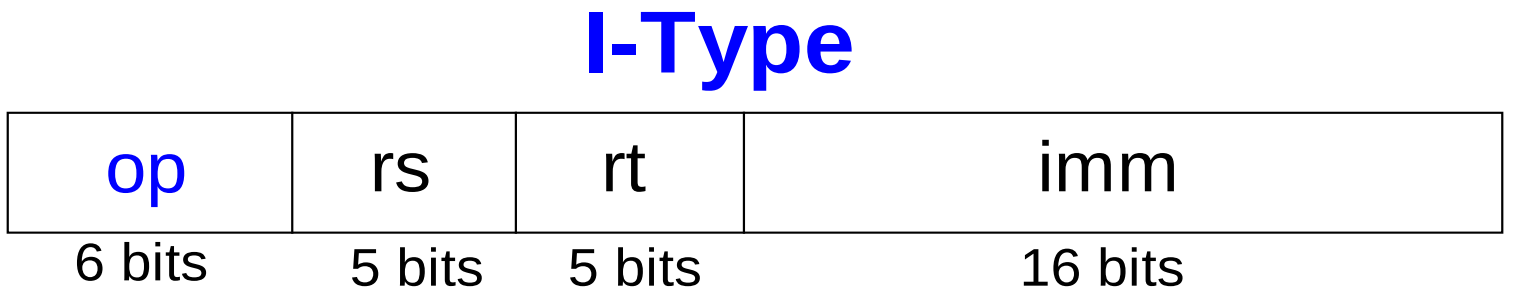
op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Порядок слідування регістрів в коді асемблера:

```
add rd, rs, rt
```

Інструкції I-типу

- *Immediate-тип*
- 3 операнди:
  - rs: номер регістру-джерела (від 0 до 31)
  - rt: номер регістру-призначення (від 0 до 31)
  - imm: другий операнд-джерело (16-бітна знакова константа)
- Інші поля:
  - op: код операції, *opcode*
  - Одноманітність запорука простоти: всі інструкції містять *opcode*
  - В інструкціях I-типу виконувана операція повністю визначена полем *opcode*



Приклади інструкцій I-типу

Assembly Code	op	rs	rt	imm
`addi $s0, $s1, 5`	8	17	16	5
`addi $t0, $s3, -12`	8	19	8	-12
`lw $t2, 32($0)`	35	0	10	32
`sw $s1, 4($t1)`	43	9	17	4
	6 bits	5 bits	5 bits	16 bits

Field Values

Різний порядок слідування регістрів в асемблері і машинній мові:

addi rt, rs, imm

lw rt, imm(rs)

sw rt, imm(rs)

Machine Code

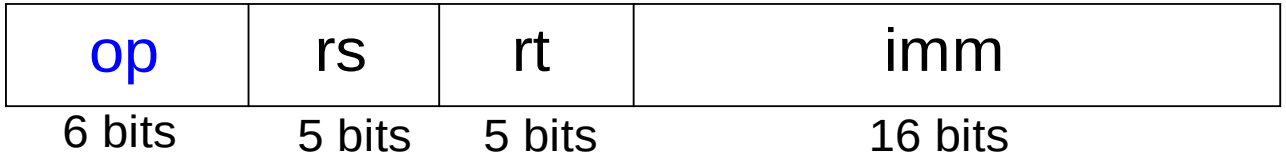
op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	(0x22300005)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
6 bits	5 bits	5 bits	16 bits	

Машинна мова: інструкції J-типу

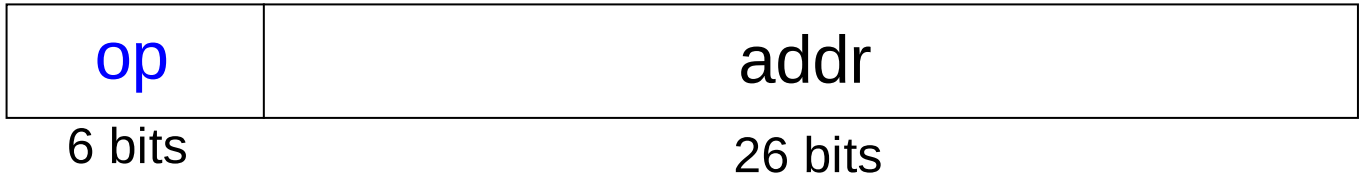
- *Jump-тип*
- 26-бітна адреса (addr), операнд інструкція
- Використовується для інструкцій безумовного переходу типу jump (j)
- Підсумок – формати інструкцій

R-Type

I-Type



J-Type



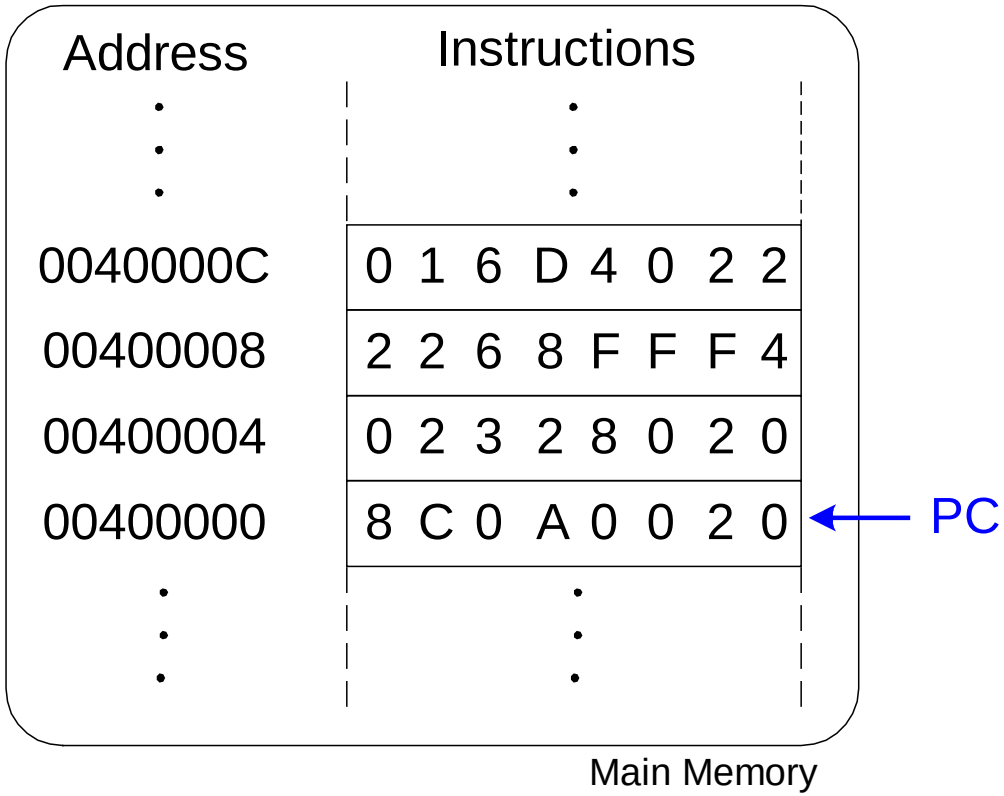
Можливості програми, яка зберігається

- Програма, написана на машинній мові – послідовність чисел, які задають інструкції
- Як і любі інші двійкові числа, ці інструкції можна зберігати в пам’яті. Такий підхід називається концепцією збережуваної програми
- Запуск нової програми не вимагає великих затрат часу і зусиль на зміну або реконфігурацію апаратного забезпечення; все, що для цього потрібно – записати нову програму у пам’ять.
- Дві програми, які виконуюь зовсім різні алгоритми, відрізняються тільки послідовністю інструкцій
- Виконання програми:
  - процесор послідовно видобуває (зчитує) інструкції з пам’яті програм
  - для кожної зчитаної інструкції процесор виконує відповідну операцію

Збережувана програма

Assembly Code	Machine Code
`lw $t2, 32($0)`	`0x8C0A0020`
`add $s0, $s1, $s2`	`0x02328020`
`addi $t0, $s3, -12`	`0x2268FFF4`
`sub $t0, $t3, $t5`	`0x016D4022`

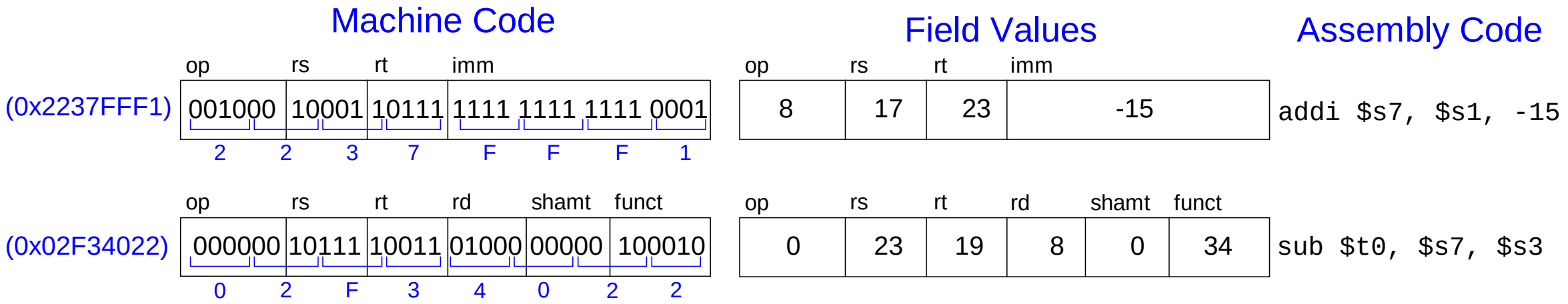
Stored Program



Лічильник команд (Program counter – PC): містить адресу виконуваної процесором інструкції

Виконання машинного коду

- Спочатку аналізується код операції opcode. В залежності від значення цього поля:
- Якщо opcode=0
  - значить іструкція **R-типу**
  - операція визначається полем *funct*
- Інакше
  - операція визначається полем *opcode*



Програмування

- Високорівневі мови:
  - C, Java, Python, C#
  - Алгоритм роботи програми описується на високому рівні абстракції
- Часто використовувані високорівневі програмні конструкції:
  - умови if/else
  - цикли for
  - цикли while
  - масиви
  - виклики функцій

Логічні інструкції

Інструкція	Приклад	Зміст	Пояснення
І, AND	and $1,$2,$3	$1=$2&$3	Побітове AND
АБО, OR	or $1,$2,$3	$1=$2|$3	Побітове OR
Виключальне АБО, XOR	xor $1,$2,$3	$1=$2&^$3	Побітове XOR
І, AND безпосереднє	andi $1,$2,100	$1=$2&100	Побітове AND і число 100
АБО, OR безпосереднє	ori $1,$2,100	$1=$2|100	Побітове OR і число 100
Викл. АБО, XOR безпосереднє	xori $1,$2,100	$1=$2^100	Побітове XOR і число 100
АБО НЕ, NOR	nor $1,$2,$3	$1=~($1|$2)	Побітове NOR

and: інструкція “І” **маскує** (обнуляє) біти числа:

0xF234012F AND 0x000000FF = 0x0000002F

or: інструкція “АБО” **об’єднує** біти двох чисел

0xF2340000 OR 0x000012BC = 0xF23412BC

nor: інструкція “АБО-НЕ” **інвертує** біти:

A NOR $0 = NOT A

andi, ori, xori – 16-бітна константа розширюється нулями, а не старшим бітом

Логічні інструкції. Приклад 1.

Source Registers

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

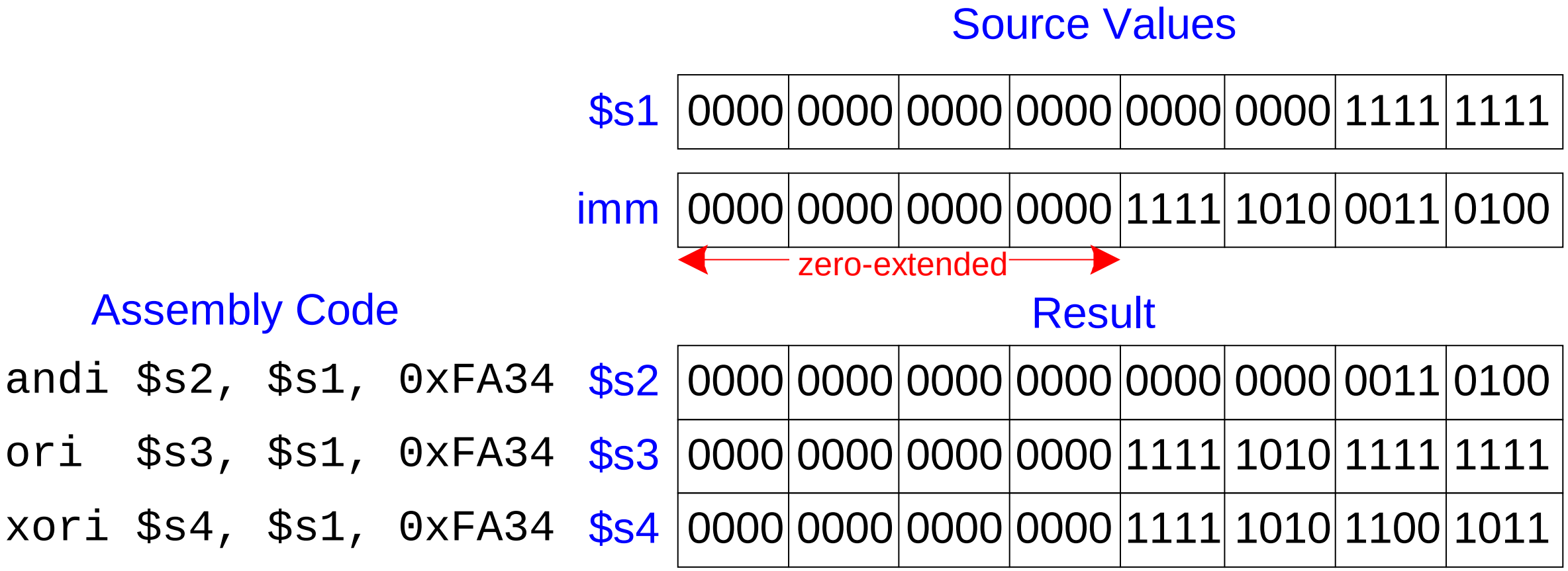
Assembly Code

```
and $s3, $s1, $s2
or  $s4, $s1, $s2
xor $s5, $s1, $s2
nor $s6, $s1, $s2
```

Result

$s3	0100	0110	1010	0001	0000	0000	0000	0000
$s4	1111	1111	1111	1111	1111	0000	1011	0111
$s5	1011	1001	0101	1110	1111	0000	1011	0111
$s6	0000	0000	0000	0000	0000	1111	0100	1000

Логічні інструкції. Приклад 2.



Інструкції зсуву

Інструкція	Приклад	Зміст	Пояснення
Зсув вліво логічний	sll $1,$2,K	$1=$2<<K	Зсув вліво на K бітів
Зсув вправо логічний	slr $1,$2,K	$1=$2>>K	Зсув вправо на K бітів
Зсув вправо арифметичний	sra $1,$2,K	$1=(int 32) $2>>>K	Зсув вправо ариф. на K бітів
Зсув вліво логічний на змінну кількість розрядів	sllv $1,$2,$3	$1=$2<<$3	Зсув вліво на $3 бітів
Зсув вправо логічний на змінну кількість розрядів	srlv $1,$2,$3	$1=$2>>$3	Зсув вправо на $3 бітів
Зсув вправо арифметичний на змінну кількість розрядів	srav $1,$2,$3	$1=$2>>>10	Зсув вправо арифметичний на $3 бітів

Інструкції зсуву. Приклад

Assembly Code

Field Values

Assembly Code	op	rs	rt	rd	shamt	funct
`sll $t0, $s1, 2`	0	0	17	8	2	0
`srl $s2, $s1, 2`	0	0	17	18	2	2
`sra $s3, $s1, 2`	0	0	17	19	2	3
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	00000	10001	01000	00010	000000	(0x00114080)
000000	00000	10001	10010	00010	000010	(0x00119082)
000000	00000	10001	10011	00010	000011	(0x00119883)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Завантаження і записування констант

Інструкція	Приклад	Зміст	Пояснення
Завантаження байту	lb $t,C($s)	$t=*(int8*)($s+C)	
Завантаження півслова	lh $t,C($s)	$t=*(int16*)($s+C)	
Завантаження слова	lw $t,C($c)	$t=*(int32*)($s+C)	
Завантаження беззнакового байту	lbu $t,C($s)	$t=*(uint8*)($s+C)	
Завантаження половини беззнакового слова	Lhu $t,C($c)	$t=*(uint16*)($s+C)	
Записування байту	sb $t,C($s)	*(uint8*)($s+C)=($t & 0xff)	
Записування півслова	sh $t,C($s)	*(uint16*)($s+C)=($t & 0xffff)	
Записування слова	sw $t,C($s)	*(uint32*)($s+C)=$t	

Завантаження констант

- Завантаження 16-бітної константи в регістр з використанням addi:

Код на C

```
//int: 32-бітне знакове число
int a = 0x4f3c;
```

Код на асемблері MIPS

```
# $s0 = a
addi $s0, $0, 0x4f3c
```

- Завантаження 32-бітної константи в регістр з використанням інструкцій lui (load upper immediate) і ori:

Код на C

```
int a = 0xFEDC8765;
```

Код на асемблері MIPS

```
# $s0 = a
lui $s0, 0xFEDC
ori $s0, $s0, 0x8765
```

- lui завантажує 16-бітну константу в старші 16 біт регістра і обнуляє молодші 16 біт

Арифметичні інструкції

Інструкція	Приклад	Зміст	Пояснення
Додавання	add $1,$2,$3	$1=$2+$3	
Віднімання	sub $1,$2,$3	$1=$2-$3	
Додавання безпосереднє	addi $1,$2,100	$1=$2+100	Додавання константи
Додавання беззнакове	addu $1,$2,$3	$1=$2+$3	Беззнакові числа (не за mod2)
Віднімання беззнакове	subu $1,$2,$3	$1=$2-$3	Беззнакові числа (не за mod2)
Додавання безпосереднє беззнакове	addiu $1,$2,100	$1=$2+100	Беззнакові числа (не за mod2)
Множення без переповнення	mul $1,$2,$3	$1=$2*$3	Результат тільки 32 біти !
Множення	mult $2,$3	$hi,$low=$2*$3	Старші 32 біти в hi, молодші 32 біти в ho
Ділення	div $2,$3	$hi,$low=$2/$3	Залишок в hi, частка в lo
Пересилання з hi	mfhi $1	$1=Hi	
Пересилання з lo	mflo $1	$1=lo	

Множення, ділення

- Спеціальні регістри для результату: lo, hi
- 32-біти × 32-біти - множення, 64-бітний результат
  - mult $s0, $s1
  - Результат в {hi, lo}
- 32-бітне ділення, 32-бітні частка і залишок від ділення
  - div $s0, $s1
  - частка в lo
  - залишок від ділення в hi
- Зчитування із спеціальних регістрів lo/hi в регістри загального призначення
  - mflo $s2
  - mfhi $s3

Умовні і безумовні переходи

Інструкція	Приклад	Зміст	Пояснення
Перехід за рівністю	beq $s, $t, C	PC+=4, if (s==t) PC+=(C<<2)	
Перехід за не рівністю	bne $s, $t, C	PC+=4, if (s!=t) PC+=(C<<2)	
Перехід за <= 0	blez $s, $t, C	PC+=4, if (s<=0) PC+=(C<<2)	
Перехід за > 0	bgtz $s, $t, C	PC+=4, if (s>0) PC+=(C<<2)	
Перехід за не рівністю	bne $s, $t, C	PC+=4, if (s!=t) PC+=(C<<2)	
Перехід за < 0	bltz $s, $t, C	PC+=4, if (s<0) PC+=(C<<2)	
Перехід за >= 0	bgez $s, $t, C	PC+=4, if (s>=0) PC+=(C<<2)	
Безумовний перехід jump	J A	PC=(PC & 0xf0000_0000) | (A<<2)	
Непрямий перехід jump register	Jr $sPC=$s		

Переходи

- Дозволяють виконати інструкції не тільки послідовно, а при необхідності змінити порядок їх виконання
- Типи переходів:
  - **Умовні**
    - Перехід за заданою адресою, якщо вміст двох регістрів однаковий (branch if equal, beq)
    - Перехід за заданою адресю, якщо вміст двох регістрів не однаковий (branch if not equal, bne)
  - **Безумовні**
    - Безумовний перехід за адресою, визначений константою (jump, j)
    - Безумовний перехід за адресою, яка зберігається у регістрі загального призначення (jump register, jr)
    - Безумовний перехід за адресою процедури, визначеною константою, із збереженням адреси повернення з процедури в 31-й регістр (jump and link, jal)

Умовний перехід, якщо рівність (beq)

```
# Код на асемблері MIPS
 addi $s0, $0, 4          # $s0 = 0 + 4 = 4
 addi $s1, $0, 1          # $s1 = 0 + 1 = 1
 sll  $s1, $s1, 2         # $s1 = 1 << 2 = 4
 beq  $s0, $s1, target    # перехід на позначку target
 addi $s1, $s1, 1         # інструкція не виконується
 sub  $s1, $s1, $s0       # інструкція не виконується
target:                   # позначка
 add  $s1, $s1, $s0       # $s1 = 4 + 4 = 8
```

Позначка в коді асемблер визначає інструкцію, на яку виконується перехід. Позначка є текстом, який не може бути ключовим словом асемблера і за яким ставиться двокрапка (:). Позначки у асемблерному коді є посиланнями на інструкції програми. Коли асемблерний код транслюється в машинний, позначки замінюються відповідними адресами інструкцій

Умовний перехід, якщо рівність (beq)

```
# Код на асемблері MIPS
 addi $s0, $0, 4          # $s0 = 0 + 4 = 4
 addi $s1, $0, 1          # $s1 = 0 + 1 = 1
 sll  $s1, $s1, 2         # $s1 = 1 << 2 = 4
 beq  $s0, $s1, target    # перехід на позначку target
 addi $s1, $s1, 1         # інструкція не виконується*
 sub  $s1, $s1, $s0       # інструкція не виконується
target:                   # позначка
 add  $s1, $s1, $s0       # $s1 = 4 + 4 = 8
```

* На практиці, через конвеєризацію в процесорах MIPS реалізоване так зване відкладене галуження: інструкція, розміщена зразу за умовним або безумовним переходом, виконується завжди не залежно від того, виконується перехід чи ні. Ця особливість не врахована в даному асемблерном коді

Умовний перехід, якщо не рівність

```
# Код на асемблері MIPS
  addi    $s0, $0, 4          # $s0 = 0 + 4 = 4
  addi    $s1, $0, 1          # $s1 = 0 + 1 = 1
  sll     $s1, $s1, 2         # $s1 = 1 << 2 = 4
  bne     $s0, $s1, target    # перехід не виконується
  addi    $s1, $s1, 1         # $s1 = 4 + 1 = 5
  sub     $s1, $s1, $s0       # $s1 = 5 - 4 = 1
target:
  add     $s1, $s1, $s0     # $s1 = 1 + 4 = 5
```

Безумовний перехід (j)

```
# Код на асемблері MIPS
   addi $s0, $0, 4              # $s0 = 4
      addi $s1, $0, 1           # $s1 = 1
      j    target               # перехід на позначку target
      sra  $s1, $s1, 2          # не виконується
      addi $s1, $s1, 1          # не виконується
      sub  $s1, $s1, $s0        # не виконується
   target:
      add  $s1, $s1, $s0  # $s1 = 1 + 4 = 5
```

Безумовний перехід (jr)

```
# Код на асемблері MIPS
0x00002000        addi $s0, $0, 0x2010
0x00002004        jr   $s0
0x00002008        addi $s1, $0, 1
0x0000200C        sra  $s1, $s1, 2
0x00002010        lw   $s3, 44($s1)
```

jr – інструкція R-типу

Високорівневі конструкції

- Умови if, if/else
- Цикли while, for

Умова if

Код на C

```
if (i == j)
  f = g + h;

f = f - i;
```

Код на асемблері MIPS

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
    bne $s3, $s4, L1
    add $s0, $s1, $s2

L1: sub $s0, $s0, $s3
```

У даному випадку код на C перевіряє умову (i == j), а код на асемблері перевіряє протилежну умову (i != j). Результат один і той же.

Умова if/else

Код на C

```
if (i == j)
  f = g + h;
else
  f = f – i;
```

Код на асемблері MIPS

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
      bne $s3, $s4, L1
      add $s0, $s1, $s2
      j   done
L1:   sub $s0, $s0, $s3
done:
      ... #наступні інструкції
```

Цикл while

Код на C

```
// determines the
   power
// of x such that 2^x
   = 128
int pow = 1;
int x   = 0;

while (pow != 128) {
   pow = pow * 2;
   x = x + 1;
}
```

Код на асемблері MIPS

```
# $s0 = pow, $s1 = x

        addi $s0, $0, 1
        add  $s1, $0, $0
        addi $t0, $0, 128
while:  beq  $s0, $t0, done
        sll  $s0, $s0, 1
        addi $s1, $s1, 1
        j    while
done:
```

Код на C перевіряє умову (**pow != 128**), а код на асемблері перевіряє протилежну умову (**pow == 128**). Результат один і той же.

Цикл for

for (ініціалізація; умова; операція циклу)

{оператор/оператори}

- **ініціалізація:** код ініціалізації виконується до того, як цикл for почнеться
- **умова:** перевіряється на початку кожної ітерації
- **операція циклу:** виконується в кінці кожної ітерації
- **{оператор/оператори}:** виконується поки умова істинна. Якщо в результаті перевірки вияснилося, що умова фальшива, виконання циклу for завершується.

Код на C

```
// додає числа від 0 до 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
  sum = sum + i;
}
```

Код на асемблері MIPS

```
# $s0 = i, $s1 = sum
        addi $s1, $0, 0
        add  $s0, $0, $0
        addi $t0, $0, 10
for:    beq  $s0, $t0, done
        add  $s1, $s1, $s0
        addi $s0, $s0, 1
        j    for
done:
```

Перевірка нерівності

Код на C

```
// суммирует степени двойки
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
  sum = sum + i;
}
```

Код на асемблері MIPS

```
# $s0 = i, $s1 = sum
        addi $s1, $0, 0
        addi $s0, $0, 1
        addi $t0, $0, 101
loop:   slt  $t1, $s0, $t0
        beq  $t1, $0, done
        add  $s1, $s1, $s0
        sll  $s0, $s0, 1
        j    loop
done:
```

$t1 = 1 якщо i < 101

Масиви

- Масиви використовуються для доступу до великої кількості даних одного типу
- Масив розміщується в комірках пам’яті із строго послідовними адресами і займає непервний участок пам’яті
- Кожний масив складається з послідовності елементів однакового розміру, і кожний елемент масиву має порядковий номер, який називається і**ндексом**
- **Розмір**: кількість елементів, які містить масив
- Для доступу до елементу масиву з мови С, необхідно знати ім’я масиву і індекс елементу
- Для доступу до елементу масиву з мови асемблера, необхідно знати індекс елементу і адресу початку масиву

Доступ до елементів масиву

```
// Код на C
  int array[5];
  array[0] = array[0] * 2;
  array[1] = array[1] * 2;
# Код на асемблері MIPS
# завантаження базової адреси 0x1234_8000 масиву в $s0
  lui  $s0, 0x1234       #0x1234 завантаження у старшу половину $S0
  ori  $s0, $s0, 0x8000  #0x8000 завантаження у молодшу половину $S0

  lw   $t1, 0($s0)       # $t1 = array[0]
  sll  $t1, $t1, 1       # $t1 = $t1 * 2
  sw   $t1, 0($s0)       # array[0] = $t1

  lw   $t1, 4($s0)       # $t1 = array[1]
  sll  $t1, $t1, 1       # $t1 = $t1 * 2
  sw   $t1, 4($s0)       # array[1] = $t1
```

Масиви і цикл for

```
// Код на C
int array[1000];
int i;

for (i=0; i < 1000; i = i + 1)
     array[i] = array[i] * 8;
```

```
# Код на асемблері MIPS
# $s0 = базова адреса масиву, $s1 = i
# ініціалізація
  lui  $s0, 0x23B8         # $s0 = 0x23B80000
  ori  $s0, $s0, 0xF000    # $s0 = 0x23B8F000
  addi $s1, $0, 0          # i = 0
  addi $t2, $0, 1000       # $t2 = 1000

loop:
  slt  $t0, $s1, $t2       # i < 1000?
  beq  $t0, $0, done       # if not then done
  sll  $t0, $s1, 2         # $t0 = i * 4 (byte offset)
  add  $t0, $t0, $s0       # address of array[i]
  lw   $t1, 0($t0)         # $t1 = array[i]
  sll  $t1, $t1, 3         # $t1 = array[i] * 8
  sw   $t1, 0($t0)         # array[i] = array[i] * 8
  addi $s1, $s1, 1         # i = i + 1
  j    loop                # repeat
done:
```

Виклики функцій

- **Викликаюча функція** (у даному випадку, main)
- **Викликувана функція** (у даному випадку, sum)

Код на C

```
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

int sum(int a, int b)
{
  return (a + b);
}
```

Домовленості про виклики функцій

- **Викликаюча функція:**
  - передає **аргументи** у викликувану функцію
  - передає керування викликуваній функції (робить перехід на початкову інструкцію викликуваної функції)
- **Викликувана функція:**
  - **виконує** необхідні операції
  - **повертає** результат викликаючій функції
  - **повертає керування** у точку виклику (робить перехід на наступну інструкцію за викликом функції)
  - **не повинна змінювати (перезаписувати)** вміст регістров або пам’яті, які використовує викликаюча функція

Домовленості про виклики функцій в MIPS

- **Виклик функції:** здійснюється за допомогою інструкції `jal` (jump and link), яка робить безумовний перехід на адресу початку підпрограми і зберігає адресу повернення в 31-му регістрі ($ra)
- **Повернення** з функції: за допомогою інструкції `jr` (jump register), яка робить безумовний перехід, на адресу повернення, яка зберігається в 31-му регістрі ($ra)
- **Аргументи**: регістри $a0 - $a3
- **Значення. яке повертається**: регістри $v0-$v1
- **Викликувана функція** повинна виконувати обчислення і повертати значення, але не спричиняти побічні ефекти (перезаписувати області пам’яті і регістри, які використовує викликаюча функція)

Виклики функцій

Код на C

```
int main() {
  simple();
  a = b + c;
}

void simple()
{ return;}
```

Код на асемблері MIPS

```
0x00400200 main: jal  simple
0x00400204       add  $s0, $s1,
  $s2
...
0x00401020 simple: jr $ra
```

jal: безумовний перехід на початкову інструкцію функції simple із збереженням адреси повернення регістрі $ra

$ra = PC + 4 = 0x00400204*

jr $ra: безумовний перехід на адресу повернення, яка зберігається в регістрі $ra (0x00400204)

* В дійсності, через конвеєризацію, адресою повернення, яка знаходиться в $ra, є адреса наступної через одну інструкції після jal. Ця особливість не враховується в показаному асемблерному коді і адресою повернення вважається адреса інструкції, наступної після jal.

Аргументи і значення повернення

Домовленості про виклики MIPS:

- Аргументи функції передаються через регістри: $a0 - $a3
- Значення що повертається знаходиться в регістрах: $v0-$v1

Код на C

```
int main()
{
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);  // 4 аргументи
  ...
}

int diffofsums(int f, int g, int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result;                // значення що повертається
}
```

Аргументи і значення повернення

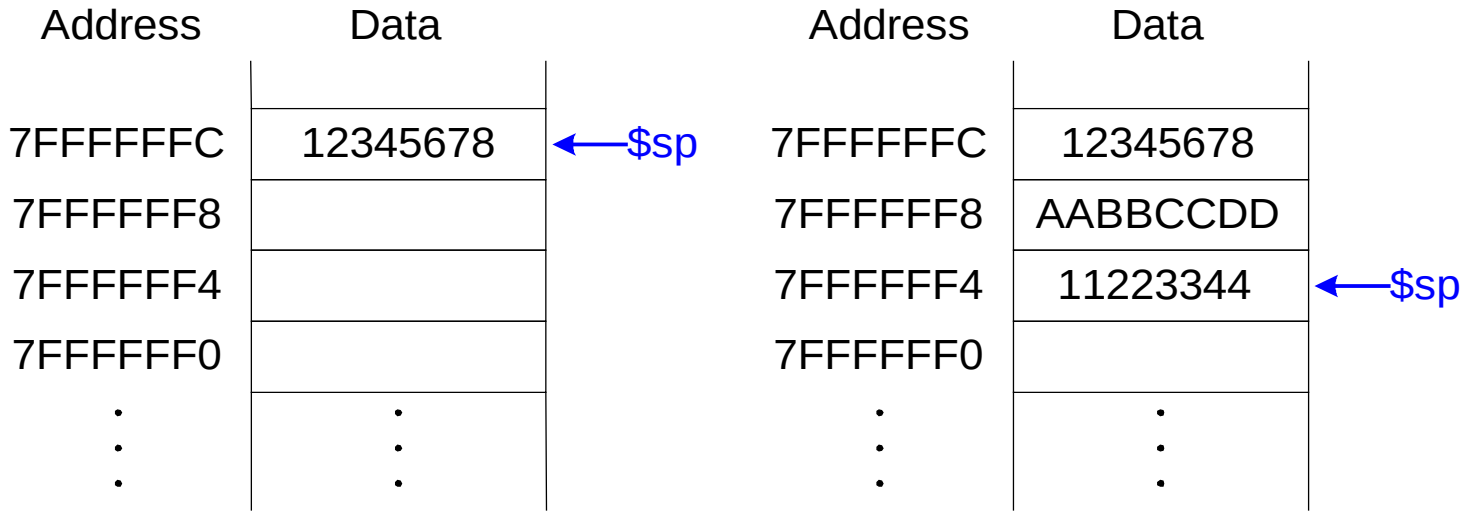
Код на асемблері MIPS

```
# $s0 = y
main:
   ...
   addi $a0, $0, 2     # задання аргументу 0 = 2
   addi $a1, $0, 3     # задання аргументу 1 = 3
   addi $a2, $0, 4     # задання аргументу 2 = 4
   addi $a3, $0, 5     # задання аргументу 3 = 5
   jal  diffofsums     # виклик функції
   add  $s0, $v0, $0   # y = значення що повертається в $v0
   ...

# $s0 = result
diffofsums:            # перезаписує три регістри $t0, $t1, $s0
   add $t0, $a0, $a1   # $t0 = f + g
   add $t1, $a2, $a3   # $t1 = h + i
   sub $s0, $t0, $t1   # result = (f + g) - (h + i)
   add $v0, $s0, $0    # записування значение що повертається в $v0
   jr  $ra             # повернення у викликаючу функцію
```

Стек

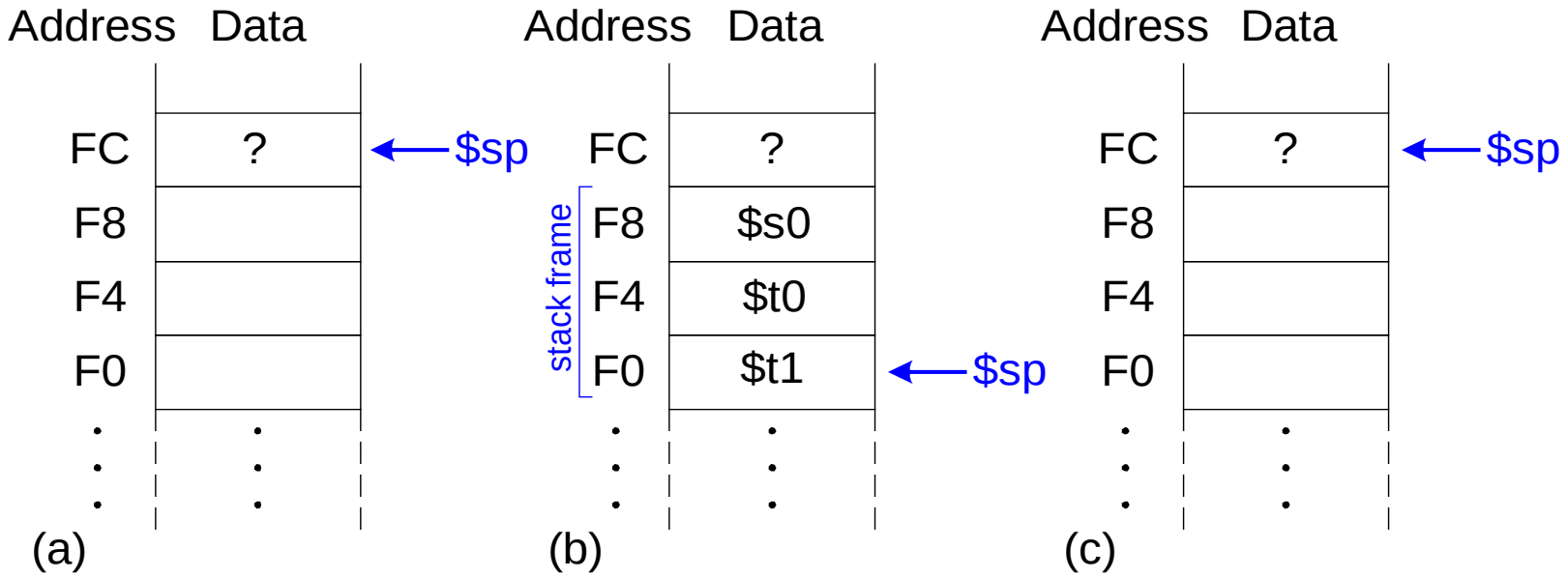
- Стек є спосіб організації пам’яті для тимчасового зберігання змінних
- Стек організований за принципом останній зайшов-перший вийшов (last-in-first-out, LIFO)
- Стек **розширюється** (займає більше пам’яті), якщо процесору потрібно більше місця, і **звужується** (займає менше пам’яті), якщо процесору не потрібні збережені там змінні
- Стек розширюється вниз (в сторону молодших адрес)
- Регістр вказівника стеку (stack pointer, $sp) містить адресу верхівки стеку
- Верхівка стеку – елемент, що знаходиться на верхівці стеку
- При розширенні стеку, його верхівка зсовується в сторону молодших адрес



Зберігання регістрів у стек

```
# $s0 = result
diffofsums:                 # функція перезаписувала регістри $t0, $t1, $s0
  addi $sp, $sp, -12        # виділення місця у стеку
                            # для зберігання 3-х регістрів
  sw   $s0, 8($sp)          # збереження регістру $s0
  sw   $t0, 4($sp)          # збереження регістру $t0
  sw   $t1, 0($sp)          # збереження регістру $t1
  add  $t0, $a0, $a1        # $t0 = f + g
  add  $t1, $a2, $a3        # $t1 = h + i
  sub  $s0, $t0, $t1        # result = (f + g) - (h + i)
  add  $v0, $s0, $0         # put return value in $v0
  lw   $t1, 0($sp)          # відновлення регістру $t1
  lw   $t0, 4($sp)          # відновлення регістру $t0
  lw   $s0, 8($sp)          # відновлення регістру $s0
  addi $sp, $sp, 12         # відновлення розміру стеку
  jr   $ra                  # повернення у викликаючу функцію
```

Стек під час виклику diffofsums



Регістри

Збережувані(Preserved) Зберігає викликувана функція	Не збережувані (Non preserved) Зберігає викликаюча функція
$s0-$s7	$t0-$t9
$ra	$a0-$a3
$sp	$v0-$v1
Стек вище $sp	Стек нище $sp

Виклик функції із функції

```
proc1:
  addi $sp, $sp, -4    # виділення місця в стеку
  sw   $ra, 0($sp)     # збереження регістру $ra
  jal  proc2           # виклик функції proc2
  ...
  lw   $ra, 0($sp)     # відновлення регістру $ra
  addi $sp, $sp, 4     # відновлення розміру стека
  jr  $ra              # повернення у викликаючу функцію
```

Зберігання збережуваних регістрів

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -4   # виділення місця у стеку
                      # для зберігання регістра $s0
  sw  $s0, 0($sp)     # збереження регістра $s0
                      # не потрібно зберігати $t0 або $t1
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result = (f + g) - (h + i)
  add $v0, $s0, $0    # записування результату в  $v0
  lw  $s0, 0($sp)     # відновлення регістру $s0
  addi $sp, $sp, 4    # відновлення розміру стеку
  jr  $ra             # повернення у викликаючу функцію
```

Рекурсивні виклики функцій

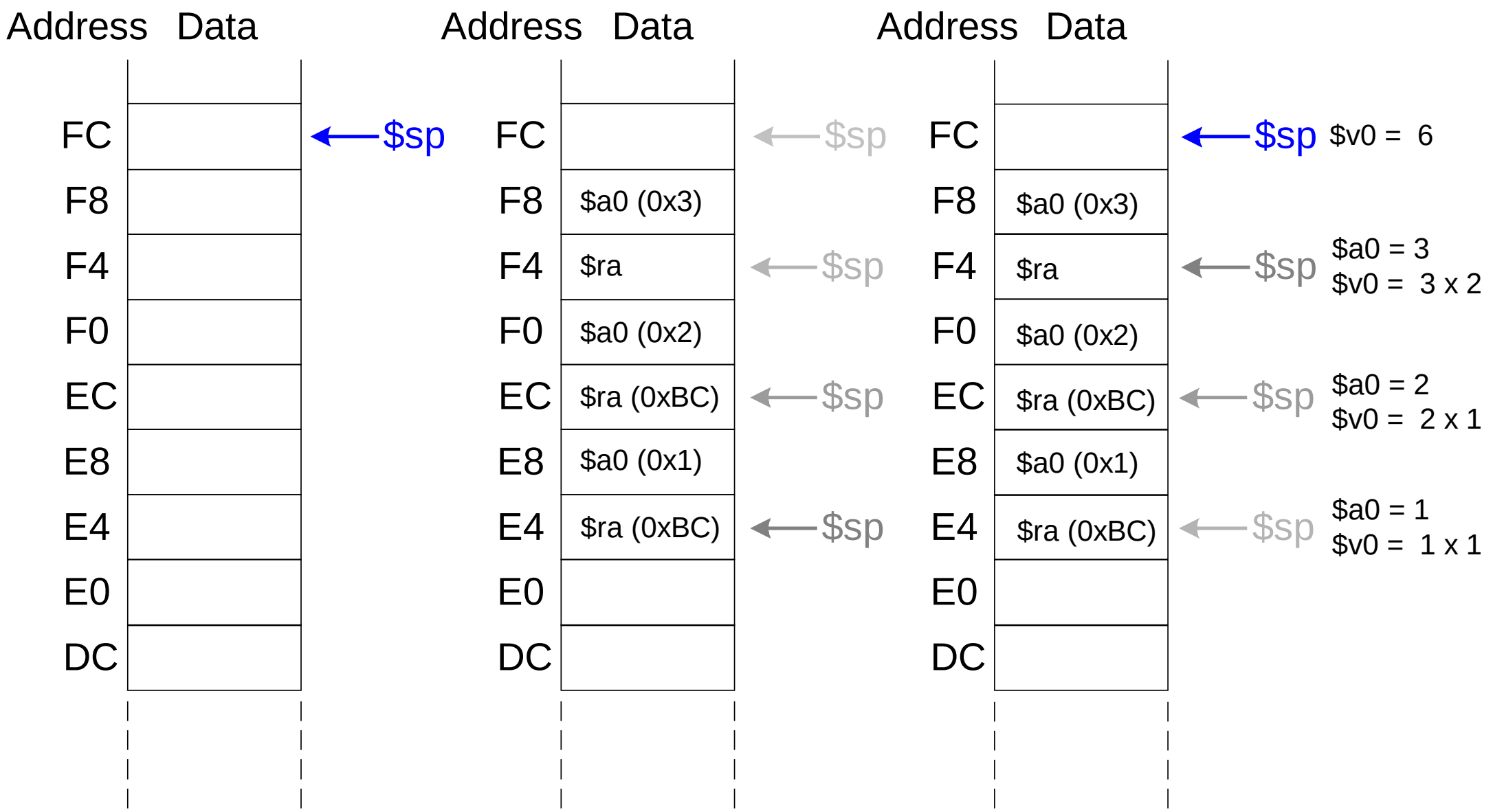
Високорівневий код

```
int factorial(int n) {
  if (n <= 1) return 1;
  else  return (n * factorial(n-1));
}
```

Код на асемблері MIPS

```
0x90 factorial: addi $sp, $sp, -8  # виділення місця у стеку
0x94            sw   $a0, 4($sp)   # збереження $a0
0x98            sw   $ra, 0($sp)   # збереження $ra
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0 # a <= 1 ?
0xA4            beq  $t0, $0, else # ні: перехід до else
0xA8            addi $v0, $0, 1    # так: повернення значення=1
0xAC            addi $sp, $sp, 8   # відновлення $sp
0xB0            jr   $ra           # повернення
0xB4      else: addi $a0, $a0, -1  # n = n - 1
0xB8            jal  factorial     # рекурсивний виклик
0xBC            lw   $ra, 0($sp)   # відновлення $ra
0xC0            lw   $a0, 4($sp)   # відновлення $a0
0xC4            addi $sp, $sp, 8   # відновлення $sp
0xC8            mul  $v0, $a0, $v0 # n * factorial(n-1)
0xCC            jr   $ra           # повернення
```

Стек під час рекурсивних викликів



Виклики функцій: підсумки

- **Викликаюча функція**
  - Зберігає не збережувані регістри, які може модифікувати викликувана функція (наприклад: $t0-t9)
  - Записує аргументи викликуваної функції в $a0-$a3
  - Робить перехід jal на викликувану функцію
  - Після повернення з викликуваної функції зчитує результат з $v0
  - відновлює необхідні не збережувані регістри
- **Викликувана функція**
  - Зберігає збережувані регістри, які планується використати (наприклад $ra, $s0-$s7)
  - Виконує необхідні операції
  - Записує результат в $v0
  - Відновлює збережені збережувані регістри
  - Виконує повернення у викликаючу функцію (jr $ra)

Режими адресації

Способи адресації в MIPS:

- Регістрова (Register)
- Безпосередня (Immediate)
- Базова (Base Addressing)
- Відносно лічильника команд (PC-Relative)
- Псевдопряма (Pseudo Direct)

Режими адресації

Регістрова

- Операнди зберігаються в регістрах
  - **Приклад:** add $s0, $t2, $t3
  - **Приклад:** sub $t8, $s1, $0

Безпосередня

- 16-бітна константа-операнд зберігається безпосередньо у коді інструкції
  - **Приклад:** addi $s4, $t5, -73
  - **Приклад:** ori $t3, $t7, 0xFF

Режими адресації

Базова адресація

- Адреса операнда в пам’яті розраховується як:

  базова адреса в регістрі + 16-бітова знакова константа-зміщення, яка зберігається в тілі інструкції

  - **Приклад:** lw $s4, 72($0)
    - адреса = $0 + 72

  - **Приклад:** sw $t2, -25($t1)
    - адреса = $t1 - 25

Адресація відносно лічильника команд

- Інструкції умовного переходу (`beq, bne`) використовують адресацію відносно лічильника команд для визначення нового значення лічильника команд у тому випадку, якщо потрібно здійснити перехід
- Для визначення нового значения лічильника команд PC, знакове зміщення у байтах додається до адреси інструкції, наступної за інструкцією умовного переходу (PC=PC+4+signed_offset)
- Константа, яка зберігається у тілі інструкції умовного переходу, задає зміщення в 32-розрядних словах. Для отримання signed_offset в байтах, цю константу необхідно помножити на 4 (зсунути вліво на 2 розряди)

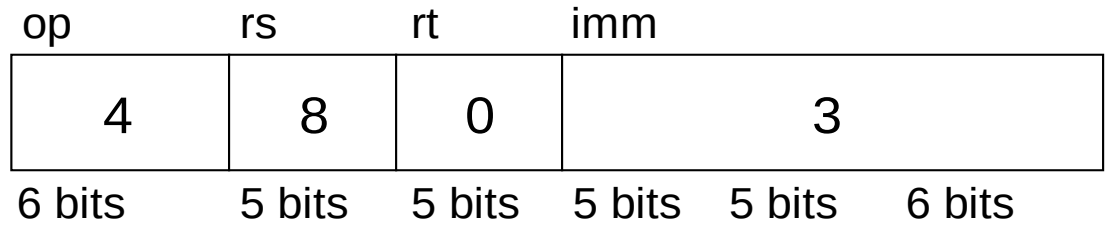
Адресація відносно лічильника команд

```
0x10            beq  $t0, $0, else
0x14            addi $v0, $0, 1
0x18            addi $sp, $sp, i
0x1C            jr        $ra
0x20      else: addi $a0, $a0, -1
0x24            jal  factorial
```

Assembly Code

```
beq $t0, $0, else
(beq $t0, $0, 3)
```

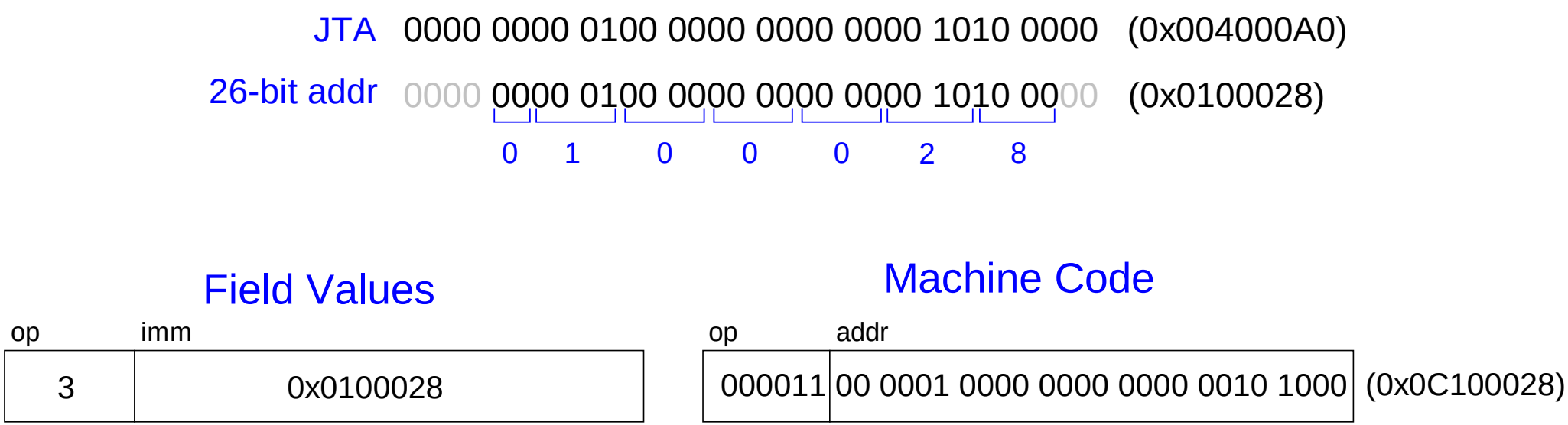
Field Values



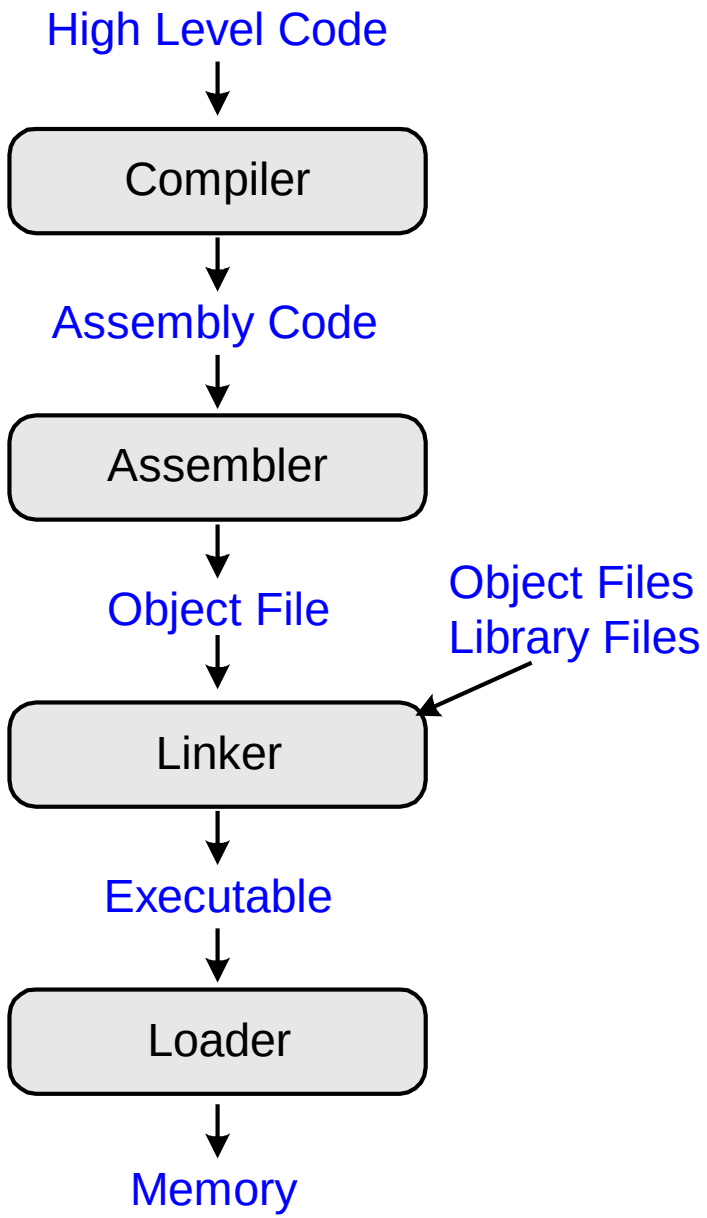
Режими адресації

Псевдопряма адресація

```
0x0040005C          jal     sum
...
0x004000A0  sum:    add     $v0, $a0, $a1
```



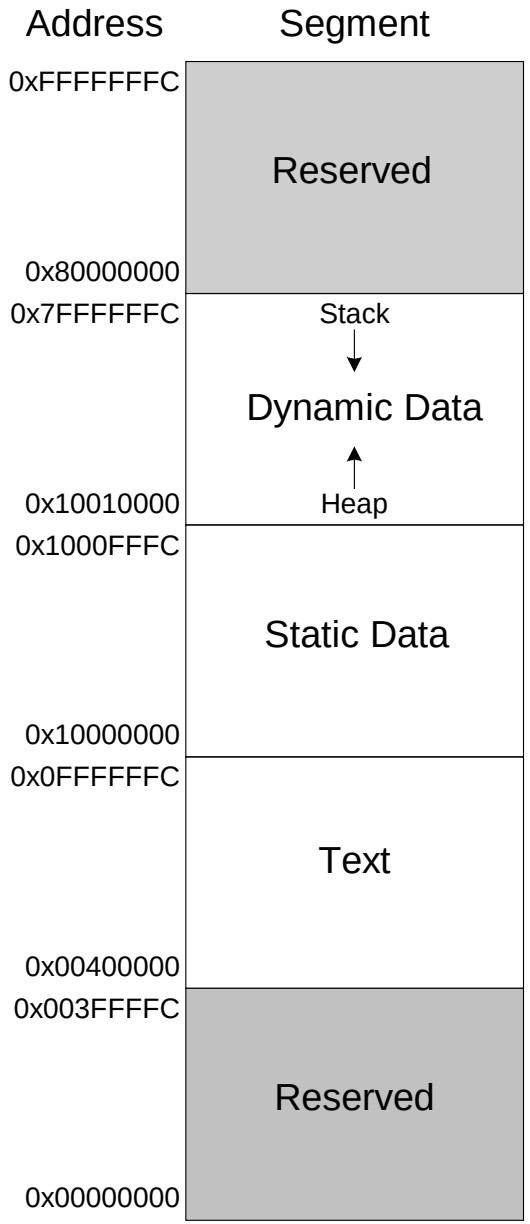
Компіляція і запуск програми



Що зберігається у пам’яті

- Інструкції в сегменті коду (англ.: text segment)
- Дані
  - *сегмент глобальних даних* (англ.: global data segment): містить глобальні змінні, які ініціалізуються до початку виконання програми.
  - *Сегмент динамічних даних* (англ.: dynamic data segment) містить стек і кучу. В момент запуску програми цей сегмент не містить даних – вони динамічно виділяються і звільняються у ньому в процесі виконання програми.
- На скільки велика пам’ять?
  - 2^{32} = 4 гігабайти (4 GB)
  - з адреси 0x0000_0000 до адреси 0xFFFF_FFFF

Карта пам’яті архітектури MIPs



Приклад програми: код на C – асемблер MIPS

```
// глобальні змінні
int f, g, y;

int main(void)
{
  f = 2;
  g = 3;
  y = sum(f, g);
  return y;
}

int sum(int a, int b) {
  return (a + b);
}
```

```
.data
f:
g:
y:
.text
main:
  addi $sp, $sp, -4   # виділення місця
  sw   $ra, 0($sp)    # збереження $ra
  addi $a0, $0, 2     # $a0 = 2
  sw   $a0, f         # f = 2
  addi $a1, $0, 3     # $a1 = 3
  sw   $a1, g         # g = 3
  jal  sum            # виклик sum
  sw   $v0, y         # y = sum()
  lw   $ra, 0($sp)    # відновлення $ra
  addi $sp, $sp, 4    # відновлення $sp
  jr   $ra            # повернення в ОС
sum:
  add  $v0, $a0, $a1  # $v0 = a + b
  jr   $ra            # повернення
```

Приклад програми: таблиця символів

Символ	Адреса
f	0x10000000
g	0x10000004
y	0x10000008
main	0x00400000
sum	0x0040002C

Приклад програми: виконуваний файл

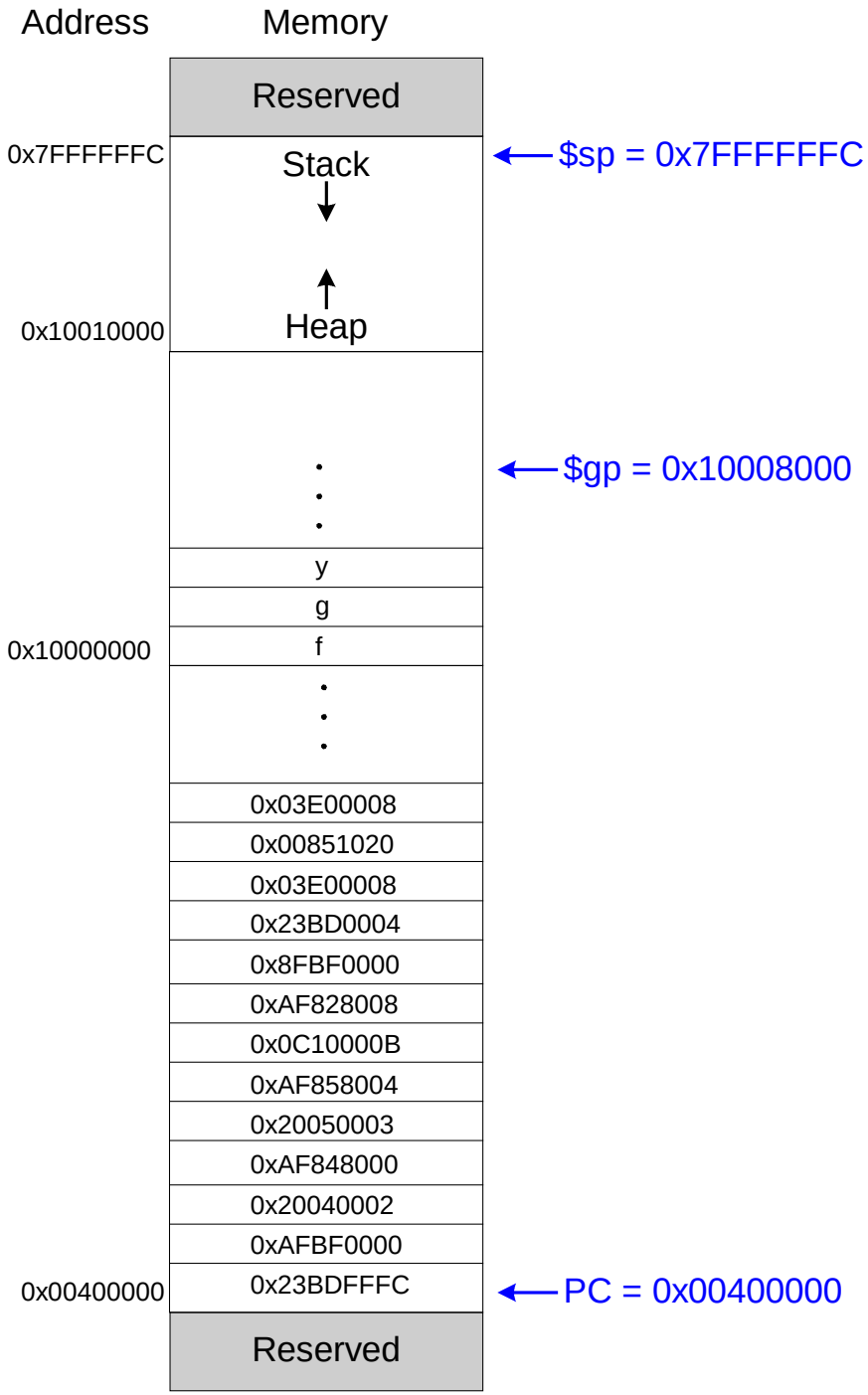
Executable file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)

Text segment	Address	Instruction
	0x00400000	0x23BDFFFC
	0x00400004	0xAFBF0000
	0x00400008	0x20040002
	0x0040000C	0xAF848000
	0x00400010	0x20050003
	0x00400014	0xAF858004
	0x00400018	0x0C10000B
	0x0040001C	0xAF828008
	0x00400020	0x8FBF0000
	0x00400024	0x23BD0004
	0x00400028	0x03E00008
	0x0040002C	0x00851020
	0x00400030	0x03E00008

Data segment	Address	Data
	0x10000000	f
	0x10000004	g
	0x10000008	y

```
addi $sp, $sp, -4
sw   $ra, 0 ($sp)
addi $a0, $0, 2
sw   $a0, 0x8000 ($gp)
addi $a1, $0, 3
sw   $a1, 0x8004 ($gp)
jal  0x0040002C
sw   $v0, 0x8008 ($gp)
lw   $ra, 0 ($sp)
addi $sp, $sp, -4
jr   $ra
add  $v0, $a0, $a1
jr   $ra
```

Приклад програми: розміщення в пам’яті



Додаткові інформація

- Псевдоінструкції
- Винятки
- Інструкції для операцій з плаваючою крапкою

Псевдоінструкції

Псевдоінструкції	Інструкції MIPS
`li $s0, 0x1234AA77`	`lui $s0, 0x1234` `ori $s0, 0xAA77`
`clear $t0`	`add $t0, $0, $0`
`move $s1, $s2`	`add $s2, $s1, $0`
`nop`	`sll $0, $0, 0`

Винятки

- Винятки (англ.: exception) – незапланований виклик функції-обробника винятку (*exception handler*)
- Винятки бувають:
  - апаратні (наприклад, натискання клавіші на клавіатурі). Такі винятки називають *перериваннями (*англ.: *interrupt).*
  - програмні (наприклад, невідома інструкція, ділення на нуль). Такі винятки називають *пастками (*англ.: *trap).*
- При виникненні винятку процесор:
  - записує в спеціальний регістр код причини винятку
  - зберігає значення лічильника команд (PC) на момент виникнення винятку (адреса інструкції, яка спричинила виняток)
  - переходить на функцію-обробник винятків за адресою 0x8000_0180
  - після виконання обробника виняток повертає керування перерваній програмі, виконуючи перехід за раніше збереженою адресою

Регістри обробки винятків

- Це окремі регістри, які не входять в регістровий файл
  - **Cause**: містить код причини винятку
  - **EPC** (Exception PC): містить значення лічильника команд на момент виникнення винятку
- EPC і Cause є частиною Співпроцесора 0
- Інструкція зчитування регістра Співпроцесора 0 в регістр загального призначення:
  - mfc0 $t0, EPC
  - копіює вміст регістру EPC в $t0

Коди причин винятків

Винятки	Код причини
Апаратне переривання	0x00000000
Системний виклик	0x00000020
Точка зупинки / Ділення на 0	0x00000024
Невідома інструкція	0x00000028
Арифметичне переповнення	0x00000030

Оброблення винятків

- Процесор зберігає код причини винятку регістр Cause і адресу інструкції, яка спричинила виняток в регістр EPC
- Процесор робить перехід на функцію-обробник винятку за адресою 0x8000_0180
- Обробник винятку:
  - зберігає необхідні регістри в стек
  - зчитує регістр причини винятку (Cause) mfc0 $t0, Cause
  - обробляє виняток для даної причини
  - відновлює значення регістрів
  - повертає керування перерваній програмі

    mfc0 $k0, EPC

    jr $k0

Операції з плаваючою крапкою

- Для операцій з плаваючою крапкою використовується окремий співпроцесор (Coprocessor 1)
- MIPS має 32 32-розрядних регістри для зберігання операндів у форматі з плаваючою крапкою одинарної точності ($f0-$f31)
- Для зберігання операндів з подвійною точністю (64 розряди) використовується 2 таких регістри
  - наприклад, $f0 і $f1, $f2 і $f3 і т. д.
  - тому для операцій з числами подвійної точності доступні тільки 16 парних регістрів: $f0, $f2, $f4 і т. д.

Операції з плаваючою крапкою

Ім’я	Номер регістру	Використання
$fv0 - $fv1	0, 2	Значення повернення
$ft0 - $ft3	4, 6, 8, 10	Тимчасові змінні
$fa0 - $fa1	12, 14	Аргументи функцій
$ft4 - $ft5	16, 18	Тимчасові змінні
$fs0 - $fs5	20, 22, 24, 26, 28, 30	Збережувані змінні

Формат інструкцій з плаваючою крапкою

- Opcode = 17 (010001_2)
- Інструкції для одинарної точності:
  - cop = 16 (010000_2)
  - add.s, sub.s, div.s, neg.s, abs.s і т.д.
- Інструкції для подвійної точності:
  - cop = 17 (010001_2)
  - add.d, sub.d, div.d, neg.d, abs.d і т.д.
- 3 операнди-регістра:
  - fs, ft: операнди-джерела
  - fd: операнд-приймач

F-Type

Інші операції з плаваючою крапкою

- Інструкції порівняння (встановлюють/скидують прапор умов fpcond)
  - Перевірка на рівність: c.seq.s, c.seq.d
  - Перевірка на менше: c.lt.s, c.lt.d
  - Перевірка на менше-дорівнює: c.le.s, c.le.d
- Умовні переходи
  - bclf: перехід, якщо fpcond має значення ФАЛШЬ
  - bclt: перехід, якщо fpcond має значення ІСТИНА
- Обмін даними між регістрами з плаваючою крапкою і пам’яттю
  - lwc1: lwc1 $ft1, 42($s1)
  - swc1: swc1 $fs2, 17($sp)