Лекція

з дисципліни: «Комп’ютерна схемотехніка та архітектура комп’ютерів»

на тему: «Архітектура MIPS»

ПЛАН ЛЕКЦІЇ

1. Загальний огляд.
2. Мова Асемблера.
3. Операнди: регістри, пам'ять і константи.
4. Машинна мова
5. Виконання програм.
6. Арифметичні та логічні інструкції
7. Режими адресації
8. Архітектура x86

ЛІТЕРАТУРА

1. Рябенький В. М. Цифрова схемотехніка / Рябенький В. М., Жуйков В. Я., Гулий В. Д. – Львів: “Новий Світ-2000”, 2020. — 736 с.
2. Девід М. Харріс і Сара Л. Харріс. Цифрова схемотехніка та архітектура комп'ютера / пер. з англ. 2-е вид. - Morgan Kaufman, 2013. - 1621 с.
3. Мельник А.О. Персональні суперкомп’ютери: архітектура, проектування, застосування: монографія / А.О. Мельник, В.А. Мельник. – Львів: Видавництво Львівської політехніки, 2013. – 516 с.

1. Загальний огляд.

Архітектура - це те, як бачить комп'ютер програміст. Вона визначається набором команд (мовою) і місцем знаходження операндів (регістри і пам'ять). Існує безліч різних архітектур, таких як: x86, MIPS (англ. Microprocessor without Interlocked Pipeline Stages), SPARC і PowerPC.

Щоб зрозуміти архітектуру будь-якого комп'ютера, потрібно в першу чергу вивчити його мову. Слова в мові комп'ютера називаються «інструкціями» або «командами», а словниковий запас комп'ютера - «системою команд». Інструкція комп'ютера визначає операцію, яку потрібно виконати, і її операнди. Операнди - це вхідні дані, з якими проводиться операція, і отримувані результати. Операнди можуть перебувати в пам'яті, в регістрах або всередині самої інструкції.

Апаратне забезпечення комп'ютера «розуміє» тільки нулі і одиниці, тому інструкції закодовані двійковими числами в форматі, який називається машинною мовою. Так само як ми використовуємо букви і інші символи для подання мови у вигляді, зручному для зберігання, передачі та інших маніпуляцій, комп'ютери використовують двійкові числа, щоб кодувати машинну мову (Асемблер).

Майже всі архітектури визначають основні інструкції, такі як додавання, віднімання і перехід, які працюють з комірками пам'яті або регістрами.

Архітектура комп'ютера не визначає структуру апаратного забезпечення, яка її реалізує. Найчастіше існують різні апаратні реалізації однієї і тієї ж архітектури. Наприклад, компанії Intel і Advanced Micro Devices (AMD) виробляють різні мікропроцесори, які відносяться до архітектури x86. Всі вони можуть виконувати одні і ті ж програми, але при цьому в їх основі лежить різне апаратне забезпечення, тому ці процесори мають різне співвідношення продуктивності, ціни і енергоспоживання.

Взаємне розташування регістрів, пам'яті, АЛП і інших будівельних блоків, з яких складається мікропроцесор, називають *мікроархітектурою*,

При розробці архітектури необхідно дотримуватись чотирьох простих принципів, сформульованих Паттерсоном і Хеннессі, авторів книги «Архітектура комп’ютера і проектування комп’ютерних систем»,: (1) для простоти дотримуйтеся однаковості; (2) типовий сценарій повинен бути швидким; (3) чим менше, тим швидше; (4) хороша розробка вимагає хороших компромісів.

2. Мова Асемблера

Мова асемблера - це зручне для сприйняття людиною представлення рідної мови комп'ютера. Кожна інструкція мови асемблера задає операцію, яку необхідно виконати, а також операнди, які будуть використані під час виконання.

Інструкції

Найбільш часта операція, яка виконується комп'ютером, - це додавання.

Перша частина інструкції асемблера, add, називається мнемонікою і визначає, яку операцію потрібно виконати. Операція здійснюється над b і c, операндами-джерелами (*операнди*), а результат записується в a, операнд-призначення (*результат*).

Код на мові високого рівня

a = b+c

Код на мові асемблера MIPS

add a, b, c

Формат інструкції на віднімання такий же, як у інструкції add, тільки операція називається sub.

Код на мові високого рівня

a = b-c

Код на мові асемблера MIPS

sub a, b, c

Більш складний код

Код на мові високого рівня

a = b + c − d; // single-line comment

/* multiple-line comment */

Код на мові асемблера MIPS

sub t, c, d # t = c − d

add a, b, t # a = b + t

У програмі на мові асемблера в прикладі коду використовується тимчасова змінна t для зберігання проміжного результату.

При використанні системи команд MIPS типовий сценарій стає швидким тому, що вона включає в себе тільки прості і постійно використовувані команди. Кількість команд спеціально залишають невеликим, щоб апаратне забезпечення для їх підтримки було простим і швидким. Більш складні операції, які використовуються не так часто, виконуються за допомогою послідовності декількох простих команд. З цієї причини MIPS відноситься до комп'ютерних архітектур зі скороченим набором команд (англ .: reduced instruction set computer, RISC). Архітектури з великою кількістю складних інструкцій, такі як архітектура x86 від Intel, називаються комп'ютерами зі складним набором команд (англ .: complex instruction set computer, CISC).

Архітектура RISC використовує невелику кількість різних команд, що зменшує складність апаратного забезпечення і розмір інструкцій. Наприклад, код операції в системі команд, що складається з 64 простих інструкцій, потребує $\log_2 64 = 6$ біт, а в системі команд з 256 складних інструкцій потребує вже $\log_2 256 = 8$ біт. У CISC-машинах складні команди, навіть якщо вони використовуються дуже рідко, збільшують накладні витрати на виконання всіх інструкцій, включаючи і найпростіші.

3. Операнди: регістри, пам'ять і константи

Інструкція працює з операндами. У прикладі коду змінні a, b і c є операндами. Але комп'ютери оперують нулями і одиницями, а не іменами змінних. Інструкція повинна знати місце, звідки вона зможе брати двійкові дані.

Операнди можуть перебувати в регістрах або пам'яті, а ще вони можуть бути константами, записаними в тілі самої інструкції. Комп'ютери використовують різні місця для зберігання операндів, щоб підвищити швидкість виконання і / або більш ефективно розміщувати дані. Звернення до операндів-констант або операндів, що знаходяться в регістрах, відбувається швидко, але вони можуть вмістити лише невелику кількість даних. Інші дані зберігаються в повільній пам'яті.

Регістри

Щоб команди могли швидко виконуватися, вони повинні швидко отримувати доступ до операндів. Але читання операндів з пам'яті займає багато часу, тому більшість архітектур надають невелику кількість регістрів для зберігання найбільш часто використовуваних операндів. Архітектура MIPS використовує 32 регістра, які називають набором регістрів або *регістровим файлом*. Чим менше кількість регістрів, тим швидший до них доступ.

Прочитати дані з невеликого набору регістрів (наприклад, з 32 регістрів) можна набагато швидше, ніж з 1000 регістрів або з великої пам'яті. Невеликі регістрові файли зазвичай складаються з маленького масиву пам'яті SRAM. Такий масив використовує невеликий дешифратор адреси, підключений бітовими лініями до відносно малої кількості комірок пам'яті, завдяки чому кола з найбільшою затримкою виходять коротші, ніж при доступі до великої пам'яті.

Імена регістрів MIPS починаються зі знака $. Змінні a, b і c довільно розміщені в регістрах $s0, $s1 і $s2. Ім'я $s1 вимовляють як «регістр s1». Інструкція складає 32-бітові значення, що зберігаються в $s1 (b) і $S2 (c) і записує 32-бітний результат в $s0 (a).

Код на мові високого рівня

```
a = b+c
```

Код на мові асемблера MIPS

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

MIPS зазвичай зберігає змінні в 18 з 32 регістрів: $s0 - $s7 і $t0 - $t9. Регістри, імена яких починаються на $s, називають зберігаємими (англ.: saved) регістрами. Відповідно до угоди про використання регістрів MIPS ці регістри використовуються для розміщення в них змінних таких, як a, b і c. Зберігаємі регістри мають особливе значення в контексті виклику процедур. Регістри, імена яких починаються з $t, називають тимчасовими (англ.: temporary) регістрами. Вони використовуються для зберігання тимчасових змінних.

Код на мові високого рівня

```
a = b + c − d;
```

Код на мові асемблера MIPS

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

Набір регістрів

В архітектурі MIPS визначено 32 регістри загального призначення. У кожного регістра є ім'я і порядковий номер від 0 до 31. У Табл. 1 перераховані імена, порядкові номери і ролі кожного регістру. $0 завжди містить нуль тому, що ця константа часто використовується в комп'ютерних програмах.

Табл. 1

Назва	Номер	Призначення
$0	0	Константа нуль
$at	1	Тимчасовий регістр для потреб асемблера
$v0–$v1	2–3	Значення функцій, які повертаються
$a0–$a3	4–7	Аргументи функцій
$t0–$t7	8–15	Тимчасові змінні
$s0–$s7	16–23	Зберігаємі змінні
$t8–$t9	24–25	Тимчасові змінні
$k0–$k1	26–27	Тимчасові змінні операційної системи
$gp	28	Глобальний вказівник (англ.: global pointer)
$sp	29	Вказівник стеку (англ.: stack pointer)
$fp	30	Вказівник кадру стеку (англ.: frame pointer)
$ra	31	Регістр адреси повернення з функції

Пам'ять

Дані також можна зберігати в пам'яті. У порівнянні з регістровим файлом пам'ять має багато місця для зберігання даних, але доступ до неї може тривати довше. З цієї причини часто використовувані змінні зберігаються в регістрах. Комбінуючи пам'ять і регістри, програма може отримувати доступ до великих обсягів даних досить швидко. Архітектура MIPS використовує 32 бітні адреси пам'яті і 32-бітові слова з даними.

MIPS використовує побайтову адресацію пам'яті. Це означає, що кожен байт пам'яті має унікальну адресу. Однак, для кращого розуміння, ми спочатку покажемо адресацію пам'яті по кожному слову.

На Рис. 1 зображений масив пам'яті з послівною адресацією. Видно, що кожне 32-бітове слово даних (англ .: word) має унікальну 32-бітову адресу (англ .: word address). І 32-бітові адреси слів, і 32-бітові значення (англ .: data) на Рис. 1 записані в шістнадцятковій системі числення. Як видно з рисунка, значення 0xF2F1AC07 зберігається в пам'яті за адресою 1 (шістнадцяткові числа часто записуються з префіксом 0x). При графічному зображенні пам'яті традиційно розміщують менші адреси внизу, а великі - нагорі.

Рис. 1 Пам'ять по кожному слову

MIPS використовує команду *завантажити слово* (англ .: load word), *lw*, для читання слова даних з пам'яті в регістр. У прикладі коду наведено завантаження слова, що знаходиться в пам'яті за адресою 1, в регістр $s3. Інструкція *lw* визначає необхідну адреса в пам'яті як суму *базової адреси* і *зсуву*. Базова адреса (записаний в дужках в інструкції) є регістром. Зсув (записано перед дужками) є константою. У прикладі коду базова адреса - це регістр $ 0, що містить значення 0, а зсув - це 1, тому інструкція *lw* читає значення з пам'яті за адресою ($ 0 + 1) = 1. Після виконання команди завантаження слова (lw) в $s3 з'являється значення 0xF2F1AC07, яке знаходилося в пам'яті за адресою 1, як було показано на Рис. 1.

Приклад коду на мові асемблера MIPS

```
# This assembly code (unlike MIPS) assumes word-addressable memory
lw $s3, 1($0)        # read memory word 1 into $s0
```

Аналогічним чином MIPS використовує інструкцію зберегти слово (англ .: store word), *sw*, для запису даних з регістра в пам'ять.

Приклад коду на мові асемблера MIPS

```
# This assembly code (unlike MIPS) assumes word-addressable memory
sw $s7, 5($0)        # write $s7 to memory word 5
```

Модель пам'яті MIPS має адресацію не по кожному слову, а побайтову, при якій кожен байт даних має унікальну адресу.

Так як 32-бітове слово складається з чотирьох 8-бітних байтів, то адреса кожного слова (англ .: word address) кратн чотирьом, як показано на Рис.2, де 32-бітові адреси і значення слів теж представлені в шістнадцятковій системі числення.

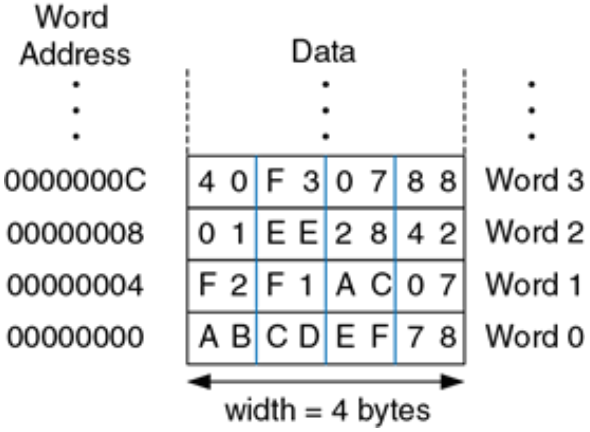

Рис.2 Пам'ять MIPS з побайтовою адресацією

Архітектура MIPS також включає інструкції lb і sb, які завантажують і зберігають окремі байти, а не слова.

Пам'ять з побайтовою адресацією може бути організована з прямим порядком слідування байтів (від молодшого до старшого; англ.: little- endian) або зі зворотнім (від старшого до молодшого; англ.: big-endian), як показано на Рис. 3. В обох випадках найстарший байт (англ.: most significant byte, MSB)

знаходиться зліва, а наймолодший байт (англ.: least significant byte, LSB) – з права.

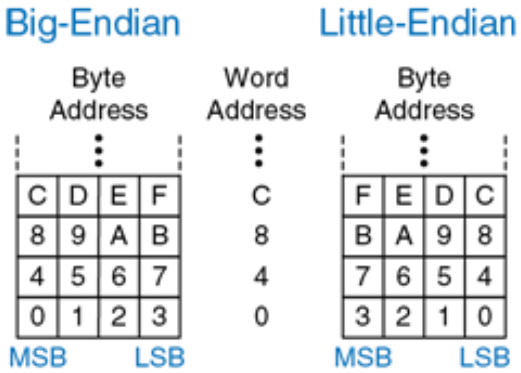

Рис. 3. Адресація даних з прямим та зворотнім порядком слідування байтів

Адресація по словах однакова в обох моделях, тобто одна і та ж адреса слова вказує на одні і ті ж чотири байти.

Архітектура x86 компанії Intel, яка використовується в персональних комп'ютерах, має прямий порядок проходження байтів. Деякі процесори MIPS використовують прямий порядок, інші - зворотний.

Константи

Команди завантаження і збереження слова (lw і sw) також демонструють використання констант в командах MIPS. Ці константи називають безпосередніми операндами (англ .: immediate), тому що їх значення знаходяться безпосередньо усередині команди і не вимагають звертання до регістрів або пам'яті. Ще одна часто використовувана команда MIPS, яка використовує безпосередній операнд - це команда сумування з константою addi (англ .: add immediate), яка додає константу до значення регістра.

Код на мові високого рівня

```
a = a + 4;
b = a − 12;
```

Код на мові асемблера MIPS

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

Константа, що знаходиться всередині команди, є 16-бітовим числом, представленим в додатковому коді, і може приймати значення з діапазону [-32,768; 32,767]. Так як віднімання еквівалентно додаванню з від’ємним числом, то для простоти реалізації в архітектурі MIPS відсутня команда subi.

4. Машинна мова

Програму, написану на мові асемблера, переводять з послідовності мнемонік в послідовність нулів і одиниць, яку називають *машинною мовою*.

Для простоти потрібно дотримуватися одноманітності, і найбільш однаковим представленням команд в машинній мові було б таке, де кожна команда займала б рівно одне слово пам'яті. Довжина команд в архітектурі MIPS

становить 32 біти, при цьому деякі з них використовують тільки частину з цих біт.

Для простоти можна було б також визначити єдиний формат для всіх інструкцій, але такий підхід обернувся б серйозними обмеженнями. В архітектурі MIPS як компроміс використовуються три формати інструкцій: типу R, типу I і типу J.

Інструкції типу R

Назва типу R є скороченням від *регістрового типу* (англ. Register-type). На Рис. 4 показаний машинний формат команди типу R. 32-бітна команда складається з шести полів: op, rs, rt, rd, shamt і funct. Кожне поле складається з п'яти або шести біт.

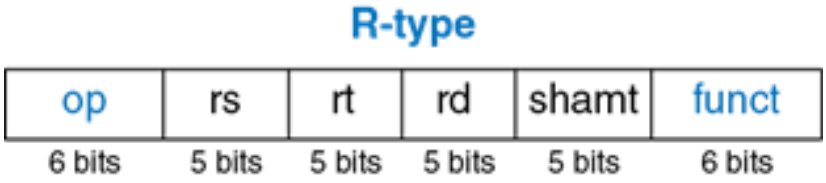

Рис. 4

Операція, яка виконується командою, закодована двома полями, зазначеними синім кольором: полем op (opcode або кодом операції) і полем funct (функцією). У всіх команд типу R поле opcode дорівнює нулю. Операція, яка виконується цими командами, визначається виключно полем funct. Наприклад, поля opcode і funct у інструкції add рівні 0 (000000_2) та 32 (100000_2) відповідно. Аналогічно, у команди sub поля opcode і funct рівні 0 і 34.

Операнди закодовані трьома полями: rs, rt і rd. Поля містять номери регістрів, наведені в Табл. 1. Наприклад, $s0 - це регістр з номером 16. Регістри rs і rt є регістрами-джерелами, а rd - регістром-призначенням (або регістром результату).

П'яте поле, shamt, використовується тільки для операцій зсуву. В таких командах двійкове значення, збережене в 5-бітному поле shamt, задає величину зсуву (англ.: shift amount). У всіх інших команд типу R поле shamt дорівнює 0.

На Рис. 5 показаний машинний код для двох інструкцій типу R - add і sub.

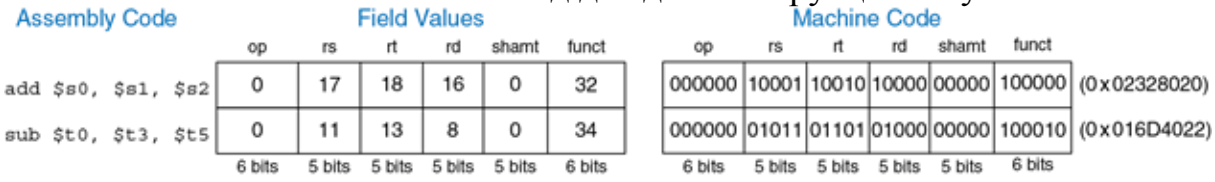

Рис. 5

Інструкції типу I

Назва типу I є скороченням від *безпосереднього* типу (англ. Immediate-type). Інструкції типу I використовують в якості операндів два регістри і один безпосередній операнд (константу).

На Рис. 6. показаний формат машинної команди типу I. 32-бітна команда складається з чотирьох полів: op, rs, rt і imm. Перші три поля (op, rs і rt) аналогічні таким же полям в командах типу R. Поле imm (скор. від англ. Immediate) містить 16-бітну константу.

Рис. 6.

Операція визначається виключно полем opcode, зазначеним синім кольором. Операнди задані в трьох полях: rs, rt і imm. Поля rs і imm завжди використовуються як операнди-джерела. Поле rt в деяких командах (наприклад, addi і lw) містить номер регістра-призначення, в інших (наприклад, sw) - номер регістра-джерела.

Інструкції типу I містять 16-бітну константу imm, але константи беруть участь в 32-бітових операціях. Наприклад, інструкція lw додає 16-бітовий зсув до 32-бітної базової адреси. 16-бітові константи спочатку будуть розширені до 32 біт в такий спосіб: у невід'ємних констант верхні 16 біт будуть заповнені нулями, а у негативних констант вони будуть заповнені одиницями. Після розширення константи буде виконана основна операція інструкції.

Винятком з правила є логічні операції (andi, ori, xori), які замість розширення знаку роблять доповнення нулями, заповнюючи верхню половину тепер уже 32-бітної константи нулями.

Інструкції типу J

Назва типу J є скороченням від англійського слова стрибок (англ .: jump). Цей формат використовується тільки для інструкцій безумовного переходу і розгалуження.

Для безумовних переходів програма може використовувати інструкції трьох типів: звичайний безумовний перехід (j, від англ. Jump), безумовний перехід з поверненням (jal, від англ. Jump and link) і безумовний перехід по регістру (jr, від англ. Jump register) . Безумовний перехід (j) здійснює перехід до інструкції по зазначеній мітці. Безумовний перехід з поверненням (jal) схожий на j, але додатково зберігає адресу повернення і використовується при виклику функцій. Безумовний перехід по регістру (jr) здійснює перехід до інструкції, адреса якої зберігається в одному з регістрів процесора.

Як показано на Рис. 7, в форматі команд цього типу визначений тільки один 26-бітний операнд addr.

Рис. 7

Як і інші команди, команди типу J починаються з 6-бітного поля коду операції (opcode). Решта бітів використовуються для вказівки адреси переходу (addr).

5.Виконання програм

Програма, написана на машинній мові - це послідовність чисел (в архітектурі MIPS - 32-бітних чисел), що представляють інструкції. Як і будь-які інші двійкові числа, ці інструкції можна зберігати в пам'яті. Цей підхід називається концепцією *збереженої програми* (англ .: stored program concept). Використовуючи цей підхід, комп'ютер може виконувати будь-які додатки, починаючи від простого калькулятора і закінчуючи текстовими процесорами і програвачами відео, просто міняючи збережену програму.

У збереженій програмі команди зчитуються, або *вибираються* (англ .: fetch) з пам'яті і виконуються процесором.

На Рис. 8 показано, як машинні команди зберігаються в пам'яті.

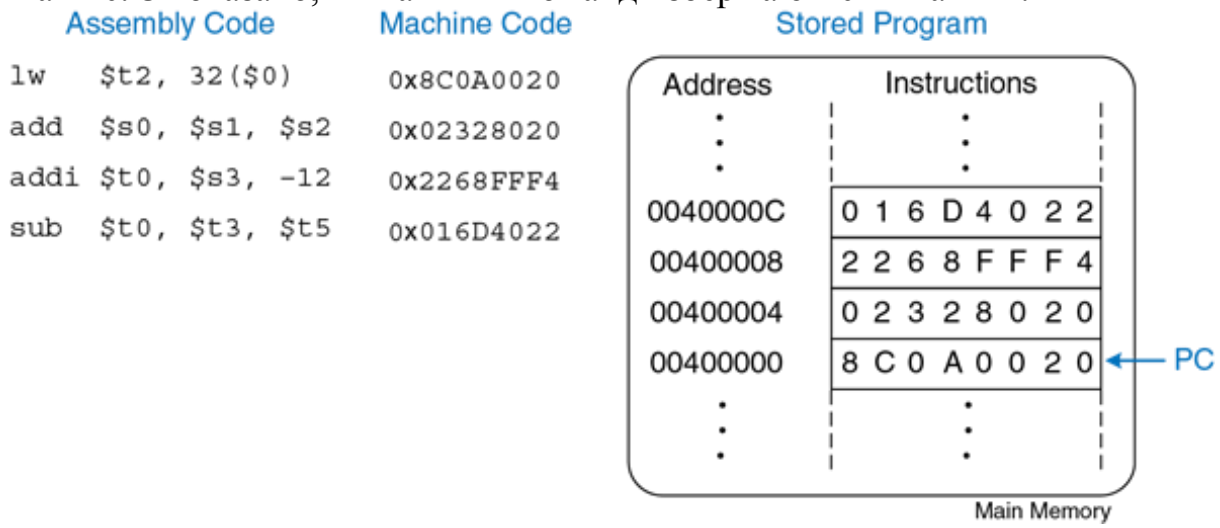

Рис. 8

Щоб запустити, або виконати, збережену програму, процесор послідовно вибирає її команди з пам'яті. Далі обрані команди розшифровуються (дешифруються) і виконуються апаратним забезпеченням. Адреса поточної команди зберігається в 32-бітному регістрі, який називають лічильником команд (англ.: program counter, PC). Лічильник команд - це окремий регістр, він не пов'язаний з 32 регістрами загального призначення.

Архітектурний стан (англ .: architectural state) мікропроцесора містить стан програми. Архітектурний стан процесорів MIPS включає в себе вміст регістрового файлу і лічильника команд. Якщо операційна система в будь-який момент виконання програми збереже архітектурний стан, то зможе цю програму перервати, зробити ще щось, а потім відновити архітектурний стан, після чого перервана програма продовжить виконуватися, навіть не дізнавшись, що її взагалі переривали.

6. Арифметичні та логічні інструкції

В архітектурі MIPS є логічні операції *AND, OR, XOR і NOR*. Відповідні їм однойменні інструкції типу *R* виробляють побітові операції над значеннями двох регістрів-джерел і поміщають результат в регістр-призначення. На Рис. 9 продемонстровані приклади виконання цих операцій з двома вихідними значеннями, 0xFFFF0000 і 0x46A1F0B7.

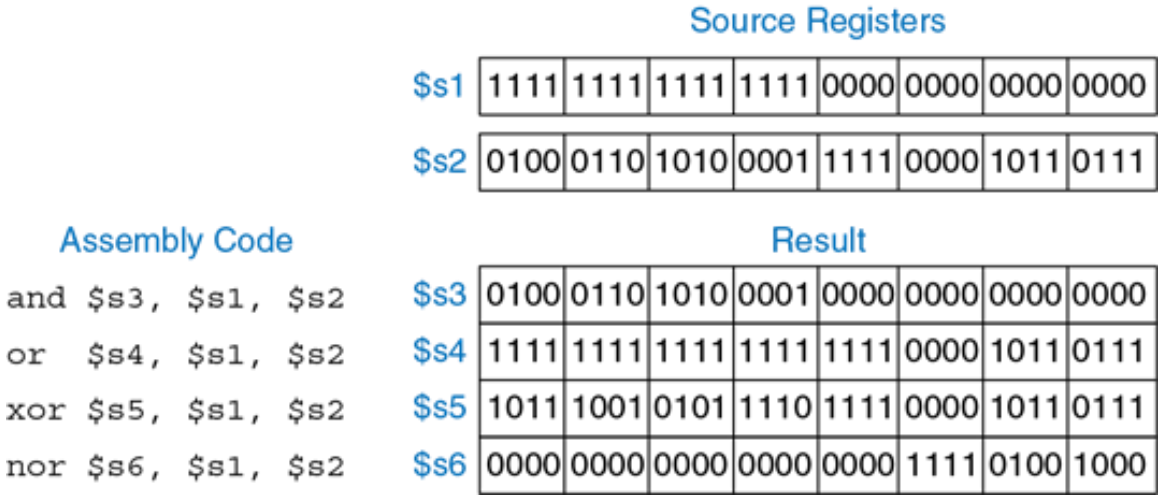

Рис. 9

В архітектурі MIPS не визначена операція инвертирования бітів NOT, але так як A NOR $0 = NOT A, то інструкцію NOR можна використовувати в якості заміни.

Логічні інструкції також можуть працювати з безпосередніми операндами. Це такі інструкції типу *I*, як *andi, ori і xori*.

Інструкції зсуву зсувають значення в регістрі вліво або вправо на будь-яку задану кількість біт, аж до 31. Операції зсуву фактично множать або ділять зсунуті значення на степінь двійки. В архітектурі MIPS існують такі інструкції зсуву: *sll* (логічний зсув вліво), *srl* (логічний зсув вправо) і *sra* (арифметичний зсув вправо).

На Рис. 10 показаний машинний код інструкцій *sll, srl і sra*. У регістрі rt (тобто $ s1) зберігається 32-бітове значення, яке потрібно зсувати, поле shamt задає величину зсуву (4). Результат зсуву поміщається в регістр rd.

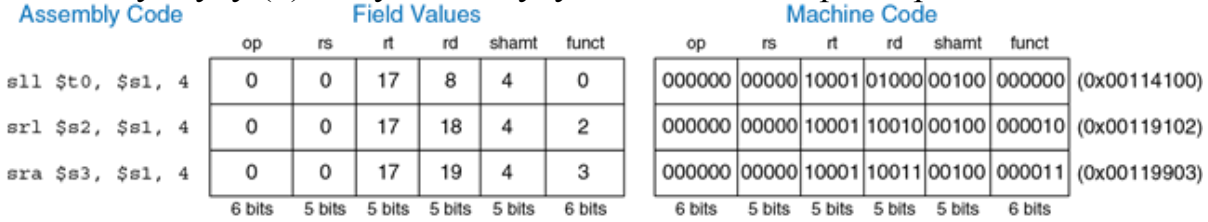

Рис. 10

Завантаженні констант

Інструкцію *addi* використовують для присвоєння змінним значень 16-бітних констант. Для присвоєння змінним значень 32-бітних констант слід використовувати інструкцію lui (англ.: load upper immediate), яка завантажує константу в старші 16 біт регістра і обнулює молодші 16 бітів, а також інструкцію ori для завантаження константи в молодші 16 біт без зміни старших.

Умовні переходи

Система команд MIPS містить дві основні інструкції умовного переходу: *розгалуження при рівності* (beq, від англ. Branch if equal) і *розгалуження при нерівності* (bne, від англ. Branch if not equal). Інструкція beq здійснює перехід, коли значення двох регістрів однакове, а bne здійснює перехід, якщо воно не однакове.

7. Режими адресації

В архітектурі MIPS використовуються п'ять режимів адресації: регістровий (англ.: register-only), безпосередній (англ.: immediate), базовий (англ.: base), відносно лічильника команд (англ.: PC-relative) і псевдопрямий (англ.: pseudo-direct). Перші три режими (регістровий, безпосередній і базовий) визначають способи читання і запису операндів. Останні два (режим адресації щодо лічильника команд і псевдопрямий режим) визначають способи запису лічильника команд (англ.: program counter, PC).

Регістрова адресація

При регістрової адресації регістри використовуються для всіх операндів-джерел і операндів-призначень (іншими словами - для всіх операндів і результату). Всі інструкції типу R використовують саме такий режим адресації.

Безпосередня адресація

При безпосередній адресації в якості операндів поряд з регістрами використовують 16-бітові константи (безпосередні операнди). Цей режим адресації використовують деякі інструкції типу I, такі як додавання з константою (addi) і завантаження константи в старші 16 біт регістра (lui).

Базова адресація

Інструкції для доступу в пам'ять, такі як завантаження слова (lw) і збереження слова (sw), використовують базову адресацію. Ефективна адреса операнда в пам'яті обчислюється шляхом додавання базової адреси в регістрі rs і 16-бітного зміщення з розширеним знаком, що є безпосереднім операндом.

Адресація відносно лічильника команд

Інструкції умовного переходу, або розгалуження, використовують адресацію щодо лічильника команд для визначення нового значення лічильника команд в тому випадку, якщо потрібно здійснити перехід. Зсув зі знаком додається до лічильника команд (PC) для визначення нового значення PC, тому ту адресу, куди буде здійснено перехід, називають адресою відносно лічильника команд.

Псевдопряма адресація

При прямій адресації адреса переходу задається всередині інструкції. Інструкції безумовного переходу j і jal в ідеалі могли б використовувати пряму адресацію для визначення 32-бітного цільової адреси переходу (англ .: jump target address, JTA), що вказує адресу інструкції, яка буде виконана наступною. Однак, в форматі інструкцій типу J немає достатньої кількості біт для того, щоб задати повну 32-бітову адресу переходу. Шість старших біт інструкції займає код операції (поле opcode), тому для адреси переходу залишається тільки 26 біт. На щастя, два молодших біта адреси переходу (JTA1: 0) завжди повинні бути рівні нулю, тому що всі інструкції вирівняні за словами. Наступні 26 біт адреси переходу (JTA27: 2) беруться з поля addr інструкції. Чотири старших біта адреси переходу (JTA31: 28) беруться з чотирьох старших біт значення PC + 4. Такий спосіб адресації називається псевдопрямим.

8. Архітектура x86

Практично всі персональні комп'ютери використовують процесори з архітектурою x86. Архітектура x86, інша назва IA-32 - це 32-розрядна архітектура, спочатку розроблена компанією Intel.

Архітектура x86 є прикладом CISC-архітектури (від англ .: Complex Instruction Set Computer - комп'ютер з повним набором команд). На відміну команд в RISC-архітектурах, таких як MIPS, кожна CISC-команда здатна зробити більше роботи. Через це програми для CISC-архітектур зазвичай складаються з меншої кількості команд. Команди мають змінну довжину, яка часто менше 32 біт. Недолік же полягає в тому, що складні команди важко дешифрувати, до того ж вони, як правило, працюють повільніше.

Команди x86 можуть працювати як з регістрами і безпосередніми операндами, так і з зовнішньою пам'яттю. Це частково компенсує невеликий набір регістрів. Команди x86 містять тільки два операнди: операнд-джерело і операнд-джерело / призначення. Отже, команда x86 завжди записує результат на місце одного з операндів.

У той час як MIPS завжди оперує з 32-бітними словами даних, команди x86 можуть використовувати 8-, 16- або 32-бітові дані.

Архітектура x86 має більшу, ніж у MIPS, систему команд. На відміну від MIPS, де команди завжди мають довжину 32 біти, довжина команди x86 може становити від 1 до 15 байтів.

Архітектура x86 дозволяє писати коротші програми, тому що її складні команди еквівалентні кільком простим командам MIPS і до того ж закодовані так, щоб займати мінімум місця в пам'яті. Однак архітектура x86 - це мішанина з всіляких рішень і функціональності, накопичених за роки розробки. У цій архітектурі занадто мало регістрів, її команди складно дешифрувати, а набір команд важко пояснити.

На сьогоднішній день в світі широко використовується велика кількість різних архітектур, але якщо ви добре зрозумієте одну з них, то вивчити інші буде досить просто. При вивченні нової архітектури ви повинні задати наступні головні питання:

Яка довжина слова даних?

Які регістри доступні?

Як організована пам'ять?

Які є інструкції?