**Вступ**

В ході виконання даного курсового проекту передбачає ознайомлення та опанування архітектуру CISC – комп’ютера. Основні принципи даної архітектури, які запропонував Джон фон Нейман:

1. Інформація кодується в двійковому представленні.

2. Інформація в комп’ютері ділиться на команди і дані.

3. Різнотипні за змістом слова розрізняються за способом застосування, а не по способу кодування.

4. Слова інформації розміщаються в комірках пам’яті та ідентифікуються номерами комірок – адресами слів.

5. Пам’ять є лінійною.

6. Пам’ять має довільну адресацію.

7. Команди і дані зберігаються в одній пам’яті.

8. Алгоритми представляються у вигляді послідовності керуючих слів, як називаються командами. Команда визначається найменуванням операції та слів інформації, які в ній приймають участь. Алгоритм записаний у вигляді послідовності команд, називається програмою.

9. Весь набір виконуваних комп’ютером команд називається системою команд комп’ютера.

10. Виконання обчислень, які визначені алгоритмом, являють собою послідовне виконання команд в порядку визначеному програмою.Для виконання задачі на комп’ютері необхідно:

- забезпечити вибірку команди програми із його пам’яті в заданій послідовності, організувати звернення до неї за відповідними адресами;

- забезпечити розпізнавання типів виконуваних операцій;

- організувати звернення до пам’яті за відповідними адресами для вибірки необхідних для виконання кожної команди даних;

- організувати виконання над даними операцій відповідно до вказівок команд;

- запам’ятати результат обчислень.

Комп'ютер виконує кожну команду як послідовність простих операцій:

1. Вибірка чергової команди із основної пам'яті.

2. Визначення типу вибраної команди, тобто її дешифрування.

3. Визначення адрес даних, необхідних для виконання цієї команди.

4. Виконання операцій пересилання даних (зчитування даних із пам'яті в регістри процесора).

5. Виконання операції відповідно до її коду в полі коду операції команди.

6. Визначення адрес, за якими запам'ятовуються результати.

7. Запам'ятовування результатів.

8. Підготовка до виконання наступної команди, тобто обчислення її адреси. Для процесора комп'ютера із складною системою команд характерні наступні особливості:

- виконання команди за багато тактів, оскільки для цього потрібно здійснити багаторазові операції звернення до основної пам'яті та до програмно-доступних регістрів процесора;

- орієнтація АЛП на виконання великої кількості операцій, що пов'язано з розширеним складом системи команд;

- складна система розпізнавання команди, що пов'язано з великою кількістю методів адресації та великою кількістю форматів команд різної розрядності;

- програмне дешифрування команд з метою зменшення затрат обладнання;

- складна організація конвеєризації виконання команд, що пов'язано, в першу чергу, різнотипністю їх виконання;

- орієнтація структури на виконання команд типу регістр-пам'ять та пам'ять-пам'ять.

Основні елементи процесора - арифметико-логічний пристрій, пристрій керування і регістрова пам'ять або, як її ще називають, надоперативний запам'ятовуючий пристрій. До складу регістрової пам'яті, в свою чергу, входять наступні вузли - програмний лічильник, регістри: адреси, команди, даних, слова стану програми, а також регістровий файл, який складається з програмно доступних регістрів. Cтруктура регістрової (надоперативної) пам'яті процесора складається з регістрів спеціального та зального призначення. До регістрів спеціального призначення належать:

- регістри адреси (РгА);

- регістри команд (РгК);

- програмний лічильник(ПЛ)

- регістри даних (РгД).

РгА зберігає адресу даного або команди при зверненні до основної пам'яті. РгД зберігає операнд при його запису або зчитуванні з основної пам'яті. В ролі операнда може бути дане, команда або адреса. РгК зберігає команду після її зчитування з основної пам'яті. ПЛ підраховує команди та зберігає адресу поточної команди. Комп'ютер з архітектурою Джона фон Неймана має один програмний лічильник. Більшість комп'ютерів мають в складі процесора тригери для зберігання бітів стану процесора, або, як їх іще називають, прапорців. Кожен прапорець має спеціальне призначення. Частина прапорців вказує на результати арифметичних і логічних операцій: додатній результат (Р), від'ємний результат (N), нульовий результат (Z), перенос (С), арифметичне переповнення (V), і т. д. В системі команд комп'ютера є команди, які вказують процесору коли встановити чи скинути ці тригери. Інша частина прапорців вказує режими захисту пам'яті. Існують також прапорці, які вказують пріоритети виконуваних програм. В деяких процесорах додаткові тригери служать для зберігання кодів умов, формуючи регістр кодів умов. Взяті разом описані прапорці формують слово стану програми (ССП), а відповідні тригери - регістр ССП. Регістри загального призначення (РЗП) є програмно доступними. Зазвичай 'їх називають регістровим файлом. Вони можуть використовуватись програмістом в якості регістрів для зберігання вхідних та вихідних даних, а також проміжних результатів обчислень, в якості адресних та індексних регістрів при виконанні операцій модифікації адрес.

**Вхідні дані на проектування.**

1. Реалізація додаткових команд. Необхідно реалізувати 8 додаткових команд. Серед них 3 арифметичні, 3 логічні та 2 команди керування згідно варіанту. Команди не мають повторюватися.

Арифметичні:

|  |  |  |
| --- | --- | --- |
| № | Мнемонічний код | Зміст |
| 1. | DEC regA | Зменшити regA на 1 |
| 2. | XADD regA regB destReg | Додати і обміняти операнди місцями destReg=regA+regB  regA<=>regB |
| 3. | IDIV regA regB destReg | Знакове ділення destReg=regA/regB |

Логічні:

|  |  |  |
| --- | --- | --- |
| № | Мнемонічний код | Зміст |
| 1. | SHL regA regB destReg | Логічний зсув вліво destReg=regA << regB |
| 2. | OR regA regB destReg | Логічне побітове АБО destReg=regA | regB |
| 3. | NEG regA destReg | Заміна знаку на протилежний |

Керування. Умовні переходи:

|  |  |  |
| --- | --- | --- |
| № | Мнемонічний код | Зміст |
| 1. | JMA regA regB offSet | Беззнакове більше if (regA> regB) PC=PC+1+offSet |
| 2. | JMNE regA regB offSet | Не рівно if (regA!= regB) PC=PC+1+offSet |

**Система команд.**

Різноманітність типів даних, форм представлення та опрацювання, необхідні дії для обробки та керування ходом виконання обчислень призводить до необхідності використання різноманітних команд – набора команд. Кожен процесор має власний набір команд, який називається системою команд процесора.

Система команд характеризується трьома аспектами:

- формат,

- способи адресації,

- система операцій.

Форматом команди – є довжина команди, кількість, розмір, положення, призначення та спосіб кодування полів. Команди мають включати наступні види інформації:

- тип операції, яку необхідно реалізувати в даній команді (поле команду операції - КОП)

- місце в пам’яті звідки треба взяти перший операнд (А1);

- місце в пам’яті звідки треба взяти другий операнд (А2);

- місце в пам’яті куди треба помістити результат (А3).

Кожному з цих видів інформації відповідає своя частина двійкового слова – поле. Реальна система команд зазвичай має команди декількох форматів, тип формату визначає КОП. Команда в комп'ютері зберігається в двійковій формі. Вона вказує тип операції, яка має бути виконаною, адреси операндів, над якими виконується операція, та адреси розміщення результатів виконання операції. Відповідно до цього команда складається з двох частин, коду операції та адресної частини. КОП займає k розрядів. Ним може бути закодовано до N = 2k різних операцій. Кількість двійкових розрядів, які відводяться під код операції, вибирається таким чином, щоб ними можна було закодувати всі виконувані в даному комп'ютері операції. Якщо деякий комп'ютер може виконувати Nc різних операцій, то мінімальна розрядність поля коду операції k визначається наступним чином: k = [log Nc], де вираз в дужках означає заокруглення до більшого цілого.

Поле адреси (адресна частина) займає m розрядів. В ньому знаходяться адреси операндів. Кожна адреса займає mi розрядів, де і - номер адреси (і=1,2,...1), 1 – кількість адресних полів. Кожною адресою можна адресувати пам'ять ємністю 2ті слів.

Розмір команди k + m повинен бути узгодженим з розміром даних, тобто бути з ним однаковим або кратним цілому числу, що спрощує організацію роботи з пам'яттю. Як правило, розмір команди рівний 8, 16, 32 біти. При написанні програми крім двійкової можуть використовуватись й інші форми представлення команд: вісімкова, шістнадцяткова, символьна (мнемонічна). Використання вісімкового і шістнадцяткового кодування дозволяє скоротити записи і спростити роботу програміста.

Мнемонічне кодування спрощує процес написання, читання і відлагодження програми. Основний принцип такого кодування - кожна команда представляється 3-х або 4-х буквеним символом, який показує назву команди. Деякі приклади мнемонічного кодування:

ADD - додати (add),

SUB - відняти (subtract),

MPY - перемножити (multiply),

DIV - поділити (divide),

Операнди також представляються символічно. Наприклад команда ADD R Y означає додавання вмісту комірки пам'яті Y до вмісту регістра R. Зауважимо, що операція виконується над вмістом, а не над адресою комірки пам'яті та регістра. Таким чином, з'являється можливість написання машинних програм в символічній формі. Повний набір символічних назв і правила їх використання утворюють мову програмування, відому як асемблерна мова. Символічні імена називаються мнемонічними, а правила їх використання для створення команд і програм називаються синтаксисом мови. Програма, яка переводить із мнемонічного коду асемблерної мови в машинний, називається асемблером.

2. Реалізувати додатковий спосіб адресації. Передбачити, що 3 інструкції

підтримують інший вид адресації згідно варіанту.

Примітка: безадресний варіант передбачає створення стеку та реалізацію додатковий команд наведених в таблиці.

|  |  |
| --- | --- |
| № | Адресація |
| 1 | Безадресна – реалізація стеку. Максимальна глибина 32 слова по 32  розряди. |

Безадресні команди.

|  |  |
| --- | --- |
| Мнемонічний код | Зміст |
| Push | Зчитати з стеку в 1 регістр |
| Pop | Записати в стек з 1 регістру |

Безадресна адресація.

Варіанти інтерпретації бітів (розрядів) поля адреси з метою знаходження операнда називаються способами адресації. Коли команда вказує на операнд, він може знаходитись в самій команді, в основній або зовнішній пам'яті чи в регістровій пам'яті процесора. За роки існування комп'ютерів була створена своєрідна технологія адресації, яка передбачає реалізацію різних способів адресації, чому послужило ряд причин:

- забезпечення ефективного використання розрядної сітки команди;

- забезпечення ефективної апаратної підтримки роботи з масивами даних;

- забезпечення задання параметрів операндів;

- можливість генерації великих адрес на основі малих.

Безадресна – поле адреси в команді відсутнє. Адреса операнда, або немає змісту

або є по замовчуванню(наприклад дії на спеціальним регістром - акумулятором).

Безадресні команди неможливо використати для інших регістрів чи комірок пам’яті.

Одним з різновидів безадресної адресації є використання стеку.

3. Регістри стану: CF –регістр переносу, SF – регістр знаку, ZF – регістр 0.

Регіст ознаки нуля:

|  |  |  |
| --- | --- | --- |
| № | Мнемонічний код | Зміст |
| 1. | CMP regA regB | Порівняти regA regB і встановити прапорці   |  |  |  |  | | --- | --- | --- | --- | |  | CF | SF | ZF | | regA<regB | 1 | 1 | 0 | | regA=regB | 0 | 0 | 1 | | regA>regB | 0 | 0 | 0 | |
| 2. | BSR regA destReg | Побітове сканування в зворотньому  напрямку(від старших до молодших) regA в пошуках  біта з 1 , повертає номер позиції в destReg. Якщо 1  знайдено ZF=1, інакше ZF=0 |
| 3. | JE offSet | Перейти, якщо менше, if(ZF==0)PC=offset |

**Роз’яснення та аналіз основних принципів побудови симулятора комп’ютера.**

***Асемблерна мова та асемблер.***

В першій частині даного курсового проекту необхідно написати програму, яка перетворює вхідну програму на мові асемблер в мову машинних кодів. Програма має перетворити асемблерні імена команд в числові еквіваленти, наприклад асемблерну команду beq в 100, також перетворити символьні імена адрес в числові значення. Результуючий файл має складатися з послідовності 32 бітних інструкцій (біти 31-25 інструкції завжди рівні 0).

Формат лінійки асемблерного коду наступний (<пробіл> означає послідовність табуляцій і/або пробілів):

***мітка*** *<пробіл>* ***інструкція****<пробіл>****поле№1****<пробіл>* ***поле№2****<пробіл>* ***поле№3****<пробіл>****коментар***

Крайнє ліве поле лінійки асемблерного коду – поле мітки. Коректна мітка має складатися максимуму з 6 символів, символами можуть бути літери або цифри, але починатися з букви. Поле мітки є необов’язковим, проте пробіл після даного поля є обов’язковим. Мітки дозволяють значно спростити процес написання асемблер них програм, в іншому випадку прийшлось би модифікувати всі адресні частини кожен раз коли додавався рядок коду!

Після не обов’язкової мітки іде обов’язковий пробіл. Далі іде поле назви інструкції, в якому може бути ім’я будь якої асемблерної інструкції зазначені вище в таблиці. Після пробілів ідуть відповідні поля. Всі поля можуть зберігати або десяткові значення або мітки. Кількість полів залежить від інструкції, поля які не використовуються ігноруються.

Інструкції r-типу (add, nand) потребують наявності 3 полів: поле№1 – regA, поле№2 regB поле№3 destReg.

Інструкції і-типу (lw,sw,beq) вимагають 3 полів: поле№1 – regA, поле№2 regB поле№3 – числове значення зміщення чи символьна адреса. Числове значення може бути як додатнім так і відємним. Символьні адреси описані нижче.

Інструкція J-типу (jalr) вимагає 2 полів: поле№1 – regA, поле№2 regB

Інструкція 0-типу (noop, halt) не вимагає жодного.

Символьні адреси посилаються на відповідні мітки. Для інструкцій lw та sw асемблер має згенерувати зміщення, яке дорівнює адресі мітки. Вона може використовуватися з 0 регістром, тоді буде посилання на мітку, або може використовуватися з не нульовим базовим регістром у якості індексу масиву, який починається з мітки. Для інструкції beq, асемблер має перетворити мітку в числове зміщення куди має відбуватися перехід.

Після останнього поля має йти пробіл за яким може розміщуватися коментар. Коментар закінчується з кінцем лінії асемблерної програми. Коментарі дуже важливі для отримання зрозумілої асемблерної програми, тому що інструкції самі по собі мало зрозумілі.

Крім інструкцій СК, асемблерна програма може містити директиви для асемблера. В даному курсовому проекті для асемблера використовується лише одна директива - .fill (зверніть увагу на точку попереду). Директива . fill повідомляє компілятору про те, що він має зберегти число за адресою відповідно де дана інструкція знаходиться. Директива .fill використовує одне поле, в якому може бути як число так і символьна адреса. Наприклад «.fill 32» означає зберегти число 32 за адресою де дана інструкція знаходиться. (Оскільки в нас кожен рядок програми відповідає адресі починаючи з 0, то відповідно адреса буде дорівнювати номеру рядка - 1). Директива . fill з символьною адресою збереже адрессу даної мітки.

Асемблер має виконувати два проходи через асемблерну програму. На першому проході, асемблер має вирахувати адреси кожної символьної мітки. Виходячи з того, що перша інструкція знаходить по нульовій адресі. На другому проході, асемблер має генерувати машинні інструкції (у вигляді десткових чисел) для кожного рядку асемблерної мови. Зауважимо, що програмі імена файлів мають передаватися у якості аргументів командного рядка. Асемблер має зберігати в результуючому файлі лише машинні команди у вигляді десяткових чисел, одну команду в одному рядку. Порушення даного формату призведе до того, що вихідний файл не можна буде виконати. Інший вивід (наприклад для відладки ) програма може виконувати у консоль.

Асемблер має визначати наступні помилки в асемблер ній програмі: використання не визначених міток, використання однакових міток, використання змішення яке перевищує 16 біт, не визначені команди. Асемблер має повертати 1, якщо він визначив помилку та 0 у випадку успішного виходу з програми. Асемблер не має визначати помилки виконання програми, тобто помилки які виникають під час виконання програми.

Одною з частин виконання даного курсової роботи створити можину тестів для того щоб протестувати роботу асемблера. Створення наботу тестів є розповсюдженою практикою при розробці ПЗ. Це дозволяє впевнитися в правильності виконання програми при її модифікаціях. Створення всебічних тестів дозволить глибше зрозуміти специфіку проекту та вашої програми, та допоможе налагодити програму. Множиною тестів буде набір невеличких асемблер них програм в якості вхідних еталонів. При захисті курсової роботи необхідно продемонструвати не менше як 20 тестів, кожен з яких складається не менше ніж з 50 рядків. Важливо створити тести для перевірки можливості асемблера визначити помилки в коді.

***Поведінкова симуляція.***

Другою частино даної курсової роботи є створення програми, яка може від симулювати роботу любого вірного машинного коду СК. Вхідним має бути файл з машинним кодом програми, якій має створити асемблер. Наприклад, якщо назва програми *simulate* та машинний код зберігається в файлі *program.mc,* програма має запускатися наступним чином:

*simulate program.mc > output*

При цьому весь вивід буде виконуватися в файл "output".

Симулятор має розпочинати роботи з ініціалізації вмісту всіх регістрів

Симулятор має виконувати програму доти не зустріне команду halt.

Симулятор має виводити вивід стану комп’ютера перед виконанням кожної інструкції та один раз перед виходом з програми. Вивід стану має включати вивід вмісту всіх регістрів, ПЛ, пам’яті. Пам’ять має виводитися лише для комірок визначених в файлі з машинними кодами (наприклад у наведеному вище прикладі це адреси від 0 до 9).

Так само як і для асемблера, необхідно написати набір тестів і для перевіркироботи симулятора СК. Набір тестів для симулятра є простою задачею адже вже буде набір тестів для асемблера. Отже необхідно дише відібрати коректні програми, в якості вхідних тестових файлів для симулятора. При захисті необхідно буде представити набір тестів і для симулятора. Кожен тест має виконуватися не менше як на 200 інструкціях, набір тестів має складатися не менше ніж з 20 тестових прикладів.

***Асемблерне множення.***

Останньою частиною курсової роботи буде створення асемблерної програми для множення двох чисел. Вхідними будуть числа, які мають зчитуватися з пам’яті розташованій в комірках з назвами "mcand" та "mplier". Результат має бутив першому регістрі при завершені виконання. Вхідні числа мають бути 15 бітовими додатними. Програма множення має бути ефективною, тобто не має перевищувати 50 рядків та виконання не має виходити за 1000 інструкцій для будь-яких вхідних даних. Для цього можна використати цикли та зміщення, алгоритми послідовного сумування є занадто довними.

**Результати роботи.**

В ході виконання курсової роботи був розроблений асемблер, який дозволяє переводити всі команди, які були подані, як вхідні дані у машинні іструкції, а також асемблер виконує всі перевірки, які потрібно виконувати згідно з завданням і у випадку помилки видає відповідне повідомлення.

Правельність роботи асемблера була перевірена за допомогою спеціально розроблених тестів, які написані за допомогою мови асемблера. Ці тести перевіряють корекність роботи всіх реалізованих команд і директив.

Приклад роботи програми асемблера демострує тестова програма:

lw 0 1 x1

lw 0 2 x2

start xadd 1 3 1

idiv 1 3 1

or 1 3 1

shl 1 3 1

neg 1 1

cmp 1 3

bsr 1 3

dec 2

jma 0 2 done

jmne 0 2 start

done halt

x1 .fill 4

x2 .fill 10

x3 .fill 5

В даній програмі використовуються всі команди, які потрібно реалізувавти згідно з завданням. В результаті роботи програми асемблера буде строрений файл машинних інструкцій:

8454157

8519694

38469633

30081025

46858241

42663937

50921472

63635456

67829760

34603008

54657025

58916854

25165824

4

10

5

Даний приклад показує, що програма асемблера корекно перетворила всі команти в машинні інструкці.

Також був розроблений симулятор, який дозволяє дешефрувати всі машинні всі машинні іструкції, і згідно з командандою виконати операцію, а також створю звіт потактового виконання програми з виводом стану памяті і регістрів. Крім того для симулятора була розроблена