**Міністерство освіти і науки України**

**Національний університет «Львівська політехніка»**



Звіт

до лабораторної роботи № 6

з дисципліни «Комп’ютерні системи»

на тему: Побудова функціональної схеми процесорів архітектури RISC CPU

Варіант №6

Виконав:

ст.гр. КІ-38

Гетьман Р.Б.

Прийняв:

Козак Н.Б.

**Львів 2022**

**Мета роботи:** навчитись розробляти нові функціональні модулі з метою розширення функціонування базової архітектури процесорів RISC, що відповідає програмній моделі, а також створення функціональної схеми.

**Теоретичні відомості:**

RISC CPU це процесор, у якому реалізована архітектура із *скороченим* *набором команд* (RISC – Reduced Instruction Set Computer).

Головні зусилля у архітектурі RISC направлені на побудову максимально ефективного конвейєра команд , тобто такого, де всі команди вибираються із пам’яті і поступають у ЦП на обробку у вигляді рівномірного потоку, причому ні одна команда неповинна знаходитися у стані очікування, а ЦП повинен залишатися завантаженим на протязі усього часу.

Цю умову відносно просто можна реалізувати для етапу виборки. Необхідно лише, щоб всі команди мали стандартну довжину, яка дорівнює ширині шини даних, що з’єднує ЦП і пам’ять.

Крім однакової довжини команд, важливо мати відносно просту підсистему декодування і управління: складний пристрій управління (УУ) буде вносити додаткові затримки у формування сигналів управління. Шлях суттєвого спрощення УУ – скорочення числа команд, що входять до складу системи команд ЦП, форматів команд і даних, а також видів адресації.

Якщо підсумувати наведені вище та інші вимоги до архітектури із скороченим набором команд, то концепцію RISC-комп’ютера можна звести до наступних положень:

- виконання всіх (або, не менше, 75 % команд) за один цикл;

- стандартна, у одне слово, довжина всіх команд, яка дорівнює природній довжині слова і ширині шини даних і така, що допускає уніфіковану потокову обробку усіх команд;

- мале число команд (не більше 128);

- мала кількість форматів команд (не більше 4);

- мале число способів адресації (не більше 4);

- доступ до пам’яті тільки через команди „Читання” і „Запис”;

- всі команди, за виключенням „Читання” і „Запис”, використовують внутріщньопроцесорні між регістрові пересилання;

- пристрій управління „жорсткою” логікою;

- відносно великий (не менше 30) процесорний файл регістрів загального призначення (як відомо, у сучасних RISC CPU число РОН може перевищувати 500).

**Завдання:**

Згідно варіанту в журналі та узгоджено з викладачем обрати один з процесорів заданого сімейства на базі RISC архітектури, в звіті навести основну інформацію по процесору, схему та характеристику.

Внести необхідні зміни в проект з 5-ї лабораторної (типи кешів, розмірності шин даних, назви та типи сигналів, операції, буферні модулі і т.д.) з метою реалізації фукнціоналу з обраного процесору. В звіті представити коротко внесені зміни з описом ходу роботи, а також кінцеву функціональну схему програмної моделі.

**6) IBM's Power Architecture**

**Особливості архітектури Power компанії IBM:**

Архітектура POWER у багатьох відношеннях являє собою традиційну RISC-архітектуру. Вона дотримується найбільш важливих відмінних рис RISC: фіксованої довжини команд, архітектури регістр-регістр, простих способів адресації, простих (що не вимагають інтерпретації) команд, великого реєстрового файлу і трехоперандного (неруйнівного) формату команд. Однак архітектура POWER має також декілька додаткових властивостей, які відрізняють її від інших RISC-архітектур.

Одним з головних факторів, який відрізняє архітектуру POWER від багатьох інших RISC-архітектур, є відсутність механізму "затриманих переходів". Зазвичай цей механізм забезпечує виконання команди, наступної за командою умовного переходу, перед виконанням самого переходу. Цей механізм ефективно працював в ранніх RISC-машинах для заповнення "бульбашки", що з'являється при оцінці умов для вибору напрямку переходу та вибірки нового потоку команд. Однак в більш новітніх, суперскалярних машинах, цей механізм може виявитися неефективним, оскільки один такт затримки команди переходу може призвести до появи декількох "бульбашок", які не можуть бути покриті за допомогою одного архітектурного слота затримки. Майже всі такі машини, щоб усунути вплив цих "бульбашок", змушені вводити додаткове обладнання (наприклад, кеш-пам'ять адрес переходів). У таких машинах механізм затриманих переходів стає не тільки мало ефективним, а й привносить значну складність в логіку обробки послідовності команд. Замість цього архітектура переходів POWER була організована для підтримки методики "попереднього перегляду умовних переходів" (branch-lockahead) і методики "згортання переходів" (branch-folding).

**Зміни в реалізації процесора**

Для реалізації підтримки попереднього перегляду умовних переходів був доданий один псевдо-модуль у файл **main.cpp**, що сам по собі не має окремо реалізованого блоку, а лише модифікує отримані дані на етапі фетчингу (**модуль \*Fetch\*)** та декодування (**модуль \*Decode\*).**

**Внесені зміни в реалізації модулів:**

(додані/змінені рядки коду були виділені **чорним** кольором)

**//main.cpp**

**…**

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Branch \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// BPU: fetch\_inst = instruction

// BPU: fetch\_pc = program\_counter

// BPU: fetch\_valid = instruction\_valid

// BPU: branch\_inst\_addr = branch\_instruction\_address

// BPU: branch\_target\_address = branch\_target\_address

// BPU: branch\_valid = branch\_valid

sc\_signal<unsigned > pred\_branch\_address("PRED\_BRANCH\_ADDRESS");

sc\_signal<bool> pred\_branch\_valid("PRED\_BRANCH\_VALID") ;

sc\_signal<bool> pred\_tellid("PRED\_TELLID") ;

sc\_signal<unsigned> pred\_instruction("PRED\_INSTRUCTION") ;

sc\_signal<bool> pred\_inst\_valid("PRED\_INST\_VALID") ;

sc\_signal<unsigned > pred\_inst\_pc("PRED\_INST\_PC");

**…**

**//fetch.h**

struct fetch : sc\_module {

sc\_in<unsigned > ramdata; // instruction from RAM

sc\_in<unsigned > branch\_address; // branch target address

sc\_in<bool> next\_pc; // pc ++

**sc\_in<bool> branch\_valid; // branch\_valid**

sc\_in<bool> stall\_fetch; // STALL\_FETCH

sc\_in<bool> interrupt; // interrrupt

sc\_in<unsigned> int\_vectno; // interrupt vector number

sc\_in<bool> bios\_valid; // BIOS input valid

sc\_in<bool> icache\_valid; // Icache input valid

sc\_in<bool> pred\_fetch; // branch prediction fetch

**sc\_in<unsigned > pred\_branch\_address; // branch target address**

**sc\_in<bool> pred\_branch\_valid; // branch prediction fetch**

sc\_out<bool> ram\_cs; // RAM chip select

sc\_out<bool> ram\_we; // RAM write enable for SMC

sc\_out<unsigned > address; // address send to RAM

sc\_out<unsigned > smc\_instruction; // for self-modifying code

sc\_out<unsigned> instruction; // instruction send to ID

sc\_out<bool> instruction\_valid; // inst valid

sc\_out<unsigned > program\_counter; // program counter

sc\_out<bool> interrupt\_ack; // interrupt acknowledge

sc\_out<bool> branch\_clear; // clear outstanding branch

**sc\_out<bool> pred\_fetch\_valid; // branch prediction fetch**

sc\_out<bool> reset; // reset

sc\_in\_clk CLK;

**//decode.h**

struct decode : sc\_module {

sc\_in<bool> resetin; // input reset

sc\_in<unsigned> instruction; // fetched instruction

sc\_in<unsigned> pred\_instruction;// fetched instruction

sc\_in<bool> instruction\_valid; // input valid

sc\_in<bool> pred\_inst\_valid; // input valid

sc\_in<bool> destreg\_write; // register write enable

sc\_in<unsigned> destreg\_write\_src; // which register to write?

sc\_in<signed> alu\_dataout; // data from ALU

sc\_in<signed> dram\_dataout; // data from Dcache

sc\_in<bool> dram\_rd\_valid; // Dcache read data valid

sc\_in<unsigned> dram\_write\_src; // Dcache data write to which reg

sc\_in<signed> fpu\_dout; // data from FPU

sc\_in<bool> fpu\_valid; // FPU data valid

sc\_in<unsigned> fpu\_destout; // write to which register

sc\_in<bool> clear\_branch; // clear outstanding branch

sc\_in<bool> display\_done; // display to monitor done

sc\_in<unsigned > pc; // program counter from IFU

sc\_in<bool> pred\_on; // branch prediction is on

**sc\_out<unsigned> br\_instruction\_address; //branch invoke instruct**

sc\_out<bool> next\_pc; // next pc ++ ?

**sc\_out<bool> branch\_valid; // branch valid signal**

**sc\_out<unsigned > branch\_target\_address; //branch target address**

sc\_out<bool> mem\_access; // memory access valid

sc\_out<unsigned > mem\_address; // memory physical address

sc\_out<int> alu\_op; // ALU/FPU/MMU Opcode

sc\_out<bool> mem\_write; // memory write enable

sc\_out<unsigned> alu\_src; // destination register number

sc\_out<bool> reg\_write; // not implemented

sc\_out<signed int> src\_A; // operand A

sc\_out<signed int> src\_B; // operand B

sc\_out<bool> forward\_A; // data forwarding to operand A

sc\_out<bool> forward\_B; // data forwarding to operand B

sc\_out<bool> stall\_fetch; // stall fetch due to branch

sc\_out<bool> decode\_valid; // decoder output valid

sc\_out<bool> float\_valid; // enable FPU

sc\_out<bool> mmx\_valid; // enable MMU

sc\_out<bool> pid\_valid; // load process ID

sc\_out<signed> pid\_data; // process ID value

sc\_in\_clk CLK;

Структура самого ж RISC CPU змін не зазнала

Структурна схема:

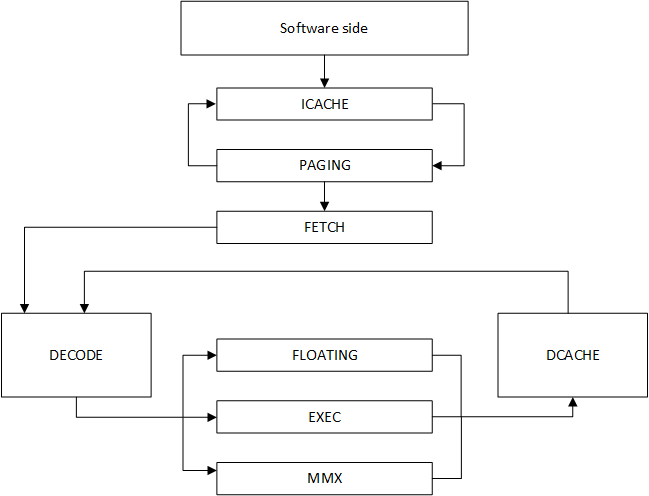


Рис. 1. Структура RISC CPU

**Висновок:** на даній лабораторній роботі я навчився розробляти нові функціональні модулі з метою розширення функціонування базової архітектури процесорів RISC.