**Міністерство освіти і науки України**

**Національний університет «Львівська політехніка»**



Звіт

до лабораторної роботи № 5

з дисципліни «Комп’ютерні системи»

на тему: Дослідження програмної моделі RISC CPU

Варіант №6

Виконав:

ст.гр. КІ-38

Гетьман Р.Б.

Прийняв:

Козак Н.Б.

**Львів 2022**

**Мета роботи:** Навчитися здійснювати оцінку структури об’єкта (RISC CPU) на існуючій програмній моделі. Навчитись встановлювати структуру інтерфейсів об’єкта .

**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:**

1. Дослідити програмну модель RISC CPU;
2. Визначити склад програмної моделі RISC CPU.
3. Визначити призначення блоків у структурі RISC CPU.
4. Визначити зв’язки між структурними блоками RISC CPU (інтерфейси).
5. Визначити структури інтерфейсів між блоками RISC CPU.
6. Визначити окремі потоки у структурі інтерфейсів:

* інформаційні;
* керування.

Програмна модель складається з модулів:

\*ICASHE\* – модуль кешування інструкцій

\*BIOS\* – система реалізації bios

\*Paging\* – модуль сторінкового блоку

\*Fetch\* – модуль вибірки команд.

\*Decode\* – модуль декодування команд

\*DCACHE\* – модуль кешування даних

\*Execute\* – модуль виконання команд

\*Floating point\* – модуль обробки чисел з рухомою комою

\*PIC\* – модуль обробки переривань

\*MMX\* – модуль mmx

Структура інтерфейсу модулів:

1. \*ICHACHE\*

struct icache : sc\_module {

sc\_in<unsigned > datain; // modified instruction

sc\_in<bool> cs; // chip select

sc\_in<bool> we; // write enable for SMC

sc\_in<unsigned > addr; // address

sc\_in<bool> ld\_valid; // load valid

sc\_in<signed> ld\_data; // load data value

sc\_out<unsigned > dataout; // ram data out

sc\_out<bool> icache\_valid; // output valid

sc\_out<bool> stall\_fetch; // stall fetch if busy

sc\_in\_clk CLK;

1. \*BIOS\*

struct bios : sc\_module {

sc\_in<unsigned > datain; // modified instruction

sc\_in<bool> cs; // chip select

sc\_in<bool> we; // write enable for SMC

sc\_in<unsigned > addr; // physical address

sc\_out<unsigned > dataout; // ram data out

sc\_out<bool> bios\_valid; // out valid

sc\_out<bool> stall\_fetch; // stall fetch if output not valid

sc\_in\_clk CLK;

1. \*Paging\*

sc\_in<unsigned > paging\_din; // input data

sc\_in<bool> paging\_csin; // chip select

sc\_in<bool> paging\_wein; // write enable

sc\_in<unsigned > logical\_address; // logical address

sc\_in<unsigned > icache\_din; // data from BIOS/icache

sc\_in<bool> icache\_validin; // data valid bit

sc\_in<bool> icache\_stall; // stall IFU if busy

sc\_out<unsigned > paging\_dout; // output data

sc\_out<bool> paging\_csout; // output cs to cache/BIOS

sc\_out<bool> paging\_weout; // write enable to cache/BIOS

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

sc\_out<unsigned > dataout; // dataout from memory

sc\_out<bool> data\_valid; // data valid

sc\_out<bool> stall\_ifu; // stall IFU if busy

sc\_in\_clk CLK;

1. \*Fetch\*

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;

1. \*Decode\*

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 instruction

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;

1. \*DCACHE\*

struct dcache : sc\_module {

sc\_in<signed> datain; // input data

sc\_in<unsigned> statein; // input state bit MESI(=3210)

sc\_in<bool> cs; // chip select

sc\_in<bool> we; // write enable

sc\_in<unsigned > addr; // address

sc\_in<unsigned> dest; // write back to which register

sc\_out<unsigned> destout; // write back to which register

sc\_out<signed> dataout; // dataram data out

sc\_out<bool> out\_valid; // output valid

sc\_out<unsigned> stateout; // state output

sc\_in\_clk CLK;

1. \*Execute\*

struct exec : sc\_module {

sc\_in<bool> reset; // reset not used.

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

sc\_in<int> opcode; // opcode from ID

sc\_in<bool> negate; // not implemented

sc\_in<int> add1; // not implemented

sc\_in<bool> shift\_sel; // not implemented

sc\_in<signed int> dina; // operand A

sc\_in<signed int> dinb; // operand B

sc\_in<bool> forward\_A; // data forwarding A valid

sc\_in<bool> forward\_B; // data forwarding B valid

sc\_in<unsigned> dest; // destination register number

sc\_out<bool> C; // Carry bit

sc\_out<bool> V; // Overflow bit

sc\_out<bool> Z; // Zero bit

sc\_out<signed int> dout; // output data

sc\_out<bool> out\_valid; // output valid

sc\_out<unsigned> destout; // write to which registers?

sc\_in\_clk CLK;

1. \*Floating point\*

struct floating : sc\_module {

sc\_in<bool> in\_valid; // input valid bit

sc\_in<int> opcode; // opcode

sc\_in<signed int> floata; // operand A

sc\_in<signed int> floatb; // operand B

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

sc\_out<signed int> fdout; // FPU output

sc\_out<bool> fout\_valid; // output valid

sc\_out<unsigned> fdestout; // write to which register

sc\_in\_clk CLK;

10) \*PIC\*

struct pic : sc\_module {

sc\_in<bool> ireq0; // interrupt request 0

sc\_in<bool> ireq1; // interrupt request 1

sc\_in<bool> ireq2; // interrupt request 2

sc\_in<bool> ireq3; // interrupt request 3

sc\_in<bool> cs; // chip select

sc\_in<bool> rd\_wr; // read or write

sc\_in<bool> intack\_cpu; // interrupt acknowledge from CPU

sc\_out<bool> intreq; // interrupt request to CPU

sc\_out<bool> intack; // interrupt acknowledge to devices

sc\_out<unsigned> vectno; // vector number

11) \*MMX\*

struct mmxu : sc\_module {

sc\_in<bool> mmx\_valid; // MMX unit enable

sc\_in<int> opcode; // opcode

sc\_in<signed int> mmxa; // operand A

sc\_in<signed int> mmxb; // operand B

sc\_in<unsigned> dest; // Destination register number

sc\_out<signed int> mmxdout; // MMX output

sc\_out<bool> mmxout\_valid; // MMX output valid

sc\_out<unsigned> mmxdestout; // destination number

sc\_in\_clk CLK;

Структурна схема:

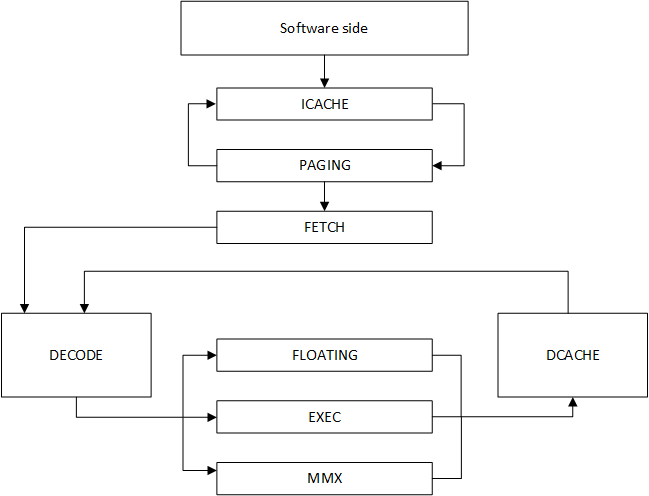


Рис. 1. Структура RISC CPU

Вхідні дані (тестова програма на асемблері **test.asm**)

movi R6, 10

movi R6, 10

movi R6, 10

movi R6, 10

movi R6, 10

movi R6, 10

movi R5, 5

mov R5, R5

mov R5, R5

mov R5, R5

mov R5, R5

mov R5, R5

addi R5, R5, 1

bne R5, R6, -6

add R1, R2, R3

add R2, R3, R4

Результат тестування програми (вміст файлу **printout**)

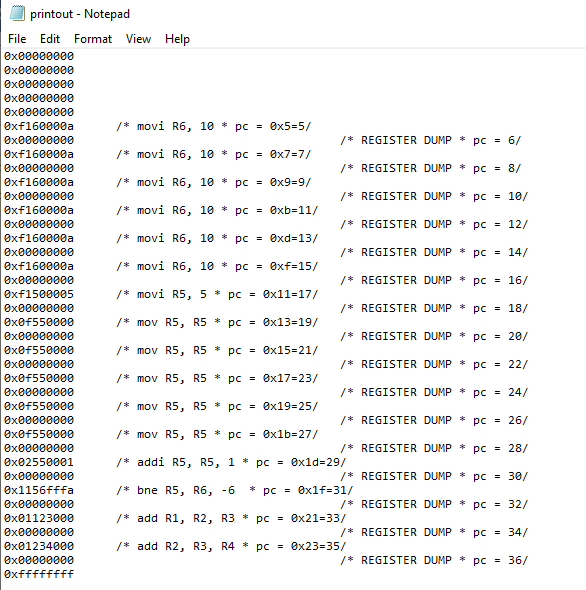


Рис. 2. Результуючий файл

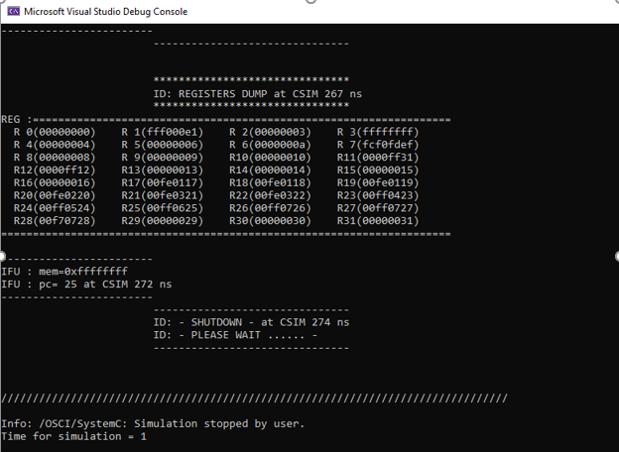


Рис. 3. Вивід в консолі

**Висновок:** Під час виконання даної лабораторної роботи, я навчився здійснювати оцінку структури об’єкта (RISC CPU) на існуючій програмній моделі.