Міністерство освіти і науки, молоді та спорту України

Національний університет “Львівська політехніка”

Кафедра ЕОМ



**Звіт**

з лабораторної роботи № 6

з дисципліни: “Комп’ютерні системи”

на тему: “Побудова функціональної схеми процесорів архітектури RISC CPU”

Виконав: студент гр. КІ-33

Лобай Р. І.

Прийняв: викладач

Козак Н. Б.

Львів

2020

**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 до лабораторної**

1. Реалізувати поставлене завдання до лабораторної роботи;
2. Для кожного з блоків, що входять до складу розробленої програмної моделі на базі RISC CPU визначити вхідні і вихідні сигнали інтерфейсів;
3. Побудувати повну функціональну схему розробленої програмної моделі на базі RISC CPU;
4. Розглянути взаємозв’язок блоків у структурі і визначити можливі типи конвеєрів.

**Номер студента у списку: 9**

**Варіант: 9 mod 7 = 2 – The MIPS line**

**Хід роботи**

**Варіант 2**

**Коротка інформація про архітектуру MIPS**

MIPS (Microprocessor without Interlocked Pipeline Stages) – система команд і мікропроцесорних архітектур, розроблених компанією MIPS Computer Systems згідно з концепцією проектування процесорів RISC (тобто процесорів із спрощеним набором команд)

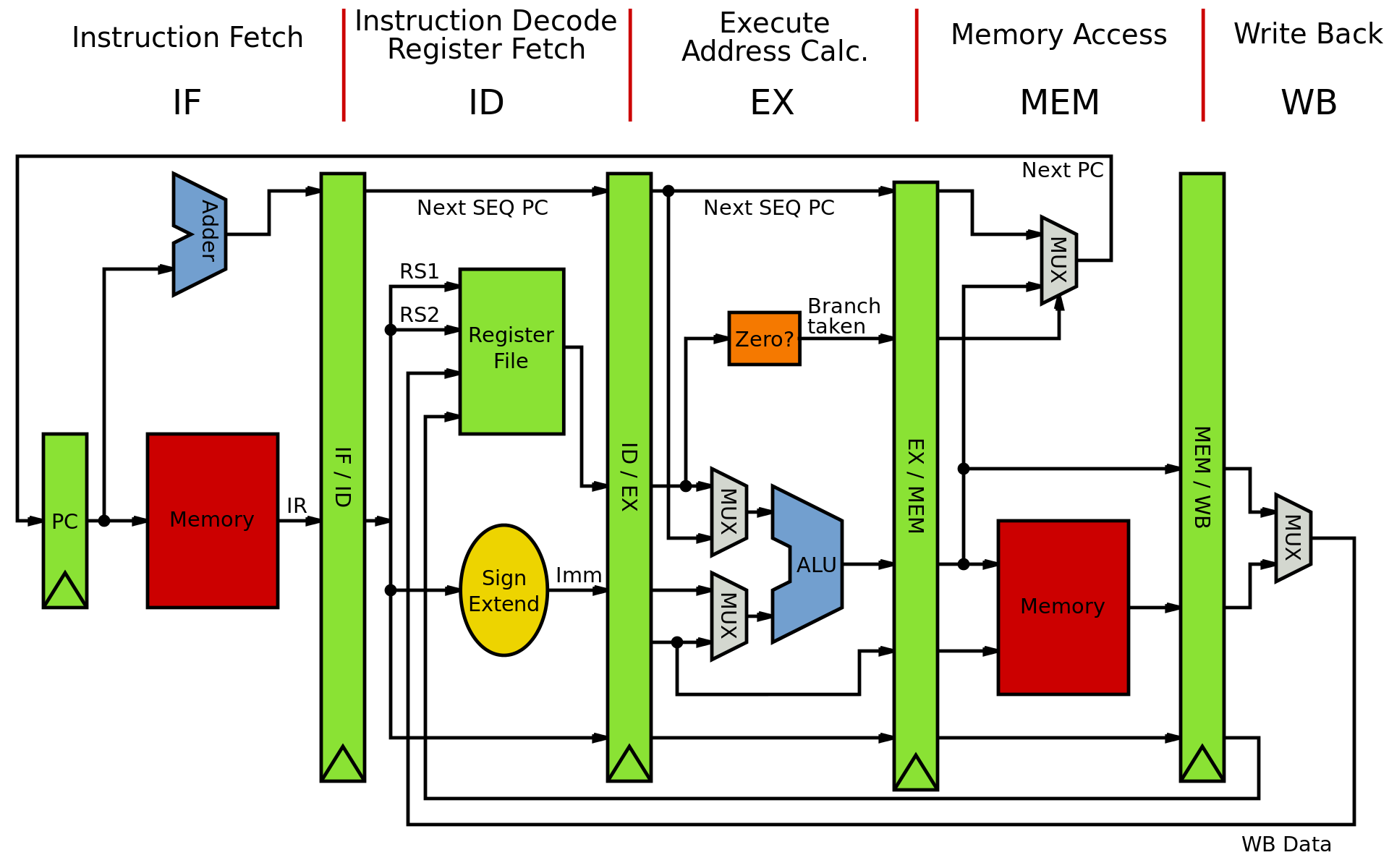


Рис. 1. Конвеєр MIPS, що проходить 5 стадій (отримання інструкції, декодування, виконання, доступ до пам’яті й вивід)

В архітектурі MIPS визначено 32 регістри загального призначення. У кожного регістру є ім’я та порядковий номер від 0 до 31

Табл.1. Набір регістрів MIPS

Інструкції діляться на три типи: R, I та J. Кожна інструкція починається з 6-бітного коду. У доповнення до коду, інструкції R-типу визначають три регістри, область розміру зсуву регістру й область функції; інструкції I-типу визначають два регістри й безпосереднє значення; інструкції J-типу складаються х коду операції та 26-бітного адресу переходу

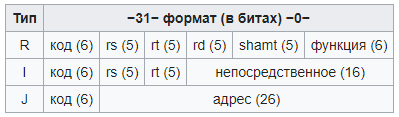


Табл. 2. Формати інструкцій архітектури MIPS

**Результати виконання завдання**

**Код внесених змін**

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;

// Parameter

int memory\_latency; // just a dummy for syntax

void init\_param(int given\_delay\_cycles) {

memory\_latency = given\_delay\_cycles;

}

//Constructor

SC\_CTOR(fetch) {

SC\_CTHREAD(entry, CLK.pos());

}

// Process functionality in member function below

void entry();

};

Fetch.cpp

while (true) {

if (addr\_tmp == 5) {

reset.write(false);

}

if (interrupt.read() == true) {

ram\_cs.write(true);

addr\_tmp = int\_vectno.read();

ram\_we.write(false);

wait(memory\_latency);

datai\_tmp = ramdata.read();

printf("IF ALERT: \*\*INTERRUPT\*\*\n");

cout.setf(ios::hex, ios::basefield);

cout << "------------------------" << endl;

cout << "IFU :" << " mem=0x" << datai\_tmp << endl;

cout << "IFU : pc= " << addr\_tmp;

cout.setf(ios::dec, ios::basefield);

cout << " at CSIM " << sc\_time\_stamp() << endl;

cout << "------------------------" << endl;

instruction\_valid.write(true);

instruction.write(datai\_tmp);

ram\_cs.write(false);

interrupt\_ack.write(true);

if (next\_pc.read() == true) { addr\_tmp++; }

wait();

instruction\_valid.write(false);

interrupt\_ack.write(false);

wait();

}

if (branch\_valid.read() == true) {

printf("IFU ALERT: \*\*BRANCH\*\*\n");

lock\_tmp++;

ram\_cs.write(true);

addr\_tmp = branch\_address.read();

ram\_we.write(false);

wait(memory\_latency);

wait\_until((bios\_valid.delayed() == true) || (icache\_valid.delayed() == true));

datai\_tmp = ramdata.read();

cout.setf(ios::hex, ios::basefield);

cout << "------------------------" << endl;

cout << "IFU :" << " mem=0x" << datai\_tmp << endl;

cout << "IFU : pc= " << addr\_tmp;

cout.setf(ios::dec, ios::basefield);

cout << " at CSIM " << sc\_time\_stamp() << endl;

cout << "------------------------" << endl;

instruction\_valid.write(true);

instruction.write(datai\_tmp);

ram\_cs.write(false);

if (next\_pc.read() == true) { addr\_tmp++; }

wait();

instruction\_valid.write(false);

wait();

}

else {

lock\_tmp = 0;

ram\_cs.write(true);

address.write(addr\_tmp);

ram\_we.write(false);

wait(memory\_latency); // For data to appear

wait\_until((bios\_valid.delayed() == true) || (icache\_valid.delayed() == true));

datai\_tmp = ramdata.read();

cout.setf(ios::hex, ios::basefield);

cout << "------------------------" << endl;

cout << "IFU :" << " mem=0x" << datai\_tmp << endl;

cout << "IFU : pc= " << addr\_tmp;

cout.setf(ios::dec, ios::basefield);

cout << " at CSIM " << sc\_time\_stamp() << endl;

cout << "------------------------" << endl;

instruction\_valid.write(true);

instruction.write(datai\_tmp);

program\_counter.write(addr\_tmp);

branch\_clear.write(false);

ram\_cs.write(false);

if (next\_pc.read() == true) { addr\_tmp++; }

wait();

instruction\_valid.write(false);

wait();

}

if (lock\_tmp == 1) {

branch\_clear.write(true);

wait();

}

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\_in\_clk CLK;

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

signed int cpu\_reg[32]; //CPU register

signed int vcpu\_reg[32]; //virtual CPU register

bool cpu\_reg\_lock[32]; //lock architectural state register

unsigned int pc\_reg; //pc register

unsigned int jalpc\_reg; //jump back register

//Constructor

SC\_CTOR(decode) {

SC\_CTHREAD(entry, CLK.pos());

FILE \*fp = fopen("register", "r");

int size = 0;

unsigned mem\_word;

printf("\*\* ALERT \*\* ID: initialize Architectural Registers\n");

while (fscanf(fp, "%x", &mem\_word) != EOF) {

cpu\_reg[size] = mem\_word;

size++;

}

pc\_reg = 0;

jalpc\_reg = 0;

for (int j = 0; j < 32; j++) vcpu\_reg[j] = 0;

for (int k = 0; k < 32; k++) cpu\_reg\_lock[k] = 0;

}

// Process functionality in member function below

void entry();

};

ICache.h

#define BOOT\_LENGTH 5

#define MAX\_CODE\_LENGTH 500

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;

**Функціональна схема**



**Висновок:**

Під час виконання цієї лабораторної роботи я розробив нові функціональні моделі з метою розширення базової архітектури процесорів RISC