**LIST OF INSTRUCTION:**

**R-type: 0**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Opcode | Rs1 | Rs2 | Rd | Funct3 |
| 15:12 | 11:9 | 8:6 | 5:3 | 2:0 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Order | Command | Opcode | Funct3 | Describe |  |
| 1 | ADD | 0000 | 000 | rd ← rs1 + rs2 | ☑ |
| 2 | SUB | 0000 | 001 | rd ← rs1 – rs2 | ☑ |
| 3 | AND | 0000 | 011 | rd ← rs1 & rs2 | ☑ |
| 4 | OR | 0000 | 100 | rd ← rs1 | rs2 | ☑ |
| 5 | XOR | 0000 | 101 | rd ← rs1 ^ rs2 | ☑ |
| 6 | SLT (set on less than) | 0000 | 110 | Rd= (a<b)?1:0 | ☑ |
| 7 | SGT (set on greater than) | 0000 | 111 | Rd= (a>b)?1:0 | ☑ |
| 8 | SETE (set on equal) | 0000 | 010 | Rd= (a==b)?1:0 | ☑ |

**I-type: 1**

|  |  |  |  |
| --- | --- | --- | --- |
| Opcode | Rs1 | Rd | imm |
| 15:12 | 11:9 | 8:6 | 5:0 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Order | Command | Opcode | Describe |  |
| 1 | ADDI | 0001 | rd ← rs1 + imm | ☑ |
| 2 | SUBI | 0010 | rd ← rs1 - imm | ☑ |
| 3 | ANDI | 0101 | rd← rs1 & imm | ☑ |
| 4 | ORI | 0110 | rd ← rs1 | imm | ☑ |
| 5 | XORI | 0100 | rd ← rs1 ^ imm | ☑ |
| 6 | LSLI | 0011 | rd ← rs1 << imm | ☑ |
| 7 | LSRI | 1111 | rd ← rs1 >> imm | ☑ |

**L-type:**

**LD**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Opcode | Rs1 | Rd | imm |  |
| 15:12 | 11:9 | 8:6 | 5:0 |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | LD | 0111 | rd ← Mem[rs1 + offset] | ☑ |

**LI: 2**

|  |  |  |
| --- | --- | --- |
| Opcode | Rd | imm |
| 15:12 | 11:9 | 8:0 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | LI | 1000 | rd ← imm | ☑ |

**S-type: 3**

|  |  |  |  |
| --- | --- | --- | --- |
| Opcode | Rs1 | Rs2 | Imm |
| 15:12 | 11:9 | 8:6 | 5:0 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Order | Command | Opcode | Describe |  |
| 1 | ST | 1001 | Mem[rs1 + offset] ← rs2 | ☑ |

**B-type: 4**

|  |  |  |
| --- | --- | --- |
| Opcode | Rs1 | Imm |
| 15:12 | 11:9 | 8:0 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Order | Command | Opcode | Describe |  |
| 1 | BEQZ | 1010 | if(rs1 == 0) PC += imm | ☑ |
| 2 | BNQZ | 1011 | if(rs1 ≠ 0) PC += imm | ☑ |

**J-type: 5**

|  |  |
| --- | --- |
| Opcode | Imm |
| 15:12 | 11:0 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Order | Command | Opcode | Describe |  |
| 1 | JMP | 1100 | PC ← PC + imm | ☑ |
| 2 | CALL | 1101 | PC -> Mem[sp]  PC = PC + offset | ☑ |

**SYS-type: 6**

|  |  |  |
| --- | --- | --- |
| Opcode | Imm | Funct3 |
| 15:12 | 11:3 | 2:0 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Order | Command | Opcode | Funct3 | Describe |  |
| 1 | NOP | 1110 | 000 | DO NOTHING | ☑ |
| 2 | RET | 1110 | 001 | COME BACK FROM INTERRUPT | ☑ |
| 3 | EI | 1110 | 010 | ENA INTERRUPT | ☑ |
| 4 | DI | 1110 | 011 | DIS INTERRUPT | ☑ |

A screenshot of a computer program

AI-generated content may be incorrect.

A diagram of a computer process

AI-generated content may be incorrect.

In CPU RISC, basiclly, there are 5 stages in it which is IF- fetch instruction from memory, ID- decode instruction and read registers, EX- perform ALU operation, MEM- access memory if needed( use in load, store, call, return), WB- write result to register( usually uses in R-type, I-type).

**I/ CONSTRUCTION:**

A diagram of a computer

AI-generated content may be incorrect.

**1.1. INSTRUCTION MEMORY:**

Belonging to the above diagram, Instruction memory will be built with (212 /2 ) = 2048 16-bit instructions. My CPU is organised that followed Harvard architecture, which is divided into 2 parts: Data memory and Instruction memory. This action helps CPU to reduce conflict between bus and decrease power consumption. And I will combine between instruction memory block with instruction decoder block. This action will simplify the whole block. With initial instructions inside Ins\_Mem which are represent for commands that we want CPU comply. And I will divide it into 2 parts – 8-bits MSB and 8-bits LSB for ease controll.

In RISC CPU, instructions are designed to be more simple and unified so that it can optimise the processing procedure. Each instructions in MIPS has also the 32-bit static length and divided into 3 baisc forms: R-type, I-type, B-type, J-type and S-type.

Instruction consists of:

+ R-type: This instruction will process inside ALU which are addition, subtraction, logic algorithms and shift left, right. Its construction includes classes:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Opcode | Rs1 | Rs2 | Rd | Funct3 |
| 15:12 | 11:9 | 8:6 | 5:3 | 2:0 |

Where opcode represents what kind of action that ALU will process, which occupies 4 most significant bits. Rs1, Rs2, Rd is a source register and destination register, respectively. That means it contain values for calculating. Due to lack of instruction length so we also design funct3 to expand opcode function, hence we can have more options without increasing of opcode length. And I will show you all things that my CPU can process below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Order | Command | Opcode | Funct3 | Describe |
| 1 | ADD | 0000 | 000 | rd ← rs1 + rs2 |
| 2 | SUB | 0000 | 001 | rd ← rs1 – rs2 |
| 3 | AND | 0000 | 011 | rd ← rs1 & rs2 |
| 4 | OR | 0000 | 100 | rd ← rs1 | rs2 |
| 5 | XOR | 0000 | 101 | rd ← rs1 ^ rs2 |
| 6 | LSL (logical shift left) | 0000 | 110 | rd ← rs1 << rs2 |
| 7 | LSR (logical shift right) | 0000 | 111 | rd ← rs1 >> rs2 |
| 8 | Set on less than | 0000 | 010 | Rd= (a<b)?1:0 |

+ I-type: this type is almost the same to R-type but this action will work with constant instead of registers like R-type. Specially, in I-type, we have 2 commands to load directly value into registers. Its construction includes classes:

|  |  |  |  |
| --- | --- | --- | --- |
| Opcode | Rs1 | Rd | imm |
| 15:12 | 11:9 | 8:6 | 5:0 |

Where Opcode, Rs1, Rd is the same to R-type, the different point is that we have immadiate slot which can contain constant for further calculations.

|  |  |  |  |
| --- | --- | --- | --- |
| Order | Command | Opcode | Describe |
| 1 | ADDI | 0001 | rd ← rs1 + imm |
| 2 | SUBI | 0010 | rd ← rs1 - imm |
| 3 | ANDI | 0101 | rd← rs1 & imm |
| 4 | ORI | 0110 | rd ← rs1 | imm |
| 5 | XORI | 0100 | rd ← rs1 ^ imm |
| 6 | LSLI | 0011 | rd ← rs1 << imm |
| 7 | LD | 0111 | rd ← Mem[rs1 + offset] |

|  |  |  |
| --- | --- | --- |
| Opcode | Rd | imm |
| 15:12 | 11:9 | 8:0 |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | LI | 1000 | rd ← imm |

+ S-type: fundamentally, this is a command that write back value after calculating into data memory, it is completely reverse to load command. And here is construction and detail:

|  |  |  |  |
| --- | --- | --- | --- |
| Opcode | Rs1 | Rs2 | Imm |
| 15:12 | 11:9 | 8:6 | 5:0 |

|  |  |  |  |
| --- | --- | --- | --- |
| Order | Command | Opcode | Describe |
| 1 | ST | 1001 | Mem[rs1 + offset] ← rs2 |

With the opcode 10001, CPU know that it has to put the value/ address from ALU’s output then feed to data memory with address rs1 + offser, where offset equals immediate. Because its control signals from control unit will different so we decided to separate it into many type that we can control easily later.

+ B-type: in almost high-level programming languages, they have many types of conditional command like if, while, for,… So to represent it we have B-type:

|  |  |  |  |
| --- | --- | --- | --- |
| Opcode | Rs1 | Rs2 | Imm |
| 15:12 | 11:9 | 8:6 | 5:0 |

|  |  |  |  |
| --- | --- | --- | --- |
| Order | Command | Opcode | Describe |
| 1 | BEQZ | 1010 | if(rs1 == rs2) PC += imm |
| 2 | BNQZ | 1011 | if(rs1 ≠ rs2) PC += imm |

Actually, we have totally 2 commands such as BEQZ- branch if equal to zero and BNQZ- branch if not equal to zero. Inside ALU, we have a variable zero to check whether 2 registers equal or not then generating a signal to control. If condition is satisfied, index of PC will be increased by immediate.

+ J-type: Just like the above command but in J-type, they don’t need any condition to execute instead of that, they will jump to any position that we want to. Such as go to label in C program.

|  |  |
| --- | --- |
| Opcode | Imm |
| 15:12 | 11:0 |

|  |  |  |  |
| --- | --- | --- | --- |
| Order | Command | Opcode | Describe |
| 1 | JMP | 1100 | PC ← PC + imm |
| 2 | JAL | 1101 | x2 (RA) ← PC + 2 (lưu trực tiếp vào thanh ghi x2)  PC ← PC + imm |

+ SYS-type: These are system control instructions, which typically do not directly manipulate data but instead affect CPU control flow (such as interrupt handling, function calls, returns, etc.).

|  |  |  |
| --- | --- | --- |
| Opcode | Imm | Funct3 |
| 15:12 | 11:3 | 2:0 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Order | Command | Opcode | Funct3 | Describe |
| 1 | NOP | 1110 | 000 | DO NOTHING |
| 2 |  |  |  |  |
| 3 | RET | 1110 | 001 | COME BACK FROM INTERRUPT |
| 4 | EI | 1110 | 010 | ENA INTERRUPT |
| 5 | DI | 1110 | 011 | DIS INTERRUPT |
| 6 | CALL | 1110 | 100 | Nhảy đến địa chỉ và lưu PC hiện tại vào stack |

Input from PC and the output including instruction, opcode, register address also a funct3 for further support opcode. Here is my code:

module Ins\_Mem(address,opcode,rd,rs1,rs2,funct3,instruction);

input [11:0]address;

output [3:0] opcode;

output [2:0] rd;

output [2:0] rs1;

output [2:0] rs2;

output [2:0] funct3;

wire [2:0] instr\_type;

reg [7:0] imem[0:17];

output reg [15:0] instruction;

initial begin

imem[0]<=8'b0000\_1001; // thuc hien phep cong

imem[1]<=8'b0101\_1000;

imem[2]<=8'b1001\_0000; // store rs3 vao rs0 + 5

imem[3]<=8'b1100\_0101;

imem[4]<=8'b0011\_0100;

imem[5]<=8'b0010\_0011;

imem[6]<=8'b0100\_0000;

imem[7]<=8'b0001\_0010;

imem[8]<=8'b0101\_0111;

imem[9]<=8'b0010\_0010;

imem[10]<=8'b0110\_0010;

imem[11]<=8'b0001\_0010;

imem[12]<=8'b0111\_0001;

imem[13]<=8'b0001\_0011;

imem[14]<=8'b1000\_0110;

imem[15]<=8'b0001\_0011;

imem[16]<=8'b1001\_0001;

imem[17]<=8'b0011\_0001;

end

always @(\*)begin

instruction = {imem[address],imem[address+1]};

end

// Tach cac thanh phan

assign opcode = instruction[15:12];

assign instr\_type =

(opcode == 4'b0000) ? 3'd0 : // R-type

(opcode >= 4'b0001 && opcode <= 4'b1000) ? 3'd1 : // I-type

(opcode == 4'b1001) ? 3'd2 : // S-type

(opcode == 4'b1010 || opcode == 4'b1011) ? 3'd3 : // B-type

(opcode == 4'b1100 || opcode == 4'b1101) ? 3'd4 : // J-type

(opcode == 4'b1110) ? 3'd5 : // SYS-type

3'd7; // unknown

// R-type

assign rd = (instr\_type == 3'd0 ) ? instruction[5:3] :(instr\_type == 3'd1 ) ? instruction[8:6] :3'b000 ;

assign rs1 = (instr\_type <= 3'd3) ? instruction[11:9] : 3'b000;

assign rs2 = (instr\_type == 3'd0 || instr\_type == 3'd2 || instr\_type == 3'd3) ? instruction[8:6] : 3'b000;

assign funct3 =

(instr\_type == 3'd0 || instr\_type == 3'd5) ? instruction[2:0] : 3'b000;

endmodule

**1.2. REGISTER FILE:**

For accessing particular registers to save datas or addresses of further purposes, we need a block which contain many different kind of registers. Now, we will build a register file which has 7 registers inside as same as the firgure that I gave you before. I set initial value for all registers for preventing faults. Here is my code:

module reg\_file(rs1,rs2,rd,data,reg\_wrt,readA\_out,readB\_out,r3,clk);

input reg\_wrt, clk;

input [2:0]rs1,rs2,rd;

input [15:0]data;

output [15:0]readA\_out,readB\_out;

output [15:0] r3;

reg [15:0] x [0:7];

initial begin

x[0]=0;//R0 contains zero

x[1]=2; // Stack pointer

x[2]=4; // Return address

x[3]=6; // Function argument/ result

x[4]=8; // General purpose

x[5]=10; // General purpose

x[6]=12; // General purpose

x[7]=14; // Link register/temp/loop/ var

end

always @(posedge clk)

begin

if(reg\_wrt==1)

x[rd]<=data;

end

assign readA\_out = x[rs1];

assign readB\_out = x[rs2];

assign r3 = x[rd];

endmodule

**1.3. MUX:**

Using this multiplexer to select what we allow to go through, data from register file or immediate block. This value will process for different purposes in the future. This will be reused for many times because in the diagram we use 4 2-inputs mux.

module MUX\_alu\_2\_1(input[15:0] B, imm,input alu\_src, output [15:0] outmux );

assign outmux = (alu\_src == 0) ? B : imm;

endmodule

**1.4. ALU CONTROL:**

For controlling the operation of ALU, we have to use a block which is named ALU\_control. This block is responsible for combine signals such as alu\_op and funct3 to create control\_sig. Then, this signal will be decoded into ALU\_control signal such as ADD, SUB, AND,…

module ALU\_control( input [15:0] instr, input [1:0] alu\_op, output reg [3:0] ALU\_control);

wire [2:0] funct3;

wire [4:0] control\_sig;

assign funct3 = instr[2:0];

assign control\_sig = {alu\_op,funct3};

always@(control\_sig)

begin

case(control\_sig)

5'b00\_000: ALU\_control = 4'b0000; // ADD

5'b00\_001: ALU\_control = 4'b0001; // SUB

5'b00\_010: ALU\_control = 4'b0010; // Set on less than

5'b00\_011: ALU\_control = 4'b0011; // AND

5'b00\_100: ALU\_control = 4'b0100; // OR

5'b00\_101: ALU\_control = 4'b0101; // XOR

5'b00\_110: ALU\_control = 4'b0110; // LSL

5'b00\_111: ALU\_control = 4'b0111; // LSR

default: ALU\_control = 4'b0000;

endcase

end

endmodule

**1.5 ALU:**

This is one of the most crucial blocks in CPU. It’s responsible for caluculating numbers, logic algorithms,… And here is my code:

module ALU( input [15:0] A,B,input [3:0]alu\_op, output reg [15:0] ALU\_out, output zero );

assign zero = (ALU\_out==16'd0) ? 1'b1: 1'b0;

always@(\*)

begin

case(alu\_op)

4'b0000: ALU\_out = A + B;

4'b0001: ALU\_out = A - B;

4'b0010: begin if (A<B)

ALU\_out = 1;

else ALU\_out = 0;

end

4'b0011: ALU\_out = A & B;

4'b0100: ALU\_out = A | B;

4'b0101: ALU\_out = A ^ B;

4'b0110: ALU\_out = A << B;

4'b0111: ALU\_out = A >> B;

default: ALU\_out = A + B;

endcase

end

endmodule

**1.6. DATA MEMORY**

Because we decided to build a Harvard CPU architecture so we have 2 separate data memory. This memory will store all datas from the output of ALU or directly register file. Then output of this block will go back to the register file for storing if we want to access in the future.

module data\_mem (

input mem\_write\_en, clk,

input mem\_read\_en,

input [15:0] addr,

input [15:0] write\_data,

output [15:0] read\_data

);

reg [15:0] memory [0:2047];

wire [10:0] addr\_data;

assign addr\_data = addr[10:0];

always @(posedge clk) begin

if (mem\_write\_en) begin

memory[addr] <= write\_data;

end

end

assign read\_data = (mem\_read\_en) ? memory[addr] : 16'd0;

endmodule

**1.7. TOP MODULE**

module TOP( input reg\_wrt, clk, mem\_write\_en, mem\_read\_en, memtoreg,alu\_src, input[2:0]immtype, input[1:0]alu\_op, input[11:0] address, output [15:0] data\_reg,readA\_out,readB\_out,instruction,r3, output [2:0] rs1,rs2,rd );

wire zero;

wire [3:0] opcode, alu\_sel;

wire [2:0] funct3;

wire [15:0] ALU\_out, imm\_out, outmux, read\_data;

Ins\_Mem ic1 (.address(address),.opcode(opcode),.rd(rd),.rs1(rs1),.rs2(rs2),.funct3(funct3),.instruction(instruction));

reg\_file ic2 (.reg\_wrt(reg\_wrt),.rs1(rs1),.rs2(rs2),.rd(rd),.readA\_out(readA\_out),.readB\_out(readB\_out),.data(data\_reg),.r3(r3),.clk(clk));

MUX\_alu\_2\_1 ic3 (.B(readB\_out),.imm(imm\_out),.alu\_src(alu\_src),.outmux(outmux));

ALU ic4 (.A(readA\_out),.B(outmux),.alu\_op(alu\_sel),.ALU\_out(ALU\_out),.zero(zero));

ALU\_control ic5 (.instr(instruction),.alu\_op(alu\_op),.ALU\_control(alu\_sel));

Imm\_gen ic6 (.instruction(instruction),.imm\_type(immtype),.imm\_out(imm\_out));

data\_mem ic7 (.mem\_write\_en(mem\_write\_en),.clk(clk),.mem\_read\_en(mem\_read\_en),.addr(ALU\_out),.write\_data(readB\_out),.read\_data(read\_data));

MUX\_alu\_2\_1 ic8 (.B(read\_data),.imm(ALU\_out),.alu\_src(memtoreg),.outmux(data\_reg));

**endmodule**

**SIMULATION**

**1)**

**A diagram of a computer

AI-generated content may be incorrect.**

Here is my schematic of all things that I made, I will observe the outputs like instruction, rd, rs1, rs2,… This is action for checking whether my circuit work properly or not. Here is my testbench for adding two registers x4 and x5 with intial values 8 and 10, respectively and then storing its value into x3 with initial value is 6.

module HA\_tb();

reg reg\_wrt, mem\_write\_en, mem\_read\_en, memtoreg, alu\_src;

reg [2:0] immtype;

reg [1:0] alu\_op;

reg [11:0] address;

reg clk;

wire [15:0] data\_reg;

wire [15:0] instruction;

wire [2:0] rs1;

wire [2:0] rs2;

wire [2:0] rd;

wire [15:0] readA\_out;

wire [15:0] readB\_out;

wire [15:0] r3;

// G?i module chia

TOP uut (

.clk(clk),

.reg\_wrt(reg\_wrt),

.alu\_src(alu\_src),

.alu\_op(alu\_op),

.mem\_write\_en(mem\_write\_en),

.mem\_read\_en(mem\_read\_en),

.memtoreg(memtoreg),

.immtype(immtype),

.address(address),

.data\_reg(data\_reg),

.instruction(instruction),

.rs1(rs1),

.rs2(rs2),

.rd(rd),

.r3(r3),

.readA\_out(readA\_out),

.readB\_out(readB\_out)

);

initial begin

clk = 0;

forever #5 clk = ~clk;

end

initial begin

address = 8'b0000\_0000\_0000; // dia chi PC = 0

reg\_wrt = 0; // khong cho phep ghi vao reg\_file

alu\_src = 0; // chon dau vao ALU la gia tri tu reg\_file

alu\_op = 2'b00; // tin hieu tu Unit Control, 2'b00 tuong ung voi phep cong

mem\_write\_en = 0; // khong cho phep ghi vao data memory

immtype = 0; // khong dung immediate

mem\_read\_en = 0; // khong cho phep doc du lieu tu data memory

memtoreg = 1; // du lieu tu ALU se luu vao reg\_file thay vi data memory

#20;

reg\_wrt = 1; // cho phep du lieu tu ALU ghi vao reg\_file

// #5;

// reg\_wrt = 0; // khong cho phep ghi vao reg\_file

// mem\_write\_en = 1; //

end

endmodule

A screenshot of a computer

AI-generated content may be incorrect.

We can see that when address = 0, it will point to the first instruction inside instruction memory then we can see that it equals 0000\_100\_101\_011\_000. With 0000: opcode for adding, 100: address of x4 register, 101: address of x5 register, 011: address of x3 register, 000: funct3 for opcode’s complementary function. When it is decoded into 3 parts like that, addresses of registers will feed into reg\_file to take values of those registers. As a result, we can see readA\_out and readB\_out, these are value of register x4 and x5.

Then these values are calculated in ALU then output data\_reg[15:0] – output of ALU which equals to 18. This process works properly, then when reg\_wrt signal is activated, waiting for rising edge of clk, r3[15:0] which is the result register, it will store the value of the output of ALU or data memory, it will store the value of the output ALU, then it equals to 18. This process works perfectly.

**2)**

Here, I will test I-type, load value from data memory to register file and then subtract two registers. With the summary as below:

Load Mem[0] 🡪 x4 where Mem[0] = 20

Load Mem[1] 🡪 x5 where Mem[1] = 6

Sub, x3 🡨 x4 - x5

Testbench:

module HA\_tb();

reg reg\_wrt, mem\_write\_en, mem\_read\_en, memtoreg, alu\_src;

reg [2:0] immtype;

reg [1:0] alu\_op;

reg [11:0] address;

reg clk;

wire [15:0] data\_reg;

wire [15:0] instruction;

wire [2:0] rs1;

wire [2:0] rs2;

wire [2:0] rd;

wire [15:0] readA\_out;

wire [15:0] readB\_out;

wire [15:0] r3;

// G?i module chia

TOP uut (

.clk(clk),

.reg\_wrt(reg\_wrt),

.alu\_src(alu\_src),

.alu\_op(alu\_op),

.mem\_write\_en(mem\_write\_en),

.mem\_read\_en(mem\_read\_en),

.memtoreg(memtoreg),

.immtype(immtype),

.address(address),

.data\_reg(data\_reg),

.instruction(instruction),

.rs1(rs1),

.rs2(rs2),

.rd(rd),

.r3(r3),

.readA\_out(readA\_out),

.readB\_out(readB\_out)

);

initial begin

clk = 0;

forever #5 clk = ~clk;

end

initial begin

// load mem[addr + 0] -> x4

address = 8'b0000\_0000\_0000; // dia chi PC = 0

reg\_wrt = 1; // cho phep ghi vao reg\_file

alu\_src = 1; // chon dau vao ALU la gia tri tu imm\_gen

alu\_op = 2'b00; // tin hieu tu Unit Control, 2'b00 tuong ung voi phep cong

mem\_write\_en = 0; // khong cho phep ghi vao data memory

immtype = 0;

mem\_read\_en = 1; // cho phep doc du lieu tu data memory

memtoreg = 0; // du lieu se duoc lay tu data memory

#10;

// load mem[addr + 1] -> x5

address = 8'b0000\_0000\_0010;

#10;

//Sub, x3 <- x4 -x5

address = 8'b0000\_0000\_0100;

alu\_src = 0;

memtoreg = 1;

#10;

address = 8'b0000\_0000\_0110;

end

endmodule

A screenshot of a computer

AI-generated content may be incorrect.