# 数字逻辑与处理器基础 单周期处理器大作业

姓名：陈彦旭

学号：2022010597 班级：无24

## 实验目的

1. 掌握 MIPS 单周期处理器的控制通路和数据通路的设计原理和 RTL 实现方法；
2. 利用 Vivado 对处理器进行时序和面积分析，掌握评估处理器性能的关键指标；
3. 添加简易神经网络运算单元，掌握从算法映射到硬件的加速原理；

## Part1： MIPS 单周期 CPU 设计

（a）根据对各个控制信号的理解，完成 MIPS 指令集子集与控制信号的真值表（如下表所示，填 0、1、 2、 x 等），并根据填写的真值表完成 `single-cycle` 文件夹中控制器模块 `Control.v` 的 Verilog 代码实现。

控制信号解读：

PCSrc[1:0]：00—分支或顺序指令，01—跳转指令，else—jr指令

Branch：是否为分支指令， Branch & Zero 有效时为分支指令

RegWrite：寄存器堆写入有效

RegDest[1:0]：写入寄存器选择，00—rt，01—rd，else—$ra

MemRead：内存读取有效

MemWrite：内存写入有效

MemtoReg[1:0]：写回寄存器堆的来源，00—ALUOut，01—内存读取结果，else—PC+4（jal指令）

ALUSrc1：ALU 的第一个操作数来源，0—寄存器堆第一个读取端口 $rs ，1—Instruction[10:6]（shamt）

ALUSrc2：ALU 的第二个操作数来源，0—寄存器堆第二个读取端口 $rt ，1—立即数

ExtOp：立即数扩展，1—符号扩展，0—无符号扩展

LuOp：加载立即数到高位，只有 lui 用到。1—lui，0—立即数扩展结果

对于 lui 指令：将立即数加载到寄存器的高16位，R[rt]={imm, 16'b0}

控制信号真值表：

| | PCSrc[1:0] | Branch | RegWrite | RegDest[1:0] | MemRead | MemWrite | MemtoReg[1:0] | ALUSrc1 | ALUSrc2 | ExtOp | LuOp |
|---|---|---|---|---|---|---|---|---|---|---|---|
| lw | 00 | 0 | 1 | 00 | 1 | 0 | 01 | 0 | 1 | 1 | 0 |
| sw | 00 | 0 | 0 | x | x | 1 | x | 0 | 1 | 1 | 0 |
| lui | 00 | 0 | 1 | 00 | x | 0 | x | 0 | 1 | x | 1 |

| | PCSrc[1:0] | Branch | RegWrite | RegDest[1:0] | MemRead | MemWrite | MemtoReg[1:0] | ALUSrc1 | ALUSrc2 | ExtOp | LuOp |
|---|---|---|---|---|---|---|---|---|---|---|---|
| add | 00 | 0 | 1 | 01 | x | 0 | 00 | 0 | 0 | x | x |
| addu | 00 | 0 | 1 | 01 | x | 0 | 00 | 0 | 0 | x | x |
| sub | 00 | 0 | 1 | 01 | x | 0 | 00 | 0 | 0 | x | x |
| subu | 00 | 0 | 1 | 01 | x | 0 | 00 | 0 | 0 | x | x |
| addi | 00 | 0 | 1 | 00 | x | 0 | 00 | 0 | 1 | 1 | 0 |
| addiu | 00 | 0 | 1 | 00 | x | 0 | 00 | 0 | 1 | 1 | 0 |
| mul | 00 | 0 | 1 | 01 | x | 0 | 00 | 0 | 0 | x | x |
| and | 00 | 0 | 1 | 01 | x | 0 | 00 | 0 | 0 | x | x |
| or | 00 | 0 | 1 | 01 | x | 0 | 00 | 0 | 0 | x | x |
| xor | 00 | 0 | 1 | 01 | x | 0 | 00 | 0 | 0 | x | x |
| nor | 00 | 0 | 1 | 01 | x | 0 | 00 | 0 | 0 | x | x |
| andi | 00 | 0 | 1 | 00 | x | 0 | 00 | 0 | 1 | 0 | 0 |
| sll | 00 | 0 | 1 | 01 | x | 0 | 00 | 1 | 0 | x | x |
| srl | 00 | 0 | 1 | 01 | x | 0 | 00 | 1 | 0 | x | x |
| sra | 00 | 0 | 1 | 01 | x | 0 | 00 | 1 | 0 | x | x |
| slt | 00 | 0 | 1 | 01 | x | 0 | 00 | 0 | 0 | x | x |
| sltu | 00 | 0 | 1 | 01 | x | 0 | 00 | 0 | 0 | x | x |
| slti | 00 | 0 | 1 | 00 | x | 0 | 00 | 0 | 1 | 1 | 0 |
| sltiu | 00 | 0 | 1 | 00 | x | 0 | 00 | 0 | 1 | 1 | 0 |
| beq | 00 | 1 | 0 | x | x | 0 | x | 0 | x | 1 | x |
| j | 01 | 0 | 0 | x | x | 0 | x | x | x | x | x |
| jal | 01 | 0 | 1 | 10 | x | 0 | 10 | x | x | x | x |
| jr | 10 | 0 | 0 | x | x | 0 | x | x | x | x | x |

根据上面的真值表，在文件 `Control.v` 中添加控制信号代码：

```
// Control.v
module Control(
        input  [6 -1:0] OpCode   ,
        input  [6 -1:0] Funct    ,
        output [2 -1:0] PCSrc    ,
        output Branch            ,
        output RegWrite          ,
        output [2 -1:0] RegDst   ,
        output MemRead           ,
        output MemWrite          ,
        output [2 -1:0] MemtoReg ,
        output ALUSrc1           ,
        output ALUSrc2           ,
        output ExtOp             ,
        output LuOp              ,
        output [4 -1:0] ALUOp
    );


    // Your code below (for question 1)
    //
    //
    // PCSrc: 1-j,jal. 2-jr. 0-branch,else
    assign PCSrc[1:0] = (OpCode == 6'h02 || OpCode == 6'h03)? 1 : (OpCode == 6'h0 && Funct == 6'h08) ? 2 : 0;

```

```
    // branch: 1-beq. 0-else
    assign Branch = (OpCode == 6'h04) ? 1'b1 : 1'b0;

    // RegWrite: 0-sw,beq,j,jr. 1-else
    assign RegWrite = (OpCode == 6'h2b || OpCode == 6'h04 || OpCode == 6'h02 || (OpCode == 6'h0 && Funct == 6'h08)) ? 0 : 1;

    // RegDst: 0-lw,lui,addi,addiu,andi,slti,sltiu. 2-jal. 1-else
    assign RegDst = (OpCode == 6'h23 || OpCode == 6'h0f || OpCode == 6'h08 || OpCode == 6'h09 || OpCode == 6'h0c || OpCode == 6'h0a || OpCode == 6'h0b) ? 0 : (OpCode == 6'h03) ? 2 : 1;

    // MemRead: 1-lw. 0-else
    assign MemRead = (OpCode == 6'h23) ? 1 : 0;

    // MemWrite: 1-sw. 0-else
    assign MemWrite = (OpCode == 6'h2b) ? 1 : 0;

    // MemtoReg: 1-lw. 2-jal. 0-else
    assign MemtoReg = (OpCode == 6'h23) ? 1 : (OpCode == 6'h03 || (OpCode == 6'h00 && Funct == 6'h09)) ? 2 : 0;

    // ALUSrc1: 1-sll,srl,sra. 0-else
    assign ALUSrc1 = (OpCode == 6'h00 && (Funct == 6'h00 || Funct == 6'h02 || Funct == 6'h03)) ? 1 : 0;

    // ALUSrc2: 1-lw,sw,lui,addi,addiu,andi,slti,sltiu. 0-else
    assign ALUSrc2 = (OpCode == 6'h23 || OpCode == 6'h2b || OpCode == 6'h0f || OpCode == 6'h08 || OpCode == 6'h09 || OpCode == 6'h0c || OpCode == 6'h0a || OpCode == 6'h0b) ? 1 : 0;

    // ExtOp: 0-andi. 1-else
    assign ExtOp = (OpCode == 6'h0c) ? 0 : 1;

    // LuOp: 1-lui. 0-else
    assign LuOp = (OpCode == 6'h0f) ? 1 : 0;
    //
    //
    // Your code above (for question 1)

    // set ALUOp
    assign ALUOp[2:0] =
           (OpCode == 6'h00)? 3'b010:
           (OpCode == 6'h04)? 3'b001:
           (OpCode == 6'h0c)? 3'b100:
           (OpCode == 6'h0a || OpCode == 6'h0b)? 3'b101:
           (OpCode == 6'h1c && Funct == 6'h02)? 3'b110:
           3'b000; //mul

    assign ALUOp[3] = OpCode[0];

endmodule
```

（b）阅读 MIPS Assembly 1-1 中的指令代码。这段程序运行足够长时间后，寄存器$v0, $v1, $a0, $a1, $a2, $a3, $t0, $t2, $t3 （分别对应 2-11 号寄存器）中的值应该是多少？

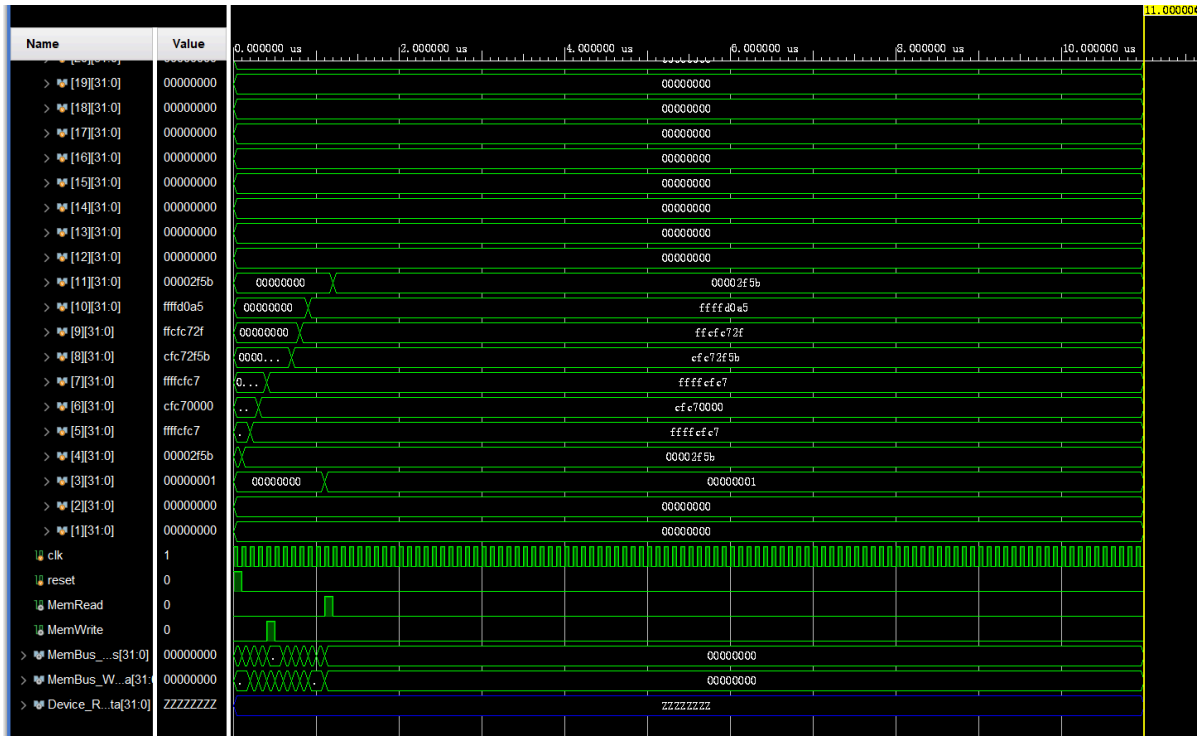
计算得到：

| 寄存 | 值 |
| --- | --- |
| $v0 | 0 |
| $v1 | 1 |
| $a0 | 12123 |
| $a1 | -12345 |
| $a2 | -809041920 |
| $a3 | -12345 |
| $t0 | -809029797 |
| $t1 | -3160273 |
| $t2 | -12123 |
| $t3 | 12123 |

（c）将 Inst-q1-1.txt 中的代码粘贴至 `InstructionMemory.v` 的相应位置，`Data-q1.txt` 中的代码粘贴至 `DataMemory.v` 的相应位置；以 `test_cpu.v` 为顶层文件进行仿真。请给出 b 问中所有寄存器的仿真波形图，验证计算结果和仿真结果是否一致，验证单周期处理器的正确性。

控制信号的仿真波形如下：


寄存器的波形如下：




根据波形，得到对应寄存器的值如下表：

| 寄存器编号 | 寄存器 | 值（十六进制） | 值（十进制） |
|---|---|---|---|
| 2 | $v0 | 0000 0000 | 0 |
| 3 | $v1 | 0000 0001 | 1 |
| 4 | $a0 | 0000 2F5B | 12123 |
| 5 | $a1 | FFFF CFC7 | -12345 |
| 6 | $a2 | CFC7 0000 | -809041920 |
| 7 | $a3 | FFFF CFC7 | -12345 |
| 8 | $t0 | CFC7 2F5B | -809029797 |
| 9 | $t1 | FFCF C72F | -3160273 |
| 10 | $t2 | FFFF D0A5 | -12123 |
| 11 | $t3 | 0000 2F5B | 12123 |

可见仿真结果与计算结果一致。

（d）阅读 MIPS Assembly 1-2 中的指令代码。该代码实现了乘累加运算，这段程序运行足够长时间后，寄存器 $a0, $a1, $s0 中的值应该是多少?

如下表：

| 寄存器 | 值 |
|---|---|
| $a0 | 32 |

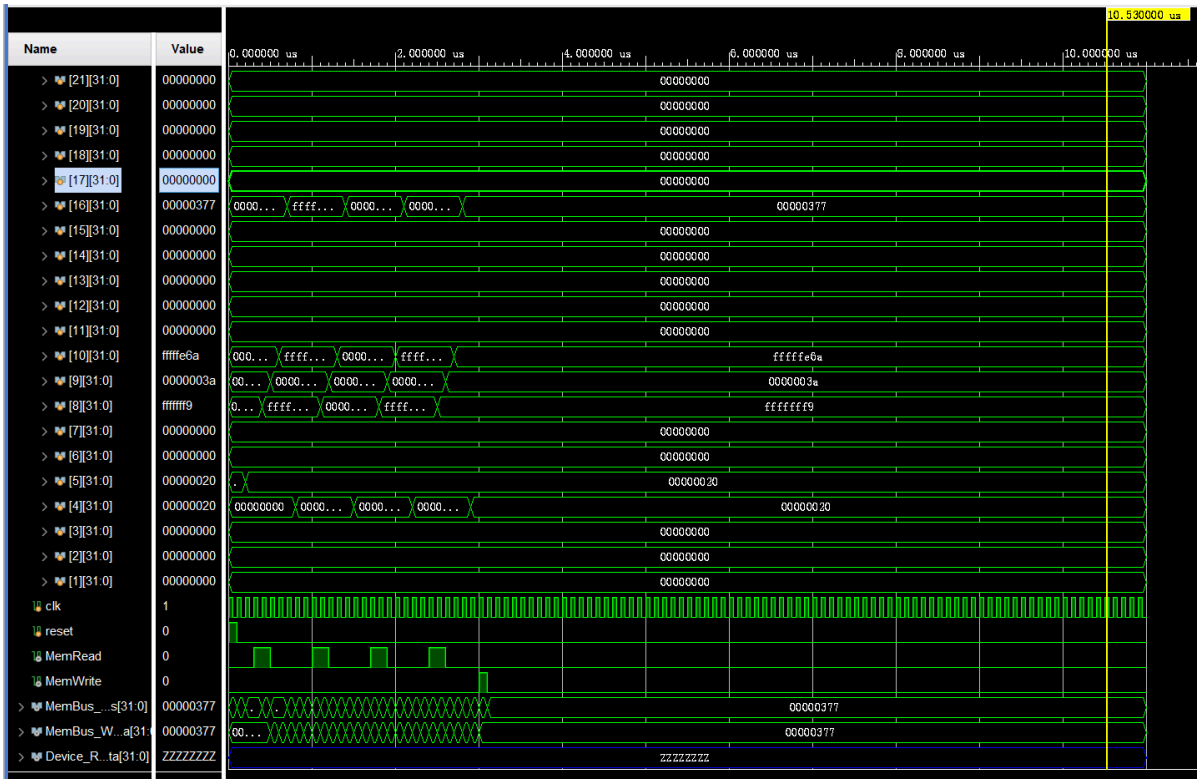
| 寄存器 | 值 |
| --- | --- |
| $a1 | 32 |
| $s0 | 887 |

其中 $s0 中的计算结果由下式得到：

$$-45\times3+40\times36+(-2)\times6+(-7)\times58=887 \tag{1}$$

（e）将 `Inst-q1-2.txt` 中的代码（对应 MIPS Assembly 1-2 ）粘贴至 `InstructionMemory.v` 的相应位置，`Data-q1.txt` 中的代码粘贴至 `DataMemory.v` 的相应位置；使用 Vivado 等软件进行仿真，顶层仿真模块为 `test_cpu.v`。请给出 d 问中所有寄存器的仿真波形图，验证乘法指令的功能正确性。

仿真结果如下：




| 寄存器编号 | 寄存器 | 值（十六进制） | 值（十进制） |
| --- | --- | --- | --- |
| 4 | $a0 | 0000 0020 | 32 |
| 5 | $a1 | 0000 0020 | 32 |
| 16 | $s0 | 0000 0377 | 887 |

可见仿真结果与计算结果一致。

（f）基于 Vivado 工具对处理器进行综合并开展静态时序分析。要求使用附件中的约束文件 (xdc_for_both.xdc)，FPGA 型号选择为：xc7a35tcsg324-1。根据 Vivado的资源和时序分析报告，分析说明 CPU 所可能达到的最高时钟频率和硬件资源开销。请在实验报告中附上综合分析资源和时序报告截图。

时序分析：

Design Runs | DRC | Methodology | Power | Timing

**Design Timing Summary**

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 973.317 ns | Worst Hold Slack (WHS): | 0.248 ns | Worst Pulse Width Slack (WPWS): | 499.500 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 16776 | Total Number of Endpoints: | 16776 | Total Number of Endpoints: | 8393 |

**All user specified timing constraints are met.**

最高时钟频率为：

$$f_{max} = \frac{1}{1000 - 973.317}\text{GHz} = 37.477\text{MHz} \tag{2}$$

逻辑资源占用：

Reports | Design Runs | DRC | Methodology | Power | Timing | Utilization

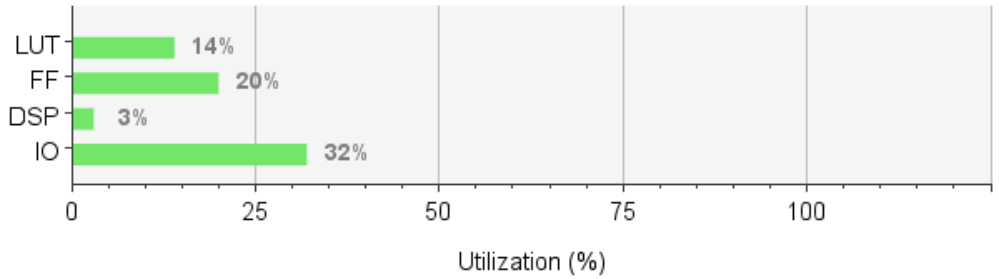
Hierarchy

| Name | Slice LUTs (20800) | Slice Registers (41600) | F7 Muxes (16300) | F8 Muxes (8150) | Slice (8150) | LUT as Logic (20800) | DSPs (90) | Bonded IOB (210) | BUFGCTRL (32) |
|---|---|---|---|---|---|---|---|---|---|
| CPU | 2935 | 8392 | 1088 | 512 | 3735 | 2935 | 3 | 68 | 1 |
| alu1 (ALU) | 431 | 0 | 0 | 0 | 192 | 431 | 3 | 0 | 0 |
| data_memory1 (DataMemory) | 2186 | 8192 | 1088 | 512 | 3637 | 2186 | 0 | 0 | 0 |
| register_file1 (RegisterFile) | 320 | 192 | 0 | 0 | 131 | 320 | 0 | 0 | 0 |

Reports | Design Runs | DRC | Methodology | Power | Timing | Utilization

**Summary**

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 2935 | 20800 | 14.11 |
| FF | 8392 | 41600 | 20.17 |
| DSP | 3 | 90 | 3.33 |
| IO | 68 | 210 | 32.38 |



# Part2：神经网络加速单元设计

（a）`Control.v` 和 `ALUControl.v` 文件中（如有需要，也可以在其他文件中进行相应修改）补充 mac 指令相关控制逻辑的 RTL 实现，并在 `ALU.v` 文件中实现 2个 8-bit 向量的 MAC 运算。

mac 指令的控制信号：

| PCSrc[1:0] | Branch | RegWrite | RegDest[1:0] | MemRead | MemWrite | MemtoReg[1:0] | ALUSrc1 | ALUSrc2 | ExtOp | LuOp |
|---|---|---|---|---|---|---|---|---|---|---|
| 00 | 0 | 1 | 01 | x | 0 | 00 | 0 | 0 | x | x |

可见 mac 指令与 add 指令控制信号相同，都可以归为普通的 R 型指令，因此在文件 `Control.v` 中无需作出修改。

在文件 `ALUControl.v` 中，增加 ALU 运算类型 `aluMAC` ：

```
// ALUControl.v
module ALUControl(
        input  [4 -1:0] ALUOp      ,
        input  [6 -1:0] Funct      ,
        output reg [5 -1:0] ALUCtl ,
        output Sign
    );

    // Add your code below (for question 2 and question 3)
    // funct number for different operation
    parameter aluAND = 5'b00000;
    parameter aluOR  = 5'b00001;
    parameter aluADD = 5'b00010;
    parameter aluSUB = 5'b00110;
    parameter aluSLT = 5'b00111;
    parameter aluNOR = 5'b01100;
    parameter aluXOR = 5'b01101;
    parameter aluSLL = 5'b10000;
    parameter aluSRL = 5'b11000;
    parameter aluSRA = 5'b11001;
    parameter aluMUL = 5'b11010; //mul
    // for question 2 and 3:
    parameter aluMAC = 5'b11011; //mac

    // Sign means whether the ALU treats the input as a signed number or an unsigned number
    assign Sign = (ALUOp[2:0] == 3'b010)? ~Funct[0]: ~ALUOp[3];

    // set aluFunct
    reg [4:0] aluFunct;
    always @(*)
    case (Funct)
        6'b00_0000:
            aluFunct <= aluSLL; //sll
        6'b00_0010:
            aluFunct <= aluSRL; //srl
        6'b00_0011:
            aluFunct <= aluSRA; //sra
        6'b10_0000:
```

```
            aluFunct <= aluADD; //add
        6'b10_0001:
            aluFunct <= aluADD; //addu
        6'b10_0010:
            aluFunct <= aluSUB; //sub
        6'b10_0011:
            aluFunct <= aluSUB; //subu
        6'b10_0100:
            aluFunct <= aluAND; //and
        6'b10_0101:
            aluFunct <= aluOR; //or
        6'b10_0110:
            aluFunct <= aluXOR; //xor
        6'b10_0111:
            aluFunct <= aluNOR; //nor
        6'b10_1010:
            aluFunct <= aluSLT; //slt
        6'b10_1011:
            aluFunct <= aluSLT; //sltu
        // for question2: mac
        6'b10_1101:
            aluFunct <= aluMAC; //mac:0x2e
        default:
            aluFunct <= aluADD;
    endcase

    // set ALUCtrl
    always @(*)
    case (ALUOp[2:0])
        3'b000:
            ALUCtl <= aluADD;
        3'b001:
            ALUCtl <= aluSUB; //beq
        3'b100:
            ALUCtl <= aluAND; //andi
        3'b101:
            ALUCtl <= aluSLT; //slti, sltiu
        3'b010:
            ALUCtl <= aluFunct;
        3'b110:
            ALUCtl <= aluMUL; //mul
        default:
            ALUCtl <= aluADD;
    endcase
    // Add your code above (for question 2 and question 3)

endmodule
```

在文件 `ALU.v` 中，`in1` 代表 `$rs`，`in2` 代表 `$rt`，因此修改代码如下：

```
// Hint: ALU port may be changed (for question 3)
module ALU(
        input [32 -1:0] in1      ,
```

```
        input [32 -1:0] in2      ,
        input [5 -1:0] ALUCtl    ,
        input Sign               ,
        output reg [32 -1:0] out ,
        output zero              ,
        // for question3:
        output reg [32 -1:0] out2
    );
    // zero means whether the output is zero or not
    assign zero = (out == 0);

    wire ss;
    assign ss = {in1[31], in2[31]};

    wire lt_31;
    assign lt_31 = (in1[30:0] < in2[30:0]);

    // lt_signed means whether (in1 < in2)
    wire lt_signed;
    assign lt_signed = (in1[31] ^ in2[31])?
           ((ss == 2'b01)? 0: 1): lt_31;

    // Add your code below (for question 2 and question 3)

    // different ALU operations
    always @(*)
    case (ALUCtl)
        5'b00000:
            out <= in1 & in2;
        5'b00001:
            out <= in1 | in2;
        5'b00010:
            out <= in1 + in2;
        5'b00110:
            out <= in1 - in2;
        5'b00111:
            out <= {31'h00000000, Sign? lt_signed: (in1 < in2)};
        5'b01100:
            out <= ~(in1 | in2);
        5'b01101:
            out <= in1 ^ in2;
        5'b10000:
            out <= (in2 << in1[4:0]);
        5'b11000:
            out <= (in2 >> in1[4:0]);
        5'b11001:
            out <= ({{32{in2[31]}}, in2} >> in1[4:0]);
        5'b11010:
            out <= in1 * in2; // mul
        // for question2: mac
        5'b11011:
            out <= $signed(in1[7:0]) * $signed(in2[7:0]) +
            $signed(in1[15:8]) * $signed(in2[15:8]) +
            $signed(in1[23:16]) * $signed(in2[23:16]) +
            $signed(in1[31:24]) * $signed(in2[31:24]);
        default:
```

```
            out <= 32'h00000000;
    endcase

    // Add your code above (for question 2 and question 3)

endmodule
```

使用 `$signed()` 使得 8 bit 向量变为有符号整数，作为补码参与运算。

（b）阅读 MIPS Assembly 2 中的指令代码。这段程序运行足够长时间后，寄存器 $s0, $s1, $s2, $s3 中的值应该是多少？

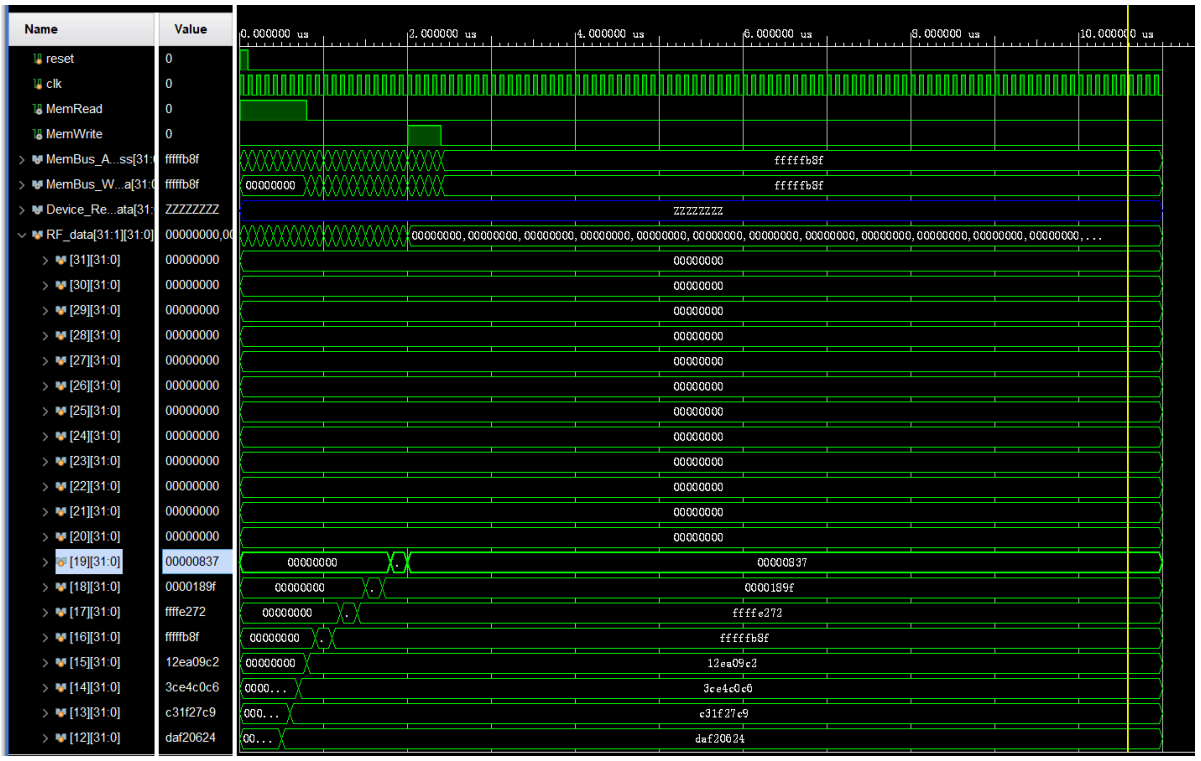
根据代码我们得到：

$$
\begin{aligned}
t0 &= 0xd328fef9 = 11010011\_00101000\_11111110\_11111001 = -45\_40\_2\_7 \\
t1 &= 0x0324063a = 00000011\_00100100\_00000110\_00111010 = 3\_36\_6\_58 \\
t2 &= 0x12da0c13 = 00010010\_11011010\_00001100\_00010011 = 18\_38\_12\_19 \\
t3 &= 0xde1015d6 = 11011110\_00010000\_00010101\_11010110 = -34\_16\_21\_42 \\
t4 &= 0xdaf20624 = 11011010\_11110010\_00000110\_00100100 = -38\_14\_6\_36 \\
t5 &= 0xc31f27c9 = 11000011\_00011111\_00100111\_11001001 = -61\_31\_39\_55 \\
t6 &= 0x3ce4c0c6 = 00111100\_11100100\_11000000\_11000110 = 60\_28\_64\_58 \\
t7 &= 0x12ea09c2 = 00010010\_11101010\_00001001\_11000010 = 18\_22\_9\_62 \\
s0 &= t0 \times t4 = -45 \times -38 + 40 \times -14 + -2 \times 6 + -7 \times 36 = 886 \\
a0 &= t1 \times t5 = 3 \times -61 + 36 \times 31 + 6 \times 39 + 58 \times -55 = -2023 \\
s0 &= t0 \times t4 + t1 \times t5 = -1137 \\
s1 &= t0 \times t6 = -45 \times 60 + 40 \times -28 + -2 \times -64 + -7 \times -58 = -3286 \\
a0 &= t1 \times t7 = 3 \times 18 + 36 \times -22 + 6 \times 9 + 58 \times -62 = -4280 \\
s1 &= t0 \times t6 + t1 \times t7 = -7566 \\
s2 &= t2 \times t4 = 18 \times -38 + -38 \times -14 + 12 \times 6 + 19 \times 36 = 604 \\
a0 &= t3 \times t5 = -34 \times -61 + 16 \times 31 + 21 \times 39 + -42 \times -55 = 5699 \\
s2 &= t2 \times t4 + t3 \times t5 = 6303 \\
s3 &= t2 \times t6 = 18 \times 60 + -38 \times -28 + 12 \times -64 + 19 \times -58 = 274 \\
a0 &= t3 \times t7 = -34 \times 18 + 16 \times -22 + 21 \times 9 + -42 \times -62 = 1829 \\
s3 &= t2 \times t6 + t3 \times t7 = 2103
\end{aligned}
$$

| 寄存器 | 值 |
| --- | --- |
| $s0 | -1137 |
| $s1 | -7566 |
| $s2 | 6303 |
| $s3 | 2103 |

（c）将 Inst-q2.txt 的代码粘贴至 `InstructionMemory.v` 的相应位置，将 `Data-q2-q3.txt` 的代码粘贴至 `DataMemory.v` 的相应位置，以 `test_cpu.v` 为顶层模块进行仿真。请给出 b 问中所有寄存器的仿真波形图，验证 b 问中计算结果是否与仿真结果一致，验证 mac 指令的功能正确性。

仿真结果如下：




可得：

| 寄存器编号 | 寄存器 | 值（十六进制） | 值（十进制） |
|---|---|---|---|
| 16 | $s0 | FFFF FB8F | -1137 |
| 17 | $s1 | FFFF E272 | -7566 |
| 18 | $s2 | 0000 189F | 6303 |
| 19 | $s3 | 0000 0837 | 2103 |

可见，寄存器中存储的计算结果与仿真结果一致，验证了 mac 指令的正确性。

（d）请简要分析，实验内容 2 引入的低位宽单周期 mac 指令与实验内容 1 通过组合精简指令实现乘累加运算这两种实现方式的区别与特点。

实验1中相乘累加的操作，是直接对寄存器中存储的数进行相乘，因此操作数的位宽是32，数字范围是。而且实验1是对“乘累加”的算法级实现，拆解为一个个简单的指令，并通过循环和访存实现，速度较慢。

实验2中 mac 指令所操作的两个寄存器，将32位划分为4个区域，因此没个寄存器各自存储了4个8 bit 数，范围只有。但是 mac 指令是对乘累加的指令级实现，通过增加 ALU 的计算复杂度完成计算，运行速度快，性能大大提升。

# Part3：激活函数计算

（a）在实验内容 2 的基础上添加 relu 指令数据通路与控制逻辑的 RTL 实现。（提示：该指令 1 个周期内需要从 $rs, $rd 读取数据，再写入 $rs 和$rd，因此需要在寄存器堆中增加一个写端口和写使能信号，并且可能需要修改 ALU 的输出端口）。请在实验报告中简要写出你的设计思路，并粘贴关键代码。

relu 指令的功能：

$$relu \quad rd \quad rs: \quad rd = max(rd, 0) \quad rs = max(rs, 0) \tag{3}$$

根据 relu 指令，寄存器堆原有的 `WriteRegister` 控制信号 `RegDest=2'01`，即为 rd，写入的数据 `WriteData=max{rd, 0}` 由 ALU 计算得到。现在增加一个能够写入 `$rs` 的端口 `WriteRegister2`，添加写使能控制信号 `RegWrite2`，且有 `WriteRegister2=Instruction[25:21]`，`WriteData2=max{rs, 0}`。要从寄存器堆的第二个读取端口 `Read_register2` 读取到寄存器 rd，可以根据 `RegWrite2` 判断读取 rt 还是 rd，也即增加：`assign Read_register2 = (RegWrite2 == 1)? Instruction[15:11] : Instruction[20:16];`。

relu 指令的控制信号：

| PCSrc[1:0] | Branch | RegWrite | RegWrite2 | RegDest[1:0] | MemRead | MemWrite | MemtoReg[1:0] | ALUSrc1 | ALUSrc2 | ExtOp | LuOp |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | 0 | 1 | 1 | 01 | x | 0 | 00 | 0 | 0 | x | x |

除了增加一个寄存器堆写使能信号 `RegWrite2` 以外，其他控制信号也都与 add 指令相同，因此文件 `Control.v` 文件无需进行修改。

为寄存器堆中增加一个写端口和写使能信号，在文件 `RegisterFile.v` 中添加三个信号：新的写入寄存器编号 `Write_register2`，新的寄存器写使能控制信号 `RegWrite2`，新的写入寄存器数据 `Write_data2`。

```
// Hint: Add a new write port & write-enable signal for the registerfile (for question 3)
module RegisterFile(
        input  reset                  ,
        input  clk                    ,
        input  RegWrite               ,
        input  [5 -1:0]  Read_register1 ,
        input  [5 -1:0]  Read_register2 ,
        input  [5 -1:0]  Write_register ,
        input  [32 -1:0] Write_data     ,
        output [32 -1:0] Read_data1     ,
        output [32 -1:0] Read_data2     ,
        // question3: Add a write port and write enable signal
        input  [5 -1:0]  Write_register2,
        input  RegWrite2,
        input  [32 -1:0] Write_data2
    );


    // RF_data is an array of 32 32-bit registers
    // here RF_data[0] is not defined because RF_data[0] identically equal to 0
    reg [31:0] RF_data[31:1];

    // read data from RF_data as Read_data1 and Read_data2
    assign Read_data1 = (Read_register1 == 5'b00000)? 32'h00000000: RF_data[Read_register1];
    assign Read_data2 = (Read_register2 == 5'b00000)? 32'h00000000: RF_data[Read_register2];

    integer i;
    // write Wrtie_data to RF_data at clock posedge
    always @(posedge reset or posedge clk)
        if (reset)
```

```
            for (i = 1; i < 32; i = i + 1)
                RF_data[i] <= 32'h00000000;
        else begin
            if (RegWrite && (Write_register != 5'b00000))
                RF_data[Write_register] <= Write_data;
            if (RegWrite2 && (Write_register2 != 5'b00000))
                RF_data[Write_register2] <= Write_data2;
        end
endmodule
```

文件 `Control.v` 中添加控制信号 `RegWrite2`：

```

module Control(
        input  [6 -1:0] OpCode   ,
        input  [6 -1:0] Funct    ,
        output [2 -1:0] PCSrc    ,
        output Branch            ,
        output RegWrite          ,
        output [2 -1:0] RegDst   ,
        output MemRead           ,
        output MemWrite          ,
        output [2 -1:0] MemtoReg ,
        output ALUSrc1           ,
        output ALUSrc2           ,
        output ExtOp             ,
        output LuOp              ,
        output [4 -1:0] ALUOp    ,
        // for question3:
        output RegWrite2

    );

    // Add your code below (for question 2 and question 3)

    // Your code below (for question 1)
    //
    //
    // PCSrc: 1-j,jal. 2-jr. 0-branch,else
    assign PCSrc[1:0] = (OpCode == 6'h02 || OpCode == 6'h03)? 1 : (OpCode == 6'h0 && Funct == 6'h08) ? 2 : 0;

    // branch: 1-beq. 0-else
    assign Branch = (OpCode == 6'h04) ? 1'b1 : 1'b0;

    // RegWrite: 0-sw,beq,j,jr. 1-else
    assign RegWrite = (OpCode == 6'h2b || OpCode == 6'h04 || OpCode == 6'h02 || (OpCode == 6'h0 && Funct == 6'h08)) ? 0 : 1;

    // RegDst: 0-lw,lui,addi,addiu,andi,slti,sltiu. 2-jal. 1-else
    assign RegDst = (OpCode == 6'h23 || OpCode == 6'h0f || OpCode == 6'h08 || OpCode == 6'h09 || OpCode == 6'h0c || OpCode == 6'h0a || OpCode == 6'h0b) ? 0 : (OpCode == 6'h03) ? 2 : 1;
```

```
38
39      // MemRead: 1-lw. 0-else
40      assign MemRead = (OpCode == 6'h23) ? 1 : 0;
41
42      // MemWrite: 1-sw. 0-else
43      assign MemWrite = (OpCode == 6'h2b) ? 1 : 0;
44
45      // MemtoReg: 1-lw. 2-jal. 0-else
46      assign MemtoReg = (OpCode == 6'h23) ? 1 : (OpCode == 6'h03 || (OpCode == 6'h00 && Funct == 6'h09)) ? 2 : 0;
47
48      // ALUSrc1: 1-sll,srl,sra. 0-else
49      assign ALUSrc1 = (OpCode == 6'h00 && (Funct == 6'h00 || Funct == 6'h02 || Funct == 6'h03)) ? 1 : 0;
50
51      // ALUSrc2: 1-lw,sw,lui,addi,addiu,andi,slti,sltiu. 0-else
52      assign ALUSrc2 = (OpCode == 6'h23 || OpCode == 6'h2b || OpCode == 6'h0f || OpCode == 6'h08 || OpCode == 6'h09 || OpCode == 6'h0c || OpCode == 6'h0a || OpCode == 6'h0b) ? 1 : 0;
53
54      // ExtOp: 0-andi. 1-else
55      assign ExtOp = (OpCode == 6'h0c) ? 0 : 1;
56
57      // LuOp: 1-lui. 0-else
58      assign LuOp = (OpCode == 6'h0f) ? 1 : 0;
59      //
60      //
61      // Your code above (for question 1)
62
63      // set ALUOp
64      assign ALUOp[2:0] =
65              (OpCode == 6'h00)? 3'b010:
66              (OpCode == 6'h04)? 3'b001: //beq
67              (OpCode == 6'h0c)? 3'b100: //andi
68              (OpCode == 6'h0a || OpCode == 6'h0b)? 3'b101: //slti, sltiu
69              (OpCode == 6'h1c && Funct == 6'h02)? 3'b110: //mul
70              3'b000;
71
72
73      assign ALUOp[3] = OpCode[0];
74
75      // question3:relu
76      assign RegWrite2 = (OpCode == 6'h0 && Funct == 6'h2e) ? 1 : 0;
77
78      // Add your code above (for question 2 and question 3)
79
80  endmodule
```

文件 `ALUControl.v` 中添加 ALU 运算类型 `aluRELU`：

```
1
2   module ALUControl(
3           input  [4 -1:0] ALUOp      ,
```

```
        input  [6 -1:0] Funct      ,
        output reg [5 -1:0] ALUCtl ,
        output Sign
    );

    // Add your code below (for question 2 and question 3)
    // funct number for different operation
    parameter aluAND = 5'b00000;
    parameter aluOR  = 5'b00001;
    parameter aluADD = 5'b00010;
    parameter aluSUB = 5'b00110;
    parameter aluSLT = 5'b00111;
    parameter aluNOR = 5'b01100;
    parameter aluXOR = 5'b01101;
    parameter aluSLL = 5'b10000;
    parameter aluSRL = 5'b11000;
    parameter aluSRA = 5'b11001;
    parameter aluMUL = 5'b11010; //mul
    // for question 2 and 3:
    parameter aluMAC = 5'b11011; //mac
    parameter aluRELU = 5'b11100; //relu

    // Sign means whether the ALU treats the input as a signed number or an unsigned number
    assign Sign = (ALUOp[2:0] == 3'b010)? ~Funct[0]: ~ALUOp[3];

    // set aluFunct
    reg [4:0] aluFunct;
    always @(*)
    case (Funct)
        6'b00_0000:
            aluFunct <= aluSLL; //sll
        6'b00_0010:
            aluFunct <= aluSRL; //srl
        6'b00_0011:
            aluFunct <= aluSRA; //sra
        6'b10_0000:
            aluFunct <= aluADD; //add
        6'b10_0001:
            aluFunct <= aluADD; //addu
        6'b10_0010:
            aluFunct <= aluSUB; //sub
        6'b10_0011:
            aluFunct <= aluSUB; //subu
        6'b10_0100:
            aluFunct <= aluAND; //and
        6'b10_0101:
            aluFunct <= aluOR; //or
        6'b10_0110:
            aluFunct <= aluXOR; //xor
        6'b10_0111:
            aluFunct <= aluNOR; //nor
        6'b10_1010:
            aluFunct <= aluSLT; //slt
        6'b10_1011:
            aluFunct <= aluSLT; //sltu
```

```
        // for question2: mac
        6'b10_1101:
            aluFunct <= aluMAC; //mac:0x2e
        // for question3: relu
        6'b10_1110:
            aluFunct <= aluRELU; //relu:0x2d
        default:
            aluFunct <= aluADD;
    endcase

    // set ALUCtrl
    always @(*)
    case (ALUOp[2:0])
        3'b000:
            ALUCtl <= aluADD;
        3'b001:
            ALUCtl <= aluSUB; //beq
        3'b100:
            ALUCtl <= aluAND; //andi
        3'b101:
            ALUCtl <= aluSLT; //slti, sltiu
        3'b010:
            ALUCtl <= aluFunct;
        3'b110:
            ALUCtl <= aluMUL; //mul
        default:
            ALUCtl <= aluADD;
    endcase
    // Add your code above (for question 2 and question 3)

endmodule
```

在文件 `ALU.v` 中增加一个 ALU 输出端口 `out2`：

```
// Hint: ALU port may be changed (for question 3)
module ALU(
        input [32 -1:0] in1      ,
        input [32 -1:0] in2      ,
        input [5 -1:0] ALUCtl    ,
        input Sign               ,
        output reg [32 -1:0] out ,
        output zero              ,
        // for question3:
        output reg [32 -1:0] out2
    );
    // zero means whether the output is zero or not
    assign zero = (out == 0);

    wire ss;
    assign ss = {in1[31], in2[31]};

    wire lt_31;
    assign lt_31 = (in1[30:0] < in2[30:0]);
```

```
20
21      // lt_signed means whether (in1 < in2)
22      wire lt_signed;
23      assign lt_signed = (in1[31] ^ in2[31])?
24             ((ss == 2'b01)? 0: 1): lt_31;
25
26      // Add your code below (for question 2 and question 3)
27
28      // different ALU operations
29      always @(*)
30      case (ALUCtl)
31          5'b00000:
32              out <= in1 & in2;
33          5'b00001:
34              out <= in1 | in2;
35          5'b00010:
36              out <= in1 + in2;
37          5'b00110:
38              out <= in1 - in2;
39          5'b00111:
40              out <= {31'h00000000, Sign? lt_signed: (in1 < in2)};
41          5'b01100:
42              out <= ~(in1 | in2);
43          5'b01101:
44              out <= in1 ^ in2;
45          5'b10000:
46              out <= (in2 << in1[4:0]);
47          5'b11000:
48              out <= (in2 >> in1[4:0]);
49          5'b11001:
50              out <= ({{32{in2[31]}}, in2} >> in1[4:0]);
51          5'b11010:
52              out <= in1 * in2; // mul
53          // for question2: mac
54          5'b11011:
55              out <= $signed(in1[7:0]) * $signed(in2[7:0]) +
56              $signed(in1[15:8]) * $signed(in2[15:8]) +
57              $signed(in1[23:16]) * $signed(in2[23:16]) +
58              $signed(in1[31:24]) * $signed(in2[31:24]);
59          // for question3: relu
60          5'b11100:
61              out <= (in2[31] == 0) ? in2 : 32'h00000000;
62          default:
63              out <= 32'h00000000;
64      endcase
65
66      // for question3: relu
67      always @(*)
68      case(ALUCtl)
69          5'b11100:
70              out2 <= (in1[31] == 0) ? in1 : 32'h00000000;
71          default:
72              out2 <= 32'h00000000;
73      endcase
74
75      // Add your code above (for question 2 and question 3)
```

```
76
77
78  endmodule
```

在文件 `CPU.v` 中修改代码如下：

```
1   module CPU(
2           input  reset                          ,
3           input  clk                            ,
4           output MemRead                        ,
5           output MemWrite                       ,
6           output [32 -1:0] MemBus_Address       ,
7           output [32 -1:0] MemBus_Write_Data    ,
8           input  [32 -1:0] Device_Read_Data
9       );
10
11      // PC register
12      reg  [31 :0] PC;
13      wire [31 :0] PC_next;
14      wire [31 :0] PC_plus_4;
15
16      always @(posedge reset or posedge clk)
17          if (reset)
18              PC <= 32'h00000000;
19          else
20              PC <= PC_next;
21
22      assign PC_plus_4 = PC + 32'd4;
23
24      // Instruction Memory
25      wire [31 :0] Instruction;
26      InstructionMemory instruction_memory1(
27                           .Address        (PC             ),
28                           .Instruction    (Instruction    )
29                       );
30
31      // Control
32      wire [2 -1:0] RegDst    ;
33      wire [2 -1:0] PCSrc     ;
34      wire          Branch    ;
35      wire          MemRead   ;
36      wire          MemWrite  ;
37      wire [2 -1:0] MemtoReg  ;
38      wire          ALUSrc1   ;
39      wire          ALUSrc2   ;
40      wire [4 -1:0] ALUOp     ;
41      wire          ExtOp     ;
42      wire          LuOp      ;
43      wire          RegWrite  ;
44      // for question3:
45      wire          RegWrite2 ;
46
47      Control control1(
```

```
                .OpCode     (Instruction[31:26] ),
                .Funct      (Instruction[5 : 0] ),
                .PCSrc      (PCSrc              ),
                .Branch     (Branch             ),
                .RegWrite   (RegWrite           ),
                .RegDst     (RegDst             ),
                .MemRead    (MemRead            ),
                .MemWrite   (MemWrite           ),
                .MemtoReg   (MemtoReg           ),
                .ALUSrc1    (ALUSrc1            ),
                .ALUSrc2    (ALUSrc2            ),
                .ExtOp      (ExtOp              ),
                .LuOp       (LuOp               ),
                .ALUOp      (ALUOp              ),
                .RegWrite2  (RegWrite2          )
            );

    // Register File
    wire [32 -1:0] Databus1;
    wire [32 -1:0] Databus2;
    wire [32 -1:0] Databus3;
    wire [5  -1:0] Write_register;
    // for question3:
    wire [5 -1:0] Read_register2;
    wire [5 -1:0] Write_register2;
    wire [32 -1:0] Write_Data2;
    //

    assign Read_register2 = (RegWrite2 == 1)? Instruction[15:11] :
Instruction[20:16];
    assign Write_register2 = (RegWrite2 == 1)? Instruction[25:21] :
Instruction[15:11];

    assign Write_register = (RegDst == 2'b00)? Instruction[20:16]: (RegDst
== 2'b01)? Instruction[15:11]: 5'b11111;

    // Add a new write port and write enable signal for RegisterFile (for
question-4)
    RegisterFile register_file1(
                    .reset          (reset              ),
                    .clk            (clk                ),
                    .RegWrite       (RegWrite           ),
                    .Read_register1 (Instruction[25:21] ),
                    .Read_register2 (Read_register2     ),
                    .Write_register (Write_register     ),
                    .Write_data     (Databus3           ),
                    .Read_data1     (Databus1           ),
                    .Read_data2     (Databus2           ),
                    // for question3:
                    .Write_register2(Write_register2    ),
                    .RegWrite2      (RegWrite2          ),
                    .Write_data2    (Write_Data2        )
                );

    // Extend
    wire [32 -1:0] Ext_out;
```

```
100     assign Ext_out = { ExtOp? {16{Instruction[15]}}: 16'h0000,
    Instruction[15:0]};
101
102     wire [32 -1:0] LU_out;
103     assign LU_out = LuOp? {Instruction[15:0], 16'h0000}: Ext_out;
104
105     // ALU Control
106     wire [5 -1:0] ALUCtl;
107     wire          Sign  ;
108
109     ALUControl alu_control1(
110                     .ALUOp  (ALUOp              ),
111                     .Funct  (Instruction[5:0]   ),
112                     .ALUCtl (ALUCtl             ),
113                     .Sign   (Sign               )
114                 );
115
116     // ALU
117     wire [32 -1:0] ALU_in1;
118     wire [32 -1:0] ALU_in2;
119     wire [32 -1:0] ALU_out;
120     wire           Zero   ;
121
122     assign ALU_in1 = ALUSrc1? {27'h00000, Instruction[10:6]}: Databus1;
123     assign ALU_in2 = ALUSrc2? LU_out: Databus2;
124
125     wire [32 -1:0] out2;
126
127     ALU alu1(
128             .in1    (ALU_in1    ),
129             .in2    (ALU_in2    ),
130             .ALUCtl (ALUCtl     ),
131             .Sign   (Sign       ),
132             .out    (ALU_out    ),
133             .zero   (Zero       ),
134             .out2   (out2       )
135         );
136
137     // Data Memory
138     wire [32 -1:0] Memory_Read_Data ;
139     wire           Memory_Read      ;
140     wire           Memory_Write     ;
141     wire [32 -1:0] MemBus_Read_Data ;
142
143     DataMemory data_memory1(
144                     .reset      (reset              ),
145                     .clk        (clk                ),
146                     .Address    (MemBus_Address     ),
147                     .Write_data (MemBus_Write_Data  ),
148                     .Read_data  (Memory_Read_Data   ),
149                     .MemRead    (Memory_Read        ),
150                     .MemWrite   (Memory_Write       )
151                 );
152
153     // --------  Memory Control --------
154     assign Memory_Read          = MemRead ;
```

```
155     assign Memory_Write          = MemWrite;
156     assign MemBus_Address        = ALU_out ;
157     assign MemBus_Write_Data     = Databus2;
158     assign MemBus_Read_Data      = Memory_Read_Data;
159     // --------  Memory Control --------
160 
161     // write back
162     assign Databus3 = (MemtoReg == 2'b00)? ALU_out: (MemtoReg == 2'b01)?
    MemBus_Read_Data: PC_plus_4;
163     // for question3:
164     assign Write_Data2 = out2;
165 
166     // PC jump and branch
167     wire [32 -1:0] Jump_target;
168     assign Jump_target = {PC_plus_4[31:28], Instruction[25:0], 2'b00};
169 
170     wire [32 -1:0] Branch_target;
171     assign Branch_target = (Branch & Zero)? PC_plus_4 + {LU_out[29:0],
    2'b00}: PC_plus_4;
172 
173     assign PC_next = (PCSrc == 2'b00)? Branch_target: (PCSrc == 2'b01)?
    Jump_target: Databus1;
174 
175 endmodule
```

（b）阅读 MIPS Assembly 3 中的指令代码。该代码实现了计算神经网络的矩阵乘法与激活函数运算。这段程序运行足够长时间后，寄存器 $s0, $s1, $s2, $s3 中的值应该是多少？

根据实验二我们得到：

$$\begin{aligned} \$s0 &= -1137 \\ \$s1 &= -7566 \end{aligned} \tag{4}$$

执行 relu 指令后，$s0, $s1 内的值变为0。

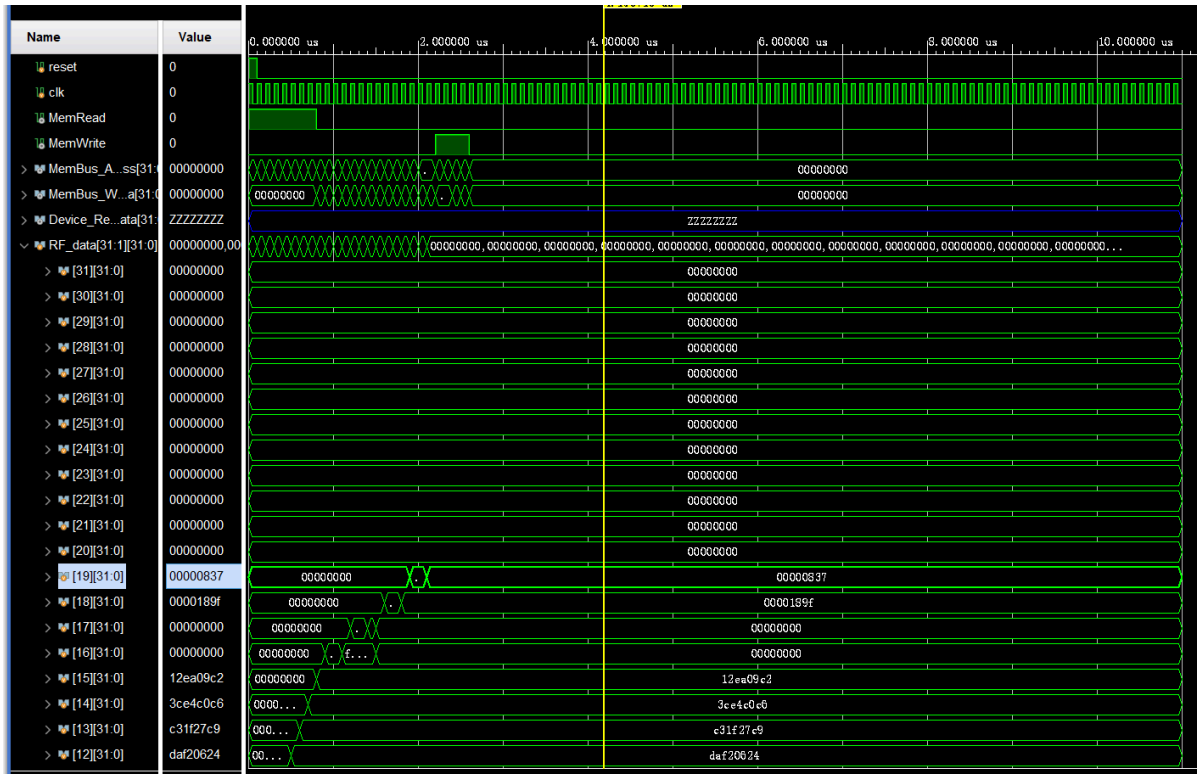
$$\begin{aligned} \$s2 &= 6303 \\ \$s3 &= 2103 \end{aligned} \tag{5}$$

执行 relu 指令后，$s2, $s3 内的值不变。因此得到下表：

| 寄存器 | 值 |
| --- | --- |
| $s0 | 0 |
| $s1 | 0 |
| $s2 | 6303 |
| $s3 | 2103 |

（c）Inst-q3.txt 的代码粘贴至 `InstructionMemory.v` 的相应位置，将 `Data-q2-q3.txt` 的代码粘贴至 `DataMemory.v` 的相应位置。以 `test_cpu.v` 为顶层模块进行仿真。请给出 b 问中所有寄存器的仿真波形图，验证 b 问中计算结果是否与仿真结果一致，验证 relu 指令的功能正确性。

仿真波形如下：




根据波形图可得：

| 寄存器编号 | 寄存器 | 值（十六进制） | 值（十进制） |
| --- | --- | --- | --- |
| 16 | $s0 | 0000 0000 | 0 |
| 17 | $s1 | 0000 0000 | 0 |
| 18 | $s2 | 0000 189F | 6303 |
| 19 | $s3 | 0000 0837 | 2103 |

可见，仿真结果与计算结果一致，验证了 relu 指令的正确性。

（d）根据 Vivado 的资源和时序分析报告，分析说明 CPU 所可能达到的最高时钟频率和硬件资源开销。请在实验报告中附上综合分析资源和时序报告截图。

时序分析：

**Design Timing Summary**

| Setup | | Hold | | Pulse Width | |
| --- | --- | --- | --- | --- | --- |
| Worst Negative Slack (WNS): | 949.841 ns | Worst Hold Slack (WHS): | 0.481 ns | Worst Pulse Width Slack (WPWS): | 499.500 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 18376 | Total Number of Endpoints: | 18376 | Total Number of Endpoints: | 9193 |

**All user specified timing constraints are met.**

计算出最高时钟频率为：

$$f_{max} = \frac{1}{1000 - 949.841}\text{GHz} = 19.937\text{MHz} \tag{6}$$

逻辑资源占用情况：

Reports | Design Runs | Power | Methodology | DRC | Timing | Utilization

Hierarchy

| Name | Slice LUTs (20800) | Slice Registers (41600) | F7 Muxes (16300) | F8 Muxes (8150) | Slice (8150) | LUT as Logic (20800) | Bonded IOB (210) | BUFGCTRL (32) |
|---|---|---|---|---|---|---|---|---|
| CPU | 4521 | 9192 | 445 | 125 | 3348 | 4521 | 68 | 1 |
| alu1 (ALU) | 134 | 0 | 2 | 0 | 97 | 134 | 0 | 0 |
| data_memory1 (DataMemory) | 2629 | 8192 | 282 | 124 | 2882 | 2629 | 0 | 0 |
| register_file1 (RegisterFile) | 1771 | 992 | 161 | 1 | 638 | 1771 | 0 | 0 |

**Summary**

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 4521 | 20800 | 21.74 |
| FF | 9192 | 41600 | 22.10 |
| IO | 68 | 210 | 32.38 |