西安交通大学
XI'AN JIAOTONG UNIVERSITY

《计算机组成与系统结构专题实验》

Experiment on Computer Organization and Architecture

计算机科学与技术学院 实验中心

2023 年 11 月

实验内容

实验六 CPU 综合设计

1）设计一个基于 MIPS 指令集的 CPU ，支持以下指令： {add, sub, addi, lw, sw, beq, j, nop}

2） CPU 需要包含寄存器组、 RAM 模块、 ALU 模块、指令译码模块。

3）该 CPU 能运行基本的汇编指令。 （ D~C+ ）

以下为可选内容：

4）实现多周期 CPU （ B-~B+ ）

5）实现以下高级功能之一（ A-~A+ ）

(1). 实现 5 级流水线 CPU

(2). 实现超标量

(3). 实现 4 路组相联缓存

可基于 RISC V 、 ARM 指令集实现。

如发现代码为抄袭代码，成绩一律按不及格处理。

实验内容

实验六 CPU 综合设计

- 本次实验在前五次实验基础上，对各模块进行集成，形成一个完整 CPU
- PC 指向当前指令在 IM 中地址（实验四）
- 从 IM 获取指令后，对其进行译码，生成相应控制信号（实验五）
- 根据控制信号， ALU 等模块产生相应结果（要求必须使用实验三完成的 ALU ，其中加法器为用基本门器件实现）
- 将结果写回相应寄存器（实验四）
- 执行完成后 PC 地址加 4 （可为 PC 寄存器专门配置一个加法器实现自增）

MIPS 指令解释

本次实验用到的指令解释及其操作码
R-Type 指令：

1. add rd, rs, rt：将 rs 和 rt 寄存器中的值相加，并将结果存储在 rd 寄存器中。
   操作码： 000000
   功能码： 20

2. sub rd, rs, rt：将 rs 寄存器中的值减去 rt 寄存器中的值，并将结果存储在 rd 寄存器中。
   操作码： 000000
   功能码： 22

实验内容

MIPS 指令操作码

I-Type 指令：

1. addi rt, rs, immediate： 将 rs 寄存器中的值与立即数相加，并将结果存储在 rt 寄存器中。

   操作码： 001000

2. lw rt, offset(rs)： 将地址为 rs + offset 的内存中的值加载到 rt 寄存器中。

   操作码： 100011

3. sw rt, offset(rs)： 将 rt 寄存器中的值存储到地址为 rs + offset 的内存中。

   操作码： 101011

4. beq rs, rt, offset： 如果 rs 和 rt 寄存器中的值相等，则跳转到当前指令地址加上 offset 的地址。

   操作码： 000100

MIPS 指令操作码

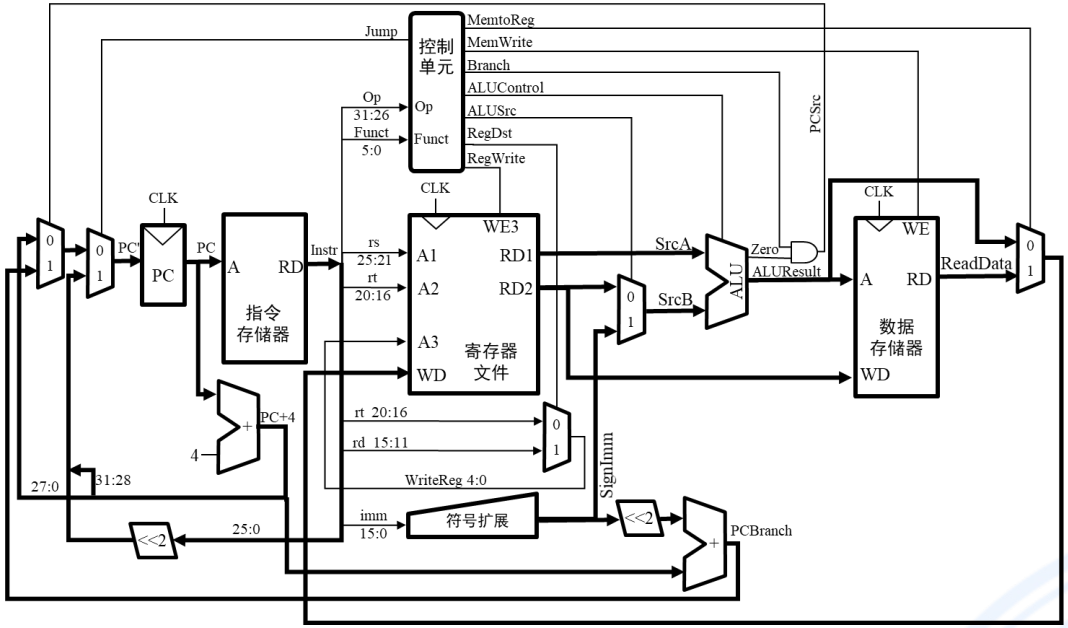
J-Type 指令：
j target：无条件跳转到目标地址。
操作码： 000010

NOP 指令：
nop：空指令，不执行任何操作。
操作码： 000000

ALU 操作码：
00000：加法
00001：减法
00010：逻辑与
00011：逻辑或
00100：逻辑非

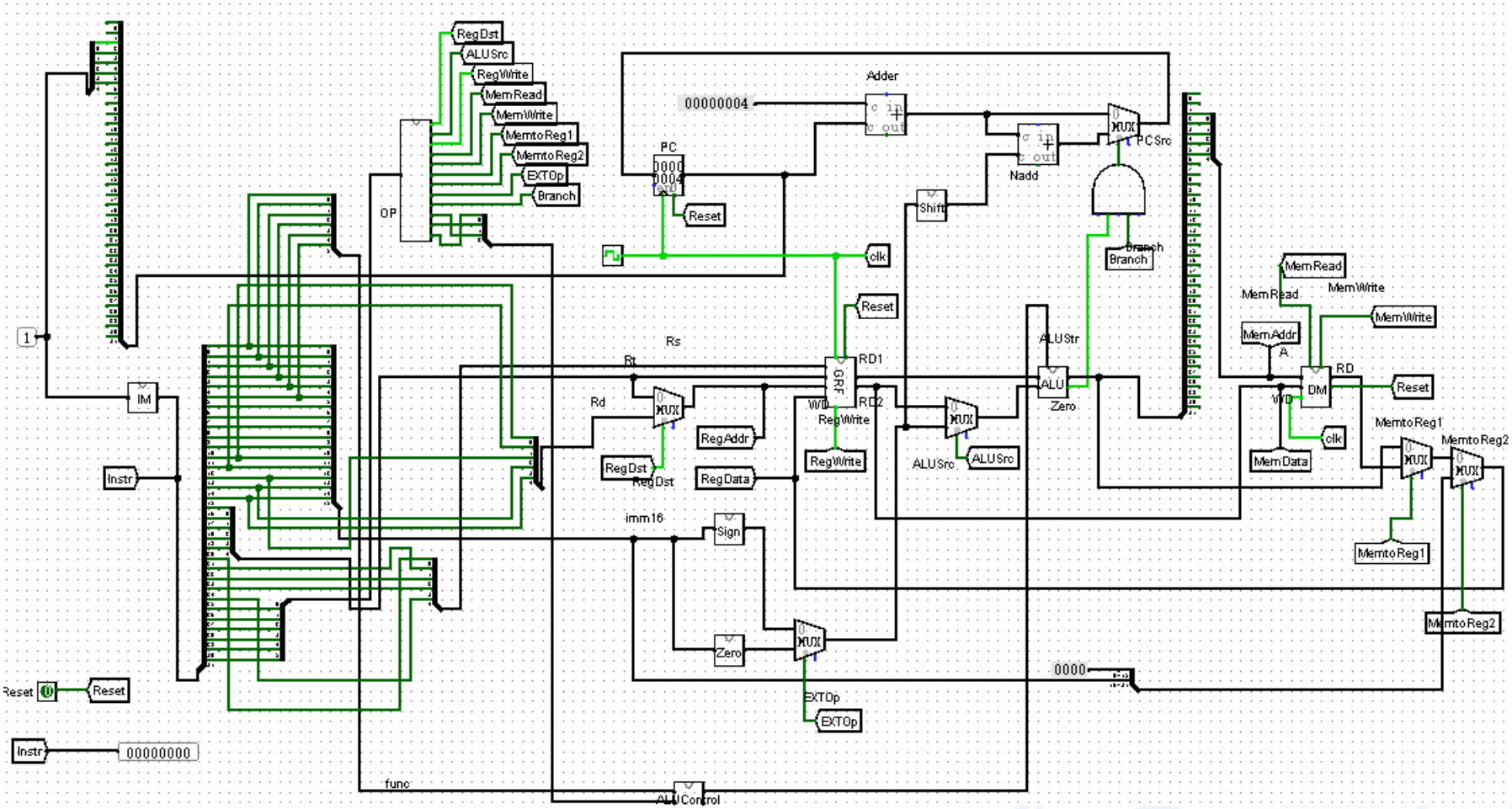
实验内容

单周期 CPU 原理



实验内容

单周期 CPU 实现



单周期 CPU 实现

单周期 CPU 代码实现框架如下：

```
module SimpleMIPSCPU(
    input wire clk,    // 时钟
    input wire rst,    // 复位
     input wire [31:0] instruction,   // 输入指令
    output reg [31:0] result  // 输出结果
);

// 寄存器文件
reg [31:0] registers [0:31];

// 控制信号
reg RegWrite, MemtoReg, MemWrite, ALUSrc, Branch, Jump, Zero;
reg [5:0] opcode;
reg [4:0] rs, rt, rd;

// 立即数
wire [31:0] immediate;

// ALU 结果
wire [31:0] alu_result;

// 内存
reg [31:0] memory [0:1023];
```

单周期 CPU 实现

```
// 运算单元
ALU alu (
    .A(registers[rs]),
    .B(ALUSrc ? immediate : registers[rt]),
    .Op(opcode),
    .Result(alu_result),
    .Zero(Zero)
);
```

```
// 主控制单元
ControlUnit ctrl (
    .opcode(opcode),
    .RegWrite(RegWrite),
    .MemtoReg(MemtoReg),
    .MemWrite(MemWrite),
    .ALUSrc(ALUSrc),
    .Branch(Branch),
    .Jump(Jump)
);
```

单周期 CPU 实现

```
// 数据存储单元
DataMemory dmem (
    .address(alu_result),
    .write_data(registers[rt]),
    .mem_read(MemWrite),
    .mem_write(MemWrite),
    .read_data(result)
);
```

```
// 寄存器写入
always @(posedge clk or posedge rst) begin
    if (rst) begin
        // 复位时将寄存器清零
        registers <= 32'b0;
    end else if (RegWrite) begin
        // 写入寄存器
          registers[rd] <= MemtoReg ? result :
alu_result;
    end
end
```

实验内容

单周期 CPU 实现

```
// 立即数生成
assign immediate = instruction[15:0];

// 控制信号生成
always @* begin
    // 从指令中提取字段
    opcode = instruction[31:26];
    rs = instruction[25:21];
    rt = instruction[20:16];
    rd = instruction[15:11];

    // 生成控制信号
        {RegWrite, MemtoReg, MemWrite,
ALUSrc, Branch, Jump} =
ctrl.control_signals;
end
```

```
// 同步复位
always @(posedge clk or posedge rst) begin
    if (rst) begin
        // 复位时初始化
        result <= 32'b0;
    end else begin
        // 输出结果
            result <= MemtoReg ? result :
alu_result;
    end
end

endmodule
```

实验内容

多周期 CPU 实现

多周期 CPU Verilog 框架（RF,DM 等参考之前实验实现）

```
module MultiCycleCPU (
    input clk,
    input reset
);
    // 定义状态
    parameter FETCH = 3'b000;
    parameter DECODE = 3'b001;
    parameter EXECUTE = 3'b010;
    parameter MEM_ACCESS = 3'b011;
    parameter WRITE_BACK = 3'b100;

    reg [2:0] state, next_state;

    // 内部信号
    reg [31:0] PC_in;
    wire [31:0] PC_out, Instruction;
    reg [31:0] IR; // 指令寄存器
    wire [31:0] ReadData1, ReadData2, ALUResult, MemReadData;
    reg [31:0] SignExtImm;
    reg [31:0] ALUInputB;
    reg [4:0] WriteReg; // 修改为 reg 类型
    wire [3:0] ALUControl;
    wire Zero;
```

实验内容

多周期 CPU 实现

```
// 控制信号
    wire RegWrite, RegDst, ALUSrc, Branch,
MemWrite, MemToReg, Jump, PCSrc;
    wire [1:0] ALUOp;

    // 初始化 PC_in
    initial begin
        PC_in = 0;
    end
```

```
// 实例化程序计数器
    ProgramCounter PC (
        .clk(clk),
        .reset(reset),
        .PC_in(PC_in),
        .PC_out(PC_out)
    );

    // 实例化指令存储器
    InstructionMemory IM (
        .Address(PC_out),
        .Instruction(Instruction)
    );
```

实验内容

多周期 CPU 实现

```
// 寄存器组
    RegisterFile RF(
        .clk(clk),
        .RegWrite(RegWrite),
        .ReadReg1(Instruction[25:21]),
        .ReadReg2(Instruction[20:16]),
        .WriteReg(WriteReg),
                        .WriteData(MemtoReg    ?
MemReadData : ALUResult),
        .ReadData1(ReadData1),
        .ReadData2(ReadData2)
    );
```

```
// 实例化数据存储器
    DataMemory DM (
        .clk(clk),
        .MemWrite(MemWrite),
        .MemRead(MemRead),
        .Address(ALUResult),
        .WriteData(ReadData2),
        .ReadData(MemReadData)
    );
```

实验内容

多周期 CPU 实现

```
// 控制单元
    ControlUnit CU(
        .clk(clk),
        .Opcode(Instruction[31:26]),
        .RegDst(RegDst),
        .ALUSrc(ALUSrc),
        .MemtoReg(MemtoReg),
        .RegWrite(RegWrite),
        .MemRead(MemRead),
        .MemWrite(MemWrite),
        .Branch(Branch),
        .ALUOp(ALUOp)
    );
```

```
// ALU 控制单元
    ALUControlUnit ALUCtrl(
        .ALUOp(ALUOp),
        .Funct(Instruction[5:0]),
        .ALUControl(ALUControl)
    );

    // 符号扩展逻辑
    always @(*) begin
        SignExtImm = {{16{IR[15]}}, IR[15:0]}; // 使用 IR
    end
```

实验内容

多周期 CPU 实现

```
// ALU 输入选择逻辑
    always @(*) begin
        if (ALUSrc)
            ALUInputB = SignExtImm;
        else
            ALUInputB = ReadData2;
    end
// ALU 操作
    ALU alu (
        .clk(clk),
        .A(ReadData1),
        .B(ALUInputB),
        .ALUControl(ALUControl),
        .Result(ALUResult),
        .Zero(Zero)
    );
```

```
// 状态机
    always @(posedge clk or posedge reset)
begin
        if (reset) begin
            state <= FETCH;
            IR <= 0;
            PC_in <= 0; // 重置时 PC 归零
        end else begin
            state <= next_state;
                // 控制信号设置
            if (state == FETCH) begin
                    IR <= Instruction; // 在 FETCH
状态下写入新的指令
                        PC_in <= PC_out + 4; // 在
FETCH 状态下更新 PC
            end
        end
    end
```

实验内容

多周期 CPU 实现

```
// 下一状态逻辑
  always @(*) begin
    case (state)
      FETCH: next_state = DECODE;
      DECODE: next_state = EXECUTE;
      EXECUTE: begin
        case (IR[31:26])
                    6'b100011, 6'b101011: next_state =
MEM_ACCESS; // LW, SW
          6'b000100: next_state = FETCH; // BEQ
            6'b001000: next_state = WRITE_BACK; //
ADDI
           6'b000000: next_state = WRITE_BACK; // R-
type
          default: next_state = FETCH;
        endcase
      end
MEM_ACCESS: begin
          if (IR[31:26] == 6'b100011) // LW
            next_state = WRITE_BACK;
          else
            next_state = FETCH; // SW
        end
        WRITE_BACK: next_state = FETCH;
        default: next_state = FETCH;
      endcase
    end

    // 写寄存器选择逻辑
    always @(*) begin
      if (RegDst)
        WriteReg = IR[15:11]; // R-type 指令写
入 rd
      else
        WriteReg = IR[20:16]; // I-type 指令写
入 rt
    end
endmodule
```

实验内容

多周期 CPU 实现 (CU 实现)

```
module ControlUnit(
    input clk,
    input [5:0] Opcode,
    output reg RegDst,
    output reg ALUSrc,
    output reg MemtoReg,
    output reg RegWrite,
    output reg MemRead,
    output reg MemWrite,
    output reg Branch,
    output reg [1:0] ALUOp
);
    always @(posedge clk) begin
        case (Opcode)
            6'b000000: begin // R 型指令
                RegDst   = 1;
                ALUSrc   = 0;
                MemtoReg = 0;
                RegWrite = 1;
                MemRead  = 0;
                MemWrite = 0;
                Branch   = 0;
                ALUOp    = 2'b10;
            end
            6'b100011: begin // LW
                RegDst   = 0;
                ALUSrc   = 1;
                MemtoReg = 1;
                RegWrite = 1;
                MemRead  = 1;
                MemWrite = 0;
                Branch   = 0;
                ALUOp    = 2'b00;
            end
            6'b101011: begin // SW
                RegDst   = 0; // 无意义
                ALUSrc   = 1;
                MemtoReg = 0; // 无意义
                RegWrite = 0;
                MemRead  = 0;
                MemWrite = 1;
                Branch   = 0;
                ALUOp    = 2'b00;
            end
            6'b000100: begin // BEQ
                RegDst   = 0; // 无意义
                ALUSrc   = 0;
                MemtoReg = 0; // 无意义
                RegWrite = 0;
                MemRead  = 0;
                MemWrite = 0;
                Branch   = 1;
                ALUOp    = 2'b01;
            end
            6'b001101: begin // ORI
                RegDst   = 0;
                ALUSrc   = 1;
                MemtoReg = 0;
                RegWrite = 1;
                MemRead  = 0;
                MemWrite = 0;
                Branch   = 0;
                ALUOp    = 2'b11;
            end
            6'b001111: begin // LUI
                RegDst   = 0;
                ALUSrc   = 1;
                MemtoReg = 0;
                RegWrite = 1;
                MemRead  = 0;
                MemWrite = 0;
                Branch   = 0;
                ALUOp    = 2'b00; // ALU 操作不重要
            end
            default: begin // NOP 或未知指令
                RegDst   = 0;
                ALUSrc   = 0;
                MemtoReg = 0;
                RegWrite = 0;
                MemRead  = 0;
                MemWrite = 0;
                Branch   = 0;
                ALUOp    = 2'b00;
            end
        endcase
    end
endmodule
```

实验内容

五级流水 CPU 实现

5 级流水 CPU 框架如下：

```
// Forwarding Unit
//  添加前推单元来解决数据冒险
// ...

// Branch Prediction Unit
//  添加分支预测单元来解决控制冒险
// ...

// Data Hazard Handling in Execute Stage (EX)
module ExecuteStage (
    // ...  其他输入输出
    input wire [31:0] alu_result,
    input wire [4:0] rs2,
    input wire [1:0] forward_ex,
    output wire [31:0] alu_in2
);
```

```
 //  数据冒险解决逻辑
      always @(posedge clk or posedge rst)
begin
       if (rst) begin
          alu_in2 <= 0;
       end else begin
          //  使用前推单元的信号进行前推
           alu_in2 <= (forward_ex == 2'b10) ?
alu_result :
                          (forward_ex == 2'b01) ?
alu_result :
                          (forward_ex == 2'b00) ?
rs2 : 0;
       end
    end
endmodule
```

实验内容

五级流水 CPU 实现

```
// Data Hazard Handling in Memory Stage (MEM)
module MemoryStage (
    // ... 其他输入输出
    input wire [31:0] alu_result,
    input wire [4:0] rs2,
    input wire [1:0] forward_mem,
    output wire [31:0] mem_data
);
 // 数据冒险解决逻辑
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            mem_data <= 0;
        end else begin
            // 使用前推单元的信号进行前推
            mem_data <= (forward_mem == 2'b10) ? alu_result :
                        (forward_mem == 2'b01) ? alu_result :
                        (forward_mem == 2'b00) ? rs2 : 0;
        end
    end
endmodule
```

```
// Control Hazard Handling in IF Stage (Instruction Fetch)
module InstructionFetchStage (
    // ... 其他输入输出
    input wire [31:0] pc,
    input wire branch,
    input wire branch_taken,
    output wire [31:0] next_pc
);
    // 分支预测的简单实现
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            next_pc <= 0;
        end else begin
            // 使用分支预测单元的信号进行分支预测
            next_pc <= branch ? (branch_taken ? target_address :
pc + 4) : pc + 4;
        end
    end
endmodule
```

实验内容　附：　MIPS 指令操作码

Arithmetic and Logical Instructions

Instruction	Opcode/Function	Syntax	Operation
add	100000	f $d, $s, $t	$d = $s + $t
addu	100001	f $d, $s, $t	$d = $s + $t
addi	001000	f $d, $s, i	$d = $s + SE(i)
addiu	001001	f $d, $s, i	$d = $s + SE(i)
and	100100	f $d, $s, $t	$d = $s & $t
andi	001100	f $d, $s, i	$t = $s & ZE(i)
div	011010	f $s, $t	lo = $s / $t; hi = $s % $t
divu	011011	f $s, $t	lo = $s / $t; hi = $s % $t
mult	011000	f $s, $t	hi:lo = $s * $t
multu	011001	f $s, $t	hi:lo = $s * $t
nor	100111	f $d, $s, $t	$d = ~($s | $t)
or	100101	f $d, $s, $t	$d = $s | $t
ori	001101	f $d, $s, i	$t = $s | ZE(i)
sll	000000	f $d, $t, a	$d = $t << a
sllv	000100	f $d, $t, $s	$d = $t << $s
sra	000011	f $d, $t, a	$d = $t >> a
srav	000111	f $d, $t, $s	$d = $t >> $s
srl	000010	f $d, $t, a	$d = $t >>> a
srlv	000110	f $d, $t, $s	$d = $t >>> $s
sub	100010	f $d, $s, $t	$d = $s - $t
subu	100011	f $d, $s, $t	$d = $s - $t
xor	100110	f $d, $s, $t	$d = $s ^ $t
xori	001110	f $d, $s, i	$d = $s ^ ZE(i)

附： MIPS 指令操作码

Constant-Manipulating Instructions

Instruction	Opcode/Function	Syntax	Operation
lhi	011001	o $t, immed32	HH ($t) = i
llo	011000	o $t, immed32	LH ($t) = i

Comparison Instructions

Instruction	Opcode/Function	Syntax	Operation
slt	101010	f $d, $s, $t	$d = ($s < $t)
sltu	101001	f $d, $s, $t	$d = ($s < $t)
slti	001010	f $d, $s, i	$t = ($s < SE(i))
sltiu	001001	f $d, $s, i	$t = ($s < SE(i))

附： MIPS 指令操作码

Branch Instructions

Instruction	Opcode/Function	Syntax	Operation
beq	000100	o $s, $t, label	if ($s == $t) pc += i << 2
bgtz	000111	o $s, label	if ($s > 0) pc += i << 2
blez	000110	o $s, label	if ($s <= 0) pc += i << 2
bne	000101	o $s, $t, label	if ($s != $t) pc += i << 2

Jump Instructions

Instruction	Opcode/Function	Syntax	Operation
j	000010	o label	pc += i << 2
jal	000011	o label	$31 = pc; pc += i << 2
jalr	001001	o labelR	$31 = pc; pc = $s
jr	001000	o labelR	pc = $s

附： MIPS 指令操作码

Load Instructions

Instruction	Opcode/Function	Syntax	Operation
lb	100000	o $t, i ($s)	$t = SE (MEM [$s + i]:1)
lbu	100100	o $t, i ($s)	$t = ZE (MEM [$s + i]:1)
lh	100001	o $t, i ($s)	$t = SE (MEM [$s + i]:2)
lhu	100101	o $t, i ($s)	$t = ZE (MEM [$s + i]:2)
lw	100011	o $t, i ($s)	$t = MEM [$s + i]:4

Store Instructions

Instruction	Opcode/Function	Syntax	Operation
sb	101000	o $t, i ($s)	MEM [$s + i]:1 = LB ($t)
sh	101001	o $t, i ($s)	MEM [$s + i]:2 = LH ($t)
sw	101011	o $t, i ($s)	MEM [$s + i]:4 = $t

附： MIPS 指令操作码

Data Movement Instructions

Instruction	Opcode/Function	Syntax	Operation
mfhi	010000	f $d	$d = hi
mflo	010010	f $d	$d = lo
mthi	010001	f $s	hi = $s
mtlo	010011	f $s	lo = $s

Exception and Interrupt Instructions

Instruction	Opcode/Function	Syntax	Operation
trap	011010	o i	Dependent on OS; different values for immed26 specify different operations.

实验内容

附：完整 ALU_OP

ALU_OP	十进制	运算功能
0000	0	Result = X << Y　逻辑左移　（Y 取低五位）　Result2=0
0001	1	Result = X >>>Y　算术右移　（Y 取低五位）　Result2=0
0010	2	Result = X >> Y　逻辑右移　（Y 取低五位）　Result2=0
0011	3	Result = $(X * Y)_{[31:0]}$;　Result2 = $(X * Y)_{[63:32]}$　无符号乘法
0100	4	Result = X/Y;　Result2 = X%Y　无符号除法
0101	5	Result = X + Y　(Set OF/UOF)
0110	6	Result = X - Y　(Set OF/UOF)
0111	7	Result = X & Y　按位与
1000	8	Result = X | Y　按位或
1001	9	Result = X⊕Y　按位异或
1010	10	Result = ~(X |Y)　按位或非
1011	11	Result = (X < Y) ? 1 : 0　符号比较
1100	12	Result = (X < Y) ? 1 : 0　无符号比较

实验内容

附：寄存器文件作用

编号	寄存器名称	寄存器描述
0	zero	第0号寄存器，其值始终为0
1	$at	保留寄存器
2～3	$v0~v1	values, 保存表达式或函数返回结果
4-7	$a0~a3	aruments, 作为函数的前4个参数
8～15	t0 t7	temporaries，供汇编程序使用的临时寄存器
16～23	s0 s7	saved values，子函数使用时需要先保存原寄存器的值
24～25	$t8~t9	temporaries, 供汇编程序的临时寄存器，补充$t0~t7
26~27	k0 k1	保留，中断处理函数使用
28	$gp	global pointer，全局指针
29	$sp	stack pointer, 堆栈指针，指向堆栈的栈顶
30	$fp	frame pointer, 保存栈指针
31	$ra	return address, 返回地址

西安交通大学
XI'AN JIAOTONG UNIVERSITY

谢谢！