简易 CPU 的 Verilog 实现

概述

使用 Verilog 实现一个简易的 CPU（以下命名为 ECPU），并仿真了从 0 到 99 求和的机器码指令验证其可正常工作。ECPU 和实际的 CPU 相比有以下不同：没有异常处理，遇到错误指令直接停机；没有流水线设计，执行完当前指令之后才会取下一条指令；指令内存与数据内存分开管理，互不相关；将内存都组织为了 reg 类型的数组形式，寻址时直接进行下标查找。在实际情况中这样的 CPU 是完全无法实现和使用的，但我们的重点在于实现 CPU 对指令的处理流程。

1 实现的指令集及其编码

ECPU 的指令集可以看作 80x86 指令集的一个子集，只可以对十六字节的整数进行操作。具体实现有以下：

a 停机指令与空操作：ECPU 中实现 halt 停机指令和 nop 空操作指令；

b 数据传输指令 mov：在 ECPU 指令集中这条指令用四条语句来代替：rrmov，irmov，rmmov，mrmov。分别实现寄存器到寄存器传输，整数到寄存器传输，寄存器到内存传输，内存到寄存器数据传输；

c 整数算数运算指令 OPq：在 ECPU 中实现四个整数操作，加、减、与、异或；

d 跳转操作 jxx：具体有无条件跳转 jmp，有条件跳转 jl，je，jnz；

e 堆栈操作指令：push，pop；

f 函数调用返回指令 call，ret。

每条指令按照指令指示符+寄存器指示符+常数指示符三部分构成。其中指令指示符占据 8bit，前 4bit 为指令类型码，后 4bit 为指令功能码；如果这条指令使用寄存器，则加一 8bit 寄存器指示符，每 4bit 指示一个寄存器编码（寄存器按照 0 到 14 进行编码，15 表示不使用寄存器）；如果这条指令需要整数，则还有一个 16bit 的整数码。指令为小端序。ECPU 每次读取指令可以根据指令指示符确定这条指令具有的长度，从而确定下一条指令开始的地址，这也保证了指令解释的唯一性。

具体的指令编码见表 1-1，编码的长度从 8bit 到 32bit 不等，全部使用十六进制数字来表示，填入‘-’字符表示这条指令不需要这一部分。

表 1-1 ECPU 指令集编码

指令	指令指示码		寄存器指示码		常数码
halt	0	0	-	-	-
nop	1	0	-	-	-
rrmov rA, rB	2	0	rA	rB	-
irmov V, rB	3	0	F	rB	V
rmmov rA, D(rB)	4	0	rA	rB	
mrmov D(rB), rA	5	0	rA	rB	D
OPq rA, rB	6	fn	rA	rB	-
jxx D	7	fn	-	-	D
call D	8	0	-	-	D
ret	9	0	-	-	-
push rA	A	0	rA	F	-
pop rA	B	0	rA	F	-

2 ECPU 硬件结构

ECPU 的硬件有以下：

a 寄存器数组，一共拥有 15 个寄存器，每个寄存器都为 16 位，256 大小的 reg 数组。每 16 位当作一个寄存器，命名为 rax, rbx, rcx, rdx, rsp, rbp, rsi, rdi, r8, r9, r10, r11, r12, r13, r14，依次编码为 0 到 14，可以与指令中的寄存器指示码进行配合确定唯一寄存器；

b 标志寄存器，只使用 ZF，SF，OF 三个标志寄存器。

c 指令内存，做一个 4000 大小的 reg 类型数组作为指令内存；

d 数据内存，和指令内存大小相同，形式也相同，通过数组下标进行确定寻址；

e PC，一个 16 位寄存器，确定当前指令的地址。

3 ECPU 分阶段处理指令

按照取指，译码，执行，访存，写回，更新 PC 这六个阶段进行处理指令。

取指阶段：ECPU 从指令内存中取出 PC 所指向的数据，前 4bit 给 icode 作为指令码标识，计算是否为合法指令，instr_valid，根据指令码表示计算此条指令是否需要寄存器和

是否需要整数，分别为 need_regids 和 need_valC；接着 4bit 作为功能码给 ifun，接着 4bit 给 rA，接着 4bit 给 rB。最后根据是否需要寄存器将从 PC 开始 8bit 还是 16bit 给 valC，并根据是否需要寄存器和是否需要常数计算下一条指令的 PC。

译码阶段：根据取指阶段得到的 icode, rA, rB 从需要进行读操作的寄存器中得到值，例如如果确定 icode 是 push 则需要取得 rA 和 rsp 的值，如果是 rrmov，那么就需要取出 rA 和 rB 的值，将值暂存于 valA 与 valB。

执行阶段：根据 icode 确定要进行运算的两个对象 aluA 和 aluB，aluA 和 aluB 有以下集中可能：从取指阶段得到的 valC（例如进行 rmmov，mrmov 操作计算内存地址），从译码阶段得到的 valA，valB（例如所有的算数操作），还有 8 或者-8（入栈和出栈操作），还有 0（不进行操作，但需要将内容转移，方便写回）；然后根据 ifun 确定要执行的是什么操作，加、减、与、异或；得到 aluA，aluB，ifun 就可以根据这三者进行计算，得到结果 valE；同时，执行阶段要进行标志寄存器的更新，根据 icode 确定标志寄存器是否需要更新，根据 valE 确定更新后的标志寄存器的新值。

访存阶段：根据 icode 确定需要进行的是读操作还是写操作（读写操作不会同时发生），并且根据 icode 确定要读写取据的内存位置 mem_addr 和要进行写入的数据 mem_data，如果是读操作得到读数据的结果 valM。

写回阶段：根据 icode 确定要进行写回的寄存器，标识 dstM 表示此处写回来自内存的数据，dstE 表示此处写回来自执行阶段的数据，如果是写回来自内存的数据，则使用 dstM 端口写回 valM，如果是其他，则使用 dstE 端口写回 valE。

更新 PC 阶段：根据是 ret，jxx 的 icode 和标志寄存器还有 valP 进行更新 PC 操作。

4 Verilog 代码

Verilog 实现的 ECPU

```
module CPU(clk,
        icode, ifun, rA, rB, valC, valP, instr_valid, need_regids, need_valC,
        srcA, srcB,valA, valB,
        aluA, aluB,alufun,setCC,ZF, SF, OF,valE,
        mem_addr, mem_data,write, read,valM, CS,
        dstE, dstM,
        PC,
        CPU_state, REGS, Memory
);

        input clk;
        input    [4000 : 0]CS;
        output reg [255: 0]REGS;
```

```
        output reg [4000 : 0]Memory;
        output reg [2 : 0]CPU_state;
        reg [2 : 0] n_state;

        localparam fetch = 0, decode = 1, execute = 2, memory = 3,
              write_back = 4, PCupdate = 5, stop = 6, error = 7;

        localparam halt = 0, nop = 1, rrmov = 2, irmov = 3, rmmov = 4, mrmov = 5, OPq = 6,
              jxx = 7, call = 8, ret = 9, push = 10, pop = 11;
        localparam aluAdd = 0, aluSub = 1, aluAnd = 2, aluXor = 3;
        localparam jmp = 0, jl = 1, je = 2, jnz = 3;

        localparam irax = 0, irbx = 1, ircx = 2, irdx = 3, irsp = 4, irbp = 5,
        irsi = 6, irdi = 7, ir8 = 8, ir9 = 9, ir10 = 10, ir11 = 11,
        ir12 = 12, ir13 = 13, ir14 = 14, irno = 15;

        //取指
        output [3 : 0]icode, ifun, rA, rB;
        output [15: 0]valC, valP;
        output instr_valid, need_regids, need_valC;
        //译码
        output [3 : 0]srcA, srcB;
        output [15 : 0]valA, valB;
        //执行
        output [15: 0]aluA, aluB;
        output [3 : 0]alufun;
        output setCC;
        output ZF, SF, OF;
        output [15: 0]valE;

        //访存
        output [15: 0]mem_addr, mem_data;
        output write, read;
        output [15: 0]valM;
        //写回
        output [3 : 0]dstE, dstM;
        //更新 PC
        output reg [15 :0]PC;

        //取指
        assign instr_valid = icode > 12 ? 1 : 0;
        assign need_regids = ((icode >= 2 && icode <= 6) || (icode == 10 || icode == 11)) ? 1 : 0;
        assign need_valC = ((icode >= 3 && icode <= 8) && icode != 6) ? 1 : 0;
        assign icode = CS[PC+: 4];
        assign ifun = CS[(PC + 4)+: 4];
        assign rA = CS[(PC + 8)+: 4];
        assign rB = CS[(PC +12)+: 4];
        assign valC = need_regids? CS[(PC +16)+: 16] : CS[(PC + 8)+: 16];
        assign valP = PC + need_regids * 8 + need_valC * 16 + 8;

        //译码
        assign srcA = ((icode == rrmov) || (icode == rmmov) || (icode == OPq) || (icode == push)) ?
rA :
                              (icode == ret? irsp : 15);
        assign srcB = ((icode == rmmov) || (icode == mrmov) || (icode == OPq)) ? rB :
                              (((icode == push) || (icode == pop) || (icode == call) || (icode == ret))?
irsp : 15);
        assign valA =    REGS[(srcA * 16)+: 16];
        assign valB =    REGS[(srcB * 16)+: 16];
```

```
    //执行
    assign aluA = ((icode == rrmov)|| (icode == OPq))? valA :
                        ((icode == irmov || icode == rmmov || icode == mrmov)? valC:
                            ((icode == call || icode == push)? -8 :
                                ((icode == pop)? 8 : 1)));
    assign aluB = (icode == rmmov || icode == mrmov || icode == OPq || icode == call || icode ==
push ||
                        icode == ret || icode == pop)? valB : ((icode == rrmov || icode == irmov) ?
0 : 1);
    assign alufun = icode == OPq? ifun : 0;
    assign setCC = icode == OPq? 1 : 0;
    assign valE = alufun == aluAdd? aluA + aluB :
        (alufun == aluSub? aluB - aluA :
            (alufun == aluAdd? aluA & aluB :
                (alufun == aluXor? aluA ^ aluB : 1)));

    assign ZF = setCC? (valE == 0? 1 : 0) : ZF;
    assign SF = setCC? (valE[15] ==    1? 1 : 0) : SF;
    assign OF = 0;

    //访存
    assign mem_addr = (icode == rmmov || icode == mrmov || icode == call || icode == push)?
valE :
                        ((icode == pop || icode == ret) ? valA : 1);
    assign mem_data = (icode == rmmov || icode == push)? valA :
                        (icode == call? valP : 1);
    assign read = (icode == mrmov || icode == pop || icode == ret)? 1 : 0;
    assign write = (icode == rmmov || icode == push || icode == call)? 1 : 0;
    assign valM = read? Memory[mem_addr+: 16] : 15;

    //写回
    assign dstM = (icode == mrmov || icode == pop)? rA : 15;
    assign dstE = (icode == rrmov || icode == irmov || icode == OPq)? rB :
                        ((icode == push || icode == pop || icode == call || icode == ret)? irsp : 15);


    reg x;
    initial begin
        PC <= 0;
        CPU_state <= fetch;
        REGS <= 0;
        Memory <= 0;
    end

    always @(posedge clk) begin
        case (CPU_state)
            fetch : begin
                CPU_state <= instr_valid? error : decode;
            end
            decode : begin
                CPU_state <= (icode == halt)? stop : execute;
            end
            execute : begin
                CPU_state <= memory;
            end
            memory : begin
                if(write)
                    Memory[mem_addr+: 16] <= mem_data;
                else
```

```
                        x <= 0;
                    CPU_state <= write_back;
                end
                write_back : begin
                    REGS[(dstE * 16)+: 16] <= valE;
                    REGS[(dstM * 16)+: 16] <= valM;
                    CPU_state <= PCupdate;
                end
                PCupdate : begin
                    PC <= (icode == call ||
                        (icode == jxx && (ifun == 0 || (ifun == jnz && ZF == 0) || (ifun == je
 && ZF == 1) || (ifun == jl && SF == 1))))?
                        valC : (icode == ret? valM : valP);
                    CPU_state <= fetch;
                end
                halt : begin
                    CPU_state <= halt;
                end
                error : begin
                    CPU_state <= fetch;
                end
            endcase
        end
endmodule //
```

4 仿真代码

仿真代码

```
module sim();
    reg clk;
    wire [3 : 0]icode, ifun, rA, rB;
    wire [15: 0]valC, valP;
    wire instr_valid, need_regids, need_valC;
    //译码
    wire [3 : 0]srcA, srcB;
    wire [15 : 0]valA, valB;
    //执行
    wire [15: 0]aluA, aluB;
    wire [3 : 0]alufun;
    wire setCC;
    wire ZF, SF, OF;
    wire [15: 0]valE;
    //访存
    wire [15: 0]mem_addr, mem_data;
    wire write, read;
    wire [15: 0]valM;
    reg [4000 : 0]CS;
    //写回
    wire [3 : 0]dstE, dstM;
    //更新 PC
    wire [15 :0]PC;
  wire [ 2 :0]CPU_state;
    wire [255: 0]REGS;
    wire [4000 : 0]Memory;
    CPU XPU(clk,
    icode, ifun, rA, rB, valC, valP, instr_valid, need_regids, need_valC,
```

```
        srcA, srcB,valA, valB,
        aluA, aluB,alufun,setCC,ZF, SF, OF,valE,
        mem_addr, mem_data,write, read,valM, CS,
        dstE, dstM,
        PC, CPU_state, REGS, Memory);
        /*
            0000    irmov 0, rax        30 f0 00 00
            0020    irmov 0, rdx        30 f3 00 00
            0040    irmov 1, rbx        30 f1 10 00
            0060    irmov 100, rcx      30 f2 46 00
            0080    Opadd rbx, rax      60 10
            0090    Opadd rax, rdx      60 03
            00a0    Opsub rbx, rcx      61 12
            00b0    jne   0080          73 08 00
            00d0    halt                10
        */
        initial begin
            clk = 0;
            CS = 228'b
 0000_0001_0000_0000_1000_0000_0011_0111_0010_0001_0001_0110_0011_0000_0000_0110_
 0000_0001_0000_0110_0000_0000_0110_0100_0010_1111_0000_0011_0000_0000_0000_0001_
 0001_1111_0000_0011_0000_0000_0000_0000_0011_1111_0000_0011_0000_0000_0000_0000_
 0000_1111_0000_0011;
        end
        always begin
            #5 clk = ~clk;
        end
 endmodule //
```

5 仿真波形及解释

仿真文件执行的代码，代码功能为计算从 0 - 99 的数字。截取其中一段仿真来验证 ECPU 的执行：

```
    0000    irmov 0, rax        30 f0 00 00
    0020    irmov 0, rdx        30 f3 00 00
    0040    irmov 1, rbx        30 f1 10 00
    0060    irmov 100, rcx      30 f2 46 00
    0080    Opadd rbx, rax      60 10
    0090    Opadd rax, rdx      60 03
    00a0    Opsub rbx, rcx      61 12
    00b0    jne   0080          73 08 00
    00d0    halt                10
```

最初执行的指令为 irmov 0, rax 在取指阶段，PC 应该为 0000，然后 icode 是 3，ifun 是 0 表示加法，rA 为 f 表示不使用，rB 为 0 表示使用的寄存器为 rax，valC 为 0，valP 为 PC + 8 + 16（寄存器和常数都需要）；进入译码阶段，不需要取得任何寄存器的值；再进入执行阶段，首先 aluA 变为 valC 即为 0，aluB 变为 0，由于 ifun 为加法，所以得到的 valE 也为 0，不更新标志寄存器；进入访存阶段，不需要进行读写操作；进入写回阶段更新 dstE 为 rA 即 0，即将 valE 写回到 rax。

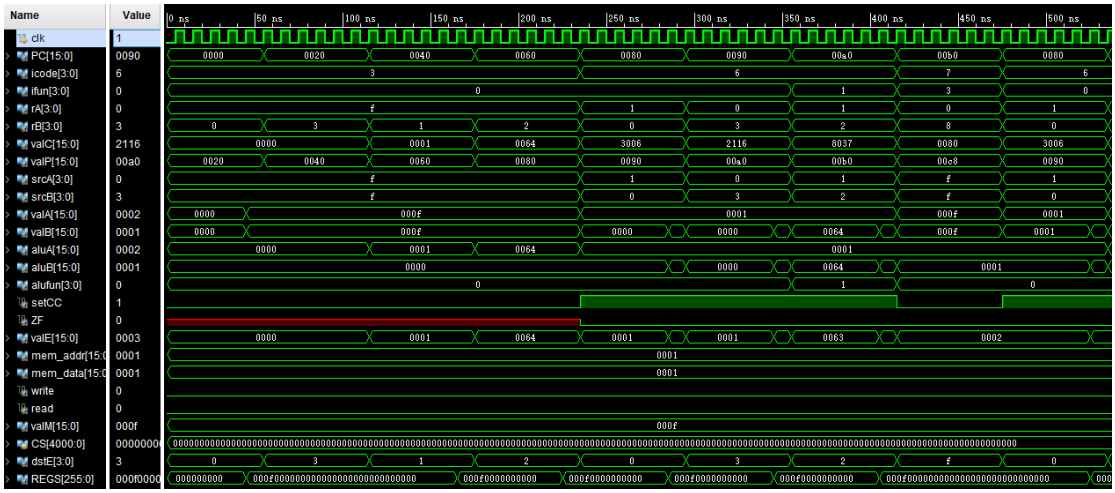

图 1 仿真图

限于篇幅，剩余指令不做一一阐述，读者可以仔细阅读代码，自行仿真。

6 参考文献

[1]Randal E.Bryant,David R.O'Hallaron.深入理解计算机系统基础.机械工业出版社,2016,7