数字逻辑与处理器基础
单周期处理器

无 81 马啸阳 2018011054

2020 年 5 月 19 日

1 处理器结构

1. RegDst 信号控制的是寄存器堆的写地址。输入 2 对应的输出 31 对应寄存器 $ra。只有在 jal 指令中，一定写入寄存器 $ra，故当且仅当执行 jal 指令时 RegDst 信号为 2。

2. ALUSrc1 信号控制的是 ALU 的第一个输入。输入 1 对应的指令 [10:6] 是 shamt[4:0]，即位移量，在 sll、srl、sra 这三个位移指令中需要位移量。其它指令中不需要 ALU（ALUSrc1 信号为 X），或者 ALU 第一个输入为寄存器 $rs 中读得的数据（ALUSrc1 信号为 0）。

3. MemtoReg 信号控制的是寄存器堆写入数据。输入 2 对应的是 PC+4。jal 和 jalr 指令会将 PC+4 写入寄存器，其它指令不会写 PC+4 这一指令地址。因此当且仅当执行 jal 和 jalr 时 MemtoReg 信号为 2。

4. PCSrc 信号控制的是程序计数器的输入。输入 2 对应的是寄存器 $rs 中取出的数据。仅在 jr 和 jalr 指令中，程序计数器的下一个值是从寄存器中读出的。因此当且仅当执行 jal 和 jalr 时 PCSrc 信号为 2。

5. ExtOp 信号控制的是立即数的扩展。输入 1 对应符号扩展，输入 0 对应无符号扩展。lw、sw、addi、addiu、slti、sltiu、beq 是符号扩展，ExtOp 输入 1，andi 是无符号扩展，ExtOp 输入 0。

6. nop 指令全 0，目前对应 sll $0, $0, 0。这条指令实际上也相当于空指令（$0 寄存器不可修改且对 0 移位无变化），因此不必修改处理器结构。

	PCSrc[1:0]	Branch	RegWrite	RegDst[1:0]	MemRead	MemWrite	MemtoReg[1:0]	ALUSrc1	ALUSrc2	ExtOp	LuOp
lw	0	0	1	0	1	0	1	0	1	1	0
sw	0	0	0	X	0	1	X	0	1	1	0
lui	0	0	1	0	0	0	0	0	1	X	1
add	0	0	1	1	0	0	0	0	0	X	X
addu	0	0	1	1	0	0	0	0	0	X	X
sub	0	0	1	1	0	0	0	0	0	X	X
subu	0	0	1	1	0	0	0	0	0	X	X
addi	0	0	1	0	0	0	0	0	1	1	0
addiu	0	0	1	0	0	0	0	0	1	1	0
and	0	0	1	1	0	0	0	0	0	X	X
or	0	0	1	1	0	0	0	0	0	X	X
xor	0	0	1	1	0	0	0	0	0	X	X
nor	0	0	1	1	0	0	0	0	0	X	X
andi	0	0	1	0	0	0	0	0	1	0	0
sll	0	0	1	1	0	0	0	1	0	X	X
srl	0	0	1	1	0	0	0	1	0	X	X
sra	0	0	1	1	0	0	0	1	0	X	X
slt	0	0	1	1	0	0	0	0	0	X	X
sltu	0	0	1	1	0	0	0	0	0	X	X
slti	0	0	1	0	0	0	0	0	1	1	0
sltiu	0	0	1	0	0	0	0	0	1	1	0
beq	0	1	0	X	0	0	X	0	0	1	0
j	1	X	0	X	0	0	X	X	X	X	X
jal	1	X	1	2	0	0	2	X	X	X	X
jr	2	X	0	X	0	0	X	X	X	X	X
jalr	2	X	1	1	0	0	2	X	X	X	X

表 1: 控制信号真值表

2 完成控制器

补全 Control.v 代码如下，即按照控制信号真值表按照指令给定所有控制信号。

```
module Control(OpCode, Funct,
        PCSrc, Branch, RegWrite, RegDst,
        MemRead, MemWrite, MemtoReg,
        ALUSrc1, ALUSrc2, ExtOp, LuOp, ALUOp);
        input [5:0] OpCode;
        input [5:0] Funct;
        output [1:0] PCSrc;
        output Branch;
        output RegWrite;
        output [1:0] RegDst;
        output MemRead;
        output MemWrite;
        output [1:0] MemtoReg;
        output ALUSrc1;
        output ALUSrc2;
        output ExtOp;
        output LuOp;
        output [3:0] ALUOp;

        assign PCSrc[1:0] =
                (OpCode == 6'h02 || OpCode == 6'h03)? 2'b01:
                (OpCode == 6'h00 && (Funct == 5'h08 || Funct == 5'h09))? 2'b10:
                2'b00;

        assign Branch =
                (OpCode == 6'h04)? 1'b1:
                1'b0;

        assign RegWrite =
                (OpCode == 6'h2b || OpCode == 6'h02 || OpCode == 6'h04 ||
                (OpCode == 6'h00 && Funct == 5'h08))? 1'b0:
                1'b1;

        assign RegDst[1:0] =
                (OpCode == 6'h23 || OpCode == 6'h0f || OpCode == 6'h08 ||
                OpCode == 6'h09 || OpCode == 6'h0c || OpCode == 6'h0a ||
```

```
37              OpCode == 6'h0b)? 2'b00:
38              (OpCode == 6'h03)? 2'b10:
39              2'b01;
40
41      assign MemRead =
42              (OpCode == 6'h23)? 1'b1:
43              1'b0;
44
45      assign MemWrite =
46              (OpCode == 6'h2b)? 1'b1:
47              1'b0;
48
49      assign MemtoReg[1:0] =
50              (OpCode == 6'h23)? 2'b01:
51              (OpCode == 6'h03 ||
52              (OpCode == 6'h00 && Funct == 5'h09))? 2'b10:
53              2'b00;
54
55      assign ALUSrc1 =
56              (OpCode == 6'h00 && (Funct == 5'h00 ||
57              Funct == 5'h02 || Funct == 5'h03))? 1'b1:
58              1'b0;
59
60      assign ALUSrc2 =
61              (OpCode == 6'h23 || OpCode == 6'h2b ||
62              OpCode == 6'h0f || OpCode == 6'h08 ||
63              OpCode == 6'h09 || OpCode == 6'h0c ||
64              OpCode == 6'h0a || OpCode == 6'h0b)? 1'b1:
65              1'b0;
66
67      assign ExtOp =
68              (OpCode == 6'h0c)? 1'b0:
69              1'b1;
70
71      assign LuOp =
72              (OpCode == 6'h0f)? 1'b1:
73              1'b0;
74
75      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:
                3'b000;

        assign ALUOp[3] = OpCode[0];

endmodule
```

对于指令寄存器中的汇编代码，其含义按注释补充于如下代码中。其中对于 $a0 与 $t2 所存的值，$a0 最高位为 0，$t2 最高位为 1，因而 slt 中比较有符号数时 $a0 为正，$t2 为负，$a0>$t2；而 sltu 时比较无符号数，$t2 最高位更高，$a0<$t2。对于 addi 和 addiu，忽略溢出没有区别。

```
    addi $a0, $zero, 12345      # $a0=12345=0x00003039
    addiu $a1, $zero, -11215    # $a1=-11215=0xffffd431
    sll $a2, $a1, 16            # $a2=0xd4310000
    sra $a3, $a2, 16            # $a3=0xffffd431
    beq $a3, $a1, L1            # $a1=$a2, so jump to L1
    lui $a0, -11111             # not operated
L1:
    add $t0, $a2, $a0           # $t0=$a2+$a0=0xd4313039
    sra $t1, $t0, 8             # $t1=0xffd43130
    addi $t2, $zero, -12345     # $t2=-12345=0xffffcfc7
    slt $v0, $a0, $t2           # $a0>$t2, so $v0=0
    sltu $v1, $a0, $t2          # $a0<$t2, so $v1=1
Loop:
    j Loop
```

程序执行充分长时间后，在最后一句 j Loop 上死循环。最终各寄存器的值如下（此处使用 java 记号，» 代表算术右移）。

$a0=12345=0x00003039
$a1=-11215=0xffffd431
$a2=$a1<<16=0xd4310000
$a3=$a2>>16=0xffffd431
$t0=$a2+$a0=0xd4313039
$t1=$t0>>8=0xffd43130

$t2=-12345=0xffffcfc7

$v0=0

$v1=1

某寄存器中存放着数 0xffffcfc7 时并不能区分它是有符号数还是无符号数，这在 slt 和 sltu 的两条指令中已经证实。

仿真结果 1如图所示，寄存器堆的值如图 2所示。PC 的值每周期增加 4，代表当前周期所执行的指令地址。除了 beq 指令（Branch 信号为高的周期）使得下一条指令跳转到 L1 标签，PC 增加了 8。

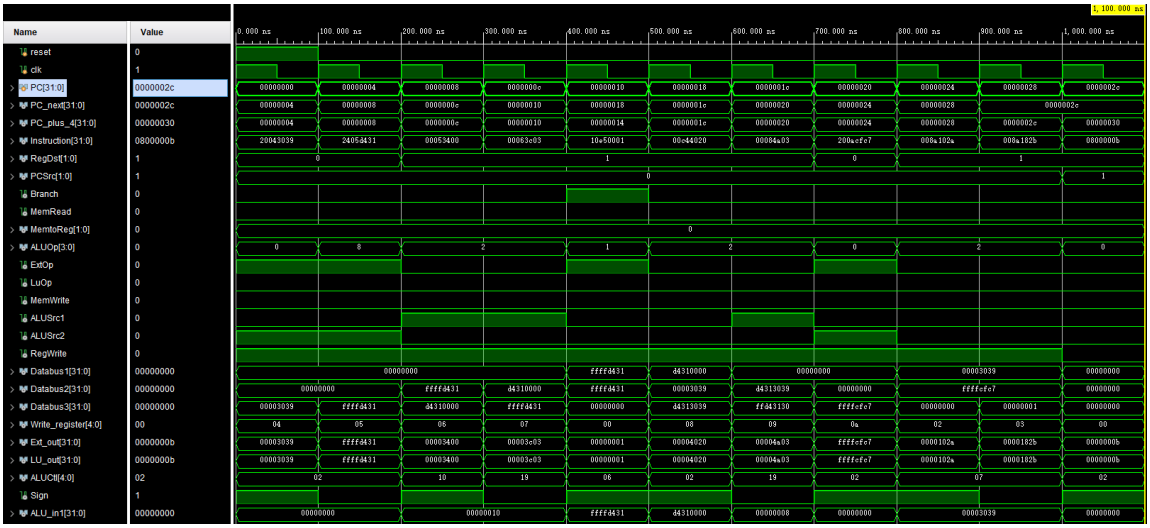

图 1: 仿真结果

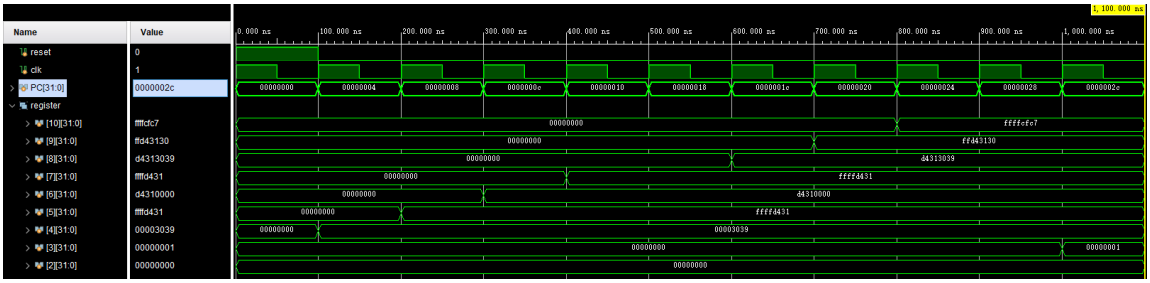

图 2: 寄存器堆仿真结果

Branch 信号在第 5 个周期为 1，即对应第 5 条 beq 指令，PC 在这一周期到下一周期间

增加了 8，即跳过一条指令。

100~200ns 间，PC 是 4，对应 addiu $a1, $zero, -11215 指令，此时 $a1 的值是 0x00000000。200~300ns 间，$a1 的值是 0xffffd431。下一条指令立即使用到了 $a1 的值，但不会发生错误。因为 ALU 计算完毕后，要在下一周期开始的上升沿写入寄存器，而读是任意时刻进行的，因而写完后立刻反映在输出上，交给 ALU 的值是写入后的正确的值。

运行足够长的时间后，由图 2所示，寄存器的值如下，与预期结果一致。

$a0=0x00003039
$a1=0xffffd431
$a2=0xd4310000
$a3=0xffffd431
$t0=0xd4313039
$t1=0xffd43130
$t2=0xffffcfc7
$v0=0
$v1=1

3 执行汇编程序

```
    addi $a0, $zero, 4      # $a0=4
    jal sum                 # jump to sum, $ra points to Loop
Loop:
    beq $zero, $zero, Loop  # infinite Loop
sum:
    # if($a0<1)
    #   return 0;
    # else
    #   goto L1;
    #   (return $a0+sum($a0-1))
    addi $sp, $sp, -8       # $sp=$sp-8
    sw $ra, 4($sp)          # save $ra in ($sp)[4]
    sw $a0, 0($sp)          # save $a0 in ($sp)[0]
    slti $t0, $a0, 1        # if $a0<1 then $t0=1
    beq $t0, $zero, L1      # if $t0==0 ($a0>=1) then jump to L1
    xor $v0, $zero, $zero   # $v0=0
    addi $sp, $sp, 8        # $sp=$sp+8
```

```
18      jr $ra                  # jump to $ra
19  L1:
20      # return $a0+sum($a0-1)
21      addi $a0, $a0, -1       # $a0=$a0-1
22      jal sum                 # jump to sum, $ra points to next instruction
23      lw $a0, 0($sp)          # load ($sp)[0] to $a0
24      lw $ra, 4($sp)          # load ($sp)[4] to $ra
25      addi $sp, $sp, 8        # $sp=$sp+8
26      add $v0, $a0, $v0       # $v0=$a0+$v0
27      jr $ra                  # jump to $ra
```

这段代码展现了函数调用，主程序调用 sum(n)（若第一条指令中 4 改为 n)。sum 中为条件分支，忽略其中对寄存器的保存恢复，则 sum 中代码如上述注释。当 $a0<1 时返回 0，反之跳转至 L1，L1 调用 sum，返回 $a0-1+sum($a0-1)。因此此程序完成 $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$，n<=0 时返回 0，计算结果置于 $v0 中，然后跳至 Loop，进入死循环，各寄存器值不变。

根据各指令格式将各语句翻译为机器码，得到如下 18 条机器码指令（以下为十六进制)。

```
20040004
0c000003
1000ffff
23bdfff8
afbf0004
afa40000
28880001
11000003
00001026
23bd0008
03e00008
2084ffff
0c000003
8fa40000
8fbf0004
23bd0008
00821020
03e00008
```

beq 语句是 PC 相对寻址，标签对应于相对下一条指令的 PC 位移。beq $t0, $zero, L1 翻

译为 0x11000003（6'h04, 5'h08, 5'h00, 16'h0003），这里 6'h04 对应 beq 的 OpCode，5'h08 对应 $t0 的地址，5'h00 对应 $zero 地址，16'h003 为位移量。由于 L1 在此语句下一条语句（xor $v0, $zero, $zero）的三条指令后，因此位移为 3。其它标签类似。如 beq $zero, $zero, Loop 中，由于 Loop 即自身，相对于下一条指令是一条指令之前，因此位移为-1（即 16'hffff），从而指令翻译为 0x1000ffff（6'h04, 5'h00, 5'h00, 16'hffff）。

而 jal 是伪直接寻址，标签翻译为指令地址的第 27 至 2 位。sum 的地址是 0x0c，因而 jal sum 中 sum 翻译为 26'h0000003，从而 jal sum 为 0x0c100003（6'h03, 26'h0000003）。

立即数中负数以补码表示，例如-8 的 16 位十六进制补码为 0xfff8，-1 为 0xffff。之后它们会进行符号扩展送入 ALU。addi $sp, $sp, -8 被翻译为 0x23bdfff8（6'h08, 5'h1d, 5'h1d, 16'hfff8），其中 5'h1d（29）对应 $sp 地址。addi $a0, $a0, -1 被翻译为 0x2084ffff（6'h08, 5'h04, 5'h04, 16'hffff），其中 5'h04 对应 $a0 地址。

修改 InstructionMemory.v 如下存储前述汇编指令。

```
module InstructionMemory(Address, Instruction);
        input [31:0] Address;
        output reg [31:0] Instruction;

        always @(*)
                case (Address[9:2])
                        8'd0: Instruction <= 32'h20040004;
                        8'd1: Instruction <= 32'h0c000003;
                        8'd2: Instruction <= 32'h1000ffff;
                        8'd3: Instruction <= 32'h23bdfff8;
                        8'd4: Instruction <= 32'hafbf0004;
                        8'd5: Instruction <= 32'hafa40000;
                        8'd6: Instruction <= 32'h28880001;
                        8'd7: Instruction <= 32'h11000003;
                        8'd8: Instruction <= 32'h00001026;
                        8'd9: Instruction <= 32'h23bd0008;
                        8'd10: Instruction <= 32'h03e00008;
                        8'd11: Instruction <= 32'h2084ffff;
                        8'd12: Instruction <= 32'h0c000003;
                        8'd13: Instruction <= 32'h8fa40000;
                        8'd14: Instruction <= 32'h8fbf0004;
                        8'd15: Instruction <= 32'h23bd0008;
                        8'd16: Instruction <= 32'h00821020;
                        8'd17: Instruction <= 32'h03e00008;
                        default: Instruction <= 32'h00000000;
```

```
26                  endcase
27
28  endmodule
```

仿真结果如图 3所示。执行充分长时间后，$v0 的值为 0x0000000a，确为 1+2+3+4=10，$a0 的值在多次压栈弹栈后回到原来的值 0x00000004，符合预期。

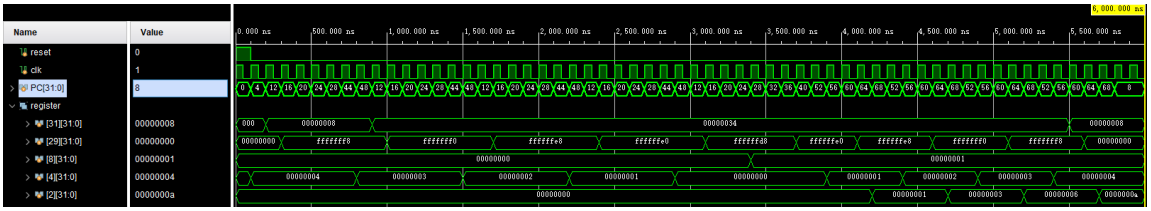

图 3: 汇编程序仿真结果

PC 的值每周期为当前执行的指令地址。例如第二周期 PC=0x00000004，执行第二条指令 jal sum，跳转至 sum，从而第三周期 PC=0x0000000b，到达 sum 的开始位置。

其余各寄存器变化按时间变化列举如下，以下均为十六进制。

第一周期将 4 置入 $a0，第二周期开始 $a0=0x00000004。

第二周期将当前指令下一条地址置入 $ra，第三周期开始 $ra=0x00000008，同时 PC 跳转至 sum，即 0x0000000c。

第三周期将 $sp 减 8，在后两个周期进行压栈，第四周期开始 $sp 变为 0xfffffff8（此处 $sp 初始化为 0，从而栈在数据存储器的最后反向增长）。

第七周期执行 beq $t0, $zero, L1，跳转至 L1，第八周期开始 PC 变为 0x0000002c。

第八周期 $a0 减 1，第九周期开始 $a0=0x00000003 准备调用 sum(3)。

第九周期执行 jal sum，调用 sum(3)，将下一条指令地址置入 $ra，第十周期开始 $ra=0x00000034，同时 PC 跳转至 sum，即 0x0000000c。

第十周期与第三周期指令一致，第十一周期开始 $sp 减 8 变为 0xfffffff0。

第十四周期与第七周期指令一致，跳转至 L1。

第十五周期与第八周期指令一致，$a0 减 1，第九周期开始 $a0=0x00000002 准备调用 sum(2)。

第十六周期与第九周期指令一致，执行 jal sum，调用 sum(2)，下一条指令仍然是 0x00000034，$ra 没有实际变化，PC 跳转至 sum。

第十七至二十三周期、第二十四至三十周期与第三至九、第十至十六周期一致。第十八周期开始 $sp 变为 0xffffffe8，第二十三周期开始 $a0 变为 0x00000001 准备调用 sum(1)，第二十五周期开始 $sp 变为 0xffffffe0，第三十周期开始 $a0 变为 0x00000000 准备调用 sum(0)。

第三十一周期将 $sp 减 8，第三十二周期开始 $sp 变为 0xfffffd8。

第三十五周期执行 beq $t0, $zero, L1，由于 $a0 已递归至 0，因此不跳转。

第三十六周期置 $v0 为 0。sum(0) 返回 0。

第三十七周期开始弹栈，$sp 增 8，第三十八周期开始 $sp=0xffffffe0。

第三十八周期执行 jr $ra，sum(0) 执行完毕，进入 sum(1) 的函数栈，第三十九周期开始 PC 跳转至 lw $a0, 0($sp) 一句指令，即 0x00000038。

第三十九周期由栈中恢复 $a0，第四十周期开始 $a0=0x00000001。

第四十周期由栈中恢复 $ra，而 sum(1) 的 $ra 仍是 0x00000034，实际不变。

第四十一周期开始弹栈，$sp 增 8，第四十二周期开始 $sp=0xffffffe8。

第四十二周期执行 add $v0, $a0, $v0，第四十三周期开始 $v0=0x00000001。sum(1) 返回 1。

第四十三周期执行 jr $ra，sum(1) 执行完毕，进入 sum(2) 函数栈，第四十四周期开始 PC 跳转至 0x00000038。

第四十四至四十八周期、第四十九至五十三周期与第三十九至四十三周期指令一致。第四十五周期开始 $a0 变为 0x00000002，第四十六周期开始恢复 $ra，第四十七周期开始 $sp 变为 0xfffffff0，第四十八周期开始 $v0=2+$v0=3（sum(2) 返回 3）。第五十周期开始 $a0 变为 0x00000003，第五十一周期开始恢复 $ra，第五十二周期开始 $sp 变为 0xfffffff8，第五十三周期开始 $v0=3+$v0=6（sum(3) 返回 6）。

第五十四周期 PC 又跳回 0x00000034，进入 sum(4) 函数栈。同上，第五十五周期开始 $a0 变为 0x00000004，第五十六周期开始恢复 $ra，$ra 变为 0x00000008 为 Loop 的地址，第五十七周期开始 $sp 变为 0x00000000，第五十八周期开始 $v0=4+$v0=10（sum(4) 返回 10）。第五十八周期执行 jal $ra 后，第五十九周期 PC 跳转为 0x00000008，跳转到 Loop。此后所有寄存器与 PC 值不再改变。