实验二 多周期 MIPS CPU

张海斌*

2019 年 5 月

目 录

1 概要 1

2 功能展示 1

3 代码结构 2

4 模块设计 2

5 数据通路 3

6 控制器 4

7 多周期状态转移的改进 5

8 仿真结果 6

1 概要

多周期处理器与单周器处理器相比有很多相似的地方，比如都可以分为控制器和数据通路两大部份，都需要除通用寄存器外的寄存器存储程序运行状态（如需要寄存器记录 PC 位置）。多周期处理器与单周器处理器中，有许多部件也是可以通用的。但是在存储器上有一点区别是多周期处理器中需要执行的机器代码与内存空间是在同一存储空间中，这也就更符合冯·诺依曼体系结构。

2 功能展示

LED[3:0] 四位显示当前多周期处理器的状态的二进制编码，状态编码方式见 Table 1。LED[15] 显示 CPU 时钟信号状态。从左向右前两个七段数码管显示当前 PC 值，接着的两个

* 学号 17307130118

状态名称	状态编码	解释
FETCH	4’b0000	取指令
DECODE	4’b0001	解码机器指令
MEMADR	4’b0010	`lw` 或 `sw` 指令中计算内存地址
MEMRD	4’b0011	读取内存数据
MEMWB	4’b0100	写回数据到寄存器中
MEMWR	4’b0101	写回数据到内存中
REX	4’b0110	R-Type 指令执行阶段（进行计算）
RWB	4’b0111	R-Type 指令计算结果写回寄存器
BEX	4’b1000	`beq` 或 `bne` 指令执行阶段
IEX	4’b1001	I-Type 指令执行阶段
IWB	4’b1010	I-Type 指令写回阶段
JEX	4’b1011	`jump` 指令执行阶段

Table 1: 有限状态机状态编码

数码管显示写入 PC 使能寄存器的值（只有当有使能信号的时候才会写入 PC），最后四位七段数码管显示内存或寄存器中的值。SW[0] 控制状态的清零，SW[1] 控制 CPU 的运行和暂停，SW[9:4] 提供需要查看的寄存器或内存的地址：当 SW[2] 为 0 时，以 SW[8:4] 的内容为寄存器地址，显示相应寄存器中的数据在右侧四个七段数码管上；当 SW[2] 为 1 时，以 SW[9:4] 的内容为内存地址，在相同的七段数码管上显示内存的数据。

3 代码结构

Figure 1 展现了 Verilog 中各个模块的层次结构。`top` 是上板子的顶层文件，下面有两大模块 `mips_top` `display` 及两个时钟分频器 `clkdiv`。`mips_top` 和 `display`，分别为多周期处理器的顶层模块和负责处理板子显示的模块。在显示模块中包括了处理八个七段数码管连续显示的 `async7seg` 模块，而它下含的 `Hex7Seg` 则是将十六进制数转化为七段数码管上数字或字母显示的信号。`mips_top` 模块下主要分为两个模块：`controller` 控制器和 `datapath` 数据通路。

4 模块设计

`display` 模块中，主要是使用 `case` 语句判断信号条件，从而决定七段数码管右侧四个所要展示的数据（见 Listing 1），另一方面为了避免暂停 CPU 的时候导致时钟暂停，从而影响不同的七段数码管同时显示数据的功能，我在 `top` 模块中使用了两个时钟分频模块 `clkdiv` 分别为 CPU 和七段数码管的显示提供时钟信号。

在多周期处理器中，由于保存 PC 的寄存器受到使能信号的控制，需要设计新的带使能端的寄存器 `enreg`。还有在数据通路中，出现了四选一多路选择器，这也需要添加相应模块。而其它的一些基本模块不需要进行修改，其中包括接收是否需要符号位扩展信号的扩位器 `ext`、时钟沿触发器 `flopr`、运算单元 `ALU`、左移两位的移位器 `sl2`、二选一多路选择器 `mux2` 和寄

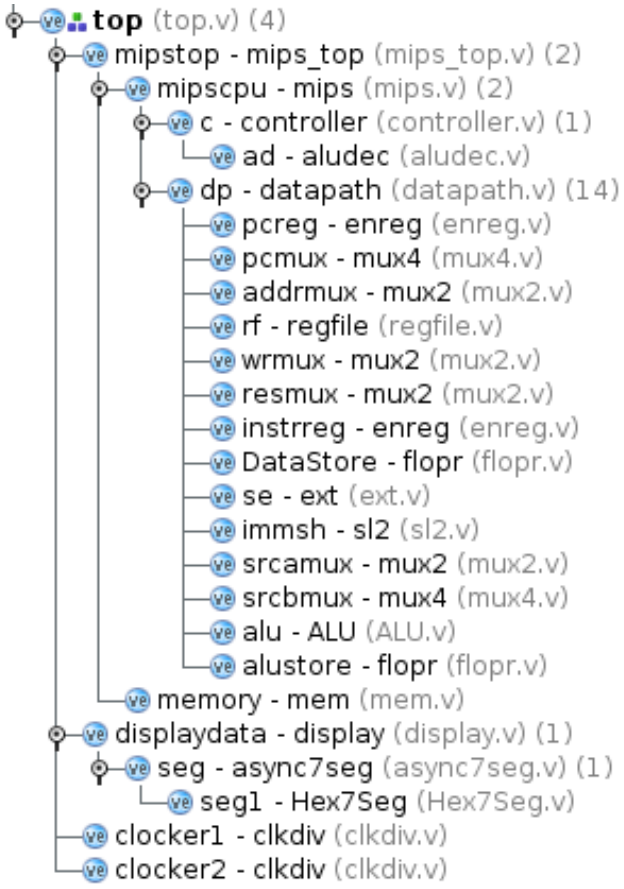

Figure 1: Verilog 模块层次结构图

存器文件 `regfile`。内存模块中新添加了初始化代码，这原本是在指令内存 `imem` 中进行的操作，而现在在我们的多周期处理器中内存同时肩负保存运行的机器指令以及内存数据的职责，具体代码可以参考 Listing 2。

在 CPU 顶层文件 `mips_top` 中包括 `mips` CPU 核心模块和数据存储的内存模块 `mem`，以及在 `mips` CPU 核心模块中包括控制器 `controller` 和数据通路 `datapath` 都不需要进行过多的解释说明。

5 数据通路

数据通路模块 `datapath` 这一部份我大部份是按照 Figure 2中的线路进行连接，不过有几个地方我进行了修改。一个是寄存器文件输出 `RD1` 和 `RD2` 后有一个寄存器，它被用来稳定 `A` 和 `B` 的信号。因为在一条指令周期中没有非常必要的作用我将它删去了。由于在 I-Type 指令中存在位操作运算，所以需要对 16 位的立即数进行 0 扩展，因此数据通路中的扩位器需要有有符号和无符号扩展两种操作，同时需要一位信号 `ExtOp` 来控制这个模块的操作。最后一点不同在于，书中 `PCEn` 是在数据通路中通过 `Zero`、 `Branch` 和 `PCWrite` 三个信号的逻辑操作得到的，而我将 `PCEn` 信号的生成放在了控制器 `controller` 中。也就是说，数据通路将 `Zero` 信号传递给 `controller`，然后 `controller` 将 `PCEn` 信号回传给数据通路。

数据通路中，我认为值得注意的一个地方是内存与寄存器文件之间的 `Data` 寄存器

```
        case(Ctl[0])
            1'b0:
            begin
                DispReadReg <= Addr[4:0];
                DispReadMem <= 6'b0;
                Data[15:0] <= DispRegData;
            end
            1'b1:
            begin
                DispReadReg <= 5'b0;
                DispReadMem <= Addr;
                Data[15:0] <= DispMemData[15:0];
            end
        endcase
```

Listing 1: `display` 模块数据显示的选择

与 `ALUResult` 与 `ALUOut` 之间的寄存器。它们的存在让进入的数据信号晚一个时钟周期才向后传递。这在控制信号的设计中需要注意理解。

6 控制器

多周期处理器中控制器信号要比单周器控制器更加复杂，因为多周期处理器中一条指令需要多个时钟周期才嫩完成，在执行同一条指令中的不同时钟周期中，控制信号也会有所差异，而单周器处理器在每个周期内只需要根据机器指令生成对应的控制信号。我使用了有限状态机来实现这个控制器。Table 1列出了有限状态机的状态编码。不过其中 I-Type 我并不是指所有的 I-Type 指令，而是代表 I-Type 中所有类似于 R-Type 指令使用 ALU 进行数据计算并将结果写回到寄存器中的运算指令。

所以在控制器中，我将多种 I-Type 指令的状态合并到一起，用 `IEX` 和 `IWB` 代表它们的状态。而当前状态用四位的寄存器 `state` 储存。

在状态转移图中，我将 `beq` 和 `bne` 的状态合并在一起，将多个 I-Type 的状态合并到一起，减少了状态的总数，但同时输出信号的逻辑运算过程更加复杂了。 `beq` 和 `bne` 的六位操作码中，只有最低位不同，因此我用 `assign bne = Op[0];` 获得是否为 `bne` 的信号，然后由 `assign PCEn = ((Zero ^ bne) & Branch) | PCWrite;` 得到 `PCEn` 的结果。 `Branch` 只有在处于状态 `BEX` 的时候才是 1；在状态不是 `BEX` 的时候， `Branch` 始终为 0，所以此时 `PCEn` 只受到 `PCWrite` 的控制，不会造成异常。代码见 Listing 3。

对于 `ALUCtl` ALU 控制信号的产生上，我会从两个 `ALUOp` 中选择一个传递到 `ALUCtl` 。其中， `ALUOp0` 是根据有限状态机当前状态生成的，而 `ALUOp1` 是只根据 Op Code 和 Funct 而决定的在执行阶段所需要的 ALU 控制信号，它是由模块 `aludec` 生成的，随之一起生成的信号还与 `ExtOp1` 扩位运算信号。当处于 `REX` 或 `IEX` 状态时，会使用 `ALUOp1` 信号与 `ExtOp1` 信号

```
module mem(
    input CLK, WE,
    input [31:0] A, WD,
    output [31:0] RD,
    input [5:0] ReadAddr,
    output [31:0] Data);
// Instruction and Data Memory
    reg [31:0] RAM[63:0];

    initial
        $readmemh("/PAHT/TO/memfile.dat", RAM); // $
    assign RD = RAM[A[31:2]];
    assign Data = RAM[ReadAddr];

    always @(posedge CLK)
        if (WE)
            RAM[A[31:2]] <= WD;
        else
            RAM[A[31:2]] <= RAM[A[31:2]];
endmodule
```

Listing 2: `mem` 内存模块

作为输出信号，其它情况都会使用 `ALUOp0` 信号与 `ExtOp0` 信号作为输出。另外，`ExtOp1` 信号始终为 1。相关代码见 Listing 4。

所以 `aludec` 模块的作用就是根据 Op Code 和 Funct 生成在执行阶段所需要的 `ALUOp` 和 `ExtOp` 信号。这两个信号也只有在 `REX` 或 `IEX` 状态时才会被使用。通过这种方法避免了不同的 I-Type 需要添加不同的状态。

在 `controller` 模块中，分别有一个 `case` 语句来判断有限状态机的下一个周期的状态以及生成当前状态下的输出信号，这部份输出信号被并为了总线 `ctls`。`assign {PCWrite, MemWrite, IRWri` 是 `ctls` 的连线情况。而下一周期的状态由当前状态和 Op Code 共同决定。

7 多周期状态转移的改进

在列出了状态输出表之后，我发现在某些状态下使用的部件非常少，有的不同周期可以进行合并。`MEMWB`、`RWB`、`IWB` 三个状态与 `FETCH` 状态的有效输出信号恰好相错开，也就是说，我们可以在前三个状态执行的同时，将下一个指令周期的 `FETCH` 阶段的工作同时完成掉。这样，执行完前三个状态后，可以直接跳到 `DECODE` 阶段。而这对程序的运行没有任何影响。而且这个改进过程只涉及到了相关状态的状态转移和状态输出的改变，比较容易实现。

修改前后，仿真跑出来的时间分别为 620ns 和 510ns，说明这个优化对处理器运行速度

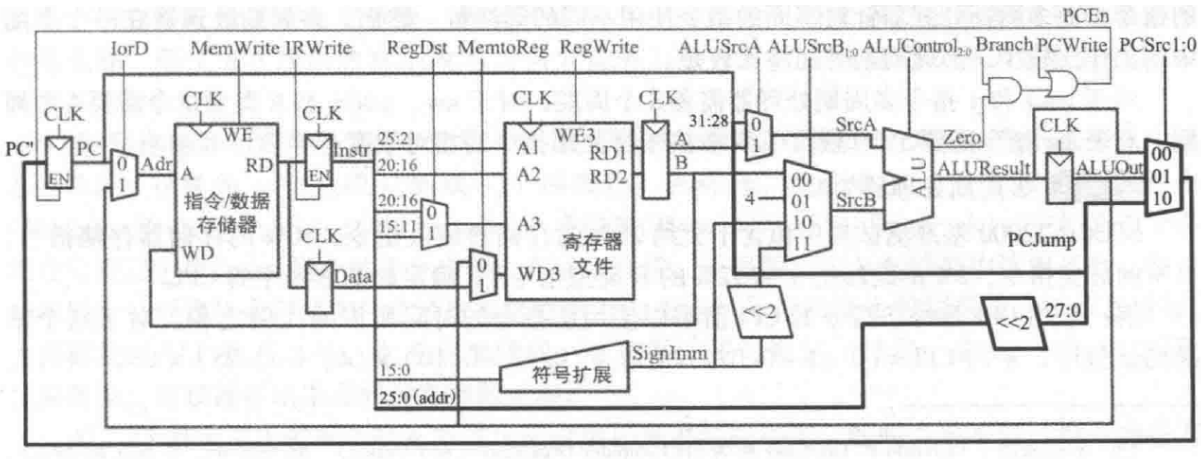

Figure 2: 多周期数据通路线路图

```
    // Branch 只有在 BEQ 或 BNE 的执行阶段才为 1
    always @(*)
        if (state == BEX)
            Branch <= 1;
        else
            Branch <= 0;
```

Listing 3: `Branch` 信号控制

的提升还是比较明显的。

8 仿真结果

Figure 4是测试用的代码。Figure 5和 Figure 6是没有优化状态转移前的代码测试结果；Figure 7和 Figure 8是优化状态转移后的代码测试结果。其中需要注意的地方是，`PC` 并不一定代表当前执行代码的位置，因为在 `FETCH` 阶段会将 `PC` 加 4 并写入 `PC`，这样 `PC` 就指向下一条指令了，但实际上处理器还没有执行到下一条指令。同理，对于跳转指令 `jump` 或 `beq` 或 `bne`，只有根据下一个状态为 0 的时候的 `PC` 的值才能判断是否发生了跳转。

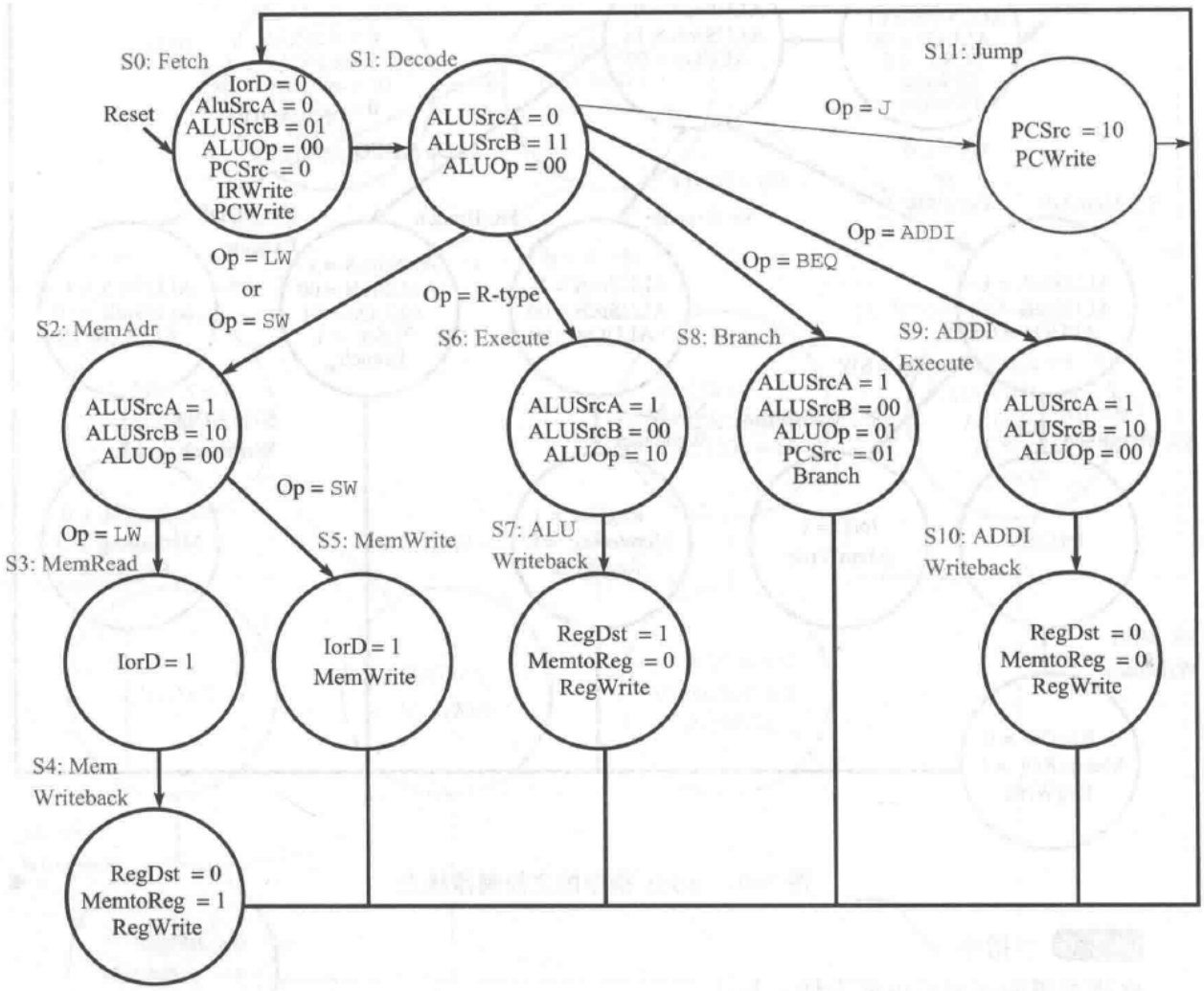

Figure 3: 控制器状态转移图

```
    // AluCtl 和 ExtOp 的在 R-Type/I-Type Execute 阶段的特殊控制
    always @(*)
        if (state == REX || state == IEX)
            {AluCtl, ExtOp} <= {AluOp1, ExtOp1};
        else
            {AluCtl, ExtOp} <= {AluOp0, ExtOp0};
```

Listing 4: `ALUOp` 和 `ExtOp` 信号控制

#	Assembly	Description	Address	Machine
main:	addi $2, $0, 5	# initialize $2 = 5	0	20020005
	addi $3, $0, 12	# initialize $3 = 12	4	2003000c
	addi $7, $3, -9	# initialize $7 = 3	8	2067fff7
	or $4, $7, $2	# $4 = (3 OR 5) = 7	c	00e22025
	and $5, $3, $4	# $5 = (12 AND 7) = 4	10	00642824
	add $5, $5, $4	# $5 = 4 + 7 = 11	14	00a42820
	beq $5, $7, end	# shouldn't be taken	18	10a7000a
	slt $4, $3, $4	# $4 = 12 < 7 = 0	1c	0064202a
	beq $4, $0, around	# should be taken	20	10800001
	addi $5, $0, 0	# shouldn't happen	24	20050000
around:	slt $4, $7, $2	# $4 = 3 < 5 = 1	28	00e2202a
	add $7, $4, $5	# $7 = 1 + 11 = 12	2c	00853820
	sub $7, $7, $2	# $7 = 12 - 5 = 7	30	00e23822
	sw $7, 68($3)	# [80] = 7	34	ac670044
	lw $2, 80($0)	# $2 = [80] = 7	38	8c020050
	j end	# should be taken	3c	08000011
	addi $2, $0, 1	# shouldn't happen	40	20020001
end:	sw $2, 84($0)	# write mem[84] = 7	44	ac020054

Figure 4: 测试代码

Figure 5: 状态转移未优化 左图

Figure 6: 状态转移未优化 右图

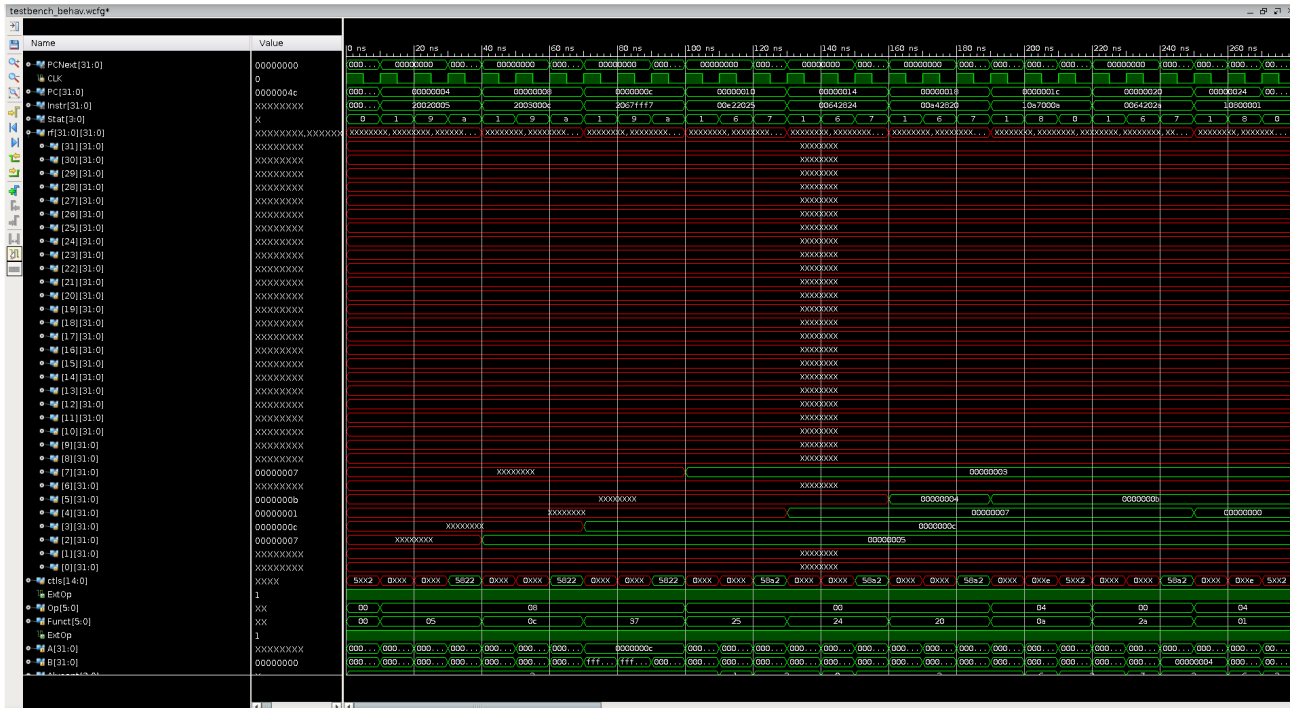

Figure 7: 状态转移已优化 左图

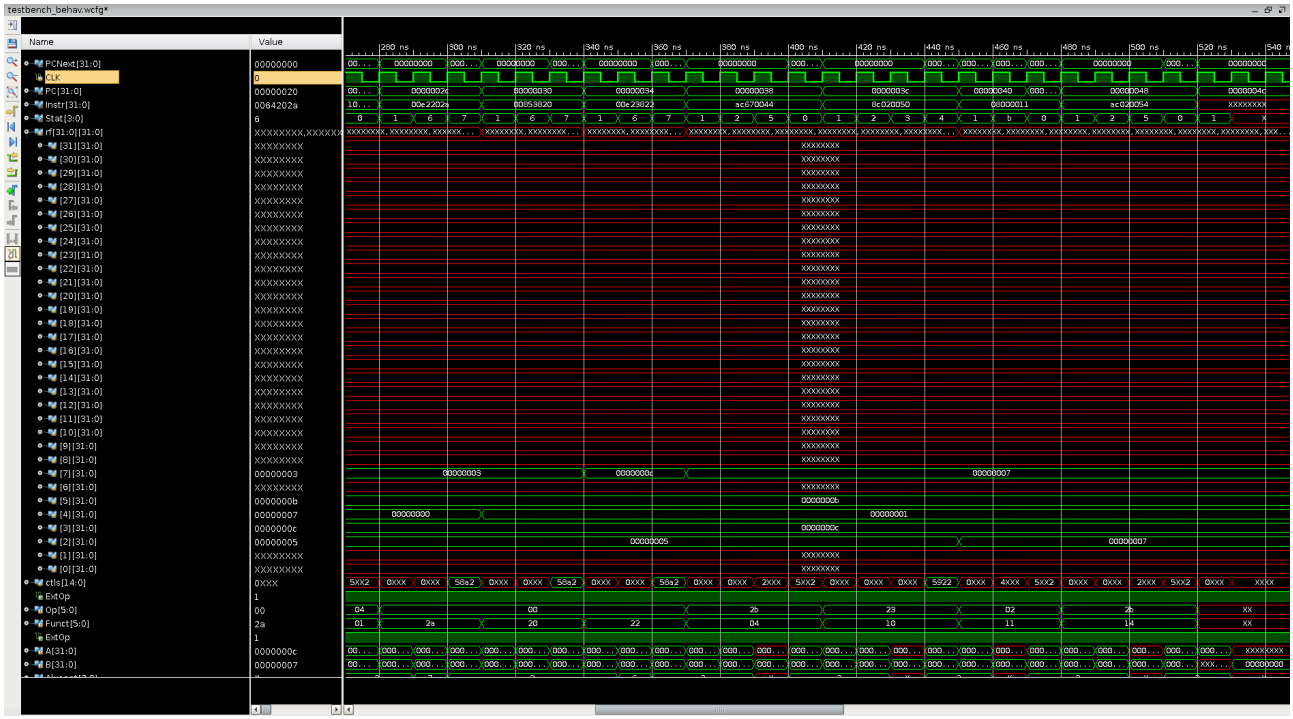

Figure 8: 状态转移已优化 右图