****

**《计算机组成原理实验》**

**实验报告**

**（实验二）**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **学院名称** | **：** | 数据科学与计算机学院 | | | | | |
| **专业（班级）** | **：** |  | | | | | |
| **学生姓名** | **：** |  | | | | | |
| **学号** | **：** |  | | | | | |
| **时间** | **：** | 2018 | 年 | 11 | 月 | 17 | 日 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  | **成绩** | **:** |  |
| **实验二** | **：** | **单周期 CPU 设计与实现** | | | |

* + - 1. **实验目的**
  1. 掌握单周期 CPU 数据通路图的构成、原理及其设计方法；
  2. 掌握单周期 CPU 的实现方法，代码实现方法；
  3. 认识和掌握指令与 CPU 的关系；
  4. 掌握测试单周期 CPU 的方法。
     + 1. **实验内容**

设计一个单周期CPU，该CPU至少能实现所述的 16 条指令功能操作。

* + - 1. **实验原理**

简单的 MIPS CPU 由 5 个阶段组成：

**取指令**

**IF**

**指令译码**

**ID**

**指令执行**

**EXE**

**存储器访问**

**MEM**

**结果写回**

**WB**

图 1：MIPS CPU 的五个阶段，循环执行

下面我将为每个阶段编写模块并阐述其功能、实现方法以及与其他模块的连接。

**IF 阶段模块：**

**PC 寄存器模块**负责维护 PC 寄存器的值，在时钟上升沿时更新 PC，允许写入 PC，或者让 PC 自增，还可以暂停 PC 的更新（比如在 halt 指令执行后，或者其他情况）。其端口表和连线方式如下表：

|  |  |  |
| --- | --- | --- |
| 端口类型 | 端口名 | 说明 |
| 输入端 | stall | 阻塞 PC 更新，由 halt 指令产生 |
| 输入端 | rw | 低电平为 PC 自增，高电平时为跳转指令写入 PC。与执行阶段产生的 take\_branch 信号直接相连。 |
| 输入端 | write | 要跳转到的地址。与执行阶段产生的 branch\_target 信号直接相连。 |
| 输出端 | pc | 当前 PC 寄存器的值 |
| 输出端 | next\_pc | PC 寄存器将要更新的值 |

表 1：PC 寄存器模块的端口表

实现这样的 PC 寄存器也很简单，下面是关键逻辑代码：

|  |
| --- |
| reg first\_n; // 复位后的第一次时钟不更新寄存器的值，原因在写板时会详细介绍  assign next\_pc = rw == `MEM\_WRITE ? write : pc + 4; // 下一条指令地址  always @(posedge clk, negedge rst\_n) begin // 在时钟的上升沿时更新 PC  if (!rst\_n) begin  pc <= 0; first\_n <= 0;  end else begin  if (!first\_n) first\_n <= 1;  if (!stall && first\_n) // 如果不是 halt 且不是第一次时钟脉冲，  pc <= next\_pc; // 则更新 pc 寄存器的值  end  end |

**一级指令缓存模块**负责加载 PC 寄存器指定地址的指令数据提供给译码器处理。实际上就是一个 ROM 的读取器。端口表如下：

|  |  |  |
| --- | --- | --- |
| 输入端 | addr[31:0] | 指令地址，直接与 PC 寄存器的的 PC 相连 |
| 输出端 | data[31:0] | 读出的指令数据，直接与译码器的 raw\_inst 端口相连 |

表 2：一级指令缓存模块端口表

下面是一级指令缓存模块的关键代码：

|  |
| --- |
| always @\* // 存储器的读取一定是无条件触发的  begin  data[31:24] <= dat[addr ]; // 因为存储器单元大小为一个字节  data[23:16] <= dat[addr + 1]; // 所以读出一条指令就需要读取  data[15: 8] <= dat[addr + 2]; // 连续 4 个字节的数据  data[ 7: 0] <= dat[addr + 3]; // 再拼接在一起  end |

**ID 阶段模块：**

**译码器模块**负责将要执行的指令分解成一系列控制信号，向之后阶段的模块提供运行指令。其端口表以及产生的控制信号表如下：

|  |  |  |
| --- | --- | --- |
| 端口名 | 低电平 | 高电平 |
| raw\_inst[31:0] | 输入的指令数据，直接与一级指令缓存的数据输出端 data 连接 | |
| rs\_addr[4:0] | 输出当前指令使用的 rs 寄存器编号，直接与寄存器组的 rs\_addr 相连 | |
| rt\_addr[4:0] | 输出当前指令使用的 rt 寄存器编号，经过 MUX 选择后与寄存器组的 rt\_addr 相连 | |
| rd\_addr[4:0] | 输出当前指令使用的 rd 寄存器编号，与寄存器组的 write\_addr 直接相连 | |
| imm[31:0] | 经过符号/零扩展的立即数或者 shift amount | |
| alu\_op[3:0] | 指令的 ALU 操作码，参见 ALU 操作码表 | |
| b\_ctrl | 指令的第二个操作数来自 rt | 指令的第二个操作数来自立即数 |
| mem\_enable | 指令不操作存储器 | 指令操作存储器：lw、sw |
| wb\_reg | 指令不写回寄存器：sw、beq、bne、bltz、j、halt | 指令写回寄存器：add、sub、addiu、andi、and、ori、or、sll、slti、lw |
| wb\_src | 写回寄存器组的数据来自 ALU | 写回寄存器组的数据来自存储器 |
| jump | 其他指令 | 无条件跳转指令：j |
| branch | 其他指令 | 跳转指令：beq、bne、bltz、j |
| halt | 其他指令 | 会导致停机的指令：halt |

表 3：译码器端口表及控制信号表

为了方便实现控制信号的输出，我们列出每个控制信号的真值与指令之间的关系。注意到部分指令没有立即数，因此其符号扩展信号 ext 的值可以为任意值，我们以斜杠表示信号的值为任意值。同样可以为任意值的信号还有 wb\_src，如果 wb\_reg 为低电平，也就是相应的指令不需要写回寄存器，那么写回寄存器的数据来源是什么不会影响操作逻辑，因此这部分指令的 wb\_src 信号也为任意值。最后，部分不需要使用 ALU 的指令的 alu\_op 信号也为任意值，我们一般给这些指令的 alu\_op 设置为加法，alu\_op 的值与操作的关系参见 ALU 操作码表。最后对 ext 进行补充，操作数二除了可能来自寄存器和立即数，还可能来自 shamt，因此我们可以为 ext 新增一个值 2 表示操作数二来自 shamt 的无符号扩展结果（我们要实现的指令中的 shamt 都是无符号扩展的）。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | wb\_reg | ext | alu\_op | b\_ctrl | mem\_en | wb\_src | jump | branch | halt |
| add | 1 | \ | 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| sub | 1 | \ | 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| addiu | 1 | 1 | 4 | 1 | 0 | 0 | 0 | 0 | 0 |
| andi | 1 | 0 | 6 | 1 | 0 | 0 | 0 | 0 | 0 |
| and | 1 | \ | 6 | 0 | 0 | 0 | 0 | 0 | 0 |
| ori | 1 | 0 | 7 | 1 | 0 | 0 | 0 | 0 | 0 |
| or | 1 | \ | 7 | 0 | 0 | 0 | 0 | 0 | 0 |
| sll | 1 | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| slti | 1 | 1 | 12 | 1 | 0 | 0 | 0 | 0 | 0 |
| sw | 0 | 1 | 4 | 1 | 1 | \ | 0 | 0 | 0 |
| lw | 1 | 1 | 4 | 1 | 1 | 1 | 0 | 0 | 0 |
| beq | 0 | 1 | 13 | 0 | 0 | \ | 0 | 1 | 0 |
| bne | 0 | 1 | 14 | 0 | 0 | \ | 0 | 1 | 0 |
| bltz | 0 | 1 | 12 | 0 | 0 | \ | 0 | 1 | 0 |
| j | 0 | \ | \ | 0 | 0 | \ | 1 | 1 | 0 |
| halt | 0 | \ | \ | 0 | 0 | \ | 0 | 0 | 1 |

表 4：控制信号真值表

得到了控制信号的真值表后我们就可以设计译码器了。事实上我们只需要借助 Verilog 的 case 语句就可以轻松实现该真值表。关键代码如下：

|  |
| --- |
| always @\* // 我们在 ID 阶段便将立即数的扩展后的值计算出来，这个可以和寄存器读的操作同时进行  if (sign\_extend == `SIGN\_EXT) // $signed 和 $unsigned 便能实现符号  imm <= $signed(raw\_inst[`RAW\_INST\_IMM\_BUS]); // 和无符号扩展  else if (sign\_extend == `ZERO\_EXT) // raw\_inst 这里将立即数读出  imm <= $unsigned(raw\_inst[`RAW\_INST\_IMM\_BUS]);  else // 同时立即数还可能来自指令的 shamt 域的无符号扩展后的结果  imm <= shamt\_wire;  `define decode(rs\_wire, rt\_wire, rd\_wire, wb\_reg\_wire, inst\_wire, exec\_op\_wire, sign\_ext, b\_ctrl\_wire, wb\_src\_wire) \  rs\_addr <= rs\_wire; \ // decode 宏表示设置当前指令的控制信号。  rt\_addr <= rt\_wire; \ // 由于我们采用 case 语句来枚举所有可能情况，  rd\_addr <= rd\_wire; \ // 这会导致每一个 case 的重复代码很多，  wb\_reg <= wb\_reg\_wire; \ // 因此通过 decode 宏来解决大量的重复代码。  inst <= inst\_wire; \ // 综合器会将巨大的 case 语句的所有情况，  exec\_op <= exec\_op\_wire; \ // 全部使用多路选择器实现，我们只需要  sign\_extend <= sign\_ext; \ // 将 case 语句当做真值表实现即可  b\_ctrl <= b\_ctrl\_wire; \  wb\_src <= wb\_src\_wire; \  illegal <= 0  always @\*  begin  case (op\_wire) // 通过 case 语句来枚举 opcode 的所有可能情况  6'b000000: begin // add  `decode(rs\_wire, rt\_wire, rd\_wire, `REG\_WB, `INST\_ADD, `ALU\_OP\_ADD, `ZERO\_EXT, `B\_REG, `WB\_ALU); // 为 add 指令设置控制信号  end  …  endcase  end  assign mem\_enable = inst == `INST\_LW || inst == `INST\_SW ? `MEM\_EN : `MEM\_DIS;  assign jump = `INST\_J ? `JUMP : `JUMP\_N;  assign branch = (inst == `INST\_J || inst == `INST\_BLTZ || inst == `INST\_BEQ || inst == `INST\_BNE) ? `BR : `BR\_N;  assign halt = inst == `INST\_HALT; |

寄存器组模块用于读取和写入寄存器。其端口表如下：

|  |  |  |
| --- | --- | --- |
| 类型 | 端口名 | 说明 |
| 输入 | rs\_addr[4:0] | 操作数一寄存器的地址，与译码器的 rs\_addr 直接相连 |
| 输出 | rs\_data[31:0] | 操作数一寄存器的数据，与 ALU 的 rs 端口直接相连 |
| 输入 | rt\_addr[4:0] | 操作数二寄存器的地址，与译码器的 rt\_addr 直接相连 |
| 输出 | rt\_data[31:0] | 操作数二寄存器数据，与 ALU 的 rt 端口经过 MUX 相连 |
| 输入 | rw | 目前状态是否需要保存存储器，和译码器的 RegWre 信号直接相连 |
| 输入 | write\_addr[4:0] | 要写入寄存器的地址，与解码器的 write\_addr 直接相连 |
| 输入 | write\_data[31:0] | 要写入寄存器的数据，若回写的是 ALU 的数据，则 MUX 选择 ALU 的结果；若回写的是内存数据，则 MUX 选择内存控制器的数据 |

表 5：寄存器组模块的端口表

根据上述信息，我们可以轻易地做出来一个寄存器组模块：

|  |
| --- |
| assign rs\_data = (rs\_addr == 0) ? 0 : file[rs\_addr];  assign rt\_data = (rt\_addr == 0) ? 0 : file[rt\_addr];  assign write\_read = (write\_addr == 0) ? 0 : file[write\_addr];  wire write\_enabled = rw == `MEM\_WRITE && (write\_addr != 0) && !stall\_in;  always @(negedge clk, negedge rst\_n) begin  if (!rst\_n) begin  for (i = 1; i < `REG\_SIZE; i = i + 1)  file[i] <= 0;  end else begin  // perform write  if (write\_enabled) file[write\_addr] <= write\_data;  end  end |

很显然实现结果如上所示，在回写寄存器时寄存器的值发生了改变。

**执行模块：**

**ALU** 负责进行计算算术逻辑指令的结果、计算内存操作指令要操作的内存地址、判断跳转指令是否需要跳转等操作，我们为了方便 ALU 的实现，将许多指令的计算操作提炼成同一个 ALU 的操作码信号，这样 ALU 的操作码信号就不需要和指令条数一样多了。ALU 的操作信号表如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 代号 | 操作码 | 语句 | 说明 |
| SLL | 1 | rd = rs << rt | 左移 |
| ADD | 4 | rd = rs + rt | 加法 |
| SUB | 5 | rd = rs – rt | 减法 |
| AND | 6 | rd = rs & rt | 按位与 |
| OR | 7 | rd = rs | rt | 按位或 |
| LT | 12 | rd = rs < rt ? 1 : 0 | 小于 |
| EQ | 13 | rd = rs == rt ? 1 : 0 | 等于 |
| NE | 14 | rd = rs != rt ? 1 : 0 | 不等于 |

表 6：ALU 操作码表

ALU 的端口表如下：

|  |  |  |
| --- | --- | --- |
| 输入端 | op[3:0] | ALU 操作码 |
| 输入端 | rs[31:0] | ALU 的第一个操作数的值 |
| 输入端 | rt[31:0] | ALU 的第二个操作数的值 |
| 输出端 | rd[31:0] | ALU 的运算结果 |

表 7：ALU 的端口表

ALU 的实现也很简单，也是一些 case 语句，需要确保变量在所有情况下都有赋值以避免产生锁存器：

|  |
| --- |
| always @\* begin  case(op) // 检查 ALU 操作码  `ALU\_OP\_SLL: begin // 如果是 SLL 指令的操作码，执行左移运算  rd = rs << rt;  end  …  endcase  end |

我们在 EX 阶段中，ALU 计算跳转指令的 zero 信号是高还是低时，我们同时还要计算跳转的目标地址。因此 **EX 阶段执行模块**的关键代码如下：

|  |
| --- |
| always @\* begin  branch\_target <= 0;  if (branch) begin  if (jump) begin // 如果是无条件跳转指令 j  if (inst == `INST\_J)  branch\_target <= { pc4[DATA\_WIDTH-1:28], raw\_inst[`RAW\_INST\_ADDR\_BUS], 2'b00 };  else // 我们不支持 J 指令以外的无条件跳转指令  branch\_target <= 0;  end else begin // 条件跳转语句，bt <= pc + 4 + 4 \* imm  branch\_target <= pc4 + {imm[DATA\_WIDTH-3:0], 2'b00};  end  end  end |

**访存阶段：**

一级数据缓存表示内存，用于存储数据。

信号表如下：

|  |  |  |
| --- | --- | --- |
| 输入端 | addr[31:0] | 访存的内存胆原路径 |
| 输入端 | enable | 是否启用存储器 |
| 输入端 | rw | 低电平时为读模式，高电平下为做饭模式 |
| 输入端 | write[31:0] | 我们要写入数据的寄存器编号 |
| 输出端 | read[31:0] | 从寄存器中读取的数据是什么 |

表 8：一级数据缓存模块的信号表

下面是一级数据缓存的关键代码：

|  |
| --- |
| always @\* begin // 存储器读取数据的时候一定是要无条件触发  if (enable) // 因为下降沿触发就会和寄存器写入冲突了，存储器写入的不是  begin // 当前已经写入的数据，而是上条指令的数据  read[31:24] <= dat[addr ]; // 存储器的单元大小为一个字节，而  read[23:16] <= dat[addr + 1]; // 存储器中一次性读取 4 个字的数据  read[15: 8] <= dat[addr + 2];  read[ 7: 0] <= dat[addr + 3];  end else read <= 'bZZZZZZZZ; // 内存禁用时读取的数据为 'b ZZZZZZZZ 从房间中带出  end    always @(negedge clk, negedge rst\_n) begin  if (!rst\_n) begin  for (i = 0; i < 128; i = i + 1)  dat[i] <= 0;  end else begin  if (enable && rw == `MEM\_WRITE) begin // 下降沿写入  dat[addr ] <= write[31:24];  dat[addr + 1] <= write[23:16];  dat[addr + 2] <= write[15: 8];  dat[addr + 3] <= write[ 7: 0];  end  end  end |

**回写阶段：**回写阶段就是寄存器组的写入。我们特意在下降沿时进行存储器和寄存器组的写入就是为了分阶段。IF、ID、EX 阶段的执行都可以合并在一个阶段中，但 MEM 和 WB 阶段不可以，因为寄存器和存储器的写入都是要在时钟沿发生的。所以最后我们的单周期 CPU 实际上分成了两个阶段：IF/ID/EX/MEM只读 和 MEM写/WB。

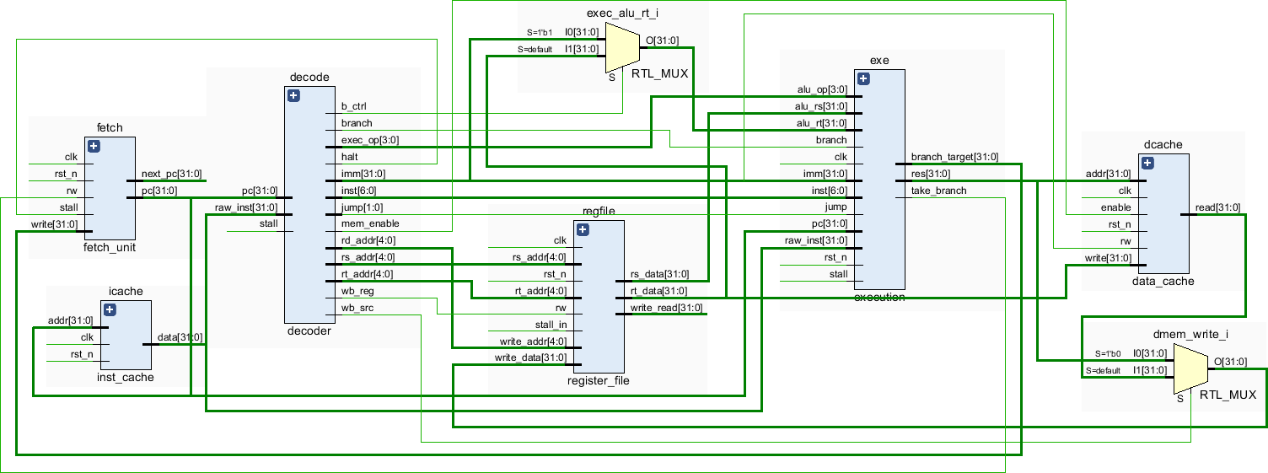
**数据通路图：**

数据通路图通过 Vivado

综合得到如右图 2 所示，图中省略了时钟和复位端口 clk 和 rst\_n 的连线。

* + - 1. **实验器材**

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

****

* + - 1. **实验过程与结果**

1. **设计**

CPU 设计的思想便是分阶段模块化实现。设计方法是：首先 5 个阶段可以分成 5 个大模块；每个阶段内部的组件，再分成一些小模块。这样 CPU 的实现就是模块化的实现。最后我们在顶层模块中将各个阶段的的大模块按照右侧的数据通路图连接在一起就可以了。设计时需要考虑到时间的问题，时钟沿触发器读取数据的时机问题。这样我们以分阶段的方式实现 CPU 就可以方便我们日后设计多周期 CPU，只需要简单地修改部分无条件的动作为时钟上升沿触发的动作即可。

1. **验证**

在完成了 CPU 的模块代码编写之后，我们需要通过仿真查看时序图来检验我们的 CPU 是否能按照标准正常无误地执行我们提供的测试指令流。

先介绍时序图中每个信号的用途和意义：

|  |  |  |  |
| --- | --- | --- | --- |
| 信号 | 用途 | 信号 | 用途 |
| inst | 指令的十六进制码 | rs\_addr | 操作数一寄存器编号 |
| pc | 当前指令的地址 | rs\_data | 操作数一寄存器值 |
| next\_pc | 下一条指令的地址 | rt\_addr | 操作数二寄存器编号 |
| alu\_res | ALU 运算结果 | rt\_data | 操作数二寄存器值 |
| write\_data | 要写入寄存器的值 | imm | 扩展过的立即数 |
| write\_addr | 要写入的寄存器编号 | br\_tgt | 跳转指令的目标地址 |
| write\_en | 是否要写入寄存器 | mem\_read | 内存读取的值 |
| alu\_op | ALU 操作码 | jump | 是否是无条件跳转指令 |
| zero | ALU 的值是否为零 | branch | 是否是跳转指令 |
| b\_ctrl | 操作数二来自寄存器或立即数 | fetch\_rw | 下一条指令地址来自 pc+4 还是 br\_tgt |
| take\_branch | 是否跳转 | clk/rst\_n | 时钟信号/复位信号 |

表 9：时序图信号表

需要补充强调的是，我在波形图中所提供的 rs(\_data/\_addr) 和 rt(\_data/\_addr) 两个信号是相对于 ALU 而言的而不是指令本身，也就是说 rs 和 rt 表示的是 ALU 的两个操作数，而不是指令的 rs 和 rt 寄存器。如果当前的指令是 I 型指令，那么此时 ALU 的第二个操作数来自扩展过的立即数而不是 rt。对于这些指令，波形图中可以看到其 rt\_addr 和 rt\_data 均为零而不是 inst[20:16] 的值，表示 ALU 的第二个操作数不来自寄存器。

首先对于前 5 条指令，它们的波形图如下图：



图 3：第 1~5 条指令的波形图

上图中执行的 pc=0x0~0x10 的这五条指令分别是：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| pc | 指令 | 操作 | alu\_op | b\_ctrl |
| 00000000 | addiu $1,$0,8 | $1=$0+8=8 | ADD(4) | 来自立即数(1) |
| 00000004 | ori $2,$0,2 | $2=$0+2=2 | OR(7) | 来自立即数(1) |
| 00000008 | add $3,$2,$1 | $3=$2+$1=10 | ADD(4) | 来自寄存器(0) |
| 0000000C | sub $5,$3,$2 | $5=$3-$2=8 | SUB(5) | 来自寄存器(0) |
| 00000010 | and $4,$5,$2 | $4=$5 & $2=0 | AND(6) | 来自寄存器(0) |

表 10：图对应执行的指令信息

从上图和表中可见 ALU 的第二个操作数选择信号 b\_ctrl 的值是正确的，而且 ALU正的第二个操作数的值也是正确地选择了 rt 寄存器的值或者是立即数。且查看 imm 的值发现立即数扩展是正确的，addiu 是符号扩展，ori 是零扩展。

下图是执行次序中的第 6~10 条指令，地址范围在 00000014~0000001c 之间。

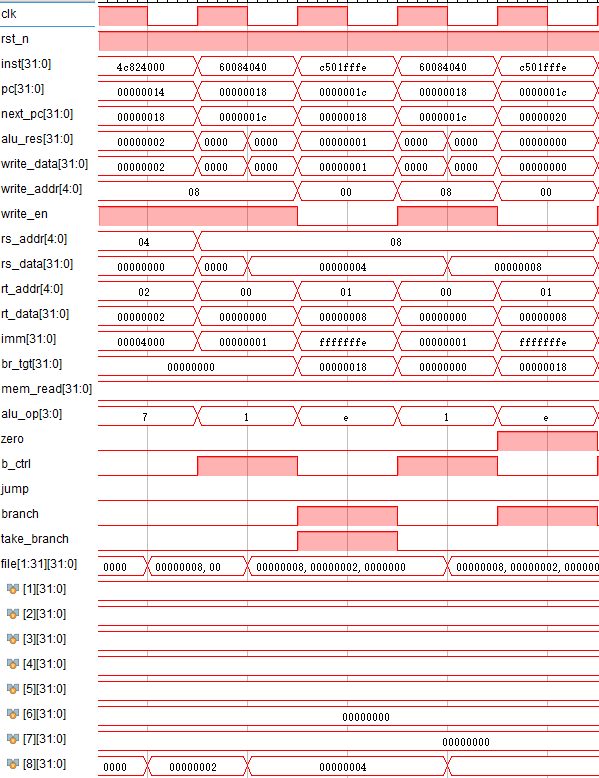


图 4：第 6~10 条指令的时序图

下表是上图中执行的 5 条指令的一些关键信息（br 指 branch 信号，take\_br 指 take\_branch 信号，其中跳转指令的回写寄存器信号 write\_en 为低电平）：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| PC | 指令 | 操作 | br | take\_br | write\_en | rs\_data |
| 00000014 | or $8,$4,$2 | $8=$4 | $2=2 | 0 | 0 | 1 | 0 |
| 00000018 | sll $8,$8,1 | $8=$8<<1=4 | 0 | 0 | 1 | 2->4 |
| 0000001C | bne $8,$1,-2 | $8 != $1 | 1 | 1 | 0 | 4 |
| 00000018 | sll $8,$8,1 | $8=$8<<1=8 | 0 | 0 | 1 | 4->8 |
| 0000001C | bne $8,$1,-2 | $8==$1 | 1 | 0 | 0 | 8 |

表 11： 第 6~10 条指令的执行信息表

可以看到，sll 指令执行完成后期 ALU 的运算结果是 8，而正确的运算结果应该是 4。这代表了我们的 CPU 运算出错了吗？答案是否定的。sll 指令在执行的时候其 ALU 的运算结果、要写入寄存器的结果等信号在时钟的下降沿发生了改变，这是因为我们现在执行的这个 sll 执行的 rt 和 rd 寄存器是同一个寄存器。也就是说当下降沿触发了寄存器回写之后，rs\_data 的值就发生改变（因为 $8 已经变成了新的值），那么此时 ALU 的运算结果自然跟着变化（4->8），相当于执行了两次这条指令。但是由于寄存器回写已经完成，所以下降沿后的 ALU 的运算结果是不会被使用到的（或者说不会被写入寄存器），那么对后续的指令的正确执行不会造成影响。我们可以从这些指令的 rs\_data 信号中一窥究竟：第一次执行 00000018 的 sll 指令时 rs 寄存器（$8）的值为 2，经过下降沿后可以从图中看到，rs\_data 的值变成了 4，从图中最下方的寄存器值来看，8 号寄存器的值也在下降沿时发生了改变（2->4），而没有变成 8。证明了下降沿之后的运算结果不会对程序执行的正确性造成影响。

其次我们再来看条件跳转指令。第一次的 bne 的结果是 $8 != $1，所以 ALU 的运算结果为 1，zero 信号即为 0。所以此时 take\_branch 信号，也就是跳转信号，为高电平，下一次时钟上升沿时 PC 寄存器就会跳转到跳转指令计算得到的目标地址 br\_tgt 处，从图中看到是 00000018，所以执行 bne 时的 next\_pc 信号为 00000018 而不是 00000020。而第二次的 bne 的运算结果是 $8 != $1 为假，所以 ALU 的运算结果是 0，zero 信号即为 1。此时不进行跳转，take\_branch 信号为低电平，next\_pc 为 pc+4，也就是下一条指令的地址 00000020。

**下图是第 11~15 条指令的执行时序图**，地址范围是 00000020~0000002C。

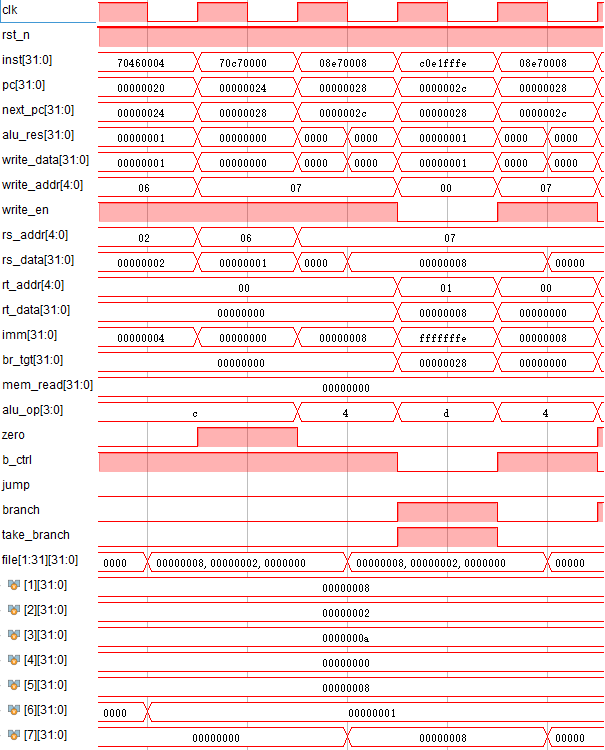


图 5：第 11~15 条指令的执行时序图

下表是上图中执行的 5 条指令的关键信息：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| PC | 指令 | 操作 | alu\_op |  |  |
| 00000020 | slti $6,$2,4 | $6=$2<4=1 | LT(12) |  |  |
| 00000024 | slti $7,$6,0 | $7=$6<0=0 | LT(12) |  |  |
| 00000028 | addiu $7,$7,8 | $7=$7+8=8 | ADD(4) |  |  |
| 0000002C | beq $7,$1,-2 | $7==$1=1 | EQ(13) |  |  |
| 00000028 | addiu $7,$7,8 | $7=$7+8=16 | ADD(4) |  |  |

表 12：第 11~15 条指令的信息表

这 5 条指令主要是运算类指令，可以看到 write\_addr 和 write\_data 和表中指令的操作一致。其中 addiu 指令的源寄存器和目的寄存器的编号一致，因此在时序图中下降沿时寄存器的值发生了改变，而且 alu\_res 和 write\_data 也跟着改变（相当于执行两次指令）。

**下图是第 16~20 条指令的时序图**，地址范围是 0000002C~0000003C。

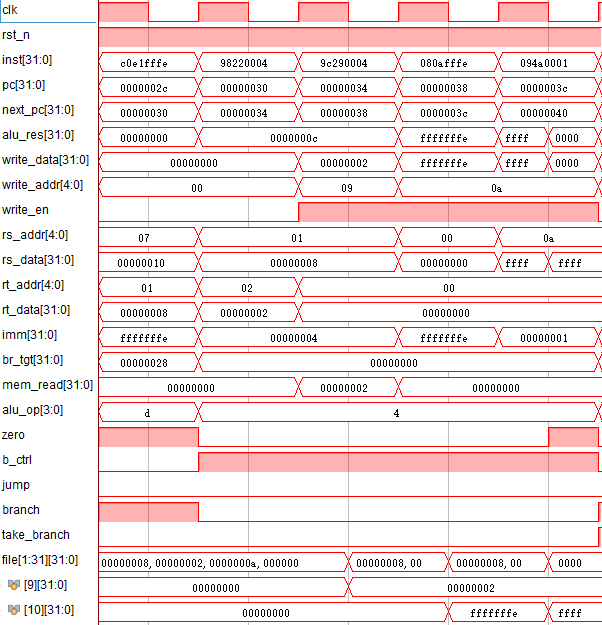


图 6：第 16~20 条指令的时序图

下表 13 为第 16~20 条指令的关键信息表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| PC | 指令 | 操作 | alu\_op |  |
| 0000002C | beq $7,$1,-2 | $7==$1 为假 | EQ(13) |  |
| 00000030 | sw $2,4($1) | $2->mem[$1+4=12] | ADD(4) |  |
| 00000034 | lw $9,4($1) | $9<-mem[$1+4=12]=2 | ADD(4) |  |
| 00000038 | addiu $10,$0,-2 | $10=$0-2=-2 | ADD(4) |  |
| 0000003C | addiu $10,$10,1 | $10=$10+1=-1 | ADD(4) |  |

表 13：第 16~20 条指令的操作信息表

这 5 条指令中包含操作存储器的 sw 和 lw 指令。首先对于这两条指令，我们需要预先处理出要操作的内存地址，也就是 $1+4=8+4=12，这是一个加法操作，因此我们令 ALU 的操作码为 ADD，操作数为 rs 寄存器和立即数，这样 ALU 的运算结果便是我们要操作的内存地址。

然后对于 sw 指令，我们在时钟的下降沿时存储器将数据（也就是 rt 寄存器的值）写入存储器。首先，存储器的写入一定要在时钟沿时发生：我们在之前的指令中可以看到，对于源操作数和目的操作数相同的指令来说，一旦将结果写入存储器，那么 ALU 等运算器的值会更新成相当于两次执行指令的结果。如果我们写入寄存器组是不通过时钟的下降沿，而是上升沿，是有问题的。因为对于单周期 CPU 来说，我们在时钟的上升沿更新 PC 寄存器，那么如果我们在时钟的上升沿写入寄存器，那么写入的时机就是下一条指令执行的时间，这样下一条指令要读取寄存器，就必须无条件地读取，而不能在时钟的上升沿时读取，否则就会读取到寄存器的上一次的值而不是当前的值。

对于 lw 指令，如果我们在时钟的下降沿写入寄存器，那么我们就不可以在时钟的下降沿时读取存储器，否则我们写入寄存器的值就不是我们当前从存储器中读取到的数据。但是我们又不可以在时钟的上升沿时读取存储器，因为我们在时钟的上升沿更新 PC 寄存器，因此我们在时钟的上升沿后得出我们要读取的存储器单元的地址。那么如果我们在上升沿时读取存储器，那么地址就是上一条指令要操作的存储器地址，这也是不行的。因此我们要无条件地（以连线的方式在任意时刻，而不是在时钟沿时）将数据从存储器中读出。这样就不会造成时间上的问题。从图中可以看到在 lw 指令执行的时期，mem\_read 信号的值为 00000002，已经正确地将数据从存储器中读取出来了。同时下方的寄存器信号中也可以看到 9 号寄存器的值发生了正确的改变。

**下图是第 21~25 条指令的时序图**，地址范围是 00000040~00000048：

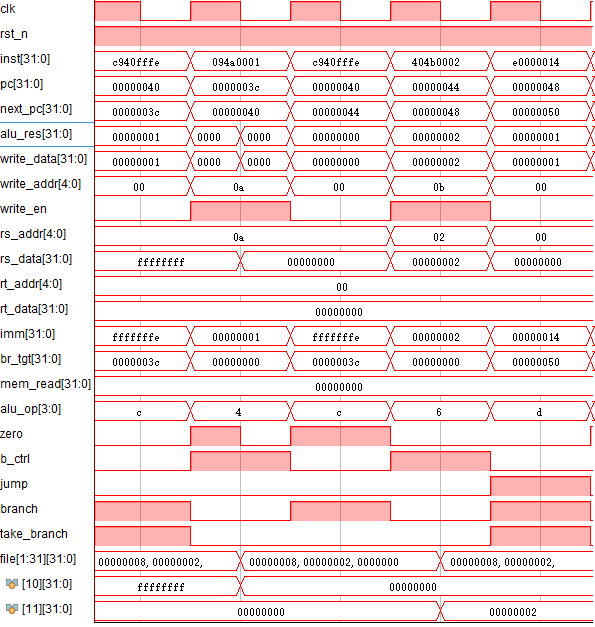


图 7：第 21~25 条指令的时序图

下表是第 21~25 条指令的关键信息表：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 地址 | 指令 | 操作 | alu\_op | br | br\_tgt | take\_br |
| 00000040 | bltz $10,-2 | $10<0 为真 | LT(12) | 1 | 0x3C | 1 |
| 0000003C | addiu $10,$10,1 | $10=$10+1=0 | ADD(4) | 0 | / | 0 |
| 00000040 | bltz $10,-2 | $10<0 为假 | LT(12) | 1 | 0x3C | 0 |
| 00000044 | andi $11,$2,2 | $11=$2 & 2=2 | AND(6) | 0 | / | 0 |
| 00000048 | j 0x00000050 | PC<-00000050 | 无操作 | 1 | 0x50 | 1 |

表 14：第 21~25 条指令的关键信息表

这里我的 bltz 指令的实现方式是将 bltz 指令当成 blt 指令执行，因为 bltz 指令的 rt 寄存器处的地址正好是 0，而 0 号寄存器的值就是我们要比较的另一个操作数 0。所以我们只要将 ALU 的操作码设置为小于比较，就可以直接实现 bltz 指令。

我们看到，指令中 bltz 和 j 的跳转标志 branch 信号均为高电平，此时寄存器回写信号为低电平。通过观察波形图我们可以看到第一次和第二次的 bltz 指令的跳转目的地址均为 0x0000003C，但是由于第二次 bltz 的进行跳转，因此直接执行下一条指令 andi。 j 指令的 branch\_target（br\_tgt）为 0x00000050，由于 j 指令是无条件跳转，因此其 take\_branch 信号即为高电平。

**下图是第 24~26 条指令的时序图**，指令地址范围是 00000044~00000050。

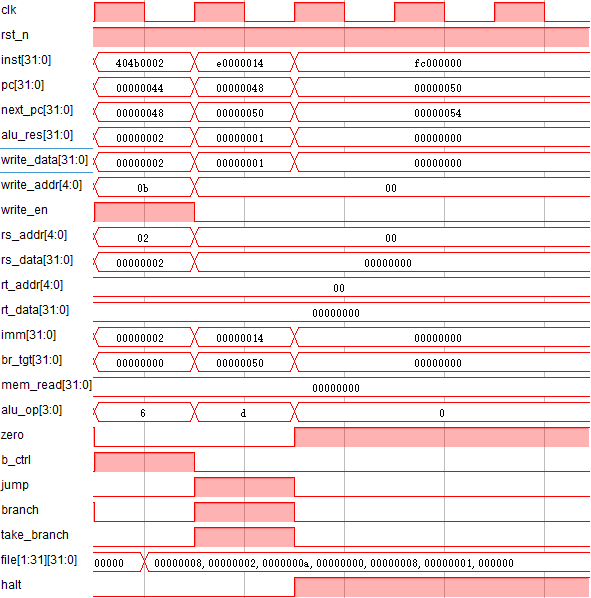


图 8：第 24~26 条指令的时序图

下表是第 24~26 条指令的关键信息表。从上图中我们看到，由于 j 指令是无条件跳转指令，因此 PC 寄存器即更新到 00000050，其 next\_pc 的值也为 00000050，说明跳转成功。也因此，位于地址 0000004C 的指令 or $8,$4,$2 被跳过，因此这条指令未被执行。接着我们便跳到了 halt 指令，此时程序终止信号 halt 变成高电平，而根据 PC 寄存器实现，halt 信号为高电平时 PC 不再更新，因此 CPU 的执行也被终止，直到下一次复位。

|  |  |  |  |
| --- | --- | --- | --- |
| 地址 | 指令 | 操作 | halt |
| 00000044 | andi $11,$2,2 | $11=$2 & 2 | 0 |
| 00000048 | j 0x00000050 | PC <- 00000050 | 0 |
| 00000050 | halt | 停止 PC 寄存器的更新 | 1 |

表 15：第 24~26 条指令的关键信息表

3. **实现**

那么如何验证我们的 CPU 是否能在 Basys3 板上正常运行呢？首先为了综合我们需要添加一个综合用的顶层模块，以便我们对 CPU 进行控制。我们要对 CPU 控制的有 CPU 的时钟以及其复位信号；其次我们要读取的 CPU 的状态必须能方便我们进行调试以便我们能确保我们的 CPU 是可以正确运行的。我们可以将仿真中所用到的信号都显示出来，比如一些只有两种状态的信号，我们就可以借助 LED 灯来显示，亮表示高电平，暗表示低电平；对于那些不止两种状态的信号、总线，我们可以借助数码管显示，鉴于数码管只有 4 位，能显示的内容不多，且考虑到我们的测试程序的数据都不大于 256，我们可以只显示这些数据的低 8 位，也就是模 256 之后的数据（如果数据不大于 256，那么显示的就是原数据）。但是，因为 CPU 的运行速度很快，因此我们会来不及看 CPU 的运行是否正确，因此我们不能让 CPU 自动运行（比如使用分频器），要使用按钮手动控制 CPU 的执行。手动控制时的按钮信号需要防抖。

调试的方法就很简单了，既然我们可以手动控制 CPU 的指令执行，我们就可以一条一条语句看 CPU 的各个信号输出是否符合预期，如果不符合预期再查找问题即可。

下面是为了实现上述要求而编写的综合用的顶层模块：

|  |
| --- |
| module main\_top(  input clk, // 系统时钟  input btn0, // 按钮的信号  output [6:0] seg, // 七段显示管的显示信号  output reg [3:0] an, // 位选信号  input [15:0] sw, // 16 个开关的信号  output reg [15:0] LED // 16 盏 LED 的控制端口  );    wire trigger; // 按钮信号防抖后的信号  debouncer deb(clk, btn0, trigger); // 将 btn0 防抖同步后输出 trigger    wire rst\_n = sw[0]; // 令 0 号开关提供复位信号  wire [2:0] mode = sw[15:13]; // 令 13~15 号开关提供    reg [3:0] S[0:3];  reg [3:0] BCD;  reg [1:0] display\_digit = 0;  reg [3:0] display\_digit\_bit;  wire clk\_display;    clock\_div#(50000, 19) div2(clk, rst\_n, clk\_display); // 分频器  bcd\_encoder encoder(1, BCD, seg); // BCD 码转七段显示管信号    main main(/\* CPU 的控制与输出端口表 \*/);    always @(posedge clk\_display)  display\_digit = display\_digit + 1;    always @\*  begin  case(display\_digit) // 枚举位选信号  0: an <= 4'b1110;  1: an <= 4'b1101;  2: an <= 4'b1011;  3: an <= 4'b0111;  default: an <= 4'b0000;  endcase  BCD <= S[display\_digit];  end    always @\* // LED 灯显示的信号表请参见下文  begin  LED[0] <= 1; // 常亮表示 CPU 是否在运行  LED[1] <= fetch\_rw; // PC 寄存器是 +4 还是被跳转指令覆盖  LED[2] <= zero; // ALU 的 zero 信号  LED[3] <= reg\_write\_en; // 是否回写寄存器  LED[4] <= b\_ctrl; // 使用寄存器还是立即数作为第二操作数  LED[5] <= take\_branch; // 是否跳转  LED[6] <= jump; // 是否是无条件跳转指令  LED[7] <= branch; // 是否是跳转指令  LED[15] <= halt; // 是否停止 CPU 的执行  end    always @\* // 数码管显示的信号表请参见下文  case(mode)  0: begin {S[3], S[2]} <= pc[7:0]; {S[1], S[0]} <= next\_pc[7:0]; end // 高位是当前指令的地址，低位是下一条指令的地址  1: begin S[3] <= rs\_addr / 10; S[2] <= rs\_addr % 10; {S[1], S[0]} <= rs\_data[7:0]; end // 高位是 rs 寄存器的编号，低位是其数据  2: begin S[3] <= rt\_addr / 10; S[2] <= rt\_addr % 10; {S[1], S[0]} <= rt\_data[7:0]; end // 高位是 rt 寄存器的编号，低位是其数据  3: begin {S[3], S[2]} <= alu\_res[7:0]; {S[1], S[0]} <= mem\_read[7:0]; end // 高位是 ALU 运算结果，低位是内存读数据的结果  4: begin S[3] <= reg\_write\_addr / 10; S[2] <= reg\_write\_addr % 10; {S[1], S[0]} <= reg\_write\_data[7:0]; end  5: begin {S[3], S[2]} <= imm[7:0]; {S[1], S[0]} <= branch\_target[7:0]; end // 高位是立即数，低位是跳转指令目标地址  6: begin S[3] <= alu\_op; {S[2], S[1], S[0]} <= alu\_res[11:0]; end // 高位是 ALU 操作码，低位是 ALU 运算结果  7: begin {S[3], S[2]} <= reg\_write\_read; {S[1], S[0]} <= 0; end // 高位是回写寄存器的值，检查是否成功回写寄存器  default: begin {S[3], S[2], S[1], S[0]} <= 0; end  endcase  endmodule |

为了方便调试，我将部分 CPU 的关键信息输出在七段显示管上以及 LED 灯上，七段显示管的显示表如下：

|  |  |  |
| --- | --- | --- |
| 开关 | 高两位 | 低两位 |
| 000 | 当前 PC 寄存器的值的低 8 位 | PC 寄存器将更新的值的低 8 位 |
| 001 | rs 寄存器的编号（十进制） | rs 寄存器的值的低 8 位 |
| 010 | rt 寄存器的编号（十进制） | rt 寄存器的值的低 8 位 |
| 011 | ALU 的运算结果的低 8 位 | 从内存中读取的数据的低 8 位 |
| 100 | 回写寄存器的编号（十进制） | 回写寄存器的数据的低 8 位 |
| 101 | 扩展后的立即数的低 8 位 | 跳转目标地址的低 8 位 |
| 110 | ALU 操作码 | ALU 运算结果的低 12 位 |
| 111 | 回写寄存器的值（用于检查数据是否真的写入寄存器） | |

表 16：七段显示管显示信息表

|  |  |  |  |
| --- | --- | --- | --- |
| LED编号 | 信号名 | 低电平 | 高电平 |
| 0 | 始终为高电平，用于标记当前 CPU 是否被写入 Basys3 学习板 | | |
| 1 | fetch\_rw | PC 寄存器 + 4 | PC 寄存器回写跳转地址 |
| 2 | zero | ALU 的结果不为 0 | ALU 的结果为 0 |
| 3 | RegWre | 不回写寄存器 | 回写寄存器 |
| 4 | b\_ctrl | ALU 的第二个操作数来自寄存器 | ALU 的第二个操作数来自扩展后的立即数 |
| 5 | take\_branch | 不进行跳转 | 进行跳转 |
| 6 | jump | 不是无条件跳转指令 | 是 j 指令 |
| 7 | branch | 不是跳转指令 | 是 bne、beq、bltz、j |
| 15 | halt | CPU 正常执行 | CPU 执行结束，终止 |

表 17：16 个 LED 灯的作用说明

接下来我展示第 20~25 条指令，一共 6 条指令的执行流程：

下图是第 20 条指令的执行信息，可以看到图和图的显示证明了其正确性：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| pc | next\_pc | 指令 | write\_addr | write\_data |
| 0000003C | 00000040 | addiu $10,$10,1 | $10 | FFFFFFFF(-1) |

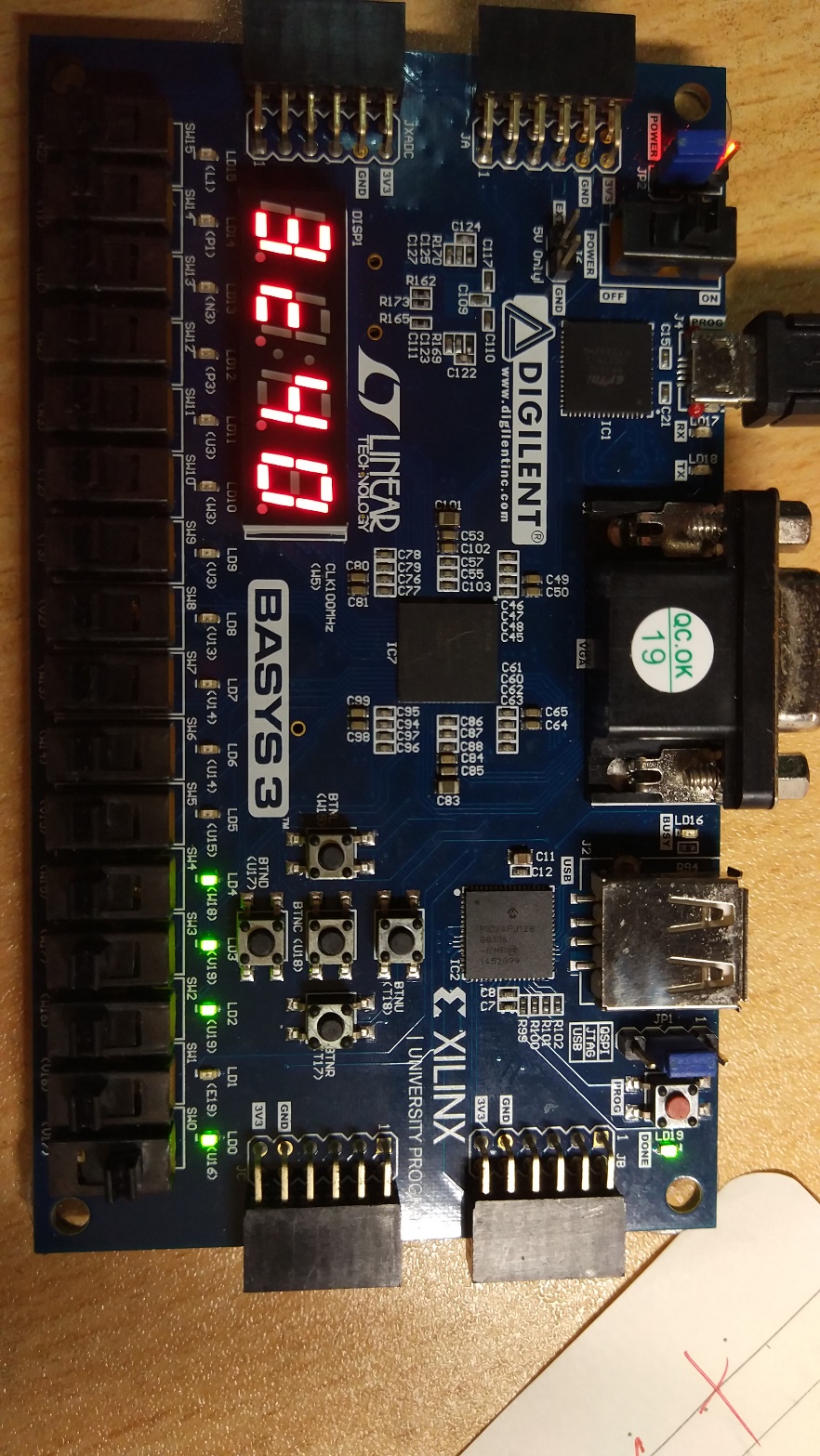


图 9：地址为 0000003C 的指令状态，可见 fetch\_rw=0、zero=1（源汇寄存器相同）、RegWre=1、bCtrl=1、take\_branch=0、jump=0、branch=0、halt=0

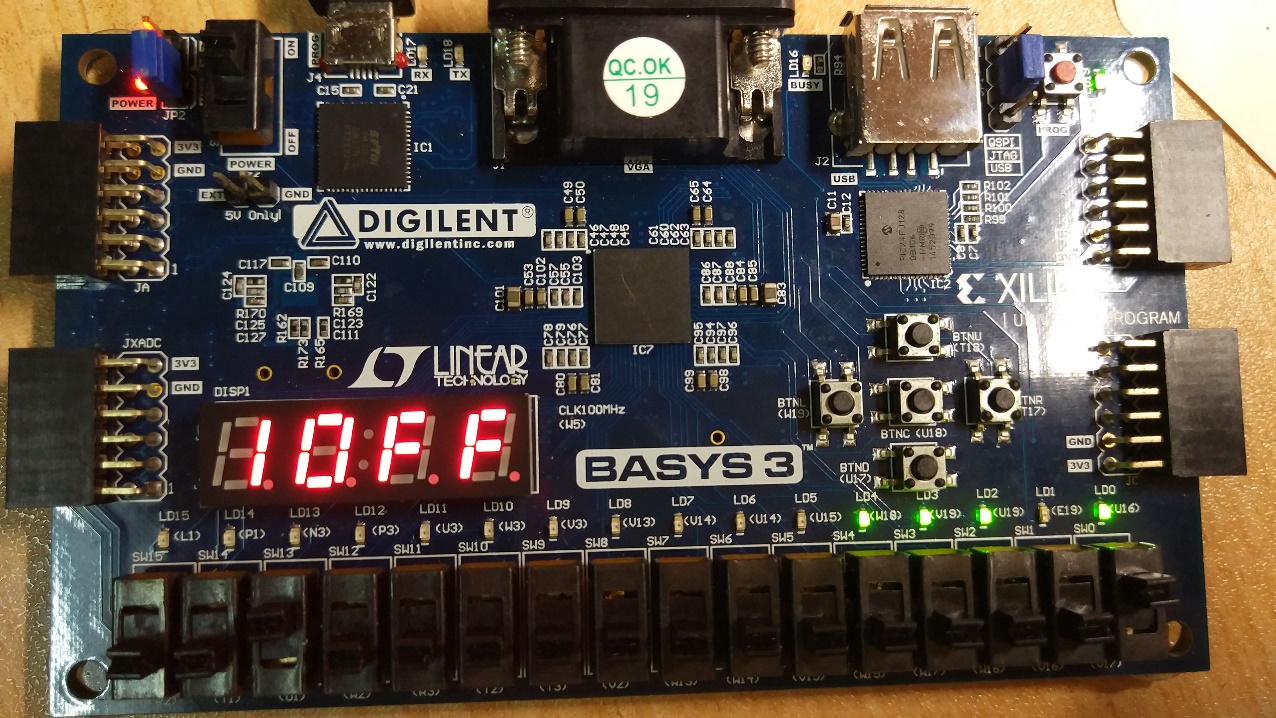


图 10：地址为 0000003C 的指令的 rs 寄存器编号以及现在 rs 寄存器的数据（因为源汇寄存器一致，因此其值即为该指令运行完成后的值），其中 SW12 调至高电平

下图是第 21 条指令的执行信息：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| pc | next\_pc | 指令 | 操作 | take\_branch |
| 00000040 | 0000003C | bltz $10,-2 | $10=-1<0 为真 | 1 |

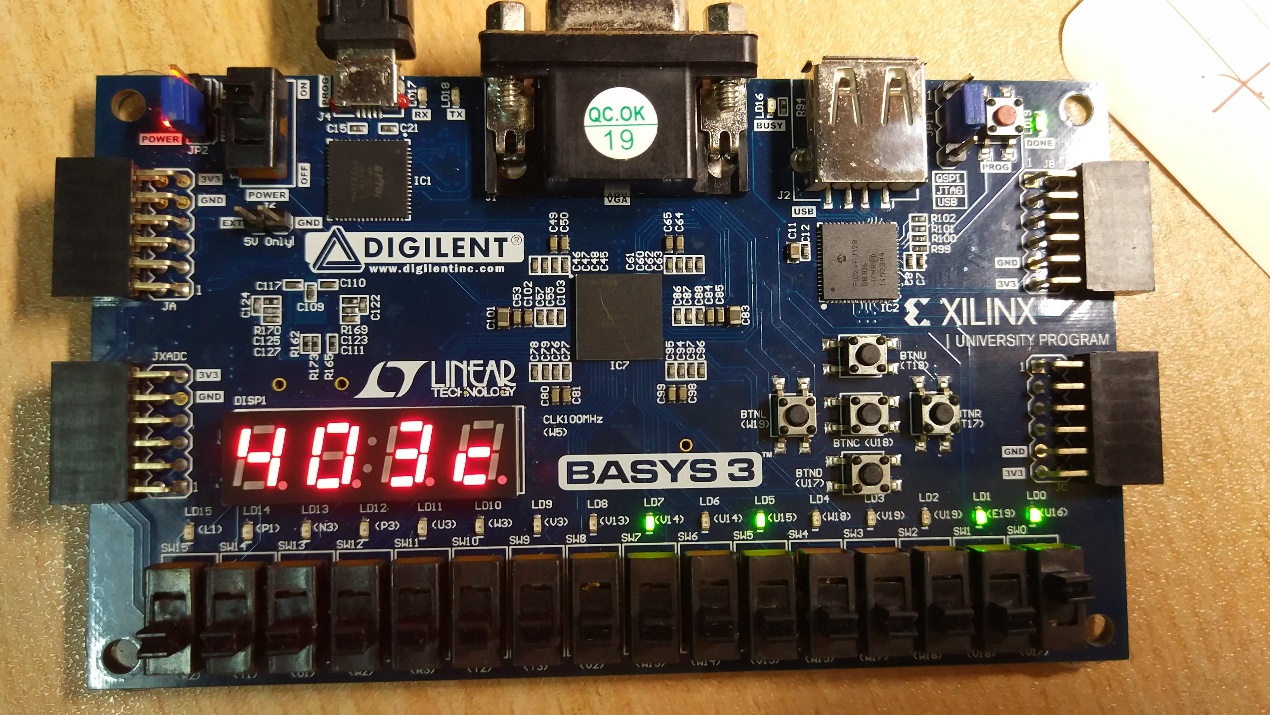


图 11：地址为 00000040 的指令发生了跳转，下一条要执行的指令的地址为 0000003C。从图中从右到左的信号分别为：始终亮的 LED 0，LED 1 表示的 fetch\_rw=1（发生跳转要覆盖 PC 寄存器的值）、zero=0（因为结果为真，发生跳转）、RegWre=0（跳转指令不需要回写寄存器）、BCtrl=0（两个操作数均为寄存器值）、TakeBranch=1、jump=0、branch=1、halt=0

下图是第 22 条指令的执行信息：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| pc | next\_pc | 指令 | write\_addr | write\_data |
| 0000003C | 00000040 | addiu $10,$10,1 | $10 | 0 |

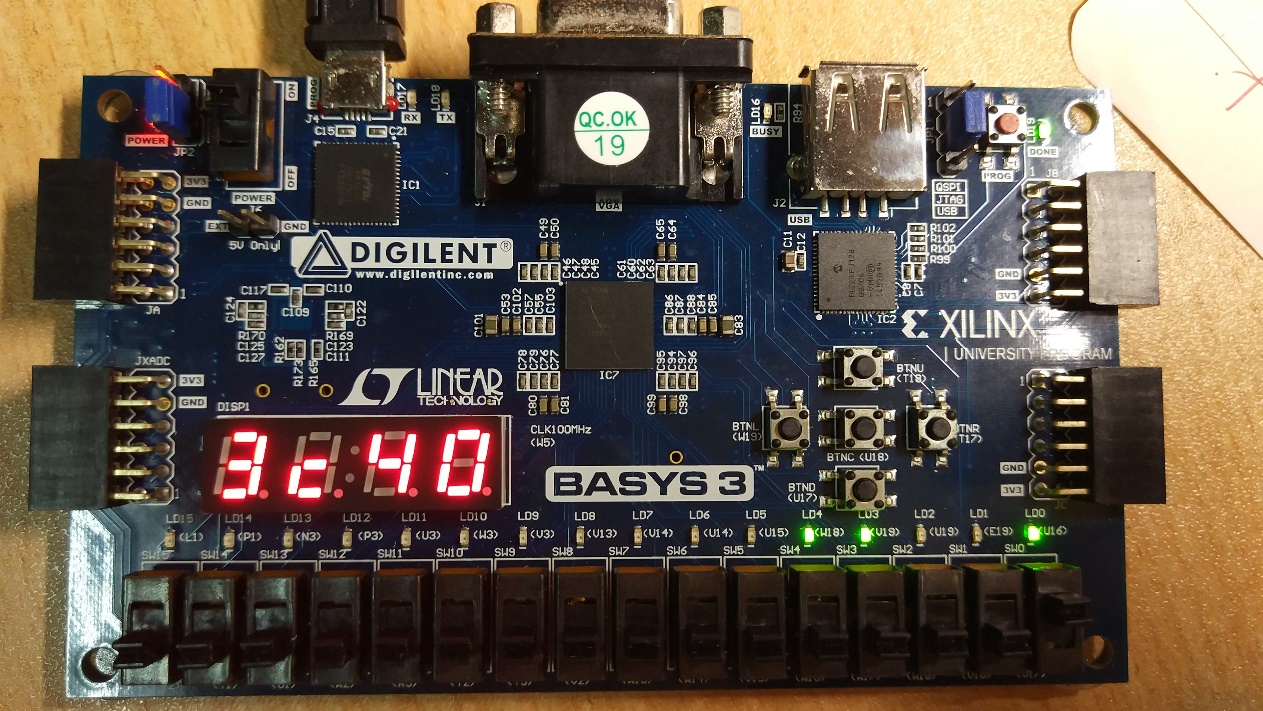


图 12：地址为 0000003C 的指令的运行图，下一条指令为 00000040

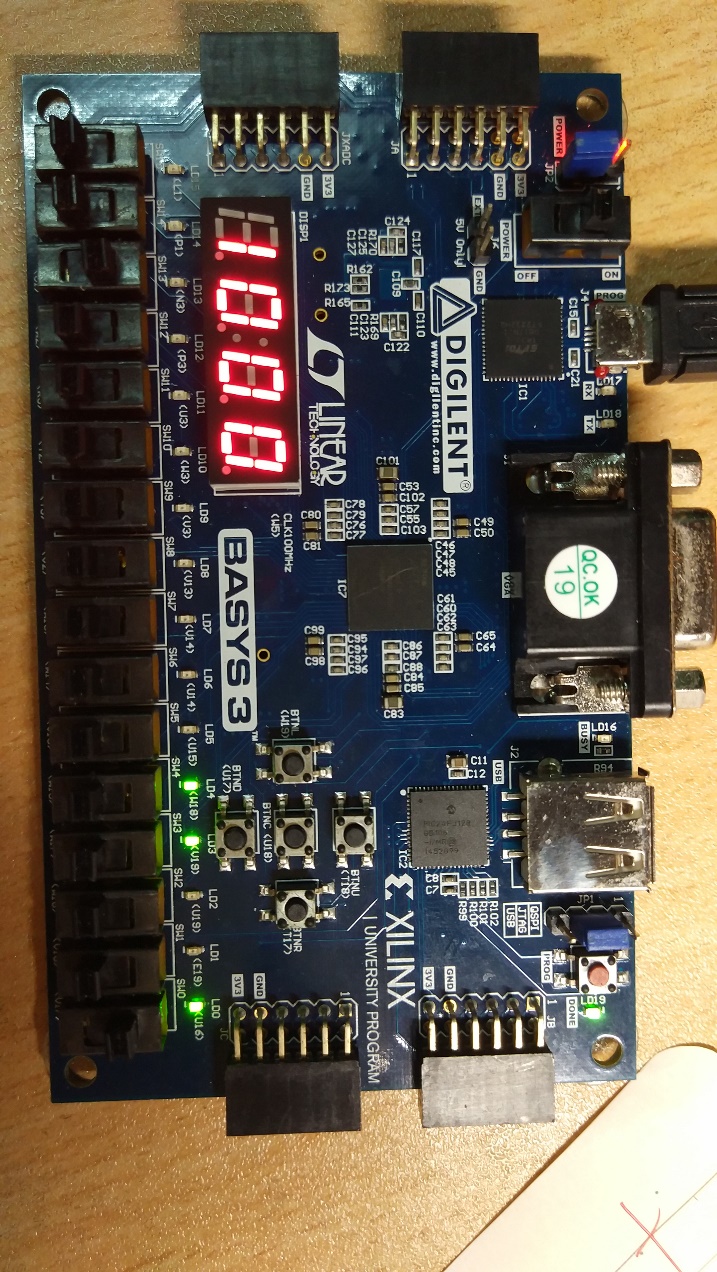


图 13：运行地址为 0000003C 的指令时，rs 寄存器的编号为 10，其值为 0

下图是第 23 条指令的执行信息：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| pc | next\_pc | 指令 | 操作 | take\_branch |
| 00000040 | 00000044 | bltz $10,-2 | $10=0<0 为假 | 0 |

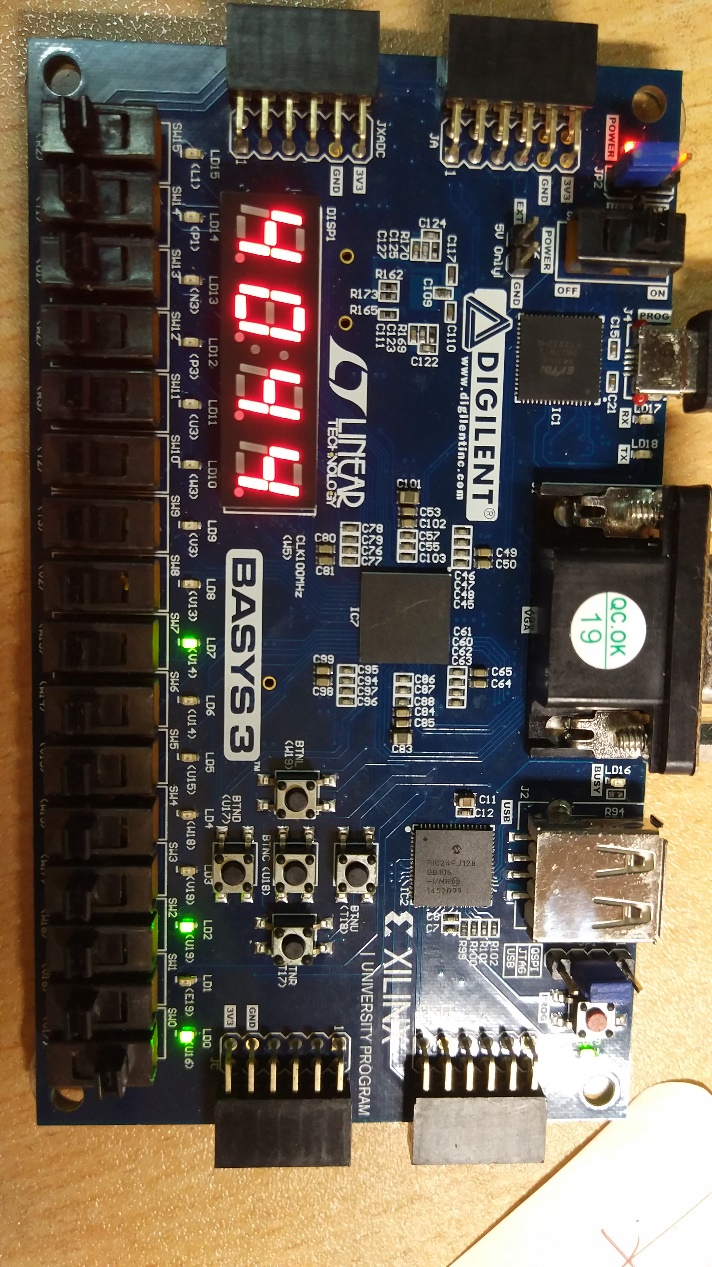


图 14：地址为 00000040 的指令的执行信息，可见图中表示 take\_branch 的 LED 5 不亮，说明不发生跳转，因此下一条指令的地址就是 00000044

下图是第 24 条指令的执行信息：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| pc | next\_pc | 指令 | write\_addr | write\_data |
| 00000044 | 00000048 | andi $11,$2,2 | $11 | 2 |

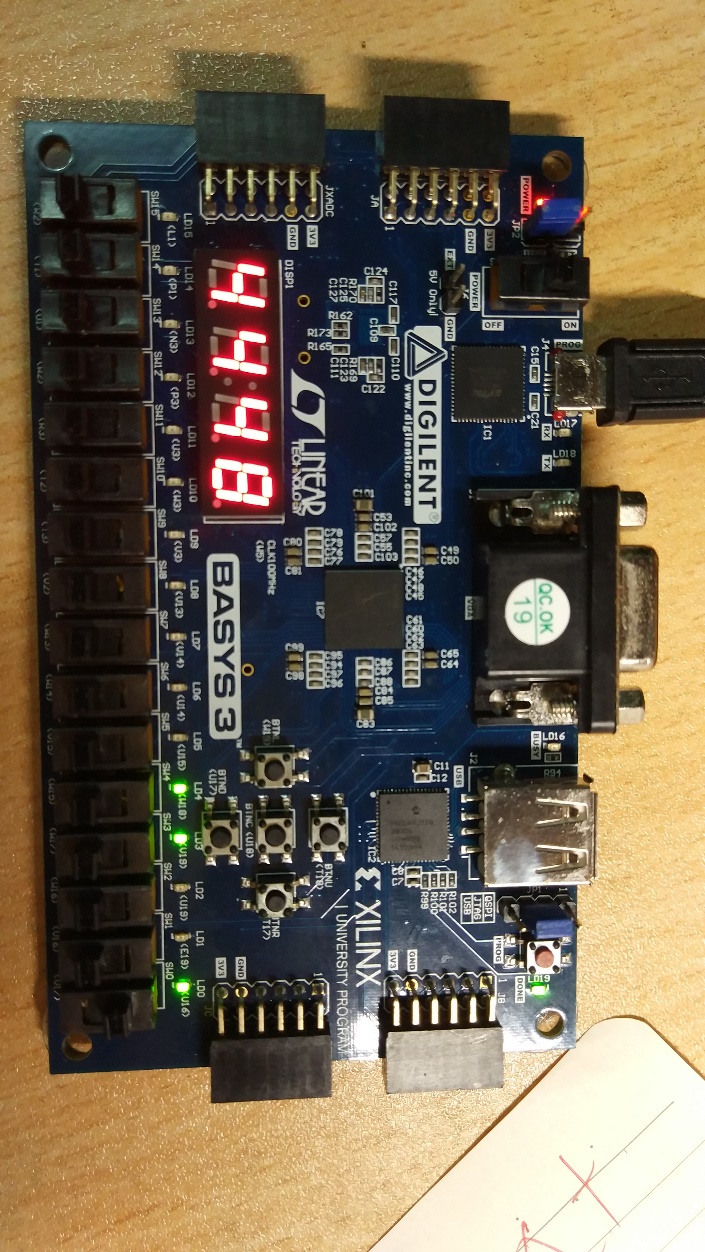


图 15：地址为 00000044 的指令的执行状态，可见 RegWre=1，BCtrl=1

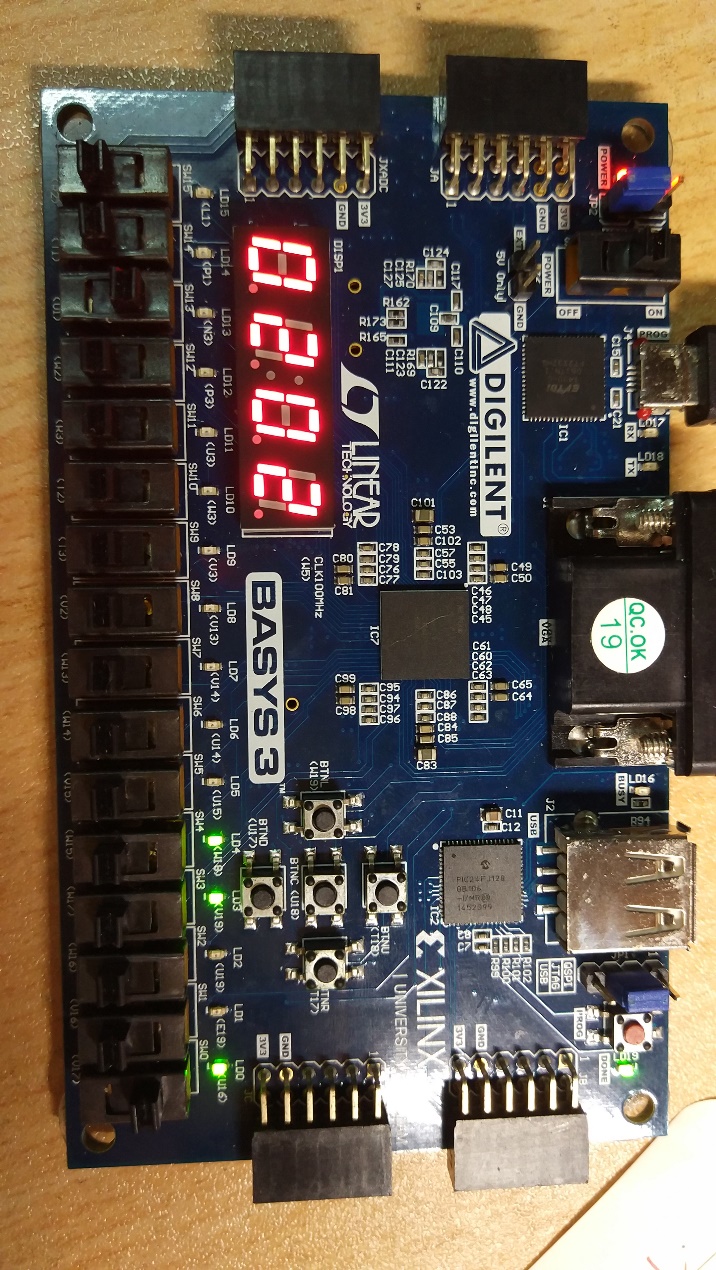


图 16：地址为 00000044 的指令的 rs 寄存器的值为 2，其值为 2（可以看到 SW13 被上拨）

下图是第 25 条指令的执行信息：

|  |  |  |
| --- | --- | --- |
| pc | next\_pc | 指令 |
| 00000048 | 00000050 | j 0x00000050 |

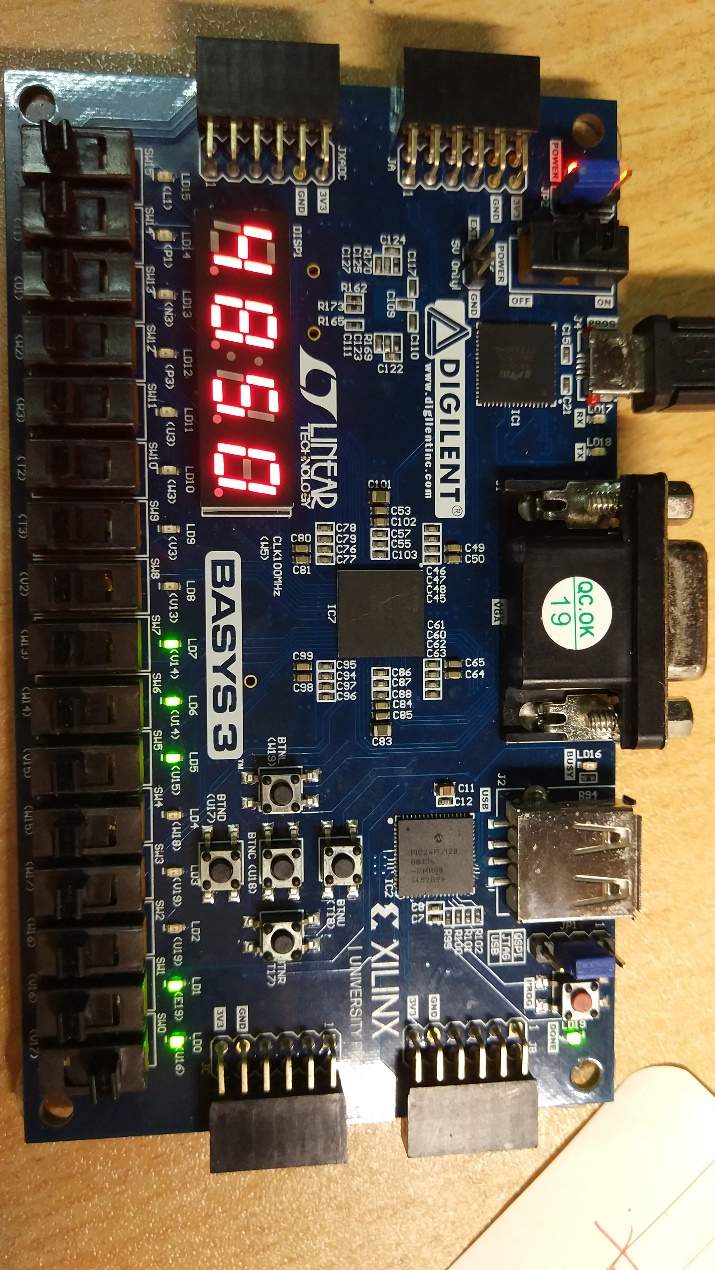


图 17：地址为 000000048 的指令的状态，可见 fetch\_rw=1、RegWre=0、BCtrl=0、TakeBranch=1、Jump=1、Branch=1、halt=0

* + - 1. **实验心得**

在实现数据存储器 RAM 的时候，我一开始以为对数据存储器的读和写都是要在下降沿进行。到了仿真查看波形图时发现 lw 指令虽然能读到存储器的数据，也就是 mem\_read 信号的值确实是 2，但是 9 号寄存器的值却变成了高阻态 32’bZZZZZZZZ。即写入寄存器的值是 lw 指令读存储器之前的 mem\_read 信号的值。发现因为写入寄存器和读取存储器的时机是在同一个下降沿，而联想到在数字电路课程中学过的知识：触发器更新值的时候读取的值上升沿或者是下降沿之前的值。这也就解释了为什么 lw 指令无法正常将数据写入寄存器。

在实现数据存储器 RAM 的时候，为了解决 lw 指令的问题，我将读取存储器的时机改成上升沿，可喜地发现我读取的值变成了正确的值，整个波形也符合预期。但是我在写本篇实验报告时，在总结第一点时便联想到在上升沿时读取存储器同样是不对的。为了证明我的想法，我特地修改了测试程序代码，在 sw 和 lw 指令中插入了一条运行结果为 0 的指令，发现 lw 指令的时序图就错误了，内存读取到的数据是 0 而不是 2。这是因为 lw 指令使用的内存地址是通过 ALU 运算得到的，但是在时钟上升沿之前 ALU 的运算结果并不是 lw 指令的内存地址的运算结果，而是上一条指令的运算结果，而老师提供的测试程序未能验证该问题，就是因为 lw 指令的上一条指令 sw 指令的运行结果和 lw 指令是一样的，这样 lw 指令在上升沿时读取数据就没有问题。而我特地在 lw 指令之前加入了一条运行结果为 0 的指令，就会导致在上升沿读取数据的时候读取的是地址为 0 的内存单元的值（以上一条指令的运行结果作为当前指令的内存地址）。因此为了解决这个问题，我将 lw 指令读取存储器的时机改为连线而不是时钟触发就解决了问题。**建议在测试程序的 sw 指令和 lw 指令之间插入一条 addiu $1,$0,0 指令来检验学生在读取内存时是否是上升沿读取，因为上升沿读取内存是错误的。**

对于某些指令，我在查看时序图时发现其 ALU 的运算结果在下降沿发生了变化；在写板后，查看这些指令的运算结果发现是错误的，这让我误以为我的 CPU 的实现有问题。但后来发现其实不是的。这些指令的共性是操作数寄存器和写回的寄存器是同一个寄存器，也就是说在写回后 ALU 的运算结果就会更新成执行了两次的值，因此我在写板的时候特地添加了显示写回寄存器的当前值以便检查是否是该问题，结果发现确实是的。

写板的时候我一开始使用提供的按钮直接当做 Cpu 的时钟信号，在 Implementation 阶段失败，提示 Poor placement for routing between an IO pin and BUFG. 等信息。在上网查找资料后未能找到有效的信息。在仔细思考后发现，Vivado 的综合器无法综合的提示是其无法正确地进行时钟信号的布线。但是分频器连出的信号却可以被 Vivado 综合器识别。在考虑了两者的区别后，我联想到 Vivado 的综合器只支持综合与板载晶振信号同步的信号作为时钟信号。在将按钮信号经过与时钟信号有关的防抖处理后成功解决了该问题。

我在测试写板的正确性时，发现执行的顺序很奇怪，几个跳转指令都不止跳了一次，而是跳了很多次。为了查找问题，我添加了更多的调试信息显示到 Led 和 七段显示管上。发现第一条指令执行后无法写入寄存器，而其他指令可以正常写入寄存器。思考后发现，我在第一次点按按钮后就执行完了第二条指令。也就是说第一条指令缺失了一次下降沿，导致无法回写寄存器。为了解决这个问题，我才在 PC 寄存器的实现中加入了复位后的第一次时钟信号不更新 PC 寄存器。这样运行的结果就是，第一次点按按钮时执行的就是地址为 00000000 的正确的了。