****

**《计算机组成原理实验》**

**实验报告**

**（实验三）**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **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 | 12 | 月 | 9 | 日 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  | **成绩** | **:** |  |
| **实验三** | **：** | **多周期 CPU 设计与实现** | | | |

* + - 1. **实验目的**
  1. 认识和掌握多周期数据通路图的构成、原理及其设计方法；
  2. 掌握多周期 CPU 的实现方法，代码实现方法；
  3. 编写一个编译器，将 MIPS 汇编程序编译为二进制机器码；
  4. 掌握多周期 CPU 的测试方法；
  5. 掌握多周期 CPU 的实现方法。
     + 1. **实验内容**

设计一个多周期CPU，该 CPU 至少能实现指定的 19 条指令功能操作。

* + - 1. **实验原理**

简单的 MIPS CPU 由 5 个阶段组成： IF（取指令）、ID（解码、读寄存器）、EXE（执行，ALU 运算）、MEM（内存读写）、WB（写回寄存器）。其中对于 JAL 指令，其 ID 阶段同时进行寄存器写回操作从而省去了 WB 阶段。

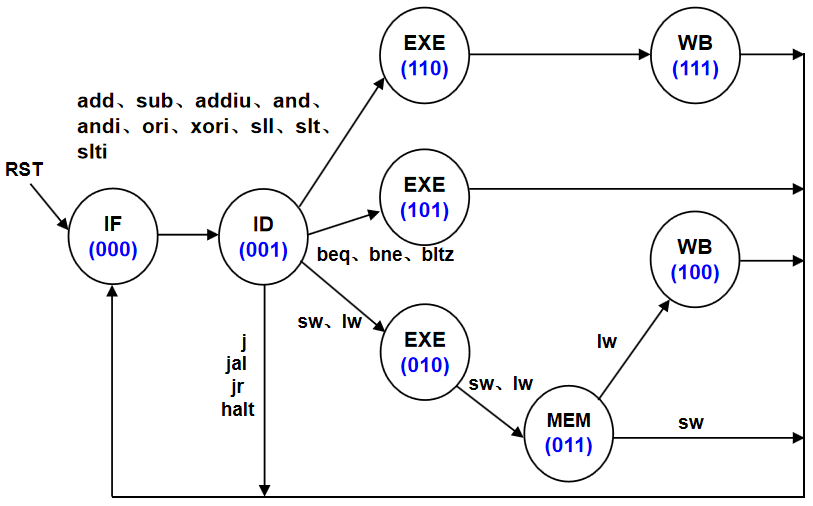


图 1：MIPS CPU 的五个阶段，根据指令特征选择执行

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。

**状态控制器模块：**

**状态控制器模块**是多周期 CPU 新增的控制模块，负责控制 CPU 的执行状态，决定当前要执行哪一个周期。状态控制器在时钟的上升沿，根据上的状态转移图决定 CPU 下一步将进行哪一个阶段。下列的所有模块的端口表均省略了 clk 和 rst\_n 信号。

|  |  |  |
| --- | --- | --- |
| 端口类型 | 端口名 | 说明 |
| 输入端 | clk | 时钟信号 |
| 输入端 | rst\_n | 复位信号，低电平时复位。 |
| 输入端 | inst | 当前指令类型，由解码器产生。决定下一步阶段。 |
| 输入端 | mem\_en | 当前指令是否是内存操作指令，由解码器产生。决定是否需要将 EXE 阶段转移到 MEM 阶段 |
| 输出端 | state | 当前的状态，在时钟的上升沿更新为 next\_state。 |
| 输出端 | next\_state | 下一步状态，用于在时钟上升沿时判断当前状态的组件使用。 |

表 1：state\_unit 端口表

|  |
| --- |
| wire arithmetic = inst == `INST\_ADD || inst == `INST\_SUB ||  inst == `INST\_ADDU || inst == `INST\_AND || inst == `INST\_OR ||  inst == `INST\_XOR || inst == `INST\_SLL || inst == `INST\_SLT;  wire branch = inst == `INST\_BEQ || inst == `INST\_BNE || inst == `INST\_BLTZ;  wire halt = inst == `INST\_HALT; // 是否是 halt 指令  // arithmetic - 是否是算术指令，branch - 是否是条件跳转指令  always @\*  case (state)  `STATE\_IF: next\_state = `STATE\_ID;  `STATE\_ID:  if (arithmetic || branch || mem\_enable) next\_state = `STATE\_EXE;  else if (halt) next\_state = `STATE\_HALT;  else next\_state = `STATE\_IF; // j、jal、jr 指令  `STATE\_EXE:  if (arithmetic) next\_state = `STATE\_WB;  else if (branch) next\_state = `STATE\_IF;  else if (mem\_enable) next\_state = `STATE\_MEM;  else next\_state = `STATE\_INVALID;  `STATE\_MEM:  if (inst == `INST\_LW) next\_state = `STATE\_WB;  else next\_state = `STATE\_IF;  `STATE\_WB : next\_state = `STATE\_IF;  `STATE\_HALT: next\_state = `STATE\_HALT;  `STATE\_INIT: next\_state = `STATE\_IF;  `STATE\_INVALID: next\_state = `STATE\_INVALID;  default:  next\_state = `STATE\_IF;  endcase  // 在时钟上升沿时更新执行周期  always @(posedge clk, negedge rst\_n)  if (!rst\_n) state <= `STATE\_INIT;  else state <= next\_state; |

**IF 阶段模块：**

IF 阶段在上升沿时更新 PC 寄存器，后半周期从指令存储器中读取对应地址的指令内容。

**PC 寄存器模块**负责维护 PC 寄存器的值，在时钟上升沿，且本次状态为 IF 阶段时更新 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，或者在其他的周期）。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 | 由状态控制器产生，非 IF、HALT 阶段时均暂停 PC 更新。 |
| 输入端 | rw | 低电平为 PC + 4，高电平时为跳转指令写入 PC。与执行阶段产生的 take\_branch 信号、ID 阶段的 jump 信号相连。 |
| 输入端 | write | 要跳转到的地址。与解码阶段产生的 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 2：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 (!stall && !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：

|  |  |  |
| --- | --- | --- |
| 端口类型 | 端口名 | 说明 |
| 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 3：一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 |

**IF/ID 阶段流水线寄存器模块**相当于 IR 指令寄存器，负责在时钟上升沿且当前要执行解码阶段时将 IF 的输出 pc、指令传给 ID 阶段模块。其端口表如下：

|  |  |  |
| --- | --- | --- |
| 端口类型 | 端口名 | 说明 |
| 输入/出 | pc\_in/out | IF 阶段的 PC 寄存器的值 → ID 阶段使用的 PC 值 |
| 输入/出 | inst\_in/out | IF 阶段从指令存储器获得的指令 → ID 及之后阶段的 IR 寄存器值 |

表 4：IF/ID 模块端口表

下面是 IF/ID 阶段流水线寄存器模块的代码，之后的 ID/EXE，MEM/WB 模块的代码也类似。而且这些模块的触发更新时间均在变更阶段为相应阶段时才触发。

|  |
| --- |
| always @(posedge clk, negedge rst\_n)  if (!rst\_n) begin  out <= 0; // 复位时清零  end else begin  if (flush) begin // 强制清零，在非下一个阶段执行时  out <= 0; // 强制清零使得下一个阶段硬件保持低电平  end else if (!stall) begin // 如果当前未被阻塞  out <= in; // 或是到了要执行的时候向下一个阶段  end // 的硬件提供输入以便运行。  end |

**ID 阶段模块：**

ID 阶段上升沿时 IF/ID 流水线寄存器模块将指令存储器读取出的指令暂存在 IR 寄存器中，同时前半周期进行解码操作。ID 阶段的后半周期进行寄存器组数据读取操作，以便算术逻辑指令进行运算、条件跳转指令计算条件。对于跳转指令，ID 阶段的后半周期同时进行跳转地址的计算；对于 jal 指令，ID 阶段的后半周期还同时将 PC + 4 写入 31 号寄存器。

**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 |

表 5：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 3 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 | ext | alu\_op | b\_ctrl | mem\_en | wb\_reg | wb\_src | jump | branch |
| add | \ | 4 | 0 | 0 | 1 | 0 | 0 | 0 |
| sub | \ | 5 | 0 | 0 | 1 | 0 | 0 | 0 |
| addiu | 1 | 4 | 1 | 0 | 1 | 0 | 0 | 0 |
| and | \ | 6 | 0 | 0 | 1 | 0 | 0 | 0 |
| andi | 0 | 6 | 1 | 0 | 1 | 0 | 0 | 0 |
| ori | 0 | 7 | 1 | 0 | 1 | 0 | 0 | 0 |
| xori | 0 | 8 | 1 | 0 | 1 | 0 | 0 | 0 |
| sll | 3 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| slti | 1 | 12 | 1 | 0 | 1 | 0 | 0 | 0 |
| slt | \ | 12 | 0 | 0 | 1 | 0 | 0 | 0 |
| sw | 1 | 4 | 1 | 1 | 0 | \ | 0 | 0 |
| lw | 1 | 4 | 1 | 1 | 1 | 1 | 0 | 0 |
| beq | 1 | 13 | 0 | 0 | 0 | \ | 0 | 1 |
| bne | 1 | 14 | 0 | 0 | 0 | \ | 0 | 1 |
| bltz | 1 | 12 | 0 | 0 | 0 | \ | 0 | 1 |
| j | \ | \ | 0 | 0 | 0 | \ | 1 | 1 |
| jr | \ | \ | 0 | 0 | 0 | \ | 1 | 1 |
| jal | \ | \ | 0 | 0 | 1 | 0 | 1 | 1 |
| halt | \ | \ | 0 | 0 | 0 | \ | 0 | 0 |

表 6：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 if (sign\_extend == `ADDR\_EXT) // jal 写回寄存器数据  imm <= pc + 4;  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 == `INST\_JR || inst == `INST\_JALR) ? `JUMP\_REG : ((inst == `INST\_J || inst == `INST\_JAL) ? `JUMP : `JUMP\_N);  assign branch = (inst == `INST\_J || inst == `INST\_JR || inst == `INST\_JAL || inst == `INST\_BLTZ || inst == `INST\_BEQ || inst == `INST\_BNE) ? `BR : `BR\_N; |

**寄存器组模块**用于读取和写入寄存器，其中时钟下降沿时写入寄存器（ID 阶段且为 jal 指令时或者 WB 阶段有效，所以同时需要在时钟下降沿时 stall\_in 信号为低电平、write\_en 信号为高电平时才写入寄存器）。其端口表如下：

|  |  |  |
| --- | --- | --- |
| 类型 | 端口名 | 说明 |
| 输入 | stall\_in | 在非 ID 阶段且指令为 jal 且非 WB 阶段时为高电平，阻塞写入操作 |
| 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 与立即数选择后相连 |
| 输入 | write\_en | 目前状态是否需要保存存储器，和译码器的 wb\_reg 信号直接相连 |
| 输入 | write\_addr[4:0] | 要写入寄存器的地址，当指令为 jal 时与解码器的 write\_addr 相连（ID 阶段回写）；否则与 MEM/WB 阶段衔接模块的 write\_addr 相连（也就是执行到 WB 阶段的指令的 write\_addr 端口） |
| 输入 | write\_data[31:0] | 要写入寄存器的数据，若在 WB 阶段：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；若在 ID 阶段：当前指令为 jal 时选择写回 pc + 4。 |

表 7：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 = write\_en == `MEM\_WRITE && (write\_addr != 0) && !stall\_in; // 当前指令要回写寄存器、且回写寄存器不是零号寄存器、且未被阻塞（当前阶段不是 ID 阶段且 JAL 指令或 WB 阶段）时  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 |

**跳转目标地址模块**负责计算当前执行的跳转指令要跳转到的指令地址。端口如下：

|  |  |  |
| --- | --- | --- |
| 类型 | 端口名 | 说明 |
| 输入 | pc[31:0] | 当前指令的地址 |
| 输入 | raw\_inst[31:0] | 当前指令的数据 |
| 输入 | alu\_rs[31:0] | 当前指令的 rs 寄存器的值 |
| 输入 | imm[31:0] | 当前指令（条件跳转指令）的立即数 |
| 输入 | jump[1:0] | 当前指令的跳转类型，与解码器的 jump 端口直接相连 |
| 输出 | br\_tgt[31:0] | 跳转地址 |

表 8：pc\_targeter 模块端口表

实现如下：

|  |
| --- |
| wire [`DATA\_BUS] pc4 = pc + 4;  always @\*  if (jump == `JUMP) // j, jal  branch\_target = { pc4[DATA\_WIDTH-1:28], raw\_inst[`RAW\_INST\_ADDR\_BUS], 2'b00 };  else if (jump == `JUMP\_REG) // jr, jalr  branch\_target = alu\_rs;  else if (jump == `JUMP\_COND) // beq, bne, bltz  branch\_target = pc4 + {imm[DATA\_WIDTH-3:0], 2'b00};  else  branch\_target = 0; |

ID/EXE 阶段流水线寄存器模块负责在 EXE 阶段的上升沿时将当前指令的信息加载到寄存器中（也就是 ADR、BDR 寄存器的实现）。该模块端口表如下：

|  |  |
| --- | --- |
| inst[6:0] | 要执行 EXE 阶段的指令的类型 |
| alu\_op[4:0] | 要执行 EXE 阶段的指令的 ALU 操作码 |
| alu\_rs[31:0] | ID 阶段读取到的 rs 寄存器的值 |
| alu\_rt[31:0] | 第二操作数，并在此选择是立即数还是 rt 寄存器值 |
| mem\_enable | 中继给 MEM 阶段的数据存储器使用 |
| mem\_write[31:0] | 中继给 MEM 阶段的数据存储器使用 |
| wb\_src | 通知 WB 阶段当前指令的写回类型是 ALU 还是内存 |
| jump[1:0] | 用于判断要执行 EXE 阶段的指令是不是条件跳转指令 |

表 9：ID/EXE 阶段流水线寄存器模块端口表

**执行模块：**

执行阶段整个周期都用于 ALU 计算操作结果。

**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 |
| NOP | 0 | 无操作 | 无操作 |
| 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 |
| XOR | 8 | rd = rs ^ rt | 按位异或 |
| 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 10：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 11：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 |

**EXE/MEM 阶段流水线寄存器模块**负责将 ALU 的结果传递给一级数据缓存，也就是 相当于 ALUoutDR 寄存器。

**访存阶段：**

在访存阶段，对于 lw 指令，下降沿时利用后半周期读取数据存储器相应地址的值；对于 sw 指令，下降沿时利用后半周期向数据存储器相应地址写入数据。其他指令没有访存阶段。

**一级数据缓存模块**即数据存储器，用于存储数据。在 MEM 阶段的时钟上升沿时读取或写入。其信号表如下：

|  |  |  |
| --- | --- | --- |
| 输入端 | addr[31:0] | 操作数据存储器内存单元的地址，与 ALU 输出端 ALUoutDR 直接相连。 |
| 输入端 | enable | 是否启用存储器，仅在当前指令是 lw 或 sw，且当前状态是 MEM 阶段时启用存储器 |
| 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] | 要写入数据存储器的数据 |
| 输出端 | read[31:0] | 数据存储器对应地址的数据 |

表 12：一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 @\*  if (enable) begin  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 <= 'bZ; // 内存禁用时设置为高阻态以便数据存储器的串并联  always @(negedge clk, negedge rst\_n)  if (!rst\_n)  for (i = 0; i < 128; i = i + 1)  dat[i] <= 0;  else if (enable && rw == `MEM\_WRITE) begin  dat[addr ] <= write[31:24];  dat[addr + 1] <= write[23:16];  dat[addr + 2] <= write[15: 8];  dat[addr + 3] <= write[ 7: 0];  end |

**MEM/WB 阶段流水线寄存器模块**负责在时钟上升沿时将是否要回写寄存器、回写的数据、回写的寄存器编号传递给回写阶段的寄存器组模块。

**回写阶段：**回写阶段就是寄存器组的写入。我们在下降沿时进行寄存器组的写入，是因为 MEM/WB 阶段的寄存器的写入在时钟的上升沿，我们写入寄存器时要获得写入的值，一种是在上升沿时拿流水线寄存器的输入数据（因为上升沿时流水线寄存器还未完成写入），或者在下降沿时拿流水线寄存器的输出数据。从数据的稳定性方面考虑，在下降沿时拿寄存器的数据要更稳定一些。同时对于 jal 指令，因为解码在 ID 阶段的上升沿完成，要回写也必须等到 ID 阶段的下降沿完成。因此寄存器组的写入就在下降沿时（且阶段为 ID 或 WB）完成。

**数据通路图：**

数据通路图通过 Vivado 综合得到，不过由于过于复杂，在此省略。

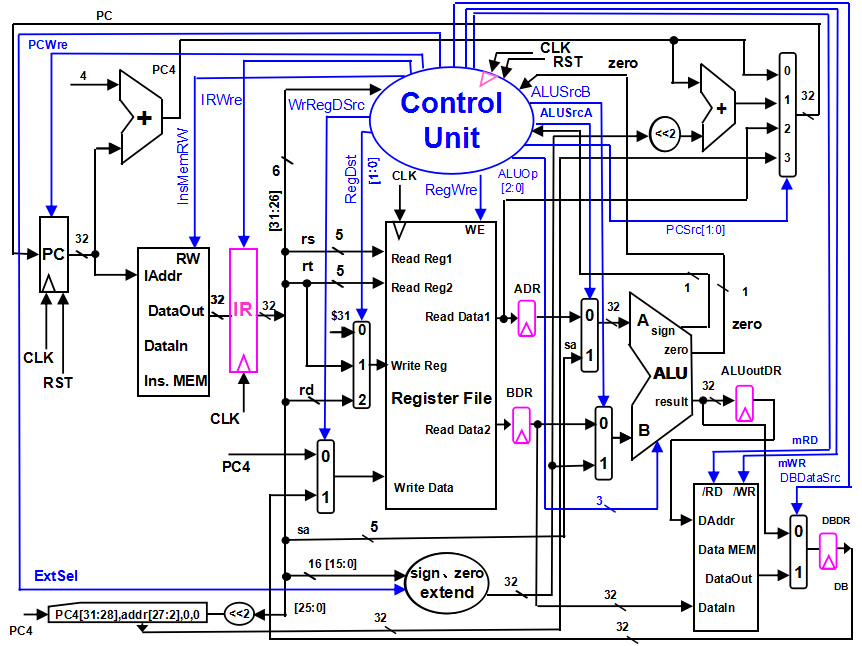


图 2：多周期 CPU 数据通路图

* + - 1. **实验器材**

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

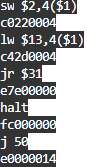
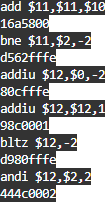
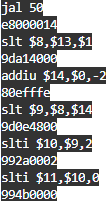
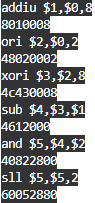
* + - 1. **实验过程与结果**

1. **制作 MIPS 指令编译器**

首先在设计 CPU 前我们需要预先设计 MIPS 指令的编译器以便通过 MIPS 汇编程序生成可执行的 MIPS 机器码，实现比较简单，代码如下：

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  typedef unsigned int uint;  struct bad\_lexical\_cast : public exception  {  };  struct compilation\_error : public exception  {  };  void to\_lower(string &str) // 使字符串变成小写，不区分大小写  {  for (string::iterator it = str.begin(); it != str.end(); ++it)  \*it = tolower(\*it);  }  bool is\_name\_char(int ch) // 判断是不是标识符的字符，比如指令名就是标识符  {  return isdigit(ch) || isalpha(ch) || ch == '\_';  }  class lexical\_analyzer  {  public:  explicit lexical\_analyzer(const string &code)  : code(code)  {  }  std::vector<string> analyze()  {  std::vector<string> result;  string t;  while (t = next\_token(), !t.empty())  result.push\_back(t);  return result;  }  private:  string code;  int cur = 0;  int next\_char()  {  return cur < (int)code.size() ? code[cur++] : EOF;  }  int peek\_char(int offset = 0)  {  return cur + offset < (int)code.size() ? code[cur + offset] : EOF;  }  string next\_token()  {  string token;  int c = next\_char();  while (isspace(c))  c = next\_char();  if (c != EOF)  {  token += (char)c;  if (c == '$') // 解析寄存器  {  while (c = peek\_char(), isdigit((char)c))  token += (char)next\_char();  }  else if (isdigit(c) || c == '-') // 解析数字  {  while (c = peek\_char(), isdigit((char)c) || isalpha((char)c))  token += (char)next\_char();  }  else if (!is\_name\_char(c)) // 解析符号  {  //  }  else  { // 解析指令  while (c = peek\_char(), is\_name\_char(c))  token += (char)next\_char();  to\_lower(token); // case insensitive.  }  }  return token;  }  };  class syntax\_analyzer  {  vector<string> tokens;  int ptr;  public:  syntax\_analyzer(vector<string> tokens)  : tokens(tokens), ptr(0) {}  string peek\_token(int offset = 0) const  {  return ptr + offset >= (int)tokens.size() ? "" : tokens[ptr + offset];  }  string next\_token()  {  return ptr >= (int)tokens.size() ? "" : tokens[ptr++];  }  void assert\_next\_token(const string &expected)  {  assert(next\_token() == expected);  }  };  uint R\_inst(uint opcode, uint rs, uint rt, uint rd, uint sa)  { // 通过 rs, rt, rd, sa, funct 构造 R 型指令的机器码  return opcode << 26 | rs << 21 | rt << 16 | rd << 11 | sa << 6;  }  uint I\_inst(uint opcode, uint rs, uint rt, uint imm)  { // 通过 rs、rt、imm 构造 I 型指令的机器码  return opcode << 26 | rs << 21 | rt << 16 | imm;  }  uint J\_inst(uint opcode, uint addr) // 通过 addr 构造 J 型指令的机器码  {  return opcode << 26 | addr;  }  template <typename T>  T lexical\_cast(const string &str, int base = 0)  {  return strtol(str.c\_str(), nullptr, base);  }  uint parse\_register(string token) // 解析寄存器编号  {  assert(token.length() > 0 && token[0] == '$');  return lexical\_cast<uint>(token.substr(1));  }  function<uint(syntax\_analyzer &)> parse\_arithmetic(uint opcode)  { // 解析 R 型指令  return [=](syntax\_analyzer &sa) {  uint rd = parse\_register(sa.next\_token());  sa.assert\_next\_token(",");  uint rs = parse\_register(sa.next\_token());  sa.assert\_next\_token(",");  uint rt = parse\_register(sa.next\_token());  return R\_inst(opcode, rs, rt, rd, 0);  };  }  // 一些类似的解析函数，用于解析不同类型的汇编命令  map<string, function<uint(syntax\_analyzer &)>> parsers;  uint parse(string inst)  {  lexical\_analyzer la(inst);  syntax\_analyzer sa(la.analyze());  string op = sa.next\_token();  for (const auto &p : parsers)  {  if (op == p.first)  return p.second(sa);  }  return 0;  }  int main()  { // 所有可以解析的指令列表，参数为其 opcode  parsers["add"] = parse\_arithmetic(0x00);  parsers["sub"] = parse\_arithmetic(0x01);  parsers["addiu"] = parse\_imm(0x02);  parsers["and"] = parse\_arithmetic(0x10);  parsers["andi"] = parse\_imm(0x11);  parsers["ori"] = parse\_imm(0x12);  parsers["xori"] = parse\_imm(0x13);  parsers["sll"] = parse\_shift(0x18);  parsers["slti"] = parse\_imm(0x26);  parsers["slt"] = parse\_arithmetic(0x27);  parsers["sw"] = parse\_mem(0x30);  parsers["lw"] = parse\_mem(0x31);  parsers["beq"] = parse\_branch(0x34);  parsers["bne"] = parse\_branch(0x35);  parsers["bltz"] = parse\_imm\_z(0x36);  parsers["j"] = parse\_j(0x38);  parsers["jr"] = parse\_jr(0x39);  parsers["jal"] = parse\_j(0x3A);  parsers["halt"] = parse\_only\_opcode(0x3F);  string line;  while (getline(cin, line))  {  cout << hex << parse(line) << endl;  }  return 0;  } |

**运行结果如下图所示：**

****

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；5 个阶段之间的流水线寄存器可以设计成 4 个大模块；每个阶段内部的组件，再分成一些小模块，比如 ALU、寄存器组等。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。这样实现的好处是方便我们日后将多周期的 CPU 修改成流水线的 CPU。

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 | 跳转指令类型 |
| zero | ALU 的值是否为零 | clk/rst\_n | 时钟信号/复位信号 |
| 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 | 条件跳转指令是否跳转 | file/dat | 寄存器组/数据存储器 |
| state | 当前阶段码 | next\_state | 下一个阶段的编号 |

表 13：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 6 条指令**，它们的波形图如下图：

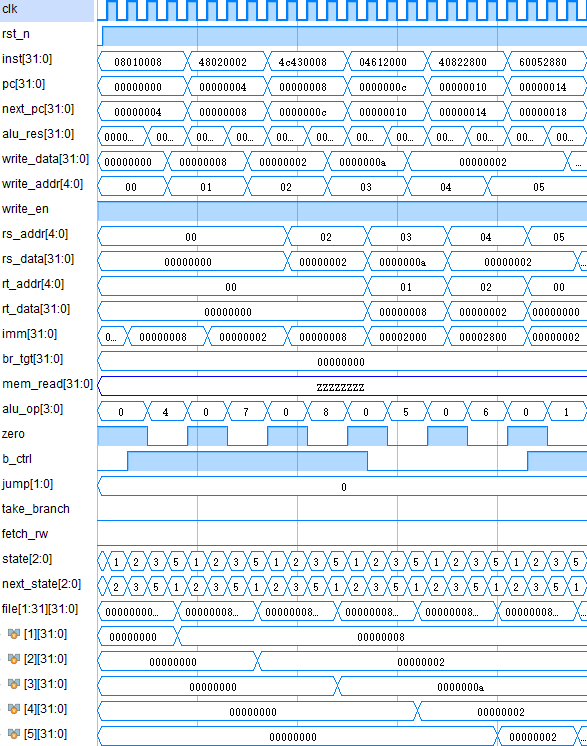


图 3：第 1~6 条指令的波形图

上图中执行的 pc=0x0~0x10 的这五条指令分别是：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| pc | 指令 | 操作(alu\_res) | alu\_op | ext | b\_ctrl |
| 00 | addiu $1,$0,8 | $1=$0+8=8 | ADD(4) | 1 | 立即数(1) |
| 04 | ori $2,$0,2 | $2=$0+2=2 | OR(7) | 0 | 立即数(1) |
| 08 | xori $3,$2,8 | $3=$2^8=10 | XOR(8) | 0 | 寄存器(0) |
| 0C | sub $4,$3,$1 | $4=$3-$1=2 | SUB(5) | \ | 寄存器(0) |
| 10 | and $5,$4,$2 | $5=$4&$2=2 | AND(6) | \ | 寄存器(0) |
| 14 | sll $5,$5,$2 | $5=$5<<2=8 | SLL(1) | \ | 寄存器(0) |

表 14：图对应执行的指令信息

从上图和表中可见：1. 首先在第一次时钟的上升沿前，state 为 0 表示初始状态，在第一次时钟的上升沿后才变成 1，即 IF 阶段；2. 每条指令的时钟周期数都是 4，因为这 6 条指令都是算术逻辑指令，都只需要 IF、ID、EXE、WB，4 个阶段；3. ALU 的第二个操作数选择信号 b\_ctrl 的值是正确的，而且 ALU 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；4. 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 7~10 条指令**，地址范围在 00000018~0000001C 之间。

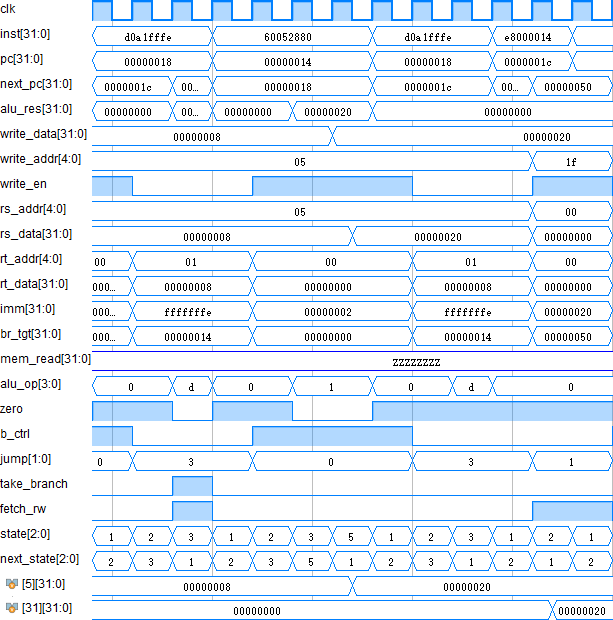


图 4：第 7~10 条指令的时序图

下表是上图中执行的 4 条指令的一些关键信息。其中条件跳转指令的回写寄存器信号 write\_en 为低电平、jal 指令的 write\_en 为高电平（因为要写回 31 号寄存器）。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| PC | 指令 | 操作 | jump | take\_br | write\_en | fetch\_rw |
| 18 | beq $5,$1,-2 | 跳转 | 3 | 1 | 0 | 1 |
| 14 | sll $5,$5,2 | $5=$5<<2=32 | 0 | \ | 1 | 0 |
| 18 | beq $5,$1,-2 | 不跳转 | 3 | 0 | 0 | 0 |
| 1C | jal 0x00000050 | 跳转 | 1 | \ | 1 | 1 |

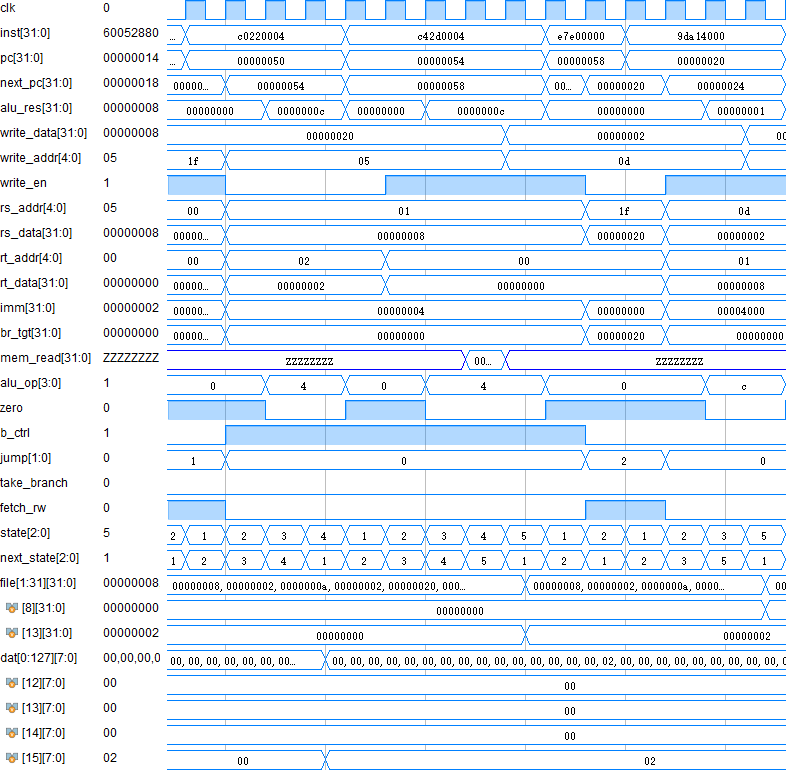
表 14： 第 7~10 条指令的执行信息表

可以看到，首先在 state=1 时的时钟上升沿时 PC 发生了更新，同时指令内容也跟着更新，所以在图中的第一次时钟上升沿时指令内容就变成了 D0A1FFFE，然后再第二次时钟上升沿时 state=2，也就是进入 ID 阶段，此时有信号 write\_en、b\_ctrl、rs\_data、rt\_data 等信号发生了改变，这是因为 CPU 在 ID 阶段的上升沿时进行了解码操作，将指令相应的各个控制型号计算了出来，再过来的第三次时钟周期，此时 state=3，表明当前的阶段是 EXE 阶段，此时 alu\_op、alu\_res、take\_branch、fetch\_rw 信号发生了改变，这是因为在 EXE 阶段的时钟上升沿时 ALU 获得了输入数据后进行了计算，得到了 beq 指令的运算结果，同时得到了是否要跳转的信号。

然后再过来的一次时钟周期就应用了跳转走到了下一条指令（PC=0x14），同时 state 变回了 1（IF 阶段）。看到这条指令的第四个时钟周期，也就是 state 变化成 5（WB 阶段）的时候，我们看到在这个时钟周期的上升沿时 write\_data 信号发生了改变，这是因为在 WB 阶段的上升沿 DBDR 寄存器接收了 ALU 的运算结果。在这个周期的时钟下降沿我们可以看到 5 号寄存器（最下面的信号）发生了改变，从 0x08 变化到 0x20。

最后我们再看到 E8000014 指令（PC=1C），其第二个时钟周期的 state 为 2（ID 阶段），其下降沿时 31 号寄存器的值改变成 0x20，也就是 1C+4，是正确的值。

**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 11~14 条指令的执行时序图

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| PC | 指令 | 操作 | alu\_op | mem\_en | fetch\_rw |
| 50 | sw $2,4($1) | $1+4=12,[12]<-2 | ADD(4) | 1 | 0 |
| 54 | lw $13,4($1) | $1=4=12,$13=[12]=2 | ADD(4) | 1 | 0 |
| 58 | jr $31 | 跳转到 $31=0x20 | \ | 0 | 1 |
| 20 | slt $8,$13,$1 | $8=$13<$1=2<8=1 | LT(12) | 0 | 0 |

表 15：上图中执行的第 11~14 条指令的信息表

前两条是内存操作指令。sw 指令执行时一共花费了 4 个时钟周期，分别是 IF、ID、EXE、MEM。其中在 EXE 阶段的上升沿时可以看到 alu\_res 为 0x0C，也就是说计算出了相应要操作的内存地址，然后在 MEM 阶段（也就是第 4 个周期）的下降沿时我们可以看到 dat[15] 从 0 变成了 0x02，这是因为我们一次性操作了 12~15 四个内存单元，而且 MIPS 是大端模式，因此高地址存低位，所以 dat[12~14] 都为 0 而 dat[15]=2。

接下来的 lw 指令执行一共花费了 5 个时钟周期，把所有阶段按顺序执行了一遍。其中在 EXE 阶段的上升沿时可以看到 alu\_res 也为 0x0C，在 MEM 阶段的上升沿时 mem\_read 信号从高阻态变成了 0x02，这表明 lw 指令成功读取的内存数据。在其 WB 阶段的下降沿时可以看到 8 号寄存器的值发生了改变。

接下来的 jr 指令只用了 2 个时钟周期，分别是 IF、ID。在 ID 阶段的上升沿可以看到 br\_tgt 信号变成了 0x20，说明 jr 指令成功读取了寄存器值并发生了正确的跳转。

**下图是第 15~19 条指令的时序图**，地址范围是 00000024~00000034。

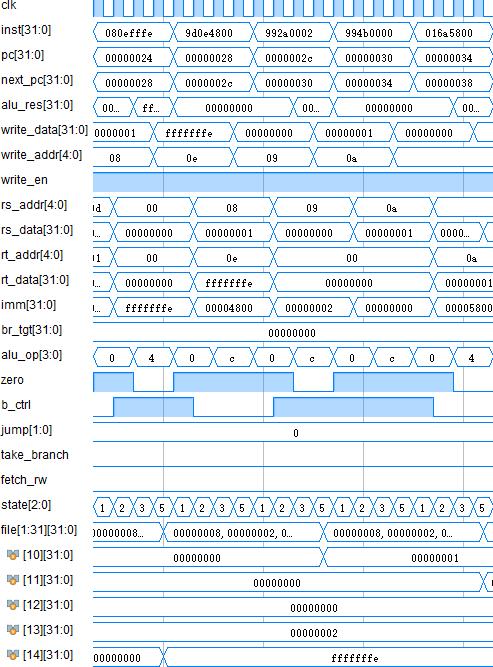


图 6：第 15~19 条指令的时序图

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| PC | 指令 | 操作(alu\_res) | alu\_op | b\_ctrl |
| 24 | addiu $14,$0,-2 | $14=$0-2=-2 | ADD(4) | 1 |
| 28 | slt $9,$8,$14 | $9=$8<$14=1<-2=0 | LT(12) | 0 |
| 2C | slti $10,$9,2 | $10=$9<2=0<2=1 | LT(12) | 1 |
| 30 | slti $11,$10,0 | $11=$10<0=1<0=0 | LT(12) | 1 |
| 34 | add $11,$11,$10 | $11=$11+$10=1 | ADD(4) | 0 |

表 16：第 14~19 条指令的操作信息表

由于上图比较小，看不到 alu\_res 信号的值，但是由于上述的 5 条指令都是算术类指令，也就是说回写寄存器的值就是 ALU 的运算结果，事实上我们查看 write\_data 就能发现在 WB 周期的时钟上升沿其值就变成了我们计算得到的 alu\_res。

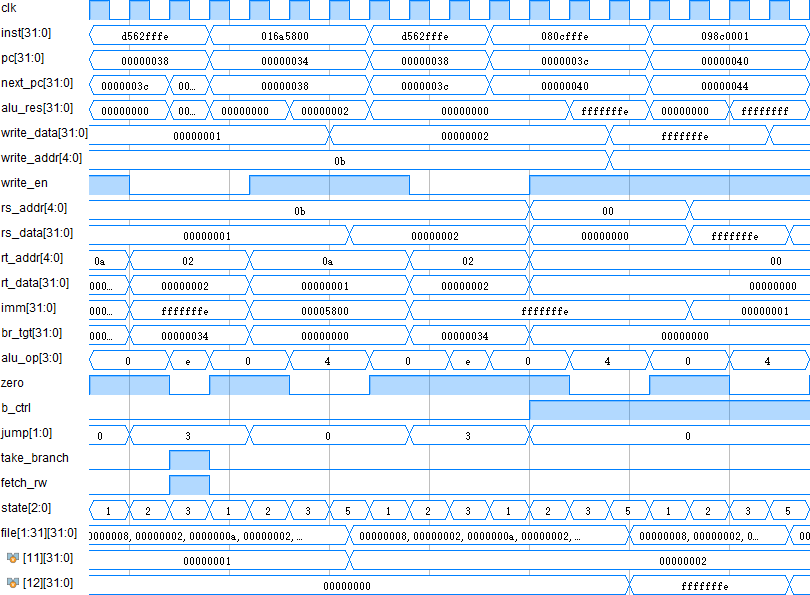
**下图是第 20~24 条指令的时序图**，地址范围是 00000034~00000040：

图 7：第 20~24 条指令的时序图

下表是第 21~25 条指令的关键信息表：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| PC | 指令 | 操作 | alu\_op | jump | write\_en | take\_br |
| 38 | bne $11,$2,-2 | 跳转 | NE(13) | 3 | 0 | 1 |
| 34 | add $11,$11,$10 | $11=0+1=1 | ADD(4) | 0 | 1 | 0 |
| 38 | bne $11,$2,-2 | 不跳转 | NE(13) | 3 | 0 | 0 |
| 3C | addiu $12,$0,-2 | $12=0-2=-2 | ADD(4) | 0 | 1 | 0 |
| 40 | addiu $12,$12,1 | $12=-2+1=-1 | ADD(4) | 0 | 1 | 0 |

表 17：第 20~24 条指令的关键信息表

由于执行的指令相似，因此上述 5 条指令的介绍请参见后 5 条指令的介绍。

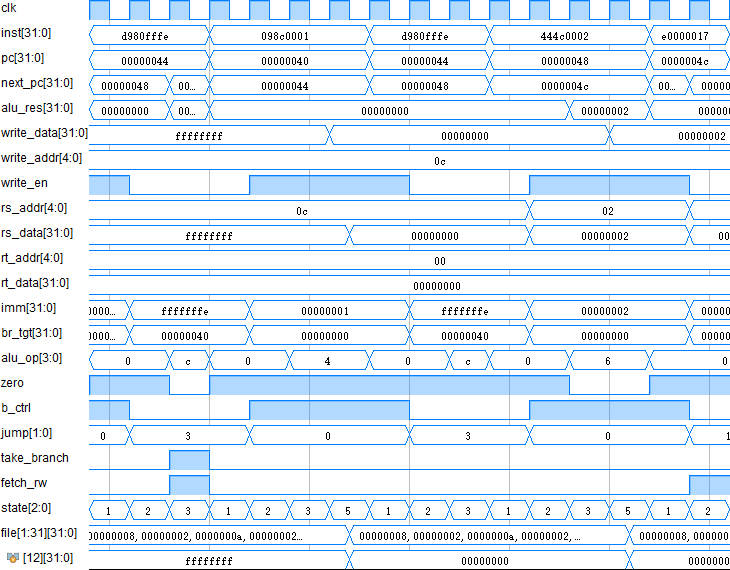
 **下图是第 25~29 条指令的时序图**，指令地址范围是 00000044~0000004C。

图 8：第 25~29 条指令的时序图

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| PC | 指令 | 操作 | alu\_op | jump | write\_en | take\_br |
| 44 | bltz $12,-2 | 跳转 | LT(12) | 3 | 0 | 1 |
| 40 | addiu $12,$12,1 | $12=-1+1=0 | ADD(4) | 0 | 1 | 0 |
| 44 | bltz $12,-2 | 不跳转 | LT(12) | 3 | 0 | 0 |
| 48 | andi $12,$2,2 | $12=$2&2=2 | AND(6) | 0 | 1 | 0 |
| 4C | j 0x0000005C | 跳转 | NOP(0) | 1 | 0 | 0 |

表 18：第 25~29 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，bltz 一共使用了 3 个时钟周期，在 EXE 阶段的上升沿中 take\_branch 信号变为高电平。J 指令的 take\_branch 为低电平，原因是该信号是条件跳转指令的。

**下图是第 29~30 条指令的时序图**，指令地址范围是 0000004C~0000005C：

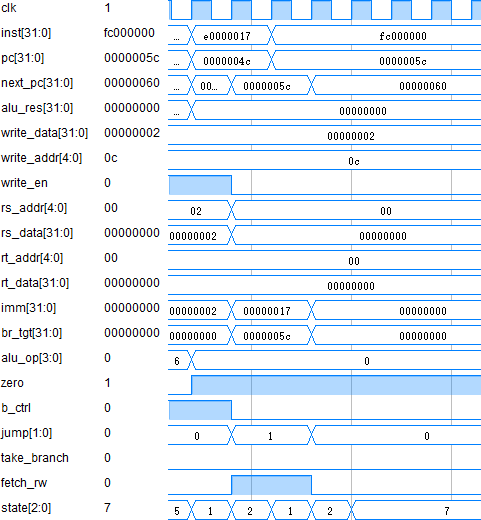


图 9：第 29~30 条指令的时序图

下表是第 29~30 条指令的关键信息表。从上图中我们看到，由于 j 指令在 ID 阶段的时钟上升沿时计算出了要跳转的地址后跳转到了 0x5C，也就是 halt 指令。该指令在 ID 阶段后的下一个周期使得状态变成 7，也就是停机状态从而阻塞 CPU 所有模块的工作从而实现停机，直到下一次复位。

|  |  |  |  |
| --- | --- | --- | --- |
| 地址 | 指令 | 操作 | halt |
| 4C | j 0x0000005C | PC <- 0000005C | 0 |
| 5C | halt | 停止 PC 寄存器的更新 | 1 |

表 19：第 29~30 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    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[0]; // 跳转指令信号低位（0 不跳转、1 j、jal）  LED[7] <= jump[1]; // 跳转指令信号高位（2 jr、jalr、3 bne、beq、bltz）  LED[10] <= state[0]; // 当前指令状态（0 刚复位、1 IF、2 ID）  LED[11] <= state[1]; // 当前指令状态（3 EXE、4 MEM、5 WB）  LED[12] <= state[2]; // 当前指令状态（6 不合法、7 HALT）  LED[15] <= state == `STATE\_HALT; // CPU 是否结束运行  end    always @\* // 数码管显示的信号表请参见下文  case(mode)  // 高位是当前指令的地址，低位是下一条指令的地址  0: begin {S[3], S[2]} <= pc[7:0]; {S[1], S[0]} <= next\_pc[7:0]; end  // 高位是 rs 寄存器的编号，低位是其数据  1: begin S[3] <= rs\_addr / 10; S[2] <= rs\_addr % 10; {S[1], S[0]} <= rs\_data[7:0]; end  // 高位是 rt 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 ALU 运算结果，低位是内存读数据的结果  3: begin {S[3], S[2]} <= alu\_res[7:0]; {S[1], S[0]} <= mem\_read[7:0]; end  // 高位是回写寄存器的编号，低位是回写寄存器的写入值  4: begin S[3] <= reg\_write\_addr / 10; S[2] <= reg\_write\_addr % 10; {S[1], S[0]} <= reg\_write\_data[7:0]; end  // 高位是立即数，低位是跳转指令目标地址  5: begin {S[3], S[2]} <= imm[7:0]; {S[1], S[0]} <= branch\_target[7:0]; end  // 高位是 ALU 操作码，低位是回写寄存器的实际值，用于检查是否成功写入  6: begin S[3] <= alu\_op; {S[2], S[1], S[0]} <= reg\_write\_read; end  // 高位是当前阶段编号，低位是下一个阶段的编号  7: begin {S[3], S[2]} <= state; {S[1], S[0]} <= next\_state; end  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 | 当前阶段编号 | 下一个阶段的编号 |

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[0] | 不是跳转指令或 jr | j、jal 或条件跳转指令 |
| 7 | jump[1] | 不是跳转指令或 j、jal | jr 或条件跳转指令 |
| 10 | state[0] | INIT(0)、IF(1)、ID(2)、EXE(3)、MEM(4)、WB(5)、INVALID(6)、HALT(7) | |
| 11 | state[1] |
| 12 | state[2] |
| 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 7~11 条指令，一共 5 条指令的执行流程：

下图是第 7 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 | jump | br\_tgt | take\_branch |
| 18 | 14 | beq $5,$1,-2 | 3（条件跳转） | 14 | 1 |



图 10：第 7 条指令在 IF 阶段，可以看到 LD12~10 为灭灭亮，这说明 state=3'b001。由于跳转目标地址还未计算出来，所以可以看到下一条指令的地址暂时是 pc+4。



图 11：第 7 条指令在 ID 阶段，可以看到 LD3 灭，是因为 beq 指令不回写寄存器；LD7 和 LD6 均亮，表示当前指令的 jump 为 3，也就是条件跳转指令；LD12~10 为灭亮灭，说明 state=3'b010=2=ID 阶段。另外，现在数码管显示的是 rs 寄存器为 5 号寄存器，其值为 8



图 12：第 7 条指令在 ID 阶段，数码管显示的是 rt 寄存器为 1 号寄存器，其值为 8



图 13：第 7 条指令在 EXE 阶段，可以看到 LD5（即 take\_branch 信号）亮，说明要跳转；LD2 （即 zero 信号）暗，说明 ALU 的运算结果不为零，要跳转；现在数码管显示的是 ALU 的运算结果为 1，说明两个数相等。



图 14：第 7 条指令的 EXE 阶段，可以看到数码管显示的下一条指令地址已经是 0x14，而且 LD1（即 fetch\_rw 信号）亮，表示要覆盖 PC 寄存器从而跳转

下图是第 8 条指令的执行信息：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| pc | next\_pc | 指令 | 操作 | alu\_op |
| 14 | 18 | sll $5,$5,2 | $5=$5<<2=8<<2=32 | SLL(1) |



图 15：第 8 条指令在 IF 阶段

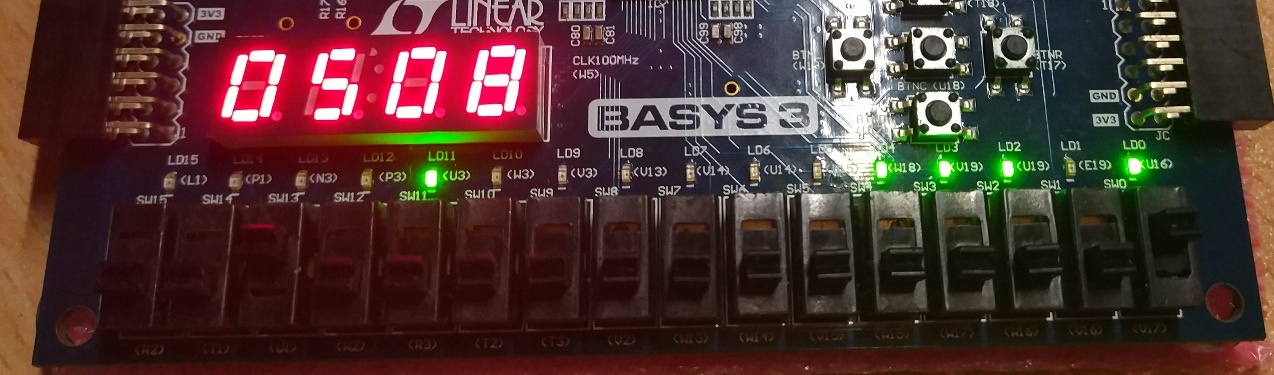


图 16：第 8 条指令在 ID 阶段，可以看到其 rs 寄存器的值为 8



图 17：第 8 条指令在 EXE 阶段，可以看到 ALU 的运算结果是 0x20



图 18：第 8 条指令在 WB 阶段，可以看到其 rs 寄存器的值确实变成了 20

下表是第 9 条指令的执行信息：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| pc | next\_pc | 指令 | jump | br\_tgt | take\_br | feth\_rw |
| 18 | 1C | beq $5,$1,-2 | 3 | 14 | 0 | 0 |



图 19：第 9 条指令在 EXE 阶段，可以看到 LD2（即 zero 信号）亮，说明 ALU 运算结果为 0；同时 LD5（即 take\_branch 信号）灭，说明不发生跳转

下图是第 10 条指令的执行信息：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| pc | next\_pc | 指令 | jump | fetch\_rw | RegWre |
| 1C | 50 | jal 0x50 | 1（跳转到立即数） | 1 | 1 |

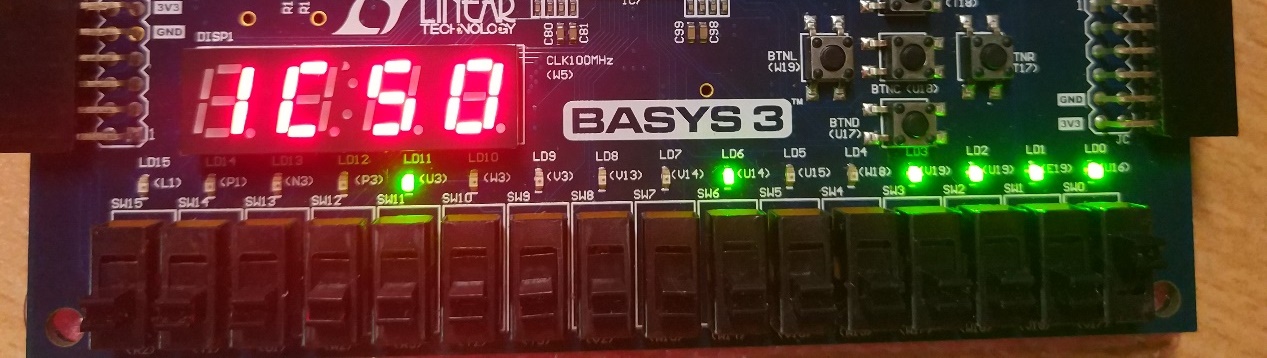


图 20：第 10 条指令在 ID 阶段，可以看到 LD1（即 fetch\_rw 信号）亮，说明要写入 PC 寄存器，即发生了跳转；同时 LD3 （即是否回写寄存器信号）亮，说明要回写 31 号寄存器。



图 21：第 10 条指令在 ID 阶段，可以看到要回写 31 号寄存器，值为 0x20



图 22：ID 阶段，当前模式查看写回的寄存器的实际值，可以看到 0x20 被成功写入

下图是第 11 条指令的执行信息：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| pc | next\_pc | 指令 | write\_addr | write\_data |
| 50 | 54 | sw $2,4($1) | 4+8=12 | $2=2 |



图 23：第 11 条指令在 EXE 阶段，可以看到 rt 寄存器为 2 号寄存器，其值为 2，表示写入内存的数据是 2；LD3（即 reg\_write\_en 信号）暗，说明指令不会回写寄存器

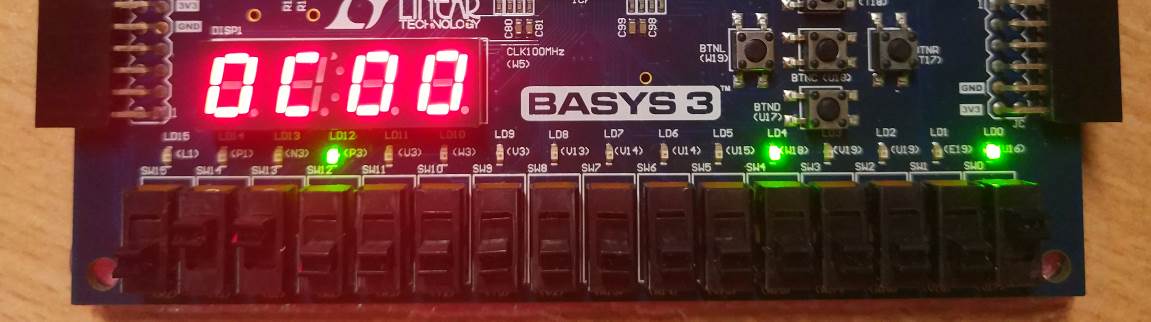


图 24：第 11 条指令在 MEM 阶段，可以看到 ALU 的运算结果为 0x0C，说写入数据存储器的单元地址是 0x0C。

下图是第 12 条指令的执行信息：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| pc | next\_pc | 指令 | mem\_read | mem\_addr |
| 54 | 58 | lw $13,4($1) | 2 | 4+8=12 |



图 25：第 12 条指令在 MEM 阶段；当前指令的地址是 0x54，下一条指令的地址是 0x58；可见 LD3（即寄存器回写信号为高电平）亮，说明 lw 指令会回写寄存器



图 26：第 12 条指令在 MEM 阶段，可见数码管高两位显示了 ALU 的运算结果是 0x0C，说明 lw 指令操作的数据存储器单元是 0x12；数码管低两位显示了数据存储器读取到的数据为 0x02

下表是第 13 条指令的关键信息：

|  |  |  |  |
| --- | --- | --- | --- |
| pc | next\_pc | 指令 | jump |
| 58 | 20 | jr $31 | 2 |



图 27：第 13 条指令在 ID 阶段，可以看到下一条指令的地址为 0x20；LD1（即 fetch\_rw 信号）亮，说明该指令将会发生跳转；LD3（即 reg\_write\_en 信号）暗，说明该跳转指令不会回写存储器（和 jal 指令不同）；LD7~6（即 jump 信号）为 2'b10，说明该指令是跳转到寄存器类型的指令

* + - 1. **实验心得**

经过本次的多周期 CPU 设计，我学会了如何规划各个时序电路在时钟周期内部的触发时机、组合电路和时序电路和混合使用方法。首先各个阶段的更新都应该在时钟的上升沿时触发，这样每个执行周期都恰好占用一整个时钟周期，一方面流水线 CPU 也是这么干的（也必须这么干，必须要统一调度），另一方面也方便时序图的查看和写板的检查。这意味着写板后我们可以按一次按钮变化一次 CPU 阶段从而方便检查。

那么如何实现每个周期在时钟的上升沿才触发计算呢？经过仔细思考，我发现如果要在时钟的上升沿更新 PC 寄存器，那么 IR 寄存器应该是在 IF 阶段的下降沿触发更新还是在 ID 阶段的上升沿触发更新呢？我思考后认为应当在 ID 阶段的上升沿更新成指令的值。这是因为 ID 阶段的译码器（控制单元）实际上是组合逻辑电路，如果 IR 寄存器在 IF 阶段的下降沿触发了更新，那么控制信号就提前到 IF 阶段的下降沿就会计算出来。而这么做的不好之处在于 IF 阶段的指令存储器读写速度在实际情况下会稍慢一些，要花费整个时钟周期来完成指令的读取，也就是说如果 IR 寄存器在 IF 阶段的下降沿更新实际上可能导致结果的谬误。同时因为 IR 寄存器在 ID 阶段的上升沿更新，译码器又是组合电路，因此在 ID 阶段的上升沿后才会完成译码，从而区别开来了 ID 阶段。

寄存器组的回写时机是一个难点。由于 jal 指令的回写被规定成在 ID 阶段就进行，如果要回写寄存器组，必须在译码完成后才能进行，也就是说留给写回寄存器的时机仅剩 ID 阶段的下降沿。而又因此寄存器组在 WB 阶段的回写也被跟着调整成在时钟下降沿时触发。也就是说这个和计算机组成原理的理论不符，理论（流水线模型下）寄存器读写操作应该先写后读，这是因为寄存器的读写操作仅需时钟周期的一半（速度相比 ALU 的计算、取指要更快）。而寄存器的先写后读能保证读写不冲突从而确保数据的稳定性。不过由于这里是多周期的 CPU 而不是流水线 CPU，因此寄存器的读写操作并不会同时发生，所以数据的稳定性可以保证。

我将多周期新加入的 4 个划分阶段的寄存器的时钟触发沿都统一成了时钟的上升沿，这是因为流水线寄存器的更新也都在时钟的上升沿触发，这样多周期的 CPU 代码就更易于修改成流水线 CPU（因为流水线寄存器组的更新必须要统一同一个时刻一起更新）；另一方面，这些寄存器的作用也相当于强制分开处理器的 5 个阶段（IR 的更新表示 ID 阶段的进入、ADR/BDR 的更新意味着 EXE 阶段的进入、ALUoutDR 的更新意味着 MEM 阶段的进入，DBDR 的更新意味着 WB 阶段的进入）。对于 ID 和 EXE 阶段，译码器、寄存器组读取、ALU、跳转目标地址的计算都是组合电路而非时序电路，寄存器的更新就将这些阶段强制分离；而对于 MEM/WB 阶段，数据存储器、寄存器组的写操作是时钟相关的，对于这些指令，我将触发时机设置成了时钟的下降沿以便模块能从 ALUoutDR 和 DBDR 拿到数据。毕竟如果将数据存储器和寄存器组的触发时机设置成上升沿，那么输入将拿到 ALUoutDR 和 DBDR 在上一次时钟周期的值而不是这一次上升沿之后更新成的值。

经过这次实验，我再次认识到了拥有良好的编码习惯、遵守代码规范的重要性。Verilog 代码规范要求调用每个模块时的端口列表应该标注模块的端口名，这样就可以避免因为漏接端口而导致后面的端口全部错位而导致的问题。这种问题表现成复位前可能这些端口是 0，但是复位后就变成高阻态的问题；亦或是某些端口的值不对。

本次实验在完成了代码的编写之后一次性通过了仿真波形图的检查和写板的检查。这也归功于严格遵守代码规范，不使用可能综合失败的语法特性或者是其他代码。同时，也因为编写代码时认真思考了整个 CPU 的运行，仔细考虑了每个阶段每个模块对于每条指令会发生的所有情况并一一进行了解决。还因为单周期 CPU 的代码结构化良好，一开始就考虑到了多周期 CPU 的实现，从而大大降低了多周期 CPU 的代码修改难度。才可以实现一次性的代码测试通过。这说明代码的模块化十分重要。

* + - 1. **学期实验心得**

本次学期完成了课程安排的三个实验：MIPS 汇编程序的编写与运行、单周期 CPU 的设计与验证、多周期 CPU 的设计与验证（以及 MIPS 汇编程序的汇编器的实现）。这次实验课程内容上至软件、下至硬件，通过学习设计，了解了 MIPS 体系架构的基本组成、概念，并且加深了理论课相关知识的理解。而且借由 MIPS CPU 的设计，理解了 C 语言的诸多底层特性，比如指针地址、结构体内存分配方式；也知道了 CPU 设计的困难、复杂性，知道了我国目前芯片设计的困难，也希望我国的芯片产业能够获得更好的发展。

本学期第一个汇编实验让我学会了简单的 MIPS 汇编程序的设计，熟悉了 MIPS 基本指令的使用方法。通过先写 C 语言程序再写汇编程序，我了解了编译器到底在做什么工作。并且熟悉了 PCSim 软件的使用方法，同时知道了 MIPS 的异常机制，还经过艰苦的调试，深刻地记住了访存必须对齐，以及 int 类型的数组必须 4 字节对齐地声明。

本学期的第二个，单周期 CPU 实验的过程让我了解了如何编写一个简单的单周期 MIPS CPU。让我对 CPU 的工作机制拥有了一个深刻的印象。同时我也进一步地了解了 Verilog 程序的仿真与综合，学会了 FPGA 程序的编写。在踩了了很多坑之后，我了解了如何编写 Verilog 代码才可以成功地通过仿真，并且在 FPGA 学习板上正确地运行并计算结果。比如 clk 信号必须是与时钟同步的信号；always 语句的敏感信号表不可以时钟沿触发和无条件触发混合；always 无条件触发的时候使用 \* 而不是一一列出避免漏写敏感信号；always 的时钟敏感时的代码比如先判断是否是复位信号而不是先写逻辑，否则综合器默认第一条 if 的 then 语句执行的是复位从而导致综合预期不符的问题；always 的 case 语句必须将所有的情况都列出来并且必须写 default 子句，否则如果信号为 X 或者 Z 时因为没有匹配的值从而引发问题，或者缺漏情况导致原本应该是组合电路，却产生了一个锁存器耗费大量的硬件资源。最后在验证的时候我充分利用了 Basys 3 学习板上的硬件资源，通过利用 LED 灯同时显示更多的信号来加快我定位问题的速度。

本学期的第三个，多周期 CPU 实验的过程让我更加深入地了解了如何编写时序电路，以及将多个大延迟切割成几个小延迟带来的电路的稳定性。在这次实验，我深入知道了如何查看电路的时序图，熟悉了 Vivado 软件的仿真功能、以及更为强大的断点、单步调试功能。通过这些便捷、功能强大的调试工具，我才能快速地完成 CPU 代码的编写与验证

最后在课程的设置上，我很满意这次课程的学习。上课的时候老师认真详细地向我们讲解 CPU 的数据通路，并且提醒我们设计过程中的可能的坑，从而加快我们做实验的速度。检查的时候 TA 还会详细地询问我们对数据通路的理解程度，保证我们对 CPU 的体系结构拥有充分的理解。总之，这次的课程学习我收获了很多。