**北京科技大学 计算机与通信工程学院**

**课程设计报告**

**课程名称**： 计算机组成原理课程设计

**学生姓名**： 刘旭阳

**专 业**： 物联网工程

**班 级**： 物联191

**学 号**： 41927039

**指导教师**： 张磊

**报告成绩**：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**实验地点**： 机电楼304

**实验时间**：2021年8月1日----2021年12月20日

**北京科技大学实验报告**

学院：计算机与通信工程学院 专业：物联网工程 班级：物联191

姓名：刘旭阳 学号：41927039 实验日期：2021年10月20日

**一、课设目的与要求**

* 学会处理器的设计方法：单周期/流水线。
* 掌握处理器设计过程中指令扩展的方法。
* 能够运用现代工具独立实现一个完整的处理器。
* 了解处理器功能测试的方法：仿真测试及FPGA测试。
* 了解处理器性能测试的方法。
* 计算机系统观的建立，对所设计的处理器在整个计算机系统中的位置有所了解。

**二、实验设备（环境）及要求**

龙芯实验箱一体化实验平台/CG平台/流水线处理器关键技术虚拟仿真实验平台

OS：Win10 64位

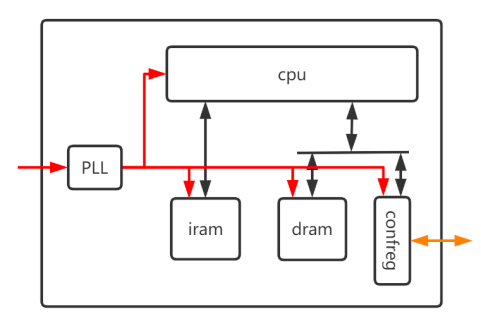
Software：Vivado2018.3开发工具

VirtualBox虚拟机+Ubuntu16.04.6（正文用五号宋体）

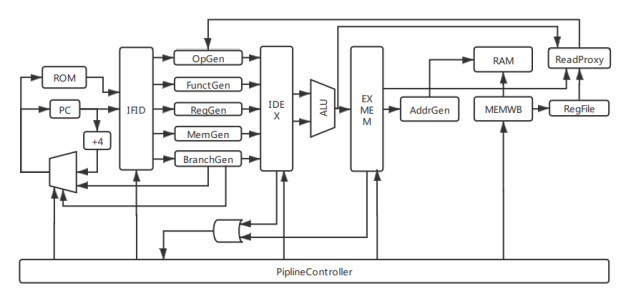
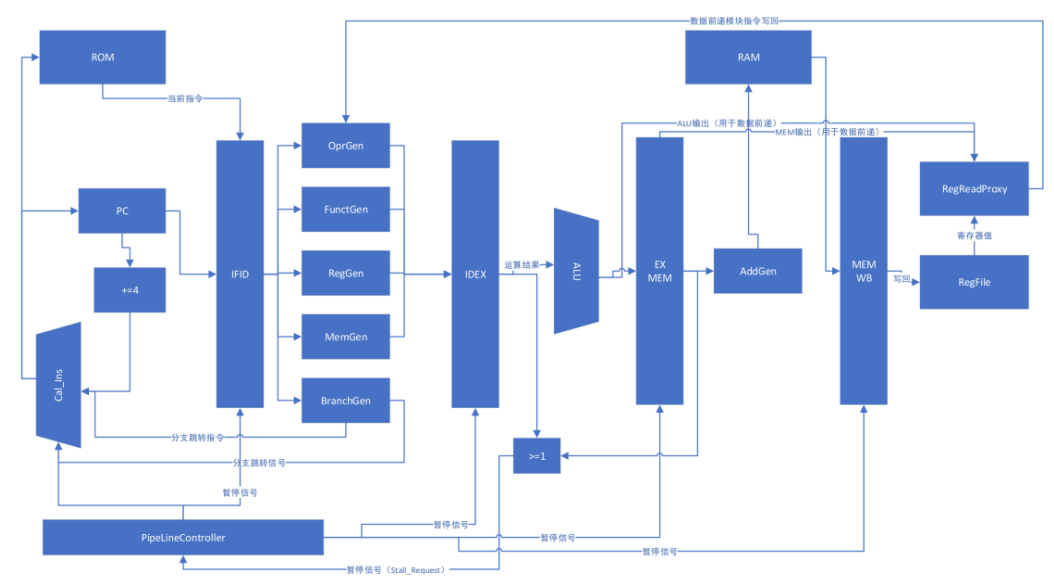
**三、设计过程与结果分析**

**Part 1 验证实验，代码阅读分析**

**1 TinyMIPS总体结构框图**



总体结构框图

****

总体数据通路图

**2 ADDIU指令（典型指令）的设计过程**

**2.1指令格式**

31 26 25 21 20 16 15 0

|  |  |  |  |
| --- | --- | --- | --- |
| 001001 | Rs | Rt | Imm |

ADDIU为典型的I型指令，三个操作数分别为源寄存器号、目的寄存器号和立即数。该指令先将Rs的值与有符号扩展的Imm相加，在将结果送入Rt。

**2.2实现代码**

**2.2.1 操作码宏定义**

|  |
| --- |
| ‘define OP\_ADDIU 6'b001001 |

**2.2.2 ID-RegGen产生寄存器访问信号**

|  |
| --- |
| case (op)        // arithmetic & logic (immediate)        `OP\_ADDIU,        // memory accessing        `OP\_LB, `OP\_LW, `OP\_LH, `OP\_LBU: begin          reg\_read\_en\_1 <= 1;          reg\_read\_en\_2 <= 0;          //只需一个寄存器读信号，因为ADDIU的其中一个源操作数为立即数          reg\_addr\_1 <= rs;          reg\_addr\_2 <= 0;  end |

**2.2.3 ID-FunctGen产生功能码**

|  |
| --- |
| case (op)        `OP\_SPECIAL: funct <= funct\_in;        `OP\_LUI: funct <= `FUNCT\_OR;        `OP\_LB, `OP\_LBU, `OP\_LW,        `OP\_LH, `OP\_SB, `OP\_SW, `OP\_ADDIU: funct <= `FUNCT\_ADDU;        //ADDIU在EX段执行加法运算，因此生成功能码为FUNCT\_ADDU        `OP\_JAL: funct <= `FUNCT\_OR;        default: funct <= `FUNCT\_NOP;  endcase |

**2.2.4 ID-OperandGen取操作数**

|  |
| --- |
| wire[`DATA\_BUS] sign\_ext\_imm = {{16{imm[15]}}, imm};  case (op)//取操作数1      // immediate      `OP\_ADDIU, `OP\_LUI,      // memory accessing      `OP\_LB, `OP\_LW,`OP\_LH, `OP\_LBU, `OP\_SB, `OP\_SW: begin          operand\_1 <= reg\_data\_1;        end  case (op)//取操作数2        `OP\_LUI: begin          operand\_2 <= zero\_ext\_imm\_hi;        end        // arithmetic & logic (immediate)        `OP\_ADDIU,        // memory accessing        `OP\_LB, `OP\_LW,`OP\_LH, `OP\_LBU, `OP\_SB, `OP\_SW: begin          operand\_2 <= sign\_ext\_imm;  end |

**2.2.5 EX指令执行**

|  |
| --- |
| wire[`DATA\_BUS] operand\_2\_mux =        (funct == `FUNCT\_SUBU || funct == `FUNCT\_SLT)          ? (~operand\_2) + 1 : operand\_2;  //求出操作数2的补码  wire[`DATA\_BUS] result\_sum = operand\_1 + operand\_2\_mux;  //将求补码后的操作数2与操作数1直接相加  case (funct)        // jump with link & logic        `FUNCT\_JALR, `FUNCT\_OR: result <= operand\_1 | operand\_2;        `FUNCT\_AND: result <= operand\_1 & operand\_2;        `FUNCT\_XOR: result <= operand\_1 ^ operand\_2;        // comparison  `FUNCT\_SLT, `FUNCT\_SLTU: result <= {31'b0, operand\_1\_lt\_operand\_2};     // arithmetic        `FUNCT\_ADDU, `FUNCT\_SUBU,`FUNCT\_ADD: result <= result\_sum;         //结果输出 |

**2.2.6 WB写回**

|  |
| --- |
| always @(posedge clk) begin      if (rst) begin//寄存器堆复位        for (i = 0; i < 32; i = i + 1) begin          registers[i] <= 0;        end      end      else if (write\_en && |write\_addr) begin        registers[write\_addr] <= write\_data;  //写使能与目标寄存器合法（不为0）时，写回      end  end |

**2.3仿真测试**

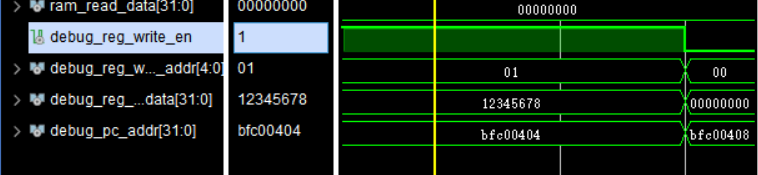
一：

测试代码：

|  |
| --- |
| lui $8, 0x1234  addiu $8,$8,0x5678 |

仿真波形：

****

****

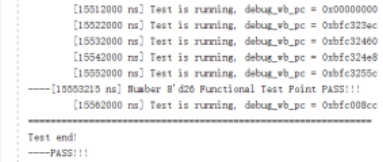
仿真结果分析：

如图，可以看到，我们的cpu首先执行LUI指令，在8寄存器的高16位写入0x1234，由仿真波形可以看出LUI指令执行后8号寄存器的确变成了0x12340000。再将8号寄存器与0x5678相加，结果送入8号寄存器，同样可以看到，这一步执行后，8号寄存器的值变为0x12345678，结果与预期相符，验证了ADDIU指令的正确性。

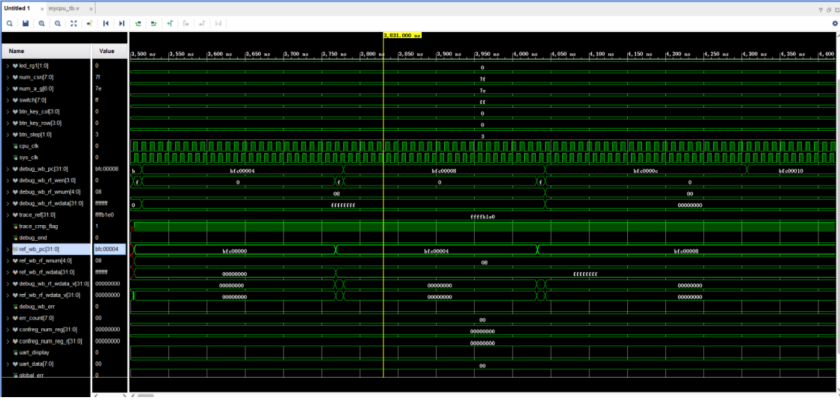
二：

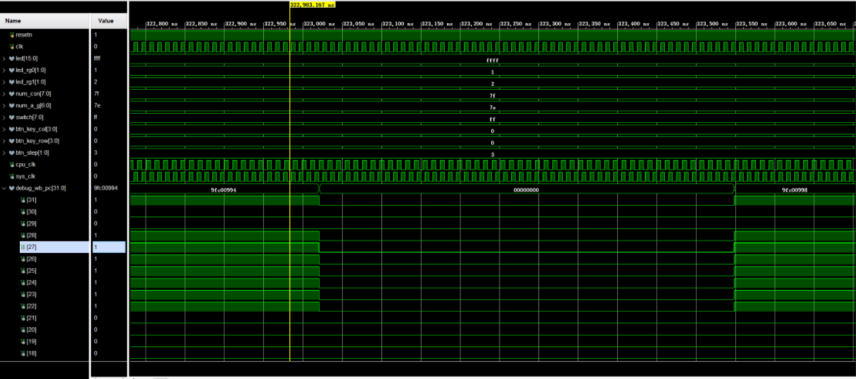
仿真测试代码文件mycpu\_tb,其中是对自设计 CPU 的 22 条指令以及 4 个分支 跳转指令的测试过程，通过测试会对测试结果进行 Pass，否则报错 Error。

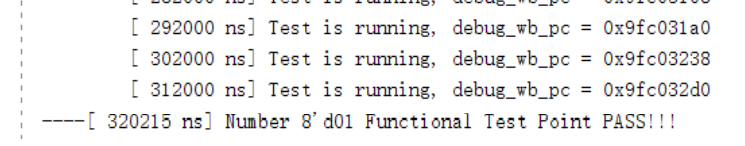
指令测试过程： 测试全部通过。



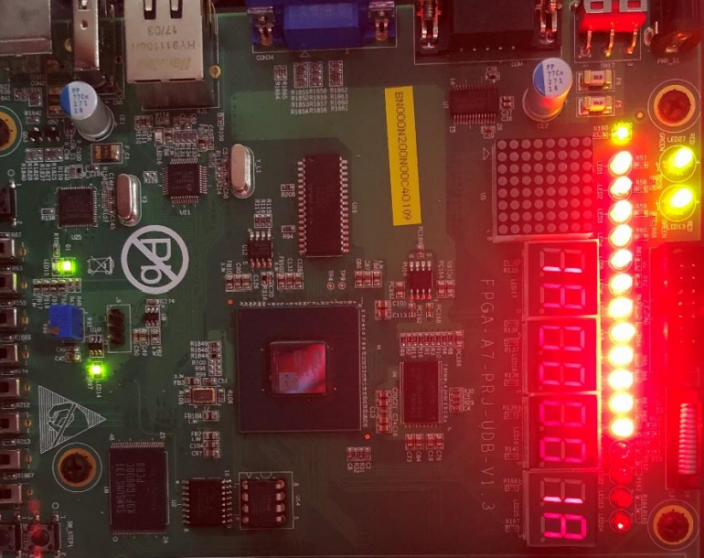
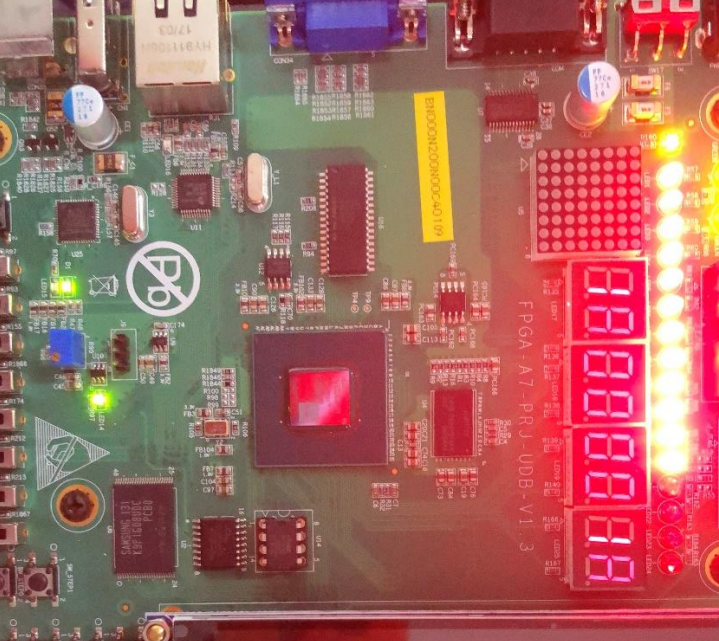
波形图（第一个指令为 ADDU）







**2.3 FPGA测试**

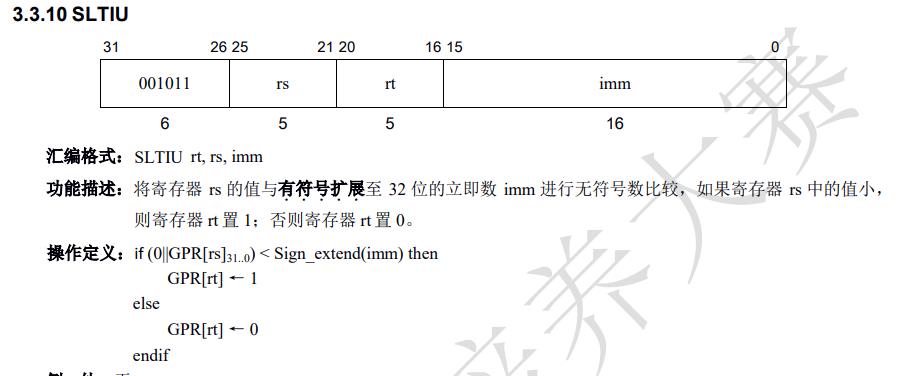


**Part 2 扩展实验（在TinyMIPS\_Extend基础上扩展指令，运算类、跳转类、访存类均有）**

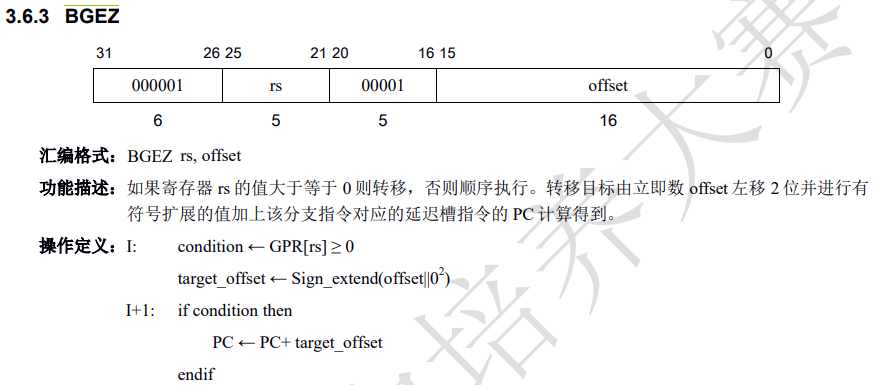
**总述:共实现了3条指令扩展，分别是：SLTIU，BGEZ，SH**

**1 指令格式**

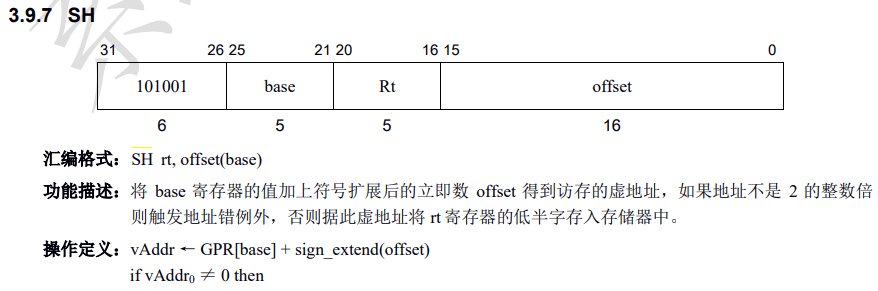
SLTIU指令：

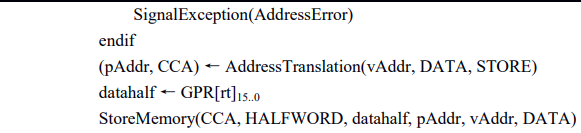


BGEZ指令：



SH指令：





**2分析指令功能及执行过程，画出数据通路图**

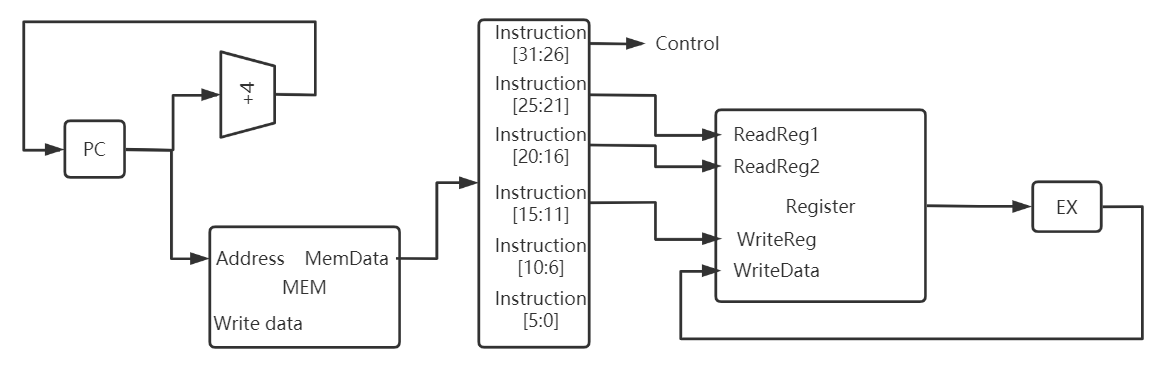
**2.1 SLTIU指令**

功能描述：将寄存器 rs 的值与有符号扩展至32位的立即数imm进行无符号数比较，如果寄存器 rs 中的值小，则寄存器 rt 置 1；否则寄存器 rt 置 0。

执行过程：

1. 取指：从ROM中取出SLTIU指令
2. 译码：Rs读使能，Rt写使能；生成SLTU功能码；取出操作数1，并将立即数有符号扩展后作为操作数2
3. 执行：根据SLTU功能码进行对应操作，结果送入下一阶段
4. 访存：此指令不需要访问内存
5. 回写：将Rt寄存器号与运算结果送入寄存器堆模块，写入对应寄存器

数据通路图：



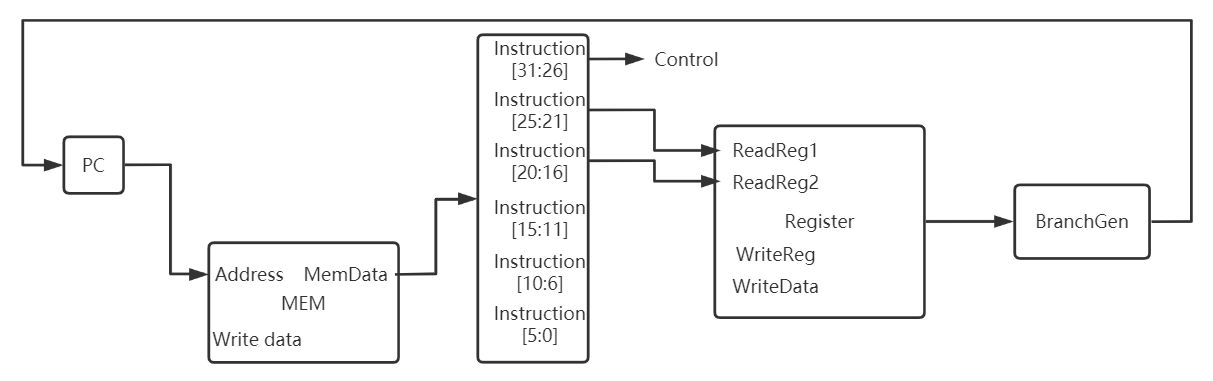
**2.2 BGEZ指令**

功能描述：如果寄存器 rs 的值大于等于 0 则转移，否则顺序执行。转移目标由立即数 offset 左移 2 位并进行有符号扩展的值加上该分支指令对应的延迟槽指令的 PC 计算得到。

执行过程：

1. 取指：从ROM中取出BGEZ指令
2. 译码：Rs读使能，Rt读使能；根据操作数与0的大小关系确定branch是否使能
3. 此后，下一阶段直接在PC中分支即可

数据通路图：



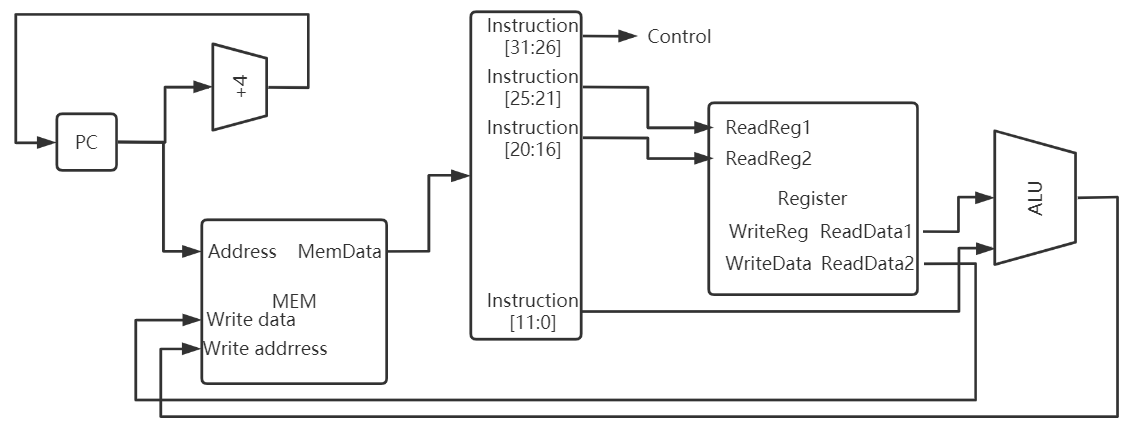
**2.3 SH指令**

功能描述：将 base 寄存器的值加上符号扩展后的立即数 offset 得到访存的虚地址，如果地址不是 2 的整数倍则触发地址错例外，否则据此虚地址将 rt 寄存器的低半字存入存储器中。

执行过程：

1. 取指：从ROM中取出SH指令
2. 译码：Base读使能，Rt读使能；生成FUNCT\_ADDU功能码；从Base中取出操作数1，将Offset符号扩展后的值作为操作数2；
3. 执行：根据FUNCT\_ADDU功能码进行加法操作，结果送入下一阶段
4. 访存：内存写使能；执行阶段的加和即为内存写地址；操作数1低16位为待写数据；写成功输出2'b00,否则输出2'b10
5. 回写：如果是00，正常写回低16位，如果是10，需要向前提16位，然后开始写入。

数据通路图：



**3 代码实现**

**3.1 SLTIU**

1. 在opcode文件中添加定义：

|  |
| --- |
| ~define FUNCT\_SLTIU 6’b001011 |

2. 在ID模块的RegGen模块下添加代码：

|  |
| --- |
| //因为SLTIU指令只需要寄存器rs的值，所以在此处需要读寄存器时，只需要读第一个寄存器就可以了，rt寄存器不需要读取。        `OP\_SLTIU : begin          reg\_read\_en\_1 <= 1;          reg\_read\_en\_2 <= 0;          reg\_addr\_1 <= rs;          reg\_addr\_2 <= 0;        end  //SLTIU指令是经过比较后，将1或者0置入rt寄存器，所以此处将写信号置1，并且写入数据的地址置成rt寄存器。        `OP\_SLTIU: begin          reg\_write\_en <= 1;          reg\_write\_addr <= rt;        end |

3. 在ID模块的FunctGen模块下添加代码：

|  |
| --- |
| //由于原本实验中已经给出SLTU的功能实现方法，而SLTIU在功能实现方面与SLTU相同，因此可以在此直接调用SLTU的功能。  `OP\_SLTIU: funct <= `FUNCT\_SLTU;        default: funct <= `FUNCT\_NOP;      endcase    end |

4. 在ID模块下的OperandGen模块下添加代码：

|  |
| --- |
| //SLTIU指令的操作数1是寄存器rs中的值，此处将rs中的值赋值给操作数1。        `OP\_SLTIU: begin          operand\_1 <= reg\_data\_1;        end  //操作数2是经过符号位扩展后的立即数，即 wire[`DATA\_BUS] sign\_ext\_imm = {{16{imm[15]}}, imm}。   `OP\_SLTIU: begin          operand\_2 <= sign\_ext\_imm;        end |

5. 在EX模块下编写SLTU调用的代码：

|  |
| --- |
| //此处为SLTU的执行代码（SLTIU指令和其相同）。   wire operand\_1\_lt\_operand\_2 = funct == `FUNCT\_SLT ?          // op1 is negative & op2 is positive          ((operand\_1[31] && !operand\_2[31]) ||            // op1 & op2 is positive, op1 - op2 is negative            (!operand\_1[31] && !operand\_2[31] && result\_sum[31]) ||            // op1 & op2 is negative, op1 - op2 is negative            (operand\_1[31] && operand\_2[31] && result\_sum[31]))        : (operand\_1 < operand\_2);  `FUNCT\_SLT, `FUNCT\_SLTU: result <= {31'b0, operand\_1\_lt\_operand\_2}; |

**3.2 BGEZ**

1. 在opcode文件中添加定义：

|  |
| --- |
| ~define FUNCT\_BGEZ      6’b000001 |

2. 在ID模块下的RegGen模块下添加代码：

|  |
| --- |
| //BGEZ指令需要读取rs和rt寄存器中的值。   `OP\_BGEZ：begin          reg\_read\_en\_1 <= 1;          reg\_read\_en\_2 <= 1;          reg\_addr\_1 <= rs;          reg\_addr\_2 <= rt;        end |

3. 在ID模块下的BranchGen模块下添加代码：

|  |
| --- |
| //BGEZ指令需要进行有符号位的比较，所以使用到了$signed（）这个将括号内的数转换成有符号的代码，其中的sign\_ext\_imm\_sll2是源代码中已经给出的wire[`DATA\_BUS] sign\_ext\_imm\_sll2 = {{14{inst[15]}}, inst[15:0], 2'b00};也就是有符号的扩展立即数的代码。  `OP\_BGEZ: begin         if(inst[20:16] ==  5'b00001) begin           if ($signed(reg\_data\_1) >= $signed(reg\_data\_2)) begin            branch\_flag <= 1;            branch\_addr <= addr\_plus\_4 + sign\_ext\_imm\_sll2;           end           else begin            branch\_flag <= 0;            branch\_addr <= 0;           end          end        end |

**3.3 SH**

1. 在opcode文件中添加定义：

|  |
| --- |
| ~define FUNCT\_SH        6’b101001 |

2. 在RegGen模块下添加代码：

|  |
| --- |
| //SH指令需要调用rs和rt寄存器的值，所以读取使能全部设为1。   `OP\_SH,: begin          reg\_read\_en\_1 <= 1;          reg\_read\_en\_2 <= 1;          reg\_addr\_1 <= rs;          reg\_addr\_2 <= rt;        end |

3. 在FunctGen模块下添加代码：

|  |
| --- |
| //SH指令存在了加操作，因此在此处调用已经写好的ADDU加操作指令。   `OP\_SH: funct <= `FUNCT\_ADDU; |

4. 在OperandGen模块中添加代码：

|  |
| --- |
| //SH指令将rs寄存器中的值设为操作数1，将符号位扩展后的立即数设为操作数2。  wire[`DATA\_BUS] sign\_ext\_imm = {{16{imm[15]}}, imm};  `OP\_SH: begin          operand\_1 <= reg\_data\_1;        end  `OP\_SH: begin          operand\_2 <= sign\_ext\_imm;        end |

5. 在MemGen模块下添加代码：

|  |
| --- |
| //首先给出写的使能信号。   `OP\_SH: mem\_write\_flag <= 1;  //因为存储写入时，SH指令是半字，所以后两位置1，表明指令需要的数据是低位。  `OP\_SH: mem\_sel <= 4'b0011;  //此处给出写回的数据。  `OP\_SH: mem\_write\_data <= reg\_data\_2; |

6. 在MEM模块下添加代码：

|  |
| --- |
| //此处是得到写信号，然后判断写回的是哪两位，因为是半字，所以只有两种情况，00时读取后低位，10时读取高位。  if (mem\_sel\_in == 4'b0011) begin   // half word          case (address[1:0])            2'b00: ram\_write\_sel <= 4'b0011;            2'b10: ram\_write\_sel <= 4'b1100;            default: ram\_write\_sel <= 4'b0000;          endcase        end  //此处给出的是写回的数据信号，如果是00，正常写回低16位，如果是10，需要向前提16位，然后开始写入。   if (mem\_sel\_in == 4'b0011) begin          case (address[1:0])            2'b00: ram\_write\_data <= mem\_write\_data;            2'b10: ram\_write\_data <= mem\_write\_data << 16;          endcase        end |

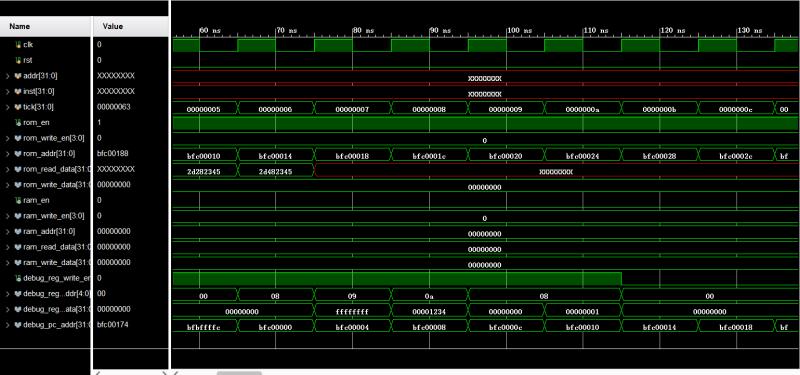
**4 指令功能测试**（需有测试激励，仿真波形及分析，测试部分可将所有指令放在一起做。）

**4.1 SLTIU**

用于测试的指令段：

|  |
| --- |
| addu    $8,$0,$0  addiu   $9,$0,0xFFFF  addiu   $10,$0,0x1234  sltiu   $8,$9,0x2345  sltiu   $8,$10,0x2345 |

仿真波形：



分析：

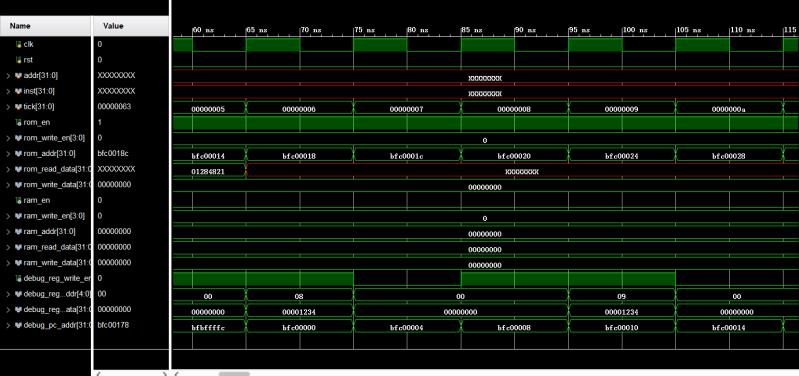
从图中可以看出，8号寄存器的值先为0，后为1，因为9号寄存器中的值经过符号位扩展后大于0x2345，所以8号寄存器值为0，又因为10号寄存器里的值经过符号扩展后小于0x2345，所以8号寄存器值为1，符合指令预期结果。

**4.2 BGEZ**

用于测试的指令段：

|  |
| --- |
| addiu   $8,$0,0x1234      bgez    $8,label      nop      addu    $9,$8,$8  label:      addu    $9,$9,$8 |

仿真波形：



分析：

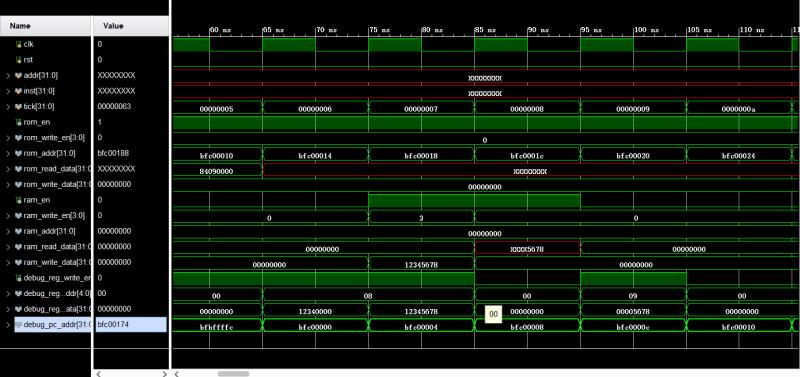
从图中可以看出，9号寄存器里的值为0x1234，说明转移指令跳转成功，否则根据代码，若未跳转则9号寄存器内值应为3倍的8号寄存器值。结果证明程序运行成功。

**4.3 SH**

用于测试的指令段：

|  |
| --- |
| lui $8,0x1234  addiu   $8,$8,0x5678  sh  $8,0x0($0)  lh  $9,0x0($0) |

仿真波形：



分析：

从上图可以看出，程序成功将0x1234导入8号寄存器的高16位，将0x5678导入8号寄存器的低16位，调用LH指令显示存入的数值，可以看见存入的数值为0x5678，存入了后半个字，所以证明SH指令运行成功。

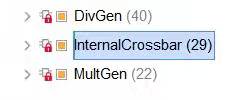
**Part 3 创新实验（演示系统介绍）实验分组编号：20（填写实验箱分组中组号）**

**系统概述：实现了什么功能，扩展了哪些内容，如何测试，有何发现。**

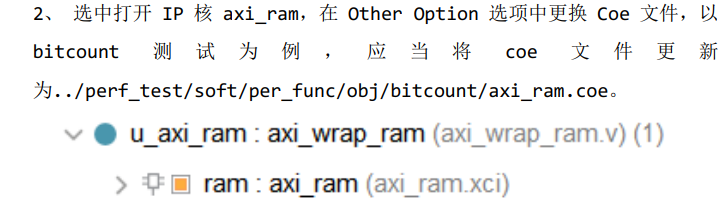
**如果是进行了性能测试，需说明性能测试过程和结果。**

**性能测试：**

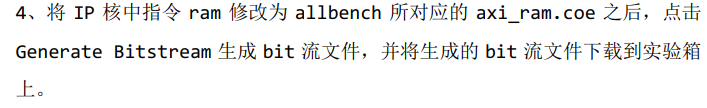
在通过全部89个测试点后，升级vivado版本，与功能测试类似，首先利用 Vivado 打 开perf\_test/soc\_axi\_perf/run\_vivado/mycpu\_prj1/mycpu.xpr 创建工程， 并在工程中导入 cpu 文件。



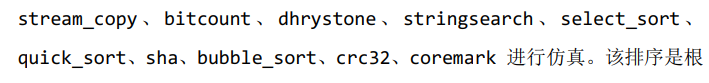
更新ip核，











对十个测试文件重复以上操作，记录16 进制的 SoC count 即 cpu 外部时钟的周期数

后来知道可以（**首先使用 allbench 中的 axi\_ram.coe 直接进行综合上板测试，根据 上板测试结果再单独跑错误性能测试的仿真。**）

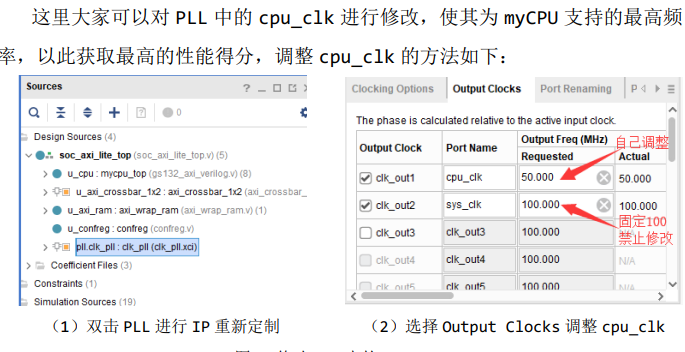


****

由于没有进行性能改进，所以性能分为0.923

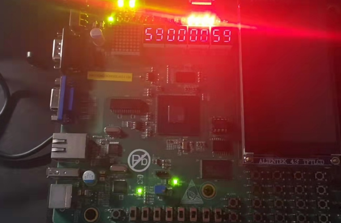
也可以修改mycpu的频率已获取更高性能分

（把频率调高即可，感觉没有太大难度，没再做）



**功能点测试+性能测试：**

系统概述：实现了所有异常指令，并通过全部89个测试点



1.1实现思路

TinyMIPS异常处理的实现思路是：为了实现精确异常并且按照指令的执行顺序对异常进行处理，在流水线的各个阶段收集异常信息，并传递到流水线的访存阶段，在访存阶段统一处理异常信息。流水线各个阶段需要收集的异常信息如下：

1. 在流水线的译码阶段判断是否是系统调用异常、是否是返回指令、无效指令。
2. 在流水线执行阶段判断是否有溢出异常。
3. 在流水线访存阶段检查是否有中断发生。

在流水线访存阶段，处理器将结合协处理器CP0中相关寄存器的值，判断异常是否处理，如果需要处理，那么转移到EPC寄存器保存的地址处，同时，也要清除流水线上写阶段外的全部信息，修改协处理器CP0中相关寄存器的值。

1.2具体实现

1. 在流水线译码阶段ID模块中判断是否是系统调用指令syscall、断点例外指令break、异常返回指令eret、无效指令，将这些信息传到执行阶段。与此同时，将指令地址也传递到执行阶段。
2. 流水线执行EX模块会进一步判断是否有溢出异常。并将该信息与译码阶段给出的异常信息进行融合，然后将其传到访存阶段。同时，将指令地址传入访存阶段，是否位于延迟槽的delayslot\_flag也被传递到访存阶段。
3. 流水线访存阶段MEM模块会依据传递过来的异常信息、Cause寄存器的值、Status寄存器的值综合判断是否需要处理异常，例如地址错例外、是否能产生中断。最终的异常信息传递给PipelineController模块，PipelineController模块据此信号判断是否出现异常或例外，出现就将flush置为1送到PC和各阶段、各寄存器中，消除异常指令以及其后的指令，同时将生成异常处理地址送入pc。
4. 如果要处理异常，还需要修改协处理器CP0中EPC、Status、Cause等寄存器的值，所以访存阶段给出的最终的异常信息送入CP0模块，同时送入的还有发生异常的指令是否在延迟槽中、发生异常的指令的地址。CP0模块根据这些信息修改相应寄存器的值。

**下面详细说明加减法整型溢出的异常指令的扩展**

1. 功能简介

在第二部分的指令扩展部分，ADD，SUB等R型运算指令的功能介绍中，有触发整型溢出例外的情况，而在原来的工程中并没有实现对这种异常情况的处理。本次小组创新实验，主要完成了加减法整型溢出的异常指令的扩展。

1. 测试方法

编写指令序列和测试激励进行仿真，通过观察仿真波形来验证异常指令扩展的正确性。

1. 探索发现

深入理解了CPU是如何在收到“清除”指令之后对后续指令进行清除的。更加清楚了硬件与软件的边界，CPU与操作系统的界限。对整个流水线CPU有了更加清晰的认识。

**1 系统总体架构图**

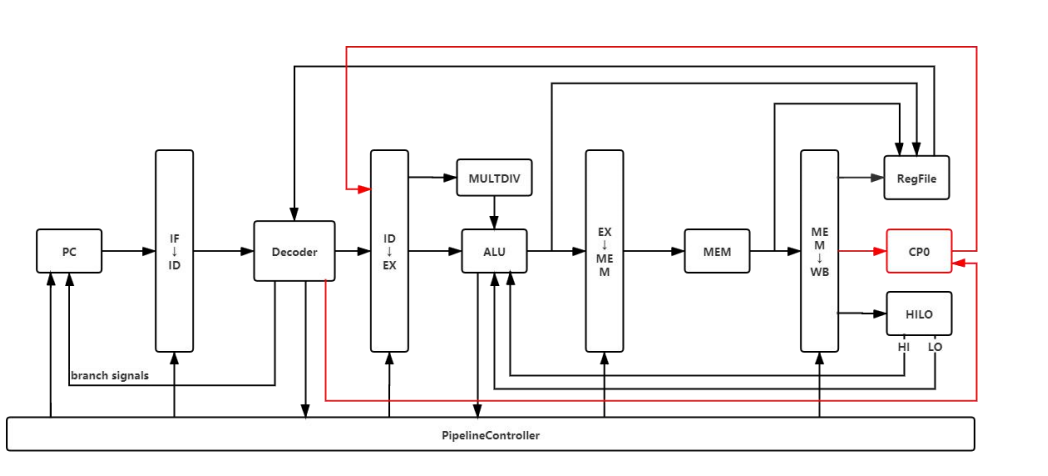
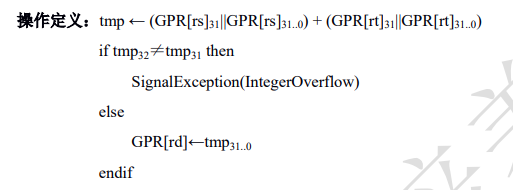


图 6

**2 设计过程**

1. 信号产生

整型溢出异常信号overflow\_flag，由加减法的溢出运算产生，而加减法的计算在ex级。因此，该信号应该产生于ex级的加减计算过程中。那么，如何判定一个加减法运算产生了溢出？ADD指令的功能描述中，对溢出的判定方法有很清楚的描述，如下图：



由此，ex级作如下改动

|  |
| --- |
| wire[32:0] tmp = {operand\_1[31],operand\_1} + {operand\_2\_mux[31],operand\_2\_mux};  //按照公式，将两个操作数的符号位扩展一位并进行加操作，结果赋给定义的变量tmp  assign overflow\_flag\_out = tmp[32] ^ tmp[31];  //如果tmp的最高位和次高位相同，则没有发生溢出，不相同则发生溢出，用异或来计算。 |

1. 信号传递

整型溢出异常信号虽然在ex产生，但是并没有在ex级进行处理，而是由专门的协处理器模块CP0以及pipeline\_controller模块进行处理。由于流水特性，不能直接将信号由ex级传入对应模块，而是应该按照顺序往后传。经过的模块有exmem，CP0, pipline\_controller。我们只要在输入和输出上分别添加一个信号，然后使用assign赋值即可。

1. 信号处理
2. CP0模块

该模块的主要功能就是对异常信号进行处理，该模块也有几个代表CPU状态，中断原因等的变量。异常的产生必然会导致CPU的某些状态，PC值，中断原因等信息的改变，具体代码如下

|  |
| --- |
| always @(posedge clk) begin  if (rst) begin  reg\_status <= 32'h0040ff00;  end  else if (break\_flag || syscall\_flag || overflow\_flag) begin  //如果产生中断，CPU的状态要改变，进而去执行相应的处理程序，这里把状态设为1，说明正常情况下，CPU状态应该为0  reg\_status[1] <= 1;  end |

|  |
| --- |
| always @(posedge clk) begin  if (rst) begin  reg\_cause <= 32'h0;  end  else if (break\_flag || syscall\_flag || overflow\_flag) begin  //产生中断之后，还要区分不同的中断原因，以便执行不同的操作，如何区分不同的中断？这里使用中断码进行区分，不同的中断具有不同的中断码，这里根据不同的中断设置不同的中断码。中断码已经在头文件中设置好了，这里直接拿来用。  reg\_cause[31] <= delayslot\_flag;  reg\_cause[6:2] <= break\_flag ? `CP0\_EXCCODE\_BP :  syscall\_flag ? `CP0\_EXCCODE\_SYS :  overflow\_flag ? `CP0\_EXCCODE\_OV : 0;  end |

|  |
| --- |
| always @(posedge clk) begin  if (rst) begin  reg\_epc <= 32'h0;  end  else if (break\_flag || syscall\_flag || overflow\_flag) begin  //执行完处理程序之后，返回到原来的位置继续执行指令，该操作即将对应的指令地址进行赋值  reg\_epc <= exc\_epc;  end |

1. Pipline\_controller模块

|  |
| --- |
| assign flush = stall\_all ? 0 : (eret\_flag || syscall\_flag || break\_flag || overflow\_flag) ? 1 : 0;  //flush是一个清零信号量，当出现中断时，应该对该中断之后的指令进行清零  else if (syscall\_flag || break\_flag || overflow\_flag) begin  exc\_pc <= 32'hbfc0\_0380;  end  //如果产生中断，则PC应该跳转到对应的处理程序的位置，假设该位置是bfc0\_0380。对于跳转到该位置执行什么样的程序，我们并不关心，那是操作系统应该考虑的问题，并非是CPU设计者。 |

3指令序列测试

|  |
| --- |
| //ADDIU $1,$0,7fff 1号寄存器中存入16位的最大的数  //0010 0100 0000 0001 1110 1111 1111 1111  24 01 7f ff  //SLL $2,$1,16 将1号寄存器中的数据向左移动16位，并存入2号寄存器，此时1号寄存器中的数据是0000 7fff，2号寄存器中的数据是7fff 0000，两个数据相加仍然不会产生溢出  //0000 0000 0000 0001 0001 0100 0000 0000  00 01 14 00  //SLL $1,$1,2 将1号寄存器中的数据再向左移动2位，得到0001 fffe，此时两个寄存器中的数据相加将会产生溢出，达到检测的目的  //0000 0000 0000 0001 0000 1000 1000 0000  00 01 08 80  //ADD $3,$2,$1  //0000 0000 0100 0001 0001 1000 0010 0000  00 41 18 20 |

**3 实验现象及分析**

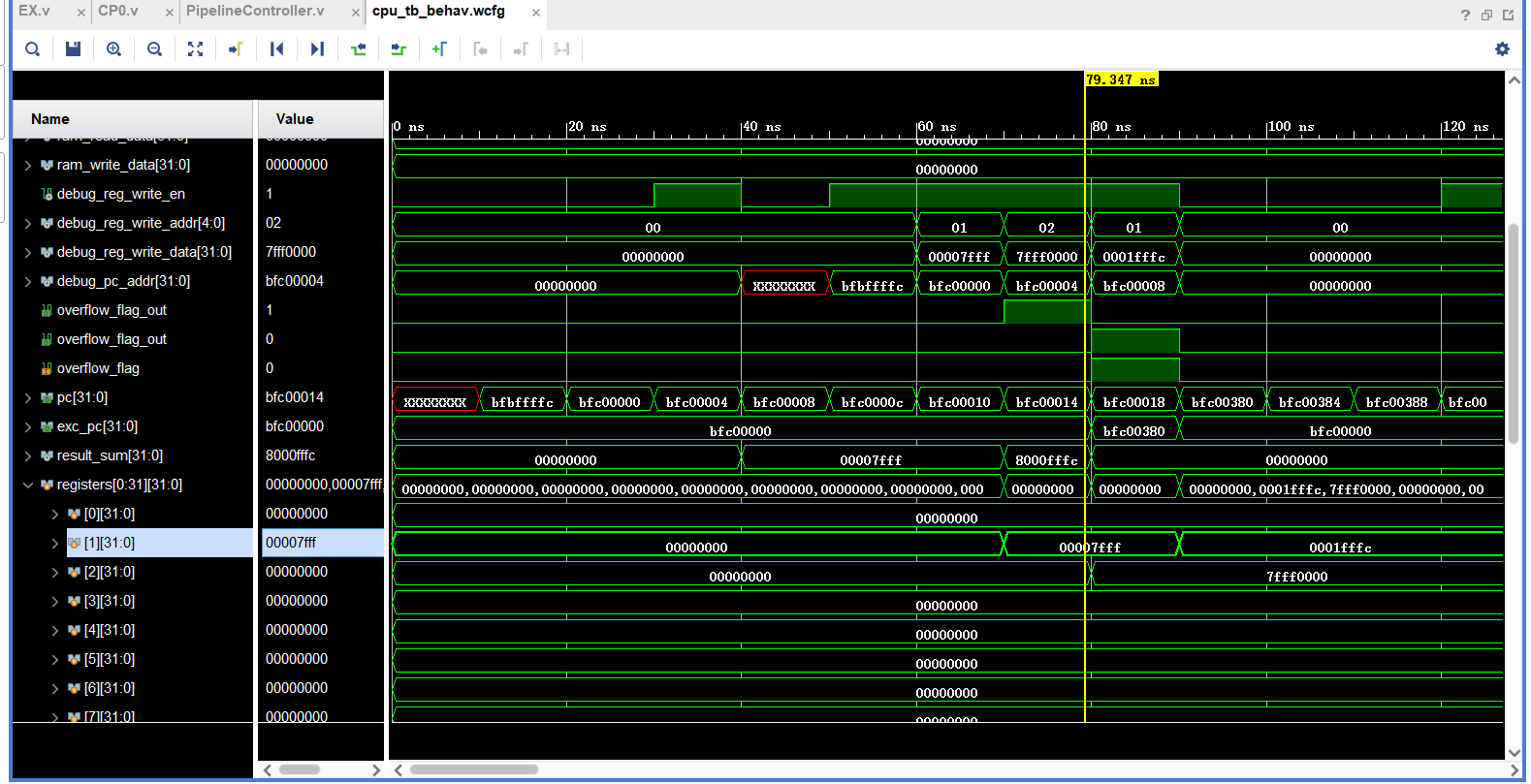


图 7

如上图所示，1号寄存器中已经存入0000 7fff，接着向左移动16位，并存入2号寄存器中，可以看到2号寄存器中的值由0变为7fff 0000，接着1号寄存器中的数据再向左移动两位，数据由0000 7fff变为0001 fffc，符合我们的预期。

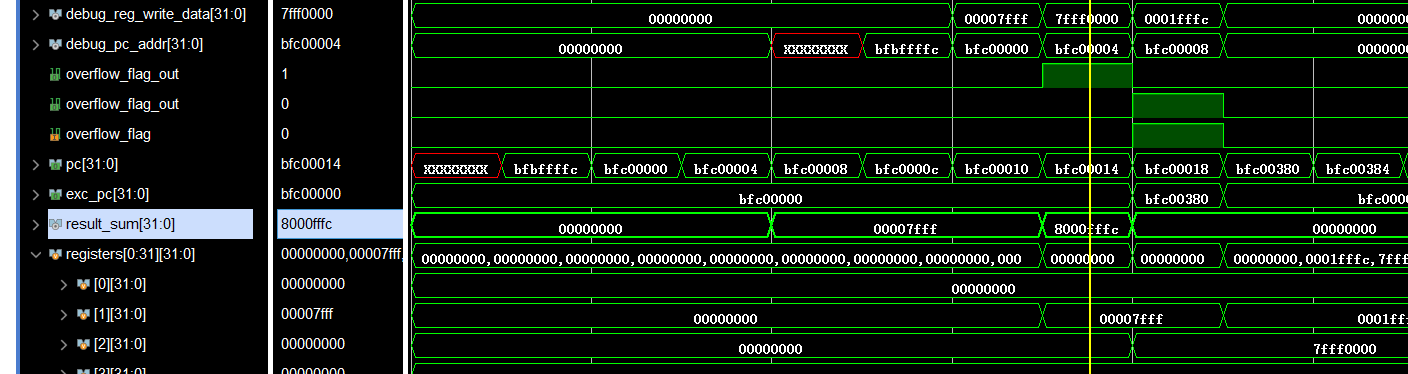


图 8

如上图所示，result\_sum为ex级的加法计算结果，从图中可以看到，结果为8000 fffc，发生溢出，一个时钟周期之后overflow信号量变为1，再过一个周期，overflow传入exmem级，再过一个周期，overflow传入pipline\_controller，exc\_pc值变为bfc00380，此时CPU应该转去执行对应的处理程序，说明整型溢出异常扩展实现

**4 组内成员主要工作及贡献比例**

刘旭阳 负责指令扩展+功能点测试+性能测试+答辩，协助流水灯扩展 36%

何枚瑾 负责扩展演示流水灯，增加cache提升性能，ppt制作 20%

曹威 增加扩展模块vga接口屏幕展示 17%

王兆丹 负责增加扩展模块sram接口和flash接口ppt制作，协助修改代码 17%

冯业莲 增加扩展模块串口通信部分 10%

**四：结论（讨论）**

1. **结论**

本次实验中我学习了 TinyMips 的体系结构以及五级流水线 CPU 的设计与逐层实现，在 TinyMips 的引导下逐渐了解了流水线 CPU 的工作原理和设计方式以及指令执行过程，以及各种指令的运行所需要的基础的元器件，并且对取指、译码、执行、写回、访存五个基本模块有一定的理解，完成了 TinyMIPS 处理器的仿真测试和 FPGA 测试。

此外，我对Verilog 语言也有了更好的掌握。在后续的实验中，我扩展了新的指令，包括运算级指令，跳转级指令和访存级指令。这让我对于工程的理解更加深入，更了解 CPU 的工作原理。我也学会了将汇编指令转换为 2 进制来验证指令的正确性。此外这次实验还用到了实验箱，我了解了如何通过 vivado 将程序跑在实验箱上，来直观的看到程序运行的结果。同时在 vivado 软件中进行仿真的过程中我还了解到了有关代码、激励的编写与测试具体方式，调试水平进一步上升。在上板实验中，我了解了之所以比特流文件可以导入实验箱中，并且可以实现判断验证的功能是通过confreg.v文件实现的，通过拨动实验箱上的拨码，可以控制实验箱验证的速度，实现了对实验的快慢控制。但是我在实验过程中，有一些地方不是很明白，在老师和助教的帮助下解决了一部分的难题，对我完成这次课程设计有很大的帮助。对于 CPU 和流水线的理解，我认为自己虽然会使用，但还是不够扎实，需要在类似实验的环境里面多动手多去尝试才能更好地理解计算机组成原理的相关内容。

**2、讨论**

对于个别繁琐指令的编写需要不停地调试，无法做到一次性就将所有需要更改的地方更改完成，需要自己慢慢地调试。并且，对于延迟槽的使用不太熟练，经常忘记在某些跳转指令或者其他指令之后加入nop指令，使得延迟槽中存放有其他指令，造成一些未预料到的后果。我认为提高本次的课程，相当与对数字逻辑和计算机组成原理等课程的复习。通过本课 程也相当与在写课程上的进一步提高，第一次编写流水线类型的 CPU。我个人感觉学完这些 对于 CPU 等硬件进行更加深入的了解。

**附:**

**TinyMIPS实现的MIPS指令:**

表1-1 算术运算指令

|  |  |
| --- | --- |
| 指令助记格式 | 指令功能简述 |
| ADDU rd,rs,rt | 无符号加（无溢出异常） |
| ADDIU rt,rs,imm | 无符号加立即数（无溢出异常） |
| SUBU rd,rs,rt | 无符号减（无溢出异常） |
| SLT rd,rs,rt | 有符号小于置1 |
| SLTU rd,rs,rt | 无符号小于置1 |

表1-2 逻辑运算指令

|  |  |
| --- | --- |
| 指令助记格式 | 指令功能简述 |
| AND rd,rs,rt | 按位与 |
| LUI rt,imm | 寄存器高位置立即数 |
| OR rd,rs,rt | 按位或 |
| XOR rd,rs,rt | 按位异或 |

表1-3 移位指令

|  |  |
| --- | --- |
| 指令助记格式 | 指令功能简述 |
| SLL rd,rt,sa | 立即数逻辑左移 |
| SLLV rd,rs,rt | 寄存器逻辑左移 |
| SRAV rd,rs,rt | 寄存器算术右移 |
| SRLV rd,rt,sa | 寄存器逻辑右移 |

表1-4 分支跳转指令

|  |  |
| --- | --- |
| 指令助记格式 | 指令功能简述 |
| BEQ rs,rt,offest | 相等时分支转移 |
| BNE rs,rt,offest | 不等时分支转移 |
| JAL target | 无条件直接跳转，并保存返回地址 |
| JALR rd,rs | 无条件寄存器跳转，并保存返回地址 |

表1-5 访存指令

|  |  |
| --- | --- |
| 指令助记格式 | 指令功能简述 |
| LB rt,offest(base) | 访存读字节（8位），有符号扩展 |
| LBU rt,offest(base) | 访存读字节（8位），无符号扩展 |
| LW rt,offest(base) | 访存读字（32位） |
| SB rt,offest(base) | 访存写字节（8位） |
| SW rt,offest(base) | 访存写字（32位） |

**TinyMIPS\_Extend新增的指令：**

OpCode | Description |

| ------- | ----------- |

| JR | N/A |

| ORI | N/A |

| ANDI | N/A |

| MFHI | N/A |

| MFLO | N/A |

| MTHI | N/A |

| MTLO | N/A |

| DIV | N/A |

| DIVU | N/A |

| MULT | N/A |

| MULTU | N/A |

| MFC0 | N/A |

| MTC0 | N/A |

| ERET | N/A |

| BREAK | N/A |

| SYSCALL | N/A |

**可选的MIPS指令见附件A02:**

**自行扩展的指令包括：**

表2-1 算术运算指令

|  |  |
| --- | --- |
| 指令助记格式 | 指令功能简述 |
| SLTIU | 将寄存器 rs 的值与有符号扩展 至 32 位的立即数 imm 进行无符号数比较，如果寄存器 rs 中的值小，则寄存器 rt 置 1；否则寄存器 rt 置 0。 |
|  |  |

表2-2 逻辑运算指令

|  |  |
| --- | --- |
| 指令助记格式 | 指令功能简述 |
|  |  |
|  |  |

表2-3 移位指令

|  |  |
| --- | --- |
| 指令助记格式 | 指令功能简述 |
|  |  |
|  |  |

表2-4 分支跳转指令

|  |  |
| --- | --- |
| 指令助记格式 | 指令功能简述 |
| BGEZ | 如果寄存器 rs 的值大于等于 0 则转移，否则顺序执行。转移目标由立即数 offset 左移 2 位并进行有符号扩展的值加上该分支指令对应的延迟槽指令的 PC 计算得到。 |
|  |  |

表2-5 访存指令

|  |  |
| --- | --- |
| 指令助记格式 | 指令功能简述 |
| SH | 将 base 寄存器的值加上符号扩展后的立即数 offset 得到访存的虚地址，如果地址不是 2 的整数倍则触发地址错例外，否则据此虚地址将 rt 寄存器的低半字存入存储器中。 |
|  |  |

表2-6其它指令（包括数据移动指令、自陷指令、特权指令）

|  |  |
| --- | --- |
| 指令助记格式 | 指令功能简述 |
|  |  |
|  |  |

**北京科技大学实验报告**

学院：计算机与通信工程学院 专业：物联网工程 班级：物联191

姓名：刘旭阳 学号：41927039 实验日期：2021年12月10日

**五、教师评审**

|  |  |
| --- | --- |
| **教师评语** | **实验成绩** |
| （虽然课设主要侧重于验证问题，但是建议各位老师从解决“工程技术问题”，特别是“复杂工程问题”的角度去评审学生课设过程及代码阅读报告，主要内容包括提出问题、分析问题、解决问题及验证问题。**要有较详细的评审意见**。）  签名：  日期： |  |