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： 王丹琳

专　　业： 计算机科学与技术

班　　级： 计 184

学　　号： 41824179

指导教师： 张磊

报告成绩：

实验地点： 机电楼 304

实验时间： 2020 年 11 月 22 日----2021 年 3 月 4 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：　计算机科学与技术　　班级：　计 184

姓名：　王丹琳　　学号：　41824179　　实验日期：　2020 年 12 月 28 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。
- 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

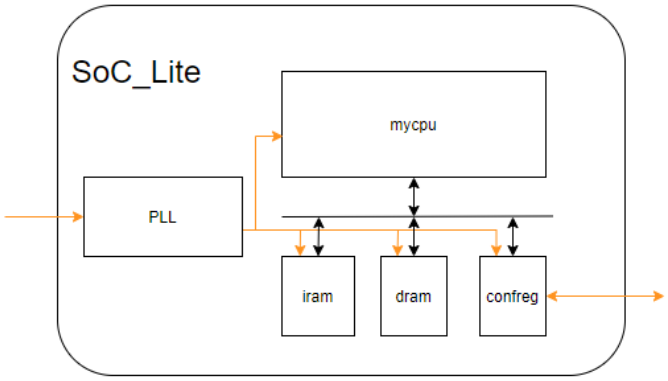
VirtualBox 虚拟机+Ubuntu16.04.6

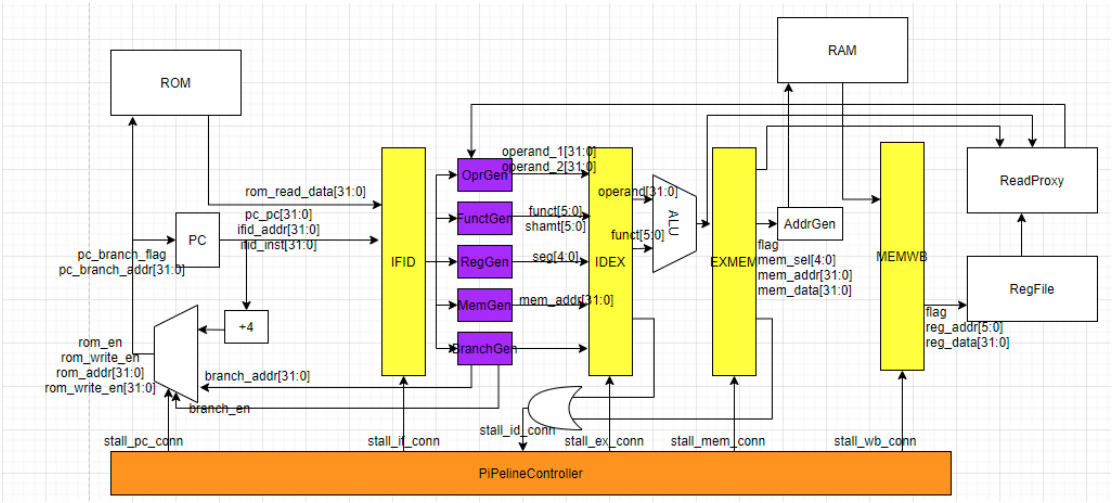
三、设计过程与结果分析

Part 1 验证实验，代码阅读分析

1 TinyMIPS 总体结构框图（自己动手重新画图，推荐画图工具：gliffy 或 draw.io 或者其它）

```
├─bus
├─core
|  ├─pipeline
|  ├─stage
|  |  ├─ex
|  |  ├─id
|  |  ├─if
|  |  ├─mem
|  |  └─wb
|  └─storage
├─include
├─ip
|  └─InternalCrossbar
|      ├─doc
|      ├─hdl
|      ├─sim
|      ├─simulation
|      └─synth
├─mmu
└─sim
```





2 ADDU 指令（典型指令）的设计过程

2.1 指令格式

3.3.3 ADDU

31 26	25 21	20 16	15 11	10 6	5 0
000000	rs	rt	rd	00000	100001

格式：ADDU rd, rs, rt

目标：将寄存器 rs 的值与寄存器 rt 的值相加，结果写入 rd 寄存器中。

描述：GPR[rd] ← GPR[rs] + GPR[rt]

2.2 实现代码（需有恰当的注释）

ADDU 为 R 型指令，Opcode 为 special，funct 为 100001

```
//funct.v
`define FUNCT_ADDU       6'b100001
```

由于需要两个操作数 operand_1，operand_2 分别来自 rs,rt，故 OperateGen.v 中进行生成

```
// generate operand_1
  always @(*) begin
    case (op)
      ……
      `OP_SPECIAL: begin
          operand_1 <= funct == `FUNCT_JALR ? link_addr : reg_data_1;
      end
// generate operand_2
  always @(*) begin
    case (op)
      ……
      `OP_SPECIAL: begin
        operand_2 <= reg_data_2;
      end
```

ADDU 需要写寄存器，故 RegGen.v 进行修改

```
// generate read address
  always @(*) begin
    case (op)
      ……
      // r-type
      `OP_SPECIAL: begin
        reg_read_en_1 <= 1;
        reg_read_en_2 <= 1;
        reg_addr_1 <= rs;
        reg_addr_2 <= rt;
      end
// generate write address
  always @(*) begin
    case (op)
      ……
      `OP_SPECIAL: begin
        reg_write_en <= 1;
        reg_write_addr <= rd;
      end
```

FunctGen.v 中生成 ALU 运算码

```
//FunctGen.v
// generating FUNCT signal in order for the ALU to perform operations
  always @(*) begin
    case (op)
      `OP_SPECIAL: funct <= funct_in;
       ……
```

EX.v 执行加法

```
//EX.v
// calculate result
  always @(*) begin
    case (funct)
      ......
      // arithmetic
      `FUNCT_ADDU, `FUNCT_SUBU: result <= result_sum;
```

WB.v 完成写回

```
else begin
      result_out <= result_in;
    end
```

2.3 仿真测试（需列出指令的测试波形以及程序段的测试结果并分别说明）

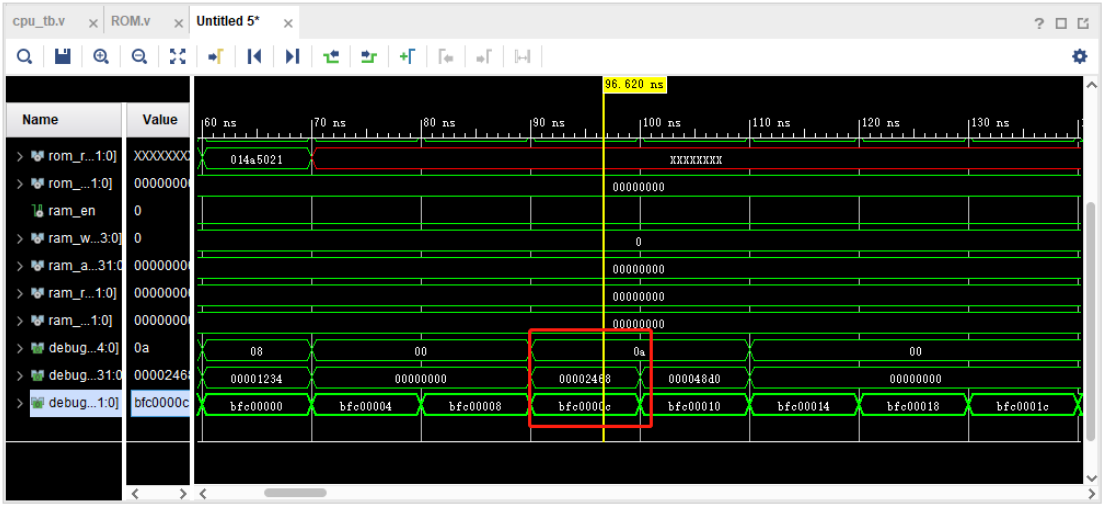
addiu $8, $0, 0x1234（不跳转）

blez $8, label

nop

addu $10,$8,$8

如下图，8 号寄存器值加 8 号寄存器值存入 10 号寄存器，最终 10 号寄存器值为 0x2468



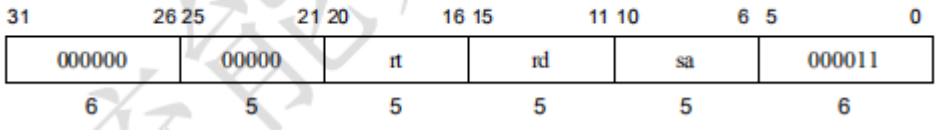
2.4 FPGA 测试

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

总述:共实现了 inst 文件夹中 n1-n64 的指令扩展（除 CDE 中支持的 22 条指令），分别是：SRA,BLEZ,J,LH 等，其中指令 SRA 的设计实现过程如下：（三类指令分别扩展一条以上）

Part 2.1 移位运算指令（SRA）

3.5.4 SRA



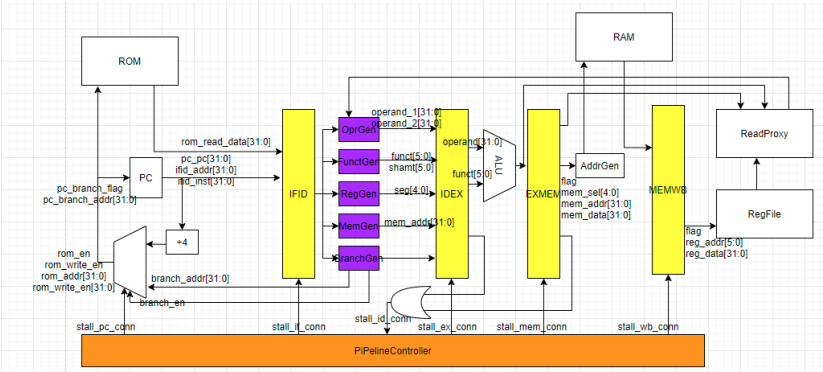
格式：SRA rd, rt, sa

目标：由立即数 sa 指定移位量，对寄存器 rt 的值进行算术右移，结果写入寄存器 rd 中。

描述：s ← sa

$GPR[rd] \leftarrow (GPR[rt]_{31})^{s} \,||\, GPR[rt]_{31..s}$

2 分析指令功能及执行过程，画出数据通路图



SRA 功能：由立即数 sa 指定移位量，对寄存器 rt 的值进行算术右移，结果写入寄存器 rd 中。

数据通路： IF 级取出指令；

ID 级进行译码，ID 中含 BranchGen，FunctGen，MemGen，OperandGen，RegGen：BranchGen 用于控制指令跳转，FunctGen 用于生成 ALU 的计算码，MemGen 用于读写内存的控制信号，OperandGen 用于根据指令格式生成 operand_1，operand_2，RegGen 用于生成寄存器的读写信号。

SRA 指令执行的过程中，在 ID 级被译码，操作数只有一个，来自 rs 寄存器，需要写入 rd 寄存器，在 RegGen 模块生成寄存器堆的控制信号。在 OperandGen 生成操作数。同时在 FunctGen 模块生成控制信号。由于 SRA 指令为 special 所以不需要添加。由于 SRA 指令没有用到 RAM，也不涉及跳转，所以不涉及 MenGen 和 BranchGen。之后，修改 EX 级：给出功能码实现移位，result 信号会在流水线上传递下去，一直到 WB 级写回 rd 寄存器。

3 代码实现

基本思路就是沿着数据通路添加指令。

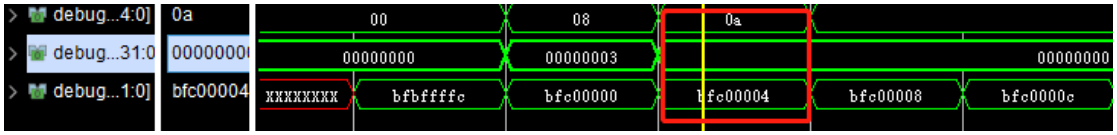
```
//opcode.v
`define OP_SRA          6'b000000
//funct.v
`define FUNCT_SRA       6'b000011
//仿照仿照 SRAV，在 EX.v 中进行添加
`FUNCT_SRA: result <= ({32{operand_2[31]}} << (6'd32 - {1'b0, shamt})) |
operand_2 >> shamt;
```

4 指令功能测试

针对 SRA 指令的测试汇编：

```
24 08 00 03       // addiu $8,$0,0x3                  24 08 ff ff
00 08 50 83     // sra $10,$8,2
```

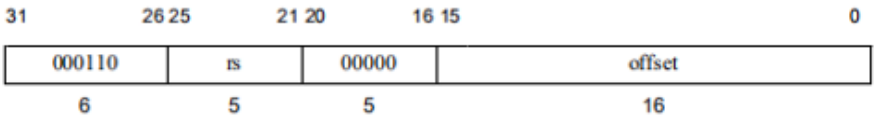
当 SRA 指令正常运行时，结果应该为全零，仿真波形图如下：



与预期相符，拓展成功

Part 2.2 分支型指令（BLEZ）

3.6.5 BLEZ

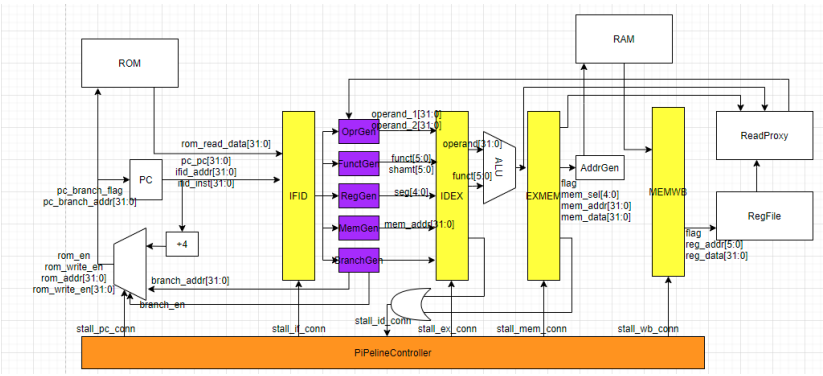


格式：BLEZ rs, offset

目标：如果寄存器 rs 的值小于等于 0 则转移，否则顺序执行。转移目标由立即 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。

描述：I: condition ← GPR[rs] ≤ 0
target_offset ← Sign_extend(offset||0^2)
I+1: if condition then
PC ← PC+ target_offset
endif

2 分析指令功能及执行过程，画出数据通路图



BLEZ 功能：如果寄存器 rs 的值小于等于 0 则转移，否则顺序执行。转移目标由立即 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。

数据通路： IF 级取出指令；

ID 级进行译码，ID 中含 BranchGen，FunctGen，MemGen，OperandGen，RegGen：BranchGen 用于控制指令跳转，FunctGen 用于生成 ALU 的计算码，MemGen 用于读写内存的控制信号，OperandGen 用于根据指令格式生成 operand_1，operand_2，RegGen 用于生成寄存器的读写信号。

BLEZ 指令执行的过程中:在 ID 级被译码：需要读一个寄存器 rs，IF 级的 PC 正常工作，在 RegGen 模块生成寄存器堆的控制信号。OperandGen，FunctGen 模块不涉及。不涉及内存读写，MemGen 也不考虑。由于还需要生成指令跳转的控制信号，所以只需要修改 RegGen，BranchGen。EX 及之后的级在分支指令中不涉及。

3 代码实现

基本思路就是沿着数据通路添加指令。

```
//opcode.v
`define OP_BLEZ         6'b000110
```

由指令格式可知，BLEZ 指令只需要读一个寄存器 rs，据此修改/stage/id/RegGen.v

```
// RegGen.v
  always @(*) begin
    case (op)
      `OP_BGEZ, `OP_BGTZ, `OP_BLEZ,    // 略去 OP_BLTZ
      `OP_LB, `OP_LW, `OP_LBU, `OP_LH, `OP_LHU: begin
        reg_read_en_1 <= 1;
        reg_read_en_2 <= 0;
        reg_addr_1 <= rs;
        reg_addr_2 <= 0;
      end
    ……
```

```
    endcase
  end
```

BLEZ 指令在 ID 级就完成对 PC 的修改，直接在 ID 级的`BranchGen.v`中添加下面的代码

```
//BranchGen.v
always @(*) begin
    case (op)
    ……
    `OP_BLEZ: begin
            if(reg_data_1[31]==1'b1 || reg_data_1 == 0) begin
              branch_flag <= 1;
              branch_addr <= addr_plus_4 + sign_ext_imm_sll2;
            end
            else begin
                branch_flag <= 0;
                branch_addr <= 0;
            end
          end
    ……
```

4 指令功能测试

针对 BLEZ 指令的测试汇编：

24 08 12 34 // addiu $8, $0, 0x1234（不跳转）　　24 08 00 00(跳转)

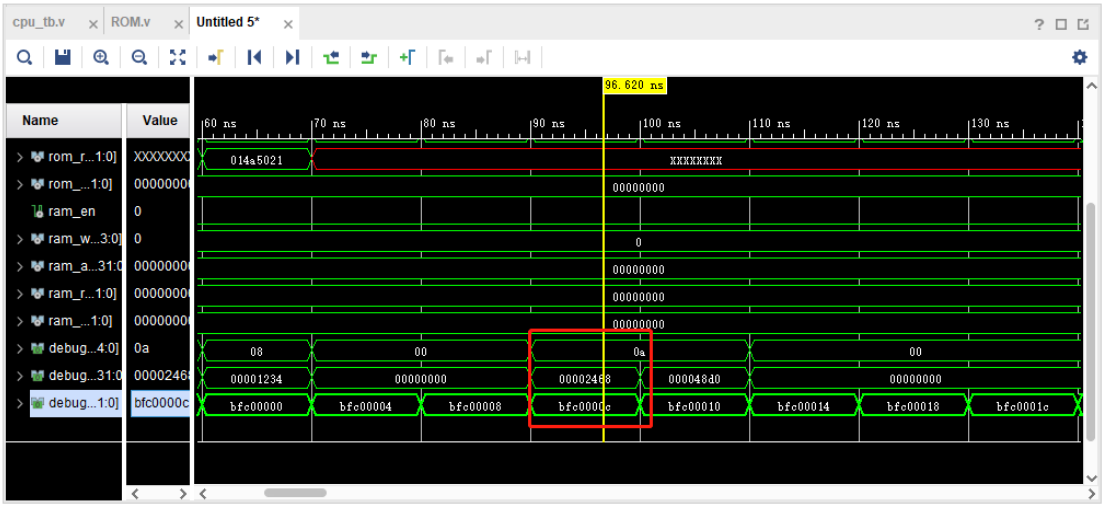
19 00 00 02 // blez $8, label

00 00 00 00 // nop

01 08 50 21 // addu $10,$8,$8

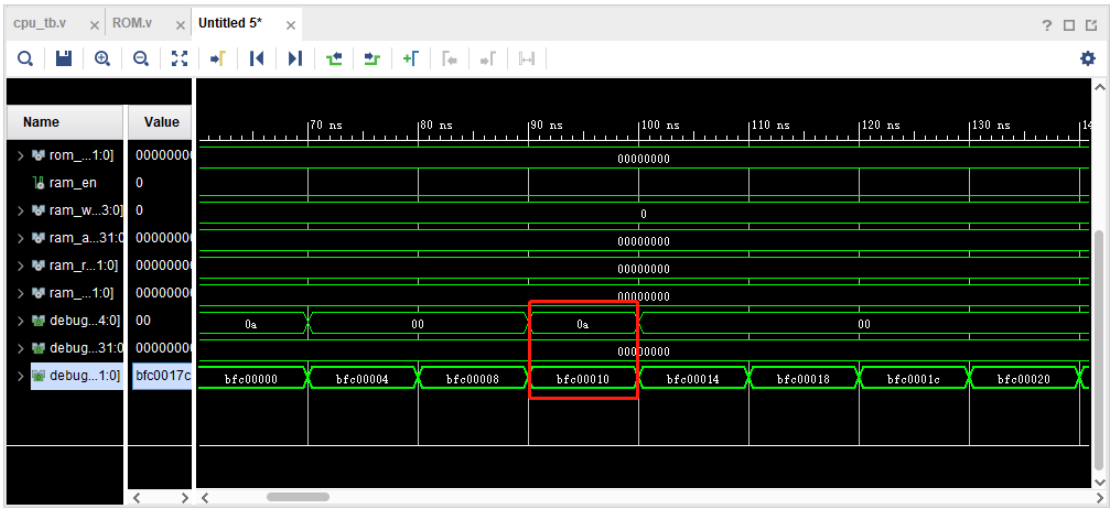
01 4a 50 21 // label:　addu $10, $10, $10

当 BLEZ 指令正常运行时，PC 应当在执行完 blez $8, label 继续执行，不发生跳转，仿真波形图如下：



从红框中可以看到，PC 在 BLEZ 指令结束后，执行了一条 nop 指令，然后执行了 01 08 50 21 这条指令，正是 addu $10,$8,$8；

当指令第一条变为 24 08 00 00 时，PC 应当在执行完 blez $8, label 发生跳转至 01 4a 50 21，仿真波形图如下：

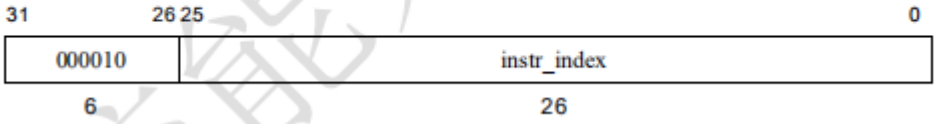


从红框中可以看到，PC 在 BLEZ 指令结束后，执行了一条 nop 指令，然后跳转到了 label: addu $10, $10, $10

综上，说明 BLEZ 指令功能正确。

Part 2.3 跳转型指令（J）

3.6.9 J



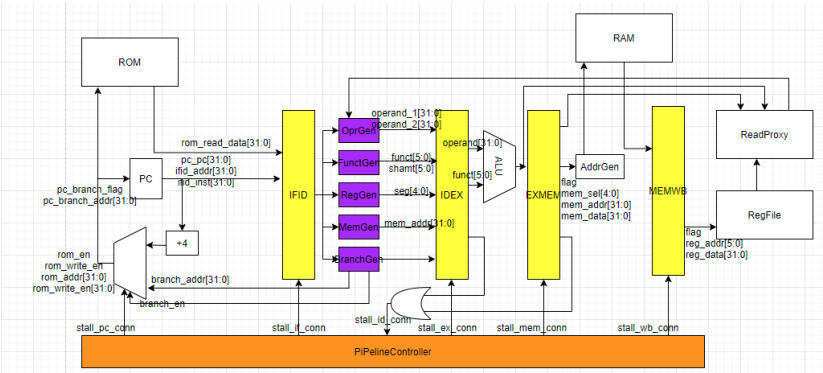
格式：J target

目标：无条件跳转。跳转目标由该分支指令对应的延迟槽指令的 PC 的最高 4 位与立即数 instr_index 左移 2 位后的值拼接得到。

描述：I:

I+1: PC ← $PC_{31..28}$ || instr_index || 0^2

2 分析指令功能及执行过程，画出数据通路图



J 功能：无条件跳转。跳转目标由该分支指令对应的延迟槽指令的 PC 的最高 4 位与立即数 instr_index 左移 2 位后的值拼接得到。

数据通路： IF 级取出指令；

ID 级进行译码，ID 中含 BranchGen，FunctGen，MemGen，OperandGen，RegGen：BranchGen 用于控制指令跳转，FunctGen 用于生成 ALU 的计算码，MemGen 用于读写内存的控制信号，OperandGen 用于根据指令格式生成 operand_1，operand_2，RegGen 用于生成寄存器的读写信号。

J 指令执行的过程中：IF 级的 PC 正常工作，由于是无条件跳转，故 RegGen 模块不涉及。无操作数 OperandGen 也不涉及，FunctGen 模块不涉及。不涉及内存读写，MemGen 也不涉及。所以只需要生成指令跳转的控制信号，修改 RegGen，BranchGen。EX 及之后的级在跳转指令中不涉及。

3 代码实现

基本思路就是沿着数据通路添加指令。

```
//opcode.v
`define OP_J            6'b000010
```

在 ID 级 J 指令需要对 PC 进行修改，故直接在 ID 级的`BranchGen.v`中添加下面的代码：

```
`OP_J: begin
    branch_flag <= 1;
    branch_addr <= {addr_plus_4[31:28], jump_addr, 2'b00};
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。）

针对 J 指令的测试汇编：

```
24 08 00 01 // addiu $8, $0, 1
25 08 00 02 // addiu $8, $8, 2
08 00 00 06 // j 0x6
00 00 00 00 // nop
25 08 00 08 // addiu $8, $8, 8
25 08 00 08 // addiu $8, $8, 8
25 08 00 05 // addiu $8, $8, 5
25 08 00 08 // addiu $8, $8, 8
```

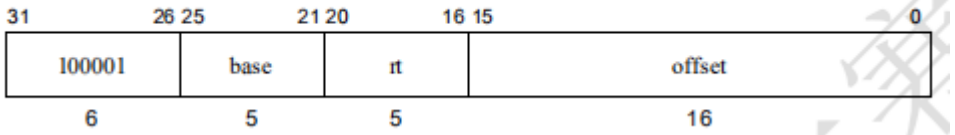
正常运行时，PC 应当在执行完 j 0x6 之后跳转到 addiu $8, $8 ,5 并继续执行，下面是

仿真波形：

从红框中可以看到，PC 在 J 指令结束后，执行了一条 nop 指令，然后跳转到了 2508005 这条指令，正是 addiu $8, $8 ,5，说明 J 指令功能正确。

Part 2.4 访存型指令（LH）

3.9.3 LH

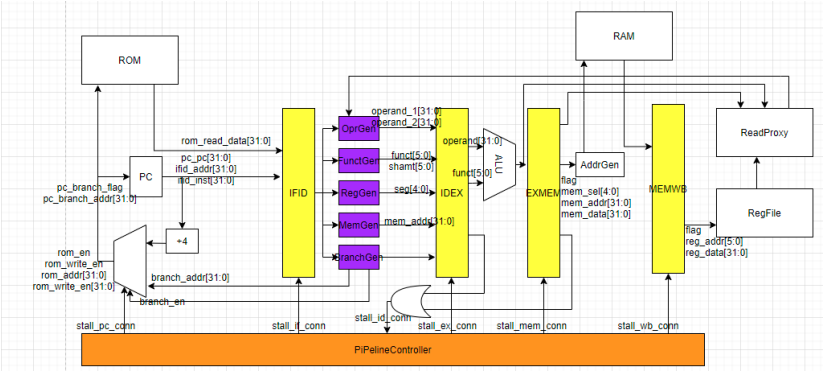


格式：LH rt, offset(base)

目标：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，否则据此虚地址从存储器中读取连续 2 个字节的值并进行符号扩展，写入到 rt 寄存器中。

描述：
```
vAddr ← GPR[base] + sign_extend(offset)
if vAddr0 ≠ 0 then
      SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
memhalf ← LoadMemory(CCA, HALFWORD, pAddr, vAddr, DATA)
GPR[rt] ← sign_extend(memhalf15..0)
```

2 分析指令功能及执行过程，画出数据通路图



LH 功能：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，否则据此虚地址从存储器中读取连续 2 个字节的值并进行符号扩展，写入到 rt 寄存器中。

数据通路： IF 级取出指令；

ID 级进行译码，ID 中含 BranchGen，FunctGen，MemGen，OperandGen，RegGen：BranchGen

用于控制指令跳转，FunctGen 用于生成 ALU 的计算码，MemGen 用于读写内存的控制信号，OperandGen 用于根据指令格式生成 operand_1, operand_2，RegGen 用于生成寄存器的读写信号。

J 指令执行的过程中：取指令，IF 级译码，由于需要读 rs 寄存器，故 RegGen 模块需要生成读信号。操作数两个：OperandGen 生成 operand_1, operand_2。做加法：FunctGen 模块产生 addu 操作码。涉及内存读写，MemGen 写使能置 1，生成对存储器的控制信号。由于是半字（half-word）的操作，故增加 mem_sel_in == 4'b0011。然后流水线继续执行，在 WB 级寄存器写入半个字。

3 代码实现

基本思路就是仿照着已经实现的 LB，LBU 指令,沿着数据通路添加指令。

```
//opcode.v
`define OP_LH          6'b100001
```

由指令格式可知，LH 指令需要读一个寄存器 rs。写一个寄存器 rt，据此修改 /stage/id/RegGen.v

```
// RegGen.v
// generate read address
  always @(*) begin
    case (op)
      `OP_LB, `OP_LW, `OP_LBU, `OP_LH, `OP_LHU: begin
        reg_read_en_1 <= 1;
        reg_read_en_2 <= 0;
        reg_addr_1 <= rs;
        reg_addr_2 <= 0;
      end
     ......

// generate write address
  always @(*) begin
    case (op)
      ......
      `OP_LB, `OP_LBU, `OP_LW, `OP_LH, `OP_LHU: begin
        reg_write_en <= 1;
        reg_write_addr <= rt;
      end
      ......
```

由于两个操作数是 base 寄存器的值和符号扩展的立即数，所以修改/id/OperandGen.v

```
// OperandGen.v
// generate operand_1
  always @(*) begin
    case (op)
      `OP_LB, `OP_LW, `OP_LBU, `OP_SB, `OP_SW, `OP_LH, `OP_LHU, `OP_SH: begin
        operand_1 <= reg_data_1;
      end
      ……
// generate operand_2
  always @(*) begin
    case (op)
      // memory accessing
      `OP_LB, `OP_LW, `OP_LBU, `OP_SB, `OP_SW, `OP_LH, `OP_LHU, `OP_SH: begin
        operand_2 <= sign_ext_imm;
       ……
```

由指令可知，内存地址是由 base 寄存器的值和符号扩展的立即数无符号相加，所以要生成 ALU 的运算码。让 ALU 去做加法运算。功能码是`FUNCT_ADDU，据此修改 FunctGen.v：

```
// FunctGen.v
always @(*) begin
    case (op)
    ……
      `OP_SB, `OP_SW, `OP_SH, `OP_ADDIU: funct <= `FUNCT_ADDU;
```

由于需要都读内存，修改/MemGen.v

```
//MemGen.v
…
`OP_LH: mem_read_flag <= 1;
…
`OP_LH: mem_sign_ext_flag <= 1;
…
`OP_LH: mem_sel <= 4'b0011;
…
```

接下来就是，修改访存级 MEM，由于是从存储器中读取连续 2 个字节的值并进行符号扩展

```
// MEM.v
 ……
 else if(mem_sel_in == 4'b0011) begin
 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
 ……
```

```
else 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;
default: ram_write_data <= 0;
……
```

最后 WB 级，写回寄存器

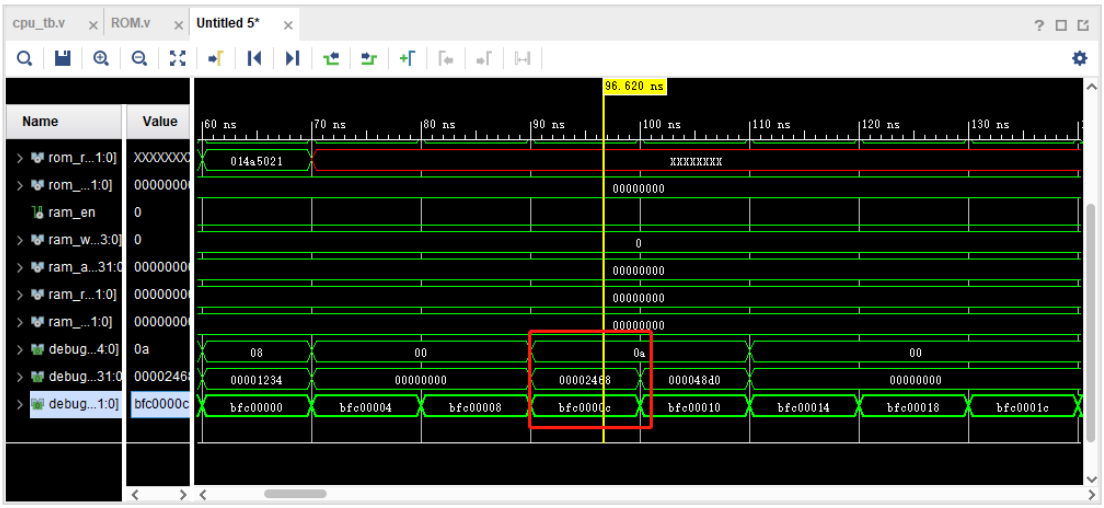
```
else if (mem_sel == 4'b0011) begin
        case(address[1:0])
          2'b00: result_out <= mem_sign_ext_flag ? {{16{ram_read_data[15]}},
ram_read_data[15:0]} : {16'b0, ram_read_data[15:0]};
          2'b10: result_out <= mem_sign_ext_flag ? {{16{ram_read_data[31]}},
ram_read_data[31:16]} : {16'b0, ram_read_data[31:16]};
          default: result_out <= 0;
        endcase
      end
```

4 指令功能测试

针对 LH 指令的测试汇编：

```
24 08 00 08// addiu $8,$0,0x8
24 09 0f 00// addiu $9,$0,0x0F00
ad 09 00 00// sw $9,0($8)
85 0a 00 00// lh $10,0($8)
85 0b 00 02// lh $11,2($8)
```

当 LH 指令正常运行时，RAM 中的数据会到 10 号 11 号寄存器中，仿真波形图如下：



从红框中可以看到，LH 指令确实将 RAM 中的数据分别加载到了$10 和$11 中，并且可以发现是小端存储（低字节存储在低地址）

Part 3 创新实验（演示系统介绍）实验分组编号：29

见同组雷方雨实验报告

四：结论（讨论）

（对照课设内容，简要总结课设期间完成的主要工作：对照课设目的，着重说明通过课程设计过程所取得的收获和能力的达成情况；存在的问题及可能的改进方向；其它需要说明的问题）

1、结论（实验总结）

在此次课程设计中，从观看指导视频到仿真平台的实操，我掌握了数据前递与流水线暂停的一些知识，这些都是计组课堂上所接触不到的知识。同时，我也深刻了解了五级流水线 CPU 的各个级所负责的功能，学习了数据相关的处理方式（例如写后读），了解了一些特殊的数据读写方式。仿真实验平台的可视化也让我对这些概念有了直观的、深刻认识。经过第一次的验收，我也在助教的提问下，找到了自己知识的薄弱点，在助教的解答下，对一些没注意到的地方有了新的认识。

在寒假过程中，通过学习指导书与视频，与小组成员讨论，我和同组成员一起拓展指令并通过测试了 88 个测试点，这也让我收获匪浅。起初最先遇到的问题就是跑 trace 比对时，修改了 start.s 也进行了编译，但是 vivado 上永远都是通过了 26 个测试点，这个问题困扰了我好几天，所写的指令得不到测试，最后在同组成员的讨论下成功解决了问题，对 vivado 这个环境有了新的认识。紧接着就是拓展乘除法，这个只是在课堂中学过理论知识，而实操让我也明白了其中的原理，通过上网查找资料，阅读推荐书籍，根据波形图一点点的去找哪里代码存在问题，最终成功实现让我开心了好几天。异常指令的拓展我是通过学习老师下发的资料进行拓展，其中也遇到了一些小波折，但是也都解决了。

计算机组成原理计组课设的开设，让我学会了许多以前所不会接触的知识，对于理解计算机的 CPU 有很大的帮助，我也觉得这门课应该值得更多的学时，值得我们花更多的时间来学习。

2、讨论（问题归纳，课程建议等）

经过本门课程的学习，收获颇多，个人希望这门课能有更多的学时来让我们学习，同时也希望能早一点开设，在大二下学完《计算机组成原理》后的暑假小学期安排计组课程设计，这样能够让我们在理论学习完之后及时应用知识，加深对计算机组成原理的了解，同时也可以减轻大三上繁重的学习实验负担。

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）

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
SLTI	将寄存器 rs 的值与有符号扩展至 32 位的立即数 imm 进行有符号数比较，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
XOR	寄存器 rs 中的值与寄存器 rt 中的值按位逻辑异或，结果写入寄存器 rd 中。
……	……

表 2-3 移位指令

指令助记格式	指令功能简述
SRA	由立即数 sa 指定移位量，对寄存器 rt 的值进行算术右移，结果写入寄存器 rd 中。
……	……

表 2-4 分支跳转指令

指令助记格式	指令功能简述

BLTZ	如果寄存器 rs 的值小于 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
LH	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，否则据此虚地址从存储器中读取连续 2 个字节的值并进行符号扩展，写入到 rt 寄存器中。
……	……

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
MFHI	将 HI 寄存器的值写入到寄存器 rd 中
……	……

北京科技大学实验报告

学院： 计算机与通信工程学院　　专业： 计算机科学与技术　　班级： 计 184

姓名： 王丹琳　　学号： 41824179　　实验日期： 2021 年 2 月 1 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：	