计算机组成原理实验指导书

主编 张晓彤 刘宏岚 张磊等

计算机实验室
二零一八年四月

北京科技大学计算机与通信工程学院

目录

前言

本书从计算机科学与技术专业教学指导委员会发布的计算机组成原理课程设计的大纲出发，重点介绍了单周期 MIPS 处理器的设计思路与实现步骤；并用 Verilog 语言实现了该处理器的 8 条指令，希望起到抛砖引玉的作用。学生可在此基础上发挥自己的潜能，自行设计完成计算机组成原理课内实验与课程设计。

本手册可供学过数字逻辑课程的相关专业的同学在计算机组成原理课程实验中参考使用。对于没有开设数字逻辑课程的专业可选择使用。

由于前两章在数字逻辑课程中已经介绍，在此不再做介绍，直接进入单周期 MIPS 处理器设计章节。

在编写过程中，得到了张晓彤老师和刘宏岚老师的大力支持，并提供了很多宝贵的意见；宛嵇祥、张培、赵雪松等同学为本指导书的编写也做了大量工作，在此一并表示感谢。

作者希望本书内容能引起对 CPU 和复杂数字逻辑系统设计有兴趣的电子工程师们的注意，加入我国集成电路的设计队伍，提高我国电子产品的档次。

由于作者的经验与学识有限，不足之处敬请读者批评、指正。

编者

2018 年 4 月

第一章 Vivado 开发平台及应用介绍（略）

第二章 硬件描述语言 Verilog 介绍（略）

第三章 单周期 CPU 设计与实现

3.1 指令系统

MIPS 指令集优点：

① 指令集采用 32-bit 编码，因为指令长度相同，能够降低译码的复杂度，减少译码延迟，并能很容易的使用流水线来提高处理器的效率。

② 设计的嵌入式处理器只有 load 和 store 指令对存储系统进行访问。简单的指令寻址方式简化了嵌入式微处理器控制器和数据通路的设计。

③ 处理器采用通用寄存器组结构，在这种结构中，指令可以从寄存器堆读出操作数，并将执行的结果回写到寄存器堆中。

MIPS 指令集分类：

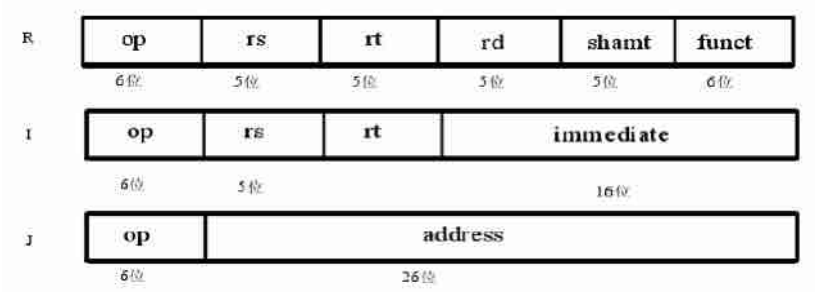


图 1 Mips 三种不同的指令类型

字段名	字段意义说明
op	指令操作码
rs	源操作数 1
rt	源操作数 2 或目的操作数
rd	目的操作数
shamt	位移量，在移位运算中存储位移量
funct	功能码，当指令为 R 类型时，根据功能码判断指令的操作类型
immediate	当其为 I 型指令时为立即数，分支指令时，为跳转偏移量
address	跳转绝对地址，为指令 J 的指令格式

图 2　指令字段说明

① 算数逻辑指令，可能是 R 型或 I 型指令，例如 addu(无符号加法)，addiu（无符号立即数加法)。
② 存储访问指令，用于存取存储器，指令格式与 I 型指令相同，意义不同，例如 lw(加载一个 word)，sw(存储一个 word)。
③ 分支和跳转指令，能改变程序的执行过程，可能是 J 型或 I 型指令，例如 j(跳转)，beq（相等跳转)。
④ 协处理器及其 CP0 指令，和专用指令，未予实现。

根据指令类型，至少需要实现三种寻址方式：
① 寄存器寻址：源操作数或目的操作数为寄存器的指令，主要有 R 型指令，I 型指令，通过指明寄存器号来读取操作数，例如 addu，addiu。
② 基址加偏移寻址：寄存器与偏移量相加得到地址，例如 lw，sw。
③ PC 相对寻址：PC 与地址偏移相加得到地址，例如 beq，j。

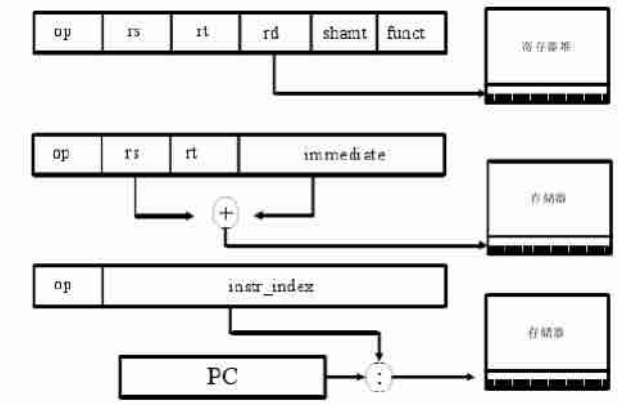


图 3　指令寻址方式

寄存器结构采用标准的 32 位寄存器堆，共 32 个寄存器，标号为 0-31。其中第 0 寄存器永远为全 0，第 31 寄存器是跳转链接地址寄存器。它在链接型跳转指令下会自动存入返回地址值。对于其它寄存器，可由软件自由控制。

寄存器编号	助记符	用途
$0	zero	常数 0
$1	at	汇编暂存寄存器
$2 $3	v0, v1	表达式结果或子程序返回值
$4-$7	a0-a3	过程调用的前几个参数
$8-$15	t0-t7	临时变量，过程调用时不需要恢复
$16-$23	s0-s7	临时变量，过程调用时需要恢复
$24 $25	t8 t9	临时变量，过程调用时不需要恢复
$26 $27	k0 k1	留给操作系统，通常被中断或例外用来保存参数
$28	gp	全局指针
$29	sp	堆栈指针
$30	s8/fp	第 9 寄存器变量，过程调用时作为帧指针
$31	ra	过程返回地址

图 4　MIPS 的规范中寄存器含义

以 MIPS 指令系统的简化版本为例，在简化版本中，实现如下指令：

Syntax	Instruction						Operation
addu rd,rs,rt	0	rs	rt	rd	0	0x21	(rd) = (rs) + (rt)
addiu rt,rs,imm	9	rs	rt	imm			(rt) = (rs) + imm
lw rt,offset(rs)	0x23	rs	rt	offset			(rt) = MEM[(rs) + offset]
sw rt,offset(rs)	0x2b	rs	rt	offset			MEM[(rs) + offset] = (rt)
beq rs,rt,offset	4	rs	rt	offset			if (rs) == (rt) : pc += offset<<2
j target	2	target					pc = (pc & 0xf0000000) | (target << 2)

图 5　指令的语法、格式、功能

3.2 设计

3.2.1 顶层设计

分析指令构建所需的数据通路：

① 指令从 IFU(取指令模块)中获得,指令存储器需要 32 位的地址来查找指令，它存储在一个 32 位的 pc 寄存器中，pc 寄存器每个周期加 4。

② addu 指令：addu rd,rs,rt，将 rs 寄存器中的值和 rt 寄存器中的值相加存入 rd 寄存器中。加法操作要将从 RegFile(寄存器堆模块)取出的两个操作数送到 ALU（算数逻辑单元）模块运算。

③ addiu 指令：addiu rt,rs,imm，将 rs 寄存器中的值和 imm(立即数) 符号扩展后的值相加存入 rt 寄存器中。对比 addu，addiu，它们都执行加法，有一个操作数都是来自 rs 寄存器，但另一个操作数 addu 是来自 rt 寄存器，addiu 则是来自指令中的立即数。所以在数据通路中需要一个数选器区分。

④ lw 指令：lw rt,offset(rs)，将 rs 寄存器中的值加上 offset 符号扩展后的值作为地址在 DataMem(数据存储器模块)中找到数据存入 rt 寄存器。

⑤ sw 指令：sw rt,offset(rs)，将 rt 寄存器中的值存入 DataMem，地址为 rs 寄存器中的值加上 offset 符号扩展后的值。

⑥ beq 指令：beq rs,rt,offset，比较 rs 和 rt 寄存器中的值，如果相等 pc 就跳转到地址为(符号扩展)(offset<<2)+(pc)处，如果不相等 pc 就不发生跳转。

⑦ j 指令：j target，pc 跳转到 target 左移两位后替换 pc 低 28 位的地址处。

综上所述设计数据通路，分为包含取指令的 IFU 模块，包含寄存器堆及其读写的 RegFile 模块，包含算术和逻辑运算的 ALU 模块，包含数据存取的 DataMem 模块，包含读写控制、数选器控制、操作方式控制的 Control 模块，**要注意 IFU 模块中的指令存储器和 DataMem 模块中的数据存储器实际对应了 CPU 中的指令和数据高速缓存，而不是整个计算机中的内存。**以下的模块关系分析省去了时钟和复位信号。

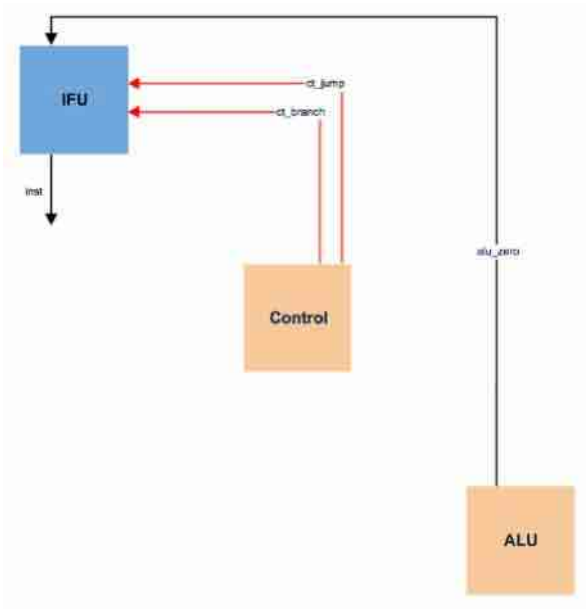


图 6　IFU 模块

有了模块的划分以后开始分析模块的关系，IFU 的功能就是取出指令，所以它的输出就是指令。如果只是每个时钟取出一条指令，然后 pc+4，那么除了时钟和复位信号外就不需要其它输入，但我们要实现的 6 条指令中有分支 beq 和跳转 j 指令，这两条指令需要修改 pc 的值，所以 Control 模块需要发出两个控制信号给 IFU 当前是分支或者跳转指令，分支指令是有条件的，需要判断(rs)减(rt)是否为 0，需要 ALU 模块给出运算结果是否为零的信号。

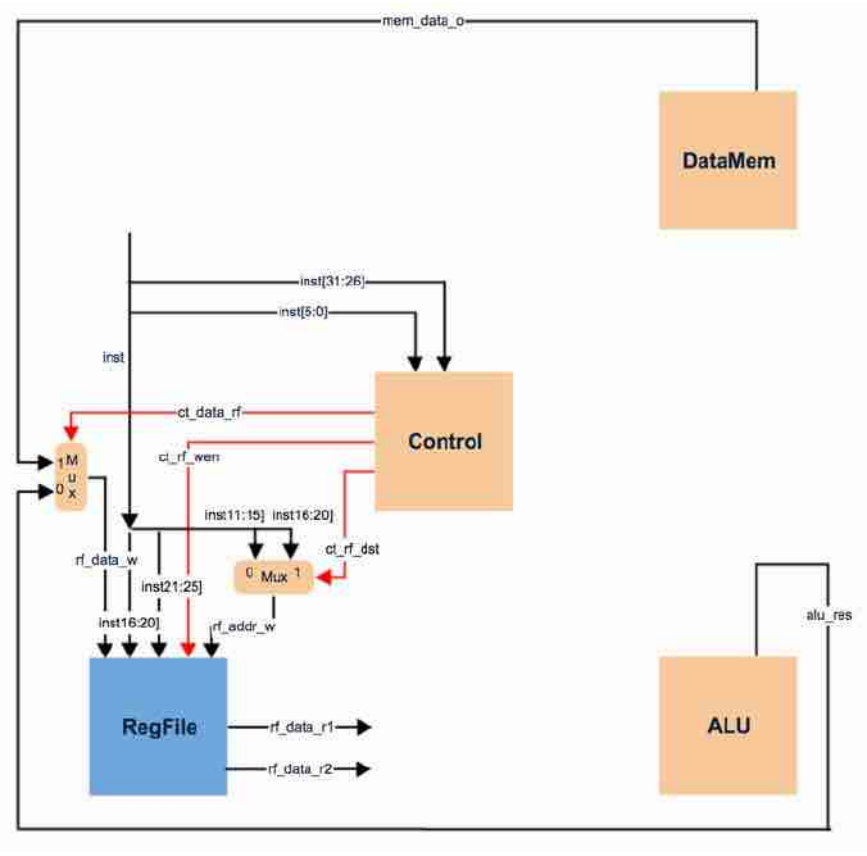


图 7　RegFile 模块

RegFile 需要根据地址读出或写入数据。读操作一次读出两个寄存器的值，它们的地址分别为指令的 rs，rt 字段，对于不同的指令不做区分，对于那些不需要的数据搁置不理。写操作的地址不同的指令不相同，例如 addu 指令写寄存器堆的地址为指令的 rd 字段，而 addiu 指令写寄存器堆的地址为指令的 rt 字段，所以需要一个数选器进行区分，Control 模块给出数选器的控制信号 ct_rf_dst。要保证数据在合适的时候写入寄存器，写操作需要 Control 模块给出一个写使能信号 ct_rf_wen。写入的数据对于不同的指令也不同，可能来自 ALU 的运算结果 alu_res，例如 addu 指令要写回寄存器堆的数据就是 ALU 的运算结果，也可能来自 DateMem 模块的输出 mem_data_o，例如 lw 指令就是从 DataMem 将数据加载到寄存器堆，所以也需要一个数选器区分写入数据的来源，由 Control 模块给出数选器的控制信号 ct_data_rf。

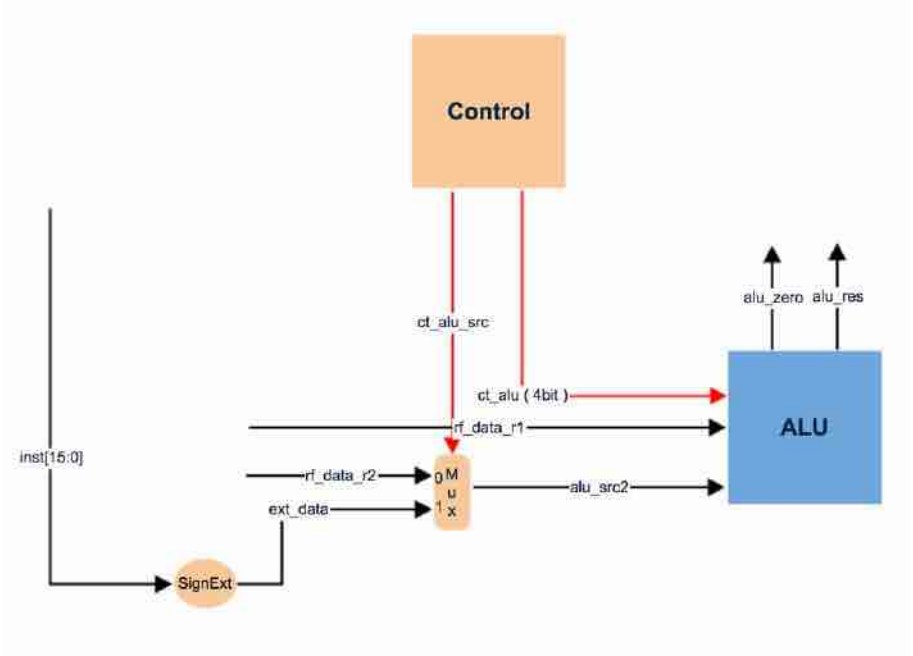


图 8　ALU 模块（略去了之前介绍过的模块）

ALU 模块负责进行数据运算，要区分进行何种运算就需要输入一个选择信号，它来自 Control，这里我们选 4bit 的信号，4bit 意味着可以编码 16 种运算。还有两个数据源，其中一个直接来自寄存器堆读出的数据 rf_data_r1，另一个可能来自寄存器的堆的输出，例如 addu 指令两个数据源都是从寄存器堆中读出的，还有可能来自指令的低 16 位符号扩展为 32 位的结果，例如 addiu 指令一个数据源来自寄存器堆，另一个数据源则是指令中的立即数符号扩展的结果，所以要通过一个数选器区分数据的来源，由 Control 模块给出数选器的控制信号 ct_alu_src。ALU 的输出除了运算结果还应该有一些标志信号用于判断特殊运算结果，例如分支指令 beq 的分支条件(rs)-(rt)是否为 0，这个为 0 的标志信号 alu_zero 就是由 ALU 模块送到 IFU 的。

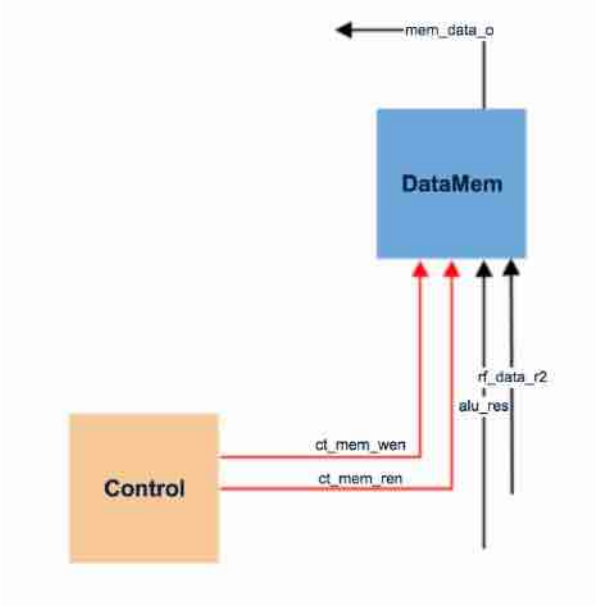


图 9　DataMem 模块

DataMem 模块需要存储运算后的数据，或者直接存储来自寄存器堆读出的数据，所以它需要具备读写功能。

lw 指令需要使用 DataMem 的读操作从 DataMem 模块将数据读取到指定的寄存器中，读操作的地址由 ALU 模块运算得到 alu_res，它是由 lw 中的 rs 字段寄存器的值加上 offset 字段符号扩展后的值，DataMem 同时需要 Control 给出读使能信号 ct_mem_ren，读出的数据由 mem_data_o 输出。

sw 指令需要使用 DataMem 的写操作将制定寄存器中的数据存到 DataMem 模块中，写入的地址同读操作获得地址方法一样，写入的数据是 rt 寄存器中的值，写操作由写使能信号 ct_mem_wen 控制。

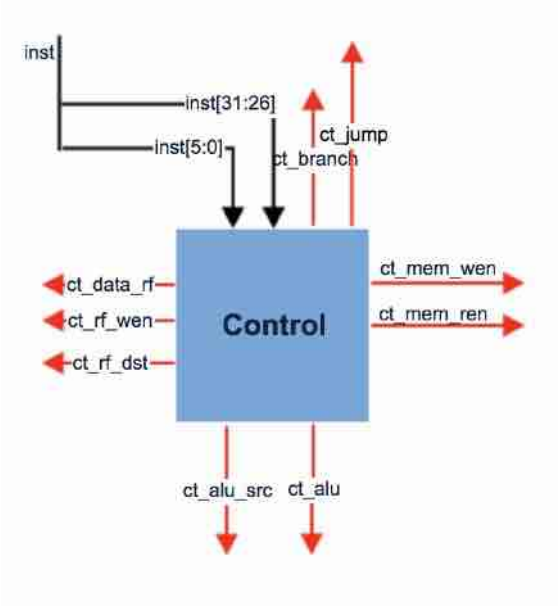


图 10　Control 模块

Control 模块的输入是指令的高六位和低六位，输出是本节之前提到的所有控制信号。最高六位用于区分不同的指令，最低六位用于区分 R 型指令的不同操作，因为需要运算的不光有 R 型指令，所以将指令低六位和由 Control 模块根据

不同指令生成的 2bit 的 ct_alu_op 控制信号一起送入子模块 ALUCt(ALU 控制，图上未画出)，ALUCt 返回 4bit 的 ct_alu 信号由 Control 模块输出。

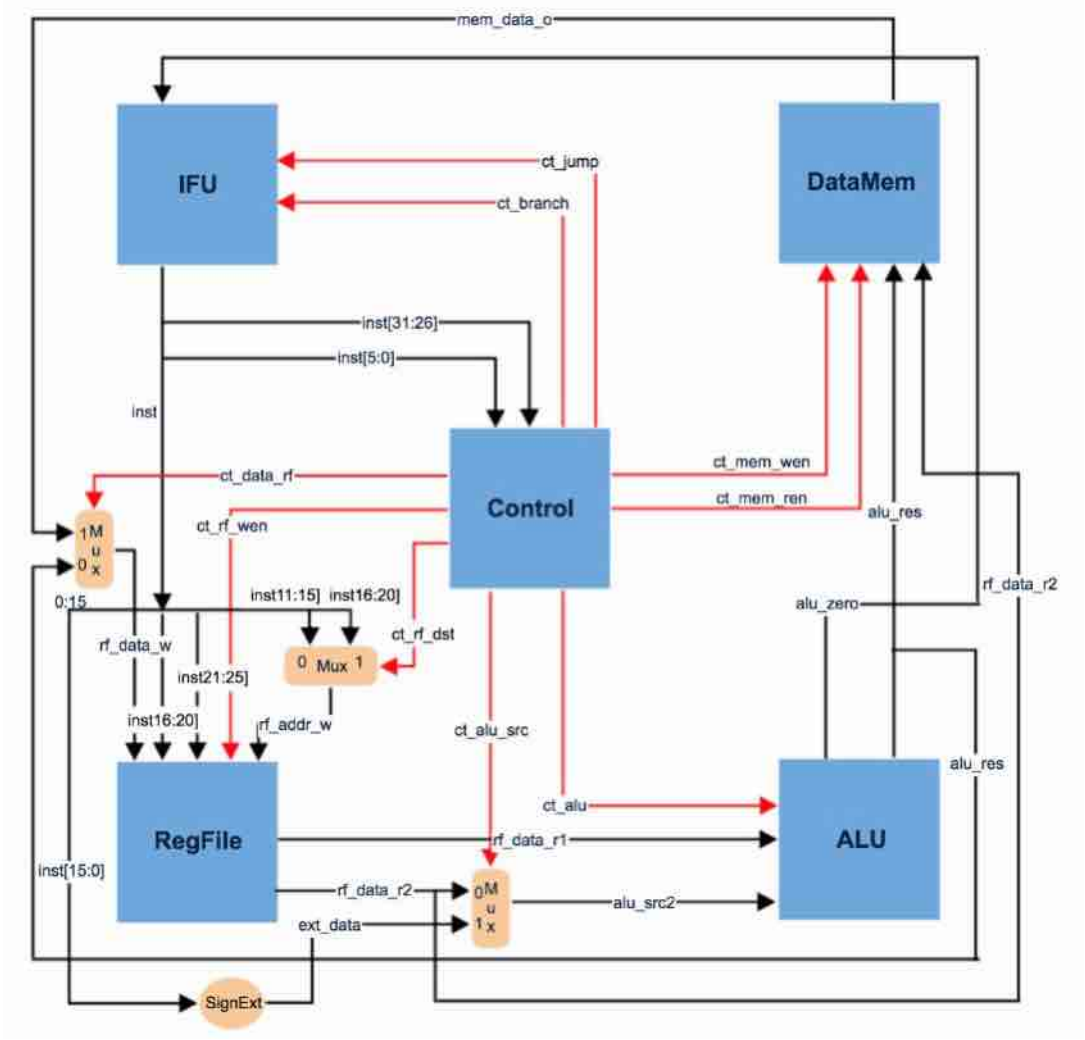


图 11　顶层设计图，接口名字不做限制，省略了时钟与复位信号

经过设计分析最终得出了模块之间的连接关系图也就是顶层模块，单周期 MIPSCPU 已经有了雏形，接下来将具体的看一看每一个模块的内部实现。

3.2.2 模块设计

IFU 模块：

IFU（Instruction Fetch Unit，取指单元）模块的功能就是从指令内存中取指令。在单周期 CPU 设计中有两个内存模块：IFU 模块内部的指令内存和 DataMem 模块的数据内存。其中指令内存为只读，数据内存可读可写。指令内存比较简单，一个常见的写法是：

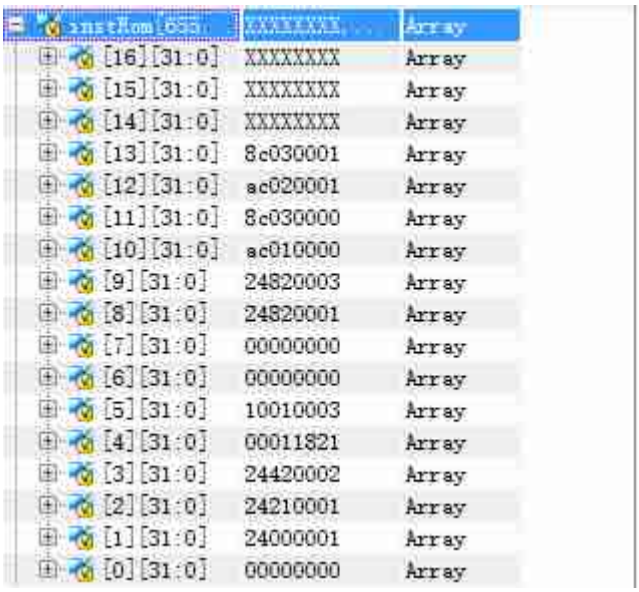
```
reg[31:0] inst_addr;
reg[31:0] instRom[65535:0];//总共有 2^16 个寄存器，内存单元宽度为 4 字节
initial $readmemh("inst.data",instRom);
assign inst=instRom[inst_addr[31:2]];
```

这种写法不需要时钟控制，任意时刻的数据总是与给出的地址对应，符合单周期 CPU 的要求。有几点需要注意：

1. 32 位地址、内存单位大小是 1 字节的的最大可寻址空间是 2^32B=4GB，我们在仿真时不需要这么大的内存空间，所以在实现上我们可以忽略高位地址，只用低位进行寻址即可。
2. 上面的代码的内存单位是 32bit。如果设计成 8bit 或者 16bit 的内存单位，就需要注意字节序的问题。一般将小端的 MIPS 指令称为 mipsel 与大端序 MIPS 区分。以数据 0x0A0B0C0D 为例，如果起始地址是 0x0，内存单位是 8bit，那么在大端序情况下，低地址存放高位，即 0x0 地址存放 0x0A，0x1 地址存放 0x0B，0x2 地址存放 0x0C，0x3 地址存放 0x0D；小端序是低地址存放低位，0x0 地址存放 0x0D，0x1 地址存放 0x0C，0x2 地址存放 0x0B，0x3 地址存放 0x0D。
3. 注意到上面的代码使用了$readmemh 函数，这个函数的功能是从指定文件读取数据初始化寄存器的内容。读取的数据文件的内容只能包含：空白位置(空格，换行，tab 和 form-feeds)，注释行(//)，十六进制的数字。

```
00000000
//第0条指令
24000001
//addiu $0 $0 1
24210001
//addiu $1 $1 1
24420002
//addiu $2 $2 2
00011821
//addu $3 $0 $1
10010003
//beq $0 $1 8
//08000002
//跳转到第2条指令
00000000
00000000
24820001
24820003
ac010000
8c030000
ac020001
8c030001
```

inst.data 文件内容

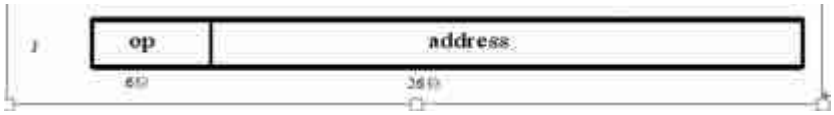


执行$readmemh("inst.data",instRom)后的instRom

另外这个文件的位置也要注意。在 Vivado 中执行仿真时，“当前目录”并不是工程的根目录：如果工程名字是 soc，那么行为仿真时的“当前目录”实际上位于“soc\soc.sim\sim_1\behav\xsim”文件夹。时序仿真等和这个类似，但是文件夹不一样。注意创建工程后并没有上述文件夹，需要先执行一遍对应的仿真（让 Vivado 创建对应的目录），然后把写好的数据文件放到对应目录。直接写出数据文件的绝对地址也可以，但是将工程复制到其他电脑上时要注意修改路径。

$readmemh 只接受十六进制的数据。另外一个类似的函数$readmemb 接受二进制的数据，语法和前者类似。

IFU 的内存模块构建之后，就可以从指令内存中取指令了。当 IFU 第一条指令从地址为 0x00 处取指。如果执行的指令是跳转指令，例如 jump 指令。



指令格式如上，需要执行的操作是 PC={PC[31:28], address, 2'b00}。如果执行的是分支指令，且满足条件，需要执行的操作是 PC=PC+立即数<<2。如果不是上述两种情况，那么 PC=PC+4。

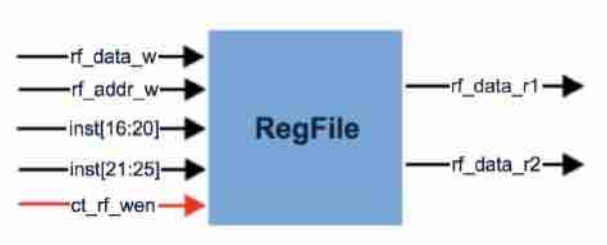
据此完成 IFU 模块的实现代码：

```
1.  `timescale 1ns / 1ps
2.  module IFU(
3.  input clk,rst,
4.  input alu_zero,ct_branch,ct_jump,
5.  output[31:0] inst
6.      );
```

```
7.      reg[31:0] pc;
8.      reg[31:0] instRom[65535:0];//指令存储器空间为 256KB
9.      wire[31:0] ext_data;//符号扩展后的值
10.
11.     initial $readmemh("data/inst.data",instRom);//加载指令文件到存储器
12.
13.     assign inst=instRom[pc[31:2]];//取指令
14.
15.     assign ext_data = {{16{inst[15]}},inst[15:0]};//符号扩展
16.
17.     always @ (posedge clk)
18.         if(!rst)
19.             pc <= 0;
20.         else begin
21.             if(ct_jump)
22.                     //此处需要补充代码
23.             else if(ct_branch && alu_zero)
24.                     //此处需要补充代码
25.             else
26.                 pc <= pc + 4;
27.         end
28. endmodule
```

RegFile 模块：

MIPS 的寄存器堆是 32 个 32 位的寄存器。在单周期 CPU 设计中，我们希望能在给出地址的同时就能在出口得到对应的数据；为了节约时钟，我们可以在每个 clk 的下边沿检查控制信号来确定要不要改写寄存器的内容。考虑到 MIPS 指令中很多指令都要求 rs、rt 两个参数，所以我们的寄存器堆设计为具有两个读端口、一个写端口：



其中，ct_rf_wen 是写信号。在时钟的下边沿，寄存器堆会检查这个信号。如果它为 1，那么就将 rf_data_w 的 32 比特数据写入 rf_addr_w 表示的寄存器。余下的四个信号是读取地址和数据输出端口，地址为 5 位（2^5=32）,数据输出为 32 位。

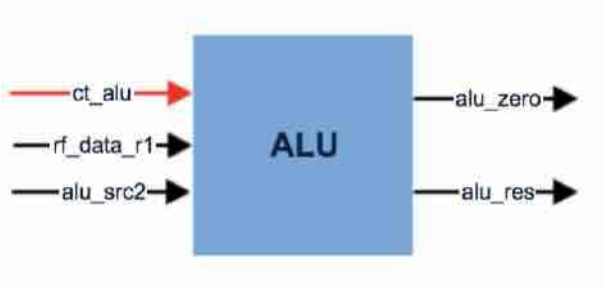
代码如下(**图中接口名是顶层模块连接时的名字与模块内部不一定一样**)：

```
`timescale 1ns / 1ps
module RegFile(
    input clk,
    // 写使能信号
    input rf_wen,
    // 读地址
    input[4:0] rf_addr_r1,
    input[4:0] rf_addr_r2,
    // 写入地址和写入数据
    input[4:0] rf_addr_w,
    input[31:0] rf_data_w,
    // 输出端口
    output[31:0] rf_data_r1,
    output[31:0] rf_data_r2
    );

    reg[31:0] file[31:0];

    integer i;
    initial begin
        for(i = 0; i < 32; i=i+1) file[i] = 32'b0;
    end

    assign rf_data_r1 = file[rf_addr_r1];
    assign rf_data_r2 = file[rf_addr_r2];


    always@(negedge clk) begin
        if (rf_wen) begin
            //在此补充完成控制信号控制寄存器堆写操作
        end
    end
endmodule
```

ALU 模块：

ALU 主要用来执行加减法、比较等指令。在单周期 CPU 中只要求实现在一个时钟周期能够完成的指令，所以我们将 ALU 设计为不受时钟控制的组合逻辑电路，也就是说在任意时刻，ALU 的输出总是与此时的输入和控制信号相对应。首先我们来分析指令结构。ALU 参与的指令主要有 R 型和 I 型：在 R 型指令中，op 和 funct 两段确定了 ALU 执行的操作；在 I 型指令中则是由 op 来确定。将这

两个信号传递给控制器，然后指示 ALU 模块做什么运算。这样我们就得到了一个简单的 ALU 通路。下面是 ALU 的设计图：



rf_data_r1 和 alu_src2 是 ALU 的两个操作数，alu_res 表示运算结果，alu_zero 用于表示结果是否为 0。aluct_sel 信号指示 ALU 执行什么样的操作。aluct_sel 的取值可以自由确定，属于实现细节。下面给出一个控制信号表样例：

指令 inst[31:16]	aluct_sel	
addu(000 000)	0010	加法
lw(100 011)	0010	
sw(101 011)	0010	
addiu(001 001)	0010	
beq(000 100)	0110	减法

beq 指令在两个输入相等时跳转，我们可以用减法来模拟：如果两个数相等，那么 alu_zero 就是 1。将 alu_zero 接入 IFU 模块，配合其他控制信号，就可以实现跳转功能。

注意到上表中还出现了 lw 和 sw 两个指令。以 sw 为例，它的格式是：

1010 11ss ssst tttt iiii iiii iiii iiii

其作用是将$t 寄存器的内容写入 MEM[$s+offset]，offset 是后 16 位确定的偏移量。注意到内存地址做了一个加法，所以我们可以将 sw 也看作一个“加法指令”，它的结果 alu_res 连接到内存的地址输入端。通过控制器的控制信号，我们就可以实现用 ALU 计算出目标地址，然后将寄存器的内容写入到内存中去。这样设计可以节约一个专门用于计算地址的模块。lw 也是一样的原理，具体的数据流向是由控制器确定的。

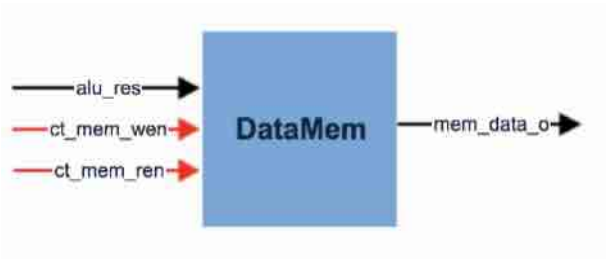
结合 Control 模块部分的介绍，补充完整 ALU 模块代码：

```
1.  `timescale 1ns / 1ps
2.  module ALU(
3.  input rst,
4.  input[3:0] alu_ct,
5.  input[31:0] alu_src1,alu_src2,
6.  output alu_zero,
```

```
7.  output reg [31:0] alu_res
8.      );
9.      assign alu_zero= (alu_res==0)?1:0;
10.     always@(*)
11.         if(!rst)begin
12.             alu_res = 32'bz;
13.         end
14.         else begin
15.             case(alu_ct)
16. //在此补充代码：当 alu_ct 为 4'b0010，执行加法运算；为 4'b0110 时，执行减法运算。
17.                 default:begin end
18.             endcase
19.         end
20. endmodule
21.
```

DataMem 模块：

数据内存的实现需要加上写功能和额外的控制信号：



其中 mem_wen 为写信号，每个时钟的下边沿检测此信号，如果它是 1，那么就将数据 mem_data_i 写入 mem_addr 表示的内存地址；mem_ren 是读信号，任意时刻此信号为 1 时，就把 mem_addr 表示的内存地址的内容输出到 mem_data_o。

指令和数据各自使用独立总线的设计属于哈佛架构。MIPS 指令集没有给出写指令内存的指令。需要指出的是，在我们的单周期 CPU 中的指令内存和数据内存并不是实际意义上的内存，而更类似 CPU 内部的指令缓存和数据缓存。现代操作系统一般都具有动态加载程序的功能，而指令内存的只读特性是与之冲突的。实践中，在内存里指令和数据往往是在一起的，利用 CPU 的高速缓存区分为“指令内存”和“数据内存”。可以理解为，在 CPU 内部我们实现的是哈佛架构，在 CPU 外部则是冯诺依曼模型（与效率、功耗等指标相比，是否忠实地实现某种架构往往微不足道）。这样我们就可以通过数据内存修改指令，允许动态加载程序

的功能。

据此补充完整 DataMem 模块代码：

```
1.  `timescale 1ns / 1ps
2.  module DataMem(
3.  input clk,mem_wen,mem_ren,
4.  input[31:0] mem_addr,
5.  input[31:0] mem_data_i,
6.  output[31:0] mem_data_o
7.      );
8.      reg[7:0] data_mem0[0:66535];
9.      reg[7:0] data_mem1[0:66535];
10.     reg[7:0] data_mem2[0:66535];
11.     reg[7:0] data_mem3[0:66535];
12.     //写操作
13.
14.     always@(negedge clk)begin
15.         if(mem_wen)begin
16.             data_mem0[mem_addr[31:2]] <= mem_data_i[7:0];
17.             data_mem1[mem_addr[31:2]] <= mem_data_i[15:8];
18.             data_mem2[mem_addr[31:2]] <= mem_data_i[23:16];
19.             data_mem3[mem_addr[31:2]] <= mem_data_i[31:24];
20.         end
21.     end
22.     //读操作
23.     //在此补充数据存储读操作代码
24.
25. endmodule
```

Control 模块：

控制	信号名	R型	lw	sw	beq	j	addiu
输入	ct_inst(inst[31:26])	0	1	1	0	0	0
		0	0	0	0	0	0
		0	0	1	0	0	1
		0	0	0	1	0	0
		0	1	1	0	1	0
		0	1	1	0	0	1
输出	ct_rf_dst	1	0	x	x	x	0
	ct_rf_wen	1	1	0	0	0	1
	ct_alu_src	0	1	1	0	x	1
	ct_alu_op	10	00	00	01	xx	00
	ct_branch	0	0	0	1	0	0
	ct_mem_ren	0	1	0	0	0	0
	ct_mem_ wen	0	0	1	0	0	0
	ct_data_rf	0	1	x	x	x	0
	ct_jump	0	0	0	0	1	0

Control 模块内部还包含负责控制 ALU 的子模块 ALUCt， 复位信号、指令的 0-5 位 funct 字段和 2bit 的 ct_alu_op 作为内部信号送到子模块 ALUCt 中，ALUCt 返回 4bit 的 ct_alu 信号由 Control 模块输出。

据此补充完整 ALUCt 模块实现代码：

```
1.  module ALUCt(
2.      input rst,
3.      input[5:0] funct,
4.      input[1:0] alu_ct_op,
5.      output reg[3:0] alu_ct
6.      );
7.      always@(*)
8.          if(!rst)
9.              alu_ct <= 0;
10.         else
11.             case(alu_ct_op)
12.                 2'b00:alu_ct= 4'b0010;
13.                 2'b01:alu_ct= 4'b0110;
14.                 2'b10:begin case(funct)
15. //在此补充代码：当指令中 funct 段为 100001 时，alu_ct 输出 4'b0010（执行加法操作）。
16.                         default:begin
17.                         end
18.                     endcase end
```

```
19.                 default:begin end
20.             endcase
21. Endmodule
```

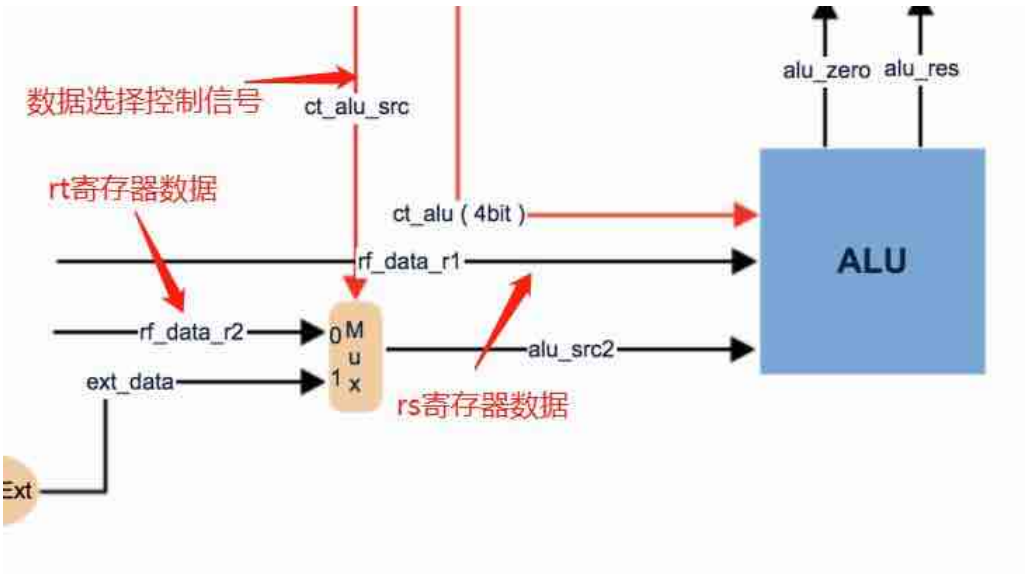
通过看数据通路设计图，可以知道 Control 模块的功能是根据指令的高六位（指令类型，例如 R 型指令、lw 指令）和指令的低六位产生各个模块的控制信号。Control 模块对各个模块产生的控制信号及作用如下：

模块	控制信号	作用
IFU	ct_jump	执行跳转指令时变为有效信号
	ct_branch	执行分支指令时变为有效信号
DataMem	ct_mem_wen	往 DataMem 写入数据时变为有效信号
	ct_mmm_ren	从 DataMem 读出数据时变为有效信号
ALU	ct_alu	选择 ALU 要执行的运算，例如选择执行加法或其他运算
Mux	ct_alu_src	二选一多路选择器的控制信号
	ct_rf_dst	
	ct_data_rf	
RegFile	ct_rf_wen	往 RegFile 写入数据数据时变为有效信号

以 R 型指令为例，讲解各个控制信号的变化。当输入一条指令时，首先应该识别该指令的类型。识别的方法如下：

指令的高 6 位是标明了指令类型。R 型的指令高六位是 ct_inst[5:0]=6’b000000 ， 定义一个变量 inst_r, 令 inst_r=(!ct_inst[5])&&(!ct_inst[4])&&(!ct_inst[3])&&(!ct_inst[2])&&(!ct_inst[1])&&(!ct_inst[0])当指令为 R 型时，inst_r=1,当指令不是 R 型时，inst_r=0.其余指令的识别方法可以此为参考。

此时已经知道指令为 R 型。我们要执行的操作是(rd)=(rs)《运算符》(rt).首先需要执行的操作是在 RegFile 模块中读取 rs、rt 寄存器中的数据，RegFile 模块不需要读使能信号，所以 Control 对这一步骤不需要控制，然后是将读到的两个数据放入 ALU 中。

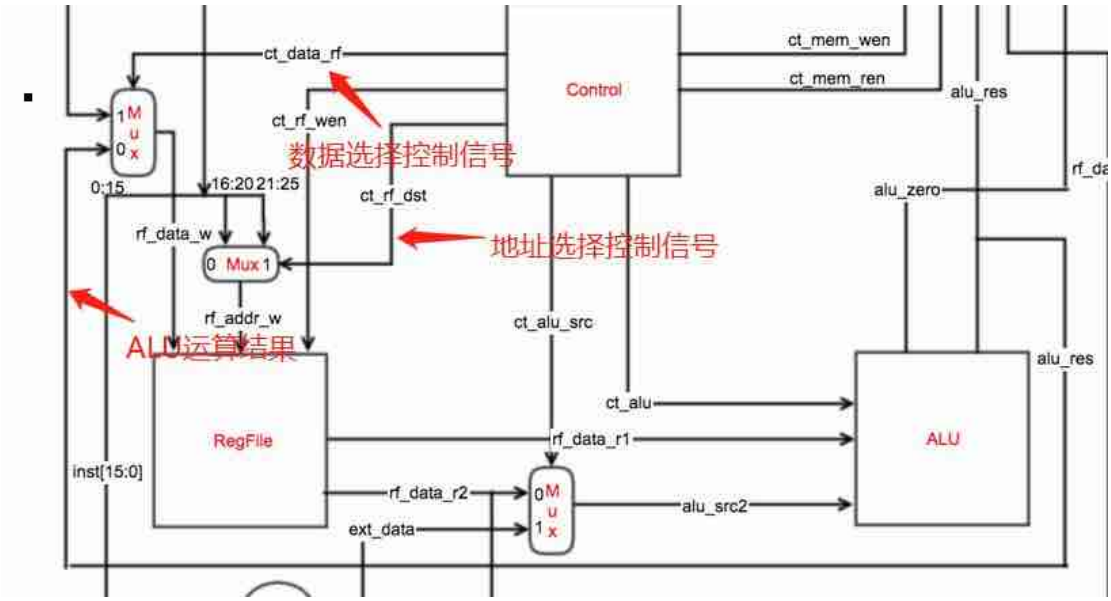


从图上可以看出，rs 寄存器数据直接放入 ALU 中，但是 rt 寄存器数据需要经过二选一多路选择器，才能放入 ALU 中。所以当检测到指令是 R 型指令时，将 ct_alu_src 设置为 0，这样我们便可以选中 rt 寄存器中的数据进入 ALU。

两个源数据进入 ALU 后。紧接着 ALU 要进行运算。ALU 执行何种运算需要 ct_alu 信号的控制。ct_alu 信号的设置方法如下：

R 型指令执行的运算类型需要依据指令的低六位 funct[5:0]来判断。在 Control 模块内用一个专门的模块 ALUCt 来产生 ct_alu 信号。将指令的 0-5 位 funct 字段和 2bit 的 ct_alu_op 作为内部信号送到子模块 ALUCt 中作为 ct_alu 信号的判断依据。使用 ct_alu_op 信号来标明当前指令的种类。通过 ct_alu_op 信号判断指令为 R 型，需要进一步判断 funct 段，如果 funct 段标明要执行加法运算。给 ct_alu 赋值为 4’b0010，控制 ALU 进行加法运算。

ALU 执行之后，需要将 ALU 的运算结果通过二选一多路选择器放入到寄存器 rd。



从上图可以看出，需要将数据选择控制信号 ct_data_rf 设置为 0，选中 ALU 的运算结果。将地址选择控制信号设置为 1，选择 rd 寄存器的地址。

对于 Control 产生的其他外部控制信号，因为这条指令的执行不需要用到其余的操作，所以都设为无效信号 0。

据此补充完整控制模块 Control 的代码：

```
1.  `timescale 1ns / 1ps
2.
3.  module Control(
4.      input rst,
5.      input[5:0] ct_inst,
6.      input[5:0] aluct_inst,
7.      output ct_rf_dst,
8.      output ct_rf_wen,
9.      output ct_alu_src,
10.     output[3:0] ct_alu,
11.     output ct_mem_wen,
12.     output ct_mem_ren,
13.     output ct_data_rf,
14.     output ct_branch,
15.     output ct_jump
16.     );
17.     wire inst_r,inst_lw,inst_sw,inst_beq,inst_j,inst_addiu;
18.     wire[1:0] ct_alu_op;
19.     ALUCt aluct0(rst,aluct_inst,ct_alu_op,ct_alu);
20.     //二级逻辑阵列
21.     //与阵
```

```
22.     assign inst_r  =
    (!ct_inst[5])&&(!ct_inst[4])&&(!ct_inst[3])&&(!ct_inst[2])&&(!ct_inst[1])&&(
    !ct_inst[0]);
23. //在此补充完整其余 5 条指令 inst_lw，inst_sw，inst_beq， inst_j，inst_addiu 的表
    达式。
24.
25.     //或阵
26.     assign ct_rf_dst = rst?inst_r:0;
27.     assign ct_rf_wen = rst?inst_r || inst_lw||inst_addiu:0;
28.     assign ct_alu_src = inst_lw || inst_sw||inst_addiu;
29.     assign ct_alu_op[1:0] = {inst_r,inst_beq};
30. //在此补充完整其余控制信号的表达式：ct_branch, ct_mem_ren, ct_mem_wen,
    ct_data_rf, ct_jump
31.
32. endmodule
```

CPU 顶层模块

综合上述模块，可分析出顶层模块 CPU 的实现如下：

```
1.  `timescale 1ns / 1ps
2.  module CPU(
3.  input clk,rst
4.      );
5.      //ifu
6.      wire[31:0] inst;
7.      //Contol 模块输出的控制信号
8.      wire
    ct_rf_dst,ct_rf_wen,ct_alu_src,ct_data_rf,ct_branch,ct_jump,ct_mem_wen,ct_me
    m_ren;
9.      wire[3:0] ct_alu;
10.     //RegFile 模块的输入输出
11.     wire[4:0] rf_addr_w;
12.     wire[31:0] rf_data_r1,rf_data_r2,rf_data_w;
13.     //ALU 模块的输入输出
14.     wire alu_zero;
15.     wire[31:0] alu_src2;
16.     wire[31:0] alu_res;
17.     //符号扩展的结果
18.     wire[31:0] ext_data;
19.     //DataMem 的输出
20.     wire[31:0] mem_data_o;
```

```
21.     //选择要写的寄存器地址
22.     assign rf_addr_w = ct_rf_dst?inst[15:11]:inst[20:16];
23.     //选择要写入寄存器堆的数据
24.     assign rf_data_w = ct_data_rf?mem_data_o:alu_res;
25.     //alu_src2 是指令后 16 位符号扩展的结果或者寄存器堆读的第二寄存器的值
26.     assign ext_data = {{16{inst[15]}},inst[15:0]};
27.     assign alu_src2 = ct_alu_src?ext_data:rf_data_r2;
28.
29.      IFU ifu0(clk,rst,alu_zero,ct_branch,ct_jump,inst);
30.      Control
    ct0(rst,inst[31:26],inst[5:0],ct_rf_dst,ct_rf_wen,ct_alu_src,ct_alu,ct_mem_w
    en,ct_mem_ren,ct_data_rf,ct_branch,ct_jump);
31.      RegFile
    rf0(clk,ct_rf_wen,inst[25:21],inst[20:16],rf_addr_w,rf_data_w,rf_data_r1,rf_
    data_r2);
32.      ALU alu0(rst,ct_alu,rf_data_r1,alu_src2,alu_zero,alu_res);
33.      DataMem
    datamem0(clk,ct_mem_wen,ct_mem_ren,alu_res,rf_data_r2,mem_data_o);
34. endmodule
```

3.3 仿真

3.3.1 分模块仿真

在 Vivado 中仿真有下面几种：

1. behavioral simulation 行为级仿真，也是通常说的功能仿真
2. post-synthesis function simulation 综合后的功能仿真
3. post-synthesis timing simulation 综合后带时序信息的仿真
4. post-implementation function simulation 布线后的功能仿真
5. post-implementation timing simulation 布线后的时序仿真，最接近真实的时序波形

仿真之前需要编写 tb 文件（testbench），该文件指定了对应模块的输入如何随着时间而变化。我们以一个简单的 PC 为例，代码如下：

```
module PC(
    input clk,
    input rst,
    output [31:0] addr
    );
    reg[31:0] pc = 32'b0;
    assign addr = pc;
    wire[31:0] nPC;
    assign nPC = pc + 4;
    always@(posedge clk or posedge rst) begin
        if(rst) pc = 32'b0;
        else pc = nPC;
    end
endmodule
```

包含一个时钟信号、一个重置信号、一个输出地址。每当时钟来临时，将 PC 加 4。下面我们编写对应的 testbench 验证 PC 的功能：

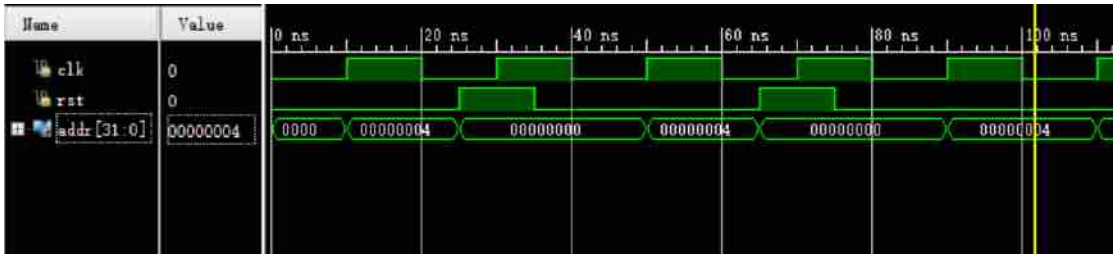
在 Vivado 中按下 Alt+A，然后选择“Add or create simulation source”,创建一个名字为 PC_tb.v 的文件。创建后会弹出编辑端口的窗口。Tb 文件不需要任何输入输出，所以直接回车即可。PC_tb.v 内容如下：

```
`timescale 1ns / 1ps

module PC_tb;
    reg clk = 0;
    reg rst = 0;
    wire[31:0] addr;
    PC pc(clk, rst, addr);
    initial begin
```

```
        forever #10 clk = ~clk;
    end
    initial begin
        #25 rst = 1;
        #10 rst = 0;
        #30 rst = 1;
        #10 rst = 0;
    end
endmodule
```

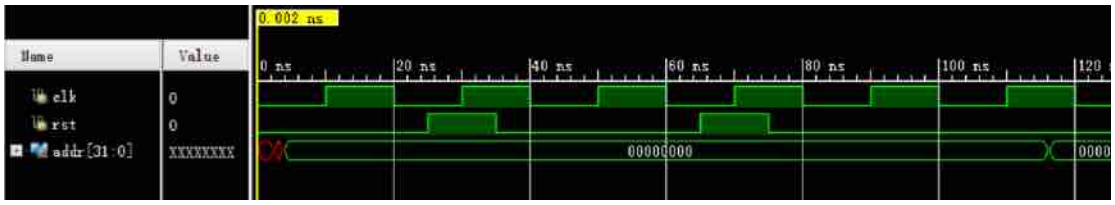
这个 tb 的内容是，首先将 clk 每 10ns 翻转一次，也就是一个周期为 20ns 或 50MHz 的时钟。然后在仿真的第 25ns 时，使重置信号有效，持续 10ns；仿真的第 65ns 时，使重置信号有效，持续 10ns。

在 Flow 菜单中选择 Run simulation->Behavoral Simulation，执行完毕后可以看到仿真波形：

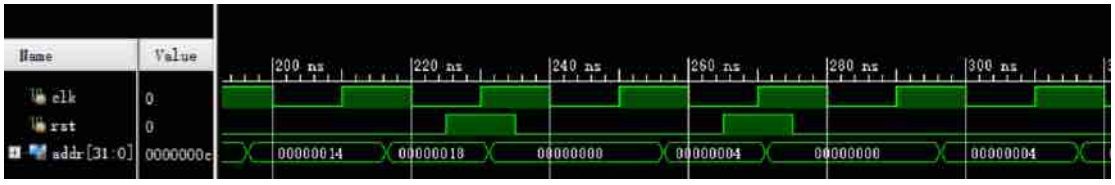


可以看到，在第二个时钟来临时，地址被重置为 0，下一个时钟后加上 4；在第 4 个时钟来临时，地址再次被重置为 0，符合我们的预期。

行为仿真主要是执行语义上的检查，没有引入实际电路中普遍存在的延时，所以一般我们需要做时序仿真来确定在有延时的情况下我们的电路能否正常工作。保持 tb 文件不变，按下 F11 执行综合。综合过程可能会给出多个警告、严重警告或者错误，建议仔细阅读，找出潜在的错误。综合完毕后，对应的仿真选项就可以选择，执行 Post-synthesis timing simulation，结果如下：



这时我们发现输出信号直到 110ns 之后才有我们期望的输出，在这前 110ns 我们的重置信号没有任何作用。这可能是留给设备的“初始化”时间。为了避免这个问题，我们修改 tb 文件，在第二个 initial 段的开头加上一行“#200;”，也就是将重置信号延时 200ns。再次执行仿真，这次我们直接看 200ns 之后的波形：

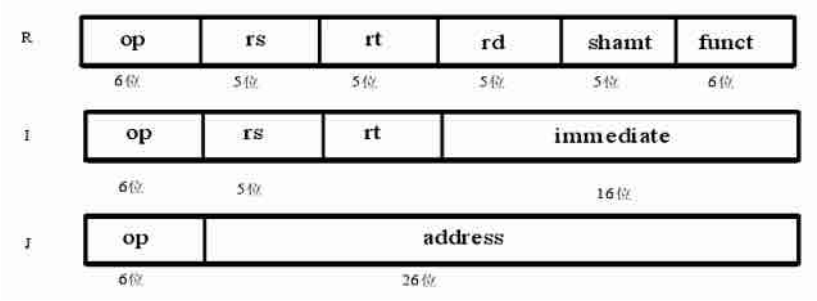


结果同样符合预期，同时注意到 addr 变化与对应的时钟相比多了一些延迟。因

为这个延迟的存在，tb 中的 clk 的频率不能设置太高，否则即使源代码正确，输出结果也是不正常的。

3.3.2 整体模块仿真

下面的仿真按照样例 CPU 的结构实现，要求事先写好 inst.data 文件（inst.data 参见前文）。第一条指令是 24000001，这是 16 进制，换成 2 进制形式为：0010_0100_0000_0000_0000_0000_0000

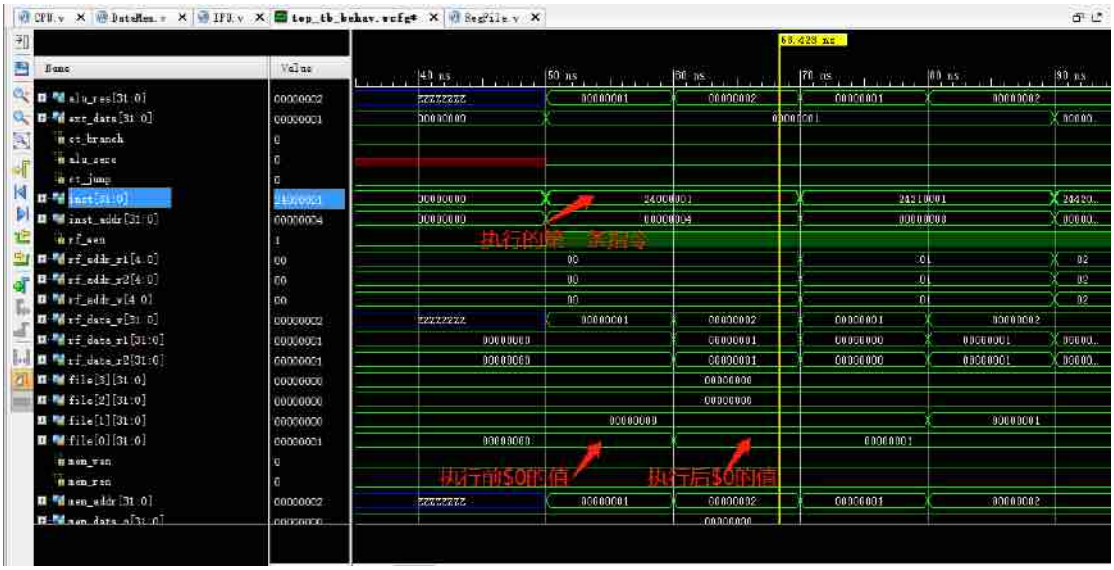


Mips 三种不同的指令类型

字段名	字段意义说明
op	指令操作码
rs	源操作数 1
rt	源操作数 2 或目的操作数
rd	目的操作数
shamt	位移量，在移位运算中存储位移量
funct	功能码，当指令为 R 类型时，根据功能码判断指令的操作类型
immediate	当其为 I 型指令时为立即数，分支指令时，为跳转偏移量
address	跳转绝对地址，为指令 J 的指令格式

指令字段说明

参考上述两张图，可以得知这条指令的操作码是 001001。通过查询可知，这是 addiu 指令,属于 I 型指令，执行的操作是 rt=rs+立即数。rs=$0,rt=$0.立即数为 1。加入 RegFile 模块中的 file[1][31:0]变量的仿真波形如下图：

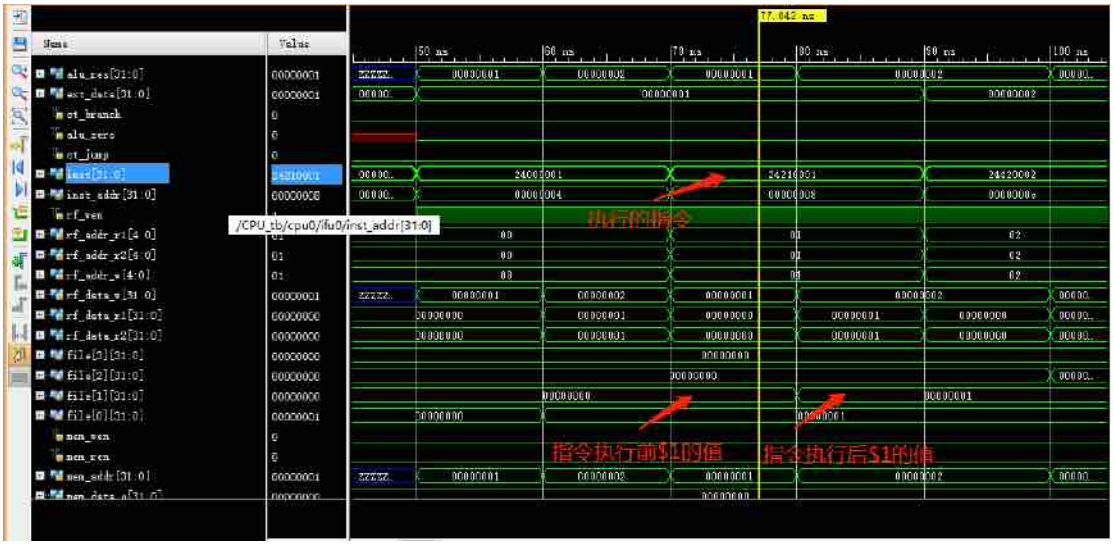


指令执行之前$0 的值为 0，指令执行之后，$0 的值为 1，结果正确。

IFU 从 inst.data 取到的第二条指令是 24210001，换成进制：

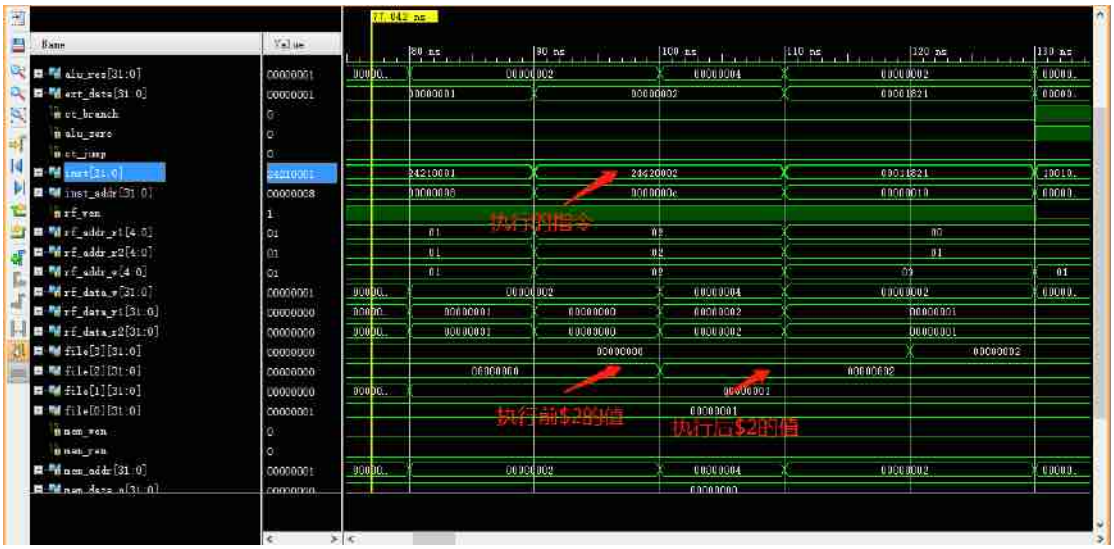
0010_0100_0010_0001_0000_0000_0000_0001

参考指令类型和指令段说明，我们可以得知这条指令的操作码还是 001001，rs=$1,rt=$1,立即数是 1，执行的操作是 rt=rs+立即数。加入 RegFile 模块中的 file[0][31:0]变量的仿真波形后，波形如下：

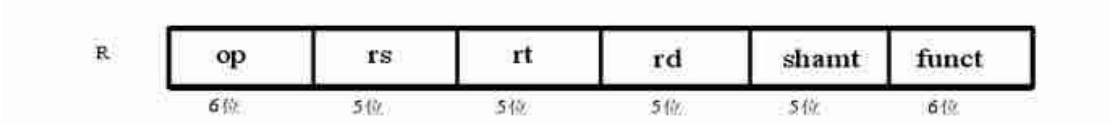


指令执行前$1 的值为 0，指令执行后$1 的值为 1.执行结果正确

执行的第三条指令是 24420002，可知操作码是 001001，rt=$2,rs=$2.,立即数为 2，执行的操作是 rt=rs+立即数。加入 RegFile 模块中的 file[2][31:0]变量的仿真波形。



执行前$2 的值为 0，执行后$2 的值为 2.结果正确。
执行的第四条指令是 00011821，换成二进制为：
0000_0000_0000_0001_0001_1000_0010_0001
操作码是 000000，查询可知，这属于 R 型指令。指令格式如下：

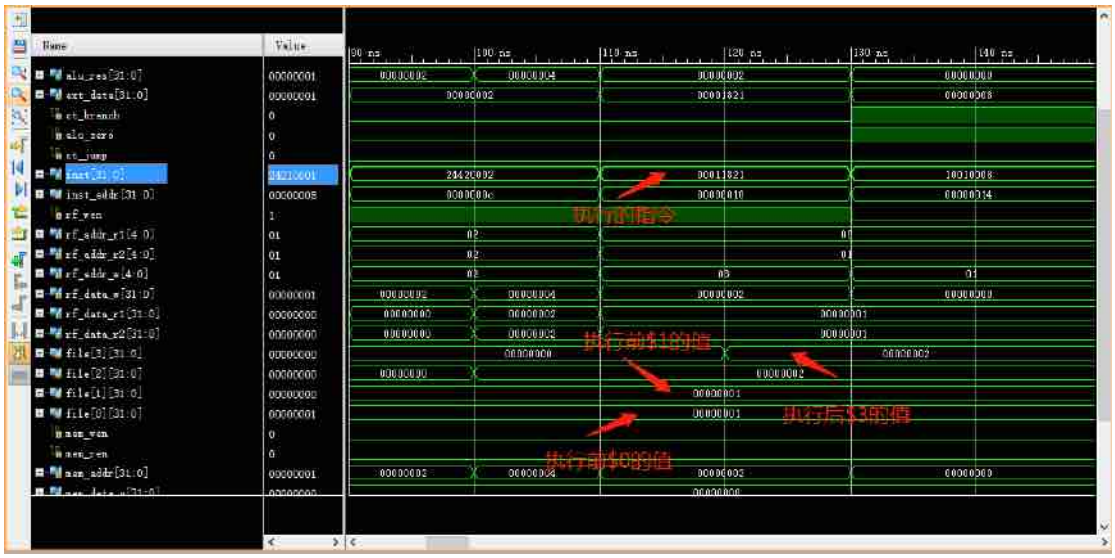


最后 6 位 funct 字段功能码为 100001,查询可知这是要执行 addu 指令。执行的操作是 rd=rs+rt.

ADDU – *Add unsigned (no overflow)*

Description:	Adds two registers and stores the result in a register
Operation:	$d = $s + $t; advance_pc (4);
Syntax:	addu $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0001

rs=$0,rt=$1,rd=$3.换言之执行的操作是$3=$0+$1. 加入 RegFile 模块中的 file[3][31:0]变量的仿真波形。

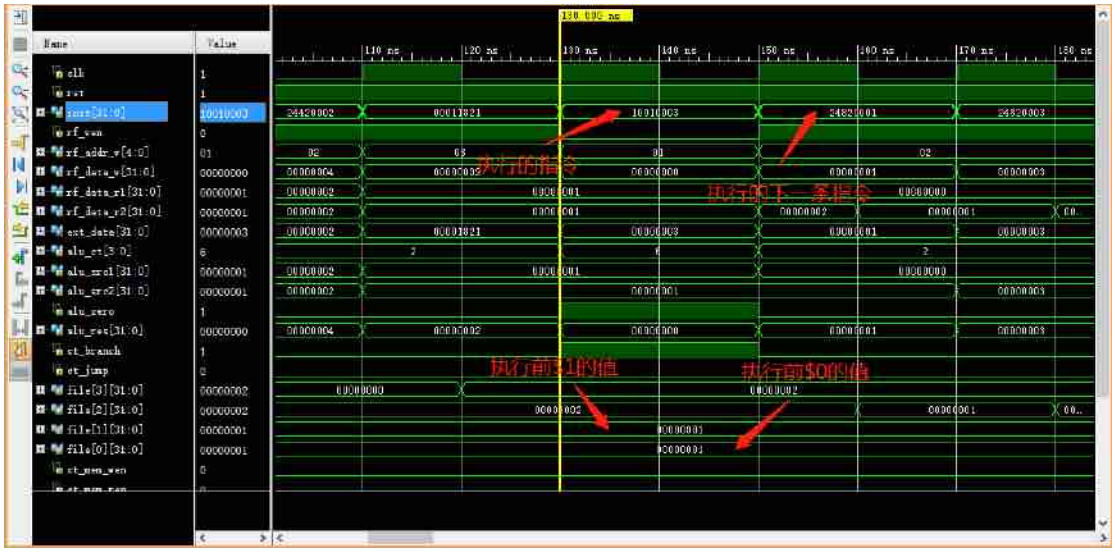


执行前$0 的值为 1，$1 的值为 1，执行后$3 的值为 2，执行结果正确。
执行的第五条指令是 10010008，换成二进制为：
0001_0000_0000_0001_0000_0000_0000_1000
操作码为 000100，查询可知是 beq 指令。

BEQ – *Branch on equal*

Description:	Branches if the two registers are equal
Operation:	if $s == $t advance_pc (offset << 2)); else advance_pc (4);
Syntax:	beq $s, $t, offset
Encoding:	0001 00ss ssst tttt iiii iiii iiii iiii

属于 I 型指令。执行的操作是如果 rs 寄存器的值等于 rt 寄存器的值，那么下一条需要执行的指令的地址为 PC+immediate。否则顺序执行下一条指令。
rs=$0,rt=$1,立即数为 3。



$0 的值为 1，$1 的值为 1，$0=$1,所以执行的下一条的指令的地址为 PC+3<<2，也就跳转到下面第三条指令。示例如下：

下面的第二条指令为 24820001,执行结果正确。

3.3.3 运行简单程序

下面我们编写一个简单的求斐波那契数列前 N 项的程序（N>=2）。对应的 C 语言代码是：

```
int fac(int n)
{
    assert(n >= 2); // 断言，没有作用
    int x0 = 1, x1 = 1;
    // 这里严格说要用小于的但是此处未实现（//▽//）
    for(int i = 1; i != n; ++i) {
        int x2 = x0 + x1;
        x0 = x1; x1 = x2;
    }
    return x1;
}
```

我们来把它改写成 MIPS 汇编程序。从代码我们可以看到它至少需要加法、跳转和移动指令（move）。

MIPS 指令集还规定了很多伪指令。也就是说这些指令实际是用其他指令实现的。例如，移动指令 move $t, $s，我们可以用 addiu $t,$s,0 来模拟；将 16 位立即数加载到寄存器 li $t, C，我们可以用 addiu $t, $0, C 来模拟。

变量我们就以寄存器代替。下面是汇编程序：

```
nop                  // 我们的 IFU 结构决定了第一条是空，没有作用
li $t0, 1            // t0 是 x0
li $t1, 1            // t1 是 x1
li $t2, 10           // t2 号作为 n，设置为 10
li $t3, 1            // t3 号做为 i
beq $t3, $t4, 6      // 如果 i 和 n 相等那么往后跳 6 条指令
addu $t5, $t0, $t1 // x2 = x0 + x1
move $t0, $t1        // x0 = x1
```

```
move $t1, $t5       // x1 = x2
addiu $t3, $t3, 1   // i++
j 5                 // 跳转到 beq 这条指令
```

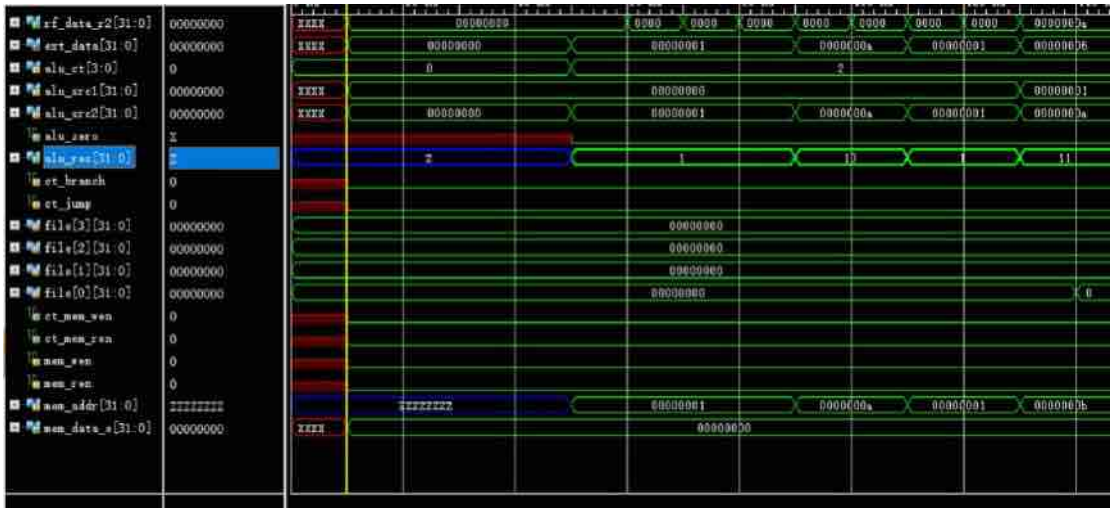
然后我们将伪指令替换为实际的指令，同时把 t 开头的寄存器替换为实际的寄存器编号：

```
nop                 // 我们的 IFU 结构决定了第一条是空，没有作用
addiu $8, $0, 1     // t0 是 x0
addiu $9, $0, 1     // t1 是 x1
addiu $10, $0, 10   // t2 号作为 n，设置为 10
addiu $11, $0, 1    // t3 号做为 i
beq $11, $10, 6     // 如果 i 和 n 相等那么往后跳 6 条指令
addu $12, $8, $9    // x2 = x0 + x1 ， t4 是 x2
addiu $8, $9, 0     // x0 = x1
addiu $9, $12, 0    // x1 = x2
addiu $11, $11, 1   // i++
j 5                 // 跳转到 beq 这条指令
0010 01ss ssst tttt iiii iiii iiii iiii
```

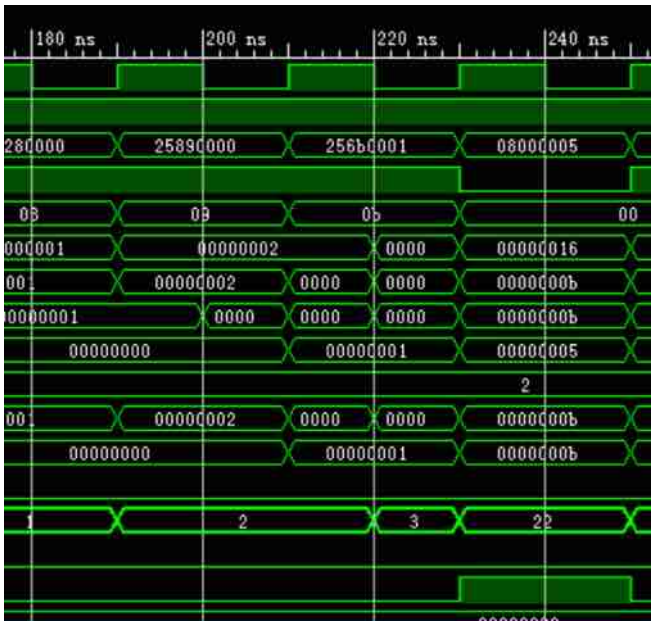
然后翻译成二进制代码，$8 是 01000，$9 是 01001，$10 是 01010，$11 是 01011，$12 是 01100。

```
0000 0000 0000 0000 0000 0000 0000 0000 // nop
0010 0100 0000 1000 0000 0000 0000 0001 // addiu $8, $0, 1
0010 0100 0000 1001 0000 0000 0000 0001 // addiu $9, $0, 1
0010 0100 0000 1010 0000 0000 0000 1010 // addiu $10, $0, 10
0010 0100 0000 1011 0000 0000 0000 0001 // addiu $11, $0, 1
0000 0001 0110 1010 0000 0000 0000 0110 // beq $11, $10, 6
0000 0001 0000 1001 0110 0000 0010 0001 // addu $12, $8, $9
0010 0101 0010 1000 0000 0000 0000 0000 // addiu $8, $9, 0
0010 0101 1000 1001 0000 0000 0000 0000 // addiu $9, $12, 0
0010 0101 0110 1011 0000 0000 0000 0001 // addiu $11, $11, 1
0000 1000 0000 0000 0000 0000 0000 0101 // j 5
```

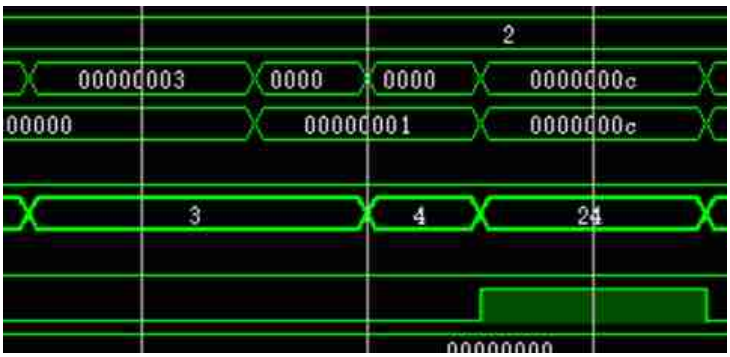
把上面的内容变成 16 进制，然后粘贴到 inst.data 文件，执行行为仿真：



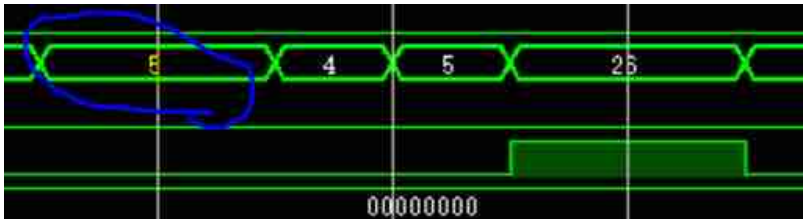
注意到斐波那契数列的下一项结果在 j 5 指令的前面第二条指令时写入$9 寄存器，在 j 5 指令往前数 4 条时计算并写入$12 寄存器。我们来看每次 ct_jump 为高的前两个周期时 alu_res 的输出：



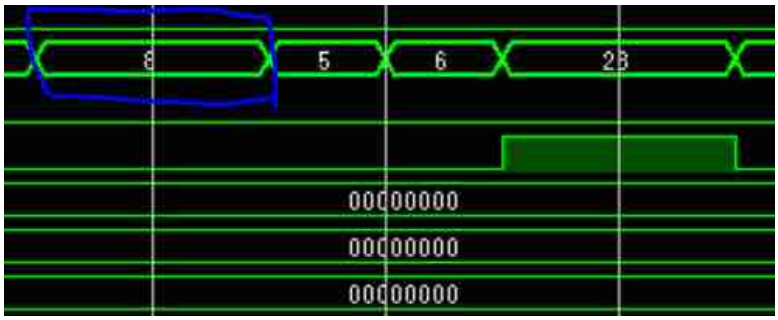
第一个是 2，没有问题。（1+1=2）



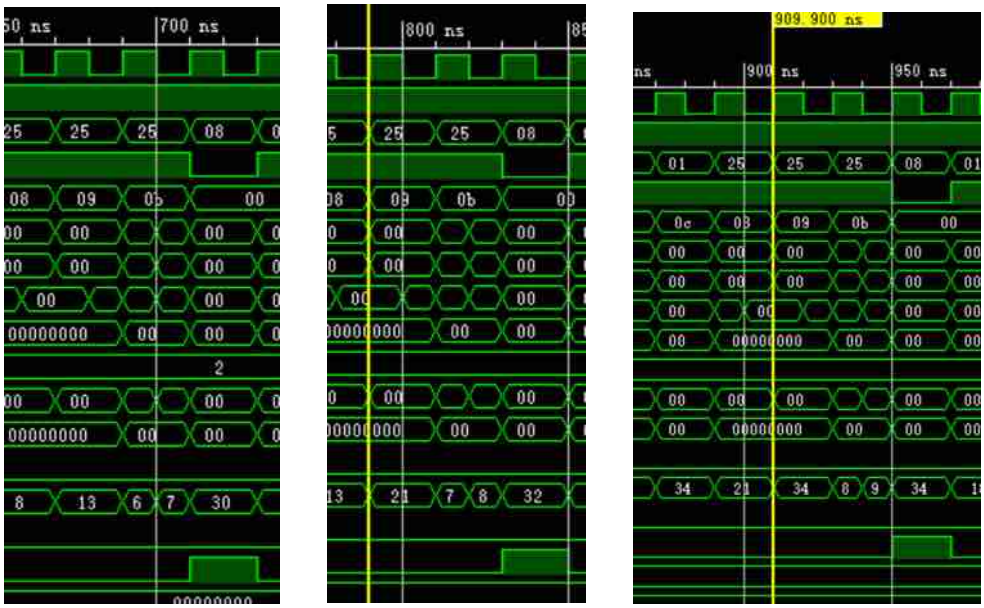
第二个是 3（1+2=3）



第三个 5（2+3=5）。4 是下边沿写入寄存器后导致输出变化，不用管。按照第一张图上的时钟，我们只要看左边第一条白线位置的值即可。



第四个，8



第五个，13；第六个，21。第七个，34。结果是正确的。

3.4 自主创新

http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html 这个页面给出了详细的 MIPS 指令集。从中自选指令（例如 jr、sll 等）扩展当前单周期 CPU 的功能，编写代码并仿真。要求给出扩展后的控制信号表格，仿真波形图，和对仿真波形的具体分析（参照 3.3.2 部分）。

[提示]

Jr 指令的功能是跳转到对应寄存器给出的地址。首先观察 jr 的指令结构：

```
0000 00ss sss0 0000 0000 0000 0000 1000
```

$s 是五位的寄存器编号。我们将它和 addu 指令对比：

```
0000 00ss ssst tttt dddd d000 0010 0001
```

可以看到与 addu 相比它的 st 是 0，而 MIPS 中 0 号寄存器的内容恰好始终为 0，那么我们可以有这样的思路：

1. 将这两个地址输入寄存器堆，在输出端口得到$s 和$0；
2. 原来的设计中，$s 和$t 直接接入 ALU，其中$t 是否传入 ALU 由一个控制信号控制。所以很自然地，我们在 ALU 的输出端口得到了$s 的值；
3. 原设计上 IFU 没有接入 ALU 的输出端口。这里我们就给 IFU 添加一个控制信号，当此信号为 1 时，指定下一跳地址为 ALU 的输出。（在 IFU 中增加一个多选器即可）

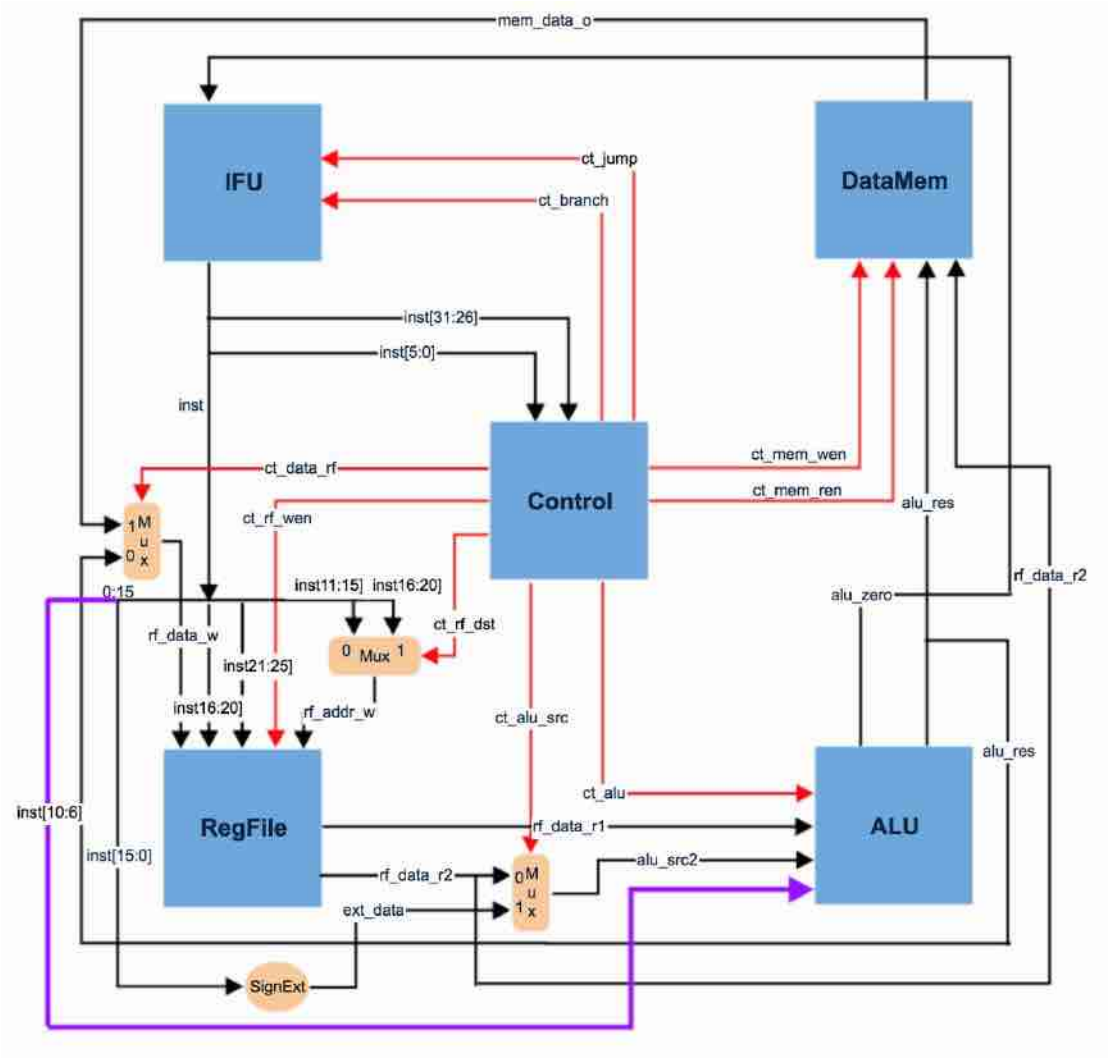
   显然这个信号由 funct 给出。当 op 为 0，funct=6’b001000 时，该信号就为 1。同时，让 ALU 接受$s 和$0 的内容，在 ALU 出口得到寄存器的值，IFU 就可以拿到下一条指令的地址。

再来看 sll 指令。它的功能是逻辑左移：

```
0000 00ss ssst tttt dddd dhhh hh00 0000
            sll $d, $t, h
```

将$t 的内容逻辑左移 h 位，存入$d。shamt 端存放了偏移量 h。显然该指令需要使用 ALU 进行计算，而将 ALU 的计算结果存入寄存器堆的功能已经由对应的控制信号做好。设计的方案是：

为 ALU 的控制信号 ct_alu 增加一个取值用来表示左移位（ct_alu 是 4bit 够用），然后把 shamt 直接接入 ALU。当指示 ALU 计算左移位时，忽略 rs 而直接用 rt（alu_src2）和 shamt 计算。修改之后的 ALU 具有 3 个输入：alu_src1、alu_src2 和 shamt，只有计算移位指令时才用到 shamt：



sll 指令还有个特别之处：如果把位移值改成 0，$s、$d 和$t 都写成 0，那么实际上这条指令没有做任何操作，也就是所谓的 NOP 指令（全 0）。

参考文献

[1] Patterson D A, Hennessy J L. 计算机组成与设计硬件/软件接口[M]. 机械工业出版社, 2007.

[2] MIPS Instruction Reference [EB/OL]. http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html, 1998–09–10

附录 1 指令说明

课内实验需要完成的指令参考说明：（31 条指令）

助记符	指令格式						示例	示例含义	操作及解释
BIT#	31..26	25..21	20..16	15..11	10..6	5..0			
R-型	**op**	**rs**	**rt**	**rd**	**shamt**	**func**			
add	000000	rs	rt	rd	00000	100000	add $1, $2, $3	$1 = $2 + $3	(rd) <-(rs) + (rt); rs = $2, rt = $3, rd = $1
addu	000000	rs	rt	rd	00000	100001	addu $1, $2, $3	$1 = $2 + $3	(rd) <- (rs) + (rt); rs = $2, rt = $3, rd = $1
sub	000000	rs	rt	rd	00000	100010	sub $1, $2, $3	$1 = $2 - $3	(rd) <- (rs) - (rt); rs = $2, rt = $3, rd = $1
subu	000000	rs	rt	rd	00000	100011	subu $1, $2, $3	$1 = $2 - $3	(rd) <- (rs) - (rt); rs = $2, rt = $3, rd = $1
and	000000	rs	rt	rd	00000	100100	and $1, $2, $3	$1 = $2 & $3	(rd) <- (rs) & (rt); rs = $2, rt = $3, rd = $1
or	000000	rs	rt	rd	00000	100101	or $1, $2, $3	$1 = $2 | $3	(rd) <- (rs) | (rt); rs = $2, rt = $3, rd = $1
xor	000000	rs	rt	rd	00000	100110	xor $1, $2, $3	$1 = $2 ^ $3	(rd) <- (rs) ^ (rt); rs = $2, rt = $3, rd = $1
nor	000000	rs	rt	rd	00000	100111	nor $1, $2, $3	$1 = ~($2 | $3)	(rd) <- ~((rs) | (rt)); rs = $2, rt = $3, rd = $1
slt	000000	rs	rt	rd	00000	101010	slt $1, $2, $3	If($2<$3) $1=1 else $1=0	If(rs<rt) rd=1 else rd=0; rs = $2, rt = $3, rd = $1
sltu	000000	rs	rt	rd	00000	101000	sltu $1, $2, $3	If($2<$3) $1=1 else $1=0	If(rs<rt) rd=1 else rd=0; rs = $2, rt = $3, rd = $1 （无符号数）
sll	000000	00000	rt	rd	shamt	000000	sll $1,$2,10	$1=$2<<10	(rd)<-(rt)<<shamt,rt=$2,rd=$1,shamt=10
srl	000000	00000	rt	rd	shamt	000010	srl $1,$2,10	$1=$2>>10	(rd)<-(rt)>>shamt,rt=$2,rd=$1,shamt=10 （逻辑右移）
sra	000000	00000	rt	rd	shamt	000011	sra $1,$2,10	$1=$2>>10	(rd)<-(rt)>>shamt,rt=$2,rd=$1,shamt=10 （算术右移，注意符号位保留）

sllv	000000	rs	rt	rd	00000	000100	sllv $1, $2, $3	$1=$2<<$3	(rd)<-(rt)<<(rs),rs=$3,rt=$2,rd=$1
srlv	000000	rs	rt	rd	00000	000110	srlv $1, $2, $3	$1=$2>>$3	(rd)<-(rt)>>(rs),rs=$3,rt=$2,rd=$1（逻辑右移）
srav	000000	rs	rt	rd	00000	000111	srav $1, $2, $3	$1=$2>>$3	(rd)<-(rt)>>(rs),rs=$3,rt=$2,rd=$1（算术右移，注意符号位保留）
jr	000000	rs	00000	00000	00000	001000	jr $31	goto $31	(PC)<-(rs)
I-型	**op**	**rs**	**rt**	**immediate**					
addi	001000	rs	rt	immediate			addi $1,$2,10	$1 = $2 + 10	(rt)<-(rs)+(sign-extend)immediate,rt=$1,rs=$2
addiu	001001	rs	rt	immediate			addiu $1,$2,10	$1 = $2 + 10	(rt)<-(rs)+(sign-extend)immediate,rt=$1,rs=$2
andi	001100	rs	rt	immediate			andi $1,$2,10	$1 = $2 & 10	(rt)<-(rs)&(zero-extend)immediate,rt=$1,rs=$2
ori	001101	rs	rt	immediate			ori $1, $2, 10	$1 = $2 | 10	(rt)<-(rs)|(zero-extend)immediate,rt=$1,rs=$2
xori	001110	rs	rt	immediate			xori $1, $2, 10	$1 = $2 ^ 10	(rt)<-(rs)^(zero-extend)immediate,rt=$1,rs=$2
lui	001111	00000	rt	immediate			lui $1, 10	$1 = 10*65536	(rt)<-immediate<<16&0FFFF0000H，将 16 位立即数放到目的寄存器高 16 位，目的寄存器的低 16 位填 0
lw	100011	rs	rt	offset			lw $1, 10($2)	$1 = Memory[$2+10]	(rt)<-Memory[(rs)+(sign_extend)offset], rt=$1,rs=$2
sw	101011	rs	rt	offset			sw $1, 10($2)	Memory[$2+10]=$1	Memory[(rs)+(sign_extend)offset]<-(rt), rt=$1,rs=$2
beq	000100	rs	rt	offset			beq $1, $2, 40	If($1=$2) gotoPC+4+40	If((rt)=(rs)) then (PC)<-(PC)+4+((sign_extend)offset<<2),rs=$1,rt=$2

bne	000101	rs	rt	offset	bne $1,$2,40	If($1≠$2) gotoPC+4+40	If((rt) ≠(rs)) then (PC)<-(PC)+4+((sign_extend)offset<<2),rs=$1,rt=$2
slti	001010	rs	rt	immediate	slti $1,$2,10	If($2<10)$1=1 else $1=0	If((rs)<(sign_extend)immediate) then (rt)<-1; else (rt)<-0,rs=$2,rt=$1;
sltiu	001011	rs	rt	immediate	sltiu $1,$2,10	If($2<10)$1=1 else $1=0	If((rs)<(zero_extend)immediate) then (rt)<-1; else (rt)<-0,rs=$2,rt=$1;
J-型	**op**	**address**					
J	000010	address			j 10000	goto 10000	(PC)<-((zero_extend)address<<2),address=10000/4
jal	000011	address			jal 10000	$31=PC+4 goto 10000	($31)<-(PC)+4;(PC)<-((zero_extend)address<<2),address=10000/4

注：标注绿色的指令为文中实现的指令。

附录 2 计算机组成原理课内实验（共 16 学时）

实验内容	学时数	实验要求
取指单元模块（IFU）设计	2	补充完整代码，并对模块进行仿真验证，在实验报告中进行仿真波形分析
寄存器堆模块（Regfile）设计	2	补充完整代码，并对模块进行仿真验证，在实验报告中进行仿真波形分析
算术逻辑单元模块（ALU）设计	2	补充完整代码，并对模块进行仿真验证，在实验报告中进行仿真波形分析

数据存储模块（DataMem）设计	2	补充完整代码，并对模块进行仿真验证，在实验报告中进行仿真波形分析
控制器模块（Control）设计	4	补充完整代码，并加入顶层模块，实现完整的工程。分析仿真波形，说明处理器功能仿真是否正确。
单周期 CPU 指令扩展与仿真	4	每人扩展两条指令，并进行仿真验证指令功能正确性。

附录 3 指令分组

去掉已经实现的 6 条指令还有 25 条指令可供扩展。将指令分为 32 组，每个同学实现一组（2 条指令为一组）。学生可先自愿选择难度较大的指令。其他由抽签决定。

指令组号	指令名称	指令组号	指令名称	指令组号	指令名称	指令组号	指令名称
1	jr, jal	9	nor, xori	17	subu,lui	25	slt, xori
2	jr, lui	10	slt, addi	18	and,bne	26	sltu, andi
3	sllv, lui	11	sltu, andi	19	jr, slti	27	sll, addi
4	srav, bne	12	sll, ori	20	slt, andi	28	srl, bne
5	bne, slti	13	srl, xori	21	sltu, ori	29	sra, or
6	slti, jal	14	sra, jr	22	jal, subu	30	sllv, xor
7	srlv, nor	15	add, jal	23	or,slti	31	addi, nor
8	srav, and	16	sub, jr	24	xor,sltiu	32	andi, subu