数字逻辑与处理器基础实验

流水线 MIPS 处理器实验报告

学号：2020010768

班级：无 05

姓名：付宇辉

时间：2022 年 7 月 16 日

目录

一、实验名称与内容

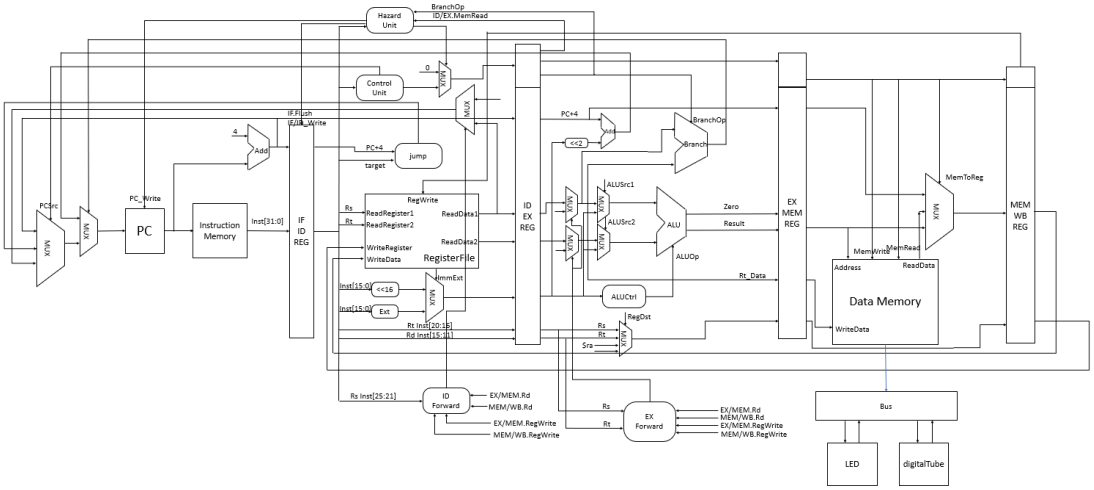
实验名称：流水线 MIPS 处理器的设计

实验内容：我选择的是实验内容 1，即，将春季学期实验四设计的单周期（多周期）MIPS 处理器改进为流水线结构，并利用此处理器完成字符串搜索算法。

二、设计方案

（一）总体设计

根据理论课的学习，使用 Verilog，设计了支持 MIPS32 核心指令的五级流水线。总体设计的数据通路如下，主体的设计结构与下图一致。



设计的 CPU 支持的指令如下：

1. R 型指令：add、addu、sub、subu、and、or、xor、nor、slt、sltu、sll、、srl、sra、jr、jalr；
2. I 型指令：lui、addi、addiu、andi、sltiu、lw、sw、beq、bne、blez、bgtz、bltz；

3. J 型指令：j、jal。

上述指令的具体格式参见附录。

CPU 为五级流水线设计，分为取指令（IF）、译指令（ID）、执行（EX）、访存（MEM）、写回（WB）五个阶段。采用了完全的 Forwarding 解决数据依赖冒险，模拟寄存器的先写后读；利用 stall 一个周期的方式解决 load-use 冒险；分支指令提前到 EX 阶段进行判断，J 型指令与 jr 和 jalr 在 ID 阶段进行跳转。

下面对各个方面介绍 CPU 的设计细节。

（二）IF 阶段

IF 阶段进行的是 PC 的更新和指令存储器的访问。

当时钟上升沿来的时候，PC 会根据控制信号进行更新。不进行跳转，更改为 PC+4；分支跳转，更新为 branch_target；直接跳转更新为 jump_target。PC 的写入信号收到 PCWrite 控制，通过控制写入信号可以进行 stall 周期，来起到解决竞争和冒险的问题。

（三）ID 阶段

ID 阶段进行的是对 IF 阶段取出指令的译码并进行访问寄存器堆取值，同时进行立即数的扩展，通过 Control 模块进行控制信号的生成，通过 Hazard 模块进行冒险的判断，进行跳转目标的生成，ID 阶段的转发。

ID 阶段利用 IF 和 ID 之间的寄存器，进行操作。对寄存器堆的读写，是通过指令译码选择，并且进行数据转发存入下一个级间寄存器。立即数扩展有三种方式，分别是左移 16 位、符号扩展、零扩展。最重要的部分是将指令的 OpCode 和 Funct 部分输入 Control 模块，生成控制信号，并且让控制信号在各个流水线级间进行传递，保证流水线各个部分工作能够正常进行。

（四）EX 阶段

EX 阶段进行的是 ALU 运算，同时生成分支的目标和 branch_hazard 信号。
我在这里直接秉持着“不重复造轮子的原则”，直接将数字逻辑与处理器基础理论课大作业部分的 ALU 和 ALUControl 模块直接复用。ALU 接收两个 32 位整型数据，支持位运算、加法、减法、小于比较功能。控制信号由 ALUControl 接收 ID_EX_Reg 存储的控制信号后产生。
Branch 接收两个 32 位整型数据，同样根据控制信号进行等于、不等于、小于等于零、小于零、大于零的判断，输出信号 branch_hazard，当其为高电平时进行分支跳转。同时将立即数左移两位，与 PC+4 求和，得到分支跳转的目标。
ALU 的输入时读取上一个级间寄存器与 EX 阶段的转发的结果选择得到的。

（五）MEM 阶段

MEM 阶段进行对数据存储器的访存操作，同时与外设 LED 和数码管进行交互。数据存储功能；0x40000000～0x7FFFFFFF（字节地址）为外设地址空间，对其地址的读写对应到相应的外设资源。
MEM 阶段根据 EX_MEM_Reg 中存储的 MemToReg 信号，从 PC、PC+4，Rt 内容、访存结果中选择写入寄存器堆的数据。

（六）WB 阶段

WB 阶段是对寄存器堆的写入操作。
通过写入信号的控制，我们将上个级间寄存器需要写入的数据，写入寄存器堆，完成一个指令整个执行的过程。
写入的数据也会进行数据的转发操作，为了避免数据冒险。

（七）设计细节

I. 分支与跳转

CPU 支持 j、jal、jr、jalr 以及多条分支指令，分别在 ID 阶段和 EX 阶段进行判断和执行。对于 j、jal、jr、jalr 四条指令，在 ID 阶段控制信号产生模块生成 jump_hazard 信号，传入冒险处理模块。冒险处理模块产生 IF_Flush 信号，将 IF_ID_Reg 的指令置为 32 位 0，从而清理掉下一个周期 ID 阶段将译码的信号。

对于分支指令，处理器默认不跳转，即 IF 和 ID 不暂停工作。在分支指令的 ID 阶段，控制信号产生模块根据分支指令的类型，产生 BranchOp 并存储到流水线寄存器中。在 EX 阶段，Branch 模块接收转发后的 rs_data 和 rt_data 作为输入，并进行相关判断。当判断为真时，branch_hazard 信号拉高，冒险模块随即产生 IF_Flush 和 ID_Flush 指令，将对应的 IF_ID_Reg 和 ID_EX_Reg 中的关键信号改为 0，即清理掉错误的指令。而当判断为假时，处理器继续运行，可以减少一定的 CPI。简言之，此 CPU 对分支的预测总为假。

II. 避免冒险

上面介绍了分支跳转如何解决，下面着重介绍如何避免数据冒险。

整体来说，避免数据冒险是通过转发解决的。根据理论课的学习，设计了四条转发路径，来解决数据冒险。

EX_MEM 转发到 ID：对于前前条指令执行写入寄存器的操作，不可能从寄存器中读取数据，而 jr 和 jalr 需要读取 Rs 寄存器的数据。所以将 EX 的结果从 EX_MEM_Reg 转发到 ID，从而不用添加 nop 或者 stall 便可以在 ID 阶段跳转。

MEM_WB 转发到 ID：实际写寄存器在 WB 阶段后的一个周期开始，我们将 MEM_WB_Reg 中的 write_data 转发到 ID，模拟了先写后读，保证 WB 阶段的同时，ID 可以读取到正确的数据。

EX_MEM 转发到 EX：对于前一条指令或者需要写入寄存器的指令，可以采取从

EX_MEM_Reg 转发到 EX 阶段的策略，将同一时刻正在 WB 写入数据用于 EX 的计算中。

MEM_WB 转发到 EX：对于前前条、执行访存或者需要写入寄存器的指令，可以从 MEM_WB_Reg 转发到 EX 阶段，将 WB 阶段写入的数据用于 EX 的计算中。

对于 Load-Use 冒险，采取阻塞一个周期的方式，然后可以通过上述转发操作，解决。

III. 外设处理

直接通过写入特定地址的数据存储器当中的数据来控制外设的工作状态。

我通过直接编写汇编代码，将外设的控制信号写入特定的存取器地址，来控制外设。

但是由于编写汇编代码的复杂性，目前只能将一位十六进制数显示在数码管上，这是需要改进的地方，希望后续能够找到更好的解决方案。

三、文件清单与关键代码

（一）文件清单

本次报告的目录结构如下：

```
├──assets 报告中使用的图片
|      critical_path.jpg
|      critical_path_graph.jpg
|      instruction_count.jpg
|      running_periods.jpg
|      schematic.jpg
|      timing_summary.jpg
|      utilization_hierarchy.jpg
|      utilization_summary.jpg
├──report 实验报告
```

```
    |      report.docx
    |      report.pdf
    ├──src  代码文件
    |   ├──constraints
    |   |      CPU.xdc 约束文件
    |   ├──simulation
    |   |      tb.v CPU testbench 文件
    |   └──source
    |          ALU.v ALU 计算模块
    |          ALUControl.v ALU 控制信号生成单元
    |          Branch.v 分支判断
    |          Control.v 控制信号生成单元
    |          CPU.v 顶层文件
    |          DataMem.v 数据存储器
    |          EX_Forward.v EX 阶段转发单元
    |          EX_MEM_Reg.v EX_MEM 级间寄存器
    |          Hazard.v 冒险控制单元
    |          ID_EX_Reg.v ID_EX 级间寄存器
    |          ID_Forward.v ID 阶段转发单元
    |          IF_ID_Reg.v IF_ID 级间寄存器
    |          InstMem.v 指令存储器
    |          MEM_WB_Reg.v MEM_WB 级间寄存器
    |          PC.v PC 寄存器
    |          RegisterFile.v 寄存器堆
    └──src
       |  test.dat 测试用字符串和模式串
       |  generate_InstMem.py 通过 MARS 十六进制指令文件生成 InstMem 文件
       ├──asm 编写的汇编文件
       |      bf.asm
       |      horspool.asm
       |      kmp.asm
       ├──inst_hex MARS 生成的十六进制指令
       |      bf.hex
       |      horspool.hex
       |      kmp.hex
       └──inst_mem 生成的指令存储器文件
              bf.txt
              horspool.txt
              kmp.txt
```

（二）关键代码

ID 阶段转发代码

```
assign rs_data_foward_id = (id_forward_1 == 2'b00)? rs_data:
                           (id_forward_1 == 2'b01)? ex_mem.ALUOut:
                           mem_wb.Write_data;
assign rt_data_foward_id = id_forward_2? mem_wb.Write_data: rt_data;
```

EX 阶段转发代码

```
wire [31:0] rs_data_foward_ex, rt_data_foward_ex;
assign rs_data_foward_ex = (ex_forward_1 == 2'b01)? ex_mem.ALUOut:
                           (ex_forward_1 == 2'b10)? mem_wb.Write_data:
                           id_ex.Rs_Data;
assign rt_data_foward_ex = (ex_forward_2 == 2'b01)? ex_mem.ALUOut:
                           (ex_forward_2 == 2'b10)? mem_wb.Write_data:
                           id_ex.Rt_Data;

wire [31:0] alu_src1, alu_src2;
assign alu_src1 = (id_ex.ALUSrc[1:0] == 2'b01)? id_ex.Imm:
                  (id_ex.ALUSrc[1:0] == 2'b10)? 32'h0:
                  rs_data_foward_ex;
assign alu_src2 = id_ex.ALUSrc[2]? id_ex.Imm: rt_data_foward_ex;
```

Hazard 控制代码

```
wire load_use_hazard;
assign load_use_hazard = reset? 1'b0: (ID_EX_MemRead && (ID_EX_Rt != 0)
&& ((ID_EX_Rt == IF_ID_Rs) || (ID_EX_Rt == IF_ID_Rt)));

assign PC_Wen = ~load_use_hazard;
assign IF_Wen = ~load_use_hazard;

assign IF_Flush = reset? 1'b0: (jump_hazard || branch_hazard);
assign ID_Flush = reset? 1'b0: (branch_hazard || load_use_hazard);
```

PC 更新代码

```
assign PC_plus4 = {PC_o[31], PC_o[30:0] + 31'd4};
assign PC_next = branch_hazard? branch_target:
                 (PCSrc == 2'b00)? PC_plus4:
                 (PCSrc == 2'b01)? jump_target:
```

```
                (PCSrc == 2'b10)? jr_target:
                32'h00400000;
```

四、综合与实现情况

（一）时序性能

本次的管脚约束中，将时钟周期设置为 10.00ns，观察综合后的时序裕量为 0.440ns

Design Timing Summary

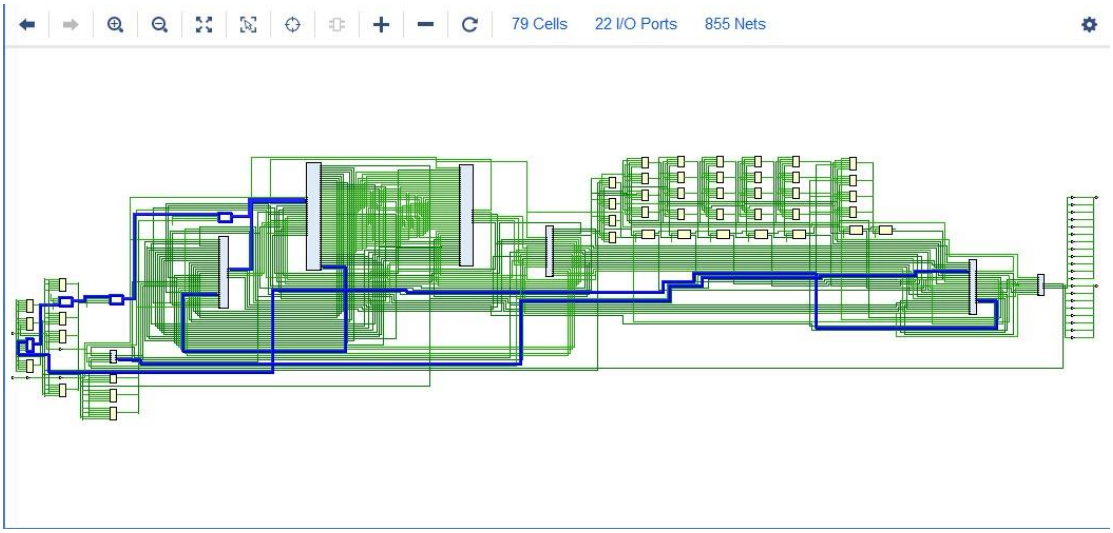
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.440 ns	Worst Hold Slack (WHS):	0.134 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	35289	Total Number of Endpoints:	35289	Total Number of Endpoints:	17799

All user specified timing constraints are met.

预计时钟周期最短应为 9.560ns，因此频率最高大致为

$$f = \frac{1}{9.560ns} = 104.6MHz$$

观察关键路径：



Path 1	0.440	18	19	63	mem_wb/RegWrite_reg/C	pc/PC_o_reg[29]/D	9.442	4.177	5.265	10.0	CLK
Path 2	0.459	18	19	63	mem_wb/RegWrite_reg/C	pc/PC_o_reg[31]/D	9.423	4.158	5.265	10.0	CLK
Path 3	0.532	18	19	63	mem_wb/RegWrite_reg/C	pc/PC_o_reg[30]/D	9.350	4.085	5.265	10.0	CLK
Path 4	0.553	18	19	63	mem_wb/RegWrite_reg/C	pc/PC_o_reg[28]/D	9.329	4.064	5.265	10.0	CLK
Path 5	0.554	17	18	63	mem_wb/RegWrite_reg/C	pc/PC_o_reg[25]/D	9.328	4.063	5.265	10.0	CLK
Path 6	0.573	17	18	63	mem_wb/RegWrite_reg/C	pc/PC_o_reg[27]/D	9.309	4.044	5.265	10.0	CLK

可以看到关键路径是包含数据寄存器 DataMem 和冒险检测单元的路径。因

此处理器的频率主要被数据存储器和冒险检测单元所限制。

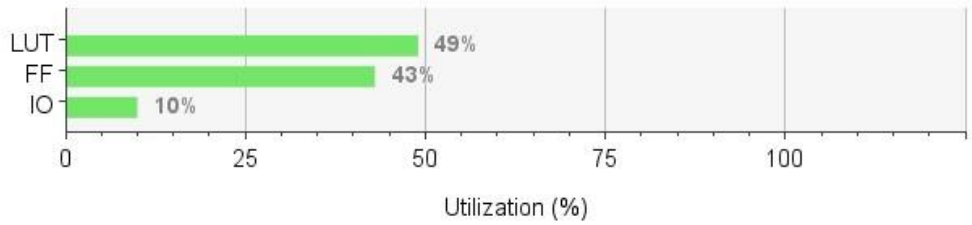
（二）逻辑资源占用情况

处理器的逻辑资源占用情况如下：

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Bonded IOB (210)	BUFGCTRL (32)
N CPU	10196	17798	4064	1856	22	1
data_mem (DataMem)	8567	16404	3808	1792	0	0
ex_mem (EX_MEM_Reg)	688	107	0	0	0	0
id_ex (ID_EX_Reg)	69	161	0	0	0	0
if_id (IF_ID_Reg)	11	64	0	0	0	0
mem_wb (MEM_WB_Reg)	0	38	0	0	0	0
pc (PC)	136	32	0	0	0	0
regs (RegisterFile)	715	992	256	64	0	0

Summary

Resource	Utilization	Available	Utilization %
LUT	10196	20800	49.02
FF	17798	41600	42.78
IO	22	210	10.48

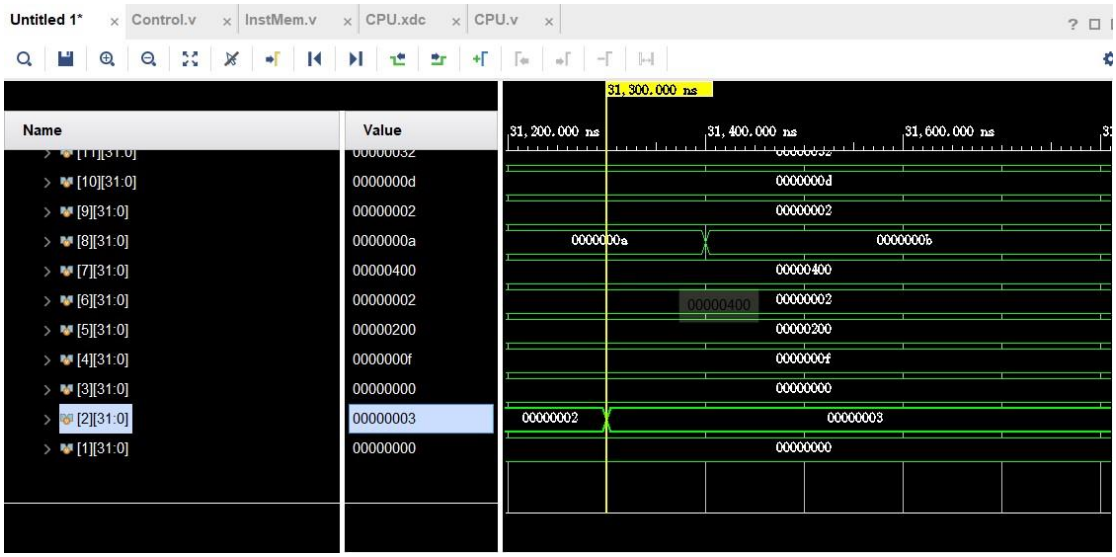


五、仿真验证

本次实验经过了三种方法匹配字符串，汇编程序见的 test 文件夹。

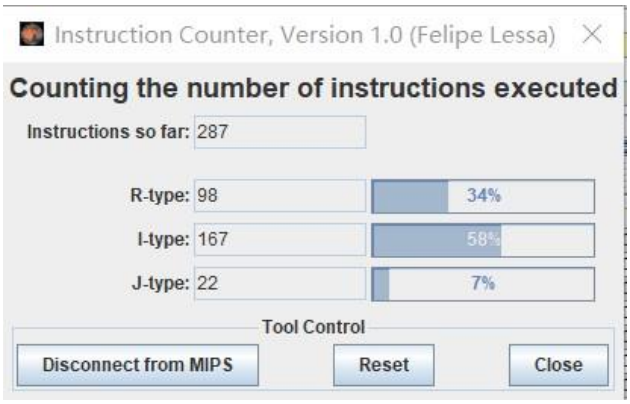
进行多次测试，答案均是正确的。

进行仿真结果如下：



可以看到，最终寄存器$v0 的值为 3。

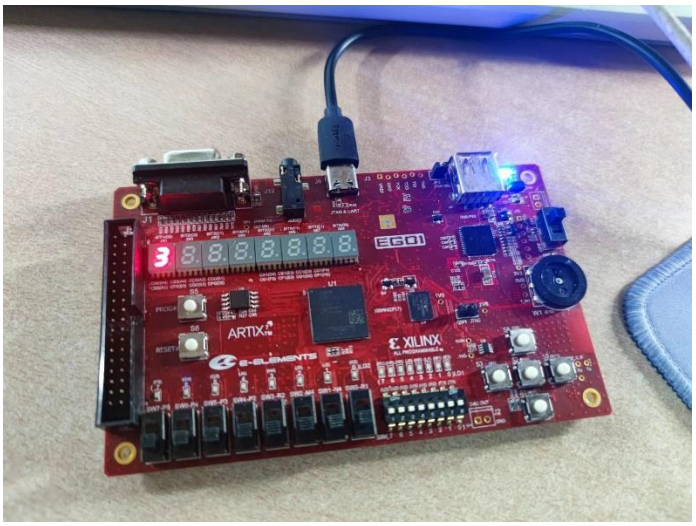
同时我们用 MARS 模拟器统计一个汇编文件当中的指令数，得到总共有 287 条指令。



同时我们通过 vivado 仿真的结果看出，经过了 312 个周期，CPU 计算得到了结果。所以此时用的 CPI 为 312/287=1.09，但是主频相对于单周期处理器有巨大的提升。

六、硬件调试情况

将设计代码使用 Vivado 2021.2 烧录到 FPGA 上之后运行，七段数码管上很快便出现结果“3”。通过按下复位键，也发现结果正确。



七、心得体会

这次实验是我到目前为止花时间最多的一次，同时也遇到了很多 bug 需要解决。但同时也有很多地方做了妥协，比如没有很好进行外设的实现，而是折中的利用汇编代码来有瑕疵的实现了数码管显示结果的要求（具体见上文描述），同时由于外设实现的不好，也没有完成 UART 的选做要求，这是我需要努力的地方。

同时，我基本上完全独立完成了本次实验。在编写代码的过程中，有一些在理论课上不懂的地方，通过实验我也有了更深的了解，我非常喜欢这种在实践中学习以及巩固知识的过程，也让我对课上所讲的流水线有了更深的理解。

在遇到的具体的 bug 中，我觉得最让我烦心的是对于冒险的处理，stall 和转发的控制花费了我大量的时间，最后将自己写的汇编程序正确地跑在 CPU 上的时候也有很大的成就感。

这次实验也有很多不足之处。首先，处理器的频率没有很高，没有进行更深入的优化；第二，存储器没有使用更好的 IP Core，因此在切换汇编程序等操作上耗费了一些时间；第三，一些细节上也编写的稍显繁琐，没有进行优化，等等。

总而言之，本次实验让我有很多收获。最后感谢老师在这一学期的倾情讲授，感谢助教的悉心指导，没有老师和助教的帮助，我也难以得到这么多的收获。

附录 支持的指令格式

指令格式表						
	OpCode [5:0]	Rs [4:0]	Rt [4:0]	Rd [4:0]	Shamt [4:0]	Funct [5:0]
	OpCode [5:0]	Rs [4:0]	Rt [4:0]	imm/offset [15:0]		
	OpCode [5:0]	target [25:0]				
R 型算术指令						
add	0	rs	rt	rd	0	0x20
addu	0	rs	rt	rd	1	0x21
sub	0	rs	rt	rd	2	0x22
subu	0	rs	rt	rd	3	0x23
and	0	rs	rt	rd	0	0x24
or	0	rs	rt	rd	1	0x25
xor	0	rs	rt	rd	2	0x26
nor	0	rs	rt	rd	3	0x27
slt	0	rs	rt	rd	0	0x2a
sltu	0	rs	rt	rd	0	0x2b
sll	0	0	rt	rd	shamt	0
srl	0	0	rt	rd	shamt	0x02
sra	0	0	rt	rd	shamt	0x03
I 型算术指令						
lui	0x0f	0	rt	imm		
addi	0x08	rs	rt	imm		
addiu	0x09	rs	rt	imm		
andi	0x0c	rs	rt	imm		
sltiu	0x0b	rs	rt	imm		
存取指令						
lw	0x23	rs	rt	offset		
sw	0x2b	rs	rt	offset		
分支指令						
beq	0x04	rs	rt	offset		
bne	0x05	rs	rt	offset		
blez	0x06	rs	0	offset		
bgtz	0x07	rs	0	offset		
bltz	0x01	rs	0	offset		
跳转指令						
j	0x02	target				
jal	0x03	target				
jr	0	rs	0			0x08
jalr	0	rs	0	rd	0	0x09