Logisim单周期CPU设计报告

一，CPU设计方案综述

（一）总体设计概述

本设计为由logisim实现的最多支持32条指令的单周期MIPS-CPU，支持addu，subu，ori，lw，sw，beq，lui，nop。为了实现这这些功能，本实验设计出的CPU被分为了IFU（指令存储单元），GRF（寄存器组），ALU（算术逻辑运算单元），DM（数据存储器），EXT（位扩展器），Controller（控制器），spliter（分解指令）这7个模块。

（二）关键模块定义

1.GRF

A 介绍

通用寄存器组，用于临时存放和调用数据，数据为32位。

B 端口定义

寄存器堆

端口	输入输出(位宽)	描述
RA1	I [4:0]	指定寄存器堆中的一个，将其中数据输出到RD1
RA2	I [4:0]	指定寄存器堆中的一个，将其中数据输出到RD2
WA	I [4:0]	指定32个寄存器中的一个，写入WD数据
WD	I [31:0]	输入的数据
reset	I	异步复位端口
clk	I	时钟信号
WE	I	WE为1：可写入；WE为0，不可写入
RD1	O [31:0]	RA1指定寄存器的值
RD2	O [31:0]	RA2指定寄存器的值

C 功能介绍

功能名称	功能描述
读入数据	当WE为1时，可以将WD写入WA指定的寄存器中
输出数据	可以将RA1和RA2指定的寄存器中的值通过RD1和RA2输出
复位	当reset信号为1时，实现异步复位

2.DM

A 介绍

数据存储器，可以用于存放和调用数据。

B 端口定义

端口	输入输出（位宽）	描述
WA	I [5:0]	待写入（输出）地址
WD	I [31:0]	待写入数据
reset	I	异步复位信号
clk	I	时钟信号
Store	I	写入使能端
Load	I	输出使能端
RD	O [31:0]	输出数据

C 功能定义

功能名称	功能描述
读入数据	当WE为1且时钟上升沿到来时，写入数据
输出数据	将WA指定的内存中存储的数据输出
复位	当reset信号为1时，实现异步复位

3.ALU

A 介绍

算数逻辑运算单元，用于实现指令执行过程中所需要的数学运算

B 端口定义

端口	输入输出（位宽）	描述
A	I [31:0]	参与运算的第一个数据
B	I [31:0]	参与运算的第二个数据
Op	I [2:0]	000:+ 001:- 010:|
C	O [31:0]	运算结果
zero	O	C是0为1，否则为0

C 功能定义

功能名称	功能描述
数学运算	将输入的A，B值按照Op的值进行二元运算，输出结果
判0	判断运算结果是否为0

4.IFU

A 介绍

指令存储单元，内部涵盖指令计数器（PC）和指令寄存器（IM），用于取出当前执行的指令和存放下一个指令。

B 端口定义

端口	输出输出（位宽）	描述
nextPC	I [31:0]	下一条指令地址
clk	I	时钟信号
reset	I	异步复位信号
intr	O [31:0]	从IM中取出的当前指令
PC	O [31:0]	当前指令地址

C 功能定义

功能名称	功能描述
取指令	将当前指令取出用于后续逻辑处理
存指令	将处理后的写一条指令地址存入nextPC
复位	当reset信号为1，PC回归32b'0

5.EXT

A 介绍

用于进行位扩展操作，主要用于将16位立即数扩展成32位

B 端口定义

端口	输入输出（位宽）	描述
in	I [16:0]	输入16位立即数
sign	I [31:0]	输出符号扩展后的数据
unsign	I [31:0]	输出无符号扩展后的数据
bigimm	I [31:0]	输出将立即数低位扩展后的数据

C 功能定义

功能名称	功能描述
无符号扩展	对立即数进行无符号扩展
有符号扩展	对立即数进行有符号扩展
低位扩展	将立即数加载到高16位，低位补充0

6.Controller

A 介绍

中心控制器，用于产生支持各条指令数据通路的片选和控制信号。

B 端口定义

端口	输入输出（位宽）	描述
op	I [5:0]	每条指令的六位操作码
function	I [5:0]	每条指令的六位功能码
beq	O	是否为beq指令
MemtoReg	O	是否从存储器向寄存器堆写入数据
MemWrite	O	是否向存储器中写入数据
MemRead	O	是否输出寄存器中的数据
RegWrite	O	是否向寄存器堆中写入数据
RegDst	O	是否选择rd作为寄存器写入地址（R）
ALUSrc	O	是否选择符号扩展立即数参与运算
ALUCtrl	O [2:0]	ALU运算模式选择信号
lui	O	是否为lui指令
ori	O	是否为ori指令

C 功能定义（真值表）

	nop	addu	subu	lw	sw	beq	lui	ori
op	000000	000000	000000	100011	101011	000100	001111	001101
function	000000	100001	100011	略	略	略	略	略
beq	0	0	0	0	0	1	0	0
MemtoReg	0	0	0	1	0	0	0	0
MemWrite	0	0	0	0	1	0	0	0
MemRead	0	0	0	1	0	0	0	0
RegWrite	0	1	1	1	0	0	1	1
RegDst	0	1	1	0	0	0	0	0
ALUSrc	0	0	0	1	1	0	0	0
ALUCtrl	000	000	001	000	000	001	000	010
lui	0	0	0	0	0	0	1	0
ori	0	0	0	0	0	0	0	1

7.spliter

A 介绍

用于将32位指令代码分解成易于处理的众多模块。

B 端口定义

端口	输入输出（位宽）	描述
intr	I [31:0]	输入指令
imm	O [15:0]	15位立即数
function	O [5:0]	6位功能码
rd	O [4:0]	寄存器堆输入地址
rt	O [4:0]	寄存器堆操作地址1
rs	O [4:0]	寄存器堆操作地址2
op	O [4:0]	6位操作码
j	O [25:0]	26位跳转立即数

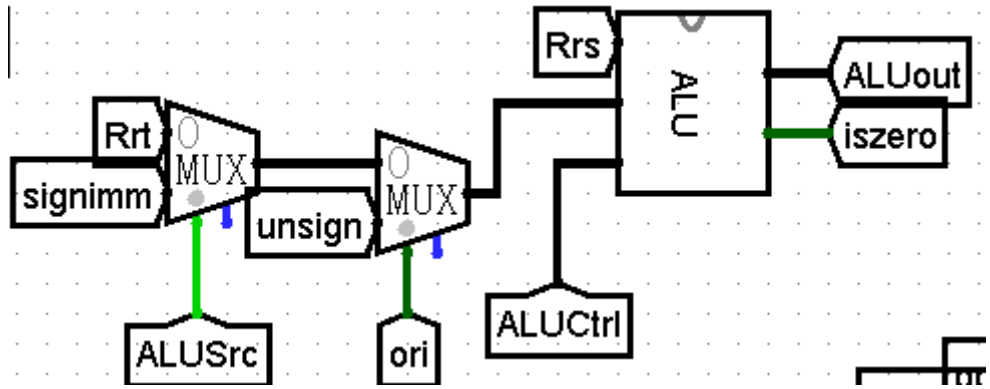
C 功能定义

功能名称	功能描述
分解指令	将32位指令代码分解成imm,function,rs,rt,rd,op和j

（三）重要机制及其实现方法

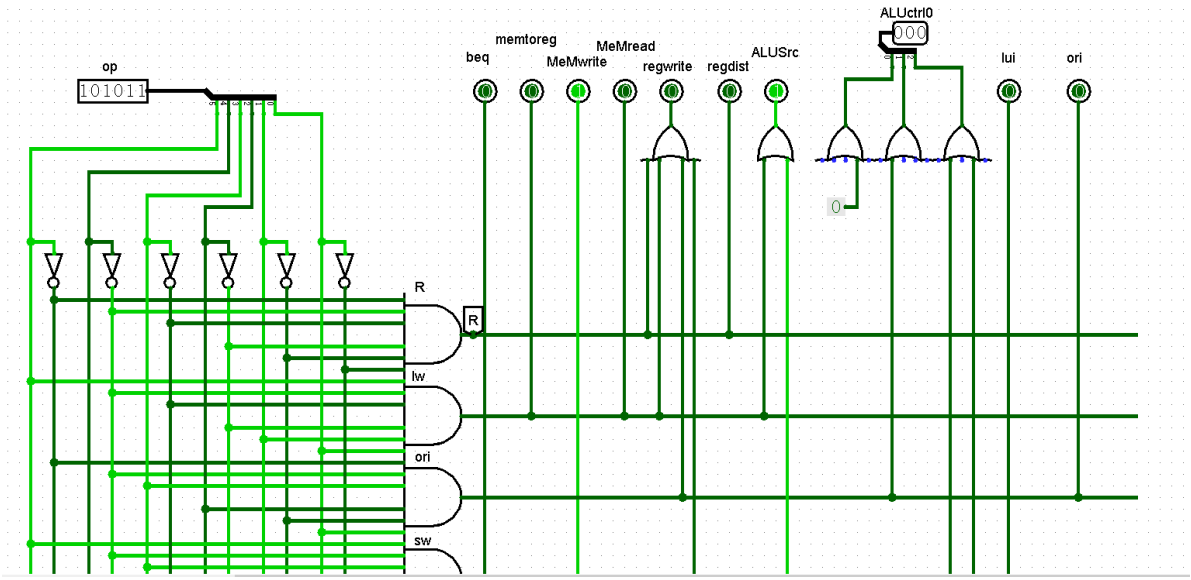
1 数据通路

首先列出数据通路表，然后进行合并，实现时利用多路选择器Mux加控制信号进行构建。如下图（ALU数据通路），三种输入通过Mux连接，同时ALUSrc和ori共同控制ALU的第二个参与运算的单元。



2 控制信号搭建

利用6个一位输入阵列加一个与门来得到操作指令类别，在运用或门阵列实现控制信号生成。如下图。



二，测试方案

（一）典型样例测试

1 测试ALU&GRF

```
.text
ori $1 $1 8094908
addu $1 $1 $1
addu $2 $2 $1
addu $3 $3 $1
addu $4 $4 $1
addu $5 $5 $1
addu $6 $6 $1
addu $7 $7 $1
addu $8 $8 $1
addu $9 $9 $1
addu $10 $10 $1
addu $11 $11 $1
addu $12 $12 $1
addu $13 $13 $1
addu $14 $14 $1
addu $15 $15 $1
addu $16 $16 $1
addu $17 $17 $1
addu $18 $18 $1
addu $19 $19 $1
addu $20 $20 $1
addu $21 $21 $1
addu $22 $22 $1
addu $23 $23 $1
addu $24 $24 $1
addu $25 $25 $1
addu $26 $26 $1
addu $27 $27 $1
addu $28 $28 $1
addu $29 $29 $1
addu $30 $30 $1
addu $31 $31 $1
```

之后

```
subu $2 $2 $1
subu $3 $3 $1
subu $4 $4 $1
subu $5 $5 $1
subu $6 $6 $1
subu $7 $7 $1
subu $8 $8 $1
subu $9 $9 $1
subu $10 $10 $1
subu $11 $11 $1
subu $12 $12 $1
subu $13 $13 $1
subu $14 $14 $1
subu $15 $15 $1
subu $16 $16 $1
subu $17 $17 $1
subu $18 $18 $1
subu $19 $19 $1
subu $20 $20 $1
subu $21 $21 $1
subu $22 $22 $1
subu $23 $23 $1
subu $24 $24 $1
subu $25 $25 $1
subu $26 $26 $1
subu $27 $27 $1
subu $28 $28 $1
```

2 测试sw&lw

```
.text
ori $1 $1 80949
sw $1 4($0)
lw $2 4($0)
sw $2 8($0)
lw $3 8($0)
sw $3 12($0)
lw $4 12($0)
sw $4 16($0)
lw $5 16($0)
sw $5 20($0)
lw $6 20($0)
sw $6 24($0)
lw $7 24($0)
sw $7 28($0)
lw $8 28($0)
sw $8 32($0)
lw $9 32($0)
sw $9 36($0)
lw $10 36($0)
sw $10 40($0)
lw $11 40($0)
sw $11 44($0)
lw $12 44($0)
sw $12 48($0)
lw $13 48($0)
sw $13 52($0)
lw $14 52($0)
sw $14 56($0)
lw $15 56($0)
```

3 测试beq

```
.text
start:
ori $1 $1 4
ori $3 $3 4
beq $1 $3 start
```

（二）自动化生成测试数据

例：c++编程生成lw，sw测试数据：

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    printf("ori $1 $1 8094908\n");
    for(int i=1;i<=15;i++)
    {
        printf("sw $%d %d($0)\n",i,i*4);
        printf("lw $%d %d($0)\n",i+1,i*4);
    }
}
```

三，思考题

（一）现在我们的模块中IM使用ROM， DM使用RAM， GRF使用Register，这种做法合理吗？ 请给出分析，若有改进意见也请一并给出。

合理，指令存储器是只读的，所以只能用只读存储器ROM。我们需要支持堆DM经行修改的指令，所以DM必须使用可修改存储器RAM。GRF是寄存器堆，使用由D触发器构成的寄存器，便于快速访存。

(二) 事实上，实现nop空指令，我们并不需要将它加入控制信号真值表，为什么，给出你的理由

nop空指令输入时，所有数据通路中的控制信号都为0，而默认的数通路为将rs寄存器的值和rt寄存器的值加起来存入rd寄存器，而对nop指令rs，rt，rd都为0，所以实际上在对0号寄存器做操作，而无论怎么对$0做操作，都不会改变该寄存器的值，对整体不产生影响，故等价于不做任何操作。

（三）上文提到，MARS不能导出PC与DM起始地址均为0的机器码。实际上，可以通过为DM增添片选信号，来避免手工修改的麻烦，请查阅相关资料进行了解，并阐释为了解决这个问题，你最终采用的方法。

不需要在DM上增加片选信号，在本实验中可以通过将计算出的地址的7到31位置零的方法来实现起始地址为0的访问。

（四）除了编写程序进行测试外，还有一种验证CPU设计正确性的办法——形式验证。 形式验证的含义是根据某个或某些形式规范或属性，使用数学的方法证明其正确性或非正确性。请搜索“形式验证（Formal Verification)”了解相关内容后，简要阐述相比于测试，形式验证的优劣之处。

优点：形式验证可以从数学的角度来对电路进行分析，对于一些难以用样例或样例缺乏的电路设计也可以准确的检验其正确性，因此，这种验证方法具有普适性。此外，由于形式验证对电路做了整体的逻辑分析，因此其得出的结果可以涵盖更多的情况，因而可信度会更高。

缺点：相较于编写一些程序或选取一些典型样例经行验证，形式验证的复杂度大大提升，并且逻辑过于复杂时，可能会出现错误分析，从而导致误判的情况出现。