计算机组成原理 P3 单周期 CPU 实验报告

一、CPU 设计方案综述

（一）总体设计概述

本CPU为Logisim实现的单周期MIPS - CPU，支持的指令集包含{addu, subu, and, or, sll(nop), sllv, slt, jr, j, jal, beq, ori, lui, addi, lw, sw, lh, sw, lb, sb}。为了实现这些功能，CPU 主要包含了 PC、NPC、IM、GRF、ALU、DM、EXT、Controller。

模块间的协同关系主要为：PC 存储当前指令地址，NPC 计算下一条指令地址、IM 存储所有指令、GRF 为 32 个通用寄存器、ALU 执行主要计算、DM 存储数据到内存中、EXT 对立即数进行位扩展、Controller 计算控制信号。除 Controller 之外的模块组成数据通路，Controller 则控制数据通路中数据的选择、计算等。

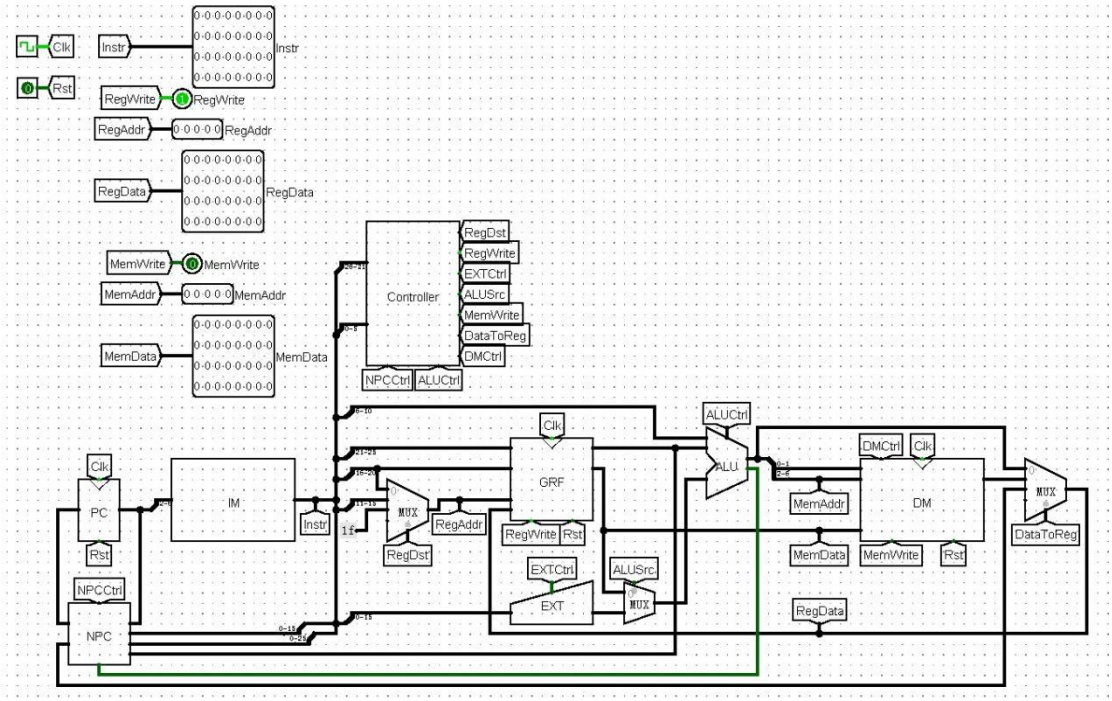

图 1 CPU 整体电路

（二）关键模块定义

1. PC（Program Counter）

端口定义：

表 1 PC 端口定义

序号	信号名	方向	位数	描述
1	NextPC	I	32	下一条指令的地址
2	Clk	I	1	时钟信号
3	Rst	I	1	异步复位信号
4	PC	O	32	当前指令地址

内部逻辑说明：

用一个 32 位寄存器存储当前指令地址，当时钟上升沿到来且使能信号为 1 时将新的指令地址写入寄存器。当异步复位信号有效时就将寄存器复位为起始地址 0x00000000。

2. NPC（Next PC）

端口定义：

表 2 NPC 端口定义

序号	信号名	方向	位数	描述
1	PC	I	32	当前指令地址
2	Offset	I	16	branch 跳转的偏移量
3	Index	I	26	j 或 jar 跳转地址的 2-27 位
4	Reg	I	32	jr 指令的跳转地址
5	Brflag	I	1	branch 跳转条件是否满足
6	Ctrl	I	3	选择下条指令的地址
7	PCAdd4	O	32	当前指令地址加 4
8	NextPC	O	32	下一条指令地址

内部逻辑说明：

分别计算出 PC+4、branch 跳转地址、j 或 jal 跳转地址、jr 跳转地址，然后根据 Brflag 和 Ctrl 信号选择要输出的下条指令地址 NextPC。PCAdd4 用于 jal 指令将 PC+4 的值存入$ra 寄存器。

3. IM（Instruction Memory）

端口定义：

表 3 IM 端口定义

序号	信号名	方向	位数	描述
1	A	I	5	所取指令的地址
2	RD	O	32	32 位指令

内部逻辑说明：

用 ROM 存储所有指令。根据地址取出相应指令

4. GRF（General Register File）

端口定义：

表 4 GRF 端口定义

序号	信号名	方向	位数	描述
1	Clk	I	1	时钟信号
2	Rst	I	1	异步复位信号，将寄存器中的值全部清零
3	WE	I	1	写使能信号
4	A1	I	5	5 位地址输入信号，指定 32 个寄存器中的一个，将其中存储的数据读出到 RD1
5	A2	I	5	5 位地址输入信号，指定 32 个寄存器中的一个，将其中存储的数据读出到 RD2
6	A3	I	5	5 位地址输入信号，指定 32 个寄存器中的一个作为写入的目标寄存器
7	WD	I	32	32 位数据输入信号
8	RD1	O	32	输出 A1 指定的寄存器中的 32 位数据
9	RD2	O	32	输出 A2 指定的寄存器中的 32 位数据

内部逻辑说明：

通过多路分解器进行输入数据 WD 和使能信号 WE 的分配，A3 为选择信号；通过多路选择器进行输出数据 RD1 和 RD2 的选择，A1 和 A2 分别为选择信号。

5. ALU（Arithmetic Logical Unit）

端口定义：

表 5 ALU 端口定义

序号	信号名	方向	位数	描述
1	SrcA	I	32	操作数 A
2	SrcB	I	32	操作数 B
3	Shamt	I	5	移位位数
4	Ctrl	I	4	控制信号，选择一种计算结果输出
5	Out	O	32	计算结果
6	Flag	O	1	输出逻辑运算的结果，用于 branch 类指令

内部逻辑说明：

支持加、减、与、或、异或、移位、比较等运算。分开进行每种计算，最后通过多路选择器和 Ctrl 信号来选择输出结果。

6. DM（Data Memory）

端口定义：

表 6 DM 端口定义

序号	信号名	方向	位数	描述
1	A	I	5	5 位地址输入信号，将数据写入该地址
2	ByteSel	I	2	用于对半字和字节进行读写
3	Ctrl	I	2	控制信号，选择对字、半字还是字节操作
4	WE	I	1	写使能信号
5	Clk	I	1	时钟信号
6	Rst	I	1	异步复位信号，将内存清零
7	WD	I	32	32 位数据输入信号
8	RD	O	32	输出地址 A 存储的数据

内部逻辑说明：

用 RAM 存储数据，对地址 A 中的数据进行读写。以半字、字节为单位进行存入的时候，要注意和同一字地址中其他字节进行拼接，读取的时候注意进行位扩展。

7. EXT（Extender）

端口定义：

表 7 EXT 端口定义

序号	信号名	方向	位数	描述
1	In16	I	16	16 位立即数输入信号
2	Ctrl	I	1	控制信号，选择符号扩展还是零扩展
3	Out32	I	32	32 位数据输出信号

内部逻辑说明：

将 16 位立即数进行零扩展或符号扩展成 32 位后输出，使用 Bit Extender 实现。

8.Controller

端口定义：

表 8 Controller 端口定义

序号	信号名	方向	位数	描述
1	Op	I	5	指令中的 Op 字段
2	Funct	I	2	R 型指令中的 funct 字段
3	RegDst	O	2	GRF 的写入地址选择信号
4	RegWrite	O	1	GRF 写使能信号
5	EXTCtrl	O	1	EXT 控制信号
6	ALUSrc	O	1	ALU 操作数选择信号
7	MemWrite	O	1	DM 写使能信号
8	DataToReg	O	2	GRF 的写入数据选择信号
9	DMCtrl	O	2	DM 控制信号
10	NPCCtrl	O	3	NPC 控制信号
11	ALUCtrl	O	4	ALU 控制信号

内部逻辑说明：

通过与逻辑来实现指令的识别，再通过或逻辑实现从指令到控制信号的映射。为了简化识别电路，可以先将 R 指令识别出来，之后对 R 指令再单独进行识别。

ins	and	or	sllv	slt	jr	addi	jal	sh	sb	lh	lb
op	000000	000000	000000	000000	000000	001000	000011	101001	101000	100001	100000
funct	100100	100101	000100	101010	001000	-	-	-	-	-	-
NPCCtrl[2:0]	000	000	000	000	011	000	010	000	000	000	000
RegDst[1:0]	01	01	01	01	-	00	10	-	-	0	0
RegWrite	1	1	1	1	1(rd=0)	1	1	0	0	1	1
EXTCrtl	-	-	-	-	-	1	-	1	1	1	1
ALUScr	0	0	0	0	-	1	-	1	1	1	1
ALUCtrl[3:0]	0000	0001	1000	1001	-	0010	-	0010	0010	0010	0010
MemWrite	0	0	0	0	0	0	0	1	1	0	0
DataToReg[1:0]	00	00	00	00	-	00	10	-	-	01	01
DMCtrl[1:0]	-	-	-	-	-	-	-	01	10	01	10

ins	addu	subu	ori	lw	sw	beq	lui	sll(nop)	j
op	000000	000000	001101	100011	101011	000100	001111	000000	000010
funct	100001	100011	-	-	-	-	-	000000	-
NPCCtrl[2:0]	000	000	000	000	000	001	000	000	010
RegDst[1:0]	01	01	00	00	-	-	00	01	-
RegWrite	1	1	1	1	0	0	1	1	0
EXTCrtl	-	-	0	1	1	-	0	-	-
ALUScr	0	0	1	1	1	-	1	0	-
ALUCtrl[3:0]	0010	0011	0001	0010	0010	0010	0110	0111	-
MemWrite	0	0	0	0	1	0	0	0	0
DataToReg[1:0]	00	00	00	01	-	-	00	00	-
DMCtrl[1:0]	-	-	-	00	sll00	-	-	-	-

图 2 指令与控制信号对照表

（三）重要机制实现方法

1. 跳转

NPC 模块和 ALU 模块协同工作支持指令 beq 的跳转机制。

NPC 模块内置了判定单元和计算单元来独立支持指令 j、jal、jr 的跳转机制。

2. 半字、字节存取

通过两位的控制信号 DMCtrl 来判断是对字、半字还是字节进行操作。由于 RAM 是以字编址，在对半字、字节读入时，需要将无关的数据与要写入的数据进行拼接，再整体写入 RAM。读取时则取出整个字，选出某个位置的半字或字节进行扩展后再输出。

3. 控制信号

通过与逻辑来识别指令，对于 R 型指令，先通过 Op 识别出该类型是否为 R 型指令，再通过 Funct 具体识别是哪一条 R 型指令。通过或逻辑产生控制信号，当一条指令需要的控制信号为 1 时，将其连到或门上。对于多位的控制信号，分别对每一位用一个或门控制。

二、测试方案

（一）典型测试样例

1. 计算指令测试

```
ori $a0, $0, 0
ori $a1, $0, 1
ori $a2, $0, 65535
ori $a3, $a0, 123
lui $a0, 65535
lui $a1, 0
lui $a2, 2
lui $a3, 456
addu $t0, $a0, $a1
subu $t1, $a1, $a2
sll $t2, $a3, 2
sllv $t3, $a3, $a1
slt $t4, $a0, $a1
```

2. 跳转指令测试

```
ori $t0, $0, 0
ori $s0, $0, 5
for_begin:
beq $t0, $s0, for_end
    jal func
```

```
addi $t0, $t0, 1
j for_begin
for_end:
beq $0, $0, end
func:
    jr $ra
end:
```

3. 存取指令测试

```
ori $a0, 1
ori $a1, 2
ori $a2, 3
ori $a3, 4
sw $a0, 0($0)
sw $a1, 0($a3)
sh $a2, 8($0)
sh $a3, 10($0)
sb $a0, 12($0)
sb $a1, 13($0)
sb $a2, 14($0)
sb $a3, 15($0)
lw $t0, 0($0)
lw $t1, 0($a3)
lh $t2, 8($0)
lh $t3, 10($0)
lb $t4, 12($0)
lb $t5, 13($0)
lb $t8, 14($0)
lb $t7, 15($0)
```

（二）自动测试工具

1. 测试样例生成器

代码来源于讨论区

```
void print(int x)
{
    if(a[x].type==0) cout<<"nop"<<endl;
    if(a[x].type==1) cout<<"addu $"<<a[x].r1<<",$"<<a[x].r2<<",$"<<a[x].r3<<endl;
    if(a[x].type==2) cout<<"subu $"<<a[x].r1<<",$"<<a[x].r2<<",$"<<a[x].r3<<endl;
    if(a[x].type==3) cout<<"lui $"<<a[x].r1<<","<<a[x].r3<<endl;
    if(a[x].type==4) cout<<"ori $"<<a[x].r1<<",$"<<a[x].r2<<","<<a[x].r3<<endl;
    if(a[x].type==5) cout<<"lw $"<<a[x].r1<<","<<a[x].r3<<"($"<<a[x].r2<<")"<<endl;
    if(a[x].type==6) cout<<"sw $"<<a[x].r1<<","<<a[x].r3<<"($"<<a[x].r2<<")"<<endl;
    if(a[x].type==7) cout<<"beq $"<<a[x].r1<<",$"<<a[x].r2<<",branch"<<a[x].r3<<endl;
}
int main()
{
    //0nop 1addu 2subu 3lui 4ori 5lw 6sw 7beq
    int i,j;
    srand(time(0));
    size_im=128,size_dm=128,size_op=8,small_reg=8;
    get_small();
    for(i=1; i<=small_reg; i++)
        a[i]=(op(4,small[i],0,val[small[i]]=r(0,size_dm-1,0)));
    for(i=small_reg+1; i<=size_im; i++)
    {
        int op0=r(0,size_op-1,0),r1,r2,r3;
        if(op0==0) a[i]=(op(0,0,0,0));
        if(op0==1||op0==2) //addu与subu
        {
            a[i]=op(op0,nosmall(),r(0,27,1),r(0,27,1));
        }
        if(op0==3||op0==4) //lui与ori
        {
            a[i]=(op(op0,nosmall(),r(0,27,1),r(0,65535,0)));
        }
        if(op0==5) //lw
        {
            r1=r(0,size_dm-1,0)*4;
            r2=r(1,small_reg,0);
```

图 3 测试样例生成器部分代码

```
ori $26,$0,99
ori $14,$0,121
ori $9,$0,127
ori $17,$0,85
ori $23,$0,40
ori $13,$0,42
ori $20,$0,94
ori $7,$0,73
lw $18,94($13)
ori $5,$22,32164
ori $0,$17,64233
subu $8,$18,$18
lw $25,97($26)
nop
lui $3,3810
lw $15,45($9)
sw $11,227($7)
beq $19,$5,branch1
```

图 4 生成的测试样例

2. 自动执行脚本

通过 python 和命令行进行自动化测试，可以实现 mips 代码自动导出、IM 指令导入、logisim 自动运行并将记录的结果格式化，以便和 mips 执行的结果进行比对。

```
dataPath = testPath + "in1.txt"
logPath = testPath + "log1.txt"
ansPath = testPath + "out1.txt"

# export hex code

os.system("java -jar " + marsPath + " " + mipsPath + " nc mc CompactTextAtZero a dump .text HexText " + dataPath)

# load code to IM, generate test circuit

testData = open(dataPath).read()
circ = open(myCircPath).read()
circ = re.sub(r"addr/data: 5 32([\s\S]*)</a>", "addr/data: 5 32\n" + testData + "</a>", circ)
with open(testCircPath, "w") as testCirc:
    testCirc.write(circ)

# export logisim result

os.system("java -jar " + lgsmPath + " " + testCircPath + " -tty table >" + logPath)

# formalize output

with open(ansPath, "w") as ans, open(logPath, "r") as log:
    logText = log.readlines()
    for line in logText:
        line = re.sub(r"[\s]+", "", line)
        ans.write("@" + "{:>08x}".format(int(line[0:32], 2)) + ":\t")
        if (line[32] == '1' and line[33:38] != "00000"):
            ans.write("$" + "{:2d}".format(int(line[33:38], 2)) + " <= " + "{:>08x}".format(int(line[38:70],2)) + "\t")
        if (line[70] == '1'):
            ans.write("#" + "{:>08x}".format(4 * int(line[71:76], 2)) + " <= " + "{:>08x}".format(int(line[76:108], 2)) + "\t")
        ans.write("\n")
```

图 5 自动执行脚本部分代码

```
@34040002:      $ 4 <= 00000002
@34050001:      $ 5 <= 00000001
@00853004:      $ 6 <= 00000004
@34073014:      $ 7 <= 00003014
@34080000:      $ 8 <= 00000000
@0106102a:      $ 2 <= 00000000
@10400002:
@00000000:
```

图 6 格式化后的输出结果

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。

目前来看是合理的。因为一段指令在执行过程中不应该发生改变，所以指令存储器可用只读的 ROM；而数据存储器需要同时实现读写功能，并且只根据地址进行读写，所以适合用 RAM；对于 GRF，需要实现一个端口的写入和两个端口的读取，并且要求 0 寄存器的值始终为 0，所以用 32 个独立的寄存器来实现较好。

但在实际的系统中，指令存储器和数据存储器相互独立是不现实的。大多数计算机有一块单独的大容量内存来存储指令和数据，并且支持读和写操作。

（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？请给出你的理由。

空指令 nop 实际上是指令 sll $0, $0, 0，机器码的 32 位全为 0。如果 CPU 实现了 sll 指令，那么 nop 会将 0 寄存器中的值左移 0 位后写入 0 寄存器，不会造成任何影响。

（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。

logisim 中的 RAM 地址最高支持 24 位，当访问的地址超过 24 位的时候，可以使用多片 RAM，低 24 位作为 RAM 的地址，高位连接到片选信号用于选择哪一片 RAM。

但在我实际的设计中，因为 mips 导出的数据是从 0x00000000 开始存放的而

指令是从 0x00003000 开始存放的，两者起始地址的 2-7 位均是 0，而由于电路中的 IM 和 DM 地址均只有 5 位，不会造成影响，所以暂未特殊处理这个问题。如果之后需要扩大地址范围，对于 PC 来说，可以将地址减去 0x00003000；对于 DM 来说，可以利用片选信号。

（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。

优势：可以对指定描述的所有可能的情况进行验证，覆盖率达到了百分之百；利用数学上的方法将待验证电路和功能描述或参考设计直接进行比较，不需要使用仿真测试平台和激励；验证时间短，可以很快发现和改正电路设计中的错误，可以缩短设计周期。

劣势：对一套系统建立数学模型是有很高的时间成本的，当前产品迭代很快，有了 bug 也可以很快修复，如果对每个版本都去进行形式验证会很费时间。其次，形式验证目前不能有效验证电路的性能，如电路的时延和功耗。