CPU设计文档

> 2023年《计算机组成》（实验）P3 Logisim单周期CPU设计文档
>
> *by 匡亦煊 22371092*

概述

- 处理器为 32 位单周期处理器，应支持的指令集为：`add, sub, ori, lw, sw, beq, lui, nop`，其中：
  - `nop` 为空指令，机器码 `0x00000000`，不进行任何有效行为（修改寄存器等）。
  - `add, sub` 按无符号加减法处理（不考虑溢出）。
- 需要采用**模块化**和**层次化**设计。顶层有效的驱动信号要求包括且仅包括**异步复位信号 reset**（clk 请使用内置时钟模块）。

RTL描述分析

开始时，面对仅8条指令，我就因为被陷入反复的细节而感到困扰。

于是，我将指令大致分为3类，并归纳这些指令的RTL描述：

1. 运算类

指令	RTL描述的操作过程
`add`	$\mathrm{R[rd]} \leftarrow \mathrm{R[rs]} + \mathrm{R[rt]}$
`sub`	$\mathrm{R[rd]} \leftarrow \mathrm{R[rs]} - \mathrm{R[rt]}$
`ori`	$\mathrm{R[rt]} \leftarrow \mathrm{R[rs]}\ \mathrm{OR}\ \mathrm{zero\_extend(imm16)}$
`lui`	$\mathrm{R[rt]} \leftarrow \text{高位}\mathrm{\_extend(imm16)}$

2. 访存类

指令	RTL描述的操作过程
`lw`	$\mathrm{Addr} \leftarrow \mathrm{R[rs]} + \mathrm{sign\_extend(imm16)}$ $\mathrm{R[rt]} \leftarrow \mathrm{Mem[Addr]}$
`sw`	$\mathrm{Addr} \leftarrow \mathrm{R[rs]} + \mathrm{sign\_extend(imm16)}$ $\mathrm{Mem[Addr]} \leftarrow \mathrm{R[rt]}$

3. 跳转类

指令	RTL描述的操作过程
`beq`	$\mathrm{PC} \leftarrow (\mathrm{R[rs]} == \mathrm{R[rt]})\,?\,\mathrm{PC} + 4 + \mathrm{sign\_extend(imm16} << 2) : \mathrm{PC} + 4$

将RTL描述放在一起提取 **共性** 之后，可以发现，excute的部分无非就是：ALU运算、写入到寄存器、写入到DM。而决定每条指令 **个性** 的就是一些控制信号。

控制信号分析

仔细辨别各指令共性下的个性后，设计控制信号若干如下：

- NPC模块的 `NPCOp` 控制信号，控制：
  - 简单的PC+4
  - 有条件跳转`beq`、`blt`……

  （是待以下扩展的）
  - 无条件跳转`j`、`jal`（与`j`相比，需要回写$\mathrm{R}[31]=\mathrm{PC}+4$）
  - 跳转到寄存器`jr`
  - ……
- EXT模块的 `EXTOp` 控制信号，控制：
  - 00：zero extend
  - 01：sign extend
  - 10：加载到高位（`lui`）
  - 11：……
- 写入寄存器的**来源数据**控制信号 `RegSrc`，控制：
  - 00：ALU的res
  - 01：DM的MemRD
  - 10：EXT的extend(imm)
  - 11：……
- 写入寄存器的**目的地址**控制信号 `RegDst`，控制：
  - 00：rt
  - 01：rd
  - 10：……
  - 11：……
- 写入寄存器的使能信号 `RegWrite`
  - 0
  - 1
- ALU模块的B端信号的来源控制信号 `ALUSrc`，控制：
  - 0：GRF的RegRD2
  - 1：EXT的extend(imm)
- ALU模块的运算符控制信号 `ALUOp`
  - 00：加
  - 01：减
  - 10：或
  - 11：……
- 写入DM模块的使能信号 `MemWrite`

> 基本上每个控制信号，我都多留1-2个空位，方便课上扩展。

关键模块构造

1. IFU

端口定义

信号名称	方向	描述
NPC[31:0]	I	下一个PC值
rst	I	异步复位信号
clk	I	时钟信号
PC[31:0]	O	当前PC值
Instr[31:0]	O	当前PC指向的32位指令

功能定义

序号	功能	描述
1	复位	当异步复位有效时，将当前PC值复位为0x00003000
2	取指	当时钟上升沿到来时，更新PC值，并取出PC值指向的指令

2. NPC

端口定义

信号名称	方向	描述
PC[31:0]	I	当前PC值
NPCOp[1:0]	I	控制PC更新的方式，具体见功能定义 00：顺序更新 01：beq更新 10：..... 11：.....
Extend_Imm[31:0]	I	EXT模块输出端的立即数
Imm26[25:0]	I	j型跳转指令的地址，来自指令的25:0位
$reg[31:0]	I	GRF模块读出的某个寄存器的值
eq?	I	ALU模块输出端的 `R[rs]==R[rt]` 判断结果 0：不相等 1：相等
NPC[31:0]	O	下一个PC值
PC+4[31:0]	O	当前PC值+4

功能定义

序号	功能	描述
1	顺序更新	PC ← PC + 4
2	beq更新	PC ← (R[rs] == R[rt]) ? PC + 4 +sign_extend(imm<<2)
3	……	……
4	……	……

3. GRF

端口定义

信号名称	方向	描述
A1[4:0]	I	5位读寄存器地址（编号）
A2[4:0]	I	5位读寄存器地址（编号）
A3[4:0]	I	5位写寄存器地址（编号）
WD[31:0]	I	32位写入数据
WE	I	写使能信号
rst	I	异步复位信号
clk	I	时钟信号
RegRD1[31:0]	O	A1寄存器读出值
RegRD2[31:0]	O	A2寄存器读出值

功能定义

序号	功能	描述
1	复位	当异步复位有效时，将所有寄存器的值复位为0
2	读数据	从RegRD1、RegRD2端口读出A1、A2寄存器的值
3	写数据	当 **时钟上升沿到来**，**且使能信号WE有效**时，将WD写入到A3寄存器

4. ALU

端口定义

信号名称	方向	描述
A[31:0]	I	参与运算的第一个数
B[31:0]	I	参与运算的第二个数

信号名称	方向	描述
ALUOp[1:0]	I	控制运算的方法，具体见功能定义 00：算术加法 01：算术减法 10：逻辑或运算 11：……
eq?	O	输出 (A == B) ? 1 : 0
res[31:0]	O	运算结果

功能定义

序号	功能	描述
1	算术加法	res = A + B
2	算术减法	res = A - B
3	逻辑或运算	res = A OR B
4	……	……

5. EXT

端口定义

信号名称	方向	描述
Imm[15:0]	I	待扩展的16位立即数，来自I型指令的15:0位
EXTOp[1:0]	I	控制立即数的扩展方式，具体见功能定义： 00：无符号扩展 01：符合扩展 10：加载到高位 11：……
Extend_Imm[31:0]	O	扩展后的32位立即数

功能定义

序号	功能	描述
1	无符号扩展（zero extend）	高位补0
2	符合扩展（sign extend）	符合位为0，高位补0 符号位为1，高位补1
3	加载到高位（lui）	Imm加载到高16位，低16位补0
4	……	……

6. DM

端口定义

信号名称	方向	描述
MemAddr[31:0]	I	待写入数据/读出数据在内存中的地址
MemData[31:0]	I	待写入内存的数据
MemWrite	I	写使能信号
rst	I	异步复位信号
clk	I	时钟信号
MemRD[31:0]	O	读出的数据

功能定义

序号	功能	描述
1	异步复位	当异步复位信号有效时，将内存中的数据全部都复位为0
2	读数据	从MemRD端口读出内存中地址为MemAddr的数据
3	写数据	当 **时钟上升沿到来**，**且使能信号MemWrite有效**时，将MemData写入到内存中地址为MemAddr的位置
4	……	……

7. Ctrl

端口定义

信号名称	方向	描述
op[5:0]	I	所有指令的操作码，来自指令的31:26位
funct[5:0]	I	R型指令的辅助识别码，来自R型指令的5:0为
RegDst[1:0]	O	控制哪个寄存器写入数据： 00：Rt 01：Rd 10：…… 11：……

信号名称	方向	描述
RegSrc[1:0]	O	控制寄存器写入哪里的数据： 00：ALU的运算结果（如`add`、`sub`、`ori`） 01：DM的读出数据（如`lw`） 10：EXT扩展后的立即数（如`lui`） 11：……
RegWrite	O	控制GRF的写使能端何时有效： 0：GRF的写使能端无效 1：GRF的写使能端有效
EXTOp[1:0]	O	控制16位立即数采取何种方式扩展： 00：无符号扩展 01：符合扩展 10：加载到高位 11：……
ALUSrc	O	控制ALU的B端数据来自哪里： 0：GRF的RD2端读出数据 1：EXT扩展后的立即数
ALUOp[1:0]	O	控制ALU采取何种算术/逻辑运算： 00：算术加法 01：算术减法 10：逻辑或 11：……
NPCOp[1:0]	O	控制PC采取何种方式更新： 00：顺序更新 01：beq

控制信号真值表

		R	**add**	**sub**	**ori**	**lui**	**beq**	**lw**	**sw**
	op	000000			001101	001111	000100	100011	101011
	funct		100000	100010					
RegDst[1]			0	0	0	0		0	
RegDst[0]			1	1	0	0		0	
RegSrc[1]			0	0	0	1		0	
RegSrc[0]			0	0	0	0		1	
RegWrite			1	1	1	1		1	
EXTOp[1]					0	1	0	0	0
EXTOp[0]					0	0	1	1	1
ALUSrc			0	0	1		**0**	1	1
ALUOp[1]			0	0	1				
ALUOp[0]			0	1	0				
NPCOp[1]							0		

		R	add	sub	ori	lui	beq	lw	sw
NPCOp[0]							1		

> 可以发现，前5行要考虑的都是同一些指令，即需要写入到寄存器的那些指令

> `beq` 虽然有立即数，是I型指令，但是ALU运算的两个数据都来自寄存器（Rs和Rt），对应ALUSrc=0

> 控制信号基本都预留1-2个空位，方便课上扩展

测试

1. 运算类指令测试

测试 `ori` 指令和GRF阵列。

通过Python程序生成指令

```python
for i in range(32):
    print("ori ${},$zero,0x{:0>2d}{:0>2d}".format(i,i,i))
```

指令：

```
ori $0,$zero,0x0000
ori $1,$zero,0x0101
ori $2,$zero,0x0202
ori $3,$zero,0x0303
ori $4,$zero,0x0404
ori $5,$zero,0x0505
ori $6,$zero,0x0606
ori $7,$zero,0x0707
ori $8,$zero,0x0808
ori $9,$zero,0x0909
ori $10,$zero,0x1010
ori $11,$zero,0x1111
ori $12,$zero,0x1212
ori $13,$zero,0x1313
ori $14,$zero,0x1414
ori $15,$zero,0x1515
ori $16,$zero,0x1616
ori $17,$zero,0x1717
ori $18,$zero,0x1818
ori $19,$zero,0x1919
ori $20,$zero,0x2020
ori $21,$zero,0x2121
ori $22,$zero,0x2222
ori $23,$zero,0x2323
ori $24,$zero,0x2424
ori $25,$zero,0x2525
ori $26,$zero,0x2626
ori $27,$zero,0x2727
ori $28,$zero,0x2828
ori $29,$zero,0x2929
```

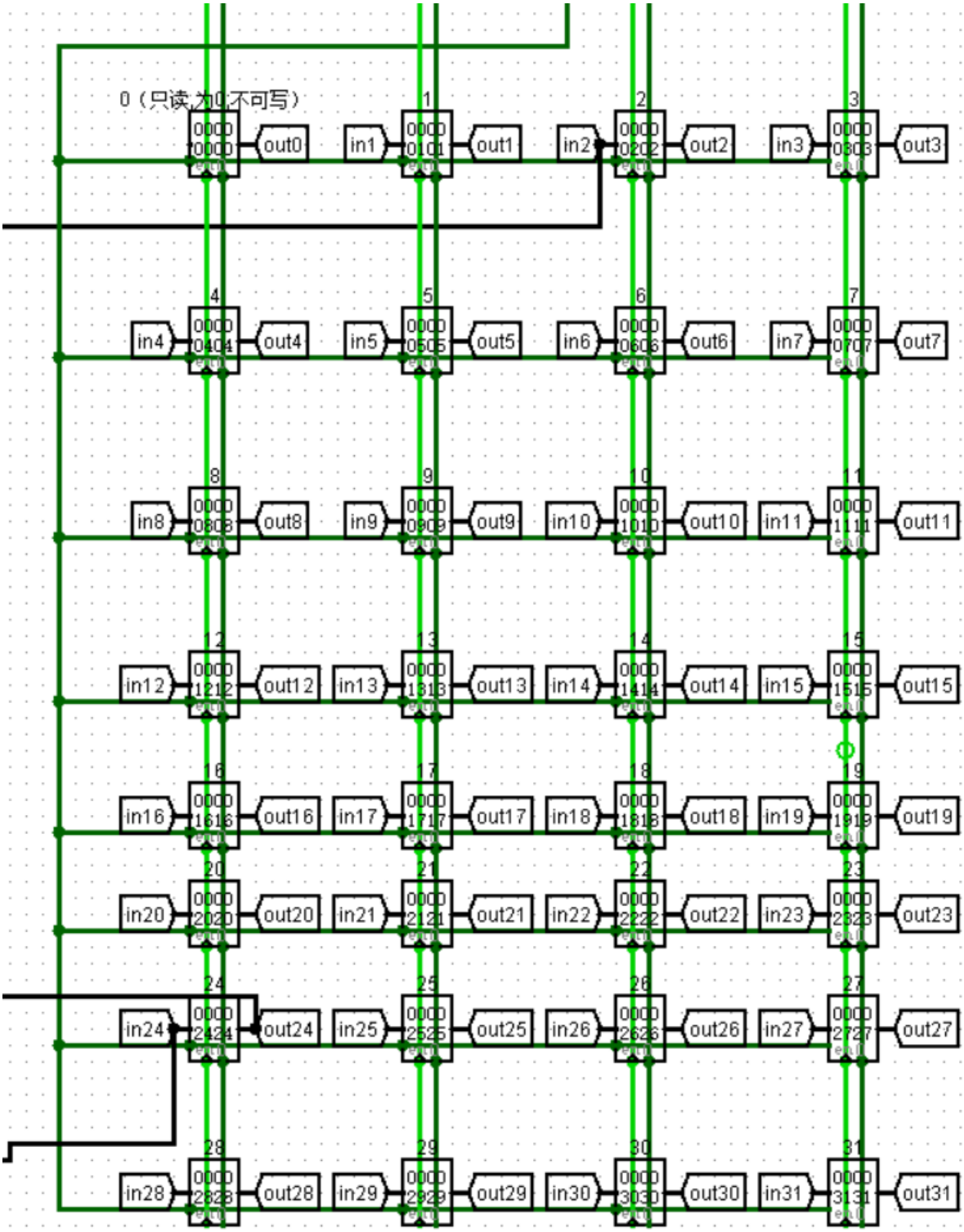
```
31 ori $30,$zero,0x3030
32 ori $31,$zero,0x3131
```

MARS结果：

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000101
$v0	2	0x00000202
$v1	3	0x00000303
$a0	4	0x00000404
$a1	5	0x00000505
$a2	6	0x00000606
$a3	7	0x00000707
$t0	8	0x00000808
$t1	9	0x00000909
$t2	10	0x00001010
$t3	11	0x00001111
$t4	12	0x00001212
$t5	13	0x00001313
$t6	14	0x00001414
$t7	15	0x00001515
$s0	16	0x00001616
$s1	17	0x00001717
$s2	18	0x00001818
$s3	19	0x00001919
$s4	20	0x00002020
$s5	21	0x00002121
$s6	22	0x00002222
$s7	23	0x00002323
$t8	24	0x00002424
$t9	25	0x00002525
$k0	26	0x00002626
$k1	27	0x00002727
$gp	28	0x00002828
$sp	29	0x00002929
$fp	30	0x00003030
$ra	31	0x00003131
pc		0x00003080
hi		0x00000000
lo		0x00000000

CPU结果：



测试 `add` 和 `sub`

```python
print("ori $1,$0,1")
for i in range(2,19):
    print("add ${},${},${}".format(i,i-1,i-1))
for i in range(19,32):
    print("sub ${},${},${}".format(i,i-1,i-2))
```

指令:

```
ori $1,$0,1
add $2,$1,$1
add $3,$2,$2
add $4,$3,$3
```

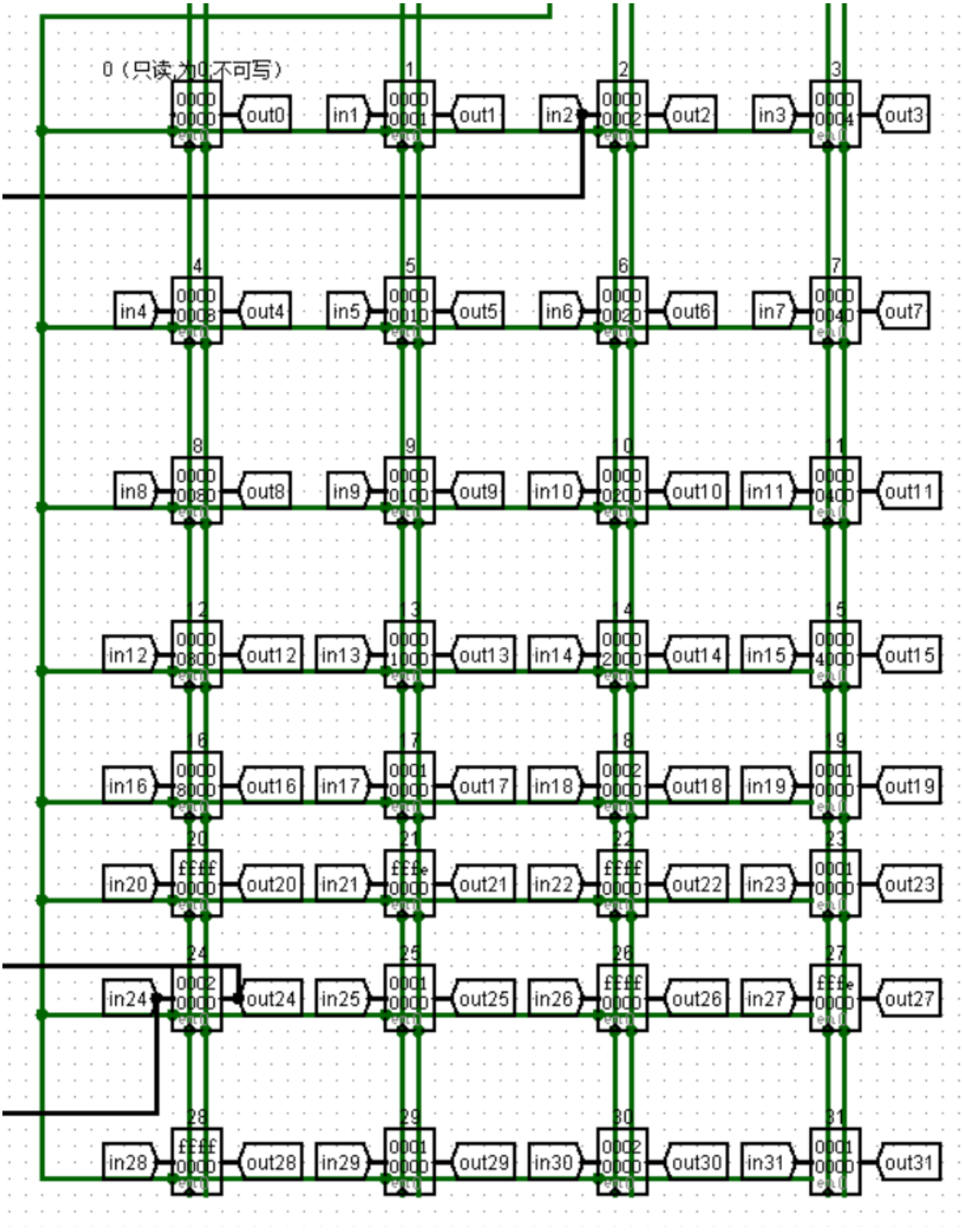
```
add $5,$4,$4
add $6,$5,$5
add $7,$6,$6
add $8,$7,$7
add $9,$8,$8
add $10,$9,$9
add $11,$10,$10
add $12,$11,$11
add $13,$12,$12
add $14,$13,$13
add $15,$14,$14
add $16,$15,$15
add $17,$16,$16
add $18,$17,$17
sub $19,$18,$17
sub $20,$19,$18
sub $21,$20,$19
sub $22,$21,$20
sub $23,$22,$21
sub $24,$23,$22
sub $25,$24,$23
sub $26,$25,$24
sub $27,$26,$25
sub $28,$27,$26
sub $29,$28,$27
sub $30,$29,$28
sub $31,$30,$29
```

MARS结果：

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000001
$v0	2	0x00000002
$v1	3	0x00000004
$a0	4	0x00000008
$a1	5	0x00000010
$a2	6	0x00000020
$a3	7	0x00000040
$t0	8	0x00000080
$t1	9	0x00000100
$t2	10	0x00000200
$t3	11	0x00000400
$t4	12	0x00000800
$t5	13	0x00001000
$t6	14	0x00002000
$t7	15	0x00004000
$s0	16	0x00008000
$s1	17	0x00010000
$s2	18	0x00020000
$s3	19	0x00010000
$s4	20	0xffff0000
$s5	21	0xfffe0000
$s6	22	0xffff0000
$s7	23	0x00010000
$t8	24	0x00020000
$t9	25	0x00010000
$k0	26	0xffff0000
$k1	27	0xfffe0000
$gp	28	0xffff0000
$sp	29	0x00010000
$fp	30	0x00020000
$ra	31	0x00010000
pc		0x0000307c
hi		0x00000000
lo		0x00000000

CPU结果：



2. 访存类指令测试

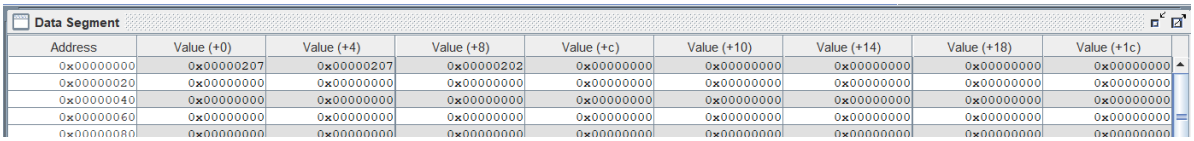
指令：

```
ori $0,$zero,0x0000
ori $1,$zero,0x0001
ori $2,$zero,0x0202
ori $3,$zero,0x0005



sw $2,3($1)
lw $4,3($1)
sw $4,7($1)
```

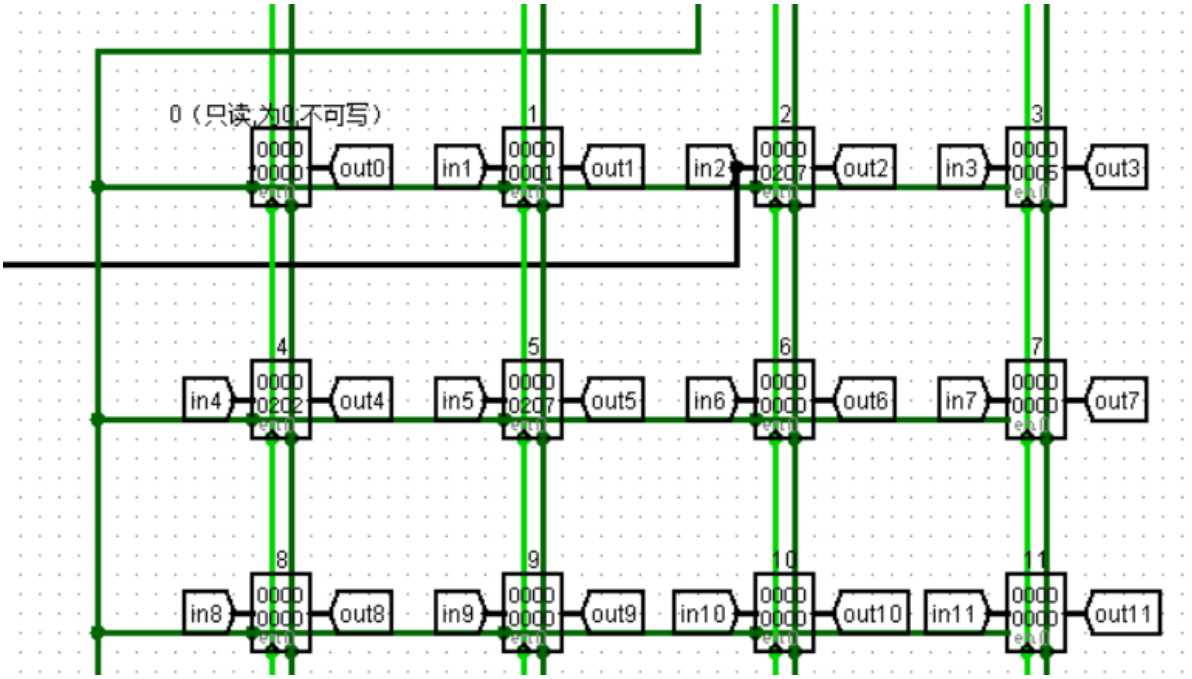
```
11
12  add $2,$2,$3
13
14  sw $2,-1($3)
15  sw $2,-5($3)
16  lw $5,-1($3)
```

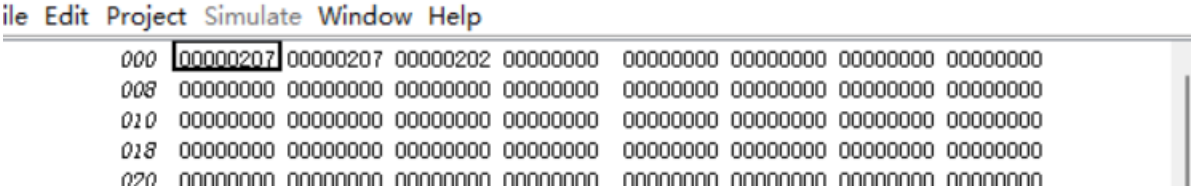
MARS结果：

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000001
$v0	2	0x00000207
$v1	3	0x00000005
$a0	4	0x00000202
$a1	5	0x00000207
$a2	6	0x00000000
$a3	7	0x00000000



CPU结果：





3. 跳转类指令测试

指令：

```
ori $1,$0,0x1234
add $2,$1,$0

ori $9,$9,0x1230
sub $10,$1,$9
add $10,$10,$10
add $10,$10,$10

lui $1,0x1234

ori $31,$0,0xffff

sw $31,0($10)

sub $31,$31,$1



beq $1,$2,yes1
    sw $31,-4($10)
    sw $31,-8($10)
    sw $31,-12($10)
yes1:
    sw $31,-16($10)

lui $2,0x1234
beq $1,$2,yes2
    sw $31,4($10)
    sw $31,8($10)
    sw $31,12($10)
    sw $31,16($10)
yes2:
    sw $31,24($10)

ori $20,$0,0
ori $21,$0,1
back:
    add $21,$21,$21
    sub $2,$2,$21


    beq $1,$2,back
```

MARS结果：

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x12340000
$v0	2	0x1233fffe
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00001230
$t2	10	0x00000010
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000002
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x00001800
$sp	29	0x00002ffc
$fp	30	0x00000000
$ra	31	0xedccffff
pc		0x0000306c
hi		0x00000000
lo		0x00000000

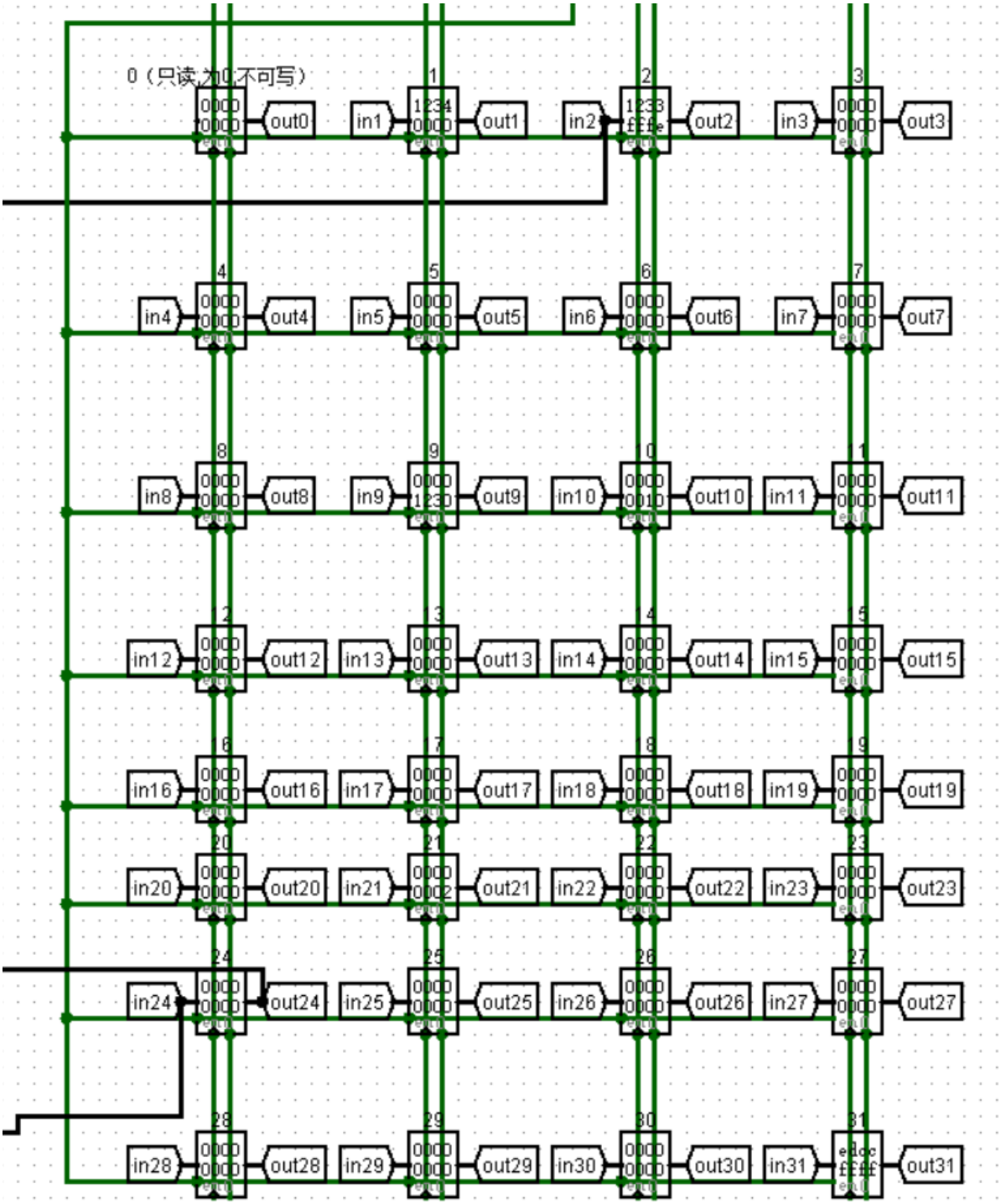
Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0xedccffff	0xedccffff	0xedccffff	0xedccffff	0x0000ffff	0x00000000	0x00000000	0x00000000
0x00000020	0x00000000	0x00000000	0xedccffff	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000000a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000000c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000000e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

CPU结果：

File Edit Project Simulate Window Help

000	edccffff	edccffff	edccffff	edccffff	0000ffff	00000000	00000000	00000000
008	00000000	00000000	edccffff	00000000	00000000	00000000	00000000	00000000
010	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
018	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
020	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
028	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
030	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
038	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
040	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
048	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
050	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
058	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
060	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
068	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
070	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000



4. 压力测试

通过下面的Python程序生成仅一千条指令，对 *除`beq`以外 (~~因为太菜了，不会写~~)* 的指令进行压力测试：

```
import random

def generate():

    op = random.randint(1,9)



    def generate_add():
        print("add ${},${},${}".format(random.randint(1,31),random.randint(1,31),random.randint(1,31)))
    def generate_sub():
        print("sub ${},${},${}".format(random.randint(1, 31), random.randint(1, 31), random.randint(1, 31)))

    def generate_lui():
        hex = ['0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f']
        imm = random.choice(hex) + random.choice(hex) + random.choice(hex) +random.choice(hex)
        print("lui ${},0x{}".format(random.randint(1,31),imm))

    def generate_lw():
        offset = ['0','4','8','12','16','20','24','28',
                  '32','36','40','44','48','52','56','60']
        print("lw ${},{}($0)".format(random.randint(1,31),random.choice(offset)))

    def generate_lw2():
        offset = ['0', '4', '-8', '-12', '-16', '-20', '-24', '-28',
                  '32', '-36', '-40', '-44', '-48', '-52', '-56', '-60']
        print("ori $1,$0,0x003c")
        print("lw ${},{}($1)".format(random.randint(1, 31), random.choice(offset)))

    def generate_sw():
        offset = ['0','4','8','12','16','20','24','28',
                  '32','36','40','44','48','52','56','60']
        print("sw ${},{}($0)".format(random.randint(1,31),random.choice(offset)))

    def generate_sw2():
        offset = ['0', '4', '-8', '-12', '-16', '-20', '-24', '-28',
                  '32', '-36', '-40', '-44', '-48', '-52', '-56', '-60']
        print("ori $1,$0,0x003c")
        print("sw ${},{}($1)".format(random.randint(1, 31), random.choice(offset)))
    if op == 1:
        generate_add()
```

```
    elif op == 2:
        generate_sub()
    elif op==3:
        generate_lui()
    elif op==4:
        generate_lw()
    elif op==5:
        generate_lw2()
    elif op==6:
        generate_sw()
    else:
        generate_sw2()

for i in range(700):
    generate()
```

由于数据随机性强，MARS容易出现异常中断，我将MARS中的add（sub）都替换为addu（subu），再与CPU的结果进行对比。

结果初步吻合。

思考题

1. **上面我们介绍了通过 FSM 理解单周期 CPU 的基本方法。请大家指出单周期 CPU 所用到的模块中，哪些发挥状态存储功能，哪些发挥状态转移功能。**
   - 状态存储：IM、GRF、DM
   - 状态转移：NPC，ALU
2. **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。
   - 没有改进意见。
3. **在上述提示的模块之外，你是否在实际实现时设计了其他的模块？如果是的话，请给出介绍和设计的思路。**
   - 我另外设计了一个 `NPC` 模块。
   - 功能介绍见《关键模块构造》部分。
   - 设计思路：

     ①从Ctrl引出ALUOp

     ②从ALU引出 `R[rs]==R[rd]` 的结果；

     引入到NPC判断是非跳转类还是beq跳转。

     同时考虑到课上扩展，我将可能出现的PC更新方式作大致分类，罗列如下：

     - **非跳转类**：$\mathrm{PC} \leftarrow \mathrm{PC}+4$
     - **b 跳转类**：
       $\mathrm{PC} \leftarrow (\mathrm{R}[\mathrm{rs}] == \mathrm{R}[\mathrm{rt}])\ ?\mathrm{PC}+4+\mathrm{sign\_extend}(\mathrm{Imm}||0^2) : \mathrm{PC}+4$
       - `beq`
       - `beqz`
       - `blt`
       - ……
     - **j 跳转类**：$\mathrm{PC} \leftarrow \mathrm{PC}_{31..28}||\mathrm{Imm}_{25..0}||0^2$

        - `j`
        - `jal`：还需要回写R[31]←PC+4
    - **jr跳转类**：$\mathrm{PC} \leftarrow \mathrm{R}[\mathrm{rs}]$
        - `jr`
        - `jalr`

4. **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？**
    - nop各为全为0，因此产生控制信号也全为0
        - NOPOp为0，不会跳转
        - RegWrite，不会改变寄存器堆
        - MemWrite，不会改变内存
    - 综上，nop不会对CPU造成影响
5. **阅读 Pre 的 “MIPS 指令集及汇编语言” 一节中给出的测试样例，评价其强度（可从各个指令的覆盖情况，单一指令各种行为的覆盖情况等方面分析），并指出具体的不足之处。**
    - 寄存器覆盖较少
    - `beq` 测试时，没有测试offset为负数的情况，测试不够充分
    - ……