计算机组成原理实验报告

一、 CPU 设计方案综述

(一) 总体设计概述

此 CPU 为通过 Logisim 电路模拟工具实现的单周期 CPU，支持 MIPS 指令集中的{beq, gez, bgtz, blez, bltz, bne, addiu, andi, lb, lbu, lh, lhu, lui, lw, ori, sb, sh, slti, sltiu, sw, xori, j, jal, addu, and, jalr, jr, nor, or, sll, sllv, slt, sltu, sra, srav, srl, srlv, subu, xor, nop}指令。为了实现这些指令，CPU 主要包含了 Controller（控制器）、IFU（取指令单元）、GRF（通用寄存器组，也称为寄存器文件、寄存器堆）、ALU（算术逻辑单元）、DM（数据存储器）等基本部件，通过 Extender（位扩展器）、Multiplexer（多路选择器）、Splitter（分线器）等内置器件组合连接成数据通路。

实现此 CPU 时，将其分为“机制”（mechanism）与“策略”（policy）两大部分，即将数据通路相关模块与核心控制模块分开考虑，并采用模块化和层次化设计方法。顶层有效的驱动信号要求包括且仅包括：异步复位信号 reset。顶层模块如图 1 所示。

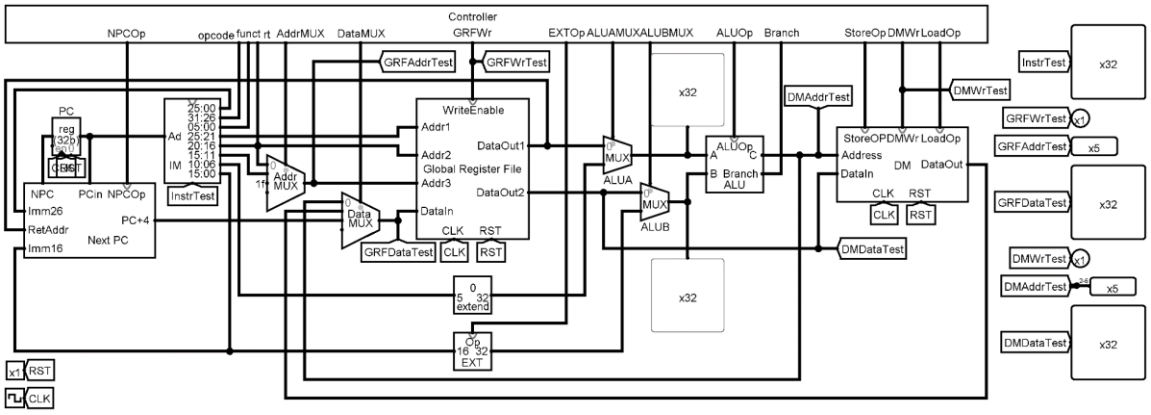

图 1 CPU 顶层模块

(二) 数据通路相关模块实现方法

1. IFU

包含 PC（程序计数器，见上方）、NPC（Next PC，如图 2 所示）、IM（指令存储器，如图 3 所示）及相关逻辑。PC 由起始值为 0x00000000 的具有异步复位功能的 Register 实现，复位值为起始地址。NPC 由计算下一条指令地址的逻辑实现。IM 由容量为32 bit × 32 = 4 B × 32 = 128 B的 ROM 实现。因 IM 实际地址宽度

仅为 5 位，且 Logisim 中 ROM 每个连续的地址都能取一个 word，故取 PC 中储存的地址的第[6:2]位用于在 IM 中寻址。为便于在调试时看到 PC 中的值，故未将 PC、NPC 和 IM 封装在 IFU 这一大模块中。

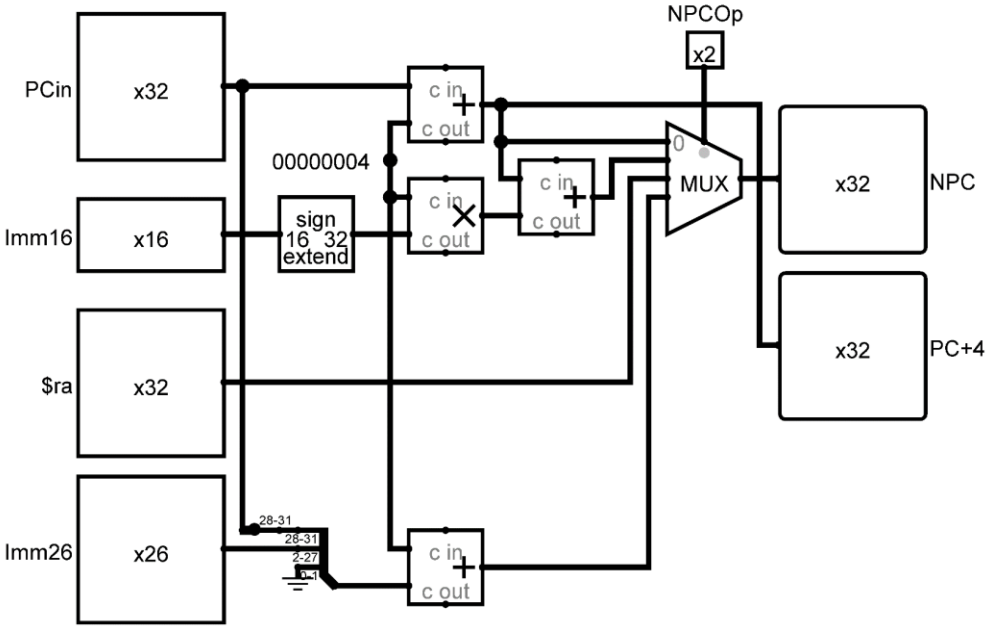

图 2 NPC

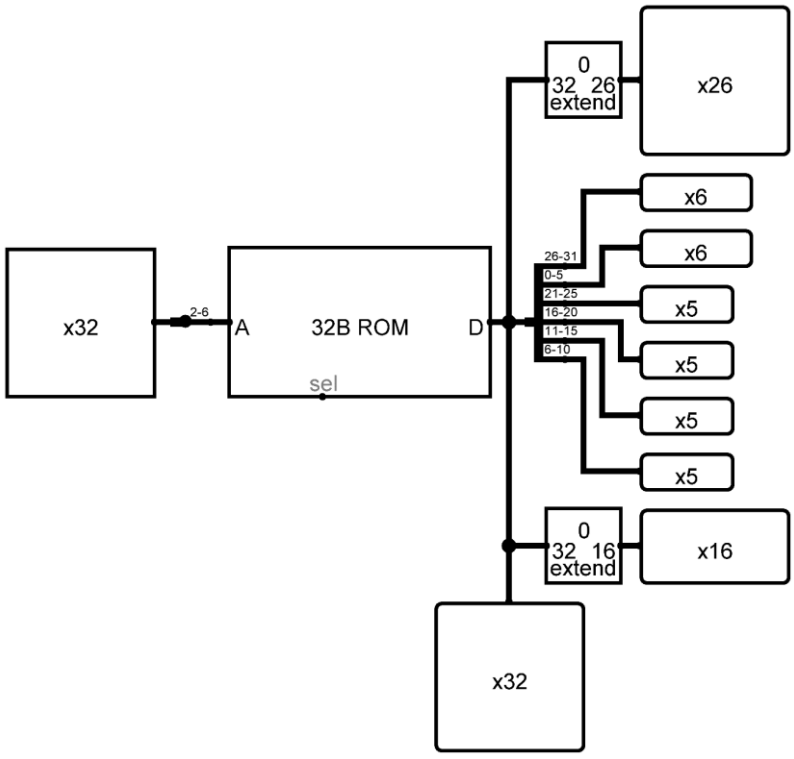

图 3 IM

2. GRF

包含 31 个具有 write-enable 和异步复位功能的 `Register`，如图 4 所示。由于 `0` 号寄存器的值始终为 0，采用 `Ground`（接地线）代替。复位值为 0。

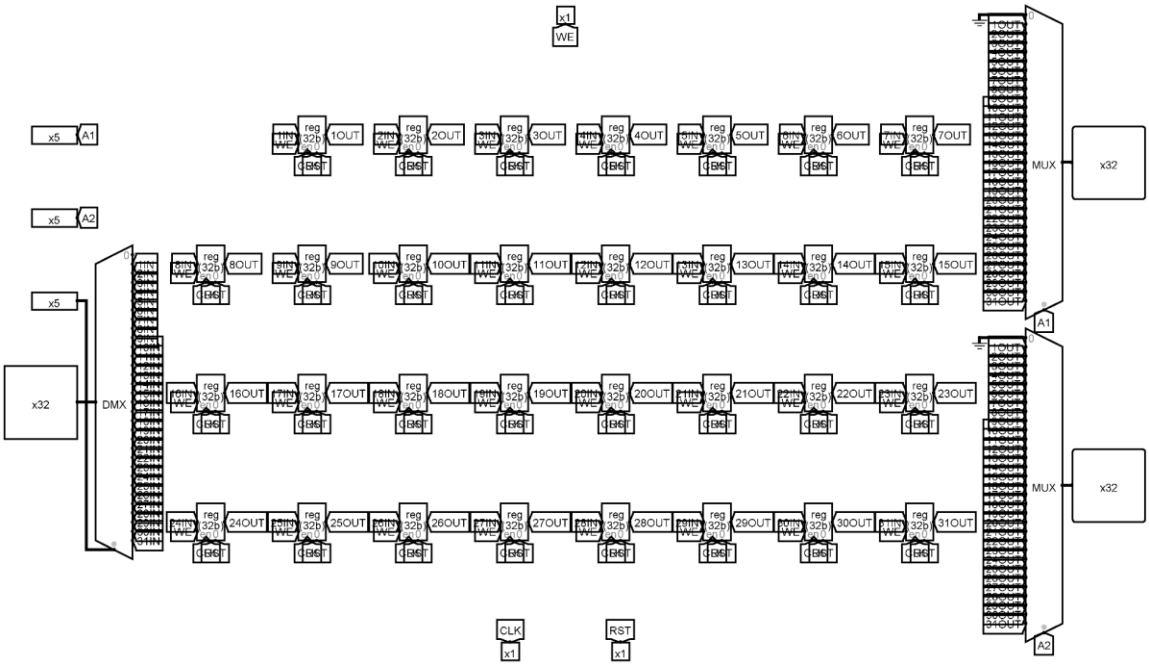

图 4 GRF

3. ALU

支持 32 位加、减、与、或、或非、异或、移位、比较等功能，不检测溢出。输出结果由 `ALUOp` 信号通过 `Multiplexer` 控制。如图 5 所示。

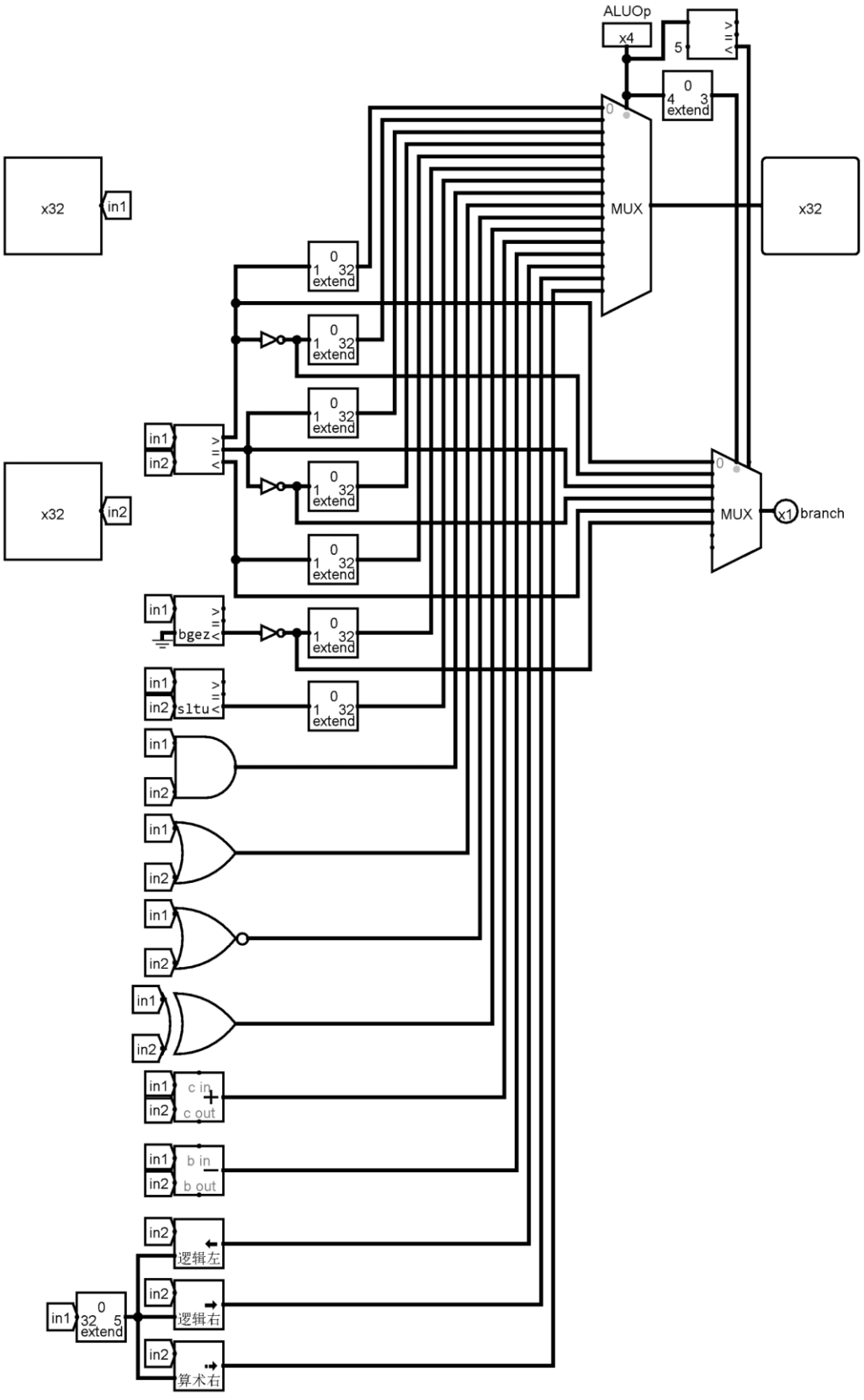
ALUOp
x4
5
0
4 3
extend
MUX
x32
x32
in1
0
1 32
extend
in2
x32
in2
bgez
sltu
MUX
x1
branch
c in
c out
b in
b out
逻辑左
逻辑右
算术右
0
32 5
extend

图 5 ALU

4. DM

由起始地址为 `0x00000000` 的具有异步复位功能的容量为32 bit × 32 = 4 B × 32 = 128 B的 `RAM` 实现，复位值为 `0x00000000`。`RAM` 使用双端口模式，即设置 `RAM` 的 Data Interface 属性为 Separate load and store ports。因 DM 实际地址宽度仅为5位，且 Logisim 中 `RAM` 每个连续的地址都能取一个 word，故取地址的第`[6:2]`位来以 word 为单位寻址，辅以第 `1` 位来得到 half-word、第`[1:0]`位来得到 byte。如图 6 所示。

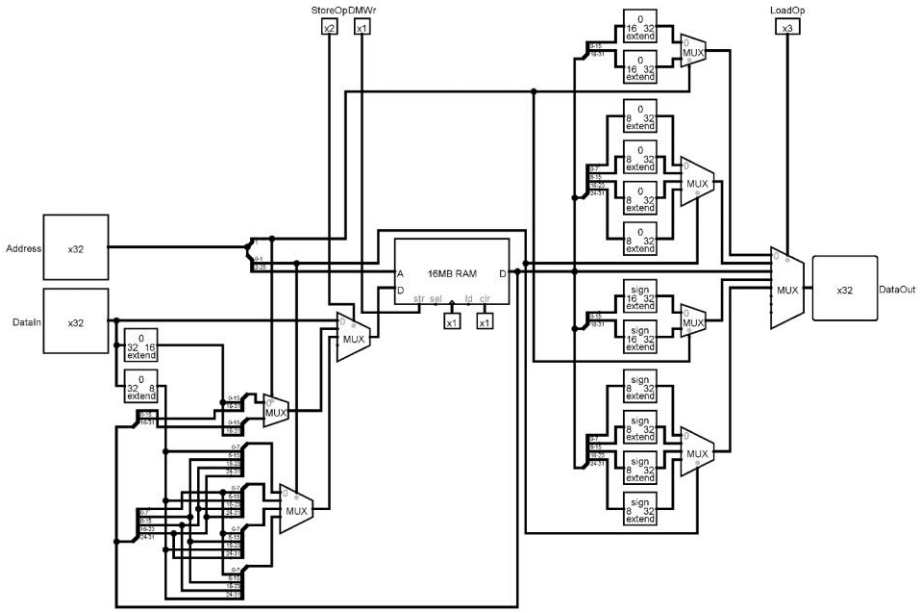

图 6 DM

(三) 核心控制模块实现方法

Controller 将每一条机器指令中的信息，解码为传输给 CPU 各部分的控制信号。实现 Controller 时，可以把解码的逻辑分解为“与逻辑”和“或逻辑”两部分。这样能体现了抽象与模块化的思想，使两部分逻辑目的明确，有助于扩展与调试。“与逻辑”的功能是识别，将输入的机器码识别为相应指令。“或逻辑”的功能是生成，根据输入指令的不同，产生不同的控制信号。由于“或逻辑”要建立从指令到控制信号的映射，理论上可将真值表输入 Logisim 中的 Combinational Analysis 工具自动生成电路。

根据指令编码及其数据通路，控制信号如表 1 所示。

单周期CPU.xlsx

为简化布线且便于添加指令和控制信号，本实验利用了 Logisim 中的 `Tunnel`、`Constant` 和 `Priority Encoder` 进行高度抽象和模块化，并预留了一些空间，如图 7 所示。

表 1 控制信号

Type	Instruction	opcode	funct	NPCOp	GRFWr	AddrMUX	DataMUX	EXTOp	ALUOp	ALUAMUX	ALUBMUX	DMWr	StoreOp	LoadOp
B	**beq**	**000100**		**0b01**	**0**				**0b0010**	**0**	**0**	**0**		
B	bgez	000001		0b01	0				0b0101	0	0	0		
B	bgtz	000111		0b01	0				0b0000	0	0	0		
B	blez	000110		0b01	0				0b0001	0	0	0		
B	bltz	000001		0b01	0				0b0100	0	0	0		
B	bne	000101		0b01	0				0b0011	0	0	0		
I	addiu	001001		0b00	1	0b00	0b00	0b01	0b1011	0	1	0		
I	andi	001100		0b00	1	0b00	0b00	0b00	0b0111	0	1	0		
I	lb	100000		0b00	1	0b00	0b01	0b01	0b1011	0	1	0		0b001
I	lbu	100100		0b00	1	0b00	0b01	0b01	0b1011	0	1	0		0b100
I	lh	100001		0b00	1	0b00	0b01	0b01	0b1011	0	1	0		0b000
I	lhu	100101		0b00	1	0b00	0b01	0b01	0b1011	0	1	0		0b011
I	**lui**	**001111**		**0b00**	**1**	**0b00**	**0b00**	**0b10**	**0b1011**	**0**	**1**	**0**		
I	**lw**	**100011**		**0b00**	**1**	**0b00**	**0b01**	**0b01**	**0b1011**	**0**	**1**	**0**		**0b010**
I	**ori**	**001101**		**0b00**	**1**	**0b00**	**0b00**	**0b00**	**0b1000**	**0**	**1**	**0**		
I	sb	101000		0b00	0			0b01	0b1011	0	1	1	0b10	0b001
I	sh	101001		0b00	0			0b01	0b1011	0	1	1	0b01	0b000
I	slti	001010		0b00	1	0b00	0b00	0b01	0b0100	0	1	0		
I	sltiu	001011		0b00	1	0b00	0b00	0b01	0b0110	0	1	0		
I	**sw**	**101011**		**0b00**	**0**			**0b01**	**0b1011**	**0**	**1**	**1**	**0b00**	**0b010**

Type	Instruction	**opcode**	**funct**	**NPCOp**	**GRFWr**	**AddrMUX**	**DataMUX**	**EXTOp**	**ALUOp**	**ALUAMUX**	**ALUBMUX**	**DMWr**	**StoreOp**	**LoadOp**
I	xori	001110		0b00	1	0b00	0b00	0b00	0b1010	0	1	0		
J	j	000010		0b11	0							0		
J	jal	000011		0b11	1	0b10	0b10					0		
R	**addu**	**000000**	**100001**	**0b00**	**1**	**0b01**	**0b00**		**0b1011**	**0**	**0**	**0**		
R	and	000000	100100	0b00	1	0b01	0b00		0b0111	0	0	0		
R	jalr	**000000**	001001	0b10	1	0b01	0b10					0		
R	jr	000000	001000	0b10	0							0		
R	nor	**000000**	100111	0b00	1	0b01	0b00		0b1001	0	0	0		
R	or	000000	100101	0b00	1	0b01	0b00		0b1000	0	0	0		
R	sll	**000000**	000000	0b00	1	0b01	0b00		0b1101	1	0	0		
R	sllv	000000	000100	0b00	1	0b01	0b00		0b1101	0	0	0		
R	slt	**000000**	101010	0b00	1	0b01	0b00		0b0100	0	0	0		
R	sltu	000000	101011	0b00	1	0b01	0b00		0b0110	0	0	0		
R	sra	**000000**	000011	0b00	1	0b01	0b00		0b1111	1	0	0		
R	srav	000000	000111	0b00	1	0b01	0b00		0b1111	0	0	0		
R	srl	**000000**	000010	0b00	1	0b01	0b00		0b1110	1	0	0		
R	srlv	000000	000110	0b00	1	0b01	0b00		0b1110	0	0	0		
R	**subu**	**000000**	**100011**	**0b00**	**1**	**0b01**	**0b00**		**0b1100**	**0**	**0**	**0**		
R	xor	000000	100110	0b00	1	0b01	0b00		0b1010	0	0	0		
	nop	**000000**	**000000**		**0**							**0**		

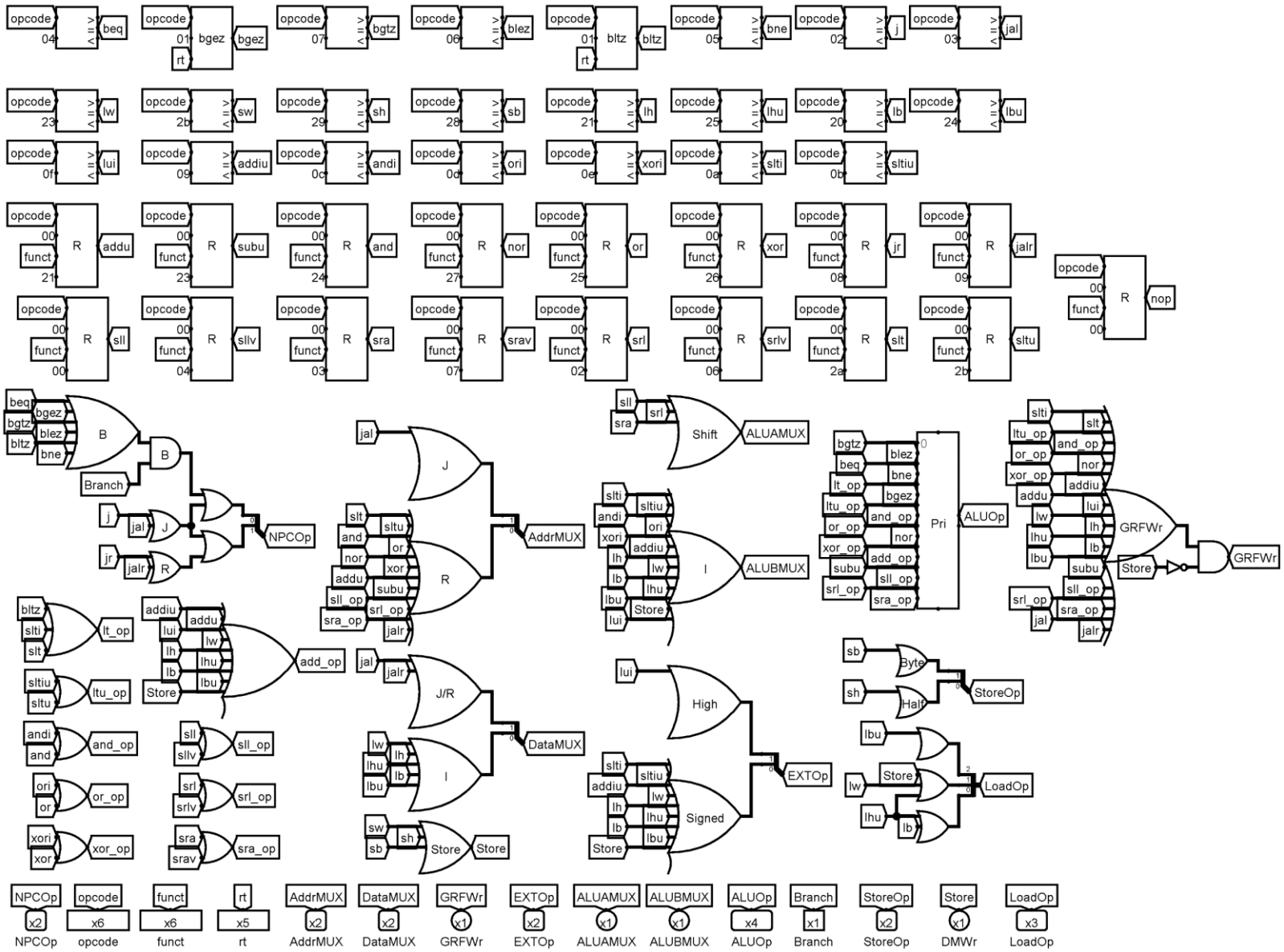

图 7 Controller

二、 测试方案

在搭建电路时，首先通过理论分析确保各模块与指令数据通路逻辑的正确性。然后简单改变各模块的输入，将其输出与正确输出比较。之后在 Windows 平台下使用 Python 3 编写自动测试程序，随机生成指令进行仿真并与正确结果比较，代码如下。若要提高仿真速度，可使用多进程。由于时间和精力所限，目前仅测试了要求的{addu, subu, ori, lw, sw, beq, lui, nop}指令，且 beq 指令仅进行了手工测试。

```
1   import os
2   import subprocess
3   import re
4   import random
5
6   def simulate(name, regex, addr_width, rom_content, raw_circuit, logisim
    _path):
7       circuit = regex.sub(f"addr/data: {addr_width} 32\n{rom_content}</a>
    ", raw_circuit)
8       with open(f"{name}.circ", "w", encoding="utf-8") as f:
9           f.write(circuit)
10      with open(f"{name}.txt", "w", encoding="utf-8") as f:
11          subprocess.Popen(f"java -jar {logisim_path} {name}.circ -
    tty table", shell=False, stdout=f)
12
13  path = os.path.dirname(os.path.realpath(__file__))
14  os.chdir(path)
15
16  Logisim_path = "Logisim.jar"
17  MARS_path = "MARS.jar"
18  query_addr_width = 5
19  master_addr_width = 5
20  test_time = 10
21  with open("CPU-query.circ", encoding="utf-8") as file:
```

```
    raw_query_circuit = file.read()
with open("CPU-master.circ", encoding="utf-8") as file:
    raw_master_circuit = file.read()

query_regex = re.compile(rf"addr/data: {master_addr_width} 32[\s\S]*?</a>")
master_regex = re.compile(rf"addr/data: {master_addr_width} 32[\s\S]*?</a>")
for j in range(test_time):
    print(f"Testing case {j}...")
    with open(f"test_{j}.asm", "w") as file:
        for i in range(5):
            target = random.randint(0, 27)
            source = random.randint(0, 27)
            imm = random.randint(0, 12284)
            file.write(f"lui\t${i}, {imm}\n")
        for i in range(5):
            target = random.randint(0, 27)
            source = random.randint(0, 27)
            imm = random.randint(0, 12284)
            file.write(f"ori\t${i}, {imm}\n")
        for i in range(5):
            target = random.randint(0, 27)
            source = random.randint(0, 27)
            imm = random.randint(0, 7) * 4
            file.write(f"sw\t${target}, {imm}(${source})\n")
            file.write(f"lw\t${target}, {imm}(${source})\n")
        for i in range(5):
            target = random.randint(0, 27)
            source = random.randint(0, 27)
            rd = random.randint(0, 27)
            file.write(f"addu\t${target}, ${source}, ${rd}\n")
```

```
52          for i in range(5):
53              target = random.randint(0, 27)
54              source = random.randint(0, 27)
55              rd = random.randint(0, 27)
56              file.write(f"subu\t${target}, ${source}, ${rd}\n")
57          file.write(f"nop\nnop\n")
58
59      subprocess.call(f"java -
    jar {MARS_path} test_{j}.asm nc mc CompactTextAtZero a dump .text HexTe
    xt rom_{j}.txt")
60      with open(f"rom_{j}.txt") as file:
61          rom_content = file.read()
62
63      query_name = f"query_{j}"
64      simulate(query_name, query_regex, query_addr_width, rom_content, ra
    w_query_circuit, Logisim_path)
65
66      master_name = f"master_{j}"
67      simulate(master_name, master_regex, master_addr_width, rom_content,
     raw_master_circuit, Logisim_path)
68
69      output = subprocess.Popen(f"fc {query_name}.txt {master_name}.txt",
     shell=True, stdout=subprocess.PIPE).communicate()[0].decode("gbk")
70      print(output)
71      if "FC: 找不到差异" in output:
72          print(f"Test case {j} succeeded!")
73      else:
74          print(f"Test case {j} failed!\n")
```

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。

这是最简便的做法。不足在前文已述，每个连续的地址都能取一个 word，故只能取地址的第[6:2]位来以 word 为单位寻址，辅以第 1 位来得到 half-word、第[1:0]位来得到 byte。

(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。

因为 nop 指令编码为 0x00000000，相当于 sll $0, $0, 0，而 0 号寄存器本来就不能被写入，也不会影响 PC、IM、GRF 和 DM 中的值，也不会影响评测用的输出信号。但为了完整性，还是将其加入了控制信号真值表。

(三) 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，并阐释为了解决这个问题，你最终采用的方法。

在 MARS 中可以分别导出.data 段和.text 段的数据，所以这不成问题。

(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。

形式验证的优点是对设计的正确性有理论保障，增强设计正确性的信心。缺点是很多时候需要人工完成，耗费时间精力且仍可出错。无论如何，测试都必不可少。