计算机组成课程设计

P3 课下测试 – Logisim 单周期

周美廷-76066002

*由于本人是留学生，随最终文档依然使用中文写但为了本人无需翻来覆去看英文版和中文版，于是将文档写成两种语言

1. 顶层设计 (The Basic Idea)

以下便是整个 CPU 的顶层设计图（Below is the design pic of the outer layer of the CPU）

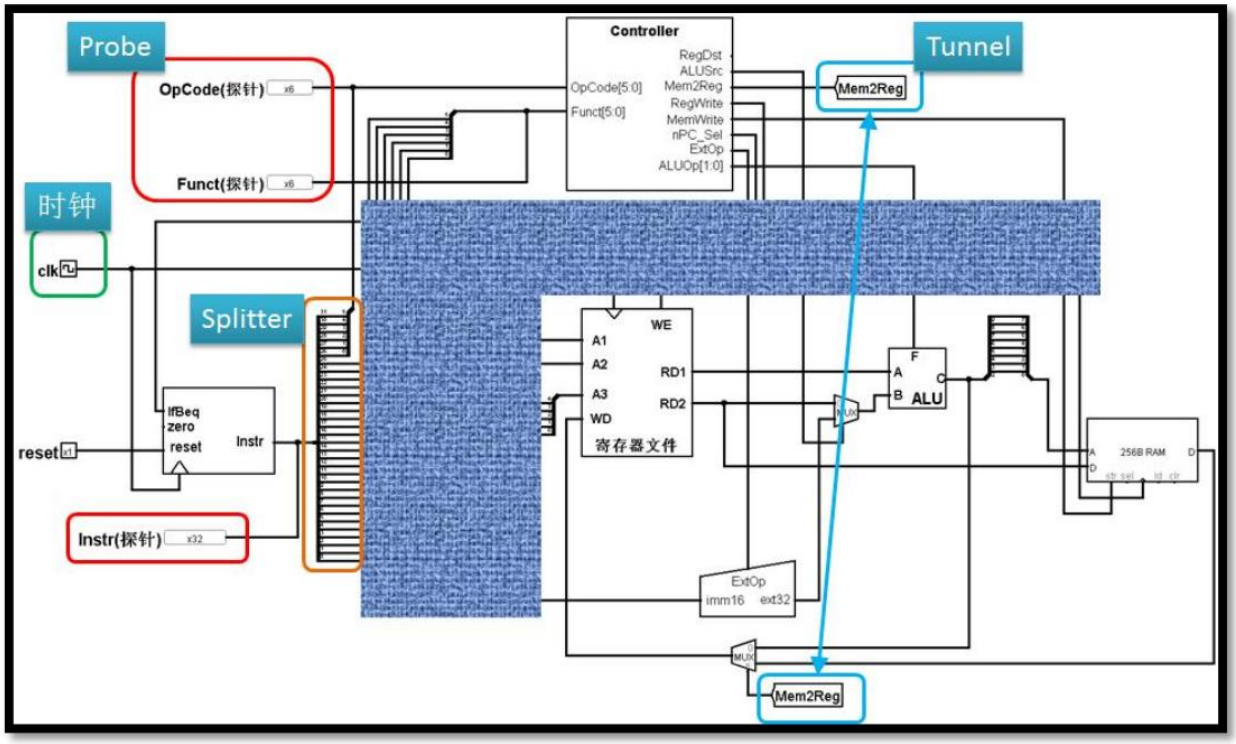

图 1 顶层设计图

以下图片便是本人在 Logisim 设计出的 CPU（Below is the actual design that I implemented in Logisim）

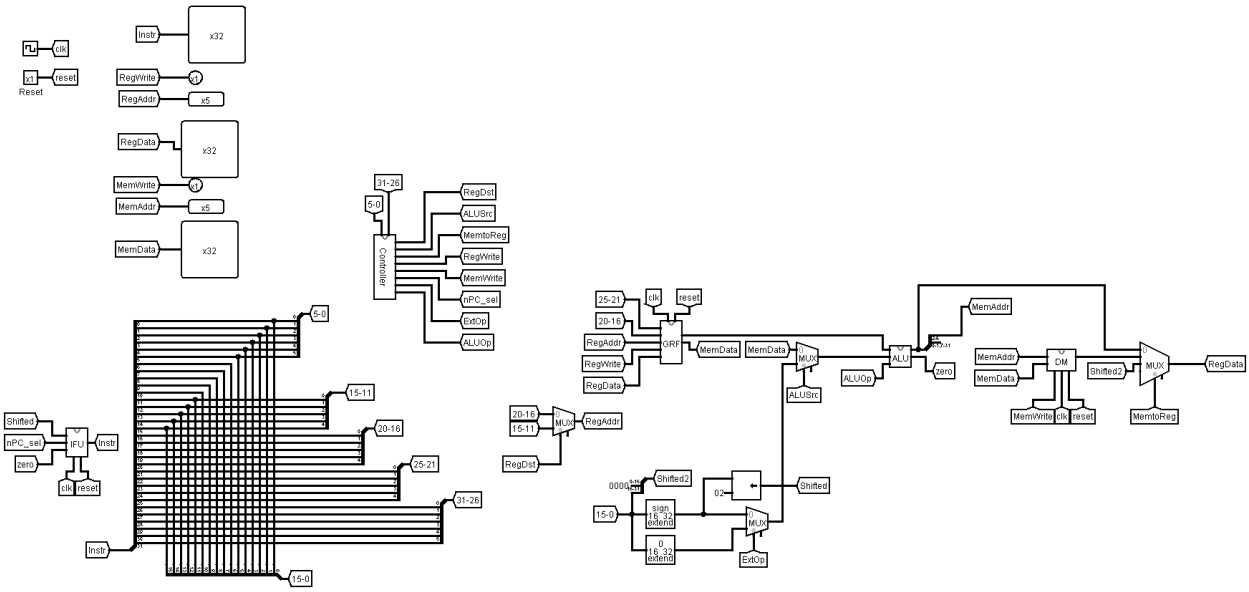

图 2 Logisim 中的顶层设计

That’s it for the overview of the CPU, next is the specifications of each module.

2. 模块规格 (Module Specifications)

这种单周期 CPU 应采用模块化设计，因此它包含多个具有不同用途的模块。

This single-cycle CPU is meant to be modularly designed and therefore it consists of several modules with different purposes.

I. IFU（取指令单元 / **Instruction Fetch Unit**）

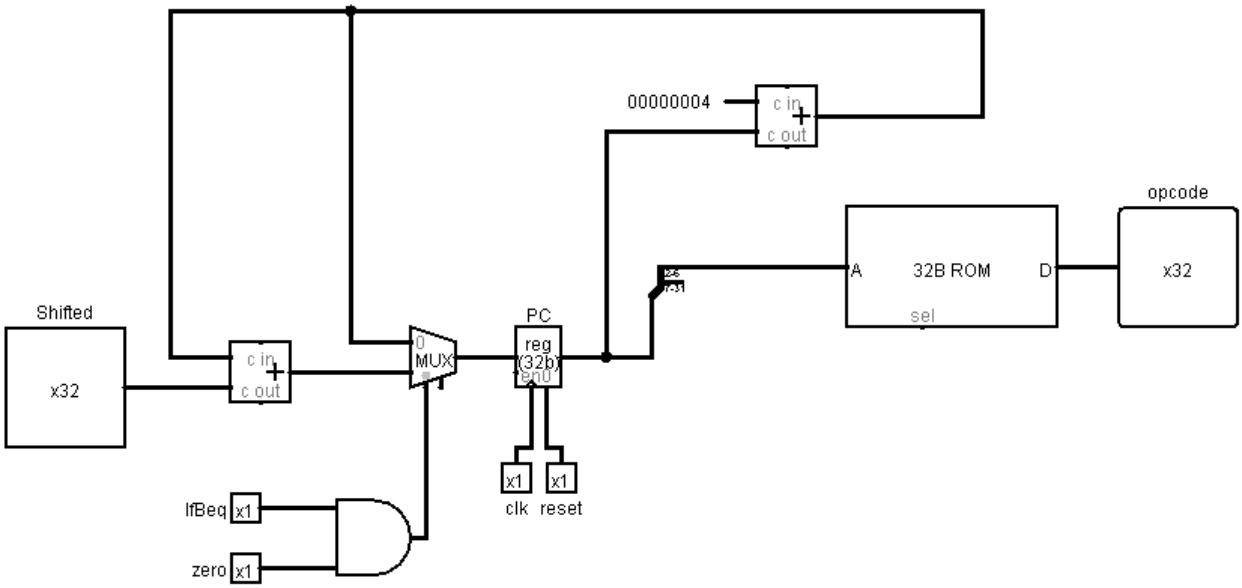

图 3 IFU 模块设计图

It is already clear from the name that this module is meant to generate the instruction code (later named opcode in the overall design). To implement this module, there are two main modules, PC (Program Counter) and IM (Instruction Memory).

设计说明/Design Details:

- PC 用寄存器实现，应具有复位功能（PC is implemented by using **Registers**, it should have **Reset** feature）
- 起始地址/Starting address: 0x00000000
- IM 用 ROM 实现，容量为 32bit * 32（IM is implemented by using **ROM**, the capacity is **32bit * 32**）
- 因 IM 实际地址宽度仅为 5 位，故需要使用恰当的方法将 PC 中储存的地址同 IM 联系起来(The Address Bit Width of the IM is 5, therefore there should be a way to connect the address saved in PC with IM)。

模块流程/Design Workflow:

- 来自 PC 的地址将被处理进行下一个操作，首先它将 +4（常量），这是为了实现 PC + 4

Address from PC will be processed for the next process, it will be first added by 4 (constant), this is to implement the PC+4

- 然后 PC + 4（地址）的结果将被带到多路复用器，因为如果它是跳转指令之类的指令，这将不起作用，因为跳转指令将直接跳转到相应的地址/标签。 这就是为什么多路复用器由 Beq 指令的 bool 信号和 Zero bool 的 AND 逻辑判断的（零 bool 是从 ALU 产生的，如果两个 ALU 数据输入相等则它将为 TRUE）。

  The result of PC+4 (the address) will then be brought to a Multiplexer, because this wouldn't work if it's instructions such as jump instructions because jump instructions will directly jump to the corresponding address/label. And this is why the multiplexer is decided by an AND logic input of Beq instruction bool and Zero bool (the zero bool is generated from the ALU, it will be TRUE if both of the ALU inputs are equal).
- 那么多路复用器的第一个输入是普通 PC + 4，而第二个输入是 PC + 4 和移位地址的总和（相应标签所在的地址）

  So then the first input of the Multiplexer is the normal PC+4 while the second input is the sum of PC+4 and the shifted address (the address of where the corresponding label is)
- 指令代码（MIPS 指令的机器代码）保存在 IM（ROM）中，并将根据 PC 的结果进行访问（判断是否使用 PC + 4 访问下一个地址还是 PC + X）

  Instruction codes (The machine code of MIPS instructions) are saved in the IM (ROM) and will be accessed according to the result of the PC (whether just using the PC+4 which means accessing the next address or PC+X)
- PC 访问的指令代码将被传送到名为操作码的输出

  The instruction code that's accessed by the PC will be transferred to the output called opcode

表 1 IFU 规格

功能名称	方向	功能描述
IfBeq	I	当前置零是否为 beq 1：指令为 beq 0：指令不为 beq
Shifted	I	第 0-15 指令被 Extend 后移植

zero	I	ALU 计算是否为 0 1：为 0 0：不为 0
clk	I	时钟信号
reset	I	复位信号
opcode[31:0]	O	32 位 MIPS 指令

II. GRF（通用寄存器组/General Register File）

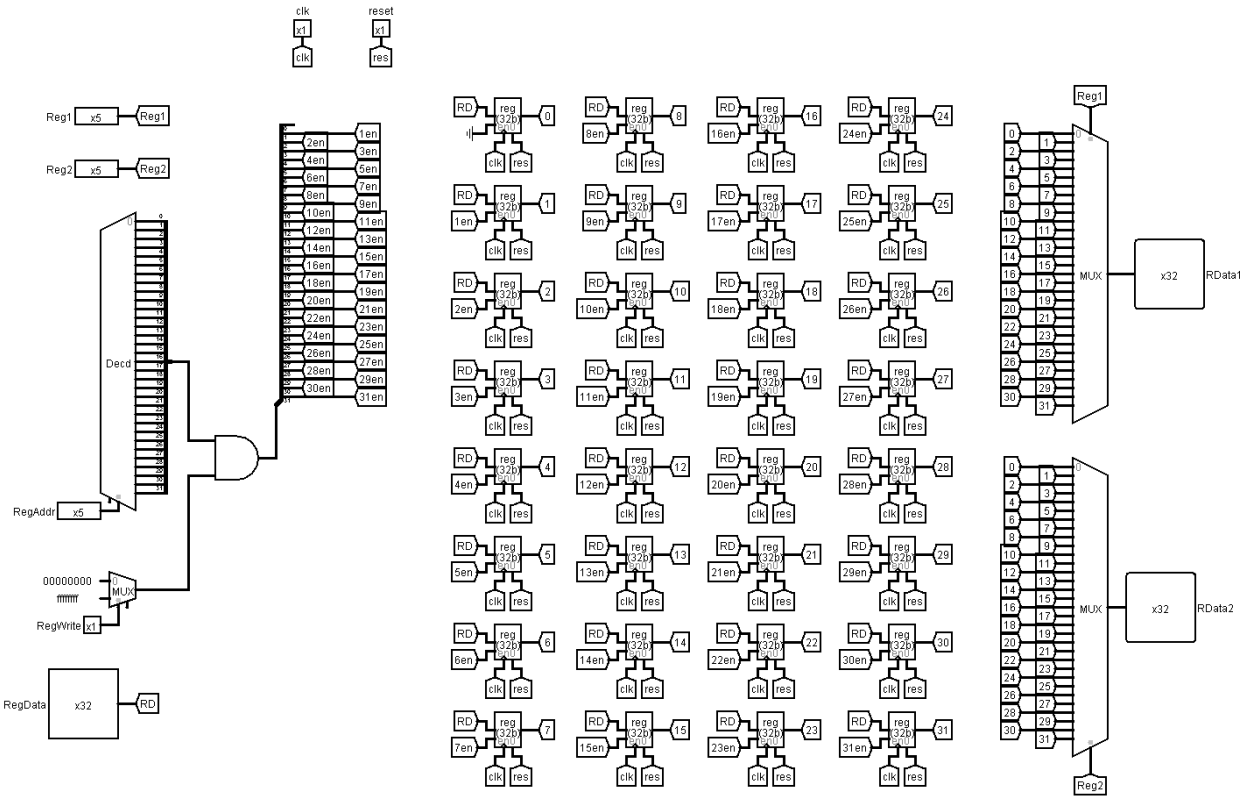

图 4 GRF 模块设计图

This module is meant to ‘turn’ the MIPS instruction code into several parts to be able to do further operations of different instructions. Then data of previous operations will be stored to one of the 32 registers. These 32 registers represent the 32 registers in MIPS. The MIPS instruction code itself consists of several structure parts, and each type of instructions have different structures. For example, for ADD instruction, the format is **addu rd, rs, rt**. This could be explained as rs (25-21) + rt (20-16) = rd (15-11). In case of BEQ instruction, the

format is **beq rs, rt, offset**, this could be explained as if rs (25-21) == rt (20-16), the jump to offset (15-0).

设计说明/Design Details:

- 用具有写使能的寄存器实现，寄存器总数为 32 个（This is implemented with write-enabled registers, in total there are 32 registers）
- 0 号寄存器的值始终保持为 0。其他寄存器初始值均为 0，无需专门设置（The value of Register No. 0 is always 0. The other registers have an initial value of 0 so they don't require special settings）

模块流程/Design Workflow:

- In total, there are 5 different inputs in this GRF module:
  - The first two inputs are taken from the instruction code generated by IFU (Address no. 25-21 (rs/base/offset) and 20-16 (rt))
  - The third one is either Address no 20-16 (rt) in case of **ori, lw, sw, beq, lui** or 15-11 (rd) in case of **addu** and **subu**
  - The fourth one is the RegWrite signal (this will be TRUE if the current instruction isn't **sw** nor **beq**)
  - The fifth one is Register Data (RegData)
- The very obvious part of this module is the number of registers that it has (32 registers), so let's start from this part. The RegData will be loaded to each register.
- The RegAddr (third input) will be decoded into 32 bit, and transferred to an AND logic. The other input for the AND logic is the signal of whether the instruction is **sw/beq** or not. This combination basically just decides which register should have the write function enabled therefore allowing data to be written to the corresponding register
- The result of the registers will then be the content of Output 1 and Output 2, as for which register result will be used it will be decided by the first 2 inputs

表 2 GRF 规格

功能名称	方向	功能描述
RegData[31:0]	I	写入数据

RegWrite	I	读写控制信号 1：写 0：读
clk	I	时钟信号
reset	I	复位信号
RegAddr[4:0]	I	写寄存器地址
Reg1[4:0]	I	读寄存器地址 1
Reg2[4:0]	I	读寄存器地址 2
RData1[31:0]	O	32 位数据 1
RData2[31:0]	O	32 位数据 2

III. ALU（算术逻辑单元/Arithmetic Logic Unit）

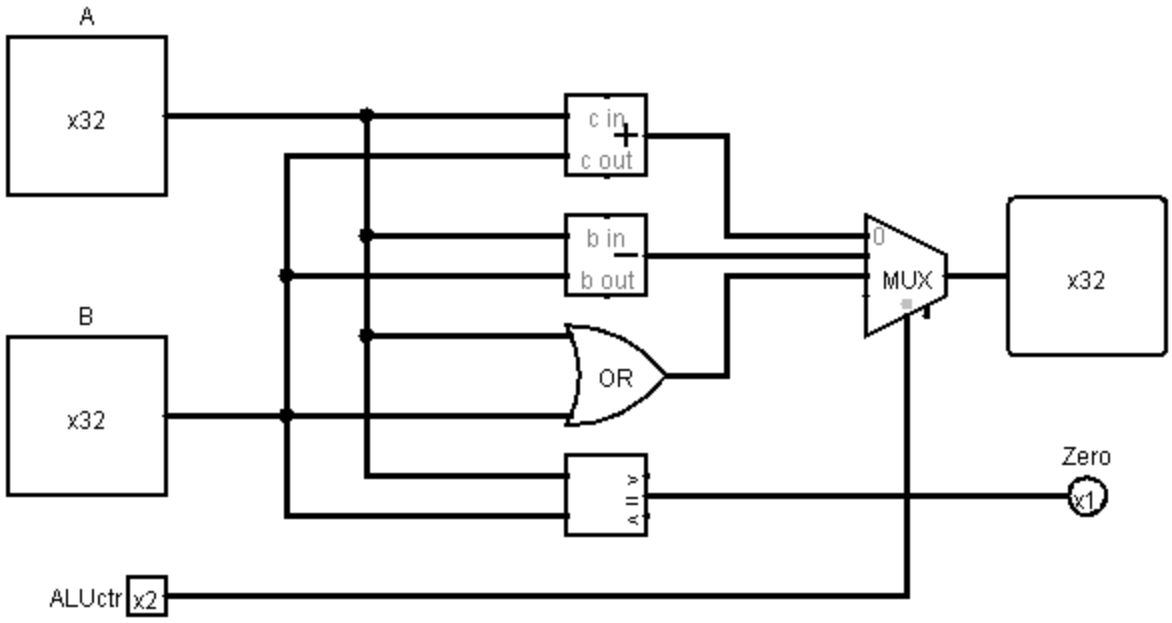

图 5 ALU 模块设计图

ALU is the most common module, this module is meant to perform arithmetic operations such as addition, subtraction, comparation, etc.

设计说明/Design Details:

- 提供 32 位加、减、或运算及大小比较功能（Provides 32-bit addition, subtraction, OR operation and size comparison）
- 可以不支持溢出（不检测溢出）（Overflow may not be supported (no overflow detected)）

模块流程/Design Workflow:

- According to the instruction that's through ALUctr input (this instruction is generated by the Controller module), the module will perform corresponding operation. The **ori** instruction will cause ALU to perform OR operation (10), the **subu** will cause ALU to perform subtraction operation (01), and the other instructions will simply cause ALU to perform addition operation.
- At the same time, both data inputs A and B will be checked, if the values are equal, then the Zero signal will be TRUE

表 3 ALU 规格

功能名称	方向	功能描述
A[31:0]	I	输入数据 A
B[31:0]	I	输入数据 B
ALUctr[1:0]	I	ALU 控制信号 00：加法运算 01：减法运算 10：或运算 11：比较
zero	O	计算结果是否为 0
Output[31:0]	O	输出结果 32 位

IV. DM（数据存储器/Data Memory）

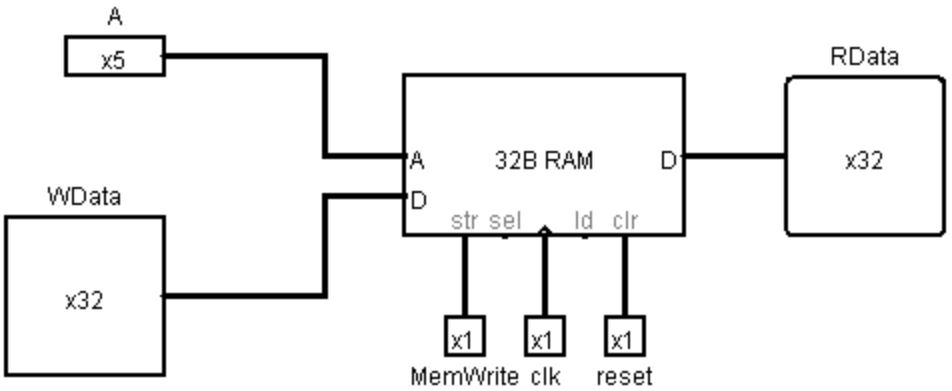

图 6 DM 模块设计图

The DM module is meant to save data generated from previous operations.

设计说明/Design Details:

- 使用 RAM 实现，容量为 32bit * 32（Implemented by using RAM with a capacity of 32bit * 32）
- 起始地址/Starting Address: 0x00000000
- RAM 应使用双端口模式，即设置 RAM 的 Data Interface 属性为 Separate load and store ports（The RAM should use dual port mode, which sets the **Data Interface** property of RAM to **Separate load and store ports**）

模块流程/Design Workflow:

- This is just a simple module to store data to the RAM, whether the CPU will write the data is decided by the MIPS instruction sent to MemWrite. If the instruction is **sw**, MemWrite will be TRUE, therefore storing the data to the RAM.
- For later uses, the data will be chosen by the input A, this input is generated by the ALU module, which means if the operation doesn't include any data that were stored to the RAM, the output of this module would be 0

表 4 DM 规格

功能名称	方向	功能描述
A[4:0]	I	MemAddr 地址

WData[31:0]	I	输入数据 32 位
MemWrite	I	读写控制信号
clk	I	时钟信号
reset	I	复位信号
RData[31:0]	O	输出结果 32 位

V. EXT（Extender）

在本人的设计中正是使用 Logisim 内置的 Bit Extender （In this design, I'm using the Logisim built-in Bit Extender plugin）

VI. Controller（控制器）

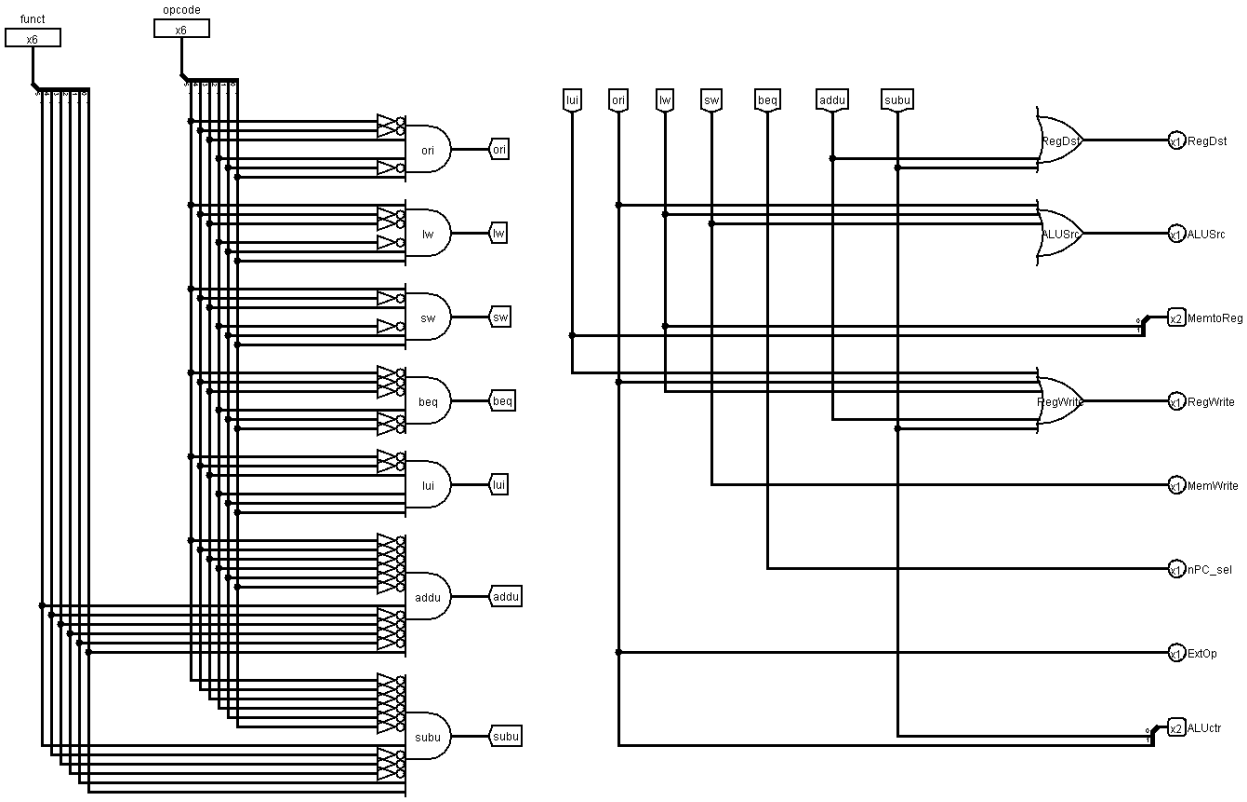

图 7 控制器模块设计图

As mentioned in the requirements, this module is divided into two parts: AND logic and OR logic. The AND logic will identify the input machine code as the corresponding instruction while the OR logic will generate different control according to the input instruction.

模块流程/Design Workflow:

- The input of this module is the 5-0 and 31-26 address bit of the instruction code generated from IFU. While most MIPS instruction opcode are located in the 31-26 bit of the instruction code, it works differently for **addu** and **subu**. The opcode for both **addu** and **subu** are 000000 but they got function code in the 5-0 bit of the instruction code. The complete code is listed in the table below.
- RegDst means that the register that will be used is the rd register, this is used by **addu** and **subu** while the other instructions will just use the rt register.
- MemtoReg literally means loading data from the memory to the register. Of course, this signal will be true if the instruction code is a loading instruction such as **lw** or **lui**
- RegWrite signal is to let the CPU write the data into a register. This will be used by instructions that cause changes to data so that it needs to be written (saved) in the register such as **lui, ori, lw, addu, subu**
- MemWrite is a signal to let the CPU write the data into memory. This is only used by instructions that will store data such as **sw**
- The nPC_sel signal basically is just a bool to see if it's a branch instruction. Since in this project there's only one branch instruction **beq**, therefore only **beq** uses this signal. This signal will then be used in the IFU module to decide whether to use PC+4 or PC+X for the address movements

表 5 控制器指令真值表

funct	100001	100011	n/a					000000
opcode	000000	000000	001101	100011	101011	000100	001111	000000
	addu	subu	ori	lw	sw	beq	lui	nop
RegDst	1	1	0	0	x	x	0	x
ALUSrc	0	0	1	1	1	0	x	x

MemtoReg[1:0]	00	00	00	01	xx	xx	10	xx
RegWrite	1	1	1	1	0	0	1	0
MemWrite	0	0	0	0	1	0	0	0
nPC_sel	0	0	0	0	0	1	0	0
ExtOp	x	x	1	0	0	x	0	x
ALUctr[1:0]	Add	Subtract	Or	Add	Add	xx	xx	xx

表 6 控制器规格

功能名称	方向	功能描述
funct[5:0]	I	Function 6 位
opcode[5:0]	I	Opcode 6 位
RegDst	O	写地址控制
ALUSrc	O	控制 ALU
MemtoReg[1:0]	O	DM 读控制
RegWrite	O	GRF 读写控制
MemWrite	O	DM 写控制
nPC_sel	O	beq 指令标志
ExtOp	O	Ext 扩展控制
ALUctr[1:0]	O	ALU 运算操作控制

3. 测试程序说明 (Testing Program)

测试指令集：

```
.data
    testarr: .space 8

.text
lui $t0, 0x1234                    #Load upper immediate (of high-order 16-bit),write
0x12340000 to $t0
```

```
ori $t1, $t0, 0x0001        #Set $t1 to bitwise OR of $t0 and 0x12340001
lui $t0, 0x0000             #Load upper immediate (of high-order 16-bit),write
0x00000000 to $t0
ori $t1, $t0, 0x0000        #Set $t1 to bitwise OR of $t0 and 0x0000000
nop                         #Null op

addu $t2, $t1, 3            #Set $t2 to 0x10000003
subu $t3, $t1, $t2          #Set $t3 to (ALU operation $t1 - $t2 = 0xfffffffd), no overflow
(nothing changed)
nop                         #Null op

sw $t3, testarr($zero)      #Write $t3's value to DM[testarr+$zero] 0xfffffffd
beq $t3, $t2, any           #ALU checks if GRF[t3]==GRF[t2], if yes then goto any, else
PC+4
lw $t3, testarr($zero)      #Load DM[testarr+$zero] to GRF[t2] 0xfffffffd
nop                         #Null op

any: ori $t0, $t0, 0x1010   #Set $t0 to bitwise OR of itself ($t0) and 0x1000001 ->
0x10001010
```

机器码：

```
00000000
3c081234
35090001
3c080000
35090000
00000000
00000000
3c010000
34210003
01215021
012a5823
00000000
ac0b0000
```

116a0002

8c0b0000

00000000

35081010

4. 思考题

I. (L0.T2)

1. 若 PC（程序计数器）位数为 30 位，试分析其与 32 位 PC 的优劣。

   优势：32 位 PC 可以访问 PC+4，比 30 位的访问的空间多。

   劣势：32 位 PC 的设计比 30 位 PC 的更复杂。由于 32 位 PC 的低两位都是 0，如果用 30 位 PC 的话，每次 PC+1 即可，相当于 32 位 PC 的高 30 位

2. 现在我们的模块中 IM 使用 ROM， DM 使用 RAM， GRF 使用寄存器，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。

   IM 里的值是不能被改变的，所以 IM 应该用 ROM

   DM 存储操作数据，有可能会有更新，所以用 RAM

   GRF 是临时存储单元，所以用寄存器。

II. (L0.T3)

1. 结合上文给出的样例真值表，给出 RegDst， ALUSrc， MemtoReg，RegWrite, nPC_Sel, ExtOp 与 op 和 func 有关的布尔表达式（表达式中只能使用“与、或、非”3 种基本逻辑运算。）

   $RegDst = \overline{Op5}\ \overline{Op4}\ \overline{Op3}\ \overline{Op2}\ \overline{Op1}\ \overline{Op0}\ Func5\ \overline{Func4}\ \overline{Func3}\ \overline{Func2}\ Func0$

   $RegWrite = \overline{Op5}\ \overline{Op4}\ \overline{Op3}\ \overline{Op2}\ \overline{Op1}\ \overline{Op0}\ Func5\ \overline{Func4}\ \overline{Func3}\ \overline{Func2}\ Func0$

   $+\ \overline{Op5}\ \overline{Op4}\ Op3\ Op2\ Op1\ Op0$

   $+\ Op5\ \overline{Op4}\ \overline{Op3}\ \overline{Op2}\ Op1\ Op0$

   $ALUSrc = \overline{Op5}\ \overline{Op4}\ Op3\ Op2\ Op0 + Op5\ \overline{Op4}\ \overline{Op2}\ Op1\ Op0$

   $MemtoReg = Op5\ \overline{Op4}\ \overline{Op3}\ \overline{Op2}\ Op1\ Op0$

   $nPC\_Sel = \overline{Op5}\ \overline{Op4}\ \overline{Op3}\ Op2\ \overline{Op1}\ \overline{Op0}$

$ExtOp = Op5\ \overline{Op4}\ \ \overline{Op2}\ Op1\ Op0$

2. 充分利用真值表中的 X 可以将以上控制信号化简为最简单的表达式， 请给出化简后的形式。

$RegDst = \overline{Op5}\ \ \overline{Op4}\ \ \overline{Op3}\ \ \overline{Op2}\ \ \overline{Op1}\ \ \overline{Op0}\ Func5\ \overline{Func4}\ \ \overline{Func3}\ \ \overline{Func2}\ Func0$

$RegWrite = \overline{Op5}\ \ \overline{Op4}\ \ \overline{Op3}\ \ \overline{Op2}\ \ \overline{Op1}\ \ \overline{Op0}\ Func5\ \overline{Func4}\ \ \overline{Func3}\ \ \overline{Func2}\ Func0$

$+\ \overline{Op5}\ \ \overline{Op4}\ Op3\ Op2\ Op1\ Op0$

$+ Op5\ \overline{Op4}\ \ \overline{Op3}\ \ \overline{Op2}\ Op1\ Op0$

$ALUSrc = \overline{Op5}\ \ \overline{Op4}\ Op3\ Op2\ Op0 + Op5\ \overline{Op4}\ \ \overline{Op2}\ Op1\ Op0$

$MemtoReg = Op5\ \overline{Op4}\ \ \overline{Op3}\ \ \overline{Op2}\ Op1\ Op0$

$nPC\_Sel = \overline{Op5}\ \ \overline{Op4}\ \ \overline{Op3}\ Op2\ \overline{Op1}\ \ \overline{Op0}$

$ExtOp = Op5\ \overline{Op4}\ \ \overline{Op2}\ Op1\ Op0$

3. 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 的机器码是 0x00000000（全 0）。所以 nop 执行的时候，MemWrite,WriteEn 等要么是 0，要么是 x。

III. (L0.T4)

1. 前文提到，“可能需要手工修改指令码中的数据偏移”，但实际上只需再增加一个 DM 片选信号,就可以解决这个问题。请阅读相关资料并设计一个 DM 改造方案使得无需手工修改数据偏移。

假设 DM 为 256MB，在 0x30000000~0x3FFFFFFF 区间，在做选择时，只需要当前指令取出来。ALU 运算后得到的地址的高 4 位与 0x3 作比较(无符号)，大于则片选信号为 1，小于则为 0。

2. 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，简要阐述相比与测试，形式验证的优劣。

优势：

1. 形式验证是用数学方法将待验证电路和功能描述或参考设计直接进行比较，所以测试者不必考虑如何获得测试向量
2. 形式验证对指定描述所有可能的情况进行验证，而不仅仅对其中的一个子集进行多次试验，因此有效地克服了模拟验证的不足。
3. 形式验证可以短时间内进行从系统级到门级的验证，所以可以尽快发现和改正电路设计中的一些错误。

劣势：

形式验证还不能有效的验证电路的性能（电路的时延和功耗等）。