# 单周期CPU设计文档2

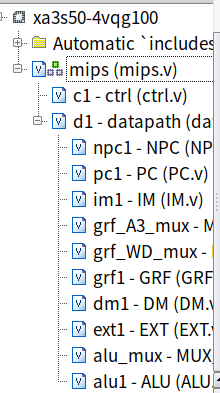
## CPU设计方案综述

支持指令：

addu、subu、ori、lui

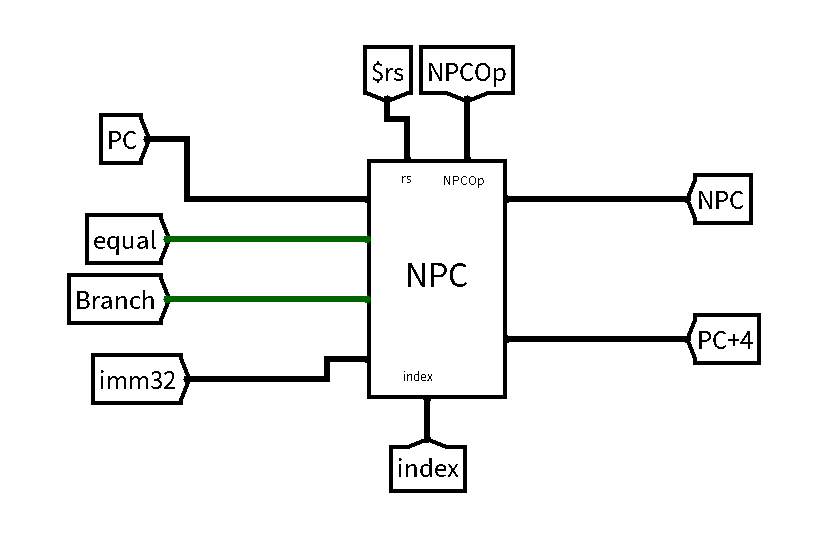
sw、lw、sb、lb、sh、lh

beq、jal、j、jalr、jr



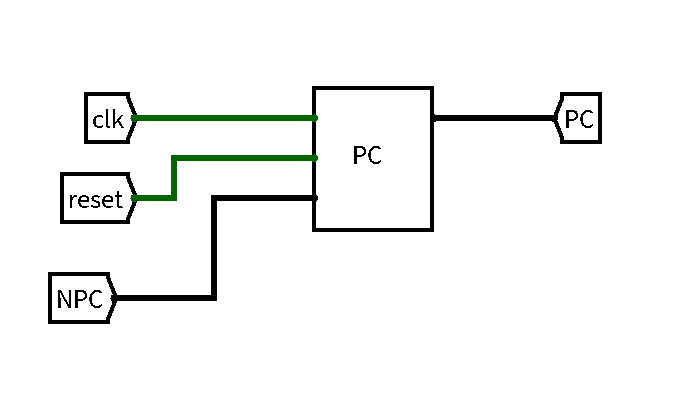
### NPC

| 序号 | 信号名 | 方向 | 描述 |
| --- | --- | --- | --- |
| 1 | pc[31:0] | I | 32位地址信号 |
| 2 | equal | I | A1(rs)、A2(rt)对应GRF寄存器中的值是否相等  1：相等  0：不相等 |
| 3 | imm32[31:0] | I | 16位immediate的32位**符号扩展** （beq） |
| 4 | index[25:0] | I | 26位instr\_index（j、jal） |
| 5 | Grs[31:0] | I | rs寄存器保存值（jalr、jr）PC  GPR[rs] |
| 6 | NPCOp[2:0] | I | NPC将会是  000：PC+4  001：PCbranch（beq）  010：PC31..28instr\_index02 PCJ(al)(jal、j)  011：GPR[rs] PCJ(al)r(jalr、jr) |
| 7 | npc[31:0] | O | 下一个更新的PC值 |
| 8 | pc4[31:0] | O | 现在PC的值+4（jalr、jal会用到） |



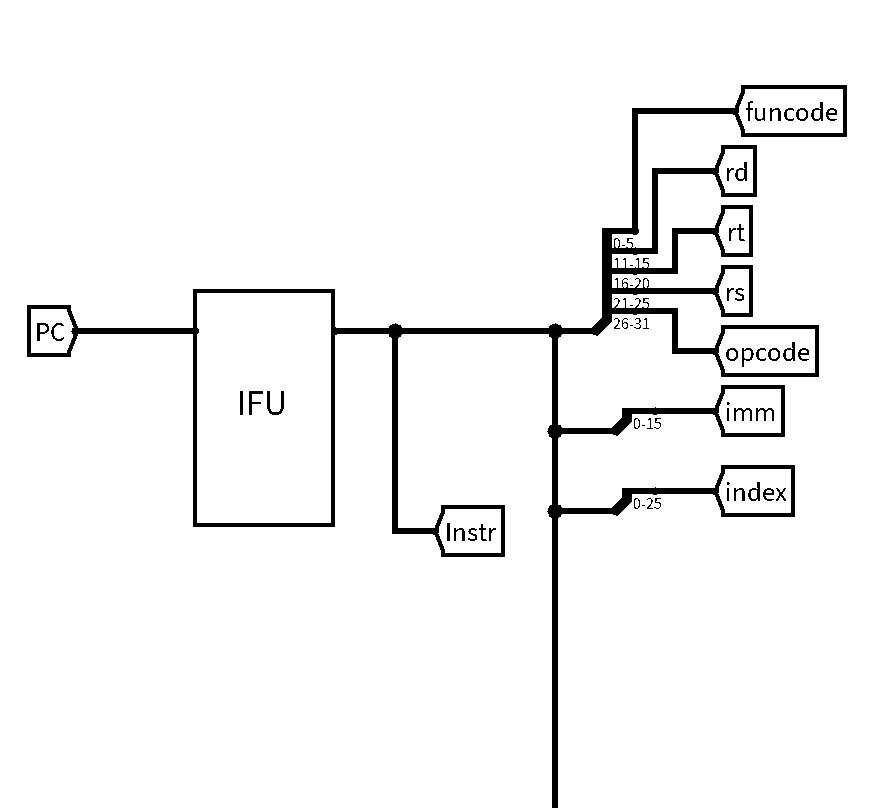
### PC

| 序号 | 信号名 | 方向 | 描述 |
| --- | --- | --- | --- |
| 1 | clk | I | 时钟信号 |
| 2 | reset | I | 同步复位信号，将PC值置为**0x00003000**  高电平有效 |
| 3 | npc[31:0] | I | 要更新的PC值 |
| 4 | pc[31:0](reg) | O | 现在的PC值 |



### IFU

| 序号 | 信号名 | 方向 | 描述 |
| --- | --- | --- | --- |
| 1 | pc[31:0] | I | 程序计数器信号 |
| 2 | Instr[31:0] | O | 现在执行de机器码 |



reg [31:0] ROM [0:4095];

initial begin

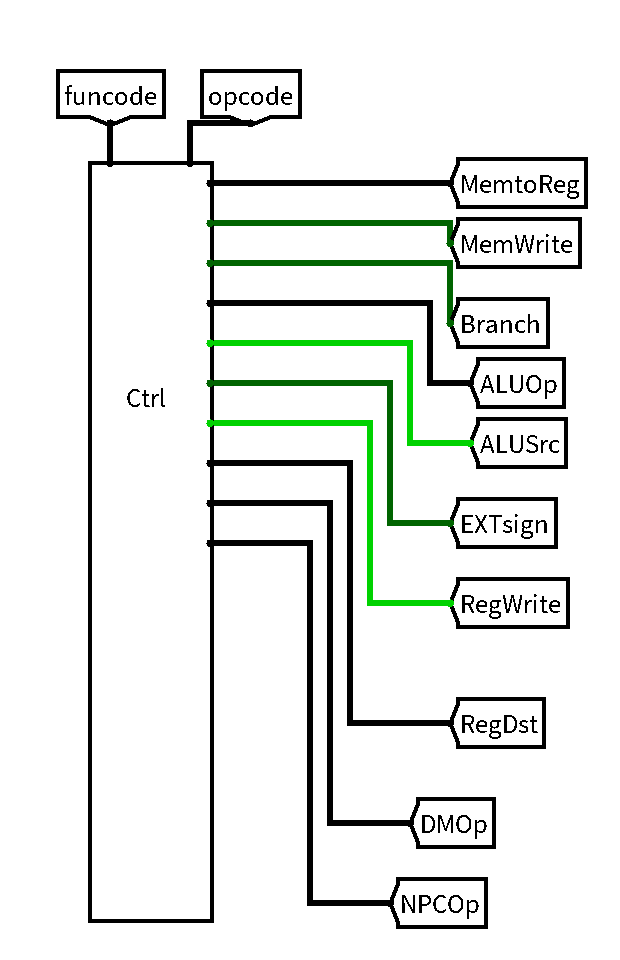
$readmemh("code.txt", ROM , 0);

end

一定得是0：4095（ISE）

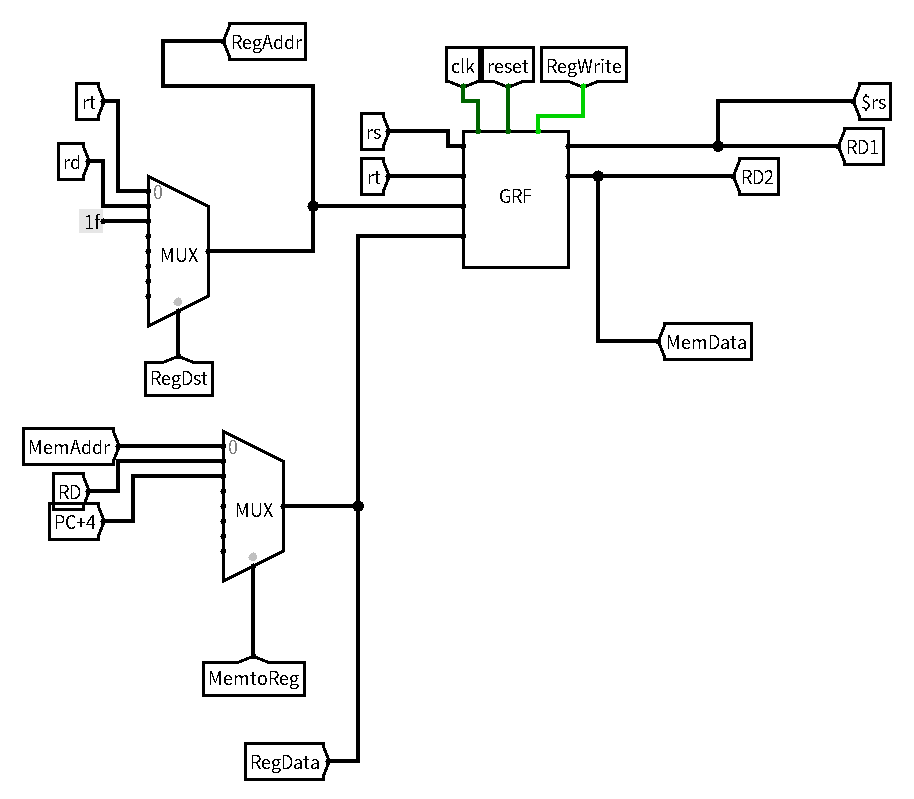
### Controller

| 序号 | 信号名 | 方向 | 描述 |
| --- | --- | --- | --- |
| 1 | instr[31:0] | I | Instr 32位控制信号 |
| 2 | MemtoReg[2:0] | O | 写入GRF的数据的选择信号：  Reg\_rd:000：MemAddr / ALUresult  Reg\_rd:001：RD  Reg\_pcplus4:010:PC+4(jal、jalr) |
| 3 | MemWrite | O | DM的写入使能信号  0：禁止  1：允许（sh、b、w） |
| 4 | ALUOp[2:0] | O | ALU控制信号  ALU\_add:000：无符号加  ALU\_sub:001：无符号减  ALU\_ori:010：或立即数  ALU\_lui:011：lui |
| 5 | ALUSrc[2:0] | O | 参与ALU运算的第二个数  ALUSrc\_Grt:000：MemData  ALUSrc\_imm32:001：imm32（s、l、ori、lui） |
| 6 | Extsign | O | imm16无符号或符号扩展的选择信号  0：无符号扩展  1：符号扩展（l、s、beq） |
| 7 | RegWrite | O | GRF写使能信号  0:不能向GRF 中写入数据  1:能向GRF 中写入数据（lwhb、add、sub、ori、lui、jalr、jr） |
| 8 | RegDst[2:0] | O | GRF写入地址选择信号  RegDst\_rt:000：rt  RegDst\_rd:001：rd（R型add、sub、jalr）  RegDst\_31:010：$ra（jal） |
| 9 | DMOp[2:0] | O | 向内存中读写方式  DM\_word:000：lw、sw  DM\_halfword:001：lh、sh  DM\_byte:010：lb、sb |
| 10 | NPCOp[2:0] | O | NPC将会是  NPC\_else:000：PC+4  NPC\_beq:001：PCbranch（beq）  NPC\_j\_jal:010：PC31..28instr\_index02 PCJ(al)(jal、j)  NPC\_jr\_jalr:011：GPR[rs] PCJ(al)r(jalr、jr) |



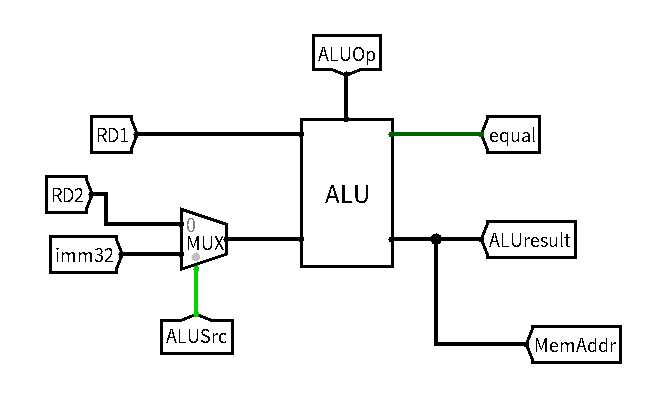
### GRF

| 序号 | 信号名 | 方向 | 描述 |
| --- | --- | --- | --- |
| 1 | pc[31:0] | I | pc 32位地址 |
| 2 | clk | I | 时钟信号 |
| 3 | reset | I | 同步复位信号，将寄存器置为**0**；高电平有效 |
| 4 | RegWrite | I | 写使能信号  1:可向 GRF 中写入数据  0:不能向GRF 中写入数据 |
| 5 | A1[4:0] | I | rs对应寄存器 |
| 6 | A2[4:0] | I | rt对应寄存器 |
| 7 | A3[4:0]  (RegAddr) | I | RegDst决定:  000：rt  001：rd（R型add、sub、jalr）  010：$ra（jal） |
| 8 | WD[31:0]  (RegData) | O | MemtoReg决定:  000：MemAddr / ALUresult  001：RD(DmData)  010:PC+4(jal、jalr)  32位数据输入信号 |
| 9 | RD1[31:0]  （$rs） | O | 输出A1指定的寄存器中的32位数据 |
| 10 | RD2[31:0]  （MemData） | O | （$rt）  输出A2指定的寄存器中的32位数据 |



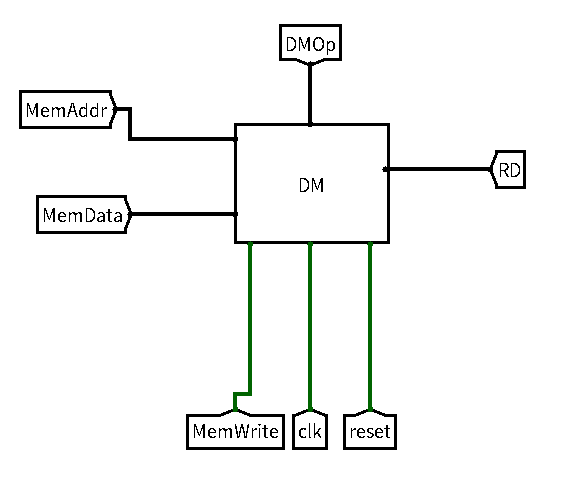
### ALU

| 序号 | 信号名 | 方向 | 描述 |
| --- | --- | --- | --- |
| 1 | A[31:0] | I | RD1  参与运算的第一个数 |
| 2 | B[31:0] | I | ALUSrc：  0：MemData  1：imm32  参与运算的第二个数 |
| 3 | ALUOp[2:0] | I | 000：无符号加  001：无符号减  010：或立即数  011：lui |
| 4 | equal | O | A与B是否相等  0：不相等  1：相等 |
| 5 | ALUresult[31:0]  (MemAddr) | O | MemAddr  A与B做运算后的结果 |



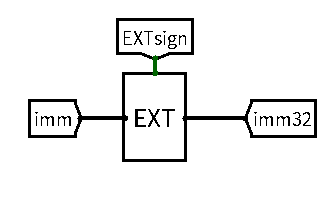
### DM

| 序号 | 信号名 | 方向 | 描述 |
| --- | --- | --- | --- |
| 1 | pc[31:0] | I | pc 32位地址信号 |
| 2 | clk | I | 时钟信号 |
| 3 | reset | I | 同步复位到0  高电平有效 |
| 4 | MemWrite | I | DM的写入使能信号  0：禁止  1：允许（sh、b、w） |
| 5 | MemAddr[31:0] | I | ALUresult |
| 6 | DMOp[2:0] | I | 向内存中读写方式  000：lw、sw  001：lh、sh  010：lb、sb |
| 7 | din[31:0] | I | DM写入32位数据 |
| 8 | dout[32:0] | O | DM读出32位数据 |



### EXT

| 序号 | 信号名 | 方向 | 描述 |
| --- | --- | --- | --- |
| 1 | imm[15:0] | I | 原16位的立即数信号 |
| 2 | EXTsign | I | 无符号或符号扩展的选择信号  0：无符号扩展  1：符号扩展 |
| 3 | imm32[31:0] | O | 扩展后的32位的立即数信号 |



### MUX\_8\_32 与 MUX\_8\_5

| 序号 | 信号名 | 方向 | 描述 |
| --- | --- | --- | --- |
| 1 | Op[2:0] | I | 选择信号 |
| 2 | data1[31:0] | I | 32位信号1 |
| 3 | data2[31:0] | I | 32位信号2 |
| 4 | data3[31:0] | I | 32位信号3 |
| 5 | data4[31:0] | I | 32位信号4 |
| ... | data?[31:0] | I | 32位信号？ |
| 6 | out[31:0] | O | 选择的32位信号 |
| 序号 | 信号名 | 方向 | 描述 |
| 1 | Op[2:0] | I | 选择信号 |
| 2 | data1[4:0] | I | 5位信号1 |
| 3 | data2[4:0] | I | 5位信号2 |
| 4 | data3[4:0] | I | 5位信号3 |
| 5 | data4[4:0] | I | 5位信号4 |
| ... | data?[4:0] | I | 5位信号？ |
| 6 | out[4:0] | O | 选择的5位信号 |

实例化的时候 如果数据不够

.data?(32’b0)

.data?(5’b0)

## 测试方案

**code.txt 一定要放在dictionary下面！！！！！！！！！！**

**1.用我自己生成的数据对拍**

考虑极端数据值

跳转，且目标在此跳转指令之前

跳转，且目标是此跳转指令

跳转，且目标在此跳转指令之后

Offset base 的 正负零情况等

**2.上次的测试代码 这次还能继续用[^]-[^]**

### 测试代码1

#ori

ori $a0, $0, 1

ori $a0,$a0,123

ori $a1, $0, 65535

ori $0,$0,1

#lui

lui $t1,65535

lui $t2, 123

lui $t3, 0xffff

lui $0,65535

#add

add $s0,$t1,$a1 #++

add $s1,$a0,$t3 #+-

add $s2,$t3,$t3 #--

#sub

sub $t4,$a1,$t2 #++ big small

sub $t5,$t2,$a1 #++ small big

sub $t6,$a1,$t3 #+-

sub $t7,$t3,$a1 #-+

sub $t8,$t1,$t3 #--

#sw

ori $a2, $0, 16

sw $t3,4($a2) #base+ offset+

sw $t4,0($a2) #base+ offset0

sw $t5,-4($a2) #base+ offset-

sw $t6,4($0) #base0 offset+

sw $t6,0($0) #base0 offset0

sub $a3,$a3,$a2

sw $t3,20($a2) #base- offset+

#lw

lw $t0,4($a2) #base+ offset+

lw $t1,0($a2) #base+ offset0

lw $t2,-4($a2) #base+ offset-

lw $t3,4($0) #base0 offset+

lw $t4,0($0) #base0 offset

lw $t5,20($a2) #base- offset+

#lh

lh $t2,4($a2)#base+ offset+

lh $t3,0($a2)#base+ offset0

lh $t4,-4($a2)#base+ offset-

lh $t7,4($0)#base0 offset+

lh $t6,0($0)#base0 offset

lh $t1,20($a2)#base- offset+

#sb

sb $t2,4($a2)#base+ offset+

sb $t3,0($a2)#base+ offset0

sb $t4,-4($a2)#base+ offset-

sb $t7,4($0)#base0 offset+

sb $t6,0($0)#base0 offset

sb $t1,20($a2)#base- offset+

#lb

lb $t1,4($a2)#base+ offset+

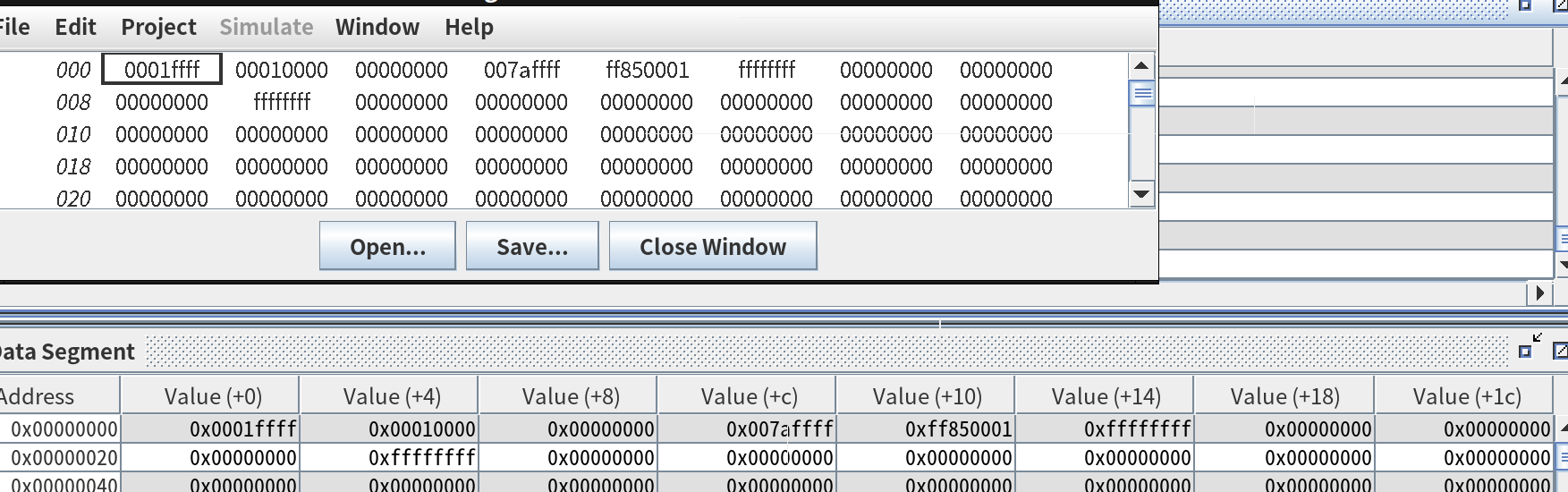
lb $t2,0($a2)#base+ offset0

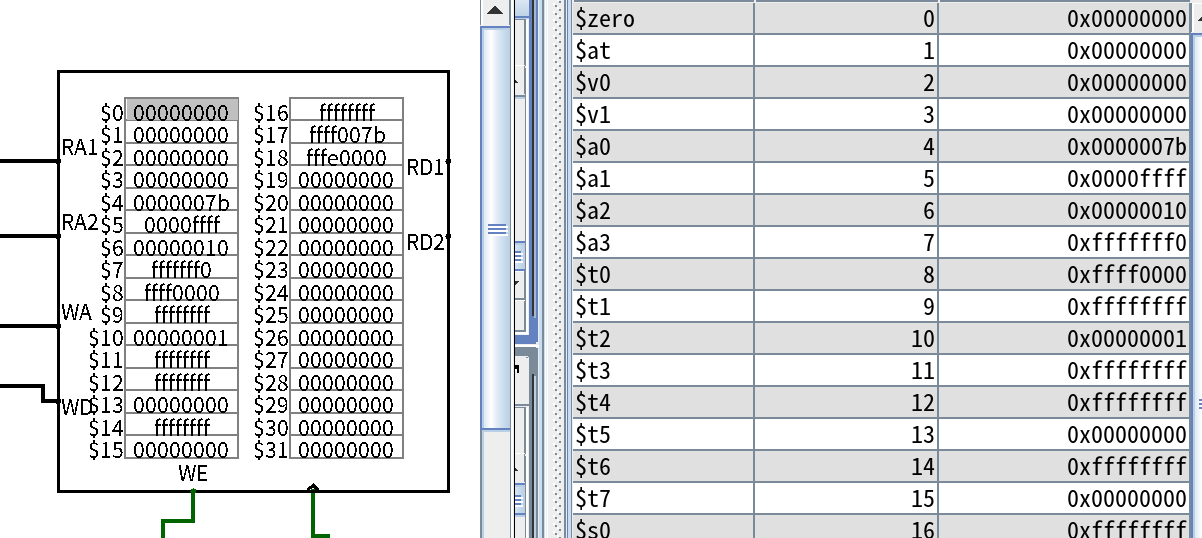
lb $t3,-4($a2)#base+ offset-

lb $t5,4($0)#base0 offset+

lb $t3,0($0)#base0 offset

lb $t4,20($a2)#base- offset+





### 测试代码2

#beq

ori $t0,1

ori $t1,2

add $a0,$0,$t0

add $a1,$0,$t1

beq1:

add $a0,$a0,$t0

beq $a0, $a1, beq1 #this one

beq $a0,$a0,beq2\_true #after

beq2\_true:

sub $a0,$a0,$t0

beq $a0,$a1,beq2\_true #before

#j

sub $a0,$a0,$t0

j\_begin:

beq $a0,$a1,j\_end

add $a0,$a0,$t0

j j\_begin

j\_end:

#jal、j的测试代码自动生成

## **思考题汇总**

**1.阅读下面给出的 DM 的输入示例中（示例 DM 容量为 4KB，即 32bit × 1024字），根据你的理解回答，这个 addr 信号又是从哪里来的？地址信号 addr 位数为什么是 [11:2] 而不是 [9:0] ？**



（1）目前都是ALUresult 因为现在我涉及到的指令的地址都是offset+GRF（base）的运算结果

（2）因为DM容量4KB且DM中是按字节寻址，所以我们的addr是4的倍数，需要取[11:2]位

**2.思考上述两种控制器设计的译码方式，给出代码示例，并尝试对比各方式的优劣。**

（1）指令对应的控制信号如何取值

assign DMOp = (is\_lw || is\_sw )?`DM\_word:

(is\_lh || is\_sh )?`DM\_halfword:

(is\_lb || is\_sb )?`DM\_byte:

`DM\_word;

assign NPCOp = (is\_beq )?`NPC\_beq:

(is\_j || is\_jal )?`NPC\_j\_jal:

(is\_jr || is\_jalr )?`NPC\_jr\_jalr:

`NPC\_else;

…………

（2）记录下控制信号每种取值所对应的指令

always@(\*)begin

if(is\_lw)begin

DMOp = `DM\_word;

RegWrite = 1;

……

end

else if(is\_lh)begin

DMOp = `DM\_halfword;

RegWrite = 1;

……

end

…………

end

（1）优：易拓展指令，能清晰记录每个控制信号的不同取值情况

劣：易遗漏指令

（2）优：所有指令不重不漏，记录控制信号每种取值所对应的指令

劣：容易遗漏对应指令的信号！

**3.在相应的部件中，复位信号的设计都是同步复位，这与 P3 中的设计要求不同。请对比同步复位与异步复位这两种方式的 reset 信号与 clk 信号优先级的关系。**

同步复位中clk优先级比reset高，因为只有时钟上升沿，才判断是否reset高电平复位。

异步复位中reset优先级比clk高，只要reset为高电平，无论是否时钟上升沿，均需进行复位。

**4.C 语言是一种弱类型程序设计语言。C 语言中不对计算结果溢出进行处理，这意味着 C 语言要求程序员必须很清楚计算结果是否会导致溢出。因此，如果仅仅支持 C 语言，MIPS 指令的所有计算指令均可以忽略溢出。 请说明为什么在忽略溢出的前提下，addi 与 addiu 是等价的，add 与 addu 是等价的。提示：阅读《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》中相关指令的 Operation 部分。**

P34、36中 ADD、ADDI指令如是说道：

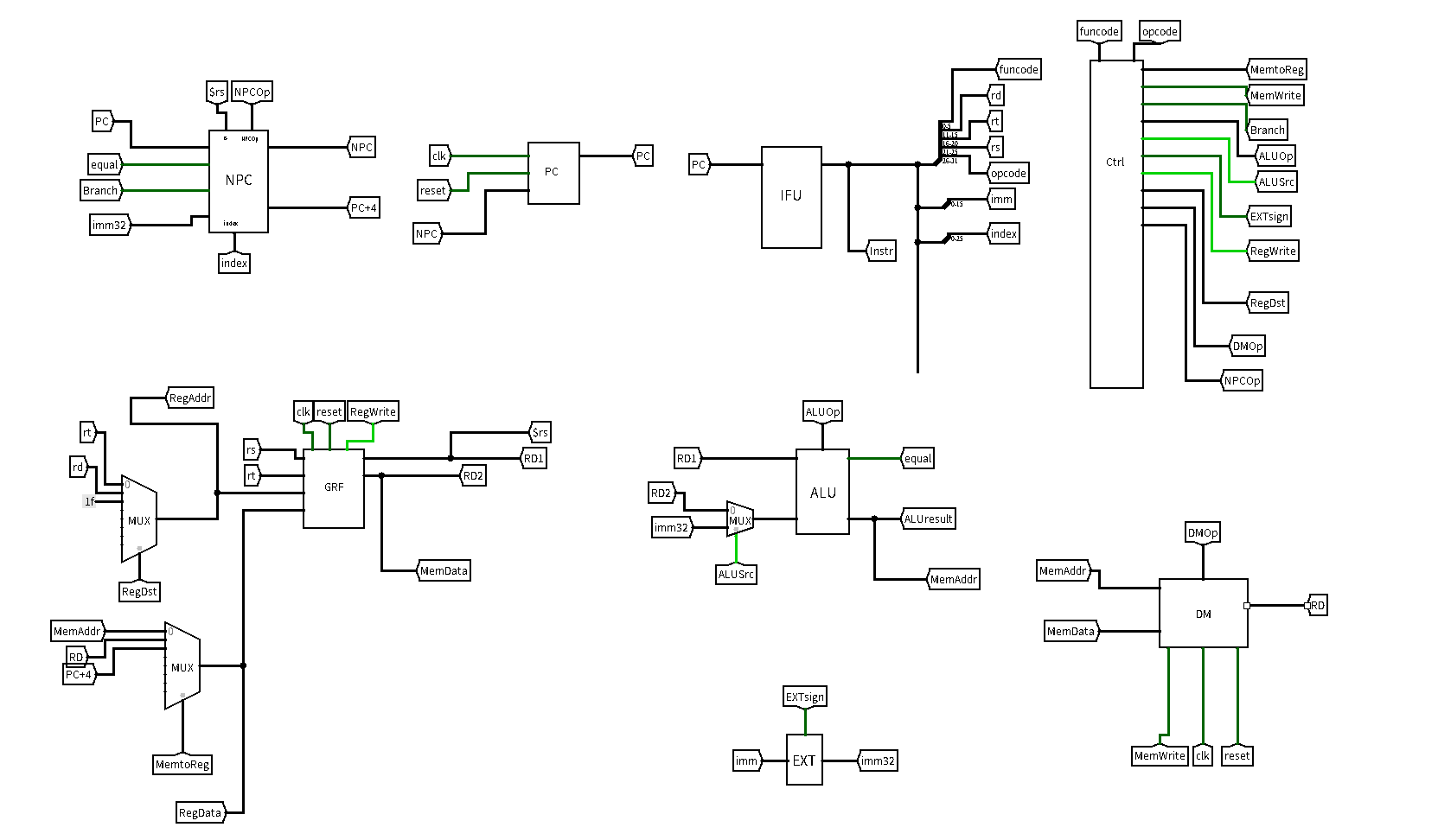
“ADDU/ADDIU performs the same arithmetic operation but does not trap on overflow.”

观察RTL描述，二者区别仅在溢出时，add/addi会SignalException(IntegerOverflow)，而addu/addiu会忽略。

故忽略溢出的前提下，addi 与 addiu 是等价的，add 与 addu 是等价的

## 设计草稿

### 模块化



### 翻译

采用类比的思想

从指令集中找相似的

记得查opcode funcode有无冲突

记得看输出的格式和东西