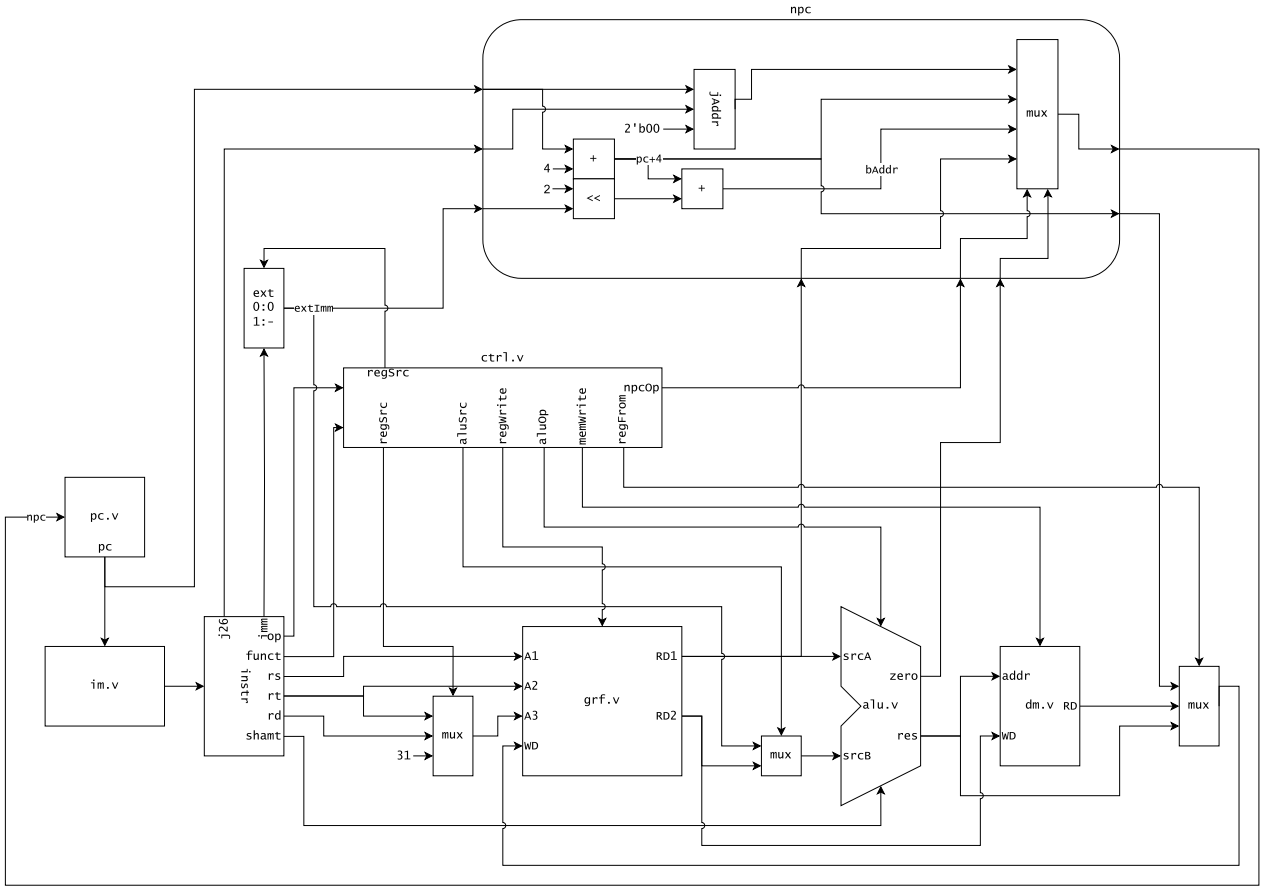
Verilog 单周期 CPU 设计文档

数据通路结构图



接口设计

mips.v 顶层模块

Sig	Type	Descript.
clk	I	时钟信号
reset	I	同步复位

这次设计和 Logisim 不同的是，将和各个模块封装的组合逻辑提取到顶层设计中，使得完整的数据通路明面存在于顶层设计文件里，修改更加方便。将 EXT 和 NPC 模块拆成了数据通路放到顶层设计中。

pc.v 程序计数器

Sig	Type	Descript.
clk	I	时钟信号
reset	I	同步复位
npc	I[31:0]	下一指令的地址

Sig	Type	Descript.
pc	O[31:0]	当前指令的地址

内部定义一个 reg 型变量 addr。

时钟上升沿到来时，若 reset 为 1，则将 addr 置为初始地址 0x3000。

从 Logisim 角度看，pc + npc 部分只有一个寄存器，若在 pc 模块内定义了 reg 型变量，其他地方（包括接口设计）就不要定义 reg 型变量了。

事实上，我只在 pc，im，grf，dm 内部各自定义了需要的 reg 型变量，并且对他们进行了初始化。其他模块以及顶层模块内都是 wire 型变量。

（未尝试）也许可以在顶层设计中定义 reg 型变量，通过参数传入各个模块进行修改。这种思路暴露寄存器在顶层模块中，不确定是否更好。

im.v 指令存储器

Sig	Type	Descript.
pc	I[31:0]	待取指令的地址
instr	O[31:0]	指令

内部用 reg 型实现了一个 ROM，并且在 initial 模块中读取 code.txt 的内容。

访问 ROM 的地址与 Logisim 实现相同，为 $(pc-0x3000)_{13:2}$。

grf.v 寄存器堆

Sig	Type	Descript.
clk	I	时钟信号
reset	I	同步复位
regWrite	I	写使能
readAddr1	I[4:0]	读寄存器 1 地址
readAddr2	I[4:0]	读寄存器 2 地址
writeAddr	I[4:0]	写寄存器地址
writeData	I[31:0]	写入值
readData1	O[31:0]	读取值 1
readData2	O[31:0]	读取值 2

内部用 reg 型实现了一个寄存器堆，在时钟上升沿到来时，若 reset 为 1，则全部置为 0，否则若写使能为 1，则更新写寄存器的内容。

这里需要实现 0 号寄存器始终为 0 的逻辑，我采用的方法是在每次更新完写寄存器的内容后，立即将 0 号寄存器置为 0。可能不是最好的方法。

在线评测中似乎忽略了 0 号寄存器的输出，通过检测其他寄存器的关联输出是否正确，来检查 0 号寄存器是否始终置 0。

alu.v 算术逻辑单元

Sig	Type	Descript.
aluOp	I[3:0]	选择运算类型
shamt	I[4:0]	移位量
srcA	I[31:0]	寄存器值 1
srcB	I[31:0]	运算值 2
zero	O	结果是否为 0
aluRes	O[31:0]	运算结果

使用 wire 型来连接各部分运算逻辑，无法使用 case 而只能使用三目运算符，让代码看起来有些不美观。

ALU 编码表

aluOp	功能
000	A << B(shamt)
001	A OR B
010	A + B
011	{B[15:0], 16'b0}
100	A - B
101	
110	A ^ B
111	

dm.v 数据存储器

Sig	Type	Descript.
clk	I	内置时钟
reset	I	异步复位
memWrite	I	写使能
memAddr	I[31:0]	地址
writeData	I[31:0]	写入值
readData	O[31:0]	读取值

ctrl.v 控制单元

Sig	Type	Descript.
opcode	I[5:0]	opcode 字段
funct	I[5:0]	funct 字段
ext	O	符号扩展信号，在顶层设计中，选择将 imm16 符号扩展/零扩展
aluSrc	O	I 型指令用，选择立即数作为 srcB 进行计算
memWrite	O	Store 类指令用，将寄存器中的值赋给内存
regWrite	O	寄存器堆写使能
regDst	O[1:0]	写寄存器地址选择，0 表示 rt，1 表示 rd，2 表示 $31
regFrom	O[1:0]	写寄存器数据选择，0 表示从 alu 获取，1 表示获取内存，2 表示获取 pc + 4
npcOp	O[2:0]	计算下一指令的地址，0 表示 pc + 4，1 表示 B 类指令，2 表示 J 类指令，3 表示 JR 类指令
aluOp	O[3:0]	运算类型选择

编码

控制单元编码表

Op	OpCode/funct	ext	aluSrc	memWrite	regWrite	regDst	regFrom	npcOp	aluOp
R-Type	000000				1	1			
add	100000								2
sub	100010								4
xor	100110								5
jr	001000							3	
j	000010							2	
jal	000011				1	2	2	2	
beq	000100	1						1	4
ori	001101		1		1				1
lui	001111		1		1				3
lw	100011	1	1		1		1		2
sw	101011	1	1	1					2

测试用例

```
ori $28, $0, 0
ori $29, $0, 0
ori $1, $0, 13398
add $1, $1, $1
lw $1, 4($0)
sw $1, 4($0)
lui $2, 30840
sub $3, $2, $1
lui $5, 4660
ori $4, $0, 5
nop
sw $5, -1($4)
lw $3, -1($4)
beq $3, $5, label_3044
nop
beq $0, $0, label_3084
nop
label_3044:
ori $7, $3, 1028
beq $7, $3, label_3084
nop
lui $8, 30583
ori $8, $8, 65535
sub $0, $0, $8
ori $0, $0, 4352
add $10, $7, $6
ori $8, $0, 0
ori $9, $0, 1
ori $10, $0, 1
label_3070:
add $8, $8, $10
beq $8, $9, label_3070
jal label_3088
nop
add $10, $10, $10
label_3084:
beq $0, $0, label_3084
label_3088:
add $10, $10, $10
jr $31
nop
```

思考题

- 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]？
    - addr 信号由 alu 计算得到，原本是一个 32 位数，转为地址则取 10 位。选取位数 [11:2] 而不是 [9:0]，可以由外部 32 位信号直接获取第 2 至 11 位的部分，因为访问地址是字对齐，所以从第 2 位开始取。

文件	模块接口定义
dm.v	dm(clk, reset, MemWrite, addr, din, dout); input clk; input reset; input MemWrite; input [11:2] addr; input [31:0] din; output [31:0] dout;

- 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。
    - 以 aluOp 为例，第一种方式（指令对应的控制信号如何取值）是对控制信号各位，用或逻辑将各个指令组合起来。比如 aluOp[0] = a | b | c; 表示该信号第 0 位当 a、b、c 信号中有高电平时为 1，不关心其余信号。第二种方式（控制信号每种取值所对应的指令）是判断当前指令是否为指定指令，直接将 aluOp 赋值为对应值。比如 if(opcode == a) aluOp = 3;。
    - 第二种整体赋值较为方便，添加信号的时候不容易漏掉；第一种写法更加简单，和 Logisim 的与或逻辑类似。
- 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：clk 优先级高，当 clk 上升沿时才判断 reset 信号是否为高电平。
    - 异步复位：reset 优先级高，二者中至少有一个处于上升沿时就判断 reset 信号是否为高电平。
- 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。
    - 查询 MIPS-C 指令集可知，add 和 addu 的计算过程是一样的，add 比 addu 多了一步判断溢出的过程。如果不考虑溢出，二者操作上等价。addi 与 addiu 同理。