P4 - Verilog单周期CPU - 设计文档

要求实现一个支持`add, sub, ori, lw, sw, beq, lui, nop, jal, jr`的单周期CPU。

本次可以当作P3的小迭代开发，整体框架与P3基本一致，只不过需要对NPC、GRF写入部分的输入稍加修改。

此外，此处的复位方式为**同步复位**，而非P3中的异步复位。

设计草稿

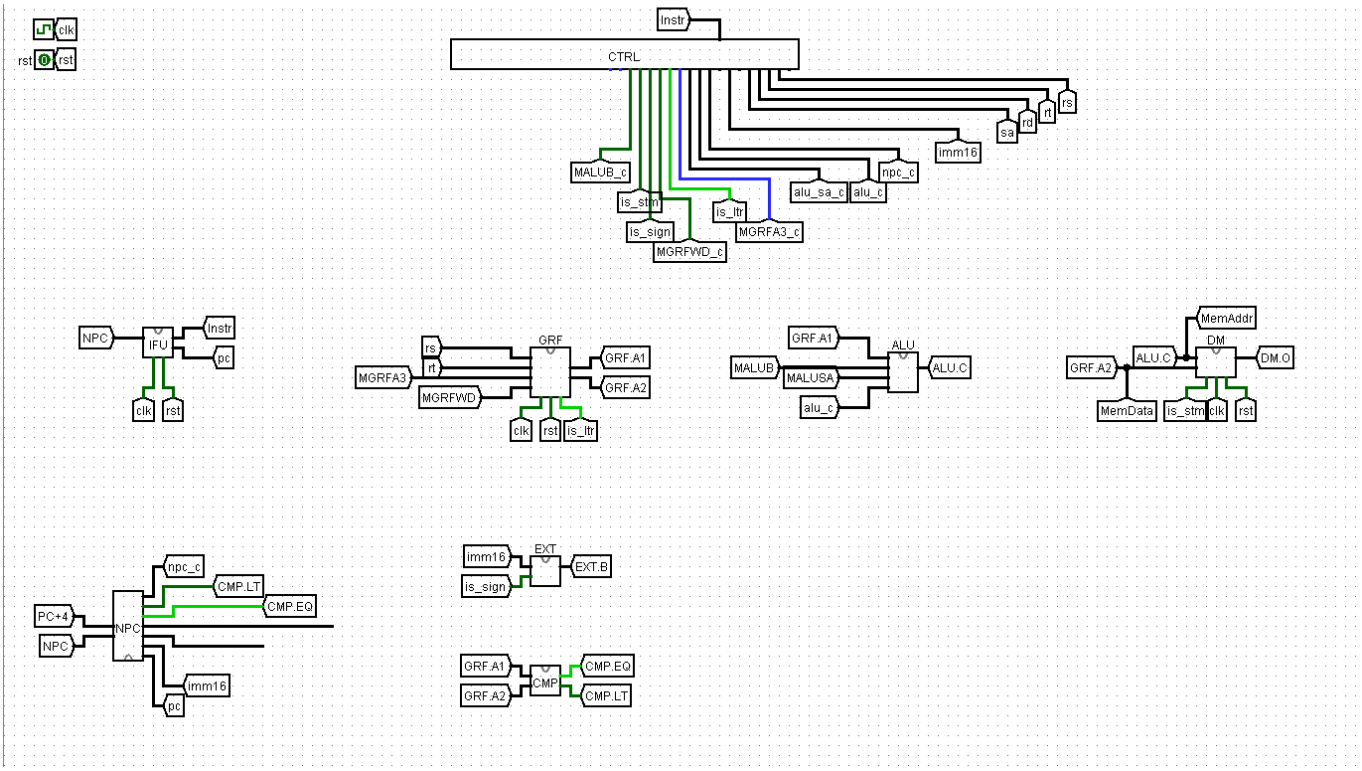
命名规则

出于工程化开发的考量，我们此时进行命名规则的规定：

- 模块：模块取名全大写，实例化时首字母大写，其余部分小写。如`ALU Alu()`
- 控制信号取值：仅在使能信号处直接使用0或1，其余全部使用宏封装；宏名称满足`<module>_[<ctrl_name>_]<operation>`或`<mux>_<signal_name>`的命名格式，且控制信号的命名全使用小写字符。如`npc_op_norm`与`alu_add`
- `CTRL`的控制信号输出端口：全小写命名，遵循`<module/mux_name>_<ctrl_name>`的形式，如`malub_c`，`npc_cond`。此外，`CTRL`的其它输出端口也是全小写命名。
- `wire`型变量：前序全部添加下划线以作区分，如`_wire`。若为控制信号，则命名为全小写。
- 模块端口：全大写。
- 模块的控制信号端口：`<module_name><ctrl_name>`，其中`<ctrl_name>`首字母大写，如`ALUOp`
- 接入模块端口的`wire`：`_<module_name>_<socket_name>`。如`_ALU_A`与`_ALU_Op`。
- 上一条规则对于`CTRL`需要稍加修改。若为控制信号，则为`_CTRL_<module_name>_<ctrl_name>`，如一根接上`CTRL`输出的线：`_CTRL_ALU_Op`

整体架构

图中Logisim结构仅作结构示意，不代表最终实现。



与P3时完成的架构近乎一致。

在对流水线相关知识继续学习后，决定将进行比较操作的 `CMP` 模块移到 `GRF` 所在的D级，与P3中ALU与CMP同处于E级的设计不同。

- C(Command)级： `IFU`，`NPC`
- D(Data)级： `CTRL`，`GRF`，`EXT`，`CMP`
- E(Execute)级： `ALU`
- M(Memory)级： `DM`
- W(Write_reg)级: `GRF的A3, WE, WD部分`； 显然我们现在还没有分这个层的必要，故直接接上层次2里的GRF了。

模块规格

主要是对NPC进行完善，并在控制模块中给出新的控制信号。

其他较P3中设计并无变化，故不加详细说明，直接复制P3中端口定义与控制信号。

控制模块同样单独给出。

NPC

计算下一PC值。

将NPC的跳转操作分为 `norm,imm16,imm26,imm32` 四种情况，分别对应正常 `PC+4`， `branch` 类指令， `j` 类指令，以及 `jr` 一类直接将PC赋为指定32位值的指令。

对于`branch`类操作，考虑到`branch`类指令有多种比较条件，故这里引入`NPCCond`端口，指明B类指令的比较条件。

根据比较结果输入`LT,EQ`进行是否符合跳转条件的判断。若符合，则按B类指令规则进行跳转，反之则执行正常的PC+4操作。

J类指令由于是无条件跳转，故不需对比较条件进行判断。

此处不引入`PC+4`输出。对于`jal`指令，PC+4（不考虑延迟槽，考虑则应该是+8）的PC来源为IFU输出的PC。

端口定义

端口名	方向	说明
PC[31:0]	I	读入当前PC值
IMM16[15:0]	I	传入`branch`类指令的立即数
IMM26[25:0]	I	传入J型指令的26位立即数
IMM32[31:0]	I	传入`jal`等指令需要的32位立即数
EQ	I	来自CMP的比较结果
LT	I	来自CMP的比较结果
NPCOp[2:0]	I	控制信号
NPCCond[2:0]	I	指示B类指令的比较条件
NPC[31:0]	O	计算出的NPC值
clk	I	时钟信号
rst	I	**同步**复位信号

控制信号

NPCOp	3'b000	3‘b001	3‘b010	3'b011
功能	PC+4	branch	j26	j32
说明	正常操作	执行`branch`类指令	执行J型指令	执行`jal`类32位立即数指令

NPCCond	3'b000	3'b001	3'b010	3'b011	3'b100
功能	EQ	LT	GT	LE	GE
说明	==	<	>	<=	>=

IFU

仍然采用了P3中计算相对PC的取指令方法。后续如有这方面的扩展，需要留意。

端口定义

端口/位宽	方向	说明
NPC[31:0]	I	输入NPC单元计算得出的NPC值
INS[31:0]	O	读出PC指向的IM地址之指令
PC[31:0]	O	传出当前PC值
clk	I	时钟信号
reset	I	**同步**复位

CMP

进行**有符号**比较。

端口定义

名称/位宽	方向	说明
A[31:0]	I	输入RS
B[31:0]	I	输入RT或IMM
EQ	O	(A==B)
LT	O	(A < B)

EXT

端口定义

端口/位宽	方向	说明
A[15:0]	I	输入的16位立即数
EXTOp	I	指示拓展方式
B[31:0]	O	扩展结果

控制信号

ext_signed	1'b0	1'b1
功能	ext_ze	ext_se
说明	零扩展	符号扩展

GRF

端口定义

端口/位宽	方向	说明
A1[4:0]	I	要读取的$[rs]之地址
A2[4:0]	I	要读取的$[rt]之地址
A3[4:0]	I	要写入的$[rd]之地址
WD[31:0]	I	写入的数据
We	I	写使能信号
RD1[31:0]	O	读取的$[rs]
RD2[31:0]	O	读取的$[rt]
rst	I	**同**步复位信号
clk	I	时钟信号

ALU

端口定义

名称/位宽	方向	说明
A[31:0]	I	输入RS
ALUOp[3:0]	I	ALU控制信号
B[31:0]	I	输入RT或IMM
SHAMT[4:0]	I	输入移位位数
C[31:0]	O	输出结果

控制信号

ALUOp	4'b0000	4'b0001	4'b0010	4'b0011
功能	alu_add	alu_sub	alu_or	alu_sll
说明	加运算	减运算	或运算	左移运算

DM

端口定义

端口/位宽	方向	说明
ADDR[31:0]	I	DM地址
WD[31:0]	I	待写入的数据
We	I	写使能信号
clk	I	时钟信号
rst	I	**同**步复位信号
RD[31:0]	O	读出指定地址的数据

控制模块

具体实现的话，源码较文字描述更为直观，我们这里主要谈搭建思路。

仍然是分为`DEC, OR, AND`三级进行搭建。`DEC`将机器码分解成各信号区块，`OR`进行指令的翻译，`AND`进行控制信号的生成。

在`AND`层级，我们实际上做的**不是生成控制信号，而是生成各控制信号的条件布尔变量。** 在全部条件生成后，我们再通过`assign`语句生成控制信号。

以ALU的控制信号为例：

```
// ALU

wire _alu_sub, _alu_or, _alu_sll; // _alu_add = _add | _lw | _sw   set as
default

assign _alu_sub = _sub,

           _alu_or = _ori,
```

```
            _alu_sll = _sll | _lui;

// Signal

assign alu_op = (_alu_sub) ? `alu_sub :

                (_alu_or) ? `alu_or :

                (_alu_sll) ? `alu_sll : `alu_add;
```

由于正则表达式进行条件匹配的格式限制，我们必须留一个情况作默认值。此处ALU的默认控制信号取值就是`alu_add`。

数据通路与MUX

数据通路

对于IFU输入必来自NPC此类的显然连接，我们在表中略去,在下单独给出。

IFU.NPC = NPC.NPC
CTRL.INS = IFU.INS
NPC.PC = IFU.PC
ALU.A = GRF.RD1
CMP.A1 = GRF.RD1，CMP.A2=GRF.RD2

输入/指令	add	sub	sll(nop)	ori	lw	sw	b
NPC.NPCOp	npc_norm	npc_norm	npc_norm	npc_norm	npc_norm	npc_norm	n
NPC.Cond	/	/	/	/	/	/	E
NPC.IMM32	/	/	/	/	/	/	/
GRF.A3	CTRL.rd	CTRL.rd	CTRL.rd	CTRL.rt	CTRL.rt	/	/
GRF.We	1	1	1	1	1	0	0
GRF.WD	ALU.C	ALU.C	ALU.C	ALU.C	DM.RD	/	/
EXT.EXTOp	/	/	/	0	1	1	/
ALU.ALUOp	alu_add	alu_sub	alu_sll	alu_ori	alu_add	alu_add	/
ALU.B	GRF.A2	GRF.A2	GRF.A2	EXT.B	EXT.B	EXT.B	/
ALU.SHAMT	/	/	CTRL.shamt	/	/	/	/

输入/指令	add	sub	sll(nop)	ori	lw	sw	b
DM.ADDR	/	/	/	/	ALU.C	ALU.C	/
DM.We	0	0	0	0	0	1	0
DM.WD	/	/	/	/	/	GRF.A2	/

(1):此处的移位方式并不重要，用sign_extend也是可以的...

MUX

MALUB_c	3'b000	3'b001
MALUB	GRF.A2	EXT.B

MALUSA_c	3'b000	3'b001
MALUSA	CTRL.SA	5'd16

MGRFA3_c	3'b000	3'b001	3'b010
MGRFA3	CTRL.rd	CTRL.rt	5'd31

MGRFWD_c	3'b000	3'b001	3'b010
MGRFWD	ALU.C	DM.RD	(IFU.PC+4)

测试方案

在P3的测试程序上，多加了对 `jal,jr` 指令的测试。

```
.text
# 操控$5为0xffff_ffff,$6为0x8000_0000
# 测试$0
ori $0,$0,65535
# 若通过，测试ori；zero_ext
ori $3,$0,4096 # 高第三位为1
ori $3,$3,65534 #最后一位0，高第三位为1
# 若通过，测试lui
lui $3,65535
```

```
lui $0,65535
# 若通过，测试减
ori $4,$0,5 #init
ori $2,$0,1
sub $3,$4,$2 #正常减法
ori $2,$0,6
sub $3,$3,$2 #跨到-1
ori $2,$0,1
sub $3,$6,$2 # -2137483648 - 1，溢出到2137483647
sub $0,$3,$1 # $0
# 若通过，测试加
ori $4,$0,5 #init
ori $2,$0,1
add $3,$4,$2 #正常+法
ori $2,$0,1
sub $3,$0,$2 #$3跨到-1
add $3,$3,$2 #-1跨到0
ori $2,$0,1
add $3,$5,$2 # 2137483647 + 1，溢出到-2137483648
add $0,$3,$2 # $0
# sll
ori $3,$0,65535
sll $3,$3,16
sll $0,$3,1
sll $0,$0,0 #nop，无事发生
# 通过，测试sw
ori $3,$0,1
ori $2,$0,4
sw $3,0($2)
sw $3,4($2)
sw $3,-4($2)
# lw
# 完成上述测试后，0，4，8应存有1
ori $4,$0,1
ori $2,$0,4
lw $3,4($2)
lw $3,0($2)
lw $3,-4($2)
lw $0,0($2) # $0
# beq
```

```
ori $2,1
ori $3,2
ori $4,1
that:
beq $2,$3,that
beq $2,$2,next
ori $2,$0,16
next:
ori $3,$0,16
jal calc
ori $2,$0,255
ori $3,$0,255
li $v0,10
syscall
calc:
ori $2,$0,1
ori $3,$0,1
jr $ra
```

思考题

数据通路设计

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] ？

文件	模块接口定义

dm.v	dm(clk,reset,MemWrite,addr,din,dout);
	input clk; //clock
	input reset; //reset
	input MemWrite; //memory write enable
	input [11:2] addr; //memory's address for write
	input [31:0] din; //write data
	output [31:0] dout; //read data

我们都知道，内存的地址是以**字节**为单位的。这里的`addr`是由ALU算出的，以字节为单位的内存地址。

而我们此时是以**字**为单位进行数据的存储，故我们需要将这一地址转换为以字为单位的地址。很简单，除4即可，这对应就是将字节地址向右移动2位。

我们知道，2^10 = 1024；1024字需要10位的以字为单位的内存地址来表示；这么一来，就得到了选取内存地址的[11:2]作为实际内存信号的结论。

你问为什么不是[9:0]？因为[9:0]表示的是以字节为单位，可表示1024个字节的地址。

这个样例有点短视，以后要加`lb`,`sb`等指令的时候就得重新把低两位加进去。

控制器设计

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。

- 记录下 **指令对应的控制信号如何取值**

```
case(ins)
add:begin
   alu_op<=`alu_add;
   grf_we<=`grf_we;
end
...
endcase
```

这种写法的唯一好处是，可以根据数据通路表直观地写出每条指令对应的控制信号取值。

但是这一写法在根据波形进行debug时会带来诸多不便，这一点我们在后面细说。

- 记录下**控制信号每种取值所对应的指令**
这是本设计采取的方案。

```
wire add;
assign add = !(|opcode) && (funct == 6'b100000);
assign _alu_add = add | ... | ...;
assign alu_op = (_alu_add)? `alu_add :
                                ......;
```

这种写法的好处是，可以直观的看见控制信号取某一值时，对应的有哪些指令。

这为按照波形进行debug提供了充分的便利。

我们至少在通过ISE的ISim查看波形时，并不能直观地通过IFU传出的指令机器码，看出正在执行的指令。能直观看到的，而且评测机里给出的，是PC值。

在查看波形时，我们能看到的是控制信号的取值。如果采用记录控制信号对应指令的办法，**我们能很快由不符预期的控制信号取值，发现是否有指令漏加了控制信号；** 而记录指令对应的控制信号如何取值的方法，则不会这样直观，有可能需要逐条指令进行排查。

至于劣势，则是没法直观地查看一条指令对应的控制信号；考虑到数据通路表的存在，以及控制信号取值一般也有表，这一问题基本可以忽略不计。

测试相关

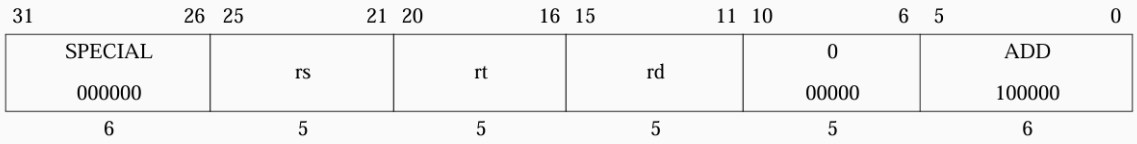
1. 在相应的部件中，复位信号的设计都是**同步复位**，这与 P3 中的设计要求不同。请对比**同步复位**与**异步复位**这两种方式的 reset 信号与 clk 信号优先级的关系。

   在同步复位中，clk信号优先级高于reset信号；仅在clk信号有效时，reset信号有效才会引起复位；

   而在异步复位中，可以认为reset优先级更高。无论clk是否有效，只要reset有效，就触发复位。

   但从常见写法 `always@(posedge clk or posedge reset)` 来看，二者可以认为是平级的。
2. 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。

Add Word **ADD**



Format: `ADD rd, rs, rt` **MIPS32**

Purpose:

To add 32-bit integers. If an overflow occurs, then trap.

Description: `GPR[rd] ← GPR[rs] + GPR[rt]`

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* to produce a 32-bit result.

- If the addition results in 32-bit 2's complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs.
- If the addition does not overflow, the 32-bit result is placed into GPR *rd*.

Restrictions:

None

Operation:

```
temp ← (GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp
endif
```

Exceptions:

Integer Overflow

Programming Notes:

ADDU performs the same arithmetic operation but does not trap on overflow.

Programming Notes里已经写了。二者的运算操作相同，只不过 `addu` 没有与溢出相关的 `Exception`。忽略溢出的话，二者显然等价。

`addi/addiu` 同理，不再赘述。