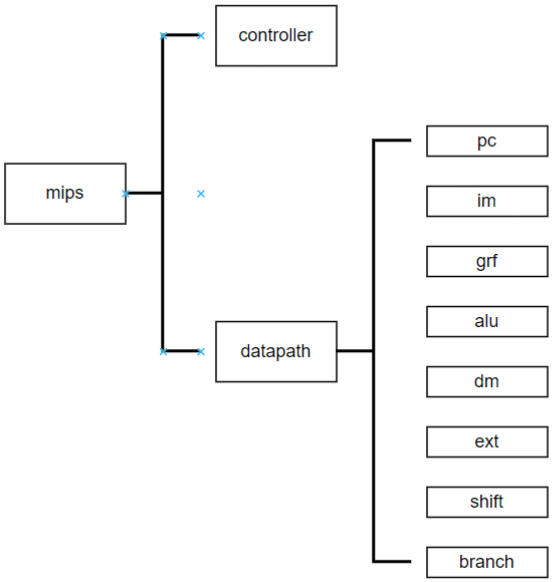
CPU设计文档

构造



具体部件

- **Controller（控制器，根据 splitter 得到的 6 位 funct 码和 6 位 instr_index 码确定指令的类型并输出对应的控制信号）**
    - 控制器的设计，从最基本的层面来说，是一个**译码**的过程，将每一条机器指令中包含的信息，转化为给 CPU 各部分的控制信号
    - assign疯狂赋值
    - 指令的opcode和funct

funct	**100000**	**100010**	---------	---------	---------	---------	---------	---------	**001000**	----------
opcode	000000	000000	001101	100011	101011	000100	001111	000011	000000	----------
	add	sub	ori	lw	sw	beq	lui	jal	jr	nop
Wreg_sel (RegDst)	01(Rd)	01	00(Rt)	00	00	x	00	10(Reg31)	00	x
Wdata_sel (MemtoReg)	00(ALU)	00	00	01(DM)	01	x	10(Shift)	11(Adder)	00	x
W_en (RegWrite)	1	1	1	1	0	0	1	1	0	0
ALUop_sel (ALUOp)	010(加)	110(减)	001(或)	010(加)	010(加)	110(减)	x	x	x	x
ALUin_sel (ALUsrc)	0(Rdata2)	0	1(EXT)	1	1	0	x	x	x	x
DM_sel (MemWrite)	0	0	0	0	1	0	0	x	x	x
Branch(Branch)	0	0	0	0	0	1	0	0	x	x
Jal	0	0	0	0	0	0	0	1	0	x
Jr	0	0	0	0	0	0	0	0	1	x
EXT_sel	x	x	01(符号扩展)	00(0扩展)	00	00	00	00	x	x
Shift_sel	x	x	x	x	x	0	1	x	x	x

- # Datapath
  - PC
    - 初始化，起始地址 0x00003000
    - 同步复位
    - 根据PC_sel确定下一条指令地址

指令	**PC_sel**	**Next_PC**
无	00	PC+4
branch	01	PC + 4 + Shift_out
jal	10	{{PC[31 : 28]}, instruction[25:0], {2'b00}}
jr	11	grf(Registe[31])

  - IM
    - 根据地址输出指令
    - assign
    - 初始化，起始地址 0x00000000
    - 容量为 4096 × 32bit。
  - GRF
    - 寄存器堆
    - 同步复位
    - assign读，always写
    - 初始化

        - 0寄存器特判

    - ALU
        - assign疯狂输出
        - zero判定

    - DM
        - 注意读写判定
        - assign读，always写
        - 同步复位,复位值为 0x00000000
        - 初始化
        - 容量为 3072 × 32bit

    - EXT
        - assign输出
        - 注意拓展类型

    - Shift
        - assign输出

    - branch
        - assign输出
        - 注意各个跳转指令判定

结构设计

模块

- PC
- IM
- GRF：寄存器相关
    - Reg1,Reg2
    - Wreg
    - Wdata
- ALU：运算
- DM：内存
- EXT：位扩展
- Shift：移位
- Controller：控制器

结构图

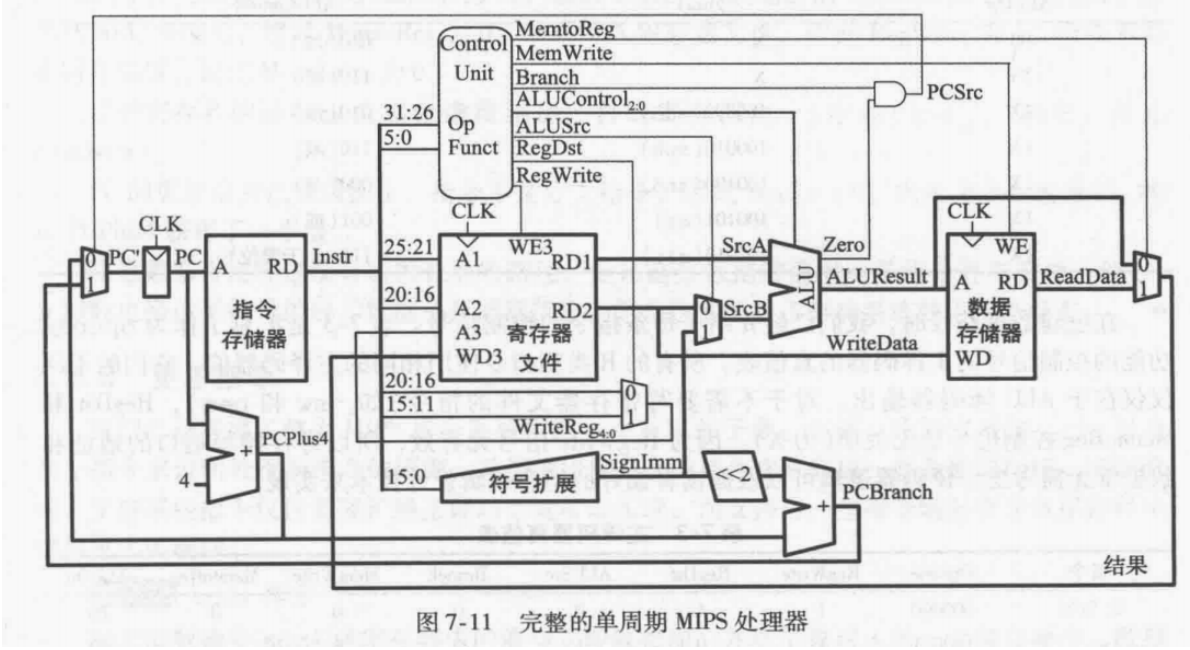


图 7-11　完整的单周期 MIPS 处理器

测试方案

```
ori $28,$0,0
ori $29,$0,0
ori $1,$0,0x3456
add $1,$1,$1
lw $1,4($0)
sw $1,4($0)
lui $2, 0x7878
sub $3,$2,$1
lui $5, 0x1234
ori $4,$0,5
nop
sw $5,-1($4)
lw $3,-1($4)
beq $3, $5, loop1
nop
beq $0, $0, loop2
nop
loop1:
ori $7,$3,0x404
beq $7,$3,loop2
nop
lui $8,0x7777
ori $8,$8,0xffff
sub $0,$0,$8
ori $0,$0,0x1100
add $10,$7,$6
ori $8,$0,0
ori $9,$0,1
ori $10,$0,1
```

```
loop4:
add $8,$8,$10
beq $8,$9,loop4
jal loop5
nop
add $10,$10,$10
loop2:
beq $0,$0,loop2
loop5:
add $10, $10, $10
jr $31
nop
```

思考题

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]？
    - addr信号来自于ALU的输出，无论是从内存中读取还是写入，其地址都要经过ALU的运算。

- 因为mips架构的cpu的指令存储器是按字寻址的，所以用verilog建模IM时也是按字寻址的，[11:2]相当于忽略了最后两位。

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

   第一种方式：

```
always@(*) begin
    if(add) begin
        Wreg_sel <= 2'b01;
        Wdata_sel <= 2'b00;
        W_en <= 1;
        ALUop A= 010;
    end
    else if(sub) begin
       ...
    end
    ...
end
```

第二种方式：

```
assign Wreg_sel = ((opcode == `ALU) && ((funct == `add) || (funct == `sub ))) ?
2'b01 :
                        (opcode == `jal) ? 2'b10 : 2'b00;
assign Wdata_sel = ...
...
```

优劣比较：

- 第一种方式可以更好的记录指令对应的控制信号如何取值，更容易拓展指令，
- 第二种方式可以更好的记录控制信号每种取值所对应的指令,更明确各种控制信号。

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clk信号优先级更高，只有在时钟上升沿到来时，reset信号为高电平才进行复位
    - 异步复位reset信号的优先级更高，只要reset为高电平，无论是否是时钟上升沿，都进行复位
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。

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

    ADDU执行相同的算术运算，但不会在溢出时捕获。

    The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

    指令名称中的术语“无符号”是用词不当;该操作为32位模运算，不会发生溢出告警。该指令适用于无符号算术，如地址算术，或忽略溢出的整数算术环境，如C语言算术。

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

    ADDIU执行相同的算术运算，但不会在溢出时捕获。

    The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

    指令名称中的术语“无符号”是用词不当;该操作为32位模运算，不会发生溢出告警。该指令适用于无符号算术，如地址算术，或忽略溢出的整数算术环境，如C语言算术。

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