

# Verilog 单周期 CPU 设计文档

## 一、CPU 设计方案综述

本 CPU 为 Verilog 实现的单周期 CPU（32 位），支持的指令集包含 {addu, subu, ori, lw, sw, beq, lui, nop}，并进行了适当的指令扩展。

该 CPU 采用层次化，模块化的设计，主要包含 IFU, GRF, EXT, ALU, DM, Controller 等子模块。处理器顶层包含两个输入端口时钟信号 clk 和复位信号 reset。

## 二、关键模块定义

### 1、IFU

#### ① 基本描述

IFU 内部主要包括 PC, NPC, IM（容量 32bit\*1024，起始地址为 0x00003000）以及相关逻辑。在 NPC 中产生下一条指令的地址，当时钟上升沿到来时，PC 更新指令地址并将其输出，IM 根据地址输出对应指令。

#### ② 端口说明

信号名	方向	描述
CLK	I	时钟信号
Reset	I	复位信号
NPCOp[1:0]	I	控制 NPC 进行相应的操作：  00：当前为顺序执行指令，NPC 输出 PC+4

		<p>01: 当前指令为 beq, 作为决定是否跳转的条件之一</p> <p>10: 当前指令为 jal, NPC 输出 PC31..28  instr_index  02</p> <p>11: 当前指令为 jr, NPC 输出 GRF[rs]</p>
RA[31:0]	I	将 GRF[rs]的值输入 IFU
Zero	I	相等标志信号, 判断 ALU 两操作数是否相等
Instr[31:0]	0	根据地址取到的当前指令
PC4[31:0]	0	输出 PC+4 作为地址
PC[31:0]	0	输出当前执行指令的地址

### ③ 功能定义

序号	功能名称	功能描述
1	复位	当复位信号有效时, PC 被设置为 0x00003000
2	取指令	根据当前 PC 的值从 IM 中取出相应的指令并通过 Instr 端口输出
3	输出 PC+4	在 PC4 输出端口输出 PC+4
4	输出 PC	在 PC 端口输出当前执行指令的地址 PC
5	计算 NPC	<p>NPC 根据 NPCOp 取值确定:</p> <p>00: PC+4</p>

		01: $PC+4+sign\_extend(offset \parallel 00)$  (Zero 为真时); $PC+4$ (Zero 为假时)  10: $PC31..28 \parallel instr\_index \parallel 00$  11: $GRF[rs]$
6	更新 PC	当时钟上升沿到来时，更新 PC 为 NPC

2、 GRF

① 基本描述

GRF 模块内部具有 32 个具有写使能和复位功能的寄存器，0 号寄存器内的值始终为 0。GRF 支持同时读取两个寄存器的值以及写入一个寄存器的操作。

② 端口说明

信号名	方向	描述
CLK	I	时钟信号
Reset	I	复位信号
PC	I	当前执行指令的地址
A1[4:0]	I	地址输入信号，将对应地址寄存器的值输出至 RD1
A2[4:0]	I	地址输入信号，将对应地址寄存器的值输出至 RD2

A3[4:0]	I	地址输入信号，指定要进行写入的寄存器
RFWr	I	写使能信号
WD[31:0]	I	要写入寄存器的值
RD1	O	数据输出信号，输出 A1 地址对应的寄存器的值
RD2	O	数据输出信号，输出 A2 地址对应的寄存器的值

③ 功能定义

序号	功能名称	功能描述
1	复位	当复位信号有效时，GRF 中的寄存器全部复位（初值设置为 0x00000000）
2	读取数据	读取 A1 和 A2 地址所对应寄存器的数据至 RD1 和 RD2 输出端口
3	写入数据	当时钟上升沿到来时，如果 RFWr 信号有效，则将 WD 输入端口的数据写入 A3 地址所对应的寄存器中（无视对于 0 号寄存器的写入），并输出相应的写入信息

3、 EXT

① 基本描述

EXT 用于将 16 位立即数进行符号（无符号）扩展成 32 位、将十六位立即数加载到高位并输出。

② 端口说明

信号名	方向	描述
imm16[15:0]	I	数据输入信号，输入要进行扩展的 16 位立即数
EXTOp	I	控制信号：  0：无符号扩展  1：符号扩展
Ext[31:0]	O	数据输出信号，输出扩展完毕的数据
Imm32[31:0]	O	数据输出信号，输出加载到高位的立即数

③ 功能定义

序号	功能名称	功能描述
1	无符号扩展	将 16 位立即数无符号扩展为 32 位
2	符号扩展	将 16 位立即数符号扩展为 32 位
3	立即数加载到高位	将 16 位立即数加载到高位成为 32 位输出数据

4、 ALU

① 基本描述

ALU 对输入的两个操作数（32bit）进行加、减、或、大小比较功能，输出运算的结果以及比较结果。

② 端口说明

信号名	方向	描述
A[31:0]	I	数据输入信号，输入 ALU 的第一个操作数
B[31:0]	I	数据输入信号，输入 ALU 的第二个操作数
ALUOp[1:0]	I	控制信号：  00: A+B  01: A-B  10: A B
C[31:0]	O	数据输出信号，输出 ALU 的计算结果
Zero	O	数据输出信号，输出两操作数进行相等比较的结果

③ 功能定义

序号	功能名称	功能定义
1	加法	将两操作数相加
2	减法	将两操作数相减
3	或运算	将两操作数按位或
4	相等比较	判断两操作数是否相等，相等则 Zero 为真，反之为假

5、 DM

① 基本描述

DM 用于数据存储（容量为 32bit\*1024，起始地址为 0x00000000）。DM 支持复位功能，采用单向双端口设计。每当时钟上升沿到来时，如果写使能有效则能将数据写入对应地址，每时每刻根据地址信号读出相应数据。

② 端口说明

信号名	方向	描述
CLK	I	时钟信号
Reset	I	复位信号
PC	I	当前执行指令的地址
DMWr	I	写使能信号
A[31:0]	I	地址信号，指定要进行操作的存储单元的地址
WD[31:0]	I	数据输入信号，输入要写入存储单元的数据
RD[31:0]	O	数据输出信号，输出地址对应的存储单元的数据

③ 功能定义

序号	功能名称	功能描述
1	复位	当复位信号有效时，每一个存储单元都被复位为 0x00000000
2	读取	根据 A 地址信号输出对应存储单元的数据至 RD 输出端口
3	写入	当时钟上升沿到来时，如果 DMWr 有效，则根据 A 地址信号将 WD 输入端口中的数据写入对应的存储单元，同时输出写入信息

## 6、 DC

### ① 基本描述

DC 用于将 Instr 中的各位按照不同编码规则进行处理，方便之后各个子模块接收相应的指令信息

### ② 端口说明

信号名	方向	描述
Instr	I	数据输入信号，当前执行指令
opcode[5:0]	0	数据输出信号，当前指令的 opcode
funct[5:0]	0	数据输出信号，当前指令的 funct
rs[4:0]	0	数据输出信号，当前指令的 rs
rt[4:0]	0	数据输出信号，当前指令的 rt



rd[4:0]	0	数据输出信号，当前指令的 rd
imm16[15:0]	0	数据输出信号，当前指令的 imm
index26[25:0]	0	数据输出信号，当前指令的 index

③ 功能定义

序号	功能名称	功能描述
1	指令解析	将输入的指令进行处理，输出相应的信息，方便其他子模块接收信息

7、DC

① 基本描述

MUX（多路选择器），当有多个信号同时输入一个端口时需要根据控制信号进行选择，使用时根据输入信号的个数选择不同的 MUX

② 端口说明

信号名	方向	描述
Ini[?:0]	I	第 i 个数据的输入信号
Sel[?:0]	I	数据输出信号，选择相应的输入数据
Out[?:0]	O	数据输出信号，输出对应的数据

③ 功能定义

序号	功能名称	功能描述
1	选择输入信号	MUX 功能部件的集合，根据需要实例化即可

三、 数据通路设计

部 件	IFU	GRF				EXT	ALU		DM	
输 入 信 号	RA	A1	A2	A3	WD	imm	A	B	A	WD
ad du		IFU. Instr[2 5:21]	IFU. Instr[2 0:16]	IFU. Instr[15:11]	ALU. C		GRF. RD1	GRF. RD2		
su bu		IFU. Instr[2 5:21]	IFU. Instr[2 0:16]	IFU. Instr[15:11]	ALU. C		GRF. RD1	GRF. RD2		
or i		IFU. Instr[2 5:21]		IFU. Instr[20:16]	ALU. C	IFU. Instr[15:0]	GRF. RD1	GRF. RD2		
lw		IFU. Instr[2 5:21]		IFU. Instr[20:16]	DM. RD	IFU. Instr[15:0]	GRF. RD1	EXT. Ext	ALU. C	

sw		IFU. Instr[25:21]	IFU. Instr[20:16]			IFU. Instr[15:0]	GRF. RD1	EXT. Ext	ALU. C	GRF. RD2
beq		IFU. Instr[25:21]	IFU. Instr[20:16]				GRF. RD1	GRF. RD2		
lui				IFU. Instr[20:16]	IFU. Instr[15:0]   0 <sup>16</sup>					
jall				0x1f	IFU. PC4					
jr	GRF. RD1	IFU. Instr[25:21]								
no p										
综合	RA	IFU. Instr[25:21]	IFU. Instr[20:16]	IFU. Instr[15:11] IFU. Instr[20:16] 0x1f	ALU. C, DM. RD, IFU. PC4, IFU. Instr[15:0]   0 <sup>16</sup>	IFU. Instr[15:0]	GRF. RD1	GRF. RD2 EXT. Ext	ALU. C	GRF. RD2

输出	0 端口	1 端口	2 端口	3 端口
----	------	------	------	------

GRF. A3	IFU. Instr[15:11]	IFU. Instr[20:16]	0x1f	
GRF. WD	ALU. C	DM. D	IFU. PC4	IFU. Imm32
ALU. B	GRF. RD2	EXT. Ext		

## 四、 控制器设计

### ① 基本思路

通过指令的 opcode 和 funct 产生数据通路所需要的控制信号，具体操作为先通过与阵列得到指令变量，再通过或阵列得到各控制信号的取值。

### ② 真值表

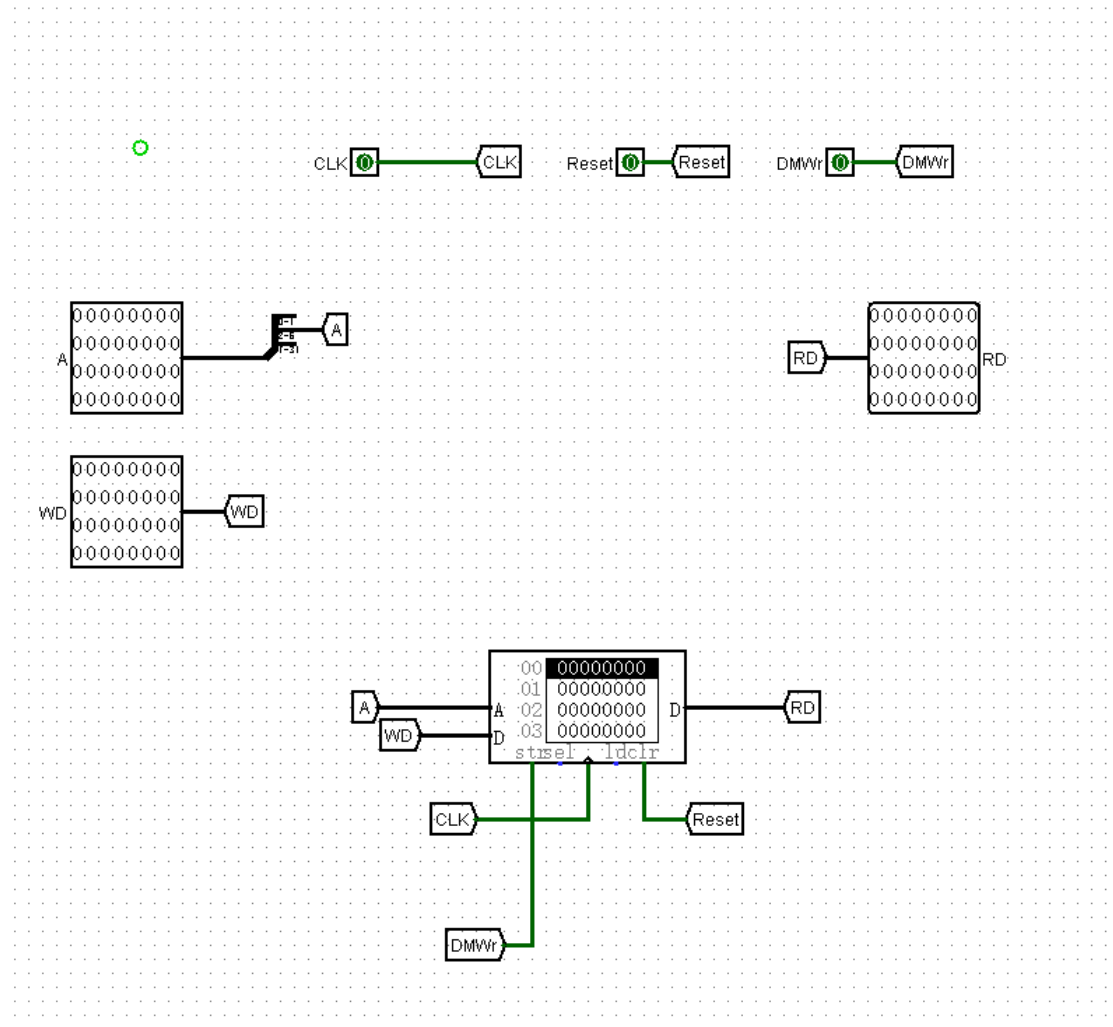
指令	NPCOp[1:0]	GRFWr	EXTOp	ALUOp[1:0]	DMWr	A3Sel[1:0]	WDSel[1:0]	BSel
addu  (000000/100001)	00	1	x	00	0	00	00	0
subu  (000000/100011)	00	1	x	01	0	00	00	0
ori  (001101)	00	1	0	10	0	01	00	1

lw  (100011)	00	1	1	00	0	01	01	1
sw  (101011)	00	0	1	00	1	xx	xx	1
beq  (000100)	01	0	x	01	0	xx	xx	0
jal  (000011)	10	1	x	xx	0	10	10	x
jr  (000000/001000)	11	0	x	xx	0	xx	xx	x
lui  (001111)	00	1	x	xx	0	01	11	x

## 五、 顶层及子模块电路图

**MIPS:**

**DM :**



EXT:





```
.data

a: .word 1:32

.text

ori $t0,$t0,7 #$t0=7

ori $t1,$t1,15 #t1=15

ori $t3,$t3,4 #t3=2

ori $t5,$t5,1 #t5=1

nop

ori $t6,$t6,1

beq $t5,$t6,next

ori $t0,$t0,6

ori $t1,$t1,6

ori $t3,$t3,6

next:

addu $t1,$t0,$t0 #t1=14

subu $t2,$t1,$t0 #t2=7

lw $t4,a($t3)

sw $t1,a($t3)
```

```

jal test

addu $s0,$s0,$t5

subu $s0,$s0,$t5

test:

lui $s0,1

jr $ra

```

## ② 期望结果

```

@00003000: $ 8 <= 00000007
@00003004: $ 9 <= 0000000f
@00003008: $11 <= 00000004
@0000300c: $13 <= 00000001
@00003014: $14 <= 00000001
@00003028: $ 9 <= 0000000e
@0000302c: $10 <= 00000007
@00003030: $12 <= 00000000
@00003034: *00000004 <= 0000000e
@00003038: $31 <= 0000303c
@00003044: $16 <= 00010000
@0000303c: $16 <= 00010001
@00003040: $16 <= 00010000
@00003044: $16 <= 00010000
@0000303c: $16 <= 00010001
@00003040: $16 <= 00010000
@00003044: $16 <= 00010000
@0000303c: $16 <= 00010001
@00003040: $16 <= 00010000
@00003044: $16 <= 00010000

```

(...)

## ③ 测试结果

```

@00003000: $ 8 <= 00000007
@00003004: $ 9 <= 0000000f
@00003008: $11 <= 00000004
@0000300c: $13 <= 00000001
@00003014: $14 <= 00000001
@00003028: $ 9 <= 0000000e
@0000302c: $10 <= 00000007
@00003030: $12 <= 00000000
@00003034: *00000004 <= 0000000e
@00003038: $31 <= 0000303c
@00003044: $16 <= 00010000
@0000303c: $16 <= 00010001
@00003040: $16 <= 00010000
@00003044: $16 <= 00010000
@0000303c: $16 <= 00010001
@00003040: $16 <= 00010000
@00003044: $16 <= 00010000
@0000303c: $16 <= 00010001
@00003040: $16 <= 00010000
@00003044: $16 <= 00010000

```

(...)

## 七、思考题

- ① 根据你的理解，在下面给出的 DM 的输入示例中，地址信号 **addr** 位数为什么是[11:2]而不是[9:0]？这个 **addr** 信号又是从哪里来的？

文件	模块接口定义
dm.v	<pre> 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 </pre>

因为本 CPU 的 DM 的内部是用 1024 个（10bit）32 位寄存器实现的，最小存储单元（32bit），而不是（8bit），所以首地址应该是 4 的倍数，所以应该截取最后两位，所以是[11:2]而不是[9:0]。

这个信号从 ALU 的 C 端口来，在 ALU 中完成了地址的计算。

- ② 思考 Verilog 语言设计控制器的译码方式，给出代码示例，并尝试对比各方式的优劣。

### 1、与或阵

```
`define Rtype 6'b000000
`define ADDU_FC 6'b100001
`define SUBU_FC 6'b100011
`define ORI 6'b001101
`define LW 6'b100011
`define SW 6'b101011
`define BEQ 6'b000100
`define JAL 6'b000011
`define JR 6'b001000
`define LUI 6'b001111

module controller(
    input [5:0] opcode,
    input [5:0] funct,
    output [1:0] NPCOp,
    output GRFWr,
    output EXTOp,
    output [1:0] ALUOp,
    output DMWr,
    output [1:0] A3Sel,
    output [1:0] WDSel,
    output BSel
);
    wire addu,subu,ori,lw,sw,beq,jal,jr,lui;

    assign addu = (opcode == `Rtype)&(funct == `ADDU_FC);
    assign subu = (opcode == `Rtype)&(funct == `SUBU_FC);
    assign ori = (opcode == `ORI);
    assign lw = (opcode == `LW);
    assign sw = (opcode == `SW);
    assign beq = (opcode == `BEQ);
    assign jal = (opcode == `JAL);
    assign jr = (opcode == `Rtype)&(funct == `JR);
    assign lui = (opcode == `LUI);
    //指令变量的与阵列

    assign NPCOp = {jal|jr,beq|jr};
    assign GRFWr = ori|addu|subu|lw|jal|lui;
    assign EXTOp = lw|sw;
    assign ALUOp = {ori,subu|beq};
    assign DMWr = sw;
    assign A3Sel = {jal,ori|lw|lui};
    assign WDSel = {jal|lui,lw|lui};
    assign BSel = ori|lw|sw;
    //控制信号或阵列

endmodule
```

优点：接近底层实现，代码短，之后扩展指令的时候很方便

缺点：不太直观，不能一眼看出当前指令下各个控制信号的取值

## 2、Always case 语句

```
always@(*)begin
    case(opcode)
        `Rtype:begin
            case(funcnt)
                `ADDU:begin
                    NPCOp <= 2'b00;
                    GRFWr <= 0;
                    ALUOp <= 2'b00;
                    DMWr <= 0;
                    A3Sel <= 2'b00;
                    WDSel <= 2'b00;
                    BSel <= 0;
                end
                //subu等R型指令
            endcase
        end
        //I J 型指令
        `ORI:begin
            //...
        end
    endcase
end
```

优点：直观简单，能一眼看出当前指令下的控制信号的取值

缺点：代码量大，需要写很多代码，当需要扩展指令的时候，代码量比较大

## 3、在相应的部件中，**reset** 的优先级比其他控制信号（不包括 **clk** 信号）都要**高**，且相应的设计都是**同步复位**。清零信号 **reset** 所驱动的部件具有什么共同特点？

PC，GRF，DM 等都需要有确定的初始值，若不能复位，这些部件内部存储的值不确定，可能使 CPU 没法正常工作。

## 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。

add 和 addu、addi 和 addiu 的区别仅仅在于溢出时会不会触发异常，C 语言在执行加法时即使溢出也不会触发异常，即 C 语言忽略溢出；MIPS 如果仅支持 C 语言，那么就不用关心异常的触发与否，即忽略溢出的前提下，这些指令是等价的。

5、根据自己的设计说明单周期处理器的优缺点。

优点：设计简单，结构简单。

缺点：所有指令的时钟周期等长，时钟周期只能由关键路径决定，使得某些执行得很快的指令也得执行相同的时钟周期，造成时间的浪费，部件的空闲，没有得到有效利用。