Verilog单周期CPU设计文档

作者：李健健

1. CPU设计方案综述
2. 总体设计概述

使用Verilog开发一个简单的单周期CPU，总体概述如下：

1. 此CPU为32位CPU
2. 此CPU为单周期设计
3. 此CPU支持的指令集为：

{addu, subu, ori, lw, sw, beq, lui, jal, jr,nop}

1. nop机器码为0x00000000
2. addu, subu不支持溢出
3. 关键模块定义
4. IM
5. 端口说明

表1-IM端口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 信号名 | 方向 | 描述 |
| 1 | PC[31:0] | I | 时钟信号 |
| 2 | instr[31:0] | O | 指令 |

1. 功能定义

表2-IM功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 描述 |
| 1 | 取指令 | 就是取指令 |

1. PC
2. 端口说明

表3-PC端口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 信号名 | 方向 | 描述 |
| 1 | clk | I | 时钟信号 |
| 2 | reset | I | 复位信号 |
| 3 | NPC | I | 下一条指令所在IM地址 |
| 4 | PC | O | 当前指令所在IM地址 |

1. 功能定义

表4-PC功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 描述 |
| 1 | 存储指令的地址 | 保存当前执行指令在IM中的地址 |

1. NPC
2. 端口说明

表5-NPC端口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 信号名 | 方向 | 描述 |
| 1 | branch | I | 分支信号 |
| 2 | j | I | 跳转信号 |
| 3 | jr | I | 指令是不是jr |
| 4 | PC[31:0] | I | 当前PC值 |
| 5 | RegJump[31:0] | I | 跳转寄存器中地址值 |
| 6 | imm26[25:0] | I | 26位立即数 |
| 7 | PC4[31:0] | O | PC+4 |
| 8 | NPC[31:0] | O | 根据各种指令计算出的下一个PC值 |

1. 功能定义

表6-NPC功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 描述 |
| 1 | 计算下一个PC的值 |  |
| 2 | 输出PC4 | 为jr指令写入寄存器做准备 |

1. GRF
2. 端口说明

表7-GRF端口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 信号名 | 方向 | 描述 |
| 1 | clk | I | 时钟信号 |
| 2 | reset | I | 异步复位信号，将32个寄存器中全部清零  1：清零  0：无效 |
| 3 | WE | 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 |
| 4 | A1[4:0] | I | 5位地址输入信号，指定32个寄存器中的一个，将其中存储的数据读出到RD1 |
| 5 | A2[4:0] | I | 5位地址输入信号，指定32个寄存器中的一个，将其中存储的数据读出到RD2 |
| 6 | A3[4:0] | I | 5位地址输入信号，指定32个寄存器中的一个，作为RD的写入地址 |
| 7 | WD[31:0] | I | 32位写入数据 |
| 8 | RD1[31:0] | O | 输出A1指定的寄存器的32位数据 |
| 9 | RD2[31:0] | O | 输出A2指定的寄存器的32位数据 |

1. 功能定义

表8-GRF功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 描述 |
| 1 | 异步复位 | reset为1时，将所有寄存器清零 |
| 2 | 读数据 | 将A1和A2地址对应的寄存器的值分别通过RD1和RD2读出 |
| 3 | 写数据 | 当WE为1且时钟上升沿来临时，将WD写入到A3对应的寄存器内部 |

1. ALU
2. 端口说明

表9-ALU端口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 信号名 | 方向 | 描述 |
| 1 | A[31:0] | I | 参与运算的第一个数 |
| 2 | B[31:0] | I | 参与运算的第二个数 |
| 3 | AluOp[2:0] | I | 决定ALU做何种操作  000：无符号加  001：无符号减  010：与  011：或  100：将B[15:0]做为res[31:16],res[15:0]=0 |
| 4 | eq | O | A与B是否相等  0：不相等  1：相等 |
| 5 | res[31:0] | O | A与B做运算后的结果 |

1. 功能定义

表10-ALU功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 描述 |
| 1 | 加运算 | res = A + B |
| 2 | 减运算 | res = A - B |
| 3 | 与运算 | res = A & B |
| 4 | 或运算 | res = A | B |
| 5 | 加载高位运算 | res = {B[15:0], 16’h0} |

1. DM
2. 端口说明

表11-DM端口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 信号名 | 方向 | 描述 |
| 1 | clk | I | 时钟信号 |
| 2 | reset | I | 异步复位信号  0：无效  1：内存值全部清零 |
| 3 | WE | I | 写使能信号  0：禁止写入  1：允许写入 |
| 4 | MemAddr[31:0] | I | 读取或写入信号地址 |
| 5 | WD[31:0] | I | 32为写入数据 |
| 6 | RD[31:0] | O | 32位读出数据 |

1. 功能定义

表12-DM功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 描述 |
| 1 | 异步复位 | 当reset为1时，DM中所有数据清零 |
| 2 | 写入数据 | 当WE有效时，时钟上升沿来临时，WD中数据写入A对应的DM地址中 |
| 3 | 读出数据 | RD永远读出A对应的DM地址中的值 |

1. EXT
2. 端口说明

表13-EXT端口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 信号名 | 方向 | 描述 |
| 1 | imm16[15:0] | I | 代扩展的16位信号 |
| 2 | sign | I | 无符号或符号扩展选择信号  0：无符号扩展  1：符号扩展 |
| 3 | imm32[31:0] | O | 扩展后的32位的信号 |

1. 功能定义

表14-EXT功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 描述 |
| 1 | 无符号扩展 | 当sign为0时，将imm16无符号扩展输出 |
| 2 | 符号扩展 | 当sign为1时，将imm16符号扩展输出 |

1. MUX
2. 功能说明

行为级建模Multiplexer

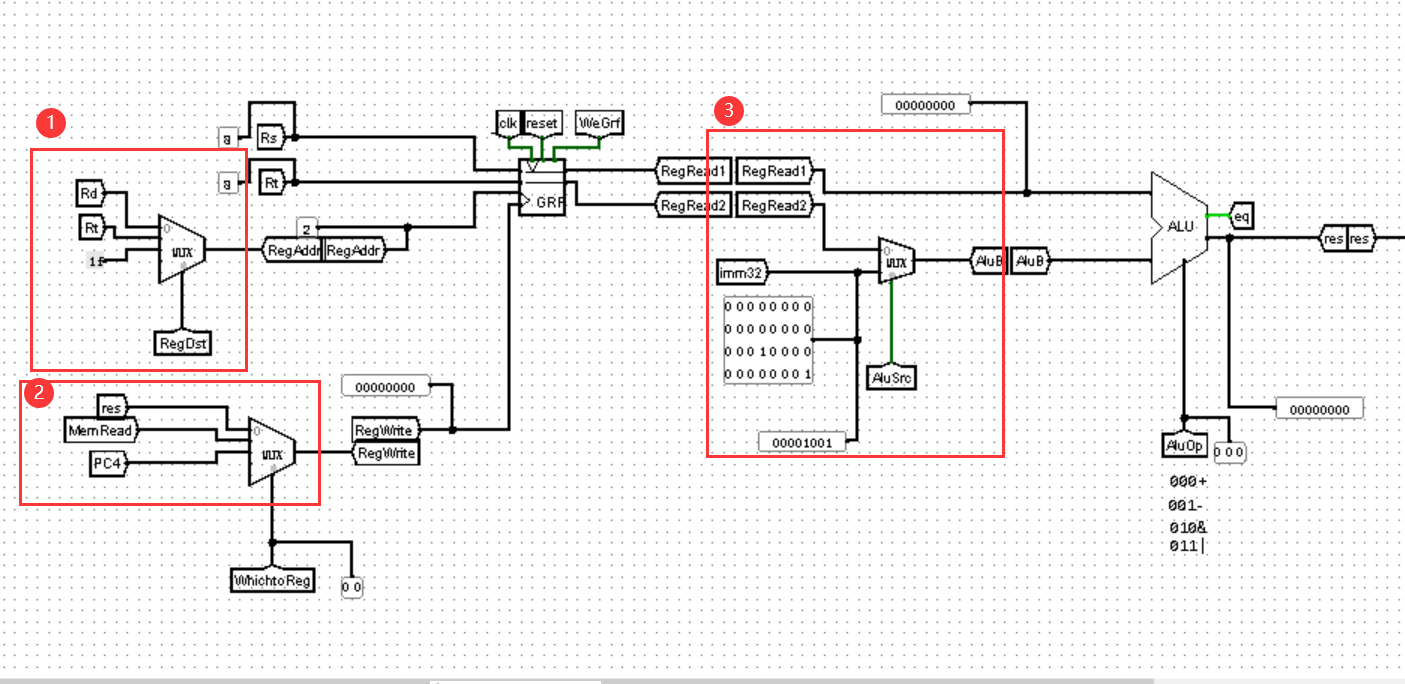


图1-MUX应用之处

分别应用于GRF写入地址、GRF写入数据和ALU第二个运算符的选择上

1. Controller
2. 端口说明

表15-Controller端口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 信号名 | 方向 | 描述 |
| 1 | instr[31:0] | I | instr[31:26],6位控制信号 |
| 2 | eq | I | RegRead1和RegRead是否相等 |
| 3 | WeGrf | O | GRF写使能信号  0：禁止写入  1：允许写入 |
| 4 | WeDm | O | DM的写入信号  0：禁止写入  1：允许写入 |
| 5 | RegDst[1:0] | O | GRF写入地址选择信号  0：Rd  1：Rt |
| 6 | WhichtoReg[1:0] | O | 将何种数据写入GRF？  00：ALU计算结果  01：DM读出信号  11：upperImm |
| 7 | AluSrc | O | 参与ALU运算的第二个数，来自GRF还是imm  0：来自GRF  1：imm |
| 8 | AluOp[2:0] | O | ALU的控制信号 |
| 9 | sign | O | 是否对imm16进行符号扩展  0：不进行符号扩展  1：进行符号扩展 |
| 10 | branch | O | instr是否为beq信号  0：不是  1：是 |
| 11 | JType | O | 是不是J型指令，只有j和jal是 |
| 12 | jr | O | 是不是jr指令 |

1. 真值表

表16-Controller内部真值对应

he

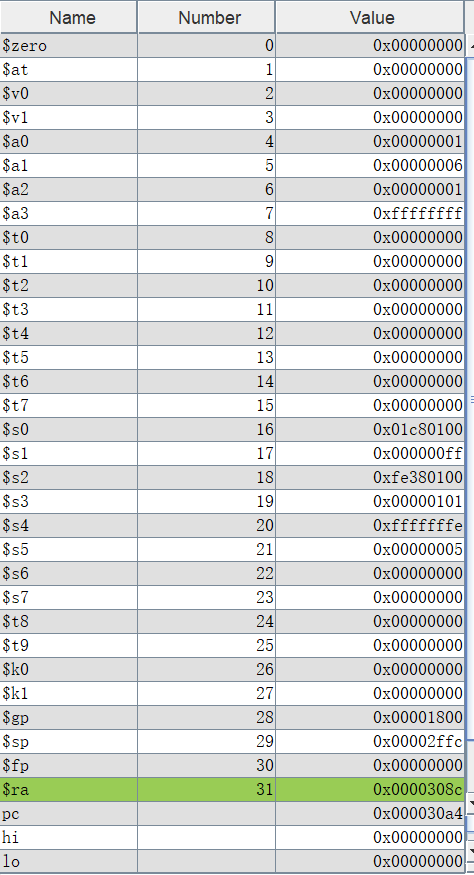
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 端口 | addu | subu | ori | lw | sw | lui | beq | jal | jr |
| op | 000000 | 000000 | 001101 | 100011 | 101011 | 001111 | 000100 | 000011 | 000000 |
| func | 100001 | 100011 |  |  |  |  |  |  | 001000 |
| WeGrf | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| WeDm | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| RegDst | 00 | 00 | 01 | 01 | 00 | 01 | 00 | 10 | 00 |
| WhichtoReg | 00 | 00 | 00 | 01 | 00 | 00 | 00 | 10 | 00 |
| AluSrc | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| AluOp | 000 | 001 | 011 | 000 | 000 | 100 | 000 | 000 | 000 |
| sign | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| branch | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| JType | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| jr | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

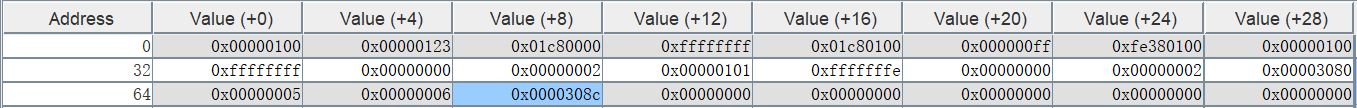
1. 测试方案
2. 测试代码：

.text

ori $a0,$0,0x100  
ori $a1,$a0,0x123  
lui $a2,456  
lui $a3,0xffff  
ori $a3,$a3,0xffff  
addu $s0,$a0,$a2  
addu $s1,$a0,$a3  
addu $s4,$a3,$a3  
subu $s2,$a0,$a2  
subu $s3,$a0,$a3  
sw $a0,0($0)  
sw $a1,4($0)  
sw $a2,8($0)  
sw $a3,12($0)  
sw $s0,16($0)  
sw $s1,20($0)  
sw $s2,24($0)  
sw $s3,44($0)  
sw $s4,48($0)  
lw $a0,0($0)  
lw $a1,12($0)  
sw $a0,28($0)  
sw $a1,32($0)  
ori $a0,$0,1  
ori $a1,$0,2  
ori $a2,$0,1  
beq $a0,$a1,loop1  
beq $a0,$a2,loop2  
loop1: sw $a0,36($t0)  
loop2: sw $a1,40($t0)  
jal loop3  
jal loop3  
sw $s5,64($t0)  
ori $a1,$a1,4  
jal loop4  
loop3:sw $a1,56($t0)  
sw $ra,60($t0)  
ori $s5,$s5,5  
jr $ra  
loop4: sw $a1,68($t0)  
sw $ra,72($t0)

1. MARS中运行结果





1. 该CPU运行输出结果

@00003000: $ 4 <= 00000100  
@00003004: $ 5 <= 00000123  
@00003008: $ 6 <= 01c80000  
@0000300c: $ 7 <= ffff0000  
@00003010: $ 7 <= ffffffff  
@00003014: $16 <= 01c80100  
@00003018: $17 <= 000000ff  
@0000301c: $20 <= fffffffe  
@00003020: $18 <= fe380100  
@00003024: $19 <= 00000101  
@00003028: \*00000000 <= 00000100  
@0000302c: \*00000004 <= 00000123  
@00003030: \*00000008 <= 01c80000  
@00003034: \*0000000c <= ffffffff  
@00003038: \*00000010 <= 01c80100  
@0000303c: \*00000014 <= 000000ff  
@00003040: \*00000018 <= fe380100  
@00003044: \*0000002c <= 00000101  
@00003048: \*00000030 <= fffffffe  
@0000304c: $ 4 <= 00000100  
@00003050: $ 5 <= ffffffff  
@00003054: \*0000001c <= 00000100  
@00003058: \*00000020 <= ffffffff  
@0000305c: $ 4 <= 00000001  
@00003060: $ 5 <= 00000002  
@00003064: $ 6 <= 00000001  
@00003074: \*00000028 <= 00000002  
@00003078: $31 <= 0000307c  
@0000308c: \*00000038 <= 00000002  
@00003090: \*0000003c <= 0000307c

1. 思考题
2. 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？

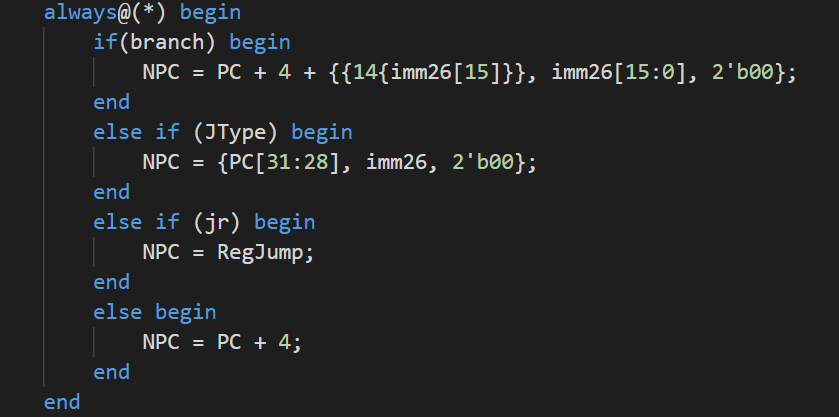


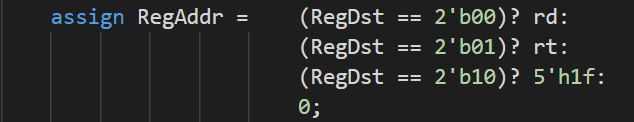
MIPS中以字节为单位，我们的DM中，以32位的register为单位。

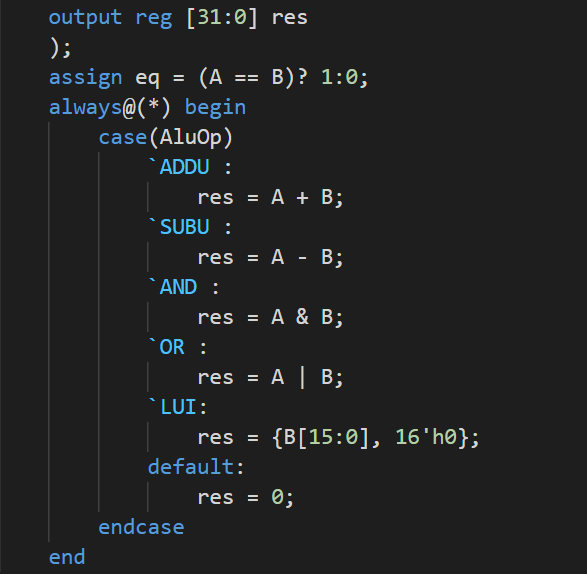
addr是ALU单元的输出端口接过来的，代表的是要读取的DM存储器的地址。

1. 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。

if-else, assign, case-endcase







assign不需要自己再定义寄存器；case-endcase和assign可以通过宏定义的方式，使代码更加美观，增强可读性。

1. 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？

特点就是都是存储器，PC、GRF、DM

1. 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。

addi与addiu的区别在于，当出现溢出时，

addiu忽略溢出，将溢出的最高位舍弃；

addi会报告SignalException(IntegerOverflow)。

故忽略溢出，二者等价。

1. 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，扩展性好，要加什么指令，一目了然

缺点：时钟频率取决于执行时间最长的指令，拖了执行时间短的指令的后腿。