2018年4月12日19:43:37

微处理器的设计原理

1. 简单的微处理器由运算器和控制器两部分组成。
2. 运算器由算术逻辑运算单元ALU、通用寄存器、程序状态字PSW、三态缓冲器和多路选择器组成
3. 控制器由程序计数器PC、指令寄存器IR、指令译码器、操作控制器和时序产生器组成。
4. CISC微处理器的操作控制器一般采用微程序控制器，RISC微处理器的操作控制器一般采用硬连线控制器，控制器的控制方式常采用联合控制方式。
5. 微程序控制器主要由:控制存储器、微指令寄存器和地址转移逻辑电路三大部分组成。

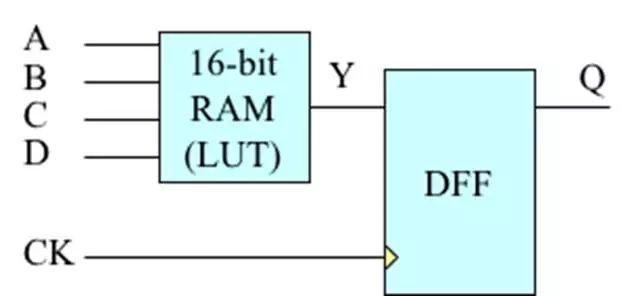
2018年4月13日09:43:50

1. FPGA器件的设计输入有多种方式，如绘制原理图、编写代码或是调用IP内核。
2. 可实现为硬件电路的语法为可综合的语法；而不能够实现到硬件电路中，却常常可作为仿真验证的高层次语法则称为行为级语法。
3. RTL视图是用户输入代码进行综合后的逻辑功能视图。它很好地将用户的代码用逻辑门的方式诠释出来。
4. 在现代逻辑设计中，时序逻辑是设计的核心，而寄存器又是时序逻辑的基础。
5. 数字电路按照逻辑功能一般可以分为组合逻辑和时序逻辑。组合逻辑不含有任何用于存储比特信号的电路，它的输出只和当前电路的输入有关，比如加法器电路。

时序逻辑可以含有组合逻辑，并且一定有用于存储比特信号的电路（一般为寄存器），时序逻辑的输出值不仅和当前输入值有关，一般还和电路的原状态有关。

2018年4月14日09:50:41

1. 各种逻辑门需要实现的输入\输出关系大多时候是通过这个LUT来实现的。
2. 此外，现代的PAL或GAL器件虽然理论上可以擦除原有的内容重新编程，可是实际上一旦焊到电路板上，要想修改，就得把它们烫下来，重新编程，再重新焊回到电路板上，非常麻烦。（不过，作者确实见过别的同事不幸地做过这样的事情。）因此使用PAL或GAL这类器件基本没有后悔药可以吃。而当一个器件中晶体管数量进一步增加后，没有人能一次就把复杂的逻辑功能设计对，这就需要为用户提供后悔药，让器件可以在焊到电路板上之后仍然可以随时重新编程与设置，也就是说，要做到现场可编程。于是人们就设计出了**现场可编程门阵列**（field-programmable gate array, FPGA）**器件**。
3. 在实际器件中，每一个查询表的后面都有一个寄存器（DFF），组成一个逻辑单元，如图3所示。



1. 所谓查询表实质上是一个小小的存储器，对于一个4输入（或6输入）的查询表，里面有16个（或64个）存储单元，每个存储单元包含一个比特。查询表的输入线是存储器的地址线，根据这几根输入线的不同逻辑组合，存储器指向相应的存储单元，将预先存在那里的比特信息读出，驱动输出端，从而生成需要的逻辑功能。

2018年4月14日21:15:27

1. 总线是一组能为多个部件共享的公共信息传送线路，可以分时地发送与接收各部件的信息。总线在传送数据格式上可采用串行传送或并行传送，能一次并行传送的数据位数被定义为总线的数据通路宽度，而单位时间传送的数据量定义为总线的数据传输率。

比如常用的IIC、USB。

1. 算术逻辑运算单元ALU的核心是加法器。
2. 一台计算机所能执行的全部指令的总和，称为该机的指令系统或称为指令集合。指令系统是是程序员编制程序的基础，利用各种程序设计语言编制的程序，必需通过解释或编译转换为机器可以识别与执行的指令才能执行。
3. 指令系统表明一台计算机具有哪些硬件功能，CPU的基本任务是解释执行指令代码，所以指令系统又在很大程度上决定这计算机如何工作，它是硬件逻辑设计的基础。

从加强指令的功能角度出发，希望指令系统中包含尽可能多的指令。具有复杂指令集合的计算机，被称为复杂指令集计算机，简称为CISC。

1. 早期的指令系统只考虑对单个数据的运算操作，现在是在指令中直接提供对各种数据结构的硬件支持能力，如对数组、表、队列、堆栈等的操作指令，从而向高级语言靠拢。
2. 从提高指令本身的执行效率角度出发，希望指令比较简单，能在一个时钟周期内执行完毕。于是出现了精简指令系统的计算机,即RISC。
3. 一个完善的指令系统应满足的要求：
4. 完备性： 指令系统丰富、功能齐全、使用方便。
5. 有效性：程序占用存储空间小，执行速度快。
6. 规整性：包括指令系统的对称性、均匀性和数据格式的一致性。
7. 兼容性：各机种上基本软件可以通用。完全兼容不可能，只能向上兼容。即低档机上运行的软件可以在高档机上运行。
8. **指令类型**
9. 传送类指令

单纯的传送指令将数据从一个地方（源地址）传送到另一个地方（目的地址）。传送结束后，源地址中的内容保持不变，并被“复制”到目的地址。

对三个问题进行说明：

1. 规定传送范围：寄存器和存储器之间、寄存器与寄存器之间。速度与价格的均衡。
2. 指明传送单位：数据可以按字节、字、双字或数组为单位进行传送。
3. 设置寻址方式：传送指令以什么方式给出数据的源地址和目的地址，这涉及到寻址方式的选择。
4. 输入/输出指令

输入/输出指令也是一种传送指令，只是传送的一方为输入输入设备，主机与外围设备进行各种信息交换，可由执行输入输出指令来实现。如主机执行输出指令向外设送数或发出各类控制指令

1. 算术逻辑运算指令

计算机的运算可以分为算术运算，逻辑运算两大类。

1. 算术运算指令

加(ADD)、减(SUB)、加1(INC)、减1(DEC)等

乘除运算和浮点运算不是所有机器都具备的，因为它们选需要专门的乘除运算部件和浮点运算部件作为硬件支持。

b． 逻辑运算指令

基本的逻辑运算只有与、或、非、异或这4种。

因此在一个指令系统中往往也只设置这4种逻辑运算指令。

这些指令都按位进行逻辑运算，故也称它们为位操作指令。

位移操作，例如乘法、除法中的左移、右移等，也常常归入算术逻辑运算指令。

1. 程序控制指令

主要用来控制指令执行的顺序，即克制程序的流程，比如在什么情况下程序要进行转移，往何处转移等等。

1. 转移指令

转移指令又分无条件转移指令和条件转移指令。

遇到无条件转移指令就无条件地转移到指令所指明的地址，除操作码外只给出转移地址。

条件转移指令主要用于程序的分支。条件转移指令除操作码和转移地址外，还需要设置转移条件。

JZ为零转移，JNZ为非零转移，JS为负转移，JNS为正转移，JC为进位转移，JNC为无进位转移，JO为溢出转移,JNO为无溢出转移。

1. 循环指令
2. 转子程序指令与返回指令

主程序执行的过程中需要调用子程序时了，执行一条转子指令，程序便转到支子程序的入口地址。

1. 软中断指令

由程序指令引起中断，称为软中断。

e.其他指令

2018年4月14日23:12:02

[8051单片机](https://www.baidu.com/s?wd=8051%E5%8D%95%E7%89%87%E6%9C%BA&from=1012015a&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9mHmsujubmW-BnAnvuWnL0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHRdn16sP1f3)共有111条指令，按指令功能分，可分为  
数据传送类29条，算术运算类24条，逻辑运算类24条，位操作类12条，控制转移类2搜索2条。  
指令[寻址方式](https://www.baidu.com/s?wd=%E5%AF%BB%E5%9D%80%E6%96%B9%E5%BC%8F&from=1012015a&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9mHmsujubmW-BnAnvuWnL0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHRdn16sP1f3)共7种，分别是：立即数寻址，[直接寻址](https://www.baidu.com/s?wd=%E7%9B%B4%E6%8E%A5%E5%AF%BB%E5%9D%80&from=1012015a&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9mHmsujubmW-BnAnvuWnL0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHRdn16sP1f3)，[寄存器寻址](https://www.baidu.com/s?wd=%E5%AF%84%E5%AD%98%E5%99%A8%E5%AF%BB%E5%9D%80&from=1012015a&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9mHmsujubmW-BnAnvuWnL0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHRdn16sP1f3)，[寄存器间接寻址](https://www.baidu.com/s?wd=%E5%AF%84%E5%AD%98%E5%99%A8%E9%97%B4%E6%8E%A5%E5%AF%BB%E5%9D%80&from=1012015a&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9mHmsujubmW-BnAnvuWnL0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHRdn16sP1f3)，变址寻址，[相对寻址](https://www.baidu.com/s?wd=%E7%9B%B8%E5%AF%B9%E5%AF%BB%E5%9D%80&from=1012015a&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9mHmsujubmW-BnAnvuWnL0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHRdn16sP1f3)，位寻址。  
标准格式：  
标号:操作码 [(目的操作数),(源操作数)];注释

8051的每一条指令都有对应的汇编表示方法。

比如汇编指令： MOV A,#30H

这是一个双字节指令。它的机器指令是：74H,30H。 74H代表MOV A操作，A代表特殊寄存器ACC,ACC在数据池的地址是E0H。

软核处理器一见到首字节指令是74H,那它默认就知道三件事：

1. 它是双字节指令，紧接着取到的下一字节不能当成新指令，而是它的一部分。
2. 它的目的地址是ACC。ACC在数据池的地址是E0H,而指令首字节是74H也已经默认指定的目的地址是它。
3. 此操作是移动数据操作，操作的方式是把下一字节的数据放入ACC。

#30H表示是立即数寻址，是可操作的数，无须从数据池里面找数据的方法，就称为立即寻址。#30H表示一个可以操作的数，而不是地址。

2018-4-16 09:45:27

1. 指令格式

一条指令一般应提供两方面的信息：第一部分为操作码，用以指明操作的性质，即要求CPU做何操作。第二部分称做地址码，用以指明操作数、操作结果、下一条指令的有关信息。

基本形态为：

操作码OP 地址码AD

1. 设计指令格式时需要考虑的问题：
2. 指令字长需要多少位，是定字长还是变字长
3. 操作码结构需要多少位，位数与位置是固定还是可扩展
4. 地址结构，即指令的执行涉及哪些地址
5. 寻址方式，即如何获得操作数地址，是直接给出还是间接给出，或是经过变址计算获得。
6. 指令字长

指令字长位数越多，所能表示的操作信息和地址信息也就越多，指令功能越丰富。位数多占用的存储空间就越多，读取指令时间增多。简单指令执行时间短，复杂指令执行时间长。

现代计算机广泛采用变长指令格式，能短则短，能长则长

1. 操作码格式

操作码的位数决定了操作类型的多少，位数越多，所能表示的操作的类型也越多。

指令字长确定后，如果地址码所占位数越多，则操作码所占的位数也就越少。

1. 定长操作码

操作码的位数与位置固定，通常占据指令最前面几位。当指令字长较长时，允许有足够的操作码位数，可采用定长的操作码。

（2）可扩展操作码

当指令字长较短时，可以利用某些指令中地址段位数的减少，扩展操作码的位数。

1. 地址结构

指令所要指明的操作数，可能在存储器中，也可能爱CPU的寄存器中，因此指令中给出的地址信息，可能是存储单元的地址码，也可能是寄存器号。

指令的地址结构是指在指令中给出几个地址？哪些地址？按照地址结构的不同，可将指令分为三地址指令、二地址指令、一地址指令、零地址指令等几种。

1. 寻址方式

指令中如何提供操作数或提供操作数地址，称为寻址方式或编址方式。换句话说，寻址方式是规定如何对地址字段做出解释，以找到操作数。

* 1. 存储结构与存储方式

以什么形式给出地址信息，与存储结构及其存取方式有关。

1. CPU中的寄存器

如果操作数在CPU寄存器中，指令中只需给出寄存器号，或隐含约定寄存器。

1. 主存储器

如果操作数或地址在主存储器中，则应给出相应的存储单元号，即地址码。

1. 堆栈

堆栈是一种后进先出LIFO顺序存取的存储结构。在多数情况下，堆栈是主存储器中的一个区域。

1. 外存储器
2. 外围设备
   1. 常见的寻址方式

立即寻址、直接寻址、间接寻址、变址寻址为最基本的寻址方式。

1. 立即寻址

由指令直接给出操作数，在取指令的同时也就取出了操作数，这样的寻址方式称为立即寻址。

1. 直接寻址（绝对寻址）

指令直接给出操作数地址，根据该地址可读取操作数。从这个地址可以直接读取操作数，地址不再变化，所以称为 直接寻址。

1. 寄存器须知方式

操作数在指定寄存器中，指令给出寄存器号，这是直接寻址的一种方式

1. 直接寻址方式

操作数在主存储器某指定单元中，指令中给出该单元的地址码。

1. 间接寻址

为了使所提供的地址可以修改，即具有可变性，需要采取间接寻址方式。

1. 寄存器间接寻址方式

操作数在主存中，在指令中给出存放操作数地址寄存器号

指令中给出某指定寄存器号，寄存器内容为操作数地址，按照地址访问某主存单元，该单元的内容为操作数。

寄存器号—>操作数地址—>操作数

寄存器数目远少于存储单元数，因此寄存器号所占指令代码位数也较少，可有效缩短指令长度。

1. 间址方式

指令中给出间址单元的地址码，按照该地址访问主存中的某间址单元，从中读取操作数地址，按照操作数地址再次访问某主存单元，该单元的内容是操作数

间址单元地址🡪操作数地址—>操作数

支持了程序的循环，实现了创新点 共享。

1. 自增型寄存器间址方式

指令中给出指定的寄存器号，该寄存器的内容为操作数地址，按照该地址访问主存单元，该单元的内容为操作数，然后寄存器的内容加1。

这种寻址方式适于数组型操作，或队其他内容的连续数据块操作。

1. 自减型寄存器间接寻址
2. 堆栈寻址方式

堆栈指针寄存器SP,隐含约定SP的内容为栈顶单元的地址码。

1. 多重间址
2. 变址、基址寻址
3. 变址寻址方式

指令中给出变址寄存器号和一个形式地址，变址寄存器的内容与形式地址相加，得到操作数有效的地址。按照有效地址访问主存单元，该单元的内容为操作数。

1. 基址寻址方式

当指令字长有限而存储空间较大时，常采用基址寻址方式，以扩展指令的寻址空间。

指令中给出基址寄存器号，形式地址段给出位移量，基址寄存器内容与位移量相加，得到操作数有效地址，按照有效地址访问某主存单元，该单元的内容为操作数。

CPU中有专门的基址寄存器

采用基址寻址方式的目的，是为了扩展有限字长指令的寻址空间。基址寄存器应能提供全字长地址码，足以指向主存的任一单元。

1. 基址加变址方式

基址寻址方式的目的是扩大有限字长指令的寻址空间，而变址寻址方式的目的是为灵活修改地址以适应连续区间的操作。如果要同时兼有这两种功能，可以采用基址加变址的寻址方式。

操作数=基址量+变址量+位移量

这样的寻址方式可以方便处理两维数组。

1. 相对寻址方式

如果选择程序计数器PC作为基址寄存器，就成为相对寻址。PC内容与位移量相加，得到操作数有效地址。显然这种寻址方式是以PC内容为基准的，相对于指令地址移动若干单元，所以称为相对寻址。

相对寻址对实现程序转移特别有利。

1. 页面寻址方式

如果不是将PC内容与位移量进行算术加，而是将PC内容的高位段与位移量相拼接，相对寻址就演变成页面寻址。

1. 扩展型变址寻址方式
   1. 堆栈操作

为了便于处理程序的多重嵌套，如保存多个转子或多重中断时的有关参量，广泛使用了堆栈的存储结构。

2018年4月16日15:21:38

1. RISC

精简指令系统计算机

程序执行时间=I\*CPI\*T

为了提高CPU速度的措施：

（1）选取使用频率最高的一些单指令

（2）采用多级流水线结构：关键是减少CPI即每条指令的周期数。采用流水线技术可使每一时刻都有多条指令重叠执行。

（3） 采用加载/存储结构

（4）延迟加载指令和转移指令

（5）采用简单的指令格式

（6）采用高速缓存结构

（7）采用优化编译程序

2018年4月16日15:32:23

1. **中央处理器组织**

中央处理器是控制器与运算器的总称

1. 控制器的功能

存储器与控制器之间的信息流动称为指令流，存储器与运算器之间的信息流动称为数据流。

1. 控制器的组成

（1）程序计数器PC

又称指令计数器或指令地址寄存器，有些机器用来存放正在执行的指令地址，有些机器用来存放要执行的下一条指令的地址。

(2) 指令寄存器IR

用以存放从内存取出来的现行指令，以便在整条指令执行过程中，完成一 条指令的全部功能控制。

（3） 指令译码器

又称操作码译码器，它对指令寄存器中的操作码进行分析解释，产生相应的控制信号，提供给操作控制信号形成部件。

（4） 脉冲源及启停控制线路

（5）时序信号产生部件

（6）微操作控制信号形成部件

微操作控制信号形成部件是产生微命令的。微命令是构成控制信号序列的最小单位。该部件产生微命令的一句是指令操作码、运行状态、时序信号及被控功能部件反馈的状态条件信号，形成不同指令所需要的操作控制信号序列。

（7）中断机构

（8）总线控制逻辑

2018年4月16日16:41:26

1. 控制器类型

控制器的输入是机器指令代码，输出是微操作控制信号，因此微操作信号发生器是控制器的核心。

1. 寄存器是暂存各类信息的基本部件，在中央处理机内一般有如下寄存器：
2. 通用寄存器

如：累加器、变址寄存器、地址指针、计数器等

通用寄存器常被看成运算器的一个组成部分。

1. 专用寄存器

专门用来完成某一种功能

如：程序计数器PC、指令寄存器IR、存储器地址寄存器MAR和存储器数据寄存器MBR,状态标志寄存器。

状态标志寄存器PSW，用来存放程序状态，程序状态字的各位表征程序和机器运算的状态。

1. CPU内部的数据通路
2. 通过ALU传送的数据通路
3. 通过总线传送的数据通路
4. CPU与存储器间的数据通路

主存储器通诺地址寄存器MAR、数码寄存器MBR与CPU交换信息。

1. CPU与IO设备的数据通路
2. 指令的执行过程
3. 取指令、
4. 分析指令

采用组合逻辑控制器，通过译码电路译出指令各字段的含义，根据指令含义并结合有关的工作状态形成微操作命令序列。

1. 取操作数、
2. 执行指令
3. 指令的执行方式

控制器对于指令的执行通常有三种方式：顺序方式、重叠方式和流水线方式。

7.1 顺序执行方式

顺序执行有两层意思，一是各条机器指令间是按顺序串行执行，二是一条机器指令内各个微操作的串行执行。

顺序执行的优点是控制简单，缺点是速度上不去。

* 1. 重叠执行方式

在一条指令的操作完成前，就开始执行下一条指令。

* 1. 流水控制方式

它基于重叠的原理，但是在更高的程度上重叠。

将一个复杂的处理过程分成M个复杂程度相当、处理时间大致相等的子过程，每个子过程由一个独立的功能部件来完成。处理对象在每个子过程的线路上连续流动，在同一时间，M个部件同时进行不同的操作。

目前最常见的流水控制方式有指令流水控制和运算流水控制。

1. 指令流水控制

它是将指令的执行步骤并行。一条指令的执行过程分为取指令、译码、取操作数、执行四个并行处理的子过程，每个子过程都设置相应的专门部件来完成。

1. 运算流水控制

指运算操作步骤的并行，是用来处理比较复杂的算术运算操作的。如流水线加法器、流水线乘法器、流水线快速傅里叶变换器。

2018年4月16日21:15:56

1. 一台模型机的总体设计

在对一台计算机进行总体设计时，应该考虑以下两个问题：

1. 拟定指令系统

具备哪些功能，要设置哪些寄存器，如何识别操作码，如何寻址、取数等。

1. 确定总体结构
2. 总线结构

CUP 、主存储器、外围设备等通过外总线连接在一起。外总线由地址总线、数据总线和控制总线构成。

内总线NBUS将ALU和各寄存器连接在一起，各寄存器的信息通过ALU实现相互间的传送，传送往哪个寄存器由CP脉冲确定。外总线与内总线通过MAR,MBR连接在一起，实现内外总线的信息互换。

2018年4月16日23:11:48

1. CPLD是基于门阵列来实现逻辑功能的，FPGA是基于查找表look up table结构的。

查找表简称LUT，本质上就是一个RAM。FPGA多使用4输入的LUT,所以每一个LUT可以看成一个有4位地址线的16\*1的RAM。

2018年4月19日17:06:51

锁存器和触发器

1. 大多数数字系统中，除了需要具有逻辑运算和算术运算功能的组合逻辑电路外，还要具有存储功能的电路，而构成存储电路的基本存储单元是锁存器和触发器。
2. 锁存器和触发器都可以存储二进制信号，区别是：

锁存器是对脉冲电平敏感的存储电路，在特定输入脉冲电平作用下改变状态；触发器是对脉冲边沿敏感的存储电路，在时钟脉冲的上升沿或下降沿的变乱瞬间改变状态。

1. 锁存器按照逻辑功能的不同，主要有SR型和D型。
2. 触发器按照逻辑功能的不同，主要有RS型触发器，JK型触发器，D型触发器和T型触发器。

2018年4月21日10:21:19

1. 时序逻辑电路

电路的任何一个时刻的输出状态不仅取决于当时的输入信号，还与电路的原状态有关。

时序电路中必须含有记忆能力的存储器件。

1. 寄存器

在数字电路中，寄存器就是一种在某一特定信号（通常是时钟信号）的控制下用来存储一组二进制数据的时序逻辑电路。寄存器又称数据锁存器，其功能是接收、存储和输出数据，主要由触发器和控制门组成。寄存器一般由多个触发器连接起来，采用一个公共洗脑进行控制，同时各个触发器的数据端口仍然各自独立地接收数据。

1. 计数器

典型的时序逻辑电路，用来记忆时钟脉冲的具体个数。

1. 存储器
   1. RAM电路由地址译码器、存储矩阵和读写控制电路组成。

地址译码器一般分为行地址译码器和列地址译码器两部分。

* 1. ROM电路结构包含存储矩阵、地址译码器和输出缓冲器三个部分。

2018年4月21日10:51:58

1. 一个完整的VHDL程序通常包括库（library）、程序包（package）、实体说明（entity declaration）、结构体（architecture body）和配置（configuration）5个部分组成。
2. VHDL提供的库可以分为两大类：设计库和资源库。
3. VHDL中存在的库大体可以分为五种：ieee库、std库、work库、面向ASIC的库和用户自定义的库。

其中std库和work库是设计库，其他3种都是资源库。

1. 一般来说，设计人员在编写一个设计单元的VHDL程序时都要使用ieeeu和iee库中的std\_logic\_1164程序包。

library ieee；

use ieee.std\_logic\_1164.all;

entity and2 is

port(a:in std\_logic;

b:in std\_logic;

c:out std\_logic);

end and2;

1. 实体定义了模块与外部电路的接口，以及需要传送给结构体的参数。

entity 实体名 is

[generic (参数名：数据类型);]

[port (端口说明)]

end [entity 实体名];

1. 结构体

结构体是对实体功能的具体描述，跟在实体说明的后面。

每个实体可以有多个结构体，，每个结构体对应着实体不同的结构和算法实现方案。

具有不同结构体的实体利用配置语句可以指定用于仿真或综合的结构体。

architecture 结构体名 of 实体名 is

[结构体说明语句;]

begin

功能描述语句（并行处理语句）；

end [architecture] 结构体名；

结构体说明语句位于architecture和begin之间，用于对结构体内部所用的信号、常数、数据类型和函数等进行定义。

比如： signal qout:std\_logic\_vector(7 downto 0);

结构体功能描述语句位于 begin和end之间，这些语句具体描述了结构体的行为和结构。

在结构体中的功能描述语句是并行描述语句，也就是说，这些语句是并行执行的，并不以语句的书写顺序为执行顺序。

包含5种不同类型的并行语句，分别是：块语句、进程语句、信号赋值语句、子程序调用语句和元件例化语句。

1. VHDL中的4种描述风格：行为描述、数据流描述、结构描述以及由前3种组合的混合描述。
   1. 行为描述

只描述了所期望电路的功能额行为，即电路的输入与输出间转换行为，而没有包含任何实现这些功能的硬件结构信息，包含硬件电路的连接结构、元件或其他功能单元的层次结构。

行为描述是一种高层次的描述方式，只需关注功能单元的正确行为即可，无需在具体硬件结构或门级电路实现及具体实现芯片的性能结构上花精力，即编程与硬件无关。

将VHDL的行为描述语句转换为门级描述由VHDL开发工具自动完成，是VHDL综合器的任务。

* 1. 数据流描述

也称为RTL(寄存器传输级)描述方式，类似于布尔方程。

这种描述主要反映数据经过一定的逻辑运算后在输入和输出间的传递，可以清楚地看到数据流出的方向、路径和结果。

* 1. 结构描述

结构描述主要以元件或已经完成的功能模块为基础，应用于元件例化的VHDL程序设计中。

1. 程序包

为了使一组信号定义、数据类型、子程序说明、属性说明以及元件说明等对多个设计实体及其相应的结构体都成为可见，VHDL提供了程序包结构。

1. 配置

配置主要用来描述各种设计实体和元件之间的连接关系以及设计实体和结构体之间的连接关系。例如，设计人员可以利用配置语句来选择不同的结构体。

2018年4月22日10:14:07

1. 数据对象

在VHDL语言中，凡是可以被赋予一个值的客体叫做数据对象。VHDL的数据对象主要包括4种：常量constant、变量variable、信号signal、文件file。

1. 变量和信号的区别
2. 变量赋值是直接的、立即生效的、没有延迟的，而信号赋值是有一定延迟的
3. 变量只是当前值，而信号除当前值外还有许多相关信息
4. 进程对信号敏感，而对变量不敏感
5. 变量只在定义它的进程、过程和函数中可见；而信号则可以是多个进程的全局信号
6. 变量在硬件中没有一定的对应关系，而信号是硬件中连续的抽象描述。

信号和变量的赋值：

目标信号<=表达式；

目标变量:=表达式；

1. VHDL的基本描述语句分为并行语句和顺序语句两大类。

2018年4月23日09:25:27

1. 微程序控制器的基本控制思想是：把操作控制信号编成微指令，存放到一个只读存储器里面，当机器运行时，相应部件逐条地读出这些微指令，从而产生全机所需的各种操作控制信号，使相应部件执行所规定的操作。
2. 微程序控制器主要由控制存储器、微指令寄存器和地址转移逻辑电路组成，其中微指令寄存器又分为微地址寄存器和微命令寄存器两大部分。
3. 微命令的编码有三种方式：直接表示法、编码表示法和混合表示法。若微命令字段采用直接表示法，则微指令称为全水平型微指令。

2018年4月23日21:21:45

1. 程序状态字PSW里面有CF和ZF两个端口，CF是借位标志，ZF是零标志。ALU在运行的时候会修改PSW的这两个标志，因此在CPU的框图里面，ALU和PSW的这两个端口是相连的。

2018年4月24日10:30:56

1. 微程序控制器由控制存储器、微指令寄存器和地址转移逻辑电路三大部分组成。

控制存储器用来存放全部指令系统的微程序。

微指令寄存器用来存放由控制存储器读出来的一条微指令信息。

地址转移逻辑电路承担自动完成修改微地址的任务。（当程序出现分支时）

2018年4月24日15:29:14

1. SW\_B 数据输入控制信号

ALU\_B ALU数据输出控制信号

RS\_B 源寄存器数据输出控制信号

RD\_B 目的寄存器输出控制信号

CS\_I 只读存储器片选控制信号

1. I3、I2 源寄存器编码

I1、I0 目的寄存器编码

1. 地址寄存器单元

地址寄存器不存放外围设备的地址，只存放主存储器的地址，当时钟控制端CLK出现上升沿时，将程序计数器的值送人地址寄存器，地址寄存器的输出直接送往主存储器的地址输入端。

1. 主存储器单元用来存放CPU要运行的程序和数据。

在时钟信号的控制下，将程序计数器的值送往地址寄存器，地址寄存器的输出再送往主存储器，主存储器里面存放的是CPU要运行的程序和数据。

教材32页下面的注释语句也说明了 ROM中从0地址开始依次存放机器语言源程序

但要注意的是参考代码的指令都是8位的。

这里和微程序控制器里面的控制存储器不一样，控制存储器用来存放实现全部指令系统的微程序。

1. 一条微指令对应一个CPU周期，一个CPU周期包含4个T周期。因此RAM的读写信号RD\_D的有效电平范围设计在T2和T3时钟周期。
2. 指令寄存器由LS273组成，存放当前正在执行的指令代码。

clk上升沿时，指令代码经过5选1数据选择器和1:2分配器后，由内部数据总线送人指令寄存器。

1. 指令寄存器的内容经过一个转换电路CONVERT后分为4位的操作码（指令寄存器中的I7~I4）、源寄存器的编码（指令寄存器中的I3、I2）和目的寄存器的编码（指令寄存器中的I1、I0）。

4位操作码直接往操作控制器。

1. 教材上的crom是操作控制器。
2. 微指令代码为24位，其中高18位为微命令，低6位为后继微地址。见教材16页和47页。

控制存储器里面存放的是所有微指令代码，控制存储器会将18位微指令发送给为命令寄存器。

微指令寄存器分为微地址寄存器和微命令寄存器。

控制存储器存放的是微指令，主存储器rom存放的机器语言源程序。

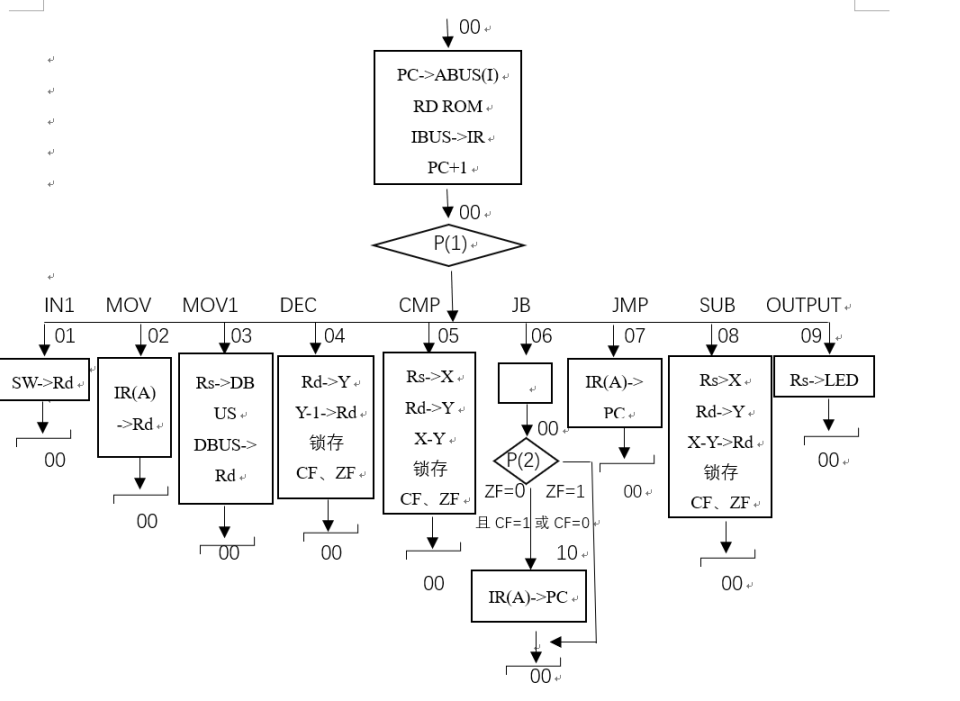
2018年4月25日09:45:46

1. 关于微指令代码的长度

教材上给的参考代码是26位的，而张帅给的师兄的参考是25位的，唯一的区别在于对alu的控制信号，教材上是s0、s1、s2,而师兄的s0、s1，少了一位，其余都是一模一样的。实际上用两位就可以了。

2018年4月25日15:50:14

1. 微程序流程图



每个方框代表一个微指令，需要一个CPU周期的时间来执行，有的指令（汇编指令，不是微指令）需要执行多个微指令才能完成一个操作，因此需要地址转移逻辑。上面的流程图基本上都没有转移，在教材上的参考里面，在使用了RAM的地方都需要地址转移，因为它要多步执行，比如先写地址，然后再读。

那么问题来了，不是时钟每过一个CPU周期就执行一条汇编机器指令吗，微指令地址又转移了，那么下一条汇编指令何时运行呢？

实际上汇编指令的执行顺序是通过程序计数器来控制的，每执行一条汇编指令，PC加一，也就是说每加载一条汇编指令，pc值加一，LDPC是程序计数器的节拍信号，在PC模块的vhdl代码里面可以看出，ldpc每从一低电平上升到高电平时pc就加一，LDPC也是被微程序控制器控制的，在微指令代码的第二位。

从流程图可以看出，只有最上面的方框里面才给pc加一，也就是取指令，后继的指令若执行完了回到00的微地址，就代表取下一条新的指令了。

LOAD和LDPC都是PC的输入控制信号。

PC连接着存汇编指令的rom，rom里面的汇编指令都是按顺序排列的，pc的值为多少，就取rom中的哪条指令。

PC不仅可以加一，还可以让输出等于输入，也就能实现跳转功能。

2018年4月25日20:04:24

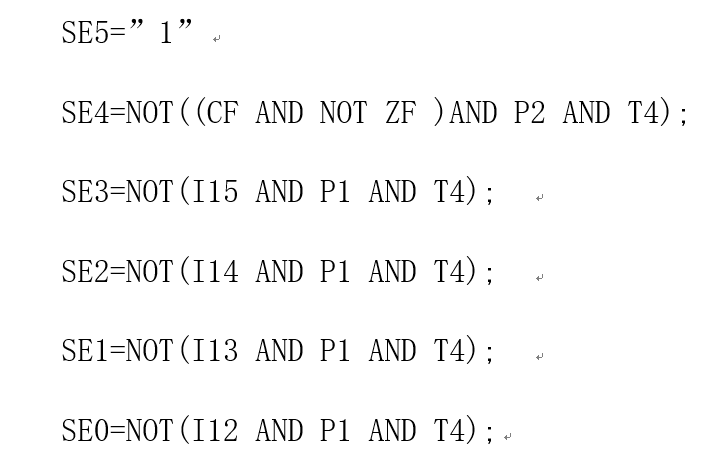
1. 关于地址转移逻辑电路

从微程序流程图可以看出，在需要程序分支的地方，都有P操作，通过P操作，选择后继微指令地址。

地址转移逻辑电路是根据输入的汇编指令的操作码和ZF、CF、P1、P2在T4时刻做出判断的，用来设置微地址寄存器aa的异步置1信号SE5~SE0。

异步置1信号SE是低电平有效，即SE为0时，对应输出的UA为1，否则UA为上个微指令的后继地址的对应位。

在微指令代码表里面，大部分时候P1和P2都为0，根据地址转移逻辑电路



当P1或P2为0的时候，对应的SE必为1，SE是低电平置1，高电平就无效了。也就是说这时候，汇编指令中提取的操作码通过地址逻辑转移后没变，如果程序比较复杂，分支很多，转移电路表达式就复杂。

在微指令代码表里面只有两个地方出现了P为1的情况，P为1，则SE就可能为0，比如操作码为0001，传递给地址转移逻辑单元时I5~I2就是0001，由于微地址为6位，因此是SE5~SE0,UA5~UA0。

在第一个P测试的地方，就是P1为1，P2为0，这时SE5和SE4都为1，即异步置1无效，SE3~SE2为1110，即SE5~SE1都为1，SE0为0，这样得到的微地址为000001。

每条汇编指令都需要从微程序的最上面开始执行，即先执行取指令操作，微地址为000000

比如DEC指令，操作码为04，即0100，执行000000地址的微指令，P1为1，P2为0，I5~I2为：0100，SE3~SE0为：1011，那么只有SE2为0，即置1，其余为0.得到的微地址为000100，

再比如JMP指令07,即0111，I5~I2为0111，SE3~SE0为1000，即最低的3位都要置1，其余位为0,这里的0来自000000微地址的后继微地址。得到的后继微地址就为000111，

对于JB指令，操作码为06，即0110，经过P1操作后得到的微地址为000110，然后要进行P2操作，这时P1为0，P2为1，那么SE3~SE0都为1，即无效，SE5=1,SE4与ZF和CF有关。若ZF=0,CF=1,则SE4=0,若ZF=1.CF=0,则SE4=1,

因为JB指令的后继微地址为10（十六进制），在上面的两种ZF、CF情况下都要跳转到这个地址，因此JB的后继微地址应该为10，而不是为00。

实际上这里的逻辑转移电路，输入是多少，输出就是多少，操作码中有1的地方，对应的SE就为0，然后又强制置1，即没变，操作码中为0的地方，对应的SE为1，输出根据000000微地址的后继微地址还是000000来设置，即还是0。

也就是说不管输入的操作码是多少，得到的后继微地址就是多少，只是在前面加了两个0而已。

2018年4月25日21:23:52

1.微程序流程图中的每个方框代表一个微指令，框下面的菱形从属于它上面的框。也就是说P操作的框属于它上面的方框。不是单独执行一个CPU周期。

2018年4月26日21:33:4

1. 逻辑地址转移电路表达式

SE5=”1”

SE4=”1”

SE3= NOT ((((CF AND NOT ZF) AND P2) OR (P1 AND I15) )AND T4)

SE2= NOT ((((CF AND NOT ZF) AND P2) OR (P1 AND I14) )AND T4)

SE1= NOT ((((CF AND NOT ZF) AND P2) OR (P1 AND I13) )AND T4)

SE0= NOT ((((CF AND NOT ZF) AND P2) OR (P1 AND I12) )AND T4)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **LOAD** | **LDPC** | **LDAR** | **LDIR** | **LDRI** | **LDPSW** | **RS\_B** | **S1** | **S0** | **ALU\_B** | **SW-B** | **LED\_B** | **RD\_D** | **CS\_D** | **RAM\_B** | **CS\_I** | **ADDR\_B** | **P1** | **P2** | **Ua5-Ua0** |

LOAD、LDPC: 程序计数器的控制信号

LDADR: RAM的地址寄存器

LDIR:指令寄存器的节拍信号

LDRI:通用寄存器的节拍信号

LDPSW: PSW的节拍信号

RS\_B、ALU\_B、ADDR\_B:MUX3\_2的控制信号，输出数据的选择控制

LED\_B:分配器FEN2的控制信号

SW\_B、RAM\_B:MUX3\_1的控制信号，输入数据的选择控制

RD\_D、CS\_D:RAM的读写控制信号

CS\_I: ROM的片选信号

1. 在微命令寄存器里面

IF(T2'EVENT AND T2='1')THEN

DATAOUT(18 DOWNTO 0)<=O(18 DOWNTO 0);

END IF;

LDPC<= DATAOUT(17) AND T4;

LOAD<=DATAOUT(18);

LDPC在T4的时候可能会置1，其余都为0，也就是说，在一条微指令信号里面，如果LDPC为1，那么它一定会有上升沿，再加上若LOAD为1的话，pc就能加一了。进一步理解，若执行两个000000的微地址指令，也会pc加一，不用执行其他微指令，给ldpc设置为0。

LDIR<=DATAOUT(15) AND T3;

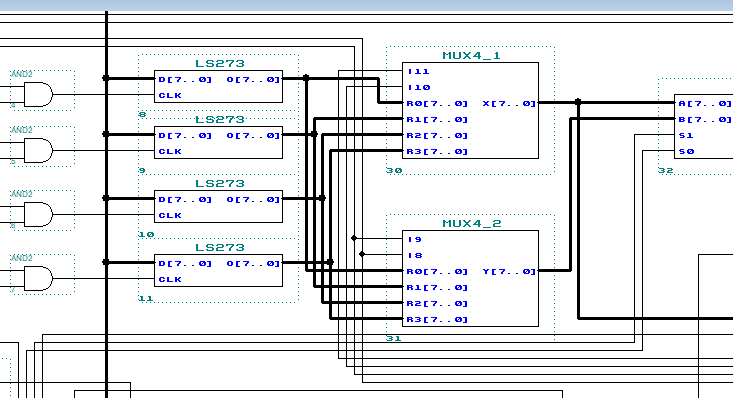
LDIR指令寄存器的节拍，在T3时刻，出现上升沿。只有LDIR为高的时候，才会在T3时刻出现上升沿。若LDIR为0，整个CPU周期它都没有上升沿，那样就不会把汇编指令传递给convert单元了。也就是说在000000地址的微指令，需要设置LDIR为1，其余的微指令都设置为0.

LDRI<=DATAOUT(14) AND T4;

控制通用寄存器是节拍信号，也就是说要用到通用寄存器的微指令，LDRI都要设置为1，那么为什么CMP指令的LDRI不为1呢

实际上I11~I8，汇编指令里面的4位用来选择通用寄存器，当读寄存器的时候，同过这几位就能选择读取哪个寄存器了，而当要写入通用寄存器的时候就需要给出节拍信号LDRI,这就是为什么CMP指令对应的微指令的LDRI为0，而不是为1了。其他指令：IN,MOV,MOV1,DEC,SUB的LDRI都为1，因为它们要写入寄存器。总之，要写入通用寄存器的命令，LDRI设置为1，否则设置为0。

LDPSW<=DATAOUT(13) AND T4; LDPSW是PSW的节拍信号，在DEC、CMP、SUB命令会修改CF和ZF标志，因此这几个命令的LDPSW要设置为1.而JB指令只是读取PSW里面的ZF和CF。



对于ALU\_B、RS\_B、ADDR\_B这个MUX3\_2的控制信号，它们大部分情况下都为1，也就是让DBUS输出全为0。

对于MOV指令，RS\_B、ALU\_B、ADDR\_B分别为1、1、0,那么在MUX3\_2里面DBUS<=AOUT,也就是选择汇编指令里面的操作数为输入

对于MOV1指令，RS\_B、ALU\_B、ADDR\_B分别为0、1、1 , 则DBUS<=RSOUT; 也就是选择源通用寄存器里面的数据

上面两种情况，LED\_B都为1，也就是将DBUS的数据输入到MUX3\_1,而不是直接输出到数码管，因为这里的数据还要用

对于上面两条指令，SW\_B和RAM\_B都为1，即在MUX3\_1里面选择FEN2为输入，FEN2在输入给DBUS,这个外部数据总线是连接到4个通用寄存器的，加上LDRI为1，和I8、I9就能给通用寄存器赋值了。

对于DEC指令，RS\_B、ALU\_B、ADDR\_B分别为1、0、1 ,MUX3\_2会选择ALU的计算结果为输入。SW\_B和RAM\_B都为1，即在MUX3\_1里面选择FEN2为输入，FEN2在输入给DBUS,

JB指令会执行ALU运算，不仅会修改ZF和CF，还将相减的结果输入到数据总线里面，但是LDRI为0，即不能将数据赋值给通用寄存器。

2018年4月27日20:09:33

1. 为什么在波形仿真图里面看到的PC的值要比对应的汇编指令的地址要大一？

应为第一个CPU周期并没有执行汇编语句，而是取指令，counter产生一个时钟信号，带T4,然后通过微程序控制器给LDPC设置为1，然后通过PC单元，这时pc的值加一了，但还没有取汇编指令，

**LDIR<=DATAOUT(15) AND T3;**

**LDPC<= DATAOUT(17) AND T4;**

关键在这里，在第一个CPU周期里，T4时会将PC加一，在T3时会给指令寄存器一个时钟节拍，也就是说一开始在PC还是0时，已经在T3时刻将rom里面的第一条汇编指令取出给CONVERT了。因为一开始pc的值为0，rom的addr也为0，取的是第一条汇编指令，只要来一个ldir就会回将汇编指令传递给convert。

即在第一个CPU周期里面，T3时刻取了第一条汇编指令，T4时刻给PC加一。然后取的汇编指令立马通过convert再传递给微程序控制器，

应该是，一开始通过时序电路，微程序控制器先执行000000地址的微指令，给LDPC和LDIR都设置为1，但LDIR先设置为1，在T3时刻就将第一条汇编指令取出了，然后将操作码赋给操作控制器，在T3时刻就计算程序分支，然后开始执行下一条微指令。

2018年4月29日16:35:26

1. 计算机中基本采用补码加减法

当操作码要求做加法运算时，可直接将补码表示的两操作数相加，不必考虑他们的符号是正或是负数。

1. 求补码的原码

即对补码再做求补码运算，得到的就是原码

2018年4月29日22:05:27

SE5<='1';

SE4<=NOT(( CF AND (NOT ZF)) AND P2 AND T4);

SE3<=NOT(I15 AND P1 AND T4);

SE2<=NOT(I14 AND P1 AND T4);

SE1<=NOT(I13 AND P1 AND T4);

SE0<=NOT(I12 AND P1 AND T4);

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **微地址** | **LOAD** | **LDPC** | **LDAR** | **LDIR** | **LDRI** | **LDPSW** | **RS\_B** | **S1** | **S0** | **ALU\_B** | **SW-B** | **LED\_B** | **RD\_D** | **CS\_D** | **RAM\_B** | **CS\_I** | **ADDR\_B** | **P1** | **P2** | **Ua5-Ua0** |
| **000000** | **1** | **1** | **0** | **1** | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **1** | **1** | **0** | **000000** |
| **000001** | **1** | **0** | **0** | **0** | **1** | **0** | **1** | **0** | **0** | **1** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** |  |
| **000010** | **1** | **0** | **0** | **0** | **1** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **0** |  |
| **000011** | **1** | **0** | **0** | **0** | **1** | **0** | **0** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** |  |
| **000100** | **1** | **0** | **0** | **0** | **1** | **1** | **1** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** |  |
| **000101** | **1** | **0** | **0** | **0** | **0** | **1** | **1** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** |  |
| **000110** | **1** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **1** |  |
| **000111** | **0** | **1** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **0** |  |
| **001000** | **1** | **0** | **0** | **0** | **1** | **1** | **1** | **0** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** |  |
| **001001** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **1** | **0** | **1** | **1** | **1** | **1** | **1** | **0** | **0** |  |
| **010000** | **0** | **1** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **0** |  |
| **001010** | **1** | **0** | **0** | **0** | **1** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **0** | **001011** |
| **001011** | **1** | **0** | **0** | **0** | **0** | **1** | **1** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **000000** |

MVO R1,00H 0010000100000000 00000000

MOV R2,04H 0010001000000100 00000001

L1: IN1 R0 0001000000000000 00000010

DEC R2 0100001000000000 00000011

CMP1 R2,00H 和0相比，判断大小 1010111000000000 00000100

JB OUT 小于0的话，就退出 0110000000001101 00000101

CPM1 00H, R0 1010110000000000 00000110

JB L1 0110000000000010 00000111

CMP R0,R1 0101000100000000 00001000

JB L2 0110000000001011 00001001

JMP L1 0111000000000010 00001010

L2: MOV1 R0, R1 0011000100000000 00001011

JMP L1 0111000000000010 00001100

OUT: NOT R1 1000110100000000 00001101

OUTPUT R1 1001010000000000 00001110

DOUT<="0010000100000000"WHEN ADDR="00000000"AND CS='0'ELSE -- MOV R1,00H

"0010001000000100"WHEN ADDR="00000001"AND CS='0'ELSE -- MOV R2,04H

"0001000000000000"WHEN ADDR="00000010"AND CS='0'ELSE -- L1: IN1 R0

"0100001000000000"WHEN ADDR="00000011"AND CS='0'ELSE -- DEC R2

"1010111000000000"WHEN ADDR="00000100"AND CS='0'ELSE -- CMP1 R2,00H

"0110000000001101"WHEN ADDR="00000101"AND CS='0'ELSE -- JB OUT

"1010110000000000"WHEN ADDR="00000110"AND CS='0'ELSE -- CPM1 00H, R0

"0110000000000010"WHEN ADDR="00000111"AND CS='0'ELSE -- JB L1

"0101000100000000"WHEN ADDR="00001000"AND CS='0'ELSE -- CMP R0,R1

"0110000000001011"WHEN ADDR="00001001"AND CS='0'ELSE -- JB L2

"0111000000000010"WHEN ADDR="00001010"AND CS='0'ELSE -- JMP L1

"0011000100000000"WHEN ADDR="00001011"AND CS='0'ELSE -- L2: MOV1 R0, R1

"0111000000000010"WHEN ADDR="00001100"AND CS='0'ELSE -- JMP L1

"1000110100000000"WHEN ADDR="00001101"AND CS='0'ELSE -- OUT: SUB R3,R1

"1001010000000000"WHEN ADDR="00001110"AND CS='0'ELSE -- OUTPUT R1

"00000000";

DOUT<="0010000100000000"WHEN ADDR="00000000"AND CS='0'ELSE

"0010001000000100"WHEN ADDR="00000001"AND CS='0'ELSE

"0001000000000000"WHEN ADDR="00000010"AND CS='0'ELSE

"0100001000000000"WHEN ADDR="00000011"AND CS='0'ELSE

"1010111000000000"WHEN ADDR="00000100"AND CS='0'ELSE

"0110000000001101"WHEN ADDR="00000101"AND CS='0'ELSE

"1010111000000000"WHEN ADDR="00000110"AND CS='0'ELSE

"0110000000000010"WHEN ADDR="00000111"AND CS='0'ELSE

"0101000100000000"WHEN ADDR="00001000"AND CS='0'ELSE

"0110000000001011"WHEN ADDR="00001001"AND CS='0'ELSE

"0111000000000010"WHEN ADDR="00001010"AND CS='0'ELSE

"0011000100000000"WHEN ADDR="00001011"AND CS='0'ELSE

"0111000000000010"WHEN ADDR="00001100"AND CS='0'ELSE

"1000110100000000"WHEN ADDR="00001101"AND CS='0'ELSE

"1001010000000000"WHEN ADDR="00001110"AND CS='0'ELSE

"00000000";

BEGIN

CASE ADDR IS

WHEN"000000"=>DATAOUT<="1101001001111110110000000";

WHEN"000001"=>DATAOUT<="1000101001011111100000000";

WHEN"000010"=>DATAOUT<="1000101001111111000000000";

WHEN"000011"=>DATAOUT<="1000100001111111100000000";

WHEN"000100"=>DATAOUT<="1000111100111111100000000";

WHEN"000101"=>DATAOUT<="1000011011111111100000000";

WHEN"000110"=>DATAOUT<="1000001001111111101000000";

WHEN"000111"=>DATAOUT<="0100001001111111000000000";

WHEN"001000"=>DATAOUT<="1000111000111111100000000";

WHEN"001001"=>DATAOUT<="1000000001101111100000000";

WHEN"010000"=>DATAOUT<="0100001001111111000000000";

WHEN"001010"=>DATAOUT<="1000101001111111000001011";

WHEN"001011"=>DATAOUT<="1000101001111111000000000";

WHEN OTHERS =>DATAOUT<="0100001011111110000000000";

END CASE;

UA(5 DOWNTO 0)<=DATAOUT(5 DOWNTO 0);

D(18 DOWNTO 0)<=DATAOUT(24 DOWNTO 6);

END PROCESS;

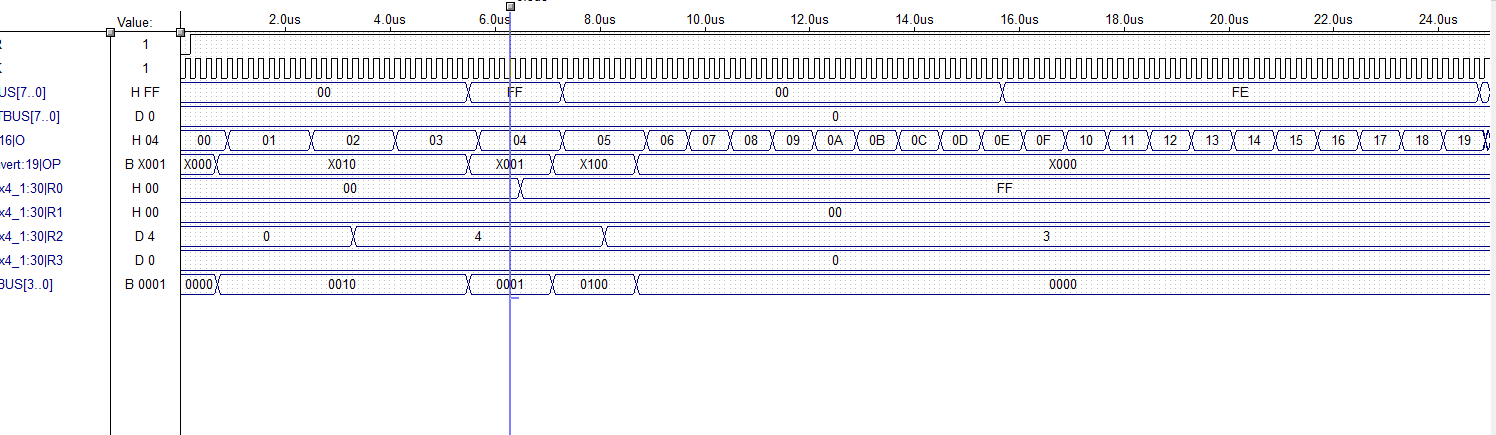
2018年4月30日02:39:38

1. 真的很坑啊，出现了一个奇怪的问题，想了一晚上到现在终于知道怎么解决了

仿真图里面要现在汇编指令的操作码，师兄选择的是|convert:19|OP

他的仿真结果里面，这个输出就是正常的，我自己写了个汇编程序，仿真一下，发现很多xx，也就是说值未知，然后我仔细检查了一下，我改变的地方只有rom和controm，测试后发现是改了rom导致的，然后我将rom里面的汇编指令删减到只有原来的第一条，即MOV R1,00H

然后仿真结也有问题，这个|convert:19|OP全是XXXX,真的很奇怪，这个程序的执行流程我都已经非常的清楚了，哪个时刻执行什么都想明白了，可是为什么不行呢，然后我逐条地从原来的汇编里面删除，发现删除某些指令后，显示不正常。比如IN1 R0,可是这又是为什么呢？我又问了下室友，他们看了下都不知道是为什么。我不搞明白不罢休。后来我尝试在top里面添加一输出引脚连接到|convert:19|OP，然后在仿真里面观察它，奇怪的结果出来了，如果用师兄的完整汇编代码，新添加的和|convert:19|OP结果是一模一样的，如果删除某些汇编指令，那么|convert:19|OP显示不正常，而新添加的输出引脚显示却正常。这下能正常观察汇编的操作码了，但是为什么|convert:19|OP不行呢



2018年4月30日14:14:12

1. 从昨天到现在，发现了一个问题，导致我做的东西都没有用。本来想设计一个新的指令，以便和师兄的不一样。于是决定设计一个比较立即数和寄存器中数据大小的指令CMP1,本本质上就是将立即数先存入一个没用的寄存器，再和别的寄存器想比较。也就是说吧原来的MOV和CMP合并成一条指令CMP1了。可是到现在才发现根本实现不了，除非改原理图。因为，要把一个立即数写入寄存器，就得设置目的寄存器的控制引脚I8,I9，因为2-4译码器的输入控制是I8I9来控制的，而I10I11是控制源寄存器的。也就是说如果我写一个指令CMP1 R2,00H 想把R2的值和0这个数相比较，先把0赋值给别的讲寄存器,比如R3,那么I9I8就为11，控制I11I10为10，那么通过ALU的就是R2减去00H,如果是CMP 00H,R2呢，00H先赋值给R3,结果还是R2减00H,没法比较。

还有我原来的汇编指令也设计的有问题

CMP1 R2,00H 和0相比，判断大小 1010111000000000 00000100

因为没有分清源寄存器和目的寄存器，给R3赋值，应该给I9I8设置为11,I11I12设置为10，而我设置反了。导致在仿真的时候观察R3的值为0，在8.5us的时候R2却变成了输入的数据。

**2018年4月30日15:03:31**

SE5<='1';

SE4<=NOT((((CF AND (NOT ZF)) AND P2) OR(ZF AND P3)) AND T4);

SE3<=NOT(I15 AND P1 AND T4);

SE2<=NOT(I14 AND P1 AND T4);

SE1<=NOT(I13 AND P1 AND T4);

SE0<=NOT(I12 AND P1 AND T4);

SE3= NOT ((((CF AND NOT ZF) AND P2) OR (P1 AND I15) )AND T4)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **微地址** | **P3** | **LOAD** | **LDPC** | **LDAR** | **LDIR** | **LDRI** | **LDPSW** | **RS\_B** | **S1** | **S0** | **ALU\_B** | **SW-B** | **LED\_B** | **RD\_D** | **CS\_D** | **RAM\_B** | **CS\_I** | **ADDR\_B** | **P1** | **P2** | **Ua5-Ua0** |
| **000000** | **0** | **1** | **1** | **0** | **1** | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **1** | **1** | **0** | **000000** |
| **000001** | **0** | **1** | **0** | **0** | **0** | **1** | **0** | **1** | **0** | **0** | **1** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** |  |
| **000010** | **0** | **1** | **0** | **0** | **0** | **1** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **0** |  |
| **000011** | **0** | **1** | **0** | **0** | **0** | **1** | **0** | **0** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** |  |
| **000100** | **0** | **1** | **0** | **0** | **0** | **1** | **1** | **1** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** |  |
| **000101** | **0** | **1** | **0** | **0** | **0** | **0** | **1** | **1** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** |  |
| **000110** | **0** | **1** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **1** |  |
| **000111** | **0** | **0** | **1** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **0** |  |
| **001000** | **0** | **1** | **0** | **0** | **0** | **1** | **1** | **1** | **0** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** |  |
| **001001** | **0** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **1** | **0** | **1** | **1** | **1** | **1** | **1** | **0** | **0** |  |
| **010000** | **0** | **0** | **1** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **0** |  |
| **001010** | **1** | **1** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **0** | **000000** |

JZ的操作码设置为1010

|  |  |  |
| --- | --- | --- |
| 地址（十六进制） | 汇编语言源程序 | 机器语言源程序 |
| 00 | MOV R1,00H | 0010000100000000 |
| 01 | MOV R2,04H | 0010001000000100 |
| 02 | MOV R3,00H | 0010001100000000 |
| 03 | L1: IN1 R0 | 0001000000000000 |
| 04 | DEC R2 | 0100001000000000 |
| 05 | JZ OUT | 1010000000001101 |
| 06 | CMP R3,R0 | 0101110000000000 |
| 07 | JB L1 | 0110000000000011 |
| 08 | CMP R0,R1 | 0101000100000000 |
| 09 | JB L2 | 0110000000001011 |
| 0A | JMP L1 | 0111000000000011 |
| 0B | L2: MOV1 R0,R1 | 0011000100000000 |
| 0C | JMP L1 | 0111000000000011 |
| 0D | OUT: SUB R3,R1 | 1000110100000000 |
| 0E | OUTPUT R1 | 1001010000000000 |

2018年4月30日21:43:35

MOV R1,11H

MOV R2,4H --输入5个数

L0: IN1 R0

STOI R0,(R1)

INC R1

DEC R2

JB L3

JMP L0

----输入完5个数了

L3:MOV R2,4H --第一轮比较4次

MOV R3,20H

STOI R2,(R3)

--在RAM的20H地址写入4H这个数

LAD (R3),R0 --读取20H地址中的数到R0寄存器

MOV R1,10H

L1: DEC R0

JB L2

INC R1

LAD (R1),R2

LAD (R1+1),R3

CMP R2,R3

JB L1

STOI R3,(R1)

STOI R2,(R1+1)

JMP L1

L2: MOV R3,20H

LAD (R3),R0

DEC R0

MOV R2,01H --和1比较，看20H的数是否减到0了

CMP R0,R2

JB OUT --第4轮减到0，比1小，则比较完成

STOI R0,(R3)

MOV R1,10H

JMP L1

OUT: MOV R1,11H

MOV R0 4H

OUT2: LAD (R1) R2

OUTPUT R2

DEC R0

JB OUT3

INC R1

JMP OUT2

OUT3:NOP

不用JNZ或者JZ指令的原因是，若用了就得增加一个P操作，要增加微指令代码表的列数和地址转移逻辑电路，虽然我做的B类题目已经有3个P操作了，但我还是不想增加一个P操作。

2018年5月1日01:04:22

MOV R1,11H

MOV R2,4H --输入5个数

L0: IN1 R0

STOI R0,(R1)

INC R1

DEC R2

JB L1

JMP L0

----输入完5个数了,通用寄存器随便用

L1:MOV R0,04H --第一轮比较4次

MOV R3,20H

STOI R0,(R3)

--在RAM的20H地址写入4H这个数

MOV R1,10H

L2: DEC R0

JB L3

INC R1

LAD (R1),R2

INC R1

LAD (R1),R3

DEC R1

CMP R2,R3

JB L2

STOI R3,(R1)

INC R1

STOI R2,(R1)

DEC R1

JMP L2

L3: MOV R3,20H

LAD (R3),R0

DEC R0

MOV R2,01H --和1比较，看20H的数是否减到0了

CMP R0,R2

JB OUT --第4轮减到0，比1小，则比较完成

STOI R0,(R3)

MOV R1,10H

JMP L2

OUT: MOV R1,11H

MOV R0,04H

OUT2: LAD (R1) R2

OUTPUT R2

DEC R0

JB OUT3

INC R1

JMP OUT2

OUT3:NOP

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 指令助记符 | 指令格式 | | | | 功能 |
| 15-12 | 11 10 | 9 8 | 7------0 |
| IN1 Rd | 0001 | ×× | Rd | ×××××××× | 输入设备→Rd |
| MOV Rd,im | 0010 | ×× | Rd | im | 立即数→Rd |
| LAD (Rs),Rd | 0011 | Rs | Rd | ×××××××× | ((Rs))→Rd |
| STOI Rs,(Rd) | 0100 | Rs | Rd | ×××××××× | (Rs)→(Rd) |
| INC Rd | 0101 | ×× | Rd | ×××××××× | (Rd)+1→Rd,锁存CF,ZF |
| DEC Rd | 0110 | ×× | Rd | ×××××××× | (Rd)-1→Rd,锁存CF,ZF |
| CMP Rs,Rd | 0111 | Rs | Rd | ×××××××× | (Rs)-(Rd),锁存CF，ZF |
| JB addr | 1000 | ×× | ×× | addr | 若小于，则addr→PC |
| JMP addr | 1001 | ×× | ×× | addr | add→PC |
| OUTPUT Rs | 1010 | Rs | ×× | ×××××××× | (Rs)→输出设备 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **微地址** | **P3** | **LOAD** | **LDPC** | **LDAR** | **LDIR** | **LDRI** | **LDPSW** | **RS\_B** | **S1** | **S0** | **ALU\_B** | **SW-B** | **LED\_B** | **RD\_D** | **CS\_D** | **RAM\_B** | **CS\_I** | **ADDR\_B** | **P1** | **P2** | **Ua5-Ua0** |
| **000000** | **0** | **1** | **1** | **0** | **1** | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **1** | **1** | **0** | **000000** |
| **000001** | **0** | **1** | **0** | **0** | **0** | **1** | **0** | **1** | **0** | **0** | **1** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** |  |
| **000010** | **0** | **1** | **0** | **0** | **0** | **1** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **0** |  |
| **000011** | **0** | **1** | **0** | **0** | **0** | **1** | **0** | **0** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** |  |
| **000100** | **0** | **1** | **0** | **0** | **0** | **1** | **1** | **1** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** |  |
| **000101** | **0** | **1** | **0** | **0** | **0** | **0** | **1** | **1** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** |  |
| **000110** | **0** | **1** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **1** |  |
| **000111** | **0** | **0** | **1** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **0** |  |
| **001000** | **0** | **1** | **0** | **0** | **0** | **1** | **1** | **1** | **0** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** |  |
| **001001** | **0** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **1** | **0** | **1** | **1** | **1** | **1** | **1** | **0** | **0** |  |
| **010000** | **0** | **0** | **1** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **0** |  |
| **001010** | **1** | **1** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **0** | **000000** |

2018年5月1日22:57:34

也许波形文件和gdf文件名要一致，否则无法仿真