自选大实验之 计算机系统

1. 实验目标

本实验的目标是在Nexys4开发板上实现一个简单的计算机系统，能够运行简单的指令，并处理一定量的输入输出。在所有功能开发完毕后，希望能够完成基本的terminal功能，即键盘输入命令，并在显示器上输出结果。以下内容只是设计参考，具体实现时可以根据自己的兴趣选做一部分或者进行裁剪和修改。

1. CPU

CPU部分建议参考32位MIPS指令集。该指令集是经典的RISC指令集，实现起来较为简单。MIPS 32的所有指令长度都是32bit，分为三种基本类型：

* R-Type：含3个寄存器操作数
* I-Type：2个寄存器操作数，及一个16bit立即数操作数
* J-Type：跳转，26bit立即数操作数

格式如下：

../../../Computer%20Architecture/beamer%20slides/02_ISA_Principles/mipsisaa.pdf

其中opcode必定为指令前6bit，源、目的寄存器也都在特定位置出现，所以指令解码非常方便。其中包含以下几类指令：

* ALU指令：可以是2寄存器操作数，结果送入目的寄存器，或一个寄存器一个立即数，结果送入目的寄存器。包括有无符号的加减法，移位，及逻辑操作等等。
* Load Store指令：一般是寄存器+立即数偏移量，读取/写入内存。**注意：**MIPS中所有数据都需要先load进入寄存器才能进行操作，不能像x86一样直接对内存数据进行算术处理。
* 分支与跳转指令：条件分支包括BEQ，BNE等等，根据寄存器内容选择是否跳转（没有flag寄存器）。无条件跳转是26位立即数。JAL用于函数调用，自动将返回地址放入r31寄存器。**注意：**MIPS中有跳转延迟槽的概念，跳转指令下一条指令不管是否跳转都会执行。所以，建议在所有跳转指令后加上一条NOP(全0，add r0,r0,r0)。

MIPS 32共32个32bit的寄存器（5 bit寄存器地址），其中寄存器r0中的内容总是0，寄存器r31中存储函数调用的返回地址（在JAL指令中实现，如果实现该指令的话）。

建议实现以下指令，如果有需要，可以按MIPS指令集自行扩充

|  |  |  |
| --- | --- | --- |
| 指令 | 功能 | 编码 |
| add rd,rs,rt | rs+rt->rd, 无溢出才装入 | 0x0/6,rs/5,rt/5,rd/5,0/5,0x20/6 |
| addu rd,rs,rt | rs+rt->rd, 不判溢出 | 0x0/6,rs/5,rt/5,rd/5,0/5,0x21/6 |
| addi rt,rs,imm | rs+imm->rt, 无溢出装入 | 0x8/6,rs/5,rt/5,imm/16 |
| addiu rt,rs,imm | rs+imm->rt，不判溢出，符号扩展 | 0x9/6,rs/5,rt/5,imm/16 |
| sub rd,rs,rt | rs-rt->rd, 无溢出装入 | 0x0/6,rs/5,rt/5,rd/5,0/5,0x22/6 |
| subu rd,rs,rt | rs-rt->rd, 不判溢出 | 0x0/6,rs/5,rt/5,rd/5,0/5,0x23/6 |
| nor rd,rs,rt | rs nor rt ->rd | 0x0/6,rs/5,rt/5,rd/5,0/5,0x27/6 |
| xori rt,rs,imm | rs 与imm无符号扩展xor -> rd | 0xe/6,rs/5,rt/5,imm/16 |
| slt rd,rs,rt | rs比rt小，rd置一，有符号数 | 0x0/6,rs/5,rt/5,rd/5,0/5,0x2a/6 |
| sltu rd,rs,rt | rs比rt小，rd置一，无符号数 | 0x0/6,rs/5,rt/5,rd/5,0/5,0x2b/6 |
| slti rt,rs,imm | rs与符号扩展imm比较，置rt | 0xa/6,rs/5,rt/5,imm/16 |
| sltiu rt,rs,imm | rs与符号扩展imm比较，置rt | 0xb/6,rs/5,rt/5,imm/16 |
| blez rs,imm | rs<0跳转，imm\*4 | 0x6/6,rs/5,0/5,imm/16 |
| j target | 无条件跳转 target\*4 | 0x2/6,imm/26 |
| lw rt,offset(base) | Base加上offset地址装入rt | 0x23/6, base/5,rt/5,offset/16 |
| sw rt,offset(base) | Rt写入Base加上offset地址 | 0x2b/6, base/5,rt/5,offset/16 |

可以考虑使用最简单的单周期实现，即每条指令在1个周期内完成读指令，解码，计算和写回操作。时钟频率可以调整到系统能够在一个CPU时钟周期内完成所有操作。

总体框图如下：



主程序下分以下几个大的子模块:

* ALU: 参考之前的ALU实现，

|  |  |  |  |
| --- | --- | --- | --- |
| * ALUop<3:0> | 操作类型 | Subctr | ALUctr<2:0> |
| 0 0 0 0 | add | 0 | 1 1 0 |
| 0 0 0 1 | sub | 1 | 1 1 0 |
| 0 0 1 0 | clz | 0 | 0 0 0 |
| 0 0 1 1 | clo | 1 | 0 0 0 |
| 0 1 0 0 | and | 0 | 1 0 0 |
| 0 1 0 1 | slt | 1 | 1 0 1 |
| 0 1 1 0 | or | 0 | 0 1 0 |
| 0 1 1 1 | sltu | 1 | 1 0 1 |
| 1 0 0 0 | nor | 0 | 0 1 1 |
| 1 0 0 1 | xor | 1 | 0 0 1 |

* 寄存器组：采用之前的寄存器实现，包括读和写，地址译码等。
* 指令ROM，采用RAM实现，利用coe文件初始化。具体方式是用*MARS MIPS模拟器编程* [*http://courses.missouristate.edu/KenVollmar/mars/*](http://courses.missouristate.edu/KenVollmar/mars/)，编译后通过mem dump将Code段（及数据段）转换为二进制，然后自编Python小程序将MARS的二进制格式转为coe格式固化在RAM中。具体指令地址起始范围自行确定。如果觉得每次vivado编译麻烦，可以通过串口发送机器码给FPGA，串口程序自动写入指令ROM，然后通过switch reset CPU运行。
* 数据RAM：使用RAM实现，读取可以不用时钟控制，写入可使用下降沿。数据RAM可以用coe文件初始化。具体RAM地址分配自行确定，需要考虑给外设：如键盘和显示器分配特定内存空间。
* PC部分如上图所示，采用一个PC寄存器，在时钟上升沿更新。PC更新源包括3个，Branch，jump，和PC+4，利用控制信号选择。PC部分单独一个模块。由于本实验仅实现blez，所以只用了一位branch信号，同Less，Zero联合判断是否要branch。
* 指令解码部分分成两块。一块是将指令rom输出信号直接分成op,func,rs,rt,rd和imm等。另一块是通过op/func信号输出各类控制信号。

具体的控制信号如下表，控制信号的含义与教材一致

| 指令 | OP | Func | Jump | Branch | Regwr | Ofctrl | Extop | Regdst | Alusrc | ALUctrl | Memtoreg | Memwr |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| add | 00h | 20h | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0000 | 0 | 0 |
| addu | 00h | 21h | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0000 | 0 | 0 |
| addi | 08h | - | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0000 | 0 | 0 |
| addiu | 09h | - | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0000 | 0 | 0 |
| sub | 00h | 22h | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0001 | 0 | 0 |
| subu | 00h | 23h | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0001 | 0 | 0 |
| nor | 00h | 27h | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1000 | 0 | 0 |
| xori | 0eh | - | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1001 | 0 | 0 |
| slt | 00h | 2ah | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0101 | 0 | 0 |
| sltu | 00h | 2bh | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0111 | 0 | 0 |
| slti | 0ah | - | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0101 | 0 | 0 |
| sltiu | 0ah | - | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0111 | 0 | 0 |
| blez | 06h | - | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0001 | 0 | 0 |
| j | 02h | - | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0000 | 0 | 0 |
| lw | 23h | - | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0000 | 1 | 0 |
| sw | 2bh | - | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0000 | 0 | 1 |

建议在CPU调试过程中先分块实现，分别仿真模拟，再整合到一起去。

1. 输入输出

输入输出分为以下功能，每个模块可以是独立的，与CPU并行运行。

* 显示器：实现标准字符显示界面。显示器部分独立于CPU，自动进行扫描控制和字符显示。实现方式是，规定字符界面宽度和高度，如70列\*30行字符，在CPU内存中划出70\*30 Byte区域作为显存，存储要显示字符的ASCII码。CPU可以对该区域写ASCII码，确定要显示什么字符。显示器模块只是每次扫描时读取这块显存，显示对应的字符。具体方式是在扫描时扫到对应位置，读取显存的ASCII码，根据ASCII码计算每个点对应的字模点阵地址，读取字模的RGB值。需要在RAM中预先存储好一套 ASCII字库，内存组织方式请自行确定。附件为一简单的9\*16 ASCII字库。字符覆盖范围是ascii[6:0]，忽略ascii最高位。每个字符是9\*16点阵，每一行以12bit表示（对应bit为1为亮，为0是暗，低bit位在显示器左边），每个字符16行。共128个字符。

可以在显示区域中自行划出一块区域用硬连线的方式来显示CPU当前PC、指令和寄存器状态等等。



* 键盘：复用之前的键盘程序，在内存中开辟一块键盘缓存区（可以仅单个byte），将键盘输入转换为ASCII码存入，并将键盘输入指示置1。
* 七段数码管和LED：可根据自己的调试需要输出。
* 串口控制器：接收主控电脑发送的机器码，按特定位置写入指令ROM。可用一个拨动开关设置编程状态，编程开始时计数器清零，每收到一个byte填入对应ROM位置，编程结束后开关关闭，PC置为程序第一条指令，程序开始运行。
* 拨动开关和按钮：自行设计。为方便调试，可以设置reset按钮，重置CPU的PC和寄存器。可以设置CPU时钟来源，在单步调试时，用按钮作为CPU时钟， 每按一下运行一条指令，正常运行时使用内部时钟。

1. 软件部分

软件部分用MIPS汇编编写，建议下载MIPS模拟器[*http://courses.missouristate.edu/KenVollmar/mars/*](http://courses.missouristate.edu/KenVollmar/mars/) 。

软件编写完成后，可以用模拟器提供的mem dump功能将代码段机器码存储下来，并用python转换成coe格式。如用串口控制器编程，可以转换成二进制格式。如果有数据段，可以类似用mem dump存储并写入FPGA中。

软件功能：

1. 主循环：不停轮询接口状态。如果只处理键盘输入，只需轮询键盘缓冲器和标志位。在有键盘输入时，跳转到键盘输入处理。
2. 键盘处理：将键盘输入存入命令缓冲区，如果有回车键，跳转到命令分析部分。
3. 命令分析：匹配字符串，根据命令执行对应的子程序，并将执行结果输出到屏幕上。执行完成后跳回主循环。扩展需求，需要处理显示换行，滚动等功能。

可以支持简单的命令，如：

* 打入hello，显示Hello World!
* 打入LED 1 on，打开LED 1，打入LED 1 off，关闭LED 1.
* 打入time，显示时间
* 打入简单表达式，如9+1，输出结果
* 打入未知命令，输出 Unknow Command。

示例MIPS 汇编

# Compute first twelve Fibonacci numbers and put in array

.data

fibs: .word 0 : 12 # "array" of 12 words to contain fib values

size: .word 12 # size of "array"

.text

la $t0, fibs # load address of array

la $t5, size # load address of size variable

lw $t5, 0($t5) # load array size

li $t2, 1 # 1 is first and second Fib. number

add.d $f0, $f2, $f4

sw $t2, 0($t0) # F[0] = 1

sw $t2, 4($t0) # F[1] = F[0] = 1

addi $t1, $t5, -2 # Counter for loop, will execute (size-2) times

loop: lw $t3, 0($t0) # Get value from array F[n]

lw $t4, 4($t0) # Get value from array F[n+1]

add $t2, $t3, $t4 # $t2 = F[n] + F[n+1]

sw $t2, 8($t0) # Store F[n+2] = F[n] + F[n+1] in array

addi $t0, $t0, 4 # increment address of Fib. number source

addi $t1, $t1, -1 # decrement loop counter

bgtz $t1, loop # repeat if not finished yet.