**计算组成原理**

**单周期CPU模拟器的实现**

项目到期：11月30日下午23:59

1. **简介**

在这个项目中，您将使用C语言实现一个称为MiniCPU的单周期CPU模拟器。 您的MiniCPU将演示MIPS处理器的一些功能以及数据路径和控制信号的原理。 MiniCPU应该读入包含MIPS机器代码（以下指定格式）的文件，并逐周期模拟MIPS处理器的工作。 将为您提供一个名为component.c的C文件，该文件实现了单周期数据路径的每个组件，您需要修改并填写此文件中的函数主体。

1. **模拟器规格**

**2.1模拟指令**

附录中的图1列出了14条指令。 请注意，您无需处理导致异常的情况。

* 1. **登记要处理**

MiniCPU应该处理32个通用寄存器。 在程序开始时，将寄存器初始化为minicpu.c中指定的值

* 1. **内存使用情况**
     + - MiniCPU的内存大小为64kB（地址0x0000至0xFFFF）。
       - 系统假定所有程序都从内存位置0x4000开始。
       - 所有指令在存储器中都按字对齐，即所有指令的地址均为4的倍数。
       - 模拟器（和MIPS处理器本身）将内存视为一个段。 （将内存分为文本，数据和堆栈段只能由编译器/汇编器完成。）
       - 在程序启动时，除“ -asc”文件中指定的内存外，所有内存均初始化为零，如提供的代码所示。
       - 内存采用以下格式：

例如 将32位数字0xaabbccdd存储在内存地址0x0 – 0x3中。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Mem[0] | | | |
| Address | 0x0 | 0x1 | 0x2 | 0x3 |
| Content | aa | bb | cc | dd |

* 1. **MiniCPU应该停止的条件**

如果遇到以下情况之一，则全局标志Halt设置为1，因此模拟停止。

* + - * 遇到非法指令。 图1中超出指令列表的指令是非法的。
      * 跳转到未字对齐的地址（4的倍数）
      * lw或sw的地址未按字对齐
      * 访问数据或跳转到内存以外的地址。
  1. **输入机器代码文件的格式**

MiniCPU将十六进制格式的机器代码（文件名为xxx.asc）作为输入。 .asc文件的示例如下所示。 任何行上“＃”之后的代码均视为注释。

20010000 #addi $1, $0, 0  
200200c8 #addi $2, $0, 200  
10220003 #beq $1, $2, 3  
00000020 #delay slot  
20210001 #addi $1, $1, 1  
00000020 #no operation

当遇到非法指令（例如0x00000000）时，模拟结束。

* 1. **分支寻址注意事项**

MIPS以及MiniCPU中的分支偏移与下一条指令（PC + 4）有关。 例如：

**Assembly code**

beq $1, $2, label

beq $3, $4, label

label: beq $5, $6, label

**Machine codes**

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 1 | 2 | 0x0001 |

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 3 | 4 | 0x0000 |

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 5 | 6 | 0xffff |

Opcode Rs Rt Offset

6 bits 5 bits 5 bits 16 bits

1. **资源**

**3.1 提供的文件**

请从ftp://ftp.must.edu.mo/下载project.zip，以下是解压缩后的文件：

minicpu.c minicpu.h component.c minicpuasm.pl incommand

test01.asm test01.asc test02.asm test02.asc

这些文件包含模拟器的主程序和其他支持功能。 该代码应该是不言自明的。 您需要填写并修改component.c中的功能。 **您不允许修改minicpu.c和minicpu.h。 您的所有作品都应仅放在component.c中。 您无权添加新文件。 否则，您的程序将不会得分。**

* 1. **MIPS汇编器**

提供一个简单的汇编程序minicpuasm.pl是为了方便您测试MinCPU。 该命令是：

$minicpuasm.pl filename.asm > filename.asc

其中filename.asm是您的汇编代码文件，而filename.asc是十六进制格式的输出机器代码文件。

1. **component.c中的功能**

首先，您需要在component.c中完成一个模拟ALU操作的函数（ALU（…））

**void ALU(unsigned A, unsigned B, char ALUControl, unsigned \*ALUresult, char \*Zero)**

**{**

**if(ALUControl==0x0)\*ALUresult=A+B; //add**

**}**

* ALU(…)

1.根据ALUControl对输入参数A和B进行操作。

2.将结果输出到ALUresult。

3.如果结果为零，则将0分配给1；否则，为0。 否则，分配0。

4.下表显示了ALU的操作。

|  |  |
| --- | --- |
| ALUControl | Meaning |
| 000 | Z = A + B |
| 001 | Z = A – B |
| 010 | if A < B, Z = 1; otherwise, Z = 0 |
| 011 | if A < B, Z = 1; otherwise, Z = 0 (A and B are unsigned integers) |
| 100 | Z = A AND B |
| 101 | Z = A OR B |
| 110 | Shift B left by 16 bits |
| 111 | Z = NOR(A,B) |

其次，您需要在component.c中填写9个函数。 每个函数都模拟数据路径的一部分的操作。 附录中的图2显示了数据路径以及您需要模拟的数据路径部分。

在minicpu.c中，函数Step（）是MiniCPU的核心功能。 该函数调用您需要实现的9个函数，以模拟在数据路径的组件之间传递的信号和数据。 彻底阅读Step（）以便理解信号和数据传递，并实现9个功能。

* instruction\_fetch(…)

1.从内存中提取PC寻址的指令并将其写入指令。

2.如果遇到无效指令，则返回1；否则，返回1。 否则，返回0。

**int instruction\_fetch(unsigned PC, unsigned \*Mem, unsigned \*instruction)**

**{ \*instruction=Mem[PC>>2];**

**return 0;**

**}**

* instruction\_partition(…)

1.将指令分为几部分（op，r1，r2，r3，funct，offset和jsec）。

2.阅读minicpu.c的41至47行以获取更多信息。

**void instruction\_partition(unsigned instruction, unsigned \*op, unsigned \*r1, unsigned \*r2, unsigned \*r3, unsigned \*funct, unsigned \*offset, unsigned \*jsec)**

**{**

**\*op = instruction >> 26;**

**}**

* instruction\_decode(…)

1.根据操作码（op）解码指令。

2.为结构控件中的变量（控制信号）分配适当的值。

控制信号值的含义：

对于MemRead，MemWrite或RegWrite，值1表示启用，0表示禁用，2表示“无关”。

对于RegDst，Jump，Branch，MemtoReg或ALUSrc，值0或1表示多路复用器的选定路径； 2表示“无关”。

下表显示了ALUOp值的含义。

|  |  |
| --- | --- |
| value (binary) | Meaning |
| 000 | ALU will do addition or “don’t care” |
| 001 | ALU will do subtraction |
| 010 | ALU will do “set less than” operation |
| 011 | ALU will do “set less than unsigned” operation |
| 100 | ALU will do “and” operation |
| 101 | ALU will do “or” operation |
| 110 | ALU will shift left *extended\_value* by 16 bits |
| 111 | The instruction is an R-type instruction |

3.如果发生停止情况，则返回1；否则，返回1。 否则，返回0。

**int instruction\_decode(unsigned op, struct\_controls \*controls)**

**{**

**if(op==0x0){ //R-format**

**controls->RegWrite = 1;**

**controls->RegDst = 1;**

**controls->ALUOp = 7;**

**}**

**else return 1; //invalid instruction**

**return 0;**

**}**

* read\_register(…)

1.从Reg读取r1和r2寻址的寄存器，并将读取的值分别写入data1和data2。

**void read\_register(unsigned r1, unsigned r2, unsigned \*Reg, unsigned \*data1, unsigned \*data2)**

**{**

**\*data1 = Reg[r1];**

**\*data2 = Reg[r2];**

**}**

* sign\_extend(…)

1.将offset的符号扩展值分配给extended\_value。

**void sign\_extend(unsigned offset, unsigned \*extended\_value)**

**{**

**}**

* ALU\_operations(…)

1.基于ALUOp和funct，对data1，data2或extended\_value执行ALU操作。

2.调用函数ALU（…）执行实际的ALU操作。

3.将结果输出到ALUresult。

4.如果发生停止情况，则返回1；否则返回1。 否则，返回0。

**int ALU\_operations(unsigned data1, unsigned data2, unsigned extended\_value, unsigned funct, char ALUOp, char ALUSrc, unsigned \*ALUresult, char \*Zero)**

**{**

**switch(ALUOp){**

**// R-type**

**case 7:**

**// funct = 0x20 = 32, add**

**if (funct==0x20) ALU(data1, data2, 0x0, ALUresult, Zero);**

**else return 1; //invalid funct**

**break;**

**default:**

**return 1; //invalid ALUop**

**}**

**return 0;**

**}**

* rw\_memory(…)

1.根据MemWrite或MemRead的值确定内存写操作或内存读操作。

2.读取ALUresult寻址到memdata的内存位置的内容。

3.将data2的值写入ALUresult寻址的存储位置。

4.如果发生停止情况，则返回1；否则返回1。 否则，返回0。

**int rw\_memory(unsigned ALUresult, unsigned data2, char MemWrite, char MemRead, unsigned \*memdata, unsigned \*Mem)**

**{**

**if (MemRead==1){**

**\*memdata = Mem[ALUresult>>2];**

**}**

**return 0;**

**}**

* write\_register(…)

1. 将数据（ALUresult或memdata）写入由r2或r3寻址的寄存器（Reg）。

**void write\_register(unsigned r2, unsigned r3, unsigned memdata, unsigned ALUresult, char RegWrite, char RegDst, char MemtoReg, unsigned \*Reg)**

**{**

**Reg[r2] = memdata;**

**}**

* PC\_update(…)

1. 更新程序计数器（PC）。

**void PC\_update(unsigned jsec, unsigned extended\_value, char Branch, char Jump, char Zero, unsigned \*PC)**

**{**

**\*PC+=4;**

**}**

文件minicpu.h是头文件，其中包含存储控制信号的结构的定义以及上述功能的原型。 这些功能可能包含一些参数。 阅读minicpu.h以获取更多信息。

1. **注意事项**

1.该项目将使用Dev C ++进行编译和标记。您可以从网站（http://www.bloodshed.net/dev/devcpp.html）下载它，并将其安装在计算机上。请记住，您应该下载并安装用于C / C ++的Dev C ++。

2.一些指令可能会尝试写入寄存器$ zero，我们假设它们是有效的。但是，您的模拟器应始终保持$ zero 0的值。

3.您不应该在component.c中执行任何“打印”或“ printf（）”操作；否则，该操作会打乱标记过程，并扣除您的标记。

4.运行已编译的可执行文件：

A．在Windows中，打开命令提示符。

B．转到您的工作目录。

C．类型：your\_executable input\_asc\_file <incommand>

其中incommand是下载的文件。输出显示所有寄存器和存储器位置的值，使您可以检查模拟器是否可以产生正确的结果。

5.要调试程序，请检查所有寄存器和存储器的值是否与汇编程序匹配。以下是示例输出：

cmd:

cont

cmd:

$zero 00000000 $at 00000000 $v0 00000000 $v1 00000000

$a0 00000000 $a1 00000000 $a2 00000000 $a3 00000000

$t0 00000002 $t1 00000003 $t2 00000005 $t3 00000001

$t4 00000002 $t5 00000003 $t6 00000002 $t7 00000000

$s0 00010000 $s1 00000001 $s2 00000001 $s3 00000001

$s4 00000001 $s5 00000000 $s6 00000000 $s7 00000000

$t8 00000000 $t9 00000000 $k0 00000000 $k1 00000000

$gp 0000c000 $sp 0000fffc $fp 00000000 $ra 00000000

$pc 00004034 $stat 00000000 $lo 00000000 $hi 00000000

cmd:

00000 00000002

00004-03ffc 00000000

04000 20080002

04004 20090003

04008 01285020

0400c 01285822

04010 01286024

04014 01286825

04018 ac080000

0401c 8c0e0000

04020 3c100001

04024 0109882a

04028 0109902b

0402c 29130003

04030 2d140003

04034-0fffc 00000000

cmd:

quit

1. **要提交您的工作，请在组件的开头component.c，键入您的英文名称，例如**

/\*

\* Designer: name, student id, email address

\*/

Email your *component.c* to [yyliang.fit.must@gmail.com](mailto:yyliang.fit.must@gmail.com). The subject of your email must be: **CO101 Project student\_name**.