《计算机结构与组成》课程设计

一共两道题：

题 1 为软件设计，在已有代码框架下，完成一个 MIPS 汇编器的初步开发。（70 分）

题 2 为硬件设计，设计一个简单的 CPU 逻辑电路。（完成基础部分 30 分，全部实现获得附加分 20 分）基础部分：寄存器组，加法器/减法器，指令解码

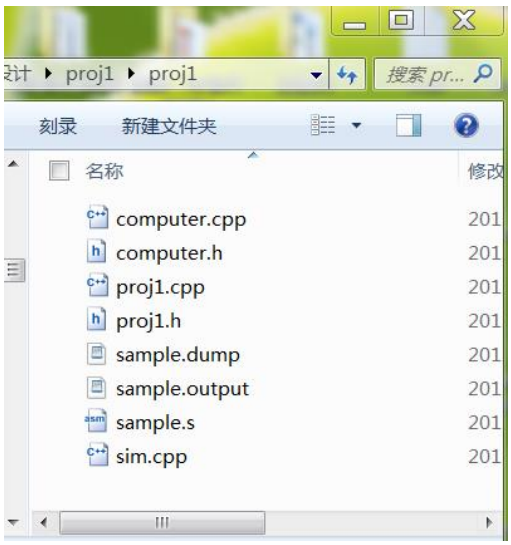
要求：独立完成，最终按照格式模板撰写报告，并打印。每位同学需要将自己的课程设计文档（报告 word 文档，C++代码，电路设计文件等）放在自己的文件夹下（格式：班序号＋姓名，如 11116101 胡鑫）提交给本班学习委员，学习委员将本班文档一起刻光盘。学习委员将打印报告收齐，请按照同学们的班级序号将报告排好顺序和本班光盘一起，在 2018 年 1 月 3 日上午交到教 1 楼 301。

题一

需要独立完成。开发结果需要提交一个 C++实现文件 proj1.cpp.

如何开始

将压缩文件包 proj1 解压到某个文件夹，并将相关文件夹命名为"proj1". 该文件夹包含文件: sim.cpp, computer.h, computer.cpp, proj1.cpp, proj1.h, sample.s, sample.dump, sample.output 等。这些文件描述详见下面。



背景材料

本题你需要构建 MIPS 汇编语言一个子集的指令解释器. 它将实现取指、反汇编, 解码, 并执行 MIPS 机器指令。准确的说法是，你将构建一个简化版的 MARS! 当然，本题和 MARS 有一个非常重要的区别。MARS 输入为汇编语言, 而本次作业的输入是.dump 文件（即机器码程序），因此，MARS 还包含一个汇编器。

内容

以下文件：sim.cpp, computer.h, computer.cpp, proj1.cpp and proj1.h 组成了 MIPS 仿真器的一个框架. 通过在文件 proj1.cpp 中增加代码来完成整个程序。你的仿真器必须能仿真以下 MIPS 机器指令:

```
addu    Rdest, Rsrc1, Rsrc2
addiu   Rdest, Rsrc1, imm
subu    Rdest, Rsrc1, Rsrc2
sll     Rdest, Rsrc, shamt
srl     Rdest, Rsrc, shamt
and     Rdest, Rsrc1, Rsrc2
andi    Rdest, Rsrc, imm
or      Rdest, Rsrc1, Rsrc2
ori     Rdest, Rsrc, imm
lui     Rdest, imm
slt     Rdest, Rsrc1, Rsrc2
beq     Rsrc1, Rsrc2, raddr
bne     Rsrc1, Rsrc2, raddr
j       address
jal     address
jr      Rsrc
lw      Rdest, offset (Radd)
sw      Rsrc, offset (Radd)
```

一旦完成, 你的程序应能仿真所有在 MIPS 上能运行的任何真实程序, 当然除浮点数和中断外。

框架代码

框架代码首先做下面这些事情.

1. 将机器码读入"内存", 从"地址" 0x00400000 处开始执行。(与MARS约定相同, 从0x0000000到 0x00400000 的地址未用) 假定程序的长度不超过 1024 个字。包含源代码的程序通过命令行给定.
2. 栈指针初始化为 0x00404000, 其他所有寄存器初始化为 0x00000000, 并将程序计数器(PC)初始化为 0x00400000.
3. 数据内存的首地址为 0x00401000, 末地址为 0x00404000. 在同一内存数组中存储数据和指令.
4. 设置用于管理整个程序如何与用户交互的标识(flags).

然后, 进行一个循环, 反复取指和执行指令, 在程序运行中打印(printing)相关信息:

- 所执行的机器指令, 其地址, 反汇编形式;
- 如果发现不支持的指令, disassembled(反汇编)程序应调用 exit(0) i;
- 程序计数器(PC)的新值;
- 当前寄存器状态的信息;
- 内存内容的信息.

你的任务是编写 disassembled (反汇编)和 simulateInstr(指令仿真)函数的代码, 当然可能还有一些相关的你希望用到的辅助函数的代码。可以假设仿真指令不会出现寻址错误. 如果程序遇到没有列出的指令, 它将退出.

框架程序支持以下命令行选项:

-i	以 "交互 interactive 方式"运行代码. 在此模式下, 仿真每个指令前, 程序输出">"提示符, 等待你输入回车继续。如果你输入"q" (意为"quit")加一个回车, 程序将退出. 如果不指定此选项, 终止程序的唯一方式是仿真一个上面没有列出的指令。
-r	在执行一个指令后列出所有寄存器的值. 当未指定此选项时, 只列出指令所影响的寄存器的值; 对于分支, 跳转, 写内存这些不影响任何寄存器的指令, 框架代码输出没有寄存器受影响的信息.(当仿真的指令不影响任何寄存器时, 你的代码需要给出一个信号, 方法是在 changedReg 的参数中返回一个合适的值给 simulate 和 simulateInstr.)
-m	在执行一个指令后列出所有包含非零值的内存地址的值. 当未指定此选项时, 只列出指令所影响的内存地址的指; 对于所有非 sw 指令,框架代码输出没有内存地址受影响的信息. (当仿真指令不影响内存时, 你的代码需要给出一个信号, 方法是在 changedMem 参数中返回一个合适的值给 simulate 和 simulateInstr.)
-d	这是调试标识, 你会发现它很有用的.

在框架代码中的几乎所有输出都是在 printInfo 函数中完成的. 另外,disassembled 函数的结果(output)也会输出, 这需要特别注意。尽管只是以文本方式输出指令和操作数, 由于我们使用自动脚本来给你判分. 因此, 结果(output)必须严格遵守下面的说明。下面是详细的输出格式:

- 反汇编的指令的指令名后必须跟一个"跳格字符 tab" (C 语言中字符为'\t'), 指令后的各项用逗号+空格来分隔.
- 对指令 addiu, srl, sll, lw 和 sw, 立即数的值必须以十进制数输出 (如果为负数, 前面要有负号) 不能有前导 0, 除非为 0 (输出 0).
- 对指令 andi, ori, 和 lui, 立即数必须为十六进制, 且前面有 0x, 并且没有前导 0, 除非值为 0 (输出 0x0).

- 对于分支和跳转指令，目标地址必须以完整的 8 位 16 进制数输出，包含前导 0.（请注意此格式和分支跳转汇编语言指令的区别） 最后，分支和跳转的目标地址必须为绝对地址，而不是与 PC 相关的.
- 所有 16 进制值必须使用小写字母，且没有前导 0x.
- 参数字段之间的分隔为逗号+空格.
- 寄存器以数字方式指定，没有前导 0（如$10 and $3），而不能以名字方式给定(如$t2).
- 以存储字节(sb)指令为例，需要返回"sb\t$10, -4($21)".
- 你有责任管理所有你返回字符串关联的内存. 如果使用 malloc()，其后你需要以某种方式调用 free() 来释放内存（注意不能修改 computer.cpp）. 提示：静态或全局变量.
- 如果 lw 或 sw 指令访问无效内存地址，你的代码必须输出和 computer.cpp 中的 contents()函数完全相同的错误信息，然后调用 exit(0)

以下是正确(good)的指令示例:

```
addiu     $1, $0, -2
lw        $1, 8($3)
srl       $6, $7, 3
ori       $1, $1, 0x1234
lui       $10, 0x5678
j         0x0040002c
bne       $3, $4, 0x00400044
```

以下是错误(bad)指令的示例:

```
addiu     $1, $0, 0xffffffff      # addiu 不应输出 16 进制
sw        $1, 0x8($3)             # sw 不应输出 16 进制
sll       $a1, $a0, 3             # 应使用寄存器号，而不是寄存器名
srl       $6,$7,3                 # 参数间缺空格
ori       $1 $1 0x1234            # 缺逗号
lui       $t0, 0x0000ABCD         # 16 进制数应该小写且 not zero extended
j         54345                   # 地址应为 16 进制
jal       00400548                # 缺前导 0x
bne       $3, $4, 4               # 需要完整的 16 进制目标地址
```

文件 sample.s 和 sample.output 中提供了一个输出示例，你可以用它来做一个基本测试"sanity check". 我们没有提供其他输入测试文件. 你必须自己在编写 MIPS 汇编测试用例，使用 MARS (mars-cs61c)来汇编，然后输出（dump）二进制代码. 参见"提示"中关于如何做的建议.

请不要修改框架程序. 如果我们发现其有错误，我们将提供一个新版本.

提示

很多同学都想马上编写出整个代码，然后测试。这样做调试会比较困难。以下是我们建议的完成本项目的方法:

1. 仔细阅读和理解项目规范及源程序.
2. 在编写 C 代码之前，用 MIPS 汇编编写一个简单的测试用例，该用例只测试指定的一条指令；在 MARS 中汇编成二进制代码（见下面）. 注意有些指令对依赖于更多的其他代码。例如，lw 和 sw 指令需要内存代码能正确工作。请在选择指令时考虑这些情形。
3. 对上一步所选取的指令，选择某条具体指令来实现，实现该指令使用的 C 语言代码最少，然后使用测试用例来测试，第一步也许需要编写测试用例所需要的反汇编助手函数.
4. 选择另一个指令来测试. 合理的做法也许是选择与第一个相类似的一组指令来实现。编写第二个测试文件，该文件只使用你所实现的指令集。
5. 重复第 3 步和第 4 步，直到能支持所有指令.
6. 让你的代码更清晰一些。注意保证缩进正确，并添加注释，这对调试和理解所编写的代码会有帮助。
7. 提交你的解答。

编写测试用例

测试的第一步是用 MIPS 汇编语言编写测试程序. 下面是如何编写 MIPS 测试代码的一些重要指南:

- 深吸一口气，忘记你要编写一个模拟器。编写一个简单、短小、并且有意义的 MIPS 程序。程序越有意义，越容易检验仿真器能工作。
- 显然，编写测试代码时应只使用已经支持的指令。
- 我们的仿真器中，程序从内存地址 0x400000 开始，数据从内存地址 0x401000 开始，栈始于 0x00404000。 MARS 初始化栈指针为 0x7fffeffc. 当编写测试程序时，请确保此差异不会产生问题。
- MARS 将跟随在汇编指示器.data 后的所有内容顺序放到内存中。但这不会反映到 MARS dumps 出来的二进制文件中。此 dump 文件将只包含指令。因此，在使用 MARS 时，MARS 会帮你把数据装入到内存中，现在你必须用指令自己装入。
  举例来说，假定你想编写一个 MIPS 程序 foo，该程序使用 5 个字的数组，数组元素初始化为整数 1, ..., 5. 正常的情况下，你应该编写象如下形式的代码:

```
        .data
foo:    .word 1,2,3,4,5
```

  本项目，你不能使用.data 段. 取而代之的是你应该让你的程序来初始化数组:

```
__start:
    lui     $t0, 0x1001
    ori     $t0, 0x0000
    addiu   $t1, 1
    sw      $t1, 0($t0)
    addiu   $t1, 2
    sw      $t1, 4($t0)
    addiu   $t1, 3
    sw      $t1, 8($t0)
    addiu   $t1, 4
    sw      $t1, 12($t0)
    addiu   $t1, 5
    sw      $t1, 16($t0)
```

你也可以使用循环来完成。这看来是件麻烦又花时间的事，但它将大大减化模拟器编程.

- 验证测试用例的结果。 例如, 如果测试程序从内存的一块区域复制一段数据到另一块区域, 在程序结束后，你应该知道需要看内存的哪一段来验证是否正确。命令行参数-r, -m, 和 -c 对产生正确的结果很有用，它能为模拟器提供正确性的依据。

生成二进制文件

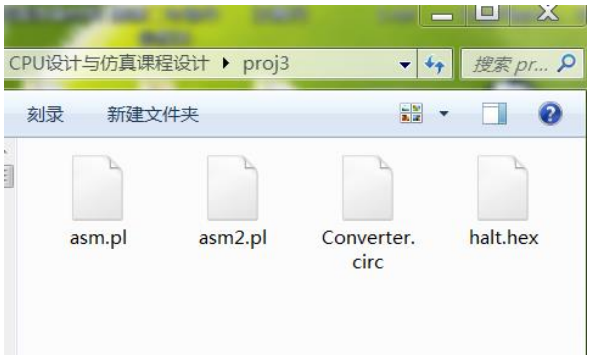
可以直接为模拟器生成过程代码。以下是具体步骤:

1. 用 MIPS 汇编编写测试程序. 确保只使用所支持的指令, 不要使用汇编指示来建立.data 内存. 假定你的测试文件命名为 test0.s. 使用单个不支持的指令(如 slti)来终止程序. 如前所述, 执行不支持的指令将使模拟器退出.
2. 在你的机器上使用 Mars 到你的本机上.
3. 使用 MARS 调试你的 MIPS 代码.
4. Dump the code from MARS using the Save Dump button in the lower left of the data segment window. Be sure to select the .text segment in the dropdown before doing this, or you might accidentally dump the data memory. You should also select the Binary dump format for this project. MARS will dump the binary instruction into a file of your choosing.
5. Now, test0.dump is a valid input to the simulator. We recommend that you save all of your .s and .dump test files in an orderly fashion. This way, when you think you are done with the simulator, you will have a comprehensive battery of tests to put it through before submitting.

题二

概述(Overview)

本工程需要用到的全部文件在 proj3 文件夹里面:



本题需要使用 Logisim 来创建一个 16-位单时钟周期 CPU. 该 CPU 是一个全新的 CPU,与老师上课所讲的有很大的不同,因此,请*仔细* 阅读本文档，其中包含了很多(课上没有讲的)新内容，同时也回答了各种可能的疑问。

解题步骤指导

第一步:可以通读一下本指南

第二步:做一下相关的实验,了解 LOGISIM 的一些模块和组件,如 RAM 和显示单元(后面有专门介绍),尽量深入地理解相关模块的用法及原理.

第三步:可以制作一个寄存器组(也称寄存器文件)模块(组件).作为第一小步你可以先放四个寄存器,然后,做出一个输出.即可选择四个寄存器中的任意一个的值输出.接下来,你可以再做一个输出,然后完成输入,即将外部的值写到某个寄存器中.在完成该寄存器文件的设计后,你应该认真的检查,其是否满足设计的要求,即需要进行测试.

第四步:可以制作一个 ALU,该 ALU 暂时可以仅实现 ADD,SUB,AND,OR 四种运算,将来再扩展更多功能.(对于 ADD,SUB 等有两种办法实现,其一是自己用基本的与或非门电路搭建,其二是直接使用 LOGISIM 自己提供的内建库实现)

第五步:可以制作一个下一条指令的逻辑,即 PC(程序计数寄存器)的逻辑,最初你可以简单的实现,只用让该寄存器的值,在每次时钟信号到达时加一即可，以后再考虑更复杂的情形。

第六步：也许此时你可以用上面构建的模块，搭建一个最简单的 CPU 了，该 CPU 只能进行加法运算。你可以先做一个大致的模型，实现加法。而我们希望是，你的 CPU 应该包括以下器件：

1）寄存器文件
2）PC 寄存器及每时钟周期 PC+1 的逻辑（需要认真阅读一下后面的说明）
3）ALU
4）指令内存（为了简单，建议你使用系统提供的 ROM，而不是 RAM）

试着将以下汇编指令翻译成机器码：
add $2,$1,$0
add $1,$1,$0
add $1,$2,$0
然后将机器码存入指令内存中。

如果你成功地完成了此最简单的 CPU，则时钟每跳一次，将执行一条指令。（当然，为了你看到一些有意义的结果，你可能需要手动设置一下各个寄存器的初值,如将寄存器 1 的值设为 1，寄存器 2 的值设为 2，寄存器 3 的值设为 3）

第七步：你可以进一步编写一些新的 CPU 指令，如 SUB，AND，OR 等，来对你前面做的工作进行一些进一步的测试。当然，对于 SUB，AND，OR，你可能需要修改 ALU 的相应控制，否则，还是在做加法。

第八步：你可以编写更多的模块，如零扩展和符号扩展，然后将其组合成一个通用的扩展器。来实现有立即数的运算了。

第九步：你可以加入 LOAD 及 STORE 逻辑。当然，这里需要加入数据内存了。

第十步：也许，你可以把所有的工作集成在一起，完成 DATAPATH（数据通道）了。

第十一步：设计 CONTROLPATH（控制通道）

最后，你的 CPU 就完成了。

指令集结构 Instruction Set Architecture (ISA)

需要实现一个简单的 16-位处理器(即每个指令字长为 16 位,寄存器也是 16 位)，该处理器有四个寄存器($r0 到$r3). 具有独立的数据和指令内存(即有两个内存,一个指令内存,一个数据内存)。

重要注意事项：由于 Logisim 的限制，也为了让事情更简单一些，我们以半字(16 位)为单位 对内存编址！这和 MIPS 不同，MIPS 指令是字长是 32 位,而内存是以字节(8 位)为单位编址.

下表给出了指令编码表。通过查询 opcode 字段(高四位，即 15-12 位)的值，可知半字编码所对应的指令 . 注意，表中的 opcode 不到 16 个，而 funct 也不到 8 个. 原因是指令少一些，使同学们更容易实现(呵呵,好象比老师上课讲的 CPU 指令数还是多了很多).

15-12	11	10	9	8	7	6	5	4	3	2	1	0	
0	rs		rt		rd		party bits!		funct				参见 R-type Instructions
1	rs		rt		immediate-u								disp: DISP[imm] = $rs
2	rs		rt		immediate-u								lui: $rt = imm << 8
3	rs		rt		immediate-u								ori: $rt = $rs | imm
4	rs		rt		immediate-s								addi: $rt = $rs + imm
5	rs		rt		immediate-u								andi: $rt = $rs & imm
6	rs		rt		immediate-s								lw: $rt = MEM[$rs + imm]
7	rs		rt		immediate-s								sw: MEM[$rs+imm] = $rt
8	jump address												jump
9	rs		rt		offset								beq
10	rs		rt		offset								bne

R-Type Instructions

funct	meaning
0	or: $rd = $rs | $rt
1	and: $rd = $rs & $rt
2	add: $rd = $rs + $rt
3	sub: $rd = $rs - $rt
4	sllv: $rd = $rs << $rt

5	srlv: $rd = $rs >> $rt
6	srav: $rd = $rs >> $rt
7	slt: $rd = ($rs < $rt) ? 1 : 0

注意:这里 opcodes 的顺序和作业 hw6 是不同的. 这样做的原因是，我们希望全零的指令(0x0000)为 NOP, 因此，将 or 置为 funct 值为零的指令.

srl vs. sra

和 MIPS 一样, srl 和 sra 的主要区别是符号扩展. 由于 sra 的含义是算术右移, 故该运算的操作数是用补码表示的有符号数，因此该数应做符号扩展. 即如果符号位为 1 (即最高位，第 15 位),则在右移之后，还应将空出来的高位部分填充 1 ，否则填充 0 . srl 则视其操作数为一组独立的逻辑值，进行 0 扩展。

jump

和 MIPS.地址一样, jump 指令的参数是伪绝对地址。这里的地址是下一个要执行指令的低 12 位。高四位从当前 PC 处获得. 这里，和 MIPS 不一样的是，不需要在最后加零。其原因是，这里的 CPU 是半字寻址的, 故每个可能的地址都存储一个有效的 16 位指令.

```
PC = (PC & 0xF000) | address
```

beq/bne

beq 指令参数相对于下一条指令的**带符号**的相对偏移量，这和 MIPS 一样. beq 可如下表示:

```
if $r0 == $r1
        PC = PC + 1 + offset
else
        PC = PC + 1
```

bne 指令和上面指令的区别是测试指令中，用!=来代替原来的==

immediate 字段

一些 immediate 字段为无符号数，故应进行 zero-扩展, 而另一些则看成有符号数，需要进行符号扩展。每个指令具体进行何种方式的扩展，请参见本页上面的图。

immediate-s 代表 **符号（SIGNED）** 立即数.
immediate-u 代表 **无符号（UNSIGNED）** 立即数.

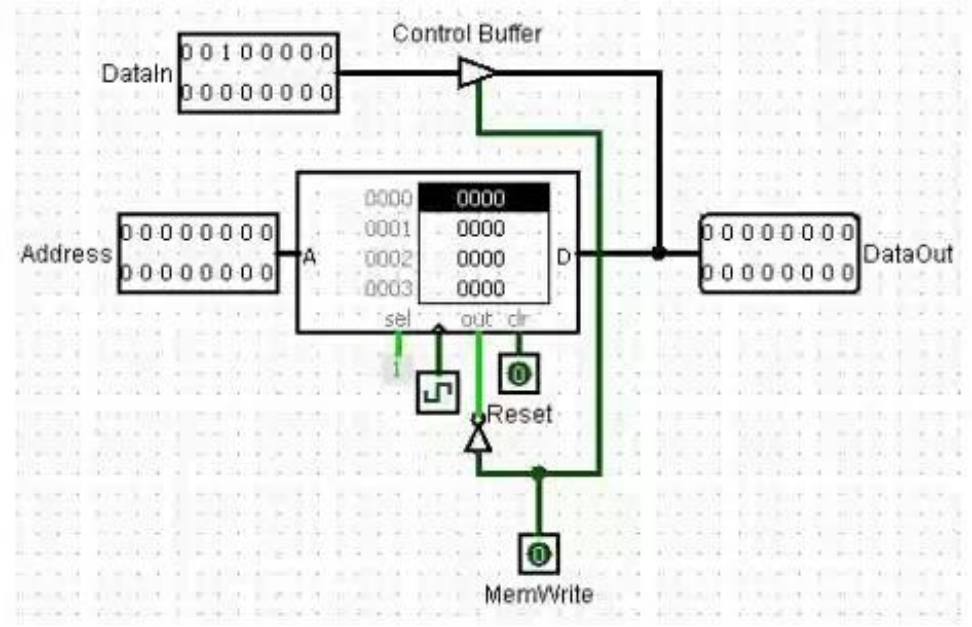
Logisim

强烈建议大家下载 Logisim，并在本机上运行，并经常保存你的.circ 文件，我们将使用的 Logisim 官方版本是 v2.1.6.

使用 logisim 的过程中如果出现问题，可以重启（*REBOOT）！* 不必浪费时间跟踪 bug，这不是你的错。但如果重启后还不能解决问题，则极有可能该 bug 是由你的代码引起的！出现此情况，请发信给我。

RAM 模块（Modules）

Logisim RAM 模块可在内建的内存库中找到. 为了将该库加到你的工程中, 选择"Project/Load Library/Built-in Library..." ，然后选中 Memory 模块.

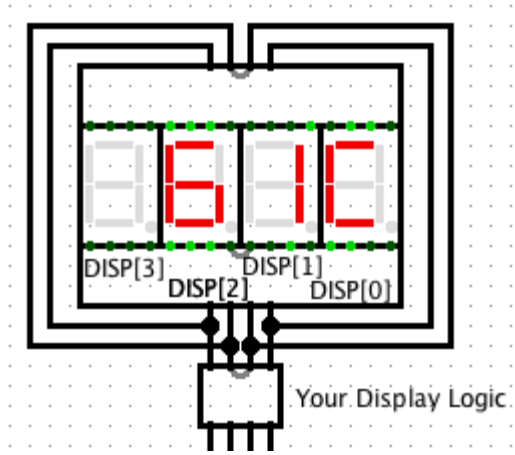


最佳的学习方式是进行实践. 下面将告诉你如何开始. Logisim 关于 RAM 模块的帮助页可在这里找到, 但可能帮助不是很大。"A" 选择要访问的地址(Address). "sel" 主要是决定 RAM 模块是否活动（当"sel"为低电平时, "D"未定义,即无值).时钟输入提供内存写同步信号. "out" 决定是读内存还是写内存. 如果"out" 为高电平, 则"D" 将输出由地址"A"决定的内存地址的内容. "clr"为高时，将立即设置内存的所有内容为 0. 在此模块中, "D"既作输入也作输出。这就要求你必须在"D"的输入端使用控制缓冲区(Control buffer)来防止输出及内存内容的可能冲突. 可以使用"poke" 工具来修改模块的内容。

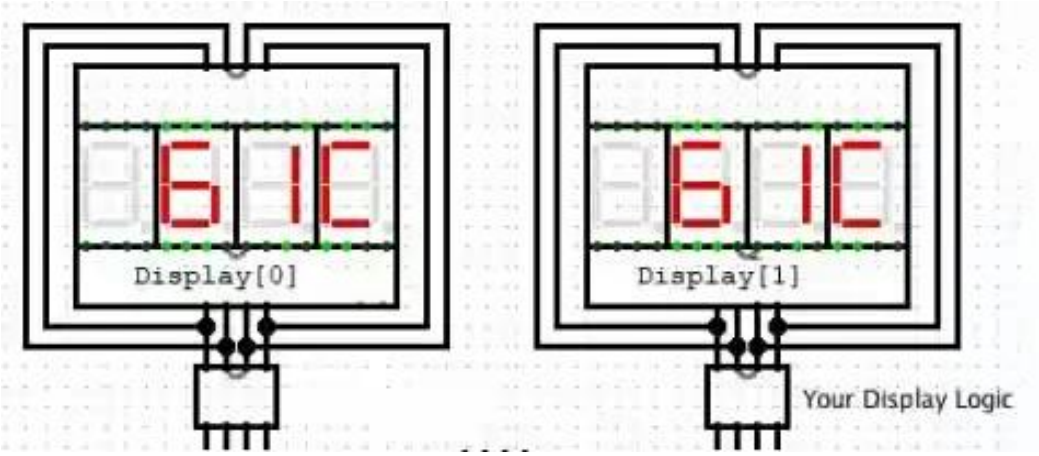
可以使用 Logisim 的 ROM 模块来作为指令内存. 它对 RAM 模块作了很大的简化。RAM 和 ROM 模块都可以的通过使用"right-click/Load Image..."来从文件中装入。

显示指令和发光二极管 The Display Instruction AND Overflow LEDs

如你所知，每个计算机都由五个部分组成：控制, 数据通道, 内存, 输入和输出. 相应的, 在本项目中，需要包含四个七段发光二级管作为显示输出. 其形状如下图所示:



或者



注意到上面的图像中有一些空值. 这看来是不太可能的，除非你用很可怕的方式实现，因此不用理会这些。

选项 1:

disp 指令设置寄存器的值到第 imm 个七段数码管显示. 该值将一直显示直到下一次 disp 指令替换显示的下标。如果立即数超出了你所设计的显示范围，可以忽略不显示，也可以进行循环显示。我们提供了一个用于转换的库函数，以方便七段数码管的显示处理。该函数可以在 proj3 文件夹里面有，你可以使用菜单选项"Load Library/Logisim Library"将其包含进来. 参考示例（examples）段来获得如何更好地把这些发送到你的项目中的办法. 每个单个 1 6 进制值 0-f 将会正确显示. 0xff 显示为空（任何一段都不亮，因为寄存器为 1 6 位宽，删除高 8 位）.

更新：本部分有些模糊，特做如下更新：如果按上面指定的方式产生显示（Displays），会忽略要显示的寄存器的高 8 位！这样，我们帮你做的逻辑电路（只取 8 位）将能和寄存器一起工作. **当然，你也可以不这样来实现显示（DISPLAY）!!**

选项 2:

另一种做法是，实现多个 4 位七段数码管来一次显示 1 6 位寄存器的内容. 为了实现这一点，需要至少两组七段数码管，每一组都与一个索引关联. 老师更喜欢这种做法，希望你能理解.

所有其他输入将显示一个'?'. 需要至少 4 个七段数码管来显示,同时还需要至少 2 个有效的立即数用于 disp 指令. 当然,更多的显示会使你所设计的CPU 更强大.因此如果你能更早的完成, 这是可以扩展的地方.

还需要**两个 LED** 单元来指示两类溢出. 它们仅当在主电路中可见时 (viewable)显示.

测试

在你完成了 CPU 设计后，可以编写程序在 CPU 上运行以测试 CPU 能否工作! 下面我们具体编写两个程序来测试处理器的功能。首先是一个非常简单的程序用以进行最最基本的测试。proj3/halt.hex 程序装入相同的立即数到两个寄存器，如果相等则分支到上一句（－ 1 ）。让你的 CPU 能运行 lui 和 ori 指令是*非常* 重要的, 因为他们使得自动测试成为可能。建议大家编写多个小程序来分别单独测试其他指令。

编写一个程序,该程序将内存中最开始的两个半字(MEM[0] and MEM[1])相乘,将结果存到 MEM[2]中. 程序的最后一条指令必须是 halt (该指令将无限制地跳转或者分支到自身). Feel free to clobber the original arguments. Save the ROM image in a file "mult.hex."

Write a program that displays the lower nine bits of the first half-word of memory in octal (base 8). For example, if the first half-word were 0x829f, the seven segment displays would read 237. Again, you may clobber any memory values you like and your program must end with a halt. Save the ROM image in a file "octal.hex".

汇编

我们提供了一个简单的汇编器，使你能用汇编语言来简化编程。在使用之前，你可以先手写几个机器代码编写的程序，这是一个好的练习，并且让你感到有点相当酷!

汇编器可以在 proj3/asm.pl 下载。 *这是 Nathan Kallus' v2.0 汇编器. 如果你想看看最异类的汇编器, 可以看看到处留情的* ***Gilbert Chou 构建的*** *v3.0 汇编器，可以在这里* proj3/asm2.pl. *该汇编器可以使用标号（labels），注解（comments）和 1 6 进制无符号立即数. 当然其 bug 可能超多!*

该汇编器的输入文件形式如下:

```
lui $r0, 85
ori $r0, $r0, 68
lui $r1, 85
ori $r1, $r1, 68
beq $r0, $r1, -1
```

在需要寄存器的地方，寄存器必须是$r0, $r1, $r2, $r3 之一. 不支持标号，立即数必须是 1 0 进制的,并且不检查数是否越界. 任意空行或者错误的指令将对应于执行一个 nop 指令。逗号是是可选的，但$是必选的. 在 UNIX 下,可使用如下命令启动汇编器(在 WINDOWS 下你需要下载一个 PERL 解释器，如 ActivePerl，或者 STRAWBERRY-PERL 安装，然后在命令提示符下运行）：

```
% perl asm.pl <input.s > output.hex
```

提示与注解

本部分包含提示与注解来帮助你开始，同时激发一些学术争论. 现在你可以跳过本节，当你遇到麻烦时再回来。

显示逻辑

You might want to make some sort of specialized register file for keeping track of current display values. The desired functionality is about halfway between a register file and a RAM module.

Jumps in the Assembler

If you are calculating your jump addresses off of line numbers, keep in mind that the first instruction in your program is at address 0, but the first line is often labelled as line 1. So remember to subtract 1 off of all your addresses or insert a nop at the beginning just before you run the assembler.

思考的问题

1) 哪个指令(及何种参数)的作用是 nops?
2) 有些什么样的方式来中止一个程序?
3) 如何实现乘法? 除法?

Logisim's Combination Analysis Feature

Logisim offers some functionality for automating circuit implementation given a truth table, or vice versa. Though not disallowed (enforcing such a requirement is impractical), use of this feature is discouraged. Remember that you will not be allowed a laptop running Logisim on the final.

Key Differences From MIPS

1) The zero register isn't special. $r0 is just a regular register like $r1.
2) Memory is addressed every 16 bits, allowing which we will call a HALF-WORD. We could have simply redefined WORD to mean 16 bit however we felt this would have been too confusing, so we decided to call it HALF-WORD addressing. Unlike MIPS, every memory address holds a valid 16 bit HALF-WORD that could be an instruction or an entire integer.
3) Data memory and instruction memory are distinct. This is what we tell you in MIPS as well, but when we talk about cacheing we'll find out they actually live in the same address space.
4) The `disp` instruction borders on being a CISC style instruction.

其它需求

在实现 CPU 时，你可以使用任何 logisim 内建的电路组件。

You must have your instruction ROM module and data RAM module visible from the main circuit of your Logisim project. You must include an array at least four seven segment displays in the main circuit as well. Feel free to add additional seven segment displays if it does not clutter your CPU.

Use the label tool to organize your CPU. In particular label the control, datapath, and display sections, but it can also be useful to label specific busses and wires. It could make debugging a lot easier!

给专家的一些额外工作

一旦建立好 CPU 并开始运行，进行下面的尝试。

1) Write an assembler that can handle labels, long branches and jumps, and pseudo instructions. Feel free to define your own register and/or memory conventions to make this work (declare MEM[15] as assembler reserved, etc.)
2) Try writing a program that divides MEM[0] by MEM[1] and stores the quotient in MEM[2] and the remainder in MEM[3].
3) Use your program from part 2 to write a program that displays a location in memory as a decimal value! (be careful about jumps, being a linker can be tough).
4) Write something crazy that we haven't thought of.

提交

你需要提交以下文件:
cpu.circ
mult.hex
octal.hex