P4(Verilog单周期CPU)

▼ Contents

▼ Problems

- [x] ~~怎么可以命令行驱动Vivado呢，或者用脚本操作~~

— Solution：在Vivado附带的doc中有一章Logic Simulation中有对命令行驱动的方法（要先进入-mode tcl），或者用batchmode驱动tcl文件，tcl文件的命令就把vivado里面tcl console中的命令拷贝进来就好了

—Solution 2：用iverilog自动化，也更简单一点，同时跑的也更快一点。

设计要求

- **处理器应支持指令集为：{addu, subu, ori, lw, sw, beq, lui, jal, jr,nop}。**
- addu, subu 可以不支持溢出。
- 处理器为**单周期**设计。
- **不**需要考虑**延迟槽**。
- 支持**同步复位**。
- **需自行构造测试集**，验证设计的正确性。（通过课下自动测试并 **不意味着** 你的设计 **完全不存在问题**）
- **之后每节说明性内容后都附有文档撰写建议与相关思考题，请务必完成相关思考题并附在 CPU 设计文档后**

- 最后需要提交的内容： CPU 设计文档（含思考题）、 Verilog 文件打包成.zip 为后缀的格式（全部.v 文件放在同一个文件夹下压缩后提交，**注意压缩包大小限制**）。
- 顶层文件为 **mips.v**，接口定义如下：

文件	模块接口定义

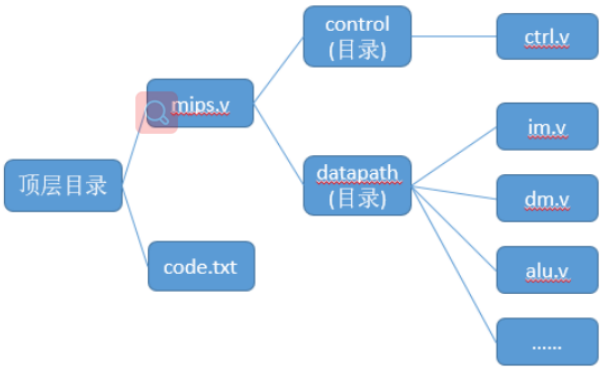
mips.v	`module mips(clk,reset);`
	`input clk; //clock`
	`input reset; //reset`

基本思路

写verilog时，为了满足“高内聚，低耦合”的要求，会进行跨文件，多模块的Verilog工程化项目

其实本质上就是对Logisim搭的P3翻译一下，但是，**翻译的技巧**也会决定了代码的复杂度

一般推荐的层次化图



整体设计已经完成，接下来就是具体的模块化与层次化设计。

- 单周期处理器包括控制器和数据通路，将其放入 mips.v 的层次下。code.txt 中存储相应的指令码。
- control 模块占一个独立的 Verilog HDL 文件，实现控制器这个单一的职责，降低模块间的耦合度。
- datapath 中的每个 module 都由一个独立的 Verilog HDL 文件组成。在增加相应功能或者修改代码时，减小了对其他模块的影响和修改，条例更加清晰，且对后续的设计也是非常有用和方便的。
- 一个 Verilog HDL 文件中可以定义多个 module，因此建议所有 mux（包括不同位数、不同端口数的所有 MUX）都建模在一个 mux.v 中。

这样设计的好处是整个工程具有极好的**局部性**（包括所谓的**模块化**等等）。记住模块化的要领不是从模块的大小出发，而是从**模块功能的独立性**出发。

IM设计

注意，Verilog是没有ROM这种东西的，即已经搭建完的存储器模块，需要自行设计

Key：要生产匹配需求数量的32位寄存器的阵列

而logisim中导入code.txt中机器码的过程，ISE中需要用`$readmemh`

`$readmemh`

功能： 读取文件到存储器

格式：

- `$readmemh("<数据文件名>",<存储器名>);`
- `$readmemh("<数据文件名>", <存储器名>, <起始地址>);`
- `$readmemh("<数据文件名>", <存储器名>, <起始地址>, <结束地址>);`

功能： `$readmemh` 函数会根据绝对/相对路径找到需要访问的文件，按照 ASCII 的解码方式将文件字节流解码并读入容器。文件中的内容必须是十六进制数字 0~f 或是不定值 x，高阻值 z（字母大小写均可），不需要前导 0x，不同的数用空格或换行隔开。假设存储器名为 arr，起始地址为 s，结束地址为 d，那么文件中用空格隔开的数字会依次读入到 arr[s],arr[s+1]... 到 arr[d]。**假如数字的位数大于数组元素的位数，那么只有低位会被读入，剩下的高位会被忽略。**

此系统任务用来从文件中读取数据到存储器中，类似于 C 语言中的 fread 函数。

例如：

```
module im;
reg[31:0] im_reg[0:2047];
initialbegin$readmemh("code.txt",im_reg);
endendmodule
```

仿真后即可将 code.txt 中的内容读入 `im_reg` 存储器中。

控制器设计

设计流程

首先分析各个部件从控制器中所需的控制信号，并全部列举出来，比如 ALU 需要控制器输出一个 ALUOp 来控制其执行何种运算。在 P3 时，已经完成了控制器电路的布局布线，所以在此只需要加上新增指令所产生的控制信号即可。

可以在设计文档中列一张表，来记录控制器的映射逻辑。

这项工作对于后续的仿真调试和课上测试都非常有帮助。因为仿真调试时可以首先查看表格是否正确，再去确认是否和代码对应，Bug 一目了然。对于添加指令，只需在表格后面加入新的操作码和对应的控制信号的值，再将其映射到 Verilog HDL 代码中，非常方便。至于表格的具体设计，则因人而异，可以记录下 **指令对应的控制信号如何取值**，也可以记录下**控制信号每种取值所对应的指令**，

实现过程

对着Logisim翻译就完了，注意对MUX的建模（Verilog每这种东西）

让代码看起来没那么复杂的关键在于datapath，也就是数据通路的设计

其实就是将Logisim里main函数连导线的工作，通过实例化将各个模块串联起来。

其中，可以注意各个信号，input，output，wire的选取

但实测好像影响不大（？）

对于我个人编写来说

当一个信号是Controller的output，并只用作其他模块的input时，那么令这个信号为input

当一个信号只是Controller的input，并是其他模块的output时，那么令这个信号为output

当一个信号是某个模块的output，同时是别的模块的input时，用wire链接

调试建议

仿真出现高阻值z代表并未连接到有效输入上，首先应考虑连线问题，

仿真出现不定值x的情况时首先考虑初始化问题

自动化测试

Mars

结合修改版Mars，help中提供的cmd代码，大体如下

`java -jar Mars.jar nc mc CompactDataAtZero [ProgramName.asm] >std.txt`

可以Run I/O信息输入到std.txt文件

同时`java -jar Mars.jar nc mc CompactDataAtZero dump HexText .text code.txt [ProgramName.asm]`

可以把.text的16进制机器码输入到code.txt（要事先创建）

Vivado

不会用命令行控制vivado仿真😭，说不定要写脚本? 以后再试试，但在课堂上又是用ise（x），要不试试vcs算了，或者看看课堂上有iverilog没有

~~因此只能每次手动restart了，注意下文件的读取吧~~

问题解决，可以用vivado和iverilog两种方法

iverilog具体见教程

vivado看:

— Solution：在Vivado附带的doc中有一章Logic Simulation中有对命令行驱动的方法（要先进入-mode tcl），或者用batchmode驱动tcl文件，tcl文件的命令就把vivado里面tcl console中的命令拷贝进来就好了

```
while(1)
{
  //system("D:\\Archive\\cpp\\BUAA_CO\\cmake-build-debug\\BUAA_CO.exe");
```

```
  //system("fc std.txt src.txt");
  system("D:\\Archive\\ISE\\Monocycle_CPU\\Monocycle_CPU.sim\\sim_1\\behav\\xsim\\BUAA_CO.exe");
  system("java -jar Mars.jar nc mc CompactDataAtZero dump .text HexText code.txt Test.asm 1> std.txt 2>err.txt");
  printf("Data have been already generated\n");
  system("Vivado -mode batch -source start.tcl");
  //system("open_project D:/Archive/ISE/Monocycle_CPU/Monocycle_CPU.xpr");
  system("cls");
  system("fc /W std.txt src.txt");
  //  system("pause");
  if(system("fc /W std.txt src.txt")) system("pause");
  system("cls");
}
```

文件输出

Key: `$fopen` 和 `$dfisplay`

同时要注意的是，我们每次仿真时要保证文件为内容清空，但DM和GRF执行输入时要在原先的基础上输入

我对此的处理方式是在 `$fdisplay` 模块中使用“a+”方式打开，在tb里选择“w”方式打开

自动生成与对拍

vivado

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
  while(1)
  {
    system("D:\\Archive\\cpp\\BUAA_CO\\cmake-build-debug\\BUAA_CO.exe"); //这个是自动生成MIPS代码的程序，看情况修改
    //system("fc std.txt src.txt");

    //关于MIPS的command相关在help中也有
    //这边就是上面提到的结合体，（nc是为忽视copyright信息，可以删掉看一下变化）
    system("java -jar Mars.jar nc mc CompactDataAtZero dump .text HexText code.txt Test.asm 1> std.txt 2>err.txt");

    //提示数据生成成功
    printf("Data have been already generated\n");

    //调用先前写的tcl文件，依次来驱动仿真
    system("Vivado -mode batch -source start.tcl");

    //输出的东西有点多，不清屏看的有点难受
    system("cls");
    system("fc /W std.txt src.txt");
    //遇到不一样的就暂停
    if(system("fc /W std.txt src.txt")) system("pause"); ;

    //清屏用的
    system("cls");
  }
```

tcl文件里面长这样（把tcl console相关信息复制过来就好了）

```
open_project D:/Archive/ISE/Monocycle_CPU/Monocycle_CPU.xpr
update_compile_order -fileset sources_1
launch_simulation
```

iverilog

用iverilog自动化，也更简单一点，同时跑的也更快一点, 问题在于iverilog的仿真结果与ise和vivado有时候会存在差异，一般是处在阻塞赋值和非阻塞赋值上好像。

```
#include<bits/stdc++.h>
using namespace std;
int main()
{

  while(1)
  {
    system("D:\\Archive\\cpp\\BUAA_CO\\cmake-build-debug\\BUAA_CO.exe");
    system("java -jar Mars.jar nc mc CompactDataAtZero dump .text HexText code.txt Test.asm 1> std.txt 2>err.txt");
    printf("Data have been already generated\n");
    system("iverilog -o a.vvp DM.v IM.v Controller.v datapath.v ALU.v mips.v tb.v GRF.v dasm.v MUX.v PC.v EXT.v NPC.v");
    system("vvp a.vvp");
    system("fc /W std.txt src.txt");
//    system("pause");
    if(system("fc /W std.txt src.txt")) system("pause");
    system("cls");
  }
}
```

Warning

> ⚠️ 不要忘记P1的坑

1. 符号数的 `$signed`
2. for循环统计变量在always块中的清零
3. 看清新加的指令与RegWrite，MemWrite，SignExt, ALUSrc等的关系，要记得修改
4. 要记得实例化的连线

课上

jr写错了，de了至少一个半小时 😣,然后发现是关于jr的RTL理解错误，以后复查的时候一定要对着指令集再看一遍，太离谱了😑

注意要点

1. for循环中，循环次数要是稳定的，即上下限要是常数，不能是常数（仿真倒是可以仿真，就是评测的时候会TLE，说是如果不稳定不满足可综合的要求）
2. 对于带条件的指令（尤其是于branch and link相关），最好在controller里就把相关的条件用组合逻辑进一步判断（如对于beq，可以在controller 传入一个zero）,然后

   ```
   assign beq_w = (op == 6'b000100);
   assign beq = beq & Zero;
   ```

会更方便指令的执行

课上题

bnezalc

if GPR[rs] ≠ 0

PC = PC + 4 + sign_extend(offset||0^2)

else

PC = PC + 4

slo

$GPR[rt] = GPR[rs]_{31..imm} || \ GPR[rs]_{imm-1:0}$

访存

忘了，调完jr指令就来不及了，题都没怎么看，就记得有个循环右移

附带一些文件

▼ 用来生成MIPS程序的

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/f3831f7b-feca-442a-ab58-28954f284e85/MIPS_generate.cpp

▼ 用来对拍的

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/80c2ff95-2a74-4263-8a51-9f0cff422410/test.cpp

▼ 本人写的CPU

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/3e26fd88-5740-48d1-bff3-b97ba52836dc/CPU.zip