Y86-64 Simulator 说明文档

队伍人员

- 高庆麾 19307130062
- 孙若诗 19307130296

目录及文件说明

- `my_pipe/`
    - 初始为空，用于存放 `Y86-64 Simulator` 对 `y86-code/` 下 .yo 文件进行模拟的结果 .pipe 文件
- `my_pipe_honor/`
    - 初始为空，用于存放 `Y86-64 Simulator` 对 `y86-code-honor/` 下 .yo 文件进行模拟的结果 .pipe 文件
- `psim/`
    - 存放官方给出的标准模拟器，用于生成每个 .yo 文件的标准模拟结果
    - 目录下有四个文件，分别是 `psim`、`psim.backup`、`psim.bad.backup`、`psim.bad2.backup`
    - 其中 `psim` 是当前使用的标准模拟器，实际是 `psim.bad2.backup` 的拷贝。`psim.backup` 是原始的标准模拟器，`psim.bad.backup` 是经过修改后运行出现错误的标准模拟器，`psim.bad2.backup` 是经过修改后正确运行的模拟器，也是当前正在使用的标准模拟器
    - 当前的 `psim` 标准模拟器的修改为：省去了一些事件输出，如 `Execute: ALU: + 0x0 0x0 --> 0x0`、`Wrote 0x13 to address 0x1f8`、`Writeback: Wrote 0x200 to register %rsp` 等，以便比对模拟结果
- `std_pipe/`
    - 初始为空，用于存放 标准模拟器 对 `y86-code/` 下 .yo 文件进行模拟的结果 .pipe 文件
- `std_pipe_honor/`
    - 初始为空，用于存放 标准模拟器 对 `y86-code-honor/` 下 .yo 文件进行模拟的结果 .pipe 文件
- `y86-code/`
    - 存放用于测试的样例程序文件，其后缀为 .yo
- `y86-code-honor/`
    - 存放用于测试的样例程序文件，其后缀为 .yo
- `main`
    - 自行编写的模拟器 `Y86-64 Simulator` 的可执行文件
- `main.cpp`
    - 自行编写的模拟器 `Y86-64 Simulator` 的源代码
- `Makefile`
    - 自行编写的 Makefile 文件，支持如下命令：

        - make run ，用 `Y86-64 Simulator` 测试所有 `y86-code/` 下的样例程序 .yo ，并将模拟结果 .pipe 存放在 `my_pipe/` 下
        - make run s=1 ，用 标准模拟器 测试所有 `y86-code/` 下的样例程序 .yo ，并将模拟结果 .pipe 存放在 `std_pipe/` 下
        - make run h=1 ，用 `Y86-64 Simulator` 测试所有 `y86-code-honor/` 下的样例程序 .yo ，并将模拟结果 .pipe 存放在 `my_pipe_honor/` 下
        - make run s=1 h=1 ，用 标准模拟器 测试所有 `y86-code-honor/` 下的样例程序 .yo ，并将模拟结果 .pipe 存放在 `std_pipe_honor/` 下
        - make clean ，清除 **Makefile 所在目录下** 的全部 .yo 和 .pipe 文件。这常用于模拟器运行失败时，相关文件会被保留，需要手动进行清理的情况
- `README.md`
    - `markdown` 格式的说明文档
- `README.pdf`
    - `pdf` 格式的说明文档，由 `README.md` 导出而来

除上述文件与目录外，提交目录中不含有任何其他文件或目录

运行方法

- 使用上文提到的 make 命令进行测试
- 直接 ./main < [input_file] > [output_file]

设计详情

总体框架和设计原则

- **中心原则**：以硬件设计风格为基调，对硬件单元（寄存器、线路）及其相互作用进行模拟
    - 由此，整个模拟器中，有如下自定义类型：
        - `mem_t` 内存单元类型
        - `reg_t` 寄存器类型
        - `cpu_t` CPU控制层类型
        - `ptr_t` 指针类型
        - `wire_t` 线路类型
        - `stat_t` 状态指示器类型
        - `cc_t` 条件码类型
        - `imm_t` 立即数类型
        - ...
    - 后期设计中，为避免严格类型区分造成的编程复杂度开销和风险，主要仅使用两种类型：
        - `reg_t` 寄存器类型
        - `wire_t` 线路类型
    - 硬件风格编程的优点：
        - 本质上是对电路图的模拟，只要画出详细而正确的电路图，很容易得到正确运行的模拟器
        - 由于本质是对硬件的模拟，能相当容易地解决 所有硬件寄存器**同时**在时钟上升沿更新的问题（这是比较麻烦且很容易出错的模拟部分，但是通过对

硬件进行模拟可以很容易地进行处理，且几乎没有任何编程错误风险）
        - 很容易处理硬件电路**并行计算**的特点
- 设计框架
    - 基本硬件组件及工作流程
        - 组合逻辑块、系统硬件逻辑块（包含内存、寄存器文件、条件码等的更新）、阶段寄存器等
        - 标准工作流程：初始化、运算、更新（加载）、运算、更新（加载）、...
        - 建议先阅读标准工作流程中各个阶段的介绍，以便更快理解整体设计思想
    - 组合逻辑块
        - 所有的组合逻辑块，在模拟中等价于无返回值函数，其中输入变量采用**值传递**，输出变量采用**引用传递**，输入变量可以是 `reg_t` 或 `wire_t` 类型，输出变量 **一般为** `wire_t` 类型
    - 系统硬件逻辑块
        - 也可看做组合逻辑块，不同点在于需要真正更新系统硬件（系统硬件作为输出变量），如内存、寄存器文件、条件码等
    - 阶段寄存器（阶段划分）
        - 除标准的五大阶段 F（取值）, D（译码）, E（执行）, M（访存）, W（写回）外，额外添加 L（逻辑控制）阶段，该阶段控制各个阶段是否暂停或添加气泡
        - 并不实际存在，由很多阶段内硬件寄存器构成，两个阶段之间存在很多线路，这些线路视作属于下层阶段（如 D 阶段和 E 阶段之间的线路认为属于 D 阶段）
        - 控制该阶段的初始化（init）、运行（run）和加载（flush）
    - 初始化阶段
        - 为各个阶段寄存器添加气泡，以便初始化
        - 添加气泡后，需要对每个阶段寄存器进行一次加载，以将所有阶段寄存器包含的硬件寄存器以及相关系统硬件全部初始化。这与标准工作流程中先运算再加载有所不同
        - 此处的加载顺序需要特别注意，要将逻辑控制阶段寄存器（见上文 “阶段寄存器” 部分）放在最后更新，以避免将气泡洗掉
    - 运算阶段
        - 最重要部分（也是最主要部分）是通过当前阶段的所有 `reg_t` 类型变量（代表硬件寄存器），计算出所有 `wire_t` 类型变量（代表线路），这也是对硬件电路进行模拟的核心（硬件寄存器更新后，其输出线路值随之发生改变）
        - 需要严格注意各个组件的依赖关系，如一个组件 A 依赖组件 B，则 A 必须在 B 之后执行，否则 A 就相当于使用了 B 错误的**旧**输出值进行了运算
        - 需要注意各个阶段寄存器的运算顺序。正确的顺序为按指令执行的**逆序**，也即 W, M, E, D, F 的顺序进行运算，最后对 L 阶段运算。这是由于各阶段之间存在明显的依赖关系
    - 更新（加载）阶段
        - 也称 flush 阶段，刷新所有的硬件寄存器，也就是将所有新得到的 `wire_t` 写入对应的 `reg_t` 中。这是对时钟上升沿寄存器更新的模拟
        - 需要注意各个阶段寄存器的更新顺序，正确的顺序为先加载 L，以明确各个其他阶段是否需要暂停或添加气泡并及时进行干扰。其他阶段顺序理论上可以任意，但实际采用指令执行的顺序，也即 F, D, E, M, W 进行加载
        - 加载时更新所有下一阶段寄存器所包含的硬件寄存器，输入变量为这些寄存器对应的线路，输出变量为这些寄存器。对应线路和寄存器名称完全相

同，线路所属阶段是寄存器对应阶段的上一阶段，直接将对应线路值传递给寄存器即可
        - 每个阶段寄存器首先判断是否暂停，如需要，直接将气泡指示器和暂停指示器清零，然后退出。其次判断是否添加气泡，若需要，则将包含的各个硬件寄存器复位，再将气泡指示器和暂停指示器清零，然后退出。否则，正常将所有 `wire_t` 传递给对应 `reg_t` 即可
        - 有些硬件寄存器会直接传给下一个阶段的对应硬件寄存器，此时我们定义一个 Pass 函数，仍然将这些寄存器值先放入线路中，再在加载时通过线路传给下一阶段寄存器，保证处理的同一性，减少特例以降低编程风险
    - 命名规范
        - 硬件寄存器：以对应阶段的大写字母开头，加下划线，加该寄存器名称
        - 线路：以所属阶段的小写字母开头，加下划线，加该线路名称
        - 组合逻辑块：一般以该逻辑块名称命名，若各个阶段之间发生重名，则在名称前加对应阶段大写字母加下划线
        - 系统硬件逻辑块：以系统硬件逻辑块本身名称命名
        - 系统硬件，以系统硬件本身名称命名

编程细节处理

- 编程风格
    - 几乎所有数值均采用十六进制表示，方便调试且更贴近硬件模拟
    - 采用许多名称空间，将所属变量和功能归类，便于测试和查错
    - 所有硬件都定义为全局变量，便于逻辑控制时引用
    - 由于上一条的存在，所有运算组件（组合逻辑块、系统硬件逻辑块）均定义在上述硬件全局变量**之上**，同时在参数表内列出所有组件相关变量，防止编程错误引用到该组件本不该引用的变量上，避免了编程风险
- （宏）定义部分
    - 最大内存范围、最大寄存器数量、PC 起始位
    - 标准程序状态码以及补充的程序状态：气泡对应的标号
    - 所有指令的字节块、代码部分以及功能部分对应的标号
    - 所有寄存器对应的标号
    - 所有硬件模拟所需的类型定义
- 多层次架构
    - 定义层
        - 宏定义、类型定义
        - 工具箱
            - 编写了从标号转化到对应指令、寄存器、状态码并能判断是否合法的函数（不合法时输出 "ERROR" 名称）
        - 硬件组件层
            - 定义了所有组合逻辑块、系统硬件逻辑块、阶段寄存器的运算及加载函数
        - 硬件定义层
            - 定义了所有硬件寄存器、系统硬件以及线路
            - 定义在全局
    - 控制层
        - 阶段控制层
            - 定义了每个阶段的初始化、运算和更新方法
            - 定义了每个阶段的相关硬件信息输出方法（高度模拟 标准模拟器 的输出格式）

        - CPU 整体控制层
            - 定义了 CPU 初始化、运算和更新方法（也就是把所有阶段合在一起，以便调整并确定每个阶段的初始化、运算和更新顺序等）
    - 用户层
        - 模拟器初始化层
            - 编写了从 .yo 文件中读取内存字节的函数
        - 单元测试层
            - 对特定函数功能进行正确性测试
        - Y86 用户层
            - 控制整个 Y86 模拟器的初始化、运行和终止行为
    - 主函数调用
        - 可以选择进行测试还是进行模拟

已修复 bug 清单

- 工具箱未考虑不合法情况，修复后在不合法情况下输出 "ERROR"
- 指令不合法的判断逻辑有误
- 使用 `reg_t` 定义指针并在无符号整数环境下进行判断会出现问题，改为特别定义的有符号整数 `ptr_t` 指针类型
- PC 增加器中至少需增加指令字节对应的一个字节
- 对阶段寄存器中暂停和气泡实现机制的理解有误，正确的做法是在正常加载前进行判断
- F 阶段寄存器比较特殊，需要加载下一阶段的硬件寄存器 **以及** 自身包含的 predPC
- 逻辑块中某些情况下未涉及的输出变量必须复位
- 条件码更新逻辑中注意每个条件码仅有一位大小，取与如果大于 1 必须移到数值 1
- 由于在无符号数环境中操作，需要手动用符号位判断正负而非直接判断正负（因为数值上永远非负）
- ALU 中运算是 B 操作数在前，A 操作数在后
- 注意数值溢出的判断方法
- 跳转条件运算中，注意不同条件对应的条件码运算式
- 每个逻辑块（函数）中要注意是否有变量没有用到，避免遗漏
- 注意不要将读写内存位搞反
- 注意每个字节长度为 8 位
- 注意在读写内存时要判断内存位置是否合法
- 注意每个逻辑块的依赖关系以及相应的执行顺序
- 注意初始化时的特殊加载逻辑（ L 最后执行）
- 注意从 .yo 文件中 fetch 内存内容时，不要完全以 'x' 作为标志（因为右侧说明部分中也可能含有）

这就是说明文档的全部内容，感谢观赏！

2020.12.2