1. Hardware

// 注释部分为我的参考意见。

a. Register

基础类，用于构造Computer各个部件，用于实现各部件的基本显示，存取，转换等。

某些方法需按寄存器的bit位数来进行重载。

//特殊的Register，包括PC包括MFR等在实际操作时可以考虑将他们考虑成其他类。

//实际上虽然几乎所有Register都是一个Word的信息，但由于不同种类的Register之间的共性并不大，

//虽然对于GPR而言， 会需要fetch一个下标来决定究竟是R0-R3中的哪个（X同理），但并不存在一条指令（设为指令Inst），

//让我们需要 Inst 参数1.参数2... 参数X...参数N... 而参数X决定目标寄存器到底是GPR还是MBR还是MAR。

//因此实际上考虑共性的话， Register可以作为一个Superclass, 所有不同种类的寄存器Extends Register， 细节部分各自Override

2. Computer

a. CPU

CPU各个部件，PC，IR等等。由Register类声明构造。

//意见同上，因此实际IR之类还是可以拥有自己的类

//实际上 内存访问方法的入口可以建立在这里。 有效地址的逻辑可以封装CPU内部。

b. Cache

Register类型的一维数组（暂）。

//Temp Comment-PS: 虽然Register类型内部确实基本都是一个Word，但对于Cache和Memory而言也许Word数组更为直观些。

//逻辑上来看的话就算考虑到每次访问cache一半都是需要向寄存器取值，但这层封装似乎没有额外的有利之处。反正Memory本来就是个Data阵列。

//真要考虑大家都是Word这一点的话那反倒该去定义一个Word类---更没必要了。 仅供参考。

c. Memory

Register类型的一维数组。

//同上

//基础内存方法建立于此（即不带任何cache考量的直接访问），仅配套基本的地址检查。

//注意Expandable这一点， 也就是说需要一个方法实现扩展。 实际上一开始初始化4096就可以了，限制一个top value访问。 别忘了而已。

//

3. Operation

a. Decode

Instruction的分解并送到CPU的相应部件。

//建议定义一个实现字符串-->Binary指令的Decode类，可以在Simulate调用Decode来实现SingleStep。 但Binary指令的理解可用Interface--这是多样性最丰富的部分，Interface可以带来一定优势//，有利于模拟回调。Instruction的分解部分实际应该算作指令执行的一个显式部分（如何Fetch自然取决于Opcode）。

// 再次吐槽java的method不能直接变量化。

b. Compute

用于模拟硬件上的计算。

//可以进一步细分，例如加法器自成一类，但Accum方法可以实现于Register内部。

//在这基础上， 可以定义 “ZSCO发生变化”本身为一个类，每次运算后传递这个信号给CC即可。

c. Transfer

用于模拟硬件上的数据转移。

//Alert:注意下各种情形下Transfer的可行性是否存在。

4. Execution

a. Command

LDR,STR等命令的分解与执行。

//Command的定义部分， 包括分解和如何执行建议放置到Interface，统一用interface Instruction(此处的统一便于下文的举例)。

//例如用一个 public interface Instruction指令名 =new Instruction{.......}的方式实现并且统一封装在CPU类内部，这样一旦被Decode，可直接从Decode到的Opcode映射到对应

//Interface，从而实现回调的效果。

//同时不同指令的内在逻辑将可以互不干扰的存在于不同code。 不需要在Decode内部产生大量的分支。

//注： 核心在于将不同指令的逻辑步骤回调化。

//Instructor

b. Load/Words

从txt读入到memory

5. Simulate

a. Simulate

整个的模拟器运行主程序的流程。

6. Interface

a. Interface

界面

//同Execution部分

7 基础型

意见: 实际上short和int都可以 (byte[]的话理论也可以实现扩展，不过繁琐、目前没需求而且没见到特别的有利之处，可以忽视)。

但注意：

short的优点：牵涉到overflow和有sign big的操作，无需我们特别处理内容。 而且本身恰好是当前定义的word。

short的缺点： 位运算之类的时候会默认转化为int之类，反正要小心coding时别因此出现bug。

bug的典型就是位移的位数如果仍然基于short长度考量，很容易出现表面上符合逻辑但实际结果相差甚远的情形。

int的优点： short的缺点。

int的缺点： 就是short的优点。

大家及时查看和修正啊。。。

//大家及时查看和修正啊。。。。。。。。