数字电路与数字系统实验

2020 秋学期

MD5 摘要

191220152 张城铨

191220029 傅小龙

目录

前言.....4
实现功能.....4
软件部分（傅小龙）.....4
硬件部分（张城铨）.....4
第一部分 MD5 模块.....5
1.1 概述.....5
1.2 发展历史.....5
MD2.....5
MD4.....5
MD5.....6
1.3 MD5 算法原理.....6
1.3.1 填充.....6
1.3.2 初始化变量.....7
1.3.3 处理分组数据.....7
1.3.4 输出.....10
1.4 MD5 应用.....10
1.4.1 一致性验证.....10
1.4.2 数字签名.....11
1.4.3 安全访问认证.....11
1.5 MD5 特点.....11
1.6 java 代码实现 MD5 摘要算法.....12
参考：.....15
1.7 Verilog 实现 MD5 摘要算法.....16
1.7.1 数字抽象.....16
1.7.2 建立模型.....17
第二部分——io 输入输出.....23
*模块划分.....23
2.1 键盘输入模块.....24
2.1.1 原理.....24
参考：.....24
2.1.2 具体代码.....25
2.1.3 遇到的一些问题（时序）.....26
2.1.4 data_2_asc 代码.....27
2.2 状态控制模块.....29
2.2.1 状态转移图.....29
2.2.2 具体实现.....29
2.2.3 与 MD5 模块的交互.....31
2.3 显示模块.....32
2.3.1 原理.....32
2.3.2 字符 rom & 显存 ram.....33
2.3.3 地址计算及处理.....33

2.3.4 遇到的问题......34
三、实验启示......35

前言

实现功能

软件部分（傅小龙）

- 实现对字符串（< 56 位）加密处理
- verilog 与 java 程序的对拍测试

- 工作量：>30h

硬件部分（张城铨）

- 有限状态机控制软件、硬件间的交互
- vga 显示与键盘间的交互

工作量：>25h

第一部分 MD5 模块

1.1 概述

MD5 即 Message-Digest Algorithm 5（信息-摘要算法 5），用于确保信息传输完整一致。是计算机广泛使用的杂凑算法之一（又译摘要算法、哈希算法），主流编程语言普遍已有 MD5 实现。将数据（如汉字）运算为另一固定长度值，是杂凑算法的基础原理，MD5 的前身有 MD2、MD3 和 MD4。

1.2 发展历史

MD2

Rivest 在 1989 年开发出 MD2 算法。在这个算法中，首先对信息进行数据补位，使信息的字节长度是 16 的倍数。然后，以一个 16 位的检验和追加到信息末尾，并且根据这个新产生的信息计算出散列值。后来，Rogier 和 Chauvaud 发现如果忽略了检验将和 MD2 产生冲突。MD2 算法加密后结果是唯一的（即不同信息加密后的结果不同）。

MD4

为了加强算法的安全性，Rivest 在 1990 年又开发出 MD4 算法。MD4 算法同样需要填补信息以确保信息的比特位长度减去 448 后能被 512 整除（信息比特位长度 mod 512 = 448）。然后，一个以 64 位二进制表示的信息的最初长度被添加进来。信息被处理成 512 位 damg?rd/merkle 迭代结构的区块，而且每个区块要通过三个不同步骤的处理。Den boer 和 Bosselaers 以及其他人很快的发现了攻击 MD4 版本中第一步和第三步的漏洞。Dobbertin 向大家演示了如何利用一部普通的个人电脑在几分钟内找到 MD4 完整版本中的冲突（这个冲突实际上是一种漏洞，它将导致对不同的内容进行加密却可能得到相同的加密后结果）。毫无疑问，MD4 就此被淘汰掉了。

尽管 MD4 算法在安全上有个这么大的漏洞，但它对在其后才被开发出来的好几种信息安全加密算法的出现却有着不可忽视的引导作用。

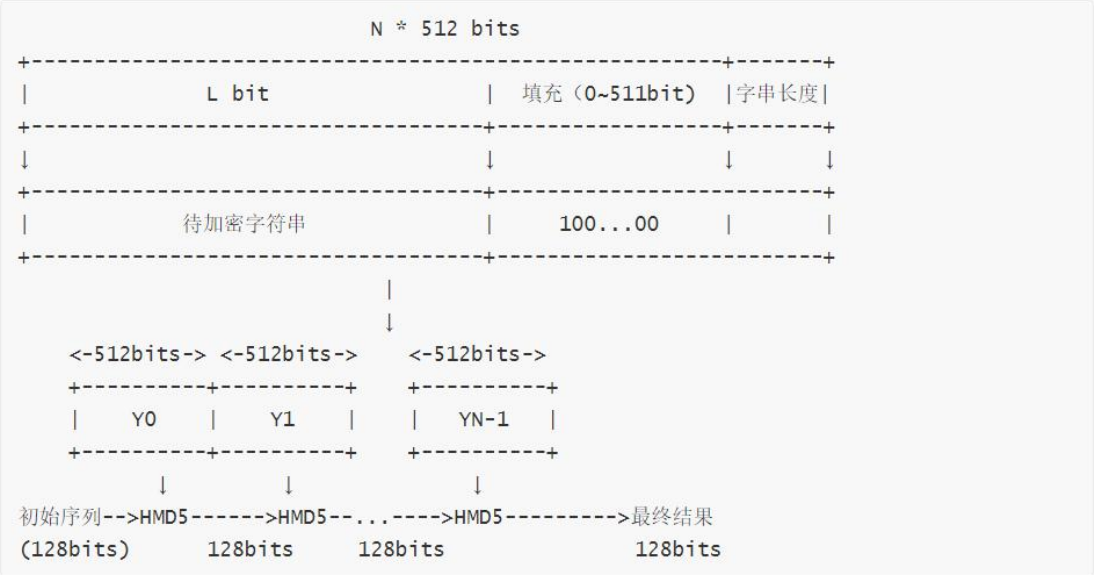
MD5

1991 年，Rivest 开发出技术上更为趋近成熟的 md5 算法。它在 MD4 的基础上增加了"安全-带子"（safety-belts）的概念。虽然 MD5 比 MD4 复杂度大一些，但却更为安全。这个算法很明显的由四个和 MD4 设计有少许不同的步骤组成。在 MD5 算法中，信息-摘要的大小和填充的必要条件与 MD4 完全相同。Den boer 和 Bosselaers 曾发现 MD5 算法中的假冲突（pseudo-collisions），但除此之外就没有其他被发现的加密后结果了。

1.3 MD5 算法原理

对于 MD5 算法可以简要的概括为：MD5 是以 512 位分组来处理输入信息，且每一分组又被划分为 16 个 32 位子分组，经过了一系列的处理后，算法的输出由四个 32 位分组组成，将这四个 32 位分组级联后将生成一个 128 位散列值。

总体流程图如下：



1.3.1 填充

在 MD5 算法中，首先需要对信息进行填充，使其位长对 512 求余的结果等于 448，并且填充必须进行，即使其位长对 512 求余的结果等于 448。因此，信息的位长（Bits Length）将被扩展至 $N*512+448$，N 为一个非负整数，N 可以是零。

填充的方法如下：

1) 在信息的后面填充一个 1 和无数个 0，直到满足上面的条件时才停止用 0 对信息的填充。

2) 在这个结果后面附加一个以 64 位二进制表示的填充前信息长度（bit），如果二进制表示的填充前信息长度超过 64 位，则取低 64 位。

经过这两步的处理，信息的位长为 N * 512 + 448 + 64 = (N + 1) * 512，即长度恰好是 512 的整数倍。这样做的原因是为满足后面处理中对信息长度的要求。

1.3.2 初始化变量

初始的 128 位值为初始链接变量，这些参数用于第一轮的运算，以大端字节序来表示，他们分别为： A=0x01234567， B=0x89ABCDEF， C=0xFEDCBA98， D=0x76543210。

（每一个变量给出的数值是高字节存于内存低地址，低字节存于内存高地址，即大端字节序。在程序中变量 A、B、C、D 的值分别为 0x67452301，0xEFCDAB89，0x98BADCFE， 0x10325476）

1.3.3 处理分组数据

每一分组的算法流程如下：

第一分组需要将上面四个链接变量复制到另外四个变量中：A 到 a，B 到 b，C 到 c，D 到 d。从第二分组开始的变量为上一分组的运算结果，即 A = a，B = b，C = c， D = d。

主循环有四轮（MD4 只有三轮），每轮循环都很相似。第一轮进行 16 次操作。每次操作对 a、b、c 和 d 中的其中三个作一次非线性函数运算，然后将所得结果加上第四个变量，文本的一个子分组和一个常数。再将所得结果向左环移一个不定的数，并加上 a、b、c 或 d 中之一。最后用该结果取代 a、b、c 或 d 中之一。

以下是每次操作中用到的四个非线性函数（每轮一个）。

```
F( X ,Y ,Z ) = ( X & Y ) | ( (~X) & Z )
G( X ,Y ,Z ) = ( X & Z ) | ( Y & (~Z) )
H( X ,Y ,Z ) = X ^ Y ^ Z
I( X ,Y ,Z ) = Y ^ ( X | (~Z) )
```

这四个函数的说明：如果 X、Y 和 Z 的对应位是独立和均匀的，那么结果的每一位也应是独立和均匀的。

F 是一个逐位运算的函数。即，如果 X，那么 Y，否则 Z。函数 H 是逐位奇偶操作的函数。

假设 M_j 表示消息的第 j 个子分组（从 0 到 15），常数 t_i 是 4294967296 * abs(sin(i))的整数部分， i 取值从 1 到 64，单位是弧度(rad)。(4294967296=232)

现定义：

FF(a ,b ,c ,d ,M_j ,s ,t_i) 操作为 a = b + ((a + F(b,c,d) + Mj + ti) << s)

GG(a ,b ,c ,d ,M_j ,s ,t_i) 操作为 a = b + ((a + G(b,c,d) + Mj + ti) << s)

HH(a ,b ,c ,d ,M_j ,s ,t_i) 操作为 a = b + ((a + H(b,c,d) + Mj + ti) << s)

II(a ,b ,c ,d ,M_j ,s ,t_i) 操作为 a = b + ((a + I(b,c,d) + Mj + ti) << s)

注意：这里的<<表示循环左移位，不是左移位。

这四轮（共 64 步）是：

第一轮

```
FF(a ,b ,c ,d ,M0 ,7 ,0xd76aa478)
FF(d ,a ,b ,c ,M1 ,12 ,0xe8c7b756)
FF(c ,d ,a ,b ,M2 ,17 ,0x242070db)
FF(b ,c ,d ,a ,M3 ,22 ,0xc1bdceee)
FF(a ,b ,c ,d ,M4 ,7 ,0xf57c0faf)
FF(d ,a ,b ,c ,M5 ,12 ,0x4787c62a)
FF(c ,d ,a ,b ,M6 ,17 ,0xa8304613)
FF(b ,c ,d ,a ,M7 ,22 ,0xfd469501)
FF(a ,b ,c ,d ,M8 ,7 ,0x698098d8)
FF(d ,a ,b ,c ,M9 ,12 ,0x8b44f7af)
FF(c ,d ,a ,b ,M10 ,17 ,0xffff5bb1)
FF(b ,c ,d ,a ,M11 ,22 ,0x895cd7be)
FF(a ,b ,c ,d ,M12 ,7 ,0x6b901122)
FF(d ,a ,b ,c ,M13 ,12 ,0xfd987193)
FF(c ,d ,a ,b ,M14 ,17 ,0xa679438e)
FF(b ,c ,d ,a ,M15 ,22 ,0x49b40821)
```

第二轮

```
GG(a ,b ,c ,d ,M1 ,5 ,0xf61e2562)
GG(d ,a ,b ,c ,M6 ,9 ,0xc040b340)
GG(c ,d ,a ,b ,M11 ,14 ,0x265e5a51)
GG(b ,c ,d ,a ,M0 ,20 ,0xe9b6c7aa)
GG(a ,b ,c ,d ,M5 ,5 ,0xd62f105d)
GG(d ,a ,b ,c ,M10 ,9 ,0x02441453)
GG(c ,d ,a ,b ,M15 ,14 ,0xd8a1e681)
GG(b ,c ,d ,a ,M4 ,20 ,0xe7d3fbc8)
GG(a ,b ,c ,d ,M9 ,5 ,0x21e1cde6)
GG(d ,a ,b ,c ,M14 ,9 ,0xc33707d6)
GG(c ,d ,a ,b ,M3 ,14 ,0xf4d50d87)
GG(b ,c ,d ,a ,M8 ,20 ,0x455a14ed)
```

```
GG(a ,b ,c ,d ,M13 ,5 ,0xa9e3e905)
GG(d ,a ,b ,c ,M2 ,9 ,0xfcefa3f8)
GG(c ,d ,a ,b ,M7 ,14 ,0x676f02d9)
GG(b ,c ,d ,a ,M12 ,20 ,0x8d2a4c8a)
```

第三轮

```
HH(a ,b ,c ,d ,M5 ,4 ,0xfffa3942)
HH(d ,a ,b ,c ,M8 ,11 ,0x8771f681)
HH(c ,d ,a ,b ,M11 ,16 ,0x6d9d6122)
HH(b ,c ,d ,a ,M14 ,23 ,0xfde5380c)
HH(a ,b ,c ,d ,M1 ,4 ,0xa4beea44)
HH(d ,a ,b ,c ,M4 ,11 ,0x4bdecfa9)
HH(c ,d ,a ,b ,M7 ,16 ,0xf6bb4b60)
HH(b ,c ,d ,a ,M10 ,23 ,0xbebfbc70)
HH(a ,b ,c ,d ,M13 ,4 ,0x289b7ec6)
HH(d ,a ,b ,c ,M0 ,11 ,0xeaa127fa)
HH(c ,d ,a ,b ,M3 ,16 ,0xd4ef3085)
HH(b ,c ,d ,a ,M6 ,23 ,0x04881d05)
HH(a ,b ,c ,d ,M9 ,4 ,0xd9d4d039)
HH(d ,a ,b ,c ,M12 ,11 ,0xe6db99e5)
HH(c ,d ,a ,b ,M15 ,16 ,0x1fa27cf8)
HH(b ,c ,d ,a ,M2 ,23 ,0xc4ac5665)
```

第四轮

```
II(a ,b ,c ,d ,M0 ,6 ,0xf4292244)
II(d ,a ,b ,c ,M7 ,10 ,0x432aff97)
II(c ,d ,a ,b ,M14 ,15 ,0xab9423a7)
II(b ,c ,d ,a ,M5 ,21 ,0xfc93a039)
II(a ,b ,c ,d ,M12 ,6 ,0x655b59c3)
II(d ,a ,b ,c ,M3 ,10 ,0x8f0ccc92)
II(c ,d ,a ,b ,M10 ,15 ,0xffeff47d)
II(b ,c ,d ,a ,M1 ,21 ,0x85845dd1)
II(a ,b ,c ,d ,M8 ,6 ,0x6fa87e4f)
II(d ,a ,b ,c ,M15 ,10 ,0xfe2ce6e0)
II(c ,d ,a ,b ,M6 ,15 ,0xa3014314)
II(b ,c ,d ,a ,M13 ,21 ,0x4e0811a1)
II(a ,b ,c ,d ,M4 ,6 ,0xf7537e82)
II(d ,a ,b ,c ,M11 ,10 ,0xbd3af235)
II(c ,d ,a ,b ,M2 ,15 ,0x2ad7d2bb)
II(b ,c ,d ,a ,M9 ,21 ,0xeb86d391)
```

所有这些完成之后，将 `a`、`b`、`c`、`d` 分别在原来基础上再加上 `A`、`B`、`C`、`D`。

即 a = a + A，b = b + B，c = c + C，d = d + D;

然后用下一分组数据继续运行以上算法。

1.3.4 输出

最后的输出是 a、b、c 和 d 的级联。

以下是前面所述 MD5 算法实现的一些运算结果:

```
MD5 ("") = d41d8cd98f00b204e9800998ecf8427e
MD5 ("a") = 0cc175b9c0f1b6a831c399e269772661
MD5 ("abc") = 900150983cd24fb0d6963f7d28e17f72
MD5 ("message digest") = f96b697d7cb7938d525a2f31aaf161d0
MD5 ("abcdefghijklmnopqrstuvwxyz") = c3fcd3d76192e4007dfb496cca67e13b
MD5 ("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz") =
f29939a25efabaef3b87e2cbfe641315
MD5 ("8a683566bcc7801226b3d8b0cf35fd97") =cf2cb5c89c5e5eeebef4a76becddfcfd
```

1.4 MD5 应用

1.4.1 一致性验证

MD5 的典型应用是对一段信息（Message）产生信息摘要（Message-Digest），以防止被篡改。比如，在 Unix 下有很多软件在下载的时候都有一个文件名相同，文件扩展名为.md5 的文件，在这个文件中通常只有一行文本，大致结构如：

MD5 (tanajiya.tar.gz) = 38b8c2c1093dd0fec383a9d9ac940515

这就是 tanajiya.tar.gz 文件的数字签名。MD5 将整个文件当作一个大文本信息，通过其不可逆的字符串变换算法，产生了这个唯一的 MD5 信息摘要。为了让读者朋友对 MD5 的应用有个直观的认识，笔者以一个比方和一个实例来简要描述一下其工作过程：

大家都知道，地球上任何人都有自己独一无二的指纹，这常常成为司法机关鉴别罪犯身份最值得信赖的方法；与之类似，MD5 就可以为任何文件（不管其大小、格式、数量）产生一个同样独一无二的“数字指纹”，如果任何人对文件做了任何改动，其 MD5 值也就是对应的“数字指纹”都会发生变化。

我们常常在某些软件下载站点的某软件信息中看到其 MD5 值，它的作用就在于我们可以在下载该软件后，对下载回来的文件用专门的软件（如 Windows MD5 Check 等）做一次 MD5 校验，以确保我们获得的文件与该站点提供的文件为同一文件。

具体来说文件的 MD5 值就像是这个文件的“数字指纹”。每个文件的 MD5 值是不同的，如果任何人对文件做了任何改动，其 MD5 值也就是对应的“数字指纹”就会发生变化。比如下载服务器针对一个文件预先提供一个 MD5 值，用户下载完该文件后，用我这个算法重新计算下载文件的 MD5 值，通过比较这两个值是否相同，就能判断下载的文件是否出错，或者说下载的文件是否被篡改了。

利用 MD5 算法来进行文件校验的方案被大量应用到软件下载站、论坛数据库、系统文件安全等方面。

1.4.2 数字签名

MD5 的典型应用是对一段 Message(字节串)产生 fingerprint(指纹），以防止被“篡改”。举个例子，你将一段话写在一个叫 readme.txt 文件中，并对这个 readme.txt 产生一个 MD5 的值并记录在案，然后你可以传播这个文件给别人，别人如果修改了文件中的任何内容，你对这个文件重新计算 MD5 时就会发现（两个 MD5 值不相同）。如果再有一个第三方的认证机构，用 MD5 还可以防止文件作者的“抵赖”，这就是所谓的数字签名应用。

1.4.3 安全访问认证

MD5 还广泛用于操作系统的登陆认证上，如 Unix、各类 BSD 系统登录密码、数字签名等诸多方面。如在 Unix 系统中用户的密码是以 MD5（或其它类似的算法）经 Hash 运算后存储在文件系统中。当用户登录的时候，系统把用户输入的密码进行 MD5 Hash 运算，然后再去和保存在文件系统中的 MD5 值进行比较，进而确定输入的密码是否正确。通过这样的步骤，系统在并不知道用户密码的明码的情况下就可以确定用户登录系统的合法性。这可以避免用户的密码被具有系统管理员权限的用户知道。MD5 将任意长度的“字节串”映射为一个 128bit 的大整数，并且是通过该 128bit 反推原始字符串是困难的，换句话说就是，即使你看到源程序和算法描述，也无法将一个 MD5 的值变换回原始的字符串，从数学原理上说，是因为原始的字符串有无穷多个，这有点象不存在反函数的数学函数。所以，要遇到了 md5 密码的问题，比较好的办法是：你可以用这个系统中的 md5()函数重新设一个密码，如 admin，把生成的一串密码的 Hash 值覆盖原来的 Hash 值就行了。

1.5 MD5 特点

MD5 算法具有以下特点：

1、压缩性：任意长度的数据，算出的 MD5 值长度都是固定的。

2、容易计算：从原数据计算出 MD5 值很容易。

3、抗修改性：对原数据进行任何改动，哪怕只修改 1 个字节，所得到的 MD5 值都有很大区别。

4、强抗碰撞：已知原数据和其 MD5 值，想找到一个具有相同 MD5 值的数据（即伪造数据）是非常困难的。

MD5 的作用是让大容量信息在用数字签名软件签署私人密钥前被"压缩"成一种保密的格式（就是把一个任意长度的字节串变换成一定长的十六进制数字串）。除了 MD5 以外，其中比较有名的还有 sha-1、RIPEMD 以及 Haval 等。

1.6 java 代码实现 MD5 摘要算法

java 实现 MD5 摘要算法的相关文件位于 Quartus 项目文件夹的 java/文件夹内，用于检验 Verilog 语言实现的 MD5 算法正确性.

```
import java.io.*;
public class MD5 {
  //四个链接变量
      private final int A = 0x67452301;
      private final int B = 0xefcdab89;
      private final int C = 0x98badcfe;
      private final int D = 0x10325476;
      //ABCD's temp val
      private int Atemp,Btemp,Ctemp,Dtemp;

      /*
      *常量 ti
      *公式:floor(abs(sin(i+1))×(2pow32)
      */
      private final int K[]={
          0xd76aa478,0xe8c7b756,0x242070db,0xc1bdceee,
          0xf57c0faf,0x4787c62a,0xa8304613,0xfd469501,0x698098d8,
          0x8b44f7af,0xffff5bb1,0x895cd7be,0x6b901122,0xfd987193,
          0xa679438e,0x49b40821,0xf61e2562,0xc040b340,0x265e5a51,
          0xe9b6c7aa,0xd62f105d,0x02441453,0xd8a1e681,0xe7d3fbc8,
          0x21e1cde6,0xc33707d6,0xf4d50d87,0x455a14ed,0xa9e3e905,
          0xfcefa3f8,0x676f02d9,0x8d2a4c8a,0xfffa3942,0x8771f681,
          0x6d9d6122,0xfde5380c,0xa4beea44,0x4bdecfa9,0xf6bb4b60,
          0xbebfbc70,0x289b7ec6,0xeaa127fa,0xd4ef3085,0x04881d05,
          0xd9d4d039,0xe6db99e5,0x1fa27cf8,0xc4ac5665,0xf4292244,
          0x432aff97,0xab9423a7,0xfc93a039,0x655b59c3,0x8f0ccc92,
          0xffeff47d,0x85845dd1,0x6fa87e4f,0xfe2ce6e0,0xa3014314,
          0x4e0811a1,0xf7537e82,0xbd3af235,0x2ad7d2bb,0xeb86d391};
      //cnt of shifting left or right
      private final int s[]={7,12,17,22,7,12,17,22,7,12,17,22,7,
          12,17,22,5,9,14,20,5,9,14,20,5,9,14,20,5,9,14,20,
```

```
        4,11,16,23,4,11,16,23,4,11,16,23,4,11,16,23,6,10,
        15,21,6,10,15,21,6,10,15,21,6,10,15,21};


    private void init(){
        Atemp = A;
        Btemp = B;
        Ctemp = C;
        Dtemp = D;
    }
    /*
    *shift right
    */
    private int shift(int a,int s){
        return(a << s) | (a >>> (32 - s));// higher bits set to 0 when shifting right
    }
    /*
    *main loop
    */
    private void MainLoop(int M[]){
        int F, g;
        int a = Atemp;
        int b = Btemp;
        int c = Ctemp;
        int d = Dtemp;
        for(int i = 0; i < 64; i ++){
            if(i < 16){
                F = (b & c) | ((~b) & d);
                g = i;
            }else if(i < 32){
                F = (d & b) | ((~d) & c);
                g = (5 * i + 1) % 16;
            }else if(i < 48){
                F = b ^ c ^ d;
                g = (3 * i + 5) % 16;
            }else{
                F = c ^ (b | (~d));
                g = (7 * i) % 16;
            }
            int tmp = d;
            d = c;
            c = b;
            b = b + shift(a + F + K[i] + M[g], s[i]);
            a = tmp;
```

```
            }
            Atemp = a + Atemp;
            Btemp = b + Btemp;
            Ctemp = c + Ctemp;
            Dtemp = d + Dtemp;
        }
        //填充函数
        private int[] add(String str){
            int num = ((str.length() + 8) / 64) + 1;//512bits, 64bytes as a group
            int strByte[] = new int[num * 16];//64/4=16 integers
            for(int i = 0; i < num * 16; i++){//init 0
                strByte[i] = 0;
            }
            int i;
            for(i = 0; i < str.length(); i++){
                strByte[i >> 2] |= str.charAt(i) << ((i % 4) * 8);//one integer store 4 bytes
            }
            strByte[i >> 2] |= 0x80 << ((i % 4) * 8);//add '1' at the end
            //add origin length
            strByte[num*16-2]=str.length()*8;
                return strByte;
        }
        //call func to encode
        public String getMD5(String source){
            init();
            int strByte[] = add(source);
            for(int i = 0; i < strByte.length / 16; i++){
            int num[] = new int[16];
            for(int j = 0; j < 16;j ++){
                num[j] = strByte[i * 16 + j];
            }
            MainLoop(num);
            }
            return changeHex(Atemp) + changeHex(Btemp) + changeHex(Ctemp) +
changeHex(Dtemp);

        }
        //integer to string
        private String changeHex(int a){
            String str = "";
            for(int i = 0; i < 4; i++){
                str += String.format("%2s", Integer.toHexString(((a >> i * 8) % (1<<8)) &
0xff)).replace(' ', '0');
            }
```

```
            return str;
        }


        private static MD5 instance;
        public static MD5 getInstance(){
            if(instance == null){
                instance = new MD5();
            }
            return instance;
        }

        private MD5(){};

        public static void main(String[] args)throws IOException{
BufferedReader   buf;
buf = new BufferedReader(new InputStreamReader(System.in));
String in = new String("");
System.out.println("Input a string:");
in = buf.readLine();
while(!in.equals(">exit")){
String str = MD5.getInstance().getMD5(in);
System.out.println(str);
System.out.println("Input a string:");
in = buf.readLine();
}
        }
}
```

参考：

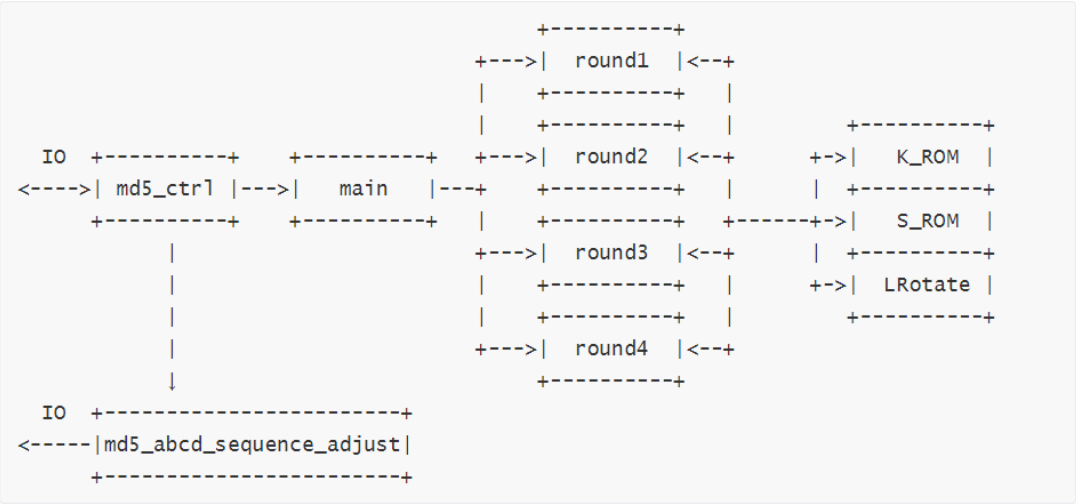
https://en.wikipedia.org/wiki/MD5

http://hubingforever.blog.163.com/blog/static/171040579201210781650340/

1.7 Verilog 实现 MD5 摘要算法

1.7.1 数字抽象

运算部分负责将用户输入的字符串进行加密. 其各模块间关系示意图如下所示:



md5_ctrl: 从 io 模块获取用户输入的字串 [511:0] indata 和开始运算的信号 start. 将由 main 模块提供的加密好的字串 [127:0] MD5code 传给 md5_abcd_sequence_adjust 模块做调序处理, 并将结束信号 finish 发送给 io 模块. 在本模块内实现对待加密字串的长度填充, 将填充完毕的字串分块, 以 512bits 为一块依次送给 main 模块进行 MD5 运算, 并以 start, blockstart 作为开始处理信号传给 main 模块.

main: 接收由 md5_ctrl 模块传入的待加密块 [511:0] indata 和开始信号 start, blockstart. 处理该分组数据, 过程及原理详见 1.3.3 处理分组数据, 这里不再赘述.

其中 4 轮 16 次的操作分别封装于 round1, round2, round3, round4 模块中, main 模块将待加密块和当前处理次数 [3:0] in_round 传给这四个模块进行运算.

round1: 对输入 [511:0] datain 分为四块 a、b、c 和 d 并做 (a + (b & c) | ((~b) & d) + Mj + K) << S 运算(详见 1.3.3 处理分组数据 中的介绍), 运算结果以 [31:0] result 返回给 main 模块. 其中, Mj 是当前处理字串的第 j 个子分组, K, S 分别存放于 K_ROM, S_ROM 中, 由操作次数 [3:0]in_round 作为选择信号. 这里的 << 操作为循环移位, 由模块 LRotate 实现, [4:0] S 为左移次数.

round2: 对输入 [511:0] datain 做 (a + (d & b) | ((~d) & c) + Mj + K) << S 运算(详见 1.3.3 处理分组数据 中的介绍), 运算结果以 [31:0] result 返回给 main 模块.

round3：对输入`[511:0] datain`做`(a + b ^ c ^ d + Mj + K) << S`运算(详见`1.3.3 处理分组数据`中的介绍)，运算结果以`[31:0] result`返回给`main`模块.

round4：对输入`[511:0] datain`做`(a + c ^ (b | (~d)) + Mj + K) << S`运算(详见`1.3.3 处理分组数据`中的介绍)，运算结果以`[31:0] result`返回给`main`模块.

K_ROM：存放`4294967296 * abs(sin(i))`的整数部分，`i`取值从`1`到`64`.

S_ROM：存放每次运算所需的左循环移位数 S.

LRotate：将传入的操作数`[31:0] in`循环左移`[4:0] s`位.

md5_abcd_sequence_adjust：对`md5_ctrl`传入的计算结果`[127:0] MD5_output`的四部分`a`、`b`、`c`和`d`调整为`d`、`c`、`b`和`a`，将处理后的 MD5 串`[1023:0] MD5_result`和长度`[7:0] MD5_result_len`传给 io 相关模块进行输出.

1.7.2 建立模型

md5_ctrl:

```
//接口设计
module md5_ctrl(in_clk, indata, start, len, MD5code, finish);
input start, in_clk;//开始运算信号和时钟信号
   input[63:0] len;//待加密字串的长度
   input[511:0] indata;//输入
output[127:0] MD5code;//运算结果
output reg finish = 1'b0;//运算结束信号

//时序控制信号
reg start_cope = 1'b1, start_block = 1'b1; //总开始信号和分块处理信号
reg last = 1'b0, first = 1'b0, over = 1'b0;
reg blocknop = 1'b0, times = 1'b0;
reg additional = 1'b0, last_finish = 1'b0;
//若输入多于56个字符,则填充的信息会溢出至第二块，需要对第二个512bits的信息块进行运算，使用 reset 信号来重置以上状态变量并计算第二个信息块的 MD5 散列值
reg [1:0]reset = 2'b0;
```

在`start`信号为`1`时，`md5_ctrl`模块将对输入进行填充，填充规则详见`1.3.1 填充`节中的说明. 这里只考虑输入字符数量小于 63 个的情况.

```
if (start) begin
last = 1'b1;
copedata0 = indata;
if(len[8:0] < 9'd448) begin
/*填充
    *处理后应满足 bits≡448(mod512),字节就是 bytes≡56（mode64)
    *填充方式为先加一个 1,其它位补零
    *最后加上 64 位的原来长度
*/
copedata0[len[8:0] + 7] = 1'b1;
copedata0[511:448] = len[63:0];
start_block = 1'b1;
blocknop = 1'b1;
end
else begin
times = 1'b1;
if(len[8:0] < 9'd504) begin
copedata0[len[8:0] + 7] = 1'b1;
end
else begin
copedata1[7] = 1'b1;
end
copedata1[511:448] = len[63:0];
start_block = 1'b1;
blocknop = 1'b1;
end
end
```

具体的运算交给 `main` 模块实现.

md5_ctrl:

```
//接口设计
module main(clk, start, blockstart, indata, finish, MD5code, over);
input clk, start, blockstart, over;
//时钟信号，总开始信号，分块开始信号，结束控制信号
input[511:0] indata;//待加密串
output reg finish = 1'b0;//运算结束信号
output[127:0] MD5code;//运算结果
```

参照 `1.3.2 初始化变量` 节，初始链接变量`[127:0] linkvar` 初始化为：

```
initial begin
   linkvar[31:0] = 32'h67452301;
   linkvar[63:32] = 32'hefcdab89;
   linkvar[95:64] = 32'h98badcfe;
   linkvar[127:96] = 32'h10325476;
end
```

参照 1.3.3 处理分组数据节，需要将输入的字串做 FF(), GG(), HH(), II()四种运算各 16 次，共 64 次．以模块 round1, round2, round3, round4 分别实现上面这四个函数中的((a + X(b,c,d) + Mj + ti) << s)部分，返回结果为[31:0] result1，[31:0] result2，[31:0] result3，[31:0] result4．以[6:0] cnt 记录已运算次数，integer endrounds = 7'd64 为总运算次数．运算用到的 4 个变量 a,b,c,d 值的更新实现如下：

```
if(cnt < endrounds) begin
a_next = d;
d_next = c;
c_next = b;
if(cnt <= 7'd15)
b_next = b + result1;
else if(cnt <= 7'd31)
b_next = b + result2;
else if(cnt <= 7'd47)
b_next = b + result3;
else if(cnt <= 7'd63)
b_next = b + result4;
end
else begin
a_next = a;
b_next = b;
c_next = c;
d_next = d;
end
```

每完成一次运算 cnt 将加 1，当 cnt >= endrounds 时将 finish 信号置 1，标志着运算结束，将由[127:0] MD5code 返回计算结果:

```
assign MD5code = { a, b, c, d };
```

round1:

```
//接口设计
module round1(indata, in_round, link_var, result);
```

```
  input [511:0] datain; //输入字串
  input [3:0] rounds; //当前运算次数
  input [127:0] link_var; //链接变量
  output [31:0] result; //当前运算结果
```

round1 模块返回表达式(a + (b & c) | ((~b) & d) + Mj + K) << S 的值.其中的 a、b、c、 d 分别为

链接变量中的 4 个 int 型变量:

```
assign A = link_var[31:0];
assign B = link_var[63:32];
assign C = link_var[95:64];
assign D = link_var[127:96];
```

Mj 的值由下式求得:

```
assign M = datain[{g, 5'd0} +: 32];
// 算符'+:' 变量[起始地址 +: 数据位宽] <-等价于-> 变量[(起始地址+数据位宽-1):起始地址]
//变量[结束地址 -: 数据位宽] <-等价于-> 变量[结束地址：(结束地址-数据位宽+1)]
```

K, S 的值由 K_ROM, S_ROM 模块根据 rounds 的值查表给出.

单次的运算结果如下求得：

```
assign F = ((B & C) | (~B & D)) + A + K + M;
LRotate lr1(.in(F), .s(S), .out(result));
```

LRotate 模块将中间结果 F 循环左移 S 位即可得到结果 result.

round2, **round3**, **round4** 的实现与 **round1** 类似，只是对中间结果 F 的求值方式有所不同. 具体的表达式在 1.3.3 处理分组数据节和 3.1 数字抽象中已有所介绍，这里不再赘述.

K_ROM:

供 **round[i]**模块查询当前运算用到的 K 值. 为缩短运算时长，使用 case 语句实现.

```
module K_ROM(cnt, K);
  input [5:0] cnt;
  output reg [31:0] K;
```

```
    always @(*)
        case(cnt)
            6'd0: K = 32'hd76aa478;
            6'd1: K = 32'he8c7b756;
            6'd2: K = 32'h242070db;
            //.....
            6'd63: K = 32'heb86d391;
        endcase
endmodule
```

S_ROM:

供 **round[i]**模块查询当前运算用到的 S 值. 为缩短运算时长，使用 case 语句实现.

```
module S_ROM(cnt, S);
input [5:0] cnt;
    output reg [4:0] S;

    always @(*)
        case(cnt)
            6'd0: S = 5'd7;
            6'd1: S = 5'd12;
            6'd2: S = 5'd17;
            //.....
            6'd63: S = 5'd21;
        endcase
endmodule
```

LRotate:

```
//接口设计
module LRotate(in, s, out); //left shift bit in rotation
    input [31:0] in; //待旋转数据
    input [4:0] s; //左旋 s 位
    output reg [31:0] out; //旋转结果
```

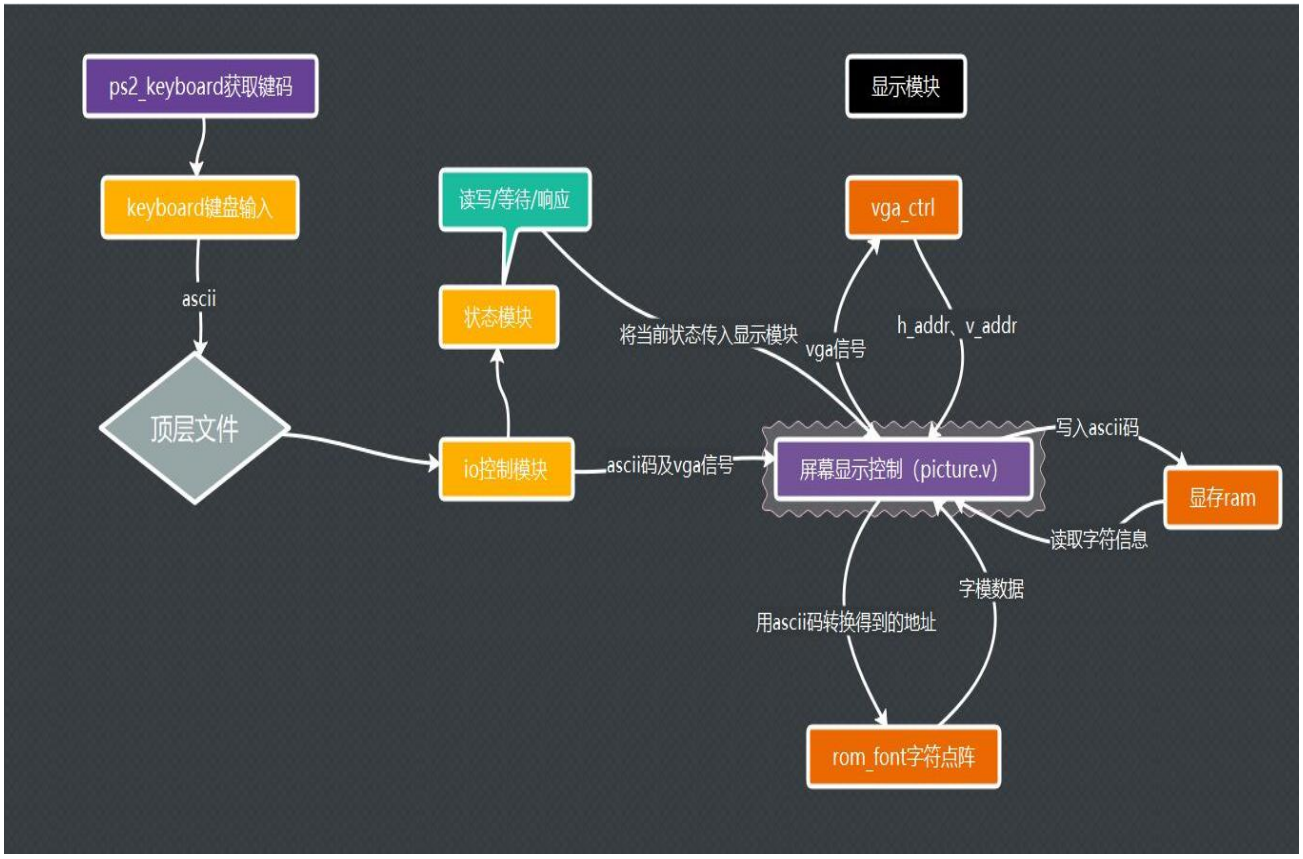
左旋位数 s 满足 0 <= s <= 31，表达式通式为 out = {in[30 - s: 0], in[31: 31-s]};模块内对不同的 s 值用 case 语句实现.

```
case(s)
5'd0: out = in;
5'd1: out = {in[30:0], in[31:31]};
5'd2: out = {in[29:0], in[31:30]};
```

```
5'd3: out = {in[28:0], in[31:29]};
//.............
5'd31: out = {in[0], in[31:1]};
endcase
```

第二部分——io 输入输出

*模块划分



2.1 键盘输入模块

此模块在之前的几次基础实验中多次复用到

在音频实验中使用的是一段式状态机实现多个按键和声

本实验中套用了二段式状态机的模板

2.1.1 原理

二段式状态机：用两个 always 块来描述状态转移和输出的状态机

其中第一个 always 模块采用同步时序描述状态转移；

第二个 always 模块采用组合逻辑判断状态转移条件，描述状态转移规律以及输出

较一段式状态机而言，更利于代码的优化与条件的添加。

其他具体的内容与一段式的相近，不再详细赘述

参考：

http://web.stanford.edu/class/ee183/tutorial/ee183_tutorial.pdf

2.1.2 具体代码

```
/* 现态逻辑 */
    ps2_keyboard ps2_kbd(.clk(clk), .clrn(1'b1), .ps2_clk(ps2_clk), .ps2_data(ps2_data),
    .data(data), .ready(ready), .nextdata_n(~next_data_next), .overflow(overflow));

    always @(posedge clk) begin
        cur_in_info <= next_in_info;
        cur_end_mark <= next_end_mark;
        cur_show_data <= next_show_data;
        cur_shift <= next_shift;
        cur_move_next <= next_data_next;
    end
```

现态逻辑中调用 ps2_keyboard 模块获取按键码。

```
        next_in_info = cur_in_info;
        next_show_data = cur_show_data;
        next_shift = cur_shift;
        next_data_next = ready;

        //读取数据
        if(ready) begin
            if(data == 8'hf0) next_end_mark = 1'b1;
            else begin
                next_show_data = data;
                if(cur_end_mark) begin
                    next_in_info[data] = 1'b0;
                    next_show_data = 8'hff;
                    next_end_mark = 1'b0;

                    if(data == 8'h12 || data == 8'h59) begin
                        next_shift = 1'b0;
                    end
                    //if(data == 8'h58) begin
                       // cap = ~cap;
                   // end
                end
                else if(!cur_in_info[data]) begin
                    next_in_info[data] = 1'b1;
                    if(data == 8'h12 || data == 8'h59) begin
                        next_shift = 1'b1;
                    end
                    /*if(data == 8'h58) begin
                        cap = ~cap;
                    end*/

                end
                else begin
                end
                next_end_mark = 1'b0;
            end
        end
        else begin
            next_show_data = cur_show_data;
        end
    end


    assign cur_data = next_show_data;
    data_2_asc d2a(.cap(cap), .shift(next_shift), .data(cur_data), .ascii(ascii));
    assign valid = ascii != 8'd0;
endmodule
```

次态逻辑下进行数据的转换（data_2_asc）与输出（ascii）

2.1.3 遇到的一些问题（时序）

1. 在最初调试过程中，起初 data_2_asc 模块沿用了之前的 readmemh 读取.txt 文件中的键码对应 ascii 数据的方式，发现与显示模块配合时，读取时间过长。故直接改成 case 语句对 ascii 码进行赋值，分为 shift 按下和 shift 松开两种情况（放弃了 cap 实现大小写）。
2. 接 1 中放弃通过判断 cap 大写锁定是否打开来输入大写字符的想法，是因为加入对 caps 状态的赋值判断后，字符输入时的显示屏上会偶尔出现一些像素点闪烁的现象（未知 bug），猜测依旧是时序除了小问题，故删除了这一状态，直接改为用 shift 进行大小写输入。

2.1.4 data_2_asc 代码

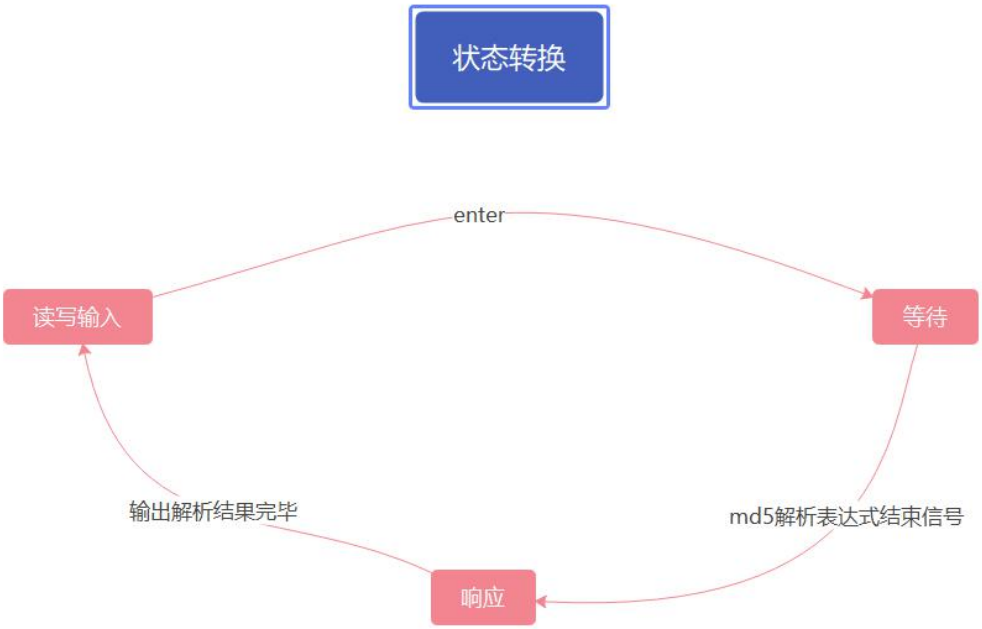
```
module data_2_asc(cap, shift, data, ascii);
    input shift, cap;
    input [7:0] data;
    output reg [7:0] ascii;
    reg [7:0] ascii_no_shift, ascii_shift;
   /*
reg [7:0] asc[255:0];
reg [7:0] shi[255:0];
initial
begin
    //$readmemh("F:/quartus/keyboard/ascii.txt", asc, 0, 255);
    $readmemh ("ascii.txt", asc, 0, 255);
    $readmemh ("shift.txt", shi, 0, 255);
end
   */
    always @* begin
        ascii = shift ? ascii_shift : ascii_no_shift;
    end
   /* always @*
begin
   if (data != 8'hff && data != 8'h00)begin
   ascii_no_shift = asc[data];
   ascii_shift = shi[data];
   end
   else begin
   ascii_no_shift = 8'h00;
   ascii_shift = 8'h00;
   end
end */


    always @* begin
        case(data)
            //alphabet
            8'h1c:  ascii_no_shift = 8'h61;
            8'h32:  ascii_no_shift = 8'h62;
            8'h21:  ascii_no_shift = 8'h63;
            8'h23:  ascii_no_shift = 8'h64;
            8'h24:  ascii_no_shift = 8'h65;
            8'h2b:  ascii_no_shift = 8'h66;
            8'h34:  ascii_no_shift = 8'h67;
            8'h33:  ascii_no_shift = 8'h68;
            8'h43:  ascii_no_shift = 8'h69;
            8'h3b:  ascii_no_shift = 8'h6a;
            8'h42:  ascii_no_shift = 8'h6b;
            8'h4b:  ascii_no_shift = 8'h6c;
            8'h3a:  ascii_no_shift = 8'h6d;
            8'h31:  ascii_no_shift = 8'h6e;
            8'h44:  ascii_no_shift = 8'h6f;
            8'h4d:  ascii_no_shift = 8'h70;
            8'h15:  ascii_no_shift = 8'h71;
            8'h2d:  ascii_no_shift = 8'h72;
            8'h1b:  ascii_no_shift = 8'h73;
            8'h2c:  ascii_no_shift = 8'h74;
            8'h3c:  ascii_no_shift = 8'h75;
            8'h2a:  ascii_no_shift = 8'h76;
            8'h1d:  ascii_no_shift = 8'h77;
            8'h22:  ascii_no_shift = 8'h78;
            8'h35:  ascii_no_shift = 8'h79;
            8'h1a:  ascii_no_shift = 8'h7a;
            //number from 0 to 9
            8'h45:  ascii_no_shift = 8'h30;
            8'h16:  ascii_no_shift = 8'h31;
            8'h1e:  ascii_no_shift = 8'h32;
            8'h26:  ascii_no_shift = 8'h33;
            8'h25:  ascii_no_shift = 8'h34;
            8'h2e:  ascii_no_shift = 8'h35;
            8'h36:  ascii_no_shift = 8'h36;
            8'h3d:  ascii_no_shift = 8'h37;
            8'h3e:  ascii_no_shift = 8'h38;
            8'h46:  ascii_no_shift = 8'h39;
            8'h29:  ascii_no_shift = 8'h20; // empty space
            8'h66:  ascii_no_shift = 8'h08; //backspace
            8'h5a:  ascii_no_shift = 8'h0d; //enter
            8'h0d:  ascii_no_shift = 8'h09; // tab
            8'h76:  ascii_no_shift = 8'h1b; // esc
            8'h54:  ascii_no_shift = 8'h5b; // [
            8'h5b:  ascii_no_shift = 8'h5d; // ]
            8'h4c:  ascii_no_shift = 8'h3b; // ;
            8'h52:  ascii_no_shift = 8'h27; // '
            8'h41:  ascii_no_shift = 8'h2c; // ,
            8'h49:  ascii_no_shift = 8'h2e; // .
            8'h4a:  ascii_no_shift = 8'h2f; // /
            8'h55:  ascii_no_shift = 8'h3d;// =
            8'h4e:  ascii_no_shift = 8'h2d;// -
            8'h5d:  ascii_no_shift = 8'h5c;// \
            8'h0e:  ascii_no_shift = 8'h60; // `
            default: ascii_no_shift = 8'h00;
        endcase
    end
```

```
    always @* begin
        case(data)
            8'h1c:  ascii_shift = 8'h41;
            8'h32:  ascii_shift = 8'h42;
            8'h21:  ascii_shift = 8'h43;
            8'h23:  ascii_shift = 8'h44;
            8'h24:  ascii_shift = 8'h45;
            8'h2b:  ascii_shift = 8'h46;
            8'h34:  ascii_shift = 8'h47;
            8'h33:  ascii_shift = 8'h48;
            8'h43:  ascii_shift = 8'h49;
            8'h3b:  ascii_shift = 8'h4a;
            8'h42:  ascii_shift = 8'h4b;
            8'h4b:  ascii_shift = 8'h4c;
            8'h3a:  ascii_shift = 8'h4d;
            8'h31:  ascii_shift = 8'h4e;
            8'h44:  ascii_shift = 8'h4f;
            8'h4d:  ascii_shift = 8'h50;
            8'h15:  ascii_shift = 8'h51;
            8'h2d:  ascii_shift = 8'h52;
            8'h1b:  ascii_shift = 8'h53;
            8'h2c:  ascii_shift = 8'h54;
            8'h3c:  ascii_shift = 8'h55;
            8'h2a:  ascii_shift = 8'h56;
            8'h1d:  ascii_shift = 8'h57;
            8'h22:  ascii_shift = 8'h58;
            8'h35:  ascii_shift = 8'h59;
            8'h1a:  ascii_shift = 8'h5a;
            8'h16:  ascii_shift = 8'h21;//!
            8'h1e:  ascii_shift = 8'h40;//@
            8'h26:  ascii_shift = 8'h23;//#
            8'h25:  ascii_shift = 8'h24;//$
            8'h2e:  ascii_shift = 8'h25;//%
            8'h36:  ascii_shift = 8'h5e;//^
            8'h3d:  ascii_shift = 8'h26;//&
            8'h3e:  ascii_shift = 8'h2a;//*
            8'h46:  ascii_shift = 8'h28;//(
            8'h45:  ascii_shift = 8'h29;//)
            8'h54:  ascii_shift = 8'h7b; // {
            8'h5b:  ascii_shift = 8'h7d; // }
            8'h4c:  ascii_shift = 8'h3a;// :
            8'h52:  ascii_shift = 8'h22;// "
            8'h41:  ascii_shift = 8'h3c;// <
            8'h49:  ascii_shift = 8'h3e;// >
            8'h4a:  ascii_shift = 8'h3f;// ?
            8'h0e:  ascii_shift = 8'h7e;// ~
            8'h5d:  ascii_shift = 8'h7c;// |
            8'h55:  ascii_shift = 8'h2b; // +
            8'h4e:  ascii_shift = 8'h5f; // _
            default: ascii_shift = 8'h00;
    endcase
end
endmodule
```

2.2 状态控制模块

2.2.1 状态转移图



一共分为 3 个状态：读写输入 READ，等待 WAIT，响应 RESPONSE
分别代表：从键盘读取输入、等待 md5 算法部分解析、接收 md5 解析结果

2.2.2 具体实现

首先状态初始化为 READ

```
initial
begin
    state = `READ;
    length_cnt = 8'd0;
    res_temp = 8'd0;
    finish = 1'b0;
end
```

1. READ 状态下

读入键盘输入并传给显示模块进行显存读写输入

```
if(cur_kb_ascii == `ENTER) begin
    line_finish <= 1'b1;
    state <= `WAIT;
    en <= 1'b1;
end
```

2. 接收到回车键 enter 后状态切换到 WAIT 等待阶段，此时状态模块控制显示模块什么也不做，等待 MD5 内核模块进行解析操作。

```
`WAIT: begin
    if(ready) begin
        state <= `RESPONSE;
        length_cnt <= result_len;
    end
```

3. 解析完成后，将传回一个完成的 ready 信号和解析结果 result，状态模块接收到该信号有效时，切换到响应阶段。
4. 相应阶段下由 io 控制模块逐个字符读出 result 块中的数据并传入显示模块进行显示。

```
`RESPONSE: begin
    if(res_temp != length_cnt) begin
        en <= 1'b1;
        begin
            ascii_in <= cur_ascii;
            res_temp <= res_temp + 8'd1;
        end

        state <= `RESPONSE;
    end
    else begin
        en <= 1'b1;
        ascii_in <= `ENTER;
        state <= `READ;
```

当 result 块中的数据全部读完之后，响应阶段切换到读取键盘输入阶段，继续从键盘获取输入，完成一个状态转移的周期。

2.2.3 与 MD5 模块的交互

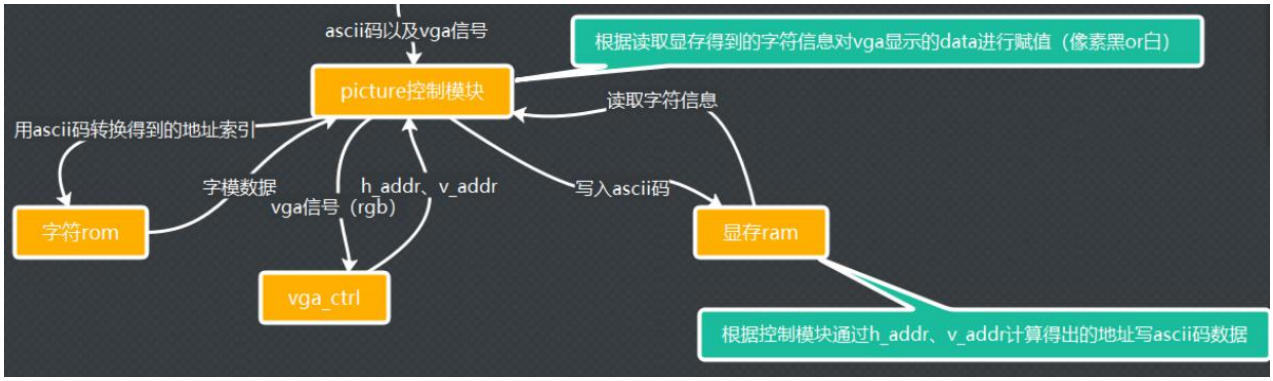
键盘输入有效时（接受的按键码都是常规字符且处于 READ 状态），需要将读入的 ascii 码存入一个**数据块并输出给 MD5 模块（MD5 模块将进行对该数据块的处理）**，在接收到回车 enter 状态转移到 WAIT 阶段的同时将 finish 信号置为 1，MD5 模块接收到 finish 信号高有效时开始对该数据块进行处理，同时停止接收键盘输入

```
always @(posedge kb_clk)
begin
    if(input_valid) begin
        if(kb_input == kb_pre_input && cnt < 12) begin
            cnt <= cnt + 1;
        end
        else begin
            if(kb_input != kb_pre_input) begin
                cnt <= 0;
                kb_pre_input <= kb_input;
                kb_buff[kb_buff_rear+:8] <= kb_input;
                kb_buff_rear <= kb_buff_rear + 10'd8;
            end
            else begin
                if(cnt == 0) begin
                    kb_pre_input <= kb_input;
                    kb_buff[kb_buff_rear+:8] <= kb_input;
                    kb_buff_rear <= kb_buff_rear + 10'd8;
                end
                cnt <= cnt + 1;
            end
        end
    end
    else begin
        cnt <= 0;
        kb_pre_input <= 8'd0;
    end
end
```

数据块用一个 kb_buff 的数组模拟一个队列实现。
其中用一个整型变量 cnt 处理按键长按问题
与实验十一中处理长按的原理相同，当当前读取按键的码与前一个相同时，cnt++，直到 cnt 大于等于 12 时，认为是在长按该按键，进行数据块的存储与显示器的输出，在 cnt 小于 12 的过程中，始终默认只输出、存储一次该按键数据，其他文件设备不进行任何操作。

2.3 显示模块

2.3.1 原理



实验十一原理模块划分图（部分）

1. 我们用 vga_ctrl（时钟信号是用 clkgen 生成的 VGA_CLK）获取 h_addr 和 v_addr 的数据并做一系列运算后，要去 rom 中读取字模数据，这两个过程始终是一起的，共用同一个时钟信号 clk。

```
rom_font rf(.address({ascii, mod16}), .clock(clk), .q(font));

vga_ctrl vc(.pclk(VGA_CLK), .reset(1'b0), .vga_data(vga_data), .h_addr(scan_h), .v_addr(scan_v), .hsync(hsync), .vsync(vsync),
.valid(valid), .vga_r(vga[23:16]), .vga_g(vga[15:8]), .vga_b(vga[7:0]));
```

2. 之后是写显存、读显存的过程，选择双核 ram。

3. 显示光标则又需要一个频率比以上慢得多的时钟信号

```
clkgen #(5) c1(.clkin(~clk), .rst(1'b0), .clken(1'b1), .clkout(curse_clk));
always @*
begin
//curse
    if(curse_h == vga_h && curse_v == vga_v && mod9 == 4'b0) begin
        vga_data = curse_clk ? `PINK : 24'h000000;
    end
    else begin
        vga_data = font[mod9] ? 24'hffffff : 24'h000000;
    end
end
```

2.3.2 字符 rom & 显存 ram

根据 vga_font.txt 生成的 mif 文件大小如图所示(可以通过 excel 和 quartus 中的 txt 文件实现用 txt 文件完成 mif 文件的初始化），建立单核只读 rom
再将字符 rom 中读取的字模信息写入显存，同时再从显存中读取数据，是一个边写边读的过程。所以要建立一个双核的读写 ram
（具体内容同实验十一）

2.3.3 地址计算及处理

显示器 640x480 的像素映射到显存中 70x30 块，
v_addr 要除以 480/30（=16） 得到显存 ram 中对应位置的横坐标，
h_addr 除以 9 得到显存 ram 中对应位置的纵坐标。除以 16 即右移 4 位实现。**再将所得的（横坐标，纵坐标）转换为一维线性地址，即横坐标 * 70 + 纵坐标，即为显存 ram 中的地址**。
但是，代码测试时发现，乘法处理过慢会导致一系列的时序问题，**故直接改成一行显示 64 个字符，即转换为一维线性地址时，用横坐标 * 64 + 纵坐标，乘 64** 可以用移位快速实现。

```
ram ram1(.clock(~clk), .data(next_wr_char), .rdaddress({vga_v, vga_h}),
.wraddress({next_wr_v, next_wr_h}), .wren(next_wr_en), .q(ascii));
```

如图，vga_v 是 v_addr 右移四位的结果，vga_h 是 h_addr 处以九后的结果。
vga_v 与 vga_h 拼起来恰好就是显存地址。

根据改地址从显存 ram 中读取出对应字符的 ascii 码，再到字符 rom 中取出对应的字模信息。 注意：字模点阵 mif 文件中每个字符占了 16x9 的大小，且字符逐行存储，故读取地址应是 8 位 ascii 码作为高 8 位，低 4 位是列地址模 16 即低四位

```
rom_font rf(.address({ascii, mod16}), .clock(clk), .q(font));
```

2.3.4 遇到的问题

上述计算过程中，只有一个除以 9 无法通过移位实现。
实验十一中代码量小、其他要处理的内容也少，可以直接除以 9，而本实验中由于其他模块内容较多，直接除以 9 处理器来不及算，显示屏上会出现奇怪的 bug。故开了一个模块用于使用 case 语句对除 9 模 9 地址的计算进行直接赋值。

```
module div_mod(h, v, map_h, map_v, mod9, mod16);
    input [9:0] h;
    input [9:0] v;

    output [4:0] map_v;
    output reg[6:0] map_h;
    output reg[3:0] mod9;
    output [3:0] mod16;

    assign map_v = v[8:4];
    assign mod16 = v[3:0];

    always @*
    case(h)
        10'd0: begin map_h=7'd0; mod9=4'd0; end
        10'd1: begin map_h=7'd0; mod9=4'd1; end
        10'd2: begin map_h=7'd0; mod9=4'd2; end
        10'd3: begin map_h=7'd0; mod9=4'd3; end
        10'd4: begin map_h=7'd0; mod9=4'd4; end
        10'd5: begin map_h=7'd0; mod9=4'd5; end
        10'd6: begin map_h=7'd0; mod9=4'd6; end
        10'd7: begin map_h=7'd0; mod9=4'd7; end
        10'd8: begin map_h=7'd0; mod9=4'd8; end
        10'd9: begin map_h=7'd1; mod9=4'd0; end
        10'd10: begin map_h=7'd1; mod9=4'd1; end
        10'd11: begin map_h=7'd1; mod9=4'd2; end
        10'd12: begin map_h=7'd1; mod9=4'd3; end
        10'd13: begin map_h=7'd1; mod9=4'd4; end
        10'd14: begin map_h=7'd1; mod9=4'd5; end
        10'd15: begin map_h=7'd1; mod9=4'd6; end
        10'd16: begin map_h=7'd1; mod9=4'd7; end
        10'd17: begin map_h=7'd1; mod9=4'd8; end
        10'd18: begin map_h=7'd2; mod9=4'd0; end
        10'd19: begin map_h=7'd2; mod9=4'd1; end
        10'd20: begin map_h=7'd2; mod9=4'd2; end
```

（有近千行，就不全部复制了）

三、实验启示

1. 团队合作一定要多交流，之前基础实验的一些好的模板和代码框架可以互相借鉴。

2. 时间与空间往往不能兼得，往往要通过反复的测试调整找到一个能够实现较高效率的平衡点。

3. FSM 永远滴神

4. 善于利用工具是个好习惯（比如弄个 java 对拍）