Duke ECE152 – Spring 2011 – Project Part 7: Program

200 Points. Must be submitted electronically by 10:00am on Weds, April 27 .

This is NOT an easy assignment. Start early! That way you have time to get help.

In this final part of the project, you will write a program in assembly to run on your processor. You will use your processor from Project 6 or a processor with further enhancements, download your design onto an FPGA prototyping board, and run the program that you write.

Project Part 7a: Programming Your Processor

You will write a program that takes as input a 128-bit hash of a secret "password" that is known to contain between 1-3 characters (inclusive), recovers the original string, and prints that string to the screen. The input characters are encoded in ASCII and are strictly alphanumeric (0-9, a-z, A-Z for a total of 62 possible characters). The hash is computed using a variant of the MD5 hashing algorithm. C code which illustrates the hashing algorithm is shown in Appendix A. A few example strings and their hashes are provided in Appendix B for testing. Appendix C contains notes on real-world applications of this problem and implications.

Remember that your task is not to simply perform the hash; your job is to recover the input given a hashed output. You must explicitly compute any hashes you use; in other words, it is not acceptable to place pre-computed values in a memory table. You will need to place the "magic value" constants shown in the code of Appendix A into memory, but those should be the only large lists of constants you require. A skeleton assembly file which has the requisite constants in assembler-friendly signed decimal format is provided at http://www.ee.duke.edu/~sorin/ece152/MD5_Skeleton.asm.

Project Part 7b: Demonstration

After writing (and hopefully simulating your program), you will download your programmed processor to one of the Altera DE2 FPGA prototyping boards and demonstrate the program. Do not wait until demo day to complete this final step; just because your design works in Quartus and your program simulates does not mean that there will not be hiccups getting it onto the board. If you desire to run your processor at a clock frequency other than 50 MHz, you may edit the "clk0_divide_by" and "clk0_multiply_by" parameters in the pll.vhd file.

Project Part 7c: Faster Is Better

Bonus points will be awarded to the three groups whose demos are the fastest (and correct!): 50 points for first place, 40 points for second place, and 30 points for third place. Groups may (but are not guaranteed to) earn up to 20 bonus points for implementing interesting features, even if their demo is not one of the fastest three. Be sure to tell us about these when you demo.

You are encouraged to boost the performance of your processor using any combination of the following three approaches. Remember that this is strictly optional; full credit is awarded for a correctly programmed unmodified processor.

- Optimize your code, particularly with respect to algorithms. Although we deviate from MD5 slightly, the core operations remain the same and thus many of the same optimizations can be applied. In addition to having efficient code in general, you could also make modifications which will lead to increased performance on your particular microarchitecture (eg, taking advantage of your knowledge regarding when stalls would occur and when bypassing may be possible).
- Extend the ISA. There are many unused instruction and ALU opcodes which you may use for any purpose you want. You will want to modify the assembler to recognize any opcodes that you add to generate appropriate binary output. You may only extend the ISA; you may not remove or modify any of the original functionality. Code written for the base Duke152-S11-32 ISA must still run correctly on your processor simply by changing memory initialization files.
- Enhance the microarchitecture. If you want to add branch prediction, more pipeline stages, a 2-wide pipeline, or any other modern microarchitectural feature, you may do so. Again, code written for the base ISA must still run correctly on your processor simply by changing memory initialization files.

Submitting This Assignment

To submit this assignment, create a packaged and compressed file named project7.zip or project7.tar.gz of the assembly code and all the files needed to implement your hardware design. If you modified your processor from Project 6 then include a readme file explaining the changes. If you modified the assembler then include the new executable, modified source code, and a readme file explaining the changes. Email your packaged and compressed file as an attachment along with all group members' names and NetIDs to DukeECE152Spring2011@gmail.com. You will also demonstrate your working processor and program on an FPGA board in lab during class at 10:20am on Wednesday, April 27.

Appendix A: C code to perform our hashing function

```
const unsigned int r[64] = {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};
const unsigned int k[64] = {0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee,
                            0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501,
                            0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be,
                            0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821,
                            0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa,
                            0xd62f105d, 0x2441453 , 0xd8a1e681, 0xe7d3fbc8,
                            0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed,
                            0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a,
                            0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c,
                            0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70,
                            0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x4881d05 ,
                            0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665,
                            0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039,
                            0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1,
                            0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1,
                            0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391};
const unsigned int h0 = 0x67452301;
const unsigned int h1 = 0xEFCDAB89;
const unsigned int h2 = 0x98BADCFE;
const unsigned int h3 = 0x10325476;

void transform() {
      unsigned int a = h3;
      unsigned int b = h2;
      unsigned int c = h1;
      unsigned int d = h0;
      unsigned int f,g,i,t;
      // msg[] is an array of unsigned ints
      // with length 16 where each of the first
      // fifteen entries contains one ASCII
      // character from the message (leaving the
      // upper three bytes of each entry as 0).
      // The final element is equal to 3 plus
      // the number of characters in the message.
      for(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;
            }

            t = d;
            d = c;
            c = b;
            b = (b + rotL(a + f + k[i] + msg[g], r[i]));
            a = t;
      }
```

```
        a += h3;
        b += h2;
        c += h1;
        d += h0;
        printf("%x%x%x%x\n",a,b,c,d);
}

unsigned int rotL(unsigned int in, unsigned int amt) {
        return (in << amt) | (in >> (32-amt));
}
```

Appendix B: Test strings and corresponding hashes

Hash(“”) = c43e6b706ce1fb1b28e60a18159ede6b

Hash(“123”) = c283d7efa049ec1338ef5d8d5cd02e9c

Hash(“abc”) = 3c371c4a413a93be08935f39e0ff1c89

Hash(“A1”) = 8d5c3bee0a5565a8d91b4dcd8ef4b496

Hash(“i4i”) = d5b78470e648cb8d3977ee43e5d02779

Hash(“C3P”) = 3d523f2b55239cdab09725457eaf2bb2

Appendix C: Background notes

The algorithm you are asked to implement is identical to the actual MD5 algorithm in most respects. Most of the key differences are in the representation of the input message (endianness, stop bit, etc.) rather than the algorithm itself; however, our adaptation of the algorithm is for a single 512-bit chunk, whereas MD5 has some state tracking to allow for multiple 512-bit chunks. Those interested in the particulars of the MD5 specification should look through Rivest’s original paper at http://tools.ietf.org/html/rfc1321.

One-way hashing functions in general serve many purposes in computing. Since the output is of fixed (and relatively short) length for an input which is generally of variable length, the function can compress at the cost of recoverability of the original data. This property is utilized in checksums and error correction. Another major application, which we’re focusing on, relies on the irreversibility of the hashing function as a form of encryption. It is not uncommon to store login credentials in a central database in encrypted form. When websites reassure you that they only store your password in an encrypted format, there is a strong possibility that it is being stored as a hash. When you attempt to login, the password you type in would then be hashed on the client machine (your PC), transmitted, and compared to the hash stored in the database. This is done to avoid ever transmitting the plaintext password over the air or storing it unencrypted in a central location.

However, problems can still arise if access to the central database is compromised, such as in the incident involving the Gawker sites network in December 2010. In that attack, about 1.3 million users' credentials, including password hashes, were compromised. Using publicly available tools and relatively little computational power (a single 8-core Xeon), 200,000 plaintext passwords were recovered from those hashes in under an hour. It was found that 99.45% of those passwords contained only alphanumeric characters, just as in our scenario. In a malicious attack, this low-hanging fruit would be the first target.