Advanced Computer Organization

Exam #2

Alex Salo

1. USB devices goes through the following states and stages when attached to the bus:
States:

1. **Attached**
2. **Powered**
3. **Default**
4. **Address:** address has been assigned by system/hub
5. **Configured:** device has been configured
6. **(Possibly) Suspended:** post was suspended
7. **(Possibly) Error**

Stages:

1. **Setup stage.** Here request is sent, which consists of three packets: setup token with the address and endpoint number, data packet and the handshake for acknowledging successful receipt (or indication of an error). If the function successfully receives the setup data (PID, CRC are correct) it reply with ACK; otherwise - ognores the data and doesn't send a handshake package.
2. **Data stage.** Here data is being transferred via one or multiple In or OUT transfers. When the host is ready to receive control data it issues an IN token; function after making sure token has no errors, reply with a DATA packet. When the host needs to send the device a control data packet, it issues an OUT token and then data packet with control data. If it contains errors - function ignores the packet; else function issues ACK.
3. **Status Stage.** Here the status of the overall request is being reported.

2.

CPU wants to read data via SCSI disk drive. A conversation starts with the BUS-FREE state. CPU tries to establish a conversation with the target and to perform a data transfer.

1. CPU must arbitrate for the bus (ARBITRATION phase): BSY is set to True. The data line D0-D7 is set to low. Device examines the data bus for low signals. It there is a low data line corresponding to a higher address than the CPU's address, CPU terminates. Otherwise goes to the selection phase:
2. CPU enters the SELECTION phase. CPU, as a winner of the arbitration phase, sets the SEL signal to True and selects the device. CPU sets the data line corresponding to the disk to low, sets ATN to True, I/o to false, BSY to false. Then disk sees that and sets BSY=True. Now CPU sees BSY==True, it sets SEL=false. Communication link is now established. Bus lines are released for data transfer, which is controlled by disk.
3. DATA TRANSFER PHASE:
    a. COMMAND: CPU sends the COMMAND “give me the data”
    b. DATA IN: bytes are transferred from disk to CPU (keeping BSY=true)
        i. disk places data on D0-D7 plus a parity bit on DPARITY
        ii. disk waits ,then asserts REQ
        iii. CPU sees REQ==true and strobes data off the D0-D7 and DPARITY lines.
        iv. CPU asserts ACK
        v. disk sees ACK and sets REQ=false
        vi. CPU sees REQ==false and sets ACK=false.
    c. MESSAGE IN: disk sends “Command complete” message to CPU
    d. BUS FREE: disk releases the bus, BSY=false.

For 16 bit SCSI the difference would be in the ARBITRATION phase. Since it has 16 channels it should handle priorities differently.

3. To compute the results I wrote a simple R script:

```
ent = function(x){
  return(x * log2(1/x))
}

sysent = function(prob) {
  sum = 0
  for (i in 1:length(prob))
      sum = sum + ent(prob[i])
  print(paste("Entropy:", sum, ", Info Per Symbol:", sum / length(prob),
sep=" "))
}

sysent(c(0.5, 0.5))
# [1] "Entropy: 1 , Info Per Symbol: 0.5"

sysent(c(0.25,0.25,0.25,0.25))
# [1] "Entropy: 2 , Info Per Symbol: 0.5"

sysent(c(0.7, 0.1, 0.1, 0.1))
# [1] "Entropy: 1.3568 , Info Per Symbol: 0.3392"

sysent(c(0.1, 0.2, 0.3, 0.4))
# [1] "Entropy: 1.8464 , Info Per Symbol: 0.4616"
```

System 1			System 2			System 3			System 4		
Msg	Prb	H(x)	Msg	Prb	H(x)	Msg	Prb	H(x)	Msg	Prb	H(x)
1	0.5	0.5	a	0.25	0.5	A	0.7	0.36	X	0.1	0.33
0	0.5	0.5	b	0.25	0.5	B	0.1	0.33	Y	0.2	0.46
			c	0.25	0.5	C	0.1	0.33	Z	0.3	0.52
			d	0.25	0.5	D	0.1	0.33	W	0.4	0.528
System Entropy			System Entropy			System Entropy			System Entropy		
1			2			1.3568			1.8464		

Information per symbol is tricky. It could be thought of as system entropy [https://en.wikibooks.org/wiki/Data_Coding_Theory/Information].
Alternatively, we can calculate it as average per symbol, then information per symbol is 0.5, 0.5, 0.34 and 0.46 for systems 1,2,3 and 4, respectively.
Another way to think of it is to measure the lowest entropy, then information per symbol is 0.5, 0.5, 0.33 and 0.33 for systems 1, 2, 3 and 4, respectively.

4. Booth multiplication (assuming 8 bit integers): 11110000=-16 (A) * 00001111=15 (X), 00010001=-15 (Y)
Expected: -240 (16 bit result)

Operations:

- 00 - do nothing
- 01 - add multiplicand to upper product's register
- 10 - subtract multiplicand from upper product's register
- 11 - do nothing

Steps:

```
0001000            // added -A
00001000 0         // shifted right
00000100 00        // shifted
00000010 000       // shifted
00000001 0000      // shifted
11110001 0000      // added A
11111000 10000     // shifted
11111100 010000    // shifted
11111110 0010000   // shifted
11111111 00010000  // shifted, got the result = -240
```

5.
For the floating addition/subtraction algorithm has to handle the exponents and mantissas. The outline is the following:

1. Normalize X and Y to scientific notation
2. Adjust Y to have the same exponent as X, thus making the mantissa less
3. Add two mantissas (*of X and Y), keep the exponent

The difference is also in the way the floating point addition produces bias; how it is using the two guard bits and one sticky bit, which are carried beyond the precision of the operands.

Overall, floating point addition subtraction has a lot of hazards and requires careful implementation. For example, the problem occurs when two very close numbers are subtracted from one another.

6. Non-restoring integer division
Input:

N - dividend
D - divisor

Output:

Q - quotient
R - remainder

Consider Dividend = quotient * divisor + remainder ⇔ N = Q*D +R
Restoring Division:

1. Put N in register A, and Q in register B, zero in register P and do n divide steps (n is the length of Q):
    a. shift register pair (P, A) one bit left
    b. substract the contents of B from P, put the result back in P
    c. if P >=0, A[i] = 1
       else   A[i] = 0 and P = P + D // restore the value of P

After n cycles, A will have the Q, and P will have the remainder.

On contrast, non-restoring division skips the restoring step and instead deals with negative residuals:
For n cycles:

A. shift pair (P, A) one bit left
B. if P is negative - and add B to P
   if P is positive - subtract B from P
C. if P is negative set A[i] to 0
   otherwise set A[i] to 1

After the loop is finished, the Q is in A.
To compute remainder: If P if negative - add B to P. Now P is remainder.

Thus restoring division has one extra step each cycle - restoration, while non-restoring has to deal with the sign.

7.
To detect single bit error with Hamming code we have to satisfy:
m + k + 1 <= 2 ^ k
8 <= 2 ^ k - k - 1
k = 3 - > 8 - 3 - 1 < 8 - bad; **k = 4** -> 16 - 4 - 1 = 11 - fits!

Thus Hamming code would be 8 + 4 = **12 bit long**.

01101001 -> ??0?110?1001:
p1 = ?01010 - even = 0
p2 = ?01000 - odd = 1
p4 = ?1101 - odd = 1
p8 = ?1001 - even = 0
->codeWord **01**0**1**110**0**1001

00010001 -> ??0?001?0001
p1 = **?**?**0**?**0**0**1**?**0**0**0**1 - odd = 1
p2 = ?**?0**?0**01**?0**00**1 - odd = 1
p3 = ??0**?001**?000**1** - even = 0
p4 = ??0?001**?0001** - odd = 1
->codeWord **11**0**0**001**1**0001

11111010 -> ??1?111?1010
p1 = **?**?**1**?**1**1**1**?**1**0**1**0 - odd = 1
p2 = ?**?1**?1**11**?1**01**0 - even = 0
p3 = ??1**?111**?101**0** - odd = 1
p4 = ??1?111**?1010** - even = 0
->codeWord **10**1**1**111**0**1010

10101001 -> ??1?010?1001
p1 = ??**1**?**0**1**0**?**1**0**0**1 - even = 0
p2 = ?**?1**?0**10**?1**00**1 - even = 0
p3 = ??1**?010**?100**1** - even = 0
p4 = ??1?010**?1001** - even = 0
->codeWord **00**1**0**010**0**1001

01010011 -> ??0?101?0011
p1 = ??**0**?**1**0**1**?**0**0**1**1 - odd = 1
p2 = ?**?0**?1**01**?0**01**1 - even = 0
p3 = ??0**?101**?001**1** - odd = 1
p4 = ??0?101**?0011** - even = 0
->codeWord **10**0**1**101**0**0011
To check the correctness, let us use the following java client:

```java
package information_theory;
import java.util.Arrays;

/**
 * @author Aleksandr Salo
 * Client to compute HammingCode
 * For class Advanced Computer Organization
 */
public class HammingCode {

        private static int countBits(int x) {
        return x == 0 ? 0 : countBits(x & (x - 1)) + 1;
        }

        public static String hammingCode(String input) {
        int M = input.length();

        // 1. read bits
        char[] inputchars = input.toCharArray();
        int[] bits = new int[M];
        for (int i = 0; i < M; i++)
        bits[i] = Character.getNumericValue(inputchars[i]);
        System.out.println("Input: " + Arrays.toString(bits));

        // 2. figure out redundancy bits length k
        int k = 2;
        while (Math.pow(2, k) - k - 1 < M)
        k++;
        System.out.println("Choose k = "+ k +
                " to detecet and correct single bit error");

        // 3. prepare resulting Hamming code
        int[] hamcode = new int[M+k];
        int m = 0;
        for (int i = 0; i < M+k; i++) //start with 2
        if (countBits(i + 1) == 1) // if power of 2
                hamcode[i] = 0; // init parity bits with 0
        else
                hamcode[i] = bits[m++];
        System.out.println("With Empty parity: " +
Arrays.toString(hamcode));

        // 4. compute parity bits
        for (int parityBit = 0; parityBit < k; parityBit++) {
        int skip = 1 << parityBit;
        int cur = skip - 1;
        int parity = 0;
        while (cur < M + k) {
                for (int i = cur; i < cur + skip && i < M + k;
i++)
                parity += hamcode[i];
                cur += 2 * skip; // skip two
        }
        //System.out.println(parity);
        hamcode[parityBit] = parity % 2; // set bit if odd
        }
        System.out.println("Hamming Code Word: " +
Arrays.toString(hamcode));

        // 5. make resulting string
        String s = "";
        for (int i = 0; i < hamcode.length; i++)
        s += String.valueOf(hamcode[i]);
        return s;
        }
```

```java
// test client
        public static void main(String[] args) {
        String[] input = new String[] {
                "01101001",
                "00010001",
                "11111010",
                "10101001",
                "01010011"
        };

        // Compute Ham Codes
        String[][] hamCodes = new String[input.length][2];
        for (int i = 0; i < input.length; i++) {
        hamCodes[i][0] = input[i];
        hamCodes[i][1] = hammingCode(input[i]);
        }

        // print Message, CodeWord
        for (int i = 0; i < input.length; i++)
        System.out.println(Arrays.toString(hamCodes[i]));
        }
}
```

Result:

```
[01101001, 010111001001]
[00010001, 110000110001]
[11111010, 101111101010]
[10101001, 001001001001]
[01010011, 100110100011]
```

8.
The largest block of data (with detectable double-bit errors) is 2^19-1 = **524287.**

CRC code is 19 bit long (n = 19), since polynomial is related to a mersenne prime and hence irreducible - doesn’t divide any polynomial of lower degree.

9. Polynomial $X^8 + X^6 + X^5 + X + 1$ looks like 101100011 in binary. Here are CRC codes in hex, computed by the program presented below. Please note, that program was written to simulate the process of how I would do the derivations by hand, thus not claiming to be very efficient.

Integer in HEX	CRC Code
376034ae	f
1a659bdf	de
a284c439	74
b084006a	51
a1daee92	37

```java
// Client
    public static void main(String[] args) {
        int[] Poly = new int[]{8, 6, 5, 1};
        int[][] hexNums = new int[][]{
             {0x376034ae, 0},
             {0x1a659bdf, 0},
             {0xa284c439, 0},
             {0xb084006a, 0},
             {0xa1daee92, 0},
        };

        for (int i = 0; i < hexNums.length; i++) {
             hexNums[i][1] = computeCRC(hexNums[i][0], Poly, false);
             System.out.println(Integer.toHexString(hexNums[i][0])
                     + ", CRC code: " + Integer.toHexString(hexNums[i][1]));
        }
    }
    // To print 32-bit integers in consistent format
    private static void pf(int num, String prefix) {
        System.out.println(prefix + String.format("%32s",
             Integer.toBinaryString(num)).replace(' ', '0'));
    }

    public static int computeCRC(int input, int[] codePowers, boolean verbose) {
        int n = codePowers.length;

        // 1. get the binary representation of the poly's powers
        int poly = 1;
        for (int i = 0; i < n; i++)
             poly += 2 << codePowers[i] - 1;
        int tmpCode = poly;

        // 2. align poly with the MSB
        int leadingZeros = Integer.numberOfLeadingZeros(input);
        poly = poly << (31 - codePowers[0] - leadingZeros);

        int moveLater = 0;
        if (verbose) pf(input, "msg : ");

        // 3. Divide until input is less than a divider
        while (input > 2 << codePowers[0] || input < 0) {
             // 3.1 XOR
             input = input ^ poly;

             // 3.2 print
             if (verbose) {
             pf(poly, "poly: ");
             System.out.println("--------------------------------");
             pf(input, "msg : ");
             }

             // 3.3 align poly with MSB
             int newLeadZ = Integer.numberOfLeadingZeros(input);
             moveLater = newLeadZ - leadingZeros;
             if (input > 2 << codePowers[0]) {
             poly = poly >> moveLater;
             if (poly < 0)
                    poly = tmpCode << (31 - codePowers[0] - moveLater);
             leadingZeros = newLeadZ;
             }
        }

        // 4. Can't divide anymore, finish by finding remainder

        // 4.1 Move poly and input by the length of the poly
        input = input << codePowers[0]; // make some space for a quoitet
        poly = (poly << codePowers[0]) >> moveLater;
        leadingZeros = Integer.numberOfLeadingZeros(input);
        if (verbose) {
            System.out.println("\nAppending Space for finding remainder...\n");
            pf(input, "msg : ");
        }

        // 4.2 Divide the same way
        while (input > 1 << codePowers[0]) {
            input = input ^ poly;
            if (verbose) {
            pf(poly, "poly: ");
            System.out.println("--------------------------------");
            pf(input, "msg : ");
            }
            int newLeadZ = Integer.numberOfLeadingZeros(input);
            poly = poly >> (newLeadZ - leadingZeros);
            leadingZeros = newLeadZ;
        }

        // 5. Collect the remainder - CRC code
        return input;
      }
```

Which produces **de** the following (for input = **1a659bdf**):

```
msg : 00011010011001011001101111011111
poly: 00010110001100000000000000000000
---------------------------------
msg : 00001100010101011001101111011111
poly: 00001011000110000000000000000000
---------------------------------
msg : 00000111010011011001101111011111
poly: 00000101100011000000000000000000
---------------------------------
msg : 00000010110000011001101111011111
poly: 00000010110001100000000000000000
---------------------------------
msg : 00000000000001111001101111011111
poly: 00000000000001011000110000000000
---------------------------------
msg : 00000000000000100001011111011111
poly: 00000000000000101100011000000000
---------------------------------
msg : 00000000000000001101000111011111
poly: 00000000000000001011000110000000
---------------------------------
```

```
msg : 00000000000000000110000001011111
poly: 00000000000000000101100011000000
---------------------------------
msg : 00000000000000000011100010011111
poly: 00000000000000000010110001100000
---------------------------------
msg : 00000000000000000001010011111111
poly: 00000000000000000001011000110000
---------------------------------
msg : 00000000000000000000001011001111
poly: 00000000000000000000001011000110
---------------------------------
msg : 00000000000000000000000000001001

Appending Space for remainder...

msg : 00000000000000000000100100000000
poly: 00000000000000000000101100011000
---------------------------------
msg : 00000000000000000000001000011000
poly: 00000000000000000000001011000110
---------------------------------
msg : 00000000000000000000000011011110
                        d   e
```

10.

- **SISD** - Single Instruction stream, Single Data stream - standard sequential computer exploiting instruction-level parallelism. Example: superscalar processor (i.e. Alpha 21164)
- **SIMD** - Single Instruction stream, Multiple Data streams - same instruction is executed by multiple processors using different data streams exploiting data-level parallelism. Example: ILLIAC IV, various vector architecture, GPU.
- **MIMD** - Multiple Instruction stream, Multiple Data streams - each processor fetches its own instructions and operates its own data - targets task-level parallelism. MIMD is more flexible than SIMD but inherently more expensive too. Often used in clusters and warehouse-scale computers that exploit request-level parallelism, where many independent tasks can proceed in parallel with little need for communication or synchronization. Example: Intel Xeon Phi

11.
Cache coherence algorithm is used to track the state of any sharing of a data block.
Snoopy cache coherence uses the following approach: every cache that has a copy of data from a block of physical memory also has a copy of the sharing status of the block, while no centralized state is kept. The **caches are all accessible via some broadcast medium** (a bus or switch), and all cache controllers monitor (snoop) on the medium to figure out whether or not they have a copy of a block that is requested on a bus or switch access.
This approach is useful in multiprocessor systems which are all attached to single shared memory.
To maintain coherence snoopy protocols often use write invalidation protocol - ensuring that a processor has an exclusive access to data item before it writes that item. This exclusive access ensures that no other readable or writable copy of an item exist when the write occurs: all other cached copies of the items we invalidated.
A snooping coherence protocol is usually implemented by incorporating a finite state controller in each node.