EECS 281: Homework #3 Due: Tuesday, 28.09.2004

0. Practice the wakerly problems (see website soln.) and do not hand in: 2.1(a,b,e), 2.7a, 2.9a, 2.10a, 2.12a, 2.37.

1. Using standard C++ precision and data types, convert the following into two's complement big-endian binary and if not, then show why not?:

(1a) unsigned char x = 'A';	01000001	(1e) unsigned char x = 0255;	10101101
(1b) unsigned char x = 0x255;	Doesn't fit(10-bits)	unsigned char x = 255;	11111111
(1c) signed char x = 255;	Doesn't fit(9-bits)	(1e) unsigned char x = 0128;	illegal octal digit 8
(1d) unsigned char x = 128;	10000000	unsigned char x = 0xfa;	11111010
unsigned char x = 35;	00100011	(1d) unsigned char x = -35;	**unsigned** cannot be negative
signed char x = 127;	01111111	(1c) signed char x = 128;	illegal range: -128 to 127
signed char x = -128;	10000000	signed char x = -0x2;	11111110
(1e) signed char x = -07;	11111001	signed short x = -2;	1111111111111110
(1f) signed short x = 35;	0000000000100011	signed short x = -35;	1111111111011101
(1g) signed short x = 'a';	0000000001100001	(1h) signed short x = -'a';	1111111110011111

Notes:

(1a) ASCII 'A' is 65 or 0x41 or b"01000001" (2'complement is a "representation" of a number. Only negative numbers need the bit's complemented and add one).

(1b) 0x255 is b"0010 0101 0101" is at least a 10 bit number and "unsigned char x" is an 8 bit number. The C compiler gives the following message, "warning: large integer implicitly truncated to unsigned type."

(1c) For an 8-bit signed character, the range is -128 to 127 (i.e. -2^7 to $2^7 - 1$). Although, 255 can fit as an 8-bit "unsigned" number b"**1**1111111", we see that the most significant bit is '**1**' and that would imply it is a **negative** number, which it is NOT! In order to make it a positive number, we must add a sign bit of '0'. Now 255 can be represented as a signed 9-bit number of b"011111111". The C compiler should give the following message, "warning: overflow in implicit constant conversion." Some compilers may not catch this.

(1d) For an 8-bit unsigned character, the range is 0 to 255 (i.e. 0 to $2^8 - 1$). So 128 is acceptable.

(1e) Any number with a leading zero is octal. And the only legal digits are 0,1,2,3,4,5,6 and 7.

(1f) signed short is a 16-bit number, whose range is -32,768 to 32,767 (i.e. -2^{15} to $2^{15} - 1$).

(1g) ASCII 'a' is 'A'+0x20 = 0x41+0x20 = (byte) 0x61 = (short) 0x0061;

(1h) ASCII 'a' is (short) 0x0061; The negative of 0x0061 $\Rightarrow$ -0x0061 $\Rightarrow$ ($\sim$0x0061)+1 $\Rightarrow$ ($\sim$0000000001100001)+1 $\Rightarrow$ (1111111110011110)+1 $\Rightarrow$ 1111111110011111.

2. Assume VHDL data types convert the following into two's complement big-endian binary:

(2a) signal x: std_logic_vector(4 downto 0):= b"10111";	10111
(2b) signal x: std_logic_vector(0 to 4):= b"10111";	11101
signal x: std_logic_vector(7 downto 0):= o"45";	00100101
signal x: std_logic_vector(0 to 7):= x"ab";	11010101

Notes:

(2a) The bit string implies NO endian information! Assign the leftmost digit of the 5-bit string to the leftmost number declared in the vector. $x(4) \Leftarrow 1$, $\mathbf{x(3)} \Leftarrow \mathbf{0}$, $x(2) \Leftarrow 1$, $x(1) \Leftarrow 1$, $x(0) \Leftarrow 1$. The most significant bit is always the highest digit declared (i.e. x(4)). So the final number as big endian would be x(4)x(3)x(2)x(1)x(0)" or b"10111". Obviously the same.

(2b) Assign the leftmost digit of the 5-bit string to the leftmost number declared in the vector. $x(0) \Leftarrow 1$, $\mathbf{x(1)} \Leftarrow \mathbf{0}$, $x(2) \Leftarrow 1$, $x(3) \Leftarrow 1$, $x(4) \Leftarrow 1$. The most significant bit is always the highest digit declared (i.e. x(4)). So the final number as big endian would be x(4)x(3)x(2)x(1)x(0)" or b"11101".

3. Using C++/C#/Java operator precedence, add the correct parenthesis (signed int a, b, ..., w, x, y, z;):

a = (((w & x) & y) & z);	a = ((w | x) | (y & z));
a = (((~ x) & y) & (~ z));	a = (x | (y & z));
a = (x | (y ^ (w & z)));	a &= ((x & y) & z);
a = ((x * y) + z);	a = (z + (y * z));
a = ((z + (((y * z) % w) / v)) - c);	a = ((x & y) | z);
a = (z | (y & z));	a = ((~ z) ^ (y & z));
a += ((((b + (c >> d)) & e) ^ f) | ((((~ g) % h) * i) - j));	

4. Using VHDL operator precedence, add the correct parenthesis:
a <= ((((b + (c SRL d)) AND e) XOR f) OR ((((NOT g) MOD h) * i) - j));

5. Using C++ convert the following into two's complement big-endian binary:
where unsigned char u, a=0x85, b=0x96, c=02; signed char s, w=0x80, x=0x96, y=0, z=0x15;
For addition and subtraction indicate if overflow and/or carry has occurred.
Show work on a seperate piece of paper.

(5a)(5g) u = ~ a ;	01111010	(5b) u = - a ;	01111011
(5g) u = a & b;	10000100	(5c) u = a | b & c;	10000111
(5g) u = a ^ b;	00010011	(5d) u = a + b;	00011011(unsigned overflow due to C_{out})
(5g) u = a ^ 'A';	11000100	u = a + 'A';	11000110(No unsigned overflow)
(5h) u = a - b;	11101111(No signed overflow)	(5o) u = a * b;	Overflow (duplicate)
(5i) u = a << 2;	00010100 (logical shift)	(5j) u = a >> c;	00100001 (logical shift)
(5o) u = a * b;	Overflow	u = a % b;	10000101
u = a / b;	00000000	(5b) u = - a;	01111011 (duplicate)
(5f) s = - w ;	10000000(signed Overflow)	(5n) s = - z ^ ~ x;	10000010
(5g) s = w & x;	10000000	(5g) s = w ^ x;	00010110
(5e) s = w + x;	00010110(signed Overflow)	(5m) s = w - x;	11101010(No signed overflow)
(5l) s = x << 2;	11011000 (arithmetic shift)	(5k) s = x >> 2;	11100101(arithmetic shift)

Notes: TIP: Always double check your work by doing it in decimal and seeing if it the result fits in the numeric range of the data type. So, the number 0x85 is unsigned 10000101 ⇒ 128+4+1 ⇒ 133. The number 0x80 is unsigned 10000000 ⇒ 128, and signed -128. The number 0x96 is unsigned 10010110 ⇒ 128+16+4+2 ⇒ 150; and signed -128+16+4+2 ⇒ -106

(5a) 1's complement of variable a (NOT 2's complement or negate of variable a). 1's complement is the $d_i = \sim d_i$; or flip each bit (i.e. 0 to 1 and 1 to 0). See also Note (5g)

(5b) 2's complement of variable a. Doesn't matter if the variable is signed or unsigned just do it: The negative of 0x85 ⇒ -0x85 ⇒ (∼0x85)+1 ⇒ (∼10000101)+1 ⇒ (01111010)+1 ⇒ 01111011.

(5c) First determine precedence! u= a | (b & c); Note that the AND has higher precedence than the OR. So, (b & c) ⇒ (0x96 & 02) ⇒ (10010110 & 00000010) ⇒ (00000010). Now, a | (b & c) ⇒ a | (00000010) ⇒ 0x85 | (00000010) ⇒ 10000101 | (00000010) ⇒ 10000111.

(5d) "Signed" number overflow (i.e. out of the number's legal range) occurs when the Carry in to the sign bit differs with the Carry out. So, **Signed Overflow** = C_{out} **XOR** C_{in}.

A "signed" number result is BAD (i.e. out of range) when an signed Overflow occurs. Warning, overflow says nothing about "unsigned" number results.

An "unsigned" number result is BAD (i.e. out of range) when the Carry Out of the most significant bit of a number occurs (unsigned overflow). Warning, just having a Carry out says nothing about signed numbers! So, **Unsigned Overflow** = C_{out}.

Signed or unsigned overflows can never happen to bitwise operations (AND, OR, NOT, XOR) because they never produce a Carry out or a Carry in!

Signed or Unsigned numbers do NOT play any role in ADDITION. Just add them blindly. Unfortunately, C programs cannot catch overflows at execution time. You must check after each add to catch it!

So, "u=a+b" is u=(unsigned char)0x85 + (unsigned char)0x96; ⇒ u=10000101 + 10010110; ⇒ $C_{out} = 1$ and u=00011101; Since these numbers are unsigned, we only care about the Carry out.

So, let's just do it again in decimal: "u=a+b" ⇒ "u=0x85+0x96" ⇒ "133+150" ⇒ 283 which is greater than the unsigned range 0 to 255! Also, 283 requires 9 bits or 100011101. and if we seperate the bit 8 with bits 7-0, we get 1 and 00011101. This is the same as doing C_{out} and 8 bit addition result.

(5e) So, "s=w+x" ⇒ "s=(signed)0x80 + (signed)0x96" ⇒ "s=10000000 + 10010110" ⇒ $C_{out} = 1$, $C_{in} = 0$, s="00000110" and signed Overflow=C_{out} XOR C_{in} = 1 XOR 0=1.

So, let's just do it again in decimal: "s=w+x" ⇒ "s=(signed)0x80 + (signed)0x96" ⇒ "s=(signed)-128 + (signed)-128+16+4+2" (quick conversion by adding power's of 2) ⇒ "s=-128 + -106" ⇒ "s=-234" which is out of range of -128 to 127. Note, that "s=-234" is a 9 bit signed number b"**1**00010110".

(5f) Again, the computer will execute it, it's up to the programmer to know when it overflows. So first, just do it: "s = - w" ⇒ "s= -0x80" ⇒ "s= (∼ 0x80)+1" ⇒ "s= (∼ 10000000)+1" ⇒ "s= (01111111)+1" ⇒ $C_{out} = 0$, $C_{in} = 1$, s=10000000, signed overflow=C_{out} XOR C_{in} = 0 XOR 1 = 1.

(5g)Signed or unsigned overflows can never happen to bitwise operations (AND, OR, XOR) because they never produce a Carry out or a Carry in! Signed or Unsigned numbers do NOT play any role in bitwise operations (i.e. AND, &, OR, |, NOT, ∼, XOR, ^). Just ignore they are signed or unsigned.

(5h)So, "u=a-b" ⇒ "u=(unsigned char)0x85 - (unsigned char)0x96; ⇒ u=10000101 - 10010110; ⇒ u=10000101 + (-10010110); ⇒ u=10000101 + ((∼10010110)+1) ⇒ u=10000101 + ((01101001)+1) ⇒ u=10000101 + (01101010) ⇒ u=11101111, $C_out = 0$, no unsigned overflow.

So, let's just do it again in decimal: "u=a-b" ⇒ "u=(unsigned char)0x85 - (unsigned char)0x96; ⇒ "u=(unsigned char)133 - (unsigned char)150" ⇒ "u=-17" (don't worry that it's signed here) ⇒ "u=-(00010001)" ⇒ "u=(∼00010001)+1" ⇒ "u=(11101110)+1" ⇒ "u=11101111" (it's the same).

(5i) If the number to be shifted is "unsigned" then it is called a "logical shift" and we DO NOT care about the sign bit. And if we need and extra bit to shift in us a '0' bit. Thus, for an 8-bit number, "$d_7d_6d_5d_4d_3d_2d_1d_0$" becomes "lost ⇐ d_7 ⇐ d_6 ⇐ d_5 ⇐ d_4 ⇐ d_3 ⇐ d_2 ⇐ d_1 ⇐ d_0 ⇐ 0".

So, "u=a << 2" ⇒ "u=(unsigned char)0x85 << 2" ⇒ "u=10000101 << 2" ⇒ "u=1000010100" ⇒ "u=00010100" (drop the leftmost 2 bits). Note, that we shifted out '1' bits and the resulting number is in decimal 16+4 = 20. This means that there is a LOSS of precision and the resulting number is wrong! But the computer doesn't care, that's the programmer's job to know that and use the CORRECT integer size.

So, let's just do it again in decimal: "u=a << 2" ⇒ "u=(unsigned char)133 << 2" ⇒ "u= 133 * 2^2" ⇒ "u= 133 * 4" ⇒ "u=532" (this is a 10-bit number: 1000010100=512+16+4=512+20). Now, if we chop off the upper bits (i.e. 512), we will get the same result of an 8-bit shift of 20!

(5j) So, "u=a >> c" ⇒ "u=(unsigned char)0x85 >> 2" (logical shift) ⇒ "u=10000101 >> 2" ⇒ "u=00100001" (drop rightmost bits). Note this number in decimal is 33.

So, let's just do it again in decimal: "u=a >> c" ⇒ "u=(unsigned char)0x85 >> 2" (logical shift) ⇒ "u=(unsigned char)133 / 2^2" ⇒ "u=133 / 4" ⇒ "u=33.25" (drop the fraction) ⇒ "u=33". Note it's the same.

(5k) If the number to be shifted is "signed" then it is called a "arithmetic shift" and we DO care about the sign bit. The sign bit does not participate in the shift. That means that we never shift the sign bit, that always stays unchanged. As we shift right arithmetic, the sign bit supplies a copy of itself: "$d_7d_6d_5d_4d_3d_2d_1d_0$" becomes "$d_7 \Rightarrow d_6d_5d_4d_3d_2d_1d_0$". Whereas a "shift right logical" becomes "$0 \Rightarrow d_7d_6d_5d_4d_3d_2d_1d_0$".

So, "s=x >> 2" ⇒ "s=(signed char)0x96 >> 2" ⇒ "s=**1**0010110 >> 2" (sign bit in bold) ⇒ "s=**111**00101" (last two bits lost). Note: The decimal of 11100101 is -128+64+32+4+1 which is -27

So, let's just do it again in decimal: "s=x >> 2" ⇒ "s=**1**0010110 >> 2" ⇒ "s=**-128**+16+4+2 >> 2" ⇒ "s=(signed char)-106 / 2^2" ⇒ "s=-106 / 4" ⇒ "s=-26.5" (if negative then round up the fraction) ⇒ "s=-27".

(5l) The C compiler on a 32-bit machine will give the incorrect answer of b"01011000". This result is due the fact that C only does arithmetic shift for 32-bits and then storing it in a character will truncate the number. The correct answer is b"11011000" because the sign bit must never be altered in a shift arithmetic. If the compiler was designed for a true 8-bit machine, then it will give the correct answer that I have. This is why JAVA Language defines not only the "data types precisely" but also "demands" the machine architecture implement those data types. Thus, JAVA's philosophy is let the **software designer's tell the machine designers what to do**. C's philosopy is different, **let the machine designers tell you what the data types are** and then C defines an integer which is natural to machine.

(5m) A quick decimal check will tell you if you will overflow. "s=w - x" ⇒ "s=-128 - (-106)" ⇒ "s=-128 + 106" ⇒ "s=-22" (obviously within the range of a signed 8-bit number: -128 to 127). And what is -22 ⇒ -(16+4+2) ⇒ -(00010110) ⇒ (∼00010110)+1 ⇒ (11101001)+1 ⇒ (11101010).

(5n) When mixing bitwise and arithmetic operations, overflow has no real meaning. So, just number crunch away! "s= -z ^ ∼ x" ⇒ "s= ((∼ z)+1) ^ ∼ x" ⇒ "s= ((∼ 0x15)+1) ^ ∼ 0x96" ⇒ "s= ((∼ 00010101)+1) ^ ∼ 10010110" ⇒ "s= ((11101010)+1) ^ 01101001" ⇒ "s= 11101011 ^ 01101001" ⇒ "s= 10000010"

(5o) A quick decimal check will tell you if you will overflow. "u=a * b" ⇒ "u= (unsigned char)0x85 * (unsigned char)0x96" ⇒ "u= 133 * 150" ⇒ "u= 19950" which is greater than the range of 0 to 255.