Sergio Garcia Tapia
Digital Design and Computer Architecture: RISC-V, by Sarah and David Harris
Chapter 5: Digital Building Blocks
March 7, 2024

Exercise 5.1. What is the delay for the following types of 64-bit adders? Assume that each two-input gate delay is 150 ps and that a full adder delay is 450 ps.

(a) a ripple-carry adder

(b) a carry-lookahead adder with 4-bit blocks

(c) a prefix adder

Solution:

(a) The delay of a ripple-carry adder is governed by the equation

$$t_{\text{ripple}} = Nt_{FA}$$

where N is the number of full adders chained together, and t_{FA} is the delay of a full adder. Since a full adder is a two-input gate, the assumption is that $t_{FA} = 150$ ps. Moreover, since the adder deals with 64-bit, we have $N = 64$. Therefore the delay is:

$$t_{\text{ripple}} = 64 \cdot 450 \text{ ps } = 28800 \text{ ps } = 288 \text{ ns.}$$

(b) The delay of a carry-lookahead adder with k-bit blocks is governed by the equation

$$t_{CLA} = t_{pg} + t_{pg\_\text{block}} + \left(\frac{N}{k} - 1\right) t_{\text{AND\_OR}} + kt_{FA},$$

where t_{pg} is the delay of the column propagate and generates (which is a single AND or OR gate) to generate P_i and G_i, t_{pg_block} is the delay to find the block propagate and generate signals $P_{i:j}$ and $G_{i:j}$ for a k-bit block, and $t_{\text{AND\_OR}}$ is the delay from C_{in} to C_{out} through the final AND/OR logic of the k-bit CLA block. Hence, the critical path is as follows

(i) Compute all column generate and propagate signals P_i and G_i in parallel for all blocks; since they all go through AND or OR, this costs $t_{pg} = 150$ ps delay.

(ii) Compute all block generate and propagate signals, $P_{i:j}$ and $G_{i:j}$, where the critical path goes into computing $G_{i:j}$. For 4-bit blocks, this consists of 6 AND/OR gates, so $t_{pg\_\text{block}} = 6 \cdot 150$ ps $= 900$ ps.

(iii) Compute C_{out} using $G_{i:j}$ and the already-computed result of $C_{in}P_{i:j}$. This computes C_{out}, which is C_3, and then it goes through the rest of the blocks until it reaches the last one. The total cost is thus $\left(\frac{64}{4} - 1\right) t_{\text{AND\_OR}} = 15 \cdot 150$ ps $=$ 2250 ps .

(iv) The last on the critical path is the cost of the ripple-carry adder that consists of four full adders, each of which has three 2-input gates in its critical path. Therefore, $kt_{FA} = 4 \cdot (3 \cdot 150 \text{ ps }) = 1800$ ps. We finally have

$$t_{CLA} = 150 \text{ ps} + 900 \text{ ps} + 2250 \text{ ps } + 1800 \text{ ps} = 5100 \text{ ps} = 5.1 \text{ ns}.$$

(c) The equation governing the delay of a prefix adder is

$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_\text{prefix}}) + t_{\text{XOR}}$$

where t_{pg} is the precomputation delay of all P_i and G_i (which happens in parallel and costs a single 2-input gate), t_{pg_prefix} is the delay of a prefix cell, and t_{XOR} is the delay due to the final XOR gate to compute S_i. Note that the critical path delay for a single prefix block is two 2-input gates to compute $G_{i:j}$. Hence, the total delay is

$$t_{PA} = 150 \text{ ps} + \log_2 64 \cdot (300 \text{ ps}) + 150 \text{ ps} = 2100 \text{ ps} = 2.1 \text{ ns}.$$

Exercise 5.3. Explain why a designer might choose to use a ripple-carry adder instead of a carry look-ahead adder.

Solution: A designer may want a less complex circuit, which they can get by using a ripple-carry adder. Related, they may also be looking to use less hardware and/or power. Their specifications may not require the speed improvements that they would get by using a carry-lookahead adder.

Solution:

Exercise 5.5. The prefix network shown in Figure 1 uses black cells to compute all of the prefixes.

Some of the block propagate signals are not actually necessary. Design a "gray cell" that receives G and P signals for bits $i{:}k$ and $k{-}1{:}j$ but produces only $G_{i:j}$, not $P_{i:j}$. Redraw the prefix network, replacing black cells with gray cells wherever possible.

Solution: See Figure 2. To understand it, note that in the second level, all cells depend on cells from on a $P_{i:j}$ value from the first level, so all blocks must be black. In the third level, $G_{5:-1}$ and $G_{6:-1}$ both depend on $P_{2:-1}$, so that corresponding cell must be black. On the other hand, none of the cells in the third level depend on $P_{1:-1}$. In the fourth level, all cells, which comptue $G_{7:-1}, \ldots, G_{14:-1}$, depend on $P_{6:-1}$, so the corresponding block must be black.

Exercise 5.8. Design the following comparators for 32-bit unsigned numbers. Sketch the schematics.

(a) not equal

(b) greater than or equal to

(c) less than

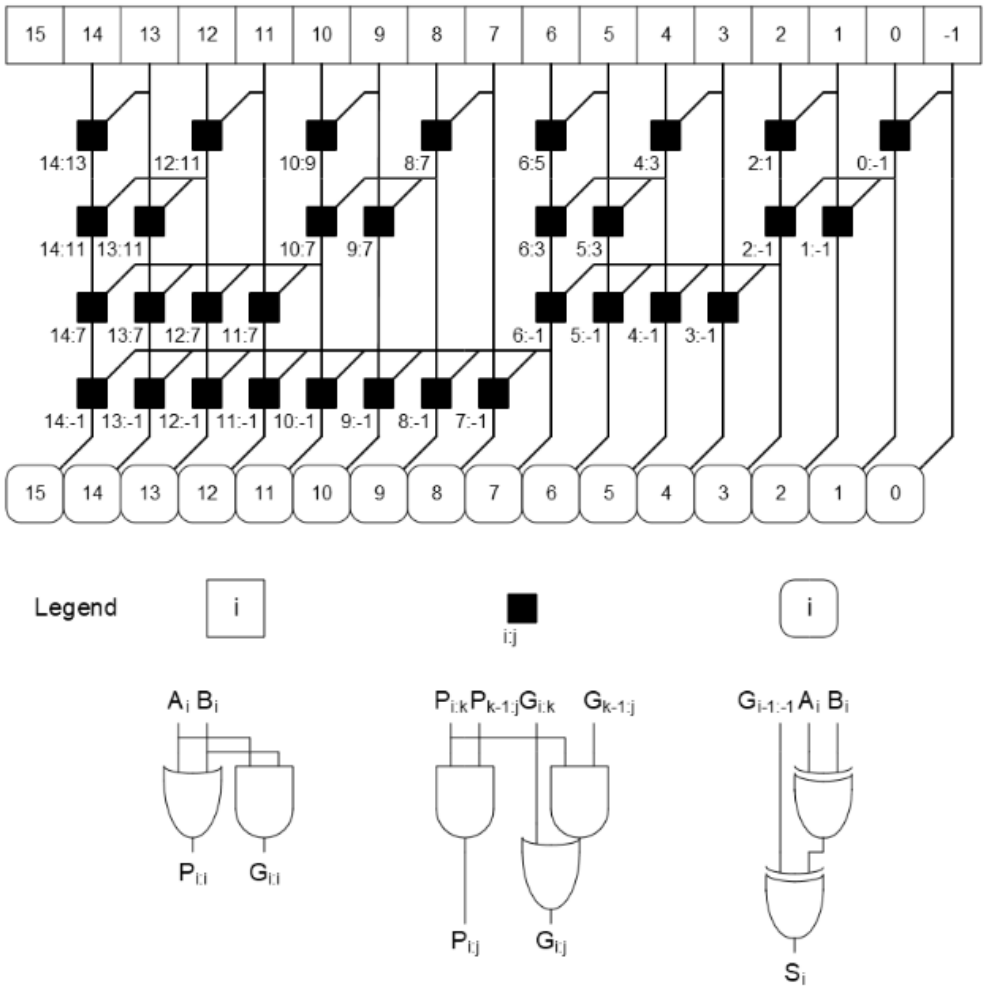

Figure 1: 16-bit prefix adder

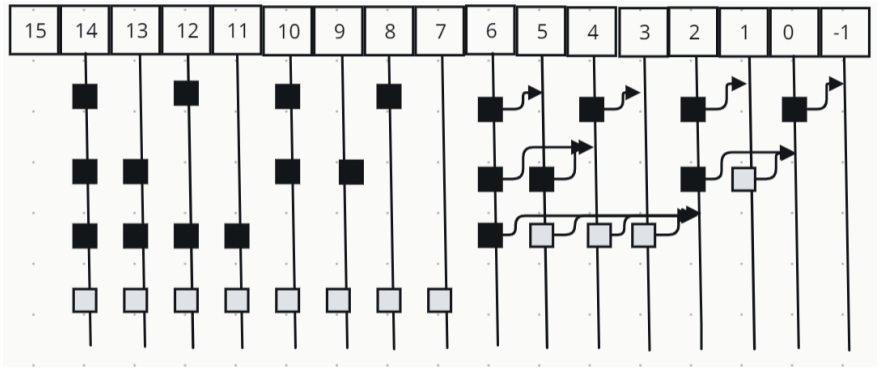

Figure 2: Exercise 5.8: 16-bit prefix adder with "grey cells".

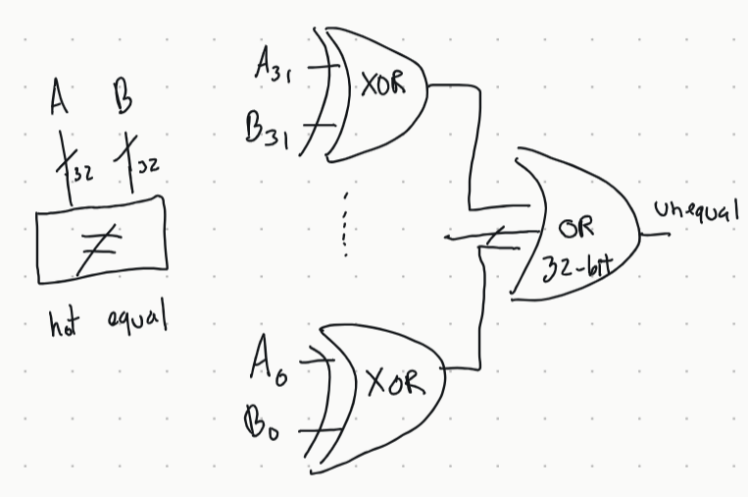

Figure 3: Exercise 5.8: 32-bit unequal unsigned comparator

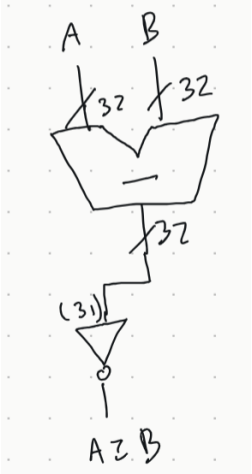

Figure 4: Exercise 5.8: 32-bit greater than or equal to unsigned comparator

Figure 5: Exercise 5.8: 32-bit less than unsigned comparator

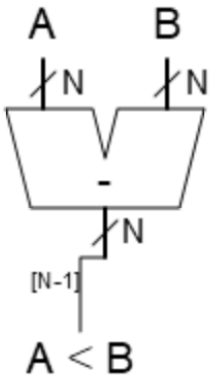

Figure 6: N-bit signed comparator

Solution:

(a) See Figure 3. We take the XOR of A_iB_i, and then OR all results.

(b) See Figure 4. We subtract, and if the most significant bit is 0, then $A \geq B$.

(c) See Figure 5. We subtract, and if the most significant bit is 1, then $A < B$.

Exercise 5.9. Consider the signed comparator of Figure 6.

(a) Give an example of two 4-bit signed numbers A and B for which a 4-bit signed comparator correctly computes $A < B$.

(b) Give an example of two 4-bit signed numbers A and B for which a 4-bit signed comparator incorrectly computes $A < B$.

(c) In general, when does the N-bit signed comparator operator incorrectly?

Solution: (a) It operates correctly on $A = 0_2 = 0 = 0000$ and $B = 1_2 = 0001$.

(b) It operates incorrectly on $A = 7_2 = 0111$ and $B = (-1)_2 = 1111$.

(c) As mentioned in the text, it operates incorrectly when there is overflow. Specifically, if A and B have opposite signs and the sign of the output S is not the same sign as A.

Exercise 5.10. Modify the N-bit signed comparator of Figure 6 to operate correctly when $A < B$ for all N-bit signed inputs A and B.

Solution: As discussed in Exercise 5.9, the "less than" signed comparator computes $A < B$ incorrectly when A and B have opposite signs and the sign of the output S is not the same as the sign of A; this indicates overflow has occurred. Letting V be the overflow signal and Y be the less-than signal, their equations are

$$V = A_{31}\bar{B}_{31}\bar{S}_{31} + \bar{A}_{31}B_{31}S_{31};$$
$$Y = V \oplus S.$$

See Figure 7.

Exercise 5.11. Design the 32-bit ALU shown in Figure 8 using your favorite HDL. You can make the to-level module either behavior or structural.

Solution: See code listing for `./hdl/11-alu-32/alu_32.vhd`:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity alu_32 is
  port(a, b:   in  STD_LOGIC_VECTOR(31 downto 0);
       control: in STD_LOGIC_VECTOR(1 downto 0);
         result: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture synth of alu_32 is
begin
  process(all) begin
    case control is
       when "11" => result <= (a or b);
        when "10" => result <= (a and b);
        when "01" => result <= STD_LOGIC_VECTOR(signed(a) - signed(b));
        when "00" => result <= STD_LOGIC_VECTOR(signed(a) + signed(b));
        when others => result <= (others => '0');
     end case;
  end process;
end;
```

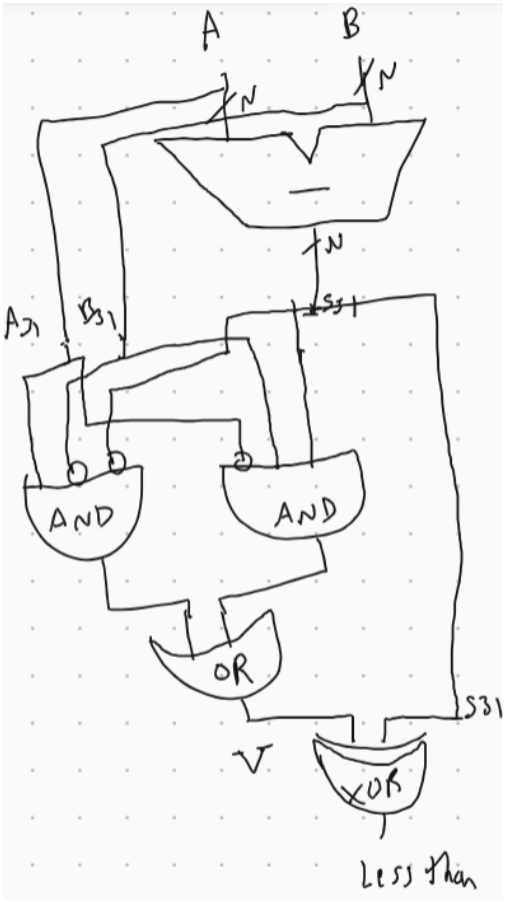

Figure 7: Unsigned N-bit comparator that accounts for overflow

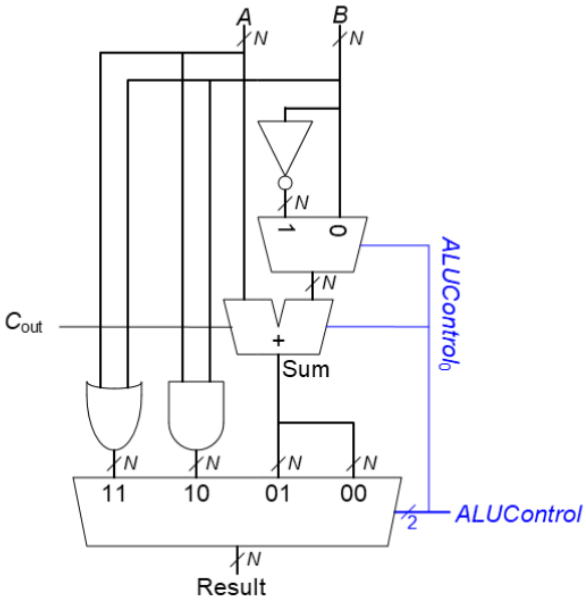

Figure 8: N-bit ALU

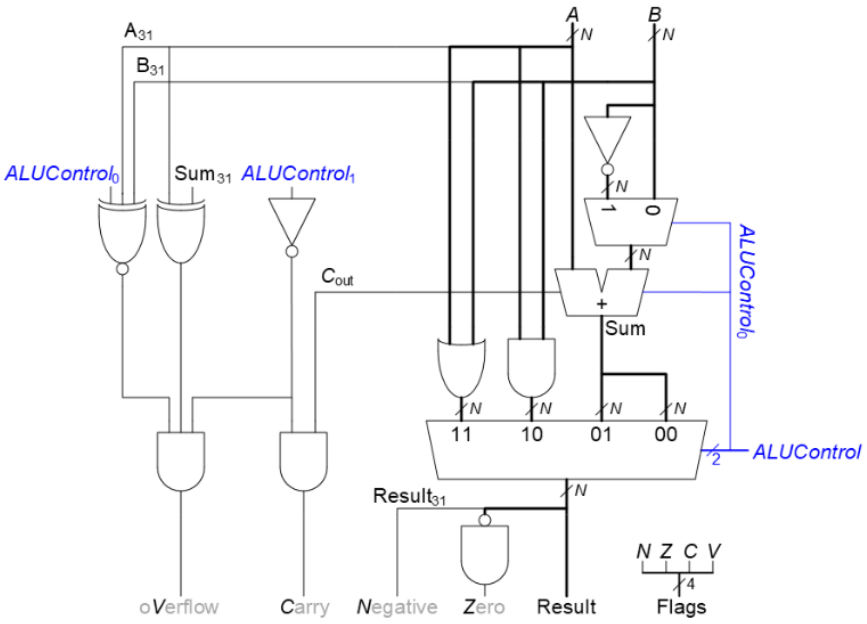

Figure 9: N-bit ALU with flags

Exercise 5.12. Design the 32-bit ALU shown in Figure 9 using your favorite HDL. You can make the to-level module either behavior or structural.

Solution: See code listing for `./hdl/12-alu-32-flags/alu_32_flags.vhd`:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity alu_32_flags is
  port(a, b:       in  STD_LOGIC_VECTOR(31 downto 0);
       control:    in  STD_LOGIC_VECTOR(1 downto 0);
         result:     out STD_LOGIC_VECTOR(31 downto 0);
         N, Z, C, V: out STD_LOGIC);
end;

architecture synth of alu_32_flags is
  signal preresult: STD_LOGIC_VECTOR(31 downto 0);
  signal sum: STD_LOGIC_VECTOR(32 downto 0); -- extra bit for the carry

  signal a_opp_sum, can_overflow: STD_LOGIC;
begin
  process(all) begin
     if control(0) = '0' then
       sum <= '0' & STD_LOGIC_VECTOR(signed(a) + signed(b));
     else
       sum <= '0' & STD_LOGIC_VECTOR(signed(a) - signed(b));
```

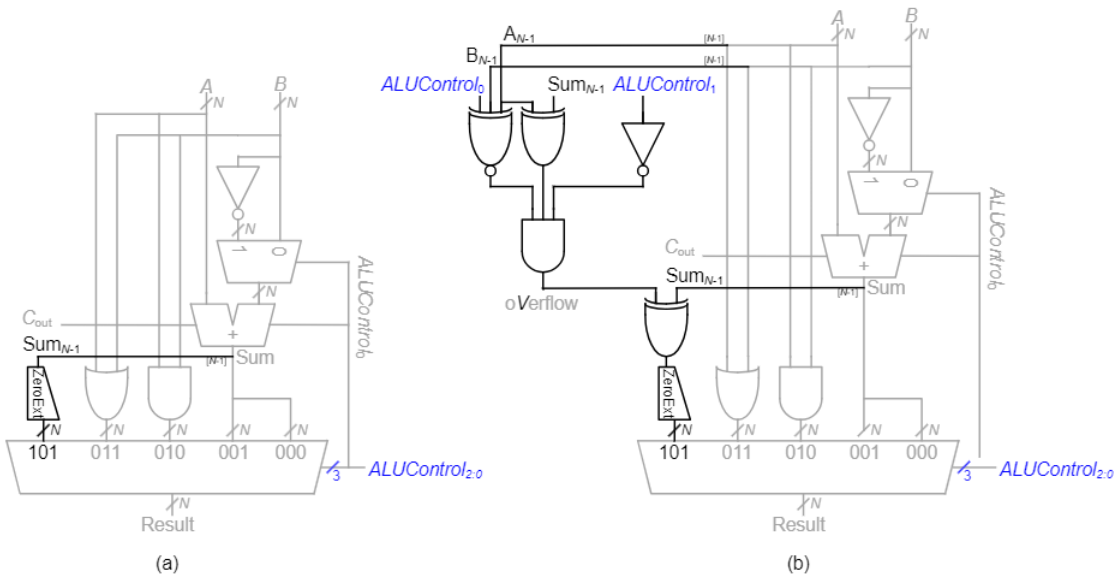

Figure 10: N-bit ALU with flags

```
    end if;

   case control is
      when "00" => preresult <= sum(31 downto 0);
       when "01" => preresult <= sum(31 downto 0);
       when "10" => preresult <= a and b;
       when "11" => preresult <= a or b;
       when others => preresult <= (others => '0');
    end case;

    C <= (not control(1)) and sum(32); -- carry bit.
    N <= preresult(31);
    gen: for i in 0 to 31 loop
      Z <= Z and not preresult(i);
    end loop;

    a_opp_sum <= sum(31) xor a(31);
   can_overflow <= not (control(0) xor a(31) xor b(31));

   V <= a_opp_sum and can_overflow and (not control(1));
    result <= preresult;
  end process;
end;
```

Exercise 5.13. Design the 32-bit ALU shown in Figure 10 (a) using your favorite HDL. You can make the to-level module either behavior or structural.

Solution: See code listing for `./hdl/13-alu-32-slt/alu_32_slt.vhd`:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity alu_32_slt is
  port(  a, b: in STD_LOGIC_VECTOR(31 downto 0);
       control: in STD_LOGIC_VECTOR(2 downto 0);
         result: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture synth of alu_32_slt is
begin
  process(all) begin
    case control is
       when "000" => result <= STD_LOGIC_VECTOR(signed(a) + signed(b));
        when "001" => result <= STD_LOGIC_VECTOR(signed(a) - signed(b));
        when "010" => result <= a and b;
        when "011" => result <= a or b;
        when "101" => if a < b then result <= (0 => '1', others => '0');
                      else          result <= (others => '0');
                          end if;
        when others => result <= (others => '0');
     end case;
  end process;
end;
```

Exercise 5.14. Design the 32-bit ALU shown in Figure 10 (b) using your favorite HDL. You can make the to-level module either behavior or structural.

Solution: See code listing for `./hdl/14-alu-32-slt-overflow/alu_32_slt_overflow.vhd`:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity alu_32_slt_overflow is
  port(  a, b: in STD_LOGIC_VECTOR(31 downto 0);
       control: in STD_LOGIC_VECTOR(2 downto 0);
         result: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture synth of alu_32_slt_overflow is
  signal tempsum: STD_LOGIC_VECTOR(31 downto 0);
  signal can_overflow, a_opp_sum, V: STD_LOGIC;
begin
  process(all) begin
    case control is
```

```
        when "000" => result <= STD_LOGIC_VECTOR(signed(a) + signed(b));
        when "001" => result <= STD_LOGIC_VECTOR(signed(a) - signed(b));
        when "010" => result <= a and b;
        when "011" => result <= a or b;
        when "101" => tempsum <= STD_LOGIC_VECTOR(signed(a) - signed(b));
                      can_overflow <= not (control(0) xor a(31) xor b(31));
                      a_opp_sum <= a(31) xor tempsum(31);
                          V <= can_overflow and a_opp_sum and not control(1);
                          result <= (0 => V and tempsum(31), others => '0');
        when others => result <= (others => '0');

    end case;
  end process;
end;
```

Exercise 5.19. Build an *unsigned comparison unit* that compares two unsigned numbers A and B. The unit's input is the *Flags* signal (N, Z, C, V) from the ALU of Figure **??**, with the ALU performing subtraction: $A-B$. The unit's outputs are HS, LS, HI, and LO, which indicate that A is higher than or the same as (HS), lower than or the same as (LS), high (HI), or lower (LO) than B.

(a) Write minimal equations for HS, LS, HI and LO in terms of N, Z, C, and V.

(b) Sketch circuits for HS, LS, HI, and LO.

Solution: (a) As discussed in the text, when using the subtraction of two inputs A and B to do comparison, we know that $A < B$ if there is no carry. On the other hand, $A \geq B$ is there is a carry. We can then use the zero signal to know whether it can be less than or equal to or greater than. The N and V signals are don't-cares. The equations are below:

$$HS = C, \quad LS = Z + C, \quad HI = \bar{Z}C, \quad LO = \bar{C}$$

(b) See the sketch in Figure 11.

Exercise 5.20. Build an *signed comparison unit* that compares two unsigned numbers A and B. The unit's input is the *Flags* signal (N, Z, C, V) from the ALU of Figure **??**, with the ALU performing subtraction: $A - B$. The unit's outputs are GE, LE, GT, and LT, which indicate that A is greater than or equal to (GE), less than or equal to (LE), greater than (GT), or less than (LT) than B.

(a) Write minimal equations for GE, LE, GT and LT in terms of N, Z, C, and V.

(b) Sketch circuits for GE, LE, GT, and LT.

Solution: (a) As discussed in the text, the N (negative) flag communicates that $A - B$ is negative. But to use it for signed magnitude comparison, we must also use the V flag to account for overflow. The equations are below:

$$LT = N \oplus V, \quad GE = \overline{N \oplus V}, \quad LE = Z + N \oplus V, \quad GT = \bar{Z} \cdot \overline{N \oplus V}$$

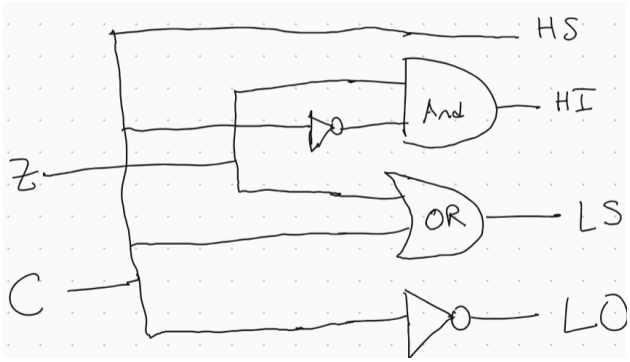

Figure 11: Exercise 5-19: Unsigned Comparison Unit

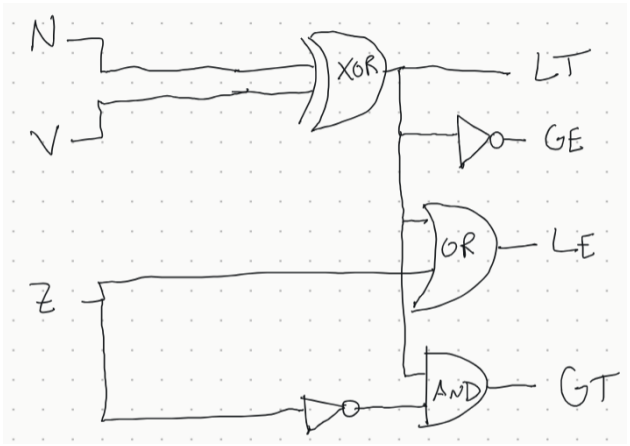

Figure 12: Exercise 5-20: signed Comparison Unit

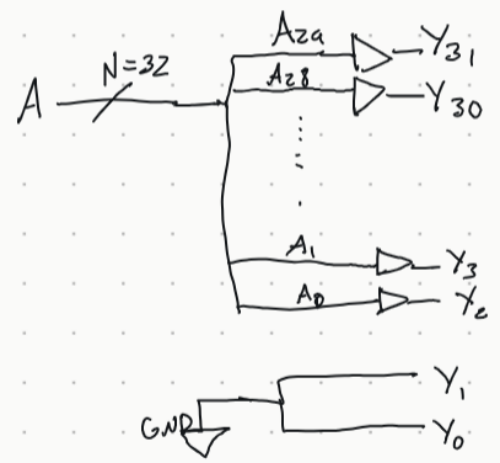

Figure 13: Exercise 21: Shifter that always shifts its 32-bit input left by 2 bits.

(b) See the sketch in Figure 12.

Exercise 5.21. Design a shifter that always shifts a 32-bit input left by 2 bits. The input and output are both 32 bits. Explain the design in words and sketch a schematic. Implement your design in your favorite HDL.

Solution: When the input A is shifted left by 2, the output will always receive 0 in its two least significant fits. See the sketch in Figure refexercise-21-32bit-shift-left-2. See code listing for `./hdl/21-shifter-left-32bit-2/shifter_left_32bit_2.vhd`:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity shifter_left_32bit_2 is
  port(a: in STD_LOGIC_VECTOR(31 downto 0);
       y: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture synth of shifter_left_32bit_2 is
begin
  y <= a(29 downto 0) & "00";
end;
```

Exercise 5.24. Explain how to build any N-bit shifter or rotator using only $N \log_2 N$ 2:1 multiplexers.

Solution: Given an N-bit input, we also accept a $\log_2 N$-bit input named *shamt* which indicates the amount by which to shift. If we could use $N : 1$ multiplexers, then N such multiplexers, each with the *shamt* input as its select signal. To build an $N : 1$ multiplexer, we need $log_2 N$ $2 : 1$ multiplexers. Therefore, the N multiplexers, each $N : 1$, would take $N \log_2 N$ $2 : 1$ multiplexers.

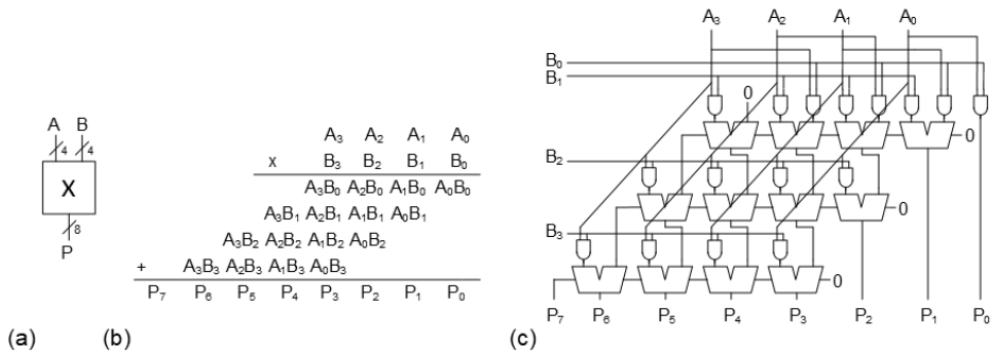

Figure 14: Exercise 26: 4 by 4 multiplier

Consider the left shift case. Arrange the $N:1$ multiplexers so that the topmost multiplexer has Y_{N-1} as its output, and the bottommost has Y_0 as its output. Then arrange the select signals from top to bottom, increasing from 0 to $N-1$. If we are given the 0 shift signal, then no shift occurs, so A_{N-1} should be connected to 0 in the topmost multiplexer, A_{N-2} is connected to 0, and so on. If the signal is a (64-bit) 1, then A_{N-2} should be connected so that $Y_{N-1} = A_{N-2}$, and in the second multiplexer, A_{N-3} should be connected to the 1 signal so that $Y_{N-2} = A_{N-3}$, and so on.

Exercise 5.26. Find the critical path for the unsigned 4×4 multiplier from Figure **??** in terms of an AND gate delay (t_{AND}) and a full adder (t_{FA}). What is the delay of an $N \times N$ multiplier built the same way?

Solution: All partial products are computed simultaneously, which accounts for 1 AND gate. Then, since P_7 depends on the results of the previous carries, it is on the critical path. From the function and the circuit implementation, there are 8 adders on the critical path. That is, the delay is

$$t_{\text{MULT},4\times4} = t_{\text{AND}} + 8t_{FA}$$

In an $N \times N$ multiplier, we would have $N-1$ levels of adders.There are 2 adders of each level on the critical path, except that the last level's adders are all on the critical path. That is:

$$\begin{aligned} t_{\text{MULT},N\times N} &= t_{\text{AND}} + ((N-2)\cdot 2 + N)t_{FA} \\ &= t_{\text{AND}} + (3N-4)t_{FA} \end{aligned}$$

To verify this is correct, if $N = 4$, then $3 \cdot 4 - 4 = 8$.

Exercise 5.29. A *sign extension unit* extends a two's complement number from M to N ($N > M$) bits by copying the most significant bit of the input into the upper bits of the output (see section 1.4.6). It receives an M-bit input A and produces an N-bit output Y. Sketch a circuit for a sign extension unit with a 4-bit input and an 8-bit output. Write the HDL for your design.

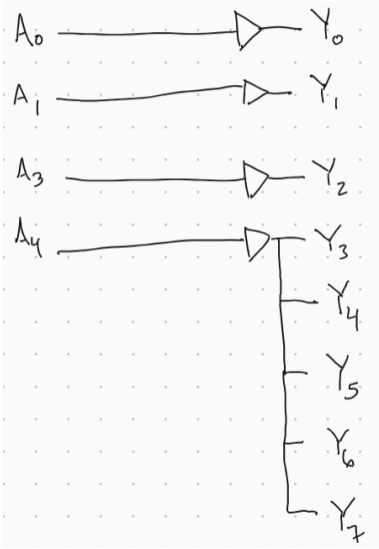

Figure 15: Exercise 29: Sign Extension Unit (4 bits to 8 bits)

Solution: See Figure 15. See also the code listing `./hdl/29-sign-extension-4-8/sign_extension_4_8.v`

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity sign_extension_4_8 is
  port(a: in STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of sign_extension_4_8 is
begin
  y(3 downto 0) <= a;
  y(7 downto 4) <= (others => a(3));
end;
```

Exercise 5.30. A *zero extension unit* extends an unsigned number from M to N bits ($N > M$) by putting zeros into the upper bits of the output (see section 1.4.6). Sketch a circuit for a zero extension unit with a 4-bit input and an 8-bit output. Write the HDL for your design.

Solution: See Figure 16. See also the code listing `./hdl/30-zero-extension-4-8/zero_extension_4_8.v`

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity zero_extension_4_8 is
  port(a: in STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(7 downto 0));
```

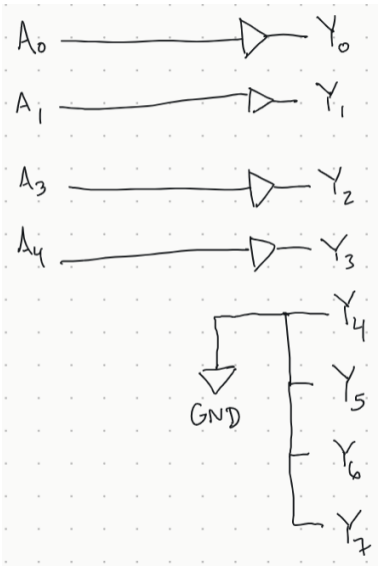

Figure 16: Exercise 30: Zero Extension Unit (4 bits to 8 bits)

```
end;

architecture synth of zero_extension_4_8 is
begin
  y(3 downto 0) <= a;
  y(7 downto 4) <= (others => '0');
end;
```

Exercise 5.31. Compute $111001.000_2/001100.000_2$ in binary using the standard division algorithm from elementary school. Show your work.

Solution: See Figure 17. Note that the standard subtraction, we add $\bar{B}+1$ at each step.

Exercise 5.32. What is the range of numbers that can be represented by the following number systems?

(a) U12.12 format (24-bit unsigned fixed-point numbers with 12 integer bits and 12 fraction bits).

(b) 24-bit sign/magnitude fixed-point numbers with 12 integer bits and 12 fraction bits.

(c) Q12.12 format (24-bit two's complement fixed-point numbers with 12 integer bits and 12 fraction bits).

Solution: (a) The smallest number is 0. The highest is asserts 12 integer bits and 12 fraction bits. Since $\sum_{i=0}^{11} 2^i = 2^{12}-1$, and $\sum_{i=1}^{12} 2^{-i} = 1-2^{-12}$, so the maximum value is $2^{12}-1+1^{-12} = 2^{12}-2^{-12}$.

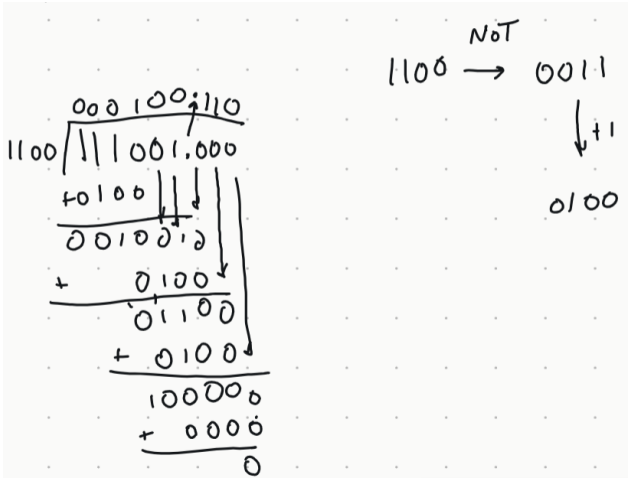

Figure 17: Exercise 31: Binary Long Division

(b) In this case, the the maximum value is the same, obtained when all but the sign bit are 1: $2^{12} - 2^{-12}$. For the smallest number, all bits are 1 (including the sign bit). The minimum value is the negative of the maximum value.

(c) Maximum is $2^{12} - 2^{-12}$, and the minimum value is $-2^{12} + 2^{-12} - 1$, which is 1 lower than the maximum value.

Exercise 5.33. Express the following base 10 numbers in 16-bit fixed point sign/magnitude format with eight integer bits and eight fraction bits. Express your answer in hexadecimal.

(a) -13.5625

(b) 42.3125

(c) -17.15625

Solution:

(a) To do the conversion, we start with 13.5625, and do repeated division by powers of 2, subtracting each time:

$$\begin{aligned}
13.5625 \geq 2^3 &\implies 1 \\
5.5625 \geq 2^2 &\implies 1 \\
1.5625 < 2^1 &\implies 0 \\
1.5625 \geq 2^0 &\implies 1 \\
0.5625 \geq 2^{-1} &\implies 1 \\
0.0625 < 2^{-2} &\implies 0 \\
0.0625 < 2^{-3} &\implies 0 \\
0.0625 \geq 2^{-4} &\implies 1 \\
0 &
\end{aligned}$$

Therefore, $13.5625_{10} = 00001101.10010000_2$. For 16-bit fixed-point sign/magnitude format, we use 8 fraction bits and 8 integer bits, setting the sign bit to 1 for negative numbers. Hence, the number is

$$10001101.10010000_2 = \texttt{0x8D.90}$$

(b) We repeat the same steps:

$$\begin{aligned}
42.3125 \geq 2^5 &\implies 1 \\
10.3125 < 2^4 &\implies 0 \\
10.3125 \geq 2^3 &\implies 1 \\
2.3125 < 2^2 &\implies 0 \\
2.3125 \geq 2^1 &\implies 1 \\
0.3125 < 2^0 &\implies 0 \\
0.3125 < 2^{-1} &\implies 0 \\
0.3125 \geq 2^{-2} &\implies 1 \\
0.0625 < 2^{-3} &\implies 0 \\
0.0625 \geq 2^{-4} &\implies 1 \\
0 &
\end{aligned}$$

Therefore, $42.3125_{10} = 00101010.0101 = \texttt{0x2A.50}$.

(c) Finally, same as part (a), though I will skip some powers:

$$\begin{aligned}
17.15625 \geq 2^4 &\implies 1 \\
1.15625 \geq 2^0 &\implies 1 \\
0.15625 \geq 2^{-3} &\implies 1 \\
0.03125 \geq 2^{-5} &\implies 1
\end{aligned}$$

Therefore, $17.15625_{10} = 10001.00101_2$, so its negative has a 1 as a sign bit, meaning $-17.15625_{10} = 10010001.00101000_2 = \texttt{0x91.28}$.

Exercise 5.35. Express the base 10 numbers in Exercise 5.33 in Q8.8 format (16-bit fixed-point two's complement format with eight integer bits and eight fraction bits). Express your answer in hexadecimal.

Solution: (a) Recall from Exercise 5.33 that $13.5625_{10} = 00001101.10010000_2$, and hence, in two's complement we have $-13.5625_{10} = 11110010.01110000_2 = \texttt{0xF2.70}$.

(b) Same as Exercise 33: $42.3125_{10} = 00101010.0101 = \texttt{0x2A.50}$.

(c) Recall from Exercise 5.33 that $17.15625_{10} = 00010001.00101000_2$. When negating it using two's complement, we get $-17.15625_{10} = 11101110.11011000_2 = \texttt{0xEE.D8}$.

Exercise 5.37. Express the base 10 numbers in Exercise 5.33 in IEEE 754 single-precision floating-point format. Express your answer in hexadecimal.

Solution: The IEEE 754 single-precision format uses 32 bits, where

- 1 bit is assigned to most significant bit for sign
- 8 bits are assigned to a biased exponent
- 23 bits are assigned to the fraction portion

(a) Recall that $13.5625_{10} = 00001101.10010000_2$ from Exercise 5.33. Since the number is negative, the sign bit will be 1. By moving the decimal place to the left 3 times, we get 1.1011001×2^3, meaning the exponent is 3 and the mantissa is 11011001_2 (with more 0s padded until it has a total of 23 bits). Since the leading 1 is implicit and is not to be included in the mantissa, we remove it to have 1011001_2. Finally, the biased exponent is the original exponent plus a constant bias, which for a 32-bit floating-point number is 127. That is: $3 + 127 = 130 = 10000010_2$, which is the biased exponent. Hence the IEEE 754 single-precision floating point format representation of -13.5625_{10} is

Sign (1-bit)	Biased Exponent (8-bits)	Fraction
1	1000 0010	101 1001 0000 0000 0000 0000

(b) Recall that $42.3125_{10} = 00101010.0101_2$. Since the number is positive, the sign bit is 0. By moving the decimal place left 5 times, we get $1.010100101_2 \times 2^5$. Since the leading bit is always 1, the fraction bit does not use the leading bit of the mantissa, so the leading portion of the fraction part of the representation is 010100101. Moreover, since the exponent is 5, we bias it by adding 127, getting $132 = 10000101_2$ so the IEEE 754 single-point floating point representation of 42.3125_{10} is:

Sign (1-bit)	Biased Exponent (8-bits)	Fraction
0	1000 0101	010 1001 0100 0000 0000 0000

(c) We are converting -17.15625_{10}, so the sign bit is 1. From Exercise 5.33, we have $17.15625_{10} = 10001.00101_2$. By moving the decimal place left 4 times, we get $1.000100101_2 \times 2^4$. Removing the leading bit, we get the leading fraction part which is 000100101_2. We add 127 to the exponent to bias it, getting $4 + 127 = 131 = 10000100_2$. The IEEE 754 single-point floating point representation of -17.15625_{10} is

Sign (1-bit)	Biased Exponent (8-bits)	Fraction
1	1000 0100	000 1001 0100 0000 0000 0000

Exercise 5.39. Convert the following Q4.4 (two's complement binary fixed-point numbers) to base 10. The implied binary point is explicitly shown to aid in your interpretation.

(a) 0101.1000

(b) 1111.1111

(c) 1000.0000

Solution:

(a) Since the leading bit is 0, we have a positive number. Then we add the powers of 2:

$$0101.1000_2 = 2^2 + 2^0 + 2^{-1} = 5.5_{10}$$

(b) Since the leading bit is 1, this is a negative number. We take its two's complement by inverting all of the bits and adding 1, yielding: 0000.0001_2. Then we convert it to base 10:

$$0000.0001_2 = 2^{-4} = -0.0625_{10}$$

(c) Since the leading bit is 1, the number is negative. We take its two's complement by inverting all of the bits and adding 1, yielding: 1000.0000_2. Then we convert it to base 10:

$$1000.0000_2 = -2^3 = -8_{10}$$

Exercise 5.41. When adding two floating-point numbers, the number with smaller exponent is shifted. Why is this? Explain in words and give an example to justify your explanation.

Solution: When fraction bits of both numbers are converted to a mantissa, they both have 24 bits (a 1 to the left of the decimal, followed by a decimal point, followed by 23 bits). However, the numbers do not have the same order of magnitude; they differ by an order of magnitude indicated by their exponent, which must be accounted for. Rather than move both numbers according to their order of magnitude, we express the smaller number in the same order of magnitude as the larger number. We do so by taking the difference of their exponents. We shift the smaller number a number of places equals to the absolute value of the difference. Then, the exponent used for both is the larger of the two. For example, consider

$$3.87 \times 10^3 + 2.41 \times 10^5$$

in base 10. We cannot just add 2.81 and 3.87. Instead, since their exponents differ by -22, the we shift the decimal in 3.87 by 2 units left to make it 0.0387×10^5. Now we have

$$0.0387 \times 10^5 + 2.41 \times 10^5 = 2.4487 \times 10^5$$

A similar case holds in base 2. For example, recall from Exercise 5.33 that the floating-point representation of 42.3125_{10} is

Sign (1-bit)	Biased Exponent (8-bits)	Fraction
0	1000 0101	010 1001 0100 0000 0000 0000

and the floating-point representation of -17.15625_{10} is

Sign (1-bit)	Biased Exponent (8-bits)	Fraction
1	1000 0100	000 1001 0100 0000 0000 0000

To add them, we follow the 8-step process described in Section 5.3.2:

1. Extract the exponent and fraction bits:

0	1000 0101	010 1001 0100 0000 0000 0000
1	1000 0100	000 1001 0100 0000 0000 0000

2. Then, prepend leading 1 to form the mantissa:

0	1000 0101	**1**.010 1001 0100 0000 0000 0000
1	1000 0100	**1**.000 1001 0100 0000 0000 0000

3. We compare exponents, i.e., subtract them:

0	1000 0101	1.010 1001 0100 0000 0000 0000
1	1000 0100	1.000 1001 0100 0000 0000 0000
	1	

4. The result of the previous step was 1, meaning that the second operand has a smaller exponent and order of magnitude; this is correct, since 42.3125_{10} uses a base 2 exponent of 5, while -17.15625_{10} uses a base 2 exponent of 4. Hence, we shift the base 2 representation of -17.15625_{10} right by 1 unit and change its exponent to match the larger 1:

0	1000 0101	1.010 1001 0100 0000 0000 0000 0
1	1000 0101	0.100 0100 1010 0000 0000 0000 0

The addition can then take place in the next step, keeping in mind that the second operand is negative.

Exercise 5.44. Expand the steps in Section 5.3.2 for performing floating-point addition to work for negative as well as positive floating-point numbers.

Solution:

1. Extract exponent and fraction bits.
2. Prepend leading 1 to form the mantissa.
3. Compare exponents.
4. Shift smaller mantissa if necessary.
5. Take two's complement of first operand's mantissa if its sign bit is 1.
6. If the second operand is positive, negative it using the two's complement. Otherwise, leave it unchanged.

7. Add the mantissas.
8. Normalize mantissa and adjust exponent if necessary.
9. Round result.
10. Assemble exponent and fraction back into floating-point numbers.

Exercise 5.45. Consider IEEE 754 single-precision floating-point numbers.

(a) How many numbers can be represented by the IEEE 754 single-precision floating-point format? You need not count $\pm\infty$ or NaN.

(b) How many additional numbers could be represented if $\pm\infty$ and NaN were not represented?

(c) Explain why $\pm\infty$ and NaN are given special representations.

Solution: (a) The 32-bit representation means that 2^{32} binary sequences can be represented. We subtract 1 because 0 has a positive and negative representation. We subtracting 2 due to $\pm\infty$. Next, note that NaN occurs when all exponents bits are asserted and the fraction bit is nonzero, regardless of the sign bit. Since the fraction portion has 23 bits, there are $2^{23} - 1$ nonzero combinations. we double it to account for the don't care in the sign bit, so there are $2^{24} - 2$ ways to represent NaN. Hence, the total amount of numbers that can be represented with IEEE 754 single-precision floating point format is

$$2^{32} - 1 - 2 - (2^{24} - 1) = 2^{32} - 2^{24} - 2 = 4,278,190,078_{10}$$

(b) If $\pm\infty$ and NaN were not represented, we would be able to represent $2^{32} - 1 = 4,294,967,294$ numbers.

(c) It's possible for floating-point operations to yield numbers bigger than what can be represented with a finite set of bits. To indicate this, the $\pm\infty$ representations communicate that an operation caused overflow. Meanwhile, NaN represents operations that do not yield a *real* number. Their values can be used to make decisions about how to proceed with a calculation.