1 The simplest bit blasting transformations for the signed comparisons $\geq_s$ and $>_s$ check for the sign bit (which is 1 for negative numbers), and afterwards relies on the respective unsigned comparisons:

$$\begin{aligned}\mathbf{B}_r(\mathbf{x}_{k+1} \geq_s \mathbf{y}_{k+1}) =& (\neg x_k \wedge y_k) \vee (x_k \wedge y_k \wedge \mathbf{B}(\mathbf{y}[k-1{:}0] \geq_u \mathbf{x}[k-1{:}0])) \vee \\ & (\neg x_k \wedge \neg y_k \wedge \mathbf{B}(\mathbf{x}[k-1{:}0] \geq_u \mathbf{y}[k-1{:}0])) \\ \mathbf{B}_r(\mathbf{x}_{k+1} >_s \mathbf{y}_{k+1}) =& \mathbf{B}_r(\mathbf{x}_{k+1} \geq_s \mathbf{y}_{k+1}) \wedge \mathbf{B}_r(\mathbf{x}_{k+1} \neq \mathbf{y}_{k+1})\end{aligned}$$

2 (a) The replacement is incorrect. For example, the SMT encoding

```
(declare-const x (_ BitVec 8))
(declare-const c1 (_ BitVec 8))
(declare-const c2 (_ BitVec 8))
(declare-const before (_ BitVec 8))
(declare-const after (_ BitVec 8))
(assert (= before (bvudiv (bvlshr x c1) c2)))
(assert (= after (bvudiv x (bvshl c1 c2))))
(assert (not (= before after)))
(check-sat)
(get-model)
```

shows that for c1 = $\mathbf{x01}_8$, c2 = $\mathbf{x05}_8$, and x = $\mathbf{x44}_8$ the left-hand side evaluates to $\mathbf{x06}_8$ while the right-hand side evaluates to $\mathbf{x02}_8$. A counterexample is also found when the bit vector size is changed to 16.

(b) The replacement is incorrect. For example, the SMT encoding

```
(declare-const p (_ BitVec 8))
(declare-const x (_ BitVec 8))
(declare-const a (_ BitVec 8))
(declare-const b (_ BitVec 8))
(declare-const before (_ BitVec 8))
(declare-const after (_ BitVec 8))
(define-fun is-power-of-two ((x (_ BitVec 8))) Bool
  (= #x00 (bvand x (bvsub x #x01))))
(assert (is-power-of-two p))
(assert (= before (bvudiv x (bvlshr (bvshl p a) b))))
(assert (= after (bvudiv x (bvshl p (bvsub a b)))))
(assert (not (= before after)))
(check-sat)
(get-model)
```

shows that for $\mathtt{a} = \mathbf{x00}_8$, $\mathtt{b} = \mathbf{x02}_8$, $\mathtt{p} = \mathbf{x80}_8$, and $\mathtt{x} = \mathbf{x7e}_8$ the left-hand side evaluates to $\mathbf{x03}_8$ while the right-hand side evaluates to $\mathbf{xff}_8$. A counterexample is also found when the bit vector size is changed to 16.

(c) The replacement is correct since the following SMT encoding is unsatisfiable:

```
(declare-const a (_ BitVec 8))
(declare-const b (_ BitVec 8))
(declare-const before (_ BitVec 8))
(declare-const after (_ BitVec 8))
(assert (= before (bvadd (bvsub #x00 a) (bvsub #x00 b))))
(assert (= after (bvsub #x00 (bvadd a b))))
(assert (not (= before after)))
(check-sat)
```

This is still true when changing the bit width to 16.

$\boxed{4}$ See `abs_and_avg.py`. In the following, we assume for simplicity that all numbers have four bits.

(a) If the shift to compute the mask is implemented arithmetically (shifting in sign bits), the hack is correct. Otherwise, a counterexample can be found. For the first hack, if `x=-6`, the hack yields `-6` instead of `6`. For the second, if `x=-3`, the hack yields `-5` instead of `3`.

(b) First case: `x` and `y` are unsigned. Then `(x + y) >> 1` is correct (the shift will always shift in 0s, according to the C standard). However, `((x ^ y) >> 1) + (x & y)` can behave differently than `(x + y) / 2`, e.g. for `x = 10` and `y = 14` the division overflows and yields `4` while the bit hack avoids the overflow and gives `12`. (So one could also say that the hack is more correct.)

Second case: `x` and `y` are signed. For `(x + y) >> 1`, the result changes now depending on the implementation of the right shift operator. However, for both implementations the results differ: If `x = -1` and `y = 0`, the expression `(x+y)/2` yields 0. Using an arithmetic shift, `(x + y) >> 1` yields `-1`, and using a logical shift `7`. Also the second hack behaves differently depending on the implementation of the shift operator, but differs in both cases from `(x+y)/2`. If the shift is arithmetic, for `x = 7` and `y = -7` the expression `(x+y)/2` evaluates to 0, while the hack yields `-1`. If the shift is logical, for `x = 6` and `y = -6` the expression `(x+y)/2` evaluates to 0, while the hack yields `7`.