

# Veriopt Theories

March 17, 2023

## Contents

<b>1</b>	<b>Canonicalization Optimizations</b>	<b>1</b>
1.1	AbsNode Phase . . . . .	3
1.2	AddNode Phase . . . . .	8
1.3	AndNode Phase . . . . .	11
1.4	BinaryNode Phase . . . . .	16
1.5	ConditionalNode Phase . . . . .	16
1.6	MulNode Phase . . . . .	20
1.7	Experimental AndNode Phase . . . . .	30
1.8	NotNode Phase . . . . .	41
1.9	OrNode Phase . . . . .	42
1.10	ShiftNode Phase . . . . .	46
1.11	SignedDivNode Phase . . . . .	47
1.12	SignedRemNode Phase . . . . .	48
1.13	SubNode Phase . . . . .	48
1.14	XorNode Phase . . . . .	53
1.15	NegateNode Phase . . . . .	55
1.16	AddNode . . . . .	58
1.17	NegateNode . . . . .	59

## 1 Canonicalization Optimizations

```
theory Common
imports
  OptimizationDSL.Canonicalization
  Semantics.IRTreeEvalThms
begin

lemma size-pos[size-simps]: 0 < size y
  apply (induction y; auto?)
  by (smt (z3) add-2-eq-Suc' add-is-0 not-gr0 size.elims size.simps(12) size.simps(13)
size.simps(14) size.simps(15) zero-neq-numeral zero-neq-one)
```

**lemma** *size-non-add*[*size-simps*]:  $\text{size } (\text{BinaryExpr op } a \ b) = \text{size } a + \text{size } b + 2$   
 $\longleftrightarrow \neg(\text{is-ConstantExpr } b)$

**by** (*induction b*; *induction op*; *auto simp: is-ConstantExpr-def*)

**lemma** *size-non-const*[*size-simps*]:

$\neg \text{is-ConstantExpr } y \implies 1 < \text{size } y$

**using** *size-pos* **apply** (*induction y*; *auto*)

**by** (*metis Suc-lessI add-is-1 is-ConstantExpr-def le-less linorder-not-le n-not-Suc-n numeral-2-eq-2 pos2 size.simps(2) size-non-add*)

**lemma** *size-binary-const*[*size-simps*]:

$\text{size } (\text{BinaryExpr op } a \ b) = \text{size } a + 2 \longleftrightarrow (\text{is-ConstantExpr } b)$

**by** (*induction b*; *auto simp: is-ConstantExpr-def size-pos*)

**lemma** *size-flip-binary*[*size-simps*]:

$\neg(\text{is-ConstantExpr } y) \longrightarrow \text{size } (\text{BinaryExpr op } (\text{ConstantExpr } x) \ y) > \text{size } (\text{BinaryExpr op } y \ (\text{ConstantExpr } x))$

**by** (*metis add-Suc not-less-eq order-less-asm plus-1-eq-Suc size.simps(11) size.simps(2) size-non-add*)

**lemma** *size-binary-lhs-a*[*size-simps*]:

$\text{size } (\text{BinaryExpr op } (\text{BinaryExpr op}' a \ b) \ c) > \text{size } a$

**by** (*metis add-lessD1 less-add-same-cancel1 pos2 size-binary-const size-non-add*)

**lemma** *size-binary-lhs-b*[*size-simps*]:

$\text{size } (\text{BinaryExpr op } (\text{BinaryExpr op}' a \ b) \ c) > \text{size } b$

**by** (*metis IRExpr.disc(42) One-nat-def add.left-commute add.right-neutral is-ConstantExpr-def less-add-Suc2 numeral-2-eq-2 plus-1-eq-Suc size.simps(11) size-binary-const size-non-add size-non-const trans-less-add1*)

**lemma** *size-binary-lhs-c*[*size-simps*]:

$\text{size } (\text{BinaryExpr op } (\text{BinaryExpr op}' a \ b) \ c) > \text{size } c$

**by** (*metis IRExpr.disc(42) add.left-commute add.right-neutral is-ConstantExpr-def less-Suc-eq numeral-2-eq-2 plus-1-eq-Suc size.simps(11) size-non-add size-non-const trans-less-add2*)

**lemma** *size-binary-rhs-a*[*size-simps*]:

$\text{size } (\text{BinaryExpr op } c \ (\text{BinaryExpr op}' a \ b)) > \text{size } a$

**by** (*smt (verit, best) less-Suc-eq less-add-Suc2 less-add-same-cancel1 linorder-neqE-nat not-add-less1 order-less-trans pos2 size.simps(4) size-binary-const size-non-add*)

**lemma** *size-binary-rhs-b*[*size-simps*]:

$\text{size } (\text{BinaryExpr op } c \ (\text{BinaryExpr op}' a \ b)) > \text{size } b$

**by** (*metis add.left-commute add.right-neutral is-ConstantExpr-def lessI numeral-2-eq-2 plus-1-eq-Suc size.simps(11) size.simps(4) size-non-add trans-less-add2*)

**lemma** *size-binary-rhs-c*[*size-simps*]:

$\text{size } (\text{BinaryExpr op } c \ (\text{BinaryExpr op}' a \ b)) > \text{size } c$

```

by simp

lemma size-binary-lhs[size-simps]:
  size (BinaryExpr op x y) > size x
  by (metis One-nat-def Suc-eq-plus1 add-Suc-right less-add-Suc1 numeral-2-eq-2
size-binary-const size-non-add)

lemma size-binary-rhs[size-simps]:
  size (BinaryExpr op x y) > size y
  by (metis IRExpr.disc(42) add-strict-increasing is-ConstantExpr-def linorder-not-le
not-add-less1 size.simps(11) size-non-add size-non-const size-pos)

lemmas arith[size-simps] = Suc-leI add-strict-increasing order-less-trans trans-less-add2

definition well-formed-equal :: Value  $\Rightarrow$  Value  $\Rightarrow$  bool
  (infix  $\approx$  50) where
    well-formed-equal v1 v2 = (v1  $\neq$  UndefVal  $\longrightarrow$  v1 = v2)

lemma well-formed-equal-defn [simp]:
  well-formed-equal v1 v2 = (v1  $\neq$  UndefVal  $\longrightarrow$  v1 = v2)
  unfolding well-formed-equal-def by simp

end

1.1 AbsNode Phase

theory AbsPhase
  imports
    Common
  begin

  phase AbsNode
    terminating size
  begin

```

```

lemma abs-pos:
  fixes v :: ('a :: len word)
  assumes 0  $\leq_s$  v
  shows (if v < s 0 then  $-$  v else v) = v
  by (simp add: assms signed.leD)

lemma abs-neg:
  fixes v :: ('a :: len word)
  assumes v < s 0
  assumes  $-(2 \wedge (\text{Nat.size } v - 1)) <_s v$ 
  shows (if v < s 0 then  $-$  v else v) =  $-$  v  $\wedge$  0 < s  $-v$ 

```

by (smt (verit, ccfv-SIG) assms(1) assms(2) signed-take-bit-int-greater-eq-minus-exp  
 signed-take-bit-int-greater-eq-self-iff sint-0 sint-word-ariths(4) word-sless-alt)

**lemma** *abs-max-neg*:  
 fixes  $v :: ('a :: \text{len word})$   
 assumes  $v <_s 0$   
 assumes  $-(2^{\wedge}(\text{Nat.size } v - 1)) = v$   
 shows  $-v = v$   
 using *assms*  
 by (metis *One-nat-def add.inverse-neutral double-eq-zero-iff mult-minus-right size-word.rep-eq*)

**lemma** *final-abs*:  
 fixes  $v :: ('a :: \text{len word})$   
 assumes *take-bit* ( $\text{Nat.size } v$ )  $v = v$   
 assumes  $-(2^{\wedge}(\text{Nat.size } v - 1)) \neq v$   
 shows  $0 \leq_s (\text{if } v <_s 0 \text{ then } -v \text{ else } v)$

**proof** (cases  $v <_s 0$ )  
 case *True*  
 then show ?thesis  
**proof** (cases  $v = -(2^{\wedge}(\text{Nat.size } v - 1)))$   
 case *True*  
 then show ?thesis using *abs-max-neg*  
 using *assms* by *presburger*  
 next  
 case *False*  
 then have  $-(2^{\wedge}(\text{Nat.size } v - 1)) <_s v$   
 unfolding *word-sless-def* using *signed-take-bit-int-greater-self-iff*  
 by (smt (verit, best) *One-nat-def diff-less double-eq-zero-iff len-gt-0 lessI less-irrefl*  
*mult-minus-right neg-equal-0-iff-equal signed.rep-eq signed-of-int*  
*signed-take-bit-int-greater-eq-self-iff signed-word-eqI sint-0 sint-range-size*  
*sint-sbintrunc' sint-word-ariths(4) size-word.rep-eq unsigned-0 word-2p-lem*  
*word-sless.rep-eq word-sless-def*)  
 then show ?thesis  
 using *abs-neg abs-pos signed.nless-le* by *auto*  
 qed  
 next  
 case *False*  
 then show ?thesis using *abs-pos* by *auto*  
 qed

**lemma** *wf-abs*:  $\text{is-IntVal } x \implies \text{intval-abs } x \neq \text{UndefVal}$   
 using *intval-abs.simps* unfolding *new-int.simps*  
 using *is-IntVal-def* by *force*

**fun** *bin-abs* :: 'a :: len word  $\Rightarrow$  'a :: len word **where**  
*bin-abs* v = (if (v < s 0) then (- v) else v)

**lemma** *val-abs-zero*:  
*intval-abs* (new-int b 0) = new-int b 0  
**by** *simp*

**lemma** *less-eq-zero*:  
**assumes** *val-to-bool* (val[(IntVal b 0) < (IntVal b v)])  
**shows** *int-signed-value* b v > 0  
**using** *assms* **unfolding** *intval-less-than.simps*(1) **apply** *simp*  
**by** (metis *bool-to-val.elims val-to-bool.simps*(1))

**lemma** *val-abs-pos*:  
**assumes** *val-to-bool*(val[(new-int b 0) < (new-int b v)])  
**shows** *intval-abs* (new-int b v) = (new-int b v)  
**using** *assms* **using** *less-eq-zero* **unfolding** *intval-abs.simps new-int.simps*  
**by** *force*

**lemma** *val-abs-neg*:  
**assumes** *val-to-bool*(val[(new-int b v) < (new-int b 0)])  
**shows** *intval-abs* (new-int b v) = *intval-negate* (new-int b v)  
**using** *assms* **using** *less-eq-zero* **unfolding** *intval-abs.simps new-int.simps*  
**by** *force*

**lemma** *val-bool-unwrap*:  
*val-to-bool* (*bool-to-val* v) = v  
**by** (metis *bool-to-val.elims one-neq-zero val-to-bool.simps*(1))

**lemma** *take-bit-unwrap*:  
b = 64  $\Rightarrow$  *take-bit* b (v1::64 word) = v1  
**by** (metis *size64 size-word.rep-eq take-bit-length-eq*)

**lemma** *bit-less-eq-def*:  
**fixes** v1 v2 :: 64 word  
**assumes** b  $\leq$  64  
**shows** *sint* (*signed-take-bit* (b - Suc (0::nat)) (*take-bit* b v1))  
< *sint* (*signed-take-bit* (b - Suc (0::nat)) (*take-bit* b v2))  $\longleftrightarrow$   
*signed-take-bit* (63::nat) (*Word.rep* v1) < *signed-take-bit* (63::nat) (*Word.rep*  
v2)  
**using** *assms* **sorry**

**lemma** *less-eq-def*:  
**shows** *val-to-bool*(val[(new-int b v1) < (new-int b v2)])  $\longleftrightarrow$  v1 < s v2  
**unfolding** *new-int.simps intval-less-than.simps bool-to-val-bin.simps bool-to-val.simps*

```

      int-signed-value.simps
    apply (simp add: val-bool-unwrap) apply auto[1]
    unfolding word-sless-def apply auto[1]
    unfolding signed-def apply auto[1] apply auto[2]
    using bit-less-eq-def apply (metis bot-nat-0.extremum take-bit-0)
    by (metis bit-less-eq-def bot-nat-0.extremum take-bit-0)

lemma val-abs-always-pos:
  assumes intval-abs (new-int b v) = (new-int b v')
  shows  $0 \leq_s v'$ 
  using assms
proof (cases v = 0)
  case True
  then have v' = 0
    using val-abs-zero assms
  by (smt (verit, ccfv-threshold) Suc-diff-1 bit-less-eq-def bot-nat-0.extremum
    diff-is-0-eq
    len-gt-0 len-of-numeral-defs(2) order-le-less signed-eq-0-iff take-bit-0
    take-bit-signed-take-bit take-bit-unwrap)
  then show ?thesis by simp
next
  case neq0: False
  then show ?thesis
  proof (cases val-to-bool(val[(new-int b 0) < (new-int b v)]))
    case True
    then show ?thesis using less-eq-def
    using assms val-abs-pos
    by (smt (verit, ccfv-SIG) One-nat-def Suc-leI bit.compl-one bit-less-eq-def
      cancel-comm-monoid-add-class.diff-cancel diff-zero len-gt-0 len-of-numeral-defs(2)
      mask-0 mask-1 one-le-numeral one-neq-zero signed-word-eqI take-bit-dist-subL
      take-bit-minus-one-eq-mask take-bit-not-eq-mask-diff take-bit-signed-take-bit
      zero-le-numeral)
  next
  case False
  then have val-to-bool(val[(new-int b v) < (new-int b 0)])
    using neq0 less-eq-def
    by (metis signed.neqE)
  then show ?thesis using val-abs-neg less-eq-def unfolding new-int.simps
    intval-negate.simps
    by (metis signed.nless-le take-bit-0)
qed
qed

```

```

lemma intval-abs-elim:
  assumes intval-abs  $x \neq \text{UndefVal}$ 
  shows  $\exists t v . x = \text{IntVal } t v \wedge \text{intval-abs } x = \text{new-int } t \text{ (if int-signed-value } t v < 0 \text{ then } -v \text{ else } v)$ 
  using assms
  by (meson intval-abs.elims)

lemma wf-abs-new-int:
  assumes intval-abs (IntVal  $t v$ )  $\neq \text{UndefVal}$ 
  shows intval-abs (IntVal  $t v$ ) = new-int  $t v \vee \text{intval-abs}$  (IntVal  $t v$ ) = new-int  $t (-v)$ 
  using assms
  using intval-abs.simps(1) by presburger

lemma mono-undef-abs:
  assumes intval-abs (intval-abs  $x$ )  $\neq \text{UndefVal}$ 
  shows intval-abs  $x \neq \text{UndefVal}$ 
  using assms
  by force

lemma val-abs-idem:
  assumes intval-abs(intval-abs( $x$ ))  $\neq \text{UndefVal}$ 
  shows intval-abs(intval-abs( $x$ )) = intval-abs  $x$ 
  using assms
proof –
  obtain  $b v$  where in-def: intval-abs  $x = \text{new-int } b v$ 
  using assms intval-abs-elim mono-undef-abs by blast
  then show ?thesis
  proof (cases val-to-bool(val[(new-int  $b v$ ) < (new-int  $b 0$ )]))
  case True
  then have nested: (intval-abs (intval-abs  $x$ )) = new-int  $b (-v)$ 
  using val-abs-neg intval-negate.simps in-def
  by simp
  then have  $x = \text{new-int } b (-v)$ 
  using in-def True unfolding new-int.simps
  by (smt (verit, best) intval-abs.simps(1) less-eq-def less-eq-zero less-numeral-extra(1)

    mask-1 mask-eq-take-bit-minus-one neg-one.elims neg-one-signed new-int.simps

    one-le-numeral one-neg-zero signed.neqE signed.not-less take-bit-of-0
val-abs-always-pos)
  then show ?thesis using val-abs-always-pos
  using True in-def less-eq-def signed.leD
  using signed.nless-le by blast
next
case False
then show ?thesis
  using in-def by force

```

qed  
qed

**lemma** *val-abs-negate*:  
**assumes** *intval-abs (intval-negate x) ≠ UndefVal*  
**shows** *intval-abs (intval-negate x) = intval-abs x*  
**using** *assms apply (cases x; auto)*  
**apply** (*metis less-eq-def new-int.simps signed.dual-order.strict-iff-not signed.less-linear*  
*take-bit-0*)  
**by** (*smt (verit, ccfv-threshold) add.inverse-neutral intval-abs.simps(1) less-eq-def*  
*less-eq-zero*  
*less-numeral-extra(1) mask-1 mask-eq-take-bit-minus-one neg-one.elims neg-one-signed*  
*new-int.simps one-le-numeral one-neq-zero signed.order.order-iff-strict take-bit-of-0*  
*val-abs-always-pos*)

Optimisations

**optimization** *AbsIdempotence*: *abs(abs(x)) ⟶ abs(x)*  
**apply** *auto[1]*  
**by** (*metis UnaryExpr unary-eval.simps(1) val-abs-idem*)

**optimization** *AbsNegate*: *(abs(−x)) ⟶ abs(x)*  
**apply** *auto[1]* **using** *val-abs-negate*  
**by** (*metis unary-eval.simps(1) unfold-unary*)

end

end

## 1.2 AddNode Phase

**theory** *AddPhase*  
**imports**  
*Common*  
**begin**

**phase** *AddNode*  
**terminating** *size*  
**begin**

**lemma** *binadd-commute*:  
**assumes** *bin-eval BinAdd x y ≠ UndefVal*  
**shows** *bin-eval BinAdd x y = bin-eval BinAdd y x*  
**using** *assms intval-add-sym* **by** *simp*



**optimization** *AddShiftConstantRight*:  $((\text{const } v) + y) \mapsto y + (\text{const } v)$  when  $\neg(\text{is-ConstantExpr } y)$   
 using *size-non-const*  
 apply (*metis add-2-eq-Suc' less-Suc-eq plus-1-eq-Suc size.simps(11) size-non-add*)  
 unfolding *le-expr-def*  
 apply (*rule impI*)  
 subgoal premises 1  
 apply (*rule allI impI*)  
 done  
 subgoal premises 2 for *m p va*  
 apply (*rule BinaryExprE[OF 2]*)  
 subgoal premises 3 for *x ya*  
 apply (*rule BinaryExpr*)  
 using 3 apply *simp*  
 using 3 apply *simp*  
 using 3 *binadd-commute* apply *auto*  
 done  
 done  
 done  
 done

**optimization** *AddShiftConstantRight2*:  $((\text{const } v) + y) \mapsto y + (\text{const } v)$  when  $\neg(\text{is-ConstantExpr } y)$   
 unfolding *le-expr-def*  
 apply (*auto simp: intval-add-sym*)  
 using *size-non-const*  
 by (*metis add-2-eq-Suc' lessI plus-1-eq-Suc size.simps(11) size-non-add*)

**lemma** *is-neutral-0* [*simp*]:  
 assumes 1: *intval-add (IntVal b x) (IntVal b 0)  $\neq$  UndefVal*  
 shows *intval-add (IntVal b x) (IntVal b 0) = (new-int b x)*  
 using 1 by *auto*

**optimization** *AddNeutral*:  $(e + (\text{const } (\text{IntVal } 32 \ 0))) \mapsto e$   
 unfolding *le-expr-def* apply *auto*  
 using *is-neutral-0 eval-unused-bits-zero*  
 by (*smt (verit) add-cancel-left-right intval-add.elims val-to-bool.simps(1)*)

**ML-val**  $\langle @\{term \ \langle x = y \rangle\} \rangle$

**lemma** *NeutralLeftSubVal*:  
 assumes *e1 = new-int b ival*

**shows**  $val[(e1 - e2) + e2] \approx e1$   
**apply** *simp* **using** *assms* **by** (*cases e1*; *cases e2*; *auto*)

**optimization** *RedundantSubAdd*:  $((e_1 - e_2) + e_2) \mapsto e_1$   
**apply** *auto* **using** *eval-unused-bits-zero* *NeutralLeftSubVal*  
**unfolding** *well-formed-equal-defn*  
**by** (*smt* (*verit*) *evalDet* *intval-sub.elims* *new-int.elims*)

**lemma** *allE2*:  $(\forall x y. P x y) \implies (P a b \implies R) \implies R$   
**by** *simp*

**lemma** *just-goal2*:  
**assumes** *1*:  $(\forall a b. (intval-add (intval-sub a b) b \neq UndefinedVal \wedge a \neq UndefinedVal$   
 $\longrightarrow$   
 $intval-add (intval-sub a b) b = a))$   
**shows**  $(BinaryExpr BinAdd (BinaryExpr BinSub e_1 e_2) e_2) \geq e_1$   
**unfolding** *le-expr-def* *unfold-binary* *bin-eval.simps*  
**by** (*metis 1 evalDet evaltree-not-undef*)

**optimization** *RedundantSubAdd2*:  $e_2 + (e_1 - e_2) \mapsto e_1$   
**apply** (*metis add.commute add-less-cancel-right less-add-Suc2 plus-1-eq-Suc size-binary-const*  
*size-non-add trans-less-add2*)  
**by** (*smt* (*verit*, *del-insts*) *BinaryExpr BinaryExprE RedundantSubAdd(1) bi-*  
*nadd-commute le-expr-def rewrite-preservation.simps(1)*)

**lemma** *AddToSubHelperLowLevel*:  
**shows**  $intval-add (intval-negate e) y = intval-sub y e$  (*is ?x = ?y*)  
**by** (*induction y*; *induction e*; *auto*)

**print-phases**

**lemma** *val-redundant-add-sub*:  
**assumes**  $a = new-int bb\ ival$   
**assumes**  $val[b + a] \neq UndefinedVal$   
**shows**  $val[(b + a) - b] = a$

```

using assms apply (cases a; cases b; auto)
by presburger

```

```

lemma val-add-right-negate-to-sub:
  assumes  $\text{val}[x + e] \neq \text{UndefVal}$ 
  shows  $\text{val}[x + (-e)] = \text{val}[x - e]$ 
  using assms by (cases x; cases e; auto)

```

```

lemma exp-add-left-negate-to-sub:
   $\text{exp}[-e + y] \geq \text{exp}[y - e]$ 
  apply (cases e; cases y; auto)
  using AddToSubHelperLowLevel by auto

```

Optimisations

```

optimization RedundantAddSub:  $(b + a) - b \mapsto a$ 
  apply auto
  by (smt (verit) evalDet intval-add.elims new-int.elims val-redundant-add-sub
    eval-unused-bits-zero)

```

```

optimization AddRightNegateToSub:  $x + -e \mapsto x - e$ 
  apply (metis Nat.add-0-right add-2-eq-Suc' add-less-mono1 add-mono-thms-linordered-field(2)
    less-SucI not-less-less-Suc-eq size-binary-const size-non-add size-pos)
  using AddToSubHelperLowLevel intval-add-sym by auto

```

```

optimization AddLeftNegateToSub:  $-e + y \mapsto y - e$ 
  apply (smt (verit, best) One-nat-def add commute add-Suc-right is-ConstantExpr-def
    less-add-Suc2
    numeral-2-eq-2 plus-1-eq-Suc size.simps(1) size.simps(11) size-binary-const
    size-non-add)
  using exp-add-left-negate-to-sub by blast

```

**end**

**end**

### 1.3 AndNode Phase

```

theory AndPhase
  imports
    Common
    Proofs.StampEvalThms
begin

```

**context** *stamp-mask*

**begin**

**lemma** *AndRightFallthrough*:  $((\text{and } (\text{not } (\downarrow x)) (\uparrow y)) = 0) \longrightarrow \text{exp}[x \ \& \ y] \geq \text{exp}[y]$

**apply** *simp* **apply** (*rule impI*; (*rule allI*)+)  
**apply** (*rule impI*)  
**subgoal** *premises p* **for** *m p v*  
**proof** –  
  **obtain** *xv* **where** *xv*:  $[m, p] \vdash x \mapsto xv$   
  **using** *p(2)* **by** *blast*  
  **obtain** *yv* **where** *yv*:  $[m, p] \vdash y \mapsto yv$   
  **using** *p(2)* **by** *blast*  
  **have**  $v = \text{val}[xv \ \& \ yv]$   
  **by** (*metis BinaryExprE bin-eval.simps(4) evalDet p(2) xv yv*)  
  **then have**  $v = yv$   
  **by** (*smt (verit) eval-unused-bits-zero intval-and.elims new-int.elims new-int-bin.elims*  
*p(2)*  
   *unfold-binary xv yv p(1) not-down-up-mask-and-zero-implies-zero*)  
  **then show** *?thesis* **using** *yv* **by** *simp*  
**qed**  
**done**

**lemma** *AndLeftFallthrough*:  $((\text{and } (\text{not } (\downarrow y)) (\uparrow x)) = 0) \longrightarrow \text{exp}[x \ \& \ y] \geq \text{exp}[x]$

**apply** *simp* **apply** (*rule impI*; (*rule allI*)+)  
**apply** (*rule impI*)  
**subgoal** *premises p* **for** *m p v*  
**proof** –  
  **obtain** *xv* **where** *xv*:  $[m, p] \vdash x \mapsto xv$   
  **using** *p(2)* **by** *blast*  
  **obtain** *yv* **where** *yv*:  $[m, p] \vdash y \mapsto yv$   
  **using** *p(2)* **by** *blast*  
  **have**  $v = \text{val}[xv \ \& \ yv]$   
  **by** (*metis BinaryExprE bin-eval.simps(4) evalDet p(2) xv yv*)  
  **then have**  $v = xv$   
  **by** (*smt (verit) and.commute eval-unused-bits-zero intval-and.elims new-int.simps*  
   *new-int-bin.simps p(2) unfold-binary xv yv p(1) not-down-up-mask-and-zero-implies-zero*)  
  **then show** *?thesis* **using** *xv* **by** *simp*  
**qed**  
**done**  
**end**

**phase** *AndNode*

**terminating** *size*

**begin**

**lemma** *bin-and-nots*:  
 $(\sim x \ \& \ \sim y) = (\sim (x \mid y))$   
**by** *simp*

**lemma** *bin-and-neutral*:  
 $(x \ \& \ \sim \text{False}) = x$   
**by** *simp*

**lemma** *val-and-equal*:  
**assumes**  $x = \text{new-int } b \ v$   
**and**  $\text{val}[x \ \& \ x] \neq \text{UndefVal}$   
**shows**  $\text{val}[x \ \& \ x] = x$   
**using** *assms* **by** (*cases*  $x$ ; *auto*)

**lemma** *val-and-nots*:  
 $\text{val}[\sim x \ \& \ \sim y] = \text{val}[\sim (x \mid y)]$   
**apply** (*cases*  $x$ ; *cases*  $y$ ; *auto*) **by** (*simp* *add: take-bit-not-take-bit*)

**lemma** *val-and-neutral*:  
**assumes**  $x = \text{new-int } b \ v$   
**and**  $\text{val}[x \ \& \ \sim (\text{new-int } b' \ 0)] \neq \text{UndefVal}$   
**shows**  $\text{val}[x \ \& \ \sim (\text{new-int } b' \ 0)] = x$   
**using** *assms* **apply** (*cases*  $x$ ; *auto*) **apply** (*simp* *add: take-bit-eq-mask*)  
**by** *presburger*

**lemma** *val-and-zero*:  
**assumes**  $x = \text{new-int } b \ v$   
**shows**  $\text{val}[x \ \& \ (\text{IntVal } b \ 0)] = \text{IntVal } b \ 0$   
**using** *assms* **by** (*cases*  $x$ ; *auto*)

**lemma** *exp-and-equal*:  
 $\text{exp}[x \ \& \ x] \geq \text{exp}[x]$   
**apply** *auto*  
**by** (*smt* (*verit*) *evalDet intval-and.elims new-int.elims val-and-equal eval-unused-bits-zero*)

**lemma** *exp-and-nots*:  
 $\text{exp}[\sim x \ \& \ \sim y] \geq \text{exp}[\sim (x \mid y)]$   
**apply** (*cases*  $x$ ; *cases*  $y$ ; *auto*) **using** *val-and-nots*  
**by** *fastforce+*

**lemma** *exp-sign-extend*:  
**assumes**  $e = (1 \ll In) - 1$

```

shows   BinaryExpr BinAnd (UnaryExpr (UnarySignExtend In Out) x)
        (ConstantExpr (new-int b e))
        ≥ (UnaryExpr (UnaryZeroExtend In Out) x)

apply auto
subgoal premises p for m p va
proof -
  obtain va where va: [m,p] ⊢ x ↦ va
  using p(2) by auto
  then have va ≠ UndefVal
  by (simp add: evaltree-not-undef)
  then have 1: intval-and (intval-sign-extend In Out va) (IntVal b (take-bit b
e)) ≠ UndefVal
  using evalDet p(1) p(2) va by blast
  then have 2: intval-sign-extend In Out va ≠ UndefVal
  by auto
  then have 21: (0::nat) < b
  using eval-bits-1-64 p(4) by blast
  then have 3: b ⊆ (64::nat)
  using eval-bits-1-64 p(4) by blast
  then have 4: - ((2::int) ^ b div (2::int)) ⊆ sint (signed-take-bit (b - Suc
(0::nat)) (take-bit b e))
  by (simp add: 21 int-power-div-base signed-take-bit-int-greater-eq-minus-exp-word)
  then have 5: sint (signed-take-bit (b - Suc (0::nat)) (take-bit b e)) < (2::int)
^ b div (2::int)
  by (simp add: 21 3 Suc-le-lessD int-power-div-base signed-take-bit-int-less-exp-word)
  then have 6: [m,p] ⊢ UnaryExpr (UnaryZeroExtend In Out)
    x ↦ intval-and (intval-sign-extend In Out va) (IntVal b (take-bit b e))
  apply (cases va; simp)
  apply (simp add: ⟨(va::Value) ≠ UndefVal⟩) defer
  subgoal premises p for x3
  proof -
    have va = ObjRef x3
    using p(1) by auto
    then have sint (signed-take-bit (b - Suc (0::nat)) (take-bit b e)) <
(2::int) ^ b div (2::int)
    by (simp add: 5)
    then show ?thesis
    using 2 intval-sign-extend.simps(3) p(1) by blast
  qed

subgoal premises p for x4
proof -
  have sg1: va = ObjStr x4
  using 2 p(1) by auto
  then have sint (signed-take-bit (b - Suc (0::nat)) (take-bit b e)) <
(2::int) ^ b div (2::int)
  by (simp add: 5)
  then show ?thesis
  using 1 sg1 by auto

```

qed

```

subgoal premises p for x21 x22
proof -
  have sgg1: va = IntVal x21 x22
  by (simp add: p(1))
  then have sgg2: sint (signed-take-bit (b - Suc (0::nat)) (take-bit b e))
    < (2::int) ^ b div (2::int)
  by (simp add: 5)
  then show ?thesis
  sorry
qed
done
then show ?thesis
by (metis evalDet p(2) va)
qed
done

```

```

lemma val-and-commute[simp]:
  val[x & y] = val[y & x]
  apply (cases x; cases y; auto)
  by (simp add: word-bw-comms(1))

```

Optimisations

```

optimization AndEqual: x & x ⟶ x
using exp-and-equal by blast

```

```

optimization AndShiftConstantRight: ((const x) & y) ⟶ y & (const x)
  when ¬(is-ConstantExpr y)
using size-flip-binary by auto

```

```

optimization AndNots: (~x) & (~y) ⟶ ~(x | y)
  apply (metis add-2-eq-Suc' less-SucI less-add-Suc1 not-less-eq size-binary-const
    size-non-add)
  using exp-and-nots by auto

```

```

optimization AndSignExtend: BinaryExpr BinAnd (UnaryExpr (UnarySignExtend
  In Out) (x))
  ⟶ (const (new-int b e))
  ⟶ (UnaryExpr (UnaryZeroExtend In Out) (x))
  when (e = (1 << In) - 1)
using exp-sign-extend by simp

```

**optimization** *AndNeutral*:  $(x \& \sim(\text{const } (\text{IntVal } b \ 0))) \mapsto x$   
 when  $(\text{wf-stamp } x \wedge \text{stamp-expr } x = \text{IntegerStamp } b \ \text{lo } \text{hi})$   
**apply** *auto*  
**by**  $(\text{smt } (\text{verit}) \ \text{Value.sel}(1) \ \text{eval-unused-bits-zero } \text{intval-and.elims } \text{intval-word.simps}$   
 $\text{new-int.simps } \text{new-int-bin.simps } \text{take-bit-eq-mask})$

**optimization** *AndRightFallThrough*:  $(x \& y) \mapsto y$   
 when  $((\text{and } (\text{not } (\text{IRExpr-down } x)) (\text{IRExpr-up } y)) = 0)$   
**by**  $(\text{simp add: IRExpr-down-def IRExpr-up-def})$

**optimization** *AndLeftFallThrough*:  $(x \& y) \mapsto x$   
 when  $((\text{and } (\text{not } (\text{IRExpr-down } y)) (\text{IRExpr-up } x)) = 0)$   
**by**  $(\text{simp add: IRExpr-down-def IRExpr-up-def})$

**end**

**end**

## 1.4 BinaryNode Phase

**theory** *BinaryNode*  
**imports**  
*Common*  
**begin**

**phase** *BinaryNode*  
**terminating** *size*  
**begin**

**optimization** *BinaryFoldConstant*:  $\text{BinaryExpr } op \ (\text{const } v1) \ (\text{const } v2) \mapsto \text{ConstantExpr } (\text{bin-eval } op \ v1 \ v2)$   
**unfolding** *le-expr-def*  
**apply**  $(\text{rule allI impI})+$   
**subgoal premises** *bin* **for** *m p v*  
**print-facts**  
**apply**  $(\text{rule } \text{BinaryExprE}[OF \ \text{bin}])$   
**subgoal premises** *prems* **for** *x y*  
**print-facts**

**proof** —

**have** *x*:  $x = v1$  **using** *prems* **by** *auto*  
**have** *y*:  $y = v2$  **using** *prems* **by** *auto*  
**have** *xy*:  $v = \text{bin-eval } op \ x \ y$  **using** *prems x y* **by** *simp*  
**have** *int*:  $\exists \ b \ vv . v = \text{new-int } b \ vv$  **using** *bin-eval-new-int prems* **by** *fast*



```

    show ?thesis
    unfolding prems x y xy
    apply (rule ConstantExpr)
    using prems x y xy int sorry
  qed
done
done

```

```
print-facts
```

```
end
```

```
end
```

## 1.5 ConditionalNode Phase

```

theory ConditionalPhase
  imports
    Common
    Proofs.StampEvalThms
begin

```

```

phase ConditionalNode
  terminating size
begin

```

```

lemma negates:  $\exists v b. e = \text{IntVal } b \ v \wedge b > 0 \implies \text{val-to-bool } (\text{val}[e]) \longleftrightarrow$ 
 $\neg(\text{val-to-bool } (\text{val}[\neg e]))$ 
  unfolding intval-logic-negation.simps
  by (metis (mono-tags, lifting) intval-logic-negation.simps(1) logic-negate-def new-int.simps
    of-bool-eq(2) one-neq-zero take-bit-of-0 take-bit-of-1 val-to-bool.simps(1))

```

```

lemma negation-condition-intval:
  assumes  $e = \text{IntVal } b \ ie$ 
  assumes  $0 < b$ 
  shows  $\text{val}[(\neg e) \ ? \ x : y] = \text{val}[e \ ? \ y : x]$ 
  using assms by (cases e; auto simp: negates logic-negate-def)

```

```

lemma negation-preserve-eval:
  assumes  $[m, p] \vdash \text{exp}[\neg e] \mapsto v$ 
  shows  $\exists v'. ([m, p] \vdash \text{exp}[e] \mapsto v') \wedge v = \text{val}[\neg v']$ 
  using assms by auto

```

```

lemma negation-preserve-eval-intval:
  assumes  $[m, p] \vdash \text{exp}[\neg e] \mapsto v$ 
  shows  $\exists v' b \ vv. ([m, p] \vdash \text{exp}[e] \mapsto v') \wedge v' = \text{IntVal } b \ vv \wedge b > 0$ 
  using assms
  by (metis eval-bits-1-64 intval-logic-negation.elims negation-preserve-eval unfold-unary)

```

**optimization** *NegateConditionFlipBranches*:  $((!e) \text{ ? } x : y) \mapsto (e \text{ ? } y : x)$   
**apply** *simp using* *negation-condition-intval negation-preserve-eval-intval*  
**by** (*smt* (*verit*, *best*) *ConditionalExpr ConditionalExprE Value.distinct(1) evalDet*  
*negates negation-preserve-eval*)

**optimization** *DefaultTrueBranch*:  $(\text{true} \text{ ? } x : y) \mapsto x$  .

**optimization** *DefaultFalseBranch*:  $(\text{false} \text{ ? } x : y) \mapsto y$  .

**optimization** *ConditionalEqualBranches*:  $(e \text{ ? } x : x) \mapsto x$  .

**optimization** *condition-bounds-x*:  $((u < v) \text{ ? } x : y) \mapsto x$   
*when* (*stamp-under* (*stamp-expr* *u*) (*stamp-expr* *v*)  $\wedge$  *wf-stamp* *u*  $\wedge$  *wf-stamp* *v*)  
**using** *stamp-under-defn* **by** *fastforce*

**optimization** *condition-bounds-y*:  $((u < v) \text{ ? } x : y) \mapsto y$   
*when* (*stamp-under* (*stamp-expr* *v*) (*stamp-expr* *u*)  $\wedge$  *wf-stamp* *u*  $\wedge$  *wf-stamp* *v*)  
**using** *stamp-under-defn-inverse* **by** *fastforce*

**lemma** *val-optimize-integer-test*:  
**assumes**  $\exists v. x = \text{IntVal } 32 \ v$   
**shows**  $\text{val}[(x \ \& \ (\text{IntVal } 32 \ 1)) \ \text{eq} \ (\text{IntVal } 32 \ 0)] \text{ ? } (\text{IntVal } 32 \ 0) : (\text{IntVal } 32 \ 1)] =$   
 $\text{val}[x \ \& \ \text{IntVal } 32 \ 1]$   
**using** *assms* **apply** *auto*  
**apply** (*metis* (*full-types*) *bool-to-val.simps(2) val-to-bool.simps(1)*)  
**by** (*metis* (*mono-tags, lifting*) *and-one-eq bool-to-val.simps(1) even-iff-mod-2-eq-zero*  
*odd-iff-mod-2-eq-one val-to-bool.simps(1)*)

**optimization** *ConditionalEliminateKnownLess*:  $((x < y) \text{ ? } x : y) \mapsto x$   
*when* (*stamp-under* (*stamp-expr* *x*) (*stamp-expr* *y*)  
 $\wedge$  *wf-stamp* *x*  $\wedge$  *wf-stamp* *y*)  
**using** *stamp-under-defn* **by** *fastforce*

**optimization** *ConditionalEqualIsRHS*:  $((x \text{ eq } y) \text{ ? } x : y) \mapsto y$   
**apply** *auto*  
**by** (*smt* (*verit*) *Value.inject(1) bool-to-val.simps(2) bool-to-val-bin.simps evalDet*  
*intval-equals.elims val-to-bool.elims(1)*)

**optimization** *normalizeX*:  $((x \text{ eq } \text{const } (\text{IntVal } 32 \ 0)) \text{ ? } (\text{const } (\text{IntVal } 32 \ 0)) : (\text{const } (\text{IntVal } 32 \ 1))) \mapsto x$   
*when* (*IRExpr-up* *x* = 1)  $\wedge$  *stamp-expr* *x* = *IntegerStamp*  
*b 0 1*

```

apply auto
subgoal premises p for m p v xa
proof —
  obtain xa where xa:  $[m, p] \vdash x \mapsto xa$ 
  using p by blast
  have  $\exists$ :  $[m, p] \vdash \text{if } \text{val-to-bool } (\text{intval-equals } xa \ (\text{IntVal } (32::\text{nat}) \ (0::64 \text{ word})))$ 
    then ConstantExpr (IntVal (32::nat) (0::64 word))
    else ConstantExpr (IntVal (32::nat) (1::64 word))  $\mapsto v$ 
  using evalDet p( $\exists$ ) p(5) xa
  using p(4) p(6) by blast
  then have  $\exists$ : xa = IntVal 32 0 | xa = IntVal 32 1
  sorry
  then have  $\exists$ : v = xa
  sorry
  then show ?thesis
  using xa by auto
qed
done

```

```

optimization normalizeX2:  $((x \text{ eq } (\text{const } (\text{IntVal } 32 \ 1))) \ ?$ 
   $(\text{const } (\text{IntVal } 32 \ 1)) : (\text{const } (\text{IntVal } 32 \ 0))) \mapsto x$ 
  when  $(x = \text{ConstantExpr } (\text{IntVal } 32 \ 0) \mid (x =$ 
ConstantExpr (IntVal 32 1))) .

```

```

optimization flipX:  $((x \text{ eq } (\text{const } (\text{IntVal } 32 \ 0))) \ ?$ 
   $(\text{const } (\text{IntVal } 32 \ 1)) : (\text{const } (\text{IntVal } 32 \ 0))) \mapsto$ 
   $x \oplus (\text{const } (\text{IntVal } 32 \ 1))$ 
  when  $(x = \text{ConstantExpr } (\text{IntVal } 32 \ 0) \mid (x = \text{ConstantExpr}$ 
(IntVal 32 1))) .

```

```

optimization flipX2:  $((x \text{ eq } (\text{const } (\text{IntVal } 32 \ 1))) \ ?$ 
   $(\text{const } (\text{IntVal } 32 \ 0)) : (\text{const } (\text{IntVal } 32 \ 1))) \mapsto$ 
   $x \oplus (\text{const } (\text{IntVal } 32 \ 1))$ 
  when  $(x = \text{ConstantExpr } (\text{IntVal } 32 \ 0) \mid (x = \text{ConstantExpr}$ 
(IntVal 32 1))) .

```

```

lemma stamp-of-default:
  assumes stamp-expr x = default-stamp
  assumes wf-stamp x
  shows  $([m, p] \vdash x \mapsto v) \longrightarrow (\exists \ vv. \ v = \text{IntVal } 32 \ vv)$ 
  using assms
  by (metis default-stamp valid-value-elim( $\exists$ ) wf-stamp-def)

```

```

optimization OptimiseIntegerTest:
   $((x \ \& \ (\text{const } (\text{IntVal } 32 \ 1))) \text{ eq } (\text{const } (\text{IntVal } 32 \ 0))) \ ?$ 
   $(\text{const } (\text{IntVal } 32 \ 0)) : (\text{const } (\text{IntVal } 32 \ 1))) \mapsto$ 

```

```

      x & (const (IntVal 32 1))
      when (stamp-expr x = default-stamp ∧ wf-stamp x)
apply simp apply (rule impI; (rule allI)+; rule impI)
subgoal premises eval for m p v
proof –
  obtain xv where xv: [m, p] ⊢ x ↦ xv
  using eval by fast
  then have x32: ∃ v. xv = IntVal 32 v
  using stamp-of-default eval by auto
  obtain lhs where lhs: [m, p] ⊢ exp[(((x & (const (IntVal 32 1)))) eq (const (IntVal
32 0)))) ?
    (const (IntVal 32 0)) : (const (IntVal 32 1))]] ↦ lhs
  using eval(2) by auto
  then have lhsV: lhs = val[((xv & (IntVal 32 1)) eq (IntVal 32 0)) ? (IntVal 32
0) : (IntVal 32 1)]
  using xv evaltree.BinaryExpr evaltree.ConstantExpr evaltree.ConditionalExpr
  by (smt (verit) ConditionalExprE ConstantExprE bin-eval.simps(11) bin-eval.simps(4)
evalDet intval-conditional.simps unfold-binary)
  obtain rhs where rhs: [m, p] ⊢ exp[x & (const (IntVal 32 1))] ↦ rhs
  using eval(2) by blast
  then have rhsV: rhs = val[xv & IntVal 32 1]
  by (metis BinaryExprE ConstantExprE bin-eval.simps(4) evalDet xv)
  have lhs = rhs using val-optimise-integer-test x32
  using lhsV rhsV by presburger
  then show ?thesis
  by (metis eval(2) evalDet lhs rhs)
qed
done

```

```

optimization opt-optimise-integer-test-2:
  (((x & (const (IntVal 32 1))) eq (const (IntVal 32 0)))) ?
    (const (IntVal 32 0)) : (const (IntVal 32 1)))] ↦→
    x
  when (x = ConstantExpr (IntVal 32 0) | (x = ConstantExpr (IntVal
32 1))) .

```

**end**

**end**

## 1.6 MulNode Phase

**theory** *MulPhase*

**imports**

*Common*

*Proofs.StampEvalThms*

**begin**

**fun** *mul-size* :: *IRExpr*  $\Rightarrow$  *nat* **where**

*mul-size* (*UnaryExpr op e*) = (*mul-size e*) + 2 |

*mul-size* (*BinaryExpr BinMul x y*) = ((*mul-size x*) + (*mul-size y*) + 2) \* 2 |

*mul-size* (*BinaryExpr op x y*) = (*mul-size x*) + (*mul-size y*) + 2 |

*mul-size* (*ConditionalExpr cond t f*) = (*mul-size cond*) + (*mul-size t*) + (*mul-size f*) + 2 |

*mul-size* (*ConstantExpr c*) = 1 |

*mul-size* (*ParameterExpr ind s*) = 2 |

*mul-size* (*LeafExpr nid s*) = 2 |

*mul-size* (*ConstantVar c*) = 2 |

*mul-size* (*VariableExpr x s*) = 2

**phase** *MulNode*

**terminating** *mul-size*

**begin**

**lemma** *bin-eliminate-redundant-negative*:

*uminus* (*x* :: '*a*::*len word*') \* *uminus* (*y* :: '*a*::*len word*') = *x* \* *y*

**by** *simp*

**lemma** *bin-multiply-identity*:

(*x* :: '*a*::*len word*') \* 1 = *x*

**by** *simp*

**lemma** *bin-multiply-eliminate*:

(*x* :: '*a*::*len word*') \* 0 = 0

**by** *simp*

**lemma** *bin-multiply-negative*:

(*x* :: '*a*::*len word*') \* *uminus* 1 = *uminus x*

**by** *simp*

**lemma** *bin-multiply-power-2*:

(*x* :: '*a*::*len word*') \* ( $2^j$ ) = *x* << *j*

**by** *simp*

**lemma** *take-bit64* [*simp*]:

**fixes** *w* :: *int64*

**shows** *take-bit 64 w* = *w*

**proof** —

```

have Nat.size w = 64
  by (simp add: size64)
then show ?thesis
  by (metis lt2p-lem mask-eq-iff take-bit-eq-mask verit-comp-simplify1 (2) wsst-TYs(3))
qed

```

```

lemma mergeTakeBit:
  fixes a :: nat
  fixes b c :: 64 word
  shows take-bit a (take-bit a (b) * take-bit a (c)) =
    take-bit a (b * c)
by (smt (verit, ccfv-SIG) take-bit-mult take-bit-of-int unsigned-take-bit-eq word-mult-def)

```

```

lemma val-eliminate-redundant-negative:
  assumes val[-x * -y] ≠ UndefVal
  shows val[-x * -y] = val[x * y]
  using assms apply (cases x; cases y; auto)
  using mergeTakeBit by auto

```

```

lemma val-multiply-neutral:
  assumes x = new-int b v
  shows val[x * (IntVal b 1)] = val[x]
  using assms by force

```

```

lemma val-multiply-zero:
  assumes x = new-int b v
  shows val[x * (IntVal b 0)] = IntVal b 0
  using assms by simp

```

```

lemma val-multiply-negative:
  assumes x = new-int b v
  shows val[x * intval-negate (IntVal b 1)] = intval-negate x
  by (smt (verit) Value.disc(1) Value.inject(1) add.inverse-neutral intval-negate.simps(1))

```

```

  is-IntVal-def mask-0 mask-eq-take-bit-minus-one new-int.elims of-bool-eq(2)
take-bit-dist-neg
take-bit-of-1 val-eliminate-redundant-negative val-multiply-neutral val-multiply-zero

verit-minus-simplify(4) zero-neq-one assms)

```

```

lemma val-MulPower2:
  fixes i :: 64 word
  assumes y = IntVal 64 (2 ^ unat(i))
  and 0 < i

```

```

and      i < 64
and      val[x * y] ≠ UndefVal
shows    val[x * y] = val[x << IntVal 64 i]
using    assms apply (cases x; cases y; auto)
subgoal premises p for x2
proof -
  have 63: (63 :: int64) = mask 6
  by eval
  then have (2::int) ^ 6 = 64
  by eval
  then have uint i < (2::int) ^ 6
  by (metis linorder-not-less lt2p-lem of-int-numeral p(4) size64 word-2p-lem
word-of-int-2p
      wsst-TYs(3))
  then have and i (mask 6) = i
  using mask-eq-iff by blast
  then show x2 << unat i = x2 << unat (and i (63::64 word))
  unfolding 63
  by force
qed
by presburger

```

```

lemma val-MulPower2Add1:
  fixes i :: 64 word
  assumes y = IntVal 64 ((2 ^ unat(i)) + 1)
  and      0 < i
  and      i < 64
  and      val-to-bool(val[IntVal 64 0 < x])
  and      val-to-bool(val[IntVal 64 0 < y])
  shows    val[x * y] = val[(x << IntVal 64 i) + x]
  using    assms apply (cases x; cases y; auto)
  subgoal premises p for x2
  proof -
    have 63: (63 :: int64) = mask 6
    by eval
    then have (2::int) ^ 6 = 64
    by eval
    then have and i (mask 6) = i
    using mask-eq-iff by (simp add: less-mask-eq p(6))
    then have x2 * ((2::64 word) ^ unat i + (1::64 word)) = (x2 * ((2::64 word)
^ unat i)) + x2
    by (simp add: distrib-left)
    then show x2 * ((2::64 word) ^ unat i + (1::64 word)) = x2 << unat (and i
(63::64 word)) + x2
    by (simp add: 63 ‹and (i::64 word) (mask (6::nat)) = i›)
  qed
  using val-to-bool.simps(2) by presburger

```

```

lemma val-MulPower2Sub1:
  fixes  $i :: 64 \text{ word}$ 
  assumes  $y = \text{IntVal } 64 \ ((2 \wedge \text{unat}(i)) - 1)$ 
  and  $0 < i$ 
  and  $i < 64$ 
  and  $\text{val-to-bool}(\text{val}[\text{IntVal } 64 \ 0 < x])$ 
  and  $\text{val-to-bool}(\text{val}[\text{IntVal } 64 \ 0 < y])$ 
  shows  $\text{val}[x * y] = \text{val}[(x << \text{IntVal } 64 \ i) - x]$ 
  using assms apply (cases  $x$ ; cases  $y$ ; auto)
  subgoal premises  $p$  for  $x2$ 
  proof -
    have  $63: (63 :: \text{int}64) = \text{mask } 6$ 
    by eval
    then have  $(2 :: \text{int}) \wedge 6 = 64$ 
    by eval
    then have  $\text{and } i (\text{mask } 6) = i$ 
    using mask-eq-iff by (simp add: less-mask-eq  $p(6)$ )
    then have  $x2 * ((2 :: 64 \text{ word}) \wedge \text{unat } i - (1 :: 64 \text{ word})) = (x2 * ((2 :: 64 \text{ word}) \wedge \text{unat } i)) - x2$ 
    by (simp add: right-diff-distrib')
    then show  $x2 * ((2 :: 64 \text{ word}) \wedge \text{unat } i - (1 :: 64 \text{ word})) = x2 << \text{unat } (\text{and } i (63 :: 64 \text{ word})) - x2$ 
    by (simp add: 63 and (i::64 word) (mask (6::nat)) = i)
    qed
    using val-to-bool.simps(2) by presburger

```

```

lemma val-distribute-multiplication:
  assumes  $x = \text{new-int } 64 \ xx \wedge q = \text{new-int } 64 \ qq \wedge a = \text{new-int } 64 \ aa$ 
  shows  $\text{val}[x * (q + a)] = \text{val}[(x * q) + (x * a)]$ 
  apply (cases  $x$ ; cases  $q$ ; cases  $a$ ; auto) using distrib-left assms by auto

```

```

lemma val-MulPower2AddPower2:
  fixes  $i \ j :: 64 \text{ word}$ 
  assumes  $y = \text{IntVal } 64 \ ((2 \wedge \text{unat}(i)) + (2 \wedge \text{unat}(j)))$ 
  and  $0 < i$ 
  and  $0 < j$ 
  and  $i < 64$ 
  and  $j < 64$ 
  and  $x = \text{new-int } 64 \ xx$ 
  shows  $\text{val}[x * y] = \text{val}[(x << \text{IntVal } 64 \ i) + (x << \text{IntVal } 64 \ j)]$ 
  using assms
  proof -
    have  $63: (63 :: \text{int}64) = \text{mask } 6$ 
    by eval
    then have  $(2 :: \text{int}) \wedge 6 = 64$ 

```



```

    by eval
  then have n: IntVal 64 ((2 ^ unat(i)) + (2 ^ unat(j))) =
    val[(IntVal 64 (2 ^ unat(i))) + (IntVal 64 (2 ^ unat(j)))]

    using assms by (cases i; cases j; auto)
  then have 1: val[x * ((IntVal 64 (2 ^ unat(i))) + (IntVal 64 (2 ^ unat(j))))]
=
    val[(x * IntVal 64 (2 ^ unat(i))) + (x * IntVal 64 (2 ^ unat(j)))]

    using assms val-distribute-multiplication val-MulPower2 by simp
  then have 2: val[(x * IntVal 64 (2 ^ unat(i)))] = val[x << IntVal 64 i]
  by (smt (verit) Value.distinct(1) intval-mul.simps(1) new-int.simps new-int-bin.simps
    assms
      val-MulPower2)
  then show ?thesis
    by (smt (verit, del-insts) 1 Value.distinct(1) assms(1) assms(3) assms(5)
      assms(6)
        intval-mul.simps(1) n new-int.simps new-int-bin.elims val-MulPower2)
  qed

```

**thm-oracles** *val-MulPower2AddPower2*

```

lemma exp-multiply-zero-64:
  exp[x * (const (IntVal 64 0))] ≥ ConstantExpr (IntVal 64 0)
  using val-multiply-zero apply auto
  by (smt (verit) Value.inject(1) constantAsStamp.simps(1) int-signed-value-bounds
    intval-mul.elims
      mult-zero-right new-int.simps new-int-bin.simps nle-le numeral-eq-Suc take-bit-of-0

      unfold-const valid-stamp.simps(1) valid-value.simps(1) zero-less-Suc wf-value-def)

```

```

lemma exp-multiply-neutral:
  exp[x * (const (IntVal b 1))] ≥ x
  using val-multiply-neutral apply auto
  by (smt (verit) Value.inject(1) eval-unused-bits-zero intval-mul.elims mult.right-neutral

      new-int.elims new-int-bin.elims)

```

**thm-oracles** *exp-multiply-neutral*

```

lemma exp-MulPower2:
  fixes i :: 64 word
  assumes y = ConstantExpr (IntVal 64 (2 ^ unat(i)))
  and 0 < i
  and i < 64
  and exp[x > (const IntVal b 0)]
  and exp[y > (const IntVal b 0)]
  shows exp[x * y] ≥ exp[x << ConstantExpr (IntVal 64 i)]

```

```

    using assms apply simp
  by (metis ConstantExprE equiv-exprs-def unfold-binary)

lemma exp-MulPower2Add1:
  fixes i :: 64 word
  assumes y = ConstantExpr (IntVal 64 ((2 ^ unat(i)) + 1))
  and 0 < i
  and i < 64
  and exp[x > (const IntVal b 0)]
  and exp[y > (const IntVal b 0)]
shows exp[x * y] ≥ exp[(x << ConstantExpr (IntVal 64 i)) + x]
  using assms apply simp
  by (metis (no-types, lifting) ConstantExprE equiv-exprs-def unfold-binary)

lemma exp-MulPower2Sub1:
  fixes i :: 64 word
  assumes y = ConstantExpr (IntVal 64 ((2 ^ unat(i)) - 1))
  and 0 < i
  and i < 64
  and exp[x > (const IntVal b 0)]
  and exp[y > (const IntVal b 0)]
shows exp[x * y] ≥ exp[(x << ConstantExpr (IntVal 64 i)) - x]
  using assms apply simp
  by (metis (no-types, lifting) ConstantExprE equiv-exprs-def unfold-binary)

lemma exp-MulPower2AddPower2:
  fixes i j :: 64 word
  assumes y = ConstantExpr (IntVal 64 ((2 ^ unat(i)) + (2 ^ unat(j))))
  and 0 < i
  and 0 < j
  and i < 64
  and j < 64
  and exp[x > (const IntVal b 0)]
  and exp[y > (const IntVal b 0)]
shows exp[x * y] ≥ exp[(x << ConstantExpr (IntVal 64 i)) + (x << Constant-
Expr (IntVal 64 j))]
  using assms apply simp
  by (metis (no-types, lifting) ConstantExprE equiv-exprs-def unfold-binary)

lemma greaterConstant:
  fixes a b :: 64 word
  assumes a > b
  and y = ConstantExpr (IntVal 64 a)
  and x = ConstantExpr (IntVal 64 b)
shows exp[y > x]
  apply auto

```

**sorry**

**lemma** *exp-distribute-multiplication*:

**shows**  $\text{exp}[(x * q) + (x * a)] \geq \text{exp}[x * (q + a)]$

**sorry**

Optimisations

**optimization** *EliminateRedundantNegative*:  $-x * -y \mapsto x * y$

**using** *mul-size.simps* **apply** *auto*

**by** (*metis BinaryExpr val-eliminate-redundant-negative bin-eval.simps(2)*)

**optimization** *MulNeutral*:  $x * \text{ConstantExpr } (\text{IntVal } b \ 1) \mapsto x$

**using** *exp-multiply-neutral* **by** *blast*

**optimization** *MulEliminator*:  $x * \text{ConstantExpr } (\text{IntVal } b \ 0) \mapsto \text{const } (\text{IntVal } b \ 0)$

**apply** *auto*

**by** (*smt (verit) Value.inject(1) constantAsStamp.simps(1) int-signed-value-bounds intval-mul.elims*

*mult-zero-right new-int.simps new-int-bin.simps take-bit-of-0 unfold-const*

*valid-stamp.simps(1) valid-value.simps(1) val-multiply-zero*)

**optimization** *MulNegate*:  $x * -(\text{const } (\text{IntVal } b \ 1)) \mapsto -x$

**apply** *auto*

**by** (*smt (verit) Value.distinct(1) Value.sel(1) add.inverse-inverse intval-mul.elims*

*intval-negate.simps(1) mask-eq-take-bit-minus-one new-int.simps new-int-bin.simps*

*take-bit-dist-neg unary-eval.simps(2) unfold-unary val-multiply-negative*

*val-eliminate-redundant-negative val-multiply-negative wf-value-def*)

**fun** *isNonZero* :: *Stamp*  $\Rightarrow$  *bool* **where**

*isNonZero* (*IntegerStamp* *b lo hi*) = (*lo* > 0) |

*isNonZero* - = *False*

**lemma** *isNonZero-defn*:

**assumes** *isNonZero* (*stamp-expr* *x*)

**assumes** *wf-stamp* *x*

**shows** ( $[m, p] \vdash x \mapsto v \longrightarrow (\exists vv \ b. (v = \text{IntVal } b \ vv \wedge \text{val-to-bool val}[(\text{IntVal } b \ 0) < v]))$ )

**apply** (*rule impI*) **subgoal** *premises* *eval*

**proof** –

**obtain** *b lo hi* **where** *xstamp*: *stamp-expr* *x* = *IntegerStamp* *b lo hi*

**by** (*meson isNonZero.elims(2) assms*)

**then obtain** *vv* **where** *vdef*: *v* = *IntVal* *b* *vv*

**by** (*metis assms(2) eval valid-int wf-stamp-def*)

**have** *lo* > 0

**using** *assms(1) xstamp* **by** *force*

**then have** *signed-above*: *int-signed-value* *b* *vv* > 0

```

    using assms unfolding wf-stamp-def
    using eval vdef xstamp by fastforce
  have take-bit b vv = vv
    using eval eval-unused-bits-zero vdef by auto
  then have vv > 0
    by (metis bit-take-bit-iff int-signed-value.simps not-less-zero signed-eq-0-iff
      signed-take-bit-eq-if-positive take-bit-0 take-bit-of-0 verit-comp-simplify1 (1)
      word-gt-0 signed-above)
  then show ?thesis
    using vdef signed-above
    by simp
qed
done

optimization MulPower2:  $x * y \mapsto x << \text{const } (\text{IntVal } 64 \ i)$ 
  when  $(i > 0 \wedge 64 > i \wedge y = \text{exp}[\text{const } (\text{IntVal } 64 \ (2 \wedge \text{unat}(i))])$ 

  defer
  apply simp apply (rule impI; (rule allI)+; rule impI)
  subgoal premises eval for m p v
proof -
  obtain xv where xv:  $[m, p] \vdash x \mapsto xv$ 
    using eval(2) by blast
  then obtain xv where xv:  $xv = \text{IntVal } 64 \ xv$ 
    by (smt (verit) ConstantExprE bin-eval.simps(2) evalDet intval-bits.simps int-
      val-mul.elims new-int-bin.simps unfold-binary eval)
  obtain yv where yv:  $[m, p] \vdash y \mapsto yv$ 
    using eval(1) eval(2) by blast
  then have lhs:  $[m, p] \vdash \text{exp}[x * y] \mapsto \text{val}[xv * yv]$ 
    by (metis bin-eval.simps(2) eval(1) eval(2) evalDet unfold-binary xv)
  have  $[m, p] \vdash \text{exp}[\text{const } (\text{IntVal } 64 \ i)] \mapsto \text{val}[(\text{IntVal } 64 \ i)]$ 
    by (smt (verit, ccfv-SIG) ConstantExpr constantAsStamp.simps(1) eval-bits-1-64
      take-bit64 validStampIntConst wf-value-def valid-value.simps(1) xv xv)
  then have rhs:  $[m, p] \vdash \text{exp}[x << \text{const } (\text{IntVal } 64 \ i)] \mapsto \text{val}[xv << (\text{IntVal } 64 \ i)]$ 
    using xv xv using evaltree.BinaryExpr
  by (metis Value.simps(5) bin-eval.simps(8) intval-left-shift.simps(1) new-int.simps)
  have  $\text{val}[xv * yv] = \text{val}[xv << (\text{IntVal } 64 \ i)]$ 
    by (metis ConstantExprE eval(1) evaltree-not-undef lhs yv val-MulPower2)
  then show ?thesis
    by (metis eval(1) eval(2) evalDet lhs rhs)
qed
done

```

```

optimization MulPower2Add1:  $x * y \mapsto (x << \text{const } (\text{IntVal } 64\ i)) + x$ 
  when  $(i > 0 \wedge$ 
     $64 > i \wedge$ 
     $y = \text{ConstantExpr } (\text{IntVal } 64\ ((2 \wedge \text{unat}(i)) + 1))$  )

  defer
  apply simp apply (rule impI; (rule allI)+; rule impI)
  subgoal premises p for m p v
  proof -
    obtain xv where xv:  $[m, p] \vdash x \mapsto xv$ 
    using p by fast
    then obtain xvv where xvv:  $xv = \text{IntVal } 64\ xvv$ 
    by (smt (verit) p ConstantExprE bin-eval.simps(2) evalDet intval-bits.simps
    intval-mul.elims
    new-int-bin.simps unfold-binary)
    obtain yv where yv:  $[m, p] \vdash y \mapsto yv$ 
    using p by blast
    have ygezero:  $y > \text{ConstantExpr } (\text{IntVal } 64\ 0)$ 
    using greaterConstant p wf-value-def by fastforce
    then have 1:  $0 < i \wedge$ 
       $i < 64 \wedge$ 
       $y = \text{ConstantExpr } (\text{IntVal } 64\ ((2 \wedge \text{unat}(i)) + 1))$ 
    using p by blast
    then have lhs:  $[m, p] \vdash \text{exp}[x * y] \mapsto \text{val}[xv * yv]$ 
    by (metis bin-eval.simps(2) evalDet p(1) p(2) xv yv unfold-binary)
    then have  $[m, p] \vdash \text{exp}[\text{const } (\text{IntVal } 64\ i)] \mapsto \text{val}[(\text{IntVal } 64\ i)]$ 
    by (metis wf-value-def verit-comp-simplify1(2) zero-less-numeral ConstantExpr
    constantAsStamp.simps(1) take-bit64 validStampIntConst valid-value.simps(1))
    then have rhs2:  $[m, p] \vdash \text{exp}[x << \text{const } (\text{IntVal } 64\ i)] \mapsto \text{val}[xv << (\text{IntVal } 64\ i)]$ 
    by (metis Value.simps(5) bin-eval.simps(8) intval-left-shift.simps(1) new-int.simps
    xv xvv
    evaltree.BinaryExpr)
    then have rhs:  $[m, p] \vdash \text{exp}[(x << \text{const } (\text{IntVal } 64\ i)) + x] \mapsto \text{val}[(xv << (\text{IntVal } 64\ i)) + xv]$ 
    by (metis (no-types, lifting) intval-add.simps(1) rhs2 bin-eval.simps(1)
    Value.simps(5)
    evaltree.BinaryExpr intval-left-shift.simps(1) new-int.simps xv xvv)
    then have simple:  $\text{val}[xv * (\text{IntVal } 64\ (2 \wedge \text{unat}(i)))] = \text{val}[xv << (\text{IntVal } 64\ i)]$ 
    using val-MulPower2 sorry
    then have  $\text{val}[xv * yv] = \text{val}[(xv << (\text{IntVal } 64\ i)) + xv]$ 
    sorry
    then show ?thesis
    by (metis 1 evalDet lhs p(2) rhs)
  qed
done

```

```

optimization MulPower2Sub1:  $x * y \mapsto (x << \text{const } (\text{IntVal } 64 \ i)) - x$ 
    when  $(i > 0 \wedge$ 
         $64 > i \wedge$ 
         $y = \text{ConstantExpr } (\text{IntVal } 64 \ ((2 \wedge \text{unat}(i)) - 1))$  )

defer
apply simp apply (rule impI; (rule allI)+; rule impI)
subgoal premises p for m p v
proof –
    obtain xv where xv:  $[m, p] \vdash x \mapsto xv$ 
    using p by fast
    then obtain xvv where xvv:  $xv = \text{IntVal } 64 \ xvv$ 
    by (smt (verit) p ConstantExprE bin-eval.simps(2) evalDet intval-bits.simps
    intval-mul.elims
        new-int-bin.simps unfold-binary)
    obtain yv where yv:  $[m, p] \vdash y \mapsto yv$ 
    using p by blast
    have ygezero:  $y > \text{ConstantExpr } (\text{IntVal } 64 \ 0)$ 
    by (smt (verit, del-insts) eq-iff-diff-eq-0 mask-0 mask-eq-exp-minus-1 power-inject-exp

        uint-2p unat-eq-zero word-gt-0 zero-neq-one greaterConstant p)
    then have 1:  $0 < i \wedge$ 
         $i < 64 \wedge$ 
         $y = \text{ConstantExpr } (\text{IntVal } 64 \ ((2 \wedge \text{unat}(i)) - 1))$ 
    using p by blast
    then have lhs:  $[m, p] \vdash \text{exp}[x * y] \mapsto \text{val}[xv * yv]$ 
    by (metis bin-eval.simps(2) evalDet p(1) p(2) xv yv unfold-binary)
    then have  $[m, p] \vdash \text{exp}[\text{const } (\text{IntVal } 64 \ i)] \mapsto \text{val}[(\text{IntVal } 64 \ i)]$ 
    by (metis wf-value-def verit-comp-simplify1(2) zero-less-numeral ConstantExpr

        constantAsStamp.simps(1) take-bit64 validStampIntConst valid-value.simps(1))
    then have rhs2:  $[m, p] \vdash \text{exp}[x << \text{const } (\text{IntVal } 64 \ i)] \mapsto \text{val}[xv << (\text{IntVal } 64 \ i)]$ 
    by (metis Value.simps(5) bin-eval.simps(8) intval-left-shift.simps(1) new-int.simps
    xv xvv
        evaltree.BinaryExpr)
    then have rhs:  $[m, p] \vdash \text{exp}[(x << \text{const } (\text{IntVal } 64 \ i)) - x] \mapsto \text{val}[(xv << (\text{IntVal } 64 \ i)) - xv]$ 
    by (smt (verit, ccfv-threshold) bin-eval.simps(3) new-int-bin.simps intval-sub.simps(1)

        rhs2 bin-eval.simps(1) Value.simps(5) evaltree.BinaryExpr intval-left-shift.simps(1)

        new-int.simps xv xvv )
    then have  $\text{val}[xv * yv] = \text{val}[(xv << (\text{IntVal } 64 \ i)) - xv]$ 
    using 1 exp-MulPower2Sub1 ygezero sorry
    then show ?thesis
    by (metis evalDet lhs p(1) p(2) rhs)
qed
done

```

end

end

## 1.7 Experimental AndNode Phase

**theory** *NewAnd*

**imports**

*Common*

*Graph.Long*

**begin**

**lemma** *bin-distribute-and-over-or*:

$bin[z \ \& \ (x \ | \ y)] = bin[(z \ \& \ x) \ | \ (z \ \& \ y)]$

**by** (*smt* (*verit*, *best*) *bit-and-iff* *bit-eqI* *bit-or-iff*)

**lemma** *intval-distribute-and-over-or*:

$val[z \ \& \ (x \ | \ y)] = val[(z \ \& \ x) \ | \ (z \ \& \ y)]$

**apply** (*cases* *x*; *cases* *y*; *cases* *z*; *auto*)

**using** *bin-distribute-and-over-or* **by** *blast+*

**lemma** *exp-distribute-and-over-or*:

$exp[z \ \& \ (x \ | \ y)] \geq exp[(z \ \& \ x) \ | \ (z \ \& \ y)]$

**apply** *simp* **using** *intval-distribute-and-over-or*

**using** *BinaryExpr* *bin-eval.simps*(4,5)

**using** *intval-or.simps*(1) **unfolding** *new-int-bin.simps* *new-int.simps* **apply** *auto*

**by** (*metis* *bin-eval.simps*(4) *bin-eval.simps*(5) *intval-or.simps*(2) *intval-or.simps*(5))

**lemma** *intval-and-commute*:

$val[x \ \& \ y] = val[y \ \& \ x]$

**by** (*cases* *x*; *cases* *y*; *auto* *simp*: *and.commute*)

**lemma** *intval-or-commute*:

$val[x \ | \ y] = val[y \ | \ x]$

**by** (*cases* *x*; *cases* *y*; *auto* *simp*: *or.commute*)

**lemma** *intval-xor-commute*:

$val[x \ \oplus \ y] = val[y \ \oplus \ x]$

**by** (*cases* *x*; *cases* *y*; *auto* *simp*: *xor.commute*)

**lemma** *exp-and-commute*:

$exp[x \ \& \ z] \geq exp[z \ \& \ x]$

**apply** *simp* **using** *intval-and-commute* **by** *auto*

**lemma** *exp-or-commute*:

$exp[x \ | \ y] \geq exp[y \ | \ x]$

**apply** *simp* **using** *intval-or-commute* **by** *auto*

```

lemma exp-xor-commute:
   $\text{exp}[x \oplus y] \geq \text{exp}[y \oplus x]$ 
  apply simp using intval-xor-commute by auto

lemma bin-eliminate-y:
  assumes  $\text{bin}[y \ \& \ z] = 0$ 
  shows  $\text{bin}[(x \mid y) \ \& \ z] = \text{bin}[x \ \& \ z]$ 
  using assms
  by (simp add: and.commute bin-distribute-and-over-or)

lemma intval-eliminate-y:
  assumes  $\text{val}[y \ \& \ z] = \text{IntVal } b \ 0$ 
  shows  $\text{val}[(x \mid y) \ \& \ z] = \text{val}[x \ \& \ z]$ 
  using assms bin-eliminate-y by (cases x; cases y; cases z; auto)

lemma intval-and-associative:
   $\text{val}[(x \ \& \ y) \ \& \ z] = \text{val}[x \ \& \ (y \ \& \ z)]$ 
  apply (cases x; cases y; cases z; auto)
  by (simp add: and.assoc)+

lemma intval-or-associative:
   $\text{val}[(x \mid y) \mid z] = \text{val}[x \mid (y \mid z)]$ 
  apply (cases x; cases y; cases z; auto)
  by (simp add: or.assoc)+

lemma intval-xor-associative:
   $\text{val}[(x \oplus y) \oplus z] = \text{val}[x \oplus (y \oplus z)]$ 
  apply (cases x; cases y; cases z; auto)
  by (simp add: xor.assoc)+

lemma exp-and-associative:
   $\text{exp}[(x \ \& \ y) \ \& \ z] \geq \text{exp}[x \ \& \ (y \ \& \ z)]$ 
  apply simp using intval-and-associative by fastforce

lemma exp-or-associative:
   $\text{exp}[(x \mid y) \mid z] \geq \text{exp}[x \mid (y \mid z)]$ 
  apply simp using intval-or-associative by fastforce

lemma exp-xor-associative:
   $\text{exp}[(x \oplus y) \oplus z] \geq \text{exp}[x \oplus (y \oplus z)]$ 
  apply simp using intval-xor-associative by fastforce

lemma intval-and-absorb-or:
  assumes  $\exists b \ v. x = \text{new-int } b \ v$ 
  assumes  $\text{val}[x \ \& \ (x \mid y)] \neq \text{UndefVal}$ 
  shows  $\text{val}[x \ \& \ (x \mid y)] = \text{val}[x]$ 

```



```

using assms apply (cases x; cases y; auto)
by (metis (mono-tags, lifting) intval-and.simps(5))

lemma intval-or-absorb-and:
  assumes  $\exists b\ v.\ x = \text{new-int } b\ v$ 
  assumes  $\text{val}[x \mid (x \ \&\ y)] \neq \text{UndefVal}$ 
  shows  $\text{val}[x \mid (x \ \&\ y)] = \text{val}[x]$ 
  using assms apply (cases x; cases y; auto)
  by (metis (mono-tags, lifting) intval-or.simps(5))

lemma exp-and-absorb-or:
   $\text{exp}[x \ \&\ (x \mid y)] \geq \text{exp}[x]$ 
  apply auto using intval-and-absorb-or eval-unused-bits-zero
  by (smt (verit) evalDet intval-or.elims new-int.elims)

lemma exp-or-absorb-and:
   $\text{exp}[x \mid (x \ \&\ y)] \geq \text{exp}[x]$ 
  apply auto using intval-or-absorb-and eval-unused-bits-zero
  by (smt (verit) evalDet intval-or.elims new-int.elims)

lemma
  assumes  $y = 0$ 
  shows  $x + y = \text{or } x\ y$ 
  using assms
  by simp

lemma no-overlap-or:
  assumes  $\text{and } x\ y = 0$ 
  shows  $x + y = \text{or } x\ y$ 
  using assms
  by (metis bit-and-iff bit-xor-iff disjunctive-add xor-self-eq)

context stamp-mask
begin

lemma intval-up-and-zero-implies-zero:
  assumes  $\text{and } (\uparrow x) (\uparrow y) = 0$ 
  assumes  $[m, p] \vdash x \mapsto xv$ 
  assumes  $[m, p] \vdash y \mapsto yv$ 
  assumes  $\text{val}[xv \ \&\ yv] \neq \text{UndefVal}$ 

```

```

shows  $\exists b. \text{val}[xv \ \& \ yv] = \text{new-int } b \ 0$ 
using assms apply (cases xv; cases yv; auto)
using up-mask-and-zero-implies-zero
apply (smt (verit, best) take-bit-and take-bit-of-0)
by presburger

lemma exp-eliminate-y:
  and  $(\uparrow y) (\uparrow z) = 0 \longrightarrow \text{BinaryExpr BinAnd } (\text{BinaryExpr BinOr } x \ y) \ z \geq \text{BinaryExpr BinAnd } x \ z$ 
apply simp apply (rule impI; rule allI; rule allI; rule allI)
subgoal premises p for m p v apply (rule impI) subgoal premises e
proof –
  obtain xv where xv:  $[m, p] \vdash x \mapsto xv$ 
  using e by auto
  obtain yv where yv:  $[m, p] \vdash y \mapsto yv$ 
  using e by auto
  obtain zv where zv:  $[m, p] \vdash z \mapsto zv$ 
  using e by auto
  have lhs:  $v = \text{val}[(xv \mid yv) \ \& \ zv]$ 
  using xv yv zv
  by (smt (verit, best) BinaryExprE bin-eval.simps(4) bin-eval.simps(5) e evalDet)
  then have  $v = \text{val}[(xv \ \& \ zv) \mid (yv \ \& \ zv)]$ 
  by (simp add: intval-and-commute intval-distribute-and-over-or)
  also have  $\exists b. \text{val}[yv \ \& \ zv] = \text{new-int } b \ 0$ 
  using intval-up-and-zero-implies-zero
  by (metis calculation e intval-or.simps(5) p unfold-binary yv zv)
  ultimately have rhs:  $v = \text{val}[xv \ \& \ zv]$ 
  using intval-eliminate-y lhs by force
  from lhs rhs show ?thesis
  by (metis BinaryExpr BinaryExprE bin-eval.simps(4) e xv zv)
qed
done
done

```

```

lemma leadingZeroBounds:
  fixes x :: 'a::len word
  assumes  $n = \text{numberOfLeadingZeros } x$ 
  shows  $0 \leq n \wedge n \leq \text{Nat.size } x$ 
  using assms unfolding numberOfLeadingZeros-def
  by (simp add: MaxOrNeg-def highestOneBit-def nat-le-iff)

```

```

lemma above-nth-not-set:
  fixes x :: int64
  assumes  $n = 64 - \text{numberOfLeadingZeros } x$ 
  shows  $j > n \longrightarrow \neg(\text{bit } x \ j)$ 
  using assms unfolding numberOfLeadingZeros-def
  by (smt (verit, ccfv-SIG) highestOneBit-def int-nat-eq int-ops(6) less-imp-of-nat-less max-set-bit size64 zerosAboveHighestOne)

```

**no-notation** *LogicNegationNotation* (!-)

**lemma** *zero-horner*:

*horner-sum of-bool 2 (map (λx. False) xs) = 0*  
**apply** (*induction xs*) **apply** *simp*  
**by** *force*

**lemma** *zero-map*:

**assumes**  $j \leq n$   
**assumes**  $\forall i. j \leq i \longrightarrow \neg(f\ i)$   
**shows**  $\text{map } f\ [0..<n] = \text{map } f\ [0..<j] @ \text{map } (\lambda x. \text{False})\ [j..<n]$   
**apply** (*insert assms*)  
**by** (*smt (verit, del-insts) add-diff-inverse-nat atLeastLessThan-iff bot-nat-0.extremum leD map-append map-eq-conv set-upt upt-add-eq-append*)

**lemma** *map-join-horner*:

**assumes**  $\text{map } f\ [0..<n] = \text{map } f\ [0..<j] @ \text{map } (\lambda x. \text{False})\ [j..<n]$   
**shows**  $\text{horner-sum of-bool } (2::'a::\text{len word})\ (\text{map } f\ [0..<n]) = \text{horner-sum of-bool } 2\ (\text{map } f\ [0..<j])$   
**proof** –  
**have**  $\text{horner-sum of-bool } (2::'a::\text{len word})\ (\text{map } f\ [0..<n]) = \text{horner-sum of-bool } 2\ (\text{map } f\ [0..<j]) + 2 \wedge \text{length } [0..<j] * \text{horner-sum of-bool } 2\ (\text{map } f\ [j..<n])$   
**using** *horner-sum-append*  
**by** (*smt (verit) assms diff-le-self diff-zero le-add-same-cancel2 length-append length-map length-upt map-append upt-add-eq-append*)  
**also have**  $\dots = \text{horner-sum of-bool } 2\ (\text{map } f\ [0..<j]) + 2 \wedge \text{length } [0..<j] * \text{horner-sum of-bool } 2\ (\text{map } (\lambda x. \text{False})\ [j..<n])$   
**using** *assms*  
**by** (*metis calculation horner-sum-append length-map*)  
**also have**  $\dots = \text{horner-sum of-bool } 2\ (\text{map } f\ [0..<j])$   
**using** *zero-horner*  
**using** *mult-not-zero* **by** *auto*  
**finally show** *?thesis* **by** *simp*  
**qed**

**lemma** *split-horner*:

**assumes**  $j \leq n$   
**assumes**  $\forall i. j \leq i \longrightarrow \neg(f\ i)$   
**shows**  $\text{horner-sum of-bool } (2::'a::\text{len word})\ (\text{map } f\ [0..<n]) = \text{horner-sum of-bool } 2\ (\text{map } f\ [0..<j])$   
**apply** (*rule map-join-horner*)  
**apply** (*rule zero-map*)  
**using** *assms* **by** *auto*

**lemma** *transfer-map*:

**assumes**  $\forall i. i < n \longrightarrow f\ i = f'\ i$   
**shows**  $(\text{map } f\ [0..<n]) = (\text{map } f'\ [0..<n])$   
**using** *assms* **by** *simp*

```

lemma transfer-horner:
  assumes  $\forall i. i < n \longrightarrow f\ i = f'\ i$ 
  shows horner-sum of-bool (2::'a::len word) (map f [0.. $n$ ]) = horner-sum of-bool
    2 (map f' [0.. $n$ ])
  using assms using transfer-map
  by (smt (verit, best))

lemma L1:
  assumes  $n = 64 - \text{numberOfLeadingZeros } (\uparrow z)$ 
  assumes  $[m, p] \vdash z \mapsto \text{IntVal } b\ zv$ 
  shows and  $v\ zv = \text{and } (v \bmod 2^n)\ zv$ 
proof -
  have nle:  $n \leq 64$ 
  using assms
  using diff-le-self by blast
  also have and  $v\ zv = \text{horner-sum of-bool } 2\ (\text{map } (\text{bit } (\text{and } v\ zv))\ [0.. $64$ ])$ 
  using horner-sum-bit-eq-take-bit size64
  by (metis size-word.rep-eq take-bit-length-eq)
  also have ... = horner-sum of-bool 2 (map ( $\lambda i. \text{bit } (\text{and } v\ zv)\ i$ ) [0.. $64$ ])
  by blast
  also have ... = horner-sum of-bool 2 (map ( $\lambda i. ((\text{bit } v\ i) \wedge (\text{bit } zv\ i))$ ) [0.. $64$ ])
  using bit-and-iff by metis
  also have ... = horner-sum of-bool 2 (map ( $\lambda i. ((\text{bit } v\ i) \wedge (\text{bit } zv\ i))$ ) [0.. $n$ ])
  proof -
  have  $\forall i. i \geq n \longrightarrow \neg(\text{bit } zv\ i)$ 
  using above-nth-not-set assms(1)
  using assms(2) not-may-implies-false
  by (smt (verit, ccfv-SIG) One-nat-def diff-less int-ops(6) leadingZerosAddHighestOne linorder-not-le nat-int-comparison(2) not-numeral-le-zero size64 zero-less-Suc zerosAboveHighestOne)
  then have  $\forall i. i \geq n \longrightarrow \neg((\text{bit } v\ i) \wedge (\text{bit } zv\ i))$ 
  by auto
  then show ?thesis using nle split-horner
  by (metis (no-types, lifting))
qed
  also have ... = horner-sum of-bool 2 (map ( $\lambda i. ((\text{bit } (v \bmod 2^n)\ i) \wedge (\text{bit } zv\ i))$ ) [0.. $n$ ])
  proof -
  have  $\forall i. i < n \longrightarrow \text{bit } (v \bmod 2^n)\ i = \text{bit } v\ i$ 
  by (metis bit-take-bit-iff take-bit-eq-mod)
  then have  $\forall i. i < n \longrightarrow ((\text{bit } v\ i) \wedge (\text{bit } zv\ i)) = ((\text{bit } (v \bmod 2^n)\ i) \wedge (\text{bit } zv\ i))$ 
  by force
  then show ?thesis
  by (rule transfer-horner)
qed
  also have ... = horner-sum of-bool 2 (map ( $\lambda i. ((\text{bit } (v \bmod 2^n)\ i) \wedge (\text{bit } zv\ i))$ ) [0.. $64$ ])

```

```

proof -
  have  $\forall i. i \geq n \longrightarrow \neg(\text{bit } zv \ i)$ 
    using above-nth-not-set assms(1)
    using assms(2) not-may-implies-false
    by smt (verit, ccfv-SIG) One-nat-def diff-less int-ops(6) leadingZerosAddHighestOne linorder-not-le nat-int-comparison(2) not-numeral-le-zero size64 zero-less-Suc zerosAboveHighestOne)
  then show ?thesis
    by (metis (no-types, lifting) assms(1) diff-le-self split-horner)
qed
also have  $\dots = \text{horner-sum of-bool } 2 \ (\text{map } (\text{bit } (\text{and } (v \text{ mod } 2^n) \ zv)) \ [0..<64])$ 
  by (meson bit-and-iff)
also have  $\dots = \text{and } (v \text{ mod } 2^n) \ zv$ 
  using horner-sum-bit-eq-take-bit size64
  by (metis size-word.rep-eq take-bit-length-eq)
finally show ?thesis
  using  $\langle \text{and } (v::64 \text{ word}) \ (zv::64 \text{ word}) = \text{horner-sum of-bool } (2::64 \text{ word})$ 
     $(\text{map } (\text{bit } (\text{and } v \ zv)) \ [0::\text{nat}..<64::\text{nat}]) \rangle \langle \text{horner-sum of-bool } (2::64 \text{ word}) \ (\text{map}$ 
     $(\lambda i::\text{nat}. \text{bit } ((v::64 \text{ word}) \text{ mod } (2::64 \text{ word}) \wedge (n::\text{nat})) \ i \wedge \text{bit } (zv::64 \text{ word})$ 
     $i) \ [0::\text{nat}..<64::\text{nat}]) = \text{horner-sum of-bool } (2::64 \text{ word}) \ (\text{map } (\text{bit } (\text{and } (v \text{ mod } (2::64 \text{ word}) \wedge n) \ zv)) \ [0::\text{nat}..<64::\text{nat}]) \rangle \langle \text{horner-sum of-bool } (2::64 \text{ word}) \ (\text{map}$ 
     $(\lambda i::\text{nat}. \text{bit } ((v::64 \text{ word}) \text{ mod } (2::64 \text{ word}) \wedge (n::\text{nat})) \ i \wedge \text{bit } (zv::64 \text{ word}) \ i)$ 
     $[0::\text{nat}..<n]) = \text{horner-sum of-bool } (2::64 \text{ word}) \ (\text{map } (\lambda i::\text{nat}. \text{bit } (v \text{ mod } (2::64 \text{ word}) \wedge n) \ i \wedge \text{bit } zv \ i) \ [0::\text{nat}..<64::\text{nat}]) \rangle \langle \text{horner-sum of-bool } (2::64 \text{ word})$ 
     $(\text{map } (\lambda i::\text{nat}. \text{bit } (v::64 \text{ word}) \ i \wedge \text{bit } (zv::64 \text{ word}) \ i) \ [0::\text{nat}..<64::\text{nat}]) =$ 
     $\text{horner-sum of-bool } (2::64 \text{ word}) \ (\text{map } (\lambda i::\text{nat}. \text{bit } v \ i \wedge \text{bit } zv \ i) \ [0::\text{nat}..<n::\text{nat}]) \rangle \langle \text{horner-sum of-bool } (2::64 \text{ word}) \ (\text{map } (\lambda i::\text{nat}. \text{bit } (v::64 \text{ word}) \ i \wedge \text{bit } (zv::64 \text{ word}) \ i) \ [0::\text{nat}..<n::\text{nat}]) = \text{horner-sum of-bool } (2::64 \text{ word}) \ (\text{map } (\lambda i::\text{nat}. \text{bit } (v \text{ mod } (2::64 \text{ word}) \wedge n) \ i \wedge \text{bit } zv \ i) \ [0::\text{nat}..<n]) \rangle \langle \text{horner-sum of-bool } (2::64 \text{ word}) \ (\text{map } (\text{bit } (\text{and } ((v::64 \text{ word}) \text{ mod } (2::64 \text{ word}) \wedge (n::\text{nat})) \ (zv::64 \text{ word}))) \ [0::\text{nat}..<64::\text{nat}]) = \text{and } (v \text{ mod } (2::64 \text{ word}) \wedge n) \ zv \rangle \langle \text{horner-sum of-bool } (2::64 \text{ word}) \ (\text{map } (\text{bit } (\text{and } (v::64 \text{ word}) \ (zv::64 \text{ word}))) \ [0::\text{nat}..<64::\text{nat}]) = \text{horner-sum of-bool } (2::64 \text{ word}) \ (\text{map } (\lambda i::\text{nat}. \text{bit } v \ i \wedge \text{bit } zv \ i) \ [0::\text{nat}..<64::\text{nat}]) \rangle$ 
    by presburger
qed

```

**lemma** *up-mask-upper-bound:*

```

assumes  $[m, p] \vdash x \mapsto \text{IntVal } b \ xv$ 
shows  $xv \leq (\uparrow x)$ 
using assms
by (metis (no-types, lifting) and.idem and.right-neutral bit.conj-cancel-left bit.conj-disj-distrib(1) bit.double-compl ucast-id up-spec word-and-le1 word-not-dist(2))

```

**lemma** *L2:*

```

assumes  $\text{numberOfLeadingZeros } (\uparrow z) + \text{numberOfTrailingZeros } (\uparrow y) \geq 64$ 
assumes  $n = 64 - \text{numberOfLeadingZeros } (\uparrow z)$ 
assumes  $[m, p] \vdash z \mapsto \text{IntVal } b \ zv$ 
assumes  $[m, p] \vdash y \mapsto \text{IntVal } b \ yv$ 
shows  $yv \text{ mod } 2^n = 0$ 

```

```

proof –
  have  $yv \bmod 2^{\wedge n} = \text{horner-sum of-bool } 2 \ (\text{map } (\text{bit } yv) \ [0..<n])$ 
    by (simp add: horner-sum-bit-eq-take-bit take-bit-eq-mod)
  also have  $\dots \leq \text{horner-sum of-bool } 2 \ (\text{map } (\text{bit } (\uparrow y)) \ [0..<n])$ 
    using up-mask-upper-bound assms(4)
    by (metis (no-types, opaque-lifting) and.right-neutral bit.conj-cancel-right bit.conj-disj-distrib(1)
bit.double-compl horner-sum-bit-eq-take-bit take-bit-and ucast-id up-spec word-and-le1
word-not-dist(2)))
  also have  $\text{horner-sum of-bool } 2 \ (\text{map } (\text{bit } (\uparrow y)) \ [0..<n]) = \text{horner-sum of-bool } 2$ 
(map (λx. False) [0..<n])
  proof –
    have  $\forall i < n. \neg(\text{bit } (\uparrow y) \ i)$ 
      using assms(1,2) zerosBelowLowestOne
      by (metis add commute add-diff-inverse-nat add-lessD1 leD le-diff-conv numberOfTrailingZeros-def)
    then show ?thesis
      by (metis (full-types) transfer-map)
  qed
  also have  $\text{horner-sum of-bool } 2 \ (\text{map } (\lambda x. \text{False}) \ [0..<n]) = 0$ 
    using zero-horner
    by blast
  finally show ?thesis
    by auto
qed

thm-oracles L1 L2

lemma unfold-binary-width-add:
  shows  $([m,p] \vdash \text{BinaryExpr BinAdd } xe \ ye \mapsto \text{IntVal } b \ \text{val}) = (\exists \ x \ y. \$ 
 $(([m,p] \vdash xe \mapsto \text{IntVal } b \ x) \wedge$ 
 $([m,p] \vdash ye \mapsto \text{IntVal } b \ y) \wedge$ 
 $(\text{IntVal } b \ \text{val} = \text{bin-eval BinAdd } (\text{IntVal } b \ x) \ (\text{IntVal } b \ y)) \wedge$ 
 $(\text{IntVal } b \ \text{val} \neq \text{UndefVal})$ 
 $)) \text{ (is } ?L = ?R)$ 
  proof (intro iffI)
    assume  $\mathcal{I}: ?L$ 
    show  $?R$  apply (rule evaltree.cases[OF 3])
      apply force+ apply auto[1]
      apply (smt (verit) intval-add.elims intval-bits.simps)
      by blast
  next
    assume  $R: ?R$ 
    then obtain  $x \ y$  where  $[m,p] \vdash xe \mapsto \text{IntVal } b \ x$ 
      and  $[m,p] \vdash ye \mapsto \text{IntVal } b \ y$ 
      and  $\text{new-int } b \ \text{val} = \text{bin-eval BinAdd } (\text{IntVal } b \ x) \ (\text{IntVal } b \ y)$ 
      and  $\text{new-int } b \ \text{val} \neq \text{UndefVal}$ 
      by auto
    then show  $?L$ 
      using  $R$  by blast

```

qed

**lemma** *unfold-binary-width-and:*

**shows**  $([m,p] \vdash \text{BinaryExpr BinAnd } xe \ ye \mapsto \text{IntVal } b \ \text{val}) = (\exists \ x \ y. \\ (([m,p] \vdash xe \mapsto \text{IntVal } b \ x) \wedge \\ ([m,p] \vdash ye \mapsto \text{IntVal } b \ y) \wedge \\ (\text{IntVal } b \ \text{val} = \text{bin-eval BinAnd } (\text{IntVal } b \ x) \ (\text{IntVal } b \ y)) \wedge \\ (\text{IntVal } b \ \text{val} \neq \text{UndefVal})) \text{ (is } ?L = ?R)$

**proof** (*intro iffI*)  
**assume**  $?L$   
**show**  $?R$  **apply** (*rule evaltree.cases[OF ?L]*)  
**apply** *force+* **apply** *auto[1]* **using** *intval-and.elims intval-bits.simps*  
**apply** (*smt (verit) new-int.simps new-int-bin.simps take-bit-and*)  
**by** *blast*

**next**  
**assume**  $?R$   
**then obtain**  $x \ y$  **where**  $[m,p] \vdash xe \mapsto \text{IntVal } b \ x$   
**and**  $[m,p] \vdash ye \mapsto \text{IntVal } b \ y$   
**and**  $\text{new-int } b \ \text{val} = \text{bin-eval BinAnd } (\text{IntVal } b \ x) \ (\text{IntVal } b \ y)$   
**and**  $\text{new-int } b \ \text{val} \neq \text{UndefVal}$   
**by** *auto*  
**then show**  $?L$   
**using**  $R$  **by** *blast*

qed

**lemma** *mod-dist-over-add-right:*

**fixes**  $a \ b \ c :: \text{int64}$   
**fixes**  $n :: \text{nat}$   
**assumes**  $1: 0 < n$   
**assumes**  $2: n < 64$   
**shows**  $(a + b \bmod 2^n) \bmod 2^n = (a + b) \bmod 2^n$   
**using** *mod-dist-over-add*  
**by** (*simp add: 1 2 add.commute*)

**lemma** *numberOfLeadingZeros-range:*

$0 \leq \text{numberOfLeadingZeros } n \wedge \text{numberOfLeadingZeros } n \leq \text{Nat.size } n$   
**unfolding** *numberOfLeadingZeros-def highestOneBit-def* **using** *max-set-bit*  
**by** (*simp add: highestOneBit-def leadingZeroBounds numberOfLeadingZeros-def*)

**lemma** *improved-opt:*

**assumes**  $\text{numberOfLeadingZeros } (\uparrow z) + \text{numberOfTrailingZeros } (\uparrow y) \geq 64$   
**shows**  $\text{exp}[(x + y) \ \& \ z] \geq \text{exp}[x \ \& \ z]$   
**apply** *simp* **apply** (*(rule allI)+; rule impI*)  
**subgoal** **premises** *eval* **for**  $m \ p \ v$

**proof** –  
**obtain**  $n$  **where**  $n: n = 64 - \text{numberOfLeadingZeros } (\uparrow z)$   
**by** *simp*  
**obtain**  $b \ \text{val}$  **where**  $\text{val}: [m, p] \vdash \text{exp}[(x + y) \ \& \ z] \mapsto \text{IntVal } b \ \text{val}$

```

    by (metis BinaryExprE bin-eval-new-int eval new-int.simps)
  then obtain xv yv where addv:  $[m, p] \vdash \text{exp}[x + y] \mapsto \text{IntVal } b \ (xv + yv)$ 
    apply (subst (asm) unfold-binary-width-and) by (metis add.right-neutral)
  then obtain yv where yv:  $[m, p] \vdash y \mapsto \text{IntVal } b \ yv$ 
    apply (subst (asm) unfold-binary-width-add) by blast
  from addv obtain xv where xv:  $[m, p] \vdash x \mapsto \text{IntVal } b \ xv$ 
    apply (subst (asm) unfold-binary-width-add) by blast
  from val obtain zv where zv:  $[m, p] \vdash z \mapsto \text{IntVal } b \ zv$ 
    apply (subst (asm) unfold-binary-width-and) by blast
  have addv:  $[m, p] \vdash \text{exp}[x + y] \mapsto \text{new-int } b \ (xv + yv)$ 
    apply (rule evaltree.BinaryExpr)
    using xv apply simp
    using yv apply simp
    by simp+
  have lhs:  $[m, p] \vdash \text{exp}[(x + y) \ \& \ z] \mapsto \text{new-int } b \ (\text{and } (xv + yv) \ zv)$ 
    apply (rule evaltree.BinaryExpr)
    using addv apply simp
    using zv apply simp
    using addv apply auto[1]
    by simp
  have rhs:  $[m, p] \vdash \text{exp}[x \ \& \ z] \mapsto \text{new-int } b \ (\text{and } xv \ zv)$ 
    apply (rule evaltree.BinaryExpr)
    using xv apply simp
    using zv apply simp
    apply force
    by simp
  then show ?thesis
proof (cases numberOfLeadingZeros ( $\uparrow z$ ) > 0)
  case True
    have n-bounds:  $0 \leq n \wedge n < 64$ 
      using diff-le-self n numberOfLeadingZeros-range
      by (simp add: True)
    have and (xv + yv) zv = and ((xv + yv) mod  $2^n$ ) zv
      using L1 n zv by blast
    also have ... = and ((xv + (yv mod  $2^n$ )) mod  $2^n$ ) zv
      using mod-dist-over-add-right n-bounds
      by (metis take-bit-0 take-bit-eq-mod zero-less-iff-neq-zero)
    also have ... = and (((xv mod  $2^n$ ) + (yv mod  $2^n$ )) mod  $2^n$ ) zv
      by (metis bits-mod-by-1 mod-dist-over-add n-bounds order-le-imp-less-or-eq
        power-0)
    also have ... = and ((xv mod  $2^n$ ) mod  $2^n$ ) zv
      using L2 n zv yv
      using assms by auto
    also have ... = and (xv mod  $2^n$ ) zv
      using mod-mod-trivial
      by (smt (verit, best) and.idem take-bit-eq-mask take-bit-eq-mod word-bw-assocs(1))
    also have ... = and xv zv
      using L1 n zv by metis
  finally show ?thesis

```



```

    using eval lhs rhs
    by (metis evalDet)
next
case False
then have numberOfLeadingZeros ( $\uparrow z$ ) = 0
    by simp
then have numberOfTrailingZeros ( $\uparrow y$ )  $\geq$  64
    using assms(1)
    by fastforce
then have  $yv = 0$ 
    using  $yv$ 
    by (metis (no-types, lifting) L1 L2 add-diff-cancel-left' and.comm-neutral
and.idem bit.compl-zero bit.conj-cancel-right bit.conj-disj-distrib(1) bit.double-compl
less-imp-diff-less linorder-not-le word-not-dist(2))
    then show ?thesis
        by (metis add.right-neutral eval evalDet lhs rhs)
qed
qed
done

thm-oracles improved-opt

```

end

```

phase NewAnd
terminating size
begin

```

```

optimization redundant-lhs-y-or:  $((x \mid y) \& z) \mapsto x \& z$ 
    when  $((\text{and } (IRExpr\text{-up } y) (IRExpr\text{-up } z)) = 0)$ 
    apply (simp add: IRExpr-up-def)
    using simple-mask.exp-eliminate-y by blast

```

```

optimization redundant-lhs-x-or:  $((x \mid y) \& z) \mapsto y \& z$ 
    when  $((\text{and } (IRExpr\text{-up } x) (IRExpr\text{-up } z)) = 0)$ 
    apply (simp add: IRExpr-up-def)
    using simple-mask.exp-eliminate-y
    by (meson exp-or-commute mono-binary order-refl order-trans)

```

```

optimization redundant-rhs-y-or:  $(z \& (x \mid y)) \mapsto z \& x$ 
    when  $((\text{and } (IRExpr\text{-up } y) (IRExpr\text{-up } z)) = 0)$ 
    apply (simp add: IRExpr-up-def)
    using simple-mask.exp-eliminate-y
    by (meson exp-and-commute order.trans)

```

```

optimization redundant-rhs-x-or:  $(z \& (x \mid y)) \mapsto z \& y$ 
      when  $((\text{and } (IRExpr\text{-up } x) (IRExpr\text{-up } z)) = 0)$ 
  apply (simp add: IRExpr-up-def)
  using simple-mask.exp-eliminate-y
  by (meson dual-order.trans exp-and-commute exp-or-commute mono-binary or-
der-refl)

```

**end**

**end**

## 1.8 NotNode Phase

```

theory NotPhase
  imports
    Common
  begin

  phase NotNode
    terminating size
  begin

```

```

lemma bin-not-cancel:
   $\text{bin}[\neg(\neg(e))] = \text{bin}[e]$ 
  by auto

```

```

lemma val-not-cancel:
  assumes  $\text{val}[\sim(\text{new-int } b \ v)] \neq \text{UndefVal}$ 
  shows  $\text{val}[\sim(\sim(\text{new-int } b \ v))] = (\text{new-int } b \ v)$ 
  by (simp add: take-bit-not-take-bit)

```

```

lemma exp-not-cancel:
   $\text{exp}[\sim(\sim a)] \geq \text{exp}[a]$ 
  using val-not-cancel apply auto
  by (metis eval-unused-bits-zero intval-logic-negation.cases new-int.simps intval-not.simps(1)

      intval-not.simps(2) intval-not.simps(3) intval-not.simps(4))

```

Optimisations

```

optimization NotCancel:  $\text{exp}[\sim(\sim a)] \mapsto a$ 
  by (metis exp-not-cancel)

```

**end**

end

## 1.9 OrNode Phase

```
theory OrPhase
  imports
    Common
begin
```

```
context stamp-mask
begin
```

Taking advantage of the truth table of or operations.

#	x	y	$x y$
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	1

If row 2 never applies, that is,  $\text{canBeZero } x \ \& \ \text{canBeOne } y = 0$ , then  $(x|y) = x$ .

Likewise, if row 3 never applies,  $\text{canBeZero } y \ \& \ \text{canBeOne } x = 0$ , then  $(x|y) = y$ .

```
lemma OrLeftFallthrough:
  assumes (and (not ( $\downarrow x$ )) ( $\uparrow y$ )) = 0
  shows  $\text{exp}[x \mid y] \geq \text{exp}[x]$ 
  using assms
  apply simp apply ((rule allI)+; rule impI)
  subgoal premises eval for m p v
  proof -
    obtain b vv where e:  $[m, p] \vdash \text{exp}[x \mid y] \mapsto \text{IntVal } b \ vv$ 
    by (metis BinaryExprE bin-eval-new-int new-int.simps eval(2))
    from e obtain xv where xv:  $[m, p] \vdash x \mapsto \text{IntVal } b \ xv$ 
    apply (subst (asm) unfold-binary-width)
    by force+
    from e obtain yv where yv:  $[m, p] \vdash y \mapsto \text{IntVal } b \ yv$ 
    apply (subst (asm) unfold-binary-width)
    by force+
    have vdef:  $v = \text{intval-or } (\text{IntVal } b \ xv) (\text{IntVal } b \ yv)$ 
    by (metis bin-eval.simps(5) eval(2) evalDet unfold-binary xv yv)
    have  $\forall i. (\text{bit } xv \ i) \mid (\text{bit } yv \ i) = (\text{bit } v \ i)$ 
    by (metis assms bit-and-iff not-down-up-mask-and-zero-implies-zero xv yv)
    then have  $\text{IntVal } b \ xv = \text{intval-or } (\text{IntVal } b \ xv) (\text{IntVal } b \ yv)$ 
    by (smt (verit, ccfv-threshold) and.idem assms bit.conj-disj-distrib eval-unused-bits-zero
```

```

    intval-or.simps(1) new-int.simps new-int-bin.simps not-down-up-mask-and-zero-implies-zero

    word-ao-absorbs(3) xv yv
  then show ?thesis
    using xv vdef by presburger
qed
done

lemma OrRightFallthrough:
  assumes (and (not (↓y)) (↑x)) = 0
  shows  $\text{exp}[x \mid y] \geq \text{exp}[y]$ 
  using assms
  apply simp apply ((rule allI)+; rule impI)
  subgoal premises eval for m p v
  proof -
    obtain b vv where e:  $[m, p] \vdash \text{exp}[x \mid y] \mapsto \text{IntVal } b \text{ } vv$ 
      by (metis BinaryExprE bin-eval-new-int new-int.simps eval(2))
    from e obtain xv where xv:  $[m, p] \vdash x \mapsto \text{IntVal } b \text{ } xv$ 
      apply (subst (asm) unfold-binary-width)
      by force+
    from e obtain yv where yv:  $[m, p] \vdash y \mapsto \text{IntVal } b \text{ } yv$ 
      apply (subst (asm) unfold-binary-width)
      by force+
    have vdef:  $v = \text{intval-or } (\text{IntVal } b \text{ } xv) (\text{IntVal } b \text{ } yv)$ 
      by (metis bin-eval.simps(5) eval(2) evalDet unfold-binary xv yv)
    have  $\forall i. (\text{bit } xv \text{ } i) \mid (\text{bit } yv \text{ } i) = (\text{bit } yv \text{ } i)$ 
      by (metis assms bit-and-iff not-down-up-mask-and-zero-implies-zero xv yv)
    then have  $\text{IntVal } b \text{ } yv = \text{intval-or } (\text{IntVal } b \text{ } xv) (\text{IntVal } b \text{ } yv)$ 
      by (metis (no-types, lifting) assms eval-unused-bits-zero intval-or.simps(1)
new-int.elims
new-int-bin.elims stamp-mask.not-down-up-mask-and-zero-implies-zero
stamp-mask-axioms
word-ao-absorbs(8) xv yv)
    then show ?thesis
      using vdef yv by presburger
qed
done

end

phase OrNode
  terminating size
begin

lemma bin-or-equal:
   $\text{bin}[x \mid x] = \text{bin}[x]$ 
  by simp

```

**lemma** *bin-shift-const-right-helper*:

$x \mid y = y \mid x$

**by** *simp*

**lemma** *bin-or-not-operands*:

$(\sim x \mid \sim y) = (\sim(x \& y))$

**by** *simp*

**lemma** *val-or-equal*:

**assumes**  $x = \text{new-int } b \ v$

**and**  $(\text{val}[x \mid x] \neq \text{UndefVal})$

**shows**  $\text{val}[x \mid x] = \text{val}[x]$

**apply** (*cases*  $x$ ; *auto*) **using** *bin-or-equal* *assms*

**by** *auto*+

**lemma** *val-elim-redundant-false*:

**assumes**  $x = \text{new-int } b \ v$

**and**  $\text{val}[x \mid \text{false}] \neq \text{UndefVal}$

**shows**  $\text{val}[x \mid \text{false}] = \text{val}[x]$

**using** *assms* **apply** (*cases*  $x$ ; *auto*) **by** *presburger*

**lemma** *val-shift-const-right-helper*:

$\text{val}[x \mid y] = \text{val}[y \mid x]$

**apply** (*cases*  $x$ ; *cases*  $y$ ; *auto*)

**by** (*simp* *add: or.commute*)+

**lemma** *val-or-not-operands*:

$\text{val}[\sim x \mid \sim y] = \text{val}[\sim(x \& y)]$

**apply** (*cases*  $x$ ; *cases*  $y$ ; *auto*)

**by** (*simp* *add: take-bit-not-take-bit*)

**lemma** *exp-or-equal*:

$\text{exp}[x \mid x] \geq \text{exp}[x]$

**using** *val-or-equal* **apply** *auto*[1]

**by** (*smt* (*verit*, *ccfv-SIG*) *evalDet* *eval-unused-bits-zero* *intval-negate.elims* *int-val-or.simps*(2)

*intval-or.simps*(6) *intval-or.simps*(7) *new-int.simps* *val-or-equal*)

**lemma** *exp-elim-redundant-false*:

$\text{exp}[x \mid \text{false}] \geq \text{exp}[x]$

**using** *val-elim-redundant-false* **apply** *auto*[1]

**by** (*smt* (*verit*) *Value.sel*(1) *eval-unused-bits-zero* *intval-or.elims* *new-int.simps*

*new-int-bin.simps* *val-elim-redundant-false*)

Optimisations

**optimization** *OrEqual*:  $x \mid x \longmapsto x$

```

    by (meson exp-or-equal)

optimization OrShiftConstantRight: ((const x) | y)  $\mapsto$  y | (const x) when  $\neg$ (is-ConstantExpr y)
  using size-flip-binary apply force
  apply auto[1]
  by (simp add: BinaryExpr unfold-const val-shift-const-right-helper)

optimization EliminateRedundantFalse: x | false  $\mapsto$  x
  by (meson exp-elim-redundant-false)

optimization OrNotOperands: ( $\sim$ x |  $\sim$ y)  $\mapsto$   $\sim$ (x & y)
  apply (metis add-2-eq-Suc' less-SucI not-add-less1 not-less-eq size-binary-const
size-non-add)
  apply auto[1]
  by (metis BinaryExpr UnaryExpr bin-eval.simps(4) intval-not.simps(2) unary-eval.simps(3)

      val-or-not-operands)

optimization OrLeftFallthrough:
  x | y  $\mapsto$  x when ((and (not (IExpr-down x)) (IExpr-up y)) = 0)
  using simple-mask.OrLeftFallthrough by blast

optimization OrRightFallthrough:
  x | y  $\mapsto$  y when ((and (not (IExpr-down y)) (IExpr-up x)) = 0)
  using simple-mask.OrRightFallthrough by blast

end

end

```

## 1.10 ShiftNode Phase

```

theory ShiftPhase
  imports
    Common
  begin

  phase ShiftNode
    terminating size
  begin

  fun intval-log2 :: Value  $\Rightarrow$  Value where
    intval-log2 (IntVal b v) = IntVal b (word-of-int (SOME e. v=2e)) |
    intval-log2 - = UndefVal

  fun in-bounds :: Value  $\Rightarrow$  int  $\Rightarrow$  int  $\Rightarrow$  bool where
    in-bounds (IntVal b v) l h = (l < sint v  $\wedge$  sint v < h) |

```

*in-bounds - l h = False*

**lemma**

**assumes** *in-bounds (intval-log2 val-c) 0 32*

**shows** *intval-left-shift x (intval-log2 val-c) = intval-mul x val-c*

**apply** (*cases val-c; auto*) **using** *intval-left-shift.simps(1) intval-mul.simps(1) intval-log2.simps(1)*

**sorry**

**lemma** *e-intval:*

*n = intval-log2 val-c  $\wedge$  in-bounds n 0 32  $\longrightarrow$*

*intval-left-shift x (intval-log2 val-c) =*

*intval-mul x val-c*

**proof** (*rule impI*)

**assume** *n = intval-log2 val-c  $\wedge$  in-bounds n 0 32*

**show** *intval-left-shift x (intval-log2 val-c) =*

*intval-mul x val-c*

**proof** (*cases  $\exists v . val-c = IntVal 32 v$* )

**case** *True*

**obtain** *vc where val-c = IntVal 32 vc*

**using** *True by blast*

**then have** *n = IntVal 32 (word-of-int (SOME e. vc=2<sup>e</sup>))*

**using**  *$\langle n = intval-log2 val-c \wedge in-bounds n 0 32 \rangle$  intval-log2.simps(1) by*

*presburger*

**then show** *?thesis sorry*

**next**

**case** *False*

**then have**  *$\exists v . val-c = IntVal 64 v$*

**sorry**

**then obtain** *vc where val-c = IntVal 64 vc*

**by** *auto*

**then have** *n = IntVal 64 (word-of-int (SOME e. vc=2<sup>e</sup>))*

**using**  *$\langle n = intval-log2 val-c \wedge in-bounds n 0 32 \rangle$  intval-log2.simps(1) by*

*presburger*

**then show** *?thesis sorry*

**qed**

**qed**

**optimization** *e:*

*x \* (const c)  $\longmapsto$  x << (const n) when (n = intval-log2 c  $\wedge$  in-bounds n 0 32)*

**using** *e-intval*

**using** *BinaryExprE ConstantExprE bin-eval.simps(2,7) sorry*

**end**

**end**

### 1.11 SignedDivNode Phase

**theory** *SignedDivPhase*

**imports**

*Common*

**begin**

**phase** *SignedDivNode*

**terminating** *size*

**begin**

**lemma** *val-division-by-one-is-self-32*:

**assumes**  $x = \text{new-int } 32 \ v$

**shows**  $\text{intval-div } x \ (\text{IntVal } 32 \ 1) = x$

**using** *assms* **apply** (*cases*  $x$ ; *auto*)

**by** (*simp add: take-bit-signed-take-bit*)

**end**

**end**

### 1.12 SignedRemNode Phase

**theory** *SignedRemPhase*

**imports**

*Common*

**begin**

**phase** *SignedRemNode*

**terminating** *size*

**begin**

**lemma** *val-remainder-one*:

**assumes**  $\text{intval-mod } x \ (\text{IntVal } 32 \ 1) \neq \text{UndefVal}$

**shows**  $\text{intval-mod } x \ (\text{IntVal } 32 \ 1) = \text{IntVal } 32 \ 0$

**using** *assms* **apply** (*cases*  $x$ ; *auto*) **sorry**

**value** *word-of-int* (*sint* ( $x2::32 \ \text{word}$ ) *smod* 1)

**end**

**end**



### 1.13 SubNode Phase

```
theory SubPhase
  imports
    Common
    Proofs.StampEvalThms
begin

phase SubNode
  terminating size
begin

lemma bin-sub-after-right-add:
  shows  $((x :: ('a :: len) \text{ word}) + (y :: ('a :: len) \text{ word})) - y = x$ 
  by simp

lemma sub-self-is-zero:
  shows  $(x :: ('a :: len) \text{ word}) - x = 0$ 
  by simp

lemma bin-sub-then-left-add:
  shows  $(x :: ('a :: len) \text{ word}) - (x + (y :: ('a :: len) \text{ word})) = -y$ 
  by simp

lemma bin-sub-then-left-sub:
  shows  $(x :: ('a :: len) \text{ word}) - (x - (y :: ('a :: len) \text{ word})) = y$ 
  by simp

lemma bin-subtract-zero:
  shows  $(x :: 'a :: len \text{ word}) - (0 :: 'a :: len \text{ word}) = x$ 
  by simp

lemma bin-sub-negative-value:
   $(x :: ('a :: len) \text{ word}) - (-(y :: ('a :: len) \text{ word})) = x + y$ 
  by simp

lemma bin-sub-self-is-zero:
   $(x :: ('a :: len) \text{ word}) - x = 0$ 
  by simp

lemma bin-sub-negative-const:
   $(x :: 'a :: len \text{ word}) - (-(y :: 'a :: len \text{ word})) = x + y$ 
  by simp

lemma val-sub-after-right-add-2:
  assumes  $x = \text{new-int } b \ v$ 
  assumes  $\text{val}[(x + y) - y] \neq \text{UndefVal}$ 
  shows  $\text{val}[(x + y) - y] = \text{val}[x]$ 
```

```

using bin-sub-after-right-add
using assms apply (cases x; cases y; auto)
by (metis (full-types) intval-sub.simps(2))

lemma val-sub-after-left-sub:
  assumes val[(x - y) - x] ≠ UndefVal
  shows val[(x - y) - x] = val[-y]
  using assms apply (cases x; cases y; auto)
  using intval-sub.elims by fastforce

lemma val-sub-then-left-sub:
  assumes y = new-int b v
  assumes val[x - (x - y)] ≠ UndefVal
  shows val[x - (x - y)] = val[y]
  using assms apply (cases x; cases y; auto)
  by (metis (mono-tags) intval-sub.simps(5))

lemma val-subtract-zero:
  assumes x = new-int b v
  assumes intval-sub x (IntVal b 0) ≠ UndefVal
  shows intval-sub x (IntVal b 0) = val[x]
  using assms by (induction x; simp)

lemma val-zero-subtract-value:
  assumes x = new-int b v
  assumes intval-sub (IntVal b 0) x ≠ UndefVal
  shows intval-sub (IntVal b 0) x = val[-x]
  using assms by (induction x; simp)

lemma val-sub-then-left-add:
  assumes val[x - (x + y)] ≠ UndefVal
  shows val[x - (x + y)] = val[-y]
  using assms apply (cases x; cases y; auto)
  by (metis (mono-tags, lifting) intval-sub.simps(5))

lemma val-sub-negative-value:
  assumes val[x - (-y)] ≠ UndefVal
  shows val[x - (-y)] = val[x + y]
  using assms by (cases x; cases y; auto)

lemma val-sub-self-is-zero:
  assumes x = new-int b v ∧ val[x - x] ≠ UndefVal
  shows val[x - x] = new-int b 0
  using assms by (cases x; auto)

lemma val-sub-negative-const:
  assumes y = new-int b v ∧ val[x - (-y)] ≠ UndefVal
  shows val[x - (-y)] = val[x + y]
  using assms by (cases x; cases y; auto)

```

```

lemma exp-sub-after-right-add:
  shows  $\exp[(x + y) - y] \geq \exp[x]$ 
  apply auto
  by (smt (verit) evalDet eval-unused-bits-zero intval-add.elims new-int.simps
    val-sub-after-right-add-2)

lemma exp-sub-after-right-add2:
  shows  $\exp[(x + y) - x] \geq \exp[y]$ 
  using exp-sub-after-right-add apply auto
  by (smt (z3) Value.inject(1) diff-eq-eq evalDet eval-unused-bits-zero intval-add.elims

    intval-sub.elims new-int.simps new-int-bin.simps take-bit-dist-subL bin-eval.simps(1)

    bin-eval.simps(3) intval-add-sym unfold-binary)

lemma exp-sub-negative-value:
   $\exp[x - (-y)] \geq \exp[x + y]$ 
  apply simp
  by (smt (verit) bin-eval.simps(1) bin-eval.simps(3) evaltree-not-undef unary-eval.simps(2)

    unfold-binary unfold-unary val-sub-negative-value)

lemma exp-sub-then-left-sub:
   $\exp[x - (x - y)] \geq \exp[y]$ 
  using val-sub-then-left-sub apply auto
  subgoal premises p for m p xa xaa ya
  proof –
    obtain xa where xa:  $[m, p] \vdash x \mapsto xa$ 
    using p(2) by blast
    obtain ya where ya:  $[m, p] \vdash y \mapsto ya$ 
    using p(5) by auto
    obtain xaa where xaa:  $[m, p] \vdash x \mapsto xaa$ 
    using p(2) by blast
    have 1:  $\text{val}[xa - (xaa - ya)] \neq \text{UndefVal}$ 
    by (metis evalDet p(2) p(3) p(4) p(5) xa xaa ya)
    then have  $\text{val}[xaa - ya] \neq \text{UndefVal}$ 
    by auto
    then have  $[m, p] \vdash y \mapsto \text{val}[xa - (xaa - ya)]$ 
    by (metis 1 Value.exhaust evalDet eval-unused-bits-zero evaltree-not-undef
      intval-sub.simps(6) intval-sub.simps(7) new-int.simps p(5) val-sub-then-left-sub
      xa xaa ya)
    then show ?thesis
    by (metis evalDet p(2) p(4) p(5) xa xaa ya)
  qed
done

```

**thm-oracles** *exp-sub-then-left-sub*

Optimisations

**optimization** *SubAfterAddRight*:  $((x + y) - y) \mapsto x$   
**using** *exp-sub-after-right-add* **by** *blast*

**optimization** *SubAfterAddLeft*:  $((x + y) - x) \mapsto y$   
**using** *exp-sub-after-right-add2* **by** *blast*

**optimization** *SubAfterSubLeft*:  $((x - y) - x) \mapsto -y$   
**apply** (*metis Suc-lessI add-2-eq-Suc' add-less-cancel-right less-trans-Suc not-add-less1*

*size-binary-const size-binary-lhs size-binary-rhs size-non-add*)  
**apply** *auto*  
**by** (*metis evalDet unary-eval.simps(2) unfold-unary val-sub-after-left-sub*)

**optimization** *SubThenAddLeft*:  $(x - (x + y)) \mapsto -y$   
**apply** *auto*  
**by** (*metis evalDet unary-eval.simps(2) unfold-unary val-sub-then-left-add*)

**optimization** *SubThenAddRight*:  $(y - (x + y)) \mapsto -x$   
**apply** *auto*  
**by** (*metis evalDet intval-add-sym unary-eval.simps(2) unfold-unary val-sub-then-left-add*)

**optimization** *SubThenSubLeft*:  $(x - (x - y)) \mapsto y$   
**using** *size-simps* **apply** *simp*  
**using** *exp-sub-then-left-sub* **by** *blast*

**optimization** *SubtractZero*:  $(x - (\text{const IntVal } b \ 0)) \mapsto x$   
**apply** *auto*  
**by** (*smt (verit) add.right-neutral diff-add-cancel eval-unused-bits-zero intval-sub.elims*  
*intval-word.simps new-int.simps new-int-bin.simps*)

**thm-oracles** *SubtractZero*

**optimization** *SubNegativeValue*:  $(x - (-y)) \mapsto x + y$   
**apply** (*metis add-2-eq-Suc' less-SucI less-add-Suc1 not-less-eq size-binary-const*  
*size-non-add*)  
**using** *exp-sub-negative-value* **by** *simp*

**thm-oracles** *SubNegativeValue*

**lemma** *negate-idempotent*:  
**assumes**  $x = \text{IntVal } b \ v \wedge \text{take-bit } b \ v = v$   
**shows**  $x = \text{val}[-(-x)]$

```

using assms
using is-IntVal-def by force

optimization ZeroSubtractValue:  $((\text{const IntVal } b \ 0) - x) \mapsto (-x)$ 
                                     when  $(\text{wf-stamp } x \wedge \text{stamp-expr } x = \text{IntegerStamp } b \ \text{lo} \ \text{hi} \wedge \neg(\text{is-ConstantExpr } x))$ 
  defer
  apply auto unfolding wf-stamp-def
  apply  $(\text{smt } (\text{verit}) \ \text{diff-0} \ \text{intval-negate.simps}(1) \ \text{intval-sub.elims} \ \text{intval-word.simps}$ 
         $\text{new-int-bin.simps} \ \text{unary-eval.simps}(2) \ \text{unfold-unary})$ 
  using add-2-eq-Suc' size.simps(2) size-flip-binary by presburger

optimization SubSelfIsZero:  $(x - x) \mapsto \text{const IntVal } b \ 0$  when
                                      $(\text{wf-stamp } x \wedge \text{stamp-expr } x = \text{IntegerStamp } b \ \text{lo} \ \text{hi})$ 
  apply simp-all
  apply auto
  using IRExpr.disc(42) One-nat-def size-non-const apply presburger
  by  $(\text{smt } (\text{verit}, \text{best}) \ \text{wf-value-def} \ \text{ConstantExpr} \ \text{evalDet} \ \text{eval-bits-1-64} \ \text{eval-unused-bits-zero}$ 
         $\text{new-int.simps} \ \text{take-bit-of-0} \ \text{val-sub-self-is-zero} \ \text{validDefIntConst} \ \text{valid-int} \ \text{wf-stamp-def})$ 

end

end

### 1.14 XorNode Phase

theory XorPhase
  imports
    Common
    Proofs.StampEvalThms
  begin

  phase XorNode
    terminating size
  begin

  lemma bin-xor-self-is-false:
     $\text{bin}[x \oplus x] = 0$ 
    by simp

```

```

lemma bin-xor-commute:
  bin[x  $\oplus$  y] = bin[y  $\oplus$  x]
  by (simp add: xor.commute)

lemma bin-eliminate-redundant-false:
  bin[x  $\oplus$  0] = bin[x]
  by simp

lemma val-xor-self-is-false:
  assumes val[x  $\oplus$  x]  $\neq$  UndefVal
  shows val-to-bool (val[x  $\oplus$  x]) = False
  using assms by (cases x; auto)

lemma val-xor-self-is-false-2:
  assumes (val[x  $\oplus$  x])  $\neq$  UndefVal
  and x = IntVal 32 v
  shows val[x  $\oplus$  x] = bool-to-val False
  using assms by (cases x; auto)

lemma val-xor-self-is-false-3:
  assumes val[x  $\oplus$  x]  $\neq$  UndefVal  $\wedge$  x = IntVal 64 v
  shows val[x  $\oplus$  x] = IntVal 64 0
  using assms by (cases x; auto)

lemma val-xor-commute:
  val[x  $\oplus$  y] = val[y  $\oplus$  x]
  apply (cases x; cases y; auto)
  by (simp add: xor.commute)+

lemma val-eliminate-redundant-false:
  assumes x = new-int b v
  assumes val[x  $\oplus$  (bool-to-val False)]  $\neq$  UndefVal
  shows val[x  $\oplus$  (bool-to-val False)] = x
  using assms apply (cases x; auto)
  by meson

lemma exp-xor-self-is-false:
  assumes wf-stamp x  $\wedge$  stamp-expr x = default-stamp
  shows exp[x  $\oplus$  x]  $\geq$  exp[false]
  using assms apply auto unfolding wf-stamp-def
  by (smt (z3) validDefIntConst IntVal0 Value.inject(1) bool-to-val.simps(2)
    constantAsStamp.simps(1) evalDet int-signed-value-bounds new-int.simps unf-
fold-const
    val-xor-self-is-false-2 valid-int valid-stamp.simps(1) valid-value.simps(1) wf-value-def)

lemma exp-eliminate-redundant-false:

```

```

shows  $\exp[x \oplus \text{false}] \geq \exp[x]$ 
using val-eliminate-redundant-false apply auto
subgoal premises p for m p xa
proof -
  obtain xa where xa:  $[m, p] \vdash x \mapsto xa$ 
  using p(2) by blast
  then have  $\text{val}[xa \oplus (\text{IntVal } 32 \ 0)] \neq \text{UndefVal}$ 
  using evalDet p(2) p(3) by blast
  then have  $[m, p] \vdash x \mapsto \text{val}[xa \oplus (\text{IntVal } 32 \ 0)]$ 
  apply (cases xa; auto) using eval-unused-bits-zero xa by auto
  then show ?thesis
  using evalDet p(2) xa by blast
qed
done

```

Optimisations

```

optimization XorSelfIsFalse:  $(x \oplus x) \mapsto \text{false}$  when
  (wf-stamp x  $\wedge$  stamp-expr x = default-stamp)
  using size-non-const apply force
  using exp-xor-self-is-false by auto

optimization XorShiftConstantRight:  $((\text{const } x) \oplus y) \mapsto y \oplus (\text{const } x)$  when
 $\neg(\text{is-ConstantExpr } y)$ 
  using size-flip-binary apply force
  unfolding le-expr-def using val-xor-commute
  by auto

optimization EliminateRedundantFalse:  $(x \oplus \text{false}) \mapsto x$ 
  using exp-eliminate-redundant-false by blast

```

end

end

## 1.15 NegateNode Phase

```

theory NegatePhase
  imports
    Common
begin

  phase NegateNode
    terminating size
begin

```

**lemma** *bin-negative-cancel*:  
 $-1 * (-1 * ((x::('a::len) word))) = x$   
**by** *auto*

**lemma** *val-negative-cancel*:  
**assumes** *intval-negate* (*new-int* *b* *v*)  $\neq$  *UndefVal*  
**shows**  $val[-(- (new-int\ b\ v))] = val[new-int\ b\ v]$   
**using** *assms* **by** *simp*

**lemma** *val-distribute-sub*:  
**assumes**  $x \neq UndefVal \wedge y \neq UndefVal$   
**shows**  $val[-(x - y)] = val[y - x]$   
**using** *assms* **by** (*cases* *x*; *cases* *y*; *auto*)

**lemma** *exp-distribute-sub*:  
**shows**  $exp[-(x - y)] \geq exp[y - x]$   
**using** *val-distribute-sub* **apply** *auto*  
**using** *evaltree-not-undef* **by** *auto*

**thm-oracles** *exp-distribute-sub*

**lemma** *exp-negative-cancel*:  
**shows**  $exp[-(-x)] \geq exp[x]$   
**using** *val-negative-cancel* **apply** *auto*  
**by** (*metis* (*no-types*, *opaque-lifting*) *eval-unused-bits-zero* *intval-negate.elims*  
*intval-negate.simps*(1) *minus-equation-iff* *new-int.simps* *take-bit-dist-neg*)

**lemma** *exp-negative-shift*:  
**assumes** *stamp-expr*  $x = IntegerStamp\ b'\ lo\ hi$   
**and**  $unat\ y = (b' - 1)$   
**shows**  $exp[-(x >> (const\ (new-int\ b\ y)))] \geq exp[x >>> (const\ (new-int\ b\ y))]$   
**apply** *auto*  
**subgoal** *premises* *p* **for** *m* *p* *xa*  
**proof**  $-$   
**obtain** *xa* **where** *xa*:  $[m, p] \vdash x \mapsto xa$   
**using** *p*(2) **by** *auto*  
**then** **have** 1: *intval-negate* (*intval-right-shift* *xa* (*IntVal* *b* (*take-bit* *b* *y*)))  $\neq$  *UndefVal*  
**using** *evalDet* *p*(1) *p*(2) **by** *blast*  
**then** **have** 2: *intval-right-shift* *xa* (*IntVal* *b* (*take-bit* *b* *y*)))  $\neq$  *UndefVal*  
**by** *auto*  
**then** **have** 3:  $- ((2::int) \wedge b\ div\ (2::int)) \sqsubseteq sint\ (signed-take-bit\ (b - Suc\ (0::nat))\ (take-bit\ b\ y))$   
**by** (*smt* (*verit*, *del-insts*) *One-nat-def* *diff-le-self* *gr0I* *half-nonnegative-int-iff*  
*linorder-not-le* *lower-bounds-equiv* *power-increasing-iff* *signed-0* *signed-take-bit-int-greater-eq-minus-exp-word*  
*signed-take-bit-of-0* *sint-greater-eq* *take-bit-0*)



```

    then have 4: sint (signed-take-bit (b - Suc (0::nat)) (take-bit b y)) < (2::int)
    ^ b div (2::int)
    by (metis Suc-le-lessD Suc-pred eval-bits-1-64 int-power-div-base p(4) signed-take-bit-int-less-exp-word
size64 unfold-const wsst-TYs(3) zero-less-numeral)
    then have 5: (0::nat) < b
    using eval-bits-1-64 p(4) by blast
    then have 6: b  $\sqsubseteq$  (64::nat)
    using eval-bits-1-64 p(4) by blast
    then have 7: [m,p]  $\vdash$  BinaryExpr BinURightShift x
    (ConstantExpr (IntVal b (take-bit b y)))  $\mapsto$ 
    intval-negate (intval-right-shift xa (IntVal b (take-bit b y)))
    apply (cases y; auto)

subgoal premises p for n
proof -
  have sg1: y = word-of-nat n
  by (simp add: p(1))
  then have sg2: n < (18446744073709551616::nat)
  by (simp add: p(2))
  then have sg3: b  $\sqsubseteq$  (64::nat)
  by (simp add: 6)
  then have sg4: [m,p]  $\vdash$  BinaryExpr BinURightShift x
  (ConstantExpr (IntVal b (take-bit b (word-of-nat n))))  $\mapsto$ 
  intval-negate (intval-right-shift xa (IntVal b (take-bit b (word-of-nat
n))))
  sorry
  then show ?thesis
  by simp
qed
done
then show ?thesis
by (metis evalDet p(2) xa)
qed
done

```

Optimisations

```

optimization NegateCancel:  $\neg(\neg(x)) \mapsto x$ 
  using exp-negative-cancel by blast

```

```

optimization DistributeSubtraction:  $\neg(x - y) \mapsto (y - x)$ 
  apply (smt (z3) add.left-commute add-2-eq-Suc' add-diff-cancel-left' is-ConstantExpr-def
less-Suc-eq-0-disj plus-1-eq-Suc size.simps(11) size-binary-const size-non-add
zero-less-diff)
  using exp-distribute-sub by simp

```

```

optimization NegativeShift:  $-(x >> (\text{const } (\text{new-int } b \ y))) \mapsto x >>> (\text{const } (\text{new-int } b \ y))$ 
   $\text{when } (\text{stamp-expr } x = \text{IntegerStamp } b' \ \text{lo } \text{hi} \wedge \text{unat } y = (b' - 1))$ 
  using exp-negative-shift by simp

end

end
theory TacticSolving
  imports Common
begin

fun size :: IRExpr  $\Rightarrow$  nat where
  size (UnaryExpr op e) = (size e) * 2 |
  size (BinaryExpr BinAdd x y) = (size x) + ((size y) * 2) |
  size (BinaryExpr op x y) = (size x) + (size y) |
  size (ConditionalExpr cond t f) = (size cond) + (size t) + (size f) + 2 |
  size (ConstantExpr c) = 1 |
  size (ParameterExpr ind s) = 2 |
  size (LeafExpr nid s) = 2 |
  size (ConstantVar c) = 2 |
  size (VariableExpr x s) = 2

lemma size-pos[simp]:  $0 < \text{size } y$ 
  apply (induction y; auto?)
  subgoal premises prems for op a b
    using prems by (induction op; auto)
  done

phase TacticSolving
  terminating size
begin

```

## 1.16 AddNode

```

lemma value-approx-implies-refinement:
  assumes lhs  $\approx$  rhs
  assumes  $\forall m \ p \ v. ([m, p] \vdash \text{elhs} \mapsto v) \longrightarrow v = \text{lhs}$ 
  assumes  $\forall m \ p \ v. ([m, p] \vdash \text{erhs} \mapsto v) \longrightarrow v = \text{rhs}$ 
  assumes  $\forall m \ p \ v1 \ v2. ([m, p] \vdash \text{elhs} \mapsto v1) \longrightarrow ([m, p] \vdash \text{erhs} \mapsto v2)$ 
  shows  $\text{elhs} \geq \text{erhs}$ 
  using assms unfolding le-expr-def well-formed-equal-def
  using evalDet evaltree-not-undef
  by metis

method explore-cases for x y :: Value =
  (cases x; cases y; auto)

```

```

method explore-cases-bin for  $x :: IRExpr =$ 
  (cases  $x$ ; auto)

method obtain-approx-eq for  $lhs\ rhs\ x\ y :: Value =$ 
  (rule meta-mp[where  $P = lhs \approx rhs$ ], defer-tac, explore-cases  $x\ y$ )

method obtain-eval for  $exp :: IRExpr$  and  $val :: Value =$ 
  (rule meta-mp[where  $P = \bigwedge m\ p\ v. ([m, p] \vdash exp \mapsto v) \implies v = val$ ], defer-tac)

method solve for  $lhs\ rhs\ x\ y :: Value =$ 
  (match conclusion in  $size - < size - \Rightarrow \langle simp \rangle$ )?,
  (match conclusion in ( $elhs :: IRExpr$ )  $\geq$  ( $erhs :: IRExpr$ ) for  $elhs\ erhs \Rightarrow \langle$ 
    (obtain-approx-eq  $lhs\ rhs\ x\ y$ )?)

```

**print-methods**

```

thm BinaryExprE
optimization opt-add-left-negate-to-sub:
   $-x + y \mapsto y - x$ 

  apply (solve  $val[-x1 + y1]$   $val[y1 - x1]$   $x1\ y1$ )
  apply simp apply auto using evaltree-not-undef sorry

```

## 1.17 NegateNode

```

lemma val-distribute-sub:
   $val[-(x-y)] \approx val[y-x]$ 
  by (cases  $x$ ; cases  $y$ ; auto)

optimization distribute-sub:  $-(x-y) \mapsto (y-x)$ 
  apply simp
  using val-distribute-sub apply simp
  using unfold-binary unfold-unary by auto

```

```

lemma val-xor-self-is-false:
  assumes  $x = IntVal\ 32\ v$ 
  shows  $val[x \oplus x] \approx val[false]$ 
  apply simp using assms by (cases  $x$ ; auto)

```

```

definition wf-stamp  $:: IRExpr \Rightarrow bool$  where
  wf-stamp  $e = (\forall m\ p\ v. ([m, p] \vdash e \mapsto v) \longrightarrow valid-value\ v\ (stamp-expr\ e))$ 

```

```

lemma exp-xor-self-is-false:
  assumes  $stamp-expr\ x = IntegerStamp\ 32\ l\ h$ 
  assumes wf-stamp  $x$ 
  shows  $exp[x \oplus x] \geq exp[false]$ 

```

**unfolding** *le-expr-def* **using** *assms* **unfolding** *wf-stamp-def*  
**using** *val-xor-self-is-false* *evaltree-not-undef*  
**by** (*smt* (*z3*) *wf-value-def* *bin-eval.simps(6)* *bin-eval-new-int* *constantAsStamp.simps(1)* *evalDet*  
*int-signed-value-bounds* *new-int.simps* *new-int-take-bits* *unfold-binary* *un-*  
*fold-const* *valid-int*  
*valid-stamp.simps(1)* *valid-value.simps(1)* *well-formed-equal-defn*)

**lemma** *val-or-commute[simp]*:  
 $val[x \mid y] = val[y \mid x]$   
**apply** (*cases* *x*; *cases* *y*; *auto*)  
**by** (*simp* *add: or.commute*)**+**

**lemma** *val-xor-commute[simp]*:  
 $val[x \oplus y] = val[y \oplus x]$   
**apply** (*cases* *x*; *cases* *y*; *auto*)  
**by** (*simp* *add: word-bw-comms(3)*)

**lemma** *exp-or-commutative*:  
 $exp[x \mid y] \geq exp[y \mid x]$   
**by** *auto*

**lemma** *exp-xor-commutative*:  
 $exp[x \oplus y] \geq exp[y \oplus x]$   
**by** *auto*

**lemma** *OrInverseVal*:  
**assumes**  $n = IntVal\ 32\ v$   
**shows**  $val[n \mid \sim n] \approx new-int\ 32\ (-1)$   
**apply** *simp* **using** *assms* **using** *word-or-not* **apply** (*cases* *n*; *auto*) **using** *take-bit-or*  
**by** (*metis* *bit.disj-cancel-right* *mask-eq-take-bit-minus-one*)

**optimization** *OrInverse*:  $exp[n \mid \sim n] \mapsto (const\ (new-int\ 32\ (not\ 0)))$   
*when* (*stamp-expr* *n* = *IntegerStamp* 32 *l* *h*  $\wedge$  *wf-stamp* *n*)  
**unfolding** *size.simps* **apply** (*simp* *add: Suc-lessI*)  
**apply** *auto* **using** *OrInverseVal* **unfolding** *wf-stamp-def*  
**by** (*smt* (*z3*) *wf-value-def* *constantAsStamp.simps(1)* *evalDet* *int-signed-value-bounds*  
*mask-eq-take-bit-minus-one* *new-int.elims* *new-int-take-bits* *unfold-const* *valid-int*  
*valid-stamp.simps(1)* *valid-value.simps(1)* *well-formed-equal-defn*)

**optimization** *OrInverse2*:  $exp[\sim n \mid n] \mapsto (const\ (new-int\ 32\ (not\ 0)))$   
*when* (*stamp-expr* *n* = *IntegerStamp* 32 *l* *h*  $\wedge$  *wf-stamp* *n*)  
**using** *OrInverse* *exp-or-commutative* **by** *auto*

```

lemma XorInverseVal:
  assumes  $n = \text{IntVal } 32 \ v$ 
  shows  $\text{val}[n \oplus \sim n] \approx \text{new-int } 32 \ (-1)$ 
  apply simp using assms using word-or-not apply (cases  $n$ ; auto)
  by (metis (no-types, opaque-lifting) bit.compl-zero bit.xor-compl-right bit.xor-self

    mask-eq-take-bit-minus-one take-bit-xor)

optimization XorInverse:  $\text{exp}[n \oplus \sim n] \mapsto (\text{const } (\text{new-int } 32 \ (\text{not } 0)))$ 
  when (stamp-expr  $n = \text{IntegerStamp } 32 \ l \ h \wedge \text{wf-stamp } n$ )
  unfolding size.simps apply (simp add: Suc-lessI)
  apply auto using XorInverseVal
  by (smt (verit) wf-value-def constantAsStamp.simps(1) evalDet int-signed-value-bounds

    intval.xor.elims mask-eq-take-bit-minus-one new-int.elims new-int-take-bits
  unfold-const
    valid-stamp.simps(1) valid-value.simps(1) well-formed-equal-defn wf-stamp-def)

optimization XorInverse2:  $\text{exp}[(\sim n) \oplus n] \mapsto (\text{const } (\text{new-int } 32 \ (\text{not } 0)))$ 
  when (stamp-expr  $n = \text{IntegerStamp } 32 \ l \ h \wedge \text{wf-stamp } n$ )
  using XorInverse exp-xor-commutative by auto

end

end

theory ProofStatus
  imports
    AbsPhase
    AddPhase
    AndPhase
    ConditionalPhase
    MulPhase

    NegatePhase
    NewAnd
    NotPhase
    OrPhase
    ShiftPhase
    SignedDivPhase
    SignedRemPhase
    SubPhase
    TacticSolving
    XorPhase
  begin

  declare  $[[\text{show-types=false}]]$ 
  print-phases
  print-phases!

```

```
print-methods

print-theorems

thm opt-add-left-negate-to-sub
thm-oracles AbsNegate

export-phases ⟨Full⟩

end
```