Veriopt Theories

April 17, 2024

Contents

1 Canonicalization Optimizations

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

lemma *size-pos*[*size-simps*]: *0 < size y*
  apply (*induction y*; *auto?*)
  subgoal for *op*
    apply (*cases op*)
    by (*smt* (*z3*) *gr0I one-neq-zero pos2 size.elims trans-less-add2*)+

done

lemma *size-non-add*[*size-simps*]: *size* (*BinaryExpr op a b*) = *size a* + *size b* + *2* ⟷ ¬(*is-ConstantExpr b*)
by (*induction b*; *induction op*; *auto simp*: *is-ConstantExpr-def*)

lemma *size-non-const*[*size-simps*]:
¬ *is-ConstantExpr y* ⟹ *1* < *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*]:
size (*BinaryExpr op a b*) = *size a* + *2* ⟷ (*is-ConstantExpr b*)
by (*induction b*; *auto simp*: *is-ConstantExpr-def size-pos*)

lemma *size-flip-binary*[*size-simps*]:
¬(*is-ConstantExpr y*) ⟶ *size* (*BinaryExpr op* (*ConstantExpr x*) *y*) > *size* (*BinaryExpr op y* (*ConstantExpr x*))
by (*metis add-Suc not-less-eq order-less-asym plus-1-eq-Suc size.simps*(*2*,*11*) *size-non-add*)

lemma *size-binary-lhs-a*[*size-simps*]:
size (*BinaryExpr op* (*BinaryExpr op′ a b*) *c*) > *size a*
by (*metis add-lessD1 less-add-same-cancel1 pos2 size-binary-const size-non-add*)

lemma *size-binary-lhs-b*[*size-simps*]:
size (*BinaryExpr op* (*BinaryExpr op′ a b*) *c*) > *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*]:
size (*BinaryExpr op* (*BinaryExpr op′ a b*) *c*) > *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*]:
size (*BinaryExpr op c* (*BinaryExpr op′ a b*)) > *size a*
apply *auto*
by (*metis trans-less-add2 less-Suc-eq less-add-same-cancel1 linorder-neqE-nat not-add-less1 pos2*
order-less-trans size-binary-const size-non-add)

lemma *size-binary-rhs-b*[*size-simps*]:
size (*BinaryExpr op c* (*BinaryExpr op′ a b*)) > *size b*
by (*metis add.left-commute add.right-neutral is-ConstantExpr-def lessI numeral-2-eq-2 plus-1-eq-Suc size.simps*(*4*,*11*) *size-non-add trans-less-add2*)

lemma *size-binary-rhs-c*[*size-simps*]:
size (*BinaryExpr op c* (*BinaryExpr op′ a b*)) > *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* ⇒ *Value* ⇒ *bool*
(**infix** ≈ *50*) **where**
well-formed-equal v_1 v_2 = (v_1 ≠ *UndefVal* ⟶ v_1 = v_2)

lemma *well-formed-equal-defn* [*simp*]:
well-formed-equal v_1 v_2 = (v_1 ≠ *UndefVal* ⟶ v_1 = v_2)
unfolding *well-formed-equal-def* **by** *simp*

end

1.1 AbsNode Phase

theory *AbsPhase*
imports
Common Proofs.StampEvalThms
begin

phase *AbsNode*
terminating *size*
begin

Note:

We can't use (<*s*) for reasoning about *intval-less-than*. (<*s*) will always treat the 64^{th} bit as the sign flag while *intval-less-than* uses the b^{th} bit depending on the size of the word.

value *val*[*new-int 32 0* < *new-int 32 4294967286*] — 0 < -10 = False
value (*0*::*int64*) <*s 4294967286* — 0 < 4294967286 = True

lemma *signed-eqiv*:
assumes *b* > *0* ∧ *b* ≤ *64*

shows *val-to-bool* (*val*[*new-int b v* < *new-int b v′*]) = (*int-signed-value b v* < *int-signed-value b v′*)
using *assms*
by (*metis* (*no-types*, *lifting*) *ValueThms.signed-take-take-bit bool-to-val.elims bool-to-val-bin.elims int-signed-value.simps intval-less-than.simps*(*1*) *new-int.simps one-neq-zero 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* **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* **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-64*:
assumes *0* < *b* ∧ *b* ≤ *64*
assumes *take-bit b v* = *v*
shows *take-bit 64 v* = *take-bit b v*
using *assms*
by (*metis min-def nle-le take-bit-take-bit*)

A special value exists for the maximum negative integer as its negation is itself. We can define the value as *set-bit* ((*b*::*nat*) − (*1*::*nat*)) (*0*::*64 word*) for any bit-width, b.

value (*set-bit 1 0*)::*2 word* — 2
value −(*set-bit 1 0*)::*2 word* — 2
value (*set-bit 31 0*)::*32 word* — 2147483648
value −(*set-bit 31 0*)::*32 word* — 2147483648

lemma *negative-def*:
fixes *v* :: *′a*::*len word*
assumes *v* <*s 0*
shows *bit v* (*LENGTH*(*′a*) − *1*)
using *assms*
by (*simp add*: *bit-last-iff word-sless-alt*)

lemma *positive-def*:
fixes *v* :: *′a*::*len word*
assumes *0* <*s v*
shows ¬(*bit v* (*LENGTH*(*′a*) − *1*))
using *assms*

by (*simp add*: *bit-last-iff word-sless-alt*)

lemma *negative-lower-bound*:
fixes v :: *'a::len word*
assumes $(2\hat{}(LENGTH('a) - 1)) <s\ v$
assumes $v <s\ 0$
shows $0 <s\ (-v)$
using *assms*
by (*smt* (*verit*) *signed-0 signed-take-bit-int-less-self-iff sint-ge sint-word-ariths*(*4*) *word-sless-alt*)

lemma *min-int*:
fixes x :: *'a::len word*
assumes $x <s\ 0$
assumes $x \neq (2\hat{}(LENGTH('a) - 1))$
shows $2\hat{}(LENGTH('a) - 1) <s\ x$
using *assms* **sorry**

lemma *negate-min-int*:
fixes v :: *'a::len word*
assumes $v = (2\hat{}(LENGTH('a) - 1))$
shows $v = (-v)$
using *assms*
by (*metis One-nat-def add.inverse-neutral double-eq-zero-iff mult-minus-right verit-minus-simplify*(*4*))

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

lemma
$abs(abs(x)) = abs(x)$
for x :: *'a::len word*
proof (*cases* $0 \leq s\ x$)
case *True*
then show *?thesis*
by *force*
next
case *neg*: *False*
then show *?thesis*
proof (*cases* $x = (2\hat{}LENGTH('a) - 1)$)
case *True*
then show *?thesis*
using *negate-min-int*
by (*simp add*: *word-sless-alt*)
next
case *False*

```
    then show ?thesis using min-int negative-lower-bound
      using negate-min-int by force
  qed
qed
```

We need to do the same proof at the value level.

```
lemma invert-intval:
  assumes int-signed-value b v < 0
  assumes b > 0 ∧ b ≤ 64
  assumes take-bit b v = v
  assumes v ≠ (2^(b − 1))
  shows 0 < int-signed-value b (−v)
  using assms apply simp sorry

lemma negate-max-negative:
  assumes b > 0 ∧ b ≤ 64
  assumes take-bit b v = v
  assumes v = (2^(b − 1))
  shows new-int b v = intval-negate (new-int b v)
  using assms apply simp using negate-min-int sorry

lemma val-abs-always-pos:
  assumes b > 0 ∧ b ≤ 64
  assumes take-bit b v = v
  assumes v ≠ (2^(b − 1))
  assumes intval-abs (new-int b v) = (new-int b v′)
  shows val-to-bool (val[(new-int b 0) < (new-int b v′)]) ∨ val-to-bool (val[(new-int
b 0) eq (new-int b v′)])
proof (cases v = 0)
  case True
  then have isZero: intval-abs (new-int b 0) = new-int b 0
    by auto
  then have IntVal b 0 = new-int b v′
    using True assms by auto
  then have val-to-bool (val[(new-int b 0) eq (new-int b v′)])
    by simp
  then show ?thesis by simp
next
  case neq0: False
  have zero: int-signed-value b 0 = 0
    by simp
  then show ?thesis
  proof (cases int-signed-value b v > 0)
    case True
    then have val-to-bool(val[(new-int b 0) < (new-int b v)])
      using zero apply simp
     by (metis One-nat-def ValueThms.signed-take-take-bit assms(1) val-bool-unwrap)
    then have val-to-bool (val[new-int b 0 < new-int b v′])
      by (metis assms(4) val-abs-pos)
```

```
    then show ?thesis
      by blast
  next
    case neg: False
    then have val-to-bool (val[new-int b 0 < new-int b v'])
    proof −
      have int-signed-value b v ≤ 0
        using assms neg neq0 by simp
      then show ?thesis
      proof (cases int-signed-value b v = 0)
        case True
        then have v = 0
       by (metis One-nat-def Suc-pred assms(1) assms(2) dual-order.refl int-signed-value.simps
signed-eq-0-iff take-bit-of-0 take-bit-signed-take-bit)
        then show ?thesis
          using neq0 by simp
      next
        case False
        then have int-signed-value b v < 0
          using ‹int-signed-value (b::nat) (v::64 word) ⊑ (0::int)› by linarith
        then have new-int b v' = new-int b (−v)
          using assms using intval-abs.elims
          by simp
        then have 0 < int-signed-value b (−v)
          using assms(3) invert-intval
         using ‹int-signed-value (b::nat) (v::64 word) < (0::int)› assms(1) assms(2)
by blast
        then show ?thesis
             using ‹new-int (b::nat) (v'::64 word) = new-int b (− (v::64 word))›
assms(1) signed-eqiv zero by presburger
      qed
    qed
    then show ?thesis
      by simp
  qed
qed

lemma intval-abs-elims:
  assumes intval-abs x ≠ UndefVal
  shows ∃ t v . x = IntVal t v ∧
                intval-abs x = new-int t (if int-signed-value t v < 0 then − v else v)
  by (meson intval-abs.elims assms)

lemma wf-abs-new-int:
  assumes intval-abs (IntVal t v) ≠ UndefVal
  shows intval-abs (IntVal t v) = new-int t v ∨ intval-abs (IntVal t v) = new-int
t (−v)
  by simp
```

```
lemma mono-undef-abs:
  assumes intval-abs (intval-abs x) ≠ UndefVal
  shows intval-abs x ≠ UndefVal
  using assms by force


lemma val-abs-idem:
  assumes valid-value x (IntegerStamp b l h)
  assumes val[abs(abs(x))] ≠ UndefVal
  shows val[abs(abs(x))] = val[abs x]
proof −
  obtain b v where in-def: x = IntVal b v
    using assms intval-abs-elims mono-undef-abs by blast
  then have bInRange: b > 0 ∧ b ≤ 64
    using assms(1)
    by (metis valid-stamp.simps(1) valid-value.simps(1))
  then show ?thesis
  proof (cases int-signed-value b v < 0)
    case neg: True
    then show ?thesis
    proof (cases v = (2^(b − 1)))
      case min: True
      then show ?thesis
      by (smt (z3) assms(1) bInRange in-def intval-abs.simps(1) intval-negate.simps(1)
negate-max-negative new-int.simps valid-value.simps(1))
    next
      case notMin: False
      then have nested: (intval-abs x) = new-int b (−v)
        using neg val-abs-neg in-def by simp
      also have int-signed-value b (−v) > 0
        using neg notMin invert-intval bInRange
        by (metis assms(1) in-def valid-value.simps(1))
      then have (intval-abs (new-int b (−v))) = new-int b (−v)
      by (smt (verit, best) ValueThms.signed-take-take-bit bInRange int-signed-value.simps
intval-abs.simps(1) new-int.simps new-int-unused-bits-zero)
      then show ?thesis
        using nested by presburger
    qed
  next
    case False
    then show ?thesis
    by (metis (mono-tags, lifting) assms(1) in-def intval-abs.simps(1) new-int.simps
valid-value.simps(1))
  qed
qed
```

Optimisations **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*
  by (*simp add*: *intval-add-sym*)

optimization *AddShiftConstantRight*: ((*const v*) + *y*) ⟼ *y* + (*const v*) *when* ¬(*is-ConstantExpr y*)
  apply (*metis add-2-eq-Suc′ less-Suc-eq plus-1-eq-Suc size.simps*(*11*) *size-non-add*)
  using *le-expr-def binadd-commute* **by** *blast*

optimization *AddShiftConstantRight2*: ((*const v*) + *y*) ⟼ *y* + (*const v*) *when* ¬(*is-ConstantExpr y*)
  using *AddShiftConstantRight* **by** *auto*

lemma *is-neutral-0* [*simp*]:
  assumes *val*[(*IntVal b x*) + (*IntVal b 0*)] ≠ *UndefVal*
  shows *val*[(*IntVal b x*) + (*IntVal b 0*)] = (*new-int b x*)
  by *simp*

lemma *AddNeutral-Exp*:
  shows *exp*[(*e* + (*const* (*IntVal 32 0*)))] ≥ *exp*[*e*]
  apply *auto*
  subgoal premises *p* **for** *m p x*
  proof −
    obtain *ev* **where** *ev*: [*m*,*p*] ⊢ *e* ↦ *ev*
      using *p* **by** *auto*
    then obtain *b evx* **where** *evx*: *ev* = *IntVal b evx*
    by (*metis evalDet evaltree-not-undef intval-add.simps*(*3*,*4*,*5*) *intval-logic-negation.cases p*(*1*,*2*))
    then have *additionNotUndef*: *val*[*ev* + (*IntVal 32 0*)] ≠ *UndefVal*
      using *p evalDet ev* **by** *blast*
    then have *sameWidth*: *b* = *32*
      by (*metis evx additionNotUndef intval-add.simps*(*1*))
    then have *unfolded*: *val*[*ev* + (*IntVal 32 0*)] = *IntVal 32* (*take-bit 32* (*evx*+*0*))
      by (*simp add*: *evx*)

then have *eqE*: *IntVal 32* (*take-bit 32* (*evx*+*0*)) = *IntVal 32* (*take-bit 32* (*evx*))
by *auto*
then show *?thesis*
by (*metis ev evalDet eval-unused-bits-zero evx p*(*1*) *sameWidth unfolded*)
qed
done

optimization *AddNeutral*: (*e* + (*const* (*IntVal 32 0*))) ⟼ *e*
using *AddNeutral-Exp* **by** *presburger*

ML-val ‹@{*term* ‹*x* = *y*›}›

lemma *NeutralLeftSubVal*:
assumes *e1* = *new-int b ival*
shows *val*[(*e1* − *e2*) + *e2*] ≈ *e1*
using *assms* **by** (*cases e1*; *cases e2*; *auto*)

lemma *RedundantSubAdd-Exp*:
shows *exp*[((*a* − *b*) + *b*)] ≥ *a*
apply *auto*
subgoal premises *p* **for** *m p y xa ya*
proof −
obtain *bv* **where** *bv*: [*m*,*p*] ⊢ *b* ↦ *bv*
using *p*(*1*) **by** *auto*
obtain *av* **where** *av*: [*m*,*p*] ⊢ *a* ↦ *av*
using *p*(*3*) **by** *auto*
then have *subNotUndef*: *val*[*av* − *bv*] ≠ *UndefVal*
by (*metis bv evalDet p*(*3*,*4*,*5*))
then obtain *bb bvv* **where** *bInt*: *bv* = *IntVal bb bvv*
by (*metis bv evaltree-not-undef intval-logic-negation.cases intval-sub.simps*(*7*,*8*,*9*))
then obtain *ba avv* **where** *aInt*: *av* = *IntVal ba avv*
by (*metis av evaltree-not-undef intval-logic-negation.cases intval-sub.simps*(*3*,*4*,*5*)
subNotUndef)
then have *widthSame*: *bb*=*ba*
by (*metis av bInt bv evalDet intval-sub.simps*(*1*) *new-int-bin.simps p*(*3*,*4*,*5*))
then have *valEval*: *val*[((*av*−*bv*)+*bv*)] = *val*[*av*]
using *aInt av eval-unused-bits-zero widthSame bInt* **by** *simp*
then show *?thesis*
by (*metis av bv evalDet p*(*1*,*3*,*4*))
qed
done

optimization *RedundantSubAdd*: ((e_1 − e_2) + e_2) ⟼ e_1
using *RedundantSubAdd-Exp* **by** *blast*

lemma *allE2*: (∀ *x y*. *P x y*) ⟹ (*P a b* ⟹ *R*) ⟹ *R*
by *simp*

lemma *just-goal2*:
assumes $(\forall\ a\ b.\ (val[(a - b) + b] \neq UndefVal \wedge a \neq UndefVal \longrightarrow val[(a - b) + b] = a))$
shows $(exp[(e_1 - e_2) + e_2]) \geq e_1$
unfolding *le-expr-def unfold-binary bin-eval.simps* **by** (*metis assms evalDet eval-tree-not-undef*)

optimization *RedundantSubAdd2*: $e_2 + (e_1 - e_2) \longmapsto e_1$
using *size-binary-rhs-a* **apply** *simp* **apply** *auto*
by (*smt* (*z3*) *NeutralLeftSubVal evalDet eval-unused-bits-zero intval-add-sym intval-sub.elims new-int.simps well-formed-equal-defn*)

lemma *AddToSubHelperLowLevel*:
shows $val[-e + y] = val[y - e]$ (**is** $?x = ?y$)
by (*induction y*; *induction e*; *auto*)

print-phases

lemma *val-redundant-add-sub*:
assumes $a = new\text{-}int\ bb\ ival$
assumes $val[b + a] \neq UndefVal$
shows $val[(b + a) - b] = a$
using *assms* **apply** (*cases a*; *cases b*; *auto*) **by** *presburger*

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

lemma *exp-add-left-negate-to-sub*:
$exp[-e + y] \geq exp[y - e]$
by (*cases e*; *cases y*; *auto simp*: *AddToSubHelperLowLevel*)

lemma *RedundantAddSub-Exp*:
shows $exp[(b + a) - b] \geq a$
apply *auto*
subgoal premises *p* **for** *m p y xa ya*
proof −

obtain *bv* **where** *bv*: $[m,p] \vdash b \mapsto bv$
using *p(1)* **by** *auto*
obtain *av* **where** *av*: $[m,p] \vdash a \mapsto av$
using *p(4)* **by** *auto*
then have *addNotUndef*: $val[av + bv] \neq UndefVal$
by (*metis bv evalDet intval-add-sym intval-sub.simps(2) p(2,3,4)*)
then obtain *bb bvv* **where** *bInt*: $bv = IntVal\ bb\ bvv$
by (*metis bv evalDet evaltree-not-undef intval-add.simps(3,5) intval-logic-negation.cases intval-sub.simps(8) p(1,2,3,5)*)
then obtain *ba avv* **where** *aInt*: $av = IntVal\ ba\ avv$
by (*metis addNotUndef intval-add.simps(2,3,4,5) intval-logic-negation.cases*)
then have *widthSame*: $bb=ba$
by (*metis addNotUndef bInt intval-add.simps(1)*)
then have *valEval*: $val[((bv+av)-bv)] = val[av]$
using *aInt av eval-unused-bits-zero widthSame bInt* **by** *simp*
then show *?thesis*
by (*metis av bv evalDet p(1,3,4)*)
qed
done

Optimisations

optimization *RedundantAddSub*: $(b + a) - b \longmapsto a$
using *RedundantAddSub-Exp* **by** *blast*

optimization *AddRightNegateToSub*: $x + -e \longmapsto x - e$
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*: $-e + y \longmapsto y - e$
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** *simp*

end

end

1.3 AndNode Phase

theory *AndPhase*
imports
Common

```
  Proofs.StampEvalThms
begin

context stamp-mask
begin

lemma AndCommute-Val:
  assumes val[x & y] ≠ UndefVal
  shows val[x & y] = val[y & x]
  using assms apply (cases x; cases y; auto) by (simp add: and.commute)

lemma AndCommute-Exp:
  shows exp[x & y] ≥ exp[y & x]
  using AndCommute-Val unfold-binary by auto

lemma AndRightFallthrough: (((and (not (↓ x)) (↑ y)) = 0)) ⟶ exp[x & y] ≥
exp[y]
  apply simp apply (rule impI; (rule allI)+; rule impI)
  subgoal premises p for m p v
    proof −
      obtain xv where xv: [m, p] ⊢ x ↦ xv
        using p(2) by blast
      obtain yv where yv: [m, p] ⊢ y ↦ yv
        using p(2) by blast
      obtain xb xvv where xvv: xv = IntVal xb xvv
          by (metis bin-eval-inputs-are-ints bin-eval-int evalDet is-IntVal-def p(2)
unfold-binary xv)
      obtain yb yvv where yvv: yv = IntVal yb yvv
          by (metis bin-eval-inputs-are-ints bin-eval-int evalDet is-IntVal-def p(2)
unfold-binary yv)
      have equalAnd: v = val[xv & yv]
        by (metis BinaryExprE bin-eval.simps(6) evalDet p(2) xv yv)
     then have andUnfold: val[xv & yv] = (if xb=yb then new-int xb (and xvv yvv)
else UndefVal)
        by (simp add: xvv yvv)
      have v = yv
        apply (cases v; cases yv; auto)
        using p(2) apply auto[1] using yvv apply simp-all
        by (metis Value.distinct(1,3,5,7,9,11,13) Value.inject(1) andUnfold equa-
lAnd new-int.simps
          xv xvv yv eval-unused-bits-zero new-int.simps not-down-up-mask-and-zero-implies-zero
            equalAnd p(1))+
      then show ?thesis
        by (simp add: yv)
    qed
  done

lemma AndLeftFallthrough: (((and (not (↓ y)) (↑ x)) = 0)) ⟶ exp[x & y] ≥
exp[x]
```

using *AndRightFallthrough AndCommute-Exp* **by** *simp*

end

phase *AndNode*
terminating *size*
begin

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

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

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

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

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

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

lemma *exp-and-equal*:
$exp[x \ \& \ x] \geq exp[x]$
apply *auto*
subgoal premises *p* **for** *m p xv yv*
proof−

```
    obtain xv where xv: [m,p] ⊢ x ↦ xv
      using p(1) by auto
    obtain yv where yv: [m,p] ⊢ x ↦ yv
      using p(1) by auto
    then have evalSame: xv = yv
      using evalDet xv by auto
    then have notUndef: xv ≠ UndefVal ∧ yv ≠ UndefVal
      using evaltree-not-undef xv by blast
    then have andNotUndef: val[xv & yv] ≠ UndefVal
      by (metis evalDet evalSame p(1,2,3) xv)
    obtain xb xvv where xvv: xv = IntVal xb xvv
       by (metis Value.exhaust-sel andNotUndef evalSame intval-and.simps(3,4,9)
notUndef)
    obtain yb yvv where yvv: yv = IntVal yb yvv
      using evalSame xvv by auto
    then have widthSame: xb=yb
      using evalSame xvv by auto
    then have valSame: yvv=xvv
      using evalSame xvv yvv by blast
    then have evalSame0: val[xv & yv] = new-int xb (xvv)
      using evalSame xvv by auto
    then show ?thesis
      by (metis eval-unused-bits-zero new-int.simps evalDet p(1,2) valSame width-
Same xv xvv yvv)
  qed
  done

lemma exp-and-nots:
  exp[~x & ~y] ≥ exp[~(x | y)]
   using val-and-nots by force

lemma exp-sign-extend:
  assumes e = (1 << 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 notUndef: 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: notUndef) defer
      using 2 apply fastforce+
      sorry
    then show ?thesis
      by (metis evalDet p(2) va)
  qed
 done

lemma exp-and-neutral:
 assumes wf-stamp x
 assumes stamp-expr x = IntegerStamp b lo hi
 shows exp[(x & ~(const (IntVal b 0)))] ≥ x
 using assms apply auto
 subgoal premises p for m p xa
 proof−
  obtain xv where xv: [m,p] ⊢ x ↦ xv
    using p(3) by auto
  obtain xb xvv where xvv: xv = IntVal xb xvv
    by (metis assms valid-int wf-stamp-def xv)
  then have widthSame: xb=b
    by (metis p(1,2) valid-int-same-bits wf-stamp-def xv)
  then show ?thesis
     by (metis evalDet eval-unused-bits-zero intval-and.simps(1) new-int.elims
new-int-bin.elims
       p(3) take-bit-eq-mask xv xvv)
 qed
 done

lemma val-and-commute[simp]:
  val[x & y] = val[y & x]
 by (cases x; cases y; auto simp: 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*)
by (*metis add-2-eq-Suc′ less-SucI less-add-Suc1 not-less-eq size-binary-const size-non-add exp-and-nots*)+

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* & ~(*const* (*IntVal b 0*))) ⟼ *x*
when (*wf-stamp x* ∧ *stamp-expr x* = *IntegerStamp b lo hi*)
using *exp-and-neutral* **by** *fast*

optimization *AndRightFallThrough*: (*x* & *y*) ⟼ *y*
when (((*and* (*not* (*IRExpr-down x*)) (*IRExpr-up y*)) = *0*))
by (*simp add*: *IRExpr-down-def IRExpr-up-def*)

optimization *AndLeftFallThrough*: (*x* & *y*) ⟼ *x*
when (((*and* (*not* (*IRExpr-down y*)) (*IRExpr-up x*)) = *0*))
by (*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*: *BinaryExpr op* (*const v1*) (*const v2*) ⟼ *ConstantExpr* (*bin-eval op v1 v2*)
unfolding *le-expr-def*
apply (*rule allI impI*)+

```
  subgoal premises bin for m p v
    apply (rule BinaryExprE[OF bin])
    subgoal premises prems for x y
    proof -
      have x: x = v1
        using prems by auto
      have y: y = v2
        using prems by auto
      have xy: v = bin-eval op x y
        by (simp add: prems x y)
      have int: ∃ b vv . v = new-int b vv
        using bin-eval-new-int prems by fast
      show ?thesis
         by (metis ConstantExpr prems(1) x y int bin eval-bits-1-64 new-int.simps
new-int-take-bits
            wf-value-def validDefIntConst)
      qed
    done
  done

end

end
```

1.5 ConditionalNode Phase

```
theory ConditionalPhase
  imports
    Common
    Proofs.StampEvalThms
begin

phase ConditionalNode
  terminating size
begin

lemma negates: ∃ v b. e = IntVal b v ∧ b > 0 ⟹ val-to-bool (val[e]) ⟷
¬(val-to-bool (val[!e]))
  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 = IntVal b ie
  assumes 0 < b
  shows val[(!e) ? x : y] = val[e ? y : x]
  by (metis assms intval-conditional.simps negates)

lemma negation-preserve-eval:
```

assumes $[m,\ p] \vdash exp[!e] \mapsto v$
shows $\exists\, v'.\ ([m,\ p] \vdash exp[e] \mapsto v') \wedge v = val[!v']$
using *assms* **by** *auto*

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

optimization *NegateConditionFlipBranches*: ((!*e*) *?* *x* : *y*) $\longmapsto$ (*e* *?* *y* : *x*)
apply *simp* **apply** (*rule allI*; *rule allI*; *rule allI*; *rule impI*)
subgoal premises *p* **for** *m p v*
proof −
obtain *ev* **where** *ev*: $[m,p] \vdash e \mapsto ev$
using *p* **by** *blast*
obtain *notEv* **where** *notEv*: *notEv* = *intval-logic-negation ev*
by *simp*
obtain *lhs* **where** *lhs*: $[m,p] \vdash$ *ConditionalExpr* (*UnaryExpr UnaryLogicNegation e*) *x y* $\mapsto$ *lhs*
using *p* **by** *auto*
obtain *xv* **where** *xv*: $[m,p] \vdash x \mapsto xv$
using *lhs* **by** *blast*
obtain *yv* **where** *yv*: $[m,p] \vdash y \mapsto yv$
using *lhs* **by** *blast*
then show *?thesis*
by (*smt* (*z3*) *le-expr-def ConditionalExpr ConditionalExprE Value.distinct*(*1*) *evalDet negates p*
negation-preserve-eval negation-preserve-eval-intval)
qed
done

optimization *DefaultTrueBranch*: (*true* *?* *x* : *y*) $\longmapsto$ *x* **.**

optimization *DefaultFalseBranch*: (*false* *?* *x* : *y*) $\longmapsto$ *y* **.**

optimization *ConditionalEqualBranches*: (*e* *?* *x* : *x*) $\longmapsto$ *x* **.**

optimization *condition-bounds-x*: (($u < v$) *?* *x* : *y*) $\longmapsto$ *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*: (($u < v$) *?* *x* : *y*) $\longmapsto$ *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-optimise-integer-test*:
assumes ∃ *v*. *x* = *IntVal 32 v*
shows *val*[((*x* & (*IntVal 32 1*)) *eq* (*IntVal 32 0*)) *?* (*IntVal 32 0*) : (*IntVal 32 1*)] =
val[*x* & *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*) *bool-to-val.simps*(*1*) *val-to-bool.simps*(*1*) *even-iff-mod-2-eq-zero odd-iff-mod-2-eq-one and-one-eq*)

optimization *ConditionalEliminateKnownLess*: ((*x* < *y*) *?* *x* : *y*) ⟼ *x*
when (*stamp-under* (*stamp-expr x*) (*stamp-expr y*)
∧ *wf-stamp x* ∧ *wf-stamp y*)
using *stamp-under-defn* **by** *fastforce*

lemma *ExpIntBecomesIntVal*:
assumes *stamp-expr x* = *IntegerStamp b xl xh*
assumes *wf-stamp x*
assumes *valid-value v* (*IntegerStamp b xl xh*)
assumes [*m*,*p*] ⊢ *x* ↦ *v*
shows ∃ *xv*. *v* = *IntVal b xv*
using *assms* **by** (*simp add*: *IRTreeEvalThms.valid-value-elims*(*3*))

lemma *intval-self-is-true*:
assumes *yv* ≠ *UndefVal*
assumes *yv* = *IntVal b yvv*
shows *intval-equals yv yv* = *IntVal 32 1*
using *assms* **by** (*cases yv*; *auto*)

lemma *intval-commute*:
assumes *intval-equals yv xv* ≠ *UndefVal*
assumes *intval-equals xv yv* ≠ *UndefVal*
shows *intval-equals yv xv* = *intval-equals xv yv*
using *assms* **apply** (*cases yv*; *cases xv*; *auto*) **by** (*smt* (*verit*, *best*))

definition *isBoolean* :: *IRExpr* ⇒ *bool* **where**
isBoolean e = (∀ *m p cond*. (([*m*,*p*] ⊢ *e* ↦ *cond*) ⟶ (*cond* ∈ {*IntVal 32 0*, *IntVal 32 1*})))

lemma *preserveBoolean*:
assumes *isBoolean c*
shows *isBoolean exp*[!*c*]
using *assms isBoolean-def* **apply** *auto*
by (*metis* (*no-types*, *lifting*) *IntVal0 IntVal1 intval-logic-negation.simps*(*1*) *logic-negate-def*)

optimization *ConditionalIntegerEquals-1*: *exp*[*BinaryExpr BinIntegerEquals* (*c ? x* : *y*) (*x*)] ⟼ *c*

```
                                    when stamp-expr x = IntegerStamp b xl xh ∧
wf-stamp x ∧
                                       stamp-expr y = IntegerStamp b yl yh ∧
wf-stamp y ∧
                                    (alwaysDistinct (stamp-expr x) (stamp-expr
y)) ∧
                                    isBoolean c
  apply (metis Canonicalization.cond-size add-lessD1 size-binary-lhs) apply auto
  subgoal premises p for m p cExpr xv cond
  proof -
    obtain cond where cond: [m,p] ⊢ c ↦ cond
      using p by blast
    have cRange: cond = IntVal 32 0 ∨ cond = IntVal 32 1
      using p cond isBoolean-def by blast
    then obtain yv where yVal: [m,p] ⊢ y ↦ yv
      using p(15) by auto
    obtain xvv where xvv: xv = IntVal b xvv
      by (metis p(1,2,7) valid-int wf-stamp-def)
    obtain yvv where yvv: yv = IntVal b yvv
      by (metis ExpIntBecomesIntVal p(3,4) wf-stamp-def yVal)
    have yxDiff: xvv ≠ yvv
      by (smt (verit, del-insts) yVal xvv wf-stamp-def valid-int-signed-range p yvv)
    have eqEvalFalse: intval-equals yv xv = (IntVal 32 0)
       unfolding xvv yvv apply auto by (metis (mono-tags) bool-to-val.simps(2)
yxDiff)
    then have valEvalSame: cond = intval-equals val[cond ? xv : yv] xv
      apply (cases cond = IntVal 32 0; simp) using cRange xvv by auto
    then have condTrue: val-to-bool cond ⟹ cExpr = xv
      by (metis (mono-tags, lifting) cond evalDet p(11) p(7) p(9))
    then have condFalse: ¬(val-to-bool cond) ⟹ cExpr = yv
      by (metis (full-types) cond evalDet p(11) p(9) yVal)
    then have [m,p] ⊢ c ↦ intval-equals cExpr xv
      using cond condTrue valEvalSame by fastforce
    then show ?thesis
      by blast
  qed
  done


lemma negation-preserve-eval0:
  assumes [m, p] ⊢ exp[e] ↦ v
  assumes isBoolean e
  shows ∃ v'. ([m, p] ⊢ exp[!e] ↦ v')
  using assms
proof -
  obtain b vv where vIntVal: v = IntVal b vv
    using isBoolean-def assms by blast
  then have negationDefined: intval-logic-negation v ≠ UndefVal
    by simp
```

```
  show ?thesis
    using assms(1) negationDefined by fastforce
qed

lemma negation-preserve-eval2:
  assumes ([m, p] ⊢ exp[e] ↦ v)
  assumes (isBoolean e)
  shows ∃ v'. ([m, p] ⊢ exp[!e] ↦ v') ∧ v = val[!v']
  using assms
proof −
  obtain notEval where notEval: ([m, p] ⊢ exp[!e] ↦ notEval)
    by (metis assms negation-preserve-eval0)
  then have logicNegateEquiv: notEval = intval-logic-negation v
    using evalDet assms(1) unary-eval.simps(4) by blast
  then have vRange: v = IntVal 32 0 ∨ v = IntVal 32 1
    using assms by (auto simp add: isBoolean-def)
  have evaluateNot: v = intval-logic-negation notEval
   by (metis IntVal0 IntVal1 intval-logic-negation.simps(1) logicNegateEquiv logic-negate-def
      vRange)
  then show ?thesis
    using notEval by auto
qed

optimization ConditionalIntegerEquals-2: exp[BinaryExpr BinIntegerEquals (c ?
x : y) (y)] ⟼ (!c)
                                        when stamp-expr x = IntegerStamp b xl xh ∧
wf-stamp x ∧
                                            stamp-expr y = IntegerStamp b yl yh ∧
wf-stamp y ∧
                                         (alwaysDistinct (stamp-expr x) (stamp-expr
y)) ∧
                                         isBoolean c
 apply (smt (verit) not-add-less1 max-less-iff-conj max.absorb3 linorder-less-linear
add-2-eq-Suc'
        add-less-cancel-right size-binary-lhs add-lessD1 Canonicalization.cond-size)
  apply auto
  subgoal premises p for m p cExpr yv cond trE faE
  proof −
    obtain cond where cond: [m,p] ⊢ c ↦ cond
      using p by blast
    then have condNotUndef: cond ≠ UndefVal
      by (simp add: evaltree-not-undef)
    then obtain notCond where notCond: [m,p] ⊢ exp[!c] ↦ notCond
      by (meson p(6) negation-preserve-eval2 cond)
    have cRange: cond = IntVal 32 0 ∨ cond = IntVal 32 1
      using p cond by (simp add: isBoolean-def)
    then have cNotRange: notCond = IntVal 32 0 ∨ notCond = IntVal 32 1
     by (metis (no-types, lifting) IntVal0 IntVal1 cond evalDet intval-logic-negation.simps(1)
        logic-negate-def negation-preserve-eval notCond)
```

then obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *x* ↦ *xv*
using *p* **by** *auto*
then have *trueCond*: (*notCond* = *IntVal 32 1*) ⟹ [*m*,*p*] ⊢ (*ConditionalExpr c x y*) ↦ *yv*
by (*smt* (*verit*, *best*) *cRange evalDet negates negation-preserve-eval notCond p*(*7*) *cond*
zero-less-numeral val-to-bool.simps(*1*) *evaltree-not-undef ConditionalExpr ConditionalExprE*)
obtain *xvv* **where** *xvv*: *xv* = *IntVal b xvv*
by (*metis p*(*1*,*2*) *valid-int wf-stamp-def xv*)
then have *opposites*: *notCond* = *intval-logic-negation cond*
by (*metis cond evalDet negation-preserve-eval notCond*)
then have *negate*: (*intval-logic-negation cond* = *IntVal 32 0*) ⟹ (*cond* = *IntVal 32 1*)
using *cRange intval-logic-negation.simps negates* **by** *fastforce*
have *falseCond*: (*notCond* = *IntVal 32 0*) ⟹ [*m*,*p*] ⊢ (*ConditionalExpr c x y*) ↦ *xv*
unfolding *opposites* **using** *negate cond evalDet p*(*13*,*14*,*15*,*16*) *xv* **by** *auto*
obtain *yvv* **where** *yvv*: *yv* = *IntVal b yvv*
by (*metis p*(*3*,*4*,*7*) *wf-stamp-def ExpIntBecomesIntVal*)
have *yxDiff*: *xv* ≠ *yv*
by (*metis linorder-not-less max.absorb1 max.absorb4 max-less-iff-conj min-def xv yvv*
wf-stamp-def valid-int-signed-range p(*1*,*2*,*3*,*4*,*5*,*7*))
then have *trueEvalCond*: (*cond* = *IntVal 32 0*) ⟹
[*m*,*p*] ⊢ *exp*[*BinaryExpr BinIntegerEquals* (*c ? x* : *y*) (*y*)]
↦ *intval-equals yv yv*
by (*smt* (*verit*) *cNotRange trueCond ConditionalExprE cond bin-eval.simps*(*13*) *evalDet p*
falseCond unfold-binary val-to-bool.simps(*1*))
then have *falseEval*: (*notCond* = *IntVal 32 0*) ⟹
[*m*,*p*] ⊢ *exp*[*BinaryExpr BinIntegerEquals* (*c ? x* : *y*) (*y*)]
↦ *intval-equals xv yv*
using *p* **by** (*metis ConditionalExprE bin-eval.simps*(*13*) *evalDet falseCond unfold-binary*)
have *eqEvalFalse*: *intval-equals yv xv* = (*IntVal 32 0*)
unfolding *xvv yvv* **apply** *auto* **by** (*metis* (*mono-tags*) *bool-to-val.simps*(*2*) *yxDiff yvv xvv*)
have *trueEvalEquiv*: [*m*,*p*] ⊢ *exp*[*BinaryExpr BinIntegerEquals* (*c ? x* : *y*) (*y*)] ↦ *notCond*
apply (*cases notCond*) **prefer** *2*
apply (*metis IntVal0 Value.distinct*(*1*) *eqEvalFalse evalDet evaltree-not-undef falseEval p*(*6*)
intval-commute intval-logic-negation.simps(*1*) *intval-self-is-true logic-negate-def*
negation-preserve-eval2 notCond trueEvalCond yvv cNotRange cond)
using *notCond cNotRange* **by** *auto*
show *?thesis*
using *ConditionalExprE*
by (*metis cNotRange falseEval notCond trueEvalEquiv trueCond falseCond*

intval-self-is-true
yvv p(*9*,*11*) *evalDet*)
qed
done

optimization *ConditionalExtractCondition*: *exp*[(*c ? true* : *false*)] ⟼ *c*
when isBoolean c
using *isBoolean-def* **by** *fastforce*

optimization *ConditionalExtractCondition2*: *exp*[(*c ? false* : *true*)] ⟼ !*c*
when isBoolean c
apply *auto*
subgoal premises *p* **for** *m p cExpr cond*
proof−
obtain *cond* **where** *cond*: [*m*,*p*] ⊢ *c* ↦ *cond*
using *p*(*2*) **by** *auto*
obtain *notCond* **where** *notCond*: [*m*,*p*] ⊢ *exp*[!*c*] ↦ *notCond*
by (*metis cond negation-preserve-eval2 p*(*1*))
then have *cRange*: *cond* = *IntVal 32 0* ∨ *cond* = *IntVal 32 1*
using *isBoolean-def cond p*(*1*) **by** *auto*
then have *cExprRange*: *cExpr* = *IntVal 32 0* ∨ *cExpr* = *IntVal 32 1*
by (*metis* (*full-types*) *ConstantExprE p*(*4*))
then have *condTrue*: *cond* = *IntVal 32 1* ⟹ *cExpr* = *IntVal 32 0*
using *cond evalDet p*(*2*) *p*(*4*) **by** *fastforce*
then have *condFalse*: *cond* = *IntVal 32 0* ⟹ *cExpr* = *IntVal 32 1*
using *p cond evalDet* **by** *fastforce*
then have *opposite*: *cond* = *intval-logic-negation cExpr*
by (*metis* (*full-types*) *IntVal0 IntVal1 cRange condTrue intval-logic-negation.simps*(*1*)
logic-negate-def)
then have *eq*: *notCond* = *cExpr*
by (*metis* (*no-types*, *lifting*) *IntVal0 IntVal1 cExprRange cond evalDet nega-
tion-preserve-eval*
intval-logic-negation.simps(*1*) *logic-negate-def notCond*)
then show *?thesis*
using *notCond* **by** *auto*
qed
done

optimization *ConditionalEqualIsRHS*: ((*x eq y*) *? x* : *y*) ⟼ *y*
apply *auto*
subgoal premises *p* **for** *m p v true false xa ya*
proof−
obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *x* ↦ *xv*
using *p*(*8*) **by** *auto*
obtain *yv* **where** *yv*: [*m*,*p*] ⊢ *y* ↦ *yv*
using *p*(*9*) **by** *auto*
have *notUndef*: *xv* ≠ *UndefVal* ∧ *yv* ≠ *UndefVal*
using *evaltree-not-undef xv yv* **by** *blast*
have *evalNotUndef*: *intval-equals xv yv* ≠ *UndefVal*

```
      by (metis evalDet p(1,8,9) xv yv)
    obtain xb xvv where xvv: xv = IntVal xb xvv
      by (metis Value.exhaust evalNotUndef intval-equals.simps(3,4,5) notUndef)
    obtain yb yvv where yvv: yv = IntVal yb yvv
      by (metis evalNotUndef intval-equals.simps(7,8,9) intval-logic-negation.cases
notUndef)
     obtain vv where evalLHS: [m,p] ⊢ if val-to-bool (intval-equals xv yv) then x
else y ↦ vv
      by (metis (full-types) p(4) yv)
    obtain equ where equ: equ = intval-equals xv yv
      by fastforce
    have trueEval: equ = IntVal 32 1 ⟹ vv = xv
      using evalLHS by (simp add: evalDet xv equ)
    have falseEval: equ = IntVal 32 0 ⟹ vv = yv
      using evalLHS by (simp add: evalDet yv equ)
    then have vv = v
      by (metis evalDet evalLHS p(2,8,9) xv yv)
    then show ?thesis
     by (metis (full-types) bool-to-val.simps(1,2) bool-to-val-bin.simps equ evalNo-
tUndef falseEval
          intval-equals.simps(1) trueEval xvv yv yvv)
  qed
  done


optimization normalizeX: ((x eq const (IntVal 32 0)) ?
                              (const (IntVal 32 0)) : (const (IntVal 32 1))) ⟼ x
                           when stamp-expr x = IntegerStamp 32 0 1 ∧ wf-stamp x ∧
                                 isBoolean x
  apply auto
  subgoal premises p for m p v
    proof -
      obtain xa where xa: [m,p] ⊢ x ↦ xa
        using p by blast
       have eval: [m,p] ⊢ if val-to-bool (intval-equals xa (IntVal 32 0))
                       then ConstantExpr (IntVal 32 0)
                       else ConstantExpr (IntVal 32 1) ↦ v
         using evalDet p(3,4,5,6,7) xa by blast
       then have xaRange: xa = IntVal 32 0 ∨ xa = IntVal 32 1
         using isBoolean-def p(3) xa by blast
      then have 6: v = xa
        using eval xaRange by auto
      then show ?thesis
        by (auto simp: xa)
    qed
  done


optimization normalizeX2: ((x eq (const (IntVal 32 1))) ?
```

(*const* (*IntVal 32 1*)) : (*const* (*IntVal 32 0*))) ⟼ *x*
when (*x* = *ConstantExpr* (*IntVal 32 0*) |
(*x* = *ConstantExpr* (*IntVal 32 1*))) **.**

optimization *flipX*: ((*x eq* (*const* (*IntVal 32 0*))) *?*
(*const* (*IntVal 32 1*)) : (*const* (*IntVal 32 0*))) ⟼ *x* ⊕ (*const* (*IntVal 32 1*))
when (*x* = *ConstantExpr* (*IntVal 32 0*) |
(*x* = *ConstantExpr* (*IntVal 32 1*))) **.**

optimization *flipX2*: ((*x eq* (*const* (*IntVal 32 1*))) *?*
(*const* (*IntVal 32 0*)) : (*const* (*IntVal 32 1*))) ⟼ *x* ⊕ (*const* (*IntVal 32 1*))
when (*x* = *ConstantExpr* (*IntVal 32 0*) |
(*x* = *ConstantExpr* (*IntVal 32 1*))) **.**

lemma *stamp-of-default*:
assumes *stamp-expr x* = *default-stamp*
assumes *wf-stamp x*
shows ([*m*, *p*] ⊢ *x* ↦ *v*) ⟶ (∃ *vv*. *v* = *IntVal 32 vv*)
by (*metis assms default-stamp valid-value-elims*(*3*) *wf-stamp-def*)

optimization *OptimiseIntegerTest*:
(((*x* & (*const* (*IntVal 32 1*))) *eq* (*const* (*IntVal 32 0*))) *?*
(*const* (*IntVal 32 0*)) : (*const* (*IntVal 32 1*))) ⟼
x & (*const* (*IntVal 32 1*))
when (*stamp-expr x* = *default-stamp* ∧ *wf-stamp x*)
apply (*simp*; *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 *ConditionalExprE ConstantExprE bin-eval.simps*(*4*,*11*) *evalDet xv unfold-binary*
intval-conditional.simps
by *fastforce*
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*(*6*) *evalDet xv*)
have *lhs* = *rhs*
using *val-optimise-integer-test x32 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* ⇒ *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*]
  by (*cases x*; *cases y*; *auto simp*: *mergeTakeBit*)

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

by (*auto simp*: *assms*)

lemma *val-multiply-zero*:
assumes $x = new\text{-}int\ b\ v$
shows $val[x * (IntVal\ b\ 0)] = IntVal\ b\ 0$
by (*simp add*: *assms*)

lemma *val-multiply-negative*:
assumes $x = new\text{-}int\ b\ v$
shows $val[x * -(IntVal\ b\ 1)] = val[-x]$
unfolding *assms*(*1*) **apply** *auto*
by (*metis bin-multiply-negative mergeTakeBit take-bit-minus-one-eq-mask*)

lemma *val-MulPower2*:
fixes *i* :: *64 word*
assumes $y = IntVal\ 64\ (2\ \hat{}\ unat(i))$
and $0 < i$
and $i < 64$
and $val[x * y] \neq UndefVal$
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*) *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*))
by (*auto simp*: *63*)
qed
by *presburger*

lemma *val-MulPower2Add1*:
fixes *i* :: *64 word*
assumes $y = IntVal\ 64\ ((2\ \hat{}\ unat(i)) + 1)$
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*
by (*simp add*: *less-mask-eq p*(*6*))
then have *x2* ∗ (*2* ^ *unat i* + *1*) = (*x2* ∗ (*2* ^ *unat i*)) + *x2*
by (*simp add*: *distrib-left*)
then show *x2* ∗ (*2* ^ *unat i* + *1*) = *x2* << *unat* (*and i 63*) + *x2*
by (*simp add*: *63* ‹*and i* (*mask 6*) = *i*›)
qed
using *val-to-bool.simps*(*2*) **by** *presburger*

lemma *val-MulPower2Sub1*:
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*
by (*simp add*: *less-mask-eq p*(*6*))
then have *x2* ∗ (*2* ^ *unat i* − *1*) = (*x2* ∗ (*2* ^ *unat i*)) − *x2*
by (*simp add*: *right-diff-distrib′*)
then show *x2* ∗ (*2* ^ *unat i* − *1*) = *x2* << *unat* (*and i 63*) − *x2*
by (*simp add*: *63* ‹*and i* (*mask 6*) = *i*›)
qed
using *val-to-bool.simps*(*2*) **by** *presburger*

lemma *val-distribute-multiplication*:
assumes *x* = *IntVal b xx* ∧ *q* = *IntVal b qq* ∧ *a* = *IntVal b aa*
assumes *val*[*x* ∗ (*q* + *a*)] ≠ *UndefVal*
assumes *val*[(*x* ∗ *q*) + (*x* ∗ *a*)] ≠ *UndefVal*
shows *val*[*x* ∗ (*q* + *a*)] = *val*[(*x* ∗ *q*) + (*x* ∗ *a*)]
using *assms* **apply** (*cases x*; *cases q*; *cases a*; *auto*)
by (*metis* (*no-types*, *opaque-lifting*) *distrib-left new-int.elims new-int-unused-bits-zero mergeTakeBit*)

```
lemma val-distribute-multiplication64:
  assumes x = new-int 64 xx ∧ q = new-int 64 qq ∧ a = new-int 64 aa
  shows val[x ∗ (q + a)] = val[(x ∗ q) + (x ∗ a)]
  using assms apply (cases x; cases q; cases a; auto)
  using distrib-left by blast


lemma val-MulPower2AddPower2:
  fixes i j :: 64 word
  assumes y = IntVal 64 ((2 ^ unat(i)) + (2 ^ unat(j)))
  and     0 < i
  and     0 < j
  and     i < 64
  and     j < 64
  and     x = new-int 64 xx
  shows   val[x ∗ y] = val[(x << IntVal 64 i) + (x << IntVal 64 j)]
  proof −
    have 63: (63 :: int64) = mask 6
      by eval
    then have (2 :: int) ^ 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)))]

      by 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-multiplication64 by simp
   then have 2: val[(x ∗ IntVal 64 (2 ^ unat(i)))] = val[x << IntVal 64 i]
       by (metis (no-types, opaque-lifting) Value.distinct(1) intval-mul.simps(1)
new-int.simps
        new-int-bin.simps assms(2,4,6) val-MulPower2)
   then show ?thesis
    by (metis (no-types, lifting) 1 Value.distinct(1) n intval-mul.simps(1) new-int-bin.elims
        new-int.simps val-MulPower2 assms(1,3,5,6))
   qed

thm-oracles val-MulPower2AddPower2


lemma exp-multiply-zero-64:
  shows exp[x ∗ (const (IntVal b 0))] ≥ ConstantExpr (IntVal b 0)
  apply auto
  subgoal premises p for m p xa
  proof −
    obtain xv where xv: [m,p] ⊢ x ↦ xv
```

using *p*(*1*) **by** *auto*
obtain *xb xvv* **where** *xvv*: *xv* = *IntVal xb xvv*
by (*metis evalDet p*(*1*,*2*) *xv evaltree-not-undef intval-is-null.cases intval-mul.simps*(*3*,*4*,*5*))
then have *evalNotUndef*: *val*[*xv* ∗ (*IntVal b 0*)] ≠ *UndefVal*
using *p evalDet xv* **by** *blast*
then have *mulUnfold*: *val*[*xv* ∗ (*IntVal b 0*)] = *IntVal xb* (*take-bit xb* (*xvv*∗*0*))
by (*metis new-int.simps xvv new-int-bin.simps intval-mul.simps*(*1*))
then have *isZero*: *val*[*xv* ∗ (*IntVal b 0*)] = (*new-int xb* (*0*))
by (*simp add*: *mulUnfold*)
then have *eq*: (*IntVal b 0*) = (*IntVal xb* (*0*))
by (*metis Value.distinct*(*1*) *intval-mul.simps*(*1*) *mulUnfold new-int-bin.elims xvv*)
then show *?thesis*
using *evalDet isZero p*(*1*,*3*) *xv* **by** *fastforce*
qed
done

lemma *exp-multiply-neutral*:
exp[*x* ∗ (*const* (*IntVal b 1*))] ≥ *x*
apply *auto*
subgoal premises *p* **for** *m p xa*
proof −
obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *x* ↦ *xv*
using *p*(*1*) **by** *auto*
obtain *xb xvv* **where** *xvv*: *xv* = *IntVal xb xvv*
by (*smt* (*z3*) *evalDet intval-mul.elims p*(*1*,*2*) *xv*)
then have *evalNotUndef*: *val*[*xv* ∗ (*IntVal b 1*)] ≠ *UndefVal*
using *p evalDet xv* **by** *blast*
then have *mulUnfold*: *val*[*xv* ∗ (*IntVal b 1*)] = *IntVal xb* (*take-bit xb* (*xvv*∗*1*))
by (*metis new-int.simps xvv new-int-bin.simps intval-mul.simps*(*1*))
then show *?thesis*
by (*metis bin-multiply-identity evalDet eval-unused-bits-zero p*(*1*) *xv xvv*)
qed
done

thm-oracles *exp-multiply-neutral*

lemma *exp-multiply-negative*:
exp[*x* ∗ −(*const* (*IntVal b 1*))] ≥ *exp*[−*x*]
apply *auto*
subgoal premises *p* **for** *m p xa*
proof −
obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *x* ↦ *xv*
using *p*(*1*) **by** *auto*
obtain *xb xvv* **where** *xvv*: *xv* = *IntVal xb xvv*
by (*metis array-length.cases evalDet evaltree-not-undef intval-mul.simps*(*3*,*4*,*5*) *p*(*1*,*2*) *xv*)
then have *rewrite*: *val*[−(*IntVal b 1*)] = *IntVal b* (*mask b*)
by *simp*

```
    then have evalNotUndef: val[xv ∗ −(IntVal b 1)] ≠ UndefVal
      unfolding rewrite using evalDet p(1,2) xv by blast
    then have mulUnfold: val[xv ∗ (IntVal b (mask b))] =
                        (if xb=b then (IntVal xb (take-bit xb (xvv∗(mask xb)))) else
UndefVal)
      by (metis new-int.simps xvv new-int-bin.simps intval-mul.simps(1))
    then have sameWidth: xb=b
      by (metis evalNotUndef rewrite)
    then show ?thesis
     by (metis evalDet eval-unused-bits-zero new-int.elims p(1,2) rewrite unary-eval.simps(2)
xvv
          unfold-unary val-multiply-negative xv)
  qed
  done

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 ConstantExprE equiv-exprs-def unfold-binary assms by fastforce

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 ConstantExprE equiv-exprs-def unfold-binary assms by fastforce

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 ConstantExprE equiv-exprs-def unfold-binary assms by fastforce

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 << ConstantExpr (IntVal 64 j))]*
using *ConstantExprE equiv-exprs-def unfold-binary assms* **by** *fastforce*

lemma *greaterConstant*:
fixes *a b :: 64 word*
assumes *a > b*
and *y = ConstantExpr (IntVal 32 a)*
and *x = ConstantExpr (IntVal 32 b)*
shows *exp[BinaryExpr BinIntegerLessThan y x] ≥ exp[const (new-int 32 0)]*
using *assms*
apply *simp* **unfolding** *equiv-exprs-def* **apply** *auto*
sorry

lemma *exp-distribute-multiplication*:
assumes *stamp-expr x = IntegerStamp b xl xh*
assumes *stamp-expr q = IntegerStamp b ql qh*
assumes *stamp-expr y = IntegerStamp b yl yh*
assumes *wf-stamp x*
assumes *wf-stamp q*
assumes *wf-stamp y*
shows *exp[(x * q) + (x * y)] ≥ exp[x * (q + y)]*
apply *auto*
subgoal premises *p* **for** *m p xa qa xb aa*
proof −
  obtain *xv* **where** *xv*: *[m,p] ⊢ x ↦ xv*
    using *p* **by** *simp*
  obtain *qv* **where** *qv*: *[m,p] ⊢ q ↦ qv*
    using *p* **by** *simp*
  obtain *yv* **where** *yv*: *[m,p] ⊢ y ↦ yv*
    using *p* **by** *simp*
  then obtain *xvv* **where** *xvv*: *xv = IntVal b xvv*
    by *(metis assms(1,4) valid-int wf-stamp-def xv)*
  then obtain *qvv* **where** *qvv*: *qv = IntVal b qvv*
    by *(metis qv valid-int assms(2,5) wf-stamp-def)*
  then obtain *yvv* **where** *yvv*: *yv = IntVal b yvv*
    by *(metis yv valid-int assms(3,6) wf-stamp-def)*
  then have *rhsDefined*: *val[xv * (qv + yv)] ≠ UndefVal*
    by *(simp add: xvv qvv)*
  have *val[xv * (qv + yv)] = val[(xv * qv) + (xv * yv)]*
    using *val-distribute-multiplication* **by** *(simp add: yvv qvv xvv)*
  then show *?thesis*

by (*metis bin-eval.simps*(*1*,*3*) *BinaryExpr p*(*1*,*2*,*3*,*5*,*6*) *qv xv evalDet yv qvv Value.distinct*(*1*)
yvv intval-add.simps(*1*))
qed
done

Optimisations

optimization *EliminateRedundantNegative*: $-x * -y \longmapsto x * y$
apply *auto*
by (*metis BinaryExpr val-eliminate-redundant-negative bin-eval.simps*(*3*))

optimization *MulNeutral*: *x* ∗ *ConstantExpr* (*IntVal b 1*) ⟼ *x*
using *exp-multiply-neutral* **by** *blast*

optimization *MulEliminator*: *x* ∗ *ConstantExpr* (*IntVal b 0*) ⟼ *const* (*IntVal b 0*)
using *exp-multiply-zero-64* **by** *fast*

optimization *MulNegate*: *x* ∗ −(*const* (*IntVal b 1*)) ⟼ −*x*
using *exp-multiply-negative* **by** *presburger*

fun *isNonZero* :: *Stamp* ⇒ *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*] ⊢ *x* ↦ *v*) ⟶ (∃ *vv b*. (*v* = *IntVal b vv* ∧ *val-to-bool val*[(*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 eval vdef xstamp wf-stamp-def* **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 signed-eq-0-iff take-bit-of-0 signed-above verit-comp-simplify1*(*1*) *word-gt-0 signed-take-bit-eq-if-positive*)
then show *?thesis*
using *vdef signed-above* **by** *simp*
qed
done

lemma *ExpIntBecomesIntValArbitrary*:
assumes *stamp-expr x* = *IntegerStamp b xl xh*
assumes *wf-stamp x*
assumes *valid-value v* (*IntegerStamp b xl xh*)
assumes [*m*,*p*] ⊢ *x* ↦ *v*
shows ∃ *xv*. *v* = *IntVal b xv*
using *assms* **by** (*simp add*: *IRTreeEvalThms.valid-value-elims*(*3*))

optimization *MulPower2*: *x* ∗ *y* ⟼ *x* << *const* (*IntVal 64 i*)
when (*i* > *0* ∧ *stamp-expr x* = *IntegerStamp 64 xl xh* ∧
wf-stamp x ∧
64 > *i* ∧
y = *exp*[*const* (*IntVal 64* (*2* ^ *unat*(*i*)))])
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*(*2*) **by** *blast*
then have *notUndef*: *xv* ≠ *UndefVal*
by (*simp add*: *evaltree-not-undef*)
obtain *xb xvv* **where** *xvv*: *xv* = *IntVal xb xvv*
by (*metis wf-stamp-def eval*(*1*) *ExpIntBecomesIntValArbitrary xv*)
then have *w64*: *xb* = *64*
by (*metis wf-stamp-def intval-bits.simps ExpIntBecomesIntValArbitrary xv eval*(*1*))
obtain *yv* **where** *yv*: [*m*, *p*] ⊢ *y* ↦ *yv*
using *eval*(*1*,*2*) **by** *blast*
then have *lhs*: [*m*, *p*] ⊢ *exp*[*x* ∗ *y*] ↦ *val*[*xv* ∗ *yv*]
by (*metis bin-eval.simps*(*3*) *eval*(*1*,*2*) *evalDet unfold-binary xv*)
have [*m*, *p*] ⊢ *exp*[*const* (*IntVal 64 i*)] ↦ *val*[(*IntVal 64 i*)]
by (*smt* (*verit*, *ccfv-SIG*) *ConstantExpr constantAsStamp.simps*(*1*) *eval-bits-1-64*
take-bit64 xv xvv
validStampIntConst wf-value-def valid-value.simps(*1*) *w64*)
then have *rhs*: [*m*, *p*] ⊢ *exp*[*x* << *const* (*IntVal 64 i*)] ↦ *val*[*xv* << (*IntVal 64*
i)]
by (*metis Value.simps*(*5*) *bin-eval.simps*(*10*) *intval-left-shift.simps*(*1*) *new-int.simps*
xv xvv
evaltree.BinaryExpr)
have *val*[*xv* ∗ *yv*] = *val*[*xv* << (*IntVal 64 i*)]
by (*metis ConstantExprE eval*(*1*) *evaltree-not-undef lhs yv val-MulPower2*)
then show *?thesis*
by (*metis eval*(*1*,*2*) *evalDet lhs rhs*)
qed
done

optimization *MulPower2Add1*: *x* ∗ *y* ⟼ (*x* << *const* (*IntVal 64 i*)) + *x*
when (*i* > *0* ∧ *stamp-expr x* = *IntegerStamp 64 xl xh* ∧
wf-stamp x ∧
64 > *i* ∧

y = *ConstantExpr* (*IntVal 64* ((*2* ^ *unat*(*i*)) + *1*)))
apply *simp* **apply** (*rule impI*; (*rule allI*)+; *rule impI*)
subgoal premises *p* **for** *m p v*
proof −
obtain *xv* **where** *xv*: [*m*, *p*] ⊢ *x* ↦ *xv*
using *p* **by** *fast*
then obtain *xvv* **where** *xvv*: *xv* = *IntVal 64 xvv*
using *p* **by** (*metis valid-int wf-stamp-def*)
obtain *yv* **where** *yv*: [*m*, *p*] ⊢ *y* ↦ *yv*
using *p* **by** *blast*
have *ygezero*: *y* > *ConstantExpr* (*IntVal 64 0*)
using *greaterConstant p wf-value-def* **sorry**
then have *1*: *0* < *i* ∧
i < *64* ∧
y = *ConstantExpr* (*IntVal 64* ((*2* ^ *unat*(*i*)) + *1*))
using *p* **by** *blast*
then have *lhs*: [*m*, *p*] ⊢ *exp*[*x* ∗ *y*] ↦ *val*[*xv* ∗ *yv*]
by (*metis bin-eval.simps*(*3*) *evalDet p*(*2*) *xv yv unfold-binary*)
then have [*m*, *p*] ⊢ *exp*[*const* (*IntVal 64 i*)] ↦ *val*[(*IntVal 64 i*)]
by (*metis wf-value-def verit-comp-simplify1*(*2*) *zero-less-numeral ConstantExpr take-bit64*
constantAsStamp.simps(*1*) *validStampIntConst valid-value.simps*(*1*))
then have *rhs2*: [*m*, *p*] ⊢ *exp*[*x* << *const* (*IntVal 64 i*)] ↦ *val*[*xv* << (*IntVal 64 i*)]
by (*metis Value.simps*(*5*) *bin-eval.simps*(*10*) *intval-left-shift.simps*(*1*) *new-int.simps xv xvv*
evaltree.BinaryExpr)
then have *rhs*: [*m*, *p*] ⊢ *exp*[(*x* << *const* (*IntVal 64 i*)) + *x*] ↦ *val*[(*xv* << (*IntVal 64 i*)) + *xv*]
by (*metis* (*no-types*, *lifting*) *intval-add.simps*(*1*) *bin-eval.simps*(*1*) *Value.simps*(*5*) *xv xvv*
evaltree.BinaryExpr intval-left-shift.simps(*1*) *new-int.simps*)
then have *simple*: *val*[*xv* ∗ (*IntVal 64* (*2* ^ *unat*(*i*)))] = *val*[*xv* << (*IntVal 64 i*)]
using *val-MulPower2* **sorry**
then have *val*[*xv* ∗ *yv*] = *val*[(*xv* << (*IntVal 64 i*)) + *xv*]
using *val-MulPower2Add1* **sorry**
then show *?thesis*
by (*metis 1 evalDet lhs p*(*2*) *rhs*)
qed
done

optimization *MulPower2Sub1*: *x* ∗ *y* ⟼ (*x* << *const* (*IntVal 64 i*)) − *x*
when (*i* > *0* ∧ *stamp-expr x* = *IntegerStamp 64 xl xh* ∧ *wf-stamp x* ∧
64 > *i* ∧
y = *ConstantExpr* (*IntVal 64* ((*2* ^ *unat*(*i*)) − *1*)))
apply *simp* **apply** (*rule impI*; (*rule allI*)+; *rule impI*)

```
  subgoal premises p for m p v
  proof -
    obtain xv where xv: [m,p] ⊢ x ↦ xv
      using p by fast
    then obtain xvv where xvv: xv = IntVal 64 xvv
      using p by (metis valid-int wf-stamp-def)
    obtain yv where yv: [m,p] ⊢ y ↦ yv
      using p by blast
    have ygezero: y > ConstantExpr (IntVal 64 0) sorry
    then have 1: 0 < i ∧
              i < 64 ∧
              y = ConstantExpr (IntVal 64 ((2 ^ unat(i)) - 1))
      using p by blast
    then have lhs: [m, p] ⊢ exp[x * y] ↦ val[xv * yv]
      by (metis bin-eval.simps(3) evalDet p(2) xv yv unfold-binary)
    then have [m, p] ⊢ exp[const (IntVal 64 i)] ↦ val[(IntVal 64 i)]
     by (metis wf-value-def verit-comp-simplify1(2) zero-less-numeral ConstantExpr
take-bit64
        constantAsStamp.simps(1) validStampIntConst valid-value.simps(1))
    then have rhs2: [m, p] ⊢ exp[x << const (IntVal 64 i)] ↦ val[xv << (IntVal
64 i)]
     by (metis Value.simps(5) bin-eval.simps(10) intval-left-shift.simps(1) new-int.simps
xv xvv
         evaltree.BinaryExpr)
     then have rhs: [m, p] ⊢ exp[(x << const (IntVal 64 i)) - x] ↦ val[(xv <<
(IntVal 64 i)) - xv]
      using 1 equiv-exprs-def ygezero yv by fastforce
    then have val[xv * yv] = val[(xv << (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.JavaLong
begin

lemma intval-distribute-and-over-or:
  val[z & (x | y)] = val[(z & x) | (z & y)]
  by (cases x; cases y; cases z; auto simp add: bit.conj-disj-distrib)
```

lemma *exp-distribute-and-over-or*:
  exp[*z* & (*x* | *y*)] ≥ *exp*[(*z* & *x*) | (*z* & *y*)]
  apply *auto*
  by (*metis bin-eval.simps*(*6*,*7*) *intval-or.simps*(*2*,*6*) *intval-distribute-and-over-or*
BinaryExpr)

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* ⊕ *y*] = *val*[*y* ⊕ *x*]
  by (*cases x*; *cases y*; *auto simp*: *xor.commute*)

lemma *exp-and-commute*:
  exp[*x* & *z*] ≥ *exp*[*z* & *x*]
  by (*auto simp*: *intval-and-commute*)

lemma *exp-or-commute*:
  exp[*x* | *y*] ≥ *exp*[*y* | *x*]
  by (*auto simp*: *intval-or-commute*)

lemma *exp-xor-commute*:
  exp[*x* ⊕ *y*] ≥ *exp*[*y* ⊕ *x*]
  by (*auto simp*: *intval-xor-commute*)

lemma *intval-eliminate-y*:
  assumes *val*[*y* & *z*] = *IntVal b 0*
  shows *val*[(*x* | *y*) & *z*] = *val*[*x* & *z*]
  using *assms* **by** (*cases x*; *cases y*; *cases z*; *auto simp add*: *bit.conj-disj-distrib2*)

lemma *intval-and-associative*:
  val[(*x* & *y*) & *z*] = *val*[*x* & (*y* & *z*)]
  by (*cases x*; *cases y*; *cases z*; *auto simp*: *and.assoc*)

lemma *intval-or-associative*:
  val[(*x* | *y*) | *z*] = *val*[*x* | (*y* | *z*)]
  by (*cases x*; *cases y*; *cases z*; *auto simp*: *or.assoc*)

lemma *intval-xor-associative*:
  val[(*x* ⊕ *y*) ⊕ *z*] = *val*[*x* ⊕ (*y* ⊕ *z*)]
  by (*cases x*; *cases y*; *cases z*; *auto simp*: *xor.assoc*)

lemma *exp-and-associative*:

exp[(*x* & *y*) & *z*] ≥ *exp*[*x* & (*y* & *z*)]
using *intval-and-associative* **by** *fastforce*

lemma *exp-or-associative*:
exp[(*x* | *y*) | *z*] ≥ *exp*[*x* | (*y* | *z*)]
using *intval-or-associative* **by** *fastforce*

lemma *exp-xor-associative*:
exp[(*x* ⊕ *y*) ⊕ *z*] ≥ *exp*[*x* ⊕ (*y* ⊕ *z*)]
using *intval-xor-associative* **by** *fastforce*

lemma *intval-and-absorb-or*:
assumes ∃ *b v* . *x* = *new-int b v*
assumes *val*[*x* & (*x* | *y*)] ≠ *UndefVal*
shows *val*[*x* & (*x* | *y*)] = *val*[*x*]
using *assms* **apply** (*cases x*; *cases y*; *auto*)
by (*metis* (*full-types*) *intval-and.simps*(*6*))

lemma *intval-or-absorb-and*:
assumes ∃ *b v* . *x* = *new-int b v*
assumes *val*[*x* | (*x* & *y*)] ≠ *UndefVal*
shows *val*[*x* | (*x* & *y*)] = *val*[*x*]
using *assms* **apply** (*cases x*; *cases y*; *auto*)
by (*metis* (*full-types*) *intval-or.simps*(*6*))

lemma *exp-and-absorb-or*:
exp[*x* & (*x* | *y*)] ≥ *exp*[*x*]
apply *auto*
subgoal premises *p* **for** *m p xa xaa ya*
proof−
obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *x* ↦ *xv*
using *p*(*1*) **by** *auto*
obtain *yv* **where** *yv*: [*m*,*p*] ⊢ *y* ↦ *yv*
using *p*(*4*) **by** *auto*
then have *lhsDefined*: *val*[*xv* & (*xv* | *yv*)] ≠ *UndefVal*
by (*metis evalDet p*(*1*,*2*,*3*,*4*) *xv*)
obtain *xb xvv* **where** *xvv*: *xv* = *IntVal xb xvv*
by (*metis Value.exhaust-sel intval-and.simps*(*2*,*3*,*4*,*5*) *lhsDefined*)
obtain *yb yvv* **where** *yvv*: *yv* = *IntVal yb yvv*
by (*metis Value.exhaust-sel intval-and.simps*(*6*) *intval-or.simps*(*6*,*7*,*8*,*9*) *lhsDefined*)
then have *valEval*: *val*[*xv* & (*xv* | *yv*)] = *val*[*xv*]
by (*metis eval-unused-bits-zero intval-and-absorb-or lhsDefined new-int.elims xv xvv*)
then show *?thesis*
by (*metis evalDet p*(*1*,*3*,*4*) *xv yv*)
qed
done

lemma *exp-or-absorb-and*:
  exp[*x* | (*x* & *y*)] ≥ *exp*[*x*]
  apply *auto*
  subgoal premises *p* **for** *m p xa xaa ya*
  proof−
    obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *x* ↦ *xv*
      using *p*(*1*) **by** *auto*
    obtain *yv* **where** *yv*: [*m*,*p*] ⊢ *y* ↦ *yv*
      using *p*(*4*) **by** *auto*
    then have *lhsDefined*: *val*[*xv* | (*xv* & *yv*)] ≠ *UndefVal*
      by (*metis evalDet p*(*1*,*2*,*3*,*4*) *xv*)
    obtain *xb xvv* **where** *xvv*: *xv* = *IntVal xb xvv*
      by (*metis Value.exhaust-sel intval-and.simps*(*3*,*4*,*5*) *intval-or.simps*(*2*,*6*) *lhs-Defined*)
    obtain *yb yvv* **where** *yvv*: *yv* = *IntVal yb yvv*
      by (*metis Value.exhaust-sel intval-and.simps*(*6*,*7*,*8*,*9*) *intval-or.simps*(*6*) *lhs-Defined*)
    then have *valEval*: *val*[*xv* | (*xv* & *yv*)] = *val*[*xv*]
      by (*metis eval-unused-bits-zero intval-or-absorb-and lhsDefined new-int.elims xv xvv*)
    then show *?thesis*
      by (*metis evalDet p*(*1*,*3*,*4*) *xv yv*)
  qed
  done

lemma
  assumes *y* = *0*
  shows *x* + *y* = *or x y*
  by (*simp add*: *assms*)

lemma *no-overlap-or*:
  assumes *and x y* = *0*
  shows *x* + *y* = *or x y*
  by (*metis bit-and-iff bit-xor-iff disjunctive-add xor-self-eq assms*)

context *stamp-mask*
begin

lemma *intval-up-and-zero-implies-zero*:
  assumes *and* (↑*x*) (↑*y*) = *0*

assumes $[m, p] \vdash x \mapsto xv$
assumes $[m, p] \vdash y \mapsto yv$
assumes $val[xv \,\&\, yv] \neq UndefVal$
shows $\exists\ b\ .\ val[xv \,\&\, yv] = new\text{-}int\ b\ 0$
using *assms* **apply** (*cases xv*; *cases yv*; *auto*)
apply (*metis eval-unused-bits-zero stamp-mask.up-mask-and-zero-implies-zero stamp-mask-axioms*)
by *presburger*

lemma *exp-eliminate-y*:
and (↑*y*) (↑*z*) = *0* ⟶ *exp*[(*x* | *y*) & *z*] ≥ *exp*[*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*] ⊢ *x* ↦ *xv*
using *e* **by** *auto*
obtain *yv* **where** *yv*: [*m*,*p*] ⊢ *y* ↦ *yv*
using *e* **by** *auto*
obtain *zv* **where** *zv*: [*m*,*p*] ⊢ *z* ↦ *zv*
using *e* **by** *auto*
have *lhs*: *v* = *val*[(*xv* | *yv*) & *zv*]
by (*smt* (*verit*, *best*) *BinaryExprE bin-eval.simps*(*6*,*7*) *e evalDet xv yv zv*)
then have *v* = *val*[(*xv* & *zv*) | (*yv* & *zv*)]
by (*simp add*: *intval-and-commute intval-distribute-and-over-or*)
also have ∃ *b*. *val*[*yv* & *zv*] = *new-int b 0*
by (*metis calculation e intval-or.simps*(*6*) *p unfold-binary intval-up-and-zero-implies-zero yv*
zv)
ultimately have *rhs*: *v* = *val*[*xv* & *zv*]
by (*auto simp*: *intval-eliminate-y lhs*)
from *lhs rhs* **show** *?thesis*
by (*metis BinaryExpr BinaryExprE bin-eval.simps*(*6*) *e xv zv*)
qed
done
done

lemma *leadingZeroBounds*:
fixes *x* :: *'a*::*len word*
assumes *n* = *numberOfLeadingZeros x*
shows *0* ≤ *n* ∧ *n* ≤ *Nat.size x*
by (*simp add*: *MaxOrNeg-def highestOneBit-def nat-le-iff numberOfLeadingZeros-def assms*)

lemma *above-nth-not-set*:
fixes *x* :: *int64*
assumes *n* = *64* − *numberOfLeadingZeros x*
shows *j* > *n* ⟶ ¬(*bit x j*)
by (*smt* (*verit*, *ccfv-SIG*) *highestOneBit-def int-nat-eq int-ops*(*6*) *less-imp-of-nat-less size64*
max-set-bit zerosAboveHighestOne assms numberOfLeadingZeros-def)

```
no-notation LogicNegationNotation (!-)

lemma zero-horner:
  horner-sum of-bool 2 (map (λx. False) xs) = 0
  by (induction xs; auto)

lemma zero-map:
  assumes j ≤ n
  assumes ∀ i. j ≤ i ⟶ ¬(f i)
  shows map f [0..<n] = map f [0..<j] @ map (λx. False) [j..<n]
 by (smt (verit, del-insts) add-diff-inverse-nat atLeastLessThan-iff bot-nat-0.extremum
leD assms
      map-append map-eq-conv set-upt upt-add-eq-append)

lemma map-join-horner:
  assumes map f [0..<n] = map f [0..<j] @ map (λx. False) [j..<n]
 shows horner-sum of-bool (2::'a::len word) (map f [0..<n]) = horner-sum of-bool
2 (map f [0..<j])
proof –
  have horner-sum of-bool (2::'a::len word) (map f [0..<n]) = horner-sum of-bool
2 (map f [0..<j]) + 2 ^ length [0..<j] ∗ horner-sum of-bool 2 (map f [j..<n])
    using assms apply auto
     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 horner-sum-append)
   also have ... = horner-sum of-bool 2 (map f [0..<j]) + 2 ^ length [0..<j] ∗
horner-sum of-bool 2 (map (λx. False) [j..<n])
    by (metis calculation horner-sum-append length-map assms)
  also have ... = horner-sum of-bool 2 (map f [0..<j])
    using zero-horner mult-not-zero by auto
  finally show ?thesis
    by simp
qed

lemma split-horner:
  assumes j ≤ n
  assumes ∀ i. j ≤ i ⟶ ¬(f i)
 shows horner-sum of-bool (2::'a::len word) (map f [0..<n]) = horner-sum of-bool
2 (map f [0..<j])
  by (auto simp: assms zero-map map-join-horner)

lemma transfer-map:
  assumes ∀ i. i < n ⟶ f i = f' i
  shows (map f [0..<n]) = (map f' [0..<n])
  by (simp add: assms)

lemma transfer-horner:
  assumes ∀ i. i < n ⟶ 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*])
by (*smt* (*verit*, *best*) *assms transfer-map*)

lemma *L1*:
assumes *n* = *64* − *numberOfLeadingZeros* (↑*z*)
assumes [*m*, *p*] ⊢ *z* ↦ *IntVal b zv*
shows *and v zv* = *and* (*v mod 2^n*) *zv*
proof −
have *nle*: *n* ≤ *64*
using *assms diff-le-self* **by** *blast*
also have *and v zv* = *horner-sum of-bool 2* (*map* (*bit* (*and v zv*)) [*0*..<*64*])
by (*metis size-word.rep-eq take-bit-length-eq horner-sum-bit-eq-take-bit size64*)
also have ... = *horner-sum of-bool 2* (*map* (λ*i*. *bit* (*and v zv*) *i*) [*0*..<*64*])
by *blast*
also have ... = *horner-sum of-bool 2* (*map* (λ*i*. ((*bit v i*) ∧ (*bit zv i*))) [*0*..<*64*])
by (*metis bit-and-iff*)
also have ... = *horner-sum of-bool 2* (*map* (λ*i*. ((*bit v i*) ∧ (*bit zv i*))) [*0*..<*n*])
proof −
have ∀ *i*. *i* ≥ *n* ⟶ ¬(*bit zv i*)
by (*smt* (*verit*, *ccfv-SIG*) *One-nat-def diff-less int-ops*(*6*) *leadingZerosAddHighestOne assms*
linorder-not-le nat-int-comparison(*2*) *not-numeral-le-zero size64 zero-less-Suc*

zerosAboveHighestOne not-may-implies-false)
then have ∀ *i*. *i* ≥ *n* ⟶ ¬((*bit v i*) ∧ (*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* (λ*i*. ((*bit* (*v mod 2^n*) *i*) ∧ (*bit zv i*))) [*0*..<*n*])
proof −
have ∀ *i*. *i* < *n* ⟶ *bit* (*v mod 2^n*) *i* = *bit v i*
by (*metis bit-take-bit-iff take-bit-eq-mod*)
then have ∀ *i*. *i* < *n* ⟶ ((*bit v i*) ∧ (*bit zv i*)) = ((*bit* (*v mod 2^n*) *i*) ∧ (*bit zv i*))
by *force*
then show *?thesis*
by (*rule transfer-horner*)
qed
also have ... = *horner-sum of-bool 2* (*map* (λ*i*. ((*bit* (*v mod 2^n*) *i*) ∧ (*bit zv i*))) [*0*..<*64*])
proof −
have ∀ *i*. *i* ≥ *n* ⟶ ¬(*bit zv i*)
by (*smt* (*verit*, *ccfv-SIG*) *One-nat-def diff-less int-ops*(*6*) *leadingZerosAddHighestOne assms*
linorder-not-le nat-int-comparison(*2*) *not-numeral-le-zero size64 zero-less-Suc*

zerosAboveHighestOne not-may-implies-false)
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *assms*(*1*) *diff-le-self split-horner*)
qed
also have ... = *horner-sum of-bool 2* (*map* (*bit* (*and* (*v mod 2^n*) *zv*)) [*0*..<*64*])
by (*meson bit-and-iff*)
also have ... = *and* (*v mod 2^n*) *zv*
by (*metis size-word.rep-eq take-bit-length-eq horner-sum-bit-eq-take-bit size64*)
finally show *?thesis*
using ‹*and* (*v*::*64 word*) (*zv*::*64 word*) = *horner-sum of-bool* (*2*::*64 word*) (*map* (*bit* (*and v zv*)) [*0*::*nat*..<*64*::*nat*])› ‹*horner-sum of-bool* (*2*::*64 word*) (*map* (λ*i*::*nat*. *bit* ((*v*::*64 word*) *mod* (*2*::*64 word*) ^ (*n*::*nat*)) *i* ∧ *bit* (*zv*::*64 word*) *i*) [*0*::*nat*..<*64*::*nat*]) = *horner-sum of-bool* (*2*::*64 word*) (*map* (*bit* (*and* (*v mod* (*2*::*64 word*) ^ *n*) *zv*)) [*0*::*nat*..<*64*::*nat*])› ‹*horner-sum of-bool* (*2*::*64 word*) (*map* (λ*i*::*nat*. *bit* ((*v*::*64 word*) *mod* (*2*::*64 word*) ^ (*n*::*nat*)) *i* ∧ *bit* (*zv*::*64 word*) *i*) [*0*::*nat*..<*n*]) = *horner-sum of-bool* (*2*::*64 word*) (*map* (λ*i*::*nat*. *bit* (*v mod* (*2*::*64 word*) ^ *n*) *i* ∧ *bit zv i*) [*0*::*nat*..<*64*::*nat*])› ‹*horner-sum of-bool* (*2*::*64 word*) (*map* (λ*i*::*nat*. *bit* (*v*::*64 word*) *i* ∧ *bit* (*zv*::*64 word*) *i*) [*0*::*nat*..<*64*::*nat*]) = *horner-sum of-bool* (*2*::*64 word*) (*map* (λ*i*::*nat*. *bit v i* ∧ *bit zv i*) [*0*::*nat*..<*n*::*nat*])› ‹*horner-sum of-bool* (*2*::*64 word*) (*map* (λ*i*::*nat*. *bit* (*v*::*64 word*) *i* ∧ *bit* (*zv*::*64 word*) *i*) [*0*::*nat*..<*n*::*nat*]) = *horner-sum of-bool* (*2*::*64 word*) (*map* (λ*i*::*nat*. *bit* (*v mod* (*2*::*64 word*) ^ *n*) *i* ∧ *bit zv i*) [*0*::*nat*..<*n*])› ‹*horner-sum of-bool* (*2*::*64 word*) (*map* (*bit* (*and* ((*v*::*64 word*) *mod* (*2*::*64 word*) ^ (*n*::*nat*)) (*zv*::*64 word*))) [*0*::*nat*..<*64*::*nat*]) = *and* (*v mod* (*2*::*64 word*) ^ *n*) *zv*› ‹*horner-sum of-bool* (*2*::*64 word*) (*map* (*bit* (*and* (*v*::*64 word*) (*zv*::*64 word*))) [*0*::*nat*..<*64*::*nat*]) = *horner-sum of-bool* (*2*::*64 word*) (*map* (λ*i*::*nat*. *bit v i* ∧ *bit zv i*) [*0*::*nat*..<*64*::*nat*])› **by** *presburger*
qed

lemma *up-mask-upper-bound*:
assumes [*m*, *p*] ⊢ *x* ↦ *IntVal b xv*
shows *xv* ≤ (↑*x*)
by (*metis* (*no-types*, *lifting*) *and.right-neutral bit.conj-cancel-left bit.conj-disj-distribs*(*1*) *bit.double-compl ucast-id up-spec word-and-le1 word-not-dist*(*2*) *assms*)

lemma *L2*:
assumes *numberOfLeadingZeros* (↑*z*) + *numberOfTrailingZeros* (↑*y*) ≥ *64*
assumes *n* = *64* − *numberOfLeadingZeros* (↑*z*)
assumes [*m*, *p*] ⊢ *z* ↦ *IntVal b zv*
assumes [*m*, *p*] ⊢ *y* ↦ *IntVal b yv*
shows *yv mod 2^n* = *0*
proof −
have *yv mod 2^n* = *horner-sum of-bool 2* (*map* (*bit yv*) [*0*..<*n*])
by (*simp add*: *horner-sum-bit-eq-take-bit take-bit-eq-mod*)
also have ... ≤ *horner-sum of-bool 2* (*map* (*bit* (↑*y*)) [*0*..<*n*])
by (*metis* (*no-types*, *opaque-lifting*) *and.right-neutral bit.conj-cancel-right word-not-dist*(*2*) *bit.conj-disj-distribs*(*1*) *bit.double-compl horner-sum-bit-eq-take-bit take-bit-and ucast-id*
up-spec word-and-le1 assms(*4*))

also have *horner-sum of-bool 2* (*map* (*bit* (↑*y*)) [*0*..<*n*]) = *horner-sum of-bool 2* (*map* (λ*x*. *False*) [*0*..<*n*])
  proof −
    have ∀ *i* < *n*. ¬(*bit* (↑*y*) *i*)
      by (*metis add.commute add-diff-inverse-nat add-lessD1 leD le-diff-conv zeros-BelowLowestOne*
          numberOfTrailingZeros-def assms(*1*,*2*))
    then show *?thesis*
      by (*metis* (*full-types*) *transfer-map*)
  qed
  also have *horner-sum of-bool 2* (*map* (λ*x*. *False*) [*0*..<*n*]) = *0*
    by (*auto simp*: *zero-horner*)
  finally show *?thesis*
    by *auto*
qed

thm-oracles *L1 L2*

lemma *unfold-binary-width-add*:
  shows ([*m*,*p*] ⊢ *BinaryExpr BinAdd xe ye* ↦ *IntVal b val*) = (∃ *x y*.
          (([*m*,*p*] ⊢ *xe* ↦ *IntVal b x*) ∧
           ([*m*,*p*] ⊢ *ye* ↦ *IntVal b y*) ∧
           (*IntVal b val* = *bin-eval BinAdd* (*IntVal b x*) (*IntVal b y*)) ∧
           (*IntVal b val* ≠ *UndefVal*)
        )) (**is** *?L* = *?R*)
  using *unfold-binary-width* **by** *simp*

lemma *unfold-binary-width-and*:
  shows ([*m*,*p*] ⊢ *BinaryExpr BinAnd xe ye* ↦ *IntVal b val*) = (∃ *x y*.
          (([*m*,*p*] ⊢ *xe* ↦ *IntVal b x*) ∧
           ([*m*,*p*] ⊢ *ye* ↦ *IntVal b y*) ∧
           (*IntVal b val* = *bin-eval BinAnd* (*IntVal b x*) (*IntVal b y*)) ∧
           (*IntVal b val* ≠ *UndefVal*)
        )) (**is** *?L* = *?R*)
  using *unfold-binary-width* **by** *simp*

lemma *mod-dist-over-add-right*:
  fixes *a b c* :: *int64*
  fixes *n* :: *nat*
  assumes *0* < *n*
  assumes *n* < *64*
  shows (*a* + *b mod 2^n*) *mod 2^n* = (*a* + *b*) *mod 2^n*
  using *mod-dist-over-add* **by** (*simp add*: *assms add.commute*)

lemma *numberOfLeadingZeros-range*:
  0 ≤ *numberOfLeadingZeros n* ∧ *numberOfLeadingZeros n* ≤ *Nat.size n*
  by (*simp add*: *leadingZeroBounds*)

lemma *improved-opt*:

assumes *numberOfLeadingZeros* (↑*z*) + *numberOfTrailingZeros* (↑*y*) ≥ *64*
shows *exp*[(*x* + *y*) & *z*] ≥ *exp*[*x* & *z*]
apply *simp* **apply** ((*rule allI*)+; *rule impI*)
subgoal premises *eval* **for** *m p v*
proof −
obtain *n* **where** *n*: *n* = *64* − *numberOfLeadingZeros* (↑*z*)
by *simp*
obtain *b val* **where** *val*: [*m*, *p*] ⊢ *exp*[(*x* + *y*) & *z*] ↦ *IntVal b val*
by (*metis BinaryExprE bin-eval-new-int eval new-int.simps*)
then obtain *xv yv* **where** *addv*: [*m*, *p*] ⊢ *exp*[*x* + *y*] ↦ *IntVal b* (*xv* + *yv*)
apply (*subst* (*asm*) *unfold-binary-width-and*) **by** (*metis add.right-neutral*)
then obtain *yv* **where** *yv*: [*m*, *p*] ⊢ *y* ↦ *IntVal b yv*
apply (*subst* (*asm*) *unfold-binary-width-add*) **by** *blast*
from *addv* **obtain** *xv* **where** *xv*: [*m*, *p*] ⊢ *x* ↦ *IntVal b xv*
apply (*subst* (*asm*) *unfold-binary-width-add*) **by** *blast*
from *val* **obtain** *zv* **where** *zv*: [*m*, *p*] ⊢ *z* ↦ *IntVal b zv*
apply (*subst* (*asm*) *unfold-binary-width-and*) **by** *blast*
have *addv*: [*m*, *p*] ⊢ *exp*[*x* + *y*] ↦ *new-int b* (*xv* + *yv*)
using *xv yv evaltree.BinaryExpr* **by** *auto*
have *lhs*: [*m*, *p*] ⊢ *exp*[(*x* + *y*) & *z*] ↦ *new-int b* (*and* (*xv* + *yv*) *zv*)
using *addv zv* **apply** (*rule evaltree.BinaryExpr*) **by** *simp+*
have *rhs*: [*m*, *p*] ⊢ *exp*[*x* & *z*] ↦ *new-int b* (*and xv zv*)
using *xv zv evaltree.BinaryExpr* **by** *auto*
then show *?thesis*
proof (*cases numberOfLeadingZeros* (↑*z*) > *0*)
case *True*
have *n-bounds*: *0* ≤ *n* ∧ *n* < *64*
by (*simp add*: *True n*)
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*
by (*metis take-bit-0 take-bit-eq-mod zero-less-iff-neq-zero mod-dist-over-add-right n-bounds*)
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 assms* **by** *auto*
also have ... = *and* (*xv mod 2^n*) *zv*
by (*smt* (*verit*, *best*) *and.idem take-bit-eq-mask take-bit-eq-mod word-bw-assocs*(*1*)
mod-mod-trivial)
also have ... = *and xv zv*
by (*metis L1 n zv*)
finally show *?thesis*
by (*metis evalDet eval lhs rhs*)
next
case *False*
then have *numberOfLeadingZeros* (↑*z*) = *0*

```
      by simp
    then have numberOfTrailingZeros (↑y) ≥ 64
      using assms by fastforce
    then have yv = 0
        by (metis (no-types, lifting) L1 L2 add-diff-cancel-left' and.comm-neutral
linorder-not-le
        bit.conj-cancel-right bit.conj-disj-distribs(1) bit.double-compl less-imp-diff-less
yv
          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 | y) & z) ⟼ x & z
                                when (((and (IRExpr-up y) (IRExpr-up z)) = 0))
  by (simp add: IRExpr-up-def)+

optimization redundant-lhs-x-or: ((x | y) & z) ⟼ y & z
                                when (((and (IRExpr-up x) (IRExpr-up z)) = 0))
  by (simp add: IRExpr-up-def)+

optimization redundant-rhs-y-or: (z & (x | y)) ⟼ z & x
                                when (((and (IRExpr-up y) (IRExpr-up z)) = 0))
  by (simp add: IRExpr-up-def)+

optimization redundant-rhs-x-or: (z & (x | y)) ⟼ z & y
                                when (((and (IRExpr-up x) (IRExpr-up z)) = 0))
  by (simp add: IRExpr-up-def)+

end

end
```

1.8 NotNode Phase

theory *NotPhase*
  imports
    Common
begin

phase *NotNode*
  terminating *size*
begin

lemma *bin-not-cancel*:
 bin[¬(¬(*e*))] = *bin*[*e*]
  by *auto*

lemma *val-not-cancel*:
  assumes *val*[~(*new-int b v*)] ≠ *UndefVal*
  shows *val*[~(~(*new-int b v*))] = (*new-int b v*)
  by (*simp add*: *take-bit-not-take-bit*)

lemma *exp-not-cancel*:
   exp[~(~*a*)] ≥ *exp*[*a*]
  apply *auto*
  subgoal premises *p* **for** *m p x*
  proof −
    obtain *av* **where** *av*: [*m*,*p*] ⊢ *a* ↦ *av*
      using *p*(*2*) **by** *auto*
    obtain *bv avv* **where** *avv*: *av* = *IntVal bv avv*
     by (*metis Value.exhaust av evalDet evaltree-not-undef intval-not.simps*(*3*,*4*,*5*)
p(*2*,*3*))
    then have *valEval*: *val*[~(~*av*)] = *val*[*av*]
    by (*metis av avv evalDet eval-unused-bits-zero new-int.elims p*(*2*,*3*) *val-not-cancel*)
    then show *?thesis*
      by (*metis av evalDet p*(*2*))
  qed
  done

Optimisations

optimization *NotCancel*: *exp*[~(~*a*)] ⟼ *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	$x \vert y$
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	1

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

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

```
lemma OrLeftFallthrough:
  assumes (and (not (↓x)) (↑y)) = 0
  shows exp[x | y] ≥ 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] ⊢ exp[x | y] ↦ IntVal b vv
      by (metis BinaryExprE bin-eval-new-int new-int.simps eval(2))
    from e obtain xv where xv: [m, p] ⊢ x ↦ IntVal b xv
      apply (subst (asm) unfold-binary-width) by force+
    from e obtain yv where yv: [m, p] ⊢ y ↦ IntVal b yv
      apply (subst (asm) unfold-binary-width) by force+
    have vdef: v = val[(IntVal b xv) | (IntVal b yv)]
      by (metis bin-eval.simps(7) eval(2) evalDet unfold-binary xv yv)
    have ∀ i. (bit xv i) | (bit yv i) = (bit xv i)
      by (metis assms bit-and-iff not-down-up-mask-and-zero-implies-zero xv yv)
    then have IntVal b xv = val[(IntVal b xv) | (IntVal b yv)]
     by (metis (no-types, lifting) and.idem assms bit.conj-disj-distrib eval-unused-bits-zero
yv xv
        intval-or.simps(1) new-int.simps new-int-bin.simps not-down-up-mask-and-zero-implies-zero

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

```
  done

lemma OrRightFallthrough:
  assumes (and (not (↓y)) (↑x)) = 0
  shows exp[x | y] ≥ 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] ⊢ exp[x | y] ↦ IntVal b vv
      by (metis BinaryExprE bin-eval-new-int new-int.simps eval(2))
    from e obtain xv where xv: [m, p] ⊢ x ↦ IntVal b xv
      apply (subst (asm) unfold-binary-width) by force+
    from e obtain yv where yv: [m, p] ⊢ y ↦ IntVal b yv
      apply (subst (asm) unfold-binary-width) by force+
    have vdef: v = val[(IntVal b xv) | (IntVal b yv)]
      by (metis bin-eval.simps(7) eval(2) evalDet unfold-binary xv yv)
    have ∀ i. (bit xv i) | (bit yv i) = (bit yv i)
      by (metis assms bit-and-iff not-down-up-mask-and-zero-implies-zero xv yv)
    then have IntVal b yv = val[(IntVal b xv) | (IntVal b yv)]
       by (metis (no-types, lifting) assms eval-unused-bits-zero intval-or.simps(1)
new-int.elims yv
              new-int-bin.elims stamp-mask.not-down-up-mask-and-zero-implies-zero
stamp-mask-axioms xv
          word-ao-absorbs(8))
    then show ?thesis
      using vdef yv by presburger
  qed
  done

end

phase OrNode
  terminating size
begin


lemma bin-or-equal:
  bin[x | x] = bin[x]
  by simp

lemma bin-shift-const-right-helper:
 x | y = y | x
  by simp

lemma bin-or-not-operands:
 (~x | ~y) = (~(x & y))
  by simp
```

```
lemma val-or-equal:
  assumes x = new-int b v
  and     val[x | x] ≠ UndefVal
  shows   val[x | x] = val[x]
  by (auto simp: assms)

lemma val-elim-redundant-false:
  assumes x = new-int b v
  and     val[x | false] ≠ UndefVal
  shows   val[x | false] = val[x]
  using assms by (cases x; auto; presburger)

lemma val-shift-const-right-helper:
   val[x | y] = val[y | x]
  by (cases x; cases y; auto simp: or.commute)

lemma val-or-not-operands:
 val[~x | ~y] = val[~(x & y)]
  by (cases x; cases y; auto simp: take-bit-not-take-bit)


lemma exp-or-equal:
  exp[x | x] ≥ exp[x]
  apply auto[1]
  subgoal premises p for m p xa ya
  proof−
    obtain xv where xv: [m,p] ⊢ x ↦ xv
      using p(1) by auto
    obtain xb xvv where xvv: xv = IntVal xb xvv
     by (metis evalDet evaltree-not-undef intval-is-null.cases intval-or.simps(3,4,5)
p(1,3) xv)
    then have evalNotUndef: val[xv | xv] ≠ UndefVal
      using p evalDet xv by blast
    then have orUnfold: val[xv | xv] = (new-int xb (or xvv xvv))
      by (simp add: xvv)
    then have simplify: val[xv | xv] = (new-int xb (xvv))
      by (simp add: orUnfold)
    then have eq: (xv) = (new-int xb (xvv))
      using eval-unused-bits-zero xv xvv by auto
    then show ?thesis
      by (metis evalDet p(1,2) simplify xv)
  qed
  done

lemma exp-elim-redundant-false:
 exp[x | false] ≥ exp[x]
  apply auto[1]
  subgoal premises p for m p xa
```

proof−
obtain *xv* **where** *xv*: $[m,p] \vdash x \mapsto xv$
using *p*(*1*) **by** *auto*
obtain *xb xvv* **where** *xvv*: *xv* = *IntVal xb xvv*
by (*metis evalDet evaltree-not-undef intval-is-null.cases intval-or.simps*(*3*,*4*,*5*) *p*(*1*,*2*) *xv*)
then have *evalNotUndef*: *val*[*xv* | (*IntVal 32 0*)] ≠ *UndefVal*
using *p evalDet xv* **by** *blast*
then have *widthSame*: *xb*=*32*
by (*metis intval-or.simps*(*1*) *new-int-bin.simps xvv*)
then have *orUnfold*: *val*[*xv* | (*IntVal 32 0*)] = (*new-int xb* (*or xvv 0*))
by (*simp add*: *xvv*)
then have *simplify*: *val*[*xv* | (*IntVal 32 0*)] = (*new-int xb* (*xvv*))
by (*simp add*: *orUnfold*)
then have *eq*: (*xv*) = (*new-int xb* (*xvv*))
using *eval-unused-bits-zero xv xvv* **by** *auto*
then show *?thesis*
by (*metis evalDet p*(*1*) *simplify xv*)
qed
done

Optimisations

optimization *OrEqual*: $x \mid x \longmapsto x$
by (*meson exp-or-equal*)

optimization *OrShiftConstantRight*: ((*const x*) | *y*) ⟼ *y* | (*const x*) *when* ¬(*is-ConstantExpr y*)
using *size-flip-binary* **by** (*auto simp*: *BinaryExpr unfold-const val-shift-const-right-helper*)

optimization *EliminateRedundantFalse*: *x* | *false* ⟼ *x*
by (*meson exp-elim-redundant-false*)

optimization *OrNotOperands*: (~*x* | ~*y*) ⟼ ~(*x* & *y*)
apply (*metis add-2-eq-Suc′ less-SucI not-add-less1 not-less-eq size-binary-const size-non-add*)
using *BinaryExpr UnaryExpr bin-eval.simps*(*4*) *intval-not.simps*(*2*) *unary-eval.simps*(*3*)

val-or-not-operands **by** *fastforce*

optimization *OrLeftFallthrough*:
x | *y* ⟼ *x when* ((*and* (*not* (*IRExpr-down x*)) (*IRExpr-up y*)) = *0*)
using *simple-mask.OrLeftFallthrough* **by** *blast*

optimization *OrRightFallthrough*:
x | *y* ⟼ *y when* ((*and* (*not* (*IRExpr-down y*)) (*IRExpr-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* ⇒ *Value* **where**
  intval-log2 (*IntVal b v*) = *IntVal b* (*word-of-int* (*SOME e. v=2^e*)) |
  intval-log2 - = *UndefVal*

fun *in-bounds* :: *Value* ⇒ *int* ⇒ *int* ⇒ *bool* **where**
  in-bounds (*IntVal b v*) *l h* = (*l* < *sint v* ∧ *sint v* < *h*) |
  in-bounds - *l h* = *False*

lemma
  assumes *in-bounds* (*intval-log2 val-c*) *0 32*
  shows *val*[*x* << (*intval-log2 val-c*)] = *val*[*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* ∧ *in-bounds n 0 32* ⟶
    val[*x* << (*intval-log2 val-c*)] = *val*[*x* ∗ *val-c*]
proof (*rule impI*)
  assume *n* = *intval-log2 val-c* ∧ *in-bounds n 0 32*
  show *val*[*x* << (*intval-log2 val-c*)] = *val*[*x* ∗ *val-c*]
    proof (*cases* ∃ *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^e*))
        using ‹*n* = *intval-log2 val-c* ∧ *in-bounds n 0 32*› *intval-log2.simps*(*1*) **by** *presburger*
      then show *?thesis* **sorry**
    next
      case *False*
      then have ∃ *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^e*))
using ‹*n* = *intval-log2 val-c* ∧ *in-bounds n 0 32*› *intval-log2.simps*(*1*) **by** *presburger*
then show *?thesis* **sorry**
qed
qed

optimization *e*:
x ∗ (*const c*) ⟼ *x* << (*const n*) *when* (*n* = *intval-log2 c* ∧ *in-bounds n 0 32*)
using *e-intval 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* = *new-int 32 v*
shows *intval-div x* (*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 *intval-mod x* (*IntVal 32 1*) ≠ *UndefVal*
  shows *intval-mod x* (*IntVal 32 1*) = *IntVal 32 0*
  using *assms* **apply** (*cases x*; *auto*) **sorry**

value *word-of-int* (*sint* (*x2*::*32 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*) *word*) + (*y*::(′*a*::*len*) *word*)) − *y* = *x*
  by *simp*

lemma *sub-self-is-zero*:
  shows (*x*::(′*a*::*len*) *word*) − *x* = *0*
  by *simp*

lemma *bin-sub-then-left-add*:
  shows (*x*::(′*a*::*len*) *word*) − (*x* + (*y*::(′*a*::*len*) *word*)) = −*y*
  by *simp*

lemma *bin-sub-then-left-sub*:
  shows (*x*::(′*a*::*len*) *word*) − (*x* − (*y*::(′*a*::*len*) *word*)) = *y*
  by *simp*

lemma *bin-subtract-zero*:
  shows (*x* :: ′*a*::*len word*) − (*0* :: ′*a*::*len word*) = *x*
  by *simp*

lemma *bin-sub-negative-value*:
 (*x* :: (′*a*::*len*) *word*) − (−(*y* :: (′*a*::*len*) *word*)) = *x* + *y*
  by *simp*

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

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

lemma *val-sub-after-right-add-2*:
assumes $x = new\text{-}int\ b\ v$
assumes $val[(x + y) - y] \neq UndefVal$
shows $val[(x + y) - y] = x$
using *assms* **apply** (*cases x*; *cases y*; *auto*)
by (*metis* (*full-types*) *intval-sub.simps*(*2*))

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

lemma *val-sub-then-left-sub*:
assumes $y = new\text{-}int\ b\ v$
assumes $val[x - (x - y)] \neq UndefVal$
shows $val[x - (x - y)] = y$
using *assms* **apply** (*cases x*; *auto*)
by (*metis* (*mono-tags*) *intval-sub.simps*(*6*))

lemma *val-subtract-zero*:
assumes $x = new\text{-}int\ b\ v$
assumes $val[x - (IntVal\ b\ 0)] \neq UndefVal$
shows $val[x - (IntVal\ b\ 0)] = x$
by (*cases x*; *simp add*: *assms*)

lemma *val-zero-subtract-value*:
assumes $x = new\text{-}int\ b\ v$
assumes $val[(IntVal\ b\ 0) - x] \neq UndefVal$
shows $val[(IntVal\ b\ 0) - x] = val[-x]$
by (*cases x*; *simp add*: *assms*)

lemma *val-sub-then-left-add*:
assumes $val[x - (x + y)] \neq UndefVal$
shows $val[x - (x + y)] = val[-y]$
using *assms* **apply** (*cases x*; *cases y*; *auto*)
by (*metis* (*mono-tags*, *lifting*) *intval-sub.simps*(*6*))

lemma *val-sub-negative-value*:

```
  assumes val[x − (−y)] ≠ UndefVal
  shows   val[x − (−y)] = val[x + y]
  by (cases x; cases y; simp add: assms)

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

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


lemma exp-sub-after-right-add:
  shows exp[(x + y) − y] ≥ x
  apply auto
  subgoal premises p for m p ya xa yaa
  proof−
    obtain xv where xv: [m,p] ⊢ x ↦ xv
      using p(3) by auto
    obtain yv where yv: [m,p] ⊢ y ↦ yv
      using p(1) by auto
    obtain xb xvv where xvv: xv = IntVal xb xvv
       by (metis Value.exhaust evalDet evaltree-not-undef intval-add.simps(3,4,5)
intval-sub.simps(2)
         p(2,3) xv)
    obtain yb yvv where yvv: yv = IntVal yb yvv
     by (metis evalDet evaltree-not-undef intval-add.simps(7,8,9) intval-logic-negation.cases
yv
         intval-sub.simps(2) p(2,4))
    then have lhsDefined: val[(xv + yv) − yv] ≠ UndefVal
      using xvv yvv apply (cases xv; cases yv; auto)
      by (metis evalDet intval-add.simps(1) p(3,4,5) xv yv)
     then show ?thesis
      by (metis ‹⋀thesis. (⋀(xb) xvv. (xv) = IntVal xb xvv ⟹ thesis) ⟹ thesis›
evalDet xv yv
        eval-unused-bits-zero lhsDefined new-int.simps p(1,3,4) val-sub-after-right-add-2)
  qed
  done

lemma exp-sub-after-right-add2:
  shows exp[(x + y) − x] ≥ y
  using exp-sub-after-right-add apply auto
  by (metis bin-eval.simps(1,2) intval-add-sym unfold-binary)

lemma exp-sub-negative-value:
 exp[x − (−y)] ≥ exp[x + y]
```

apply *auto*
subgoal premises *p* **for** *m p xa ya*
proof −
obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *x* ↦ *xv*
using *p*(*1*) **by** *auto*
obtain *yv* **where** *yv*: [*m*,*p*] ⊢ *y* ↦ *yv*
using *p*(*3*) **by** *auto*
then have *rhsEval*: [*m*,*p*] ⊢ *exp*[*x* + *y*] ↦ *val*[*xv* + *yv*]
by (*metis bin-eval.simps*(*1*) *evalDet p*(*1*,*2*,*3*) *unfold-binary val-sub-negative-value xv*)
then show *?thesis*
by (*metis evalDet p*(*1*,*2*,*3*) *val-sub-negative-value xv yv*)
qed
done

lemma *exp-sub-then-left-sub*:
exp[*x* − (*x* − *y*)] ≥ *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*] ⊢ *x* ↦ *xa*
using *p*(*2*) **by** *blast*
obtain *ya* **where** *ya*: [*m*, *p*] ⊢ *y* ↦ *ya*
using *p*(*5*) **by** *auto*
obtain *xaa* **where** *xaa*: [*m*, *p*] ⊢ *x* ↦ *xaa*
using *p*(*2*) **by** *blast*
have *1*: *val*[*xa* − (*xaa* − *ya*)] ≠ *UndefVal*
by (*metis evalDet p*(*2*,*3*,*4*,*5*) *xa xaa ya*)
then have *val*[*xaa* − *ya*] ≠ *UndefVal*
by *auto*
then have [*m*, *p*] ⊢ *y* ↦ *val*[*xa* − (*xaa* − *ya*)]
by (*metis 1 Value.exhaust eval-unused-bits-zero evaltree-not-undef xa xaa ya new-int.simps*
intval-sub.simps(*6*,*7*,*8*,*9*) *evalDet val-sub-then-left-sub*)
then show *?thesis*
by (*metis evalDet p*(*2*,*4*,*5*) *xa xaa ya*)
qed
done

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

lemma *SubtractZero-Exp*:
exp[(*x* − (*const IntVal b 0*))] ≥ *x*
apply *auto*
subgoal premises *p* **for** *m p xa*
proof−
obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *x* ↦ *xv*
using *p*(*1*) **by** *auto*
obtain *xb xvv* **where** *xvv*: *xv* = *IntVal xb xvv*

by (*metis array-length.cases evalDet evaltree-not-undef intval-sub.simps*(*3*,*4*,*5*) *p*(*1*,*2*) *xv*)
then have *widthSame*: *xb*=*b*
by (*metis evalDet intval-sub.simps*(*1*) *new-int-bin.simps p*(*1*) *p*(*2*) *xv*)
then have *unfoldSub*: *val*[*xv* − (*IntVal b 0*)] = (*new-int xb* (*xvv*−*0*))
by (*simp add*: *xvv*)
then have *rhsSame*: *val*[*xv*] = (*new-int xb* (*xvv*))
using *eval-unused-bits-zero xv xvv* **by** *auto*
then show *?thesis*
by (*metis diff-zero evalDet p*(*1*) *unfoldSub xv*)
qed
done

lemma *ZeroSubtractValue-Exp*:
assumes *wf-stamp x*
assumes *stamp-expr x* = *IntegerStamp b lo hi*
assumes ¬(*is-ConstantExpr x*)
shows *exp*[(*const IntVal b 0*) − *x*] ≥ *exp*[−*x*]
using *assms* **apply** *auto*
subgoal premises *p* **for** *m p xa*
proof−
obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *x* ↦ *xv*
using *p*(*4*) **by** *auto*
obtain *xb xvv* **where** *xvv*: *xv* = *IntVal xb xvv*
by (*metis constantAsStamp.cases evalDet evaltree-not-undef intval-sub.simps*(*7*,*8*,*9*) *p*(*4*,*5*) *xv*)
then have *unfoldSub*: *val*[(*IntVal b 0*) − *xv*] = (*new-int xb* (*0*−*xvv*))
by (*metis intval-sub.simps*(*1*) *new-int-bin.simps p*(*1*,*2*) *valid-int-same-bits wf-stamp-def xv*)
then show *?thesis*
by (*metis UnaryExpr intval-negate.simps*(*1*) *p*(*4*,*5*) *unary-eval.simps*(*2*) *verit-minus-simplify*(*3*)
evalDet xv xvv)
qed
done

Optimisations

optimization *SubAfterAddRight*: ((*x* + *y*) − *y*) ⟼ *x*
using *exp-sub-after-right-add* **by** *blast*

optimization *SubAfterAddLeft*: ((*x* + *y*) − *x*) ⟼ *y*
using *exp-sub-after-right-add2* **by** *blast*

optimization *SubAfterSubLeft*: ((*x* − *y*) − *x*) ⟼ −*y*
by (*smt* (*verit*) *Suc-lessI add-2-eq-Suc′ add-less-cancel-right less-trans-Suc not-add-less1 evalDet*
size-binary-const size-binary-lhs size-binary-rhs size-non-add BinaryExprE bin-eval.simps(*2*)
le-expr-def unary-eval.simps(*2*) *unfold-unary val-sub-after-left-sub*)+

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

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

optimization *SubThenSubLeft*: $(x - (x - y)) \longmapsto y$
using *size-simps exp-sub-then-left-sub* **by** *auto*

optimization *SubtractZero*: $(x - (const\ IntVal\ b\ 0)) \longmapsto x$
using *SubtractZero-Exp* **by** *fast*

thm-oracles *SubtractZero*

optimization *SubNegativeValue*: $(x - (-y)) \longmapsto x + y$
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** *blast*

thm-oracles *SubNegativeValue*

lemma *negate-idempotent*:
assumes $x = IntVal\ b\ v \land take\text{-}bit\ b\ v = v$
shows $x = val[-(-x)]$
by (*auto simp*: *assms is-IntVal-def*)

optimization *ZeroSubtractValue*: $((const\ IntVal\ b\ 0) - x) \longmapsto (-x)$
when ($wf\text{-}stamp\ x \land stamp\text{-}expr\ x = IntegerStamp\ b\ lo\ hi \land \neg(is\text{-}ConstantExpr\ x)$)
using *size-flip-binary ZeroSubtractValue-Exp* **by** *simp+*

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

end

end

1.14 XorNode Phase

theory *XorPhase*
  imports
    Common
    Proofs.StampEvalThms
begin

phase *XorNode*
  terminating *size*
begin

lemma *bin-xor-self-is-false*:
 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*
  by (*cases x*; *auto simp*: *assms*)

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*
  by (*auto simp*: *assms*)

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*
  by (*auto simp*: *assms*)

lemma *val-xor-commute*:

```
  val[x ⊕ y] = val[y ⊕ x]
  by (cases x; cases y; auto simp: xor.commute)

lemma val-eliminate-redundant-false:
  assumes x = new-int b v
  assumes val[x ⊕ (bool-to-val False)] ≠ UndefVal
  shows   val[x ⊕ (bool-to-val False)] = x
  using assms by (auto; meson)


lemma exp-xor-self-is-false:
 assumes wf-stamp x ∧ stamp-expr x = default-stamp
 shows   exp[x ⊕ x] ≥ exp[false]
  using assms apply auto
  subgoal premises p for m p xa ya
  proof−
    obtain xv where xv: [m,p] ⊢ x ↦ xv
      using p(3) by auto
    obtain xb xvv where xvv: xv = IntVal xb xvv
     by (metis Value.exhaust-sel assms evalDet evaltree-not-undef intval-xor.simps(5,7)
p(3,4,5) xv
          valid-value.simps(11) wf-stamp-def)
    then have unfoldXor: val[xv ⊕ xv] = (new-int xb (xor xvv xvv))
      by simp
    then have isZero: xor xvv xvv = 0
      by simp
    then have width: xb = 32
      by (metis valid-int-same-bits xv xvv p(1,2) wf-stamp-def)
    then have isFalse: val[xv ⊕ xv] = bool-to-val False
      unfolding unfoldXor isZero width by fastforce
    then show ?thesis
     by (metis (no-types, lifting) eval-bits-1-64 p(3,4) width xv xvv validDefIntConst
IntVal0
           Value.inject(1) bool-to-val.simps(2) evalDet new-int.simps unfold-const
wf-value-def)
  qed
  done

lemma exp-eliminate-redundant-false:
  shows exp[x ⊕ false] ≥ exp[x]
  using val-eliminate-redundant-false apply auto
  subgoal premises p for m p xa
    proof −
      obtain xa where xa: [m, p] ⊢ x ↦ xa
        using p(2) by blast
      then have val[xa ⊕ (IntVal 32 0)] ≠ UndefVal
        using evalDet p(2,3) by blast
      then have [m, p] ⊢ x ↦ val[xa ⊕ (IntVal 32 0)]
        using eval-unused-bits-zero xa by (cases xa; auto)
```

then show *?thesis*
using *evalDet p(2) xa* **by** *blast*
qed
done

Optimisations

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

optimization *XorShiftConstantRight*: $((\textit{const } x) \oplus y) \longmapsto y \oplus (\textit{const } x)$ *when* $\neg(\textit{is-ConstantExpr } y)$
using *size-flip-binary val-xor-commute* **by** *auto*

optimization *EliminateRedundantFalse*: $(x \oplus \textit{false}) \longmapsto x$
using *exp-eliminate-redundant-false* **by** *auto*

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 $val[-(\textit{new-int } b\ v)] \neq \textit{UndefVal}$
shows $val[-(-(\textit{new-int } b\ v))] = val[\textit{new-int } b\ v]$
by *simp*

lemma *val-distribute-sub*:
assumes $x \neq \textit{UndefVal} \wedge y \neq \textit{UndefVal}$

```
  shows   val[−(x − y)] = val[y − x]
  by (cases x; cases y; auto)


lemma exp-distribute-sub:
  shows exp[−(x − y)] ≥ exp[y − x]
  by (auto simp: val-distribute-sub evaltree-not-undef)

thm-oracles exp-distribute-sub

lemma exp-negative-cancel:
  shows exp[−(−x)] ≥ exp[x]
  apply auto
 by (metis (no-types, opaque-lifting) eval-unused-bits-zero intval-negate.elims new-int.simps
     intval-negate.simps(1) minus-equation-iff 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)))] ≥ exp[x >>> (const (new-int b y))]
  apply auto
  subgoal premises p for m p xa
  proof −
    obtain xa where xa: [m,p] ⊢ x ↦ xa
      using p(2) by auto
    then have 1: val[−(xa >> (IntVal b (take-bit b y)))] ≠ UndefVal
      using evalDet p(1,2) by blast
    then have 2: val[xa >> (IntVal b (take-bit b y))] ≠ UndefVal
      by auto
   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) zero-less-numeral
        signed-take-bit-int-less-exp-word size64 unfold-const wsst-TYs(3))
    then have 5: (0::nat) < b
      using eval-bits-1-64 p(4) by blast
    then have 6: b ⊑ (64::nat)
      using eval-bits-1-64 p(4) by blast
    then have 7: [m,p] ⊢ BinaryExpr BinURightShift x
               (ConstantExpr (IntVal b (take-bit b y))) ↦
                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 ⊑ (64::nat)
```

```
            by (simp add: 6)
          then have sg4: [m,p] ⊢ BinaryExpr BinURightShift x
                (ConstantExpr (IntVal b (take-bit b (word-of-nat n)))) ↦
                 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: −(−(x)) ⟼ x
  using exp-negative-cancel by blast


optimization DistributeSubtraction: −(x − y) ⟼ (y − x)
 apply (smt (verit, best) 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 exp-distribute-sub nat-add-left-cancel-less less-add-eq-less
add-Suc lessI
         trans-less-add2 size-binary-rhs Suc-eq-plus1 Nat.add-0-right old.nat.inject
         zero-less-Suc)
   using exp-distribute-sub by simp


optimization NegativeShift: −(x >> (const (new-int b y))) ⟼ x >>> (const
(new-int b y))
                                   when (stamp-expr x = IntegerStamp b' lo hi ∧ unat y
= (b' − 1))
  using exp-negative-shift by simp

end

end
theory TacticSolving
  imports Common
begin

fun size :: IRExpr ⇒ 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* < *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* ≈ *rhs*
assumes ∀ *m p v*. ([*m*, *p*] ⊢ *elhs* ↦ *v*) ⟶ *v* = *lhs*
assumes ∀ *m p v*. ([*m*, *p*] ⊢ *erhs* ↦ *v*) ⟶ *v* = *rhs*
assumes ∀ *m p v1 v2*. ([*m*, *p*] ⊢ *elhs* ↦ *v1*) ⟶ ([*m*, *p*] ⊢ *erhs* ↦ *v2*)
shows *elhs* ≥ *erhs*
by (*metis assms*(*4*) *le-expr-def evaltree-not-undef*)

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* ≈ *rhs*], *defer-tac*, *explore-cases x y*)

method *obtain-eval* **for** *exp*::*IRExpr* **and** *val*::*Value* =
(*rule meta-mp*[**where** *P*=⋀*m p v*. ([*m*, *p*] ⊢ *exp* ↦ *v*) ⟹ *v* = *val*], *defer-tac*)

method *solve* **for** *lhs rhs x y* :: *Value* =
(*match* **conclusion in** *size* - < *size* - ⇒ ‹*simp*›)*?*,
(*match* **conclusion in** (*elhs*::*IRExpr*) ≥ (*erhs*::*IRExpr*) **for** *elhs erhs* ⇒ ‹
(*obtain-approx-eq lhs rhs x y*)*?*›)

print-methods

thm *BinaryExprE*
optimization *opt-add-left-negate-to-sub*:
−*x* + *y* ⟼ *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*)] ≈ *val*[*y*−*x*]
by (*cases x*; *cases y*; *auto*)

optimization *distribute-sub*: −(*x*−*y*) ⟼ (*y*−*x*)
using *val-distribute-sub unfold-binary unfold-unary* **by** *auto*

lemma *val-xor-self-is-false*:
assumes *x* = *IntVal 32 v*
shows *val*[*x* ⊕ *x*] ≈ *val*[*false*]
by (*cases x*; *auto simp*: *assms*)

definition *wf-stamp* :: *IRExpr* ⇒ *bool* **where**
wf-stamp e = (∀ *m p v*. ([*m*, *p*] ⊢ *e* ↦ *v*) ⟶ *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* ⊕ *x*] >= *exp*[*false*]
by (*smt* (*z3*) *wf-value-def bin-eval.simps*(*8*) *bin-eval-new-int constantAsStamp.simps*(*1*) *evalDet*
int-signed-value-bounds new-int.simps new-int-take-bits unfold-binary unfold-const valid-int
valid-stamp.simps(*1*) *valid-value.simps*(*1*) *well-formed-equal-defn val-xor-self-is-false*

le-expr-def assms wf-stamp-def)

lemma *val-or-commute*[*simp*]:
val[*x* | *y*] = *val*[*y* | *x*]
by (*cases x*; *cases y*; *auto simp*: *or.commute*)

lemma *val-xor-commute*[*simp*]:
val[*x* ⊕ *y*] = *val*[*y* ⊕ *x*]
by (*cases x*; *cases y*; *auto simp*: *word-bw-comms*(*3*))

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

lemma *exp-or-commutative*:
exp[*x* | *y*] ≥ *exp*[*y* | *x*]
by *auto*

lemma *exp-xor-commutative*:
exp[*x* ⊕ *y*] ≥ *exp*[*y* ⊕ *x*]

by *auto*

lemma *exp-and-commutative*:
exp[*x* & *y*] ≥ *exp*[*y* & *x*]
by *auto*

— — New Optimisations - submitted and added into Graal —

lemma *OrInverseVal*:
assumes *n* = *IntVal 32 v*
shows *val*[*n* | ~*n*] ≈ *new-int 32* (−*1*)
apply (*auto simp*: *assms*)
by (*metis bit.disj-cancel-right mask-eq-take-bit-minus-one take-bit-or*)

optimization *OrInverse*: *exp*[*n* | ~*n*] ⟼ (*const* (*new-int 32* (*not 0*)))
when (*stamp-expr n* = *IntegerStamp 32 l h* ∧ *wf-stamp n*)
apply (*auto simp*: *Suc-lessI*)
subgoal premises *p* **for** *m p xa xaa*
proof −
obtain *nv* **where** *nv*: [*m*,*p*] ⊢ *n* ↦ *nv*
using *p*(*3*) **by** *auto*
obtain *nbits nvv* **where** *nvv*: *nv* = *IntVal nbits nvv*
by (*metis evalDet evaltree-not-undef intval-logic-negation.cases intval-not.simps*(*3*,*4*,*5*)
nv
p(*5*,*6*))
then have *width*: *nbits* = *32*
by (*metis Value.inject*(*1*) *nv p*(*1*,*2*) *valid-int wf-stamp-def*)
then have *stamp*: *constantAsStamp* (*IntVal 32* (*mask 32*)) =
(*IntegerStamp 32* (*int-signed-value 32* (*mask 32*)) (*int-signed-value*
32 (*mask 32*)))
by *auto*
have *wf*: *wf-value* (*IntVal 32* (*mask 32*))
unfolding *wf-value-def stamp* **apply** *auto* **by** *eval*+
then have *unfoldOr*: *val*[*nv* | ~*nv*] = (*new-int 32* (*or* (*not nvv*) *nvv*))
using *intval-or.simps OrInverseVal nvv width* **by** *auto*
then have *eq*: *val*[*nv* | ~*nv*] = *new-int 32* (*not 0*)
by (*simp add*: *unfoldOr*)
then show *?thesis*
by (*metis bit.compl-zero evalDet local.wf new-int.elims nv p*(*3*,*5*) *take-bit-minus-one-eq-mask*
unfold-const)
qed
done

optimization *OrInverse2*: *exp*[~*n* | *n*] ⟼ (*const* (*new-int 32* (*not 0*)))
when (*stamp-expr n* = *IntegerStamp 32 l h* ∧ *wf-stamp n*)
using *OrInverse exp-or-commutative* **by** *auto*

lemma *XorInverseVal*:
assumes *n* = *IntVal 32 v*
shows *val*[*n* ⊕ ~*n*] ≈ *new-int 32* (−*1*)

apply (*auto simp*: *assms*)
by (*metis* (*no-types*, *opaque-lifting*) *bit.compl-zero bit.xor-compl-right bit.xor-self take-bit-xor*
mask-eq-take-bit-minus-one)

optimization *XorInverse*: *exp*[*n* ⊕ ~*n*] ⟼ (*const* (*new-int 32* (*not 0*)))
when (*stamp-expr n* = *IntegerStamp 32 l h* ∧ *wf-stamp n*)
apply (*auto simp*: *Suc-lessI*)
subgoal premises *p* **for** *m p xa xaa*
proof−
obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *n* ↦ *xv*
using *p*(*3*) **by** *auto*
obtain *xb xvv* **where** *xvv*: *xv* = *IntVal xb xvv*
by (*metis evalDet evaltree-not-undef intval-logic-negation.cases intval-not.simps*(*3*,*4*,*5*) *xv*
p(*5*,*6*))
have *rhsDefined*: [*m*,*p*] ⊢ (*ConstantExpr* (*IntVal 32* (*mask 32*))) ↦ (*IntVal 32* (*mask 32*))
by (*metis ConstantExpr add.right-neutral add-less-cancel-left neg-one-value numeral-Bit0*
new-int-unused-bits-zero not-numeral-less-zero validDefIntConst zero-less-numeral
verit-comp-simplify1(*3*) *wf-value-def*)
have *w32*: *xb*=*32*
by (*metis Value.inject*(*1*) *p*(*1*,*2*) *valid-int xv xvv wf-stamp-def*)
then have *unfoldNot*: *val*[(¬*xv*)] = *new-int xb* (*not xvv*)
by (*simp add*: *xvv*)
have *unfoldXor*: *val*[*xv* ⊕ (¬*xv*)] =
(*if xb*=*xb then* (*new-int xb* (*xor xvv* (*not xvv*))) *else UndefVal*)
using *intval-xor.simps*(*1*) *XorInverseVal w32 xvv* **by** *auto*
then have *rhs*: *val*[*xv* ⊕ (¬*xv*)] = *new-int 32* (*mask 32*)
using *unfoldXor w32* **by** *auto*
then show *?thesis*
by (*metis evalDet neg-one.elims neg-one-value p*(*3*,*5*) *rhsDefined xv*)
qed
done

optimization *XorInverse2*: *exp*[(~*n*) ⊕ *n*] ⟼ (*const* (*new-int 32* (*not 0*)))
when (*stamp-expr n* = *IntegerStamp 32 l h* ∧ *wf-stamp n*)
using *XorInverse exp-xor-commutative* **by** *auto*

lemma *AndSelfVal*:
assumes *n* = *IntVal 32 v*
shows *val*[~*n* & *n*] = *new-int 32 0*
apply (*auto simp*: *assms*)
by (*metis take-bit-and take-bit-of-0 word-and-not*)

optimization *AndSelf*: *exp*[(~*n*) & *n*] ⟼ (*const* (*new-int 32* (*0*)))
when (*stamp-expr n* = *IntegerStamp 32 l h* ∧ *wf-stamp n*)
apply (*auto simp*: *Suc-lessI*) **unfolding** *size.simps*

by (*metis* (*no-types*) *val-and-commute ConstantExpr IntVal0 Value.inject*(*1*) *evalDet wf-stamp-def*
eval-bits-1-64 new-int.simps validDefIntConst valid-int wf-value-def AndSelf-Val)

optimization *AndSelf2*: *exp*[*n* & (~*n*)] ⟼ (*const* (*new-int 32* (*0*)))
when (*stamp-expr n* = *IntegerStamp 32 l h* ∧ *wf-stamp n*)
using *AndSelf exp-and-commutative* **by** *auto*

lemma *NotXorToXorVal*:
assumes *x* = *IntVal 32 xv*
assumes *y* = *IntVal 32 yv*
shows *val*[(~*x*) ⊕ (~*y*)] = *val*[*x* ⊕ *y*]
apply (*auto simp*: *assms*)
by (*metis* (*no-types*, *opaque-lifting*) *bit.xor-compl-left bit.xor-compl-right take-bit-xor*

word-not-not)

lemma *NotXorToXorExp*:
assumes *stamp-expr x* = *IntegerStamp 32 lx hx*
assumes *wf-stamp x*
assumes *stamp-expr y* = *IntegerStamp 32 ly hy*
assumes *wf-stamp y*
shows *exp*[(~*x*) ⊕ (~*y*)] ≥ *exp*[*x* ⊕ *y*]
apply *auto*
subgoal premises *p* **for** *m p xa xb*
proof −
obtain *xa* **where** *xa*: [*m*,*p*] ⊢ *x* ↦ *xa*
using *p* **by** *blast*
obtain *xb* **where** *xb*: [*m*,*p*] ⊢ *y* ↦ *xb*
using *p* **by** *blast*
then have *a*: *val*[(~*xa*) ⊕ (~*xb*)] = *val*[*xa* ⊕ *xb*]
by (*metis assms valid-int wf-stamp-def xa xb NotXorToXorVal*)
then show *?thesis*
by (*metis BinaryExpr bin-eval.simps*(*8*) *evalDet p*(*1*,*2*,*4*) *xa xb*)
qed
done

optimization *NotXorToXor*: *exp*[(~*x*) ⊕ (~*y*)] ⟼ (*x* ⊕ *y*)
when (*stamp-expr x* = *IntegerStamp 32 lx hx* ∧ *wf-stamp x*) ∧
(*stamp-expr y* = *IntegerStamp 32 ly hy* ∧ *wf-stamp y*)
using *NotXorToXorExp* **by** *simp*

end

— New optimisations - submitted, not added into Graal yet —

context *stamp-mask*
begin

```
lemma ExpIntBecomesIntValArbitrary:
  assumes stamp-expr x = IntegerStamp b xl xh
  assumes wf-stamp x
  assumes valid-value v (IntegerStamp b xl xh)
  assumes [m,p] ⊢ x ↦ v
  shows ∃xv. v = IntVal b xv
  using assms by (simp add: IRTreeEvalThms.valid-value-elims(3))

lemma OrGeneralization:
  assumes stamp-expr x = IntegerStamp b xl xh
  assumes stamp-expr y = IntegerStamp b yl yh
  assumes stamp-expr exp[x | y] = IntegerStamp b el eh
  assumes wf-stamp x
  assumes wf-stamp y
  assumes wf-stamp exp[x | y]
  assumes (or (↓x) (↓y)) = not 0
  shows exp[x | y] ≥ exp[(const (new-int b (not 0)))]
  using assms apply auto
  subgoal premises p for m p xvv yvv
  proof −
    obtain xv where xv: [m, p] ⊢ x ↦ IntVal b xv
      by (metis p(1,3,9) valid-int wf-stamp-def)
    obtain yv where yv: [m, p] ⊢ y ↦ IntVal b yv
      by (metis p(2,4,10) valid-int wf-stamp-def)
    obtain evv where ev: [m, p] ⊢ exp[x | y] ↦ IntVal b evv
      by (metis BinaryExpr bin-eval.simps(7) unfold-binary p(5,9,10,11) valid-int
wf-stamp-def
          assms(3))
    then have rhsWf: wf-value (new-int b (not 0))
        by (metis eval-bits-1-64 new-int.simps new-int-take-bits validDefIntConst
wf-value-def)
    then have rhs: (new-int b (not 0)) = val[IntVal b xv | IntVal b yv]
      using assms word-ao-absorbs(1)
      by (metis (no-types, opaque-lifting) bit.de-Morgan-conj word-bw-comms(2) xv
down-spec
           word-not-not yv bit.disj-conj-distrib intval-or.simps(1) new-int-bin.simps
ucast-id
          or.right-neutral)
    then have notMaskEq: (new-int b (not 0)) = (new-int b (mask b))
      by auto
    then show ?thesis
      by (metis neg-one.elims neg-one-value p(9,10) rhsWf unfold-const evalDet xv
yv rhs)
    qed
    done
end
```

```
phase TacticSolving
  terminating size
begin



lemma constEvalIsConst:
  assumes wf-value n
  shows [m,p] ⊢ exp[(const (n))] ↦ n
  by (simp add: assms IRTreeEval.evaltree.ConstantExpr)

lemma ExpAddCommute:
  exp[x + y] ≥ exp[y + x]
  by (auto simp add: Values.intval-add-sym)

lemma AddNotVal:
  assumes n = IntVal bv v
  shows val[n + (~n)] = new-int bv (not 0)
  by (auto simp: assms)

lemma AddNotExp:
  assumes stamp-expr n = IntegerStamp b l h
  assumes wf-stamp n
  shows exp[n + (~n)] ≥ exp[(const (new-int b (not 0)))]
  apply auto
  subgoal premises p for m p x xa
  proof −
    have xaDef: [m,p] ⊢ n ↦ xa
      by (simp add: p)
    then have xaDef2: [m,p] ⊢ n ↦ x
      by (simp add: p)
    then have xa = x
      using p by (simp add: evalDet)
    then obtain xv where xv: xa = IntVal b xv
      by (metis valid-int wf-stamp-def xaDef2 assms)
    have toVal: [m,p] ⊢ exp[n + (~n)] ↦ val[xa + (~xa)]
       by (metis UnaryExpr bin-eval.simps(1) evalDet p unary-eval.simps(3) un-
fold-binary xaDef)
    have wfInt: wf-value (new-int b (not 0))
      using validDefIntConst xaDef by (simp add: eval-bits-1-64 xv wf-value-def)
    have toValRHS: [m,p] ⊢ exp[(const (new-int b (not 0)))] ↦ new-int b (not 0)
      using wfInt by (simp add: constEvalIsConst)
    have isNeg1: val[xa + (~xa)] = new-int b (not 0)
      by (simp add: xv)
    then show ?thesis
      using toValRHS by (simp add: ‹(xa::Value) = (x::Value)›)
    qed
   done
```

optimization *AddNot*: *exp*[*n* + (~*n*)] ⟼ (*const* (*new-int b* (*not 0*)))
when (*stamp-expr n* = *IntegerStamp b l h* ∧ *wf-stamp n*)
apply (*simp add*: *Suc-lessI*) **using** *AddNotExp* **by** *force*

optimization *AddNot2*: *exp*[(~*n*) + *n*] ⟼ (*const* (*new-int b* (*not 0*)))
when (*stamp-expr n* = *IntegerStamp b l h* ∧ *wf-stamp n*)
apply (*simp add*: *Suc-lessI*) **using** *AddNot ExpAddCommute* **by** *simp*

lemma *TakeBitNotSelf*:
(*take-bit 32* (*not e*) = *e*) = *False*
by (*metis even-not-iff even-take-bit-eq zero-neq-numeral*)

lemma *ValNeverEqNotSelf*:
assumes *e* = *IntVal 32 ev*
shows *val*[*intval-equals* (¬*e*) *e*] = *val*[*bool-to-val False*]
by (*simp add*: *TakeBitNotSelf assms*)

lemma *ExpIntBecomesIntVal*:
assumes *stamp-expr x* = *IntegerStamp 32 xl xh*
assumes *wf-stamp x*
assumes *valid-value v* (*IntegerStamp 32 xl xh*)
assumes [*m*,*p*] ⊢ *x* ↦ *v*
shows ∃ *xv*. *v* = *IntVal 32 xv*
using *assms* **by** (*simp add*: *IRTreeEvalThms.valid-value-elims*(*3*))

lemma *ExpNeverNotSelf*:
assumes *stamp-expr x* = *IntegerStamp 32 xl xh*
assumes *wf-stamp x*
shows *exp*[*BinaryExpr BinIntegerEquals* (¬*x*) *x*] ≥
exp[(*const* (*bool-to-val False*))]
using *assms* **apply** *auto*
subgoal premises *p* **for** *m p xa xaa*
proof −
obtain *xa* **where** *xa*: [*m*,*p*] ⊢ *x* ↦ *xa*
using *p*(*5*) **by** *auto*
then obtain *xv* **where** *xv*: *xa* = *IntVal 32 xv*
by (*metis p*(*1*,*2*) *valid-int wf-stamp-def*)
then have *lhsVal*: [*m*,*p*] ⊢ *exp*[*BinaryExpr BinIntegerEquals* (¬*x*) *x*] ↦
val[*intval-equals* (¬*xa*) *xa*]
by (*metis p*(*3*,*4*,*5*,*6*) *unary-eval.simps*(*3*) *evaltree.BinaryExpr bin-eval.simps*(*13*)
xa UnaryExpr
evalDet)
have *wfVal*: *wf-value* (*IntVal 32 0*)
using *wf-value-def* **apply** *rule*
by (*metis IntVal0 intval-word.simps nat-le-linear new-int.simps numeral-le-iff*
wf-value-def
semiring-norm(*71*,*76*) *validDefIntConst verit-comp-simplify1*(*3*) *zero-less-numeral*)

then have *rhsVal*: [*m*,*p*] ⊢ *exp*[(*const* (*bool-to-val False*))] ↦ *val*[*bool-to-val False*]
by *auto*
then have *valEq*: *val*[*intval-equals* (¬*xa*) *xa*] = *val*[*bool-to-val False*]
using *ValNeverEqNotSelf* **by** (*simp add*: *xv*)
then show *?thesis*
by (*metis bool-to-val.simps*(*2*) *evalDet p*(*3*,*5*) *rhsVal xa*)
qed
done

optimization *NeverEqNotSelf*: *exp*[*BinaryExpr BinIntegerEquals* (¬*x*) *x*] ⟼
exp[(*const* (*bool-to-val False*))]
when (*stamp-expr x* = *IntegerStamp 32 xl xh* ∧ *wf-stamp x*)
apply (*simp add*: *Suc-lessI*) **using** *ExpNeverNotSelf* **by** *force*

— New optimisations - not submitted / added into Graal yet —

lemma *BinXorFallThrough*:
shows *bin*[(*x* ⊕ *y*) = *x*] ⟷ *bin*[*y* = *0*]
by (*metis xor.assoc xor.left-neutral xor-self-eq*)

lemma *valXorEqual*:
assumes *x* = *new-int 32 xv*
assumes *val*[*x* ⊕ *x*] ≠ *UndefVal*
shows *val*[*x* ⊕ *x*] = *val*[*new-int 32 0*]
using *assms* **by** (*cases x*; *auto*)

lemma *valXorAssoc*:
assumes *x* = *new-int b xv*
assumes *y* = *new-int b yv*
assumes *z* = *new-int b zv*
assumes *val*[(*x* ⊕ *y*) ⊕ *z*] ≠ *UndefVal*
assumes *val*[*x* ⊕ (*y* ⊕ *z*)] ≠ *UndefVal*
shows *val*[(*x* ⊕ *y*) ⊕ *z*] = *val*[*x* ⊕ (*y* ⊕ *z*)]
by (*simp add*: *xor.commute xor.left-commute assms*)

lemma *valNeutral*:
assumes *x* = *new-int b xv*
assumes *val*[*x* ⊕ (*new-int b 0*)] ≠ *UndefVal*
shows *val*[*x* ⊕ (*new-int b 0*)] = *val*[*x*]
using *assms* **by** (*auto*; *meson*)

lemma *ValXorFallThrough*:
assumes *x* = *new-int b xv*
assumes *y* = *new-int b yv*
shows *val*[*intval-equals* (*x* ⊕ *y*) *x*] = *val*[*intval-equals y* (*new-int b 0*)]
by (*simp add*: *assms BinXorFallThrough*)

lemma *ValEqAssoc*:
val[*intval-equals x y*] = *val*[*intval-equals y x*]

apply (*cases x*; *cases y*; *auto*) **by** (*metis* (*full-types*) *bool-to-val.simps*)

lemma *ExpEqAssoc*:
exp[*BinaryExpr BinIntegerEquals x y*] ≥ *exp*[*BinaryExpr BinIntegerEquals y x*]
by (*auto simp add*: *ValEqAssoc*)

lemma *ExpXorBinEqCommute*:
exp[*BinaryExpr BinIntegerEquals* (*x* ⊕ *y*) *y*] ≥ *exp*[*BinaryExpr BinIntegerEquals* (*y* ⊕ *x*) *y*]
using *exp-xor-commutative mono-binary* **by** *blast*

lemma *ExpXorFallThrough*:
assumes *stamp-expr x* = *IntegerStamp b xl xh*
assumes *stamp-expr y* = *IntegerStamp b yl yh*
assumes *wf-stamp x*
assumes *wf-stamp y*
shows *exp*[*BinaryExpr BinIntegerEquals* (*x* ⊕ *y*) *x*] ≥
exp[*BinaryExpr BinIntegerEquals y* (*const* (*new-int b 0*))]
using *assms* **apply** *auto*
subgoal premises *p* **for** *m p xa xaa ya*
proof −
obtain *b xv* **where** *xa*: [*m*,*p*] ⊢ *x* ↦ *new-int b xv*
using *intval-equals.elims*
by (*metis new-int.simps eval-unused-bits-zero p*(*1*,*3*,*5*) *wf-stamp-def valid-int*)
obtain *yv* **where** *ya*: [*m*,*p*] ⊢ *y* ↦ *new-int b yv*
by (*metis Value.inject*(*1*) *wf-stamp-def p*(*1*,*2*,*3*,*4*,*8*) *eval-unused-bits-zero xa new-int.simps*
valid-int)
then have *wfVal*: *wf-value* (*new-int b 0*)
by (*metis eval-bits-1-64 new-int.simps new-int-take-bits validDefIntConst wf-value-def xa*)
then have *eval*: [*m*,*p*] ⊢ *exp*[*BinaryExpr BinIntegerEquals y* (*const* (*new-int b 0*))] ↦
val[*intval-equals* (*xa* ⊕ *ya*) *xa*]
by (*metis* (*no-types*, *lifting*) *ValXorFallThrough constEvalIsConst bin-eval.simps*(*13*) *evalDet xa*
p(*5*,*6*,*7*,*8*) *unfold-binary ya*)
then show *?thesis*
by (*metis evalDet new-int.elims p*(*1*,*3*,*5*,*7*) *take-bit-of-0 valid-value.simps*(*1*) *wf-stamp-def xa*)
qed
done

lemma *ExpXorFallThrough2*:
assumes *stamp-expr x* = *IntegerStamp b xl xh*
assumes *stamp-expr y* = *IntegerStamp b yl yh*
assumes *wf-stamp x*
assumes *wf-stamp y*
shows *exp*[*BinaryExpr BinIntegerEquals* (*x* ⊕ *y*) *y*] ≥

exp[*BinaryExpr BinIntegerEquals x* (*const* (*new-int b 0*))]
by (*meson assms dual-order.trans ExpXorBinEqCommute ExpXorFallThrough*)

optimization *XorFallThrough1*: *exp*[*BinaryExpr BinIntegerEquals* ($x \oplus y$) *x*] $\longmapsto$

exp[*BinaryExpr BinIntegerEquals y* (*const* (*new-int b 0*))]
when (*stamp-expr x* = *IntegerStamp b xl xh* ∧ *wf-stamp x*) ∧
(*stamp-expr y* = *IntegerStamp b yl yh* ∧ *wf-stamp y*)
using *ExpXorFallThrough* **by** *force*

optimization *XorFallThrough2*: *exp*[*BinaryExpr BinIntegerEquals x* ($x \oplus y$)] $\longmapsto$

exp[*BinaryExpr BinIntegerEquals y* (*const* (*new-int b 0*))]
when (*stamp-expr x* = *IntegerStamp b xl xh* ∧ *wf-stamp x*) ∧
(*stamp-expr y* = *IntegerStamp b yl yh* ∧ *wf-stamp y*)
using *ExpXorFallThrough ExpEqAssoc* **by** *force*

optimization *XorFallThrough3*: *exp*[*BinaryExpr BinIntegerEquals* ($x \oplus y$) *y*] $\longmapsto$

exp[*BinaryExpr BinIntegerEquals x* (*const* (*new-int b 0*))]
when (*stamp-expr x* = *IntegerStamp b xl xh* ∧ *wf-stamp x*) ∧
(*stamp-expr y* = *IntegerStamp b yl yh* ∧ *wf-stamp y*)
using *ExpXorFallThrough2* **by** *force*

optimization *XorFallThrough4*: *exp*[*BinaryExpr BinIntegerEquals y* ($x \oplus y$)] $\longmapsto$

exp[*BinaryExpr BinIntegerEquals x* (*const* (*new-int b 0*))]
when (*stamp-expr x* = *IntegerStamp b xl xh* ∧ *wf-stamp x*) ∧
(*stamp-expr y* = *IntegerStamp b yl yh* ∧ *wf-stamp y*)
using *ExpXorFallThrough2 ExpEqAssoc* **by** *force*

end

context *stamp-mask*
begin

lemma *inEquivalence*:
assumes [*m*, *p*] ⊢ *y* ↦ *IntVal b yv*
assumes [*m*, *p*] ⊢ *x* ↦ *IntVal b xv*
shows (*and* (↑*x*) *yv*) = (↑*x*) ⟷ (*or* (↑*x*) *yv*) = *yv*
by (*metis word-ao-absorbs*(*3*) *word-ao-absorbs*(*4*))

lemma *inEquivalence2*:
assumes [*m*, *p*] ⊢ *y* ↦ *IntVal b yv*
assumes [*m*, *p*] ⊢ *x* ↦ *IntVal b xv*
shows (*and* (↑*x*) (↓*y*)) = (↑*x*) ⟷ (*or* (↑*x*) (↓*y*)) = (↓*y*)
by (*metis word-ao-absorbs*(*3*) *word-ao-absorbs*(*4*))

lemma *RemoveLHSOrMask*:
assumes (*and* (↑*x*) (↓*y*)) = (↑*x*)
assumes (*or* (↑*x*) (↓*y*)) = (↓*y*)
shows *exp*[*x* | *y*] ≥ *exp*[*y*]
using *assms* **apply** *auto*
subgoal premises *p* **for** *m p v*
proof −
obtain *b ev* **where** *exp*: [*m*, *p*] ⊢ *exp*[*x* | *y*] ↦ *IntVal b ev*
by (*metis BinaryExpr bin-eval.simps*(*7*) *p*(*3*,*4*,*5*) *bin-eval-new-int new-int.simps*)
from *exp* **obtain** *yv* **where** *yv*: [*m*, *p*] ⊢ *y* ↦ *IntVal b yv*
apply (*subst* (*asm*) *unfold-binary-width*) **by** *force+*
from *exp* **obtain** *xv* **where** *xv*: [*m*, *p*] ⊢ *x* ↦ *IntVal b xv*
apply (*subst* (*asm*) *unfold-binary-width*) **by** *force+*
then have *yv* = (*or xv yv*)
using *assms yv xv* **apply** *auto*
by (*metis* (*no-types*, *opaque-lifting*) *down-spec ucast-id up-spec word-ao-absorbs*(*1*)
word-or-not
word-ao-equiv word-log-esimps(*3*) *word-oa-dist word-oa-dist2*)
then have (*IntVal b yv*) = *val*[(*IntVal b xv*) | (*IntVal b yv*)]
apply *auto* **using** *eval-unused-bits-zero yv* **by** *presburger*
then show *?thesis*
by (*metis p*(*3*,*4*) *evalDet xv yv*)
qed
done

lemma *RemoveRHSAndMask*:
assumes (*and* (↑*x*) (↓*y*)) = (↑*x*)
assumes (*or* (↑*x*) (↓*y*)) = (↓*y*)
shows *exp*[*x* & *y*] ≥ *exp*[*x*]
using *assms* **apply** *auto*
subgoal premises *p* **for** *m p v*
proof −
obtain *b ev* **where** *exp*: [*m*, *p*] ⊢ *exp*[*x* & *y*] ↦ *IntVal b ev*
by (*metis BinaryExpr bin-eval.simps*(*6*) *p*(*3*,*4*,*5*) *new-int.simps bin-eval-new-int*)
from *exp* **obtain** *yv* **where** *yv*: [*m*, *p*] ⊢ *y* ↦ *IntVal b yv*
apply (*subst* (*asm*) *unfold-binary-width*) **by** *force+*
from *exp* **obtain** *xv* **where** *xv*: [*m*, *p*] ⊢ *x* ↦ *IntVal b xv*
apply (*subst* (*asm*) *unfold-binary-width*) **by** *force+*
then have *IntVal b xv* = *val*[(*IntVal b xv*) & (*IntVal b yv*)]
apply *auto*
by (*smt* (*verit*, *ccfv-threshold*) *or.right-neutral not-down-up-mask-and-zero-implies-zero*
p(*1*)
bit.conj-cancel-right word-bw-comms(*1*) *eval-unused-bits-zero yv word-bw-assocs*(*1*)
word-ao-absorbs(*4*) *or-eq-not-not-and*)
then show *?thesis*
by (*metis p*(*3*,*4*) *yv xv evalDet*)

qed
done

lemma *ReturnZeroAndMask*:
assumes *stamp-expr x = IntegerStamp b xl xh*
assumes *stamp-expr y = IntegerStamp b yl yh*
assumes *stamp-expr exp*[*x & y*] *= IntegerStamp b el eh*
assumes *wf-stamp x*
assumes *wf-stamp y*
assumes *wf-stamp exp*[*x & y*]
assumes (*and* (↑*x*) (↑*y*)) = *0*
shows *exp*[*x & y*] ≥ *exp*[*const* (*new-int b 0*)]
using *assms* **apply** *auto*
subgoal premises *p* **for** *m p v*
proof −
obtain *yv* **where** *yv*: [*m*, *p*] ⊢ *y* ↦ *IntVal b yv*
by (*metis valid-int wf-stamp-def assms*(*2*,*5*) *p*(*2*,*4*,*10*) *wf-stamp-def*)
obtain *xv* **where** *xv*: [*m*, *p*] ⊢ *x* ↦ *IntVal b xv*
by (*metis valid-int wf-stamp-def assms*(*1*,*4*) *p*(*3*,*9*) *wf-stamp-def*)
obtain *ev* **where** *exp*: [*m*, *p*] ⊢ *exp*[*x & y*] ↦ *IntVal b ev*
by (*metis BinaryExpr bin-eval.simps*(*6*) *p*(*5*,*9*,*10*,*11*) *assms*(*3*) *valid-int wf-stamp-def*)
then have *wfVal*: *wf-value* (*new-int b 0*)
by (*metis eval-bits-1-64 new-int.simps new-int-take-bits validDefIntConst wf-value-def*)
then have *lhsEq*: *IntVal b ev = val*[(*IntVal b xv*) & (*IntVal b yv*)]
by (*metis bin-eval.simps*(*6*) *yv xv evalDet exp unfold-binary*)
then have *newIntEquiv*: *new-int b 0 = IntVal b ev*
apply *auto* **by** (*smt* (*z3*) *p*(*6*) *eval-unused-bits-zero xv yv up-mask-and-zero-implies-zero*)
then have *isZero*: *ev = 0*
by *auto*
then show *?thesis*
by (*metis evalDet lhsEq newIntEquiv p*(*9*,*10*) *unfold-const wfVal xv yv*)
qed
done

end

phase *TacticSolving*
terminating *size*
begin

lemma *binXorIsEqual*:
bin[((*x* ⊕ *y*) = (*x* ⊕ *z*))] ⟷ *bin*[(*y = z*)]
by (*metis* (*no-types*, *opaque-lifting*) *BinXorFallThrough xor.left-commute xor-self-eq*)

```
lemma binXorIsDeterministic:
  assumes y ≠ z
  shows bin[x ⊕ y] ≠ bin[x ⊕ z]
  by (auto simp add: binXorIsEqual assms)

lemma ValXorSelfIsZero:
  assumes x = IntVal b xv
  shows val[x ⊕ x] = IntVal b 0
  by (simp add: assms)

lemma ValXorSelfIsZero2:
  assumes x = new-int b xv
  shows val[x ⊕ x] = IntVal b 0
  by (simp add: assms)

lemma ValXorIsAssociative:
  assumes x = IntVal b xv
  assumes y = IntVal b yv
  assumes val[(x ⊕ y)] ≠ UndefVal
  shows val[(x ⊕ y) ⊕ y] = val[x ⊕ (y ⊕ y)]
  by (auto simp add: word-bw-lcs(3) assms)

lemma ValXorIsAssociative2:
  assumes x = new-int b xv
  assumes y = new-int b yv
  assumes val[(x ⊕ y)] ≠ UndefVal
  shows val[(x ⊕ y) ⊕ y] = val[x ⊕ (y ⊕ y)]
  using ValXorIsAssociative by (simp add: assms)

lemma XorZeroIsSelf64:
  assumes x = IntVal 64 xv
  assumes val[x ⊕ (IntVal 64 0)] ≠ UndefVal
  shows  val[x ⊕ (IntVal 64 0)] = x
  using assms apply (cases x; auto)
  subgoal
  proof −
    have take-bit (LENGTH(64)) xv = xv
      unfolding Word.take-bit-length-eq by simp
    then show ?thesis
      by auto
   qed
  done

lemma ValXorElimSelf64:
  assumes x = IntVal 64 xv
  assumes y = IntVal 64 yv
  assumes val[x ⊕ y] ≠ UndefVal
  assumes val[y ⊕ y] ≠ UndefVal
```

shows *val*[*x* ⊕ (*y* ⊕ *y*)] = *x*
proof −
have *removeRhs*: *val*[*x* ⊕ (*y* ⊕ *y*)] = *val*[*x* ⊕ (*IntVal 64 0*)]
by (*simp add*: *assms*(*2*))
then have *XorZeroIsSelf*: *val*[*x* ⊕ (*IntVal 64 0*)] = *x*
using *XorZeroIsSelf64* **by** (*simp add*: *assms*(*1*))
then show *?thesis*
by (*simp add*: *removeRhs*)
qed

lemma *ValXorIsReverse64*:
assumes *x* = *IntVal 64 xv*
assumes *y* = *IntVal 64 yv*
assumes *z* = *IntVal 64 zv*
assumes *z* = *val*[*x* ⊕ *y*]
assumes *val*[*x* ⊕ *y*] ≠ *UndefVal*
assumes *val*[*z* ⊕ *y*] ≠ *UndefVal*
shows *val*[*z* ⊕ *y*] = *x*
using *ValXorIsAssociative ValXorElimSelf64 assms*(*1*,*2*,*4*,*5*) **by** *force*

lemma *valXorIsEqual-64*:
assumes *x* = *IntVal 64 xv*
assumes *val*[*x* ⊕ *y*] ≠ *UndefVal*
assumes *val*[*x* ⊕ *z*] ≠ *UndefVal*
shows *val*[*intval-equals* (*x* ⊕ *y*) (*x* ⊕ *z*)] = *val*[*intval-equals y z*]
using *assms* **apply** (*cases x*; *cases y*; *cases z*; *auto*)
subgoal premises *p* **for** *yv zv* **apply** (*cases* (*yv* = *zv*); *simp*)
subgoal premises *p*
proof −
have *isFalse*: *bool-to-val* (*yv* = *zv*) = *bool-to-val False*
by (*simp add*: *p*)
then have *unfoldTakebityv*: *take-bit LENGTH*(*64*) *yv* = *yv*
using *take-bit-length-eq* **by** *blast*
then have *unfoldTakebitzv*: *take-bit LENGTH*(*64*) *zv* = *zv*
using *take-bit-length-eq* **by** *blast*
then have *unfoldTakebitxv*: *take-bit LENGTH*(*64*) *xv* = *xv*
using *take-bit-length-eq* **by** *blast*
then have *lhs*: (*xor* (*take-bit LENGTH*(*64*) *yv*) (*take-bit LENGTH*(*64*) *xv*) =
xor (*take-bit LENGTH*(*64*) *zv*) (*take-bit LENGTH*(*64*) *xv*)) =
(*False*)
unfolding *unfoldTakebityv unfoldTakebitzv unfoldTakebitxv*
by (*simp add*: *binXorIsEqual word-bw-comms*(*3*) *p*)
then show *?thesis*
by (*simp add*: *isFalse*)
qed
done
done

lemma *ValXorIsDeterministic-64*:

assumes *x* = *IntVal 64 xv*
assumes *y* = *IntVal 64 yv*
assumes *z* = *IntVal 64 zv*
assumes *val*[*x* ⊕ *y*] ≠ *UndefVal*
assumes *val*[*x* ⊕ *z*] ≠ *UndefVal*
assumes *yv* ≠ *zv*
shows *val*[*x* ⊕ *y*] ≠ *val*[*x* ⊕ *z*]
by (*smt* (*verit*, *best*) *ValXorElimSelf64 ValXorIsAssociative ValXorSelfIsZero Value.distinct*(*1*)
assms Value.inject(*1*) *val-xor-commute valXorIsEqual-64*)

lemma *ExpIntBecomesIntVal-64*:
assumes *stamp-expr x* = *IntegerStamp 64 xl xh*
assumes *wf-stamp x*
assumes *valid-value v* (*IntegerStamp 64 xl xh*)
assumes [*m*,*p*] ⊢ *x* ↦ *v*
shows ∃ *xv*. *v* = *IntVal 64 xv*
using *assms* **by** (*simp add*: *IRTreeEvalThms.valid-value-elims*(*3*))

lemma *expXorIsEqual-64*:
assumes *stamp-expr x* = *IntegerStamp 64 xl xh*
assumes *stamp-expr y* = *IntegerStamp 64 yl yh*
assumes *stamp-expr z* = *IntegerStamp 64 zl zh*
assumes *wf-stamp x*
assumes *wf-stamp y*
assumes *wf-stamp z*
shows *exp*[*BinaryExpr BinIntegerEquals* (*x* ⊕ *y*) (*x* ⊕ *z*)] ≥
exp[*BinaryExpr BinIntegerEquals y z*]
using *assms* **apply** *auto*
subgoal premises *p* **for** *m p x1 y1 x2 z1*
proof −
obtain *xVal* **where** *xVal*: [*m*,*p*] ⊢ *x* ↦ *xVal*
using *p*(*8*) **by** *simp*
obtain *yVal* **where** *yVal*: [*m*,*p*] ⊢ *y* ↦ *yVal*
using *p*(*9*) **by** *simp*
obtain *zVal* **where** *zVal*: [*m*,*p*] ⊢ *z* ↦ *zVal*
using *p*(*12*) **by** *simp*
obtain *xv* **where** *xv*: *xVal* = *IntVal 64 xv*
by (*metis p*(*1*) *p*(*4*) *wf-stamp-def xVal ExpIntBecomesIntVal-64*)
then have *rhs*: [*m*,*p*] ⊢ *exp*[*BinaryExpr BinIntegerEquals y z*] ↦ *val*[*intval-equals yVal zVal*]
by (*metis BinaryExpr bin-eval.simps*(*13*) *evalDet p*(*7*,*8*,*9*,*10*,*11*,*12*,*13*) *valXorIsEqual-64 xVal*
yVal zVal)
then show *?thesis*
by (*metis xv evalDet p*(*8*,*9*,*10*,*11*,*12*,*13*) *valXorIsEqual-64 xVal yVal zVal*)
qed
done

optimization *XorIsEqual-64-1*: *exp*[*BinaryExpr BinIntegerEquals* $(x \oplus y)$ $(x \oplus z)$] $\longmapsto$
exp[*BinaryExpr BinIntegerEquals y z*]
when (*stamp-expr* x = *IntegerStamp 64 xl xh* $\wedge$ *wf-stamp* x) $\wedge$
(*stamp-expr* y = *IntegerStamp 64 yl yh* $\wedge$ *wf-stamp* y) $\wedge$
(*stamp-expr* z = *IntegerStamp 64 zl zh* $\wedge$ *wf-stamp* z)
using *expXorIsEqual-64* **by** *force*

optimization *XorIsEqual-64-2*: *exp*[*BinaryExpr BinIntegerEquals* $(x \oplus y)$ $(z \oplus x)$] $\longmapsto$
exp[*BinaryExpr BinIntegerEquals y z*]
when (*stamp-expr* x = *IntegerStamp 64 xl xh* $\wedge$ *wf-stamp* x) $\wedge$
(*stamp-expr* y = *IntegerStamp 64 yl yh* $\wedge$ *wf-stamp* y) $\wedge$
(*stamp-expr* z = *IntegerStamp 64 zl zh* $\wedge$ *wf-stamp* z)
by (*meson dual-order.trans mono-binary exp-xor-commutative expXorIsEqual-64*)

optimization *XorIsEqual-64-3*: *exp*[*BinaryExpr BinIntegerEquals* $(y \oplus x)$ $(x \oplus z)$] $\longmapsto$
exp[*BinaryExpr BinIntegerEquals y z*]
when (*stamp-expr* x = *IntegerStamp 64 xl xh* $\wedge$ *wf-stamp* x) $\wedge$
(*stamp-expr* y = *IntegerStamp 64 yl yh* $\wedge$ *wf-stamp* y) $\wedge$
(*stamp-expr* z = *IntegerStamp 64 zl zh* $\wedge$ *wf-stamp* z)
by (*meson dual-order.trans mono-binary exp-xor-commutative expXorIsEqual-64*)

optimization *XorIsEqual-64-4*: *exp*[*BinaryExpr BinIntegerEquals* $(y \oplus x)$ $(z \oplus x)$] $\longmapsto$
exp[*BinaryExpr BinIntegerEquals y z*]
when (*stamp-expr* x = *IntegerStamp 64 xl xh* $\wedge$ *wf-stamp* x) $\wedge$
(*stamp-expr* y = *IntegerStamp 64 yl yh* $\wedge$ *wf-stamp* y) $\wedge$
(*stamp-expr* z = *IntegerStamp 64 zl zh* $\wedge$ *wf-stamp* z)
by (*meson dual-order.trans mono-binary exp-xor-commutative expXorIsEqual-64*)

lemma *unwrap-bool-to-val*:
shows (*bool-to-val a* = *bool-to-val b*) = ($a = b$)
apply *auto* **using** *bool-to-val.elims* **by** *fastforce+*

lemma *take-bit-size-eq*:
shows *take-bit 64 a* = *take-bit LENGTH(64)* (*a::64 word*)
by *auto*

lemma *xorZeroIsEq*:
bin[(*xor xv yv*) = *0*] = *bin*[*xv* = *yv*]
by (*metis binXorIsEqual xor-self-eq*)

lemma *valXorEqZero-64*:
  assumes *val*[(*x* ⊕ *y*)] ≠ *UndefVal*
  assumes *x* = *IntVal 64 xv*
  assumes *y* = *IntVal 64 yv*
  shows *val*[*intval-equals* (*x* ⊕ *y*) ((*IntVal 64 0*))] = *val*[*intval-equals* (*x*) (*y*)]
  using *assms* **apply** (*cases x*; *cases y*; *auto*)
  unfolding *unwrap-bool-to-val take-bit-size-eq Word.take-bit-length-eq* **by** (*simp add*: *xorZeroIsEq*)

lemma *expXorEqZero-64*:
  assumes *stamp-expr x* = *IntegerStamp 64 xl xh*
  assumes *stamp-expr y* = *IntegerStamp 64 yl yh*
  assumes *wf-stamp x*
  assumes *wf-stamp y*
  shows *exp*[*BinaryExpr BinIntegerEquals* (*x* ⊕ *y*) (*const* (*IntVal 64 0*))] ≥
    exp[*BinaryExpr BinIntegerEquals* (*x*) (*y*)]
  using *assms* **apply** *auto*
  subgoal premises *p* **for** *m p x1 y1*
  proof −
    obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *x* ↦ *xv*
      using *p* **by** *blast*
    obtain *yv* **where** *yv*: [*m*,*p*] ⊢ *y* ↦ *yv*
      using *p* **by** *fast*
    obtain *xvv* **where** *xvv*: *xv* = *IntVal 64 xvv*
      by (*metis p*(*1*,*3*) *wf-stamp-def xv ExpIntBecomesIntVal-64*)
    obtain *yvv* **where** *yvv*: *yv* = *IntVal 64 yvv*
      by (*metis p*(*2*,*4*) *wf-stamp-def yv ExpIntBecomesIntVal-64*)
    have *rhs*: [*m*,*p*] ⊢ *exp*[*BinaryExpr BinIntegerEquals* (*x*) (*y*)] ↦ *val*[*intval-equals xv yv*]
      by (*smt* (*z3*) *BinaryExpr ValEqAssoc ValXorSelfIsZero Value.distinct*(*1*) *bin-eval.simps*(*13*) *xvv*
        evalDet p(*5*,*6*,*7*,*8*) *valXorIsEqual-64 xv yv*)
    then show *?thesis*
      by (*metis evalDet p*(*6*,*7*,*8*) *valXorEqZero-64 xv xvv yv yvv*)
  qed
  done

optimization *XorEqZero-64*: *exp*[*BinaryExpr BinIntegerEquals* (*x* ⊕ *y*) (*const* (*IntVal 64 0*))] ⟼
      exp[*BinaryExpr BinIntegerEquals* (*x*) (*y*)]
    when (*stamp-expr x* = *IntegerStamp 64 xl xh* ∧ *wf-stamp x*) ∧
      (*stamp-expr y* = *IntegerStamp 64 yl yh* ∧ *wf-stamp y*)
  using *expXorEqZero-64* **by** *fast*

lemma *xorNeg1IsEq*:
  bin[(*xor xv yv*) = (*not 0*)] = *bin*[*xv* = *not yv*]

using *xorZeroIsEq* **by** *fastforce*

lemma *valXorEqNeg1-64*:
assumes *val*[(*x* ⊕ *y*)] ≠ *UndefVal*
assumes *x* = *IntVal 64 xv*
assumes *y* = *IntVal 64 yv*
shows *val*[*intval-equals* (*x* ⊕ *y*) (*IntVal 64* (*not 0*))] = *val*[*intval-equals* (*x*) (¬*y*)]
using *assms* **apply** (*cases x*; *cases y*; *auto*)
unfolding *unwrap-bool-to-val take-bit-size-eq Word.take-bit-length-eq* **using** *xorNeg1IsEq* **by** *auto*

lemma *expXorEqNeg1-64*:
assumes *stamp-expr x* = *IntegerStamp 64 xl xh*
assumes *stamp-expr y* = *IntegerStamp 64 yl yh*
assumes *wf-stamp x*
assumes *wf-stamp y*
shows *exp*[*BinaryExpr BinIntegerEquals* (*x* ⊕ *y*) (*const* (*IntVal 64* (*not 0*)))] ≥
exp[*BinaryExpr BinIntegerEquals* (*x*) (¬*y*)]
using *assms* **apply** *auto*
subgoal premises *p* **for** *m p x1 y1*
proof −
obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *x* ↦ *xv*
using *p* **by** *blast*
obtain *yv* **where** *yv*: [*m*,*p*] ⊢ *y* ↦ *yv*
using *p* **by** *fast*
obtain *xvv* **where** *xvv*: *xv* = *IntVal 64 xvv*
by (*metis p*(*1*,*3*) *wf-stamp-def xv ExpIntBecomesIntVal-64*)
obtain *yvv* **where** *yvv*: *yv* = *IntVal 64 yvv*
by (*metis p*(*2*,*4*) *wf-stamp-def yv ExpIntBecomesIntVal-64*)
obtain *nyv* **where** *nyv*: [*m*,*p*] ⊢ *exp*[(¬*y*)] ↦ *nyv*
by (*metis ValXorSelfIsZero2 Value.distinct*(*1*) *intval-not.simps*(*1*) *yv yvv intval-xor.simps*(*2*)
UnaryExpr unary-eval.simps(*3*))
then have *nyvEq*: *val*[¬*yv*] = *nyv*
using *evalDet yv* **by** *fastforce*
obtain *nyvv* **where** *nyvv*: *nyv* = *IntVal 64 nyvv*
using *nyvEq intval-not.simps yvv* **by** *force*
have *notUndef*: *val*[*intval-equals xv* (¬*yv*)] ≠ *UndefVal*
using *bool-to-val.elims nyvEq nyvv xvv* **by** *auto*
have *rhs*: [*m*,*p*] ⊢ *exp*[*BinaryExpr BinIntegerEquals* (*x*) (¬*y*)] ↦ *val*[*intval-equals xv* (¬*yv*)]
by (*metis BinaryExpr bin-eval.simps*(*13*) *notUndef nyv nyvEq xv*)
then show *?thesis*
by (*metis bit.compl-zero evalDet p*(*6*,*7*,*8*) *rhs valXorEqNeg1-64 xvv yvv xv yv*)
qed
done

optimization *XorEqNeg1-64*: *exp*[*BinaryExpr BinIntegerEquals* (*x* ⊕ *y*) (*const*

(*IntVal 64* (*not 0*)))] ⟼
exp[*BinaryExpr BinIntegerEquals* (*x*) (¬*y*)]
when (*stamp-expr x* = *IntegerStamp 64 xl xh* ∧ *wf-stamp x*) ∧
(*stamp-expr y* = *IntegerStamp 64 yl yh* ∧ *wf-stamp y*)
using *expXorEqNeg1-64* **apply** *auto* **sorry**

end

end
theory *ProofStatus*
imports
AbsPhase
AddPhase
AndPhase
ConditionalPhase
MulPhase

NegatePhase
NewAnd
NotPhase
OrPhase
ShiftPhase
SignedDivPhase
SignedRemPhase
SubPhase
TacticSolving
XorPhase
begin

declare [[*show-types=false*]]
print-phases
print-phases!

print-methods

print-theorems

thm *opt-add-left-negate-to-sub*

export-phases ‹*Full*›

end