Veriopt Theories

March 23, 2023

Contents

1   Canonicalization Optimizations

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

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

lemma *size-non-add*[*size-simps*]: *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(11) size.simps(2)*
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*
  by (*smt* (*verit, best*) *less-Suc-eq less-add-Suc2 less-add-same-cancel1 linorder-neqE-nat*
not-add-less1 order-less-trans pos2 size.simps(4) size-binary-const size-non-add)

lemma *size-binary-rhs-b*[*size-simps*]:
  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(11) size.simps(4) 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* $\Rightarrow$ *Value* $\Rightarrow$ *bool*
  (**infix** $\approx$ *50*) **where**
  well-formed-equal v_1 v_2 = ($v_1 \neq$ *UndefVal* $\longrightarrow v_1 = v_2$)

lemma *well-formed-equal-defn* [*simp*]:
  well-formed-equal v_1 v_2 = ($v_1 \neq$ *UndefVal* $\longrightarrow v_1 = v_2$)
  unfolding *well-formed-equal-def* **by** *simp*

end

1.1   AbsNode Phase

theory *AbsPhase*
 imports
   Common
begin

phase *AbsNode*
  terminating *size*
begin

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

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

by (*smt* (*verit, ccfv-SIG*) *assms*(*1*) *assms*(*2*) *signed-take-bit-int-greater-eq-minus-exp*

    signed-take-bit-int-greater-eq-self-iff sint-0 sint-word-ariths(*4*) *word-sless-alt*)

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

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

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

          word-sless.rep-eq word-sless-def)
    then show *?thesis*
      using *abs-neg abs-pos signed.nless-le* **by** *auto*
  qed
next
  case *False*
  then show *?thesis* **using** *abs-pos* **by** *auto*
qed

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

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

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

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

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

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

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

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

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

lemma *less-eq-def*:

  shows *val-to-bool*(*val*[(*new-int b v1*) < (*new-int b v2*)]) ⟷ *v1* <*s v2*
  unfolding *new-int.simps intval-less-than.simps bool-to-val-bin.simps bool-to-val.simps*

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

lemma *val-abs-always-pos*:
  assumes *intval-abs* (*new-int b v*) = (*new-int b v′*)
  shows $0 \leq s\ v′$
  using *assms*
proof (*cases v = 0*)
  case *True*
  then have $v′ = 0$
    using *val-abs-zero assms*
      by (*smt* (*verit, ccfv-threshold*) *Suc-diff-1 bit-less-eq-def bot-nat-0.extremum*
diff-is-0-eq
        len-gt-0 len-of-numeral-defs(2) order-le-less signed-eq-0-iff take-bit-0
        take-bit-signed-take-bit take-bit-unwrap)
  then show *?thesis* **by** *simp*
next
  case *neq0*: *False*
  then show *?thesis*
  proof (*cases val-to-bool*(*val*[(*new-int b 0*) < (*new-int b v*)]))
    case *True*
    then show *?thesis* **using** *less-eq-def*
      using *assms val-abs-pos*
      by (*smt* (*verit, ccfv-SIG*) *One-nat-def Suc-leI bit.compl-one bit-less-eq-def*
      cancel-comm-monoid-add-class.diff-cancel diff-zero len-gt-0 len-of-numeral-defs(2)

      mask-0 mask-1 one-le-numeral one-neq-zero signed-word-eqI take-bit-dist-subL

       take-bit-minus-one-eq-mask take-bit-not-eq-mask-diff take-bit-signed-take-bit

        zero-le-numeral)
  next
    case *False*
    then have *val-to-bool*(*val*[(*new-int b v*) < (*new-int b 0*)])
      using *neq0 less-eq-def*
      by (*metis signed.neqE*)
      then show *?thesis* **using** *val-abs-neg less-eq-def* **unfolding** *new-int.simps*
intval-negate.simps
      by (*metis signed.nless-le take-bit-0*)
  qed

qed

lemma *intval-abs-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*)
  using *assms*
  by (*meson intval-abs.elims*)

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*)
  using *assms*
  using *intval-abs.simps*(*1*) **by** *presburger*

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 *intval-abs*(*intval-abs*(*x*)) ≠ *UndefVal*
  shows *intval-abs*(*intval-abs*(*x*)) = *intval-abs x*
  using *assms*
proof −
  obtain *b v* **where** *in-def*: *intval-abs x = new-int b v*
    using *assms intval-abs-elims mono-undef-abs* **by** *blast*
  then show *?thesis*
  proof (*cases val-to-bool*(*val*[(*new-int b v*) < (*new-int b 0*)]))
    case *True*
    then have *nested*: (*intval-abs* (*intval-abs x*)) = *new-int b* (−*v*)
      using *val-abs-neg intval-negate.simps in-def*
      by *simp*
    then have *x = new-int b* (−*v*)
      using *in-def True* **unfolding** *new-int.simps*
    by (*smt* (*verit, best*) *intval-abs.simps*(*1*) *less-eq-def less-eq-zero less-numeral-extra*(*1*)

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

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

qed
qed

lemma *val-abs-negate*:
　assumes *intval-abs (intval-negate x) $\neq$ UndefVal*
　shows　 *intval-abs (intval-negate x) = intval-abs x*
　using *assms* **apply** (*cases x; auto*)
　apply (*metis less-eq-def new-int.simps signed.dual-order.strict-iff-not signed.less-linear*

　　　　take-bit-0)
　by (*smt* (*verit, ccfv-threshold*) *add.inverse-neutral intval-abs.simps(1) less-eq-def*
less-eq-zero
　　less-numeral-extra(1) mask-1 mask-eq-take-bit-minus-one neg-one.elims neg-one-signed

　　new-int.simps one-le-numeral one-neq-zero signed.order.order-iff-strict take-bit-of-0

　　val-abs-always-pos)

Optimisations

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

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

end

end

1.2　AddNode Phase

theory *AddPhase*
　imports
　　Common
begin

phase *AddNode*
　terminating *size*
begin

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

optimization *AddShiftConstantRight*: $((const\ v)\ +\ y) \longmapsto y\ +\ (const\ v)$ *when*
¬(*is-ConstantExpr y*)
  using *size-non-const*
 apply (*metis add-2-eq-Suc′ less-Suc-eq plus-1-eq-Suc size.simps(11) size-non-add*)
 unfolding *le-expr-def*
 apply (*rule impI*)
 subgoal premises *1*
  apply (*rule allI impI*)+

  subgoal premises *2* **for** *m p va*
   apply (*rule BinaryExprE*[*OF 2*])
   subgoal premises *3* **for** *x ya*
    apply (*rule BinaryExpr*)
    using *3* **apply** *simp*
    using *3* **apply** *simp*
    using *3 binadd-commute* **apply** *auto*
    done
   done
  done
 done

optimization *AddShiftConstantRight2*: $((const\ v)\ +\ y) \longmapsto y\ +\ (const\ v)$ *when*
¬(*is-ConstantExpr y*)
 unfolding *le-expr-def*
  apply (*auto simp*: *intval-add-sym*)

 using *size-non-const*
 by (*metis add-2-eq-Suc′ lessI plus-1-eq-Suc size.simps(11) size-non-add*)

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

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

ML-val ‹@{*term* ‹*x* = *y*›}›

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

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

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

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

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

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

lemma *AddToSubHelperLowLevel*:
shows $intval\text{-}add\ (intval\text{-}negate\ e)\ y = intval\text{-}sub\ y\ e$ (**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 $val[x + (-e)] = val[x - e]$
  using *assms* **by** (*cases x*; *cases e*; *auto*)

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

Optimisations

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

optimization *AddRightNegateToSub*: $x + -e \longmapsto x - e$
  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** *blast*

end

end

1.3   AndNode Phase

theory *AndPhase*
  imports
    Common
    Proofs.StampEvalThms
begin

context *stamp-mask*
begin

lemma *AndRightFallthrough*: $(((and\ (not\ (\downarrow x))\ (\uparrow y)) = 0)) \longrightarrow exp[x\ \&\ y] \geq$
$exp[y]$
  apply *simp* **apply** (*rule impI*; (*rule allI*)+)
  apply (*rule impI*)
  subgoal premises *p* **for** *m p v*
  proof −
    obtain *xv* **where** *xv*: $[m,\ p] \vdash x \mapsto xv$
      using *p(2)* **by** *blast*
    obtain *yv* **where** *yv*: $[m,\ p] \vdash y \mapsto yv$
      using *p(2)* **by** *blast*
    have $v = val[xv\ \&\ yv]$
      by (*metis BinaryExprE bin-eval.simps(4) evalDet p(2) xv yv*)
    then have $v = yv$
    by (*smt (verit) eval-unused-bits-zero intval-and.elims new-int.elims new-int-bin.elims*
p(2)

        unfold-binary xv yv p(1) not-down-up-mask-and-zero-implies-zero)
    then show *?thesis* **using** *yv* **by** *simp*
  qed
  done

lemma *AndLeftFallthrough*: $(((and\ (not\ (\downarrow y))\ (\uparrow x)) = 0)) \longrightarrow exp[x\ \&\ y] \geq$
$exp[x]$
  apply *simp* **apply** (*rule impI*; (*rule allI*)+)
  apply (*rule impI*)
  subgoal premises *p* **for** *m p v*
  proof −
    obtain *xv* **where** *xv*: $[m,\ p] \vdash x \mapsto xv$
      using *p(2)* **by** *blast*
    obtain *yv* **where** *yv*: $[m,\ p] \vdash y \mapsto yv$
      using *p(2)* **by** *blast*
    have $v = val[xv\ \&\ yv]$
      by (*metis BinaryExprE bin-eval.simps(4) evalDet p(2) xv yv*)
    then have $v = xv$
    by (*smt (verit) and.commute eval-unused-bits-zero intval-and.elims new-int.simps*

      new-int-bin.simps p(2) unfold-binary xv yv p(1) not-down-up-mask-and-zero-implies-zero)
    then show *?thesis* **using** *xv* **by** *simp*
  qed
  done
end

phase *AndNode*
  terminating *size*
begin

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

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

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

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

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

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

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

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

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

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

qed

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

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

Optimisations

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

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

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

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

optimization *AndNeutral*: $(x$ & $\sim(const\ (IntVal\ b\ 0))) \longmapsto x$
  when (*wf-stamp x* $\wedge$ *stamp-expr x* = *IntegerStamp b lo hi*)
    apply *auto*
  by (*smt* (*verit*) *Value.sel*(*1*) *eval-unused-bits-zero intval-and.elims intval-word.simps*

    new-int.simps new-int-bin.simps take-bit-eq-mask)

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

optimization *AndLeftFallThrough*: $(x$ & $y) \longmapsto x$
                    when $(((and\ (not\ (IRExpr\text{-}down\ y))\ (IRExpr\text{-}up\ x)) = 0))$
  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*) $\longmapsto$ *ConstantExpr* (*bin-eval op v1 v2*)
 unfolding *le-expr-def*
 apply (*rule allI impI*)+
 subgoal premises *bin* **for** *m p v*
  print-facts
  apply (*rule BinaryExprE*[*OF bin*])
  subgoal premises *prems* **for** *x y*
   print-facts

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

16

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

print-facts

end

end

1.5   ConditionalNode Phase

theory *ConditionalPhase*
 imports
  Common
  Proofs.StampEvalThms
begin

phase *ConditionalNode*
 terminating *size*
begin

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

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

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*] $\mapsto v'$) $\wedge v' = IntVal\ b\ vv \wedge b > 0$
 using *assms*
 by (*metis eval-bits-1-64 intval-logic-negation.elims negation-preserve-eval unfold-unary*)

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

optimization *DefaultTrueBranch*: (*true ? x : y*) ⟼ x **.**

optimization *DefaultFalseBranch*: (*false ? x : y*) ⟼ y **.**

optimization *ConditionalEqualBranches*: (*e ? x : x*) ⟼ x **.**

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

optimization *condition-bounds-y*: ((u < v) ? x : y) ⟼ y
    when (*stamp-under* (*stamp-expr v*) (*stamp-expr u*) ∧ *wf-stamp u* ∧ *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*) *and-one-eq bool-to-val.simps(1) even-iff-mod-2-eq-zero
odd-iff-mod-2-eq-one val-to-bool.simps(1)*)

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

optimization *ConditionalEqualIsRHS*: ((x eq y) ? x : y) ⟼ y
  apply *auto*
  by (*smt* (*verit*) *Value.inject(1) bool-to-val.simps(2) bool-to-val-bin.simps evalDet*

    intval-equals.elims val-to-bool.elims(1))

optimization *normalizeX*: ((x eq const (IntVal 32 0)) ?
                    (const (IntVal 32 0)) : (const (IntVal 32 1))) ⟼ x
                    when (*IRExpr-up x = 1*) ∧ *stamp-expr x = IntegerStamp*
b 0 1

apply *auto*
subgoal premises *p* **for** *m p v xa*
  proof −
    obtain *xa* **where** *xa*: $[m,p] \vdash x \mapsto xa$
      using *p* **by** *blast*
    have *3*: $[m,p] \vdash$ *if val-to-bool (intval-equals xa (IntVal (32::nat) (0::64 word)))*
               then ConstantExpr (IntVal (32::nat) (0::64 word))
               else ConstantExpr (IntVal (32::nat) (1::64 word)) $\mapsto v$
      using *evalDet p(3) p(5) xa*
      using *p(4) p(6)* **by** *blast*
    then have *4*: *xa = IntVal 32 0 | xa = IntVal 32 1*
      sorry
    then have *6*: *v = xa*
      sorry
    then show *?thesis*
      using *xa* **by** *auto*
  qed
done

optimization *normalizeX2*: $((x\ eq\ (const\ (IntVal\ 32\ 1)))\ ?$
                 $(const\ (IntVal\ 32\ 1)) : (const\ (IntVal\ 32\ 0))) \longmapsto x$
                     when $(x = ConstantExpr\ (IntVal\ 32\ 0) | (x =$
ConstantExpr (IntVal 32 1))) **.**

optimization *flipX*: $((x\ eq\ (const\ (IntVal\ 32\ 0)))\ ?$
              $(const\ (IntVal\ 32\ 1)) : (const\ (IntVal\ 32\ 0))) \longmapsto$
              $x \oplus (const\ (IntVal\ 32\ 1))$
              when $(x = ConstantExpr\ (IntVal\ 32\ 0) | (x = ConstantExpr$
(IntVal 32 1))) **.**

optimization *flipX2*: $((x\ eq\ (const\ (IntVal\ 32\ 1)))\ ?$
              $(const\ (IntVal\ 32\ 0)) : (const\ (IntVal\ 32\ 1))) \longmapsto$
              $x \oplus (const\ (IntVal\ 32\ 1))$
              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 $([m,\ p] \vdash x \mapsto v) \longrightarrow (\exists\ vv.\ v = IntVal\ 32\ vv)$
  using *assms*
  by (*metis default-stamp valid-value-elims(3) wf-stamp-def*)

optimization *OptimiseIntegerTest*:
     $(((x\ \&\ (const\ (IntVal\ 32\ 1)))\ eq\ (const\ (IntVal\ 32\ 0)))\ ?$
     $(const\ (IntVal\ 32\ 0)) : (const\ (IntVal\ 32\ 1))) \longmapsto$

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


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




end

end
```

1.6 MulNode Phase

theory *MulPhase*
  imports
    Common
    Proofs.StampEvalThms
begin

fun *mul-size* :: *IRExpr ⇒ nat* **where**
  mul-size (UnaryExpr op e) = (mul-size e) + 2 |
  mul-size (BinaryExpr BinMul x y) = ((mul-size x) + (mul-size y) + 2) ∗ 2 |
  mul-size (BinaryExpr op x y) = (mul-size x) + (mul-size y) + 2 |
  mul-size (ConditionalExpr cond t f) = (mul-size cond) + (mul-size t) + (mul-size
f) + 2 |
  mul-size (ConstantExpr c) = 1 |
  mul-size (ParameterExpr ind s) = 2 |
  mul-size (LeafExpr nid s) = 2 |
  mul-size (ConstantVar c) = 2 |
  mul-size (VariableExpr x s) = 2

phase *MulNode*
  terminating *mul-size*
begin

lemma *bin-eliminate-redundant-negative*:
  uminus (x :: 'a::len word) ∗ uminus (y :: 'a::len word) = x ∗ y
  by *simp*

lemma *bin-multiply-identity*:
  (x :: 'a::len word) ∗ 1 = x
  by *simp*

lemma *bin-multiply-eliminate*:
  (x :: 'a::len word) ∗ 0 = 0
  by *simp*

lemma *bin-multiply-negative*:
  (x :: 'a::len word) ∗ uminus 1 = uminus x
  by *simp*

lemma *bin-multiply-power-2*:
  (x:: 'a::len word) ∗ (2^j) = x << j
  by *simp*

lemma *take-bit64* [*simp*]:
  fixes *w* :: *int64*
  shows *take-bit 64 w = w*
proof −

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

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

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

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

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

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

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

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

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

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

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

lemma *val-MulPower2Sub1*:
  fixes *i* :: *64 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 $(2::int)\ \hat{}\ 6 = 64$
        by *eval*
      then have *and i* (*mask 6*) = *i*
        using *mask-eq-iff* **by** (*simp add: less-mask-eq p(6)*)
      then have $x2 * ((2::64\ word)\ \hat{}\ unat\ i - (1::64\ word)) = (x2 * ((2::64\ word)$
$\hat{}\ unat\ i)) - x2$
        by (*simp add: right-diff-distrib'*)
      then show $x2 * ((2::64\ word)\ \hat{}\ unat\ i - (1::64\ word)) = x2 << unat\ (and\ i$
$(63::64\ word)) - x2$
        by (*simp add: 63 ‹and (i::64 word) (mask (6::nat)) = i›*)
      qed
      using *val-to-bool.simps(2)* **by** *presburger*

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

lemma *val-MulPower2AddPower2*:
  fixes *i j* :: *64 word*
  assumes $y = IntVal\ 64\ ((2\ \hat{}\ unat(i)) + (2\ \hat{}\ unat(j)))$
  and $0 < i$
  and $0 < j$
  and $i < 64$
  and $j < 64$
  and $x = new\text{-}int\ 64\ xx$
  shows $val[x * y] = val[(x << IntVal\ 64\ i) + (x << IntVal\ 64\ j)]$
  using *assms*
  proof −
    have *63*: $(63 :: int64) = mask\ 6$
      by *eval*
    then have $(2::int)\ \hat{}\ 6 = 64$

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

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

   val[(x ∗ IntVal 64 (2 ^ unat(i))) + (x ∗ IntVal 64 (2 ^ unat(j)))]

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

thm-oracles *val-MulPower2AddPower2*

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

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

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

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

thm-oracles *exp-multiply-neutral*

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

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

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

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

lemma *exp-MulPower2AddPower2*:
  fixes $i\ j$ :: *64 word*
  assumes $y = ConstantExpr\ (IntVal\ 64\ ((2 \mathbin{\char`^} unat(i)) + (2 \mathbin{\char`^} unat(j))))$
  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 $exp[x * y] \geq exp[(x << ConstantExpr\ (IntVal\ 64\ i)) + (x << Constant\text{-}$
$Expr\ (IntVal\ 64\ j))]$
  using *assms* **apply** *simp*
  by (*metis* (*no-types, lifting*) *ConstantExprE equiv-exprs-def unfold-binary*)

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

sorry

lemma *exp-distribute-multiplication*:
  shows $exp[(x * q) + (x * a)] \geq exp[x * (q + a)]$
  sorry

Optimisations

optimization *EliminateRedundantNegative*: $-x * -y \longmapsto x * y$
  using *mul-size.simps* **apply** *auto*
  by (*metis BinaryExpr val-eliminate-redundant-negative bin-eval.simps(2)*)

optimization *MulNeutral*: $x * ConstantExpr\ (IntVal\ b\ 1) \longmapsto x$
  using *exp-multiply-neutral* **by** *blast*

optimization *MulEliminator*: $x * ConstantExpr\ (IntVal\ b\ 0) \longmapsto const\ (IntVal\ b$
0)
   apply *auto*
  by (*smt (verit) Value.inject(1) constantAsStamp.simps(1) int-signed-value-bounds intval-mul.elims*
      mult-zero-right new-int.simps new-int-bin.simps take-bit-of-0 unfold-const
      valid-stamp.simps(1) valid-value.simps(1) val-multiply-zero)

optimization *MulNegate*: $x * -(const\ (IntVal\ b\ 1)) \longmapsto -x$
  apply *auto*
  by (*smt (verit) Value.distinct(1) Value.sel(1) add.inverse-inverse intval-mul.elims*

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

      take-bit-dist-neg unary-eval.simps(2) unfold-unary val-multiply-negative
      val-eliminate-redundant-negative val-multiply-negative wf-value-def)

fun *isNonZero* :: *Stamp* $\Rightarrow$ *bool* **where**
  isNonZero (IntegerStamp b lo hi) = $(lo > 0)$ |
  isNonZero - = False

lemma *isNonZero-defn*:
  assumes *isNonZero (stamp-expr x)*
  assumes *wf-stamp x*
  shows $([m, p] \vdash x \mapsto v) \longrightarrow (\exists\ vv\ b.\ (v = IntVal\ b\ vv \land val\text{-}to\text{-}bool\ val[(IntVal\ b$
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* **unfolding** *wf-stamp-def*
   using *eval vdef xstamp* **by** *fastforce*
  have *take-bit b vv = vv*
   using *eval eval-unused-bits-zero vdef* **by** *auto*
  then have *vv > 0*
   by (*metis bit-take-bit-iff int-signed-value.simps not-less-zero signed-eq-0-iff*
     signed-take-bit-eq-if-positive take-bit-0 take-bit-of-0 verit-comp-simplify1(*1*)
word-gt-0
     signed-above)
  then show *?thesis*
   using *vdef signed-above*
   by *simp*
qed
  done

optimization *MulPower2*: $x * y \longmapsto x << const$ (*IntVal 64 i*)
                 when (*i > 0* $\wedge$
                     64 > i $\wedge$
                     y = exp[*const* (*IntVal 64* (*2* $\hat{}$ *unat*(*i*)))])
  defer
  apply *simp* **apply** (*rule impI*; (*rule allI*)+; *rule impI*)
  subgoal premises *eval* **for** *m p v*
proof −
  obtain *xv* **where** *xv*: [*m, p*] ⊢ *x* ↦ *xv*
   using *eval*(*2*) **by** *blast*
  then obtain *xvv* **where** *xvv*: *xv = IntVal 64 xvv*
   by (*smt* (*verit*) *ConstantExprE bin-eval.simps*(*2*) *evalDet intval-bits.simps int-val-mul.elims*
     new-int-bin.simps unfold-binary eval)
  obtain *yv* **where** *yv*: [*m, p*] ⊢ *y* ↦ *yv*
   using *eval*(*1*) *eval*(*2*) **by** *blast*
  then have *lhs*: [*m, p*] ⊢ *exp*[*x* * *y*] ↦ *val*[*xv* * *yv*]
   by (*metis bin-eval.simps*(*2*) *eval*(*1*) *eval*(*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
     validStampIntConst wf-value-def valid-value.simps(*1*) *xv xvv*)
  then have *rhs*: [*m, p*] ⊢ *exp*[*x* << *const* (*IntVal 64 i*)] ↦ *val*[*xv* << (*IntVal 64 i*)]
   using *xv xvv* **using** *evaltree.BinaryExpr*
  by (*metis Value.simps*(*5*) *bin-eval.simps*(*8*) *intval-left-shift.simps*(*1*) *new-int.simps*)
  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*) *eval*(*2*) *evalDet lhs rhs*)
qed
  done

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

  defer
  apply *simp* **apply** (*rule impI*; (*rule allI*)+; *rule impI*)
 subgoal premises *p* **for** *m p v*
 proof −
  obtain *xv* **where** *xv*: $[m, p] \vdash x \mapsto xv$
   using *p* **by** *fast*
  then obtain *xvv* **where** *xvv*: $xv = IntVal \ 64 \ xvv$
   by (*smt* (*verit*) *p ConstantExprE bin-eval.simps(2) evalDet intval-bits.simps intval-mul.elims*
      new-int-bin.simps unfold-binary)
  obtain *yv* **where** *yv*: $[m, p] \vdash y \mapsto yv$
   using *p* **by** *blast*
  have *ygezero*: $y > ConstantExpr \ (IntVal \ 64 \ 0)$
   using *greaterConstant p wf-value-def* **by** *fastforce*
  then have *1*: $0 < i \ \wedge$
          $i < 64 \ \wedge$
          $y = ConstantExpr \ (IntVal \ 64 \ ((2 \ \hat{} \ unat(i)) + 1))$
   using *p* **by** *blast*
  then have *lhs*: $[m, p] \vdash exp[x * y] \mapsto val[xv * yv]$
   by (*metis bin-eval.simps(2) evalDet p(1) p(2) xv yv unfold-binary*)
  then have $[m, p] \vdash exp[const \ (IntVal \ 64 \ i)] \mapsto val[(IntVal \ 64 \ i)]$
   by (*metis wf-value-def verit-comp-simplify1(2) zero-less-numeral ConstantExpr*

     constantAsStamp.simps(1) take-bit64 validStampIntConst valid-value.simps(1))
  then have *rhs2*: $[m, p] \vdash exp[x \ll const \ (IntVal \ 64 \ i)] \mapsto val[xv \ll (IntVal \ 64 \ i)]$
   by (*metis Value.simps(5) bin-eval.simps(8) intval-left-shift.simps(1) new-int.simps xv xvv*
      evaltree.BinaryExpr)
  then have *rhs*: $[m, p] \vdash exp[(x \ll const \ (IntVal \ 64 \ i)) + x] \mapsto val[(xv \ll (IntVal \ 64 \ i)) + xv]$
    by (*metis* (*no-types, lifting*) *intval-add.simps(1) rhs2 bin-eval.simps(1) Value.simps(5)*
      evaltree.BinaryExpr intval-left-shift.simps(1) new-int.simps xv xvv)
  then have *simple*: $val[xv * (IntVal \ 64 \ (2 \ \hat{} \ unat(i)))] = val[xv \ll (IntVal \ 64 \ i)]$
   using *val-MulPower2* **sorry**
  then have $val[xv * yv] = val[(xv \ll (IntVal \ 64 \ i)) + xv]$
   sorry
  then show *?thesis*
   by (*metis 1 evalDet lhs p(2) rhs*)
 qed
 done

29

optimization *MulPower2Sub1*: $x * y \longmapsto (x << const\ (IntVal\ 64\ i)) - x$
                        when $(i > 0\ \wedge$
                                $64 > i\ \wedge$
                                $y = ConstantExpr\ (IntVal\ 64\ ((2\ \widehat{}\ unat(i)) - 1))\ )$
  defer
  apply *simp* **apply** (*rule impI*; (*rule allI*)+; *rule impI*)
 subgoal premises *p* **for** *m p v*
 proof −
  obtain *xv* **where** *xv*: $[m,p] \vdash x \mapsto xv$
   using *p* **by** *fast*
  then obtain *xvv* **where** *xvv*: $xv = IntVal\ 64\ xvv$
   by (*smt* (*verit*) *p ConstantExprE bin-eval.simps(2) evalDet intval-bits.simps*
intval-mul.elims
     new-int-bin.simps unfold-binary)
  obtain *yv* **where** *yv*: $[m,p] \vdash y \mapsto yv$
   using *p* **by** *blast*
  have *ygezero*: $y > ConstantExpr\ (IntVal\ 64\ 0)$
  by (*smt* (*verit, del-insts*) *eq-iff-diff-eq-0 mask-0 mask-eq-exp-minus-1 power-inject-exp*

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

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

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

     new-int.simps xv xvv)
  then have $val[xv * yv] = val[(xv << (IntVal\ 64\ i)) - xv]$
   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 *bin-distribute-and-over-or*:
  bin[z & (x | y)] = bin[(z & x) | (z & y)]
  by (*smt (verit, best) bit-and-iff bit-eqI bit-or-iff*)

lemma *intval-distribute-and-over-or*:
  val[z & (x | y)] = val[(z & x) | (z & y)]
  apply (*cases x*; *cases y*; *cases z*; *auto*)
  using *bin-distribute-and-over-or* **by** *blast+*

lemma *exp-distribute-and-over-or*:
  exp[z & (x | y)] ≥ exp[(z & x) | (z & y)]
  apply *simp* **using** *intval-distribute-and-over-or*
  using *BinaryExpr bin-eval.simps(4,5)*
  using *intval-or.simps(1)* **unfolding** *new-int-bin.simps new-int.simps* **apply** *auto*
  by (*metis bin-eval.simps(4) bin-eval.simps(5) intval-or.simps(2) intval-or.simps(5)*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma
  assumes $y = 0$
  shows $x + y = or\ x\ y$
  using *assms*
  by *simp*

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

context *stamp-mask*
begin

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

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

lemma *exp-eliminate-y*:
 and (↑*y*) (↑*z*) = *0* ⟶ *BinaryExpr BinAnd* (*BinaryExpr BinOr x y*) *z* ≥ *BinaryExpr BinAnd x z*
 apply *simp* **apply** (*rule impI*; *rule allI*; *rule allI*; *rule allI*)
 subgoal premises *p* **for** *m p v* **apply** (*rule impI*) **subgoal premises** *e*
 proof −
   obtain *xv* **where** *xv*: [*m,p*] ⊢ *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*]
     using *xv yv zv*
       by (*smt* (*verit, best*) *BinaryExprE bin-eval.simps(4) bin-eval.simps(5) e evalDet*)
   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*
     using *intval-up-and-zero-implies-zero*
     by (*metis calculation e intval-or.simps(5) p unfold-binary yv zv*)
   ultimately have *rhs*: *v = val*[*xv & zv*]
     using *intval-eliminate-y lhs* **by** *force*
   from *lhs rhs* **show** *?thesis*
     by (*metis BinaryExpr BinaryExprE bin-eval.simps(4) e xv zv*)
 qed
 done
 done

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

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

no-notation *LogicNegationNotation* (!-)

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

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

lemma *map-join-horner*:
  assumes *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 $^\wedge$ length [0..<j] $*$ horner-sum of-bool 2 (map f [j..<n])*
    using *horner-sum-append*
    by (*smt (verit) assms diff-le-self diff-zero le-add-same-cancel2 length-append length-map length-upt map-append upt-add-eq-append*)
  also have *... = horner-sum of-bool 2 (map f [0..<j]) + 2 $^\wedge$ length [0..<j] $*$ horner-sum of-bool 2 (map (λx. False) [j..<n])*
    using *assms*
    by (*metis calculation horner-sum-append length-map*)
  also have *... = horner-sum of-bool 2 (map f [0..<j])*
    using *zero-horner*
    using *mult-not-zero* **by** *auto*
  finally show *?thesis* **by** *simp*
qed

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

lemma *transfer-map*:
  assumes $\forall i.\ i < n \longrightarrow f\ i = f'\ i$
  shows *(map f [0..<n]) = (map f' [0..<n])*
  using *assms* **by** *simp*

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

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

proof −
  have $\forall\, i.\ i \geq n \longrightarrow \neg(\textit{bit zv i})$
    using *above-nth-not-set assms*(*1*)
    using *assms*(*2*) *not-may-implies-false*
  by (*smt* (*verit*, *ccfv-SIG*) *One-nat-def diff-less int-ops*(*6*) *leadingZerosAddHighestOne linorder-not-le nat-int-comparison*(*2*) *not-numeral-le-zero size64 zero-less-Suc zerosAboveHighestOne*)
  then show *?thesis*
    by (*metis* (*no-types*, *lifting*) *assms*(*1*) *diff-le-self split-horner*)
  qed
  also have ... = *horner-sum of-bool 2* (*map* (*bit* (*and* (*v mod 2*$\widehat{\ }$*n*) *zv*)) [*0*..<*64*])
    by (*meson bit-and-iff*)
  also have ... = *and* (*v mod 2*$\widehat{\ }$*n*) *zv*
    using *horner-sum-bit-eq-take-bit size64*
    by (*metis size-word.rep-eq take-bit-length-eq*)
  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* (*λi*::*nat*. *bit* ((*v*::*64 word*) *mod* (*2*::*64 word*) $\widehat{\ }$ (*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*) $\widehat{\ }$ *n*) *zv*)) [*0*::*nat*..<*64*::*nat*])› ‹*horner-sum of-bool* (*2*::*64 word*) (*map* (*λi*::*nat*. *bit* ((*v*::*64 word*) *mod* (*2*::*64 word*) $\widehat{\ }$ (*n*::*nat*)) *i* ∧ *bit* (*zv*::*64 word*) *i*) [*0*::*nat*..<*n*]) = *horner-sum of-bool* (*2*::*64 word*) (*map* (*λi*::*nat*. *bit* (*v mod* (*2*::*64 word*) $\widehat{\ }$ *n*) *i* ∧ *bit zv i*) [*0*::*nat*..<*64*::*nat*])› ‹*horner-sum of-bool* (*2*::*64 word*) (*map* (*λi*::*nat*. *bit* (*v*::*64 word*) *i* ∧ *bit* (*zv*::*64 word*) *i*) [*0*::*nat*..<*64*::*nat*]) = *horner-sum of-bool* (*2*::*64 word*) (*map* (*λi*::*nat*. *bit v i* ∧ *bit zv i*) [*0*::*nat*..<*n*::*nat*])› ‹*horner-sum of-bool* (*2*::*64 word*) (*map* (*λi*::*nat*. *bit* (*v*::*64 word*) *i* ∧ *bit* (*zv*::*64 word*) *i*) [*0*::*nat*..<*n*::*nat*]) = *horner-sum of-bool* (*2*::*64 word*) (*map* (*λi*::*nat*. *bit* (*v mod* (*2*::*64 word*) $\widehat{\ }$ *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*) $\widehat{\ }$ (*n*::*nat*)) (*zv*::*64 word*))) [*0*::*nat*..<*64*::*nat*]) = *and* (*v mod* (*2*::*64 word*) $\widehat{\ }$ *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* (*λi*::*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 $xv \leq (\uparrow x)$
  using *assms*
  by (*metis* (*no-types*, *lifting*) *and.idem 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*))

lemma *L2*:
  assumes *numberOfLeadingZeros* ($\uparrow z$) + *numberOfTrailingZeros* ($\uparrow y$) ≥ 64
  assumes $n = 64 - \textit{numberOfLeadingZeros}\ (\uparrow z)$
  assumes [*m*, *p*] ⊢ *z* ↦ *IntVal b zv*
  assumes [*m*, *p*] ⊢ *y* ↦ *IntVal b yv*
  shows *yv mod 2*$\widehat{\ }$*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])*
    using *up-mask-upper-bound assms(4)*
   by (*metis (no-types, opaque-lifting) and.right-neutral bit.conj-cancel-right bit.conj-disj-distribs(1)*
bit.double-compl horner-sum-bit-eq-take-bit take-bit-and ucast-id up-spec word-and-le1
word-not-dist(2))
  also have *horner-sum of-bool 2 (map (bit (↑y)) [0..<n]) = horner-sum of-bool 2*
(map (λx. False) [0..<n])
  proof −
    have *∀ i < n. ¬(bit (↑y) i)*
      using *assms(1,2) zerosBelowLowestOne*
      by (*metis add.commute add-diff-inverse-nat add-lessD1 leD le-diff-conv num-*
berOfTrailingZeros-def)
    then show *?thesis*
      by (*metis (full-types) transfer-map*)
  qed
  also have *horner-sum of-bool 2 (map (λx. False) [0..<n]) = 0*
    using *zero-horner*
    by *blast*
  finally show *?thesis*
    by *auto*
qed

thm-oracles *L1 L2*

lemma *unfold-binary-width-add*:
  shows *([m,p] ⊢ 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*)
proof (*intro iffI*)
  assume *3*: *?L*
  show *?R* **apply** (*rule evaltree.cases[OF 3]*)
    apply *force+* **apply** *auto[1]*
    apply (*smt (verit) intval-add.elims intval-bits.simps*)
    by *blast*
next
  assume *R*: *?R*
  then obtain *x y* **where** *[m,p] ⊢ xe ↦ IntVal b x*
    and *[m,p] ⊢ ye ↦ IntVal b y*
    and *new-int b val = bin-eval BinAdd (IntVal b x) (IntVal b y)*
    and *new-int b val ≠ UndefVal*
    by *auto*
  then show *?L*
    using *R* **by** *blast*

qed

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*)
proof (*intro iffI*)
  assume *3*: *?L*
  show *?R* **apply** (*rule evaltree.cases*[*OF 3*])
    apply *force+* **apply** *auto*[*1*] **using** *intval-and.elims intval-bits.simps*
    apply (*smt* (*verit*) *new-int.simps new-int-bin.simps take-bit-and*)
    by *blast*
next
  assume *R*: *?R*
  then obtain *x y* **where** [m,p] ⊢ *xe* ↦ *IntVal b x*
      and [m,p] ⊢ *ye* ↦ *IntVal b y*
      and *new-int b val* = *bin-eval BinAnd* (*IntVal b x*) (*IntVal b y*)
      and *new-int b val* ≠ *UndefVal*
    by *auto*
  then show *?L*
    using *R* **by** *blast*
qed

lemma *mod-dist-over-add-right*:
  fixes *a b c* :: *int64*
  fixes *n* :: *nat*
  assumes *1*: *0 < n*
  assumes *2*: *n < 64*
  shows (*a + b mod 2^n*) *mod 2^n* = (*a + b*) *mod 2^n*
  using *mod-dist-over-add*
  by (*simp add: 1 2 add.commute*)

lemma *numberOfLeadingZeros-range*:
  0 ≤ numberOfLeadingZeros n ∧ *numberOfLeadingZeros n ≤ Nat.size n*
  unfolding *numberOfLeadingZeros-def highestOneBit-def* **using** *max-set-bit*
  by (*simp add: highestOneBit-def leadingZeroBounds numberOfLeadingZeros-def*)

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

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

thm-oracles *improved-opt*

end

phase *NewAnd*
  terminating *size*
begin

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

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

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

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

end

end

1.8   NotNode Phase

theory *NotPhase*
  imports
    Common
begin

phase *NotNode*
  terminating *size*
begin

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

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

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

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

Optimisations

optimization *NotCancel*: $exp[^\sim(^\sim a)] \longmapsto a$
  by (*metis exp-not-cancel*)

end

end

1.9   OrNode Phase

theory *OrPhase*
  imports
    Common
begin

context *stamp-mask*
begin

Taking advantage of the truth table of or operations.

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

If row 2 never applies, that is, 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* ($\downarrow x$)) ($\uparrow y$)) = *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* = *intval-or* (*IntVal b xv*) (*IntVal b yv*)
      by (*metis bin-eval.simps(5) 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* = *intval-or* (*IntVal b xv*) (*IntVal b yv*)
    by (*smt* (*verit, ccfv-threshold*) *and.idem assms bit.conj-disj-distrib eval-unused-bits-zero*

$intval\text{-}or.simps(1)$ *new-int.simps new-int-bin.simps not-down-up-mask-and-zero-implies-zero*

        word-ao-absorbs(3) xv yv)

  then show *?thesis*

    using *xv vdef* **by** *presburger*

 qed

 done

lemma *OrRightFallthrough*:

 assumes $(and\ (not\ (\downarrow y))\ (\uparrow x)) = 0$

 shows $exp[x \mid y] \geq exp[y]$

 using *assms*

 apply *simp* **apply** $((rule\ allI)+;\ rule\ impI)$

 subgoal premises *eval* **for** *m p v*

 proof $-$

  obtain *b vv* **where** *e*: $[m,\ p] \vdash exp[x \mid y] \mapsto IntVal\ b\ vv$

    by *(metis BinaryExprE bin-eval-new-int new-int.simps eval(2))*

  from *e* **obtain** *xv* **where** *xv*: $[m,\ p] \vdash x \mapsto IntVal\ b\ xv$

    apply *(subst (asm) unfold-binary-width)*

    by *force+*

  from *e* **obtain** *yv* **where** *yv*: $[m,\ p] \vdash y \mapsto IntVal\ b\ yv$

    apply *(subst (asm) unfold-binary-width)*

    by *force+*

  have *vdef*: $v = intval\text{-}or\ (IntVal\ b\ xv)\ (IntVal\ b\ yv)$

    by *(metis bin-eval.simps(5) eval(2) evalDet unfold-binary xv yv)*

  have $\forall\ i.\ (bit\ xv\ i)\ |\ (bit\ yv\ i) = (bit\ yv\ i)$

    by *(metis assms bit-and-iff not-down-up-mask-and-zero-implies-zero xv yv)*

  then have $IntVal\ b\ yv = intval\text{-}or\ (IntVal\ b\ xv)\ (IntVal\ b\ yv)$

    by *(metis (no-types, lifting) assms eval-unused-bits-zero intval-or.simps(1)* *new-int.elims*

       new-int-bin.elims stamp-mask.not-down-up-mask-and-zero-implies-zero *stamp-mask-axioms*

      word-ao-absorbs(8) xv yv)

  then show *?thesis*

    using *vdef yv* **by** *presburger*

 qed

 done

end

phase *OrNode*

 terminating *size*

begin

lemma *bin-or-equal*:

 $bin[x \mid x] = bin[x]$

 by *simp*

lemma *bin-shift-const-right-helper*:
$x \mid y = y \mid x$
 by *simp*

lemma *bin-or-not-operands*:
$(^\sim x \mid {}^\sim y) = (^\sim(x \ \& \ y))$
 by *simp*

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

lemma *val-elim-redundant-false*:
  assumes $x = new\text{-}int \ b \ v$
  and $val[x \mid false] \neq UndefVal$
  shows $val[x \mid false] = val[x]$
   using *assms* **apply** (*cases x*; *auto*) **by** *presburger*

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

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

lemma *exp-or-equal*:
   $exp[x \mid x] \geq exp[x]$
    using *val-or-equal* **apply** *auto[1]*
   by (*smt* (*verit*, *ccfv-SIG*) *evalDet eval-unused-bits-zero intval-negate.elims int-
val-or.simps(2)*
       intval-or.simps(6) intval-or.simps(7) new-int.simps val-or-equal)

lemma *exp-elim-redundant-false*:
 $exp[x \mid false] \geq exp[x]$
   using *val-elim-redundant-false* **apply** *auto[1]*
   by (*smt* (*verit*) *Value.sel(1) eval-unused-bits-zero intval-or.elims new-int.simps*

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

Optimisations

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

by (*meson exp-or-equal*)

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

optimization *EliminateRedundantFalse*: *x* | *false* ⟼ *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*)
  apply *auto*[*1*]
 by (*metis BinaryExpr UnaryExpr bin-eval.simps*(*4*) *intval-not.simps*(*2*) *unary-eval.simps*(*3*)

    val-or-not-operands)

optimization *OrLeftFallthrough*:
  x | *y* ⟼ *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 *intval-left-shift x (intval-log2 val-c) = intval-mul x val-c*
   apply (*cases val-c; auto*) **using** *intval-left-shift.simps(1) intval-mul.simps(1)*
intval-log2.simps(1)
  sorry

lemma *e-intval*:
  n = intval-log2 val-c ∧ in-bounds n 0 32 ⟶
    intval-left-shift x (intval-log2 val-c) =
    intval-mul x val-c
proof (*rule impI*)
  assume *n = intval-log2 val-c ∧ in-bounds n 0 32*
  show *intval-left-shift x (intval-log2 val-c) =*
    intval-mul 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*
  using *BinaryExprE ConstantExprE bin-eval.simps(2,7)* **sorry**

end

end

1.11 SignedDivNode Phase

theory *SignedDivPhase*
  imports
    Common
begin

phase *SignedDivNode*
  terminating *size*
begin

lemma *val-division-by-one-is-self-32*:
  assumes *x = 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 $val[(x + y) - y] = val[x]$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Optimisations

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

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

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

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

optimization *SubThenAddLeft*: $(x - (x + y)) \longmapsto -y$
  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* **apply** *simp*
  using *exp-sub-then-left-sub* **by** *blast*

optimization *SubtractZero*: $(x - (const\ IntVal\ b\ 0)) \longmapsto x$
  apply *auto*
 by (*smt (verit) add.right-neutral diff-add-cancel eval-unused-bits-zero intval-sub.elims*

     intval-word.simps new-int.simps new-int-bin.simps)

thm-oracles *SubtractZero*

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

thm-oracles *SubNegativeValue*

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

using *assms*
using *is-IntVal-def* **by** *force*

optimization *ZeroSubtractValue*: $((const\ IntVal\ b\ 0) - x) \longmapsto (-x)$
$when\ (wf\text{-}stamp\ x \wedge stamp\text{-}expr\ x = IntegerStamp\ b\ lo$
$hi \wedge \neg(is\text{-}ConstantExpr\ x))$
 defer
 apply *auto* **unfolding** *wf-stamp-def*
 apply (*smt* (*verit*) *diff-0 intval-negate.simps(1) intval-sub.elims intval-word.simps*

   new-int-bin.simps unary-eval.simps(2) unfold-unary)
 using *add-2-eq-Suc′ size.simps(2) size-flip-binary* **by** *presburger*

optimization *SubSelfIsZero*: $(x - x) \longmapsto const\ IntVal\ b\ 0\ when$
$(wf\text{-}stamp\ x \wedge stamp\text{-}expr\ x = IntegerStamp\ b\ lo\ hi)$
 apply *simp-all*
 apply *auto*
 using *IRExpr.disc(42) One-nat-def size-non-const* **apply** *presburger*
 by (*smt* (*verit, best*) *wf-value-def ConstantExpr evalDet 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)

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*
  using *assms* **by** (*cases x*; *auto*)

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

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

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

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

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

lemma *exp-eliminate-redundant-false*:

54

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

Optimisations

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

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

optimization *EliminateRedundantFalse*: $(x \oplus false) \longmapsto x$
    using *exp-eliminate-redundant-false* **by** *blast*

end

end

1.15   NegateNode Phase

theory *NegatePhase*
 imports
   Common
begin

phase *NegateNode*
 terminating *size*
begin

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

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

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

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

thm-oracles *exp-distribute-sub*

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

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

then have *4*: *sint (signed-take-bit (b − Suc (0::nat)) (take-bit b y)) < (2::int)*
$\smallfrown$ *b div (2::int)*
   by (*metis Suc-le-lessD Suc-pred eval-bits-1-64 int-power-div-base p(4) signed-take-bit-int-less-exp-word size64 unfold-const wsst-TYs(3) zero-less-numeral*)
   then have *5*: *(0::nat) < b*
     using *eval-bits-1-64 p(4)* **by** *blast*
   then have *6*: *b ⊑ (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 (z3) add.left-commute add-2-eq-Suc′ add-diff-cancel-left′ is-ConstantExpr-def*

     less-Suc-eq-0-disj plus-1-eq-Suc size.simps(11) size-binary-const size-non-add

       zero-less-diff)
 using *exp-distribute-sub* **by** *simp*

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

end

end
theory *TacticSolving*
  imports *Common*
begin

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

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

phase *TacticSolving*
  terminating *size*
begin

1.16   AddNode

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

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

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

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

method *obtain-eval* **for** *exp*::*IRExpr* **and** *val*::*Value* =
  (*rule meta-mp*[**where** $P=\bigwedge m\ p\ v.\ ([m,\ p] \vdash exp \mapsto v) \Longrightarrow v = val$], *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*:
  $-x + y \longmapsto y - x$

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

1.17   NegateNode

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

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

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

definition *wf-stamp* :: *IRExpr* ⇒ *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 $exp[x \oplus x] >= exp[false]$

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

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

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

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

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

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

optimization *OrInverse*: $exp[n \mid {}^{\sim}n] \longmapsto (const\ (new\text{-}int\ 32\ (not\ 0)))$
                  when (*stamp-expr n* = *IntegerStamp 32 l h* $\wedge$ *wf-stamp n*)
  unfolding *size.simps* **apply** (*simp add*: *Suc-lessI*)
  apply *auto* **using** *OrInverseVal* **unfolding** *wf-stamp-def*
 by (*smt* (*z3*) *wf-value-def constantAsStamp.simps*(*1*) *evalDet int-signed-value-bounds*

    mask-eq-take-bit-minus-one new-int.elims new-int-take-bits unfold-const valid-int

    valid-stamp.simps(*1*) *valid-value.simps*(*1*) *well-formed-equal-defn*)

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

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

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

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

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

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

end

end
theory *ProofStatus*
  imports
    AbsPhase
    AddPhase
    AndPhase
    ConditionalPhase
    MulPhase

    NegatePhase
    NewAnd
    NotPhase
    OrPhase
    ShiftPhase
    SignedDivPhase
    SignedRemPhase
    SubPhase
    TacticSolving
    XorPhase
begin

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

61

print-methods

print-theorems

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

export-phases ‹*Full*›

end