

Veriopt Theories

September 14, 2022

Contents

1	Canonicalization Phase	1
2	Optimizations for Abs Nodes	2
3	Optimizations for Add Nodes	7
4	Optimizations for And Nodes	10
4.1	Conditional Expression	14
5	Optimizations for Mul Nodes	17
6	Optimizations for Not Nodes	38
7	Optimizations for Or Nodes	39
8	Optimizations for SignedDiv Nodes	42
9	Optimizations for SignedRem Nodes	43
10	Optimizations for Sub Nodes	43
11	Optimizations for Xor Nodes	49
12	Optimizations for Negate Nodes	51
12.1	AddNode	52
12.2	NegateNode	53

1 Canonicalization Phase

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

```

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

lemma size-non-add[size-simps]:  $op \neq \text{BinAdd} \implies \text{size } (\text{BinaryExpr } op \ a \ b) =$ 
 $\text{size } a + \text{size } b$ 
  by (induction op; auto)

lemma size-non-const[size-simps]:
   $\neg \text{is-ConstantExpr } y \implies 1 < \text{size } y$ 
  using size-pos apply (induction y; auto)
  subgoal premises prems for op a b
    apply (cases op = BinAdd)
    using size-non-add size-pos apply auto
    by (simp add: Suc-lessI one-is-add) +
  done

lemmas arith[size-simps] = Suc-leI add-strict-increasing

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

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

end
theory AbsPhase
  imports
    Common
begin

2 Optimizations for Abs Nodes

phase AbsNode
  terminating size
begin

```

```

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

lemma abs-neg:
  fixes  $v :: ('a :: \text{len word})$ 
  assumes  $v <_s 0$ 
  assumes  $-(2^{\wedge}(\text{Nat.size } v - 1)) <_s v$ 
  shows  $(\text{if } v <_s 0 \text{ then } -v \text{ else } v) = -v \wedge 0 <_s -v$ 
  by (smt (verit, ccfv-SIG) assms(1) assms(2) signed-take-bit-int-greater-eq-minus-exp

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

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

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

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

    word-sless.rep-eq word-sless-def)
  then show ?thesis

```

```

    using abs-neg abs-pos signed.nless-le by auto
  qed
next
  case False
  then show ?thesis using abs-pos by auto
qed

```

```

lemma wf-abs: 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  $\Rightarrow$  'a :: len word where
  bin-abs v = (if (v < 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  $\implies$  take-bit b (v1::64 word) = v1
  by (metis size64 size-word.rep-eq take-bit-length-eq)

```

```

lemma bit-less-eq-def:

```

```

fixes  $v1\ v2 :: 64\ \text{word}$ 
assumes  $b \leq 64$ 
shows  $\text{sint}(\text{signed-take-bit}(b - \text{Suc}(0::\text{nat}))(\text{take-bit } b\ v1))$ 
   $< \text{sint}(\text{signed-take-bit}(b - \text{Suc}(0::\text{nat}))(\text{take-bit } b\ v2)) \longleftrightarrow$ 
   $\text{signed-take-bit}(63::\text{nat})(\text{Word.rep } v1) < \text{signed-take-bit}(63::\text{nat})(\text{Word.rep}$ 
 $v2)$ 
using assms sorry

lemma less-eq-def:

  shows  $\text{val-to-bool}(\text{val}[(\text{new-int } b\ v1) < (\text{new-int } b\ v2)]) \longleftrightarrow v1 <_s v2$ 
unfolding new-int.simps intval-less-than.simps bool-to-val-bin.simps bool-to-val.simps
int-signed-value.simps apply (simp add: val-bool-unwrap)
apply auto unfolding word-sless-def apply auto
unfolding signed-def apply auto 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  $\text{intval-abs}(\text{new-int } b\ v) = (\text{new-int } b\ v')$ 
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  $\text{val-to-bool}(\text{val}[(\text{new-int } b\ v) < (\text{new-int } b\ 0)])$ 
using neq0 less-eq-def
by (metis signed.neqE)
            then show ?thesis using val-abs-neg less-eq-def unfolding new-int.simps
intval-negate.simps
by (metis signed.nless-le take-bit-0)

```

qed

qed

lemma *intval-abs-elim*:

assumes *intval-abs* $x \neq \text{UndefVal}$

shows $\exists t v . x = \text{IntVal } t v \wedge \text{intval-abs } x = \text{new-int } t \text{ (if int-signed-value } t v < 0 \text{ then } -v \text{ else } v)$

using *assms*

by (*meson intval-abs.elims*)

lemma *wf-abs-new-int*:

assumes *intval-abs* $(\text{IntVal } t v) \neq \text{UndefVal}$

shows $\text{intval-abs } (\text{IntVal } t v) = \text{new-int } t v \vee \text{intval-abs } (\text{IntVal } t v) = \text{new-int } t (-v)$

using *assms*

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

lemma *mono-undef-abs*:

assumes *intval-abs* $(\text{intval-abs } x) \neq \text{UndefVal}$

shows *intval-abs* $x \neq \text{UndefVal}$

using *assms*

by *force*

lemma *val-abs-idem*:

assumes *intval-abs* $(\text{intval-abs}(x)) \neq \text{UndefVal}$

shows *intval-abs* $(\text{intval-abs}(x)) = \text{intval-abs } x$

using *assms*

proof –

obtain $b v$ **where** *in-def*: *intval-abs* $x = \text{new-int } b v$

using *assms intval-abs-elim mono-undef-abs* **by** *blast*

then show *?thesis*

proof (*cases val-to-bool*(*val*[(*new-int* $b v$) < (*new-int* $b 0$)]))

case *True*

then have *nested*: $(\text{intval-abs } (\text{intval-abs } x)) = \text{new-int } b (-v)$

using *val-abs-neg intval-negate.simps in-def*

by *simp*

then have $x = \text{new-int } b (-v)$

using *in-def True unfolding new-int.simps*

by (*smt* (*verit*, *best*) *intval-abs.simps(1) less-eq-def less-eq-zero less-numeral-extra(1)*)

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

one-le-numeral one-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  $x \neq \text{UndefVal} \wedge \text{intval-negate } x \neq \text{UndefVal} \wedge \text{intval-abs}(\text{intval-negate } x) \neq \text{UndefVal}$ 
  shows  $\text{intval-abs } (\text{intval-negate } x) = \text{intval-abs } x$ 
  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:  $\text{abs}(\text{abs}(x)) \mapsto \text{abs}(x)$ 
  apply auto
  by (metis UnaryExpr unary-eval.simps(1) val-abs-idem)

optimization AbsNegate:  $\text{abs}(-x) \mapsto \text{abs}(x)$ 
  apply auto using val-abs-negate
  by (metis evaltree-not-undef unary-eval.simps(1) unfold-unary)

end

end

theory AddPhase
  imports
    Common
begin

```

3 Optimizations for Add Nodes

```

phase AddNode
  terminating size
begin

```

```

lemma binadd-commute:
  assumes  $\text{bin-eval } \text{BinAdd } x \ y \neq \text{UndefVal}$ 

```

shows *bin-eval BinAdd x y = bin-eval BinAdd y x*
using *assms intval-add-sym* **by** *simp*

optimization *AddShiftConstantRight*: $((\text{const } v) + y) \mapsto y + (\text{const } v)$ *when*
 $\neg(\text{is-ConstantExpr } y)$
using *size-non-const* **apply** *fastforce*
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*: $((\text{const } v) + y) \mapsto y + (\text{const } v)$ *when*
 $\neg(\text{is-ConstantExpr } y)$
unfolding *le-expr-def*
apply *(auto simp: intval-add-sym)*

using *size-non-const* **by** *fastforce*

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

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

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


```

lemma NeutralLeftSubVal:
  assumes  $e1 = \text{new-int } b \text{ ival}$ 
  shows  $\text{val}[(e1 - e2) + e2] \approx e1$ 
  apply simp using assms by (cases e1; cases e2; auto)

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

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

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

optimization RedundantSubAdd2:  $e_2 + (e_1 - e_2) \mapsto e_1$ 
  by (smt (verit, del-insts) BinaryExpr BinaryExprE RedundantSubAdd(1) binadd-commute le-expr-def rewrite-preservation.simps(1))

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

optimization AddToSub:  $-e + y \mapsto y - e$ 
  using AddToSubHelperLowLevel by auto

print-phases

```

```

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

```

```

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

```

```

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

```

Optimisations

```

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

```

```

optimization AddRightNegateToSub:  $x + -e \mapsto x - e$ 
  using AddToSubHelperLowLevel intval-add-sym by auto

```

```

optimization AddLeftNegateToSub:  $-e + y \mapsto y - e$ 
  using exp-add-left-negate-to-sub by blast

```

end

```

end
theory AndPhase
  imports
    Common

```

begin

4 Optimizations for And Nodes

```

phase AndNode
  terminating size
begin

```

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

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

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

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

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

lemma *val-and-sign-extend*:
assumes $e = (1 << In) - 1$
shows $\text{val}[(\text{intval-sign-extend } In \ Out \ x) \ \& \ (\text{IntVal } 32 \ e)] = \text{intval-zero-extend } In \ Out \ x$
using *assms* **apply** (*cases* x ; *auto*)
sorry

lemma *val-and-sign-extend-2*:
assumes $e = (1 << In) - 1 \wedge \text{intval-and } (\text{intval-sign-extend } In \ Out \ x) \ (\text{IntVal } 32 \ e) \neq \text{UndefVal}$
shows $\text{val}[(\text{intval-sign-extend } In \ Out \ x) \ \& \ (\text{IntVal } 32 \ e)] = \text{intval-zero-extend } In \ Out \ x$
using *assms* **apply** (*cases* x ; *auto*)
sorry

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

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

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

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

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*: $x \ \& \ x \longmapsto x$
using *exp-and-equal* **by** *blast*

optimization *AndShiftConstantRight*: $((\text{const } x) \ \& \ y) \longmapsto y \ \& \ (\text{const } x)$
when $\neg(\text{is-ConstantExpr } y)$
using *val-and-commute* **apply** *auto*
sorry

optimization *AndNots*: $(\sim x) \ \& \ (\sim y) \longmapsto \sim(x \mid y)$
using *exp-and-nots* **by** *auto*

optimization *AndSignExtend*: *BinaryExpr* *BinAnd* (*UnaryExpr* (*UnarySignExtend* *In Out*) *x*)

$(\text{ConstantExpr } (\text{IntVal } 32 \ e))$
 $\longmapsto (\text{UnaryExpr } (\text{UnaryZeroExtend } \text{In } \text{Out}) \ x)$
when $(e = (1 \ll \text{In}) - 1)$

apply *simp-all*
apply *auto*
sorry

optimization *AndNeutral*: $(x \ \& \ \sim(\text{const } (\text{IntVal } b \ 0))) \longmapsto x$

```

when (wf-stamp x ∧ stamp-expr x = IntegerStamp b lo hi)
apply auto using val-and-neutral
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)

```

end

```

context stamp-mask
begin

```

```

lemma AndRightFallthrough: (((and (not (↓ x)) (↑ y)) = 0)) → exp[x & y] ≥
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] ⊢ x ↦ xv
  using p(2) by blast
  obtain yv where yv: [m, p] ⊢ y ↦ yv
  using p(2) by blast
  have v = val[xv & yv]
  using p(2) xv yv
  by (metis BinaryExprE bin-eval.simps(4) evalDet)
  then have v = yv
  using p(1) not-down-up-mask-and-zero-implies-zero
  by (smt (verit) eval-unused-bits-zero intval-and.elims new-int.elims new-int-bin.elims
p(2) unfold-binary xv yv)
  then show ?thesis using yv by simp
qed
done

```

```

lemma AndLeftFallthrough: (((and (not (↓ y)) (↑ x)) = 0)) → exp[x & y] ≥
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] ⊢ x ↦ xv
  using p(2) by blast
  obtain yv where yv: [m, p] ⊢ y ↦ yv
  using p(2) by blast
  have v = val[xv & yv]
  using p(2) xv yv
  by (metis BinaryExprE bin-eval.simps(4) evalDet)
  then have v = xv
  using p(1) not-down-up-mask-and-zero-implies-zero

```

```

    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)
    then show ?thesis using xv by simp
qed
done

end

end
theory BinaryNode
  imports
    Common
begin

phase BinaryNde
  terminating size
begin

optimization BinaryFoldConstant: BinaryExpr op (const v1) (const v2)  $\mapsto$  ConstantExpr (bin-eval op v1 v2)
  apply (cases op; simp)
  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:  $\exists b\ vv. v = \text{new-int } b\ vv$  using bin-eval-new-int prems by fast
    show ?thesis
      unfolding prems x y xy
      apply (rule ConstantExpr)
      apply (rule validDefIntConst)
      using prems x y xy int sorry
    qed
  done
done

print-facts

end

```

end

4.1 Conditional Expression

```
theory ConditionalPhase
  imports
    Common
    Proofs.StampEvalThms
begin
```

```
phase ConditionalNode
  terminating size
begin
```

```
lemma negates: is-IntVal e  $\implies$  val-to-bool (val[e])  $\equiv \neg$ (val-to-bool (val[!e]))
  using intval-logic-negation.simps unfolding logic-negate-def
  sorry
```

```
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)
```

```
optimization NegateConditionFlipBranches: (!e) ? x : y  $\mapsto$  (e ? y : x)
  apply simp using negation-condition-intval
  by (smt (verit, ccfv-SIG) ConditionalExpr ConditionalExprE Value.collapse Value.exhaust-disc
    evaltree-not-undef intval-logic-negation.simps(4) intval-logic-negation.simps negates
    unary-eval.simps(4) unfold-unary)
```

```
optimization DefaultTrueBranch: (true ? x : y)  $\mapsto$  x .
```

```
optimization DefaultFalseBranch: (false ? x : y)  $\mapsto$  y .
```

```
optimization ConditionalEqualBranches: (e ? x : x)  $\mapsto$  x .
```

```
optimization condition-bounds-x: ((u < v) ? x : y)  $\mapsto$  x
  when (stamp-under (stamp-expr u) (stamp-expr v)  $\wedge$  wf-stamp u  $\wedge$  wf-stamp v)
  apply simp apply (rule impI) apply (rule allI)+ apply (rule impI)
  using stamp-under-defn
  by force
```

```
optimization condition-bounds-y: ((u < v) ? x : y)  $\mapsto$  y
  when (stamp-under (stamp-expr v) (stamp-expr u)  $\wedge$  wf-stamp u  $\wedge$  wf-stamp v)
  apply simp apply (rule impI) apply (rule allI)+ apply (rule impI)
  using stamp-under-defn-inverse
  by force
```

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

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

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

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

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

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

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

lemma *stamp-of-default*:
assumes *stamp-expr* $x = \text{default-stamp}$
assumes *wf-stamp* x
shows $([m, p] \vdash x \mapsto v) \longrightarrow (\exists vv. v = \text{IntVal } 32 \ vv)$
using *assms*
by $(\text{metis } \text{default-stamp } \text{valid-value-elim}(3) \ \text{wf-stamp-def})$

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

optimization *opt-optimize-integer-test-2*:
 $((x \ \& \ (\text{const } (\text{IntVal } 32 \ 1))) \text{ eq } (\text{const } (\text{IntVal } 32 \ 0)))) \ ?$

```

      (const (IntVal 32 0)) : (const (IntVal 32 1)))  $\mapsto$ 
       $\begin{array}{c} x \\ \text{when } (x = \text{ConstantExpr } (\text{IntVal } 32\ 0) \mid (x = \text{ConstantExpr } (\text{IntVal } \\ 32\ 1)))) \\ \text{done} \end{array}$ 

```

```

end

end
theory MulPhase
  imports
    Common
    Proofs.StampEvalThms
begin

```

5 Optimizations for Mul Nodes

```

phase MulNode
  terminating 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) * (2j) = 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 testt:
  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 testt by auto

lemma val-multiply-neutral:
  assumes x = new-int b v
  shows val[x] * (IntVal b 1) = val[x]
  using assms times-Value-def 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 add: times-Value-def)

lemma val-multiply-negative:
  assumes x = new-int b v
  shows x * intval-negate (IntVal b 1) = intval-negate x
  using assms times-Value-def
  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)

```

```

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

```

```

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

```

```

    then show  $x2 * ((2::64 \text{ word}) \wedge \text{unat } i + (1::64 \text{ word})) = x2 \ll \text{unat } (\text{and } i$ 
 $(63::64 \text{ word})) + x2$ 
    by (simp add: 63 and (i::64 word) (mask (6::nat)) = i)
  qed
done

```

lemma *val-MulPower2Sub1*:

```

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

```

lemma *val-distribute-multiplication*:

```

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

```

lemma *val-MulPower2AddPower2*:

```

  fixes  $i \ j :: 64 \text{ word}$ 
  assumes  $y = \text{IntVal } 64 ((2 \wedge \text{unat}(i)) + (2 \wedge \text{unat}(j)))$ 
  and  $0 < i$ 
  and  $0 < j$ 
  and  $i < 64$ 

```

```

and     $j < 64$ 
and     $x = \text{new-int } 64 \ xx$ 
shows   $x * y = \text{val}[(x \ll \text{IntVal } 64 \ i) + (x \ll \text{IntVal } 64 \ j)]$ 
using  assms
proof -
  have 63:  $(63 :: \text{int64}) = \text{mask } 6$ 
    by eval
  then have  $(2 :: \text{int}) \wedge 6 = 64$ 
    by eval
  then have  $n: \text{IntVal } 64 \ ((2 \wedge \text{unat}(i)) + (2 \wedge \text{unat}(j))) =$ 
     $\text{val}[(\text{IntVal } 64 \ (2 \wedge \text{unat}(i))) + (\text{IntVal } 64 \ (2 \wedge \text{unat}(j)))]$ 

    using assms by (cases i; cases j; auto)
  then have  $\text{val}[x * ((\text{IntVal } 64 \ (2 \wedge \text{unat}(i))) + (\text{IntVal } 64 \ (2 \wedge \text{unat}(j))))] =$ 
     $\text{val}[(x * \text{IntVal } 64 \ (2 \wedge \text{unat}(i))) + (x * \text{IntVal } 64 \ (2 \wedge \text{unat}(j)))]$ 

    using assms val-distribute-multiplication val-MulPower2 by simp
  then have  $\text{val}[(x * \text{IntVal } 64 \ (2 \wedge \text{unat}(i)))] = \text{val}[x \ll \text{IntVal } 64 \ i]$ 
    using assms val-MulPower2 sorry
  then show ?thesis
    sorry
qed

```

```

lemma exp-multiply-zero-64:
   $\text{exp}[x * (\text{const } (\text{IntVal } 64 \ 0))] \geq \text{ConstantExpr } (\text{IntVal } 64 \ 0)$ 
  using val-multiply-zero apply auto
  using 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
  by (smt (verit))

```

```

lemma exp-multiply-neutral:
   $\text{exp}[x * (\text{const } (\text{IntVal } b \ 1))] \geq x$ 
  using val-multiply-neutral apply auto sorry

```

```

lemma exp-MulPower2:
  fixes  $i :: 64 \ \text{word}$ 
  assumes  $y = \text{ConstantExpr } (\text{IntVal } 64 \ (2 \wedge \text{unat}(i)))$ 
  and     $0 < i$ 
  and     $i < 64$ 
  and     $\text{exp}[x > (\text{const } \text{IntVal } b \ 0)]$ 
  and     $\text{exp}[y > (\text{const } \text{IntVal } b \ 0)]$ 
  shows  $\text{exp}[x * y] \geq \text{exp}[x \ll \text{ConstantExpr } (\text{IntVal } 64 \ i)]$ 
  using assms apply simp using val-MulPower2
  by (metis ConstantExprE equiv-exprs-def unfold-binary)

```

```

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

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

optimization MulEliminator:  $x * \text{ConstantExpr } (\text{IntVal } b \ 0) \mapsto \text{const } (\text{IntVal } b \ 0)$ 
  apply auto using val-multiply-zero
  using 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)
  by (smt (verit))

optimization MulNegate:  $x * -(\text{const } (\text{IntVal } b \ 1)) \mapsto -x$ 
  apply auto using val-multiply-negative
  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 times-Value-def unary-eval.simps(2) unfold-unary
    val-eliminate-redundant-negative)

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

lemma isNonZero-defn:
  assumes isNonZero (stamp-expr x)
  assumes wf-stamp x
  shows ( $[m, p] \vdash x \mapsto v \longrightarrow (\exists vv \ b. (v = \text{IntVal } b \ vv \wedge \text{val-to-bool val}[(\text{IntVal } b \ 0) < v]))$ )
  apply (rule impI) subgoal premises eval
proof -
  obtain b lo hi where xstamp: stamp-expr x = IntegerStamp b lo hi
  using assms
  by (meson isNonZero.elims(2))
  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
    using signed-above
  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)
  then show ?thesis
    using vdef using signed-above
    by simp
qed
done

```

```

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

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

```



```

end

end
theory NewAnd
  imports
    Common
    Graph.Long
begin

lemma bin-distribute-and-over-or:
   $bin[z \ \& \ (x \mid y)] = bin[(z \ \& \ x) \mid (z \ \& \ y)]$ 
  by (smt (verit, best) bit-and-iff bit-eqI bit-or-iff)

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

lemma exp-distribute-and-over-or:
   $exp[z \ \& \ (x \mid y)] \geq exp[(z \ \& \ x) \mid (z \ \& \ y)]$ 
  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:
   $val[x \mid y] = val[y \mid x]$ 
  by (cases x; cases y; auto simp: or.commute)

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

lemma exp-and-commute:
   $exp[x \ \& \ z] \geq exp[z \ \& \ x]$ 
  apply simp using intval-and-commute by auto

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

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

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

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

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

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

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

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

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

lemma intval-or-absorb-and:

```

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

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

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

definition *IRExpr-up* :: *IRExpr* \Rightarrow *int64* **where**
IRExpr-up *e* = *not 0*

definition *IRExpr-down* :: *IRExpr* \Rightarrow *int64* **where**
IRExpr-down *e* = *0*

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

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

context *stamp-mask*
begin

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

```

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

```

lemma *exp-eliminate-y*:

```

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

```

lemma *leadingZeroBounds*:

```

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

```

lemma *above-nth-not-set*:

```

fixes x :: int64
assumes  $n = 64 - \text{numberOfLeadingZeros } x$ 
shows  $j > n \longrightarrow \neg(\text{bit } x \ j)$ 
using assms unfolding numberOfLeadingZeros-def

```

by (*smt* (*verit*, *ccfv-SIG*) *highestOneBit-def int-nat-eq int-ops*(6) *less-imp-of-nat-less max-set-bit size64 zerosAboveHighestOne*)

no-notation *LogicNegationNotation* (!-)

lemma *zero-horner*:

horner-sum of-bool 2 (map (λx. False) xs) = 0

apply (*induction xs*) **apply** *simp*

by *force*

lemma *zero-map*:

assumes $j \leq n$

assumes $\forall i. j \leq i \longrightarrow \neg(f\ i)$

shows $\text{map } f\ [0..<n] = \text{map } f\ [0..<j] @ \text{map } (\lambda x. \text{False})\ [j..<n]$

apply (*insert assms*)

by (*smt* (*verit*, *del-insts*) *add-diff-inverse-nat atLeastLessThan-iff bot-nat-0.extremum leD map-append map-eq-conv set-upt upt-add-eq-append*)

lemma *map-join-horner*:

assumes $\text{map } f\ [0..<n] = \text{map } f\ [0..<j] @ \text{map } (\lambda x. \text{False})\ [j..<n]$

shows $\text{horner-sum of-bool } (2::'a::\text{len word})\ (\text{map } f\ [0..<n]) = \text{horner-sum of-bool } 2\ (\text{map } f\ [0..<j])$

proof –

have $\text{horner-sum of-bool } (2::'a::\text{len word})\ (\text{map } f\ [0..<n]) = \text{horner-sum of-bool } 2\ (\text{map } f\ [0..<j]) + 2 \wedge \text{length } [0..<j] * \text{horner-sum of-bool } 2\ (\text{map } f\ [j..<n])$

using *horner-sum-append*

by (*smt* (*verit*) *assms diff-le-self diff-zero le-add-same-cancel2 length-append length-map length-upt map-append upt-add-eq-append*)

also have $\dots = \text{horner-sum of-bool } 2\ (\text{map } f\ [0..<j]) + 2 \wedge \text{length } [0..<j] * \text{horner-sum of-bool } 2\ (\text{map } (\lambda x. \text{False})\ [j..<n])$

using *assms*

by (*metis calculation horner-sum-append length-map*)

also have $\dots = \text{horner-sum of-bool } 2\ (\text{map } f\ [0..<j])$

using *zero-horner*

using *mult-not-zero* **by** *auto*

finally show *?thesis* **by** *simp*

qed

lemma *split-horner*:

assumes $j \leq n$

assumes $\forall i. j \leq i \longrightarrow \neg(f\ i)$

shows $\text{horner-sum of-bool } (2::'a::\text{len word})\ (\text{map } f\ [0..<n]) = \text{horner-sum of-bool } 2\ (\text{map } f\ [0..<j])$

apply (*rule map-join-horner*)

apply (*rule zero-map*)

using *assms* **by** *auto*

lemma *transfer-map*:

assumes $\forall i. i < n \longrightarrow f\ i = f'\ i$

```

shows (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-sum of-bool
    2 (map f' [0.. $n$ ])
  using assms using transfer-map
  by (smt (verit, best))

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

```

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

lemma *up-mask-upper-bound:*

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

lemma *L2:*

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

```

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

thm-oracles L1 L2

lemma unfold-binary-width-add:
shows  $([m,p] \vdash \text{BinaryExpr BinAdd } xe \ ye \mapsto \text{IntVal } b \ val) = (\exists \ x \ y. \$ 
 $(([m,p] \vdash xe \mapsto \text{IntVal } b \ x) \wedge$ 
 $([m,p] \vdash ye \mapsto \text{IntVal } b \ y) \wedge$ 
 $(\text{IntVal } b \ val = \text{bin-eval BinAdd } (\text{IntVal } b \ x) \ (\text{IntVal } b \ y)) \wedge$ 
 $(\text{IntVal } b \ val \neq \text{UndefVal})) \text{ (is } ?L = ?R)$ 
proof (intro iffI)
  assume  $\exists: ?L$ 
  show  $?R$  apply (rule evaltree.cases[OF 3])
    apply force+ apply auto[1]
    apply (smt (verit) intval-add.elims intval-bits.simps)
    by blast
next
  assume  $R: ?R$ 
  then obtain  $x \ y$  where  $[m,p] \vdash xe \mapsto \text{IntVal } b \ x$ 
    and  $[m,p] \vdash ye \mapsto \text{IntVal } b \ y$ 
    and  $\text{new-int } b \ val = \text{bin-eval BinAdd } (\text{IntVal } b \ x) \ (\text{IntVal } b \ y)$ 
    and  $\text{new-int } b \ val \neq \text{UndefVal}$ 
    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] ⊢ exp[x + y] ↦ IntVal b (xv + yv)
  apply (subst (asm) unfold-binary-width-and) by (metis add.right-neutral)
then obtain yv where yv: [m, p] ⊢ y ↦ IntVal b yv
  apply (subst (asm) unfold-binary-width-add) by blast
from addv obtain xv where xv: [m, p] ⊢ x ↦ IntVal b xv
  apply (subst (asm) unfold-binary-width-add) by blast
from val obtain zv where zv: [m, p] ⊢ z ↦ IntVal b zv
  apply (subst (asm) unfold-binary-width-and) by blast
have addv: [m, p] ⊢ exp[x + y] ↦ new-int b (xv + yv)
  apply (rule evaltree.BinaryExpr)
  using xv apply simp
  using yv apply simp
  by simp+
have lhs: [m, p] ⊢ exp[(x + y) & z] ↦ new-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] ⊢ exp[x & z] ↦ new-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 (↑z) > 0)
case True
  have n-bounds: 0 ≤ n ∧ n < 64
    using diff-le-self n numberOfLeadingZeros-range
    by (simp add: True)
  have and (xv + yv) zv = and ((xv + yv) mod 2n) zv
    using L1 n zv by blast
  also have ... = and ((xv + (yv mod 2n)) mod 2n) 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 2n) + (yv mod 2n)) mod 2n) 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 2n) mod 2n) zv
    using L2 n zv yv
    using assms by auto
  also have ... = and (xv mod 2n) zv
    using mod-mod-trivial
  by (smt (verit, best) and.idem take-bit-eq-mask take-bit-eq-mod word-bw-assocs(1))
  also have ... = and xv zv

```

```

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

thm-oracles improved-opt

lemma falseBelowN-nBelowLowest:
  assumes  $n \leq \text{Nat.size } a$ 
  assumes  $\forall i < n. \neg(\text{bit } a \ i)$ 
  shows lowestOneBit  $a \geq n$ 
proof (cases { $i. \text{bit } a \ i$ } = {})
case True
  then show ?thesis unfolding lowestOneBit-def MinOrHighest-def
    using assms(1) trans-le-add1 by presburger
next
case False
  have  $n \leq \text{Min } (\text{Collect } (\text{bit } a))$ 
  by (metis False Min-ge-iff assms(2) finite-bit-word linorder-le-less-linear mem-Collect-eq)
  then show ?thesis unfolding lowestOneBit-def MinOrHighest-def
    using False by presburger
qed

lemma noZeros:
  fixes  $a :: 64 \text{ word}$ 
  assumes zeroCount  $a = 0$ 
  shows  $i < \text{Nat.size } a \longrightarrow \text{bit } a \ i$ 
  using assms unfolding zeroCount-def size64
  using zeroCount-finite by auto

lemma zerosAboveOnly:
  fixes  $a :: 64 \text{ word}$ 

```

assumes $\text{numberOfLeadingZeros } a = \text{zeroCount } a$
shows $\neg(\text{bit } a \ i) \longrightarrow i \geq (64 - \text{numberOfLeadingZeros } a)$
sorry

lemma *consumes*:

assumes $\text{numberOfLeadingZeros } (\uparrow z) + \text{bitCount } (\uparrow z) = 64$
and $\uparrow z \neq 0$
and $\text{and } (\uparrow y) (\uparrow z) = 0$
shows $\text{numberOfLeadingZeros } (\uparrow z) + \text{numberOfTrailingZeros } (\uparrow y) \geq 64$
proof –
obtain n **where** $n = 64 - \text{numberOfLeadingZeros } (\uparrow z)$
by *simp*
then have $n = \text{bitCount } (\uparrow z)$
by (*metis add-diff-cancel-left' assms(1)*)
have $\text{numberOfLeadingZeros } (\uparrow z) = \text{zeroCount } (\uparrow z)$
using *assms(1) size64 ones-zero-sum-to-width*
by (*metis add.commute add-left-imp-eq*)
then have $\forall i. \neg(\text{bit } (\uparrow z) \ i) \longrightarrow i \geq n$
using *assms(1) zerosAboveOnly*
using $\langle (n::\text{nat}) = (64::\text{nat}) - \text{numberOfLeadingZeros } (\uparrow (z::\text{IRExpr})) \rangle$ **by** *blast*
then have $\forall i < n. \text{bit } (\uparrow z) \ i$
using *leD* **by** *blast*
then have $\forall i < n. \neg(\text{bit } (\uparrow y) \ i)$
using *assms(3)*
by (*metis bit.conj-cancel-right bit-and-iff bit-not-iff*)
then have $\text{lowestOneBit } (\uparrow y) \geq n$
by (*simp add: \langle (n::nat) = (64::nat) - numberOfLeadingZeros (\uparrow (z::IRExpr)) \rangle*)
falseBelowN-nBelowLowest size64
then have $n \leq \text{numberOfTrailingZeros } (\uparrow y)$
unfolding *numberOfTrailingZeros-def*
by *simp*
have $\text{card } \{i. i < n\} = \text{bitCount } (\uparrow z)$
by (*simp add: \langle (n::nat) = bitCount (\uparrow (z::IRExpr)) \rangle*)
then have $\text{bitCount } (\uparrow z) \leq \text{numberOfTrailingZeros } (\uparrow y)$
using $\langle (n::\text{nat}) \sqsubseteq \text{numberOfTrailingZeros } (\uparrow (y::\text{IRExpr})) \rangle$ **by** *auto*
then show *?thesis* **using** *assms(1)* **by** *auto*
qed

thm-oracles *consumes*

lemma *right*:

assumes $\text{numberOfLeadingZeros } (\uparrow z) + \text{bitCount } (\uparrow z) = 64$
assumes $\uparrow z \neq 0$
assumes $\text{and } (\uparrow y) (\uparrow z) = 0$
shows $\text{exp}[(x + y) \ \& \ z] \geq \text{exp}[x \ \& \ z]$

```

apply simp apply (rule allI)+
  subgoal premises p for m p v apply (rule impI) subgoal premises e
proof –
  obtain j where j: j = highestOneBit ( $\uparrow z$ )
    by simp
  obtain xv b where xv: [m,p]  $\vdash x \mapsto \text{IntVal } b \text{ } xv$ 
    using e
    by (metis EvalTreeE(5) bin-eval-inputs-are-ints bin-eval-new-int new-int.simps)
  obtain yv where yv: [m,p]  $\vdash y \mapsto \text{IntVal } b \text{ } yv$ 
    using e EvalTreeE(5) bin-eval-inputs-are-ints bin-eval-new-int new-int.simps
    by (smt (verit) Value.sel(1) bin-eval.simps(1) evalDet intval-add.elims xv)
  obtain xyv where xyv: [m, p]  $\vdash \text{exp}[x + y] \mapsto \text{IntVal } b \text{ } xyv$ 
    using e EvalTreeE(5) bin-eval-inputs-are-ints bin-eval-new-int new-int.simps
    xv yv
    by (metis BinaryExpr Value.distinct(1) bin-eval.simps(1) intval-add.simps(1))
  then obtain zv where zv: [m,p]  $\vdash z \mapsto \text{IntVal } b \text{ } zv$ 
    using e EvalTreeE(5) bin-eval-inputs-are-ints bin-eval-new-int new-int.simps
    Value.sel(1) bin-eval.simps(4) evalDet intval-and.elims
    by (smt (verit) new-int-bin.simps)
  have xyv = take-bit b (xv + yv)
    using xv yv xyv
    by (metis BinaryExprE Value.sel(2) bin-eval.simps(1) evalDet intval-add.simps(1))
  then have v = IntVal b (take-bit b (and (take-bit b (xv + yv)) zv))
    using zv
    by (smt (verit) EvalTreeE(5) Value.sel(1) Value.sel(2) bin-eval.simps(4) e
evalDet intval-and.elims new-int.simps new-int-bin.simps xyv)
    then have veval: v = IntVal b (and (xv + yv) zv)
    by (metis (no-types, lifting) eval-unused-bits-zero take-bit-eq-mask word-bw-comms(1)
word-bw-lcs(1) zv)
    have obligation: (and (xv + yv) zv) = (and xv zv)  $\implies$  [m,p]  $\vdash \text{BinaryExpr}$ 
BinAnd x z  $\mapsto v$ 
    by (smt (verit) EvalTreeE(5) Value.inject(1)  $\langle (v::\text{Value}) = \text{IntVal } (b::\text{nat})$ 
(take-bit b (and (take-bit b ((xv::64 word) + (yv::64 word))) (zv::64 word)))  $\rangle \langle (xyv::64$ 
word) = take-bit (b::nat) ((xv::64 word) + (yv::64 word))  $\rangle$  bin-eval.simps(4) e
evalDet eval-unused-bits-zero evaltree.simps intval-and.simps(1) take-bit-and xv xyv
zv)
    have per-bit:  $\forall n . \text{bit } (\text{and } (xv + yv) \text{ } zv) \text{ } n = \text{bit } (\text{and } xv \text{ } zv) \text{ } n \implies (\text{and } (xv +$ 
yv) zv) = (and xv zv)
    by (simp add: bit-eq-iff)
  show ?thesis
    apply (rule obligation)
    apply (rule per-bit)
    apply (rule allI)
    subgoal for n
  proof (cases n  $\leq j$ )
    case True

  then show ?thesis sorry

```

```

next
  case False
  then have  $\neg(\text{bit } zv \ n)$ 
    by (metis j linorder-not-less not-may-implies-false zerosAboveHighestOne zv)
  then have  $v: \neg(\text{bit } (and \ (xv + yv) \ zv) \ n)$ 
    by (simp add: bit-and-iff)
  then have  $v': \neg(\text{bit } (and \ xv \ zv) \ n)$ 
    by (simp add:  $\langle \neg \text{bit } (zv::64 \ word) \ (n::nat) \rangle \text{ bit-and-iff}$ )
  from  $v \ v'$  show ?thesis
    by simp
qed
done
qed
done
done

end

lemma ucast-zero:  $(ucast \ (0::int64)::int32) = 0$ 
  by simp

lemma ucast-minus-one:  $(ucast \ (-1::int64)::int32) = -1$ 
  apply transfer by auto

interpretation simple-mask: stamp-mask
  IRExpr-up :: IRExpr  $\Rightarrow$  int64
  IRExpr-down :: IRExpr  $\Rightarrow$  int64
  unfolding IRExpr-up-def IRExpr-down-def
  apply unfold-locales
  by (simp add: ucast-minus-one)+

phase NewAnd
  terminating size
begin

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

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

optimization redundant-rhs-y-or:  $(z \ \& \ (x \mid y)) \mapsto z \ \& \ x$ 
  when  $((and \ (IRExpr-up \ y) \ (IRExpr-up \ z)) = 0)$ 
  using simple-mask.exp-eliminate-y

```

```

    by (meson exp-and-commute order.trans)

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

end

end

theory NotPhase
  imports
    Common
begin

```

6 Optimizations for Not Nodes

```

phase NotNode
  terminating size
begin

```

```

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

```

```

lemma val-not-cancel:
  assumes val[¬(new-int b v)]  $\neq$ .UndefVal
  shows val[¬(¬(new-int b v))] = (new-int b v)
  using bin-not-cancel
  by (simp add: take-bit-not-take-bit)

```

```

lemma exp-not-cancel:
  shows exp[¬(¬a)]  $\geq$  exp[a]
  using unary-eval.simps(3) val-not-cancel
  apply (cases a; simp)
  apply (smt (verit, ccfv-SIG) unary-eval.simps(3) unary-eval-new-int un-
  fold-unary val-not-cancel)
  sorry

```

```

optimization NotCancel: exp[¬(¬a)]  $\mapsto$  a
  by (metis exp-not-cancel)

```

```

end

end
theory OrPhase
  imports
    Common
    NewAnd
begin

```

7 Optimizations for Or Nodes

```

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  $x = new\_int\ b\ v$ 
  assumes  $x \neq UndefinedVal \wedge ((intval\_or\ x\ x) \neq UndefinedVal)$ 
  shows  $val[x \mid x] = val[x]$ 
  apply (cases x; auto) using bin-or-equal assms
  by auto+

```

```

lemma val-elim-redundant-false:
  assumes  $x = new\_int\ b\ v$ 
  assumes  $x \neq UndefinedVal \wedge (intval\_or\ x\ (bool\_to\_val\ False)) \neq UndefinedVal$ 
  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[~x | ~y] = val[~(x & y)]
apply (cases x; cases y; auto)
by (simp add: take-bit-not-take-bit)

```

```

lemma exp-or-equal:
  exp[x | x] ≥ exp[x]
  apply simp using val-or-equal sorry

```

```

lemma exp-elim-redundant-false:
  exp[x | false] ≥ exp[x]
  apply simp using val-elim-redundant-false
  apply (cases x) sorry

```

```

optimization OrEqual: x | x ⟶ x
  by (meson exp-or-equal le-expr-def)

```

```

optimization OrShiftConstantRight: ((const x) | y) ⟶ y | (const x) when ¬(is-ConstantExpr y)
  unfolding le-expr-def using val-shift-const-right-helper size-non-const
  apply simp apply auto
  sorry

```

```

optimization EliminateRedundantFalse: x | false ⟶ x
  by (meson exp-elim-redundant-false le-expr-def)

```

```

optimization OrNotOperands: (~x | ~y) ⟶ ~ (x & y)
  apply auto using val-or-not-operands
  by (metis BinaryExpr UnaryExpr bin-eval.simps(4) intval-not.simps(2) unary-eval.simps(3))

```

```

optimization OrLeftFallthrough: (x | y) ⟶ x
  when (((and (not (IExpr-down x)) (IExpr-up y)) = 0))
  by (simp add: IExpr-down-def IExpr-up-def)

```

```

optimization OrRightFallthrough: (x | y) ⟶ y
  when (((and (not (IExpr-down y)) (IExpr-up x)) = 0))
  by (meson exp-or-commute OrLeftFallthrough(1) order.trans rewrite-preservation.simps(2))

```

```

end

```

```

end
theory ShiftPhase
  imports
    Common
  begin

```

```

phase ShiftNode
  terminating size
begin

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

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

lemma
  assumes in-bounds (intval-log2 val-c) 0 32
  shows intval-left-shift x (intval-log2 val-c) = intval-mul x val-c
  apply (cases val-c; auto) using intval-left-shift.simps(1) intval-mul.simps(1)
intval-log2.simps(1)
  sorry

lemma e-intval:
  n = intval-log2 val-c  $\wedge$  in-bounds n 0 32  $\longrightarrow$ 
    intval-left-shift x (intval-log2 val-c) =
      intval-mul x val-c
proof (rule impI)
  assume n = intval-log2 val-c  $\wedge$  in-bounds n 0 32
  show intval-left-shift x (intval-log2 val-c) =
    intval-mul x val-c
  proof (cases  $\exists v. \text{val-c} = \text{IntVal } 32\ v$ )
    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=2e))
      using  $\langle n = \text{intval-log2 } \text{val-c} \wedge \text{in-bounds } n\ 0\ 32 \rangle$  intval-log2.simps(1) by
presburger
      then show ?thesis sorry
    next
      case False
      then have  $\exists v. \text{val-c} = \text{IntVal } 64\ v$ 
      sorry
      then obtain vc where val-c = IntVal 64 vc
      by auto
      then have n = IntVal 64 (word-of-int (SOME e. vc=2e))
      using  $\langle n = \text{intval-log2 } \text{val-c} \wedge \text{in-bounds } n\ 0\ 32 \rangle$  intval-log2.simps(1) by
presburger
      then show ?thesis sorry
  qed
qed

```

```

optimization e:
   $x * (\text{const } c) \mapsto x << (\text{const } n) \text{ when } (n = \text{intval-log2 } c \wedge \text{in-bounds } n \ 0 \ 32)$ 
  using e-intval
  using BinaryExprE ConstantExprE bin-eval.simps(2,7) sorry

end

end
theory SignedDivPhase
  imports
    Common
begin

```

8 Optimizations for SignedDiv Nodes

```

phase SignedDivNode
  terminating size
begin

```

```

lemma val-division-by-one-is-self-32:
  assumes  $x = \text{new-int } 32 \ v$ 
  shows  $\text{intval-div } x \ (\text{IntVal } 32 \ 1) = x$ 
  using assms apply (cases x; auto)
  by (simp add: take-bit-signed-take-bit)

```

```

end

end
theory SignedRemPhase
  imports
    Common
begin

```

9 Optimizations for SignedRem Nodes

```

phase SignedRemNode
  terminating size
begin

```

```

lemma val-remainder-one:
  assumes  $\text{intval-mod } x \ (\text{IntVal32 } 1) \neq \text{UndefVal}$ 
  shows  $\text{intval-mod } x \ (\text{IntVal32 } 1) = \text{IntVal32 } 0$ 

```

```

    using assms apply (cases x; auto) sorry

value word-of-int (sint (x2::32 word) smod 1)

end

end
theory SubPhase
  imports
    Common
begin

```

10 Optimizations for Sub Nodes

```

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

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

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

lemma *val-subtract-zero*:
assumes $x = \text{new-int } b\ v$
assumes $\text{intval-sub } x\ (\text{IntVal } 32\ 0) \neq \text{UndefVal}$
shows $\text{intval-sub } x\ (\text{IntVal } 32\ 0) = \text{val}[x]$
using *assms apply (induction x; simp)*
by *presburger*

lemma *val-zero-subtract-value*:
assumes $x = \text{new-int } b\ v$
assumes $\text{intval-sub } (\text{IntVal } 32\ 0)\ x \neq \text{UndefVal}$
shows $\text{intval-sub } (\text{IntVal } 32\ 0)\ x = \text{val}[-x]$
using *assms apply (induction x; simp)*
by *presburger*

lemma *val-zero-subtract-value-64*:
assumes $x = \text{new-int } b\ v$
assumes $\text{intval-sub } (\text{IntVal } 64\ 0)\ x \neq \text{UndefVal}$
shows $\text{intval-sub } (\text{IntVal } 64\ 0)\ x = \text{val}[-x]$
using *assms apply (induction x; simp)*
by *presburger*

lemma *val-sub-then-left-add*:
assumes $\text{val}[x - (x + y)] \neq \text{UndefVal}$
shows $\text{val}[x - (x + y)] = \text{val}[-y]$

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

lemma *val-sub-negative-value*:
assumes $\text{val}[x - (-y)] \neq \text{UndefVal}$
shows $\text{val}[x - (-y)] = \text{val}[x + y]$
using *assms* **by** (*cases* *x*; *cases* *y*; *auto*)

lemma *val-sub-self-is-zero*:
assumes $x = \text{new-int } b \ v \wedge x - x \neq \text{UndefVal}$
shows $\text{val}[x - x] = \text{new-int } b \ 0$
using *assms* **by** (*cases* *x*; *auto*)

lemma *val-sub-negative-const*:
assumes $y = \text{new-int } b \ v \wedge \text{val}[x - (-y)] \neq \text{UndefVal}$
shows $\text{val}[x - (-y)] = \text{val}[x + y]$
using *assms* **by** (*cases* *x*; *cases* *y*; *auto*)

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

lemma *exp-sub-after-right-add2*:
shows $\text{exp}[(x + y) - x] \geq \text{exp}[y]$
using *exp-sub-after-right-add* **apply** *auto*
using *bin-eval.simps*(1) *bin-eval.simps*(3) *intval-add-sym* *unfold-binary*
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*)

lemma *exp-sub-negative-value*:
 $\text{exp}[x - (-y)] \geq \text{exp}[x + y]$
apply *simp* **using** *val-sub-negative-value*
by (*smt* (*verit*) *bin-eval.simps*(1) *bin-eval.simps*(3) *evaltree-not-undef* *minus-Value-def*
unary-eval.simps(2) *unfold-binary* *unfold-unary*)

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

lemma *exp-sub-then-left-sub*:
assumes $\text{wf-stamp } x \wedge \text{stamp-expr } x = \text{IntegerStamp } b \ lo \ hi$

```

shows  $\exp[x - (x - y)] \geq \exp[y]$ 
using val-sub-then-left-sub assms
proof -
  have 1:  $\exp[x - (x - y)] = \exp[x - x + y]$ 
  apply simp
  sorry
  have  $\exp[x - (x - y)] \geq \exp[(\text{const } (\text{new-int } b \ 0)) + y]$ 
  sorry
  have  $\exp[(\text{const } \text{IntVal } b \ 0) + y] \geq \exp[y]$ 
  sorry
  then show ?thesis
  using 1 by fastforce
qed

```

Optimisations

```

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

```

```

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

```

```

optimization SubAfterSubLeft:  $((x - y) - x) \mapsto -y$ 
  apply auto
  by (metis evalDet unary-eval.simps(2) unfold-unary val-sub-after-left-sub)

```

```

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

```

```

optimization SubThenAddRight:  $(y - (x + y)) \mapsto -x$ 
  apply auto
  apply (metis less-1-mult less-one linorder-neqE-nat mult.commute mult-1 numeral-1-eq-Suc-0
    one-eq-numeral-iff one-less-numeral-iff semiring-norm(77) size-pos zero-less-iff-neq-zero)
  by (metis evalDet intval-add-sym unary-eval.simps(2) unfold-unary val-sub-then-left-add)

```

```

optimization SubThenSubLeft:  $(x - (x - y)) \mapsto y$ 
  when (wf-stamp  $x \wedge \text{stamp-expr } x = \text{IntegerStamp } b \text{ lo hi}$ )
  using exp-sub-then-left-sub by blast

```

```

optimization SubtractZero:  $(x - (\text{const } \text{IntVal } b \ 0)) \mapsto x$ 
  when (wf-stamp  $x \wedge \text{stamp-expr } x = \text{IntegerStamp } b \text{ lo hi}$ )
  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)

optimization *SubNegativeValue*: $(x - (-y)) \mapsto x + y$
 defer using *exp-sub-negative-value* apply *simp*
 sorry

optimization *ZeroSubtractValue*: $((\text{const IntVal } b \ 0) - x) \mapsto (-x)$
 when $(\text{wf-stamp } x \wedge \text{stamp-expr } x = \text{IntegerStamp } b \ \text{lo} \ hi)$

apply *auto* unfolding *wf-stamp-def* defer
 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)

sorry

fun *forPrimitive* :: *Stamp* \Rightarrow *int64* \Rightarrow *IRExpr* **where**
forPrimitive (*IntegerStamp* *b lo hi*) *v* = *ConstantExpr* (if take-bit *b v* = *v* then
 (*IntVal* *b v*) else *UndefVal*) |
forPrimitive - = *ConstantExpr* *UndefVal*

lemma *unfold-forPrimitive*:

forPrimitive *s v* = *ConstantExpr* (if is-IntegerStamp *s* \wedge take-bit (stp-bits *s*) *v* =
v then (*IntVal* (stp-bits *s*) *v*) else *UndefVal*)
 by (cases *s*; auto)

lemma *forPrimitive-size[size-simps]*: size (*forPrimitive* *s v*) = 1
 by (cases *s*; auto)

lemma *forPrimitive-eval*:

assumes *s* = *IntegerStamp* *b lo hi*
 assumes take-bit *b v* = *v*
 shows $[m, p] \vdash \text{forPrimitive } s \ v \mapsto (\text{IntVal } b \ v)$
 unfolding *unfold-forPrimitive* using *assms* apply *auto*
 apply (rule evaltree.ConstantExpr)
 sorry

lemma *evalSubStamp*:

assumes $[m, p] \vdash \text{exp}[x - y] \mapsto v$
 assumes *wf-stamp* *exp*[*x - y*]
 shows $\exists b \ \text{lo hi. stamp-expr } \text{exp}[x - y] = \text{IntegerStamp } b \ \text{lo hi}$
proof –
 have valid-value *v* (stamp-expr *exp*[*x - y*])
 using *assms* unfolding *wf-stamp-def* by *auto*
 then have stamp-expr *exp*[*x - y*] \neq *IllegalStamp*
 by force


```

then show ?thesis
  unfolding stamp-expr.simps using stamp-binary.simps
  by (smt (z3) stamp-binary.elims unrestricted-stamp.simps(2))
qed

```

```

lemma evalSubArgsStamp:
  assumes  $[m, p] \vdash \text{exp}[x - y] \mapsto v$ 
  assumes  $\exists lo\ hi. \text{stamp-expr exp}[x - y] = \text{IntegerStamp } b\ lo\ hi$ 
  shows  $\exists lo\ hi. \text{stamp-expr exp}[x] = \text{IntegerStamp } b\ lo\ hi$ 
  using assms sorry

```

```

optimization SubSelfIsZero:  $(x - x) \mapsto \text{forPrimitive } (\text{stamp-expr exp}[x - x])\ 0$ 
when  $((\text{wf-stamp } x) \wedge (\text{wf-stamp exp}[x - x]))$ 
  apply simp apply (rule impI; (rule allI)+; rule impI)
  subgoal premises eval for m p v
  proof -
    obtain b where  $\exists lo\ hi. \text{stamp-expr exp}[x - x] = \text{IntegerStamp } b\ lo\ hi$ 
    using evalSubStamp eval
    by meson
  then show ?thesis sorry
qed
done

```

end

end

theory XorPhase

imports

Common

begin

11 Optimizations for Xor Nodes

phase XorNode

terminating size

begin

lemma bin-xor-self-is-false:

$\text{bin}[x \oplus x] = 0$

by simp

lemma bin-xor-commute:

$\text{bin}[x \oplus y] = \text{bin}[y \oplus x]$

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 $val[x \oplus x] \neq UndefinedVal$
shows $val\text{-}to\text{-}bool\ (val[x \oplus x]) = False$
using *assms* **by** (*cases* *x*; *auto*)

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

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

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

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

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

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

```

by (smt (z3) validDefIntConst)

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

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

optimization EliminateRedundantFalse:  $(x \oplus \text{false}) \mapsto x$ 
    using val-eliminate-redundant-false apply auto sorry

optimization MaskOutRHS:  $(x \oplus \text{const } y) \mapsto \text{UnaryExpr UnaryNot } x$ 
    when  $((\text{stamp-expr } (x) = \text{IntegerStamp bits } l \ h))$ 

    unfolding le-expr-def apply auto
    sorry

end

end

theory NegatePhase
    imports
        Common
begin

```

12 Optimizations for Negate Nodes

```

phase NegateNode
    terminating size
begin

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

value (2 :: 32 word) >>> (31 :: nat)
value  $\neg((2 :: 32 \text{ word}) >> (31 :: \text{nat}))$ 

lemma bin-negative-shift32:
    shows  $\neg((x :: 32 \text{ word}) >> (31 :: \text{nat})) = x >>> (31 :: \text{nat})$ 
    sorry

```

```

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

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

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

lemma exp-negative-cancel:
  shows  $\text{exp}[-(-x)] \geq \text{exp}[x]$ 
  using val-negative-cancel apply (cases x; simp)
  using unary-eval-new-int apply force
  sorry

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

optimization DistributeSubtraction:  $-(x - y) \mapsto (y - x)$ 
  apply simp-all
  apply auto
  by (simp add: BinaryExpr evaltree-not-undef val-distribute-sub)

optimization NegativeShift:  $-(x >> (\text{const } (\text{IntVal } b\ y))) \mapsto x >>> (\text{const } (\text{IntVal } b\ y))$ 
   $\text{when } (\text{stamp-expr } x = \text{IntegerStamp } b' \text{ lo hi} \wedge \text{unat } y = (b' - 1))$ 
  apply simp-all apply auto
  sorry

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
```

12.1 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  $\approx$  rhs], defer-tac, explore-cases x y)
```

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

```
method solve for lhs rhs x y :: Value =
  (match conclusion in size - < size -  $\Rightarrow$   $\langle simp \rangle$ )?,
```

(*match conclusion in* (*elhs::IRExpr*) \geq (*erhs::IRExpr*) **for** *elhs erhs* \Rightarrow \langle
 (*obtain-approx-eq lhs rhs x y*)?)

print-methods

thm *BinaryExprE*

optimization *opt-add-left-negate-to-sub*:

$-x + y \mapsto y - x$

apply (*solve val*[-*x1* + *y1*] *val*[*y1* - *x1*] *x1 y1*)

apply simp apply auto using evaltree-not-undef sorry

12.2 NegateNode

lemma *val-distribute-sub*:

$\text{val}[-(x-y)] \approx \text{val}[y-x]$

by (*cases x; cases y; auto*)

optimization *distribute-sub*: $-(x-y) \mapsto (y-x)$

apply simp

using *val-distribute-sub* **apply simp**

using *unfold-binary unfold-unary* **by auto**

lemma *val-xor-self-is-false*:

assumes $x = \text{IntVal } 32 \ v$

shows $\text{val}[x \oplus x] \approx \text{val}[\text{false}]$

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

definition *wf-stamp* :: *IRExpr* \Rightarrow *bool* **where**

$\text{wf-stamp } e = (\forall m \ p \ v. ([m, p] \vdash e \mapsto v) \longrightarrow \text{valid-value } v \ (\text{stamp-expr } e))$

lemma *exp-xor-self-is-false*:

assumes $\text{stamp-expr } x = \text{IntegerStamp } 32 \ l \ h$

assumes *wf-stamp* x

shows $\text{exp}[x \oplus x] \geq \text{exp}[\text{false}]$

unfolding *le-expr-def* **using** *assms* **unfolding** *wf-stamp-def*

using *val-xor-self-is-false evaltree-not-undef*

by (*smt (z3) 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 unfold-const
 valid-int valid-stamp.simps(1) valid-value.simps(1) well-formed-equal-defn*)

lemma *val-or-commute[simp]*:

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

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

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

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

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

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

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

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

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

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

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

```

val-xor.elims
  mask-eq-take-bit-minus-one new-int.elims new-int-take-bits unfold-const valid-stamp.simps(1)

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

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

end

end

theory ProofStatus
  imports
    AbsPhase
    AddPhase
    AndPhase
    ConditionalPhase
    MulPhase

    NegatePhase
    NewAnd
    NotPhase
    OrPhase
    ShiftPhase
    SignedDivPhase
    SignedRemPhase
    SubPhase
    TacticSolving
    XorPhase
  begin

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

  print-methods

  print-theorems

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

  export-phases ⟨Full⟩

end

```