

Veriopt

September 11, 2022

Abstract

The Veriopt project aims to prove the optimization pass of the GraalVM compiler. The GraalVM compiler includes a sophisticated Intermediate Representation (IR) in the form of a sea-of-nodes based graph structure. We first define the IR graph structure in the Isabelle/HOL interactive theorem prover. We subsequently give the evaluation of the structure a semantics based on the current understanding of the purpose of each IR graph node. Optimization phases are then encoded including the static analysis passes required for an optimization. Each optimization phase is proved to be correct by proving that a bisimulation exists between the unoptimized and optimized graphs. The following document has been automatically generated from the Isabelle/HOL source to provide a very comprehensive definition of the semantics and optimizations introduced by the Veriopt project.

Contents

1	Optimizations for Abs Nodes	3
2	Optimizations for Add Nodes	8
3	Optimizations for And Nodes	11
3.1	Conditional Expression	14
4	Optimizations for Mul Nodes	17
5	Optimizations for Negate Nodes	23
6	Optimizations for Not Nodes	24
7	Optimizations for Or Nodes	25
8	Optimizations for SignedDiv Nodes	27
9	Optimizations for Sub Nodes	27
10	Optimizations for Xor Nodes	31

```

theory AbsPhase
  imports
    Common
begin

```

1 Optimizations for Abs Nodes

```

phase AbsNode
  terminating size
begin

```

```

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

```

```

lemma abs-neg:
  fixes v :: ('a :: len word)
  assumes v <s 0
  assumes -(2 ^ (Nat.size v - 1)) <s v
  shows (if v <s 0 then - v else v) = - v ∧ 0 <s -v
  by (smt (verit, ccfv-SIG) assms(1) assms(2) signed-take-bit-int-greater-eq-minus-exp
    signed-take-bit-int-greater-eq-self-iff sint-0 sint-word-ariths(4) word-sless-alt)

```

```

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

```

```

lemma final-abs:
  fixes v :: ('a :: len word)
  assumes take-bit (Nat.size v) v = v
  assumes - (2 ^ (Nat.size v - 1)) ≠ v
  shows 0 ≤s (if v <s 0 then -v else v)

```

```

proof (cases v <s 0)
  case True
  then show ?thesis
  proof (cases v = - (2 ^ (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 < s 0) then (- v) else v)

```

```

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

```

```

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

```

```

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

```

```

lemma val-abs-neg:

```

```

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

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

lemma take-bit-unwrap:
  b = 64  $\implies$  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 word
  assumes b  $\leq$  64
  shows sint (signed-take-bit (b - Suc (0::nat)) (take-bit b v1))
    < sint (signed-take-bit (b - Suc (0::nat)) (take-bit b v2))  $\longleftrightarrow$ 
    signed-take-bit (63::nat) (Word.rep v1) < signed-take-bit (63::nat) (Word.rep
v2)
  using assms sorry

lemma less-eq-def:
  shows val-to-bool(val[(new-int b v1) < (new-int b v2)])  $\longleftrightarrow$  v1 < s v2
  unfolding new-int.simps intval-less-than.simps bool-to-val-bin.simps bool-to-val.simps
int-signed-value.simps apply (simp add: val-bool-unwrap)
  apply auto 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 intval-abs (new-int b v) = (new-int b v')
  shows 0  $\leq_s$  v'
  using assms
proof (cases v = 0)
  case True
  then have v' = 0
  using val-abs-zero assms
  by (smt (verit, ccfv-threshold) Suc-diff-1 bit-less-eq-def bot-nat-0.extremum
diff-is-0-eq len-gt-0 len-of-numeral-defs(2) order-le-less signed-eq-0-iff take-bit-0 take-bit-signed-take-bit
take-bit-unwrap)
  then show ?thesis by simp
next
  case neq0: False
  then show ?thesis
  proof (cases val-to-bool(val[(new-int b 0) < (new-int b v)]))
  case True

```

```

    then show ?thesis using less-eq-def
    using assms val-abs-pos
    by (smt (verit, ccfv-SIG) One-nat-def Suc-leI bit.compl-one bit-less-eq-def
cancel-comm-monoid-add-class.diff-cancel diff-zero len-gt-0 len-of-numeral-defs(2)
mask-0 mask-1 one-le-numeral one-neq-zero signed-word-eqI take-bit-dist-subL take-bit-minus-one-eq-mask
take-bit-not-eq-mask-diff take-bit-signed-take-bit zero-le-numeral)
  next
  case False
  then have val-to-bool(val[(new-int b v) < (new-int b 0)])
    using neq0 less-eq-def
    by (metis signed.neqE)
  then show ?thesis using val-abs-neg less-eq-def unfolding new-int.simps
intval-negate.simps
    by (metis signed.nless-le take-bit-0)
  qed

```

qed

lemma *intval-abs-elim:*

```

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

```

lemma *wf-abs-new-int:*

```

  assumes intval-abs (IntVal t v) ≠ 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 (intval-abs x) ≠ UndefVal
  shows  $\text{intval-abs } x \neq \text{UndefVal}$ 
  using assms
  by force

```

lemma *val-abs-idem:*

```

  assumes intval-abs(intval-abs(x)) ≠ UndefVal
  shows  $\text{intval-abs}(\text{intval-abs}(x)) = \text{intval-abs } x$ 
  using assms

```

proof –

```

  obtain b v where in-def:  $\text{intval-abs } x = \text{new-int } b v$ 
  using assms intval-abs-elim mono-undef-abs by blast
  then show ?thesis
  proof (cases val-to-bool(val[(new-int b v) < (new-int b 0)]))

```

```

case True
then have nested: (intval-abs (intval-abs x)) = new-int b (-v)
  using val-abs-neg intval-negate.simps in-def
  by simp
then have x = 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 ≠ UndefVal ∧ intval-negate x ≠ UndefVal ∧ intval-abs(intval-negate
x) ≠ UndefVal
  shows intval-abs (intval-negate x) = intval-abs x
  using assms apply (cases x; auto)
  apply (metis less-eq-def new-int.simps signed.dual-order.strict-iff-not signed.less-linear

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

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

  val-abs-always-pos)

Optimisations

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

optimization AbsNegate: (abs(-x))  $\mapsto$  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

```

2 Optimizations for Add Nodes

```

phase AddNode
  terminating size
begin

```

```

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

```

```

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

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

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

optimization *RedundantSubAdd*: ((*e1* - *e2*) + *e2*) \mapsto *e1*
apply *auto* **using** *eval-unused-bits-zero* *NeutralLeftSubVal*
unfolding *well-formed-equal-defn*
by (*smt* (*verit*) *evalDet* *intval-sub.elims* *new-int.elims*)

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

lemma *just-goal2*:
assumes *1*: ($\forall a b. (intval-add (intval-sub a b) b \neq UndefVal \wedge a \neq UndefVal$
 \longrightarrow
intval-add (*intval-sub* *a* *b*) *b* = *a*))
shows (*BinaryExpr* *BinAdd* (*BinaryExpr* *BinSub* *e1* *e2*) *e2*) \geq *e1*
unfolding *le-expr-def* *unfold-binary* *bin-eval.simps*
by (*metis* 1 *evalDet* *evaltree-not-undef*)

optimization *RedundantSubAdd2*: *e2* + (*e1* - *e2*) \mapsto *e1*
by (*smt* (*verit*, *del-insts*) *BinaryExpr* *BinaryExprE* *RedundantSubAdd*(1) *bin-add-commute* *le-expr-def* *rewrite-preservation.simps*(1))

lemma *AddToSubHelperLowLevel*:
shows *intval-add* (*intval-negate* *e*) *y* = *intval-sub* *y* *e* (**is** ?*x* = ?*y*)

```

    by (induction y; induction e; auto)

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

```

3 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:
   $(x \ \& \ \sim \text{False}) = x$ 
  by simp

```

```

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

```

```

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

```

```

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

```

```

lemma val-and-sign-extend:
  assumes  $e = (1 << \text{In}) - 1$ 
  shows  $\text{val}[(\text{intval-sign-extend } \text{In } \text{Out } x) \ \& \ (\text{IntVal } 32 \ e)] = \text{intval-zero-extend } \text{In } \text{Out } x$ 
  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:
   $\text{exp}[x \ \& \ x] \geq \text{exp}[x]$ 
  apply auto using val-and-equal eval-unused-bits-zero
  by (smt (verit) evalDet intval-and.elims new-int.elims)

```

```

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

```

```

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

```

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

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

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

apply *simp-all*
apply *auto*
sorry

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

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

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

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

proof –

obtain *xv* **where** *xv*: $[m, p] \vdash x \mapsto xv$

using *p*(2) **by** *blast*

obtain *yv* **where** *yv*: $[m, p] \vdash y \mapsto yv$

using *p*(2) **by** *blast*

have $v = \text{val}[xv \& yv]$

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

```

exp[x]
apply simp apply (rule impI; (rule allI)+)
apply (rule impI)
subgoal premises p for m p v
proof –
  obtain xv where xv: [m, p] ⊢ 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

3.1 Conditional Expression

theory ConditionalPhase

imports

Common

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

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

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

definition *wff-stamps* :: *bool* **where**
wff-stamps = $(\forall m \ p \ \text{expr} \ \text{val} . ([m, p] \vdash \text{expr} \mapsto \text{val}) \longrightarrow \text{valid-value} \ \text{val} \ (\text{stamp-expr} \ \text{expr}))$

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-optimise-integer-test*:
assumes *is-IntVal32 x*
shows *intval-conditional (intval-equals val[(x & (IntVal32 1))] (IntVal32 0))*
(IntVal32 0) (IntVal32 1) =
val[x & IntVal32 1]
apply simp-all
apply auto
using *bool-to-val.elims intval-equals.elims val-to-bool.simps(1) val-to-bool.simps(3)*
sorry

optimization *ConditionalEliminateKnownLess*: $((x < y) \ ? \ x : y) \mapsto x$
when (stamp-under (stamp-expr x) (stamp-expr y)
 \wedge wf-stamp x \wedge wf-stamp y)
apply auto
using *stamp-under.simps wf-stamp-def val-to-bool.simps*
sorry

optimization *ConditionalEqualIsRHS*: $((x \text{ eq } y) \ ? \ x : y) \mapsto y$
apply simp-all apply auto using *Canonicalization.intval.simps(1) evalDet*
intval-conditional.simps evaltree-not-undef
by (*metis* (*no-types*, *opaque-lifting*) *Value.discI(2) Value.distinct(1) intval-and.simps(3)*
intval-equals.simps(2) val-optimise-integer-test val-to-bool.simps(2))

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

optimization *OptimiseIntegerTest*:
 $((((x \ \& \ (\text{const } (\text{IntVal } 32 \ 1))) \text{ eq } (\text{const } (\text{IntVal } 32 \ 0))) \ ?$
 $(\text{const } (\text{IntVal } 32 \ 0)) : (\text{const } (\text{IntVal } 32 \ 1))) \mapsto$
 $x \ \& \ (\text{const } (\text{IntVal } 32 \ 1))$
 $\text{when } (\text{stamp-expr } x = \text{default-stamp})$
apply *simp-all*
apply *auto*
using *val-optimise-integer-test* **sorry**

optimization *opt-optimise-integer-test-2*:
 $((((x \ \& \ (\text{const } (\text{IntVal } 32 \ 1))) \text{ eq } (\text{const } (\text{IntVal } 32 \ 0))) \ ?$
 $(\text{const } (\text{IntVal } 32 \ 0)) : (\text{const } (\text{IntVal } 32 \ 1))) \mapsto$
 x
 $\text{when } (x = \text{ConstantExpr } (\text{IntVal } 32 \ 0) \mid (x = \text{ConstantExpr } (\text{IntVal}$
 $32 \ 1)))$
done


```

optimization opt-conditional-eliminate-known-less:  $((x < y) \text{ ? } x : y) \mapsto x$ 
               when  $((\text{stamp-under } (\text{stamp-expr } x) (\text{stamp-expr } y)) \mid$ 
                $((\text{stpi-upper } (\text{stamp-expr } x)) = (\text{stpi-lower } (\text{stamp-expr } y))))$ 
                $\wedge \text{wf-stamp } x \wedge \text{wf-stamp } y$ )
  unfolding le-expr-def apply auto
using stamp-under.simps wf-stamp-def
sorry

end

end
theory MulPhase
  imports
    Common
begin

```

4 Optimizations for Mul Nodes

```

phase MulNode
  terminating size
begin

lemma bin-eliminate-redundant-negative:
   $\text{uminus } (x :: 'a::\text{len word}) * \text{uminus } (y :: 'a::\text{len word}) = x * y$ 
  by simp

lemma bin-multiply-identity:
   $(x :: 'a::\text{len word}) * 1 = x$ 
  by simp

lemma bin-multiply-eliminate:
   $(x :: 'a::\text{len word}) * 0 = 0$ 
  by simp

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

lemma bin-multiply-power-2:
   $(x :: 'a::\text{len word}) * (2^j) = x << j$ 
  by simp

```

```

lemma take-bit64[simp]:
  fixes  $w :: \text{int64}$ 
  shows take-bit 64 w = w
proof -
  have  $\text{Nat.size } w = 64$ 
  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  $a :: \text{nat}$ 
  fixes  $b\ c :: 64\ \text{word}$ 
  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  $\text{val}[-x * -y] \neq \text{UndefVal}$ 
  shows  $\text{val}[-x * -y] = \text{val}[x * y]$ 
  using assms apply (cases x; cases y; auto)
  using testt by auto

lemma val-multiply-neutral:
  assumes  $x = \text{new-int } b\ v$ 
  shows  $\text{val}[x] * (\text{IntVal } b\ 1) = \text{val}[x]$ 
  using assms times-Value-def by force

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

lemma val-multiply-negative:
  assumes  $x = \text{new-int } b\ v$ 
  shows  $x * \text{intval-negate } (\text{IntVal } b\ 1) = \text{intval-negate } x$ 
  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  $i :: 64 \text{ word}$ 
  assumes  $y = \text{IntVal } 64 \ (2 \wedge \text{unat}(i))$ 
  and  $0 < i$ 
  and  $i < 64$ 
  and  $(63 :: \text{int64}) = \text{mask } 6$ 
  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  $x * y = \text{val}[x << \text{IntVal } 64 \ i]$ 
  using assms apply (cases  $x$ ; cases  $y$ ; auto)
  apply (simp add: times-Value-def)
  subgoal premises  $p$  for  $x2$ 
  proof –
    have  $63: (63 :: \text{int64}) = \text{mask } 6$ 
    using assms(4) by blast
    then have  $(2::\text{int}) \wedge 6 = 64$ 
    by eval
    then have  $\text{uint } i < (2::\text{int}) \wedge 6$ 
    by (smt (verit, ccfv-SIG) numeral-Bit0 of-int-numeral one-eq-numeral-iff)
   $p(6)$  uint-2p
    word-less-def word-not-simps(1) word-of-int-2p)
    then have  $\text{and } i (\text{mask } 6) = i$ 
    using mask-eq-iff by blast
    then show  $x2 << \text{unat } i = x2 << \text{unat } (\text{and } i (63::64 \text{ word}))$ 
    unfolding 63
    by force
  qed
done

```

```

lemma val-MulPower2Add1:
  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  $x * y = \text{val}[(x << \text{IntVal } 64 \ i) + x]$ 
  using assms apply (cases  $x$ ; cases  $y$ ; auto)
  apply (simp add: times-Value-def)
  subgoal premises  $p$  for  $x2$ 
  proof –
    have  $63: (63 :: \text{int64}) = \text{mask } 6$ 
    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: distrib-left)
  then show  $x2 * ((2::64 \text{ word}) \wedge \text{unat } i + (1::64 \text{ word})) = x2 << \text{unat } (and\ i\ (63::64 \text{ word})) + x2$ 
    by (simp add: 63 and (i::64 word) (mask (6::nat)) = i)
  qed
done

```

lemma *val-MulPower2Sub1*:

```

  fixes i :: 64 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  $x * y = \text{val}[(x << \text{IntVal } 64\ i) - x]$ 
  using assms apply (cases x; cases y; auto)
  apply (simp add: times-Value-def)
  subgoal premises p for x2
  proof -
    have  $63 :: \text{int64} = \text{mask } 6$ 
    by eval
    then have  $(2::\text{int}) \wedge 6 = 64$ 
    by eval
    then have  $and\ i\ (\text{mask } 6) = i$ 
    using mask-eq-iff by (simp add: less-mask-eq p(6))
    then have  $x2 * ((2::64 \text{ word}) \wedge \text{unat } i - (1::64 \text{ word})) = (x2 * ((2::64 \text{ word}) \wedge \text{unat } i)) - x2$ 
    by (simp add: right-diff-distrib')
    then show  $x2 * ((2::64 \text{ word}) \wedge \text{unat } i - (1::64 \text{ word})) = x2 << \text{unat } (and\ i\ (63::64 \text{ word})) - x2$ 
    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 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  $x * y = \text{val}[(x << \text{IntVal } 64 \ i) + (x << \text{IntVal } 64 \ j)]$ 
using assms
proof –
  have  $63 :: \text{int64} = \text{mask } 6$ 
  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  $\text{val}[(x * \text{IntVal } 64 \ (2 \wedge \text{unat}(i)))] = \text{val}[x << \text{IntVal } 64 \ i]$ 
  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  $i < 64$ 
  shows  $\text{exp}[x * y] \geq \text{exp}[x << \text{ConstantExpr } (\text{IntVal } 64 \ i)]$ 
  using assms val-MulPower2

```

```

sorry

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)

optimization MulPower2:  $x * y \mapsto x << \text{const } (\text{IntVal } 64 \ i)$ 
  when ( $i > 0 \wedge 64 > i \wedge$ 
     $y = (\text{ConstantExpr } (\text{IntVal } 64 \ (2 \wedge \text{unat}(i))))$ )
  defer
  using exp-MulPower2
  apply blast
  sorry

end

end
theory NegatePhase
  imports
    Common
  begin

```

5 Optimizations for Negate Nodes

```
phase NegateNode
  terminating size
begin
```

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

```
value (2 :: 32 word) >>> (31 :: nat)
value -((2 :: 32 word) >> (31 :: nat))
```

```
lemma bin-negative-shift32:
  shows  $-((x :: 32 word) >> (31 :: nat)) = x >>> (31 :: nat)$ 
  sorry
```

```
lemma val-negative-cancel:
  assumes  $intval\_negate\ (new\_int\ b\ v) \neq UndefinedVal$ 
  shows  $val[-(-(new\_int\ b\ v))] = val[new\_int\ b\ v]$ 
  using assms by simp
```

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

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

```
lemma exp-negative-cancel:
  shows  $exp[-(-x)] \geq exp[x]$ 
  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:  $-(x >> (\text{const } (\text{IntVal } b \ y))) \mapsto$ 
 $x >>> (\text{const } (\text{IntVal } b \ y))$ 
 $\text{when } (\text{stamp-expr } x = \text{IntegerStamp } b' \ \text{lo } \text{hi} \wedge \text{unat } y$ 
 $= (b' - 1))$ 
apply simp-all apply auto
sorry

end

end
theory NotPhase
imports
Common
begin

```

6 Optimizations for Not Nodes

```

phase NotNode
terminating size
begin

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

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

lemma exp-not-cancel:
shows  $\text{exp}[\sim(\sim a)] \geq \text{exp}[a]$ 
apply simp using val-not-cancel sorry

optimization NotCancel:  $\text{exp}[\sim(\sim a)] \mapsto a$ 
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:
   $val[\sim x \mid \sim y] = val[\sim(x \& y)]$ 

```

```

apply (cases x; cases y; auto)
by (simp add: take-bit-not-take-bit)

lemma exp-or-equal:
  exp[x | 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  $\mapsto$  x
  by (meson exp-or-equal le-expr-def)

optimization OrShiftConstantRight: ((const x) | y)  $\mapsto$  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  $\mapsto$  x
  by (meson exp-elim-redundant-false le-expr-def)

optimization OrNotOperands: (¬x | ¬y)  $\mapsto$  ¬ (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)  $\mapsto$  x
  when (((and (not (IExpr-down x)) (IExpr-up y)) = 0))
  by (simp add: IExpr-down-def IExpr-up-def)

optimization OrRightFallthrough: (x | y)  $\mapsto$  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 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 SubPhase
  imports
    Common
begin
```

9 Optimizations for Sub Nodes

```
phase SubNode
  terminating size
begin
```

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

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

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

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

```
lemma bin-subtract-zero:
```

```

shows (x :: 'a::len word) - (0 :: 'a::len word) = x
by simp

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

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

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

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

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

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

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

lemma val-zero-subtract-value:
  assumes x = new-int b v
  assumes intval-sub (IntVal 32 0) x ≠ UndefVal
  shows intval-sub (IntVal 32 0) x = 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)

lemma exp-sub-then-left-sub:
  exp[x - (x - y)] ≥ exp[y]
proof -
  have exp[x - (-y)] ≥ exp[x + y]
    using exp-sub-negative-value by simp
  then have exp[x - (x - y)] ≥ exp[x - x + y]
    using exp-sub-negative-value sorry
  then show ?thesis
    sorry
qed

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

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

optimization SubAfterSubLeft: ((x - y) - x) ↦ -y
  apply auto
  apply (metis One-nat-def less-add-one less-numeral-extra(3) less-one linorder-neqE-nat
    pos-add-strict size-pos)
  by (metis evalDet unary-eval.simps(2) unfold-unary val-sub-after-left-sub)

optimization SubThenAddLeft: (x - (x + y)) ↦ -y
  apply auto
  apply (simp add: Suc-lessI one-is-add)
  by (metis evalDet unary-eval.simps(2) unfold-unary
    val-sub-then-left-add)

optimization SubThenAddRight: (y - (x + y)) ↦ -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)) ↦ y
  using val-sub-then-left-sub sledgehammer sorry

```

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

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

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

apply *simp-all*
apply *auto*
apply (*meson* *less-add-same-cancel1* *less-trans-Suc* *size-pos*)
by (*smt* (*verit*) *Value.inject*(1) *eq-iff-diff-eq-0* *evalDet* *intval-sub.elims* *new-int.elims*
new-int-bin.elims *take-bit-of-0* *unfold-const* *validDefIntConst* *valid-stamp.simps*(1)
valid-value.simps(1) *wf-stamp-def*)

end

end

theory *XorPhase*

imports
Common

begin

10 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*:

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

lemma *bin-eliminate-redundant-false*:

$bin[x \oplus 0] = bin[x]$
by *simp*

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

assumes $val[x \oplus x] \neq UndefinedVal$
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-stamp  $x \wedge$  stamp-expr  $x =$  default-stamp
shows  $\text{exp}[x \oplus x] \geq \text{exp}[\text{false}]$ 
  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 ∧ stamp-expr x = default-stamp)
  apply auto[1]
  apply (simp add: Suc-lessI one-is-add) 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

```