Veriopt

August 25, 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

```
theory AbsPhase
  imports
    Common

begin
```

1 Optimizations for Abs Nodes

```
phase AbsPhase
  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 ^ (Word.size-word-inst.size-word 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 ^ (Word.size-word-inst.size-word v − 1)) = v
  shows −v = v
  sorry



lemma final-abs:
  fixes v :: ('a :: len word)
  assumes − (2 ^ (Word.size-word-inst.size-word 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 ^ (Word.size-word-inst.size-word v − 1)))
    case True
```

then show *?thesis* **using** *abs-max-neg*
using *assms* **by** *presburger*
next
case *False*
then have $-\ (2\ \hat{}\ (\mathit{Word.size\text{-}word\text{-}inst.size\text{-}word}\ v\ -\ 1)) <s\ v$
sorry

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*: $(\mathit{is\text{-}IntVal32}\ x \lor \mathit{is\text{-}IntVal64}\ x) \Longrightarrow \mathit{intval\text{-}abs}\ x \neq \mathit{UndefVal}$
by (*metis Value.disc*(*1*) *Value.disc*(*6*) *intval-abs.simps*(*1*) *intval-abs.simps*(*2*) *is-IntVal32-def*
is-IntVal64-def)

fun *bin-abs* :: *'a* ::*len word* ⇒ *'a* ::*len word* **where**
$\mathit{bin\text{-}abs}\ v = (\mathit{if}\ (v <s\ 0)\ \mathit{then}\ (-\ v)\ \mathit{else}\ v)$

lemma *val-abs-zero-32*:
$\mathit{intval\text{-}abs}\ (\mathit{IntVal32}\ 0) = \mathit{IntVal32}\ 0$
by *simp*

lemma *val-abs-pos-32*:
assumes $\mathit{is\text{-}IntVal32}\ x \land \mathit{val\text{-}to\text{-}bool}(\mathit{val}[(\mathit{IntVal32}\ 0) < x])$
shows $\mathit{intval\text{-}abs}\ x = x$
using *assms* **apply** (*cases x*; *auto*)
by (*metis bool-to-val.elims signed.less-asym val-to-bool.simps*(*1*))

lemma *val-abs-neg-32*:
assumes $\mathit{is\text{-}IntVal32}\ x \land \mathit{val\text{-}to\text{-}bool}(\mathit{val}[x < (\mathit{IntVal32}\ 0)])$
shows $\mathit{intval\text{-}abs}\ x = \mathit{intval\text{-}negate}\ x$
using *assms*
by (*cases x*; *auto*)

lemma *abs-zero-64*:
$\mathit{intval\text{-}abs}\ (\mathit{IntVal64}\ 0) = \mathit{IntVal64}\ 0$
by *simp*

lemma *val-abs-pos-64*:
assumes *is-IntVal64 x* ∧ *val-to-bool*(*val*[(*IntVal64 0*) < *x*])
shows *intval-abs x* = *x*
using *assms* **apply** (*cases x*; *auto*)
by (*metis bool-to-val.elims signed.less-asym val-to-bool.simps*(*1*))

lemma *val-abs-neg-64*:
assumes *is-IntVal64 x* ∧ *val-to-bool*(*val*[*x* < (*IntVal64 0*)])
shows *intval-abs x* = *intval-negate x*
using *assms*
by (*cases x*; *auto*)

lemma *val-abs-idem*:
assumes *x* ≠ *UndefVal* ∧ *intval-abs x* ≠ *UndefVal* ∧ *intval-abs*(*intval-abs*(*x*)) ≠ *UndefVal*
shows *intval-abs*(*intval-abs*(*x*)) = *intval-abs x*
using *assms* **apply** (*cases x*; *auto*)
using *final-abs* **using** *abs-max-neg*
by *fastforce*+

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*)
using *final-abs abs-max-neg* **apply** *fastforce* **defer**
using *final-abs abs-max-neg* **apply** *fastforce*
using *final-abs abs-max-neg abs-pos abs-neg val-abs-neg-64 val-abs-neg-32 wf-abs*
sorry

optimization *abs-idempotence*: *abs*(*abs*(*x*)) ⟼ *abs*(*x*)
apply *auto*
by (*metis UnaryExpr intval-abs.simps*(*3*) *unary-eval.simps*(*1*) *val-abs-idem*)

optimization *abs-negate*: (*abs*(−*x*)) ⟼ *abs*(*x*)
apply *auto*
by (*metis UnaryExpr intval-negate.simps*(*3*) *unary-eval.simps*(*1*) *val-abs-negate*)

end

end
theory *AddPhase*

```
  imports
    Common
begin
```

2 Optimizations for Add Nodes

```
phase SnipPhase
  terminating size
begin

optimization BinaryFoldConstant: BinaryExpr op (const v1) (const v2) ⟼ 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: is-IntVal v using bin-eval-int prems by auto
      show ?thesis
        unfolding prems x y xy
        apply (rule ConstantExpr)
        apply (rule validIntConst)
        using prems x y xy int by auto+
      qed
    done
  done

print-facts

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: ((const v) + y) ⟼ y + (const v) when ¬(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: ((const v) + y) ⟼ y + (const v) when
¬(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 x (IntVal32 0) ≠ UndefVal
  shows intval-add x (IntVal32 0) = x
  using 1 by (induction x; simp)
```

```
optimization AddNeutral: (e + (const (IntVal32 0))) ⟼ e
  unfolding le-expr-def apply auto
  unfolding is-neutral-0 apply auto
  done
```

```
ML-val ‹@{term ‹x = y›}›
```

```
optimization NeutralLeftSub[intval]: ((e1 - e2) + e2) ⟼ e1
   prefer 3 unfolding intval.simps

   using intval-add.simps intval-sub.simps
    apply (metis (no-types, lifting) diff-add-cancel val-to-bool.cases)
  unfolding le-expr-def unfold-binary unfold-const unfold-valid32 bin-eval.simps
```

```
  subgoal premises p1
    apply ((rule allI)+; rule impI)
    subgoal premises p2 for m p v
      print-facts
    proof −
      obtain x y xa where xa: [m,p] ⊢ e₁ ↦ xa and xa ≠ UndefVal
        and y: [m,p] ⊢ e₂ ↦ y      and y ≠ UndefVal
        and x: x = intval-sub xa y and x ≠ UndefVal
        and v: v = intval-add x y  and v ≠ UndefVal
        by (metis evalDet p2 evaltree-not-undef)
      then have v = intval-add (intval-sub xa y) y by auto
      then have v = xa
        print-facts
        using p1 p2 apply simp
      by (smt (z3) Value.distinct(9) Value.inject(1) Value.inject(2) ‹v ≠ UndefVal›
x ‹x ≠ UndefVal› diff-add-cancel intval-add.elims intval-sub.elims)
      then show [m,p] ⊢ e₁ ↦ v
        by (simp add: xa)
      thm intval-add.elims
    qed
    done
  using size-non-const by fastforce


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

lemma just-goal2:
  assumes 1: (∀ a b. (intval-add (intval-sub a b) b ≠ UndefVal ∧ a ≠ UndefVal
⟶
    intval-add (intval-sub a b) b = a))
  shows (BinaryExpr BinAdd (BinaryExpr BinSub e₁ e₂) e₂) ≥ e₁


  unfolding le-expr-def unfold-binary bin-eval.simps
  by (metis 1 evalDet evaltree-not-undef)


optimization NeutralRightSub[intval]: e₂ + (e₁ − e₂) ⟼ e₁
  using NeutralLeftSub(1) intval-add-sym apply auto[1]
  oops


lemma NeutralRightSub-1: intval-add a (intval-sub b a) ≠ UndefVal ∧
    b ≠ UndefVal ⟶
    intval-add a (intval-sub b a) = b (is ?U1 ∧ ?U2 ⟶ ?C)
proof
```

assume *?U1* ∧ *?U2*

then have *i*: (*is-IntVal32 a* ∧ *is-IntVal32 b*) ∨ (*is-IntVal64 a* ∧ *is-IntVal64 b*)
(**is** *?I32* ∨ *?I64*)
by (*metis Value.exhaust-disc intval-add.simps*(*10*) *intval-add.simps*(*15*) *intval-add.simps*(*16*) *intval-add-sym intval-sub.simps*(*12*) *intval-sub.simps*(*5*) *intval-sub.simps*(*8*) *intval-sub.simps*(*9*) *is-IntVal32-def is-IntVal64-def is-ObjRef-def is-ObjStr-def*)
then show *?C*
proof (*rule disjE*)
assume *i32*: *?I32*
show *?C* **using** *i32 add.commute is-IntVal32-def* **by** *auto*
next
assume *i64*: *?I64*
show *?C* **using** *i64 add.commute is-IntVal64-def* **by** *auto*
qed
qed

lemma *NeutralRightSub-2*:

((*intval-add a* (*intval-sub b a*) ≠ *UndefVal* ∧ *b* ≠ *UndefVal*
⟶ *intval-add a* (*intval-sub b a*) = *b*)
⟹ *BinaryExpr BinAdd* e_2 (*BinaryExpr BinSub* e_1 e_2) ≥ e_1)
unfolding *le-expr-def unfold-binary*
subgoal premises *1*
apply (*rule allI*)+
subgoal for *m p v*
apply *auto*
subgoal premises *2* **for** *a b c*
thm *evalDet*[*OF 2*(*1*) *2*(*5*)]
unfolding *evalDet*[*OF 2*(*1*) *2*(*5*)]
using *2*(*2*) *2*(*4*) *NeutralRightSub-1* ‹*a* = *c*› **by** *fastforce*
done
done
done

lemma *NeutralRightSub-3*:
(*size* e_1 < *size* (*BinaryExpr BinAdd* e_2 (*BinaryExpr BinSub* e_1 e_2)))
using *size-non-const* **by** *fastforce*

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

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

print-phases

lemma *val-redundant-add-sub*:
  assumes $val[b + a] \neq UndefVal$
  shows $val[(b + a) - b] = a$
  using *assms* **by** (*cases* a; *cases* b; *auto*)

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

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

optimization *opt-redundant-sub-add*: $(b + a) - b \longmapsto a$
   apply *auto* **using** *val-redundant-add-sub*
  by (*metis evalDet*)

optimization *opt-add-right-negate-to-sub*: $(x + (-e)) \longmapsto x - e$
   using *AddToSubHelperLowLevel intval-add-sym* **by** *auto*

optimization *opt-add-left-negate-to-sub*: $-x + y \longmapsto y - x$
  using *exp-add-left-negate-to-sub* **by** *blast*

end

end

theory *AndPhase*
  imports
    Common
    NewAnd
begin

3 Optimizations for And Nodes

phase *AndPhase*
  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 $val[x \ \& \ x] \neq UndefVal$
  shows $val[x \ \& \ x] = x$
   using *assms*
  by (*cases x*; *auto*)

lemma *val-and-nots*:
  $val[\sim x \ \& \ \sim y] = val[\sim(x \mid y)]$
  by (*cases x*; *cases y*; *auto*)

lemma *val-and-neutral-32*:
  assumes *is-IntVal32 x*
  shows $val[x \ \& \ \sim(IntVal32\ 0)] = x$
   using *assms*
  by (*cases x*; *auto*)

lemma *val-and-neutral-64*:
  assumes *is-IntVal64 x*
  shows $val[x \ \& \ \sim(IntVal64\ 0)] = x$
   using *assms*
  by (*cases x*; *auto*)

lemma *val-and-sign-extend*:
  assumes $e = (1 << In)-1$
  shows $val[(intval\text{-}sign\text{-}extend\ In\ Out\ x) \ \& \ (IntVal32\ e)] = intval\text{-}zero\text{-}extend\ In$

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

lemma *val-and-sign-extend-2*:
  assumes $e = (1 << In) - 1 \wedge$ *intval-and* (*intval-sign-extend In Out x*) (*IntVal32 e*) $\neq$ *UndefVal*
  shows *val*[(*intval-sign-extend In Out x*) & (*IntVal32 e*)] = *intval-zero-extend In Out x*
  using *assms* **apply** (*cases x*; *auto*)
  sorry

lemma *val-and-zero-32*:
  assumes *is-IntVal32 x*
  shows *val*[*x* & (*IntVal32 0*)] = *IntVal32 0*
   using *assms*
  by (*cases x*; *auto*)

lemma *val-and-zero-64*:
  assumes *is-IntVal64 x*
  shows *val*[*x* & (*IntVal64 0*)] = *IntVal64 0*
   using *assms*
  by (*cases x*; *auto*)

lemma *exp-and-equal*:
  exp[*x* & *x*] $\geq$ *exp*[*x*]
   apply *simp* **using** *val-and-equal*
  by (*metis bin-eval.simps*(*4*) *evalDet evaltree-not-undef unfold-binary*)

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

lemma *exp-and-neutral-64*:
  exp[*x* & ~(*const* (*IntVal64 0*))] $\geq$ *x*
  apply (*cases x*; *simp*) **using** *val-and-neutral-64 bin-eval.simps*(*4*)
   apply (*smt* (*verit*) *BinaryExprE Value.collapse*(*1*) *intval-and.simps*(*12*) *intval-not.simps*(*2*)
      is-IntVal-def unary-eval.simps(*3*) *unary-eval-int unfold-const64 unfold-unary*)

      using *val-and-neutral-64 bin-eval.simps*(*4*)
   apply (*metis* (*no-types*, *lifting*) *BinaryExprE UnaryExprE Value.collapse*(*1*) *bin-eval-int*
      intval-and.simps(*12*) *intval-not.simps*(*2*) *is-IntVal-def unary-eval.simps*(*3*) *unfold-const64*)
      using *val-and-neutral-64 bin-eval.simps*(*4*) *unary-eval.simps*(*3*)

apply (*smt* (*z3*) *BinaryExprE UnaryExprE Value.discI*(*2*) *Value.distinct*(*9*) *intval-and.elims*
intval-not.simps(*2*) *unfold-const64*)
using *val-and-neutral-64 bin-eval.simps*(*4*) *unary-eval.simps*(*3*) *bin-and-neutral*

unfold-const64 intval-and.elims intval-not.simps(*2*)
sorry

lemma *exp-and-neutral-32*:
exp[x & $\sim$(*const* (*IntVal32 0*))] $\geq x$
apply *simp-all* **apply** (*cases x*; *simp*) **using** *val-and-neutral-32 bin-eval.simps*(*4*)

apply (*metis* (*no-types*, *lifting*) *UnaryExprE Value.collapse*(*2*) *intval-and.simps*(*5*)

intval-not.simps(*1*) *is-IntVal-def unary-eval.simps*(*3*) *unary-eval-int unfold-binary*
unfold-const32)
using *val-and-neutral-32 bin-eval.simps*(*4*)
apply (*smt* (*verit*) *UnaryExprE Value.collapse*(*2*) *bin-eval-int intval-and.simps*(*5*)

intval-not.simps(*1*) *is-IntVal-def unary-eval.simps*(*3*) *unfold-binary unfold-const32*)
using *val-and-neutral-32 bin-eval.simps*(*4*) *unary-eval.simps*(*3*)
unfold-const32 intval-and.elims
apply (*smt* (*z3*) *BinaryExprE UnaryExprE Value.discI*(*1*) *Value.distinct*(*1*) *intval-and.simps*(*12*))
using *val-and-neutral-32 bin-eval.simps*(*4*) *unary-eval.simps*(*3*) *bin-and-neutral*

unfold-const32 intval-and.elims intval-not.simps(*2*)
sorry

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

optimization *opt-AndShiftConstantRight*: ((*const* x) & y) $\longmapsto y$ & (*const* x)
when $\neg$(*is-ConstantExpr* y)
using *intval-and-commute bin-eval.simps*(*4*) **apply** *auto*
sorry

optimization *opt-and-right-fall-through*: (x & y) $\longmapsto y$
when (((*and* (*not* (*IRExpr-down* x)) (*IRExpr-up* y)) = *0*))
by (*simp add*: *IRExpr-down-def IRExpr-up-def*)

optimization *opt-and-left-fall-through*: (x & y) $\longmapsto x$
when (((*and* (*not* (*IRExpr-down* y)) (*IRExpr-up* x)) = *0*))
by (*simp add*: *IRExpr-down-def IRExpr-up-def*)

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

optimization *opt-and-sign-extend*: *BinaryExpr BinAnd* (*UnaryExpr* (*UnarySignExtend In Out*) *x*)
(*ConstantExpr* (*IntVal32 e*))
$\longmapsto$ (*UnaryExpr* (*UnaryZeroExtend In Out*) *x*)
when (e = (1 << *In*) − 1)
 apply *simp-all*
 apply *auto*
 sorry

definition *wf-stamp* :: *IRExpr* ⇒ *bool* **where**
 wf-stamp e = (∀ *m p v*. ([*m*, *p*] ⊢ *e* ↦ *v*) ⟶ *valid-value v* (*stamp-expr e*))

optimization *opt-and-neutral-32*: (x & $\sim$(*const* (*IntVal32 0*))) $\longmapsto$ x
 when (*wf-stamp x* ∧ *stamp-expr x* = *default-stamp*)
 apply *auto*
 apply (*cases x*; *simp*) **using** *unary-eval.simps unfold-const32 val-and-neutral-32*
 sorry

end

end

3.1 Conditional Expression

theory *ConditionalPhase*
 imports
  Common
begin

phase *Conditional*
 terminating *size*
begin

lemma *negates*: *is-IntVal32 e* ∨ *is-IntVal64 e* ⟹ *val-to-bool* (*val*[*e*]) ≡ ¬(*val-to-bool* (*val*[!*e*]))
 using *intval-logic-negation.simps* **unfolding** *logic-negate-def*
 by (*smt* (*verit*, *best*) *Value.collapse*(*1*) *is-IntVal64-def val-to-bool.simps*(*1*) *val-to-bool.simps*(*2*)

zero-neq-one)

lemma *negation-condition-intval*:
 assumes $e \neq$ *UndefVal* $\wedge \neg$(*is-ObjRef e*) $\wedge \neg$(*is-ObjStr e*)
 shows *val*[(!*e*) *?* *x* : *y*] = *val*[*e* *?* *y* : *x*]
 using *assms* **by** (*cases e*; *auto simp*: *negates logic-negate-def*)

optimization *negate-condition*: ((!*e*) *?* *x* : *y*) $\longmapsto$ (*e* *?* *y* : *x*)
  apply *simp* **using** *negation-condition-intval*
 by (*smt* (*verit*, *ccfv-SIG*) *ConditionalExpr ConditionalExprE Value.collapse*(*3*) *Value.collapse*(*4*) *Value.exhaust-disc evaltree-not-undef intval-logic-negation.simps*(*4*) *intval-logic-negation.simps*(*5*) *negates unary-eval.simps*(*4*) *unfold-unary*)

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

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

optimization *equal-branches*: (*e* *?* *x* : *x*) $\longmapsto$ *x* **.**

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

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

optimization *b*[*intval*]: ((*x eq y*) *?* *x* : *y*) $\longmapsto$ *y*
 sorry

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 *val-conditional-eliminate-known-less*: $((x < y) \; ? \; x : y) \longmapsto x$
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 *opt-conditional-eq-is-RHS*: ((*BinaryExpr BinIntegerEquals* x y) $?$ x : y) $\longmapsto y$
apply *simp-all* **apply** *auto* **using** *b*
apply (*metis* (*mono-tags*, *lifting*) *Canonicalization.intval.simps*(*1*) *evalDet*
intval-conditional.simps intval-equals.simps(*10*))
done

optimization *opt-normalize-x*: ((x *eq const* (*IntVal32 0*)) $?$
(*const* (*IntVal32 0*)) : (*const* (*IntVal32 1*))) $\longmapsto x$
when (x = *ConstantExpr* (*IntVal32 0*) | (x = *ConstantExpr* (*IntVal32 1*)))
done

optimization *opt-normalize-x2*: ((x *eq* (*const* (*IntVal32 1*))) $?$
(*const* (*IntVal32 1*)) : (*const* (*IntVal32 0*))) $\longmapsto x$
when (x = *ConstantExpr* (*IntVal32 0*) | (x = *ConstantExpr* (*IntVal32 1*)))
done

optimization *opt-flip-x*: ((x *eq* (*const* (*IntVal32 0*))) $?$
(*const* (*IntVal32 1*)) : (*const* (*IntVal32 0*))) $\longmapsto$
$x \oplus$ (*const* (*IntVal32 1*))
when (x = *ConstantExpr* (*IntVal32 0*) | (x = *ConstantExpr* (*IntVal32 1*)))
done

optimization *opt-flip-x2*: ((x *eq* (*const* (*IntVal32 1*))) $?$
(*const* (*IntVal32 0*)) : (*const* (*IntVal32 1*))) $\longmapsto$
$x \oplus$ (*const* (*IntVal32 1*))
when (x = *ConstantExpr* (*IntVal32 0*) | (x = *ConstantExpr* (*IntVal32 1*)))
done

optimization *opt-optimise-integer-test*:
(((x & (*const* (*IntVal32 1*))) *eq* (*const* (*IntVal32 0*))) $?$
(*const* (*IntVal32 0*)) : (*const* (*IntVal32 1*))) $\longmapsto$

x & (*const* (*IntVal32 1*))
when (*stamp-expr x* = *default-stamp*)
apply *simp-all*
apply *auto*
using *val-optimise-integer-test* **sorry**

optimization *opt-optimise-integer-test-2*:
(((*x* & (*const* (*IntVal32 1*))) *eq* (*const* (*IntVal32 0*))) *?*
(*const* (*IntVal32 0*)) : (*const* (*IntVal32 1*))) $\longmapsto$
x
when (*x* = *ConstantExpr* (*IntVal32 0*) | (*x* = *ConstantExpr* (*IntVal32 1*)))
done

optimization *opt-conditional-eliminate-known-less*: ((*x* < *y*) *?* *x* : *y*) $\longmapsto$ *x*
when (((*stamp-under* (*stamp-expr x*) (*stamp-expr y*)) |
((*stpi-upper* (*stamp-expr x*)) = (*stpi-lower* (*stamp-expr y*))))
$\wedge$ *wf-stamp x* $\wedge$ *wf-stamp y*)
unfolding *le-expr-def* **apply** *auto*
using *stamp-under.simps wf-stamp-def val-conditional-eliminate-known-less*
sorry

end

end
theory *MulPhase*
imports
Common
begin

4 Optimizations for Mul Nodes

phase *MulPhase*
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*:
$(x:: 'a::len\ word) * (2\hat{}j) = x << j$
by *simp*

lemma *val-eliminate-redundant-negative*:
assumes $(intval\text{-}negate\ x * intval\text{-}negate\ y) \neq UndefVal$
shows $val[-x * -y] = val[x * y]$
by (*cases x*; *cases y*; *auto*)

lemma *val-multiply-neutral-32*:
assumes *is-IntVal32 x*
shows $val[x] * (IntVal32\ 1) = val[x]$
using *assms is-IntVal32-def times-Value-def* **by** *fastforce*

lemma *val-multiply-neutral-64*:
assumes *is-IntVal64 x*
shows $val[x] * (IntVal64\ 1) = val[x]$
using *assms* **by** (*metis Value.collapse*(*2*) *intval-mul.simps*(*2*) *mult.right-neutral times-Value-def*)

lemma *val-multiply-zero-32*:
assumes *is-IntVal32 x*
shows $val[x] * (IntVal32\ 0) = IntVal32\ 0$
using *assms* **by** (*metis Value.collapse*(*1*) *intval-mul.simps*(*1*) *mult-not-zero times-Value-def*)

lemma *val-multiply-zero-64*:
assumes *is-IntVal64 x*
shows $val[x] * (IntVal64\ 0) = IntVal64\ 0$
using *assms intval-mul.simps*(*2*) **by** (*metis Value.collapse*(*2*) *mult-zero-right times-Value-def*)

lemma *val-multiply-negative-32*:
assumes *is-IntVal32 x*
shows $x * intval\text{-}negate\ (IntVal32\ 1) = intval\text{-}negate\ x$
using *assms is-IntVal32-def times-Value-def* **by** *force*

lemma *val-multiply-negative-64*:

assumes *is-IntVal64 x*
shows *x* ∗ *intval-negate* (*IntVal64 1*) = *intval-negate x*
using *assms is-IntVal64-def times-Value-def* **by** *fastforce*

fun *intval-log2* :: *Value* ⇒ *Value* **where**
intval-log2 (*IntVal32 v*) = *IntVal32* (*word-of-int* (*SOME e. v=2^e*)) |
intval-log2 (*IntVal64 v*) = *IntVal64* (*word-of-int* (*SOME e. v=2^e*)) |
intval-log2 - = *UndefVal*

lemma *largest-32*:
assumes *y* = *IntVal32* (*4294967296*) ∧ *i* = *intval-log2 y*
shows *val-to-bool*(*val*[*i* < *IntVal32* (*32*)])
using *assms* **apply** (*cases y*; *auto*)
sorry

lemma *log2-range*:
assumes *is-IntVal32 y* ∧ *intval-log2 y* = *i*
shows *val-to-bool* (*val*[*i* < *IntVal32* (*32*)])
using *assms* **apply** (*cases y*; *cases i*; *auto*)
sorry

lemma *val-multiply-power-2-last-subgoal*:
assumes *y* = *IntVal32 yy*
and *x* = *IntVal32 xx*
and *val-to-bool* (*val*[*IntVal32 0* < *x*])
and *val-to-bool* (*val*[*IntVal32 0* < *y*])

shows *x* ∗ *y* = *IntVal32* (*xx* << *unat* (*and* (*word-of-nat* (*SOME e. yy* = *2^e*)) *31*))
using *intval-left-shift.simps*(*1*) *assms* **apply** (*cases x*; *cases y*; *auto*)
sorry

value *IntVal32 x2* ∗ *IntVal32 x2a*
value *IntVal32* (*x2* << *unat* (*and* (*word-of-nat* (*SOME e. x2a* = *2^e*)) *31*))

value *val*[(*IntVal32 2*) ∗ (*IntVal32 4*)]
value *val*[(*IntVal32 2*) << (*IntVal32 2*)]
value *IntVal32* (*2* << *unat* (*and* (*2*::*32 word*) (*31*::*32 word*)))

lemma *val-multiply-power-2-2*:
assumes *is-IntVal32 y*
and *intval-log2 y* = *i*
and *val-to-bool* (*val*[*IntVal32 0* < *i*])

and *val-to-bool* (*val*[i < *IntVal32 32*])
and *val-to-bool* (*val*[*IntVal32 0* < x])
and *val-to-bool* (*val*[*IntVal32 0* < y])

shows $x * y$ = *val*[x << i]
using *assms* **apply** (*cases x*; *cases y*; *auto*)
apply (*simp add*: *times-Value-def*)
using *val-multiply-power-2-last-subgoal times-Value-def assms* **by** *auto*

lemma *val-multiply-power-2*:
fixes j :: *32 word*
assumes *is-IntVal32 x* $\wedge$ $j \geq 0$ $\wedge$ *j-AsNat* = (*nat* (*Values.intval* (*IntVal32 j*)))
shows x * *IntVal32* (*2* ^ *j-AsNat*) = *intval-left-shift x* (*IntVal32 j*)
using *assms* **apply** (*cases x*; *cases j*; *cases j-AsNat*; *auto*)
sorry

lemma *exp-multiply-zero-64*:
exp[x * (*const* (*IntVal64 0*))] $\geq$ *ConstantExpr* (*IntVal64 0*)
apply (*cases x*; *auto*) **using** *val-multiply-zero-64 unfold-const64*
apply (*metis intval-mul.simps*(*12*) *is-IntVal32-def is-IntVal-def times-Value-def unary-eval-int*)
using *val-multiply-zero-64 unfold-const64*
apply (*metis bin-eval-int intval-mul.simps*(*12*) *is-IntVal32-def is-IntVal-def times-Value-def*)
using *val-multiply-zero-64 intval-mul.simps*(*2*) *unfold-const64*
apply (*metis* (*no-types*, *opaque-lifting*) *Value.exhaust intval-mul.simps*(*12*) *intval-mul.simps*(*8*)
intval-mul.simps(*9*) *mult.commute mult-zero-left*)
using *val-multiply-zero-64 bin-multiply-eliminate intval-mul.simps*(*2*) *unfold-const64*
intval-mul.simps(*12*)
apply (*smt* (*verit*, *ccfv-SIG*) *Value.disc*(*8*) *Value.sel*(*2*) *intval-mul.simps*(*11*) *intval-mul.simps*(*8*)
is-ObjRef-def times-Value-def val-to-bool.elims(*3*) *val-to-bool.simps*(*4*)
valid-value.simps(*19*) *wf-bool.elims*(*2*) *wf-bool.elims*(*3*))

using *val-multiply-zero-64 bin-multiply-eliminate intval-mul.simps*(*2*) *unfold-const64*
intval-mul.simps(*12*)
sorry

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

optimization *opt-MultiplyNeutral32*: $x * ConstantExpr\ (IntVal32\ 1) \longmapsto x$
apply *auto* **using** *val-multiply-neutral-32 bin-eval.simps*(*2*)
by (*smt* (*z3*) *Value.discI*(*1*) *Value.distinct*(*9*) *intval-mul.elims times-Value-def*)

optimization *opt-MultiplyNeutral64*: $x * ConstantExpr\ (IntVal64\ 1) \longmapsto x$
apply *auto* **using** *val-multiply-neutral-64*
by (*metis Value.exhaust evaltree-not-undef intval-mul.simps*(*12*) *intval-mul.simps*(*13*)

intval-mul.simps(*14*) *is-IntVal64-def times-Value-def*)

optimization *opt-MultiplyZero32*: $x * ConstantExpr\ (IntVal32\ 0) \longmapsto const\ (IntVal32\ 0)$
apply *auto* **using** *val-multiply-zero-32*
by (*metis Value.disc*(*2*) *Value.exhaust intval-mul.simps*(*3*) *intval-mul.simps*(*5*) *intval-mul.simps*(*8*)
intval-mul.simps(*9*) *times-Value-def unfold-const32*)

optimization *opt-MultiplyZero64*: $x * ConstantExpr\ (IntVal64\ 0) \longmapsto const\ (IntVal64\ 0)$
using *exp-multiply-zero-64* **by** *simp*

optimization *opt-MultiplyNegative32*: $x * -(const\ (IntVal32\ 1)) \longmapsto -x$
apply *auto* **using** *val-multiply-negative-32*

sorry

optimization *opt-MultiplyNegative64*: $x * -(const\ (IntVal64\ 1)) \longmapsto -x$
apply *auto* **using** *val-multiply-negative-64*
sorry

end

end
theory *NegatePhase*
imports
Common
begin

5 Optimizations for Negate Nodes

phase *NegatePhase*
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 e* $\neq$ *UndefVal*
shows *val*[$-(-(e))$] = *val*[e]
by (*metis* (*no-types*, *lifting*) *assms intval-negate.elims intval-negate.simps*(*1*)
intval-negate.simps(*2*) *verit-minus-simplify*(*4*))

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

lemma *exp-distribute-sub*:
shows *exp*[$-(x-y)$] $\geq$ *exp*[$y-x$]
apply (*cases x*; *cases y*; *auto*) **using** *unfold-binary val-distribute-sub*
apply *auto*[*1*]
apply (*metis BinaryExpr UnaryExpr bin-eval.simps*(*3*) *val-distribute-sub*)
using *bin-eval.simps*(*3*) *val-distribute-sub* **apply** *auto*[*1*]
apply (*metis BinaryExpr ParameterExpr UnaryExpr bin-eval.simps*(*3*) *int-val-sub.simps*(*10*)
val-distribute-sub)
apply (*metis BinaryExpr LeafExpr UnaryExpr bin-eval.simps*(*3*) *intval-sub.simps*(*10*)

val-distribute-sub)
apply (*metis BinaryExpr ConstantExpr UnaryExpr bin-eval.simps*(*3*) *eval-tree-not-undef*
val-distribute-sub)
apply (*metis BinaryExpr UnaryExpr bin-eval.simps*(*3*) *val-distribute-sub*)
apply (*metis BinaryExpr bin-eval.simps*(*3*) *val-distribute-sub*) **using** *val-distribute-sub*

apply *auto*[*1*]
apply (*metis BinaryExpr ParameterExpr bin-eval.simps*(*3*) *evaltree-not-undef val-distribute-sub*)
apply (*metis BinaryExpr LeafExpr bin-eval.simps*(*3*) *val-distribute-sub valid-value.simps*(*4*))
apply (*metis BinaryExpr ConstantExpr bin-eval.simps*(*3*) *evaltree-not-undef val-distribute-sub*)
using *unfold-binary val-distribute-sub* **apply** *auto*[*1*]

```
        using val-distribute-sub apply auto[1]
        using unfold-binary val-distribute-sub apply auto[1]
  apply (smt (verit, best) ConditionalExpr ParameterExpr bin-eval.simps(3) int-
val-sub.simps(10)
        unfold-binary val-distribute-sub)
  apply (smt (verit, ccfv-SIG) BinaryExpr ConditionalExpr LeafExpr bin-eval.simps(3)

        intval-sub.simps(10) val-distribute-sub)
  apply (smt (verit, ccfv-SIG) ConditionalExpr ConstantExpr bin-eval.simps(3)
intval-sub.simps(10)
        unfold-binary val-distribute-sub)
  apply (metis BinaryExpr ParameterExpr UnaryExpr bin-eval.simps(3) int-
val-sub.simps(3)
        val-distribute-sub)
  apply (metis BinaryExpr ParameterExpr bin-eval.simps(3) evaltree-not-undef
val-distribute-sub)
  apply (smt (verit, ccfv-SIG) BinaryExpr ConditionalExpr ParameterExpr bin-eval.simps(3)

        evaltree-not-undef val-distribute-sub)
  apply (metis BinaryExpr ParameterExpr bin-eval.simps(3) evaltree-not-undef
val-distribute-sub)
  apply (metis LeafExpr ParameterExpr bin-eval.simps(3) evaltree-not-undef un-
fold-binary
        val-distribute-sub)
  apply (metis ConstantExpr ParameterExpr bin-eval.simps(3) unfold-binary
val-distribute-sub
        valid-value.simps(4))
  apply (metis BinaryExpr LeafExpr UnaryExpr bin-eval.simps(3) intval-sub.simps(3)

        val-distribute-sub)
  apply (metis BinaryExpr LeafExpr bin-eval.simps(3) val-distribute-sub valid-value.simps(4))
  apply (smt (verit, ccfv-SIG) ConditionalExpr LeafExpr bin-eval.simps(3) int-
val-sub.simps(3)
        unfold-binary val-distribute-sub)
  apply (metis LeafExpr ParameterExpr bin-eval.simps(3) evaltree-not-undef un-
fold-binary
        val-distribute-sub)
  apply (metis LeafExpr bin-eval.simps(3) intval-sub.simps(10) intval-sub.simps(3)
unfold-binary
        val-distribute-sub)
  apply (metis BinaryExpr ConstantExpr LeafExpr bin-eval.simps(3) evaltree-not-undef

        val-distribute-sub)
  apply (metis BinaryExpr ConstantExpr UnaryExpr bin-eval.simps(3) eval-
tree-not-undef
        val-distribute-sub)
  apply (metis BinaryExpr ConstantExpr bin-eval.simps(3) evaltree-not-undef
val-distribute-sub)
  apply (smt (verit, ccfv-SIG) BinaryExpr ConditionalExpr ConstantExpr bin-eval.simps(3)
```

evaltree-not-undef val-distribute-sub)
apply (*metis BinaryExpr ConstantExpr ParameterExpr bin-eval.simps*(*3*) *intval-sub.simps*(*10*)
intval-sub.simps(*3*) *val-distribute-sub*)
apply (*metis BinaryExpr ConstantExpr LeafExpr bin-eval.simps*(*3*) *evaltree-not-undef*
val-distribute-sub)
apply (*metis BinaryExpr ConstantExpr bin-eval.simps*(*3*) *evaltree-not-undef val-distribute-sub*)
done

optimization *negate-cancel*: $-(-(e)) \longmapsto e$
apply *simp-all*
by (*metis unary-eval.simps*(*2*) *unfold-unary val-negative-cancel*)

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

optimization *negative-shift-32*: $-($*BinaryExpr BinRightShift x* (*const* (*IntVal32 31*)))) $\longmapsto$
BinaryExpr BinURightShift x (*const* (*IntVal32 31*))
when (*stamp-expr x* = *default-stamp*)
apply *simp-all* **apply** *auto*
sorry

end

end
theory *NotPhase*
imports
Common
begin

6 Optimizations for Not Nodes

phase *NotPhase*
terminating *size*
begin

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

lemma *val-not-cancel*:
assumes $val[{\sim}e] \neq UndefVal$
shows $val[{\sim}({\sim}e)] = e$
using *bin-not-cancel*
by (*metis* (*no-types*, *lifting*) *assms intval-not.elims intval-not.simps*(*1*) *intval-not.simps*(*2*))

lemma *exp-not-cancel*:
shows $exp[{\sim}({\sim}a)] \geq exp[a]$
apply *simp* **using** *val-not-cancel*
by (*metis UnaryExprE unary-eval.simps*(*3*))

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

end

end
theory *OrPhase*
imports
Common
NewAnd
begin

7 Optimizations for Or Nodes

phase *OrPhase*
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 $x \neq \mathit{UndefVal} \wedge ((\mathit{intval\text{-}or}\ x\ x) \neq \mathit{UndefVal})$
shows $\mathit{val}[x \mid x] = \mathit{val}[x]$
apply (*cases x*; *auto*) **using** *bin-or-equal assms*
by *auto+*

lemma *val-elim-redundant-false*:
assumes $x \neq \mathit{UndefVal} \wedge (\mathit{intval\text{-}or}\ x\ (\mathit{bool\text{-}to\text{-}val}\ \mathit{False})) \neq \mathit{UndefVal}$
shows $\mathit{val}[x \mid \mathit{false}] = \mathit{val}[x]$
using *assms* **apply** (*cases x*)
by *simp+*

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

lemma *val-or-not-operands*:
$\mathit{val}[{\sim}x \mid {\sim}y] = \mathit{val}[{\sim}(x\ \&\ y)]$
by (*cases x*; *cases y*; *auto*)

lemma *exp-or-equal*:
$\mathit{exp}[x \mid x] \geq \mathit{exp}[x]$
apply *simp* **using** *val-or-equal*
by (*metis bin-eval.simps*(*5*) *evalDet evaltree-not-undef unfold-binary*)

lemma *exp-elim-redundant-false*:
$\mathit{exp}[x \mid \mathit{false}] \geq \mathit{exp}[x]$
apply *simp* **using** *val-elim-redundant-false*
apply (*cases x*)
by (*metis BinaryExprE bin-eval.simps*(*5*) *bool-to-val.simps*(*2*) *evaltree-not-undef unfold-const32*)+

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

optimization *OrShiftConstantRight*: $((\mathit{const}\ x) \mid y) \longmapsto y \mid (\mathit{const}\ x)$ *when* $\neg(\mathit{is\text{-}ConstantExpr}\ y)$
unfolding *le-expr-def* **using** *val-shift-const-right-helper size-non-const*
apply *simp* **apply** *auto*
sorry

optimization *elim-redundant-false*: $x \mid \mathit{false} \longmapsto x$
by (*meson exp-elim-redundant-false le-expr-def*)

optimization *or-not-operands*: $(\sim x \mid \sim y) \longmapsto \sim (x \,\&\, y)$
apply *auto* **using** *val-or-not-operands*
by (*metis bin-eval.simps*(*4*) *intval-not.simps*(*3*) *unary-eval.simps*(*3*) *unfold-binary*

unfold-unary)

optimization *or-left-fall-through*: $(x \mid y) \longmapsto x$
when (((*and* (*not* (*IRExpr-down x*)) (*IRExpr-up y*)) = *0*))
by (*simp add*: *IRExpr-down-def IRExpr-up-def*)

optimization *or-right-fall-through*: $(x \mid y) \longmapsto y$
when (((*and* (*not* (*IRExpr-down y*)) (*IRExpr-up x*)) = *0*))
by (*meson exp-or-commute or-left-fall-through*(*1*) *order.trans rewrite-preservation.simps*(*2*))

end

end
theory *SignedDivPhase*
imports
Common
begin

8 Optimizations for SignedDiv Nodes

phase *SignedDivPhase*
terminating *size*
begin

lemma *val-division-by-one-is-self-32*:
assumes *is-IntVal32 x*
shows *intval-div x* (*IntVal32 1*) = *x*
using *assms* **by** (*cases x*; *auto*)

end

end
theory *SubPhase*
imports
Common
begin

9 Optimizations for Sub Nodes

phase *SubPhase*
  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 $((x + y) - y \neq UndefVal)$
  shows $val[(x + y) - (y)] = val[x]$
  using *bin-sub-after-right-add*
  using *assms* **apply** (*cases x*; *cases y*; *auto*) **apply** (*simp add*: *minus-Value-def plus-Value-def*)
  by (*simp add*: *minus-Value-def plus-Value-def*)+

lemma *val-sub-after-left-sub*:
assumes $((x - y) - x \neq \mathit{UndefVal})$
shows $\mathit{val}[(x - y) - x] = \mathit{val}[-y]$
using *assms* **apply** (*cases x*; *cases y*; *auto*) **apply** (*simp add*: *minus-Value-def*)
by (*simp add*: *minus-Value-def*)+

lemma *val-sub-then-left-sub*:
assumes $(x - (x - y) \neq \mathit{UndefVal})$
shows $\mathit{val}[x - (x - y)] = \mathit{val}[y]$
using *assms* **apply** (*cases x*; *cases y*; *auto*) **apply** (*simp add*: *minus-Value-def*)
by (*simp add*: *minus-Value-def*)+

lemma *val-subtract-zero*:
assumes *intval-sub x* (*IntVal32 0*) $\neq$ *UndefVal*
shows *intval-sub x* (*IntVal32 0*) $= \mathit{val}[x]$
using *assms* **by** (*induction x*; *simp*)

lemma *val-zero-subtract-value*:
assumes *intval-sub* (*IntVal32 0*) $x \neq$ *UndefVal*
shows *intval-sub* (*IntVal32 0*) $x = \mathit{val}[-x]$
using *assms* **by** (*induction x*; *simp*)

lemma *val-zero-subtract-value-64*:
assumes *intval-sub* (*IntVal64 0*) $x \neq$ *UndefVal*
shows *intval-sub* (*IntVal64 0*) $x = \mathit{val}[-x]$
using *assms* **by** (*induction x*; *simp*)

lemma *val-sub-then-left-add*:
assumes $(x - (x + y) \neq \mathit{UndefVal})$
shows $\mathit{val}[x - (x + y)] = \mathit{val}[-y]$
using *assms* **apply** (*cases x*; *cases y*; *auto*) **apply** (*simp add*: *minus-Value-def plus-Value-def*)
by (*simp add*: *minus-Value-def plus-Value-def*)+

lemma *val-sub-negative-value*:
assumes $(x - (\mathit{intval\text{-}negate}\ y) \neq \mathit{UndefVal})$
shows $\mathit{val}[x - (-y)] = \mathit{val}[x + y]$
using *assms* **apply** (*cases x*; *cases y*)
by (*simp add*: *minus-Value-def plus-Value-def*)+

lemma *val-sub-self-is-zero*:
assumes *is-IntVal32* $x \wedge x - x \neq \mathit{UndefVal}$
shows $\mathit{val}[x - x] = \mathit{IntVal32}\ 0$
using *assms* **by** (*cases x*; *auto*)

lemma *val-sub-self-is-zero-2*:
assumes *is-IntVal64* $x \wedge x - x \neq \mathit{UndefVal}$

shows *val*[$x - x$] = *IntVal64 0*
using *assms* **by** (*cases x*; *auto*)

lemma *val-sub-negative-const*:
assumes *is-IntVal32 y* $\vee$ *is-IntVal64 y* $\wedge$ $x - ($*intval-negate y*$) \neq$ *UndefVal*
shows $x - ($*intval-negate y*$) = x + y$
using *assms* **apply** (*cases x*; *cases y*; *auto*)
by (*simp add*: *minus-Value-def plus-Value-def*)+

lemma *exp-sub-after-right-add*:
shows *exp*[$(x+y)-y$] $\geq$ *exp*[x]
apply *auto*
by (*metis evalDet minus-Value-def plus-Value-def val-sub-after-right-add-2*)

lemma *exp-sub-negative-value*:
exp[$x - (-y)$] $\geq$ *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*)

optimization *sub-after-right-add*: $((x + y) - y) \longmapsto x$
apply *auto*
by (*metis evalDet minus-Value-def plus-Value-def val-sub-after-right-add-2*)

optimization *sub-after-left-add*: $((x + y) - x) \longmapsto y$
apply *auto*
by (*metis add.commute evalDet minus-Value-def plus-Value-def val-sub-after-right-add-2*)

optimization *sub-after-left-sub*: $((x - y) - x) \longmapsto -y$
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 minus-Value-def unary-eval.simps*(*2*) *unfold-unary val-sub-after-left-sub*)

optimization *sub-then-left-add*: $(x - (x + y)) \longmapsto -y$
apply *auto*
apply (*simp add*: *Suc-lessI one-is-add*)
by (*metis evalDet minus-Value-def plus-Value-def unary-eval.simps*(*2*) *unfold-unary*

val-sub-then-left-add)

optimization *sub-then-right-add*: $(y - (x + y)) \longmapsto -x$
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 minus-Value-def plus-Value-def unary-eval.simps*(*2*) *unfold-unary*
val-sub-then-left-add)

optimization *sub-then-left-sub*: $(x - (x - y)) \longmapsto y$
apply *auto*
by (*metis evalDet minus-Value-def val-sub-then-left-sub*)

optimization *subtract-zero*: $(x - (const\ IntVal32\ 0)) \longmapsto x$
apply *auto*
by (*metis val-subtract-zero*)

optimization *subtract-zero-64*: $(x - (const\ IntVal64\ 0)) \longmapsto x$
apply *auto*
by (*smt* (*z3*) *Value.sel*(*2*) *diff-zero intval-sub.elims intval-sub.simps*(*12*))

optimization *sub-negative-value*: $(x - (-y)) \longmapsto x + y$
using *exp-sub-negative-value*
sorry

optimization *zero-sub-value*: $((const\ IntVal32\ 0) - x) \longmapsto -x$
unfolding *size.simps*
apply *simp-all*
apply *auto*
sorry

optimization *zero-sub-value-64*: $((const\ IntVal64\ 0) - x) \longmapsto -x$
unfolding *size.simps*
apply *simp-all*
apply *auto*
sorry

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

optimization *opt-sub-self-is-zero32*: $(x - x) \longmapsto const\ IntVal32\ 0\ when$
$(wf\text{-}stamp\ x \wedge stamp\text{-}expr\ x = default\text{-}stamp)$

apply *simp-all*
apply *auto* **sorry**

end

end
theory *XorPhase*
imports
Common
begin

10 Optimizations for Xor Nodes

phase *XorPhase*
terminating *size*
begin

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

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

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

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

lemma *val-xor-self-is-false-2*:
assumes $(val[x \oplus x]) \neq UndefVal \land is\text{-}IntVal32\ x$
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 $val[x \oplus x] \neq UndefVal \land is\text{-}IntVal64\ x$
shows $val[x \oplus x] = IntVal64\ 0$

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

definition *wf-stamp* :: *IRExpr* ⇒ *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 val-xor-self-is-false-2 wf-stamp-def* **apply** (*cases x*; *auto*)
using *bin-xor-self-is-false*
apply (*smt* (*verit*, *ccfv-threshold*) *evalDet intval-xor.simps*(*1*) *unfold-const32 un-fold-unary*
valid-int32)
apply (*smt* (*verit*, *best*) *BinaryExpr evalDet is-IntVal32-def unfold-const32 valid-int32*)
apply (*smt* (*verit*, *best*) *ConditionalExpr evalDet is-IntVal32-def unfold-const32 valid-int32*)
apply (*metis Value.disc*(*2*) *unfold-const32 valid-int32*)
by (*metis is-IntVal32-def unfold-const32 valid-int32*)+

optimization *xor-self-is-false*: $(x \oplus x) \longmapsto false$ *when*
$(wf\text{-}stamp\ x \wedge stamp\text{-}expr\ x = default\text{-}stamp)$
apply *auto*[*1*]
apply (*simp add*: *Suc-lessI one-is-add*) **using** *exp-xor-self-is-false*
by *auto*

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

optimization *EliminateRedundantFalse*: $(x \oplus \mathit{false}) \longmapsto x$
  using *val-eliminate-redundant-false* **apply** *auto*
  by (*metis*)

optimization *opt-mask-out-rhs*: $(x \oplus \mathit{const}\ y) \longmapsto \mathit{UnaryExpr}\ \mathit{UnaryNot}\ x$
  when $((\mathit{stamp\text{-}expr}\ (x) = \mathit{IntegerStamp}\ \mathit{bits}\ l\ h))$

  unfolding *le-expr-def* **apply** *auto*
  sorry

end

end