To be OR NOT to be

Instruction	Operation	Unit	RISC-V Extension
andn rd, rs1, rs2	rd = rs1 & ~rs2;	Scalar	Zbb or Zbkb
orn rd, rs1, rs2	rd = rs1 | ~rs2;		
xnor rd, rs1, rs2	rd = rs1 ^ ~rs2;		
vandn.vv vd, vs1, vs2	for (int i = 0; i < vl; i++) vd[i] = vs1[i] & ~vs2[i];	Vector	Zvkb
vandn.vx vd, vs1, rs2	for (int i = 0; i < vl; i++) vd[i] = vs1[i] & ~rs2;		

Input	Output
(and rs1, (not rs2))	(andn rs1, rs2)
(or rs1, (not rs2))	(orn rs1, rs2)
(xor rs1, (not rs2))	(xnor rs1, rs2)
(vand vs1, (not vs2))	(vandn vs1, vs2)
(vand vs1, (not rs2))	(vandn vs1, rs2)

Essentially:

Input	Output
(and rs1, (not rs2))	(andn rs1, rs2)
(or rs1, (not rs2))	(orn rs1, rs2)
(xor rs1, (not rs2))	(xnor rs1, rs2)
(vand vs1, (not vs2))	(vandn vs1, vs2)
(vand vs1, (not rs2))	(vandn vs1, rs2)

Real patterns:

```
let Predicates = [HasStdExtZbbOrZbkb] in {
def : Pat<(XLenVT (and GPR:$rs1, (not GPR:$rs2))), (ANDN GPR:$rs1, GPR:$rs2)>;
def : Pat<(XLenVT (or  GPR:$rs1, (not GPR:$rs2))), (ORN  GPR:$rs1, GPR:$rs2)>;
def : Pat<(XLenVT (xor GPR:$rs1, (not GPR:$rs2))), (XNOR GPR:$rs1, GPR:$rs2)>;
}

foreach vti = AllIntegerVectors in {
  let Predicates = !listconcat([HasStdExtZvkb],
                               GetVTypePredicates<vti>.Predicates) in {
    def : Pat<(vti.Vector (and (riscv_vnot vti.RegClass:$rs1),
                               vti.RegClass:$rs2)),
              (!cast<Instruction>("PseudoVANDN_VV_"#vti.LMul.MX)
                 (vti.Vector (IMPLICIT_DEF)),
                 vti.RegClass:$rs2,
                 vti.RegClass:$rs1,
                 vti.AVL, vti.Log2SEW, TA_MA)>;
    def : Pat<(vti.Vector (and (riscv_splat_vector
                                  (not vti.ScalarRegClass:$rs1)),
                               vti.RegClass:$rs2)),
              (!cast<Instruction>("PseudoVANDN_VX_"#vti.LMul.MX)
                 (vti.Vector (IMPLICIT_DEF)),
                 vti.RegClass:$rs2,
                 vti.ScalarRegClass:$rs1,
                 vti.AVL, vti.Log2SEW, TA_MA)>;
  }
}

foreach vti = AllIntegerVectors in {
  let Predicates = !listconcat([HasStdExtZvkb],
                               GetVTypePredicates<vti>.Predicates) in {
    def : Pat<(vti.Vector (riscv_and_vl (riscv_xor_vl
                                           (vti.Vector vti.RegClass:$rs1),
                                           (riscv_splat_vector -1),
                                           (vti.Vector vti.RegClass:$passthru),
                                           (vti.Mask V0),
                                          VLOpFrag),
                                        (vti.Vector vti.RegClass:$rs2),
                                        (vti.Vector vti.RegClass:$passthru),
                                        (vti.Mask V0),
                                       VLOpFrag)),
              (!cast<Instruction>("PseudoVANDN_VV_"#vti.LMul.MX#"_MASK")
                 vti.RegClass:$passthru,
                 vti.RegClass:$rs2,
                 vti.RegClass:$rs1,
                 (vti.Mask V0),
                 GPR:$vl,
                 vti.Log2SEW,
                 TAIL_AGNOSTIC)>;

    def : Pat<(vti.Vector (riscv_and_vl (riscv_splat_vector
                                           (not vti.ScalarRegClass:$rs1)),
                                        (vti.Vector vti.RegClass:$rs2),
                                        (vti.Vector vti.RegClass:$passthru),
                                        (vti.Mask V0),
                                       VLOpFrag)),
              (!cast<Instruction>("PseudoVANDN_VX_"#vti.LMul.MX#"_MASK")
                 vti.RegClass:$passthru,
                 vti.RegClass:$rs2,
                 vti.ScalarRegClass:$rs1,
                 (vti.Mask V0),
                 GPR:$vl,
                 vti.Log2SEW,
                 TAIL_AGNOSTIC)>;
  }
}
```

Optimization #1

Op with constant

↓

Inverted op with inverted constant

Optimization #2

Inversion before the loop

↓

Inverted op in the loop

Optimization #3

Target-independent transforms emitting VANDN

- RISC-V instructions are 32-bit (plus optional 16-bit “compressed” instructions)
- Many instructions have variants with a sign-extended 12-bit immediate

Instructions	Operation	Comment
addi rd, x0, simm12	rd = 0 + simm12	Load small constants by adding to register X0 which is hardwired to zero
lui rd, simm20	rd = simm20 << 12	“Load Upper Immediate”
lui rd, simm20 addi rd, rd, simm12	rd = (simm20 << 12) + simm12	“Load Upper Immediate” followed by an addition

Input	Output
(and rs1, C)	(andn rs1, ~C)
(or rs1, C)	(orn rs1, ~C)
(xor rs1, C)	(xnor rs1, ~C)
(vand vs1, C)	(vandn vs1, ~C)

Example

Before	After
lui a1, HI+1 addi a1, a1, -1 or a0, a0, a1	lui a1, ~HI orn a0, a0, a1

llvm/lib/Target/RISCV/MCTargetDesc/RISCVMatInt.cpp
(>500 LOC)

Emits a sequence of LUI, ADDI, ADDIW, SLLI, SLLI.UW, RORI, NOT,
BSETI, BCLRI, PACK, SH1ADD, SH2ADD, SH3ADD instructions

How to tell if inverting the constant is profitable?

```
int OrigImmCost = RISCVMatInt::getIntMatCost(APInt(64,  Imm), ...);
int NegImmCost  = RISCVMatInt::getIntMatCost(APInt(64, ~Imm), ...);
if (NegImmCost < OrigImmCost)
    PerformTransform();
```

```
inverted = (not mask);
for (...) { result = (and src, inverted); }
```

```
for (...) { result = (and src, (not mask)); }
```

```
for (...) { result = (andn src, mask); }
```

and similarly for "or", "xor" and "vand".

Low-hanging fruit: port this to x86 and Arm.
https://github.com/llvm/llvm-project/issues/108840

```
bool RISCVTargetLowering::hasAndNotCompare(SDValue Y) const {
   EVT VT = Y.getValueType();

   // FIXME: Support vectors once we have tests.
   if (VT.isVector())
     return false;

   return (Subtarget.hasStdExtZbb() || Subtarget.hasStdExtZbkb()) &&
          (!isa<ConstantSDNode>(Y) || cast<ConstantSDNode>(Y)->isOpaque());
 }
```

Don't study LLVM internals too much (i.e. **how** things work), instead:

1. Decide **what** change to make
   a. Invent a transform yourself (example: my optimization #1)
   b. Pick up a GitHub ticket (example: my optimization #2)
   c. Read the code (TODO/FIXME, example: my optimization #3)

2. Find **where** to do your change
   - Dump intermediate representations
   - Look for similar transforms

3. Decide **how** to do your change last
   - Often obvious if you know **what**&**where**
   - Other contributors can help you

Thank you