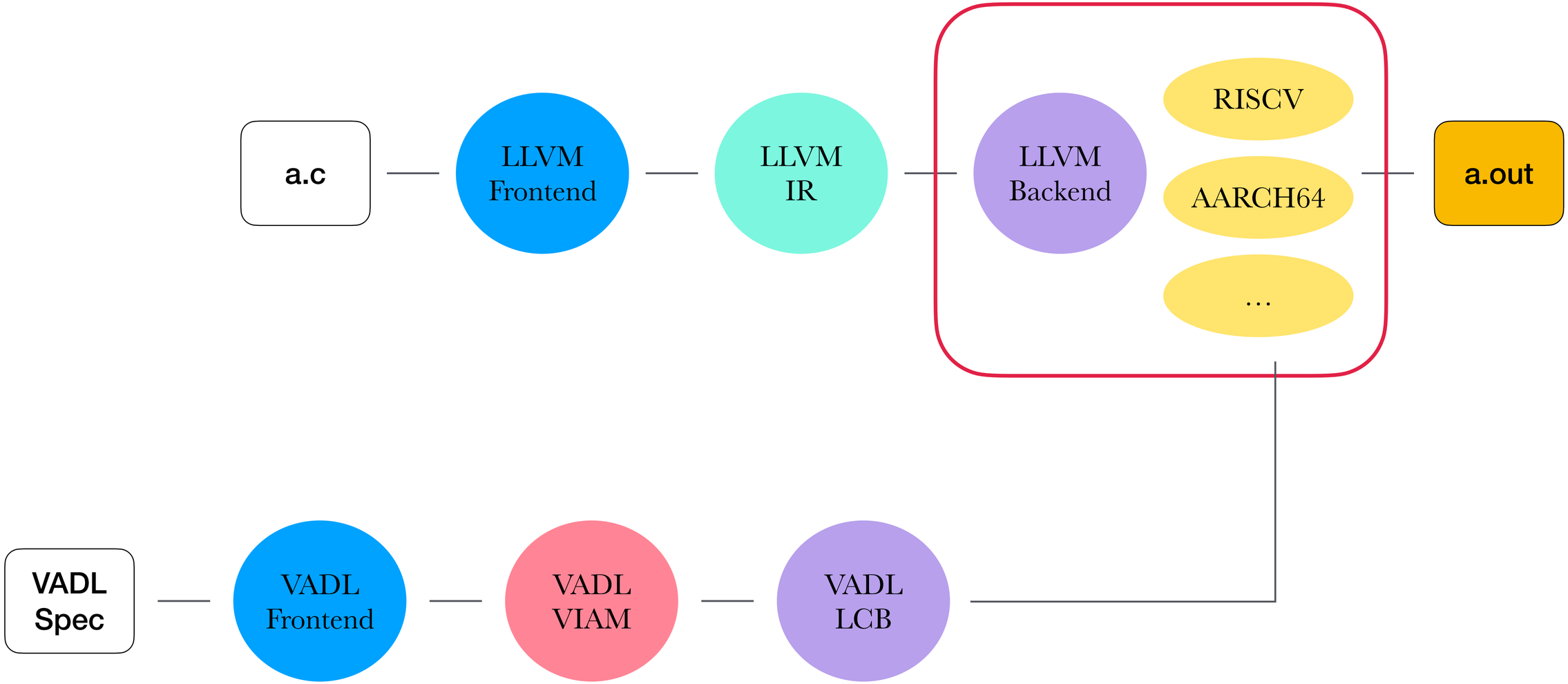
Automatic LLVM compiler backend generation with OpenVADL

By Kevin Per (2025)

OpenVADL

- Vienna Architecture Description Language developed at TU Vienna
- Generating simulator, HDL and compiler automatically
- The initial implementation started 2019
  - Original VADL / Old VADL
- And since summer 2024 reimplementation for Open Source (OpenVADL)
- 12 people

OpenVADL


a.c
LLVM Frontend
LLVM IR
LLVM Backend
RISCV
AARCH64
...
a.out
VADL Spec
VADL Frontend
VADL VIAM
VADL LCB

OpenVADL

```
instruction set architecture RV32I = {

}
```


OpenVADL

```
instruction set architecture RV32I = {

  [X(0) = 0]                                  // register with index 0 always is 0
  register file    X : Index   -> Regs
  program counter PC : Address                // points to the start of the current insert
  memory         MEM : Address -> Byte        // byte addressed memory

  format Rtype : Inst =
    { funct7 : Bits7                          // [31..25] 7 bit function code
    , rs2    : Index
    , rs1    : Index
    , funct3 : Bits3
    , rd     : Index
    , opcode : Bits7
    , shamt  = rs2 as UInt                    // 5 bit unsigned shift amount
    }

  format Itype : Inst =
    { imm    : Bits<12>                       // [31..20] 12 bit immediate value
    , rs1    : Index
    , funct3 : Bits3
    , rd     : Index
    , opcode : Bits7
    , immS   = imm as SIntR                   // sign extended immediate value
    }


}
```


OpenVADL

```
instruction set architecture RV32I = {

  [X(0) = 0]                                  // register with index 0 always is 0
  register file    X : Index   -> Regs
  program counter PC : Address                // points to the start of the current insert
  memory         MEM : Address -> Byte        // byte addressed memory

  format Rtype : Inst =
    { funct7 : Bits7                          // [31..25] 7 bit function code
    , rs2    : Index
    , rs1    : Index
    , funct3 : Bits3
    , rd     : Index
    , opcode : Bits7
    , shamt  = rs2 as UInt                    // 5 bit unsigned shift amount
    }

  format Itype : Inst =
    { imm    : Bits<12>                       // [31..20] 12 bit immediate value
    , rs1    : Index
    , funct3 : Bits3
    , rd     : Index
    , opcode : Bits7
    , immS   = imm as SIntR                   // sign extended immediate value
    }


}
```


OpenVADL

```
instruction set architecture RV32I = {

  [X(0) = 0]                                  // register with index 0 always is 0
  register file    X : Index   -> Regs
  program counter PC : Address                // points to the start of the current insert
  memory         MEM : Address -> Byte        // byte addressed memory

  format Rtype : Inst =
    { funct7 : Bits7                          // [31..25] 7 bit function code
    , rs2    : Index
    , rs1    : Index
    , funct3 : Bits3
    , rd     : Index
    , opcode : Bits7
    , shamt  = rs2 as UInt                    // 5 bit unsigned shift amount
    }

  format Itype : Inst =
    { imm    : Bits<12>                       // [31..20] 12 bit immediate value
    , rs1    : Index
    , funct3 : Bits3
    , rd     : Index
    , opcode : Bits7
    , immS   = imm as SIntR                   // sign extended immediate value
    }


}
```


OpenVADL

```
instruction set architecture RV32I = {

  [X(0) = 0]                                  // register with index 0 always is 0
  register file    X : Index   -> Regs
  program counter PC : Address                // points to the start of the current insert
  memory         MEM : Address -> Byte        // byte addressed memory

  format Rtype : Inst =
    { funct7 : Bits7                          // [31..25] 7 bit function code
    , rs2    : Index
    , rs1    : Index
    , funct3 : Bits3
    , rd     : Index
    , opcode : Bits7
    , shamt  = rs2 as UInt                    // 5 bit unsigned shift amount
    }

  format Itype : Inst =
    { imm    : Bits<12>                       // [31..20] 12 bit immediate value
    , rs1    : Index
    , funct3 : Bits3
    , rd     : Index
    , opcode : Bits7
    , immS   = imm as SIntR                   // sign extended immediate value
    }


}
```


OpenVADL

```
instruction set architecture RV32I = {

  instruction ADD : Rtype =
    X(rd) := X(rs1) + X(rs2)
  encoding ADD = { opcode = 0b011'0011, funct3 = 0b000, funct7 = 0b000'0001 }
  assembly ADD = (mnemonic, " ", register(rd), ",", register(rs1), ",", register(rs2))

}
```


OpenVADL

```
instruction set architecture RV32I = {

  instruction ADD : Rtype =
    X(rd) := X(rs1) + X(rs2)
  encoding ADD = { opcode = 0b011'0011, funct3 = 0b000, funct7 = 0b000'0001 }
  assembly ADD = (mnemonic, " ", register(rd), ",", register(rs1), ",", register(rs2))

}
```


OpenVADL

```
instruction set architecture RV32I = {

  instruction ADD : Rtype =
    X(rd) := X(rs1) + X(rs2)
  encoding ADD = { opcode = 0b011'0011, funct3 = 0b000, funct7 = 0b000'0001 }
  assembly ADD = (mnemonic, " ", register(rd), ",", register(rs1), ",", register(rs2))

}
```


OpenVADL

```
instruction set architecture RV32I = {

  instruction ADD : Rtype =
    X(rd) := X(rs1) + X(rs2)
  encoding ADD = { opcode = 0b011'0011, funct3 = 0b000, funct7 = 0b000'0001 }
  assembly ADD = (mnemonic, " ", register(rd), ",", register(rs1), ",", register(rs2))

  instruction ADDI : Itype =
       X(rd) := X(rs1) + immS
  encoding ADDI = { opcode = 0b011'0111, funct3 = 0b001, funct7 = 0b000'0001 }
  assembly ADDI = (mnemonic, " ", register(rd), ",", register(rs1), ",", decimal(imm))

}
```


OpenVADL

```
instruction set architecture RV32I = {

  instruction ADD : Rtype =
    X(rd) := X(rs1) + X(rs2)
  encoding ADD = { opcode = 0b011'0011, funct3 = 0b000, funct7 = 0b000'0001 }
  assembly ADD = (mnemonic, " ", register(rd), ",", register(rs1), ",", register(rs2))

  instruction ADDI : Itype =
      X(rd) := X(rs1) + immS
  encoding ADDI = { opcode = 0b011'0111, funct3 = 0b001, funct7 = 0b000'0001 }
  assembly ADDI = (mnemonic, " ", register(rd), ",", register(rs1), ",", decimal(imm))

                                                        format Itype : Inst =
                                                            { imm    : Bits<12>
                                                            , rs1    : Index
                                                            , funct3 : Bits3
                                                            , rd     : Index
                                                            , opcode : Bits7
                                                            , immS   = imm as SIntR
                                                            }

}
```


OpenVADL

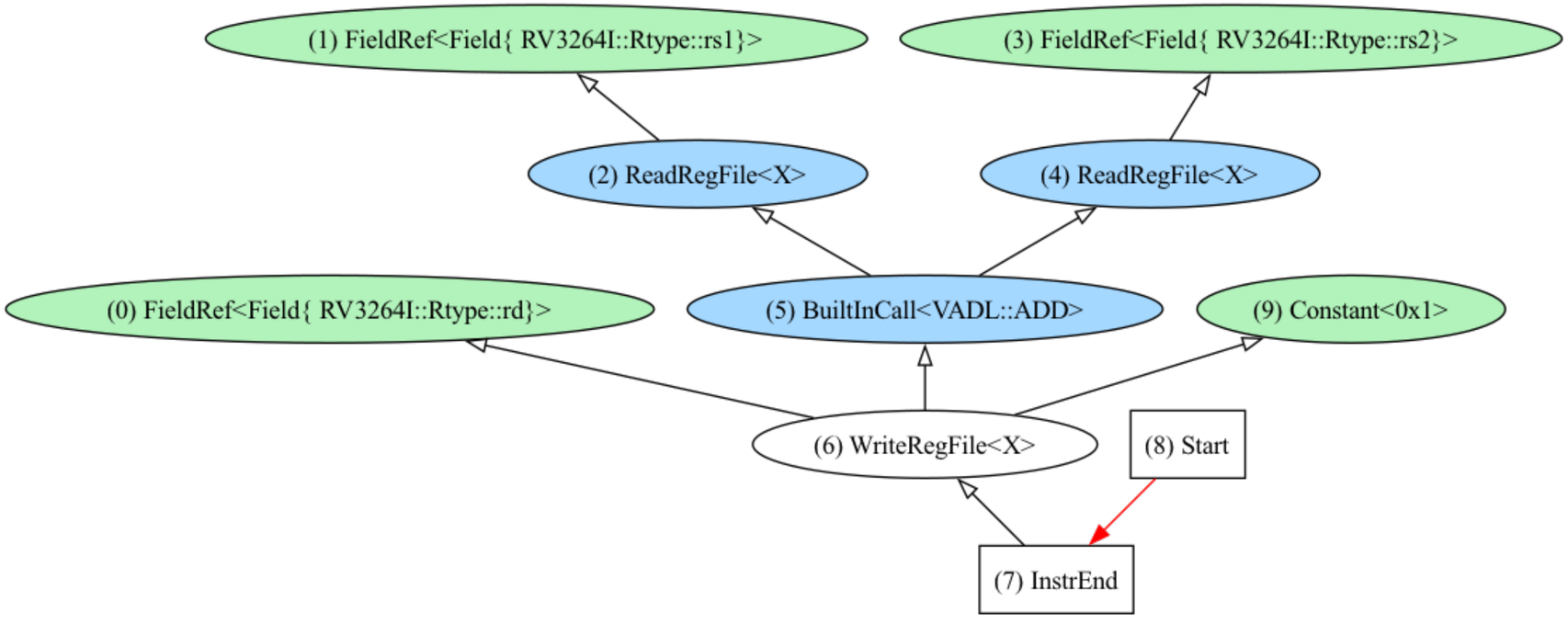
```
instruction set architecture RV32I = {

  instruction ADD : Rtype =
    X(rd) := X(rs1) + X(rs2)
  encoding ADD = { opcode = 0b011'0011, funct3 = 0b000, funct7 = 0b000'0001 }
  assembly ADD = (mnemonic, " ", register(rd), ",", register(rs1), ",", register(rs2))

  instruction ADDI : Itype =
       X(rd) := X(rs1) + immS
  encoding ADDI = { opcode = 0b011'0111, funct3 = 0b001, funct7 = 0b000'0001 }
  assembly ADDI = (mnemonic, " ", register(rd), ",", register(rs1), ",", decimal(imm))

  instruction LB : Itype =
    {
        let addr = X( rs1 ) + immS in
        {
            X(rd) := MEM( addr ) as SInt<32>
        }
    }
    …

}
```


OpenVADL

```
instruction set architecture RV32I = {
  instruction ADD : Rtype =
    X(rd) := X(rs1) + X(rs2)
  encoding ADD = { opcode = 0b011'0011, funct3 = 0b000, funct7 = 0b000'0001 }
  assembly ADD = (mnemonic, " ", register(rd), ",", register(rs1), ",", register(rs2))

  instruction ADDI : Itype =
       X(rd) := X(rs1) + immS
  encoding ADDI = { opcode = 0b011'0111, funct3 = 0b001, funct7 = 0b000'0001 }
  assembly ADDI = (mnemonic, " ", register(rd), ",", register(rs1), ",", decimal(imm))
  instruction LB : Itype =
    {
        let addr = X( rs1 ) + immS in
        {
            X(rd) := MEM( addr ) as SInt<32>
        }
    }
    …

  instruction JAL : Jtype =
      {
          let retaddr = PC.next in {
              PC    := (PC + immS) & (-(2 as SIntR))
              X(rd) := retaddr
          }
      }
    …
}
```


OpenVADL

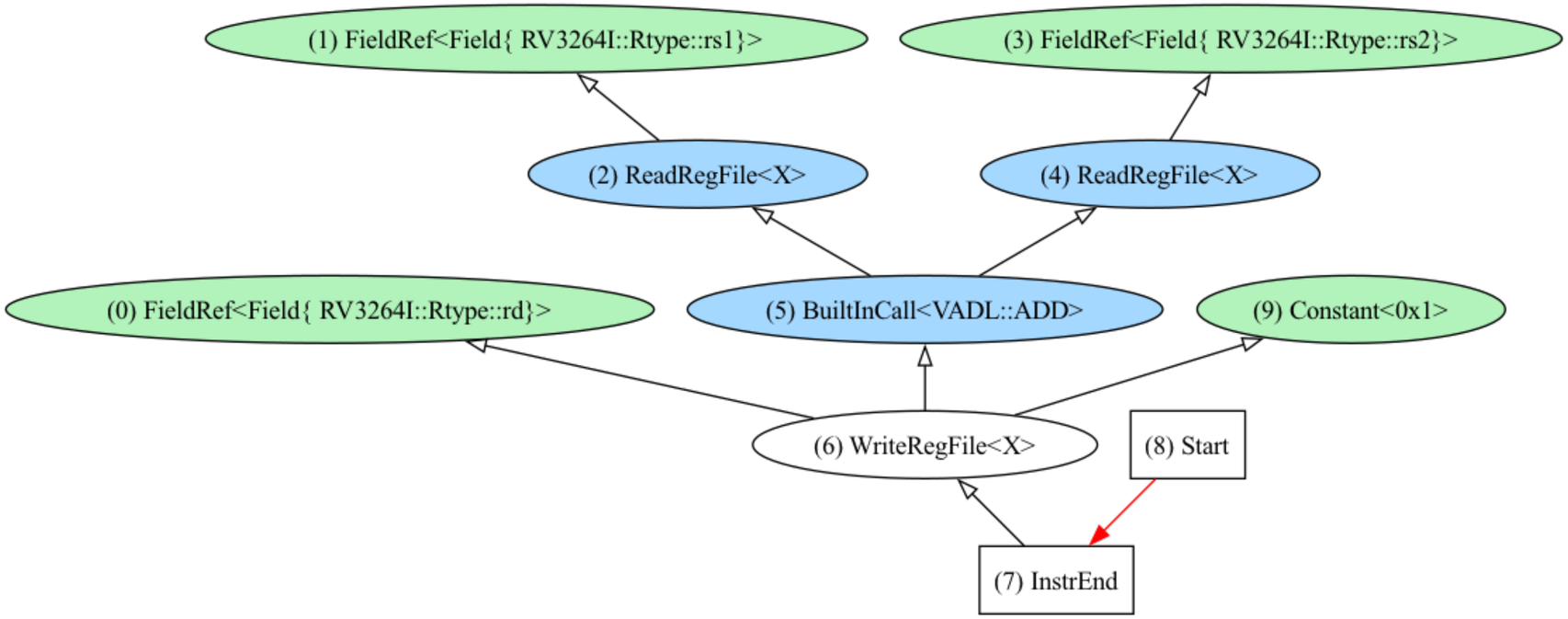
VIAM



ADD Behavior

OpenVADL

VIAM

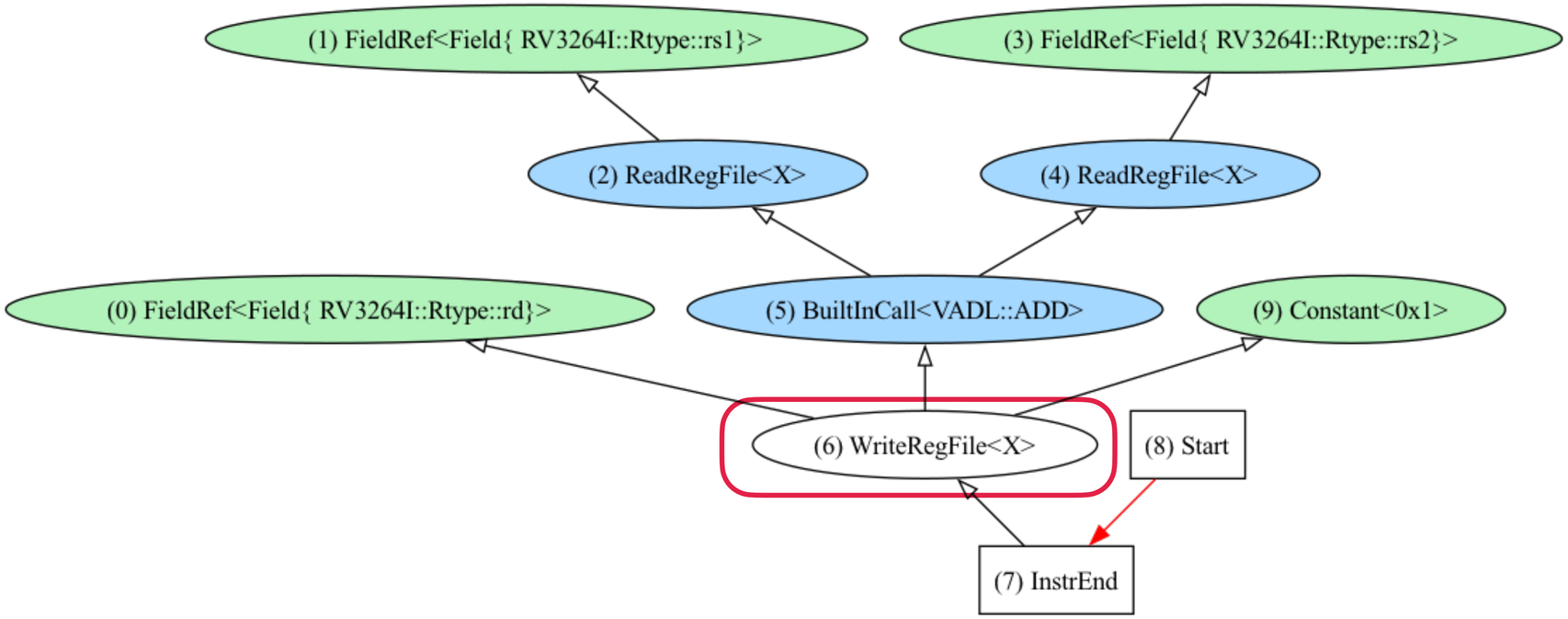


ADD Behavior

```
def : Pat<(add X:$rs1, X:$rs2),
        (ADD X:$rs1, X:$rs2)>;
```


OpenVADL

VIAM

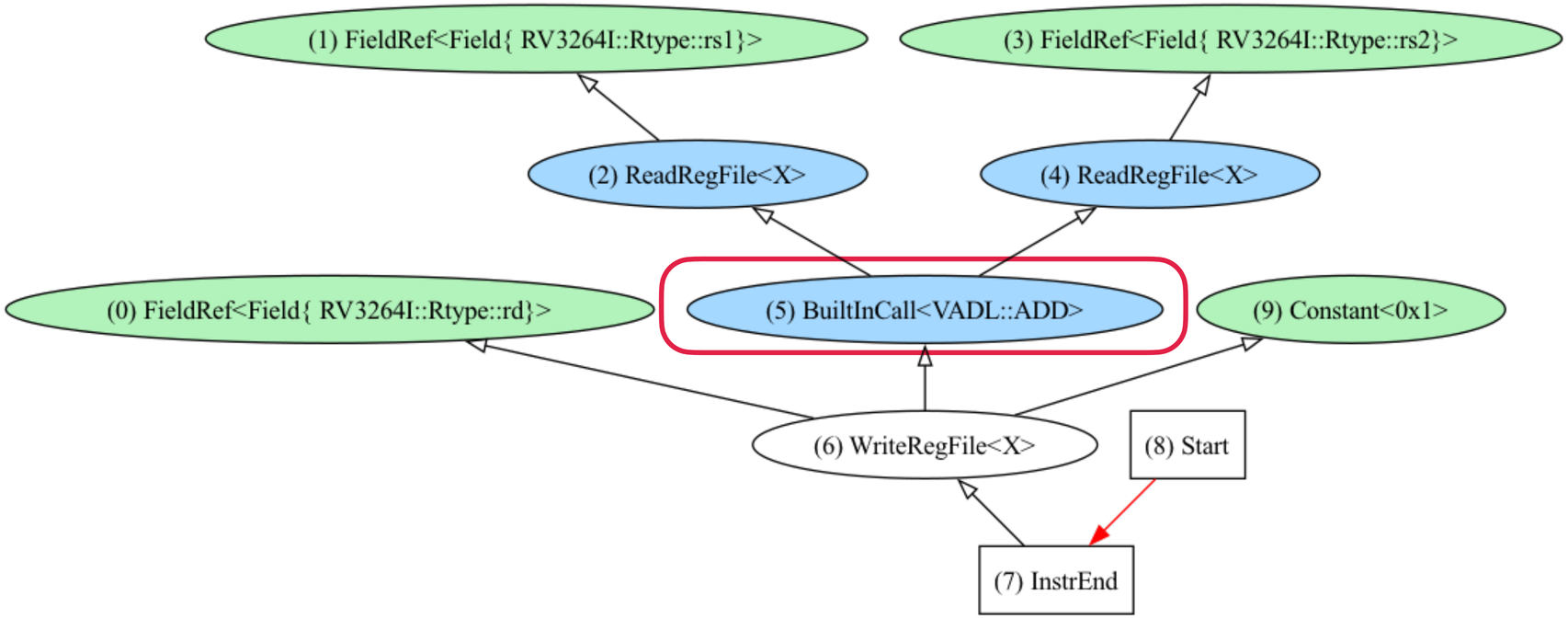


ADD Behavior

```
def : Pat<(add X:$rs1, X:$rs2),
        (ADD X:$rs1, X:$rs2)>;
```


OpenVADL

VIAM

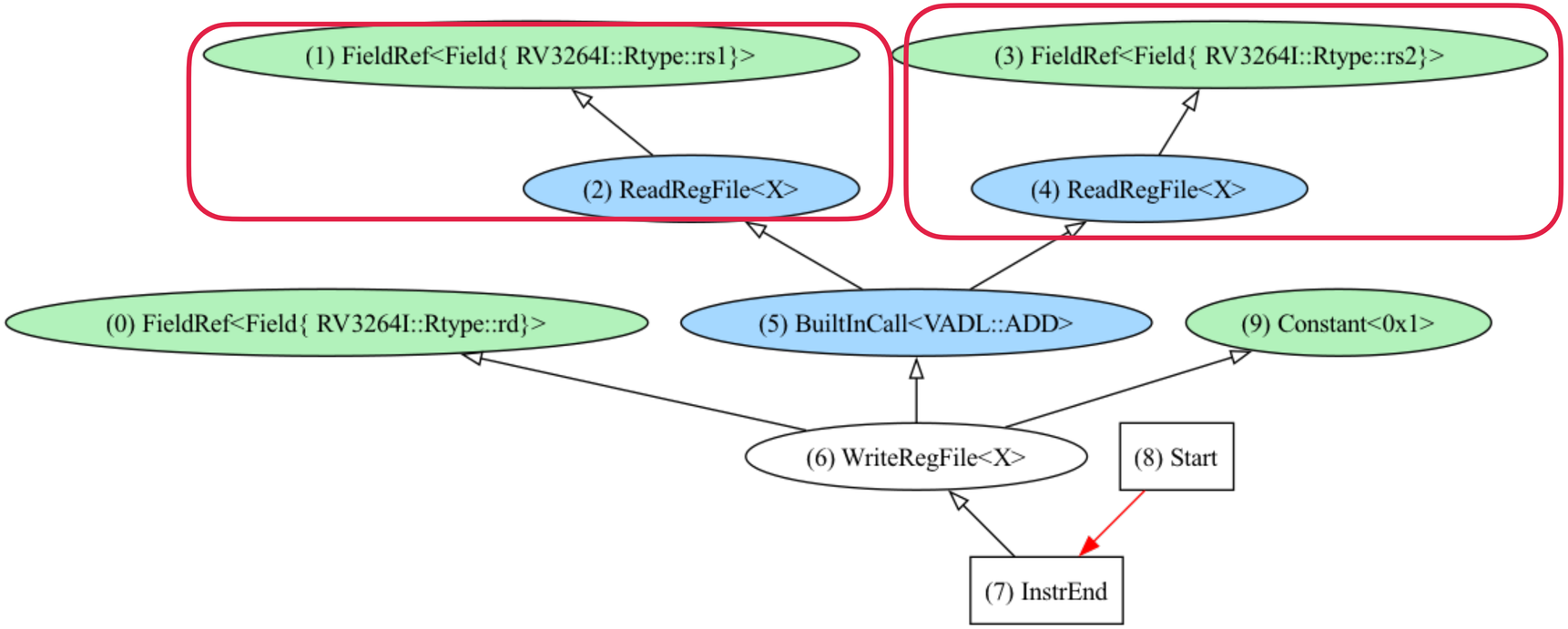


ADD Behavior

```
def : Pat<(add X:$rs1, X:$rs2),
        (ADD X:$rs1, X:$rs2)>;
```


OpenVADL

VIAM



ADD Behavior

```
def : Pat<(add X:$rs1, X:$rs2),
        (ADD X:$rs1, X:$rs2)>;
```

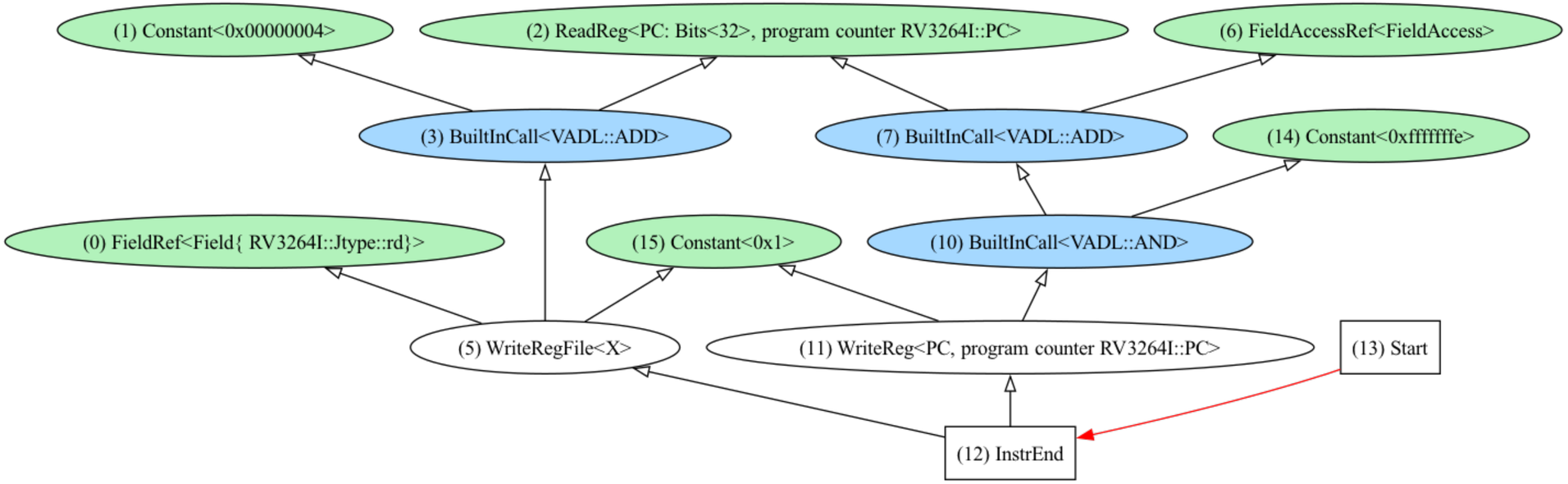

OpenVADL

- Arithmetic
- Logic
- Comparisons
- Unconditional Jumps
- Conditional Jumps
- …

```
instruction set architecture RV32I = {
 instruction JAL : Jtype =
     {
         let retaddr = PC.next in {
             PC    := (PC + immS) & (-(2 as SIntR))
             X(rd) := retaddr
         }
     }
   …

  def : Pat<(br bb:$imm),
        (J RV32I_Jtype_immAsLabel:$imm)>;

}
```


OpenVADL

```
instruction set architecture RV32I = {
 instruction JAL : Jtype =
     {
         let retaddr = PC.next in {
             PC     := (PC + immS) & (-(2 as SIntR))
             X(rd) := retaddr
         }
     }
   …
```

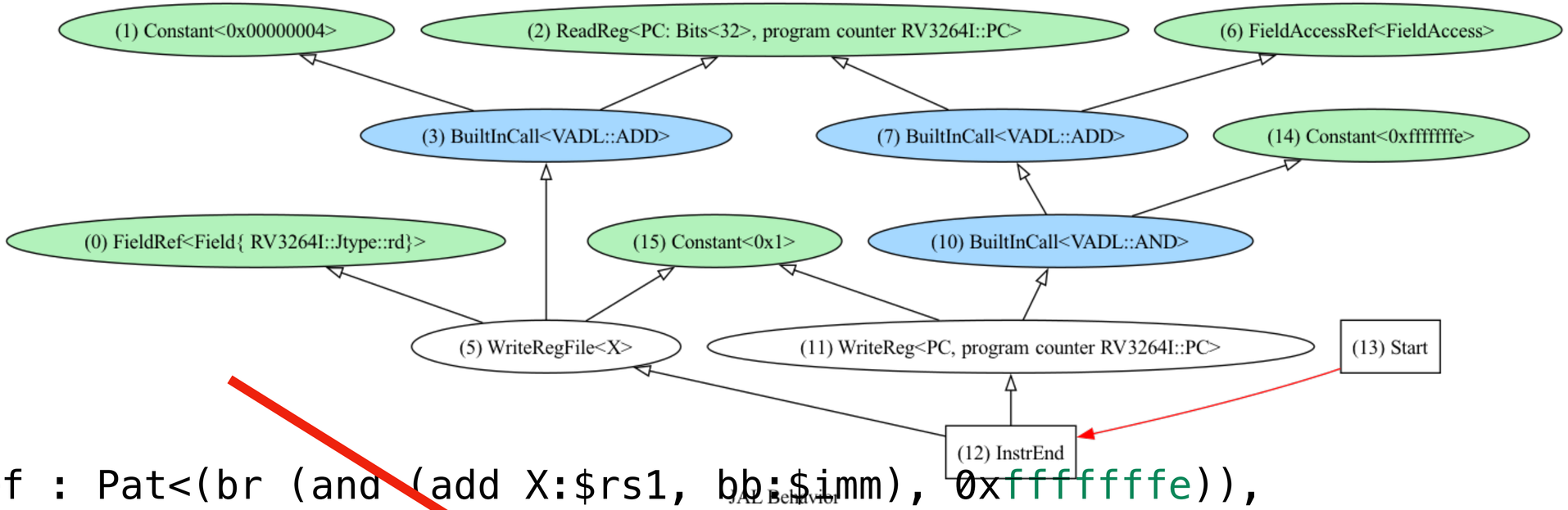


JAL Behavior

```
}
```


OpenVADL

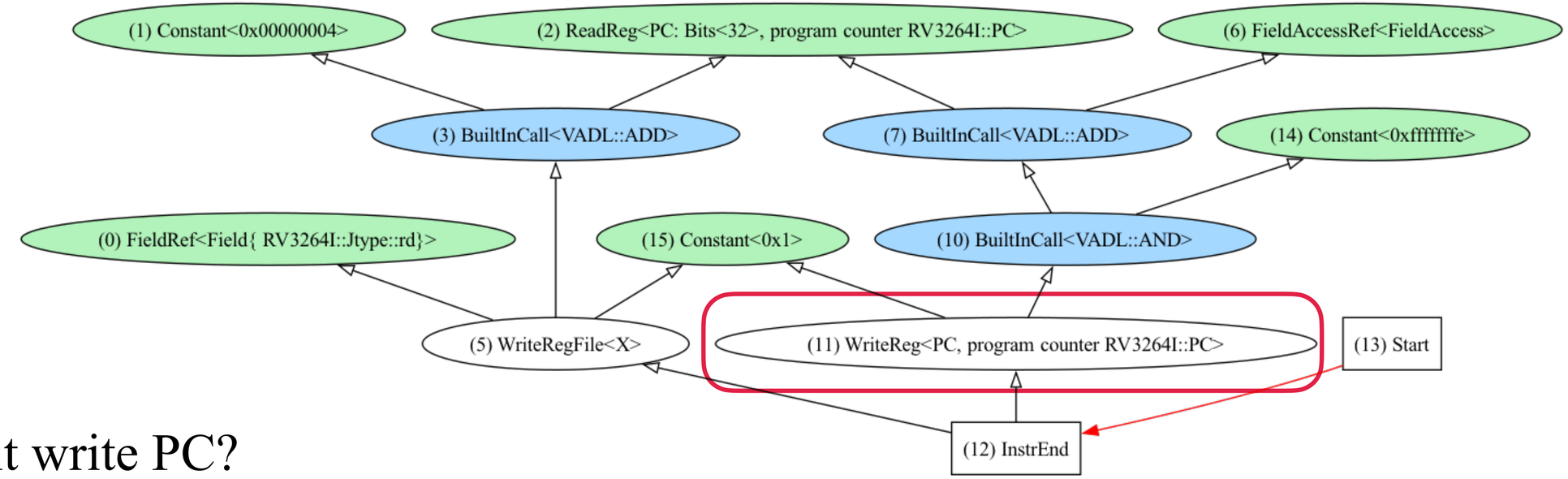
```
instruction set architecture RV32I = {
 instruction JAL : Jtype =
     {
         let retaddr = PC.next in {
             PC     := (PC + immS) & (-(2 as SIntR))
             X(rd) := retaddr
         }
     }
   …
```




```
     def : Pat<(br (and (add X:$rs1, bb:$imm), 0xfffffffe)),
             (J RV32I_Jtype_immAsLabel:$imm)>;

}
```


OpenVADL

```
instruction set architecture RV32I = {
 instruction JAL : Jtype =
     {
         let retaddr = PC.next in {
             PC     := (PC + immS) & (-(2 as SIntR))
             X(rd) := retaddr
         }
     }
  …
```



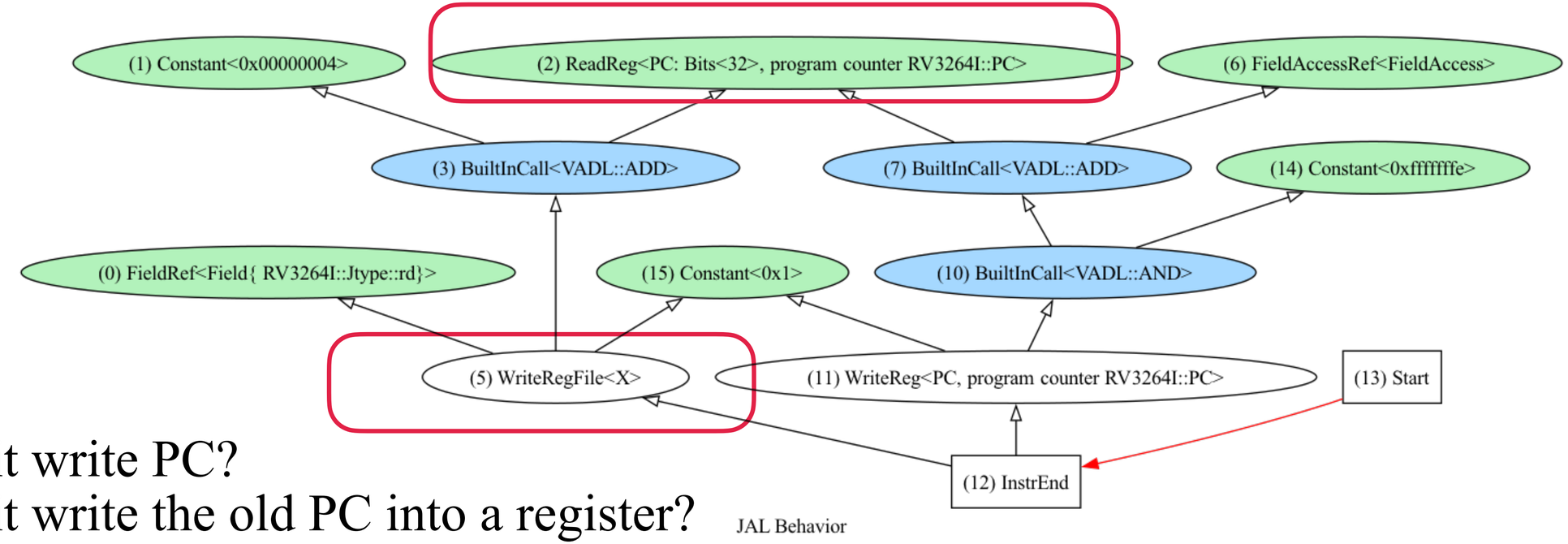
JAL Behavior

Does it write PC?

```
}
```


OpenVADL

```
instruction set architecture RV32I = {
 instruction JAL : Jtype =
     {
         let retaddr = PC.next in {
             PC     := (PC + immS) & (-(2 as SIntR))
             X(rd) := retaddr
         }
     }
  …
```



JAL Behavior

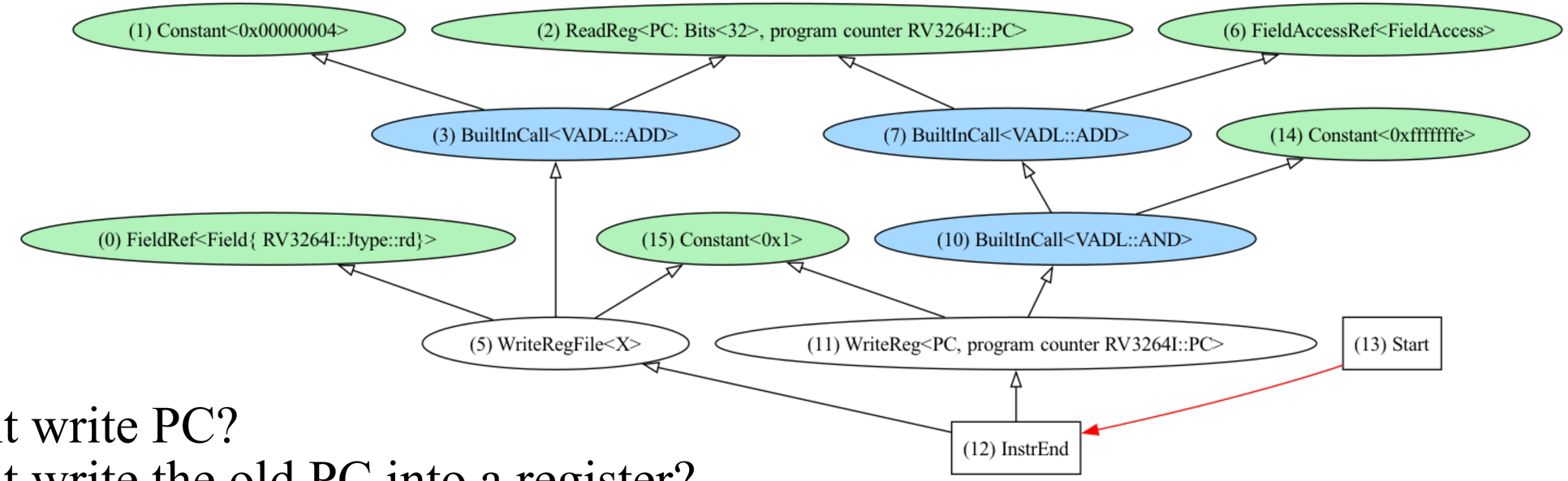
Does it write PC?
Does it write the old PC into a register?

```
}
```


OpenVADL

```
instruction set architecture RV32I = {
 instruction ??? : Jtype =
    {
        let retaddr = PC.next in {
            PC := (PC + immS) & (-(2 as SIntR))
            X(rd) := retaddr
        }
    }
  …
```

Jump and Link



JAL Behavior

Does it write PC?
Does it write the old PC into a register?

```
}
```


OpenVADL

```
instruction set architecture RV32I = {

 instruction JAL : Jtype =
     {
         let retaddr = PC.next in {
             PC    := (PC + immS) & (-(2 as SIntR))
             X(rd) := retaddr
         }
     }
   …

 pseudo instruction J( offset : Bits<20> ) =
     {
         JAL{ rd = 0 as Bits5, imm = offset }
     }


   …

  def : Pat<(br bb:$imm),
        (J RV32I_Jtype_immAsLabel:$imm)>;


}
```

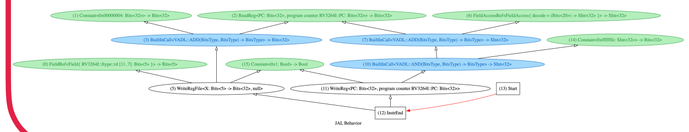

OpenVADL

```
instruction set architecture RV32I = {

 instruction JAL : Jtype =
     {
         let retaddr = PC.next in {
             PC    := (PC + immS) & (-(2 as SIntR))
             X(rd) := retaddr
         }
     }
   …

 pseudo instruction J( offset : Bits<20> ) =
     {
         JAL{ rd = 0 as Bits5, imm = offset }
     }

   …

  def : Pat<(br bb:$imm),
        (J RV32I_Jtype_immAsLabel:$imm)>;


}
```


OpenVADL

Recap

Repeat for all supported "classes" (Branch, Jump, …)



Check properties of dataflow

Found a match

Found no match

Pattern is known to have certain form

Check for Red Flags? (Multiple Side Effects, PC)

Ok

Recursive Lowering Method

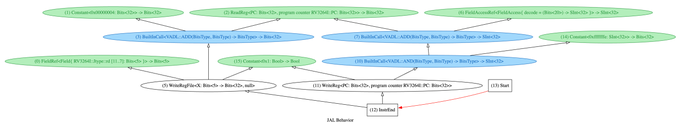
Not ok

No Pattern

OpenVADL

Recap

Repeat for all supported
"classes" (Branch, Jump, …)



Check properties
of dataflow

Found a match

Found no match

Pattern is known
to have certain
form

Check for
Red Flags?
(Multiple Side Effects, PC)

Ok

Recursive
Lowering
Method

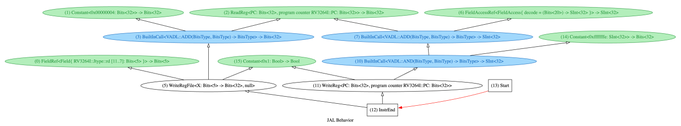
Not ok

No Pattern

OpenVADL

Recap

Repeat for all supported
"classes" (Branch, Jump, …)



Check properties
of dataflow

Found a match

Found no match

Pattern is known
to have certain
form

Check for
Red Flags?
(Multiple Side Effects, PC)

Ok

Recursive
Lowering
Method

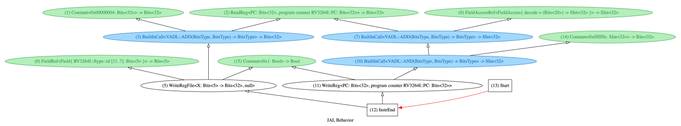
Not ok

No Pattern

OpenVADL

Recap

Repeat for all supported
"classes" (Branch, Jump, …)



Check properties
of dataflow

Found a match

Found no match

Pattern is known
to have certain
form

Check for
Red Flags?
(Multiple Side Effects, PC)

Ok

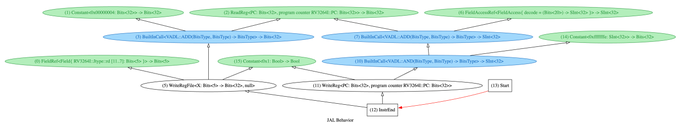
Recursive
Lowering
Method

Not ok

No Pattern

OpenVADL

Recap



Repeat for all supported
"classes" (Branch, Jump, …)

Check properties
of dataflow

Found a match

Found no match

Pattern is known
to have certain
form

Check for
Red Flags?
(Multiple Side Effects, PC)

Ok

Recursive
Lowering
Method

Not ok

No Pattern

OpenVADL

Extras

- Edge Case Pruning

```
instruction ADDDIV : Rtype =
     X(rd) :=
       if X(rs2) = 0 then
         0 as Regs
       else
         (X(rs1) + X(rs2)) / X(rs2)
```


OpenVADL

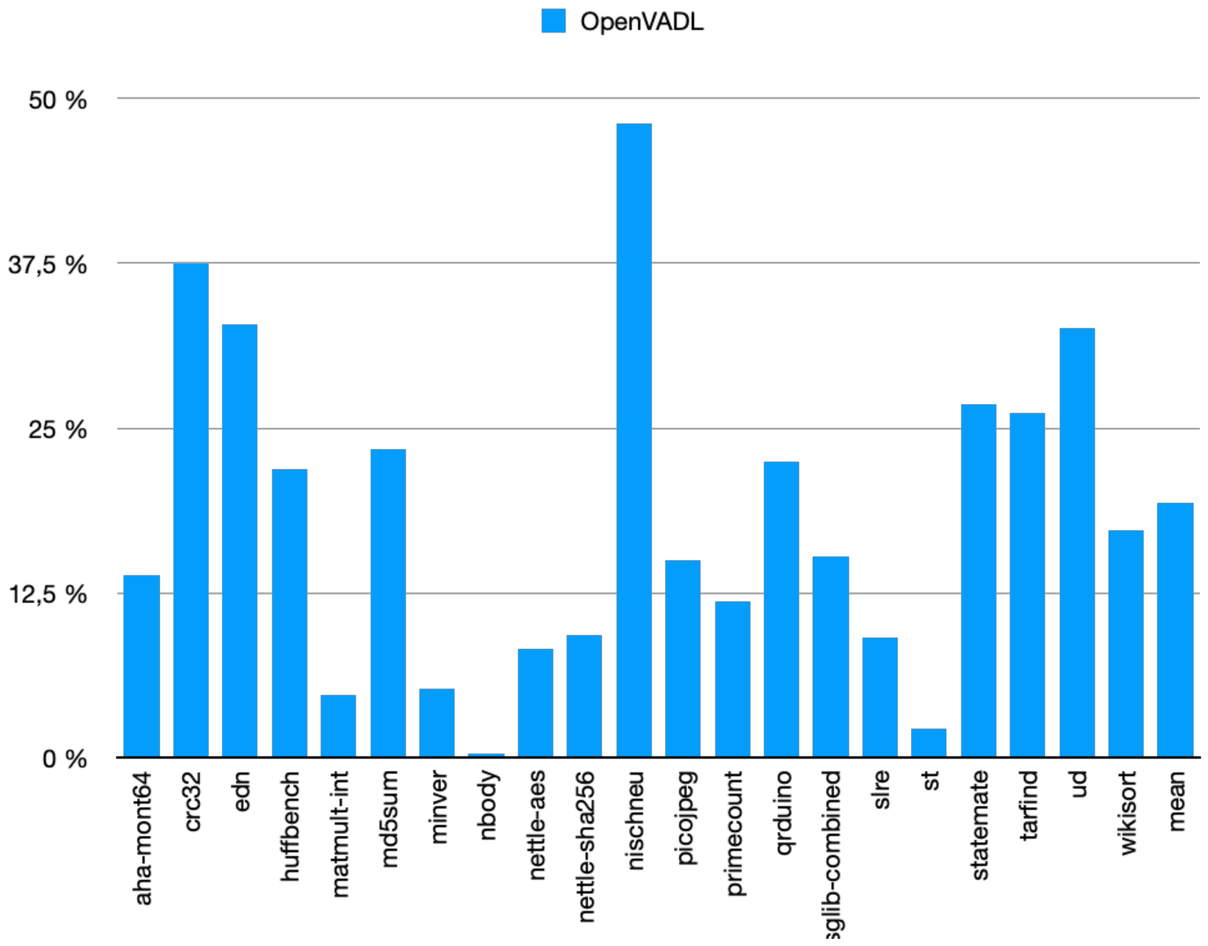
Extras

- Edge Case Pruning

```
instruction ADDDIV : Rtype =
     X(rd) :=
       if X(rs2) = 0 then
         0 as Regs
       else
         (X(rs1) + X(rs2)) / X(rs2)
```


OpenVADL

Performance



Number Of Instructions RV32im -O3 relative to Upstream (lower better)
mean: 19.4%

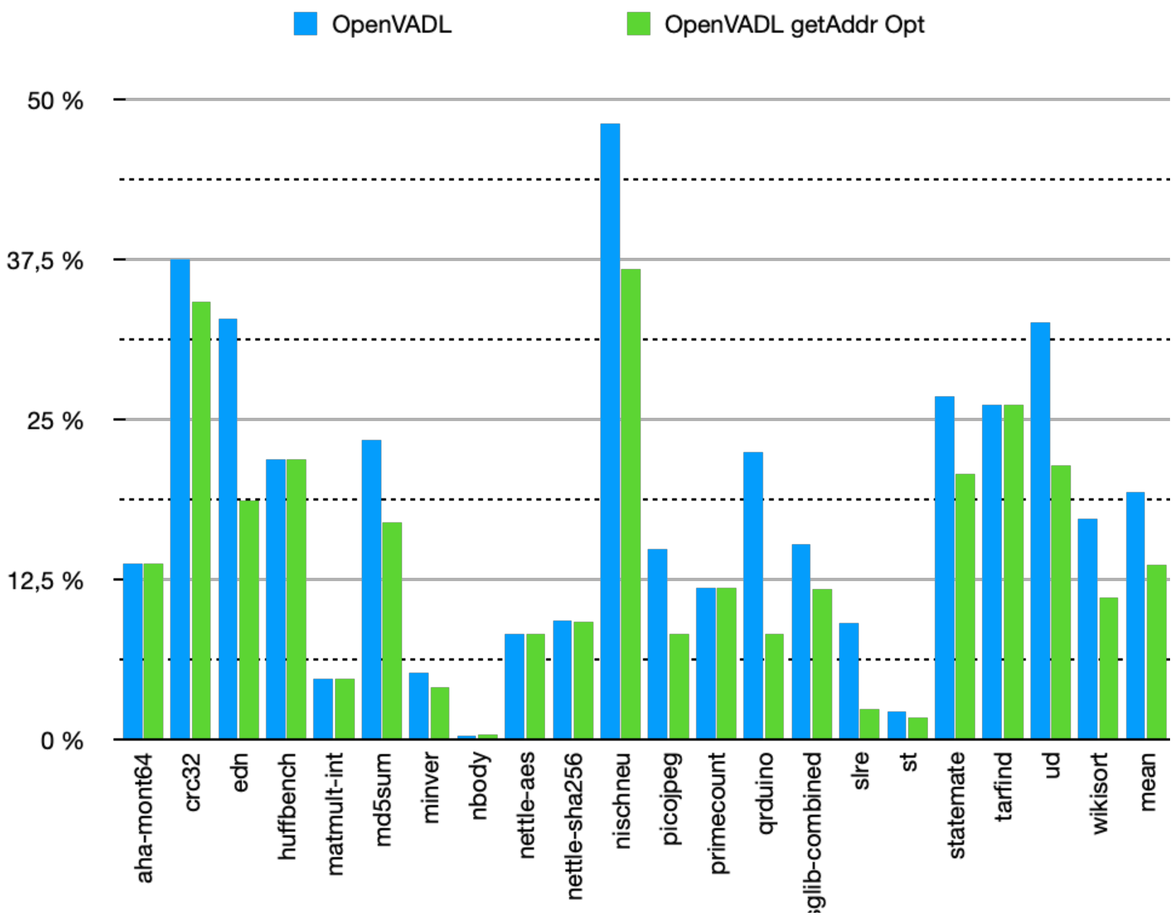
OpenVADL

Performance

```
// ISelLowering.cpp when lowering JumpTable, GlobalAddress, BlockAddress and ConstantPool
case CodeModel::Small:
  {
      SDValue AddrHi = getTargetNode(N, DL, Ty, DAG, rv32imBaseInfo::MO_RV3264I_hi_Itype_imm);
      SDValue AddrLo = getTargetNode(N, DL, Ty, DAG, rv32imBaseInfo::MO_RV3264I_lo_Itype_imm);
      SDValue MNHi = DAG.getNode(rv32imISD::HI, DL, Ty, AddrHi);
      return DAG.getNode(rv32imISD::ADD_LO, DL, Ty, MNHi, AddrLo);
      /*
      SDValue Addr = getTargetNode(N, DL, Ty, DAG, 0);
      return SDValue(DAG.getMachineNode(rv32im::nonPicLA, DL, Ty, Addr), 0);
      */
  }
```


OpenVADL

Performance



Number Of Instructions RV32im -O3 relative to Upstream (lower better)
mean: 13.6%

OpenVADL

Performance

```
constant sequence( rd : Bits<5>, val : SInt<32> ) =
{
   LUI { rd = rd, imm = hi( val ) }
   ADDI { rd = rd, rs1 = rd, imm = lo( val ) }
}

constant sequence( rd : Bits<5>, imm : SInt<12> ) =
{
   ADDI{ rd = rd, rs1 = 0, imm = imm }
}
```

```
// Called in DAGToDAGISel to materialize constants
InstSeq generateInstSeqImpl(int64_t Val, rv32imMatInt::InstSeq &Res ) {


    if(Val >= -2048 && Val <= 2047) {
      Res.emplace_back(rv32im::constMat1, Val);
      return Res;
    }
    if(Val >= -2147483648 && Val <= 2147483647) {
      Res.emplace_back(rv32im::constMat0, Val);
      return Res;
    }

    llvm_unreachable("not supported immediate");
  }
```


OpenVADL

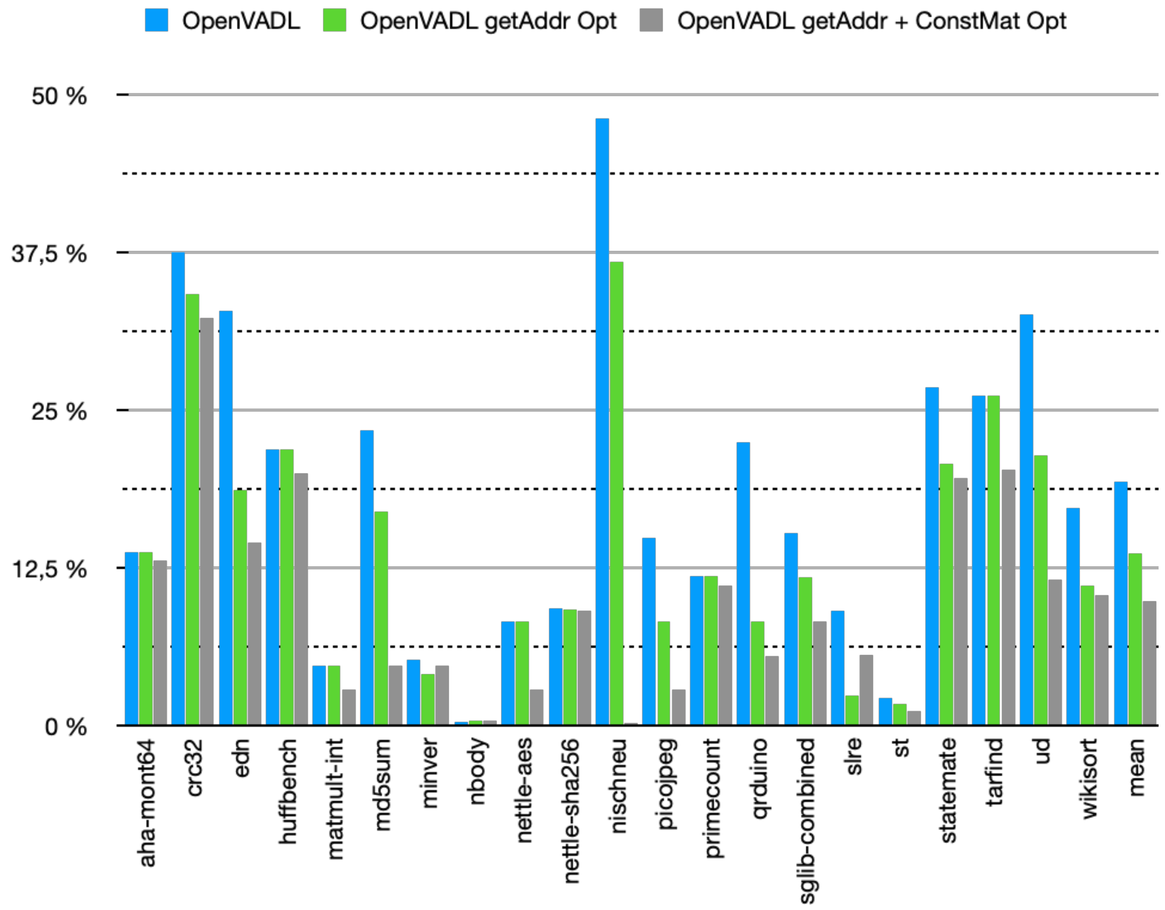
Performance

```
constant sequence( rd : Bits<5>, val : SInt<32> ) =
{
   LUI { rd = rd, imm = hi( val ) }
   ADDI { rd = rd, rs1 = rd, imm = lo( val ) }
}

constant sequence( rd : Bits<5>, imm : SInt<12> ) =
{
   ADDI{ rd = rd, rs1 = 0, imm = imm }
}


// Called in DAGToDAGISel to materialize constants
InstSeq generateInstSeqImpl(int64_t Val, rv32imMatInt::InstSeq &Res ) {


    if(Val >= -2048 && Val <= 2047) {
      Res.emplace_back(rv32im::constMat1, Val);
      return Res;
    }
    if(Val >= -2147483648 && Val <= 2147483647) {
      Res.emplace_back(rv32im::constMat0, Val);
      return Res;
    }

    llvm_unreachable("not supported immediate");
  }
```

```
// Do it like Upstream
InstSeq generateInstSeqImpl(int64_t Val, rv32imMatInt::InstSeq &Res ) {
    if (isInt<32>(Val)) {
        // Depending on the active bits in the immediate Value v, the following
        // instruction sequences are emitted:
        //
        // v == 0                          : ADDI
        // v[0,12) != 0 && v[12,32) == 0 : ADDI
        // v[0,12) == 0 && v[12,32) != 0 : LUI
        // v[0,32) != 0                    : LUI+ADDI(W)
        auto Hi20 = RV3264I_Utype_immUp_encoding(2047 + 1 + Val);
        int64_t Lo12 = SignExtend64<12>(Val);

        if (Hi20)
          Res.emplace_back(rv32im::LUI, Hi20);

        if (Lo12 || Hi20 == 0) {
          unsigned AddiOpc = rv32im::ADDI;
          Res.emplace_back(AddiOpc, Lo12);
        }
        return Res;
      }

      int64_t Lo12 = SignExtend64<12>(Val);
      Val = (uint64_t)Val - (uint64_t)Lo12;

      int ShiftAmount = 0;
      bool Unsigned = false;

// continued
```


OpenVADL

Performance



Number Of Instructions RV32im -O3 relative to Upstream (lower better)
mean: 9.9%

OpenVADL

Future work

- RISCV64im
- AArch64
- WIP Bachelor's Thesis for
  - Assembly Parser / Linker
  - Instruction Scheduling
- Anything with the word "ABI" in it
  - ELF
  - Calling Conventions
- Floating Point or Vector support missing

https://openvadl.org/
https://github.com/OpenVADL/open-vadl

OpenVADL

Backup

```
srand:                                  # @srand
# %bb.0:                                # %entry
        ADDI sp,sp,-16
        SW ra,12(sp)                            # 4-byte Folded Spill
        SW fp,8(sp)                             # 4-byte Folded Spill
        ADDI fp,sp,16
        LW fp,8(sp)                             # 4-byte Folded Reload
        LW ra,12(sp)                            # 4-byte Folded Reload
        ADDI sp,sp,16
        JALR zero,0(ra)
.Lfunc_end0:
        .size   srand, .Lfunc_end0-srand
                                        # -- End function
        .globl  x                       # -- Begin function x
        .type   x,@function
x:                                      # @x
# %bb.0:                                # %entry
        ADDI sp,sp,-16
        SW ra,12(sp)                            # 4-byte Folded Spill
        SW fp,8(sp)                             # 4-byte Folded Spill
        ADDI fp,sp,16
.LBB1_1:                                # %while.body
                                        # =>This Inner Loop Header: Depth=1
        JAL zero,.LBB1_1
```

OpenVADL

```
srand:                                  # @srand
# %bb.0:                                # %entry
        ret
.Lfunc_end0:
        .size   srand, .Lfunc_end0-srand
                                        # -- End function
        .globl  x                       # -- Begin function x
        .p2align        1
        .type   x,@function
x:                                      # @x
# %bb.0:                                # %entry
.LBB1_1:                                # %while.body

        j       .LBB1_1
.Lfunc_end1:
```

Upstream

OpenVADL

Backup

```
srand:                                  # @srand
# %bb.0:                                # %entry
        ADDI sp,sp,-16
        SW ra,12(sp)                            # 4-byte Folded Spill
        SW fp,8(sp)                             # 4-byte Folded Spill
        ADDI fp,sp,16
        LW fp,8(sp)                             # 4-byte Folded Reload
        LW ra,12(sp)                            # 4-byte Folded Reload
        ADDI sp,sp,16
        JALR zero,0(ra)
.Lfunc_end0:
        .size   srand, .Lfunc_end0-srand
                                        # -- End function
        .globl  x                       # -- Begin function x
        .type   x,@function
x:                                      # @x
# %bb.0:                                # %entry
        ADDI sp,sp,-16
        SW ra,12(sp)                            # 4-byte Folded Spill
        SW fp,8(sp)                             # 4-byte Folded Spill
        ADDI fp,sp,16
.LBB1_1:                                # %while.body
                                        # =>This Inner Loop Header: Depth=1
        JAL zero,.LBB1_1
```

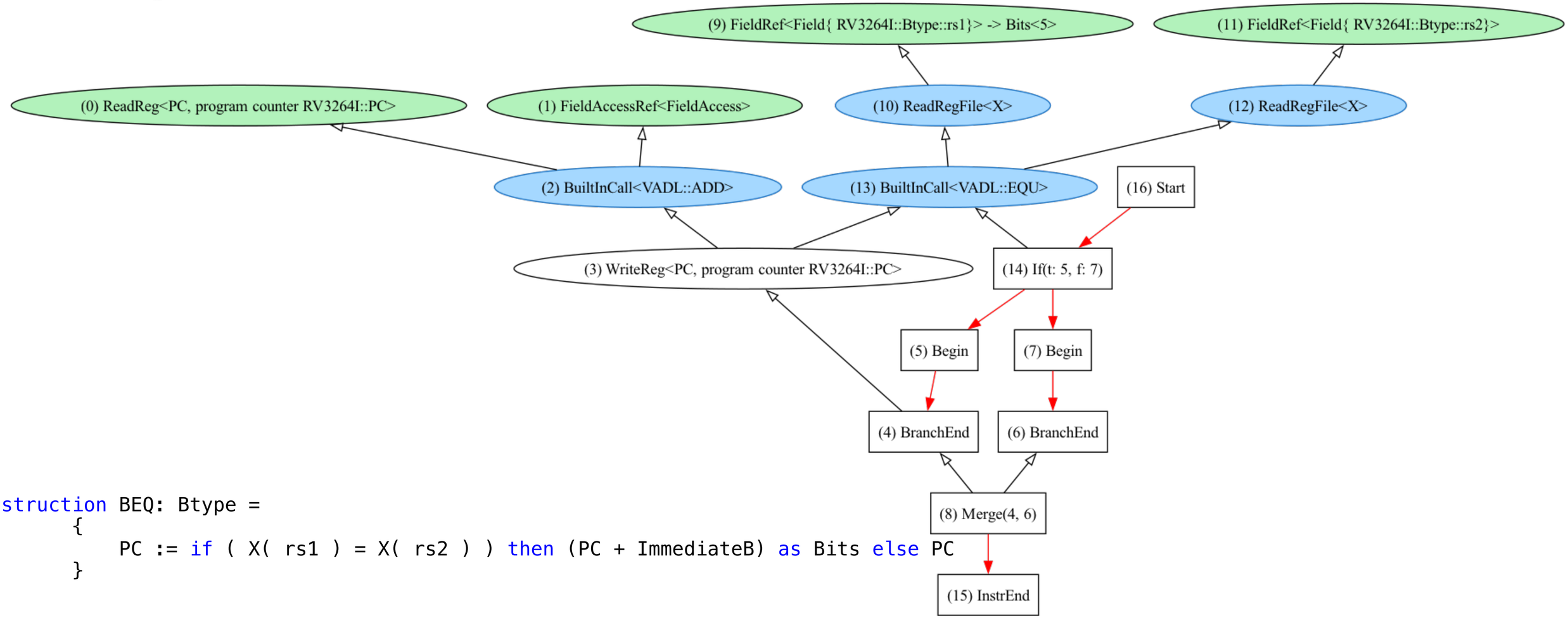
OpenVADL

```
srand:                                  # @srand
# %bb.0:                                # %entry
        ret
.Lfunc_end0:
        .size   srand, .Lfunc_end0-srand
                                        # -- End function
        .globl  x                       # -- Begin function x
        .p2align        1
        .type   x,@function
x:                                      # @x
# %bb.0:                                # %entry
.LBB1_1:                                # %while.body

        j       .LBB1_1
.Lfunc_end1:
```

Upstream

OpenVADL

BEQ

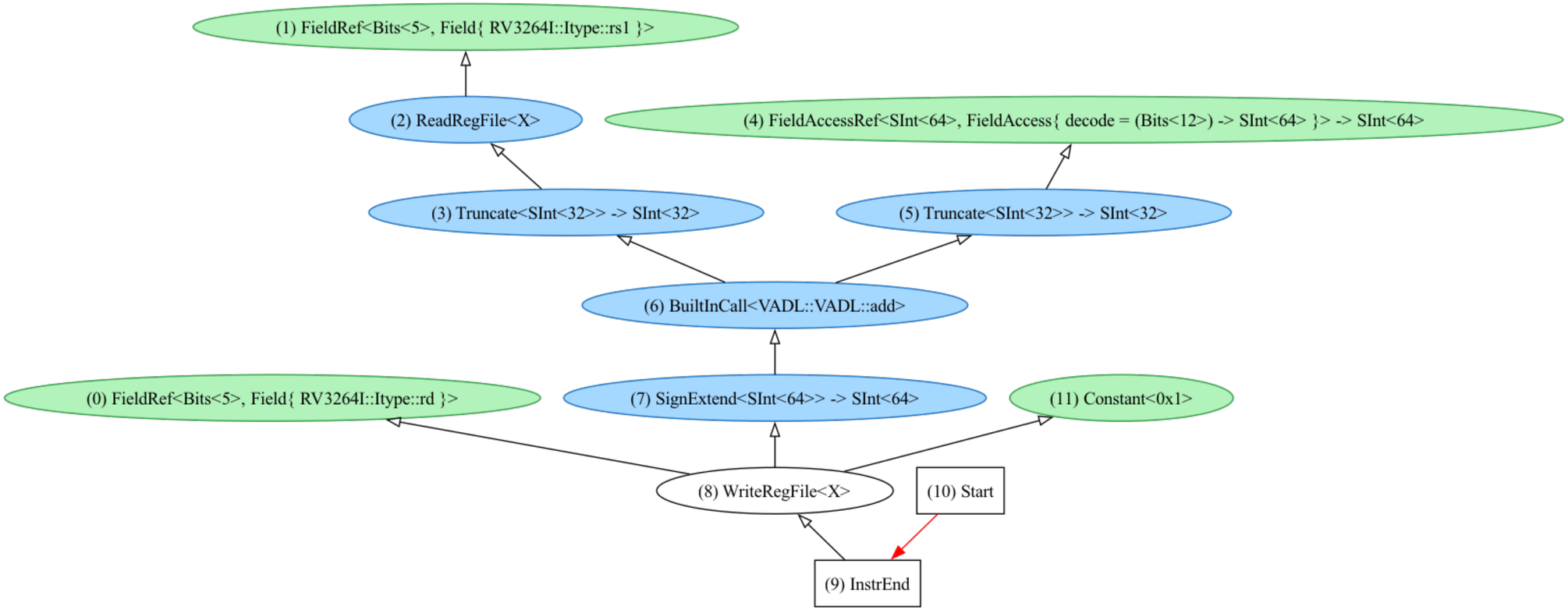


BEQ Behavior

```
instruction BEQ: Btype =
        {
            PC := if ( X( rs1 ) = X( rs2 ) ) then (PC + ImmediateB) as Bits else PC
        }
```


OpenVADL

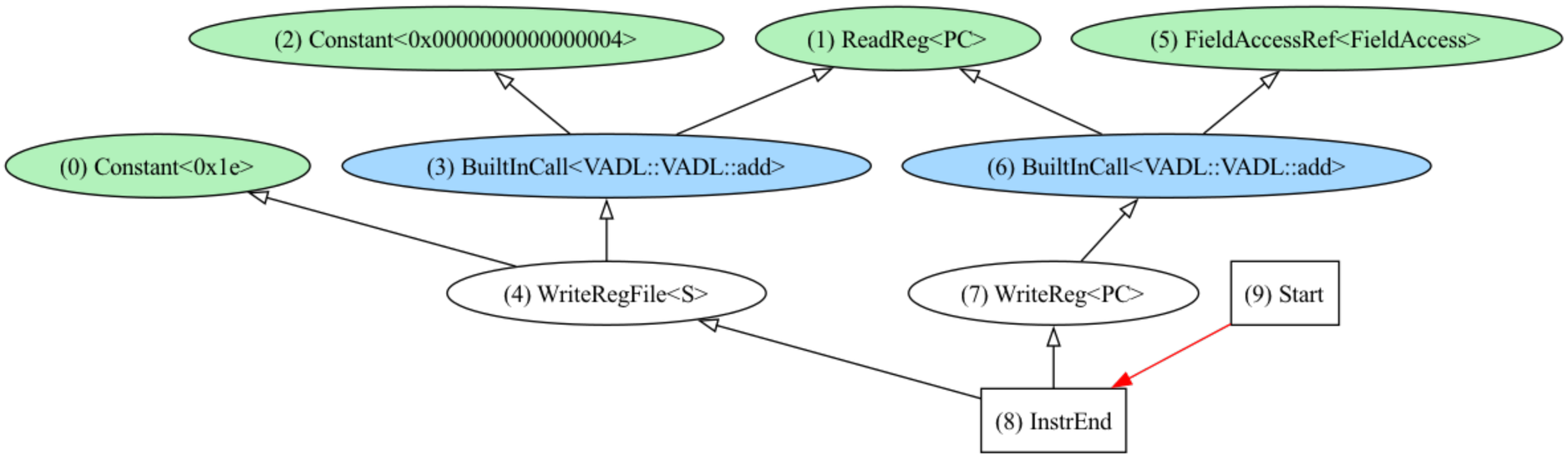
ADDIW



ADDIW Behavior

OpenVADL

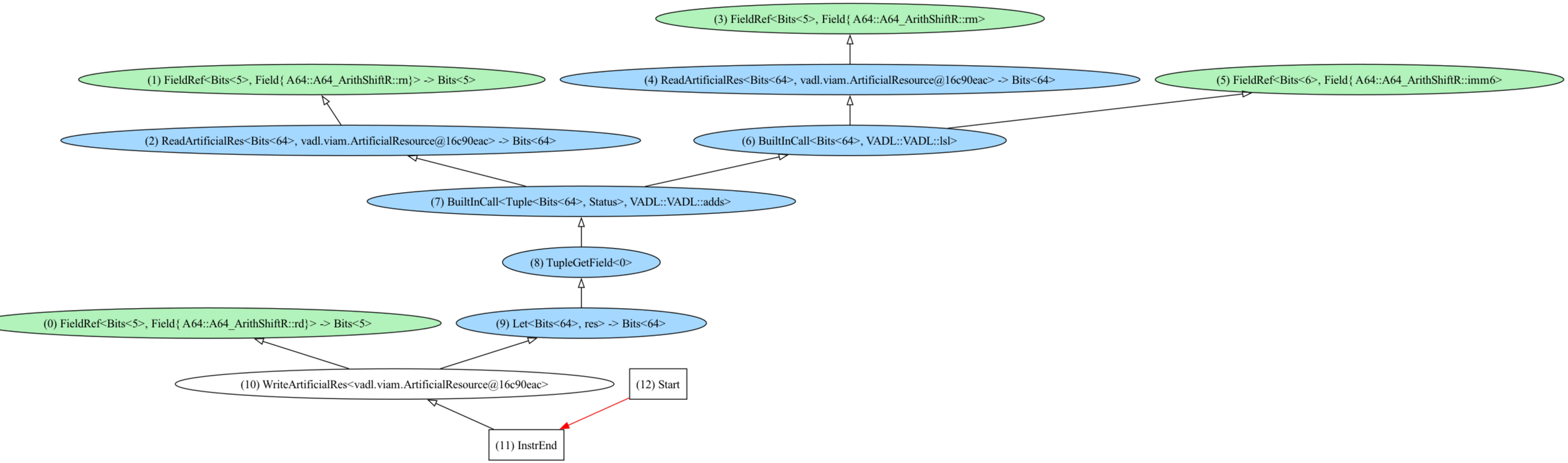
BL in AArch64



BL Behavior

OpenVADL

ADD64LSL Immediate



ADD64LSL Behavior

OpenVADL