HARDWARE LOOPS IN THE IPU BACKEND

Janek van Oirschot

GRAPHCORE

INTELLIGENCE PROCESSING UNIT (IPU)

- Chip with many cores (tiles) with inter-core communication
- Each tile has its own memory
- Each tile has multiple threads
- 2 execution pipelines (can run in parallel)
  - Main (for integer instructions + registers)
  - Aux (for float instructions + registers)
- Granularity of this talk: a single tile

HARDWARE LOOPS?

Currently, the LLVM IPU backend targets 2 instructions as hardware loops

- rpt (repeat) instruction
- brnzdec (branch if non-zero, decrement) instruction

RPT INSTRUCTION

```
rpt $reg, zimm8
rpt zimmX, zimm8
```

Op #1: Trip count of the loop
Op #2: Number of loop body instruction bundle

```
void foo (int *out, int *in, unsigned size) {
  for (int i = 0; i < size; ++i) {
    *out += in[i];
  }
}
```

```
foo:
  ld32 $m3, $m0, $m15, 0
  {
    rpt $m2, 2
    fnop
  }
  {
    ld32step $m4, $m15, $m1+=, 1
    fnop
  }
  {
    add $m3, $m3, $m4
    fnop
  }
  {
    st32 $m3, $m0, $m15, 0
    fnop
  }
  br $m10
```

BRNZDEC INSTRUCTION

```
void foo (int *out, int *in, unsigned size) {
  for (int i = 0; i < size; ++i) {
    *out += in[i];
  }
}
```

```
brnzdec $reg, zimm19
```

Op #1: Value to be tested and decremented
Op #2: Target address of branch

```
foo:
  brz $m2, .LBB0_3
  add $m2, $m2, -1
  ld32 $m3, $m0, $m15, 0
.LBB0_2:
  ld32step $m4, $m15, $m1+=, 1
  add $m3, $m3, $m4
  st32 $m3, $m0, $m15, 0
  brnzdec $m2, .LBB0_2
.LBB0_3:
  br $m10
```

RPT VS BRNZDEC

`rpt`

- Trip count depends on IPU architecture
- Requires instructions in body to be bundled
- Maximum number of bundles in loop body is 256 bundles (constrained by operand bits)
- Implicitly skips over loop body if trip count == 0
- Control (e.g., branch) and System (e.g., trap) instructions **cannot** be executed in the body
  - -> no loop nests within

`brnzdec`

- Trip count limited to 32 bits available for registers
- Maximum number of instructions constrained by available bits for instruction encoding: 19 bits == 0x7FFFF left shifted twice for 4 byte alignment: 0x1FFFFC

RPT

`rpt`

- Trip count depends on IPU architecture
- Requires instructions in body to be bundled
- Maximum number of bundles in loop body is 256 bundles (constrained by operand bits)
- Implicitly skips over loop body if trip count == 0
- Control (e.g., branch) and System (e.g., trap) instructions **cannot** be executed in the body
  - -> no loop nests within

```
foo:
  ld32 $m3, $m0, $m15, 0
  {
    rpt $m2, 2
    fnop
  }
  {
    ld32step $m4, $m15, $m1+=, 1
    fnop
  }
  {
    add $m3, $m3, $m4
    fnop
  }
  {
    st32 $m3, $m0, $m15, 0
    fnop
  }
  br $m10
```

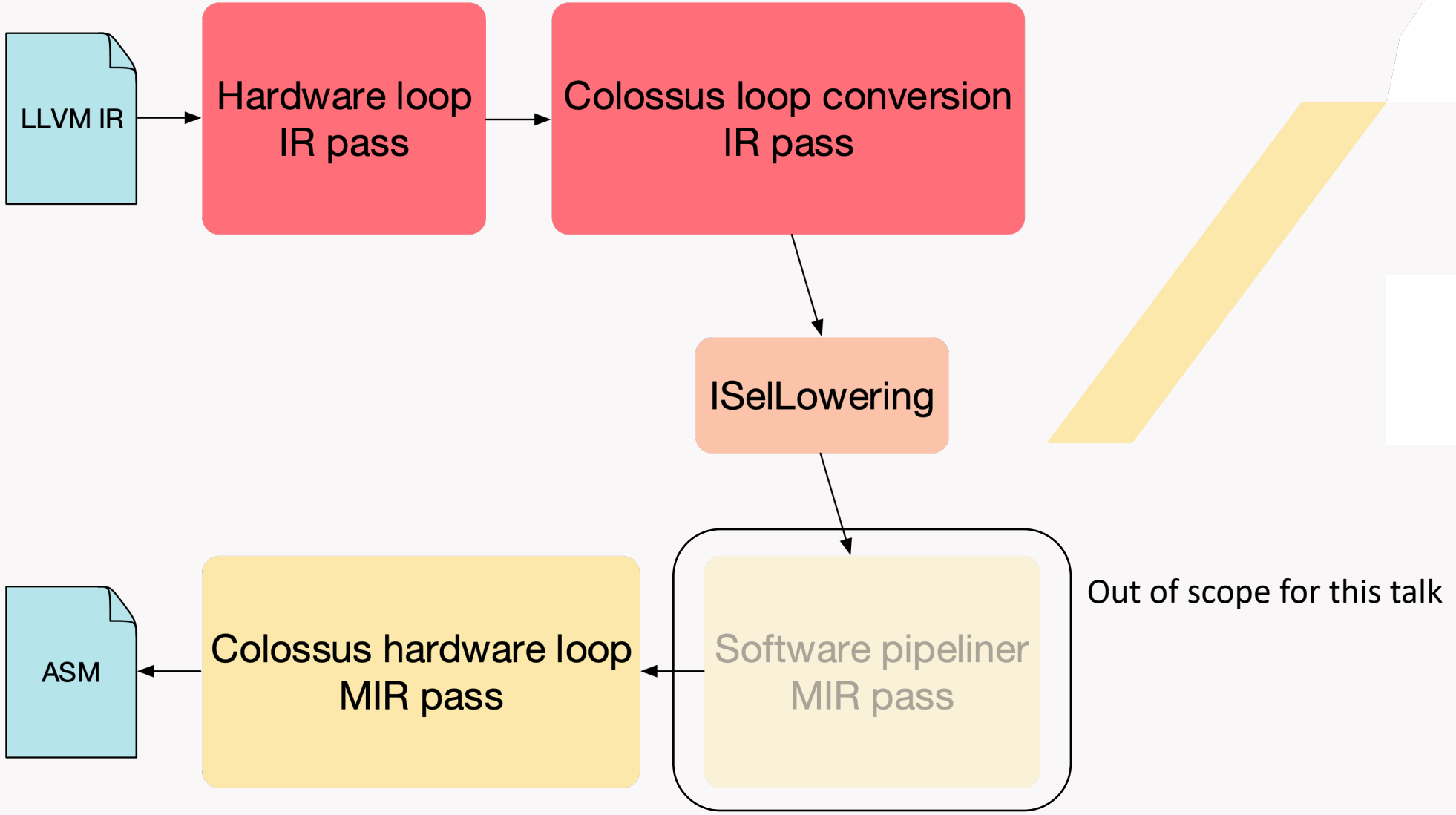
BRNZDEC

```
foo:
  brz $m2, .LBB0_3
  add $m2, $m2, -1
  ld32 $m3, $m0, $m15, 0
.LBB0_2:
  ld32step $m4, $m15, $m1+=, 1
  add $m3, $m3, $m4
  st32 $m3, $m0, $m15, 0
  brnzdec $m2, .LBB0_2
.LBB0_3:
  br $m10
```

`brnzdec`

- Trip count limited to 32 bits available for registers
- Maximum number of instructions constrained by available bits for instruction encoding: 19 bits == 0x7FFFF left shifted twice for 4 byte alignment: 0x1FFFFC

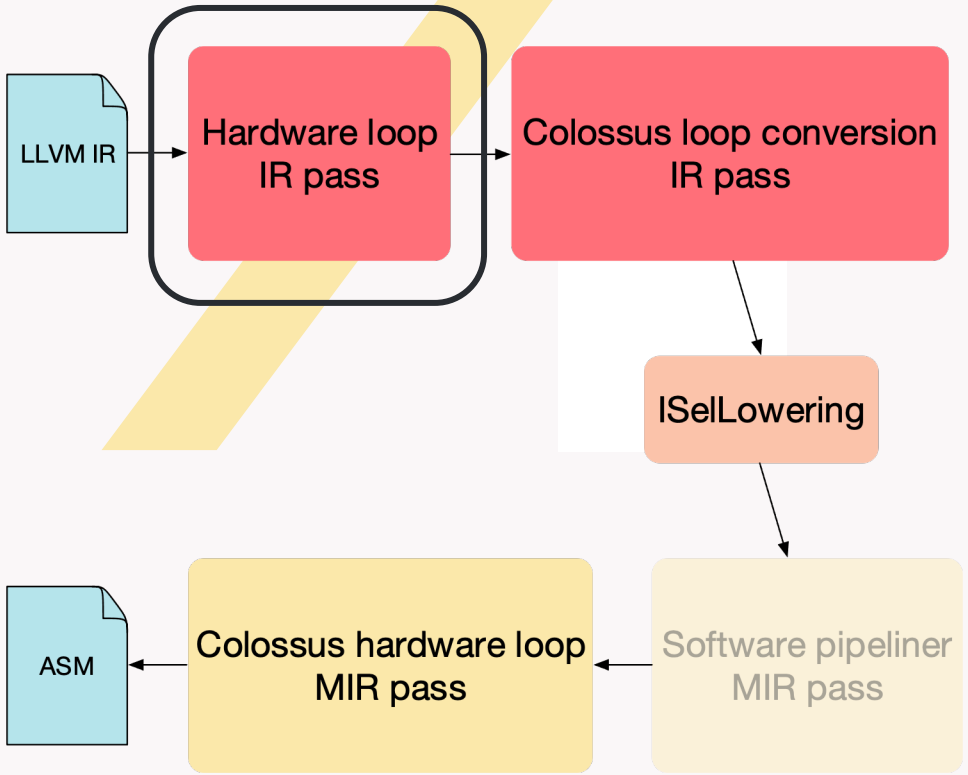
HIGH LEVEL VIEW



HARDWARE LOOP IR PASS

```
bool isHardwareLoopProfitable(Loop *L,
                              ScalarEvolution &SE,
                              AssumptionCache &AC,
                              TargetLibraryInfo *LibInfo,
                              HardwareLoopInfo &HWLoopInfo);
```

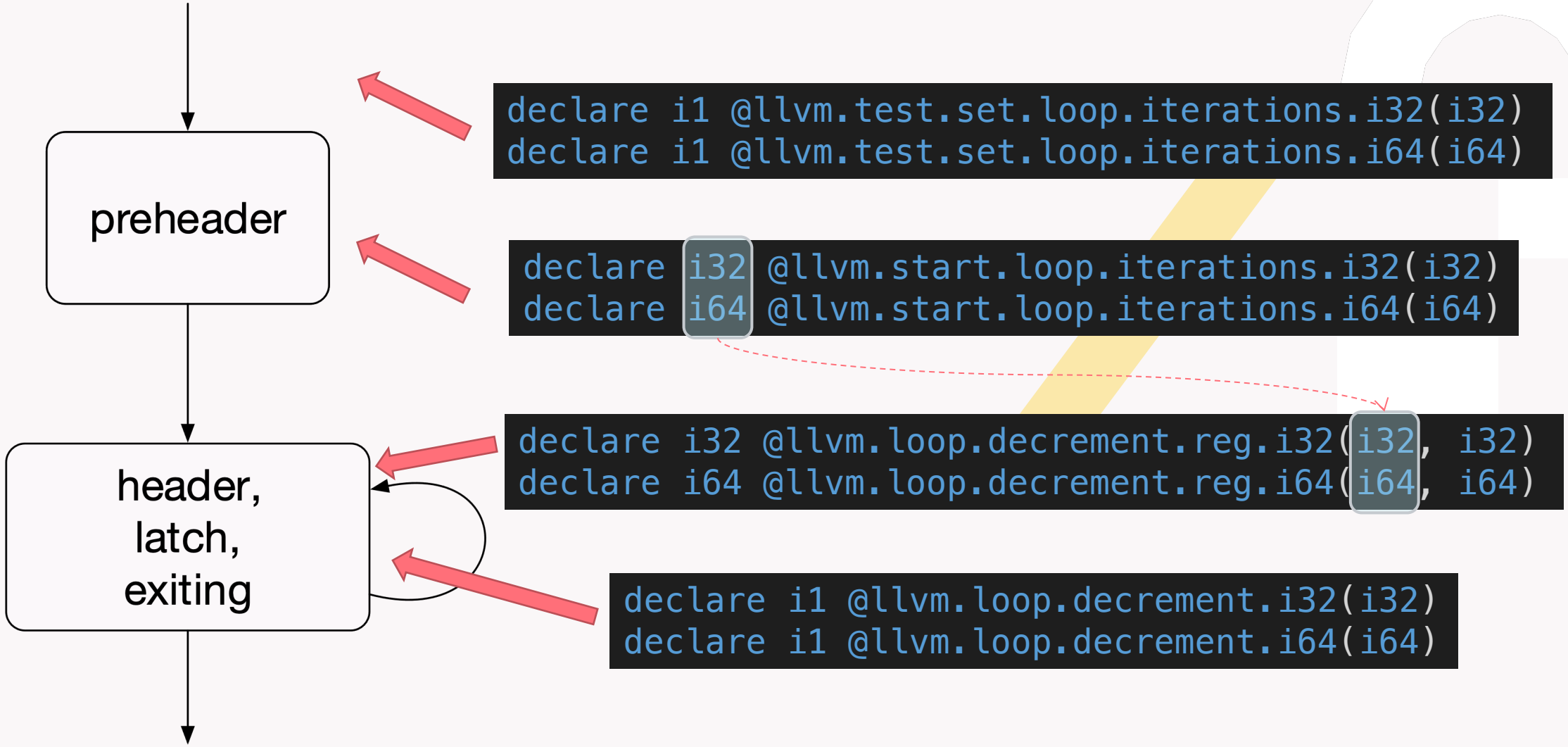
- Used to be an IPU custom pass but now we use the LLVM target independent Hardware Loop IR pass
- Run as part of CodeGenPrepare passes, as opposed to addPreISel passes of other targets, any later and the assumptions are removed from IR
- IsHardwareLoopProfitable TargetTransformInfo hook
  - Processing for assumptions
  - Emit different hardware loop settings depending on whether we want a rpt or brnzdec



HARDWARE LOOP IR PASS

```
auto emitRptHwLoop = [&HWLoopInfo, &i32, &i64](unsigned bitwidth) {
  HWLoopInfo.IsNestingLegal = true;
  HWLoopInfo.CounterInReg = false;
  HWLoopInfo.CountType = bitwidth == 64 ? i64 : i32;
  HWLoopInfo.LoopDecrement = ConstantInt::get(HWLoopInfo.CountType, 1);
  HWLoopInfo.PerformEntryTest = true;
};

auto emitBrnzdecHwLoop = [&HWLoopInfo,&i32,&i64](unsigned bitwidth) {
  HWLoopInfo.IsNestingLegal = true;
  HWLoopInfo.CounterInReg = true;
  HWLoopInfo.CountType = bitwidth == 64 ? i64 : i32;
  HWLoopInfo.LoopDecrement = ConstantInt::get(HWLoopInfo.CountType, 1);
};
```

HARDWARE LOOP IR PASS

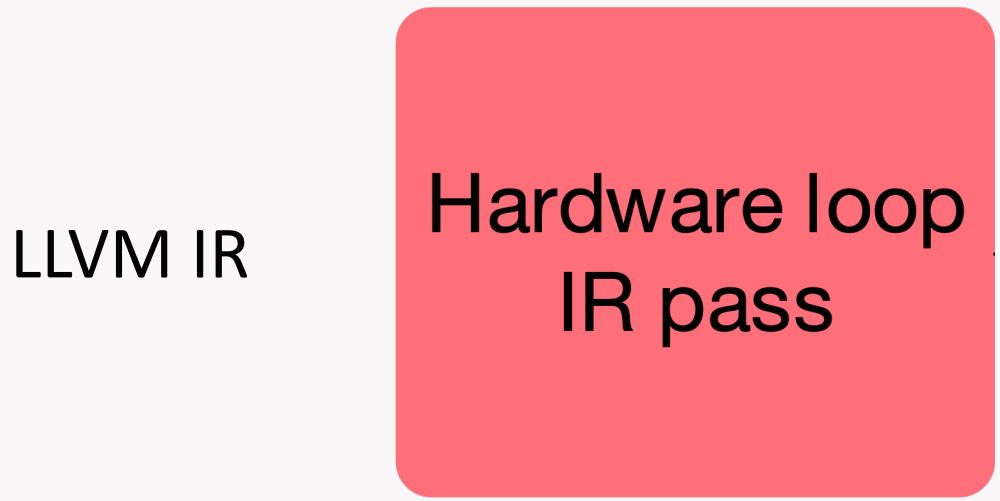


HARDWARE LOOP IR PASS

- rpt is prioritized over brnzdec
- The constraints checked at this stage are trip count fit, loop nest level, and some trivial architecture constrains

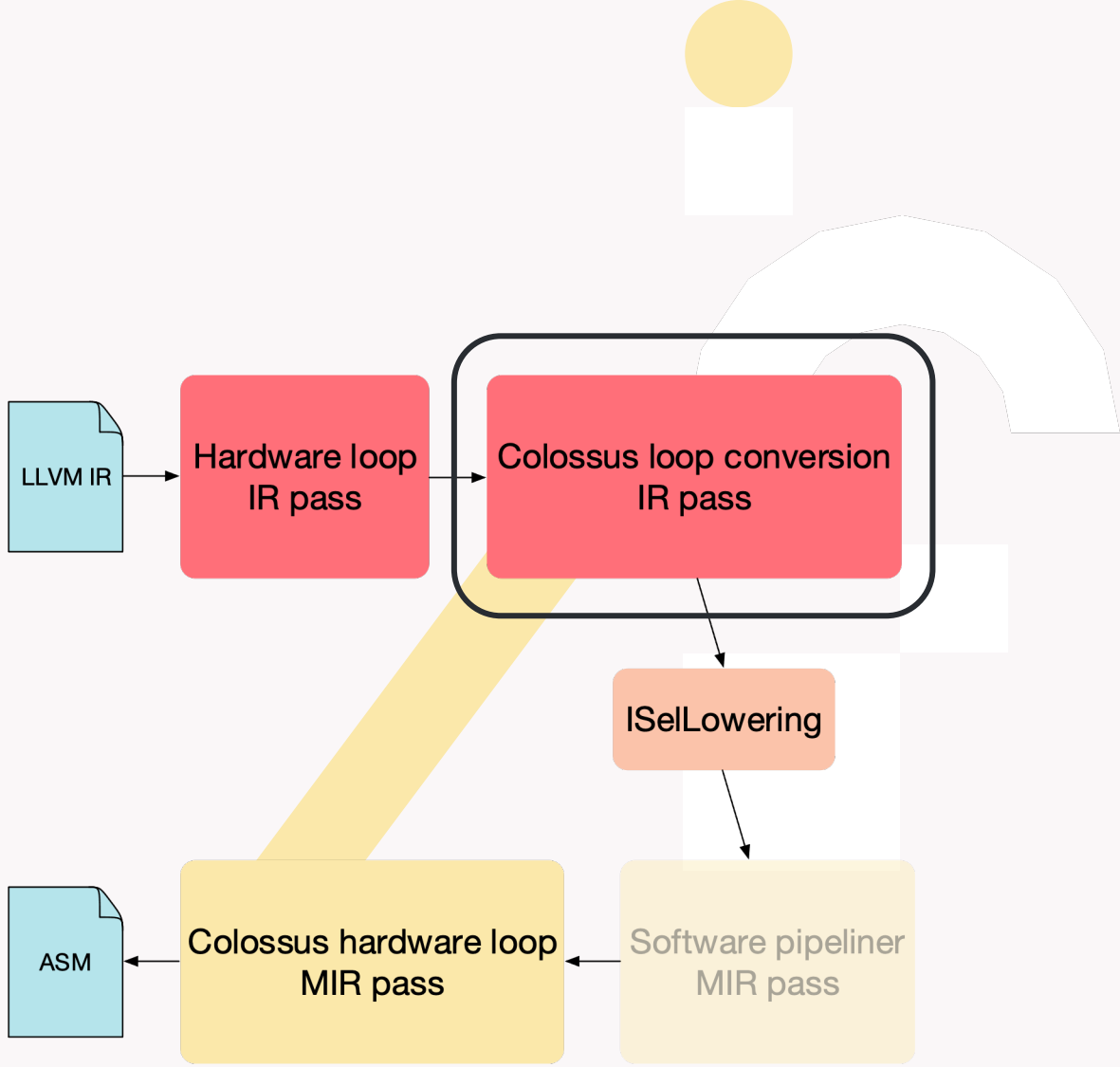
```
if (!hasSubLoop && (isWithinRptRange || isKBValid) && ...) {
  emitRptHwLoop(bitwidth);
  return true;
}

if (isWithinBrnzdecRange || isKBValid_Brnzdec) {
  emitBrnzdecHwLoop(bitwidth);
  return true;
}
```

HARDWARE LOOP IR PASS



LOOP CONVERSION IR PASS

- Run as part of addPreISel passes
- Mechanical pass that converts LLVM target independent hardware loop intrinsics in IPU specific ones
- The IPU specific intrinsics are from the time when the hardware loop IR pass was target specific



LOOP CONVERSION IR PASS

```
define void @foo(i32* %out, i32* %in, i32 %size) {
entry:
  %0 = call i1 @llvm.test.set.loop.iterations.i32(i32 %size)
  br i1 %0, label %preheader, label %exit
preheader:
  %pre = load i32, i32* %out, align 4
  br label %body
body:
  %lsr.iv = phi i32* [ %in, %preheader ], [ %gep, %body ]
  %1 = phi i32 [ %add, %body ], [ %pre, %preheader ]
  %2 = load i32, i32* %lsr.iv, align 4
  %add = add nsw i32 %1, %2
  store i32 %add, i32* %out, align 4
  %gep = getelementptr i32, i32* %lsr.iv, i32 1
  %3 = call i1 @llvm.loop.decrement.i32(i32 1)
  br i1 %3, label %body, label %exit
exit:
  ret void
}
```

```
define void @foo(i32* %out, i32* %in, i32 %size) {
entry:
  %guard = call i32 @llvm.colossus.cloop.guard(i32 %size, i32 1)
  %guard.tr = trunc i32 %guard to i1
  br i1 %guard.tr, label %preheader, label %exit
preheader:
  %pre = load i32, i32* %out, align 4
  %begin = call i32 @llvm.colossus.cloop.begin(i32 %size, i32 1)
  br label %body
body:
  %lsr.iv = phi i32* [ %in, %preheader ], [ %gep, %body ]
  %0 = phi i32 [ %add, %body ], [ %pre, %preheader ]
  %1 = phi i32 [ %begin, %preheader ], [ %end.iv, %body ]
  %2 = load i32, i32* %lsr.iv, align 4
  %add = add nsw i32 %0, %2
  store i32 %add, i32* %out, align 4
  %gep = getelementptr i32, i32* %lsr.iv, i32 1
  %end = call { i32, i32 } @llvm.colossus.cloop.end(i32 %1, i32 1)
  %end.iv = extractvalue { i32, i32 } %end, 0
  %end.cc = extractvalue { i32, i32 } %end, 1
  %end.cc.tr = trunc i32 %end.cc to i1
  br i1 %end.cc.tr, label %body, label %exit
exit:
  ret void
```

LOOP CONVERSION IR PASS

```
define void @foo(i32* %out, i32* %in, i32 %size) {
entry:
  %0 = call i1 @llvm.test.set.loop.iterations.i32(i32 %size)
  br i1 %0, label %preheader, label %exit
preheader:
  %pre = load i32, i32* %out, align 4
  br label %body
body:
  %lsr.iv = phi i32* [ %in, %preheader ], [ %gep, %body ]
  %1 = phi i32 [ %add, %body ], [ %pre, %preheader ]
  %2 = load i32, i32* %lsr.iv, align 4
  %add = add nsw i32 %1, %2
  store i32 %add, i32* %out, align 4
  %gep = getelementptr i32, i32* %lsr.iv, i32 1
  %3 = call i1 @llvm.loop.decrement.i32(i32 1)
  br i1 %3, label %body, label %exit
exit:
  ret void
}
```

```
define void @foo(i32* %out, i32* %in, i32 %size) {
entry:
  %guard = call i32 @llvm.colossus.cloop.guard(i32 %size, i32 1)
  %guard.tr = trunc i32 %guard to i1
  br i1 %guard.tr, label %preheader, label %exit
preheader:
  %pre = load i32, i32* %out, align 4
  %begin = call i32 @llvm.colossus.cloop.begin(i32 %size, i32 1)
  br label %body
body:
  %lsr.iv = phi i32* [ %in, %preheader ], [ %gep, %body ]
  %0 = phi i32 [ %add, %body ], [ %pre, %preheader ]
  %1 = phi i32 [ %begin, %preheader ], [ %end.iv, %body ]
  %2 = load i32, i32* %lsr.iv, align 4
  %add = add nsw i32 %0, %2
  store i32 %add, i32* %out, align 4
  %gep = getelementptr i32, i32* %lsr.iv, i32 1
  %end = call { i32, i32 } @llvm.colossus.cloop.end(i32 %1, i32 1)
  %end.iv = extractvalue { i32, i32 } %end, 0
  %end.cc = extractvalue { i32, i32 } %end, 1
  %end.cc.tr = trunc i32 %end.cc to i1
  br i1 %end.cc.tr, label %body, label %exit
exit:
  ret void
```

LOOP CONVERSION IR PASS

```
define void @foo(i32* %out, i32* %in, i32 %size) {
entry:
  %0 = call i1 @llvm.test.set.loop.iterations.i32(i32 %size)
  br i1 %0, label %preheader, label %exit
preheader:
  %pre = load i32, i32* %out, align 4
  br label %body
body:
  %lsr.iv = phi i32* [ %in, %preheader ], [ %gep, %body ]
  %1 = phi i32 [ %add, %body ], [ %pre, %preheader ]
  %2 = load i32, i32* %lsr.iv, align 4
  %add = add nsw i32 %1, %2
  store i32 %add, i32* %out, align 4
  %gep = getelementptr i32, i32* %lsr.iv, i32 1
  %3 = call i1 @llvm.loop.decrement.i32(i32 1)
  br i1 %3, label %body, label %exit
exit:
  ret void
}
```

```
enum class metadata : uint16_t {
  none = 0,
  tripCountOKForRpt = 1,
};
```

```
define void @foo(i32* %out, i32* %in, i32 %size) {
entry:
  %guard = call i32 @llvm.colossus.cloop.guard(i32 %size, i32 1)
  %guard.tr = trunc i32 %guard to i1
  br i1 %guard.tr, label %preheader, label %exit
preheader:
  %pre = load i32, i32* %out, align 4
  %begin = call i32 @llvm.colossus.cloop.begin(i32 %size, i32 1)
  br label %body
body:
  %lsr.iv = phi i32* [ %in, %preheader ], [ %gep, %body ]
  %0 = phi i32 [ %add, %body ], [ %pre, %preheader ]
  %1 = phi i32 [ %begin, %preheader ], [ %end.iv, %body ]
  %2 = load i32, i32* %lsr.iv, align 4
  %add = add nsw i32 %0, %2
  store i32 %add, i32* %out, align 4
  %gep = getelementptr i32, i32* %lsr.iv, i32 1
  %end = call { i32, i32 } @llvm.colossus.cloop.end(i32 %1, i32 1)
  %end.iv = extractvalue { i32, i32 } %end, 0
  %end.cc = extractvalue { i32, i32 } %end, 1
  %end.cc.tr = trunc i32 %end.cc to i1
  br i1 %end.cc.tr, label %body, label %exit
exit:
  ret void
```

LOOP CONVERSION IR PASS

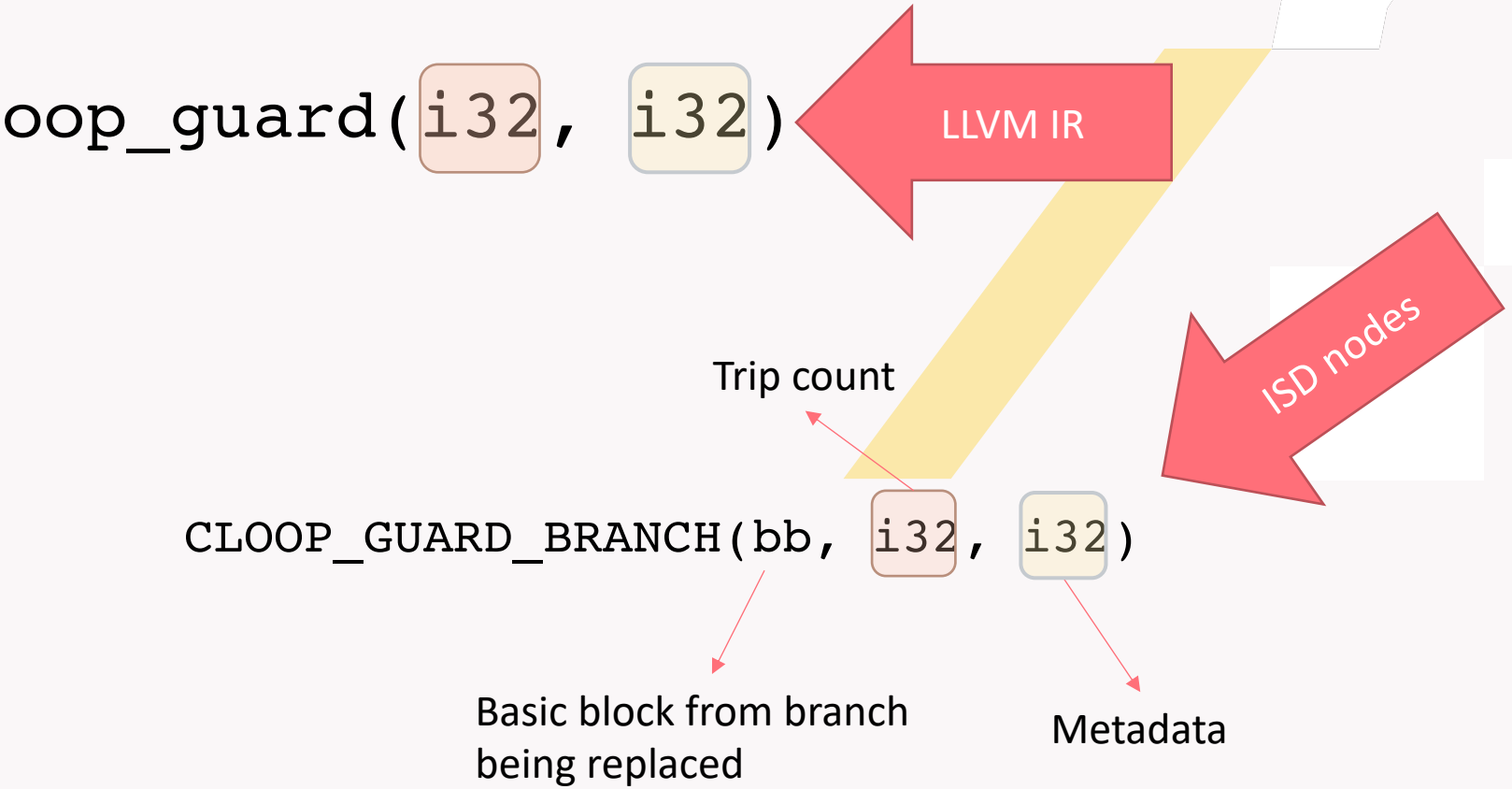
LLVM IR with hardware loop intrinsics

Colossus loop conversion IR pass

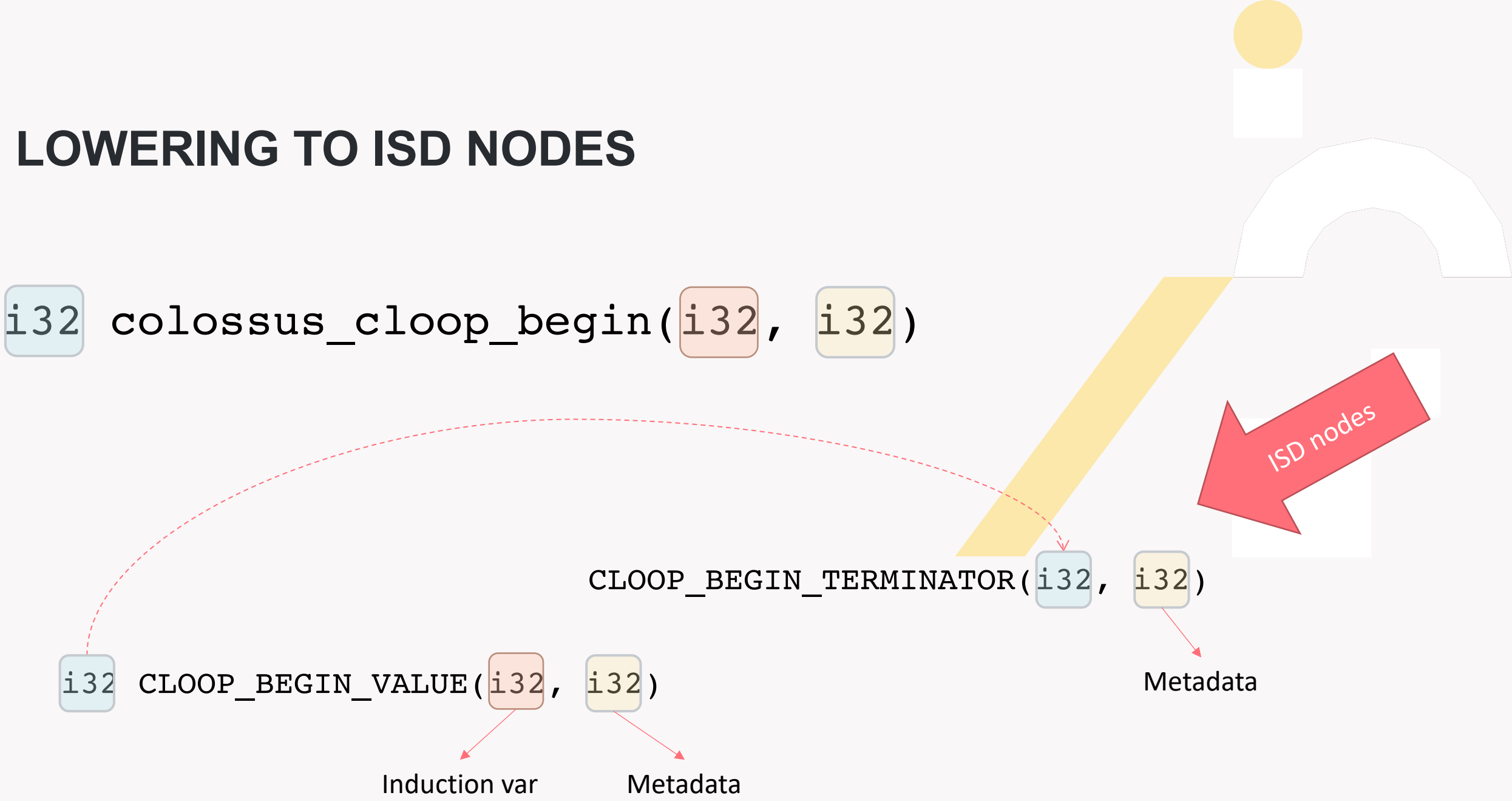
LLVM IR with IPU hardware loop intrinsics

LOWERING TO ISD NODES

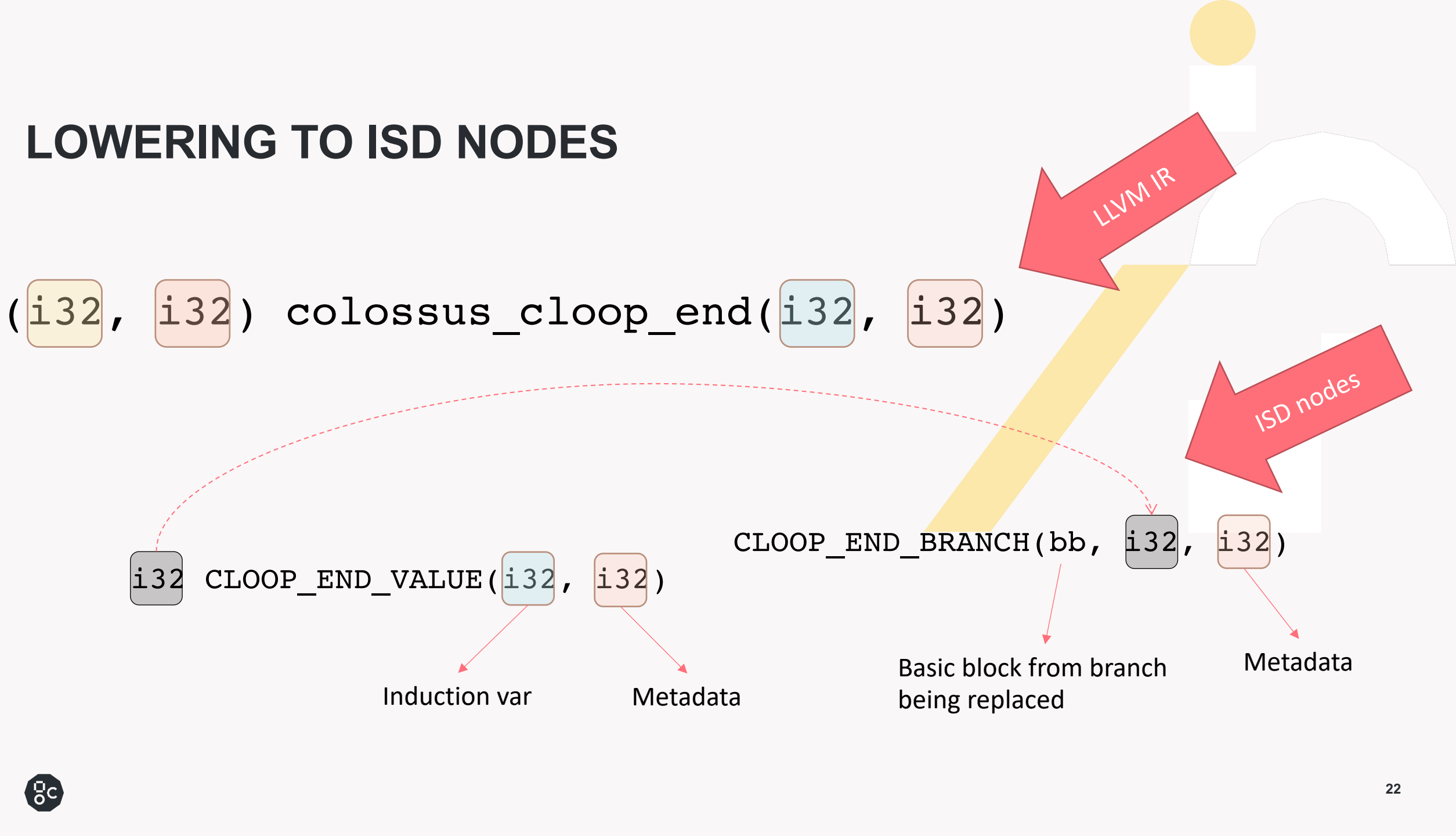
```
i32 colossus_cloop_guard(i32, i32)
```



LOWERING TO ISD NODES



LOWERING TO ISD NODES



INSTRUCTION SELECTION

Almost 1:1 mapping from ISD node with same-name pseudo instructions

```
def ColossusCloopBeginValue :
      SDNode<"ColossusISD::CLOOP_BEGIN_VALUE",
      ...>;
```

```
def CLOOP_BEGIN_VALUE : Pseudo<(outs MR:$op0),
                        (ins MR:$op1, imm16zi:$meta),
                        ...>;
```

```
def : Pat<(i32 (ColossusCloopBeginValue i32MR:$op0, imm16zi:$meta)),
          (i32 (CLOOP_BEGIN_VALUE i32MR:$op0, imm16zi:$meta))>;
```

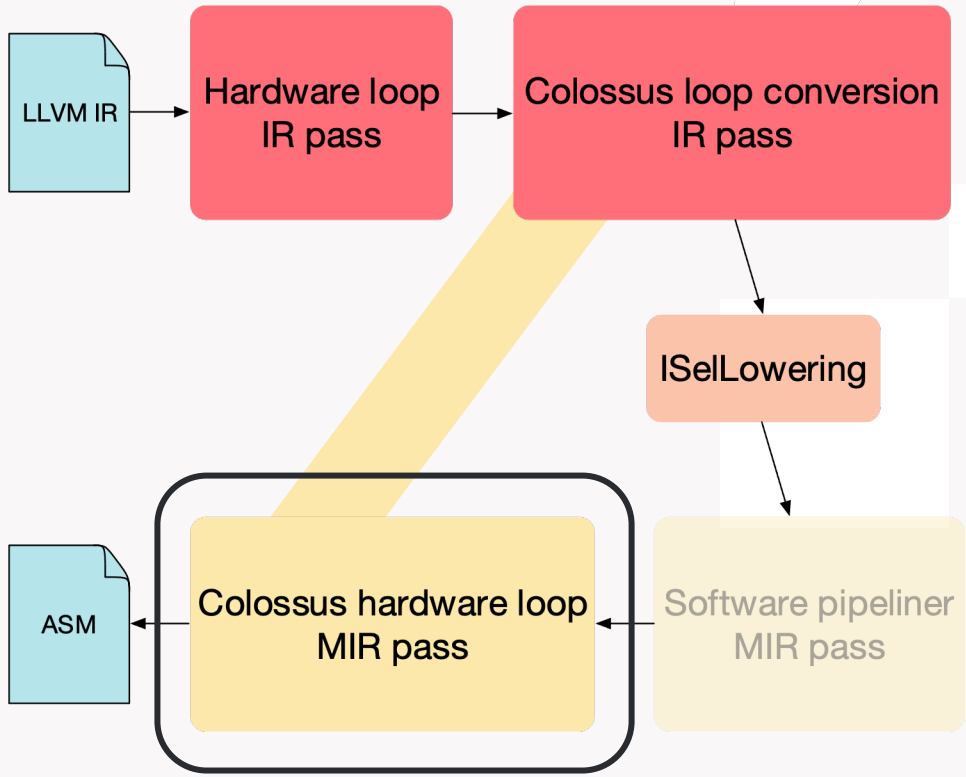
INSTRUCTION SELECTION

LLVM IR with IPU hardware loop intrinsics

LLVM MIR with IPU hardware loop pseudo instructions

HARDWARE LOOP MIR PASS

- Run as part of the addPreEmitPass passes
  - After register allocation but just before emitting assembly
- Removes all pseudo instructions and emits final hardware loop instruction
  - Late in pass pipeline because the rpt instruction bundle count can only be accurately computed near the end of all code transformations



Before hardware loop mir pass

HARDWARE LOOP MIR PASS

- Run as part of the addPreEmitPass passes
  - After register allocation but just before emitting assembly
- Removes all pseudo instructions and emits final hardware loop instruction
  - Late in pass pipeline because the rpt instruction bundle count can only be accurately computed near the end of all code transformations

```
void foo (int *out, int *in, unsigned size) {
  for (int i = 0; i < size; ++i) {
    *out += in[i];
  }
}
```

```
bb.0.entry:
  successors: %bb.1(0x50000000), %bb.3(0x30000000)
  liveins: $m0, $m1, $m2
  CLOOP_GUARD_BRANCH $m2, %bb.3, 1

bb.1.for.body.preheader:
; predecessors: %bb.0
  successors: %bb.2(0x80000000); %bb.2(100.00%)
  liveins: $m0, $m1, $m2
  $m2 = CLOOP_BEGIN_VALUE killed $m2(tied-def 0), 1
  $m3 = LD32_ZI $m0, $mzero, 0, 0
  CLOOP_BEGIN_TERMINATOR $m2, 1

bb.2.for.body:
; predecessors: %bb.1, %bb.2
  successors: %bb.2(0x7c000000), %bb.3(0x04000000)
  liveins: $m0, $m1, $m2, $m3
  $m1, $m4 = LD32STEP_SI $mzero, killed $m1(tied-def 0),
1, 0
  $m2 = CLOOP_END_VALUE killed $m2(tied-def 0), 1
  $m3 = nsw ADD killed $m3, killed $m4, 0
  ST32_ZI $m3, $m0, $mzero, 0, 0
  CLOOP_END_BRANCH $m2, %bb.2, 1

bb.3.for.cond.cleanup:
; predecessors: %bb.0, %bb.2
  RTN $lr, 0
```

After hardware loop mir pass

HARDWARE LOOP MIR PASS

- Run as part of the addPreEmitPass passes
  - After register allocation but just before emitting assembly
- Removes all pseudo instructions and emits final hardware loop instruction
  - Late in pass pipeline because the rpt instruction bundle count can only be accurately computed near the end of all code transformations

```
void foo (int *out, int *in, unsigned size) {
  for (int i = 0; i < size; ++i) {
    *out += in[i];
  }
}
```

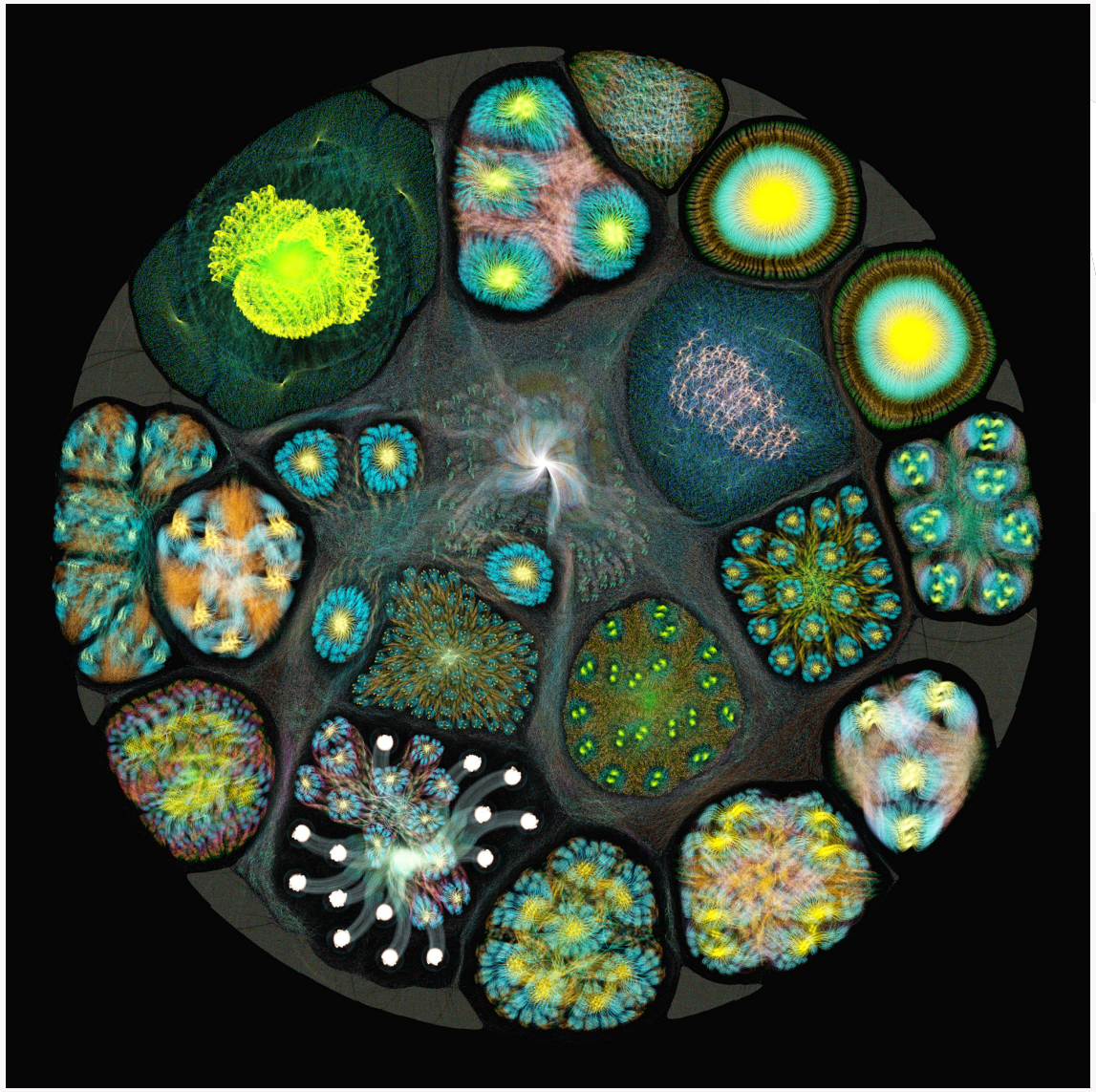
```
bb.0.entry:
  successors: %bb.1(0x80000000); %bb.1(100.00%)
  liveins: $m0, $m1, $m2

bb.1.for.body.preheader:
; predecessors: %bb.0
  successors: %bb.2(0x80000000); %bb.2(100.00%)
  liveins: $m0, $m1, $m2
  $m3 = LD32_ZI $m0, $mzero, 0, 0

bb.2.for.body:
; predecessors: %bb.1
  successors: %bb.3(0x80000000); %bb.3(100.00%)
  liveins: $m0, $m1, $m2, $m3
  RPT $m2, 2, 1 {
    $a13 = SETZI_A 0, 1
  }
  $m1, $m4 = LD32STEP_SI $mzero, killed $m1(tied-def 0),
1, 1 {
    $a13 = SETZI_A 0, 1
  }
  $m3 = nsw ADD killed $m3, killed $m4, 1 {
    $a13 = SETZI_A 0, 1
  }
    ST32_ZI $m3, $m0, $mzero, 0, 1 {
    $a13 = SETZI_A 0, 1
  }

bb.3.for.cond.cleanup:
; predecessors: %bb.2
  RTN $lr, 0
```

```
bb.0.entry:
  successors: %bb.1(0x50000000), %bb.3(0x30000000)
  liveins: $m0, $m1, $m2
  CLOOP_GUARD_BRANCH $m2, %bb.3, 1

bb.1.for.body.preheader:
; predecessors: %bb.0
  successors: %bb.2(0x80000000); %bb.2(100.00%)
  liveins: $m0, $m1, $m2
  $m2 = CLOOP_BEGIN_VALUE killed $m2(tied-def 0), 1
  $m3 = LD32_ZI $m0, $mzero, 0, 0
  CLOOP_BEGIN_TERMINATOR $m2, 1

bb.2.for.body:
; predecessors: %bb.1, %bb.2
  successors: %bb.2(0x7c000000), %bb.3(0x04000000)
  liveins: $m0, $m1, $m2, $m3
  $m1, $m4 = LD32STEP_SI $mzero, killed $m1(tied-def
0),
  $m2 = CLOOP_END_VALUE killed $m2(tied-def 0), 1
  $m3 = nsw ADD killed $m3, killed $m4, 0
  ST32_ZI $m3, $m0, $mzero, 0, 0
  CLOOP_END_BRANCH $m2, %bb.2, 1

bb.3.for.cond.cleanup:
; predecessors: %bb.0, %bb.2
  RTN $lr, 0
```

```
bb.0.entry:
  successors: %bb.1(0x80000000); %bb.1(100.00%)
  liveins: $m0, $m1, $m2

bb.1.for.body.preheader:
; predecessors: %bb.0
  successors: %bb.2(0x80000000); %bb.2(100.00%)
  liveins: $m0, $m1, $m2
  $m3 = LD32_ZI $m0, $mzero, 0, 0

bb.2.for.body:
; predecessors: %bb.1
  successors: %bb.3(0x80000000); %bb.3(100.00%)
  liveins: $m0, $m1, $m2, $m3
  RPT $m2, 2, 1 {
    $a13 = SETZI_A 0, 1
  }
  $m1, $m4 = LD32STEP_SI $mzero, killed $m1(tied-def 0),
1, 1 {
    $a13 = SETZI_A 0, 1
  }
  $m3 = nsw ADD killed $m3, killed $m4, 1 {
    $a13 = SETZI_A 0, 1
  }
    ST32_ZI $m3, $m0, $mzero, 0, 1 {
    $a13 = SETZI_A 0, 1
  }

bb.3.for.cond.cleanup:
; predecessors: %bb.2
  RTN $lr, 0
```

HARDWARE LOOP USERS

- Most of our users are internal, working on Poplibs, and considered “power users”
  - Comfortable and aware of IPU architecture
  - Will often use assembly for optimal solution
- Try to convince them to use C++ without performance loss
  - builtin_assume
  - rptsize_t
  - [WIP] pragma



HARDWARE LOOP USERS: ASSUME

- Allow user to use __builtin_assume calls to insert assumptions on trip count

```
void foo (int *out, int *in, unsigned size) {
  for (int i = 0; i < size; ++i) {
    *out += in[i];
  }
}
```

Emits brnzdec

```
void assume_foo (int *out, int *in, unsigned size) {
  __builtin_assume(size < 0xFFFF);
  for (int i = 0; i < size; ++i) {
    *out += in[i];
  }
}
```

Emits rpt

HARDWARE LOOP USERS: RPTSIZE_T

- Unsigned int type to be used as trip count to target rpt instructions
  - Bit width will be rpt instruction trip count width (and change according to architecture, if changed)
- C++ only
  - Uses _BitInt(n) (formerly _ExtInt(n))
  - Overloads operations with other (unsigned) int types

```
void bar (int *out, int *a, unsigned n) {
  for (int i = 0; i < n; ++i) {
    *out += a[i];
  }
}

void rptsizebar1 (int *out, int *a, rptsize_t n) {
  for (int i = 0; i < n; ++i) {
    *out += a[i];
  }
}

void rptsizebar2 (int *out, int *a, rptsize_t n) {
  for (rptsize_t i = 0; i < n; ++i) {
    *out += a[i];
  }
}
```

HARDWARE LOOP USERS: RPTSIZE_T

- Unsigned int type to be used as trip count to target rpt instructions
  - Bit width will be rpt instruction trip count width (and change according to architecture, if changed)
- C++ only
  - Uses _BitInt(n) (formerly _ExtInt(n))
  - Overloads operations with other (unsigned) int types

```
_Z3barPiS_j:
  brz $m2, .LBB0_3
  add $m2, $m2, -1
  ld32 $m3, $m0, $m15, 0
.LBB0_2:
  ld32step $m4, $m15, $m1+=, 1
  add $m3, $m3, $m4
  st32 $m3, $m0, $m15, 0
  brnzdec $m2, .LBB0_2
.LBB0_3:
  br $m10
_Z11rptsizebar1PiS_6ap_intILj12ELb0EE:
  ldz16 $m2, $m2, $m15, 0
  ld32 $m3, $m0, $m15, 0
  {
    rpt $m2, 2
    fnop
  }
  {
    ld32step $m4, $m15, $m1+=, 1
    fnop
  }
  {
    add $m3, $m3, $m4
    fnop
  }
  {
    st32 $m3, $m0, $m15, 0
    fnop
  }
  br $m10
```

HARDWARE LOOP USERS: PRAGMA

- Not all loops have a trip count explicitly defined in a variable or have exit conditions dependent on an iterating integer (e.g., !container.empty())
- Hence, we're thinking of adding pragma support for said loops such that power users can still force rpt instructions

pragma hwloop arg	semantics
force_any	ignore loop count & profitability (try brnzdec if not possible)
force_rpt	ignore loop count & profitability (error if not possible)
force_brnzdec	ignore rpt (implies brnzdec)
none	disable rpt & brnzdec

THANK YOU

Janek van Oirschot
janekvo@graphcore.ai