LLVM RISC-V Retrospect & Outlook
回顾与展望

肖玮 Wei Xiao (wei3.xiao@intel.com)

Contributors:
刘天乐 Tianle Liu (tianle.l.liu@intel.com)
王天晴 Tianqing Wang (tianqing.wang@intel.com)
闻浩海 Haohai Wen (haohai.wen@intel.com)
裴根 Gen Pei (gen.pei@intel.com)

Agenda

- LLVM RISC-V Target Support Overview
- LLVM Architecture Overview
- RISC-V & LLVM FE
- RISC-V & LLVM ME
- RISC-V & LLVM BE
- RISC-V Performance Analysis w/ QEMU

LLVM RISC-V Target Support Overview

- Base ISAs

Architecture	Description
riscv32	RISC-V with XLEN=32 (i.e. RV32I or RV32E)
riscv64	RISC-V with XLEN=64 (i.e. RV64I)

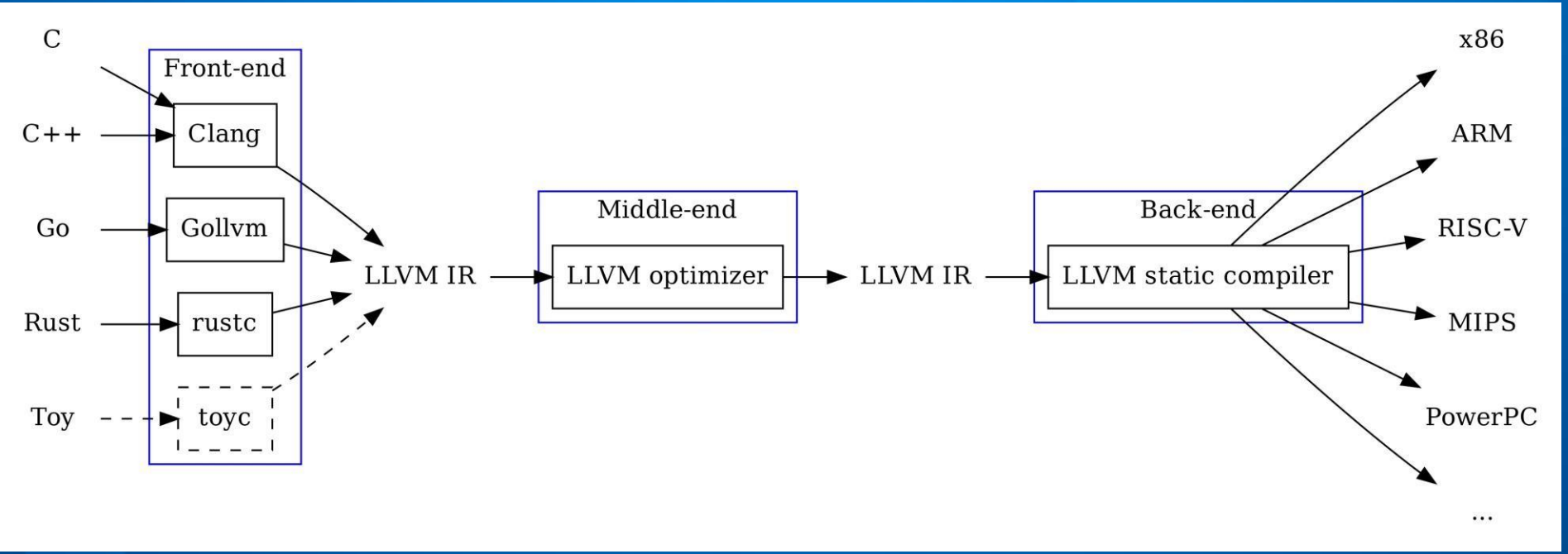
- Extensions

Table 109 Ratified Extensions by Status

Extension	Status
A	Supported
B	Supported
C	Supported
D	Supported
F	Supported
E	Supported (See note)
H	Assembly Support
M	Supported
.....	

- Experimental Extensions
  - ✓ experimental-zacas
  - ✓ ~~experimental-zihintntl~~
  - ✓ ~~experimental-zvfh~~
  - ✓ ~~experimental-zca~~
  - ✓ ...
- Vendor Extensions
  - ✓ XVentanaCondOps
  - ✓ XTHeadVdot
  - ✓ XSfvcp
  - ✓ ...
- Profiles
  - ✓ rvi20u32
  - ✓ rvi20u64
  - ✓ ...

LLVM Architecture Overview



- Classical design to support multiple source languages or target architectures
- Serves a broader set of programmers which naturally leads to more enhancements and improvements to the compiler
- LLVM IR is a Complete Code Representation
- LLVM is a Collection of Libraries

RISC-V & LLVM FE

RISCV ABI generate different type for structure return type

Source code	X86 IR from FE	RISCV IR from FE	Comment of RISCV (Xlen = 64)

struct A {
int
int
}
	%struct.A = type { i32, i32 }
	define dso_local i64 @_Z5myfunv()
		%struct.A = type { i32, i32 }
		define dso_local i64 @_Z5myfunv()
			Aggregates which are <= 2*XLEN will be passed in registers if possible

struct A {
long
int
}
	%struct.A = type { i64, i32 }
	define dso_local { i64, i32 } @_Z5myfunv()
		%struct.A = type { i64, i32 }
		define dso_local [2 x i64] @_Z5myfunv()
			Use a single XLEN int if possible, 2*XLEN if 2*XLEN alignment is required, and a 2-element XLEN array if only XLEN alignment is required.

struct A {
long*
int
}
	%struct.A = type { ptr, i32 }
	define dso_local { ptr, i32 } @_Z5myfunv()
		%struct.A = type { ptr, i32 }
		define dso_local [2 x i64] @_Z5myfunv()	

struct A {
long
double
}
	%struct.A = type { i64, double }
	define dso_local { i64, double }
	@_Z5myfunv()
		%struct.A = type { i64, double }
		define dso_local { i64, double } @_Z5myfunv()
			Determine if a struct is eligible for passing according to the floating point calling convention (i.e., when flattened it contains a single fp value, fp+fp, or int+fp of appropriate size). If so, NeededArgFPRs and NeededArgGPRs are incremented appropriately.

struct A {
long*
int
long
};
	%struct.A = type { ptr, i32, i64 }
	define dso_local void @_Z5myfunv(ptr noalias nocapture writeonly sret(%struct.A) align 8 %agg.result)
		%struct.A = type { ptr, i32, i64 }
		define dso_local void @_Z5myfunv(ptr noalias nocapture writeonly sret(%struct.A) align 8 %agg.result)	

RISC-V & LLVM FE

"char" in RISC-V target

	X86-64 clang	Rv64 clang
bool	zeroext i1	zeroext i1
char	signext i8	zeroext i8
short	signext i16	signext i16
int	i32	signext i32
long	i64	i64

```
//i8_test.cpp:
char __attribute__((noinline)) foo() {
 return -1;
}
int main( int argc, char * argv [] )
{
 char ret;
 ret = foo();
 if( ret >= 0)
  printf(">=0\n");
 else
 printf("<0\n");
}
```

Runtime Result

X86:

$ ./i8_test.x86

<0

RISC-V:

$ ./i8_test.rv

>=0

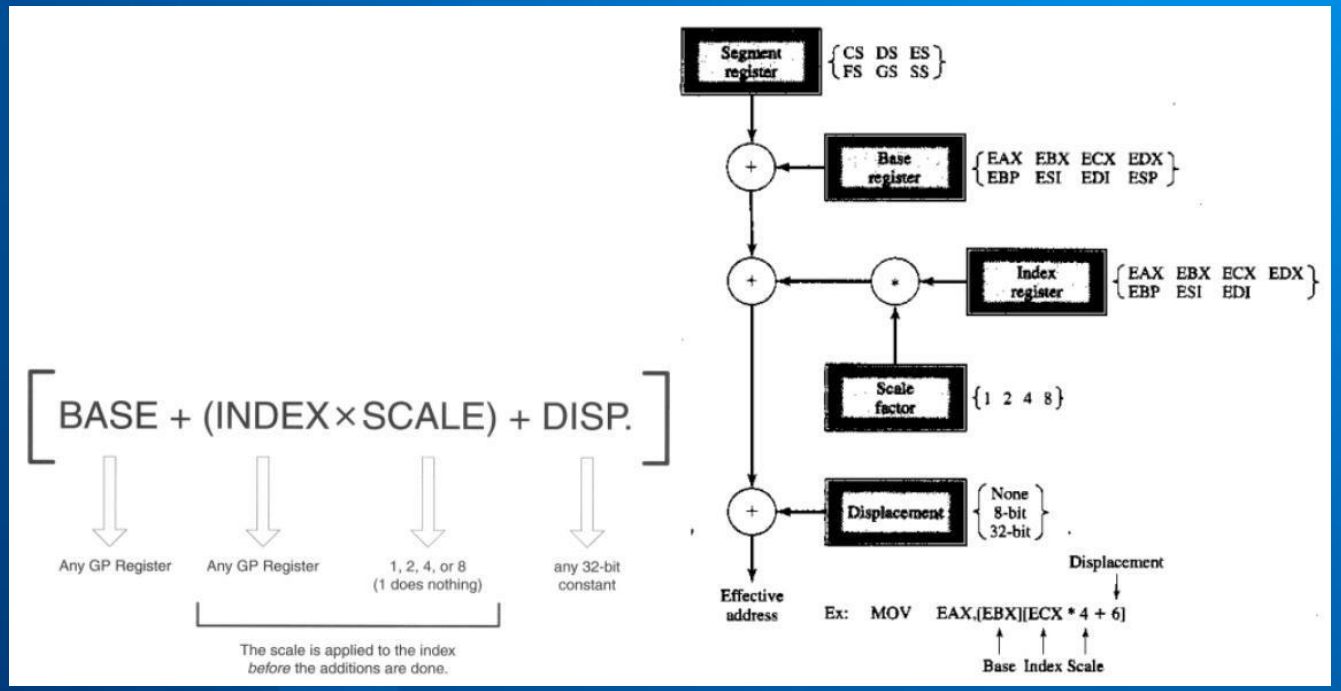
RISC-V

```
386 ; Function Attrs: mustprogress norecurse uwtable
387 define dso_local noundef signext i32 @main(i32 noundef signext %argc, ptr noundef %argv) #1 {
388 entry:
389   %call = call noundef zeroext i8 @_Z3foov()
390   %conv = zext i8 %call to i32
391   %cmp = icmp sge i32 %conv, 0
392   br i1 %cmp, label %if.then, label %if.else
393
394 if.then:                                          ; preds = %entry
395   %call1 = call signext i32 (ptr, ...) @printf(ptr noundef @.str)
396   br label %if.end
397
398 if.else:                                          ; preds = %entry
399   %call2 = call signext i32 (ptr, ...) @printf(ptr noundef @.str.1)
400   br label %if.end
401
402 if.end:                                           ; preds = %if.else, %if.then
403   ret i32 0
404 }
```

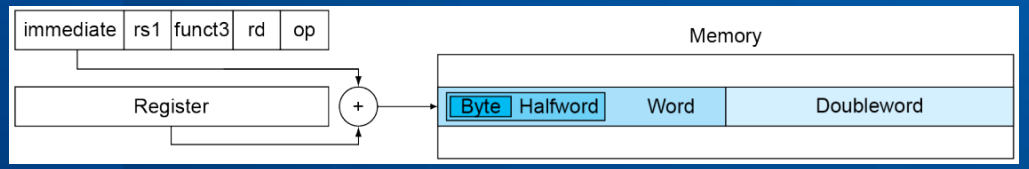
X86

```
386 ; Function Attrs: mustprogress norecurse uwtable
387 define dso_local noundef i32 @main(i32 noundef %argc, ptr noundef %argv) #1 {
388 entry:
389   %call = call noundef signext i8 @_Z3foov()
390   %conv = sext i8 %call to i32
391   %cmp = icmp sge i32 %conv, 0
392   br i1 %cmp, label %if.then, label %if.else
393
394 if.then:                                          ; preds = %entry
395   %call1 = call i32 (ptr, ...) @printf(ptr noundef @.str)
396   br label %if.end
397
398 if.else:                                          ; preds = %entry
399   %call2 = call i32 (ptr, ...) @printf(ptr noundef @.str.1)
400   br label %if.end
401
402 if.end:                                           ; preds = %if.else, %if.then
403   ret i32 0
404 }
```

RISC-V & LLVM ME: Addressing



X86



RISC-V

Register indirect with offset

```
ldr     x0, [Xn/sp, #imm]
ldr     x0, [Xn/sp]         ; #0 is implied if omitted
```

Register indirect with index

```
ldr     x0, [Xn/sp, Rn/zr, extend]
```

Extended	Effective address	Index format
[a, b, UXTW #0] [a, b, UXTW]	a + (uint32_t)b	32-bit unsigned byte offset.
[a, b, UXTW #sizeshift]	a + (uint32_t)b * size	32-bit unsigned element offset.
[a, b, SXTW #0] [a, b, SXTW]	a + (int32_t)b	32-bit signed byte offset.
[a, b, SXTW #sizeshift]	a + (int32_t)b * size	32-bit signed element offset.
[a, b, UXTX #0] [a, b, UXTX] [a, b, LSL #0] [a, b]	a + (uint64_t)b	64-bit unsigned byte offset.
[a, b, UXTX #sizeshift] [a, b, LSL #sizeshift]	a + (uint64_t)b * size	64-bit unsigned element offset.
[a, b, SXTX #0]	a + (int64_t)b	64-bit signed byte offset.
[a, b, SXTX #sizeshift]	a + (int64_t)b * size	64-bit signed element offset.

AArch64 (aka arm64)

RISC-V & LLVM ME: IPO

Inlining in RISC-V is less aggressive due to more cost

```
; cost before =-5, cost after =-5, ..., cost delta = 0
   %arrayidx = getelementptr inbounds ptr, ptr %1, i64 %idxprom
...
Cost: 230
```

X86 → Inlined

```
; cost before =-5, cost after = 0, ..., cost delta = 5
   %arrayidx = getelementptr inbounds ptr, ptr %1, i64 %idxprom
...
Cost: 250
```

RISC-V → Not Inlined

(Default threshold: 250)

Different implementation of TargetLowering::isLegalAddressingMode() among the two targets

Outlook: more ME tuning works for RISC-V

RISC-V & LLVM ME: Vectorization

Scalable vectorization is enabled by default

```
for.body:
  %iv = phi i64 [ 0, %entry ], [ %iv.next, %for.body ]
  %arrayidx = getelementptr inbounds i64, ptr %a, i64 %iv
  %elem = load i64, ptr %arrayidx
  %divrem = udiv i64 %elem, %v
  store i64 %divrem, ptr %arrayidx
  %iv.next = add nuw nsw i64 %iv, 1
  %exitcond.not = icmp eq i64 %iv.next, 1024
  br i1 %exitcond.not, label %for.end, label %for.body
```

Loop Vectorizer

```
vector.ph:                                        ; preds = %entry
  %2 = call i64 @llvm.vscale.i64()
  %3 = mul i64 %2, 2
  %n.mod.vf = urem i64 1024, %3
  %n.vec = sub i64 1024, %n.mod.vf
  %4 = call i64 @llvm.vscale.i64()
  %5 = mul i64 %4, 2
  %broadcast.splatinsert = insertelement <vscale x 2 x i64> poison, i64 %v, i64 0
  %broadcast.splat = shufflevector <vscale x 2 x i64> %broadcast.splatinsert, <vscale x 2 x i64>
 poison, <vscale x 2 x i32> zeroinitializer
  br label %vector.body

vector.body:                                      ; preds = %vector.body, %vector.ph
  %index = phi i64 [ 0, %vector.ph ], [ %index.next, %vector.body ]
  %6 = add i64 %index, 0
  %7 = getelementptr inbounds i64, ptr %a, i64 %6
  %8 = getelementptr inbounds i64, ptr %7, i32 0
  %wide.load = load <vscale x 2 x i64>, ptr %8, align 8
  %9 = udiv <vscale x 2 x i64> %wide.load, %broadcast.splat
  store <vscale x 2 x i64> %9, ptr %8, align 8
  %index.next = add nuw i64 %index, %5
  %10 = icmp eq i64 %index.next, %n.vec
  br i1 %10, label %middle.block, label %vector.body, !llvm.loop !0

middle.block:                                     ; preds = %vector.body
```

RISC-V & LLVM ME: Vectorization

Loop vectorization using fixed length vectors

```
for.body:
  %iv = phi i64 [ 0, %entry ], [ %iv.next, %for.body ]
  %arrayidx = getelementptr inbounds i64, ptr %a, i64 %iv
  %elem = load i64, ptr %arrayidx
  %divrem = udiv i64 %elem, %v
  store i64 %divrem, ptr %arrayidx
  %iv.next = add nuw nsw i64 %iv, 1
  %exitcond.not = icmp eq i64 %iv.next, 1024
  br i1 %exitcond.not, label %for.end, label %for.body
```

Loop Vectorizer

```
vector.ph:                                        ; preds = %entry
  %broadcast.splatinsert = insertelement <4 x i64> poison, i64 %v, i64 0
  %broadcast.splat = shufflevector <4 x i64> %broadcast.splatinsert, <4 x i64> poison, <4 x i32>
zeroinitializer
  br label %vector.body

vector.body:                                      ; preds = %vector.body, %vector.ph
  %index = phi i64 [ 0, %vector.ph ], [ %index.next, %vector.body ]
  %0 = add i64 %index, 0
  %1 = add i64 %index, 4
  %2 = getelementptr inbounds i64, ptr %a, i64 %0
  %3 = getelementptr inbounds i64, ptr %a, i64 %1
  %4 = getelementptr inbounds i64, ptr %2, i32 0
  %5 = getelementptr inbounds i64, ptr %2, i32 4
  %wide.load = load <4 x i64>, ptr %4, align 8
  %wide.load1 = load <4 x i64>, ptr %5, align 8
  %6 = udiv <4 x i64> %wide.load, %broadcast.splat
  %7 = udiv <4 x i64> %wide.load1, %broadcast.splat
  store <4 x i64> %6, ptr %4, align 8
  store <4 x i64> %7, ptr %5, align 8
  %index.next = add nuw i64 %index, 8
  %8 = icmp eq i64 %index.next, 1024
  br i1 %8, label %middle.block, label %vector.body, !llvm.loop !0

middle.block:                                     ; preds = %vector.body
```

RISC-V & LLVM BE

RISC-V "V" Vector Extension

- Scalable vectorization

```
.LBB0_2:                                # %vector.ph
        negw    a1, a2
        andi    a1, a1, 1024
        slli    a3, a3, 1
        mv      a4, a0
        mv      a5, a1
        vsetvli a6, zero, e64, m2, ta, ma
.LBB0_3:                                # %vector.body
                                        # =>This Inner Loop Header: Depth=1
        vl2re64.v       v8, (a4)
        vdivu.vx        v8, v8, s0
        vs2r.v  v8, (a4)
        sub     a5, a5, a2
        add     a4, a4, a3
        bnez    a5, .LBB0_3
# %bb.4:                                # %middle.block
        li      a2, 1024
        beq     a1, a2, .LBB0_7
.LBB0_5:                                # %scalar.ph
```

- Fixed length vectorization

```
        li      a2, 0
        lui     a3, 2
        add     a3, a0, a3
        vsetivli        zero, 4, e64, m2, ta, ma
.LBB0_1:                                # %vector.body
                                        # =>This Inner Loop Header: Depth=1
        addi    a4, a0, 32
        vle64.v v8, (a0)
        vle64.v v10, (a4)
        vdivu.vx        v8, v8, a1
        vdivu.vx        v10, v10, a1
        vse64.v v8, (a0)
        vse64.v v10, (a4)
        addi    a0, a0, 64
        addi    a2, a2, 8
        bne     a0, a3, .LBB0_1
# %bb.2:                                # %for.end
```

RISC-V & LLVM BE Optimization Example

```
; 'zero' register not addressable in compressed store.
                  =>   li a1, 0
sw zero, 0(a0)    =>   c.sw a1, 0(a0)
sw zero, 8(a0)    =>   c.sw a1, 8(a0)
sw zero, 4(a0)    =>   c.sw a1, 4(a0)
sw zero, 24(a0)   =>   c.sw a1, 24(a0)
```

```
; compressed stores support limited offsets
lui a2, 983065      =>   lui a2, 983065
                    =>   addi  a3, a2, -256
sw  a1, -236(a2)    =>   c.sw  a1, 20(a3)
sw  a1, -240(a2)    =>   c.sw  a1, 16(a3)
sw  a1, -244(a2)    =>   c.sw  a1, 12(a3)
sw  a1, -248(a2)    =>   c.sw  a1, 8(a3)
sw  a1, -252(a2)    =>   c.sw  a1, 4(a3)
sw  a0, -256(a2)    =>   c.sw  a0, 0(a3)
```

RISCVMakeCompressible focused on reducing code size. It looks for cases where an instruction has been selected which can't be represented by one of the compressed (16-bit as opposed to 32-bit wide) instruction forms:

- register not being one of the registers addressable from the compressed instruction
- offset being out of range

RISC-V & LLVM BE Optimization Example

The Problem

- Middle-end optimizations replace `sext` instructions with `zext` if the sign bit is known to be zero.
- **i32->i64 `zext` is never cheaper than `sext` for RISC-V and `sext` can be free.**

```
void foo(int *x, int n) {
  for (int i = 0; i < n; ++i)
    x[i] += 1;
}
```

RISC-V Assembly

```
foo:
        blez    a1, .LBB0_2
        slli    a1, a1, 32
        srli    a1, a1, 32
.LBB0_1:
        ; loop body
.LBB0_2:
        ret
```

LLVM IR

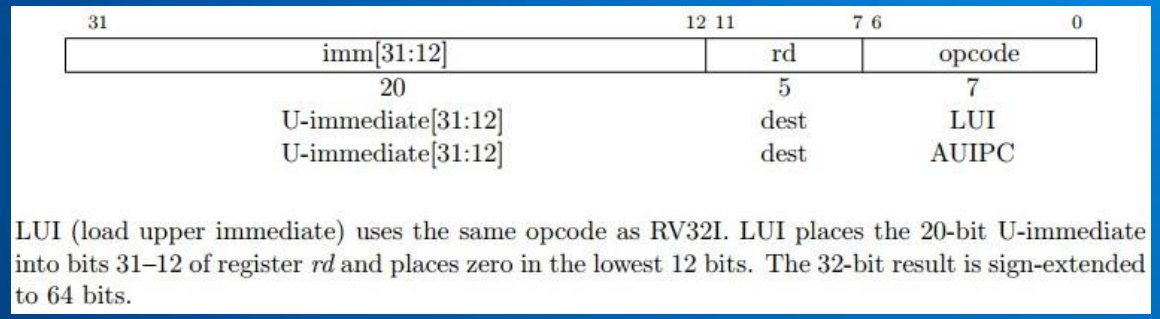
```
define void @foo(ptr %x, i32 signext %n) {
entry:
  %cmp3 = icmp sgt i32 %n, 0
  br i1 %cmp3, label %preheader, label %cleanup

preheader:
  ; sign bit of i32 %n is known 0 here.
  %wide.trip.count = zext i32 %n to i64
  br label %for.body

for.body:
  ...

cleanup:
  ret void
```

RISCVCodeGenPrepare

Partial Solution

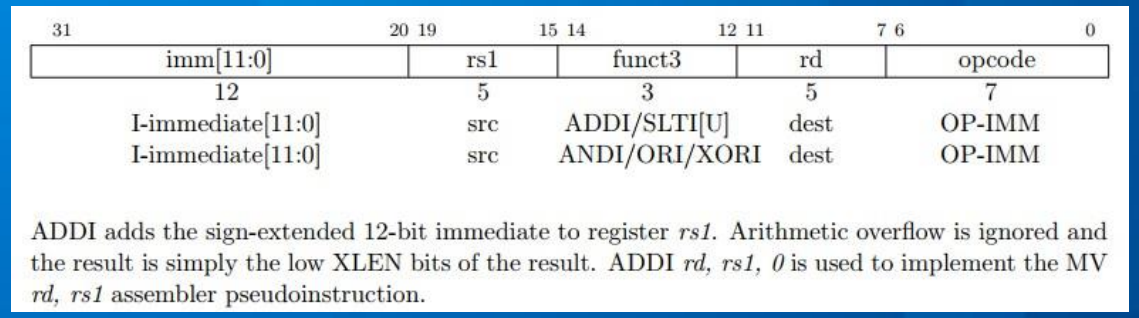
- Pre-instruction selection IR pass
- For each `zext` instruction in basic blocks with a single predecessor
  - Examine the terminator condition of predecessor
  - If condition implies the sign bit is 0 when branching to the `zext`
    - Replace `zext` with `sext`
- Looking for a single predecessor is a very simple dominance check
  - **It will miss some cases.**

Macro-fusion LUI+ADDI(W) Optimization



31 — 12	11 — 7	6 — 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC

LUI (load upper immediate) uses the same opcode as RV32I. LUI places the 20-bit U-immediate into bits 31–12 of register *rd* and places zero in the lowest 12 bits. The 32-bit result is sign-extended to 64 bits.



31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

ADDI adds the sign-extended 12-bit immediate to register *rs1*. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result. ADDI *rd, rs1, 0* is used to implement the MV *rd, rs1* assembler pseudoinstruction.

uOP

a0 = ptr @.str

```
lui     a0, %hi(.L.str)
fcvt.s.w        fa0, a1
addi    a0, a0, %lo(.L.str)
```

→

```
fcvt.s.w        fa0, a1
lui     a0, %hi(.L.str)
addi    a0, a0, %lo(.L.str)
```

3 → 2

a0 = 2048

```
li      a0, 1
slli    a0, a0, 11
```

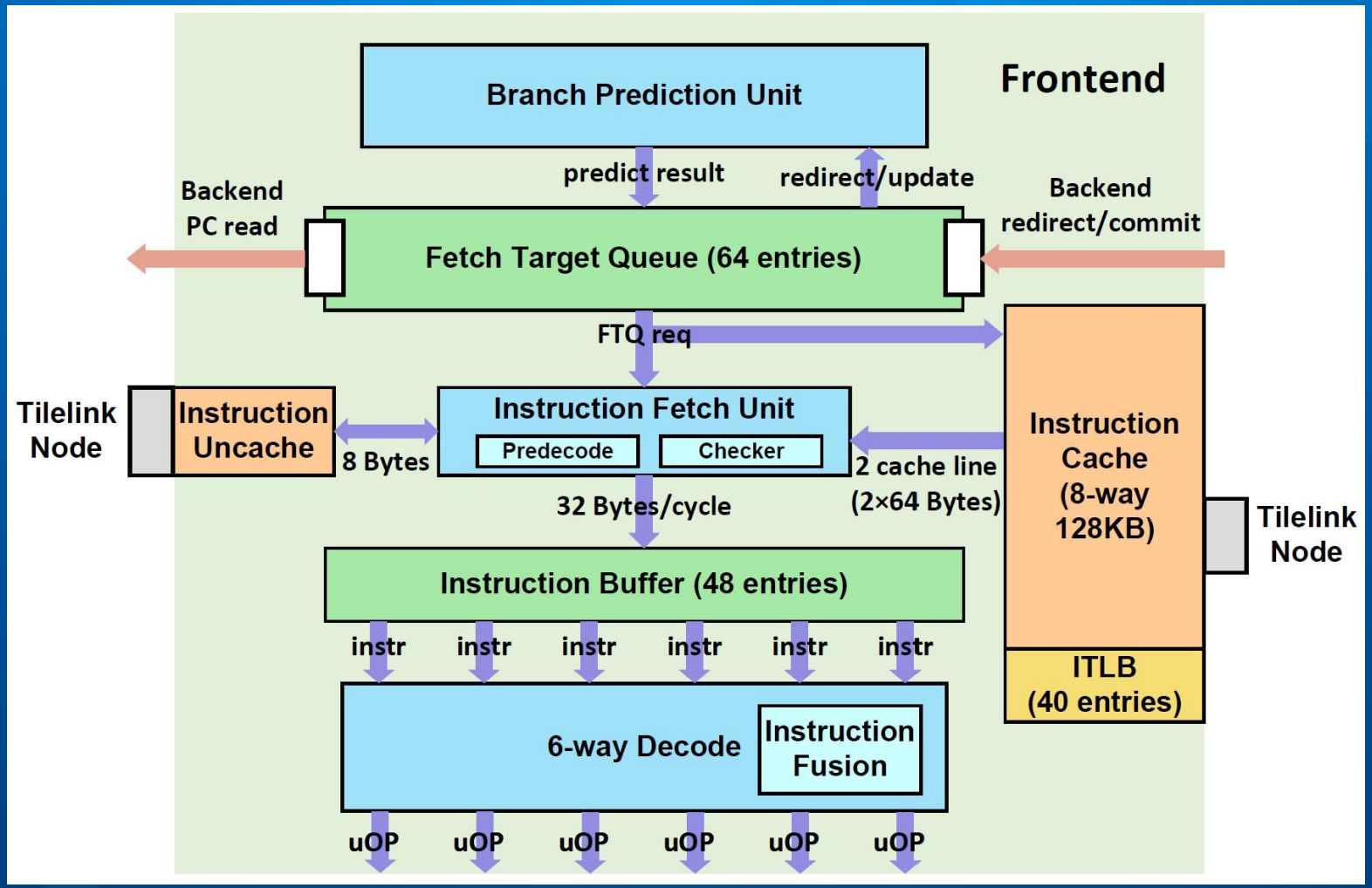
→

```
lui     a0, 1
addiw   a0, a0, -2048
```

2 → 1

Destination of LUI should be the ADDI(W) source register and destination register.

XiangShan Frontend



FusionDecoder & LLVM Optimizations

- clear upper 32 bits / get lower 32 bits: slli r1,r0, 32 + srli r1,r1,32
- clear upper 48 bits / get lower 16 bits: slli r1,r0, 48 + srli r1,r1,48
- clear upper 48 bits / get lower 16 bits: slliw r1,r0, 16 + srliw r1,r1,16
- sign-extend a 16-bit number: slliw r1,r0, 16 + sraiw r1,r1,16
- shift left by one and add: slli r1,r0, 1 + add r1,r1,r2
- shift left by two and add: slli r1,r0, 2 + add r1,r1,r2
- shift left by three and add: slli r1,r0, 3 + add r1,r1,r2
- shift zero-extended word left by one: slli r1,r0, 32 + srli r1,r0, 31
- shift zero-extended word left by two: slli r1,r0, 32 + srli r1,r0, 30
- shift zero-extended word left by three: slli r1,r0, 32 + srli r1,r0, 29
- get the second byte: srli r1,r0, 8 + andi r1,r1, 255
- shift left by four and add: slli r1,r0, 4 + add r1,r1,r2
- shift right by 29 and add: srli r1,r0, 29 + add r1,r1,r2
- shift right by 30 and add: srli r1,r0, 30 + add r1,r1,r2
- shift right by 31 and add: srli r1,r0, 31 + add r1,r1,r2
- shift right by 32 and add: srli r1,r0, 32 + add r1,r1,r2
- add one if odd, otherwise unchanged: andi r1,r0, 1 + add r1,r1,r2
- add one if odd (in word format), otherwise unchanged: andi r1,r0, 1 + addw r1,r1,r2
- addw and extract its lower 8 bits (fused into addwbyte)
- addw and extract its lower 1bit (fused into addwbit)
- addw and zext.h (fused into addwzexth)
- addw and sext.h (fused into addwsexth)
- logic operation and extract its LSB
- logic operation and extract its lower 16 bits
- OR(Cat(src1(63, 8), 0.U(8.W)), src2)
- mul 7-bit data with 32-bit data

Outlook: more BE tuning works for RISC-V uArch

RISC-V Performance Analysis w/ QEMU

Linux perf tool limitations on some RISCV platforms:

1. Hard to get instruction-level perf data
2. Hard to distinguish between reading and writing for cache miss

Therefore, it is difficult to do in-depth analysis when analyzing data reading and writing performance bottlenecks.

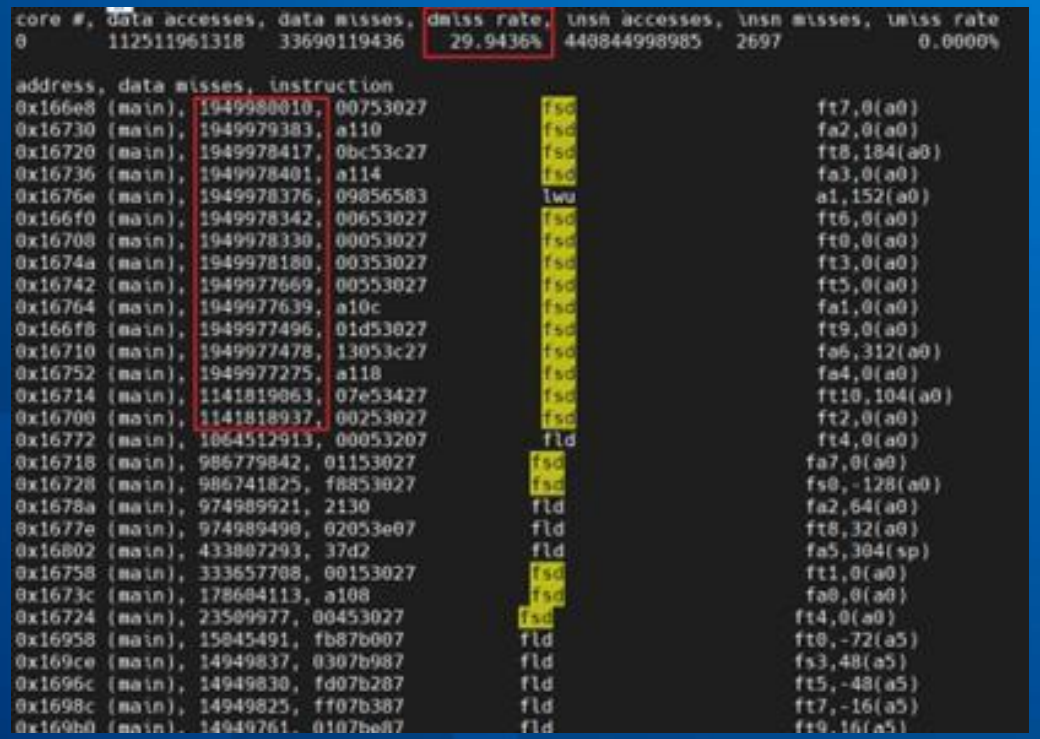
RISC-V Performance Analysis w/ QEMU

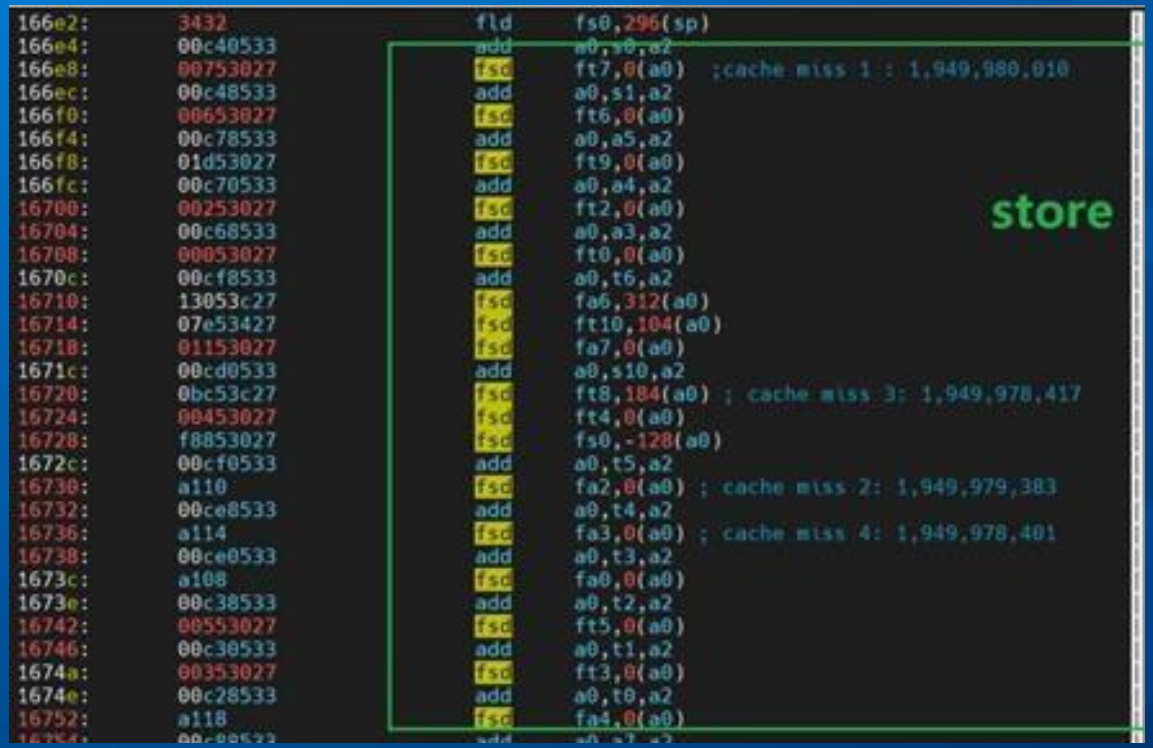
Use QEMU to simulate cache behavior:

- riscv-toolchain/qemu/contrib/plugins/cache.c
- Compile libmycache.so
- Run "qemu-riscv64 -cpu rv64 -plugin libmycache.so,dcachesize=n -d plugin ./app &> cache.log
- The default setting is 32 sets. Set according to the actual chip

Result Example

The cache miss information recorded by QEMU can be mapped to where the cache miss occurred in the bin. To carry out targeted optimization.





Legal Disclaimer & Optimization Notice

- INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.
- Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.
-

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804


intel