Anton Johansson
anjo@rev.ng

Alessandro Di Federico
ale@rev.ng

November 19 – LLVM Developer Meeting 2021

Research paid for by Qualcomm Innovation Center, Inc.

- We are building a decompiler based on QEMU and LLVM
- We do consultancy on compilers and emulators

Motivation

Enter QEMU

1st attempt: Pseudo C $\rightarrow$ TCG translator

2nd attempt: LLVM IR $\rightarrow$ TCG translator

Demo

Summary

Motivation

Enter QEMU

1st attempt: Pseudo C $\rightarrow$ TCG translator

2nd attempt: LLVM IR $\rightarrow$ TCG translator

Demo

Summary

Goal: write an fast, open-source emulator for Qualcomm® Hexagon™ DSP[1]

[1] Qualcomm and Hexagon are trademarks or registered trademarks of Qualcomm Incorporated.

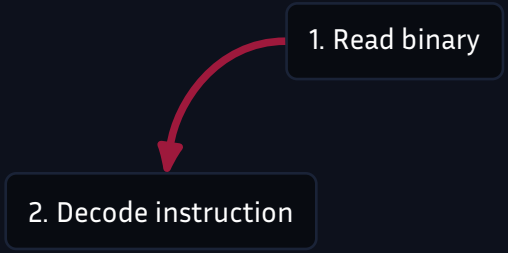
- VLIW
- > 2000 user mode instructions
- Complicated instructions designed for digital signal processing
- Example Hexagon Assembly (part of inner loop of FFT)

```
{ R17:16 = MEMD(R0++M1)
  MEMD(R6++M1) = R25:24
  R20 = CMPY(R20, R8):<<1:rnd:sat
  R11:10 = VADDH(R11:10, R13:12)
}:endloop0
```

- KVM 19: QEMU-Hexagon: Automatic Translation of the ISA Manual Pseudcode to Tiny Code Instructions

```
struct CPU {
    uint32_t gprs[16];
    uint32_t fp_regs[16];
    uint32_t flags;
    /* ... */
};
```

1. Read binary

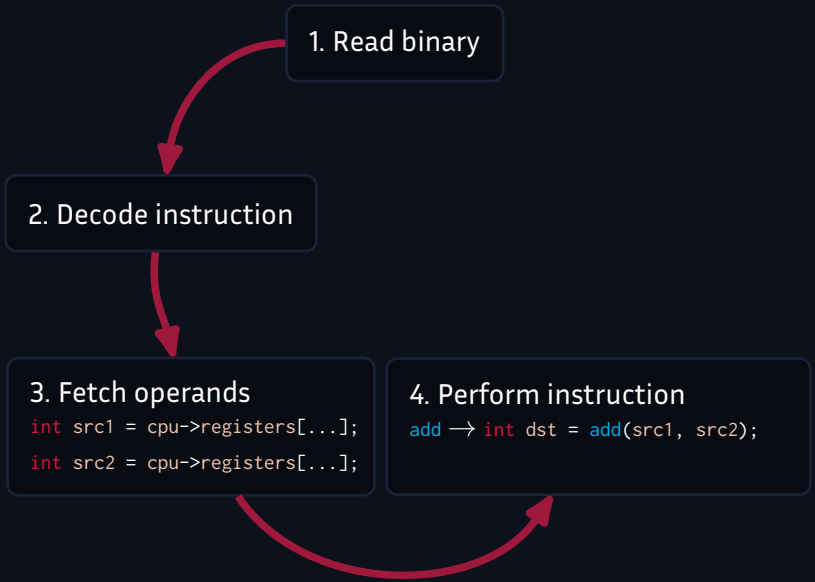

1. Read binary
2. Decode instruction

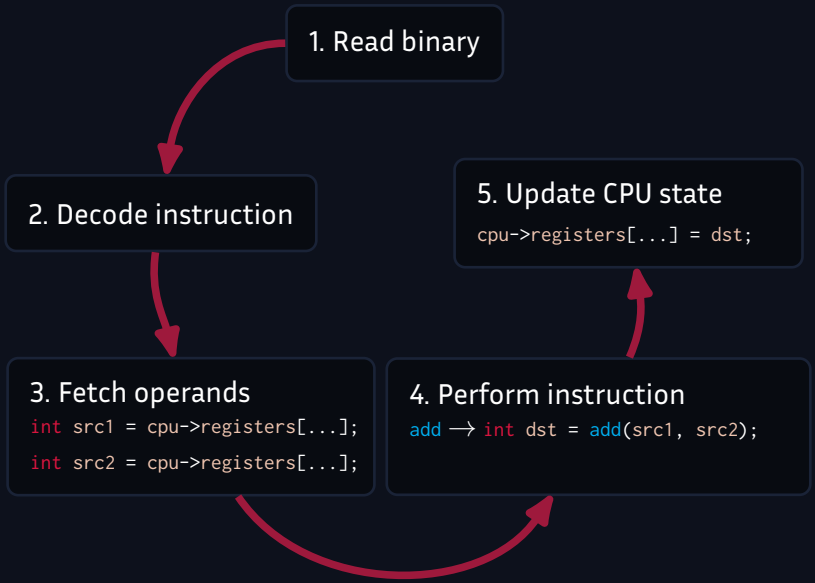
1. Read binary

2. Decode instruction

3. Fetch operands

```
int src1 = cpu->registers[...];
int src2 = cpu->registers[...];
```


1. Read binary
2. Decode instruction
3. Fetch operands
int src1 = cpu->registers[...];
int src2 = cpu->registers[...];
4. Perform instruction
add → int dst = add(src1, src2);


1. Read binary
2. Decode instruction
3. Fetch operands
int src1 = cpu->registers[...];
int src2 = cpu->registers[...];
4. Perform instruction
add → int dst = add(src1, src2);
5. Update CPU state
cpu->registers[...] = dst;

1. Read binary

2. Decode instruction

3. Fetch operands

```
int src1 = cpu->registers[...];
int src2 = cpu->registers[...];
```

4. Perform instruction

```
add ⟶ int dst = add(src1, src2);
```

5. Update CPU state

```
cpu->registers[...] = dst;
```

The add function would be something like:

```
int add(int reg_a, int reg_b) {
    return reg_a + reg_b;
}
```

```
> ./myGreatEmulator someTest
```

```
> ./myGreatEmulator someTest
.
```

```
> ./myGreatEmulator someTest
..
```

```
> ./myGreatEmulator someTest
...
```

```
> ./myGreatEmulator someTest
....
```

```
> ./myGreatEmulator someTest
.....
```

```
> ./myGreatEmulator someTest
......
```

```
> ./myGreatEmulator someTest
......
it's slow :(
```

```
> ./myGreatEmulator someTest
......
it's slow :(
```

The interpreter approach incurs in:

1. large overhead for each instruction
2. performing many memory accesses to read/write CPU state

Using an optimizing JIT compiler we could

1. inline the instruction implementation
2. forward instruction result to the next (without going through CPUState)

In emulation lingo, these are called *dynamic binary translators*.

Motivation

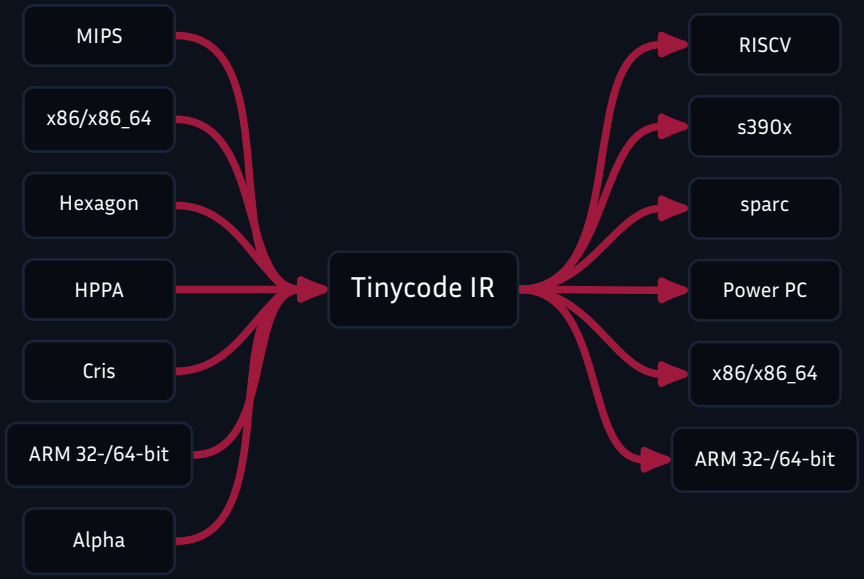
Enter QEMU

1st attempt: Pseudo C $\rightarrow$ TCG translator

2nd attempt: LLVM IR $\rightarrow$ TCG translator

Demo

Summary


MIPS
x86/x86_64
Hexagon
HPPA
Cris
ARM 32-/64-bit
Alpha
Tinycode IR
RISCV
s390x
sparc
Power PC
x86/x86_64
ARM 32-/64-bit

- Not in SSA-form
- Three variable types:
  1. Globals: mapped to parts of the CPU state
  2. Local temporaries: function scoped
  3. (Regular) temporaries: basic block scoped
- One instruction does one thing and thing only

```
ldr  r3, [fp, #-8]
```

```
ldr r3, [fp, #-8]
```

```
movi_i32 tmp1, -8
add_i32 tmp2, fp, tmp1
qemu_ld_i32 r3, tmp2
```

```
0x1000: jbe 0x2000
```

```
0x1000: jbe 0x2000
```

```
brcond_i64 tmp1,tmp2,leu,$L1
mov_i64 pc,$0x1006
exit_tb
set_label $L1
mov_i64 pc,$0x2000
exit_tb
```

- Similar to `IRBuilder` in LLVM
- C functions `tcg_gen_<op>_<width>(...)` that builds the Tinycode IR
- They emit code in a `TranslationBlock` (similar to `llvm::Function`)

Previously:

```
int add(int reg_a, int reg_b) {
    return reg_a + reg_b;
}
```

Now:

```
void add(TCGv_i32 ret,
         TCGv_i32 reg_a,
         TCGv_i32 reg_b) {
    tcg_gen_add_i32(ret, reg_a, reg_b);
}
```

Especially for complicated instructions (and if there's a lot of them)

```
void vacsh(uint64_t *RxxV, uint64_t *PeV, uint64_t RssV, uint64_t RttV) {
    for (int i = 0; i < 4; i++) {
        int xv = (int) fGETHALF(i,*RxxV);
        int sv = (int) fGETHALF(i,RssV);
        int tv = (int) fGETHALF(i,RttV);

        xv = xv + tv;
        sv = sv - tv;

        // Set result predicate
        fSETBIT(i*2,  *PeV,  (xv > sv));
        fSETBIT(i*2+1,*PeV,  (xv > sv));

        // Set result register
        fSETHALF(i,   *RxxV, fSATH(fMAX(xv,sv)));
    }
}
```

```c
void vacsh(TCGv_i64 RxxV, TCGv PeV, TCGv_i64 RssV, TCGv_i64 RttV)
{
    TCGv_i32 xv = tcg_temp_local_new_i32();
    TCGv_i32 sv = tcg_temp_local_new_i32();
    TCGv_i32 tv = tcg_temp_local_new_i32();
    tcg_gen_movi_i32(PeV, 0);
    for (int i = 0; i < 4; i++) {
        // Get xv
        TCGv_i64 tmp_0 = tcg_temp_new_i64();
        tcg_gen_sextract_i64(tmp_0, RxxV, i * 16, 16);
        tcg_gen_trunc_i64_tl(xv, tmp_0);
        tcg_temp_free_i64(tmp_0);

        // Get sv
        TCGv_i64 tmp_2 = tcg_temp_new_i64();
        tcg_gen_sextract_i64(tmp_2, RssV, i * 16, 16);
        tcg_gen_trunc_i64_tl(sv, tmp_2);
        tcg_temp_free_i64(tmp_2);

        // Get tv
        TCGv_i64 tmp_4 = tcg_temp_new_i64();
        tcg_gen_sextract_i64(tmp_4, RttV, i * 16, 16);
        tcg_gen_trunc_i64_tl(tv, tmp_4);
        tcg_temp_free_i64(tmp_4);

        // add xv, tv
        tcg_gen_add_i32(xv, xv, tv);

        // sub xv, tv
        tcg_gen_sub_i32(sv, sv, tv);
```

```
// Set predicate
int32_t qemu_tmp_0 = i * 2;
TCGv_i32 tmp_8 = tcg_temp_new_i32();
tcg_gen_setcond_i32(TCG_COND_GT, tmp_8, xv, sv);
tcg_gen_deposit_i32(PeV, PeV, tmp_8, qemu_tmp_0, 1);
tcg_temp_free_i32(tmp_8);

// Set predicate
int32_t qemu_tmp_1 = qemu_tmp_0 + 1;
TCGv_i32 tmp_9 = tcg_temp_new_i32();
tcg_gen_setcond_i32(TCG_COND_GT, tmp_9, xv, sv);
tcg_gen_deposit_i32(PeV, PeV, tmp_9, qemu_tmp_1, 1);
tcg_temp_free_i32(tmp_9);

// Compute max
TCGv_i32 tmp_10 = tcg_temp_new_i32();
tcg_gen_smax_i32(tmp_10, xv, sv);

// Saturate if needed
TCGv_i32 tmp_11 = tcg_temp_new_i32();
TCGv_i32 tmp_12 = tcg_temp_new_i32();
gen_sat_i32_ovfl(tmp_12, tmp_11, tmp_10, 16);
TCGv_i32 tmp_13 = tcg_temp_new_i32();
GET_USR_FIELD(USR_OVF, tmp_13);
tcg_gen_or_i32(tmp_13, tmp_13, tmp_12);
SET_USR_FIELD(USR_OVF, tmp_13);
tcg_temp_free_i32(tmp_13);
tcg_temp_free_i32(tmp_12);
tcg_temp_free_i32(tmp_10);

// Set result register
TCGv_i64 tmp_14 = tcg_temp_new_i64();
tcg_gen_ext_i32_i64(tmp_14, tmp_11);
tcg_temp_free_i32(tmp_11);
tcg_gen_deposit_i64(RxxV, RxxV, tmp_14, i * 16, 16);
```

```
        tcg_temp_free_i64(tmp_14);
    }
    tcg_temp_free_i32(xv);
    tcg_temp_free_i32(sv);
    tcg_temp_free_i32(tv);
}
```

```
        tcg_temp_free_i64(tmp_14);
    }
    tcg_temp_free_i32(xv);
    tcg_temp_free_i32(sv);
    tcg_temp_free_i32(tv);
}
```

Motivation

Enter QEMU

1st attempt: Pseudo C $\rightarrow$ TCG translator

2nd attempt: LLVM IR $\rightarrow$ TCG translator

Demo

Summary

Pseudo C functions

```
A2_add(RdV, in RsV, in RtV) {
    { RdV=RsV+RtV; }
}
```

Pseudo C functions

Flex/Bison Parser

```
A2_add(RdV, in RsV, in RtV) {
    { RdV=RsV+RtV; }
}
```

Pseudo C functions → Flex/Bison Parser → TCG code

```
A2_add(RdV, in RsV, in RtV) {
    { RdV=RsV+RtV; }
}
```

```
void A2_add(TCGv_i32 RdV,
            TCGv_i32 RsV,
            TCGv_i32 RtV) {
    TCGv_i32 tmp_0 = tcg_temp_new_i32();
    tcg_gen_add_i32(tmp_0, RsV, RtV);
    tcg_gen_mov_i32(RdV, tmp_0);
    tcg_temp_free_i32(tmp_0);
}
```

Pseudo C functions → Flex/Bison Parser → TCG code

```
A2_add(RdV, in RsV, in RtV) {
    { RdV=RsV+RtV; }
}
```

```
void A2_add(TCGv_i32 RdV,
            TCGv_i32 RsV,
            TCGv_i32 RtV) {
    TCGv_i32 tmp_0 = tcg_temp_new_i32();
    tcg_gen_add_i32(tmp_0, RsV, RtV);
    tcg_gen_mov_i32(RdV, tmp_0);
    tcg_temp_free_i32(tmp_0);
}
```

Pseudo C functions → Flex/Bison Parser → TCG code

```
A2_add(RdV, in RsV, in RtV) {
    { RdV=RsV+RtV; }
}
```

```
void A2_add(TCGv_i32 RdV,
            TCGv_i32 RsV,
            TCGv_i32 RtV) {
    TCGv_i32 tmp_0 = tcg_temp_new_i32();
    tcg_gen_add_i32(tmp_0, RsV, RtV);
    tcg_gen_mov_i32(RdV, tmp_0);
    tcg_temp_free_i32(tmp_0);
}
```

Pseudo C functions → Flex/Bison Parser → TCG code

```
A2_add(RdV, in RsV, in RtV) {
    { RdV=RsV+RtV; }
}
```

```
void A2_add(TCGv_i32 RdV,
            TCGv_i32 RsV,
            TCGv_i32 RtV) {
    TCGv_i32 tmp_0 = tcg_temp_new_i32();
    tcg_gen_add_i32(tmp_0, RsV, RtV);
    tcg_gen_mov_i32(RdV, tmp_0);
    tcg_temp_free_i32(tmp_0);
}
```

Pseudo C functions → Flex/Bison Parser → TCG code

```
A2_add(RdV, in RsV, in RtV) {
    { RdV=RsV+RtV; }
}
```

```
void A2_add(TCGv_i32 RdV,
            TCGv_i32 RsV,
            TCGv_i32 RtV) {
    TCGv_i32 tmp_0 = tcg_temp_new_i32();
    tcg_gen_add_i32(tmp_0, RsV, RtV);
    tcg_gen_mov_i32(RdV, tmp_0);
    tcg_temp_free_i32(tmp_0);
}
```

- ✓ Able to generate TCG for 1500 instructions

- ✓ Able to generate TCG for 1500 instructions
- ✗ Heavily tailored to Hexagon

Motivation

Enter QEMU

1st attempt: Pseudo C $\rightarrow$ TCG translator

2nd attempt: LLVM IR $\rightarrow$ TCG translator

Demo

Summary

Instead of:

Instead of:

Why not:

C functions

Instead of:

Why not:

Instead of:

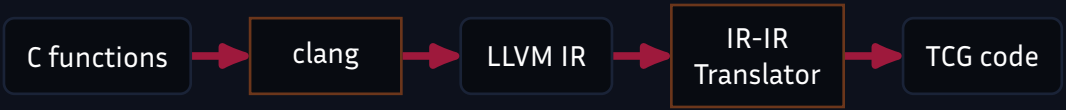
Why not:

Instead of:

Why not:

Instead of:

Why not:



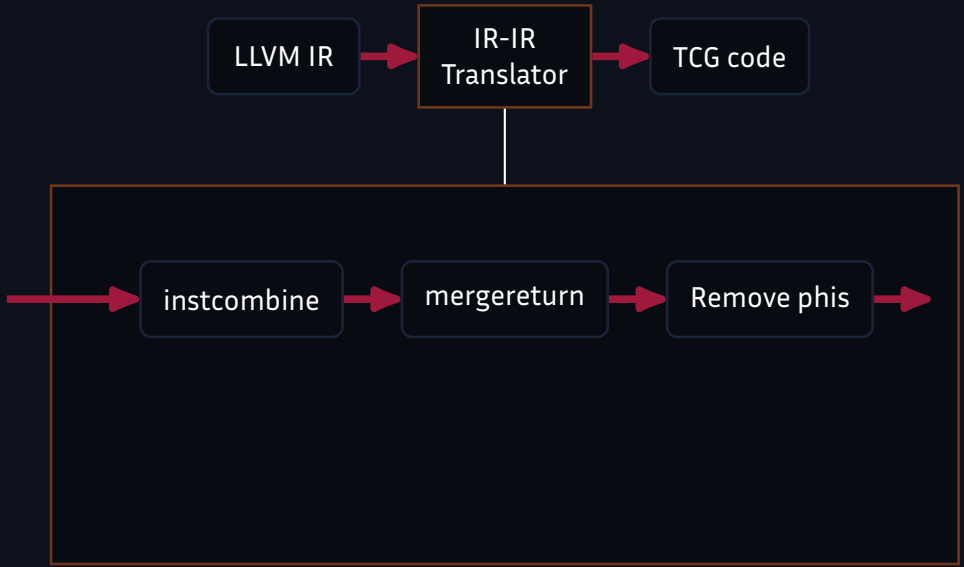
Regular LLVM Backend	Simple backend as a pass

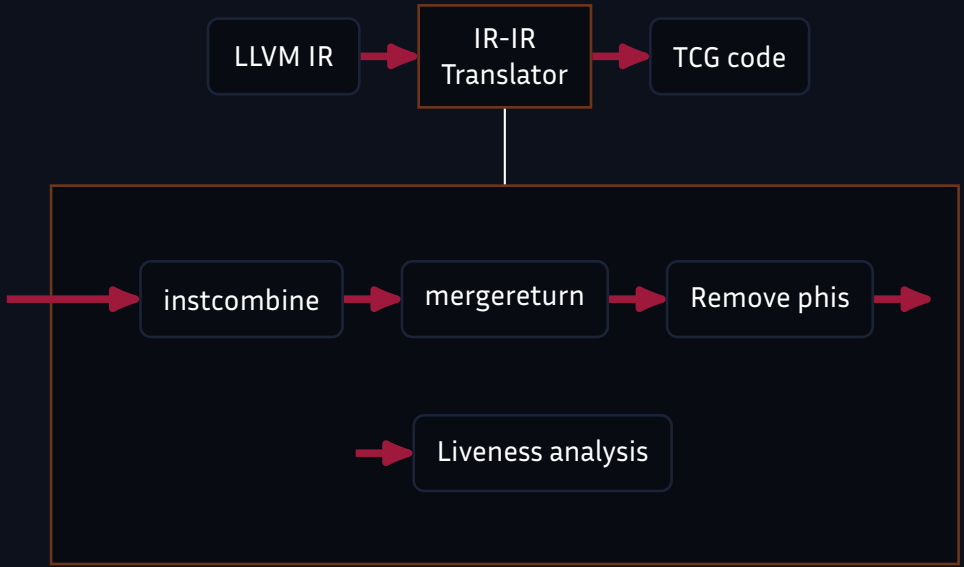
Regular LLVM Backend	Simple backend as a pass
✓ Free liveness analysis	× Need to do liveness analysis ourselves
✓ Free instruction selection	× No easy instruction selection

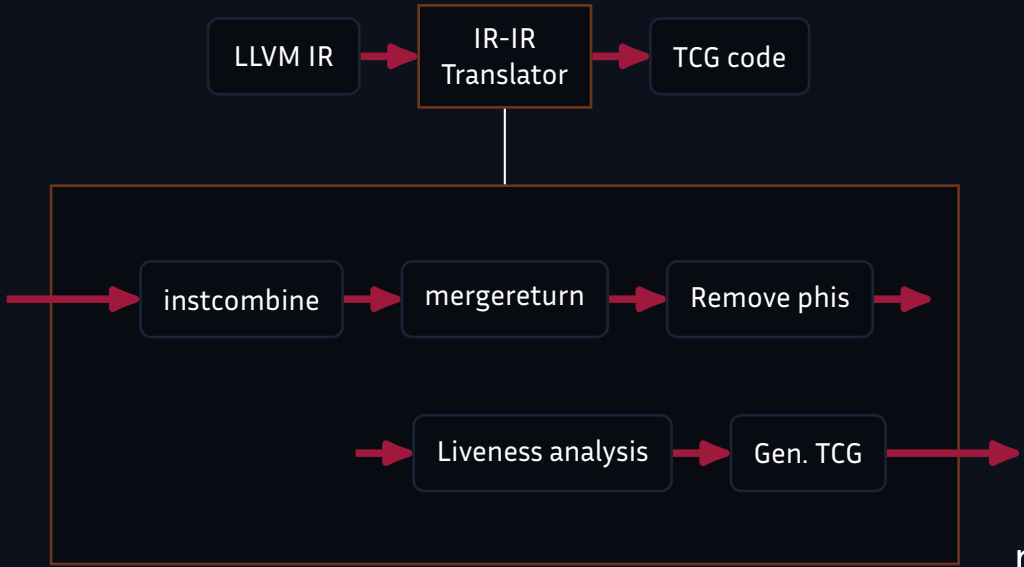
Regular LLVM Backend	Simple backend as a pass
✓ Free liveness analysis	× Need to do liveness analysis ourselves
✓ Free instruction selection	× No easy instruction selection
× Boilerplate	✓ Easier to implement
× We have no clear instruction format to emit	

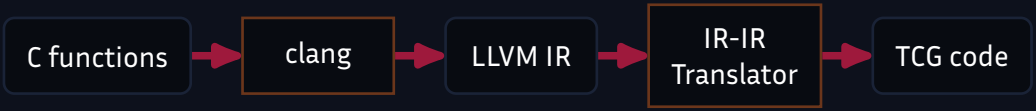

LLVM IR
IR-IR Translator
TCG code

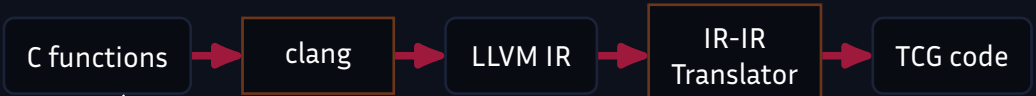

LLVM IR
IR-IR Translator
TCG code


LLVM IR
IR-IR Translator
TCG code
instcombine
mergereturn
Remove phis


LLVM IR
IR-IR Translator
TCG code
instcombine
mergereturn
Remove phis
Liveness analysis


LLVM IR
IR-IR Translator
TCG code
instcombine
mergereturn
Remove phis
Liveness analysis
Gen. TCG


C functions
clang
LLVM IR
IR-IR Translator
TCG code




```
int32_t A2_add(int32_t RsV,
               int32_t RtV) {
    int32_t RdV = 0;
    { RdV=RsV+RtV; }
    return RdV;
}
```

```
define i32 @A2_add(i32 %RsV, i32 %RtV) {
    %RdV = add i32 %RsV, %RtV
    ret i32 %RdV
}
```

C functions → clang → LLVM IR → IR-IR Translator → TCG code

```
int32_t A2_add(int32_t RsV,
               int32_t RtV) {
    int32_t RdV = 0;
    { RdV=RsV+RtV; }
    return RdV;
}
```

```
define i32 @A2_add(i32 %RsV, i32 %RtV) {
    %RdV = add i32 %RsV, %RtV
    ret i32 %RdV
}
```

C functions → clang → LLVM IR → IR-IR Translator → TCG code

```
int32_t A2_add(int32_t RsV,
               int32_t RtV) {
    int32_t RdV = 0;
    { RdV=RsV+RtV; }
    return RdV;
}
```

```
void A2_add(TCGv_i32 ret,
            TCGv_i32 v_0,
            TCGv_i32 v_1) {
    TCGv_i32 add_2 = tcg_temp_new_i32();
    tcg_gen_add_i32(add_2, v_1, v_0);
    tcg_gen_mov_i32(ret, add_2);
    tcg_temp_free_i32(add_2);
}
```

LLVM IR

```
define i32 @my_inst() {
    %1 = ...

    ; ...


    br label %bb
  bb:

    ; ...


    ret i32 ...
}
```

LLVM IR

```
define i32 @my_inst() {
  %1 = ...

  ; ...


  br label %bb
bb:

  ; ...


  ret i32 ...
}
```

LLVM IR

```llvm
define i32 @my_inst() {
  %1 = ...

  ; ...


  br label %bb
bb:

  ; ...


  ret i32 ...
}
```

TCG: Temp

```tcg
void my_inst(TCGv_i32 ret) {
    TCGv_i32 v_1 = tcg_temp_new_i32();

    // ...


    tcg_gen_br(bb);
  tcg_gen_set_label(bb);

    // ...


    tcg_gen_mov_i32(ret, ...);
}
```

LLVM IR

```
define i32 @my_inst() {
    %1 = ...

    ; ...


    br label %bb
  bb:

    ; ...


    ret i32 ...
}
```

TCG: Temp

```
void my_inst(TCGv_i32 ret) {
    TCGv_i32 v_1 = tcg_temp_new_i32();

    // ...


    tcg_gen_br(bb);
  tcg_gen_set_label(bb);

    // ...


    tcg_gen_mov_i32(ret, ...);
}
```

LLVM IR

```
define i32 @my_inst() {
  %1 = ...

  ; ...


  br label %bb
bb:

  ; ...


  ret i32 ...
}
```

TCG: Temp

```
void my_inst(TCGv_i32 ret) {
  TCGv_i32 v_1 = tcg_temp_new_i32();

  // ...


  tcg_gen_br(bb);
tcg_gen_set_label(bb);

  // ...


  tcg_gen_mov_i32(ret, ...);
}
```

TCG: Local temp

```
void my_inst(TCGv_i32 ret) {
  TCGv_i32 v_1 = tcg_temp_local_new_i32();

  // ...


  tcg_gen_br(bb);
tcg_gen_set_label(bb);

  // ...


  tcg_gen_mov_i32(ret, ...);
}
```

LLVM IR

```
define i32 @my_inst() {
  %1 = ...

  ; ...


  br label %bb
bb:

  ; ...


  ret i32 ...
}
```

TCG: Temp

```
void my_inst(TCGv_i32 ret) {
  TCGv_i32 v_1 = tcg_temp_new_i32();

  // ...

  tcg_temp_free_i32(v_1);
  tcg_gen_br(bb);
tcg_gen_set_label(bb);

  // ...


  tcg_gen_mov_i32(ret, ...);
}
```

TCG: Local temp

```
void my_inst(TCGv_i32 ret) {
  TCGv_i32 v_1 = tcg_temp_local_new_i32();

  // ...


  tcg_gen_br(bb);
tcg_gen_set_label(bb);

  // ...


  tcg_gen_mov_i32(ret, ...);
}
```

LLVM IR

```
define i32 @my_inst() {
  %1 = ...

  ; ...


  br label %bb
bb:

  ; ...


  ret i32 ...
}
```

TCG: Temp

```
void my_inst(TCGv_i32 ret) {
  TCGv_i32 v_1 = tcg_temp_new_i32();

  // ...

  tcg_temp_free_i32(v_1);
  tcg_gen_br(bb);
tcg_gen_set_label(bb);

  // ...


  tcg_gen_mov_i32(ret, ...);
}
```

TCG: Local temp

```
void my_inst(TCGv_i32 ret) {
  TCGv_i32 v_1 = tcg_temp_local_new_i32();

  // ...


  tcg_gen_br(bb);
tcg_gen_set_label(bb);

  // ...

  tcg_temp_free_i32(v_1);
  tcg_gen_mov_i32(ret, ...);
}
```

LLVM IR

```llvm
define i32 @my_inst() {
  %1 = ...

  ; ...


  br label %bb
bb:

  ; ...


  ret i32 ...
}
```

TCG: Temp

```
void my_inst(TCGv_i32 ret) {
  TCGv_i32 v_1 = tcg_temp_new_i32();

  // ...

  tcg_temp_free_i32(v_1);
  tcg_gen_br(bb);
tcg_gen_set_label(bb);

  // ...


  tcg_gen_mov_i32(ret, ...);
}
```

TCG: Local temp

```
void my_inst(TCGv_i32 ret) {
  TCGv_i32 v_1 = tcg_temp_local_new_i32();

  // ...


  tcg_gen_br(bb);
tcg_gen_set_label(bb);

  // ...

  tcg_temp_free_i32(v_1);
  tcg_gen_mov_i32(ret, ...);
}
```

We want:

1. Choose correct type for variable
2. Free as soon as possible

LLVM IR

```llvm
define i32 @my_inst() {
  %1 = ...

  ; ...


  br label %bb
bb:

  ; ...


  ret i32 ...
}
```

TCG: Temp

```
void my_inst(TCGv_i32 ret) {
  TCGv_i32 v_1 = tcg_temp_new_i32();

  // ...

  tcg_temp_free_i32(v_1);
  tcg_gen_br(bb);
tcg_gen_set_label(bb);

  // ...


  tcg_gen_mov_i32(ret, ...);
}
```

TCG: Local temp

```
void my_inst(TCGv_i32 ret) {
  TCGv_i32 v_1 = tcg_temp_local_new_i32();

  // ...


  tcg_gen_br(bb);
tcg_gen_set_label(bb);

  // ...

  tcg_temp_free_i32(v_1);
  tcg_gen_mov_i32(ret, ...);
}
```

We want:

1. Choose correct type for variable
2. Free as soon as possible

Solution: Liveness analysis

```
add <8 x i8> ...
```

```
add <8 x i8> ...
```

→

```
tcg_gen_vec_add8_i64(...)
```

Not easily vectorizable

```
int64_t A2_vaddub(int64_t RssV, int64_t RttV) {
    int64_t RddV = 0;
    for (int i = 0; i < 8; i++) {
```

Not easily vectorizable

```
int64_t A2_vaddub(int64_t RssV, int64_t RttV) {
    int64_t RddV = 0;
    for (int i = 0; i < 8; i++) {
        uint8_t byte1 = (RssV >> (i*8)) & 0xff;
        uint8_t byte2 = (RttV >> (i*8)) & 0xff;
```

Not easily vectorizable

```
int64_t A2_vaddub(int64_t RssV, int64_t RttV) {
    int64_t RddV = 0;
    for (int i = 0; i < 8; i++) {
        uint8_t byte1 = (RssV >> (i*8)) & 0xff;
        uint8_t byte2 = (RttV >> (i*8)) & 0xff;

        uint8_t sum = byte1 + byte2;
```

Not easily vectorizable

```
int64_t A2_vaddub(int64_t RssV, int64_t RttV) {
    int64_t RddV = 0;
    for (int i = 0; i < 8; i++) {
        uint8_t byte1 = (RssV >> (i*8)) & 0xff;
        uint8_t byte2 = (RttV >> (i*8)) & 0xff;

        uint8_t sum = byte1 + byte2;

        RddV = (RddV & ~(0x0ffLL << (i*8))) |
            (((uint64_t)(sum & 0x0ffLL)) << (i*8));
    }
    return RddV;
}
```

We want a pass to transform

```
int64_t A2_vaddub(int64_t RssV, int64_t RttV) {
    int64_t RddV = 0;
    for (int i = 0; i < 8; i++) {
        uint8_t byte1 = (RssV >> (i*8)) & 0xff;
        uint8_t byte2 = (RttV >> (i*8)) & 0xff;

        uint8_t sum = byte1 + byte2;

        RddV = (RddV & ~(0x0ffLL << (i*8))) |
            (((uint64_t)(sum & 0x0ffLL)) << (i*8));
    }
    return RddV;
}
```

We want a pass to transform

```c
int64_t A2_vaddub(int64_t RssV, int64_t RttV) {
    int64_t RddV = 0;
    for (int i = 0; i < 8; i++) {
        uint8_t byte1 = (RssV >> (i*8)) & 0xff;
        uint8_t byte2 = (RttV >> (i*8)) & 0xff;

        uint8_t sum = byte1 + byte2;

        RddV = (RddV & ~(0x0ffLL << (i*8))) |
            (((uint64_t)(sum & 0x0ffLL)) << (i*8));
    }
    return RddV;
}
```

```c
int64_t A2_vaddub(uint8_t *restrict RssV,
                  uint8_t *restrict RttV) {
    uint8_t RddV[8];
    for (int i = 0; i < 8; i++) {
        RddV[i] = RssV[i] + RttV[i];
    }
```

We want a pass to transform

```c
int64_t A2_vaddub(int64_t RssV, int64_t RttV) {
    int64_t RddV = 0;
    for (int i = 0; i < 8; i++) {
        uint8_t byte1 = (RssV >> (i*8)) & 0xff;
        uint8_t byte2 = (RttV >> (i*8)) & 0xff;

        uint8_t sum = byte1 + byte2;

        RddV = (RddV & ~(0x0ffLL << (i*8))) |
            (((uint64_t)(sum & 0x0ffLL)) << (i*8));
    }
    return RddV;
}
```

```c
int64_t A2_vaddub(uint8_t *restrict RssV,
                  uint8_t *restrict RttV) {
    uint8_t RddV[8];
    for (int i = 0; i < 8; i++) {
        RddV[i] = RssV[i] + RttV[i];
    }

    return ((uint64_t)RddV[0]) |
           ((uint64_t)RddV[1] << 8 * 1) |
           ((uint64_t)RddV[2] << 8 * 2) |
           ((uint64_t)RddV[3] << 8 * 3) |
           ((uint64_t)RddV[4] << 8 * 4) |
           ((uint64_t)RddV[5] << 8 * 5) |
           ((uint64_t)RddV[6] << 8 * 6) |
           ((uint64_t)RddV[7] << 8 * 7);
}
```

```
define i64 @A2_vaddub(i8* %0, i8* %1) {
```

```
define i64 @A2_vaddub(i8* %0, i8* %1) {
    %3 = bitcast i8* %0 to <8 x i8>*
    %4 = load <8 x i8>, <8 x i8>* %3, align 1
    %5 = bitcast i8* %1 to <8 x i8>*
    %6 = load <8 x i8>, <8 x i8>* %5, align 1
```

```
define i64 @A2_vaddub(i8* %0, i8* %1) {
    %3 = bitcast i8* %0 to <8 x i8>*
    %4 = load <8 x i8>, <8 x i8>* %3, align 1
    %5 = bitcast i8* %1 to <8 x i8>*
    %6 = load <8 x i8>, <8 x i8>* %5, align 1

    %7 = add <8 x i8> %6, %4

    ; ...
}
```

Motivation

Enter QEMU

1st attempt: Pseudo C $\rightarrow$ TCG translator

2nd attempt: LLVM IR $\rightarrow$ TCG translator

Demo

Summary

Motivation

Enter QEMU

1st attempt: Pseudo C $\rightarrow$ TCG translator

2nd attempt: LLVM IR $\rightarrow$ TCG translator

Demo

Summary

- LLVM + QEMU $\rightarrow$ fast JITing emulator

- LLVM + QEMU $\rightarrow$ fast JITing emulator
- Currently:
  - Emit TCG for 1500 instructions

- LLVM + QEMU $\rightarrow$ fast JITing emulator
- Currently:
  - Emit TCG for 1500 instructions
  - Work ongoing with vectorization, constant expression

- LLVM + QEMU $\rightarrow$ fast JITing emulator
- Currently:
  - Emit TCG for 1500 instructions
  - Work ongoing with vectorization, constant expression
- Plan to open source/upstream :)

Get in touch at:

info@rev.ng

Subscribe to our newsletter:

https://rev.ng/newsletter.html