KVM Forum
November 1, 2019

# QEMU™ for Qualcomm® Hexagon™

## Automatic Translation of VLIW DSP Instructions to Tiny Code

**Niccolò Izzo**
*rev.ng*
**L Taylor Simpson**
*Qualcomm Innovation Center, Inc.*

# About rev.ng

A Milan-based startup founded 2 years ago
by two researchers from Politecnico di Milano

Key business areas:

- Static and dynamic binary translation
- Compilation and program analysis techniques
- Architecture-independent decompiler (binary to C)

Niccolò Izzo
MSc Computer Science and Engineering (cum laude) – 2017
Pursuing PhD at Politecnico di Milano

Master Thesis on Rowhammer
Maintainer of LineageOS for MicroG
Twitter: @n1zzo Email: nizzo@rev.ng Callsign: IU2KIN

# About Qualcomm

Qualcomm invents breakthrough technologies that transform how the world connects, computes, and communicates. When we connected the phone to the Internet, the mobile revolution was born. Today, our inventions are the foundation for life-changing products, experiences, and industries.

L Taylor Simpson
Sr. Director, Engineering
LLVM compiler and tools team
Qualcomm Innovation Center, Inc.
PhD Computer Science – Rice University
tsimpson@quicinc.com

# QEMU Hexagon

## Overview

- Introduction to Hexagon
- Introduction to QEMU
  - Tiny Code Generator (TCG)
- Challenges
- Automated TCG generation
    - Qualcomm approach
    - rev.ng approach
- Status and next steps
- Conclusion

# Introduction to Hexagon

# Introduction to Hexagon

## Very Long Instruction Word Digital Signal Processor (VLIW DSP)

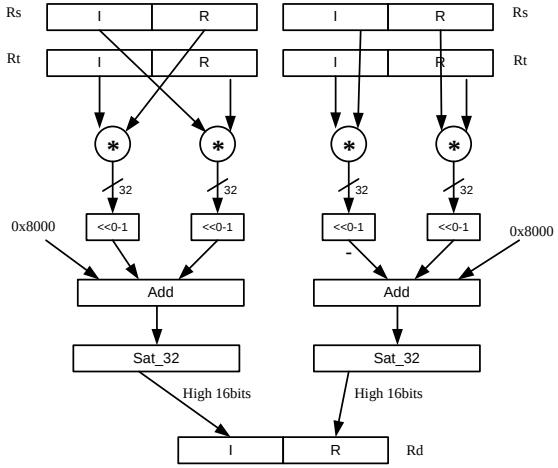
### Example from inner loop of FFT: Executing 29 “simple RISC ops” in 1 cycle

64-bit Load and

64-bit Store with post-update addressing

```
{ 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
```

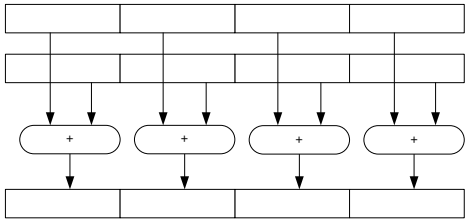
Complex multiply with round and saturation




Zero-overhead loops

- Dec count
- Compare
- Jump top

Vector 4x16-bit Add



# Introduction to QEMU

# Introduction to QEMU

- qemu.org
  - Generic and open source machine emulator and virtualizer
  - Code translation drives fast off-target simulation
- Operating modes
  - User mode
  - System mode
  - Virtualization
- Trace-based translator
  - Unit of translation is a translation block
  - Target instructions are translated to TCG ops
  - TCG ops are then transformed into host instructions
  - Translate once, execute many times

# Introduction to QEMU

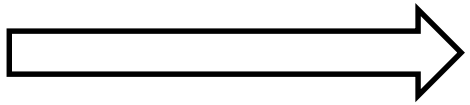
## Tiny Code Generator (TCG)

- TCG operators `tcg_gen_<op>[i]_<size>`

  <op> is the operation (e.g., add)
  [i] indicates immediate instead of register (e.g., addi)
  <size> is the size of TCG registers (usually use tl shorthand)
  Example: `tcg_gen_add_tl`

- From x86 assembly instruction to tiny code instructions

```
0x1000: call 0 x2000
0x1005:
```

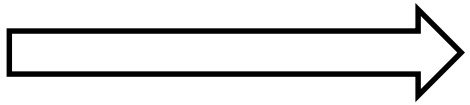


```
sub_i32 tmp0 ,esp , $0x4
qemu_st_i32 $0x1005 ,tmp0 ,leul ,0
mov_i32 esp , tmp0
movi_i32 eip , $0x2000
```

# Introduction to QEMU


- Goal: Create qemu-hexagon
- Translate binary Hexagon packets to TCG

```
{ 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
```



TCG

# Challenges

# Challenges

- Packet semantics
  - Behavior is NOT the same as executing instructions sequentially

```
{ r0 = r1; r1 = r0 }    // Swap r0 and r1
```

  - Dual jumps → Only one is executed

```
{ if (p0) jump:nt <foo>; jump <bar> }
```

  - Dual stores → Stores are serialized

```
{ memw(r3+#0) = r5; memb(r3+#0) = r4 }
```

  - .new

```
{ if (!p0.new) r0=#13; p0=cmp.eq(r0,#4) }
```

  - Multiple predicate definitions → and them together

```
{ p0=cmp.eq(r0,r1); p0=cmp.eq(r2,r3) }
```

  - Precise interrupts and exceptions → All instructions commit or none commit

- Over 2,000 user mode instructions!

# Challenges

## Implementation

QEMU executes tiny code instructions in sequential order
To preserve semantics, we have to

- Reorder instructions to solve dependencies
- Use temporary register set for .new accesses
- Commit at the end of packet to actual registers
- Commit only if no exception occur

# Automated Instruction Generation

*QTI approach*

# QEMU “helper”

- QEMU uses “helpers” to call function from TCG
- Each helper has 3 parts
- Generated via Python

**Hexagon Instruction**

```
Tag:        A2_add
Semantics:  "{RdV=RsV+RtV;}"
```

**Prototype**

```
DEF_HELPER_3(A2_add, s32, env, s32, s32)
```

**Generate call**

```
{
    DECL_RREG_d(TCGv, RdV, RdN, 0, 0);
    DECL_RREG_s(TCGv, RsV, RsN, 1, 0);
    DECL_RREG_t(TCGv, RtV, RtN, 2, 0);
    READ_RREG_s(RsV, RsN);
    READ_RREG_t(RtV, RtN);
    fWRAP_A2_add(gen_helper_A2_add(RdV, cpu_env, RsV, RtV);,
                                    { RdV=RsV+RtV;})
    WRITE_RREG(RdN,RdV);
    FREE_REG_d(RdV);
    FREE_REG_s(RsV);
    FREE_REG_t(RtV);
}
```

**Implementation**

```
int32_t HELPER(A2_add)(CPUHexagonState *env, int32_t RsV, int32_t RtV)
{
    uint32_t slot = 4; slot = slot;
    int32_t RdV = 0;
    { RdV=RsV+RtV;}
    return RdV;
}
```

# QEMU “helper”

- Advantage
  - Very quickly implement all instructions
  - Same semantics as hexagon-sim
- Disadvantages
  - Function call overhead
  - Barrier to TCG optimization

**Generate**

```
{
    DECL_RREG_d(TCGv, RdV, RdN, 0, 0);
    DECL_RREG_s(TCGv, RsV, RsN, 1, 0);
    DECL_RREG_t(TCGv, RtV, RtN, 2, 0);
    READ_RREG_s(RsV, RsN);
    READ_RREG_t(RtV, RtN);
    fWRAP_A2_add(gen_helper_A2_add(RdV, cpu_env, RsV, RtV);,
                                { RdV=RsV+RtV;})
    WRITE_RREG(RdN,RdV);
    FREE_REG_d(RdV);
    FREE_REG_s(RsV);
    FREE_REG_t(RtV);
}
```

**Override**

```
#define fWRAP_A2_add(GENHLPR, SHORTCODE) \
    tcg_gen_add_tl(RdV, RsV, RtV);
```

# Automated Instruction Generation

rev.ng approach

# Automated TCG Generation

Hexagon instructions are described in the docs with C-like snippets, e.g.:

```
Rx+=sub(Rt,Rs)
```

*Assembly syntax*

```
Rx=Rx + (Rt - Rs);
```

*Pseudo-code*

Can we **translate** these snippets into QEMU TCG generation code?

# Automated TCG Generation

Hexagon instructions are described in the docs with C-like snippets, e.g.:

```
Rx+=sub(Rt,Rs)
```

*Assembly syntax*

```
Rx=Rx + (Rt - Rs);
```

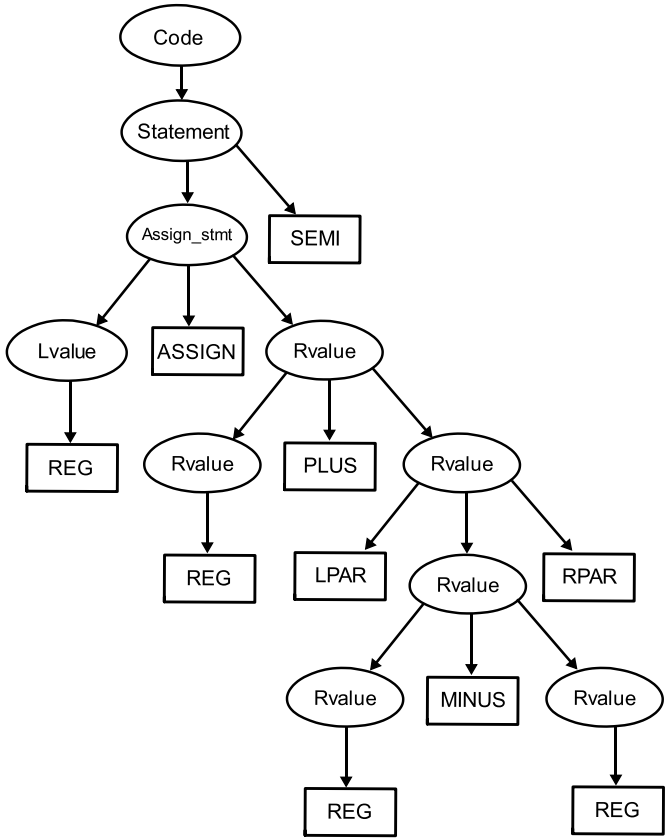
*Pseudo-code*

Can we **translate** these snippets into QEMU TCG generation code?

We used flex + bison to achieve **exactly that**

# Automated TCG Generation

```
Rx=Rx + (Rt - Rs);
```



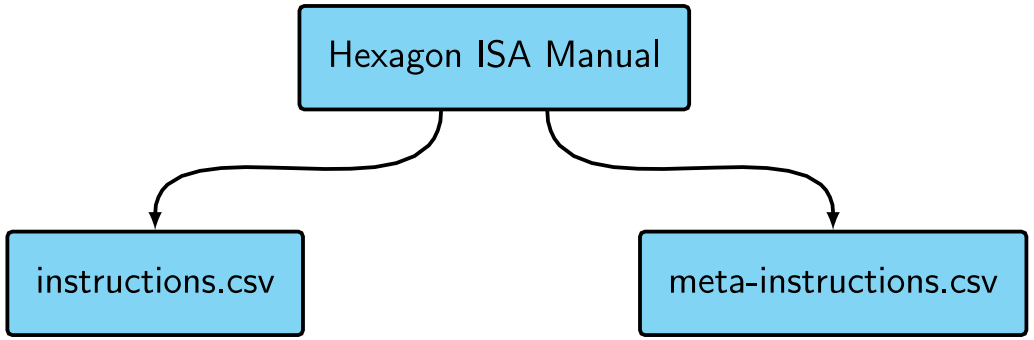

*flex-bison syntax tree*

```
regs_t function_536(DisasContext * dc ,
uint32_t x,
uint32_t t,
uint32_t s) {
    regs_t regs = { 0 };
    TCGv_i32 tmp_0 = tcg_temp_new_i32 ();
    tcg_gen_sub_i32(tmp_0 , GPR[t], GPR[s]);
    TCGv_i32 tmp_1 = tcg_temp_new_i32 ();
    tcg_gen_add_i32(tmp_1 , GPR[x], tmp_0 );
    tcg_temp_free_i32(tmp_0 );
    tcg_gen_mov_tl(GPR_new[x], tmp_1 );
    SET_USED_REG(regs , x);
    tcg_temp_free_i32(tmp_1 );
    return regs;
}
```

*TCG generation function*

# Automated TCG Generation

Two *CSV* are extracted from the ISA manual containing the instruction **encodings** and **semantic descriptions**



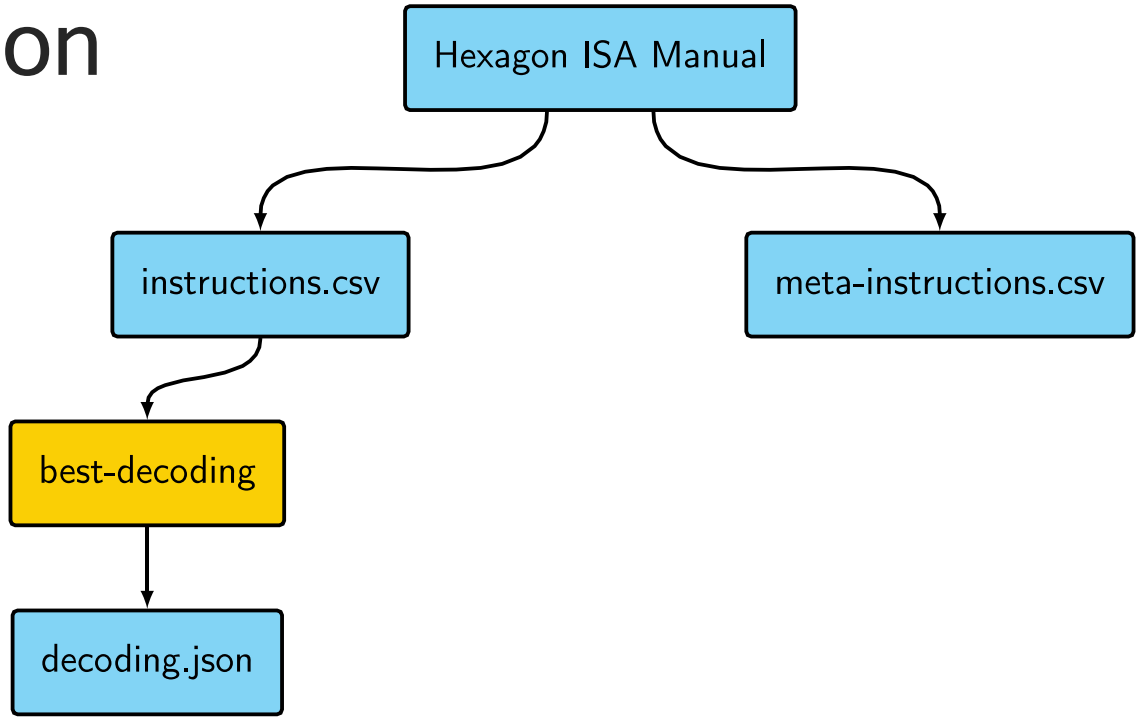


N

# Automated TCG Generation

Two *CSV* are extracted from the ISA manual containing the instruction **encodings** and **semantic descriptions**

An **optimized decoder tree** is generated from the encodings

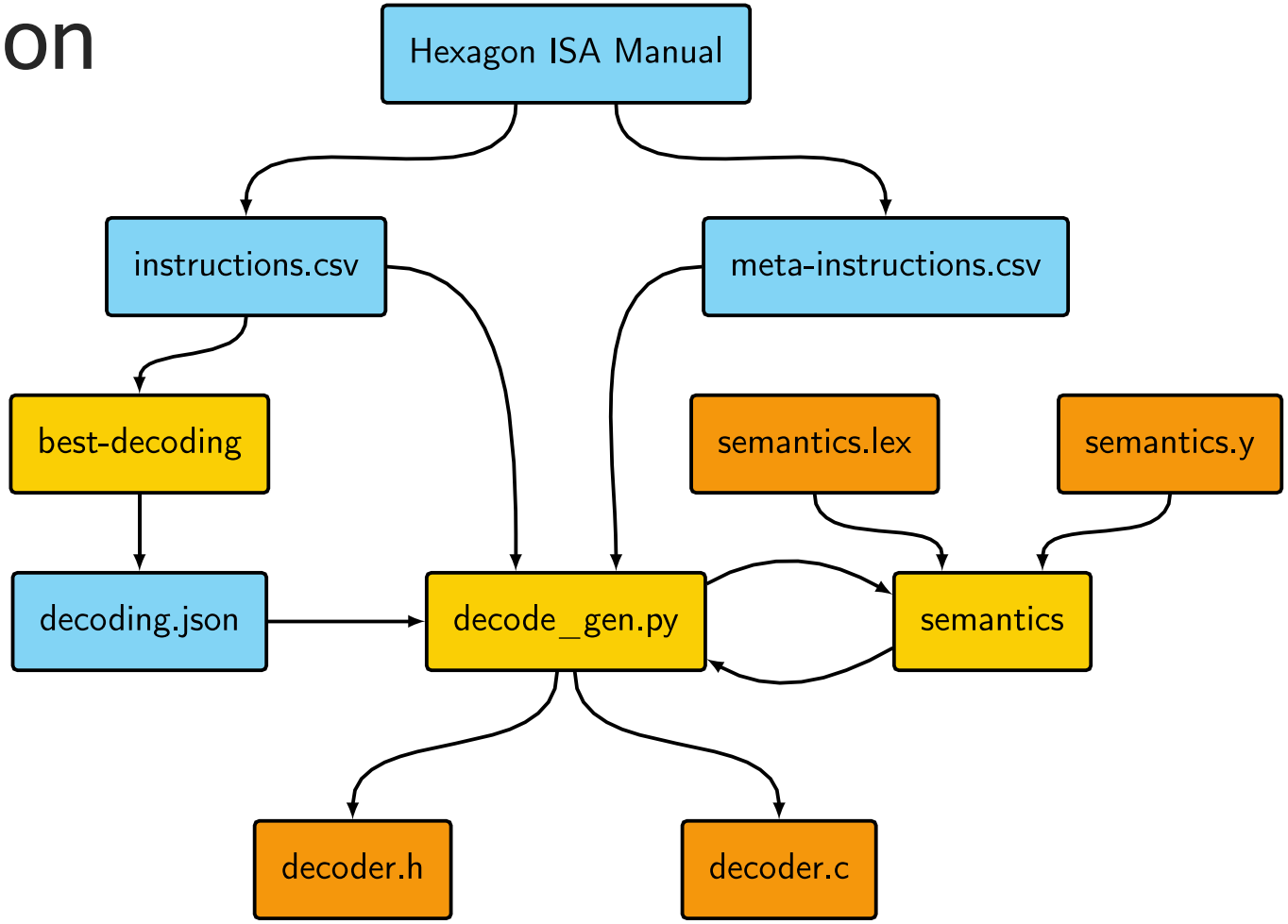


# Automated TCG Generation

Two *CSV* are extracted from the ISA manual containing the instruction **encodings** and **semantic descriptions**

An **optimized decoder tree** is generated from the encodings

The pseudocode snippets are fed into a flex-bison **generated parser**



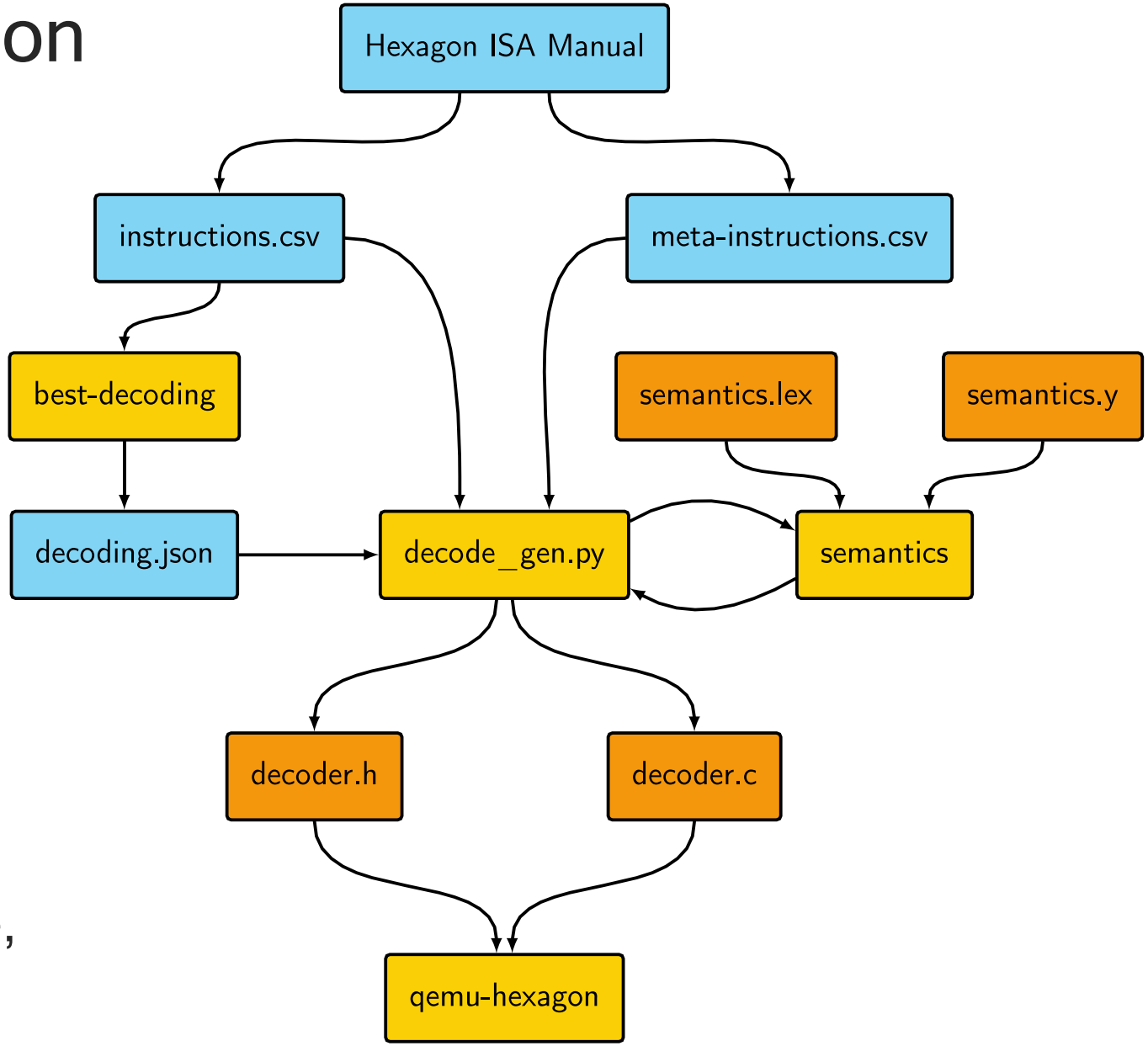
# Automated TCG Generation

Two *CSV* are extracted from the ISA manual containing the instruction **encodings** and **semantic descriptions**

An **optimized decoder tree** is generated from the encodings

The pseudocode snippets are fed into a flex-bison **generated parser**

The resulting functions are used to generate a source and header file, which are **compiled into QEMU**



# Status and Limitations

# Status

- Up to 28X faster than hexagon-sim
- Linux user space completed
- Angel/semi-hosting
- Extensively tested
- Code available
  - rev.ng implementation https://github.com/revng/qemu-hexagon
  - Qualcomm implementation https://github.com/quic/qemu

# Next Steps

- Short term
  - Merge rev.ng and QTI implementations
  - Community review
  - Merge upstream

- Long term
  - Tighter integration with Hexagon LLVM
  - System mode
  - Debug Hexagon programs with LLVM debugger (LLDB)

# Demo & Conclusion

# Conclusion

- VLIW semantics create interesting challenges
- Large number of instructions requires automated generation
- Code generator can be useful for adding support for new complex architectures
- Hexagon programs execute up to 28X faster on QEMU than current simulator

Qualcomm

# Thank you!

Follow us on:

For more information, visit us at:

www.qualcomm.com & www.qualcomm.com/blog