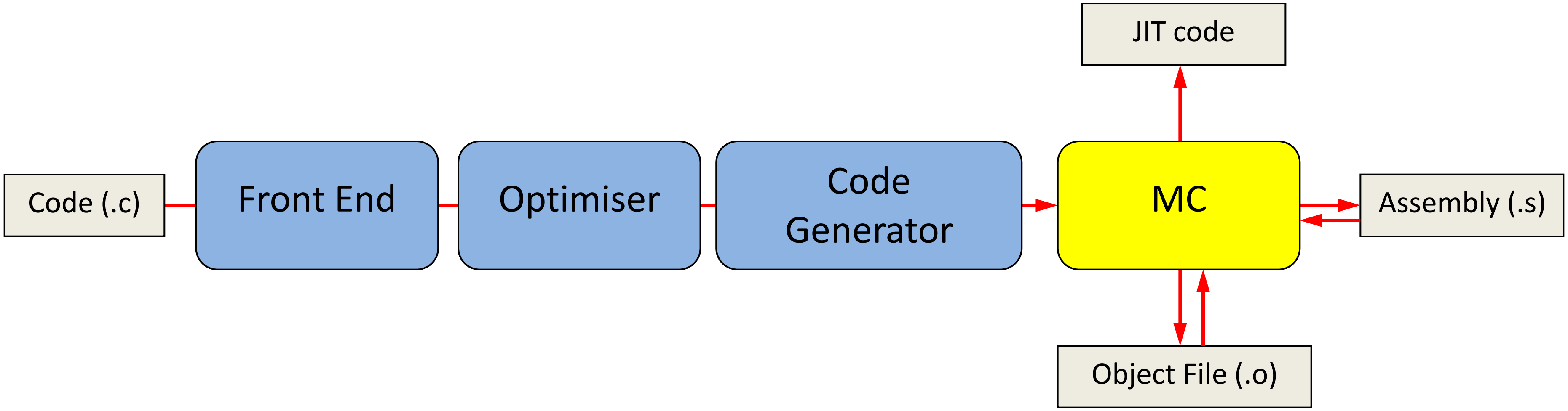
Guaranteeing the Correctness of MC for ARM

Richard Barton

The MC Layer

- The Machine Code layer is a single location for Target specific information for representing machine instructions.
- Multi-platform
- Multi-directional
- Table-generated

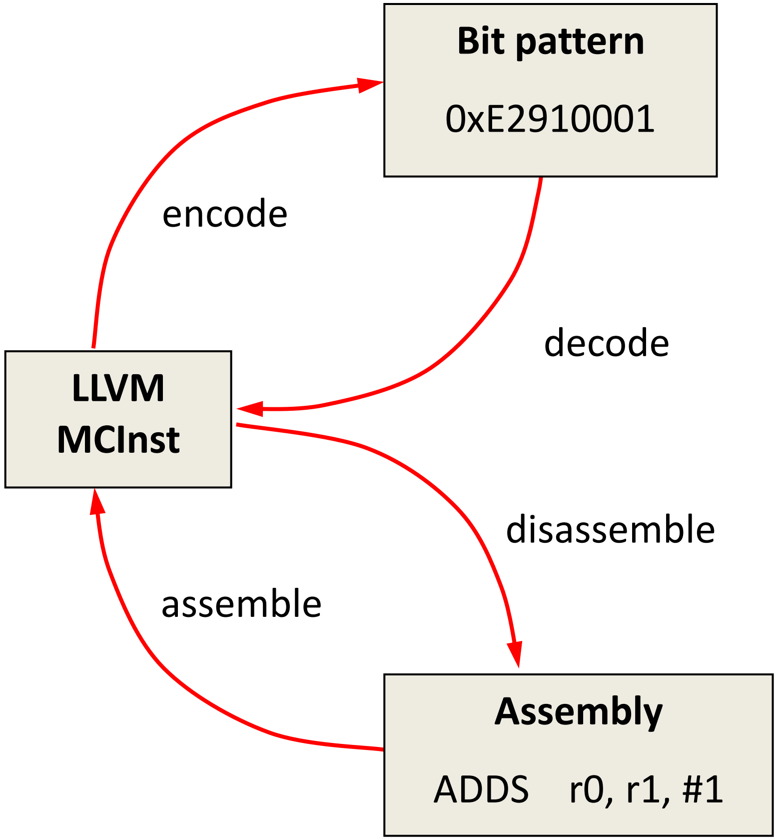


Definition of the problem

- The MC layer is a cornerstone of LLVM.
- It is used by compilers, assemblers, debuggers and JIT compilers.
- We need this component to be trustworthy in order for great tools to be built with it.
- How can we guarantee the correctness that we need?

What is the functionality of MC?

- Decode:
  - interpret instruction bit patterns
- Encode
  - output instruction bit patterns
- Assemble
  - Interpret instruction assembly
- Disassemble
  - output instruction assembly
- We will not be testing the interface between LLVM and MC.



Our Strategy for solution

- Exhaustive checking of the problem space against a known correct implementation with the same functionality.
- We think that our strategy is architecture agnostic.

Our Strategy for solution

- Exhaustive checking of the problem space against a known correct implementation with the same functionality.
- We think that our strategy is architecture agnostic.
- What do we mean by exhaustive?

Our Strategy for solution

- Exhaustive checking of the problem space against a known correct implementation with the same functionality.
- We think that our strategy is architecture agnostic.
- What do we mean by exhaustive?
- What do we mean by the whole problem space?

What is the problem space?

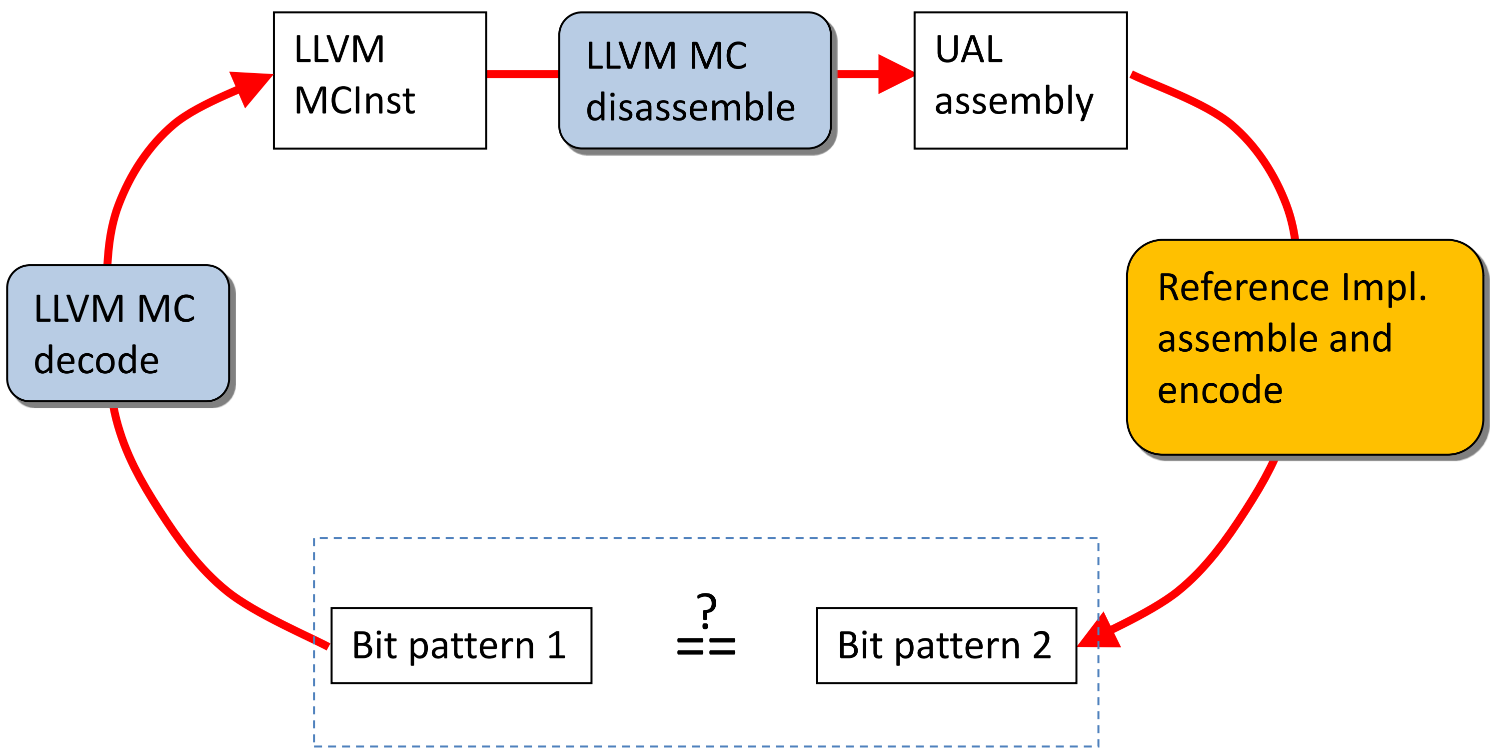
- Problem space has 4 dimensions
  - Instruction encoding
    - e.g. 0 – 2^32 for ARM
  - Instruction set
    - e.g. ARM vs. Thumb, x86_32 vs. x86_64, ...
  - Architecture variant
    - e.g. ARMv6 vs. ARMv7, MIPS IV vs. MIPS V, ...
  - MC Functionality
    - 4 possible values {encode, decode, disassemble, assemble}

What is the problem space for ARM?

- Test space has 4 dimensions
  - Instruction encoding
    - 2^32 possible values
  - Instruction set
    - 2 possible values: ARM, Thumb
  - Architecture variant
    - 28 pre-ARMv7 architecture + extensions combinations
    - 176 ARMv7 architecture + extensions combinations
    - 204 possible values
  - MC Functionality
    - 4 possible values {encode, decode, disassemble, assemble}
- The whole test space has O(7 trillion) points
  - 7,009,386,627,072 points

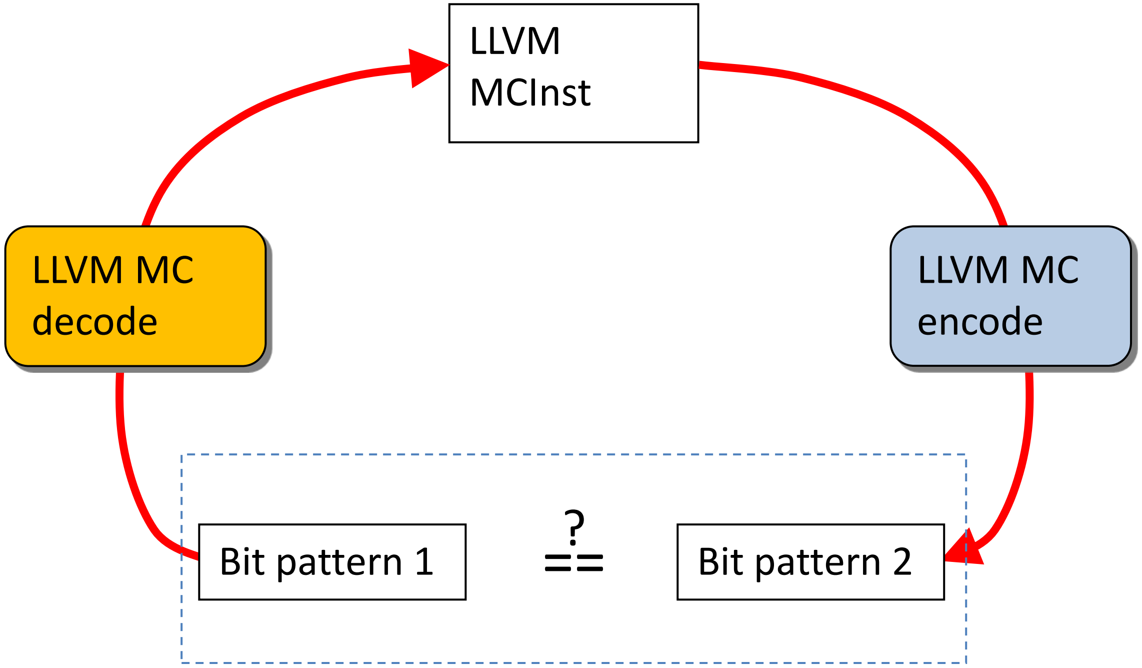
Testing decode and disassemble

- The below diagram illustrates one chain of transformations that test two MC functions.
- The ‘golden’ components are considered bug free.



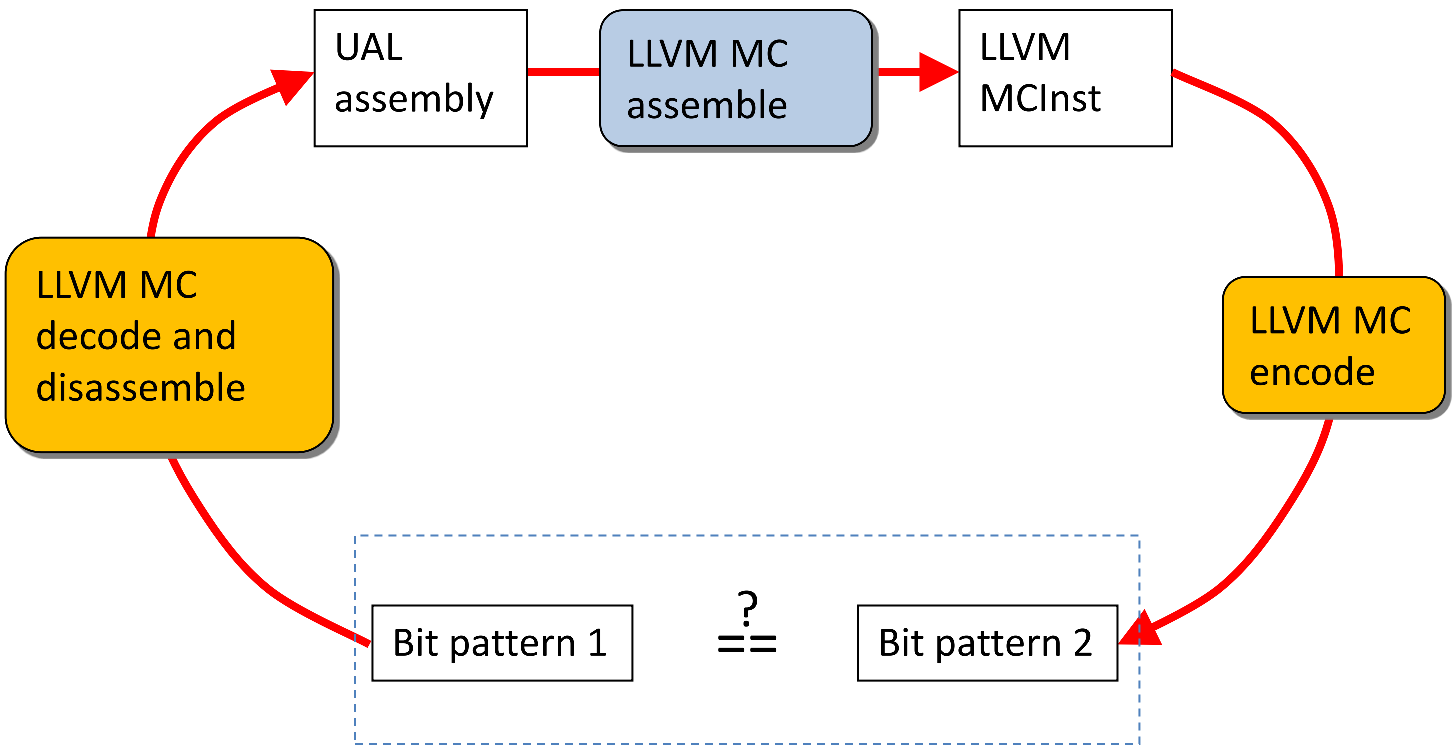
Testing encode

- Now that we have found and fixed all the bugs in MC's decoder, it becomes 'golden'
- We can use it to test encoding.



Testing assemble

- Testing assembling is similar to testing disassembling.
- We iterate over the instruction encodings in each case as they are easier to enumerate than UAL strings.



Implementation Details

- A test suite needs a name.
- We have named this test suite the MC Hammer Tests

lcodec: our reference implementation

- A set of libraries that provide an abstraction of instruction encodings. It can be regarded as an implementation of the Unified Assembler Language providing a unified view of several similar instruction sets.
- Handles encode, decode, assembling and disassembling.
- Used in the ARM Compiler toolchain.
- A golden reference implementation – no known bugs!
- Is ARM proprietary IP.

What is the test space for ARM?

- Test space has 4 dimensions
  - Instruction encoding
    - 2^32 possible values
  - Instruction set
    - 2 possible values: ARM, Thumb
  - Architecture variant
    - 204 possible values
  - MC Functionality
    - 4 possible values {encode, decode, disassemble, assemble}
- The whole test space has O(7 trillion) points
  - 7,009,386,627,072 points
  - Even at 100,000 tests/s this would take 3.3 years to cover

Can we make this smaller?

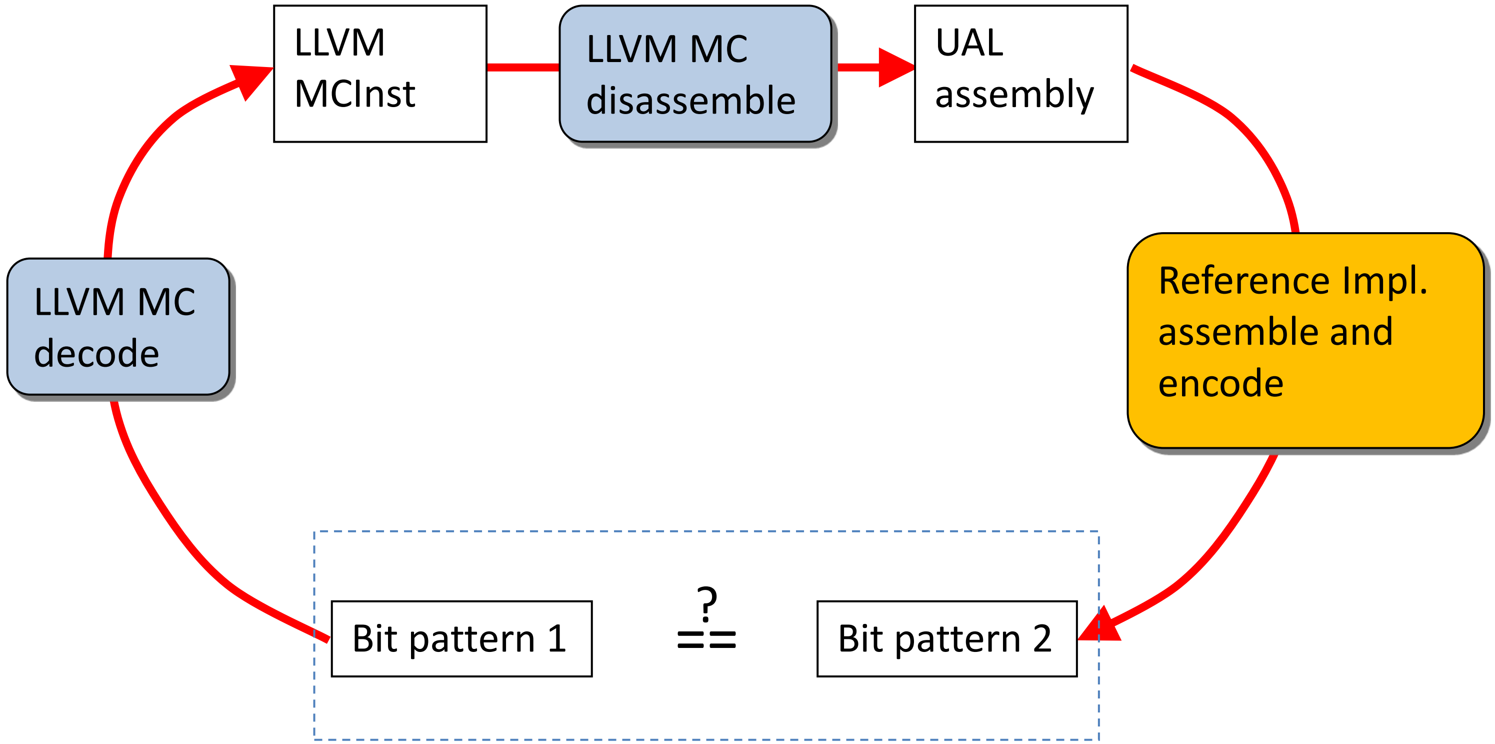
- Some cores do not support certain instruction sets
  - e.g. ARMv6M is Thumb only (Cortex-M0)
- Some architecture and extensions combinations are not permitted.
  - e.g. ARMv7 with VFPv2
- ARMv7 architecture extensions are often orthogonal
  - e.g. VFP/NEON and security extensions
- For a plain Cortex-A8 core there are O(34 billion) points
  - 34,359,738,368 points

Slicing the Test Space

- The Test Suite will run on a slice of the test space.
- A slice is a 4-tuple describing a subset of the possible values of each dimension.
  - For example:
  - 0x0 – 0x0000FFFF x ARMv5TE x Thumb x assemble
  - 0x0 – 0xFFFFFFFF x ARMv7-A + VFPv3 + Adv. SIMDv1 + Half Precision Extension + Security Extensions x ARM x encode_decode
  - 0bXXXX_0000_0001_XXXX_0000_XXXX_1001_XXXX x ARMv7-A x ARM x disassemble

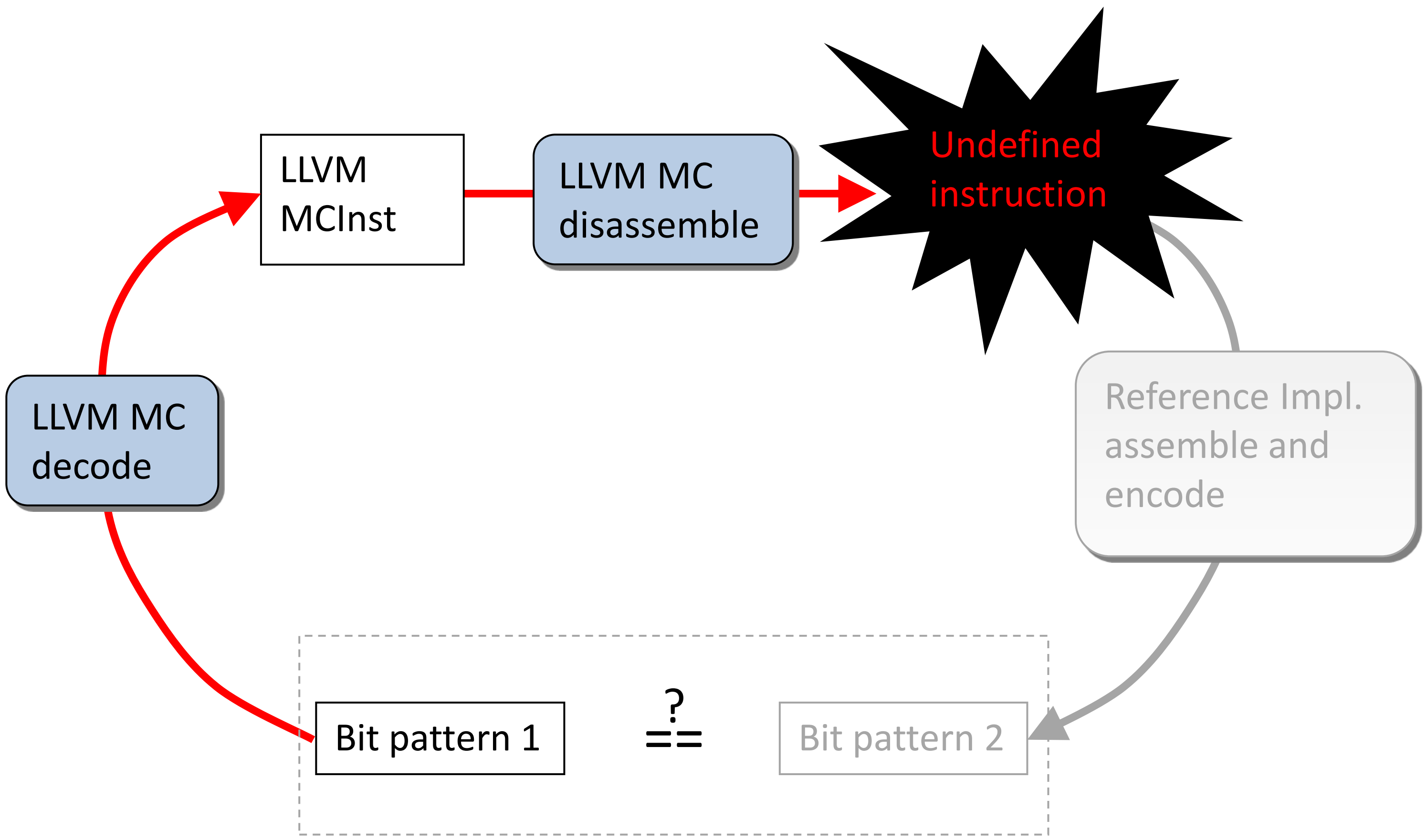
Implementation Details

- How can we ensure that undefined instructions are correctly transformed?



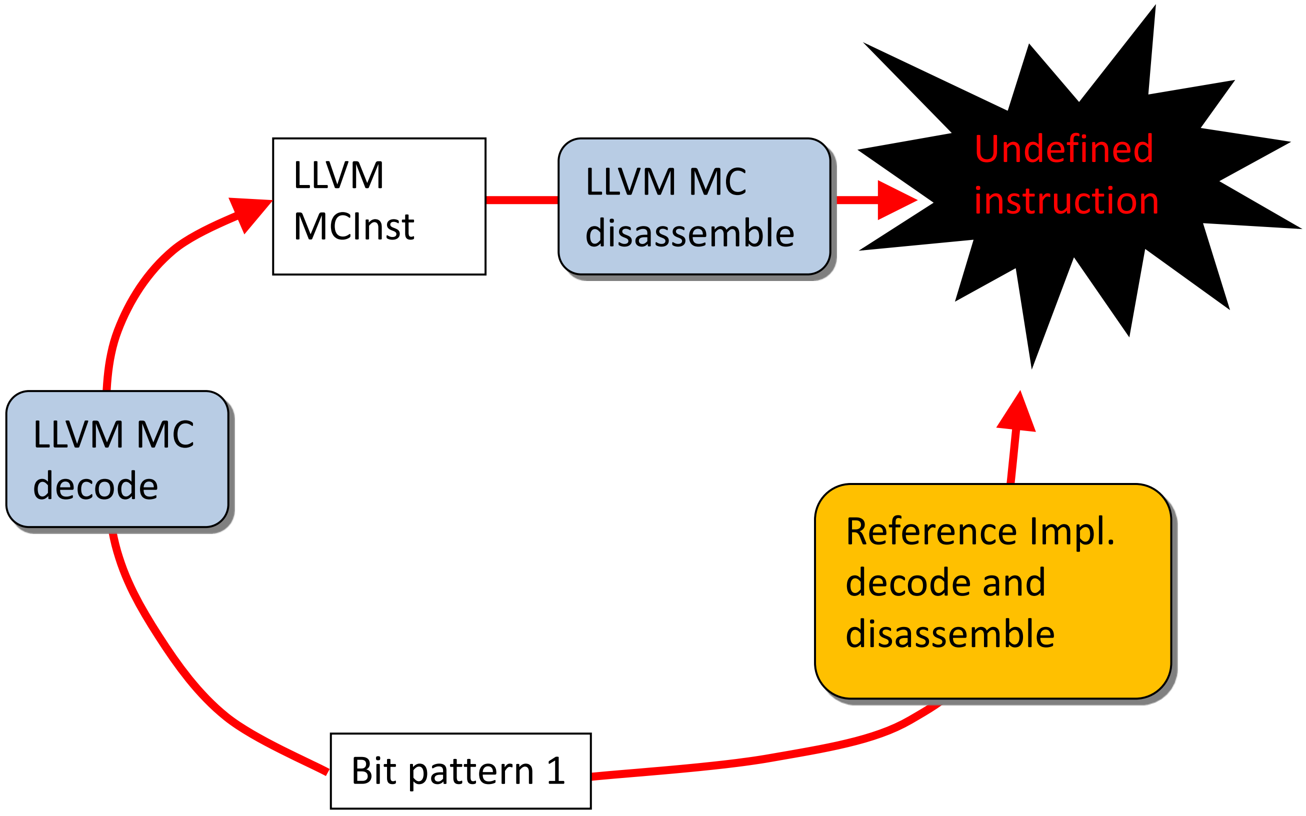
Implementation Details

- How can we ensure that undefined instructions are correctly transformed?



Implementation Details

- How can we ensure that undefined instructions are correctly transformed?



Implementation Details

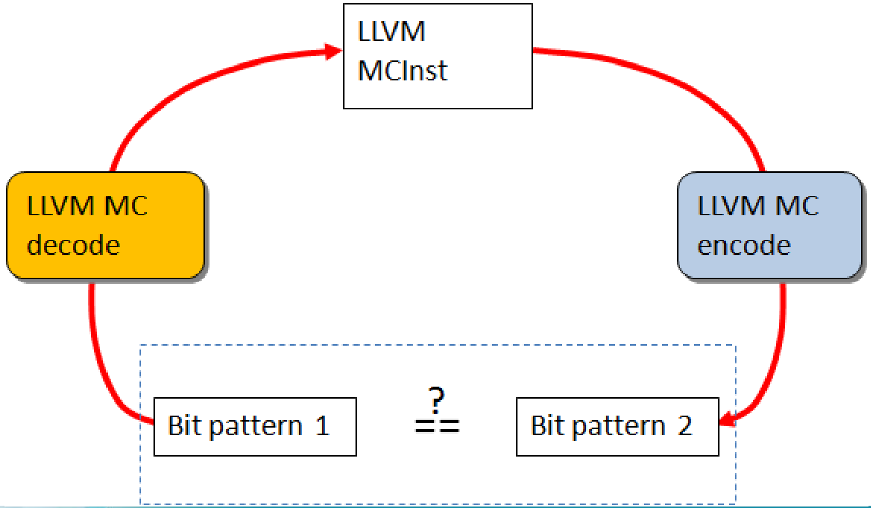
- How can we ensure that undefined instructions are correctly transformed?
- For this you will need at least some decoder implementation as well as an assembler.
  - We solve this problem by comparing Icodec's internal representation instead of bit patterns.
  - We know that MC cannot create an instruction from a bit pattern that should be an undefined instruction.

Example Bug: VCVT

- VCVT (between floating-point and fixed point)
  - *VCVTEQ.F32.S16 s0,s0,#16*
- Symptom is a SIGABRT with a bit pattern.

```
Running slice:
core_v7A+vfpneon_vfpv3_neonv1
feature_ARM
0x0 - 0x3ffffff
encode_decode

*** Killed by signal 6 *** (bitpattern eba0a40)
```



Example Bug: VCVT (2)

- Investigation showed that the Vd operand was not being mapped into the instruction encoding in tablegen, causing the MCInst to have two too few operands, and the encoder to try to read a non-existent operand.

Encoding T1/A1 VFPv3, VFPv4 (sf = 1 UNDEFINED in single-precision only variants)

```
VCVT<c>.<Td>.F64 <Dd>, <Dd>, #<fbits>
VCVT<c>.<Td>.F32 <Sd>, <Sd>, #<fbits>
VCVT<c>.F64.<Td> <Dd>, <Dd>, #<fbits>
VCVT<c>.F32.<Td> <Sd>, <Sd>, #<fbits>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	1	op	1	U	Vd				1	0	1	sf	sx	1	i	0	imm4			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	1	op	1	U	Vd				1	0	1	sf	sx	1	i	0	imm4			

```
to_fixed = (op == '1');  dp_operation = (sf == '1');  unsigned = (U == '1');
size = if sx == '0' then 16 else 32;
frac_bits = size - UInt(imm4:i);
if to_fixed then
    round_zero = TRUE;
else
    round_nearest = TRUE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
if frac_bits < 0 then UNPREDICTABLE;
```

Example Bug: VCVT (3)

- Needed to add a split in the class hierarchy for single- and double-precision versions as they encoded Vd differently

```
// Single Precision register
class AVConv1XInsS_Encode<bits<5> op1, bits<2> op2, bits<4> op3, bits<4> op4,
                          bit op5, dag oops, dag iops, InstrItinClass itin,
                          string opc, string asm, list<dag> pattern>
  : AVConv1XI<op1, op2, op3, op4, op5, oops, iops, itin, opc, asm, pattern> {
  bits<5> dst;
  // if dp_operation then UInt(D:Vd) else UInt(Vd:D);
  let Inst{22} = dst{0};
  let Inst{15-12} = dst{4-1};
}

def VTOSHS : AVConv1XInsS_Encode<0b11101, 0b11, 0b1110, 0b1010, 0,
                       (outs SPR:$dst), (ins SPR:$a, fbits16:$fbits),
                 IIC_fpCVTSI, "vcvt", ".s16.f32\t$dst, $a, $fbits", []> {
  // Some single precision VFP instructions may be executed on both NEON and
  // VFP pipelines on A8.
  let D = VFPNeonA8Domain;
}
```

Example Bug: VCVT (4)

- Created patch and added test cases.
- Re-run slice through MC Hammer to check that it is completely correct

```
slice=[0b111011101x111x1xxxxx101xx1x0xxxx][core_v7a+vfpneon_vfpv3_neonv1]
[feature_ARM][encode_decode]
```

Common errors

- Regression tests with 0-registers.
- Internal inconsistency within MC between uncommonly tested code paths. Probably assemble+encode and decode+disassemble are quite well tested but other combinations like encode/decode are not.
  - Patch for llvm-mc imminent
- MC does not have a good model of unpredictable ARM instructions.
  - Added a third failure mode for these instructions.
  - *e.g. MUL pc, r0, r1* is

How Trustworthy is MC?

- Initial indication is that ~10% of all ARM instructions for a Cortex-A8 slice are encoded incorrectly by MC.
  - 18% of ARM instructions incorrect assembled
- Test suite performance (1 thread)
  - For encode decode MC Hammer can run 7 Million tests/s. So one Cortex-A8 slice takes ~ 1 hour.
  - The assemble/disassemble tests take a few hours
- Progress
  - ~ 2 man months of effort so far
  - 14 patches submitted upstream, 8 accepted.
  - ~ 0.5% decrease in encoding bugs so far

How does this help the community?

- We think our approach is the first structured approach to improving the correctness of code generation for ARM in the MC layer.
- There is an ongoing effort to make the MC Layer more reliable.
- The methodology can easily be applied to other tool chains and other architectures.
  - Requires a reference implementation, normally an assembler.
  - Requires a unified assembly syntax

The End

- Thank you for listening.