敏捷开发软硬件工具介绍

MCU-in-the-Loop 数模混合验证与自动硬件生成

技术白皮书 / Demo 介绍

2024 年 5 月

1 平台概述

MIL（MCU-in-the-Loop）敏捷硬件开发平台解决数模混合 IC 设计中的核心矛盾:

> 纯数字仿真（*EDA Simulation*）无法还原模拟电路的高精度非线性、噪声和动态特性；传统 *HLS*（如 *Vivado HLS*）生成的 *Verilog* 代码臃肿、可读性差、时序难以精细控制。

平台的核心设计思想是**一次编码，软硬复用**：在 MCU 上运行的算法 C 代码，通过 c2v 工具链直接编译为可综合 Verilog RTL；同时 MCU 通过高速桥接板与真实模拟外设（功放、电源网络、传感器 AFE 等）直连验证，确保软件验证的控制逻辑与未来硬件行为 100% 对齐。

其核心价值主张：在流片前，让 MCU 运行真实算法与模拟电路”肉搏”。

- **验证可信度**：使用真实模拟外设在环验证，而非理想化的数字仿真模型和模拟模型
- **开发效率**：C 代码一次编写，同时用于 MCU 验证和硬件综合，消除 RTL 重写成本
- **代码质量**：c2v 编译器生成的 Verilog 无冗余控制流，时序可控，面积可预测
- **自动定点化**：基于实测数据的位宽推导，取代手工试错，避免溢出和精度损失
- **自定义流水线**：c2v 编译器支持自定义流水线，用于优化 Verilog 代码的时序和面积

- **自定义黑盒**：c2v 编译器支持自定义 verilog 黑盒
- **内部采用 64 位 integer 数据类型**：满足 90%fixpoint 数据处理应用场景。

1.1 系统架构

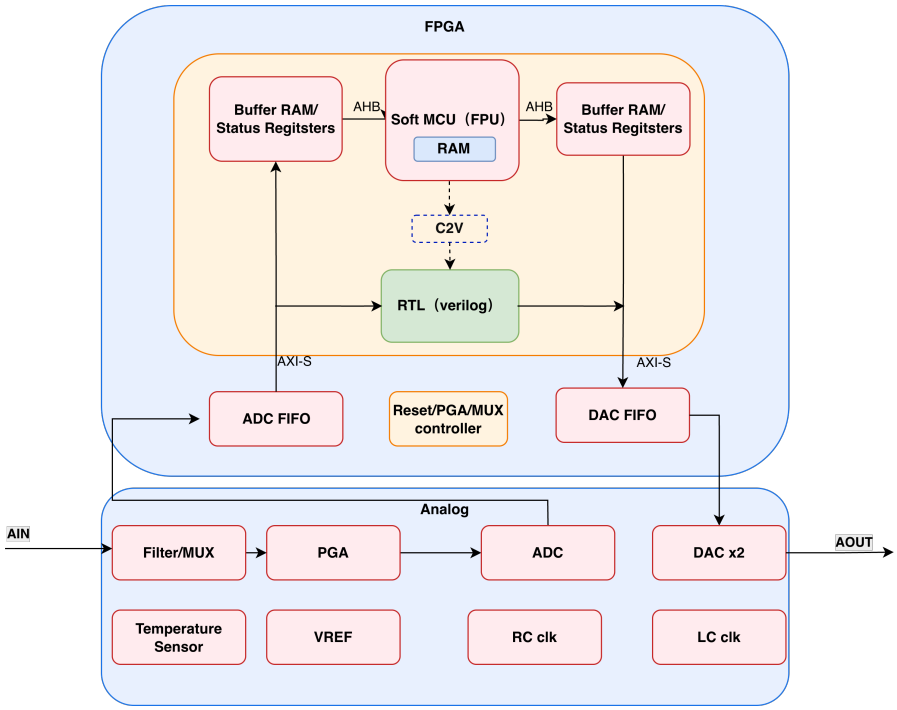

图 1: MIL 验证系统硬件架构

1.2 验证流程

1. 在 MCU 上运行算法 C 代码（如 `fixpoint_pi.c`）和真实的模拟模块进行数据交互
2. 同一套 C 代码输入 c2v 工具链，编译为 Verilog RTL
3. 将生成的 Verilog 部署到 FPGA 原型，对比 MCU 结果与硬件结果

2 c2v 编译器：C → Verilog 端到端转换

c2v 将 C 语言定点计算自动转换为 Verilog 硬件描述。`examples/` 目录提供 13 个 C 示例，覆盖基础运算、滤波器、定点对齐、流水线、结构体指针、CORDIC 与控制算法等典型场景。

2.1 编译管道

```
# 4 抽头 FIR 滤波器（static 变量 → 移位寄存器）
python3 -m c2v examples/fir_filter.c --function=fir_filter \
  -o verilog/fir_filter.v

# 一阶 IIR 低通（IQmath 定点 + 饱和）
python3 -m c2v examples/iir_lpf.c --function=iir_lpf \
  --clock-port clk -o verilog/iir_lpf.v

# 流水线 IIR（__pipeline_reg → seq.compreg，需 --clock-port）
python3 -m c2v examples/iir_lpf_pipeline.c \
  --function=iir_lpf_pipeline --clock-port clk \
  -o verilog/iir_lpf_pipeline.v

# 复数乘法流水线版
python3 -m c2v examples/struct_ptr_demo_pipeline.c \
  --function=complex_mul_pipelined --clock-port clk \
  -o verilog/struct_ptr_demo_pipeline.v

# 调试：打印中间 MLIR
python3 -m c2v examples/iir_lpf.c --function=iir_lpf --debug
```

Listing 1: examples/ 目录典型编译命令

2.2 示例：FIR 滤波器端到端

以 `fir_filter.c` 为例，展示 `static` 变量如何映射为硬件寄存器：

```
// y[n] = x[n]*c0 + x[n-1]*c1 + x[n-2]*c2 + x[n-3]*c3
int fir_filter(int x, int c0, int c1, int c2, int c3) {
    static int s0, s1, s2, s3;
    // 移位延迟线：s3 <- s2 <- s1 <- s0 <- x
    s3 = s2;
    s2 = s1;
    s1 = s0;
    s0 = x;
    return s0*c0 + s1*c1 + s2*c2 + s3*c3;
}
```

Listing 2: fir_filter.c —4 抽头顺序 FIR 滤波器

c2v 将 `static int s0..s3` 降级为 CIRCT `seq.compreg`，最终导出带 `clk/en/vld` 的 Verilog：

```
module fir_filter(
  input         clk,
```

```
  input         en,
  input  [31:0] x,
  input [31:0] c0,
  input [31:0] c1,
  input [31:0] c2,
  input [31:0] c3,
  output [31:0] out0,
  output        vld,
);

  reg [31:0] s3_reg;
  reg [31:0] s2_reg;
  reg [31:0] s1_reg;
  reg [31:0] s0_reg;
  assign vld = en;
  always @(posedge clk) begin
    if (en) begin
      s0_reg <= x;
      s1_reg <= s0_reg;
      s2_reg <= s1_reg;
      s3_reg <= s2_reg;
    end
  end // always @(posedge)
  assign out0 = x * c0 + s1_reg * c1 + s2_reg * c2 + s3_reg * c3;
endmodule
```

Listing 3: FIR 滤波器生成的 Verilog（示意）

3 定点类型系统（IQmathLib.h）

c2v 通过 `include/IQmathLib.h` 识别定点类型，将 Q 格式语义编码进类型名本身，而非仅靠注释约定。

核心原则：定点的位宽和小数位置必须通过 `_qWpF` 类型显式表达。用裸 `int` 承载定点值无法约束数值范围，任何越界赋值都会静默通过，导致硬件行为与预期不符。

3.1 `_qWpF` 命名规则

- **W** —总位宽（有符号）
- **F（p 后缀）**—小数位宽
- 数值 = 原始整数 / 2^F

- `_q12p5` 表示 12-bit 总宽、5-bit 小数，范围约 $[-128, 127]$

常用类型对照：

类型	总位宽	小数位	典型用途
`_q12p4`	12	4	滤波系数 α
`_q12p5`	12	5	信号 $x[n], y[n]$
`_q24p10`	24	10	乘法中间结果
`SAT_SIGNED(x, N)`	—	—	饱和到 N 位有符号
`_Q_ADD_SAT(...)`	—	—	对齐小数位后相加并饱和

3.2 一阶 IIR 低通滤波器 Demo

以一阶 IIR 低通滤波器（`iir_lpf.c`）为例，展示 IQmath 定点的完整用法：

$$y[n] = y[n-1] + \alpha \cdot (x[n] - y[n-1]), \quad 0 < \alpha < 1 \tag{1}$$

```c
#include "IQmathLib.h"

_q12p5 iir_lpf(_q12p5 x, _q12p4 alpha)
{
    static _q12p5 y;
    int64_t err = x - y;
    int64_t inc = alpha * err;
    y = SAT_SIGNED(y + inc, 12);
    return y;
}
```

Listing 4: iir_lpf.c —IQmath 定点一阶 IIR 低通

关键设计要点：

- **`_q12p5` / `_q12p4`** —信号与系数使用不同 Q 格式，类型名携带位宽信息
- **`static _q12p5 y`** —映射为时钟沿更新的硬件寄存器
- **`alpha * err`** —IQmath 乘法自动扩展位宽（$W_a + W_b$）
- **`SAT_SIGNED(..., 12)`** —状态更新饱和截断，防止溢出导致系统震荡

```
python3 -m c2v examples/iir_lpf.c --function=iir_lpf \
  --clock-port clk -o verilog/iir_lpf.v
```

Listing 5: 编译命令

4 examples/ 示例程序

examples/ 目录下全部 C 示例按功能分类如下：

分类	文件	说明
基础运算	bitwise_ops.c	与/或/异或、逻辑左移、算术右移
	comparison.c	等于/小于/大于/不等比较，三元选择
数字滤波	fir_filter.c	4 抽头 FIR，static 延迟线
	iir_lpf.c	一阶 IIR 低通，组合逻辑
	iir_lpf_pipeline.c	三级 __pipeline_reg 流水线版
定点算术	q_add_with_align.c	不同小数位 _Q_ADD 对齐相加
	q_add_sat.c	_Q_ADD_SAT 对齐相加并饱和
	mul_pipeline.c	两阶段乘法流水线通用演示
结构体	struct_ptr_demo.c	复数乘法组合版，结构体指针打包端口
	struct_ptr_demo_pipeline.c	6 级流水线复数乘法
CORDIC	cordic_demo.c	流水线 CORDIC + RTL blackbox
	cordic_iter.c	单步组合 CORDIC（无分支，供 FSM 复用）
控制算法	fixpoint_pi.c	Q16.16 定点 PI 控制器

4.1 基础运算

bitwise_ops.c 和 comparison.c 展示 c2v 对位运算、比较和三元运算符的支持，生成纯组合逻辑模块，适合作为语法覆盖的最小示例。

4.2 数字滤波

FIR 滤波器 见第 2 节。static 变量实现移位寄存器延迟线，是理解 c2v 时序映射的入门示例。

IIR 低通（组合版） 见第 4 节。iir_lpf.c 使用 IQmath 类型和饱和运算。

IIR 流水线版 iir_lpf_pipeline.c 在乘积、累加、饱和三处插入 __pipeline_reg，提高时钟频率：

```
#include "IQmathLib.h"
#include "pipeline_annot.h"

_q18p5 iir_lpf_pipeline(_q12p5 x, _q12p4 alpha) {
    static _q18p5 y;
```

```
    int64_t err = x - y;
    int64_t inc = alpha * err;
    int64_t p1 = __pipeline_reg(inc);
    int64_t p2 = __pipeline_reg(y + p1);
    y = __pipeline_reg(SAT_SIGNED(p2, 18));
    return y;
}
```

Listing 6: iir_lpf_pipeline.c —三级流水线 IIR（节选）

编译时必须指定 `--clock-port clk`，工具将每个 `__pipeline_reg` 替换为 `seq.compreg` 触发器。

4.3 定点对齐与饱和

不同 Q 格式的加法需要先对齐小数位。`q_add_with_align.c` 使用 `_Q_ADD` 宏，`q_add_sat.c` 使用 `_Q_ADD_SAT` 在对齐基础上增加饱和保护：

```
#include "IQmathLib.h"

_q41p9 q_add_sat(_q21p7 x, _q41p8 y)
{
    _q41p9 k = (_q41p9)_Q_ADD_SAT(x, 7, y, 8, 9, 41);
    return k;
}
```

Listing 7: q_add_sat.c —对齐饱和加法

4.4 流水线寄存器

`include/pipeline_annot.h` 声明 `__pipeline_reg`：C 仿真时为恒等函数，cgeist 编译时保留为外部调用，lowering 替换为 CIRCT `seq.compreg`。

`mul_pipeline.c` 是最小流水线演示——乘 → 加增益 → 饱和，两拍完成：

```
#include "IQmathLib.h"
#include "pipeline_annot.h"

_q12p5 mul_pipeline(_q12p5 a, _q12p5 b, _q12p4 gain)
{
    _q24p10 prod   = __pipeline_reg(a * b);
    _q13p5  scaled = __pipeline_reg(prod + gain);
    return SAT_SIGNED(scaled, 12);
}
```

Listing 8: mul_pipeline.c —两阶段乘法流水线

4.5 结构体与指针

`struct_ptr_demo.c` 实现复数乘法，展示 c2v 对结构体字段和指针参数的处理——`Complex*` 参数打包为固定位宽端口，字段通过切片访问：

```
#include "IQmathLib.h"

typedef struct { _q12p5 re; _q12p5 im; } Complex;

Complex* complex_mul(Complex* a, Complex b) {
    Complex c;
    _q24p10 re32 = a->re * b.re - a->im * b.im;
    _q24p10 im32 = a->re * b.im + a->im * b.re;
    c.re = SAT_SIGNED(re32, 22);
    c.im = SAT_SIGNED(im32, 22);
    return &c;
}
```

Listing 9: struct_ptr_demo.c —复数乘法（组合版）

`struct_ptr_demo_pipeline.c` 在每次乘法、加减后插入 `__pipeline_reg`，生成 6 级流水线，信号名与 C 源码一致（`re_prod1_reg`、`re32_clamp_hi` 等），便于调试与综合后时序分析。

4.6 CORDIC

CORDIC 示例提供两种实现策略：

流水线版(`cordic_demo.c`) C 侧调用 `cordic_sin/cos` 库函数,映射到 `rtl/cordic_rot.v` 手写流水线核；配合 16 级延迟线和 `__pipeline_reg` 实现 timing-aligned 旋转求和。编译：

```
bash examples/run_cordic_demo.sh
```

串行版（`cordic_iter.c`） 单步纯组合 CORDIC 迭代，避免 `if/switch/?:`（c2v 尚不支持 `scf.if` 降级），用手写 FSM 顶层（`rtl/cordic_serial.v`）逐拍复用：

```
typedef struct { int x; int y; int z; } CordicVec;

static int mux_addsub(int a, int b, int z_neg) {
    return a + (z_neg & b) + ((~z_neg) & (-b));
}

CordicVec cordic_iter(CordicVec v, int i, int atan_step) {
    CordicVec o;
```

```
    int z_neg = v.z >> 31;
    o.x = mux_addsub(v.x, v.y >> i, z_neg);
    o.y = mux_addsub(v.y, v.x >> i, ~z_neg);
    o.z = mux_addsub(v.z, atan_step, z_neg);
    return o;
}
```

Listing 10: cordic_iter.c —无分支 CORDIC 单步（节选）

	流水线（cordic_demo）	串行（cordic_iter）
逻辑面积	大（N 级并行）	小（1 级复用）
延迟	1–2 拍	N 拍
吞吐 II	1	$\approx N$
实现方式	手写 Verilog blackbox	c2v 步进 + 手写 FSM

4.7 定点 PI 控制器

`fixpoint_pi.c` 展示 Q16.16 格式的完整控制算法，含定点乘法截断和积分累加：

```
#define SCALE (1 << 16)

static int fx_mul(int a, int b) {
    long long p = (long long)a * (long long)b;
    return (int)(p >> 16);
}

int pi_controller(int sp, int fb, int kp, int ki) {
    static int integral;
    int err = sp - fb;
    integral = integral + err;
    int prop     = fx_mul(kp, err);
    int int_term = fx_mul(ki, integral);
    return prop + int_term;
}
```

Listing 11: fixpoint_pi.c —Q16.16 PI 控制器（节选）

c2v 自动将 `static int integral` 映射为时钟寄存器，`fx_mul` 内联为组合乘法器并截断低 16 位，保持 Q16.16 格式对齐。