2025 中国研究生创“芯”大赛·EDA 精英挑战赛

一、 赛题名称

SystemVerilog 仿真器的性能优化

二、 命题单位

芯华章科技股份有限公司

三、 赛题背景

数字前端验证仿真(Simulator)是集成电路设计和验证流程中的核心环节，其重要性贯穿从设计构思到芯片量产的全周期。集成电路数字仿真是决定芯片成败的核心竞争力，它使用计算机结合测试激励来模拟芯片在真实环境下的运行状况，帮助工程师通过各种调试手段来判断运行结果是否符合预期，通过“虚拟试错”将设计风险控制在成本最低的前端验证阶段，是现代 IC 设计“一次成功”的关键保障。数字仿真器一般有事件驱动型(Event-Driven)和周期精确型(Cycle-Based)两种类型。Event-Driven 型仿真器通过事件队列模拟驱动电路状态更新，每个信号变化触发相关逻辑的重新计算，事件驱动型仿真器需要确保时序的精确性。

集成电路验证领域的数字仿真器读取 SystemVerilog 等 HDL 语言描述，通过编译期处理生成可执行文件，在运行时对电路行为进行仿真。随着集成电路规模呈指数级增长，从早期的万门级芯片发展到如今数十亿门的 SoC，超长的仿真时间和巨大的仿真内存需求成为验证效率提升的制约瓶颈。

编译期优化是将硬件描述语言（HDL）转换为高效仿真模型的关键环节，其核心目标是通过静态分析与代码转换，消除冗余计算并构建面向仿真场景的专用执行引擎，从而达到提升仿真性能和减少仿真内存占用的目标。

对于通用编译技术，编译期优化按优化目标和手段一般可分为计算优化（如常量折叠、代数化简等），控制流优化（如条件合并、循环优化、死代码消除），访存优化（如循环分块等）等类别。部分优化策略在数字仿真器的编译期依然可能会有优秀的优化效果；部分优化策略可能需要基于不同语言特性做一些适配之后才能展现出效果。特别地，基于数字仿真器的事件队列调度特性，在编译期对调度队列或事件进行优化，是数字仿真器特有的编译期优化手段之一。当然，编译期优化可能会面临诸如编译时间增长和运行时间减少的权衡与选择、优化效果不确定性以及代码调试功能的破坏性等问题。

芯华章科技股份有限公司的数字仿真器产品“穹鼎”(GalaxSim)是一款基于事件驱动型的数字电路仿真器。穹鼎 GalaxSim 仿真器是芯华章汇聚行业专家经验，并在国内多个知名客户打磨后，推出的一款独立自主、全新构架的高性能仿真器。创新性地使用新的软件框架，提供多平台支持，并且已在多个基于 ARM 平台的国产构架上测试通过。GalaxSim 具有多种使用模式，可兼容各类验证工具进行多种联合仿真。GalaxSim 提供统一的数据接口，全面支持 IEEE1800 SystemVerilog 语法，以及 IEEE1364 Verilog 语法，和 IEEE1800.2 UVM 方法学，穹鼎 GalaxSim 适合各个层次的验证工作，从 IP 到 SoC 再到 Chiplet 验证都有很好的适用场景。

本赛题中，我们要求参赛者使用芯华章数字仿真器 GalaxSim 进行仿真优化的探索。我们将开放 GalaxSim 部分底层 DB 的数据接口，使用 GalaxSim 提供的编译期优化模块动态自动加载机制，设计并实现若干种编译期优化策略，并能正确运行，体现出运行期性能提升，并确保编译期代价增长在可接受范围内。参赛者可以利用 GalaxSim 提供的仿真器框架和运行时诊断手段来查找潜在的优化点。基于本赛题，参赛者可以直接深度参与优秀国产 EDA 数字仿真器的研发进程。

我们开放的底层数据模块是C++接口，参赛者应具备一定的C++开发能力。具备一定的编译基础可能会帮助参赛者更好地上手，具备一定的集成电路基础能够帮助参赛者更好地理解仿真过程，但这些不是必备条件。

四、 赛题解析

本节通过一些常用的编译期优化技术实例，来说明编译期优化如何对运行期性能产生影响，更多的常见优化技术参赛者可自行查阅资料。

1. 通用编译优化技

a. 循环不变量外提

```
// 示例 1
// 优化前：
int a;
// ...
for (int i=0; i<100; i++) {
      integer x = a * 2;
      y[i] = x + i;
}

// 优化后：
// a*2 在循环中不变，移到循环外，避免循环时重复计算
int a;
// ...
integer x = a * 2;
for (int i=0; i<100; i++) {
      y[i] = x + i;
}
```

b. 条件表达式优化

```
// 示例 2
// 优化前：
// 在数字电路设计 HDL 代码中，优化前的代码风格较常见
if (a == b) $display(123);
if (a == b) $display(456);

// 优化后：
```

```
// 通过合并相同条件的条件语句，可减少条件判断次数
// 比如以控制信号/使能信号特定值作为条件
if (a == b) begin
    $display(123);
    $display(456);
end
```

2. 通用编译优化技术应用到数字仿真器

```
// 示例 3
// 优化前：
assign b = c;
assign a = b;

// 优化后：
// 在数字电路中，在该组赋值中，b 相当于一组线，可直接将 net a 与 net c 相连，
// 减少一次值传播调度
assign a = c;
```

```
// 示例 4
// 优化前：
assign a = b + c + d + e + f + g

// 优化后:
// 该优化可减少计算量
// 比如当只有 f/g 值更新时，可不必重新计算"b + c + d + e"部分
// 在数字电路中，该优化会增加值传播调度代价，运用该优化的条件需仔细斟酌
assign a = tmp0 + f + g
assign tmp0 = tmp1 + d + e
assign tmp1 = b + c
```

```
// 示例 5
// 优化前：
always @(*)
a = b;

// 优化后：
// always 块调度成本高于 continue assign 值传播成本
assign a = b;
```

3. 数字仿真器特有编译优化技术

```
// 示例 6
// 优化前：
reg [255:0] w_data;
reg [256*2048 - 1 : 0] data;
assign data[0+:2048] = {w_data, 1792'b0};
assign data[2048+:2048] = {8'b0, w_data, 1784'b0};
assign data[4096+:2048] = {16'b0, w_data, 1776'b0};
//...
assign data[24576+:2048] = {96'b0, w_data, 1696'b0};

// 优化后：
// 通过优化，大幅减少了该示例中事件队列调度列表的大小，减至 1
function logic [24576+2048-1:0] F(input logic [255:0] _unused);
    F[0+:2048] = {w_data, 1792'b0};
    F[2048+:2048] = {8'b0, w_data, 1784'b0};
    //...
    F[24576+:2048] = {96'b0, w_data, 1696'b0};
endfunction
assign data[24576+2048-1 : 0] = F(w_data)
```

五、 赛题描述

为确保能如实体现参赛者所设计算法的预期优化效果，我们将默认关闭 GalaxSim 已有的优化内容。我们将关闭了已有优化内容的 GalaxSim 版本定义为 *GalaxSim-Base*。所有参赛者的优化算法都将基于 GalaxSim-Base 运行和测试。

1. 测试用例

在本赛题中，我们从两个维度对测试用例进行定义，具体定义及提供的测试用例数量如下表，其中的标题后数字为该类用例的评分权重：

	公开用例(0.3)	隐藏用例(0.7)
简单用例(0.2)	4 个	暂不公开个数
综合用例(0.8)	2 个	暂不公开个数

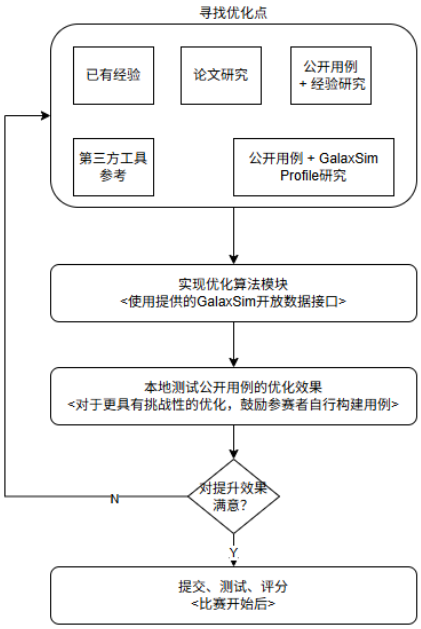
- 公开用例：参赛者可完全访问该用例，并可用于研究优化点及本地测试评分。
- 隐藏用例：比赛结束前参赛者完全不可访问，仅用于闭卷测试评分。
- 简单用例：该类用例通常只有 1-2 类瓶颈问题，测试例本身较简单，比如一个占数万行的表达式。
- 综合用例：该类用例一般代表一种真实的 IC 设计，诸如 CPU，DSP，AXI 总线模块等。该类用例可能包含多个仿真瓶颈问题，测试例本身可能也很复杂，比如 or1200, c910, xiangshan 等。
- 扩展用例：对于更具挑战的优化，参赛者可自行构建用例，并进行针对性的优化，同时芯华章团队也会对这些扩展用例做评估，一旦被认为有价值，这些用例还将被收录进公开用例的集合中（相应加入简单公开用例集和综合公开用例集），过一定时间之后对所有参赛者公开，这些用例将享受和公开用例同样的待遇，扩展用例若被采纳，参赛团队将获得一定的现金奖励。所有参赛团队会被限定可被采纳的扩展用例的数目，扩展用例的提交时间也将会有截止日期，具体规定详见之后的赛事细则。

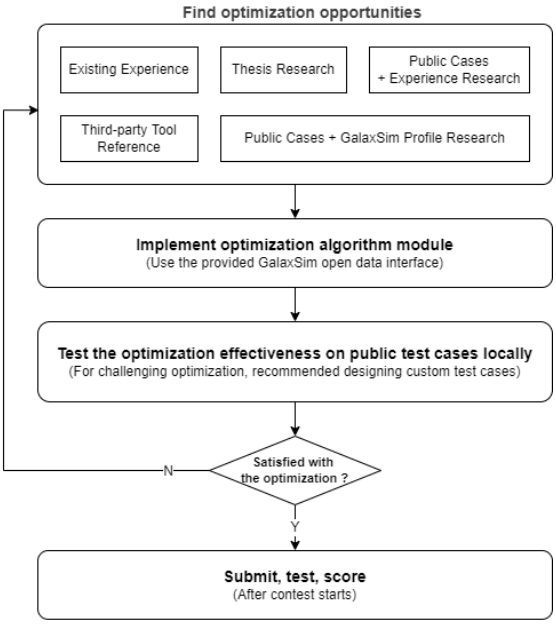
本赛题赛前将向参赛队伍提供 6 个设计示例作为参考，其中包含 4 个简单用例和 2 个综合用例。同时，我们将向参赛者提供隐藏用例用于闭卷评分，隐藏用例将在赛后对所有参赛者公布。

对于所有测试用例，我们都会附带一组基于科学采样方法获取的 GalaxSim-Base 测试数据，这组数据将作为参赛者算法优化效果的评分基准。对于每个公开用例，参赛者将获得对应的基础测试数据作为参考。

2. 优化过程

我们将提供 GalaxSim 的 license 资源，参赛者可以在我们提供的服务器或自己的机器上进行算法开发和调试优化。

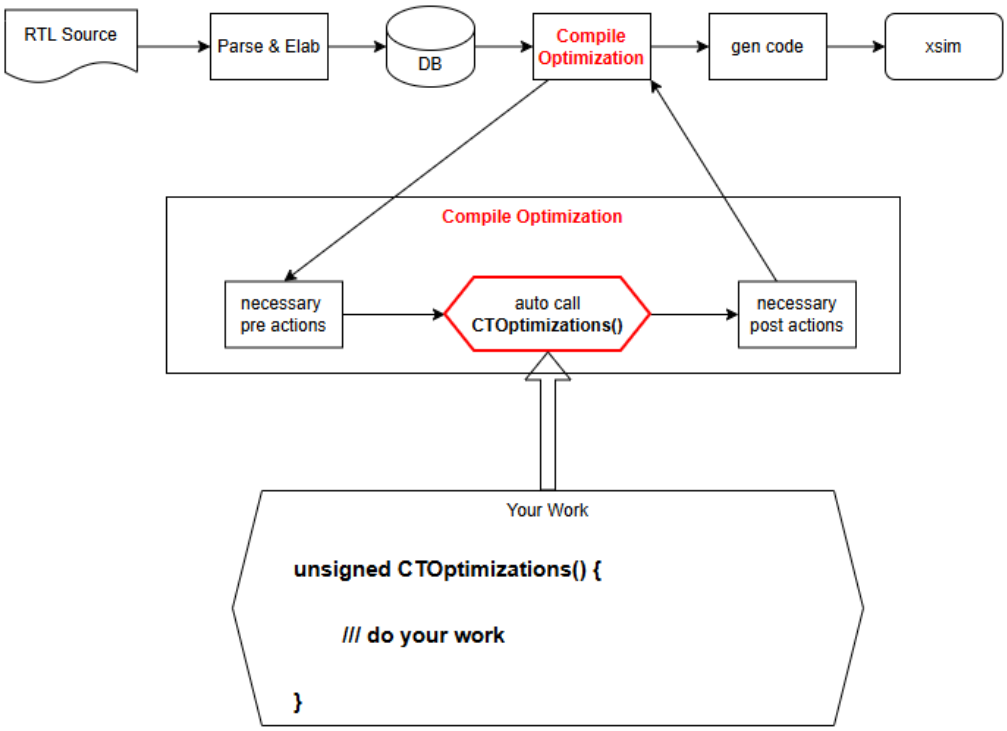




GalaxSim-Base 提供诊断选项，可供参赛者在编译期和运行期分别诊断算法优化效果。比赛开始后，专门的评分测试工具将运行 GalaxSim-Base + 参赛者优化库，并自动将结果与基准数据比较，得到测试例的分值。具体分值计算方式可参考“**评分标准**”。

除了我们以用例形式公开的优化策略外，我们还鼓励参赛者通过各种途径获取知识，设计编译期优化点，以获得更好的优化效果，并提高测试例的评分。

下图示例了参赛者的优化策略将如何集成到 GalaxSim 中。



3. 提交说明

我们约定参赛者优化算法的主入口为"*void CTOptimizations()*",参赛者的所有优化工作，包括但不限于多个优化算法的实现以及多个优化算法之间的循环迭代等，都应在该函数中完成。参赛者需以动态库形式提交自己的优化算法，动态库命名为optimizations.so。

参赛者的作品应以zip压缩文件的格式进行提交,文件名统一为"submission.zip"。submission.zip 中应包含参赛者优化算法的动态库文件(optimizations.so)及所依赖的第三方头文件和库文件。其中，optimizations.so 文件必须位于提交压缩包的根目录，其它依赖文件的目录结构由参赛者自行组织。

submission.zip 文件提交至评分服务器后，将被自动解压，并执行 GalaxSim 主程序。GalaxSim 将自动加载参赛者提交的优化算法动态库。在编译期和运行期运行成功后，评分工具将自动给出评分。

参赛者需要确保优化前后的 SystemVerilog 设计语义等价，并且保证提供的程序是可靠的。如果参赛者的优化改变了语义等价性导致仿真失败，或者参赛者的程序存在错误(如空指针未判定、指针未释放导致的内存泄露等问题)导致编译失败等问题，那么参赛者在该用例上的评分将为 0。同时，编译期过多的额外代价会影响最终评分。

六、 评分标准

我们限定 GalaxSim 为单线程运行。在评分中，所有涉及运行时间的评价指标均定义为端到端的**墙上时间**(Wall Time)。所有评分测试均基于科学采样法进行，即：在无其它运行负载的机器上，对每个测试例执行 10 次采样，去掉运行时间的最大值和最小值后，对各项指标取算术平均值，作为本次测试的采样结果。

我们定义单个指标的提升比 Ratio 如下：

Ratio = GalaxSim Base Data / Your Data

其中一份基准数据为 GalaxSim Base 的表现数据，GalaxSim Base 默认关闭了编译期优化内容。GalaxSim Base 的数据预先提供，Your Data 则在每次提交后基于科学采样方式获取。

基于此，对于每个测试例，我们设计 4 个提升比指标：

- ✧ CMR (Compile Memory Improve Ratio)

  CMR = GalaxSim Base CM / Your CM

- ✧ CTR (Compile Time Improve Ratio)

  CTR = GalaxSim Base CT / Your CT

✧ RMR (Run Memory Improve Ratio)

RMR = GalaxSim Base RM / Your RM

✧ RTR (Run Time Improve Ratio)

RTR = GalaxSim Base RT / Your RT

对于每个测试例的效果评分(Performance per case)，定义为：

Perf_per_case = 0.1 * CMR + 0.2 * CTR + 0.1 * RMR + 0.6 * RTR

例如，假设某个参赛队伍，在某个测试例的优化效果上，Memory 消耗不变，编译期性能降低为原来的 80%（时间为基准的 1.25 倍），运行期性能提升为基准的 2 倍（运行时间为基准的 0.5 倍），则该队伍在该测试例的性能得分为：

Perf_per_case = 0.1 * 1 + 0.2 * 0.8 + 0.1 * 1 + 0.6 * 2 =1.56

通过正确性检查后，根据性能得分进行排名评分(RankScore_per_case)：假设参赛人数为 N，名次为 M，得分根据以下公式计算：

RankScore_per_case = 10 * (N - M + 1) / N

第一名得 10 分，第二名得(N-1)*10/N.....

例如，假设有 26 个参赛队伍，某个队伍在某个用例上的性能得分排名第 11 位，则该队伍在该用例上的排名评分为：

RankScore_per_case = 10 * (26 - 11 + 1) / 26 = 6.15

将每个测试用例的分数加权求和得到最后总评分。

权重设计如下:

用例类型	排名评分求和权重
公开用例	0.3
隐藏用例	0.7
简单用例	0.2
综合用例	0.8

参赛队伍最终总评分计算公式：

$$\begin{aligned}\text{FinalScore} = 0.3*0.2*&\sum \text{RankScore(public basic cases)}\\ +0.3*0.8*&\sum \text{RankScore(public comprehensive cases)}\\ +0.7*0.2*&\sum \text{RankScore(hidden basic cases)}\\ +0.7*0.8*&\sum \text{RankScore(hidden comprehensive cases)}\end{aligned}$$

注意事项：

- CM/CT/RM/RT 的数据统计使用 GalaxSim 本身提供的 profile 机制
- 如果某个测试用例参赛队伍没有任何提交记录，则提升比指标均为 0
- 如果某个测试用例在编译期通过但在运行期失败，则提升比指标均为 0
- 所有得分运算过程中涉及的数字皆保留 2 位有效数（舍弃 3 位之后的有效数字，对第 3 位有效数字四舍五入）
- 如果最终有多名参赛队伍评分相同，则按以下顺序进行排序：首先，按达到该评分的时间先后顺序排序；如果时间相同，则按综合用例评分高低排序;如果综合用例评分也相同，则按综合用例的 RTR(RunTime Ratio)评分高低排序。

七、 参考资料

[1] IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language, 2017

[2] R. Allen and K. Kennedy. Optimizing Compilers for Modern Architectures. Academic Press, 2002. ch12.3

八、 附录及补充资料

1. 事件驱动型仿真模型 Event-Driven Simulation Model

对 SystemVerilog 描述的设计(dut)和测试例(testbench)进行仿真是基于离散事件执行模型推进，一般可分为事件更新(update event)和事件评估(evaluation event)，两者都

被定义为事件(event)。事件更新一般是指 net/variable 值的变化。事件评估是对事件更新的响应，update event 会触发 evaluation event。

在事件驱动仿真器中，另一个重要概念是时间(time, or simulation time)。从真实时间流逝的角度，time 被分解为时间序列，类似于数字电路中 clock 信号的方波序列。时间序列由一组时间槽(time slot)组成，一个单一的时间槽通常可简单地认为是数字电路中的一个时钟周期(cycle)。

为了更明确地定义事件之间的关系，仿真模型一般将每个单一的时间槽(time slot)分为多个不同类型的事件队列(region)，将不同类型的 RTL 处理调度到对应的 region 中，供仿真模型调用执行。比如非阻塞赋值(nonblocking assignment)会被调度到 NBA events region 进执行。在 SystemVerilog 2017 LRM (Language Reference Manual)中，定义了 17 种不同的事件队列。对参赛者而言，可不必非常清晰地掌握这些不同调度队列的差异和区别；当然如果掌握了该部分内容，可对事件驱动型仿真器进行更深层次的优化。

下面附上 SystemVerilog LRM 中仿真算法伪代码，供有兴趣的参赛者进一步理解事件驱动型仿真模型，参考者也可直接参考 SystemVerilog LRM 手册第 4 章。

```
execute_simulation {
    T = 0;
    initialize the values of all nets and variables;
    schedule all initialization events into time zero slot;
    while (some time slot is nonempty) {
        move to the first nonempty time slot and set T;
        execute_time_slot (T);
    }
}

execute_time_slot {
    execute_region (Preponed);
    execute_region (Pre-Active);
```

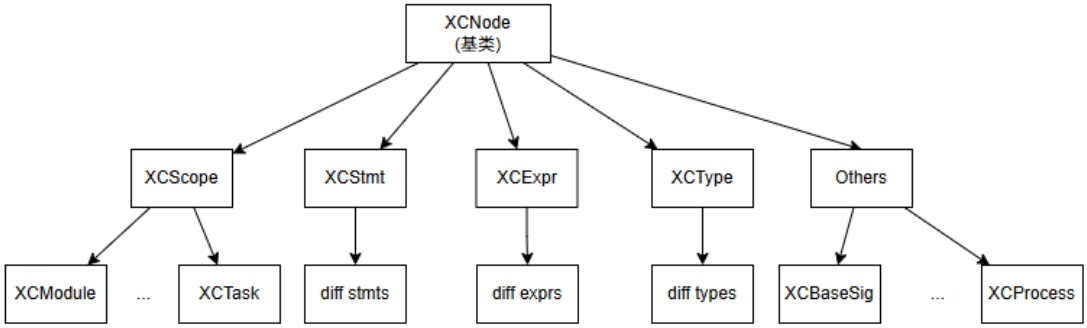
```
        while (any region in [Active ... Pre-Postponed] is nonempty) {
            while (any region in [Active ... Post-Observed] is nonempty) {
                execute_region (Active);
                R = first nonempty region in [Active ... Post-Observed];
                if (R is nonempty)
                    move events in R to the Active region;
            }
            while (any region in [Reactive ... Post-Re-NBA] is nonempty) {
                execute_region (Reactive);
                R = first nonempty region in [Reactive ... Post-Re-NBA];
                if (R is nonempty)
                    move events in R to the Reactive region;
            }
            if (all regions in [Active ... Post-Re-NBA] are empty)
                execute_region (Pre-Postponed);
        }
        execute_region (Postponed);
    }
```

```
    execute_region {
        while (region is nonempty) {
            E = any event from region;
            remove E from the region;
            if (E is an update event) {
                update the modified object;
                schedule evaluation event for any process sensitive to the object;
            } else { /* E is an evaluation event */
                evaluate the process associated with the event and possibly
                schedule further events for execution;
            }
        }
    }
```

2. 开放 DB 的设计及与 RTL 映射关系

a. 关键语法要素及 DB 设计

根据 SystemVerilog LRM 语法参考手册，基本的语法要素包括 scope, expr, operator, statement, task/function, data type, process 等，基于这些基本语法要素，我们开放的 GalaxSim 底层 DB 基本设计示意图如下。



一些常见的 HDL 语法单元及表示：

✓ XCNode

所有语法单元类的基类，提供 GetLoc(), GetAttribute()等接口。

✓ XCScope

对应 SystemVerilog scope 的概念，所有内部可包含“definition”的语法元素都可称之为一个 XCScope，比如 module, block, class, task, function 等。

XCScope 提供了对应接口对内部包含的语法元素进行增删改查操作。可参见 SystemVerilog LRM 手册第 23.9 节。

✓ XCStmt

语句，可嵌套。

✓ XCExpr

表达式，可嵌套。

✓ XCType

类型系统基类。

✓ Others

其它语法单元表示。

✓ XCDesign

表示用户的整个设计根节点。

另外，我们提供了 Visitor 模式的实现，可支持参赛者对设计进行遍历操作。

b. RTL 与 DB 设计映射关系举例

```
module top;                          <- XCModule
    reg[1:0] a = 0;                  <- XCBaseSig, (with type XCPackedArrayType, with initial value XCInt)
    reg[3:0] b;                      <- XCBaseSig
    reg clk = 0;                     <- XCBaseSig, (with type XCBuiltinType, token 'reg')
    initial begin                    <- XCProcess, (with token 'initial')
        # 1 clk = ~clk;              <- XCDelayContrl stmt, (with delay Expr, and XCBlockingAssign stmt)
    end

    M inst(clk, a, b);               <- XCInst, (with XCPortConnect)
```

```
    initial begin
        # 10;
        $display(b * 5 + 6 - b / 2);     <- XCSysTaskCall()
        $finish();                       <- XCSysTaskCall()
    end
endmodule
```

```
module M(input reg clk, input wire[1:0] a, output reg[3:0] b);
    reg[1:0] c = a;
    always @(posedge clk) begin      <- XCProcess, (with token 'always')
        c = c + 1;                   <- XCBlockingAssign
        if (c == 0) begin            <- XCIfElse
            b = b + 1;               <- XCBlockingAssign
        end
    end
endmodule
```

对于任意 DB 节点，都可以使用 *node->GetPrettyPrintedString()*获取打印出来的用户可读的 RTL 代码。

*本赛题指南未尽问题，见赛题 Q&A 文件

2025 China Postgraduate IC Innovation Competition · EDA Elite Challenge Contest

1. Problem

SystemVerilog Simulator Performance Optimization

2. Problem Designer

XEPIC Corporation Limited

3. Problem Background

Digital front-end verification simulation (Simulator) is a core process in the integrated circuit (IC) design and verification, with its significance spanning the entire lifecycle from design conception to chip mass production. Digital simulation in IC design is a critical competitive factor that determines the success or failure of a chip. It utilizes computers combined with test stimuli to simulate the chips' operation in real-world conditions, enabling engineers to check whether the results meet expectations through various debugging methods. By employing "virtual trial-and-error", it confines design risks to the front-end verification phase, where costs are minimized, making it a key enabler of "first-time success" in modern IC design. Digital simulators are typically classified into two types: Event-Driven and Cycle-Based. Event-Driven simulators simulates circuit state updates via an event queue, where each signal change triggers the recalculation of related logic, requiring strict adherence to timing accuracy.

In the field of IC verification, digital simulators read hardware description languages

(HDLs) such as SystemVerilog, process them during compilation to generate executable files, and then simulate circuit behavior at runtime. With the exponential growth in IC complexity—from early chips with tens of thousands of gates to today's billion-gate SoCs—excessively long simulation times and enormous memory requirements have become critical bottlenecks hindering verification efficiency.

Compile-time optimization is a key step in transforming HDLs into efficient simulation models. Its core objective is to eliminate redundant computations and construct specialized execution engines tailored for simulation scenarios through static analysis and code rewrite, thereby improving simulation performance and reducing simulation memory.

In general compilation techniques, compile-time optimizations can typically be classified by their objectives and methods into computation optimizations (e.g., constant folding, algebraic simplification), control flow optimizations (e.g., condition merging, loop optimization, dead-code elimination), and memory access optimizations (e.g., loop tiling). Some of these optimization strategies might directly bring results at excellent compile time, while others can only achieve effectiveness after adaptation based on specific language features. Particularly, given the event-queue scheduling nature of digital simulators, optimizing the scheduling queue or events during compilation is one of the unique compile-time optimization techniques for digital simulators. Actually, compile-time optimizations may face challenges such as trade-offs between increased compilation time and reduced runtime execution, uncertainty in optimization effectiveness, and potential disruption to debugging functionality.

X-EPIC digital simulator product "GalaxSim" is an event-driven digital circuit

simulator. As an independently developed, high-performance, new-architecture simulator, GalaxSim was created through the collective expertise of industry veterans and refined in collaboration with several leading domestic clients. GalaxSim innovatively uses a new software framework that provides multi-platform support and has been successfully tested on multiple domestic ARM-based platforms. GalaxSim supports various usage modes and is compatible with diverse verification tools for co-simulation. GalaxSim provides a unified data interface, fully supporting IEEE 1800 SystemVerilog syntax, IEEE 1364 Verilog syntax, and IEEE 1800.2 UVM methodology. Suitable for verification tasks at all levels—from IP and SoC to Chiplet validation—GalaxSim delivers excellent applicability across a wide range of scenarios.

This contest problem requires participants to explore simulation optimization using X-EPC digital simulator GalaxSim. partial underlying DB APIs of GalaxSim will be exposed. Leveraging the dynamic auto-loading mechanism of GalaxSim's compile-time optimization module, participants should devise several compile-time optimization strategies and run the simulation correctly with improved runtime performance and acceptable increased compilation overhead. Participants can utilize the simulator framework and runtime diagnostic tools of GalaxSim to identify potential optimization opportunities. Through solving this problem, participants will have the opportunity to directly engage in the development of a leading domestic EDA digital simulator, contributing to its advancement.

The exposed underlying data modules are C++ APIs, so participants should master basic C++ development skills. Having some compilation background may help participants get started more easily. Basic IC knowledge could aid in better understanding the simulation

process, but it is not mandatory requirements.

4. Problem Analysis

This section uses examples of common compile-time optimization techniques to illustrate how compile-time optimizations impact runtime performance. Participants may refer to additional materials for more optimization techniques.

1. General Compilation Optimization Techniques

c. Loop Invariant Code Motion

```
// Example 1
// Before optimization:
int a;
// ...
for (int i=0; i<100; i++) {
    integer x = a * 2;
    y[i] = x + i;
}

// After optimization:
// a*2 is loop-invariant and has been moved outside the loop to avoid redundant computations during iteration
int a;
// ...
integer x = a * 2;
for (int i=0; i<100; i++) {
    y[i] = x + i;
}
```

d. Conditional Statement Optimization

```
// Example 2
// Before optimization:
// This coding style is commonly seen in HDL code for digital circuit design
if (a == b) $display(123);
if (a == b) $display(456);

// After optimization:
```

```
    // By merging conditional statements that have same conditions, the number of condition judgments can be reduced
// For example, using specific values of control signals or enable signal as conditions
    if (a == b) begin
        $display(123);
        $display(456);
    end
```

2. Application of General Compilation Optimization Techniques to Digital Simulators

```
    // Example 3
    // Before optimization:
    assign b = c;
    assign a = b;

    // After optimization:
    // In digital circuits, within this set of assignments, “b” essentially acts as a group of wires, allowing direct connection between net “a” and net “c”,
// thereby reducing one value propagation.
    assign a = c;
```

```
    // Example 4
    // Before optimization:
    assign a = b + c + d + e + f + g

    // After optimization:
    // This optimization reduces computational load
// For example, when only “f/g” values change, recompute of "b + c + d + e" can be avoided
// In digital circuits, this optimization increases value propagation overhead, so its application requires careful consideration
    assign a = tmp0 + f + g
    assign tmp0 = tmp1 + d + e
    assign tmp1 = b + c
```

```
    // Example 5
    // Before optimization:
    always @(*)
    a = b;

    // After optimization:
    // The scheduling cost of always blocks is higher than the value propagation cost of continuous assignments.
    assign a = b;
```

3. Digital-Simulator-Specific Compilation Optimization Techniques

```
// Example 6
// Before optimization:
reg [255:0] w_data;
reg [256*2048 - 1 : 0] data;
assign data[0+:2048] = {w_data, 1792'b0};
assign data[2048+:2048] = {8'b0, w_data, 1784'b0};
assign data[4096+:2048] = {16'b0, w_data, 1776'b0};
//...
assign data[24576+:2048] = {96'b0, w_data, 1696'b0};

// After optimization:
// Through optimization, the size of the event queue scheduling list in this example is significantly reduced to just 1.
function logic [24576+2048-1:0] F(input logic [255:0] _unused);
    F[0+:2048] = {w_data, 1792'b0};
    F[2048+:2048] = {8'b0, w_data, 1784'b0};
    //...
    F[24576+:2048] = {96'b0, w_data, 1696'b0};
endfunction
assign data[24576+2048-1 : 0] = F(w_data)
```

5. Problem Description

To accurately reflect the intended optimization effects of participants' designed algorithms, the existing optimization features in GalaxSim will be disabled. The version of GalaxSim with optimizations disabled is define as **GalaxSim-Base**. All participants' optimization algorithms are executed and tested based on GalaxSim-Base.

1. Test Case

In this contest problem, test cases are defined along two dimensions. The specific definitions and the number of provided test cases are shown in the table below, where the numerical value following each category title indicates its scoring weight:

	Public Cases (0.3)	Hidden Cases (0.7)

Basic Cases (0.2)	4	Undisclosed
Comprehensive Cases (0.8)	2	Undisclosed

Test Case Categories:

- **Public Cases**
  - Fully accessible to participants.
  - Used for optimization research and local scoring evaluation.
- **Hidden Cases**
  - Completely inaccessible until the contest conclusion.
  - Used exclusively for closed-book evaluation.
- **Basic Cases**
  - Typically contain 1-2 bottleneck issues (e.g., a single expression occupying tens of thousands of lines).
  - Relatively simple test structures.
- **Comprehensive Cases**
  - Represent real-world IC designs (e.g., CPUs, DSPs, AXI bus modules).
  - May contain multiple simulation bottlenecks.
  - Complex test structures (e.g., or1200, c910, Xiangshan).
- **Extended Cases**
  - Developed by participants for challenging optimization scenarios.
  - Evaluated by X-EPIC team. Valuable cases will be added to the public case pool (classified as basic/comprehensive).
  - Adopted cases receive cash rewards. For details of submission formats,

timelines, and other rules, refer to the official contest guidelines.

For this contest problem, participating teams will receive **6 reference design examples** in advance, including:

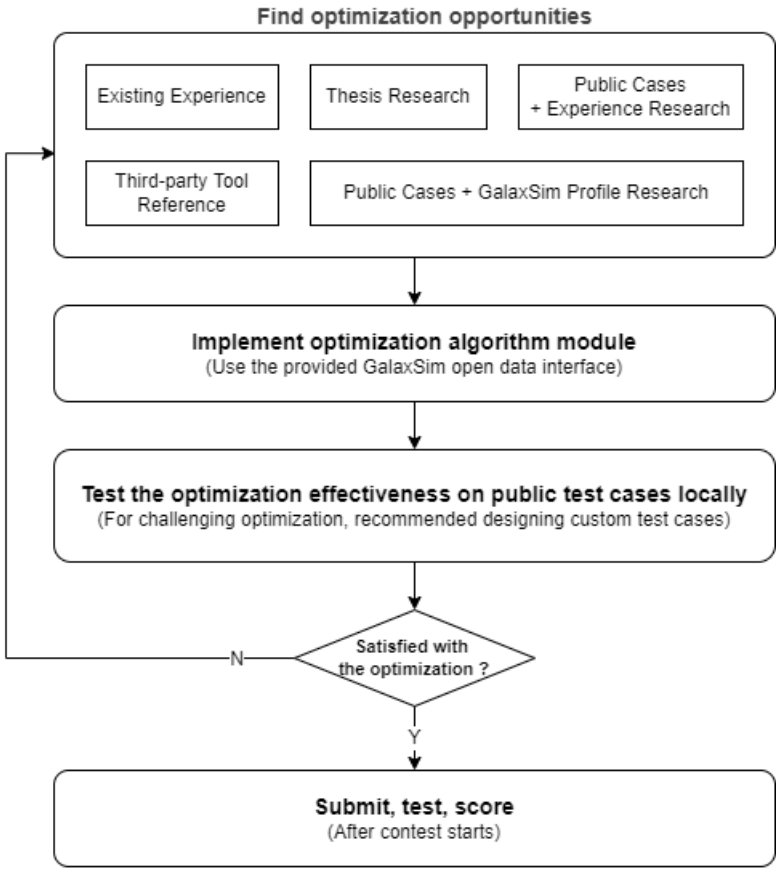
- **4 Basic Cases** (with 1-2 bottleneck issues)
- **2 Comprehensive Cases** (representing real-world IC designs like CPUs/DSPs)

Additionally, **Hidden Cases** will be provided exclusively for closed-book evaluation. All hidden cases will be disclosed to participants after the contest conclusion.

For all test cases, a set of GalaxSim-Base reference data obtained through scientific sampling methods are provided, which serve as the scoring benchmark for evaluating the optimization effectiveness of participants' algorithms.

2. Optimization Flow

GalaxSim license resources are provided for participants to conduct algorithm development and optimization debugging on offered cloud servers or own local machines.

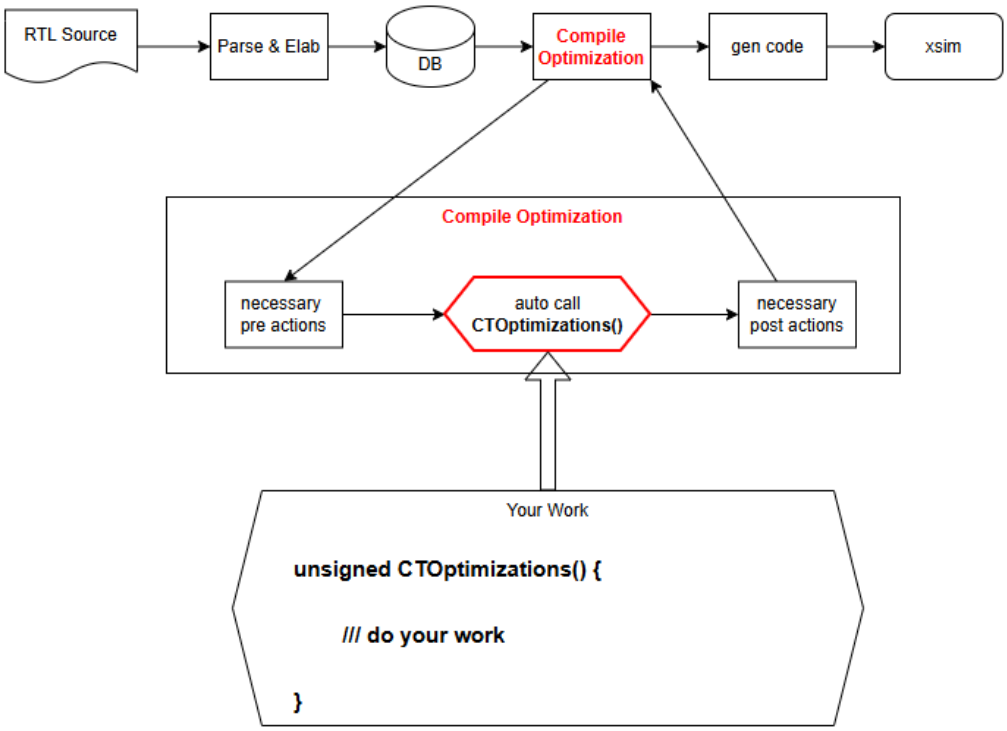


GalaxSim-Base provides diagnostic options, allowing participants to evaluate algorithm optimization effects during both compilation and runtime. After the contest begins, a dedicated scoring tool will execute GalaxSim-Base and participant optimization libraries, automatically compare results with benchmark data, and generate test case scores. For specific scoring calculation methods, please refer to the “Scoring Criteria”.

In addition to the disclosed optimization strategies in the form of test cases, participants are encouraged to acquire the knowledge of design compile-time optimization through various channels to achieve better optimization results and improve test case scores.

The following diagram illustrates how participants’ optimization strategies will be

integrated into GalaxSim.



3. Submission Instructions

Participants' optimization algorithms should use **void CTOptimizations()** as the main entry point. All optimization work, including but not limited to implementing multiple optimization algorithms and iterative cycles between them, must be completed within this function. Participants must submit their optimization algorithms as a dynamic library named **optimizations.so**.

Participants' submissions shall be in ZIP format with the unified filename **submission.zip**. The zip file must contain the dynamic library file of the optimization algorithm (**optimizations.so**) and required third-party header files and library files. Note that the optimizations.so file must be placed in the **root directory** of the submission package,

while other dependency files can be organized in directory structures at participants' discretion.

After the submission.zip file is uploaded to the scoring server, the zip file is automatically extracted and GalaxSim main program is executed. GalaxSim loads the optimization algorithm dynamic library submitted by participants. Upon successful execution of compilation and simulation run, the scoring tool generates the evaluation score.

Participants must ensure semantic equivalence of the SystemVerilog design before and after optimization and guarantee the reliability of the provided program. The test case score **will be 0** if (including but not limited to)

- The participant's optimization alters semantic equivalence causing simulation failure.
- The program contains errors that lead to compilation failure, such as unhandled null pointers, memory leaks due to unreleased pointers, etc..

Additionally, excessive compilation overhead negatively impacts the final score.

6. Scoring Criteria

GalaxSim is restricted to single-threaded execution. For scoring purposes, all runtime-related metrics are defined as end-to-end **Wall Time**. All scoring tests are conducted using scientific sampling methodology: each test case is executed 10 times on a machine without other workloads, after which the maximum and minimum runtimes are discarded, and the arithmetic mean of all remaining measurements is taken as the final sampled result for that test.

The improvement ratio for a single metric is defined as follows:

Ratio = GalaxSim Base Data / Your Data

where one benchmark dataset represents the performance data of GalaxSim-Base (with compile-time optimizations disabled by default). The **GalaxSim Base Data** is provided in advance, while **Your Data** is collected through scientific sampling after each submission.

For example, suppose a participant team achieves the following optimization results on a test case: memory consumption remains unchanged, compilation performance decreases to 80% of baseline (time becomes 1.25x baseline), and runtime performance improves to 2x baseline (runtime becomes 0.5x baseline). Then the team's performance score for this test case would be:

$$Perf\_per\_case = 0.1 * 1 + 0.2 * 0.8 + 0.1 * 1 + 0.6 * 2 = 1.56$$

After the correctness check, the ranking score (RankScore_per_case) is calculated based on performance score: Given **N** participants and a rank position **M**, the score is computed using the following formula:

$$RankScore\_per\_case = 10 * (N - M + 1) / N$$

The first place scores 10 points, the second place scores (N-1)*10/N points, and so on.

For example, if there are 26 participating teams and a team ranks 11th in performance score for a particular test case, then the team's ranking score for that test case would be:

$$RankScore\_per\_case = 10 * (26 - 11 + 1) / 26 = 6.15$$

The final total score is obtained by calculating the weighted sum of scores from each

test case.

The scoring weight is as follows:

Case Type	Summation Weights for Ranking Scores
Public	0.3
Hidden	0.7
Basic	0.2
Comprehensive	0.8

Calculation formula for the final total score of participating teams:

$$\begin{aligned} \text{FinalScore} = 0.3*0.2* &\sum \text{RankScore(public basic cases)} \\ +0.3 * 0.8 * &\sum \text{RankScore(public comprehensive cases)} \\ + 0.7 * 0.2 * &\sum \text{RankScore(hidden basic cases)} \\ + 0.7 * 0.8 * &\sum \text{RankScore(hidden comprehensive cases)} \end{aligned}$$

Cautions:

❖ CM/CT/RM/RT data statistics utilize GalaxSim's built-in profile mechanism.

❖ If a participating team has no submission record for a test case, all improvement ratio metrics will be 0.

❖ If a test case passes compilation but fails during runtime, all improvement ratio metrics will be 0.

❖ All numerical values in score calculations retain 2 significant digits (discarding digits beyond the 3rd significant digit, with rounding applied to the 3rd digit).

❖ In case of tied final scores between multiple teams, ranking will be determined in the following order:

1. Earlier teams achieving the score will rank higher.
2. If timing is identical, teams will be ranked by comprehensive case scores.
3. If comprehensive case scores are identical, ranking will be determined by RTR (Runtime Ratio) scores of comprehensive cases.

7. References

[1] IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language, 2017

[2] R. Allen and K. Kennedy. Optimizing Compilers for Modern Architectures. Academic Press, 2002. ch12.3

8. Appendixes

1. Event-Driven Simulation Model

The simulation of SystemVerilog-described designs (DUT) and testbenches (TB) operates on a discrete-event execution model, which can generally be divided into **update events** and **evaluation events** - both defined as events. An update event typically refers to changes in net/variable values. evaluation events responses to update events, where update events trigger evaluation events.

In event-driven simulators, another important concept is time (or simulation time). From the perspective of real-time progression, time is decomposed into a time sequence, analogous to the square wave sequence of clock signals in digital circuits. The time sequence consists of a set of time slots, where a single time slot can typically be simply regarded as one clock cycle in digital circuits.

To more precisely define the relationships between events, the simulation model typically divides each individual time slot into multiple regions of different event types, scheduling various RTL processing operations into their corresponding regions for execution by the simulation model. For example, nonblocking assignments are scheduled to the NBA (Nonblocking Assignment) events region for execution. The SystemVerilog 2017 LRM (Language Reference Manual) defines 17 distinct event regions. For competition participants, it is not necessary to fully master the differences between these various scheduling regions; however, understanding this aspect could enable deeper optimizations of the event-driven simulator.

Attached below is the pseudo-code of the simulation algorithm from the SystemVerilog LRM for interested participants to better understand the event-driven simulation model. For reference, please also consult Chapter 4 of the SystemVerilog LRM manual directly.

```
execute_simulation {
    T = 0;
    initialize the values of all nets and variables;
    schedule all initialization events into time zero slot;
    while (some time slot is nonempty) {
        move to the first nonempty time slot and set T;
        execute_time_slot (T);
    }
}

execute_time_slot {
    execute_region (Preponed);
    execute_region (Pre-Active);
    while (any region in [Active ... Pre-Postponed] is nonempty) {
        while (any region in [Active ... Post-Observed] is nonempty) {
            execute_region (Active);
            R = first nonempty region in [Active ... Post-Observed];
            if (R is nonempty)
                move events in R to the Active region;
```

```
            }
            while (any region in [Reactive ... Post-Re-NBA] is nonempty) {
                execute_region (Reactive);
                R = first nonempty region in [Reactive ... Post-Re-NBA];
                if (R is nonempty)
                    move events in R to the Reactive region;
            }
            if (all regions in [Active ... Post-Re-NBA] are empty)
                execute_region (Pre-Postponed);
        }
        execute_region (Postponed);
    }
```

```
    execute_region {
        while (region is nonempty) {
            E = any event from region;
            remove E from the region;
            if (E is an update event) {
                update the modified object;
                schedule evaluation event for any process sensitive to the object;
            } else { /* E is an evaluation event */
                evaluate the process associated with the event and possibly
                schedule further events for execution;
            }
        }
    }
```

2. Design of Open DB and Its Mapping Relationship with RTL

a. Key Syntax Elements and Database Design

According to the SystemVerilog LRM syntax reference manual, fundamental syntax elements include scope, expr, operator, statement, task/function, data type, and process. Based on these core syntax elements, the basic design diagram of the underlying GalaxSim database we have opened is as follows.

Common HDL syntax elements and their representations:

✓ XCNode

Base class for all syntax element classes, providing interfaces

like **GetLoc()** and **GetAttribute()**.

✓ **XCScope**

Corresponds to the SystemVerilog scope concept. Any syntax element that can contain "definitions" internally (such as module, block, class, task, function, etc.) can be called an XCScope.

XCScope provides corresponding interfaces for CRUD operations on the contained syntax elements. Refer to Section 23.9 of the SystemVerilog LRM manual.

✓ **XCStmt**

Statements, which can be nested.

✓ **XCExpr**

Expressions, which can be nested.

✓ **XCType**

Base class for the type system.

✓ **Others**

Representations of other syntax elements.

✓ **XCDesign**

Represents the root node of the use's entire design.

Additionally, an implementation of the Visitor pattern is provided to support participants in traversing the design.

b. Examples of RTL-to-Database Design Mapping Relationships

```
module top;                          <- XCModule
    reg[1:0] a = 0;                  <- XCBaseSig, (with type XCPackedArrayType, with initial value XCInt)
    reg[3:0] b;                      <- XCBaseSig
    reg clk = 0;                     <- XCBaseSig, (with type XCBuiltinType, token 'reg')
```

```
        initial begin                   <- XCProcess, (with token 'initial')
                # 1 clk = ~clk;         <- XCDelayContrl stmt, (with delay Expr, and
XCBlockingAssign stmt)
        end

        M inst(clk, a, b);              <- XCInst, (with XCPortConnect)
```

```
        initial begin
            # 10;
            $display(b * 5 + 6 - b / 2);      <- XCSysTaskCall()
            $finish();                        <- XCSysTaskCall()
        end
    endmodule
```

```
    module M(input reg clk, input wire[1:0] a, output reg[3:0] b);
        reg[1:0] c = a;
        always @(posedge clk) begin       <- XCProcess, (with token 'always')
            c = c + 1;                      <- XCBlockingAssign
            if (c == 0) begin               <- XCIfElse
                b = b + 1;                    <- XCBlockingAssign
            end
        end
    endmodule
```

For any database node, the user-readable RTL code can be obtained using **node->GetPrettyPrintedString()**.

*For questions not covered in this guide, please refer to the Q&A document