基于并发通信顺序进程和 ConPro 综合框架，利用现场可编程门阵列对算法、虚拟化机器和通信协议进行基于规则的高级硬件 RTL 综合

Stefan Bosse[1,2].3[1]不来梅大学数学与计算机科学系，德国不来梅[2]锡根大学机械工程系，德国锡根[3]数字化研究所，德国不来梅

sbosse@uni-bremen.de

摘要 — 虚拟化是对细节的抽象。算法和编程语言同样提供抽象。在异构环境和网络（例如分布式及材料集成传感器网络）中，硬件和嵌入式系统的虚拟化正变得愈发重要。通过引入直接在芯片上以硬件实现的虚拟化层，可以统一并以唯一方式访问各种硬件及低级协议范围内的通信和数据处理。如今，硬件设计仍以组件驱动（如电路板）为主，而不是将算法作为抽象层直接转换为硬件设计。程序和协议都是算法，因此不要像传统高级综合设计流程那样将它们当作设备处理！在片上系统设计中，具有占主导地位且复杂控制路径的复杂反应式系统日益重要。并发性的主要贡献体现在控制路径层面。本文深入介绍了采用最高层次综合 ConPro 编译器框架的片上系统设计方法，以及一种面向进程的编程语言。该语言提供基于并发执行、彼此通信的顺序进程（CCSP）的编程模型，并包含丰富的进程间通信原语。电路在算法层级进行建模和编程，相较于组件驱动设计，这种方式更为便捷自然。文中展示并评估了智能通信协议路由器和先进的基于栈的处理器的扩展案例研究；后者提供了程序化虚拟化层。二者共同构成一种智能节点架构，部署于高密度传感器-执行器网络中，例如材料集成智能系统。

关键词 — 高级综合；虚拟化；虚拟机、硬件抽象层；损伤诊断；材料集成嵌入式系统；实时系统；通信协议；栈式虚拟机；传感器网络；

软件和代码 — 所有软件和代码示例（案例研究）均可从以下地址下载：
`https://github.com/bsLab/conpro2`

目录

引言.2
ConPro SoC 高层次综合.5 综合流程.5 微代码中间表示.8 综合规则.10 参考栈（*RS*）优化器与调度器.14 基本块调度与数据通路并行化.17 *RTL* 综合与 *VHDL* 模型.19 *The* µ*RTL* 编程语言.25 *ConPro* 编程语言.27 *Hello World.28*

进程组合.30 数据存储对象.32 信号与组件.33 数组与结构类型.33 异常与处理.35 函数与过程.37

模块.37
ConPro 构建块.40 控制与数据处理语句.41 硬件架构.44

进程.44
模块.44
互斥调度器.45
函数.47
软件架构.48 案例研究.48

哲学家就餐问题.48 自适应通信协议路由器 *SLIP.50* 并行代码变换虚拟机 *uFORTH.57* 结论.63

参考文献.63 附录：*ConPro* 编程语言.64

I 引言

近年来，应用于嵌入式系统和传感器节点的低功耗微处理器和微控制器，在能耗与计算能力之比方面取得了显著进步。然而，随着复杂度不断提高，同时还要满足低功耗、实时性和小型化等约束，采用专用数字电路的片上系统（SoC）设计也日益受到青睐。将此类 SoC 在结构上分解为彼此独立的子模块，需要采用能够满足芯片面积和功耗限制的智能通信网络和协议（片上网络，NoC）。传统上，SoC 由微处理器内核、内存和外围组件构成。

架构和硬件层级上的寄存器传输级逻辑（RTL）必须从算法级推导而来，这要求提升 RTL 的抽象层次 **[ZHU01]**。但一般而言，大规模并行系统要求在控制和数据通路层级上对并行性与并发性进行建模。数字逻辑

系统和现场可编程门阵列（FPGA）作为快速原型工具，更适用于在控制和数据通路层级上探索和实现并行性。

采用特定指令集架构（ISA）的程序控制微处理器和微控制器的硬件设计与综合已得到广泛应用。ISA 通常由软件编译器使用。相比之下，虚拟机（VM）定义了独立于底层处理器 ISA 的自身 ISA （例如，系统虚拟机和进程虚拟机 qemu ）。VM 通常是软件。不过，虚拟化作为一种抽象、简化和统一的方法，也可以直接在硬件中实现，从而构建本文所探讨的虚拟化机器（ViM）。ViM 通常具有无缝更新和扩展 ISA 的能力。一般而言，算法也可以视为一种虚拟化，因为它们实现了某种将输入数据映射为输出数据的函数。

目前有许多综合工具和建模语言可用于创建 SoC 电路，但它们主要必须由 System-C模型和编程语言驱动 **[MEE12]**。设计人员分为软件设计人员和硬件设计人员。软件设计人员必须使用现有硬件，并遵循硬件规定的特定数据类型、数据范围、数据流、指令集和控制流约束。硬件设计人员则可以完全自由地确定范围、数据类型、数据流、操作和控制路径设计，包括利用并行性。Verilog 或 VHDL 等常见硬件建模语言赋予设计人员这种自由，但其显著缺点是需要专业知识，而且模型复杂度会急剧增加。优秀的高级综合（HLS）工具应允许设计人员自由选择数据路径和控制路径设计，并让他们在源代码层面验证这些选择。

HLS 的设计探索能力是其关键优势之一。HLS 能够根据设计需求考察多种架构并选出最优方案，而不是预先确定架构，再手动编写寄存器传输级（RTL）设计。HLS工具引导这一探索过程的方式各不相同。为确保清晰性和可移植性，源代码（行为）与设计限制（架构）之间应尽可能彻底地分离。HLS工具还应具备另一项重要特性：设计和架构的可扩展性与适应性。**[MAN17]**

除了设计完整且自包含的应用或领域专用电路外，加速器也是扩展现有软件控制系统的片上系统中的典型示例。然而，异构性和硬件专用化降低了加速器在不同平台间的规律性和可复用性，从而阻碍加速器的设计与集成 **[PIC22]**。HLS 支持丰富且快速的设计空间探索（DSE），但这一过程大多仍需手动完成。自动化设计空间探索需要专业知识。

使用广泛且广为人知的编程语言进行硬件设计，是触达更广泛开发者群体的一种有吸引力的方式。例如，Circe **[RAD02]** 使用 C语言的一个变体，并生成 RTL VHDL 结构化硬件描述。由于对象引用（使用指针）具有透明性，使用 C语言这类通用存储器映射语言会使 RTL 硬件综合变得困难，从而阻碍 RTL 映射。此外，大多数软件语言都缺少并发模型。人们曾多次尝试使用类似 C语言的语言，但存在诸多限制，例如禁止使用指针进行匿名内存访问，或采用由程序控制的（多）处理器架构，并结合传统的硬件软件协同设计；这种方式目前在 SoC设计中占据主导地位。但采用通用或应用专用处理器架构的 SoC设计会使低功耗设计更加复杂，而且并发粒度较粗。

赛灵思及其客户为电信和视频处理等应用开发的许多高度并行 FPGA系统，都需要使用一个或多个额外的微处理器。然而，管理运行在微处理器上的软件之间的交互，以及现有 FPGA内核之间的交互，颇具挑战性 **[PER13]**。

随着复杂度不断提高，需要采用更高的抽象层级，从硬件层级逐步提升到算法层级。人们使用天然具有命令式特征的编程语言，在程序控制机器上实现算法；这些机器处理数据与控制操作构成的顺序流。基于这种数据处理架构，更高级的命令式语言可以简单地映射到较低级的

命令式机器语言；这种基于规则的映射由软件编译器自动执行。

然而，在电路设计中，既没有现成的架构，也没有能够直接从更高级语言综合而来的低级语言。此外，并不存在适用于并行数据处理系统的预定义高效计算机体系结构。

另一个例子是 PICO **[KAT02]**, ，它涵盖完整的硬件设计流程，目标是片上系统以及可定制或可配置处理器，并配有定制设计的硬件模块（加速器）。它使用 C语言对寄存器传输级进行建模。采用处理器模块的程序控制方法可实现软件编译，并支持不受限制的 C语言（函数、指针），但不支持真正的位缩放数据对象。SPARK **[GUP04]**, 是一个 C到VHDL高级框架，目前受到若干限制：不支持指针、函数递归和不规则控制流跳转。它嵌入传统的硬件软件协同设计流程中，并基于用于探索并发性的推测性代码移动和循环变换。SPARK 生成纯RTL，但只支持单线程控制流。近期用于硬件软件协同设计的软件框架还包括 Vivaldi**[COR16]**。

尽管 SystemC 提供了许多适用于高层次综合的特性，但它主要用于仿真和验证，且只有其中一部分能够综合为电路。它支持真正的位缩放数据类型。线程化进程（例如 Forthes 商用综合工具 Cynthesizer 所采用的进程）可用于建模并发性。事务级（TL）用于建模进程间通信。SystemC 提供了一种高层次的方法来描述硬件行为和结构，而非算法。

这些方法都无法在使用基于 C 语言的语言进行纯 RTL 电路设计时，完全满足相关需求，尤其是在提供统一的硬件、软件和并发模型方面。

本文通过深入示例和评估，介绍并阐释并发编程与处理（ConPro）HLS 工具包：一方面，它使用带有一些面向对象特性的传统过程式编程语言；另一方面，它使用扩展了相关特性的编程语言，以支持基于并发通信顺序进程（CCSP）模型组合并行系统。与其他成熟的 HLS 工具不同，ConPro 无需用户具备硬件设计方面的专业知识，不过有需要时也可以加入这方面的知识。ConPro 及其编程语言易于使用，程度可与 Arduino 软件框架相媲美。一个简单的“hello world”电路只需不到 50 行代码。

在电路设计中（与软件设计相比），“硬件编程语言”的另一项重要要求是能够对资源分配、数据类型和范围以及综合过程本身进行细粒度控制。ConPro 提供了模块级（参数化模块）所需的细粒度控制。ConPro 会生成结构化VHDL，可使用常见的RTL综合工具（如 Xilinx ISE 或 Synopsis Design Compiler）将其综合到门级。ConPro 还会生成特定于设计的综合构建文件，以便从命令行控制 RTL 综合过程。并行性在控制通路和数据通路级别均有明确建模。ConPro 也能够自动利用数据路径并行性。

本文扩展版深入介绍了如何使用基于规则的高级综合 ConPro 编译器框架和面向进程的编程语言进行片上系统设计方法。Conpro 编程语言基于并发通信顺序进程（CCSP）模型，并提供了丰富的进程间通信原语。该语言一方面是类似 C语言或 Modula 的典型过程式顺序编程语言，另一方面则天然支持并行处理，并且只需少量语句即可访问硬件组件。Conpro 简洁易用，可与 Arduino 相媲美，即使是初学者也能设计片上系统，例如使用 FPGA 原型技术以及广泛使用的 VHDL RTL 综合软件（如 Xilinx ISE Foundation 和 WebPack）。

本文展示并评估了智能通信协议路由器和先进的基于栈的处理器的扩展案例研究，后者提供程序化虚拟化层。二者共同用作智能

节点架构，部署于高密度传感器-执行器网络中，例如用于具有特定约束的材料集成智能系统。

本文的结构如下。首先，深入介绍 ConPro 综合架构与方法。第二个主要部分围绕 ConPro 编程语言展开，并通过大量典型示例加以介绍。第三个主要部分专门介绍若干扩展案例研究及其评估。本文附录以简洁的表格概述 ConPro 编程语言。

ConPro 已成功应用于大学低年级和高年级课程，教授如何根据高级规格进行硬件设计 **[教育09]**。即使是硬件设计和硬件建模经验较少的初学者，也能够创建复杂的电路。

II ConPro SoC 高层次综合

高效的硬件设计要求对组件、协议和对象有更多了解并进行更多配置，而 C语言等传统语言无法提供这些能力，例如，在架构层级上对真正的位缩放寄存器、访问方式和实现模型的支持（例如，单端口与双端口 RAM 块、静态与动态访问同步）。通用的软件方法只涵盖算法的实现，但在硬件设计中，综合得到的电路必须连接到外部世界并与之交互（其他电路、通信链路等），因此必须有一种与命令式编程模型一致、用于连接硬件块的编程模型。此外，还必须能够轻松实现同步，因为只要存在并发性（至少在控制路径层面），同步就不可或缺。软件开发者社区所采用的多进程模型为并行性建模提供了一种通用方法；这是在算法层级实现和划分反应式系统的首选方式。ConPro编程语言与综合 **[BOS10A][BOS11A]** 解决了上述问题，并引入了一种面向片上系统的设计方法，该方法采用并发多进程模型以及先进的行为编程语言，后文各节将对此进行详细讨论。

综合流程由一组规则定义，如公式（*1.1*）所示。每组规则都包含若干子集，这些子集可通过编程块级别的参数设置进行选择（例如，循环展开等调度方式，或不同的分配规则）：

$$\text{Synthesis}: CP \xrightarrow{\chi^1} AST \xrightarrow{\chi^2} \begin{cases} \mu CODE \xrightarrow{\chi^3} \begin{cases} RTL \xrightarrow{\chi^4} VHDL \\ C \,/\, OCaML \end{cases} \\ C \,/\, OCaML \end{cases} \tag{1.1}$$

II.1 综合流程

ConPro 编译器的处理架构和综合流程如**图 1**所示。综合过程采用传统的软件编译器流程，并使用中间表示（综合层 1，保留并行处理和与 CSP 相关的特性）。此外，还提供了广泛的后端，同样支持软件输出（综合层 3）。

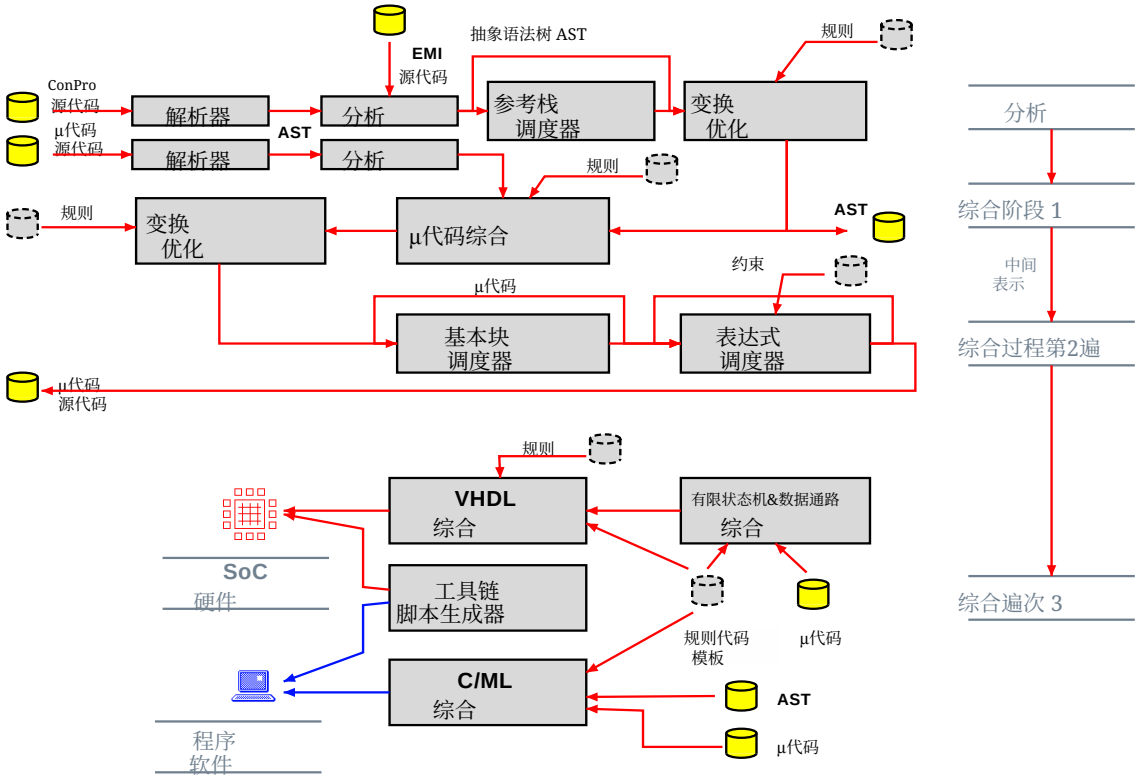


图 *1* 使用高级综合框架 *ConPro* 的设计流程，该框架将并行 *CCSP* 编程模型映射到 *SoC-RTL* 硬件及其他软件目标平台。

从高级*ConPro*程序源代码综合 RTL 数字逻辑电路需要经过不同阶段：

1. 首先，对源代码进行解析和分析。针对每个进程，构建一个保留复合语句的抽象语法树（AST）。全局和局部对象存储在符号表中（模块级别有一个全局符号表，每个进程级别各有一个符号表）。首先，对进程指令图进行优化，例如常量折叠和死对象检查，并删除这些对象及多余语句。执行多种基于规则和模式匹配的程序转换，包括推断临时表达式和寄存器。一种称为参考栈调度器的符号化源代码分析方法 **[KU92]**, 会检查（局部）数据存储对象及其在表达式中的历史记录。参考栈调度器使用表达式栈分析数据存储表达式的求值过程，每个对象对应一个表达式栈。具有表达式 $\kappa=\{\Theta\leftarrow\varepsilon_1, \Theta\leftarrow\varepsilon_1, ..\}$ 的一系列存储赋值，其对应的特定存储对象 Θ 会由参考栈转换为一系列不可变且唯一的符号变量 $\omega i : \{\omega_1\leftarrow\varepsilon_1, \omega_2\leftarrow\varepsilon_1, ..\}$。目标是通过向后代换和常量折叠来减少语句和多余存储。参考栈调度器采用尽可能晚的调度行为

2. 在指令图级别完成分析与优化后，AST 中的复杂指令（范围从表达式到循环）将被转换为一系列线性排列的 μCode 指令。μCode 级别是程序代码的一种中间表示，也用于软件编译器；但在该级别，除控制流约束外，不作任何特定于体系结构的假设。μCode 也可以导出和导入。此功能使其他编程语言前端能够采用不同的

入口级别，例如函数式语言[SHA98]。这种中间表示支持更细粒度的优化、分配和调度。从语法图到 μCode 的转换会推导出辅助指令和寄存器（例如，需要初始化、条件分支和循环计数器语句的 for 循环）。数据通路级别的并行性由 bind 指令提供，它可将多条指令绑定到同一个执行步或同一个有限状态机状态（一个时间单位）。

该变换基于一组规则 $\chi_{\kappa\to\mu}$，由默认规则和用户可选规则（约束规则）组成。这是架构综合的第一阶段，在此阶段，源语言范式被替换为目标机器范式；在本例中，目标机器是有限状态机，语句映射到状态，表达式映射到数据通路（寄存器传输级）。此外，分配的第一阶段也在此处进行。数据通路并发可通过用户指定的有界块或基本块调度器进行探索。该调度器将 mCode 指令划分为基本块。这些块在顶部只有一个控制路径入口，在尾部只有一个退出点。一个基本块（称为“主块”）中的指令会进一步划分为次级块（至少包含一条指令或一个有界块）。根据这些次级块构建数据依赖图（DDG）。最后，调度器从这些 DDG 中选择具有尽可能早行为的数据独立指令。

3. 在第一级综合之后，中间μ代码依据一组规则$\chi_{\mu\to\Gamma\Delta}$映射到抽象状态图RTL上，这些规则包括默认规则和用户可选规则。最后的转换步骤会生成VHDL代码。这种设计选择使系统能够在不改变主综合路径的情况下，添加其他或新型硬件描述语言，例如Verilog。规则集决定了临时寄存器和功能块的资源分配，并提供不同的分配策略：共享对象与非共享对象、扁平化与共享函数运算符，以及临时寄存器推断。共享寄存器和功能块会在数据通路中引入信号选择器。

RTL单元被划分为有限状态机FSM和数据通路两部分。有限状态机FSM由两个硬件块组成：一个时序硬件块用于实现状态寄存器，另一个组合逻辑硬件块用于实现状态切换网络，并由此提供控制路径；数据通路则由时序硬件块和组合逻辑硬件块构成，用于实现函数运算符，并访问全局资源和局部寄存器。

4. 使用默认规则集时，每条 μCode 指令（有界块中的指令除外）都会映射到有限状态机的一个状态，并需要一个时间单位（≥ 1 时钟周期的执行时间，具体取决于对象守卫条件和进程阻塞）。由规则集 $\chi_{\kappa\to\mu}$ 而非 $\chi_{\mu\to\Gamma\Delta}$ 决定调度。

尽管此编译器流程未采用传统的迭代调度和分配策略，但这种非迭代、约束选择式的基于规则的综合方法，仍能实现固有的调度和分配，并且会受到不同优化器的显著影响。总而言之，调度和分配可分为不同层级：

参考栈调度器
在语法图层级运行，尝试减少语句、函数运算符和存储，并对调度和分配产生影响。

基本块调度器
作用于中间 mCode 层，尝试减少语句的操作时间步，但仅影响调度。

表达式调度器

为满足时序和最长组合路径约束，主要是时钟驱动的复杂扁平表达式和嵌套扁平表达式必须使用临时寄存器进行分割，并采用扩展调度。该调度器同时影响调度和分配。

优化器

传统的常量折叠、死代码和对象消除，以及循环/分支不变代码转换，进一步减少了时间步和资源（运算符和存储）。

综合规则

综合规则集合对调度和分配的影响最大$\chi = \chi_{\kappa\to\mu} \cup \chi_{\mu\to\Gamma\Delta}$

II.2 微代码中间表示

在第一个编译阶段，ConPro 源代码会经过解析和分析，为每个进程生成复杂指令或环境的符号列表和语法图。在第二个编译阶段，进程指令图结构会被展平为微代码程序指令的线性列表（μCODE）。该中间表示可以使用 VHDL 和 Verilog 模型直接映射到寄存器传输级，将μCODE 的线性列表划分为控制通路和数据通路。更通用的μCODE 形式是通用且更自然的寄存器传输级语言 μRTL，详见第**III**节。μCODE 是μRTL 的指令式汇编器版本。

这种方法的主要优势在于，可以处理来自小型指令集的简单指令线性列表，而不必处理具有不同复杂编程环境和控制语句（例如 for 循环）的复杂图结构。此外，这种微代码方法支持采用更传统的软硬件协同设计来综合和实现微处理器内核；本文不讨论这一点，但它为综合流程的未来扩展提供了可能。μCODE 指令集汇总见**表 1**。

前两条指令与数据通路有关，跳转指令与控制路径有关。访问抽象数据类型对象（ADTO）及其实现需要使用函数应用指令。ADTO 通常是由多个进程共享的全局资源。它们需要互斥同步，并且能够提供更高层次的进程间通信。

表 *1* μ*CODE* 指令的语言语法：（左）指令，（右）操作数格式

指令	描述	操作数	描述
MOVE(dst,src)	数据传输 目标 ← 源	$立即.[i]	临时符号变量 立即用于下一状态 条指令
表达式(dst, op_1， 操作， op_2)	对表达式求值并 将结果赋值给dst。	$temp.[i]	临时存储寄存器
绑定(*N*)	将以下*N*条指令 绑定到一个有界块。	$算术逻辑单元.[i]	共享算术逻辑单元
JUMP(target)	跳转到程序位置 目标	DT{*DT*[*N*]}	数据类型

指令	描述	操作数	描述
条件假跳转(cond:目标)	跳转到程序位置 目标 如果 cond 为假	ET{*DT*[*N*]}	表达式数据类型
函数(name⟦,arg⟧)	对象操作或函数 应用	类型转换{数据类型[名称]}	数据类型转换
标签:	符号指令（分支目标）标签		
特殊指令(指令)	内部用途		
NOP	空操作占位符		

特殊指令和 label 指令属于辅助指令。bind 指令提供了一种绑定应并发执行的数据路径指令的方法；这些指令既可以在编程层显式设置，也可以由调度器和优化器通过利用数据路径层的并行性隐式推导得出。

这种方法一个显而易见的缺点是进程块内部的并行性会丧失，因为每条指令或每项操作至少占用一个时间步。为保留进程指令级别的并行性，引入了 bind 指令。该指令将后续的 *N* 条数据路径指令分组，并使组内所有指令在一个时间步内并发执行（由于保护锁和阻塞约束，可能需要更多时钟周期）。

µCODE 指令的操作数可以是寄存器、变量、信号或临时（符号）变量，并附带有关数据（DT）和表达式类型（ET）的额外信息（子类型），这些信息用于将操作数转换为目标对象（左侧）类型所需的类型和数据位宽转换（TC）。此外，还包含有关局部性以及左侧（写引用）或右侧（读引用）属性的信息，以及表达式求值是否使用共享算术逻辑单元的信息。µCODE 指令为后续的寄存器传输级综合提供初始调度和分配信息，其中包括表达式变换。**例 2**展示了一个简短的 µCODE 生成示例。与一个进程相关的 µCODE 汇编器模块划分为导入段、数据段和代码段。

例 2　　µCODE 综合示例

```
1   寄存器 d：逻辑[8]; 2进程 foo:3   开始 4   寄存器 a,b：逻辑[8]; 5   d <- 0;6   对于 i = 1 到 100 执行7   开始 8   a <- d, b <- d+1;9   d <- i + a + b;10   结束11   结束;12   ----------
µCODE13   导入 :14   开始 15   寄存器 d: L816   结束1718   数据:
```

19 开始 20 寄存器 循环 _ _0 9 i : I 寄存器 a: L8 22 8 寄存器 b: L 23 结束 24 25 代码: 26 开始 27 i1_赋值: 28 MOVE (d,0) 29 i1_赋值_结束: 30 空操作 31 i2_循环_循环: 32 MOVE (循环_i_0, 1) 33 i2_循环_循环_条件: 34 绑定 (2) 35 表达式 ($immed.[1],100,>=,循环_i_0) 36 条件假跳转 ($immed.[1],%END) 37 i3 绑定 到 4: 绑定 (3) 39 MOVE (a,d) 40 表达式 (b,d,+, 1) 41 空操作 42 i3 绑定 到 4 结束: 空操作 44 i5_赋值: 45 绑定 (2) 46 表达式 ($immed.[2],循环_i_0:L0,+,a) 47 表达式 (d,$immed.[2],+,b) 48 空操作 49 i5_赋值_结束: 50 空操作 51 i2_循环_循环_递增: 52 绑定 (3) 53 表达式 (循环_i_0,循环_i_0,+,1) 54 空操作 55 跳转 (i2_循环_循环_条件) 56 i2_循环_循环_结束: 57 结束

21 *38* *43*

II.3 综合规则

最简 以下总结了应用于 HLS 中源代码的核心综合规则 .

规则 P：进程

每个顺序执行的 *ConPro* 进程都映射为一个 RTL 模块，其中的有限状态机负责提供计算的控制流，其行为由状态转移图 $\Gamma=(\Sigma,\psi)$ 和数据通路 Δ 定义。控制状态集合为 $\Sigma=\{\sigma_1, \sigma_2, ..\}$，控制状态转移集合为 E。整个 RTL 模块由组合硬件块和状态转换硬件块组成，硬件 $=(\Phi \cup \Pi)$。

规则 SI：存储

仅有寄存器 R，其数据宽度相互独立，范围为 $[1..W_{max}]$ 位。

寄存器名称：DT[位宽];

规则 SII：存储资源共享与分配

不存在资源共享：为每个寄存器 r∈R 分配一个状态转换硬件块 τ。

$$\text{FUN ALLOCATE}^{\text{STORAGE}}: \{r_1, r_2, .. \mid r_i \in \Re\} \rightarrow \{\tau_1, \tau_2, .. \mid \tau_i \in \Pi\}$$

规则 SIII：语句调度

来自基本语句集合 κ 的每条指令 *i* 都映射到有限状态机的控制流 Γ=(Σ,ψ) 中的一个状态 σ。基本语句集合由以下内容组成： κ = {赋值、表达式求值、 条件分支、无条件分支}。

$$\text{FUN SCHEDULE}: \{i_1, i_2, .. \mid i_i \in \kappa\} \rightarrow \{\sigma_1, \sigma_2, .. \mid \sigma_i \in \Sigma\}$$

规则 SIV：数据通路

数据通路 Δ 根据控制状态集 ϕ进行分解和划分。

规则 SV：表达式资源共享与分配

不存在资源共享。对于每个功能运算符 *op*∈ϰ，都会分配一个组合逻辑硬件块 ϕ。运算符集合由 ϰ={+,-,*,/,and, or, not, shiftl,shiftr}组成。

$$\text{FUN ALLOCATE}^{\text{EXPR}}: \{op_1, op_2, .. \mid op_i \in \aleph\} \rightarrow \{\phi_1, \phi_2, .. \mid \phi_i \in \Phi\}$$

规则 SVI：复合语句

复合语句 *i*∈ε（例如循环）使用综合分解规则 χ分解为一系列基本语句：εi→{*i*1, *i*2, .. | *i* ∈κ}。复合语句集合由 ε={如果-那么-否则、分支选择、*For*循环、*While*循环、..}组成。

一些复合语句的转换规则如下所示，见**表2**。

表2 复杂 ConPro 语句的综合转换。记号：B：语句块，μCODE：x ← y；➡ MOVE(y,x)；求值(ε?)，条件假跳转 L；➡绑定(2)；表达式(t,ε)；条件假跳转(t,L)

语句	变换
对于 cni = a 到 b，执行 Bi	分配：cni ➡ ri ➡ τi 转换：cni ← a; 循环i：求值(cni < b)? , 条件假跳转 退出i; B_i; cni ← cni + 1; 跳转 循环i; $EXIT_i$: ...

语句	变换
如果 ei，则执行 Bi,1；否则执行 Bi,2	转换：求值(ei)？， 条件假跳转 LBi ,2； $B_{i,1}$； 跳转 退出i； $LB_{i,2}$: $B_{i,2}$； $EXIT_i$: ...
匹配 ei 当 v : B_1 i,1 当 v2 : Bi,2 ... 当其他 : Bi ,n	转换：求值(ei=v1)?, 条件假跳转 LBi,2; Bi ,1； 跳转 退出i; $LB_{i,}$2: 求值(ei=v2)?, 条件假跳转 LBi ,3； $B_{i,2}$； 跳转 退出i; ... $LB_{i,n}$: $B_{i,n}$； $EXIT_i$: ...

规则 SVII：状态与转换

每个状态 σi 始终有一个后继状态 σj，且 j≠i；例外情况是 j=i，此时当前语句的执行被全局对象的保护锁阻塞（守卫条件得到满足）。状态转换可以是条件转换，由状态转换函数 COND(σi, σj , *cond*) 给出，并满足前置条件 *cond*；也可以是无条件转换，由函数 NEXT(σi, σj) 给出（执行有序语句序列）：

函数转换：(σi,σj) → {ρ | ρ∈{下一状态(σi, σj),条件(σi, σj，条件)}}

每个顺序进程都会分配一个启动状态和一个结束状态（σ_0, σ_∞）。除结束状态 σ_∞ 外，每个状态至少有一个指向其他状态的输出状态转换；结束状态 σ_∞ 只有一个输出自转换。以下 **例 3** 展示了一个 Conpro-μCODE 变换示例。

示例：从 *ConPro* 程序片段进行 *3* μ*CODE* 综合（在 μ*CODE* 优化和状态压缩期间移除 *nop* 指令）

```
CONPROs <- 0;for i = 1 to 10 do 开始 t <- 1;  s <- s + i;  if s = 0 thent <- 2;
end;end;
```

~~微代码~~

数据：寄存器循环_i_0：整数[8]

代码：

i1_assign: move (s,0) i1_assign_end: 空操作 i2_for_loop: move (LOOP_i_0,1) i2_for_loop_cond: bind (2) 表达式 ($immed.[1],10, > =, LOOP_i_0) 条件假跳转 ($immed.[1],%END) i3_assign: 表达式 (s,s,+,LOOP_i_0:I8) 空操作 i3_assign_end: 空操作 i4 branch: bind (2) 表达式 ($immed.[1],s,=,0) 条件假跳转 ($immed.[1],i2_for_loop_incr) i4 branch end: 空操作 i2_for_loop_incr: bind (3) 表达式 (LOOP_i_0,LOOP_i_0,+,1) 空操作 跳转 (i2_for_loop_cond) i2_for_loop_end:

高级综合规则包括基本块调度（在抽象语法树和 μ代码 层级）、基于引用栈的优化（在抽象语法树层级）以及资源共享（尤其涉及临时存储），这些内容将在下一节中讨论。

规则 SOI：扩展存储对象集合

存储对象集合扩展为包含存储在 RAM 块中、并通过其地址选取的变量

规则 SOII：资源共享

在进程内部，临时寄存器等本地存储可供多项计算重复使用。此外，通过采用算术逻辑单元方案，可以使用一个或多个算术逻辑单元块来共享操作（算术、关系和布尔运算）。

规则 SOIII：状态压缩

可通过合并控制语句和数据语句，减少由语句序列和复杂语句分解生成的各进程的控制状态集合及状态转换图。这主要在 μ代码 层级完成。

规则 SOIV：优化

几种传统的优化技术，如重新调度（引用栈方法）、常量折叠和死对象/代码优化，能够显著改善片上系统的资源需求和执行时间，其中包括将在下一节讨论的基本块并行化。

II.4 参考栈（RS）优化器与调度器

符号 RS 方法在抽象语法树级别采用自动调度方法，可实现存储和控制流优化，该方法分为两个阶段：

1. 在抽象语法树级别，采用尽可能晚调度的行为合并表达式，以便在初始指令边界之外进行常量折叠和寄存器/变量折叠。参考栈方法会尽可能将表达式合并为大型元表达式，从而获得更优的常量折叠结果。在此级别，所有寄存器和变量赋值都会尽可能延后。只有在全局块、分支和循环边界处，才会调度包含寄存器和变量赋值的实际数据传输。2. 在中间微码级别，根据所选的表达式调度和分配模型，对这些（大型）元表达式进行后续调度，并赋予控制步赋值。在扁平模型中，元表达式会在单个时间步内完成调度。对于二元运算或共享 ALU 模型，由 *N* 个二元运算组成的表达式会通过临时寄存器传输，在 N 个时间步内完成调度。

基本上，RS 算法（见**定义 1**）会将对可变存储对象的一系列修改转换为一系列不可变符号变量，如**例 4**所示。存储赋值的刷新会尽可能延后。控制语句可根据存储对象是否出现在条件块和循环体块中触发刷新。**定义 1**中的算法是基于 **[KU92]**改编的高级版本。

例 *4* 使用参考栈分析与优化得到的源代码块（左）的可能结果（右）（情形 *2*：在此块之后不再引用 *y*，且没有副作用）。底部：将可变存储转换为不可变符号变量

x ← a + b + 1; ❶ y ← a + b + c; y ← x + 1 + c; ➡ x ← y; x ← y - 2; ❷ x ← a + b + c; DEF x_0 = a + b + 1 DEF y_0 = x_0 + 1 + c DEF x_1 = y0 - 2 T(x)=[x1,x_0]

定义 ***1 RS*** 算法

1. 每个数据对象 *x* 都使用其自己的参考栈 *T* 进行追踪：

T(x)=[*re*i-1；*re*i-2；.....； re_0]（左侧元素是栈顶，*re*是类型为栈元素的引用表达式）数据对象 x 寄存器、变量引用表达式 类型 栈 = 栈元素栈 类型 栈元素 ⫤ RS_ 自身 (对象池) | RS_ 表达式 (表达式)

| RS_分支(栈列表) | RS_循环(栈) | RS_引用(栈元素 引用列表)

2. 一项赋值

$x_i \leftarrow \varepsilon(x1, x2, ...)$

尽可能晚地延迟。

3. 在赋值中，x 首次出现在左侧时，会为对象 x 创建一个新的参考栈：

$x_i \leftarrow \varepsilon_0(x_1, x_2, ...) \Rightarrow T(x_i)=[\texttt{RS_expr}(\varepsilon_0)]$

4. 对象 x 的新参考栈在赋值或表达式的 RHS 中首次出现 x 时创建：

$T(x_i)=[\texttt{RS_自身}(x)]$

5. 每次对存储对象 x 进行新赋值时，都会将新表达式 ε（赋值的 RHS）压入栈 $T(x)$，以更新参考栈。

$x_i \leftarrow \varepsilon_n(x_1, x_2, ...) \Rightarrow T(x_i)=[\texttt{RS_expr}(\varepsilon_n), ..]$

6. 表达式中每次出现存储对象时，都会在 AST 中标记一个指向该元素参考栈当前栈顶元素的引用（栈顶为=n-1，且有 n 个栈元素）：$\uparrow T(x1)\text{top}$

$\varepsilon_n(x_1, x_2, ...) \Rightarrow \varepsilon_n(x_1{:}\uparrow T(x_1)_{\text{top}}, x_2 : \uparrow T(x_2)_{\text{top}}, ...)$

在分支或循环内部，对于只在其中被引用而未修改的对象，值 RSRS_ref([*stack references*]) 会被放置到参考栈 $T(x)$ 上。这是调度决策所需的重要信息（RS 刷新，见下一步）。

7. 在外层调度块末尾刷新所有延迟赋值；如果此前没有分支和循环等控制语句，也要进行刷新。刷新完成后，参考栈的栈顶现在为 RS_self。解析对象和栈顶表达式的依赖关系，并按正确的依赖顺序对赋值排序，然后再刷新。

在已刷新的赋值的表达式中，将所有指向其他栈元素的引用 RS_ref 重定位。

~~示例~~

x ← 12; ✎ 延迟赋值，T(x)=[12]　　y ← x + 1; ✎ 延迟赋值，T(y)=[x$_0$+1]　　x ← v; ✎ 延迟赋值，T(v)=[v]，T(x)=[v$_0$,12]　　a ← y - 1; ✎ 延迟赋值，T(a)=[y0-1]

➡ 使用栈重定位和常量折叠刷新所有延迟赋值：

y ← y$_0$=x$_0$+1=12+1=13;

a ← a_0=y0-1=x_0+1-1=12; x ← x_1=v_0=v;

8. 分支

i. 求值分支的所有条件块 *B*cond, i，总计有 b 个不同的块，每个块对应一个分支条件，*j*=0...*b*-1。每个条件块都由其自己的子栈 *T*Bcond,i(x) 处理，其中 *n* 表示条件块的数量：

$T(x)$= [RS_分支([Tn−1;Tn−2,..,T_0],..]

条件块 *B*cond, i 内所有被修改的对象都会获得一个

RS_branch([$T_{\text{Bcond},1}$= [], .., $T_{\text{Bcond, i}}$=[RS_expr(ε),..], ..])

相应子栈顶部的栈元素。所有后续修改都存储在此RS_分支栈中。首次出现在表达式中的对象 x 会获得一个RS_分支([.., *T*Bcond, i=[↑$T(x)$top], ..])栈元素。分支的初始顶部元素 *T*0要么是对先前表达式的引用RS_ref，要么是RS_self。由于副作用，循环环境中必须使用最后一种元素。

ii. 仅在各个分支中被引用的对象，也就是说，这些对象未被修改，并且对于所有*i*=0...*b*-1, *T*Bcond,i=[RS_- self | RS_ref] | ∉ RS_expr，将被转换为对先前栈元素的引用：

$T(x)$=[RS_分支([[RS_self],[RS_self];...]),Tn-2 ,..]
➡
$T(x)$=[RS_branch([[RS_ref(↑$T_{\text{n-2}}$)],[RS_ref(↑$T_{\text{n-2}}$)];..]),$T_{\text{n-2}}$,..]

iii. 求值完所有条件块后，如果实际分支栈之前存在一个表达式RS_- expr，且在条件块末尾对象已被修改，则必须在分支指令之前压入具有RS _ [1 2 0] ≠分支(*T*n-; *T*n- ;...;*T*)和*T*iRSself RSrefRSexpr（在块内修改）的对象（预期：所有可能的分支都会修改*x*）。

iv. 修改引用栈：

a.) 存储对象 *x* 在分支内部未被修改，但被引用：

$T(x)$=[RS_分支([[RS_自身],..]),RS_表达式(εi),..]
➡
$T(x)$=[RS_引用(RS_表达式(εi)),RS_分支([[RS_引用(RS_表达式(εi))
], ..]),RS_表达式(εi), ..]

b.) *x* 在分支内部被修改：

$T(x)$=[RS_分支([[RS_表达式(εj),RS_自身], ..]),RS_表达式(εi), ..]
➡
$T(x)$=[RS_自身,RS_分支([[RS_表达式(εj),RS_自身)], ..]),RS_表达式(εi), ..]

9. 循环

i. 在求值循环体块之前，先求值循环条件表达式中出现的所有对象。 ii. 求值循环的主体块 B。首次出现在循环体表达式（以及赋值的右侧）中的对象，会在栈顶获得一个包含 T_0=RS_自身的 RS_loop([T_0]) 栈元素。首次在循环块中被修改的所有对象，都会在栈顶获得一个 RS_loop(Tloop) 表达式。此后所有修改都会存储在该 RS_loop(Tloop) 表达式中。T(x)=[RS_loop(Tloop=[..]), ..]iii. 在循环体中仅被引用或出现在右侧的对象，也就是说 Tloop =[_]RS | ∉ _ self RS expr，会被转换为对先前栈元素的引用，并且该引用也会被推入栈中：T(X)=[RS_loop([RS_self]),Tn-2, ..]á T(X)=[RS_ref(↑Tn-2), RS_loop([RS_ref(↑Tn-2)]),Tn-2, ..]

iv. 对循环块求值后，如果实际循环之前有表达式 `RS_expr`，则必须在循环指令之前推入对象 `RS_loop(`*T*`)` 和 *T*≠ `[RS_self]` （在块内修改），并且在循环块末尾内部也必须推入。

v. 修改引用栈：a.) 存储对象 x 未在循环体内修改，但被引用：T(x)=[RS_循环([RS_自身]),RS_表达式(εi), ..]→ T(x)=[RS_ref(RS_表达式(εi)), RS_循环([RS_ref(RS_表达式(εi))]),RS_表达式(εi), ..]b.) x 在循环体内被修改：T(x)=[RS_循环([RS_表达式(εj),RS_自身]),RS_表达式(εi), ..]→ T(x)=[RS_自身, RS_循环([RS_表达式(εj),RS_自身)]),RS_表达式(εi), ..]

II.5 基本块调度与数据通路并行化

基本块调度器探索控制流，并将数据通路语句（赋值）分组为尽可能大的语句块，即基本块。一个基本块只有一个控制流入口（第一条语句）和一个退出点（最后一条语句之后）。通过使用数据依赖图（DDG），可以分析如何利用基本块中语句的并行执行。基本块的 DDG 是一个有向无环图，其中节点表示操作（赋值），边表示操作之间的数据依赖关系，并据此确定执行顺序。

基本块的 DDG 确实可以由一组独立图组成，也可以包含具有多个根节点的图，如**图 2**所示。

独立操作可以在有界块中于一个周期内调度并执行。目标是在有界块中调度 k 个操作，其中 *k*={1,..,*c*}，且 *c* 可以是无限制（最大值），也可以是并行操作最大数量的上限。

数据依赖图被划分为多个层级，并以此进行调度。

属于同一层级的所有操作彼此独立，可以调度到一个有界块中，如**定义 2**所示。层级划分通过计算从根节点到每个更深层节点的最长路径来完成。每个节点都会被标记为从某个根节点到达该节点的最长路径长度，该长度即为其层级。

假定有界块中包含的一组指令可以原子性地并行执行，并且每次数据传输只执行一次。

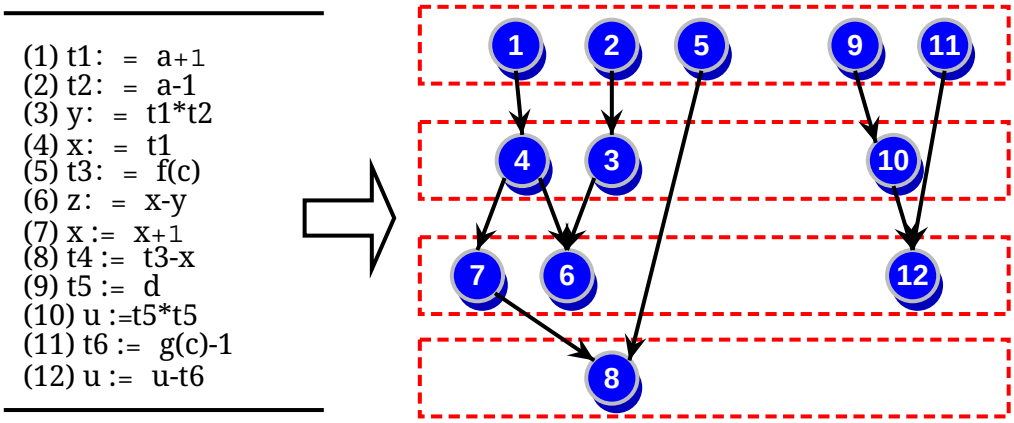


图 *2* 从赋值序列推导出数据依赖图森林，以及一种只需四条有界块语句（和周期）的可能并行调度。

这意味着，首先对有界块中所有赋值右侧的表达式求值（使用这些表达式中引用的所有存储对象的旧值）。

定义 *2 A.)* 将有序指令列表 *K={i1; ..}*（其中 *i* 属于语句集合 κ）划分为一组基本块 *B=[b1,b2,..]*，每个块均由一个有序指令列表 *(i;..)* 构成。*B.)* 用于并行化数据流独立指令子列表 *(i,..)* 的调度算法，再将其映射回扁平化的指令列表。函数 *PARTITION*：*K*=[*i1;i2;..*] → *B*=[*b1;b2;..*] *IS b*=[ ]; *B*=[ ]; *WHILE K* ≠ [ ] *DO k* ← *HEAD(K), K* ← *TAIL(K); IF KIND(k)* = ← ⊕ [ ] 数据 ← ⊕ ← [ ] *THEN b b k ELSE B B b, b ;* 函数 *SCHEDULE*：*B*=[*b1;b2;..*] → *K'*=[*(i1,i2,..);(..);..*] *IS K*=[ ]; ∀ *b* ∈ *B DO DDG* ← 从 *b* 构建数据依赖图森林；*roots* ← 数据依赖图中根节点的列表（没有前驱节点的节点）；∀ *{op* | *op* ∈ 根节点*} DO Mark(op,1); maxLP* ← 0; ∀ *{op* | *op* ∈ 数据依赖图|根节点*} DO LP* ← 0;

∀ {op' | op' ∈ 根节点} DO LP ← Max(LP, |路径(op,op')|);
Mark(op,LP); maxLP ← Max(maxLP,LP); ∀ {level | level ∈ {0..maxLP}} DO K ← K ⊕ [绑定({op ∈ 数据依赖图 | Level(op) = level})];

II.6 寄存器传输级综合与 VHDL 模型

寄存器传输级设计的综合通过将中间微代码指令列表表示转换为有限状态机和寄存器传输数据通路架构块来实现。微代码指令列表的控制通路和数据通路使用（线性）状态列表表示。状态列表由串接的状态表达式构成，这些表达式可视为控制流语句，其中包含特定状态的数据流表达式：

状态转换列表中的状态表达式由以下部分组成：

状态名称_

状态名称字符串格式：`S_i<指令标识符>_<指令名称>_<子操作码>`

下一状态_

不同的控制语句（寄存器传输级的控制路径）：

下一状态（标签）、**下一状态_指令**

控制流转向带标签状态 <标签>。下一条指令类型是列表中紧随其后的状态表达式的未解析指针。

分支（数据列表、下一状态**0**、下一状态**1**）

条件分支。数据包含对条件分支进行求值所需的布尔表达式。最多可能有两种状态转换。

选择（数据列表，分支值_状态 列表）

多分支非层次化分支。数据包含与各分支值进行比较的表达式。[分支_状态=（数据列表，下一状态_状态）]

状态数据_

指定寄存器传输级模块的数据通路，例如不同的数据传输语句，主要用于处理*VHDL*信号赋值：

数据_输入（**VHDL**_表达式）

表达式的输入参数，例如数据传输语句的右侧，例如算术逻辑单元的操作数。

数据_输出（***VHDL***_表达式）

表达式或运算单元的输出被传送至数据传输语句的左侧，例如算术逻辑单元的输出结果，或充当资源多路复用器的全局对象的输出（组合数据通路的一部分）。

数据_传输（***VHDL***_表达式）

某个 表达式、过渡局部寄存器数据传输（过渡数据通路的一部分）。

数据_信号(*VHDL*_表达式)

信号赋值（依赖状态），静态驱动。

数据_条件（*VHDL*_表达式）

条件表达式（表达式转换后的*VHDL* If/Case）。

数据_顶层(*vhdl*_表达式)，数据_顶层_定义 (*vhdl*_表达式)

实体顶层 VHDL 表达式和架构定义。

数据_定义(*vhdl*_表达式，*vhdl*_表达式)，数据_定义_转换 (*vhdl*_表达式，*vhdl*_表达式)

组合数据通路和过渡数据通路中的默认信号赋值。

VHDL 输出

最后，根据状态转换列表创建 *VHDL* 实体（每个顺序执行进程都有一个列表）。此外，还会为模块创建 *VHDL* 实体。全局对象在模块的顶层实体中实现，如**图 3**所示。

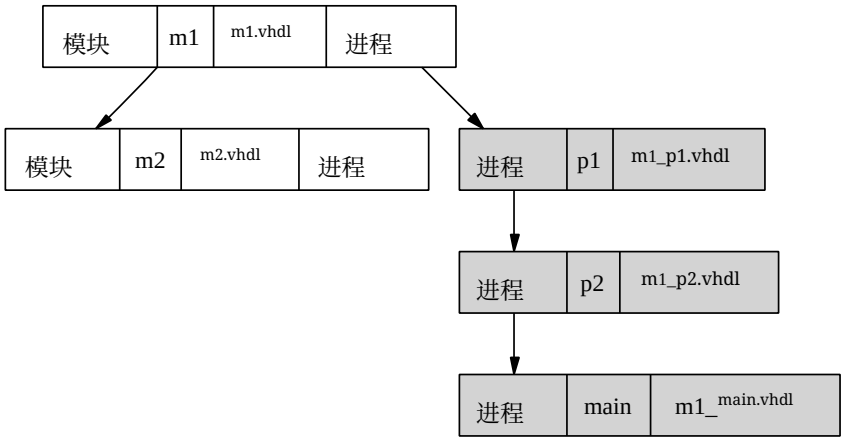


图 *3 ConPro* 模块层次结构和创建的 *VHDL* 组件。

以下示例展示了一个小型 *ConPro* 设计所综合生成的 *VHDL* 的概要。该设计包含三个进程和两个共享对象：一个寄存器数组，以及一个用于同步这三个进程的信号量。

例 *5* 一个小型 *ConPro* 设计及综合后的 *VHDL* 摘录

~~ex.cp~~

打开 Core；打开 System；打开进程；打开信号量；数组 data_输入，data_输出：寄存器[100] 的整数[16]；寄存器 mon：整数[16]；导出 mon；对象 sem：信号量，带有初始化=0；**函数** f(x：整数[16]) 返回(y：整数[16])：开始 y <- x*x; 结束;**进程** consumer1：开始 sem.down(); 对于 i = 0 到 50 执行 开始 data_out.[i] <- f(data_in.[i]); 结束; sem.up(); 结束;

进程 consumer2：开始 sem.down(); 对于 i = 51 到 99 执行 开始 data_out.[i] <- f(data_in.[i]); 结束; sem.up(); 结束;进程 主进程：开始 sem.init(0); 对于 i = 0 到 99 执行 开始 data_in.[i] <- i; 结束 ; consumer1.start(); consumer2.start(); sem.up(); 对于 i = 1 到 2 执行 开始 sem.down(); 结束; 对于 i = 0 到 99 执行 开始 mon <- data_out.[i]; 结束; 结束;

********** conpro ex.cp 至**********

ex.vhdl 实体 MOD_ex 是 端口(-- 与外部世界的连接 信号 mon_RD: 输出 std_逻辑_向量(15 downto 0); 信号 CLK: 输入 std_逻辑; 信号 RESET: 输入 std_逻辑); 架构 主进程 of MOD_ex is 进程实例 组件 ex_函数_f 端口(..) 组件 ex_1 消费者 端口(..) 组件 ex_2 消费者 端口(..) 组件 ex_主进程 端口(..) 类型 ARRAY_数据_输出_类型 is array(0 to 99) of 有符号数(15 downto 0); 信号 ARRAY_数据_输出: ARRAY_数据_输出_类型; .. 信号 PRO_消费者1_ENABLE: std_逻辑; 信号 PRO_消费者1_END: std_逻辑; 信号 PRO_消费者1_主进程_启动: std_逻辑; 信号 PRO_消费者1_主进程_GD: std_逻辑; .. 开始

ex_consumer1.vhdl 实体 ex_consumer1 is port(与外部对象、组件及外部世界的连接 signal 数组_data_out_WR: out signed(15 downto 0); signal 数组_data_out_WE: out std_logic; signal 数组_data_out_GD: in std_logic; signal 数组_data_out_SEL: out std_logic_vector(7 downto 0); signal 互斥锁_LOCK_函数_f_LOCK: out std_logic; signal 互斥锁_LOCK_函数_f_UNLOCK: out std_logic;

signal 互斥锁_LOCK_函数_f_GD: in std_logic; signal 信号量_sem_DOWN: out std_logic; signal 信号量_sem_UP: out std_logic; signal 信号量_sem_GD: in std_logic; signal REG_RET_函数_f_y_RD: in signed(15 downto 0); signal PRO_函数_f_CALL: out std_logic; signal PRO_函数_f_GD: in std_logic; signal 数组_data_in_RD: in signed(15 downto 0); signal 数组_data_in_SEL: out std_logic15_vector(7 downto 0); signal REG_ARG_函数_f_x_WR: out signed(downto 0); signal REG_ARG_函数_f_x_WE: out std_logic; signal PRO_consumer1_ENABLE: in std_logic; signal PRO_consumer1_END: out std_logic; signal conpro_System_clk: in std_logic; signal conpro_System_reset: in std_logic); end ex_consumer1; architecture 主进程 of ex consumer1 is _局部和临时数据对象 signal 循环_i_0: signed(7 downto 0); 状态处理 type pro_states is (S_1_0[0]consumer start, -- 进程 : S_1_22215[19]i fun, -- 函数 ex.cp: S_2__2877[20]i for loop, -- COUNTLOOP ex.cp: S_2__2877[20]i for loop cond, -- COUNT LOOP ex.cp: S_3_78627[22]i fun, -- 函数 ex.cp: S_4_23890[22]i assign, -- ASSIGN ex.cp: S_5_19442[22]i fun, -- 函数 ex.cp: S_6_58851[22]i assign, -- ASSIGN ex.cp: S_7_83891[22]i fun, -- 函数 ex.cp: S_2__2877[20]i for loop incr, -- COUNT LOOP ex.cp: S_8_63052[24]i fun, -- 函数 ex.cp: S_1_0[0]consumer end -- 进程 :); signal pro_state: pro_states := S_consumer1_start; signal pro_state_next: pro_states := S_consumer1_start; .. state_transition: process(..) 开始 if conpro_System_clk'event and conpro_System_clk='1' then if conpro_System_reset='1' or PRO_consumer1_ENABLE='0' then pro_state <= S_consumer1_start; else pro_state <= pro_state_next; end if; end if; end process 状态_transition; 进程实现 指令控制路径块 - Leitwerk 控制_路径：process(..) 开始 .. case pro_state is 当 S_1_0[0]consumer start => -- 进程 : pro_state_next <= S_i1_fun; 当 S_1_22215[19]i = 0 fun => -- 函数 ex.cp: if not((SEMA_sem_GD) ('')) then pro_state_next <= S_i1_fun; else

进程_状态_下一状态 <= S_i2_循环_循环；结束 if；当 S_i2_循环_循环 => -- 计数_循环2877[ex.cp:20]

进程_状态_下一状态 <= S_i2_循环_循环_条件；当 S_i2_循环_循环_条件 => -- 计数_循环2877[ex.cp:20]

如果 常量 I8 50 >= 循环i 0 则 <= 进程_状态_下一状态 S_i3_函数；否则进程_状态_下一状态 <= S_i8_函数；结束 if；当 S_3_78627[22]i 函数 => -- 函数 ex.cp: 如果非（（互斥锁_锁<= 函数_f_GD）(' ')）则进程_状态_下一状态 S_i3_函数；否则进程_状态_下一状态 <= S_i4_赋值；结束 if；当 S_4_23890[22] i <= 赋值 => -- 赋值 ex.cp: 进程_状态_下一状态 S_i5_函数；当 S_5_19442[22] i 函数 => -- 函数 ex.cp: 如果 PROFUNf GD ' ' 则 进程_状态_下一状态 <= S_i5_函数；否则进程_状态_下一状态 <= S_i6_赋值；结束 if；当 S_6_58851[22] i = 1 赋值 => -- 赋值 ex.cp: 如果 数组 数据 输出GD '' 则 <= 进程_状态_下一状态 S_i6_赋值；否则进程_状态_下一状态 <= S_i7_函数；结束 if；.. 当 S_1_0[0]消费者 <= 结束 => -- 进程：进程_状态_下一状态 S<= 消费者1_结束；PRO_消费者1_结束 '1'; 结束 进程 控制_路径；指令数据通路组合逻辑块 数据_通路：进程(..) 开始 -- 默认值 信号量_sem_DOWN <= '0'; 互斥锁_锁_函数_f_锁 <= '0'; 寄存器_ARG_函数_f_x_WR <= 转为有符号数(0,16); 寄存器_ARG_函数_f_x_WE <= '0'; 数组_数据_输入_SEL <= "00000000"; PRO_FUN_f_CALL <= '0'; 数组_数据_输出_WE <= '0'; 数组_数据_输出_SEL <= "00000000"; 数组_数据_输出_WR <= 转为有符号数(0,16); 互斥锁_锁_函数<= f_解锁 <= '0'; 信号量_sem_UP '0'; case 进程_状态 is 当 S_1_0[0]消费者 启动 => -- 进程：null ；当 S_1_22215[19]i 函数 => -- 函数 ex.cp: 信号量 _sem_DOWN <= 信号量 _sem_GD ；当 S_i2_ 循环_循环 => -- 计数_循环2877[ex.cp:20]

空；当 S_i2_循环_循环_条件 => -- COUNT_循环2877[ex.cp:20]

空；

当 S _ 3_ 78627[22]i 函数 => -- 函数 ex.cp：
互斥锁 _ 锁 _ 函数 _f=> 锁 互斥锁 _ 锁 _ 函数 _f <= GD ；当 S
_ 4_ 23890[22]i 赋值 -- 赋值 ex.cp： 寄
存器_参数_函数_f_x_写 数组_数据_输入 <= 读；寄存器_参数_函数_f_x_写使能 <= '1'；数
组_数据_输入=> 选择 <= (I_到_L(循环_i_0))；当 S
_ 5_ 19442[22]i 函数 -- 函数 ex.cp： 进程_函数_
f_调用 ''；当 S <= 1 _ 6_ 58851[22]i 赋值 => --
赋值 ex.cp： 数组 _ 数据 _ 输出 _ 写 寄存器 _ 返回值 _ 函数 _f_y <= 读；数组 _ 数据 _ 输
出_写使能 <= '1'；数组_数据_输出_选择 <= (I_到_L(循环_i_0))；.. 结束 进程 数据通路；
指令数据通路转换块 数据转换：进程(..) 开始 如果 conpro_系统_clk'事件 且 conpro_系
统_clk='1' 那么 如果 conpro_0 <= 系统_reset = '1' 那么 循环_i 到_有符号数(0,8)；否
则 case 进程状态 是 当 S
_ 1_ 0[0]消费者 启动 => -- 进程：
空；当 S _ 1_ 22215[19]i 函数 => -- 函数
ex.cp： 空；当 S
_ 2_ _ _ 2877[20]i for 循环 => --
_ _0 <= _ 8_0
COUNTLOOP ex.cp： 循环 i CONST_I ；.. 当 S
_ 2_ _ _ => i for 循环 增量 -- COUNT
_ 2877[20]循环 ex.cp： 循环i 循环 i CONST_I8_1;
_ _0 <= _ _0 +

例如 consumer2.vhdl **实体 ex_**
consumer2 是 ..

例如 FUN f.vhdl 实体 ex_FUN_f 是 端口(-- 与外部对象、组件及外部世界的连接 信号 寄存
器_RET_FUN_f_y_WR: 输出 有符号数(15 downto 0); 信号 寄存器_RET_FUN_f_y_WE: 输出 标准逻
辑; 信号 寄存器_ARG_FUN_f_x_RD: 输入 有符号数(15 downto 0); 信号 PRO_FUN_f_ENABLE: 输入
标准逻辑 ; 信号 PRO_FUN_f_END: 输出 标准逻辑 ; 信号 conpro_system_clk: 输入 标准逻辑 ;
信号 conpro_system_reset: 输入 标准 _ 逻辑); ..

例如 main.vhdl 实体
ex_主进程是 ..

III μRTL 编程语言

上一节介绍了使用微代码指令的中间表示。这些操作和声明直接表示寄存器传输级，最终可通过 VHDL 或 Verilog 模型综合到门级。并行编程语言 *Conpro* 将顺序进程映射到由以下部分组成的寄存器传输级进程块：

1. 有限状态机（FSM）实现控制路径；2. 寄存器传输实现数据通路。

多进程系统通过相互连接的多有限状态机数据通路结构来实现。RT 进程块的行为由本节单独介绍的 μ寄存器传输级语言指定（基于 **[BAR73]**）。该语言可视为一种独立的核心语言，用于建模和理解扩展的并发 CSP 编程模型。基本 μRTL 操作的语法总结见**表 3**。与先前介绍的 μCODE 汇编器语言不同，后者仅限于一种进程机，μRTL 也可用于描述多进程系统。

表 *3* 可立即综合为硬件寄存器传输级架构的最简符号并行编程语言（ε：计算表达式）。

操作	**描述**
r ← 表达式	将值赋给寄存器。
*i*1； *i*2； *i*3； ...	语句的顺序执行。
*i*1，*i*2，*i*3，...；	语句的并行执行（仅限非阻塞状态，即赋值）
分支*s*	无条件跳转到与该状态相关的另一条指令，位于此语句之前或之后*s*。
分支*cond*:*s*	满足条件时进行条件分支，否则，下一状态将执行指令。
s:	将状态分配给指令的指令标签。
o→操作(参数1,参数2,...)	对全局共享对象执行操作的过程式应用带可选参数。
r ← ε(*o*→*op*(参数1,参数2,...))	对全局共享对象执行操作的函数式应用带可选参数并返回一个值。

进程 *p* → {
第1；第2；第3；...
}
	命名进程组合的定义。进程 *p* 也被视为一个对象，支持启动和停止操作。

通道 o *o*→接收 |发送 join(p_1, p_2, ..)	一些同步原语。连接操作可用于在时间上同步不同进程的控制路径（屏障）。

从 *OCCAM* 和 Conpro 编程语言推导出 μ寄存器传输级规范的示例见 **例 6**。

例 *6*：由以下内容推导出的 μ寄存器传输级规格 *OCCAM* 和 *ConPro* 程序

1 OCCAM 2*3* [100] 整数 数据：4 整数 均值：5 顺序执行 6 均值：= 07 顺序执行 索引 = 0 循环

8 = + 100 均值：

均值 数据

9 = [ ] 100

索引 均值：均值 / 10 11 ConPro 12 13 数组 数据：整数寄存器[16]; 14 进程 主进程：15 开始 16 寄存器 均值：整数[20]; 17 均值 <- 0; 18 对于 i = 1 到 100 执行 19 开始 20 均值 <- 均值 + 数据.[i]; 21 结束; 22 均值 <- 均值 / 100; 23 结束; 24*25* μ寄存器传输级 26 27 进程 主进程 → { 28 s1：索引 ← 0，均值 ← 0; 29 s2：分支(索引=100)：s6; 30 s3：均值 ← 均值 + 数据[索引]; 31 s4：索引 ← 索引 + 1; 32 s5：分支 s2; 33 s6：均值 ← 均值 / 100; 34 }

μ寄存器传输级指令与进程代数的关系，以及μ寄存器传输级 指令的进程流行为如下所示（♦：同步和调度点）。

进程流行为	
r ← ε	á {*r* ← ε} á ♦
*i*1; *i*2; ..	á {*i*1} á ♦ á {*i*2} á ..
*i*1, *i*2, ..	á {*i*1} ‖ {*i*2}.. á ♦
o→*op*()	á {*o*→*op*()} á ◊¬保护锁(*op*) á
分支 *cond*:*s*	á (¬条件 á {*i*n+1} | 条件 á {*i*s}) á ♦
join (p1,p2, ..)	á (p_1 ‖ p_2 ‖ .. ‖ pn) á♦á(p1* ‖ p2 *‖ .. ‖ pn*)

IV ConPro 编程语言

并发性在控制路径级别上被明确建模，各进程按顺序执行一组指令，起初彼此独立。程序设计层面支持并行组合，但在运行时是静态的，以便综合具有静态资源的片上系统设计。这意味着进程数量在设计和编译时便已固定。

进程间通信（IPC）基于 CSP 模型，通过不同的对象（互斥锁、信号量、事件、定时器）提供进程同步，并使用队列或通道在进程之间交换数据。通过在数据通路层级使用有界块，可实现更细粒度的并发性；这些块可在一个时间单位内执行多条指令（仅限数据通路上的操作，例如数据赋值）。

块级并行可由基本块调度器显式或隐式启用。从编程层访问在硬件层级（VHDL）建模的硬件块有两种方法：

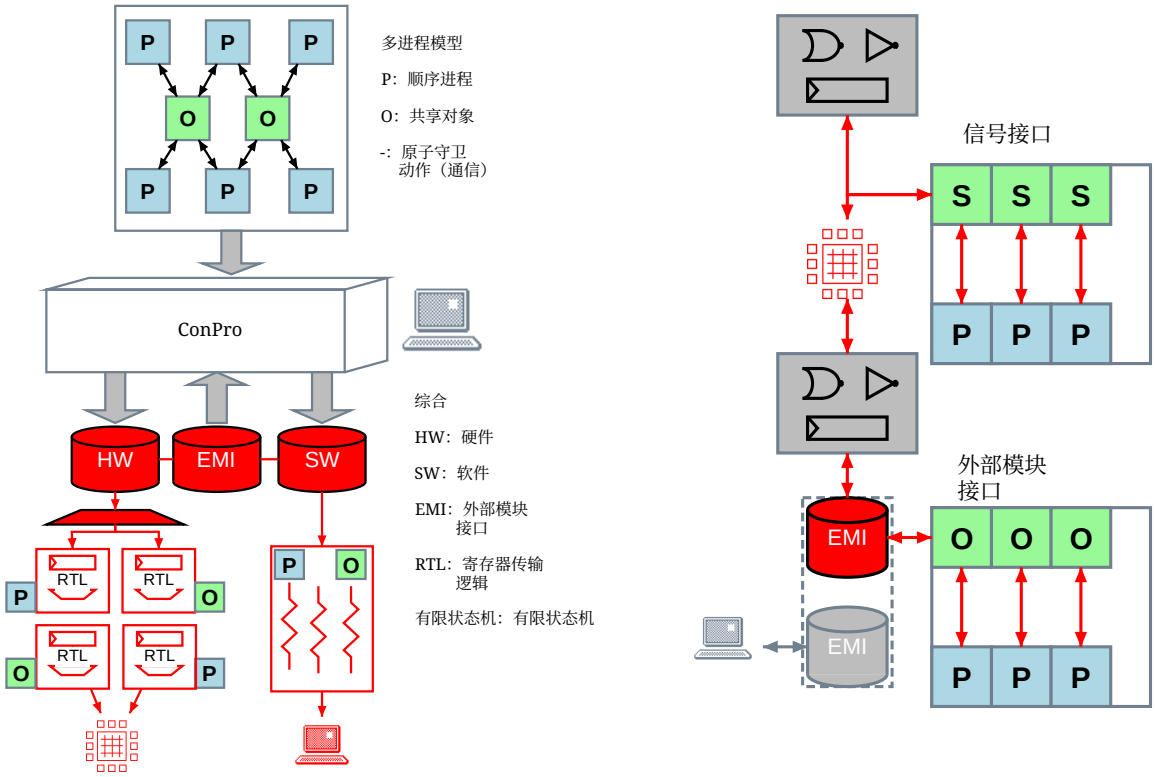
1. 采用面向对象编程方法，使用方法 2. 使用信号组件结构接口和信号

第一种方法将所有硬件块（包括 IPC）都视为抽象数据类型对象（ADTO），并为其定义一组方法，这些方法可在进程级别（运行时）和顶层访问（仅适用于配置方法，例如设置定时器的时间间隔）。硬件与 ConPro 编程模型之间的桥梁由外部模块接口（EMI）提供。

信号组件结构可用于实例化和访问外部硬件块，并构建 ConPro 设计的顶层硬件端口接口。信号是没有存储模型的互连元件，只为外部硬件块提供接口。信号也用于组件结构中。

ConPro 编程模型是硬件综合和软件综合共用的源代码，可复用同一程序源代码来实现不同的实现方案，如图 *4* 左侧所示。EMI 编程范式是这种多目标实现综合的核心组成部分，它通过封装和抽象硬件块（例如 IPC 和通信对象）发挥作用。大多数 EMI 对象都可以同时指定硬件行为（VHDL）及其操作等效且互补的软件模型，而不影响编程层，如图 *4* 右侧所示。

图 **4** 总结了 ConPro 编程语言的块结构，以及如何使用这些块来构成顶层模块设计。



*Fi*图 *4* 左：*ConPro* 编程模型作为硬件综合与软件综合的共同源头。右：来自编程层的硬件接口，（*a*）信号接口，（*b*）*EMI*

IV.1 Hello World

为了先有个初步印象，我们从 Hello World 演示开始。完整的 ConPro 代码见**例 3**。前五行打开了 Core、System 和 EMI 模块。Core 和进程模块始终是必需的。定时器模块提供定时器同步对象，而 Uart 模块提供可通过一组操作访问的串行链路通信设备。若要在表达式中使用频率和时间单位值，必须定义时钟频率（第 9 行），该频率通过第 8 行创建的系统对象来设置。第 15 至 22 行定义并导出了输入输出端口。顶层模块映射到一个组件对象（第 21 行）；基本上，这是通过第 15 至 19 行单独定义的一种结构类型来定义输入输出端口。

主进程（必须始终定义）实现了一个时间同步旋转 LED 闪光灯（第 69–81 行）。与微控制器软件类似，主进程通过实现一个无限循环来驱动 LED 闪光灯。调用定时器等待操作（第 77 行）会阻塞进程控制路径。定时器间隔在第 37 行设置（于编译时）。LED 信号（电路的输出）直接连接到寄存器（在第 28 行完成映射），并在第 79 行通过位运算进行修改。

hello world 电路的第二部分是一个完整的串行接收器和发送器，通过一条包含接收线和发送线的通用 UART 链路连接。这里有一个专用的接收进程，定义在第 40 至 66 行。该进程使用条件循环，通过调用 UART 对象的读取操作（第 51 行）等待接收一个字符。此操作会阻塞该进程的控制路径，直至接收到字符或发生错误。接收到字符并将其存储在本地进程寄存器 *d* 后，将发送字符串 `hello`。

整个设计编译后生成 1311 行 VHDL 代码（分为四个源文件：每个进程各一个、用于将所有部分连接在一起的顶层组件，以及一个 ConPro 库文件）。

例 7：带有旋转 *LED* 图案和 *UART* 文本传输的 *Hello world* 电路。

```
1 打开 Core；2 打开进程；3 打开定时器；4 打开
System；5 打开 UART；6 7 -- 硬件系统参数 8 对
象 sys：System；9 sys.clock（18500 千赫兹）；
10 sys.target（"xc3s1000-ft256-5"）；11
sys.target（"xc18v04"）；12 sys.reset_级别
（0）；13 14 -- IO 端口类型 15 类型 dev_type：
{ 16 端口 leds：输出逻辑[4]；17 端口 rx：输入
逻辑；18 端口 tx：输出逻辑；19 }；20 -- FPGA
的 IO 端口 21 组件 DEV：dev_type；22 导出
DEV；23 24 -- LED 25 寄存器 stat_leds、stat_
ev：逻辑[4]；26 寄存器 diag：逻辑[3]；27 -- 将
寄存器映射到输出端口 28 DEV.leds <<  stat_l
eds；29 30 -- UART 31 对象 myuart：UART；
32 myuart.interface（DEV.rx、DEV.tx）；33
myuart.baud（115200）；34 35 -- LED 切换定
时器 36 对象 ledtimer：定时器；37
ledtimer.time（500 毫秒）；38 39 -- UART 处理
程序 40 进程 receiver：41 开始 42 寄存器 d：逻
辑[8]；43 寄存器 err：布尔型；44 myuart.init
（）；45 myuart.start（）；46 47 err <- false；
48 当 err =  false 时循环执行 49 开始
```

50 -- 等待字符 51 myuart.read（d、err）；52 -- 发送“hello” 53 对于 i=1 到 5 执行 54 开始 55 将 i 与以下情况匹配 56 开始 57 当 1：d <- 'H'；58 当 2：d <- 'e'；59 当 3：d <- 'l'；60 当 4：d <- 'l'；61 当 5：d <- 'o'；62 结束；63 myuart.write（d、err）；64 结束；65 结束；66 结束；67 68 -- 主进程 69 进程 main：70 开始 71 ledtimer.init（）；72 ledtimer.start（）；73 receiver.start（）；74 stat_leds <- 0b0001；75 始终执行 76 开始 77 ledtimer.await（）；78 -- LED 位旋转 79 stat_leds <- stat_leds[1]@stat_leds[2]@stat_leds[3]@stat_leds[0]；80 结束；81 结束；

IV.2 进程组合

进程是 ConPro 设计模块的主要执行单元。进程按顺序执行一组指令。进程通过共享对象（进程间通信，IPC）相互通信并与环境进行通信。在控制路径和数据通路层面上，进程既可以顺序组合，也可以通过有界块并行组合，如**图 5**所示。

控制路径层面的并行进程组合采用扁平结构，仅提供粗粒度并发。

进程内部数据通路层面的并行进程组合是细粒度的，但仅限于不影响进程控制路径的非阻塞语句。ConPro 编程语言仅支持在顶层定义命名进程。相比之下，例如 OCCAM 编程模型允许在任意位置嵌入和嵌套并行及顺序进程构造器。

进程定义由一个定义唯一进程名称的头部，以及一个定义局部对象定义（类型、数据和一些抽象对象）与语句序列的进程体组成。

进程也是抽象对象。因此，有一组可应用于进程对象（标识符）的方法 **{启动、停止、调用}**。它们用于控制进程的状态。

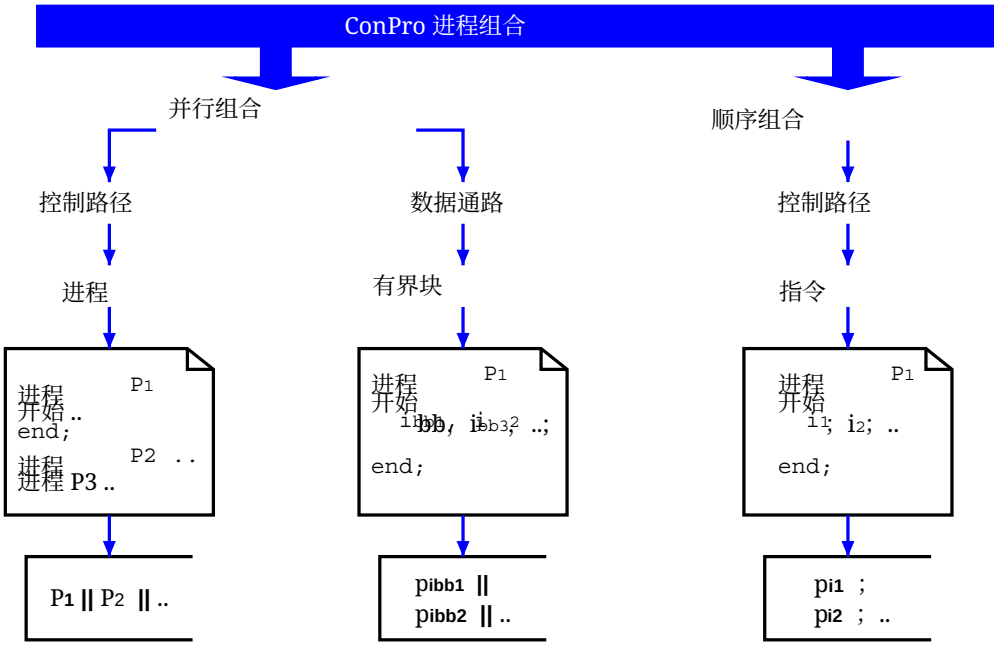


图 *5 ConPro* 中不同层次的并发性与进程组合

进程必须由另一个进程显式启动，即使用 **启动** 方法，该方法会并发地启动进程（异步调用）。使用 **调用** 方法可以同步启动进程，该方法会挂起调用进程，直到被调用进程终止。每个进程都可以使用 **停止** 方法异步停止。只有一个主进程，它会在启动时自动启动。为了简化进程集合的定义，可以定义参数化的进程数组，这些数组共享同一个进程体定义，并将其复制到不同且相互独立的物理进程中（参见**定义 3**）。

定义 *3* 使用对象方法应用进行 *ConPro* 进程定义与控制。进程是可以进行参数化的对象（即综合参数）。

```
打开 进程;
进程 进程名:
开始
   定义
   语句
结束 [ 带有 参数=值 ];

数组 并行进程名 : 进程[N]
开始
   定义
   语句
结束 [ 带有 参数=值 ];

...
      进程名.启动();
进程名.停止();
进程名.调用();
      并行进程名.[索引].启动();
...
```

共享的程序函数与过程也通过顺序进程实现。调用进程通过对函数进程对象应用调用方法来启动函数进程；该操作会使调用进程暂停，直到函数进程终止。为串行化并发的函数调用，会在函数进程上附加一个额外的函数调用包装器（互斥调度器）。

进程可以包含任意类型的顺序语句（例如，数据赋值、循环、分支）和并行数据通路赋值语句（有界块），并且可以选择将一组参数应用于进程块或进程内的任何块：

```
1   打开 进程；  2寄存器 a,b,d：逻辑
[8]；  3进程 主4    开始 5    循环 i
= 1 到 5 执行6    开始 7    d <- d + i;
8    结束 并 展开；9    开始 10 a <-
d；11   b <- d；12   结束 并 绑定；
13  结束 并 调度=基本块；
```

IV.3 数据存储对象

支持真正的位可缩放数据类型（类型 β）和存储对象（类型 α 的子集 R）。数据宽度可在 δ=1-64 位的范围内选择。数据存储对象 R 由类型的笛卡尔积（α×β）指定和定义。存储对象可以在表达式中读取，并通过赋值进行修改。

寄存器是单个存储单元，可用作共享全局对象，也可用作进程内部的局部对象。作为全局对象时，寄存器支持并发读取访问（无需互斥锁保护的调度器）和互斥锁保护的独占写入访问。如果有多个进程尝试写入该寄存器，互斥锁保护的调度器会将写入访问串行化。可用的调度器有两种，分别采用静态优先级和动态先进先出调度策略。

变量是内存块中的存储单元，可用作共享全局对象（即内存块本身），也可用作进程内部的局部对象。变量始终通过互斥锁保护的读写访问，对其所属的内存块进行独占访问。

不同数据类型和数据宽度的变量可以存储在一个或多个内存块中，并映射到通用 RAM 块。地址管理在综合过程中自动完成，对程序员是透明的。不支持直接地址引用或操作（即指针）。

内存的数据宽度始终采用物理逻辑/位向量类型，并根据内存中存储的最大变量进行缩放。为降低内存数据宽度，可以对变量进行分片，也就是说，一个变量会分散存储在多个存储单元中。

不同的存　　储块可以显式创建，并且可以将变量赋值给不同的块　　.

队列也是存储单元，具有先进先出数据传输顺序，同时也是同步进程间通信对象。它们始终用作共享全局对象。队列和通道可以直接在表达式中读取，并且可以像其他存储对象一样用于赋值的左侧。

通道与队列类似。但它们可以是带缓冲的（行为类似于只有一个单元的队列，深度为 1），也可以是无缓冲的（仅提供握手数据传输）。

在**定义 4**中总结了有关信号、数据存储和对象定义语句的内容。

定义 *4* *ConPro* 数据存储、信号对象定义及可选参数化的数据类型（*DT*：数据类型，位宽：包含符号位的位宽，⟦..⟧：可选）

```
寄存器 名称 : DT ⟦ 带有参数=值 ⟧;
变量 名称 : DT ⟦ 在块名称 ⟧;
块 名称;
信号 名称 : DT;
队列 名称 : DT ⟦ 深度=值 ⟧;
通道 名称 : DT ⟦ 深度=值 ⟧;
类型 DT = { 逻辑型, 逻辑型[位宽], 整数[位宽], 布尔型, 字符 }
类型 OT = { 寄存器, 变量, 信号, 队列, 通道 }
```

寄存器、变量、队列和通道可定义为积类型（数组和结构）以及和类型（枚举类型）。

IV.4 信号和组件

信号是没有存储模型的互连元件。它们为外部硬件块提供接口，也用于组件结构中。

信号可以像其他存储对象一样直接用于表达式中。信号可以在表达式中读取，也可以使用赋值语句为其赋值。读取信号会返回信号的实际值，但向信号写入时，仅在赋值语句执行期间（处于活动状态）为其赋予新值；否则，将为信号赋予默认值。因此，一个信号只能有一个赋值。

信号是非共享对象，没有访问调度器。只有一个进程可以为信号赋值（通常使用**等待**语句），但多个进程可以并发读取信号。此外，还可以使用**映射**语句将信号映射到寄存器输出。信号对象的定义见**定义 4**。

IV.5 数组和结构类型

数组可用于数据存储和抽象对象类型的定义，如**定义 5**所示。可以使用静态选择器（索引在编译时已知）和动态选择器（变量表达式）选择数组，所使用的点括号表示法为**.[索引]**。

定义 *5* *ConPro* 数组定义和数组元素访问（大小：数组元素的数量）

```
数组 AS: OT[大小] ，属于 DT ⟦ ，参数参数=值 ⟧;
数组 AO: 对象 obj[大小] ⟦ ，带有 参数=值 ⟧;
数组 AO: 进程[大小] ，从开始到结束
          ⟦ 带有 参数=值 ⟧;
... AS。[索引] ...
... AO。[索引]。方法 ...
```

结构类型定义仅包含数据类型（DT），不包含对象类型（OT）。针对不同用途，结构有三个不同的子类，具体定义见**定义 6**:

多类型结构

通用结构类型将具有不同数据类型的不同命名结构元素绑定到一种新的用户定义数据类型，即原生乘积类型。

位结构

此结构子类为存储对象提供位索引名称映射。所有结构元素都具有相同的数据类型。位索引可以是一个位值，也可以是位范围。此结构类型为向量数据类型的各个部分（例如逻辑向量或整数类型）提供符号化/命名选择，并指定对象数据的位访问。

组件结构

此结构定义硬件组件端口，可以是 ConPro 模块顶层端口，也可以是嵌入式硬件组件（在硬件层级建模）的端口。此结构类型只能与组件对象定义一起使用。组件类型的行为与信号类型相同。

如果结构实例化的对象类型是寄存器，则只会创建 N 个相互独立的寄存器。如果是变量对象类型，则会将 N 个对象存储在 RAM 块中。也可以创建结构类型的数组。对于每个结构元素，都会创建一个不同的数组。硬件组件端口类型同样使用结构定义，不同之处在于，必须为每个结构元素指定信号方向。信号方向需要特别注意：如果组件处于较低的层级顺序（即嵌入式外部硬件组件），则该方向是从硬件组件的外部视图来看定义的。

如果该组件属于 *ConPro* 模块的顶层端口接口，则必须从内部视图中看到它。

定义 6 *ConPro* 结构类型定义与元素选择

```
使用数据类型DT规范定义一种新建结构类型
对于每个元素:
      类型 ST: {
        E1: DT ;
        E2: DT ;
        ...
      };
      寄存器 实例: ST;

定义一种具有数据类型DT的新建组件结构类型
并为每个元素指定信号方向DIR。
      类型 CT: {
        端口 E1: DIR DT;
        端口 E2: DIR DT;
        ...
      };

定义一种新建位结构类型，并为每个元素指定位宽规范。
对于每个元素。
      类型 BT: {
        E1: 位宽;
        E2: 位宽;
        ...
      };
类型对象元素的访问
      ... 对象池.elem ...
```

以下 **示例 8** 展示了进程、对象和数据数组的用法：

例 8 ConPro 中进程、对象和数据数组的用法。1 open Core; 2 open Process; 3 open Barrier; 4 open Queue; 5 open System; 6 7 对象 b: 屏障; 8 数组 d: 寄存器[4] of 整数[8]; 9 数组 dataqueue: 队列[4] of 整数[8] with 深度=8; 10导出 d; 11 12 数组 p: 进程[4] of 13 开始 14 寄存器 t : 整数[8]; 15 for i = 1 to 5 do 16 开始 17 b.await (); 18 t <- # + 1; 19 d.[#] <- t; 20 dataqueue.[#] <- t; 21 d.[#] <- 0; 22 end; 23 end; 24 25 进程 主进程: 26 开始 27 b.init (); 28 for i = 0 to 3 do 29 p.[i].启动 (); 30 end;

IV.6 异常与处理

与现代编程语言一样，*ConPro* 编程模型支持使用信号表示异常，并通过语句进行异常处理，如**定义 7**所示。

异常是离开控制结构的唯一方式，例如循环、条件分支或函数本身。异常是抽象信号，可以在任何位置引发，并在 `try-with` 异常处理程序环境中捕获；捕获位置可以在引发异常的进程／函数内部，也可以在其外部。因此，如果异常未在引发它的进程或函数中被捕获，就会通过异常状态寄存器沿进程和函数的调用路径自动传播，并由 RT 逻辑直接实现。

定义 *7 ConPro* 对异常和异常处理程序的定义

异常的定义

```
异常 E;
```

引发异常（在进程内部）

```
... 引发 E; ...
```

已引发异常的捕获

```
尝试 I 与
开始
  当 E1: I1   ;
  当 E2: I2   ;
  ...
end;
```

异常传播的一个典型用法如**例 9**所示。

例 9 引发 ConPro 异常及异常处理。1 open Process; open Core; open Random; 2 3 exception DIVBYZERO; 4 object rnd: random with datawidth=8; 5 rnd.seed (1000); 6 7 function foo(a:int[8],b:int[8]) return (c:int[8]): 8 begin 9 if b = 0 then raise DIVBYZERO; 10 c <- a/b; 11 end; 12 13 process main: 14 begin 15 reg d : int[8]; 16 reg r : logic[8]; 17 try 18 begin 19 for i = 1 to 64 do 20 begin 21 rnd.read (r); 22 d <- foo(d,to_ int(r)); 23 end; 24 end 25 with 26 begin 27 when DIVBYZERO: d <- 0; 28 end; 29 end;

IV.7 函数与过程

函数定义由唯一的函数名称标识符、函数参数接口以及包含语句的函数体组成。函数体由局部对象定义（类型、数据和一些抽象对象）及指令序列组成。函数定义和函数调用的基本语法如**定义 8**所示。

每个函数参数以及返回值参数集合都按寄存器处理。这里只支持传值语义。函数调用前，会将函数实参的值复制到相应参数中；函数调用完成后，再复制返回值。在函数体内，所有参数和（命名的）返回参数都可以像其他寄存器一样用于赋值和表达式中。没有返回语句。最后一次赋给返回参数的值会自动返回。

函数有两种不同类型：内联函数和共享函数。内联函数类型按宏定义处理。每次应用（调用）函数时，函数调用都会被函数体替换，所有函数参数都会被函数实参替换（包括返回值参数）。共享函数通过顺序进程和调用方法实现，并使用额外的函数调用包装器。每次调用共享函数时，实参值都会传递给函数参数（全局寄存器），而返回值（如果有）则会传回。

定义 *8 ConPro*函数与过程的定义及应用

```
函数 fname (p1:DT,p2:DT,..) ⟦ 返回 (r1:DT,..) ⟧:
开始
  定义
  指令
end;

进程名();                    过程调用
进程名(i,1);
y ← fname1(x);              返回一个值的函数调用
{y1,y2} ← fname2(x);        返回两个值的函数调用
                            values
```

以下简短示例展示了函数的易用性：

1 **打开** 进程；*2***打开** Core；*34* **函数** **foo**(a:整数[8],b:整数[8]) **返回** (c: 整数 [8])：*5* 开始 *6* c <- a+b；*7* 结束；*8* **进程** 主进程：*9* 开始 *10* **寄存器** x,y,z:整数[8]；*11* x <- 5, y <- 6；*12* z <- **foo**(x,y)；*13* 结束；

IV.8 模块

一个 *ConPro* 设计层次结构由行为模块层（Module-B）组成，其中包含全局（共享）对象和进程。模块被映射为带有顶层硬件端口

接口的电路组件。结构模块（Module-S）可以由行为模块组成，并可选择性地包含内部互连组件。每个进程（以及进程级别）由局部（非共享）对象和进程指令序列组成，用于指定控制流和数据流。抽象对象类型通过抽象对象模块（Module-O）实现。

行为模块实现对象和进程。行为模块由源代码文件本身定义。实际上，模块层级只有一级，即主模块。可以使用 include 语句包含更多源代码文件。行为模块分为两类：嵌入对象和进程的模块，以及提供抽象数据类型对象（ADTO）的访问和实现的模块。这些对象主要是进程间通信和同步对象，例如互斥锁、信号量、定时器和一些通信链路。每个要使用的 ADTO 模块都必须使用 open 语句打开。

定义 9 *ConPro* 使用硬件组件和输入输出端口定义结构模块及其结构组合。

设计顶层输入输出端口定义：

```
类型 顶层_输入输出_类型:{
  端口 S1：DIR DT;
};
组件 TOP：top_io_类型;
导出 TOP;
```

定义一个具有指定名称的新结构模块：

```
模块 MS

开始
  导入
  组件
  结构类型
  映射
end;
```

行为模块的导入与组件的实例化（电路）：

```
导入 MB;
组件 C1，C2,..:MB;
```

定义并实例化互连组件的端口接口组件及其初始信号映射：

```
类型 ICT:{ 端口...};
组件 IC:ICT :=
{
  C1.TOP.S1  ,
  C1。TOP。S2  ,...
  C语言2。TOP。  ,...
};S1
```

使用映射语句进行内部互连：

```
IC。S1 >> IC.S2;
```

结构模块用于通过行为模块构建片上系统（SoC）电路。定义 9**定义 9**展示了如何使用 IO 端口结构、组件实例化和模块块来定义结构组合。结构组合通过重复使用多进程块，帮助设计复杂的 SoC 电路。

除了使用非缓冲信号的静态端口组件（方向固定）外，Ioport 模块还提供带缓冲寄存器的动态端口（如微控制器中的端口），支持部分位更新，如以下示例所示**例 10**：

例 *10* 电路硬件端口与可编程 *IO* 端口（*Ioport* 模块）

```
1 open Core; 2 open Process; 3 open Ioport; 4 5 t
ype top_port: { 6 port p_io1: 双向输入输出 logic[8];
7 port p_io2: 双向输入输出 logic[8]; 8 }; 9
component top: top_port; 10 导出 top; 11 12 对象
pt1,pt2: ioport with 数据宽度=8; 13
pt1.interface(top.p_io1); 14 pt2.interface(top.p_io2);
15 16 进程 p1: 17 开始 18 寄存器 d1,d2: logic[8]; 19
20 21 pt1.dir(0b00000000); 22 pt2.dir(0b11111111);
23 pt1.read(d1); 24 pt2.read(d2); 25 26 for i = 1 to 8
do 27 begin 28 d1 <- d1 land 0b00001111; 29 d1[4
to 7] <- to_logic(i); 30 d2[3] <- 1; 31 pt1.write(d1);
32 pt1.dir(0b11111000),pt2.write(d2); 33 d2[3] <- 0;
34 pt1.dir(0b00000000),pt2.write(d2); 35 pt1.read
(d1); 36 while d1[0] = 0 do 37 pt1.read (d1); 38
end; 39 end;
```

IV.9 ConPro 构建块

整体 ConPro 设计层次结构及组合构建块如**图 6**所示。顶层模块由进程定义、全局对象（存储、同步、通信）和用户定义类型组成，最后还包括连接进程和对象与外部世界的顶层接口端口。使用组件，结构组合可以在顶层嵌入硬件块。进程由指令和本地存储对象组成。

总而言之，ConPro 构建块的本体按降序排列如下：

1. 顶层模块和组件及其结构接口规范；2. 存储、对象、组件、进程、函数和模块接口（外部硬件组件）块；3. 外部模块接口（EMI）；4. 函数和进程中的顺序指令、有界块及本地存储；5. 类型和异常定义。

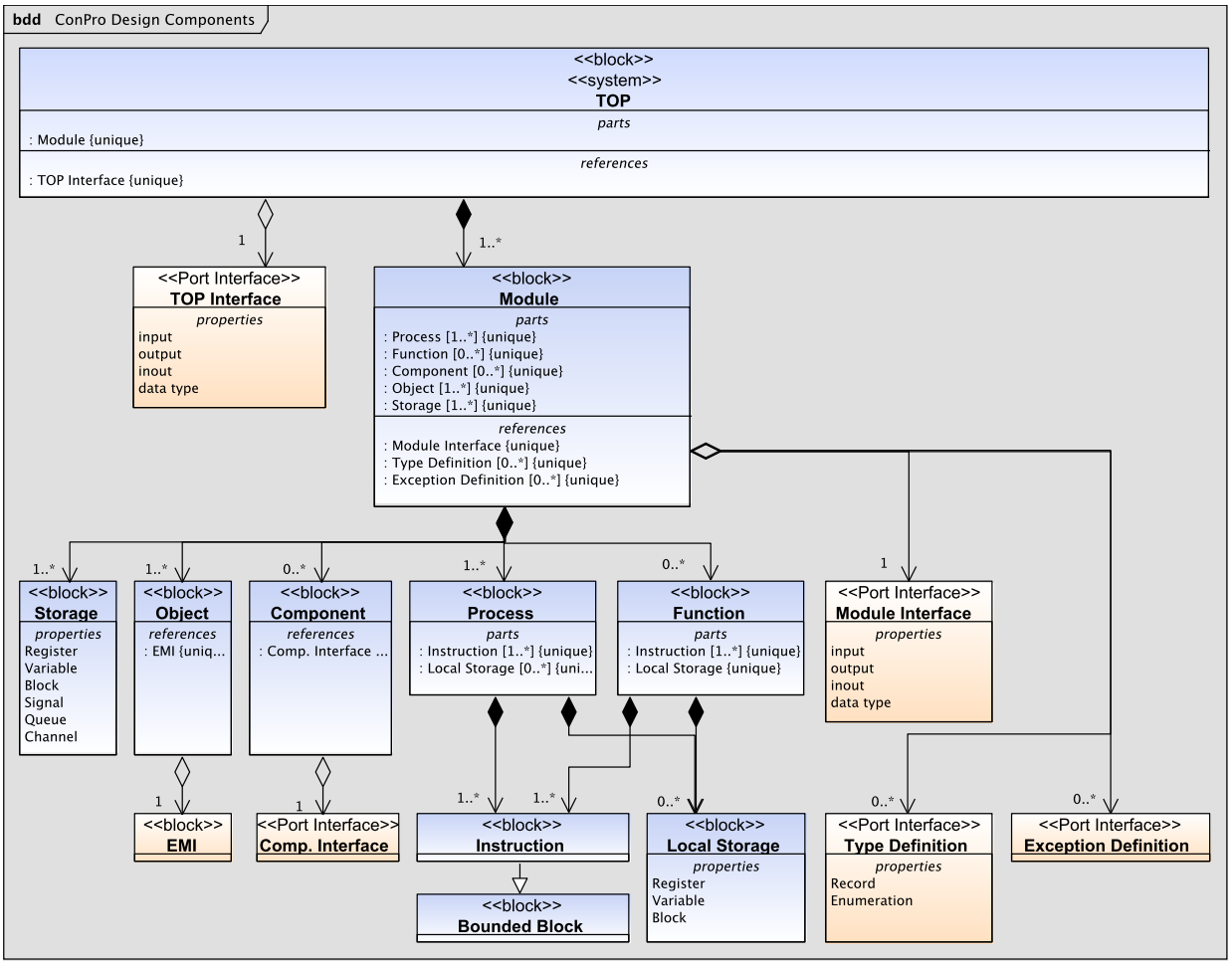


图 *6 ConPro*设计层次结构与组合构建块图

IV.10 控制与数据处理语句

数据处理

数据处理通过对表达式求值，并使用赋值将计算结果存储在寄存器或变量中来完成。赋值语句包含一个目标数据对象（寄存器、变量或信号），即左侧（LHS），以及一个表达式，即右侧（RHS）。该表达式可以任意嵌套，由算术、关系和布尔运算组成，其操作数访问寄存器、变量或信号，如**定义 10**所示。

与传统命令式编程模型一样，这里存在严格的顺序关系：先对右侧求值，然后存储结果。这通常由数据通路中适当的硬件模型来保证，并可在一个控制状态中完成处理。赋值只能用于进程体块中。

定义 *10* *ConPro* 计算语句（赋值和表达式）

赋值

变量 *x*: *DT*; ‖ 寄存器 *x*: *DT*; ‖ 信号 *x*: *DT*;

x <- expression;

微顺序时序模型：结果=求值(表达式)；写入(*x*, 结果)

表达式

表达式 ::= 值 | 表达式 运算符 表达式 .
运算符 ::= + - * / % land lor lnot band bor bnot 逻辑左移 lsr .

有界块

指令块可用于绑定指令，这些指令可以是条件分支等控制语句的一部分，也可以用于定义采用顺序或并行执行模型的子进程。指令可以进行参数化，并附加特定的处理或综合设置，如**定义 11**所示。有界块的简写形式是以逗号分隔的赋值指令语句列表。所有赋值都在一个有限状态机状态中进行调度，并在一个时间步内执行，即将存储在 RAM 块中的变量（通常需要两个时钟周期访问）替换为临时寄存器。指令之间不应存在数据相互依赖关系，否则有界块的执行结果可能会违反数据一致性。

定义 *11* 使用块的 *ConPro* 顺序与并行进程构造器。

```
开始   顺序执行. 子进程 ⟦ 开始 ⟧ 并行 子进程.
  i1; i1、i2、i3、..
  i2; ⟦ 结束 ⟧ ⟦ 并绑定 ⟧;
  ..
结束并绑定 ⟦ 参数=值 和 .. ⟧;

参数: { 绑定, 展开, 调度, 内联, ..}
```

条件分支

存在两种不同的条件分支：一种是基于布尔条件的互斥分支（if-then-else），另一种是将值与表达式求值结果进行匹配的多情况分支（match-with-when）。互斥分支可以链接起来（if-then-else-if），从而提供具有优先级和顺序的条件分支，如**定义 12**所示。

定义 12 *ConPro* 分支

```
如果 cond 则 icond=真 ⟦ 否则 icond=假 ⟧;

匹配 表达式 与
开始
    当 v1: i表达式=V1; 当 v2,v3, ..: i表达
式=v2∨表达式=v3..; ..
⟦ 当 其他: i其他; ⟧
end;
```

相比之下，多情况分支在控制和数据通路中可以并行求值匹配条件，从而使每个匹配情况的分支处理时间恒定。

循环

编程模型提供以下循环，其语法定义如下。除了常见的循环语句外，还提供了特殊循环，用于满足并行处理系统中基于事件的客户端-服务器/请求-服务操作语义以及硬件逻辑信号处理的要求。

计数循环 [for-do]

计数循环会通过递增或递减循环变量，重复执行循环体中的指令。循环迭代次数可由静态或动态限制值（寄存器、变量）指定。循环可以完全展开。在顶层，for 循环可用于生成展开后的顶层指令序列，例如使用数组映射指令。顶层 for 循环总是会展开，因此不需要展开参数

条件循环 [while-do]

只要布尔表达式求值结果为 true，就会执行循环体 .

无条件循环 [always-do]

循环体会在没有任何条件的情况下重复执行。用于基于事件的服务进程。无条件循环只能通过停止进程或引发异常来终止。

固定延迟 [wait-for-time]

进程可以恰好暂停（延迟）指定数量的时钟周期或时间单位。

条件等待 [wait-for-condition]

此语句会阻塞进程，直到布尔表达式为真。如果表达式为假且等待语句处于阻塞状态，则可以选择在此期间应用信号赋值。该表达式通常作用于全局信号或寄存器，这些信号或寄存器可能没有保护锁。

固定时钟周期应用 [wait-for-time-with]

在指定数量的时钟周期内，可以对简单表达式求值并将其赋值给信号。赋值中左侧的信号仅在一个时钟周期内保持一个值。在

此 apply 语句中，信号会将该值保持 N 个时钟周期。在 apply 语句之外，必须为该信号赋予一个默认值。此项可以在 apply 语句中选择性指定

定义 *13 ConPro* 循环

```
对于 x = 从 ⟦ 递减至 b ⟦ 步长为 vs ⟧ 执行 il    oop ;

当 条件 成立时执行 循环体;

始终执行循环;

等待时间时间;
等待条件条件;
等待时间时间，使用i0;
等待条件条件，使用i0 ⟦ 否则 i1 ⟧;
```

以下示例展示了各种循环和分支的用法:

1 ... 2 数组链路：对象 uart[NUM_LINKS]； 3 -- 在顶层配置 UART 设备 4 for i = 1 to NUM LINKS do 5 开始 6 links_com.[i-1].接口(DEVLN.links_rx[i-1],DEVLN.links_tx[i-1]); 7 links_com.[i-1].波特率(115200); 8 end; 9 10 寄存器 d : 逻辑[8]; 11 导出 d; 12 13 进程 主进程: 14 开始 15 始终 do 16 开始 17 d <- 0; 18 while d < 10 do 19 开始 20 for i = 6 downto 0 do 21 开始 22 if d[i] = 0 then 23 d[i] <- 1; 24 end; 25 end; 26 等待 1 毫秒; 27 end; 28 end;

V 硬件架构

此前介绍的扩展 CCSP 模型可以直接在硬件中实现，并映射到微芯片级别 **[BOS11A][BOS10A]**。

V.1 进程

每个 *ConPro* 进程都在硬件层级上通过有限状态机（FSM）以及数据通路的寄存器传输级（RTL）架构实现，最终创建一个互连的多有限状态机架构，如**图 7**所示

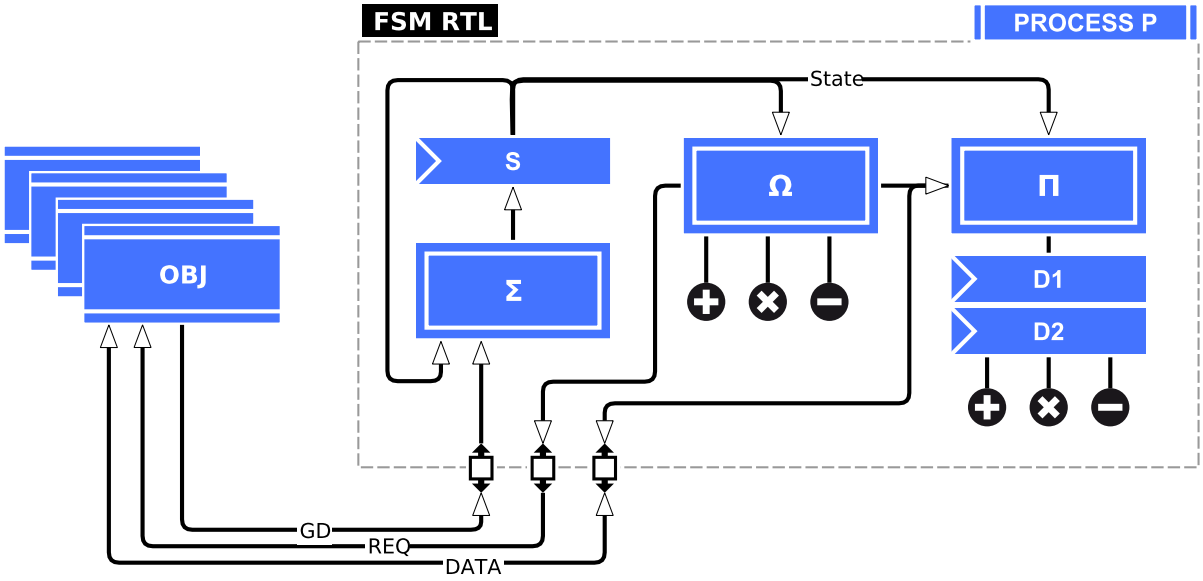


图 7 顺序 *ConPro* 进程的硬件架构及其与全局对象的互连（Σ：状态转换网络，*S*：控制状态寄存器，Ω：组合数据通路，Π：过渡数据通路）

*ConPro*进程的指令程序流程映射到有限状态机的控制流和状态上，并为每个进程额外设置一个启动状态和结束状态。复杂指令会拆分为多个状态。数据通路分为纯组合部分和带寄存器的转换部分。前者访问局部对象（只读）和全局对象（可读、可写并可进行控制访问）。对全局资源的访问始终由共享对象的访问调度器（互斥锁）进行保护。数据通路会激活请求信号（`RE/WE` 或 `CALL`）。有限状态机转换网络会读取并求值守卫信号 `GD`。当前访问该对象的状态会一直保持，直到守卫信号变为低电平（每个时钟周期只会发生一次，以避免竞态条件）。

共享程序函数与过程也通过顺序进程实现。

V.2 模块

所有层次化模块、进程、对象和实例化的硬件组件都会合并为一个扁平化的片上系统设计，并具有一个顶层端口接口，如图*8*所示。

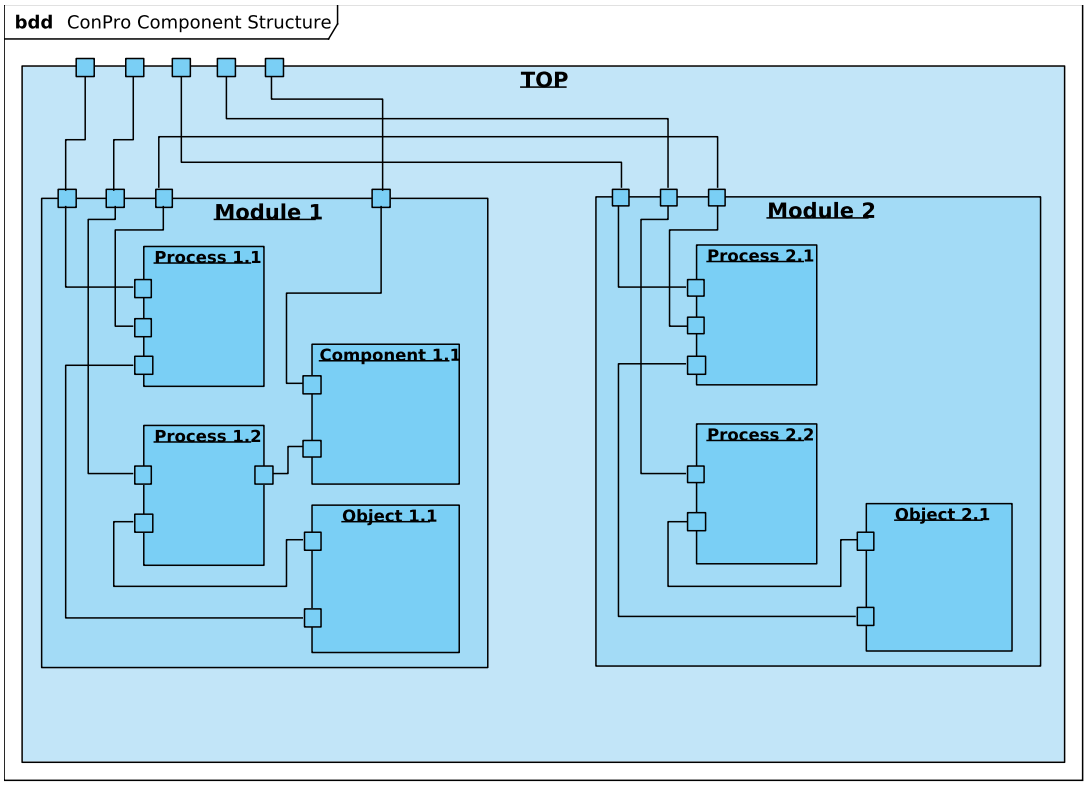


图 *8 ConPro* 分层模块架构及互连。

V.3 互斥调度器

对共享对象的访问必须由互斥锁通过互斥调度器进行内在保护。该调度器负责对并发访问进行串行化。调度器与所有访问该资源的进程相连。进程会发出请求（REQ），并等待保护锁释放（GD）；保护锁释放后，进程将被解除阻塞，这表示资源已获准使用，如**图 9**所示。

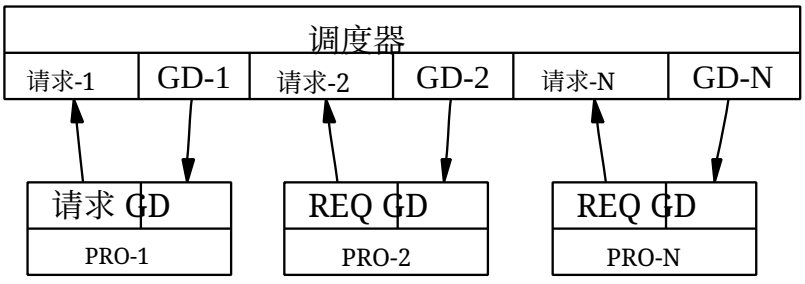


图 *9* 调度器模块架构与进程互连

有两种不同的调度器可供使用：

静态优先级调度器 [默认]

这是最简单的调度器，所需的硬件资源也最少。每个曾访问过该资源的进程都会获得一个唯一的优先级排序。如果不同进程同时访问该资源，调度器总是允许优先级最高的进程访问。存在

使用这种调度策略存在竞态条件的风险。通常，进程在源代码中出现的顺序决定其优先级：第一个进程的优先级最高，最后一个进程的优先级最低。一次经调度的访问至少需要两个时钟周期。

动态 FIFO 调度器

动态调度器通过进程队列实现公平调度。每个请求资源但竞争失败的进程都会存入一个 FIFO 有序队列中。当前一位所有者释放资源时，调度器会选择队列中等待时间最长的进程。动态调度器可避免竞态条件，但需要更多硬件资源。

这两种调度器的算法分别定义于定义 **定义 14** 和 **定义 15** 中。

Def. 14 静态优先级调度器：进程 *i* 的输入/输出：*{REQ-i,GD-i}*；共享资源块的输入/输出：*{ACT,ACK}*。进程 *i* 的请求会激活 *REQ-i*；如果资源未被锁定，则该请求会在 *if-then-else* 级联中获准交由下一进程处理。请求完成后，*ACK* 会被激活并释放锁定对象，同时释放该进程对应的 *GD-i*，以表明请求已完成。

循环 执行 ACT ← False；∀ gd ∈ {GD-1,GD-2,..} 执行 gd ← True；如果 REQ-1 ∧ ¬LOCKED 那么 LOCKED ←True；ACT ← True；为进程启动服务 *1* 2 ∧ ¬

否则如果 REQ- LOCKED 那么 LOCKED ←True；ACT ← True；为进程 *2* 启动服务 ... 否则如果 ACK ∧ REQ-1 ∧ LOCKED 那么 GD-1 ← False；LOCKED ← False；否则如果 ACK ∧ REQ-2 ∧ LOCKED 那么 GD-2 ← False；LOCKED ← False；...

定义 *15* 动态队列调度器：进程 *i* 的输入/输出：*{REQ-i,GD-i}*，共享资源块的输入/输出：*{ACT,ACK}*。进程 *i* 的请求会激活 *REQ-i*；如果资源队列为空，或该进程位于队列头部，则在 *if-then-else* 级联中向该进程授予请求。如果请求已完成，则激活 *ACK*，从资源队列中移除该进程，并释放该进程对应的 *GD-i*，以表示请求已完成。

循环 Do ACT ← False; ∀ gd ∈ {GD-1,GD-2,..} Do gd ← True; If REQ-1 ∧ LOCKED=[ ] ∧ ¬PRO-1-LOCKED Then LOCKED ← [PRO-1]; PRO-1-LOCKED ← True; OWNER ← PRO-1; ACT ← True; 为进程启动服务 *1* 2 ∧ =[ ] ∧ ¬ 2

Else If REQ- LOCKED PRO--LOCKED Then LOCKED ← [PRO-2]; PRO-2-LOCKED ← True; OWNER ← PRO-2; ACT ← True; 为进程*2*启动服务 ...

Else If REQ-1 ∧ LOCKED≠[ ] ∧ ¬PRO-1-LOCKED Then LOCKED ← LOCKED @ [PRO-1]; 将进程*1*加入队列 PRO-1-LOCKED ← True; Else If REQ-2 ∧ LOCKED≠[ ] ∧ ¬PRO-2-LOCKED Then LOCKED ← LOCKED @ [PRO-2]; 将进程*2*加入队列 PRO-2-LOCKED ← True; … Else If REQ-1 ∧ Head(LOCKED)=PRO-1 ∧ OWNER≠PRO-1 Then ACT ← True; 为进程启动服务 *1* ← 1

OWNER PRO-; Else If REQ-2 ∧ Head(LOCKED)=PRO-2 ∧ OWNER≠PRO-2 Then ACT ← True; 为进程启动服务 *2* ← 2

OWNER PRO-; … Else If ACK ∧ Head(LOCKED)=PRO-1 Then GD-1 ← False; PRO-1-LOCKED ← False; OWNER ← NONE; LOCKED ← Tail(LOCKED); Else If ACK ∧ Head(LOCKED)=PRO-2 Then GD-2 ← False; PRO-2-LOCKED ← False; OWNER ← NONE; LOCKED ← Tail(LOCKED); …

V.4 函数

共享函数通过进程和额外的函数调度器实现，该调度器用于保护函数及函数参数的并发访问。函数的输入和输出参数通过全局寄存器实现。任意时刻只能有一个进程调用函数进程，即提供可选的输入参数（通过局部寄存器）、启动函数有限状态机（起始状态），并等待函数进程终止（到达结束状态），最后将可选的返回数据传输到局部寄存器中。

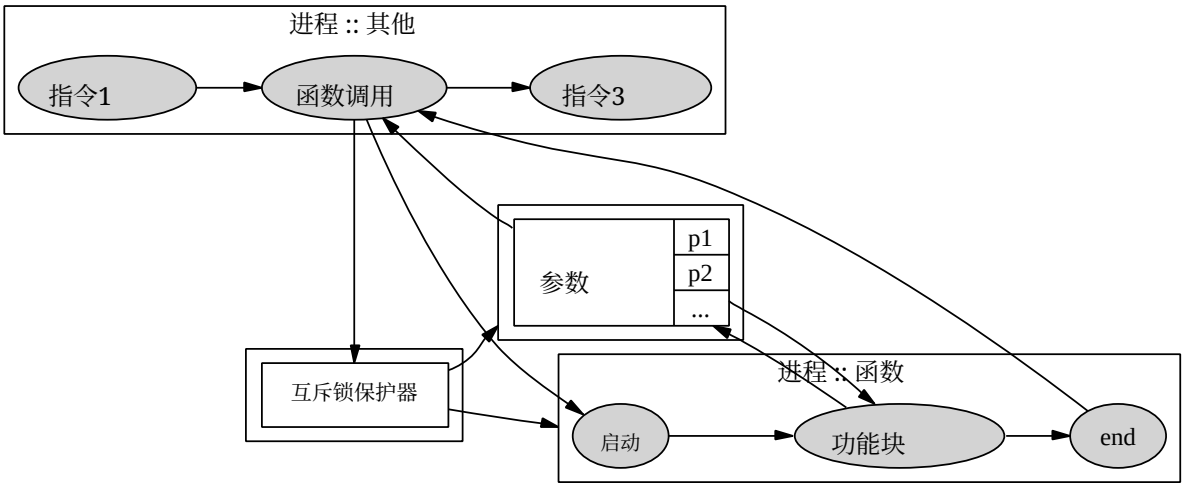


图*10* 使用互斥调度器实现共享函数调用，以串行化不同进程对函数的并发访问。

函数调用的控制流（每个进程状态图的一部分）如**图10**所示。函数进程及其互斥调度器通过硬件信号与调用进程相连。

输入信号是请求方法（即函数调用）和输入寄存器（参数）；输出信号是守卫信号（用于授予对函数的访问权限）和输出寄存器（用于返回值）。函数调用通常只需额外几个时钟周期即可完成数据传输和函数调度。

V.5 软件架构

除了推导所提出的硬件架构模型外，还可以推导出具有相同操作行为和功能行为的软件模型。

每个 *ConPro* 进程都在软件层通过轻量级进程（线程）实现，最终形成一个采用共享内存模型的多线程程序。全局对象通过受互斥锁保护且线程安全的全局函数实现。*ConPro* 函数则可实现为线程安全的可重入函数，无需互斥调度器。

针对 *C*语言 或 *OCaML* 编程语言，存在不同的后端。大多数 *ConPro* 对象和进程语句都可以直接用软件实现，但硬件块、信号以及使用信号的语句除外。EMI 接口可以提供硬件组件的软件实现。

VI 案例研究

VI.1 哲学家就餐问题

哲学家就餐问题是一个广为人知的问题，可用于说明并行和并发执行系统中的饥饿与死锁。**例 11**展示了实现哲学家进餐问题的部分程序代码。该系统由五个同时执行的进程组成，用以实现哲学家的行为，即进餐或思考。进餐时，他们需要两把叉子，也就是两种资源。每种资源由相邻的两位哲学家共享。哲学家进程通过数组构造定义（第 13 行）。每个进程的指令按顺序执行。叉子的资源管理通过信号量（抽象对象类型）实现。每个哲学家进程都尝试通过调用每个信号量的 down 方法，分配两把叉子，即左侧和右侧的叉子。若哲学家进程成功获取叉子，就会调用（内联的）函数 eat，并同时设置全局寄存器（进餐、思考）；如程序代码所示，这里用冒号而非分号表示（有界指令块）。事件对象 ev 用于同步这一组进程的启动。哲学家通过调用 await 方法等待事件（第 15 行）。主进程调用 wakeup 方法唤醒该事件（第 41 行）。所有进程均从主进程开始（第 40 行）。进程也被视为抽象对象，并提供一组用于控制进程状态的方法。

对象（如进程间通信）属于某个模块，必须先打开该模块（第 1 行）。每个模块由一组 EMI 实现文件定义，这些文件提供了该模块对象的所有必要信息（例如方法声明、对象访问，以及在硬件层和软件层上的实现）。

例 *11 ConPro* 源代码示例的组成部分：哲学家进餐问题的实现，该实现映射到多个使用信号量进行资源管理的进程上。

1 **打开** Core；**打开** 进程；**打开** 信号量；**打开** System；打开事件；*2* **对象** ev：事件；
3 **数组** eating, thinking：寄存器[5] of 逻辑；*4* **导出** eating,thinking；*5* **数组**
叉子：**语义对象**[5] with *6* 信号量.深度=8 and 信号量.调度器="先进先出"；*7* **函数**
eat(n)：*8* 开始 *9* eating.[n] <- 1,thinking.[n] <- 0；*10* 等待 5；

11 eating.[n] <- 0,thinking.[n] <- 1；12 以内联结束；13 数组 philosopher：进程[5] of 14 开始 15 ev.await ()；同步所有进程 16 if # < 4 then 所有数组索引小于 4 的进程 17 开始 18 始终执行 19 开始 20 先拿左边的叉子，再拿右边的 21 fork.[#].down ()；fork.[#+1].down ()；22 进餐 (#)；23 fork.[#].up ()；fork.[#+1].up ()；24 结束；25 结束 26 否则 27 开始 28 始终执行 29 开始 30 -- 先拿右边的叉子，再拿左边的 31 fork.[4].down ()；fork.[0].down ()；32 进餐 (#)；33 fork.[4].up ()；fork.[0].up ()；34 结束；35 结束；36 结束；37 主进程 main：38 开始 39 对于 i = 0 to 4 执行 40 philosopher.[i].启动 ()；41 ev.wakeup ()；开始游戏..... 42 以调度="基本块"结束；

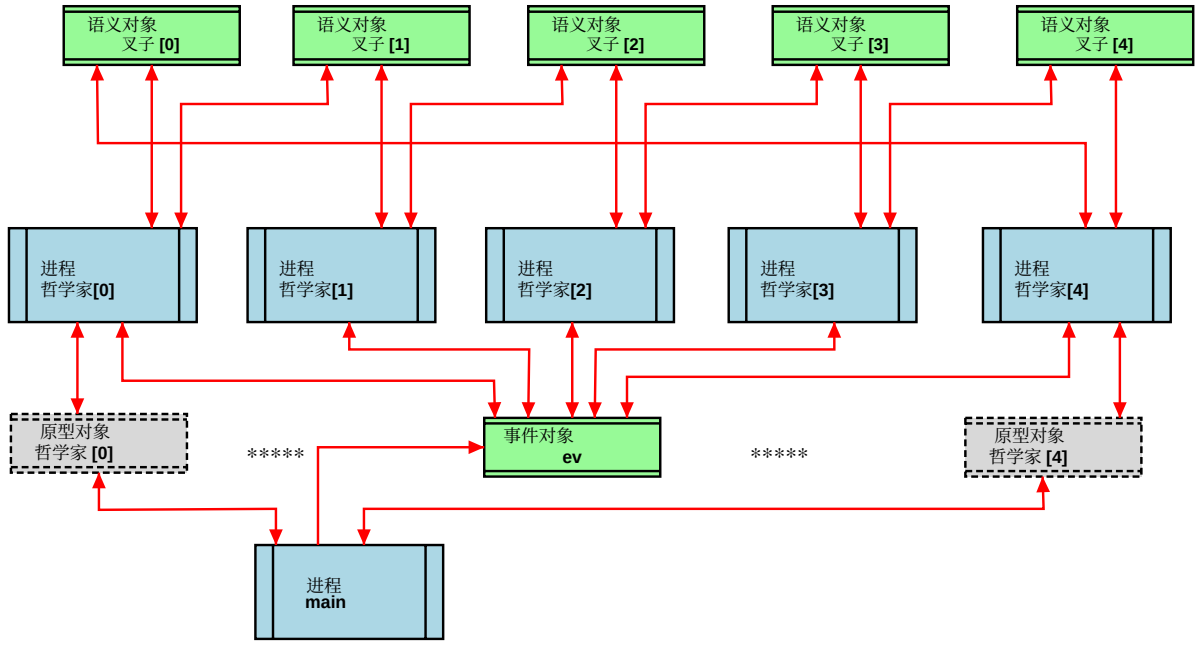


图 *11* 示例**例 11**中哲学家进餐问题实现的进程与进程间通信架构。

结构化的 CCSP 模型如**图 11**所示。进程也是对象和执行单元。每个全局共享对象都由一个互斥调度器通过串行化来解决并发访问问题。

这个人工电路可以直接在 FPGA 中实现。监视器寄存器进餐和思考 被导出到电路的外部端口，其信号可以从外部观察。事件对象*ev*用于同步所有哲学家进程的启动（精确的同步时钟事件）。假设各进程的控制路径有限状态机采用同步时钟，则电路的动态行为应始终相同。ADTO（信号量）的默认调度器行为是使用优先级多路复用器的优先级互斥调度器，即由进程编号决定并发访问的获胜者。调度顺序通常由访问相应对象资源的进程的逆序决定，即最后一个进程最先有机会赢得竞争。

由于哲学家进程以环形方式访问叉子，因此每把叉子只有两个进程会争用。进程 #0 获取信号量 0 和 1，进程 #1 获取信号量 1 和 2，依此类推。根据上述配置，信号量 #0 的进程优先级顺序为 (p#4 > p#0)，信号量 #1 的顺序为 (#p1 > p#0)，信号量 #2 的顺序为 (p#2 > p#1)，依此类推。因此，竞争中总是“左边”的哲学家会获胜（最后一个优先的进程编号）。这会导致第一轮总是出现死锁状态。

VI.2 自适应通信协议路由器 SLIP

本节展示了自适应路由 SLIP 协议栈的完整实现 **[BOS11A]**。SLIP 可根据网络规模（地址大小类别（ASC），范围为 4 至 16 位）、最大数据载荷（数据大小类别（DSC），范围为 4 至 16 位）以及网络拓扑维度大小（地址维度类别（ADC），范围为 1 至 4 位）进行扩展。网络节点通过（串行）点对点链路连接，并沿不同几何维度的度量轴排列：一维网络（ADC=1）可实现链和环，二维网络（ADC=2）可实现网格，三维网络（ADC=3）可实现

立方体等。每种维度类别均支持不完整网络（缺少链路）和不规则网络（缺少节点和链路），如**图 12**所示。

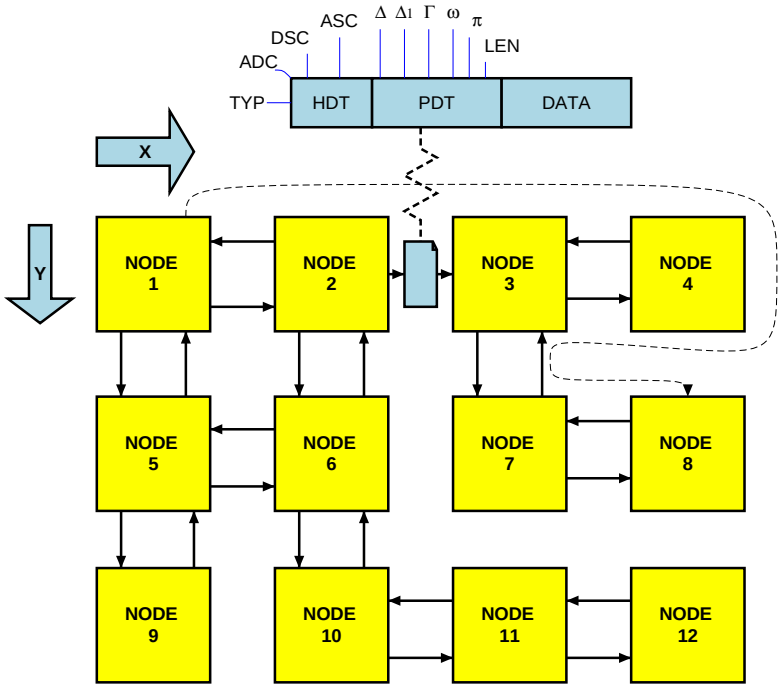


图 *12* 使用 *δ*-距离向量路由在二维网络中实现基于消息的通信。通过采用智能路由，系统支持拓扑结构不完整（缺少链路）和不规则（缺少节点）的网络。

基于消息的通信面临的一项主要挑战是路由，以及由此产生的节点唯一寻址问题。在高密度传感器网络中（例如材料集成损伤诊断系统），对节点进行绝对且唯一的寻址并不适用。SLIP 使用一种替代路由策略，即相对 δ-距离路由。δ-距离向量 Δ 指定从源代码到目标节点的路径，并统计每个维度上的节点跳数。

消息包包含一个头部描述符，用于指定数据包的类型以及可扩展参数 ASC、DSC 和 ADC，如**表 4**所示。网络地址维度 ADC 和大小类别 ASC 反映了网络拓扑，数据大小类别 DSC 表示数据载荷。消息主要有两种类型：请求和回复。

表 *4 SLIP* 消息格式（*HDT*：头部描述符，*PDT*：数据包描述符）

条目	大小 [位]	描述
HDT:ADC	2	地址维度类别
HDT:ASC	2	地址大小类别
数据大小类别	2	数据大小类别
消息类型	2	消息类型 = {请求, 回复、存活、确认}
PDT:Δ	ADC数量*ASC位数	实际增量向量
PDT:Δ^0^	ADC数量*ASC位数	原始增量向量
PDT:Γ	2*ADC数量	反向传播向量
PDT:ω	位（ADC）	首选路由方向
PDT:π	ASC位数	应用层端口
PDT:LEN	位（DSC）	数据包长度
DATA	LEN*位（DSC）	有效载荷数据

数据包描述符紧随头部描述符之后，其中包含：实际增量向量 Δ、原始增量向量 Δ^0、首选路由方向 ω、应用层端口 π、反向传播向量 Γ，以及后续数据部分的长度。数据包头的总位数取决于可扩展参数元组 {ASC、DSC、ADC} 的设置，这些设置可优化应用特定开销和能效（空间与时间方面）。每当数据包沿某个方向被转发（路由）时，增量向量都会在相应维度的条目中减少（幅度减小）。

例如，沿 x 方向路由可得到：$\Delta_l = \Delta_{l,-l}$。当 Δ=0 时，消息即已到达目的地，可以传递至应用端口 π。智能路由有多种规则，按下列顺序依次应用，直到数据包能够被路由（或被丢弃），如**定义 16**所示。首先尝试常规 XY 路由，依次沿各个方向路由数据包，目标是使每个方向上的增量计数最小。如果无法做到（由于连通性不足），则按照消息数据包描述符的 gamma 条目 Γ 所示，将数据包沿反向发送。反向路由用于避开小型区域陷阱，而反向路由则用于避开大型区域陷阱，或将数据包返回源节点（数据包无法传递）。路由决策基于当前消息条目 {Δ, Γ, ω}，并通过反映当前网络拓扑以及消息已经过的路径（包括后端陷阱）来实现自适应路由，从而通过选择不同的路由方向找到替代路径。

定义 *16* 智能路由协议 *SLIP*（简化版）*1*M：消息(Δ,Δ 0 ,Γ,ω,π,Len,数据)
*2*PRO smart_route(M)： *3* IF Δ=0 THEN DELIVER(M,π) ELSE *4* 尝试 route_normal(M) ELSE *5* 尝试 route_opposite(M) ELSE *6* 尝试 route_backward(M) ELSE DISCARD(M); *7*PRO route_normal(M)： *8* FORONE δ i ? Δ 尝试 minimize δ i： *9* route(Δ,M) WITH δ i：=(δ i +1)|δ i<0 ?(δ i −1)|δ i>0 ;

10 PRO route_opposite(M)： *11* FOREONE δ i ? Δ 尝试 minimize δ i： *12* route(Δ,M) WITH δ i：=(δ i −1)|δ i<0 ?(δ i +1)|δ i>0 ；*13* PRO route_backward(M)： *14* 发送 M（从方向 δ i 接收）*15* 返回方向 -δ i，使用 Γ i =-δ i /|δ i |;

SLIP 使用点对点链路，对排列在 n 维网络空间（线形、网格、立方体）中的节点进行 Δ 寻址，从而实现消息的智能路由。网络在节点大小、计算能力和内存方面可以是异构的。该通信协议可扩展，能够适应不同的网络拓扑和规模。网络中的节点既是服务端点，也是路由器，因此每个网络节点都必须实现路由器。不过，路由器只需了解连通性信息，即可将数据定向至相邻节点。

总而言之，路由信息始终保存在数据包中，包括：

1. 头部描述符（*HDT*），用于指定地址大小类别 *ASC* 和地址维度类别*ADC*（例如，2 表示二维网格）；2. 数据包描述符（*PDT*），其中包含路由和路径信息；最后是数据部分。

SLIP协议的实现必须处理消息格式可变的情况。SLIP 最初是为采用 ASIC 和 FPGA 目标技术的低资源片上系统实现而设计的，但也需要提供软件版本。为尽量降低资源需求，硬件实现通常会限制维度和地址空间，仅支持消息空间的一个子集，例如 ADC=2、ASC=8 （即 δ=-128,..,127）。

节点应并发处理多个串行链路连接和传入的数据包，因此协议栈是一个大规模并行系统，并采用 *ConPro* 行为式多进程模型实现。

将协议栈划分为多个并发执行并通过队列通信的进程来实现该编程模型，如 **图13** 所示。

每条链路由两个进程提供服务：一个消息解码器用于处理传入消息，一个编码器用于处理传出消息。数据包处理器 pkt_进程 会应用一组智能路由计算函数（按给定顺序依次应用 route_normal、route_opposite、route_backward，直至能够进行路由），以确定最佳路由方向。进程之间通过队列进行通信。三个数据包池分别存放数据包的 HDT、PDT 和数据部分，并通过数组实现。可复制数据包处理器，以加快数据包处理速度。

SLIP 实现划分为接收器状态机和发送器编码器。接收器状态机按顺序逐步解析传入的字节流，如**例12** 所示。

例*12 SLIP*接收器状态机（简化版）

1 **进程** pkt_进程:*2* 开始 *3* *--4* -- 处理传入的数据流并 *5* -- 解析数据包头 *6* *--7* **类型** 状态: { *8* S_rx_hdt; *9* S_rx_pdt_d1; *10* S_rx_pdt_d2; *11* S_rx_pdt_d3;

12 S_rx_pdt_dp; 13 S_rx_pdt_gam; 14 S_rx_pdt_len; 15 S_rx_data; 16 S_rx_deliver; 17 S_discard; 18 S_存活; 19 }; 20 21 寄存器 状态: 状态; 22 状态 <- S_rx_hdt; 23 始终 do 24 开始 25 匹配状态 with 26 开始 27 当 S rx hdt:*28* < {d,tmo} - 链路 _read(); 29 ind <- get_rx_pkt(d); -- 从池中分配数据包结构 30 typ <- d[6 to 7],asc <- d[4 to 5],dsc <- d[2 to 3],adc <- d[0 to 1]; 31 if adc <> ADC 代码 or*32* <> dsc DSC 代码 or *33* <> asc ASC 代码 then*34* < state - S discard *35* = else if typ HDT_类型_ALV or typ = HDT_类型_ACK then 36 state <- S 存活 *37* else 38 state <- S_rx_pdt_d1; 39 当 S_rx_pdt_d1: 40 if ASC = 4 then 41 开始 42 {d,tmo} <- 链路_read(); 43 if tmo = true then 引发 Rx_错误; 44 pkt_池_pdt.[ind].pdt_d1 <- to_int(d[0 to 3]); 45 pkt_池_pdt.[ind].pdt_d1_hop <- to_int(d[4 to 7]); 46 if ADC > 1 then 47 state < _ _ _ 2 - S rx pdt d else 49 state <- *48* S_rx_pdt_dp; 50 end; 51 if ASC = 8 then 52 开始 53 for n = 1 to 2 do ... 54 ... 55 当 S rx data: *56* < {d,tmo} - 链路_read(); 57 pkt_池_data.[off] <- d; 58 off <- off + 1, count <- count + 1; 59 if count = len then 60 state <- S_rx_deliver; 61 ... 62 end; 63 end; 64 end;

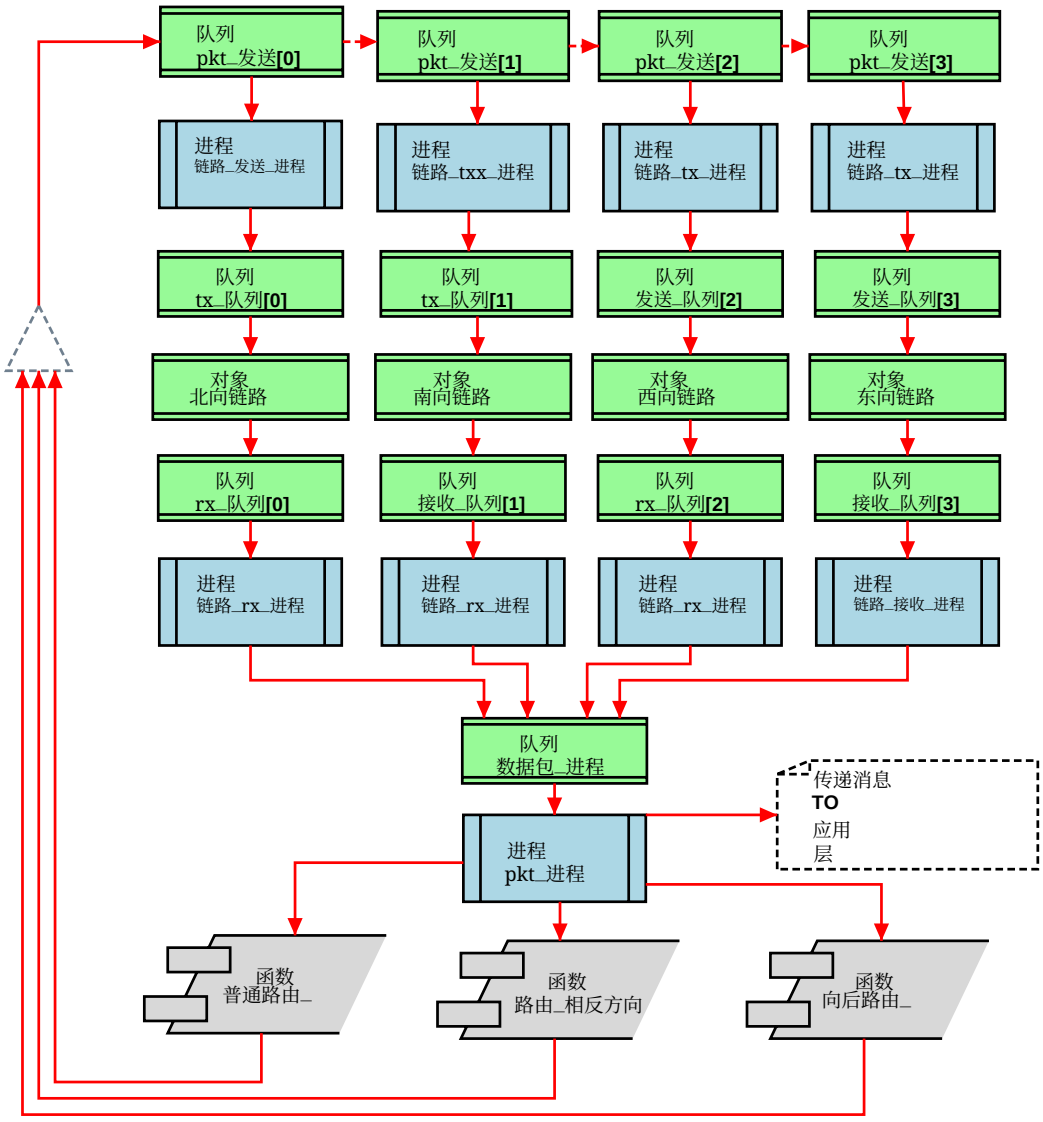


图*13 SLIP*协议的进程及进程间通信架构 *sta* *ck.*

实现了一个测试设置，其中包含*SLIP*的路由处理部分：A.）在硬件中实现（寄存器传输级片上系统，使用 Mentor Graphics Leonardo Spectrum 和 SXLIB标准单元库进行门级综合）；B.）在软件中实现（*SunOS*、*SunPro C*语言编译器）。接收一个带有 ADC=2, Δ=(2,3) 的数据包，以及节点 L=(-y,-x) 的链路配置；该数据包从第二条链路（-x）接收 [L01]，并首先由**路由_常规**规则处理（需要连接的 +x /+ y 链路）[L03],，最后由**路由_相反**规则处理[L04]，将修改后的数据包转发至链路_0 进程[LA0]。

表 **表 5** 和 **表 7** 展示了硬件 (HW) 与功能等效的软件 (SW) 实现的综合和仿真结果。它们显示出较低的资源需求和延迟。不同的检查点 Lxx 表示数据包处理的进度。括号中的数值表示相对于前一检查点的延迟进度。

表 *5* 使用数据包池实现的 *SLIP* 路由部分硬件实现所需资源的比较：*(1)* 可变数组，*(2)* 寄存器数组。*ASIC* 综合使用 *Leonardo Spectrum* 软件和 *SXLIB* 标准单元库进行。

资源	可变数组1	寄存器2
寄存器 [触发器]	767	587
面积 [门]	12475	10758
路径延迟 [纳秒]	18	16
综合源代码 CP ⟶ VHDL	1109 ⟶ 9200 行	1109 ⟶ 7900 行

对比了两种不同的存储资源模型：采用 RAM 块的可变数组和寄存器数组。令人意外的是，寄存器存储模型比 RAM 存储模型更节省资源。但这一结论仅适用于 ASIC 技术，不适用于具有预定义功能单元和内置片上块 RAM 资源的 FPGA 技术。由于 CREW 访问行为，寄存器存储模型可降低并行数据包处理的计算延迟。赛灵思 Spartan III（1000k 等效门）FPGA 的完整设计实现与 ASIC 综合结果的比较见**表 6**。

表 *6* 包含简单应用层的 *SLIP* 完整硬件实现所需资源对比：（*1*）*FPGA* 综合使用 *Xilinx ISE* 完成；（*2*）*ASIC* 综合使用 *Leonardo Spectrum* 软件和 *SXLIB*标准单元库完成

资源	FPGA1	ASIC2
寄存器 [触发器]	2925	15000
查找表 [#]	11261/15360	-
面积 [门]	-	244 千门 ≈ 2.5毫米^2 | 0.18μ米
Conpro 源代码	4000 行，34 个进程，30 个共享对象（16 队列，2 个定时器）	
综合后的 VHDL	32000 行	

表 *7 SLIP* 路由部分的硬件和软件实现的仿真结果。硬件：数据包池：（*1*）可变数组，（*2*）寄存器数组，时钟周期。软件：*SunPro CC*、*SunOS*、*USIII*、*CPU* 机器操作

检查点	时钟周期1	时钟周期2	机器操作
L01	104	102	60000
L03	113 (δ=9)	107 (δ=5)	60019 (δ=19)
L04	187 (δ=74)	148 (δ=41)	60796（δ=777）

检查点	时钟周期1	时钟周期2	机器操作
LA0	235 (δ=48)	184 (δ=36)	62305 (δ=1509)

VI.3 并行代码变换虚拟机 uFORTH

与通过通信交换数据、执行静态程序的常见程序控制处理器不同，μFORTH 基于移动代码和代码变形，可提供包含嵌入式数据的自适应机器代码。程序本身可以修改代码（以及嵌入式数据），并将其发送给其他处理器。

这种基于栈的运行时环境专为实现移动代理而设计，通过动态代码变形，在低功耗和组件高度微型化的约束下运行（但不限于这些约束）。它采用经过修改和扩展的 Forth 编程语言，以支持移动和自适应代码。Forth 是一种基于栈的解释型语言，其源代码极为紧凑（大多数操作都是零操作数机器指令）。此外，Forth 具有可扩展性。代码和用户可以在运行时定义新的语言构造（称为单词，即零操作数函数）。

Forth 程序包含由 FORTH 处理单元直接执行的内置核心指令，以及用户定义的高级单词和对象定义；这些定义会被添加到字典表中，并可从中查找。该字典在分布式系统和传感器网络中交互节点的实现中发挥着核心作用。单词可以由 Forth 程序自身或平台进行添加、更新和删除（遗忘）。用户定义的单词由一系列单词组成。

节点运行时环境中一个 Forth 处理单元（PU）的主要系统架构如**图 14**所示。

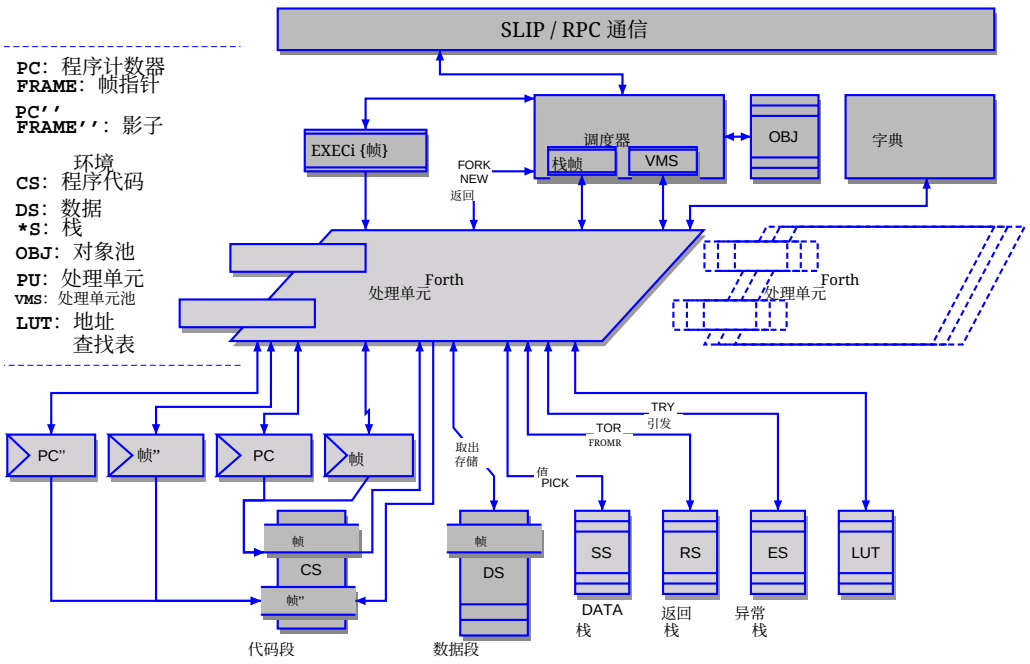


图 *14* 运行时架构，包括 μ 个 *Forth* 数据处理单元、共享内存和对象、字典、调度器以及 *SLIP* 通信路由器。

一个完整的运行时单元由通信系统、智能路由协议栈（用于连接处理器）、一个或多个配备代码变形引擎的 FORTH 处理单元，以及资源

管理、代码重定位和字典管理，以及负责管理程序执行与分配的调度器组成。这些通常是操作系统的一部分，但这里并不存在操作系统。

FORTH 处理单元最初会等待一个待执行的帧（一个 FORTH 程序）。在程序执行期间，FORTH 处理单元会与调度器交互，以执行程序分叉、帧传播、程序终止、对象创建（分配）和对象修改。对象集合由进程间通信对象（IPC：互斥锁、信号量、事件、定时器）和帧组成。变量和数组分为私有的和公共的（节点可见）。所有程序帧都可以通过查找字典中存储的引用来访问公共变量。程序字、内存变量和对象的重定位通过帧绑定查找表 LUT 完成。

调度器是本地并行执行的一组 FORTH 处理单元与通信系统之间的桥梁。通信系统是一个位于 SLIP 之上的远程过程调用（RPC）接口；SLIP 是一种容错的、基于消息的通信系统，用于借助先前介绍的 SLIP 路由器，通过智能增量距离矢量路由在节点之间传输消息（包含代码）。

简单的 FORTH 指令格式是进行代码变形的合适起点，即程序能够修改自身或生成一个修改后的副本，这主要是先前执行的计算所产生的结果。计算结果和部分处理状态可以直接存储在程序代码中，从而改变程序的行为。标准的 FORTH 核心指令集经过扩展和调整，用于在具有二维网格拓扑结构的网格网络中实现代理迁移。FORTH 程序包含在一个称为代码帧的连续内存片段中。

简单的 FORTH 指令格式（文本和二进制代码）是进行代码变形的合适起点，即程序能够修改自身或生成一个修改后的副本，这主要是先前执行的计算所产生的结果。计算结果和部分处理状态可以直接存储在程序代码中，最终改变程序的行为。标准的 FORTH 核心指令集经过扩展和调整，用于在具有二维网格拓扑结构的网格网络中实现代理迁移。在我们的 System 中，FORTH 程序包含在称为帧的连续内存片段中。

帧可以传输到远程节点和处理单元上并在那里执行。**表 8**总结了为代码变形提供的新单词。这些指令可用于修改程序的行为，并保留当前程序执行状态。在我们的系统中，FORTH 程序包含在一个连续的内存片段中，即代码帧。帧可以传输到远程节点和处理单元上并在那里执行。程序代码的修改始终在影子帧环境中进行，该环境可以与执行帧相同。这是原位代码修改所采用的默认情况。可以分配一个或多个不同的帧，用于非原位修改；如果代码变形后仍要使用执行帧，则必须采用这种方式。所有代码变形指令都作用于影子帧。执行帧和影子帧各自都有自己的代码指针。

表 *8* μ*Forth* 代码帧管理与修改扩展

Word	描述
c!（帧 -- ） RESET c!	SETC：设置用于代码变形的影子环境帧；`RESET` 将影子帧的代码指针设为影子帧的起始位置
>>c (m1 m2 --)	COPYC：切换到变形状态：将代码从程序 帧中 *m*~1~ 和 *m*~2~ 之间的内容复制到影子帧 （包括标记）
>c (--)	TOC：将程序帧中的下一个单词复制到影子帧中
s>c (n --)	STOC：从栈中弹出 *n* 个数据值，并将这些值作为单词字面量存储在影子帧中 erals in shadow frame
<m> (--)	标记：在程序帧中的任意位置设置标记
<m>@ (-- 标记)	获取标记：返回一个标记（将符号名称映射到唯一的 数字）
<m>! (--)	设置标记：在影子帧中的标记之后设置影子代码指针。 在影子帧中搜索标记，因此可以在执行 帧中就地操作，也可以在新建或复制的影子帧中操作（其中已包含代码 和标记）。可使用 STOC 和 TOC 指令编辑影子帧代码的部分范围 使用 STOC 和 TOC 指令

表 *9* μ*Forth* 扩展：提供 *1.* 字典修改与对象创建，以及 *2.* 多进程支持与帧分发。

Word	描述
变量 x 数组 [n,m] x 变量* x 数组* [n,m] x	新建一个变量或数组并分配内存。前两个 定义会创建公共对象，并将其添加到字典中。 星号定义会创建私有对象。
对象 互斥锁 x 对象 帧 f	创建（分配）一个新的进程间通信或帧对象。该对象会添加到 字典中。其他受支持的进程间通信对象类型：信号量、事件、 定时器·
导入变量 x 导入对象 x	从字典中导入变量或对象。如果未找到，则 程序执行终止（返回状态 0）
叉子（dx dy --）	将影子帧的内容发送到相对于当前节点的位置，以供执行 节点。若*dx*=0 和*dy*=0，则影子帧将在本地执行，并发—— 地在另一台 FORTH 处理单元上执行。分叉指令 返回帧序列或处理单元编号
join（id -- ）	等待分叉帧终止，或等待接收并执行 回复程序帧。
状态返回	结束程序的执行。如果状态为零，则不会生成回复。 如果状态等于 -1，则会生成空回复。最后，如果 状态等于 1，则会将影子帧的内容发送回 执行帧的原始发送方

多处理器 Forth 节点的所有架构部分，包括 SLIP 通信、μForth 处理单元、调度器、字典和重定位支持，都在多 RT 级别映射到硬件，并使用 ConPro 编译器集成到单个片上系统设计中。资源需求取决于设计参数的选择，范围为 1M–3M 个等效门（按 FPGA 架构计）。整个设计划分为 43 个并发执行的顺序进程，它们通过 24 个队列、13 个互斥锁、8 个信号量、52 个 RAM 块、59 个共享寄存器和 11 个定时器进行通信。

下面介绍一些实现细节。**例 13**中的第一个简化代码片段展示了词典的实现，主要由函数查找（第 44–69 行）和定义 （第 17–42 行，基本上使用查找算法）组成。词典存储在一个 RAM 块中（var 存储类），名称字符串则使用紧凑逻辑位向量实现。

示例 *13* μ*Forth* 单词词典（简化版，词名最大长度为 *8* 个字符 == *64* 位）

```
1 类型 dictionary_row : { 2 dict_flag: 逻辑[4]; 3 -- 词名，8 个字符，打包格式 4 dict_name:
逻辑[64]; 5 -- 帧编号（帧查找表所需） 6 dict_frame: 整数[FRAMES_WIDTH]; 7 -- 相对于帧基
址的程序计数器起始地址 8 dict_pc: 整数[FRAME_SIZE_WIDTH]; 9 -- 词定义的长度 10 dict_
len: 整数[FRAME_SIZE_WIDTH]; 11 }; 12 ... 13 数组 dict: 变量[DICT_SIZE] 的 dictionary_
row; 14 异常 Found; 15 16 -- 向字典添加新词
17                                          _                [      _      ],
18                        [      _     _      ],        [      _     _      ]

函数 def(name: 逻辑[64]，kind: row flags, frame:整数 FRAMES WIDTH len:整数 FRAME
SIZE WIDTH off:整数 FRAME SIZEWIDTH): 19 开始 20 尝试 21 开始 22 对 i = 0 至 DICT
SIZE-1 执行23                                _ 开始 24 如果 dict.[i].flag <> ROW_EMPTY 那么
25 -- 与查找相同，搜索现有词条 -- 26 .. 27 -- 未找到，新建条目 28 dict.[cfree].name <-
name; 29 dict.[cfree].frame <- frame; 30 dict.[cfree].off <- off; 31 dict.[cfree].len <- len; 32
带有 33 开始 34 当 Found: 35 开始 36 -- 仅更新条目 37 dict.[dind].frame <- frame; 38 dict.[
dind].off <- off; 39 dict.[dind].len <- len; 40 结束;
```

41 结束; 42 结束; 43 44 函数 lookup(name: 逻辑[64]，forget: 布尔型) 45 返回 (found:布尔型, frame: 整数 [FRAMES_WIDTH]，pc: 整数 [FRAME_SIZE_WIDTH]): 46 开始 47 寄存器 dind: 整数[DICT_SIZE_WIDTH]; 48 尝试 49 开始 50 对 i = 0 至 DICT SIZE-1 执行 51 开始 52 dind<-i; 53 如果 dict.[i].flag <> ROW_EMPTY 那么 54 开始 55 如果 name = dict.[i].name 则引发 Found; 56 结束; 57 结束; 58 found <- false; 59 结束 60 带有 61 开始 62 当 Found: 63 开始 64 frame <- dict.[dind].frame; 65 pc <- dict.[dind].off; 66 found <- true; 67 结束; 68 结束; 69 结束;

下一个代码片段 **例 14**展示了主执行循环。由于这些栈是私有存储，仅供虚拟机字节码解释器访问，因此在 *vmexec* 进程内部定义。每个栈都拆分为单独的寄存器（栈顶）和一个存放更深层元素的变量数组。写入寄存器只需一个时钟周期，读取可立即完成；相比之下，存储在 RAM 块中的变量通常需要两个时钟周期才能完成读写访问。多个寄存器可以并发访问，而 RAM 单元只能按顺序访问。

例： 14 µ*Forth* 主执行循环1 ...2 **类型** uForth_命令：
{ 3 ----------4 -- 数据栈 命令栈
5 ----------
6 复制; -- (S) 7 QDUP; 8 ----------
9 -- 算术运算 命令_栈 10 ----------11
ADD; -- (S) 12 SUB; -- (S) 13 -- ----------
14 -- 帧处理 命令_控制 15 -- ----------

16 叉子; 17 .. 18 }; 19 进程 vmexec: 20 开始 21 始终执行 22 开始 23 while ret = false do 24 开始 25 -- 获取下一条指令 26 单词 <- cs.[pc]; 27 -- 在一个有限状态机状态和一个时钟周期内执行大型有界块命令！ 28 cmd <- 单词[0 to 11],数据 <- to_int(单词[0 to 11]), 29 wd <- 单词[12 to 15] lor 变形, 30 sm <- BAL, rm <- BAL, 31 t0'<-t0,t1'<-t1,t2'<-t2,r0'<-r0,r1'<-r1,r2'<-r2, 32 tn_p_s1 <- (tn_p-1) land SS_SIZE_MASK, 33 rn_p_s1 <- (rn_p-1) land RS_SIZE_MASK, 34 tn_next_a1 <- (tn_next+1) land SS_SIZE_MASK, 35 rn_next_a1 <- (rn_next+1) land RS_SIZE_MASK, 36 pc <- pc + 1; 37 匹配 wd 与 38 开始 39 -- 值 => 栈 40 当 VAL: 41 开始 42 t0 <- 数据, 43 sm <- 入栈; 44 结束; 45 当 命令栈: 46 开始 47 匹配 cmd 与 48 开始 49 -- 栈 <=> 栈 50 当 复制: 51 开始 52 t0 <- t0', 53 sm <- 入栈; 54 结束; 55 .. 56 结束; 57 结束; 58 结束; 59 -- 栈操作的收尾 60 匹配 sm 与 61 开始 62 当 PUSH: 63 开始 64 t1 <- t0', 65 t2 <- t1', 66 tn.[tn_next]<-t2'; 67 tn_p <- (tn_p + 1) land SS_SIZE_MASK, 68 tn_next <- (tn_next + 1) land SS_SIZE_MASK; 69 if CHECK STACK = true then_

70 开始 *71* tn_c <- tn_c + 1；*72* 如果 tn_c >= 堆栈段寄存器_大小，则引发栈_溢出；*73* 结束；*74* 结束；*75* .....*76* 结束；*77* 结束；

一条 FORTH 字指令的典型执行时间少于 10 个时钟周期。与非优化的冯·诺依曼架构处理器相比，后者执行一条低级机器指令需要 6 个时钟周期，高级指令的执行性能显著优于经典微处理器。字节码循环得益于高度并行的数据通路操作（例如第 28—36 行），这些操作可在一个时钟周期内执行。栈操作分两个步骤完成：准备阶段（第 42—55 行）和最终化阶段（第 60—74 行）。主命令解码器按命令类别（字面量、栈、控制、变形等）划分，并配有子解码器。

VII 结论

ConPro 编程语言基于并发通信顺序多进程模型，并支持进程间通信和守卫原子操作，非常适合用于实现并行控制和数据处理系统。算法可以从传统的顺序编程中复用。常见算法可以直接编程并实现为硬件电路。ConPro 综合工具能够高效地在硬件中实现复杂算法，例如通信协议或高层虚拟化处理器；这些算法需要并行性，并在控制路径级引入并发性，其门复杂度可低于或超过 100 万。除此之外，软件后端也能从同一 Conpro 程序生成软件，其功能行为几乎相同。此外，系统还提供丰富的同步对象。硬件块和外部组件通过基于方法、面向对象的编程模型（外部模块接口）进行访问。

VIII 参考文献

[BAR73] M. R. Barbacci，《寄存器传输级（rt）系统的设计空间自动探索》，1973年，学位论文。[BOS11A] S. Bosse，《使用独特的行为编程和多进程模型结合高级综合的并行与分布式系统硬件—软件协同设计》，SPIE 2011 微技术会议论文集，2011年4月18日至20日，布拉格，EMT 102会议：超大规模集成电路与系统，DOI 10 1117 12 888122 。/。BOSA S. Bosse，《使用行为编程和多进程模型针对嵌入式系统进行复杂片上系统设计的硬件综合》，第55届 IWK 国际科学研讨会论文集，C4会议，伊尔梅瑙，2010年9月13日至17日，2010年。[BOS12A] S. Bosse、F. Pantke，《使用多智能体系统在传感器网络中进行分布式计算和可靠通信》，《生产工程：研究与开发》，2012年，ISSN：0944-6524，DOI:10.1007/s11740-012-0420-8[BOS12B] S. Bosse、F. Pantke、F. Kirchner，《使用多智能体系统在分布式低功耗传感器网络中进行数据处理与通信》，第一届系统集成智能联合研讨会论文集：产品与生产工程的新挑战，2012年6月27日至29日：汉诺威，德国，专题会议“感知材料使能技术——传感器

[10]

2012
集成”，[12] BOS C S. Bosse、F. Pantke、F. Kirchner，《使用多智能体系统和代码变形在传感器网络中进行分布式计算》，第11届人工智能与软计算国际会议 ICAISC 2012 论文集，2012年29.
4 3 5 2012 日–。 .2012
[16] 2016,
年，扎科帕内，波兰，施普林格，COR Cortes, A.、Velez, I. 与 Irizar, A.（11月）。《使用 Vivado HLS 针对 Zynq SoC 进行高级综合：图像处理案例研究》。载于《2016年电路与集成系统设计会议（DCIS）》（第1–6页）。IEEE [COU08] P. Coussy、A. Morawiec（编），《高级综合——从算法到数字电路》，施普林格，2008年[EDU09] S. Bosse，《并行与分布式嵌入式系统》课程讲义，https://www.edu-9.de/Lehre/parsys2k，在线，访问日期
1 2 2023 。. GUP S. Gupta、R.K. Gupta、N.D. Dutt、A. Nicolau，《SPARK：数字电路
[04]
高级综合的并行化方法》，克鲁维尔学术出版社 2004
[02] KAT V. Kathail、S. Aditya、R. Schreiber、B. R. Rau、D. C. Cronquist，《PICO：自动设计定制计算机》，IEEE Computer,
35 9 39 47, 2002 ()，第 pp 页 - KU D. C. Ku、G. Micheli，《时序与同
[92] 1993
步约束下 ASIC 的高级综合》，克鲁维尔，[17] MAN P. Mantovani，（2017）。《可扩展片上系统设计》。哥伦比亚大学 [MEE12] W. Meeus、Kristof Van Beeck、T. Goedemé、J. Meel 和 D. Stroobandt，2012
《当今高级综合工具概览》，Des Autom Embed Syst，[08] 2008
MER Meredith，M.（ ）。《使用 Forte 的 Cynthesizer 进行高级 SystemC 综合。高级综合：从算法到数字电路》，75-97。[PER13] T. P. Perry，（2013）。《在可配置平台上快速开发信号处理与通信系统的软件工具》（博士学位论文，格拉斯哥大学）[PHI08] P. Coussy、A. Morawiec（编），《高级综合——从算法到数字电路》，施普林格，2008年[PIC22] L Piccolboni，（2022）。《面向加速器的多功能接口》。哥伦比亚大学。[RAD02]D. Radcliffe，（2002）。《从传统编程进行硬件综合》，学位论文，信息技术与电气工程学院，昆士兰大学
[98] 1998 SHA R. Sharp，《更高层次硬件综合》，施普林
[01]
格，ZHU J. Zhu，《MetaRTL：提升 RTL 设计的抽象层次》，DATE ’01：《欧洲设计、自动化与测试会议论文集》（2001年），第71–76页

IX 附录：ConPro 编程语言

数据类型

预定义的序数数据类型 DT 汇总于**表10**中。

表10 数据类型 *DT*

语句	类型 DT	描述
整数[N]	INT	N 位数据宽度的有符号整数
逻辑 逻辑 [N]	逻辑	无符号整数和逻辑向量（值集合 {0, 1, Z, H, L}），数据宽度为 N 位。
char	CHAR	字符
bool	BOOL	布尔类型（值集合 {true, false}。
值	值	无类型值（整数、逻辑、字符、布尔型），具有数据类型和在编译时指定的位宽（仅用于常量）

值

表达式中使用的寄存器、变量和信号的数据内容值，根据值格式关联到不同的序数数据类型*DT*，汇总见**表11**。

表11 值和数据类型 *DT*

值	Type	描述
-2, -1, 0, 1 ,2 ,3 ,4, ...	INT	有符号整数（十进制格式）
0,1,2,3,4,...	整数、逻辑	无符号整数（十进制格式）
0x1,0x2,...	整数、逻辑	无符号整数或逻辑（十六进制格式：0-9、A-F、a-f）
0b110,0b0101,...	整数，逻辑	无符号整数或无符号逻辑值（二进制格式：0,1）
0l111,0lZZZ	逻辑	逻辑值（逻辑多值格式；0,1,L,H,Z）
'a','A',...	CHAR	字符
"abc"	字符串	字符串（字符数组）
真、假	BOOL	布尔值
纳秒，mi-crosec，毫-秒，秒	UNIT	时间单位（与整数值搭配使用）
hz，千赫兹，兆赫，吉赫	UNIT	频率单位（与整数值搭配使用）

表达式与赋值

表达式用于赋值、分支、函数应用和循环中。操作包括算术、关系和布尔/逻辑运算，如**表12**所示。

表 *12* 算术、关系和布尔/逻辑（按位）运算符及其适用的数据类型

运算符	Type	描述
+,-,*,/	INT, 逻辑, CHAR	加法、减法（取反）、乘法、除法。（字符：ASCII 码）
< <= > >= = <>	INT, 逻辑, CHAR	小于、小于等于、大于、大于等于 大于、等于、不等于。
与、或、异或, not	BOOL	布尔运算符
逻辑与、逻辑或、逻辑异或 or, lnot	INT, 逻辑, CHAR	按位逻辑运算符
@ a @ b @ c ..	逻辑	位向量连接运算符
~ V~B	INT	$\log_B(V)$（仅允许用于常量定义）
逻辑左移、逻辑右移 *obj* lsx *n*	整数、逻辑	静态和动态（非恒定移位参数 *n*）移位操作。左侧运算对象必须是对象（寄存器、变量、信号）。
to_逻辑 to_int to_char to_bool	INT, 布尔型、逻辑 , CHAR	类型转换（仅适用于单个对象）

表达式中的所有操作数必须属于同一类型。显式类型转换只能用于转换原生对象（寄存器、变量、信号）的数据类型。

将表达式的值赋给数据对象的操作见**定义 17**。

定义 *17* 将一个值（由表达式计算得出）赋给数据对象（寄存器、变量、信号、队列、通道）。

左侧 <- 右侧;
x <- 表达式;

函数应用（仅限非递归）由函数名和参数列表提供，参数列表可以为空（**定义 18**）。包含表达式的参数会在函数应用之前求值。函数应用可以嵌入表达式中。

定义 *18* 带参数和不带参数的函数应用（*f*）与过程执行（*p*）

目标 <- *f*(参数*1*,参数*2*,...); 目标 <- *f*();表达式(*f*(...))*p*(参数*1*,参数*2*,...);*p*();

数据对象

顺序数据处理需要存储对象（内存）。数据存储使用寄存器和变量。寄存器是可映射到 CREW 访问行为的单个内存对象，而变量则受限于具有 EREW 访问行为的内存块（RAM）。此外，还有一种无需存储、用于互连硬件组件的信号对象。数据对象被赋予特定的数据类型。

数据对象具有局部和全局可见性（作用域）。局部对象只能由一个进程访问，而全局对象则可由多个进程并发访问。这些全局对象的并发访问通过原子保护动作和每个对象对应的调度器进行仲裁和串行化。

结构类型的存储对象（仅涉及寄存器和信号）会拆分为一组彼此独立的存储对象，每个对象都与一个结构元素相关联。数据对象定义汇总于**表 13**。

表 *13* 数据对象定义（类型：结构类型）

语句	数据类型DT	描述
寄存器 R: DT; 寄存器 R：DT [N]; 寄存器 R,S,...：DT;	INT, 逻辑, 字符, 布尔型, TYPE	创建一个数据类型为 DT 的寄存器存储对象 R，并执行 op- 数据宽度 N（位）。
块 B; 变量 V：DT 位于 B 中; 变量 V：DT [N] ...	INT, 逻辑, 字符, 布尔型, TYPE	创建一个数据类型为 DT 的变量存储对象 V，并 可选择指定数据宽度 N（位）。变量绑定在 RAM 块中，必须显式指定该块。RAM 数据单元 的位宽和单元数量在编译 时确定。
信号 S: DT; 信号 S: DT [N]; 信号 S,T,...:DT;	INT, 逻辑, 字符, 布尔型, TYPE	创建一个数据类型为 DT 的互连信号对象 S 以及可选的数据宽度 N（位）。
常量 C: DT := V; 常量 C: DT [N] := V; 常量 NC: value := V;	INT, 逻辑, 字符, 布尔型, 值	创建一个值为 V、数据类型为 DT 的常量对象 C （或通用值）以及可选的数据宽度 N（位）。

IPC数据对象

有一些预定义的数据对象用于实现同步进程间通信（参见表格 **表 14** 和 **表 15**）：队列和通道。虽然队列和通道是抽象对象，但可以直接用于表达式（右侧）和赋值语句（左侧）。队列具有缓冲存储深度（大小），可通过参数 深度 指定。通道的深度始终为一，且只能通过实现握手来实现带缓冲或无缓冲。

结构类型的队列或通道会拆分为一组相互耦合的队列或通道，每个对象都与一个结构元素相关联。

表 *14 IPC* 数据对象定义（类型：结构类型）

语句	数据类型DT	描述
队列 Q：DT; 队列 Q：DT [N]; 队列 Q、R 等：DT 其中 P=V 以及.....;	int, 逻辑, 字符, 布尔型, TYPE	创建一个数据类型DT的队列对象Q，可选 参数设置。
通道Q：DT; 通道Q：DT [N]; 通道 Q、R、..: DT，含 P=V 以及 ..;	int, 逻辑, 字符, 布尔型, TYPE	创建一个数据类型为 DT 的通道对象 Q，可选 参数设置。

表 *15 IPC* 数据对象参数

参数	值	描述
深度	1..256 (∞)	队列缓冲区大小。通道深度始终为 1。

抽象对象

抽象对象只能通过一组已定义的方法进行修改。内置对象（队列、通道）和用户抽象对象的行为实现及方法接口均由 EMI ADTO编程语言定义，如**表 16**所示。

表 *16* 抽象对象定义

语句	Type	描述
对象 O：AT; 对象 O：AT 使用 P=V 以及 P=V .....;	AT: 互斥锁, 信号量, 事件 ， 定时器， ...	创建一个类型为 AO 的抽象对象 O，可选 参数列表（参数 P 和值 V）。

数组

数组可以由具有特定数组元素数据类型的寄存器、变量和信号创建。寄存器数组的每个单元仍具有 CREW 访问行为，不过动态元素选择器需要可寻址数组选择器（每个访问进程一个）。

加法 可以使用结构和抽象对象类型创建数组，如**表 1**所示 **7**.

表 *17* 数组定义（类型：结构类型）

语句	数据类型DT	描述
数组 A: 寄存器 [I] 属于 DT; 数组 A: 寄存器 [I,J,K] 属于 DT ;	int, 逻辑, 字符, 布尔型, TYPE	创建一个包含 I 个数据元素的寄存器存储数组 A 类型 DT，以及可选的数据宽度 N（位）。多维数组可以通过扩展大小参数来创建参数列表 I,J,K。
数组 A：变量 [I] 类型为 DT; 块 B; 数组 A：变量 [I] 类型为 DT，位于 B 中 ;	int, 逻辑, 字符, 布尔型, TYPE	创建一个包含 I 个数据元素的变量存储数组 A 类型 DT 和可选的数据宽度 N（位），并限定在一个 RAM 块中。指定特定块是可选的。
数组 A：sig [I] 类型为 DT 的;	int, 逻辑, 字符, 布尔型, TYPE	创建一个包含 I 个数据类型为 DT 的元素的信号数组 A 以及可选的数据宽度 N（位）。
数组 A：对象 AT [I]; 数组 A：对象 AT [I] ，其中 P=V 以及 P=V.....;	AT: 队列, 通道, ...	创建一个包含 I 个元素的抽象对象数组 A，这些元素的对象类型为 AT，并带有可选参数列表（参数 P 及其值 V）。
A. [i] <- A. [j]; A. [i,j,k] <- A. [x、y、z];	AT	在赋值左侧（写入）和右侧（读取）以及表达式中访问数组元素的方法：使用点括号选择器。

结构

有数据结构类型和组件结构类型（记录）。数据结构用于将语义上相关联的单个数据元素绑定到命名类型。组件结构定义硬件组件接口（端口信号）。数据结构类型可应用于所有数据对象定义，包括队列和通道，如**表 18**所示。

表*18* 结构类型定义及结构类型对象的定义。

语句	数据类型DT	描述
类型 T: { e1 : DT; e2 : DT; .. };	int, 逻辑, 字符, 布尔型	定义包含元素 e1、e2 等的数据结构类型 T，这些元素的类型为指定的数据类型。可以实例化此类型的寄存器、变量和信号对只能实例化此类型的对象。
类型 T: { e1 : N1; e2: N2; e3: N3 到 N4; e4: N5 递减至 N6; .. };	逻辑	定义一个位字段数据结构类型 T，其元素为 e1、e2、……，具有指定的数据宽度（数据类型为逻辑）N1、N2、……此类型的寄存器、变量和信号对象可以实例化。
类型 Tc: { e1 : DIR DT; e2 : DIR DT; .. }; DIR = {输入, 输出，双向输入输出}	int, 逻辑, 字符, 布尔型	定义一种组件结构接口类型 Tc，包含元素 e1、e2、……，其数据类型为指定类型，且数据/信号流向。
reg R: T; var V: T; . . .	TYPE	定义一个数据对象（寄存器 R、变量 V、信号 S），其类型为用户定义的结构类型 T。
组件 C: Tc;	TYPEc	实例化一个具有接口结构的组件对象，类型 Tc。
寄存器 R: T; R.e1 <- .. X <- R.e2 ..	TYPE	使用以下方式访问结构元素和位字段。 点选择器。

枚举

表 *19* 枚举类型定义

语句	对象类型	描述

类型 e : {
S1;
S2;
. .
};
 	
类型 e : {..
} with code=C语言;
 	
C语言={一元,二进制,格雷码, 名称}
	寄存器, SIGNAL, 变量	用于定义命名常量。
		枚举元素的值会在编译时计算。第一个枚举元素的索引为 1，第二个为 2，依此类推。最终的编码风格可以通过参数 code 设置。

进程

进程是主要的执行单元。各个进程彼此独立并发运行，但进程语句按顺序执行。进程具有由数据和一些抽象对象组成的本地进程空间。通过使用具有受保护的原子访问权限的全局对象来执行进程间通信和同步。并发访问由调度器进行串行化。

表*2* 进程定义与进程控制。

语句	描述
进程 P: 开始 　定义 　语句 end; 进程　　P: ... 以 P=V 和 ..; 结束	进程的定义，可包含对象定义（可选）以及一系列语句。进程可以通过应用一个参数列表（参数 P 和值 V）进行参数化。 除了主进程之外，每个进程都必须显式启动。
数组 P: 进程 [N] of 开始 　定义 　语句 end; # ≡ 进程编号	进程数组的定义，包含对象定义（可选）和一个语句序列。进程可通过 ap- 进行参数化，传递参数列表（参数 P 和值 V）。 N 是数组的大小（要创建的进程数）。
P.call(); P.start(); P.stop();	进程控制语句（进程方法）。 进程调用是一种同步操作。如果进程 P1 调用进程 P2，进程 P1 会被阻塞，直到进程 P2 完成其工作（到达结束状态）。 进程启动和停止不会阻塞正在执行的进程。 这是一种异步操作。

函数

函数通过进程实现，并带有附加参数（全局寄存器），这些参数在函数应用时以一个值进行初始化。在表达式中应用函数或执行过程时，会将（可选的）参数传递给这些参数，并启动与函数关联的进程。调用进程会被阻塞，直到函数进程结束（到达结束状态）。返回值会传回调用进程（如果是函数）。

表2 函数定义和函数应用

语句	描述
函数 f(p1:DT,..) 返回 (pn:DT): 开始 定义 语句 end; .. 以 P=V 和 .. 结束; .. 以 inline; 结束	定义一个带有（可选）形式参数 p1、p2、.. 的函数 以及一个返回参数 pn。这里没有返回语句。返回参数的值 必须在函数体内修改 。 函数和过程可以参数化。内联参数- eter 将每个函数应用或过程执行替换为 相应的语句序列。局部数据对象是共享的。 共享的。
函数 p(p1:DT,..): 开始 定义 语句 end;	不带返回参数的过程定义（可选）形式参数 p1, p2,..。没有返回语句。
P(*v1*,*v2*,...); P(); X <- f(*v1*,*v2*,..); X <- f();	带参数和不带参数的函数和过程调用。

块

进程或函数语句可以通过块语句进行分组。块可以进行参数化，例如采用不同的调度、分配或优化行为。

表*2* 语句块

语句	描述
开始 语句1; 语句2; .. end;	顺序语句组合。
开始 语句1; 语句2; .. 以 P=V 和 .. 结束;	带有附加行为的顺序语句组合，或 综合参数化（参数 P 的值为 V）。
开始 语句1; 语句2; .. 以绑定[=真]结束; ⇔ 语句1 ， 语句2，...;	并行数据语句组合。等同于以逗号分隔的列表 （仅数据赋值）语句。只能有一个数据语句 对全局对象执行受保护访问（读取全局寄存器 不受保护）。

分支

有多种分支语句可供使用。如**表24**所示，它们会根据值将程序流程转至另一个语句或一组语句。分支可以出现在模块顶层，也可以出现在块语句（进程、函数等）中。

表*2 23* 分支语句

语句	Kind	描述
如果 表达式 **则** 语句*1*; **如果** 表达式 **则** 语句*1* **否则** 语句*0*;	布尔分支	根据布尔表达式 表达式 的结果， 会执行分支，进入 语句*1*（表达式=为真），或者可选地 进入 语句*0*（表达式=为假）。如果语句不止一条， 则需要使用块。
匹配 表达式 **与** **开始** **当** *v1*: 语句*1*; **当** v1,v2,..: stmt123; **当** v1 to v2: stmtv1tov2; **当** v1 downto v2: stmtv2tov1; **其他**: *stmte*; **end**;	多值匹配- 分支	将不同的常量值与以下内容的结果进行匹配 表达式 表达式，并将相应的语句 se- 匹配成功时选中。默认 others 情况 （匹配所有其他值）是可选的。 值列表 v1,v2,.. 指定了不同的备选项 匹配同一种情况。
异常 e1,..; **try** 语句; **with** **开始** **当** *e1*: 语句*1*; **当** e1 ,e2,..: stmt123; **其他**: *stmte*; **end**;	异常处理程序 分支	在语句中引发的异常会被匹配，并在匹配成功时选择相应的语 匹配成功后将选择相应的语句。 默认处理程序 else（匹配所有其他异常） 是可选的。

循环

有多种循环语句可供使用。只要布尔条件成立，每个循环都会重复执行循环体。计数循环会遍历由范围指定的值列表，如**表24**所示。

表 *24* 循环语句

语句	Kind	描述
对于 i = a 到 |递减至 b **执行开始** 语句 **end;** 结束并展开; 以 P=V ..; 结束	计数循环	For循环会针对迭代器列表中的每个元素执行循环体语句，即值范围内的元素，其中包括边界。循环迭代器变量 *i* 保存当前迭代值。循环可以参数化。展开参数将循环体复制（b-a）+1 次，并将循环迭代器替换为当前迭代值。
while 表达式 **do** **开始** 语句 **end;**	条件循环	while循环会一直执行循环体，直到布尔表达式 表达式 为真，While循环就会执行循环体。每次循环迭代之前都会对布尔表达式进行测试。
始终执行 **开始** 语句 **end;**	无条件 Loop	此循环永不终止（除非引发异常-常）。
等待 周期; **等待** 时间; **等待** 条件; **等待** 条件 **带有** 语句; **等待** 条件 **带有** 语句*1*; **否则** 语句*0*;	延迟/阻塞 Loop	等待语句会阻塞执行，直到经过一段时间间隔或条件为真。只要条件为假，就可以选择应用信号赋值（语句1）。可选的默认语句（语句0）会在剩余时间内执行。

异常

异常用于退出控制环境，例如函数、循环或分支。异常会在控制环境中逐层传播，直到异常处理程序（语句）捕获异常，如**表 25**所示。否则会出现未捕获异常故障。

在嵌套控制环境（嵌套分支/循环或函数调用）中引发的异常会传递到上一级环境，直至到达处理程序环境。异常处理程序环境也可以嵌套。特定处理程序未捕获的异常（没有默认分支）会被重新引发。

表 *25* 异常处理程序语句

语句	Kind	描述
异常 ex1,..;	Type	命名异常信号的定义。
try 语句; **with** **开始** **当** *e1*: 语句*1*; **当** e1 ,e2,..: 语句123; **其他**: 语句*e*; **end**;	异常处理程序 分支	在语句中引发的异常会进行匹配，并在匹配成功时选择相应的 匹配成功时会选择相应的语句。 默认处理程序 else（匹配所有其他异常） 是可选的。
引发 ex;	引发异常	引发异常信号。

进程间通信对象

进程间通信通过使用全局对象来实现。虽然全局数据存储对象由互斥调度器保护，但它们不适用于进程同步。还有一些抽象的进程间通信对象，可提供不同类型的同步，汇总见**表26**。

表*26* 同步对象及其方法

对象	方法	描述
队列 open Core; 队列 q: DT ?with 深度=N 和调度器 ="先进先出" ;	*q.write*: q <- 表达式; *q.read*: x <- q; q.unlock()	队列是一个最多可容纳 N 个元素的缓冲区，并具有同步- 以先进先出顺序进行同步访问。读操作会阻塞，直至 至少有一个数据元素可用。写入 操作会在队列已满时阻塞。解锁操作 会解除所有阻塞进程的阻塞。数据类型可以是 核心类型或记录结构类型。
通道 打开 Core; 通道 c: DT ?带 模型=M; M={带缓冲, 无缓冲}	*c.*写入: c <- 表达式; *c.*读取: x <- c; c.解锁()	通道是一个容纳一元（带缓冲）或零个元素的缓冲区 采用同步访问。读操作会阻塞，直到 至少有一个数据元素可用，或有一个写操 作正在等待。写操作会一直阻塞，直到 缓冲区为空，或发生读操作。解锁 操作会解除所有阻塞进程的阻塞。
互斥锁 打开 互斥锁; 对象 o：互斥锁 ？带有 调度=S; S={fifo}	o.lock() o.unlock() o.init()	互斥锁用于对共享 资源进行互斥访问。互斥锁要么已加锁，要么已解锁。只有一个 进程可以通过加锁操作获得锁。 解锁操作会释放锁。对一个 al- 加锁 已就绪且锁定的互斥锁会阻塞进程。初始化是通 过使用初始化操作来完成的。
信号量 打开信号量; 对象 o: 信号量 ？使用 调度器=S 和 深度=N 和 初始化=V; S={"fifo"}	o.down() o.up() o.unlock() o.init(V)	信号量实现了一种异步计数器，其 值范围为 [0..深度-1]，用于生产者-消费者应 用。计数器的值永远不会变成负数。 down 操作会递减计数器。如果计数器已 经为零，则进程会被阻塞。up 操作会递增 计数器。必须使用 init 操作进行初始化。
事件 打开 Event; 对象 o：事件 ?with 锁存=B;	o.await() o.唤醒() o.init()	事件实现了基于信号的简单进程同 步。多个进程可以通过 使用等待操作等待事件。另一个进程可以唤醒 这些阻塞进程，并可通过 唤醒操作同时唤醒它们。锁存事件可防止竞态条 件，即使等待请求在唤醒操作之后到达也不例外。 tion。必须使用初始化操作进行初始化。
屏障 打开屏障; 对象 o: 屏障;	o.await() o.init()	屏障通过基于信号的简单机制，实现一组进程之间的同步， 该组包含 N 个进程。进程组 可以通过使用.....等待屏障事件（进入屏障）。 等待操作等待屏障事件（进入屏障）。如果第 N 个进程进入屏障，所有 进程会立即同时解除阻塞。 组大小 N 在编译时确定。必须通过初始化操作完成初始 需要通过使用初始化操作来完成tion。

对象	方法	描述
定时器 打开 定时器; 对象 o: 定时器 ?with mode=M; M={0,1}	o.await() o.时间(T) o.启动() o.停止() o.初始化() o.sig _操作(S,L1,L0)	定时器是一种自同步事件对象。进程可以 使用等待操作等待定时器事件。经过 必须通过时间操作设置的时间间隔后, 等待中的进程会被唤醒。定时器 必须启动，并且可以通过启动 以及停止操作。在模式=0 下，定时器会持续运行, 在模式=1 下则只运行一元时间。信 _动作操 作会将信号 S 连接到定时器，并返回当前的 状态（L0：非活动，L1：活动）。

模块

通常，主入口源文件会定义一个主模块。模块由模块行为描述（包括进程、对象定义和顶层语句）以及模块接口（隐式定义）组成。模块接口定义了组件端口接口，至少包括时钟和复位端口信号，如**表 27**所示。还可以通过导出对象（寄存器、信号）和硬件组件接口来添加其他端口组件。

还可以在结构组件级定义其他复合模块。每个复合模块由实现定义（必须导入的主模块）、从该主模块实例化的行为组件，以及用于连接所有已实例化模块组件的互连组件构成。

表 27 用户定义模块

语句	描述
定义 声明 进程 顶层语句	声明并创建一个顶层行为模块*M*（来自源文件 *m*.cp）。
模块 *MC* 开始 导入 组件 连接 映射 end;	声明并创建一个新的复合模块 *MC*，该模块源自一个行为 包含导入、组件实例化、互连和映射的模块 部分。
导入 *M*; **组件** *C1*、*C2*，..: *M*;	导入行为模块*M*（顶层主模块）以及模块 组件实例化（复制模块组件）。

类型 *ic*: {
端口 *S*: *dir* 类型;
...
};
组件 *IC* : *ic* :=
{
C1.S1 , ...
};
IC.S1 << IC.S2;
	互连组件类型的定义和互连组件的实例化
	采用模块组件的默认信号映射进行互连
	组件端口信号。
	最后，可以通过附加的语句连接互连组件信号
	映射语句。