# Fast Model简介

Fastmodel可以自己在ARM官网下载软件，

<https://developer.arm.com/Tools%20and%20Software/Fast%20Models>  
Release Name: FM000A-r11p27-19rel0

Release Quality: REL

Available Date: 2024年9月16日

安装和使用需要有ARM的license

## 快速模型库(Fast Models Portfolio)

模型库包括高性能的ARM CPU和ARM CoreLink组件模型，以及可视化组件和常用部件，例如计时器(timer)和串口(UART)。运用这些组件模型可以构建虚拟平台模型。处理器模型的开发是ARM处理器开发方法论的一个组成部分，保证了处理器模型能够先于硬件实现面世，同时保证了功能的完善。模型库会协同ARM IP发布定期更新最新的ARM IP 模型。模型库包括：

* Cortex-A系列模型，Cortex-R系列模型，Cortex-M系列模型，ARM11模型和ARM9模型
* ARM CoreLink IP和组件模型用来构建完整的虚拟平台
* 可直接运行的ARM开发板固定虚拟平台(Fixed Virtual Platform, FVP)模型
* 架构信封模型(Architecture Envelope Model, AEM) - AEMv7-A和AEMv8

## 虚拟平台构建

为了提供完整的解决方案，产品提供了相应的工具和接口用来构建虚拟平台。这包括系统画布(System Canvas)作为平台创建环境和系统生成器(System Generator)作为仿真模型生成系统。

图示

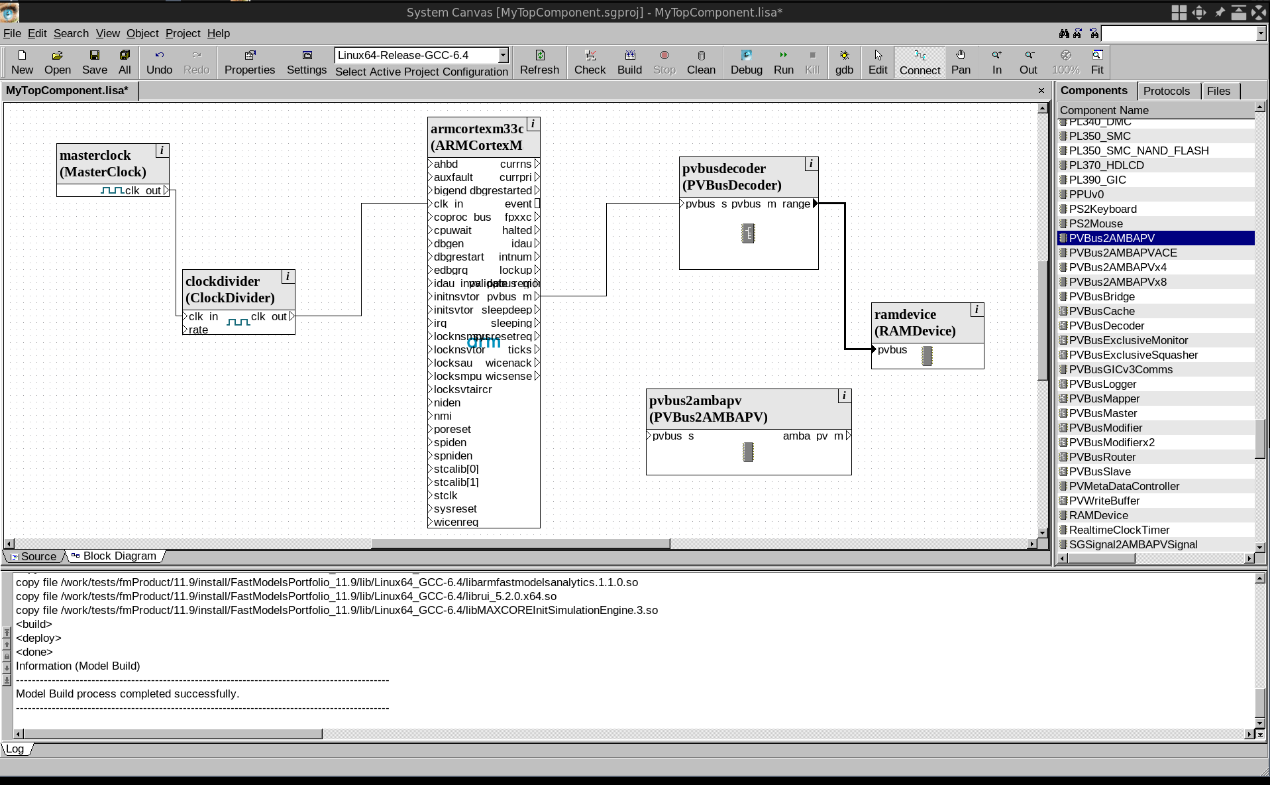
描述已自动生成

### 系统画布(System Canvas)

系统画布通过提供强大易用的图形用户界面缩短了系统设计时间。通过它，用户可以创建，编辑和管理平台设计。系统画布集成了对系统设计有用处的各种功能，从用来创建设计的LISA+编辑器，到最终虚拟平台或子系统的部署。因为简单易用的模块图设计，即使没有编程经验也可以基于快速模型库设计，配置和编译虚拟平台。

**系统画布包括：**

* 用来创建平台的简单易用的模块图编辑器
* 带语法高亮的LISA+文本编辑器
* 可以被LISA+集成脚本调用的C/C++库文件
* 运行在主机级别的调试工具用来验证调试虚拟平台



ARM社区博客链接：

* <https://community.arm.com/arm-community-blogs/b/tools-software-ides-blog/posts/getting-started-with-fast-models-and-rtx-1-of-2>

### 系统生成器(System Generator)

在系统画布里创建的平台通过系统生成器翻译并编译成仿真模型。用LISA+写成的组件和系统先被翻译成C++源代码，之后用主机上的标准软件(GCC，微软Visual Studio)编译出仿真模型。系统生成器可生成三种目标类型：

* 可独立运行的自包含虚拟平台
* 动态链接库形态的自包含虚拟平台
* 符合SystemC规范并支持TLM 2.0总线事务接口的可扩展SystemC子系统模块

系统生成器自动将和标准调试器对接的调试接口和追踪(trace)接口嵌入到模型当中，极大的增强了可用性和易用性。

快速模型的系统生成器用于生成虚拟平台(FVP)或将子系统导出为 SystemC(EVS)，它使用与 TLM-2.0 兼容的接口来实现组件间的通信。通过为每个组件提供各种参数以及使用客户组件来扩展模型组合，可以对系统建模，以便系统直接与芯片上的目标系统进行匹配。

### 模型导出

ARM和各大EDA工具厂商进行了合作来验证模型与工具的交互性，并将导出的虚拟子系统(EVS)集成进各厂商基于SystemC的设计和仿真环境当中，极大的扩展了模型的使用范围。现阶段合作厂商包括Cadence、Carbon、Mentor Graphics和Synopsys。

### 平台调试

虚拟平台模型可以使用任何支持CADI(Component Architecture Debug Interface)接口的调试器来调试。这包括ARM Development Studio 5(DS-5) 工具链和其他第三方商业调试解决方案。快速模型附带提供模型调试器(Model Debugger)为用户提供直接的调试解决方案。

**模型调试器**(Model Debugger)可以直接窥探虚拟平台里的任何组件，并可反复重新连接到模型中的不同目标组件。模型调试器主要针对在模型上的C/C++应用程序开发，包括众多常用功能：

* 标准上层源代码调试例如step、step in、step out、step over以及指令断点
* 反汇编，寄存器以及内存视窗
* 函数调用堆栈窗口，变量监视窗口，源代码窗口
* 断点管理
* 支持多核调试

# Fast Model IP

Fast model本身可以看成是一个tool kit，里面包含很多ARM的IP model作为components

我们的IP会包含这2个success kits

图片包含 日历

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

官网写的包含ARM所有IP的tools和models，fast model的reference guide里有详细的列表列出fast model包含的components

**NOTES:**

CMN-700的model需要加载从ARM Socrates生成的yml文件才生效配置，mesh网络最大支持12x12

ARM的自己的UART IP-PL011-UART有现成的模型源码，换synopsys IP替换寄存器和功能即可。

其他的非ARM的外设IP model需要自己写模型或者能在网上找到现成的模型做封装。

自研IP需要自己写模型，ESL平台已有的可以封装集成

# Fastmodel

-a 参数

-C 参数

--data 参数可以提前加载数据到指定位置

--data [INST=]FILE@[MEMSPACE:]ADDRESS



--dump 参数

# Fast Model导出扩展特性

ARM fastmodel解决方案：

导出EVS子系统，EVS + Custom SC-model

将Fast Model的子系统导出生成SystemC的模型子系统模块

图示

描述已自动生成

编写Custom IP的功能模型，通过AMBA PV总线与Fast Model导出的子系统模块进行连接

图示

描述已自动生成

## AMBA-PV协议扩展接口

基于TLM 2.0的扩展的满足AMBA总线协议的接口，作为bridge，连接fastmodel导出子系统EVS模型和外部的systemc模型

图示, 示意图

描述已自动生成

上图的案例是用fast model构建一个简单的cortex-M33的子系统

作为EVS的导出子模型有2个对外暴露的接口用来连接自定义的systemc外设model

* [AMBAPV Master port](https://developer.arm.com/docs/dui0847/c/data-structures/amba_pvamba_pv_master_socket-buswidth)–
  + **amba\_pv::amba\_pv\_master\_socket<64>** m\_port\_to\_external\_counter
  + This port can be connected to a AMBA-PV Slave port exposed by a SystemC Module.
* [AMBAPV Signal Slave export](https://developer.arm.com/docs/dui0847/c/data-structures/amba_pvsignal_slave_export-state) –
  + **amba\_pv::signal\_slave\_export<bool>** slave\_counter\_irq\_in
  + This export is connected to a amba\_pv::signal\_master\_port<bool> present in an external SystemC Module.

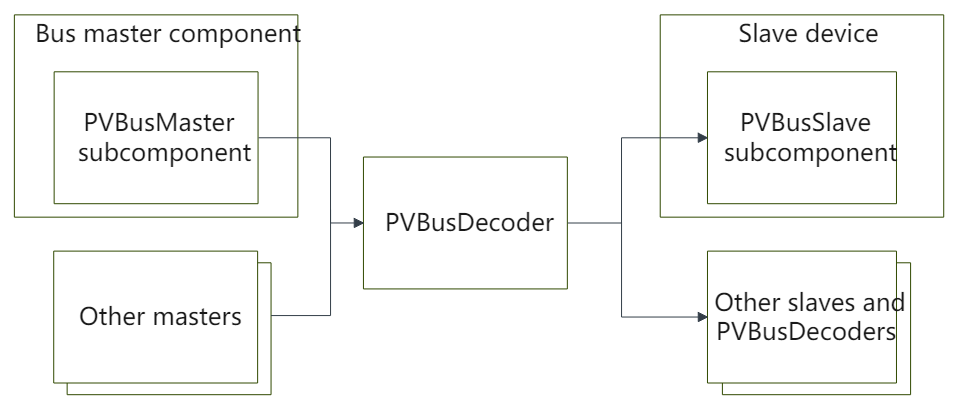
参考文献ARM社区博客：基于cortex-m33构建的platform model+systemc IP model互联

* https://community.arm.com/arm-community-blogs/b/tools-software-ides-blog/posts/cortex-m33-systemc-subsystem-using-fast-models

Github：fastmodel的一些案例

* https://github.com/ARM-software/Tool-Solutions

# PVBus



PVBus 是用于在 Fast Models 中对所有类似内存的总线进行建模的总线协议。PVBus 是一种内部协议。PVBus 组件抽象了内部细节，是 PVBus API 的接口。PVBus 组件提供总线主设备和从设备之间功能准确的通信。它们不是特定硬件的软件实现，而是软件模型环境所需的抽象组件。

PVBus是PV model的通信总线，用于模拟组件直接的总线通信。

一个master原则上只能和一个slave互联

PVBusDecoder组件接入bus系统后，可以允许一个master和多个slave互联

AMBAPV是支持AMBA协议族的总线接口。

# Bridge组件

## AMBAPV2PVBus

AMBA-PV 到 PVBus协议转换

表格

描述已自动生成

表格

描述已自动生成

* + PVBus 不支持设置了 byte\_enable 的事务（用 AXI 术语来说，即strobing事务）。此bridge组件会拒绝这些事务。
  + 此组件的变体也具有多个输入和输出端口

限制：

* + PVBus 和 AMBA-PV 之间的fast model bridge可以传输内存标记扩展 (Memory Tagging Extension, MTE) 操作（tag stores, tag loads, and tag-checked loads and stores）。
  + 这些操作是不透明传输的，因此端点必须使用 PVBus。这意味着您无法在自己的 TLM 组件中处理这些操作

## PVBus2AMBAPV

PVBus 到 AMBA-PV协议转换

表格

描述已自动生成

表格

描述已自动生成

LISA 中的 AMBAPV 协议定义在AMBAPVProtocol.lisa 指定 64 位总线宽度，因此 PVBus2AMBAPV bridge也处理 64 位总线宽度。

如果需要连接到使用总线宽度较小或较大的总线接口的组件，建议的方法是分别插入downsizer或upsizer。

或者，您可以定义具有所需位宽的新总线协议，例如 AMBAPV32，并更新相应的桥接器以在 AMBA-PV 端口上使用新协议：

master port<AMBAPV32> amba\_pv\_m

限制：

* + PVBus 和 AMBA-PV 之间的fast model bridge可以传输内存标记扩展 (Memory Tagging Extension, MTE) 操作（tag stores, tag loads, and tag-checked loads and stores）。
  + 这些操作是不透明传输的，因此端点必须使用 PVBus。这意味着您无法在自己的 TLM 组件中处理这些操作

图形用户界面, 文本, 应用程序

描述已自动生成

# BUS组件

## PVBusCache

PVBusCache管理cache-line数据并且支持事务转发(forwarding of transaction)

表格

描述已自动生成

## PVBusDecoder

PVBusDecoder允许总线事务被路由到多个slave中的一个，基于传输事务给出的地址。

图形用户界面, 应用程序, 表格

中度可信度描述已自动生成

每一个slave连接都有一个对应的地址范围映射到pvbus\_m\_range，

LISA+的语法：

decoder.pvbus\_m\_range[start..end] = slave.pvbus

start地址和end地址必须4KB对齐。可以位slave指定一个地址范围，decoder会将地址重映射到一个合适的范围。Slave的默认地址范围**[0-(*sizeOfMasterRange* – 1)]**

## PVBusModifier

允许事务任意地重映射(remapped)

表格

描述已自动生成

## PVBusMapper

允许事务任意地重映射(remapped)

表格

描述已自动生成

PVBusMapper组件行为和PVBusModifier相似，额外增加：

* 多个下游端口(master port)
* 事务可以路由到任何一个下游端口
* 它允许任意重新映射事务地址和属性

For an example of how to use PVBusMapper, see $PVLIB\_HOME/examples/LISAPlus/

RemappingWithPVBusMapper/.

需要control回调函数来确定对应的地址段映射到哪个pvbus\_m[x]

## PVBusRouter

允许事务任意地重映射(remapped)

表格

描述已自动生成

## PVBusMaster

PVBusMaster的子组件允许device产生PVBus事务

图形用户界面, 应用程序

描述已自动生成

如果希望某个组件充当总线主控器，请实例化 PVBusMaster 子组件，然后使用其control port创建一个或多个事务生成器以在总线上生成事务。

$PVLIB\_HOME/examples/LISA/BusComponents/DmaTransfer.lisa is an example component that shows this.

## PVBusSlave

PVBusSlave负责处理输入的事务，以及作为 RAM/ROM/设备内存，处理对设备地址空间映射区域的事务。

图片包含 图形用户界面

描述已自动生成

PVBusSlave的组件作为一个bus slave必须满足：

* + 提供一个PVBus slave port
  + 例化一个PVBusSlave 子组件，并把device覆盖的地址范围的size参数配置好
  + 将slave端口连接到 PVBusSlave上的 pvbus\_s 端口。

默认情况下，PVBusSlave 将所有事务转换为设备端口上的 read() 和 write() 请求。

Control port可以配置device的PVBusSlave。定义device上不同region的地址空间的行为。以4KB为粒度，单独为region配置read和write。这样就允许设置成memory-like、device-like、abort、ignore access address regions。

到memory-like regions的事务在PVBusSlave子组件内部进行处理，abort和ignore regions也会被PVBusSlave进行处理。

到device-like regions的事务会被发往device port。如果有被配置为device-like的region的device必须实现device port的read()和write()方法。

PVBusSlave组件不会显著影响PV系统的性能。然而，PVBusSlave 组件的正确实现对于整个PV系统的性能至关重要。例如，将request路由到 PVDevice 端口比让 PVBusSlave 组件内部处理request要慢。Arm 建议尽可能使用对memory-like regions的内部支持。

## PVMemoryProtectionEngine

使用独立密钥对每个加密上下文的内存事务进行加密，以防止不匹配的访问。

表格

描述已自动生成

PVMemoryProtectionEngine是Memory Protection Engine(MPE)组件的一种简化实现。

Memory Protection Engine(MPE)组件在Arm Realm Management Extension (RME) System Architecture有描述

PVMemoryProtectionEngine支持以下的特征：

* Memory encryption，内存加密
* 内存中每4KB的page会基于一个encryption key进行加密。每一个物理地址空间(Physical Address space, PAS)有单独的encryption key
* 2个或以上的encryption key可以是相同的值
* 每个PAS可配置的encryption key
* 可配置的加密block size
* 可配置的corruption strategy。可以控制加密块内访问未写入的内存内容的行为
* 加密/解密算法是数据与相应加密密钥的简单异或XOR。
* 下游内存始终以纯文本形式存储，允许调试器查看数据

例如，如果一个block现在被ns-PAS加密了，然后被rl-PAS写入了1Byte，如果block\_size\_in\_bytes设置为4KB，则这4KB page剩下的data是corrupted，甚至你通过ns-PAS去读取另外1byte的数据，都不能得到原始的数据。

此组件的主要用例是识别软件错误编程，即通过多个 PAS 访问同一物理地址。启用 PVMemoryProtectionEngine 后，当使用与写入内存中该页面的原始 PAS 不同的 PAS 访问数据时，PE 会看到加密或损坏的数据。

如果存储是共享的，则 PVMemoryProtectionEngine 组件应在物理别名点 (PoPA) 的平台中连接，否则应在 PAS 子集的每个特定存储之前连接。

PVMemoryProtectionEngine 启用后会产生运行时成本。通常，仅在调试和验证 Realm Management Monitor (RMM) 软件时才需要它。如果 RMM 软件正确，则使用错误密钥加密的内存内容将不可见。

PVMemoryProtectionEngine 不会加密或破坏 MTE 的标签数据，但将来会支持此功能。

## PVWriteBuffer

PVWriteBuffer子组件buffers PVBus事务

表格

描述已自动生成

## SimplePVBusMaster

按照配置属性和地址生成PVTransaction的组件

表格

描述已自动生成

表格

描述已自动生成

## TZSwitch

允许分开路由TrustZone secure/normal bus信号

表格

描述已自动生成

表格

描述已自动生成

TZSwitch组件允许您根据事务的 TrustZone® 安全状态控制事务路由。

默认的行为是secure事务发送到pvbus\_port\_a，normal事务发送到pvbus\_port\_b。

仅当路由决策不频繁改变（例如作为内存重映射的一部分）时，才必须使用 TZSwitches。

此组件的secure和normal参数控制初始化状态：

文本

描述已自动生成

初始配置使用的数字与运行时控制路由使用的枚举常量不同

## TZFilterUnit

# AMBA-PV协议

The AMBA-PV components and protocols permit you to model a platform that interfaces with an Arm® AMBA®-based system.

The protocols are:

* **AMBAPV**
  + Models the AMBA® protocols AXI4, AXI3, AHB and APB.
* **AMBAPVACE**
  + Models the AMBA® ACE and DVM protocols.
* **AMBAPVSignal**
  + Models interrupts.
* **AMBAPVSignalState**
  + Transfers and receives signal states.
* **AMBAPVValue**
  + Models propagation of 32-bit integer values between components.
* **AMBAPVValue64**
  + Models propagation of 64-bit integer values between components.
* **AMBAPVValueState**
  + Permits a master to retrieve the current value from a slave, using 32-bit integer values.
* **AMBAPVValueState64**
  + Permits a master to retrieve the current value from a slave, using 64-bit integer values.

There are ready-to-use components that provide you with conversions between protocols.

Related information

AMBA-PV Extensions to TLM User Guide

## AMBAPV协议

The AMBAPV protocol defines behaviors for single read and single write transactions. This covers Arm® **AMBA® AXI5, AXI4, AXI3, AHB, and APB** bus protocol families, at the PV level.

In addition, the AMBAPV protocol supports AMBA® protocol additional control information:

• Protection units.

• Exclusive access and locked access mechanisms.

• System-level caches.

• Atomic accesses, including exclusive accesses, locked accesses, and atomic transactions

## AMBAPVACE协议

This protocol defines behaviors for bus transactions. This covers Arm® **AMBA® ACE and DVM** bus protocol families, all at the PV level

In addition, this protocol provides support for AMBA® protocol additional extension information:

• Secure and privileged accesses.

• Atomic accesses.

• System-level caching and buffering control.

• Cache coherency transactions (ACE-Lite).

• Bi-directional cache coherency transactions (ACE).

• Distributed virtual memory transactions (DVM).

# NI-TOWER

图示

描述已自动生成

图形用户界面, 应用程序

描述已自动生成

图示

描述已自动生成

## Interface

Following are the requester side interfaces present in the system control NI-Tower:

1. rse\_main\_axis: Request from RSE.
2. scp\_axis: Request from SCP.
3. mcp\_axis: Request from MCP.
4. app\_axis: Request from AP.
5. lcp\_axis: Request from LCP targeting LCP/SCP
6. rse\_scp\_axis: RSE/SCP access which is filtered out by NIC400 (specifically for region 0x200000000 - 0x23FFFFFFF) and targeted to LCP address space.