Software Engineering and Applications 软件工程与应用, 2022, 11(4), 826-834
Published Online August 2022 in Hans. http://www.hanspub.org/journal/sea
https://doi.org/10.12677/sea.2022.114086

Hans汉斯

基于Linux系统的APSoc芯片动态加载BIT研究

武宏吉，陈凌珊

上海工程技术大学机械与汽车工程学院，上海

收稿日期：2022年8月3日；录用日期：2022年8月15日；发布日期：2022年8月24日

摘　要

APSoc芯片以其优秀的性能、可扩展性与灵活性得到了广泛的应用，其主要分为两大部分，分别是PS (Processing System)和PL (Programmable Logic)，其中PL部分可以通过加载不同的BIT文件实现不同的功能。本文主要研究如何将国产APSoc平台与Linux系统相结合，通过PS控制PL部分，实现系统在运行过程中可以动态地加载BIT，并通过预先设计好的AXI DMA功能以测试动态加载是否成功。测试结果表明，在Linux系统上可以实现APSoc的动态加载BIT，验证了APSoc的扩展性与灵活性。

关键词

APSoc，Linux，动态加载，BIT，AXI DMA

Research on APSoc Chip Dynamically Loading BIT Based on Linux System

Hongji Wu, Lingshan Chen

School of Mechanical and Automotive Engineering, Shanghai University of Engineering Science, Shanghai

Received: Aug. 3rd, 2022; accepted: Aug. 15th, 2022; published: Aug. 24th, 2022

Abstract

APSoc chip has been widely used for its excellent performance, scalability and flexibility. It is mainly divided into two parts: PS (Processing System) and PL (Programmable Logic), among which the PL part can achieve different functions by loading different BIT files. This paper mainly studies how to combine the domestic APSoc platform with the Linux system and control the PL part through PS to realize that the system can dynamically load BIT during the running process, and test whether the dynamic loading is successful through the pre-designed AXI DMA function. The test results show that the dynamic loading BIT of APSoc can be realized on the Linux system, which verifies the scalability and flexibility of APSoc.

文章引用: 武宏吉, 陈凌珊. 基于 Linux 系统的 APSoc 芯片动态加载 BIT 研究[J]. 软件工程与应用, 2022, 11(4): 826-834.
DOI: 10.12677/sea.2022.114086

Keywords

APSoc, Linux, Dynamically Loading, BIT, AXI DMA

1. 引言

APSoc 是新一代全可编程片上系统(全可编程指的是硬件和软件都可以编程)，它的典型代表是赛灵思推出的 Zynq 7000 系列，它在单芯片内集成了基于具有丰富特点的双核 ARM Cortex-A9 多核处理器的处理系统和 Xilinx 可编程逻辑，还包含了片上储存器、外部储存器接口和一套丰富的 I/O 外设。APSoc 具有全可编程的特性，这让设计者能使用工业标准的工具在单个平台上实现高性能和低成本的应用。APSoc 目前在汽车驾驶辅助系统、摄像机、工业电机控制、智能相机、视频设备和机器视觉等领域得到了广泛的应用，具有广阔的前景[1]。

总体上来看，一般的 APSoc 都分为 PS 和 PL 两大部分，其中 PL 部分为 FPGA (现场可编程逻辑门阵列)，它由可编程输入/输出单元、可编程逻辑单元、嵌入式块 RAM 等部分组成，它可以实现任何数字器件的功能，具有高度的灵活性与可扩展性[2]。而 PS 部分包含了单个或多个的 ARM 处理器和一组相关的处理资源，形成了一个应用处理器单元(Application Processing Unit, APU) [3]，还有众多接口，既有 PS 和 PL 之间的，也有 PS 和外部部件之间的，PS 部分是整个系统的核心，PL 可以看作是 PS 部分的外设。

Linux 是一个基于 POSIX 的多用户、多任务、支持多线程和多 CPU 的操作系统，它具有免费、开源、稳定、安全的优点，在嵌入式开发领域得到了广泛的应用[4]。因为大部分 APSoc 中有 ARM 处理器，所以也支持 Linux 系统的运行，将二者相结合，可以利用 Linux 成熟的设备管理体系，更充分地发挥 APSoc 灵活多变的优势，甚至可以实现其原本不具备的功能，具有极大的研究价值[5]。

Vivado 是一款应用广泛的 EDA 软件，它可以对 ApSoc 进行设计，其最终产生的 BIT 文件用于加载到 PL 部分以实现预先设计的功能[6]。传统加载 BIT 文件的方式为 JTAG 加载，其弊端是加载慢，加载前需要让系统停止工作，且需要专门的加载工具，非常不方便[7]。因为 APSoc 中的 PS 部分可以运行 Linux 系统，并且 PS 和 PL 之间有互通的接口，那么使用 Linux 系统对整个芯片进行控制，以实现 PL 动态加载 BIT 是存在可能性的。本文选择国产的 APSoc 芯片作为试验平台，使用 Vivado 产生带有 AXI DMA 功能的 BIT 文件，在试验平台上部署 Linux 系统，研究如何实现动态加载 BIT，并对预先设计的 AXI DMA 功能进行测试以确定试验是否成功。

2. 试验平台

本文所选的试验平台为复旦微公司的 FMQL45T900 开发板，其实物图如图 1 所示。

该开发板使用的 APSoc 芯片为 45T900，其系统框图如图 2 所示

分析图 2 发现，该芯片的 PL 与 PS 之间有通用接口，这些通用接口使用的是 AXI 协议，该协议是由 ARM 公司提出的一种高性能、高带宽、低延迟的片内总线，支持突发传输，突发传输过程中只需要首地址，支持显著传输访问和乱序访问，在 APSoc 的开发中具有重要的地位，是 PS 与 PL 之间的桥梁，也是本次试验可以开展的基础[8]。

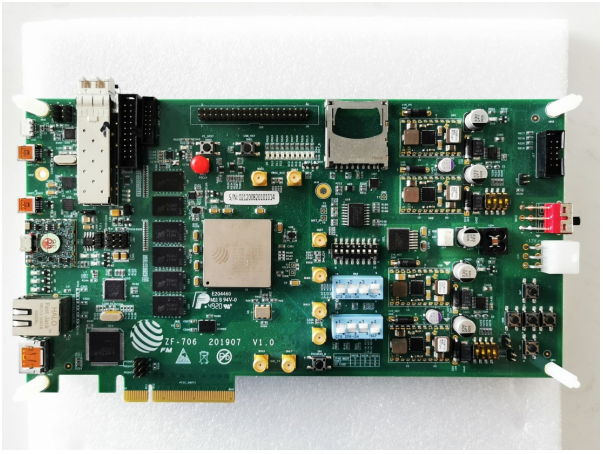

Figure 1. FMQL45T900 demoboard
图 1. FMQL45T900 开发板实物

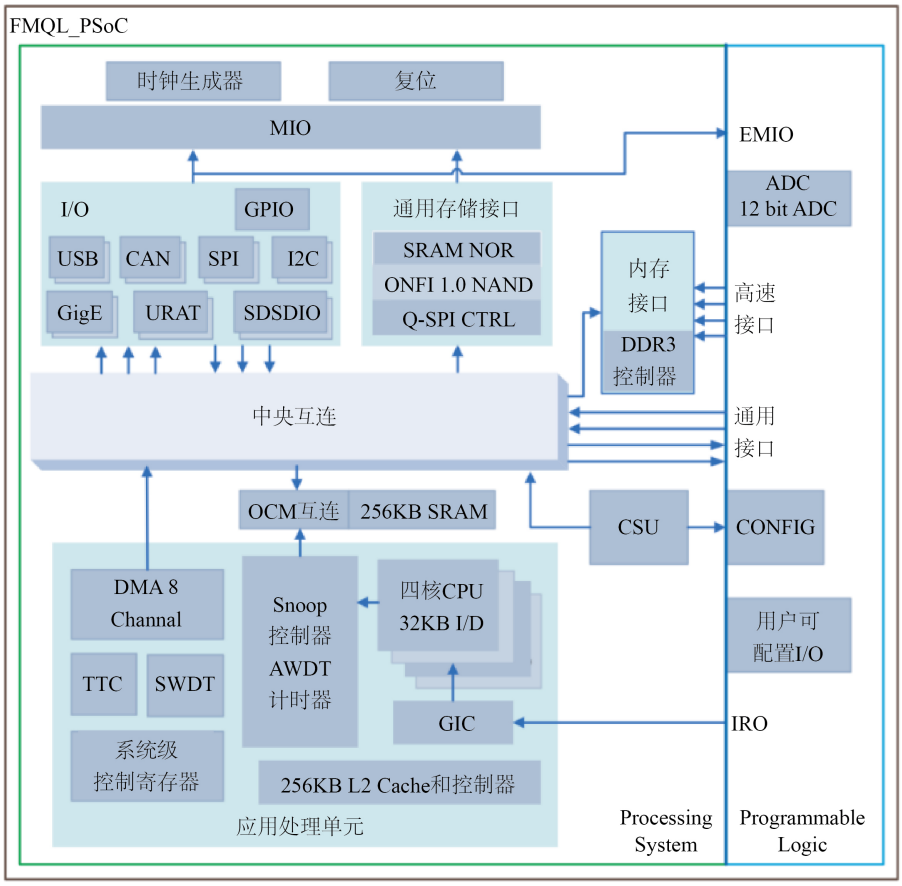

Figure 2. 45T900 system block diagram
图 2. 45T900 系统框图

AXI 接口分为四种，分别是 M_AXI_GP、S_AXI_ACP、S_AXI_GP 和 S_AXI_HP，M_AXI_GP 表示 PS 端做主机，PL 做从机，PS 通过该接口控制 PL。本次试验中，需要使用 M_AXI_GP 接口对 PL 中的 AXI DMA 模块进行控制，实现动态加载要对该接口进行释放与恢复。

3. Vivado 平台的搭建

3.1. Block Design 图

为了测试动态加载BIT是否成功，本次试验在Vivado软件中搭建了包含AXI DMA模块的工程，Block Design 如图 3 所示。

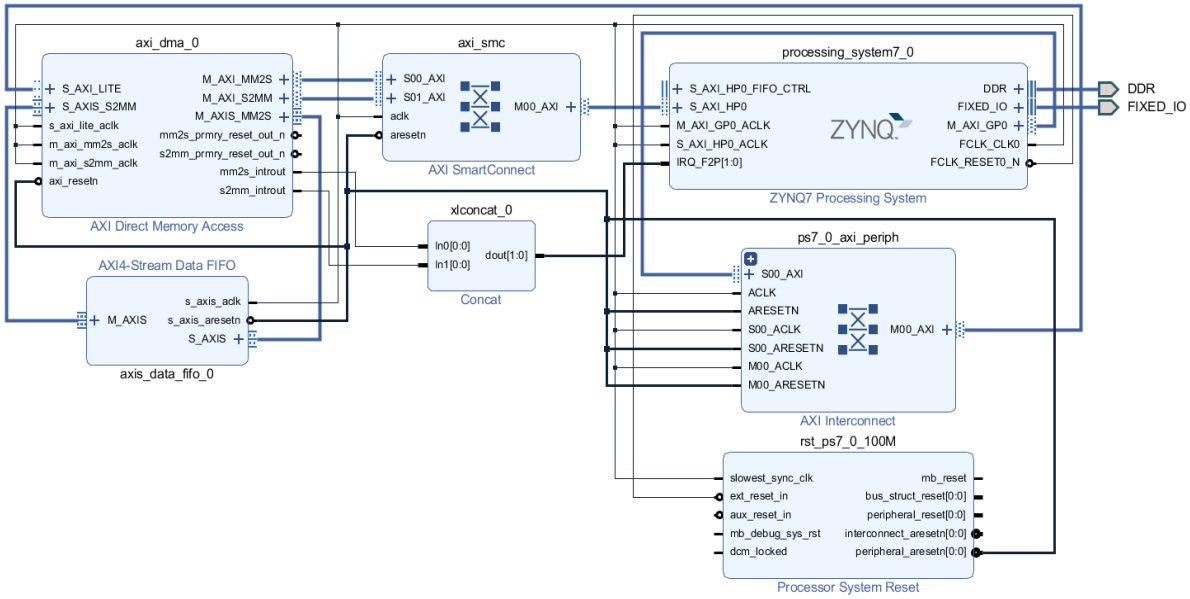

Figure 3. Block design of Vivado
图 3. Vivado 的 Block Design

该工程中，system 模块代表 PS 部分，根据开发平台的实际情况，在该模块中配置串口和 DDR3 控制器，并开启 GP 接口，以开启 PS 对 PL 的控制。axi_dma 模块为本次动态加载的测试对象，设置其字段缓冲长度的有效位数为 14，地址空间的宽度保持默认值 32，取消使能高度优化的 DMA，以便传输大量数据，开启 DMA 的读写通道，并且读写通道都设置为两个。添加 concat IP，可以将 DMA 的写中断信号与读中断信号整合为一个信号输送到 PS，简化了系统的设计。axi_smc ip 用于管理 AXI 通道，系统会将 AXI 通道集中通过该 IP 进行传输，当开启 HP 或者 GP 口时，vivado 会自动生成该 IP，无需手动添加，axis_data_fifo 作为 axi_dma 的附属 IP，也是自动生成，作用是为 axi_dma 缓冲数据，避免短时间内有大量数据经过 axi_dma，导致系统出现过载。

Vivado 可以实现自动连接 IP，但该功能无法完全满足试验的需要，经过检查，还需要将 mm2s_introut 和 s2mm_introut 与 in0、in1 相连接，然后将 dout 与 system 的 IRQ_F2P 口相连接，完成 DMA 的中断信号与 PS 的匹配。至此 block design 部分已经完成，校验无误之后生产顶层 HDL 模块用于后续的流程。

3.2. AXI DMA 分析

DMA 是现在计算机的重要组成部分，它允许不同速度的硬件设备进行沟通。而不需要依赖中央处理器的大量中断负载。否则，中央处理器需要从来源把每一片段的数据复制到寄存器，然后把它们再次写回到新的地方。在这个时间里，中央处理器无法执行其它的任务[9]。

在 45T900 中存在两种 DMA，一种是集成在 PS 中的硬核 DMA，另一种是 PL 中使用的软核 AXI DMA

IP。使用 AXI DMA 对 CPU 的占用较少，且可以降低软件的复杂度。AXI DMA 与 PS 的 DMA 控制器连接是通过 AXI_GP 接口，这个接口最高支持到 32 位宽度。AXI DMA IP 核在 AXI4 内存映射和 AXI4-Stream IP 接口之间提供高带宽直接储存访问。其可选的 scatter gather 功能还可以从基于处理器的系统中的中央处理单元(CPU)卸载数据移动任务。初始化、状态和管理寄存器通过 AXI4-Lite 从接口访问。核心的功能组成如图 4 所示。

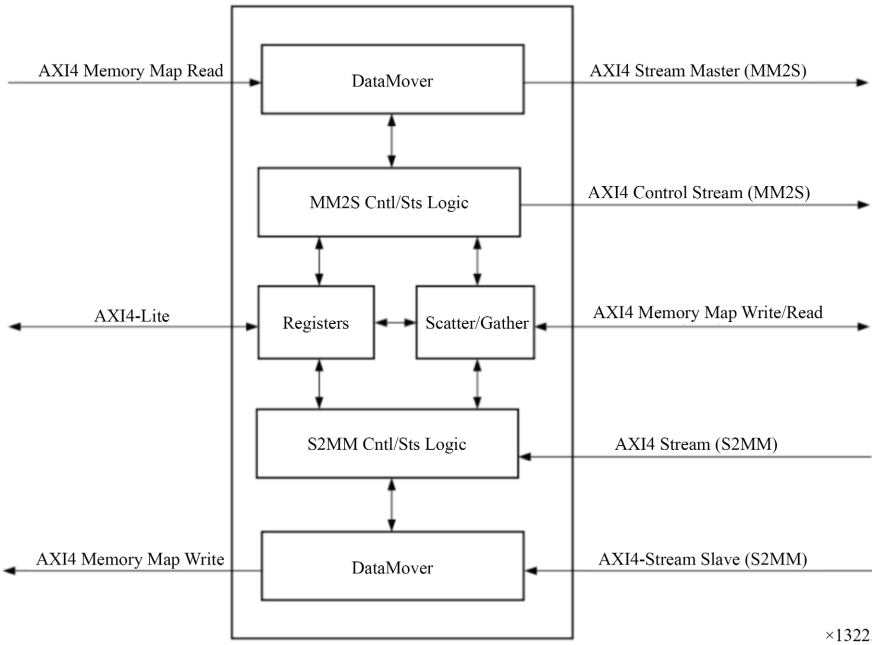

Figure 4. AXI DMA system block diagram
图 4. AXI DMA 系统框图

在上文所示的 Vivado 平台中，PS 开启了 HP0 和 GP0 接口。AXI DMA 和 AXI4 Stream Data FIFO 在 PL 中实现。处理器通过 M_AXI_GP0 接口与 AXI DMA 通信，以设置、启动和监控数据传输。数据传输通过 S_AXI_HP0 接口。AXI_DMA 通过 S_AXI_HP0 接口从 DDR 中读取数据后发送给 AXI4 Stream Data FIFO，这种情况下 AXI4 Stream Data FIFO 可以相当于带有 Stream 接口的高速 DA。

前文中有提到过，PL 可以看作 PS 的一个外设，PS 必须知道 PL 中各个 IP 的地址才能对其进行控制，所以在搭建完工程之后需要注意一下 PL 中的地址分配，后面用于 Linux 系统的定制，如图 5 所示

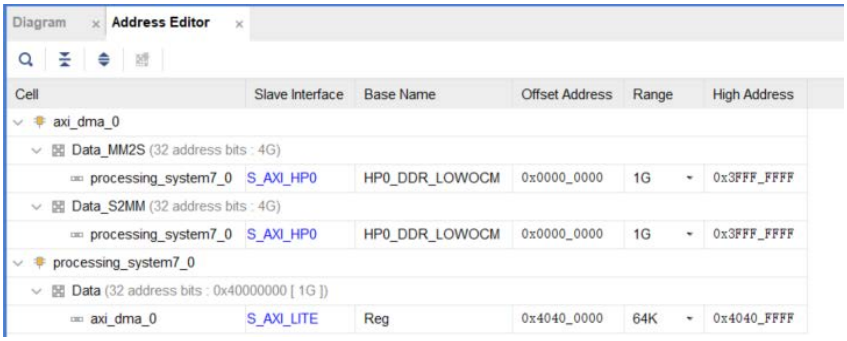

Figure 5. PL address assignment
图 5. PL 地址分配

在确认了工程没有问题之后，进行分析、综合与设计实现，最后产生本次试验要用的 BIT 文件，也就是后面动态加载的对象。

4. PS 部分软件设计

4.1. Linux 系统的制作

Linux 系统具有一切皆文件的特性，这方便了制作我们需要的系统。赛灵思公司推出的 Petalinux 工具，包括了 u-boot、Linux Kernel、device-tree、rootfs 等源码和库，可以让我们很方便的生成、配置、编译及自定义 Linux 系统。本次试验就在 Petalinux 的基础上进行系统的制作。

Linux 系统是通过设备树对设备进行管理，设备树的引入极大地方便了基于 Linux 的设备驱动开发，设备树描述了各个总线以及挂载在总线上的设备的详细信息。针对不同的目标，要定制不同的设备树相匹配。上文中设计的 AXI DMA 模块对于 Linux 系统来说也可以看作一个设备，结合上文中对 PL 中各个模块的地址分配，我们对设备树文件进行修改，如图 6 所示。

```
axi_dma_0: axidma0@40400000 {
    #dma-cells = <1>;
    clock-names = "s_axi_lite_aclk", "m_axi_mm2s_aclk", "m_axi_s2mm_aclk";
    clocks = <&clkc 15>, <&clkc 15>, <&clkc 15>;
    compatible = "xlnx,axi-dma-7.1", "xlnx,axi-dma-1.00.a";
    interrupt-names = "mm2s_introut", "s2mm_introut";
    interrupt-parent = <&intc>;
    interrupts = <0 29 4 0 30 4>;
    reg = <0x40400000 0x10000>;
    xlnx,addrwidth = <0x20>;
    xlnx,sg-length-width = <0x17>;

    dma-channel@40400000 {
        compatible = "xlnx,axi-dma-mm2s-channel";
        dma-channels = <0x1>;
        interrupts = <0 29 4>;
        xlnx,datawidth = <0x40>;
        xlnx,device-id = <0x1>;
    };
    dma-channel@40400030 {
        compatible = "xlnx,axi-dma-s2mm-channel";
        dma-channels = <0x1>;
        interrupts = <0 30 4>;
        xlnx,datawidth = <0x40>;
        xlnx,device-id = <0x0>;
    };
};
```

Figure 6. Device tree node for AXI DMA
图 6. AXI DMA 的设备树节点

该设备树节点描述了 AXI DMA 的时钟信号来源、中断号、属性名，其中 REG 表示该设备的地址范围信息，是设备树中非常重要的一个属性，而 REG 与上文中 PL 的地址分配相对应,也就是说设备的创建从硬件上来看是在 Vivado 中完成的，设备树的修改是从软件层面上添加设备。这也说明 ApSoc 可以任意根据我们的需要添加设备模块，体现了其灵活性。通过这些对 AXI DMA 的描述，驱动可以与设备相匹配并工作。

PL 端也有 RAM 资源，可以利用这些资源作为 BIT 加载的载体，为此再增添一个名为 devcfg 的设备树节点，如图 7 所示，该节点描述了 PL 的 RAM 地址空间，通过读写该设备节点，可实现对 PL 端 RAM 资源的利用，为后续的动态加载 BIT 做准备[10]。

本次试验使用赛灵思官方提供的 AXI DMA 驱动，将代码库移植到专门用于存放驱动代码的 drivers 目录，并且修改其上层的 Kconfig 和 Makefiile 文件。Kconfig 用于使新添加的驱动可以通过图形界面显示，

便于我们进行控制。Makefile 文件用于执行编译，可以将驱动代码加入到内核当中。执行编译命令，得到本次试验所需的 Linux 镜像文件，并通过 tftp 网络传送到 APSoc 的内存中，使用 uboot 的 bootm 命令启动 Linux。

```
devcfg@e0040000 {
    compatible = "fmsh,fmql-devcfg-1.0";
    reg = <0xe0040000 0x1000>;
    interrupt-parent = <0x1>;
    interrupts = <0x0 0x8 0x4>;
    clocks = <0x6 0x3f>;
    clock-names = "ref_clk";
    syscon = <0x7>;
    ddrcon = <0x8>;
    status = "okay";
};
```

Figure 7. Device tree node for DEVCFG
图 7. DEVCFG 的设备树节点

4.2. 动态加载程序设计

在前面准备的基础上，现在可以正式设计动态加载程序。图 8 为本程序的设计流程图。

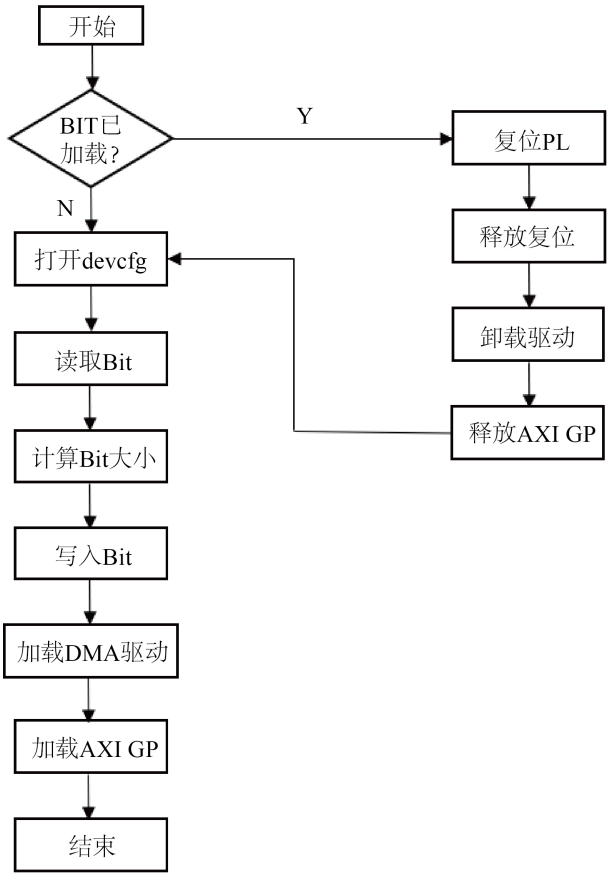

Figure 8. Program design flowchart
图 8. 程序设计流程图

函数的参数为 BIT 的文件名。进入程序后，首先判断 PL 部分是否已经加载 BIT，如果已经加载，通

过操作 PS 中的寄存器将 PL 复位并释放，从而清空已经加载的 BIT 文件。之后卸载 AXI DMA 驱动并关闭控制它的 AXI GP 接口，防止动态加载时 AXI GP 接口发出指令干扰加载。打开 devcfg 设备与 BIT 文件，保存为文件描述符，使用 Linux 中的 lseek 函数计算 BIT 文件大小，然后申请并清空大小与之相等的内存空间，之后使用 write 函数将 BIT 文件写入到 devcfg 设备中，完成 BIT 的动态加载，最后注册 DMA 的驱动，打开 AXI GP 接口，完成程序的任务。其伪代码如下。

```
Int ZynqPlLoad(const char * BIT)
{
    *(volatile    unsigned int *)(PS_USR_CMD_PL_RESET)=1;
    usleep(1000);
    *(volatile    unsigned int *)(PS_USR_CMD_PL_RESET)=0;
    ZynqLibDestory();
    fd1=open("/dev/devcfg");
    fd=open(pcPlBit);
    fileSize = lseek(fd,0,SEEK_END);
    pData = malloc(fileSize);
    memset(pData,0,fileSize);
    slRet=read(fd,pData,fileSize);
    write(fd1,pData,fileSize);
    ZynqLibInit();
Return 0;
}
```

代码中的 usleep、open、lseek、malloc、write 和 read 函数都是 linux 内核提供的系统函数，由此可见使用 linux 去进行动态加载方便了程序的实现。运用 linux 中一切皆文件的思想，将 BIT 与 devcfg 设备看成文件去读写，将复杂的加载过程抽象成具体的文件操作，其本质就是将 BIT 文件写入 PL 端的内存资源。

5. 加载试验

将动态加载的程序编译为可执行文件，命名为 pl_load。启动 linux 系统，通过 tftp 网络将 BIT 文件与 pl_load 文件传输到系统中，执行动态加载，结果如图 9 所示。

```
axidma_kxf.ko          pl_load
# cp ./top.bit.bin /lib/firmware/
# ./pl_load top.bit.bin
[ 1794.443004] Axidma1 Remove is OK!
[ 1794.446633] Axidma0 Remove is OK!
[ 1794.757054] fpga_manager fpga0: writing top.bit.bin to FMSH FMQL FPGA Manager
[ 1795.310249] xilinx-vdma 40400000.axidma0: Xilinx AXI DMA Engine Driver Probed!!
[ 1795.318043] xilinx-vdma 40430000.axidma3: Xilinx AXI DMA Engine Driver Probed!!
[ 1795.402288] axidma: axidma_dma.c: axidma_dma_init: 719: DMA: Found 1 transmit channels and 1 receive channels.
[ 1795.412619] Axidma0 Probe is OK!
[ 1795.416325] axidma: axidma_dma.c: axidma_dma_init: 719: DMA: Found 1 transmit channels and 1 receive channels.
[ 1795.426614] Axidma1 Probe is OK!
```

Figure 9. Dynamically load result
图 9. 动态加载结果

分析输出结果，可以看出 AXI DMA 的驱动首先被卸载，之后按照程序顺序将 BIT 文件写入了 devcfg 当中，最后 AXI DMA 的驱动加载成功，并找到了发送与传输的通道。可以判断程序已经执行成功。但

BIT 是否已经加载成功，需要对其包含的 AXI DMA 设备进行数据传输测试，如果测试成功，则表示系统中已经存在 AXI DMA 设备，说明 BIT 加载成功，为此将专用于 AXI DMA 测试的官方程序传入系统中，进行测试，结果如图 10 所示。

```
# ./axidma_benchmark_1ch -c 0
AXI DMA Benchmark Parameters:
        Transmit Buffer Size: 7.91 MiB
        Receive Buffer Size: 7.91 MiB
        Number of DMA Transfers: 1000 transfers

Using transmit channel 0 and receive channel 1.
Single transfer test successfully completed!
Beginning performance analysis of the DMA engine.

AXIDMA0 Timing Statistics:
        Elapsed Time: 10.52 s
        Transmit Throughput: 752.11 MiB/s
        Receive Throughput: 752.11 MiB/s
        Total Throughput: 1504.22 MiB/s
```

Figure 10. AXI DMA transfer test results
图 10. AXI DMA 传输测试结果

由测试结果可知，系统找到了 AXI DMA 的发送与接收通道，并顺利完成了传输测试，AXI DMA 设备功能正常，说明系统中已经存在 AXI DMA 设备，本次动态加载试验成功。

6. 结论与分析

本文在国产 ApSoc 平台上，应用 linux 系统研究动态加载 BIT 的方法，设计程序并测试成功，证明了动态加载的可行性与便捷性，同时也发现了 linux 与 ApSoc 相结合可以方便开发，扩展 ApSoc 的功能，具有良好的前景。

参考文献

[1] 符晓, 张国斌, 朱洪顺. Xilinx ZYNQ-7000 AP SOC 开发实践指南[M]. 北京: 清华大学出版社, 2015.

[2] 何宾, 张艳辉. Xilinx Zynq-7000 嵌入式系统设计与实现[M]. 北京: 电子工业出版社, 2017.

[3] 刘洪涛. ARM 嵌入式体系结构与接口技术[M]. 北京: 人民邮电出版社, 2010.

[4] 孙文华, 黄凌云. 嵌入式 Linux 驱动开发技术综述[J]. 广东通信技术, 2015, 35(11): 68-73.

[5] 朱其慎, 张以利. Linux 驱动程序设计初探[J]. 内江科技, 2014, 35(11): 55-57.

[6] 党宏社, 王黎, 王晓倩. 基于 Vivado HLS 的 FPGA 开发与应用研究[J]. 陕西科技大学学报(自然科学版). 2015, (1): 155-159.

[7] 虞致国, 魏敬和. 一种基于 JTAG 的 SoC 片上调试系统的设计[J]. 微电子学与计算机, 2009, 26(5): 5-8.

[8] 张浩, 魏敬和. 高效率 PLB2AXI 总线桥的设计与验证[J]. 计算机工程, 2020, 46(8): 228-234.

[9] 王修壮, 周朝显, 岳培培, 陈杰. DMA 控制器的优化与改进[J]. 科学技术与工程, 2005, 41(18): 94-95+161.

[10] 侯轶宸. 关于 FPGA 加载中的某种异常问题的解决方案[J]. 航空计算技术, 2019, 49(2): 122-124.