硬件实验

课程半期报告

(《兼容 ARM9 的软核处理器设计——基于 FPGA》学习报告)

姓名：马逸君

学号：17300180070

2019 年 9 月

本书内容综述

第一章 数字电路设计模型

1.1 数字电路的最简抽象模型

任何复杂的数字电路都可以抽象成带有输入输出的黑箱模块或其组合。

1.2 组合逻辑

组合逻辑电路的输入变化都会立即导致输出重新计算，但输出的变化(如有)会有延迟。

1.3 时序逻辑

时序逻辑最基本的器件是寄存器。
D 寄存器在时钟上升沿赋值为输入信号(有延迟)，在其它时刻保持现有状态。

1.4 同步电路

异步电路中寄存器的时钟输入来自不同的时钟源。
异步电路中采用同一时钟源的电路可以作为同一种类型来分析，而它们之间的连接必须单独考虑。

1.5 同步电路的时序路径

同步电路的时序路径分为四类：
外部输入端口输入到寄存器；寄存器输出端口输入到其它寄存器(包括自身)输入端口；寄存器输出到外部输出端口；输入端口通过组合逻辑直接到输出端口。
寄存器到寄存器之间组合逻辑消耗的时间不能超过一个时钟周期。
通常是最长的组合逻辑串决定时钟周期/频率。

1.6 RTL 描述

一个数字电路可以采用 RTL（寄存器传输级）设计描述：
第一步，确定需要用到多少寄存器；第二步，对每一个寄存器的输入端口进行描述；
第三步，从时钟频率、组合逻辑等方面确保寄存器之间正常连接。
一个寄存器最重要的属性就是其输入端口连接的组合逻辑串。
该输入描述可能是该电路中所有外部输入端口和所有寄存器输出端口的函数。
只要给出了所有寄存器的输入描述，就确定了电路所有的组合逻辑。

1.7 综合生成电路

设计数字电路时不必在硬件程序中给出确定的实现方案，我们可以用行为级或更高抽象级别的描述，而综合软件会将其综合生成我们需要的电路。

第二章 Verilog RTL 编程

2.1 Verilog 语言

assign 语句用于描述组合逻辑串。
always @ (...)的含义是，当且仅当"..."发生时重新计算语句内的表达式。
<=符号专用于寄存器描述，它的含义是符号右边的值在时钟上升沿结束后存入左边，同一个 always 语句下的所有"<="语句是并行的。

2.2 Verilog 设计的基本单元

always @ (...)语句一般用于描述寄存器的行为，可以在该语句中描述寄存器本身和它专属的一段组合逻辑。
assign 语句用于描述简单的组合逻辑，但要求在一个语句内完成描述。
always_comb 用于描述复杂的组合逻辑，系统将自动在接下来的语句中寻找敏感信号（等效于 always @ (*) ）。
function ... endfunction 语句用于描述多次调用的组合逻辑，该语句也是可综合的。

2.3 RTL 设计原则

任何编程语言都实现对信息的处理。在进行 RTL 描述时信息以寄存器的形式存在，就像 C 语言中的变量。
设计完电路后需要检验寄存器之间的相互关系是否符合预期，这通过仿真来实现。

2.4 RTL 设计要点

作者建议，描述设计时，寄存器需要与实体明确地一一对应，组合逻辑则不必完全描述出具体构造，它往往附属于寄存器。
综合软件会标出电路中最长的路径（关键路径），如果关键路径超标，则需要调整描述风格，让综合器使用较少的逻辑串来描述关键路径。

2.5 RTL 设计实例：UART 串口通信

（代码见附录）

第三章 仿真

3.1 仿真的意义

仿真就是对设计的试用过程：给予 RTL 设计一定的激励，观察响应是否正确。
RTL 设计的完整流程要求完成设计后立即检验正确性，这样才能验证当前设计能否符合对应的需求，如果可以，则在下次碰到相同需求的时候可以照搬这种设计套路，从而提高熟练度；
而不是仅仅写完 RTL 代码，然后采取哪里有问题修复哪里的策略。

3.2 testbench 文件

testbench 文件是这样一类 Verilog 程序，它给待测试的 Verilog 程序送入指定的信号，接收其输出信号(并进行一定的测试)。
一个 testbench 文件是封闭的，只为一个特定的 RTL 文件服务，一般无外部输入输出端口。

3.3 Modelsim 仿真

仿真软件默认情况下会认为信号传递没有延时。我们可以手动为每一个寄存器赋值语句给予延时，但不超过一个时钟周期。可以用`define 语句定义常量。

3.4 仿真实例：UART 串口仿真

task 语句和 function 类似，但不必返回数据。可以在仿真文件中用 task 语句表示频繁调用的代码。
可以调用系统函数，如$end 表示结束仿真，$display 表示输出调试信息。
（代码见附录）

第四章 FPGA 开发板原型验证

4.1 FPGA 内部结构

FPGA 与 CPU 的异同：都采用二进制配置文件来控制执行效果；
CPU 芯片只有一个“大脑”，包干所有“活计”，但也只能一件接一件地做事情；
FPGA 则依靠一个虚拟的“团队”，并发执行效率可能高于 CPU。
CPU 芯片的二进制配置文件由 C 语言编译而成，FPGA 则由 Verilog RTL 语言编译而成。
FPGA 称为“可编程”芯片，编译器将 Verilog 编译成二进制编程文件下载到 FPGA 中，使 FPGA 被“编程”为 Verilog RTL 功能的芯片。
Xilinx Spartan-3 系列的芯片内部构造：
四周是 IOB(Input/Output Block)，其引脚可以由用户配置成输入、输出或同时可输入输出的，电压也可由用户配置。
特殊功能部件有 DCM（生成时钟信号）、block RAM（专门用于 RAM 的实现）、Multiplier（专门用于乘法运算）。
最基本的部件是 CLB(Configurable Logic Block)。一个 CLB 内部是 4 个 Slice，分为两个 SliceL 和两个 SliceM。
每个 Slice 含有两个寄存器，用于实现 RTL 编程中的寄存器单元，其他部分（LUT、进位、MUX、运算逻辑等）实现组合逻辑。
实现组合逻辑的核心是 LUT4，它有 4 个输入端口、1 个输出端口，每种输入对应的输出可任意设定，是根据用户配置来设定的。用于实现各种各样复杂的组合逻辑。一个 LUT4 可以描述任何四输入一输出的组合逻辑。
SliceL 和 SliceM 的区别是内部存放这 16 个输出值的载体不同，SliceL 是一次写入、使用时不可更改，配置后只能作为组合逻辑；SliceM 使用寄存器，所以它可以充当 RAM，也可以用于移位寄存器。

4.2 FPGA 开发板

FPGA 开发板是以 FPGA 为核心、围绕其放置了各种外设的电路板。用户的设计可以通过和外设连接的接口，驱动外设工作或获取外设数据。
FPGA 开发板上的接口一般分为以下几类：
(1)简单电平(switch)/脉冲输入(button，按下时产生一个脉冲信号)。
(2)简单信号显示。包括 LED、七段数码管、LED 显示屏。
(3)时钟信号输入。可以直接利用焊接在开发板上的晶振，也可以接入外部时钟。
(4)存储单元。包括板上焊接的 Flash、SRAM、SDRAM 和有些开发板配备的 SD 卡槽等。
(5)专用输入输出接口。如 UART 串口、VGA 接口、USB 接口。
(6)扩展接口。用于在 FPGA 开发板上增加专用于某种应用的子板。

4.3 FPGA 设计开发流程

第一步，评估 FPGA 的片外资源和片内资源。
片外资源指 FPGA 引脚连接的资源，它是通过明确 FPGA 各个引脚具有何种功能确定的；
片内资源指 FPGA 内部结构中的资源，包括 DCM(Digital Clock Manager)、Multiplier、Block RAM，有的可以通过 Verilog RTL 描述自动调用，有的则必须通过配置 IP 的形式来进行。
第二步，围绕这些资源进行 Verilog RTL 编程。
第三步，使用仿真工具对设计进行调试。
片外资源是在 testbench 文件中“虚拟”出来的，片内资源如通过 Verilog RTL 描述的方式调用，则无需改动，如通过配置 IP 的方式，则需要调用 FPGA 生产商提供的仿真模型（当然也能手动编写仿真模型）。
第四步，FPGA 下载执行。
这一步又分为四个子步骤：
综合优化。将 Verilog RTL 描述中抽象的组合逻辑描述具体化成门级（基本逻辑单元组成的逻辑连接关系图）。
翻译(translate)。将由基本逻辑单元联系而成的网表映射成 FPGA 底层元件的逻辑结构，如把以与门、或门、非门连接的逻辑变成以 LUT4 为主体的描述。
翻译过程可以由用户的约束指定方向，比如要求面积小则会把多个逻辑门融合到尽量少的 LUT4 上，要求时间短则会优化关键路径使其经过尽量少的 LUT4。
映射(map)。把由 FPGA 底层元件连接的网表对应到该 FPGA 内部的 CLB 和 IOB 上。但仍然不是一一对应的映射，只是确定某些 CLB 分配给某些种类逻辑。
布线(place&route)。根据上一步的布局对具体 CLB 和 IOB 进行可编程连线，形成真正能够下载到 FPGA 的二进制 bit 文件。布局布线工具会进行优化，避免一个 Slice 和其它太多 Slice 连接在一起形成拥塞。

4.4 FPGA 设计内部单元

在进行 RTL 编程时，如果了解 FPGA 的构造，针对其结构进行编程，则 FPGA 的综合器可以生成非常高效的网表，在 FPGA 中执行的效率也会很高。
FPGA 内部有一些基本单元(primitive)和宏单元(macro)，在 FPGA 供应商提供的帮助文档中会列出。如果想优化 FPGA 设计，则可以在设计中直接例化，也可以通过行为相符的 HDL 描述来调用它。
以 Xilinx Spartan-3E 为例：
(1)算术功能单元。可例化调用，综合器也会直接把 HDL 中的整数乘法用这些单元级联实现。
(2)时钟单元。一般自动调用即可，也可手动干预。
(3)输入输出端口单元。一般情况下 FPGA 顶层输入输出引脚会自动根据情况应用这些单元。
(4)RAM/ROM 存储单元。欲使综合工具将相关语句例化为 Block RAM，描述风格必须和 Block RAM 的行为一致，具体参看综合器帮助文档。而在 HDL 中描述的大型 ROM 可以采用 IP 配置向导生成以 Block RAM 为基础的 ROM。
(5)寄存器和锁存器单元。几乎总是自动调用，但了解这些寄存器单元有助于我们写出风格更加相符的 HDL 描述。
(6)移位寄存器单元。不带复位信号，所以我们在 RTL 描述中描述移位寄存器时一定不要出现复位信号。
(7)Slice/CLB。
我们可以根据这些基本单元的特性，有意识地写出风格相符的 RTL 描述，引导综合工具识别为其认可的基本单元的连接。

4.5/4.6 UART 设计在 Xilinx FPGA 的下载执行

首先确定使用到的外部资源：时钟、复位按键、串口通信端口。考虑到 FPGA 开发板自带时钟和我们需要的时钟频

率不符，还需要使用片内资源 PLL，通过配置 IP 的方式来进行。
接下来设计实验，并设计出对应的顶层文件。
本例中一种可行的实验设计是，串口接收到 rx_vld/rx_data 以后，直接引入 tx_vld/tx_data 上；
也可以新增读缓冲和发送按键，用一个 1KB 的 Block RAM 作为缓冲，将 rx 读到的数据按顺序写入其中，在按下发送按键后从该 RAM 的 0 地址取数据并通过 tx 端口发送出去。
最后设计约束文件（可以通过制造商提供的软件进行，也可手动编写），指定你设计的模块的端口和 FPGA 引脚的对应关系。（如使用 FPGA 制造商提供的现成约束文件则无此必要）
完成以后，经过综合、翻译、映射、布线（这三步统称为实现，implementation）、生成码流的过程，即可下载到 FPGA 执行。
这些过程每一步都有报告供设计者查阅，如综合报告中可以检查自己的设计是否被综合成了理想的底层元件(e.g. 上面第二种实验设计中的读缓冲是否真正综合成了 block RAM)。
使用串口通信线将 FPGA 开发板和 PC 等其它设备连接，然后通过这些设备上的对应工具（如 PC 上的“迷你终端”软件）进行测试。

第五章 ARM9 微处理器编程模型

5.1 ARM 公司历史 （略）

5.2 ARM 的处理器架构概况 （略）

5.3 微处理器基本模型

微处理器只需会简单的基本操作，而简单操作组合形成的一整套动作的意义不是微处理器的责任范围，而是由计算机程序决定。
微处理器向指令池发出取指令需求、得到指令、进行执行、然后接着取下一条，如果没有中断则会一直重复这个过程。
“中断”本身不提供指令给处理器执行，而是一种提醒处理器暂时中断当前指令执行流程、切换到某段固定指令段的机制，且 CPU 有权屏蔽。
RISC 指令对 CISC 指令集中的很多指令作了分解，如将 mov (%eax), (%ecx)分解为两条指令：先将(%eax)中的数据读入 CPU 内部的暂存寄存器，再将暂存寄存器的内容写入(%ecx)。
有了内部的暂存寄存器组，RISC 指令的操作可以分解为两个处理过程：(1)数据池与寄存器组数据交换、(2)取寄存器组的数据进行处理后写回寄存器组。处理数据池的数据需要循环进行 1-2-1 的流程。

5.4 ARMv4 架构模式

ARMv4 架构微处理器有 7 种运行模式：
用户模式 USR，限制访问；系统模式 SYS，用于运行特权级操作系统；特权模式 SVC，仅供操作系统使用的一种保护模式；
快速中断模式 FIQ，用于快速高优先级中断处理；外部中断模式 IRQ，用于一般的中断处理；
数据访问异常模式 ABT，取指或取数据发生异常时进入；未定义指令异常模式 UND，执行的指令未定义时进入。
处理器启动时处于 SVC，用于对各种资源进行初始化，然后进入 USR 或 SYS。
USR 模式对资源的访问是受限的，而且它无权修改 CPU 的模式，只有借助软中断指令进入 SVC 来修改。

SYS 和 SVC 一样是访问不受限制的。
如果处在上述三种模式时发生了 IRQ 中断或 FIQ 中断而且没有被屏蔽，则系统会进入相应的中断模式，中断处理完毕后会自动恢复到先前的模式。
如果出现上述两种异常，则进入相应的异常模式，进行异常处理，然后恢复到先前的模式。（一般在进入异常模式后、异常处理前，OS 会找到发生异常的相关指令重新执行一遍，确保不是误报。）

5.5 ARMv4 架构内部寄存器 （略）

5.6 ARMv4 架构的异常中断

ARMv4 架构处理器正常情况下工作在 USR 或 SYS 模式，在出现异常中断时进入相应的模式，并修改 PC 指向相应的值。
7 种异常中断：复位，进入 SVC；FIQ 快速中断，进入 FIQ；IRQ 中断，进入 IRQ；数据处理异常，进入 ABT；取指异常，进入 ABT；未定义指令异常，进入 UND；SWI 软件中断异常，进入 SVC。

5.7/5.8 ARMv4 架构的指令集和中断 （略）

第六章 兼容 ARM9 微处理器的 Verilog RTL 设计

6.1 确定 RTL 设计的输入输出端口

显而易见进行 RTL 设计首先需要根据功能确定输入输出端口。兼容 ARM9 微处理器的端口连接到指令池和数据池，按一定法则获取指令，然后按照指令操控存储在数据池的数据。
指令池和数据池的端口也是按需设计。微处理器发出读指令使能和指令地址后，在下一周期得到指令；指令池模型需要产生取指异常中断，所以需要在给出指令的同时指出本次取指令操作是否正常。故输入信号有时钟、读使能、读地址，输出信号有指令、取指异常标志。
数据池则需要读数据、写数据，而且还分为字操作、半字操作、字节操作，所有这些操作行为都通过同一套总线发送给数据池。为此，输入端口有：时钟、使能、写使能、字节使能、地址、写数据，输出端口有：数据输出、取数据异常标志。
考虑到很多 cache 并不能如理想状况那样在一个时钟周期内释放读写结果，增加一个端口 cpu_en 以使处理器能够暂停，等待 cache 读写。
CPU 需要实现 ARMv4 架构的 7 种中断和 20 条指令，故在之前的基础上再增加 3 个中断源输入（reset、IRQ、FIQ）。至此我们就完成了输入输出端口的设计。

6.2 经典三级流水线架构

在进行微处理器的 RTL 设计之前，我们必须确定整个微处理器的流水线架构。采用流水线架构是因为，指令从取出到执行完毕是一个相对复杂的过程，为了不让一条指令占用太多 CPU 时间，我们把它分成 3 或 5 步执行。
一般的三级流水线架构：取指、译码、执行。但对于数据池相关指令，把数据池数据载入寄存器无法在一周期内完成，因为数据池也需要一个时钟周期才能释放读数据结果。这样一来，这类 LDR 指令的执行必须花费 5 周期：取指、译码、算地址、访存、回写。
我们可以看见一条 LDR 指令可以使三级流水线停顿两拍。不幸的是，LDR 指令一般是相当常见的，为了防止其严重降低效率，我们从结构上改变流水线，使其影响消失。

6.3 经典五级流水线架构

五级流水线解决了 LDR 指令延时的问题，甚至可以认为，正是因为 LDR 需要 5 时钟周期，人们才修改流水线为五级的。一般的五级流水线架构：取指、译码、算地址、访存、回写。

但在两条相关（前一条指令的输出是后一条指令的输入）的 MOV 指令相连的时候，仍然必须在五级流水线的基础上延时 2 时钟周期（否则后一条指令将得到“过时”的结果）。这个问题可以修正，我们可以把前一条指令的回写结果提前使用，即后一条指令进入执行阶段时直接取出在流水线上尚未回写的上一条指令的结果（而非寄存器内过时的结果）。但当一条 LDR 连接一条相关的 MOV 时，该方法无效（因为需要使用的数据尚未取出），仍然必须延时 2 拍。

6.4 三级流水线改进架构

为了给数据池访问指令留出足够的时间，同时又不让寄存器处理类指令出现不必要的延时，可以设计出这样的一种三级流水线：

寄存器类指令和数据池的写指令同经典三级流水线，数据池读指令 LDR 延迟一拍变为四级——取指、译码、执行、回写。这里“执行”同时完成算地址和发出访存信号两个操作。

如果遇到正处于第三级的读指令的输出和处于第二级的指令的输入相等的时候，必须将后者暂缓执行，并在第三级插入一条空指令。

6.5 适用于兼容 ARM9 微处理器的三级架构

接下来我们考察具体如何用这套流水线实现 ARM9 指令。ARM9 指令分为两大类：数据池读写指令、寄存器处理指令。

数据池读写指令本身不涉及复杂的数据操作，相对复杂的是生成读写地址，读写地址一般是 Rn 加第二操作数生成，而最复杂的第二操作数是 LDR0 和 LDR1，是通过 Rm 或立即数移位生成的。所以，数据池读写指令的复杂数据公式为：Rm>>num + Rn。

寄存器处理指令中比较复杂的是乘法 MULT 和长乘法 MULTL，它们处理数据的公式是 Rm * Rs + Rn；此外还有 DP0、DP1、DP2，它们的公式是 Rm>>Rs + Rn 或 Rm>>num + Rn。

由此可见 ARM 指令集最多需要两步操作，第一步是由 Rm 和 Rs 进行移位或乘法操作得到第二操作数，第二步是由第二操作数和 Rn 进行加法为主的运算操作。由于乘法 MULT 和长乘法 MULTL 指令的存在，乘法运算无法避免，所以必须使用乘法器。那么我们需要考虑是否可以通过乘法器实现移位运算。

DP 指令有四种移位方式：逻辑左移、逻辑右移、算术右移、循环右移。乘法器显然能实现逻辑左移；对逻辑右移，Rn 和 1<<num 相乘的 64 位结果中，我们发现高 32 位就等于 Rn<<(32-num)，问题解决；对算术右移，如果符号位为 1 则将 Rm 取反再乘法再取反；对循环右移，将 64 位乘法结果的高 32 位和低 32 位进行或操作即得。

这样一来，每条指令都可以用一个乘法器串联一个加法器来实现。那么流水线的第三级就会是一个乘法器串联一个加法器，决定设计频率的关键路径就是这两者的耗时之和。

再考虑能否打破串联，以减小关键路径。我们发现，可以在第二级译码的时候顺便进行乘法运算得到第二操作数。至于译码操作本身，因为 ARM9 指令集非常简单，只需要 21 条 assign code_is_xxx = ...的语句生成非常简单的组合逻辑即可完成，不必设计专门的译码单元。这样就在不增加流水线级数的前提下打破了串联，缩减了关键路径，从而兼容 ARM9 处理器可以运行在更高频率上。

虽然如此，这一设计会带来数据冲突问题，因为乘法操作会从寄存器组中读取 Rm 和 Rs，如果第二级乘法正在读取 Rm 和 Rs，同时第三级的指令又在改写要读取的寄存器，就会造成数据冲突，与之前讲到的 LDR 的冲突一样。

（未完待续）

第七章 在兼容 ARM9 处理器内核上运行 Hello World 程序

7.1 基于 FPGA 的 SoC 设计流程

在进行具体的 SoC 设计时，处理器的指令池、数据池、中断必须具体化到设计实体中：
指令池一般具体化为 ROM，数据池则在实现不同功能时例化为 ROM、RAM 或者寄存器（寄存器通常连接外设，对外设定义行为规则或交换数据）中的不同种类，IRQ 和 FIQ 中断的功能也应该予以定义。
这样一来，设计过程就从以寄存器为最小单元的逻辑设计转变为以微处理器为核心、带 ROM、RAM 和寄存器的独立嵌入式系统为中心的逻辑设计；
输入端口作用于该嵌入式系统的两个途径是：输入端口连接到微处理器的中断端口，或者连接到寄存器中改变某个寄存器的值；
输出端口连接某个寄存器，嵌入式系统写入这个寄存器时输出改变。
接下来在进行嵌入式软件的编写之前，必须将 ROM、RAM、寄存器分配到具体的地址，这是嵌入式系统运行的内环境；
ROM 和 RAM 直接分配连续的地址范围即可，但 ROM 和 RAM 的不能重叠；
寄存器的地址是离散的，可以直接选用某种成熟的单片机外设寄存器设定情况，也可以自定义访问规则并写在.h/.c 文件中。
在内环境定义完毕后，参照这个内环境进行 SoC 硬件和嵌入式软件的设计工作；
工作结束后，嵌入式软件生成一段.bin 代码，SoC 硬件将这段.bin 代码置入 ROM，即完成了一个全面的 SoC 设计。
欲完成输出"Hello World"的 SoC 设计，首先容易想出，除了实现嵌入式系统之外，还需给 UART 模块配置专用寄存器，以实现对串口收发数据的操作；
第一步是定义兼容 ARM9 内核的工作内环境，第二步是定义上述的专用寄存器（包括对应的行为规则）。

7.2 使用 RealView MDK 编译 Hello World 程序

在嵌入式软件 IDE——RealView MDK-ARM 中建立工程，依次定义好单片机具体型号、编译设置(如不生成 thumb 指令)、其它命令（如从生成的.axf 文件中解析出.bin），开始开发。
示例工程中的文件用途：startup.s，完成配置单片机和设定栈，初始化各个模式下的状态；Hello.c，主函数所在文件；Retarget.c，重定位 printf 依赖的一些函数使其定位到 Serial.c 的相关函数中；Serial.c，描述串口底层函数。

7.3 仿真输出 Hello World

μVision 中编译产生的.bin 文件的内容就是一行一行的 ARM 指令。我们可以用 Verilog 写一个 testbench 文件，直接用$fopen 和$fread 函数打开这个.bin 文件，将其内容读入到 ROM（指令池）中，并解析兼容 ARM 内核，最后通过$write 输出 Hello World 字符。
可以通过在波形中添加 ROM 相关的信号（rom_en/rom_addr[31:0]/rom_data[31:0])来观察指令的提取情况，添加 RAM 相关的信号（ram_cen/ram_wen/ram_flag[3:0]/ram_addr[31:0]/ram_wdata[31:0]/ram_rdata[31:0]）来观察数据池的读写情况，也可添加 r0~re 来观察微处理器内部寄存器组的变化情况。
（代码、运行结果见附录）

7.4 建立 Hello World 的 FPGA 设计工程

兼容 ARM9 处理器可以与嵌入式编译软件 μVision 完美结合：只要从 μVision 编译出.bin 文件，转换成.coe，读

入 ROM 中，即可让 CPU 按嵌入式程序的指令工作。
仿照第一节中所说的流程，不难画出以 ARM9 内核为核心的 SoC FPGA 的设计框图，具体包含兼容 ARM9 内核、UART、ROM 和 RAM（对应指令池数据池）、寄存器。采取自顶向下的方式，来完成一个完整的设计。
ROM：生成 IP，读入 hello.bin。IP 的设置根据实际需要设置为双口 ROM，各个端口的宽度、读写方式等也根据 CPU 总线情况而定，深度则根据 μVision 中设定的 ROM 大小除以宽度计算而来，根据需要为每个端口都配备 enable 引脚，并指定 init file（因为只接受.coe 格式的 init file，所以需要在仿真过程中通过$fopen、$fread、$fdisplay 将.bin 转换成.coe）。
RAM：生成 IP。大小由 μVision 中的设定指定，宽度为总线宽度 32 位，因为还涉及到字节使能 ram_flag[3:0]，故勾选 byte write enable。
寄存器：已经在μVision 中设定完成。板上实现是功能模拟（而不一定必须分配到指定的物理地址），将寄存器对应的地址（ram_addr == 32'he000_0004）写在其涉及的功能语句中即可（如 tx_vld <= ram_cen & ram_wen & (ram_addr == 32'he000_0004)）。
（运行结果见附录）

第八章 兼容 ARM9 处理器内核性能测试 Dhrystone Benchmark

8.1 Dhrystone 2.1 简介

Dhrystone 主要测试微处理器的整数性能，嵌入式处理器由于面积和低功耗限制会将其性能向整数和逻辑运算倾斜，所以用 Dhrystone 测试嵌入式处理器更能体现其测量价值。ARM 公司对其旗下的处理器内核一般采用 Dhrystone 2.1 版本进行测试。
μVision 的 DHRY 例程包含 7 个源码文件：startup.s、retarget.c、serial.c、time.c、dhry.h、dhry_1.c、dhry_2.c。前三个文件同第 7 章的 3 个文件，第 7 章在使用之前修改了这 3 个文件，我们将修改过的文件拷贝过来覆盖原来的 3 个文件。time.c 是一个定时器，用于计算处理器完成 Dhrystone 的耗时；dhry_1.c 和 dhry_2.c 是 Dhrystone 测试的主程序。
值得一提的是，Dhrystone 是以测试经过的时钟周期数作为测试时间的，所以受试处理器 1s 跑完 Dhrystone Benchmark 的次数还需要除以时钟频率(MHz)才能得到最终的性能指标 Dhrystone MIPS。另外，运行次数需要足够大。

8.2 移植 Dhrystone 2.1 进行编译

首先，我们根据实际情况对于在 FPGA 运行上不必要的代码进行删改。然后，为了避免大量修改仿真文件，我们将该工程的器件修改为之前的 LPC2100 系列，但是 LPC2101 的 ROM、RAM 太小以至于无法编译，尝试发现最低需要设为 LPC2104。最后参照上一章的操作将编译生成的.axf 转换成.bin。
当然了，为了获得尽可能高的执行效率，我们需要根据 ARM 公司提供的指导 Dhrystone Benchmark 编译的文档，设置合适的编译参数，如-O3 -Otime --no_inline --no_multifile。

8.3 仿真运行 Dhrystone Benchmark

我们直接拷贝上一章的 testbench 文件进行修改，内容包括修改时钟频率为 1MHz、按 LPC2104 设定扩大 ROM/RAM 大小、定时发送 IRQ 中断（因为定时中断是 time.c 中用到的计时方式）。
接下来还需要将输入仿真次数的语句改为直接赋值，因为仿真环境无法输入数据。再仿照上一章的方法启动仿真，

但遇系统提示 Out of heap memory，故将 startup.s 中定义的 Heap Size 扩大，再次运行略多于 3s 的仿真时间（测试需 3s，还需若干毫秒打印完整的测试信息），得到测试结果。
（运行结果见附录）

8.4 在线可编程的 FPGA SoC 设计工程

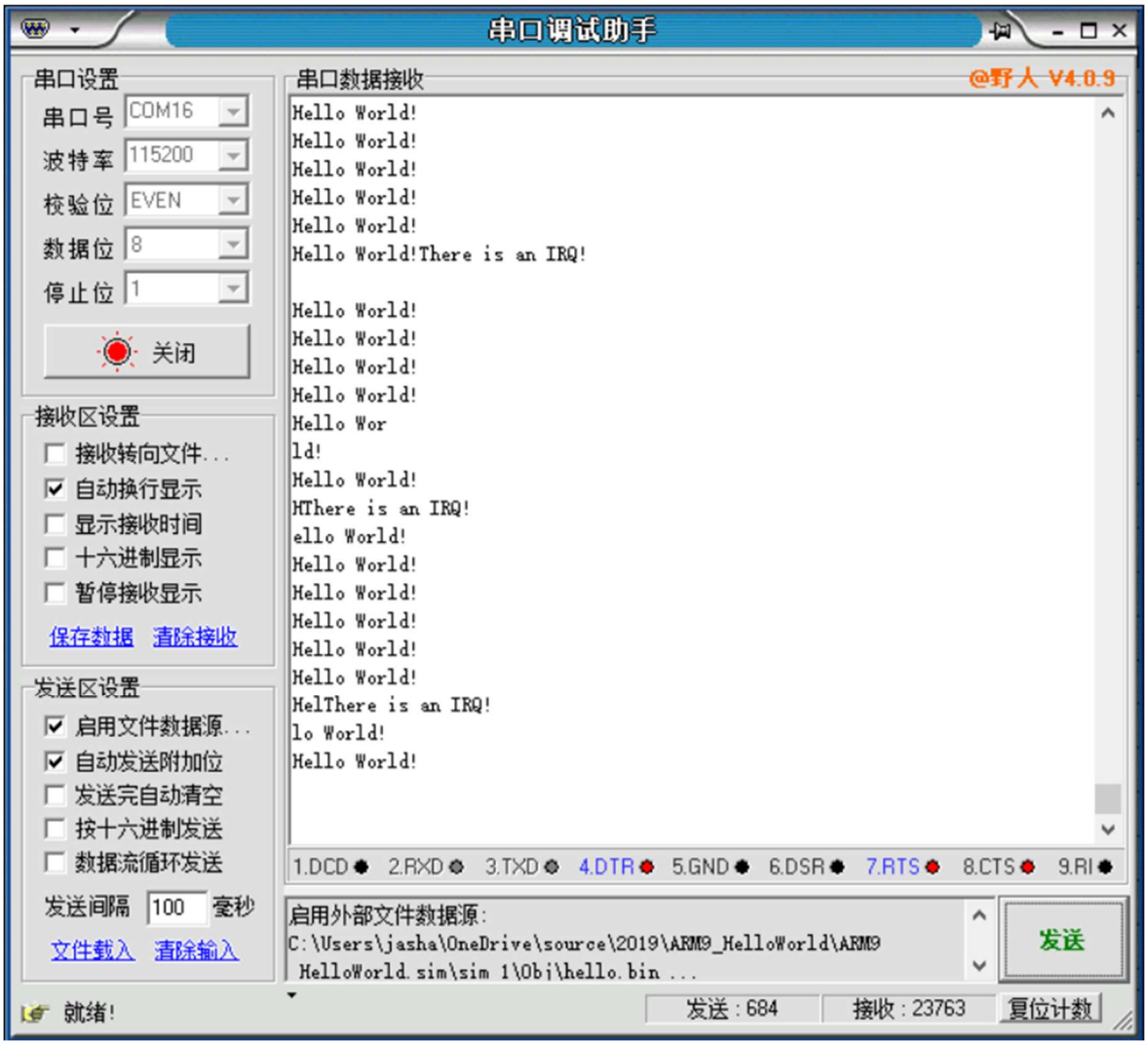
按上一章的设计，每修改一次 ROM 的内容就必须重新综合一次，时间消耗巨大。我们可以设法通过 UART 将.bin 文件发送到 FPGA 开发板的 ROM 中，解决这个问题。
这样就必须使得 ROM 可以设置为可写状态，从而这个 FPGA SoC 设计工程有两个状态：工作状态和编程状态（处理器内核不工作，ROM 可写）。所以我们将 SW[0]连接到 cpu_en 和 ROM 的 wea(Port A Write Enable)信号。并且需要更改设计，使得可以每从串口接收 4 字节数据就送入 ROM 一次。最后还需要按需添加定时发送中断的逻辑。
还需注意，处理器切换到工作态时必须手动复位一次，以使得 PC 从 0x0 再次开始工作。
这样一来，FPGA 就可以在线编程，从而可以认为是完整地模拟了单片机开发板的功能。

8.5 Dhrystone Benchmark 在开发板中运行

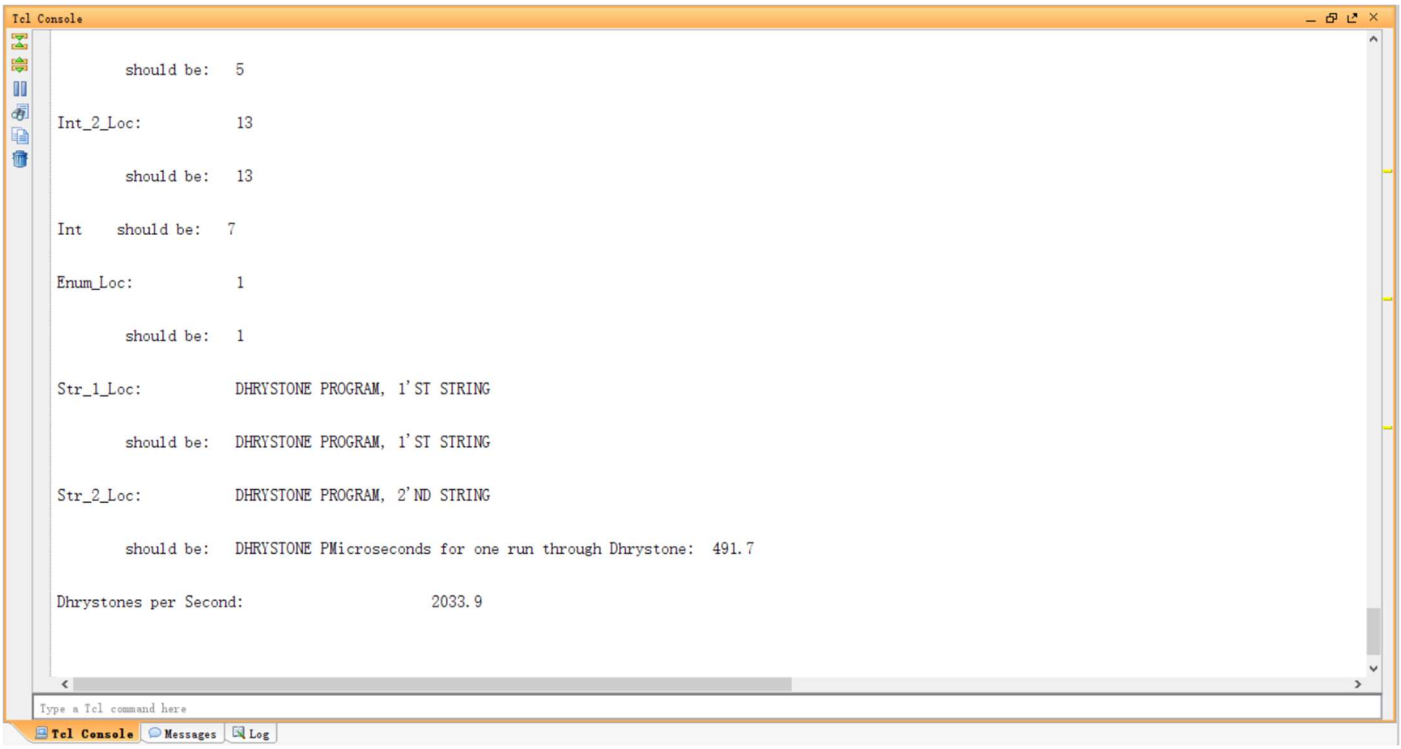
（运行结果见附录）

运行结果图

7.4 建立 Hello World 的 FPGA 设计工程



8.3 仿真运行 Dhrystone Benchmark



8.5 Dhrystone Benchmark 在开发板中运行

代码

2.5 RTL 设计实例：UART 串口通信

```
module rxtx(
    input logic clk,
    input logic rst,
    input logic rx, //接收端口
    input logic tx_vld, //发送有效（请求发送）
    input logic[7:0] tx_data, //发送数据

    output logic rx_vld, //接收有效
    output logic[7:0] rx_data, //接收数据
    output logic tx, //发送端口
    output logic tx_rdy //发送端口就绪
);
    //用 2-3 个寄存器同步来消除异步传输的不确定性
    logic rxx;
    logic rx1, rx2, rx3;
    always_ff @ (posedge clk)
    begin
        rx1 <= rx; rx2 <= rx1; rx3 <= rx2; rxx <= rx3;
    end

    //检测 rxx 的变化，该检测结果为计数器服务
    logic rx_change;
    logic rx_dly;
    always_ff @ (posedge clk)
        rx_dly <= rxx;
    assign rx_change = (rxx != rx_dly); //用一个周期之前的值与当前值比较，得到变化与否的标志

    //时钟计数器，在 rx 保持为同一个值的时候，用这个值持续的时长来计算持续了多少位。
    logic[13:0] rx_cnt;
    always_ff @ (posedge clk)
    begin
```

```
        if (rst) rx_cnt <= 14'b0;
        else if (rx_change || (rx_cnt == 14'd2603)) rx_cnt <= 14'b0;
        else rx_cnt <= rx_cnt + 1'b1;
    end

    //采样标志
    logic rx_en;
    assign rx_en = (rx_cnt == 14'd1301); //半周期时刻，该采样了

    //接收数据状态指示寄存器，是否开始接收数据
    logic data_vld;
    always_ff @ (posedge clk)
        if (rst) data_vld <= 1'b0;
        else if (rx_en && ~rxx && ~data_vld) data_vld <= 1'b1; //rxx 接收到一个低电平信号，说明接收将在一周期后开始
        else if (data_vld && (data_cnt == 4'd9) && rx_en) data_vld <= 1'b0;
        else;

    //专用于接收过程的计数器，含义是当前接收过程已接收了多少位（不含起始位），每接收 1 位则值加 1
    logic[3:0] data_cnt;
    always_ff @ (posedge clk)
        if (rst) data_cnt <= 4'b0;
        else if (data_vld) //在接收已经开始后，每周期计数
            if (rx_en) data_cnt <= data_cnt + 1'b1;
            else; //latch
        else data_cnt <= 4'b0;

    //接收到的前 8 位（0-7）：数据位
    //logic[7:0] rx_data;
    always_ff @ (posedge clk)
        if (rst) rx_data <= 8'b0;
        else if (data_vld & rx_en & ~data_cnt[3]) rx_data[data_cnt] <= rxx; //rx_data <= {rxx, rx_data[7:1]};
        else;
```

```
    //接收完毕，发送接收有效信号
    //logic rx_vld;
    always_ff @ (posedge clk)
        if (rst) rx_vld <= 1'b0;
        else rx_vld <= data_vld && rx_en && (data_cnt == 4'd9);

    //发送模块，接收到请求发送信号时暂存发送数据
    logic[7:0] tx_rdy_data;
    always_ff @ (posedge clk)
        if (rst) tx_rdy_data <= 8'b0;
        else if (tx_vld && tx_rdy) tx_rdy_data <= tx_data;
        else;

    //发送数据状态指示寄存器
    logic trans_vld;
    always_ff @ (posedge clk)
        if (rst) trans_vld <= 1'b0;
        else if (tx_vld) trans_vld <= 1'b1;
        else if (rx_en && trans_vld && (trans_cnt == 4'd10)) trans_vld <= 1'b0;
        else;

    //发送过程的计数器
    logic[3:0] trans_cnt;
    always_ff @ (posedge clk)
        if (rst) trans_cnt <= 4'b0;
        else if (trans_vld)
            if (rx_en) trans_cnt <= trans_cnt + 1'b1; //采样信号只需要在计数器中判断，其它过程中不必
            else;
        else if (!trans_vld) trans_cnt <= 4'b0;

    //发送过程
    //logic tx;
    always_ff @ (posedge clk)
        if (rst) tx <= 1'b1;
```

```
      else if (trans_vld)
         if (rx_en)
            case (trans_cnt)
            4'd0: tx <= 1'b0;
            4'd1: tx <= tx_rdy_data[0];
            4'd2: tx <= tx_rdy_data[1];
            4'd3: tx <= tx_rdy_data[2];
            4'd4: tx <= tx_rdy_data[3];
            4'd5: tx <= tx_rdy_data[4];
            4'd6: tx <= tx_rdy_data[5];
            4'd7: tx <= tx_rdy_data[6];
            4'd8: tx <= tx_rdy_data[7];
            4'd9: tx <= ^tx_rdy_data;
            4'd10: tx <= 1'b1;
            default: tx <= 1'b1;
            endcase
         else; //latch
      else tx <= 1'b1;

   //发送就绪
   //logic tx_rdy;
   assign tx_rdy = ~trans_vld;

endmodule
```

3.4 仿真实例：UART 串口仿真

```
`timescale 1ns / 1ps

module rxtx_tb();
   reg clk = 1'b0; //初值
   always clk = #20 ~clk;

   reg rst = 1'b1;
   initial #40 rst = 1'b0;
```

```
    reg RxD = 1'b1, tx_vld = 1'b0;
    reg[7:0] tx_data = 8'h0;
    logic rx_vld, TxD = 1'b0, tx_rdy; //从 X 变成 0 也会算作一次 negedge，然后触发一次意料之外的 Tx 的检测过程。所以需要赋初值为 0.
    logic[7:0] rx_data;


    rxtx u_rxtx(
           .clk    (clk),
           .rst    (rst),
           .rx     (RxD),
           .tx_vld (tx_vld),
           .tx_data(tx_data),


           .rx_vld (rx_vld),
           .rx_data(rx_data),
           .tx     (TxD),
           .tx_rdy (tx_rdy)
    );


    task rx_send;
    input[7:0] b;
    integer i;
    begin
        RxD = 1'b0; //#100
        for (i = 0; i < 8; i = i+1) #104167 RxD = b[i]; //#104267
        #104167 RxD = ^b; //#
        #104167 RxD = 1'b1;
        #104167 RxD = 1'b1;
    end
    endtask


    task tx_byte; //等到 tx_rdy 为有效的时候，将输入信号送 tx_data
    input[7:0] b;
    begin
        while (~tx_rdy)
```

```
        @ (posedge clk) ; //等待一个时钟周期再检测 tx_rdy 信号
    @ (posedge clk) ; //等待一个时钟周期
    #3 tx_vld = 1'b1; tx_data = b;
    @ (posedge clk) ;
    #3 tx_vld = 1'b0; tx_data = 8'b0;
end
endtask

//检测 RxD 端口
always @ (posedge clk)
    if (rx_vld)
        $display("--Byte %2h received @ %0d.", rx_data, $time); //显示当前时间
    else;

//检测 TxD 端口
integer i;
reg[7:0] rec_byte;
reg checkbit;
always @ (negedge TxD) //在 TxD 下降沿（起始的 0bit 出现时），开始一次检测过程
begin
    #52080 if (TxD != 1'b0) $display("--Start bit error @ %0d.", $time);
    for (i = 0; i < 8; i = i+1) #104167 rec_byte[i] = TxD;
    #104167 checkbit = TxD;
    #104167 if (TxD != 1'b1) $display("--End bit error @ %0d.", $time);
    #52080 $display("--Byte %2h transmitted @ %0d.", rec_byte, $time);
           if (checkbit != ^rec_byte) $display("--Check bit error @ %0d.", $time);
end

//检测过程
initial begin
#100 rxtx_tb.rx_send(8'b01011010);
    rxtx_tb.tx_byte(8'b10100101);
#2000000 $stop;
end
endmodule
```

7.3 仿真输出 Hello World

```
`timescale 1ns / 1ns
`define DEL 2

module tb();
    logic clk = 1'b0;
    always clk = #5 ~clk;

    logic rst = 1'b1;
    initial #10 rst = 1'b0;

    logic[7:0] rom[8191:0]; //大小为 8KB（按照 Keil 中的设定）的 ROM 块
    logic rom_en;
    logic[31:0] rom_addr;
    logic[31:0] rom_data;
    always_ff @ (posedge clk)
        if (rom_en) rom_data <= #`DEL {rom[rom_addr+3], rom[rom_addr+2], rom[rom_addr+1], rom[rom_addr]};
        else;

    integer fd, fx, i;
    initial begin
        for (i = 0; i < 8192; i = i+1) rom[i] = 0; //避免 ROM 中没有被 hello.bin 覆盖的区域为 X，供.coe 文件生成用
        fd = $fopen("../Obj/hello.bin", "rb");
        fx = $fread(rom, fd); //把.bin 文件的内容送入 ROM，作为指令池
        $fclose(fd);
        fd = $fopen("hello.coe", "w"); //把.bin 文件的内容按要求的格式写入另一个.coe 文件，供 Design Sources 中初始化 ROM IP 用
        $fdisplay(fd, "memory_initialization_radix = 16;");
        $fdisplay(fd, "memory_initialization_vector = ");
        for (i = 0; i < 8192; i = i+4)
            $fdisplay(fd, "%2h%2h%2h%2h%1s", rom[i+3], rom[i+2], rom[i+1], rom[i], i == 8188 ? ";" : ",");
        $fclose(fd);
```

```
    end


    logic[31:0] ram[511:0];
    logic ram_cen, ram_wen;
    logic[3:0] ram_flag;
    logic[31:0] ram_addr;
    logic[31:0] ram_wdata;
    logic[31:0] ram_rdata;
    always_ff @ (posedge clk)
        if (ram_cen && ~ram_wen) //读数据池
            if (ram_addr == 32'he000_0000) ram_rdata <= #`DEL 32'h0; //如果是读寄存器，则给出 SERIAL_FLAG 的值 0x0（因为现在还没有连接 UART 模块，而是通过仿真器输出数据，所以 SERIAL_FLAG 保持为 0，表示随时可以输出数据）
            else if (ram_addr[31:28] == 4'h0) ram_rdata <= #`DEL {rom[ram_addr+3], rom[ram_addr+2], rom[ram_addr+1], rom[ram_addr]}; //根据地址段，判别是读 ROM 区域还是读 RAM 区域
            else if (ram_addr[31:28] == 4'h4) ram_rdata <= #`DEL ram[ram_addr[27:2]];
            else;
        else;
    always_ff @ (posedge clk)
        if (ram_cen && ram_wen && ram_addr[31:28] == 4'h4) //写 RAM
            ram[ram_addr[27:2]] <= #`DEL {
                (ram_flag[3] ? ram_wdata[31:24] : ram[ram_addr[27:2]][31:24]),
                (ram_flag[2] ? ram_wdata[23:16] : ram[ram_addr[27:2]][23:16]),
                (ram_flag[1] ? ram_wdata[15: 8] : ram[ram_addr[27:2]][15: 8]),
                (ram_flag[0] ? ram_wdata[ 7: 0] : ram[ram_addr[27:2]][ 7: 0])
            };
        else;
    always_ff @ (posedge clk)
        if (ram_cen && ram_wen && ram_addr == 32'he000_0004) $write("%s", ram_wdata[7:0]); //写 SERIAL_OUT，也就是输出字符串
        else;


    logic irq = 1'b0;
    initial begin
        #100000 irq = 1'b1; //运行到 100000ns 时给出一周期 irq 脉冲
```

```
        #10 irq = 1'b0;
    end

    arm9_compatiable_code u_arm9(
        .clk        (clk),
        .cpu_en     (1'b1),
        .cpu_restart(1'b0),
        .fiq        (1'b0),
        .irq        (irq),
        .ram_abort  (1'b0),
        .ram_rdata  (ram_rdata),
        .rom_abort  (1'b0),
        .rom_data   (rom_data),
        .rst        (rst),

        .ram_addr   (ram_addr),
        .ram_cen    (ram_cen),
        .ram_flag   (ram_flag),
        .ram_wdata  (ram_wdata),
        .ram_wen    (ram_wen),
        .rom_addr   (rom_addr),
        .rom_en     (rom_en)
    );

endmodule
```