

重庆大学

硕士学位论文

基于FPGA的JPEG实时图像编解码系统

姓名：廖彦

申请学位级别：硕士

专业：信号与信息处理

指导教师：张玲

20070420

摘 要

JPEG 是联合图像专家组(Joint Picture Expert Group)的英文缩写, 是国际标准化组织(ISO)和 CCITT 联合制定的静态图像压缩编码标准。JPEG 的基于 DCT 变换有损压缩具有高压缩比特特点, 被广泛应用在数据量极大的多媒体以及带宽资源宝贵的网络程序中。

动态图像的 JPEG 编解码处理要求图像恢复质量高、实时性强, 本课题就是针对这两个方面的要求展开的研究。该系统由图像编码服务器端和图像解码客户端组成。其中, 服务器端实时采集摄像头传送的动态图像, 进行 JPEG 编码, 通过网络传送码流到客户端; 客户端接收码流, 经过 JPEG 解码, 恢复出原始图像送 VGA 显示。设计结果完全达到了实时性的要求。

本文从系统实现的角度出发, 首先分析了系统开发平台, 介绍 FPGA 的结构特点以及它的设计流程和指导原则; 然后从 JPEG 图像压缩技术发展的历程出发, 分析 JPEG 标准实现高压缩比高质量图像处理的原理; 针对 FPGA 在算法实现上的特点, 以及 JPEG 算法处理的原理, 按照编码和解码顺序, 研究设计了基于改进的 DA 算法的 FDCT 和 IDCT 变换, 以及按发生频率进行优化的霍夫曼查找表结构, 并且从系统整体上对 JPEG 编解码进行简化, 以提高系统的处理性能。最后, 通过分析 Nios 嵌入式微处理器可定制特性, 根据 SOPC Builder 中 Avalon 总线的要求, 把图像采集, JPEG 图像压缩和网络传输转变成用户自定义模块, 在 SOPC Builder 下把用户自定义模块添加到系统中, 由 Nios 嵌入式软核的控制下运行, 在 FPGA 芯片上实现整个 JPEG 实时图像编解码系统(Soc)。

在 FPGA 上实现硬件模块化的 JPEG 算法, 具有造价低功耗低, 性能稳定, 图像恢复后质量高等优点, 适用于精度要求高且需要对图像进行逐帧处理的远程微小目标识别和跟踪系统中以及广电系统中前期的非线性编辑工作以及数字电影的动画特技制作, 对降低成本和提高图像处理速度两方面都有非常重大的现实意义。通过在 FPGA 上实现 JPEG 编解码, 进一步探索 FPGA 在数字图像处理上的优势所在, 深入了解进行此类硬件模块设计的技术特点, 是本课题的重要学术意义所在。

关键词: JPEG, DCT, FPGA, 霍夫曼, Nios 嵌入式微处理器, Avalon 总线, Soc

ABSTRACT

JPEG is the English abbreviation of Joint Picture Experts Group (Joint Picture Expert Group). It's a static image compression standard developed jointly by International Organization for Standardization (ISO) and CCITT. JPEG lossy compression based on DCT with high compression ratio characteristics is widely used in multimedia resources and bandwidth network which need to process great volume of data.

Dynamic Image JPEG codec processing requires high image quality, real-time speed. The thesis is to do research aiming at the two requirements. The system consists of image compression server, which samples the dynamic images with a camera, processes them with JPEG encoder, then transport them through network, and image decompression client, which receives bitstream from the server and processes with JPEG decoder, recover the original images, then display them on VGA monitor. The result shows that the design can fully meet the real-time requirements.

From the system perspective, the thesis firstly analyzes the system development platform and introduces the structural characteristics of FPGA as well as its design process and guiding principles. The later to be introduced is JPEG image compression technology development process and the reason why JPEG compression standards can achieve high quality images with high compression rate. Then the thesis focuses on the the characteristics of FPGA structure to achieve JPEG algorithm, introduces the design according to the sequence of encoding and decoding, including the FDCT and IDCT transform with improved DA algorithm, and optimized Huffman table frequency by frequency order, and the simplified structure of JPEG from the whole system aspect to improve the processing performance. Finally, the analysis aims at Nios embedded processor of customizable characteristics, translating image acquisition, JPEG image compression and network transmission into user-defined modules according to SOPC Builder Avalon bus requirements with the SOPC Builder, where the user-defined module can be added to the system, under the control of soft-core Nios Embedded. The whole system is achieved on a single FPGA chip (SOC) with real-time JPEG images encoding and decoding.


JPEG algorithm as a FPGA hardware module with low power consumption, low production costs and stable performance, Image high image quality, has great advantages to apply to the high precision required for processing images frame by frame in long-range identification and tracking systems, and Early Nonlinear editing and

digital film animation graphics production in the system of Radio and Television. There is a very great practical significance to reduce costs and improve image processing speed. using FPGA to achieve JPEG encoding and decoding and further explore the advantage of FPGA in the Digital Image Processing, in-depth understanding of such hardware module design characteristics, this is the subject's most important of academic significance.

Keywords: JPEG, DCT, FPGA, Huffman, Nios embedded processor, Avalon bus, System-On-a-Chip

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得重庆大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名：

签字日期：2007年6月5日

学位论文版权使用授权书


本学位论文作者完全了解重庆大学有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权重庆大学可以将学位论文的全部或部分内内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

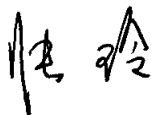
保密（☐），在 年解密后适用本授权书。

本学位论文属于

不保密（☒）。

（请只在上述一个括号内打“√”）

学位论文作者签名：

导师签名：

签字日期：2007年6月5日

签字日期：2007年6月5日

1 绪论

1.1 引言

随着宽带 Internet 以及数字多媒体技术的快速发展,以 Internet 为传输媒介的视频会议、可视电话、远程视频监控、远程医疗系统以及数字流媒体等新的视频应用层出不穷。这些应用都需要对大量的图像进行网络传输。

由于图像的信息量巨大,给存储器的存储容量、通信干线信道的带宽,以及计算机的处理速度增加极大的压力。例如:一张 A4(210mm×297mm)幅面的照片,若用中等分辨率(300dpi)的扫描仪按真彩色扫描,共有 $(300 \times 210 / 25.4) \times (300 \times 297 / 25.4)$ 个像素,每个像素占3个字节,其数据量为26M字节,数据量实在太太大。单纯靠增加存储器容量,提高信道带宽以及计算机的处理速度等方法来解决这个问题都是不现实的,必须对图像进行压缩,去掉图像信息中的冗余,减少网络传输的信息量。

JPEG 是国际标准化组织(ISO)和 CCITT 联合制定的静态图像的压缩编码标准。和相同图像质量的其它常用文件格式(如 GIF, TIFF, PCX)相比, JPEG 是目前静态图像中压缩比最高的^[1]。相同的一幅图像,在解压缩后图像质量几乎没有差异的情况下,用 JPEG 压缩得到的数据量约为 BMP 格式的 2~3%,或者 GIF 格式的 10%左右。正是由于 JPEG 的高压缩比,使得它在数据量极大的多媒体以及带宽资源宝贵的网络程序中,针对静态图像的压缩有着非常广泛的应用。

当 JPEG 编码的速度提高到一定程度,就能够对动态图像进行编码,这种 JPEG 压缩编码称为 M-JPEG(Motion-JPEG)。M-JPEG 一个很大的特点就是图像处理等待时间较少,适合于在例如视频移动侦测或物体追踪环境下进行图像处理。M-JPEG 可提供所有实际使用的图像分辨率,从用于移动电话的 QVGA 分辨率,到 4CIF 的全视频显示尺寸,甚至是更高的百万像素分辨率。系统无论在移动或复杂环境情况下都能保证视频质量,并提供高图像质量(低压缩)和低图像质量(高压缩)的灵活选择,同时能真正保证图像处理的实时性。

1.2 本课题国内外研究现状及发展趋势

1.2.1 JPEG 压缩标准的产生和应用

JPEG 是联合图片专家组 Joint Photographic Experts Group 的英文缩写,它是国际标准化组织 ISO 和国际电信联盟 ITU 的联合技术小组,于 1986 年底成立,其任务是为连续色调(灰度或彩色)静止图像压缩制定通用的国际标准。JPEG 标准包括图像编码和解码过程以及压缩图像数据的编码表示^[2]。

在 1987 年 6 月用电视测试图像进行了不同方案的主观实验测试后, JPEG 从 12 个候选方案中筛选了 3 个方案, 并对其进行了改进。1988 年进行了最终测试和评选, 结果是自适应的大小为 8×8 的离散余弦变换 DCT(Discrete Cosine Transform) 方案最佳。

1988 年至 1990 年, JPEG 进行了大量细致的工作, 于 1991 年 3 月正式提交了 ISO/CD10918 建议草案。随着多媒体应用的不断扩大, 尤其是 INTERNET 应用和无线通信的发展, 对于多媒体传输的要求越来越高, 而静止数字图像的压缩编码是整个多媒体视频技术的基石, 因此, 全世界无数的学者专家正不断完善已有标准并开发研究新算法。

1997 年 3 月, 国际标准化组织(ISO)/国际电工委员会 IEC(International Electrotechnical Commission)第一联合委员会第 29 分委员会(语音、图形、多媒体和超媒体编码委员会)第一工作组(静止图像编码组)又开始征集新一代静止图像压缩标准 JPEG2000 方案, 结果, 有 22 种方案应征参与竞争, ISO/IEC 对上述方案进行了测试, 从主观方面和客观方面作了比较。同年 11 月, 开始建立验证模型(Verification Model), 该验证模型在多次核心实验过程中将不断被修正, 最终形成 JPEG2000 标准。

JPEG 标准应用于连续变化的静止图像, 是假想为适用范围非常广泛、通用性很强的技术。它结合采用了预测编码、不定长编码等多种压缩方法, 在图像质量损失较小的情况下, 达到了较高的压缩比, 因此在静止图像压缩上得到了广泛的应用。JPEG 应用的领域包括互联网、彩色传真、打印、扫描、数字摄像、遥感、移动通信、医疗图像和电子商务等等, 目前网站上百分之八十的图像都是采用 JPEG 的压缩标准。JPEG 在数码相机、PDA、手机等手持设备和嵌入式设备中的使用正方兴未艾。

JPEG 发展而来的对动态图像的压缩—M-JPEG 压缩技术, 能对图像进行逐帧的压缩与解压缩, 方便对动态图像进行单帧编辑。在广电系统的前期节目编辑以及数字电影的特技动画制作中, 由于待处理的动态图像帧间差异较大, 并且为了保持最佳的视觉质量, 对图像的质量要求较高, 必需把帧作为基本的压缩单位。在这类应用中, M-JPEG 压缩方式对活动视频图像通过实时帧内编码过程单独地压缩每一帧, 利用其空间相关性进行帧内压缩, 在编辑过程中可以随机存取压缩的任意帧, 这刚好满足了逐帧编辑的需要。此外, M-JPEG 还具有系统实现方便, 压缩后图像恢复质量好, 且没有码率上限等优点。因此, 目前我国非线性编辑系统可精确到帧的编辑以及多层图像处理中, 就广泛采用了 M-JPEG 压缩。

1.2.2 JPEG 图像压缩的现状与趋势

JPEG 已经开发了三个图像标准^[3]。第一个直接称为 JPEG 标准,正式名称叫“连续色调静止图像的压缩编码”(Digital Compression and Coding of Continuous-tone still Images), 1992 年正式通过。如这个名字表示的一样,是具有连续黑白或彩色色调的静止图像的编码技术。

JPEG 开发的第二个标准是 JPEG-LS(ISO/IEC 14495, 1999)。JPEG-LS 仍然是静止图像无损编码,能提供接近有损压缩压缩率。

JPEG 的最新标准是 JPEG 2000(ISO/IEC 15444, 等同的 ITU-T 编号 T.800), 于 1999 年 3 月形成工作草案,2000 年底成为正式标准(第一部分)。根据 JPEG 专家组的目标,该标准将不仅能提高对图像的压缩质量,尤其是低码率时的压缩质量,而且还将得到包括根据图像质量,视觉感受和分辨率进行渐进传输,对码流的随机存取和处理、开放结构、向下兼容等许多新功能。

JPEG2000 的目标是进一步改进目前压缩算法的性能,以适应低带宽、高噪声的环境,以及医疗图像、电子图书馆、传真、Internet 网上服务和保安等方面的应用。对于有较好的图像质量,较低的比特率或者是一些特殊功能的要求(比如渐进传输和感兴趣区域编码等)时,JPEG2000 将是较好的选择。但是到目前为止,由于各种原因,还没有一种 JPEG2000 的软硬件系统产品被人们所广泛应用^[4]。事实表明,随着 JPEG 系统产品的不断优化,在大多数场合下,采用 JPEG 就可以满足要求了。而且,JPEG 的低复杂度,低成本的优点也是别的标准所无法取代的。因此,JPEG2000 不可能完全替代 JPEG。在未来很长一段时间内,JPEG 仍然将是主流的静止图像压缩标准。

目前 JPEG 图像压缩的实现主要分两种途径进行。一种是专用图像处理芯片的设计,另一种就是在 PC 机上用软件实现的 JPEG 压缩算法。

专用图像处理芯片是近年来世界范围内的研究热点,它的速度很高、性能稳定,所以在一些尖端的科技应用领域,如卫星遥感图片以及某些高速清晰摄影中具有非常广泛的应用。但是它的价格对于我们普通的消费类产品来说,还是有一定的距离。而且这些技术一般都是由国外大的厂商所掌握,他们有各自的优化算法和设计结构,但是这些技术都是不对外公开的。

由于 PC 机运算速度的提高,我们也可以用软件实现 JPEG 压缩算法。软件算法的灵活性强,可以用来满足多样的压缩要求,如对图像局部特征的压缩(在医学图像中的应用),以及如何采用优化的算法来保留图像特殊细节(如指纹识别)等等。但是由于压缩算法的计算量比较大,软件完成这些算法所消耗的时间必然比较长,无法应用在实时性场合中。

随着微电子技术和半导体工业的不断创新发展,集成电路的集成度和生产

工艺水平得到不断提高,从而使在一个半导体芯片上实现系统级的集成已成为可能,数字技术已进入片上系统(System on a Chip)时代^[5-6]。把图像处理压缩算法集成成为硬件模块,添加到嵌入式 CPU 中,成为 SOC 片上系统的一部分,是未来发展的趋势。

1.3 本课题研究的基本任务与要求

1.3.1 研究任务

本课题主要研究内容:设计一个包含有 JPEG 编码,图像网络传输和 JPEG 解码三大部分的实时图像处理系统。在图像的压缩端,要求开发板对摄像头进来的数据进行实时采集,实时压缩,并且作为网络服务器,等待客户端的连接请求;在图像的解压缩端,在网络服务器准备好之后,发起连接,然后将接收到的数据进行实时解压缩,把图像恢复出来。实时的要求是图像处理速度在每秒 25 帧以上。具体工作包括以下几个方面:

①深入了解研究 JPEG 压缩和解压缩的原理,查阅相关的快速算法,比较它们的优缺点和适用的条件,找出适合 FPGA 上并行处理,且运行效率高的算法结构。

②综合考虑 FPGA 的面积与速度平衡的关系,顺序完成量化, DCT, 游程编码, 霍夫曼编码,最后组合成一个 JPEG 解码的总模块。解码部分的设计也是一样。

③研究 SOPC Builder 下,自定义模块的主端口、从端口的设计规范,以及中断的使用方法,将自定义模块挂到 Avalon 总线上,在 CPU 的控制下运行。

④结合图像采集、图像压缩、网络传输、图像解压缩和图像显示,在 FPGA 开发板上进行整体调试和优化。

1.3.1 课题要求

通过这些工作要达到以下几个目的:

①根据 JPEG 压缩算法的特点,以适合于 FPGA 运算的结构,设计出高效的图像压缩、解压缩硬件模块,进一步探索 FPGA 在数字图像处理上的优势所在,深入了解进行此类硬件模块设计的技术特点;

②利用 Nios 嵌入式软核的可定制性,在 SOPC Builder 下将各个硬件模块组装结合成一个完整的系统,掌握 Nios 软核对嵌入式系统的控制原理,研究在一个芯片上实现多种图像压缩处理系统的可能性(SOC)。

1.4 论文的主要内容和章节安排

本文在分析 JPEG 标准以及利用 FPGA 实现图像处理算法原理的基础上,介绍了针对图像压缩解压缩系统的实时性与图像质量的要求,利用 FPGA 设计造价低功耗低,性能稳定,图像恢复后质量高等,适用于精度要求高且需要对图像进行

逐帧处理的 JPEG 实时图像编解码系统的过程。

论文的章节安排如下：第一章，绪论；第二章，介绍系统开发的平台；第三章，JPEG 编解码技术；第四章，JPEG 实时图像编解码系统的硬件设计；第五章，Nios 控制程序的设计；第六章，总结与展望。

2 系统开发平台

2.1 开发工具

Quartus II 是一个应用广泛, 功能完善的 PLD 设计软件, 它是 Altera 为了适应微电子技术及其应用的飞速发展, 尤其是 SOC (片上系统) 技术发展的需要而推出的新版本设计工具。

Quartus II 继承了 Altera 的前设计软件 Max+Plus II 的所有优点, 包含了设计输入、综合方针工具、时限分析工具、功率评估工具、PLD 布局布线工具和产品验证工具等, 功能更加完善。它的物理综合选项应用在编译的布局布线阶段, 而与采用了何种综合工具无关。Quartus II 软件用户能够利用强大的时序逼近流程特性来优化设计, 时序逼近流程由于其包含了内置物理综合工具以及丰富的图形分析和编辑工具, 提供无与伦比的交互探测能力, 使其超过按键式编译结果的性能。

此外, Quartus II 的设计子工具 SOPC Builder 更是开创了嵌入式系统设计的新理念。传统的可编程单芯片系统(SOPC)设计, 需要用户手动去分配地址空间资源, 还要面对各种结合了硬件、嵌入式软件和高速 I/O 的系统验证问题。SOPC 设计团队通常需要包括可编程逻辑器件(PLD)硬件工程师、板级硬件工程师和软件工程师等, 他们都有各自的验证需求。SOPC Builder 将处理器、存储器、I/O 口、LVDS、CDR 等系统设计需要的功能模块集成到一个 PLD 器件上, 构建成一个可编程的片上系统。它是可编程系统, 具有灵活的设计方式, 可裁减、可扩充、可升级, 并具备软硬件在系统可编程的功能, 其设计效率显著提高, 给广大的 SOC 设计开发人员带来很大的方便。

Quartus II 的开发主界面如图 2.1 所示。它分为四个主要的子窗口。其中左上方的子窗口为工程的浏览窗口, 它显示的是用户设计的顶层模块, 设计所包含的所有输入文件以及整个设计的模块结构; 左下方的子窗口为设计状态图, 在进行工程的编译综合以及仿真的时候, 窗口内会显示每一步的进度以及所耗费的时间; 右边为设计输入窗口, 用户可以选择通过不同的设计输入手段, 如原理图, VHDL 或 Verilog-HDL 来进行 FPGA 设计; 最下方的子窗口为信息提示窗口, 在程序的综合仿真过程中, Quartus II 会显示警告和错误信息, 提示用户设计中可能出现的问题所在。

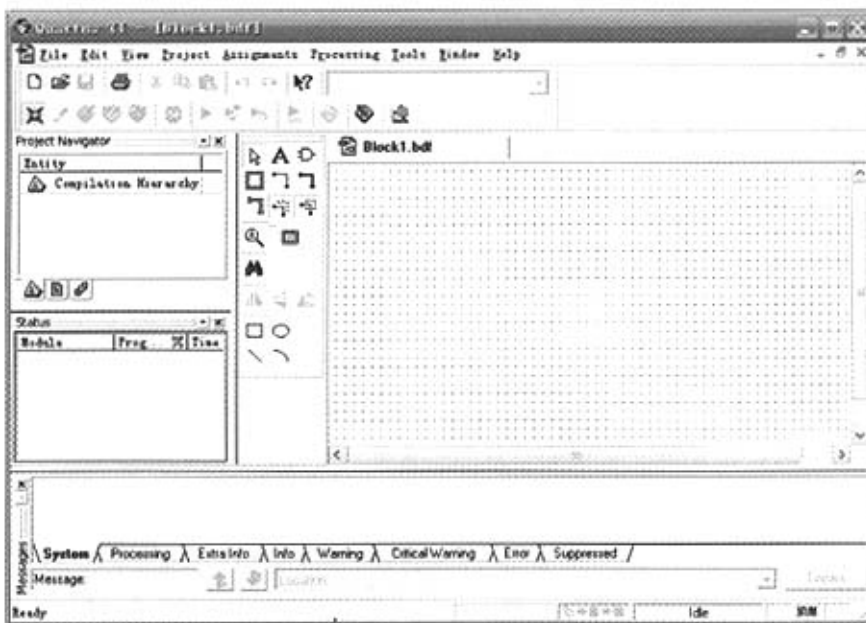


图 2.1 Quartus II 主界面

Fig 2.1 Desktop of Quartus II

2.2 硬件平台

本系统硬件的仿真平台采用的是 Altera 公司的 CycloneII 、 NiosII 开发套件，它提供可编程芯片系统（SOPC）开发所需的一切。在 Altera Nios II 系列嵌入式处理器和低成本 Cyclone II EP2C35 器件基础上，该开发套件为多种价格敏感的嵌入式应用提供理想的开发和原型设计环境。该开发套件主要包括：

- Cyclone II EP2C35F672 器件，片上资源包括 35,000 个逻辑单元以及 105K 片上 RAM；
- 4M 容量的 FLASH，8M 的 SDRAM，512K 的 SRAM 存储器
- 10/100 以太网物理层/介质访问控制（PHY/MAC）；
- 支持 NTSC/PAL/Multiformat 的视频解码芯片；
- 10 位 VGA DAC 芯片；
- 16×2 LCD 液晶显示屏，8 个七段数码显示管，以及多个 LED 显示灯；
- 丰富的引脚资源，包括 RS232，IrDA，PS/2 等。

2.2.1 FPGA 芯片 Cyclone II EP2C35F672

Cyclone II 系列FPGA是继Cyclone系列低成本FPGA在市场上取得成功之后，Altera 公司推出的更低成本的FPGA。它将低成本FPGA 的密度扩展到了68 416 个逻辑单元(LEs)，采用TSMC(台基电)的90nm工艺，与竞争对手的90nm工艺FPGA 相比，性能高出60%而功耗减低一半，而其价格则几乎可以与ASIC 产品竞争。

该系列FPGA支持Altera公司的Nios II嵌入式软核处理器。Nios II具有灵活的定制特性而且可以非常容易地实现各种外设的扩展。对于并行事务处理，可以在一个FPGA上放置多个Nios II 软核，大大提高处理器的效率，也方便多个小组同时开发，进一步加快新产品研发速度。

EP2C35F672是Altera 公司Cyclone II系列产品之一。封装为672脚的Fineline BGA，是2C35中引脚最多的封装，最多可以有475个I/O 引脚可以供用户使用。EP2C35F672由33, 216个LE组成，片上有105个M4K RAM块，每个M4K RAM块由4K(4096)位的数据RAM加512位的校验位共483, 840位RAM 组成。端口宽度根据需求配置，可以是 $\times 1$, $\times 2$, $\times 4$, $\times 8$, $\times 9$, $\times 16$, $\times 18$, $\times 32$, 或 $\times 36$ 位。在 $\times 1$, $\times 2$, $\times 4$, $\times 8$, $\times 9$, $\times 16$, $\times 18$ 等模式下，是真正的双口操作(可以配置成一读一写、两读或两写)。

此外，EP2C35F672内置35个 18×18 的硬件乘法器，利用Altera公司提供的DSP Builder和其他DSP的IP库，可以低成本地实现数字信号。片上大容量的M4K RAM以及经过专门优化的对外部存储器的高速存取特性，使它们非常适合数字信号处理器或数字协处理器的应用场合。此外，它的片上有4 个PLL，可实现多个时钟域的信号处理。再加上Altera公司提供的数字信号处理器IP核，使数字信号处理产品的开发更加容易。

2.2.2 图像采集

Analog Device公司的ADV7181B是专用于视频模/数转换的芯片，它支持ITU656 数字视频标准，可在PAL和NTSC两种制式间自由切换， 并接收行、场同步信号进行同步。通过I2C总线对ADV7181B进行设置，从摄像头过来的PAL制式模拟视频信号通过ADV7181B，转变成可被FPGA采样的数字图像信号，输出格式如图2.2所示：

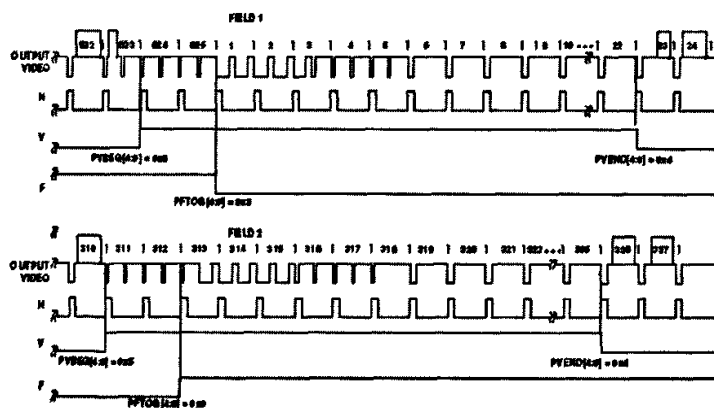


图2.2 PAL制式行场同步信号

Fig 2.2 H, V, Field signals of PAL output

2.2.3 网络传输

系统采用 DAVICOM 的 DM9000 进行网络通信，它是一款性价比高的网络芯片，拥有 10/100Mbps 高速 PHY，4K 双字节 SRAM，工作电压 3.3V，耐压 5.0V。根据 DM9000 硬件结构，编写好驱动程序，并且转变成用户自定义模块以便在 SOPC builder 中挂到 AVALON 总线上。此外，由于 Nios II 自带的 Uc/OS-II 操作系统，内含 TCP/IP 协议，在软件程序中可以直接进行调用。

2.2.4 VGA 显示

图像解码后，得到的像素值需送到 VGA 显示器显示出来。系统选用 Analog Device 公司推出的 ADV7123 高精度，高速度数模转换芯片。它的参数如下：

- 240MHz 的最大样速度；
- 三路 10 位 D/A 转换器；
- SFDR :
当时钟频率为 50MHZ；输出为 1MHZ 时，-70dB；
当时钟频率为 140MHZ；输出为 40HMZ 时，-53dB；
- 与 RS-343A/RS-170 接口输出兼容；
- DA 转换器的输出电流范围为：2mA 到 26mA；
- TTL 兼容输入；
- 单电源+5V/+3.3V 工作；
- 低功耗（3V 时最小值为 30mW）；

在本系统中，VGA 显示的时钟设置为 25.2MHZ，将解码恢复后 320*240 的图像进行插值，得到分辨率为 640*480 的图像在 VGA 显示器上显示出来。

2.3 FPGA 设计流程与指导原则

2.3.1 FPGA 设计流程

FPGA 基本开发流程^[7]如图，主要包括：设计输入(Design Entry)；设计仿真(Simulation)；设计综合(Synthesize)；布局布线(Place & Route)；配置(Configuration)。

①设计输入：主要有原理图输入和 HDL 输入两种方式，一般开发商都同时支持两种输入方式。还有的甚至提供更多的输入方式，如 Xilinx 公司的 ISE6.0 就提供四种输入方式，包括 EDIF 网表输入。有些熟悉硬件设计的工程师开始喜欢利用原理图进行设计，这种方法非常直观，但基

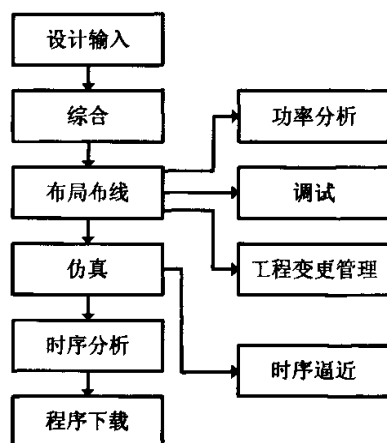


图 2.3 FPGA 设计流程

Fig 2.3 flow of FPGA design

于可移植性和规范化方面的考虑，绝大部分深入 FPGA 设计和 ASIC 设计的工程师最终都将统一到 HDL 平台上来。

②设计仿真：包含功能仿真和时序仿真两项主要内容，功能仿真忽略了综合和布局布线导致的时延等因素，仅仅从逻辑上进行仿真，这对设计思路的验证是有帮助的，但必须通过时序仿真作进一步验证，发现并修正时序问题。

③设计综合：将 HDL 语言生成用于布局布线的网表和相应的约束。综合效果直接导致设计的性能和逻辑门的利用效率，因此，许多可编程逻辑器件开发商都支持第三方综合和仿真工具，著名的有：Synplicity、Synopsys 和 ModelSim 等。

④布局布线：工具利用综合生成的网表，在 FPGA 内部进行布局布线，并生成可用于配置的比特流文件（有了比特流文件就可 down 到板子里了）。布局布线工具与可编程逻辑器件工艺及其布线资源密切相关，一般由可编程逻辑器件开发商直接提供。

2.3.2 FPGA 设计指导原则

FPGA 设计有许多内在规律可循，总结并掌握这些规律对于较深刻的理解可编程逻辑设计技术非常重要。

①面积和速度的平衡与互换原则^[8]：

这里的“面积”是指一个设计所消耗的 FPGA 逻辑资源数量，“速度”是指设计在芯片上稳定运行时所能够达到的最高频率。面积和速度是一对对立统一的矛盾体。要求一个设计同时具备设计面积最小，运行频率最高，是不现实的。从理论上讲，一个设计如果时序余量较大，所能跑的频率远远高于设计要求，那么就能通过功能模块复用较少整个设计消耗的芯片面积；反之，如果一个设计的时序要求很高，那么可以一般可以通过数据流串并转换，并行复制多个操作模块，对整个设计采取“乒乓操作”和“串并”转换的思想进行处理，在芯片输出模块处再对数据进行“并串转换”。当面积和速度冲突时，采用速度优先的原则。

②硬件原则：

硬件原则主要针对 HDL 代码编写而言。FPGA 逻辑设计所采用的硬件描述语言（HDL）同软件语言（如 C、C++等）是有本质区别的。HDL 是采用了语言形式的硬件抽象，它的最终实现结果是芯片内部的实际电路。所以，正确的编码方法是，首先要对所需要实现的硬件电路结构与连接十分清晰，然后再用适当的 HDL 语句表达出来。

硬件原则的另外一个重要理解就是“并行”和“串行”的概念。硬件系统比软件系统速度快、实时性高，其中一个重要的原因就是硬件系统中各个单元的运算是独立的。所以，在写 HDL 代码时，应充分理解硬件系统的并行处理特点，合理安排数据流的时序，提高整个的效率。

③系统原则:

系统原则包含两个层次的含义:首先,从更高层面上看,它表示的是一个硬件系统,一块单板如何进行模块划分与任务分配,什么样的算法和功能适合放在传统 FPGA 里面实现,什么样的算法和功能适合放在 DSP、CPU 里面实现,或者在使用内嵌 CPU 和 DSP block 的 FPGA 中如何划分软硬件功能;然后,具体到 FPGA 设计,就要求对设计的全局有个宏观上的合理安排,比如时钟域、模块复用、约束、面积和速度等问题。在系统上复用模块节省的面积比在代码上的细节修改效率要高得多。

④同步设计原则:

同步设计的原则是 FPGA 设计的最重要的原则之一。其基本原则是使用时钟沿触发所有的操作。

同步设计中,稳定可靠的数据采样必须遵从以下两个基本原则:一是在有效时钟沿到达前,数据输入至少已经稳定了采样寄存器的 Setup 时间之久,这条原则简称满足 Setup 时间准则;二是在有效时钟沿到达之后,数据输入至少还将稳定保持采样寄存器的 Hold 时间之久,这条原则简称满足 Hold 时间原则。

如果所有寄存器的时序要求都能满足,则同步时序设计与异步时序设计相比,在不同的 PVT 条件下,能获得更佳的系统稳定性与可靠性。

2.4 Verilog HDL 硬件设计语言

数字集成电路在过去 30 年里得到了长足的发展,EDA(电子设计自动化)技术起到了至关重要的作用。其中,用于表达设计对象的硬件描述语言(HDL)采用形式化方法,不仅可以准确、直观地对数字电路进行建模和仿真,而且极大提高了电子设计的效率和产出。为了适应目前系统芯片(System-on-a-Chip, SOC)时代的设计需要,提升设计能力和效率,国内推广和开展基于 HDL 和各种 EDA 工具的设计方法以及成为迫切需要。

Verilog HDL 是在 C 语言的基础上发展起来的一种硬件描述语言,它是由 GDA(Gateway Design Automation)公司的 Phil Moorby 在 1983 年末首创的,最初只设计了一个仿真与验证工具,之后又陆续开发了相关的故障模拟与时序分析工具。1985 年 Moorby 推出它的第三个商用仿真器 Verilog-XL,获得了巨大的成功,从而使得 Verilog HDL 迅速得到推广应用。CADENCE 公司收购了 GDA 公司后,于 1990 年公开发表了 Verilog HDL,并成立 LVI 组织以促进 Verilog HDL 成为 IEEE 标准,即 IEEE Standard 1364-1995。

Verilog 作为当今国际主流的硬件描述语言,在数字电路和芯片的前端设计中得到了广泛的应用。通过 Verilog HDL,设计者既能用结构描述,又能用高级行为

描述来创建和使用模块，可以使设计者在整个设计过程的不同阶段(从结构方案的分析比较，直到物理器件的实现)，均能使用不同级别的抽象。

2.5 本章小结

本章首先给出系统开发的软件工具 Quartus II 和它的优点，然后介绍了系统开发的硬件平台，包括选用的 FPGA 芯片的技术规格，图像采集，图像显示与网卡芯片等。最后详细阐述了 FPGA 的设计方法、设计流程，并简要介绍设计使用的 Verilog HDL 硬件描述语言。

3 JPEG 图像压缩技术

3.1 JPEG 编码的技术标准

3.1.1 JPEG 标准的组成

JPEG 的“连续色调静止图像的数字压缩和编码 (Digital Compression and Coding of Continuous-tone Still Images)”标准是由几个部分组成^[9]。它们是:

ISO/IEC 10918-1: 1994|ITU-T T.81 信息技术 -- 连续色调静止图像的数字压缩和编码: 需求和指导方针 (Information technology -- Digital compression and coding of continuous-tone still images: Requirements and guidelines)。

ISO/IEC 10918-2: 1995|ITU-T T.83 信息技术 -- 连续色调静止图像的数字压缩和编码: 依从测试 (Information technology -- Digital compression and coding of continuous-tone still images: Compliance testing)。

ISO/IEC 10918-3: 1997|ITU-T T.84 信息技术 -- 连续色调静止图像的数字压缩和编码: 扩展 (Information technology -- Digital compression and coding of continuous-tone still images: Extensions)。

ISO/IEC 10918-3: 1997|Amd 1: 1999 Provisions to allow registration of new compression types and versions in the SPIFF header。

ISO/IEC 10918-4: 1999|ITU-T T.86 信息技术 -- 连续色调静止图像的数字压缩和编码: JPEG 参数、轮廓、标签、颜色空间、APPn 标记、压缩类型和注册机构 (REGAUT) 的注册 (Information technology -- Digital compression and coding of continuous-tone still images: Registration of JPEG profiles, SPIFF profiles, SPIFF tags, SPIFF colour spaces, APPn markers, SPIFF compression types and Registration Authorities (REGAUT))。

JPEG 标准确定的目标是:

①达到 (近乎) 完美的图像质量。

②可以压缩任何连续色调的静止图片, 包括灰度和色彩, 任意的色彩空间和大多数尺寸。

③可适用于大部分通用的计算机平台, 硬件实现条件适中。

3.1.2 JPEG 标准的层次

JPEG 标准定义了三个层次^[10]:

①基本系统:

每个编码解码器必须实现一个称为基本顺序编码器的必备基本系统。基本系统必须合理地压缩/解压彩色图像,保持高压缩率,并能处理从 4 位/像素和 16 位/像素的图像。

②扩展系统:

扩展系统包括了各种的编码方式,如长度可变编码、累进编码,以及分层模式的编码。所有这些编码方法都是基本顺序编码方法的扩展,这些特殊用途的扩展可适用各种应用。

③特殊无损功能:

特殊无损功能(也称作预测无损编码法)确保在图像被压缩的分辨率下,解压缩不造成初始源数字图像中任何细节的损失。

JPEG 花费了大量的时间,致力于图像的压缩和实现。他们在思维上创新并且拥有精湛的技术,终于使 JPEG 静止图片压缩技术成为一种最广泛认可的标准。JPEG 的基本压缩方式已成为一种通用的技术,很多应用程序都采用了与之相配套的软硬件。JPEG 的算法平均压缩比为 15:1。当压缩比大于 50 倍时将可能出现方块效应。它的性能,用质量与比特率之比来衡量,是相当优越的。

3.1.3 JPEG 标准的工作模式

JPEG 是假想为适用范围非常广泛,通用性很强的技术,所以按照算法的功能分为四种工作模式,用户只需要从中选择需要的功能即可。这四种工作模式是:

①基于 DCT (Sequential DCT-based) 的顺序模式:

由 DCT (离散余弦变换)系数的形成、量化和熵编码三步组成。从左到右进行编码处理并且从上到下顺序进行扫描图像。编码的基本单元为 8×8 的像素块。

②基于 DCT (Progressive DCT-based) 的扩展模式:

生成 DCT 系数和量化中的关键步骤与基于 DCT 的顺序编码解码器相同。主要的区别在于每个图像部件由多次扫描进行编码而不是仅一次扫描。每次继续的扫描都对图像作了改善,直到达到由量化表建立的图像质量为止。

③无失真 (Lossless) 模式:

独立于 DCT 处理,用来定义一种达到无损连续色调压缩的手段。预测器将采样区域组合起来并基于采样区域预测出邻系统区域。预测出的区域对照着每一区域的完全无损采样进行预测,同时通过 Huffman 编码法或算术熵编码法对这一差别进行无损编码,对较好质量的复制通常可达到 2:1 的压缩率。

④分层 (Hierarchical) 模式:

分层模式提供了一种可实现多种分辨率的手段。每个接续层次上的图像编码在水平或垂直方向上的分辨率都被降低二倍。它所传送的数据包括所支持的最低分辨率图像,以及用于解码恢复到原有的全分辨率图像所需的、分辨率以 2 的倍

数递减的相邻图像的差分信息。

3.1.4 JPEG 的编码处理过程

JPEG 的编码处理（如图 3.1）包括：图像准备，图像处理，量化，和熵编码四个过程。

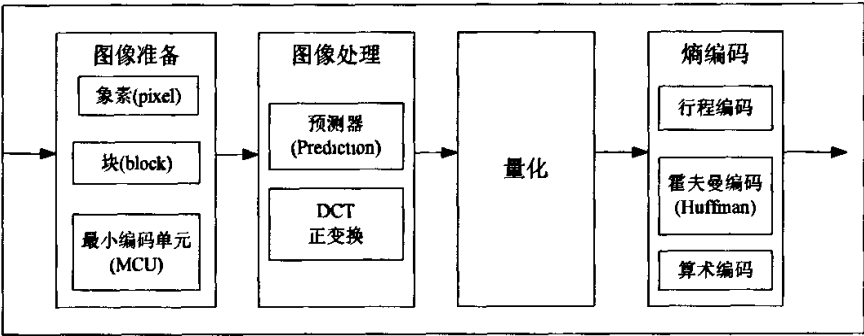


图 3.1 JPEG 编码处理过程
Fig 3.1 process of JPEG encoding

其中，量化后的熵编码部分在基本系统中可以使用霍夫曼编码，在扩展系统中可以使用除霍夫曼编码外的其他算术编码。算术编码于霍夫曼编码相比，压缩率稍微要高一些，但是相应地处理也要复杂。

3.2 JPEG 压缩的基本原理

JPEG 基本的编码器和解码器结构如图

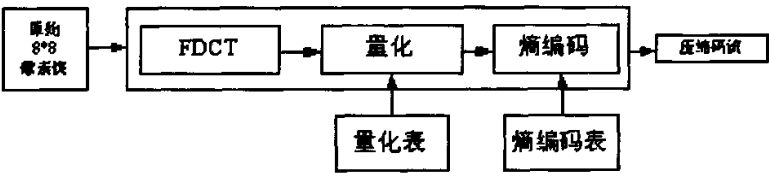


图 3.2a JPEG 编码器
Fig 3.2a JPEG encoder

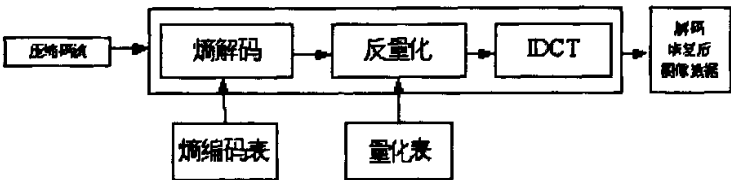


图 3.2b JPEG 编解码器
Fig 3.2b JPEG decoder

3.2.1 8×8 的 FDCT 和 IDCT

DCT 是变换编码中用得最多的一种压缩变换方法。由于余弦变换的基本图像是中心对称,且在边界处是连续的,所以利用余弦编码的压缩编码可以大大提高数据压缩比。

8×8 的前向 DCT(FDCT)和反向 DCT(IDCT)的算式如下^[11]:

前向 DCT 变换:

$$S_{vu} = \frac{1}{4} C_u C_v \sum_{x=0}^7 \sum_{y=0}^7 s_{yx} \cos \frac{(2x+1)u\pi}{16} \times \cos \frac{(2y+1)v\pi}{16} \quad (\text{式 3.1})$$

逆向 DCT 变换:

$$s_{yx} = \frac{1}{4} \sum_{x=0}^7 \sum_{y=0}^7 C_u C_v S_{vu} \cos \frac{(2x+1)u\pi}{16} \times \cos \frac{(2y+1)v\pi}{16} \quad (\text{式 3.2})$$

其中,

$$C_u, C_v = \begin{cases} 1/\sqrt{2}, & \text{对于 } u, v=0 \\ 1, & \text{其他} \end{cases} \quad (\text{式 3.3})$$

$f(x,y)$ 和 $F(u,v)$ 的矩阵形式分别表示为

$$\{f(x,y)\} = \begin{bmatrix} f(0,0) & f(0,1) & \dots & f(0,7) \\ f(1,0) & f(1,1) & \dots & f(1,7) \\ \dots & \dots & \dots & \dots \\ f(7,0) & f(7,1) & \dots & f(7,7) \end{bmatrix} \quad \{F(u,v)\} = \begin{bmatrix} F(0,0) & F(0,1) & \dots & F(0,7) \\ F(1,0) & F(1,1) & \dots & F(1,7) \\ \dots & \dots & \dots & \dots \\ F(7,0) & F(7,1) & \dots & F(7,7) \end{bmatrix}$$

这样,每一个 8×8 的数据单元通过前向 DCT 变换成 64 个 DCT 系数,其中包含一个直流分量 DC 系数(即 $F(0,0)$)和 63 个交流 AC 系数。JPEG 标准是开放式的,可以采用任何一种快速 DCT 算法。在 JPEG 基本系统中, $f(x,y)$ 为 8bit 像素,取值范围为 0~255,由此可以求出 DC 系数 $F(0,0)$ 的取值范围为 0~2040。实际上, $F(0,0)$ 是图像均值的 8 倍^[12]。

3.2.2 量化

8×8 的图象经过 DCT 变换后,其低频分量都集中在左上角,高频分量分布在右下角(DCT 变换实际上是空间域的低通滤波器)。由于人眼对高频分量远没有对低频分量敏感,大量的图像信息(如亮度)主要包含在低频中,所以编码时,可以忽略图像的高频分量,从而在视觉损失很小的情况下达到压缩的目的。

如何将高频分量去掉,这就要用到量化,它是产生信息损失的根源。这里的量化操作,就是将 8×8 像素块中的像素值除以量化表中对应的量化系数。由于量化表左上角的值较小,右上角的值较大,这样就起到了保持低频分量,抑制高频分量的目的。JPEG 使用的颜色是 YUV 格式。Y 分量代表了亮度信息,UV 分量代表了色差信息。相比而言,Y 分量更重要一些,在设计的时候可以对 Y 采用细量化,

对 UV 采用粗量化, 进一步提高压缩比^[13]。所以量化表通常有两张, 一张是针对 Y 的, 一张是针对 UV 的。

表 3.1a 亮度量化表

Table 3.1a Quantify table for luminate recommended by JPEG							
16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

表 3.1b 色度量化表

Table 3.1b Quantify table for color recommended by JPEG							
17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

这两个量化表均基于两个因素：一是人的视觉心理阈值；二是对于大量图像的观测,适应 8bit 精度,水平方向进行 2:1 抽样的图像。目前在许多一般应用的 JPEG 基本系统中均使用这两个量化表。

需要注意的是,JPEG 图像压缩的压缩比和图象质量是呈反比的,下表是压缩效率与图象质量之间的大致关系。在实际应用中,可以根据各自的需要,选择合适的压缩比。

表 3.2 压缩比与图像质量关系

Table 3.2 the relationship of compression rate and image quality	
压缩效率(单位: bits/pixel)	图像质量
0.25~0.50	中~好,可以满足某些应用
0.50~0.75	好~很好,可以满足多数应用
0.75~1.50	极好,可以满足大多数应用
1.50~2.00	与原始图像几乎一样

3.2.3 熵编码

经过 DCT 变换后，低频分量集中在左上角，其中 $F(0, 0)$ 代表了直流(DC)系数，即 8×8 子块的平均值，要对它单独编码。由于两个相邻的 8×8 子块的 DC 系数相差很小，所以对它们采用差分编码 DPCM，可以提高压缩比，也就是说对相邻的子块 DC 系数的差值进行编码。 8×8 的其它 63 个元素是交流(AC)系数，采用行程编码。为了保证低频分量先出现，高频分量后出现，以增加行程中连续“0”的个数，这 63 个元素采用了“之”字型(Zig-Zag)的排列方法^[14]，如图 3.3 所示。

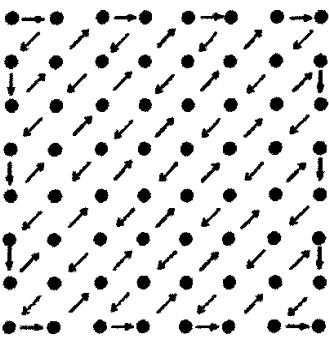


图 3.3 之字型扫描
Fig 3.3 Zig-Zag

这 63 个 AC 系数行程编码的码字表示方法如图 3.4:



图 3.4 行程编码

Fig 3.4 Runlength encoding

根据以上规则，我们得到了 DC 码字和 AC 行程码字。为了进一步提高压缩比，需要对其再进行熵编码，这里选用 Huffman 编码，分成两步：

1) 熵编码的中间格式^[15]

表示对于 AC 系数，有两个符号。符号 1 为行程和尺寸，即上面的(RunLength, Size)。(0, 0)和(15, 0)是两个比较特殊的情况。(0, 0)表示块结束标志(EOB)，(15, 0)表示 ZRL，当行程长度超过 15 时，用增加 ZRL 的个数来解决，所以最多有三个 ZRL($3\times 16+15=63$)。符号 2 为幅度值(Amplitude)。对于 DC 系数，也有两个符号，其中符号 1 为 DC 系数的尺寸(Size)，符号 2 为幅度值(Amplitude)。

2) 熵编码

对于 AC 系数, 符号 1 和符号 2 分别进行编码。零行程长度超过 15 个时, 前面每 16 个连续的零用一个符号(15, 0)表示, 块结束时也只有一个符号(0, 0)。对符号 1 进行 Huffman 编码(Y 和 UV 的 Huffman 码表不同)。对符号 2 进行变长整数 VLI 编码。举例来说: Size=6 时, Amplitude 的范围是-63~-32, 以及 32~63, 对绝对值相同, 符号相反的码字之间为反码关系。所以 AC 系数为 32 的码字为 100000, 33 的码字为 100001, -32 的码字为 011111, -33 的码字为 011110。符号 2 的码字紧接于符号 1 的码字之后。对于 DC 系数, Y 和 UV 的 Huffman 码表也不同。

由于 Huffman 是一种基于统计的无损变长编码, 出现频率最高的数值码字长度最短, 使平均码字长度接近信源熵值, 减少了数据冗余量。

3.3 本章小结

本章首先介绍了 JPEG 压缩的技术标准, 然后详细阐述 JPEG 压缩的基本原理, 包括 DCT 变换将信号的高频和低频分离, 量化将高频信号进一步压缩, 以及 DC 系数使用预测编码 DPCM, AC 系数使用变换编码, 二者都使用熵编码 Huffman。可见几乎所有传统的压缩方法在 JPEG 中都用到了。这几种方法的结合正是产生 JPEG 高压缩比的原因。实际上, 该标准是 JPEG 小组从很多种不同中方案中比较测试得到的, 必然具有很高的压缩效率。

4 JPEG 实时图像编解码系统的硬件设计

4.1 系统的总体方案和工作原理

4.1.1 总体方案

如图 4.1 所示, 系统主要由两大部分构成: 图像编码服务器端和图像解码客户端。其中图像编码服务器端负责对图像进行实时采集, 经过 JPEG 编码, 通过网络传送到客户端; 图像解码客户端接收到服务器发送过来的压缩码流, 对它进行 JPEG 解码, 然后由 VGA 显示器显示出恢复后的图像。服务器端规定, 连接的客户端数目最大不超过 5 个。

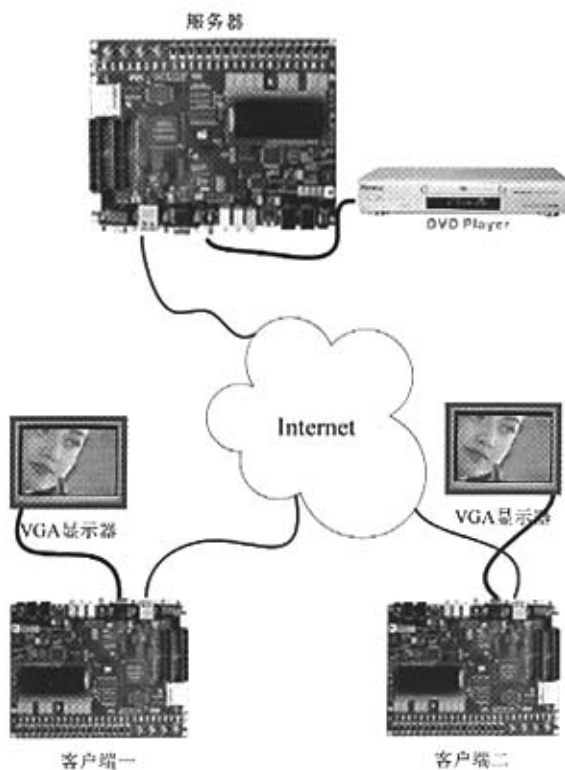


图 4.1 系统总体方案图

Fig 4.1 schedule of the system

4.1.2 工作原理

① JPEG 编码服务器端:

系统上电后, Nios 对它控制下的图像采集模块, JPEG 压缩模块以及网络传输模块进行配置, 其中包括采集来的原始图像存储的首地址, 图像处理缓存区首地

址和压缩后图像存储的首地址等；然后，网络模块检查是否有客户端请求连接，并且计算已连接客户端是否已经到了可连接的最大数目，如果没有达到最大连接数，则 CPU 产生一个线程，建立与客户端的连接，并通知图像采集模块开始采集；当图像采集模块采集完一帧图像后，向 CPU 发送中断，通知对原始图像进行编码，同样一帧图像编码完毕后，JPEG 压缩模块会向 CPU 发送中断，通知网络传输模块将压缩码流传送至客户端。

② JPEG 解码客户端：

客户端的运行过程与服务器成对称关系。当 CPU 完成一系列配置后，会向服务器端发起连接请求；请求被接受，客户端接收服务器传送过来的压缩码流，一帧数据发送完毕后，发送中断；解码模块接收到 CPU 发送的信号，开始解码，当解码出完整的一帧图像，模块向 CPU 发送中断；显示模块在得到第一帧恢复后数据后开始显示，每显示完一帧图像，进入场消隐时间，它会检查解码模块是否有恢复出新的图像，如果有，则下一帧显示新的图像，没有，则下一帧显示前一帧图像。

4.2 JPEG 编码的设计和实现

在本系统中，笔者选用的是 JPEG 的基本编码系统，即基于 DCT 变换的 JPEG 有损压缩。设计从整体功能上，划分为 2D-DCT、量化与霍夫曼编码三大模块。

4.2.1 2D-DCT 模块的设计

2D-DCT 变换的原理在上一章已经得到了充分阐述。如何实现 2D-DCT 变换，一直是 JPEG 编码的难点所在。如果直接按照 2D-DCT 变换的原始公式（式 3.1）来设计，那么计算一个 8×8 像素块将需要 4096 次乘法和 4096 次加法，计算非常复杂，耗费的资源和时间在实际应用中都难以接受，必须采用快速算法。

考虑到 2D-DCT 变换的正交特性和对称性，人们提出了许多快速 2D-DCT 算法。在这些快速算法中，一类是利用蝶形快速算法来计算 $DCT^{[16-17]}$ ，另一类是直接根据 DCT 的规律寻求快速算法^[18-21]。在第二类算法中，最常用的是采用行列快速算法，也就是我们通常所说的陈氏算法^[22]。该算法首先逐行计算一维 DCT，再逐列计算一维 DCT，从而把乘法的计算量减少了一半，其算法规律性强，特别适合 FPGA 来实现 2D-DCT 变换。

对于矩阵形式的一维 DCT 变换 $Y=CX$ ，以 8 点输入为例，其矩阵展开式如下：

$$\begin{pmatrix} y0 \\ y1 \\ y2 \\ y3 \\ y4 \\ y5 \\ y6 \\ y7 \end{pmatrix} = \begin{pmatrix} C0 & C0 & C0 & C0 & C0 & C0 & C0 & C0 \\ C1 & C3 & C5 & C7 & -C7 & -C5 & -C3 & -C0 \\ C2 & C6 & -C6 & -C2 & -C2 & -C6 & C6 & C2 \\ C3 & -C7 & -C1 & -C5 & -C5 & C1 & C7 & C3 \\ C4 & -C4 & -C4 & C4 & C4 & -C4 & -C4 & C4 \\ C5 & -C1 & C7 & C3 & -C3 & -C7 & C1 & -C5 \\ C6 & -C2 & C2 & -C6 & -C6 & C2 & -C2 & C6 \\ C7 & -C5 & C3 & -C1 & C1 & -C3 & C5 & -C7 \end{pmatrix} \begin{pmatrix} x0 \\ x1 \\ x2 \\ x3 \\ x4 \\ x5 \\ x6 \\ x7 \end{pmatrix} \quad (\text{式 4.1})$$

其中:

$$\begin{pmatrix} C0 \\ C1 \\ C2 \\ C3 \\ C4 \\ C5 \\ C6 \\ C7 \end{pmatrix} = \begin{pmatrix} \cos(4\pi/16) \\ \cos(\pi/16) \\ \cos(2\pi/16) \\ \cos(3\pi/16) \\ \cos(4\pi/16) \\ \cos(5\pi/16) \\ \cos(6\pi/16) \\ \cos(7\pi/16) \end{pmatrix}$$

观察发现, 系数矩阵 C 具有很特殊的对称性, 上式可以进一步化简, 如下式所示:

$$\begin{pmatrix} y0 \\ y2 \\ y4 \\ y6 \end{pmatrix} = \begin{pmatrix} C0 & C0 & C0 & C0 \\ C2 & C6 & -C6 & -C2 \\ C4 & -C4 & -C4 & C4 \\ C6 & -C2 & C2 & -C6 \end{pmatrix} \begin{pmatrix} x0+x7 \\ x1+x6 \\ x2+x5 \\ x3+x4 \end{pmatrix} \quad (\text{式 4.2})$$

$$\begin{pmatrix} y1 \\ y3 \\ y5 \\ y7 \end{pmatrix} = \begin{pmatrix} C1 & C3 & C5 & C7 \\ -C3 & C7 & -C1 & -C5 \\ C5 & -C1 & C7 & C3 \\ C7 & -C5 & C3 & -C1 \end{pmatrix} \begin{pmatrix} x0-x7 \\ x1-x6 \\ x2-x5 \\ x3-x4 \end{pmatrix} \quad (\text{式 4.3})$$

由于系数全部是浮点数, 为了便于在 FPGA 里面实现乘法运算, 将它们扩大 8 倍后再取整, 代入上式, 得

$$\begin{pmatrix} y0 \\ y2 \\ y4 \\ y6 \end{pmatrix} = \begin{pmatrix} 91 & 91 & 91 & 91 \\ 118 & 50 & -50 & -118 \\ 91 & -91 & -91 & 91 \\ 50 & -118 & 118 & -50 \end{pmatrix} \begin{pmatrix} x0+x7 \\ x1+x6 \\ x2+x5 \\ x3+x4 \end{pmatrix} \quad (\text{式 4.4})$$

$$\begin{pmatrix} y1 \\ y3 \\ y5 \\ y7 \end{pmatrix} = \begin{pmatrix} 126 & 106 & 71 & 25 \\ -106 & 25 & -126 & -71 \\ 71 & -126 & 25 & 106 \\ 25 & -71 & 106 & -126 \end{pmatrix} \begin{pmatrix} x0-x7 \\ x1-x6 \\ x2-x5 \\ x3-x4 \end{pmatrix} \quad (\text{式 4.5})$$

从上式看出,用 FPGA 实现陈氏快速 DCT 算法,一个 8 维的矩阵进行 1D-DCT 变换,需要 22 个整数乘法器,那么一个 2D-DCT 变换需要行变换和列变换两个 1D-DCT 变换模块,即最少需要 44 个乘法器。而本系统选用的 Cyclone II EP2C35F672 片上包含的 18×18 bits 乘法器一共只有 35 个,显然乘法器资源不够。如果使用逻辑资源来自行搭建乘法器,则有可能占用过多的逻辑资源,而且多位乘法器的处理速度也非常低,满足不了系统处理速度的要求。

为此笔者提出用三个缓存器加一个 1D-DCT 模块实现 2D-DCT 的新方法,总体结构如图 4.2 所示。切换模块负责对 1D-DCT 的输入输出数据进行控制,第一次 1D-DCT 的数据输入初始缓存 RAM 的值,即原始图像数据,经过 1D-DCT 模块作行变换后,转置存入行变换后缓存 RAM;然后数据切换模块再把行变换后缓存 RAM 里面的数据输入 1D-DCT 模块,相当于进行 1D-DCT 列变换,输出的值转置存入列变换后缓存 RAM,得到的即为 2D-DCT 处理后数据。

三个缓存 RAM 分别存储的是 8×8 像素单元,占用的是相对比较丰富的芯片存储器资源,资源消耗比 1D-DCT 硬件模块要少得多,这样可以节省大量逻辑单元。并且由于每进行一次 DCT 变换后都对结果数据进行存储,可以保证下一步操作时,输入的数据不会受到部分线路的延迟或者某些信号的干扰而导致数据错位。

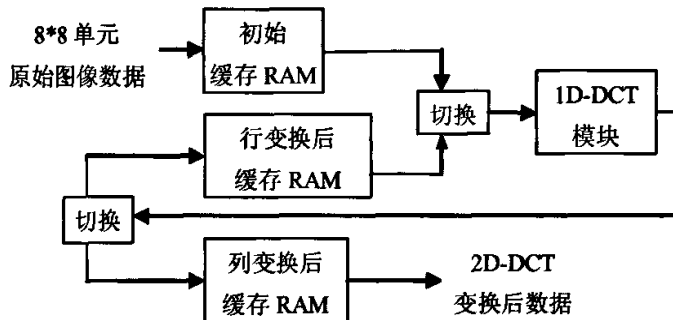


图 4.2 2D-DCT 硬件模块结构

Fig 4.2 hardware structure of 2D-DCT

在 Quartus 下面得到 2D-DCT 模块的原理框图如图 4.3 所示:

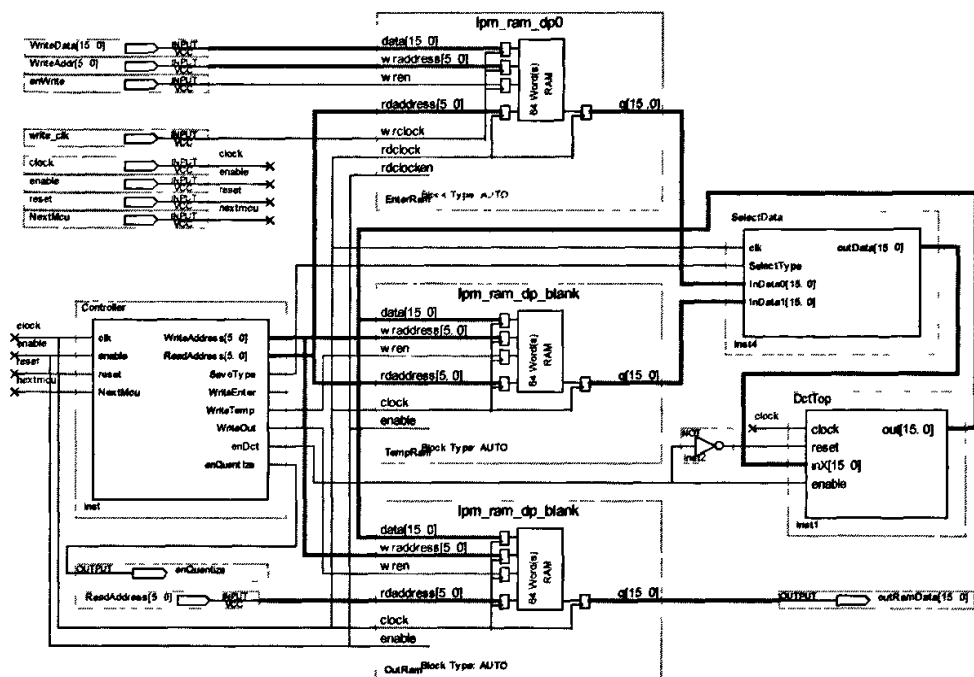


图 4.3 2D-DCT 原理图

Fig 4.3 Hardware Module of 2D-DCT

经过编译综合，得到的报告资源消耗报告如图 4.4 所示。可以看出，2D-DCT 模块占用的逻辑资源仅为芯片资源的 4%。

```

Flow Status                               Successful - Tue May 08 15:21:42 2007
Quartus II Version                       6.0 Build 178 04/27/2006 SJ Full Version
Revision Name                            DE2_NIOS
Top-level Entity Name                    dcd
Family                                   Cyclone II
Device                                   EP2C35F672C6
Timing Models                            Final
Met timing requirements                   Yes
Total logic elements                     1,302 / 33,216 ( 4 % )
Total registers                          982
Total pins                               51 / 475 ( 11 % )
Total virtual pins                       0
Total memory bits                        3,072 / 483,840 ( < 1 % )
Embedded Multiplier 9-bit elements       56 / 70 ( 80 % )
Total FLLs                              0 / 4 ( 0 % )

```

图 4.4 2D-DCT 编译报告

Fig 4.4 Compile report of 2D-DCT module

系统时钟设置为 100MHZ 时, 得到的仿真波形如下图 4.5a, 4.5b 所示:

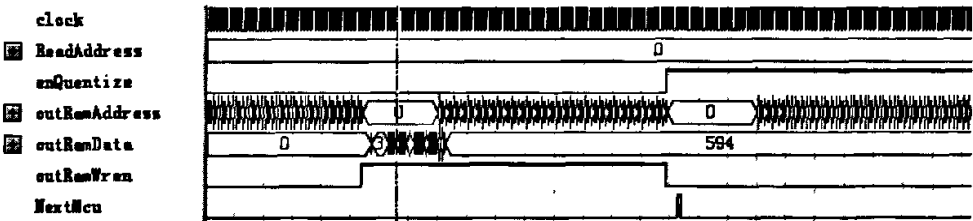


图 4.5a 2D-DCT 仿真波形

Fig 4.5a simulating waves of 2D-DCT

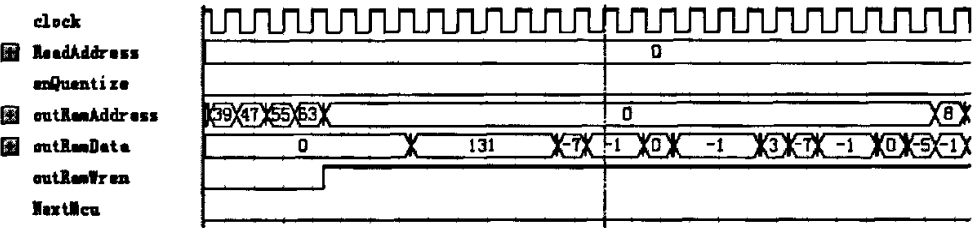


图 4.5b 2D-DCT 仿真波形细节

Fig 4.5b details of 2D-DCT simulating

本 2D-DCT 模块采用 FPGA 实现，与 PC 机上软件实现的处理速度比较如表：

表 4.1 2D-DCT 软硬件处理时间对比

Table 4.1 contrary of 2D-DCT processing time by HW and SW

实现方式	FPGA 硬件	PC 机软件
速度 us/MCU	3.46	72

4.2.2 量化模块的设计

JPEG 量化的原理是通过去掉视觉上不太重要的图像信息来提高压缩效率。在 DCT 变换后，每个 DCT 系数的量化系数是由 DCT 系数除以规定的量化步长 (step size), 然后四舍五入求出整数值。具体实现上就是把 8×8 模块上的每一个值除以量化表中对应的一个数值。

量化步长的最佳值是由输入图像及图像显示设备的特性来决定的，在 JPEG 中没有规定具体的数值^[23]。在设计中，笔者经过比较，把 JPEG 推荐的量化表中的数值分别除以 2，得到修正的量化表来代替原来的量化表。由于量化基数变小，量化的结果就会相应放大，可以保留更多图像细节，提高图像质量，但同时压缩比会降低。此外，由于 FPGA 的除法器会直接截除运算结果的小数位，在设计中还给

出了一个量化修补值，大小为修正后量化表中对应数值的 1/2。对于正的被量化值，加上修补值，对于负的被量化值，减去修补值，实际上就相当于把量化后的数据进行四舍五入处理。

经过编译综合，得到的报告资源消耗报告如图 4.6 所示。

Flow Status	Successful - Tue May 08 15:13:55 2007
Quartus II Version	6.0 Build 178 04/27/2006 SJ Full Version
Revision Name	HE2_NIOS
Top-level Entity Name	Quantize
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Met timing requirements	Yes
Total logic elements	557 / 33,216 (2 %)
Total registers	371
Total pins	49 / 475 (10 %)
Total virtual pins	0
Total memory bits	1,024 / 483,840 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

图 4.6 量化编译报告

Fig 4.6 Compile report of Quantization module

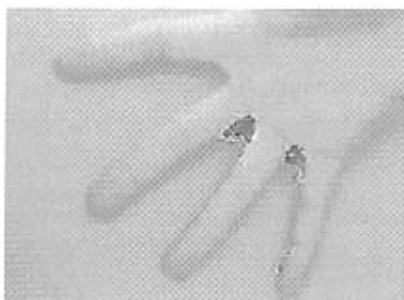
本量化模块采用 FPGA 实现，与 PC 机上软件实现的处理速度比较如表：

表 4.2 量化软硬件处理时间对比

Table 4.2 contrary of Quantization processing time by HW and SW

实现方式	FPGA 硬件	PC 机软件
速度 us/MCU	1.5	15

在其他模块相同的条件下，不进行量化修正和进行量化修正两种情况，得到的结果如图 4.7a、图 4.7b 所示，由此表明，通过对量化值进行修正，图像恢复的质量有较大的提高。



4.7a 未进行量化修正的图像效果

Fig4.7a image after JPEG decoding
without quantification amendment



4.7b 进行量化修正的图像效果

Fig4.7b image after JPEG decoding
with quantification amendment

4.2.3 熵编码模块的设计

熵编码包括游程编码和霍夫曼编码两个部分的设计：

①游程编码：

经过量化处理，图像的 AC 分量很多都变为零。游程编码通过 Z 字形扫描，可以把连续的零值 AC 分量用一个 AC 系数来表示，这对编码的压缩效率非常有利。

②霍夫曼编码

霍夫曼编码就是根据 DC 系数的码表和 AC 系数的码表，查出对应的二进制码表示方法，再对码流进行拼接。因此，霍夫曼编码模块里面很重要的一个操作就是查表。在系统中，笔者将 AC 系数与 DC 系数的霍夫曼表预先存储在片上 RAM 中，便于编码模块进行查找。

1) DC 系数的编码

在本系统中，DC 系数采用的是直接编码，省略了对码值进行预测的部分，减少了对前一 DC 系数的读取步骤，虽然损失了一部分压缩比，但是模块的复杂程度降低，增强了系统的稳定性，并且使系统最高频率 f_{MAX} 得到了提高。

进行 DC 编码时，首先求出 DC 系数对应的 SSSS(Size of Coefficient)。只用 SSSS 对应的霍夫曼编码，还不能具体表示组内的 DC 值，必须增加补充比特来指定 DC 的值。DC 用二进制表示时，最高位的表示的是 LSB(Least Significant Bit)。DC 为正数时，从 LSB 开始的 SSSS 比特，DC 为负数时，从 DC-1 的 LSB 开始的 SSSS 比特为补充比特。因此，在解码器中，补充比特的最高位为 1 时，就可以判断 DC 为正数，为 0 时，DC 为负数。DC 系数的霍夫曼编码首先就是用符号指定 DC 值大小的范畴，然后用补充比特来指定范畴内具体的 DC 值，DC 霍夫曼编码=范畴的霍夫曼编码+补充比特。

表 4.3 DC 系数(亮度分量)霍夫曼编码和补充比特^[24]

Table 4.3 Huffman codes and amendment bits of DC coefficient (for Luminance)

DC	SSSS	霍夫曼编码	补充比特
-2047...-1024,1024...2047	11	111111110	00000000000...01111111111,10000000000,11111111111
-1023...-512,512...1023	10	111111110	0000000000...01111111111,1000000000...1111111111
-511...-256,256...511	9	111111110	000000000...0111111111,100000000...111111111
-255...-128,128...255	8	11111110	00000000...01111111,10000000...11111111
-127...-64,64...127	7	111110	0000000...0111111,1000000...1111111
-63...-32, 32...63	6	11110	000000...011111,100000...111111
-31...-16, 16...31	5	1110	00000...01111,10000...11111
-15...-8, 8...15	4	101	0000...0111,1000...1111
-7...-4, 4...7	3	100	000...011,100...111
-3, -2, 2, 3	2	011	00, 01, 10, 11
-1, 1	1	010	0, 1
0	0	00	无

2) AC 系数的编码

AC 系数的编码是非零的 AC 系数和其非零系数前连续为零的 AC 系数的个数组合构成的长度可变的编码，并且与 DC 系数编码一样要添加补充比特。具体的编码顺序是，首先从第一个 AC 系数开始，按顺序观察一遍，如果存在非零的 AC 系数，将其前面零值 AC 系数的个数表示为四比特的 RRRR(Runlength)(因此， $0 \leq RRRR \leq 15$)，并且对应现在的非零的 AC 系数的范畴表示为四比特的 SSSS(求出 SSSS 的方法与 DC 编码相同)，使用这些组合在霍夫曼编码表中查出对应的霍夫曼编码。

AC 系数处理有两个特别的情况。一是当 AC 系数具有 16 个以上连续零值时，用 RRRR/SSSS=15/0 来表示连续 16 个零，当剩余的连续零值小于 16 个时，则按照前面的方法表示；二是当最后一个 AC 系数为零时，要将块末尾剩下的零值 AC 系数集合起来，用 RRRR/SSSS=0/0 表示，输出对应的编码后，立即结束该块的编码处理。但块内最后的系数为非零值时，不可以使用 EOB。

表 4.4 AC 系数(亮度分量)霍夫曼编码表^[25]

Table 4.4 Huffman codes of AC coefficient (for Luminance)

SSSS RRRR	0	1	2	...	10
0	1010(EOB)	00	01	...	1111111110000011
1	无	1100	11011	...	1111111110001000
2		11100	11111001	...	1111111110001110
...	
15	11111111001(ZRL)	111111111110101	111111111110110	...	111111111111110

在对 RRRR/SSSS 编码后，还要添加具体指定的 AC 系数大小的补充码，其添加方式与 DC 系数相同。

由于单个 AC 系数或者 DC 系数霍夫曼编码结果最长位数为 24 位，为了便于操作，笔者采用了 32 位的码流输出，因为位数较长有利于提高存储速度。

霍夫曼编码的硬件模块框图如图 4.6 所示：

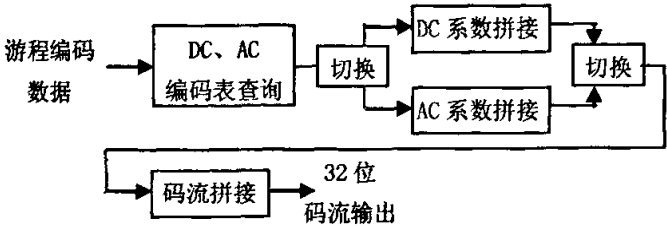


图 4.8 霍夫曼编码硬件模块结构

Fig 4.8 hardware structure of Huffman encodings

在这个硬件结构里，对时序要求最为严格的子模块就是码流拼接模块，它由三部分构成：拼接控制子模块，它负责新来码字和前面剩余码字各自需要移动的方向以及移动的位数；位移子模块，负责码字的位移；拼接子模块，将左移和右移后的两部分码字合并成一个整的 32 位码字输出。在 Quartus II 界面下，AC 系数拼接模块如图 4.7 所示，DC 系数拼接和 DC、AC 整体拼接的结构与 AC 系数拼接相同。

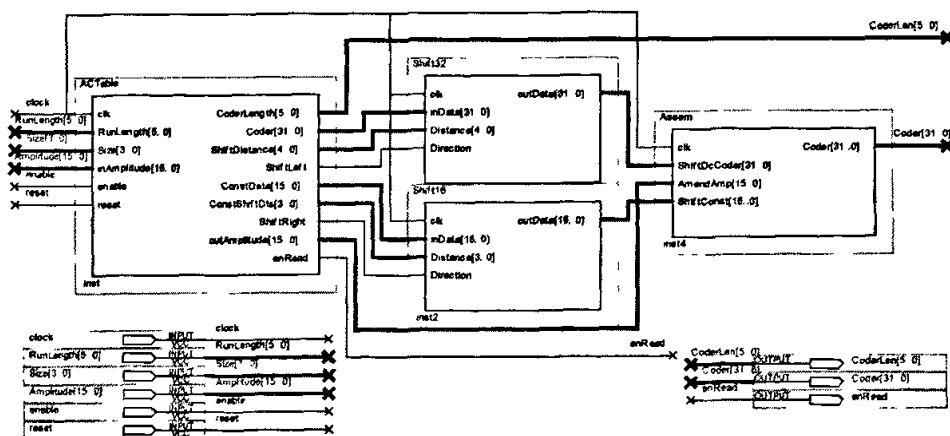


图 4.9 AC 系数拼接原理

Fig 4.9 hardware module of AC code-assembling

经过编译综合，得到的报告资源消耗报告如图 4.4 所示。

```

Fitter Status          Successful - Tue May 08 15:10:08 2007
Quartus II Version    6.0 Build 178 04/27/2006 SJ Full Version
Revision Name         DE2_NIOS
Top-level Entity Name  huffman
Family                Cyclone II
Device                EP2C35F672C8
Timing Models         Final
Total logic elements   1,887 / 33,216 ( 6 % )
Total registers        1037
Total pins             348 / 475 ( 73 % )
Total virtual pins     0
Total memory bits      2,138 / 483,840 ( < 1 % )
Embedded Multiplier 9-bit elements 0 / 70 ( 0 % )
Total PLLs             0 / 4 ( 0 % )

```

图 4.10 2D-DCT 编译报告

Fig 4.10 Compile report of 2D-DCT module

在 Quartus II 仿真霍夫曼编码模块, 得到的仿真波形如图 4.8 所示:

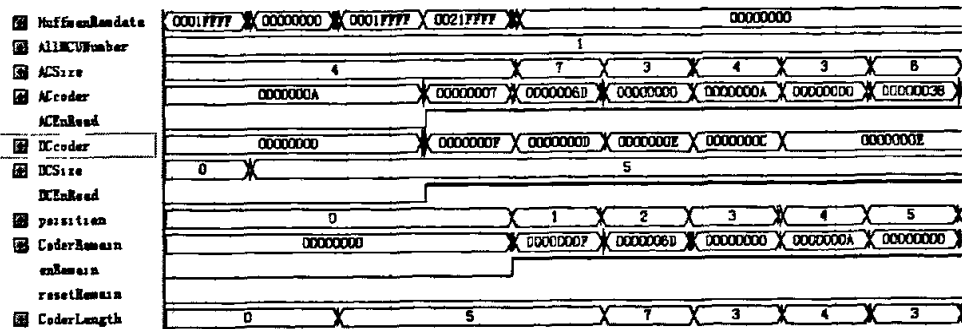


图 4.11 霍夫曼编码仿真波形

Fig 4.11 simulating waves of Huffman encoding

本量化模块采用 FPGA 实现，与 PC 机上软件实现的处理速度比较如表：

表 4.5 霍夫曼软硬件处理时间对比

Table 4.5 contrary of Huffman processing time by HW and SW

实现方式	FPGA 硬件	PC 机软件
速度 us/MCU	2.06	57

4.3 JPEG 解码的设计和实现

4.3.1 霍夫曼解码模块的设计

霍夫曼解码为编码的逆过程。首先来了解一下霍夫曼编码的数据结构^[26]：

DC 码字=SSSS(Size of DC)+DC 补充比特；

AC 码字=SSSS(Size of AC, 4bits)+RRRR(Runlength, 4bits)+AC 补充比特。

根据霍夫曼码字的结构，设计 DC 系数与 AC 系数的解码过程框架。

①DC 系数的解码

DC 系数的解码是由下面两个步骤组成的：

1) 从压缩数据（比特流）发现对应的霍夫曼编码并解码，得到编码要素中的 SSSS 范畴；

2) 继续从比特流中读取 SSSS 比特部分的补充比特，作为 DC 系数值；

②AC 系数的解码

AC 系数的解码比 DC 系数的解码要复杂些，是由五个步骤组成：

1)从比特流中发现对应的霍夫曼码字并解码，得到由零行程和非零 AC 系数范畴组合成的编码要素 RS；

2)从 RS 中得到 AC 系数范畴 SSSS 和零行程 RRRR；

3)如果 SSSS 为 0 且 RRRR 为 15，（即检测出 ZRL 情况），更新 16 个 AC 系数的地址后，返回第二步。这里，16 个 AC 系数都等于零；

4)如果 SSSS 为 0 且 RRRR 为 0, 即检测出 EOB 情况, 则终止该块的解码处理。这里, 剩下的所有 AC 系数等于零。

5)如果 SSSS 不为零, 则读出补充比特。如果这时候 DCT 系数的地址为 63 组, 则终止解码处理, 否则返回第二步。

从以上解码框架可以看出, 霍夫曼解码一个最重要的步骤就是查找霍夫曼表。如果采用最简单的方法, 即从比特流中读出每一比特, 参照编码器用霍夫曼表, 一个一个匹配比特组合, 处理的数据量相当庞大。因此, 系统按照霍夫曼编码的发生频率来准备排列后的解码器表, 以实现高效率的编码检索。

解码器表需要四种 (MINCODE, MAXCODE, VALPTR, HUFFVAL) 表, 这些表是基于编码器霍夫曼表按照下面的方法做出来的。首先, 在编码器中将使用的霍夫曼编码看成是由二进制表示的整数值, 其大小按照从小到大的顺序排列。基于这张表, 对比特长为 I 的霍夫曼编码, 作出 MINCODE(I), MAXCODE(I), VALPTR(I) 表。在这里, MINCODE(I), MAXCODE(I) 是 I 比特霍夫曼编码的最小值和最大值, VALPTR(I) 表示 MINCODE(I) 在大小顺序表中排第几位的指针。并且, 根据霍夫曼编码的大小顺序作出排列解码值的表 HUFFVAL(I)。从而可以在 HUFFVAL(I) 中用第 J 个中较小的解码值来指定霍夫曼编码表。表为 DC 系数的 MINCODE(I), MAXCODE(I), VALPTR(I), 而 AC 系数则还需要加制一张针对 SSSS 和 RRRR 的表。

表 4.6 AC 系数的 SSSS 与 RRRR 查找表

Table 4.6 SSSS & RRRR table of AC coefficient

I	VALPTR(I)	MINCODE(I)	MAXCODE(I)
1	—	—	1111111111111111
2	0	00	00
3	1	010	110
4	6	1110	1110
5	7	11110	11110
6	8	111110	111110
7	9	1111110	1111110
8	10	11111110	11111110
9	11	111111110	111111110

这时, 从比特流中查找出对应的霍夫曼编码的算法, 由以下三步构成:

a. 从比特流中的霍夫曼编码读 1 比特长的编码 CODE($I=1$);

b. CODE 与 MAXCODE(I) 比较大小, 如果 CODE < MAXCODE(I) 成立, 则表示要进行解码的霍夫曼编码的长度是 I 比特。这时解码值表 HUFFVAL(J) 的地址 J 是在 HUFFVAL 表中, 放置霍夫曼编码为 I 比特的解码器, 开始指针的 VALPTR(I)

与表示实际的霍夫曼编码在 I 比特编码中所占位置的 CODE-MINCODE(I)相加, 即由 $J = \text{VALPTR}(I) + \text{CODE-MINCODE}(I)$ 得到的。因此, 解码值就是 $\text{HUFFVAL}(J)$, 解码结束;

c. 当 $\text{CODE} > \text{MAXCODE}(I)$ 时, 因为要进行解码的霍夫曼编码的比特长在 I 比特以上, 从比特流中读取 I+1 比特, 且将 I+1 作为新的 I, 返回第二步。

而解码 AC 系数, 必须先按照 SSSS 和 RRRR 结合的 MINCODE(I), MAXCODE(I), VALPTR(I) 表解出相应的非零系数 Size 和零行程长度后, 再按照上述三步解码出 AC 系数的补充比特, 才算完整解码出 AC 系数。

在硬件实现上, 针对霍夫曼解码需要多个状态跳转的特点, 将霍夫曼解码模块设计成一个大的状态机, 它可以用算法状态机 ASM^[27] 来表示。算法状态机 (ASM) 类似软件流程, 但显示的是计算动作 (如寄存器操作) 的时间顺序, 以及在状态机输入影响下发生的时序步骤。

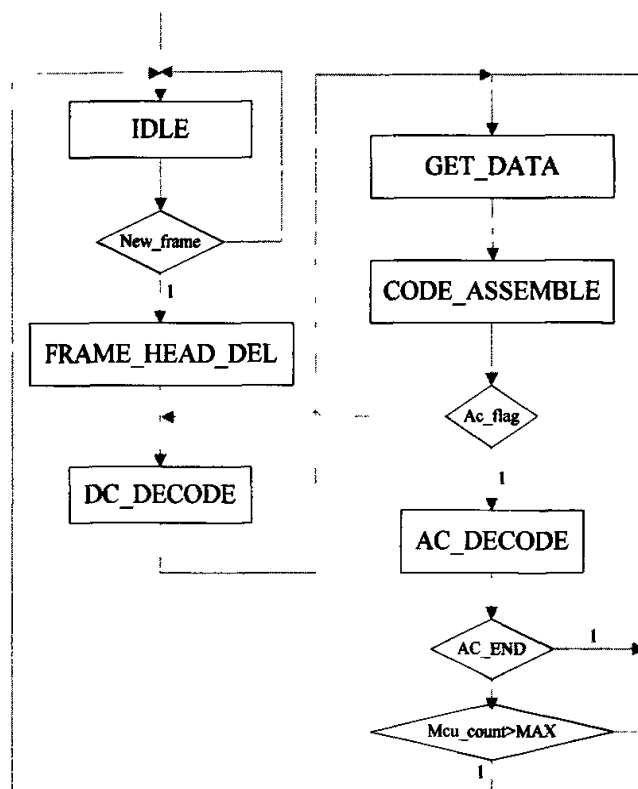


图 4.12 霍夫曼解码 ASM 图

Fig 4.12 ASM of Huffman Decoding

在 Quartus II 下经过编译综合, 得到的报告如图 4.13 所示:

```

Flow Status                               Successful - Tue May 08 15:00:52 2007
Quartus II Version                       6.0 Build 178 04/27/2006 SJ Full Version
Revision Name                            DE2_MIOS
Top-level Entity Name                    huffman_decoder
Family                                   Cyclone II
Device                                   EP2C35F672C6
Timing Models                            Final
Met timing requirements                   Yes
Total logic elements                     1,915 / 33,216 ( 6 % )
Total registers                          518
Total pins                               243 / 475 ( 51 % )
Total virtual pins                       0
Total memory bits                        3,968 / 483,840 ( < 1 % )
Embedded Multiplier 9-bit elements       0 / 70 ( 0 % )
Total PLLs                               0 / 4 ( 0 % )

```

图 4.13 霍夫曼解码编译报告

Fig 4.13 Compile report of Huffman decoder

系统时钟设置为 100MHZ 时, 得到的仿真波形如下图 4.14a, 4.14b 所示:

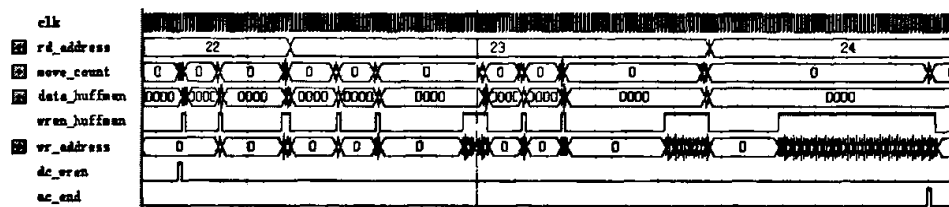


图 4.14a 霍夫曼解码仿真波形

Fig 4.14a simulate waves of Huffman decoding

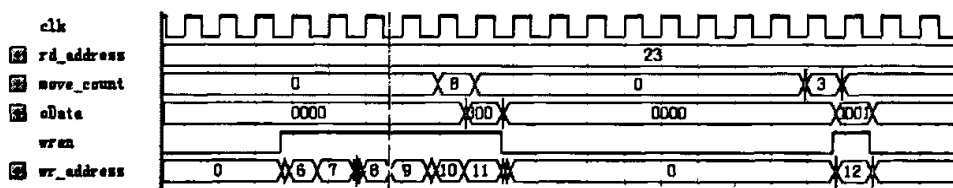


图 4.14b 霍夫曼解码仿真波形细节

Fig 4.14b detail waves of Huffman decoding

本霍夫曼解码模块采用 FPGA 实现,与 PC 机上软件实现的处理速度比较如表:

表 4.7 霍夫曼解码软硬件处理时间对比

Table 4.7 contrary of Huffman decoding time by HW and SW

实现方式	FPGA 硬件	PC 机软件
速度 us/MCU	4.6	362

4.3.2 2D-IDCT 模块的设计

2D-IDCT 与 2D-DCT 是一个对称的结构, 同样可以采用陈氏快速算法来实现 2D-IDCT 变换。

根据陈氏算法, 2D-IDCT 可以分解为:

$$\begin{pmatrix} x0 \\ x1 \\ x2 \\ x3 \end{pmatrix} = \frac{1}{2}P + \frac{1}{2}M \quad (\text{式 4.6}) \quad \begin{pmatrix} x7 \\ x6 \\ x5 \\ x4 \end{pmatrix} = \frac{1}{2}P - \frac{1}{2}M \quad (\text{式 4.7})$$

$$P = \begin{pmatrix} a & c & a & f \\ a & f & -a & -c \\ a & -f & -a & c \\ a & -c & a & -f \end{pmatrix} \begin{pmatrix} X0 \\ X2 \\ X4 \\ X6 \end{pmatrix} \quad (\text{式 4.8})$$

$$M = \begin{pmatrix} b & d & e & g \\ d & -g & -b & -e \\ e & -b & g & d \\ g & -e & d & -b \end{pmatrix} \begin{pmatrix} X1 \\ X3 \\ X5 \\ X7 \end{pmatrix} \quad (\text{式 4.9})$$

其中,

$$\begin{pmatrix} a & b & c & d & e & f & g \end{pmatrix}^T = \frac{1}{2} \left(\cos \frac{\pi}{4} \quad \cos \frac{\pi}{16} \quad \cos \frac{\pi}{8} \quad \cos \frac{3\pi}{16} \quad \cos \frac{5\pi}{16} \quad \cos \frac{3\pi}{8} \quad \cos \frac{7\pi}{16} \right)^T \quad (\text{式 4.10})$$

在设计中, 为了便于硬件实现, 将上式的余弦值换算成二进制, 然后左移 8 位, 相当于每个余弦值乘以 256, 将浮点型小数转变成 8 位整数, 再进行矩阵计算。

目前在对 IDCT 的计算中, 在上式中, P 和 M 的乘法累加一般都用 DA 算法^[28]实现。假设输入的 X_k 用二进制补码表示为: $X_k = -b_{k0} + \sum_{k=1}^K b_{kn} \cdot 2^{-n}$, b_{k0} 为符号位, $b_{k(N-1)}$ 为最低有效位 (LSB)。则乘法累加可表示如下:

$$y = \sum_{k=1}^K a_k \cdot X_k = \sum_{k=1}^K a_k \cdot (-b_{k0} + \sum_{k=1}^K b_{kn} \cdot 2^{-n}) = \sum_{n=1}^{N-1} \left[\sum_{k=1}^K a_k \cdot b_{kn} \right] \cdot 2^{-n} + \sum_{n=1}^{N-1} a_k \cdot (-b_{k0}) \quad (\text{式 4.11})$$

其中, a_k 为固定系数, 部分积 $\left[\sum_{k=1}^K a_k \cdot b_{kn} \right]$ 可预先计算并存放于 ROM 中, 用 ROM

累加器代替乘法器这样可大大降低硬件消耗。

但是,若将 DA 算法直接应用到本系统中,会带来一些问题:首先,DA 算法结构中数据的串行操作导致了低运算速度^[29]。本系统中的 IDCT 设计取输入,行变换每像素 16bit,输出 16bit,则若采用式(4.11)传统 DA 算法,即对每个数据一个时钟处理一位,则完成一个乘法运算需 16 个时钟周期。这种结构虽然面积较小,但在系统设计中,系统每 8 个时钟就可接收一行(或一列)8 个像素值,而系统却要用 16 个时钟完成乘法运算,因此数据的输入就有 8 个时钟的等待期,这不利于高速流水线处理结构的实现。

如果要加快读取速度,可以使用 ROM 来直接存储 X_k 与余弦系数的乘积,ROM 的大小取决于地址的宽度,也就是乘数的位数,乘数每增加一位,ROM 的体积就增加为原来的两倍。当运算精度较低的时候,这种方法的确是有利于运算速度的提高。而在本系统中,进行 1D-IDCT 变换的 8bits 或者 16bits,要在 ROM 里面完整保存所有的累加数值,需要占有太大容量的 ROM^[30]。

为了解决流水线的问题,并且从节约系统存储器资源考虑(解码中包含大量必要的查表操作,必须占有大量的存储器资源),笔者根据 DA 算法的原理,设计了一种改进的 2D-IDCT 快速算法,其中中心思想就是利用 FPGA 的移位寄存器来实现 2 倍乘法运算。

首先,我们观察 DC 系数的霍夫曼编码表。由表 4.1 可以看出,当 DC 系数为负数时,霍夫曼补充比特的最高位为 0,剩下位数为 DC 系数绝对值取反;当 DC 系数为正数时,补充比特的最高位为 1,剩下位数正好是 DC 系数的值。

因此,霍夫曼解码后得到的数据在输入 IDCT 模块前,系统将对它们进行一次转换。其中,DC 系数前加一个符号位,恰好为原系数的最高位,而剩下的位数,如果 DC 系数为负数,则取反,如果为正数,则保持不变。实际上,就是将 DC 系数转化为最高位为符号位,而其他低位为 DC 系数绝对值的表示形式。

如-1023,DC 系数为 0000000000,经过转化,变成 0111111111,而如果值为 1023,则转化后表示形式为 1111111111。

乘法运算的另一个输入为扩大 256 倍的余弦值,这些值是预先就可以计算出来,且对于每一组乘法运算来说,为一个固定的值,因此完全可以使用移位操作来实现 $8 \times n$ (n 为任意位数)位的乘法操作,具体代码如下:

```
module multiply(clk, iData, Times, oData);
input      clk;
input  [15:0] iData;  //待变换系数
input  [7:0]  Times;  //余弦值
output [15:0] oData;  //变换后输出
```

```

reg    [21:0]  data_0,data_1,data_2,data_3,
           data_4,data_5,data_6,data_7;
wire   [21:0]  data_06,data_15,data_24,data_37;
wire   [22:0]  total;
reg    [21:0]  data_0246,data_1357;
reg    [15:0]  data_buffer0,data_buffer1;

always@(posedge clk) begin
    if(Times[0]) data_0 <= iData[14:0]; else data_0 <= 15'b0;
    if(Times[1]) data_1 <= {iData[14:0],1'b0};
    else data_1 <= 16'b0;
    if(Times[2]) data_2 <= {iData[14:0],2'b0};
    else data_2 <= 17'b0;
    if(Times[3]) data_3 <= {iData[14:0],3'b0};
    else data_3 <= 18'b0;
    if(Times[4]) data_4 <= {iData[14:0],4'b0};
    else data_4 <= 19'b0;
    if(Times[5]) data_5 <= {iData[14:0],5'b0};
    else data_5 <= 20'b0;
    if(Times[6]) data_6 <= {iData[14:0],6'b0};
    else data_6 <= 21'b0;
    if(Times[7]) data_7 <= {iData[14:0],7'b0};
    else data_7 <= 22'b0;
end

assign data_06 = data_0 + data_6;
assign data_15 = data_1 + data_5;
assign data_24 = data_2 + data_4;
assign data_37 = data_3 + data_7;

always@(posedge clk)
begin
    data_buffer0 <= iData;
    data_buffer1 <= data_buffer0;
end

```

```

data_0246 <= data_06 + data_24;
data_1357 <= data_15 + data_37;
end
assign total = data_0246 + data_1357;
assign oData = {data_buffer1[15],total[22:8]};
endmodule

```

设余弦值 $C = \sum_{n=0}^7 a_n \cdot 2^n$ ，则 $X_k C = \sum_{n=0}^7 a_n \cdot 2^n \cdot X_k$ ；其中 a_n 相对于每一位为已知数， $X_k \cdot 2^n$ 为二进制乘法，在硬件实现上就是一个移位操作，因此 DC 系数乘以余弦系数这样一个乘法运算就变成了 8 个数相加。对于一个 1D-DCT 变换，一共需要 22 个这样的累加器。由于累加器占用的逻辑资源比较少，对于本系统来说这样的设计是比较合理的。

而在由 P,M 计算得到 $x_0 \sim x_7$ 的运算中，先将前面移位累加的数据转换成补码的形式，这样就可以直接调用 FPGA 里的加法器了。

从霍夫曼解码后的数据缓存里取 8 个像素值需要的时间为 8 个时钟周期；P,M 的计算一共需要 3 个时钟周期，其中包括取出的数据在 IDCT 入口处作移位前的数据转化需要 1 个时钟周期，移位运算为 1 个时钟周期，移位后 8 个数值分三个步骤进行加和，中间加入了一级触发器，由 P,M 得到 $x_0 \sim x_7$ 的值，设计中加入了两级触发器，时序上需要 2 个时钟周期，8 个 IDCT 变换后数据由一级触发器打一拍一次取出，1D-IDCT 变换后得到的数据转变成下一级 IDCT 变换所需要的数据类型需要 1 个时钟周期，而一次 IDCT 变换后得到的数据存储到两个 IDCT 模块中间的缓存区也需要 1 个时钟周期。也就是说，计算 IDCT 总共需要的时间也是 8 个时钟周期，这样在时序上就实现了流水线操作。

系统中 IDCT 模块的总体结构图如下：

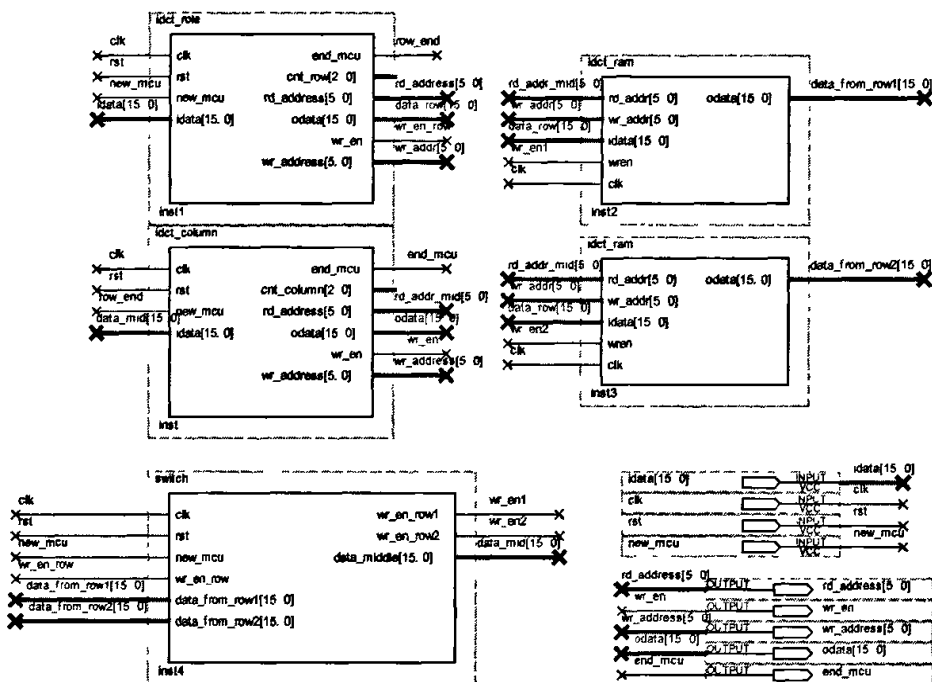


图 4.15 2D-IDCT 模块原理图

Fig 4.15 hardware module of 2D-IDCT

采用改进 DA 算法的 2D-IDCT 模块编译得到结果如图 4.16:

```

Fitter Status                Successful - Tue May 08 14:53:22 2007
Quartus II Version          6.0 Build 178 04/27/2006 SJ Full Version
Revision Name                DE2_MIOS
Top-level Entity Name       idet
Family                       Cyclone II
Device                       EP2C35F672C6
Timing Models                Final
Total logic elements         7,470 / 33,216 ( 22 % )
Total registers              5445
Total pins                   104 / 475 ( 22 % )
Total virtual pins          0
Total memory bits            0 / 483,840 ( 0 % )
Embedded Multiplier 9-bit elements 0 / 70 ( 0 % )
Total PLLs                   0 / 4 ( 0 % )

```

图 4.16 2D-IDCT 编译报告

Fig 4.16 Compile report of 2D-IDCT module

在 Quartus II 下对模块进行仿真，得到的仿真波形如图所示：

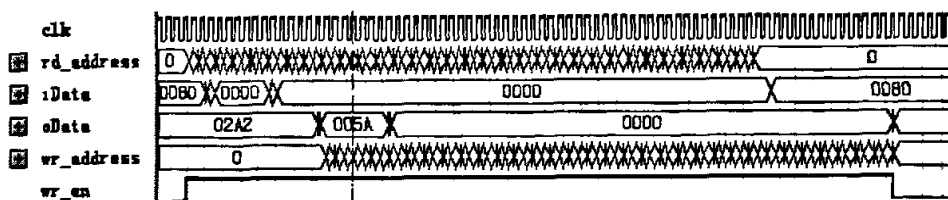


图 4.17a 2D-IDCT 仿真波形图

Fig 4.17a simulating waves of 2D-IDCT

红色虚线部分仿真波形细节如图所示:

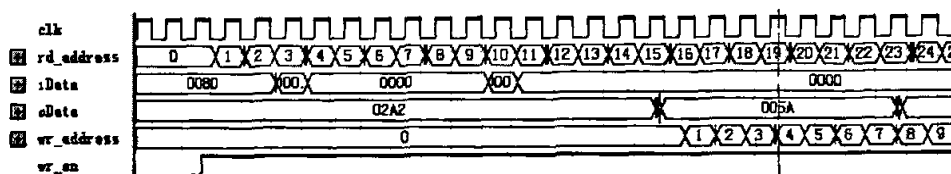


图 4.17b 2D-IDCT 仿真波形细节

Fig 4.17b detail of 2D-IDCT simulating

本 2D-IDCT 模块采用 FPGA 实现, 与 PC 机上软件实现的处理速度比较如表:

表 4.8 2D-IDCT 软硬件处理时间对比

Table 4.8 contrary of 2D-IDCT processing time by HW and SW

实现方式	FPGA 硬件	PC 机软件
速度 us/MCU	1	718

4.4 嵌入式微处理的设计

4.4.1 Nios II 嵌入式处理器简介

Altera 的 Nios II 嵌入式处理器是一个通用的 RISC(精简指令集)处理器, 在 Altera 的 FPGA 中以软核的形式实现。Nios II 处理器以其低成本, 设计灵活等特点, 在嵌入式领域得到广泛的应用。

Nios II 处理器系列包括了三种内核^[31]——快速的(Nios II/f)、经济的(Nios II/e)和标准的(Nios II/s)内核——每种都针对不同的性能范围和成本而做了优化。所有的这三种核都使用共同的 32 位的指令集结构(ISA), 可以做到各版本的二进制代码相互兼容。使用业界领先的设计软件——Altera 的 Quartus II 软件以及 SOPC Builder 工具, 工程师可以轻松的将 Nios II 的 Avalon 总线主端口和任何从端口(如内存和外设)通过 Avalon 总线互联起来, SOPC Builder 会根据需求自动加入仲裁器。

第一代的 16 位 Nios 处理器已成为可编程逻辑设计中软核嵌入式处理器的标准。32 位的 Nios II 嵌入式处理器建立在第一代的 Nios 处理器的基础上, 它不仅

提供更高的性能、更低的成本(即更少的逻辑资源占用率),还提供了齐全的开发工具以及更高的系统灵活性^[32]。

① 指令总线主端口

Nios 指令总线主端口 (Instruction Bus-Master) 是 16 位宽的端口,支持延时操作。此主端口仅仅是负责从存储器中读取指令的通道,不支持任何写操作。因为主端口支持延时操作,所以能够适用于各种不同速度的存储器。指令主端口可以在上一条指令返回之前发出新的取指请求。Nios CPU 采用“假设无分支

(branch-not-taken)”的预测方法来生成预取指的地址。由于支持具有操作延时的存储器,因此使得在使用慢速存储器时对 CPU 的影响减少到最小,并能在整体上提高系统的最高频率;只有分支预测失败时,才会有比较大的延时发生。当访问慢速存储器时,用户还可以选用片内缓存机制来提高读取指令的平均速度。

由 SOPC Builder 自动产生的 Avalon 总线具有动态总线宽度对其逻辑的功能。因此,在 Nios 指令总线主端口上可以连接 8, 16 和 32 位宽的存储器,以满足不同的应用场合的需要。

② 数据总线主端口

对于 32 位的体系结构, Nios 数据总线主端口 (Data Bus-Master) 的宽度为 32 位;对于 16 位的体系结构,则宽度为 16 位。数据主端口有以下三种用途:

- 当 CPU 执行一个数据加载指令 (LD,LDP,LDS) 时,从存储器中读取数据。
- 当 CPU 执行一个数据存储指令 (ST,STP,STS,ST8s, ST8d, ST16s, ST16d, STS8s, STS16s) 时,向存储器写入数据。
- 当 CPU 执行 TRAP 指令或者处理内,外部异常时,从中断向量表中取出中断向量。

Nios 数据总线主端口不支持延迟操作(因为预测地址没有意义),因为,数据主端口把来自从端口的延迟看作是等待周期。当 Nios 数据总线主端口被连接到具有零等待周期的存储器上时,数据加载和数据存储操作都能在一个时钟周期内完成。在数据总线主端口和指令总线主端口共享的从端口上,必须使得数据总线主端口的优先级别最高,才能获得最高的工作性能。

③ 高速缓存

Nios CPU 中的指令和数据主端口各包含一个可选的高速缓存机制。通过 SOPC Builder 中的 Nios 处理器配置向导,用户可以通过适当的适配来选择是否使用指令和数据的缓存功能。缓存位于芯片内部,大小可以配置。

指令缓存和数据缓存都采用最简单的直接映射方式。在系统运行时,用户可以对 Nios CPU 内的控制寄存器进行配置,来决定指令、数据的缓存功能是否有效。

在启用缓存功能的情况下, Nios CPU 在执行程序时,如果缓存中具有下一条

要执行的指令或者具有当前指令所使用的数据,那么 Nios CPU 就可以直接使用,从而省去从外部的存储器中获取指令或者数据的时间,我们把这种情况简称为缓存命中。当缓存有效时,缓存命中就会使得存储器的加载操作在单个时钟周期内完成;当缓存不命中时,就会引起额外的延迟。当禁止缓存时(暂时以软件的方式禁止缓存功能),访问存储器就会引起额外的延迟。但当重新启用缓存机制时,如果对存储器进行读/写操作,则将导致一个或者两个额外的延迟周期(同样情况下,对使用缓存机制的存储器,写操作也将导致一个或者两个额外的延迟周期)。

值得注意的是,高速缓存仅仅在 32 位 Nios CPU 系统中才能实现。另外, Nios 缓存机制也仅仅在 Stratix、Stratix GX 和 Cyclone 系列芯片中才能配置。

④移位单元

Nios CPU 使用固定的、不可配置的桶形移位逻辑来执行所有的移位指令(ASR, ASRI, ASL, ASLI, LSL, LSLI, RLC 和 RRC),并在 2 个时钟周期内完成(与移位的位数无关)。在做控制和数字信号处理时,移位单元对所需的移位操作提供了必要的、方便的支持。

⑤乘法支持

Nios CPU 有三种不同的方法来实现整数乘法,分别介绍如下:

1) MUL 指令

32 位的 Nios CPU 可以选择配置成 $16 \times 16 \rightarrow 32$ 的整数硬件乘法器, MUL 指令可以在 3 个或者更短的时钟周期内计算出一个 32 位的结果。MUL 选项被选中后, MUL 指令将利用由 SDK(软件开发包)自动产生的 C 语言运行支持库(C-runtime 库)来实现乘法运算。这里要注意的是, 16 位的 Nios 指令系统不支持此项。

2) MSTEP 指令

32 位的 Nios CPU 可以配置成执行单步 16×16 的乘法运算器,硬件乘法器将在 2 个时钟周期内完成部分(其余部分在软件辅助下完成)乘法运算。MSTEP 选项被选中后, SDK 生成的 C 语言运行支持库就会支持相应的乘法操作,通过使用 MSTEP 指令,可以连续实现乘法操作,例如用 16 个连续的 MSTEP 指令实现 $16 \times 16 \rightarrow 32$ 乘法。由于使用 MSTEP 所占用的 CPU 硬件资源还不到 5%,很多情况下利用如此小的硬件开销换得了对硬件乘法的支持,这无疑给软件设计带来很大的方便。因此系统将 MSTEP 设置为默认项。同样 16 位的 Nios 指令系统不支持此项。

3) 软件乘法器

当禁止 MSTEP 和 MUL 两种选项而需要完成乘法运算时,则由 SDK 中的 C 语言运行支持库利用移位和加法指令来实现整数乘法运算。软件乘法器虽然占用了很少的 CPU 硬件逻辑,但是比起硬件实现的乘法运算,其执行速度要慢的多。

⑥中断支持

Nios CPU 允许用户取消对 TRAP 指令、硬件中断或内部异常的事件处理, 这种选择仅仅适用于非常简单的系统。在这种配置下, Nios CPU 将具有以下特征:

1) 不包括 irq 输入引脚。

2) 无异常处理 (TRAP 指令未定义)。

3) 在执行 SAVE 和 RESTORE 指令使寄存器上溢/下溢时, 不产生异常中断。仅当有如下要求时才能取消陷阱指令、硬件中断或者内部异常: 需要最小化的 Nios CPU 核; 确定应用程序不会产生寄存器窗口溢出异常, 即子程序的调用深度小于寄存器窗口数。

4) 系统没有任何硬件中断源。

5) 汇编代码不包含陷阱指令。

Nios 的 SDK 在编译过的代码中不会产生陷阱指令。缺省情况下, 中断支持开启功能。

⑦Nios 片上调试模块 (OCI 模式)

Nios CPU 有个可选的片上 JTAG 调试模块, 它通过标准的 JTAG 接口实现与 CPU 的通信。此模块是 First Silicon Solutions (FS2) 公司开发的 IP 核, 被称为 Nios OCI (On-Chip Instrumentation) 调试模块。

在调试应用程序时, Nios OCI 调试模块支持用户设置硬件断点, 还可以实现软件跟踪。所跟踪的数据保存在片上的存储器或者外部的系统分析器中 (如 FS2 公司的 ISA-Nios In-System Analyzer)。该模块还可以向寄存器和存储器读/写数据, 允许在程序运行过程期间向存储器下载软件程序和检查寄存器。Nios OCI 调试模块可以控制 CPU 的运行, 无论何时, 用户都可以对正在执行的程序进行控制。也就是说, 这种控制作用的优先级别最高。

除了调试功能外, 在与主设备通信时, Nios OCI 也能被用作标准的输入/输出。该功能可以与调试功能同时使用, 且互不干扰, 但数据的传输速率比起串口要慢一些。

4.4.2 SOPC Builder

目前 SOC 概念在设计中应用广泛, 而传统 SOC 设计需要手动连接处理器与外设, 手动分配地址空间资源, 既耗时又容易出错。Altera 针对这种情况, 根据自己的可编程解决方案, 提出了 SOPC (System On a Programmable Chip), 并开发了一种智能工具, 帮助用户快捷地产生一个 SOPC 系统, 这个工具就是 SOPC Builder。

SOPC Builder^[33]是一个软件工具, 它允许用户创建一个 Nios 系统模块, 或者创建多个主设备 SOPC 模块。SOPC Builder 中包含了 Nios 处理器以及其他一些常用的外设 IP 模块, 利用 SOPC Builder, 用户可以很方便地将处理器、存储器和其

他外设模块连接起来, 形成一个完整的系统。

从用户的角度来看, SOPC Builder 是一个能够生成复杂硬件系统的工具。从内部来看, SOPC Builder 包含两个主要部分即: 图形用户界面(GUI)和系统生产程序。

SOPC Builder 图形用户界面提供管理 IP 模块、配置系统和报告错误等功能。用户通过 GUI 设计系统时, 所有的设置都保存在一个叫 PTF 类型的文件里, 可以认为 SOPC Builder 是一个 PTF 文件的专用编辑器。

系统生成程序通常从 GUI 界面中启动, 但实际上它是一个完全独立的功能程序。通过 GUI 完成设计, 就可以启动系统生成程序了。系统生成程序的执行需要从 PTF 文件读取系统配置信息和参数。PTF 文件是图像用户接口和系统生成程序之间唯一的交互渠道。PTF 文件的内容, 一部分用于图形用户接口(显示 IP 模块的信息), 一部分用于系统生成程序(总线时序信息)生产系统互连逻辑; 其他 PTF 文件的数据同时用于图形用户接口和系统生成程序。

4.4.3 Avalon 总线

Avalon 总线是一种相对简单的总线结构, 主要用于连接片内处理器与外设, 以构成可编程单芯片系统(SOPC)。Avalon 总线描述了主、从构件之间的端口连接, 以及构建之间通信的时序关系^[34]。

Avalon 总线设计的基本目标是:

- ①简单: 提供简单的、易于理解的协议;
- ②优化总线资源利用率, 节约可编程逻辑器件的逻辑单元;
- ③同步操作: 易于与片上的其他用户逻辑集成, 避免了复杂时序分析。

Avalon 总线是为 SOPC 环境而设计的, 互联逻辑由 PLD 内部的逻辑资源构成, 具有以下基本特点:

①外设接口的时钟与 Avalon 时钟是同步的。因此, 不需要复杂的异步握手/应答机制, 采用标准的同步时序分析技术就可以测出 Avalon 总线和整个系统的性能。

②所有的信号都是高电平或低电平有效, 这样有利于总线的切换(turn-around) Avalon 总线内部的多路复用器决定驱动所对应的设备。外设在未选取的情况下, 也不需要将输出置为高阻态。

③地址、数据和控制信号使用独立的专用端口, 简化了外设的设计。外设不需要地址译码和判断总线周期, 在未选中时, 也不需要禁止输出。

Avalon 总线还包括许多其他的特性和约定, 以支持由 SOPC Builder 自动生成的系统、总线和外设。这些特性和约定包括:

①最大 4GB 的地址空间。存储器和外设可以映射到 32 位地址空间中的任意位置。

- ②内置地址译码。Avalon 总线自动产生所有外设的片选信号，极大的简化了给予 Avalon 总线的设计工作。
- ③多主设备总线结构。Avalon 总线上可以包括多个主外设，并自动生产仲裁逻辑。
- ④采用向导帮助用户配置系统。SOPC Builder 提供图形化的向导，帮助用户进行总线配置（添加外设、指定主/从关系、定义地址映象等）。编译器将根据用户在向导中输入的参数自动生成 Avalon 总线结构的总线配置文件。
- ⑤动态地址对齐。如果参与传输的双方总线宽度不一致，则 Avalon 总线自动处理数据传输的细节，使得不同数据总线宽度的外设能够方便连接。

在 Avalon 总线的规范中，定义了主、从端口之间传输数据所需要的信号和时序。Avalon 总线与外设接口的信号组成，因传输类型的不同而有所差异。首先，主、从端口的定义不同而具有不同的接口属性；此外每个端口的传输类型和信号现数目也不尽相同，它们取决于 PTF 文件中的定义。

Avalon 总线规范提供了各种选项来剪裁总线的信号和时序，以满足不同类型外设的要求。基本的 Avalon 总线传输一次只能在主从端口之间传输一个数据单元。总线传输可以调整等待时间以适应于慢速的外设，也具有流传输和多主传输方式来满足告诉外设的需要。Avalon 从传输的所有信号时序都是源于从端口的基本读/写传输，同样，主端口的基本读/写传输也是主端口传输的基础。

● 主端口传输

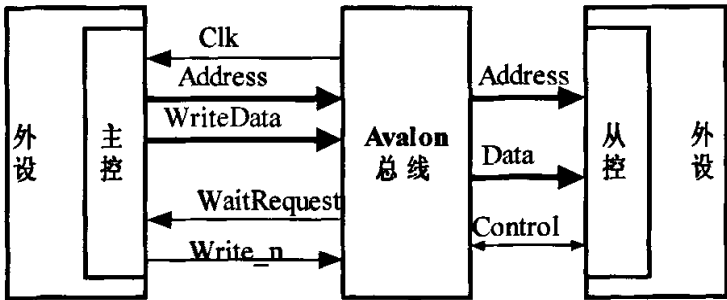


图 4.18 Avalon 主控器

Fig 4.18 Avalon master

1) 基本主端口读传输

在基本主端口读传输中，主端口通过 Avalon 总线模块提供有效的地址和读请求信号（在时钟上升沿）发起总线传输。在理想的情况下，读取的数据在下一个时钟上升沿之前从 Avalon 总线模块返回，总线传输在一个周期内结束。如果在下

一个时钟周期上升沿读取的数据还没有准备好, Avalon 总线模块就设置一个等待请求, 并使主端口暂停, 直至数据从目标从端口取回。基本主端口传输没有延迟。

主端口读传输开始于 Clk 的上升沿, 在第一个 Clk 上升沿之后, 主端口立即设置 Address、和 Read_n 有效。如果 Avalon 总线不能在第一个总线时钟周期内提供 readdata, 它会在下一个 Clk 上升沿前设置 WaitRequest 有效。如果主端口在 clk 上升沿发现 WaitRequest 有效, 它会保持等待状态。主端口必须使所有输出信号保持稳定, 直到 WaitRequest 失效后的下一个 Clk 上升沿。在 WaitRequest 失效后, 主端口在下一个 clk 上升沿捕获 readdata, 并使 address 和 read_en 失效。主端口可以在下一个总线周期立即发起另一次总线传输。

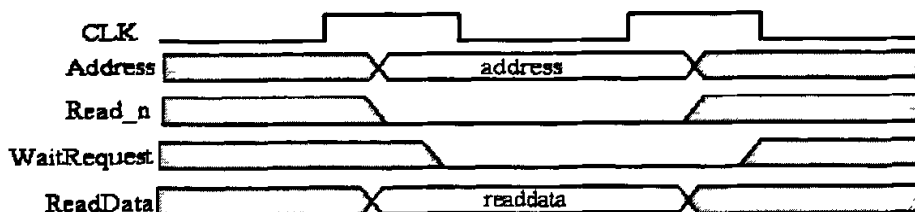


图 4.19a 主端口基本读传输

Fig 4.19a Master read operation

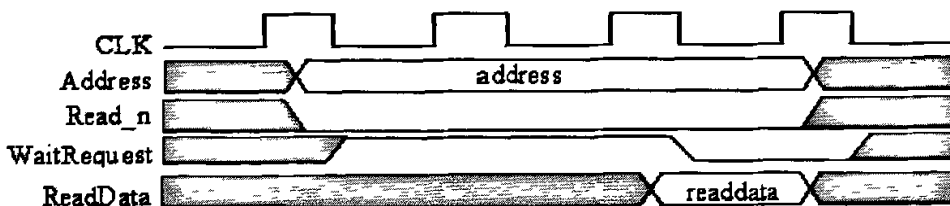


图 4.19b 主端口基本读传输

Fig 4.19b Master read operation

2) 基本主端口写传输

主端口通过向 Avalon 总线模块提供有效的地址/数据和写请求信号, 在时钟上升沿发起总线传输。在理想的情况下, 目标外设在下一个时钟上升沿之前捕获数据, 写传输在一个总线周期内结束。如果目标外设不能在一个总线周期内捕获数据, Avalon 总线便会使主端口暂停, 直至从端口捕获了数据。

主端口写传输开始于 Clk 的上升沿, 在第一个 Clk 上升沿之后, 主端口立即设置 Address、WriteData 和 Write_n 有效, 在第二个 Clk 的上升沿, 地址和数据被锁存在 Avalon 总线内部, 一个总线写周期完成。如果数据不能在第一个总线周期内被捕获, 如图 7(b) Avalon 总线模块会在 Write_n 有效的下一个 Clk 上升沿前

设置 WaitRequest 有效，使主控端口处于等待状态。主端口必须使 Address、WriteData 和 Write_n 保持稳定，直到 WaitRequest 失效后的下一个 Clk 上升沿完成对地址和数据的锁存。主端口可以在下一个总线周期立即发起另一次总线传输。

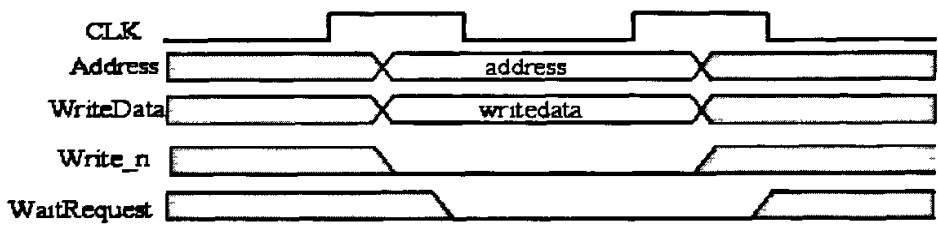


图 4.20a 主端口基本写操作

Fig 4.20a Master write operation

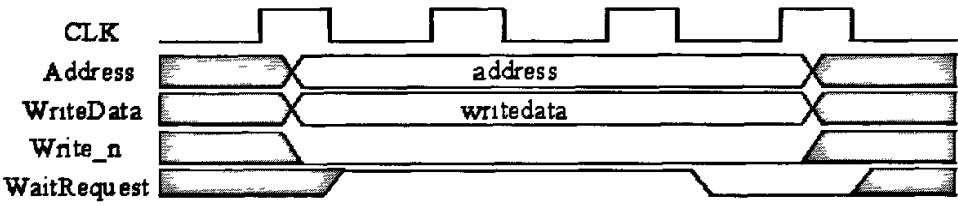


图 4.20b 主控器带等待的写操作

Fig 4.20b Master write operation with waiting

● 从端口传输

从系统级的视角来看，数据交换是发生在主外设和从外设的。然而以从外设的角度来看，数据传输发生在外设的从端口和 Avalon 总线模块之间。Avalon 总线上的某个主外设向 Avalon 总线模块的主端口发起了一次传输，Avalon 总线模块随后对相应的从端口发起总线传输。

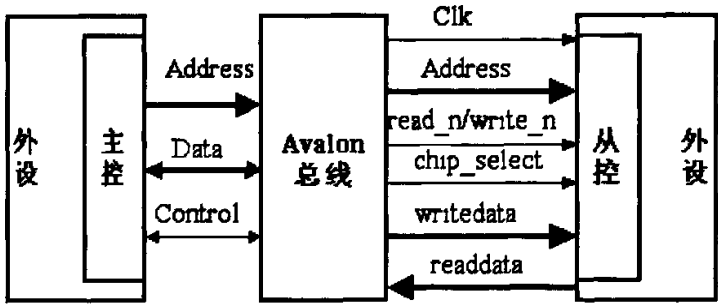


图 4.21b Avalon 从控制器

Fig 4.21b Avalon slaver

1) 基本从端口读传输

基本从端口读传输模式是所有 Avalon 从端口读传输的基础。所有的其他的从端口读传输使用的信号都包含了基本从端口读传输的信号，并扩展了基本从端口读操作时序。基本从端口读传输由 Avalon 总线模块发起，然后从端口向 Avalon 总线传输一个单元的数据。基本从端口读传输没有延迟。

图 4.13a 显示了一个基本从端口读传输的例子，在 Avalon 基本读传输中，总线传输开始于一个时钟上升沿，并在下一个时钟上升沿结束，不插入等待周期。由于传输在一个时钟周期内完成，目标外设必须能够立即、异步向 Avalon 总线模块输出相应地址中的内容。

在 Clk 的第一个上升沿，Avalon 总线向目标外设传递 address、byteenable_n 和 read_n 信号。Avalon 总线模块内部对 address 进行译码，产生片选并驱动从端口的 chipselect 信号。一旦 chipselect 信号有效，从端口在数据有效时立即驱动 readdata 输出。最后 Avalon 总线模块在下一个时钟上升沿捕获 readdata。

2) 基本从端口写传输

和从端口读传输类似，基本从端口写传输是所有 Avalon 从端口写传输的基础。所有其他的从端口写传输模式使用的信号都包含了基本从端口写传输的信号，并扩展了基本从端口写操作时序。基本从端口写传输由 Avalon 总线模块发起，然后 Avalon 总线传输一个单元的数据给从端口。基本从端口写传输没有延迟。

图 4.13b 显示了基本从端口写传输，没有等待周期、建立时间和保持时间。Avalon 总线模块提供 address、writedata、byteenable_n 和 write_n 信号，然后设置 chipselect 有效。从端口在下一个时钟上升沿捕获地址、数据和控制信号，写传输立即结束。整个传输过程仅花费一个总线周期。从外设可以在传输结束后再花费一些总线周期来实际处理写入的数据。如果外设不能在每个总线周期接受数据，则需要加入等待周期。

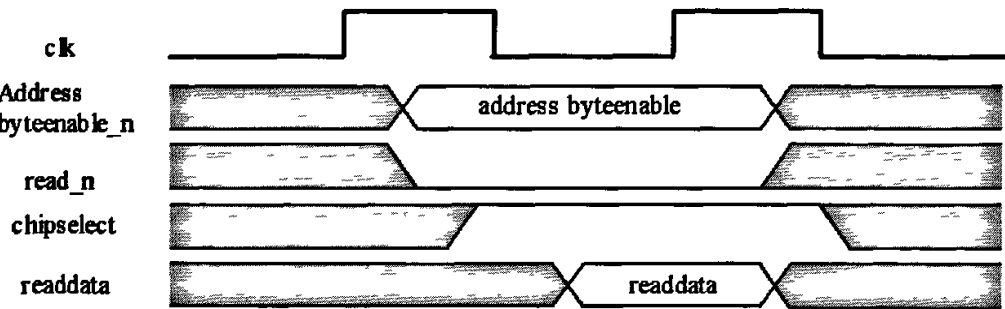


图 4.22a Avalon 从控器基本读操作
Fig 4.22a Avalon slaver read operation

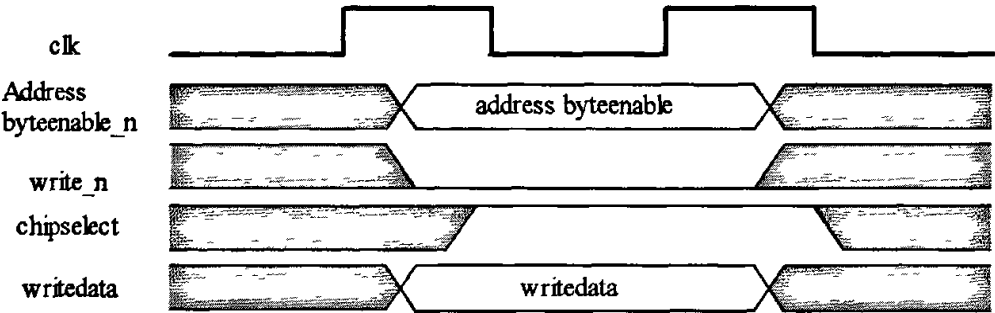


图 4.22b Avalon 从控器基本写操作

Fig 4.22b Avalon slaver write operation

4.4.4 片上系统设计

①自定义模块与 Nios II 软核处理器的接口设计

JPEG 编码服务器端与解码客户端的硬件模块分别设计成了用户自定义模块。其中，服务器端包括了图像采集，JPEG 编码和网络传输三个功能模块，而客户端则相应网络接收，JPEG 解码以及图像显示，同样为三个用户自定义模块。

以 JPEG 硬件编码用户自定义模块为例，此模块一共设置了两个主端口和四个从端口。其中主端口 M1 负责从采集模块采集到的数据缓存区主动读取数据，主端口 M2 负责将压缩后的码流存储到网络传输的缓存区；从端口 S1,S3 分别用于 CPU 对主端口 M1 读取基地址以及对主端口 M2 写入基地址的配置，从端口 S2 用于 C 通知 JPEG 模块开始压缩，从端口 S4 用于压缩后中断发送。Quartus II 界面下 JPEG 编码硬件模块与 CPU 接口如图 4.23:

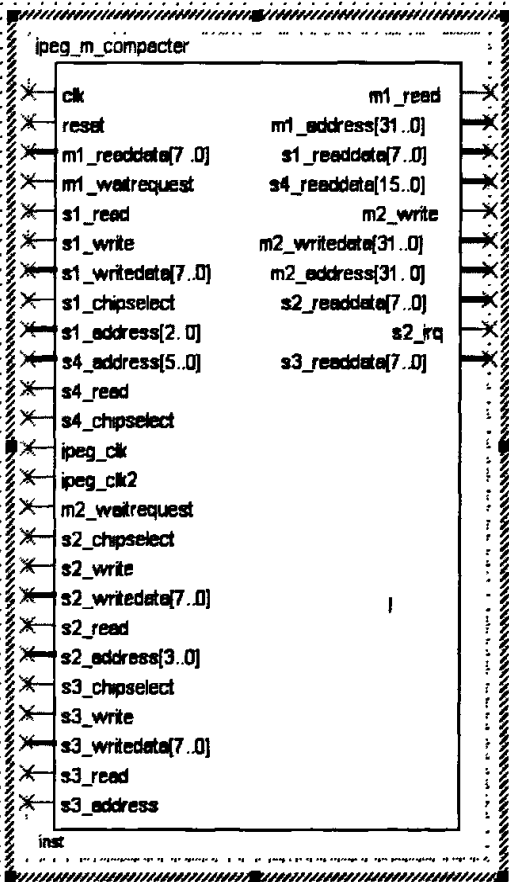


图 4.23 JPEG 编码模块接口

Fig 4.23 interface of JPEG encoding module

M1, M2 分别为有等待周期的读传输主端口和写传输主端口, 需要的接口信号有主读(写)使能, 读(写)地址信号, 读(写)等待信号以及数据信号, 其中, 使能信号, 地址信号为主端口自己产生的信号。

为了提高数据的读写存储速度, 主端口读写的数据宽度都设计成 32 位。并且, 主端口一次性读操作后, 得到的一小段数据存储在用户自定义模块内的数据缓存区内, 然后再对数据进行后续处理, 这样做的有两个好处, 一个就是由于系统中图像采集, JPEG 压缩和网络传输的时钟频率不同, 异步时钟域的数据传递必须要有一个 Buffer 作为转换; 另一个好处就是, 模块与模块之间用缓存器来隔断, 可以减少它们在接口的相关性, 这样模块的独立性与可移植性增强, 使系统更加稳定。

从端口 S0, S1, S2, S3 的设计则相对较简单, 只要采样到 CPU 传递过来的片选信号, 读写使能信号, 以及地址信号, 存储或者发送相应的数据即可。

②SOPC Builder 内的系统构建

在 SOPC Builder 中, 结合服务器端的采集模块, JPEG 编码自定义模块, 网络传输等外设, 挂到 AVALON 总线上, 组成一个包含图像采集, JPEG 压缩以及网络传输完整的系统, 结构如图 4.20 所示。

Use	Module Name	Description	Input Clo...	Base
<input checked="" type="checkbox"/>	cpu	Nios II Processor - Altera Corporation	sys_clk	0x00400000
<input checked="" type="checkbox"/>	cfi_flash_0	Flash Memory (Common Flash Interface)		0x00000000
<input checked="" type="checkbox"/>	tri_state_bridge_0	Avalon Tristate Bridge	sys_clk	
<input checked="" type="checkbox"/>	sdram_0	SDRAM Controller	sys_clk	0x00800000
<input checked="" type="checkbox"/>	npcm_controller	NPCM Serial Flash Controller	clk_50	0x00000000
<input checked="" type="checkbox"/>	jtag_uart_0	JTAG UART	sys_clk	
<input checked="" type="checkbox"/>	→ avalon_jtag_slave	Slave port		0x00401080
<input checked="" type="checkbox"/>	uart_0	UART (RS-232 serial port)	sys_clk	0x00401000
<input checked="" type="checkbox"/>	timer_0	Interval timer	sys_clk	0x00401020
<input checked="" type="checkbox"/>	timer_1	Interval timer	sys_clk	0x00401040
<input checked="" type="checkbox"/>	led_16207_0	Character LCD (16x2, Optrex 16207)	sys_clk	0x00401060
<input checked="" type="checkbox"/>	led_red	PIO (Parallel IO)	sys_clk	0x00401070
<input checked="" type="checkbox"/>	led_green	PIO (Parallel IO)	sys_clk	0x00401080
<input checked="" type="checkbox"/>	button_pio	PIO (Parallel IO)	sys_clk	0x00401090
<input checked="" type="checkbox"/>	switch_pio	PIO (Parallel IO)	sys_clk	0x004010...
<input checked="" type="checkbox"/>	DM9000A	DM9000A	sys_clk	0x004010B8
<input checked="" type="checkbox"/>	sram_0	SRAM_160bits_512K	sys_clk	0x00400000
<input checked="" type="checkbox"/>	image_collector_0	master_wlry	sys_clk	0x004000E0
<input checked="" type="checkbox"/>	jpeg_m_compacter_0	jpeg_m_compacter	sys_clk	
<input checked="" type="checkbox"/>	→ s1	Slave port		0x004000E0
<input checked="" type="checkbox"/>	← m1	Master port		
<input checked="" type="checkbox"/>	→ s2	Slave port		0x004000C0
<input checked="" type="checkbox"/>	→ s3	Slave port		0x004000FC
<input checked="" type="checkbox"/>	← m2	Master port		
<input checked="" type="checkbox"/>	→ s4	Slave port		0x00400000

图 4.24 服务器端 SOPC 系统构建

Fig 4.24 SOPC system constructure of the server

与编码服务器端的原理相似, 解码客户端在 SOPC Builder 下构建的 SOPC 如下

图所示：

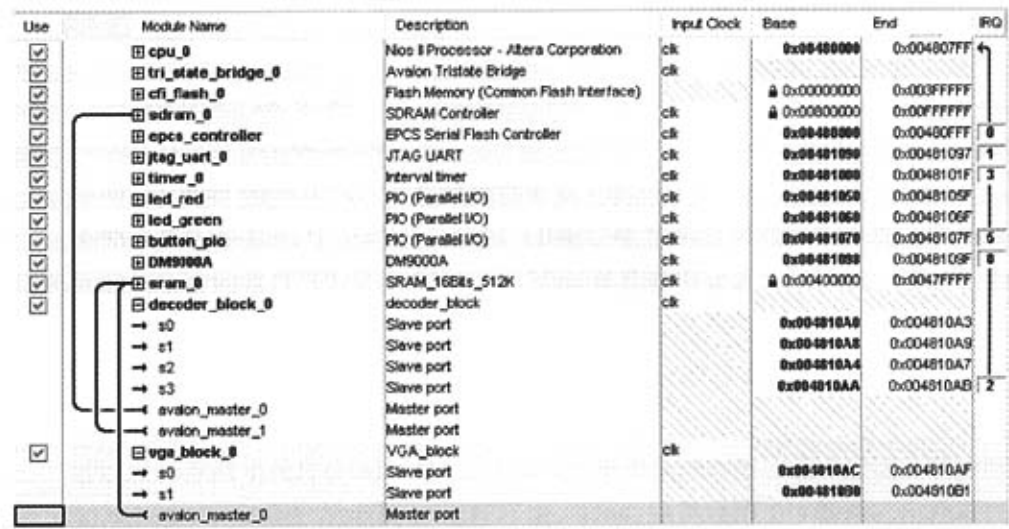


图 4.25 客户端 SOPC 系统构建

Fig 4.25 SOPC system constructure of the user

4.5 本章小结

本章首先介绍整个系统的总体方案, 然后详细讲述 JPEG 编码与解码的硬件模块具体设计。其中, 编码部分按照处理的顺序, 设计先后为 DCT 模块, 量化模块以及熵编码模块。解码部分与编码部分对称, 因此省略了设计相对简单的模块, 而着重阐述使用了优先查表方法的霍夫曼解码与改进的 DA 算法实现 2D-IDCT 模块。最后一节介绍在 SOPC Builder 中, 如何将各个分散的硬件模块按照 Avalon 总线的要求转变成用户自定义模块, 并利用 Nios II 嵌入式软核的可定制特性, 将这些模块组成一个完整的 SOC 系统。

5 Nios II 控制程序

5.1 Nios 集成开发环境

5.1.1 Nios II 软件开发工具

Altera 提供的 Nios II 软件开发环境包括以下工具^[35]:

①Nios II IDE—Nios II 集成开发环境 (IDE) 是 Nios II 处理器的软件开发图形用户界面。所有的软件开发任务都可以在 Nios II IDE 中完成, 如编辑、构建和调试程序。Nios II IDE 是其他所有工具的基础。

②GNU 工具链—Nios II 编译工具链基于标准的 GNU C (GCC) 编译器, 连接器和生成文件工具。

③硬件抽象层系统库(HAL)—HAL 具有基于美国国家标准协会 (ANSI) C 标准库 (如 `stdio.h`、`math.h`) 的主机 C 运行环境。HAL 提供通用 I/O 器件, 用户可以采用 C 标准库程序 (如 `printf()`) 对硬件进行编程。

④实时操作系统(RTOS)—Altera 提供带有 Nios II 开发包的 MicroC/OS-II RTOS。MicroC/OS-II 构建在安全线程 HAL 系统库上, 实现简单的、记录良好的 RTOS 调度程序。

⑤TCP/IP 堆栈—Altera 与 Nios II 开发包一起, 提供简单的 TCP/IP 堆栈, TCP/IP 堆栈构建在 MicroC/OS-II 之上, 使用标准的 UNIX socket 应用程序接口(API)。

⑥指令集仿真器 (ISS) —用户使用 Nios II ISS, 可以在建好目标硬件平台之前就开始开发程序。Nios II IDE 允许用户象在硬件目标上那样轻松的在 ISS 上运行程序。

⑦实例设计—所记录的软件实例证实了 Nios II 处理器和开发环境的所有优秀特性。

5.1.2 Nios II 软件开发流程

在进行 Nios 软件开发时, 通常按照下面的过程进行^[36]:

①获得目标 Nios 系统的 SDK:

从 SOPC Builder 创建的工程目录中, 得到目标 NIOS 处理器系统的 SDK。软件开发环境的基础是 SOPC Builder 生产的 SDK 目录。

②建立和编译应用软件:

首先使用文本编辑器, 用 C/C++ 或者汇编语言写应用程序的代码 (.c 或 .s), 然后使用 `nios-build` 命令或者 `Makefile` 将源代码编译成可执行的代码。

③下载可执行代码到目标板:

如果用户的目标系统中包含了 GERMS 监控程序, 则它将扮演起调试器的角

色, 允许用户允许可执行代码、读写存储器、下载代码(或数据)到存储器和擦除闪存。如果用户的目标系统中包含的是 OCI 调试模块, 则可以通过 JTAG 端口对目标系统进行代码的下载和调试;

④调试代码:

如果用户采用了 GERMS 监控程序, 那么调试信息通过 `printf()` 函数由标准输入输出设备来发送, 如果需要更细致的调试, 可以通过 JTAG 下载电缆, 用 NIOS OCI 调试模块来访问 CPU。通过 NIOS OCI 调试模块, 能够执行单步调试代码, 检查存储器和寄存器的内容等。

⑤转变成自引导代码:

有两种方式存储代码到非易失存储器中, 即存储到片外闪存或存储到片上存储器。

5.2 Nios 软件控制程序设计

5.2.1 控制程序功能

Nios 软件程序需要完成的功能有三个:

①在系统上电时, 对图像采集, JPEG 编码, JPEG 解码, 图像显示以及网络传输进行初始化配置。这些配置包括图像处理中, 数据缓存地址的分配、初始状态的配置等等。这部分程序只需要对一系列寄存器写数据即可。

②负责网络部分的所有工作, 包括建立监听端口的线程, 连接请求的发送与接收, 数据传输与接收;

③处理图像采集, 编码解码以及图像显示模块发送的中断请求。

由于 Nios II 自带的 Uc/OS-II 操作系统, 内含 TCP/IP 协议, 在软件程序中可以直接进行调用, 因此网络部分的程序设计简化了许多。

5.2.2 编解码控制软件流程

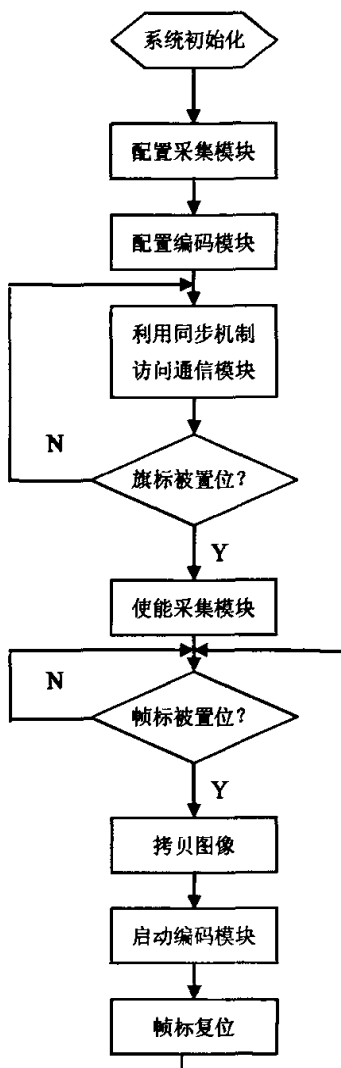


图 5.1 编码控制软件流程图

Fig 5.1 software flow of JPEG encoding control

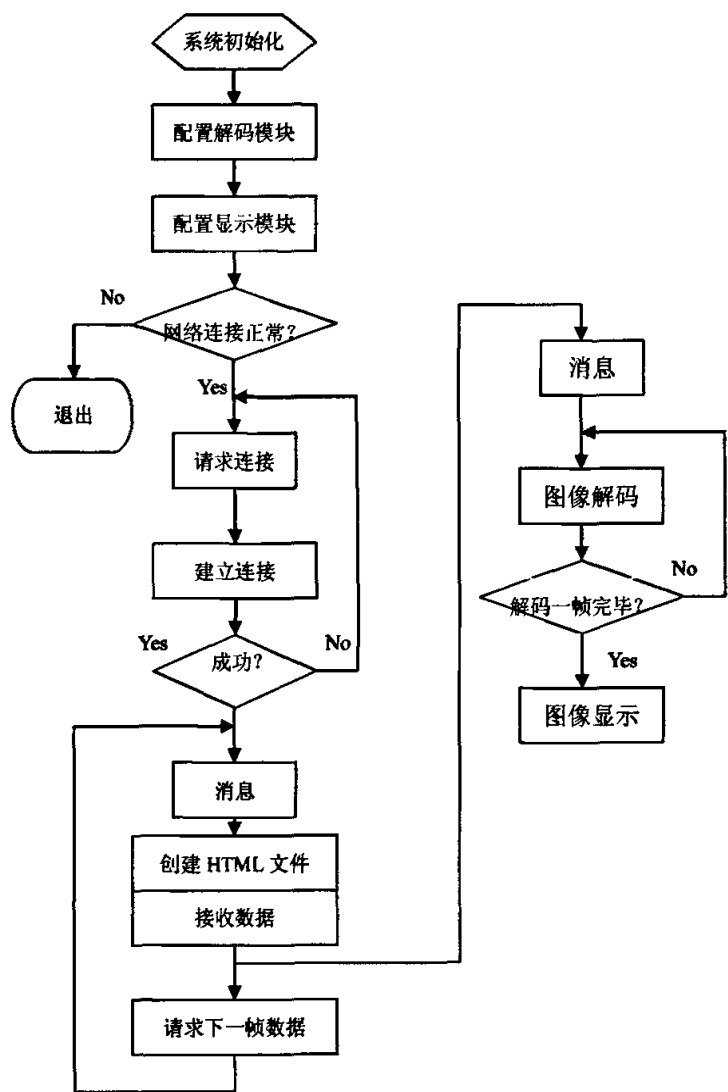


图 5.2 解码控制软件流程图

Fig 5.2 software flow of JPEG decoding control

5.3 本章小结

本章首先介绍了 Nios II 开发工具以及 Nios II 下软件的开发流程,然后重点介绍本系统 Nios II 软件控制程序的功能以及系统运行时软件执行的流程。由于 Nios II 自带的 Uc/OS-II 操作系统,因此在 Nios IDE 环境下进行系统控制的设计非常方便。

6 结 论

JPEG 标准是一个具有高压缩比、图像恢复质量高、适用范围非常广泛且系统比较容易实现的静态图像压缩编码标准。它结合采用了预测编码、不定长编码等多种压缩方法,图像压缩的效率非常高,因此在静止图像压缩上,如互联网、数字摄像、遥感、远程医疗和电子商务等等有着非常广泛的应用。在对动态图像的方面,许多对图像压缩质量要求较高且要求逐帧处理的场合,JPEG 也有着非常广阔的前景。本课题所作的基于 FPGA 的实时 JPEG 图像编解码系统,对于用硬件实现 JPEG 图像压缩处理算法,特别是在片上系统(SOC)的实现做了有益的尝试。

本文利用 FPGA 并行处理的特点,结合 JPEG 标准的算法,在 FPGA 上实现了一种对动态图像进行处理的 JPEG 实时图像编解码系统。针对系统对图像处理速度的要求,采用将 DC 系数进行直接编码,根据图像 AC, DC 系数出现的几率来重排霍夫曼码表以缩短查表时间,使用一个 1D-DCT 变换模块实现两个 2D-DCT 变换,并且对 DA(分布式)算法进行改进,使用多个累加器实现 2D-IDCT 变换中的浮点数乘法,处理速度远远超过了实时性的要求(40ms/帧)。并且,利用 Quartus II 的子工具 SOPC Builder,将图像采集, JPEG 编码、网络传输、JPEG 解码以及图像显示按照 AVALON 总线的要求,定制成用户自定义模块,使它们分别在服务器端和客户端的 Nios II 软核控制下运行。

服务器将采集到的图像进行 JPEG 编码,通过网络发送至客户端,再由客户端解码显示出恢复后的图像,是为一个完整的 JPEG 编解码片上系统(SOC)。实验结果显示, FPGA 硬件编解码系统图像处理的速度相对于软件有几十倍的提高,并且对图像处理的片上系统实现具有一定的参考价值。

当然,系统尚有一些不足,但由于时间关系,需要在以后的研究中予以改进:一是,编码部分模块直接调用了乘法器进行搭建,在速度提高上仍有空间;二是,在 JPEG 系统的扩展功能方面,没有更深一步的进行研究,实现的功能相对较简单;三是,对编解码的码率控制上,没能作很多的工作。这些问题都是该相关课题的研究重点和热点,也是本课题进一步研究和提高的方向。

致 谢

在学业即将完成之际，我要衷心感谢我的两位老师：张玲副教授和何伟副教授。

在三年的研究生学业中，张老师无论是在学术上还是在生活中，对我们的关怀和照顾都是无微不至。在平时的学习中，张老师从理论和实践两方面，都给予我大量的、极其有益的建议和指导，并在论文的撰写和审稿中倾注了大量的心血。她严谨的治学态度和对人生的孜孜不倦的追求，深深的感染着我，让我在以后的道路中，以一个积极的精神面貌去面对我的学业和人生。在此，我谨向她表示我深深的谢意！

感谢何老师长时间以来的指导。正是由于他的严格要求，我的工作和学习才得以不断进步。而且他干练的行事作风，对事业充满激情永不疲倦的奋斗精神，以及在学术上追求完美的态度，让我学到很多，受益非浅。在我以后的工作中，是我学习的好榜样。

感谢与我同窗三年的同学们，尤其是张峰同学，帮助我完成了算法的软件验证，本课题的前期研究工作才如期展开。还要感谢夏博、胡又文、陈方泉等同学，在课题仿真调试的过程中，由于他们的帮助，加速了本课题的研究进程。也正是他们对我在碰到的问题，进行了耐心的解答，使我可以把工作重心放在硬件算法和系统设计上，最后顺利完成了本课题的设计。在此，衷心地向他们表示感谢。

感谢实验室的同门师兄和师弟、师妹，给我的课题提出了不少非常好的建议。和他们在一起生活学习的三年将永远成为我人生美好的回忆。

还需要感谢的是我的父母，是他们教会我做人的道理，并且不论在物质上，还是精神上，二十多年来对我无私的付出，使我顺利完成我的学业。

感谢我的朋友潘登在我进入暂时的低谷时，给我最真诚的鼓励和帮助，让我始终朝正确的方向努力。

衷心地感谢在百忙之中评阅论文和参加答辩的各位专家、教授！

廖 彦

二〇〇七年四月

参考文献

- [1] 吕凤军.数字图像处理编程入门,清华大学出版社,1999, 287~312.
- [2] 林福宗.多媒体技术基础,清华大学出版社,2000, 178~200.
- [3] BHASKARAN, V., KONSTANTINIDES, K. 《Image and Video Compression Standards Algorithms and Architectures - Second Edition》. Kluwer Academic Publishers, USA, 1999.
- [4] 汪宇飞,JPEG 高速编码芯片的设计及其性能优化,西北工业大学硕士学位论文,2006.2.
- [5] 任爱锋,初秀琴等.基于 FPGA 的嵌入式系统设计. 第一版. 西安:西安电子科技大学出版社, 2004, 9~53
- [6] 潘松,黄继业.EDA 技术实用教程. 第一版.北京:科学出版社,2002,32~54
- [7] 任艳颖,王彬.IC 设计基础.西安电子科技大学出版社.2003.5:3~42.
- [8] 吴继华,王诚.Altera FPGA/CPLD 设计(高级篇).人民邮电出版社.2005.7:1~50.
- [9] Gregory K. Wallace. The JPEG Still Picture Compression Standard [J]. IEEE Transactions on Consumer Electronics, Dec. 1991.
- [10] [日]小野定康,铃木纯司.JPEG/JPEG2 技术.科学出版社.2004.1:128~130.
- [11] 夏良正.数字图像处理.东南大学出版社,2004.8,64~65.
- [12] 何小海,滕奇志.图像通信.西安电子科技大学出版社.2005.2:120~136.
- [13] CCITT, Digital Compression and Coding of Continuous-tone Still Images- Requirements and Guidelines [S], CCITT, T. 81 (09/92).
- [14] [日]小野定康,铃木纯司.JPEG/JPEG2 技术.科学出版社.2004.1:68~69.
- [15] [日]小野定康,铃木纯司.JPEG/JPEG2 技术.科学出版社.2004.1:70~73.
- [16] Jari A. Nikara, Jarmo H. Takala and Jaakko T. Astola. Discrete cosine and sine transforms—regular algorithms and pipeline architectures. Signal Processing, Volume 86, Issue 2, 2006.2: 230~249.
- [17] Mohamed El-Sharkawy and Waleed Eshmawy. A Fast 8×8 Pruned DCT Algorithm. Digital Signal Processing, Volume 6, Issue 3, 1996.7:145~154.
- [18] Gerlind Plonka and Manfred Tasche. Fast and numerically stable algorithms for discrete cosine transforms. Linear Algebra and its Applications, Volume 394, 2005.1:309~345.
- [19] Rui-Xiang Yin and Wan-Chi Siu. A new fast algorithm for computing prime-Length DCT through cyclic convolutions. Signal Processing, Volume 81, Issue 5. 2001.5:895~906.
- [20] Venkatram Muddhasani and Meghanad D. Wagh. Bilinear algorithms for discrete cosine transforms of prime lengths. Signal Processing, Volume 86, Issue 9, 2006.9:2393~2406.

- [21] Domagoj Babic. Discrete Cosine Transform Algorithms for FPGA Devices [M]. Zagreb, 2003.
- [22] CHEN W H, SMITH C H, FRALICK S C. A fast computational algorithm for the discrete cosine transform.[J]. IEEE Trans Commun, 1977,25,1004~1009.
- [23] [日]小野定康,铃木纯司.JPEG/JPEG2 技术.科学出版社.2004.1:67~68
- [24] [日]小野定康,铃木纯司.JPEG/JPEG2 技术.科学出版社.2004.1:72.
- [25] [日]小野定康,铃木纯司.JPEG/JPEG2 技术.科学出版社.2004.1:73.
- [26] [日]小野定康,铃木纯司.JPEG/JPEG2 技术.科学出版社.2004.1:74~76.
- [27] (美) Michael D. Ciletti 著 张雅琦, 李铮等译.Verilog HDL 高级数字设计.北京: 电子工业出版社, 2005:127~132.
- [28] 向晖,滕建辅,王承宁.基于 FPEG 和 2 位串行分布式算法的实时高速二维 DCT/IDCT 处理器研制.电子科学学刊[J], 1999,21(6):797~805.
- [29] 李莉,宁帆,魏巨升.基于 DA 算法的二维 DCT 的 FPGA 实现.多媒体技术.2006.(10):44~46.
- [30] 刘楠. 图像解码中 IDCT 的 FPGA 实现. 现代电子技术.2006.223:71~73.
- [31] Cyclone Programmable Logic Device Family Data Sheet. Altera.
- [32] 王建校, 危建国. SOPC 设计基础与实践. 西安电子科技大学出版社.2006.1:1~21.
- [33] 谭会生,张昌凡.EDA 技术及应用. 第一版. 西安:西安电子科技大学出版社.2001.197~206.
- [34] 周博. 邱卫东. 陈燕等. 挑战 SOC-基于 Nios 的 SOPC 设计与实践.清华大学出版社.2004.7:64~117.
- [35] 吴继华,王诚.Altera FPGA/CPLD 设计(高级篇).人民邮电出版社.2005.7:263~265.
- [36] 吴继华,王诚.Altera FPGA/CPLD 设计(高级篇).人民邮电出版社.2005.7:272~278.

附 录

A: 作者在攻读硕士学位期间发表的论文目录

- [1] 何伟, 廖彦, 张玲, 张烽. 基于 FPGA 的实时 M-JPEG 压缩编码. 电视技术. (录用)

B: JPEG 解码接口 Verilog HDL 程序

```

module fore_decoder(  clk,
                      rst,

                      m_read,
                      m_readdata,
                      m_address,
                      m_waitrequest,
                      s0_read,
                      s0_readdata,
                      s0_write,
                      s0_writedata,
                      s0_address,
                      s0_chipselect,
                      s1_read,
                      s1_readdata,
                      s1_write,
                      s1_writedata,
                      s1_address,
                      s1_chipselect,

                      rd_address,
                      rd_jpegdata,
                      rd_en,
                      start_frame,
                      newdata_flag,
                      end_frame
                      );

input                clk;
input                rst;

output              m_read;
input  [31:0]        m_readdata;
output  [31:0]        m_address;
input                m_waitrequest;
input                s0_read;
output  [7:0]         s0_readdata;
input                s0_write;
input  [7:0]         s0_writedata;
input  [7:0]         s0_address;
input  [1:0]         s0_chipselect;

```

```

input          s0_chipselect;
input          s1_read;
output [7:0]   s1_readdata;
input          s1_write;
input [7:0]    s1_writedata;
input          s1_address;
input          s1_chipselect;

input [5:0]    rd_address;
output [31:0]  rd_jpegdata;
input          rd_en;
output         start_frame;
input          newdata_flag;
input          end_frame;

wire [31:0]    m_address;
reg           m_read;
reg [7:0]      s0_readdata;
reg [7:0]      s1_readdata;
wire [31:0]    rd_jpegdata;

reg [31:0]     address_base;
reg [31:0]     address_offset;

reg [7:0]      temp;
reg            decoder_flag;
reg            start_frame;

reg            switch;
reg            end_flag;
reg [5:0]      wr_address;
reg            wr_en;

wire           rd_en1, rd_en2;
wire           wr_en1, wr_en2;
wire [31:0]    rd_jpegdata1, rd_jpegdata2;

reg [5:0]      CS;
reg [5:0]      NS;

reg            flag;
reg            mcu_end_flag;
reg            pre_flag;

```

```

reg    [2:0]    cnt;
reg    [31:0]   write_data;

always@(posedge clk)
  if(rst)
    s1_readdata <= 8'b0;
  else if(s1_chipselect && s1_read && s1_address==0)
    s1_readdata <= temp;

parameter      IDLE          = 6'b000000,
                FRAME_DETECT = 6'b000001,
                WAIT_FIRST   = 6'b000010,
                PRE_DECODE    = 6'b000100,
                GET_DATA      = 6'b001000,
                WAIT_RD       = 6'b010000,
                END_DETECT    = 6'b100000;

always@(posedge clk or posedge rst)
  if(rst)    CS <= IDLE;
  else      CS <= NS;

always@(CS or decoder_flag or flag or end_frame or wr_address or rd_address or
newdata_flag or pre_flag)
  case(CS)
    IDLE:          NS <= FRAME_DETECT;
    FRAME_DETECT:  if(decoder_flag)NS<=WAIT_FIRST;
                  else          NS<=FRAME_DETECT;
    WAIT_FIRST:    if(flag)      NS<=PRE_DECODE;
                  else          NS<=WAIT_FIRST;
    PRE_DECODE:    if(&wr_address) NS<=GET_DATA;
                  else          NS<=WAIT_RD;
    GET_DATA:      if(&wr_address) NS<=END_DETECT;
                  else          NS<=WAIT_RD;
    WAIT_RD:       if(pre_flag&&flag)
                      NS<=PRE_DECODE;
                  else if(flag)  NS<=GET_DATA;
                  else          NS<=WAIT_RD;
    END_DETECT:    if(end_frame) NS<=IDLE;
                  else if(rd_address==6'b111111 && newdata_flag)
                      NS<=GET_DATA;
                  else          NS<=END_DETECT;
    default:       NS<=IDLE;
  endcase

```

```

always@(posedge clk)
  case(CS)
    IDLE:      begin
      address_base  <= 32'b0;
      s0_readdata   <= 8'b0;
      decoder_flag  <= 1'b0;
      start_frame   <= 1'b0;
      wr_address    <= 6'b0;
      write_data    <= 32'b0;
      wr_en         <= 1'b0;
      address_offset<= 32'b0;
      m_read        <= 1'b0;
      temp          <= 8'b0;
      switch        <= 1'b0;
      flag          <= 1'b0;
      mcu_end_flag  <= 1'b0;
      pre_flag      <= 1'b0;
      cnt           <= 3'b0;
    end
    FRAME_DETECT:
      begin
        if(decoder_flag==0) begin
          if(s0_chipselect) begin
            if(s0_write) begin
              case(s0_address)
                2'b00:    address_base[7:0]   <= s0_writedata;
                2'b01:    address_base[15:8]  <= s0_writedata;
                2'b10:    address_base[23:16] <= s0_writedata;
                2'b11:    address_base[31:24] <= s0_writedata;
                default:   address_base        <= 32'b0;
              endcase
            end
          else if(s0_read) begin
            case(s0_address)
              2'b00:    s0_readdata <= address_base[7:0] ;
              2'b01:    s0_readdata <= address_base[15:8] ;
              2'b10:    s0_readdata <= address_base[23:16];
              2'b11:    s0_readdata <= address_base[31:24];
              default:   s0_readdata <= 8'b0;
            endcase
          end
        end
        if(s1_chipselect) begin

```

```

        if(s1_write && s1_address==0) begin
            decoder_flag <= s1_writedata[0];
            temp          <= s1_writedata;
        end
    end
end
else begin
    decoder_flag    <= 1'b0;
    address_offset  <= 32'b0;
    m_read          <= 1'b1;
    wr_address      <= 6'b0;
    wr_en           <= 1'b0;
    flag            <= 1'b0;
    cnt             <= 3'b0;
end
end
WAIT_FIRST: begin
    wait_master_read;
end
PRE_DECODE:
begin
    wr_en          <= 1'b0;
    flag           <= 1'b0;
    cnt            <= 3'b0;
    if(&wr_address) begin
        pre_flag    <= 1'b0;
        address_offset<= address_offset + 32'd4;
        m_read      <= 1'b0;
        wr_address   <= 6'b0;
        switch       <= !switch;
        start_frame  <= 1'b1;
        mcu_end_flag <= 1'b1;
    end
    else begin
        pre_flag    <= 1'b1;
        address_offset<= address_offset + 32'd4;
        m_read      <= 1'b1;
        wr_address   <= wr_address    + 6'b1;
    end
end
end
GET_DATA:
begin
    start_frame    <= 1'b0;

```

```

        wr_en          <= 1'b0;
        flag           <= 1'b0;
        cnt            <= 3'b0;
    if(mcu_end_flag) begin
        m_read         <= 1'b1;
        mcu_end_flag   <= 1'b0;
    end
    else if(&wr_address) begin
        address_offset<= address_offset + 32'd4;
        m_read         <= 1'b0;
        wr_address     <= 6'b0;
        mcu_end_flag   <= 1'b1;
    end
    else begin
        address_offset<= address_offset + 32'd4;
        m_read         <= 1'b1;
        wr_address     <= wr_address      + 6'b1;
    end
end
WAIT_RD:    begin
    wait_master_read;
end
END_DETECT:
    if(rd_address==6'b111111 && newdata_flag)
        switch        <= !switch;
    else
        temp           <= 8'b0;
default:
    begin
        address_base   <= 32'b0;
        s0_readdata    <= 8'b0;
        decoder_flag   <= 1'b0;
        start_frame    <= 1'b0;
        wr_address     <= 6'b0;
        write_data     <= 32'b0;
        wr_en          <= 1'b0;
        address_offset<= 32'b0;
        m_read         <= 1'b0;
        temp           <= 8'b0;
        switch         <= 1'b0;
        flag           <= 1'b0;
        mcu_end_flag   <= 1'b0;
        pre_flag       <= 1'b0;
    end
end

```

```

        cnt                <= 3'b0;
    end

endcase

task wait_master_read;
    case(cnt)
        3'b000:    if(!m_waitrequest) begin
                    flag                <= 1'b1;
                    wr_en                <= 1'b1;
                    write_data            <= m_readdata;
                    m_read                <= 1'b0;
                end
                else
                    cnt                <= cnt + 3'b1;
            3'b001,
            3'b010,
            3'b011,
            3'b100,
            3'b101,
            3'b110,
            3'b111:    if(flag) begin
                    flag                <= 1'b0;
                    wr_en                <= 1'b0;
                    cnt                <= 3'b0;
                end
                else if(!m_waitrequest) begin
                    flag                <= 1'b1;
                    wr_en                <= 1'b1;
                    write_data            <= m_readdata;
                    m_read                <= 1'b0;
                end
                else
                    cnt                <= cnt + 3'b1;
            endcase
    endtask

assign    m_address    = address_base + address_offset;

assign    rd_en1 = switch? rd_en : 1'b0 ;
assign    rd_en2 = switch? 1'b0  : rd_en;
assign    wr_en1 = switch? 1'b0  : wr_en;
assign    wr_en2 = switch? wr_en : 1'b0 ;
assign    rd_jpegdata = switch? rd_jpegdata1 : rd_jpegdata2;

```



```
fore_pegdata JPEGDATA1(  
    .clock      (clk),  
    .data       (write_data),  
    .rdaddress  (rd_address),  
    .rden       (rd_en1),  
    .wraddress  (wr_address),  
    .wren       (wr_en1),  
    .q          (rd_pegdata1));
```

```
fore_pegdata JPEGDATA2(  
    .clock      (clk),  
    .data       (write_data),  
    .rdaddress  (rd_address),  
    .rden       (rd_en2),  
    .wraddress  (wr_address),  
    .wren       (wr_en2),  
    .q          (rd_pegdata2));
```

```
always@(posedge clk or posedge rst)  
    if(rst)  
        end_flag <= 1'b0;  
    else if(start_frame)  
        end_flag <= 1'b0;  
    else if(end_frame)  
        end_flag <= 1'b1;
```

```
endmodule
```

作者：[廖彦](#)

学位授予单位：[重庆大学](#)

相似文献(5条)

1. 期刊论文 [李清华](#), [余松煜](#), [严枫](#) 一种通用的基于MVP的图像编解码系统的硬件实现 -[数据采集与处理](#)1999, 14(2)
德州仪器公司推出的TMS320C80, 为需要高计算能力的算法(如视频压缩算法)提供了新的思路. 本文给出一种基于C80的硬件系统设计, 也可以作为一种通用系统用于其他场合.
2. 期刊论文 [薛文通](#), [宋建社](#), [袁礼海](#), [沈涛](#), [XUE Wen-tong](#), [SONG Jian-she](#), [YUAN Li-hai](#), [SHEN Tao](#) 基于整数到整数小波变换的图像编解码器设计 -[系统工程与电子技术](#)2005, 27(4)
为设计具有低运算复杂度和渐进传输特性的图像编解码系统, 采用对图像进行整数到整数小波变换, 变换后的系数采用空间方向树结构进行了重新组织. 整数到整数小波变换的优点使编码器的运算量有大幅度降低, 空间方向树结构使编码器获得了嵌入式码流. 试验表明, 该系统可在一定程度上满足图像信息在存储、网络浏览以及传输方面的需求.
3. 学位论文 [闫允一](#) 基于小波的图像处理研究 2004
小波由于其特有的多分辨特性在图像处理领域获得了广泛的应用, 该文从提升、超分辨率、图像编解码三个方面详细研究了小波的应用. 该文提出了“提升对”(Lifting pair)和“提升算子”(Lifting operator)的概念, 进而引申出欠提升(sub-lifting)的概念, 简化了提升变换结构, 并证明“提升算子”和“反提升算子”可以实现完全重构; 从“提升算子”出发, 给出了提升算子的提升定理, 并由定义和定理给出了6个推论, 简单明了地从理论上证明了一次完整的提升可以由两次欠提升完成、相邻两次欠提升的顺序性和多级提升中提升算子的串联存在顺序性等未曾给出理论证明的命题. 该文还对提升方案和Mallat算法进行了简单比较, 并给出了实验结果. 该文提出了基于小波的图像超分辨率重建算法的一般框架, 并针对传统图像超分辨率重建中的高频能量增大缺点, 提出了改进的适当降低高频能量的SHR算法; 针对高频微小细节无法恢复的缺点提出了一种利用小波重建高频细节的SHW算法. 实验证明SHR算法和SHW算法比传统的重建算法性能均有不同程度提高. 该文还通过对实验结果的分析, 提出获得较好重建效果的一个必要条件——图像达到一定分辨率, 并对此进行了理论分析. 该文还设计了以ADV612为核心压缩芯片的视频图像编解码系统.
4. 会议论文 [王建东](#), [徐伯夏](#) 基于DM642的实时图像编解码系统研究 2007
研究了基于DM642的实时图像编解码系统, 实现了图像的采集、编码、CPCI传输、解码、实时显示、字符叠加. 其中, 基于DM642主DMA模式的CPCI总线传输技术、EDMA循环链接技术、纯数字域字符叠加技术体现了本研究的先进性. 本系统最终实现了图像的实时压缩解压缩, 具有较大压缩比和良好的图像质量.
5. 学位论文 [甘平](#) 基于DSP技术的视频图像压缩系统的实现及研究 2004
本文着重基于DSP技术的图像压缩系统的实现及研究. 以TI的TMS320C5402 DSP为处理器设计低成本视频图像压缩板. 视频图像压缩编解码一方面要将模拟视频信号转化为数字信号来传输, 另一方面利用图像压缩技术减少数据量. 本文将提供一套基于通用DSP和CPLD的图像编解码系统设计方案, 并阐述了具体实现. 同时阐述了压缩编码系统的设计和其中的关键技术问题, 并结合图像编码算法和DSP流水线的特点进行程序优化. 考虑到高速的通用微处理器DSP芯片易于满足图像处理中大数据量、复杂运算、实时性强、高传输率、设计成本等要求, 本文设计了一种以TMS320C5402为核心处理器, 基于CPLD计算机系统模型的脱机实时图像处理系统. 调试结果表明系统设计结构可行, 取得了一定效果, 为以后算法研究提供了验证平台.

本文链接: http://d.g.wanfangdata.com.cn/Thesis_Y1140972.aspx

授权使用: 陕西理工学院(sxlgxy), 授权号: 2d80ea09-7618-4770-988a-9df2010cba2e

下载时间: 2010年9月15日