# 实验报告模板

华南理工大学

《计算机组成原理》课程实验报告

实验题目： ARMv8 CPU

姓名： 陈映松 学号： 202130442563

班级： 21计科(2)班 组别：

合作者：

指导教师： 张齐

|  |
| --- |
| **实验概述** |
| 【实验目的及要求】  实验目的：   1. 掌握ARMv8指令集及其特点 2. 通过三个阶段逐步构建一个硬布线控制器版本的ARMv8“最小CPU” 3. 掌握二级/三级流水线ARMv8 CPU   实验要求：   1. 学习ARMv8指令集和汇编 2. 学习ARMv8 CPU中的程序控制 3. 学习ARMv8 CPU中的逻辑和算术运算 4. 学习ARMv8 CPU中的访存和IO   【实验环境】  Proteus 8.1  Digiblock  Java配置环境 |
| **实验内容** |
| 【实验过程】   1. 实验步骤：（“实验步骤”）   **步骤1：ARMv8 指令和汇编**  1.ARM (Advanced RISC Machine)体系架构是世界上应用最广泛的商业 CPU 体系架构，  在32位嵌入式CPU市场占比高达75%。ARM指令集各个版本的历史和相互关系如下图5.1.1 所示，与 X86 指令集一样，每一代 ARM 指令集都必须兼容以前所有的 ARM 指令集版本。从 ARMv8 指令集开始，ARM 同时支持 32 位和 64 位 CPU 架构。  2. 在Digiblock 菜单栏的 Assembly 菜单中，RARS，MARS 和 ARMv8 选项分别对应  RISC-V 和 MIPS 指令集模拟器，以及 ARMv8 指令集（最小子集）汇编器。点击 Assembly 菜单的 ARMv8 选项，打开 SCUT ARMv8 assembler，如下图所示。在左侧的 assembler directive 区域，可以输入带有标号的 ARMv8 汇编指令代码，然后按左下方的compile 按钮，在右侧的 machine code 区域，可以得到每一行 ARMv8 汇编指令对应的 32 位机器码指令（4 位二进制数字一组，分成八组，方便转换十六进制）。如果想了解 ARMv8 指令的汇编代码格式，可以按正下方的 sample code 按钮，弹出的窗口中包含了所有的指令的全部指令格式参考样例。值得注意的是，若在 assembler directive 区域输入空行，转换到 machine code 区域机器码显示的时候，会自动消掉空行。这会影响左右区域的汇编指令和机器指令对齐。强烈建议，在左侧 assembler directive 区域输入汇编指令代码的时候，不要留空行，以避免影响机器指令的对齐。此外，若在左侧 assembler directive 区域输入了错误的汇编格式语句，则在右侧 machine code 区域的对应行会提示“ERROR: 错误类型”。如下图所示，第 3 行，跳转的标号在程序中并不存在；第 5 行，ANDS 指令格式不完整，缺失参数。注意：指令/行号的大小写不一致（例如指令使用小写），中文字符（例如中文的“：”）和不存在的标号（例如“B echo”但是程序内不存在“echo: ….”语句）都会触发错误提示。  3. 本实验将支持一个 32 位 ARMv8 指令集最小子集的实现，CPU 基于硬布线控制器架构，拥有 64 位数据总线，支持 64 位数据的运算和存储。与前述控制器实验类似，本实验分成三个阶段逐步实现一个完整的 ARMv8 架构 CPU：第一阶段实现基本的程序控制功能；第二阶段增加算术和逻辑运算的功能；第三阶段实现访问存储器的功能。最后，本实验给出ARMv8 指令集（最小子集）CPU 的二级和三级流水线架构的实现。  **步骤2：ARMv8 CPU—程序控制**  1.第一阶段程序控制指令族  2.与上述程序控制指令集对应的 ARMv8 架构 CPU（第一阶段），如图所示。  3. 上述 CPU 的数据通路中，微操作信号只有四个：pcsel，Dsel，brk 和 regw，全部由硬布线逻辑生成，如下图所示。其中 pcsel=0 即 PC+1，顺序执行；pcsel=1 即无条件跳转；pcsel=2即有条件跳转，且需要根据运算状态字 VNCZ 判断是否跳转（本阶段 VNCZ 由端口赋值直接给出）。Dsel=0 即正常运行，Dsel=1 则寄存器堆 regw 的 Rw=30（此时指令是 BL/BLR，即保存断点，写入 X30 寄存器）。因为本阶段只除了 zero 情形（X31 寄存器是 0 寄存器，不允许写入），仅 Dsel=1 时需要写入寄存器，所以当 Dsel=1 且非 zero 情形，则 regw=1。  4. 采用上述扩充的指令集，编写了一段跳转程序 jump.hex，实现了程序在存储空间不断跳转，直到遇到 brk 暂停。  5. 请将上述 jump.hex 二进制代码加载到上图 5.1.1 所示电路的的指令存储器 Irom 中。完成加载后启动运行，注意观察 Irom 存储器的输出（指令）和输入（地址）：此时，指令计数器（地址）PC=0（Irom 输入端 A），Irom 输出端 D 显示第一条指令 I1=14000007（“B echo”）。然后，连续点击输入端 hand，使 hand 端口先高（红色）后低（绿色），完成了一个时钟 clk信号的上升沿和下降沿。当 hand 端口变红（上升沿）的时候，可以观察到 Irom 的输入（地址）A=7，输出（指令）D=0x54FFFF64（“B.MI alpha”）。说明刚刚第一条指令（A=0）“B echo”执行完了，所以无跳转到第二条指令(A=7) “B.MI alpha”。因为(A=7) “B.MI alpha”的指令是有条件跳转，查阅 B.cond 指令格式可以知道此时必须有 N=1 才能跳转，因此，此时需要手动在 VNCZ 端口输入 4，使得 N=1，然后再 clk 一下，如下图所示，第三条指令跳转到（A=2）D=0x9400002（alpha: BL bravo）。此时，说明条件跳转成功了。如果不设置 VNCZ(即 N=0)，可以试试第三条指令是什么。 118随后可以通过 hand 端口 clk“步进”运行。如果需要自动运行，需要把 model 端口电平设置为高（变红）。此时，时钟信号由 hand 端口下访的 clock 组件提供，clock 组件已经内设为 Freq=2(即每秒 2 次时钟方波信号)，而且设置 start real time clock，即启动开始运行。用表格记录每一次 clk 后，Irom 存储器的输出（指令）和输入（地址）的变化。  **步骤3：ARMv8 CPU—逻辑和算术运算**  1. 第二阶段逻辑和算术运算指令族，其中红色部分的指令其实是其他指令的特例，没有独立 OP 码。CMP/CMPI 指令是对应的 SUBS/SUBIS 指令的特例，两操作数相减的结果写入X31 寄存器。因为 X31 寄存器是 0 寄存器，所以无法写入，仅改变标志位。 而 MOV 指令则是其中一操作数为 0（X31 寄存器）的 ORR 指令的特例，操作结果即为另一操作数。  2. 与上述逻辑和算术运算指令集扩展对应的 ARMv8 架构 CPU（第二阶段）,如图所示。  3. 上述 CPU 的数据通路中，微操作信号除了第一阶段的四个：pcsel，Dsel，brk 和 regw，还增加了 aluop 信号，用来控制运算的种类，由硬布线逻辑生成，如下图 5.3.2 所示。所有算术运算与逻辑运算都同步执行，由 aluop 根据指令 OP 码选择具体是哪个运算的结果输出。由于所有算术和逻辑运算的时刻都有 aluop≠0 且都需要写入寄存器，所以操作信号 regw=1不仅仅是 Dsel=1 且非 zero 情形（X31 寄存器写入），还要包括所有 aluop≠0 的情形。  4. 采用上述逻辑和算术运算指令集扩展的指令集，编写了一段连续加法程序 op1.hex，  实现了“1+2+3+4+5=?”，如图所示。当然，连续加法的代码不是唯一，下图所示的 op2.hex 实现了与 op1 相同的功能：121最后，依靠加法和逻辑运算指令中的移位选项，可以方便的设计出四位原码乘法op3.hex，实现了“1011×1101=？”请将上述 OP1.hex 二进制代码加载到上图 5.3.1 所示电路的指令存储器 Irom 中。完成加载后启动运行，注意观察 Irom 存储器的输出（指令）和输入（地址）：此时，指令计数器（地址）PC=0（Irom输入端 A），Irom输出端 D显示第一条指令 I1= D2800020（“MOVZ X0, #11”）,即赋值 X0=1。此时，在仿真运行状态下， 右键点击 regs 组件，弹出一个实时数据框，显示当前寄存器堆情况。点击 hand 端口 clk 步进，Irom 显示(A=1)D=D2800021（“MOVZ X0, #13”），观察 regs实时数据框内的第一个寄存器（即 X0）的值变为 11，与上一条指令“MOVZ X0, #11”吻合。后续指令步进执行一次，用表格记录每一次 clk 后，寄存器堆 regs 的实时数据框内的寄存器实时数据变化，可以看出每条指令执行的结果。当运行到(A=6)D=54000040，该指令是“B.EQ beta”即 Z=0 时跳转。因为前一条指令(A=5)D=F1000442（“SUBIS X2,X2,#1”）。因为 X0 初始值 4，当循环第 4 次执行到该指令处时，运行结果 X2=0，触发标志位 Z=0。指令“B.EQ beta”满足条件发生跳转。注意，ARMv8CPU-II.dig 中 VNCZ 已经由运算指令的结果自动生成，保存在 VNCZ 寄存器内。。所以有条件跳转指令 B.cond 执行结果取决于前面最近一条运算指令对 VNCZ 寄存器的影响，一般 B.cond令前面必须放一条影响符号位的运算指令。  **步骤4：ARMv8 CPU—访存和IO**  1.第三阶段访存和 IO 指令族：LDUR/STUR 均支持基址寻址方式。  2.与上述访存指令集扩展对应的 ARMv8 架构 CPU（第三阶段）如图所示。  3. 如下图 5.4.2 所示，ARMv8CPU（第三阶段）访问存储器和访问 IO 是统一编址的：在24bit 逻辑存储空间中，000000H~7FFFFFH 为 Drom 空间，800000H~FFFFEFH 为 Dram 空间，最高的 16 个地址 FFFFF0H~FFFFFFH 留给 IO 访问用。如上图5.4.1 所示的 CPU 数据通路中，由硬布线逻辑生成的微操作信号除了第二阶段的五个信号：pcsel，Dsel，brk，regw 和aluop，还增加了 memw 信号，用来控制写入存储器或 OUT 端口。由于所有算术运算、逻辑123运算和存储器/IN 端口读入操作都同步执行，由 aluop 选择具体是哪个操作的结果输出。所以，aluop 的硬布线控制逻辑需要扩充，以容纳存储器操作，如下图 5.4.2 所示。此外，memw信号直接由 STUR 指令的 OP 码字段生成。  4. 采用上述扩充的访存/IO 指令，编写了一段数组转移程序 mem.hex，实现了从地址  [0x01]开始的数组顺序转移到从地址[0x10001]开始的数据空间，如下图所示。同时，编写了一段 IO 访问程序 io.hex：从 IO 地址[F1]的读端口 IN 输入数据到 X0，再将X0 据分别赋给存储器地址[0x10001]和端口 OUT 124。请将上述 mem.hex 二进制代码分别加载到上图 5.4.1 所示电路的指令存储器 Irom 中。因为 mem.hex 要把数据从 Drom 搬移到 Dram，所以先右键点击 Drom，从地址 0 开始，顺序手动输入数组：0x08、0xAA、0xFF、0x80、0x55。完成加载后启动运行，注意观察 Irom 存储器的输出（指令）、Drom 存储器输出和 Rram存储器内容（数据），以及当前寄存器堆 regs 的状况（数据）。  二、实验数据：（“实验步骤”里的列表部分）  **步骤2：ARMv8 CPU—程序控制**   1. 首次将jump.hex文件加载到指令存储器中运行后，Irom存储器的输入（指令）和输出（地址）分别是：**0**和**0x14000007。**然后再连续点击输入端hand，观察到当hand端口第一次处于上升沿时，Irom存储器的输入（指令）和输出（地址）分别是：**7**和**0x54FFFF64。** 2. 若第二次不设置VNCZ，则第三条指令是**A=8**，地址变成**0xD4200000**。分析可知此时程序未发生跳转，则顺序执行。 3. 用表格记录每一次clk后，Irom存储器的输入和输出变化。  |  |  |  | | --- | --- | --- | | clr | 输出（指令） | 输入（地址） | | 1 | 0x14000007 | 0 | | 2 | 0x54FFFF64 | 7 | | 3 | 0x94000002 | 2 | | 4 | 0xD503201F | 4 | | 5 | 0xD61F03C0 | 5 | | 6 | 0xD503201F | 3 | | 7 | 0xD503201F | 4 | | 8 | 0xD61F03C0 | 5 | | 9 | 0xD503201F | 3 |   从表格数据可知，最终程序会在4，5，3三条指令之间一直循环执行。  **步骤3：ARMv8 CPU—逻辑和算术运算**   1. 首次将OP1.hex文件加载到指令存储器中运行后，Irom存储器的输入（指令）和输出（地址）分别是：**0**和**0xD2800020。** 2. 点击hand端口clk步进，Irom显示地址为：**0xD2800021**。 3. 用表格记录每一次clk后，寄存器堆regs的实时数据框内的寄存器实时数据变化，用截屏展示9次clk下regs数据变化。数据如下：                     **步骤4：ARMv8 CPU—访存和IO**   1. 首次将mem.hex文件加载到指令存储器中运行后，Irom存储器的输入（指令）和输出（地址）分别是：**0**和**0xD2800020。** 2. 完成加载后启动运行，观察记录Irom存储器的输出（指令），Drom存储器输出和Rram存储器内容（数据），以及当前寄存器堆regs的状态（数据）。数据记录如下：  |  |  |  |  | | --- | --- | --- | --- | | clk | Irom输出 | Drom输出 | Rram数据(观察始终只有地址为1的位置有数据) | | 1 | 0xD2800021 | 1 | 0 | | 2 | 0xD2A00202 | 0x100000 | 0 | | 3 | 0xF8400020 | AA | 0 | | 4 | 0xF8001040 | 0 | AA | | 5 | 0xB1000421 | 2 | AA | | 6 | 0xD100143F | 0 | AA | | 7 | 0xF100143F | FFFFFFFFFFFFFFFD | FF | | 8 | 0x54000040 | 0 | FF | | 9 | 0x97FFFFFA | FF | FF | | 10 | 0xF8400020 | AA | 80 | |
| **小结** |
| 这是本次计组实验课程的最后一次大实验了，到此为止计组的实验课程也要告一段落了，但我觉得自己还远没有把计组实验的知识学习得足够透彻，当然还需要继续深度的思考与搜集资料丰富自我。这次的计组实验是要了解计组的一个核心部件—CPU的运行机制。任务教程的第一部分就是教会我们认识汇编代码，以及汇编代码运行时转换成的二进制机器代码的形式。第一次接触这些当然是一脸的陌生，但自己还是沉下心来认真研读指导书，一步一步跟着教程做，然后记录数据。通过以上的步骤，自己还是初步学会了一段程序是如何在CPU中运行，数据是如何在CPU与ROM，RAM之间流动的以及CPU的控制和调度全局的功能是如何体现的。 |
| **指导教师评语及成绩** |
| 评语：  成绩：           指导教师签名：                                                 批阅日期： |