**数字逻辑与处理器基础实验报告**

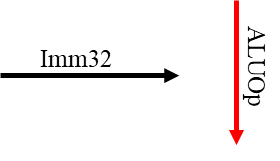
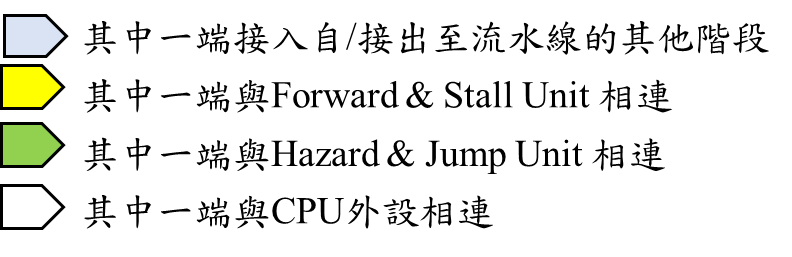
学号：2017011090

班级：无78

姓名：游子權

1. **实验名称：**流水线MIPS处理器
2. **实验目的：**于FPGA板上完成一基于MIPS指令集的五级流水线处理器，使其完成100个数据的排序功能。
3. **设计方案、原理说明与关键代码：**
   * 1. 设计原则
4. 分成五个阶段，分别为取指令（Instruction Fetch, IF）、解读指令（Instruction Decode, ID）、执行运算（Execution, EX）、存取资料记忆体（Memory Access, MEM）、写回寄存器堆（Write Back to register file, WB）。
5. 使用ROM作为Instruction Memory，采用于时钟上升沿读写的RAM作为Data Memory。采用于时钟上升沿写入的Register File。
6. 能采用转发（forwarding）时，采用完全的转发处理数据关联问题。
7. 不能转发时，采用尽量少的阻塞（stall）来解决竞争。
8. 分支（branch）、跳转（jump）类指令皆在ID阶段提前判断，在分支确认或跳转时取消IF阶段指令。
9. CPU设有七段数码管外设显示排序结果、设有定时器外设。
10. CPU可支援未知指令的异常与定时器中断。
    * 1. 系统框图：除五级流水线外，本人设置两个独立于流水线外的单元（后简称「独立单元」）来处理流水线的数据与控制冒险：分别为Hazard and Jump Unit、Forward and Stall Unit。

**【框图阅读说明】**

1. 线：线的粗细反映信号位宽，线的颜色本身没有特殊含义，但相同颜色的线上载有相同的讯号。故当图上不可避免需产生交叉时，本人用不同颜色的线区分不同信号。为免与相邻线产生混淆，在线所载讯号的名称一律记于水平线上方或铅直线右侧，如图3-1所示。

**▲图3-1：图例**

1. Input/ Output Port：形如

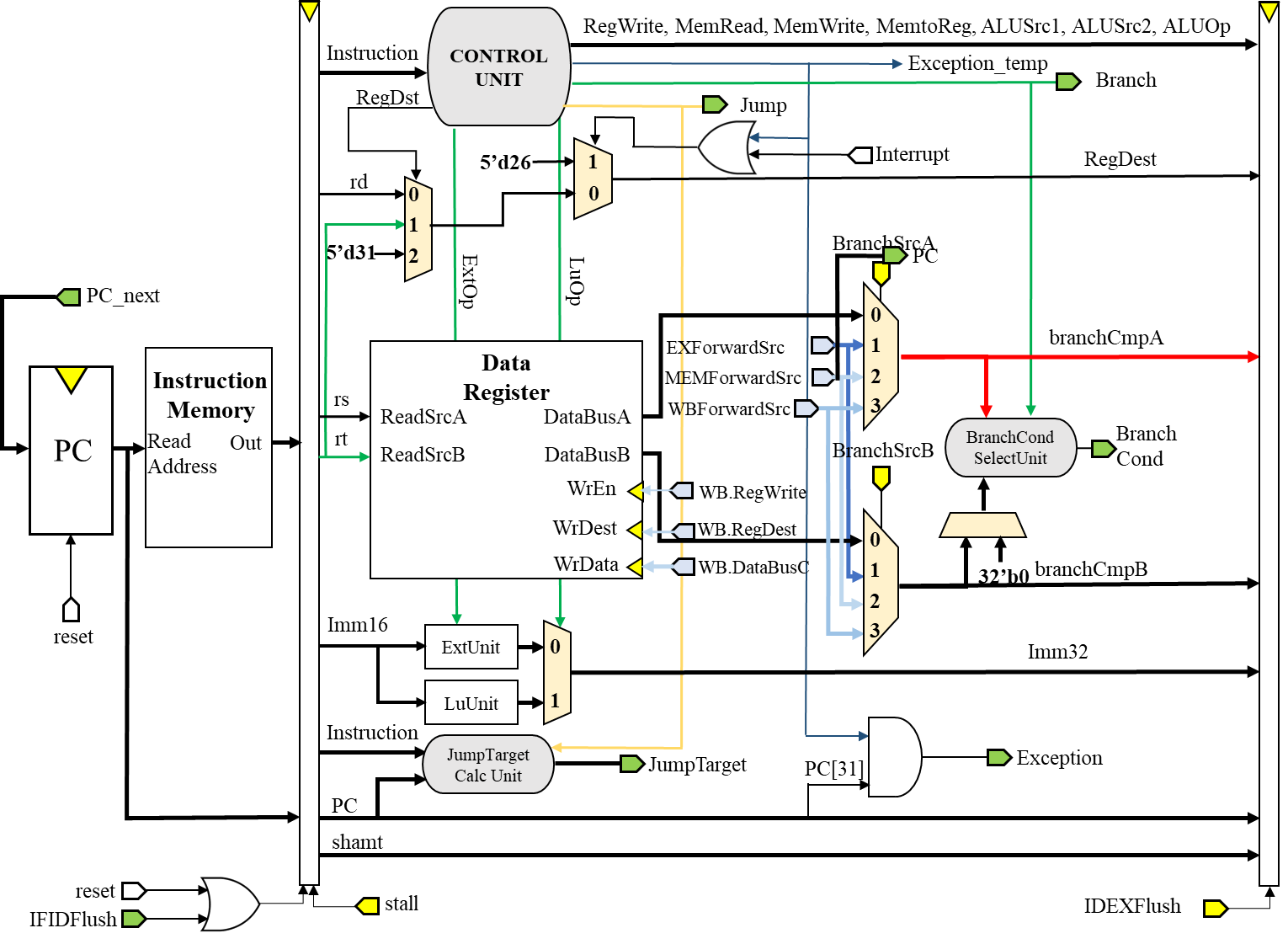
当引入/出流水线中其他阶段讯号时，或引入/出信号供Hazard and Jump Unit、Forward and Stall Unit使用时绘之。**同一阶段流水线的讯号引用不作此记号。**不同底色的Port代表不同意义，如图3-2所示。

**▲图3-2：图例**

1. 由于需使用到系统时钟信号(CLK)之装置甚多，本图中不一一标明，凡在时钟**上升沿**才写入新值的时序逻辑电路端口，均加入标志以资识别。

**【五级流水线框图】**

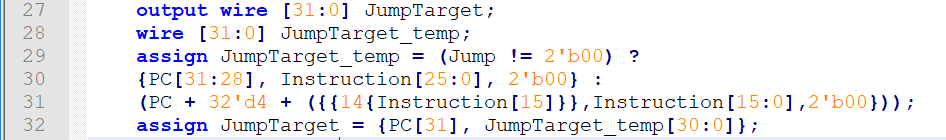
* + - 1. **IF、ID阶段**

****

**▲图3-3：流水线IF、ID阶段框图**

**说明：**

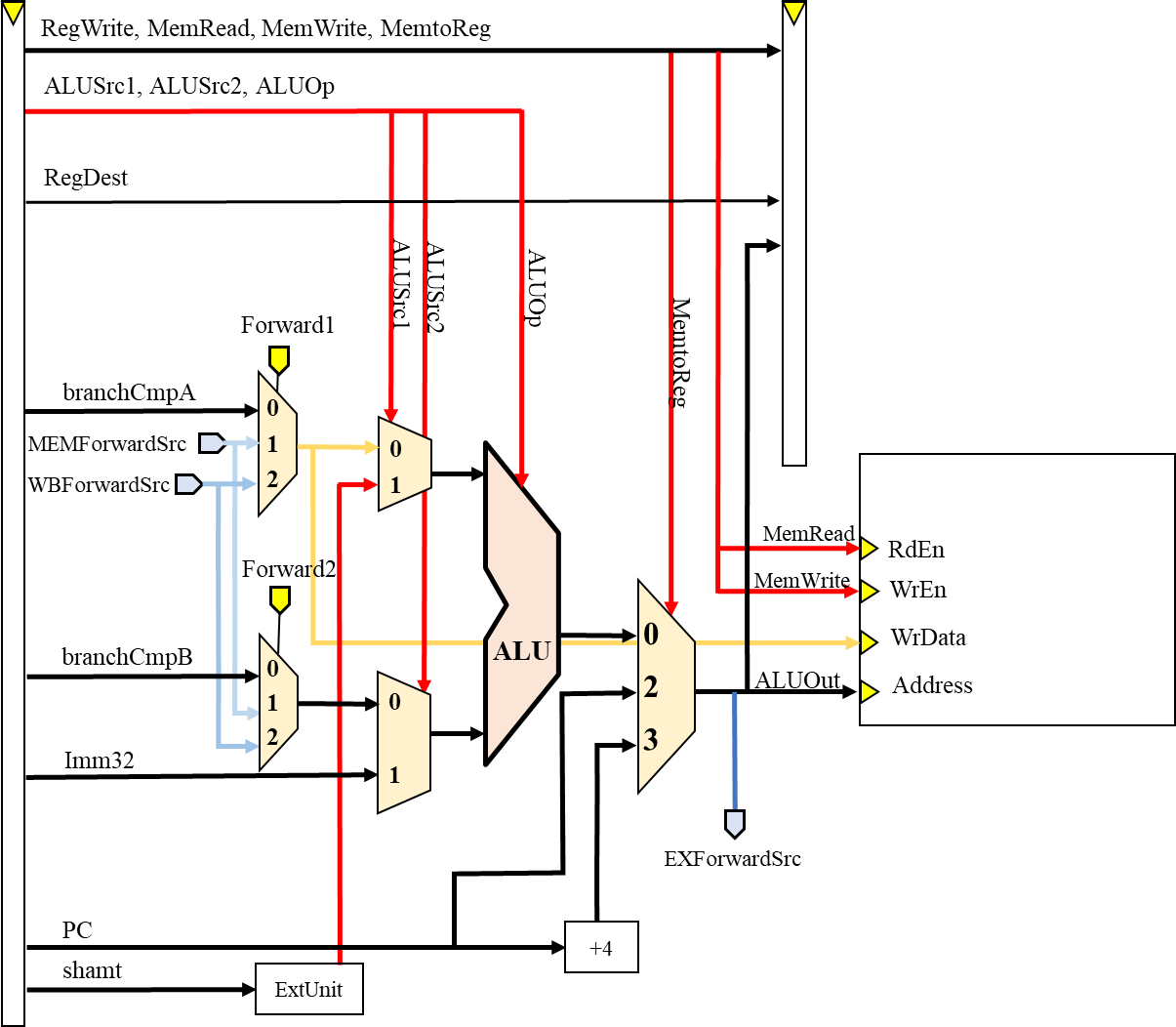
* 程序计数器（Program Counter, PC）的下一状态值PC\_next由Hazard and Jump Unit综合其他阶段之指令计算给出。
* Instruction Memory为ROM，存储已录入好之指令集。
* IF/ID Register在reset = 1或IFIDFlush = 1时，将下一状态的Instuction清零，但PC保持不变（以利stall时又遇Exception或Interruption，可以记录正常的PC）；stall=1时下一状态维持原状态不变。IFIDFlush由Hazard and Jump Unit计算给出。Stall由Forward and Stall Unit计算给出。
* ID阶段完成：指令译码、寄存器数据提取、立即数处理、转发、分支判断与跳转地址计算；遇例外与异常时，立刻改变原指令为jal Exception/Interrupt, $26：即将当前PC存入26号寄存器，并将跳转地址设至异常/中断处理程序入口地址。
* CONTROL UNIT为指令之控制信号译码单元，本人沿用单周期MIPS处理器中之译码单元，仅加入ori指令、branch及branch and link指令、异常与中断的相关判断。
* JumpTarget CalcUnit为计算跳转地址的单元，当Jump信号为非0时，采用伪直接寻址方法得到跳转址地；反之则采用PC相对寻址得出。跳转地址的PC[31]保持与本条指令之PC[31]相同。关键代码如下：（ID.v）



* BranchCond SelectUnit为根据Branch信号决定要选通哪种Branch条件（bne, beq, blez…等）的判断结果。关键代码如下：（ID.v）



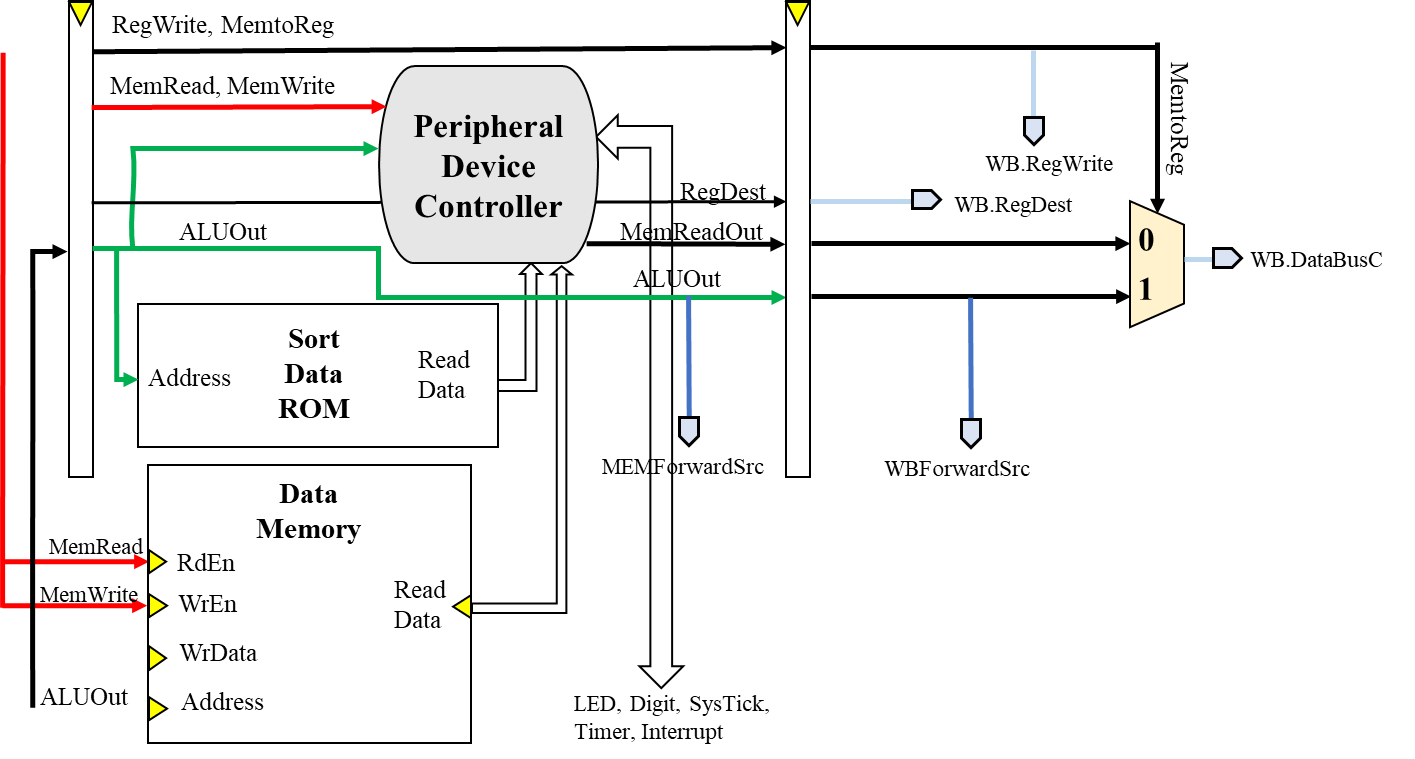
* + - 1. **EX阶段**

****

**▲图3-4：流水线EX阶段框图**

**说明：**

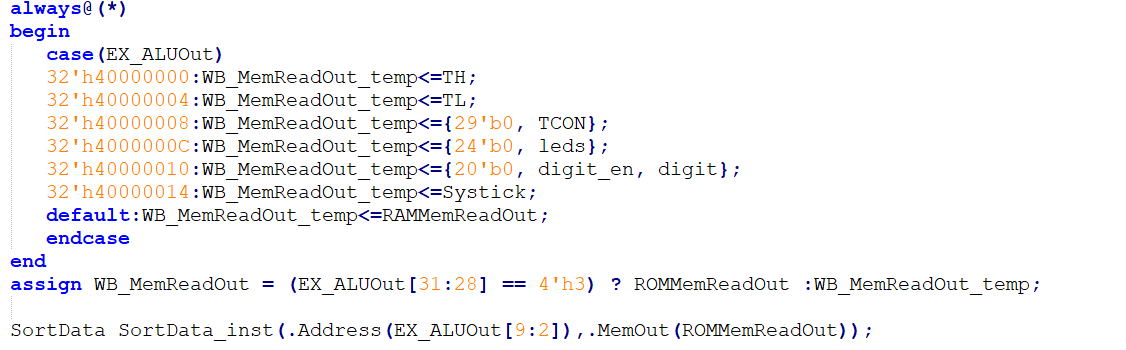
* EX阶段完成：转发、ALU计算。
* 原本MemtoReg的值只有0,1两种，当MemtoReg ==1时，把Memory读出的内容写回寄存器中，但为了因应例外与中断的以形下，须把目前的PC（中断）或下一条指令的PC（异常）存回寄存器中，为最小程度的改动流水线结构，在ID阶段中将MemtoReg信号扩为2位。原ALU输出端新增一多路选择器，当MemtoReg为2或3时分别表示该条指令解码时发生中断与异常，从而将相关的地址直接「看作」是ALU计算的输出，并交由下一阶段处理。
* 处理完因应异常与中断欲存储的内容后，MemtoReg可回复为1位。（指示欲选通Memory内容或ALUOut ）这个操作在EX/MEM Register中完成。当EX阶段MemtoReg != 1时，EX/MEM寄存器会在下个时钟上升沿向MEM阶段输出MemtoReg = 0。
  + - 1. **MEM、WB阶段**



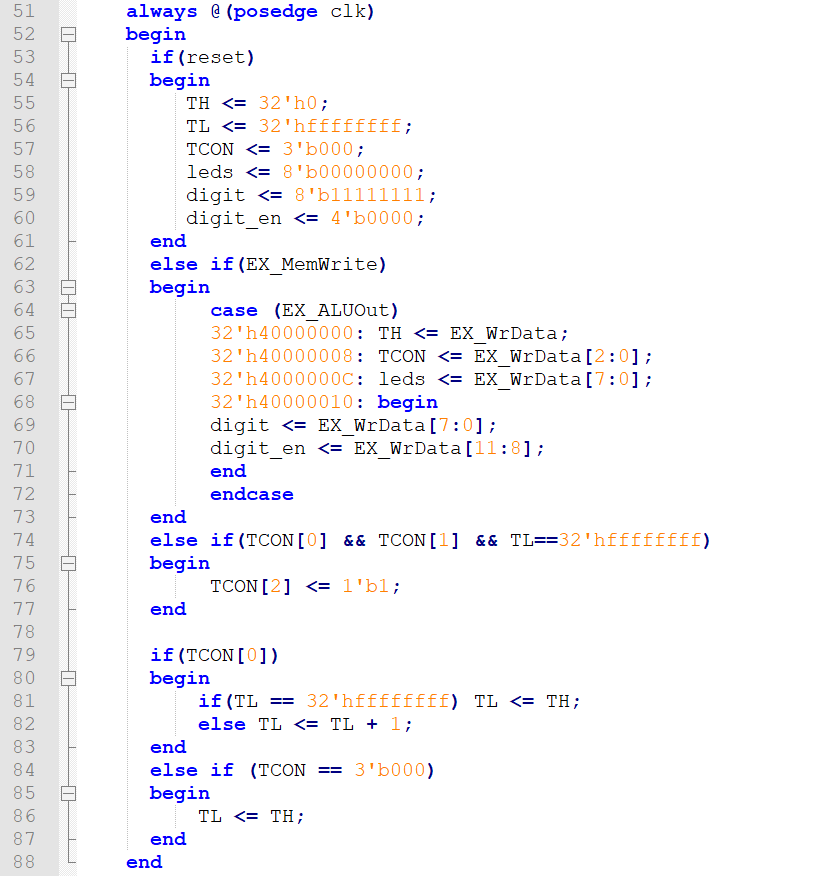
**▲图3-5：流水线MEM、WB阶段框图**

**说明：**

* MEM阶段完成：Memory与外设的写入与读出。
* Data Memory若采用FPGA提供的BRAM，可以大大减小FPGA的Flip Flop Slice的占用。为此，该RAM必须改为在时钟上升沿写入**及读取**，**且不得有清零端**。从而相关输入信号不必通过EX/MEM寄存器（因其在上升沿读出的Read Data，在下个上升沿到来之前不会改变），可直接接入Memory。
* Sort Data ROM是储存待排序数的唯读记忆体，其采用Vivado IP ROM实现。待排序数依照格式存于ain.coe文件中。
* Peripheral Device Controller是根据使用者输入的地址，判断是要从Data Memory中读数据（写数据已在时钟上升沿时完成）、从Sort Data ROM中读数据，还是要与各个外设交互（读/写）数据，其中外设的地址按照实验指导书要求，化成16进制下为4开头；本人所设之Sort Data ROM，其地址化为16进制为3开头，而Data Memory的RAM，因其仅能存512个字，故其地址化为16进制恒为0开头。本单元据此分析选通何种设备之数据。关键代码如下：（MEM.v）

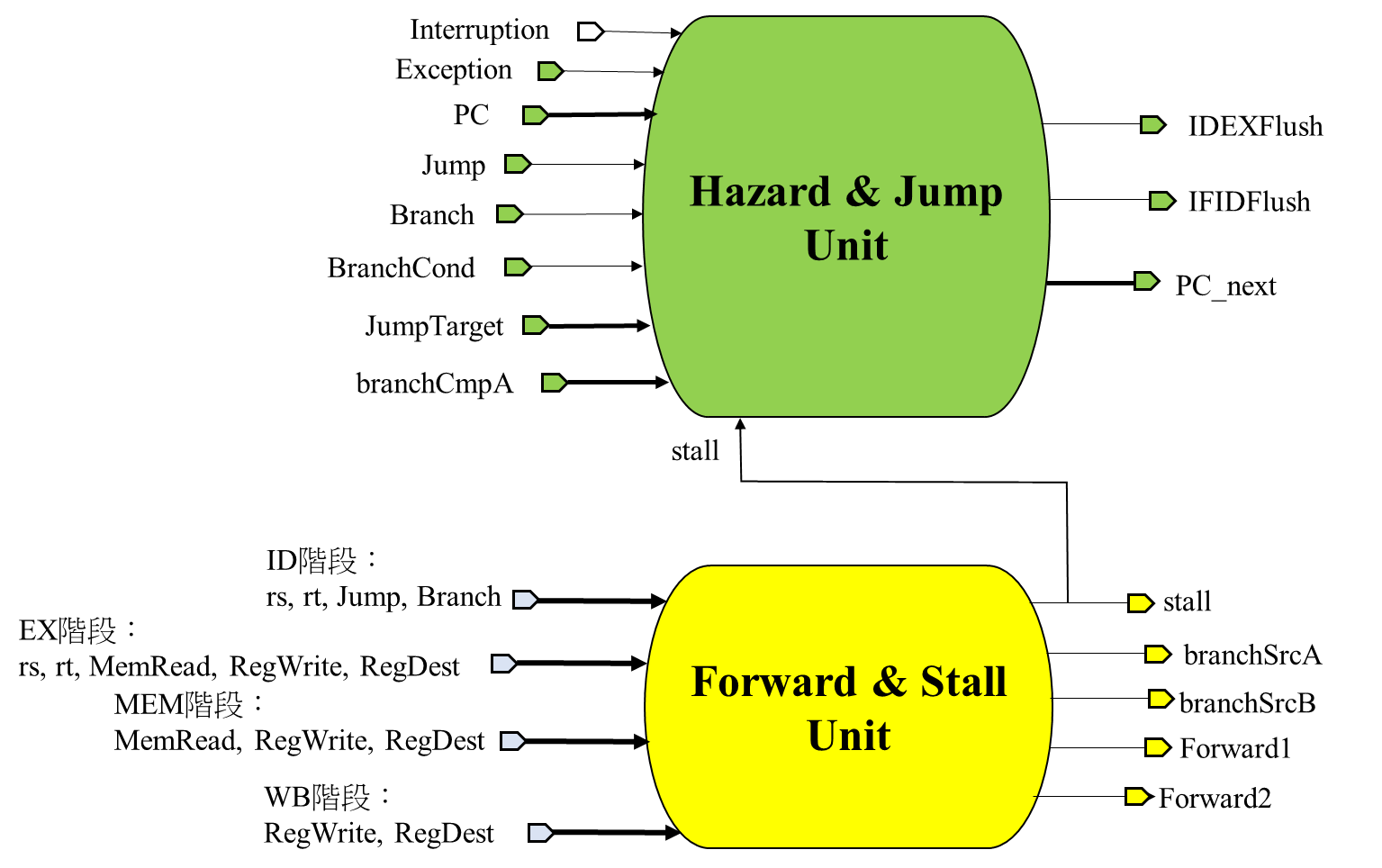


* 查定时器外设虽不属MEM阶段之单元，但考量其相关参数TL, TCON需由外设本身（硬件）与软件协同控制，故仍将其代码置于MEM.v中，与其他外设的写入共同撰写于同一过程块中。关键代码如下：（MEM.v）



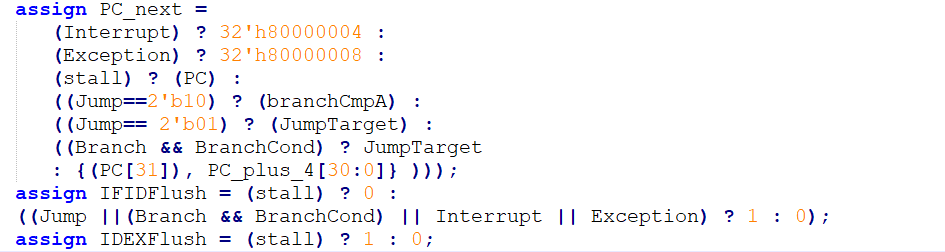
* WB阶段完成：回存寄存器数据的选通。（指示欲选通MemReadOut或ALUOut ）

**【独立单元框图】**



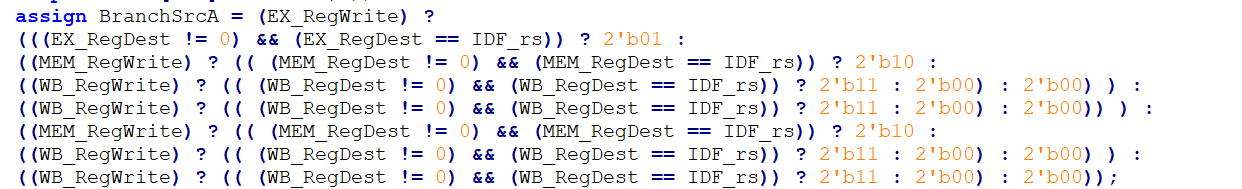
**▲图3-6：独立单元框图**

1. **Hazard and Jump Unit（hazardjump.v）：本单元分析各指令译码之结果，计算下个周期的PC。**

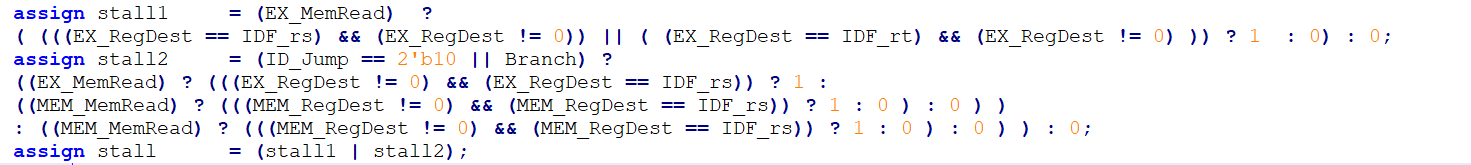
* 正常情形下：PC\_next = PC + 4。
* 若Jump不为0，表示可能为j, jal, jr, jalr，若为j, jal选通JumpTarget为PC\_next；若为jr, jalr选通branchCmpA为PC\_next。
* 若Branch不为0，表示可能为b, beq, bne, blez, bgez, bltz, bgtz, blezal, bgtzal，当分支条件成立（branchCond == 1），选通JumpTarget作为PC\_next，否则按正常情形处理。
* 发生中断、异常时，PC\_next为中断和异常处理程序的入口地址。
* 当发生阻塞（stall == 1）时，需要将IF/ID寄存器保持原值（IFIDFlush = 0），将ID/EX寄存器清零（IDEXFlush = 1）。
* 当发生任何非正常的跳转时（即PC\_next != PC + 4），需要将IF/ID寄存器清零（IFIDFlush = 1），ID/EX寄存器则保持原值（IDEXFlush = 0）。

1. **Forward and Stall Unit（Forward.v）：本单元分析后指令的处理是否依赖于前一未处理完之指令，进而转发最新计算结果或命CPU阻塞。判断转发、阻塞与否的条件较繁复，无法于此一一列举，惟皆遵从以下原则：**

* 前一指令需对寄存器或Memory的值造成改变，方有转发或阻塞之必要。
* 判断前指令之目标寄存器是否与后指令之源寄存器相同，但忽略0号寄存器。
* 转发之优先序当以愈接近当前阶段所计算出来之数为优先。以branchSrcA为例，判断转发的关键代码如下：



* 遇load-use需阻塞1个周期；遇load- jr, jalr，因跳转于ID阶段提前判断，需阻塞2个周期。关键代码如下：

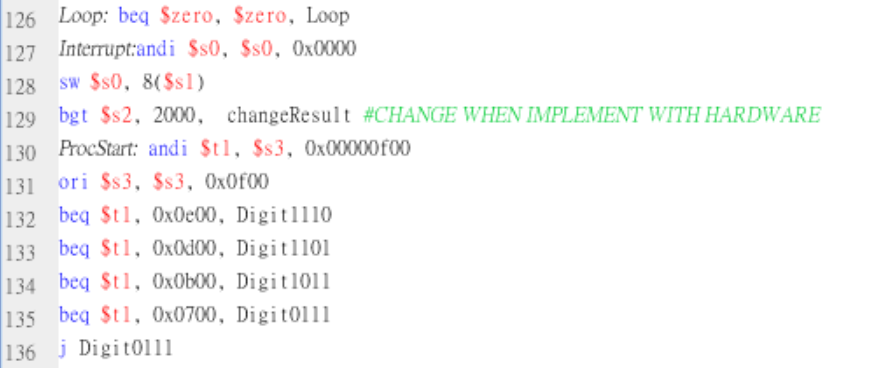


* + 1. 排序与中断汇编代码 （msort\_forhardCPU\_withInterrupt.asm）

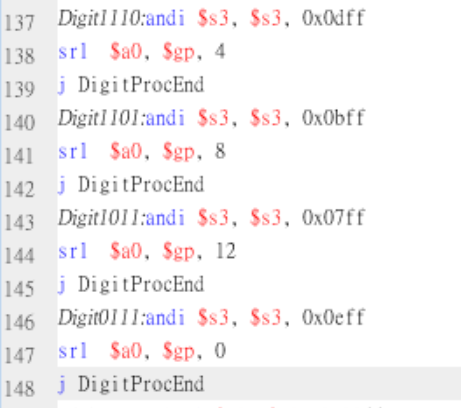
1. 排序部分：本次本人采用春季学期所写的Merge Sort算法作测试。该算法已于MARS上成功运行。本次仅需将MARS中用于分配空间的syscall函数全数改为手动分配空间的指令即可。并无太大问题。
2. 中断部分：为显示排序结果，在排序的主程序完成后，本人采用定时器中断，约每100000个时钟周期中断1次（100MHz时钟周期下约为1ms），使七段数码管做扫描显示，每作1500次扫描显示（100MHz时钟周期下约为1.5s），切换下一个欲显示的数。

* **关键代码与说明**

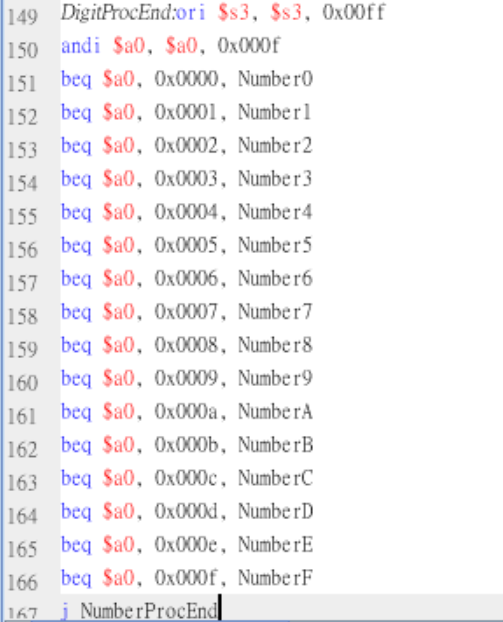
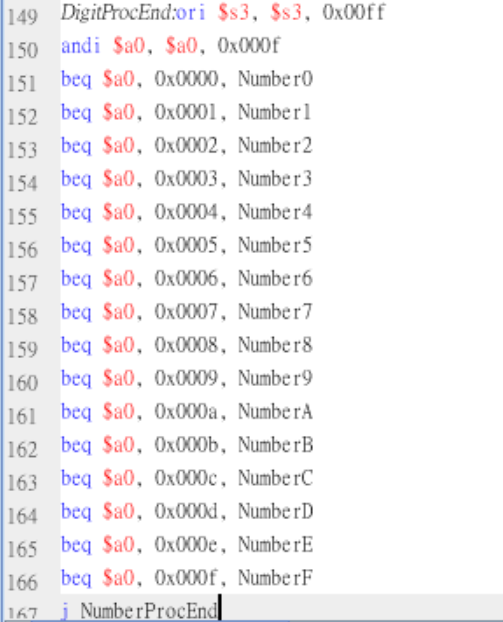
1. 第126行Loop：主程序最后一个指令（无限回圈）。
2. 第127行Interrupt：中断服务程序开始，其中$s3为七段数码管目前状态。$s2是切换下一个数与否的计数器。$s0为定时器外设的TCON变量。
3. 第130~136行：提取目前数码管的使能状态，并根据目前状态跳转到相应状态转移的处理程序内。

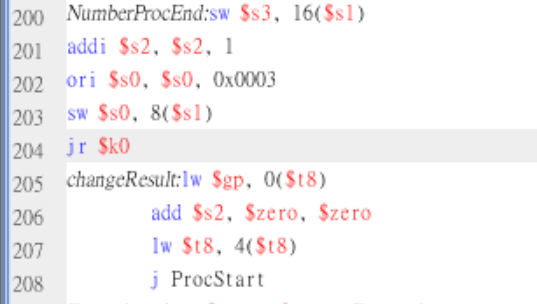


1. 第131~148行：作状态转移的处理，变更使能状态，并提取下个要扫描显示的数字。



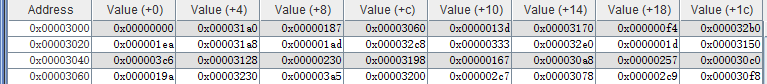
1. 第149~167行：依据欲让数码管显示的数字，跳转到相应译码的处理程序内。



1. 第168~200行：根据Basys3的管脚分配与数码管共阳极特性，完成相应数字译码。
2. 第200~204行：状态变更完成，将新状态$s3存至数码管外设对应的地址（$s0 + 16）中，将定时器状态$s0重设，跳转回中断前的主程序地址$k0。
3. 第205~208行：同一个数做一定次数的扫描显示后（即显示一定时间后，具体周期由$s2决定），归零$s2，并切换下一个欲显示的数。
4. **电路模拟情形：**
   1. 比较CPU上模拟后的排序结果与MARS上执行同样程序后的排序结果。



**(a) CPU模拟执行程序结果**



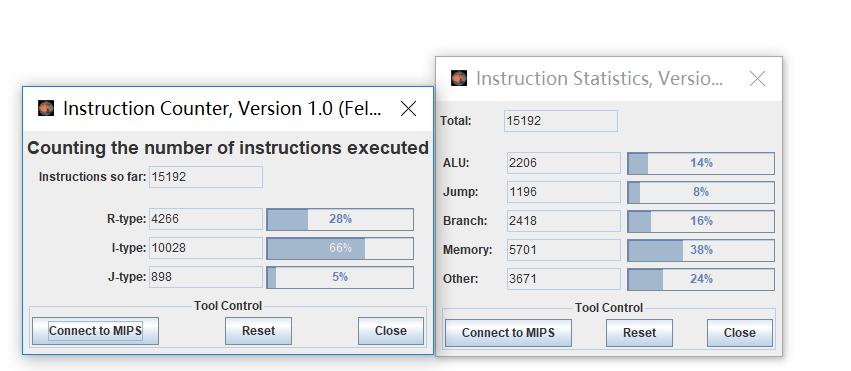
**(b)MARS执行结果**

**▲图 4-1：CPU模拟执行程序与MARS执行结果对比（节录）**

Merge Sort是以链表形式存储排序结果。本处理器的链表是自0x0000的位置开始存储，每2个字为一节点，其中第一个字为该节点所载的数字；第二个字为下一节点的地址；而MARS分配的空间则是自0x3000的位置开始存储，余与本处理器相同。可以看到排序结果中，除了每个节点所储存下一节点的地址前缀有所不同外，余皆相同。

* 1. CPU执行周期数统计：

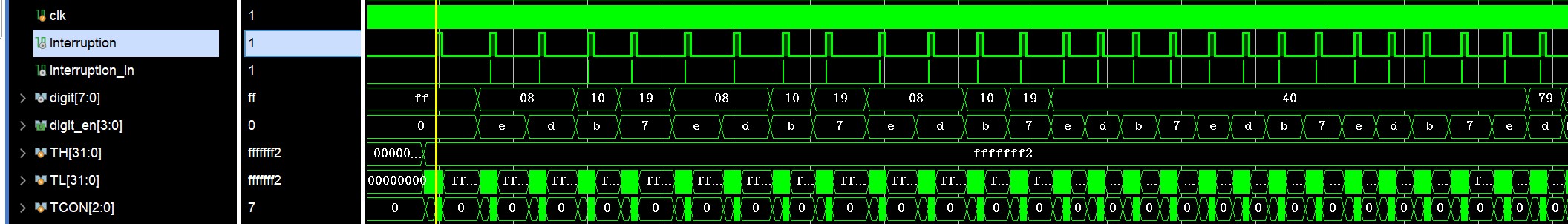
模拟结果中，CPU在周期为10ns的情形下，在80ns处完成读入Systick的初值；在188,660ns处完成读入Systick的终值，其总周期数记于$28寄存器中，值为0x49aa=18,858。MARS统计的指令总数为15,192。**本程序平均CPI约为1.24。**



**▲图 4-2：MARS统计指令数**

* 1. CPU的中断程序检查：

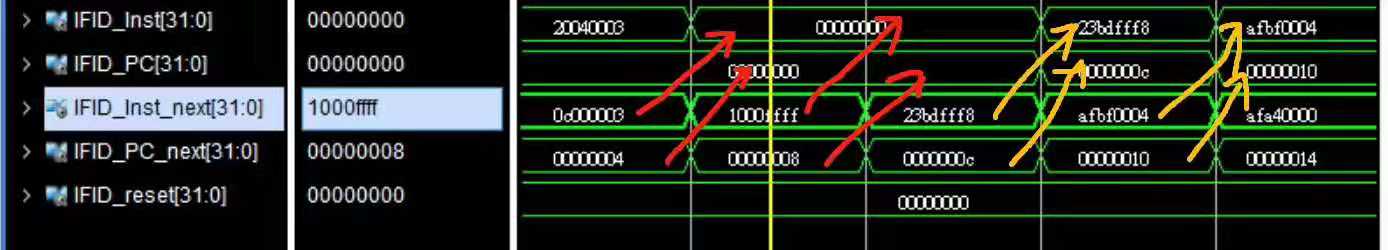
为便于仿真，我们让定时器的中断设定为约10几个周期1次，以避免过长的仿真时间，同时将切换显示数字的频率改成约10几次中断切换1次。



**▲图 4-2：CPU模拟执行程序与MARS执行结果对比**

可以看到数码管的使能信号digit\_en规律的在0xe(1110), 0xd(1101), 0xb(1011),0x7(0111)切换。且显示数字的译码信号digit亦与应该被显示的数字相符。如在本程序中，第一个被显示的数字是0x49aa（即执行总周期数），自低位到高位扫描显示的过程中，分别显示a（对应译码0x08）、a、9（对应译码0x10）、4（对应译码0x07）；下一个被显示的数是0x0000（本程序在输出执行总周期数与排序结果之间，均以0x0000间隔以资识别），无论digit\_en为何皆输出为0（对应译码0x40）。

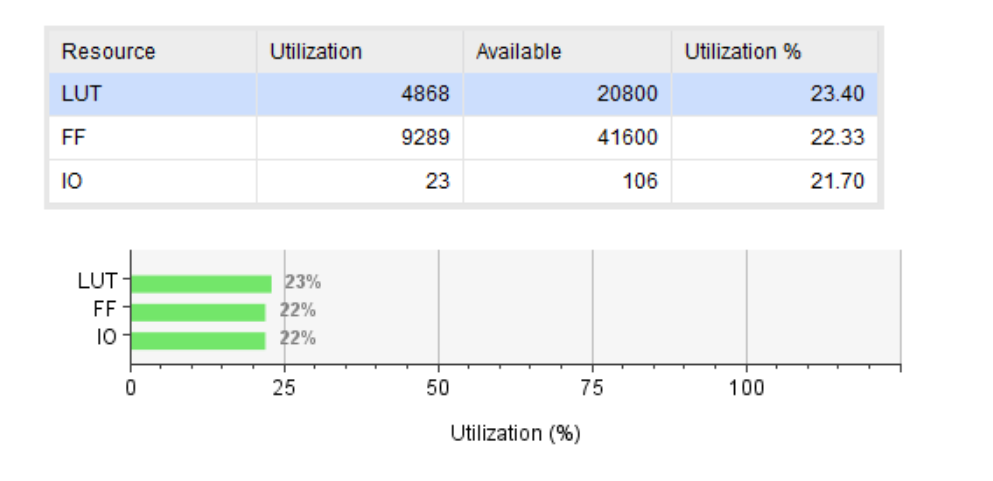
* 1. 调试过程中所遇到的**坑**包含但不限于以下几点：
* 一开始使用Vivado 2017.3版的程式来跑Simulation，发现在寄存器的代码编写正确的前提下，有些时刻寄存器竟在复位信号reset==0的情形下意外重设，我反反复复检查了好几遍，也请同学帮我检查，都看不出端倪，**最后换了Vivado 2018.3版来跑Simulation，一切正常。☺**



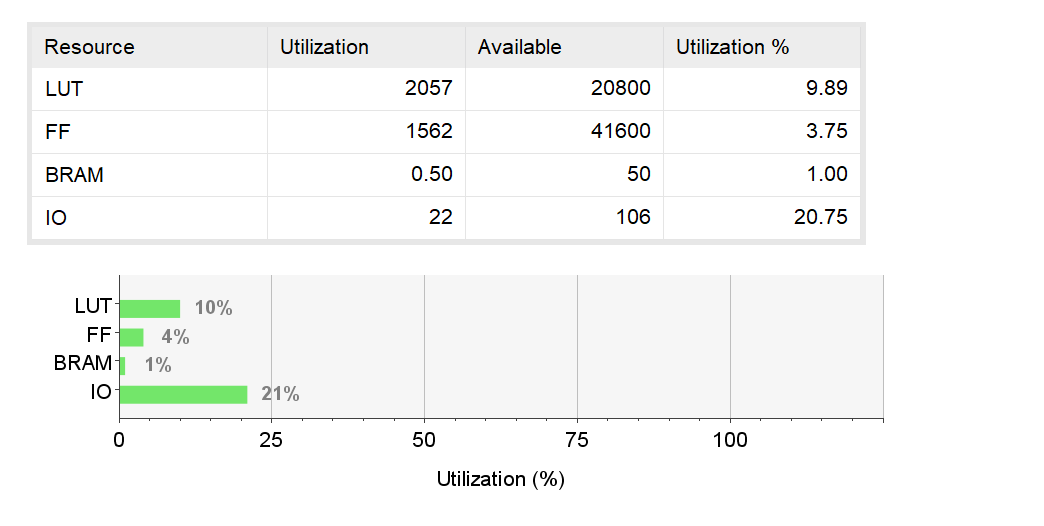
**▲图 4-3：谜之寄存器**

* wire 连接错误。如将MemWrite误连至RegWrite。
* 因中途变更设计导致部分变数（信号）在不同模组中位长不一致，定义亦不一，进而导致执行时错误。
* 忽略了Verilog预设各变数都是无符号数，因而不能以>=0、<=0直接判断正负，应比较其最高位或将该变数设为有符号数。
* 忽略了CPU阻塞时，不仅ID/EX寄存器需要清零，IF/ID寄存器需要维持原样，且其优先序应比「跳转使IF/ID清零」还要优先。否则在load-jr或load-beq, bne...之类指令时，此时同时发生stall和Jump（或Branch），若让「跳转使IF/ID清零」优先于「阻塞使IF/ID保持」，则在IF/ID清零后，阻塞后ID阶段执行的是空指令，又本设计中，跳转是在ID阶段作判断，造成该跳转指令「形同虚设」而使程序运行出现异常。
* 忽略了「进入内核态后不允许中断和异常」，未设定条件加以判断，导致进入内核态后，因无法于1个周期内将Exception和Interruption的变量重置为0，导致程序进入死循环。应将其与PC[31]作与运算。
* 发生Exception和Interruption时，因Control Signal出错，导致程序异常或中断时的地址无法存回$26，进而导致程序返回时异常。
* 族繁不及备载……说多了都是泪……

1. **性能评价（与单周期对比）：**
   * 1. 面积性能



**(a)单周期**



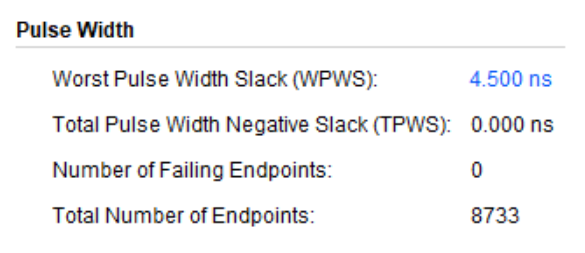
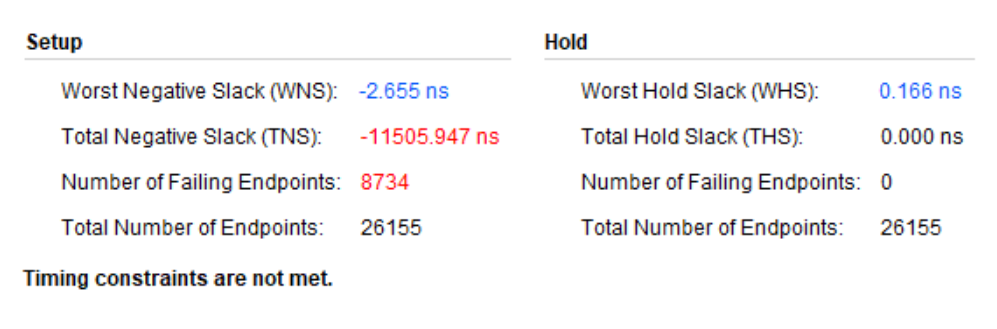
**(b)流水线**

**▲图5-1：面积资源使用情形**

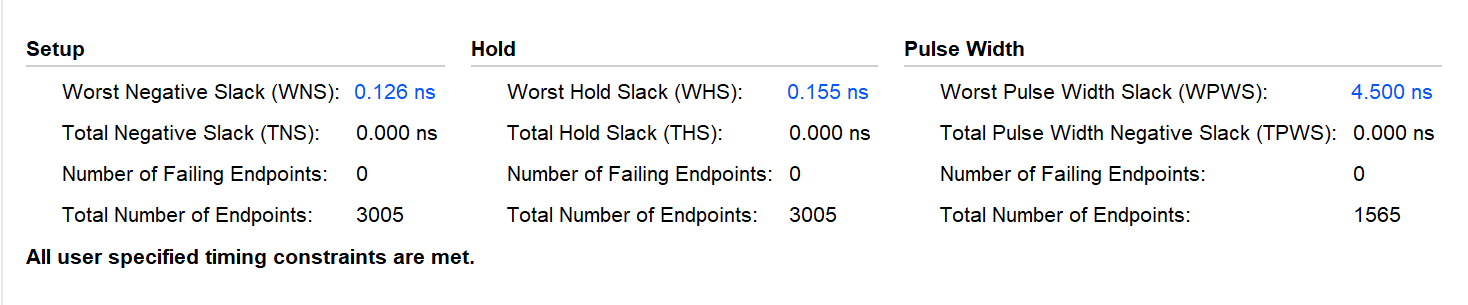
我的单周期MIPS用了4868个单位的查找表和9289个单位的触发器，占用了FPGA约1/5的资源；流水线MIPS因加入了各阶段的寄存器，理应占用更多资源，但在使用BRAM的情形下，大幅减少了LUT和FF的耗用。

* + 1. 时序性能：

在电路由100MHz时钟驱动的情形下（周期=10ns），我的单周期MIPS处理器Setup时间的裕量为-2.655ns，故电路工作的最高主频为。我的流水线MIPS处理器Setup时间的裕量为0.126ns，故电路工作的最高主频为。主频提升了28%。可见用流水线确能有效减短时钟周期的长度。



**(a)单周期**



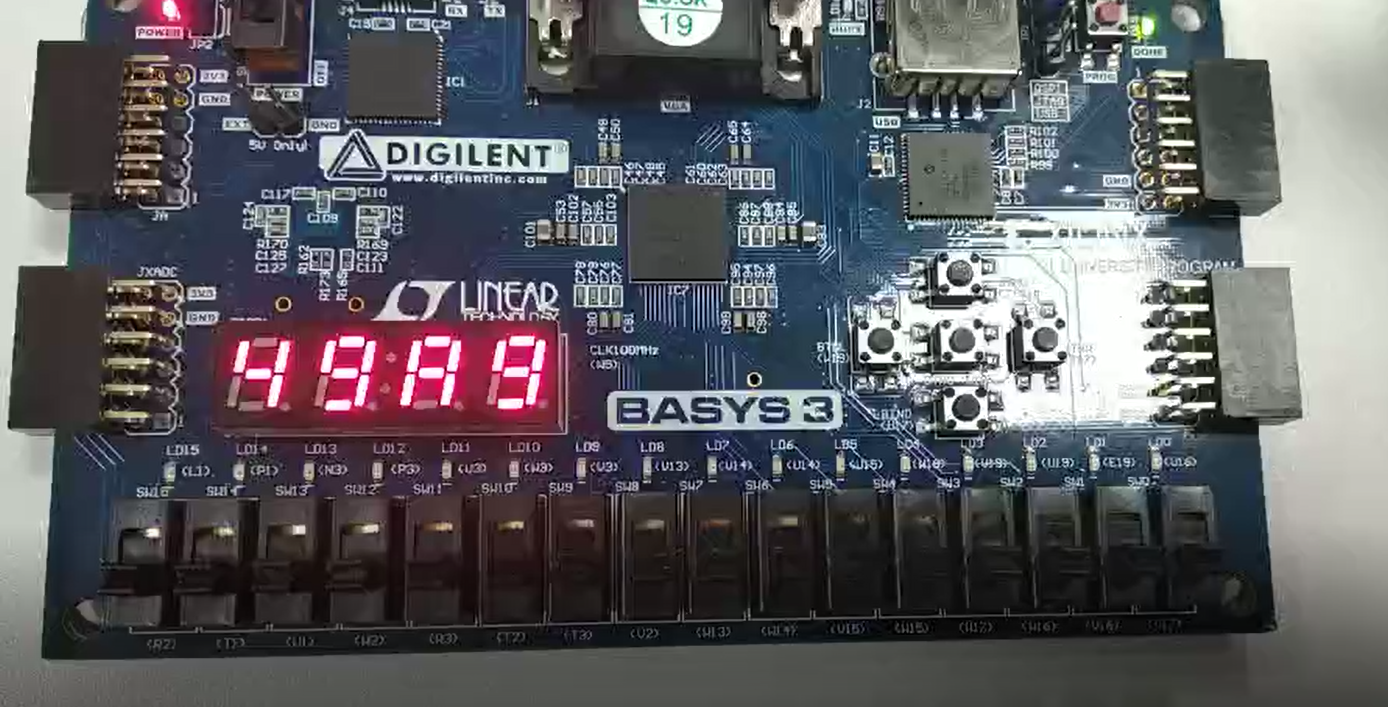
**(b)流水线**

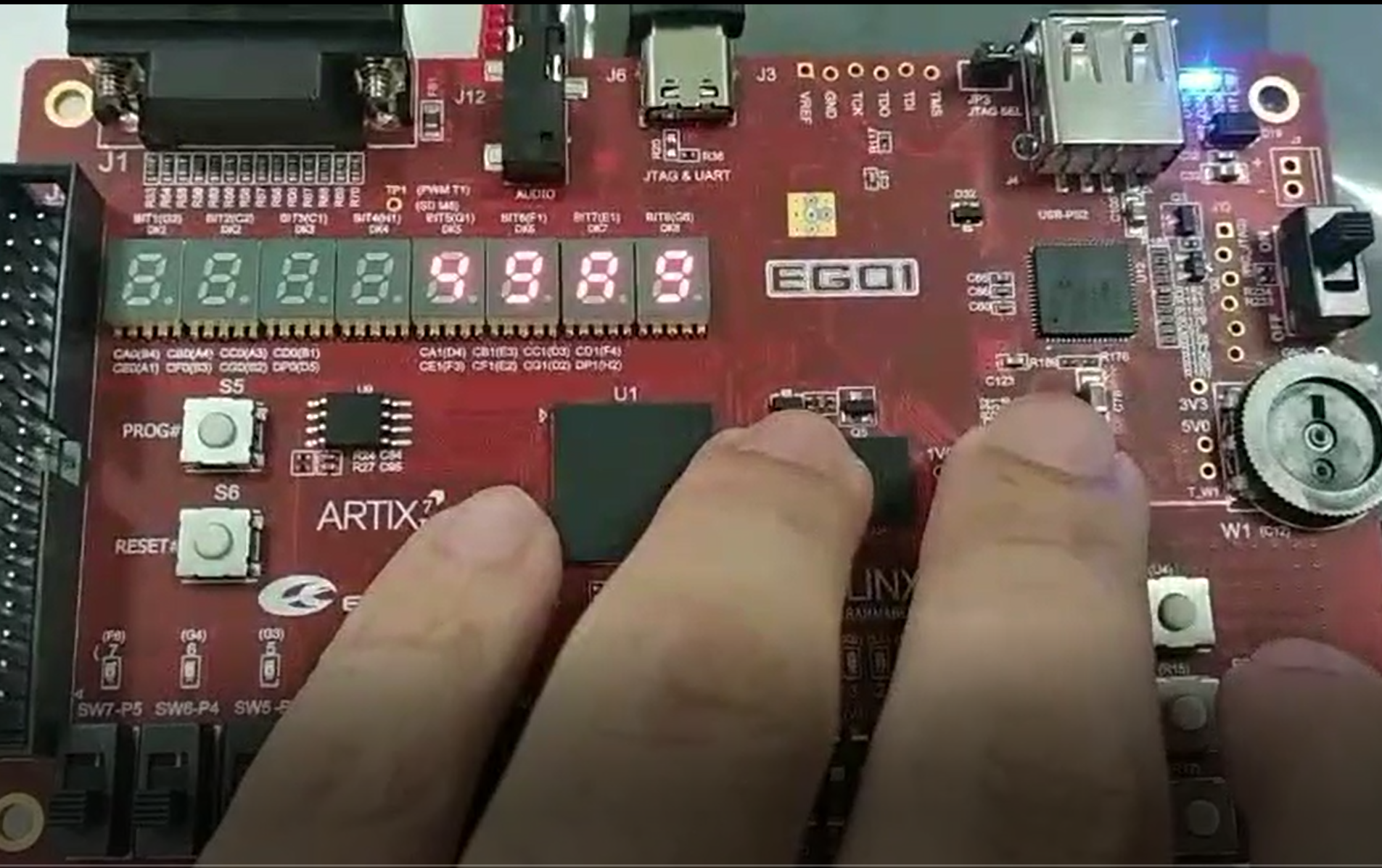
**▲图5-2：时序性能情形**

单周期之关键路径应为执行load word指令造成，而流水线处理器的关键路径如下图所示。由其所经之相关器件路径，推测应是与branch指令有关，需要提取Register中的内容（并完成转发）、完成条件判断，决定对IF/ID寄存器清零，并且改变PC的值。

1. **硬件调试情形：**

本次流水线处理器自Simulation成功起，相较于其他同学，在Synthesis、Inplementation、Generate Bitstream与硬件调试上，可说是披荆斩棘、势如破竹，丝毫没有遇到任何问题。我认为这是因我的代码写作严谨，且时时查看Vivado给出的Warning并据以修正所致。硬件调试结果与预期结果相符。详细情形可参见附录影片档。

 **▲图6-1：硬件调试情形（Basys版）**



**▲图6-2：硬件调试情形（EGO1版）**

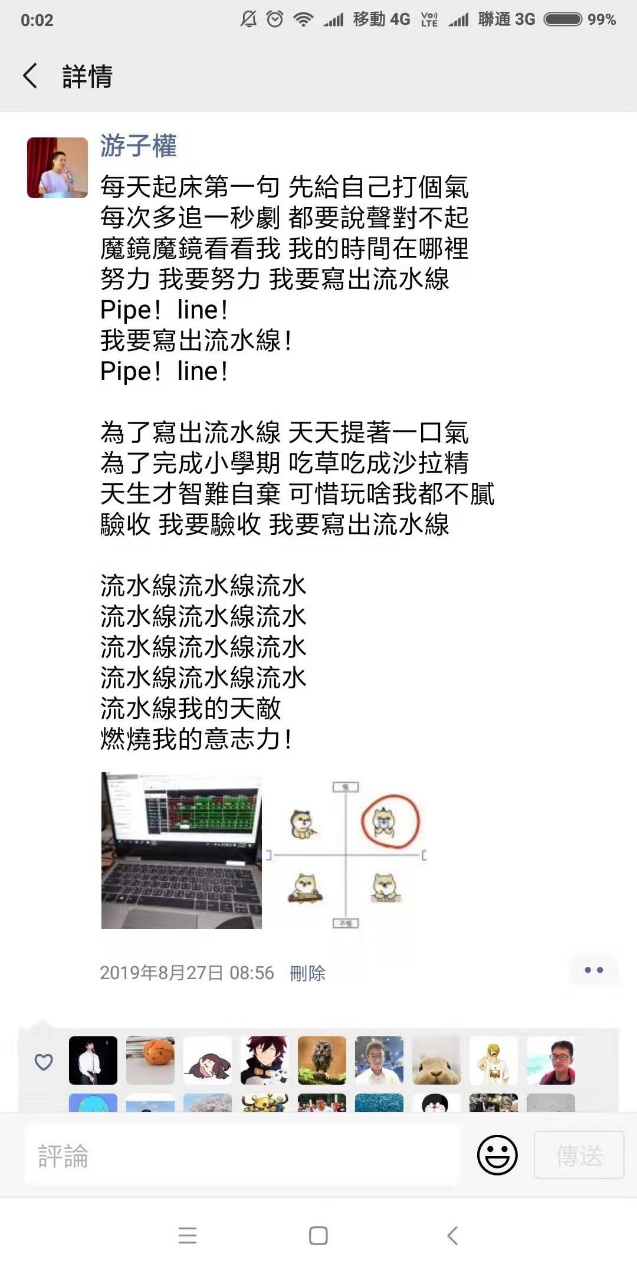
春季学期的单周期CPU完成一个指令最短需时12.655ns×1cycle/inst = 12.655ns/inst

本学期之流水线CPU完成一个指令最短需时9.874ns×1.24cycle/inst = 12.244ns/inst

**流水线比单周期快了4%。虽然不显着，但也是值得为自己喝采的成就了！**

1. **思想体会**

不啰嗦，直接上图。

****

**▲图7：拜拜，FPGA！拿走拿走别客气！**