32位MIPS处理器实验报告

汤宸 2016011018

成大立 2016011029

李云飞 2016011017

2018年8月1日

目录

[1 实验目的 2](#_Toc520897757)

[2 设计方案 2](#_Toc520897758)

[2.1 成员分工 2](#_Toc520897759)

[2.2 整体架构 2](#_Toc520897760)

[2.2.1 ALU 2](#_Toc520897761)

[2.2.2 单周期处理器 3](#_Toc520897762)

[2.2.3 流水线处理器 4](#_Toc520897763)

[2.3 模块设计 4](#_Toc520897764)

[2.3.1 ALU 4](#_Toc520897765)

[2.3.2 汇编器 5](#_Toc520897766)

[2.3.3 串口 5](#_Toc520897767)

[2.3.4 定时器外设 5](#_Toc520897768)

[2.3.5 单周期 6](#_Toc520897769)

[2.3.6 流水线 9](#_Toc520897770)

[3 仿真结果 13](#_Toc520897771)

[3.1 ALU 13](#_Toc520897772)

[3.2 单周期处理器 15](#_Toc520897773)

[3.3 流水线处理器 18](#_Toc520897774)

[4 调试情况 21](#_Toc520897775)

[4.1 单周期处理器 21](#_Toc520897776)

[4.1.1 汇编器 21](#_Toc520897777)

[4.1.2 PC最高位清零问题 21](#_Toc520897778)

[4.1.3 数据存储器读出冲突 22](#_Toc520897779)

[4.1.4 串口时序 22](#_Toc520897780)

[4.1.5 MemtoReg数据通路的修改 22](#_Toc520897781)

[4.1.6 数码管按字寻址 23](#_Toc520897782)

[4.1.7 支持接收多组操作数 23](#_Toc520897783)

[4.1.8 定时器周期 23](#_Toc520897784)

[4.2 流水线处理器 23](#_Toc520897785)

[4.2.1 寄存器堆 23](#_Toc520897786)

[4.2.2 转发单元 24](#_Toc520897787)

[4.2.3 冒险单元 24](#_Toc520897788)

[5 综合情况 24](#_Toc520897789)

[5.1 单周期处理器 24](#_Toc520897790)

[5.1.1 时序性能 24](#_Toc520897791)

[5.1.2 资源占用情况 25](#_Toc520897792)

[5.1.3 其他信息 26](#_Toc520897793)

[5.2 流水线处理器 27](#_Toc520897794)

[5.2.1 时序性能 27](#_Toc520897795)

[5.2.2 资源占用情况 28](#_Toc520897796)

[5.2.3 其他信息 29](#_Toc520897797)

[6 感想体会 30](#_Toc520897798)

[7 文件清单 32](#_Toc520897799)

# 1 实验目的

熟悉现代处理器的基本工作原理，掌握单周期和流水线处理器的设计方法。

（1）设计一个32位的ALU，支持基本的算术运算、逻辑运算和关系运算。

（2）设计一个单周期MIPS处理器，支持某些MIPS指令集的核心指令；支持中断处理和异常处理，通过定时器外设提供中断信号，并实现MIPS的UART外设；将求最大公约数的程序在处理器上执行，操作数由计算机通过UART发送给处理器，要求数码管扫描显示操作数，LED显示运算结果，并将运算结果通过UART发送回计算机。

（3）设计一个五级流水线的MIPS处理器，要求解决流水线中的数据冲突和控制冲突；将最大公约数程序在流水线CPU中运行，I/O要求同单周期处理器。

# 2 设计方案

## 2.1 成员分工

在初期编写过程中，ALU的编写、流水线数据通路的搭建由汤宸进行，控制信号的设计、定时器外设和中断处理程序、冒险检测和转发单元由成大立进行，编译器、单周期数据通路的搭建、串口的设计由李云飞进行。

在仿真调试过程中，单周期CPU的调试主要由李云飞完成，流水线CPU的调试主要由成大立和汤宸共同完成；基于Vivado综合和FPGA实现的调试和优化过程则由三人共同完成。

总体而言，虽然在任务初期，三名队员有相对明确的分工，但随着开发过程的进行，各个部分的进展情况难以预测，我们三名队员之间不再有明确清晰的分工。但三人之间能够通力合作，高效地解决开发过程中遇到的各种问题，没有甩锅或划水等现象出现。

## 2.2 整体架构

### 2.2.1 ALU

ALU的设计主要为了实现算术、逻辑、关系、位与移位运算功能。算术功能包括加法功能与减法功能；位运算包括与、或、异或、或非等功能；移位运算包括左移、逻辑右移、算数右移；关系运算包括相等、不等、小于、小于等于零、小于零、大于零的比较功能。

表1 ALU各个端口的含义说明

|  |  |  |
| --- | --- | --- |
| 名称 | 类型 | 描述 |
| A[31:0] | 输入 | 操作数1 |
| B[31:0] | 输入 | 操作数2 |
| S[31:0] | 输出 | 结果输出 |
| ALUFun[5:0] | 输入 | ALU功能 |
| Sign | 输入 | 运算符号，1：有符号；0：无符号 |

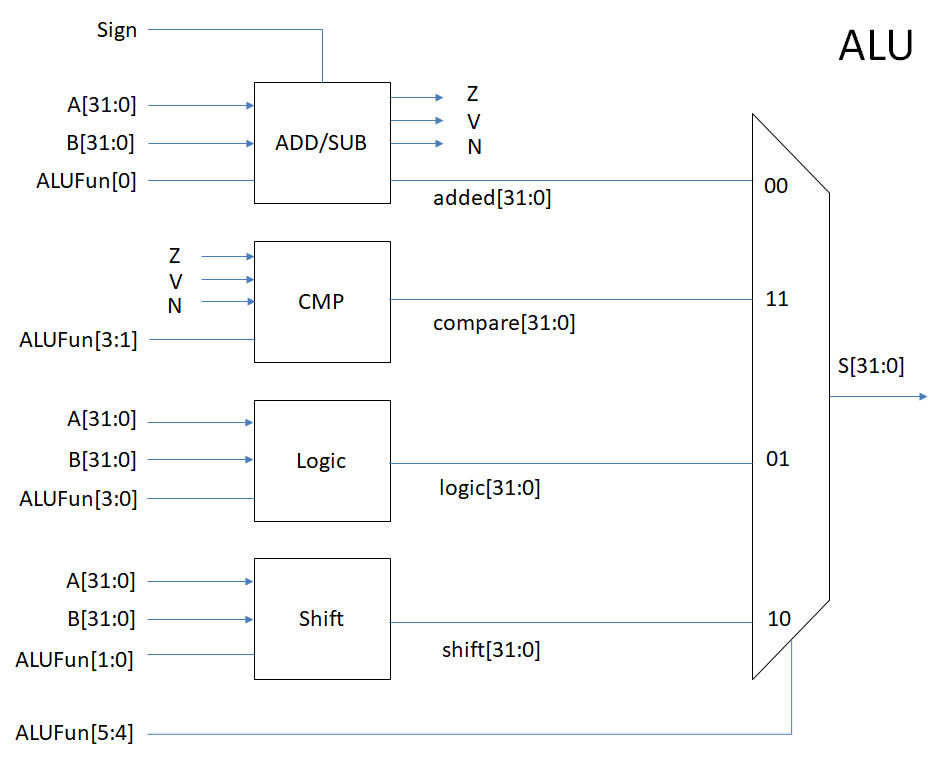


图1 ALU功能框图

### 2.2.2 单周期处理器

单周期数据通路为指令与数据存储器分开的哈佛体系架构。数据通路上的硬件有程序计数器处理单元PC、控制单元Control、指令存储器InstructinMemory、寄存器堆RegisterFile、立即数扩展单元、算数逻辑单元ALU、数据存储器DataMemory以及若干多路选择器、加法器等。此外，为了支持中断处理，在CPU之外添加了定时器外设Peripheral和串口IO设备UART。处理器时钟通过分频单元cpu\_clk生成。

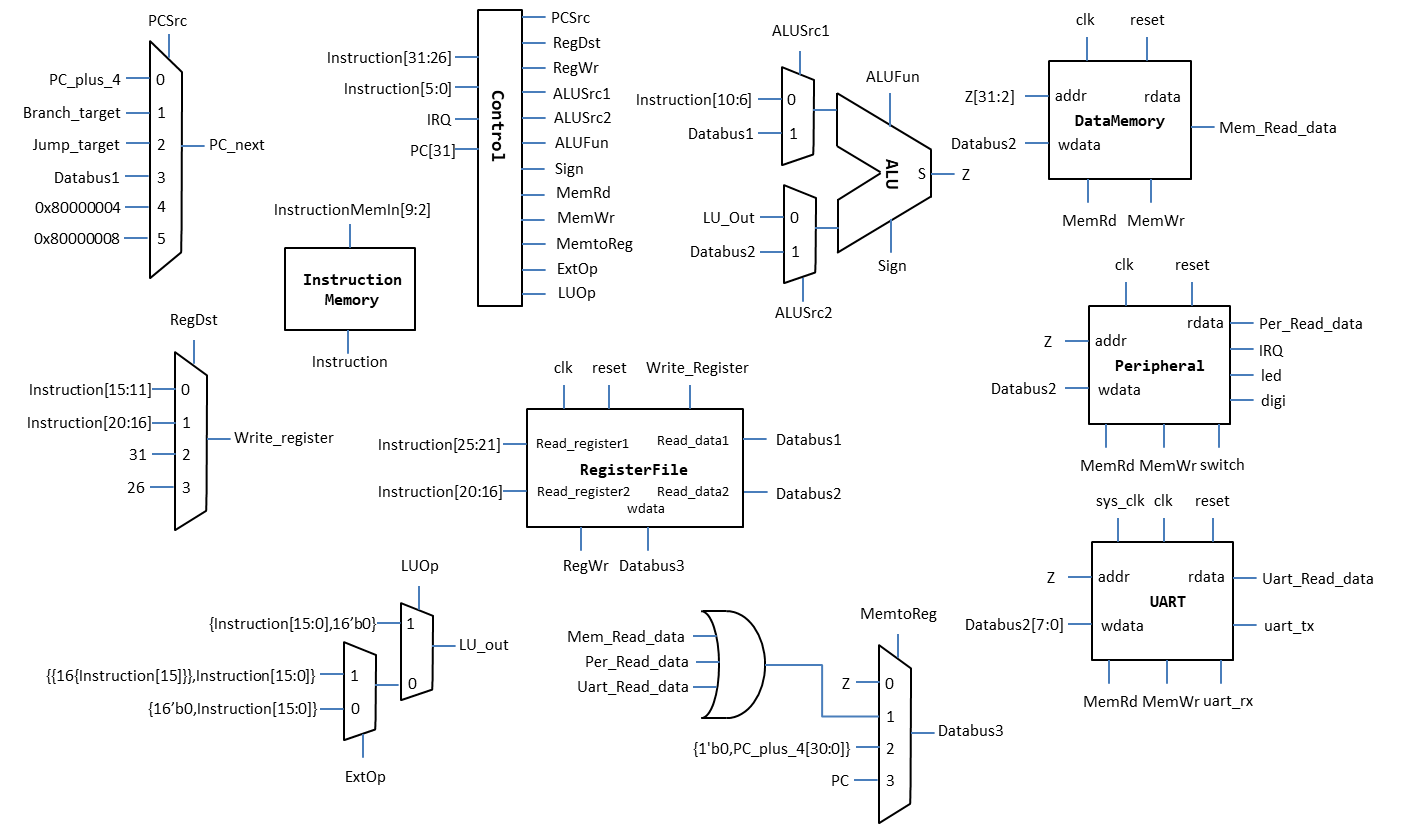


图2 单周期CPU整体框图

### 2.2.3 流水线处理器

流水线CPU的数据通路是在单周期CPU的基础上改装而成的。为了实现流水功能，需要加入IF/ID、ID/EX、EX/MEM、MEM/WB四个段间寄存器，用于保存指令所需的控制信号和数据。如图3所示（可能不够清晰，请参考附件中的“流水线框图.png”），段间寄存器将整个CPU硬件切分成了IF、ID、EX、MEM、WB五个阶段。

除了增加了段间寄存器以外，PC寄存器、RF寄存器堆、ALU和数据存储器Mem的前后都加入了更多的MUX，用来实现硬件资源的复用，应对CPU中同时运行多条指令所带来的更加复杂的控制操作。另外，分支跳转指令的数据通路不再经过ALU执行，而是直接在访问寄存器阶段判断$rs和$rt是否相等。

在流水线CPU中尤其重要的是冒险检测单元和转发单元，它们不属于任何一个流水阶段，它们负责实现不同阶段间的调度。当出现数据冒险和控制冒险时，冒险检测单元指挥整个CPU阻塞若干个周期（如果有必要的话），转发单元为各条指令及时提供必要的数据。这两个模块会在后文中详细介绍。

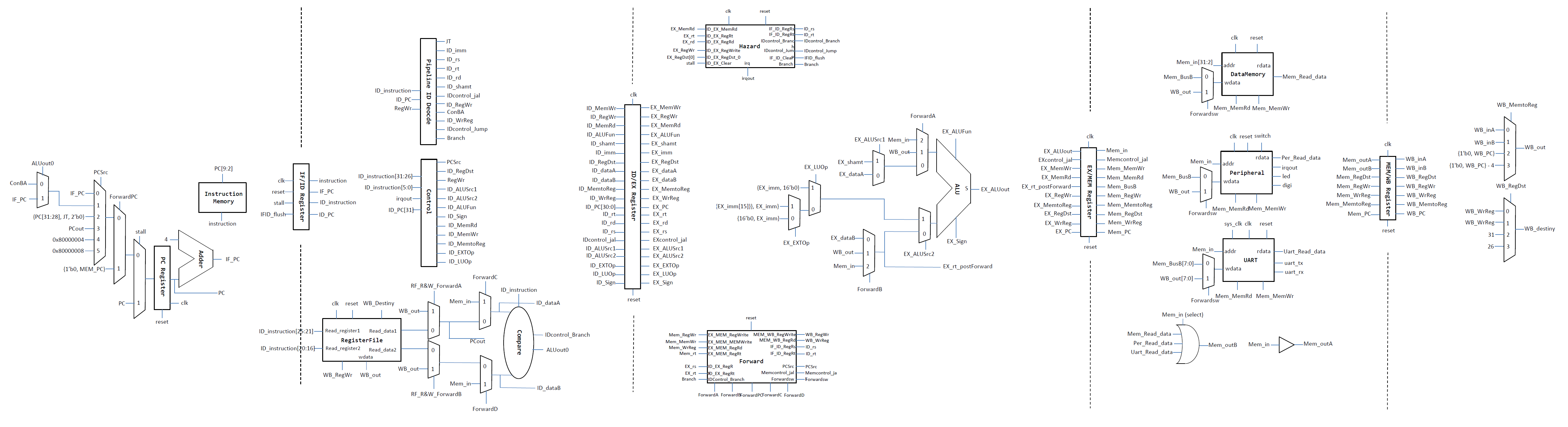


图3 流水线CPU整体框图

## 2.3 模块设计

### 2.3.1 ALU

ALU的作用是，依据ALUFun控制信号对输入的两个信号选择做算术、逻辑、关系、位与移位运算，并将结果输出。因此ALU部分包含顶层模块、加法模块、比较模块逻辑模块与移位模块共五个模块。顶层模块的作用即是调用各个模块，然后由ALUFun的高两位决定输出何数，在这里不再赘述。

#### 2.3.1.1 加法模块

加法模块实现add与sub两个功能。由ALUFun[0]控制进行加法还是减法运算。Sign决定进行有符号还是无符号运算。若为减法运算，则要先对操作数2取补码再进行运算。对两数运算结果added进行输出，同时输出对Z、V、N进行运算和输出。Z为结果为零的标识，当added为0时，输出为1；V为结果溢出标识，当为有符号运算时，若输入变量1及判断是否要进行补码处理过后的输入变量2符号一致，但added符号与它们相反，则V赋值为1；N为结果为负的标识位，当added为负数时，输出为1。

#### 2.3.1.2 比较模块

比较模块实现EQ、NEQ、LT、LEZ、LTZ、GTZ功能。由ALUFun[3:1]控制实现功能选择，由加法模块的Z、V、N来判定结果。结果由compare进行输出。

#### 2.3.1.3 逻辑模块

逻辑模块实现AND、OR、XOR、NOR、“A”功能。由ALUFun[3:0]控制功能选择输出，直接对输入变量1及输入变量2进行操作，对结果logic进行输出。

#### 2.3.1.4 移位模块

移位模块实现SLL、SRL、SRA功能，由ALUFun[1:0]控制输出选择，看成1位、2位、4位、8位、16位移位器级联组成，由输入变量1控制这几个移位器。

### 2.3.2 汇编器

汇编器的作用是将MIPS汇编代码翻译成机器码，并且写入CPU的InstructionMemory里面。本实验中的汇编器支持翻译的汇编代码文件可以包含正常指令，标签，注释，空行等，并且可以指定写入指令存储器时的首地址。为了方便调试，我们的汇编器将翻译好的机器码直接写成verilog语句，并且在每条语句之后用注释打出可读性更好的汇编指令。

### 2.3.3 串口

UART协议串口主要包括串口时钟生成、接收机和发射机三个部分。我们的串口工作在9600波特率，而FPGA的系统时钟为100MHz，因此需要一个时钟生成模块来产生合适的分频驱动串口。串口时钟频率为9600\*16 = 153.6kHz，需要对系统时钟作651分频。分频通过计数器实现，每数到651个上升沿，输出一个高电平，否则输出低电平。

我们对串口状态uart\_con的设置方式与指导书略有不同。

为了与中断处理逻辑相配合，我们对接收机和发射机的输出状态作了以下约定：

接收机的输出状态表示有没有收到新数据。当接收机收到一个数据的结束位后，输出一个高电平，然后在下一个串口时钟置0，也就是每次收完新数据，输出一个高电平脉冲。

发射机的输出表示当前发射机是否处于发送状态。在发射机发送起始位到结束位之间，发送状态置高电平，否则置低电平。

### 2.3.4 定时器外设

数码管的点亮和串口的轮询需要CPU来控制，但这些I/O设备的速度远远低于CPU的速度，即必须对系统时钟进行计数分频，才能用来控制I/O设备。为了不影响CPU执行正常程序，这一计数的任务应该交给一个专门的硬件设备（即定时器）来执行。定时器本质上就是计数器，计数器从某一用户指定的起始状态开始，每个系统时钟周期加1，增加到0xFFFFFFFF后溢出，产生一个中断信号IRQ送到CPU的控制单元，提醒CPU进行I/O工作。

CPU的控制单元接收到中断信号后，将正在执行的程序地址保存，跳转到中断处理程序地址处，并压栈保存现场。CPU首先通过sw指令，将定时器外设的中断状态（IRQ）清零，等待下一次中断来临；其次通过lw指令检查数码管的点亮情况，将下一次即将点亮的数码管通过sw指令告知外设（BCD译码是通过数据存储器查表的方法进行的，即数据存储器的0号地址存储了数字0的数码管译码结果，其余数字以此类推）；最后CPU进行串口通信，如果UART串口接收到了新数据，CPU将它搬运到寄存器堆中，如果主程序计算出了最大公约数的值，CPU将它写入串口的发送地址。完成以上操作后，CPU恢复现场，跳转回主程序继续执行，定时器继续计数，等待下一次中断到来。

### 2.3.5 单周期

#### 2.3.5.1 CPU时钟生成

Ego1开发板的系统时钟为100MHz，单周期处理器的数据通路比较长，可能不能在一个系统时钟周期内跑完。因此需要从系统时钟生成一个更慢的时钟驱动CPU。这里我们通过简单的计数方法对系统时钟作分频，可以生成占空比50%，频率为系统时钟的时钟，n取决于计数器的位宽。用系统时钟上升沿驱动计数器计数，每当计数到溢出，就使输出的clk取反，并且把计数器清零。

#### 2.3.5.2 控制单元

控制单元根据从指令存储器取出的instruction以及CPU当前所处的状态（用户态，内核态，有无中断）生成数据通路中其他模块需要的控制信号。控制信号包括PC更新的源PCSrc，写回寄存器的选择RegDst，寄存器堆写使能RegWr，ALU两个操作数的源ALUSrc1、ALUSrc2，ALU执行的运算类型选择ALUFun，ALU运算是否有符号选择Sign，数据存储器（包含DataMemory，Peripheral，UART）读使能MemRd，数据存储器写使能MemWr，写回寄存器数据源MemtoReg，立即数扩展使能ExtOp，高位立即数使能LUOp。

最高优先级的判断是当前CPU处于用户态还是内核态。用户态指的是CPU执行用户代码，允许中断；而内核态是不接收中断的，CPU在执行中断处理程序或者异常处理程序。CPU处于用户态或内核态是由PC的最高位决定的，PC[31]为1代表处于内核态。控制单元Control模块会监测PC最高位，当PC[31]为1时，中断信号不影响控制单元的输出；只有在用户态中，控制单元的输出会受到中断信号IRQ的影响。

我们在用户态的前提下判断是否发生中断。如果中断，那么控制单元应该产生能够保存现场的信号。写入寄存器目标设为26号（$k0）寄存器，即将写入的值选择当前PC值，PCSrc选择异常代码入口0x80000004。

如果在用户态且没有发生中断，就应该按照指令的opcode和funct识别出指令类型，再相应地给出数据通路各信号的选通情况。我们设计的真值表如下。

表2 控制单元真值表

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Opcode | Funct | PCSrc | RegDst | RegWr | ALUSrc1 | ALUSrc2 | ALUFun | Sign | MemRd | MemWr | MemtoReg | ExtOp | LUOp |
| add | 000000 | 100000 | 000 | 00 | 1 | 0 | 0 | 000000 | 1 | 0 | 0 | 00 | x | x |
| addu | 000000 | 100001 | 000 | 00 | 1 | 0 | 0 | 000000 | 0 | 0 | 0 | 00 | x | x |
| sub | 000000 | 100010 | 000 | 00 | 1 | 0 | 0 | 000001 | 1 | 0 | 0 | 00 | x | x |
| subu | 000000 | 100011 | 000 | 00 | 1 | 0 | 0 | 000001 | 0 | 0 | 0 | 00 | x | x |
| and | 000000 | 100100 | 000 | 00 | 1 | 0 | 0 | 011000 | 0 | 0 | 0 | 00 | x | x |
| or | 000000 | 100101 | 000 | 00 | 1 | 0 | 0 | 011110 | 0 | 0 | 0 | 00 | x | x |
| xor | 000000 | 100110 | 000 | 00 | 1 | 0 | 0 | 010110 | 0 | 0 | 0 | 00 | x | x |
| nor | 000000 | 100111 | 000 | 00 | 1 | 0 | 0 | 010001 | 0 | 0 | 0 | 00 | x | x |
| sll | 000000 | 000000 | 000 | 00 | 1 | 1 | 0 | 100000 | x | 0 | 0 | 00 | x | x |
| srl | 000000 | 000010 | 000 | 00 | 1 | 1 | 0 | 100001 | x | 0 | 0 | 00 | x | x |
| sra | 000000 | 000011 | 000 | 00 | 1 | 1 | 0 | 100011 | x | 0 | 0 | 00 | x | x |
| slt | 000000 | 101010 | 000 | 00 | 1 | 0 | 0 | 110101 | 1 | 0 | 0 | 00 | x | x |
| jr | 000000 | 001000 | 011 | x | 0 | x | x | x | x | 0 | 0 | x | x | x |
| jalr | 000000 | 001001 | 011 | 00 | 1 | x | x | x | x | 0 | 0 | 10 | x | x |
| lw | 100011 |  | 000 | 01 | 1 | 0 | 1 | 000000 | 0 | 1 | 0 | 01 | 1 | 0 |
| sw | 101011 |  | 000 | x | 0 | 0 | 1 | 000000 | 0 | 0 | 1 | x | 1 | 0 |
| lui | 001111 |  | 000 | 01 | 1 | 0 | 1 | 000000 | 0 | 0 | 0 | 00 | x | 1 |
| addi | 001000 |  | 000 | 01 | 1 | 0 | 1 | 000000 | 1 | 0 | 0 | 00 | 1 | 0 |
| addiu | 001001 |  | 000 | 01 | 1 | 0 | 1 | 000000 | 0 | 0 | 1 | 00 | 1 | 0 |
| andi | 001100 |  | 000 | 01 | 1 | 0 | 1 | 011000 | 0 | 0 | 0 | 00 | 0 | 0 |
| slti | 001010 |  | 000 | 01 | 1 | 0 | 1 | 110101 | 1 | 0 | 0 | 00 | 1 | 0 |
| sltiu | 001011 |  | 000 | 01 | 0 | 0 | 1 | 110101 | 0 | 0 | 0 | 00 | 1 | 0 |
| beq | 000100 |  | 001 | x | 0 | 0 | 0 | 110011 | x | 0 | 0 | x | 1 | 0 |
| bne | 000101 |  | 001 | x | 0 | 0 | 0 | 110001 | x | 0 | 0 | x | 1 | 0 |
| blez | 000110 |  | 001 | x | 0 | 0 | 0 | 111101 | x | 0 | 0 | x | 1 | 0 |
| bgtz | 000111 |  | 001 | x | 0 | 0 | 0 | 111111 | x | 0 | 0 | x | 1 | 0 |
| bltz | 000001 |  | 001 | x | 0 | 0 | 0 | 111011 | x | 0 | 0 | x | 1 | 0 |
| j | 000010 |  | 010 | x | 0 | x | x | x | x | 0 | 0 | x | x | x |
| jal | 000011 |  | 010 | 10 | 1 | x | x | x | x | 0 | 0 | 10 | x | x |
| 异常 |  |  | 101 | 11 | 1 | x | x | x | x | 0 | 0 | 10 | x | x |
| 中断 |  |  | 100 | 11 | 1 | x | x | x | x | 0 | 0 | 11 | x | x |

#### 2.3.5.3 PC寄存器

PC单元的作用是计算下一条指令的地址。其中PC的最高位还用来作为CPU用户态/内核态的标志位，最高位为0表示用户态，最高位为1表示内核态。用户态和内核态应该由中断决定，而不是PC。因此，需要给PC的跳转做一些规定，以满足标志位的设定。首先，只有中断和异常才能给PC最高位置1。只有jr和jalr指令才能使PC最高位置0。其他情况下PC的跳转都应该保持PC最高位不变。

PC单元的更新结果将被用来送入指令存储器取址，而且PC的最高位不参与取址。

为了满足我们的CPU功能需求，PC更新的取值有6种可能：PC\_plus\_4，分支目标Branch\_target，跳转目标Jump\_target，寄存器中的值Databus1，中断入口0x80000004，异常入口0x80000008。

PC\_plus\_4的计算需要保证PC最高位不变，因此取为原PC最高位和通常的(PC+4)结果低31位拼接。在没有发生分支、跳转的情况下，PC应该更新为PC\_plus\_4。Branch\_target计算方法如图所示，其中Z是ALU的输出。如果Z的最低位是1，说明有可能要发生条件分支，选通PC\_plus\_4+偏移量；如果Z的低位是0，说明不可能发生条件分支，应该选通正常的PC\_plus\_4。Jump\_target的值直接取为PC+4的高四位和指令中的立即数偏移拼接的结果。Databus1是为jr和jalr指令准备的，它们会将寄存器中的值赋给PC的下一次取值。



图4 Branch\_target计算方法

#### 2.3.5.4 RF寄存器堆

寄存器堆支持两个读取端和一个写入端。读取通过组合逻辑实现，可以随时读出数据；写入通过时序逻辑实现，一条指令如果要写回寄存器，会在下一个周期的时钟上升沿实现寄存器中数据的更新。写回寄存器的目标通过多路选择器选择，除了rt、rd，还有31号和26号寄存器。写回31号寄存器用来支持jal指令，写回26号寄存器用来支持异常和中断保存现场的需要。



图5 寄存器堆设计

#### 2.3.5.5 数据存储器与写回部分

我们的单周期设计中，数据存储器包含三个部分：DataMemory也即通常意义的内存，Peripheral专为外设开辟的存储段，UART专为串口开辟的存储段。这三个部分的地址都分布在4G存储空间中，其中DataMemory地址从0x00000000~0x000003ff，实际使用的地址为0x00000000~0x0000007f。Peripheral地址从0x40000000~0x40000014，UART地址从0x40000018~0x40000023。这三部分的写操作都是通过指令译码得到的控制信号使能的，由于一条指令只能指定向一个地址里写，因此三部分存储器不会产生冲突，本质上和只有DataMemory的时候并没有区别。在读存储器时，三个存储器同时被读使能，但是对于超过地址范围的存储器，rdata取0，因此只需要将三个存储器的rdata端输出或起来，就能合并出一个读到的数。



图6 写回寄存器多路选择

在写回寄存器的多路选择器中，我们设置了四种选择情况：把ALU计算结果写回；将存储器中读取的结果写回；将最高位置0的PC\_plus\_4写回；将当前PC写回。

在原先的设计中，后两路选择合并为一个PC\_plus\_4，但是该设计不能实现jr和jalr指令把PC最高位清零。而在我们的新设计中，最高位置0的PC\_plus\_4对应的是jalr,jal指令，使得以后执行jr和jalr指令的时候取出寄存器中的值最高位是0，从而使PC[31]清零。

第四路选择PC是为中断准备的。由于中断到来的时候，控制单元立刻将控制信号置为保存中断现场需要的信号，在中断到来时正在执行的指令并没有执行完。因此，在中断现场地址$k0寄存器中要存的应该是这条没有正确执行完的指令的地址，以便中断处理程序结束后跳回正常处理程序时能够把这条指令重新执行一次。

### 2.3.6 流水线

#### 2.3.6.1 数据通路

流水线CPU使用100MHz系统时钟，不需要分频；控制单元产生的控制信号的约定、PC最高位的约定、数据存储器地址的约定都和单周期相同。这里不再赘述，只着重强调某些和单周期的差异。

其一，PC的更新操作更加复杂。如果冒险检测单元命令CPU阻塞一个周期，那么PC值不变，并且stall命令的优先级是除了reset以外最高的。当不发生阻塞时，如果发现了jal-and-jr冒险，PC的值需要从MEM/WB寄存器中转发（下图中的ForwardPC）；在其他情况下，由PCSrc决定PC的来源。PCSrc从0到5，分别代表PC+4、分支跳转地址、jal指令跳转地址、jr指令目标地址、中断程序首地址和异常处理程序首地址。

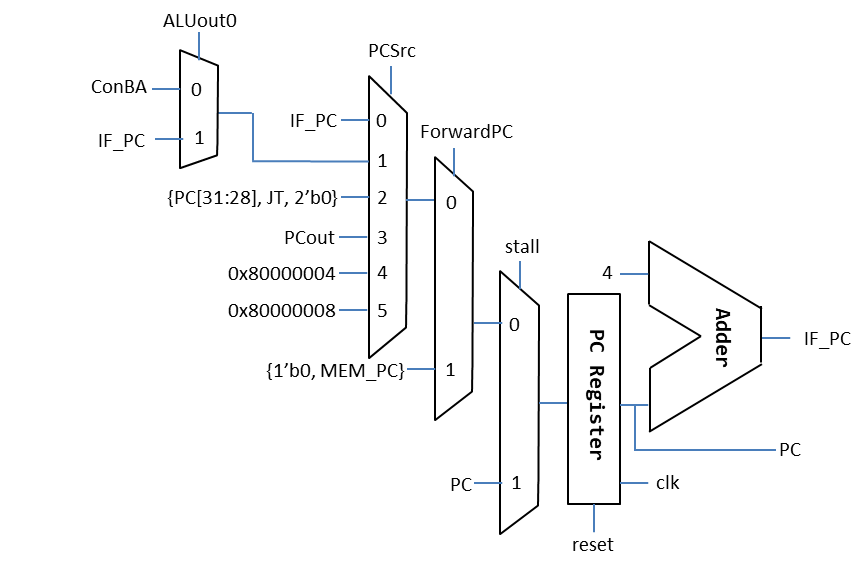


图7 流水线CPU对PC寄存器的修改

其二，RF读取的数据也有更多的可能性。为了保证RF能够在一个周期内同时读写，需要加入转发信号RF\_R&W\_Forward（在程序中没有这个变量名，只是为了画图方便起见），决定RF读出的数据是否应该来自MEM/WB寄存器。为了尽快实现beq判断，也加入了ForwardC和ForwardD信号，它们从EX/MEM寄存器转发beq指令需要的操作数（Mem\_in）。经过这两级操作的数据，或者进行相等比较，或者送入ID/EX寄存器。



图8 流水线CPU对RF寄存器堆的修改

其三，ALU的输入数据要考虑更多的情况。对于第一个操作数，除了决定它是shamt还是$rs以外，还要检查是否需要从EX/MEM寄存器或MEM/WB寄存器转发。对于第二个操作数，ALUSrc2为0时选择立即数，ALUSrc2为1时选择$rt（考虑转发）。但当ALUSrc2选择立即数时，不代表$rt的值就没有用了；在sw指令中，$rt是被存入存储器的操作数（sw指令很特殊，它既需要imm，也需要$rt）。因此EX阶段还应该有一个输出名叫EX\_rt\_postForward，用来记录转发后的$rt。

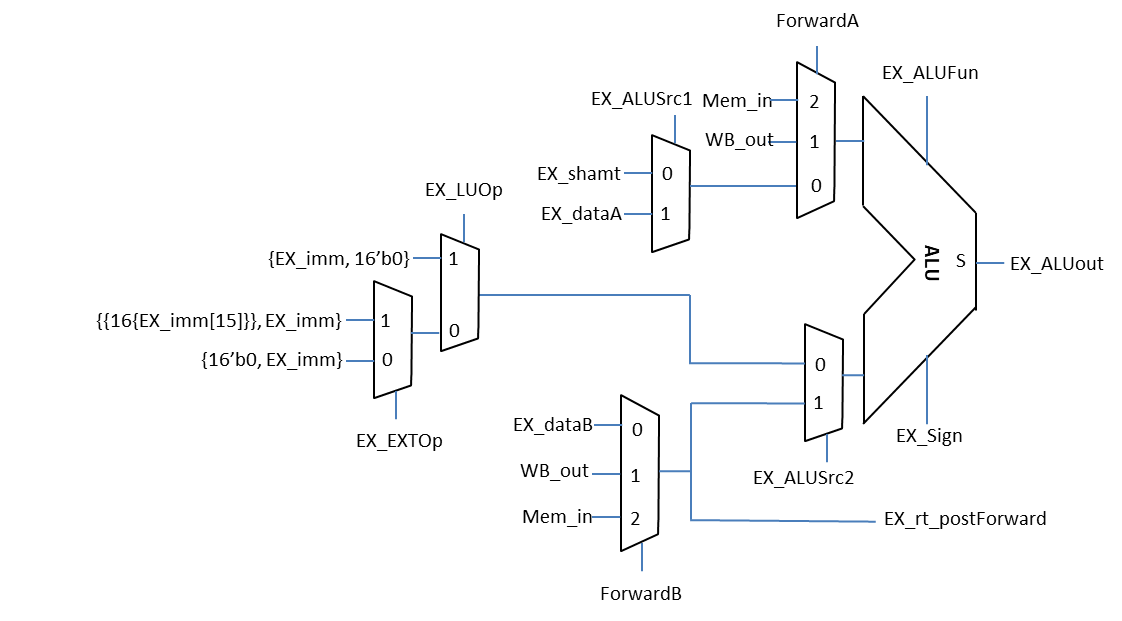


图9 流水线CPU对ALU的修改

其四，存储器访问和数据写回阶段也受到影响。存储器访问的写入数据，要考虑到write-and-store类型的转发，因此多加了一个MUX。WB\_RegDst信号控制的是寄存器堆写入地址，它们或者由$rd或$rt指定，或者是$ra或$k0。WB\_MemtoReg信号控制的是寄存器堆写入数据，WB\_inA表示不经过MEM阶段，直接将ALU结果写入；WB\_inB表示将存储器访问结果写回；2和3两个选项是为jr指令和中断信号预留的。

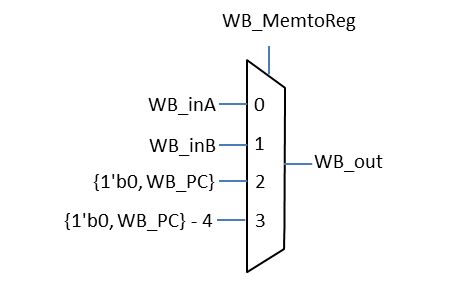
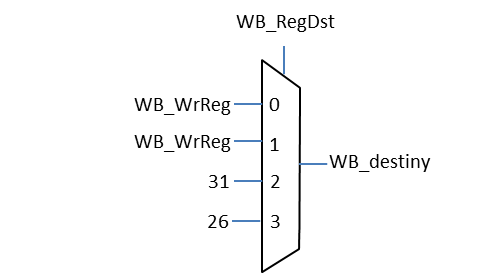
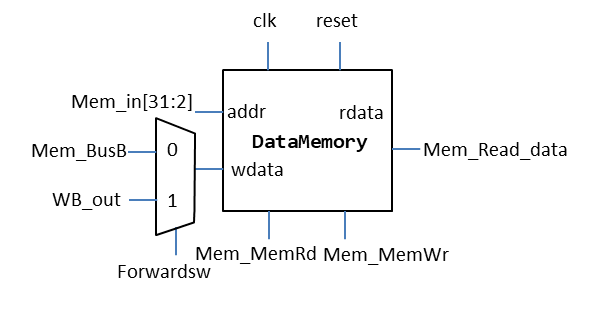


图10 流水线CPU对MEM和WB的修改

#### 2.3.6.2 转发单元

流水线CPU中共设计了7个转发通路，以应对不同的转发需求。

ForwardA和ForwardB分别控制ALU的两个操作数（$rs端和$rt端）的输入，这里解决的冲突的典型例子是R型-R型冲突。如果上一条指令要写寄存器，并且与本条指令的源寄存器相同，则将上条指令的运算结果直接从EX/MEM寄存器返回到ALU输入；如果上上条指令要写寄存器，并且与本条指令的源寄存器相同，则将上上条指令的运算结果从MEM/WB寄存器返回到ALU输入；如果上条指令和上上条指令都产生数据冲突，则以上条指令的计算结果为准。

ForwardC和ForwardD用于在ID阶段进行beq指令的跳转判断。如果上一条指令要写寄存器，本条指令是beq指令，且上条指令的目标寄存器地址与本条指令的源寄存器地址相同，则将运算结果从EX/MEM寄存器返回到RF的数据读出端进行相等判断。注意不必从ID/EX寄存器转发，因为beq指令在ID阶段会引起一个周期的阻塞（参见“冒险检测”中的write-and-branch冒险），因此beq进行跳转时，上一条指令正在进行MEM访问。也不必从MEM/WB寄存器转发，因为RF可以同时读写（这个问题后续会有说明）。

ForwardPC解决的是jal-jr冲突。如果jal指令向$ra存入的地址恰好是jr指令的跳转地址，就要进行转发。在ID阶段，检测本条指令是不是jr指令，并且检查EX/MEM寄存器中jal控制信号是否为1。如果转发条件成立，PC的新值就不应该从RF读出，而应该从EX/MEM寄存器的PC中读出。这里要从EX/MEM寄存器转发，而不从ID/EX寄存器转发，原因也是jal指令会在ID阶段阻塞一个周期，当后续的jr指令运行到ID阶段是，之前的jal指令已经到达MEM阶段了。

Forwardsw解决的是write-and-store冲突，或者在理论课的ppt上被称为存储器-存储器复制冲突，即上一条指令的计算结果，在本条指令中要存入存储器。检测上一条指令是否要写寄存器、本条指令是否要写存储器、本条指令的$rt与上条指令的目标存储器是否相同，若转发条件成立，从MEM/WB寄存器中将计算结果转发到存储器的数据写入端。

最后一个转发解决的是RF在同一个周期内读写的问题。我们发现，假如RF只有在时钟上升沿才能写入，那么做不到在同一个周期内既写入又读出。我们将RF改为组合逻辑，在任何时刻都能写入，但这会引起RF中数据的不稳定，因为组合逻辑会互相影响，造成混乱。因此必须加入转发功能，如果上上上条指令（间隔两条指令）要写寄存器，且目标寄存器地址与本条指令的源寄存器地址相同，则RF的数据输出端的数据应来自MEM/WB寄存器，而不是RF。

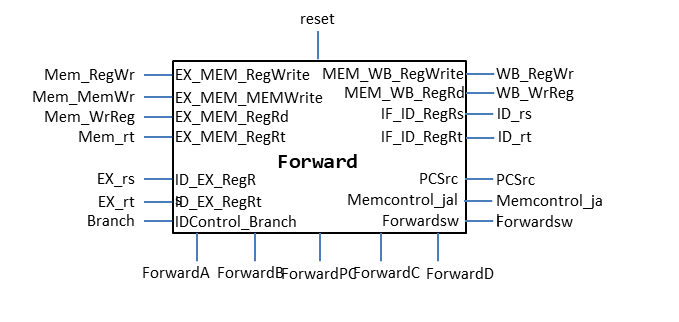


图11 转发单元输入输出框图

#### 2.3.6.3 冒险检测

冒险检测单元的主要功能是，根据需要将流水线阻塞一个周期，并清空某些段间寄存器的取值。它需要处理以下五种情况。

load-and-use冒险。在ID阶段，如果检测到上条指令是lw指令，且上条指令的$rt与本条指令的源寄存器地址相同，那么要求PC和IF/ID寄存器保持不变（阻塞一个周期），并且将ID/EX寄存器清空（相当于向lw指令和use指令中间插入了一条nop）。之后的工作由转发单元完成。

分支指令冒险。在ID阶段，如果检测到本条指令是beq指令，并且发现确实要发生跳转，则产生控制信号，通过PCSrc的选择向PC写入新值，并且清空IF/ID寄存器。这是因为PC+4这条指令并不是我们希望执行的，将它清空相当于在beq指令后面插入了一条nop。

跳转指令冒险。跳转指令不仅包括j和jal，还包括jr和jalr。这些操作和beq是类似的，在ID阶段发现要跳转后，一方面向PC写入新值，另一方面清空IF/ID寄存器，相当于阻塞了一个周期，在jump指令之后插入了一条nop。

write-and-branch冒险。如前文所述，我们在ID阶段进行beq分支跳转的判断。但如果beq的源寄存器地址是上一条指令的写入目标寄存器地址，那么在ID阶段beq指令需要的寄存器的内容还正在ALU中被计算，无论如何不能在同一个周期内获得。因此如果beq的上一条指令要写入寄存器，并且上一条指令的目标寄存器和beq的源寄存器相同，就需要stall一个周期（即保持PC和IF/ID寄存器不变，并向ID/EX寄存器中写入全0，相当于在beq之前又插入了一条nop指令）。等待ALU计算完成后，再从EX/MEM寄存器中将操作数转发回到ID阶段。

中断信号处理。中断信号与jal指令相似，都是将某个地址存入某个特殊的寄存器，并且向PC写入某个新的取值。因此大部分硬件实现可以和jal指令复用（例如阻塞和清空寄存器），只需要做一下或操作即可，但它们也存在一定的区别。首先，jal指令向$ra写入的是PC+4，但中断信号要求向$k0写入PC。这是因为中断信号的优先级更高，当ID阶段的Control译码模块检测到IRQ中断信号时，当前正在ID阶段的指令是没有被正确执行的，当中断返回后仍要执行它，因此在WB阶段，应该将MEM/WB寄存器中的PC减4之后写入$k0。另外，中断IRQ信号会保持很多个周期的高电平，直到中断处理代码关闭中断为止，不像jal指令的控制信号只有一个周期的高电平。这就导致Control模块一直在接收中断信号，来不及把它关闭。我们借用超声测距实验中产生窄脉冲的思想，将IRQ信号延时一个周期后取非，并和原来的IRQ信号做与运算，就得到了宽度只有一个周期的IRQ上升沿指示信号，用它作为Control模块的输入。



图12 冒险检测单元输入输出框图

# 3 仿真结果

## 3.1 ALU

为了调试ALU的性能，我们设计了测试代码ALUtest.v如下：

module ALUtest;

reg[31:0]A,B;

reg Sign;

reg [5:0]ALUFun;

wire[31:0]S;

ALU ALUs(.A(A),.B(B),.S(S),.Sign(Sign),.ALUFun(ALUFun));

initial

begin

A=32'h52;

B=32'h28;

Sign=1;

#104160 ALUFun=6'b000000;

#104160 ALUFun=6'b000001;

#104160 ALUFun=6'b011000;

#104160 ALUFun=6'b011110;

#104160 ALUFun=6'b010110;

#104160 ALUFun=6'b010001;

#104160 ALUFun=6'b011010;

A=32'h5;

#104160 ALUFun=6'b100000;

#104160 ALUFun=6'b100001;

#104160 ALUFun=6'b100011;

#104160 ALUFun=6'b110011;

#104160 ALUFun=6'b110001;

#104160 ALUFun=6'b110101;

#104160 B=0;

ALUFun=6'b111101;

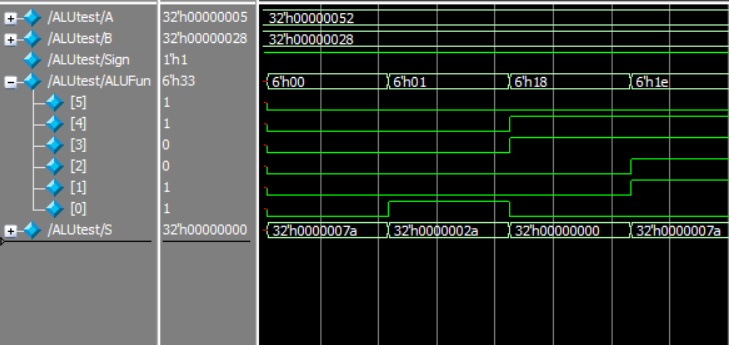
#104160 ALUFun=6'b111011;

#104160 ALUFun=6'b111111;

end

endmodule

依次改变ALUFun的值，输入信号1及输入信号2，观察仿真波形的变化。可见仿真结果均正确，和我们的预期一致。

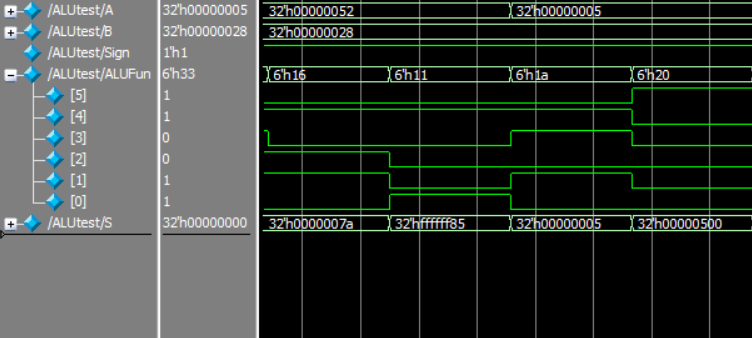


ADD

SUB

AND

OR

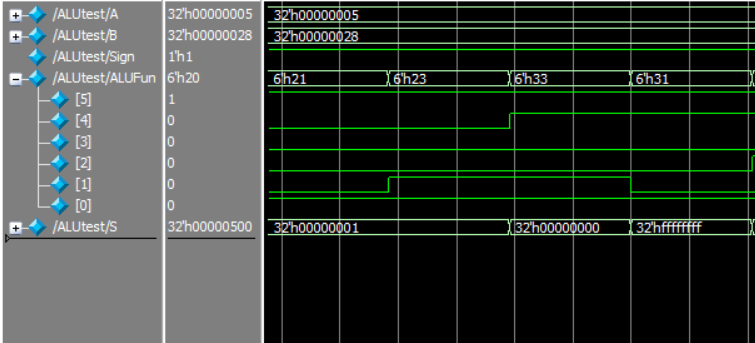


XOR

NOR

“A”

SLL

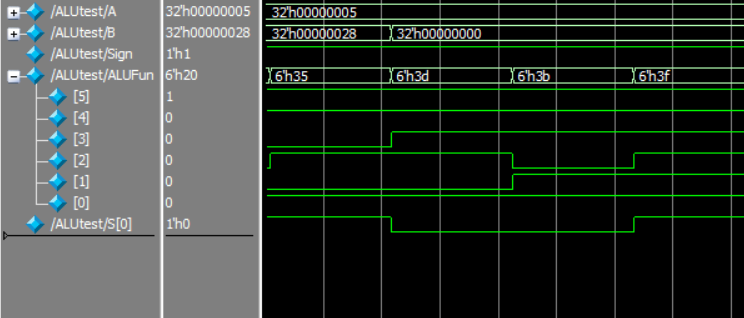


SRL

SRA

EQ

NEQ



LT

LEZ

LTZ

GTZ

图12 ALU功能仿真结果

## 3.2 单周期处理器

首先观察单周期整体仿真结果。测试数据为：输入操作数8和6，最大公约数计算结果2。检查串口的接收和发送数据正常。

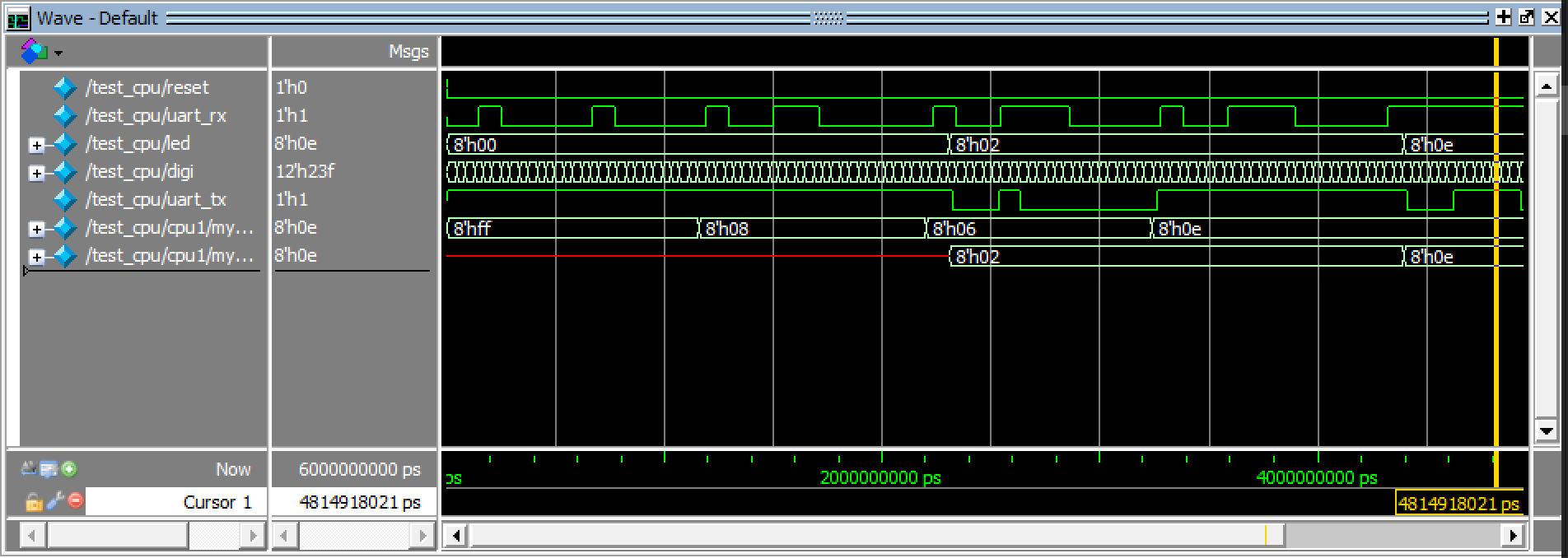


图13 单周期宏观仿真结果

其次我们关注一些重点模块内部的仿真情况。

**·串口内部变量仿真情况：**

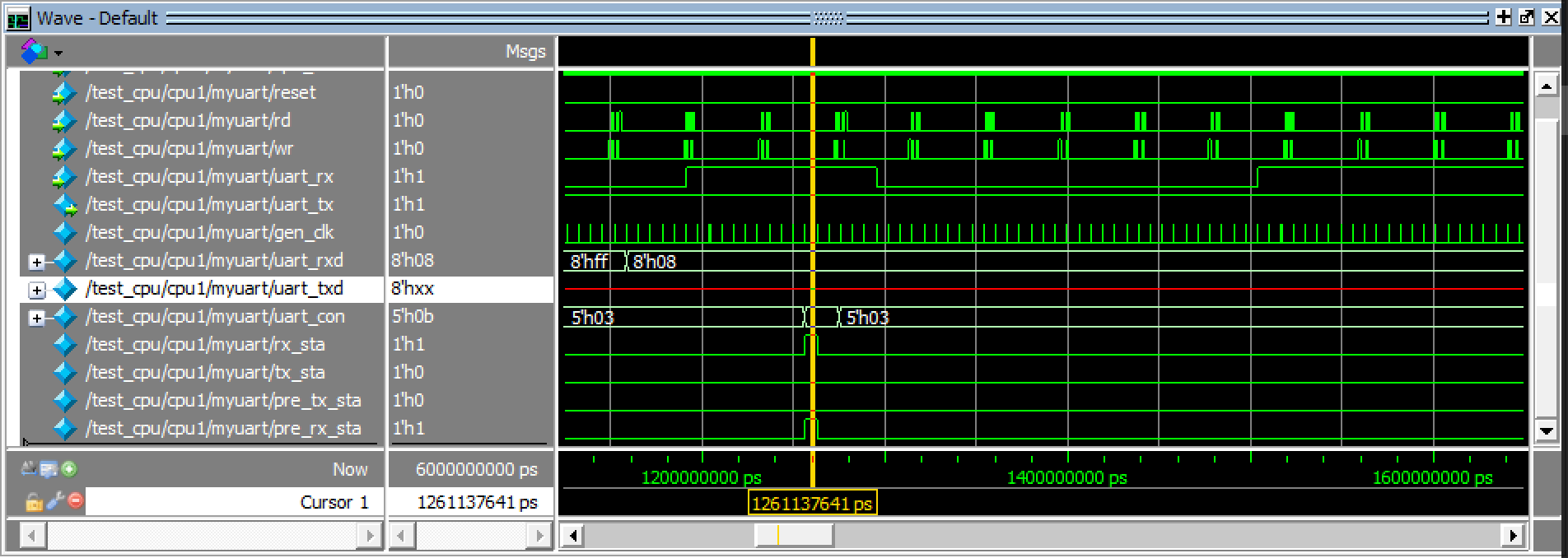


图14 单周期CPU串口内部仿真结果

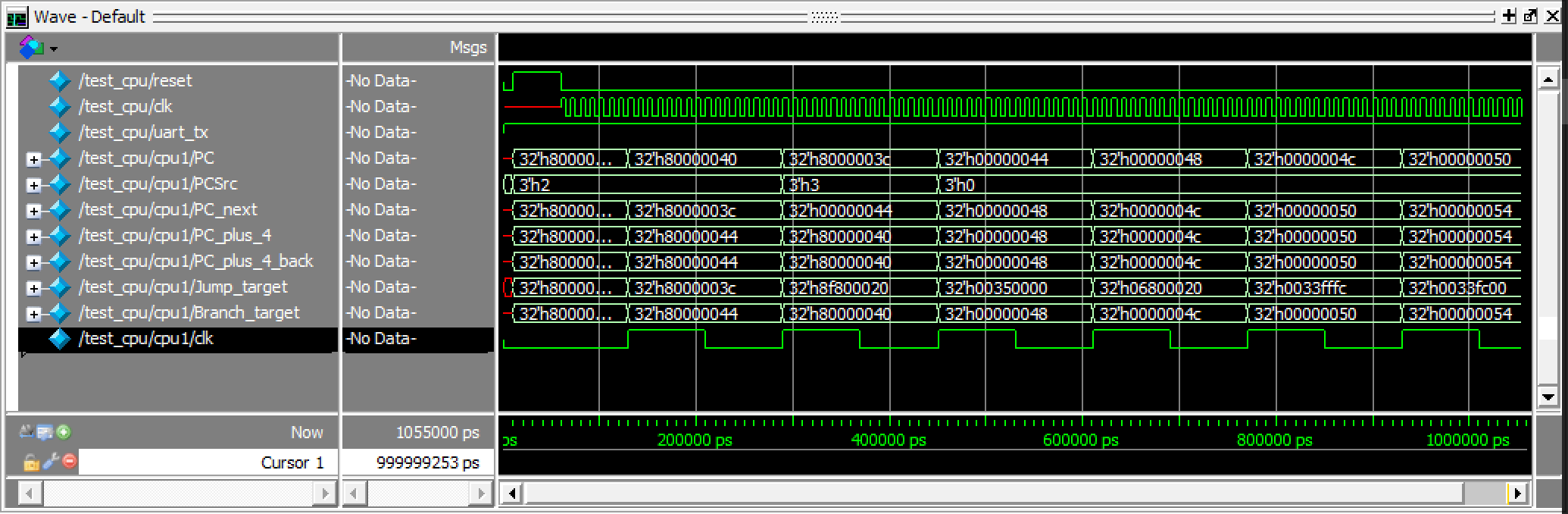
gen\_clk是根据系统时钟分频得到的串口时钟，观察可以看到在uart\_rx接收每一位比特时间内，gen\_clk都会有16个采样点，符合要求。

uart\_rxd中存放当前接收到的新数据。uart\_txd中存放的是即将发送的数据（最大公约数）。

接收到新数据的标识rx\_sta在新数据接收完毕会形成一段窄脉冲。

注意截图中uart\_txd为不定态的原因是截图展示的仿真时间是接收第一组操作数中的第二个数时，这个时候最大公约数还没有计算出来，所以即将发送数据的缓存里是不定态。

**·PC变化情况，以PC初始状态进入运行正常状态时举例：**



Jr

Jal

J

图15 单周期CPU的PC跳转仿真结果

PC在reset之后是0x80000004，第一条指令jump，PCSrc为2，表示PC即将按照Jump\_target更新，在下一周期PC变为0x80000040，首位不变，指向第16条指令，跳转正常。接着执行jal，跳至第15条指令。然后执行jr，PCSrc为3，表示按照寄存器里的值更新PC。之后开始正常执行最大公约数程序，PCSrc为0，PC不断加4。

**·存储器读取情况：**

我们一共有三块存储器，下面分别检查一下从它们当中读出数据时会不会出现冲突。

三块存储空间（数据存储器、串口、外设）每次只有一个作为源提供数据，从仿真情况来看Per\_Read\_data，Uart\_Read\_data，Mem\_Read\_data中至多有一个非0，它们的或就是Read\_data。

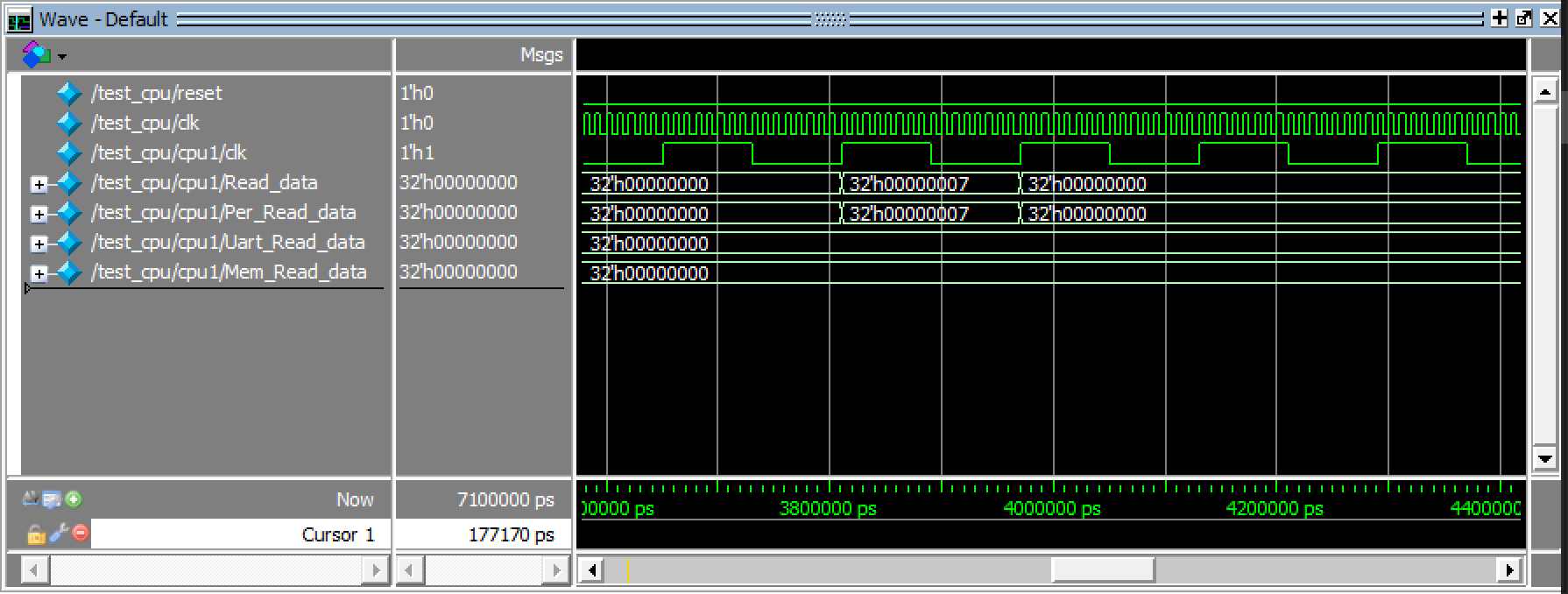


图16 单周期CPU从外设读取数据仿真结果

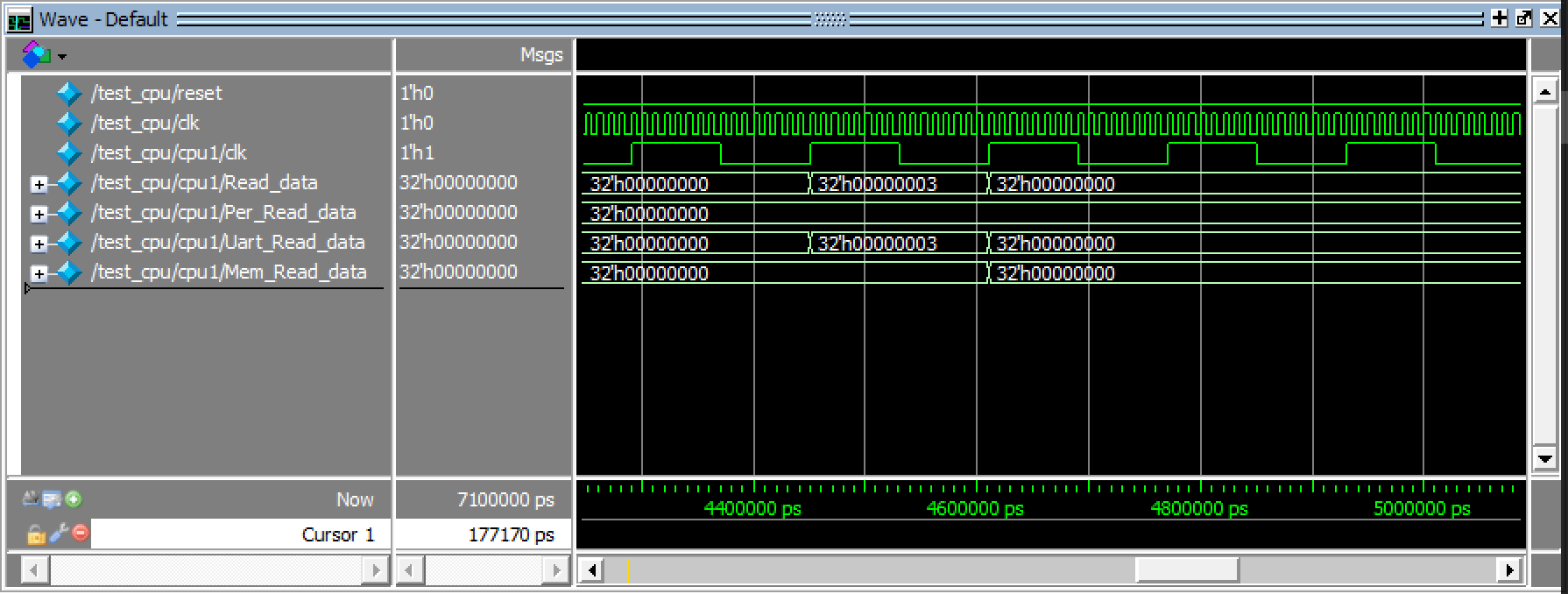


图17 单周期CPU从串口读取数据仿真结果

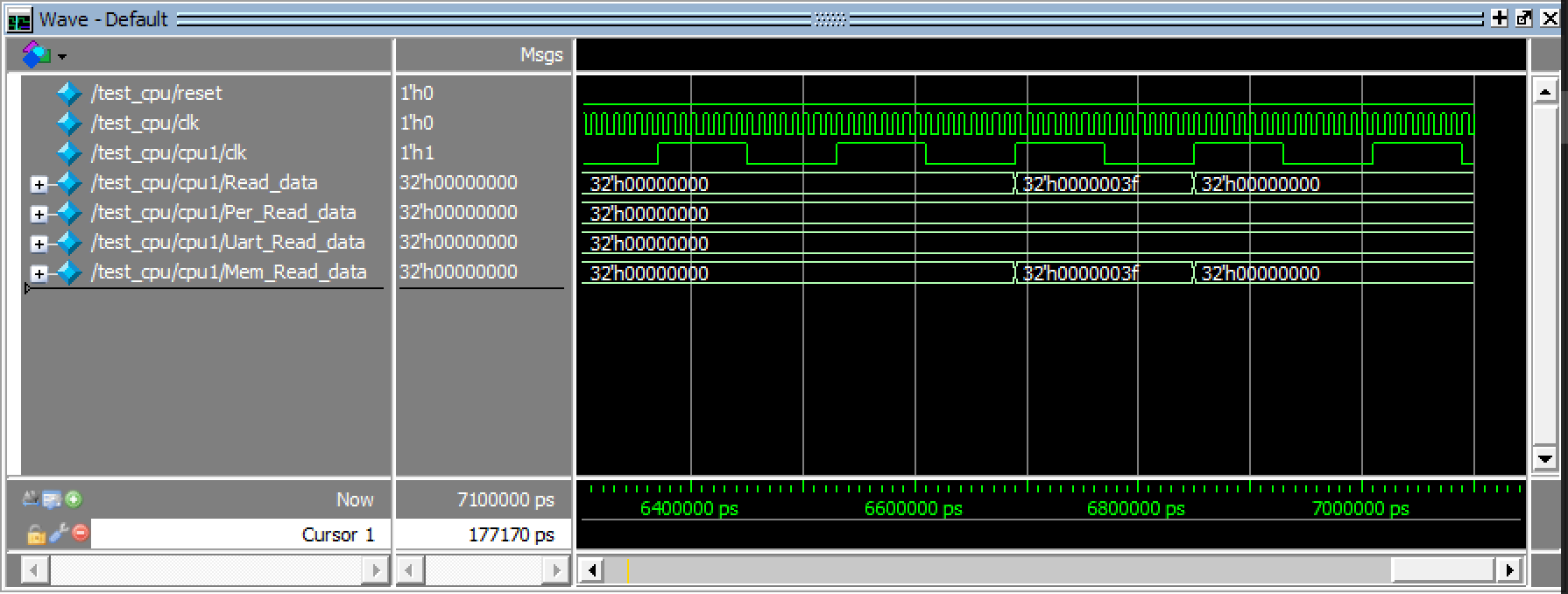


图18 单周期CPU从数据存储器读取数据仿真结果

**·中断处理情况：**

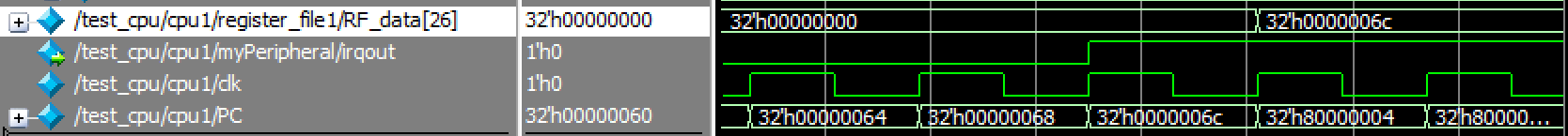


图19 单周期CPU处理中断信号仿真结果

当中断信号irqout到来时，PC的值为0x0000006c，指向的即将执行的下一条指令的地址，这时由于CPU需要去执行中断程序，所以需要把这一条指令的地址保存好，以便中断程序结束后回到这个地方继续执行。观察$k0（即图中的RF\_data[26]）信号，发现它在irqout到来的下一个周期值变为0x0000006c，现场成功保存。

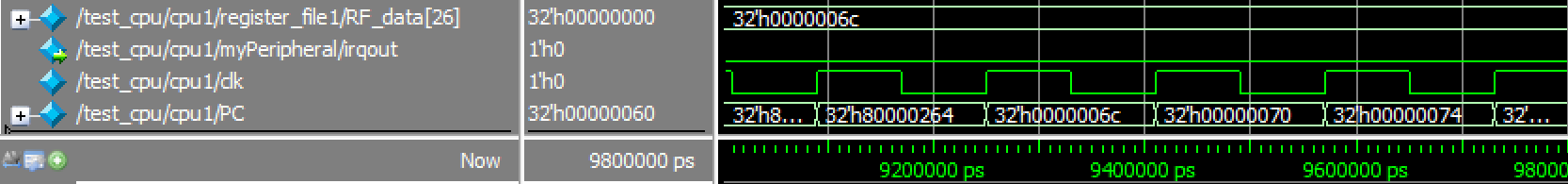


图20 单周期CPU中断程序结束后返回仿真结果

这是中断程序结束后返回最大公约数程序的仿真结果。红色所框的恰好是保存现场的时候存入的指令地址。检查红框之前一条指令地址对应的也正好是中断处理程序的最后一行，现场返回正常。

**·数码管点亮：**

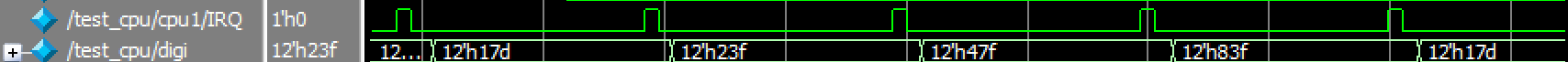


图21 单周期CPU点亮数码管仿真结果

数码管通过中断代码在软件层面上实现依次点亮刷新，四只数码管上显示的应该是两个16进制的操作数。仿真时用的操作是“8”和“6”，观察数码管的显示digi，分别为12’h17d，12’h23f，12’h47f，12’h83f。对应到数码管为6，0，8，0，在绑定数码管的时候我们将最先点亮的数码管绑在硬件上最右边，这样显示出来以后就是按照从左到右0806的顺序，符合两个操作数的要求。

## 3.3 流水线处理器

首先我们仿真观察整体功能实现情况。从宏观上来看，通过串口uart\_rx向CPU发送两个操作数0x08和0x06，当CPU通过中断处理程序读取到这两个操作数后，计算出它们的最大公约数为2，于是通过串口uart\_tx发送8’b0000\_0010。同时，LED的显示为8’b0000\_0010，数码管digi显示的分别是08和06这两个操作数。从宏观上来看，仿真结果是正确的。

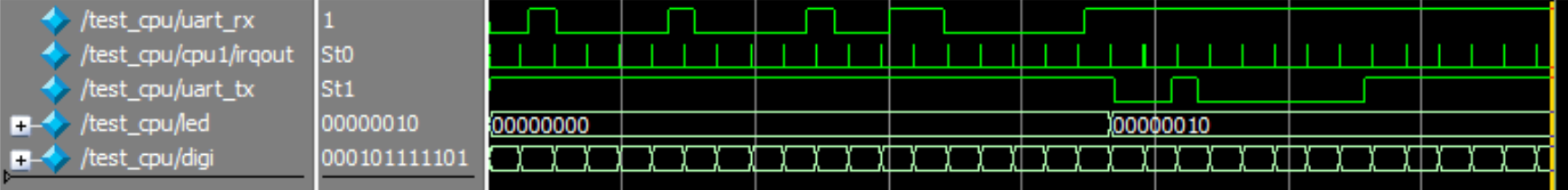


图22 流水线宏观仿真结果

现在我们来观察各个细节的仿真情况。我们分别选取能够体现IF、ID、EX、MEM和WB各自功能的典型的指令序列，结合转发、冒险和中断来说明CPU的运行情况。

首先我们来检查各种情况下PC是如何跳转的。在CPU刚刚复位时，PC的状态为0x80000000。这是一条jump指令，PC跳转到0x80000040（注意jump指令ID阶段PC的取值，即0x80000004，是无效的；虽然它成功完成了取指，但冒险检测单元会对IF/ID寄存器进行flush操作），这是因为PCSrc为2。紧接着遇到jal指令，同样在ID阶段，PCSrc变成2，使得PC跳转到0x8000003c。但0x8000003c是一条jr指令。当jr指令运行到ID阶段时，jal指令到达了MEM阶段，因此ForwardPC置为1，将$ra的值从EX/MEM寄存器转发回到PC。虽然这时候PCSrc为3，但ForwardPC的优先级更高，PCSrc在ForwardPC面前是无效的。

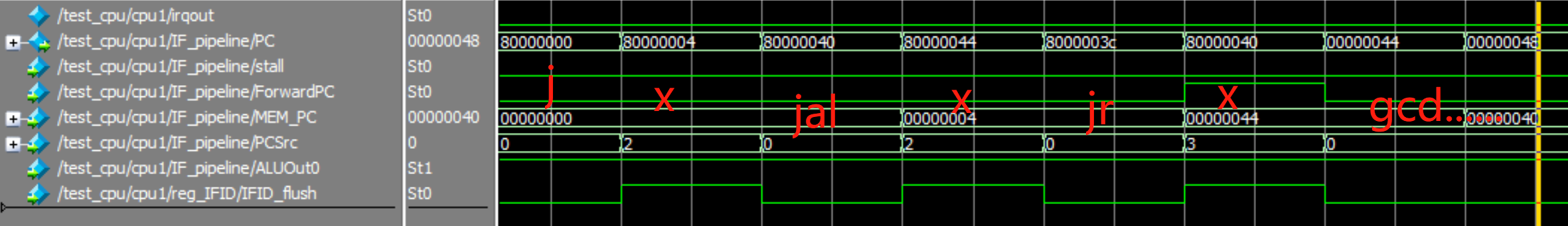


图23 流水线j型指令与PC跳转仿真结果

当中断来临时（下图中1号周期），当前这条指令（0x00000074）不能正常执行了，因为Control单元接收到IRQ信号后，将PCSrc置为4，下一个时钟沿到来时，PC变为0x80000004。但不幸的是，在2号周期内，由于ID\_PC最高位为0（注意ID\_PC比IF\_PC落后一个周期），并且IRQ为高电平，因此PCSrc仍然为4，相当于阻塞了一个周期，因此在3号周期内，PC仍然为0x80000004。直到3号周期内，ID\_PC的最高位变为1，中断信号不再对Control模块起作用，这时0x80000004指令也运行到了ID阶段，PCSrc才得以变为2（JT），继续执行中断代码。

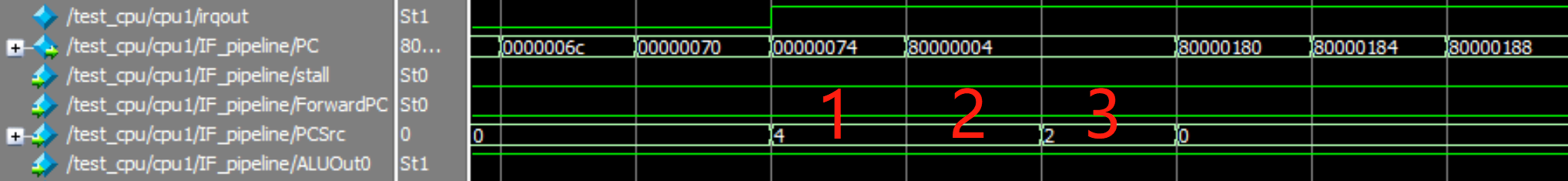


图24 流水线中断到来时PC跳转仿真结果

下图是针对beq的典型仿真例子。左起第一条指令时andi，其目标寄存器地址与beq源寄存器地址相同。当andi运行到ID阶段时，发现andi的前一条指令时lw，这里有load-and-use冲突，因此左起第二个周期是stall，那么在左起第三个周期内，andi在译码，beq在取指。到了左起第四个周期，beq进行译码，很不幸，发现它的源寄存器和andi的目标寄存器一致（虽然andi指令被stall了一个周期，但在我们的设计中，load-and-use情形只将ID/EX寄存器的MemRead，MemWrite，RegWrite三个变量清零，其余控制信号和数据照常流水，因此在左起第四个周期，andi的目标寄存器地址已经到达EX/MEM寄存器中了），那么左起第四个周期再一次stall。这里发现beq要使用andi计算出来的值，ForwardC已经变为1了，但没有关系，当阻塞结束以后，发现经过转发的ID\_dataA和ID\_dataB确实相等，因此在第五个周期内，ALUout0置为0（第四个周期内ALUout0为0，这纯属巧合），又因为PCSrc为1，PC成功跳转到了lw指令处。注意，第四和第五个周期内，PC的取值为0x800001c0，这是下一条beq指令，但这条指令根本没有被执行。当0x800001bc处的beq指令发生跳转时（左起第六个周期），冒险检测单元会发现后续不该执行的指令进入了流水线，并对IF/ID寄存器进行flush操作。

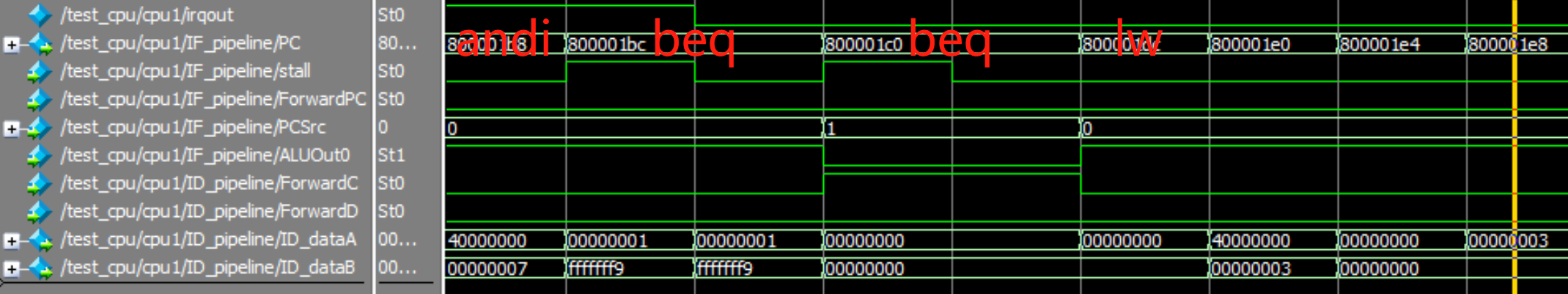


图25 流水线beq指令与PC跳转仿真结果

在ID阶段，流水线和单周期最大的不同，还是在于beq指令的提前进行，以及配套需要的ForwardC和ForwardD两个转发。但这已经在上面提到了，不再赘述。我们直接来看EX和MEM阶段的典型仿真结果。

左起第一条指令是lw，第二条指令是add（需要用到lw的结果），第三条指令是sw（需要用到add的结果）。在左起第三个周期，add进行译码，冒险检测单元发现存在load-and-use冒险，因此产生stall信号，一方面PC停止一个周期，另一方面将ID/EX的读写使能信号置为0。如上文所述，在我们的设计中，包括目标寄存器地址在内的其他控制信号照常流水，不受stall的干扰，因此在第三个周期内，ForwardA已经变成了2（表示从EX/MEM寄存器转发）。当stall结束，add指令进入EX阶段时（第五个周期），转发模块又发现，lw指令已经运行到WB阶段了，所以ForwardA及时地改变为1，从MEM/WB寄存器将操作数0x0000003f转发到ALU的输入端。在左起第七个周期，sw指令执行存储器访问阶段。在这里，我们发现add指令的目标寄存器和sw指令的源寄存器$rt是一致的，因此Forwardsw信号变为高电平，待写入数据Mem\_WriteData由WB\_out（倒数第三行）转发而来。不幸的是，这里Mem\_BusB的数值和WB\_out的恰好相同，没有体现出转发的效果所在。

值得注意的是，Mem\_BusB在第七个周期中的取值0x0000013f，实际上是来自于第六个周期中的EX\_rt\_postForward。这个信号是专为sw设计的，如上文所述，sw这条指令既需要imm，又需要$rt的值。另外，在仿真图形中出现了不定态，但这并不是代码的bug。我们为了减少综合时的warning（详见“综合情况”部分），有意地将Control单元中的无关项赋值为了x，这些不定态会传递到某些后续的信号当中，但这些不定态都不影响CPU的正常运行。我们认为，告知Vivado无关项有哪些，可以更好地帮助Vivado进行卡诺图化简。

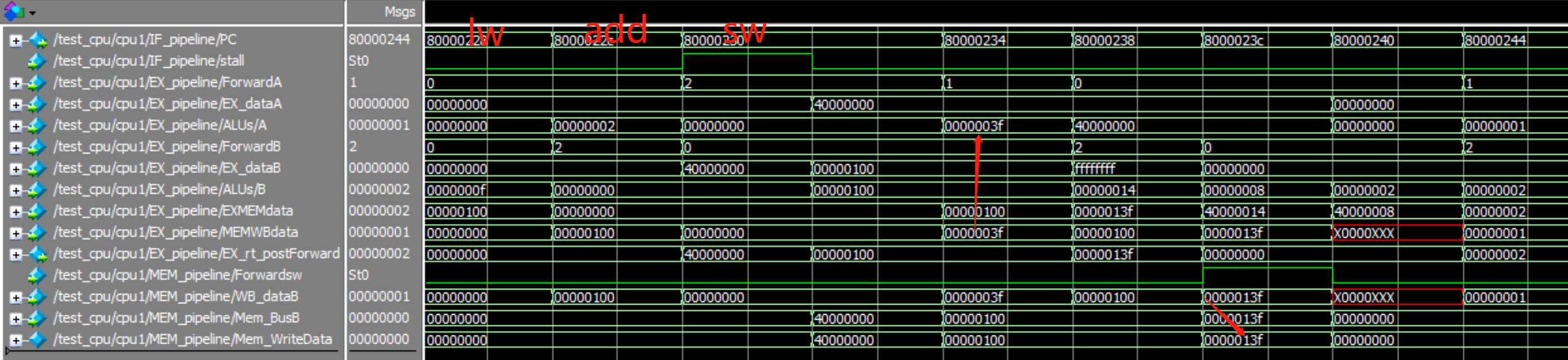


图26 流水线EX和MEM阶段典型仿真结果

MEM阶段的外设和串口的仿真结果在单周期里已经有展示了。对于WB阶段，ALU的计算结果写入RF和MEM的读取结果写入RF都是平凡的，因此我们主要展示jal指令和中断来临时WB的表现。

如图，0x80000040是一条jal指令。当它运行到WB阶段（第五个周期）时，RegDst选择为2，即31号寄存器$ra；MemtoReg也选择为2，即WB\_PC的取值（也就是最初的IF\_PC的取值，即PC+4）。因此把0x00000044写入了$ra寄存器，这也间接地改变了PC的最高位。

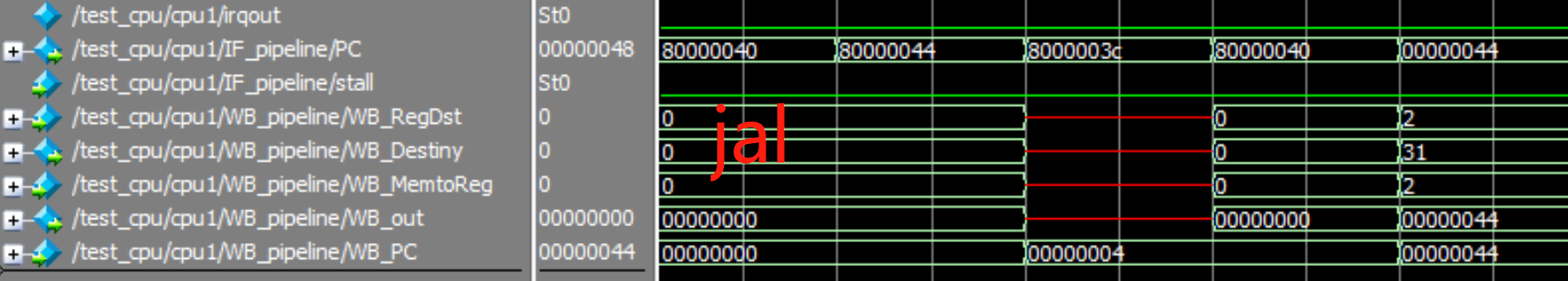


图27 流水线WB阶段处理jal指令仿真结果

当中断来临时，IRQ信号进入Control模块，就好像一条jal指令进入了ID阶段。当这条“中断指令”进行到WB阶段时可以看到，RegDst选择为3（26号寄存器$k0），MemtoReg选择为3（WB\_PC-4，即0x0000007c），将这一地址写入$k0寄存器。



图28 流水线WB阶段处理中断信号仿真结果

# 4 调试情况

## 4.1 单周期处理器

单周期处理器的调试主要涉及汇编代码、汇编器、单周期本身数据通路以及和外部设备联合的调试。调试的难度不大，只是过程比较繁琐。大部分外部单元在调通以后直接复用到流水线中，减少了后面的工作量。调试主要在软件上仿真进行，仿真结果正确以后，在硬件上通过加debug core观察中间信号以及将关键绑定在led上观察的方法很快跑通。

### 4.1.1 汇编器

我们的汇编器是用python写的一段小程序，本质上就是字符串处理。

中文注释的存在使得汇编器无法工作，经检查是因为文件编码格式被默认设为gbk，改为utf8问题解决。

写汇编器的时候没有和写汇编代码的同学约定好数制，调试时决定同时支持10进制和16进制，在后来改汇编代码的时候，这个设定使得代码编写更加自由。

处理跳转指令的时候，我一开始把偏移量写成了从PC到分支目标的差，事实上应该是从PC+4到分支目标的差。这个bug是在看数据通路仿真的时候发现的，再回过来改汇编器和指令存储器，过程比较愚蠢。这件事情告诉我做单元测试的必要性。比如应该在写好汇编器以后翻译一段涵盖所有支持指令的代码，然后与标准结果（可以从mars模拟器中获得）比对，确认没问题后再和后面的模块连接起来调。

### 4.1.2 PC最高位清零问题

在原始的指令存储器设定中，PC复位时取值为0x80000000，第0条指令会使PC跳转到求最大公约数程序入口，接着PC会开始按照正常程序的要求跳转。然而根据PC最高位的约定，在此过程中它一直会保持1，也就是在执行最大公约数代码时CPU一直处于内核态，这是不可接受的。为了使最高位置0，一定需要用寄存器中的值来给PC赋值，问题就变成了怎样比较好地给寄存器里存上最大公约数入口，而这个问题可以用jal指令解决。

最终我们的指令存储器写成伪代码是

*pseudo code*

0: jump to 16

...

...

15: jr $ra

16: jump and link 15

17: start gcd

...

### 4.1.3 数据存储器读出冲突

在烧录debug core之后发现，串口发送的操作数成功进入串口数据地址，但是写入寄存器的数竟然和串口里的不一样，总是多出几位1。一开始从软件层面检查，发现这是一个简单的lw操作，于是去查数据通路，发现lw涉及到把数据存储器里的数存到寄存器里，而我们的设计里有三个数据存储器，原则上它们的输出只能有一个是非0的。检查仿真发现会出现超过1个输出非0的情况。问题出在DataMemory只用了地址的低几位寻址，导致实际想访问外设地址的时候，也能从DataMemory里读出数来。解决方法很简单，寻址前加入高几位的判断即可。

### 4.1.4 串口时序

在第一版设计中，串口接收端接到一个新数据后rx\_status置1，只有reset才会清零；uart\_con记录串口收发状态，直连到串口收发端的使能或者输出信号。这种只能支持收一个数，不满足接收两个操作数的要求。

第二版修改了串口接收端，使得rx\_status在收到新数据后保持一个串口时钟的高电平，然后置0。rx\_status依然直连到uart\_con。这样虽然有了高低电平的变化，但是不符合uart\_con数位的规定了：约定是遇到“读”串口存储器时把uart\_con[3]清零，但这个版本所做的事情是uart\_con[3]的清零不依赖“读”信号。

最终我们的方案是手动检测rx\_status的上升沿来判断收到新数据，即在cpu\_clk的每个上升沿检查rx\_status，如果前一个周期为低电平，后一个周期为高电平，说明收到一个新数据，此时给uart\_con[3]置1，然后在遇到“读”信号的时候（汇编代码控制的把串口收到的数据读到寄存器里去）给uart\_con[3]清零。

有了设置接收状态的经验，我们对发送状态也做了类似的处理。在串口发射器里面，每一个数据发射从起始位到结束位tx\_status都置高电平，其他情况下置低电平。而在uart\_con[2]和uart\_con[4]的设置上，通过在cpu\_clk的每个上升沿检查tx\_status，如果前一个周期是高电平，后一个周期是低电平，表示发射完一个数据，置低电平，然后在遇到“写”信号的时候（把最大公约数计算结果写入串口数据地址中）把uart\_con[2]和uart\_con[4]置高电平。

### 4.1.5 MemtoReg数据通路的修改

我们原来按照理论课上的讲解，给MemtoReg设置了三种选择，分别是从ALU计算结果，存储器读取结果和PC+4选择写入寄存器。但是在调试的时候，有一次中断恰好发生在了j loop指令，检查$k0寄存器中的值，发现存入的是循环体后面的一条指令地址，这样中断结束以后就不会再执行循环体了，而为了程序正常运行应该要重新执行j loop运行一次循环。在这个bug的启发下，我们想到不仅是j指令，每个指令在遇到中断后都不能正常执行，应该保存的中断返回地址是当前指令而不是下一条指令。所以我们给MemtoReg加了一种选择，可以将PC写入寄存器。

### 4.1.6 数码管按字寻址

仿真发现在可以把两个操作数正确地读进内存的情况下，扫描显示译码结果也不对。检查发现是汇编代码和数据存储器之间的接口没有对好。传给数据存储器应该是最后两位是0的地址，而汇编代码里面写的是去掉两个0的地址。修改汇编之后问题解决。

### 4.1.7 支持接收多组操作数

实验指导书中没有明确说需要接收多组操作数，我们为了一次性给多个测试数据，决定创造这个需求。

我们一开始的汇编程序（包含中断）是只为实现一组操作数设计的，为了支持接收多组操作数，需要在接收完一组数据之后把准备接收操作数的地址清空。但是我们不能简单地用两个寄存器实现接收操作数和清零，因为这两个寄存器还会被绑定在数码管上实现操作数显示，到时候就会出现闪烁。所以我们最后用了四个寄存器$s0,$s1,$s5,$s6实现。$s5,$s6负责从串口读操作数，每当完成一组操作数求最大公约数，就把$s5,$s6清空，准备接收新的操作数。$s0,$s1是$s5,$s6的缓存，没有清空过程，负责绑定到数码管上显示。

调试主要修改了中断部分的汇编代码以及连带的指令存储器，调试结果四个寄存器的变化情况和预想的一样。

### 4.1.8 定时器周期

上板子调的时候发现，数码管每段线都有亮度，应该亮的几根线和其他的线亮度区分不明显。与其他同学讨论之后发现是数码管扫描频率过高的问题。数码管扫描只在中断时进行，只要把定时器中断频率降下来，数码管的显示就清晰了。

## 4.2 流水线处理器

### 4.2.1 寄存器堆

初始进行仿真时，发现在使用单周期的寄存器堆设计的情况下，没有办法读出刚写入的数，翻阅书本，课件上的解释是，要确保写入在前半个周期完成，而读出在后半个周期读出。由于vivado不支持三个时钟触发的触发器，因此如何用时序逻辑实现这个功能，我一开始并没有找到很好的办法。于是我把这改成了组合逻辑，既不需要时钟来触发，随时可以读写。在仿真上，这样做没有问题，但是在烧板子时，出现了写入寄存器堆错误等不稳定的情况，我们采取了用手按控制时钟检测的方法，发现写入使能、转发信号、写入寄存器编号甚至写入数据值都是对的，然而vivado上对寄存器堆的监测发现，目标寄存器的值就是不对，甚至当我们重复同一操作时，寄存器的值还有变化，这证明了组合逻辑的设计被实践证明是不稳健的。因此最后的解决办法是，依然使用单周期的寄存器堆设计，但是增加一条转发通路，以实现可以读出同一个周期内写入的数。

### 4.2.2 转发单元

流水线设计了多条转发通路，除了上文提到的，从WB阶段向ID阶段的转发以确保可以同时读写外，还有从MEM阶段向EX阶段的转发；MEM阶段向EX阶段的转发；用于提前判断分支语句的，从MEM阶段向ID阶段的转发等，这些都是经典的转发必须设计。但当我们进行调试时，我们发现了一些其它的意外情况。一开始设计时，考虑到ALU可能是critical path，我们将ALUSrc控制信号提前到ID阶段完成，而转发则是在EX阶段，结果出现了往立即数转发替代等的奇怪情况，于是我们退回了经典设计，还是在EX阶段进行数据通路的多路选择，解决了这个问题。又比如jal命令过后，当PC地址还未写入31号寄存器堆时，就要执行jr $ra操作，这时就需要一个转发，把还在MEM阶段传递的PC地址提前转发到IF段。又比如对sw写入操作数的转发，执行该操作数时，连接ALU的databusB上传输的是立即数，按照原有的设计，这个时候是没有转发的，这就会导致rt寄存器的值没有得到它应有的被转发数值，因此我们单独为这种情况设计了一个转发。

### 4.2.3 冒险单元

初始的冒险设计，包括分支指令判断成功后或者跳转指令时对instruction置零，load-use数据冒险stall一次pipeline，reg-beq数据冒险需要的一次stall。在初始调试时，发现来自于分支指令的冒险检测有问题，当提前分支判断的输入数据需要被转发时，第一个周期送入分支判断功能的数其实是无效数据，然而这个无效数据的结果可能是分支预测成功，需要跳转，那么此时冒险单元就会错误地生成flush控制指令。为此做出的修改是，在进行分支冒险判断前，先进行reg-beq数据冒险的转发判断，来避免由无效输入数据生成的flush。

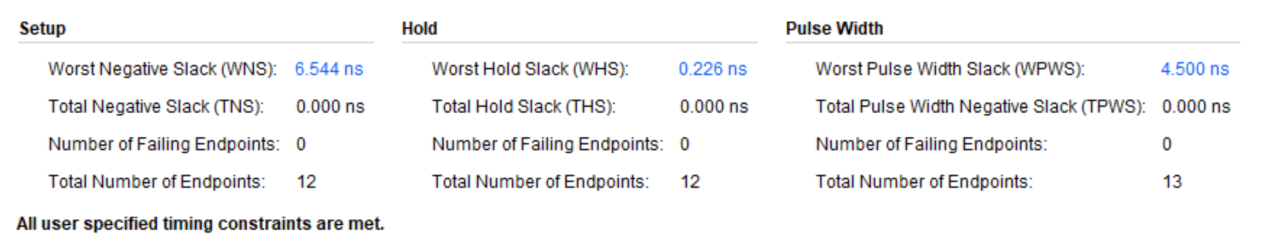
加入中断指令后，冒险单元需要单独为中断指令考虑设计。中断语句其实可以类比成一个把PC值送入26号寄存器的jal语句，这本身不难，但是在对中断信号的处理上，却下了一番功夫。当大概了解中断功能后，初始设计时，我们在中断信号处于高电平时，进行flush操作，并跳转PC，但是仿真告诉我，中断信号会维持高电平一段时间，那这里就会出问题，会使得有一个连续的flush，且由于中断信号变化与时钟是同步的，这就会导致中断信号的建立时间不够，不能确保送入触发器的是正确的值。于是我们打算改变设计，只有当前一个周期中断信号为低电平，且本周期为高电平时，才进行flush操作。由于我们采用并行赋值操作，我发现对中断信号的现态和次态赋值会出现延后一个周期的错误，在经过仔细思考过后，我把这些状态赋值改为了组合逻辑，然后使用触发器进行最终flush使能信号赋值，取得了成功。

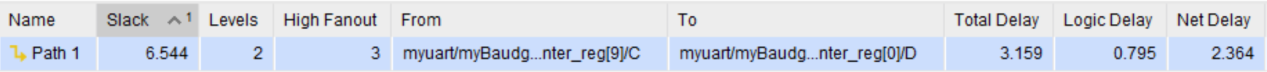
# 5 综合情况

## 5.1 单周期处理器

### 5.1.1 时序性能

根据Vivado给出的时序分析报告，当单周期CPU使用50MHz主频时，建立时间余量为6.544ns，保持时间余量为0.226ns，满足时序要求。检查关键路径，发现是串口时钟的分频模块占据了关键路径。根据以往的经验，进行高倍分频时，计数器的加法总是需要很长时间执行，因此这属于正常情况。如果将主频提高到100MHz，Vivado会报错，称时序约束无法满足。鉴于关键路径是分频模块，这一模块是必不可少的并且很难继续优化，我们认为，运行在50MHz的单周期CPU从时序性能方面来说，基本令人满意。





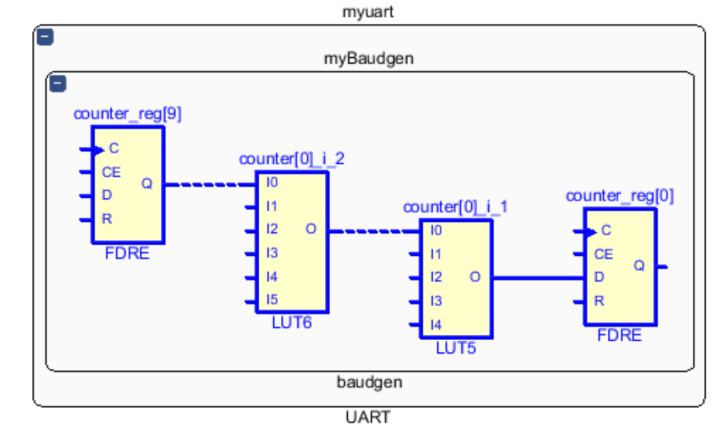
 

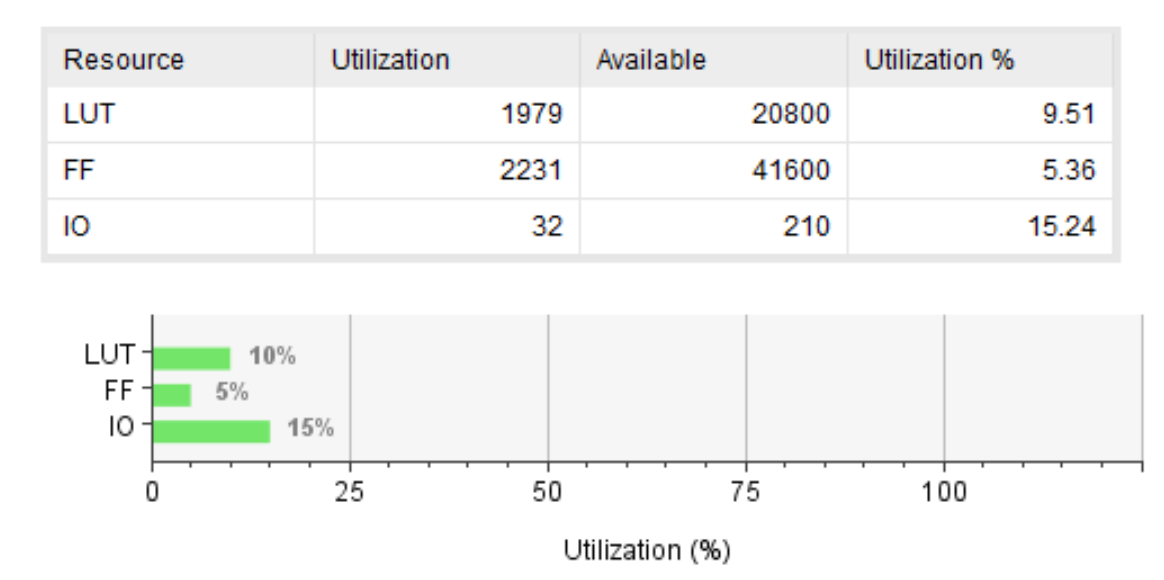
图29 Vivado为单周期CPU给出的时序分析报告、关键路径及其Schematic和Device示意

### 5.1.2 资源占用情况

根据Vivado给出的资源占用情况报告，单周期CPU共计使用查找表（LUT）1979个，触发器（FF）2231个，输入输出端口（IO）32个，占资源总量的9.5%，5.4%和15.2%。

对于触发器，DataMemory使用了1024个，RegisterFile使用了992个，这是正常的，因为数据存储器和寄存器堆都需要D触发器来存储数据。外设也使用了一部分FF，应用于定时器中的计数器，或串口中的分频模块，或串口的数据缓冲区等位置。还有一些Leaf Cell状态的触发器用于连接输出端口，每个输出端口都由一个固定的触发器来驱动，这使得CPU的输出更加稳定。

对于查找表，寄存器堆使用了1465个LUT，使用量最大；数据存储器和外设也有一定应用，但ALU对LUT的使用量反而很小。ALU只用了9个LUT，大约是因为只需要这么多；RF用了1465个，一方面是因为需要一些LUT进行MUX的选择操作，另一方面，在RF的Schematic图中看到了包括PC、外设、串口、数据存储器在内的端口名字，或许是Vivado综合和实现的过程中，将其他模块的资源占用算在了RF这里。（Vivado或许对RF有某些误解？）在这些LUT中，还观察到了F7MUX和F8MUX，查询Xilinx的相关手册得知，F7MUX和F8MUX类似于一种“轻量级”的LUT，这些多路选择器的运算延时非常小。可能是Vivado为了优化时序性能，在合适的地方选择了MUX，而没有使用LUT2。



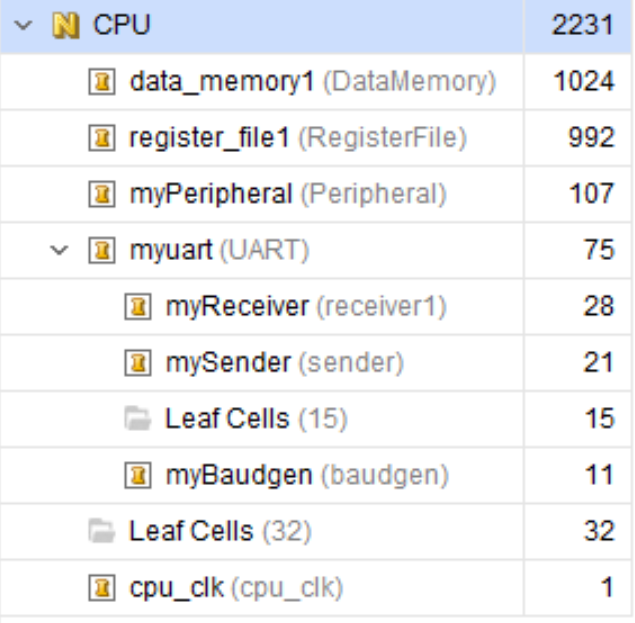
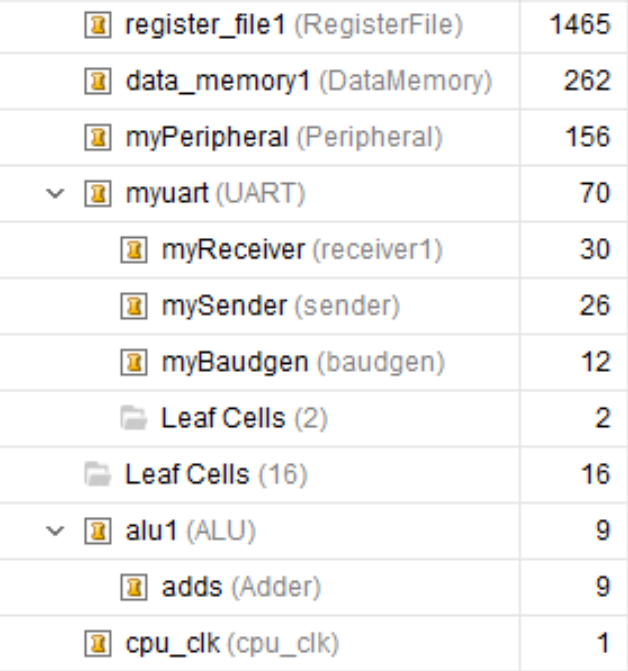
 

图30 Vivado为单周期CPU给出的资源占用情况

左下图为FF的使用情况，右下图为LUT的使用情况

### 5.1.3 其他信息

Vivado也给出了单周期CPU的功率信息。总片上功率为0.124W，其中动态功耗占42%，静态功耗占58%。随着工艺的进步，CMOS器件的动态功耗已经低于静态功耗了，可以看到，虽然直流漏电流很小，但由此引起的静态功耗已经是设计过程中不能不考虑的重要因素了。

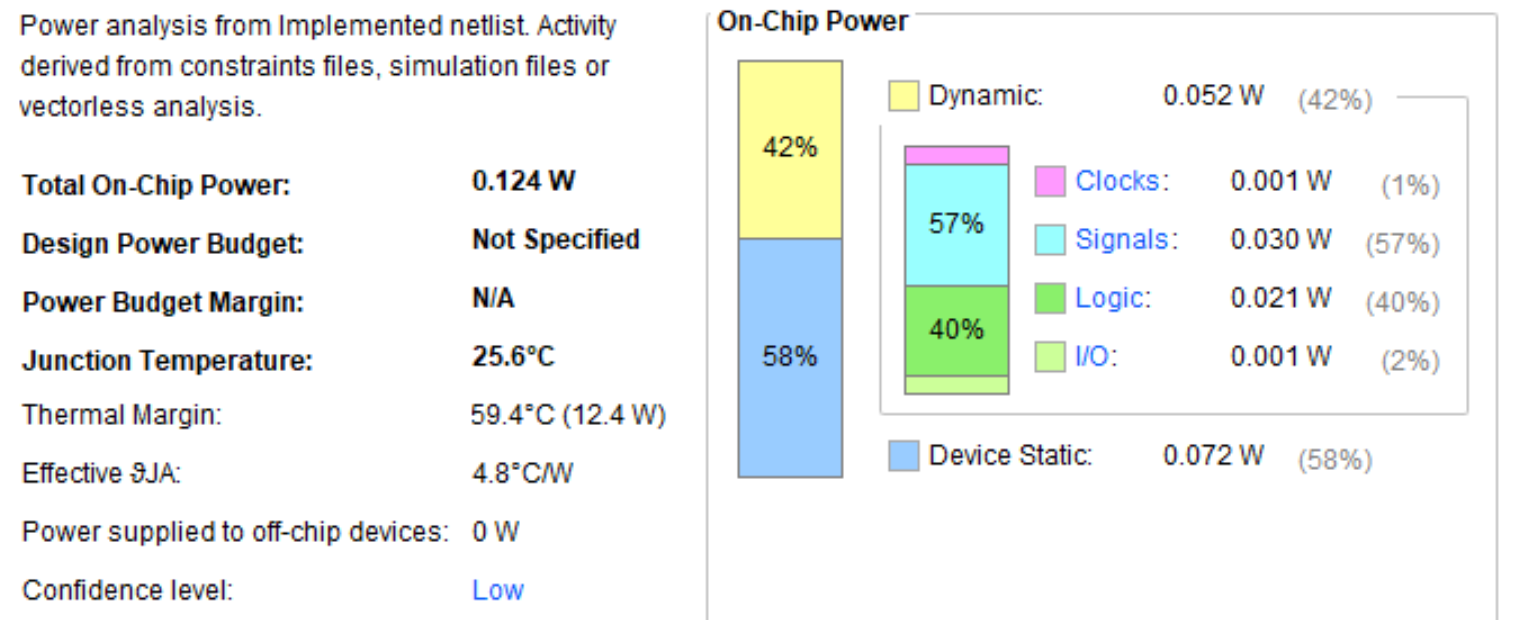


图31 Vivado为单周期CPU给出的功耗评估报告

另外值得一提的是，我们的单周期CPU代码风格经过了仔细的优化处理。我们删去了所有无用的变量，对没有接线的端口、未使用的连线都进行了修整，使得Vivado的综合过程没有Warning提示，实现过程只有一个Warning，并且出在Generate Bitstream环节。它指出的是.xdc约束文件中的漏洞，但由于能力有限，我们并没有解决这个Warning。

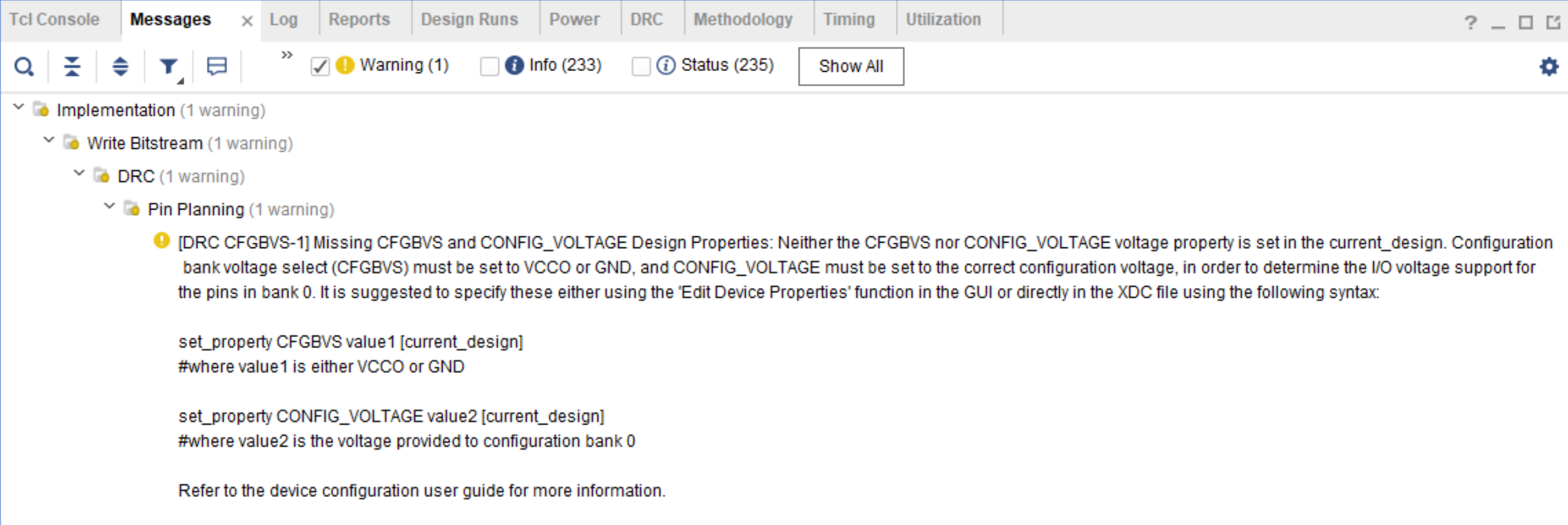
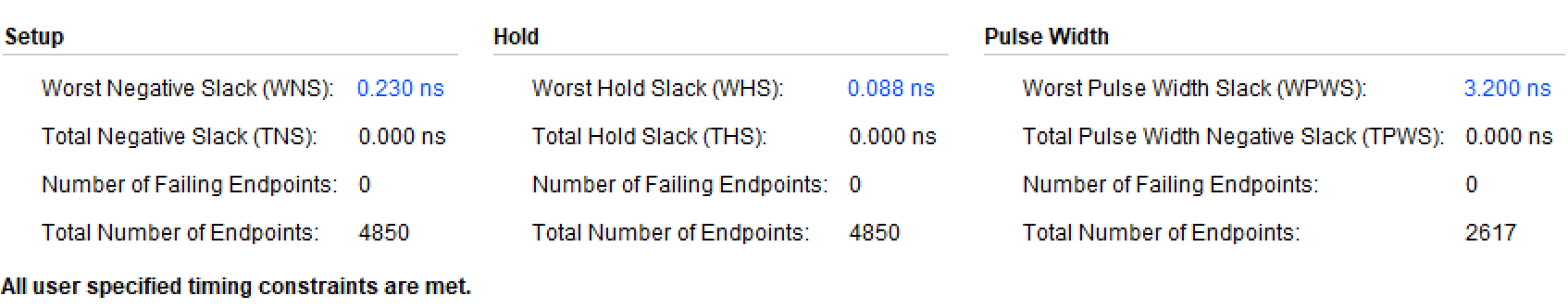


图32 单周期CPU只出现了1个Warning

## 5.2 流水线处理器

### 5.2.1 时序性能

在100MHz的时钟频率下，我们的流水线CPU可以正常运行。由于我们的硬件条件只能产生最高100MHz的时钟，所以对于更高频率的综合，只能通过Vivado软件的时钟约束来分析。我们优化之后的时钟周期为7.4ns，对应主频135MHz。根据Vivado给出的时序分析报告，建立时间裕量为0.230ns（因此时钟主频可以进一步提高到139.5MHz），保持时间裕量0.088ns，符合时钟约束。



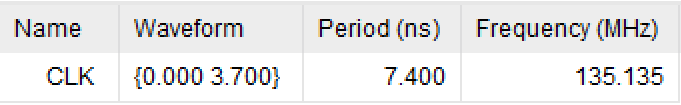
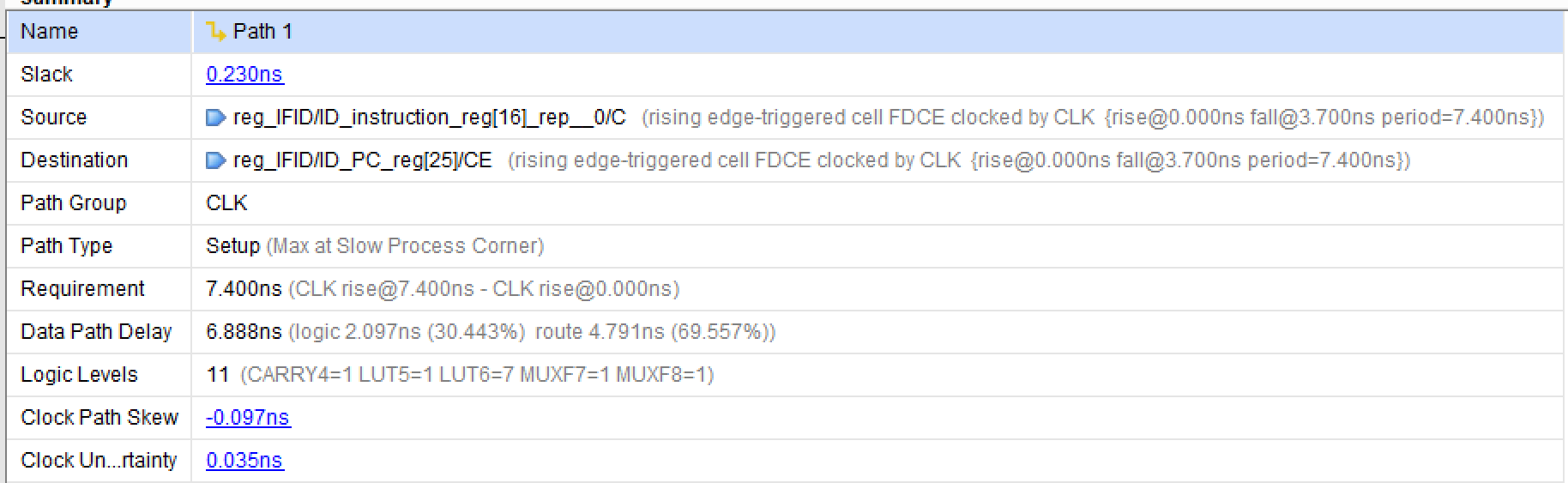
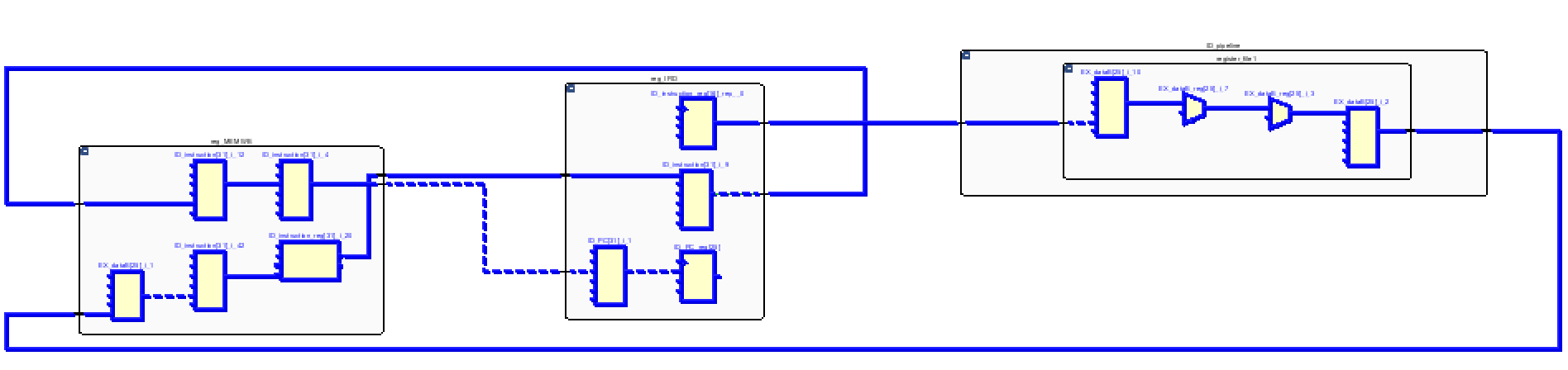


图33 Vivado为流水线CPU给出的时序分析报告

接下来我们查看关键路径综合情况。关键路径是从IF/ID段间寄存器的instruction到PC的通路，中间经过了registerfile，MEM/WB段间寄存器，IF/ID段间寄存器等通路。很难理解为什么会有如此复杂的循环往复的数据通路，我们推测，可能是因为流水线设计中指令相当于并行处理，冒险检测、转发、写回等环节破坏了流水线各段之间的独立性，关键路径数据通路可能经历了不止一次某个流水阶段，而是在CPU中“绕了很多圈”。另外我们注意到，这条路径中的逻辑延时只占30%，导线延时占到了70%，因此影响时序的更关键的因素或许是导线延时（与布线情况、扇出系数等相关）。





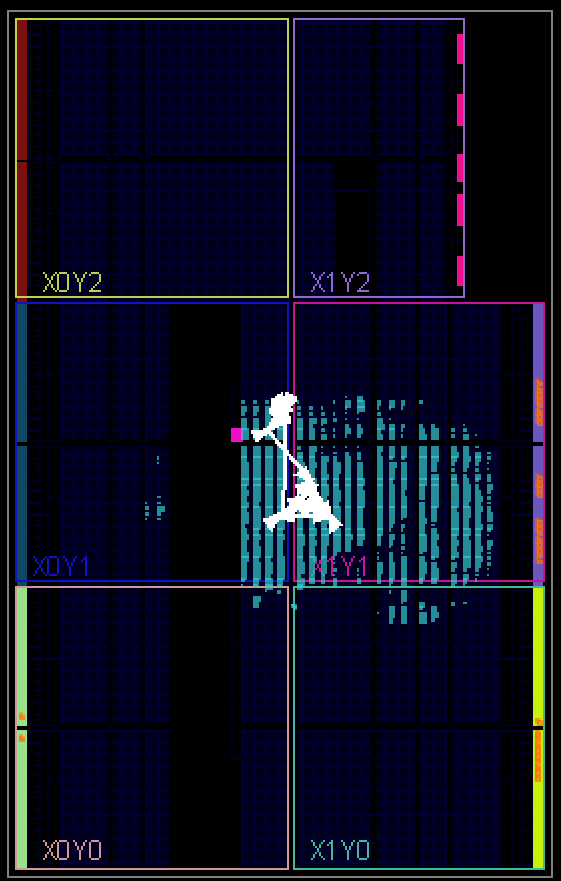
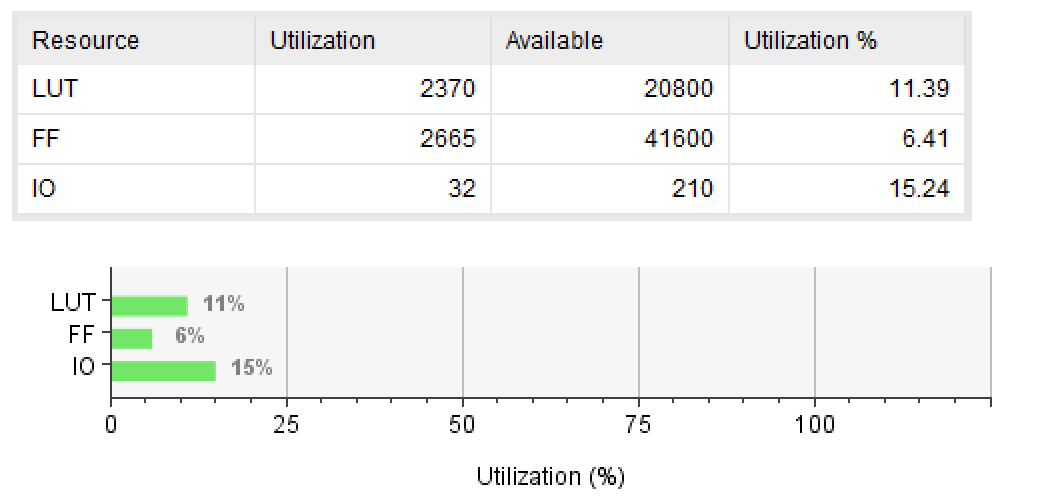


图34 Vivado报告的关键路径信息以及关键路径的Schematic和Device示意

### 5.2.2 资源占用情况



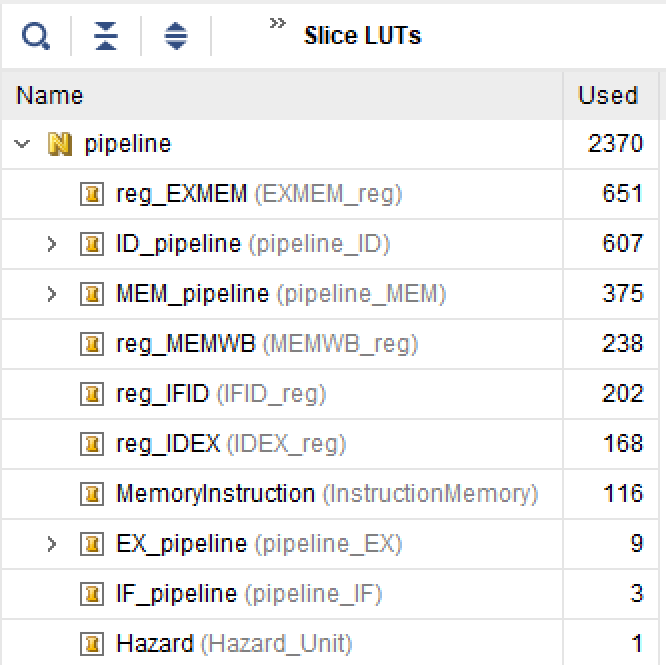
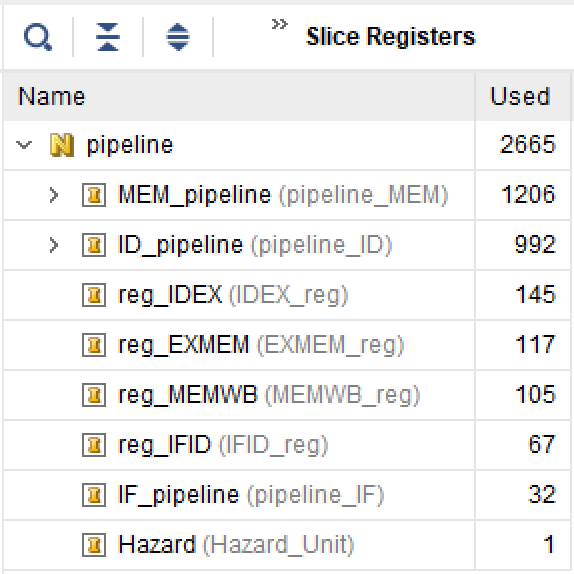
 

图35 Vivado报告的流水线CPU的资源占用情况

左下图为LUT的使用情况，右下图为FF的使用情况

根据Vivado给出的资源占用情况报告，单周期CPU共计使用查找表（LUT）2370个，触发器（FF）2665个，输入输出端口（IO）32个，占资源总量的11.4%，6.4%和15.2%。

对于触发器，MEM\_pipeline阶段由于包含DataMemory、外设和串口，使用了1206个FF；ID\_pipeline阶段由于含有RegisterFile，使用了992个FF。回忆单周期的RF也占用了992个FF，因此这些数据是正常的。和单周期相比，多出来的2665-2231=434个FF全部用于段间寄存器，段间寄存器使用的FF总数恰好为67+145+117+105=434。（注意，单周期中分频用的FF，现在转移到Hazard模块中了，Hazard模块需要这个FF做延时以产生IRQ窄脉冲。）这说明在将单周期改装为流水线的过程中，没有FF的浪费。

对于查找表，流水线的LUT分布和单周期相比存在一些变化，例如ID\_pipeline阶段只用了607个LUT。流水线新增的LUT数目为2370-1979=391，这些逻辑应该主要是用于处理冒险和转发相关的冲突情况。从流水线数据通路图中也可以看到，我们增加了大量的MUX和其他逻辑控制单元。

### 5.2.3 其他信息

观察流水线CPU的功耗信息。流水线CPU的功耗为0.154W，和单周期相比，增加了0.03W（大约24%）。这些功耗的增加全部都在动态功耗方面，静态功耗保持不变。在动态功耗中，时钟功耗和信号功耗有了明显的增长，可能分别是由于有更多的单元模块需要时钟驱动，以及流水线CPU中的数据流和控制信号流更加复杂所导致的。这其实是一件好事，因为动态功耗是硬件电路实际计算所消耗的能量，我们总是“希望”动态功耗在总功耗中占的比重越大越好。流水线CPU以多消耗24%能量的代价，将吞吐率提高了5倍（理论极限是5倍，实际上考虑到有很多stall，大约可以提升4倍左右），说明流水线CPU在功耗方面的表现也更加优秀。

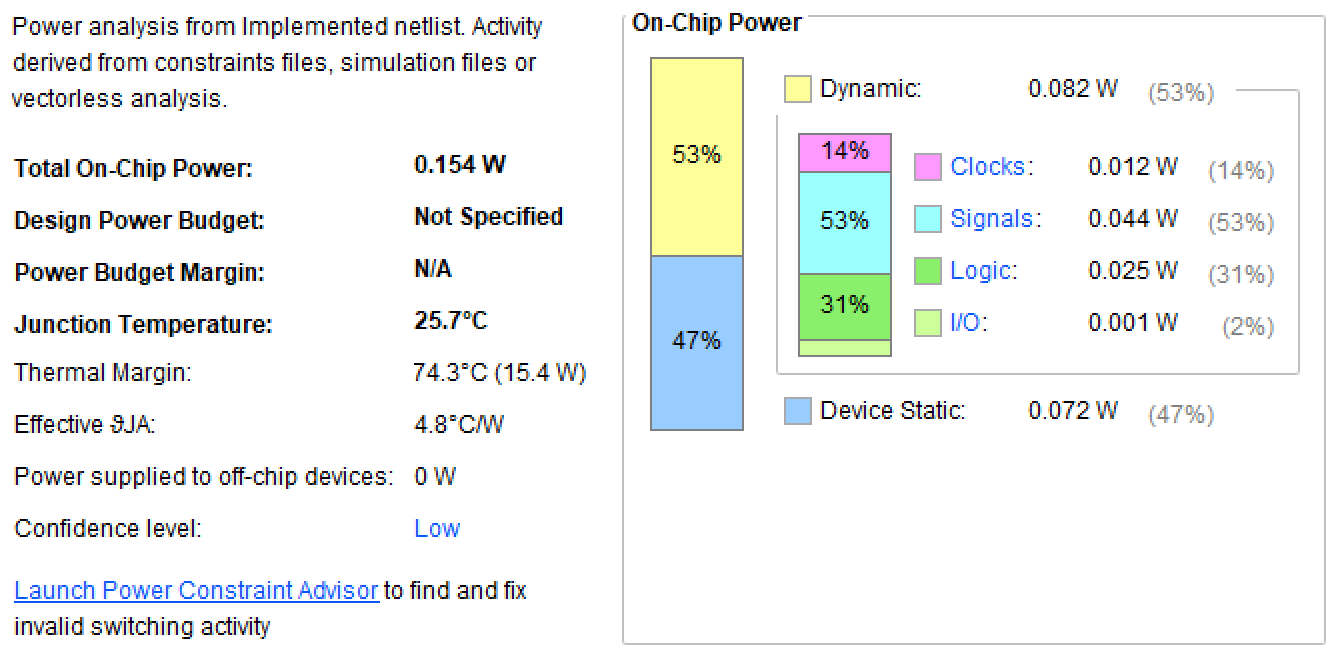


图36 Vivado报告的流水线CPU的功耗情况

同样地，我们的流水线CPU的代码风格经过了仔细的优化处理，使得综合和实现过程中的Warning信息尽可能减少。

# 6 感想体会

**成大立：**在本次实验中，我有三大收获：加深了对CPU工作原理的认识、增强了debug的能力、增强了面对数字逻辑和微处理器的自信心。

上课听讲和自己动手操作完全是两个概念。为了搭建完整的数据通路，我们必须了解每个细节的实现，包括CPU和外设的通讯方式、如何用软件进行译码和显示设备的控制、流水线所有段间寄存器需要存储的变量、流水线中所有可能的冒险和转发情况等等。这些细节的实现方式都必须对照指令集和实验参考书逐字逐句地分析，这期间我更加实际地认识到MIPS处理器的工作方式。

自己设计testbench，并通过大量仿真找出设计中的漏洞也是我第一次面对的任务。如何在Modelsim中挑选合适的波形，顺藤摸瓜找到问题的根源，需要的不仅是技巧，更是耐心。单周期的仿真调试是容易的，光标所在位置对应于同一条指令；但在流水线中，同一时间上不同硬件资源中执行的是不同的指令，再加上stall的存在，搞清楚时序关系是调试过程中的重要环节。

当然，仿真通过和在FPGA上运行程序还相差很远。例如，我们原先设计的在任何时刻都能写入（不仅仅在时钟上升沿）的寄存器堆，在仿真过程中是可以正确运行的，但在开发板上就不能正确地读写了，它内部存储的数据会发生意想不到的不稳定现象。在Vivado环境中调试的过程，以及寻找关键路径，试图看懂Vivado当中每个LUT都在做什么的经历，也让我对FPGA的工作原理、数字电路的硬件实现有了更生动的理解。Vivado虽然很慢，但真的是相当智能，能够将关键路径对应到Schematic和Device，并且Netlist甚至可以对应到源代码，这给我们优化时序性能带来了很大方便。我们发现了耗时较大的路径，往往出在串行的位置上（例如计数器分频，或者ALU移位的串行），但有时无用的变量，或者较大的扇出，也会增大数据通路的延时。

总体而言，能够手写一个简单的CPU，并跑通最大公约数程序，并不是一件容易的事情。李懋坤老师说，当年学习微处理器，还要通过DOS命令编程，编写的CPU也只有4位。这可以说是很大的进步了。

**李云飞：**只有写过了数字逻辑处理器大作业才对理论课上学到的微处理器知识有了真正的理解。但是这个大作业不仅仅是“复现”理论课上学到的数据通路，而是需要在理解的基础上对学过的微处理器设计进行一些修改。

具体来说，这次的微处理器最麻烦的事情就是加入了IO以及中断处理。虽然外设只是简单的串口、定时器，但是刚开始接触的时候，为了能把它们和微处理器整体的数据通路融为一体，还费了一番心思。在加数据通路的时候我对于微处理器的通用性理解加深了。不仅要能保证某一项功能实现，同时要保证原来所有的功能都能正常执行。实际写的时候，我果然在一开始把数据存储器一块搞砸了，没有考虑到新加的外设与原来的代码的冲突。

调试的过程比编写更加有收获。出现问题的时候分模块调试，“二分查找”快速定位bug所处位置等操作在这次作业中都被频繁使用。不过有一点做得不够好的是，写完代码没有先做单元测试再连起来整体测试。从整体开始调试固然看起来工作量小一些（如果一把能过自然是最好的），但事实上bug很多也很隐蔽，面对一个略庞大的系统深入每一个细节去调试非常耗费脑力和耐心。虽然最后也调出来了，但是过程比较痛苦，发现的错误基本都是某个模块内部的问题而不是模块连接的问题。试想如果每个模块写完以后，都能写一个比较完善的单元测试，这对于调试当前版本以及加新功能以后测试原来的功能是不是还正常都会很有帮助。

关于布线我觉得这是一件非常神奇的事情，vivado软件能够帮助我们做布线，但是具体是怎么实现的我并不清楚。在优化流水线主频时，发现时钟限制比较松的时候，device里看到的布局也很松散。然而，仅仅将时钟约束改得小一些，布局立刻就会变得紧凑很多，关键路径的时间随之减小。如果课程中能稍微讲一些vivado等工具是怎么布线的背景知识，我觉得会更好。

**汤宸：**这次实验是我上大学以来最复杂的实验之一，为其花费了很长的时间和精力。通过这次实验，我对流水线的工作原理，具体每条指令的路径等，有了更进一步地认识，也更加熟练地对课本内容等进行了掌握。同时，流水线的工作和单周期有很多相似之处，通过对流水线处理器的编写，我也对单周期的工作原理进行了进一步的梳理。

在一开始设计时，我对流水线各个级需要用到哪些控制信号，冒险和转发单元在该级的表现等等知识点上掌握得并不熟练，因此在整体框架搭建、以及各个段间寄存器要传什么值上，我花了很长时间进行思考。最后我打算先写一个粗糙的版本，然后再打补丁。

结合大立写的冒险与转发初版，我进行了第一次仿真调试。在第一次仿真时，就出现了非常致命的错误，Modelsim提醒我，我的设计中有一个环形振荡器。搜索资料告诉我，在出问题时刻每一个有变化的变量都有可能是产生环形振荡器的罪魁祸首。我找了好久，发现是在我的ID段与大立写的冒险单元之间，关于reg-beq冒险的编写问题。经过修改后解决了这个问题。

初版的编写主要是依据课本知识进行设计，但是实际操作中，我发现出现了很多课本上没讲的东西。比如第一次调试时，我就发现寄存器堆不能先写后读，于是我将其改成了组合逻辑，仿真通过。在结合串口、中断等单元进行第二次仿真调试时，出现了jal-jr冒险，即jal后，PC的值还未存入31号寄存器，紧接着就来了jr指令要调用31号寄存器堆的值，这是一个通往IF段的Forward，课本上没有。事后我想了一下，jal后跳到jr $ra指令，又跳回jal的下一条执行，那不是等同于这两条指令不存在吗？乍一看和老师在课件上讲转发编写时，特地要规避零号寄存器的说法一样没有道理，但是为了正常进入求最大公约数的程序进行执行，就需要这个操作。

实际烧板子时的调试过程更让人心力憔悴，我将时钟绑定在按钮上，进行手按时钟产生上升沿进行模拟调试，把我想看的信号绑定在led灯上。首先我先看转发单元和冒险单元产生的控制信号，及PC中间实际用到的八位的运行情况是怎样的，对比仿真一点一点调试。终于发现在某个jr $ra指令PC的跳转出现了异常。正如上文调试部分所写，我发现从寄存器堆里读出来的PC值不对，但是写PC时，所有信号都是对的，但为什么寄存器里存的数就不对呢？后来才知道，这是组合逻辑的不稳定造成的。

最后优化操作时，我虽然出力比较少，但是我也跟着学了一些通过vivado观看critical path及优化的方法，也有所收获。

总的来说，这次实验虽然很累，但是让人收获很大，不仅更加熟练地掌握了课本知识，而且对于如何调试和优化，也进行了进一步地了解。

# 7 文件清单

鉴于代码文件普遍长度较长，直接粘贴在实验报告中不利于阅读，这里仅给出文件清单，所有源代码都在附件中一并提交。

表3 32位MIPS处理器文件清单列表

|  |  |  |
| --- | --- | --- |
| 文件所属层级 | 文件名 | 文件含义 |
| \Compiler | \Compiler.py | 编译器的python脚本 |
| \exit.asm | 异常处理程序（汇编代码） |
| \exit\_instructions.v | 经过编译器编译的异常处理程序机器码 |
| \gcd\_main.asm | 最大公约数计算程序（汇编代码） |
| \gcd\_instructions.v | 经过编译器编译的最大公约数计算程序机器码 |
| \InterruptionCode.asm | 中断处理程序（汇编代码） |
| \rom\_instructions.v | 经过编译器编译的中断处理程序机器码 |
| \SingleCycle | \tb\ALUtest.v | ALU的仿真测试环境 |
| \tb\test\_cpu.v | 单周期CPU的仿真测试环境 |
| \ALU.v | ALU的RTL实现 |
| \baudgen.v | 串口的时钟分频模块 |
| \Control.v | 单周期控制信号产生模块 |
| \CPU.v | 单周期CPU顶层模块 |
| \cpu\_clk.v | 单周期CPU时钟分频模块（二分频） |
| \DataMemory.v | 单周期CPU数据存储器 |
| \InstructionMemory.v | 单周期CPU指令存储器 |
| \Peripheral.v | 单周期CPU外设（定时器、数码管和LED） |
| \receiver1.v | 串口的接收模块 |
| \RegisterFile.v | 单周期CPU寄存器堆 |
| \sender.v | 串口的发送模块 |
| \UART.v | 单周期CPU的串口的顶层模块 |
| \singleCycle\_ego.xdc | 单周期CPU的管脚约束文件 |
| \Pipeline | \ALU.v | ALU的RTL实现 |
| \baudgen.v | 流水线CPU的串口分频模块 |
| \Control.v | 流水线CPU的控制信号产生单元 |
| \DataMemory.v | 流水线CPU的数据存储器 |
| \EXMEM\_reg.v | 流水线CPU的EX/MEM段间寄存器 |
| \Forward\_Unit.v | 流水线CPU的转发单元 |
| \Pipeline | \Hazard\_Unit.v | 流水线CPU的冒险检测单元 |
| \IDEX\_reg.v | 流水线CPU的ID/EX段间寄存器 |
| \IFID\_reg.v | 流水线CPU的IF/ID段间寄存器 |
| \InstructionMemory.v | 流水线CPU的指令存储器 |
| \MEMWB\_reg.v | 流水线CPU的MEM/WB段间寄存器 |
| \Peripheral.v | 流水线CPU的外设（定时器、数码管和LED） |
| \pipeline.v | 流水线CPU的顶层设计模块 |
| \pipeline\_EX.v | 流水线CPU的EX流水阶段 |
| \pipeline\_ID.v | 流水线CPU的ID流水阶段 |
| \pipeline\_IF.v | 流水线CPU的IF流水阶段 |
| \pipeline\_MEM.v | 流水线CPU的MEM流水阶段 |
| \pipeline\_WB.v | 流水线CPU的WB流水阶段 |
| \receiver1.v | 串口的接收模块 |
| \RegisterFile.v | 流水线CPU的寄存器堆 |
| \sender.v | 串口的发送模块 |
| \UART.v | 流水线CPU的串口的顶层模块 |
| \pipeline.xdc | 流水线CPU的管脚约束文件 |