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

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

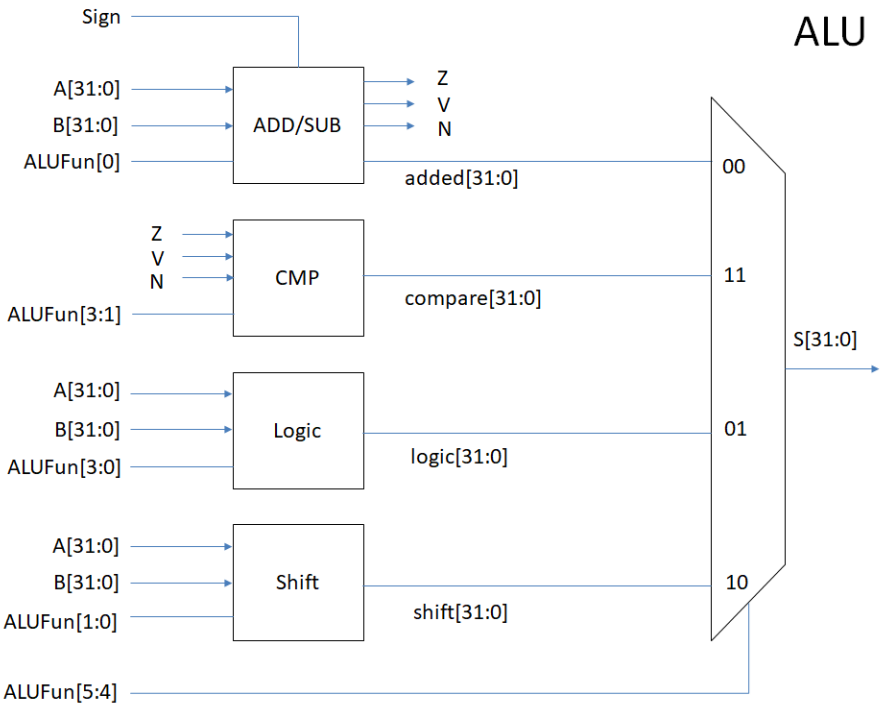


图 1 ALU 功能框图

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。

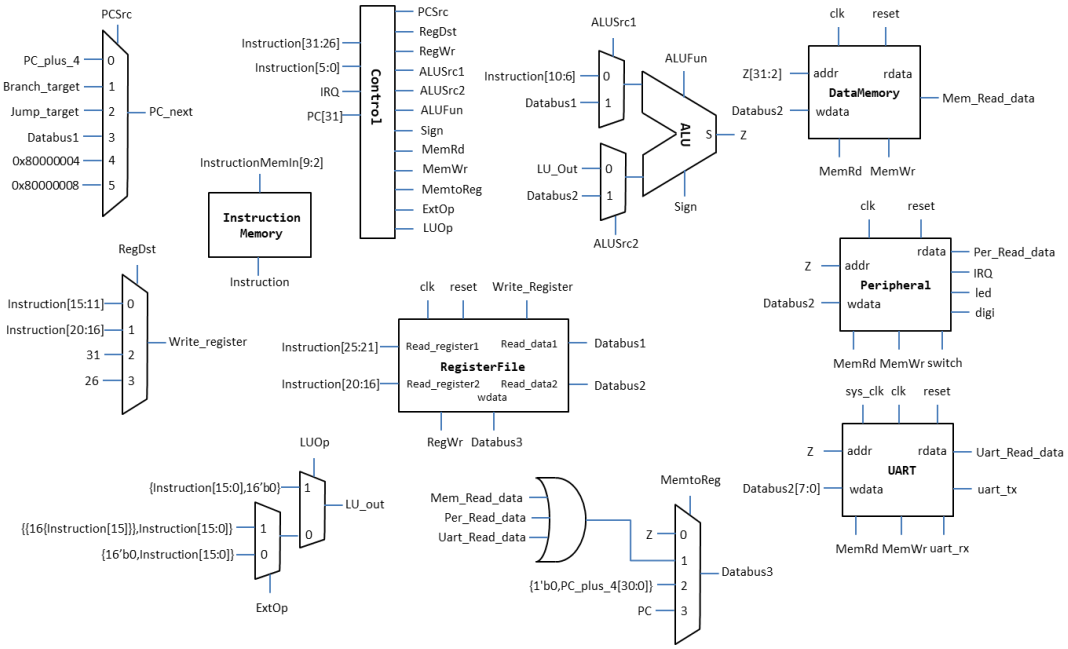


图 2 单周期 CPU 整体框图

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。

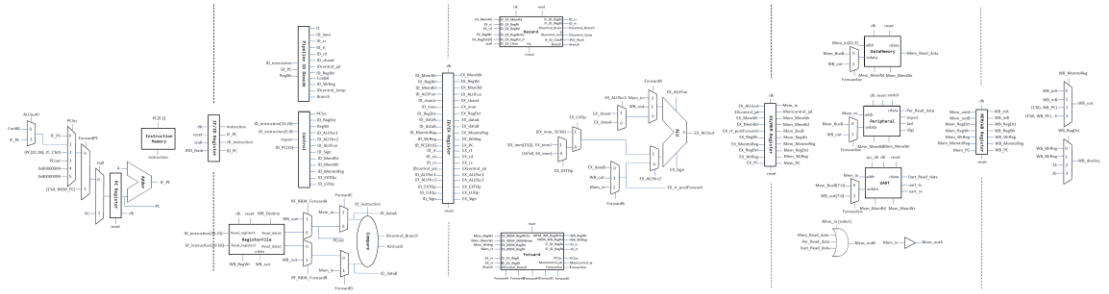


图 3 流水线 CPU 整体框图

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$\frac{1}{2^n}$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。

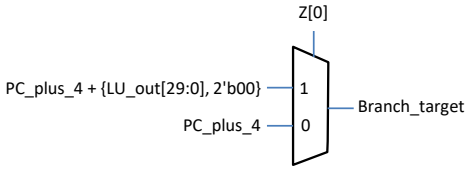


图 4 Branch_target 计算方法

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。

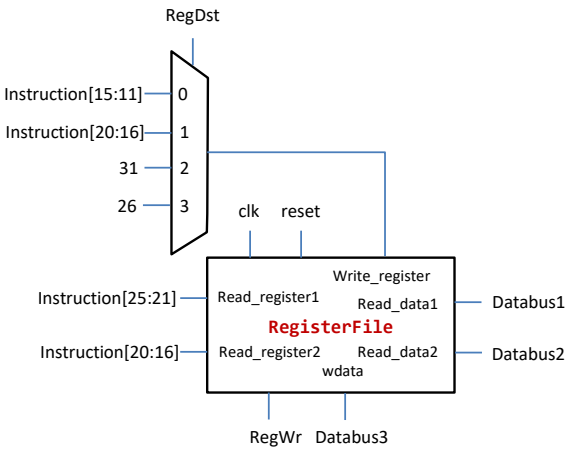


图 5 寄存器堆设计

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。

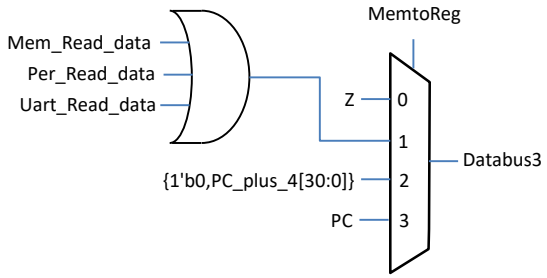


图 6 写回寄存器多路选择

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。

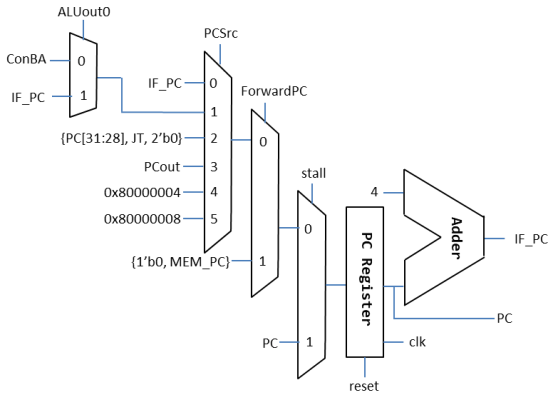


图 7 流水线 CPU 对 PC 寄存器的修改

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。

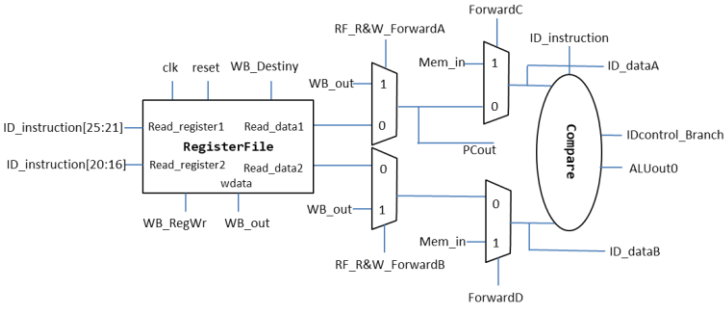


图 8 流水线 CPU 对 RF 寄存器堆的修改

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。

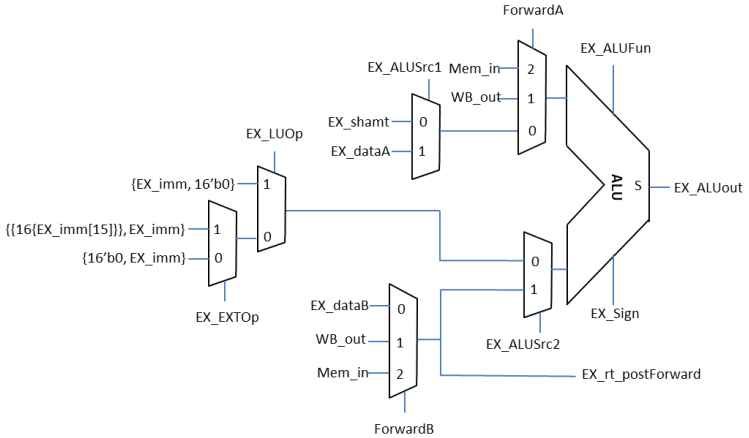


图 9 流水线 CPU 对 ALU 的修改

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。

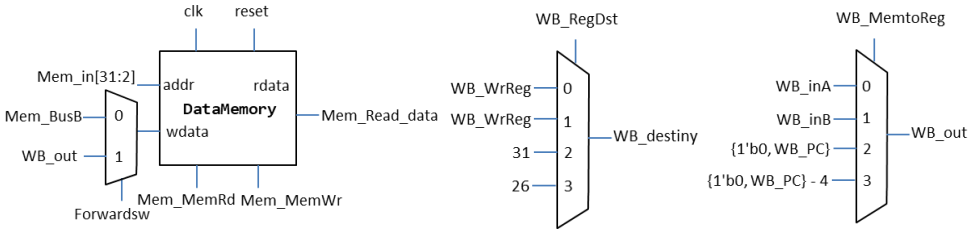


图 10 流水线 CPU 对 MEM 和 WB 的修改

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。

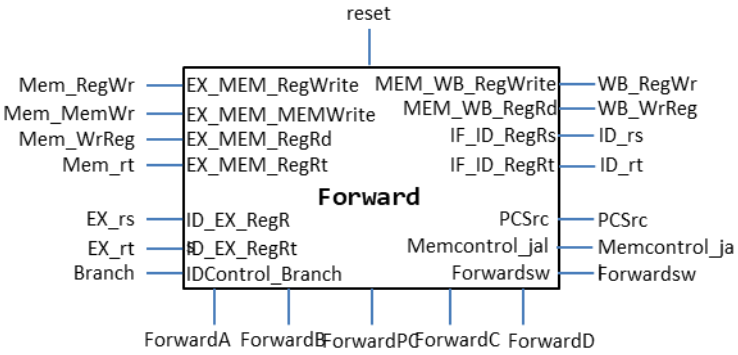


图 11 转发单元输入输出框图

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。

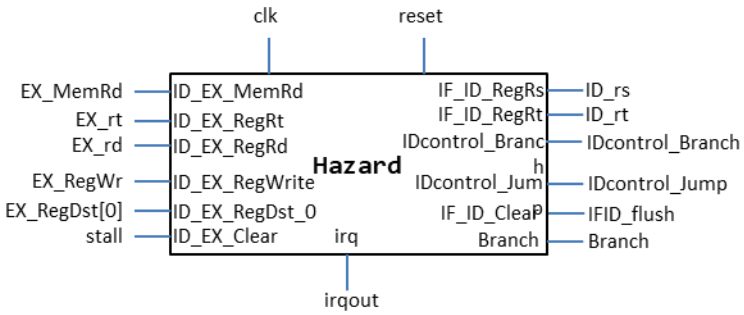


图 12 冒险检测单元输入输出框图

3 仿真结果

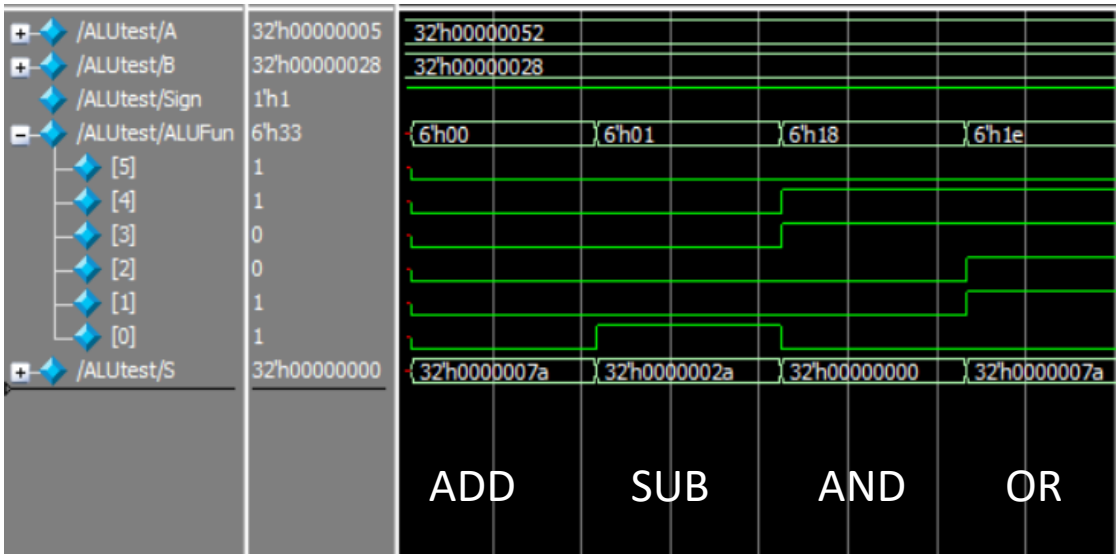
3.1 ALU

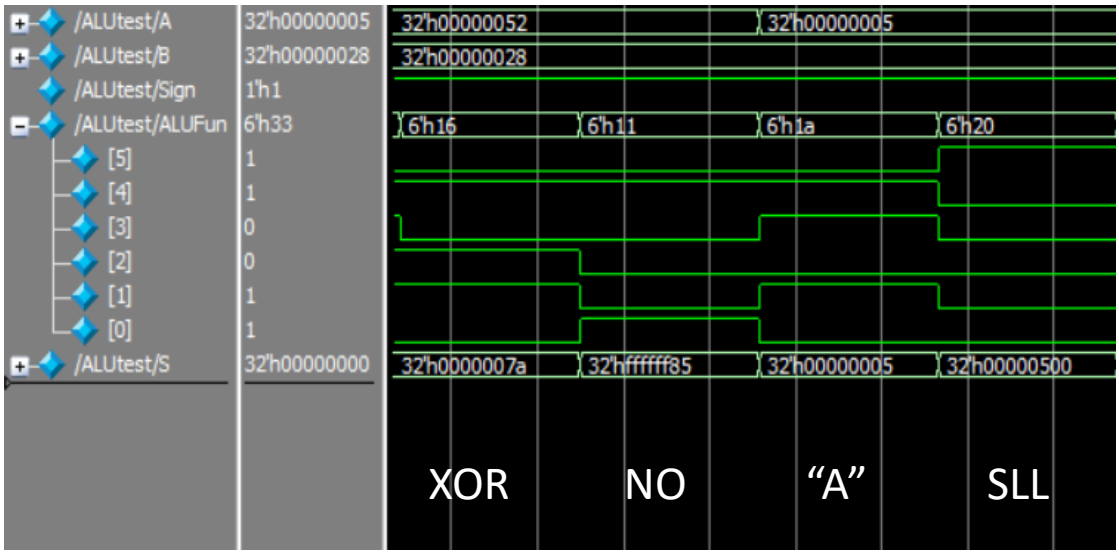
为了调试 ALU 的性能，我们设计了测试代码 ALUtest.v 如下：

```
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。





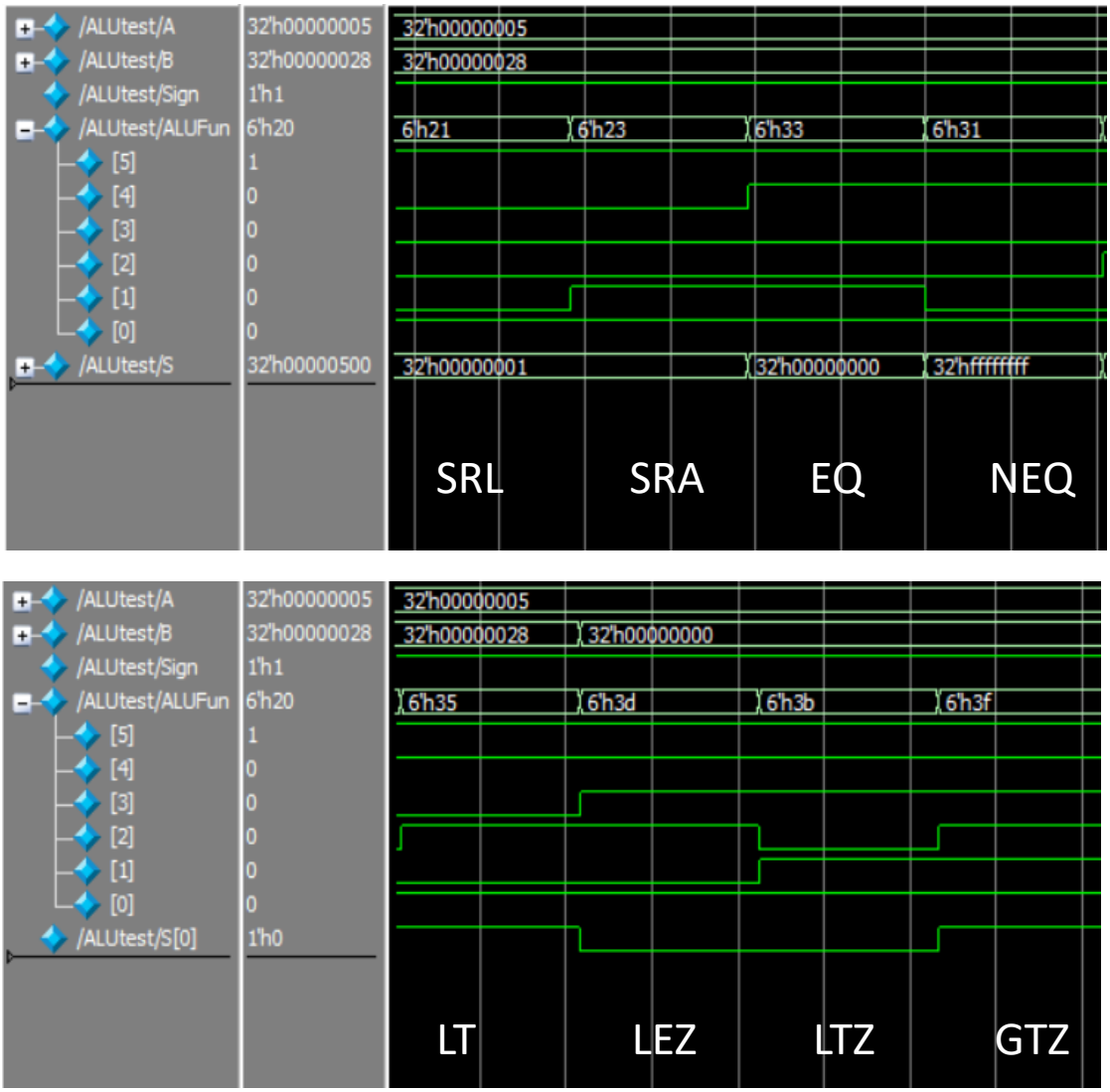


图 12 ALU 功能仿真结果

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。

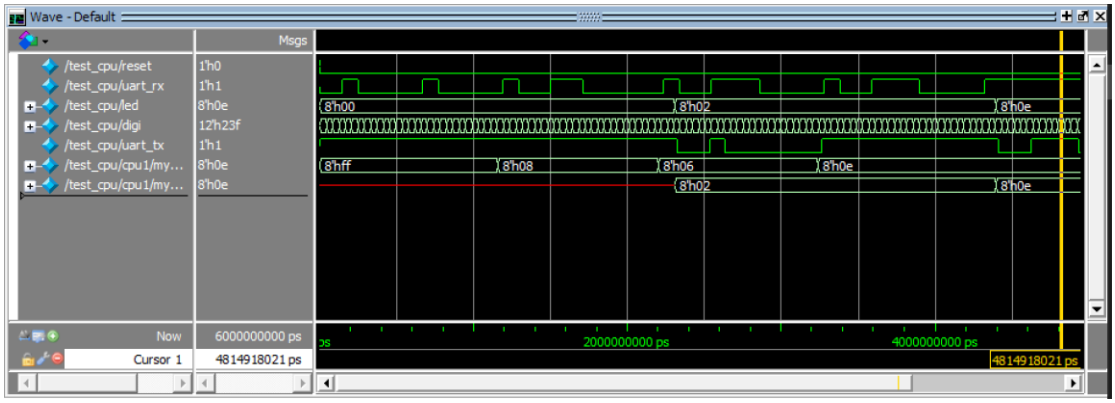


图 13 单周期宏观仿真结果

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：

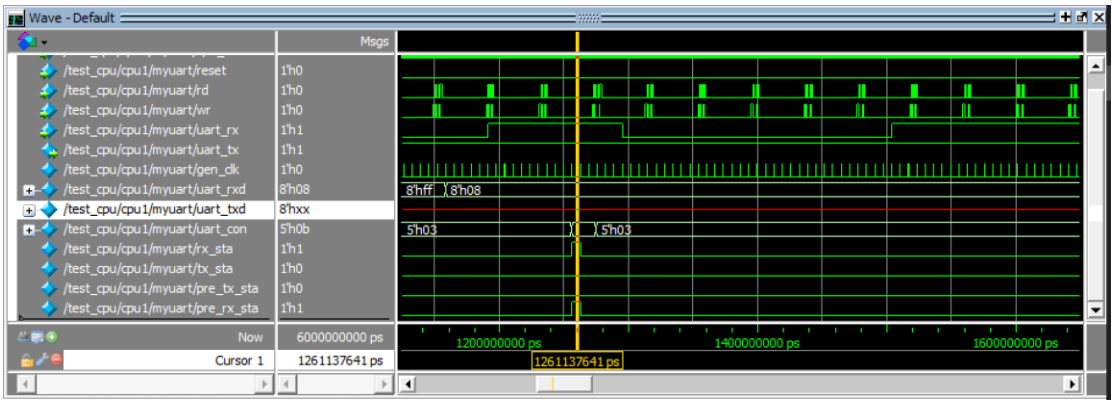


图 14 单周期 CPU 串口内部仿真结果

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：

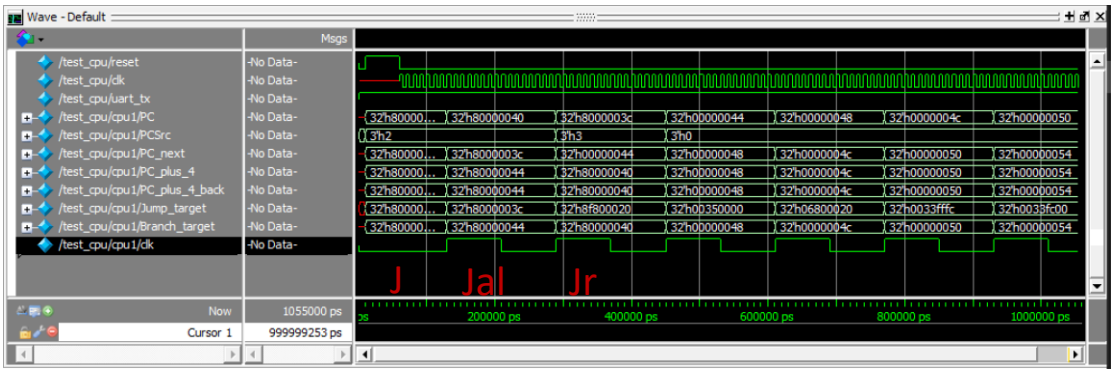


图 15 单周期 CPU 的 PC 跳转仿真结果

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。

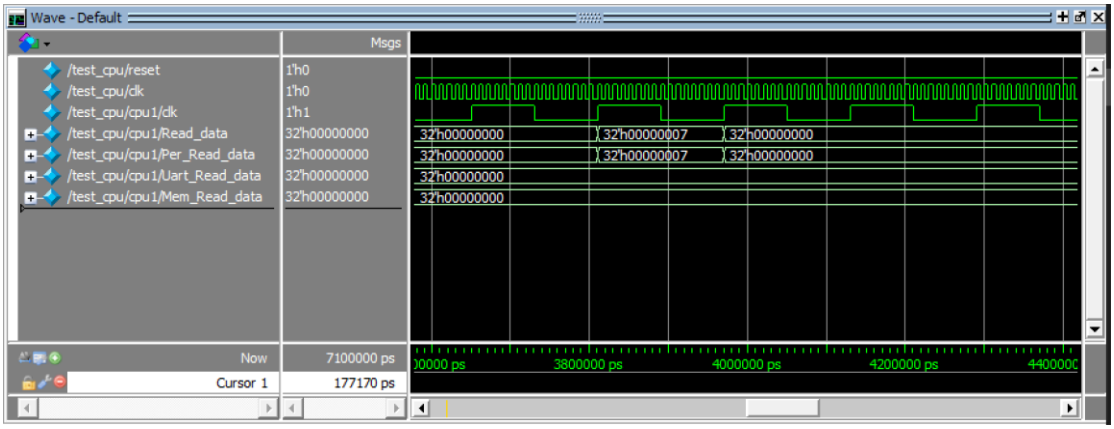


图 16 单周期 CPU 从外设读取数据仿真结果

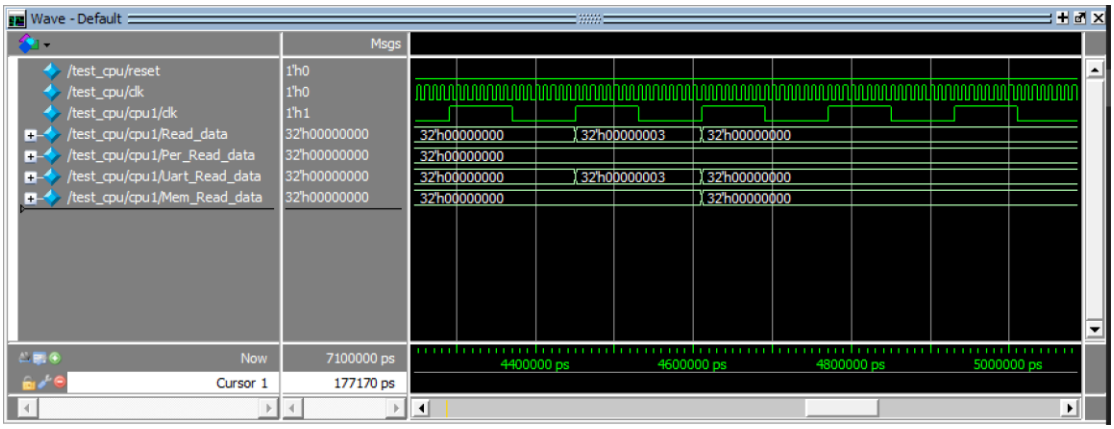


图 17 单周期 CPU 从串口读取数据仿真结果

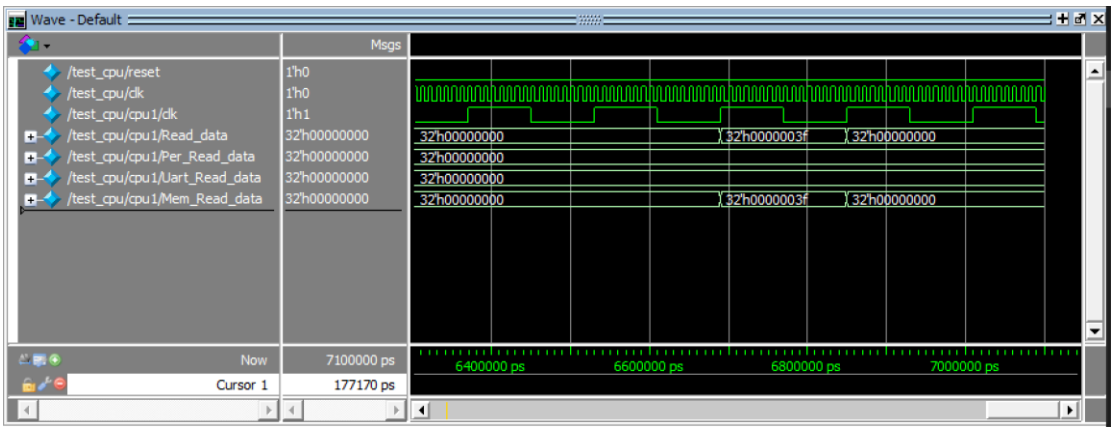


图 18 单周期 CPU 从数据存储器读取数据仿真结果

- **中断处理情况：**

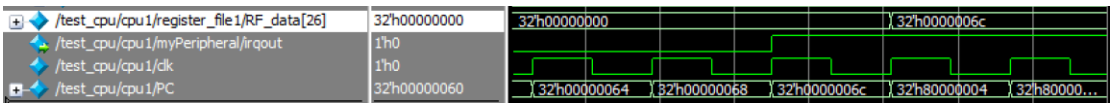


图 19 单周期 CPU 处理中断信号仿真结果

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。

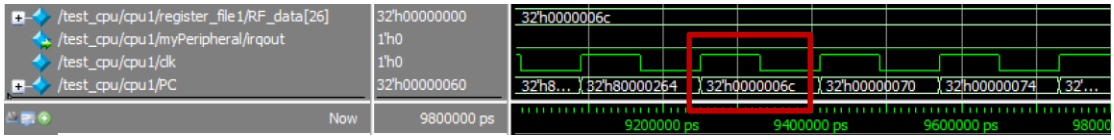


图 20 单周期 CPU 中断程序结束后返回仿真结果

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。

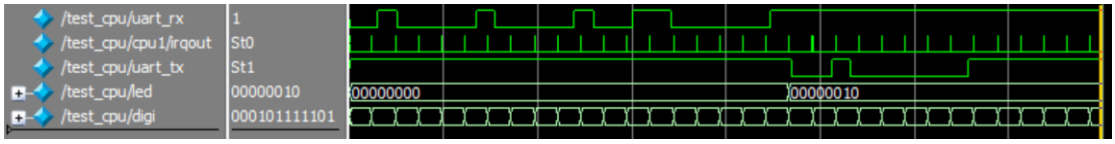


图 22 流水线宏观仿真结果

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。

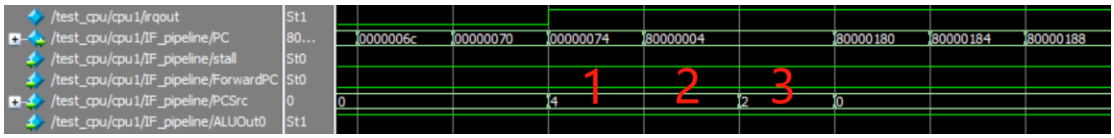


图 24 流水线中断到来时 PC 跳转仿真结果

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。

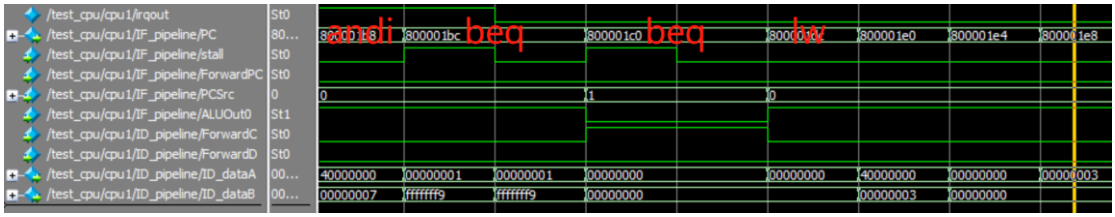


图 25 流水线 beq 指令与 PC 跳转仿真结果

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。

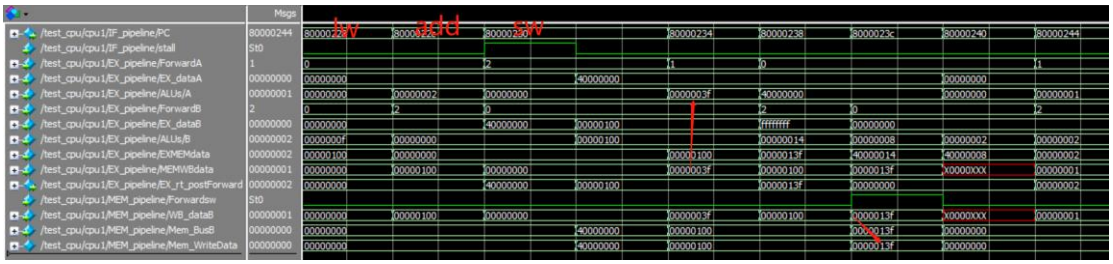


图 26 流水线 EX 和 MEM 阶段典型仿真结果

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。

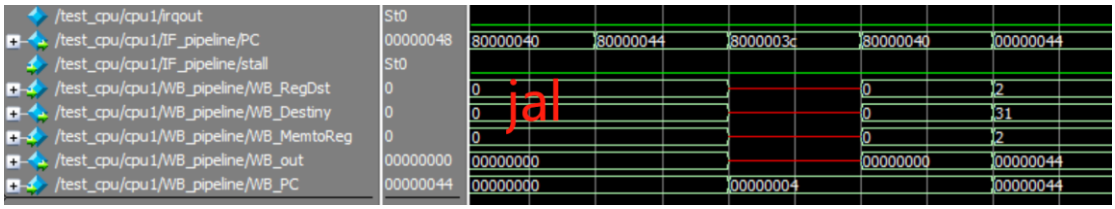


图 27 流水线 WB 阶段处理 jal 指令仿真结果

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。

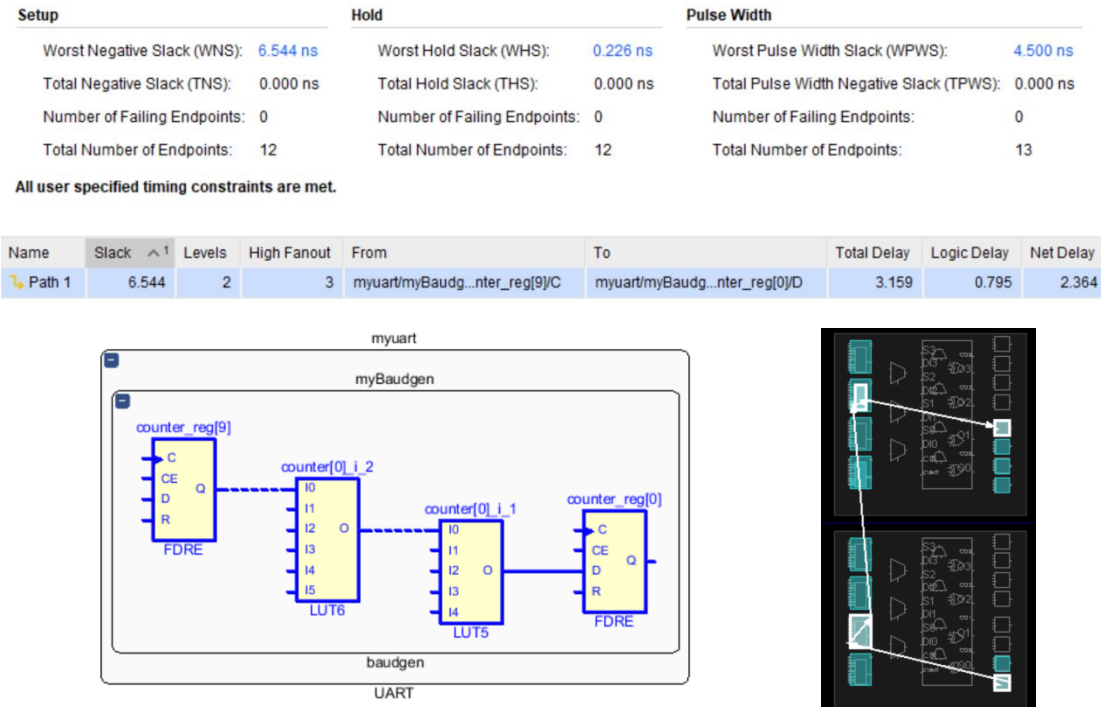


图 29 Vivado 为单周期 CPU 给出的时序分析报告、关键路径及其 Schematic 和 Device 示意

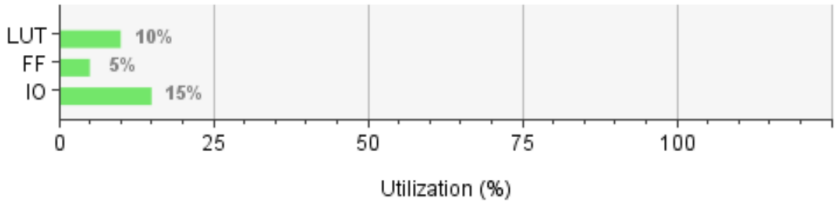
5.1.2 资源占用情况

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。

Resource	Utilization	Available	Utilization %
LUT	1979	20800	9.51
FF	2231	41600	5.36
IO	32	210	15.24



CPU	2231
data_memory1 (DataMemory)	1024
register_file1 (RegisterFile)	992
myPeripheral (Peripheral)	107
myuart (UART)	75
myReceiver (receiver1)	28
mySender (sender)	21
Leaf Cells (15)	15
myBaudgen (baudgen)	11
Leaf Cells (32)	32
cpu_clk (cpu_clk)	1

register_file1 (RegisterFile)	1465
data_memory1 (DataMemory)	262
myPeripheral (Peripheral)	156
myuart (UART)	70
myReceiver (receiver1)	30
mySender (sender)	26
myBaudgen (baudgen)	12
Leaf Cells (2)	2
Leaf Cells (16)	16
alu1 (ALU)	9
adds (Adder)	9
cpu_clk (cpu_clk)	1

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。

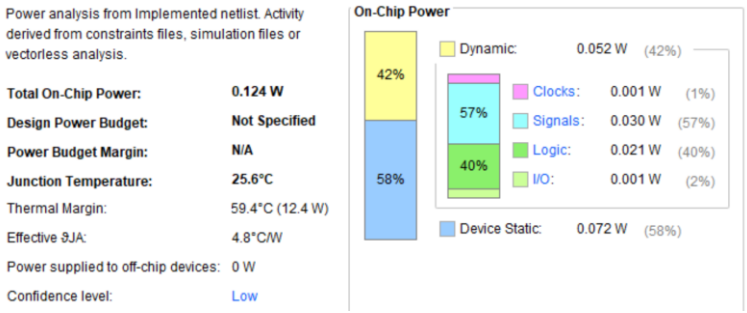


图 31 Vivado 为单周期 CPU 给出的功耗评估报告

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。

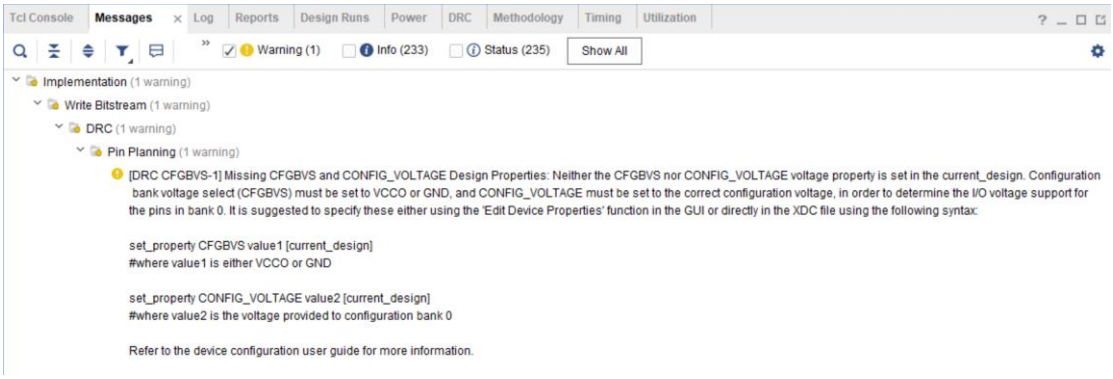


图 32 单周期 CPU 只出现了 1 个 Warning

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。

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.230 ns	Worst Hold Slack (WHS):	0.088 ns	Worst Pulse Width Slack (WPWS):	3.200 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	4850	Total Number of Endpoints:	4850	Total Number of Endpoints:	2617

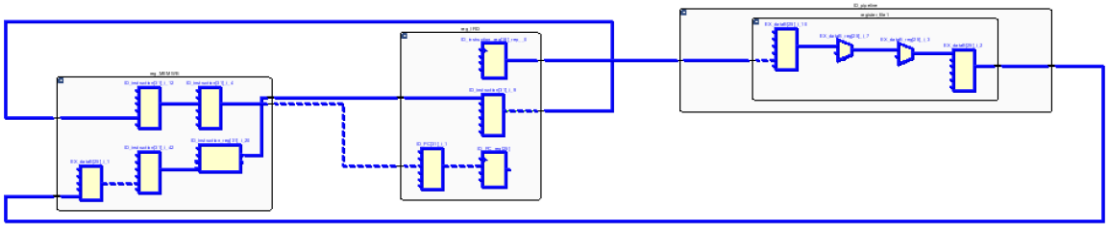
All user specified timing constraints are met.

Name	Waveform	Period (ns)	Frequency (MHz)
CLK	{0.000 3.700}	7.400	135.135

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）。

Name	Path 1
Slack	0.230ns
Source	reg_IFID/ID_instruction_reg[16]_rep__0/C (rising edge-triggered cell FDCE clocked by CLK {rise@0.000ns fall@3.700ns period=7.400ns})
Destination	reg_IFID/ID_PC_reg[25]/CE (rising edge-triggered cell FDCE clocked by CLK {rise@0.000ns fall@3.700ns period=7.400ns})
Path Group	CLK
Path Type	Setup (Max at Slow Process Corner)
Requirement	7.400ns (CLK rise@7.400ns - CLK rise@0.000ns)
Data Path Delay	6.888ns (logic 2.097ns (30.443%) route 4.791ns (69.557%))
Logic Levels	11 (CARRY4=1 LUT5=1 LUT6=7 MUXF7=1 MUXF8=1)
Clock Path Skew	-0.097ns
Clock Un...rtainty	0.035ns



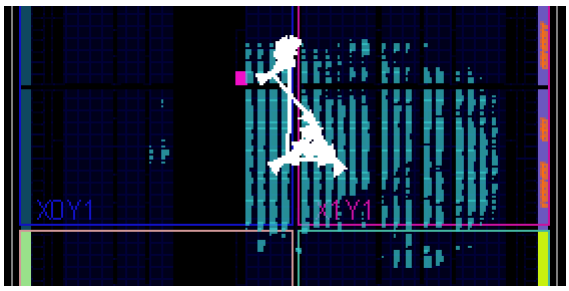
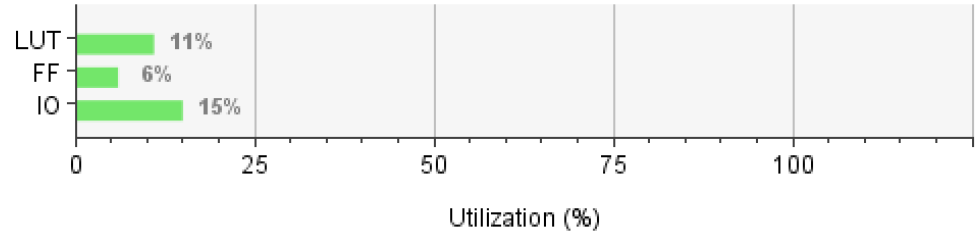
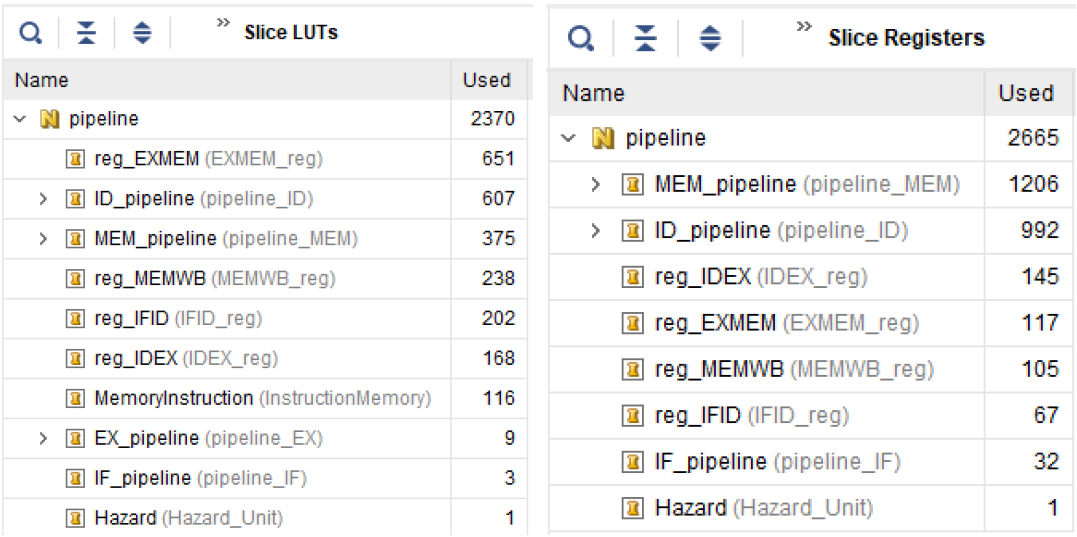


图 34 Vivado 报告的关键路径信息以及关键路径的 Schematic 和 Device 示意

5.2.2 资源占用情况

Resource	Utilization	Available	Utilization %
LUT	2370	20800	11.39
FF	2665	41600	6.41
IO	32	210	15.24





Slice LUTs

Name	Used
pipeline	2370
reg_EXMEM (EXMEM_reg)	651
ID_pipeline (pipeline_ID)	607
MEM_pipeline (pipeline_MEM)	375
reg_MEMWB (MEMWB_reg)	238
reg_IFID (IFID_reg)	202
reg_IDEX (IDEX_reg)	168
MemoryInstruction (InstructionMemory)	116
EX_pipeline (pipeline_EX)	9
IF_pipeline (pipeline_IF)	3
Hazard (Hazard_Unit)	1

Slice Registers

Name	Used
pipeline	2665
MEM_pipeline (pipeline_MEM)	1206
ID_pipeline (pipeline_ID)	992
reg_IDEX (IDEX_reg)	145
reg_EXMEM (EXMEM_reg)	117
reg_MEMWB (MEMWB_reg)	105
reg_IFID (IFID_reg)	67
IF_pipeline (pipeline_IF)	32
Hazard (Hazard_Unit)	1

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。

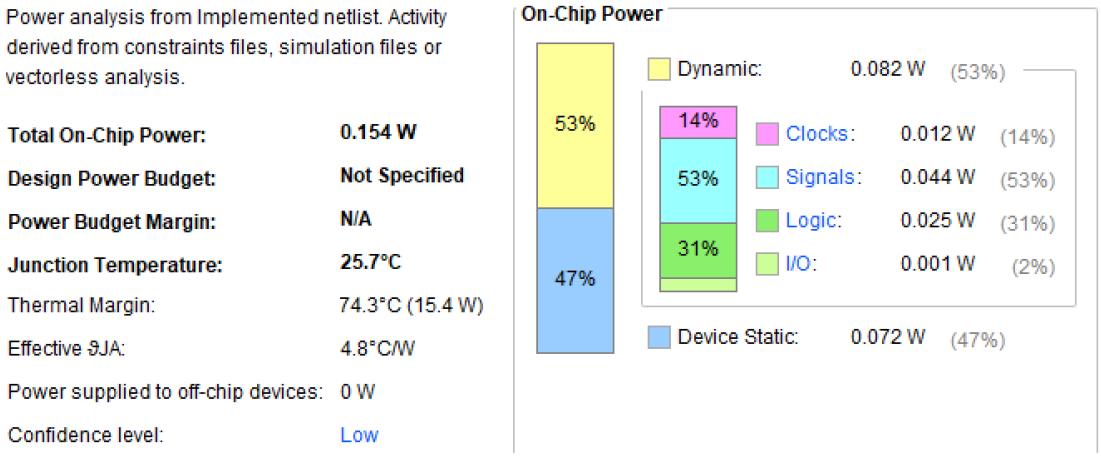


图 36 Vivado 报告的流水线 CPU 的功耗情况

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