

数字逻辑与处理器基础实验

32 位 MIPS 处理器设计实验报告

王晗

(2013011076)

July 22, 2015

Date Performed: July 15, 2015

Partners: 耿天毅 (2012011119)

陈志杰 withdrawn

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。

2 设计方案

2.1 总体结构

由于这次实验涉及的功能较多，我们将完整的 CPU 分成多个模块。指令存储器、寄存器堆、控制器、ALU 控制器、ALU、数据存储器、UART 等功能单元均在单独的 Module 中实现。其中指令存储器、寄存器堆、控制器、ALU 控制器、ALU 等单元在 Single Cycle Core 中实例化，作为单周期处理器的核心；数据存储器、UART 和定时器、LED、七段数码管、开关在 Peripheral 中实现，作为处理器的外设。处理器核心和外设在顶层模块中实例化，互相通信。

单周期 CPU 模块的结构关系如 Figure 1所示：

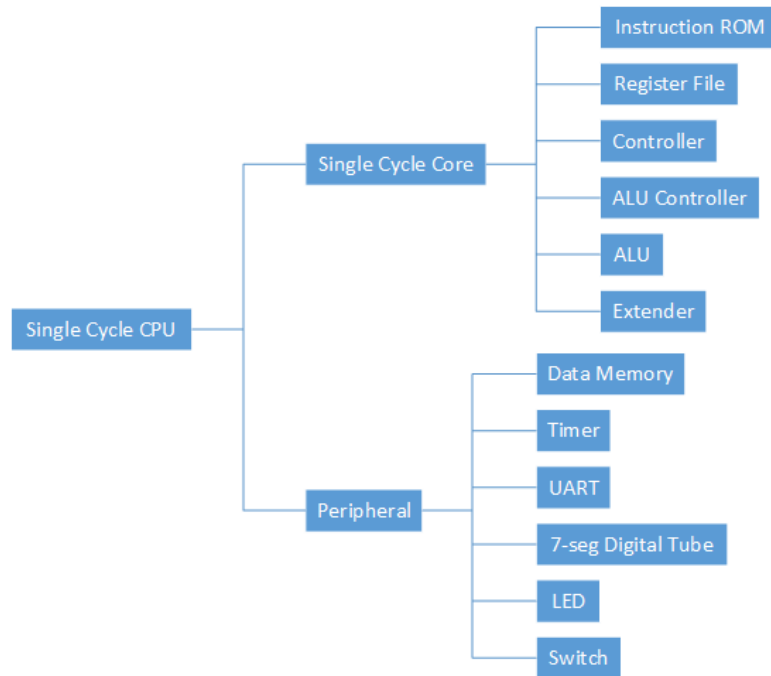


Figure 1: 单周期处理器结构

对于流水线 CPU，我们还在 Pipeline Core 中加入了流水线寄存器、冒险检测单元、数据转发单元：

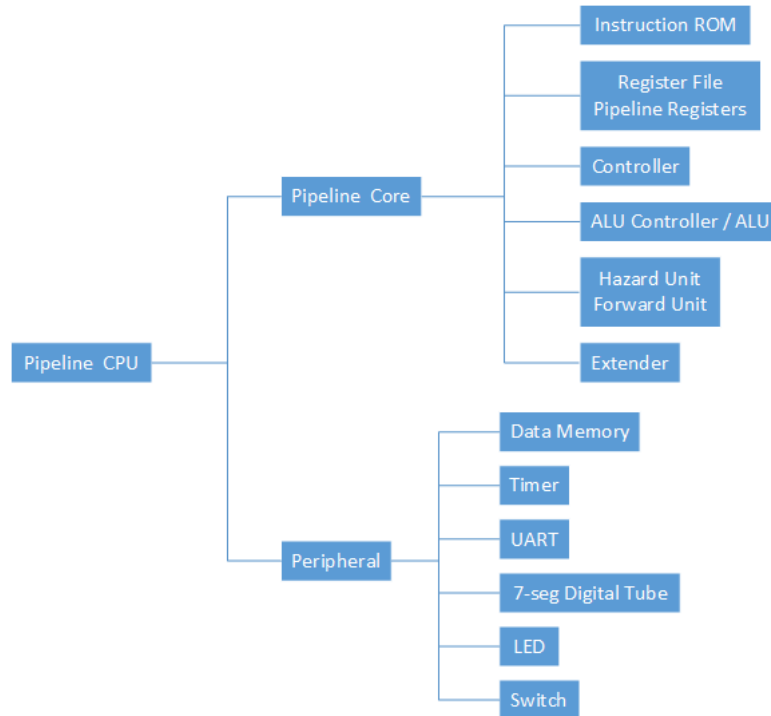


Figure 2: 流水线处理器结构

2.2 ALU ¹

ALU 模块的结构如图所示，输入两个操作数 A、B 和控制信号 ALUFun、Signed，在 ARITH 子模块中做加减法运算，CMP 子模块根据 ARITH 模块的输出进行比较判断，LOGIC 和 SHIFT 模块分别进行逻辑运算和移位运算，ALUFun 的最高两位用于控制多路选择器的输出。

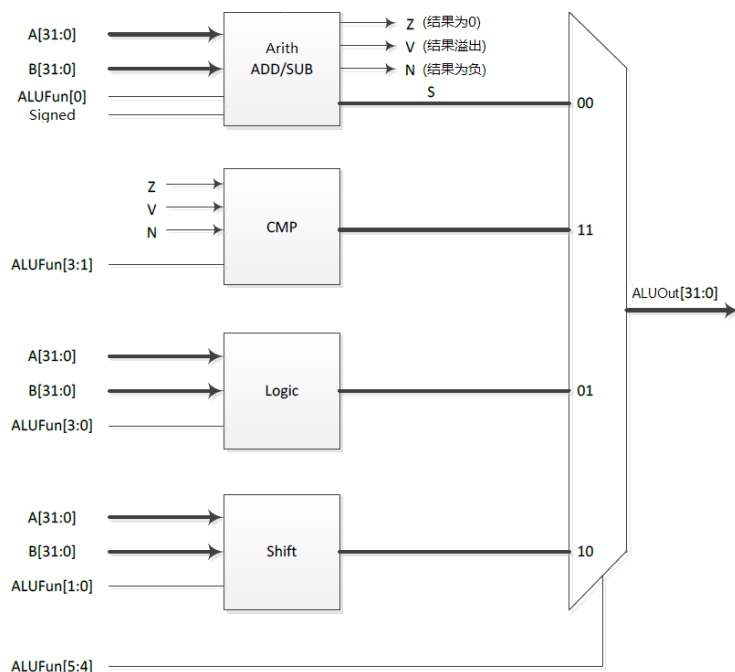


Figure 3: ALU 结构

ARITH 模块 ARITH 模块中包括减法和加法两个模块，加法模块直接通过 + 号运算，减法模块先对第二个操作数取补码，再调用加法模块做加法运算。Overflow 和 Negative 信号的产生是 ALU 中的难点：

ADD		
	Overflow	Negative
Signed	pos+pos=neg neg+neg=pos	neg+neg, pos+pos: MSB of A otherwise: MSB of S
Unsigned	big+big big+small=small small+big=small	0

Figure 4: ADD 中的 Overflow 和 Negative

¹ 原作者：陈志杰；修改：王晗

其中 pos 为正数，neg 为负数，big 为 MSB=1 的无符号数，small 为 MSB=0 的无符号数。

SUB		
	Overflow	Negative
Signed	pos-neg=neg neg-pos=pos	pos-pos, neg-neg: MSB of S otherwise: MSB of A
Unsigned	small-big big-big, small-small: MSB of S	big-small: 0 small-big: 1 otherwise: MSB of S

Figure 5: SUB 中的 Overflow 和 Negative

图中的缩写含义同上。

CMP 模块 CMP 模块直接根据 ARITH 模块产生的 Zero, Overflow, Negative 进行关系判断。

LOGIC 模块 LOGIC 模块直接根据 ALUFun[3:0] 指定的逻辑运算进行运算。

SHIFT 模块 将移位操作拆分为 16 位移位、8 位移位、4 位移位、2 位移位、1 位移位，分别用 Shamt 的每一个 bit 位控制，组合产生最后的运算结果。

2.3 寄存器堆、指令存储器、数据存储器 and 外设²

寄存器堆 直接采用 reg [31:0] RF_DATA[31:1] 实现，注意 RF_DATA[0] 不存在，读取时直接返回 0。

指令存储器 将机器码以十六进制文本的形式存放在 .rom 文件中，使用 \$readmemh 系统任务初始化一个大小为 256words 的只读存储器。

数据存储器 由于数据存储器容量设计为 256words，因此寻址时只根据 address[9:2] 寻址。

另外，0x40000000 开始的地址用于外设编址，因此数据存储器不对 0x40000000 开始的地址进行读写操作。

其他外设 定时器、LED、Switch 参考老师提供的样例代码直接在 Peripheral.v 中实现，UART 使用春季学期第四次实验的 UART 发送和接收模块，将发送模块中 Tx_Status 的定义取反，即 1 表示发送端忙碌。UART 的控制同样在 Peripheral.v 中实现，当 0x40000018 写入要发送的数据时，串口控制器自动产生一个发送使能信号。

²作者：王晗

- a. 当没有中断或异常时，根据 PCSrc 取出下一个 PC 地址，并读取指令；否则跳转至中断或异常服务程序。
- b. RegDst 控制写入的目标寄存器：R 型指令写入 Rd；I 型指令写入 Rt；jal/jalr 指令写入 \$31；中断或异常写入 \$26。
- c. ExtOp 控制立即数的符号扩展或无符号扩展。
- d. LuOp 控制立即数左移 16 位，用于 lui 指令。
- e. ALUSrc1 用于选择 RegReadData1 或 Shamt 作为 ALU 的第一个操作数，ALUSrc2 用于选择 RegReadData2 或立即数作为 ALU 的第二个操作数。
- f. MemRead、MemWrite 控制外设和数据存储器的读写。
- g. MemToReg 用于选择写入寄存器的数据来源：运算指令来源为 ALU 输出，lw 指令来源为 MemReadData，jal/jalr/中断/异常来源为 PC+4。
- h. PC 的最高位为监督位，不用于寻址。当 PC[31]=1 时为内核态，禁止中断请求；当 PC[31]=0 时为正常态，允许中断请求。Reset、异常、中断可以将 PC[31] 置 1，jr、jalr 可以将 PC[31] 清零。

2.6 流水线数据通路⁵

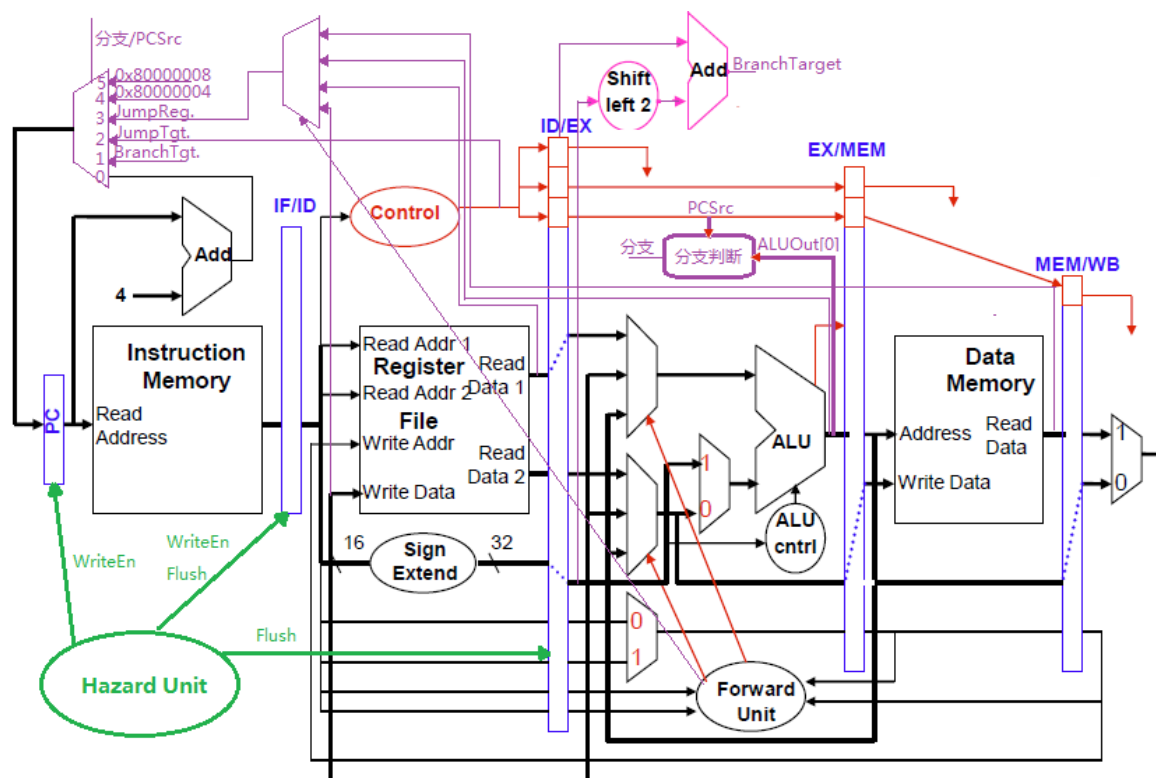


Figure 7: 流水线数据通路

⁵作者：王晗

- a. 在单周期数据通路的基础上添加流水线寄存器。
- b. 添加数据转发单元，从 EX/MEM 或 MEM/WB 寄存器转发数据，解决 ALU 运算中的数据关联问题。
- c. 完善数据转发单元，从 ALUOut、MemReadData、RegWriteData 转发数据，解决 jr 指令的数据关联问题。
- d. 添加冒险检测单元，对存在 load-use 竞争的指令阻塞一个周期，禁止 PC 寄存器写入、清空 ID/EX 寄存器。
- e. 完善冒险检测单元，在 EX 阶段进行分支时，需要清空 IF/ID 和 ID/EX 寄存器。
- f. 完善冒险检测单元，在 ID 阶段进行跳转时，需要清空 IF/ID 寄存器。

2.7 汇编代码⁶

2.8 汇编器⁷

3 关键代码及文件清单

3.1 寄存器堆的读写⁸

寄存器堆在同一周期对同一个寄存器先写再读不会引起数据冒险，为了实现这个目的，先对读取的寄存器地址做判断，如果读取地址为 0，则返回 0；如果读取地址和写入地址相同，则返回写入的值；其他情况返回寄存器堆中相应寄存器的值。

```

                                regfile.v
assign data1 = (addr1==5'b0) ? 32'b0 :
               (addr1==addr3) ? data3 :
               RF_DATA[addr1];

```

3.2 数据转发单元⁹

对于 ALU 运算的指令，输入数据来自寄存器堆时，可能存在数据冒险。为了解决这个问题，可以从 EX/MEM 或 MEM/WB 寄存器转发数据到 ALU 输入端。从 EX/MEM 转发的条件是前一条指令需要写非 \$0 寄存器，且写入地址和欲读取地址相等；从 MEM/WB 转发的条件是前两条指令需要写非 \$0 寄存器，欲读取的地址和写入地址相等，并且不能从前 EX/MEM 就近转发。

```

                                ForwardUnit.v
if (EX_MEM_RegWrite == 1'b1 &&
    EX_MEM_RegWriteAddr != 5'h00 &&

```

⁶作者：耿天毅

⁷作者：耿天毅

⁸作者：王晗

⁹作者：王晗

```

        EX_MEM_RegWriteAddr == ID_EX_InstRs)
            ForwardA = 2'b10;
5   else if (MEM_WB_RegWrite == 1'b1 &&
            MEM_WB_RegWriteAddr != 5'h00 &&
            MEM_WB_RegWriteAddr == ID_EX_InstRs)
                ForwardA = 2'b01;
        else
10            ForwardA = 2'b00;
        //ForwardB is similar to this.

```

对于 jr 指令，同样存在数据冒险。由于跳转操作在 ID 阶段完成，如果检测到 jr 的目标寄存器和 ID_EX 段写入寄存器相同，需要从 ALUOut 转发数据；如果目标寄存器和 EX_MEM 段写入寄存器相同，需要从 MemReadData 转发数据；如果目标寄存器和 MEM_WB 段写入寄存器相同，则需要从 RegWriteData 转发数据。

```

                                ForwardUnit.v
if (ID_PCSrc == 3'b011 && IF_ID_InstRs == ID_EX_RegWriteAddr
    && ID_EX_RegWriteAddr != 0 && ID_EX_RegWrite)
    ForwardJr = 2'b01;
else if (ID_PCSrc == 3'b011 &&
5   IF_ID_InstRs != ID_EX_RegWriteAddr &&
    IF_ID_InstRs == EX_MEM_RegWriteAddr &&
    EX_MEM_RegWrite &&
    EX_MEM_RegWriteAddr != 0)
    ForwardJr = 2'b10;
10  else if (ID_PCSrc == 3'b011 &&
    IF_ID_InstRs != ID_EX_RegWriteAddr &&
    IF_ID_InstRs != EX_MEM_RegWriteAddr &&
    IF_ID_InstRs == MEM_WB_RegWriteAddr &&
    MEM_WB_RegWriteAddr != 0 &&
15  MEM_WB_RegWrite)
    ForwardJr = 2'b11;
    else
        ForwardJr = 2'b00;

```

3.3 冒险检测单元¹⁰

对于 load 指令，需要阻塞一个周期，即在禁止 PC 寄存器、IF_ID 流水线寄存器写入，并清空 ID_EX 流水线寄存器；

对于跳转指令，在 ID 段完成正确的跳转，因此 IF 段的指令是错误的，需要清空 IF_ID 流水线寄存器；

¹⁰作者：王晗

对于分支指令，在 EX 段完成分支判断，因此如果成功跳转，IF、ID 段的指令均是错误的，需要清空 IF_ID 和 ID_EX 流水线寄存器。

HazardUnit.v

//参见/pipeline/HazardUnit.v

在清空寄存器时，为了保证 jal/jalr/中断/异常写入寄存器堆的 PC+4 保持正确，我们采用了如下策略：

- a. 如果 ID_EX 寄存器清空，表明一条指令被抛弃，需要将 ID_EX_PC_plus_4 的值减 4；
- b. 如果 IF_ID 寄存器清空且 ID_EX 不清空，表明一条指令被抛弃，需要将 IF_ID_PC_plus_4 的值减 4；
- c. 如果 IF_ID 和 ID_EX 寄存器同时清空，表明两条指令被抛弃，需要将 IF_ID_PC_plus_4 的值减 8。

3.4 UART 输入¹¹

3.5 最大公约数¹²

3.6 数码管译码¹³

3.7 UART 输出¹⁴

3.8 文件清单

Directory	File	Description
/assembler	/MIPSAssembler.py	Python 编写的汇编器
	/MIPSCode_singlecycle.asm	UART 收发、求最大公约数、数码管译码
	/MIPSCode_singlecycle.rom	汇编结果
	/MIPSCode_pipeline.asm	在单周期基础上延长定时器周期
	/MIPSCode_pipeline.rom	汇编结果
/ALU	/ALU.v	ALU 顶层模块
	/ADD.v	加法器
	/SUB.v	减法器
	/ARITH.v	算术运算单元
	/CMP.v	比较运算单元
	/LOGIC.v	逻辑运算单元
	/SHIFT.v	移位运算单元

¹¹作者：耿天毅

¹²作者：耿天毅

¹³作者：耿天毅

¹⁴作者：耿天毅

Directory	File	Description
/single_cycle	/ALU/*	ALU, 同上
	/Controller/Controller.v	控制单元
	/Controller/ALUController.v	ALU 控制单元
	/InstructionROM/rom.v	指令存储器
	/Peripheral/Peripheral.v	外设顶层模块
	/Peripheral/DataMem.v	数据存储器
	/Peripheral/digitube_scan.v	扫描方式工作的数码管
	/Peripheral/uart*	UART 相关模块
	/RegFile/regfile.v	寄存器堆
	/clk_gen.v	分频模块
	/single_cycle_core.v	单周期 CPU 内核
	/single_cycle.v	单周期 CPU 顶层模块
	/single_cycle_tb.v	单周期 CPU TestBench
/pipeline	/ALU/*	ALU, 同上
	/Controller/Controller.v	控制单元
	/Controller/ALUController.v	ALU 控制单元
	/InstructionROM/rom.v	指令存储器
	/Peripheral/*	外设, 同上
	/RegFile/regfile.v	寄存器堆
	/RegFile/PC_reg.v	PC 寄存器
	/RegFile/IF_ID_reg.v	IF_ID 流水线寄存器
	/RegFile/ID_EX_reg.v	ID_EX 流水线寄存器
	/RegFile/EX_MEM_reg.v	EX_MEM 流水线寄存器
	/RegFile/MEM_WB_reg.v	MEM_WB 流水线寄存器
	/ForwardUnit.v	数据转发单元
	/HazardUnit.v	冒险检测单元
	/pipeline_core.v	流水线 CPU 内核
	/pipeline.v	流水线 CPU 顶层模块
	/pipeline_tb.v	流水线 CPU TestBench

4 仿真结果及分析

4.1 单周期 LED、Switch 外设

如 Figure 8所示, 从 switch 读入 0x4a, 再将 0x4a 输出至 led。

4.3 单周期串口

如 Figure 12所示，从串口接收数据，加 1，再通过串口返回。

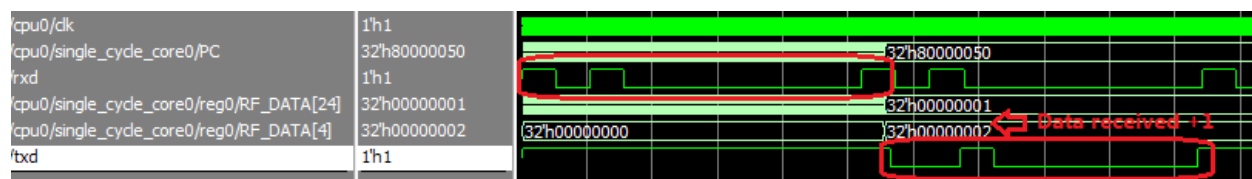


Figure 12: 单周期串口

4.4 单周期完整程序

我们现在 MARS 模拟器上进行了仿真（如 Figure 13所示），\$s1=125、\$s2=25 为两个输入参数，\$s3=25 为输出结果，最大公约数计算正确。

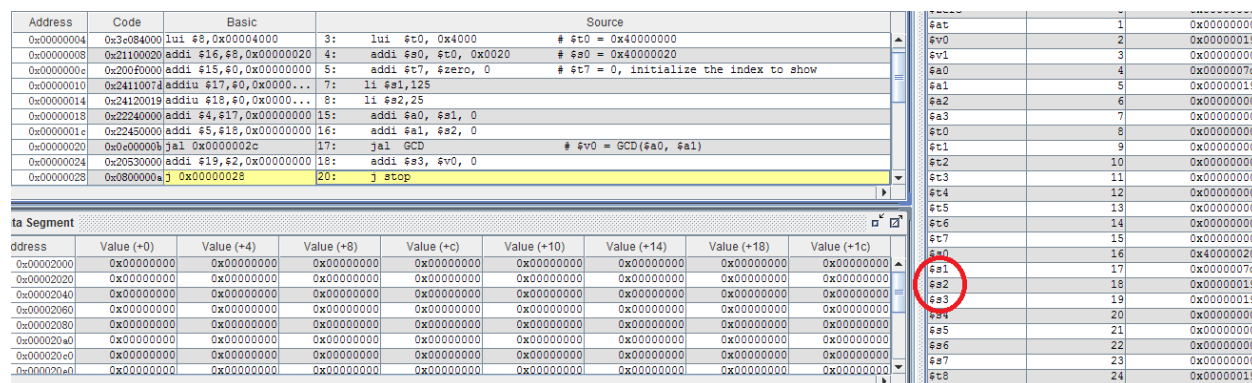


Figure 13: MARS 仿真器仿真结果

如 Figure 14所示，从 UART 读入两个操作数，分别存放至 \$s1、\$s2，最大公约数运算结果存放至 \$s3，通过 LED 显示，同时经 UART 输出。

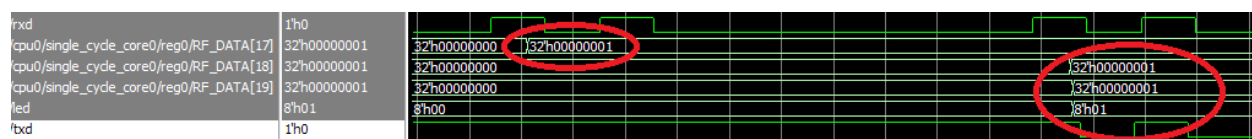


Figure 14: 单周期完整程序运行结果

4.5 流水线数据通路

如 Figure 15所示，将 \$1 置 1，5 个时钟周期后 1 被写入寄存器堆；两个空指令 (nop) 后，将 \$1 加 1 并存入 \$2，这时 \$1 的写入和读取发生在同一个时钟周期内。

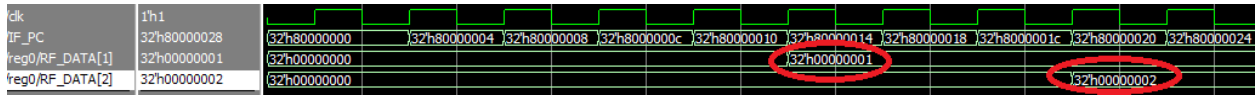


Figure 15: 流水线数据通路测试

4.6 流水线数据转发

连续执行如下三条指令：

```

test_dataforward.asm
    addi $1,$0,1
    add $2,$1,2
    add $3,$1,3

```

这三条指令之间存在数据关联，需要 Data-Forward Unit 正常工作才能正确执行。

仿真结果如 Figure 16所示：

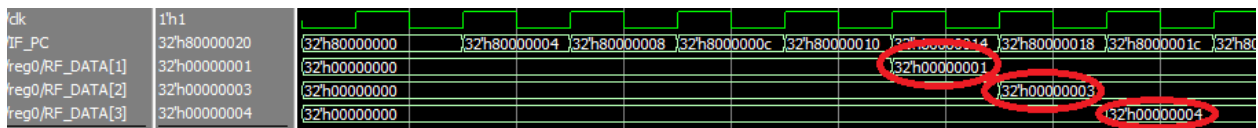


Figure 16: 流水线数据转发测试

4.7 流水线竞争冒险

执行下列指令：

```

test_loaduse.asm
    addu $1, $0, $0
    lui $1, 0x4000
    lw $2, 16($1)
    addi $2, $2, 1
    sw $2, 12($1)

```

lui 和 lw 指令之间存在数据关联，lw 和 addi 指令之间存在 load-use 竞争，需要阻塞一个周期。

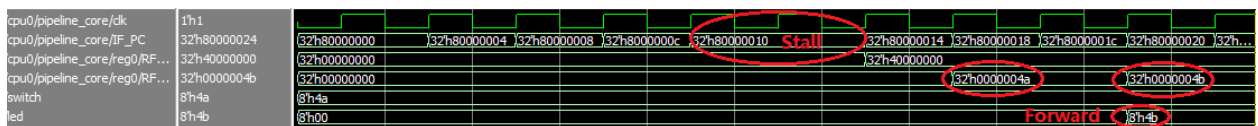


Figure 17: load-use 冒险测试

如 Figure 17所示，流水线被成功阻塞，运算结果正确。

4.8 流水线定时器、七段数码管

在流水线 CPU 上运行 Section 4.2中的汇编代码，测试流水线中的分支、跳转和中断。

如 Figure 18所示，Jump 和 Jump-to-Register 均工作正常，其中 jr 指令采用 dataforward 提前完成了跳转。

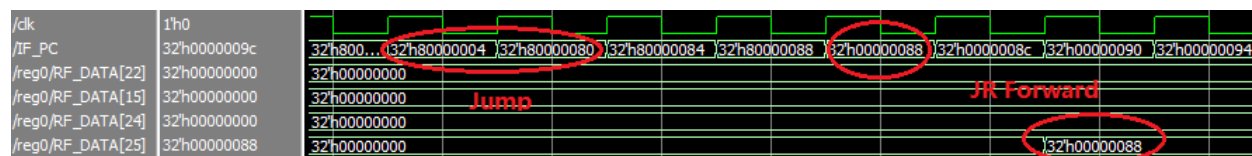


Figure 18: 流水线中的跳转

如 Figure 19所示，中断发生时，程序跳转至中断服务程序入口 0x80000004。

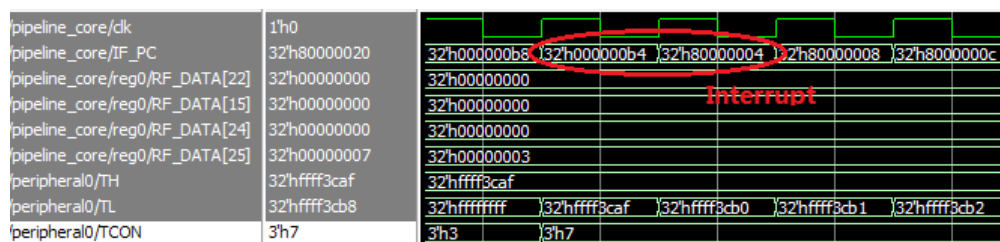


Figure 19: 流水线中的中断

如 Figure 20所示，\$24 记录的是下一个需要点亮的数码管，程序根据 \$24 的值进行分支，点亮不同的数码管，实现译码输出。

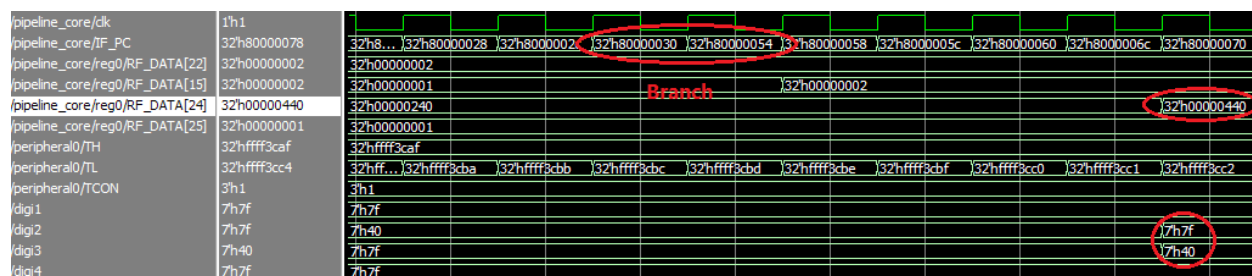


Figure 20: 流水线中的分支

4.9 流水线完整程序

由于流水线 CPU 的时钟频率明显高于单周期，因此我们在流水线中去除了分频模块，并且增大了定时器计数范围。

如 Figure 21所示，从 UART 读入两个操作数，分别存放至 \$s1、\$s2，最大公约数运算结果存放至 \$s3。

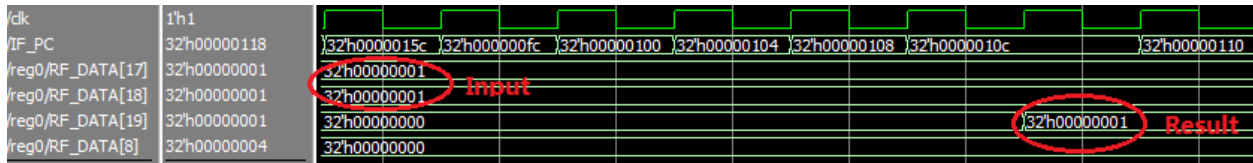


Figure 21: 流水线完整程序运行结果

5 综合情况

Version	Specification	Value
Single Cycle	Total combinational functions	8,990 / 33,216 (27%)
	Dedicated logic registers	9,435 / 33,216 (28%)
	Max frequency	31.81 MHz
	Setup time	8.564 ns
	Hold time	0.391 ns
Pipeline	Total combinational functions	9,699 / 33,216 (29%)
	Dedicated logic registers	9,982 / 33,216 (30%)
	Max frequency	63.36 MHz
	Setup time	4.218 ns
	Hold time	0.391 ns

6 硬件调试情况

- 在测试定时器和七段数码管时，汇编代码的最后一句是 `stop: j stop`，然而中断服务程序返回后程序并未停留在此处。分析数据通路后发现，\$26 寄存器存储的是 `PC+4` 地址，但是中断发生时 PC 语句并没有正确执行，因此中断服务程序返回前需要将 \$26 的值减 4，否则程序将会跳过 PC 语句。
- 理论课讨论的 MIPS 子集中并未包含 `jr` 指令，做流水线设计时也没有考虑到 `jr` 引起的数据冒险。硬件调试我们发现，(a) 中提出先将 \$26 的值减 4 再跳转，这样的设计在流水线中无法正确实现，因此我们决定增加 `jr` 指令的数据转发单元。
- 执行完整的最大公约数程序时，我们发现如果不开启定时器，则 UART 工作正常；一旦开启定时器，UART 就会在 Rx Status 无效时继续读数。仔细排查发现，问题的原因是定时器中断服务程序中使用的 \$t0 寄存器和主程序中的 \$t0 寄存器存在冲突。更换临时寄存器后，问题得到了解决。

7 心得体会

很早就听说过大二暑假的 MIPS 处理器设计，尽管任务艰巨，但我们仍然十分期待。

3 天单周期、3 天流水线，我们用最快的速度 and 严谨的代码风格高质量地完成了处理器设计实验，并且在验收时得到了老师和助教的好评。这 6 天的时间里，我们有过返工，例如陈志杰同学最初编写的 ALU，代码风格欠佳，缺乏注释，大量使用 x、y、z、t 作为变量，给后续的联调造成了许多困难；这 6 天时间里，我们也有过讨论，Coding.Net 讨论区、微信群、寝室都是我们讨论的场所，小组内外的讨论常常持续到凌晨……

实验结束了，看得见的是 Coding.Net 上一行行 commit 记录，看不见的是实验中对 CPU 的深入理解和收获。