Pipeline MIPS Processor 实验报告

16307130194 陈中钰

16 级 计算机科学技术学院

Contents

1 总体状况

1.1 流水线 MIPS 处理器

单周期处理器在一个周期内执行一条完整的指令，结构易于解释且控制单元简单，不需要其他非体系结构状态。但存在以下缺点：

- 时钟周期是由最慢的指令决定的，因此对于其他指令，每个周期都会有多余的时间，整个处理器都处于空闲状态；
- 在一个时钟周期内，只执行一条指令，因此在任意时刻，大部分处理器硬件都处于空闲状态，极大地限制了处理器的吞吐量。

因此，如要提高处理器效率，则需要减少处理器硬件空闲、提高吞吐量。而通过以下的操作，可以实现上述的优化，获得流水线处理器：

- 在单周期处理器中插入 5 个流水线寄存器，分解成 5 个流水线阶段，使得可以在每阶段流水线中同时执行 5 条指令，还可以几乎把时钟频率提高 5 倍，进而在理想情况下，整个处理器的吞吐量可以提高 5 倍；
- 每个阶段的执行所需要的数据，包括 controller 控制信号，都要储存在流水线寄存器中，随着流水线向前传播并保持同步；
- 正在并行处理的指令之间可能存在依赖关系（一条指令依赖另一条指令的结果），这时候就会产生冲突，因此需要另外设计硬件来解决冲突。

尽管引入流水线寄存器增大了硬件成本，同时引入了一些开销，使吞吐量并不能达到 5 倍之高，单条指令运行时间延长，但是流水线仍然有小成本的强大优势，并得到了广泛的应用。

1.2 指令集（共 29 条指令，红色的是添加的 19 条指令）

- 逻辑运算：addi, and, or, add, sub, andi, ori, xor, xori, nor
- 移位运算：sll, srl, sra, lui, nop
- 分支跳转：beq, bne, bgez, bgtz, blez, bltz
- 比较运算：slt, slti
- 内存读写：sw, lw
- 跳转： j, jal, jalr, jr

注意：

- **以上指令的实现按照 MIPS 指令集文档中的格式，故不再附上指令格式要求**
- **jal/jalr 指令调用函数，jr 函数返回后，紧跟 jal/jalr 的指令不会被执行**

1.3 规格

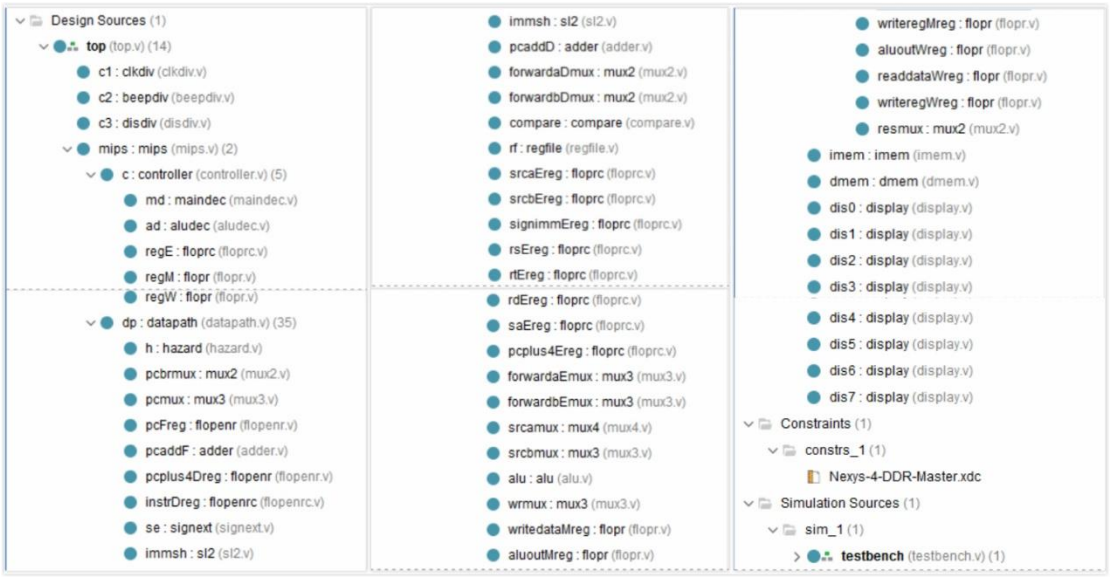
- register file: 32bit*32
- data memory: 32bit*128
- instruction memory: 32bit*128

1.4 实现功能

1. stop：屏蔽时钟，暂停运行
2. next：可在暂停的情况下，运行一条指令
3. reset：重置处理器

4. LED[15:0]：查看电路所有的控制信号
5. C[6:0] + AN[7:0]：查看电路所有的值

1.5 总体结构

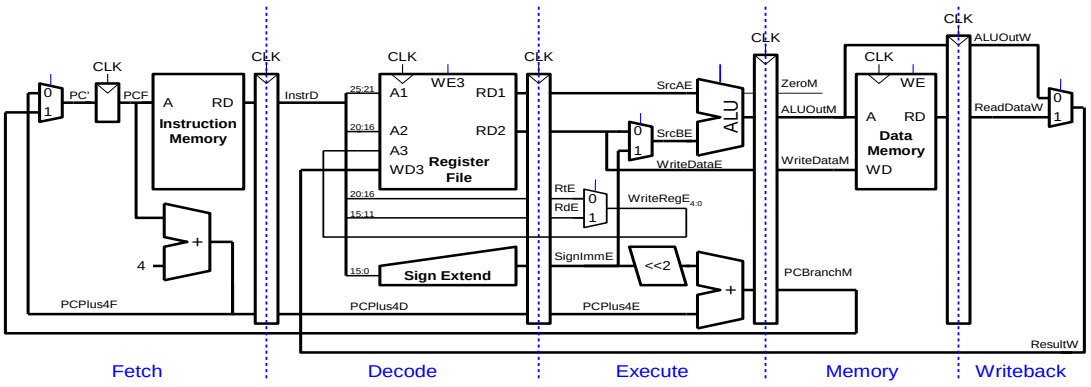


- top：顶层模块
- beepdiv，disdiv，dis0~7：显示模块
- clkdiv：分频模块
- mips，imem，dmem：处理器模块

2 流水线设计

2.1流水线阶段

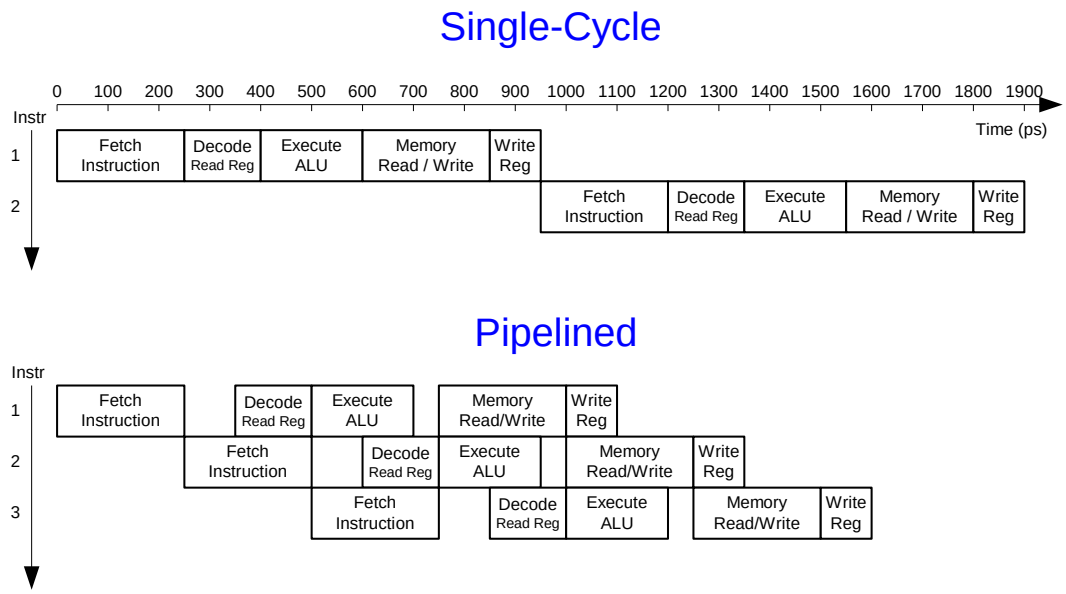
类似于多周期处理器中执行 lw 指令的 5 个步骤，可以分为 Fetch、Decode、Execute、Memory、Writeback 共 5 个阶段，在每个阶段加入流水线寄存器，储存该阶段运行所需的数据。于是，每个阶段只有整个逻辑的 1/5，并负责单条指令在该阶段的操作，并在下一个周期送入下一个阶段去运行下一步骤。



- Fetch：从 instruction memory 中读取指令；
- Decode：对 Fetch 传来的指令进行译码以便产生控制信号，并从寄存器文件中读取源操作数；
- Execute：使用 ALU 进行计算；

- Memory：读写 data memory；
- Writeback：如果需要，把结果写回 register file。

2.2特点

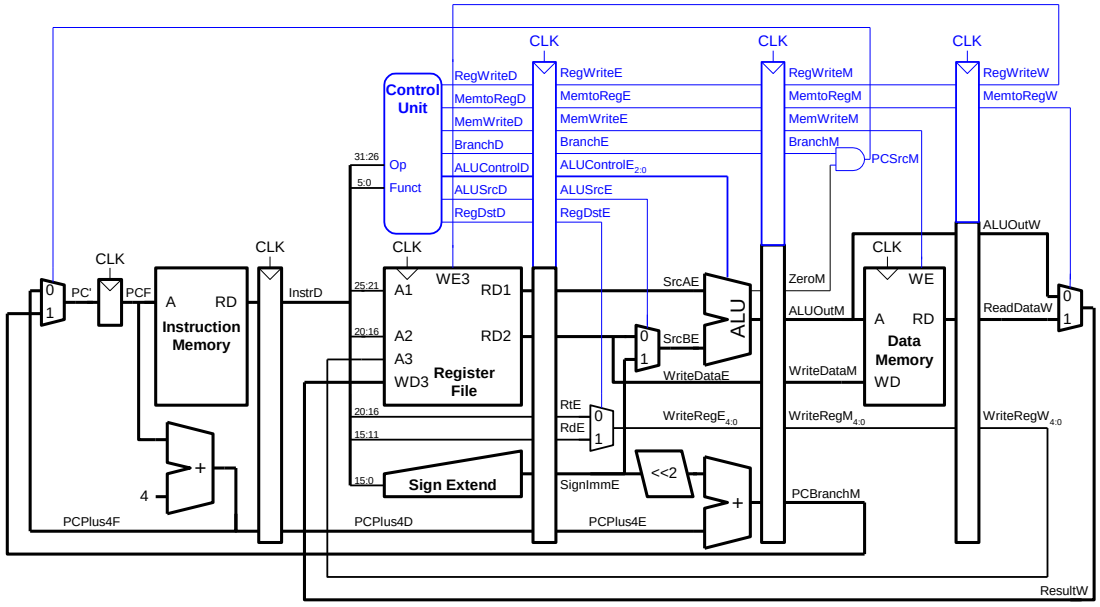


优点	缺点
• 每阶段流水线中同时执行 5 条指令； • 时钟频率几乎可以提高 5 倍； • 吞吐量可以提高近 5 倍。	• 引入流水线寄存器，增加硬件成本； • 产生流水线寄存器延时，使单条指令运行时间延长，使时钟频率并不能提高 5 倍； • 并行指令间可能存在依赖关系，引入冲突，需要额外增加硬件逻辑来解决冲突，同时还可能带来 stall 而降低流水线效率； • 5 阶段运行所需时间并不平均，memory、decode 所需时间更长，因此时钟频率需要以用时最长的阶段为准，故时钟频率并不能提高 5 倍。

尽管流水线增大了单条指令的延时，增加了硬件成本，引入冲突，但是由于处理器每秒执行上百万甚至更多条指令，所以吞吐量的提高尤为重要。由于引入了延时开销、5 阶段时间不均匀，使得吞吐量并不能达到理想的 5 倍，但是流水线依然有小成本的强大优势，并被广泛使用。

2.3流水线控制

流水线处理器与单周期使用大致相同的控制信号，但是与特定指令相关的所有信号还必须通过流水线一起向前传播，与指令的其他部分保持同步。（蓝色部分）



2.4流水线冲突

由于每阶段中有 5 条指令并行执行，当后一条指令需要前一条指令的计算结果，而前一条指令还没有执行完时，会发生冲突，需要设计硬件逻辑来解决。设计结果在下文叙述。

3 显示实现

（代码实现请看工程文件，在此不展示）

3.1disdiv

这是获得七段数码管扫描显示所需时钟的分频模块，主体是一个计数器。

- 对输入时钟的上升沿进行计数，每达到 2^x 次后，输出时钟会发生 1 次变化，产生上升沿/下降沿；
- 上述 x 取决于代码中 q[x]中的 x 值；
- 如果 x 越大则输出时钟周期越长，而 x 越小周期越短、越接近输入时钟的周期。

由于开发板在任意时刻，只能加载一个 7 段数码字，要能同时显示 8 个不同的数字，需要利用人的视觉残留，在适当的频率下轮流加载、显示 8 个 7 段数码字中的一个，可以产生 8 个 7 段数码“同时”显示的效果。

- 取 x=17，能稳定显示 8 个不同的数字；
- 如果 x 偏大则周期过长，会明显看到 8 个 7 段数码是轮流显示的；
- 如果 x 偏小则频率过高，会导致明暗不一、闪烁、部分不显示，甚至显示错误数字等视觉错觉。

3.2beepdiv

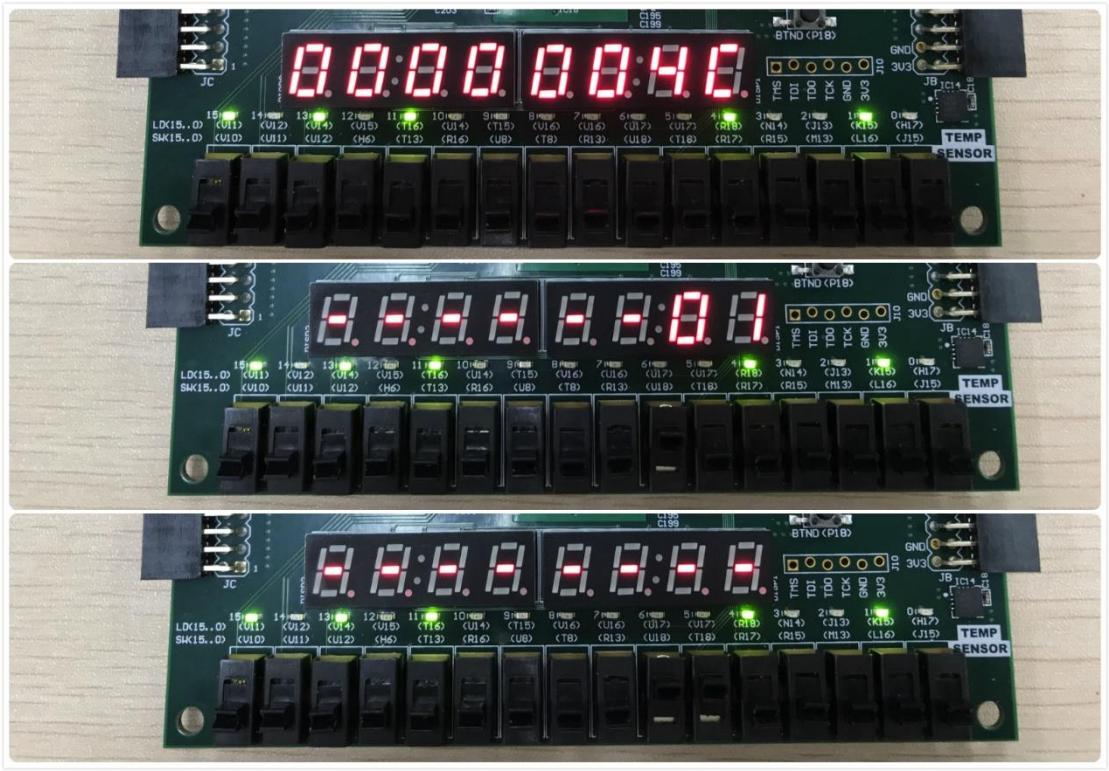
用 LED 显示控制信号时，有部分 LED 是多余无效的，于是加载一个缓慢的时钟信号 beepclk，使其以一定频率闪烁，与有效的控制信号（亮/灭）区分开。

- 原理与 disdiv 一致；
- 取 x=26，输出时钟频率约为 1Hz，使 LED 约每秒闪烁 1 次。

3.3display

这个模块把数值转换为 7 段数码字对应显示。

- 传入一个 4bit 的数值；
- 如 disop 为 1，数值有效，则转化为十六进制的 7 段数码显示，并加载到临时 C 中；
- 如 disop 为 0，数值无效，则转化为 - 符号，并加载到临时 C 中；
- 对于一个 32bit 数值，可以用 8 个 display 模块分别对其进行转换，分别加载到 C0~C7 上，并轮流扫描显示。



3.4输入

- 开关：

开关	next	stop	reset	lightsrc[2:0]	SW[9:7]	SW[6:0]
功能	在暂停时，上升沿使处理器运行单条指令	为1时暂停运行	重置处理器	选择加载到 LED 上的控制信号模块	选择要在 7 段数码上显示的模块	从SW[9:7]选出的模块中选出要显示的值
实现原理	修饰时钟 clk：next|((~stop)&clk)		if 语句判断是否加载 0 值	case 赋值语句	嵌套 case 赋值语句	

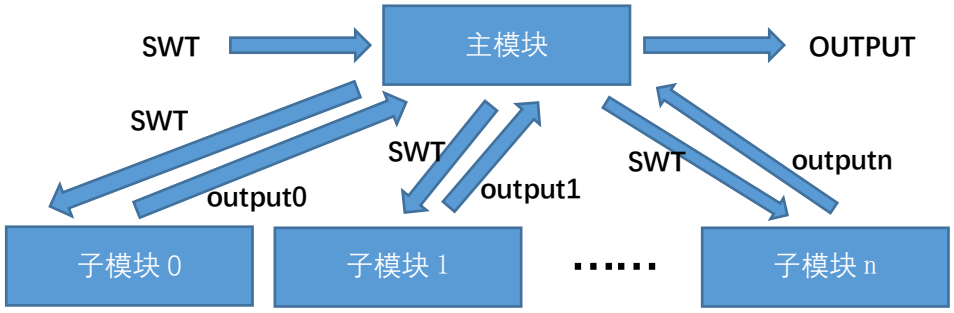
- clk :原始时钟信号（频率 :100MHz）。通过时钟分频模块产生其他频率的时钟信号。

3.5输出

3.5.1 输出值获得方式（之后的多值输出均采用此方式获得）

现假设需要在主模块中，按照 SWT 来获得 OUTPUT，分为以下两种处理方式：

1. 如果输出值的定义在主模块中
   - 直接使用 case 赋值语句进行选择；
2. 如果输出值的定义在子模块中：



   - 向子模块 x(0<=x<=n)中输入 SWT（如果子模块中只需要取 1 个值，那么就不需要传入 SWT），在子模块中用 case 赋值语句把输出值加载到 outputx 上
   - 再在主模块中用 case 赋值语句进行选择，并加载最终的输出到 OUTPUT 上
3. 如果输出值定义在子模块的子模块中
   - 子模块=>主模块，子模块的子模块=>子模块；
   - 继续用相同的方式逐级处理。
4. output 说明

output 可能不止 1 个，比如 output 可以有 2 个，一个代表取出的数值，另一个代表显示条件。

- 在七段显示中，output 不仅要获得显示的数值，还需要获得显示数值的有效性；
- 在 display 模块中，把 4bit 数值转换为 1 个七段数字，同时有 1bit 的控制信号 disop 来决定是否有效，有效输出数字，无效则输出 –
- 那么对于 32bit 的数字，要用 8 个 display 模块，把 8 个 4bit 数值转换为 8 个七段数字，同时有 8bit 的控制信号 disop[7:0]来分别决定每个数字是否有效，如果 disop[x]为 1，则代表 x 位数字有效，否则无效；
- 所以对于七段显示，要获得的 output 是数值+显示条件，即两个 output。

3.5.2 LED 显示

- 由于控制信号很重要，选择用 LED 进行显示；
- **能显示所有的控制信号；**
- lightrsrc[2:0]进行选择，使用上述取值方式获得 LED[15:0]进行显示；
- 无效信号位加载入约 1Hz 的时钟信号 beep，使其闪烁，与其他有效信号位区分开。

lightsrc[2:0]	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

0	regwriteD	memtoregD	memwriteD	alucontrolD	alusrcaD	alusrcbD	regdstD	branchD	pcsrcD	extend
1	aluopD		jumpD	cmpcontrolD	闪烁					
10	regwriteE	memtoregE	memwriteE	alucontrolE	alusrcaE	alusrcbE	regdst	闪烁		
11	regwriteM	memtoregM	memwriteM	闪烁						
100	regwriteW	memtoregW	闪烁							
101	stallF	stallD	flushE	lwstallD	branchstallD	jalrstallD	forwardaD	forwardbD	forwardaE	forwardbE	闪烁
其他	闪烁									

3.5.3 七段数码显示

1. 输出方式

- C[6:0]:在某一时刻需要输出的 7 段数码数字；
- 采用上文说明的取值方式来获得要显示的 T[31:0]，及其对应的显示条件 disop[7:0]，并分解为 8 个 4bit 数值和 8 个 1bit 显示条件，分别用 display 模块进行转换，并分解加载到 8 个 C[6:0]上；
- AN[7:0]:指定 8 个中需要点亮的 7 段数字，AN[x]为 1，则说明 x 为七段数字要显示；
- 使用 disdiv 模块分频获得的时钟，以及 0~7 计数器，轮流对 32bit 数值所产生的 8 个七段数字进行显示；
- 由于频率合适，能产生视觉残留，可以发现 8 个七段数码字“同时”显示，完整

显示十六进制的 T 值，如果有多余的无效位，显示为 - 。

2. 输出值

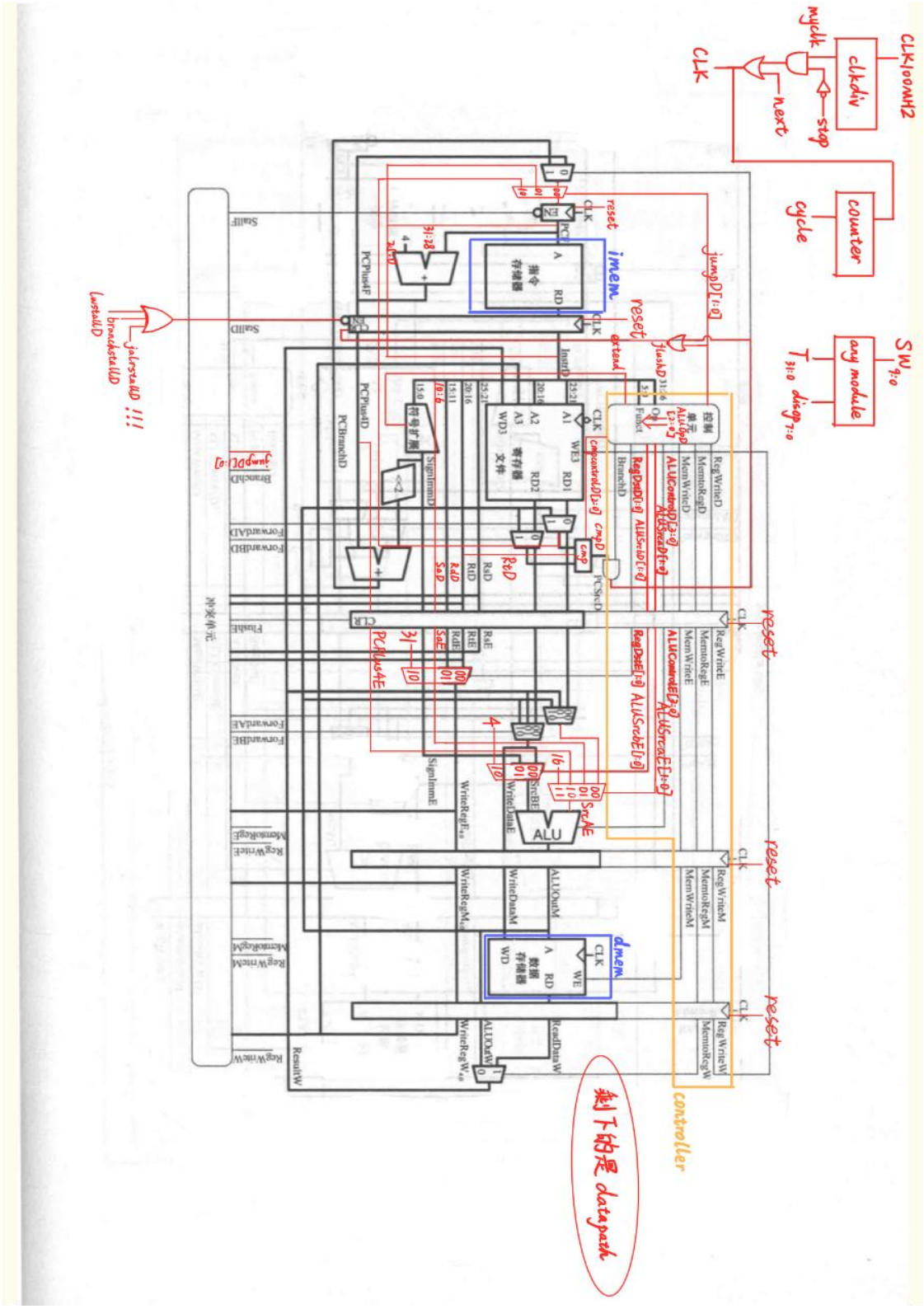
- 通过上述取值方式，以及七段数码显示方式，**可以显示除了控制信号（用 LED 进行显示）以外的全部数值，包括 5 个阶段中的每个数值，以及 regfile、imem、dmem 的完整显示，此外还能输出 cycle 数（通过计数器实现）；**
- SW[9:7]的模块对应关系

SW[9:7]	0	1	10	11	100	101	110	111
模块	Fetch	Decode	Execute	Memory	Writeback	regfile	imem	dmem

- SW[6:0]选择各个模块中的数值

4 模块实现

（代码请看工程文件，在此不展示）



（原图可以查看**硬件设计.png**）

基于课本设计进行实现，但由于额外添加 19 条指令，对处理器设计进行了修改（红色的为增加/修改的逻辑）。此外，冲突分析处理以及 hazard 模块的实现将单独进行阐述。

4.1clkdiv

获得处理器所需的时钟信号的时钟分频模块。

- 原理与 disdiv 一致；
- 若在开发板上：取 x=26，输出时钟频率约为 1Hz，约每秒运行一条指令；
- 若在 simulation：取 x=0，输出时钟频率与输入时钟一致，使程序能在短时间 10~20ms 内运行结束，从而方便查看波形图；
- 此模块从原时钟 clk 获得 myclk，再通过时钟修饰 realclk=next|((~stop)&clk)，获得处理器真正使用的时钟信号 realclk，且带有单条指令运行、暂停的动能（前文已叙述）。

4.2cycle

- 在 top 模块定义的 32 位 reg 型变量，用于记录程序运行周期数；
- intial 置 0；
- 对 realclk 上升沿进行计数，所计数目则为 cycle 数，使用计数器实现。

4.3MIPS

- MIPS：处理器中最重要的模块，**由 controller、datapath 两个模块组成**；
- datapath：含有处理器中的大部分硬件模块、5 个流水线寄存器，并连接 5 个阶段成数据通路，根据 controller 模块生成的控制信号运行，另外还包含冲突处理模块 hazard；
- controller：含有 maindec、aludec（位于 D 阶段），共同根据 instrD 生成 D 阶段的控制信号，还含有 regE、regM、regW 寄存器模块，分别为 E、M、W 阶段的控制信号寄存器，保证 D 阶段生成的控制信号随着流水线向前传播，并和对应指令保持同步，共同调控 datapath 中的各个阶段以及 dmem 的运行。

4.3.1 controller

在《数字设计和计算机体系结构》书中的基础上，做了以下**修改：**

1. aluop[2:0] + alucontrol[3:0]：由于添加了 andi、ori、xori 等逻辑运算，使得 aluop 拓宽到 3 位、alucontrol 拓宽到 4 位长；
2. alusrcaD[1:0] + alusrcaE[1:0]：由于移位指令要用到 saE、jalr 指令要用到 pcplus4E、lui 指令要用到常数 16，所以要 srcaE 共有 4 个选择，要用 MUX4 进行选择，还要添加 2 位长的 alusrca 信号，同时还要随流水线从 D 阶段传播到 E 阶段，并对 srca 进行选择；
3. alusrcbD[1:0] + alusrcbE[1:0]：由于 jalr 用到了常数 4，所以 srcbE 共有 3 个选择，要用 MUX3 进行选择，并把 alusrcb 延长为 2 位，同时还要随流水线从 D 阶段传播到 E 阶段，并对 srcb 进行选择；
4. extend：由于 addi 等指令为符号扩展，而 andi 等指令为零扩展，所以要增加 extend 信号，控制立即数扩展模块进行对应的扩展；
5. cmpcontrolD[2:0]：由于比较器有 6 种比较方式（beq、bne、bltz 等），因此要生成 3 位的控制信号，来控制比较模块进行对应的比较；
6. jumpD[1:0]：由于 j、jal 用到{pcF[31:28],instrD[25:0],2'b00}，jalr、jr 用到 srca2D，所以在选择 pcF 中共有 3 种选择，要用 MUX3 进行选择，并添加 2 为 jumpD 信号来对 pcF 进行选择；
7. regdstD[1:0] + regdstE[1:0]：由于 jal 要选择常数 31 作为 writeregE，所以一共有 3 种选择，则要改用 MUX3 进行选择，并把 regdst 延长为 2 位，

同时还要随流水线从 D 阶段传播到 E 阶段，来对 writeregE 进行选择；

4.3.2 datapath

在《数字设计和计算机体系结构》书中的基础上，做了以下**修改：**

1. jumpD[1:0]控制的地方：添加 MUX3 对 pcF 做进一步的选择（上文已描述）；
2. flushD：在 jal、jalr 等情况下同样也要进行跳转，故 D 流水线寄存器的错误指令也要 bubble 掉，所以产生 bubble 的 flushD 信号在 jumpD 为 01 和 10 的情况下也同样有效；
3. flopenrc 在 datapath 中只有 D 流水线寄存器用到了这个模块，可以被 stall 或者 bubble。如果指令被 stall 了，那么则不能被 bubble，否则就会一条指令尽管在 stall 之后就能正确运行了，却会被 bubble 掉。所以在 stall 和 bubble 同时成立的情况下，stall 具有更高的优先级；

```
always@(posedge clk,posedge reset)
if(reset)
    q<= #1 0;
else
    if(~en)//stall (if stalled, it
        q<= #1 q;
    else
        if(clear)
            q<= #1 0;
        else
            q<= #1 d;
```

4. 扩展模块：添加零扩展的选项，受 extend 信号控制；
5. saD + saE：由于移位指令在 E 阶段的 srcaE 会选择 saE 位移位位数，因此在 D 阶段要添加 saD 的获得，并通过流水线传播到 E 阶段中；
6. compare 模块：把原来只用于判断是否相等的模块，添加为 beq，bne，bgez，bgtz，blez，bltz 共 6 种比较，由 maindec 产生的 cmpcontrol[2:0]信号来选择。但由于 blez、bgtz 具有相同的 op 和 funct，所以也会有相同的 cmpcontrol，无法区分开来，因此还要添加 rtD 的输入，通过 rtD 来区分 blez、bgtz 的选择；
7. regdstE 控制的地方：改为 MUX3 进行选择（上文已描述）；
8. alusrcaE 控制的地方：添加 MUX4 进行选择（上文已描述）；
9. alusrcbE 控制的地方：改为 MUX3 进行选择（上文已描述）；

4.4 imem

imem 用于加载、储存指令，由 reg 型数组实现。在程序初始化时，通过$readmemh 从.dat 文件读入全部指令并储存。程序运行中时，传入 pc[7:2]作为 addr，在每一个上升沿读取指令。

4.5 dmem

dmem 用于储存、读取数据，由 reg 型数组实现。传入 dataadr 作为地址，读出数据，并根据 memwrite 信号决定是否要写入数据。

4.6 top

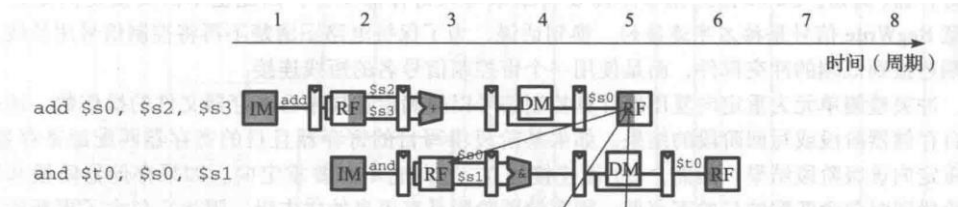
top 为顶层模块，除了用于在开发板上显示数值、控制信号外，就是组织 MIPS、dmem、imem 的结构。MIPS 进行正常的处理器运行，并从 imem 中读取 instr 指令来执行，从 dmem 中读取数据（如 lw 等）或者向 dmem 写入数据（如 sw 等）。

5 冲突处理

5.1冲突类型

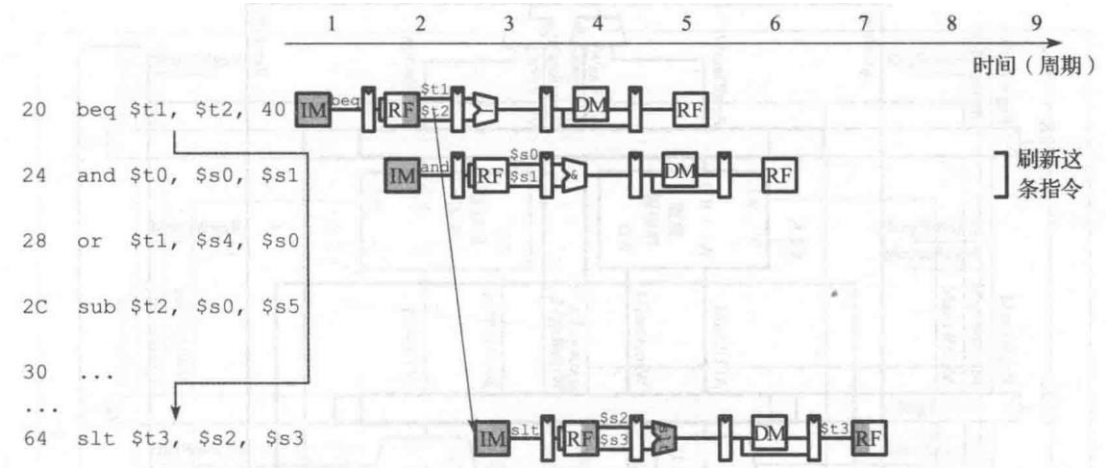
正在并行运行的指令中，如果一条指令依赖于另一条还没有结束指令的结果，那么就产生了冲突。

- 数据冲突：寄存器新的值还未写入，就需要取该寄存器的值；



add 在周期 5 时才把新值写入$s0，但 and 在周期 3 就需要取$s0 值，但此时$s0 还未更新，会取到错误的值。

- 控制冲突：在取指令时还未能确定下一条指令应该取的地址；

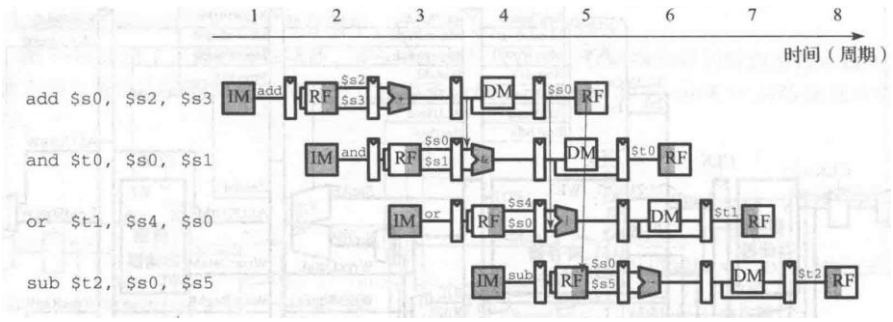


在 beq 之后并未能确定下一条指令应该取的地址。

5.2冲突解决（hazard 模块）

5.2.1 重定向（forward）

1. M/W->E（从 M 阶段或 W 阶段向 E 阶段 forward）



- [M->E 例子](memfile 程序中有验证)：add 在 M 阶段时仍未写入$s0，但处于 E 阶段的 and 已经需要$s0 了，所以要把新值 forward 到 E 阶段，则 add 和 and 指令是 M->E；
- [W->E 例子](memfile 程序中有验证)：而 add 在 W 阶段时写入$s0，但 or 已经取了$s0 的旧值去到了 E 阶段了，所以要把新值 forward 到 E 阶段，则 add 和 or 指

令是 W->E；

（在 or 之后 add 的结果就已经写入了，不再有冲突）

- hazard 模块接受在 E 阶段中指令的两个源寄存器，以及 writeregM、writeregW，同时检查 regwriteM、regwriteW 信号；
- 在 regwrite 有效，且在 E 中的 rsE 寄存器与对应阶段的 writereg 相同时，要把对应阶段的结果 forward，可以选择把 M 或 W 阶段的结果 forward，或者不 forward；
- 另外由于$0 硬连接为 0，故 rsE 为$0 时不需要重定向，去掉这种情况；
- forwardaE 实现代码：

```
forwardaE=2'b00;
if(rsE!=0)
begin
    if(rsE==writeregM & regwriteM)
        forwardaE=2'b10;
    else
        if(rsE==writeregW & regwriteW)
            forwardaE=2'b01;
end
```

- hazard 生成 forwardaE、forwardbE 信号后，控制对应的 MUX3 来选择正确的结果；
- 对于 forwardbE 的实现也是类似的，只不过 forwardaE 针对 rsE，而 forwardbE 针对 rtE，其他的都一致。

2. M->D

```
add $s0, $s2, $s3
sub $s1, $s2, $s3
 beq $s0, $s1, 1
```

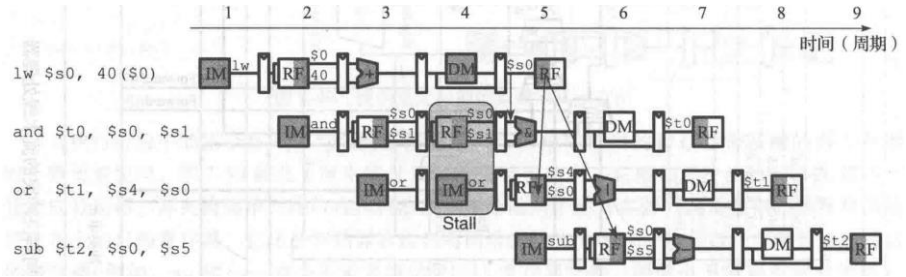
- [例子](full 程序中有验证)：当 add 在 M 阶段时仍未写入$s0 的值，但是在 D 阶段的比较操作已经需要$s0 了，所以需要把结果 forward 到 D 阶段，则 add 和 beq 为 M->D；
- 和上述 M/W->E 类重定向的实现类似；
- 获得 forwardaD、forwardbD 后，控制对应的 MUX2 进行选择；

```
//forwarding sources to D stage (branch equality)
assign forwardaD=(rsD!=0 & rsD==writeregM & regwriteM);
assign forwardbD=(rtD!=0 & rtD==writeregM & regwriteM);
```

5.2.2 阻塞（stall）

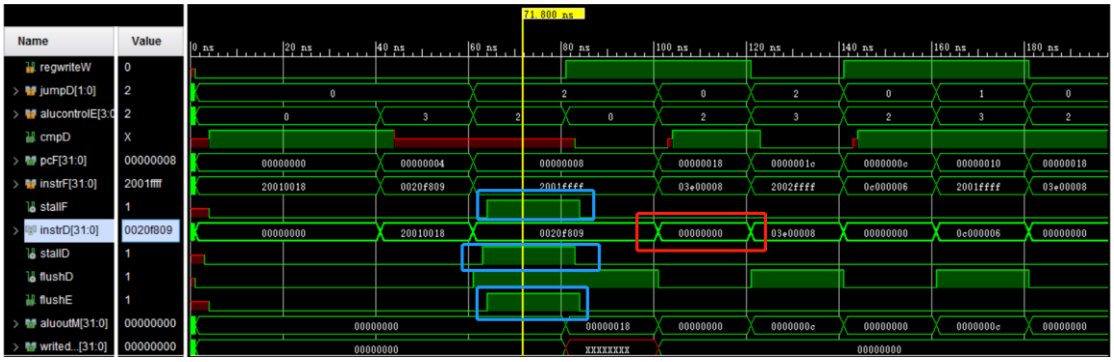
```
//stall
assign #1 lwstallD=memtoregE & (rtE==rsD | rtE==rtD);
assign #1 branchstallD=branchD &
                       (regwriteE & (writeregE==rsD | writeregE==rtD) |
                        memtoregM & (writeregM==rsD | writeregM==rtD));
assign #1 jalrstallD=(jumpD==2'b10) &
                     (regwriteE & writeregE==rsD |
                      memtoregM & writeregM==rsD);//jalr waits for the r

assign #1 stallD=lwstallD | branchstallD | jalrstallD;
assign #1 stallF=stallD;//stalling D stalls all previous stages
assign #1 flushE=stallD;//stalling D flushes next stage
```

1. lwstallD

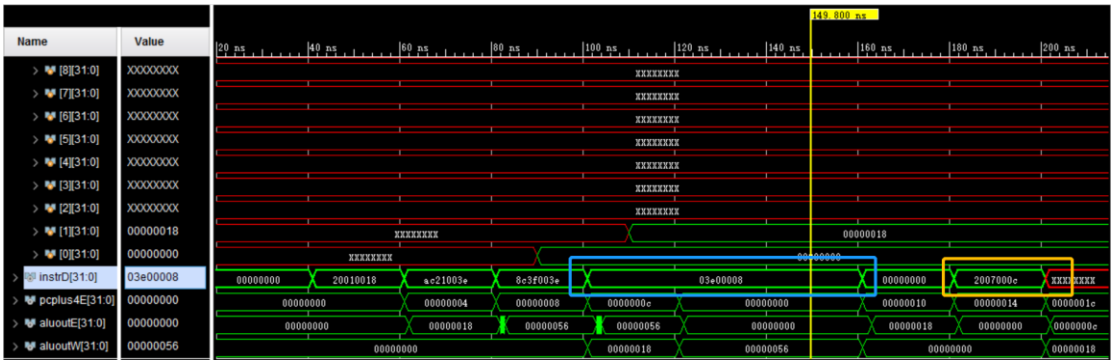


- [例子](memfile 程序中有验证)：lw 读取到$s0，但 and 就需要$s0 的值了。当 and 到 E 阶段需要$s0 的值时，lw 才到 M 阶段，要等到 W 阶段才能读出数据。因此当 and 在 D 阶段时要被 stall 一个周期，并等 lw 到了 W 阶段、and 到了 E 阶段，就可以把结果 forward 到 and 指令。

2. branchstallD
   - (memfile 程序中有验证)：处理器需要在 D 阶段完成分支判断，如果分支指令的源寄存器依赖于处于 E 阶段的 ALU 指令，或者依赖于 M 阶段的 lw 指令，处理器需要把 D 阶段 stall 至源操作数准备好为止。
3. jalrstallD
   - jalr/jr 指令在 D 阶段需要读取寄存器，如果该寄存器依赖于 E 阶段的 ALU 指令，或者依赖于 M 阶段的 lw 指令，处理器需要把 D 阶段 stall 至源操作数准备好为止；
   - jalrstallD 和 branchstallD 的原理一样，不过 branchstallD 只针对 branchD 为 1 的指令，即分支指令，而 jalrstallD 只针对 jumpD 为 10 的指令，即 jalr/jr 指令。
   - 在 addi-jalr.txt/addi-jalr.dat 程序中，验证了 jalr 依赖于 E 阶段的 ALU 指令的情况，验证结果：



   - 在 lw-jr.txt/lw-jr.dat 程序中，验证了 jr 依赖与 M 阶段的 lw 指令的情况，验证结果：



4. stallD
   - 当上述 3 种 stall 任一种有效时，最终的 stallD 就有效，所以是或运算获得 stallD；
   - 当 D 阶段被 stall 时，F 阶段也要被 stall，同时 E 阶段没有指令运行，需要 bubble 掉（flush 信号有效）。

5.3其他冲突解决

- 插入 nop：使产生冲突的指令有足够时间完成；
- 重排列：把其他指令移过来，前提是不产生新冲突，改变了位置之后不改变程序原意。

6 仿真测试

6.1代码说明

- 位于 test_files 文件夹，含有 **10 份测试代码**；
- **全部测试代码都是自己独立、手写完成的；**
- 每份测试代码有.dat、.txt 为后缀的两份同名文件。其中，**.dat 为十六进制代码**，用于输入处理器中运行；而**.txt 文件为完整的代码说明**，含有小程序说明、汇编代码、代码描述、地址、十六进制码、二进制码等内容，部分测试代码还有 C 语言代码，方便检验代码、处理器的正确性；
- 在验证正确性时请对照.txt 文件，文件中的代码描述很详细的了，每一条指令都写明了结果，方面验证；
- jal/jalr 后第一条指令不会被执行，函数返回后从 jal/jalr 后第二条指令开始执行；
- 请用 Notepad++来查看测试代码，用其他方式打开可能会对不齐！
- .txt 示例：hanoi.txt 测试代码（部分）

```
# hanoi
# in this program, it calculate hanoi(3,a,b,c)=7, and store the result in $s0, and in the end $t0=$s0=7
# and print out the sequence of moving disk in data memory: ac,ab,cb,ac,ba,bc,ac

#       Assembly            Description             Address     Machine     Binary

# main
main:   addi $29, $0, 127   # $sp = 508(4*127)      0           201d01fc    001000 00000 11101 0000 0001 1111 1100

        addi $16, $0, 0     # $s0 = 0(count)        4           20100000    001000 00000 10000 0000 0000 0000 0000
        addi $28, $0, 0     # $gp=0(output pointer) 8           201c0000    001000 00000 11100 0000 0000 0000 0000

        addi $4, $0, 3      # $a0 = 3               c           20040003    001000 00000 00100 0000 0000 0000 0011
        addi $5, $0, a      # $a1 = a               10          2005000a    001000 00000 00101 0000 0000 0000 1010
        addi $6, $0, b      # $a2 = b               14          2006000b    001000 00000 00110 0000 0000 0000 1011
        addi $7, $0, c      # $a3 = c               18          2007000c    001000 00000 00111 0000 0000 0000 1100
        jal  hanoi          # call hanoi            1c          0c00000a    000011 000000 0000 0000 0000 0000 1010
```

- .dat 示例：hanoi.dat 测试代码（部分）

```
201d01fc
20100000
201c0000
20040003
2005000a
2006000b
2007000c
0c00000a
00000000
08000035
23bdffec
afbf0000
afa40004
```

6.2仿真准备

- 在 imem 模块（imem.v 文件），需把如下地址修改为执行代码.dat 文件的实际地址

```
31
32      initial
33      $readmemh("C:/Users/ECHOES/Desktop/pipeline_pj/MIPS_pipeline_32bit/MIPS_pi
34
```

- 在 clkdiv 模块（clkdiv.v），分频取 outclk=q[0]，使输出时钟频率与输入时钟一致，使程序能在短时间 10~20ms 内运行结束，从而方便查看波形图；
- 仿真用于检验处理器是否能按照所执行的代码要求运行、能否实现相关功能，同时也能检验测试代码是否书写正确、是否达到代码设计的目标；
- **检验时主要通过观察 instrD（程序真正运行了的指令）、regfile、dmem（程序会对 regfile、dmem 进行修改）、MUX 选择输出值（指令所需数值是否取到了）、控制信号（控制处理器运行）等值来判断处理器、测试代码是否正确。**

6.3栈

- 涉及到调用函数的程序用会用到 stack；
- stack 在 dmem 中实现；
- 通过栈指针$sp 来对 stack 进行读写，一般在程序开始时把$sp 指向 dmem 的最高处 dmem[127]，并在需要用 stack 的时候向下扩大空间。

6.4寄存器集

表 6-1 MIPS 寄存器集

名字	编号	用途	名字	编号	用途
$0	0	常数 0	$t8 ~ $t9	24 ~ 25	临时变量
$at	1	汇编器临时变量	$k0 ~ $k1	26 ~ 27	操作系统临时变量
$v0 ~ $v1	2 ~ 3	函数返回值	$gp	28	全局指针
$a0 ~ $a3	4 ~ 7	函数参数	$sp	29	栈指针
$t0 ~ $t7	8 ~ 15	临时变量	$fp	30	帧指针
$s0 ~ $s7	16 ~ 23	保存变量	$ra	31	函数返回地址

- **在涉及到函数调用、递归的有意义的小程序中，均遵守上述寄存器使用原则；**
- 其他一般的程序则没按照上述原则，因为意义不大。

6.5代码验证

6.5.1 memfile.txt/memfile.dat

- memfile 测试代码由课本测试代码改编而来，含有指令集大部分指令；
- 由于有多个指令反复更改相同的寄存器，而且代码本身没有实际含义，只是零碎的指令执行，因此要想验证代码是否正确，需要一条条指令来检验，验证过程较复杂；
- 为了能更好地验证全部指令，我设计了 full 代码，请看下文。

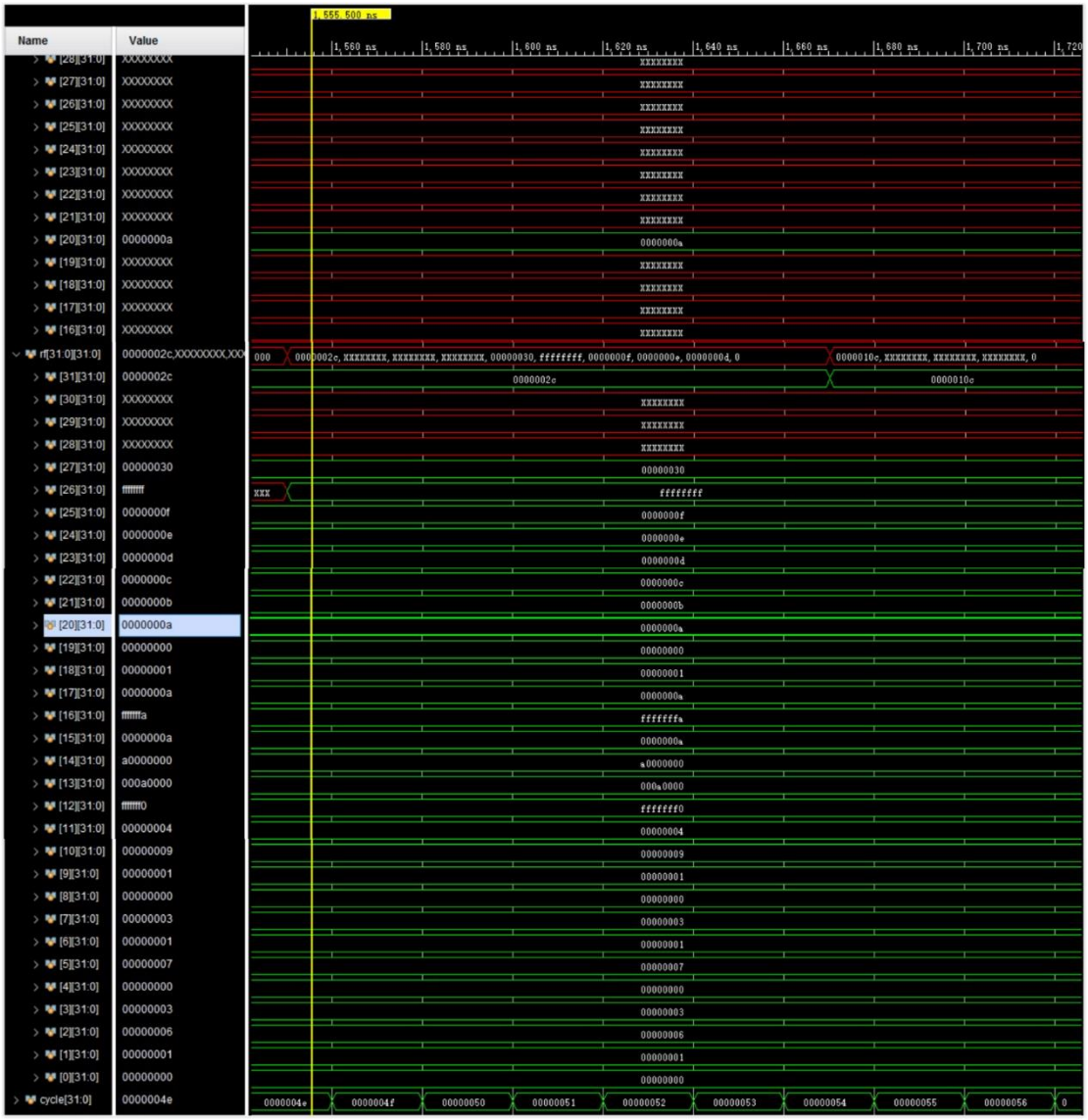
6.5.2 full.txt/full.dat

- full 代码含有全部指令；
- 每条指令会对应把结果值存到 1 个寄存器，且每条指令的计算结果互相不会覆盖，因此只需查看最后时刻的全部寄存器值、dmem 值就能验证正确性；
- full 代码的 main 函数调用 cal、shift、mem、smaller、branch 函数，每个函数并分别测试不同的指令；

- [main 函数]测试跳转指令：j，jal，jalr，jr
  main 函数使用 jal/jalr 来调用函数，在调用函数后使用 jr 返回，调用完全部函数后 j 跳转到函数结尾，向 reg[26]写入 ffffffff 代表程序结束；
- [cal 函数]测试逻辑计算指令：addi，and，or，add，sub，andi，ori，xor，xori，nor。向 reg[0~11]写入 16307130194，向 reg[12]写入 fffffff0。
- [shift 函数]测试移位指令：sll，srl，sra，lui，nop(nop 是 sll $0，$0，0 的伪指令)。输出 a 的各种移位结果。
- [mem 函数]测试内存读写指令：sw，lw。对 dmem[20]进行读写。
- [smaller 函数]测试比较指令：slt，slti。大小比较，slt 的结果应为 1，slti 的结果应为 0。

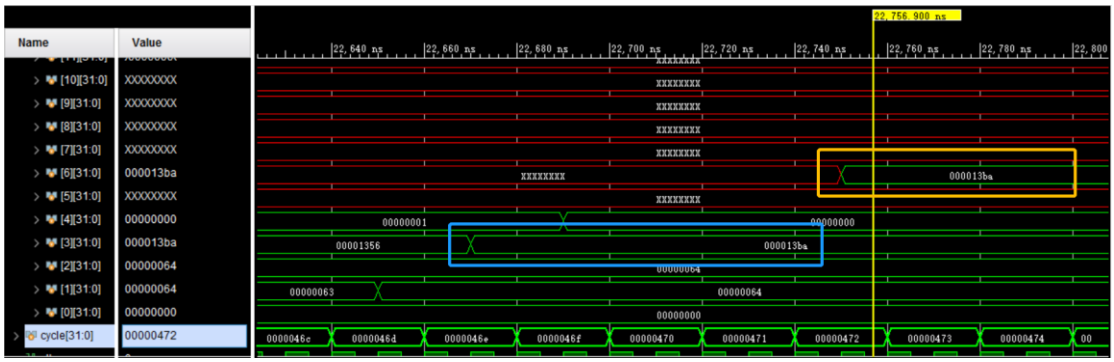
```
bgtz $12, skip      # should not be taken   c4
addi $23, $0, d     # $23 = d               c8
bgtz $6, skip       # should be taken       cc
addi $23, $0, f...f # $23 != f...f          d0
```

- [branch 函数]测试分支指令：beq，bne，bgez，bgtz，blez，bltz。每条跳转指令的验证都是如上图一样的结构——【跳转只跳转到当前指令后的第二条指令处(即只跳过了下一条指令)，而且每个跳转指令都是先测试不跳转，再测试跳转，虽然两次跳转都把结果存到同一个寄存器，但是只有两次都正确了，寄存器的结果才是正确的】
- 如上图的例子，正确结果是先不跳转、再跳转，结果 reg[23]=d；如果两次都跳转，则结果是 reg[23]=x ;如果两次都不跳转，结果为 reg[23]=ffffffff；如果先跳转、再不跳转，则结果为 reg[23]=ffffffff。所以只有两次的跳转都正确，最后的结果才是正确的。



6.5.3 loop.txt/loop.dat

- 1~100 的循环求和；
- 计算结果累计在蓝框 reg[3]，并在最后把结果写入黄框 reg[6]，结果为 0x13ba。

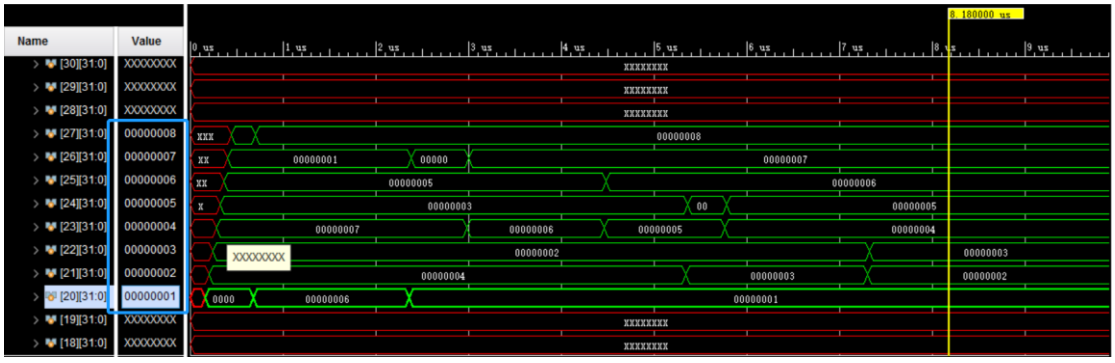


6.5.4 sort.txt/sort.dat

- 1~8 的插入排序，开始时为 **84273516；**
- **84273516->64273518->14273568->14263578->14253678->13254678->**

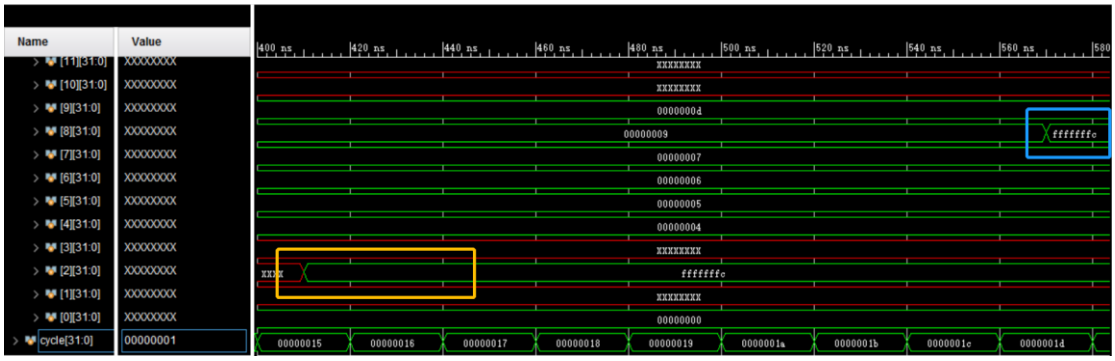
12345678；

- **蓝框中显示为最后排序结束的结果 12345678。**



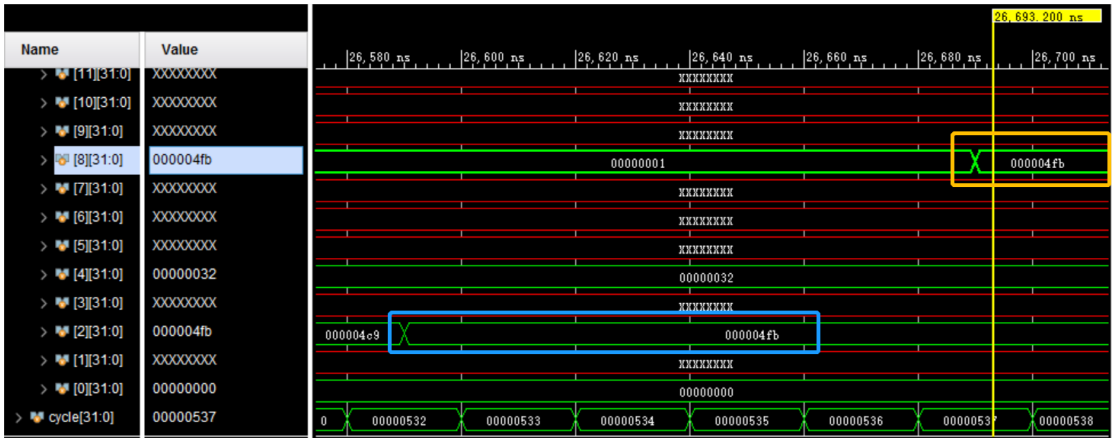
6.5.5 call.txt/call.dat

- 最基础的带参数、返回值的函数调用：dif(a,b,c,d)=(a+b)-(c+d)；
- 计算 dif(4,5,6,7)=(4+5)-(6+7)=-4=0xfffffffc，并把返回值（黄框）取到临时寄存器（蓝框）中。



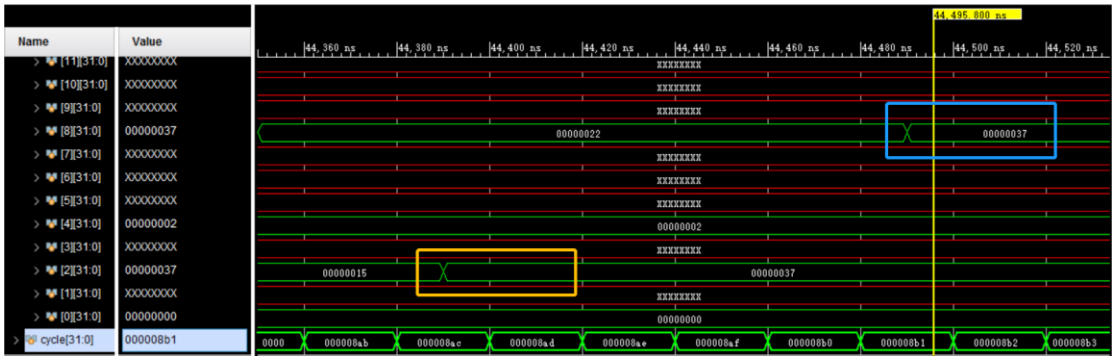
6.5.6 recursive.txt/ recursive.dat

- 递归求 1~n 和；
- 计算 sum(50)=1275=0x4fb，并把返回值（蓝框）取到临时寄存器（黄框）中。



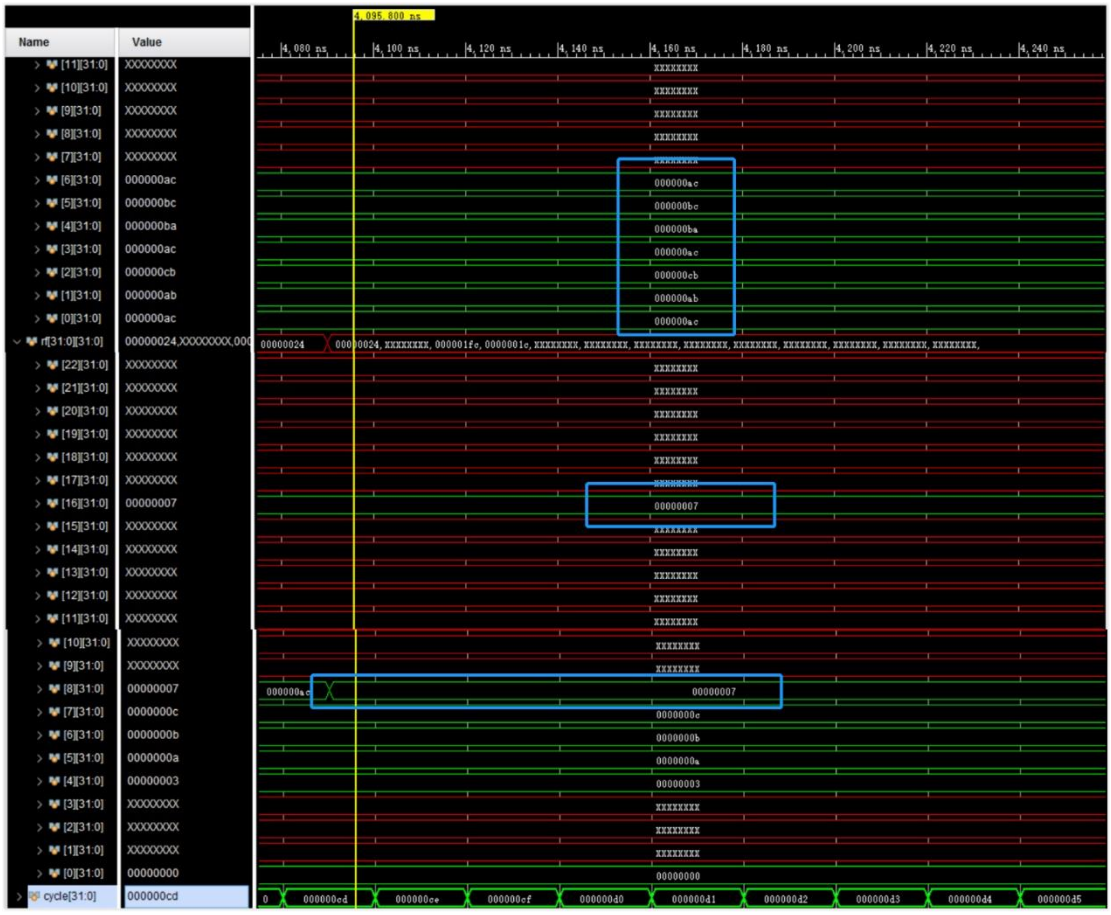
6.5.7 fibonacci.txt/ fibonacci.dat

- fibonacci 计算程序
- 计算 fibonacci(10)=55=0x37，并把返回值（黄框）取到临时寄存器（蓝框）中。



6.5.8 hanoi.txt/hanoi.dat

- 3 根杆子（a、b、c 杆子）的 hanoi tower 问题，能输出最少移动次数（全局变量 reg[16]）和移动方案（dmem 从最低位[0]开始每次移动分别输出，例如输出 ac 则代表 a->c）；
- reg[16]全局变量统计移动次数，最后把结果写入临时寄存器 reg[8]，移动方案在 dmem 中输出 ac,ab,cb,ac,ba,bc,ac。



7 性能分析

测试环境：

- 统一运行 loop 程序：循环求 1~100 和；

- `clkdiv` 分频：统一取 `q[0]`，即取和原时钟相同的频率；
- 从上到下分别是：单周期、多周期、流水线。

7.1时钟占用

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	7.437 ns	Worst Hold Slack (WHS):	0.324 ns	Worst Pulse Width Slack (WPWS):	4.500 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:	19	Total Number of Endpoints:	19	Total Number of Endpoints:	20

All user specified timing constraints are met.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	7.654 ns	Worst Hold Slack (WHS):	0.265 ns	Worst Pulse Width Slack (WPWS):	4.500 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:	19	Total Number of Endpoints:	19	Total Number of Endpoints:	20

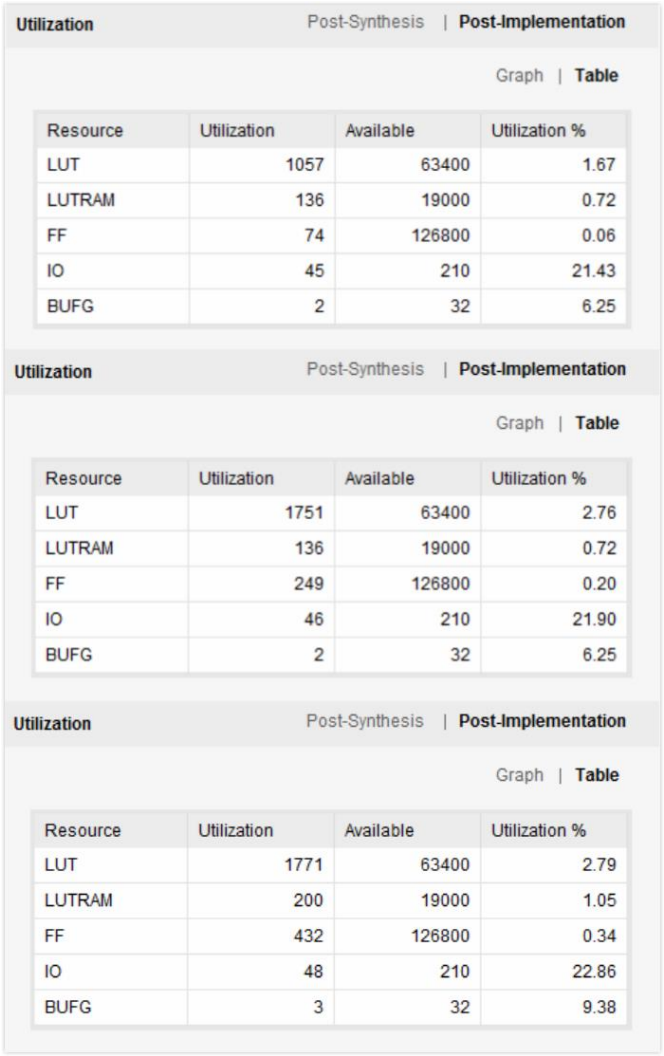
All user specified timing constraints are met.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	6.979 ns	Worst Hold Slack (WHS):	0.114 ns	Worst Pulse Width Slack (WPWS):	4.500 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:	45	Total Number of Endpoints:	45	Total Number of Endpoints:	46

All user specified timing constraints are met.

7.2资源占用



Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	1057	63400	1.67
LUTRAM	136	19000	0.72
FF	74	126800	0.06
IO	45	210	21.43
BUFG	2	32	6.25

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	1751	63400	2.76
LUTRAM	136	19000	0.72
FF	249	126800	0.20
IO	46	210	21.90
BUFG	2	32	6.25

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	1771	63400	2.79
LUTRAM	200	19000	1.05
FF	432	126800	0.34
IO	48	210	22.86
BUFG	3	32	9.38

8 实验感想

- 对流水线 MIPS 处理器的运行机制更加了解；
- 更加熟悉流水线冲突产生原因及解决方法；
- 对 MIPS 指令集有了更深的了解，并学会书写有意义的 MIPS 指令程序；
- 对 verilog 语言更加深入；
- 能熟练运用 SW 来实现功能，能使用 LED 以及 8 个 7 段显示数码来直观地显示所有数值、控制信号，能极大地方便 debug 及正确性验证。