

|  |
| --- |
| 32位MIPS多周期处理器设计 |

李晓畅 | 20307130261 | 2022年5月15日

# **一、简述**

微体系结构连接逻辑和体系结构，组合寄存器、ALU、有限状态机、存储器和其他逻辑模块，实现一种体系结构。本实验设计的32位MIPS单周期处理器就处于这一层次。这意味着我们需要在熟悉算数单元与存储单元这些更低抽象层次设计的基础上进一步理解寄存器、指令、存储器等概念以实现更高的抽象。

本实验设计的MIPS处理器产生于上世纪八十年代，它经典、简洁，虽然相对简单，但正如MIPS是最早出现的商业RISC架构那样，学习设计MIPS将有助于我们理解真实的商业体系结构。

# **二、设计原理**

**１. 设计原则**

Patterson和Hennessy提出了MIPS 体系设计的四个准则，即：

1） 简单设计有助于规整化

2） 加快常见功能

3） 越小的设计越快

4） 好的设计需要好的折中方案

我们将在下面的具体实现中深刻体会到这几点。

**２. 体系结构状态与指令集**

计算机的体系结构包括指令集和体系结构状态。

一台计算机能执行的机器指令的集合称为该机的指令集或指令系统，它是构成程序的基本元素，也是硬件设计的依据。因而指令系统的设计设计是计算机系统设计的关键之一。MIPS采用RISC技术，指令系统简单，控制逻辑得以大大简化。

MIPS处理器的体系结构状态包括程序计数器和32个寄存器。这些原件是任何MIPS处理器的必要组成部分。基于当前体系结构状态，处理器执行一条具有特定数据集的特定指令，将产生一个新的体系结构状态。

为了使得微体系结构易于理解，我们将首先着手实现MIPS指令系统的一个子集，进而考虑将其加以扩展，以实现更复杂的功能。指令选择将在下面阐述。

**3. 时钟周期**

单周期CPU指的是一条指令的执行在一个时钟周期内完成,然后开始下一条指令的执行,即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿,两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。时钟周期一般也称振荡周期

CPU在处理指令时,一般需要经过以下几个步骤：

1） 取指令(Fetch)：根据程序计数器PC中的指令地址,从存储器中取出一条指令,同时,PC根据指令字长度自动递增产生下一条指令所需要的指令地址,但遇到"地址转移"指令时,则控制器把"转移地址"送入PC,当然得到的"地址"需要做些变换才送入PC。

2） 指令译码(Decode)：对取指令操作中得到的指令进行分析并译码,确定这条指令需要完成的操作,从而产生相应的操作控制信号,用于驱动执行状态中的各种操作。

3） 指令执行(Execute)：根据指令译码得到的操作控制信号,具体地执行指令动作,然后转移到结果写回状态。

4） 存储器访问(Memory)：所有需要访问存储器的操作都将在这个步骤中执行,该步骤给出存储器的数据地址,把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

5） 结果写回(Writeback)：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

对于流水线CPU,这五个阶段分别对应了其中的一个流水段。

具体到本次的设计，我们选择100MHZ时钟作为系统时钟，又选择其下降沿为IO时钟。

**4. 并行**

系统的速度可以用延迟和通过系统信息的吞吐量来度量。在同一时间内处理多个任务可以提高吞吐量，这一方法称为并行。它有两种形式：空间和时间。

空间并行提供多个硬件，这样多个任务就可以在同一时间内一起处理。

时间并行将一个任务分成多个段，多个任务可以分配到所有段。这样虽然每一个任务仍然需要通过所有段，但这样任意给定时间内每段都有一个不同的任务，从而使多个任务可以重叠起来。时间并行也称为流水线。

**5. 如何进一步提高CPU性能**

流水线技术是提高数字系统吞吐量的有效手段。同过将单周期处理器分解为5个流水段来构成流水线处理器。因此，可以在每阶段流水线中同时执行五条指令。由于每段仅仅占有整个逻辑的五分之一，所以时钟频率几乎可以提高五倍。因此，虽然每条指令的延迟并未改变，但理想情况下吞吐量可以提高五倍。微处理器每秒执行上百万甚至数十亿条指令，所以吞吐量比延迟更重要。流水线引入了一些花销，因此吞吐量不能达到理想的要求，但是流水线依然有小成本的强大优势，所有现代高性能微处理器都使用流水线技术。

不难看出，流水线处理器相较于多周期处理器采取了相似但有差异的方法提高单周期处理器的性能。多周期处理器要求只使用一个ALU单元，一组通用存储器以降低成本，这实际上同流水线储器的思路是类似的，但目的不同。类似于在多周期处理器中为每个执行阶段安排ALU运算，流水线处理器同样着眼于合理使用所有资源，因而在同一流水段中允许五条指令同时执行，实际上是允许一组指令分别在统一时间内使用不同的资源，但不同的是其目的是减少资源空闲导致的浪费以提高吞吐效率，提高处理器性能。因为这一改进几乎可以成倍提高处理器性能，因而流水线处理器放弃了多周期处理器中的一些优化与改进，比如使用统一存储器。

# **三、实验方案**

**1. 实验目的**

1）设计实现32位MIPS流水线处理器

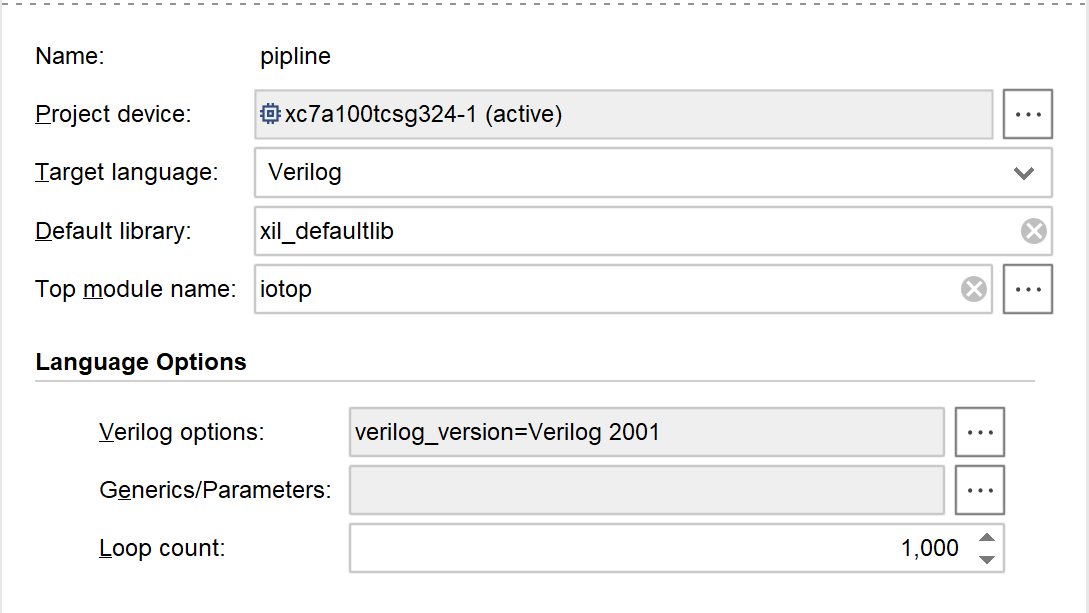
2）实现指令扩展

3）实现IO接口

4）加深对微体系结构的理解

**２. 环境和器材**

使用Ｖｉｖａｄｏ２０１８．３设计开发，应用开发板测试验证。具体如下。

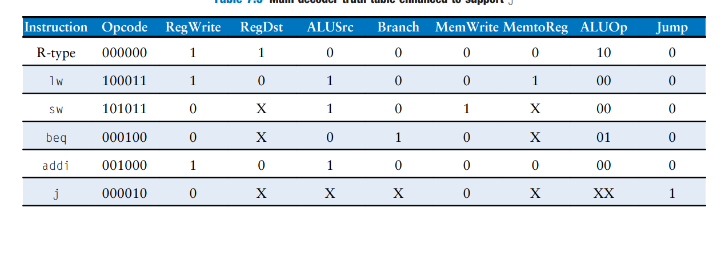


**３. 设计过程**

1）数据路径设计

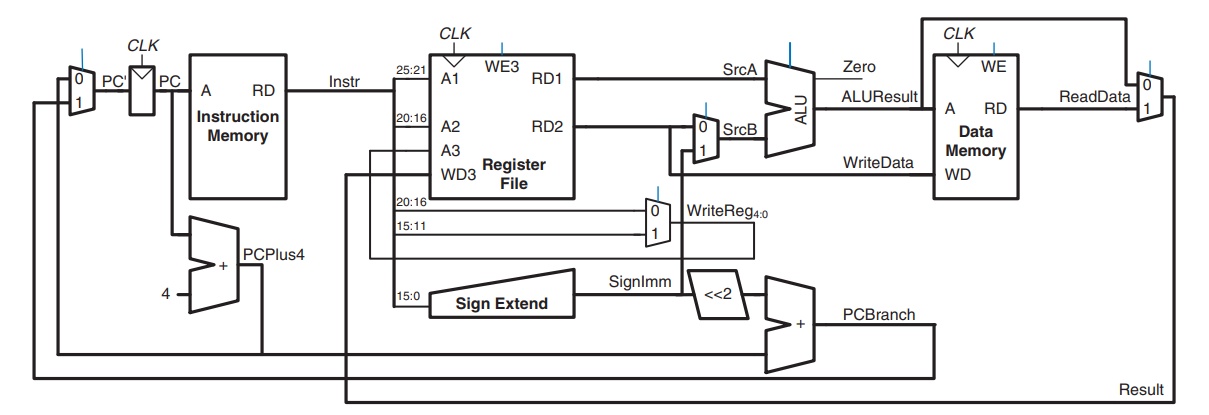
类似于在单周期和流水线中所做的，我们将考虑首先实现MIPS指令集的一个子集，接着再加以扩展。为了保证能够实现需要的功能，我们选择了下面几条指令。

前面五条我们将在设计时直接实现，而最后一条指令和立即数逻辑运算将以扩展的形式实现。



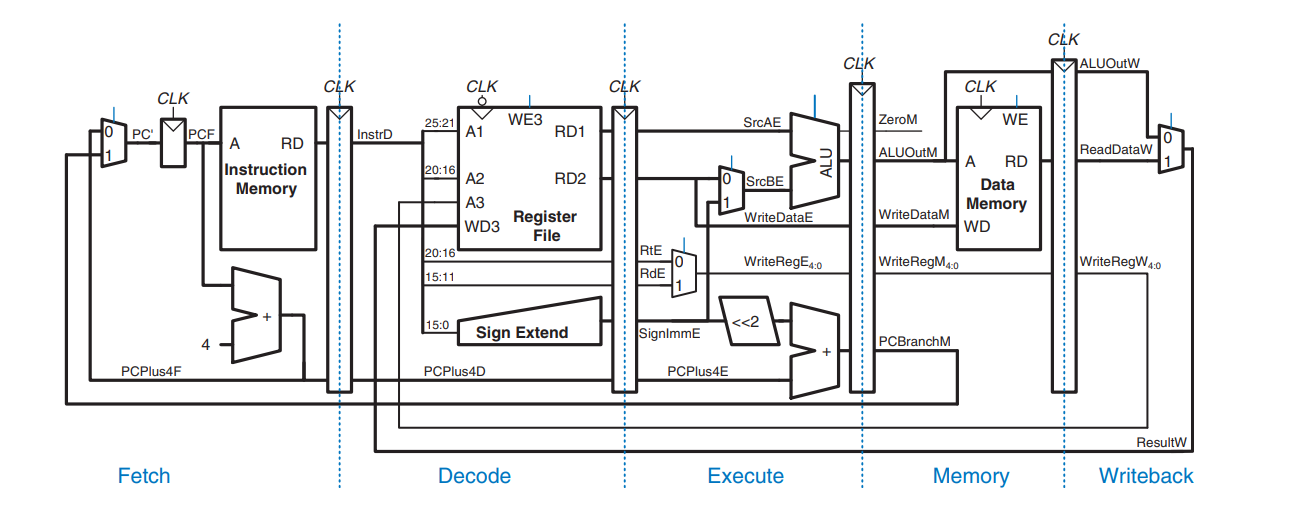
确定了需要实现的指令集，再加上一些必要的模块，我们接下来就可以考虑数据通路的设计和控制逻辑的设计了。

不同与多周期处理器的设计，在设计流水线处理器时我们有可能更直接地借鉴之前的设计。比如下面这个单周期处理器的数据通路。



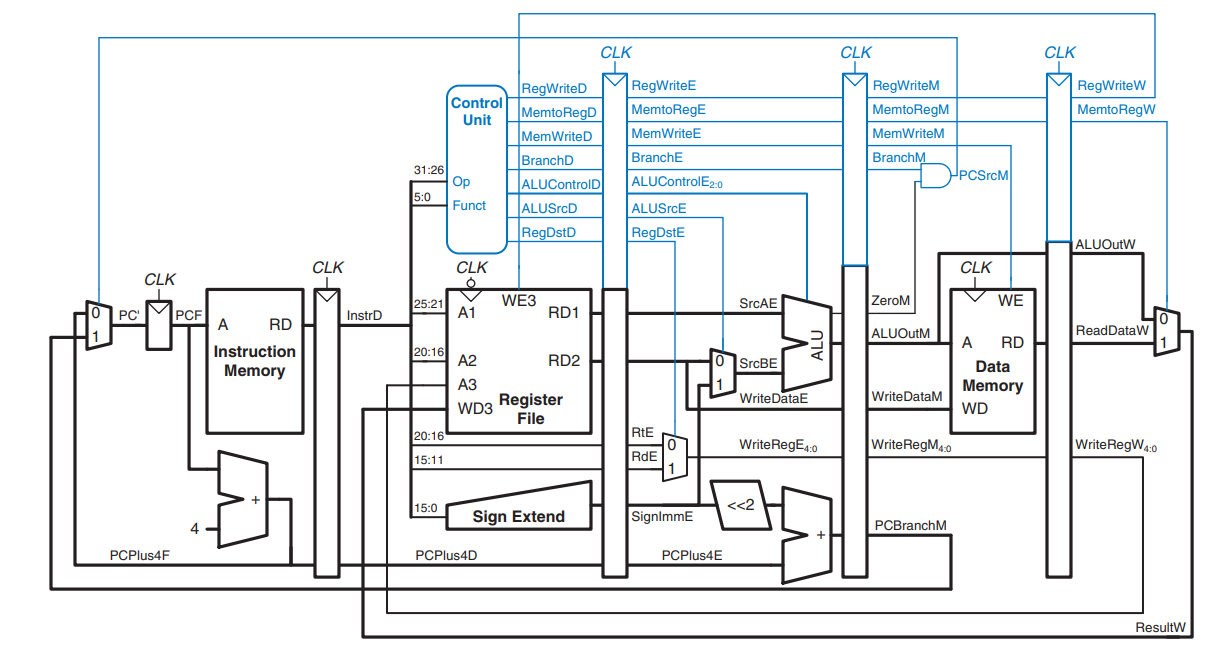
注意到流水线处理器的数据路径是由流水线寄存器将单周期处理器数据路径划分为五个阶段，我们可以考虑直接在上面这个单周期处理器数据通路的基础上加以修改。显然我们应该加入四个流水线寄存器以将数据通路分割为五个部分。参考指令执行的五个阶段，我们不难将之插入到合适的位置。由于各段同一信号可以属于不同的指令，为加以区分，我们可以为每个信号加一个后缀，FDEMW，对应其执行阶段。

另外还有一个值得注意的地方。流水线处理器中数据虽然基本上都是从前向后流，但写寄存器的情况却比较特殊。寄存器文件被划分在译码阶段，但写寄存器应该在写回阶段发生。为保证写回阶段的正确性，我们需要稍微修改数据通路，使得写回阶段的信号与指令相对应。于是就可以得到下面这样的数据通路。



我们不难从数据通路的设计中体会到流水线处理器对单周期处理器的继承和发展。

2）控制模块设计

接下来我们需要一个根据指令类型产生控制信号的模块，具体来说就是主译码器模块和ALU译码器模块。主译码器模块将基于和字段产生对应的控制信号。将数据通路和控制模块连接，我们就得到了下面的电路。不同于多周期处理器中的控制单元，虽然由于每条指令都只在译码阶段译码因而无需设状态机以译码各个阶段，因而可以使用类似于单周期处理器的译码模块，但问题在于每一流水段中执行的是无关的指令，可能需要不同的控制信号，因而需要有方法存储译码产生的控制信号，并在对应的流水段控制执行。为此我们可以类似地在各个流水段间添加上一些寄存器。于是就得到了下面这样的带控制单元的数据通路。

具体译码模块设计类似于单周期，不同主要在于需要加入寄存器，这里就不加赘述了。

5）冲突处理

流水线处理器之所以能够极大提高处理器性能，是由于多条指令同时在执行。当一条指令依赖于还没有结束的另一条指令的结果时，就会发生冲突。

有的冲突可以通过合理使用资源来规避。比如寄存器文件可以在同一个周期内完成读和写。我们只需要让写操作发生在一个周期的前一半，而读操作发生在周期的后一半，这样寄存器就可以在同一个周期内实现读和写操作而不发生冲突。但这只能够解决部分冲突。

1. 数据冲突

分析写后读RAW冲突，我们不难发现，在时间上说，先开始执行的指令确实先计算出了结果，之所以会产生冲突是因为在下面的一条或两条指令中需要使用这个结果时它暂时还没有写回正确的位置。所以原则上说，我们应该能够将前一条指令的结果重定向给后一条指令来化解RAW冲突。

实际上我们确实可以通过重定向或者说旁路技术解决很多冲突。这需要在ALU的前面增加复用器以便选择来自寄存器文件的操作数，或存储器阶段或写回阶段的结果。除了在ALU的两个端口增加复用器外，还需要冲突检测单元以根据执行阶段的两个源寄存器、存储和写回阶段的目的寄存器和写回阶段的信号确定是否要重定向以及如何重定向。我们可以得到这样的控制信号：

1. **if** ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM)
2. then ForwardAE = 10
3. **else** **if** ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW)
4. then ForwardAE = 01
5. **else** ForwardAE = 00

不过实际上并不是所有冲突都可以通过重定向解决。考虑load-use冲突。Lw指令必须等到存储阶段完成后才产生需要读出的数据，如果下一条指令在执行阶段就要用到这个数据，那么因为它的产生实际上并没有超前，我们也就不可能使用它，这是转发无法解决的问题。为此，我们只能选择阻塞流水线，将操作挂起直至数据有效时。

类似于nop指令，我们可以在流水线中插入为使用的阶段，也就是气泡来实现阻塞。气泡的产生是由在译码阶段阻塞时对执行阶段产生无效的控制信号，使气泡不执行操作，也不改变体系的状态。其控制信号的产生与重定向类似：

1. lwstall = ((rsD = = rtE) OR (rtD = = rtE)) AND MemtoRegE
2. StallF = StallD = FlushE = lwstall
3. 控制冲突

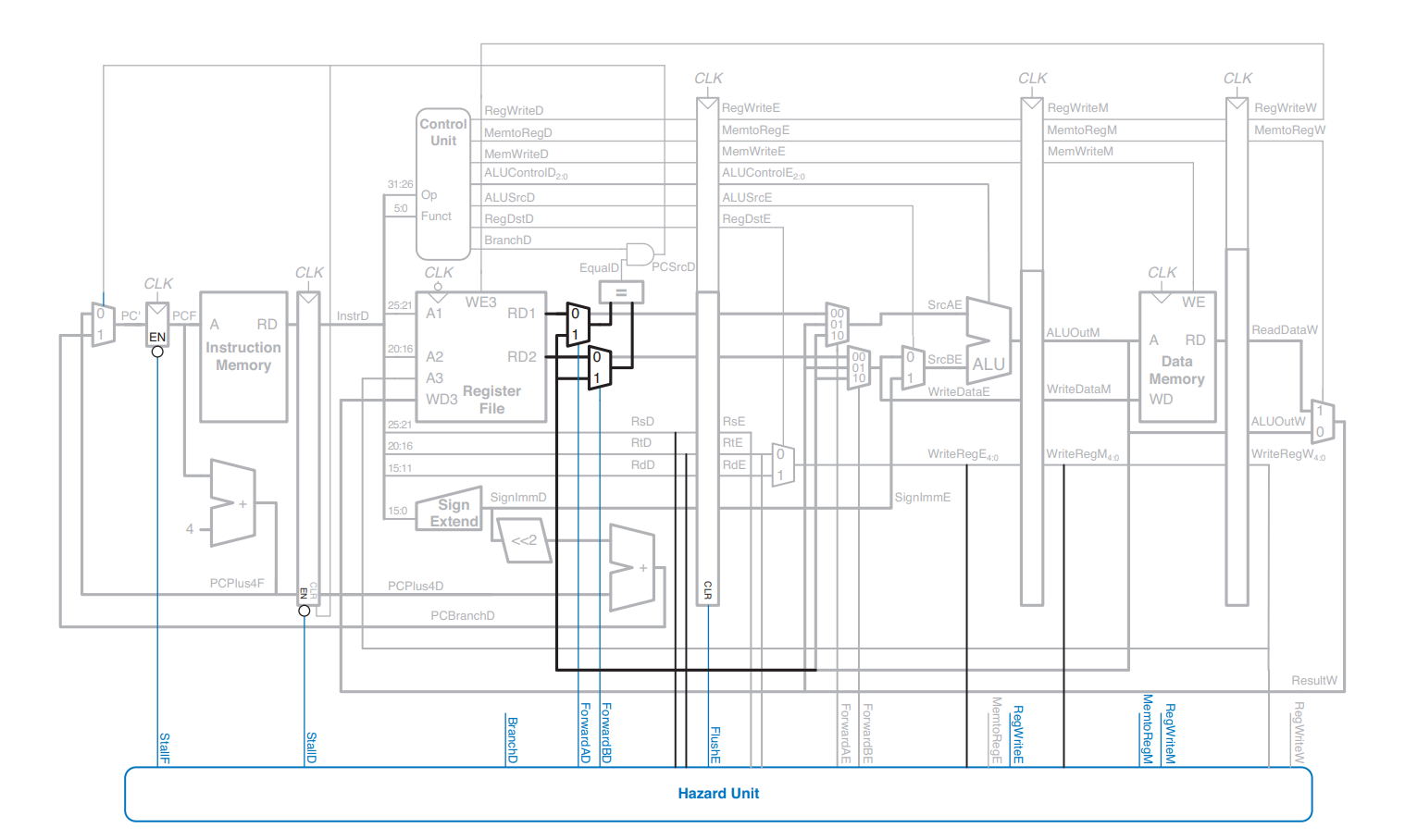
Beq将产生控制冲突，因为在取下一条指令时分支是否发生还尚未确定，所以流水线处理器不知道取那条指令。

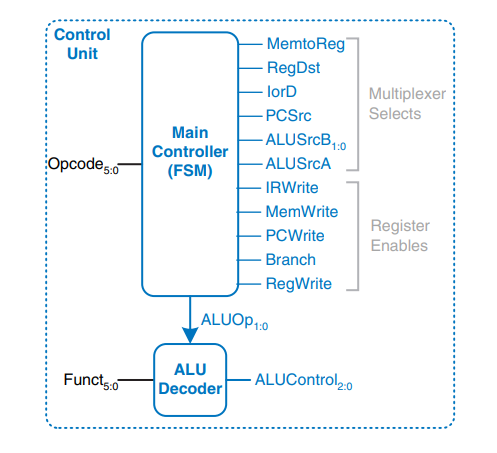
最简单的处理方式是阻塞三个周期。但是beq指令不少见，这样做会严重影响流水线的性能。

尽早确定是否发生分支可以减少分支错误预测的代价。确定分支是否发生只需要比较两个寄存器的值是否相等，使用一个相等比较器要比执行减法和零检测快，这意味着我们或许可以将之放到译码阶段计算，这样可以提前一个周期知道分支是否发生，从而降低代价。

另外还可以采用动态预测的方法来提高预测的成功率。

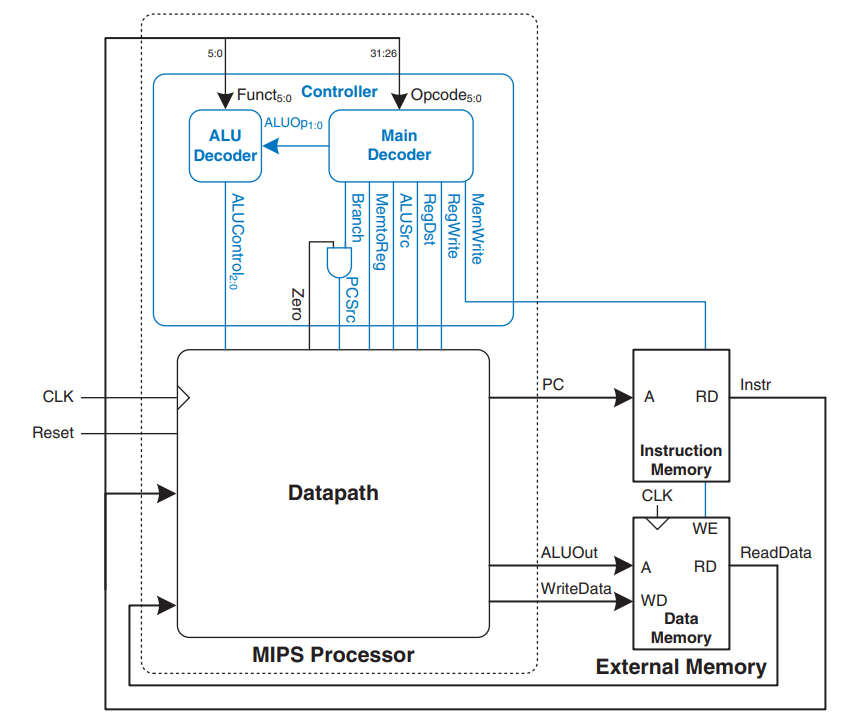
结合上面的分析，我们可以得到这样的包含控制和冲突处理模块的数据通路。



4）IO接口设计

最后，为了实现同外界交互，我们还要增加IO接口。这里我们将实现存储器映像IO接口。在该方法中，I/O接口与存储器共用同一个地址空间；每一个I/O设备占用存储器空间的一个地址；CPU利用等指令对I/O设备的管理； CPU利用存储器读写信号对I/O设备进行读写控制。换言之我们需要做的就是建立IO与存储器的通路，并设计合适的控制信号即可。

在这里，IO接口的设计同在单周期处理器中并没有太大区别，我们只需要将合适的数据接到合适的地方就可以了。那么就可以得到下面这样的设计了。



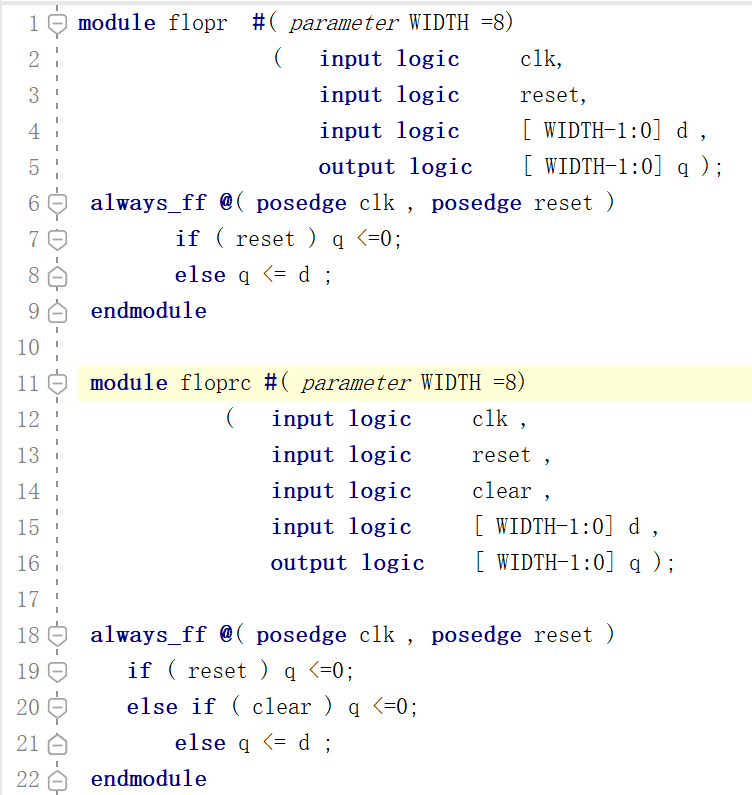
# **四、关键代码、仿真与开发板照片**

**１. 存储器和体系结构状态**

正如我们在设计过程中提到的那样，存储器和体系结构状态在设计中有着重要的地位。我们将首先实现它们。

这些基本元件同在单周期处理器中相同，我们可以直接复用，这里就不再列出了。

值得一提的是，我们需要很多寄存器来实现流水段间的状态传递。这些寄存器虽然在实现与功能上是类似的，但不同流水段寄存器却应该有不同数目的接口，有的还需要能够清空。我们可以像下面这样设计这些寄存器。



这样我们在使用时先根据需求决定选择需要清零的或不需要清零的，再根据数据总宽度决定总宽度。在控制模块中使用的寄存器也可以类似地设计。

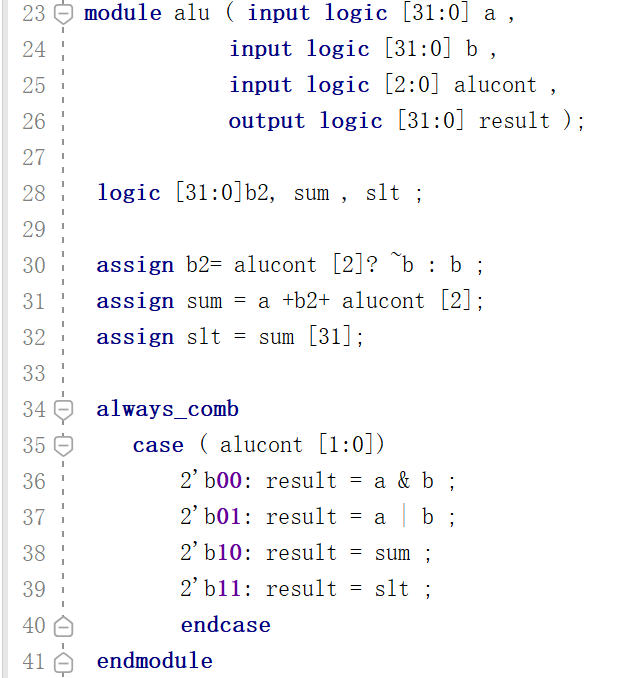
另外，我们应该修改寄存器文件的读取时间，使得寄存器文件总可以先写后读。具体实现上同单周期相似。

**２. 主要模块**

１） ALU模块

ALU模块在处理器中有着重要作用，对运算有着巨大影响。我们可以像下面这样设计它。包含两个32位输入，一个三位控制信号决定计算类型，一个零信号作为标志位输出，以及一个32位输出信号。只需要根据控制信号选择正确的运算并加以输出即可。

它可以直接复用单周期处理器中的ALU模块，不过由于我们直接在译码时就可以完成比较，为beq和bne指令设计的zero信号就可以不要了。



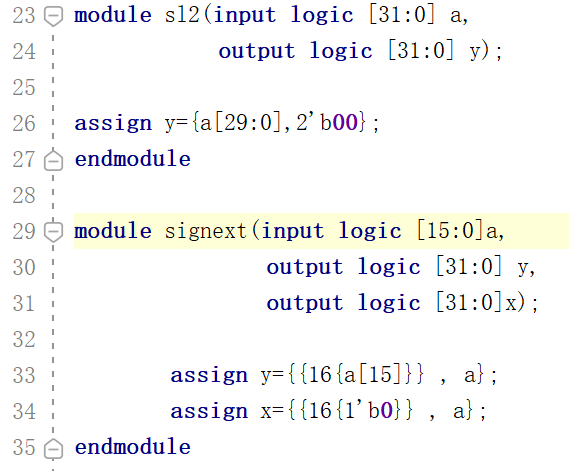
类似地，我们可以设计出加法器和相等比较器。

２） 符号扩展、０扩展和左移两位模块

我们的３２位处理器经常需要接受一些小于32位的信号，为此我们就需要对它们进行扩展。而扩展又分为符号扩展和０扩展。因而我们可以这样设计扩展模块。

扩展模块接受一个十六位的立即数，将之符号、零扩展为32位信号。对它的选择将在下一阶段进行。

左移两位与之类似。



**３. 数据通路**

就像在设计过程中提到的那样，有了上面提到的模块，我们就可以设计数据通路了。

数据通路例化了前面提到的原件，将之有机连接以实现指令。我们将数据通路作为模块单独封装了起来，并提供了同其他模块，比如控制模块间的接口，这样我们在日后更的时候这一模块仍旧是独立的。具体来说就是下面这样的实现。

1. module datapath (   input logic         clk,reset,
2. input logic [31:0]  instrF ,
3. input logic         pcsrcD , branchD , jumpD ,
4. input logic [2:0]   alucontrolE ,
5. input logic         regwriteE , memtoregE , alusrcE , regdstE ,
6. input logic [31:0]  readdataM ,
7. input logic         regwriteM , memtoregM ,
8. input logic         regwriteW , memtoregW ,
9. output logic [31:0] pcF ,
10. output logic [5:0]  opD , functD ,
11. output logic        equalD ,
12. output logic        flushE ,
13. output logic [31:0] aluoutM , writedataM ,
15. input logic immextE,
16. input logic bneD);
18. logic          stallF , flushD ,stallD;
19. logic          forwardaD , forwardbD ;
20. logic [1:0]    forwardaE , forwardbE ;
21. logic [4:0]    rsD , rtD , rdD , rsE , rtE , rdE ;
22. logic [4:0]    writeregE , writeregM , writeregW ;
23. logic [31:0]   pcnextFD , pcnextbrFD ,pcplus4F, pcbranchD ;
24. logic [31:0]   signimmD , signimmE , signimmshD ,zeroD ,zeroE;
25. logic [31:0]   srcaD ,srca2D, srcaE ,srca2E;
26. logic [31:0]   srcbD ,srcb2D, srcbE ,srcb2E,srcb3E;
27. logic [31:0]   pcplus4D, instrD ;
28. logic [31:0]   aluoutE , aluoutW ;
29. logic [31:0]   readdataW , resultW ;

32. // hazard detection
33. hazard h(  rsD,rtD,rsE,rtE,
34. writeregE , writeregM , writeregW ,
35. regwriteE , regwriteM , regwriteW,
36. memtoregE , memtoregM ,
37. branchD   , bneD,
38. forwardaD , forwardbD , forwardaE , forwardbE ,
39. stallF ,stallD, flushE );

42. // next PC logic ( operates in fetch and decode )
43. mux2#(32) pcbrmux   (pcplus4F, pcbranchD , pcsrcD , pcnextbrFD );
44. mux2#(32) pcmux     ( pcnextbrFD ,{pcplus4D[31:28], instrD [25:0],2'b00}, jumpD , pcnextFD );
46. // register file ( operates in decode and writeback )
47. regfile rf ( clk , regwriteW , rsD , rtD , writeregW , resultW , srcaD , srcbD );
49. // Fetch stage logic
50. flopenr #(32)  pcreg (clk, reset ,~stallF , pcnextFD , pcF );
51. adder          pcadd1( pcF ,32'b100,pcplus4F);
53. // Decode stage
54. flopenr    #(32)   r1D( clk , reset ,~stallD ,pcplus4F,pcplus4D);
55. flopenrc   #(32)   r2D( clk , reset ,~stallD, flushD , instrF , instrD );
56. signext            se ( instrD [15:0], signimmD ,zeroD);
57. sl2                immsh ( signimmD , signimmshD );
58. adder              pcadd2(pcplus4D, signimmshD , pcbranchD );
59. mux2#(32)          forwardadmux ( srcaD , aluoutM , forwardaD ,srca2D);
60. mux2#(32)          forwardbdmux ( srcbD , aluoutM , forwardbD ,srcb2D);
61. eqcmp              comp(srca2D,srcb2D, equalD );
63. assign opD     = instrD [31:26];
64. assign functD  = instrD [5:0];
65. assign rsD     = instrD [25:21];
66. assign rtD     = instrD [20:16];
67. assign rdD     = instrD [15:11];
68. assign flushD  = pcsrcD |jumpD ;
70. // Execute stage
71. logic [31:0] imm;
73. floprc #(32)   r1E( clk , reset , flushE , srcaD ,     srcaE );
74. floprc #(32)   r2E( clk , reset , flushE , srcbD ,     srcbE );
75. floprc #(32)   r3E( clk , reset , flushE , signimmD ,  signimmE );
76. floprc #(32)   r7E( clk , reset , flushE , zeroD ,  zeroE );
77. floprc #(5)    r4E( clk , reset , flushE , rsD ,       rsE );
78. floprc #(5)    r5E( clk , reset , flushE , rtD ,       rtE );
79. floprc #(5)    r6E( clk , reset , flushE , rdD ,       rdE );
80. mux3#(32)      forwardaemux ( srcaE , resultW , aluoutM , forwardaE ,srca2E);
81. mux3#(32)      forwardbemux ( srcbE , resultW , aluoutM , forwardbE ,srcb2E);
83. mux2#(32)      extmux(zeroE,signimmE,immextE,imm);
84. mux2#(32)      srcbmux (srcb2E, imm , alusrcE ,srcb3E);
86. alu            alu(srca2E,srcb3E, alucontrolE , aluoutE );
87. mux2#(5)       wrmux ( rtE , rdE , regdstE , writeregE );
89. // Memorry stage
90. flopr #(32)    rlM ( clk , reset , srcb2E,     writedataM );
91. flopr #(32)    r2M ( clk , reset , aluoutE ,   aluoutM );
92. flopr #(5)     r3M ( clk , reset , writeregE , writeregM );
94. // Writeback stage
95. flopr #(32) rlW    ( clk , reset , aluoutM , aluoutW );
96. flopr #(32) r2W    ( clk , reset , readdataM , readdataW );
97. flopr #(5)  r3W    ( clk , reset , writeregM , writeregW );
98. mux2  #(32) resmux ( aluoutW , readdataW , memtoregW , resultW );
99. endmodule

不难看出，数据通路的设计其实同在单周期中没有太大区别，最显著的差异来源于非状态元件，即流水段寄存器的引入。

**４． 控制模块**

为了根据和字段计算控制信号我们需要设计一个控制单元。

主译码器与ALU译码器同单周期中设计类似，这里不再赘述，我们在控制模块中将直接使用它们。不同的是，我们需要为不同的流水段保存对应的控制信号，这需要我们在控制模块中加入寄存器。

控制模块包含了输入的控制字段和标志，在处理后为各流水段输出对应的控制信号。

1. module controller ( input logic clk,reset,
2. input logic [5:0] opD , functD ,
3. input logic equalD,
4. input logic flushE,
5. output logic pcsrcD , branchD , jumpD ,
6. output logic [2:0] alucontrolE ,
7. output logic regwriteE , memtoregE , alusrcE , regdstE ,
8. output logic regwriteM , memtoregM , memwriteM ,
9. output logic regwriteW , memtoregW ,
11. output logic immextE,
12. output logic bneD);
14. logic [2:0]    alucontrolD;
15. logic          regwriteD , regdstD ;
16. logic          alusrcD ;
17. logic          memwriteD , memtoregD ;
18. logic          memwriteE ;
19. logic [2:0]    aluopD ;
21. logic immextD;
23. maindec md(opD,
24. regwriteD,regdstD,//output
25. alusrcD,branchD,
26. memwriteD,memtoregD,
27. jumpD,aluopD,
28. bneD,immextD);
30. aludec ad(functD,aluopD,
31. alucontrolD);
33. assign pcsrcD=((branchD&equalD)|(bneD&~equalD));
35. // pipeline registers
36. floprc #(9) regE ( clk , reset , flushE ,
37. { memtoregD , memwriteD , alusrcD , regdstD , regwriteD , alucontrolD ,immextD},
38. { memtoregE , memwriteE , alusrcE , regdstE , regwriteE , alucontrolE ,immextE});
40. flopr #(3) regM (  clk , reset ,
41. { memtoregE , memwriteE , regwriteE },
42. { memtoregM , memwriteM , regwriteM });
44. flopr #(2) regW (  clk , reset ,
45. { memtoregM , regwriteM },
46. { memtoregW , regwriteW });
47. endmodule

可以看出，总的来说仍旧与单周期处理器类似，进行的就是协调主译码器和ALU译码器产生控制信号，不同在于为对应流水段保存对应信号。

**5． 冲突处理模块**

在分析得到流水线处理器中可能产生的冲突后我们需要设计一个模块来处理这些冲突。

为正确预判是否需要转发，如何转发，我们需要的信息包括：执行阶段的两个源寄存器地址，存储和写回阶段的目的寄存器地址以及执行、存储、写回阶段的regwrite信号。这将产生转发信号。

对于阻塞，我们还需要知道是否发生分支，以及是否要读存储器。这将产生阻塞相关的控制信号。

具体来说就可以像下面这样。

1. module hazard ( input logic [4:0]   rsD , rtD , rsE , rtE ,
2. input logic [4:0]   writeregE , writeregM , writeregW ,
3. input logic         regwriteE,regwriteM,regwriteW,
4. input logic         memtoregE , memtoregM , branchD ,bneD,
5. output logic        forwardaD , forwardbD,
6. output logic [1:0]  forwardaE , forwardbE ,
7. output logic        stallF ,stallD, flushE );
9. logic lwstallD,branchstallD;
11. // forwarrding sources to D stage ( branch equality )
12. assign forwardaD =( rsD !=0& rsD == writeregM & regwriteM );
13. assign forwardbD =( rtD !=0& rtD == writeregM & regwriteM );
15. // forwarding sources to Estage ( AL )
16. always\_comb  begin
17. forwardaE =2'b00; forwardbE =2'b00;
18. **if** ( rsE !=0)
19. **if** ( rsE == writeregM & regwriteM ) forwardaE =2'b10;
20. **else**
21. **if** ( rsE == writeregW & regwriteW ) forwardaE =2'b01;
23. **if** ( rtE !=0)
24. **if** ( rtE == writeregM & regwriteM ) forwardbE =2'b10;
25. **else**
26. **if** ( rtE == writeregW & regwriteW ) forwardbE =2'b01;
27. end
29. //stalls
30. assign lwstallD = memtoregE &( rtE == rsD | rtE == rtD );
31. assign branchstallD = (branchD|bneD) &
32. ( regwriteE &( writeregE == rsD | writeregE == rtD ) |
33. memtoregM  &( writeregM == rsD | writeregM == rtD ));
34. assign stallD =lwstallD | branchstallD ;
35. assign stallF = stallD ;// stalling D stalls all previous stages
36. assign flushE = stallD ; //stalling D flushes next stage
37. endmodule

**6． 顶层模块**

在完成了上面这些设计后，我们只需要设计一个模块包含数据通路和控制模块，再设计一个顶层模块包含模块和指令存储器模块就可以了。比如下面这样。

1. module MIPS(    input logic clk,reset,
2. input logic [31:0] instrF , readdataM ,
3. output logic [31:0] pcF ,
4. output logic memwriteM,
5. output logic [31:0] aluoutM ,
6. output logic [31:0] writedataM );
7. logic [5:0]    opD , functD ;
8. logic          pcsrcD , equalD ;
9. logic          branchD,jumpD;
10. logic [2:0]    alucontrolE ;
11. logic          memtoregE , regwriteE , regdstE , alusrcE , flushE ;
12. logic          memtoregM , regwriteM ;
13. logic          memtoregW , regwriteW ;
14. logic          bneD,immextE;
16. controller c ( clk , reset ,
17. opD , functD , equalD ,
18. flushE ,
19. pcsrcD , branchD , jumpD ,// output
20. alucontrolE ,// output
21. regwriteE , memtoregE , alusrcE , regdstE ,// output
22. regwriteM , memtoregM , memwriteM ,// output
23. regwriteW , memtoregW,
24. immextE,bneD);// output
26. datapath dp (  clk , reset ,
27. instrF ,
28. pcsrcD , branchD , jumpD ,
29. alucontrolE ,
30. regwriteE , memtoregE , alusrcE , regdstE ,
31. readdataM ,
32. regwriteM , memtoregM ,
33. regwriteW , memtoregW ,
34. pcF ,                   // output
35. opD , functD , equalD , // output
36. flushE ,                // output
37. aluoutM , writedataM ,// output
38. immextE,
39. bneD);
41. endmodule

除了为各个流水段标识信号外，MIPS层面上它同单周期处理器是类似的。

1. module Top(input logic clk,
2. input logic reset,
3. output logic [31:0] writedata ,
4. output logic [31:0] dataadr ,
5. output logic memwrite);
6. logic [ 31:0 ] pc,instr,readdata;
7. MIPS mips(  clk, reset,
8. instr,readdata,
9. pc, memwrite, dataadr, writedata);
10. iMem imem (pc [7:2 ] ,instr );
11. dMem dmem (clk,memwrite, dataadr ,writedata,readdata);
12. endmodule

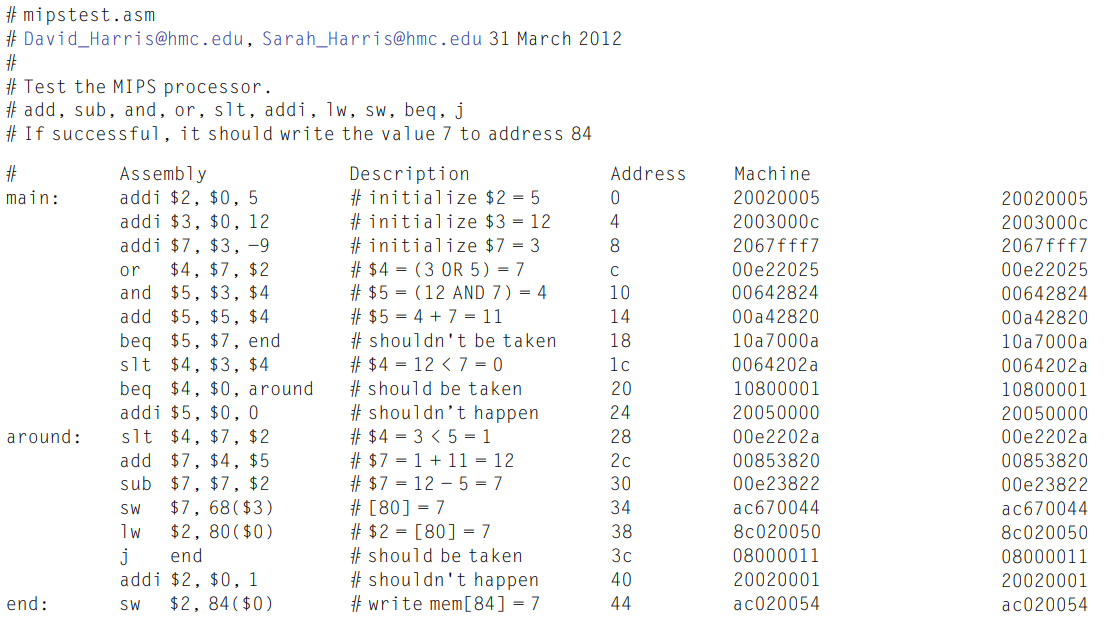
只看顶层文件我们甚至无法将其与单周期处理器区分开。

**６. 仿真**

在完成设计后我们就可以设计仿真了。为了验证指令可以正确执行，我们可以在指令存储器里写一段程序加以测试。

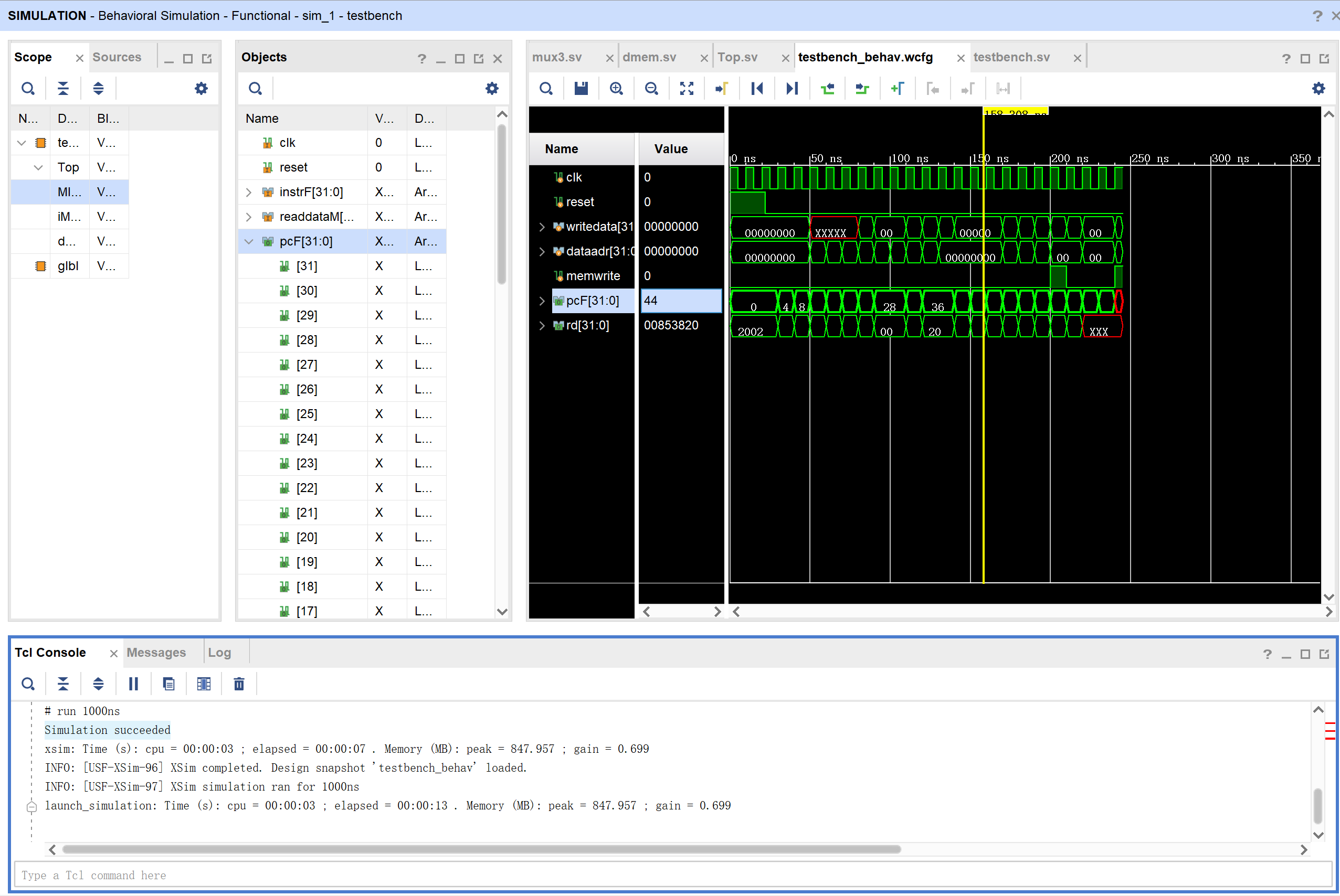
1. 未扩展的处理器

使用与单周期处理器相同的代码配合相同的测试程序可以测试前几条指令的功能。



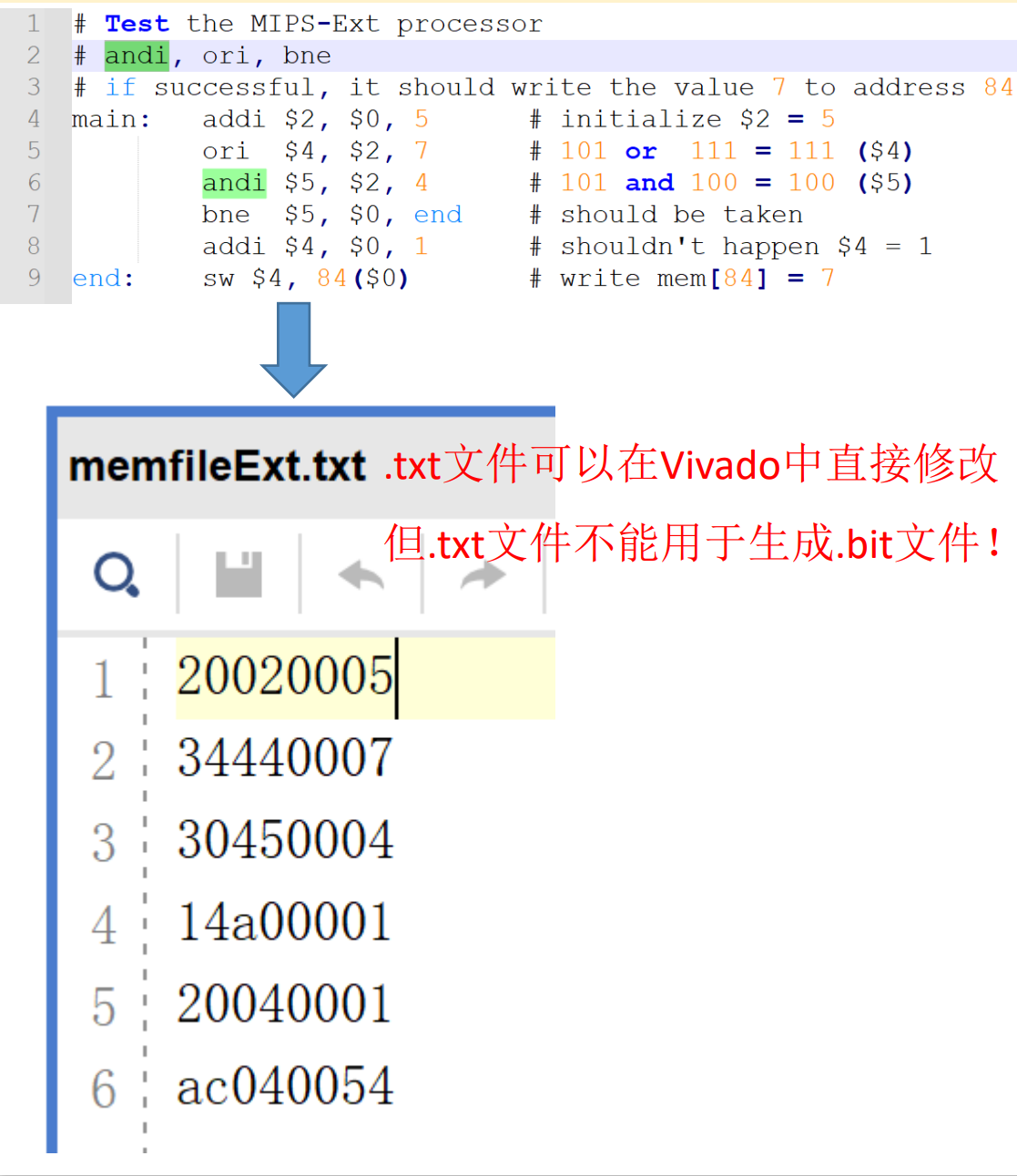
1. module testbench();
2. logic clk;
3. logic reset;
4. logic [31:0] writedata, dataadr;
5. logic memwrite;
6. // instantiate device to be tested
7. Top dut (clk, reset, writedata, dataadr, memwrite);
8. // initialize test
9. initial
10. begin
11. reset <= 1; # 22; reset <= 0;
12. end
13. // generate clock to sequence tests
14. always
15. begin
16. clk <= 1; # 5; clk <= 0; # 5;
17. end
18. // check results
19. always @(negedge clk)
20. begin
21. **if** (memwrite) begin
22. **if** (dataadr === 84 & writedata === 7) begin
23. $display("Simulation succeeded");
24. $stop;
25. end
27. **else** **if** (dataadr !== 80) begin
28. $display("Simulation failed");
29. $stop;
30. end
31. end
32. end
33. endmodule

于是就可以得到这样的仿真结果。仿真成功。

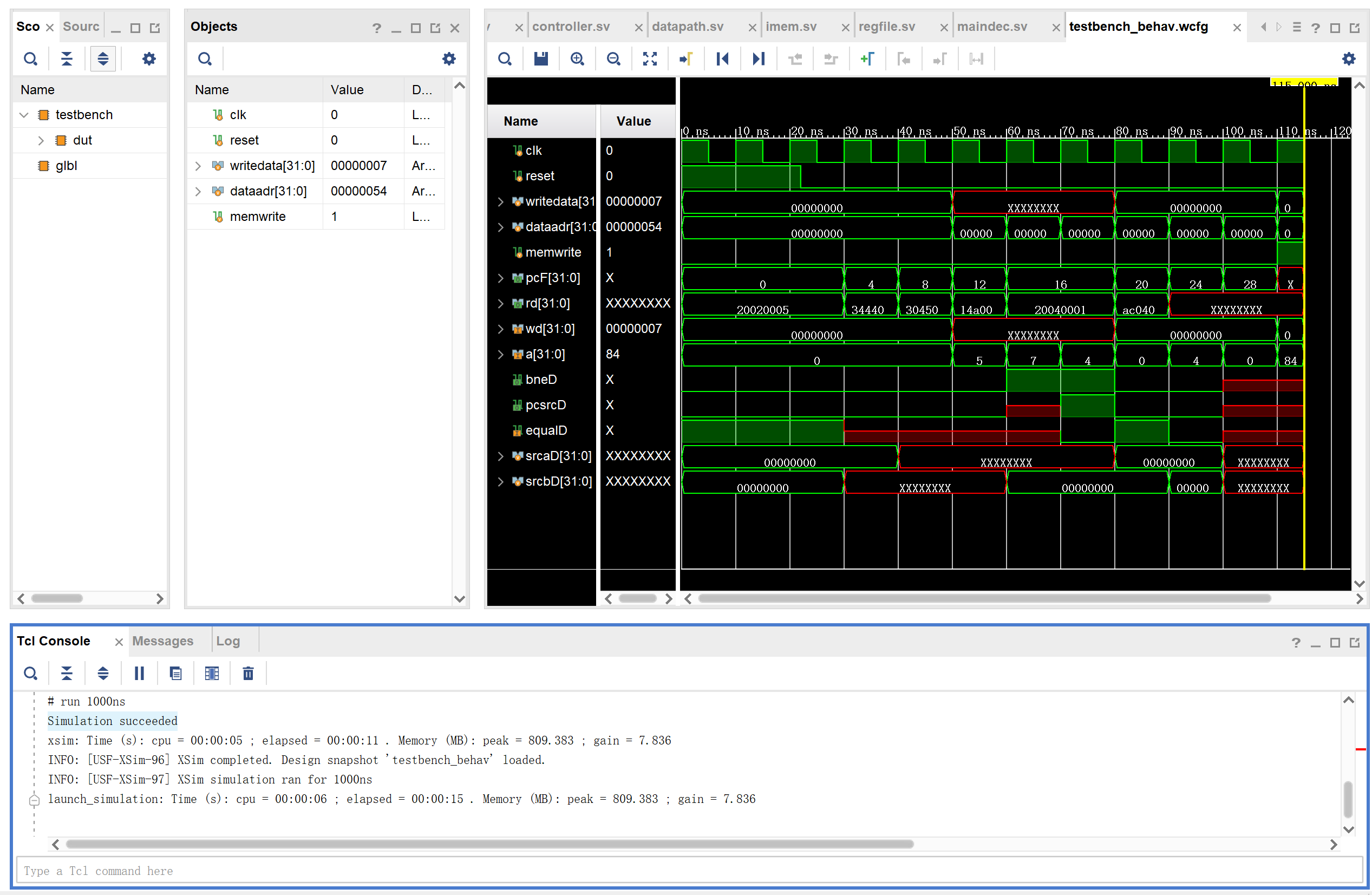


1. 扩展

在设计过程中我们就已经加入了ｊ指令、指令和立即数运算指令的控制信号，我们的扩展模块也支持０扩展。所以我们只需要另外设计程序就可以了。

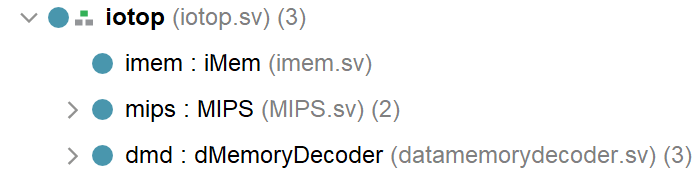


同样可以得到预期结果。



**７. IO接口设计**

由于我们之前设计已经实现了大多数模块，下面我们只需要设计一个新模块，并设计新的顶层模块就可以了。事实上这些模块和在单周期处理器中没有什么区别，只需要对应地接入接口就可以了。

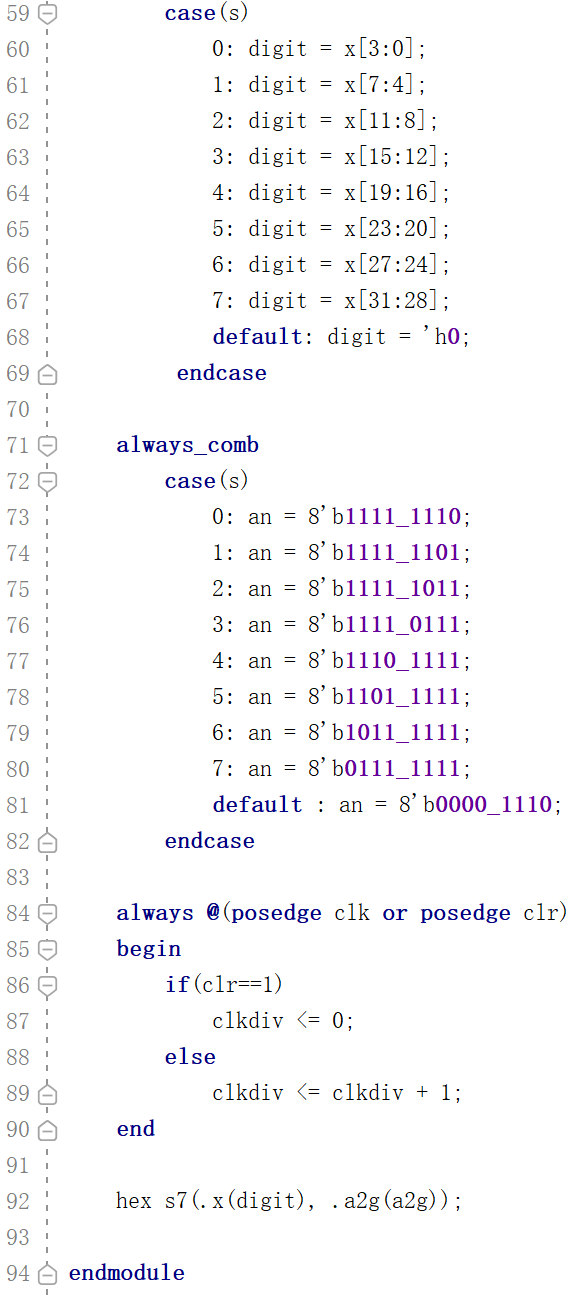
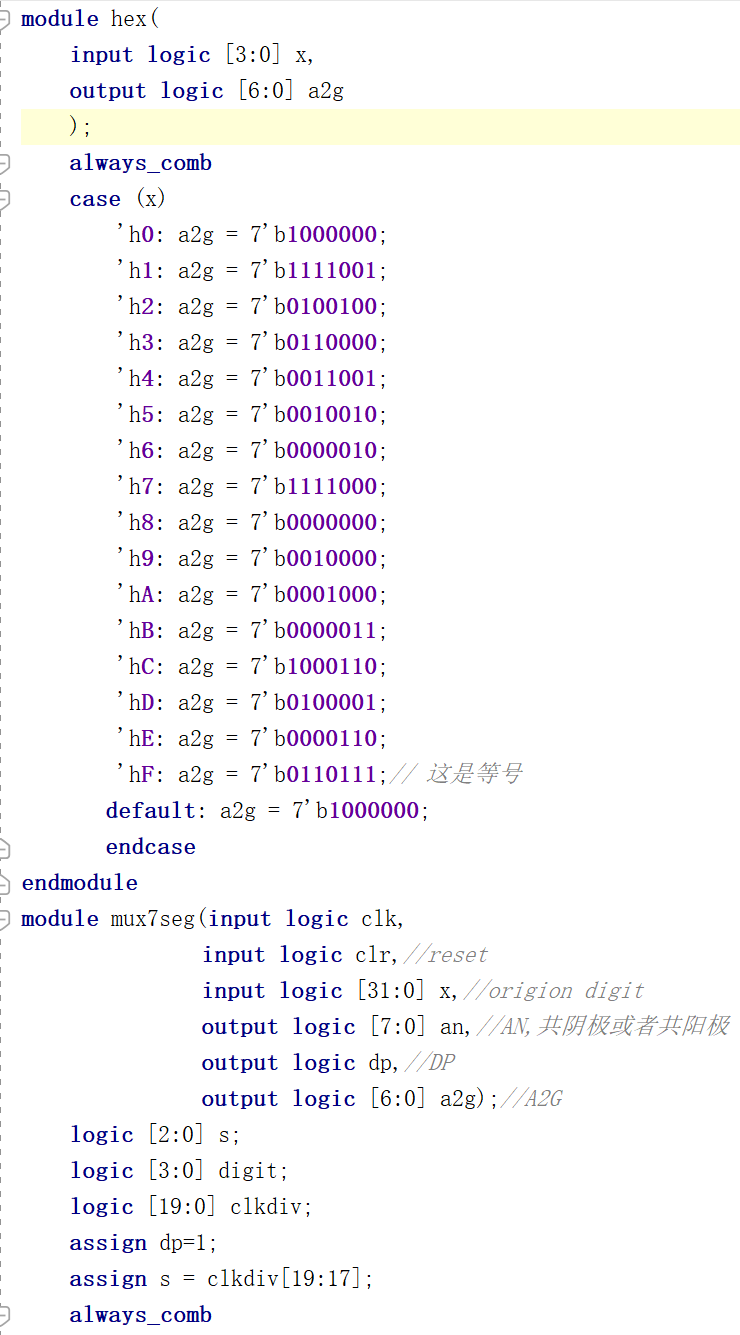


在原来数据存储器上加上附加的模块，使之能够支持IO操作。具体来说，我们需要一个额外的IO模块、一个七段数字管模块，以及一个控制模块。这同单周期处理器相同。

1. module IO(input logic clk,
2. input logic reset,
3. input logic pRead,
4. input logic pWrite,
5. input logic [1:0] addr,
6. input logic [11:0] pWriteData,
7. output logic [31:0] pReadData,
8. input logic buttonL,//LED输出
9. input logic buttonR,//Switch输入
10. input logic [15:0] **switch**,//直接显示
11. output logic [11:0] led);
13. logic [1:0] status;
14. logic [15:0] switch1;
15. logic [11:0] led1;
17. always\_ff@(posedge clk) begin
18. **if**(reset) begin
19. status <=  2'b00;
20. led1   <= 12'h00;
21. switch1 <= 16'h00;
22. end
24. **else** begin
25. //开关位置已经拨好，可以输入新数据
26. **if**(buttonR) begin
27. status[1] <= 1;
28. switch1 <= **switch**;
29. end
30. //LEDs已经准备好，可以输出新数据
31. **if**(buttonL) begin
32. status[0] <= 1;
33. led <=led1;
34. end
36. //向数据输出端口输出(LED)
37. **if**(pWrite & (addr==2'b01)) begin
38. led1 <= pWriteData[11:0];
39. status[0] <= 0;
40. end
42. end //if
43. end //always\_if
45. //读数据
46. always\_comb
47. **if**(pRead)
48. //11:数据输入端口(高)， 10:数据输入端口(低)
49. //01:数据输出端口(LED)， 00:状态端口
50. **case**(addr)
51. 2'b11: pReadData = {24'b0, **switch**[15:8]};
52. 2'b10: pReadData = {24'b0, **switch**[7:0]};
53. 2'b00: pReadData = {24'b0, 6'b000000, status};
54. **default**: pReadData = 32'b0;
55. endcase
56. **else**
57. pReadData=32'b0;
59. endmodule

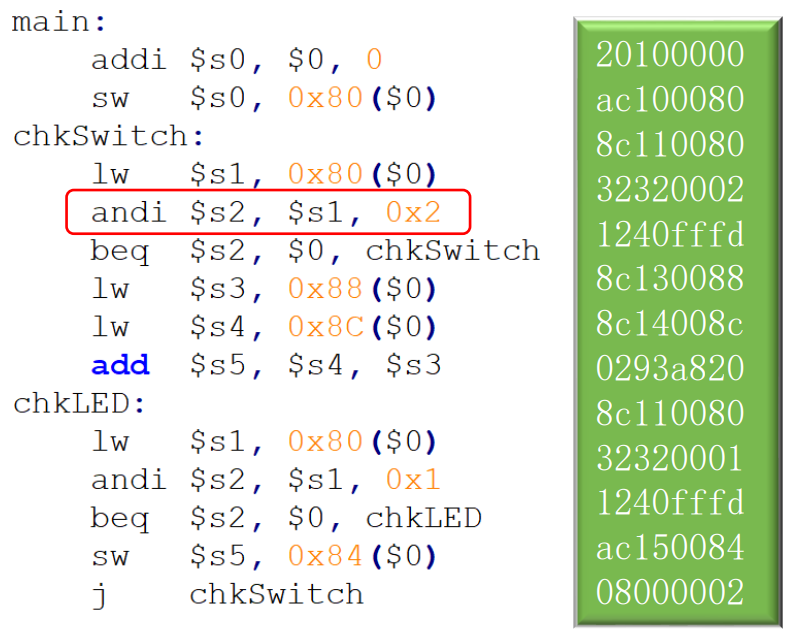
它可以从外界读取数据将之输出，也可以输出数据。

而七段数字管模块可以让输出结果显示在LED灯上。

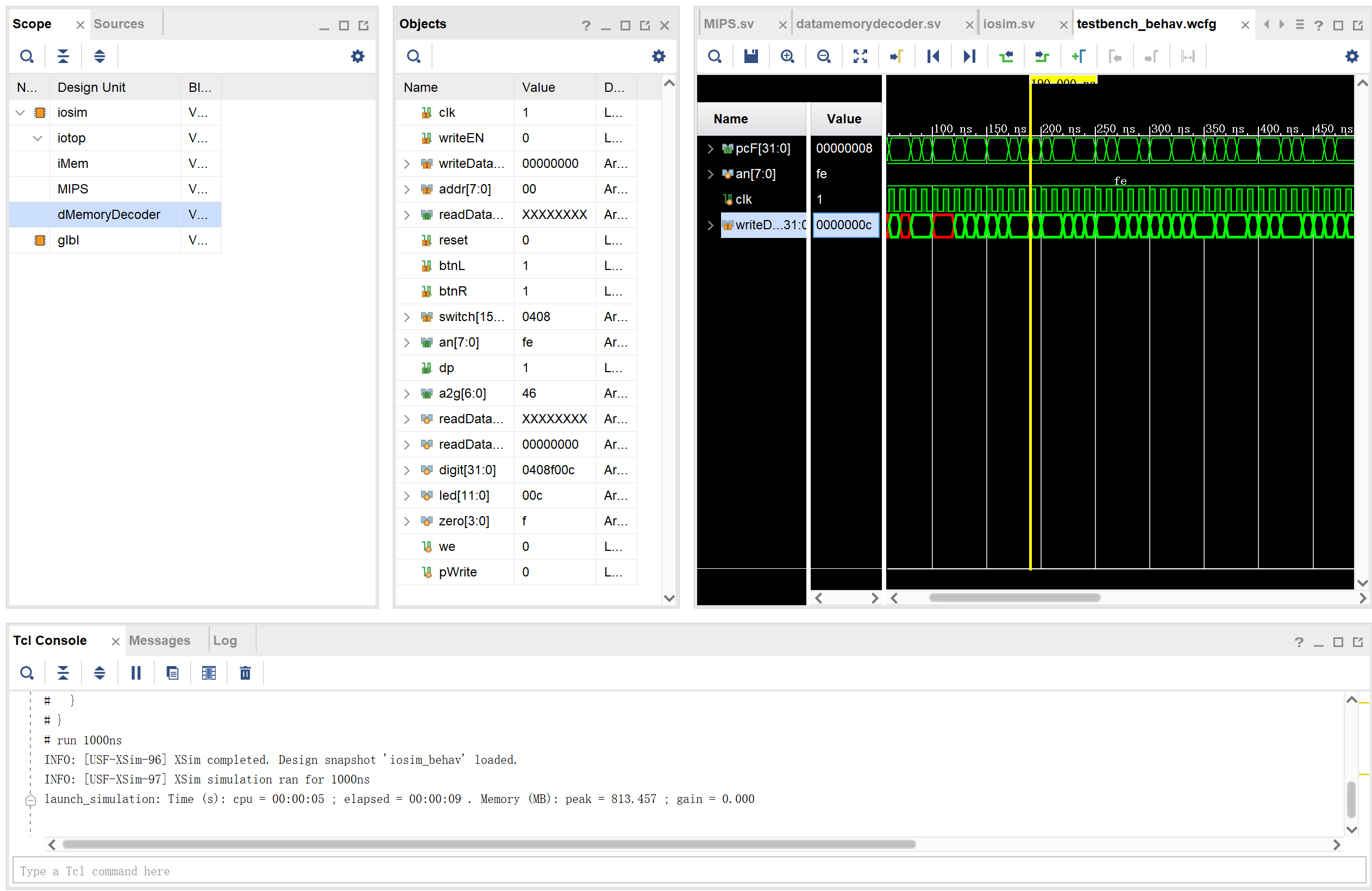


再加上控制模块，我们就实现了IO接口的设计。

在程序存储器中写入下面的程序。



这里我们可以首先进行模拟。这里以计算4+8为例，进行16进制加法。



下面我们生成二进制文件，写入开发板。

单击中间按钮(系统清零)，再单击左按钮(LED输出)，则清零； 单击右侧按钮(开关输入)，再单击左按钮(LED输出) ，显示相加结果。最终效果如下。



# **五、思考与收获**

**１. 精巧的指令集**

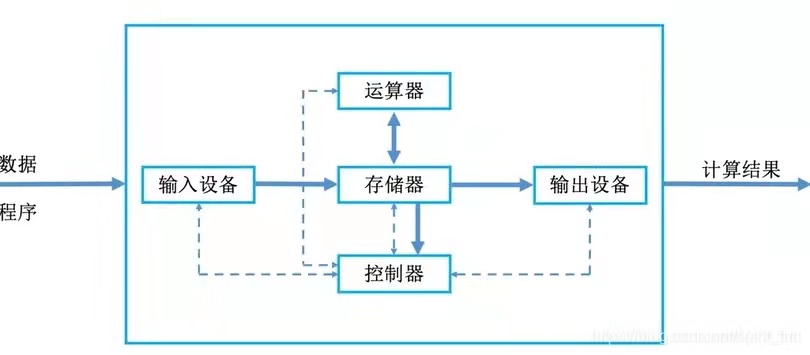
指令系统的设计设计是计算机系统设计的关键之一。从本次的设计中可以发现，一个精巧的指令集可以让设计变得更加简单而高效。

考察我们的R型指令。它有着两个控制字段，但是它们分别处于指令的最高六位和最低六位，而非连续的十二位。这一设计似乎反常，但却是很有意义的。因为我们还要实现其他类型的指令，而这些类型的指令只有前六位是控制字段，接下来的五位在很多指令中都是作为操作数的寄存器，所以R型指令也按照六位控制字段加寄存器的模式设计可以免去对寄存器地址的复用。另外，虽然我们可以把所有指令都使用统一长度的控制字段，但这样会使得有些指令位数不够，而有些又位数太多。由于R型指令操作数只有三个五位的寄存器，所以我们为它选用了共十二位的控制字段和五位移位字段，这充分利用了其长度，又为其他指令留下空间。

类似的，ｓｗ指令并没有把目的操作数放在第一个，而是采取类似ｌｗ指令的方式。我们可以设想，如果ｓｗ指令依照惯例将目的操作数写在前面，我们将不得不加上很多电路和多路器，需要产生更多控制信号。而由于这一反常设计，这两个操作所使用的电路在很多地方都是相同，电路设计得以简化。

这样的例子还有很多，而这些精巧的设计无疑让我们的设计更简单、更精巧，这不仅可以加快速度，还可以减少不必要的错误。

**２. 以存储器为核心**

在设计中，我们采取的策略是从包含状态原件的硬件开始进行设计的模式。这一模式同以存储器为核心的思想不谋而合。

**3. 体系结构、微体系结构与逻辑**

微体系结构处于体系结构和逻辑之间。一个特定的体系结构，比如现在和之前我们实现的MIPS处理器，它们可能有不同的微体系结构，它们是性能、成本和复杂性的不同折中。它们可以运行相同的程序，但是内部设计却有着很大差异。

具体到我们这里设计的流水线处理器和之前设计的单周期处理器。

一方面，它们都属于MIPS处理器，有着相同的体系结构。因而在设计中我们不论是在位宽、寄存器个数，还是在指令格式、寻址方式上都实现的是统一体系。为了解决单周期处理器存在的不足，我们着手设计的流水线处理器为了换取性能和一些成本，牺牲了复杂性。在仿真和上板验证中，我们可以使用相同的代码和测试程序，我们的处理器对外使用一致的接口，它们最后都可以得到正确的结果，这说明，不同的实现虽然存在差异，但在更高的抽象层次看来它们都是相同的，它们的差异是可以不做考虑的，或者说它们的具体实现被封装了起来。

另一方面，我们可以看出微体系结构承上启下的重要地位。其上就是体系结构，其下则是逻辑。它处于硬件和软件之间，它的实现既需要考虑更上层的需求，又要兼顾下层的实现。

**4. 扩展已有的设计**

不同于设计多周期处理器，在设计流水线处理器时我们并没有直接着手构建流水线处理器，而是在考虑到其同单周期处理器的相似性后直接在单周期处理器的基础上进行扩展。这一方式使得我们的设计更为容易，也更容易注意到不同实现间的相似于差异。

在实现中我们同样大量复用了在单周期处理器中所使用的模块，更重要的时我们采用了和单周期处理器相同的框架。我们将处理器分为数据通路、控制模块和存储器分别实现，这样在其中的一些部分发生变化时其他部分可以不加改动，也就是说它们之间是协同工作的，但也是相互独立的。因而即使可行，我们也不会保存将控制信号的寄存器和保存数据的寄存器合并，因为这样可以使得数据通路同控制模块分离。

**5. 性能、成本与复杂程度的折中**

分析我们设计的单周期处理器、多周期处理器和流水线处理器，我们不难发现我们所做的就是在性能、成本和复杂程度间取得更好的折中。

单周期处理器是相对来说直观的，易于实现的。虽然性能有很多待挖掘的空间，但是它比较简单。

多周期处理器对单周期处理器的优化基于两个出发点。一个是提高性能，为此划分了指令执行的周期；另一个是降低成本，为此使用了统一存储器和唯一一个ALU单元。同时提高了复杂程度上的代价。

流水线处理器优化单周期处理器更多基于性能考量。多级流水成倍地提高了处理器吞吐，这使得其他优化都应该服从于性能的提高。由于要求简单的结构，而且统一存储器在流水线中会存在问题，因而这些在多周期处理器中做出的优化被暂时让位给了性能。

**6. 收获**

在本次实验中，我们设计的处理器有着鲜明的结构层次。从更低的抽象等级向更高的抽象等级看，可以更好的理解原件间的相互作用，协同工作；从更高的抽象层次向更低的抽象层次看，可以了解自向下的设计思想，了解体系结构的精妙之处。