

|  |
| --- |
| 32位MIPS多周期处理器设计 |

李晓畅 | 20307130261 | 2022年4月17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：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：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：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：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：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。

因而对于单周期CPU,就是在一个时钟周期内完成这五个阶段的处理。

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. 单周期处理器的不足与多周期处理器对单周期处理器的改进**

单周期处理器有三个主要缺点。第一，它需要足够长的周期来完成最慢的指令，即指令，这将会拖慢大部分实际上很快的指令。第二，它需要三个加法器，分别用于ALU，和PC的逻辑，而加法器是相对占用芯片面积的电路，尤其是如果它们的速度比较快。第三，它采用独立的指令存储器和数据存储器，也就是说采用的是哈佛模型，但这在实际系统中是不现实的。大多数计算机有一个单独的大容量存储器来存储指令和数据，并且支持读写操作。

针对这些存在的问题，我们在多周期处理器中设计了一些对应的改进，而其核心是将指令执行过程分解为多个较短的步骤。在每个短步骤中，处理器可以读或写存储器或寄存器文件，或使用ALU。由于不同的指令可以使用不同的步骤，长指令不再会拖累系统的运行速度，简单的指令得以执行得更快，这解决了第一个问题。同时，由于段步骤的引入，加法器可以在不同的步骤中执行不同的任务，从而实现重复使用，这样我们只需要设计一个ALU中的加法器就可以应对所有情况了，这解决了第二个问题。最后，我们要求只使用一个可以存储指令和数据的组合存储器，为此我们会在第一个段步骤中从其中读出指令，而在后面的步骤中读或写入数据。这其实是同多个短步骤这一核心理念相契合的，不加区分的数据和指令实际上使得每一条短步骤执行的都是类似的任务，这反而简化了设计和实现的复杂程度。

**5. 指令执行过程**

在单周期处理器中，我们实际上实现的最小单元是单条指令。而在多周期处理器中，我们实现的单元缩小到了单个指令中的一些短步骤。为此我们需要了解指令的具体实现过程，大致就是如下的五个步骤：

1. 取指令并计算下一条指令地址。从PC指出的内存单元中取出指令送到指令寄存器，同时计算下一条指令的地址并将其送PC。
2. 对指令操作译码。不同指令的功能不同，设计的操作过程也不同，因而需要不同的操作控制信号。
3. 计算操作数地址并取源操作数。根据指令的不同，可能会需要数次访存才可以取出所有的操作数并传到目的地址。
4. 数据操作。在ALU等运算部件中对取出的操作数进行运算。
5. 目的操作数地址计算并存结果。根据寻址方式确定的操作数地址计算方式算出的目的地址，经过数次访存存储指令的结果。

# **三、实验方案**

**1. 实验目的**

1）设计实现32位MIPS多周期处理器

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。

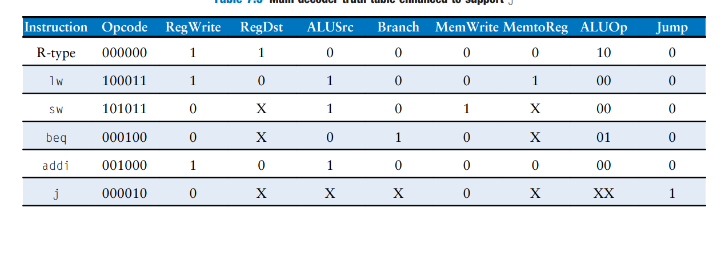


**３. 设计过程**

1）数据路径设计

正如前面提到的那样，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。

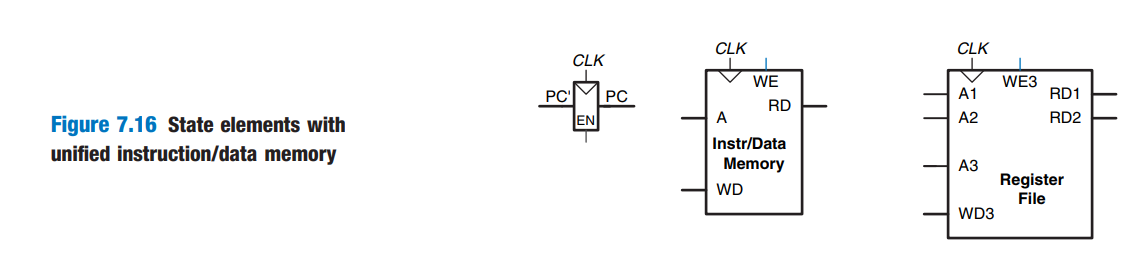


确定了需要实现的指令集，再加上一些必要的模块，我们接下来就可以考虑数据通路的设计和控制逻辑的设计了。

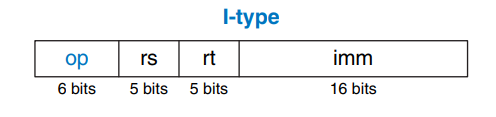
设计复杂系统时，一种好方法是从包含状态原件的硬件开始，而这些原件中就包括存储器和体系结构状态。然后在这些存储组件之间增加组合逻辑基于当前状态计算新状态。从部分存储器中读指令，然后装入和存储指令从一部分存储器读或写数据。

但同单周期处理器不同的是，为了实现多周期处理器，我们需要在下一个短步骤中继续之前的任务。为此，我们将增加非体系结构状态原件来保存中间结果，具体来说我们可以在每一个步骤结束的节点出增加寄存器原件以实现状态的保存与传递。

基于这一设计思想，在分析指令功能及所需要的组件后，我们可以从下面这些原件开始考虑。可以看到，与单周期处理器中不同，我们这里将之前使用的单独的数据存储器与指令存储器合并为统一的存储器。



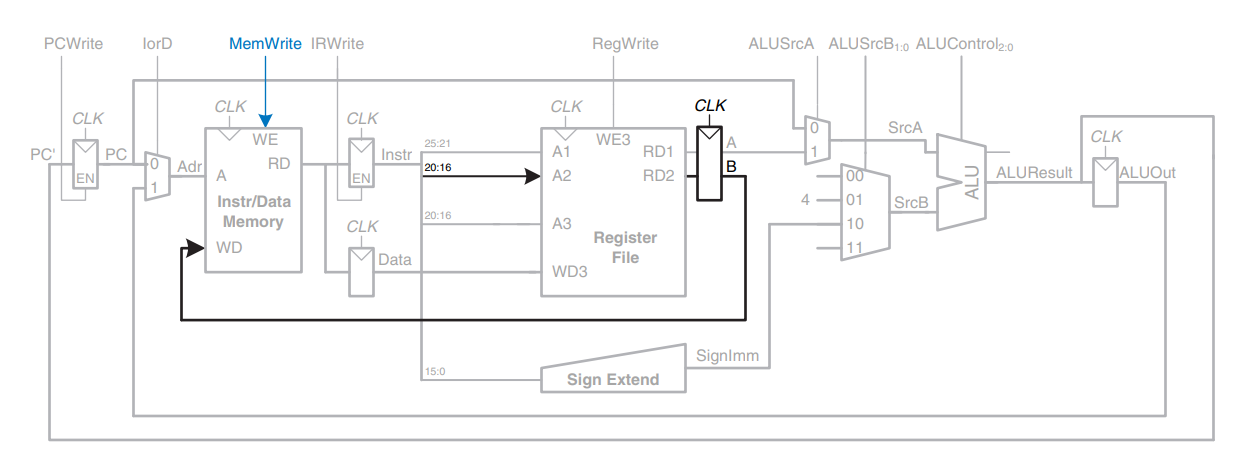
首先我们可以着手实现，指令构造数据路径，然后考虑泛化该数据路径以实现更多指令。

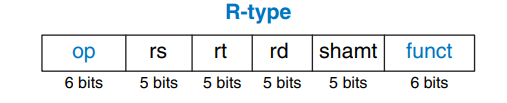
、指令就是一条I类型指令，有着这样的结构。

具体来说，op字段决定指令的操作。字段具体代表源操作数或目的操作数由具体指令决定。对于指令，rt是目的寄存器，而和是源操作数。

一方面，类似于单周期处理器设计中那样，我们进行如下分析：可以看出，立即数位数只有十六位，所以需要扩展，也就是要求添加扩展原件。内存地址将由一个基地址和一个变地址共同决定，因而需要一个原件实现运算，也就是需要算数逻辑单元ALU。另外，在执行完一条指令后，我们希望继续执行下一条指令，所以需要改变PC的值，具体到32位处理器，我们要将PC加４。

另一方面，基于多周期处理器的特点，我们还需要分析指令的执行转态，并规划如何划将其合适地进行划分。参考之前叙述的指令执行步骤，将取指令并将PC增加作为一个单独的状态加以实现应该是合理的选择，因为它会在所有指令中出现，又可以重复使用ALU。译码阶段也是必要的，这里我们可以先假设在这一阶段我们只等待译码。指令需要使用一次ALU用于计算存储器中的地址。另外还需要一个寄存器操作数。这两个阶段应该分别实现。接下来我们只需要分别实现各个短步骤，并在短步骤的节点处添加寄存器保存状态就可以了。于是类似与单周期处理器，我们可以得到下面的设计。



然后是R类型的指令实现。一条32位的R类型指令是这样的。

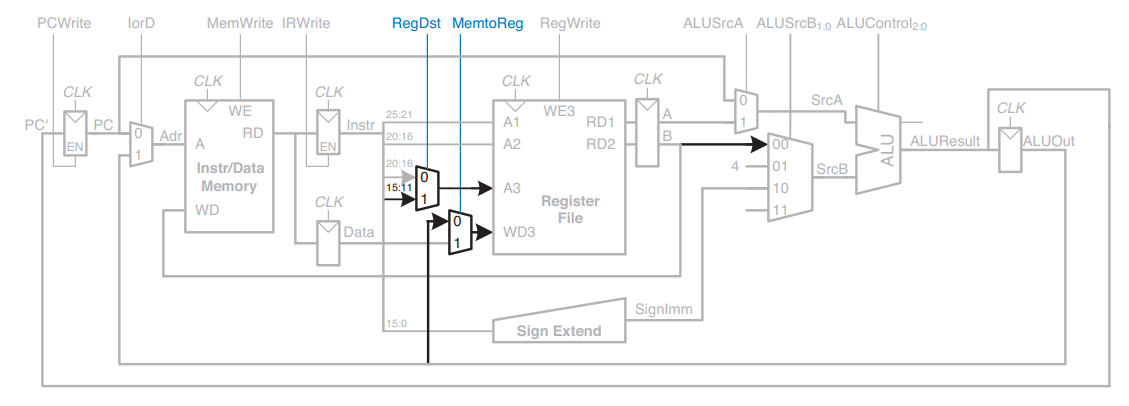
具体来说，指令操作码包括：

指令操作编码为op和两个字段各六位，其中对R类型指令而言op总为0，而决定R操作的类型。

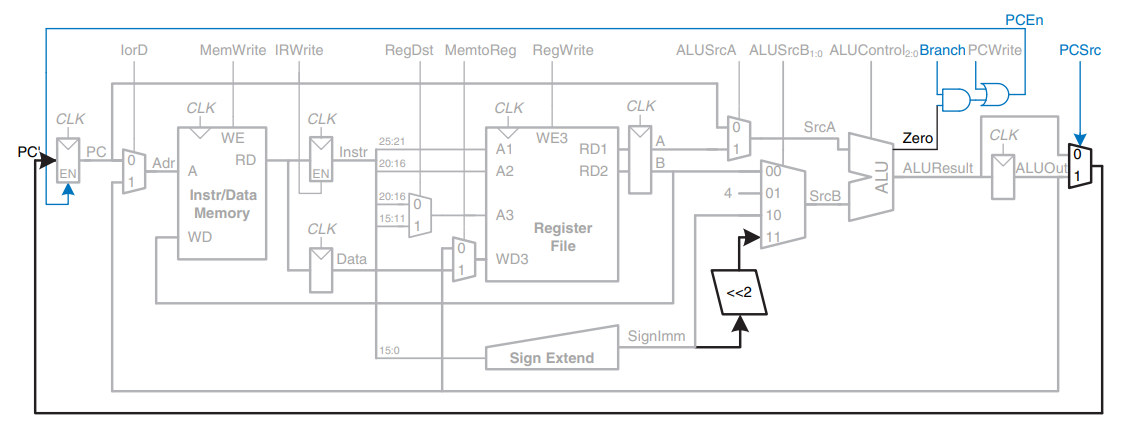
指令操作数编码为三个字段，即。前两个作为源寄存器，而最后一个是目的寄存器。

仅用于移位操作，以五位二进制数表示移位位数，而在执行其他指令时为0。

考察与之前设计电路的差别，我们发现精妙的设计使得很多通路都是相同的，只有很少的一些路线有所变化。R型指令的执行过程同指令基本是一致的。具体来说，相较于指令，我们只需要一个另外的目标寄存器地址，一个另外的写寄存器数据源和一个另外的ALU源操作数就可以了。因而我们可以添加多路器和控制信号就可以实现R类型的指令了。

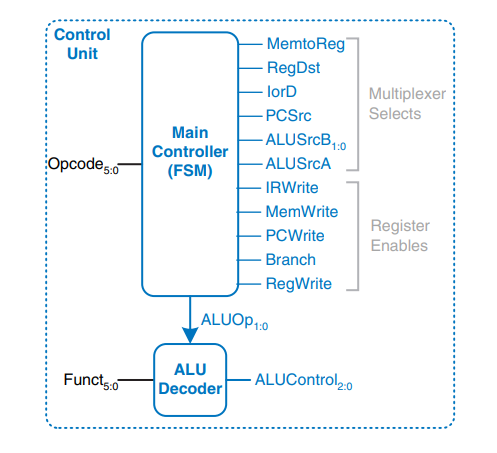
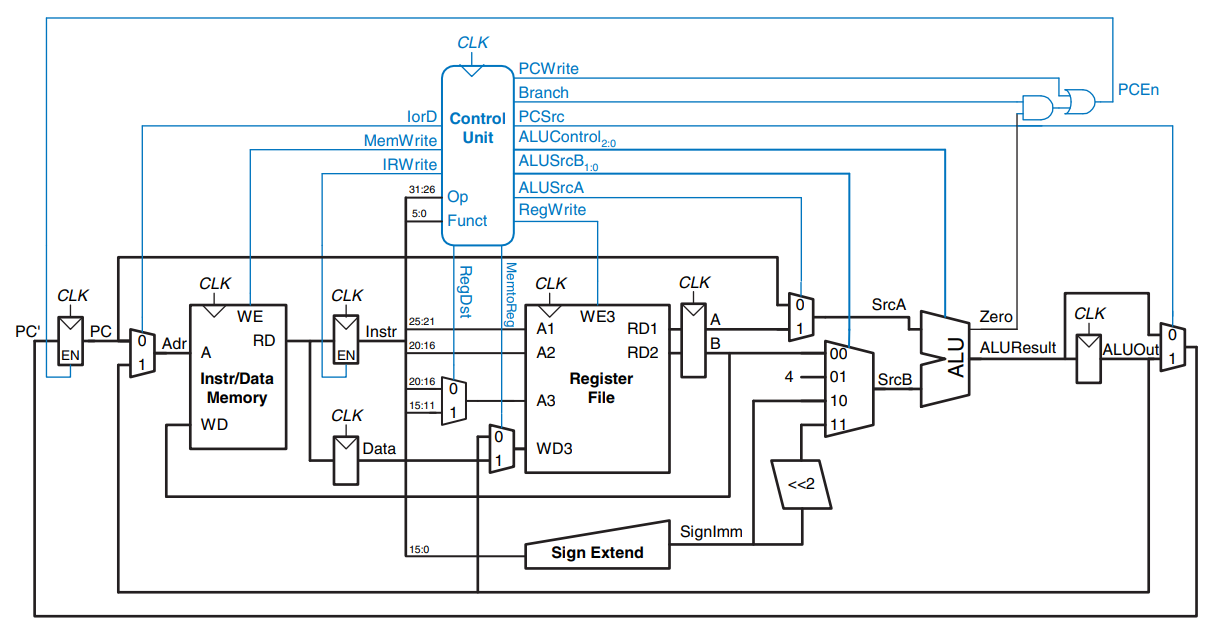


指令实际上就是I型指令，所以类似的，我们只需要继续添加组合逻辑就可以实现指令，具体来说我们需要添加一个右移两位原件以计算目的地址，需要能够受控制地基于ALU的运算结果改变PC的值。于是就得到了下面这样的结构。



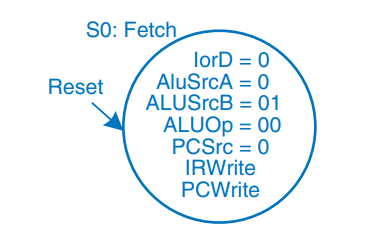
2）控制模块设计

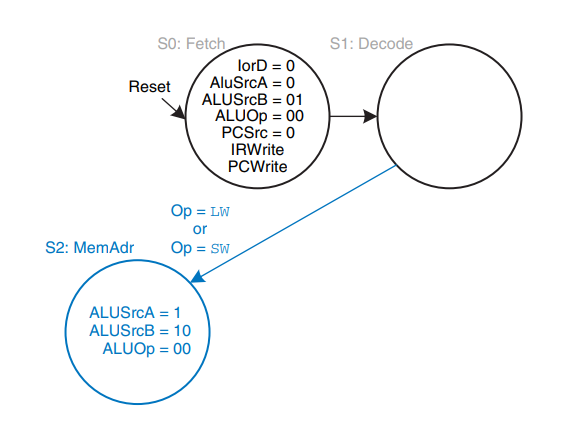
接下来我们需要一个根据指令类型产生控制信号的模块，具体来说就是主译码器模块和ALU译码器模块。主译码器模块将基于和字段产生对应的控制信号。将数据通路和控制模块连接，我们就得到了下面的电路。

下面我们将着手设计控制模块。类似于单周期处理器，控制单元仍旧由主译码器和ALU译码器组成。其中ALU译码器不变，而主译码单元却有显著差异。现在的主控制器应该在合适的周期中应用合适的控制信号的FSM，即有限状态机。控制信号序列依赖于当前正在执行的指令。

因而下面我们将着手设计主译码模块，而其核心是有限状态机的设计。类似于数据通路的设计，我们先着手产生指令的控制信号。

根据在设计数据通路时的分析，我们可以发现所以指令的第一步都是根据PC中保存的地址从存储器中取出指令。FSM在复位后总应该回到这个状态，以继续执行下一条指令。指令要从存储器中取出，因而我们要将IORD信号设置为来自指令。指令寄存器应该写允许，这样可以将指令写入寄存器中存储。同时，我们可以执行PC加4的操作，这样对大多数指令来说PC指向下一条指令的操作就提前实现了，这错开了ALU的使用。为此我们需要选择ALU的SRCA为PC值，而SRCB为4，ALU中应该执行加法。将上面这些控制信号与其值对应，我们就可以得到取指令状态下对应的输出，就像右边这样。

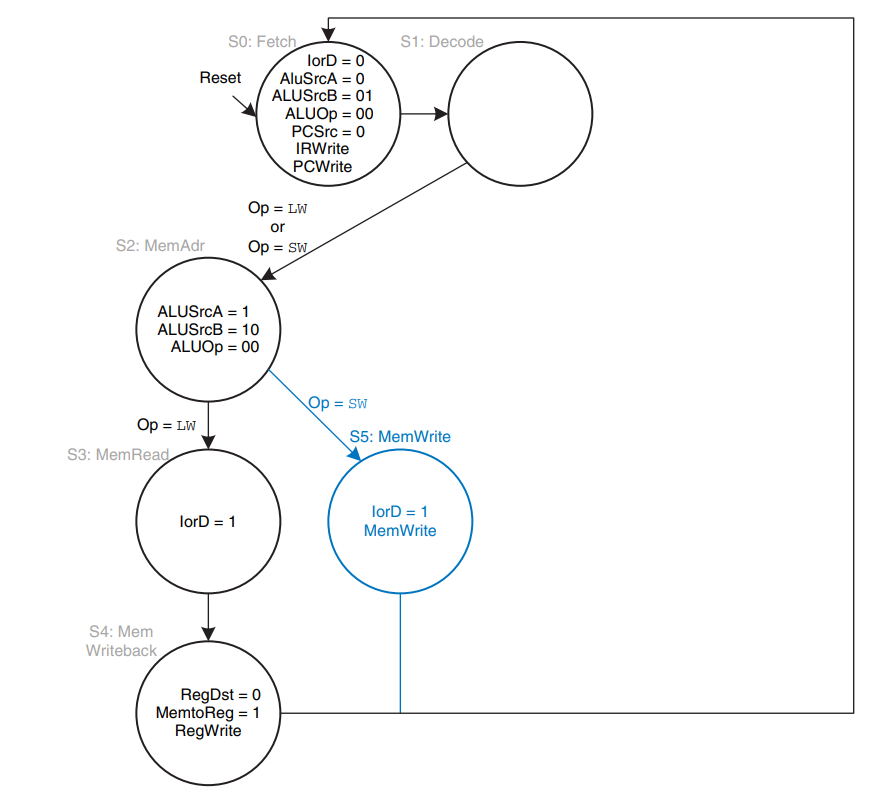
下一步我们将读寄存器文件，并对指令译码。这一阶段无需产生任何控制信号即可实现译码，但是我们仍旧需要等待一个时钟周期来完成译码。所以在FETCH状态后我们可以先进入一个无输出的DECODE状态。

在根据操作码译码后我们就得到了具体应该执行的操作了。对于指令，处理器应该在下一阶段通过基地址和立即数变址相加得到一个地址，这将作为源操作数或目的操作数。这需要ALU的SRCA选择来自寄存器数据，而SRCB来自符号扩展，ALU应该执行加法。至此我们得出了这样的有限状态机图。

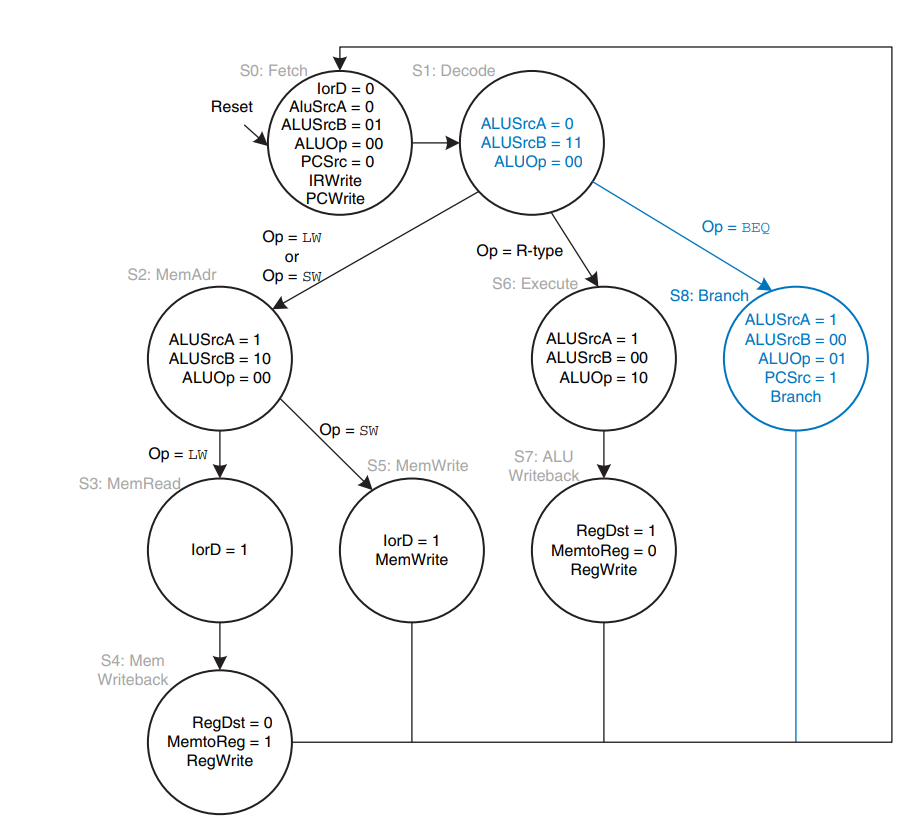
如果是指令，处理器应该紧接着从存储器中读取数据并将其写入寄存器文件。从存储器读取数据是缓慢的步骤，我们被迫为其单独设置一个状态，在这一状态下我们选择存储器的输入是数据，这样我们可以使用刚刚计算出来的地址，而读出的数据将被存储在寄存器中。最后，我们希望将数据写入寄存器文件中。这里的寄存器作为目的操作数，其数据来源是存储器。在完成这些状态后，我们就完成了指令，我们将回到取指令状态。

而如果是指令，在计算出存储器地址后，我们只需要将对应寄存器的数据写入存储器即可完成指令。

于是我们得到了这样的状态机图。

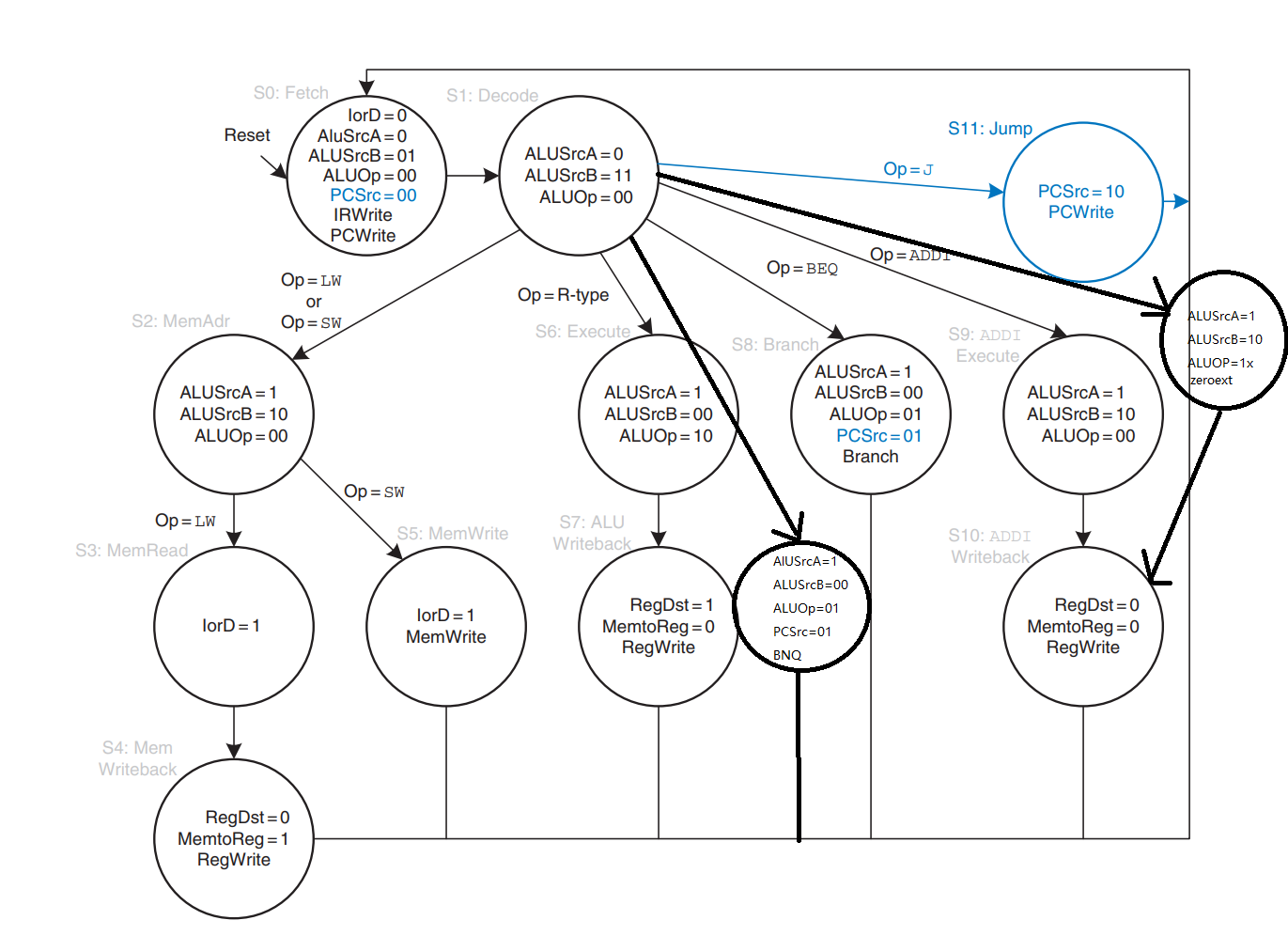


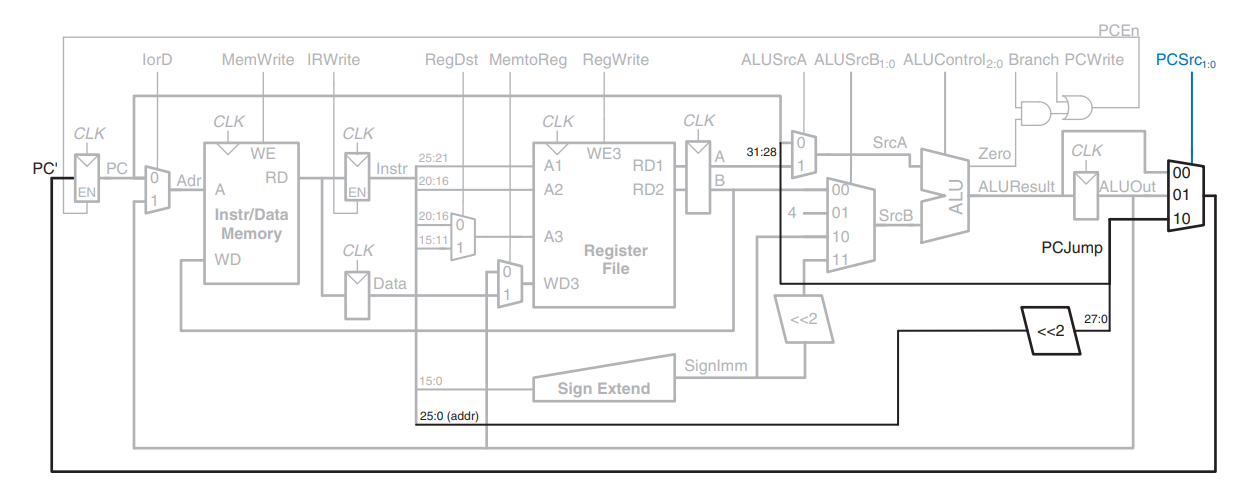
R型指令、和指令的设计过程是类似的，这里不再赘述。不过值得注意的是，我们发现在DECODE阶段ALU总是闲置的，于是我们考虑为其分配一个在接下来需要进行的运算。这一运算的执行应该没有前提条件，而它的计算结果也不应该对无关指令造成影响。参考我们安排操作在取指令阶段执行，我们把指令的地址运算安排在译码阶段似乎是个不错的选择。这样接下来如果确实是指令而且确实需要跳转，那么我们可以直接加载刚刚计算出的结果；而如果有任意一个不满足，那么我们不加载刚刚计算的结果就行，这不会影响接下来指令的执行。所以总的来说我们就得到了这样的状态机图。



3）指令扩展

我们已经实现了前几条指令了。接下来我们继续考虑跳转、立即数运算和bnq指令。相较于算数运算，逻辑运算中的立即数不应该实行符号扩展而应该实行０扩展。所以我们可以添加０扩展模块及多路器，并设计控制信号实现。对于无条件跳转指令，它需要直接计算出下一条指令的地址。考虑ｊ指令的26位立即数，我们不难发现只需要将其左移两位作为下一地址即可。而指令和指令实际上没有太大区别，只需要产生另外的控制信号即可。对应的，我们需要为它们产生合适的控制型号，而这需要我们向有限状态机中添加对应的状态与输出。数据通路的扩展设计类似于单周期扩展。而状态的设计与上面类似。最后我们就得到了下面这样的状态机图和电路图。

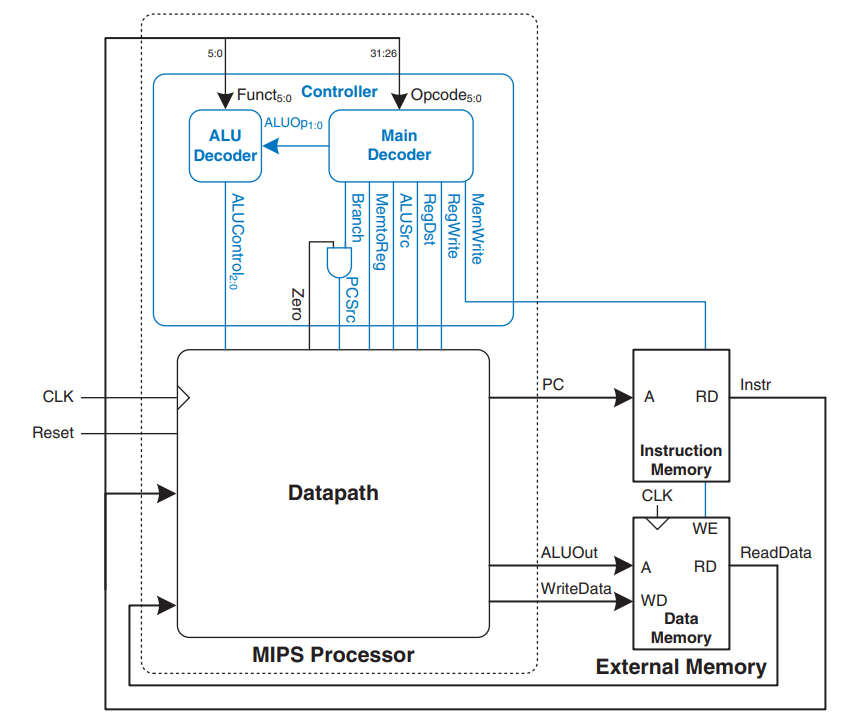




4）IO接口设计

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。

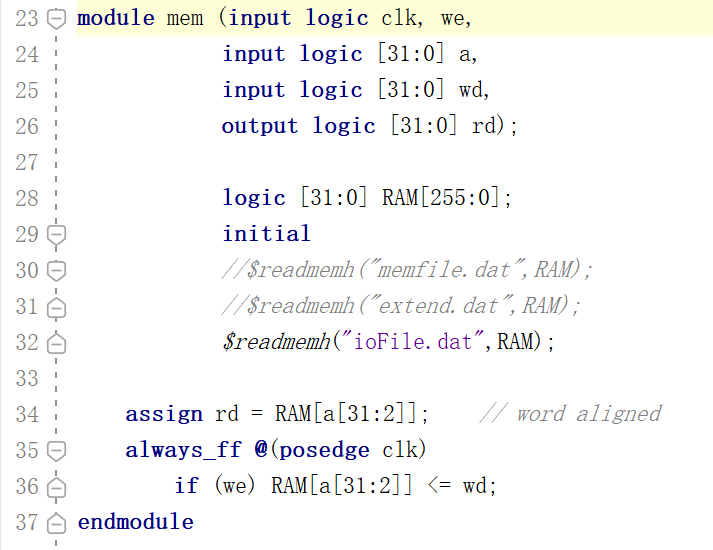


# **四、关键代码、仿真与开发板照片**

**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。与之不同的是，我们需要参照数据存储器与指令存储器，设计一个统一的通用存储器，就像下面这样。



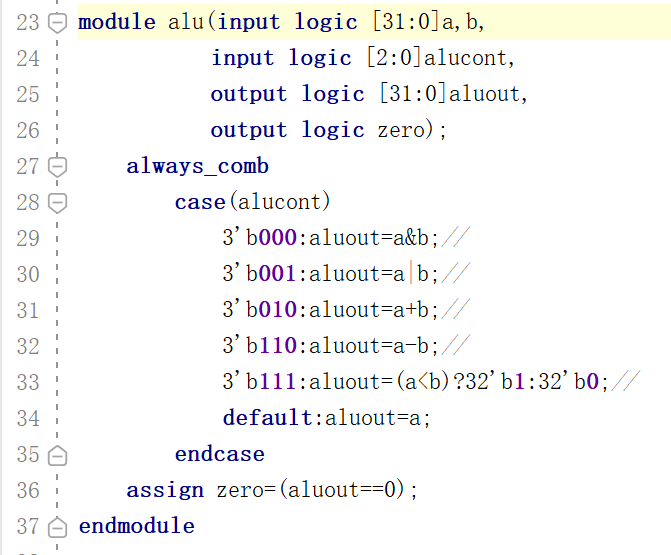
此外，我们还需要一个32位寄存器模块，它可以包含一个写使能端口。其设计与存储器类似，这里不再赘述。

**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。

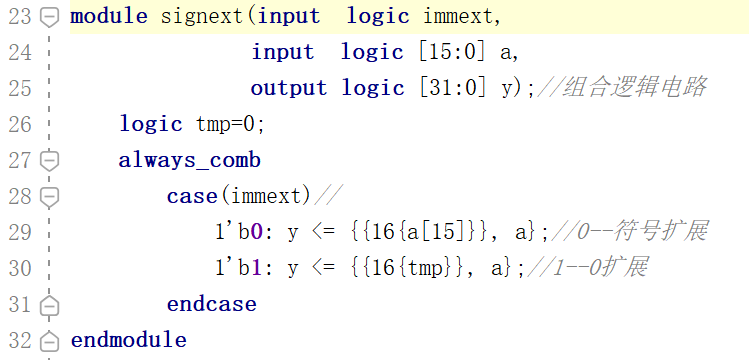
它可以直接复用单周期处理器中的ALU模块，就像下面这样。



２） 符号扩展和０扩展模块

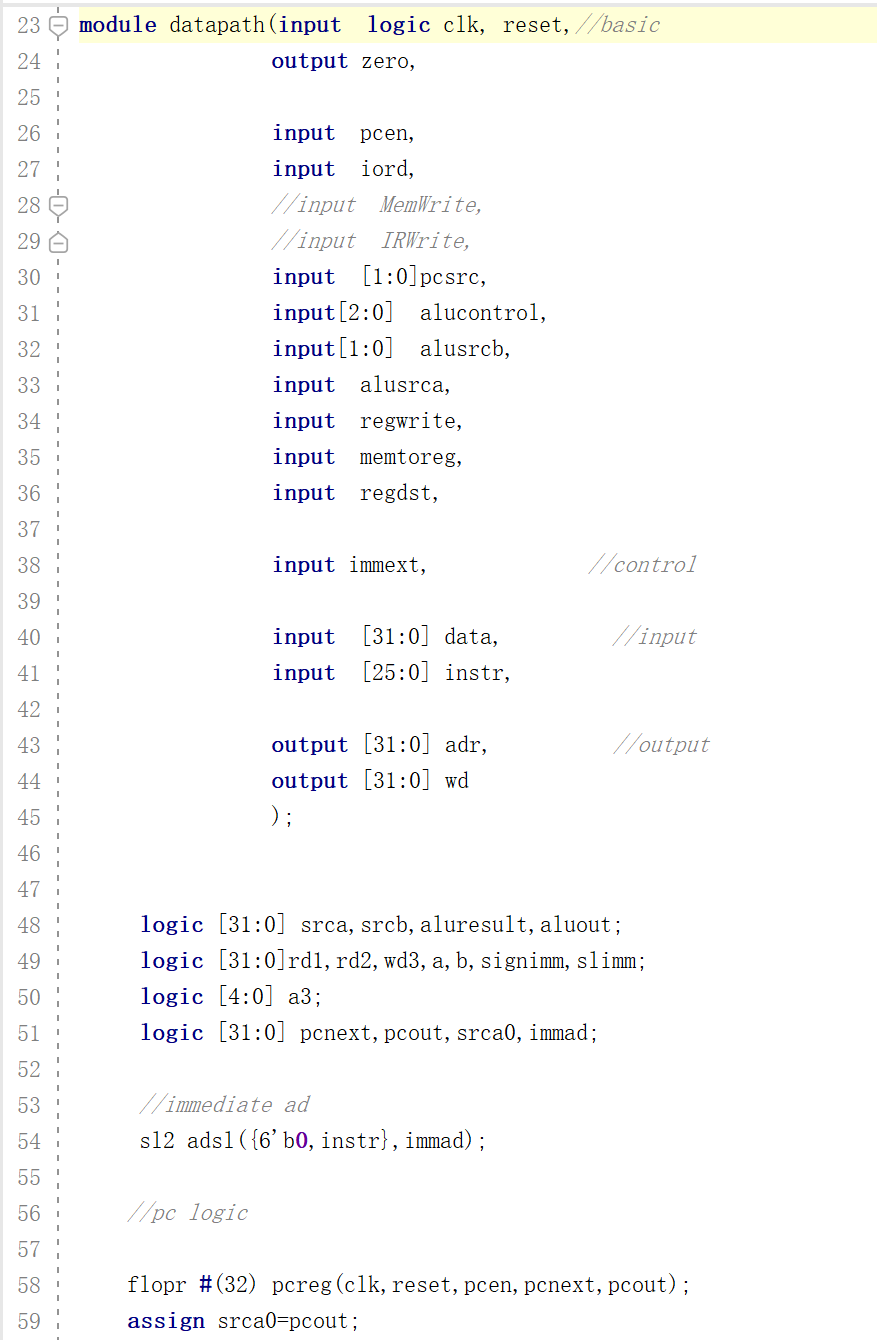
我们的３２位处理器经常需要接受一些小于32位的信号，为此我们就需要对它们进行扩展。而扩展又分为符号扩展和０扩展。因而我们可以这样设计扩展模块。

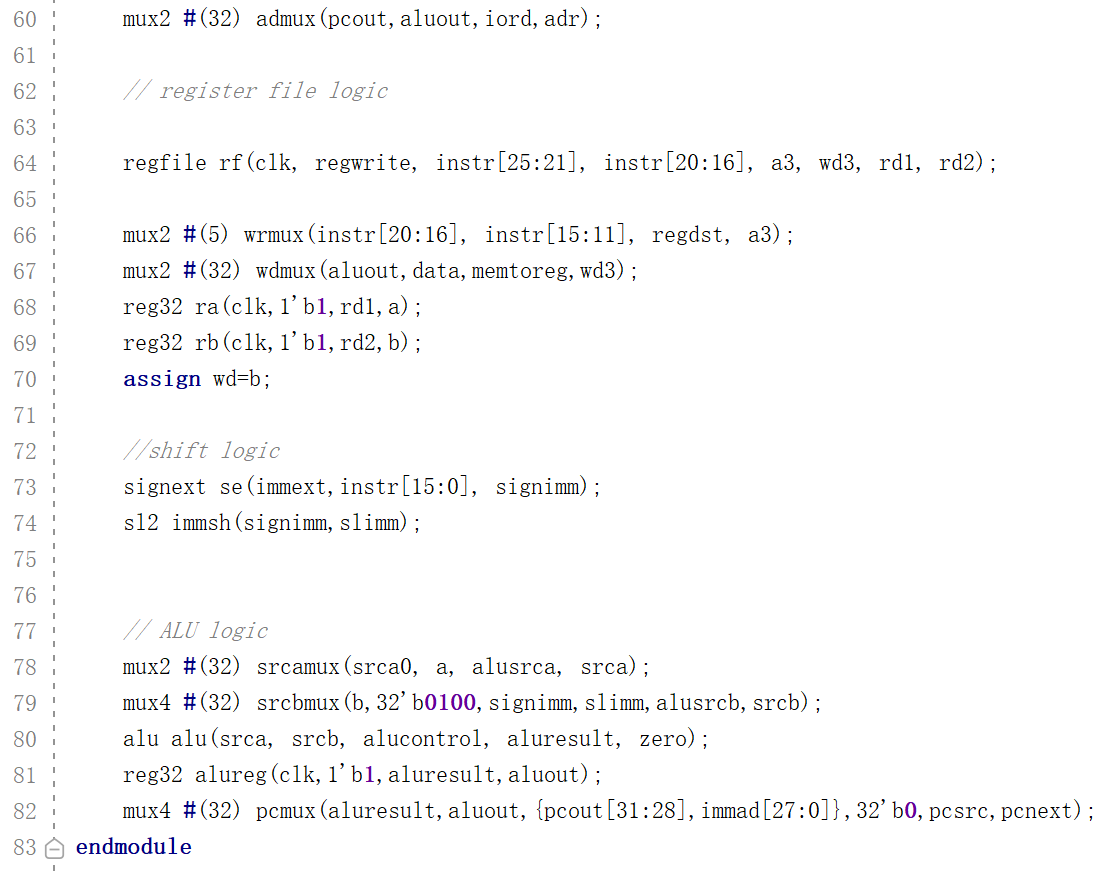
它包含一个控制信号决定进行０扩展还是符号扩展，一个32位输入和一个32位输出。这里的设计中整合了两种扩展模块。



**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，这样我们在日后更改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。



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，即寄存器的引入以及符合存储器的使用。

**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。

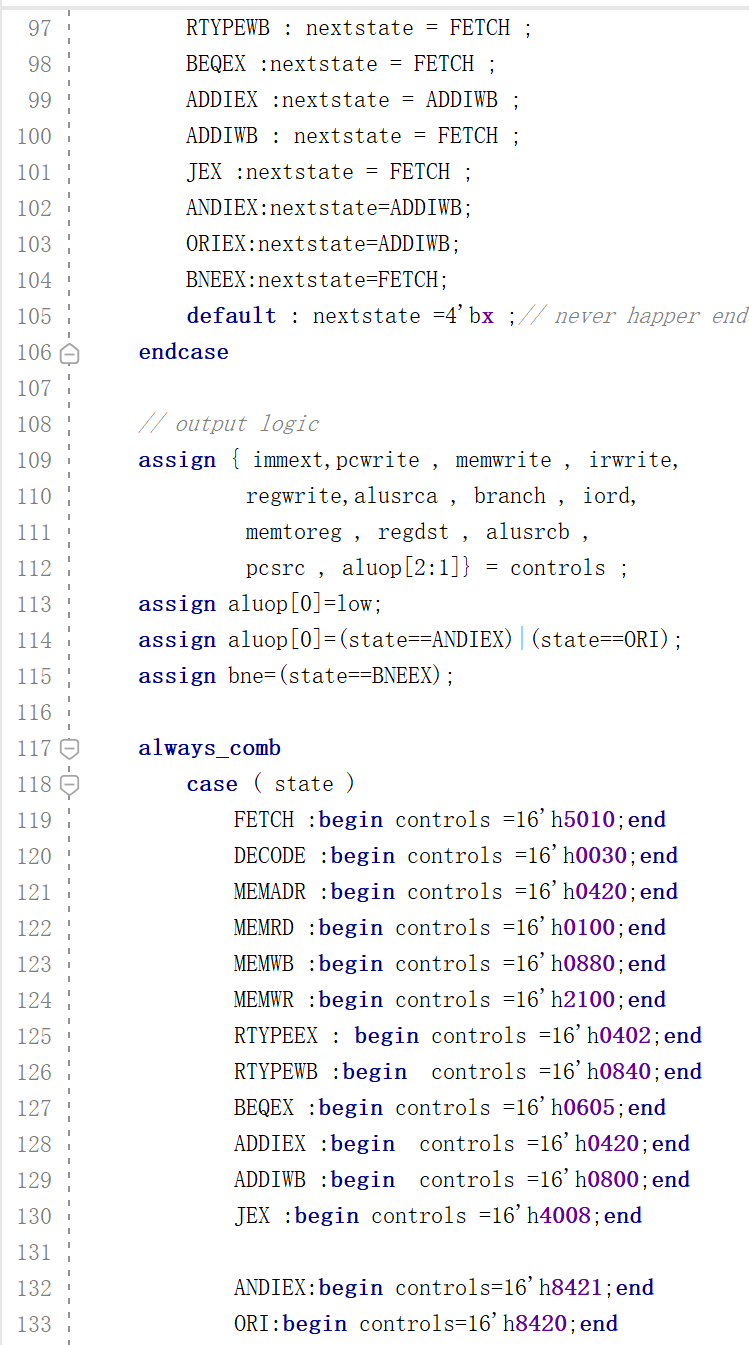
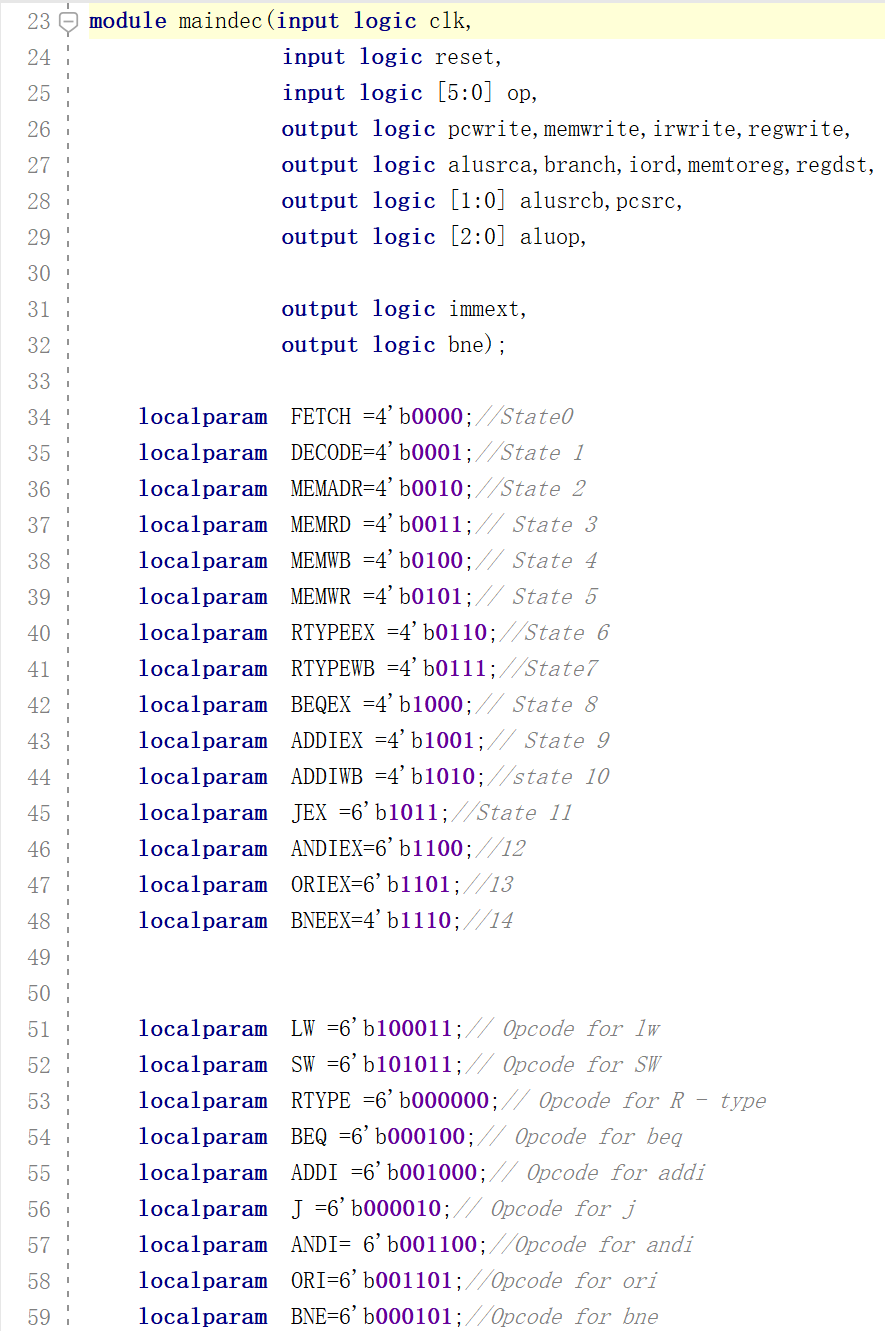
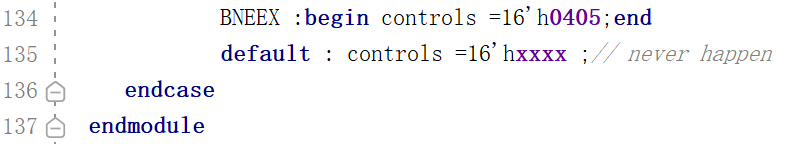
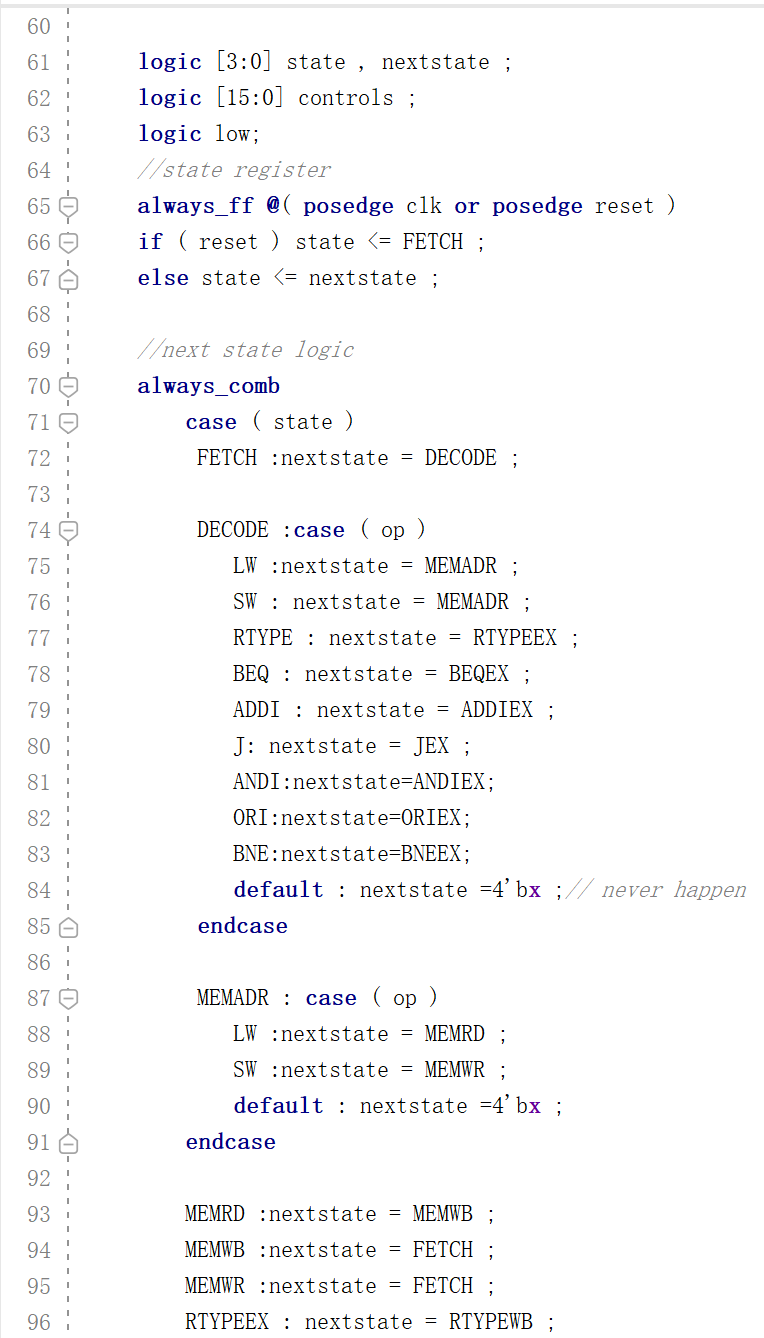
我们将单独设计主译码器与ALU译码器，再设计一个模块来例化它们。这样我们在扩展和更改时将大大获益。

控制模块包含了输入的控制字段和标志，在处理后输出控制信号。

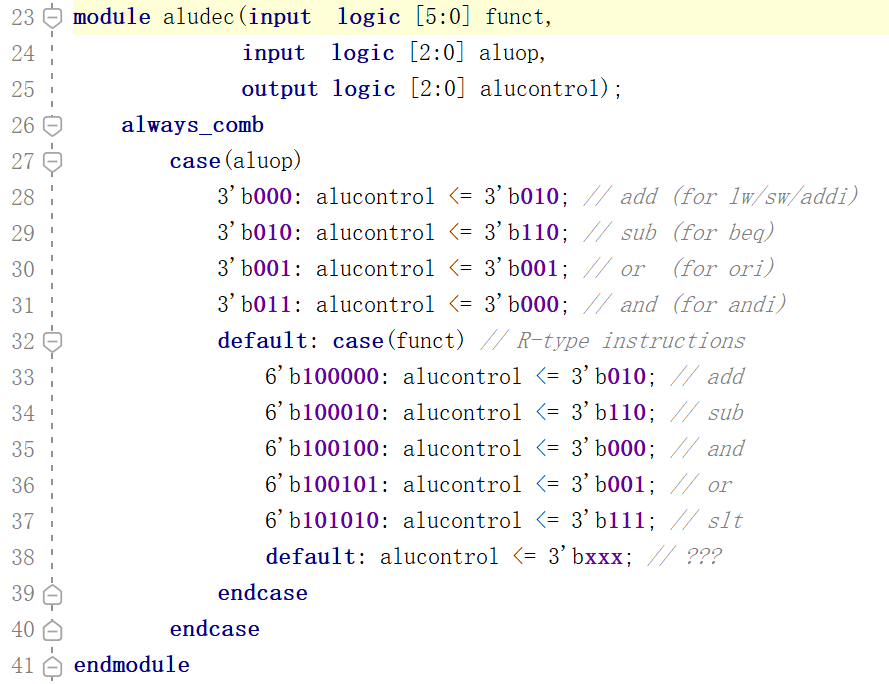


可以看出，多周期处理器的译码模块虽然需要产生更加复杂的控制信号，而且是需要时钟的时序逻辑，但总的来说仍旧与单周期处理器类似，进行的就是协调主译码器和ALU译码器产生控制信号。

主译码器模块基于字段产生控制信号，并输出。由于是时序电路，我们需要时钟信号。根据我们之前设计的有限状态机图，我们不难设计出下面这样的主译码器。

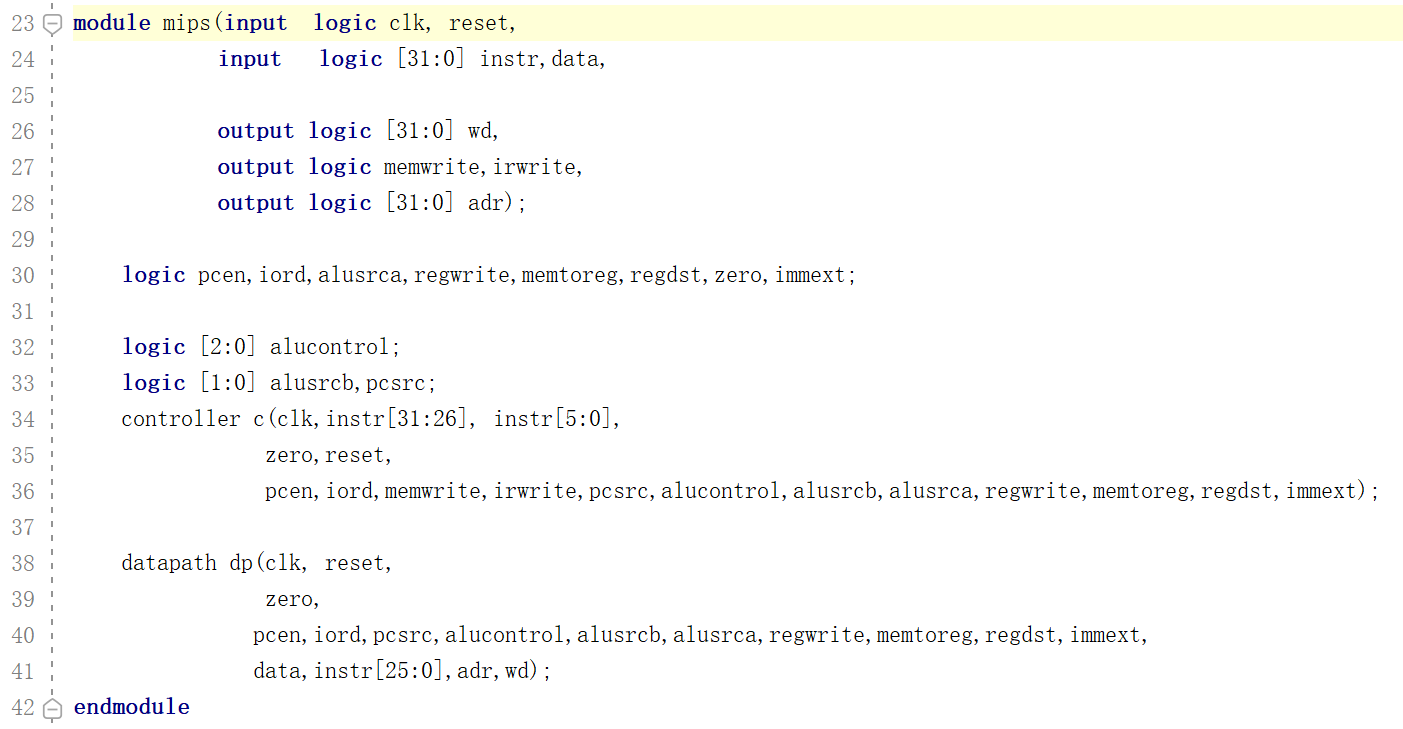


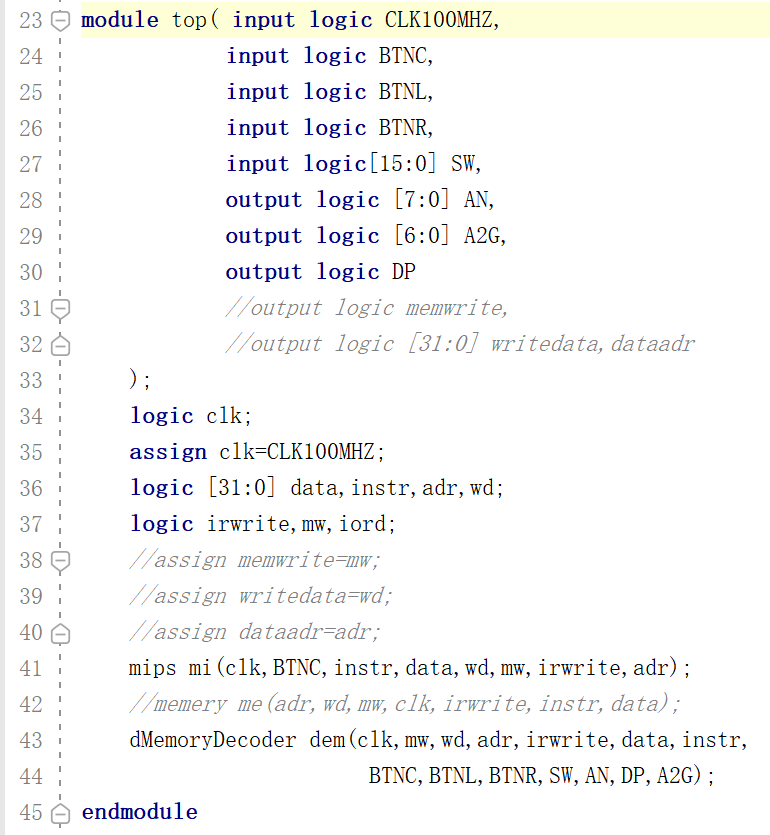
而ALU译码器根据主译码器信号和字段产生控制ALU的信号。这同样与在单周期处理器中一致。为了便于扩展，我们可以稍加改进，就像下面这样。



**５． 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。



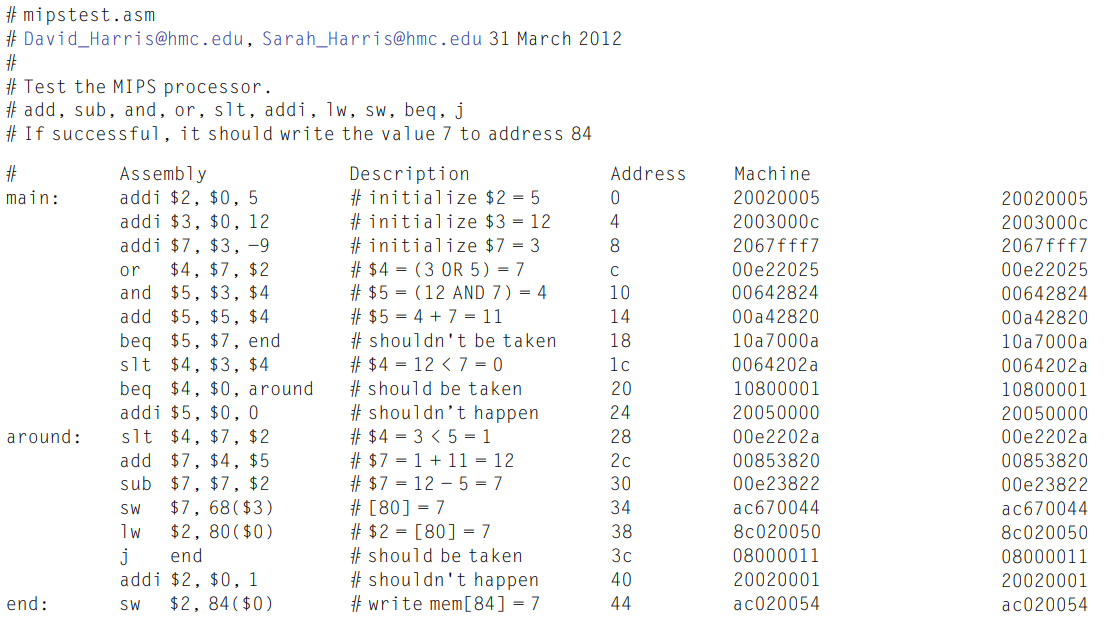


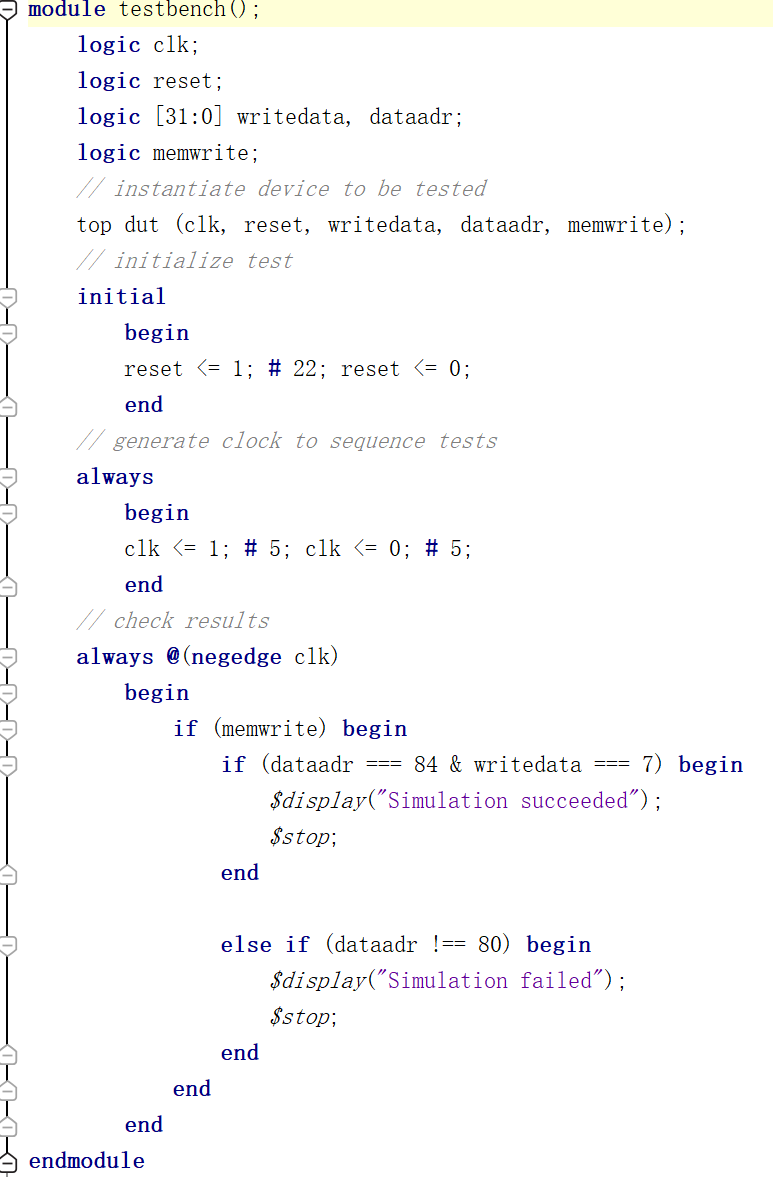
**６. 仿真**

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。

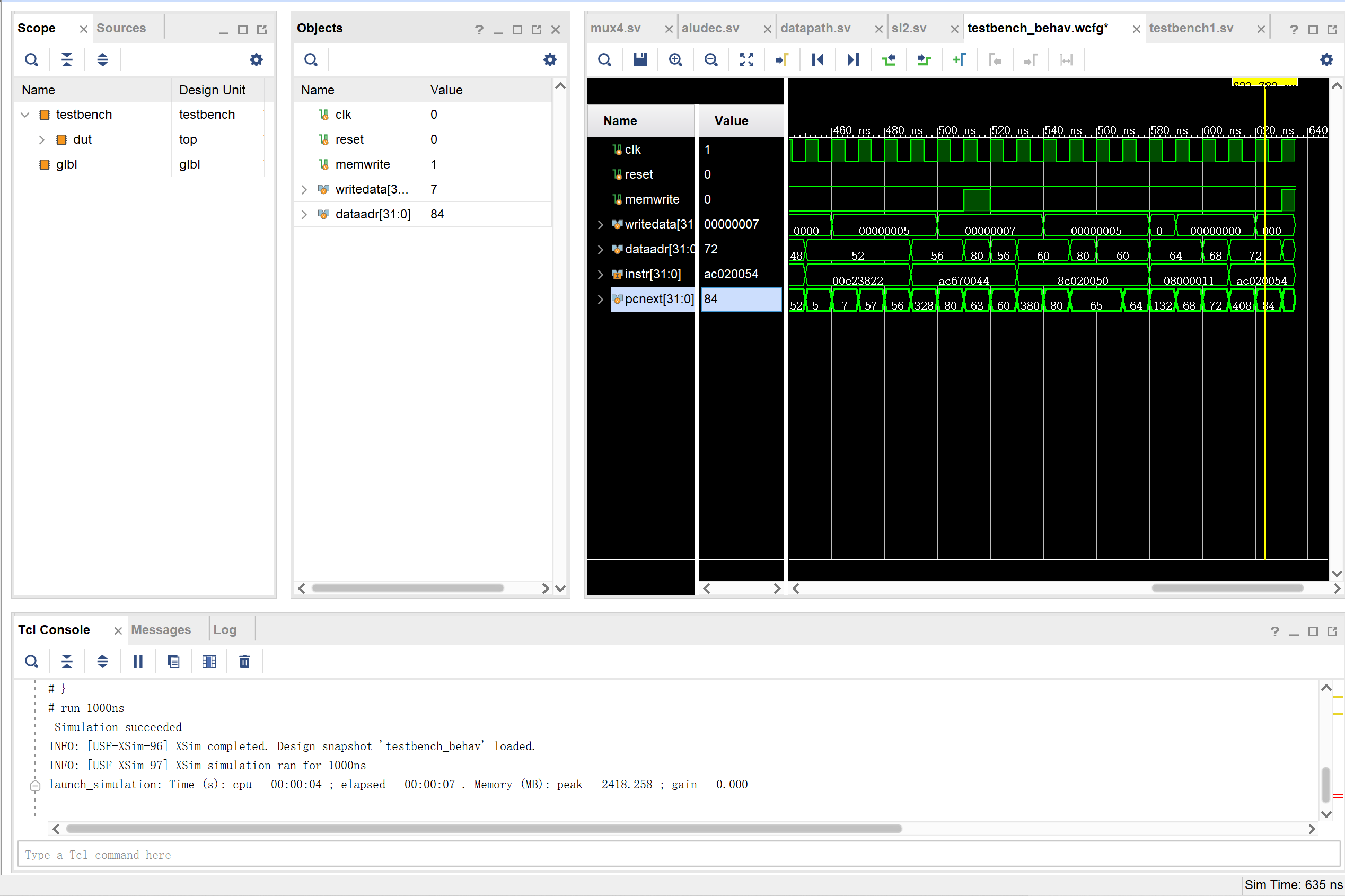
1. 未扩展的处理器

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。



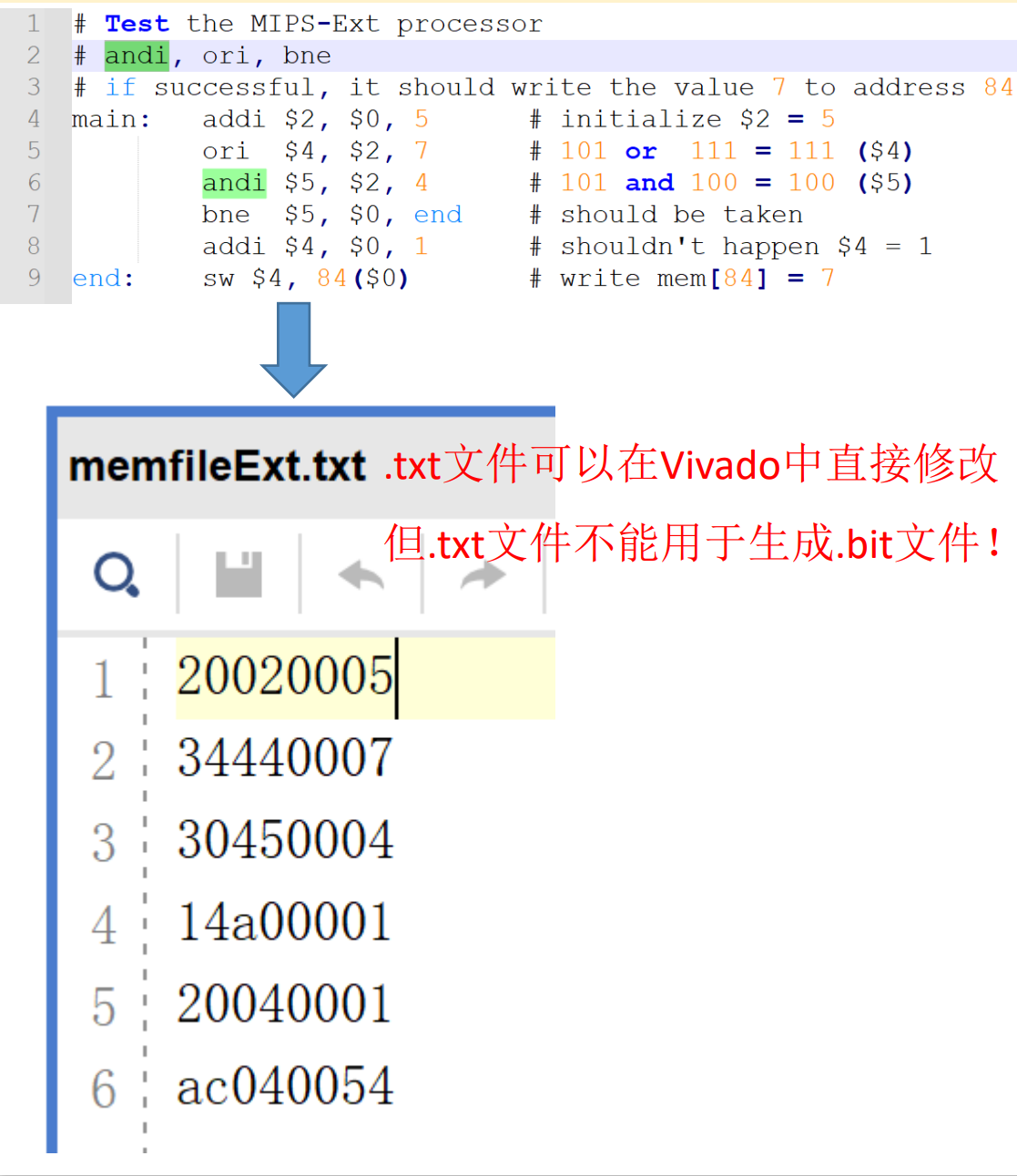


于是就可以得到这样的仿真结果。仿真成功。

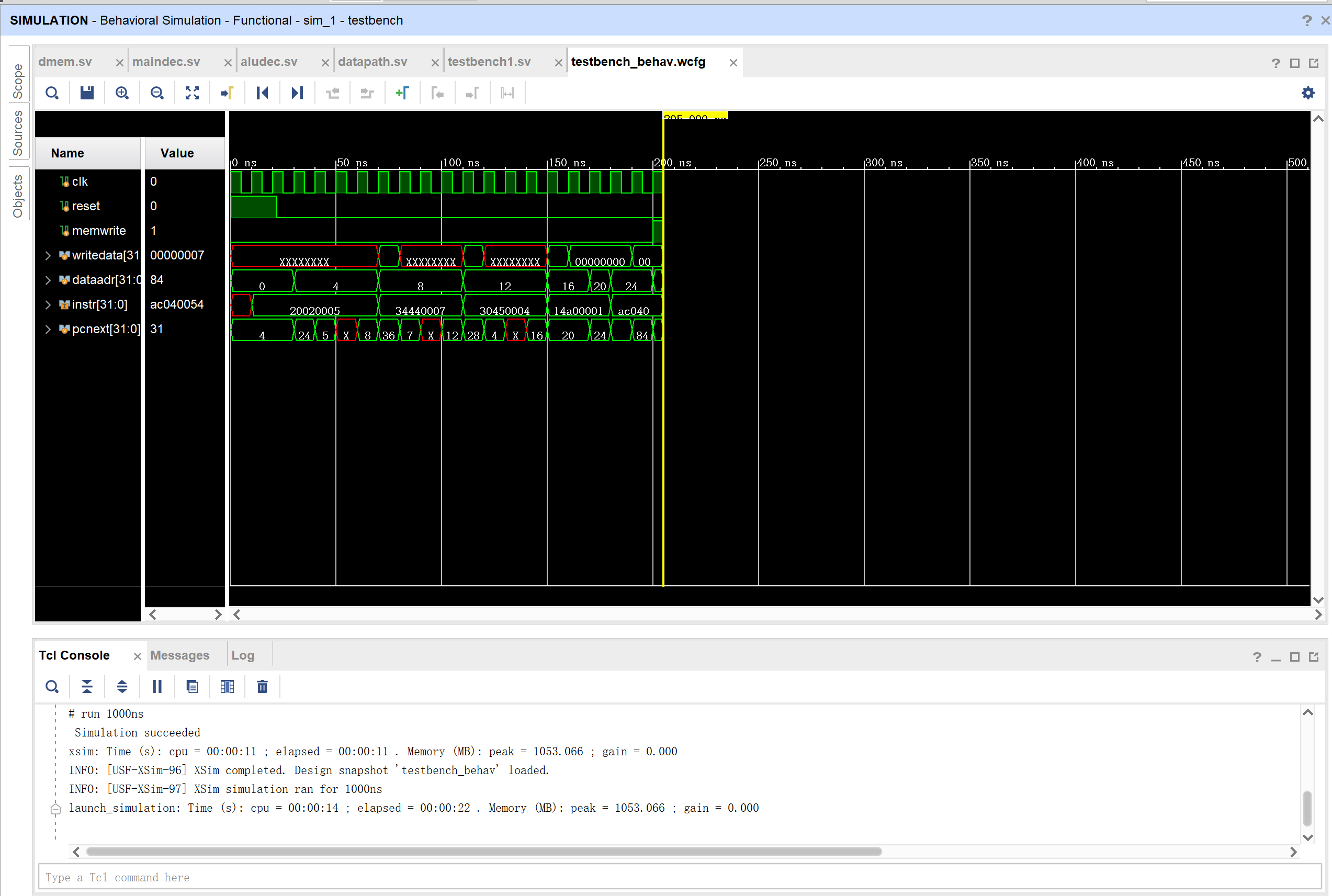


1. 扩展

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。

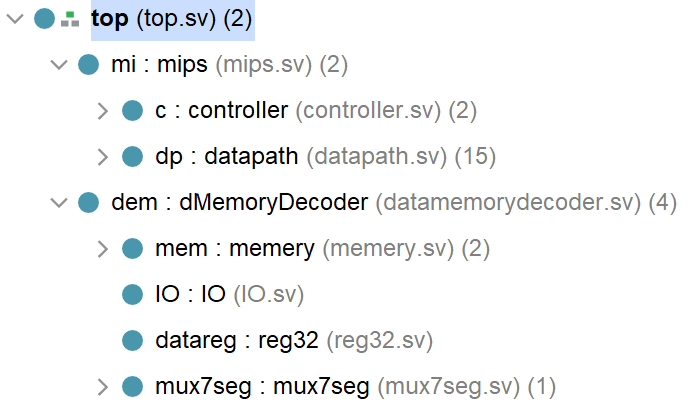


同样可以得到预期结果。

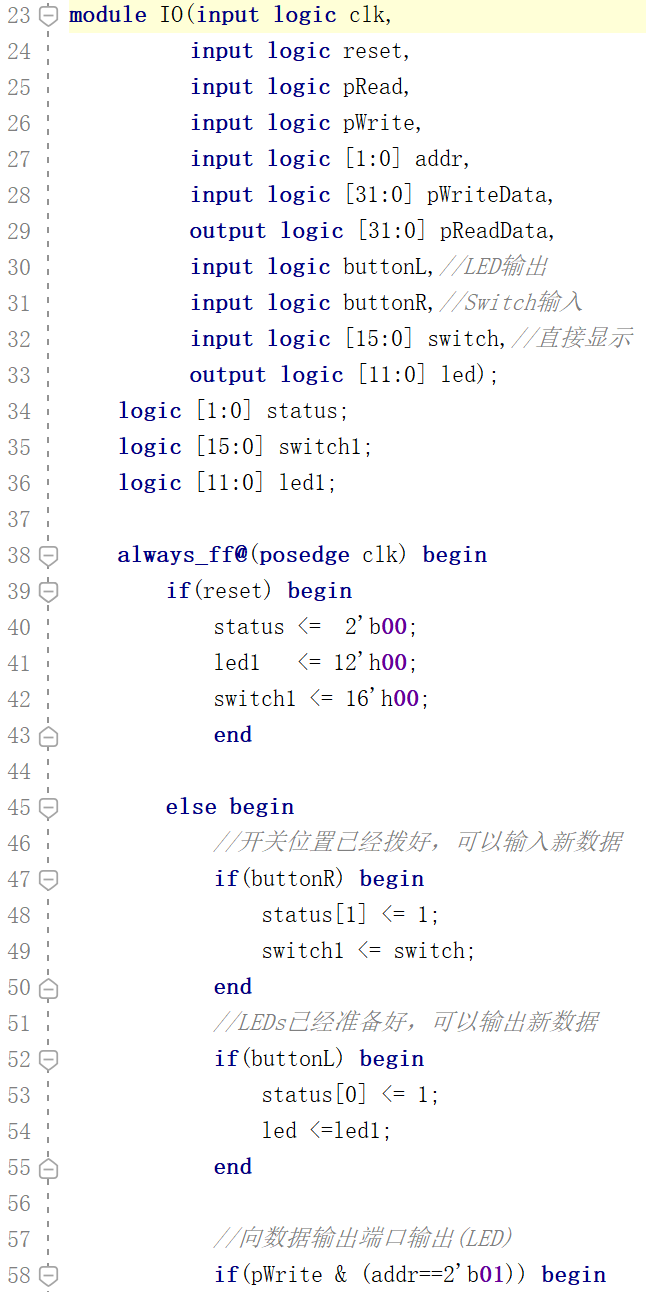


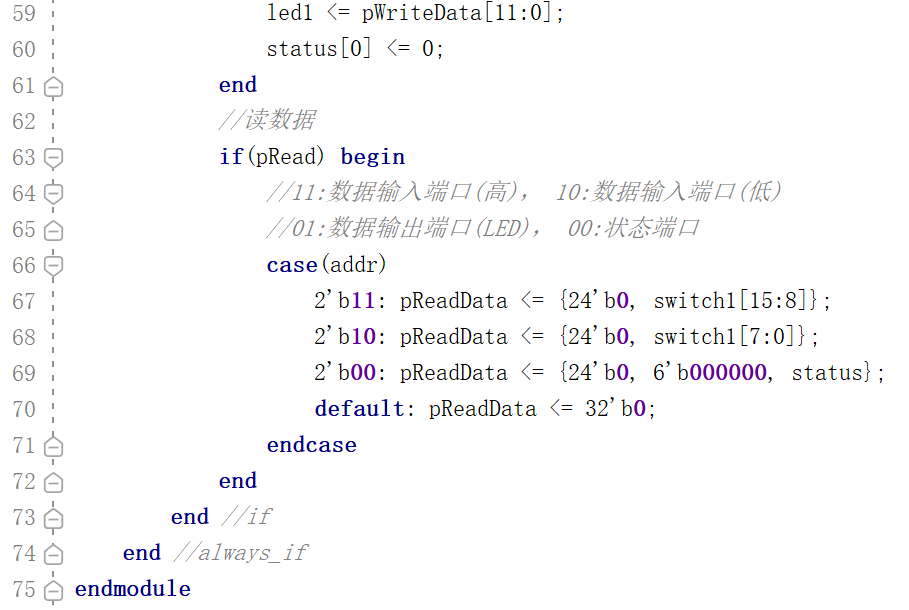
**７. IO接口设计**

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。



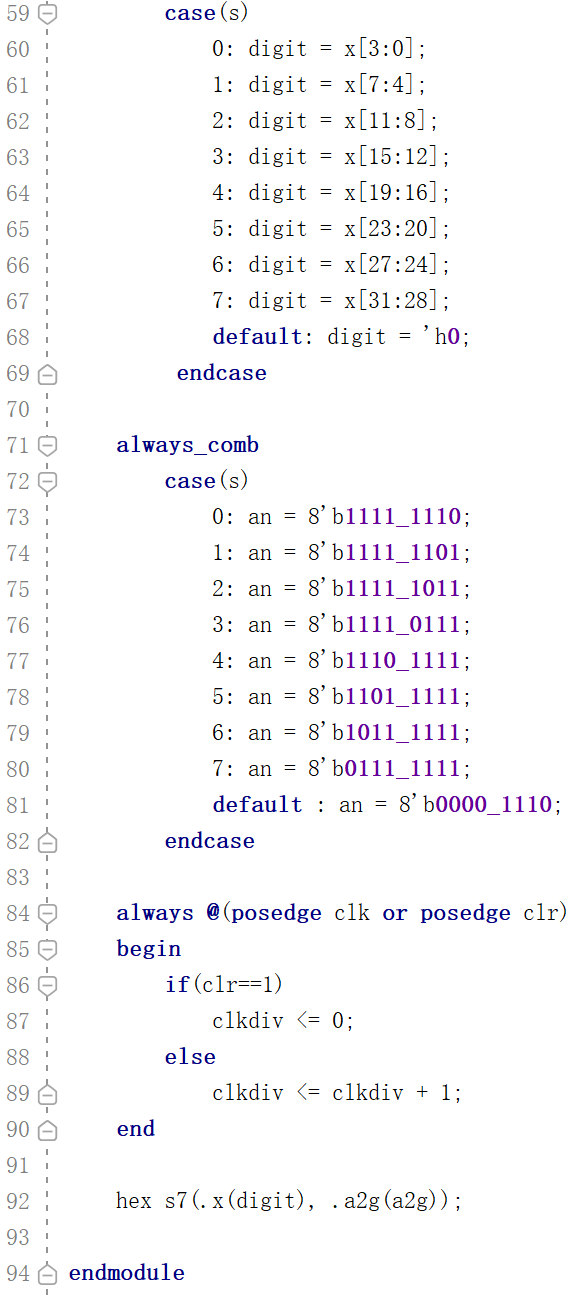
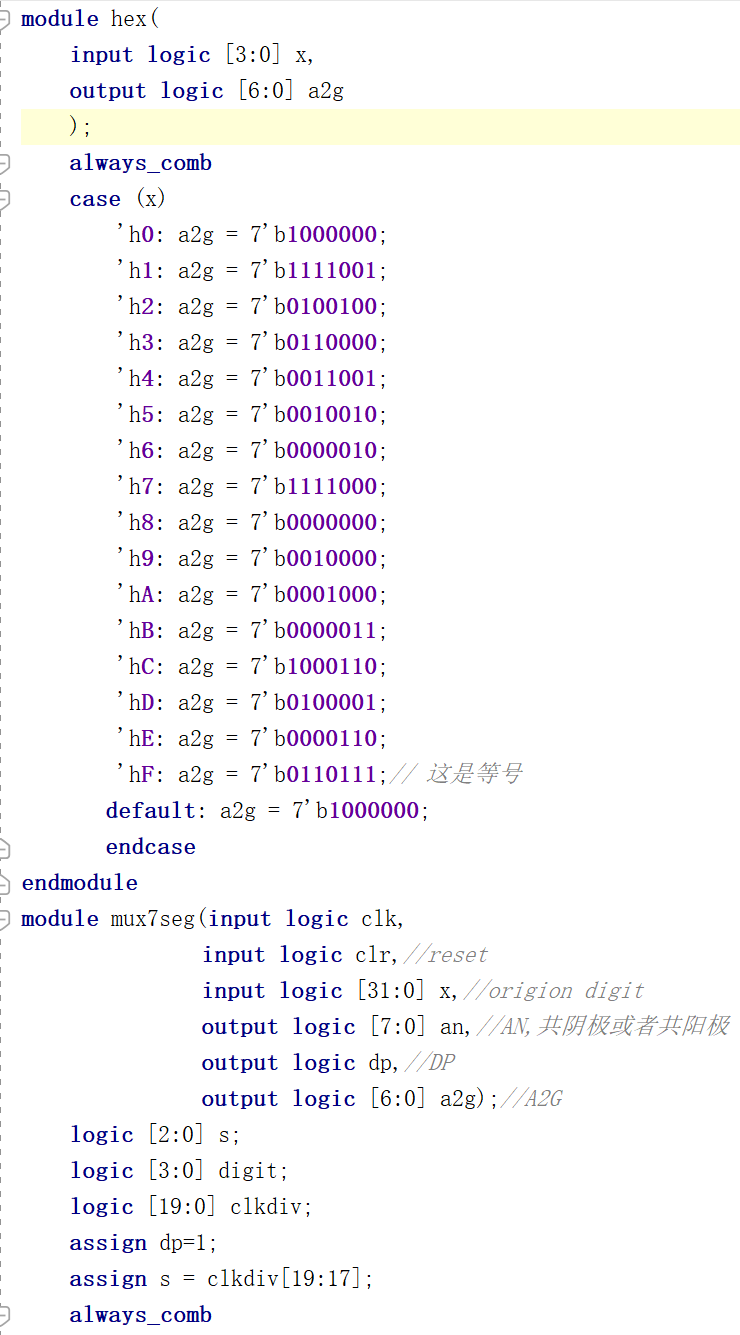
在原来数据存储器上加上附加的模块，使之能够支持IO操作。具体来说，我们需要一个额外的IO模块、一个七段数字管模块，以及一个控制模块。这同单周期处理器相同。





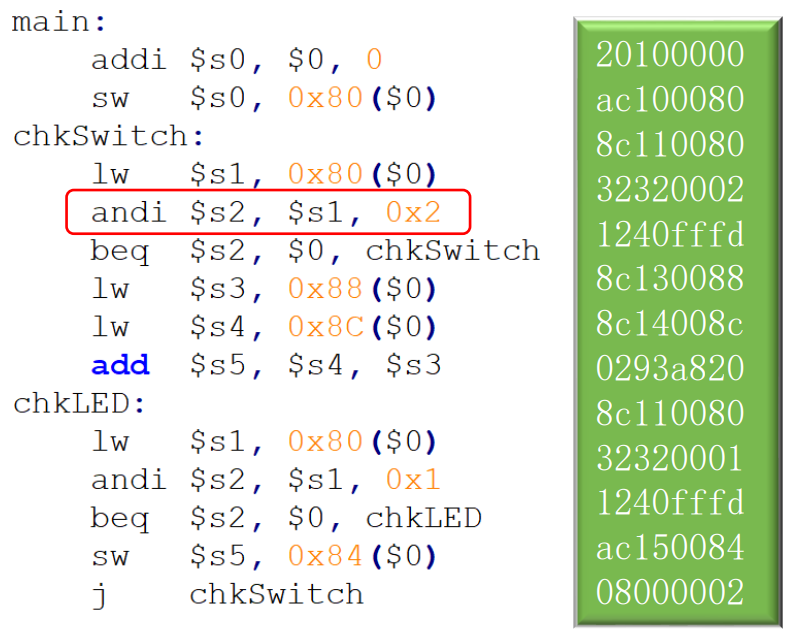
它可以从外界读取数据将之输出，也可以输出数据。

而七段数字管模块可以让输出结果显示在LED灯上。

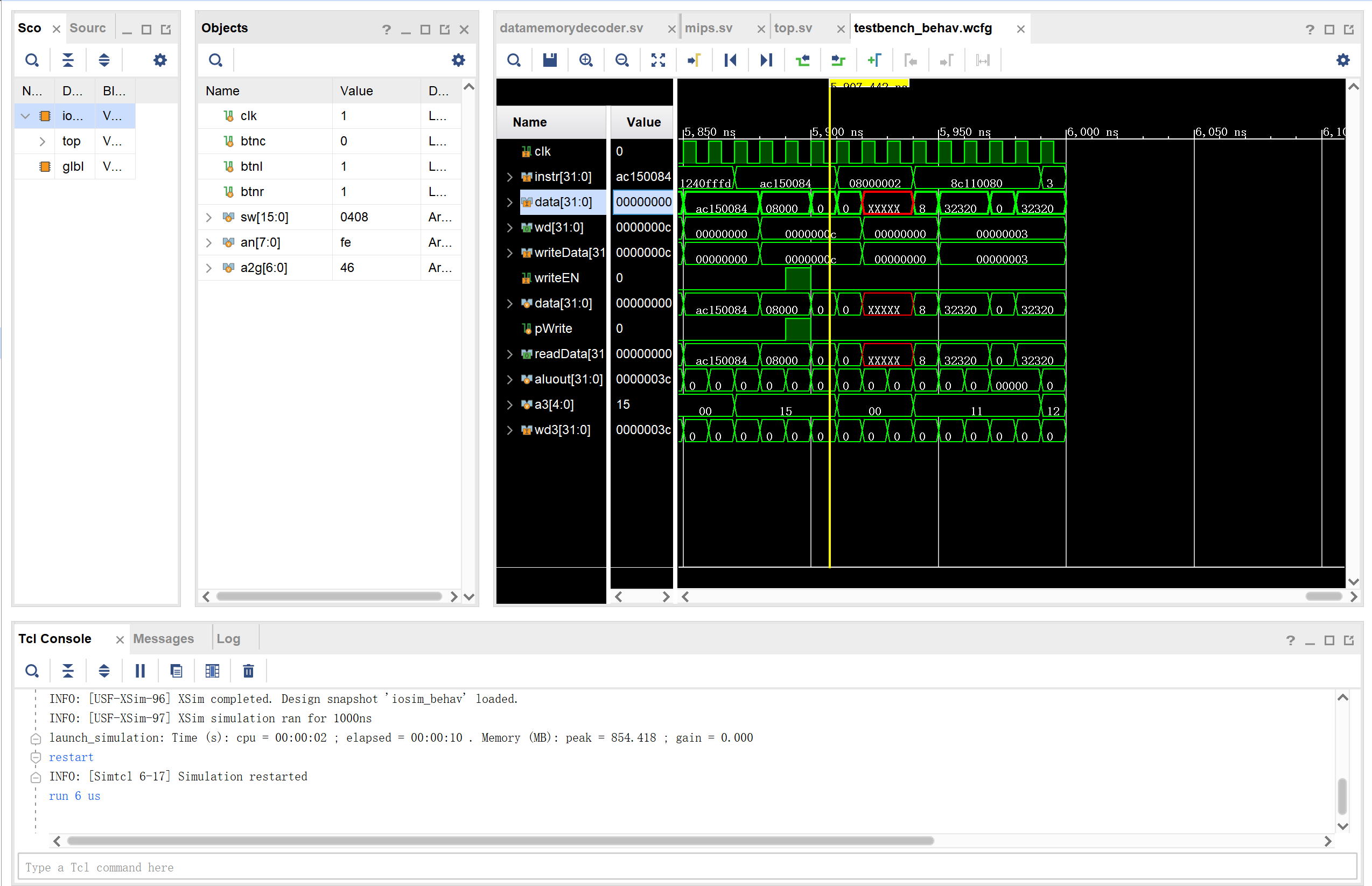


再加上控制模块，我们就实现了IO接口的设计。

在程序存储器中写入下面的程序。

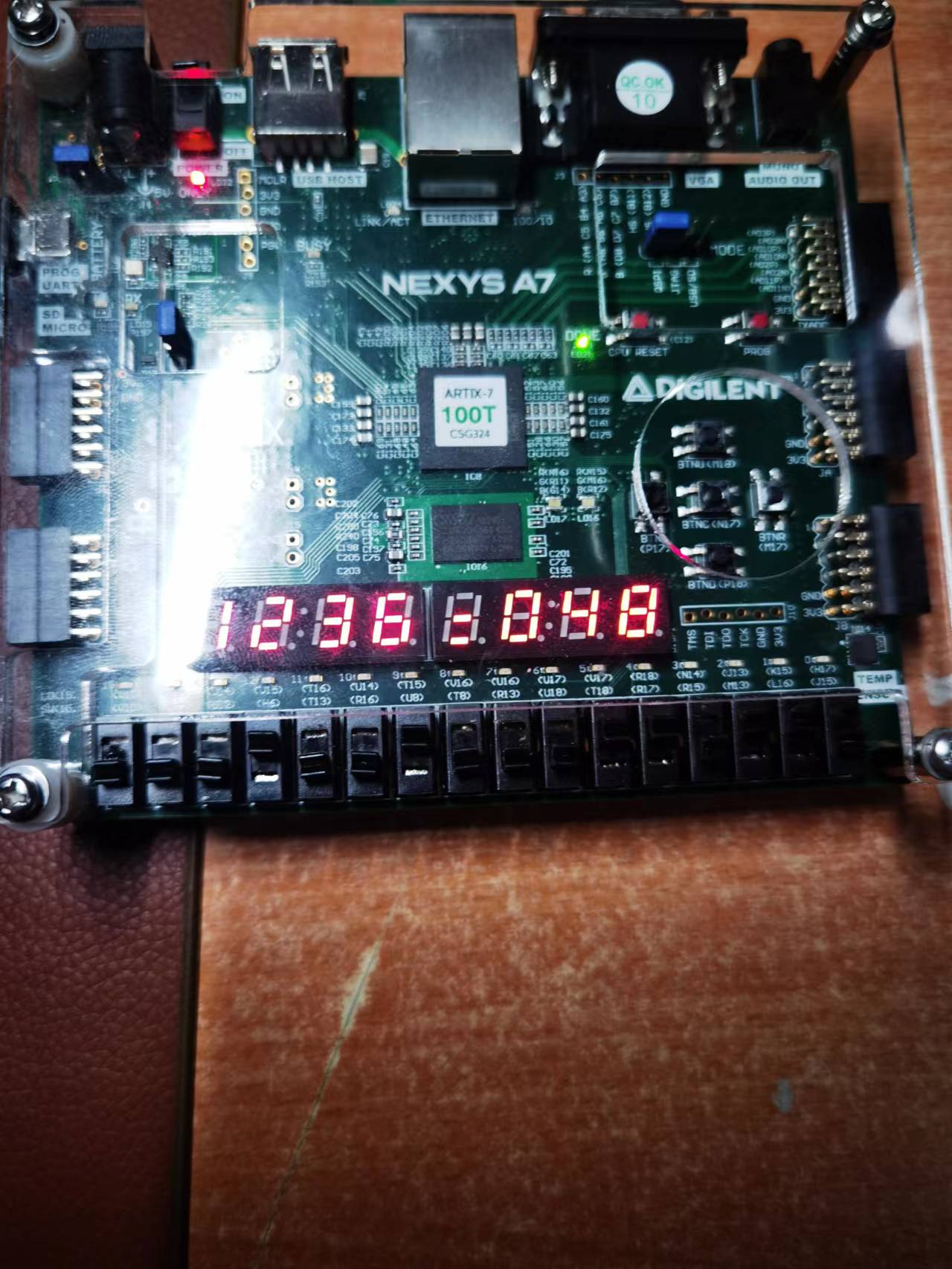


这里我们可以首先进行模拟。这里以计算4+8为例，进行16进制加法。



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**

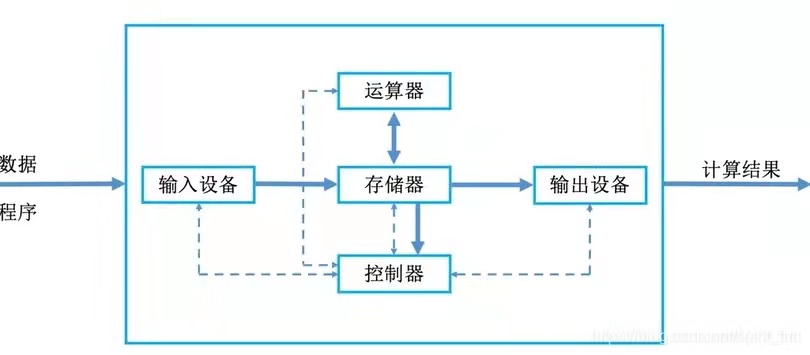
指令系统的设计设计是计算机系统设计的关键之一。从本次的设计中可以发现，一个精巧的指令集可以让设计变得更加简单而高效。

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. 恰当的状态设计**

如果我们认为多周期处理器就是针对单周期处理器的不足而生的，那么我们将明显地看出为了解决这些不足多周期处理器做出的改进。具体到状态设计上：

一方面，我们要减少快速加法器的数目，最终我们只保留了一个ALU中的加法器。但是对应相同的指令集体系结构，我们对同一条指令需要进行的加法数目并不会减少。所以在设计状态时，我们需要考虑在一个状态下只应该使用一次ALU单元，这样才能在减少ALU数目的条件下同样完成所有运算。另外，我们还为原本ALU空闲的状态合理地安排了任务，这样又可以精简状态，加快速度。

另一方面，我们使用了统一存储器存储数据和指令。这在单周期处理器中是不可能的。为了既能够从存储器中读写数据又能取得指令，我们当然需要单独设计状态这些状态。同时，由于访问内存相对来说要花费更长的时间来建立，所以我们在这些状态下我们应该尽可能缩短在这一状态下数据通过的路径，以加快速度。这样虽然在这些状态下我们几乎只能访存，但总体速度却更快。

最后，由于上面这两方面的改进，我们得以将指令分为多个部分执行。这样较快的指令就不会被较慢的指令拖累。

**4. 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多周期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，我们着手设计的多周期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。

**３. 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。