8-bit CPU（第二版）的Verilog设计

# 一、概况

CPU（Central Processing Unit）的中文全称是“中央处理器”，是计算机中的核心部件。任何计算机程序都是使用编程语言编写的，可以被分为高级语言和低级语言。高级语言采用更易于识别和记忆的字符作为关键字，且更加符合人类的思维逻辑，但是不可以被处理器所理解。[[1]](#footnote-1)因此，需要通过编译变成低级语言。汇编语言就是一种低级语言，它与指令集中的指令一一对应。

本8-bit CPU采用流水线设计，对指令同步进行“取指”“译码”“执行”“写回”四步操作。在指令集的设计上，功能简单，易于电路实现，依赖多步操作以完成复杂的功能。且具有一定的分支预测能力。

寄存器是CPU内部的小存储器，用来暂时存放一些数据。CPU的位数决定了CPU一次能处理的数据大小，也就是CPU的 运算能力。CPU的位数越高，其运算能力越强。

本8bit-CPU就是指能够一次处理8个比特的数据的处理器。内部有两个数据寄存器，分别为寄存器0和寄存器1。其中，寄存器1又称为地址寄存器，它的数据不但可以直接被指令赋值，并且可以帮助内存寻址。

由于该指令集的指令允许算术逻辑单元直接从内存中取值，因此并不属于精简指令集（RISC）。

# 二、指令集结构和设计

指令集是一组处理器能够理解和执行的机器指令的集合，通常包含算数操作（加减乘除）、逻辑操作（与或非）、数据传输操作（加载、存储）等，指令集中的每个指令都可以对应到处理器的特定电路，处理器通过解码指令来执行相应的操作。指令集是处理器的重要组成部分，不同的处理器有不同的指令集，指令集的设计直接影响了处理器的性能和功能。

常见的指令集可以被分类为精简指令集（RISC）和复杂指令集（CISC）。RISC指令集的特点是指令长度固定，指令格式简单，指令的执行时间基本相同。CISC指令集的特点是指令长度不固定，指令格式复杂，指令的执行时间不同。RISC指令集的优点是指令执行速度快，硬件实现简单，易于设计和实现。CISC指令集的优点是指令集丰富，编程灵活，可以用较少的指令完成较多的工作。常见的指令集包括x86、ARM、RISCV等，其中x86是英特尔推出的32位复杂指令集，超微半导体（AMD）在此基础上延伸出了64位指令集AMD64，目前广泛应用于家用计算机处理器[[2]](#footnote-2)。而ARM是一种精简指令集（RISC）处理器的指令集，在手机芯片上比较常见。RISCV则是一个新兴的开源免费的RISC指令集。

RISC还有一个特点是不可以直接对内存中的数据进行计算。必须先将数据存储到寄存器中，然后对寄存器之间进行计算操作。这样的好处是可以提高运算速度，但是缺点是需要更多的寄存器。因此，RISC的CPU通常都会有更多的寄存器。

## 1、原始版本的指令集设计

在本8bit-CPU的设计中，所采用的指令集的指令长度和数据长度一样，均为8bit，所以能够执行的指令种类非常受限。因此指令集只设计了2个数据寄存器供CPU调用。在这种条件下，寄存器的数量远远不够，必须允许CPU直接从内存中取值，所以这不是精简指令集。

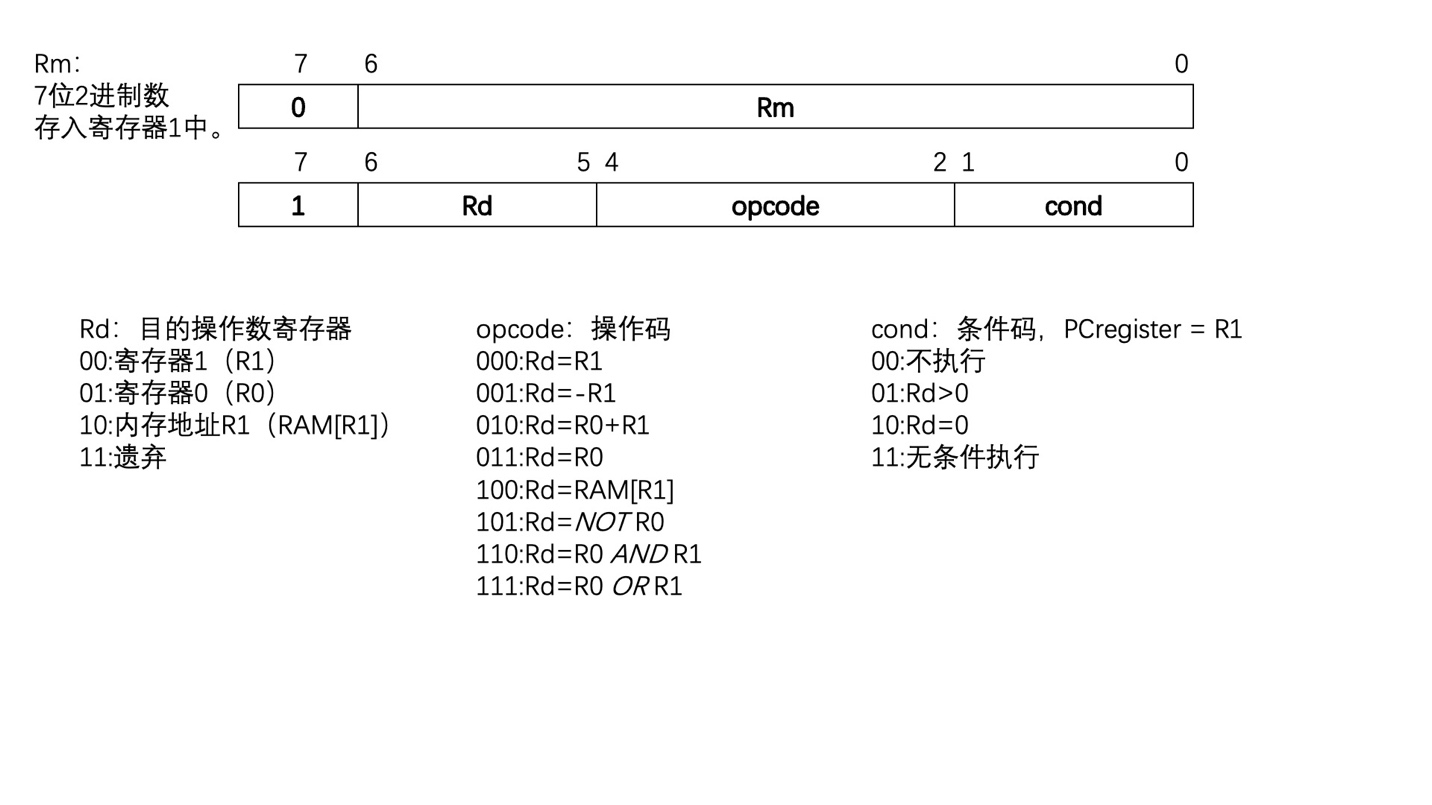
指令集由8位构成，分为“数据指令”（D指令）和“控制指令”（C指令）。

数据指令：当第7位（最左侧的一位）为0时，将整个指令作为数据存入寄存器1中。

控制指令：当第7位为1时，将指令进行正常的译码以控制整个CPU。

指令集的具体规则如下图2.2.1所示。

需要提醒的是，操作码001的取负数计算是将原数据R1当作有符号数（Signed）处理的，将数据变成它的补码。条件码01判断结果是否大于0同样是将结果当作有符号数处理的。其他的操作中有符号数和无符号数（Unsigned）的计算方式其实是一样的。

  
图2.1.1

当第7位为0时，立即数RM将会存入寄存器1中。立即数只有7位，因此最高位将会用0填充。

当第7位为1时，指令被用来控制CPU的运行。ALU算术逻辑单元是CPU运行时的主要单元，它受到opcode操作码的控制，对输入的数据进行指定运算。Rd目的操作数寄存器，指明了的ALU的输出结果将会被存到什么地方。

Cond条件码是对ALU的输出进行条件判断。通常在CPU执行指令的时候，会按照存储器中的顺序顺次执行指令。这是通过一个会自增的PC计数器（Program counter，程序计数器）实现的，在每个时钟周期，CPU将计数器中的数值作为地址从存储器的对应位置读取指令。所以当需要进行特别的跳转时，根据条件码的约束，在符合条件的情况下，PC计数器不再自增，而是加载寄存器1中存储的数值。

下面给出了一个事例，编写用于计算斐波那契数列的第10个数的程序。

譬如首先先写一个直观的伪代码。

\*a = 0

a[0] = 1

a[1] = 1

i = 0

T = 8

while i<T:

a[i+2] = a[i+1] + a[i]

i = i + 1

上面代码中的a是一个数组，并且数组的初始地址被规定为了0。数组的第一个数和第二个数都等于1。接下来的代码其实是for(i=0;i<T;i+=1)循环的另一种写法。然后我们让每个数都等于前面两个数的和。

在把它翻译为汇编语言时，需要把复杂的代码改写为可以由指令集处理的简单代码。举个例子，a[0]=1， a是一个数组，但是根据C语言的知识，a的本身其实是一个指针，而a[0]其实指向的是\*(a+0)。即\*(a+0)=1。其实就是要把内存RAM[a+0]等于1。那么我们首先要让寄存器1的值R1=a+0，因为指令集只支持读或取RAM[R1]的数据。但是，由于指令集里面唯一的立即数操作只能向寄存器1中写入数据，这将会与上面的RAM[R1]冲突。解决的方法是充分利用寄存器0，凭借指令RAM[R1]=R0来写入数据。过程略有繁琐，如下所示。

#实现a[0] = 1的方法

#计算出a+0并存储。

R1 = a # a是一个地址常数，并且已经定义过了。

R0 = R1 # let R0 = a

R1 = 0

R0 = R0 + R1 # let R0 = a + 0

R1 = &addr # &addr也是一个已经定义过的地址常数。&表示取地址。

RAM[R1] = R0 # let RAM[&addr] = a + 0

#让R0等于待存入的数

R1 = 1

R0 = R1 # let R0 = 1

#让R1等于目标地址

R1 = &addr

R1 = RAM[R1] # let R1 = a + 0

#向目标地址写入待存入的数

RAM[R1] = R0 # let \*(a+0) = 1

解决了如何向数组a中写入数据，下一个需要解决的是while逻辑。在指令集中，只有大于和等于两种判断，若要实现while i<T的循环，即要在T-i>0时跳转入循环，如下的逻辑所示。

(START)

R1 = T - i # 假设已经通过一系列操作使得R1 = T - i

if R1>0 jump LOOP

...... # 循环结束之后的代码

(LOOP)

...... # 循环内部的代码

jump START

如上面所示，当T-i>0时，跳转到LOOP，并在代码执行结束后回到START；当T-i>0不再为真时，才会运行循环结束后的代码。

最后一个需要考虑的问题是如何实现a[i+2] = a[i+1] + a[i]，短短的一行其实包含了大量的运算，如a+i、a+i+1、a+i+2、Ram[a+i]、……、a[i+1] + a[i]等。这样的操作不是两个寄存器就能实现的，需要不断的向Ram内存中写入数据。在这里我们需要定义三个常数addr、addr2、addr3，作为地址存储a+i、a+i+1和a+i+2的结果存到内存中。最后通过反复的存取来完成计算。

RAM[&addr] = a + i

RAM[&addr2] = a + i + 1

RAM[&addr3] = a + i + 2

# 上面的代码是简略的

# 将a[i]存入R0

R1 = &addr

R1 = RAM[R1] # R1 = RAM[&addr] = addr = a + i

R0 = RAM[R1] # R0 = RAM[addr] = a[i]

# 将a[i+1]存入R1

R1 = &addr2

R1 = RAM[R1] # R1 = a + i + 1

R1 = RAM[R1] # R1 = a[i + 1]

# 将计算结果存入R0

R0 = R0 + R1

# 将计算结果存入RAM[a + i + 2]

R1 = &addr3

R1 = RAM[R1] # R1 = a + i + 2

RAM[R1] = R0 # a[a + i + 2] = R0 = a[i + 1] + a[i]

即使在指令集指令非常残缺的情况下，也仍可以用基本的指令间接实现复杂的功能，最终得以计算斐波那契数列的代码。将上面的代码拼合起来，可以得到总的程序代码。

我们在上面的示范中提到过，a、&addr等都是已经定义过的地址常数。通常情况下，编程时是不需要定义这些值的，编译器或者汇编器可以自动的给它们分配一个地址。但是现在进行人工转译的时候，就有必要预先设定好这些常数的值。

例如，我们要计算10个斐波那契数，那么就要保留10个地址用来存储这些数，于是将a定义为0，表示数组从地址0开始。一直到9结束。规定&T = 10，&i = 11，&addr = 12，&addr2 = 13，&addr3 = 14，使得我们之后将汇编语言转写成机器码时，只需要将这些常数替换为对应的数即可。

#address

a=0

&T=10

&i=11

&addr = 12

&addr2 = 13

&addr3 =14

R1 = 1

R0 = R1

R1 = a

M[R1] = R0 # a[0]=1

R0 = R1

R1 = 1

R0 = R0 + R1

R1 = &addr

M[R1]=R0 # addr = a + 1 = 1

R1 = 1

R0 = R1

R1 = &addr

R1 = M[R1]

M[R1]=R0 # a[1]=1

R1= 8

R0=R1

R1 = &T

M[R1] = R0 # T = 8

R1= 0

R0=R1

R1 = &i

M[R1] = R0 # i=0

(START)R1 = &T

R0 = M[R1]

R1 = &i

R1 = M[R1]

R1 = -R1

if(R0+R1>0)jump LOOP # While i<T:

(OVER)R1 = 10

jump OVER

# Go on next page

(LOOP)R1=a

R0=R1

R1= &i

R1=M[R1]

R0 = R1 + R0

R1 = &addr

M[R1]=R0 # addr = a+i

R1 = 1

R0 = R0+R1 # R0 = a+i+1

R1 = &addr2

M[R1]=R0 # addr2 = a+i+1

R1 = 1

R0 = R0+R1 # R0 = a+i+2

R1 = &addr3

M[R1]=R0 # addr3 = a+i+2

R1 = &addr

R1 = M[R1]

R0 = M[R1] # get the value of \*(a+i)

R1 = &addr2

R1 = M[R1]

R1 = M[R1] # get the value of \*(a+i+1)

R0 = R0 + R1

R1 = &addr3

R1 = M[addr3]

M[R1]=R0 # a[i+2]=a[i]+a[i+1]

R1 = &i

R0 = M[R1]

R1 = 1

R0 = R0+R1

R1 = &i

M[R1]=R0 # i = i+1

jump START

这上面的代码中，还包含了大量的标签，如(START)(LOOP)(OVER)等，跳转时，就是要“jump”到这些标签所在的行数。

最后我们将这些代码翻译为可以被机器识别的二进制机器码即可。（仅展示开头）

00000000//R1=a

10100000//R0=R1

00000000//R1= 0

10101000//R0 = R1 + R0

00001100//R1 = &addr = 12

11001100//M[R1]=R0//addr = 0

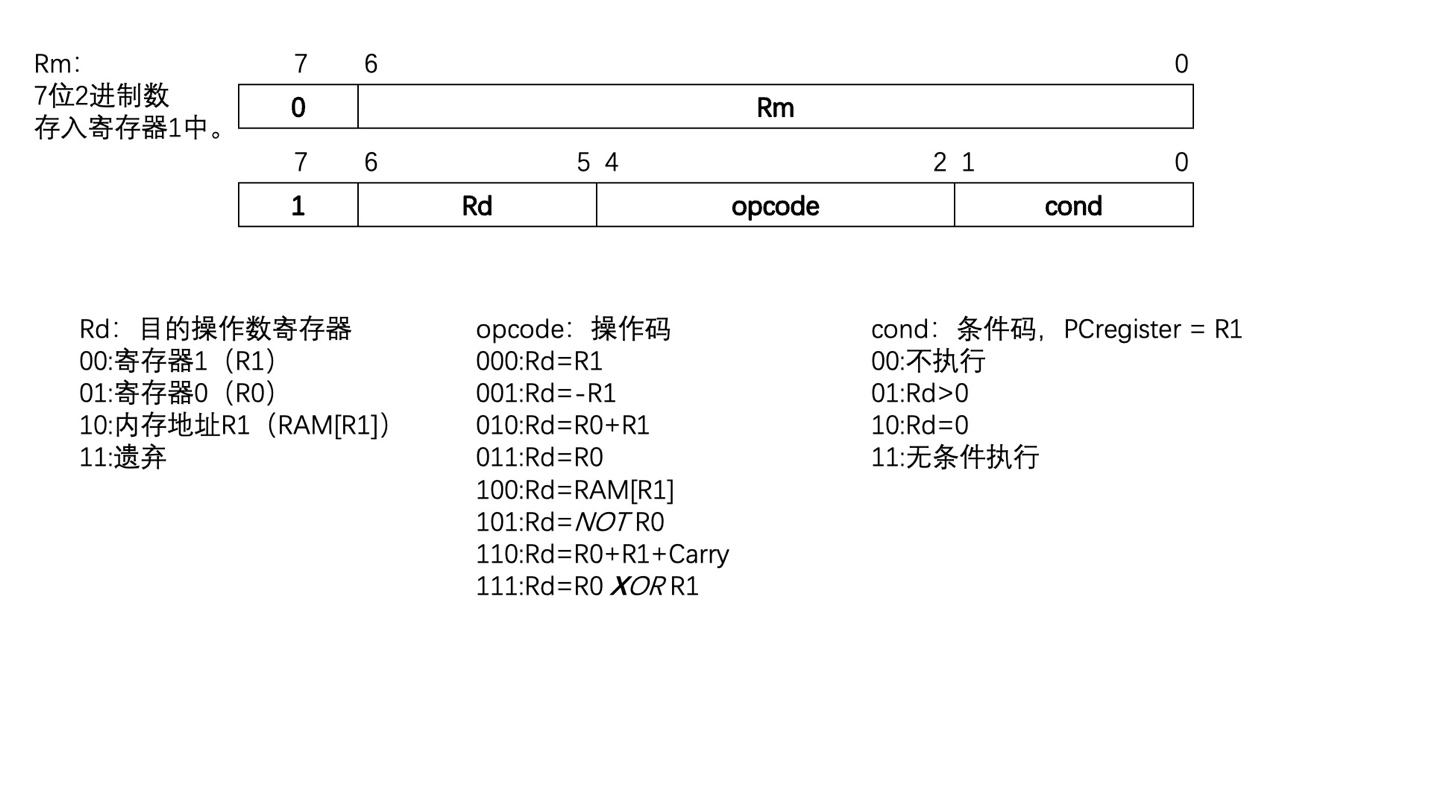
......后面的内容略......

## 2、支持大于1字节计算的指令集改进设计

在上面的指令集版本中，只有半加法（不考虑进位），因而只能对1字节（8比特）的数据进行计算，且无法判断计算是否发生溢出进位。

为了解决此问题，特地引入全加法，和一个进位寄存器（Carry\_Flag)。在ALU计算半加法时，只会计算A+B；而当ALU计算全加法时，则会计算A+B+Carry，且进位结果会被写入Carry。当处理非全加法运算时，Carry的结果保持不变。

受限于opcode操作码只有3位，ALU只能处理8种指令，因此，在这个改进版本中，将不常用的与运算和或运算优化为了更常用的全加法运算和异或运算。

图2.2.1

这里的全加法运算是对无符号数的计算。

斐波那契数列从第14个数开始，就大于255了，因此，原始版本的指令集将无法继续计算。但是，引入了全加法之后，无符号数的数据大小可以不再局限在1个字节内。

无符号整型默认的大小是4个字节，我们就按照4字节的数据处理，以计算第20个斐波那契数为例，演示全加法是怎么工作的。

\*a = 0

a[0] = 1

a[4] = 1

i = 0

T = 72

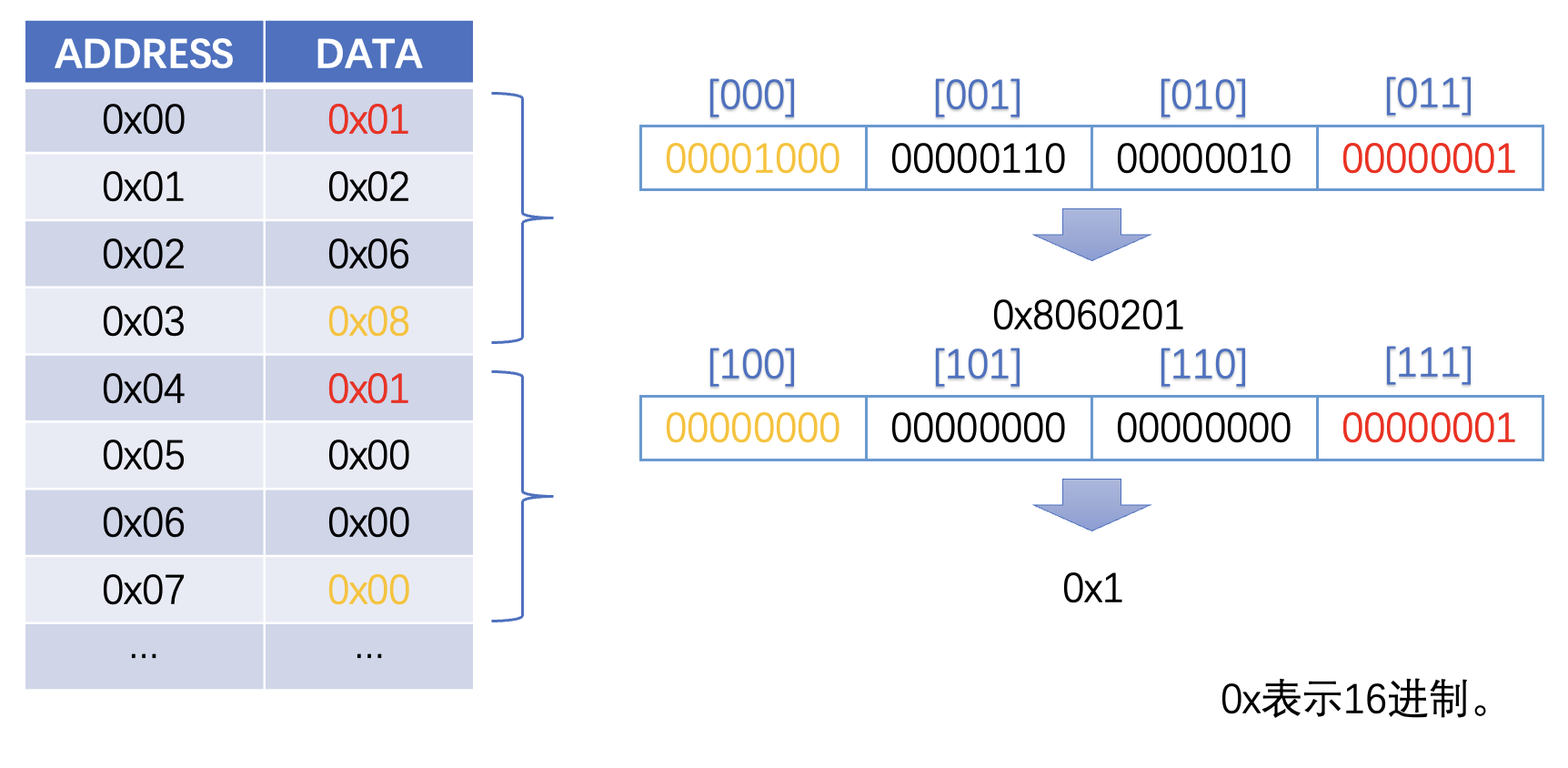
while i<T:

for j in range(i,i+4):

a[j+2] = a[j+1] + a[j]

i = i + 4

我们再次搬出这个伪代码，它和上面的代码相比，需要作出几个改进。本来只用1个字节存储斐波那契数时，a[0]就表示第一个斐波那契数、a[1]表示第二个斐波那契数……以此类推；但是现在我们用4个字节存储斐波那契数，因此，a[0]到a[3]都表示第一个斐波那契数。存储方式采用小字节序（小端序）存储，地址最小的内存位置存储数据的最低位，较大的地址存储更高位。这种存储方式虽然不符合人的阅读习惯，但是在计算时更加直观。

  
图2.2.2

所以，写入第一个斐波那契数1时，就是要a[3:0]=1，即a[3]=0，a[2]=0，a[1]=0，a[0]=1。同理第二个斐波那契数也是1，所以a[7:4]=1。最后，20个斐波那契数一共会占据80个地址，从a[0]开始，a[79]结束。

#address

a = 0

&T = 80

&i = 81

&addr = 82

&addr2 = 83

&addr3 = 84

&j = 85

变量和常量的声明与先前的代码一致，需要注意的是还多了一个j变量，这在之后会被用到。

接下来是对a数列进行初始化，寄存器在初始状态下为0，因此只需要对a[0]=1，a[4]=1即可。此外，通过调整代码的执行顺序，可以优化一行代码。

# RAM[a]=1，再令addr= a+4，

# 使得RAM[addr]=1。

R1 = 1

R0 = R1

R1 = a

M[R1] = R0 # RAM[a]=1

R0 = R1 # R0 = a

R1 = 4

R0 = R0 + R1 # R0 = a+4

R1 = &addr

M[R1]=R0 # addr = a+4

R1 = 1

R0 = R1 # R0 = 1

R1 = &addr

R1 = M[R1]

M[R1] = R0 # RAM[a+4]=1

# 通过改变顺序，可以少写一行

# 代码。

R1 = a

R0 = R1 # R0 = a

R1 = 4

R0 = R0 + R1 # R0 = a+4

R1 = &addr

M[R1]=R0 # addr = a+4

R1 = 1

R0 = R1

R1 = a

M[R1] = R0 # RAM[a]=1

R1 = &addr

R1 = M[R1]

M[R1] = R0 # RAM[a+4]=1

接下来和原始代码相比比较大的修改是加法需要加4遍。

(LOOP)R1 = 0

R0 = R1

R1 = R0 f+ R1 # flush carry\_flag = 0

直接看到LOOP循环，对于全加法，首先需要将进位标志寄存器清零。在LOOP循环内部，还要有一个FULLADD小循环。小循环本来是for(j=0;j<3;j++)，可以用下面的方式间接实现。

R1 = 2

R0 = -R1

R1 = &j

M[R1] = R0 # j = -2

(FULLADD)if j>0 jump FULLADDEND

......... # 小循环中的代码

jump FULLADD

(FULLADDEND)R1 = &i

R0 = M[R1]

R1 = 4

R0 = R0+R1

R1 = &i

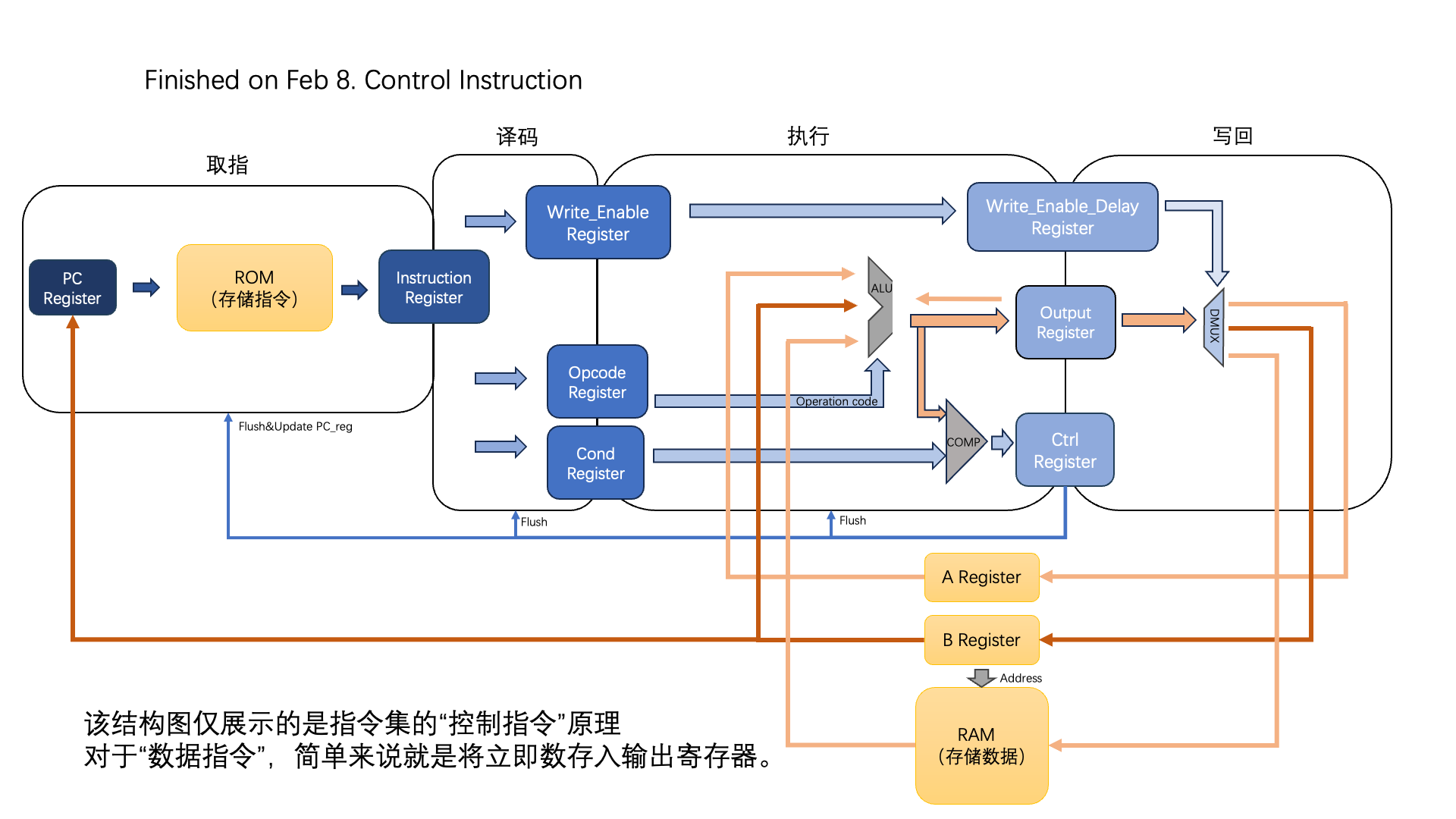
M[R1]=R0 # i = i+4

jump START

在“小循环中的代码”，我们使用之前的方法完成加法。特别的，利用“R0 = R0 f+ R1”代替“R0 = R0 + R1”，字母f表示全加。半加法的进位不会影响进位标志寄存器；全加法会把进位标志寄存器作为被加数之一，且计算结果会改变标志进位寄存器。

# 三、CPU结构

下图是CPU的“控制指令”（C指令）的流水线逻辑图，展示了CPU的流水线是如何工作的。

图3.0.1

这幅图展现出了CPU的四级流水线过程。

## 1、取指 Fetch

PC寄存器（Program Counter，程序计数器）是CPU中的一个控制单元，在默认情况下，它是一个计数器，在每个时钟周期数值加1；但是为了服务跳转指令，它也能够加载其他数值。

ROM（只读存储器）存储了操作指令，是预先设定好的。它根据PC寄存器提供的地址导出对应地址的指令，因此这个阶段叫做取指，取出来的指令会被存入“数据寄存器”中。

## 2、译码 Decode

译码阶段就是解析指令，因为这个指令集比较简单，所以译码时只是将指令拆分为了三个部分。

（这个译码阶段仅仅对指令做了一级延迟，但是没有任何的处理。不过，当需要优化处理器运行速度、扩大吞吐量时，这一级延迟将会派上用场。）**？？这里存疑，好像除了创造了一级延迟以外，没有排上任何用场？？**

## 3、执行 Execute

A Register和B Register 分别是指寄存器0和寄存器1。

ALU单元（Arithmetic Logic Unit，算术逻辑单元）是CPU的核心单元，是一个非时序的电路，它接受3位操作码，将寄存器0、寄存器1或者内存的数据进行特定的计算并输出。

RAM（随机存取存储器）是保存数据的地方。正如第二节的介绍，ALU只能读取RAM[R1]，所以RAM的地址实际是由寄存器1提供的。

ALU的输出结果会被存储到“输出寄存器”；同时也会和0做比较，并根据条件码决定控制寄存器的值。

控制寄存器是专门为跳转指令服务的，当需要跳转时，PC寄存器会传入寄存器1的值。此外，由于跳转指令的取指到控制寄存器的输出有3级延迟，在这期间，PC寄存器仍在计数，不需要的指令正在进行取指、译码和执行操作，必须要把这些指令从流水线中清空。

## 4、写回 Writeback

从译码阶段就决定好的写入寄存器（Write\_Enable Register，其实有三个寄存器，分别为寄存器0、寄存器1、RAM服务)，经过一级延迟后，在写回阶段决定了输出寄存器数据最后的目的地。

## 5、技术性问题优化

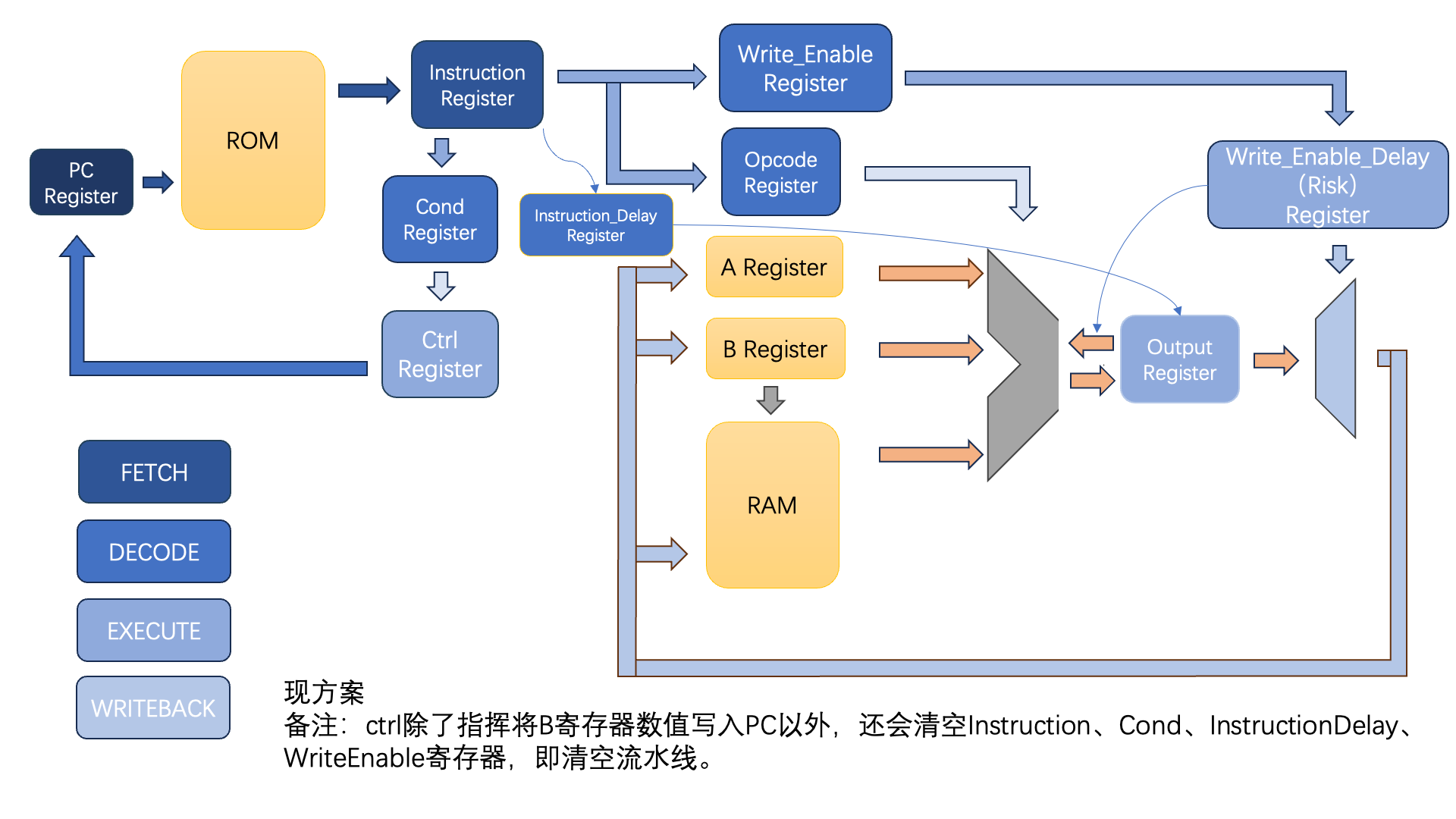
  
图3.5.1

图3是另一种画法表示的CPU结构图，它没有说明控制寄存器（Ctrl Register）是如何工作的，但是展示了“数据指令”（D指令）的工作流程以及CPU如何对抗因为写回延迟产生的“冒险”问题。

### （1）“数据指令”工作流程

存在一级指令寄存器的延迟（Instruction\_Delay Register)，从而在执行操作写入“输出寄存器”时，通过判断是“数据指令”还是“控制指令” （如上面所说数据指令就是将指令本身当作一个立即数存入R1寄存器），从而选择接收立即数还是ALU的输出结果。

### （2）数据冒险

写回级在写回数据的时候，存在延迟，如果下一行指令就需要使用尚未写回的数据，直接运算显然会产生错误。CPU直接将写入寄存器的延迟视为发生冒险的标志，在出现冒险时，优先从输出寄存器取值，从而解决了冒险的问题。

### （3）二位动态本地分支预测

在理想情况下，流水线可以在每个时钟周期吞入一个新的指令，同时结束一个旧的指令。因此，虽然一个指令从运行到结束需要4个周期，但是由于吞吐量大，使得在每个时钟周期，都可以结束一个指令。

不过当面对跳转指令时，产生的跳转将会导致原本流水线中的所有指令失效，不得不清空流水线以继续进行正常的工作。

在跳转指令执行完毕前，继续吞入新的指令，如果不需要跳转，就让CPU继续正常的执行下去；如果需要跳转，就让CPU清空流水线。这其实就是一种单方向的静态预测：总是预测条件跳转不发生。

动态预测器则可以记忆过去的跳转历史，从而在下一次遇到相同的跳转时，立即跳转而不必等待指令处理完成。

其原理是，建立一个分支历史表（Branch History Table, BHT)，记录了哪一行指令是跳转指令（即BIA，分支指令地址）、是否需要跳转（即CS，当前状态）、需要跳转到哪一行指令（即BTA，分支目标地址）。

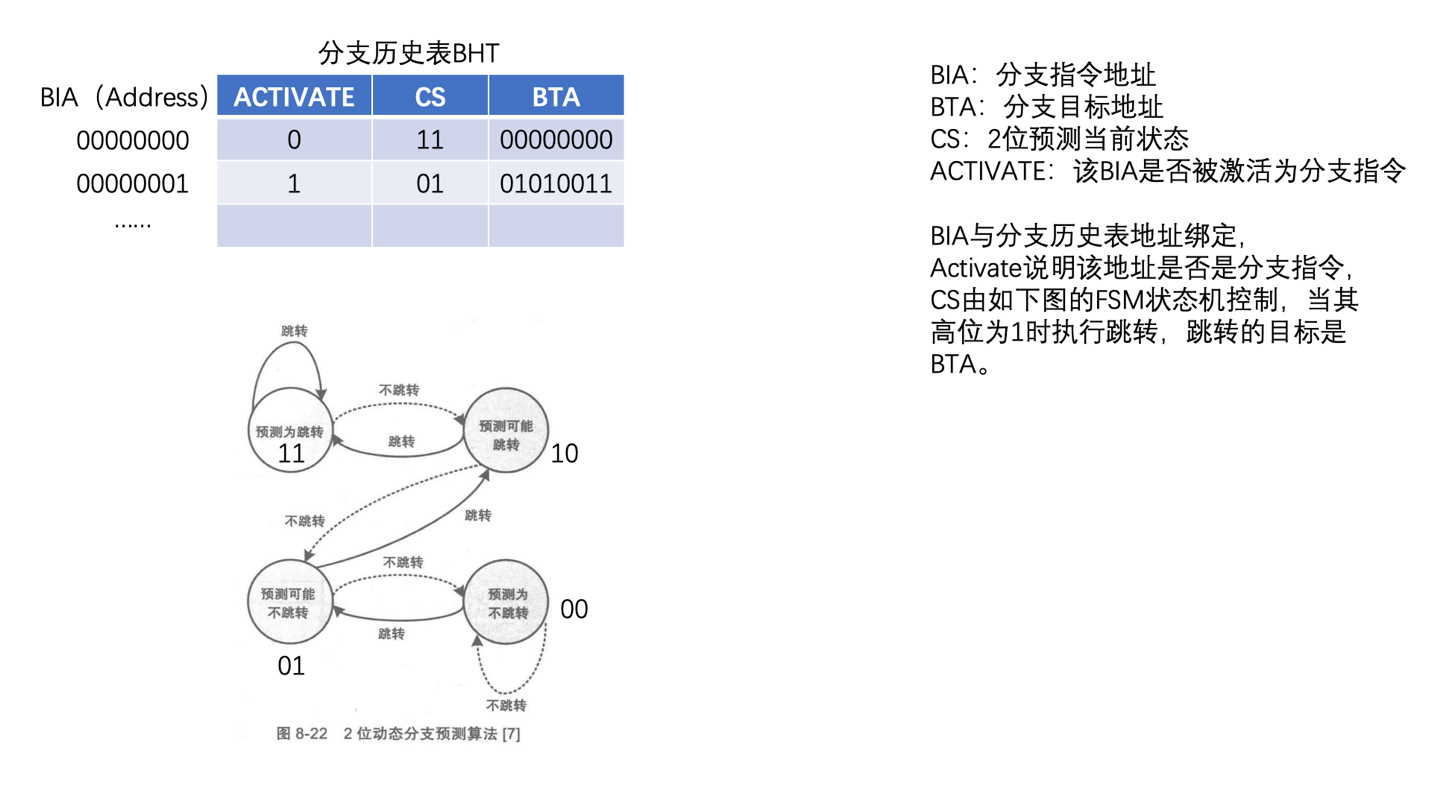
BIA会被作为分支历史表的索引。在32位系统中，如果需要完整记录32位的分支历史，所需要的存储器规模是非常恐怖的；所以通常会取指令的后12位作为索引，那么存储器的规模就可以小很多了。

在这个8位指令集的CPU系统中，地址只有8位，直接使用完整的BIA作为索引是完全可以接受的。接着通过表格中的Activate字段，判断索引是否是跳转指令。

经过研究发现，1位动态预测（CS只有1位，为1时跳转，为0时不跳转）的效果不如2位动态预测（CS有2位，为11或10时跳转，为01或00时不跳转）。

2位CS通过状态机FSM控制，如图4所示。也就是说，只有当连续两次预测错误时，动态预测器才会改变自己的预测。2位预测的效果是最好的，高于3位或更多位的预测。

分支历史表在每个时钟信号的上升沿都需要及时的输出预测结果，并没有时间写入被更新数据。所以，只能在时钟下降沿进行写入操作。

  
图3.5.2（左下角图来源于数字逻辑设计与计算机组成，机械工业出版社）

# 四、Verilog实现

主要使用Xilinx的Vivado进行设计。

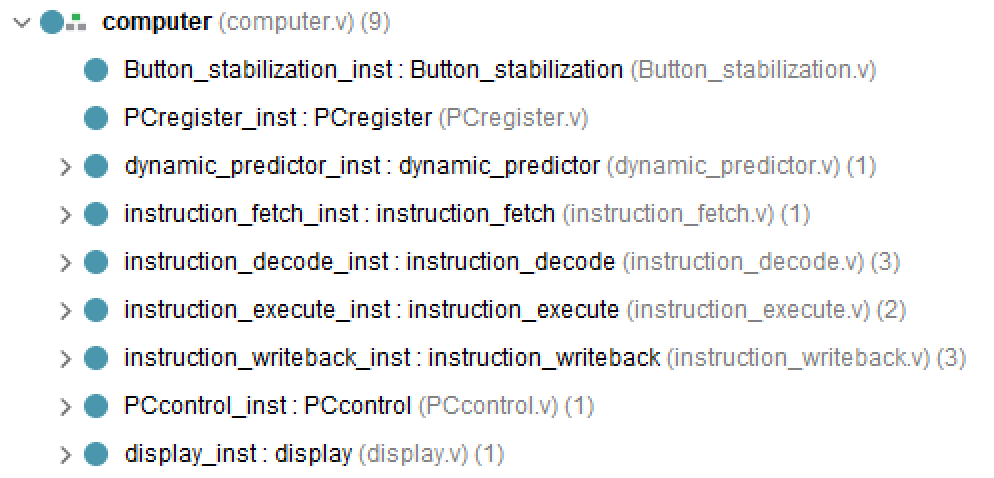


图4.0.1

程序架构将流水线的四个层级模块化，只有PC寄存器和控制寄存器独立在模块外。顶级模块下还有按钮防抖和输出显示模块。

1. 低级语言和高级语言的区别 - ps学习的文章 - 知乎https://zhuanlan.zhihu.com/p/65746260 [↑](#footnote-ref-1)
2. 英特尔在家用CPU上同样使用AMD64指令集，因其自研的64位指令集不兼容x86。 [↑](#footnote-ref-2)