

Lab7 流水线 MIPS CPU 设计

唐凯成 PB16001695

一. 实验目的

- 1、自行设计一个流水线 MIPS CPU
 - 理解流水线 CPU 的设计准则
 - 自行设计流水线 CPU 的数据通路，实现基础的 MIPS5 级流水
 - 学习流水线 MIPS CPU 的信号控制与信号的寄存模块
 - 思考流水线、多周期与单周期的本质区别
 - 处理流水线的相关与冒险，加入转发的数据通路与 Forwarding 模块
 - 处理 Lw 的数据相关，加入互锁 interlock 模块
 - 处理分支时的控制相关，默认分支失败从失败处调度
- 2、增加流水线所能执行的指令条数（最终实现 48 条指令）
 - 根据原本的数据通路添加基础的逻辑运算指令
 - 适当添加 ALU 的输出信号来增加更多分支指令
 - 修改数据通路并添加模块来添加更多访存和跳转指令
- 3、根据设计的 CPU 实现下载，在实验板上完成指令执行和内存数据的输出
 - 实现下载并成功运行指令
 - 将内存中存储结果自动滚动输出在数码管上
 - 根据开关手动输入地址，显示对应内存地址的内容

二. 实验平台

EDA 工具：Xilinx ISE14.7

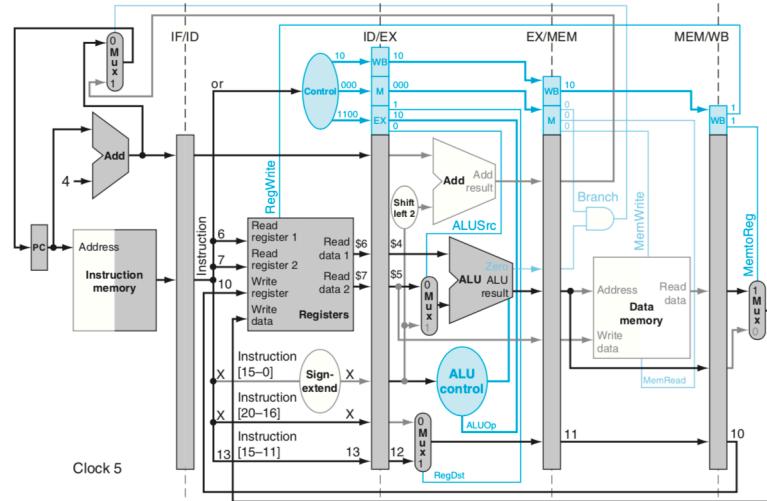
Digilent Adept 烧写软件

Nexys4 工作板

三. 实验过程

1、五级流水线基础数据通路的设计

对于基础的流水线数据通路，由类似单周期 CPU 的数据通路调整而成，本次实验初期使用如下的流水线数据通路：



2、对数据通路中各个流水段内模块的设计

(1) IF 段

a) PC 模块:

对于 PC 模块，主要为一个随时钟周期变化的锁存器。与单周期主要的区别为加入了使能输入与重置输入。重置输入给出每次指令存放的起始地址，本次设计中在重置信号 reset 变为 1 时将地址置 0，整个 CPU 开始执行第一条指令。(有关使能输入在后面 interlock 设计内补充)

b) 指令存储器:

指令存储器直接由 IP 核生成，由于指令存储器内内容无需修改，切为了保证每条指令不会因为错误操作被修改，故选用 Distribution Memory 类型生成 ROM (只读存储器)，此存储器可实现同步读，输出随输入地址直接变化，且无时钟周期信号，不受时钟信号影响。

另外由于取出时为一次性取出 32 位指令，使用 IP 核难以合成 8 位的指令，故直接使用 128*32bit 的 ROM，每个单元存放一条指令或数据，而为了处理本来指令中对 PC 地址均按照 4 的倍数自加的问题，在 ROM 地址输入端将输入地址右移两位，即处理了对应的地址自加错误的问题。

c) 下一 PC 地址的数据选择器

二选一的 32 位数据选择器，由 Branch 与 Zero 信号的与运算构成该选择器的信号输入，0 代表使用上一地址自加获得的地址，1 代表分支成功时应转跳的地址，输出结果即为 PC 的下一个新地址。

d) IF/ID 段寄存器

在流水线中为了完成每个指令在每个周期完成一个流水段的要求，需在 5 个流水段的分段间隙中加入段寄存器，主要用于寄存需要输入下一流水段的各数据，各个段寄存器中由一系列子寄存器组成，为了完成后续对流水段的清空操作，子寄存器加入重置信号。

对于 IF/ID 段寄存器，只需寄存两个数据：当前指令 (32 位) 和下一个 PC 值 (32 位)。(对于该段寄存器的使能控制在后续加入)

(2) ID 段

1) 主寄存器 REG 模块:

Reg 模块的设计基本与之前相同，加入了 reset 对寄存器文件的初始化以及另外添加了一个对于 0 号寄存器的优化：即\$zero 寄存器中始终存储 0，且无法写入 0 号寄存器。

2) 信号控制模块:

流水线 CPU 信号控制模块的设计主要类似单周期 CPU 的设计，即在 ID 段中就给出所有的输出信号，初始的基础控制信号主要有以下几个：

a) REG 段控制信号

ExtSel: 位拓展器控制信号，1 代表有符号拓展，0 代表无符号拓展。

b) EX 段控制信号

ALUSrc: ALUB 的输入选择信号，用以选择使用 reg 读出数据和立即数。

RegDst: Reg 写入地址的选择信号，用以选择 rt 或 rd 输入寄存器写入地址。

ALUop: 2 位 ALU 运算控制信号，输入 ALUcontrol 模块中得到二级控制信号。

c) MEM 段控制信号

Branch: 分支指令的使能信号，与 Zero 进行与运算来更改 PC 的使能信号。

MemWrite: 数据存储器的写使能控制信号。

MemRead: 数据存储器的读使能控制信号。

d) WB 段控制信号

MemtoReg: 写回寄存器的选择信号，用以选择写回存储器数据或 ALU 结果。

RegWrite: 寄存器的写使能控制信号。

对于不同指令信号对应的值同单周期，考虑到后期还有许多需要拓展的信号，故此处暂不列出对应指令的信号输出表。

对于输出的后面三个分段的信号，在每个流水段寄存器中再分级寄存，以此来实现信号随流水段分配到各个模块中。

3) 立即数位数扩展器：

该模块用于将 16 位立即数拓展为 32 位立即数，但由于不同情况下需要进行有符号和无符号扩展，故在扩展时，若为无符号扩展，只需在高 16 位补 0 即可；若为有符号扩展，判断最高位第 16 位是否为 1，为 1 时在高 16 位补 1，不为 1 时在高 16 位补 0。如此即可完成对 16 位立即数的符号扩展。

4) ID/EX 段寄存器：

对于 ID/EX 段寄存器，需寄存从信号控制模块传输来的 EX、MEM、WB 三段信号，以及下一个 PC 值、寄存器两个端口的读出值、位拓展后的立即数以及指令中的 rt 和 rd。

(3) EX 段

1) ALU 模块：

ALU 模块也大体保持之前 CPU 中的设计，对于 zero 输出改为使用 assign 语句而不再和 case(op) 放在一个 always 语句中，使程序结构更清晰。但对于更多指令中 ALU 还需要添加一系列信号，在后文中提出。

2) ALU 信号控制模块：

大体同之前的设计，由 ALUop 信号和 funct 生成对应的 ALU 控制信号。

3) Reg 地址的数据选择器：

用于选择 rt 还是 rd 作为传入下一阶段的地址，信号控制为 RegDst。

4) EX/MEM 段寄存器：

对于 ID/EX 段寄存器，需寄存从上一段寄存器传输来的 MEM、WB 两段信号，以及下一个 PC 加左移两位立即数的值、ALU 的输出结果、Zero 的输出结果、从寄存器第二个端口读出的数据、经选择器选的出的地址。

(4) MEM 段

1) 数据存储器：

数据存储器同样由 Distribution Memory 生成，具体选项选择 Simple port RAM，即单端口的可写入存储器，注意设置一个读出的使能，用以输入 MemRead 信号。

2) MEM/WB 段寄存器

对于 ID/EX 段寄存器，需寄存上一段寄存器传输来的 WB 段信号，以及数据存储器的读出结果、ALU 的输出结果、从上一段寄存器传入的地址。

(5) WB 段

写回内容的数据选择器：

用以选择写回寄存器的内容，控制信号为 MemtoReg。

(6) 顶层模块 PipelineCPU 的设计

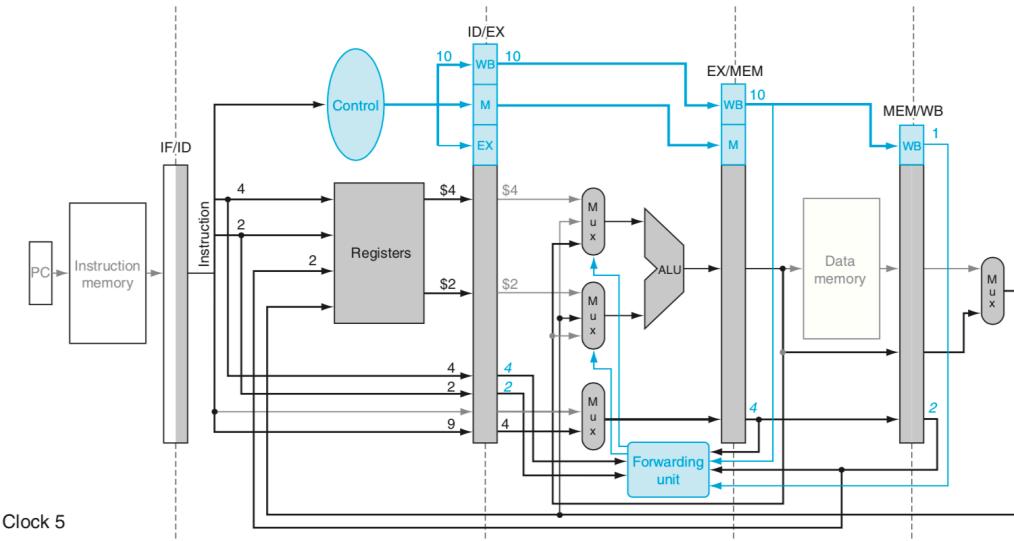
对于顶层模块，主要为对各个流水段及对应段内模块的连线，以及对拓展后立即数的左移两位等操作。

3、对于流水线 CPU 的相关问题的处理与模块设计

在完成了流水线基本数据通路的设计以后，还需处理诸多流水线中的相关问题，对于结构相关，由于是硬件的设计问题，故暂不考虑；对于数据相关，分为 forwarding 和 interlock 两块来处理。对于分支相关，使用默认分支不成功的情况，成功时清空 EX 段前的流水线。

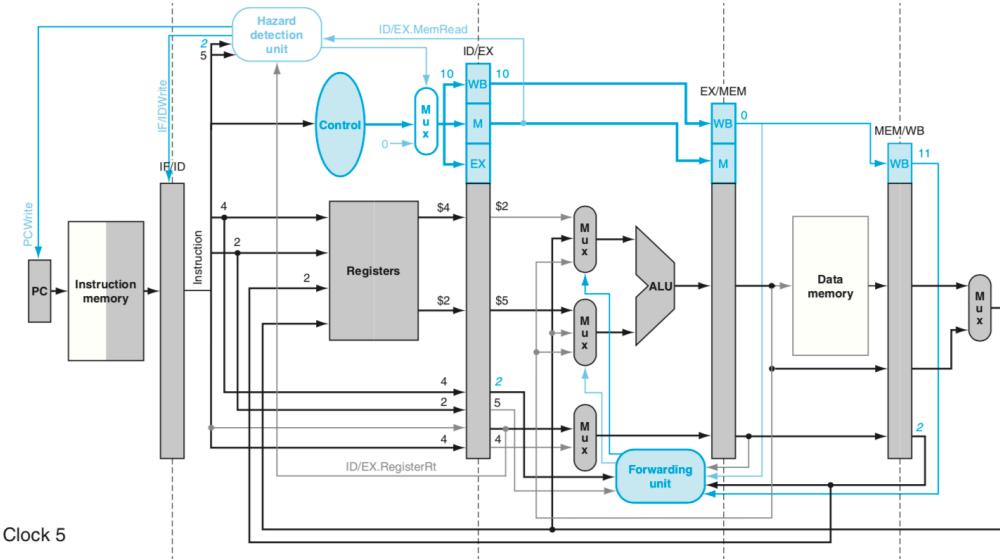
(1) 转发模块与数据通路的设计

使用转发的数据通路来处理大多数数据相关问题，设计的数据通路如下图，新建一个 forwarding 模块用以判断各个数据通路间是否需要转发，当转发条件达成时，即连接对应的数据通路，通过增加两个三端口数据选择器来完成选择。不需要转发时，默认直选原数据通路的数据。



(2) 指令锁的设计

对于跨多周期的 1w+R 型指令，由于当 R 指令读出的寄存器数据进 ALU 时，1w 指令所需取出的数字也仍未读出，故即使使用转发也无法处理改相关，只能使用一个指令锁结构，在 1w 后向流水线中插入一个气泡指令来使流水线暂停工作一周期，此时应控制 PC 和 IF 段寄存器保持之前的值不变，转而向 ID 段寄存器的信号寄存部分插入一个全 0 的信号，即可实现插入一条气泡，从而使流水线暂停工作一周期来解决对应的相关问题。



4、增加命令所需调整的数据通路

(1) 增加各类算术逻辑指令

1) 实现指令:

算术逻辑类: add、addu、sub、subu、and、or、nor、xor

立即数运算类: addi、addiu、andi、ori、xori、lui

位运算类: sll、sllv、srl、sriv、sra、srav

比较类: slt、sltlu、sltui、sltui

2) 具体实现方式:

对于要加入的基本算术逻辑类指令, 原 ALU 中也已基本实现, 只需加入对应的运算和在 ALUControl 中的判断语句即可。

对于立即数运算类, 由于要加入的指令较多, 而本来的 ALUop 只有 2 位, 最多只能再增加三个立即数运算, 而此处为了添加四个, 只能将原 ALUop 的位数改为 3 位, 即可在 ALUControl 中直接将对应的运算输出对应算术逻辑类指令的信号。对于较为特殊的 lui, 该指令将立即数的低 16 位存入 rt 寄存器的高 16 位(低 16 位补 0), 因此在 ALU 中加入一个运算 ALU_LUI, 直接将输入端 B 的低 16 位赋值给 alu_out 的高 16 位, alu_out 的低 16 位直接赋值为 0, 再将次结果写回即可。

对于位运算类, 需在 ALU 中新增位运算的控制, 对于逻辑左移与逻辑右移, 直接用<<与>>即可。而对于有些特殊的算术右移, Verilog 中本来设置有算术右移符号>>>, 但实际运用时存在问题, 查找相关资料后得知对前者的 alu_a 前端加上\$sighed(alu_a) 后即可使用>>>。

对于比较类, 直接在 ALU 中添加运算, 当 alu_a 大于(或小于) alu_b 时, alu_out 赋值为 1, 否则赋值为 0 即可实现。

(2) 增加存数和取数指令

1) 实现指令:

存数指令: sw、sb、sbu

取数指令: lw、lb、lbu、lh、lhu

2) 具体实现方式:

对于存数与取数指令不同位数的选择, 由于数据存储器设置为 32 位且一次就会读出 32 位, 所以可添加模块来根据不同的存取数模式将本来读出的 32 位数字改为所要求的 8、16、32 位数, 具体的实现主要由组合逻辑即可实现, 不过由于需要保存存数和取数的信号信息, 所以需在 MEM 阶段插入两个信号, 存数信号 store_mode(2 位) 与取数信号 load_mode(3 位), 再加入两个数组组合逻辑模块, 即可实现对应的不同模式存取数。

(3) 增加分支指令

1) 实现指令:

分支指令: bltz、beq、bne、blez、bgtz

分支并链接指令: bgezal、bltzal

2) 具体实现方式:

由于之前要实现的分支类指令只有一种, 故 Zero 输出的要求为固定的, 不需要多余的信号输入, 而现在对于不同的分支指令, 虽然大多是用 alu 的 sub 运算, 但其 Zero 输出 1 或 0 的条件要根据不同指令而变化, 因此要对 ALU 加入一个 Zero 输出条件选择的 AluzeroCtr 控制信号, 由于模式较多, 故该信号为 3 位且应插入 EX 段信号中。根据该信号的不同, ALU 中即可选择 Zero 输出 1 时

对应的要求，即到底是在 alu_out 等于 0 时输出 1 还是不等于 0 时输出 1，以及对于两条链接指令，需直接判断 alu_a 与 0 的关系。

对于后面的两天分支并链接指令，即分支成功时还需把分支的 PC 值存入 31 号寄存器中，故设计此时添加一个信号 connect 在 MEM 段中，用以确定是否要链接，当需要链接时 connect 置为 1，代表需要链接。此时再在 MEM/WB 段寄存器前加入一个 2 选 1 数据选择器，控制信号即为 connect，当信号为 0 时写回地址仍未之前 RegDst 确定的地址，信号为 1 时写回地址直接为常数 31。

确认了链接时写回的寄存器之后，还需解决此时的 RegWrite 的信号问题，由于之前 RegWrite 的信号在默认时为 0，且由于存在分支不成功无需链接的可能，所以不能在 Control 模块中直接把 RegWrite 置为 1。个人解决该方法的问题为：将 connect 信号从 MEM 段再寄存一次进入 WB 段中，此时在输入寄存器的写入使能信号前加上一数据选择器，信号控制即为寄存后的 connect，当 connect 为 0 时保留原来的 RegWrite 信号值，当 connect 为 1 时直接输出常数 1，如此即可确保分支不成功时不会错误地把地址写回寄存器中。

另外由于本来在 WB 段已没有分支成功时的 PC 地址，故还需要将分支成功的地址通过段寄存器寄存到 WB 段中，同时修改 MemtoReg 信号为 2 位，即可多选择一个 PC 地址作为写回的数据可能之一。

最后由于分支指令目前为预测分支失败，所以当分支成功时，需要清空目前多进入流水线的后面 3 条指令，只需要将 MEM 段寄存器及之前寄存器的 reset_p 置 0，即可将之前已经进入流水线的指令清除，决定 reset_p 的可以简单的为 PCSrc，即为 Branch、Zero 与 jump 组合而成，当 PCSrc 为 1 时，设置 reset 转为 1，当 PCSrc 为 0 时，设置 reset 为 1 保持不变。

(4) 增加跳转指令

1) 实现指令：

j、jr、jal、jalr

2) 具体实现方式：

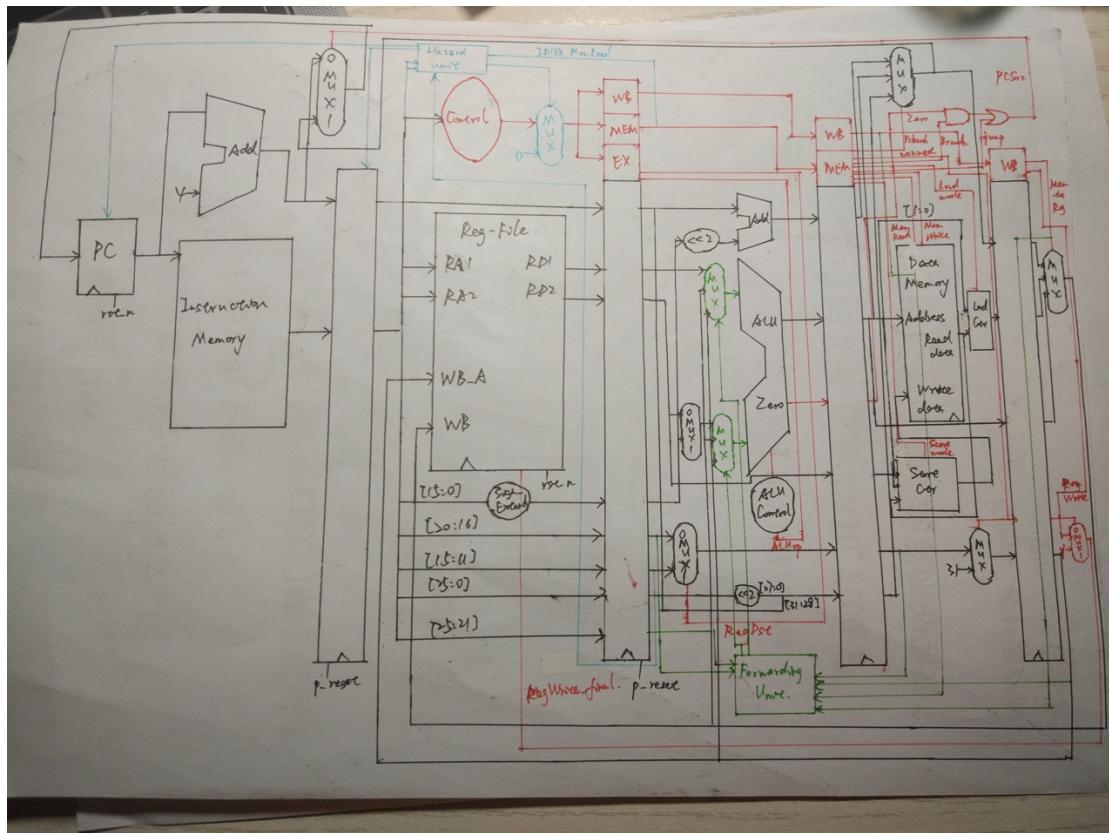
对于跳转指令，由于之前的流水线通路中没有考虑，故需要重新修改通路来实现，主要是将指令的后 26 位寄存并组合原地址到 MEM 段，同时为了实现转跳，加入信号 jump，当指令为转跳指令时，jump 为 1，其余时候为 0。jump 指令插入 MEM 段中，修改本来新地址的选择信号，本来由 Branch 和 Zero 的与运算组成，现在需再并上 jump，即只要 jump 为 1，直接置新地址。

对于 jr 指令，即从寄存器中读取转跳指令 PC，由于其 rt 对应为\$zero 寄存器，故直接设置 ALU 为加运算即可，然后可得 ALU 的运算结果即为新地址。由于此时新 PC 的改变值已有 3 中可能（改变值不包括自加的 PC+4 值），分别为分支成功时地址，跳转的 tar 地址，寄存器存储的地址，故要在 MEM 段插入新信号 PCbackSrc 与一个 3 选 1 数据选择器，用于确认改变后的 PC 值。

对于 jal 与 jalr 的链接指令，和上述分支并链接的指令实现方法相似，估值徐此时设置信号 connect 为 1 即可实现链接。

5、最终的顶层模块和数据通路

增加以上各指令后，再重新对顶层模块进行连线，以及各种信号的控制和少量加和位运算简单操作，下面为本次 MIPS CPU 流水线的最终数据通路（由于电脑绘画较为复杂故以手动绘画，其中各模块的小箭头代表有时钟 clk 输入，输入 ret 代表有重置/初始化输入）。



6、实现下载所需添加及修改的模块

为实现下载需要，即在板子上运行 CPU 执行一段指令，并输出执行完成后对应的数据存储器中的内容，因此再设计一顶层模块在 Pipeline 之上，并加入以下几个模块来实现操作板上的控制：

(1) 时钟减缓模块

用于增加每个时钟周期的时间，以使得八段数字显示器的数字变化在肉眼观察范围内。另外由于需要自动现实存储内的数据，需要再加入一个自动输入读取地址的模块，对于该模块也需要一个减慢的时钟输入。

(2) 按键防抖动模块

由于在每次触碰按键时可能会有多次触碰，故使用按键切换模式时需加入该模块用于确认按键的点按次数并防止抖动的多次输入。

(3) 地址自动读取模块

为实现内存的自动读取，需在每个延缓的时钟周期输入新的读取地址，因此加入该模块用以顺序输出从 1 到 128 的地址，并加入 `rst_n` 信号来实现每次自动读取地址的重置。

(4) 七段式显示器的控制模块

用于控制七段式显示器输出对应的数字（输出结果为 16 进制）。

(5) top 模块的设计

对于该实现下载的顶层模块，加入上述模块后，还需要加入一个模式的选择，使用一个 `always` 结构来时变量 `mode` 在 0 和 1 间切换从而实现显示器在两种模式间的切换。另外对于原 CPU 的顶层模块 `PipelineCPU`，还需略有更改，即加入一个对数据存储器的读取地址，以及对应地址的输出，为不干扰原 CPU 的运行，故将数据存储器改为双端口的 Dual Port RAM，以此来分出一个端口来读取当前下载所需读取地址的数据。

四. 实验实现结果及检测

1、对于之前单周期与多周期斐波拉契数列程序的运行结果：

(1) 斐波拉契数列的指令:

1) MARS 中的指令

```

addi    $t0, $zero, 0
addi    $t5, $zero, 20
addi    $t3,$zero, 3
addi    $t4,$zero, 3
sw     $t3, 0($t0)      # F[0] = $t3
sw     $t4, 1($t0)      # F[1] = $t4
addi   $t1, $t5, -2
loop: lw    $t3, 0($t0)      # Get value from array F[n]
      lw    $t4, 1($t0)      # Get value from array F[n+1]
      add   $t2, $t3, $t4      # $t2 = F[n] + F[n+1]
      sw    $t2, 2($t0)      # Store F[n+2] = F[n] + F[n+1] in array
      addi  $t0, $t0, 1       # increment address of Fib. number source
      addi  $t1, $t1, -1       # decrement loop counter
      bgtz $t1, loop         # repeat if not finished yet.

out:
      j out

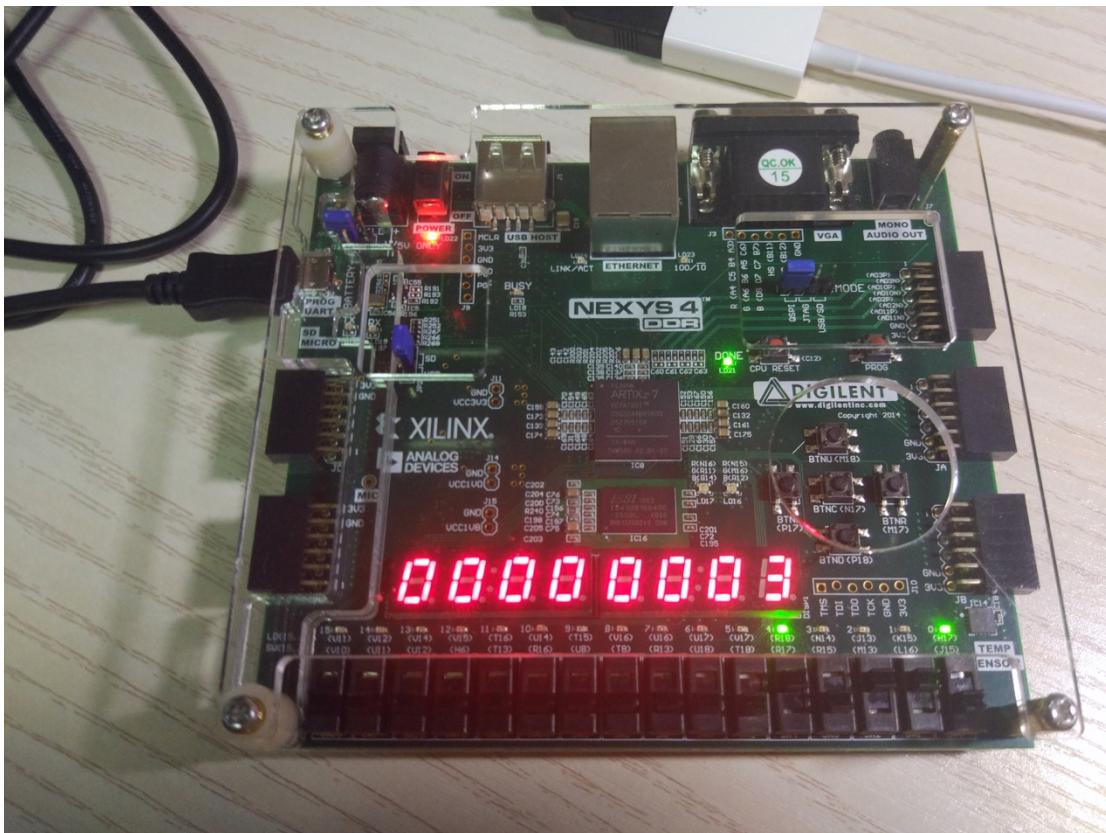
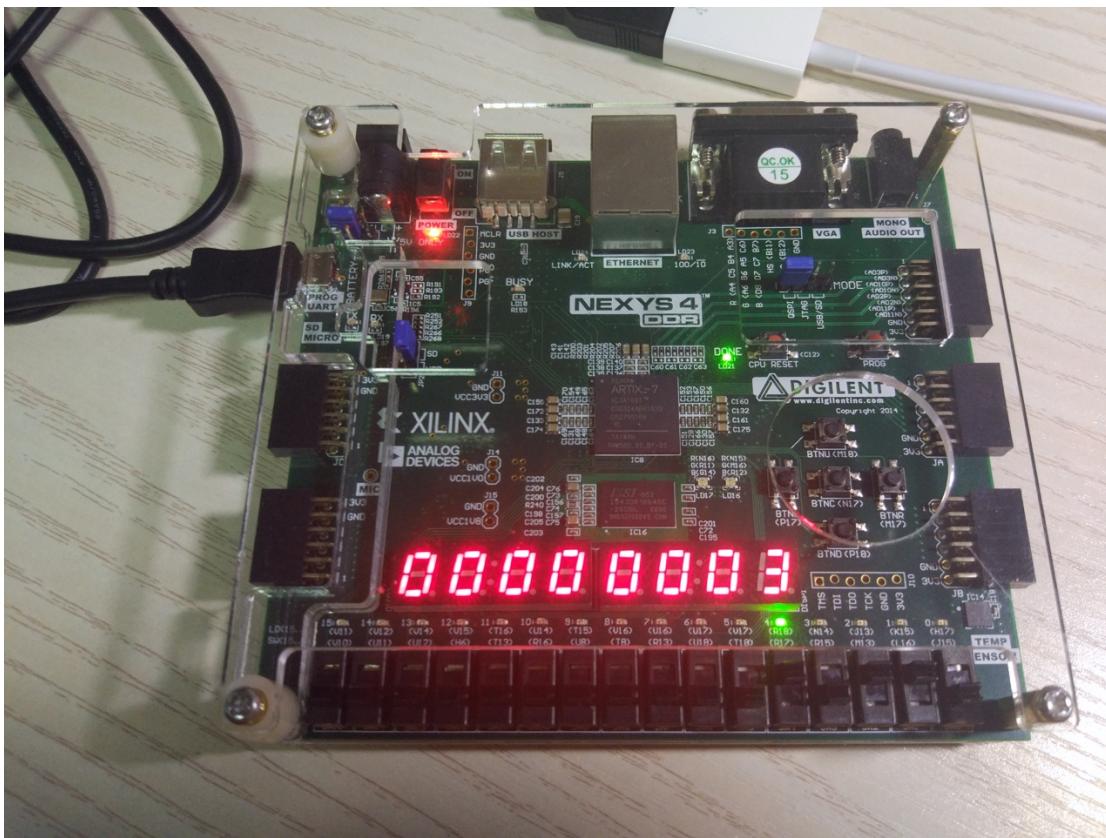
```

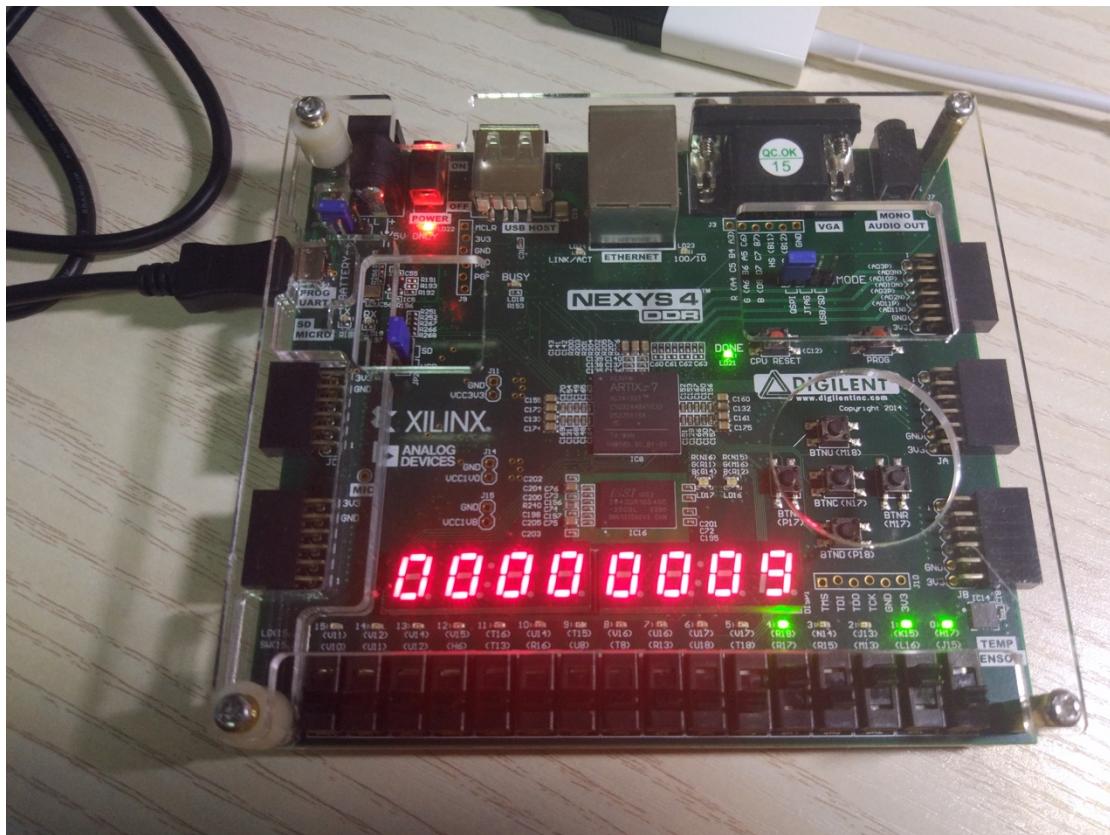
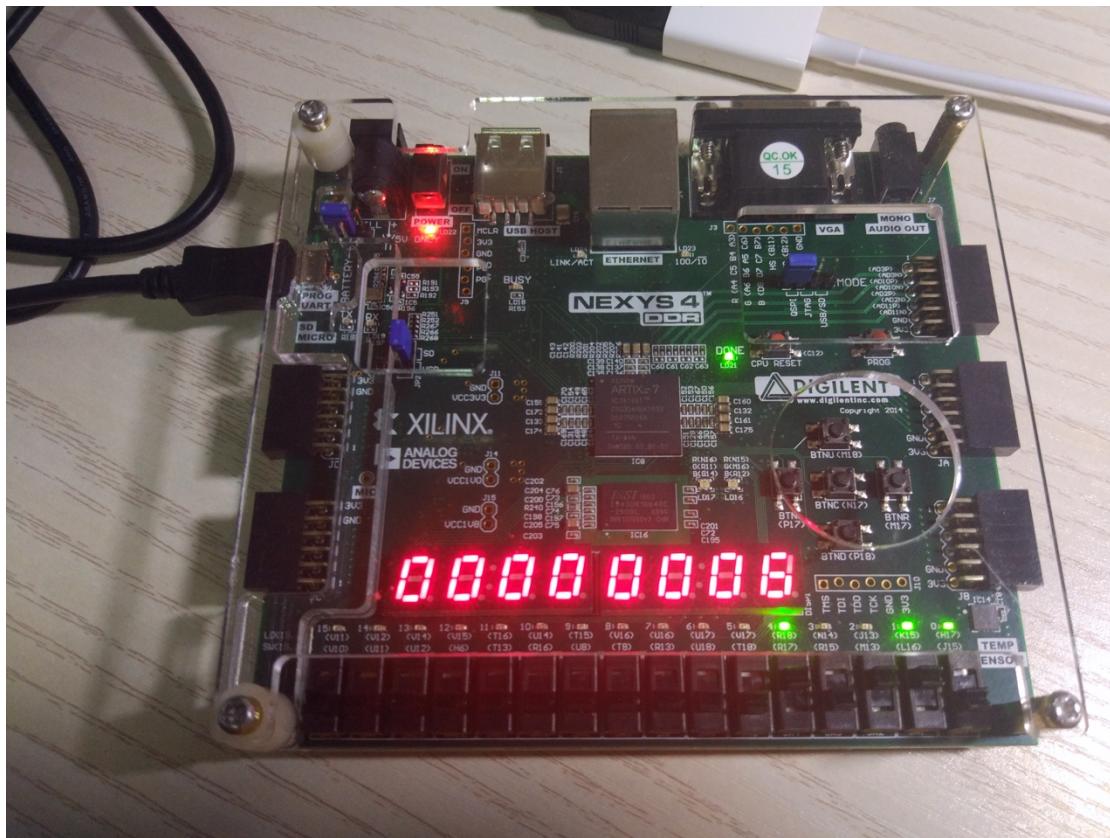
2) 指令对应的二进制数在存储器中的截图

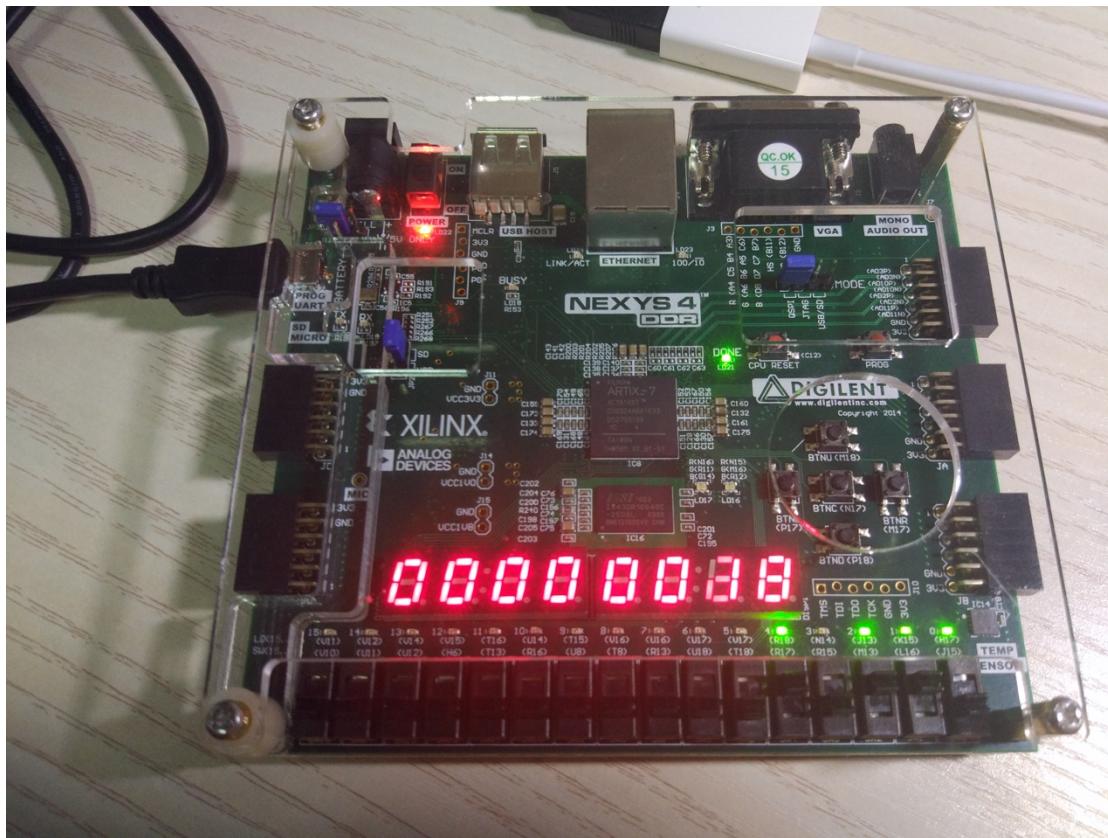
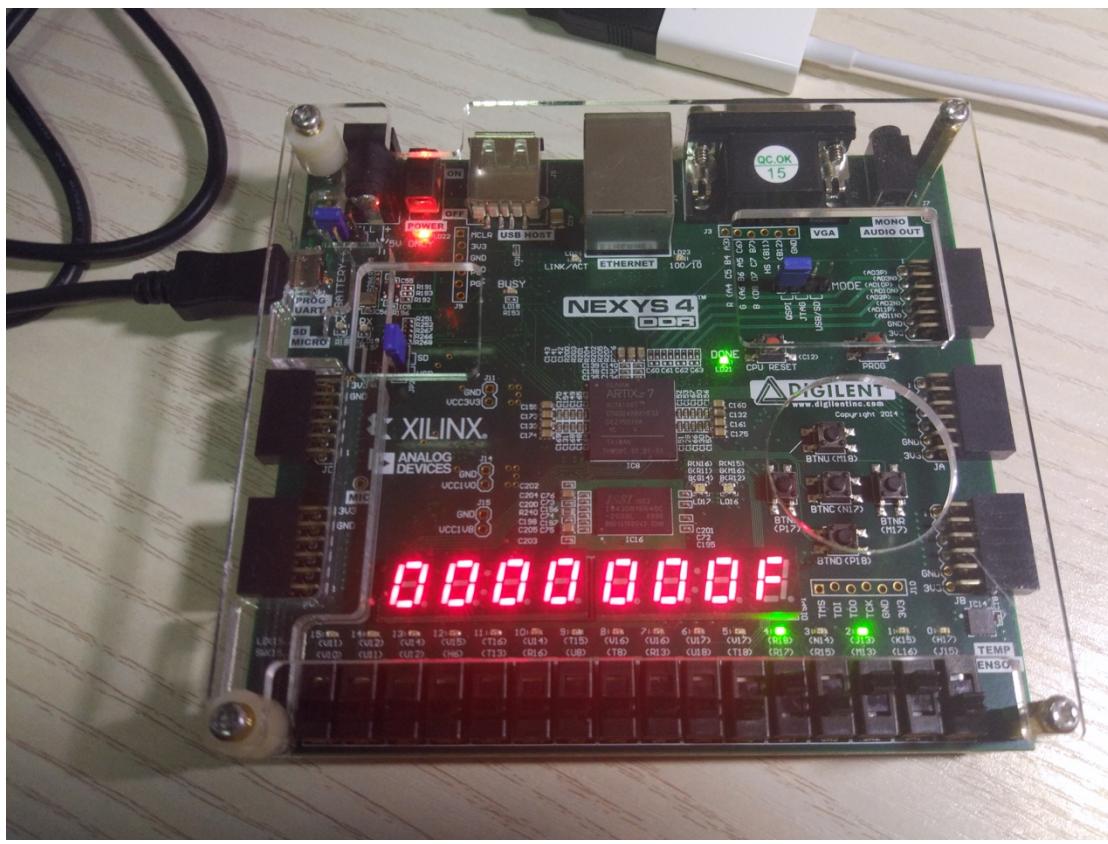
(2) 下面为运行结束时存储器内存储数据:

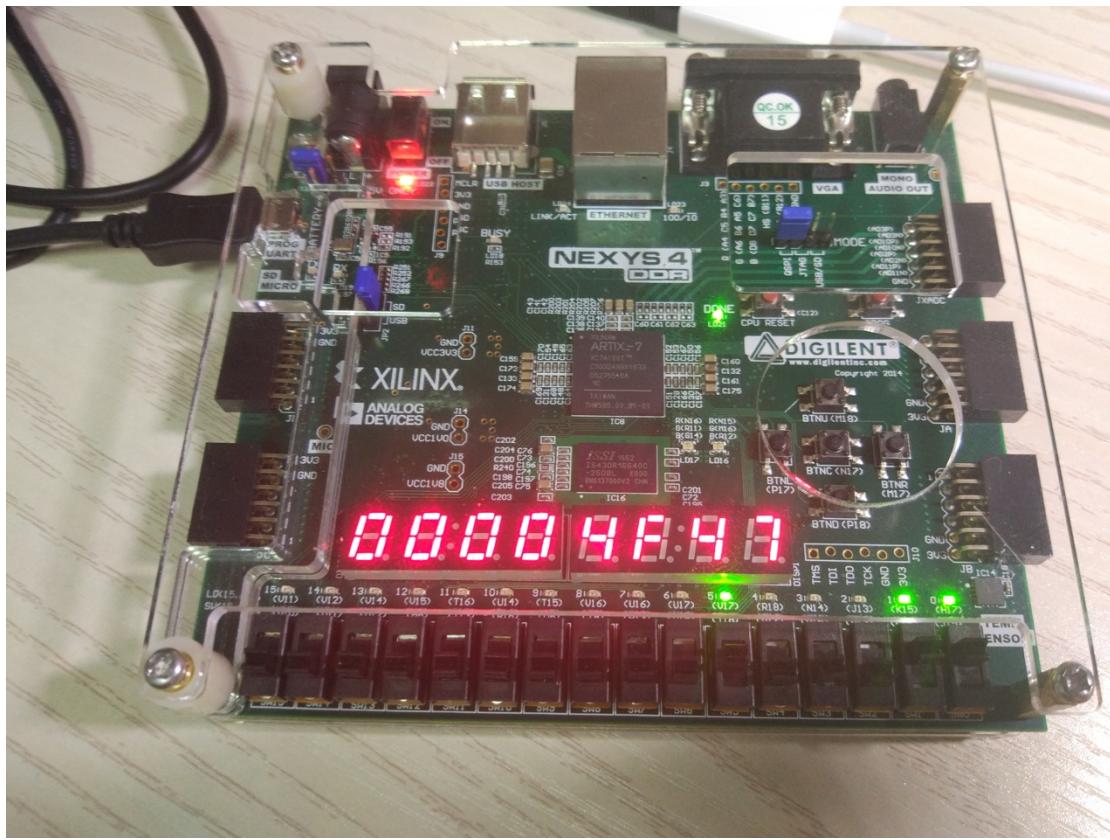
	0	1
100	3	3
102	6	9
104	15	24
106	39	63
108	102	165
110	267	432
112	699	1131
114	1830	2961
116	4791	7752
118	12543	20295

(3) 对该程序实现下载后得到的数据板照片：
其中分别选取了前几个地址的数据及最后一个地址的数据（16 进制）。









其中 led 灯表示当前地址，显示器表示当前地址存放的数据，可见斐波拉契数列可以正常运行在该操作板上。

2、运行助教代码得到的结果：

(1) 助教的汇编指令

1) 反汇编得到的 MARS 指令

```
.text
addi $8, $0, 8192
xor $9, $9, $9
xor $10, $10, $10
xor $11, $11, $11
xor $12, $12, $12
addi $9, $9, 1
addi $10, $10, 2
addi $11, $11, -1
lw $12, 0($8)
addi $8, $8, 4
add $13, $9, $11
sw $13, 0($8)
addi $8, $8, 4
add $13, $9, $10
sw $13, 0($8)
addi $8, $8, 4
```

```
sub $13, $9, $11
sw $13, 0($8)
addi $8, $8, 4
subu $13, $10, $9
sw $13, 0($8)
addi $8, $8, 4
and $13, $9, $11
sw $13, 0($8)
addi $8, $8, 4
andi $13, $11, 16
sw $13, 0($8)
addi $8, $8, 4
or $13, $9, $10
sw $13, 0($8)
addi $8, $8, 4
nor $13, $11, $9
sw $13, 0($8)
addi $8, $8, 4
xor $13, $11, $9
sw $13, 0($8)
addi $8, $8, 4
b: addi $13, $13, 1
bgtz $13, a
j b
a: sw $13, 0($8)
addi $8, $8, 4
bne $13, $9, c
xor $13, $13, $13
c: sw $13, 0($8)
addi $8, $8, 4
addi $14, $0, 200
xor $13, $13, $13
jr $14
addi $13, $13, 16
addi $13, $13, 8
sw $13, 0($8)
addi $8, $8, 4
addi $14, $0, 8192
lw $9, 0($14)
add $9, $10, $11
add $11, $9, $9
sw $11, 0($8)
```

2) 指令对应的二进制数在存储器中的截图

(2) 测试代码设计目的:

用于检测该流水线 CPU 能否很好地处理各类数据相关问题。

(3) 存储器内存储的数据结果:

11	0	0
9	-2	0
7	3	16
5	1	1
3	2	3
1	0	0

(4) 结论：结果正确，可以较好地解决数据相关问题。

3、自己的逻辑算术指令测试：

(1) 测试 MARS 指令:

, text

```
addi $t0, $zero, 1  
addi $t1, $zero, 2  
addi $t2, $zero, 3  
sw $t0, 20($zero)  
sw $t1, 24($zero)
```

```
sw $t2, 28($zero)
add $t3, $t0, $t1
sw $t3, 32($zero)
sub $t3, $t1, $t0
sw $t3, 36($zero)
and $t3, $t1, $t0
sw $t3, 40($zero)
or $t3, $t1, $t0
sw $t3, 44($zero)
nor $t3, $t1, $t0
sw $t3, 48($zero)
xor $t3, $t1, $t0
sw $t3, 52($zero)
subi $t3, $t2, 100
sw $t3, 56($zero)
ori $t3, $t2, 100
sw $t3, 60($zero)
xori $t3, $t2, 100
sw $t3, 64($zero)
lui $t3, 100
sw $t3, 68($zero)
sll $t4, $t0, 3
sw $t4, 76($zero)
sllv $t4, $t0, $t1
sw $t4, 80($zero)
addi $t5, $zero, 128
srl $t6, $t5, 5
sw $t6, 84($zero)
srlv $t6, $t5, $t2
sw $t6, 88($zero)
sra $t6, $t5, 5
sw $t6, 92($zero)
srav $t6, $t5, $t2
sw $t6, 96($zero)
slt $t7, $t0, $t1
sw $t7, 100($zero)
slti $t7, $t0, 0
sw $t7, 104($zero)
```

(2) 测试代码设计目的:

该测试代码主要用于检测各类逻辑运算指令是否可以正确执行。

(3) 运行后存储器内数据:

27	0	0	1	0
23	0	0	0	4
19	6	0	6553600	103
15	103	-97	3	-4
11	3	0	1	22
7	3	2	1	0

(4) 结论：对于前面较为简单的逻辑运算，运算结果均正确，特别的，对于逻辑右移和算术右移结果均存在问题，此处存疑。

4、自己的转跳链接类指令的测试结果

(1) MARS 指令：

```
.text
addi    $t0, $zero, 100
addi    $t5, $zero, 20
addi    $t3,$zero,3
addi    $t4,$zero,3
sw      $t3, 0($t0)
sw      $t4, 4($t0)
addi   $t1, $t5, -3
loop: lw     $t3, 0($t0)
      lw     $t4, 4($t0)
      add   $t2, $t3, $t4
      sw     $t2, 8($t0)
      addi  $t0, $t0, 4
      addi  $t1, $t1, -1
      bgezal $t1, loop
      jr    $ra
```

(2) 测试代码设计目的：

本段测试代码在原斐波拉契数列之上在循环末尾加入了 bgezal 的分支并链接以及 jr 的寄存器转跳指令，实现成功时斐波拉契数列将可以一直算下去，由于 loop 的位置被保存在\$ra 中，所以本段代码在 bgez 结束后仍会在死循环中，故会一直产生斐波拉契数列下去。

(3) 运行后存储器内数据

63	189737958	117264507	72473451	44791056
59	27682395	17108661	10573734	6534927
55	4038807	2496120	1542687	953433
51	589254	364179	225075	139104
47	85971	53133	32838	20295
43	12543	7752	4791	2961
39	1830	1131	699	432
35	267	165	102	63
31	39	24	15	9
27	6	3	3	0

(4) 结论：斐波拉契数列一直产生，故分支、跳转与链接指令设计正确。

五. 附录

1、实验源码:

(1) PC 模块

```
module PC(
    input clk,
    input reset,
    input PCWrite,
    input [31:0] newAddress,
    output reg[31:0] currentAddress
);

    always@(posedge clk or negedge reset)
    begin
        if (~reset)
            begin
                currentAddress <= 0;
            end
        else begin
            if (PCWrite)
                begin
                    currentAddress <= newAddress;
                end
            else begin
                currentAddress <= currentAddress;
            end
        end
    end
endmodule
```

(2) Control 模块

```
module Control(
    input clk,
    input [31:0] instruction,
    output reg ExtSel,
    output [22:0] Ctrsignal
);
reg
ALUSrc, RegDst, RegWrite, MemWrite, RegWritem, MemRead, MemWriteRegDst, Branch, jump, connect;
reg [1:0] PCbackSrc, storetype, MemtoReg;
reg [2:0] loadtype, AluzeroCtr;
reg [2:0] ALUop;
```

```

wire [2:0] WB;
wire [11:0] MEM;
wire [4:0] rs, rt;
wire [7:0] EX;
wire [5:0] op, funct;

assign op = instruction[31:26];
assign funct = instruction[5:0];
assign rt = instruction[20:16];
assign WB[2] = RegWrite;
assign WB[1:0] = MemtoReg;
assign MEM[11] = connect;
assign MEM[10:9] = storetype;
assign MEM[8:6] = loadtype;
assign MEM[5:4] = PCbackSrc;
assign MEM[3] = jump;
assign MEM[2] = Branch;
assign MEM[1] = MemRead;
assign MEM[0] = MemWrite;
assign EX[7:5] = AluzeroCtr;
assign EX[4:4] = RegDst;
assign EX[3:1] = ALUop;
assign EX[0:0] = ALUSrc;
assign Ctrsignal[22:15] = EX;
assign Ctrsignal[14:3] = MEM;
assign Ctrsignal[2:0] = WB;

parameter [5:0]
    R_type = 6'd0,
    addi = 6'd8,
    addiu = 6'd9,
    andi = 6'd12,
    ori = 6'd13,
    xori = 6'd14,
    lui = 6'd15,
    slti = 6'd10,
    sltui = 6'd11,
    lb = 6'd32,
    lh = 6'd33,
    lw1 = 6'd34,
    lw = 6'd35,
    lbu = 6'd36,
    hu = 6'd37,

```

```
lwr = 6' d38,
```

```
sb = 6' d40,  
sh = 6' d41,  
swl = 6' d42,  
sw = 6' d43,  
swr = 6' d46,
```

```
bltz = 6' d1,  
beq = 6' d4,  
bne = 6' d5,  
blez = 6' d6,  
bgtz = 6' d7,
```

```
j = 6' d2,  
jal = 6' d3,
```

```
halt = 6' d63;
```

```
always@(*)
```

```
begin
```

```
    RegDst = 0;  
    ALUSrc = 0;  
    ExtSel = 0;  
    RegWrite = 0;  
    MemRead = 0;  
    MemWrite = 0;  
    MemtoReg = 2' b00;  
    AluzeroCtr = 3' b000;  
    jump = 0;  
    Branch = 0;  
    ALUop = 3' b111;  
    PCbackSrc = 2' b00;  
    loadtype = 3' b100;  
    storetype = 2' b10;  
    connect = 0;
```

```
case (op)
```

```
    R_type:
```

```
        begin
```

```
            RegDst = 1;
```

```
            case (funct)
```

```
6' d8:begin//jr
Branch = 1;
ALUop = 3'b000;
AluzeroCtr = 3'b011;
PCbackSrc = 2'b10;
end
```

```
6' d9:begin//jalr
Branch = 1;
ALUop = 3'b000;
AluzeroCtr = 3'b011;
PCbackSrc = 2'b10;
connect = 1;
MemtoReg = 2'b10;
end
```

```
default:begin
ALUop = 3'b010;
RegWrite = 1;
end
endcase
end
```

```
1b:
begin
ALUSrc = 1;
ExtSel = 1;
MemRead = 1;
RegWrite = 1;
MemtoReg = 2'b01;
ALUop = 3'b000;
loadtype = 3'b000;
end
```

```
1bu:
begin
ALUSrc = 1;
ExtSel = 1;
MemRead = 1;
RegWrite = 1;
MemtoReg = 2'b01;
ALUop = 3'b000;
loadtype = 3'b001;
```

```
end

lh:
begin
ALUSrc = 1;
ExtSel = 1;
MemRead = 1;
RegWrite = 1;
MemtoReg = 2' b01;
ALUop = 3' b000;
loadtype = 3' b010;
end

lhu:
begin
ALUSrc = 1;
ExtSel = 1;
MemRead = 1;
RegWrite = 1;
MemtoReg = 2' b01;
ALUop = 3' b000;
loadtype = 3' b011;
end

lw:
begin
ALUSrc = 1;
ExtSel = 1;
MemRead = 1;
RegWrite = 1;
MemtoReg = 2' b01;
ALUop = 3' b000;
end

sw:
begin
ALUSrc = 1;
ExtSel = 1;
MemWrite = 1;
ALUop = 3' b000;
end

sb:
begin
```

```
ALUSrc = 1;
ExtSel = 1;
MemWrite = 1;
ALUop = 3'b000;
storetype = 2'b00;
end
```

```
sh:
begin
ALUSrc = 1;
ExtSel = 1;
MemWrite = 1;
ALUop = 3'b000;
storetype = 2'b01;
end
```

```
addi:
begin
ALUSrc = 1;
ExtSel = 1;
RegWrite = 1;
ALUop = 3'b000;
end
```

```
addiu:
begin
ALUSrc = 1;
ExtSel = 1;
RegWrite = 1;
ALUop = 3'b000;
end
```

```
andi:
begin
ALUSrc = 1;
ExtSel = 1;
RegWrite = 1;
ALUop = 3'b011;
end
```

```
ori:
begin
ALUSrc = 1;
ExtSel = 1;
```

```
RegWrite = 1;
ALUop = 3'b100;
end

xori:
begin
ALUSrc = 1;
ExtSel = 1;
RegWrite = 1;
ALUop = 3'b101;
end

lui:
begin
ALUSrc = 1;
RegWrite = 1;
ALUop = 3'b110;
end

slti:
begin
ALUSrc = 1;
ExtSel = 1;
RegWrite = 1;
ALUop = 3'b111;
end

sltui:
begin
ALUSrc = 1;
RegWrite = 1;
ALUop = 3'b111;
end

bltz:
begin
ExtSel = 1;
ALUop = 3'b001;
Branch = 1;
case(rt)
5'b00000:AluzeroCtr = 3'b100;//bltz
5'b00001:AluzeroCtr = 3'b110;//bgez
5'b10000:begin//bltzal
AluzeroCtr = 3'b111;
```

```
connect = 1;
MemtoReg = 2' b10;
end
5' b10001:begin//bgezal
AluzeroCtr = 3' b110;
connect = 1;
MemtoReg = 2' b10;
end
default:;
endcase
```

```
end
```

```
bgtz:
begin
ExtSel = 1;
ALUop = 3' b001;
Branch = 1;
AluzeroCtr = 3' b010;
end
```

```
beq:
begin
ExtSel = 1;
ALUop = 3' b001;
Branch = 1;
end
```

```
bne:
begin
ExtSel = 1;
ALUop = 3' b001;
Branch = 1;
AluzeroCtr = 3' b001;
end
```

```
blez:
begin
ExtSel = 1;
ALUop = 3' b001;
Branch = 1;
AluzeroCtr = 3' b101;
end
```

```

j:
begin
jump = 1;
PCbackSrc = 2' b01;
end

jal:
begin
jump = 1;
PCbackSrc = 2' b01;
MemtoReg = 2' b10;
connect = 1;
end

default:;
endcase
end

endmodule
(3) ALU 信号控制模块
module ALUcontrol(
    input [2:0] ALUop,
    input [5:0] funct,
    output reg [3:0] ALUControl,
    output reg ALUSrc_shamt
);
always@(*)
begin
ALUSrc_shamt = 0;
case(ALUop)
3'b010:
begin
case(funct)
6'b100000: ALUControl = 4'b0010;//add
6'b100001: ALUControl = 4'b0010;//addu
6'b100010: ALUControl = 4'b0110;//sub
6'b100011: ALUControl = 4'b0110;//subu
6'b100100: ALUControl = 4'b0000;//and
6'b100101: ALUControl = 4'b0001;//or
6'b100111: ALUControl = 4'b1100;//nor
6'b100110: ALUControl = 4'b0111;//xor
6'b000000: begin//sll
ALUSrc_shamt = 1;

```

```

ALUControl = 4'b1000;
end
6'b000100: ALUControl = 4'b1000;//sllv
6'b000010: begin//srl
ALUSrc_shamt = 1;
ALUControl = 4'b1001;
end
6'b000110: ALUControl = 4'b1001;//sr1v
6'b000011: begin//sra
ALUSrc_shamt = 1;
ALUControl = 4'b1010;
end
6'b000111: ALUControl = 4'b1010;//srav
6'b101010: ALUControl = 4'b0011;//slt
6'b101101: ALUControl = 4'b0011;//sltu
6'b001000: ALUControl = 4'b0010;//jr

default: ALUControl = 4'b1111;
endcase
end

3'b000: ALUControl = 4'b0010;//add
3'b001: ALUControl = 4'b0110;//sub
3'b011: ALUControl = 4'b0000;//and
3'b100: ALUControl = 4'b0001;//or
3'b101: ALUControl = 4'b0111;//xor
3'b110: ALUControl = 4'b1011;//lui
3'b111: ALUControl = 4'b0011;//slt
default:;
endcase
end
endmodule
(4) ALU 模块
module ALU(
    input signed [31:0] alu_a,
    input signed [31:0] alu_b,
    input [3:0] alu_op,
    input [2:0] AluzeroCtr,
    output reg [31:0] alu_out,
    output reg zero
);
parameter A_ADD = 4'b0010;
parameter A_SUB = 4'b0110;
parameter A_AND = 4'b0000;

```

```

parameter A_OR = 4'b0001;
parameter A_XOR = 4'b0111;
parameter A_NOR = 4'b1100;
parameter A_SLL = 4'b1000;
parameter A_SRL = 4'b1001;
parameter A_SRA = 4'b1010;
parameter A_LUI = 4'b1011;
parameter A_SLT = 4'b0011;

always @(*)
begin
    case (alu_op)
        A_ADD : alu_out = alu_a + alu_b;
        A_SUB : alu_out = alu_a - alu_b;
        A_AND : alu_out = alu_a & alu_b;
        A_OR : alu_out = alu_a | alu_b;
        A_XOR : alu_out = alu_a ^ alu_b;
        A_NOR: alu_out = ~(alu_a | alu_b);
        A_SLL: alu_out = alu_a << alu_b;
        A_SRL: alu_out = alu_a >> alu_b;
        A_SRA: alu_out = ($signed(alu_a)) >>> alu_b;
        A_LUI: begin
            alu_out[15:0] = alu_a[15:0];
            alu_out[31:16] = alu_b[15:0];
        end
        A_SLT: alu_out = ((alu_a < alu_b)? 1 : 0);

        default: alu_out = 32'h0;
    endcase
end

always@(*)
begin
    case(AluzeroCtr)
        3'b000://equal to label
        begin
            if(alu_out == 0) zero = 1;
            else zero = 0;
        end

        3'b001://inequal to label
        begin
            if(alu_out != 0) zero = 1;
            else zero = 0;
        end
    endcase
end

```

```

end

3'b010://greater to label
begin
if(alu_out > 0) zero = 1;
else zero = 0;
end

3'b011://greater or equal to label
begin
if(alu_out >= 0) zero = 1;
else zero = 0;
end

3'b100://smaller to label
begin
if(alu_out < 0) zero = 1;
else zero = 0;
end

3'b101://smaller or equal to label
begin
if(alu_out <= 0) zero = 1;
else zero = 0;
end

3'b110://a >= to label
begin
if(alu_a >= 0) zero = 1;
else zero = 0;
end

3'b111://a < to label
begin
if(alu_a < 0) zero = 1;
else zero = 0;
end
default:;
endcase
end

endmodule
(5) 寄存器模块
module REG_FILE(

```

```

    input clk,
    input reset,
    input [4:0] read_addr1,
    input [4:0] read_addr2,
    output reg [31:0] RD1,
    output reg [31:0] RD2,
    input [4:0] write_addr,
    input [31:0] WD,
    input wEna
);

reg [31:0] register [31:0];
integer i;
//assign RD1 = (read_addr1 == 0)? 0 : register[read_addr1];
//assign RD2 = (read_addr2 == 0)? 0 : register[read_addr2];

always@(posedge clk or negedge reset)
begin
if (~reset)
begin
for(i = 0;i < 32;i = i + 1)
register[i] = 0;
end
end

always@(*)
begin
if (read_addr1 == 0) RD1 <= 0;
else begin
if (read_addr1 == write_addr)
RD1 <= WD;
else RD1 <= register[read_addr1];
end
if (read_addr2 == 0) RD2 <= 0;
else begin
if (read_addr2 == write_addr)
RD2 <= WD;
else RD2 <= register[read_addr2];
end
end

always @(posedge clk)
begin
if (wEna)

```

```

begin
    if (write_addr != 0)
        register[write_addr] <= WD;
    end
end
endmodule

```

(6) 数据选择器模块

1) 2 选 1 模块:

```

module Mux(control, in1, in0, out);
    parameter EM = 31;
    input control;
    input [EM:0] in1;
    input [EM:0] in0;
    output [EM:0] out;
    assign out = control ? in1 : in0;
endmodule

```

2) 3 选 1 模块:

```

module Mux3to1(control, in0, in1, in2, out);
    parameter EM = 31;
    input [1:0] control;
    input [EM:0] in0;
    input [EM:0] in1;
    input [EM:0] in2;
    output reg [EM:0] out;
    always@(*)
begin
    case(control)
        2'b00: out = in0;
        2'b01: out = in1;
        2'b10: out = in2;
        default: out = 32'hxxxxxxxx;
    endcase
end

```

(7) 信号扩展模块

```

module SignExtend(
    input ExtSel,
    input [15:0] immediate,
    output [31:0] extendImmediate
);

```

```

assign extendImmediate[15:0] = immediate;
assign extendImmediate[31:16] = ExtSel ? (immediate[15] ?
16'hffff : 16'h0000) : 16'h0000;

```

```
endmodule
```

(8) 子数据寄存器模块:

```
module Reg(clk, reset, in, out);  
    parameter EM = 31;  
    input clk;  
    input reset;  
    input [EM:0] in;  
    output reg [EM:0] out;  
    always@(posedge clk or negedge reset)  
    begin  
        if (~reset)  
            begin  
                out <= 0;  
            end  
        else begin  
            out <= in;  
        end  
    end
```

```
endmodule
```

(9) 带使能控制的子数据寄存器模块:

```
module Reg_we(clk, reset, IRWrite, in, out);  
    parameter EM = 31;  
    input clk;  
    input reset;  
    input IRWrite;  
    input [EM:0] in;  
    output reg [EM:0] out;  
    always@(posedge clk or negedge reset)  
    begin  
        if (~reset)  
            begin  
                out <= 0;  
            end  
        else begin  
            if(IRWrite)  
                begin  
                    out <= in;  
                end  
            else begin  
                out <= out;  
            end  
        end  
    end
```

```

    end

endmodule
(10) forwarding 模块
module Forwardunit(
    input [4:0] rs,
    input [4:0] rt,
    input [4:0] rd_EX,
    input [4:0] rd_MEM,
    input RegWrite_EX,
    input RegWrite_MEM,
    output reg [1:0] ALUScrA,
    output reg [1:0] ALUScrB
);
always@(*)
begin

ALUScrA = 2' b00;
ALUScrB = 2' b00;
if ((rs == rd_EX)&&(RegWrite_EX))
begin
ALUScrA = 2' b01;
end
else if ((rs == rd_MEM)&&(RegWrite_MEM))
begin
ALUScrA = 2' b10;
end
if ((rt == rd_EX)&&(RegWrite_EX))
begin
ALUScrB = 2' b01;
end
else if ((rt == rd_MEM)&&(RegWrite_MEM))
begin
ALUScrB = 2' b10;
end

end

endmodule
(11) interlock 模块
module Hazardunit(
    input reset,
    input [4:0] rt_EX,
    input [4:0] rs_ID,

```

```

    input [4:0] rt_ID,
    input MemRead,
    output reg PCWrite,
    output reg IRWrite,
    output reg CtrStr
  );
always@(*)
begin
if (reset);
PCWrite = 1;
IRWrite = 1;
CtrStr = 1;
if((MemRead)&&(rt_EX!=5'b0)&&((rt_EX==rs_ID) || (rt_EX==rt_ID)))
begin
PCWrite = 0;
IRWrite = 0;
CtrStr = 0;
end
end
endmodule

```

(12) 各个段寄存器模块

1) IF/ID

```

module IF_ID(
  input clk,
  input reset,
  input IRWrite,
  input [31:0] nextAddress_in,
  output [31:0] nextAddress_out,
  input [31:0] Ins_in,
  output [31:0] Ins_out
);
Reg_we PCreg(clk, reset, IRWrite, nextAddress_in, nextAddress_out),
  Insreg(clk, reset, IRWrite, Ins_in, Ins_out);
endmodule

```

2) ID/EX

```

module ID_EX(
  input clk,
  input reset,
  input [2:0] WB_in,
  output [2:0] WB_out,
  input [11:0] MEM_in,
  output [11:0] MEM_out,

```

```

    input [7:0] EX_in,
    input [31:0] nextAddress_in,
    output [31:0] nextAddress_out,
    input [31:0] A_in,
    output [31:0] A_out,
    input [31:0] B_in,
    output [31:0] B_out,
    input [31:0] imm_in,
    output [31:0] imm_out,
    input [4:0] Ins25_in,
    output [4:0] Ins25_out,
    input [4:0] Ins20_in,
    output [4:0] Ins20_out,
    input [4:0] Ins15_in,
    output [4:0] Ins15_out,
    input [4:0] Ins10_in,
    output [4:0] Ins10_out,
    input [25:0] tar_in,
    output [25:0] tar_out,
    output [2:0] AluzeroCtr,
    output RegDst,
    output [2:0] ALUop,
    output ALUSrc
);
wire [7:0] EX_out;
assign AluzeroCtr = EX_out[7:5];
assign RegDst = EX_out[4:4];
assign ALUop = EX_out[3:1];
assign ALUSrc = EX_out[0:0];
Reg #(2) WBreg(clk, reset, WB_in, WB_out);
Reg #(11) MEMreg(clk, reset, MEM_in, MEM_out);
Reg #(7) EXreg(clk, reset, EX_in, EX_out);
Reg PCreg2(clk, reset, nextAddress_in, nextAddress_out),
    Areg(clk, reset, A_in, A_out),
    Breg(clk, reset, B_in, B_out),
    Immreg(clk, reset, imm_in, imm_out);
Reg #(4) Insaddrreg0(clk, reset, Ins25_in, Ins25_out),
    Insaddrreg1(clk, reset, Ins20_in, Ins20_out),
    Insaddrreg2(clk, reset, Ins15_in, Ins15_out),
    Insshamt(clk, reset, Ins10_in, Ins10_out);
Reg #(25) tarreg(clk, reset, tar_in, tar_out);
endmodule
3) EX/MEM
module EX_MEM(

```

```

input clk,
input reset,
input [2:0] WB_in,
output [2:0] WB_out,
input [11:0] MEM_in,
input [31:0] nextAddress_in,
output [31:0] nextAddress_out,
input zero_in,
output zero_out,
input [31:0] ALUresult_in,
output [31:0] ALUresult_out,
input [31:0] B_in,
output [31:0] B_out,
input [4:0] Insaddr_in,
output [4:0] Insaddr_out,
input [31:0] jaddr_in,
output [31:0] jaddr_out,
output jump,
output Branch,
output MemRead,
output MemWrite,
output [1:0] PCbackSrc,
output [2:0] loadtype,
output [1:0] storetype,
output connect
);
wire [11:0] MEM_out;
assign connect = MEM_out[11];
assign storetype= MEM_out[10:9];
assign loadtype = MEM_out[8:6];
assign PCbackSrc = MEM_out[5:4];
assign jump = MEM_out[3:3];
assign Branch = MEM_out[2:2];
assign MemRead = MEM_out[1:1];
assign MemWrite = MEM_out[0:0];
Reg #(0) zeroreg(clk, reset, zero_in, zero_out);
Reg #(2) WBreg(clk, reset, WB_in, WB_out);
Reg #(11) MEMreg(clk, reset, MEM_in, MEM_out);
Reg newPCreg(clk, reset, nextAddress_in, nextAddress_out),
    ALUreg(clk, reset, ALUresult_in, ALUresult_out),
    Breg(clk, reset, B_in, B_out),
    jumpreg(clk, reset, jaddr_in, jaddr_out);
Reg #(4) Insreg(clk, reset, Insaddr_in, Insaddr_out);
endmodule

```

4) MEM/WB

```

module MEM_WB(
    input clk,
    input reset,
    input connect_zero,
    output connect_zero_WB,
    input [31:0] nextpc,
    output [31:0] nextpc_WB,
    input [2:0] WB_in,
    input [31:0] ALUresult_in,
    output [31:0] ALUresult_out,
    input [31:0] Data_out,
    output [31:0] Data_out_WB,
    input [4:0] Insaddr_in,
    output [4:0] Insaddr_out,
    output RegWrite,
    output [1:0] MemtoReg
);
wire [2:0] WB_out;
assign RegWrite = WB_out[2];
assign MemtoReg = WB_out[1:0];
Reg #(0) connectreg(clk, reset, connect_zero, connect_zero_WB);
Reg pcreg(clk, reset, nextpc, nextpc_WB);
Reg #(2) WBreg(clk, reset, WB_in, WB_out);
Reg RAMreg(clk, reset, Data_out, Data_out_WB);
Reg ALUreg(clk, reset, ALUresult_in, ALUresult_out);
Reg #(4) Insaddrreg(clk, reset, Insaddr_in, Insaddr_out);
endmodule

```

(13) load 控制模块

```

module LoadCtr(
    input [31:0] original_data,
    input [2:0] loadtype,
    input [1:0] addr_low,
    output reg [31:0] final_data
);

parameter LOAD_LB = 3'd0;
parameter LOAD_LBU = 3'd1;
parameter LOAD_LH = 3'd2;
parameter LOAD_LHU = 3'd3;
parameter LOAD_LW = 3'd4;

always @(*)

```

```

begin
  case(loadtype)
    LOAD_LB:
    begin
      if (addr_low == 2'b00)
        final_data = $signed(original_data[7:0]);
      else if (addr_low == 2'b01)
        final_data = $signed(original_data[15:8]);
      else if (addr_low == 2'b10)
        final_data = $signed(original_data[23:16]);
      else
        final_data = $signed(original_data[31:24]);
    end
    LOAD_LBU:
    begin
      if (addr_low == 2'b00)
        final_data = original_data[7:0];
      else if (addr_low == 2'b01)
        final_data = original_data[15:8];
      else if (addr_low == 2'b10)
        final_data = original_data[23:16];
      else
        final_data = original_data[31:24];
    end
    LOAD_LH:
    begin
      if (addr_low == 2'b00)
        final_data = $signed(original_data[15:0]);
      else // addr_low == 2'b10
        final_data = $signed(original_data[31:16]);
    end
    LOAD_LHU:
    begin
      if (addr_low == 2'b00)
        final_data = original_data[15:0];
      else
        final_data = original_data[31:15];
    end
    LOAD_LW:
      final_data = original_data;
    default: final_data = 32'b0;
  endcase
end
endmodule

```

(14) store 控制模块

```
module StoreCtr(
    input [31:0] original_data,
    input [31:0] ram_read,
    input [1:0] storetype,
    input [1:0] addr_low,
    output reg [31:0] final_data
);

parameter STORE_SB = 2'd0;
parameter STORE_SH = 2'd1;
parameter STORE_SW = 2'd2;

always @(*)
begin
    case(storetype)
        STORE_SB:
        begin
            if (addr_low == 2'b00)
                final_data = {ram_read[31:8],
original_data[7:0]};
            else if (addr_low == 2'b01)
                final_data = {ram_read[31:16],
original_data[7:0], ram_read[7:0]};
            else if (addr_low == 2'b10)
                final_data = {ram_read[31:24],
original_data[7:0], ram_read[15:0]};
            else // low_two_bits = 2'b11
                final_data = {original_data[7:0],
ram_read[23:0]};
        end
        STORE_SH:
        begin
            if (addr_low == 2'b00)
                final_data = {ram_read[31:16],
original_data[15:0]};
            else final_data = {original_data[15:0],
ram_read[15:0]};
        end
        STORE_SW:
        begin
            final_data = original_data;
        end
        default:
            final_data = original_data;
    endcase
end
```

```

    end
endmodule
(15) PipelineCPU 模块
module PipelineCPU(
    input clk,
    input reset,
    input [7:0] read_addr_out,
    output [31:0] read_out
);

wire [1:0] PCbackSrc, storetype, MemtoReg, ALUScrA, ALUScrB;
wire [2:0] WB, WB_EXM, WB_MWB, ALUop, loadtype, AluzeroCtr;
wire [3:0] ALUControl;
wire [4:0]
regw_addr, rs, rt, rd, regw_addr_EXM, regw_addr_MWB, regw_addr_WB, shamt;
    wire [7:0] EX, data_address;
    wire [11:0] MEM, MEM_EXM;
    wire [22:0] Ctrsignal, consignal;
    wire [25:0] tar_out;
    wire [31:0]

A, B, B_in, shamt_in, ScrA_s, newAddress, newAddress_4, RD1, RD2, WD, Mem_out, c
urrentAddress, ALU_out, memaddr, Ins_out, ScrA, ScrB, ScrB_s, ALUOut, Data_in
, Data_out, ALUresult, ALUback, final_data_out, Data_out_WB, final_data_in;
    wire [31:0]
extendImmediate, extendImmediate_EXM, Address, JumpAddress, JumpAddress_M
, nextAddress, nextAddress_IDEX, nextAddress_EXM, nextaddress_imm, nextadd
ress_imm_M, NEXTADDR, NEXTADDR_WB, Instrution;

    wire
Zero, Zero_out, RegWrite_ori, PCWrite, IRWrite, Branch, PCSrc, ALUSrc, RegWri
te, MemWrite, ExtSel, RegDst, p_reset, p_reset2, CtrStr, ALUSrc_shamt, connec
t, connect_zero, connect_zero_WB;

    assign nextAddress=(currentAddress << 2) + 4;

    assign PCSrc=(Zero_out&Branch) | jump;

    assign nextaddress_imm=nextAddress_EXM+(extendImmediate_EXM <<
2);
    assign newAddress=newAddress_4 >> 2;

    assign JumpAddress[27:0] = tar_out << 2;
    assign JumpAddress[31:28] = nextAddress_EXM[31:28];

```

```

assign p_reset=(PCSrc==1)? 0 : 1 ;
assign p_reset2= 1;

assign EX = consignal[22:15];
assign MEM = consignal[14:3];
assign WB = consignal[2:0];

assign data_address=ALUresult[7:0] >> 2;
assign connect_zero=connect&(Zero_out | jump);

assign shamt_in[31:5] = 27'd0;
assign shamt_in[4:0] = shamt;

PC pc(clk, reset, PCWrite, newAddress, currentAddress) ;
Mux Muxnextaddress (PCSrc, NEXTADDR, nextAddress, newAddress_4) ;
Memory InsMemory (currentAddress[6:0], Ins_out) ;
//cacuROM InsMemory (currentAddress[6:0], Ins_out) ;
//bneROM InsMemory (currentAddress[6:0], Ins_out) ;

IF_ID
IFIDreg(clk, p_reset, IRWrite, nextAddress, nextAddress_IDEX, Ins_out, Instruction) ;
Control CPUctr(clk, Instrution, ExtSel, Ctrsinal) ;
Mux #(22)BubbleMux(CtrStr, Ctrsinal, 23'b0, consignal) ;
REG_FILE
registerfile(clk, reset, Instrution[25:21], Instrution[20:16], RD1, RD2, re
gw_addr, WD, RegWrite) ;
SignExtend signex(ExtSel, Instrution[15:0], extendImmediate) ;

ID_EX
IDEXreg(clk, p_reset, WB, WB_EXM, MEM, MEM_EXM, EX, nextAddress_IDEX, nextAdd
ress_EXM, RD1, A, RD2, B, extendImmediate, extendImmediate_EXM, Instrution[2
5:21], rs, Instrution[20:16], rt, Instrution[15:11], rd, Instrution[10:6], s
hamt, Instrution[25:0], tar_out, AluzeroCtr, RegDst, ALUop, ALUSrc) ;
ALUcontrol
ALUctr(ALUop, extendImmediate_EXM[5:0], ALUControl, ALUSrc_shamt) ;
ALU alu(ScrA, ScrB, ALUControl, AluzeroCtr, ALU_out, Zero) ;
Mux Muxshamt (ALUSrc_shamt, shamt_in, ScrA_s, ScrA) ;
Mux MuxALUsrc (ALUSrc, extendImmediate_EXM, B_in, ScrB) ;
Mux #(4)Muxregw(RegDst, rd, rt, regw_addr_EXM) ;
Mux3to1 MuxALUsrcA(ALUScrA, A, ALUresult, WD, ScrA_s) ;
Mux3to1 MuxALUsrcB(ALUScrB, B, ALUresult, WD, B_in) ;

EX_MEMORY EXMreg(clk, p_reset2, WB_EXM, WB_MWB, MEM_EXM,

```

```

nextaddress_imm, nextaddress_imm_M, Zero, Zero_out, ALU_out, ALUresult, B_in,
Data_in, regw_addr_EXM, regw_addr_MWB, JumpAddress, JumpAddress_M, jump,
Branch, MemRead, MemWrite, PCbackSrc, loadtype, storetype, connect) ;

    Mux3to1 Muxaddress (PCbackSrc, nextaddress_imm_M,
JumpAddress_M, ALUresult, NEXTADDR) ;
        RAMemory Datamemory (data_address, final_data_in, read_addr_out,
clk, MemWrite, MemRead, clk, Data_out, read_out, qspo, qdpo) ;
        Mux #(4) Muxregaddr (connect_zero, 5'd31, regw_addr_MWB,
regw_addr_WB) ;
        LoadCtr loadctr (Data_out, loadtype, data_address[1:0],
final_data_out) ;
        StoreCtr storectr (Data_in, Data_out, storetype, data_address[1:0],
final_data_in) ;

        MEM_WB MWBreg (clk, p_reset2, connect_zero, connect_zero_WB,
NEXTADDR, NEXTADDR_WB, WB_MWB, ALUresult, ALUback, final_data_out, Data_out_WB,
regw_addr_WB, regw_addr, RegWrite_ori, MemtoReg) ;
        Mux3to1 Muxmemtoreg (MemtoReg, ALUback, Data_out_WB,
NEXTADDR_WB, WD) ;
        Mux #(0) Muxconnect (connect_zero_WB, 1'b1, RegWrite_ori, RegWrite) ;

        Hazardunit hunit (reset, rt, Instrution[25:21], Instrution[20:16],
MEM_EXM[1:1], PCWrite, IRWrite, CtrStr) ;
        Forwardunit forwardunt (rs, rt, regw_addr_MWB, regw_addr, WB_MWB[2],
RegWrite, ALUScrA, ALUScrB) ;
endmodule

```

(16) 实现下载所需的模块

1) top 模块

```

module top(
    input clk,
    input rst,
    input mode_change,
    input [7:0] addr_in,
    output [7:0] sel,
    output [7:0] seg,
    output [7:0] led
);

    wire mode_change_stable;

    wire clk_slow_seg;
    wire clk_slow_change_addr;

```

```

wire [7:0] read_addr;
wire [7:0] read_addr_seq;
wire [7:0] read_addr_select;
wire [31:0] mem_read_result;
wire [31:0] data_display;

reg curr_mode;

initial
    curr_mode <= 1'b0;

PipelineCPU CPU(clk, ~rst, read_addr, mem_read_result);
clk_divide clk_slow_seg_gen(clk, ~rst, clk_slow_seg);
clk_divide_addr clk_slow_change_addr_gen(clk, ~rst,
clk_slow_change_addr);
button_jitter mode_change_jitter(clk, ~rst, mode_change,
mode_change_stable);
read_addr_gen read_addr_generator_mode_seq(clk_slow_change_addr,
~rst, read_addr_seq);
seg data_displayer(clk_slow_seg, ~rst, data_display, sel, seg);

always @(posedge mode_change_stable or posedge rst)
begin
    if (rst)
        curr_mode<= 1'b0;
    else
        curr_mode <= ~curr_mode;
end

assign read_addr = (curr_mode) ? addr_in : read_addr_seq;

assign data_display = mem_read_result[31:0];
assign led = read_addr[7:0];

endmodule

```

2) 按键防抖动模块

```

module button_jitter(
    input clk,
    input rst_n,
    input button_in,
    output reg button_final
);

```

```

reg [31:0] cnt;
reg cnt_en;
reg button_out;
parameter interval=1_000_000;

always @(posedge clk or negedge rst_n)
begin
    if (~rst_n) button_out<=1'b0;
    else if (cnt==interval-1) button_out<=button_in;
end

always @(posedge clk or negedge rst_n)
begin
    if (~rst_n) button_final<=1'b0;
    else if (cnt==interval-1 && button_in) button_final<=1'b1;
    else if (button_final) button_final<=1'b0;
end

always @(posedge clk or negedge rst_n)
begin
    if (~rst_n) cnt_en<=1'b0;
    else if (button_in != button_out) cnt_en<=1'b1;
    else cnt_en<=1'b0;
end

always @(posedge clk or negedge rst_n)
begin
    if (~rst_n) cnt<=32'b0;
    else if (cnt_en) cnt<=cnt+32'b1;
    else cnt<=32'b0;
end

endmodule

```

3) 时钟延缓模块 1

```

module clk_divide(
    input clk,
    input rst_n,
    output reg clk_slow
);

reg [31:0] cnt_div;

always @(posedge clk or negedge rst_n)

```

```

begin
  if (~rst_n) cnt_div<=32' h0;
  else if (cnt_div == 32' d99_999) cnt_div<=32' h0;
  else cnt_div<=cnt_div+32' h1;
end

always @(posedge clk or negedge rst_n)
begin
  if (~rst_n) clk_slow<=1' b0;
  else if (cnt_div==32' d99_999) clk_slow<=1' b1;
  else clk_slow<=1' b0;
end

endmodule

```

4) 时钟延缓模块 2

```

module clk_divide_addr(
  input clk,
  input rst_n,
  output reg clk_slow
);

reg [31:0] cnt_div;

always @(posedge clk or negedge rst_n)
begin
  if (~rst_n) cnt_div<=32' h0;
  else if (cnt_div == 32' d99_999_999) cnt_div<=32' h0;
  else cnt_div<=cnt_div+32' h1;
end

always @(posedge clk or negedge rst_n)
begin
  if (~rst_n) clk_slow<=1' b0;
  else if (cnt_div==32' d99_999_999) clk_slow<=1' b1;
  else clk_slow<=1' b0;
end

endmodule

```

5) 自动地址生成模块

```

module read_addr_gen(
  input clk,
  input rst_n,

```

```

output reg [7:0] read_addr
);

always @(posedge clk or negedge rst_n)
begin
    if (~rst_n)
        read_addr <= 8'd0;
    else
        begin
            if (read_addr < 8'b11111111) read_addr <= read_addr + 8'd1;
            else read_addr <= 8'd0;
        end
    end
endmodule

```

6) 七段显示器控制模块及其解码器

```

module seg(
    input clk,
    input rst_n,
    input [31:0] data,
    output reg [7:0] sel,
    output reg [7:0] segment
);

reg [2:0] counter;
wire [7:0] segment7;
wire [7:0] segment6;
wire [7:0] segment5;
wire [7:0] segment4;
wire [7:0] segment3;
wire [7:0] segment2;
wire [7:0] segment1;
wire [7:0] segment0;
decoder digit7(data[31:28], segment7);
decoder digit6(data[27:24], segment6);
decoder digit5(data[23:20], segment5);
decoder digit4(data[19:16], segment4);
decoder digit3(data[15:12], segment3);
decoder digit2(data[11:8], segment2);
decoder digit1(data[7:4], segment1);
decoder digit0(data[3:0], segment0);

initial

```

```

counter<=3' d0;

always @(posedge clk or negedge rst_n)
begin
  if (~rst_n)
    begin
      counter<=3' d0;
      sel<=8' b00000000;
      segment<=8' b0000_0000;
    end
  else begin
    case (counter)
      3'd0: begin sel<=8' b1111110; segment<=segment7; end
      3'd1: begin sel<=8' b11111101; segment<=segment6; end
      3'd2: begin sel<=8' b11111011; segment<=segment5; end
      3'd3: begin sel<=8' b11110111; segment<=segment4; end
      3'd4: begin sel<=8' b11101111; segment<=segment3; end
      3'd5: begin sel<=8' b11011111; segment<=segment2; end
      3'd6: begin sel<=8' b10111111; segment<=segment1; end
      3'd7: begin sel<=8' b01111111; segment<=segment0; end
    endcase
    if (counter<3'd7) counter<=counter+3'd1;
    else counter<=3'b0;
  end
end
endmodule

module decoder(
  input [3:0] hex,
  output reg [7:0] data
);

always @(*)
begin
  case(hex)
    4'h0: data = 8' b0000_0011;
    4'h1: data = 8' b1001_1111;
    4'h2: data = 8' b0010_0101;
    4'h3: data = 8' b0000_1101;
    4'h4: data = 8' b1001_1001;
    4'h5: data = 8' b0100_1001;
    4'h6: data = 8' b0100_0001;
    4'h7: data = 8' b0001_1111;
    4'h8: data = 8' b0000_0001;
  endcase
end

```

```

        4' h9: data = 8' b0000_1001;
        4' hA: data = 8' b0001_0001;
        4' hB: data = 8' b1100_0001;
        4' hC: data = 8' b0110_0011;
        4' hD: data = 8' b1000_0101;
        4' hE: data = 8' b0110_0001;
        4' hF: data = 8' b0111_0001;
        default: data = 8' b1111_1111;
    endcase
end

endmodule

```

(17) ucf 文件

```

## Clock signal
NET "clk" LOC = "E3" | IOSTANDARD = "LVCMOS33";
#Bank = 35, Pin name = #IO_L12P_T1_MRCC_35, Sch
name = clk100mhz
NET "clk" TNM_NET = sys_clk_pin;
TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 100 MHz HIGH 50%;
```

```

## Switches
NET "addr_in<0>" LOC=J15 | IOSTANDARD=LVCMOS33;
#IO_L24N_T3_RS0_15
NET "addr_in<1>" LOC=L16 | IOSTANDARD=LVCMOS33;
#IO_L3N_T0_DQS_EMCCCLK_14
NET "addr_in<2>" LOC=M13 | IOSTANDARD=LVCMOS33;
#IO_L6N_T0_D08_VREF_14
NET "addr_in<3>" LOC=R15 | IOSTANDARD=LVCMOS33;
#IO_L13N_T2_MRCC_14
NET "addr_in<4>" LOC=R17 | IOSTANDARD=LVCMOS33;
#IO_L12N_T1_MRCC_14
NET "addr_in<5>" LOC=T18 | IOSTANDARD=LVCMOS33;
#IO_L7N_T1_D10_14
NET "addr_in<6>" LOC=U18 | IOSTANDARD=LVCMOS33;
#IO_L17N_T2_A13_D29_14
NET "addr_in<7>" LOC=R13 | IOSTANDARD=LVCMOS33;
#IO_L5N_T0_D07_14

## Buttons
NET "mode_change" LOC=N17 | IOSTANDARD=LVCMOS33;
#IO_L9P_T1_DQS_14
NET "rst" LOC=P17 | IOSTANDARD=LVCMOS33;
#IO_L12P_T1_MRCC_14\
```

```

## LEDs
NET "led<0>" LOC=H17 | IOSTANDARD=LVCMOS33;
#IO_L18P_T2_A24_15

NET "led<1>" LOC=K15 | IOSTANDARD=LVCMOS33;
#IO_L24P_T3_RS1_15

NET "led<2>" LOC=J13 | IOSTANDARD=LVCMOS33;
#IO_L17N_T2_A25_15

NET "led<3>" LOC=N14 | IOSTANDARD=LVCMOS33; #IO_L8P_T1_D11_14
NET "led<4>" LOC=R18 | IOSTANDARD=LVCMOS33; #IO_L7P_T1_D09_14
NET "led<5>" LOC=V17 | IOSTANDARD=LVCMOS33;

#IO_L18N_T2_A11_D27_14
NET "led<6>" LOC=U17 | IOSTANDARD=LVCMOS33;
#IO_L17P_T2_A14_D30_14

NET "led<7>" LOC=U16 | IOSTANDARD=LVCMOS33;
#IO_L18P_T2_A12_D28_14

## 7 segment display
NET "seg<7>" LOC=T10 | IOSTANDARD=LVCMOS33;
#IO_L24N_T3_A00_D16_14

NET "seg<6>" LOC=R10 | IOSTANDARD=LVCMOS33; #IO_25_14
NET "seg<5>" LOC=K16 | IOSTANDARD=LVCMOS33; #IO_25_15
NET "seg<4>" LOC=K13 | IOSTANDARD=LVCMOS33;

#IO_L17P_T2_A26_15
NET "seg<3>" LOC=P15 | IOSTANDARD=LVCMOS33;
#IO_L13P_T2_MRCC_14

NET "seg<2>" LOC=T11 | IOSTANDARD=LVCMOS33;
#IO_L19P_T3_A10_D26_14

NET "seg<1>" LOC=L18 | IOSTANDARD=LVCMOS33;
#IO_L4P_T0_D04_14

NET "seg<0>" LOC=H15 | IOSTANDARD=LVCMOS33;
#IO_L19N_T3_A21_VREF_15

NET "sel<7>" LOC=J17 | IOSTANDARD=LVCMOS33;
#IO_L23P_T3_FOE_B_15

NET "sel<6>" LOC=J18 | IOSTANDARD=LVCMOS33;
#IO_L23N_T3_FWE_B_15

NET "sel<5>" LOC=T9 | IOSTANDARD=LVCMOS33;
#IO_L24P_T3_A01_D17_14

NET "sel<4>" LOC=J14 | IOSTANDARD=LVCMOS33;
#IO_L19P_T3_A22_15

NET "sel<3>" LOC=P14 | IOSTANDARD=LVCMOS33;
#IO_L8N_T1_D12_14

NET "sel<2>" LOC=T14 | IOSTANDARD=LVCMOS33;

```

```

#IO_L14P_T2_SRCC_14
NET "sel<1>"          LOC=K2 | IOSTANDARD=LVC MOS33; #IO_L23P_T3_35
NET "sel<0>"          LOC=U13 | IOSTANDARD=LVC MOS33;
#IO_L23N_T3_A02_D18_14

```

2、仿真代码：

```

module test;

    // Inputs
    reg clk;
    reg reset;
    reg [7:0] read_addr_out;

    // Outputs
    wire [31:0] read_out;

    // Instantiate the Unit Under Test (UUT)
    PipelineCPU uut (
        .clk(clk),
        .reset(reset),
        .read_addr_out(read_addr_out),
        .read_out(read_out)
    );

    initial begin
        // Initialize Inputs
        clk = 0;
        reset = 0;
        read_addr_out = 0;

        // Wait 100 ns for global reset to finish
        #100;
        reset =1;
        // Add stimulus here

    end

    always
    begin
        #10 clk=~clk;
    end
endmodule

```

2、CPU 设计更新日志

MIPS CPU v1.0

创建日期: 2018. 5. 14

更新日期: 2018. 5. 22

更新日志:

- 1、实现基本 5 级 MIPS 流水线
- 2、加入处理相关的模块
- 3、加入指令锁模块
- 4、完成对斐波拉契指令的流水线执行

MIPS CPU v1.1

更新日期: 2018. 5. 23

更新日志:

- 1、调整数据通路
- 2、增加 ALU 的比较 zero 信号
- 3、完成对相关运算流水线指令执行
- 4、增加部分分支指令

MIPS CPU v1.5

更新日期: 2018. 6. 6

更新日志:

- 1、调整数据通路
- 2、增加多条逻辑运算指令
- 3、具体实现的指令:
add、addu、sub、subu、and、or、nor、xor
addi、addiu、andi、ori、xori、lui
sll、sllv、srl、srlv、sra、srav
slt、sltu、slti、sltui
bltz、beq、bne、blez、bgtz
sw、lw
j
halt
- 4、实现下载:
(1) 可切换的双模式
(2) 自动运行斐波拉契指令
(3) 可手动输入查看存储器内数字
(4) 可自动顺序显示存储器内数字

MIPS CPU v2.0

更新日期: 2018. 6. 7

更新日志:

- 1、大幅度调整数据通路
- 2、增加链接和转跳分支类指令
- 3、增加具体实现的指令数:

add、addu、sub、subu、and、or、nor、xor
addi、addiu、andi、ori、xori、lui
sll、sllv、srl、srlv、sra、srav
slt、sltu、slti、sltui
bltz、beq、bne、blez、bgtz、bgezal、bltzal
sw、sb、sbu
lw、lb、lbu、lh、lhu
j、jr、jal、jalr
halt

4、加入 3 组测试数据，并实现下载