流水线 MIPS 处理器设计

赵浩然 胡沿名 冯书瀚

（电子科技大学英才实验学院，四川省成都市）

摘要：本文对 32 位 MIPS 架构的功能、指令做了简单介绍，同时借助 Vivado 设计平台利用 Verilog 语言对完成了一种 32 位（每字 4 字节）按字编址，支持多种算数运算指令的 6 周期流水线 MIPS 处理器架构，通过其指令分别实现了对两个 32 维 16 位数向量卷积运算和对一个16×16的 16 位数矩阵的行列交织功能，并针对上述基础指令功能和卷积、行列交织扩展功能进行仿真以及综合上板测试。

关键字：32 位 MIPS ；流水线 ； Vivado ；Verilog ；卷积 ；行列交织 ；FPGA

1 MIPS 简介

MIPS（Microprocessor without Interlocked Piped Stages）架构于上世纪 80 年代开发并应用，作为一种 RISC（Reduced Instruction Set Computing）处理器，MIPS 具有简洁、优性能、低功耗、高扩展性的特点（例如从初始的 32 位结构扩展到最新的 64 位结构）。

MIPS 架构采用定长指令集，覆盖了几乎全部经常使用且功能实用的指令，并利用寄存器组和高速缓存优化了数据读取、存储的进程，同时还可以支持多周期处理、流水线控制等优化方法，以提升处理器的处理性能。

以 32 位 MIPS 为例，一个基础功能完善的 MIPS 架构必须满足以下 3 种结构指令的正常执行：

R 型指令：

op	rs	rt	rd	shamt	funct

R 型指令即为寄存器指令，在 32 位 MIPS 处理器中其长度同样为 32 位，其中操作码 op 长为 6 位，用以指示指令对应的操作类型（对 R 型指令而言 op 均为 0）；rs、rt、rd 均为 5 位，对应 32 位 MIPS 中 32 个寄存器地址，其中 rd 为目的寄存器，rs、rt 均为源寄存器；shamt 字段长 5 位，专用于移位指令中记录位移量；funct 长为 6 位，用以区分 R 型指令的功能类型（考虑到 R 型指令操作码字段 op）。

I 型指令：

op	rs	rt	imm

I 型指令即为立即数指令，其 32 位中操作码 op 同样占 6 位，用以表示指令执行的操作类型；rs、rt 也为 5 位，指示源操作数所在的寄存器地址或写入的目的寄存器地址（在需要写入寄存器时 rt 记录目的寄存器地址）；imm 表示 16 位立即数，可作为指令寻址的偏移量或指令运算中的常数，在 32 位系统中 16 位的立即数通常需要进行 0 扩展或符号扩展到 32 位。

J 型指令：

op	addr

J 型指令通常为无条件跳转指令，其中操作码 op 同样为 6 位，其余 26 位用来得到跳转的目的地址（并非直接寻址，因为 addr 子段长只有 26 位而 pc 是 32 位）——通常将 pc 的前 4 位加在 addr 子段前端，并在末尾添加 00 子段（32 位 MIPS 按字节编址，所以每条指令或数据占据 4 个存储单元，地址一定能被 4 整除）得到跳转后的目的地址。

为了支持并完成指令集中包含的多种指令，MIPS 处理器需要完善且尽可能高效的硬件结构。

用最简单的描述，一个完善且能完成基础指令执行功能的 MIPS 架构需要包含以下模块：存储器，用以储存输入的指令和数据（这一部分实际上并不包含在 MIPS 处理器中，但对处理器工作的进行无疑是必须的）；寄存器组，在处理器中储存要处理的指令、数据以及指令执行所得结果等等；算术逻辑单元 ALU（Arithmetic and Logic Unit），在处理器中完成指令对应的算术运算和逻辑运

算，得到指令需要的运算结果和标志信息；总线，在各模块间传输相关的指令、数据等信息；控制器，通过对指令译码生成执行指令所需用以控制其他模块的控制信号；输入输出设备。

除了最基础的功能外，还可以对 MIPS 处理器进行诸如增加闪存等额外硬件模块、处理器多核运行以及对指令多周期处理、流水线处理等优化，以增强处理器性能。

2 指令集设计简述

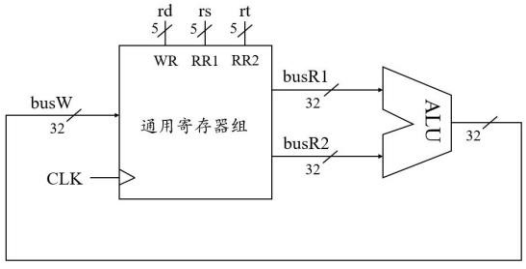
本文涉及的处理器架构设计中指令集寻址方式与 MIPS 架构相同的基址寻址、寄存器寻址、立即数寻址、PC 相对寻址和伪直接寻址等寻址方式，指令集中各指令结构均严格按照 R 型、I 型、J 型三种指令格式，具体涉及的指令如下：

2.1 add

add 指令格式为：

000000	rs	rt	rd	shamt	100000

add 指令作为 R 型指令，其操作码 op 为 000000，功能字段 funct 为 100000 在控制器中将生成对应 ALU 控制信号，其余 4 字段 rs、rt、rd、shamt 均为 5 位，shamt 字段在 add 指令中并无直接作用，该指令功能为将寄存器 rs、rt 中存储的数相加后写入寄存器 rd 中，即$R[rd] \leftarrow R[rs] + R[rt]$。其对应数据通路如下图所示（为方便表示与理解，此处数据通路仅单独表示 add 指令执行时的数据流程，且只涉及加法功能本身，忽略 PC 改变、取指译码、流水优化以及控制信号等部分，这些部分将在后文提及并阐述）。



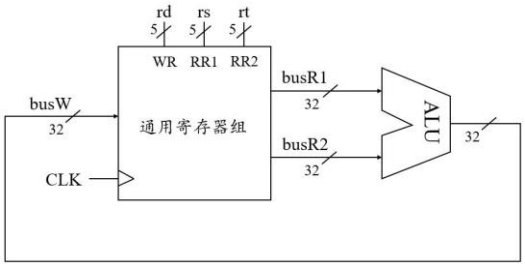
图中通用寄存器组（即 MIPS32 位通用寄存器组，由 32 个约定功能的通用寄存器组成，后文将对该模块详细阐述）中 WR 为寄存器写地址，RR1、RR2 均为寄存器读地址，3 者分别对应了指令中的 rd、rs 和 rt，三条数据总线 busR1、busR2、busW 则分别传输从 RR1、RR2 读出的寄存器内容和将写入寄存器 WR 的值；ALU 为算数逻辑单元，受相应控制信号控制，将数据总线 busR1、busR2 传输的数据执行加法运算后传输到 busW。

2.2 sub

sub 指令格式为：

000000	rs	rt	rd	shamt	100010

sub 同样作为 R 型指令，与 add 指令极为相似，只是功能字段 funct 为 100010，其功能为将寄存器 rs 中存储的数减去 rt 中储存的数后将结果写入寄存器 rd 中，即$R[rd] \leftarrow R[rs] - R[rt]$，对应数据通路如下图所示。



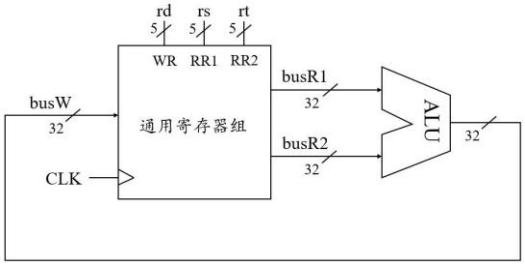
sub 指令对应数据传输与 add 指令并无差异，只是 ALU 中相关控制信号与功能执行发生改变。

2.3 mul

mul 指令格式为：

000000	rs	rt	rd	shamt	011100

mul 指令也是为 R 型指令，与 add、sub 指令本身也只是功能字段的差异——011100，其功能为将寄存器 rs、rt 中存储的数相乘后将结果写入寄存器 rd 中，即$R[rd] \leftarrow R[rs] * R[rt]$，对应数据通路如下图所示。



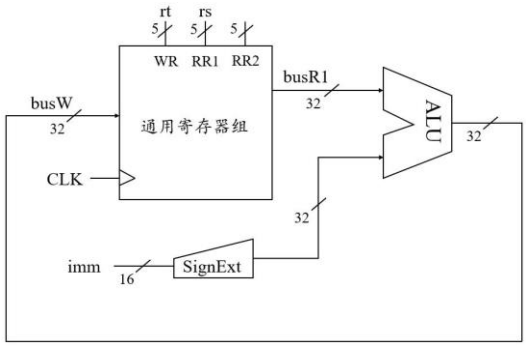
事实上以上 3 种指令，或者是所有的 R 型指令对应的数据通路基本保持一致，只是 ALU 中相关控制信号与执行不同。

2.4 addi

addi 指令格式为：

001000	rs	rt	imm

addi 指令为 I 型指令而非 R 型，不再具有功能字段 funct 且在控制器中将对操作码 op 译码以生成相应控制信号，rs 和 rt 同样为 5 位，imm 字段表示一个 16 位的立即数，需要进行符号扩展后才能在 ALU 中参与运算，该指令功能为将寄存器 rs 中存储的数与符号扩展后的立即数相加并写入寄存器 rt 中，即$R[rt] \leftarrow R[rs] + SignExt(imm)$。其对应数据通路如下图所示。



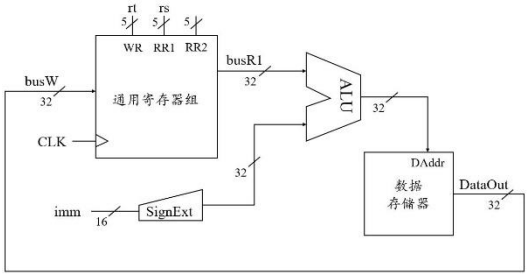
由于数据总线 busR2 并没有将数据传向 ALU，通用寄存器组中读地址 RR2 输出总线在相关控制单元控制下便不会影响进程（直接保留上次使用的值不做修改是个不错的选择）。立即数 imm 经符号扩展模块 SignExt 拓展到 32 位后传输到 ALU 参与运算，所得结果同样传输到总线 busW。

2.5 lw

lw 指令格式为：

100011	rs	rt	imm

lw 指令为 I 型指令，操作码为 100011，其中寄存器 rs 存储值与经符号扩展后的立即数 imm 相加后得到待取数的主存存储单元地址，然后从中取数再存入寄存器 rt 中，即$R[rt] \leftarrow M[R[rs] + SignExt(imm)]$。其数据通路如下图所示。



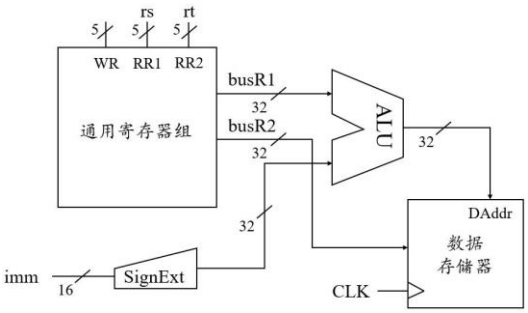
由于 lw 指令采用立即数寻址，数据存储器读地址的计算本身便需要使用立即数加法运算，所以 lw 指令的数据通路与 addi 指令数据通路相似度较高，只是 ALU 运算结果写入了数据存储器的读地址，而读取的结果 DataOut 传输到了总线 busW 上。

2.6 sw

sw 指令格式为：

101011	rs	rt	imm

sw 指令作为 I 型指令，操作码 101011 用于控制器译码时生成控制信号，其功能与 lw 指令正好相反——16 位立即数 imm 经符号扩展后与寄存器 rs 中存储结果相加后作为数据存储器目的地址并将寄存器 rt 中的存储值写入相应存储单元，即$M[R[rs] + SignExt(imm)] \leftarrow R[rt]$。对应数据通路如下图所示。



由于 sw 指令中并无数据写入通用寄存器组，所以数据总线 busW 和通用寄存器组写地址 WR 都不被使用；寄存器 rs 的存储结果由总线 busR1 传输到 ALU 并与经符号扩展的立即数 imm 相加后结果传输到数据存储器作为写地址（与 lw 指令中提及的读地址其实是同一个地址端口，二者按功能不同分开描述以方便理解），寄存器 rt 中存储结果由数据总线 busR2 传输到数据存储器并写入到地址指示的存储单元。

2.7 指令集设计总结

鉴于相关测试标准与功能要求，考虑降低开销、提升性能的目标，本次设计的 MIPS 处理器仅覆盖了 add、sub、mul 和 addi 这四种算术运算指令和 lw、sw 这两种存取数指令，对无符号数指令、逻辑运算指令、移位指令、移位指令等均不支持，但其数据通路仍略有复杂，需要以下控制信号以完成这六种指令工作的顺利执行：

2.7.1 通用寄存器组写信号

通用寄存器组的读操作被设计为组合逻辑，当读地址有效时即可取数到相应总线（无关的取数会被其他控制信号与控制单元限制而不影响当前指令的执行）；而写操作为时序逻辑，若无其他控制，在时钟上升沿到来后无论数据总线 busW 上传输何值（哪怕是 0）都会被自动写入寄存器中，因此需要一个写使能信号控制寄存器组的写操作，当且仅当时钟到达上升沿且写使能为 1 时才能写入寄存器。

2.7.2 数据存储器写信号

数据储存器读写操作均为时序逻辑，在无其他信号控制下，每个时钟上升沿到来时地址对应存储单元存储结果自动传输到 DataOut 端口，而 DataIn 端口将自动写入地址对应存储单元。同样的，无关的读操作可以依靠其他控制单元避免对指令执行带来影响，但写操作必须加以控制否则会错误的修改存储器存数情况。此外，在读写操作共用一个地址端口时写使能信号也能有效区分二者。

2.7.3 ALU 控制信号

算术逻辑单元 ALU 需要通过控制器对指令操作码或功能码译码后生成的控制信号以决定其执行的运算功能（在本设计中有 add，sub，mul 和 addi）。

2.7.4 通用寄存器组写入地址选择控制信号

对于需要写入通用寄存器组的 R 型指令和 I 型指令，其写地址分别是 rd 和 rt，因此需要一个 2 选 1 选择器并用一个 1 位控制信号控制写地址的选择。

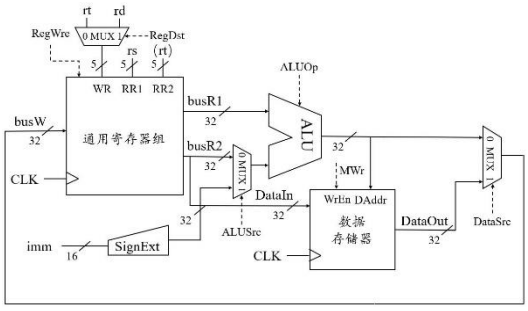
2.7.5 ALU 输入选择控制信号

ALU 的两个输入一个保持由数据总线 busR1 传输，另一个在 ALU 执行 R 型指令时与总线 busR2 相连，在执行 addi 指令时输入为经符号扩展后的立即数 imm。因此需要一个 2 选 1 选择器以及一个 1 位控制信号控制 ALU 输入的选择。

2.7.6 通用寄存器组写入选择控制信号

对于 R 型以及 addi 指令，ALU 运算结果写入通用寄存器组；而对于 sw 指令，将数据存储器读数结果写入寄存器组。因此需要一个 2 选 1 选择器以及一个 1 位控制信号控制通用寄存器组写入选择。

完整的带选择器与控制信号的数据通路如下图所示：



相关模块与控制信号将在下文详细阐述。

3 处理器模块设计

本文涉及的处理器架构设计包含的模块可主要分为以下几类：控制器、算术逻辑单元 ALU、通用寄存器、存储器、立即数扩展模块、选择器模块、下一地址逻辑模块和顶层模块。

3.1 控制器

控制器对于当前所取所取指令进行译码，由于决定控制信号的通常为指令的操作码字段和功能字段，因此控制器具有相应的两个 6 位的输入端口 opcode 和 funct。

此外，由上文指令集数据通路可知，本实验设计的 MIPS 处理器架构一共需要 6 种控制信号：寄存器写使能 RegWre、寄存器写目的地址选择信号 RegDst、ALU 源操作数选择信号 ALUSrc、ALU 功能控制信号 ALUOp、数据存储器写使能 MWr 以及寄存器写入选择信号 DataSrc；且 ALUOp 设计为三位以满足相应功能需要，因此一共需要 5 个 1 位输出端口和 1 个 3 位输出端口。

对于指令译码过程，先对操作码 opcode 进行判断，如果 opcode 为 000000（即指令为 R 型地址）再进一步对 funct 进行译码。

对指令译码结果分别如下：

add 指令：opcode=000000，funct=100000，相应控制信号分别为：ALUSrc=0、DataSrc=0、RegWre=1、MWr=0、RegDst=1、ALUOp=000；

sub 指令：opcode=000000，funct=100010，相应控制信号为：ALUSrc=0、DataSrc=0、

RegWre=1、MWr=0、RegDst=1、ALUOp=001；

mul 指令：opcode=000000，funct=100010，相应控制信号为： ALUSrc=0、 DataSrc=0、RegWre=1、MWr=0、RegDst=1、ALUOp=111；

addi 指令：opcode=001000，相应控制信号分别为：ALUSrc=1、DataSrc=0、RegWre=1、MWr=0、RegDst=0、ALUOp=000;

lw 指令：opcode=100011，相应控制信号分别为：ALUSrc=1、DataSrc=1、RegWre=1、MWr=0、RegDst=0、ALUOp=000;

sw 指令：opcode=101011，相应控制信号分别为：ALUSrc=1、DataSrc=z、RegWre=0、MWr=1、RegDst=0、ALUOp=000。

考虑到代码长度，此处仅展示部分：

```
module ControlUnit(
  input [5:0] opcode,
  input [5:0] funct,
  input zero,
  input sign,
  output ..... (此处省略)
  );
  parameter [5:0] sub = 6'b100010,
                  add = 6'b100000,
                  addi = 6'b001000,
                  lw = 6'b100011,
                  sw = 6'b101011,
                  mul = 6'b011100;
  always @(opcode or zero or sign or funct) begin
    case(opcode)
      6'b000000: //R型
        begin
        case(funct)
          add:begin
            PCWre <= 1; // add
            {ALUSrcA, ALUSrcB, DBDataSrc, RegWre, mWR, RegDst, ExtSel} <= 7'b00_0_10_1_0;
            ALUOp[2:0] <= 3'b000;
          end
      ..... (此处省略)
    endcase
  end
endmodule
```

3.2 通用寄存器组

通用寄存器组具有读、写以及复位三种操作：读操作为组合逻辑且可同时读取两个寄存器；写操作为时序逻辑，在时钟上升沿且写使能信号有效时即写入寄存器；复位信号执行异步复位功能，生效后能将寄存器组中 32 个寄存器全部还原为 0。

因而该模块一共需要时钟 clk、复位信号 Reset、写使能 WE 三个 1 位输入端口、2 个读地址和 1 个写地址 3 个 3 位输入端口以及 2 个 32 位输出端口，其代码如下所示：

```
module RegisterFile(
  input clk,Reset,WE,
  input [4:0] ReadReg1,[4:0] ReadReg2, [4:0] WriteReg,
  input [31:0] WriteData,
  output [31:0] ReadData1,[31:0] ReadData2
  );
  reg [31:0] file [1:31];
  integer i;
  assign ReadData1 = (ReadReg1 == 0) ? 0 : file[ReadReg1];
  assign ReadData2 = (ReadReg2 == 0) ? 0 : file[ReadReg2];//零号寄存器位0
  always @(negedge clk or negedge Reset) begin
    if(Reset == 0) begin
      for(i = 1; i <= 31; i=i+1) begin
        file[i] <= 0;
      end
    end
    else if(WE == 1 && WriteReg != 0)
      file[WriteReg] <= WriteData;
  end
endmodule
```

3.3 存储器模块

通常在 MIPS 处理器中，数据和指令都存储在主存储器中，在执行取指令、读数操作时才分别将对应指令、操作数从主存中取出并写入处理器中指令寄存器与数据寄存器。本次实验出于简便设计、减少开销优化性能的目的，直接在处理器架构中分别设计一个指令存储器模块和数据存储器模块，输入的指令和数据分别直接且按序写入对应存储器存储单元中。

3.3.1 指令存储器

指令存储器只执行取指令操作，该操作为组合逻辑，因此该模块只有共两个 32 位的读地址端口和指令输出端口。

指令存储器存数方式为从地址为 0 的存储单元开始按序存储，对应 PC 从 00400000 开始。同时由于本次实验设计的处理器架构为按字编址，因此利用 PC 对指令寻址时需要在减 00400000 后再右移 2 位（除 4，即从字节编址到按字编址），同时考虑到存储器大小限制当取指地址不能太大。

对应代码如下所示：

```
module InstructionMemory(
   input [31:0] IAddr,
   output  [31:0]  IDataOut
   );
  wire [31:0] dataouttmp;
  wire [31:0] addr;
  dist_mem_gen_1 im(.a(addr[12:0]),.spo(dataouttmp));
  parameter initaladdr = 32'h00400000;//指令起始地址，参考mars4_5的起始地址
  assign addr[31:0] = (IAddr-initaladdr)>>2;//修正下文件中的地址
  assign IDataOut = (addr[13])? {32'hx}:dataouttmp;//最大指令2*2**12=8192条
endmodule
```

3.3.2 数据存储器

数据存储器有读和写两种操作，相关功能与控制信号在上文已叙述详尽，在此不再赘述。因此该模块一共有 5 个端口：1 位时钟 clk 输入，1 位写使能 WR，32 位的读地址与数据写入端口以及 32 位数据输出端口。此外相关地址的计算需考虑处理器为按字编址且存储器从地址 0 开始按序储存。

对应代码如下所示：

```
module DataMemory(
   input clk,
   input [31:0] DAddr,
   input [31:0] DataIn,
   input WR,   // 写控制，1有效
   output [31:0] DataOut   // 读出32位数据
   );
   dist_mem_gen_0  ram(.clk(clk),.we(WR),.a(DAddr[7:0]),.d(DataIn),.spo(DataOut));//按字存储
endmodule
```

3.4 立即数扩展模块

指令输入的立即数只有 16 位，需要扩展到 32 位才能参与运算。通常需要考虑符号扩展与 0 扩展两种方式，但在本次实验中

所有涉及立即数运算的指令均采用符号扩展，所以该模块只用含有符号扩展功能，也不再需要控制信号控制，只需要一个 16 位的输入端口和扩展后的 32 位输出端口。

其对应代码如下：

```
module ImmediateExtend(
   input [15:0] original,
   output reg [31:0] extended
   );
   always @(*) begin
      extended[15:0] <= original;    // 低16位保持不变
      if(original[15] == 0) extended[31:16] <= 0;
      else extended[31:16] <= 16'hFFFF;
      end
   end
endmodule
```

3.5 选择器模块

本次设计的 MIPS 处理其中涉及多个 32 位 2 选 1 选择器，因此该模块只需要两个 32 位数据输入端口，1 个 1 位选择信号输入端口以及 1 个 32 位数据输出端口，根据选择信号决定输出数据为两个输入数据中的那哪一个。

对应代码如下所示：

```
module Mux2_32bits(
   input choice,
   input [31:0] in0,
   input [31:0] in1,
   output [31:0] out
   );
   assign out = (choice==0) ? in0 : in1;
endmodule
```

3.6 下一地址逻辑模块

本次实验设计的处理器架构由于指令集不覆盖跳转指令，下一地址只会来自于 PC 的按序递增。此外本次设计的下一地址 PC 自增的过程并非在 PC 模块中实现，而是在顶层模块中进行，PC 模块只负责进行下一地址的转移，该操作为时序逻辑，且有 PC 写控制信号进行控制；另外该模块还具有复位功能，能将 PC 初始化到首地址 0040000。故该模块共有 clk、Reset、PCWre 三个 1 位输入端口，1 个 32 位地址写入端口和 1 个 32 位地址输出端口。

另外，由于实验设计的处理器指令存储器容量有限，PC 的实际最大值不会是 32 位的理论最大值 FFFFFFFF，因此在模块中还设计了地址上限，当下一地址超过上限后将自动跳往 PC 最终位置。

对应代码如下所示：

```
module PC(
   input clk,
   input Reset,
   input PCWre,
   input [31:0] nextIAddr,
   output reg [31:0] currentIAddr
   );
   parameter initaladdr = 32'h00400000;
   parameter endaddr = 32'h00405000;//结束地址
   parameter endaddrto = 32'hFFFFFFF0;//结束后跳转
   initial currentIAddr <= initaladdr
   always @(posedge clk or negedge Reset) begin
      if(Reset == 0) currentIAddr = initaladdr;//起始地址
      else
      begin
         if(nextIAddr>=endaddr) currentIAddr<=endaddrto ;//设置PC最大地址防止一直运行出现bug
         if(PCWre == 1&&nextIAddr<endaddr) currentIAddr <= nextIAddr;
         if(PCWre == 0&&nextIAddr<endaddr)currentIAddr <= currentIAddr;
      end
   end
endmodule
```

3.7 算数逻辑单元

本次实验设计的 ALU 共支持 3 种运算功能：加法（add 和 addi 的区别只在调用 ALU 模块是输入端口输入对应不同的操作数来体现）、减法和乘法，其运算思路为对输入数据将相应的加法、减法和乘法结果分别计算之后，输出端口根据控制信号 ALUOp 选择对应结果；由于设计要求，该模块中所有运算语句均采用门级运算。

该模块被设计为组合逻辑，含有 3 位控制信号 ALUOp、两个 32 位输入端口和 1 个 32 位结果输出端口，以及 2 个 1 位的零标志和符号标志信号。

3.7.1 ALU 加法

ALU 的加法运算被设计为 32 位补码加法，通过将 32 位输入操作数分别以低 16 位和高 16 位结合进位输入信号调用模块 bk32 得到运算结果。

模块 bk32 运用前缀加法算法进行 16 位补码加法运算，其端口包括 1 位进位输入、2 个 16 位源操作数输入，1 个 16 位运算结果输出和 1 位输出进位标志。

在前缀加法运算进行过程中，先通过输入各位与、异或运算得到每一位的传播进位信号p_i、产生进位信号g_i，再利用公式：

$$P_{i:j} = P_{i:k} \cdot P_{k-1:j}$$

$$G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j}$$

生成所需的各个前缀信号，最后利用公式$S_i = G_{i-1:-1} \oplus A_i \oplus B_i (A_i \oplus B_i = p_i)$计算出所需结果。考虑到代码长度，该模块代码只展示部分：

```
module bk32(
 input [15:0] A,
   input [15:0] B,
   input cin,
   output [15:0] sum,
   output cout
   );
   wire [15:0] g,p,G,P;
   assign g[0]=A[0]&B[0];
   assign p[0]=A[0]^B[0];
   assign G[0]=g[0];
   assign P[0]=p[0];
   ..... (此处省略)
   wire [3:0]m,n;
   ..... (此处省略)
   assign ig=mp&hg|mg;
   assign ip=mp&hp;
   ..... (此处省略)
   assign sum[0]=p[0]^cin;
   ..... (此处省略)
   assign cout=ig;
   endmodule
```

3.7.2 ALU 减法

ALU 的减法运算同样被设计为 32 位补码减法，不过并不需要为此设计新的模块，只需要在调用加法运算模块时对减数求补即可。

3.7.3 ALU 乘法

ALU 的乘法运算被设计为 16 位补码乘法，采用布斯算法进行运算。经典布斯算法的基本运算思想如下：

两个 n 位乘数 x，y 进行补码乘法结果公式可化为：

$$[x \times y]_{补} = [2^n \times x \times \sum_{i=0}^{n-1}(Y_{i-1} - Y_i)2^{-(n-i)}]_{补}$$

此处定义 $[y]_{补} = Y_{n-1} \dots Y_1 Y_0$，且 $Y_{-1} = 0$，于是得到部分积 P_i 递推公式为：

$$[P_{i+1}]_{补} = [2^{-1}(P_i + (Y_{i-1} - Y_i)x)]_{补}$$

其中 $[P_0]_{补} = 0$，于是可知

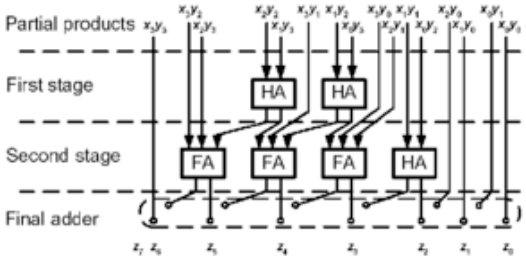
$$[x \times y]_{补} = 2^n [P_n]_{补}$$

在实际操作中经典布斯算法利用串行工作方式，按序比较乘数连续两位 $Y_i Y_{i-1}$（有 00/11,01,10 三种情况），利用递推公式更新部分积最后得到结果。

而本次乘法设计并没有按传统的布斯算法串行递推部分积，而是先比较乘数连续两位，预处理出部分积更新的部分（$+[x]_{补}$，$-[x]_{补}$ 或 0），同时记录各符号位方便最后进行符号补偿，部分代码如下：

```
module   Booth_Classic (
         input     [15 : 0]   M,                    // Multiplicand
         input     [15 : 0]   R,                    // Multiplier

         output    [15 : 0]   pp0, pp1, pp2, pp3,   // PP results
                                                    pp4, pp5, pp6, pp7,
                                                    pp8, pp9, pp10, pp11,
                                                    pp12, pp13, pp14, pp15

         output  [15 : 0]     S                    // Sign extend bit of eac
);
wire [16 : 0]          tmp;
assign tmp = {R, 1'b0};
assign pp0 = (tmp[1:0] == 2'b01 ) ? M :(tmp[1:0] == 2'b10) ? ( ~M + 1'b1 ) : 16'b0;
assign S[0] = pp0[15];
..... (此处省略)
endmodule
```

之后对于预处理过的部分积，使用 Wallace 树形乘法算法进行乘法运算，该乘法算法利用树形结构减少累加次数，运算过程中利用输入与中间结果并行计算，对运算时延有较好的优化效果，下图给出了 Wallace 树形结构的拓扑图像以及本次实验设计的 Wallace 树形乘法模块代码，由于篇幅限制此处仅显示部分：




```
module   WallaceTree16X16 (
                  input     [15:0]     pp0, pp1, pp2, pp3, pp4, pp5,
                                       pp6, pp7, pp8, pp9, pp10, pp11,
                                       pp12, pp13, pp14, pp15,
                  output    [31:0]     opa, opb
         );
//============== First Stage ==========================================
wire      [15: 0]     Fir1_S, Fir1_C, Fir2_S, Fir2_C, Fir3_S, Fir3_C, Fir4_S, Fir4_C, Fir5_S, Fir5_C;
HalfAdder             fir1ha0( pp0[1], pp1[0], Fir1_S[0], Fir1_C[0] );
FullAdder fir1fa1( pp0[2], pp1[1], pp2[0], Fir1_S[1], Fir1_C[1] );
..... (此处省略)
//============== Second Stage =========================================
wire      [15: 0]     Sec1_S, Sec1_C, Sec3_S, Sec3_C;
wire      [17: 0]     Sec2_S, Sec2_C;
..... (此处省略)
//============== Third Stage ==========================================
wire      [17: 0]     Thi1_S, Thi1_C;
wire      [18: 0]     Thi2_S, Thi2_C;
.....
//============== Fourth Stage =========================================
wire      [19: 0]     Fou1_S, Fou1_C;
wire      [15: 0]     Fou2_S, Fou2_C;
..... (此处省略)
//============== Fifth Stage ==========================================
wire      [23: 0]     Fif_S, Fif_C;
..... (此处省略)
//============== Sixth Stage ==========================================
wire      [24: 0]     Six_S, Six_C;
..... (此处省略)
//============== Result Assignment ====================================
assign    opa = { 1'b0, Six_S[24: 0], Fif_S[0], Fou1_S[0], Thi1_S[0], Sec1_S[0], Fir1_S[0], pp0[0] };
assign    opb = { Six_C[24: 0], 7'b0 };
endmodule
```

调用 Wallace 树形乘法模块后 ALU 进行 3 次 32 位补码加法，分别得到符号补偿（来自布斯编码模块所得符号位信息），初步运算结果（将 Wallace 模块调用所得 2 个 32 位输出相加所得）以及二者相加后得到的最终结果。ALU 乘法顶层模块部分代码如下：

```
module   TopMultiplier ( x_in, y_in, result_out );
input     [15: 0]     x_in, y_in;
output    [31: 0]     result_out;

// internal connections
wire      [15:0]      pp0, pp1, pp2, pp3, pp4, pp5, pp6, pp7, pp8, pp9, pp10, pp11, pp12, pp13, pp14, pp15;
wire      [31: 0]     opa, opb;
wire      [15: 0]     sign;
wire      [31: 0]     sign_compensate;
wire      [31: 0]     res_tmp;
// generate PP
Booth_Classic         booth (     .M( x_in ),
                                                .R( y_in ),
                                                .pp0( pp0 ),
                                                ..... (此处省略)
                                                .S( sign )
                                                );
// wallace tree
WallaceTree16X16 wallace (      .pp0( pp0 ),
                                                ..... (此处省略)
                                                .opa( opa),
                                                .opb( opb)
                                                );
// calculate the sign bit compensate
prefixadder32         signcomp (      .A( {~sign, 16'b0} ), .B( {15'b0, 1'b1, 16'b0} ), .Cin( 1'b0 ), .Sum( sign_compensate )
                                                );
// temporary result
prefixadder32         resulttemp (      .A( opa ), .B( opb ), .Cin( 1'b0 ), .Sum( res_tmp )
                                                );
// final result
prefixadder32         result (      .A( res_tmp ), .B( sign_compensate ), .Cin( 1'b0 ), .Sum( result_out)
                                                );
endmodule
```

3.7.4 ALU 输出选择

实验设计 AL 模块采用先计算结果在利用控制信号 ALUOp 选择结果的思路，由于只涉及加、减、乘三种运算，ALUOp 与对应运算结果关系如下表所示：

ALUOp	运算
000	add
001	sub
111	mul

于是可以得到最终结果的门级逻辑表达式：

result<=((~ALUOp[0])&(~ALUOp[1])&(~ALUOp[2])&adderoutput)|(ALUOp[0]&ALUOp[1]&ALUOp[2]&multoutput)|(ALUOp[0]&(~ALUOp[1])&(~ALUOp[2])&suboutput)

综上 ALU 模块代码如下：

```
module ALU(
   input [2:0] ALUOp,
   input [31:0] A,//rs
   input [31:0] B,//rt
   output reg[31:0] result,
   output zero,
   output sign
   );
   wire[31:0] adderoutput,multoutput,suboutput;
   wire [31:0]multemp;
   wire [15:0] addtemp,subtemp;
   assign adderoutput={{16{addtemp[15]}},addtemp};
   assign suberoutput={{16{subtemp[15]}},subtemp};
   bk32 adder(.A(A[15:0]),.B(B[15:0]),.cin(0),.sum(addtemp));//加法器
   bk32 subtractor(.A(A[15:0]),.B(~B[15:0]),.cin(1),.sum(subtemp));//减法器
   TopMultiplier multiple(.x_in(A[15:0]),.y_in(B[15:0]),.result_out(multemp));//乘法器
   assign zero = (result == 0) ? 1 : 0;
   assign sign = result[31];
   assign multoutput={{16{multemp[27]}},multemp[27:12]};
   always @(ALUOp or A or B or adderoutput or multoutput)
    begin
      result[0] <= (~ALUOp[0]&~ALUOp[1]&~ALUOp[2]&adderoutput[0])|//000
                   (ALUOp[0]&ALUOp[1]&ALUOp[2]&multoutput[0])|//111
                  (ALUOp[0]&~ALUOp[1]&~ALUOp[2]&suboutput[0]);//001
      ..... (此处省略)
   end
endmodule
```

3.8 顶层控制模块

顶层模块具有整个处理器正常运行所需的所有输入、输出、信号以及模块调用，相应代码如下所示，由于篇幅限制仅展示部分代码：

```
module CPU(
   input clk,
   input Reset,
   output [31:0] currentIAddr, nextIAddr,
   output [4:0] rs, rt,
   output [31:0] ReadData1, ReadData2, ALU_result, DataBus,
   );
   wire [5:0] opcode, funct
   wire [4:0] rd, WriteReg;
   wire [15:0] immediate;
   wire [31:0] bincode, extended, WriteData, ALU_inA, ALU_inB, DataOut
   wire PCWre, RegWre, ALUSrcA, ALUSrcB,/* mRD,*/ mWR, ExtSel, RegDst, DBDataSrc, ALU_zero, ALU_sign;
   wire [2:0] ALUOp;
   assign funct = bincode[5:0];
   assign opcode = bincode[31:26];
   assign rs = bincode[25:21];
   assign rt = bincode[20:16];
   assign rd = bincode[15:11];
   assign immediate = bincode[15:0];
   assign DataBus = WriteData;
   /* 控制单元 */
   ..... (此处省略)
   /* 5个关键底层模块 */
   PC PC(.clk(clk), .Reset(Reset), .PCWre(PCWre), .nextIAddr(nextIAddr), .currentIAddr(currentIAddr));
   InstructionMemory InstructionMemory(.IAddr(currentIAddr), .IDataOut(bincode));
   RegisterFile RegisterFile(.clk(clk), .Reset(Reset), .WE(RegWre), .ReadReg1(rs), .ReadReg2(rt), .WriteReg(WriteReg),
                             .WriteData(WriteData), .ReadData1(ReadData1), .ReadData2(ReadData2));
   ALU ALU(.ALUOp(ALUOp), .A(ALU_inA), .B(ALU_inB), .result(ALU_result), .zero(ALU_zero), .sign(ALU_sign));
   DataMemory DataMemory( .clk(clk), .DAddr(ALU_result), .DataIn(ReadData2), .WR(mWR), .DataOut(DataOut));
   /* 立即数扩展 */
   ImmediateExtend ImmediateExtend(.original(immediate), .ExtSel(ExtSel), .extended(extended));
   /* 数据选择器 */
   assign nextIAddr=currentIAddr+3'b100;
   ..... (此处省略)
endmodule
```

4 流水线优化

针对一条指令考虑其工作流程，传统的 5 周期流水线设计将其分为取指、译码、执行、存储器和写回 5 个阶段。本实验设计将执行阶段继续细分为取操作数和运算两级，对处理器进行取指、译码、取操作数、运算、存储器和写回的 6 周期流水线优化。

与传统 5 级流水线相同，本实验设计所用 6 级流水线处理器同样会遇到冲突问题，并同样采用重定向和阻塞解决冲突。

本实验流水线处理器所用重定向方法将ALU运算输入数从取数周期(正常取数)、存储器周期(ALU 运算结果)和写回周期(写入寄存器前）中进行选择，该选择由冲突单元控制。

而该流水线处理器所用阻塞处理方法则同样需要在冲突时阻止运算周期前的所有周期，包括取指、译码、取操作数，并通过使能信号控制相关流水寄存器工作，该信号同样由冲突单元控制。

该带冲突处理单元6级流水线处理器通路由于报告篇幅限制诸多细节未能清晰显示，将以附件形式提交、展示。

5 卷积运算功能设计

针对卷积运算输入的 32 维 16 位向量，其中每个元素均按补码储存，主存中各数据均储存在 32 位存储单元的低 16 位，调用 ALU 模块 16 位补码乘法功能与 32 位补码加法进行运算。

执行卷积算法时，先枚举最终结果下标（0 到 62），再枚举第一操作数下标，二者之差为第二操组数下标。另外，值得注意的是该处理器采用按字存储方法，且存储器按序存储，从第一操作数（0 到 31）开始，到第二操作数（32~63），再到卷积结果（64~126），在调用地址时需要在相应下标下进行偏移。

由于篇幅限制，部分汇编代码展示如下：

```
addi $t0,$0,0
lw $t1,64($t0)
lw $t2,0($t0)
lw $t3,32($t0)
mul $t4,$t2,$t3
add $t1,$t1,$t4
sw $t1,64($t0)
lw $t1,65($t0)
lw $t2,0($t0)
lw $t3,33($t0)
mul $t4,$t2,$t3
add $t1,$t1,$t4
lw $t2,1($t0)
lw $t3,32($t0)
mul $t4,$t2,$t3
add $t1,$t1,$t4
sw $t1,65($t0)
..... (此处省略)
```

```
lw $t1,125($t0)
lw $t2,30($t0)
lw $t3,63($t0)
mul $t4,$t2,$t3
add $t1,$t1,$t4
lw $t2,31($t0)
lw $t3,62($t0)
mul $t4,$t2,$t3
add $t1,$t1,$t4
sw $t1,125($t0)
lw $t1,126($t0)
lw $t2,31($t0)
lw $t3,63($t0)
mul $t4,$t2,$t3
add $t1,$t1,$t4
sw $t1,126($t0)
```

此外，为准确、高效生成执行卷积功能的汇编代码（手动复制粘贴修改的流程费时费力且容易出错），本实验设计使用高级程序语言 C++进行编写，相应代码入下：

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
	freopen("juanji.txt","w",stdout);
	printf("addi $t0,$0,0\n");
	for(int i=0;i<=62;i++)
	{
		printf("lw $t1,%d($t0)\n",64+i);
		for(int j=0;j<=31;j++)
		{
			if(i-j>=0 && i-j<=31)
			{
				printf("lw $t2,%d($t0)\n",j);
			printf("lw $t3,%d($t0)\n",32+i-j);
			printf("mul $t4,$t3,$t2\n");
			printf("add $t1,$t1,$t4\n");
			}
		}
		printf("sw $t1,%d($t0)\n",64+i);
	}
	return 0;
}
```

其中i代表对卷积结果下标的枚举，j代表对第一操作数下标的枚举，$i-j$表示第二操作数下标（该下标同样需要满足范围0~31），所以需要满足条件$i-j\geq 0$且$i-j\leq 31$。

6 行列交织功能设计

针对行列交织输入的16×16的 16 位矩阵，存储时从 0 号存储单元开始，以行下标为第一关键字，列下标为第二关键字升序存储，16 位元素则存储在对应 32 位存储单元的低 16 位。

执行行列交织时，先依次枚举行下标，在枚举列下标，同时为避免重复选取以及正对角线上（行、列下标相等）的元素（即只选取矩阵上三角部分），并通过下标计算待交换的两数存储单元地址，其地址计算算法如下：

对于行下标为i，列下标为j的元素，由于本处理器设计按字编址，其存储单元地址为$16\times i+j$；同理，其对应交换元素行下标为j，列下标为i，存储单元地址为$16\times j+i$。

进行交换时只用分别将两存储单元储存结果取入寄存器在交叉存储到对方地址的存储单元即可。

由于篇幅限制，部分汇编代码展示如下：

```
addi $t0,$0,0
lw $t1,1($t0)
lw $t2,16($t0)
sw $t1,16($t0)
sw $t2,1($t0)
lw $t1,2($t0)
lw $t2,32($t0)
sw $t1,32($t0)
sw $t2,2($t0)
..... (此处省略)
```

```
lw $t1,223($t0)
lw $t2,253($t0)
sw $t1,253($t0)
sw $t2,223($t0)
lw $t1,239($t0)
lw $t2,254($t0)
sw $t1,254($t0)
sw $t2,239($t0)
```

同样的，为准确、高效生成执行卷积功能的汇编代码，使用高级程序语言 C++进行编写，相应代码入下：

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
	freopen("hangliejiaozhi.txt","w",stdout);
	printf("addi $t0,$0,0\n");
	for(int i=0;i<=15;i++)
	{
		for(int j=i+1;j<=15;j++)
		{
			printf("lw $t1,%d($t0)\n",i*16+j);
			printf("lw $t2,%d($t0)\n",j*16+i);
			printf("sw $t1,%d($t0)\n",j*16+i);
			printf("sw $t2,%d($t0)\n",i*16+j);
		}
	}
	return 0;
}
```

相关变量与地址计算如上文所述。

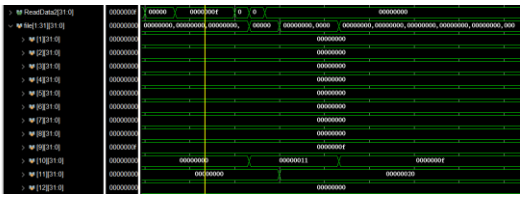
7 仿真

通过 Vivado 软件对编写的 MIPS 流水线处理器进行仿真后运行下图所示汇编指令：

```
1  addi $t1,$0,15
2  addi $t2,$0,17
3  add  $t3,$t2,$t1
4  sw $t1,0($0)
5  lw $t2,0($0)
6
7
```

上述代码执行后，寄存器 t3（11 号寄存器）存储值应为 32，寄存器 t2（10 号寄存器）存储值值应为 15，寄存器 t1（9 号寄存器）存储值值应为 15。

将代码存入指令存储器的 rom 中运行仿真进行测试，仿真情况如下图所示：



经核对后可知运行正确。

另外，针对实验设计的卷积运算和行列交织功能利用测试数据同样进行仿真测试，相应运行结果如下：

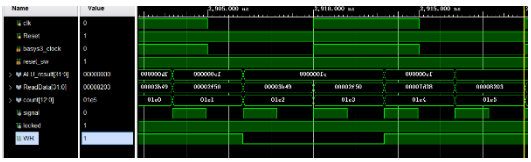
卷积：

readdata 为存储器写入端信号，注意到当且仅当存储器写使能 WR 为 1 时，READdata 的信号才有效（即 WR 表示写入存储器的卷积结果），经核对后可知运行正确，但由于篇幅限制此处仅展示两个数据。


```
1111110001111111,fc7f
0000100100001011,090b
```

行列交织：

经核对后可知运行正确，此处同样仅展示部分数据。



```
1111011010001100,
1000101111111011,
1000010101101001,
0011101101001001,3b49
0111110100101000,7d28
1001111111000010,
```

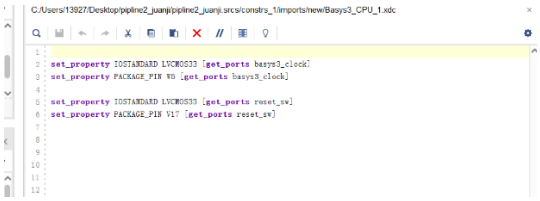
此处相应仿真结果细节将以附件形式提交。

8 综合、上板

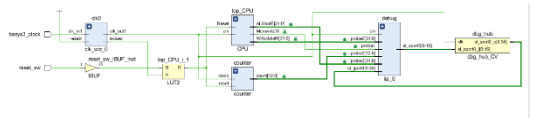
综合时选取 stage2 的卷积指令（共 4200 余条）和行列交织指令（480 余条）进行上版测试，上板顶层文件有两个输入主板时钟 clock 和复位信号 reset_sw 其中 basys3_clock 信号连接板子上 W5 100MHZ 时钟端口，reset_sw 信号连接板子上 V17 端口，通过开关对 CPU 进行复位。下面是顶层文件，本次实验设计选取的 CPU 的存储器写入端口和存储器写使能信号以及 ALU 结果作为 CPU 输出信号，当存储器写使能信号为 1 时，代表存储器写入端口的数据为有效数据，即卷积结果，ALUresult 用于 stage1 测试时的结果输出，CPU 的这三个输出都将作为探针被 ILA 核探测到，注意到，设计中还使用了分频器 IP clock_wiz,分频器将输出的时钟信号作为 CPU 的时钟信号，同时还设计了一个计数器用于测算输出所有数据消耗的时钟周期，卷积指令执行时所选取的分频器输出的时钟频率为 130MHZ，行列交织指令执行时选取的频率输出为 300MHZ

```
module onboard_CPU(
    input basys3_clock,
    input reset_sw

    );
  wire [31:0] ALU_result;

  wire [31:0] ReadData;
 wire [12:0]count;
  wire signal,locked,WR;

    CPU top_CPU(
        .clk(signal),
        .Reset(reset_sw&locked),
         .WritedataM(ReadData),
        .ALUoutE(ALU_result[31:0]),
        . MemwriteM(WR)

    );
  counter counter(.clock(signal),.reset(reset_sw&locked),.count(count));
 clk_wiz_0 clk0(.clk_out1(signal),.clk_in1(basys3_clock),.resetn(1),.locked(locked));
ila_0 debug (
    .clk(signal), // input wire clk

    .probe0(ReadData), // input wire [31:0]  probe0
    .probe1(WR), // input wire [0:0]  probe2
    .probe2(count), // input wire [12:0]  probe2
    .probe3(ALU_result) // input wire [7:0]  probe1
```

Constraint 文件如下图所示：



综合后的电路如下：

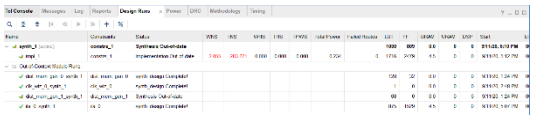


进行卷积测试的 MCU 运行后的 LUT 和 FF 消耗如下图所示：

时钟频率报告如下图所示：

Name	Waveform	Period (ns)	Frequency (MHz)
basys3_clock	{0.000 5.000}	10.000	100.000
clk_out1_clk_wiz_0	{0.000 3.846}	7.692	130.000
clkfbout_clk_wiz_0	{0.000 25.000}	50.000	20.000
dbg_hub/inst/BSCANID.u_xsdbm_id/SWT	{0.000 16.500}	33.000	30.303

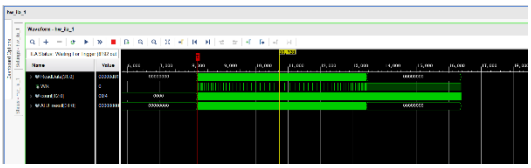
此外，在运行行列交织时，由于没有用到乘法器，本实验设计的 MCU 的频率可以达到 300MHZ，用于行列交织测试的 MCU 消耗和频率如下图所示：

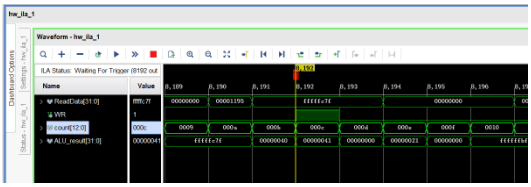


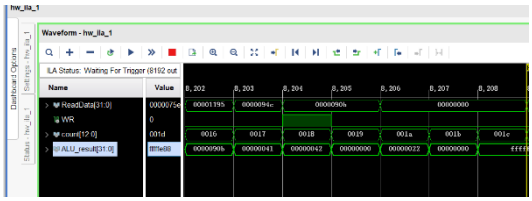
Name	Waveform	Period (ns)	Frequency (MHz)
basys3_clock	{0.000 5.000}	10.000	100.000
clk_out1_clk_wiz_0	{0.000 1.667}	3.333	300.000
clkfbout_clk_wiz_0	{0.000 5.000}	10.000	100.000
dbg_hub/inst/BSCANID.u_xsdbm_id/SWT	{0.000 16.500}	33.000	30.303

生成 bit 流，烧录到 FPGA 开发板上，利用 ILA 核探针抓取信号，设置为自动 trigger，拨动板子上 sw0 开关时，MCU 复位，开始运行程序，ILA 自动抓取信号，拨动开关 SW0MCU。

运行卷积指令的抓取信号如下：WR 为 1 时 READDATA 上的数据有效。

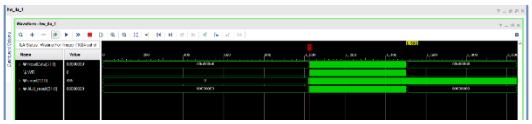


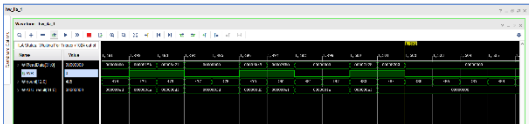




由上述结果可知，上板信号结果正确。卷积用时 5245 个周期，消耗时间 5245/130M=40.3 微秒。

运行行列交织指令的抓取信号如下：WR 为 1 时 READDATA 上的数据有效





比对数据后，结果均正确，消耗 485 个时钟周期，用时 485/300M=1.62 微秒。

如上图所示，行列交织和卷积的上板信号输出均与仿真结果一致，信号抓取正常。

卷积频率达到了 130MHZ，消耗资源为 LUT(3232)+FF(2813)=6045,消耗时间为 40.3 微秒，速度为 1/40.3（卷积/微秒），效率为 6045*40.3=242613.5。

行列交织频率达到了 300MHZ，消耗资源为 LUT(1716)+FF(2479)=4195,消耗时间为 1.61 微秒，速度为 1/1.61（行列交织/微秒）效率为 4195*1.61=6753.95。

可见行列的效率和速度要远高于卷积，卷积要进行的运算多，且消耗乘法器的资源，使得频率难以提升。

9 总结

本次实验我们完成了一次完整的 MPIS 设计过程，用到了本学期课程所学组合时序逻辑、硬件编程语言、模块设计编写、处理器通路设计、指令集设计、流水线优化以及板上工程操作等知识，充分串联、实践课程所学，让我们对数字逻辑有了更深入的理解，对完成的工程实践流程有了更直观的体验。