实验报告：5 段流水 CPU 设计

姓名：江玥璠

学号：518021910550

实验目的

1. 理解计算机指令流水线的协调工作原理，初步掌握流水线的设计和实现原理。
2. 深刻理解流水线寄存器在流水线实现中所起的重要作用。
3. 理解和掌握流水段的划分、设计原理及其实现方法原理。
4. 掌握运算器、寄存器堆、存储器、控制器在流水工作方式下，有别于实验一的设计和实现方法。
5. 掌握流水方式下，通过 I/O 端口与外部设备进行信息交互的方法。

实验内容和任务

1. 采用 Verilog 在 Quartus Ⅱ 中实现基本的具有 20 条 MIPS 指令的 5 段流水 CPU 设计。
2. 利用实验提供的标准测试程序代码，完成仿真测试。
3. 采用 I/O 统一编址方式，即将输入输出的 I/O 地址空间，作为数据存取空间的一部分，实现 CPU 与外部设备的输入输出端口设计。实验中可采用高端地址。
4. 利用设计的 I/O 端口，通过 `lw` 指令，输入 DE2 实验板上的按键等输入设备信息。即将外部设备状态，读到 CPU 内部寄存器。
5. 利用设计的 I/O 端口，通过 `sw` 指令，输出对 DE2 实验板上的 LED 灯等输出设备的控制信号（或数据信息）。即将对外部设备的控制数据，从 CPU 内部的寄存器，写入到外部设备的相应控制寄存器（或可直接连接至外部设备的控制输入信号）。
6. 利用自己编写的程序代码，在自己设计的 CPU 上，实现对板载输入开关或按键的状态输入，并将判别或处理结果，利用板载 LED 灯或 7 段 LED 数码管显示出来。
7. 例如，将一路 4bit 二进制输入与另一路 4bit 二进制输入相加，利用两组分别 2 个 LED 数码管以 10 进制形式显示“被加数”和“加数”，另外一组 LED 数码管以 10 进制形式显示“和”等。（具体任务形式不做严格规定，同学可自由创意）。
8. 在实现 MIPS 基本 20 条指令的基础上，实现 Y86 相应的基本指令。
9. 在实验报告中，汇报自己的设计思想和方法；并以汇编语言的形式，提供以上两种指令集（MIPS 和 Y86）应用功能的程序设计代码，并提供程序主要流程图。

实验仪器

- 硬件：DE1-SoC 实验板
- 软件：Altera Quartus II 13.1、 Altera ModelSim 10.1d、MIPS 指令集汇编器

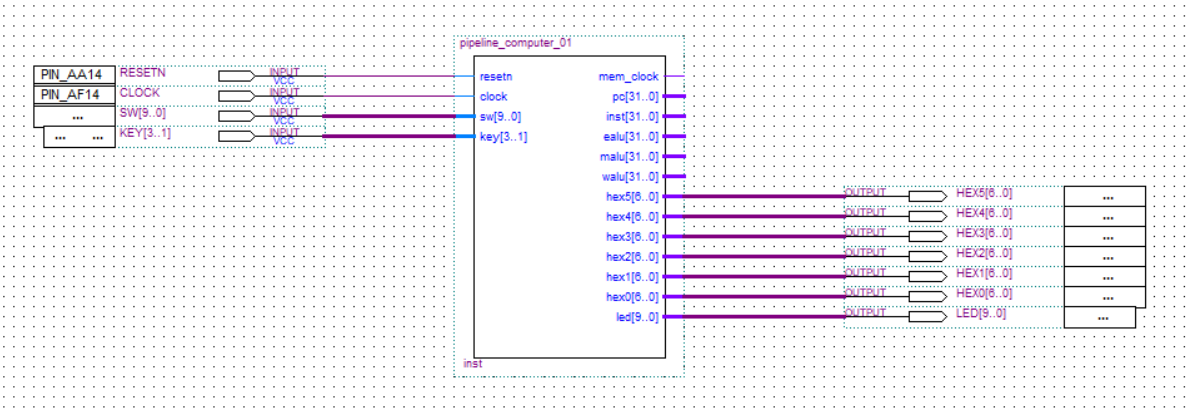
实验详情

一、设计思路

大部分的模块是基于单周期 CPU 的实验代码改进的，主要参考的是李亚民老师的《计算机原理与设计——Verilog HDL 版》一书的第 8 章：流水线 CPU 及其 Verilog HDL 设计中提供的代码。

主要的设计思路是在单周期 CPU 的基础上，将 CPU 按照指令执行的阶段分成取指、译指、执行、访存、写回五级流水线，每一级承担一条指令完整执行的部分功能，在每一级流水之间增加流水线寄存器用于储存 CPU 中的信号量。具体的实现将在后续部分进行介绍。

二、顶层设计



输入和单周期 CPU 类似，`RESETN` 为重置按钮，用 `KEY0` 代替，`CLOCK` 为板载 50MHz 时钟，`SW` 为开发板的 10 个开关，`KEY` 为剩下的 3 个按钮；

主模块的输出中有一部分没有分配管脚，这些输出是李亚民老师的代码带有的，没有做改动，可以在仿真的时候方便 debug；

`HEX5-0` 对应开发板上的 6 个 七段 LED 数码管，`LED` 对应 10 个 LED 发光二极管。

三 、部分源代码设计

1. `pipeline_computer_01` 主模块

```
//定义顶层模块 pipelined_computer，作为工程文件的顶层入口，如图 1-1 建立工程时指定。
module pipeline_computer_01 (resetn,clock,mem_clock,pc,inst,ealu,malu,walu,
                             sw, key, hex5, hex4, hex3, hex2, hex1,
hex0, led);
    input           resetn, clock;
    //定义整个计算机 module 和外界交互的输入信号，包括复位信号 resetn、时钟信号
clock、
    //以及一个和 clock 同频率但反相的 mem_clock 信号。mem_clock 用于指令同步 ROM 和
    //数据同步 RAM 使用，其波形需要有别于实验一。
    //这些信号可以用作仿真验证时的输出观察信号。
    input [9:0]     sw;
    input [3:1]     key;

    output [31:0]   pc,inst,ealu,malu,walu;
    //模块用于仿真输出的观察信号。缺省为 wire 型。
    output [6:0]    hex5,hex4,hex3,hex2,hex1,hex0;
    output [9:0]    led;
    output          mem_clock;

    wire [31:0]     pc,bpc,jpc,npc,pc4,ins,inst;
    //模块间互联传递数据或控制信息的信号线,均为 32 位宽信号。IF 取指令阶段。
    wire [31:0]     dpc4,da,db,dimm;
    //模块间互联传递数据或控制信息的信号线,均为 32 位宽信号。ID 指令译码阶段。
    wire [31:0]     epc4,ea,eb,eimm;
    //模块间互联传递数据或控制信息的信号线,均为 32 位宽信号。EXE 指令运算阶段。
    wire [31:0]     mb,mmo;
    //模块间互联传递数据或控制信息的信号线,均为 32 位宽信号。MEM 访问数据阶段。
    wire [31:0]     wmo,wdi;
    //模块间互联传递数据或控制信息的信号线,均为 32 位宽信号。WB 回写寄存器阶段。
    wire [4:0]      drn,ern0,ern,mrn,wrn;
    //模块间互联，通过流水线寄存器传递结果寄存器号的信号线，寄存器号（32 个）为 5bit。
    wire [3:0]      daluc,ealuc;
```

```
    //ID 阶段向 EXE 阶段通过流水线寄存器传递的 aluc 控制信号，4bit。
    wire [1:0]      pcsource;
    //CU 模块向 IF 阶段模块传递的 PC 选择信号，2bit。
    wire            wpcir;
    //CU 模块发出的控制流水线停顿的控制信号，使 PC 和 IF/ID 流水线寄存器保持不变。
    wire            dwreg,dm2reg,dwmem,daluimm,dshift,djal; // id stage
    //ID 阶段产生，需往后续流水级传播的信号。
    wire            ewreg,em2reg,ewmem,ealuimm,eshift,ejal; // exe stage
    //来自于 ID/EXE 流水线寄存器，EXE 阶段使用，或需要往后续流水级传播的信号。
    wire            mwreg,mm2reg,mwmem; // mem stage
    //来自于 EXE/MEM 流水线寄存器，MEM 阶段使用，或需要往后续流水级传播的信号。
    wire            wwreg,wm2reg; // wb stage
    //来自于 MEM/WB 流水线寄存器，WB 阶段使用的信号。

    //实验中可采用系统 clock 的反相信号作为 mem_clock（亦即 rom_clock），
    assign mem_clock = ~clock;

    pipepc prog_cnt ( npc,wpcir,clock,resetn,pc );
    //程序计数器模块，是最前面一级 IF 流水段的输入。

    pipeif if_stage ( pcsource,pc,bpc,da,jpc,npc,pc4,ins,mem_clock ); // IF
stage
    //IF 取指令模块，注意其中包含的指令同步 ROM 存储器的同步信号，
    //即输入给该模块的 mem_clock 信号，模块内定义为 rom_clk。// 注意 mem_clock。
    //实验中可采用系统 clock 的反相信号作为 mem_clock（亦即 rom_clock），
    //即留给信号半个节拍的传输时间。

    pipeir inst_reg ( pc4,ins,wpcir,clock,resetn,dpc4,inst ); // IF/ID 流水线
寄存器
    //IF/ID 流水线寄存器模块，起承接 IF 阶段和 ID 阶段的流水任务。
    //在 clock 上升沿时，将 IF 阶段需传递给 ID 阶段的信息，锁存在 IF/ID 流水线寄存器
    //中，并呈现在 ID 阶段。

    pipeid id_stage ( mwreg,mrn,ern,ewreg,em2reg,mm2reg,dpc4,inst,
    wrn,wdi,ealu,malu,mmo,wwreg,clock,resetn,
    bpc,jpc,pcsource,wpcir,dwreg,dm2reg,dwmem,daluc,
    daluimm,da,db,dimm,drn,dshift,djal ); // ID stage
    //ID 指令译码模块。注意其中包含控制器 CU、寄存器堆、及多个多路器等。
    //其中的寄存器堆，会在系统 clock 的下沿进行寄存器写入，也就是给信号从 WB 阶段
    //传输过来留有半个 clock 的延迟时间，亦即确保信号稳定。
    //该阶段 CU 产生的、要传播到流水线后级的信号较多。

    pipedereg de_reg (
dwreg,dm2reg,dwmem,daluc,daluimm,da,db,dimm,drn,dshift,
    djal,dpc4,clock,resetn,ewreg,em2reg,ewmem,ealuc,ealuimm,
    ea,eb,eimm,ern0,eshift,ejal,epc4 ); // ID/EXE 流水线寄存器
    //ID/EXE 流水线寄存器模块，起承接 ID 阶段和 EXE 阶段的流水任务。
    //在 clock 上升沿时，将 ID 阶段需传递给 EXE 阶段的信息，锁存在 ID/EXE 流水线
    //寄存器中，并呈现在 EXE 阶段。

    pipeexe exe_stage (
ealuc,ealuimm,ea,eb,eimm,eshift,ern0,epc4,ejal,ern,ealu ); // EXE stage
    //EXE 运算模块。其中包含 ALU 及多个多路器等。

    pipeemreg em_reg ( ewreg,em2reg,ewmem,ealu,eb,ern,clock,resetn,
    mwreg,mm2reg,mwmem,malu,mb,mrn); // EXE/MEM 流水线寄存器
    //EXE/MEM 流水线寄存器模块，起承接 EXE 阶段和 MEM 阶段的流水任务。
    //在 clock 上升沿时，将 EXE 阶段需传递给 MEM 阶段的信息，锁存在 EXE/MEM
```

```
    //流水线寄存器中，并呈现在 MEM 阶段。

    pipedmem mem_stage ( malu,mb,mmo,mwmem,mem_clock,resetn,
    sw, key, hex5, hex4, hex3, hex2, hex1, hex0, led ); // MEM stage
    //MEM 数据存取模块。其中包含对数据同步 RAM 的读写访问。// 注意 mem_clock。
    //输入给该同步 RAM 的 mem_clock 信号，模块内定义为 ram_clk。
    //实验中可采用系统 clock 的反相信号作为 mem_clock 信号（亦即 ram_clk），
    //即留给信号半个节拍的传输时间，然后在 mem_clock 上沿时，读输出、或写输入。

    pipemwreg mw_reg ( mwreg,mm2reg,mmo,malu,mrn,clock,resetn,
    wwreg,wm2reg,wmo,walu,wrn); // MEM/WB 流水线寄存器
    //MEM/WB 流水线寄存器模块，起承接 MEM 阶段和 WB 阶段的流水任务。
    //在 clock 上升沿时，将 MEM 阶段需传递给 WB 阶段的信息，锁存在 MEM/WB
    //流水线寄存器中，并呈现在 WB 阶段。

    mux2x32 wb_stage ( walu,wmo,wm2reg,wdi ); // WB stage
    //WB 写回阶段模块。事实上，从设计原理图上可以看出，该阶段的逻辑功能部件只
    //包含一个多路器，所以可以仅用一个多路器的实例即可实现该部分。
    //当然，如果专门写一个完整的模块也是很好的。
endmodule
```

这一部分的代码基本上是按照提供的代码写的，改动很小，每一个实例化的子组件都有详细的注释，后续也会详细介绍。

因为引入了 I/O 映射的原因，输入输出的参数与原本的代码有所区别，这一部分参数仅仅用在了数据存储 `pipedmem` 模块中。

与单周期 CPU 需要两个频率的时钟不同，本实验输入的时钟只有板载时钟 `clock` ，用于访存操作的时钟信号 `mem_clock` 由 `clock` 取反获得。

2. `pipepc` PC 流水线寄存器，位于取值阶段之前

```
// PC 寄存器，IF 阶段的寄存器
module pipepc(npc, wpc, clk, clrn, pc);
    input [31:0]    npc;
    input           wpc, clk, clrn;
    output [31:0]   pc;
    dffe32pc program_counter (npc,clk,clrn,wpc,pc);
endmodule
```

逻辑很简单，根据 `wpc` `clk` `clrn` 的输入，将 `npc` 的值输入到 `pc`

```
// 32bit D flip-flop with an enable signal for PC.
module dffe32pc(d, clk, clrn, e, q);
    input      [31:0] d;
    input             clk, clrn, e;
    output reg [31:0] q;

    always @(negedge clrn or posedge clk)
        if (clrn == 0) begin
            q <= -4; // Make sure instruction at address 0 will be properly
executed.
        end else begin
            if (e == 1)
                q <= d;
        end
endmodule
```

为了正常的执行代码，额外定义了一个 D 触发器专门用于选取 PC 的值，按照李亚民老师提供的代码似乎不能正常地执行第一条指令。

3. `pipeif` 取指模块

```
// IF 阶段的组合电路
module pipeif (pcsource, pc, bpc, rpc, jpc, npc, pc4, ins, imem_clk);
    input [31:0]    pc,bpc,rpc,jpc;
    input [1:0]     pcsource;
    input           imem_clk;
    output [31:0]   npc,pc4,ins;

    assign pc4 = pc + 4;

    mux4x32 next_pc (pc4,bpc,rpc,jpc,pcsource,npc);
    pipeimem inst_mem (pc,ins,imem_clk);
endmodule
```

取指阶段默认生成一个 PC + 4 的值输入到 `next_pc` 的多路选择器，其他 PC 值由 CU 传回；

`pipeimem` 子组件内有一个 `rom_1port` 组件，对应指令存储器，是利用 Altera 自带的 MegaFunction 实现的，与单周期 CPU 中一样。从中对应的 PC 地址读取一条指令执行，此处的 `imem_clk` 即为顶层设计中的 `mem_clock` 。

4. `pipeir` IF/ID 流水线寄存器模块，起承接 IF 阶段和 ID 阶段的流水任务。

```
// IR 寄存器及 PC+4 寄存器
module pipeir (pc4,ins,wir,clk,clrn,dpc4,inst);
    input [31:0]    pc4,ins;
    input               wir,clk,clrn;
    output [31:0]   dpc4,inst;
    dffe32 pc_plus4 (pc4,clk,clrn,wir,dpc4);
    dffe32 instruction (ins,clk,clrn,wir,inst);
endmodule
```

主要传递的值是 PC + 4 的值 `dpc4` 和刚刚取出的指令 `inst` ， `wir` 信号量是整体电路设计图中的 `wpcir` 信号，是为了在发生 Data Hazards 的时候可能需要暂停流水线，控制不修改 PC 和 IR。

5. `pipeid` 译指模块

```
// ID 级组合电路
module pipeid (mwreg, mrn, ern, ewreg, em2reg, mm2reg, dpc4, inst, wrn,
                    wdi, ealu, malu, mmo, wwreg, clk, clrn, bpc, jpc,
pcsource,
                    nostall, wreg, m2reg, wmem, aluc, aluimm, da, db, dimm,
rn, shift, jal);
    input [31:0]    dpc4,inst,wdi,ealu,malu,mmo;
    input [4:0]     ern,mrn,wrn;
    input           mwreg,ewreg,em2reg,mm2reg,wwreg;
    input               clk,clrn;
    output [31:0]   bpc,jpc,da,db,dimm;
    output [4:0]    rn;
    output [3:0]    aluc;
    output [1:0]    pcsource;
    output          nostall,wreg,m2reg,wmem,aluimm,shift,jal;

    wire [5:0]      op,func;
```

```
    wire [4:0]      rs,rt,rd;
    wire [31:0]     qa,qb,br_offset;
    wire [15:0]     ext16,imm;
    wire [1:0]      fwda,fwdb;
    wire                regrt,sext,rsrtequ,e;

    assign func = inst[5:0];
    assign op   = inst[31:26];
    assign rs   = inst[25:21];
    assign rt   = inst[20:16];
    assign rd   = inst[15:11];
    assign imm  = inst[15:0];
    assign jpc  = {dpc4[31:28],inst[25:0],2'b00};

    pipeidcu cu (mwreg,mrn,ern,ewreg,em2reg,mm2reg,rsrtequ,func,
                    op,rs,rt,wreg,m2reg,wmem,aluc,regrt,aluimm,
                    fwda,fwdb,nostall,sext,pcsource,shift,jal);

    regfile rf (rs,rt,wdi,wrn,wwreg,~clk,clrn,qa,qb);
    mux2x5 des_reg_no (rd,rt,regrt,rn);
    mux4x32 alu_a (qa,ealu,malu,mmo,fwda,da);
    mux4x32 alu_b (qb,ealu,malu,mmo,fwdb,db);

    assign rsrtequ = (da == db); // rsrtequ = (a == b)
    assign e = sext & inst[15];
    assign ext16 = {16{e}};
    assign dimm = {ext16,imm};
    assign br_offset = {dimm[29:0],2'b00};
    assign bpc = dpc4 + br_offset;
endmodule
```

这部分代码主要是定义了一些信号量，实例化了 `cu` `regfile` 等子模块和几个多路选择器。和单周期 CPU 中的逻辑有一点区别。

- `rsrtequ` 是判断 `rs` 和 `rt` 是否相等的标记，相当于单周期实验中的 `z`；
- `fwda` `fwdb` 是为了避免 Data Hazards 引入的 forwarding 的变量值，因此引入了多路选择器来选择实际传下去的 `da` `db` 的值；
- `bpc4` 是 `bne` `beq` 指令的跳转地址，需要回传给取指阶段的 PC 地址多路选择器；
- `regfile` 的写入在时钟的下降沿（时钟信号的输入为 `~clk`），这样可以正常读出同一个周期内写回阶段写回的值，因此加了一个取反（非门）

`cu` 的实现没有明显的改变，但是加入了 `nostall` 信号量的判定，用来判断是否有 Hazards 需要插入气泡或采取其他措施；加入了 `fwda` 和 `fwdb` 的判定，根据具体的指令，按照就近原则选择合适的 Forwarding 的新值作为 `da` 和 `db` 的值，部分代码如下：

```
// CU
module pipeidcu (mwreg,mrn,ern,ewreg,em2reg,mm2reg,rsrtequ,func,op,rs,rt,

 wreg,m2reg,wmem,aluc,regrt,aluimm,fwda,fwdb,nostall,sext,
                      pcsource,shift,jal);
    ...
    output [1:0]    fwda,fwdb;
    output          nostall;
    ...
    reg [1:0]   fwda,fwdb;
    ...
```

```
    assign nostall = ~(ewreg & em2reg & (ern != 0) & (i_rs & (ern == rs) |
i_rt & (ern == rt)));
    ...
    always @ (ewreg or mwreg or ern or mrn or em2reg or mm2reg or rs or rt)
begin
        fwda = 2'b00; // default forward a: no hazards
        if (ewreg & (ern != 0) & (ern == rs) & ~em2reg) begin
            fwda = 2'b01; // select exe_alu
        end else begin
            if (mwreg & (mrn != 0) & (mrn == rs) & ~mm2reg) begin
                fwda = 2'b10; // select mem_alu
            end else begin
                if (mwreg & (mrn != 0) & (mrn == rs) & mm2reg) begin
                    fwda = 2'b11; // select mem_lw
                end
            end
        end
        fwdb = 2'b00; // default forward b: no hazards
        if (ewreg & (ern != 0) & (ern == rt) & ~em2reg) begin
            fwdb = 2'b01; // select exe_alu
        end else begin
            if (mwreg & (mrn != 0) & (mrn == rt) & ~mm2reg) begin
                fwdb = 2'b10; // select mem_alu
            end else begin
                if (mwreg & (mrn != 0) & (mrn == rt) & mm2reg) begin
                    fwdb = 2'b11; // select mem_lw
                end
            end
        end
    end
    ...
    // 写寄存器信号
    assign wreg = (i_add | i_sub | i_and | i_or   | i_xor  |
                       i_sll | i_srl | i_sra | i_addi | i_andi |
                       i_ori | i_xori | i_lw | i_lui  | i_jal) & nostall;
    ...
    assign wmem    = i_sw & nostall;
    ...
    assign pcsource[0] = ( i_beq & rsrtequ ) | (i_bne & ~rsrtequ) | i_j |
i_jal ;
endmodule
```

列出来的代码都是与单周期 CPU 的代码相比变化的部分。中间 `always` 的部分相当于是实现了 `da` `db` 的多路选择器。涉及 `nostall` 的部分主要的逻辑是当需要 stall 时，禁止写寄存器和写内存。

6. `pipedereg` ID/EXE 流水线寄存器模块，起承接 ID 阶段和 EXE 阶段的流水任务。

```
// ID 和 EXE 之间的流水线寄存器
module pipedereg (dwreg, dm2reg, dwmem, daluc, daluimm, da, db, dimm, drn,
                       dshift, djal, dpc4, clk, clrn, ewreg, em2reg, ewmem,
                       ealuc, ealuimm, ea, eb, eimm, ern, eshift, ejal,
epc4);
    input [31:0]    da,db,dimm,dpc4;
    input [4:0]     drn;
    input [3:0]     daluc;
    input           dwreg,dm2reg,dwmem,daluimm,dshift,djal;
```

```
    input           clk,clrn;
    output [31:0]   ea,eb,eimm,epc4;
    output [4:0]    ern;
    output [3:0]    ealuc;
    output          ewreg,em2reg,ewmem,ealuimm,eshift,ejal;
    reg [31:0]      ea,eb,eimm,epc4;
    reg [4:0]       ern;
    reg [3:0]       ealuc;
    reg                 ewreg,em2reg,ewmem,ealuimm,eshift,ejal;

    always @ (negedge clrn or posedge clk) begin
        if (clrn == 0) begin
            ewreg <= 0;
            em2reg <= 0;
            ewmem <= 0;
            ealuc <= 0;
            ealuimm <= 0;
            ea <= 0;
            eb <= 0;
            eimm <= 0;
            ern <= 0;
            eshift <= 0;
            ejal <= 0;
            epc4 <= 0;
        end else begin
            ewreg <= dwreg;
            em2reg <= dm2reg;
            ewmem <= dwmem;
            ealuc <= daluc;
            ealuimm <= daluimm;
            ea <= da;
            eb <= db;
            eimm <= dimm;
            ern <= drn;
            eshift <= dshift;
            ejal <= djal;
            epc4 <= dpc4;
        end
    end
endmodule
```

这部分流水线寄存器和后续 EXE/MEM 以及 MEM/WB 的流水线寄存器的整体逻辑都是相同的，即将上一阶段属于上一条指令的信号量沿着流水线传递下去，实现指令沿流水线流动的目的。逻辑很简单就不再详细展开，后续的两个流水线寄存器也省略。

7. `pipeexe` 运算模块

```
// EXE 级的组合电路
module pipeexe (ealuc, ealuimm, ea, eb, eimm, eshift, ern0, epc4, ejal, ern,
ealu);
    input [31:0]    ea,eb,eimm,epc4;
    input [4:0]     ern0;
    input [3:0]     ealuc;
    input           ealuimm,eshift,ejal;
    output [31:0]   ealu;
    output [4:0]    ern;
```

```
    wire [31:0]      alua,alub,sa,ealu0,epc8;

    assign sa = {27'b0, eimm[10:6]};    // shift amount

    mux2x32 alu_ina (ea,sa,eshift,alua);
    mux2x32 alu_inb (eb,eimm,ealuimm,alub);
    mux2x32 save_pc8 (ealu0,epc8,ejal,ealu);

    assign epc8 = epc4 + 4;
    assign ern = ern0 | {5{ejal}};

    alu al_unit (alua,alub,ealuc,ealu0);
endmodule
```

大部分的逻辑是根据 `eshift` 和 `ealuimm` 信号，调用 `alu` 模块进行合适的计算，逻辑并不复杂，且 `alu` 模块与单周期 CPU 的代码相比也没有什么变化。

需要解释一下的是此处的 `epc8` 信号：

因为 `jal` 是 MIPS 定义的延迟转移指令之一，也就是说，`jal` 的下一条指令一定是被执行的延迟槽，也就是说，实际上在跳转之前，`jal` 的下一条指令 `epc4` 处，就已经执行了，因此 `jal` 所保存的返回地址应该是再下一条指令 `epc8` 。此外，`jal` 指令还需要把写寄存器值 `ern0` 设置为第 31 个寄存器。

8. `pipeemreg` EXE/MEM 流水线寄存器模块，起承接 EXE 阶段和 MEM 阶段的流水任务

   略。

9. `pipedmem` 访存阶段

```
// 数据存储器，初始化文件为 pipedmem.mif
module pipedmem (addr, datain, dataout, we, mem_clk, clrn,
                    sw, key, hex5, hex4, hex3, hex2, hex1, hex0, led);
    input [31:0]    addr,datain;
    input [9:0]     sw;
    input [3:1]     key;
    input           we,mem_clk,clrn;
    output reg [31:0]   dataout;
    output reg [6:0]    hex5,hex4,hex3,hex2,hex1,hex0;
    output reg [9:0]    led;

   wire         write_enable;
    wire [31:0]     memdata;

    assign write_enable = we & (addr[31:8] != 24'hffffff);

   ram_1port  dram(addr[6:2],mem_clk,datain,write_enable,memdata);

    // I/O
    always @(posedge mem_clk or negedge clrn) begin
        if (!clrn) begin // 重置所有 I/O 的值
            hex0 <= 7'b1111111;
            hex1 <= 7'b1111111;
            hex2 <= 7'b1111111;
            hex3 <= 7'b1111111;
            hex4 <= 7'b1111111;
            hex5 <= 7'b1111111;
            led <= 10'b0000000000;
```

```
        end else if (we) begin // 当 we 为 1 时触发写内存的操作，采用高地址映射 I/O
，此处为输出
            case (addr)
                32'hffffff20: hex0 <= datain[6:0];
                32'hffffff30: hex1 <= datain[6:0];
                32'hffffff40: hex2 <= datain[6:0];
                32'hffffff50: hex3 <= datain[6:0];
                32'hffffff60: hex4 <= datain[6:0];
                32'hffffff70: hex5 <= datain[6:0];
                32'hffffff80: led <= datain[9:0];
            endcase
        end
    end

    always @(*) begin // 读取内存的操作，采用高地址映射 I/O ，此处为输入
        case (addr)
            32'hffffff00: dataout <= {22'b0, sw};
            32'hffffff10: dataout <= {28'b0, key, 1'b1};
            default: dataout <= memdata;
        endcase
    end
endmodule
```

这一部分代码和单周期 CPU 的数据内存 dmem 部分相似，基本上没有改动，也是基于 Altera 的 MegaFunction 定义了 `ram_1port` 模块进行数据的访存操作，并采用高端地址映射的方式加入了 I/O 的内存映射。

10. `pipemwreg` MEM/WB 流水线寄存器模块，起承接 MEM 阶段和 WB 阶段的流水任务

    略。

11. WB 写回阶段：

    直接采用了 `mux2x32` 的多路选择器直接实现。

```
mux2x32 wb_stage ( walu,wmo,wm2reg,wdi );
```

12. `sevenseg.mif` 数据内存布局文件

```
DEPTH = 32;                 % Memory depth and width are required %
WIDTH = 32;                 % Enter a decimal number %
ADDRESS_RADIX = HEX;    % Address and value radixes are optional %
DATA_RADIX = HEX;       % Enter BIN, DEC, HEX, or OCT; unless %
CONTENT                         % otherwise specified, radixes = HEX %
BEGIN
0: 00000040;                % sevenseg code for '0' %
1: 00000079;                % sevenseg code for '1' %
2: 00000024;                % sevenseg code for '2' %
3: 00000030;                % sevenseg code for '3' %
4: 00000019;                % sevenseg code for '4' %
5: 00000012;                % sevenseg code for '5' %
6: 00000002;                % sevenseg code for '6' %
7: 00000078;                % sevenseg code for '7' %
8: 00000000;                % sevenseg code for '8' %
9: 00000010;                % sevenseg code for '9' %
END ;
```

与单周期 CPU 实验中一样，将七段 LED 数码管的数字编码预存到数据内存中，用于计算器程序的数字显示。

13. `calculator.mif` 计算器程序

```
DEPTH = 128;           % Memory depth and width are required %
WIDTH = 32;            % Enter a decimal number %
ADDRESS_RADIX = HEX;   % Address and value radixes are optional %
DATA_RADIX = HEX;      % Enter BIN, DEC, HEX, or OCT; unless %
                       % otherwise specified, radixes = HEX %
CONTENT
BEGIN
[0..7F] : 00000000;    % Range--Every address from 0 to 7F = 00000000 %

 0 : 0800003a;         % (000) start:      j main_loop             # enter
main loop, delay slot underneath                          %
 1 : 3c070000;         % (004)             lui $7, 0               # $7
stores op (0->add (default), 1->sub, 2->xor)              %
 2 : 001ef080;         % (008) sevenseg:   sll $30, $30, 2         #
calculate sevenseg table item addr to load                    %
 3 : 03e00008;         % (00c)             jr $ra                  # return,
delay slot underneath                                     %
 4 : 8fdd0000;         % (010)             lw $29, 0($30)          # load
sevenseg code of arg($30) from data memory to $29         %
 5 : 0000e820;         % (014) split:      add $29, $0, $0         # $29
stores tens digit                                         %
 6 : 23defff6;         % (018) split_loop: addi $30, $30, -10      #
decrement arg($30) by 10                                  %
 7 : 001ee7c3;         % (01c)             sra $28, $30, 31        # extend
sign digit of the result                                  %
 8 : 17800003;         % (020)             bne $28, $0, split_done # if $30
has become negative, goto split_done                      %
 9 : 00000020;         % (024)             add $0, $0, $0          # nop
padding                                                   %
 A : 08000006;         % (028)             j split_loop            # continue
loop, delay slot underneath                               %
 B : 23bd0001;         % (02c)             addi $29, $29, 1        #
increment tens digit                                      %
 C : 03e00008;         % (030) split_done: jr $ra                  # return,
delay slot underneath                                     %
 D : 23dc000a;         % (034)             addi $28, $30, 10       # get
units digit and store to $28                              %
 E : 03e0a020;         % (038) show:       add $20, $31, $0        # store
return address to $20                                     %
 F : 001dd140;         % (03c)             sll $26, $29, 5         # $26 = 32
* $29(arg2, pos)                                          %
10 : 0c000005;         % (040)             jal split               # call
split (passing $30, arg1, value), delay slot underneath %
11 : 235aff20;         % (044)             addi $26, $26, 0xff20   #
calculate sevenseg pair base addr and store to $26            %
12 : 0c000002;         % (048)             jal sevenseg            # call
split (passing $30), delay slot underneath                %
13 : 03a0f020;         % (04c)             add $30, $29, $0        # move
$29(returned tens digit) to $30                           %
14 : af5d0010;         % (050)             sw $29, 16($26)         # show
sevenseg tens digit                                       %
15 : 0c000002;         % (054)             jal sevenseg            # call
split (passing $30), delay slot underneath                %
```

```
16 : 0380f020;        % (058)              add $30, $28, $0        # move
$28(returned units digit) to $30                                  %
17 : af5d0000;        % (05c)              sw $29, 0($26)          # show
sevenseg units digit                                              %
18 : 0280f820;        % (060)              add $31, $20, $0        # restore
return address                                                    %
19 : 03e00008;        % (064)              jr $ra                  # return
                                                                  %
1A : 8c05ff10;        % (068) get_op:      lw $5, 65296($0)        # load
state of keys to $5                                               %
1B : 2006ffff;        % (06c)              addi $6, $0, -1         # store
32'bffffffff to $6                                                %
1C : 00a62826;        % (070)              xor $5, $5, $6          # $5 = ~$5
                                                                  %
1D : 30a60008;        % (074)              andi $6, $5, 0x8        # get
state of key3                                                     %
1E : 14c00008;        % (078)              bne $6, $0, add_op      # if key3
is pressed, change op to add                                      %
1F : 00000020;        % (07c)              add $0, $0, $0          # nop
padding                                                           %
20 : 30a60004;        % (080)              andi $6, $5, 0x4        # get
state of key2                                                     %
21 : 14c00006;        % (084)              bne $6, $0, sub_op      # if key2
is pressed, change op to sub                                      %
22 : 00000020;        % (088)              add $0, $0, $0          # nop
padding                                                           %
23 : 30a60002;        % (08c)              andi $6, $5, 0x2        # get
state of key1                                                     %
24 : 14c00004;        % (090)              bne $6, $0, xor_op      # if key1
is pressed, change op to xor                                      %
25 : 00000020;        % (094)              add $0, $0, $0          # nop
padding                                                           %
26 : 03e00008;        % (098)              jr $ra                  # no key
pressed, no op change, return                                     %
27 : 20c6fffb;        % (09c) add_op:      addi $6, $6, -5         #
calculate new opcode                                                  %
28 : 20c6fffd;        % (0a0) sub_op:      addi $6, $6, -3         #
calculate new opcode                                                  %
29 : 03e00008;        % (0a4) xor_op:      jr $ra                  # return,
delay slot underneath                                             %
2A : 00c03820;        % (0a8)              add $7, $6, $0          #
calculate new opcode and store to $7                                  %
2B : 14e00003;        % (0ac) do_op:       bne $7, $0, not_add     # check if
op is add                                                         %
2C : 00000020;        % (0b0)              add $0, $0, $0          # nop
padding                                                           %
2D : 03e00008;        % (0b4)              jr $ra                  # return,
delay slot underneath                                             %
2E : 00432020;        % (0b8)              add $4, $2, $3          # do add
                                                                  %
2F : 20e8ffff;        % (0bc) not_add:     addi $8, $7, -1         # check if
op is sub                                                         %
30 : 15000007;        % (0c0)              bne $8, $0, not_sub     # check if
op is sub                                                         %
31 : 00432022;        % (0c4)              sub $4, $2, $3          # do sub
                                                                  %
32 : 00042fc3;        % (0c8)              sra $5, $4, 31          # extend
sign digit of the result                                          %
```

```
33 : 10a00002;        % (0cc)             beq $5, $0, sub_done    # result
is positive, done                                        %
34 : 00000020;        % (0d0)             add $0, $0, $0          # nop
padding                                                    %
35 : 00042022;        % (0d4)             sub $4, $0, $4          # result =
-result (get abs of result)                             %
36 : 03e00008;        % (0d8) sub_done:   jr $ra                  # return
                                                        %
37 : 00000020;        % (0dc)             add $0, $0, $0          # nop
padding                                                    %
38 : 03e00008;        % (0e0) not_sub:    jr $ra                  # return,
delay slot underneath                                    %
39 : 00432026;        % (0e4)             xor $4, $2, $3          # do xor
                                                        %
3A : 8c01ff00;        % (0e8) main_loop:  lw $1, 65280($0)        # load
state of switches to $1                                  %
3B : ac01ff80;        % (0ec)             sw $1, 65408($0)        # store $1
to state of leds                                        %
3C : 302203e0;        % (0f0)             andi $2, $1, 0x3e0      #
calculate value1 and store to $2                                 %
3D : 0c00001a;        % (0f4)             jal get_op              # call
get_op, delay slot underneath                            %
3E : 00011142;        % (0f8)             srl $2, $1, 5           #
calculate value1 and store to $2                                 %
3F : 0c00002b;        % (0fc)             jal do_op               # call
do_op (passing $2 and $3), delay slot underneath         %
40 : 3023001f;        % (100)             andi $3, $1, 0x1f       #
calculate value2 and store to $3                                 %
41 : 0080f020;        % (104)             add $30, $4, $0         # move
$4(result) to $30                                        %
42 : 0c00000e;        % (108)             jal show                # call
show (passing $30 and $29), delay slot underneath        %
43 : 201d0000;        % (10c)             addi $29, $0, 0         # set pos
to 0 (right pair)                                       %
44 : 0040f020;        % (110)             add $30, $2, $0         # move
$2(value1) to $30                                        %
45 : 0c00000e;        % (114)             jal show                # call
show (passing $30 and $29), delay slot underneath        %
46 : 201d0002;        % (118)             addi $29, $0, 2         # set pos
to 2 (left pair)                                        %
47 : 0060f020;        % (11c)             add $30, $3, $0         # move
$3(value2) to $30                                        %
48 : 0c00000e;        % (120)             jal show                # call
show (passing $30 and $29), delay slot underneath        %
49 : 201d0001;        % (124)             addi $29, $0, 1         # set pos
to 1 (middle pair)                                      %
4A : 0800003a;        % (128)             j main_loop             # loop
forever                                                  %
4B : 00000020;        % (12c)             add $0, $0, $0          # nop
padding                                                    %
END ;
```

与单周期 CPU 实验的代码基本相似，这里不再过多解释代码的具体含义，详情可以参考单周期 CPU 的实验报告，唯一的不同之处是因为引入了 MIPS 的延迟槽机制，将原本位于 `j` `jr` `jal` `bne` `beq` 指令前一行的指令放到了后一行，在不影响指令正常执行的前提下，避免了 Control Hazards 的出现。

14. `pipeimem.mif/pipedmem.mif` 测试指令和数据文件

```
DEPTH = 128;           % Memory depth and width are required %
WIDTH = 32;            % Enter a decimal number %
ADDRESS_RADIX = HEX;   % Address and value radixes are optional %
DATA_RADIX = HEX;      % Enter BIN, DEC, HEX, or OCT; unless %
                       % otherwise specified, radixes = HEX %
CONTENT
BEGIN
[0..7F] : 00000000;    % Range--Every address from 0 to 7F = 00000000 %

0 : 3c010000; % (00) main:      lui     r1, 0           # address of data[0]
       %
1 : 34240050; % (04)            ori     r4, r1, 80      # address of data[0]
       %
2 : 0c00001b; % (08) call:      jal     sum             # call function
       %
3 : 20050004; % (0c) dslot1:    addi    r5, r0, 4       # counter, DELYED
SLOT(DS)  %
4 : ac820000; % (10) return:    sw      r2, 0(r4)       # store result
       %
5 : 8c890000; % (14)            lw      r9, 0(r4)       # check sw
       %
6 : 01244022; % (18)            sub     r8, r9, r4      # sub: r8 <-- r9 -
r4       %
7 : 20050003; % (1c)            addi    r5, r0, 3       # counter
       %
8 : 20a5ffff; % (20) loop2:     addi    r5, r5, -1      # counter - 1
       %
9 : 34a8ffff; % (24)            ori     r8, r5, 0xffff  # zero-extend:
0000ffff     %
A : 39085555; % (28)            xori    r8, r8, 0x5555  # zero-extend:
0000aaaa     %
B : 2009ffff; % (2c)            addi    r9, r0, -1      # sign-extend:
ffffffff     %
C : 312affff; % (30)            andi    r10, r9, 0xffff # zero-extend:
0000ffff     %
D : 01493025; % (34)            or      r6, r10, r9     # or: ffffffff
       %
E : 01494026; % (38)            xor     r8, r10, r9     # xor: ffff0000
       %
F : 01463824; % (3c)            and     r7, r10, r6     # and: 0000ffff
       %
10: 10a00003; % (40)            beq     r5, r0, shift   # if r5 = 0, goto
shift     %
11: 00000000; % (44) dslot2:    nop                     # DS
       %
12: 08000008; % (48)            j       loop2           # jump loop2
       %
13: 00000000; % (4c) dslot3:    nop                     # DS
       %
14: 2005ffff; % (50) shift:     addi    r5, r0, -1      # r5 = ffffffff
       %
15: 000543c0; % (54)            sll     r8, r5, 15      # <<15 = ffff8000
       %
16: 00084400; % (58)            sll     r8, r8, 16      # <<16 = 80000000
       %
```

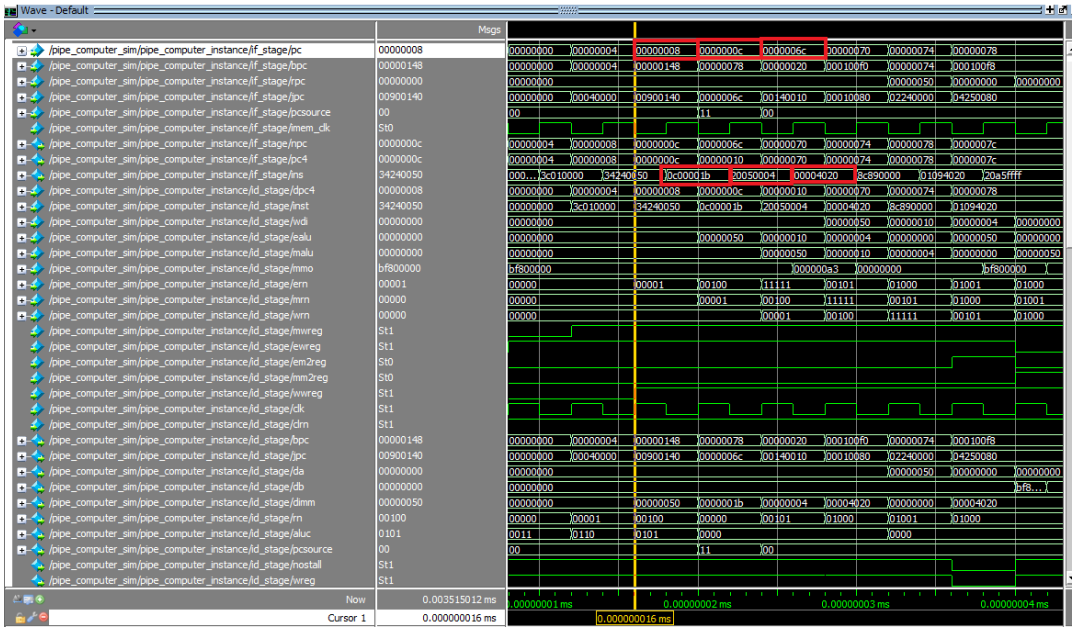
```
17: 00084403; % (5c)            sra     r8, r8, 16      # >>16 =
ffff8000(arith)    %
18: 000843c2; % (60)            srl     r8, r8, 15      # >>15 =
0001ffff(logic)    %
19: 08000019; % (64) finish:    j       finish          # dead loop
        %
1A: 00000000; % (68) dslot4:    nop                     # DS
       %
1B: 00004020; % (6C) sum:       add     r8, r0, r0      # sum
        %
1C: 8c890000; % (70) loop:      lw      r9, 0(r4)       # load data
        %
1D: 01094020; % (74)            add     r8, r8, r9      # sum
        %
1E: 20a5ffff; % (78)            addi    r5, r5, -1      # counter - 1
        %
1F: 14a0fffc; % (7c)            bne     r5, r0, loop    # finish?
        %
20: 20840004; % (80) dslot5:    addi    r4, r4, 4       # address + 4, DS
        %
21: 03e00008; % (84)            jr      r31             # return
       %
22: 00081000; % (88) dslot6:    sll     r2, r8, 0       # move result to v0,
DS     %
END;
```

```
DEPTH = 32;           % Memory depth and width are required %
WIDTH = 32;           % Enter a decimal number %
ADDRESS_RADIX = HEX;  % Address and value radixes are optional %
DATA_RADIX = HEX;     % Enter BIN, DEC, HEX, or OCT; unless %
                      % otherwise specified, radixes = HEX %
CONTENT
BEGIN
[0..1F] : 00000000;   % Range--Every address from 0 to 1F = 00000000 %
0 : BF800000; % 1 01111111 00..0 fp -1                              %
14: 000000A3; % (50) data[0]    0 + A3 = A3                         %
15: 00000027; % (54) data[1]    A3 + 27 = CA                        %
16: 00000079; % (58) data[2]    CA + 79 = 143                       %
17: 00000115; % (5C) data[3]    143 + 115 = 258                     %
END ;
```

这部分代码来自李亚民老师，是一个简单的累加程序，但是是非最优化的程序，比如在 `lw` 指令的下面故意安排了一条 `sub` 指令，使用 `lw` 指令从存储器取来的数据，以测试流水线 CPU 对 Hazards 的应对情况。以此程序的仿真测试为例，检查 CPU 的设计是否符合要求，是否能正确处理引入流水线设计带来的问题。

1. **延迟槽 Delay Slot:**

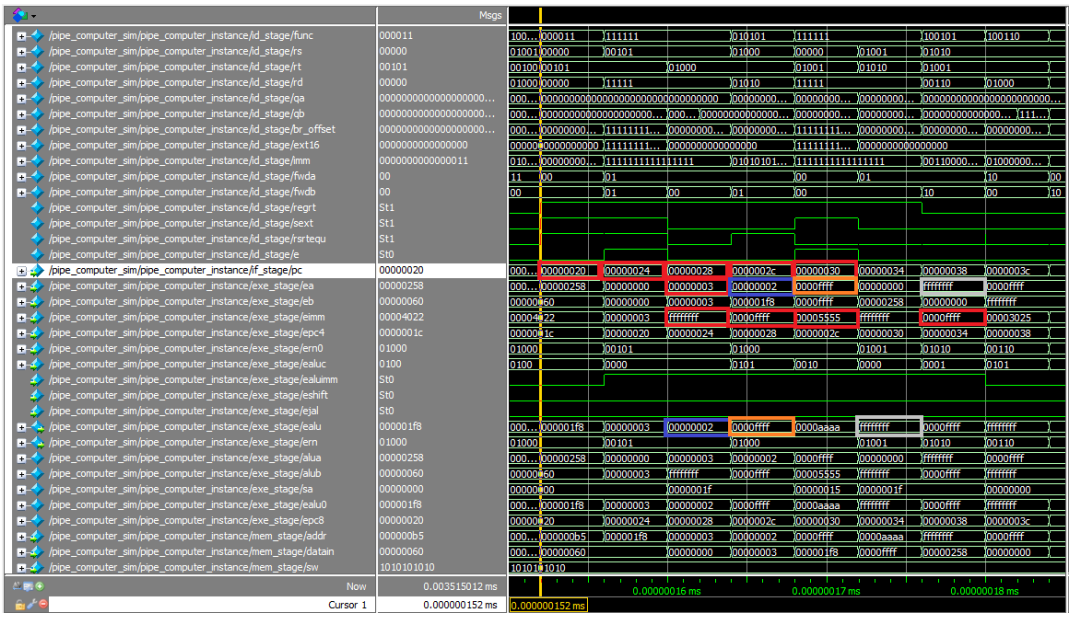
   示例代码中有多个延迟槽的使用，此处以 `dslot1` 附件的代码仿真执行情况为例，仿真波形图如下：



可以看到，在红色标记的部分，对应示例代码的 2-3 行、1B 行的汇编代码，即在执行 PC = 8 的 `jal sum` 指令之后，没有在下一个周期立即执行 PC = 6C 的 `add r8, r0, r0` 指令，而是先执行了 PC = C 处的延迟槽指令 `addi r5, r0, 4`，是符合预期的，说明本 CPU 的设计满足 MIPS 语言延迟槽的思想。

2. **内部前推 Internal Forwarding**

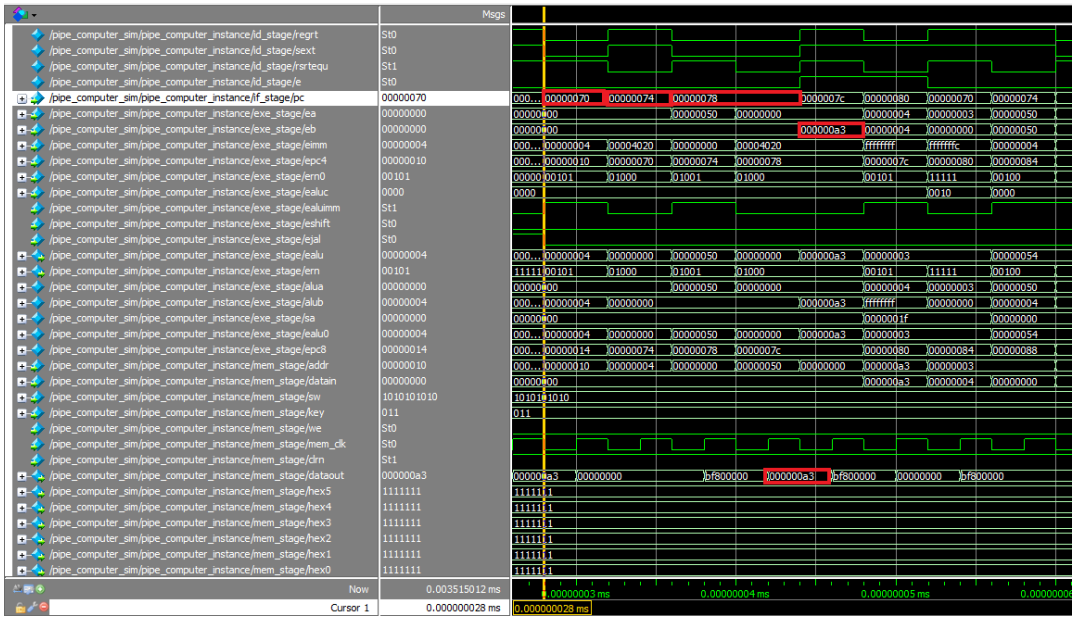
通过分析示例代码可以看到，在 `loop2` 循环中，有很多的**数据相关**情况发生，需要利用内部前推的方式解决数据依赖。汇编代码 8-C 行的执行如下：



有标记的部分是这几行代码对应的 PC，以及 EXE 阶段（要比对应的 IP 阶段晚 2 个周期）的参数和结果的值，除红色外，颜色相同的标记表示两个数值存在数据依赖，可以看出，数据依赖被正确处理，上一条指令 EXE 阶段计算出的值可以被及时前推到后续指令的对应阶段，保证了 CPU 可以按照既定的周期正常运行，数据没有出现错误。

3. **流水线暂停 Stalling**

因为示例代码不是最优化的代码，刻意保留了存在数据相关的情况，例如代码的 1C-1D 行，`lw r9, 0(r4)` 指令后续紧跟着一条 `add r8, r8, r9` 的指令，因此数据还没有及时读取到 `r9` 寄存器中，需要暂停 CPU 一个周期。仿真波形图如下：



可以看出，当处理器执行了 PC = 70 处的 `lw r9, 0(r4)` 指令之后，又继续读取了 PC = 74 处的 `add r8, r8, r9` 指令和 PC = 78 处的 `addi r5, r5, -1` 指令，由于数据相关导致流水线需要暂停，可以看到 PC = 78 指令的 IF 阶段（对应 74 指令的 ID 和 70 指令的 EXE 阶段）明显停止了一个时钟周期（表现为此处的 PC 值没有变化）等到 `lw` 指令成功读取了数据之后（对应图中最下方的标记）`add` 指令才能继续执行 EXE 阶段，流水线暂停结束。这一部分的处理逻辑也符合预期。

通过仿真测试，还可以查看每个寄存器、内存的具体数据。结合上面对流水线 CPU 关键设计的分析，最终的设计符合程序的预期，实验中设计的五段流水 CPU 可以正常处理 Hazards。

实验总结

实验结果

设计的五段流水 CPU 可以正常通过编译，在 ModelSim 中运行 Testbench 可以产生正确的波形图，且输出符合预期。将代码载入开发板后可以正常运行编译后的计算器程序，7 段 LED 数码管和 LED 发光二极管可以正常显示，10 个开关和 4 个按钮都可以正常输入，计算器运行逻辑符合预期；测试程序可以正常进行仿真运行，没有因为 Hazards 出现问题。

经验总结与反思

1. 本实验的大部分代码与单周期 CPU 相同，但需要按照 CPU 的功能将原本集中的逻辑分割成五段流水，并添加每一级之间的流水线寄存器，需要按照电路图进行设计，大部分的代码参考了李亚民老师的教材；
2. 因为 MIPS 的特点，可以通过引入延迟槽（Delay Slot）的方式解决 control hazards，而 data hazards 的解决需要添加转发旁路和多路选择器；对于需要插入气泡的指令，通过增加 `wpcir` 信号控制新指令的取指和译指；通过 `wmem` `m2reg` 指令控制访存和写回阶段是否允许当前指令写内存/寄存器；
3. 在本实验中，指令 ROM 是同步读，寄存器文件是异步读、同步写，数据 RAM 是同步读写，I/O 端口是异步读、同步写；
4. 在 Quartus 的使用中，能够正常通过编译的代码不一定能够正常运行，还需要关注编译时的 warning 信息；代码文件名也会影响程序能否正常通过编译、烧录到开发板上，顶层文件和某个 Verilog 文件重名可能会导致程序无法正常运行。
5. 能够正常仿真的代码也不一定能够在烧录到开发板之后正常运行，这部分的原因比较复杂，目前只能通过仔细编写代码、尽量减少 warning 的方式避免；