计算机组成原理project报告

开发者说明

12110916 李浩宇，负责状态控制、IO、asm，贡献比33%
12112609 刘一郴，负责pipeline CPU模块设计，hazard解决方案，贡献比33%
12110924 田若载，负责Uart通信、memory管理、ALU等基础模块，贡献比33%

版本修改记录

CPU架构设计说明

CPU特性

ISA
寻址空间设计
属于哈佛结构
寻址单位：1byte
指令空间、数据空间大小均为64KB
对外设IO的支持
采用MMIO，拨码开关对应的地址是0x00000000,0x00000004,0x00000008
LED对应的地址是0x0000000C
数码管对应的地址是0x0000010
采用中断的方式访问IO
CPU的CPI
不考虑hazard的情况下CPU的CPI为1。
对于data hazard, 只有lw引起的hazard会多产生一个cycle的花费。
对于branch hazard， 只有发生跳转会多产生一个cycle的花费。
该CPU为多周期pipelineCPU，采用经典5级流水（IF, ID, EXE, MEM, WB）。
hazard 的大致解决方案分别是(将在bonus里详细描述)：

1. structural hazard: 我们采用havard架构，指令与数据分离，也就没有structural hazard；
2. data hazard: 在ID stage， controller 会判断rs， rt 是否要被之前的代码修改，该判断是通过记录m2reg（是否从mem写回到寄存器）， wreg（是否写reg）， ern （要写的register的number）并把该三个信号流水线传递到下一级stage， 例如 em2reg（EXE stage 的 m2reg）, mm2reg（MEM stage 的 m2reg）, 命名规则很容易看出。把各级的相关信息传回ID stage 的controller， 我们就可以指导上几条instruction的相关信息，就可以选择正确的forwarding。例如，if(ewreg &(ern != 0) & (ern == rs) & ~em2reg) 就代表着要实现从EXE到ALU的forwarding. 除此之外还有MEM到ALU， LW_ALU.然后，对于 lw 和 使用拿出来

的值的情况，CPU 不得不使用一个NOP，stall 产生条件为（ewreg & em2reg & (ern != 0) & (i_rs & (ern == rs)) | (i_rt & (ern == rt)) ） 其中i_rs i_rt 代表要不要用到rs 或者 rt， 有了stall 信号后， CPU 支持stall 的具体操作为不更新PC寄存器，registers，data mem, IF ID 之间的寄存器不更新。

3. branching hazard: 我们实现了如果不跳转，没有任何NOP产生，CPU正常执行，如果跳转，产生一个NOP的花费。具体实现方式为，我们在ID stage 就能够判断该CPU会不会发生跳转，判断方式为如果是beq这种要比较值的分支指令，就直接把两个值进行比较是否相等。Jr， j， jal 毫无疑问是要分支的。这样的话我们记录是否发生分支记录为信号branch，连接到IF ID 之间的流水线寄存器， 这样下一条指令就能知道上一条指令是否发生跳转。如果没有跳转，那么CPU正常执行，没有任何花费， 如果跳转， 那么该指令（也就是紧接着分支指令的下一条指令）应该相当于NOP， 因此在解码的时候应当把该指令认为什么也不是，同时wreg， wmem都记为0，也就是不对registers 和 data memory进行修改。

CPU接口

时钟信号

`clk`： Minisys内置时钟信号 (PIN Y18)

复位信号

`reset`：按钮，按下时重置CPU至初始状态 (PIN P20)

uart接口

`start_pg`：开关，打开时设置CPU为Uart通信模式，关闭时设置CPU为工作模式 (PIN AA6)

`rx`：Uart输入 (PIN Y19)

`tx`：Uart输出 (PIN V18)

按钮

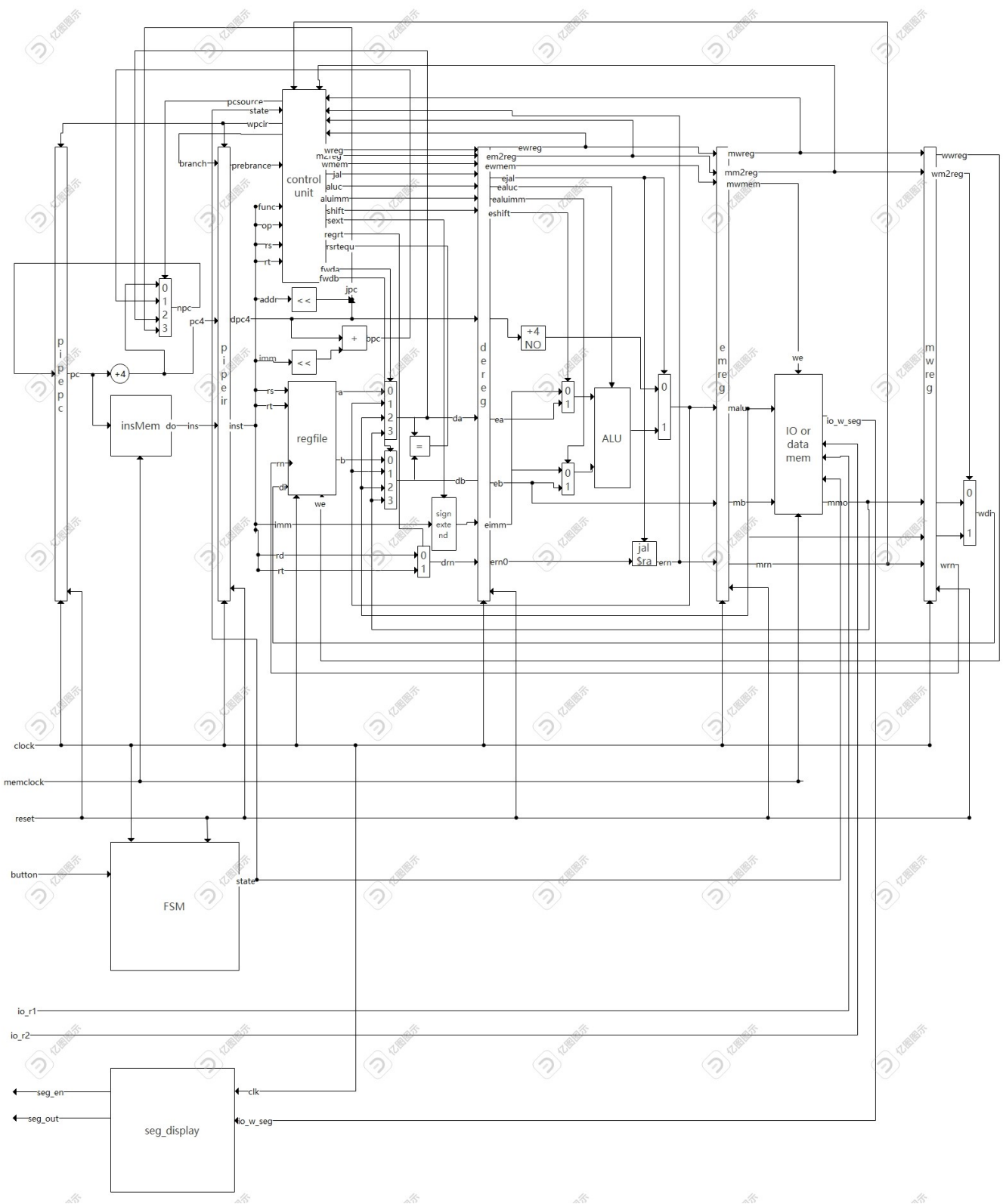
控制I/O和CPU状态的切换(PIN P4)

拨码开关

用于输入数据(PIN Y9,W9,Y7,U6,W5,W6,U5,T5,T4,R4,W4)

CPU内部结构

CPU内部各子模块的接口连接关系图



CPU内部子模块的设计说明

子模块名	端口说明	功能描述
FSM	**输入信号：** rst: 复位信号 button: 按钮 **输出信号：** state: 当前所处的状态（I/O,CPU）	根据按钮改变state来控制。
MemOrIO	**输入信号：** state: 当前所处的状态（I/O,CPU） addr_in: ALU算出的地址 m_rdata: 从内存读出来的数据 io_rdata: 从输入设备读入的数据 r_rdata: 从寄存器读出来的数据 **输出信号：** addr_out: 读/写内存的地址 m_wdata: 将要写入内存的数据 io_wdata: 要写入输出设备的数据 r_wdata: 将要写入寄存器的数据	根据state的值，决定输出信号的值： addr_out在CPU状态下就是addr_in， 在I/O状态下就是I/O设备对应的地址。 m_wdata在CPU状态下是r_rdata， 在I/O状态下是io_rdata。 io_wdata就是m_wdata。 r_wdata就是m_wdata。
seg_display	**输入信号：** clk: 时钟 out1,out2,out3: 要输出到数码管的数值 **输出信号：** seg_en,seg_out: 传给数码管的信号	利用视觉暂留，在数码管上输出结果
mux2x5	**输入信号：** s0, s1: 待选择的信号 s: 选择线 **输出信号：** y: 输出的信号	根据选择线从2个信号选择5位信号输出。

子模块名	端口说明	功能描述
mux2x32	**输入信号：** s0, s1: 待选择的信号 s: 选择线 **输出信号：** y: 输出的信号	根据选择线从2个信号中选择32位信号输出。
mux4x32	**输入信号：** a0, a1, a2, a3: 待选择的信号 s: 选择线 **输出信号：** y: 输出的信号	根据选择线从4个信号中选择32位信号输出。
shift	**输入信号：** d: 被移位数 sa: 移位量 right: 是否右移 arith: 是否符号填充 **输出信号：** r: 移位结果	对一个数进行移位。
alu	**输入信号：** a, b: 操作数 aluc: 操作码 **输出信号：** r: 计算结果 z: 结果是否为0	aluc的意义如下： 0 0 0 0 ADD 0 1 0 0 SUB 1 0 0 0 MUL 1 1 0 0 DIV X 0 0 1 AND 0 1 0 1 OR 1 1 0 1 NOR 0 0 1 0 XOR 0 1 1 0 LUI 1 0 1 0 SLT 1 1 1 0 SLTU 0 0 1 1 SLL 0 1 1 1 SRL 1 1 1 1 SRA

子模块名	端口说明	功能描述
dffe32	**输入信号：** d: 要写入的数 clk: CPU时钟信号 clrn: 清零信号 e: 写使能信号 **输出信号：** q: 寄存器的值	实现了一个32位寄存器。
regfile	**输入信号：** rna, rnb: 要读取的寄存器号 d: 要写入的数 wn: 要写入的寄存器号 we: 写使能信号 clk: CPU时钟信号 clrn: 清零信号 **输出信号：** qa, qb: 寄存器的值， 分别对应rna和rnb号寄存器	实现了32个通用寄存器，包括一个0寄存器。
pipepc	**输入信号：** npc: 要写入PC的数 wpc: 是否允许写入PC clk: CPU时钟信号 clrn: 清零信号 **输出信号：** pc: PC的值	包装了dffe32，实现了一个PC。
pipeif	**输入信号：** memclk: memory使用的时钟信号 pcsource: 下一个PC的来源控制信号 pc: 当前的PC bpc, rpc, jpc: 分别是beq、 bne/jr/j、jal的目标地址 upg_rst, upg_clk, upg_wen, upg_adr, upg_dat, upg_done: Uart相关信号 **输出信号：** pc4: PC+4的结果 npc: 下一个PC地址 inst: 具体的指令	实现了CPU的IF stage。

子模块名	端口说明	功能描述
pipeimem	**输入信号：** clk: memory使用的时钟信号 a: 要访问的地址 upg_rst, upg_clk, upg_wen, upg_adr, upg_dat, upg_done: Uart相关信号 **输出信号：** inst: 访问的值	实现了一个用于存储指令的ROM， 在Uart通信时变为RAM。
pipeir	**输入信号：** pc4: PC+4的结果 ins: 读出的指令 brance: 是否该指令发生跳转 wir: 是否允许写入指令寄存器 clk: CPU时钟信号 clrn: 清零信号 **输出信号：** dpc4, inst, prebrance: 依据wir决定是否更新的值， 与上面的输入一一对应，	实现了指令寄存器。
pipeid	**输入信号：** dpc4, inst, wdi, ealu, malu, mmo: 32bits, dpc4为IDstage中的PC+4， inst为IDstage中的instruction， wdi为WB中的寄存器输入值， ealu为EXEstage中传入的alu返回值用来forwarding， malu同理， mmo为MEMstage中memory的取出值用于forwarding； ern，wrn，mrn： 来自EXE，MEM， WB的rn信号 （要写入的寄存器编号）	IDstage 负责对当前指令的解码， 并通过内部的control unit 来控制CPU各个硬件的相应行为， 来实现正常程序执行以及forwarding和stall。

子模块名	端口说明	功能描述
	prebrance： 上一条指令是否发生branch，用于处理branch hazard mwreg, ewreg, em2reg, mm2reg， wwreg: 分别来自MEM、EXE的reg、m2reg信号，用来确定上一条指令或者上上条指令的相关信息 rsrtequ: 是否要用到的rsrt的值相同 clk， clrn：时钟控制信号 **输出信号：** bpc，jpc，a，b，imm：32bits， 分别为branch到的pc，jump到的PC，a、b为进入EXE的ALU的两个值，imm为imm值 rn: 5bits, 要写回的 register number wreg, m2reg, wmem, regrt, aluimm, sext, shift, jal: 1bit，分别是： 是否写寄存器， 是否从memory取数据写回到寄存器， 是否写memory， rt寄存器值， alu要处理的imm值， 是否扩展符号位， 是否shift，是否为jal指令 aluc: 4bits， 控制alu进行何种操作 pcsource: 2bits， 控制下一个pc的来源(pc+4、branch pc等等) nostall: 1bit， 控制是否发生stall， 即pc不进行更新， ID来自IF的输入不变。 brance:	

子模块名	端口说明	功能描述
	该指令是否发生跳转， 用于传给上一级流水寄存器	
pipeidcu	**输入信号：** mwreg, ewreg, em2reg, mm2reg: 分别来自MEM、EXE的reg、m2reg信号， 用来确定上一条指令或者上上条指令的相关信息 rsrtequ: 是否要用到的rsrt的值相同 mrn, ern: 5bits，mrn、ern是来自MEM、EXE的要写回的register number rs, st: 5bits，rs、rt 对应的register number func, op: 6bits，func、op code 用来解码 state: 8bits， 显示当前运行状态， state为4时正常运行 prebrance: 是否上一条指令发生跳转 **输出信号：** wreg, m2reg, wmem, regrt, aluimm, sext, shift, jal: 1bit，分别是： 是否写寄存器， 是否从memory取数据写回到寄存器， 是否写memory， rt寄存器值， alu要处理的imm值， 是否扩展符号位， 是否shift，是否为jal指令 aluc: 4bits， 控制alu进行何种操作 pcsource: 2bits， 控制下一个pc的来源 (pc+4、branch pc等等) fwda,dwdb: 2bits，	程序的核心模块，位于IDstage的control unit. 负责对当前ID对应的指令的解码， 控制各个硬件的相应行为； 判断是否发生forwarding以及stall， 并控制相应硬件信号来实现forwarding和stall。

子模块名	端口说明	功能描述
	控制forwarding的类型， 即控制传入alu的a、 b的值的来源. nostall: 1bit， 控制是否发生stall， 即pc不进行更新， ID来自IF的输入不变。	
pipedereg	**输入信号：** clk: CPU的时钟信号 clrn: 重置信号 dwreg, dm2reg, dwmem, daluc, daluimm, da, db, dimm, drn, dshift, djal, dpc4: ID stage要传出的信号 **输出信号：** ewreg, em2reg, ewmem, ealuc, ealuimm, ea, eb, eimm, ern, eshift, ejal, epc4: EXE stage将收到的信号， 和上面的输入信号一一对应	由ID stage到EXE stage的流水线寄存器。
pipeexe	**输入信号：** ealuc: alu的操作码 ealuimm: 是否使用立即数 ea, eb: alu的两个操作数 （来源寄存器） eimm: 立即数 eshift: 是否移位 ern0: 之前决定的写入的寄存器号 epc4: PC+4的值 ejal: 是否是jal指令 **输出信号：** ern: 要写入的寄存器号 ealu: 指令运算的结果	实现了CPU的EXE stage。

子模块名	端口说明	功能描述
pipeemreg	**输入信号：** clk: CPU的时钟信号 clrn: 重置信号 ewreg, em2reg, ewmem, ealu, eb, ern: EXE stage要传出的信号 **输出信号：** mwreg, mm2reg, mwmem, malu, mb, mrn: MEM stage将收到的信号，和上面的输入信号一一对应	由EXE stage到MEM stage的流水线寄存器。
pipemem	**输入信号：** state: 当前状态机的状态 we: 写使能信号 addr: 要读/写的地址 datain: 要写入的数 clk: CPU的时钟信号 memclk: memory使用的时钟信号 io_r1, io_r2: IO相关信号 upg_rst, upg_clk, upg_wen, upg_adr, upg_dat, upg_done: Uart相关信号 **输出信号：** dataout: 从memory中读出的值 io_w__led, io_w_seg_1, io_w_seg_2, io_w_seg_3: IO相关信号	包装了一个RAM， 实现了CPU的MEM stage。
pipemwreg	**输入信号：** clk: CPU的时钟信号 clrn: 重置信号 mwreg, mm2reg, mmo, malu, mrn: MEM stage要传出的信号 **输出信号：** wwreg, wm2reg, wmo, walu, wrn: WB stage将收到的信号，和上面的输入信号一一对应	由MEM stage到WB stage的流水线寄存器。

测试说明

测试方法： 上板
测试类型： 集成
测试用例及结果：
场景1：

	用例描述	**测试数据及结果**
3’b000	输入测试数a（仅识别a的最低7bit）， 输入完毕后在led灯上显示a， 同时用1个led灯显示a的奇校验位	0xff: led暗; 0x7e: led亮
3’b001	输入测试数a（识别a的完整8bit）， 输入完毕后在led灯上显示a， 同时用1个led灯显示a的奇校验结果	0xff: led暗; 0x7f: led亮
3’b010	先执行测试用例3’b111, 再计算 a 和 b的按位或非运算， 将结果显示在输出设备	0x0a,0x05: 0xf0
3’b011	先执行测试用例3’b111, 再计算 a 和 b的按位或运算， 将结果显示在输出设备	0x0a,0x05: 0x0f
3’b100	先执行测试用例3’b111, 再计算 a 和 b的按位异或运算， 将结果显示在输出设备	0x0a,0x07: 0x0d
3’b101	先执行测试用例3’b111, 再执行 sltu 指令， 将a和b按照无符号数进行比较， 用输出设备展示a<b的关系是否成立 (关系成立，亮灯， 关系不成立，灭灯)	0x01,0x02: led亮; 0xff,0x01: led暗
3’b110	先执行测试用例3’b111, 再执行 slt 指令， 将a和b按照有符号数进行比较， 用输出设备展示a<b的关系是否成立(关系成立，亮灯， 关系不成立，灭灯)	0x01,0x02: led亮; 0x01,0xff: led暗
3’b111	输入测试数a, 输入测试数b，在输出设备上展示a和b的值	0x01,0x02: 0x01,0x02

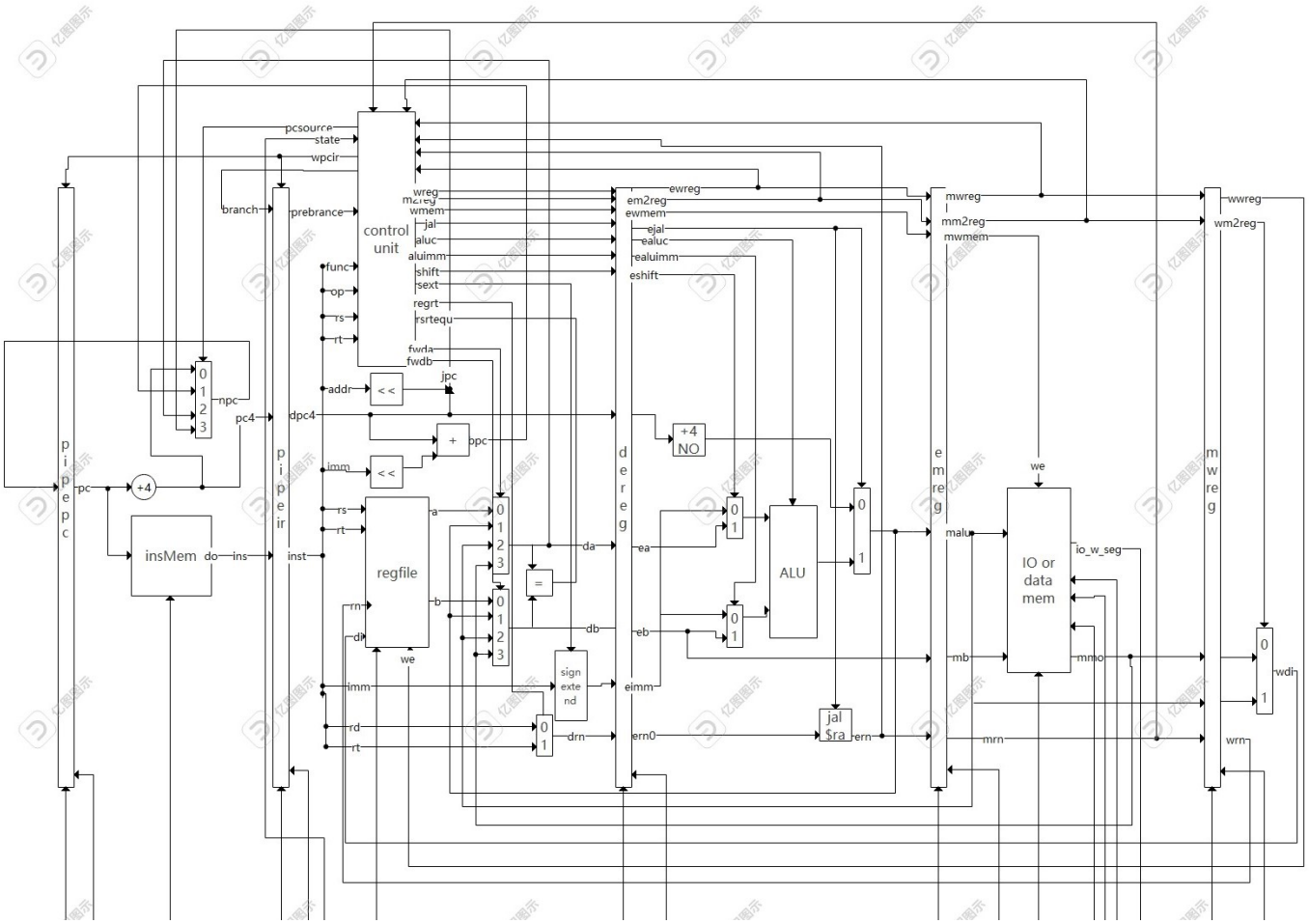
场景2：

用例编号	用例描述	测试数据及结果
3’b000	输入a的数值（a被看作有符号数）， 计算1到a的累加和， 在输出设备上显示累加和 （如果a是负数， 以闪烁的方式给与提示）	0x06: 0x15;0xff: led闪烁
3’b001	输入a的数值（a被看作无符号数）， 以递归的方式计算1到a的累加和， 记录本次入栈和出栈次数， 在输出设备上显示入栈和出栈的次数之和	0x06: 0x1c
3’b010	输入a的数值（a被看作无符号数）， 以递归的方式计算1到a的累加和， 记录入栈和出栈的数据， 在输出设备上显示入栈的参数， 每一个入栈的参数显示停留2-3秒 （说明，此处的输出不关注 $ra的入栈和出栈信息）	0x06: 0x06,0x05,0x04,0x03,0x02,0x01,0x00
3’b011	输入a的数值（a被看作无符号数）， 以递归的方式计算1到a的累加和， 记录入栈和出栈的数据， 在输出设备上显示出栈的参数， 每一个出栈的参数显示停留2-3秒 （说明，此处的输出不关注 $ra的入栈和出栈信息）	0x06: 0x00,0x01,0x02,0x03,0x04,0x05,0x06
3’b100	输入测试数a和测试数b， 实现有符号数（a， b以及相加和都是8bit， 其中的最高bit被视作符号位， 如果符号位为1， 表示的是该负数的补码）的加法， 并对是否溢出进行判断， 输出运算结果以及溢出判断	0x80,0x80: 0x00,led亮

用例编号	用例描述	测试数据及结果
3’b101	输入测试数a和测试数b， 实现有符号数（a， b以及差值都是8bit， 其中的最高bit被视作符号位， 如果符号位为1， 表示的是该负数的补码）的减法， 并对是否溢出进行判断， 输出运算结果以及溢出判断	0x80,0x7f: 0x01, led亮
3’b110	输入测试数a和测试数b， 实现有符号数（a，b都是8bit， 乘积是16bit， 其中的最高bit被视作符号位， 如果符号位为1， 表示的是该负数的补码）的乘法， 输出乘积	0x03,0xf9: 0xffeb; 0x03,0x07: 0x0015
3’b111	输入测试数a和测试数b， 实现有符号数（a，b， 商和余数都是8bit， 其中的最高bit被视作符号位， 如果符号位为1， 表示的是该负数的补码）的除法， 输出商和余数（商和余数交替显示， 各持续5秒）	0x07,0x03: 0x02,0x01(交替显示); 0xf9,0x03: 0xfe,0xff(交替显示)

bonus设计说明

bonus为*多周期pipeline*, 采用的是经典五级流水架构（IF, ID, EXE, MEM, WB）
具体实现方式为经典的插入流水线寄存器的方式。



视频

见链接 https://www.bilibili.com/video/BV1qP411D7Cc/ (https://www.bilibili.com/video/BV1qP411D7Cc/)

控制stall的方式

stall 产生有两种情况，一种是data hazard 中的lw-ALU. 也就是刚从MEM中取出数据并没有写入register而EXE要用的情况。第二种是CPU不执行状态（state！=4）

```
wire i_rs = i_add | i_sub | i_and | i_or | i_nor | i_xor | i_jr  | i_mul | i_div | i_s
                i_andi| i_ori | i_xori| i_lw | i_sw  | i_beq | i_bne | i_slti;
wire i_rt = i_add | i_sub | i_and | i_or | i_nor | i_xor | i_mul | i_div | i_slt | i_s
                i_sra | i_sw  | i_beq | i_bne;
assign nostall = (~(ewreg & em2reg & (ern != 0) & (i_rs & (ern == rs) | i_rt & (ern ==
```

- ewreg: EXE stage 中的wreg信号（是否写register）
- em2reg: EXE stage 中的 m2reg 信号（是否memory中读取并写入register）
- ern: EXE stage 中 rn 信号（要写的 register number）
- i_rs: 是否要用到rs的值

- i_rt: 是否要用到rt的值
- rs: rs所对应的register number
- rt: rt所对应的register number

同时在具体物理实现中，可以发现电路图中有个wpcir信号（也就是nostall）传输进pipepc和pipeir流水线寄存器中，控制着寄存器的值是否更新。
同时wreg，wmem也要进行更改，如果该指令为NOP，就不对register和memory进行更改。

```
assign wreg    = (i_add | i_sub | i_and | i_or  | i_nor | i_xor | i_sll | i_mul | i_di
                      i_slt | i_sltu| i_srl | i_sra | i_addi| i_andi| i_ori | i_xori|
                      i_lw  | i_lui | i_jal | i_slti) & nostall & ~prebrance; // if wr
assign wmem    =  i_sw & nostall & ~prebrance;
```

structural hazard

由于采用havard结构，不存在structural hazard的问题。

data hazard

这里存在三种forwarding ALU-ALU，MEM-ALU，LW-ALU
图示为ALU-ALU，MEM-ALU的问题产生原因与解决方案

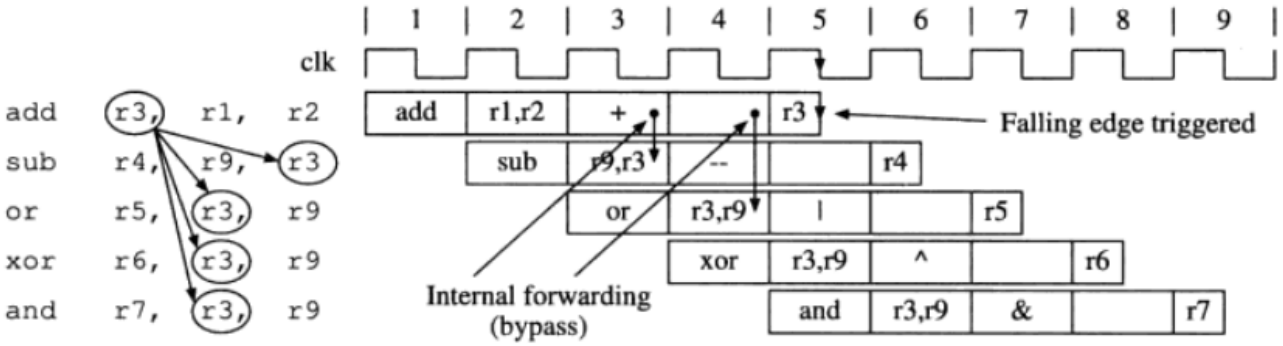
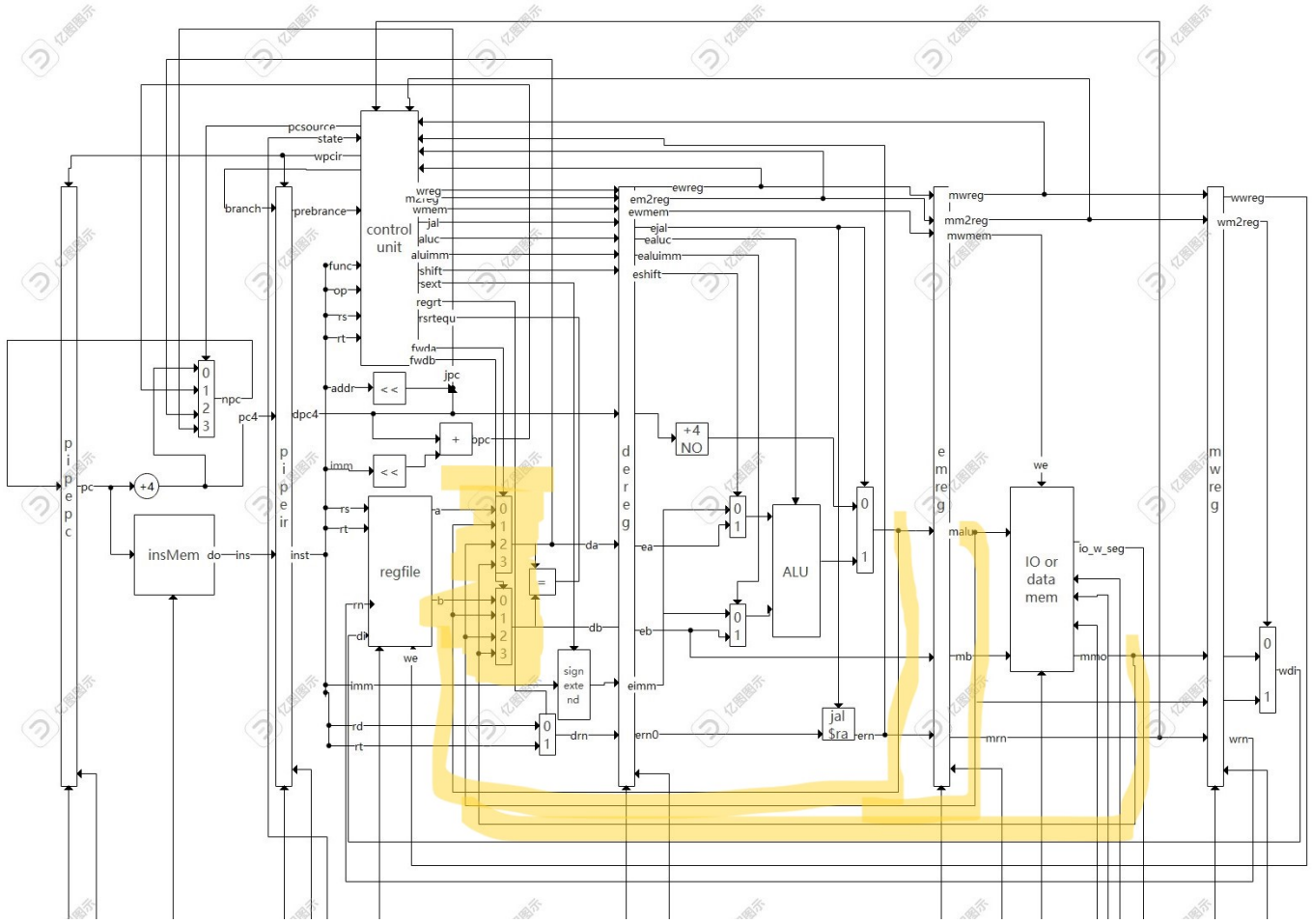


图 8.10　解决数据相关问题 —— 内部前推

图例标志了forwarding的线路



该选择的信号为fwda，fwdb，具体代码为 pipeidcu.v中

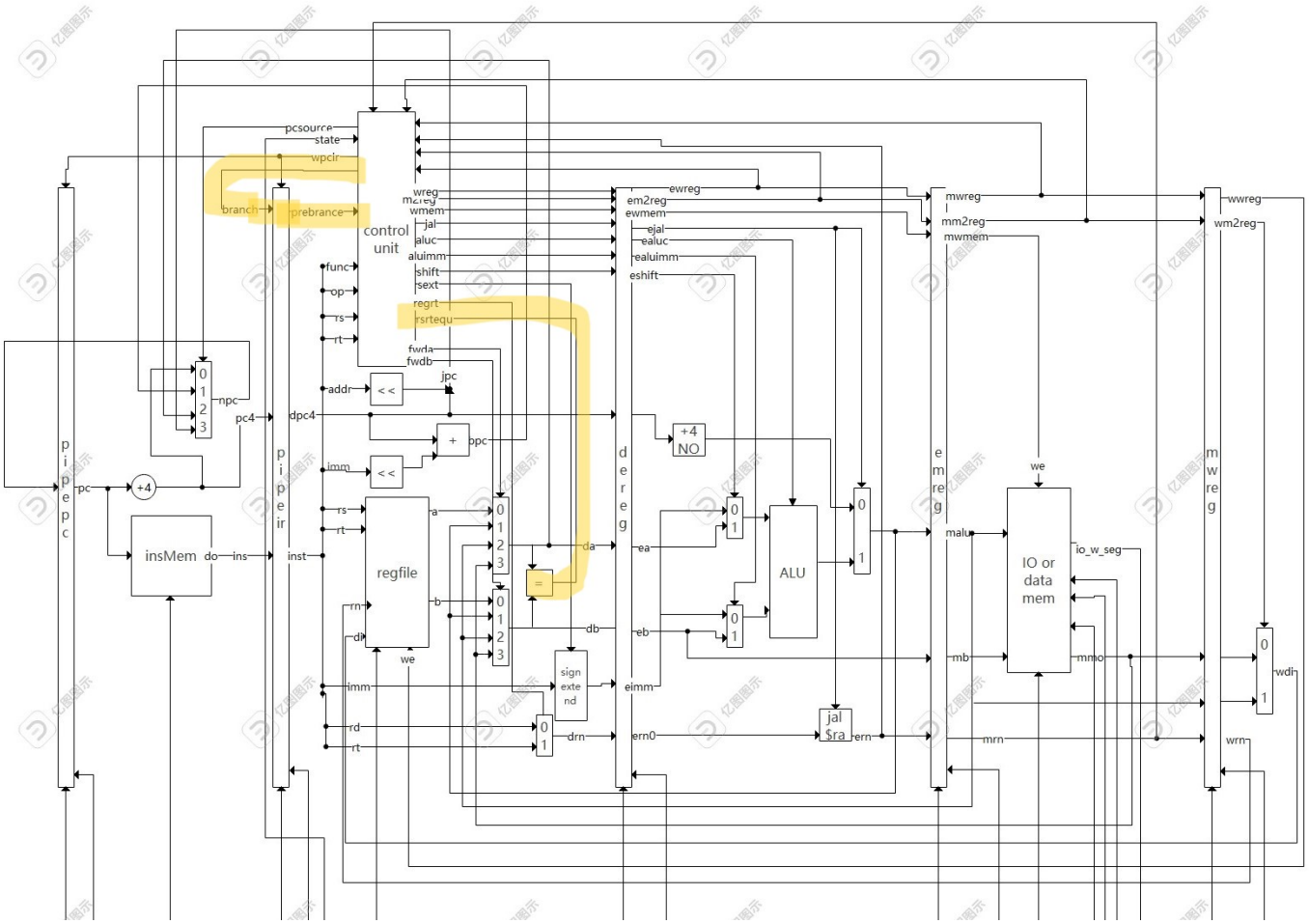
```
always @(ewreg or mwreg or ern or mrn or em2reg or mm2reg or rs or rt) begin
    fwda = 2'b00; // default: no hazards
    if(ewreg & (ern != 0) & (ern == rs) & ~em2reg) begin
        fwda = 2'b01; // exe_alu
    end else begin
        if (mwreg & (mrn != 0) & (mrn == rs) & ~mm2reg) begin
            fwda = 2'b10; //mem_alu
        end else begin
            if(mwreg & (mrn != 0) & (mrn == rs) &  mm2reg) begin
                fwda = 2'b11; //mem_lw
            end
        end
    end
    fwdb = 2'b00; // default: no hazards
    if(ewreg & (ern != 0) & (ern == rt) & ~em2reg) begin
        fwdb = 2'b01; // exe_alu
    end else begin
        if (mwreg & (mrn != 0) & (mrn == rt) & ~mm2reg) begin
            fwdb = 2'b10; //mem_alu
        end else begin
            if(mwreg & (mrn != 0) & (mrn == rt) &  mm2reg) begin
                fwdb = 2'b11; //mem_lw
            end
        end
    end
end
```

其中，LW造成的hazard还需要一个NOP也就是一个stall
stall的产生已经在上面说过了不再赘述.

branch hazard

首先我们在ID模块就能够识别该指令是否发生跳转。
实现方式为我们记录branch信号来表示是否发生跳转。
对于无条件跳转j，jr，jal自然识别为发生跳转，branch为1。

对于有条件跳转beq, bne，我们在ID模块取出两个对应的register值后直接进行比较，用rsrtequ来记录是否相同。



branch hazard 为如果这条指令发生跳转，那么由于流水线进入IF的指令也就是PC+4对应的指令应该识别为NOP，不执行。

那么可以看到branch指令被传入IFID之间的流水线寄存器，那么当PC+4指令进入ID时，若prebranch信号为1也就是上一条指令发生了跳转，那么在解码时，就不把他识别为任何指令，并且wreg，wmem设置为0.

具体代码为：

```
and(i_beq , ~op[5],  ~op[4],  ~op[3],   op[2],  ~op[1], ~op[0], ~prebrance);
and(i_bne , ~op[5],  ~op[4],  ~op[3],   op[2],  ~op[1],  op[0], ~prebrance);
and(i_j   , ~op[5],  ~op[4],  ~op[3],  ~op[2],   op[1], ~op[0], ~prebrance);
and(i_jal , ~op[5],  ~op[4],  ~op[3],  ~op[2],   op[1],  op[0], ~prebrance);
assign wreg    = (i_add | i_sub | i_and | i_or  | i_nor | i_xor | i_sll | i_mul |
                  i_slt | i_sltu| i_srl | i_sra | i_addi| i_andi| i_ori | i_xori|
                  i_lw  | i_lui | i_jal | i_slti) & nostall & ~prebrance; // if wr
assign wmem    =  i_sw & nostall & ~prebrance;
```

测试说明

测试方法： 仿真
测试类型： 集成
测试用例及结果：
通过lab课件的方法测出这个pipeline cpu的周期大概是单周期cpu的四分之一。
通过仿真测出pipeline cpu在测试场景1 3'b000测试用例中花费了46个周期，单周期cpu花了42个周期。（见视频）
所以speedup大概是$\frac{42*4}{46}=3.65$

问题及总结

设计了一个pipeline cpu并进行了测试，通过了基础测试场景，并对效率进行测试，发现相比单周期CPU显著地提升了效率。