流水线 CPU 实验报告

Xuan

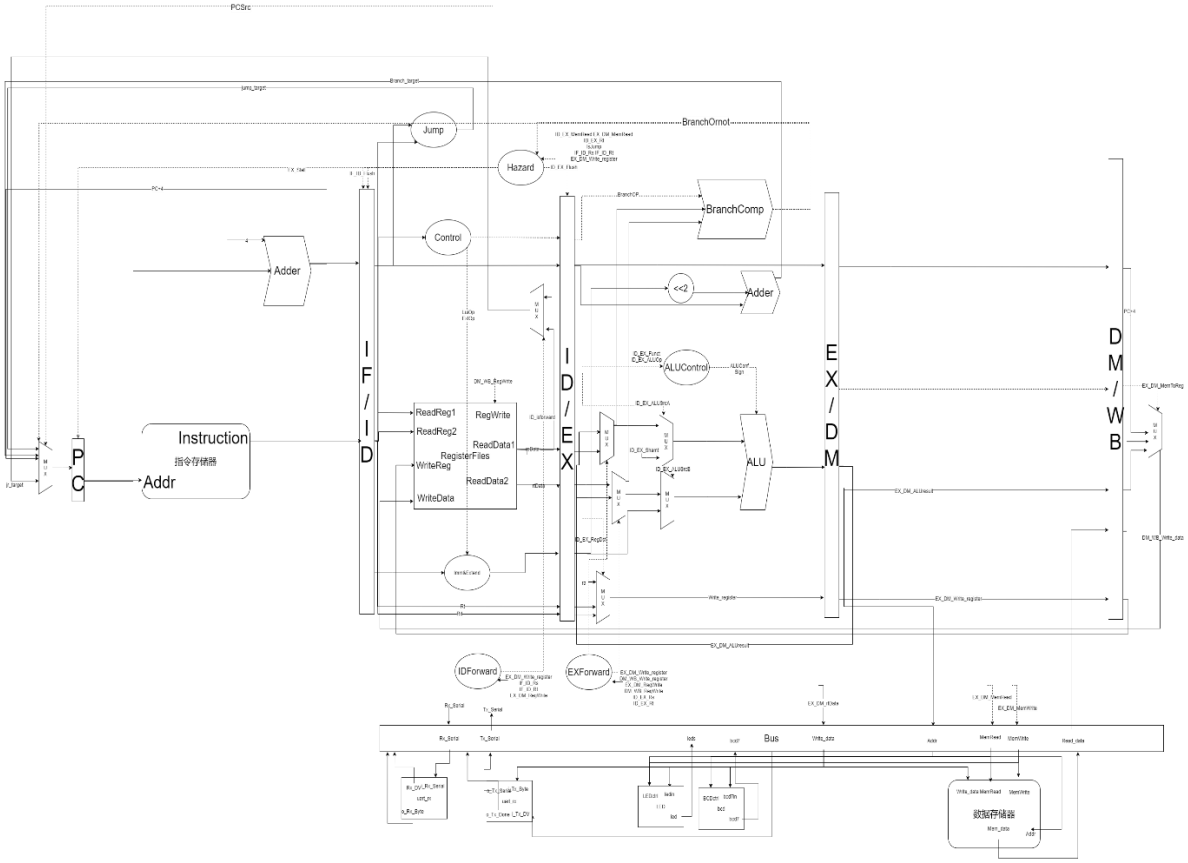
实验目的;

将春季学期实验四设计的多周期 MIPS 处理器改进为流水线结构，并利用此处理器和理论课 MIPS 汇编语言大作业中的任意一种背包问题的算法，求解背包问题

设计方案（原理说明及框图）;

总体设计

整体的数据通路如下图（可见附件 dataplane.png）



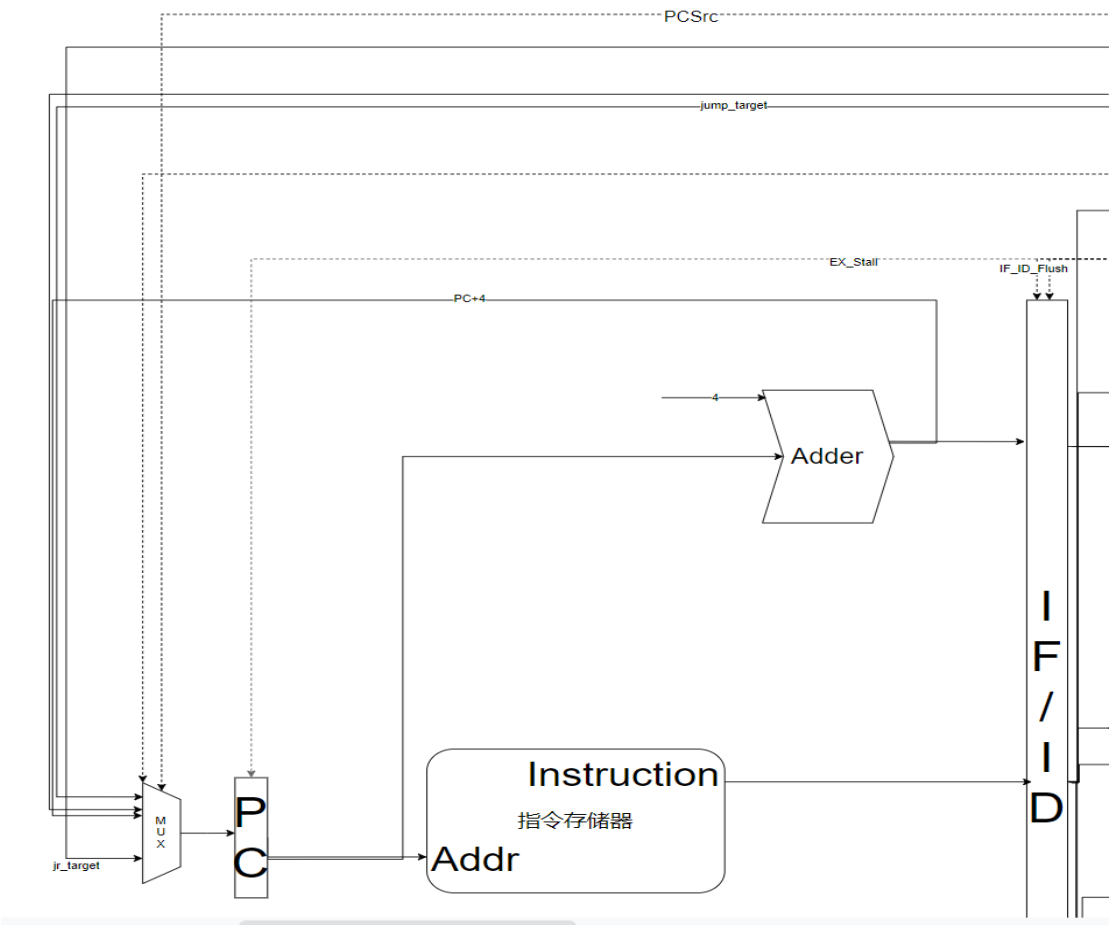
状态转移图

流水线	R类型	读存储器	写存储器	分支
IF	IR <=MemInst[PC]; PC<=PC+4			
ID	op<=IR[31:26]; A<=Reg[IR[25:21]]; B<=Reg[IR[20:16]];			
EX	ALUOut<=A op B	ALUOut<=A+sign-ext(IR[15:0])		ALUOut<=A-B PCAdd<=PC+(sign ext(IR[15:0]<<2)
MEM		MDR<=MemData[ALUOut];	MemData[ALUout] <=B	if (Zero) PC<=PCAdd
WB	Reg[IR[15:11]] <=ALUOut	Reg[IR[20:16]] <=MDR		

实现的指令:
空指令: nop (0x00000000,即 sll $0,$0,0)
存储访问指令: lw, sw, lui
算术指令: add, addu, sub, subu, addi, addiu
逻辑指令: and, or, xor, nor, andi, sll, srl, sra, slt, sltu, sltiu
分支指令(beq、bne、blez、bgtz、bltz)和 跳转指令(j、jal、jr(前一条指令不可数据冒险，否则在汇编加 nop)、jalr);
CPU 为五级流水线的设计，即分为取指令（IF)、译指令（ID)、执行（EX)、访存 （DM)、写回（WB）五个阶段。
本设计采用完全的 forwarding 电路解决数据关联问题。对于 Load-use 类竞争采取阻塞一个周期+Forwarding 的方法解决。
对于分支指令在 EX 阶段判断，在分支发生时刻取消 ID 和 IF 阶段的两条指令。
对于 J 类指令在 ID 阶段判断，并取消 IF 阶段指令。
通过在寄存器堆内部加入数据旁路的硬件方式实现寄存器实现先写后读功能。
通过在汇编层面加 nop 解决 jr 的数据冒险问题。
下面逐项介绍

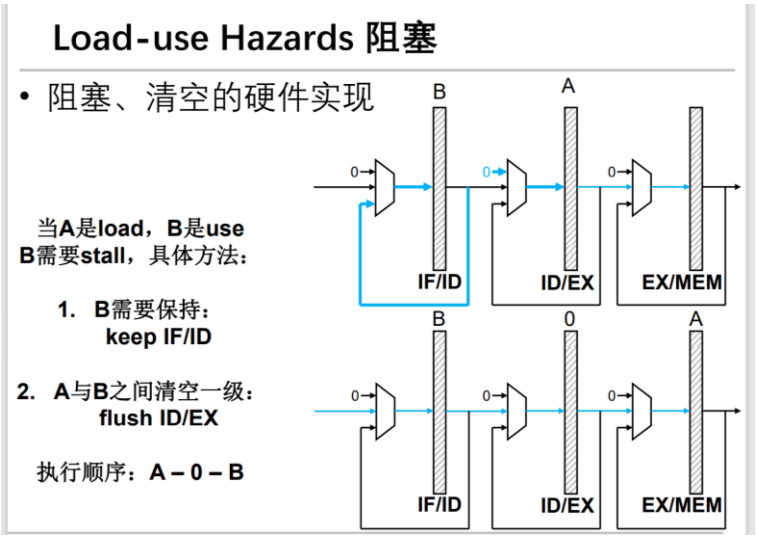
IF 阶段 .

PC 寄存器:采用 Verilog 代码编写的上升沿触发寄存器，会根据控制信号 PCSrc 和 BranchHazard 进行更新,更改为 PC+4-000(需要加法器)、jump_target-PCSrc=01、jr_target-10、branch_target-BranchHazard 中的一个,分别对应顺序执行、跳转目标、分支目标地址。且写入受控制信号 PCWrite（即~EX_Stall） 的控制。注意为了与 MARS 同步，PC 复位时设为 32'h00400000
IF_ID_reg: 为上升沿触发的 IF/ID 的级间寄存器，存储 IF 阶段取出的指令和 PC+4。若控制信号 IF_ID_Flush =1,则输出 IF_ID_Instruction、IF_ID_PC_plus_4 均为 0，否则即为这条指令 IF 阶段的指令和 PC+4。
指令存储器: 本实验采用提前把指令的十六进制机器码写入指令寄存器的方式来初始化，缺点是每次都要手动输入指令，比较麻烦。指令存储器根据 Address 提供相应的 Instruction



ID 阶段

Controller：控制单元，进行指令的译码，产生控制信号。接收指令的 OpCode 和 Funct。分控制信号写入流水线寄存器，部分直接连接到其他模块进行控制。
jump_target：在 ID 阶段要进行 jump 指令和 jr 指令跳转地址的计算
符号拓展和移位模块：根据 LuiOp 和 ExtOp 控制信号来选择是否符号拓展和移位。
LuiOp=1 时（为 lui 指令）把立即数左移 16 位；ExtOp=1 时为符号拓展，否则为零拓展。
HazardCheck：冒险检测单元，当发生 load-use 冒险时，即 ID_EX_MemRead =1（前一条指令为 lw）且 ID_EX_Rt (lw 要写入的寄存器)和 IF_ID_Rs 或 IF_ID_Rt（现在 IF_ID 阶段指令要读取的寄存器）相等时，令控制信号 EX_Stall 和 ID_EX_Flush 等于 1。即实现 keep IF/ID(PC 保持)，flush ID/EX 级间寄存器，以实现逻辑上插入一个 stall。如下图



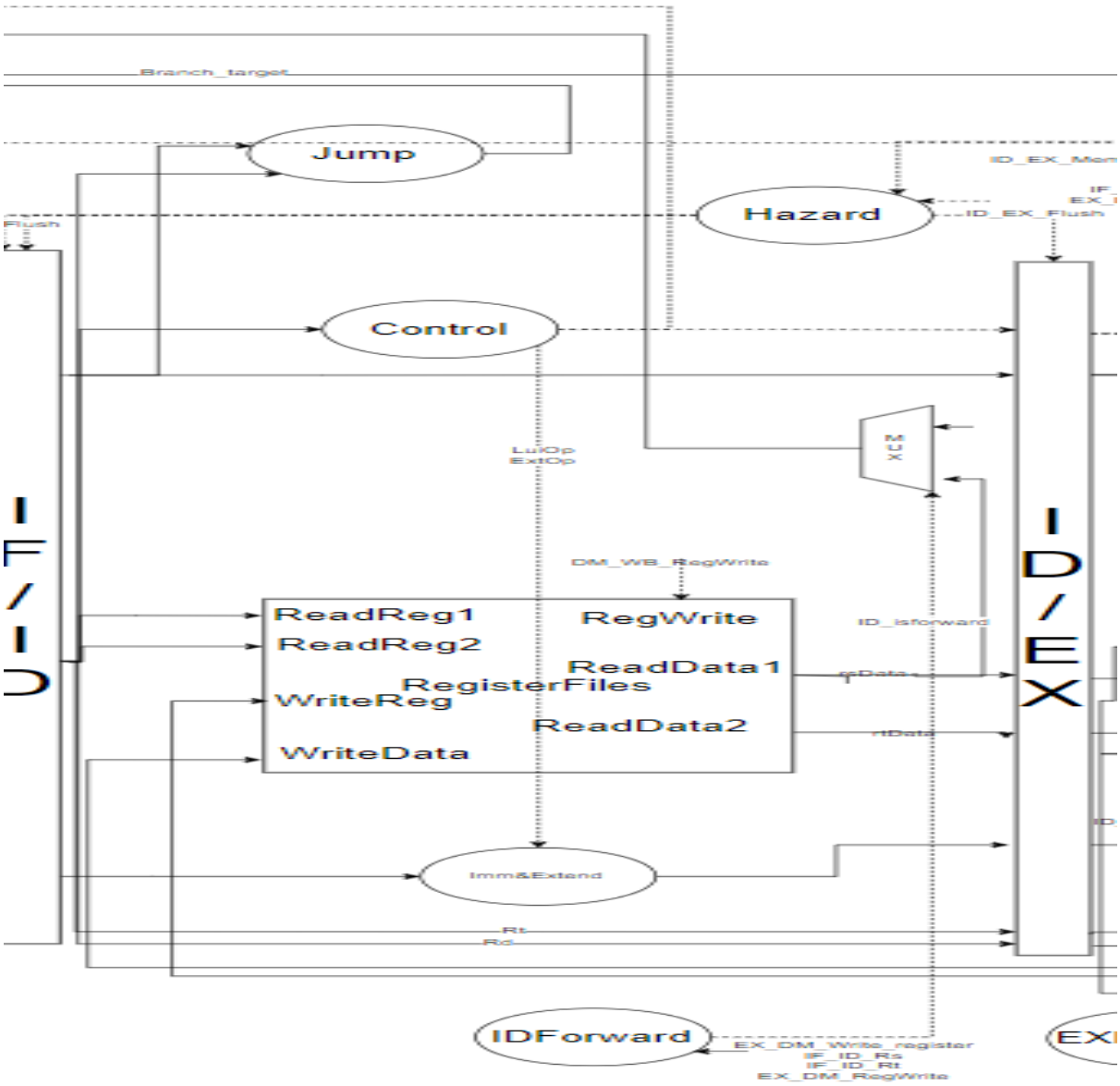
同时，在冒险检测单元会进行控制冒险 branch 和 jump 的检测。即若 ID 阶段指令为 jump, IF_ID_Flush=1(flush IF_ID),若为 Branch 指令，ID_EX_Flush 和 IF_ID_Flush=1(flush IF_ID 和 ID_EX)

寄存器堆：寄存器堆的写使能信号应为 DM/WB 阶段指令的 RegWrite，写入数据和寄存器也是 DM/WB 阶段的。;读取的两个寄存器和数据即为指令的 Rs 和 Rt，记为 rsData 和 rtData；内部通过旁路转发的形式实现先写后读。如选择读取的数据时，若读取寄存器为$0,则数据为 0；若读取的寄存器和要写入的寄存器相同且此时写使能信号为 1，则读取数据为写入的数据；否则即为寄存器堆存储的数据 RF_data。

ID 阶段转发单元：针对的是 jr 和 jalr 指令要跳转的寄存器刚好在前条或前前条指令被写入时，因为 j 型指令的跳转发生在 ID 阶段，所以需要转发到 ID 阶段。本设计只能转发 jr 和 jalr 的前前条指令是 R 型指令的情况，即 EX_DM_RegWrite=1 且 EX_DM_Write_register 不等于 0（有效）；EX_DM_Write_register（EX/DM 阶段的写入寄存器）等于 IF_ID_Rs。

若 jr 和 jalr 指令的前条指令是 R，lw 或前前条指令是 lw，需要在汇编层面加入相应条数的 nop

ID/EX 级间寄存器：为上升沿触发的 ID/EX 的级间寄存器，存储 ID 阶段指令译码得出的控制信号、PC+4 和寄存器堆读取的数据和寄存器编号。若控制信号 ID_EX_Flush =1,则输出 ID_EX_MemWrite, ID_EX_MemRead 和 ID_EX_RegWrite 均为 0；防止指令写入存储器和寄存器堆，即实现了清空 branch 指令的下两条指令



控制信号

每个控制信号所实现的具体控制功能如下：

PCSrc：PC 寄存器更新的选择信号；01-选择 jump_target; 10-选择 jr_target;else-选择 pc+4

BranchOp：Branch 比较器进行的操作的选择信号；即 BranchOrnot=1 的条件：001-进行 beq 的操作（两值相等）；010- 进行 bne 的操作(两值不相等)；011-进行 blez 的操作(In1[31] || ~(| In1))；100-进行 bgtz 的操作(~(In1[31] || (| In1)))；101-进行 bltz bgez 的操作(先根据 ID_EX_Rt 来判断是 bltz 还是 bgez，若为 bgez 则输入不为负数时跳转，若为 bltz 输入为负数时跳转)；

PCWrite：PC 寄存器的写使能信号；0-不能写 PC；1-允许写 PC。

MemWrite:指令和数据内存的写使能信号；0-不能写入 write data;1-允许写入

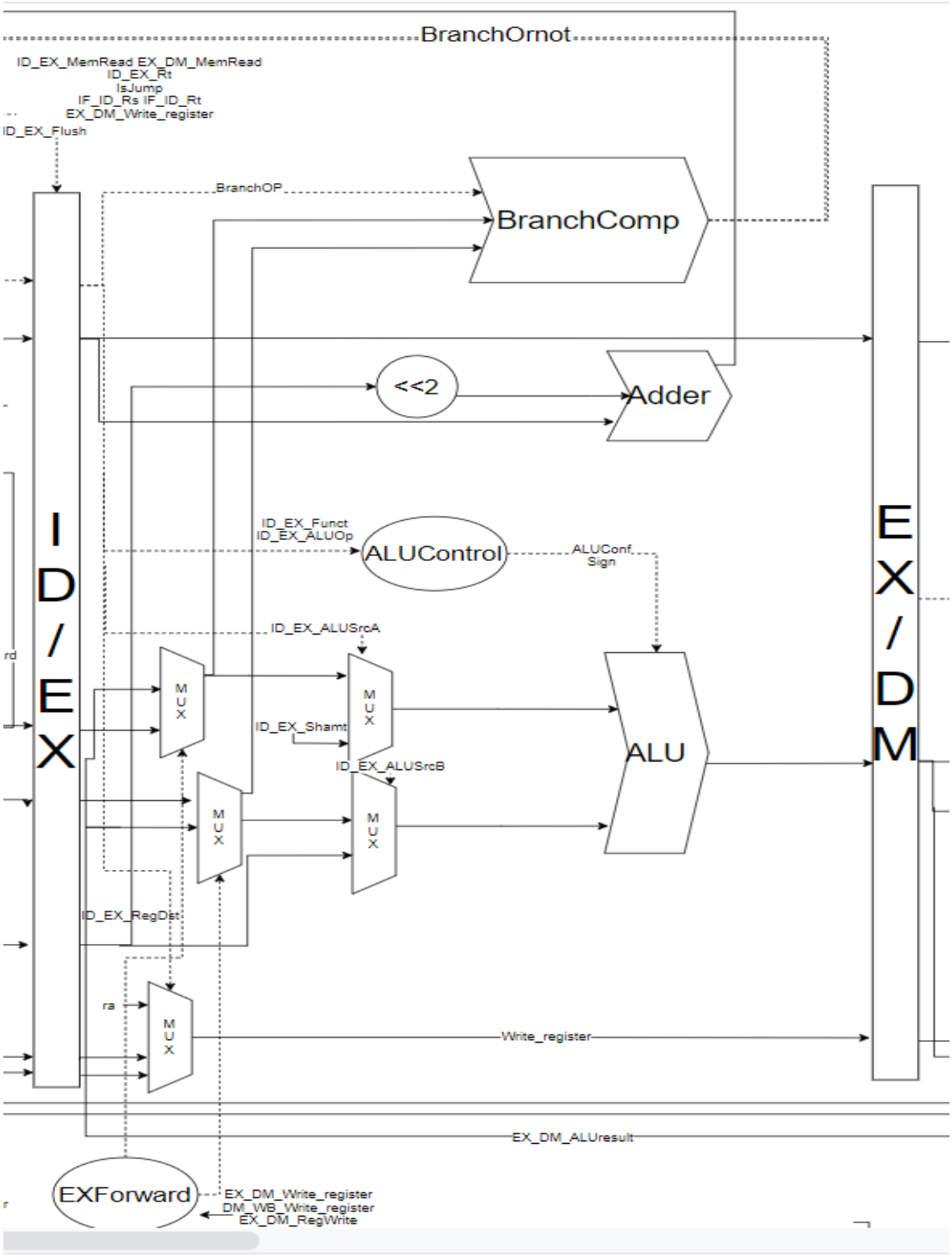
MemRead：指令和数据内存的读使能信号；0-Mem_data=0 即不能读取内存数据;1-允许读

取内存数据
MemtoReg[1:0]:控制寄存器堆的写入数据的选择信号；
10-PC+4; (jal 和 jalr 指令时为 PC+4) 01- 从 datamemory 读取的数;else--ALUOUT;
RegDst [1:0]：控制寄存器堆的写入寄存器的选择信号；00-rt;01-rd;else-$ra;
RegWrite：寄存器堆的写使能信号；0-不能写入寄存器堆；1-允许写寄存器堆。
ExtOp：符号拓展信号；1-有符号拓展；0-零拓展
LuiOp：lui 指令的移位信号；1-是 lui 指令，拓展和移位单元会输出未拓展的立即数的左移 16 位；0-不是 lui 指令，拓展和移位单元会输出拓展后的立即数
ALUSrcA:alu 第一个输入前的多路选择器的选择信号； 1：5bit 移位量的 32bit 拓展（用于 sll,sra，srl 指令）; 0-寄存器 A 的数据；
因 lui 指令的$rs=$0,默认$0 存储的值恒为 0，所以 ALUSrcA=0,无需把 0 作为输入
ALUSrcB: alu 第二个输入前的多路选择器的选择信号；1-拓展或移位后的立即数（others）0-寄存器 B 的数据（R-type）；
ALUOp[2:0]:选择 ALU 进行的操作；3'b001-aluOR（ori）;3'b011-aluXOR(xori) 3'b100-aluAND(andi);3'b101-aluSLT(slti sltiu);3'b010- aluFunct（即 R 型指令，还要根据 Funct 来确定）;else-aluADD
ALUOp[3] := OpCode[0];用于 ALUController 中判断 Sign 信号，即 Sign = (ALUOp[2:0] == 3'b010)? ~Funct[0]: ~ALUOp[3];//is R? slt slti（是 R 时根据 Funct[0]来判断） sltiu sltu（不是 R 时根据 OpCode[0]来判断）
IsJump：1-是 j/jr/jal/jalr 指令；0-不是

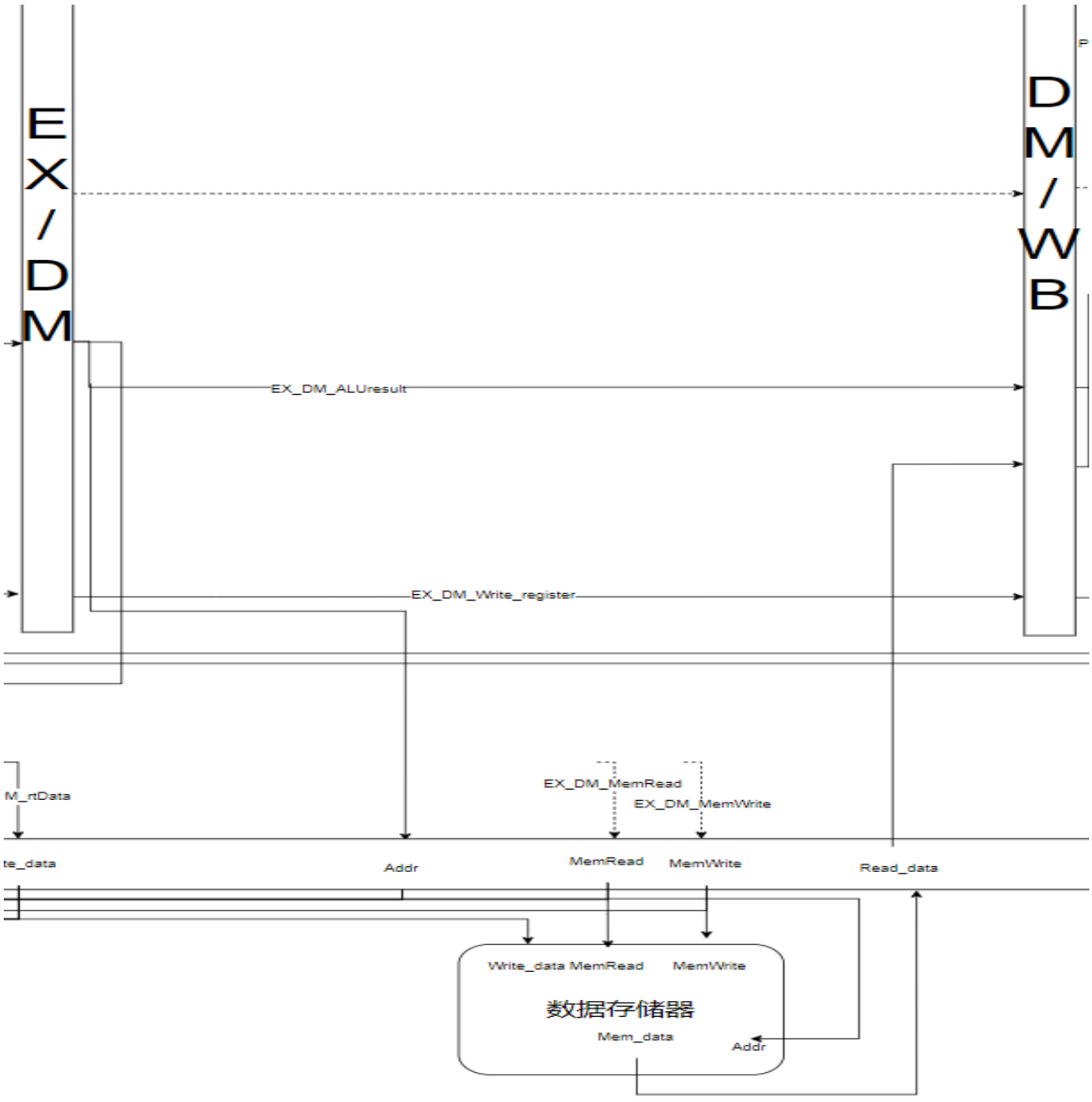
EX 阶段

ALUcontroller：接受 ID_EX 级间寄存器的控制信号（即 ALUOp 和 Funct），产生 ALU 的控制信号（ALUConf 和 Sign）。
Alu:因为 beq 的比较操作是放在 ex 阶段的 BranchComp 模块进行的，所以 alu 无需区分 beq。支持加法、减法、小于比较功能、位运算（与、或、异或、或非、左移、逻辑右移、代数右移）和溢出检测功能。
（加法存在溢出：正数与负数相加一定不会溢出；正数与正数相加（两输入符号位均为 0），结果为负数（输出符号位为 1）负数与负数相加（两输入符号位均为 1），结果为正数（输出符号位为 0）
减法存在溢出：正数减负数（In1 符号位为 0，In2 符号位为 1），结果为负数（输出符号位为 1）负数减正数（In1 符号位为 1，In2 符号位为 0），结果为正数（输出符号位为 0））
BranchComp:Branch 操作的专用比较器；因为本设计 branch 指令是在 EX 阶段跳转的，且要支持的 branch 指令的操作较多，所以需要加一个比较器。接收两个 32 位数据，根据控制信号进行等于、不等于、小于等于零、大于 等于零、小于零、大于零的判断。
Adder: 将 ID_EX 级间寄存器存储的立即数左移两位后，与存储的 PC+4 求和，得到分支目标。
EXForward：EX 阶段的转发单元。
EX/DM 级间寄存器：为上升沿触发的 EX/DM 的级间寄存器，存储 EX 阶段 ALU 计算结果、PC+4 和 DM、WB 阶段用到的控制信号（ID_EX_MemToReg、ID_EX_MemWrite、ID_EX_MemRead、ID_EX_RegWrite）和经过 EX 阶段转发单元的输出信号选择的 rt_data(即下图 ALU 第二个输入的左边的 mux 的输出信号)，选择的是 EX_DM_ALUresult（上条指令的

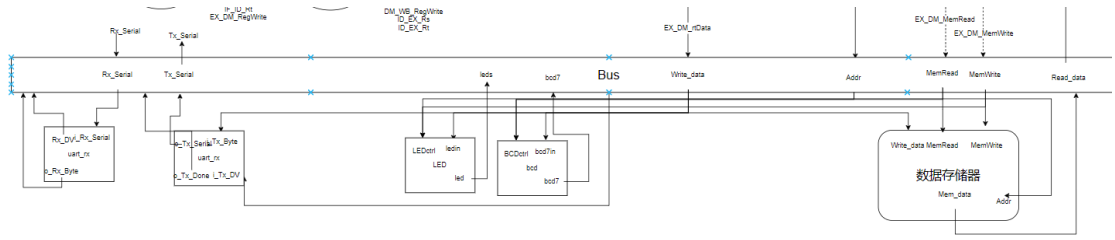
EX 阶段的结果转发到现在 ID 阶段的指令）、DM_WB_Write_data（上上条指令的 DM 阶段存储的 ALU 结果转发到现在 ID 阶段的指令）和 ID_EX_rtData，用于 lw 等指令作为数据存储器的写入数据。同时，在 EX 阶段还进行了写回寄存器堆的写入寄存器的选择（即 Rt,Rd 和$ra），所以也要存储到 EX_DM_Write_register



DM 阶段



总线



本设计根据地址访问不同外设，所以需设置不同使能信号。数据存储器是读写的信号=1 且地址在 0x00000000~0x000007FF，则可进行读写。读写的数据通过总线接受或发送到 DM/WB 级间寄存器。不用串口时数据存储器需事先进行数据的存入，使用串口则不用（全部初始化

为 0）。LED 和 bcd7 使能信号是 MemWrite=1 且为对应地址，其中 led 的输出对应 8 个 led 灯的亮灭，等于 Write_data 的低八位。对于 bcd7,总线会输出总的 bcd7（12 位），高四位作为控制四个数码管的亮灭，低七位作为七段数码管的七个输入，还有一位是 dp 信号对应小数点，本设计使其恒为 0；an 信号（即 bcd7 的高四位）另外在总线模块中用时钟分频器和四比特计数器利用人眼视觉暂留效应来轮流显示各位数字。而把 Write_data 的低 16 位和 an 作为 bcd7 模块的输入，（即根据 an 来选择 Write_data[15:0]中的四位来作为 bcd7 的四位输入），得到 7 位输出 bcd7out 作为 bcd7 的低 7 位。

外设

数据存储的地址空间被划分为 2 部分：0x00000000~0x000007FF（字节地址）为数据 RAM，可以提供数据存储功能；（本设计解决的背包问题的物品数量不超过 15，所以地址到 0x7FF 足够）0x40000000～0x7FFFFFFF（字节地址）为外设地址空间，对其地址的读写对应到相应的外设资源。具体地址划分如下：

地址（字节地址）	功能	描述
0x00000000~0x000007FF	数据存储器	512×32bits
0x4000000C	外部 LEDs	0bit：LED 0 1bit：LED 1 ..... 7bit：LED 7
0x40000010	七段数码管	0bit：CA 1bit：CB ..... 7bit：DP 8bit：AN0 9bit：AN1 10bit：AN2 11bit：AN3
0x40000018	串口发送数据 UART_TXD	串口发送数据寄存器，只有低 8bit 有效；对该地址的写操作将触发新的 UART 发送
0x4000001C	串口接收数据 UART_RXD	串口接收数据寄存器，只有低 8bit 有效
0x40000020	串口状态、控制 UART_CON	2bit：发送状态，每当 UART_TXD 中的数据发送完毕后该比特置‘1’，当执

		行对该地址的读操作后，将自动清零 3bit：接收状态，每当 UART_RXD 中已经接收到一个完整的字节时该比特置‘1’，当执行对该地址的读操作后，将自动清零 4bit：模块状态，0-发送模块处于空闲状态，1-发送模块处于发送状态

串口的控制:

若汇编指令读取了 0x4000001C 的数据，则 Read_data 会根据 UART_CON[3]的情况（每当 UART_RXD 中已经接收到一个完整的字节时该比特置'1'，当执行对该地址的读操作后，将自动清零），若 UART_CON[3]=1，则返回读取的一个完整字节，若 UART_CON[3]=0，说明还没有接收完串口的数据，返回一个标示的负数，用于在汇编层面进行不断读取的循环，直至返回的 Read_data 不再是负数，即返回了完整的一个字节。其中 UART_CON[3]是根据 Rx_DV 来进行置 1 的。因为在接收到一个完整字节后，Rx_DV 会有一个时钟周期的高电平，所以控制如下图。对于 UART_CON[2]也是同理，根据 Tx_Done 来置 1。UART_CON[4]是模块状态，可根据 Tx_Active 的状态进行调整。即发送模块处于发送状态时 Tx_Active=1。若接收到读取 0x40000020 的指令，则 UART_CON[2]和 UART_CON[3]都清零。在汇编层面上，每次读取了 0x4000001C 的数据后都要有一条读取 0x40000020 的指令来进行清零，除了最后一条存进的指令。

```
UART_CON[4] = Tx_Active;
if (UART_CON_read) UART_CON[3:2] <= 2'b00;
else begin
    if(Rx_DV) UART_CON[3]=1'b1;//recv done（最后一条存进的指令不会跟清空UART_CON的指令）
    if(Tx_Done) UART_CON[2]=1'b1;
end
```

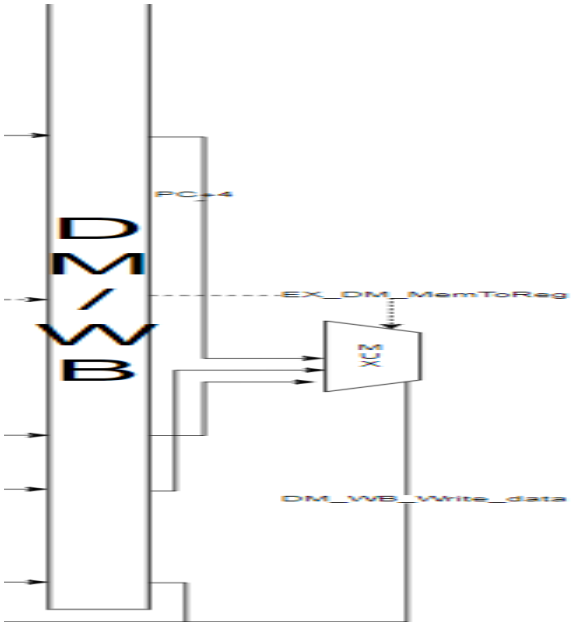
```
main:
la $s1 0x4000001C#串口接收数据UART_RXD
la $s2 0x40000020#清零UART_CON
la $s0 in_buff#存进datamemory

U_RECV1:lw $t0,0($s1) #串口接收数据UART_RXD
bltz $t0, U_RECV1
lw $t1,0($s2) #清零UART_CON
sw $t0 0($s0) #存最大重量进datamemory
U_RECV2:lw $t2,0($s1) #串口接收数据UART_RXD
bltz $t2, U_RECV2
lw $t1,0($s2) #清零UART_CON
sw $t2,4($s0) #存item_num进datamemory,t2=item_num,来判断何时结束读取
addi $s0,$s0,8
add $t3,$t2,$t2#t3=2* item_num

UART: lw $t0,0($s1) #串口接收数据UART_RXD
bltz $t0, UART
sw $t0 0($s0) #存进datamemory
subi $t3 $t3 1#还要存的量减一
beq $t3,$0,MAIN
addi $s0,$s0,4#地址加一
lw $t1,0($s2) #清零UART_CON
j UART
```

WB 阶段 .

本阶段不产生任何信号，仅在时钟上升沿根据控制信号进行写回寄存器堆的操作，同时会进行写回数据的选择（根据 EX/DM_MemToReg 选择 MemData、EX_DM_ALUresult 和 EX_DM_PC_4）



冒险和转发的方案

Jr/jalr:前一条指令要写入且写入的寄存器与跳转的寄存器相同，（若为 R）需要在汇编层面加一个 nop，（若为 lw）需要在汇编层面加两个 nop
前前条指令若为 R，可通过 IDForward 把 EX_DM_ALUresult 转发到 ID 阶段；若为 lw， 需要在汇编层面加一个 nop
Branch:前条指令为 R 或前前条指令为 R/lw，因为 branch 是在 EX 阶段比较，可直接通过 EXForward 转发，前条指令为 lw 时，HazardCheck 模块检测到之后会插入一个 stall，然后再转发

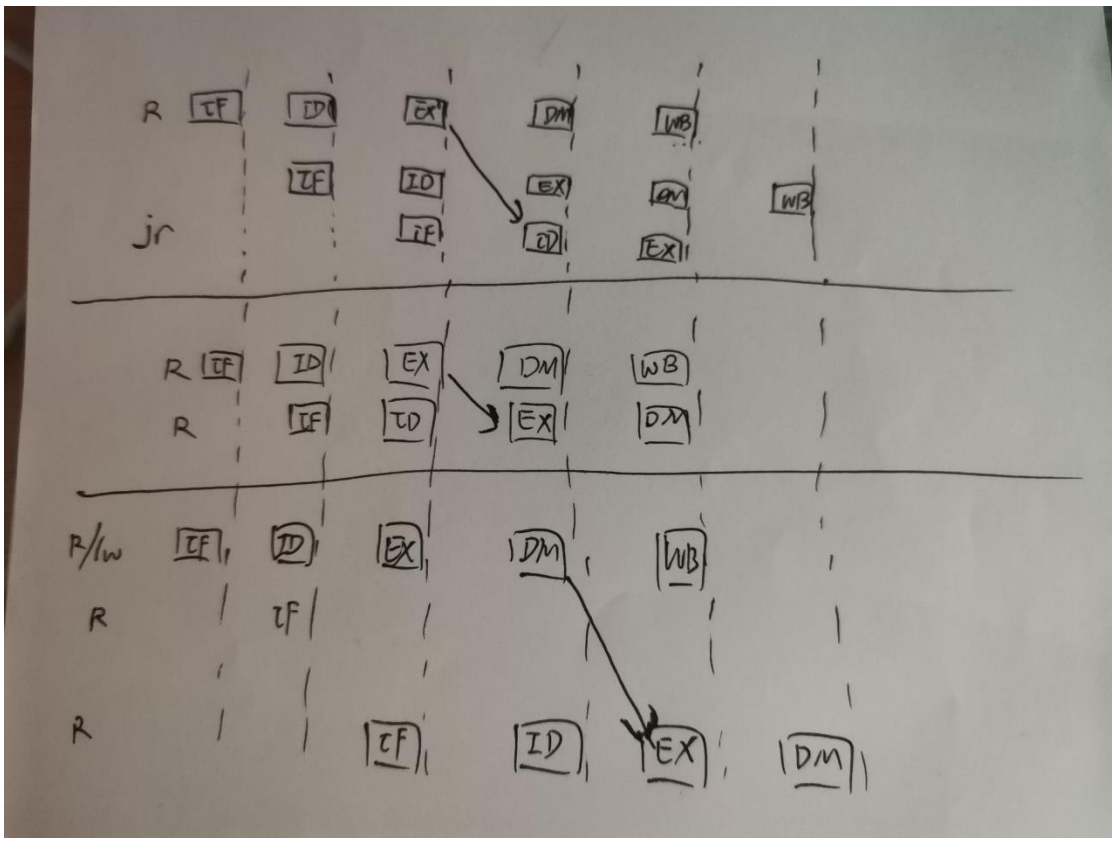
Load-use 冒险：某指令（非 jr/jalr）前条指令为 lw,lw 写入寄存器与该指令读取的寄存器相同，则先是 HazardCheck 模块检测到 ID_EX_MemRead &&(ID_EX_Rt == IF_ID_Rs ||ID_EX_Rt == IF_ID_Rt)；即该指令的 IF 结束 ID 开始插入一个 stall(Keep IF/ID;keep PC;Flush ID/EX)，此时 lw 变成前前条指令。当指令 ID 结束 EX 开始时，lw 指令处于 DM 结束 WB 开始，EXForward 模块检测到数据冒险后把 DM_WB_Write_data（MemData）转发到 ex 阶段

寄存器实现先写后读功能：在寄存器堆内部加入数据旁路，若写和读同一寄存器，直接旁路转发，但会延长读取寄存器的关键路径长度。

本设计共有两个转发模块，对于 ID 阶段的转发，对应 jr 和 jalr 的前前条指令是 R 型，如果现在处于 IF/ID 阶段的指令的 Rs 和 EX/DM 阶段的写回寄存器相等，且 EX/DM 阶段的写回寄存器不为 0，且 EX/DM 阶段的写使能信号为高，则说明该 R 指令需要对下面 jr 指令要读取的寄存器进行写回操作，所以要把 R 指令 EX 阶段的计算结果转发到 ID 阶段。Jr 和 jalr 的前条指令是 R 或 lw 以及前前条指令是 lw 的情况已说明。
对于 EX 阶段的转发，分为 EX_Forward_1 和 EX_Forward_2，对应于要转发到 EX 阶段 ALU 的两个输入。分为两种情况，一是前一条指令是 R 指令，要写回的寄存器是下一条指令要读取的寄存器，即 ID/EX 阶段的 Rs 或 Rt 和 EX/DM 阶段的写回寄存器相等，且 EX/DM 阶段的写回寄存器不为 0，且 EX/DM 阶段的写使能信号为高，则要把 EX/DM 阶段的 ALU 的计算结果转发到 EX 阶段作为 ALU 的输入。
二是前前条指令是 R 指令或 lw，要写回的寄存器是下一条指令要读取的寄存器，即 ID/EX 阶段的 Rs 或 Rt 和 DM/WB 阶段的写回寄存器相等，且 DM/WB 阶段的写回寄存器不为 0，且 DM/WB 阶段的写使能信号为高，这里涉及到要转发的是 ALU 的计算结果（R 指令）还是 MemData（lw）的问题,本设计直接转发经过多路选择器的 DM_WB_Write_data。如图

```
assign Write_data = (EX_DM_MemToReg == 2'b10) ?EX_DM_PC_4 :
(EX_DM_MemToReg == 2'b01) ? MemData :EX_DM_ALUresult;
```

数据冒险情况如下图



关键代码及文件清单;

文件清单

源代码均位于 code 文件夹中,其中 serial 文件夹内为串口的接受和发送模块

core

- configs
  - CPU.xdc --约束文件
- bus –外设文件夹
  - bcd7.v –七段数码管
  - LED.v --LED
  - clk_divider1.v –系统时钟分频器
  - fourbitcter.v –4bit 计数器，用于轮流显示
  - DataMemory_uart.v –使用串口传输的数据存储器
  - DataMemory.v –装有测试样例 1 的数据存储器（11 个物品）
  - DataMemory1.v –装有测试样例 2 的数据存储器（5 个物品）
- Tests
  - CPU_tb.v –仿真文件
  - 测试汇编代码.asm –使用串口传输的汇编代码
  - 测试汇编代码 1.asm –不用串口传输的汇编代码
  - Test.dat -- Mars 测试用读入文件

Bag.txt--不用串口传输的 16 进制码

bag_uart.txt—使用串口传输的 16 进制码

serial

uart_rx.v –接收串口数据模块

uart_tx.v -FPGA 发送到串口模块

cpu.v --流水线 cpu 顶层设计

ALU.v –ALU

ALUcontroller.v - ALU 控制信号生成单元

BranchComp.v –分支指令比较器

Bus.v 总线控制外设模块

Controller.v --控制信号生成单元

EXForward.v -- EX 阶段的转发单元

HazardCheck.v –冒险检测单元

IDForward.v -- ID 阶段的转发单元

PC_reg.v –PC 寄存器

RegisterFiles.v—寄存器堆

InstructionMemory.v –不用串口传输的指令存储器

InstructionMemory1.v –使用串口传输的指令存储器

关键代码

PC 的更新及 PC 寄存器（PCWrite 控制）

```
assign PCWrite = ~EX_Stall;
assign PC_next = BranchOrnot ? branch_target :
        (PCSrc == 2'b00) ? PC_4 :
        (PCSrc == 2'b01) ? jump_target :
        (PCSrc == 2'b10) ? jr_target :PC_4;
PC_reg PC1(.reset(reset),.clk(clk),.PCWrite(PCWrite),.PC_i(PC_next),.PC_o(PC));
```

ID 阶段转发

```
assign ID_isForward =(EX_DM_RegWrite &&(EX_DM_Write_register != 5'h0) &&(EX_DM_Write_register == IF_ID_Rs));
//last last of jr and jalr is R
assign rs_data_forward_id =(ID_isForward) ? EX_DM_ALUresult:rsData;
```

EX 阶段转发

```
// 01 from EX/DM  10 from DM/WB
assign EX_Forward_1 = (EX_DM_RegWrite &&(EX_DM_Write_register != 5'h00) &&(EX_DM_Write_register == ID_EX_Rs))? 2'b01://last R
(DM_WB_RegWrite &&(DM_WB_Write_register != 5'h00) &&(DM_WB_Write_register == ID_EX_Rs))? 2'b10//last last R or lw
: 2'b00;
assign EX_Forward_2 =(EX_DM_RegWrite &&(EX_DM_Write_register != 5'h00) &&(EX_DM_Write_register == ID_EX_Rt))? 2'b01://last R
(DM_WB_RegWrite &&(DM_WB_Write_register != 5'h00) &&(DM_WB_Write_register == ID_EX_Rt))? 2'b10//last last R or lw
: 2'b00;
```

IF/ID 级间寄存器

```
//IF/ID reg
wire IF_ID_Flush;
reg [31:0]IF_ID_Instruction,IF_ID_PC_4;
always @(posedge clk or posedge reset)
  begin
    if (reset)
      beginD
        IF_ID_Instruction <= 32'h0;
        IF_ID_PC_4 <= 32'h0;
      end
    else if(~EX_Stall)
      begin
        IF_ID_Instruction <= IF_ID_Flush ? 32'h0 : Instruction;
        IF_ID_PC_4 <= PC_4;
      end
  end
```

ID/EX 级间寄存器

```
  ID_EX_PC_4 <= IF_ID_PC_4;
  ID_EX_rsData <= rs_data_forward_id;
  ID_EX_rtData <= rtData;
  ID_EX_ExtendOrShift_Imm <= ExtendOrShift_Imm;
  ID_EX_Rs <= IF_ID_Instruction[25: 21];
  ID_EX_Rt <= IF_ID_Instruction[20: 16];
  ID_EX_Rd <= IF_ID_Instruction[15: 11];
  ID_EX_Shamt <= IF_ID_Instruction[10:6];
  ID_EX_Funct <= IF_ID_Instruction[5:0];
  ID_EX_ALUOp <= ALUOp;
  ID_EX_BranchOp <= ID_BranchOp;
  ID_EX_ALUSrcA <= ALUSrcA;
  ID_EX_ALUSrcB <= ALUSrcB;
  ID_EX_RegDst <= ID_RegDst;
  ID_EX_MemToReg <= MemToReg;
  ID_EX_MemWrite <= ID_EX_Flush ? 1'b0 : MemWrite;
  ID_EX_MemRead <= ID_EX_Flush ? 1'b0 : MemRead;
  ID_EX_RegWrite <= ID_EX_Flush ? 1'b0 : RegWrite;

assign rs_data_forward_ex = (EX_Forward_1 == 2'b01) ? EX_DM_ALUresult :
(EX_Forward_1 == 2'b10) ? DM_WB_Write_data:
ID_EX_rsData;
assign rt_data_forward_ex = (EX_Forward_2 == 2'b01) ? EX_DM_ALUresult :
(EX_Forward_2 == 2'b10) ? DM_WB_Write_data :
ID_EX_rtData;
```

J 型指令跳转地址的选择

```
assign jump_target = {IF_ID_PC_4[31: 28], IF_ID_Instruction[25: 0], 2'b00};
```

```
assign jr_target = rs_data_forward_id;
```

立即数拓展与移位

```
//extend and shift
wire [31: 0] ExtendOrShift_Imm;
assign ExtendOrShift_Imm = LuiOp?{IF_ID_Instruction[15 : 0], 16'b0}:
(ExtOp ?{{17{IF_ID_Instruction[15]}}, IF_ID_Instruction[14 : 0]}: {16'b0, IF_ID_Instruction[15 : 0]});
```

冒险检测单元

```
// Load Use Hazard
wire EX_Stall, DM_Stall; // EX_Stall for lw, and DM_Stall for lw -> jr/branch
assign EX_Stall = reset ? 1'b0 :
ID_EX_MemRead &&(ID_EX_Rt == IF_ID_Rs ||ID_EX_Rt == IF_ID_Rt);//current is ID ,last is lw
//控制冒险beq j
assign IF_ID_Flush = reset ? 1'b0 : (IsJump || BranchOrnot);
assign ID_EX_Flush = reset ? 1'b0 : BranchOrnot||EX_Stall;
```

寄存器堆硬件方式实现先写后读（旁路转发）

```
//read data
assign Read_data1 = (Read_register1 == 5'b00000)? 32'h00000000:
(Write_register == Read_register1&&RegWrite)?Write_data:RF_data[Read_register1];
assign Read_data2 = (Read_register2 == 5'b00000)? 32'h00000000:
(Write_register == Read_register2&&RegWrite)?Write_data:RF_data[Read_register2];
```

写回数据选择

```
assign Write_data = (EX_DM_MemToReg == 2'b10) ?EX_DM_PC_4 :
(EX_DM_MemToReg == 2'b01) ? MemData :EX_DM_ALUresult;

assign Write_register = ID_EX_RegDst == 2'b01 ? ID_EX_Rd :
ID_EX_RegDst == 2'b00 ? ID_EX_Rt: 5'd31 ;
```

ALU 输入数据选择

```
assign ALUIn1 = ID_EX_ALUSrcA ? ID_EX_Shamt :rs_data_forward_ex;
assign ALUIn2 = ID_EX_ALUSrcB ? ID_EX_ExtendOrShift_Imm : rt_data_forward_ex;
```

总线与外设

```
assign EN_DataMemory = Address <= 32'h000007ff&&Address>=32'h00000000;
wire [31:0] Mem_data;
DataMemory DataMemory1(.reset(reset), .clk(clk), .Address(Address),
 Write_data(Write_data), .MemRead(MemRead && EN_DataMemory), .MemWrite(MemWrite && EN_DataMemory), .Mem_data(Mem_data));
//LED
LED LED1(.clk(clk),.reset(reset),.LEDctrl(MemWrite && Address == 32'h4000000c),.ledin(Write_data[7:0]),.led(leds));
// bcd
wire [6:0]bcd7out;
bcd7 bcd7_1(.clk(clk),.reset(reset),.bcd7ctrl(MemWrite && Address == 32'h40000010),.bcd7in(Write_data[15:0]),.an(an),.dout(bcd7out));
assign bcd7={an,1'b0,bcd7out};
```

串口发送与接受控制

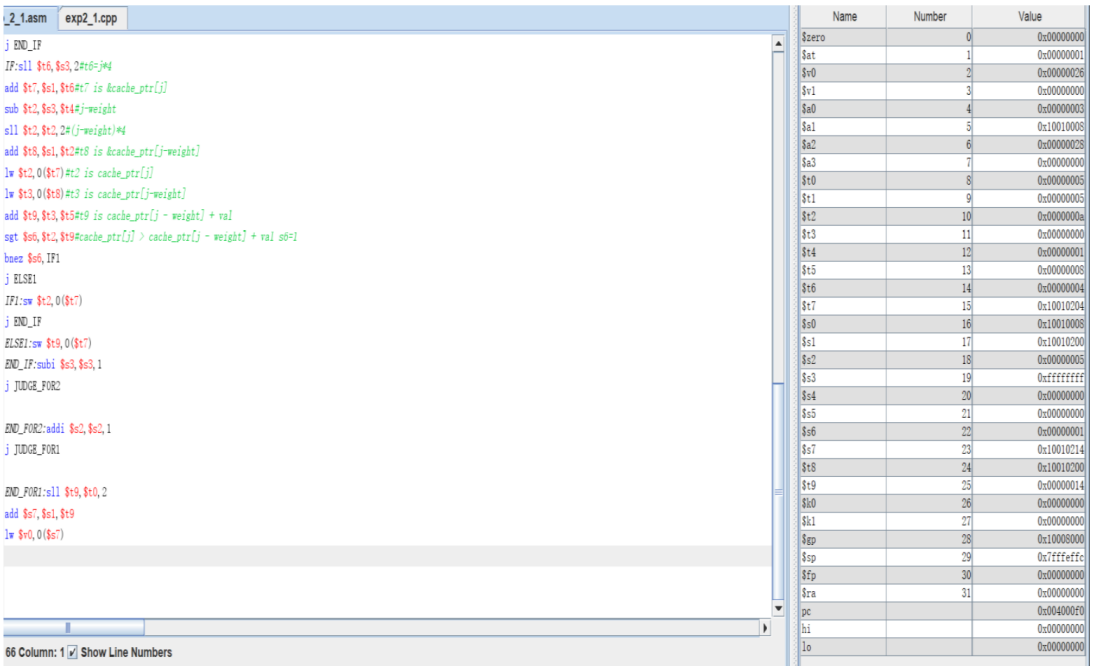
```
always@(posedge reset or posedge clk)
begin
        if(reset) UART_CON <= 32'b0;
        else begin
            UART_CON[4] = Tx_Active;
            if (UART_CON_read) UART_CON[3:2] <= 2'b00;
            else begin
                if(Rx_DV) UART_CON[3]=1'b1;//recv done (最后一条存进的指令不会跟清空UART_CON的指令)
                if(Tx_Done) UART_CON[2]=1'b1;
            end
        end
end
```

仿真结果及分析;

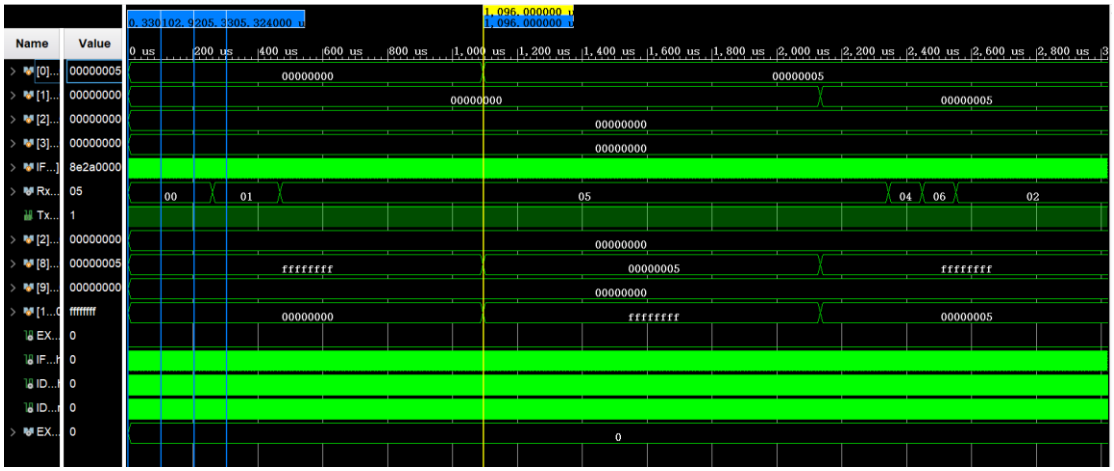
使用串口的前仿如下图，其中模拟串口发送给 FPGA 的数据为汇编大作业的样例，即 5（最大重量）,5（物品数量），2（第一个物品重量）,12（第一个物品价值），1,10，3,20，2,15，1,8。(使用的代码为第一种实现)。理论输出应为 0x26=38。

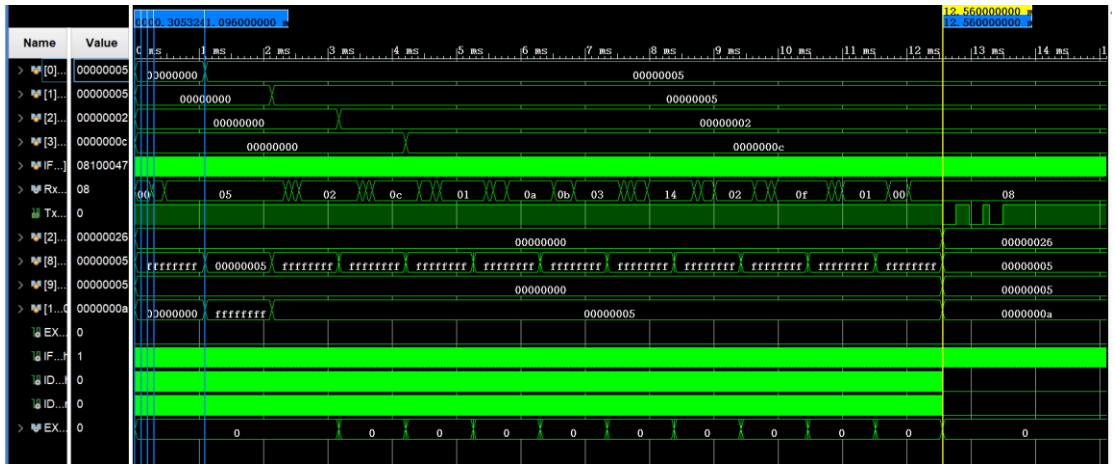
```
initial begin
    data[0]<=10'b1000001010;//最大重量为5
    data[1]<=10'b1000001010;//item_num=5
    data[2]<=10'b1000000100;//第一个物品重量2
    data[3]<=10'b1000011000;//第一个物品价值12
    data[4]<=10'b1000000010;//1
    data[5]<=10'b1000010100;//10
    data[6]<=10'b1000000110;//3
    data[7]<=10'b1000101000;//20
    data[8]<=10'b1000000100;//2
    data[9]<=10'b1000011110;//15
    data[10]<=10'b1000000010;//1
    data[11]<=10'b1000010000;//8
    r_Clock_Count<=16'b0;
    r_Bit_Index<=4'b0;
    i<=4'b0;
```

MARS 的仿真结果如下



如下图，约 1000us 开始读取数据，存到 RAM_data[0]-RAM_data[11]；（仿真图形前四个为 RAM_data 的前四个），第五个为 IF_ID_instruction(即当前处于 IF/ID 阶段的指令)；第六个为 Rx_serial 即 FPGA 接受串口数据的端口的数据传输情况，Tx_serial 为 FPGA 发送到串口的端口的数据情况，接下来的四个为寄存器堆的 RF_data[2](即$v0), RF_data[8] (即$t0), RF_data[9] (即$t1), RF_data[10] (即$t2),最后的几个信号则为冒险单元和转发单元给出的控制信号。可见仿真最终结果与 mars 仿真结果相符，且 Tx_serial 发送 0x26 给串口





如上图，执行完所有指令共用时约 13.5ms

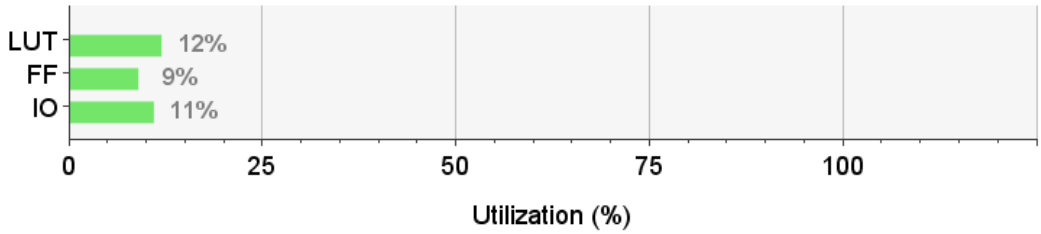
综合情况（面积和时序性能）;

针对 100MHz 时钟进行综合和实现，得到性能概览如图

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP	Start
synth_1	constrs_1	Synthesis Out-of-date								2521	3715	0.00	0	0	6/24/21, 7:20 PM
impl_1	constrs_1	Implementation Out-of-date	0.266	0.000	0.094	0.000	0.000	0.126	0	2525	3715	0.00	0	0	6/24/21, 7:21 PM

逻辑使用情况见表

Resource	Utilization	Available	Utilization %
LUT	2525	20800	12.14
FF	3744	41600	9.00
IO	24	210	11.43



总体上实现使用了近 12% 的 LUT 资源和近 9% 的寄存器资源。 一个较为复杂的 5 级流水线处理器相对于多周期处理器使用的资源更少，这体现出了流水线处理器的优势。
进一步查看各层次使用的资源，使用情况见下表，总体使用了 2525 个 LUT 和 3744 个寄存器，使用资源最多的是总线，其次是寄存器堆，可进一步减少这几处冗余的资源以优化 FPGA 的资源

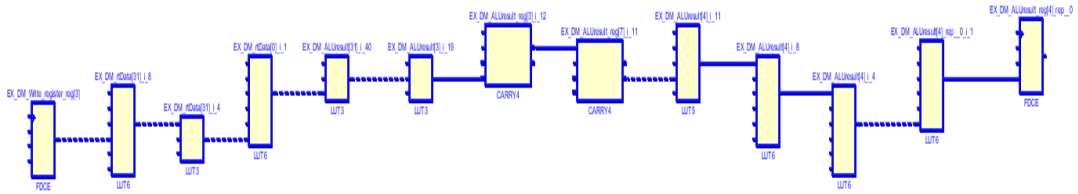
Name	Slice LUTs (20800)	Bonded IOB (210)	BUFGCTRL (32)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)
CPU	2525	24	1	3744	178	76	1606	2525
bus (Bus)	1012	0	0		44	12	923	1012
bcd7_1 (bcd7)	10	0	0		0	0	7	10
DataMemory1 (DataMemory)	894	0	0		44	12	874	894
fourbitcter1 (fourbitcter)	3	0	0		0	0	1	3
hd1 (clk_divider1)	16	0	0		0	0	10	16
LED1 (LED)	7	0	0		0	0	8	7
uart_rx_inst (uart_rx)	42	0	0		0	0	18	42
uart_tx_inst (uart_tx)	36	0	0		0	0	17	36
InstructionMemory1 (InstructionMemory)	15	0	0		0	0	22	15
PC1 (PC_reg)	180	0	0		6	0	82	180
RegisterFiles1 (RegisterFiles)	731	0	0		128	64	499	731

时序性能如下图，实现后的时序裕量为 0.266ns，则该 CPU 的主频为$\frac{1}{10-0.266}=102.74MHz$。
相比多周期处理器快了近两倍。

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.266 ns	Worst Hold Slack (WHS):	0.094 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	7082	Total Number of Endpoints:	7082	Total Number of Endpoints:	3714

All user specified timing constraints are met.

最长路径的电路和细节见下图。这一条路径是从 EX/DM 的 ALU 计算结果到 EX/DM 阶段的数据存储器的写入数据（rtData）最后到 EX/DM 阶段的写回寄存器，与理论相符



Summary

Name	Path 1
Slack	0.266ns
Source	EX_DM_Write_register_reg[3]/C (rising edge-triggered cell FDCE clocked by CLK {rise@0.000ns fall@5.000ns period=10.000ns})
Destination	EX_DM_ALUresult_reg[4]_rep__0/D (rising edge-triggered cell FDCE clocked by CLK {rise@0.000ns fall@5.000ns period=10.000ns})
Path Group	CLK
Path Type	Setup (Max at Slow Process Corner)
Requirement	10.000ns (CLK rise@10.000ns - CLK rise@0.000ns)
Data P...Delay	9.736ns (logic 2.797ns (28.730%) route 6.939ns (71.270%))
Logic Levels	11 (CARRY4=2 LUT3=3 LUT5=1 LUT6=5)
Clock ... Skew	-0.040ns
Clock U...tainty	0.035ns

Source Clock Path

Delay Type	Incr (ns)	Path...	Location	Netlist Resource(s)
(clock CLK rise edge)	(r) 0.000	0.000		
	(r) 0.000	0.000	Site: P17	clk
net (fo=0)	0.000	0.000		clk
			Site: P17	clk_IBUF_inst/I
IBUF (Prop_ibuf_I_O)	(r) 1.478	1.478	Site: P17	clk_IBUF_inst/O
net (fo=1, routed)	1.972	3.450		clk_IBUF
			Site: BU...RL_X0Y0	clk_IBUF_BUFG_inst/I
BUFG (Prop_bufg_I_O)	(r) 0.096	3.546	Site: BU...RL_X0Y0	clk_IBUF_BUFG_inst/O
net (fo=3713, routed)	1.568	5.114		clk_IBUF_BUFG
FDCE			Site: SLICE_X50Y38	EX_DM_Write_register_reg[3]/C

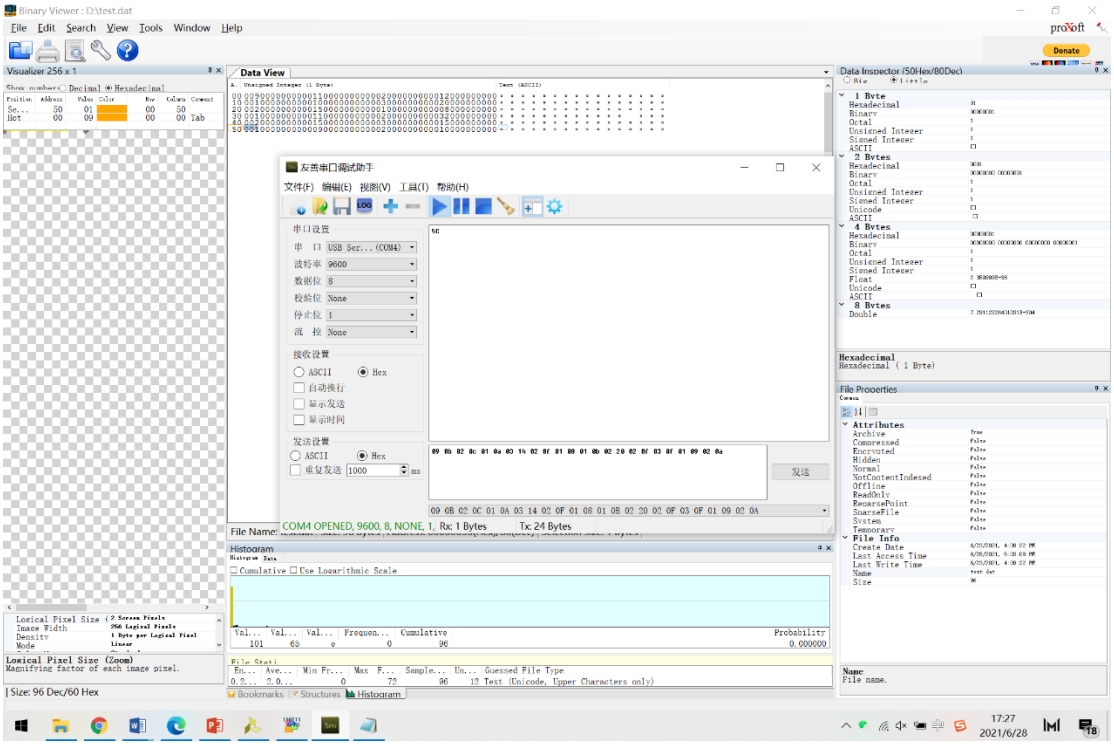
Data Path

Delay Type	Incr (ns)	Path ...	Location	Netlist Resource(s)
FDCE (Prop_fdce_C_Q)	(r) 0.518	5.632	Site: SLICE_X50Y38	EX_DM_Write_register_reg[3]/Q
net (fo=8, routed)	1.017	6.650		EX_DM_Write_register[3]
			Site: SLICE_X48Y39	EX_DM_rtData[31]_i_8/I3
LUT6 (Prop_lut6_I3_O)	(r) 0.124	6.774	Site: SLICE_X48Y39	EX_DM_rtData[31]_i_8/O
LUT5 (Prop_lut5_I3_O)	(r) 0.299	12.500	Site: SLICE_X58Y38	EX_DM_ALUresult[4]_i_11/O
net (fo=1, routed)	0.610	13.111		EX_DM_ALUresult[4]_i_11_n_0
			Site: SLICE_X58Y38	EX_DM_ALUresult[4]_i_8/I0
LUT6 (Prop_lut6_I0_O)	(r) 0.124	13.235	Site: SLICE_X58Y38	EX_DM_ALUresult[4]_i_8/O
net (fo=1, routed)	0.444	13.679		EX_DM_ALUresult[4]_i_8_n_0
			Site: SLICE_X58Y38	EX_DM_ALUresult[4]_i_4/I0
LUT6 (Prop_lut6_I0_O)	(r) 0.124	13.803	Site: SLICE_X58Y38	EX_DM_ALUresult[4]_i_4/O
net (fo=3, routed)	0.923	14.726		EX_DM_ALUresult[4]_i_4_n_0
			Site: SLICE_X56Y36	EX_DM_ALUresult[4]_rep__0_i_1/I5
LUT6 (Prop_lut6_I5_O)	(r) 0.124	14.850	Site: SLICE_X56Y36	EX_DM_ALUresult[4]_rep__0_i_1/O
net (fo=1, routed)	0.000	14.850		EX_DM_ALUresult[4]_rep__0_i_1_n_0
FDCE			Site: SLICE_X56Y36	EX_DM_ALUresult_reg[4]_rep__0/D
Arrival Time		14.850		

Destination Clock Path

Delay Type	Incr (ns)	Path ...	Location	Netlist Resource(s)
(clock CLK rise edge)	(r) 000	10.000		
	(r) 0.000	10.000	Site: P17	clk
net (fo=0)	0.000	10.000		clk
			Site: P17	clk_IBUF_inst/I
IBUF (Prop_ibuf_I_O)	(r) 1.408	11.408	Site: P17	clk_IBUF_inst/O
net (fo=1, routed)	1.868	13.276		clk_IBUF
			Site: BU...RL_X0Y0	clk_IBUF_BUFG_inst/I
BUFG (Prop_bufg_I_O)	(r) 0.091	13.367	Site: BU...RL_X0Y0	clk_IBUF_BUFG_inst/O
net (fo=3713, routed)	1.448	14.815		clk_IBUF_BUFG
FDCE			Site: SLICE_X56Y36	EX_DM_ALUresult_reg[4]_rep__0/C
clock pessimism	0.259	15.074		
clock uncertainty	-0.035	15.039		
FDCE (Setup_fdce_C_D)	0.077	15.116	Site: SLICE_X56Y36	EX_DM_ALUresult_reg[4]_rep__0
Required Time		15.116		

硬件调试情况；



思想体会

本次实验代码量较大，整体比较复杂，我大约耗时三四天做完，过程还算比较顺利。这是因为我在实际写代码时已经做好了整体的规划，包括画出了整个流水线的数据通路、设计了各个阶段的功能，以及各种冒险和数据关联问题的解决方案。但期间因为对流水线 CPU 的各种细节认知混淆，也在调试过程出现了许多 bug，总体来说，遇到的难点有以下：

1. 与多周期大作业相比，流水线的设计原理不同，所以一开始我沿用了自己的多周期大作业的代码，在此基础上修改成流水线，出现了控制信号出错和功能模块设计出错的情况。其实应该重新写框架比较清楚。
2. 寄存器堆采用先写后读的内部旁路转发方式，但因为对其结构不熟悉，转发的判断漏写了条件，导致后面的仿真出错，找了很久才发现是寄存器堆的问题。
3. 本实验外设的设计比较难理解，一开始我完全无法理解为什么外设要有相应的地址空间来访问并且用总线来传输数据，而不是像以前一样直接在顶层模块直接调用，后来才理解是要在汇编层面访问其地址。而且附加题的串口传输也比较有难度，需要把实验三的串口代码充分理解并合理调用其模块，不能简单地照搬实验三的串口到存储器的实现方法。
4. 本实验要求实现背包算法，用到数逻汇编大作业的汇编代码。但当时我写的汇编代码过于繁琐，且还要再其基础上进行优化，比如读写文件的操作要换成读写串口，如何等待串口的数据完全传输过来，而且读取的物品数量也要根据串口传输过来的第二个数据来确定，我改了很久汇编代码，最终确定用循环来实现。此外，还要合理插入 nop 指令（如本设计要求如果 jr 指令用到的寄存器在前一条指令被写入，需在汇编层面加入 nop）
5. 还有一个比较耗时间的点是指令存储器和数据存储器的数据初始化，比如我是把近百条

翻译成机器码后一一提前写进指令存储器的初始化的。有时候修改了指令就得全盘重写，等验收完之后和其他同学讨论才发现可以用 vivado 的 IP Core 来进行程序自动读写，方便修改代码和测试数据。

6. 我认为本设计还有很多可优化的地方，比如 jr 指令的冒险可以用硬件层面的转发而不是在汇编层面加入 nop，还有因为对流水线的细节不了解导致的整个设计略冗余，时序性能还可以更好。

   此外，我还学会了配置仿真波形的显示窗口。如充分使用 marker 可以加快寻找问题的速度。而且用 vscode 来作为编辑器比用 vivado 自带编辑器的速度更快。