

（深圳）

# 实验报告

开课学期： 2022夏季

课程名称： 计算机设计与实践

实验名称： CPU设计

实验性质： 综合设计型

实验学时： 52 地点： T2210

学生班级： 14班

学生学号： 200111407

学生姓名： 刘玄昊

评阅教师：

报告成绩：

实验与创新实践教育中心制

2022年7月

注：本设计报告中各个部分如果页数不够，请同学们自行扩页。原则上一定要把报告写详细，能说明设计的成果、特色和过程。报告应该详细叙述整体设计，以及设计中的每个模块。设计报告将是评定每个人成绩的重要组成部分（**设计内容及报告写作**都作为评分依据）。

|  |
| --- |
| 设计的功能描述（含所有实现的指令描述，以及单周期/流水线CPU频率） |
| 实现指令  单周期与流水线 CPU 实现指令相同，为 24 条基础指令  指令描述   | 单周期 |  | | --- | --- | | **R 型** |  | | add | (rd) ← (rs1) + (rs2) | | sub | (rd) ← (rs1) - (rs2) | | and | (rd) ← (rs1) & (rs2) | | or | (rd) ← (rs1) | (rs2) | | xor | (rd) ← (rs1) ^ (rs2) | | sll | (rd) ← (rs1) << (rs2) | | srl | (rd) ← (rs1) >> (rs2)，逻辑右移 | | sra | (rd) ← (rs1) >> (rs2)，算术右移 | | **I 型** |  | | addi | (rd) ← (rs1) + sext(imm) | | andi | (rd) ← (rs1) & sext(imm) | | ori | (rd) ← (rs1) | sext(imm) | | xori | (rd) ← (rs1) ^ sext(imm) | | slli | (rd) ← (rs1) << shamt | | srli | (rd) ← (rs1) >> shamt，逻辑右移 | | srai | (rd) ← (rs1) >> shamt，算术右移 | | lw | (rd) ← sext(Mem[(rs1) + sext(offset)][31:0]) | | jalr | t ← (pc) + 4；(pc) ← ((rs1) + sext(offset)) & ~1；(rd) ← t | | **S 型** |  | | sw | Mem[(rs1) + sext(offset)] ← (rs2)[31:0] |  | 多周期 |  | | --- | --- | | **R 型** |  | | add | (rd) ← (rs1) + (rs2) | | sub | (rd) ← (rs1) - (rs2) | | and | (rd) ← (rs1) & (rs2) | | or | (rd) ← (rs1) | (rs2) | | xor | (rd) ← (rs1) ^ (rs2) | | sll | (rd) ← (rs1) << (rs2) | | srl | (rd) ← (rs1) >> (rs2)，逻辑右移 | | sra | (rd) ← (rs1) >> (rs2)，算术右移 | | **I 型** |  | | addi | (rd) ← (rs1) + sext(imm) | | andi | (rd) ← (rs1) & sext(imm) | | ori | (rd) ← (rs1) | sext(imm) | | xori | (rd) ← (rs1) ^ sext(imm) | | slli | (rd) ← (rs1) << shamt | | srli | (rd) ← (rs1) >> shamt, 逻辑右移 | | srai | (rd) ← (rs1) >> shamt, 算术右移 | | lw | (rd) ← sext(Mem[(rs1) + sext(offset)][31:0]) | | jalr | t ← (pc) + 4；(pc) ← ((rs1) + sext(offset)) & ~1；(rd) ← t | | **S 型** |  | | sw | Mem[(rs1) + sext(offset)] ← (rs2)[31:0] | | **B 型** |  | | beq | if ((rs1) = (rs2)) (pc) ← (pc) + sext(offset) | | bne | if ((rs1) ≠ (rs2)) (pc) ← (pc) + sext(offset) | | blt | if ((rs1) < (rs2)) (pc) ← (pc) + sext(offset)，有符号比较 | | bge | if ((rs1) ≥ (rs2)) (pc) ← (pc) + sext(offset)，有符号比较 | | **U 型** |  | | lui | (rd) ← sext(imm[31:12] << 12) | | **J 型** |  | | jal | (rd) ← (pc) + 4；(pc) ← (pc) + sext(offset) | |  |  | |
| 设计的主要特色（除基本要求以外的设计） |
| 1. 分支预测功能：采用静态分支预测，默认不进行跳转。判断跳转在 EX 阶段，故而如果进行跳转，需要插入两条空指令。 2. 实现了结构清晰、可高度复用的总线外设：由于在设计时有意将 CPU、IROM、DRAM 分离，所以后续得到了很清晰的 IROM、CPU、BUS、DRAM、I/O 设备的结构，这样的结构使得代码能能够极为方便的复用和理解，并可适用于单周期和流水线的两个上板测试。 |
| 资源使用、功耗数据截图（Post Implementation；含单周期、流水线2个截图） |
| 老实说，这一个环节的必要性实在是难以恭维。  因为我们学的知识过于浅显，导致经过处理时序约束处理而产生的这些数据具有很低的参考价值。例如 Worst Negative Stack(wns) 表示一个时钟周期内最短的剩余时间，本来可以将其用于计算最大频率（`fMAX = 1/(1/F - wns)`)，但是由于没有处理时序导致其对于计算最大频率没有价值。经过测试，发现上板能够得到的最高频率比通过计算 wns 得到的理论最高频率高出太多（在上板达到最高频率时，wns 通常为较大的负数）。  各种测试都是以 trace 上板的工程来测试，这是因为计算器的汇编中的 load 和 save 指令太少，指令的类型也不全面，而 trace 的汇编就很合适。计算器上板能得到的最高频率也是相当不准的，例如在单周期 CPU 中，计算器上板可以达到 100MHz 的频率，而 trace上板在 50MHz 时就已经失败。  时钟分频设置的有效位数仅限两位，设置的再精确，实际的输出频率也是不准的。例如设置 49.84375MHz，实际输出 50MHz，设置 49.375MHz，实际输出为 49.383MHz。故而没办法也没必要追求测试出精确的最大频率（即使是有两位的有效数字，也不需要将频率设置的精确到两位），所以实验只包含有限的、粗略的几个点。  对于单周期 CPU，实验 25、30、35、40、45、50MHz 的频率；  对于流水线 CPU，实验 50、60、70、80、90、100MHz 的频率。  备注：在提升了频率之后，会出现按下 rst 才能正常跑完 trace 的情况；会出现按下 rst 之后出现不同结果的情况。  何为通过：按下 rst 后得到的结果总是唯一的 2500 0018。反例：2500 0008 和 2500 0018 各 50% 的概率，显然为不通过。  截图：选取 wns 为正数且尽量小的。  （详细见下页）  单周期 CPU   | 频率(MHz) | wns | 是否通过 | 备注 | | --- | --- | --- | --- | | 25 | 4.718ns | 是 |  | | 30 | 0.433ns | 是 |  | | 35 | -0.181ns | 是 |  | | 40 | -3.338ns | 尚未按照要求测试 |  | | 45 | -6.419ns | 尚未按照要求测试 |  | | 50 | -10.981ns | 否 | 第一次显示 0000 0000，按下 rst 后大概率显示 2500 0018，小概率显示 2500 0008 |   很不幸的是，实验室的板子已经被收走了，无法继续上板实验，没能测出最大频率。  截图：（30MHz）  流水线 CPU   | 频率(MHz) | wns | 是否通过 | 备注 | | --- | --- | --- | --- | | 50 | 4.16ns | 是 |  | | 60 | 0.578ns | 尚未按照要求测试 |  | | 70 | -0.132ns | 尚未按照要求测试 |  | | 80 | -2.056ns | 尚未按照要求测试 |  | | 90 | -2.781ns | 尚未按照要求测试 |  | | 100 | -3.944ns | 否 | 第一次显示 0000 0000，按下 rst 后大概率显示 2500 0018，小概率显示 2500 0008 |   截图：（60MHz）  Extra  在这个小小的问题上不该纠结这么长的时间，一方面缺乏继续优化的知识，一方面没有这样的精力。  一个小惊喜是，如果你不加上 clock.xdc 就跑完综合实现的话，功耗这一栏的值会相当的大——很有意思。 |

1 单周期CPU设计与实现

1.1 单周期CPU整体框图

|  |
| --- |
| 要求：无需画出模块内的具体逻辑，但要标出模块的接口信号名、模块之间信号线的信号名和位宽，以及说明每个模块的功能含义。 |
| 模块说明：  1. 取指模块  • PC：更新当前 pc 值  • NPC：得到下一条指令的 pc 值  • IROM：指令存储器，根据 pc 值得到对应的指令  2. 译指/写回模块  • SEXT：立即数扩展单元，根据控制信号对指令中的立即数进行扩展  • RF：寄存器堆，可根据地址读出或写入对应寄存器数据  • MUX4\_1：多路选择器，选择写入 RF 的数据  3. 执行模块  • MUX2\_1：选择器，选择写入 ALU.B 端口的数据  • ALU：运算器，根据控制信号执行加、减、按位与、按位或、按位异或、逻辑右移、逻辑左移、算术右移运算  4. 存储模块  • DRAM：数据存储器，用于存储数据  5. 控制模块  • CONTROLLER：控制器，根据指令生成各个器件的控制信号 |

1.2 单周期CPU模块详细设计

|  |
| --- |
| 要求：画出各个模块的详细设计图，包含内部的子模块，以及关键性逻辑；标出子模块接口信号名、各信号线的信号名和位宽，并有详细的解释说明。 |
| 取指模块  IROM：指令存储器，根据 pc 值得到对应的指令。  PC：在时钟上升沿，若 ~rst\_n 为 0，则寄存器更新为输入 pc 值 din；若 ~rst\_n 为 1，则寄存器初始化为 -4。  NPC：根据 op 控制信号决定 npc 的输出。npc 共有三个来源，分别为 pc + 4、 pc + imm、rd1 + imm。  同时，输出后续计算所需数据：pc + 4。  译指/写回模块  SEXT：立即数扩展单元，根据控制信号对指令中的立即数进行扩展。  （数据的拼接过于复杂故图略）  RF：对 32 位指令 inst 进行解析，得到两个读取地址 rR1、rR2 和一个写入地址 wR，通过组合逻辑读出对应地址信号 rD1(RF[rR1])、rD2(RF[rR2])。  MUX4\_1：多路选择器，选择写入 RF 的数据。其写入数据通过四选一选择器对数据进行选择，其写入数据包括：运算器结果 ALU.C、存储器读出结果 DRAM.rd、pc + 4、拓展后的立即数 SEXT.ext，由 wb\_sel 信号决定写入数据，由 rf\_we 信号决定是否写入。  运算模块  MUX2\_1：选择器，选择写入 ALU.B 端口的数据。  ALU：进行各种运算，并根据 alu\_op 选择所需的运算结果。其中包括加、减、按位与、按位或、按位异或、逻辑右移、逻辑左移、算术右移运算模块。输出计算结果是否为 0，以及正负数的信号。  （SUB 计算的部分简化了，本来应该是 A + (~B) + 1）  存储模块  DRAM：数据存储器，用于存储数据。  控制器模块  CONTROLLER：控制器，根据指令生成各个器件的控制信号。  （具体图略） |

1.3 单周期CPU仿真及结果分析

|  |
| --- |
| 要求：包含逻辑运算、访存、分支跳转三类指令的仿真截图以及波形分析；每类指令的截图和分析中，至少包含1条具体指令；截图需包含信号名和关键信号。 |
| * add，from add.dump   00000004 <reset\_vector>:   4: 00000093 addi x1,x0,0   8: 00000113 addi x2,x0,0   c: 00208733 add x14,x1,x2   10: 00000393 addi x7,x0,0   14: 00200193 addi x3,x0,2   18: 4c771663 bne x14,x7,4e4 <fail>  pc = c，x1 = 0，x2 = 0，x14 = x1 + x2 = 0。   * sub，from sub.dump   0000001c <test\_3>:  1c: 00100093 addi x1,x0,1  20: 00100113 addi x2,x0,1  24: 40208733 sub x14,x1,x2  28: 00000393 addi x7,x0,0  2c: 00300193 addi x3,x0,3  30: 48771a63 bne x14,x7,4c4 <fail>  pc = 24，x1 = 1，x2 = 1，x14 = x1 - x2 = 0。   * and，from and.dump   0000004c <test\_4>:  4c: 00ff00b7 lui x1,0xff0  50: 0ff08093 addi x1,x1,255 # ff00ff <\_end+0xfee0ff>  54: 0f0f1137 lui x2,0xf0f1  58: f0f10113 addi x2,x2,-241 # f0f0f0f <\_end+0xf0eef0f>  5c: 0020f733 and x14,x1,x2  60: 000f03b7 lui x7,0xf0  64: 00f38393 addi x7,x7,15 # f000f <\_end+0xee00f>  68: 00400193 addi x3,x0,4  6c: 44771863 bne x14,x7,4bc <fail>  pc = 5c，x1 = 00ff 00ff，x2 = 0f0f 0f0f，x14 = x1 & x2 = 000f 000f。   * or，from or.dump   00000070 <test\_5>:  70: f00ff0b7 lui x1,0xf00ff  74: 00f08093 addi x1,x1,15 # f00ff00f <\_end+0xf00fd00f>  78: f0f0f137 lui x2,0xf0f0f  7c: 0f010113 addi x2,x2,240 # f0f0f0f0 <\_end+0xf0f0d0f0>  80: 0020e733 or x14,x1,x2  84: f0fff3b7 lui x7,0xf0fff  88: 0ff38393 addi x7,x7,255 # f0fff0ff <\_end+0xf0ffd0ff>  8c: 00500193 addi x3,x0,5  90: 42771c63 bne x14,x7,4c8 <fail>  pc = 80，x1 = f00f f00f，x2 = f0f0 f0f0，x14 = x1 | x2 = f0ff f0ff。   * xor，from xor.dump   00000094 <test\_6>:  94: ff0100b7 lui x1,0xff010  98: f0008093 addi x1,x1,-256 # ff00ff00 <\_end+0xff00df00>  9c: 0f0f1137 lui x2,0xf0f1  a0: f0f10113 addi x2,x2,-241 # f0f0f0f <\_end+0xf0eef0f>  a4: 0020c0b3 xor x1,x1,x2  a8: f00ff3b7 lui x7,0xf00ff  ac: 00f38393 addi x7,x7,15 # f00ff00f <\_end+0xf00fd00f>  b0: 00600193 addi x3,x0,6  b4: 40709863 bne x1,x7,4c4 <fail>  pc = a4，x1 = ff00 ff00，x2 = 0f0f 0f0f，x1 = x1 ⊕ x2 = f00f f00f。   * sll，from sll.dump   0000007c <test\_7>:  7c: fff00093 addi x1,x0,-1  80: 00000113 addi x2,x0,0  84: 00209733 sll x14,x1,x2  88: fff00393 addi x7,x0,-1  8c: 00700193 addi x3,x0,7  90: 4c771263 bne x14,x7,554 <fail>  pc = 84，x1 = ffff ffff，x2 = 0，x14 = x1 << x2 = ffff ffff << 0 = ffff ffff。   * sra，from sra.dump   000000a0 <test\_8>:  a0: 800000b7 lui x1,0x80000  a4: fff08093 addi x1,x1,-1 # 7fffffff <\_end+0x7fffdfff>  a8: 00100113 addi x2,x0,1  ac: 4020d733 sra x14,x1,x2  b0: 400003b7 lui x7,0x40000  b4: fff38393 addi x7,x7,-1 # 3fffffff <\_end+0x3fffdfff>  b8: 00800193 addi x3,x0,8  bc: 4e771263 bne x14,x7,5a0 <fail>  pc = ac，x1 = 7fff ffff，x2 = 1，x14 = x1 >>> x2 = 7fff ffff >>> 1 = 3fff ffff。   * srl，from srl.dump   000000b4 <test\_9>:  b4: fff00093 addi x1,x0,-1  b8: 00700113 addi x2,x0,7  bc: 0020d733 srl x14,x1,x2  c0: 020003b7 lui x7,0x2000  c4: fff38393 addi x7,x7,-1 # 1ffffff <\_end+0x1ffdfff>  c8: 00900193 addi x3,x0,9  cc: 4a771e63 bne x14,x7,588 <fail>  pc = bc，x1 = ffff ffff，x2 = 7，x14 = x1 >> x2 = ffff ffff >> 7 = 01ff ffff。   * 访存指令，from sw.dump   00000144 <test\_10>:  144: 000020b7 lui x1,0x2  148: 02008093 addi x1,x1,32 # 2020 <tdat9>  14c: 12345137 lui x2,0x12345  150: 67810113 addi x2,x2,1656 # 12345678 <\_end+0x12343648>  154: fe008213 addi x4,x1,-32  158: 02222023 sw x2,32(x4) # 20 <reset\_vector+0x1c>  15c: 0000a283 lw x5,0(x1)  160: 123453b7 lui x7,0x12345  164: 67838393 addi x7,x7,1656 # 12345678 <\_end+0x12343648>  168: 00a00193 addi x3,x0,10  16c: 30729e63 bne x5,x7,488 <fail>  x1 = 0000 2020，x2 = 1234 5678，x4 = 0000 2000；  把 x2 的值存到基地址 0000 2000，偏移 32 的地址，即 0000 2020；  读取基地址 0000 2020，偏移 0 的地址，即 0000 2020 到 x5 中。  那么 x5 = 12345678。  x7 = 12345678，x5 == x7，故而不满足 bne，不跳转，可以注意到 pc = 164 后是 pc = 168，故没有发生跳转，对应正确。   * jal，from jal.dump   00000000 <\_start>:  0: 0040006f jal x0,4 <reset\_vector>  00000004 <reset\_vector>:  4: 00200193 addi x3,x0,2  8: 00000093 addi x1,x0,0  c: 0100026f jal x4,1c <target\_2>  00000010 <linkaddr\_2>:  10: 00000013 addi x0,x0,0  14: 00000013 addi x0,x0,0  18: 03c0006f jal x0,54 <fail>  0000001c <target\_2>:  1c: 01000113 addi x2,x0,16  20: 02411a63 bne x2,x4,54 <fail>  第一个 jal 指令没有什么作用，从可以正常到 pc = 4 看出没有问题；  第二个 jal 指令会向 x4 存储 pc + 4，即 10，跳转到 pc = 1c，也没有问题。   * jalr，from jalr.dump   00000004 <reset\_vector>:  4: 00200193 addi x3,x0,2  8: 00000293 addi x5,x0,0  c: 01800313 addi x6,x0,24  10: 000302e7 jalr x5,0(x6)  00000014 <linkaddr\_2>:  14: 0c40006f jal x0,d8 <fail>  00000018 <target\_2>:  18: 01400313 addi x6,x0,20  1c: 0a629e63 bne x5,x6,d8 <fail>  x6 = 18(16 进制)  jalr 指令使 x5 = 14(16 进制)，跳转到 pc = 18(16 进制)，没问题；  x6 = 14(16 进制)  x5 == x6，不满足 bne，不跳转，没问题。   * beq、bne，from beq.dump   00000004 <reset\_vector>:  4: 00200193 addi x3,x0,2  8: 00000093 addi x1,x0,0  c: 00000113 addi x2,x0,0  10: 00208663 beq x1,x2,1c <reset\_vector+0x18>  14: 2a301863 bne x0,x3,2c4 <fail>  18: 00301663 bne x0,x3,24 <test\_3>  1c: fe208ee3 beq x1,x2,18 <reset\_vector+0x14>  20: 2a301263 bne x0,x3,2c4 <fail>  x3 = 2，x1 = 0，x2 = 0；  pc = 10：x1 == x2，满足 beq，跳转到 pc = 1c；  pc = 1c：x1 == x2，满足 beq，跳转到 pc = 18；  pc = 18：x0 != x3，满足 bne，跳转到 pc = 24。   * blt   00000004 <reset\_vector>:  4: 00200193 addi x3,x0,2  8: 00000093 addi x1,x0,0  c: 00100113 addi x2,x0,1  10: 0020c663 blt x1,x2,1c <reset\_vector+0x18>  14: 2a301863 bne x0,x3,2c4 <fail>  18: 00301663 bne x0,x3,24 <test\_3>  1c: fe20cee3 blt x1,x2,18 <reset\_vector+0x14>  20: 2a301263 bne x0,x3,2c4 <fail>  x3 = 2，x1 = 0，x2 = 1；  pc = 10：x1 < x2，满足 blt，跳转到 pc = 1c；  pc = 1c：x1 < x2，满足 blt，跳转到 pc = 18；  pc = 18：x0 != x3，满足 bne，跳转到 pc = 24。   * bge   00000004 <reset\_vector>:  4: 00200193 addi x3,x0,2  8: 00000093 addi x1,x0,0  c: 00000113 addi x2,x0,0  10: 0020d663 bge x1,x2,1c <reset\_vector+0x18>  14: 30301863 bne x0,x3,324 <fail>  18: 00301663 bne x0,x3,24 <test\_3>  1c: fe20dee3 bge x1,x2,18 <reset\_vector+0x14>  20: 30301263 bne x0,x3,324 <fail>  x3 = 2，x1 = 0，x2 = 0；  pc = 10：x1 >= x2，满足 bge，跳转到 pc = 1c；  pc = 1c：x1 >= x2，满足 bge，跳转到 pc = 18；  pc = 18：x0 != x3，满足 bne，跳转到 pc = 24。 |

2 流水线CPU设计与实现

2.1 流水线的划分

|  |
| --- |
| 要求：画出流水线如何划分，说明每个流水级具备什么功能、需要完成哪些操作。 |
| 1. 取指阶段（instruction fetch，简称 IF 阶段）  （1） 更新 pc 值  （2） 得到当前 pc 对应的指令  （3） 可能会由于冒险检测器的控制信号而保持停顿  2. 译指阶段（instruction decode，简称 ID 阶段）  （1）由控制器得到控制信号  （2）得到寄存器堆中两个目标寄存器堆的值  （3）得到对应类型的拓展的立即数  （4）结合译指结果，在冒险检测器中判断当前是否需要数据前递以及停顿，并处理  3. 执行阶段（execute，简称 EX 阶段）  （1）ALU 根据控制信号以及操作数执行对应的运算  （2）根据控制信号和 ALU 计算结果判断是否跳转以及跳转的 pc 值，传达给 IF 阶段的 NPC  （3）完成写回寄存器堆数据的第一次选择（处理）  4. 访存阶段（memory access，简称 MEM 阶段）  （1）根据控制信号以及操作地址对存储器执行相应的访存操作  （2）完成写回寄存器堆数据的第二次选择（处理）  5. 写回阶段（write back，简称 WB 阶段）  根据控制信号向寄存器堆目标寄存器写入相应数据 |

2.2 流水线CPU整体框图

|  |
| --- |
| 要求：无需画出模块内的具体逻辑，但要标出模块的接口信号名、模块之间信号线的信号名和位宽，以及说明每个模块的功能含义。 |
| 模块说明：  1. 取指模块  • PC：更新当前 pc 值  • NPC：得到下一条指令的 pc 值  • IROM：指令存储器，根据 pc 值得到对应的指令  2. 译指模块  • CONTROLLER：控制器，根据指令生成各个器件的控制信号  • SEXT：立即数扩展单元，根据控制信号对指令中的立即数进行扩展  • RF：寄存器堆，可根据地址读出或写入对应寄存器数据  3. 执行模块  • ALU\_MUX：选择器，选择写入 ALU.B 端口的数据  • ALU：运算器，根据控制信号执行加、减、按位与、按位或、按位异或、逻辑右移、逻辑左移、算术右移运算  • NPC\_CONTROL：根据控制信号和计算结果判断是否跳转以及跳转的 pc 值，传达给 NPC  • WD\_MUX1：写回寄存器堆的数据 wD 的第一次选择  4. 存储模块  • DRAM：数据存储器，用于存储数据  • WD\_MUX2：写回寄存器堆的数据 wD 的第二次选择  5. 流水线寄存器模块  • REG\_IF\_ID、REG\_ID\_EX、REG\_EX\_MEM、REG\_MEM\_WB：用于存储各阶段正在执行的指令信息  6. 冒险检测器模块  • HAZARD\_DETECTION：检测并处理数据冒险与控制冒险：接收用于判断冒险的控制信号和数据，发出前递、停顿、清除的控制信号和相关数据 |

2.3 流水线CPU模块详细设计

|  |
| --- |
| 要求：画出各个模块的详细设计图，包含内部的子模块，以及关键性逻辑；标出子模块接口信号名、各信号线的信号名和位宽，并有详细的解释说明；此外，必须结合模块图，详细说明数据冒险、控制冒险的解决方法。 |
| 取指模块  PC：更新当前 pc 值；当控制信号 keep 为 1 时保持寄存器内容在下一个时钟周期不变。  NPC：得到下一条指令的 pc 值；当控制信号 op 为 0 时，npc 输出为当前 pc + 4，op 为 1 时，npc 输出为 EX 阶段来的指定要跳转的 pc 值。故而 op 也是跳转的标志。  IROM 模块与单周期中相应模块原理相同。  译指模块  CONTROLLER：控制器，根据指令生成各个器件的控制信号。和单周期数据通路的控制器不同，由于生成 npc\_op 所需要的 zero 和 sgn 要到 EX 阶段才能得到，故转而生成判断 jal/jalr 和 B 型指令的控制信号 jump 和 branch，传到 EX 阶段，到那时再做处理。  SEXT 和 RF 模块与单周期中相应模块原理相同。  执行模块  ALU\_MUX 和 ALU 模块与单周期中相应模块原理相同。  NPC\_CONTROL：根据控制信号 jump、branch、zero、sgn 得出跳转标志 npc\_op；根据 jump 选择 rd1 + imm 和 pc + imm 作为 npc\_change 输出。npc\_op 和 npc\_change 都将会传达给 IF 阶段的 NPC。  WD\_MUX1：写回寄存器堆的数据 wD 的第一次选择。由于还有 DRAM.rd 没有得到，故目前是三选一。这样的话，imm 和 pc + 4 就不需要继续传递下去了。  存储模块  DRAM 模块与单周期中相应模块原理相同。  WD\_MUX2：写回寄存器堆的数据 wD 的第二次选择。将前面得到的 wD\_temp 和 DRAM.rd 再做一次选择，得到最后的 wD。  流水线寄存器模块  REG\_IF\_ID、REG\_ID\_EX、REG\_EX\_MEM、REG\_MEM\_WB：用于存储各阶段正在执行的指令信息。   | 流水线寄存器 | 保存的信号 | 其他功能 | | --- | --- | --- | | REG\_IF\_ID | pc、pc4、inst | 清空、保持 | | REG\_ID\_EX | wd\_sel、alu\_op、alub\_sel、rf\_we、dram\_we、branch、jump、pc\_imm、imm、pc4、wR、rD1、rD2 | 清空、接收前递控制信号和数据 | | REG\_EX\_MEM | wd\_sel、rf\_we、dram\_we、wR、wD、alu\_c、rD2 | - | | REG\_MEM\_WB | rf\_we、wR | - |   冒险检测器模块  HAZARD\_DETECTION：检测并处理数据冒险与控制冒险：接收用于判断冒险的控制信号和数据，发出前递、停顿、清除的控制信号和相关数据。  检测冒险  数据冒险中 RAW - A/B/C 型冒险的检测  // 数据冒险: RAW  // RAW - A 相邻 wire RAW\_A\_rD1 = (wR\_EX == rR1\_ID) && rf\_we\_EX && rD1\_used && wR\_EX; wire RAW\_A\_rD2 = (wR\_EX == rR2\_ID) && rf\_we\_EX && rD2\_used && wR\_EX;  // RAW - B 间隔一条 wire RAW\_B\_rD1 = (wR\_MEM == rR1\_ID) && rf\_we\_MEM && rD1\_used && wR\_MEM; wire RAW\_B\_rD2 = (wR\_MEM == rR2\_ID) && rf\_we\_MEM && rD2\_used && wR\_MEM;  // RAW - C 间隔两条 wire RAW\_C\_rD1 = (wR\_WB == rR1\_ID) & rf\_we\_WB && rD1\_used && wR\_WB; wire RAW\_C\_rD2 = (wR\_WB == rR2\_ID) & rf\_we\_WB && rD2\_used && wR\_WB;  核心条件是 wR\_EX/MEM/WB == rR1/rR2，附加的条件用于排除其他错误检测情况，如：此阶段 RF 需要可写，即 rf\_we != 0；rD1/rD2 必须被使用；wR 不可以是 x0。  数据冒险中载入-使用型冒险的检测  wire load\_use\_hazard = (RAW\_A\_rD1 || RAW\_A\_rD2) & (wd\_sel\_EX == `DRAM\_RD);  从形式上看，载入-使用型冒险接进于 RAW-A 冒险，两条指令相邻；但区别是上一条指令是 LOAD 指令，这是关键。  控制冒险的检测  wire control\_hazard = npc\_op;  npc\_op 也即跳转的标志。只要发生跳转，就一定有控制冒险的存在。  前递处理 RAW - A/B/C 型冒险  // 前递的使能 assign rD1\_op = RAW\_A\_rD1 || RAW\_B\_rD1 || RAW\_C\_rD1; assign rD2\_op = RAW\_A\_rD2 || RAW\_B\_rD2 || RAW\_C\_rD2;  // if - else 体现了优先级: 相邻 > 间隔 1 条 > 间隔 2 条 always @ (\*) begin  if (RAW\_A\_rD1) rD1\_forward = wD\_EX;  else if (RAW\_B\_rD1) rD1\_forward = wD\_MEM;  else if (RAW\_C\_rD1) rD1\_forward = wD\_WB;  else rD1\_forward = 32'b0; end  always @ (\*) begin  if (RAW\_A\_rD2) rD2\_forward = wD\_EX;  else if (RAW\_B\_rD2) rD2\_forward = wD\_MEM;  else if (RAW\_C\_rD2) rD2\_forward = wD\_WB;  else rD2\_forward = 32'b0; end  用前递解决数据冒险，需要将控制信号（使能）和数据送到 ID/EX 流水线寄存器中。  使能为 1 的话，就将寄存器的输出变更为前递的数据。具体如下：  // 可能接收前递 rD1/ rD2 always @ (posedge clk or negedge rst\_n) begin  if (~rst\_n) rD1\_o <= 32'b0;  else if (rD1\_op) rD1\_o <= rD1\_forward;  else rD1\_o <= rD1\_i; end  always @ (posedge clk or negedge rst\_n) begin  if (~rst\_n) rD2\_o <= 32'b0;  else if (rD2\_op) rD2\_o <= rD2\_forward;  else rD2\_o <= rD2\_i; end  对于这三种冒险来说，只要发生了这三种冒险，那么使能就应该为 1。  // 前递的使能 assign rD1\_op = RAW\_A\_rD1 || RAW\_B\_rD1 || RAW\_C\_rD1; assign rD2\_op = RAW\_A\_rD2 || RAW\_B\_rD2 || RAW\_C\_rD2;  根据检测冒险的结果，填充前递的数据。  如果同时发生多种冒险，if - else 的结构也保证了相邻 > 间隔 1 条 > 间隔 2 条的优先级。  // if - else 体现了优先级: 相邻 > 间隔 1 条 > 间隔 2 条 always @ (\*) begin  if (RAW\_A\_rD1) rD1\_forward = wD\_EX;  else if (RAW\_B\_rD1) rD1\_forward = wD\_MEM;  else if (RAW\_C\_rD1) rD1\_forward = wD\_WB;  else rD1\_forward = 32'b0; end  always @ (\*) begin  if (RAW\_A\_rD2) rD2\_forward = wD\_EX;  else if (RAW\_B\_rD2) rD2\_forward = wD\_MEM;  else if (RAW\_C\_rD2) rD2\_forward = wD\_WB;  else rD2\_forward = 32'b0; end  停顿 + 前递解决载入-使用型数据冒险  处理方式：1) 停顿，插入气泡（PC，IF/ID 不变；ID/EX 置 0）；2) 前递。  停顿的实现是向器件发送 keep 信号，当接收到信号时，将寄存器的输出在下一周期保持不变；清除（插入气泡）的实现是向器件发送 flush 信号，当接收到信号时，将寄存器的输出在下一周期被置 0。  可以被停顿的器件有 PC、REG\_IF\_ID；可以被清除的器件只有 REG\_IF\_ID、REG\_ID\_EX。这是由实际实现决定的。  always @ (\*) begin  if (load\_use\_hazard) keep\_PC = 1'b1;  else keep\_PC = 1'b0; end  always @ (\*) begin  if (load\_use\_hazard) keep\_IF\_ID = 1'b1;  else keep\_IF\_ID = 1'b0; end  always @ (\*) begin  if (load\_use\_hazard || control\_hazard) flush\_ID\_EX = 1'b1;  else flush\_ID\_EX = 1'b0; end  静态分支预测解决控制冒险  静态分支预测，总是预测不跳转；清除后二条指令，即 flush IF/ID，ID/EX。  always @ (\*) begin  if (control\_hazard) flush\_IF\_ID = 1'b1;  else flush\_IF\_ID = 1'b0; end  always @ (\*) begin  if (load\_use\_hazard || control\_hazard) flush\_ID\_EX = 1'b1;  else flush\_ID\_EX = 1'b0; end |

2.4 流水线CPU仿真及结果分析

|  |
| --- |
| 要求：包含控制冒险和数据冒险三种情形的仿真截图，以及波形分析。 |
| （找冒险的办法：查看 start 的波形，找冒险检测器 HAZARD\_DETECTION 的标志信号，就可以找到相应的数据冒险；至于控制冒险，B 型和 jal、jalr 是必定涉及的，也很好找。）  通过 pc\_IF，pc\_ID，pc\_EX，pc\_MEM，pc\_WB 可以看到指令在数据通路中的流动。  RAW - A 型数据冒险（相邻）  000003c8 <test\_1035>:  3c8: 00f00093 addi x1,x0,15  3cc: 00100133 add x2,x0,x1  3d0: 00f00393 addi x7,x0,15  3d4: 40b00193 addi x3,x0,1035  3d8: e8711ce3 bne x2,x7,270 <fail>  3dc: 00140413 addi x8,x8,1  3e0: fffff0b7 lui x1,0xfffff  3e4: 0080a023 sw x8,0(x1) # fffff000 <\_end+0xffffaf90>  3e8: 000f80e7 jalr x1,0(x31)  关注 pc = 3c8 和 pc = 3cc，与 x1 有关。  可以看到 x1 在 pc = 3c8 的指令的 WB 阶段结束后，值变为 0000 000f；  而 pc = 3cc 的指令的 EX 阶段就需要使用到 x1 了，而 x1 的值仍然是原来的 0000 000d，于是就需要使用前递来解决问题。  注意到 pc = 3c8 的指令在 EX 阶段时，pc = 3cc 的指令在 ID 阶段；  此时，冒险检测器检测到发生冒险，并对 rD2 进行前递：在 ID/EX 寄存器内，rD2\_i = 0000 000d，这是原本的 x1 的值；  前递来的 rD2\_forward = 0000 000f，自然需要选取前递的值作为输出（rD2\_op = 1）。  故在 pc = 3cc 的指令到达 EX 阶段时，从 ID/EX 寄存器得到的 rD2\_o 是 0000 000f，成功完成了前递，解决了数据冒险。  RAW - B 型数据冒险（间隔一条）  0000034c <test\_1020>:  34c: 00000213 addi x4,x0,0  350: 00d00093 addi x1,x0,13  354: 00b00113 addi x2,x0,11  358: 00208733 add x14,x1,x2  35c: 00070313 addi x6,x14,0  360: 00120213 addi x4,x4,1 # 1 <\_start+0x1>  364: 00200293 addi x5,x0,2  368: fe5214e3 bne x4,x5,350 <test\_1020+0x4>  36c: 01800393 addi x7,x0,24  370: 3fc00193 addi x3,x0,1020  374: ee731ee3 bne x6,x7,270 <fail>  关注 pc = 350 和 pc = 358，与 x1 有关。  可以看到 x1 在 pc = 350 的指令的 WB 阶段结束后，值变为 0000 000d；  而 pc = 358 的指令的 EX 阶段就需要使用到 x1 了，而 x1 的值仍然是原来的 0000 0018，于是就需要使用前递来解决问题。  注意到 pc = 350 的指令在 MEM 阶段时，pc = 358 的指令在 ID 阶段；  此时，冒险检测器检测到发生冒险，并对 rD1 进行前递：在 ID/EX 寄存器内，rD1\_i = 0000 0018，这是原本的 x1 的值；  前递来的 rD1\_forward = 0000 000d，自然需要选取前递的值作为输出（rD1\_op = 1）。故在 pc = 358 的指令到达 EX 阶段时，从 ID/EX 寄存器得到的 rD1\_o 是 0000 000d，成功完成了前递，解决了数据冒险。    RAW - C 型数据冒险（间隔两条）  00000334 <test\_1017>:  334: 00d00093 addi x1,x0,13  338: 00b00113 addi x2,x0,11  33c: 002080b3 add x1,x1,x2  340: 01800393 addi x7,x0,24  344: 3f900193 addi x3,x0,1017  348: f27094e3 bne x1,x7,270 <fail>  关注 pc = 33c 和 pc = 348，与 x1 有关。  可以看到 x1 在 pc = 33c 的指令的 WB 阶段结束后，值变为 0000 0018；  而 pc = 348 的指令的 EX 阶段就需要使用到 x1 了，而 x1 的值仍然是原来的 0000 000d，于是就需要使用前递来解决问题。  注意到 pc = 33c 的指令在 WB 阶段时，pc = 348 的指令在 ID 阶段；  此时，冒险检测器检测到发生冒险，并对 rD1 进行前递：在 ID/EX 寄存器内，rD1\_i = 0000 000d，这是原本的 x1 的值；  前递来的 rD1\_forward = 0000 0018，自然需要选取前递的值作为输出（rD1\_op = 1）。  故在 pc = 348 的指令到达 EX 阶段时，从 ID/EX 寄存器得到的 rD1\_o 是 0000 0018，成功完成了前递，解决了数据冒险。  载入 - 使用型数据冒险  00000b78 <test\_12012>:  b78: 000031b7 lui x3,0x3  b7c: eec18193 addi x3,x3,-276 # 2eec <test\_37035+0x5dc>  b80: 00000213 addi x4,x0,0  b84: 000040b7 lui x1,0x4  b88: 00408093 addi x1,x1,4 # 4004 <tdat\_lw2>  b8c: 0040a703 lw x14,4(x1)  b90: 00070313 addi x6,x14,0  b94: 0ff013b7 lui x7,0xff01  b98: ff038393 addi x7,x7,-16 # ff00ff0 <\_end+0xfefcf80>  b9c: ec731a63 bne x6,x7,270 <fail>  ba0: 00120213 addi x4,x4,1 # 1 <\_start+0x1>  ba4: 00200293 addi x5,x0,2  ba8: fc521ee3 bne x4,x5,b84 <test\_12012+0xc>  关注 pc = b8c 和 pc = b90，与 x14 有关。  复习处理方式：1) 停顿，插入气泡（PC，IF/ID 不变；ID/EX 置 0）；2) 前递。  首先可以看到 x14 在 pc = b8c 的指令的 WB 阶段结束后，值变为 0ff0 0ff0，同时这个值要等到 MEM 阶段才会从 dram 里读出；  注意到 pc = b8c 的指令在 EX 阶段时，pc = b90 的指令在 ID 阶段；  此时，冒险检测器检测到发生载入 - 使用型数据冒险，故停顿，插入气泡：  PC，IF/ID 不变：可以看到 pc\_IF = 0000 0B94 保持了两个周期，pc\_ID = 0000 0b90 也保持了两个周期；  ID/EX 置 0：pc\_EX 被归 0，这样 pc = b90 的指令就会被停下一个周期。  注意到 pc = b8c 的指令在 MEM 阶段时，pc = b90 的指令仍在 ID 阶段，此时已经从 dram 中读出 0ff0 0ff0 了。  对 rD1 进行前递：在 ID/EX 寄存器内，rD1\_i = 00ff 00ff，这是原本的 x14 的值；  前递来的 rD1\_forward = 0ff0 0ff0，自然需要选取前递的值作为输出（rD1\_op = 1）。  故在 pc = b90 的指令到达 EX 阶段时，从 ID/EX 寄存器得到的 rD1\_o 是 0ff0 0ff0，成功完成了前递，解决了载入 - 使用型数据冒险。  控制冒险  00000000 <\_start>:  0: 0040006f jal x0,4 <reset\_vector>  00000004 <reset\_vector>:  4: 02500413 addi x8,x0,37  8: 01841413 slli x8,x8,0x18  c: fffff0b7 lui x1,0xfffff  10: 0080a023 sw x8,0(x1) # fffff000 <\_end+0xffffaf90>  14: 27c00fef jal x31,290 <n1\_add\_test>  18: 00000013 addi x0,x0,0  关注 pc = 0 的 jal 指令：  复习处理方式：静态分支预测，总是预测不跳转；清除后二条指令，即 flush IF/ID，ID/EX。  pc = 0 的 jal 指令，当移动到 EX 阶段，由于是 jal 指令，故控制信号 npc\_op 被赋值为 1，作为跳转的标志；  此时，冒险检测器检测到发生控制冒险，清除后二条指令，即 flush IF/ID，ID/EX，于是在下一个周期，pc\_EX，pc\_ID 就被清为 0 了。  并且，NPC 的 npc 也被修正为 4，于是在下一个周期，PC 的 pc 值就被修正为 4 了。  （PS：在想要是 imm = pc + 4 的话是不是就别跳转了，就如同数据冒险中 x0 不需要被前递一样）  pc = 14 的 jal 指令也是同理。 |

3 设计过程中遇到的问题及解决方法

|  |
| --- |
| 要求：包括设计过程中遇到的有价值的错误，或测试过程中遇到的有价值的问题。所谓有价值，指的是解决该错误或问题后，能够学到新的知识和技巧，或加深对已有知识的理解和运用。 |
| 3.1 ALU 模块设计的细微差别  3.1.1如何发现  （左为修改前，右为修改后）  在实现单周期上板之后，为了测试最高频率，需要将代码优化，故而将大多数的 reg 类型的变量修正为了 wire 类型，同时将大量的 always 块修改为 assign 和三目运算符。  本来预期可以达到更高的频率，但是令人遗憾的是，经过这样的修改后，连单周期的最低要求 25MHz 也达不到，上板 trace 的结果只显示到 2500 0010，并没有到预期的 2500 0018。  针对修改前后的差别经行漫长的上板排查后，终于定位到 ALU 上，只需要把 ALU 修改为原来的样式就可以跑到 2500 0018；此外，将 assign 和三目的组合去运行在线 trace 也无法通过所有测试，没有通过的指令是 sra 和 srai。  运行 srai 的具体测试，通过查看反汇编和波形，发现运算出错的具体情况，本来应该进行的算术右移变成了逻辑右移：A = 8000 0000，B = 01，即 8000 0000 算术右移 1 位，结果本应该是c000 0000，但是结果却是 4000 0000。  即使是查看波形，也只能得出计算出错的结论，并不能带来更多的成果（毕竟算错了就是算错了）。  即使是调整 `SRA 在三目中的位置也无济于事。  已经知道了导致错误的表层原因，却不知道深层的原因。  3.1.2理论解释  在老师的指点下，意识到三目运算是一种变种的 if-else 语句，以及理解了 case 语句和 if-else 语句的差别。  | | case | if-else |  | 逻辑上 | 各分支间无优先级 | 有优先级 |  | 运行时间上 | 并行判断 | 串行判断 |  那么在判断到 op == `SRA 时，在时序上就可能出现问题，导致错误；  或者说，在实现的时候，if-else 的电路实现是很长的一条通路，这样是不好的。  另外，简单的把 reg 改为 wire 并不会起到实际的作用，因为不和时序逻辑相关的 reg，在综合时还是会被综合为连线。  所以之前的优化都是无效的。 |

4 总结

|  |
| --- |
| 要求：谈谈学完本课程后的个人收获以及对本课程的建议和意见。请在认真总结和思考后填写总结。 |
| 通过学习这门课程，让我对计算机组成原理课内所学的知识有了进一步的理解和运用，这是仅学习理论知识所无法得到的宝贵经验。  大体知识都已经在计算机组成原理中学习，真的到实践的时候细节会更多、更复杂，但也可以反过来去深刻的理解某些理论，使得学过的理论有了实践支撑。我想这就是“实践是理论的基础，即实践对理论具有决定作用；理论对实践有反作用，科学的理论对实践具有积极的指导作用”的道理吧。  让我感触比较深的有总线外设的部分。模仿实验一的 Soc，设计了类似的总线外设。  其实在我学习总线这一章节的时候，并没有对总线有很清除的理解，虽然知道基本概念、分类、结构的基础知识，但是对总线的具体实现完全不理解，比如不明白为什么地址总线是指向 I/O 设备的，以这个问题为代表的诸多问题表现了我对总线的不理解，也让我在设计结构的时候一筹莫展，直到去看了实验一的 Soc，我才明白应该如何设计，并且在实现完成之后，我对基本的总线外设有了新的理解：  以 cpu 的视角来看，cpu 的 dram\_we 是驱使外部器件读写的动力，addr 是区别外部设备的方法，read\_data 和 write\_data 是读写的数据。  之前没有总线外设的时候，cpu 是可以直接驱使 dram 读写的；在有了外设之后，为了区分 dram 和众多外设，才需要 BUS 来进行处理，此时的 dram\_we 就不是原来的 dram 的读写使能了，而是和外部器件读写的请求，这个外部器件可以是 dram，也可以是外设；这个读写请求会和传来的地址一起，用于生成外部器件的读写使能（笼统说法），比如传来的是外设的地址，那么外设的使能（注意是使能不是读写使能）为 1，而 dram 的使能为 0，然后再用 dram\_we 来觉决定读写使能。  至于怎么区分dram 和外设，外设之间怎么区分，就是地址的作用。我去规定哪些地址是 dram 用的 ， 哪些地址又是哪些外设用的。  那么对于外设而言，addr 数据线毫无疑问的是指向外设的，因为 addr 的源头是 cpu；但是外设内部是可以对连接来的 addr 进行判断的，结合使能 io\_en，就能判断是否是选中了自己。  当然现在设计的这个外设在写上略微的简单了点，还是很基础啦，距离功能较为完善的总线外设还有很远。  最后，虽然计组学的不是很好，但是我觉得这不妨碍我认真学这门实践课。全身心的投入到这门课里是一种很好的体验（对比：之前软件构造实践时间非常紧，设计也不完善，只做个半成品其实是很让人气馁的）；而且能够利用所学知识造一个简单的 CPU，难道这不酷吗？！  总之，学习这门课我收获了很多，也很感谢老师和同学的教导！ |