

数字逻辑与处理器基础实验报告

学号：2017011090

班级：无 78

姓名：游子權

一、实验名称：流水线 MIPS 处理器

二、实验目的：于 FPGA 板上完成一基于 MIPS 指令集的五级流水线处理器，使其完成 100 个数据的排序功能。

三、设计方案、原理说明与关键代码：

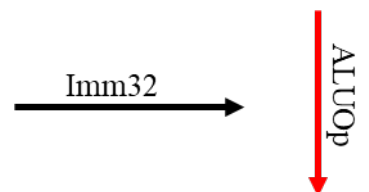
（一）设计原则

1. 分成五个阶段，分别为取指令（Instruction Fetch, IF）、解读指令（Instruction Decode, ID）、执行运算（Execution, EX）、存取资料记忆体（Memory Access, MEM）、写回寄存器堆（Write Back to register file, WB）。
2. 使用 ROM 作为 Instruction Memory，采用于时钟上升沿读写的 RAM 作为 Data Memory。采用于时钟上升沿写入的 Register File。
3. 能采用转发（forwarding）时，采用完全的转发处理数据关联问题。
4. 不能转发时，采用尽量少的阻塞（stall）来解决竞争。
5. 分支（branch）、跳转（jump）类指令皆在 ID 阶段提前判断，在分支确认或跳转时取消 IF 阶段指令。
6. CPU 设有七段数码管外设显示排序结果、设有定时器外设。
7. CPU 可支援未知指令的异常与定时器中断。





（二）系统框图：除五级流水线外，本人设置两个独立于流水线外的单元（后简称「独立单元」）来处理流水线的数据与控制冒险：分别为 Hazard and Jump Unit、Forward and Stall Unit。

【框图阅读说明】

1. 线：线的粗细反映信号位宽，线的颜色本身没有特殊含义，但相同颜色的线上载有相同的讯号。故当图上不可避免需产生交叉时，本人用不同颜色的线区分不同信号。为免与相邻线产生混淆，在线所载讯号的名称一律记于水平线上方或铅直线右侧，如图 3-1 所示。



▲图 3-1：图例

2. Input/ Output Port：形如    
 - 其中一端接入自/接出至流水线的其他阶段
 - 其中一端与 Forward & Stall Unit 相连
 - 其中一端与 Hazard & Jump Unit 相连
 - 其中一端与 CPU 外设相连

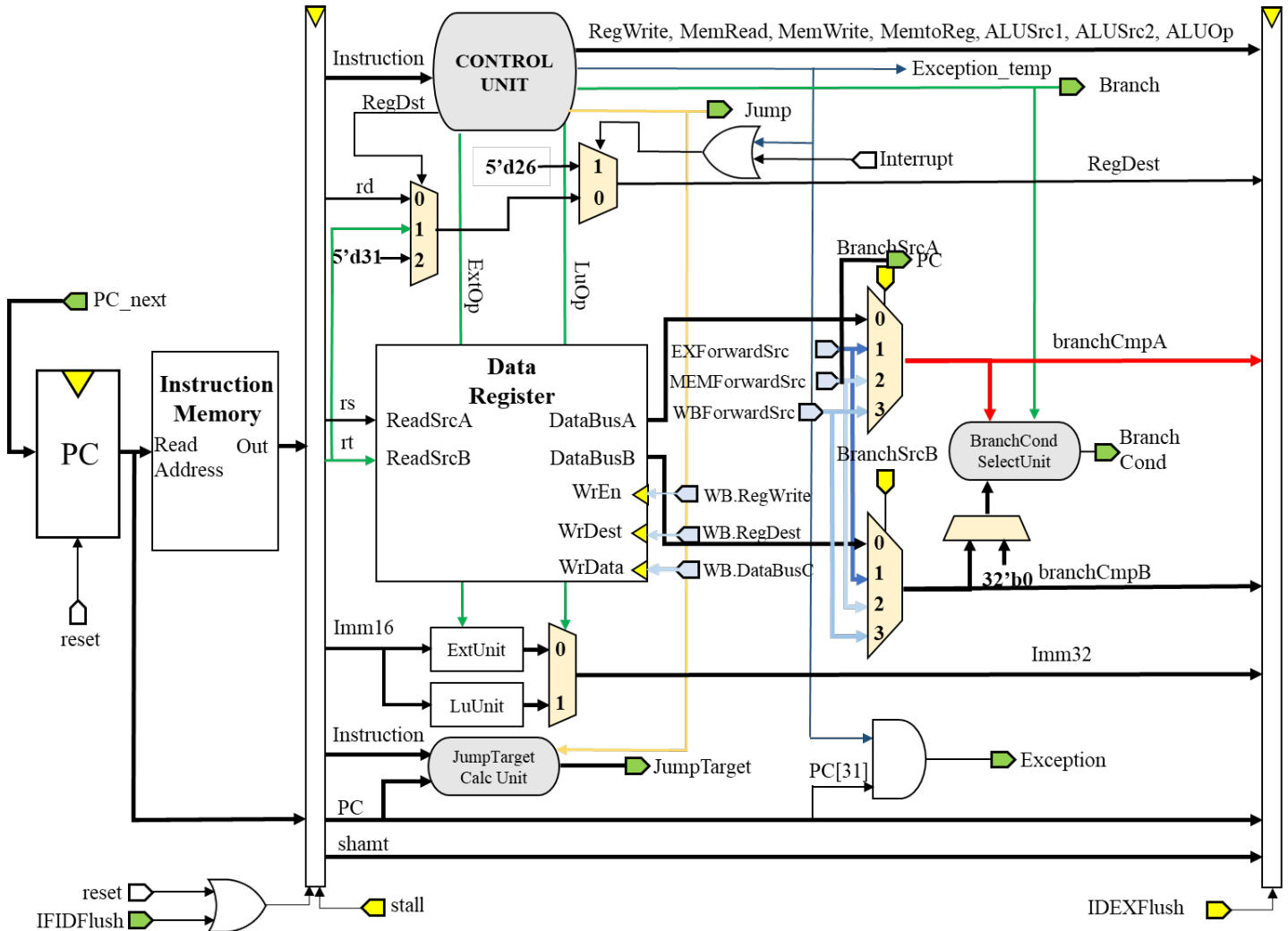
▲图 3-2：图例

当引入/出流水线中其他阶段讯号时，或引入/出信号供 Hazard and Jump Unit、Forward and Stall Unit 使用时绘之。同一阶段流水线的讯号引用不作此记号。不同底色的 Port 代表不同意义，如图 3-2 所示。

3. 由于需使用到系统时钟信号(CLK)之装置甚多，本图中不一一标明，凡在时钟上升沿才写入新值的时序逻辑电路端口，均加入▶标志以资识别。

【五级流水线框图】

1. IF、ID 阶段



▲图 3-3：流水线 IF、ID 阶段框图

说明：

- 程序计数器 (Program Counter, PC) 的下一状态值 PC_next 由 Hazard and Jump Unit 综合其他阶段之指令计算给出。
- Instruction Memory 为 ROM，存储已录入好之指令集。
- IF/ID Register 在 reset = 1 或 IFIDFlush = 1 时，将下一状态的 Instruction 清零，但 PC 保持不变（以利 stall 时又遇 Exception 或 Interruption，可以记录正常的 PC）；stall=1 时下一状态维持原状态不变。IFIDFlush 由 Hazard and Jump Unit 计算给出。Stall 由 Forward and Stall Unit 计算给出。

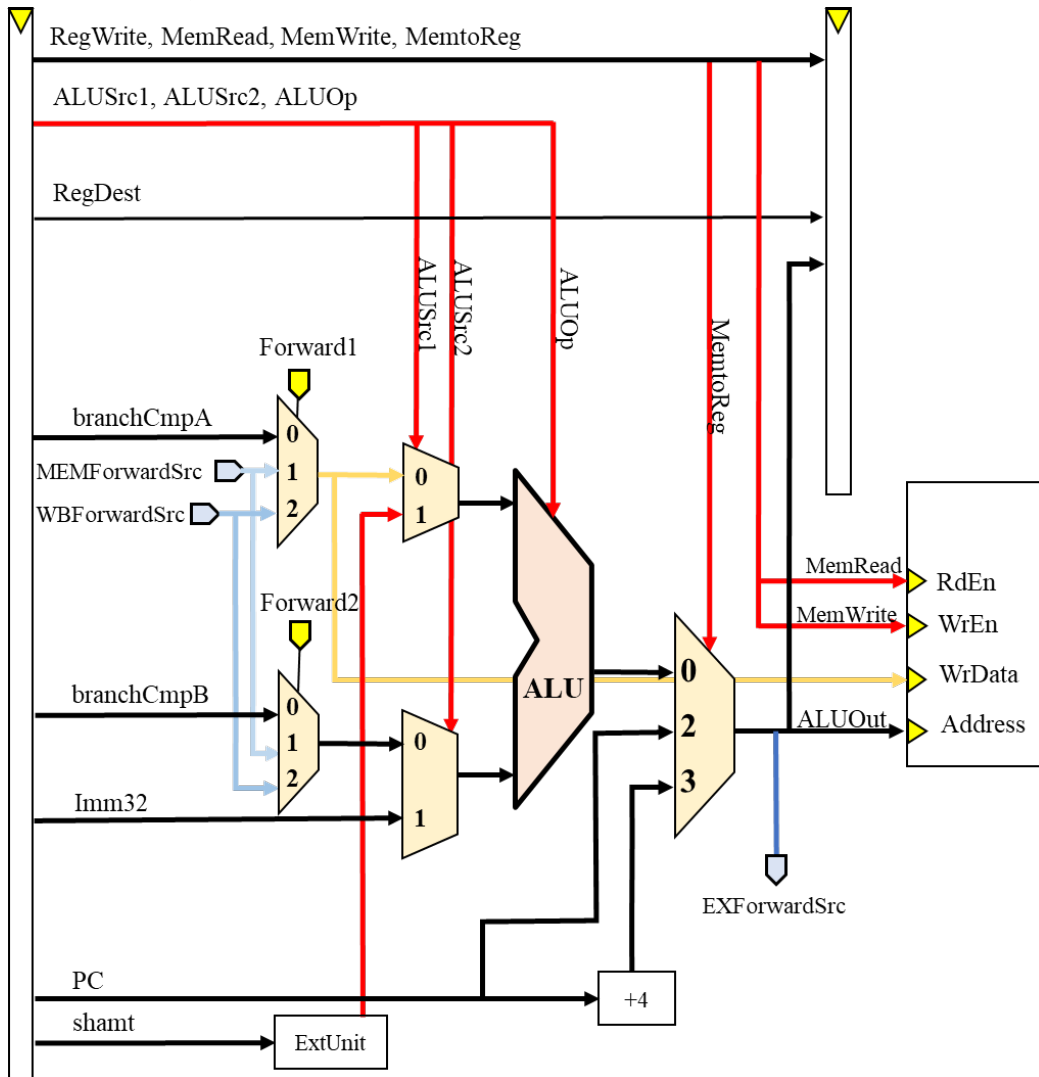
- ID 阶段完成：指令译码、寄存器数据提取、立即数处理、转发、分支判断与跳转地址计算；遇例外与异常时，立刻改变原指令为 jal Exception/Interrupt, \$26：即将当前 PC 存入 26 号寄存器，并将跳转地址设至异常/中断处理程序入口地址。
- CONTROL UNIT 为指令之控制信号译码单元，本人沿用单周期 MIPS 处理器中之译码单元，仅加入 ori 指令、branch 及 branch and link 指令、异常与中断的相关判断。
- JumpTarget CalcUnit 为计算跳转地址的单元，当 Jump 信号为非 0 时，采用伪直接寻址方法得到跳转地址；反之则采用 PC 相对寻址得出。跳转地址的 PC[31]保持与本条指令之 PC[31]相同。关键代码如下：(ID.v)

```
output wire [31:0] JumpTarget;
wire [31:0] JumpTarget_temp;
assign JumpTarget_temp = (Jump != 2'b00) ?
{PC[31:28], Instruction[25:0], 2'b00} :
(PC + 32'd4 + ({14{Instruction[15]}}, Instruction[15:0], 2'b00));
assign JumpTarget = {PC[31], JumpTarget_temp[30:0]};
```

- BranchCond SelectUnit 为根据 Branch 信号决定要选通哪种 Branch 条件（bne, beq, blez...等）的判断结果。关键代码如下：(ID.v)

```
86 //Conditioning on all sorts of Branch
87 wire BranchCond_beq;
88 wire BranchCond_bne;
89 wire BranchCond_bgtz;
90 wire BranchCond_blez;
91 wire BranchCond_bgez;
92 wire BranchCond_bltz;
93
94 assign BranchCond_beq = (branchCmpA == branchCmpB) ? 1'b1 : 1'b0;
95 assign BranchCond_bgez = (branchCmpA[31] == 0) ? 1'b1 : 1'b0;
96 assign BranchCond_bltz = ~BranchCond_bgez;
97 assign BranchCond_bgtz = BranchCond_bgez && (branchCmpA != 32'h0);
98 assign BranchCond_blez = ~BranchCond_bgtz;
99 assign BranchCond_bne = ~BranchCond_beq;
100
101 always@(*)
102 begin
103     case (BranchType)
104         3'b001: BranchCond <= BranchCond_beq;
105         3'b010: BranchCond <= BranchCond_bne;
106         3'b011: BranchCond <= BranchCond_blez;
107         3'b100: BranchCond <= BranchCond_bgtz;
108         3'b101: BranchCond <= BranchCond_bltz;
109         3'b110: BranchCond <= BranchCond_bgez;
110         default: BranchCond <= 0;
111     endcase
112 end
```

2. EX 阶段



▲图 3-4: 流水线 EX 阶段框图

说明：

- EX 阶段完成：转发、ALU 计算。
- 原本 MemtoReg 的值只有 0,1 两种，当 MemtoReg == 1 时，把 Memory 读出的内容写回寄存器中，但为了因应例外与中断的情形下，须把目前的 PC（中断）或下一条指令的 PC（异常）存回寄存器中，为最小程度的改动流水线结构，在 ID 阶段中将 MemtoReg 信号扩为 2 位。原 ALU 输出端新增一多路选择器，当 MemtoReg 为 2 或 3 时分别表示该条指令解码时发生中断与异常，从而将相关的地址直接「看作」是 ALU 计算的输出，并交由下一阶段处理。
- 处理完因应异常与中断欲存储的内容后，MemtoReg 可回复为 1 位。（指示欲选通 Memory 内容或 ALUOut）这个操作在 EX/MEM Register 中完成。当 EX 阶段 MemtoReg != 1 时，EX/MEM 寄存器会在下个时钟上升沿向 MEM 阶段输出 MemtoReg = 0。


```

always@(*)
begin
    case(EX_ALUOut)
        32'h40000000:WB_MemReadOut_temp<=TH;
        32'h40000004:WB_MemReadOut_temp<=TL;
        32'h40000008:WB_MemReadOut_temp<={29'b0, TCON};
        32'h4000000C:WB_MemReadOut_temp<={24'b0, leds};
        32'h40000010:WB_MemReadOut_temp<={20'b0, digit_en, digit};
        32'h40000014:WB_MemReadOut_temp<=Systick;
        default:WB_MemReadOut_temp<=RAMMemReadOut;
    endcase
end
assign WB_MemReadOut = (EX_ALUOut[31:28] == 4'h3) ? ROMMemReadOut :WB_MemReadOut_temp;

SortData SortData_inst(.Address(EX_ALUOut[9:2]),.MemOut(ROMMemReadOut));

```

- 查定时器外设虽不属 MEM 阶段之单元，但考量其相关参数 TL, TCON 需由外设本身（硬件）与软件协同控制，故仍将其代码置于 MEM.v 中，与其他外设的写入共同撰写于同一过程块中。关键代码如下：(MEM.v)

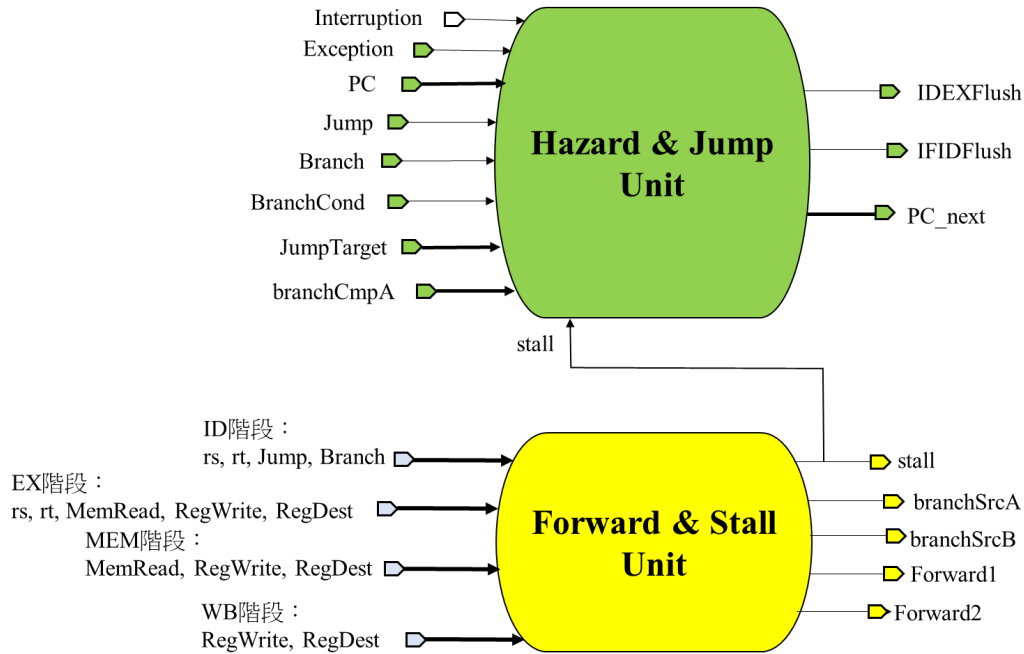
```

51 always @(posedge clk)
52 begin
53     if(reset)
54     begin
55         TH <= 32'h0;
56         TL <= 32'hffffffff;
57         TCON <= 3'b000;
58         leds <= 8'b00000000;
59         digit <= 8'b11111111;
60         digit_en <= 4'b0000;
61     end
62     else if(EX_MemWrite)
63     begin
64         case (EX_ALUOut)
65             32'h40000000: TH <= EX_WrData;
66             32'h40000008: TCON <= EX_WrData[2:0];
67             32'h4000000C: leds <= EX_WrData[7:0];
68             32'h40000010: begin
69                 digit <= EX_WrData[7:0];
70                 digit_en <= EX_WrData[11:8];
71             end
72         endcase
73     end
74     else if(TCON[0] && TCON[1] && TL==32'hffffffff)
75     begin
76         TCON[2] <= 1'b1;
77     end
78
79     if(TCON[0])
80     begin
81         if(TL == 32'hffffffff) TL <= TH;
82         else TL <= TL + 1;
83     end
84     else if (TCON == 3'b000)
85     begin
86         TL <= TH;
87     end
88 end

```

- WB 阶段完成：回存寄存器数据的选通。(指示欲选通 MemReadOut 或 ALUOut)

【独立单元框图】



▲图 3-6: 独立单元框图

1. Hazard and Jump Unit (hazardjump.v): 本单元分析各指令译码之结果，计算下个周期的 PC。

- 正常情形下： $PC_next = PC + 4$ 。
- 若 Jump 不为 0，表示可能为 j, jal, jr, jalr，若为 j, jal 选通 JumpTarget 为 PC_next；若为 jr, jalr 选通 branchCmpA 为 PC_next。
- 若 Branch 不为 0，表示可能为 b, beq, bne, blez, bgez, bltz, bgtz, blezal, bgtzal，当分支条件成立 ($branchCond == 1$)，选通 JumpTarget 作为 PC_next，否则按正常情形处理。
- 发生中断、异常时，PC_next 为中断和异常处理程序的入口地址。
- 当发生阻塞 ($stall == 1$) 时，需要将 IF/ID 寄存器保持原值 ($IFIDFlush = 0$)，将 ID/EX 寄存器清零 ($IDEXFlush = 1$)。
- 当发生任何非正常的跳转时 (即 $PC_next \neq PC + 4$)，需要将 IF/ID 寄存器清零 ($IFIDFlush = 1$)，ID/EX 寄存器则保持原值 ($IDEXFlush = 0$)。

```

assign PC_next =
  (Interrupt) ? 32'h80000004 :
  (Exception) ? 32'h80000008 :
  (stall) ? (PC) :
  ((Jump==2'b10) ? (branchCmpA) :
  ((Jump== 2'b01) ? (JumpTarget) :
  ((Branch && BranchCond) ? JumpTarget
  : {(PC[31], PC_plus_4[30:0])}));
assign IFIDFlush = (stall) ? 0 :
  ((Jump || (Branch && BranchCond) || Interrupt || Exception) ? 1 : 0);
assign IDEXFlush = (stall) ? 1 : 0;

```

2. Forward and Stall Unit (Forward.v): 本单元分析后指令的处理是否依赖于前一未处理完之指令，进而转发最新计算结果或命 CPU 阻塞。判断转发、阻塞与否的条件较繁复，无法于此一一列举，惟皆遵从以下原则：

- 前一指令需对寄存器或 Memory 的值造成改变，方有转发或阻塞之必要。
- 判断前指令之目标寄存器是否与后指令之源寄存器相同，但忽略 0 号寄存器。
- 转发之优先序当以愈接近当前阶段所计算出来之数为优先。以 branchSrcA 为例，判断转发的关键代码如下：

```
assign BranchSrcA = (EX_RegWrite) ?
((EX_RegDest != 0) && (EX_RegDest == IDF_rs)) ? 2'b01 :
(MEM_RegWrite) ? ((MEM_RegDest != 0) && (MEM_RegDest == IDF_rs)) ? 2'b10 :
(WB_RegWrite) ? ((WB_RegDest != 0) && (WB_RegDest == IDF_rs)) ? 2'b11 : 2'b00 : 2'b00 :
(WB_RegWrite) ? ((WB_RegDest != 0) && (WB_RegDest == IDF_rs)) ? 2'b11 : 2'b00 : 2'b00 :
(MEM_RegWrite) ? ((MEM_RegDest != 0) && (MEM_RegDest == IDF_rs)) ? 2'b10 :
(WB_RegWrite) ? ((WB_RegDest != 0) && (WB_RegDest == IDF_rs)) ? 2'b11 : 2'b00 : 2'b00 :
(WB_RegWrite) ? ((WB_RegDest != 0) && (WB_RegDest == IDF_rs)) ? 2'b11 : 2'b00 : 2'b00);
```

- 遇 load-use 需阻塞 1 个周期；遇 load-jr, jalr，因跳转于 ID 阶段提前判断，需阻塞 2 个周期。关键代码如下：

```
assign stall1 = (EX_MemRead) ?
( ((EX_RegDest == IDF_rs) && (EX_RegDest != 0)) || (EX_RegDest == IDF_rt) && (EX_RegDest != 0) ) ? 1 : 0 ;
assign stall2 = (ID_Jump == 2'b10 || Branch) ?
((EX_MemRead) ? ((EX_RegDest != 0) && (EX_RegDest == IDF_rs)) ? 1 :
(MEM_MemRead) ? ((MEM_RegDest != 0) && (MEM_RegDest == IDF_rs)) ? 1 : 0 : 0 ) :
: ((MEM_MemRead) ? ((MEM_RegDest != 0) && (MEM_RegDest == IDF_rs)) ? 1 : 0 : 0 ) : 0 ;
assign stall = (stall1 | stall2);
```

(三) 排序与中断汇编代码 (msort_forhardCPU_withInterrupt.asm)

1. 排序部分：本次本人采用春季学期所写的 Merge Sort 算法作测试。该算法已于 MARS 上成功运行。本次仅需将 MARS 中用于分配空间的 syscall 函数全数改为手动分配空间的指令即可。并无太大问题。
2. 中断部分：为显示排序结果，在排序的主程序完成后，本人采用定时器中断，约每 100000 个时钟周期中断 1 次（100MHz 时钟周期下约为 1ms），使七段数码管做扫描显示，每作 1500 次扫描显示（100MHz 时钟周期下约为 1.5s），切换下一个欲显示的数。

● 关键代码与说明

1. 第 126 行 Loop：主程序最后一个指令（无限回圈）。
2. 第 127 行 Interrupt：中断服务程序开始，其中 \$s3 为七段数码管目前状态。\$s2 是切换下一个数与否的计数器。\$s0 为定时器外设的 TCON 变量。
3. 第 130~136 行：提取目前数码管的使能状态，并根据目前状态跳转到相应状态转移的处理程序内。


```

126 Loop: beq $zero, $zero, Loop
127 Interrupt: andi $s0, $s0, 0x0000
128 sw $s0, 8($s1)
129 bgt $s2, 2000, changeResult #CHANGE WHEN IMPLEMENT WITH HARDWARE
130 ProcStart: andi $t1, $s3, 0x00000f00
131 ori $s3, $s3, 0x0f00
132 beq $t1, 0x0e00, Digit1110
133 beq $t1, 0x0d00, Digit1101
134 beq $t1, 0x0b00, Digit1011
135 beq $t1, 0x0700, Digit0111
136 j Digit0111

```

4. 第 131~148 行：作状态转移的处理，变更使能状态，并提取下一个要扫描显示的数字。

```

137 Digit1110: andi $s3, $s3, 0x0dff
138 srl $a0, $gp, 4
139 j DigitProcEnd
140 Digit1101: andi $s3, $s3, 0x0bff
141 srl $a0, $gp, 8
142 j DigitProcEnd
143 Digit1011: andi $s3, $s3, 0x07ff
144 srl $a0, $gp, 12
145 j DigitProcEnd
146 Digit0111: andi $s3, $s3, 0x0eff
147 srl $a0, $gp, 0
148 j DigitProcEnd

```

5. 第 149~167 行：依据欲让数码管显示的数字，跳转到相应译码的处理程序内。

```

149 DigitProcEnd: ori $s3, $s3, 0x00ff
150 andi $a0, $a0, 0x000f
151 beq $a0, 0x0000, Number0
152 beq $a0, 0x0001, Number1
153 beq $a0, 0x0002, Number2
154 beq $a0, 0x0003, Number3
155 beq $a0, 0x0004, Number4
156 beq $a0, 0x0005, Number5
157 beq $a0, 0x0006, Number6

```

```

158 beq $a0, 0x0007, Number7
159 beq $a0, 0x0008, Number8
160 beq $a0, 0x0009, Number9
161 beq $a0, 0x000a, NumberA
162 beq $a0, 0x000b, NumberB
163 beq $a0, 0x000c, NumberC
164 beq $a0, 0x000d, NumberD
165 beq $a0, 0x000e, NumberE
166 beq $a0, 0x000f, NumberF
167 j NumberProcEnd

```

6. 第 168~200 行：根据 Basys3 的管脚分配与数码管共阳极特性，完成相应数字译码。
7. 第 200~204 行：状态变更完成，将新状态 \$s3 存至数码管外设对应的地址 ($\$s0 + 16$) 中，将定时器状态 \$s0 重设，跳转回中断前的主程序地址 \$k0。
8. 第 205~208 行：同一个数做一定次数的扫描显示后（即显示一定时间后，具体周期由 \$s2 决定），归零 \$s2，并切换下一个欲显示的数。

```

200 NumberProcEnd: sw $s3, 16($s1)
201 addi $s2, $s2, 1
202 ori $s0, $s0, 0x0003
203 sw $s0, 8($s1)
204 jr $k0
205 changeResult: lw $gp, 0($t8)
206 add $s2, $zero, $zero
207 lw $t8, 4($t8)
208 j ProcStart

```

四、电路模拟情形：

(一) 比较CPU上模拟后的排序结果与MARS上执行同样程序后的排序结果。

> [25][31:0]	00000230
> [24][31:0]	0000019a
> [23][31:0]	000000c0
> [22][31:0]	00000257
> [21][31:0]	000000a8
> [20][31:0]	00000167
> [19][31:0]	00000198
> [18][31:0]	00000230
> [17][31:0]	00000128
> [16][31:0]	000003c6
> [15][31:0]	00000150
> [14][31:0]	0000001d
> [13][31:0]	000002e0
> [12][31:0]	00000333
> [11][31:0]	000002c8
> [10][31:0]	000001ad
> [9][31:0]	000001a8
> [8][31:0]	000001ea
> [7][31:0]	000002b0
> [6][31:0]	000000f4
> [5][31:0]	00000170
> [4][31:0]	0000013d
> [3][31:0]	00000060
> [2][31:0]	00000187
> [1][31:0]	000001a0
> [0][31:0]	00000000

(a) CPU 模拟执行程序结果

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00003000	0x00000000	0x000031a0	0x00000187	0x00003060	0x0000013d	0x00003170	0x000000f4	0x000032b0
0x00003020	0x000001ea	0x000031a8	0x000001ad	0x000032c8	0x00000333	0x000032e0	0x0000001d	0x00003150
0x00003040	0x000003c6	0x00003128	0x00000230	0x00003198	0x00000167	0x000030a8	0x00000257	0x000030c0
0x00003060	0x0000019a	0x00003230	0x000003a5	0x00003200	0x000002c7	0x00003078	0x000002c9	0x000030f8

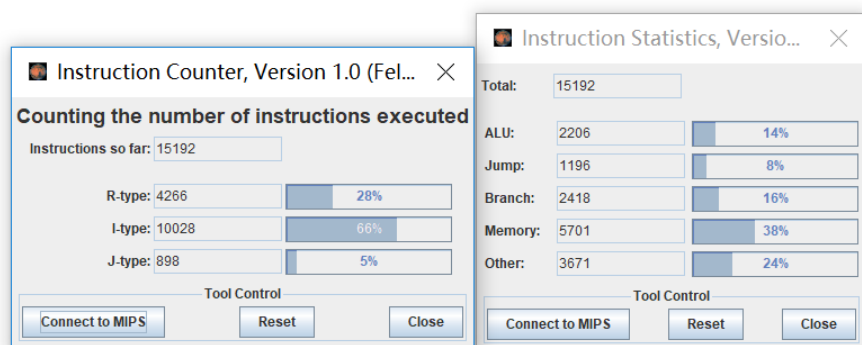
(b)MARS 执行结果

▲图 4-1: CPU 模拟执行程序与 MARS 执行结果对比 (节录)

Merge Sort是以链表形式存储排序结果。本处理器的链表是自0x0000的位置开始存储，每2个字为一节点，其中第一个字为该节点所载的数字；第二个字为下一节点的地址；而MARS分配的空间则是自0x3000的位置开始存储，余与本处理器相同。可以看到排序结果中，除了每个节点所储存下一节点的地址前缀有所不同外，余皆相同。

(二) CPU执行周期数统计：

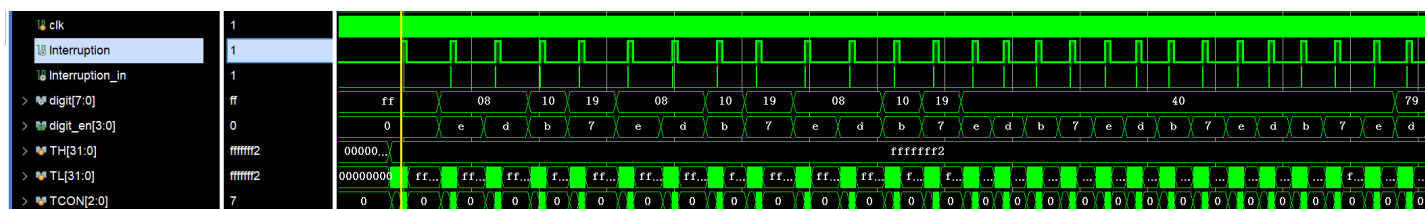
模拟结果中，CPU在周期为10ns的情形下，在80ns处完成读入Systick的初值；在188,660ns处完成读入Systick的终值，其总周期数记于\$28寄存器中，值为0x49aa=18,858。MARS统计的指令总数为15,192。本程序平均CPI约为1.24。



▲图 4-2: MARS统计指令数

(三) CPU的中断程序检查：

为便于仿真，我们让定时器的中断设定为约10几个周期1次，以避免过长的仿真时间，同时将切换显示数字的频率改成约10几次中断切换1次。



▲图 4-2: CPU 模拟执行程序与 MARS 执行结果对比

可以看到数码管的使能信号digit_en规律的在0xe(1110), 0xd(1101), 0xb(1011), 0x7(0111)切换。且显示数字的译码信号digit亦与应该被显示的数字相符。如在本程序中, 第一个被显示的数字是0x49aa (即执行总周期数), 自低位到高位扫描显示的过程中, 分别显示a (对应译码0x08)、a、9 (对应译码0x10)、4 (对应译码0x07); 下一个被显示的数字是0x0000 (本程序在输出执行总周期数与排序结果之间, 均以0x0000间隔以资识别), 无论digit_en为何皆输出为0 (对应译码0x40)。

(四) 调试过程中所遇到的坑包括但不限于以下几点：

- 一开始使用 Vivado 2017.3版的程式来跑Simulation，发现在寄存器的代码编写正确的前提下，有些时刻寄存器竟在复位信号reset==0的情形下意外重设，我反反复复检查了好几遍，也请同学帮我检查，都看不出端倪，最后换了Vivado 2018.3版来跑Simulation，一切正常。☺



游子權

謎之寄存器

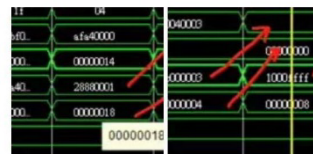
明明reset的值都是0

有時正常將下一個狀態打入寄存器中

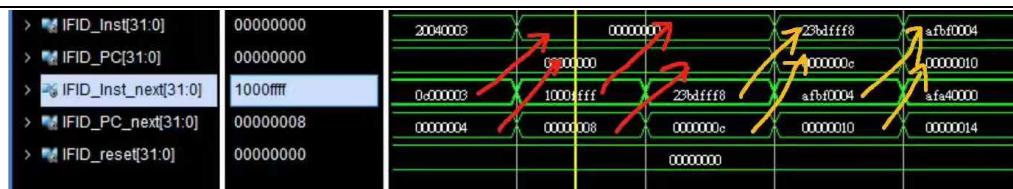
有時卻自動復位成0

這到底是什麼神秘力量呢😊

不管了我要先睡了😊



2019年8月9日 23:40 删除



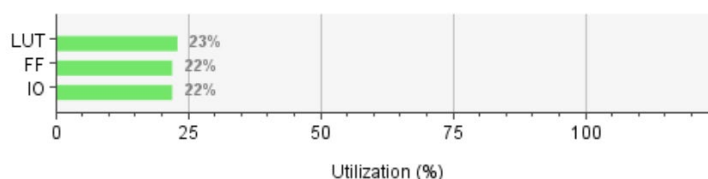
▲图 4-3：谜之寄存器

- wire 连接错误。如将MemWrite误连至RegWrite。
- 因中途变更设计导致部分变数（信号）在不同模组中位长不一致，定义亦不一，进而导致执行时错误。
- 忽略了Verilog预设各变数都是无符号数，因而不能以 ≥ 0 、 ≤ 0 直接判断正负，应比较其最高位或将该变数设为有符号数。
- 忽略了CPU阻塞时，不仅ID/EX寄存器需要清零，IF/ID寄存器需要维持原样，且其优先序应比「跳转使IF/ID清零」还要优先。否则在load-jr或load-beq, bne...之类指令时，此时同时发生stall和Jump（或Branch），若让「跳转使IF/ID清零」优先于「阻塞使IF/ID保持」，则在IF/ID清零后，阻塞后ID阶段执行的是空指令，又本设计中，跳转是在ID阶段作判断，造成该跳转指令「形同虚设」而使程序运行出现异常。
- 忽略了「进入内核态后不允许中断和异常」，未设定条件加以判断，导致进入内核态后，因无法于1个周期内将Exception和Interruption的变量重置为0，导致程序进入死循环。应将其与PC[31]作与运算。
- 发生Exception和Interruption时，因Control Signal出错，导致程序异常或中断时的地址无法存回\$26，进而导致程序返回时异常。
- 族繁不及备载……说多了都是泪……

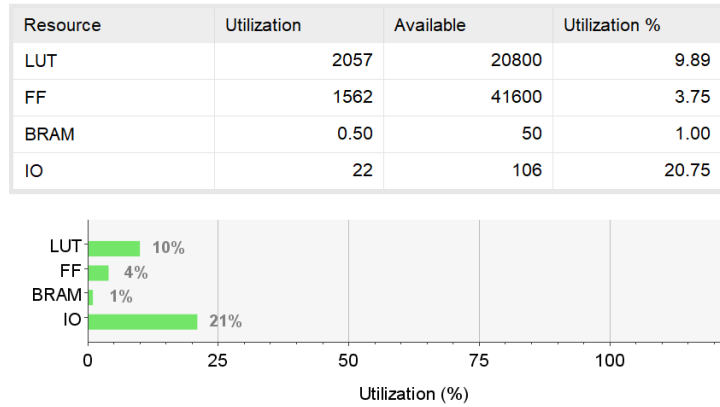
五、性能评价（与单周期对比）：

（一）面积性能

Resource	Utilization	Available	Utilization %
LUT	4868	20800	23.40
FF	9289	41600	22.33
IO	23	106	21.70



(a)单周期



(b)流水线

▲图 5-1: 面积资源使用情形

我的单周期 MIPS 用了 4868 个单位的查找表和 9289 个单位的触发器，占用了 FPGA 约 1/5 的资源；流水线 MIPS 因加入了各阶段的寄存器，理应占用更多资源，但在使用 BRAM 的情形下，大幅减少了 LUT 和 FF 的耗用。

(二) 时序性能：

在电路由 100MHz 时钟驱动的情形下 (周期=10ns)，我的单周期 MIPS 处理器 Setup 时间的裕量为 -2.655ns，故电路工作的最高主频为

$$f_{\max} = \frac{1}{10\text{ns} - (-2.655\text{ns})} \approx 79.02\text{MHz}$$

。我的流水线 MIPS 处理器 Setup 时间的裕量为 0.126ns，故电路工作的最高主频为

$$f_{\max} = \frac{1}{10\text{ns} - (0.126\text{ns})} \approx 101.28\text{MHz}$$

。主频提升了 28%。可见用流水线确能有效减短时钟周期的长度。

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): -2.655 ns	Worst Hold Slack (WHS): 0.166 ns	Worst Pulse Width Slack (WPWS): 4.500 ns
Total Negative Slack (TNS): -11505.947 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 8734	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 26155	Total Number of Endpoints: 26155	Total Number of Endpoints: 8733
Timing constraints are not met.		

(a)单周期

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0.126 ns	Worst Hold Slack (WHS): 0.155 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: 3005	Total Number of Endpoints: 3005	Total Number of Endpoints: 1565
All user specified timing constraints are met.		

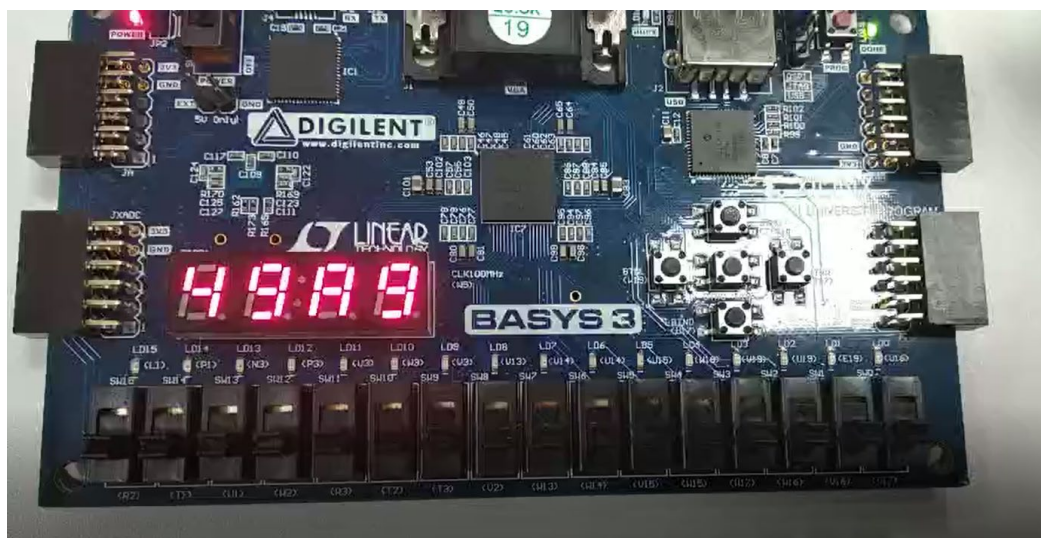
(b)流水线

▲图 1-2: 时序性能情形

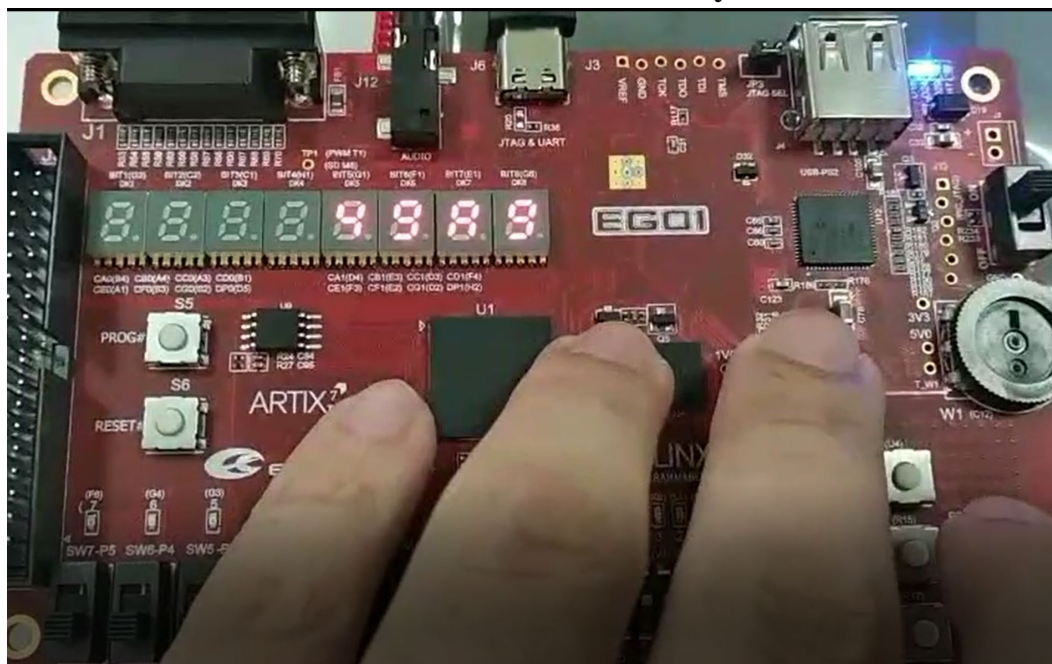
单周期之关键路径应为执行 load word 指令造成，而流水线处理器的关键路径如下图所示。由其所经之相关器件路径，推测应是与 branch 指令有关，需要提取 Register 中的内容（并完成转发）、完成条件判断，决定对 IF/ID 寄存器清零，并且改变 PC 的值。

六、硬件调试情形：

本次流水线处理器自 Simulation 成功起，相较于其他同学，在 Synthesis、Implementation、Generate Bitstream 与硬件调试上，可说是披荆斩棘、势如破竹，丝毫没有遇到任何问题。我认为这是因我的代码写作严谨，且时时查看 Vivado 给出的 Warning 并据以修正所致。硬件调试结果与预期结果相符。详细情形可参见附录影片档。



▲图 6-1：硬件调试情形（Basys3 版）



▲图 6-2：硬件调试情形（EGO1 版）

春季学期的单周期 CPU 完成一个指令最短需时 $12.655\text{ns} \times 1\text{cycle/inst} = 12.655\text{ns/inst}$
 本学期之流水线 CPU 完成一个指令最短需时 $9.874\text{ns} \times 1.24\text{cycle/inst} = 12.244\text{ns/inst}$
 流水线比单周期快了 4%。虽然不显著，但也是值得为自己喝采的成就了！

七、思想体会

不啰嗦，直接上图。

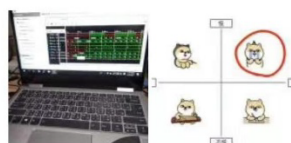


游子權

每天起床第一句 先給自己打個氣
每次多追一秒劇 都要說聲對不起
魔鏡魔鏡看看我 我的時間在哪裡
努力 我要努力 我要寫出流水線
Pipe! line!
我要寫出流水線!
Pipe! line!

為了寫出流水線 天天提著一口氣
為了完成小學期 吃草吃成沙拉精
天生才智難自棄 可惜玩啥我都不膩
驗收 我要驗收 我要寫出流水線

流水線流水線流水
流水線流水線流水
流水線流水線流水
流水線流水線流水
流水線我的天敵
燃燒我的意志力!



2019年8月27日 08:56 刪除



游子權

【調死人償命的MIPS流水線CPU】👾👾👾

FPGA實際週期數: 0x49a9 = 18857
MARS統計指令數: 15192
CPI ≈ 1.24
時序裕量 (100MHz時鐘下): 0.090ns
相應最高主頻: 101MHz

小學期2/3🎉🎉🎉
FPGA再您的見了👾👾👾



2019年9月1日 22:37 刪除



▲图 7: 拜拜，FPGA！拿走拿走别客气！

附录与文件列表

附录 A 设计框图全图

附录 B 最终 Vivado Project (请用 Vivado 2019.1 版开启)

档名	描述
/src	所有 Verilog 源代码
/ain.coe	排序数输入

附录 C 相关汇编代码

档名	描述
msort.asm	供 MARS 运行之 Merge Sort 程序
msort_forhardCPU_withInterrupt.asm	供 FPGA 硬件运行之 Merge Sort 及中断程序
msort_forhardCPU_withInterrupt_test.asm	专为仿真设计之 Merge Sort 及中断程序

附录 D 测试影片

档名	描述
Basys3.mp4	Basys3 调试情形
EGO1.mp4	EGO1 调试情形