

静态流水线设计报告

1752669 杨燕

1. 实验内容

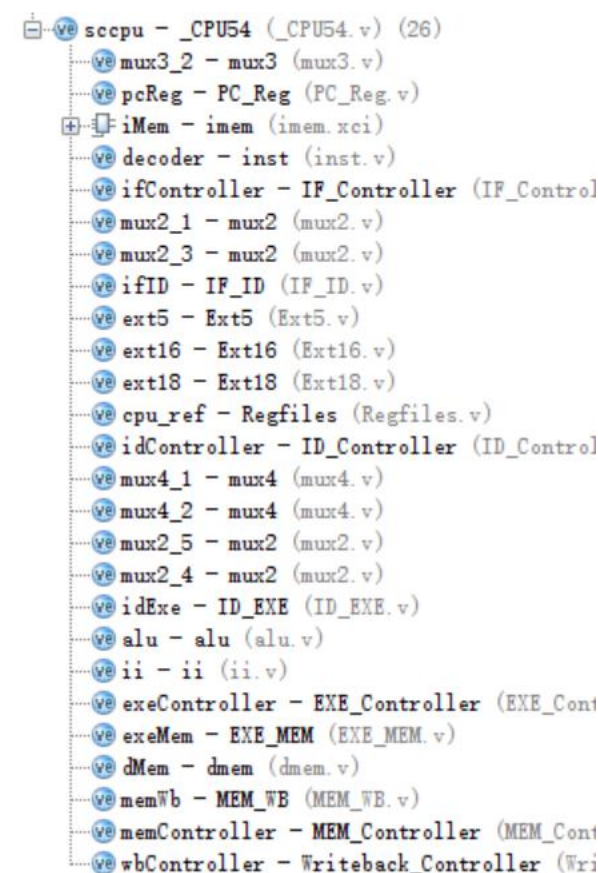
了解静态流水线的实现原理，并实现一个静态流水线模型，从而验证以下数学模型：

```
int a[m], b[m], c[m];
a[i] = a[i - 1] + 2;
b[i] = b[i - 1] + i;
m = 60;
a[0] = 1;
b[0] = 0;
if (0 <= i <= 19) c[i] = a[i];
if (20 <= i <= 39) c[i] = b[i];
if (40 <= i <= 59) c[i] = a[i] + b[i];
```

1.1. 具体方法

采用 Verilog 编写、仿真一个静态流水线，将数学模型转换为汇编代码，再使用 Mars 将汇编代码变成二进制指令，交给 CPU 运行并观察结果。

2. 工程结构



3. 总体设计

静态流水线是指在同一时间内，多功能流水线中的各段只能按同一种功能的连接方式工作的流水线。当流水线要切换到另一种功能时，必须等前面的任务都流出流水线之后，才能改变连接。

3.1. 模块划分

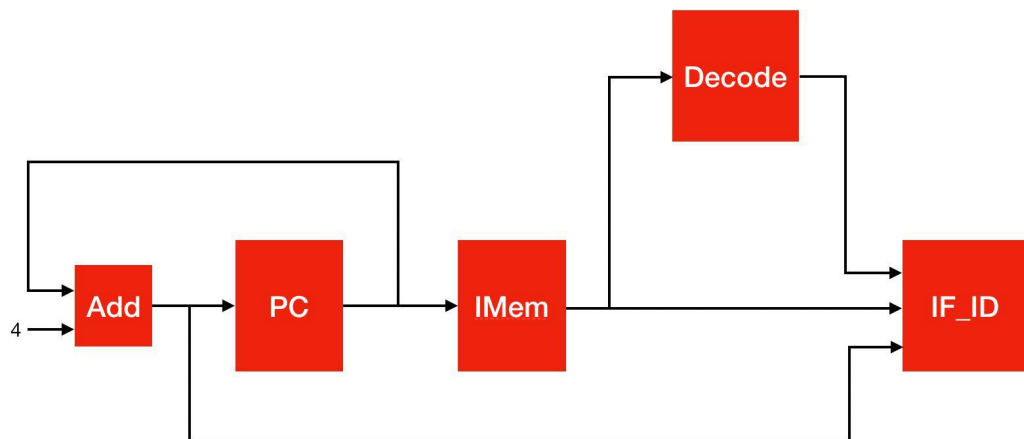
本次流水线是 5 级流水线，因此将一条指令的完整执行划分为 5 个流水段；同时，前后两个流水段之间有一个流水寄存器存放后一个流水段需要的所有数据，统称为流水寄存器，共有 4 个：

- ✓ 取指：将二进制指令取出，存到第一个流水寄存器。
- ✓ 译码：根据指令，将执行阶段所需要的所有操作数（如 ALU 的输入）存到第二个流水寄存器中。
- ✓ 执行：根据指令执行相应操作（如 add 则让 alu 执行加法），并将结果存到第三个流水寄存器。

- ✓ 访存：如果是 **lw**、**sw** 这种内存访问指令，执行相应的读取、写入操作，并将读取结果存到第四个流水寄存器。
- ✓ 回写：将运算结果存到寄存器中。

3.2. 关键点

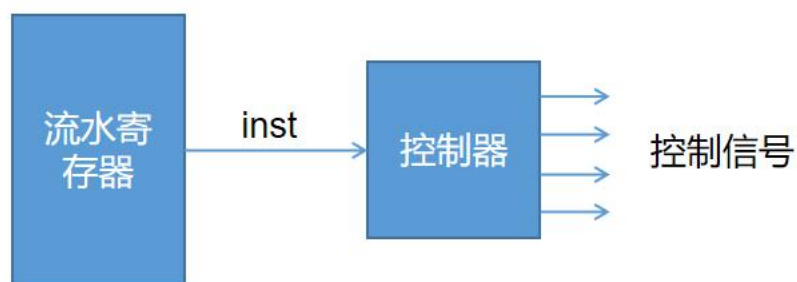
整个设计的关键在于，各个功能段相互独立，一个功能段用到的所有数据都在流水寄存器当中。例如对于取值模块，它的通路图如下所示，**IF_ID** 作为流水寄存器需要存储译码阶段需要的东西，有 **pc+4**，指令的具体内容，以及译码器译码的结果：



3.3. 控制器

本次设计是纯底层的，从数据通路开始设计，如果存在一条通路有两个相同的输入则采用选择器 + 控制信号的方式选择。

基本的思想在于，每个功能段所需要的数据从流水寄存器中取（包括指令是什么），将指令输出到该功能段的控制器中，控制器根据指令打开相应的数据通路（即输出相应的控制信号）。



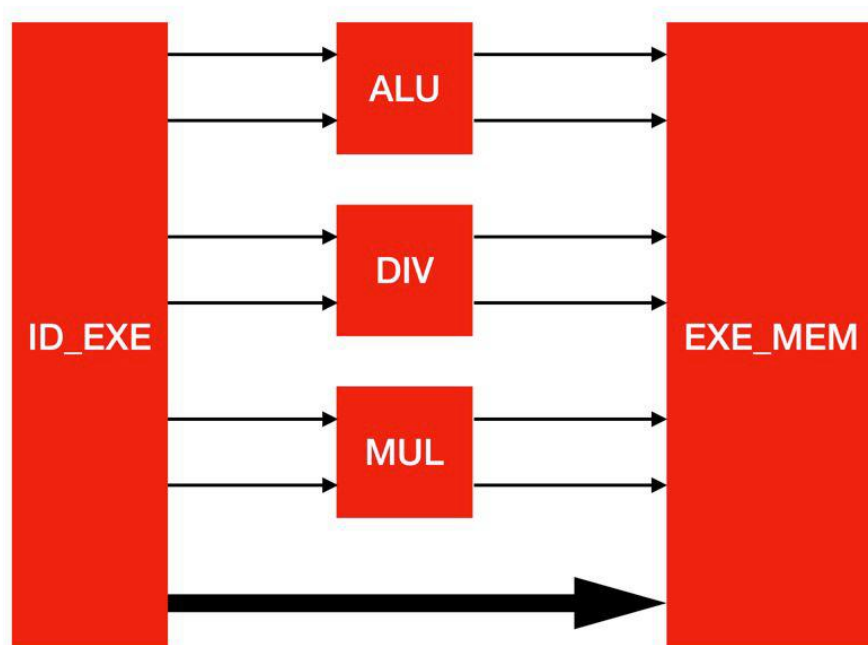
因此，每一个功能段会有相应的控制器，一共有 5 个控制器，根据指令进行控制信号的发出。

4. 设计细节

本次设计参考了实验指导书的思想，但对实验指导书的设计方式做了改良，使总体的设计更加清晰，逻辑更贴近底层，具体体现在同一个输入有多个数据源时采用多路选择器 + 控制信号，而非 if-else 语句。这样会使代码贴近硬件，执行效率高。

4.1. 数据通路的设计

除了访存、CPO 指令，大部分指令都是在执行阶段通过 ALU、除法乘法器执行，结果存入之后的流水寄存器，如图所示：



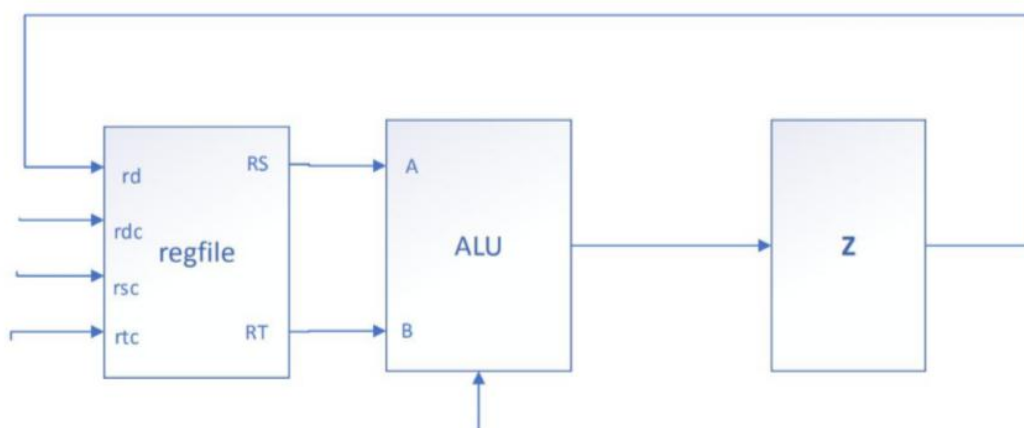
不过为了构建总体的数据通路图，需要直到每一条指令的数据通路。

每一条指令都有它的数据通路，即如何执行该指令。大部分指令在执行阶段只需要一个周期，这里列出部分指令的数据通路，这里的主要参考之前所做的 54 条指令多周期 CPU。

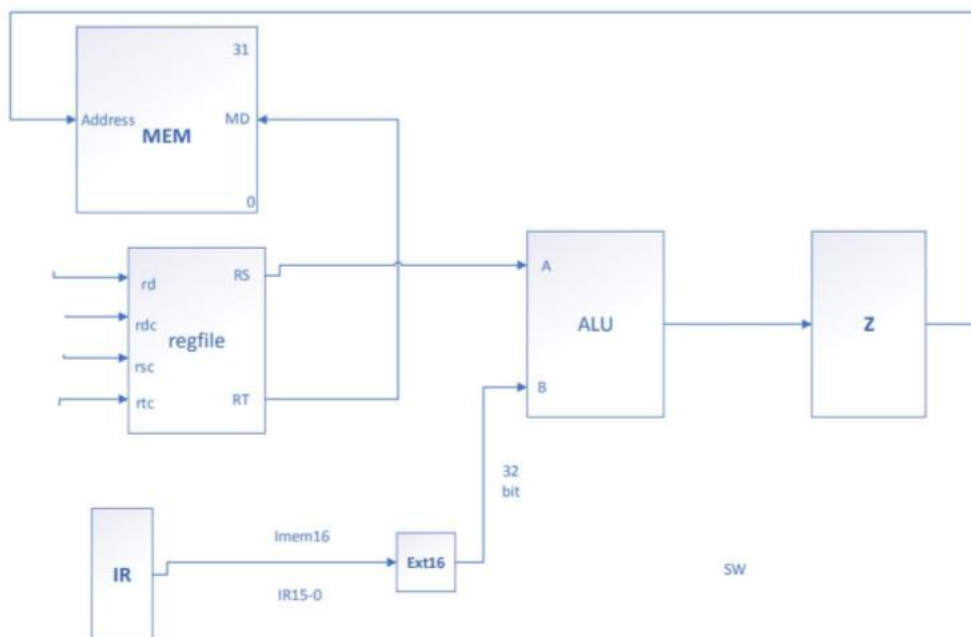
注意，实际通路中数据从流水寄存器取而非图中所示！！

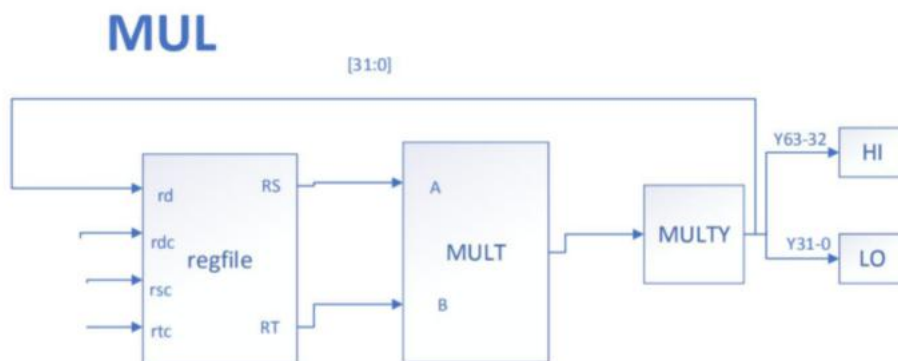
4. 1. 1. ADD 等

ADD\ADDU\SUB\SUBU\AND\XOR\OR\NOR\SLV\STL\SLTU\SRAV\SRLV



4. 1. 2. SW





4.2.1. 顶层模块

(这里只列出一部分)

```
// PC -> (in, out, clk, rst, ena)
// wire pcEnable;
// wire [31:0] pcOut;
PC_Reg #(. startAddress(32'h00400000)) pcReg(clock, reset, 1, pcIn, pcOut);

// IMem -> (pc, inst)
wire [31:0] inst;
imem iMem(pcOut[12:2], inst);

// decoding
wire [53:0] ifInst;
inst decoder(inst, ifInst);

// IfController
wire mux2_1Select;
IF_Controller ifController (
    .inst(ifInst),
    .mux2_1Select(mux2_1Select)
);
```

```
wire [31:0] ifWAddr;
mux2 mux2_1(
    .a({27'b0, inst[15:11]}),
    .b({27'b0, inst[20:16]}),
    .choose(mux2_1Select),
    .Mux2select(ifWAddr)
);

// 如果遇到转移指令，则指令清空
// 例如：
// j L2
// addi $1, $0, 100
// 不清空会导致第二条 addi 取指
wire [53:0] idInstIn;
wire mux2_3Select;
mux2 #(.width(54)) mux2_3(ifInst, 54'b0, mux2_3Select, idInstIn);
```

```

// (in, out)
wire [31:0] ext5Out;
Ext5 ext5(originInst[10:6], ext5Out);

wire [31:0] ext16Out;
Ext16 ext16({16'b0, originInst[15:0]}, 1, ext16Out);

wire [31:0] ext18Out;
Ext18 ext18(originInst[15:0], ext18Out);
assign jumpAddr_bne = ext18Out + pcOut;

// (clk, rst, wena, ov, rsAddr, rtAddr, waddr, wdata, rs, rt)
wire regFilesWEna, overflow;
// wire [31:0] mux2Out;
// 这两个值从最后的流水寄存器取
wire [31:0] wbWAddr, wbWData, rsOut, rtOut;
Regfiles cpu_ref(clock, reset, regFilesWEna, 0, originInst[25:21], originInst[20:16], wbWAddr, wbWData, rsOut, rtOut);

```

```

// wire [2:0] dMemControl;
wire dMemEna;
dmem dMem(clock, dMemWEna, ,dMemWAddr, memDMemWData, 6,);

//pipereg MEM_WB
// wire [31:0]mux2_data_out;
wire [53:0] wbInst;
// wire [31:0] wbWData;
MEM_WB memWb(
    .clock(clock),
    .reset(reset),
    .enable(1),
    .instIn(memInst),
    .inst(wbInst),
    .waddrIn(memWAddr),
    .waddr(wbWAddr),
    .wDataIn(D_ALUR),
    .wData(wbWData)
    // .mux2_data(mux2Out),
    // .mux2_data_out(mux2_data_out)
);

```

4.2.2. 流水寄存器

由于实现贴近底层，流水寄存器仅做寄存作用，不需要考虑逻辑。逻辑的处理通过选择器 + 控制器发出的控制信号决定，因此这里只列出一个流水寄存器：


```

module IF_ID(
    input clock, reset, enable,
    input [53:0] instIn, // 译码器译码后的指令
    output reg [53:0] inst,
    input [31:0] wAddrIn,
    output reg [31:0] wAddr,
    input [31:0] originInstIn,
    output reg [31:0] originInst,
    input [31:0] pcPlus4In,
    output reg [31:0] pcPlus4
);

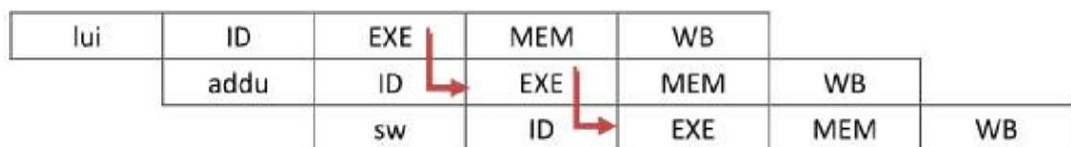
```

4.2.3. 控制器

控制器的代码在数据冲突中会有所说明。

5. 数据冲突

流水线中存在许多的数据冲突，静态流水线中只存在先写后读的冲突，本次设计中使用的处理方法是相关作用路径。



5.1. 控制器

大部分的控制器的作用是选择器的信号选择，然而，对于数据冲突，也可以用控制器来处理。具体的方法是将所有会产生冲突的数据连接到一个多路选择器中，判断所需的信息连到控制器里，满足相应的条件则选择相应的路。

5.1.1. 数据选择器

```

mux4 mux4_1(rsOut, aluOut, D_ALUR, wbWData, mux4_1Select, idRs);

mux4 mux4_2(rtOut, aluOut, D_ALUR, wbWData, mux4_2Select, idRt);

```

从各个流水寄存器中存到的中间结果连回选择器进行选择。

5.1.2. 控制器模块

模块定义：

```
module ID_Controller(  
    input [53:0] inst,  
    // 后面模块连回来判断  
    input [31:0] exeWAddr,  
    input [31:0] memWAddr,  
    input [31:0] wbWAddr,  
    input [31:0] rsAddr,  
    input [31:0] rtAddr,  
    input [31:0] rsData,  
    input [31:0] rtData,  
    output reg [1:0] mux4_1Select,  
    output reg [1:0] mux4_2Select,  
    output reg [2:0] mux5_1Select,  
    output reg [1:0] mux3_2Select,  
    output reg mux2_3Select,  
    output reg mux2_4Select,  
    output mux2_5Select,  
    output PC_enable  
);
```

选择正确的 Rt:

```
// 找 rt  
always @(*) begin  
    // 相关作用路径打开  
    // 如果是 add rd, rs, rt, 且 rd == rt  
    if ((inst_add | inst_or | inst_and | inst_bne | inst_sw | inst_lui | inst_srl | inst_addu | inst_mul))  
        if (rtAddr == exeWAddr & exeWAddr != 0)  
            mux4_2Select = 2'd1;  
        else if (rtAddr == memWAddr & memWAddr != 0)  
            mux4_2Select = 2'd2;  
        else if (rtAddr == wbWAddr & wbWAddr != 0)  
            mux4_2Select = 2'd3;  
        else  
            mux4_2Select = 2'd0;  
    else  
        mux4_2Select = 2'd0;  
end
```

选择正确的 Rs:

```

// 如果是 add rd, rs, rt 且 rd == rs
// addi rt, rs, imm
// 找 rs
always @(*) begin
    if (inst_add | inst_or | inst_and |
        inst_addi | inst_ori | inst_andi | inst_bne | inst_sw | inst_slti | inst_addu | inst_mul) begin
        if (rsAddr == exeWAddr & exeWAddr != 0)
            mux4_1Select = 2'd1;
        else if (rsAddr == memWAddr & memWAddr != 0)
            mux4_1Select = 2'd2;
        else if (rsAddr == wbWAddr & wbWAddr != 0)
            mux4_1Select = 2'd3;
        else
            mux4_1Select = 2'd0;
    end
end
end

```

对跳转指令的控制：

```

// 跳 pc
always @(*) begin
    if (inst_j) begin
        mux3_2Select = 2'd1;
        mux2_3Select = 1'd1;
    end else if (inst_bne && rsData != rtData) begin
        mux3_2Select = 2'd2;
        mux2_3Select = 1'd1;
    end else begin
        mux3_2Select = 2'd0;
        mux2_3Select = 1'd0;
    end
end
end

```

其他的选择器控制信号：

```

always @(*) begin
    if (inst_addi | inst_ori | inst_andi | inst_sw | inst_lui | inst_slti)
        mux2_4Select = 1'd1;
    else
        mux2_4Select = 1'd0;
end

assign mux2_5Select = inst_srl;

```

6. 验证结果

将数学模型转换成汇编代码，然后变成 coe 文件导入 IMem 验证结果。

6. 1. 汇编代码

.data

A:.space 240

B:.space 240

C:.space 240

D:.space 240

E:.space 240

.text

j main

exc:

nop

j exc

main:

addi \$2,\$0,0 #a[i]

addi \$3,\$0,1 #b[i]

addi \$4,\$0,0 #c[i]

addi \$13,\$0,0 #d[i]

addi \$5,\$0,4 #counter

addi \$6,\$0,0 #a[i-1]

addi \$7,\$0,1 #b[i-1]

addi \$10,\$0,0 #flag for i<20 || i<40

addi \$11,\$0,240 #sum counts

addi \$14,\$0,3

addi \$30,\$0,0

把 0 1 0 0 (\$2,...,\$13) 分别存入 A B C D

lui \$27,0x0000

addu \$27,\$27,\$0

sw \$2,A(\$27)

lui \$27,0x0000

addu \$27,\$27,\$0

sw \$3,B(\$27)

lui \$27,0x0000

addu \$27,\$27,\$0

sw \$2,C(\$27)

lui \$27,0x0000

addu \$27,\$27,\$0

sw \$3,D(\$27)

循环

loop:

\$5(4) 除以 4 (=i) 存入 \$12

srl \$12,\$5,2

\$6 加 i. 自此,\$6 就是 a[i] 而不是 a[i-1] 了

add \$6,\$6,\$12

把 a[i] 的内容存入 A[i] 中

lui \$27,0x0000

addu \$27,\$27,\$5

sw \$6,A(\$27)

\$14 (3) 乘以 \$5/4 (i) (= 3i)

mul \$15,\$14,\$12

把 \$7 (b[i-1]) 的内容加上 3i, 存入 B[i]. 自此,\$7 就是 b[i] 而不是 b[i-1]

了

add \$7,\$7,\$15

lui \$27,0x0000

```
addu $27,$27,$5
```

```
sw $7,B($27)
```

```
# $5 是否小于 80 (i 是否小于 20)? 记入 $10
```

```
slti $10,$5,80
```

```
# 若不是, 跳转
```

```
bne $10,1,c1
```

```
# (0<=i<=19)
```

```
# 把 $6 的内容存入 C[i] 中 (c[i] = a[i])
```

```
lui $27,0x0000
```

```
addu $27,$27,$5
```

```
sw $6,C($27)
```

```
# 把 $7 的内容存入 D[i] 中 (d[i] = b[i])
```

```
lui $27,0x0000
```

```
addu $27,$27,$5
```

```
sw $7,D($27)
```

```
addi $15,$6,0 # $15 $16 分别赋值为 c[i] d[i]
```

```
addi $16,$7,0
```

```
j endc
```

```
c1: # (20<=i<=39)
```

```
# i 是否小于 40 ? 若不是, 跳转到 c2
```

```
slti $10,$5,160
```

```
addi $27,$0,1
```

```
bne $10,$27,c2
```

```
# C[i] = a[i] + b[i]
add $15,$6,$7
lui $27,0x0000
addu $27,$27,$5
sw $15,C($27)
```

```
# D[i] = a[i] * b[i]
mul $16, $15,$6
lui $27,0x0000
addu $27,$27,$5
sw $16,D($27)
```

```
j endc
```

```
c2: # (i>=40)
```

```
# C[i] = a[i] * b[i]
mul $15,$6,$7
lui $27,0x0000
addu $27,$27,$5
sw $15,C($27)
```

```
# D[i] = c[i] * b[i]
mul $16,$15,$7
lui $27,0x0000
addu $27,$27,$5
sw $16,D($27)
```

```

endc:

add $28,$15,$16    # $28 = c[i] + d[i]

lui $27,0x0000

addu $27,$27,$5

sw $28,E($27)      # 将 c[i] + d[i] 存入 E[i]

addi $5,$5,4       # i = i + 1

bne $5,$11,loop    # i = 60 不跳转

break

```

最后 $E[i] = c[i] + d[i]$, 可通过验证 $E[59]$ 的正确性来验证 $c[i]$ 和 $d[i]$ 正确性

6.2. 测试代码

```

module CPU54_tb();

reg clock = 0;
reg reset = 1;
//reg [31:0] counter = 0;
//integer file;

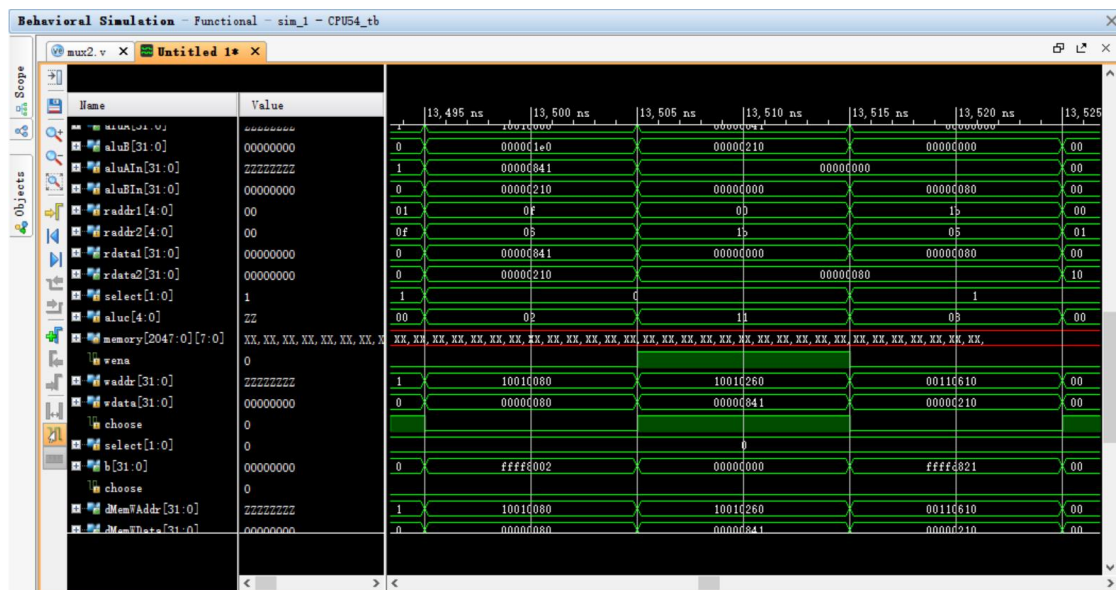
//initial file = $fopen("C:/Users/YY WYX/Desktop/result.txt");

initial #7 reset = 0;
always #5 clock = ~clock;

sccomp_dataflow s(clock, reset);

```


6.3. 波形纵览

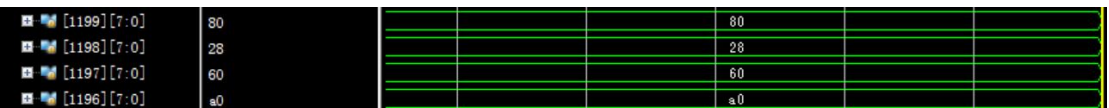


6.4. 验证 E[59]

由于这次实现的 DMem 中采用字节存储而非字存储（8 位的数组）：

```
reg [7:0] memory [2048-1:0];
```

因此，查看 DMem 的第 $(240 + 59) * 4 = 1196$ 位：





























拼起来，结果是 0xa0602880，结果正确！

6.5. 寄存器的结果

与 mars 做对比，完全一致：

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x100100ec
\$v0	2	0x00000000
\$v1	3	0x00000001
\$a0	4	0x00000000
\$a1	5	0x000000f0
\$a2	6	0x000006ea
\$a3	7	0x000014bf
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x000000f0
\$t4	12	0x0000003b
\$t5	13	0x00000000
\$t6	14	0x00000003
\$t7	15	0x008f7096
\$s0	16	0x9fd0b7ea
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x000000ec
\$gp	28	0xa0602880
\$sp	29	0x00000000
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400194
hi		0x0000000b
lo		0x9fd0b7ea

[31] [31:0]	00000000
[30] [31:0]	00000000
[29] [31:0]	00000000
[28] [31:0]	a0602880
[27] [31:0]	000000ec
[26] [31:0]	00000000
[25] [31:0]	00000000
[24] [31:0]	00000000
[23] [31:0]	00000000
[22] [31:0]	00000000
[21] [31:0]	00000000
[20] [31:0]	00000000
[19] [31:0]	00000000
[18] [31:0]	00000000
[17] [31:0]	00000000
[16] [31:0]	9fd0b7ea
[15] [31:0]	008f7096
[14] [31:0]	00000003
[13] [31:0]	00000000

  [12][31:0]	0000003b
  [11][31:0]	000000f0
  [10][31:0]	00000000
  [9][31:0]	00000000
  [8][31:0]	00000000
  [7][31:0]	000014bf
  [6][31:0]	000006ea
  [5][31:0]	000000f0
  [4][31:0]	00000000
  [3][31:0]	00000001
  [2][31:0]	00000000
  [1][31:0]	100100ec
  [0][31:0]	00000000

7. 下板结果

将 DMem 的 1196 字节开始的四个字节的值接到数码管模块，连到板子上观察结果：

