lab 3分析与设计

lab 3共有两个任务：

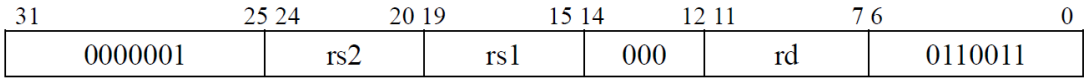
- 在原有的基础上增加实现乘除法指令，要求实现多周期乘除法器，不能用`/`和`%`运算符；
- 实现缓存，优化处理器访存；

一、增加实现的指令与功能

1、mul（R-type）

mul rd, rs1, rs2　　$x[rd] = x[rs1] \times x[rs2]$

乘*(Multiply)*. R-type, RV32M and RV64M.
把寄存器 x[*rs2*]乘到寄存器 x[*rs1*]上，乘积写入 x[*rd*]。忽略算术溢出。



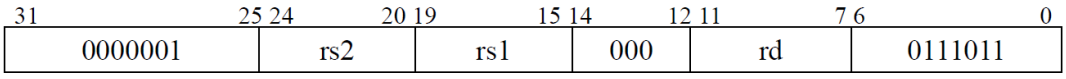
把两个寄存器数据`rd1`和`rd2`相乘的结果写入目标寄存器；

alu操作为：`MUL`，结果写回寄存器；

2、mulw（R-type）

mulw rd, rs1, rs2　　$x[rd] = sext((x[rs1] \times x[rs2])[31:0])$

乘字*(Multiply Word)*. R-type, RV64M only.
把寄存器 x[*rs2*]乘到寄存器 x[*rs1*]上，乘积截为 32 位，进行有符号扩展后写入 x[*rd*]。忽略算术溢出。



把两个寄存器数据`rd1`和`rd2`相乘的结果64位结果，截取低32位，进行符号扩展后的结果写入目标寄存器；

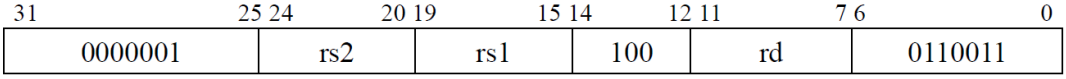
alu操作为`MUL`，结果写回寄存器；

alu的处理逻辑为：首先把两个数据进行截断，得到低32位（高位无符号扩展），得到后把两个数据送入乘法器运算，得到的结果截取低32位进行扩展得到alu结果；

3、div（R-type）

div rd, rs1, rs2　　$x[rd] = x[rs1] \div_s x[rs2]$

除法*(Divide)*. R-type, RV32M and RV64M.
用寄存器 x[*rs1*]的值除以寄存器 x[*rs2*]的值，向零舍入，将这些数视为二进制补码，把商写入 x[*rd*]。



把两个寄存器数据`rd1`和`rd2`相除的结果写入目标寄存器；

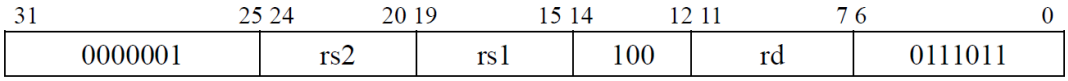
alu操作为：`DIV`，结果写回寄存器；

4、divw（R-type）

divw rd, rs1, rs2　　$x[rd] = sext(x[rs1][31:0] \div_s x[rs2][31:0])$

字除法*(Divide Word)*. R-type, RV64M.

用寄存器 x[*rs1*]的低 32 位除以寄存器 x[*rs2*]的低 32 位，向零舍入，将这些数视为二进制补码，把经符号位扩展的 32 位商写入 x[*rd*]。



把两个寄存器数据`rd1`和`rd2`低32位进行相除的结果写入目标寄存器；

alu操作为：`DIV`，结果写回寄存器；

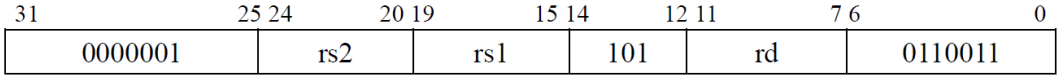
alu的处理过程与`mulw`一致，先截取操作数，进行计算，随后扩展得到的结果；

5、divu（R-type）

divu rd, rs1, rs2　　$x[rd] = x[rs1] \div_u x[rs2]$

无符号除法*(Divide, Unsigned)*. R-type, RV32M and RV64M.

用寄存器 x[*rs1*]的值除以寄存器 x[*rs2*]的值，向零舍入，将这些数视为无符号数，把商写入 x[*rd*]。



把两个寄存器数据`rd1`和`rd2`无符号相除的结果写入目标寄存器，也就是把数据视为无符号数；

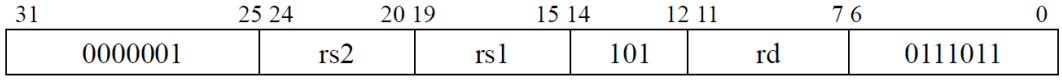
alu操作为：`DIVU`，结果写回寄存器；

6、divuw（R-type）

divuw rd, rs1, rs2　　$x[rd] = sext(x[rs1][31:0] \div_u x[rs2][31:0])$

无符号字除法*(Divide Word, Unsigned)*. R-type, RV64M.

用寄存器 x[*rs1*]的低 32 位除以寄存器 x[*rs2*]的低 32 位，向零舍入，将这些数视为无符号数，把经符号位扩展的 32 位商写入 x[*rd*]。



把两个寄存器数据`rd1`和`rd2`低32位进行无符号相除的结果写入目标寄存器，把数据视为无符号数；

alu操作为：`DIVU`，结果写回寄存器；

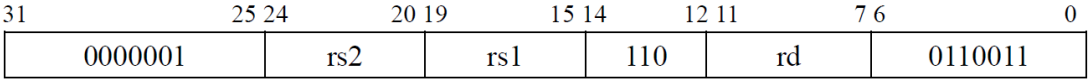
alu的处理过程与`mulw`一致，先截取操作数，进行计算，随后扩展得到的结果；

7、rem（R-type）

rem rd, rs1, rs2 $x[rd] = x[rs1] \%_s x[rs2]$

求余数*(Remainder)*. R-type, RV32M and RV64M.

x[*rs1*]除以 x[*rs2*]，向 0 舍入，都视为 2 的补码，余数写入 x[*rd*]。



把两个寄存器数据`rd1`和`rd2`余数写入目标寄存器，即取模运算；

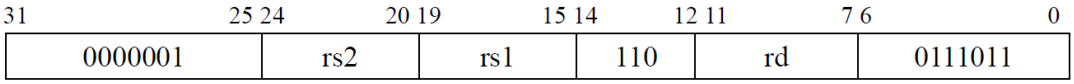
alu操作为：`MOD`，结果写回寄存器；

8、remw（R-type）

remw rd, rs1, rs2 $x[rd] = sext(x[rs1][31:0] \%_s x[rs2][31:0])$

求余数字*(Remainder Word)*. R-type, RV64M only.

x[*rs1*]的低 32 位除以 x[*rs2*]的低 32 位，向 0 舍入，都视为 2 的补码，将余数的有符号扩展写入 x[*rd*]。



把两个寄存器数据`rd1`和`rd2`低32位进行取模的结果写入目标寄存器；

alu操作为：`MOD`，结果写回寄存器；

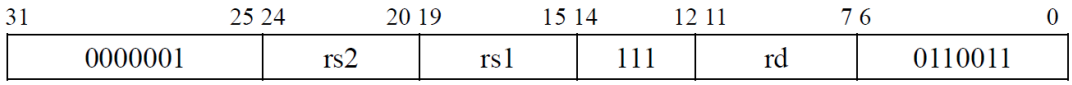
alu的处理过程与`mulw`一致，先截取操作数，进行计算，随后扩展得到的结果；

9、remu（R-type）

remu rd, rs1, rs2 $x[rd] = x[rs1] \%_u x[rs2]$

求无符号数的余数*(Remainder, Unsigned)*. R-type, RV32M and RV64M.

x[*rs1*]除以 x[*rs2*]，向 0 舍入，都视为无符号数，余数写入 x[*rd*]。



把两个寄存器数据`rd1`和`rd2`余数写入目标寄存器，数据视为无符号数；

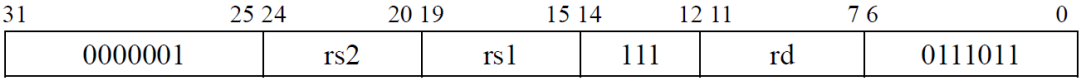
alu操作为：`MODU`，结果写回寄存器；

10、remuw（R-type）

remuw rd, rs1, rs2 $x[rd] = sext(x[rs1][31:0] \%_u x[rs2][31:0])$

求无符号数的余数字*(Remainder Word, Unsigned)*. R-type, RV64M only.

x[*rs1*]的低 32 位除以 x[*rs2*]的低 32 位，向 0 舍入，都视为无符号数，将余数的有符号扩展写入 x[*rd*]。



把两个寄存器数据`rd1`和`rd2`低32位进行无符号取模的结果写入目标寄存器；

alu操作为：`MODU`，结果写回寄存器；

alu的处理过程与`mulw`一致，先截取操作数，进行计算，随后扩展得到的结果；

二、alu的改进

为了支持乘除法指令，需要实现多周期的：乘法器`multiplier`，除法器`divider`；两种种运算都采用有限状态机控制的方式，通过竖式计算的逻辑实现电路计算；（`multiplier.sv`和`divider.sv`为两个参考的32位多周期乘除法器）

假设两种运算器都实现后，alu中应改进为时序逻辑并始终连接运算器，以乘法器为例，

乘法器的输入输出端口如下：

```
module multiplier (
    // 输入端口：时钟信号，复位信号，有效信号，符号计算信号与两个操作数
    input logic clk, reset, valid, sign,
    input u64 srca, srcb,
    // 输出端口：计算结束的握手信号与计算结果
    output logic done,
    output u64 result
);
```

alu的乘法**运算逻辑**如下：

- alu中判断是否为乘法操作，如果命中，则把乘法器有效信号置为1，发起一次乘法运算请求；
- 无论alu中的运算是什么种类，`multiplier`应始终连接两个操作数；
- 乘法操作时，`multiplier`接收到`valid`信号，开始进行乘法计算；
- alu中始终阻塞，当`multiplier`的握手信号`done`为1时，说明计算结果已有效，在下一周期完成一次握手；
- 完成握手后运算结束，alu进行一次握手，开始处理下一条指令；

综上，alu应该由组合逻辑调整为时序逻辑，接收同步时钟信号，且乘法器的线路逻辑如下：

```
module alu(
    // ...
);
    // 处理操作数
    // ...

    // 进行计算的组合逻辑
    always_comb begin
        unique case(alu_func)
            // 其他运算...
            // 乘法运算，乘法运算置位，开始运算
            MUL : mul_valid = 1'b1;
        endcase
    end

    // 处理结果
    // ...

    // 对外生成阻塞信号，运算器未计算结束时
    assign stall = (mul_valid || div_valid || mod_valid) && ~done;

    // 增加时序处理握手
    always_ff(posedge clk) begin
        // 上次运算结束，且下次运算不是乘法，则把mul_valid置为0；
        if(done && (alu_func != MUL) ) begin
            mul_valid = 1'b0;
```

```
        end
    end

    // 连接乘法器
    multiplier multiplier(
        .clk(clk), .reset(reset),
        .valid(mul_valid), .sign(sign),
        .srca(srca), .srcb(srcb),
        .done(done),
        .result(result)
    );
endmodule
```

除法器和取模器逻辑相同，需要增设对应的控制信号。

三、缓存设计

1、缓存参数

本次lab需要实现的缓存参数如下：

- word = 8 Byte　　1个字为8字节
- cache line = 16 words　　1个cache块为16字　　块内偏移offset为4位
- cache ways = 4　　cache结构为4路组相联
- cache sets = 4　　cache共4组　　缓存索引index为2位
- cache lines = 16　　cache存储共16块

2、缓存的端口设计

缓存处于cpu和memory之间，它的行为如下：

与cpu交互的一对端口：

- 接收cpu传入的请求信号，包括请求是否有效`vaild`，请求类型由`strobe`表示，请求地址`address`，请求数据`data`；
- 输出cpu访存请求的响应信号，包括握手信号`data_ok`和读取的数据；

与memory交互的一对端口：

- 发起对主存的访存请求信号，包括请求是否有效`valid`，请求类型由`strobe`表示，请求的块地址`address`，请求的数据块`data`；
- 接收主存的请求响应信号，包括单个数据握手信号与整个块的握手信号，以及读取到的块数据；

因此cache的端口设计如下，以`DCache`数据缓存为例：

```
module DCache(
    input clk, reset,
    // 与处理器的通信端口
    input dbus_req_t dreq,          // 处理器的访存请求
    output dbus_resp_t dresp,       // 处理器的访存响应
    // 与主存的通信端口
    output cbus_req_t creq,         // 对主存发起的访存请求
    input cbus_resq_t cresp,            // 从主存得到的响应
);
    // ...
endmodule
```

3、缓存的状态转化与行为

采用有限状态机控制缓存，缓存在不同状态下有不同的行为和下一状态。

缓存状态分为四种：空闲`IDLE`、标志比较`COMPARE`,取数`FETCH`与写回`WRITEBACK`，三者结合输入对应不同的输出与`state_nxt`

（即有限状态机的`input + state -> output + state_nxt`）

状态定义：

```
localparam type state_t = enum logic[2:0] {
    IDLE, COMPARE, ALLOCATE, WRITEBACK
};
```

四个状态的行为与状态函数：

（1）IDLE状态

`IDLE`的缓存处在空闲（或校验数据）的状态时等待处理器的请求；

因为`IDLE`状态下缓存不会处于向主存通信的状态，所以输入的`cresq`信号为无效信号不需要考虑，只需要考虑`dreq`的状态函数关系：

`dreq`中的控制信号只有`valid`一个，以下为`valid`不同信号对状态机的影响（分析`output`与`state_nxt`）：

- `valid == 0`，即未发起有效请求，因此缓存下个周期状态仍未`IDLE`空闲状态，输出为无效数据（`data_ok == 0`）；
- `valid == 1`，即发起有效请求，因此缓存下个周期转换为`COMPARE`状态，进行cache内访存；

（2）COMPARE状态

`COMPARE`的缓存处在判断此次请求是否命中的状态，cache通过从`meta_ram`读取的`meta`判断是否命中（判断`meta.valid`和`meta.tag`），分为两种情况：`hit`与`miss`：

- `hit`时：

  若命中的话，此时`dresp.data`已经为从`data_ram`中读取到的数据，不需要额外的逻辑去得到命中后得到的数据；判断读写请求的情况（`|strobe`），若为写请求，则设置`meta_ram`和`data_ram`的写使能为1，数据为cpu请求的数据，（`meta`的`valid`和`tag`不需要修改，只需要修改`dirty`）；

  下一个状态为`IDLE`；

- `miss`时：

  若缺失的话，此时从读取到待替换的块地址，若`meta.dirty`为1即为脏数据，发起对主存的写请求，下个周期状态为`WRITEBACK`；

  若`meta.dirty`不是脏数据，直接发起读请求，下一个周期为`ALLOCATE`状态；

  不管是读还是写，都需要设置`creq.valid`为1，读的话`dreq.addr`为待读的块首地址（即`dreq.addr`的`tag`和`index`拼接）；写的话`dreq.addr`为要替换的块首地址（即要替换的`meta.tag`和`index`拼接）；

（3）ALLOCATE状态

`ALLOCATE`状态下，缓存处于向主存发送读请求的状态，此时cpu被阻塞，无需考虑`dreq`的输入，输出和下一个状态取决于主存的响应输入（即`cresp`），只需要考虑`cresp`的状态函数关系：

`cresp`中的控制信号有两个：单个数据的握手信号`ready`与 整个事务握手信号`last`，四种组合：`00`、`01`、`10`和`11`，其中`01`情况不存在，考虑剩余三种：

- `00`（即单个与整个事务握手信号均为0），说明仍需等待数据响应，下个周期状态仍为`ALLOCATE`；
- `10`（单个数据握手，但事务不握手），说明一个数据读取完成，接着需要读取下一个地址的字（整个块还未完成），

  这个周期中需要把读到的数据存储，即需要把写回的块地址增加8，然后继续等待下一个字，下个周期状态仍未`ALLOCATE`；
- `11`（单个数据握手，整个事务握手），说明在当前周期整个块的数据传输完成，需要在下个周期前完成存储最后一个块（赋值：`dram.data = cresp.data`），下个周期处理阻塞的处理器请求，即状态为`COMPARE`；

  （此外`ALLOCATE`状态下应该同时修改`meta.tag`，因为块标志随着读取内存而改变）

（4）WRITEBACK状态

`WRITEBACK`状态下，缓存出去向内存中发送写请求的状态，此时cpu被阻塞，逻辑和`ALLOCATE`相同：

`00`与`10`状态下继续写入数据，需要调整`creq.data`，调整为块内下一个字数据，下个状态仍为`WRITEBACK`，

`11`状态下事务握手，写请求结束，下个周期为`ALLOCATE`状态，以块为单位读取；

4、组相联中cache set的结构

cache采取逻辑与存储分离的结构，cache分为两部分：

`ram`中存储的data和meta（例化`data_ram`和`meta_ram`两个存储元件）`cache_set_array`中存储的每个cache line的首地址；在`ram`中，data与meta都以word为单位进行存储，一个cache块占16个存储单元；

- 考虑如何读出cache line（逐个字读）

  `data_ram`为单端口，读取时应该根据`meta`进行判断是否命中以及命中的cache line在`ram`中的地址（data和meta的存储地址相同对应），将该地址传给`data_ram`从而实现正确的cache line读取，读取到的cache再offset；
- 考虑如何写入cache line（逐个字写）

  从cache set的数组中判断替换地址（即要写入的块地址），并以字为单元逐个写入（调整）；

ram结构：16个块每个块16个字，data_ram共256个存储单元，地址宽为8`。ram为单端口，即一次只会读出一个，需要把cache命中的地址传入（包括offset在内），直接精确到字；（写相同）

以字为存储单元以供和cpu通信，同一个块内的数据连续存放（略去末3位）。

5、cache与cpu通信

cache与cpu通信依靠`dreq`与`dresp`，通信的数据单元为字。请求与端口响应信号：

```
// dbus_req_t
typedef struct packed {
```

```
    logic    valid;   // 请求是否有效
    addr_t   addr;    // 请求地址（相较于主存的地址，由cache处理为缓存地址）
    msize_t  size;    // 请求的数据大小（字节数），为了支持不同粒度读写
    strobe_t strobe;  // 写请求时写入哪几个字节，若全写入即为'1，读请求为'0
    word_t   data;    // 待写入的数据
} dbus_req_t;

// dbus_resq_t
typedef struct packed {
    logic  addr_ok;  // 缓存是否已经接收访存地址
    logic  data_ok;  // 缓存响应数据的握手信号
    word_t data;     // 缓存响应的数据
} dbus_resp_t;
```

当cpu发起一次请求时：

读请求：cache从`addr`中截取到`index`组号，根据组号找到数据存储的`meta`，读取meta中的数据判断是否命中（**always_comb驱动**）；若命中，将该缓存块对应的块首地址赋给一个变量`position`，该变量连接`data_ram`的读端口，读取到对应的字直接返回；若未命中则与memory通信；

写请求：cache从`addr`中截取到要写入的`index`组号与`offset`块内偏移，根据`meta`判断是否命中（判断的是块是否命中），若命中，块首地址赋值给`position`，由此计算出cache中的存储地址，设置strobe与data写入数据，同时修改`meta`；若缺失则于memory通信；

6、cache与memory通信

cache与memory通信通过`creq`发起对主存的访存请求，通过`cresp`得到主存响应的数据；通信的数据单元为块，但是是以字为单元进行数据突发传输，即需要多次传输完成一次事务transaction；期间请求的数据地址始终为块的**首**地址不能改变。请求与响应端口的信号：

```
// cbus_req_t
 typedef struct packed {
     logic    valid;     // 向主存发起的请求是否有效
     logic    is_write;  // 是否为一个写事务，写回
     msize_t  size;      // 一次的字节数（以块为事务，一个字一个字传）设置为8
     addr_t   addr;      // 写回的字的起始地址（同样一个字一个字写）始终保持为首地址，
不能改变
     strobe_t strobe;    // 写回哪些字节，对于缓存来说只有'0与'1两种情况
     word_t   data;      // 写回的数据（一个字），与当前写回的地址对对应
     mlen_t   len;       // 突发的数量（写回的字的个数，一个cache line的个数）
     axi_burst_type_t burst;
 } cbus_req_t;

// cbus_req_t
 typedef struct packed {
     logic  ready;  // 单个数据的握手信号
     logic  last;   // 整个事务握手
     word_t data;   // 主存响应的数据
 } cbus_resp_t;
```

当cache发起一次请求时：

读请求：读请求发生在`ALLOCATE`状态，cache应设置`creq`的相关信号，`valid`置为1，`is_write`置为0，`size`置为8，`addr`置为此次要读取的块首地址，需要tag、index拼接而成，不考虑块内位移，因为始终为块的首地址；`strobe`置为`'0`，表示不写入，`data`无效数据，`len`设置为`MLEN16`，表示一次突发传输16个字；所有上面的请求信息，在一次事务中必须保持不变；需要注意的是，读请求需要设

置写入cache line的参数，每次`ready`为1的周期，需要把`cresp.data`存入当前缓存的对应地址（由offset和position得到），随后更新offset（自增1，准备存放下一个字），直到突发传输完成`last`为高电平。

写请求：写请求发生在`WRITEBACK`状态，cache应设置`creq`的相关信号，`valid`置为1，`is_write`置为0，`size`置为8，`addr`置为块的首地址，`strobe`置为`'1`，表示改字全部写入，`data`为当前字地址要写入的数据，`len`设置为`MLEN16`，表示突发传输16个字，除了`data`外，其他信息在一次事务中必须保持不变；需要注意的是，写请求需要设置读取cache line的参数，每次`ready`为1的周期，需要把`creq.data`向后更新一个字，即offset + 1，（position保持不变），读到下一个字设置到对内存的写请求中，直到突发传输完成。

7、参数

缓存实现中需要用到的参数：

（1）与处理器请求响应：`dreq`与`dresp`，其中`dreq`为输入无需定义，`dresp`为输出端口，需要定义和设置；

（2）与内存请求响应：`creq`与`cresp`，其中`cresp`为输入无需定义，`dreq`为输出端口，需要定义和设置；

（3）状态机相关：`state`与`state_nxt`，分别表示当前周期状态与下一周期状态；

（4）缓存访存相关：

`data_ram`，分别表示对数据存储器访存的接口信号；

`meta_write`，写入meta的信息与

`tag`、`index`与`offset`表示处理器请求地址的三个定位信号；

`meta`，表示当前操作的cache line的meta信息；

`position`，表示当前操作的data块首地址，可能是读也可能是写；

`cache_sets`：记录一个组中每个块的元信息与`data`块首地址；（内部有一个`meta_t[ASSOCIATIVITY]`与`addr[ASSOCIATIVITY]`）

四、LRU替换算法

cache为四路组相联，cache line替换策略使用LRU算法，当组内满时将最久没有被访问的cache块替换掉（把该块的组内编号赋值给position，不需要考虑组的问题，会被`dreq`中address替换掉的块组号肯定和`index`一致，用index索引组，用position索引块进行替换即可）

1、LRU替换算法基本原理

什么是组内最近最少使用，一个示例：

> 假设，一个待访问的序列为 4 3 4 2 3 1 4 2，物理块有3个，考虑这三个物理块的调度情况

- 第一轮： 4调入内存，此时物理块内容为：4；
- 第二轮： 3调入内存，此时物理块内容为：3，4；
- 第三轮： 4调入内存，此时物理块不需要从外部调入，而是修改内部顺序为：4，3；
- 第四轮： 2调入内存，此时物理块内容为：2，4，3；
- 第五轮： 3调入内存，此时物理块不需要从外部调入，而是修改内部顺序为：3，2，4；
- 第六轮： 1调入内存，此时物理块内容为：1，3，2 （内存已满，而当前最少使用的是4，所以丢弃4）；

- 第七轮： 4调入内存，此时物理块内容为：4，1，3（内存已满，而当前最少使用的是2，所以丢弃2）；
- 第八轮： 2调入内存， 此时物理块内容为：2，4，1（内存已满，而当前最少使用的是3，所以丢弃3）

可以发现，抽象为一个数组后，组内最近最少使用的就是数组最后一个元素；如果新存入或者访问一个值，则将这个值放在数组开头。如果存储容量超过上限，那么删除队尾元素，再存入新的值。

对应于cache的行为：抽象出一个数组，如果组内某一个块hit（只考虑`COMPARE`状态即可，写回后一定会进行访问，不需要为写回多设置一次调度），就将该块的组内编号设为0，其余右移；如果超出cache set的大小（关联度`ASSOCIATIVITY`）则确定替换cache line为数组末尾的编号（赋值position进行写回操作或直接进行分配）

2、LRU算法电路实现

法一：**每个cache set定义一个组内数组**`addr_t [ASSOCIATIVITY - 1 : 0] used_line`，`used_line`**数组大小为关联度**`ASSOCIATIVITY`，从0开始到`ASSOCIATIVITY - 1`分别代表最近最不常访问的cache line，初始化为cache line下标逆序；miss时直接确定`used_line[ASSOCIATIVITY - 1]`为替换的下标即可，因为初始化为逆序，所以在块未满时会按正序逐个写入。hit时将hit的组内编号设置为数组头，其余右移（从右到左循环赋值）

法二：**每个cache set定义一个组内数组**`addr_t [ASSOCIATIVITY - 1 : 0] used_line`，数组大小为关联度`ASSOCIATIVITY`，每个元素下标代表组内编号为下标的块，**数据为该块的**`hit`**情况，从0到ASSOCIATIVITY - 1**，该信号可以直接由assign驱动，并且只会在`COMPARE`状态并且hit条件下修改，修改逻辑如下：循环assign，

- 如果`i == position`，那么把`used_line[i]`置为0，表示最近一次访问；
- 如果`used_line[i] >= used_line[position_nxt]`，那么`assign used_line[i] = used_line[i]`；
- 如果`used_line[i] < used_line[position_nxt]`，那么`assign used_line[i] = used_line[i] + 1`；

通过这样的逻辑来控制`used_line`的访问情况始终保持在最近访问为`0`，最远访问为`ASSOCIAVITY - 1`；

初始化`valid == 0`的情况下，`assign used_line = ASSOCIAVITY - 1`；代码如下：

```
// 驱动used_line的组合逻辑
assign sets[i].used_line[j] = (sets[i].metas[j].valid == 0) ? '1 :
    (state != COMPARE) ? sets[i].used_line[j] :
        (i == position) ? '0 : (sets[i].used_line[j] >= used) ?
            sets[i].used_line[j] : sets[i].used_line[j] + 1;
```

组合逻辑会导致电路逻辑循环，因此应该添加时序控制，添加后的used_line时序逻辑为：

```
// 驱动used_line
always_ff@(posedge clk) begin
    if(sets[i].metas[j].valid == 0) begin
        sets[i].used_line[j] <= '1;
    end
    else if(state_nxt != COMPARE) begin
        sets[i].used_line[j] <= sets[i].used_line[j];
    end
    else if(i == position) begin
        sets[i].used_line[j] <= '0;
```

```
    end
    else if(sets[i].used_line[j] >= sets[index].used_line[position]) begin
        sets[i].used_line[j] <= sets[i].used_line[j];
    end
    else if(sets[i].used_line[j] < sets[index].used_line[position])begin
        sets[i].used_line[j] <= sets[i].used_line[j] + 1;
    end
    else begin
        sets[i].used_line[j] <= '1;
    end
end
```

五、流水线调整

1、支持乘多周期除法器

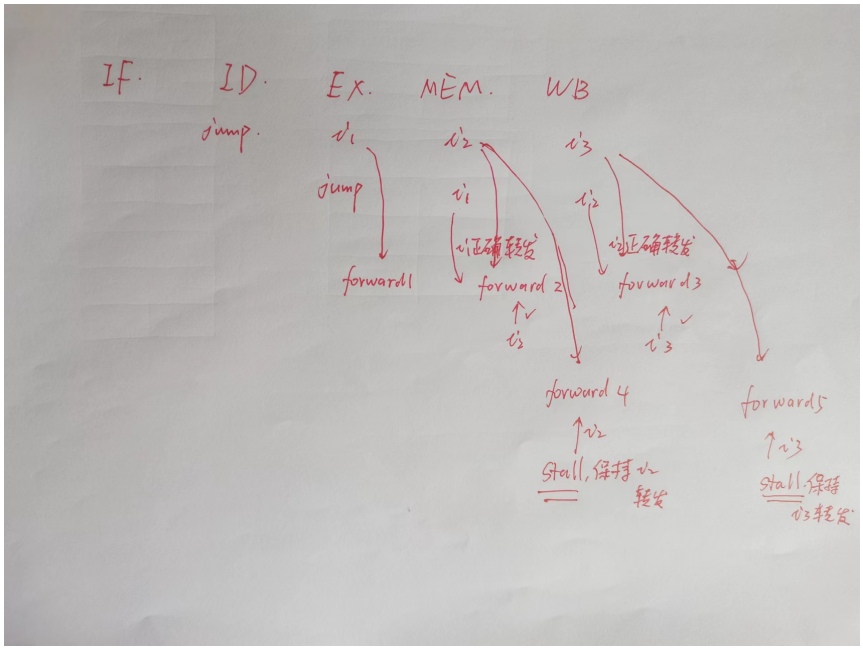
为了支持乘除法器的多周期运算，给`alu`和`execute`添加时钟与握手信号；当`execute_data_ok`为低电平时，说明`execute`在该周期未计算完成，此时需要阻塞`execute`以及之前的流水段，`memory`与`writeback`流水段继续流动并插入气泡，直到某个周期中`execute_data_ok`信号为高电平，恢复流动。

存在一种情况：`execute`计算乘除法，此时需要用到`memory`转发的数据，但`memory`为访存指令存在延迟，即`multiplier`和`divider`拿到的操作数据有可能是不对的，需要在拿到正确的数据后进行重新计算，解决方案为在满足上述条件时（具体到信号为`hazardOut.clear == 1`），将`multiplier`与`divider`的`reset`信号拉高，直到某一周期`memory`得到正确的结果`state_nxt`设置为`DOING`，下一周期开始计算。

2、转发器问题

`execute`阻塞，`memory`与`writeback`继续流动，此时会导致转发器数据覆盖，即memory会把writeback转发的数据覆盖掉（lab2时已经改进转发器解决无效数据转发的问题），若此时`execute`阻塞阶段需要用到writeback转发的数据就会导致执行错误。若通过`memory`阻塞的方式防止覆盖，在特定条件下会导致`ireq`和`dreq`循环访存一直阻塞，无法正常执行。

解决方案：添加两个备份转发器，当`execute`阻塞而`memory`不阻塞时，阻塞两个备份转发器，五个转发器进行转发；如下图：



通过两个备份转发器，在一般情况下，`forward4`与`forward5`转发器与`forward2`和`forward3`转发器的数据同步，在特殊条件下阻塞：

- `execute`阻塞而`memory`不阻塞时：`forward4`与`forward5`阻塞，1~3正常运行，备份execute之前的几条指令的转发数据，防止数据覆盖导致的数据源丢失；
- `memory`阻塞时，`forward5`阻塞保持`i3`转发的数据，`forward3`正常转发接收`i2`转发的数据；

六、错误记录

（1）驱动used_line

```
else if(state == IDLE || state_nxt != COMPARE) begin
  sets[i].used_line[j] <= sets[i].used_line[j];
end
```

（2）驱动ram访存信号

```
ALLOCATE: begin
    ram.en = 1;
    ram.strobe = 8'b11111111;
    ram.wdata  = cresp.data;
    ram.addr = sets[index].addrs[position] + {4'b0, offset};
    // 这个offset应该是ram_offset控制与内存交互式的访存
end
```

（3）compare阶段的写回

```
COMPARE: begin
    ram.en = 1;
    ram.strobe = hit ? dreq.strobe : '0;        // 不能直接写入，有可能没有hit
    ram.wdata  = dreq.data;
    ram.addr = sets[index].addrs[position] + {4'b0, offset};
end
```

在`COMPARE`阶段要把使能置为1，为了读取，但同时，为了防止未hit的情况下修改ram中的内容，只有在hit情况下才设置`strobe = dreq.strobe`，否则把`strobe`置为0，防止污染`ram`的数据；

（4）添加乘除法器后的取指问题

```
else if (~fetch_delay && ~memory_delay && ~jump_delay && ~execute_data_ok) begin
    // 这里应该是execute_data_ok，而不用取反，当没有计算完成时（execute_data_ok = 0）时阻塞，为1时更新pc
    pc <= pcnext;
end
```

（5）32位运算的问题

因为`mulw``divw`等指令是对32位数据进行乘除运算并且是有符号数，所以像原来那样采用64位低位运算的方式会影响结果，需要设置32位乘除法器，因此将乘除法器的`word_t`进行参数化，例化多个乘除法器对结果进行选择：

```
// 将WORD_BITS参数化，以divider为例，multiplier同理
module divider
    import common::*;
    import pipes::*; #(
        parameter WORD_BITS = 64
    ) (
    input logic clk, resetn, valid,
    input divider_op_t op,
    input logic unsign,
    input logic[WORD_BITS - 1 : 0] a, b,
    output logic done,
    output logic[WORD_BITS - 1 : 0] c // c = {a % b, a / b}
);
```

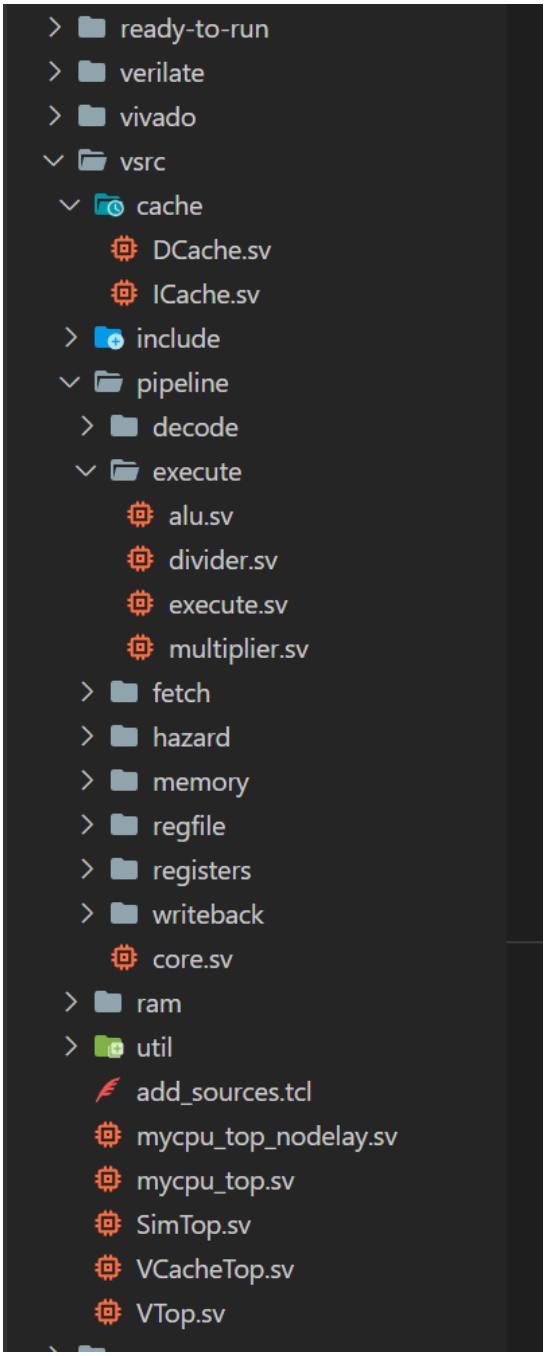
（6）srli的译码问题

`srl`与`srli`指令的高位`func`位数不一致，`srl`为7位信号，而`srli`为6位信号，在译码的时候对`srli`进行`F7`的比较会导致译码错误运算结果出错；

```
// 对srli译码，用6位func
F3_SRL_SRA_DIVU : begin
    if(func6 == F6_SRL) begin
        ctl.op = SRLI;
        ctl.func = ALU_SHIFTR;
    end
    else begin
        ctl.op = SRAI;
        ctl.func = ALU_SHIFTRS;
    end
end
// 对srl译码，用7位func
F3_SRL_SRA_DIVU : begin
    if(func7 == F7_SRL) begin
        ctl.op = SRL;
        ctl.func = ALU_SHIFTR;
    end
    else if(func7 == F7_SRA)begin
        ctl.op = SRA;
        ctl.func = ALU_SHIFTRS;
    end
    else if(func7 == F7_MUL_DIV) begin
        ctl.op = DIVU;
        ctl.func = ALU_DIVU;
    end
end
```

七、最终实现

目录结构



在原来的基础上添加了`DCache`与`ICache`以及乘除法器

verilator测试

```
[WARNING] difftest store queue overflow
Random test passed.
Single test passed.
Run paint
Aptenodytes patagonicus
Picture generated.
Compressed size=2162
Done! Result:

PgBiVkQAppeDAMu8qQDO4c4A9ebTAAYAAmsQuwjMAmIEIgJhIxECYBcAAAAGAAJrErsGzAJiBCICYSIRAmYYAAAABgACaxO7BcwCYgQiAmEiEQJgGAAAAAYAAmoHqg67A8wCZgQiAmEiEQJgGAAAAA
YAAmoLqgu7A8wCYgMiAmEhEQJgGQAAAAYAAmoNqgq7AswCYgMiAmEhEQJgGQAAAAYAAmoPqgm7AsYFIgJhHxECYBoAAAAHAAJqD6oJuwLGBCICYR8RAmAaAAAABwACahGqCLsCYgMiAmEeEQJmGwAA
AAcAAmkGmQyqB7sCYgMiAmEeEQJgGwAAAAcAAmkJmQqqB7sCYgMiAmEdEQJgGwAAAAcAAmkLmQmqBrsCYgMiAmEcEQJgHAAAAAcAAmkMmQiqBrsCYgMiAmYcEQJgHAAAAAcAAmkNmQiqBrsCYgMiAm
EaEQJmHQAAAAcAAmkOmQiqBrsCYgIiAmYaEQJgHQAAAAcAAmgGiAmZB6oGuwJiAyICYRgRAmAeAAAABwACaAiICJkHqga7AmIDIgJhFxECYB4AAAAHAAJoCYgImQaqBrsCxgMiAmEWEQJmHwAAAAcA
AmgKiAeZB6oGuwJiAyICYRURAmAfAAAABwACaAuIB5kGqga7AsYEIgJhFBECYB8AAAAIAAJoC4gGmQaqBrsCzAJiAiICZhMRAmAgAAAACAACZwR3B4gGmQaqBrsDzAJiAiICZhIRAmAgAAAACAACZw
V3B4gGmQaqBbsEzAJiAiICZhARAmYhAAAACAACZwZ3BogGmQaqBrsEzAJiAiICZg8RAmAhAAACAACZwZ3BogGmQaqBrsFzAJiAyICZg0RAmAhAAAABwACZgd3BogGmQaqBrsFzALWAmICIgJmCxEC
YCIAAAAHAAJmB3cGiAaZBqoGuwXMA90CZgIiAmYKEQJgIgAAAAgAAmcGdwaIBpkGqga7BcwE3QNmAgADZgYRAmAjAAAACAACZwZ3BogGmQaqBrsFzAXdAuYCZgIAA2YEEQJgIwAAAAgAAmcGdwaIBp
kGqga7BcwF3QPuA2YDAAVmJAAAAAgAAmYFdweIBpkGqgW7BswF3QXuAvYGZiUAAAAJAAJnA3cHiAaZBqoGuwbMBd0F7gT/BGYlAAAACQACaAqIBpkGqga7BswF3QXuA/8CZgJEAmAkAAAACQACaAmI
B5kGqga7BcwG3QXuAvYCZANEAmAkAAAACgACaAeIB5kHqga7BcwG3QTuAmYGRAJgJAAAAAoAAmgGiAiZBqoGuwbMBd0E7gJmB0QCYCQAAAAKAAJmBYgImQeqBrsGzAXdAuYCZglEAmAkAAAACwACaA
KICZkHqge7BcwG3QJkC0QCYCQAAAALAAJpCpkIqga7BswE3QJmDUQCYCQAAAAMAAJpCJkIqge7BswC3QJmAjYNRAJgJAAAAAwAAmkHmQiqB7sGzALdAmMCMwJkDEQCYCQAAAAMAAJmBpkJqge7BswC
ZgQzAmQMRAJgJAAAA0AAmkDmQqqB7sEzANmBjMCZAxEAmAkAAADQACZgyqB7sEzAJjCDMCZAxEAmAkAAADgACagqqCLsCzAJmCjMCZAxEAmAkAAADgACZgiqCbsCxgNmCjMCZgtEAmAlAAAADw
ACagaqCbsCxgIEAmMLMwJkCkQCYCUAAAAQAAJqA6oJuwNmAkQCZgwzAmQKRAJgJQAAABAAAmYCqwi7AmYERAJmDTMCZApEAmAlAAAAEQACawi7AmYFRAJjDTMCZApEAmAlAAAAEgACawW7AmYGRAJm
DjMCZApEAmAlAAAAEgACZgO7AmYIRAJjDjMCZApEAmAlAAAAEwACawK2AmQIRAJjDjMCZApEAmAlAAAAEwADZgpEAmYOMwJmC0QCYCUAAAAUAAJkCkQCYw4zAmQLRAJgJQAAABQAAmYKRANmDDMCZg
xEAmAlAAAAFQACZAxEBWYFMwNmDUQCYCUAAAAWAAJkEEQGZg9EAmAlAAAAFwACZCREAmAlAAAAGAACZCNEAmAlAAAAGAACZiJEAmYmAAAAGQACZiFEA2Y1AAAAGgACZiBEA2UlAAAAHAACZB5EA2UC
YCMAAAAdAAJmHUQCZQJWAmAiAAAAHgACZhxEAmUCVQJgIgAAACAAAmYbRAJlAlYCYCEAAAAhAANmGUQCZQJVAmAhAAAAIwADZhdEAmUCVQJmIQAAACUABGYURAJmA1UCYCAAAAAoAAVmEUQCZQJVAm
YgAAAALQAHZgtEAmUCVQJgHwAAADMAAmYKRAJmAlUCZh8AAAA0AAJmCkQCZQJVAmYeAAAANQADZghEAmYDVQJgHQAAADcAAmYIRAJmAlUCZh0AAAA4AANmB0QCZgJVAmAcAAAAOgACZghEA2UCYBsA
AAA7AAJmCEQDZhwAAAA8AANmB0QDZhsAAAA+AAJmCEQCZhoAAAA/AAJmCEQCYBkAAABAAANmBkQCZhkAAABCAAJmBkQCYBgAAABDAANmBEQCZhgAAABFAAJmBEQCYBcAAABGAAJmA0QCZhcAAABHAA
NmAkQCYBYAAABJAAJmAkYCYBUAAABKAAVmFQAAAEwAA2YVAAAABQADZgIGAmYCBgJmAwACYDgAA2YUAAAABQACYAIAAmAEAAJgAgACYDkAAmYUAAAABQADZgIGAmYCBgJmAgYCZlAAAAAHAAJgAgAK
YE8AAAAFAANmAgYCZgIGAmYFBk8AAABkAAAAZAAAAGQAAABkAAAAAE=
Run coremark
Running CoreMark for 10 iterations
2K performance run parameters for coremark.
CoreMark Size    : 666
Total time (ms)  : 689
Iterations       : 10
Compiler version : GCC9.4.0
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0xfcaf
Finised in 689 ms.
==================================================
CoreMark Iterations/Sec 14
Run dhrystone
Dhrystone Benchmark, Version C, Version 2.2
Trying 10000 runs through Dhrystone.
Finished in 1619 ms
------------------------------------------------
Dhrystone PASS        10 Marks
                vs. 100000 Marks (i7-7700K @ 4.20GHz)
Run stream
```

```
                vs. 100000 Marks (i7-7700K @ 4.20GHz)
Run stream
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 2048 (elements), Offset = 0 (elements)
Memory per array = 0.0 MiB (= 0.0 GiB).
Total memory required = 0.0 MiB (= 0.0 GiB).
Each kernel will be executed 10 times.
 The *best* time for each kernel (excluding the first iteration)
 will be used to compute the reported bandwidth.
-------------------------------------------------------------
* checktick: start=1.651739
* checktick: start=1.653423
* checktick: start=1.655070
* checktick: start=1.656827
* checktick: start=1.658511
* checktick: start=1.660224
* checktick: start=1.661934
* checktick: start=1.663589
* checktick: start=1.665365
* checktick: start=1.667163
* checktick: start=1.668996
* checktick: start=1.670813
* checktick: start=1.672593
* checktick: start=1.674368
* checktick: start=1.676092
* checktick: start=1.677791
* checktick: start=1.679516
* checktick: start=1.681260
* checktick: start=1.682961
* checktick: start=1.684699
Your clock granularity/precision appears to be 110 microseconds.
Each test below will take on the order of 16104 microseconds.
   (= 146 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:               8.4     0.004226     0.003920     0.004416
Scale:              0.6     0.055487     0.055231     0.055731
Add:                2.1     0.024179     0.022979     0.025670
Triad:              0.7     0.072372     0.071304     0.074065
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
Run conwaygame
```

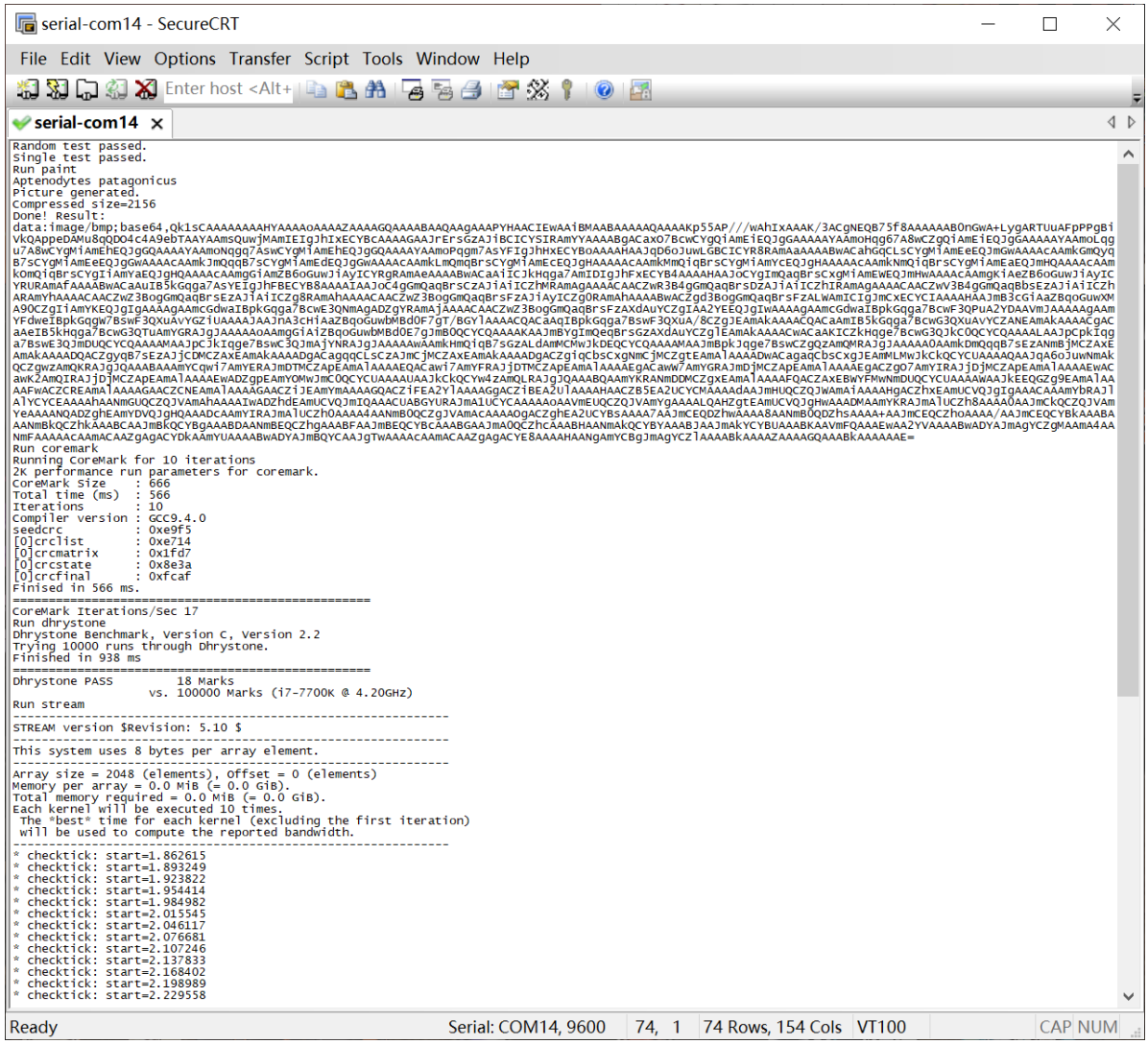
```
Solution Validates: avg error less than 1.000000e-13 on all three arrays
----------------------------------------------------------------
Run conwaygame
Play Conway's life game for 200 rounds.
seed=3663


     *
    * *
    *  *        ***   ***
     **
              *    * *    *
              *    * *    *
              *    * *    *       **
                ***   ***         **
[src/cpu/cpu-exec.c,320,cpu_exec] nemu: HIT GOOD TRAP at pc = 0x0000000080014e44
[src/cpu/cpu-exec.c,321,cpu_exec] trap code:0
[src/cpu/cpu-exec.c,62,monitor_statistic] host time spent = 23361258 us
[src/cpu/cpu-exec.c,64,monitor_statistic] total guest instructions = 34605237
[src/cpu/cpu-exec.c,65,monitor_statistic] simulation frequency = 1481308 instr/s
sh: 1: spike-dasm: not found
```

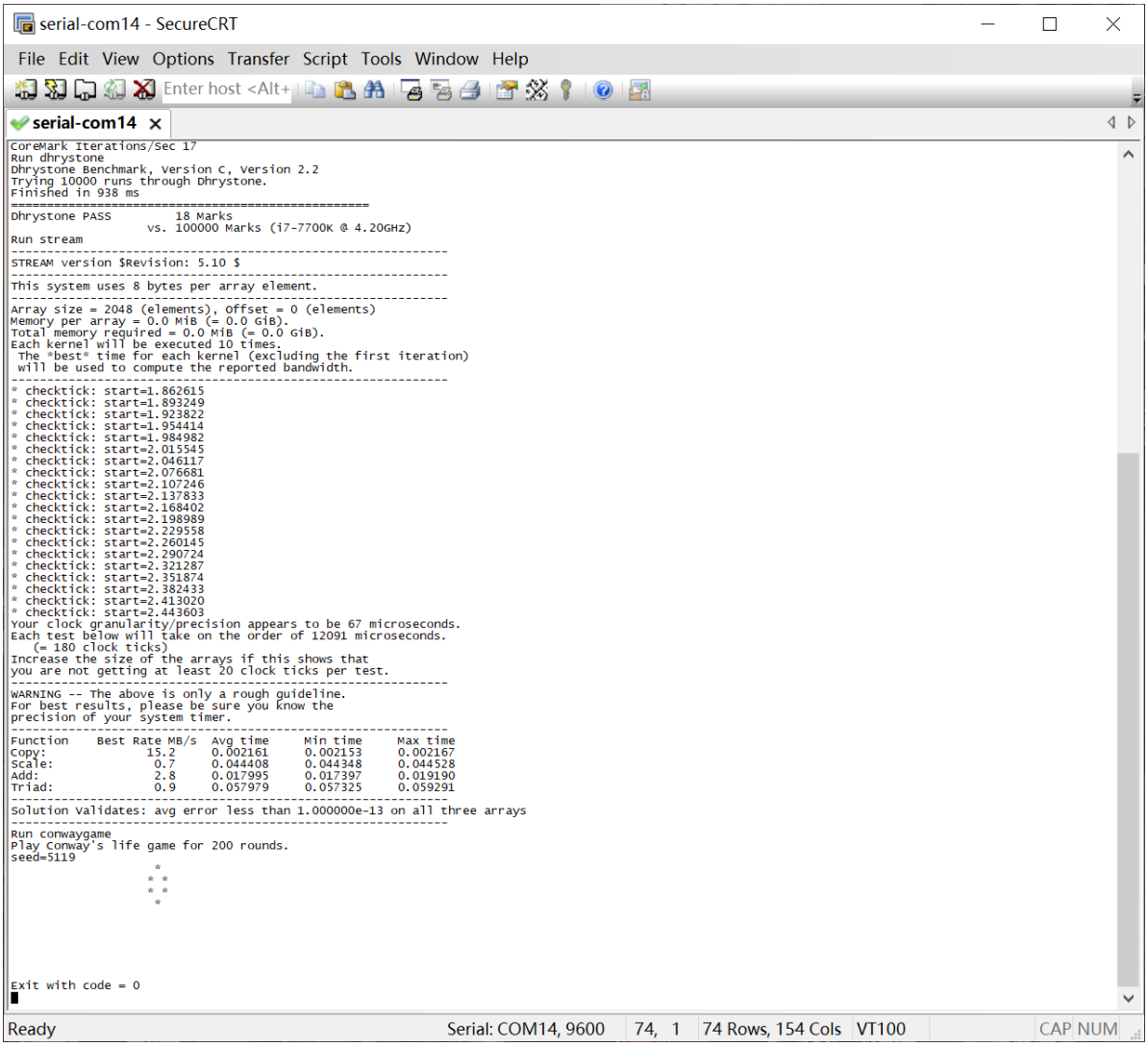
在verilator仿真测试的环境下，`TEST=paint`用时2294，如下图：

`TEST=coremark`测试结果为：`Interations/Sec 14`；`TEST=dhrystone`测试结果为：`10 Marks`；`TEST=stream`的测试结果为`Copy Best Rate = 8.4MB/s`；`TEST=conwaygame`测试结果如上。

上板串口软件测试结果


serial-com14 - SecureCRT
File Edit View Options Transfer Script Tools Window Help
Enter host <Alt+
serial-com14
Random test passed.
Single test passed.
Run paint
Aptenodytes patagonicus
Picture generated.
Compressed size=2156
Done! Result:

Run coremark
Running CoreMark for 10 iterations
2K performance run parameters for coremark.
CoreMark Size : 666
Total time (ms) : 566
Iterations : 10
Compiler version : GCC9.4.0
seedcrc : 0xe9f5
[0]crclist : 0xe714
[0]crcmatrix : 0x1fd7
[0]crcstate : 0x8e3a
[0]crcfinal : 0xfcaf
Finised in 566 ms.
CoreMark Iterations/Sec 17
Run dhrystone
Dhrystone Benchmark, Version C, Version 2.2
Trying 10000 runs through Dhrystone.
Finished in 938 ms
Dhrystone PASS 18 Marks
vs. 100000 Marks (i7-7700K @ 4.20GHz)
Run stream
STREAM version $Revision: 5.10 $
This system uses 8 bytes per array element.
Array size = 2048 (elements), Offset = 0 (elements)
Memory per array = 0.0 MiB (= 0.0 GiB).
Total memory required = 0.0 MiB (= 0.0 GiB).
Each kernel will be executed 10 times.
The *best* time for each kernel (excluding the first iteration)
will be used to compute the reported bandwidth.
* checktick: start=1.862615
* checktick: start=1.893249
* checktick: start=1.923822
* checktick: start=1.954414
* checktick: start=1.984982
* checktick: start=2.015545
* checktick: start=2.046117
* checktick: start=2.076681
* checktick: start=2.107246
* checktick: start=2.137833
* checktick: start=2.168402
* checktick: start=2.198989
* checktick: start=2.229558
Ready
Serial: COM14, 9600
74, 1
74 Rows, 154 Cols
VT100
CAP NUM



在上板测试的环境下，`TEST=paint`用时1307，如下图：

`TEST=coremark`测试结果为：`Interations/Sec 17`；`TEST=dhrystone`测试结果为：`18 Marks`；`TEST=stream`的测试结果为`Copy Best Rate = 15.2MB/s`；`TEST=conwaygame`测试结果如上。