Lab 4 - CSR

初步探索与构思

首先考虑 CSR 无论说得再天花乱坠，我们也得承认它是寄存器，因此理论上这玩意就应该跟普通寄存器坐一桌，吗？

在存储实现上其实有相当大的不同。

第一个问题：CSR 的地址空间是 32 位的，还是 64 位的？

要是这么想，那就说明我们这群年轻人还是图样图森破，sometimes Naive！Neither，CSR 其实是 12 位的，也就是说完整版的 CSR 应该有 4096 个寄存器（但实际上不需要也不正确，考虑到存在 `mstatus` 和 `sstatus` 这样换皮的寄存器）。还是要提高自己的知识水平！

真的可以用 `u64 csr[4096]` 来实现 CSR 吗？当然不可以！我跟开发板谈笑风生，结果开发板跑得比谁都快（根本无法实现嘛！）。要真的这么干我们得存多少 Wire，显然对于 CSR 来说是非常不划算的。

考虑：我们需要实现的寄存器其实只有二十多个，因此一个 5 位的地址就足够了。也就是说我们只需要实现 32 个寄存器就可以了。现在我们需要制作一个内部译码器，以将完整的 12 位 CSR 地址映射到我们内部定义的这 32 个寄存器中（注意对于 `mstatus` 和 `sstatus` 这样的寄存器，我们需要将它们的值映射到相同的寄存器中）。

```
always_comb begin
    case(target)
        CSR_MSTATUS: mapped_target = 0;
        CSR_MHARTID: mapped_target = 4;
        CSR_MIE: mapped_target = 1;
        CSR_MIP: mapped_target = 2;
```

注意到有些寄存器的一些位不可以写，那就增加一个 `mask` 输出，注意到这个 `mask` 必须用内部译码之前的地址计算，因为 `mstatus` 和 `sstatus` 能写的位就是不一样的。

```
    case(target)
        CSR_MSTATUS: mask = MSTATUS_MASK;
        CSR_SSTATUS: mask = SSTATUS_MASK;
        CSR_MIP: mask = MIP_MASK;
        CSR_MTVEC: mask = MTVEC_MASK;
        CSR_MEDELEG: mask = MEDELEG_MASK;
        CSR_MIDELEG: mask = MIDELEG_MASK;
        default: mask = ~64'h0;
    endcase
```

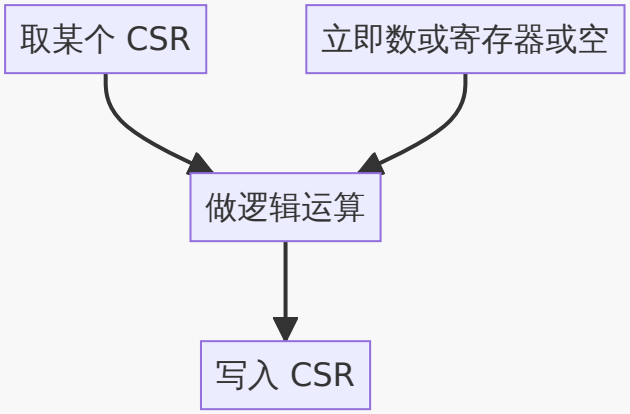
然后像正常的寄存器一样实现接口

```
module csr import common::*; import csr_pkg::*;(
    input logic clk,rst,
    input u12 read_target,
    input logic wdEn,
    input u12 write_target,
    input u64 write_data,
    output u64 read_data,
    output u64 csrs[31:0]
);
```

接线与指令实现

现在，我们考虑涉及 CSR 的指令。

实际上，我们写 CSR 的指令可以统一认为是：



由此，我们可以得出在 execute 中我们需要实现的逻辑：

```
assign CSR_write_value =
                        moduleIn.csr_op==CSRRW ? moduleIn.rs1:
                        moduleIn.csr_op==CSRRS ? moduleIn.CSR_value |
moduleIn.rs1:
                        moduleIn.csr_op==CSRRC ? moduleIn.CSR_value &
~moduleIn.rs1:
                        moduleIn.csr_op==CSRRWI ? moduleIn.imm:
                        moduleIn.csr_op==CSRRSI ? moduleIn.CSR_value |
moduleIn.imm:
                        moduleIn.csr_op==CSRRCI ? moduleIn.CSR_value &
~moduleIn.imm:
                        0;
```

好，现在参照常规寄存器的实现，我们在 Decode 阶段取出 CSR 的值，并在 Writeback 阶段写回。仿照常规寄存器的接线处理即可。

对于 decoder 模块，增加：

```
output u12 CSR_addr,
input u64 CSR_value,
```

对于 writeback 模块，增加：

```
if(moduleIn.isCSRWrite & moduleIn.valid) begin
    CSR_wbEn <= 1;
    CSR_addr <= moduleIn.CSR_addr;
    CSR_value <= moduleIn.CSR_write_value;
end else begin
    CSR_wbEn <= 0;
end
```

需要注意的是我们的 csr regfile 由于同时实现读写的译码，所以需要两个译码器模块。

Hazard 的思考

显然，在 Lab 1 的时候我们就已经制作过常规寄存器的转发了，本质上是 execute 阶段获取到的即将写入寄存器的值赶快转发给 decode 阶段的寄存器读取。

对于 CSR，我们能不能也钦点让转发来处理可能出现的 Hazard 呢？

理论上可行，并且如果你真的这么做，你会发现也的确可以通过所有的测试用例。

但是实际上不行！为什么？在下述“思考题”中回答。

我们还是选择进行清空流水线的操作。因此我们认为每个 CSR 写实际上自带了一个 Jump 到 PC+4（也就是 Memory 一旦正在处理 CSR 写入操作时，就应该让 Execute、Decode、Fetch 阶段的流水线的指令作废，Fetch 重新取下一条指令，其实这时已经在 Execute 阶段的指令）。

可以直接用 Jump 逻辑处理这个过程。

```
assign JumpEn = (moduleIn.isJump|
(moduleIn.isBranch&moduleIn.flagResult)|moduleIn.isCSRWrite) &
moduleIn.valid;
assign JumpAddr = moduleIn.isCSRWrite? moduleIn.pcPlus4 :
{moduleIn.aluOut[63:1],1'b0};
```

细节处理

连接 Difftest

既然连接的寄存器是确定的，那就不需要使用解码器了，我们直接指定好对应的寄存器，类似

```
.mepc               (csrs[6]),
```

mcycle

`mcycle` 寄存器的实现需要注意的是，它是一个 64 位的寄存器，表示 CPU 执行的时钟周期数。我们可以在时钟上升沿时对其进行自增操作。

```
if(wdEn) begin
    ...
    if(write_target!=CSR_MCYCLE)
        csrs[9] <= csrs[9] + 1;
end else
    csrs[9] <= csrs[9] + 1;
```

mstatus

注意到 mstatus 的最高位是只读的，在 ISA 中描述为：

> Location
>
> 31 when CSR[misa].MXL == 0
>
> 63 when CSR[misa].MXL == 1

> Description
>
> State Dirty.

> Read-only bit that summarizes whether either the FS, XS, or VS fields signal the presence of some dirty state.

也就是说我们需要在写入 mstatus 时，判断一下写入的 FS、XS 和 VS 字段是否为 3（脏状态），如果是，则将最高位设置为 1，否则设置为 0。

```
if(write_target==CSR_MSTATUS||write_target==CSR_SSTATUS) begin
                u64 temp_data = write_data & csr_write_mask;
                csrs[mapped_csr_addr_write][62:0] <= temp_data[62:0];
                csrs[mapped_csr_addr_write][63] <= temp_data[16:15]==3 |
                                                   temp_data[14:13]==3 |
                                                   temp_data[10:9]==3;
```

思考题

参考英文指令集手册，简述一下此次lab中各个csr寄存器的作用

https://riscv-software-src.github.io/riscv-unified-db/manual/html/landing/index.html

实现寄存器：mstatus mtvec mip mie mscratch mcause mtval mepc mcycle mhartid satp。这些寄存器均为64位宽。

1. **mstatus**（Machine Status Register）：
    - 用于保存和控制处理器的全局状态信息，如中断使能、特权模式切换等。
    - 包含多个字段，如MIE（Machine Interrupt Enable）、MPIE（Previous Machine Interrupt Enable）等。
2. **mtvec**（Machine Trap Vector Base Address Register）：
    - 保存异常和中断处理程序的入口地址。
    - 当发生异常或中断时，处理器会跳转到该寄存器指定的地址执行。
3. **mip**（Machine Interrupt Pending Register）：
    - 用于记录当前挂起的中断。
    - 包含多个位字段，如MEIP（Machine External Interrupt Pending）、MTIP（Machine Timer Interrupt Pending）等。
4. **mie**（Machine Interrupt Enable Register）：
    - 用于控制哪些中断可以被处理器响应。
    - 包含多个位字段，如MEIE（Machine External Interrupt Enable）、MTIE（Machine Timer Interrupt Enable）等。
5. **mscratch**（Machine Scratch Register）：
    - 通常用作临时存储寄存器，供操作系统或异常处理程序使用。
    - 在异常处理时，可以用来保存和恢复上下文。
6. **mcause**（Machine Cause Register）：
    - 用于记录导致异常或中断的原因。
    - 包含一个异常代码字段，指示具体的异常或中断类型。
7. **mtval**（Machine Trap Value Register）：
    - 用于保存与异常相关的附加信息，如非法指令的地址或访问错误的内存地址。
8. **mepc**（Machine Exception Program Counter）：
    - 保存发生异常或中断时的指令地址。
    - 在异常处理完成后，处理器可以从该地址恢复执行。
9. **mcycle**（Machine Cycle Counter）：
    - 用于记录处理器执行的时钟周期数。
    - 通常用于性能监控和调试。
10. **mhartid**（Machine Hardware Thread ID Register）：

    - 保存当前硬件线程的ID。
    - 在多核或多线程处理器中，用于区分不同的硬件线程。

11. **satp**（Supervisor Address Translation and Protection Register）：

    - 用于控制虚拟内存的地址转换和保护机制。
    - 包含页表基地址和地址转换模式（如Sv39、Sv48等）。

为什么一定要刷新流水线？

根据我的个人理解，给出一个最简洁的答案：CSR 的值不光是 Data Hazard 的问题，CSR 的值会直接影响后面（已进入流水线）指令的**运行状态（可能CSR一改，后面指令您就甭执行了）**。你必须确保下一条指令完全在新状态下执行。

这个问题的源头在于：CSR 控制的东西是**中断**。中断就要求：我不管你怎么实现这个处理器，但从外界来看，你的指令就是一条一条执行完的，那么你到了执行完一条指令的状态，如果我有外界干预，你就必须得假设这个干预带来的影响完全在这一条指令之后，并且对下一条指令来说一定是受到过影响之后**才开始执行的**。

因此 CSR 的值会影响到后续指令的执行状态。

举个简单的例子，假设之前 mstatus 中中断使能是关着的，那么你外界怎么发信号理论上不影响下面的指令执行。但是假设有这样两条指令：

```
csrrw t0, mstatus, t1
add a0, a1, a2
```

假设 csrrw 刚刚提交（认为已经运行完成了），add 正在 writeback 阶段，这时候突然来了个中断，这时候您 add 就甭执行了，您要是刚才已经写寄存器了那就坏菜了。

因此我们需要刷新流水线，确保后续指令的执行状态是正确的，也就是假装我们的流水线是有原子性的。

同理，其实修改 `mtvec`、`mepc` 这样的寄存器也是一样的道理。

你必须确保下一条指令完全在新状态下执行。

Lab 4 - Vivado 上板

按照步骤执行即可，注意我用的是 Vivado 2024.1，需要执行 Upgrade IP。

CPU 特性概览

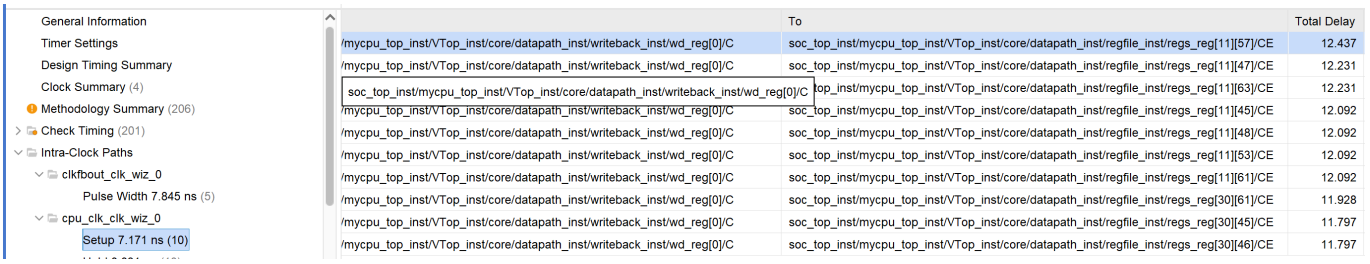
通过 Vivado 在 Impl 之后给出的报告可以大概知道目前这个 CPU 的特性：

- 目前的 CPU 频率是 25MHz，它可以稳定运行在这个频率下。

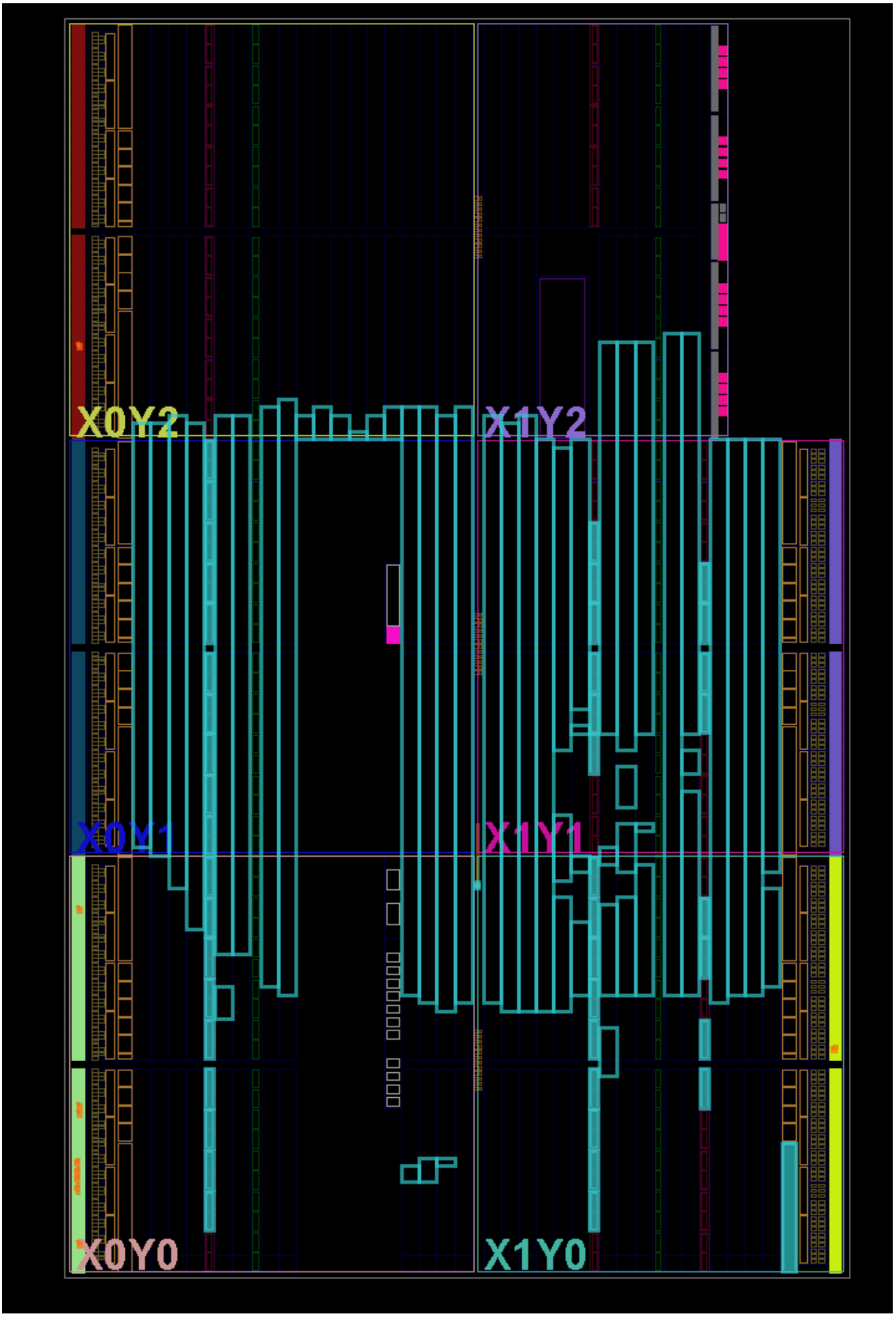
soc_top_inst/clk_wiz_0/inst/sys_clk	{0.000 5.000}	10.000	100.000
clkfbout_clk_wiz_0	{0.000 5.000}	10.000	100.000
cpu_clk_clk_wiz_0	{0.000 20.000}	40.000	25.000
sys_clk_pin	{0.000 5.000}	10.000	100.000

- 当前的 WNS 为 7.171 ns，可以认为这个 CPU 在什么都不改的情况下有能跑到 30 MHz 的潜力。瓶颈在于写回的 Mux。

Clock	Edges (WNS)	WNS (ns)	TNS (ns)	Failing Endpoints (TNS)	Total Endpoints (TNS)	Edges (WHS)	WHS (ns)	THS (ns)	Failing Endpoints (THS)	Total Endpoints (THS)	WPWS (ns)	TPWS (ns)	Failing Endpoints (TPWS)	Total Endpoints (TPWS)
soc_top_inst/clk_wiz_0/inst/sys_clk											3.000	0.000	0	1
clkfbout_clk_wiz_0											7.845	0.000	0	3
cpu_clk_clk_wiz_0	rise - fall	7.171	0.000	0	13267	rise - rise	0.031	0.000	0	13267	19.020	0.000	0	6178
sys_clk_pin	rise - rise	4.893	0.000	0	354	rise - rise	0.104	0.000	0	354	4.500	0.000	0	167



- 目前的 CPU 大约占用了 37.37% 的 LUTs 和 15% 的 FFs，说明这个 CPU 还是比较小的。


X0Y2
X1Y2
X0Y1
X1Y1
X0Y0
X1Y0

结果


COM5 - PuTTY
Button press detected!
BASYS3 GPIO/UART DEMO!
BASYS3 GPIO/UART DEMO!
BASYS3 GPIO/UART DEMO!
BASYS3 GPIO/UART DEMO!
BASYS3 GPIO/UART DEMO!
BASYS3 GPIO/UART DEMO!
BASYS3 GPIO/UART DEMO!
Random test passed.
Single test passed.
Run paint
Aptenodytes patagonicus
Picture generated.
Compressed size=2164
Done! Result:

MAABAAAAAAQAAAAKp55AP///wAhIxAAAK/3ACgNEQB75f8AAAAAAB0nGwA+LygARTUuAPpPPgBiVkQApp
eDAMu8qQDO4c4A9ebTAAYAAmsQuwjMAmIEIgJhIxECYBcAAAAGAAJrErsGzAJiBCICYSIRAmYYAAAABg
ACaxO7BcwCYqQiAmEiEQJgGAAAAAYAAmoHqg67A8wCZqQiAmEiEQJgGAAAAAYAAmoLqgu7A8wCYqMiAm
EhEQJgGQAAAAYAAmoNqgq7AswCYqMiAmEhEQJgGQAAAAYAAmoPqgm7AsYFIqJhHxECYBoAAAAAAJqD6
oJuwLGBCICYR8RAmAaAAABWAchAhGqCLsCYqMiAmEiEQJmGwAAAAcAAmkGmQyqB7sCYqMiAmEiEQJgGw
AAAAcAAmkJmQqqB7sCYqMiAmEdEQJgGwAAAAcAAmkLmQmqBrsCYqMiAmEcEQJgHAAAAAcAAmkMmQiqBr
sCYqMiAmYcEQJgHAAAAAcAAmkMmQiqBrsCYqIiAmYaEQJmHQAAAAcAAmkOmQiqBrsCYqIiAmYaEQJgHQ
AAAcAAmgSiAmZB6oGuwJiAyICYRqRAmAaAAABwCaAiICHHgqA7AmIIqJhFxECYB0AAAAIAAJoC4gGmQaqBrsCzAJiAiICZhMRAmAgAAAACAAC2wR3B4gGmQaqBrsDzAJiAi
ICZhIRAmAgAAAACAAC2wV3B4gGmQaqBbsEzAJiAiICZhARAmYhAAAACAAC2w73BogGmQaqBrsEzAJiAi
ICZg8RAmAhAAAACAAC2w3BogGmQaqBrsFzAJiAyICZgURAmAhAAABwACZgd3BogGmQaqBrsFzALWAm
ICIgJmCxECYCIAAAAHAAJmB3cGiAaZBgoGuwXMA9OCZgIiAmYKEQJgIqAAAAgAAmcGdwaIBpkGqga7Bc
wE3QNmAgADZgYRAmAjAAAACAAC2w33BogGmQaqBrsFzAXdAuYCZgIAA2YEEQJgIwAAAAgAAmcGdwaIBp
kGqga7BcwF3QPuA2YDAAVmJAAAAAgAAmYFdweIBpkGqgW7BswF3QXuAvYGZiUAAAAJAAJnA3cHiAaZBq
oGuwbMBd0F7gY/BGIlAAACQACaAqIB5kBqga7BcwG3QYuAmYGRAJqJAAAAoAAmqGlAiZBqoGuwbMBd
0E7gJmB0QCYCQAAAAKAAJmBYgImQeqBrsGzAXdAuYCZgIEAmAkAAACwACaAKIC2kHqge7BcwG3QJkC0
QCYCQAAAALAAJpCpkIqga7BswE3QJmDUQCYCQAAAAMAAJpCJkIqge7BswC3QJmAjYNRAJgJAAAAAwAAm
khmQigB7sGzALIAmMCMwJkDEQCYCQAAAAMAAJmBpkJqge7BswCZgGzAmQMRAJgJAAAAA0AAmkDmQqgB7
sFzANmBjMCZAxEAmAkAAUQAC5gyqB7sEzAJjCDMCZAxEAmAkAAAUgACagqgTLsCzAJmCjMCZAxEAm
AkAAAADgACZgiqCbsCxgNmCjMCZgtEAmAlAAAIDwACagaqCbsCxgJEAmMLMwJkCkQCYCUAAAAQAAJgA6
oJuwNmAkQCZgwzAmQKRAJgJQAAABAAAmYCqw17AmYERAJmDTMCZApEAmAlAAAAEQACaw17AmYFRAJjDT
MCZApEAmAlAAAEgACaww97AmYGRAJmDjMCZApEAmAlAAAEgACZgO7AmYIRAJjDjMCZApEAmAlAAAAEw
ACawKZAmQIRAJjDjMCZApEAmAlAAAABwADZgpEAmYCMwJmCOQCYCUAAAAIAAJkCkQCYw4zAmQLRAJgJQ
AAABQAAmYKRANmDDMCZqxEAmAlAAAAFQACZAxEBWYFMwNmDUQCYCUAAAAWAAJkEQGZg9EAmAlAAAAFw
ACZCREAmAlAAAAGAAC2CNEAmAlAAAAGAACZiJEAmYmAAAAOQAC2iFEA2Y1AAAAGgACZiBEA2U1AAAAHA
ACZB5EAZOCYCMAAAAdAAJmHUQC2QJWAmAlAAAAHgACZkxEAmUCVQJgIqAAACAAAmYbRAJlA1YCYCEAAA
AhAAMmCUQC2QJVAmAhAAAIwADZhSEAmUCVQJmIQAAACUABGYURAJmAlUCYCAAAAoAAVmEUQC2QJVAm
YgAAAAlQAHZqtEAmUCVQJgHwAAADMAAmYKRAJmAlUCZh8AAAAOAAJmCkQC2QJVAmYeAAAANQADZghEAm
YDVQJgHQAAADcAAmYIRAJmAlUCZh0AAAA4AANmB0QCZgJVAmAcAAAAOgACZghEA2UCYBsAAAA7AAJmCE
QDZhwAAAA8AANmB0QDZhsAAAA+AAJmCEQCZhoAAAA/AAJmCEQCYBkAAABAANmBkQCZhkAABCAAJmBk
QCYBgAAABAANmBkQC2hgAAABFAAJmBEQCYBcAAABCAAJmAQC2hcAAABHANmAkQCYBYAAAB3AAJmAk
YCYBUAAABKAAVmFQAAAEwAA2YVAAAABQADZgIGAmYDAAJgAgACYDqAA2YUAAAABwAGYAIAAmACAAJgOQ
ACZhQAAAAFANmAgYCZgMAAmACAAJgYwAAAAUACGACAAJgAgACYE8AAAAFAANmAgYCZgIGAmYDAAJgTw
AAAGQAAABkAAAAZAAAAGQAAAAAQ==
Run coremark


COM5 - PuTTY
Run coremark
Running CoreMark for 10 iterations
2K performance run parameters for coremark.
CoreMark Size   : 666
Total time (ms) : 11004
Iterations   : 10
Compiler version : GCC9.4.0
seedcrc   : 0xe9f5
[0]crclist   : 0xe714
[0]crcmatrix   : 0x1fd7
[0]crcstate   : 0x8e3a
[0]crcfinal   : 0xfcaf
Finised in 11004 ms.
CoreMark Iterations/Sec 0
Run dhrystone
Dhrystone Benchmark, Version C, Version 2.2
Trying 10000 runs through Dhrystone.
Finished in 15953 ms
Dhrystone PASS   1 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=17.072366
* checktick: start=17.110449
* checktick: start=17.148546
* checktick: start=17.186651
* checktick: start=17.224652
* checktick: start=17.262758
* checktick: start=17.300759
* checktick: start=17.338864
* checktick: start=17.376865
* checktick: start=17.414971
* checktick: start=17.452972
* checktick: start=17.491077
* checktick: start=17.529078
* checktick: start=17.567200
* checktick: start=17.605321
* checktick: start=17.643217
* checktick: start=17.681338
* checktick: start=17.719339
* checktick: start=17.757340
* checktick: start=17.795462
Your clock granularity/precision appears to be 517 microseconds.
Each test below will take on the order of 291214 microseconds.
   (= 563 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: 1.0   0.033454   0.033452   0.033463
Scale: 0.1   0.487877   0.482874   0.497889
Add: 0.1   0.410098   0.399093   0.432115
Triad: 0.1   0.795825   0.779818   0.827846
Solution Validates: avg error less than 1.000000e-13 on all three arrays
Run conwaygame
Play Conway's life game for 200 rounds.
seed=39020
Exit with code = 0