ysyx lab report

开始时间：2022.03.04 **实验平台**：Ubuntu 20.04 LTS

note：虽然把数字电路实验排版在了前面，但其实我是先做的PA，后做的数字电路基础实验。

1. 实验进度

2. Verilator & 数字电路基础实验

2.1 verilator 安装结果：

```
(base) ypwang@ypwangPC:~$ verilator --version
Verilator 4.210 2021-07-07 rev v4.210
(base) ypwang@ypwangPC:~$
```

双控开关仿真代码：

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include "obj_dir/Vswitch.h"
#include <verilated.h>

int main(int argc, char** argv, char** env) {
    Vswitch *top = new Vswitch;  //创建实例化仿真对象

    while (!Verilated::gotFinish()) {

        int a = rand() & 1;
        int b = rand() & 1;
        top->a = a;
        top->b = b;
        top->eval();
        printf("a = %d, b = %d, f = %d\n", a, b, top->f);
        assert(top->f == a ^ b);
    }
        top -> final();
        delete top;
        exit(EXIT_SUCCESS);
}

```

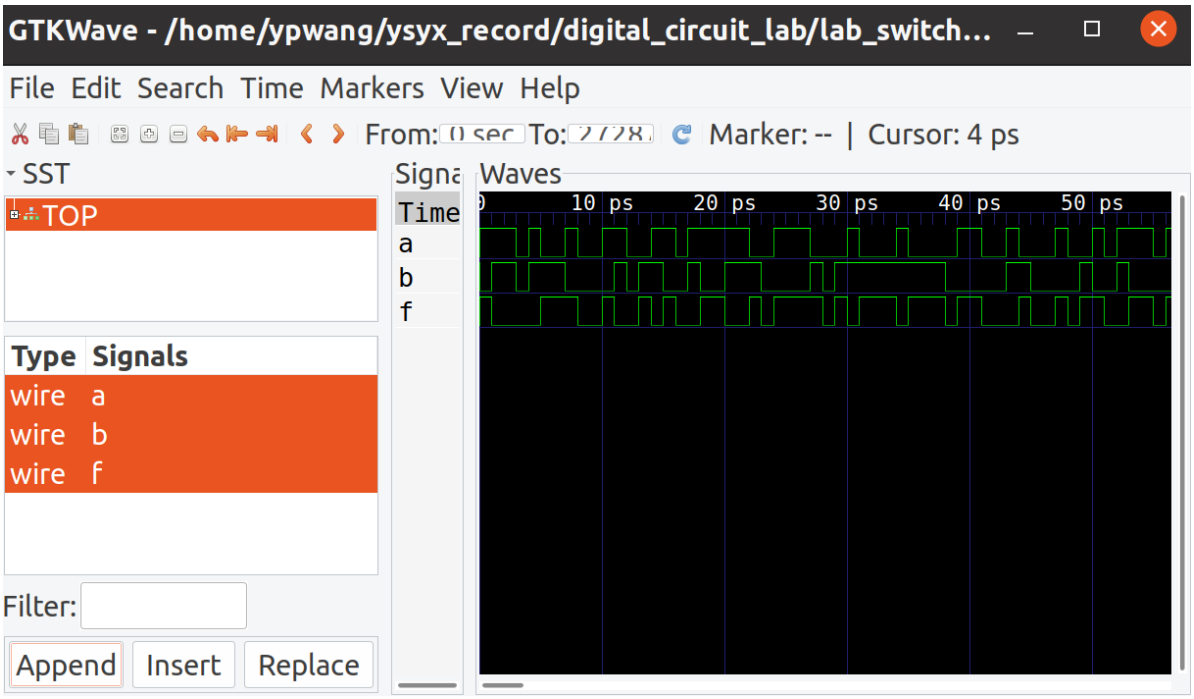
键入`Ctrl+Z`结束仿真

```
a = 1, b = 1, f = 0
a = 0, b = 1, f = 1
a = 1, b = 1, f = 0
a = 1, b = 1, f = 0
a = 1, b = 0, f = 1
a = 0, b = 0, f = 0
a = 0, b = 1, f = 1
^Z
[1]+  Stopped                 ./obj_dir/Vswitch
```

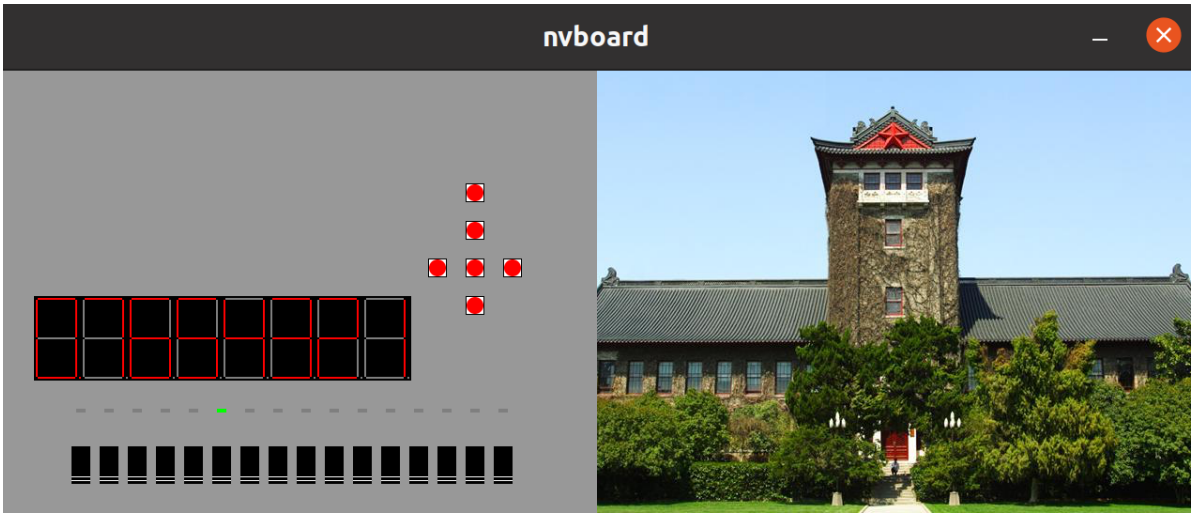
RTL 仿真行为 ：

打印波形：

参照着示例对仿真文件添加生成波形代码，然后用gtkwave查看波形为：

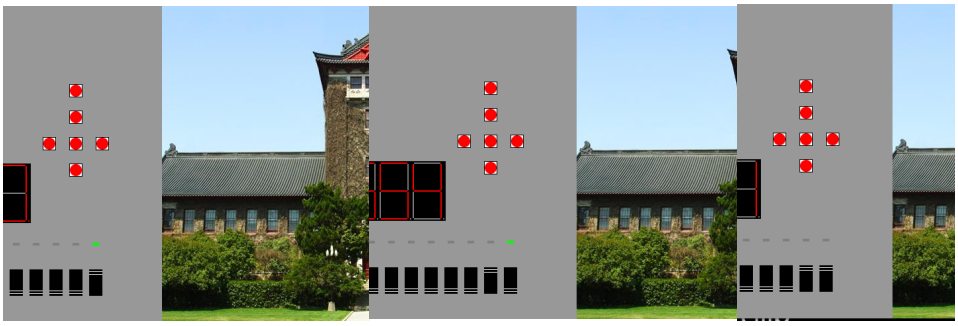


接入NVboard：



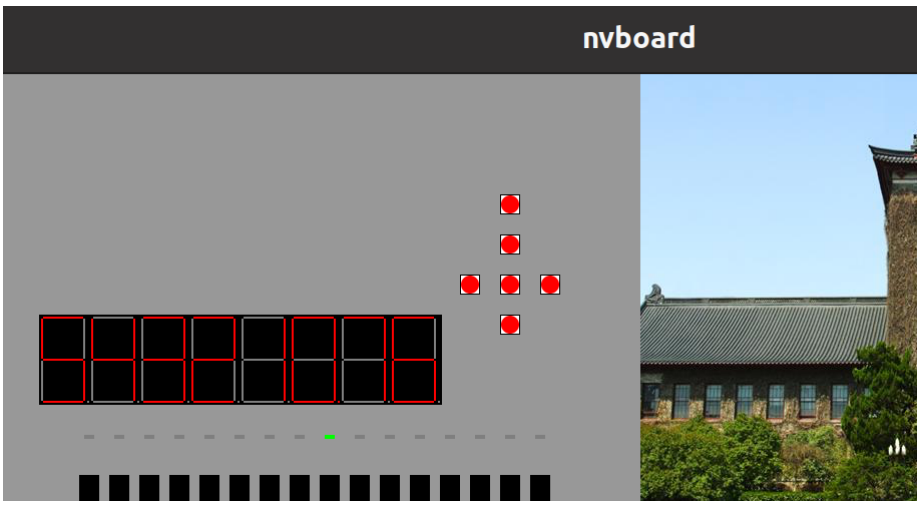
nvboard 实现双控开关:

刚开始的时候想要自己写一个makefile去管理nvboard仿真，学了几个小时发现没有办法在短时间内掌握诸多makefile、shell编程的内容，因此就直接修改了示例代码中的top.v进行双控开关的控制（暂时折中的方法，先占个坑，以后慢慢补充makefile、shell编程知识，完全写出自己的makefile），结果如下：



用最右侧的两个拨码开关控制最右边的那个LED灯，关系为异或。

流水灯接入nvboard：



由于流水灯是动态图，使用截图只能展示其中的一个状态；

2.2 数字电路基础实验

实验一：选择器

实验二：编码器

实验三：运算器

实验六：移位寄存器和桶形移位器

实验七：状态机

实验八：VGA显示

实验九：字符输入界面

PA0 开发环境配置

完成了PA0的所有内容。

编译nemu时遇到bug，查看出错信息之后发现是缺少bison，执行安装命令之后，完成正常编译，出现welcome界面。

大插曲： 刚开始做PA时，有个红色的**重要性超越实验的原则与方法**说要针对自己的Ubuntu版本选择对应的源，对这个不太了解，就没有做任何改变，一通安装相应的工具，`make menuconfig`的时候仍然能正常进入welcome界面，但是做到PA1开头测试键盘的时候，一直segmentation fault错误，看了jyy的PA习题课之后知道了如何调试这个bug，可是尝试了众多方法后无果，在群里讨论的时候有个喵喵酱的小伙伴说是不是没有更换源，我瞬间想到了自己忽略的红色提醒！换源之后，更新了一下包，这个时候才正常测试键盘，可以运行马里奥小游戏了！重要细节一定需要认真对待！

反思总结

官方手册和官方Tutorial最为可靠。

反思：每做完一小步，一定要测试是否做的正确！增量式开发！螺旋式的学习，不断的回过头去看之前做过的东西会有新的收获，并且能够优化前期不足的地方！

PA1 开天辟地的篇章

计算机可以没有寄存器吗?

计算机没有寄存器仍然可以工作。

编程模型：

计算机执行指令直接对存储器进行读写，完成寄存器需要完成的工作，但指令的读写会变得很慢；程序在编译、链接之后生成可执行文件就是一条条的指令，没有寄存器计算机执行程序的速度会变得很慢。

尝试理解计算机如何计算

计算机最擅长的就是做重复的运算；`1+2+...+100` 的计算过程就是个重复执行加法运算的过程，每一次运算首先读取寄存器里面的值进行计算，计算结束把结果存到寄存器里面，不停的重复这些动作，直到完成最终计算。

从状态机视角理解程序运行： 继续画出状态机

三元组（`PC`, `r1`, `r2`）

```
( 0, x, x) -> ( 1, 0, x) -> ( 2, 0, 0) -> ( 3, 0, 1)->
( 4, 1, 2) -> ( 5, 3, 3) ... ->
( 101, 4950, 99) ->( 102, 5050, 100)
```

kconfig生成的宏与条件编译：宏是如何工作的？

宏在使用的位置将其替换，替换的过程发生在预编译。

为什么全部都是函数?

函数对外提供接口，只需要知道函数具备的功能和API，使用方便；模块化设计方便维护和多人协同开发。

参数的处理过程：参数是从哪里来的？

参数从命令行读入；

究竟需要执行多久？

`static void execute(uint64_t n)` 函数是无符号整形变量，传入参数为 `-1` 时，转换为无符号数是该数据位下能表示的最大的数字，程序会执行的 `-1` 转化为无符号数字大小的次数！

调用 `cpu_exec()` 的时候传入了参数 `-1` ", 这一做法属于未定义行为吗? 待日后查阅C99

优美的退出

NEMU `main()` 返回值非0的时候会触发退出错误，`main()` 函数返回值为 `is_exit_status_bad()` ，进入 `is_exit_status_bad()` ：

```
int is_exit_status_bad() {
  int good = (nemu_state.state == NEMU_END && nemu_state.halt_ret == 0) ||
    (nemu_state.state == NEMU_QUIT);
  return !good;
}
```

可见，nemu的退出状态由`nemu_state.state`决定，因此可见，出错原因为`cmd_q()`函数调用时候未能正确调整`nemu_state.state`的值，就会出现以下错误：

```
Welcome to riscv64-NEMU!
For help, type "help"
(nemu) q
make: *** [/home/ypwang/ysyx_record/ysyx-workbench/nemu/scripts/native.mk:23: ru
n] Error 1
```

因此在`cmd_q()`函数调用时候，修改`nemu_state.state`的值为：

```
static int cmd_q(char *args) {
  nemu_state.state = NEMU_QUIT;    // wyp: finish normal quit
  return -1;
}
```

解决之后的效果为：

```
Welcome to riscv64-NEMU!
For help, type "help"
(nemu) q
(base) ypwang@ypwangPC:~/ysyx_record/ysyx-workbench/nemu$
```

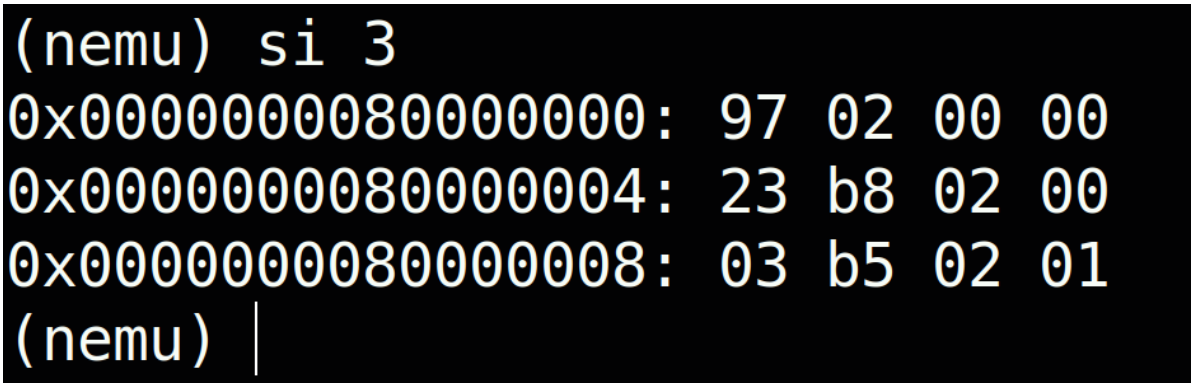
PA 1.1 简易调试器

单步执行

当nemu运行之后，键入`si [N]`命令让程序单步执行`N`条指令后暂停执行，当命令为`si`时，单步执行，因此只需要实现`sdb.c`中的`si[N]`命令

```
static int cmd_si(char *args){
  int step;
  if(args ==NULL) step = 1;
  else sscanf(args, "%d", &step);
  cpu_exec(step);                  // n steps
  return 0;
}
```

实现效果为



打印寄存器

打印寄存器需要输入的命令为`info r`,sdb.c中需要实现info r命令

```
static int cmd_info(char *args){
  if(args[0] == 'r') { isa_reg_display();}
  if(args[0] == 'w') { ;}
  return 0;
}

//isa_reg_display();的实现为：
void isa_reg_display() {
  int i;
    for(i=0; i< ARRLEN(regs); i++){   //ARRLEN(regs)为计算reg length的宏
        printf("No. %d register: %s\tx%08lx\n", i, regs[i], cpu.gpr[i]);
    }
    printf("No. PC register: PC\t %08lx\n", cpu.pc);
}
```

note: w为待实现的监视点命令

打印结果为：

```
(nemu) info r
No. 0 register: $0      x00000000
No. 1 register: ra      x00000000
No. 2 register: sp      x00000000
No. 3 register: gp      x00000000
No. 4 register: tp      x00000000
No. 5 register: t0      x00000000
No. 6 register: t1      x00000000
No. 7 register: t2      x00000000
No. 8 register: s0      x00000000
No. 9 register: s1      x00000000
No. 10 register: a0     x00000000
No. 11 register: a1     x00000000
No. 12 register: a2     x00000000
No. 13 register: a3     x00000000
No. 14 register: a4     x00000000
No. 15 register: a5     x00000000
No. 16 register: a6     x00000000
No. 17 register: a7     x00000000
No. 18 register: s2     x00000000
No. 19 register: s3     x00000000
No. 20 register: s4     x00000000
No. 21 register: s5     x00000000
No. 22 register: s6     x00000000
No. 23 register: s7     x00000000
No. 24 register: s8     x00000000
No. 25 register: s9     x00000000
No. 26 register: s10    x00000000
No. 27 register: s11    x00000000
No. 28 register: t3     x00000000
No. 29 register: t4     x00000000
No. 30 register: t5     x00000000
No. 31 register: t6     x00000000
No. PC register: PC      80000000
(nemu)
```

扫描内存

扫描客户计算机的内存数据

扫描内存则需要实现sdb中的`x N EXPR`命令，由于还未完成表达式求值，故使用`x 10 0x80 00 0000`（32-bit地址）作为输入命令

sdb.c `cmd_x`实现如下:

```
1  static int cmd_x(char *args){
2
3    if(args == NULL){
4      printf("Wrong Command! Please try again!\n");
5      return 0;
6    }
7    int num;
8    long unsigned int init_addr;
```

```
  sscanf(args, "%d %lx", &num, &init_addr);
  int i;
  for(i = 0; i < num; i++){
    printf("Memory address:0x%lx\tData: 0x%lx\n", init_addr,
vaddr_read(init_addr, 8));
    init_addr += 8; //  64-bit machine 8*8 Byte = 64-bit
  }
  return 0;
}
```

扫描内存结果：

```
Welcome to riscv64-NEMU!
For help, type "help"
(nemu) x 10 0x80000000
Memory address:0x 80000000      Data: 0x 297
Memory address:0x 80000004      Data: 0x 2b823
Memory address:0x 80000008      Data: 0x 102b503
Memory address:0x 8000000c      Data: 0x 100073
Memory address:0x 80000010      Data: 0x deadbeef
Memory address:0x 80000014      Data: 0x 49fb3392
Memory address:0x 80000018      Data: 0x 5f9db71b
Memory address:0x 8000001c      Data: 0x 270329c3
Memory address:0x 80000020      Data: 0x 43969e66
Memory address:0x 80000024      Data: 0x 65e1281f
(nemu)
```

PA1.1完成