组成原理基础 讲座

主讲人：刘泓尊

10.26

主要内容

- 作业、实验
- Git
- Verilog
- Vivado

书面作业的若干问题

1. 以下关于字符编码描述错误的是_____。

   A. ASCII 码使用一个字节的编码，包含了所有的英文大小写字符

   B. UTF-8 码使用两个字节的编码，包含了英文字符，中文字符以及日文字符等

   C. 字符编码不包含字符如何显示的信息

   D. 字符显示放大时，使用矢量字体的字符不会失真

- ASCII 码使用8 位来表示 256 种可能的字符。基础ASCII码（控制符号、字母、数字、英文标点）128个，加上扩展ASCII 码共256个。一般不认为是7bit.

- UTF8：针对Unicode的一种可变长度字符编码。UTF-8表示最小单位1字节，它使用1、2、3字节等进行编码

计算题：请使用加减交替法进行运算。x=0.1010，y=0.1101，求 x÷y，写出计算过程。

- 应求出商和余数。
- 注意掌握：IEEE754浮点的二进制表示、检错纠错码、布斯乘法、加减交替除法

书面作业的若干问题

3. 判断题（简述理由）： RISC 指令的处理器运行频率要比 CISC 指令的处理器频率高。

- 错误。和处理器架构、流水线深度、底层硬件都有关。
- 如果认为正确，且说明了“在相同功耗/相同成本”等合理的控制变量条件下，也认为对了。

4. 判断题（简述理由）： 只要运算器具有加法器和移位功能，再增加一些控制逻辑就能实现乘除运算。

- 正确。
- 如果认为“控制逻辑”不包括寄存器，所以无法实现。也认为对了

前几次小实验的若干问题

- 实验前小测

4. （多选）下列叙述中，正确的是（AB）：
   a. 在波形仿真中遇到意料之外的`X`值或`Z`值时，可以右键仿真里面的信号名称，点击查看所有driver。检查该信号的所有依赖信号值是否符合预期
   b. 位于 always @(*) 块中的 case 语句块，需要保证所有可能情况考虑完善
   c. 位于 always @(posedge clk) 块中的 if 语句块，需要保证所有分支情况考虑完善

- 组合逻辑块：所有逻辑分支里都要对驱动的wire赋值(=)，否则可能综合出锁存器latch
- 时序逻辑块：如果某一逻辑分支没有对某个寄存器赋值(<=)，则寄存器值保持不变

- 本课程实验不需要锁存器，锁存器会使得静态时序分析变得复杂
- 我们的实验板上的基本单元是LUT和FF，如果生成锁存器会消耗更多的资源

前几次小实验的若干问题

```
always_ff @(posedge clk) begin        You, seconds ago • Uncon
    if (rst) begin
        state_now <= READ_ADDR;
    end else if (~sram_stall_req && ~uart_stall_req) begin
        state_now <= state_nxt;
    end
end
```

```
always_comb begin
    unique case(state_now)
    IDLE: begin
        state_nxt = IDLE;
        if (load) state_nxt = READ;
        if (store) state_nxt = WRITE;
    end
    READ: begin
        state_nxt = IDLE;
    end
    WRITE: begin
        state_nxt = IDLE;
    end
    default: begin
        state_nxt = IDLE;
    end
    endcase
end
```

前几次小实验的若干问题

使用CPLD串口控制器，当同时需要读取和写入时，是应该优先读取还是优先写入，抑或是无所谓？为什么？（无所谓，因为有读写缓冲器。）

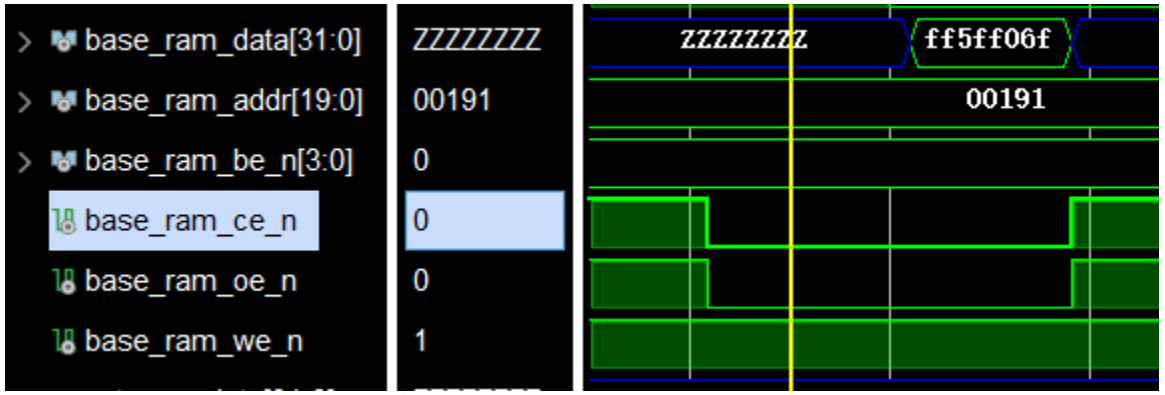
- UART：通用异步收发传输器。全双工、有读写FIFO、异步收发

前几次小实验的若干问题

- ALU实验
  - ALU本身最好写成一个组合逻辑块，不必涉及寄存器和时序逻辑。

- SRAM & UART实验
  - SRAM是没有时钟驱动的，给出信号在一段时间(~10ns)后返回数据或完成写入
  - CPLD UART 工作在11.0592M时钟下，在50MHz下维持一周期的信号可能CPLD接收不到，rdn和wrn信号可以多维持几个周期

SRAM读写



- 读SRAM
  - 平台实测12ns，50M时钟(20ns)下信号拉低一个周期可以读出数据

调试工具

内存总线分析　远程 JTAG 连接　内部信号采样

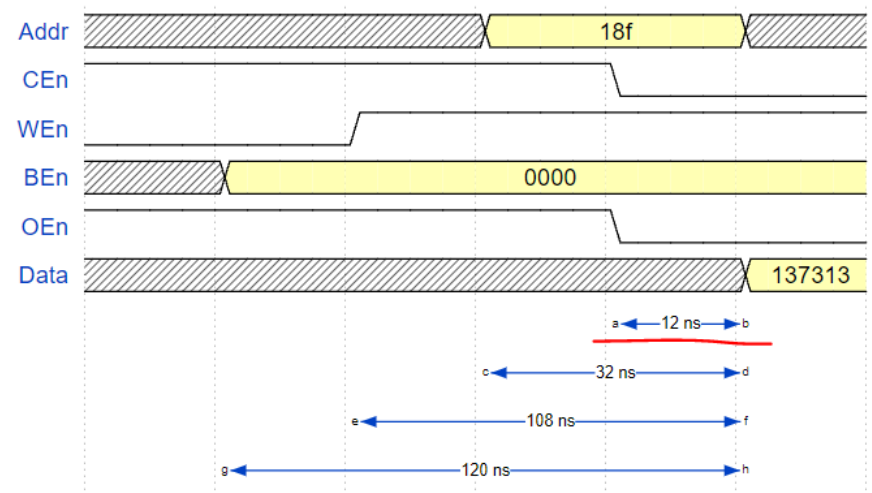
访存序列

本功能可以对BaseRAM访问进行采样。地址数据均以16进制显示。

记录数量 100　▶开始

#	读/写	地址(byte)	数据	BEn[3:0]	波形
0	R	63c	00137313	0000	
1	R	640	00031463	0000	
2	R	644	ff5ff06f	0000	
3	R	638	00528303	0000	
4	R	63c	00137313	0000	
5	R	640	00031463	0000	

访存波形

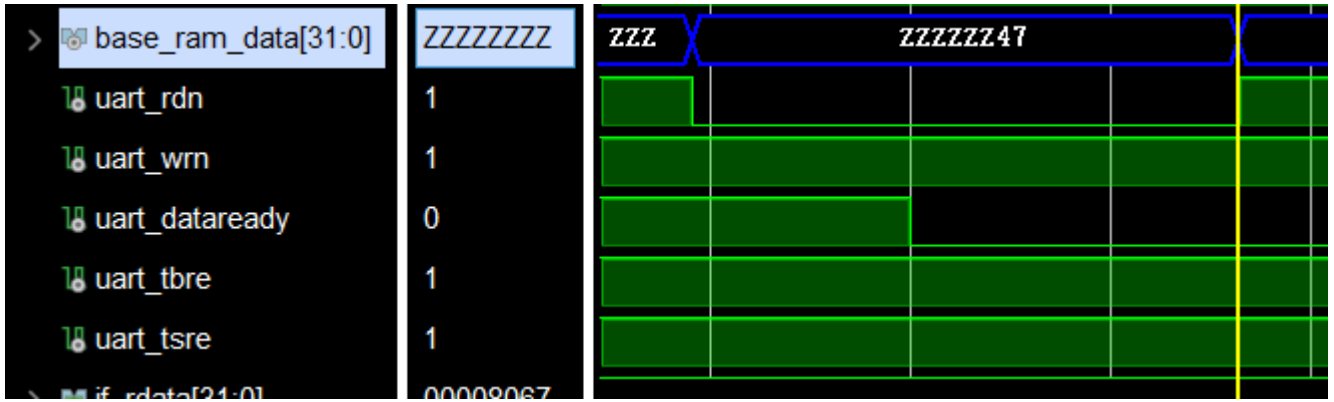


SRAM 读写

- 写SRAM
  - we_n 的下降沿锁存地址、上升沿锁存数据
  - 考虑到可能存在的延迟
    - addr 不要在 we_n 下降沿的时候才给出
    - Data 不要在 we_n 上升沿的时候就撤走
    - Addr, data 可以在 we_n 为 1‘b0 前后各维持一个周期
    - we_n拉低10ns就够了，也可以多维持一段时间

实验4 UART读写

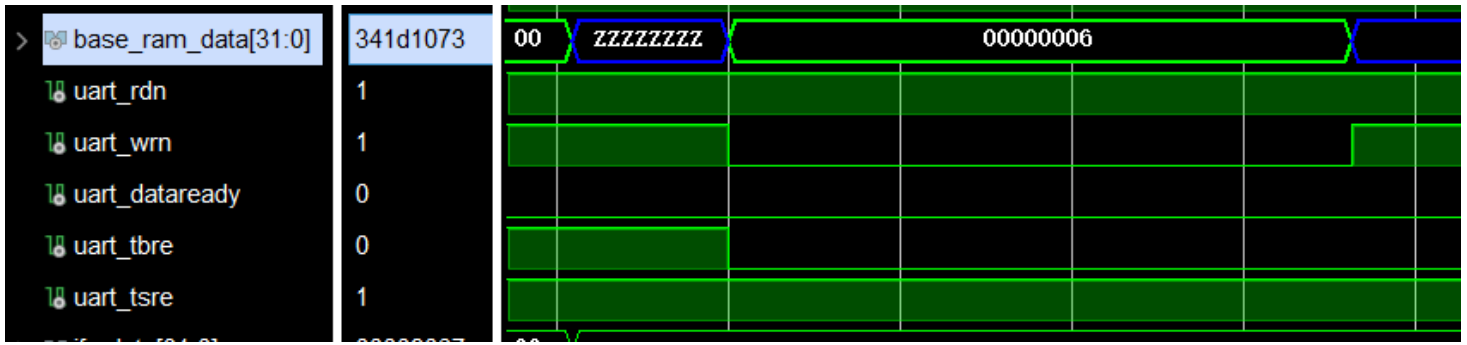
- 读串口
  - 等dataready
  - rdn拉低一个或多个周期
  - rdn 拉高




```
always_comb begin: uart_fsm
    unique case (state_now)
    IDLE: begin
        uart_wrn = 1'b1;  uart_rdn = 1'b1;
        state_nxt = IDLE;
        if (load) state_nxt = READ_WAIT_READY;
        if (store) state_nxt = WRITE_BEGIN_1;
    end
    READ_WAIT_READY: begin
        uart_wrn = 1'b1;  uart_rdn = 1'b1;
        if (uart_dataready) begin
            state_nxt = READ_RECEIVING_0;
        end else begin
            state_nxt = READ_WAIT_READY;
        end
    end
    READ_RECEIVING_0: begin
        uart_wrn = 1'b1;  uart_rdn = 1'b0;
        state_nxt = READ_RECEIVING;
    end
    READ_RECEIVING: begin
        uart_wrn = 1'b1;  uart_rdn = 1'b0;
        state_nxt = READ_FINISH;
    end
    READ_FINISH: begin
        uart_wrn = 1'b1;  uart_rdn = 1'b1;
        state_nxt = IDLE;
    end
```

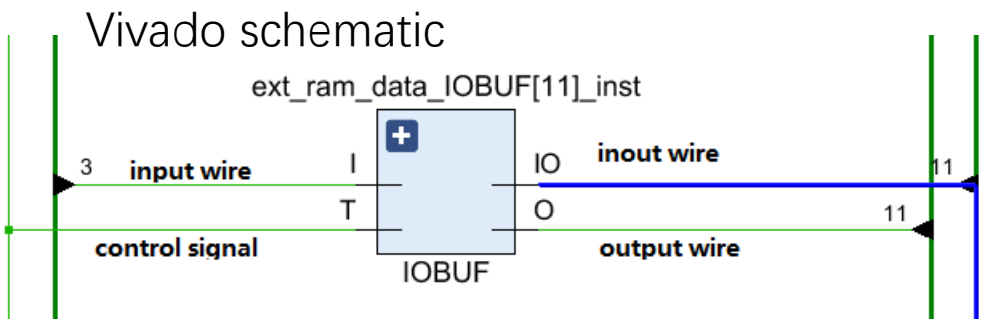
实验4 UART读写

- 写UART
  - wrn拉低并保持一个或多个周期
  - wrn拉高
  - 等tbre拉高
  - 等tsre拉高




```
    WRITE_BEGIN_1: begin
        uart_wrn = 1'b0;  uart_rdn = 1'b1;
        state_nxt = WRITE_BEGIN_2;
    end
    WRITE_BEGIN_2: begin
        uart_wrn = 1'b0;  uart_rdn = 1'b1;
        state_nxt = WRITE_BEGIN_3;
    end
    WRITE_BEGIN_3: begin
        uart_wrn = 1'b0;  uart_rdn = 1'b1;
        state_nxt = WRITE_WAIT_READY;
    end
    WRITE_WAIT_READY: begin
        uart_wrn = 1'b1;  uart_rdn = 1'b1;
        if (uart_tbre) state_nxt = WRITE_WAIT_FINISH;
        else state_nxt = WRITE_WAIT_READY;
    end
    WRITE_WAIT_FINISH: begin
        uart_wrn = 1'b1;  uart_rdn = 1'b1;
        if (uart_tsre) state_nxt = IDLE;
        else state_nxt = WRITE_WAIT_FINISH;
    end
    default: begin
        uart_rdn = 1'b1;  uart_wrn = 1'b1;
        state_nxt = IDLE;
    end
    endcase
end
```

实验4 总线控制



- 不要把inout线接入sram_controller和uart_controller两个子模块里
- inout线可以在top文件里处理，具体来说
  - 对inout线的读写需要写成三态门
  - 三态门的逻辑把inout线分成input和output两个线接入controller

```
// extram BUS control, in top.v
assign ext_ram_data = extram_write_bus ? extram_bus_data_out : 32'bz;
assign extram_bus_data_in = ext_ram_data;
```

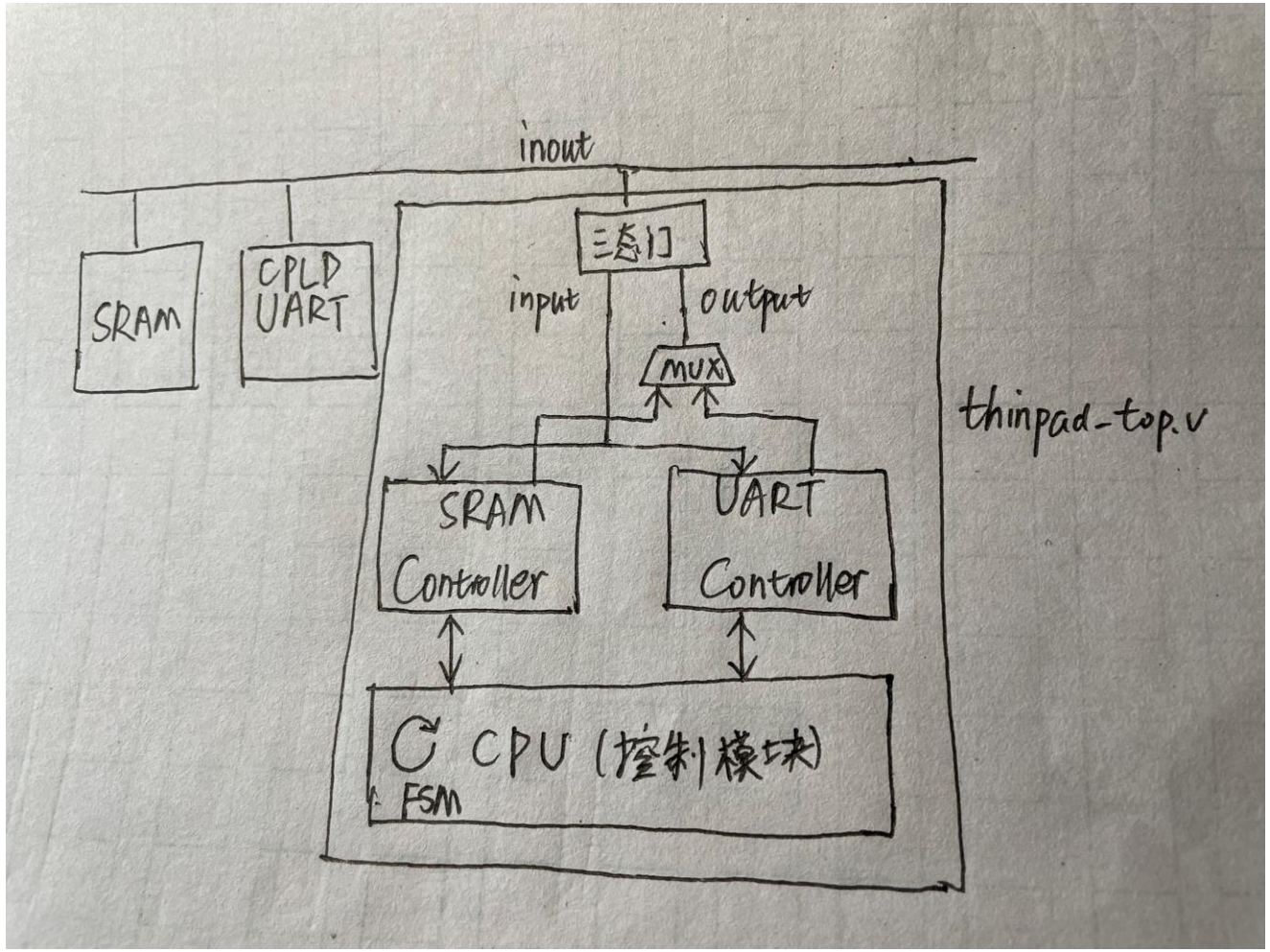
```
sram_controller extram_controller(
    .clk(clk),
    .rst(rst),

    .bus_data_in(extram_bus_data_in),   // input wire[31:0]
    .bus_data_out(extram_bus_data_out), // output wire[31:0]
    .write_bus(extram_write_bus),       // output wire, 三态门使能信号
```

实验4 总线仲裁

- SRAM和UART共享总线的冲突
  - 将inout总线拆成in和out两组线之后，分别接入sram_controller和uart_controller
  - 由于 对总线输出高阻态的逻辑 交给了上层的单一逻辑，此时便不存在sram_controller和uart_controller同时 写总线 的冲突了
  - 只需要在有sram controller或uart controller有写请求时，对总线给出正确的数据来源即可

```
// baseram BUS control, in top.v
// bus write priority: UART > BASERAM
wire[31:0] write_bus_data;
assign write_bus_data = uart_write_bus ? uart_bus_data_out : baseram_bus_data_out;
assign base_ram_data = (uart_write_bus | baseram_write_bus) ? write_bus_data : 32'bz;
assign baseram_bus_data_in = (uart_write_bus | baseram_write_bus) ? 32'b0 : base_ram_data;
assign uart_bus_data_in = (uart_write_bus | baseram_write_bus) ? 32'b0 : base_ram_data;
```

实验4 架构参考


inout
三态门
input
output
MUX
SRAM
CPLD
UART
thinpad_top.v
SRAM
Controller
UART
Controller
CPU (控制模块)
FSM

Git的基本使用

- git clone git@git.tsinghua.edu.cn:cod-ta/cod21-xxx.git
- 单分支基本操作
  - git status　// 查看改动与其他信息
  - git add .　// 提交到暂存区, 在commit前可以进行多次git add
  - git commit –m “finish uart controller”　//提交到本地仓库, 成为一个版本
  - git push origin master　// 将master分支的新版本提交到远程
- 多分支版本控制
  - git branch　　// 查看本地分支信息
  - git branch lab4　//新建分支
  - git checkout lab4 //切换到lab4分支
  - coding…
  - git add .
  - git commit –m “finish lab4”
  - git push origin lab4　// 将lab4分支的新版本提交到远程

Git的基本使用

- 多人合作
  - git pull origin // 把远程master分支合并入本地当前分支
  - git pull origin fix_lab4:lab4 // 把远程fix_lab4分支合并入本地lab4分支
  - git pull = git fetch + git merge

- 版本合并
  - git branch dev // 创建新分支进行开发
  - Developing …
  - git add .
  - git commit –m “finish lab 4” // 提交该版本
  - git checkout master // 切换回master分支
  - git pull origin master // 合并远程队友可能提交的新版本到本地的master
  - git merge –no-ff dev // 此时在master分支，把dev分支合并入master
  - 如果不能自动合并，需要手动解决冲突
  - git push origin master // 确认无误后，提交到远程master分支

Git的进阶使用

- 提交完了才发现漏掉了几个文件没有添加

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

最终你只会有一个提交——第二次提交将代替第一次提交的结果。

- 取消暂存(git add之后)的文件test.v
  - git reset HEAD test.v
- 取消对某文件的修改，将其撤销回上个版本的状态
  - git checkout -- test.v
- **WARN**: 在 Git 中任何 已提交 的东西几乎总是可以恢复的，
  然而，任何你未提交的东西丢失后很可能再也找不到了。

Verilog & SystemVerilog

- 变量命名
  - 信号极性为低有效：用 _n 后缀表示
- 类型使用
  - 需要寄存器时用 reg，纯组合逻辑用 wire
  - 对于常量信号，请使用 wire 类型并用 assign 进行赋值，而不要用 reg

```
// GOOD
wire one;
assign one = 1'b1;

// BAD
reg one;
always @(*) begin
  one = 1'b1;
end
```

```
// GOOD
wire c;
always_comb begin
  c = a + b;
end

// BAD
reg c;
always_comb begin
  c = a + b;
end

// GOOD
reg c;
always_ff @(posedge clock) begin
  c <= a + b;
end

// BAD
wire c;
always_ff @(posedge clock) begin
  c <= a + b;
end
```

Verilog & SystemVerilog

- 组合逻辑仅使用阻塞赋值(=)

```
// Verilog
always @(*) begin
  c = a + b;
end

// System Verilog
always_comb begin
  c = a + b;
end

// GOOD
assign c = a + b;
```

- 时序逻辑仅使用非阻塞赋值(<=)

```
// Verilog
always @(posedge clock) begin
  c <= a + b;
end

// System Verilog
always_ff @(posedge clock) begin
  c <= a + b;
end
```

Verilog & SystemVerilog

- always敏感信号列表
  - 不要列举敏感信号

```
// BAD
always @ (b, c) begin
  a = b + c;
end
```

  - 不要使用下降沿触发

```
// BAD: do not use negedge
always @ (negedge clock) begin
end
```

  - 不要使用除了时钟和复位以外的信号做边沿触发

```
// BAD: do not use non-clock/reset signals
always @ (posedge signal) begin
end
```

Verilog & SystemVerilog

- 复位与时钟
  - 时序逻辑不要使用时钟信号

```
// BAD
always @ (posedge clock) begin
  if (clock) begin
    a <= 1;
  end
end
```

  - 推荐使用同步复位

```
// Verilog
always @(posedge clock) begin
  if (reset) begin
    c <= 1'b0;
  end else begin
    c <= a + b;
  end
end

// System Verilog
always_ff @(posedge clock) begin
  if (reset) begin
    c <= 1'b0;
  end else begin
    c <= a + b;
  end
end
```

Verilog & SystemVerilog

- 不要在变量声明处赋值，因为不同的类型赋值的意义不同

```
// Wire
wire signal = 1;
// Equals to
wire signal;
assign signal = 1;

// Reg
reg signal = 1;
// Equals to
reg signal;
initial signal = 1;
```

- 寄存器(FF)通常情况下需要在自定义的 reset 信号有效时复位。

- initial语句只有仿真的时候有效，是不可综合上板的，不能代替复位初始化语句

Verilog & SystemVerilog

- 状态机：推荐把时序逻辑和组合逻辑分开
- 以 SRAM Controller 为例

```
// 状态常量
localparam IDLE = 2'd0;
localparam READ = 2'd1;
localparam WRITE = 2'd2;
// 当前周期的状态、下个周期的状态
reg[1:0] state_now;
/*****************************
 * WARN: 严格来说，Verilog 和 SystemVerilog
 * 不允许对 wire 类型进行 Procedural Assignment，
 * 也就是在 always 块中进行赋值，但很多环境中这个约束可以不遵守
 * 很可惜，Vivado 不允许对 wire 进行 Procedural Assignment
 * 如有需要,只能声明为reg，比如下面的state_nxt
 */
reg[1:0] state_nxt; // 因为只涉及组合逻辑，综合之后实际是wire
// 转移到下一个状态：组合逻辑
always @(*) begin
    case(state_now)
    IDLE: begin
        state_nxt = IDLE;
        if (load) state_nxt = READ;
        if (store) state_nxt = WRITE;
    end
    READ: state_nxt = IDLE;
    WRITE: state_nxt = IDLE;
    default: state_nxt = IDLE;
    endcase
end
```

```
// 状态更新：时序逻辑
always @(posedge clk) begin
    if (rst) begin
        state_now <= IDLE;
    end else begin
        state_now <= state_nxt;
    end
end
```

```
// 根据当前状态给出访存控制信号：组合逻辑
always @(*) begin
    case(state_now)
    READ: begin
        ram_be_n = 4'b0000;
        ram_ce_n = 1'b0;
        ram_oe_n = 1'b0;
        ram_we_n = 1'b1;
    end
    WRITE: begin
        ram_be_n = 4'b0000;
        ram_ce_n = 1'b0;
        ram_oe_n = 1'b1;
        ram_we_n = 1'b0;
    end
```

```
        t: begin // IDLE
        m_ce_n = 1'b1;
        m_oe_n = 1'b1;
        m_we_n = 1'b1;
        m_be_n = 4'b0000;
        e
```

SystemVerilog 状态机

```systemverilog
// SystemVerilog 状态声明
typedef enum logic[1:0] {
    IDLE,
    READ,
    WRITE
} sram_state_t;
// 变量声明        You, seconds ago • Un
sram_state_t state_now, state_nxt;
```

```systemverilog
// 状态更新：时序逻辑
always_ff @(posedge clk) begin
    if (rst) begin
        state_now <= IDLE;
    end else begin
        state_now <= state_nxt;
    end
end
```

```systemverilog
// 转移到下一个状态：组合逻辑
always_comb begin       liuhz, 2 weeks a
    unique case(state_now)
    IDLE: begin
        state_nxt = IDLE;
        if (load) state_nxt = READ;
        if (store) state_nxt = WRITE;
    end
    READ: begin
        state_nxt = IDLE;
    end
    WRITE: begin
        state_nxt = IDLE;
    end
    default: begin
        state_nxt = IDLE;
    end
    endcase
end
```

```systemverilog
// 根据当前状态给出访存控制信号：组合逻辑
always_comb begin
    unique case(state_now)
    READ: begin
        ram_be_n = 4'b0000;
        ram_ce_n = 1'b0;
        ram_oe_n = 1'b0;
        ram_we_n = 1'b1;
    end
    WRITE: begin
        ram_be_n = 4'b0000;
        ram_ce_n = 1'b0;
        ram_oe_n = 1'b1;
        ram_we_n = 1'b0;
    end
    default: begin // IDLE
        ram_ce_n = 1'b1;
        ram_oe_n = 1'b1;
        ram_we_n = 1'b1;
        ram_be_n = 4'b0000;
    end
    endcase
end
```

实验5读写串口

- 建议：不需要硬件等dataready, tbre和tsre
- 硬件只需要拉低wrn和rdn即可
- 对于串口状态的检查由软件（汇编程序）实现
- 参考监控程序的读写串口代码

```
READ_SERIAL: // 读串口：将读到的数据写入a0低八位
    li t0, 0x10000000
.TESTR:
    lb t1, %lo(5)(t0)
    andi t1, t1, 0x01  // 截取读状态位 dataready
    bne t1, zero, .RSERIAL  // 状态位非零可读进入读
    j .TESTR       // 检测验证
.RSERIAL:
    lb a0, %lo(0)(t0)
    jr ra
```

```
WRITE_SERIAL:  # 写串口：将a0的低八位写入串口
    li t0, 0x10000000
.TESTW:
    lb t1, %lo(5)(t0)  # 查看串口状态
    andi t1, t1, 0x20  # 截取写状态位
    bne t1, zero, .WSERIAL  # 状态位非零可写进入写
    j .TESTW                # 检测验证，忙等待
.WSERIAL:
    sb a0, %lo(0)(t0)  # 写入寄存器a0中的值
    jr ra
```

Hints

- Verilog coding style
  - https://github.com/thu-cs-lab/verilog-coding-standard
- 积极利用和维护问题集锦
  - https://docs.qq.com/doc/DSEJBZVlZemxiQ3lz
- 上手仿真，早用早享受
- 积极有效的提问
  - 软件：qemu, supervisor-rv的版本，编译参数，如何运行，结果怎样
  - 硬件：关键代码、仿真信息、上板结果
  - 早问早舒服
- 快乐造机，拒绝内卷

Q&A