计算机组成原理实验

指导教师： 徐爱萍

实验目的

- 融会贯通计算机组成与设计课程所教授的知识，通过对知识的综合应用，加深对CPU系统各模块的工作原理及相互联系的认识。
- 学习采用EDA（Electronic Design Automation）技术设计MIPS单周期CPU/多周期/流水线CPU的技术与方法；
- 培养科学研究的独立工作能力，取得CPU设计与仿真的实践和经验。

设计与调试任务

- 按给定的指令格式和指令系统，进行单周期CPU设计；
- 在ModelSim平台上进行仿真调试并测试结果；
- 整理出实习报告。
- 教材：
  - 数字逻辑与组成原理实践教程，张冬冬等 著

硬件描述语言Verilog

- Verilog HDL是一种硬件描述语言，Verilog HDL语言不仅定义了语法，而且对每个语法结构都定义了清晰的模拟、仿真语义。
- 用这种语言编写的模型能够使用Verilog仿真器进行验证。
- Verilog HDL中有两类数据类型：线网数据类型和寄存器数据类型。线网类型表示构件间的物理连线，而寄存器类型表示抽象的数据存储元件。
- Verilog HDL 还具有内置逻辑函数，例如&（按位与）和|（按位或）。
- 对高级编程语言结构，例如条件语句、情况语句和循环语句，语言中都可以使用。

模块

- 模块是Verilog 的基本描述单位，用于描述某个设计的功能或结构及其与其他模块通信的外部端口，一个模块可以在另一个模块中使用。
- 一个模块的基本语法如下：
- module HalfAdder (A, B, Sum, Carry) ;
- input A, B;
- output Sum, Carry;
- assign #2 Sum = A ^ B;
- assign #5 Carry = A & B;
- endmodule
- 模块的名字是HalfAdder。模块有4个端口: 两个输入端口A和B，两个输出端口Sum和Carry。由于没有定义端口的位数, 所有端口大小都为1位；同时, 由于没有各端口的数据类型说明, 这四个端口都是线网数据类型。

时延

- Verilog HDL模型中的所有时延都根据时间单位定义。下面是带时延的连续赋值语句实例。
- assign #2 *Sum = A^B;*
- # 2指2个时间单位。#2 代表2 n s。
- 如下所示:
- ` timescale 1ns /100ps
- 此语句说明时延时间单位为1 n s并且时间精度为100ps (时间精度是指所有的时延必须被限定在0 . 1 n s内)。

行为描述方式

- 设计的行为功能使用下述过程语句结构描述:
- （1） initial语句：此语句只执行一次。
- （2） always语句：此语句总是循环执行, 或者说此语句重复执行。
- （3） 连续赋值语句的语法为:
- assign [delay] LHS_ net = RHS_ expression ;
- 右边表达式使用的操作数无论何时发生变化 , 右边表达式都重新计算 , 并且在指定的时延后变化值被赋予左边表达式的线网变量。时延定义了右边表达式操作数变化与赋值给左边表达式之间的持续时间。如果没有定义时延值 , 缺省时延为0。

verilog语言中always的用法

- always@(敏感事件列表) 用于描述时序逻辑
- 敏感事件上升沿 posedge，下降沿 negedge，或电平
- 敏感事件列表中可以包含多个敏感事件，但不可以同时包括电平敏感事件和边沿敏感事件，也不可以同时包括同一个信号的上升沿和下降沿，这两个事件可以合并为一个电平敏感事件。
- 在新的verilog2001中“，”和“or”都可以用来分割敏感事件了，可以用“*”代表所有输入信号，这可以防止遗漏。
- 合法的写法：
- always@ *
- always@ (posedge clk1,negedge clk2)
- always@ (a or b)
- `timescale 100ns/100ns //定义仿真基本周期为100ns
- always #1 clk=~clk //#1代表一个仿真周期即100ns

Verilog 模块的结构由在**module**和**endmodule**关键词之间的四个主要部分组成：

- **1.** 端口信息： **module combination(a, b, c, d);**
- **2.** 输入/输出说明： **input a, b, c ;**
  output d ;

-// 输入/输出端口信号类型声明，缺省为**wire**型：

- **3.** 内部信号： **wire x;**

- **4.** 功能定义： **assign d = a | x ;**
  assign x = (b & ~c);
  endmodule

整数与实数常量例子

12	unsized decimal (zero-extended to 32 bits)
'h83a	unsized hexadecimal (zero-extended to 32 bits)
8'b1100 0001	8-bit binary
16'hff01	16-bit hexadecimal
32'bz01x	Z-extended to 32 bits
3'b1010 1101	3-bit number, truncated to 3'b101
6.3	decimal notation
32e- 4	scientific notation for 0.0032
4.1E3	scientific notation for 4100

字符串要在一行中用双引号括起来，也就是不能跨行。

"This is a normal string"

格式符

%h	%o	%d	%b	%c	%s	%t
hex	oct	dec	bin	ACSII	string	time

系统任务和函数

使用方式：**$<identifier>**

- **$**符号指示这是系统任务和函数
- 系统函数有很多，如：
  - 返回当前仿真时间**$time**
  - 显示/监视信号值**($display, $monitor)**
  - 停止仿真**$stop**
  - 结束仿真**$finish**

如： **$monitor($time, "a = %b, b = %h", a, b);**

当信号**a**或**b**的值发生变化时，系统任务**$monitor**显示当前仿真时间，信号**a**值**(**二进制格式**),** 信号**b**值（**16**进制格式）。

$display("R[%4X]=%8X", A3, rf[A3]);

Verilog HDL 数据类型

Verilog HDL 有两大类数据类型。

1. 线网类型**(net type):** 表示**Verilog**结构化元件间的物理连线。它的值由驱动元件的值决定，例如连续赋值或门的输出。如果没有驱动元件连接到线网，线网的缺省值为**z**。
2. 寄存器类型**(register type):**表示一个抽象的数据存储单元，它只能在**always**语句和**initial**语句等过程语句中被赋值，并且它的值从一个赋值到另一个赋值被保存下来。寄存器类型的变量具有**x**的缺省值。

线网数据类型-使用语法

- 例1：wire rst, data; //1位的复位线和数据线。
- 例2： wire [2:0] Addr1 ; //Addr1是3位地址线
- wire [3:1] Addr2; //Addr2是3位地址线

线网型变量使用举例

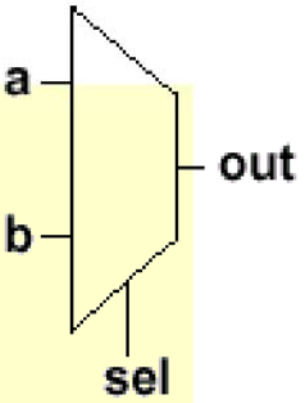
例如：两路选择器的**RTL**级描述

```
module mux2to1 (out, a, b, sel);
 input a, b, sel;
 output out;

 wire out;

 assign out=(sel)?b:a;

//input a, b, sel没声明信号类型，缺省为wire型
endmodule
```



- 寄存器类型reg-使用语法
- reg是寄存器数据类型最常见的数据类型。
- 例1： reg [3:0] counter1 ； //4 位寄存器。
- 例2： reg counter2 ； // 1位寄存器。
- 例3： reg [31:0] data_buffer1 ;
- reg [32:1] data_buffer2 ;

寄存器型变量使用举例

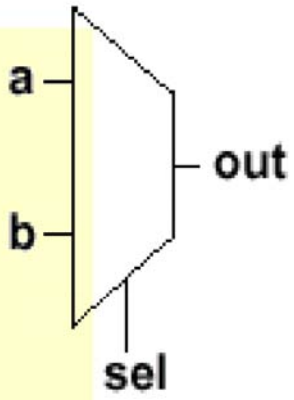
例如：两路选择器的**RTL**级描述

```
module mux2to1 (out, a, b, sel);
 input a, b, sel;
 output out;

 reg out;

always @( sel or a or b)
   if (! sel)   out = a;
   else         out = b;

//input a, b, sel没声明信号类型，缺省为wire型
endmodule
```



- 两种主要的信号类型：
- - 寄存器类型：代表 reg
- 在always 等过程块中被赋值的信号，往往代表触发器；

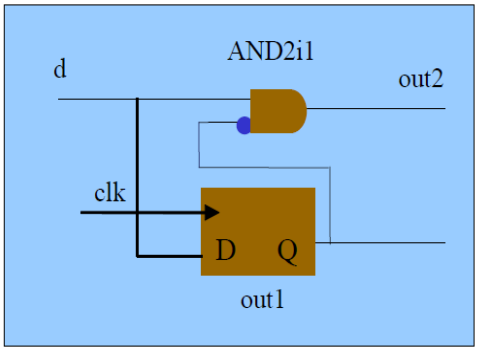
- - 连线类型：代表 wire
- 用 assign 关键词指定连续/持续赋值所描述的组合逻辑的信号或连线。

Verilog中reg与wire的不同点

- 用寄存器 (reg)类型变量生成触发器的例子：

```
module  rw2( clk, d, out1, out2 ) ;
   input clk, d;
   output out1, out2;
   reg out1;
   wire out2;

   assign out2 =d  & ~out1;
  always @(posedge clk)
    out1 <= d ;
endmodule
```



3.4.1 算术操作符

+	加
-	减
*	乘
/	除
%	模

3.4.2 逻辑运算符

逻辑操作符有：

&& (逻辑与)

|| (逻辑或)

！(逻辑非)

3.4.3 按位操作符

按位操作符有：

~（一元非）

&（二元与）

|（二元或）

^（二元异或）

~^, ^~（二元异或非）

3.4.4 关系运算符

关系操作符有：

>（大于）

<（小于）

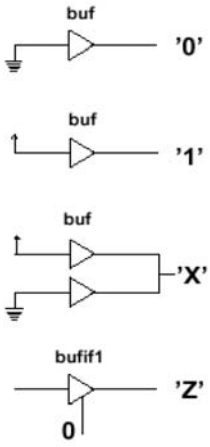
>=（不小于）

<=（不大于）

3.4.5 相等操作符

==	逻辑等
!=	逻辑不等

Verilog采用的四值逻辑系统



'0', Low, False, Logic Low, Ground,VSS, Negative Assertion

'1', High, True, Logic High, Power, VDD, VCC, Positive Assertion

'X' Unknown: Occurs at Logical Which Cannot be Resolved Conflict

'Z', High Impedance, Tri- Stated, Disabled Driver (Unknown)

3.4.6 条件操作符

条件操作符的语法为：

<LHS> = <condition> ? <true_expression>:<false_expression>;

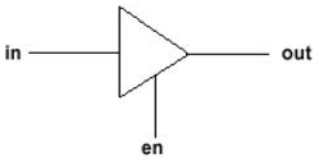
其意思是：**if condition is TRUE, then LHS=true_expression, else LHS = false_expression**

每个条件操作符必须有三个参数，缺少任何一个都会产生错误。最后一个操作数作为缺省值。

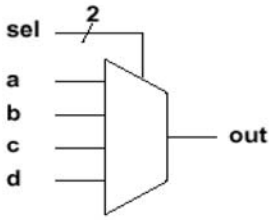
例如：**assign out = (sel == 0) ? a : b;**

若**sel**为**0**则**out =a**；若**sel**为**1**则**out = b**；如果**sel**为**x**或**z**，则结果可能为**x**或**z**。

条件操作符-使用实例




```
module likebufif( in, en, out);
    input in;
    input en;
    output out;
  assign out = (en == 1) ? in : 'bz;
endmodule
```




```
module like4to1( a, b, c, d, sel, out);
    input a, b, c, d;
    input [1: 0] sel;
    output out;
  assign out = sel == 2'b00 ? a :
               sel == 2'b01 ? b :
               sel == 2'b10 ? c : d;
endmodule
```

如果条件值为x或z，则结果可能为x或z

3.4.7 移位操作符

移位操作符有：

<< (左移)

>> (右移)

◆ 移位操作符对其左边的操作数进行向左或向右的位移位操作。

◆第二个操作数（移位位数）是无符号数。

如 **rega << 5** 意思是**rega**向左移**5**位；

rega >> 3 意思是**rega**向右移**3**位；

>>>表示算术右移

3.4.8 连接操作

连接操作是将小表达式合并形成大表达式的操作。形式如下：

{expr1, expr2, . . . , exprN}

连接操作实例

```
wire   [9:0]    d;
wire   [9:0]    a;
assign  d[9:5] =
            {a[0], a[1], a[2],  a[3], a[4] } ;
```

//以反转的顺序将**a**的低端**4**位赋给**d**的高端**4**位。

```
assign  d= {d[4:0], d[9:5] } ;
```

//高**4**位与低**4**位交换。

空白符与注释

```
module MUX2_1 (out, a, b, sel);
 // Port declarations
   output out;
    input  sel; // control input
    input     b, a;  /* data inputs */
    wire  sel_, a1, b1;
/*The netlist logic selects input "a" when
 sel = 0 and it selects "b" when sel = 1. */
   not  not1 (sel_, sel);
   and  and1 (a1, a, sel_);
   and  and2 (b1, b, sel);
   or    or1 (out, a1, b1);
endmodule
```

单行注释到行末结束

多行注释，在/* */内

格式自由

使用空白符提高可读性及代码组织。**Verilog**忽略空白符除非用于分开其它的语言标记。

8.4.2 阻塞性过程赋值

赋值操作符是“=”的过程赋值是阻塞性过程赋值。例如，

```
RegA = 52;
```

是阻塞性过程赋值。阻塞性过程赋值在其后所有语句执行前执行，即在下一语句执行前该赋值语句完成执行。如下所示：

```
always
  @(A or B or Cin)
   begin: CARRY_OUT
    reg T1,T2,T3;

    T1 = A & B;
    T2 = B & Cin;
    T3 = A & Cin;
    Cout = T1 | T2 | T3;
end
```

*T1*赋值首先发生，计算 *T1*；接着执行第二条语句， *T2*被赋值；然后执行第三条语句， *T3*被赋值；依此类推。

非阻塞性过程赋值

- 在非阻塞性过程赋值中，使用赋值符号“<=”。例如：

```
initial
  begin
    Clr <= #5 1;
    Clr <= #4 0;
    Clr <= #10 0;
end
```

第一条语句的执行使*Clr* 在第5个时间单位被赋于值1；第二条语句的执行使*Clr*在第4个时间单位被赋值为 0(从0时刻开始的第 4个时间单位)；最终，第3条语句的执行使*Clr*在第10个时间单位被赋值为0(从0时刻开始的第10个时间单位)。注意3条语句都是在0时刻执行的。此外，在这种情况下，非阻塞性赋值执行次序变得彼此不相关。*Clr*上产生的波形如图8-7所示。

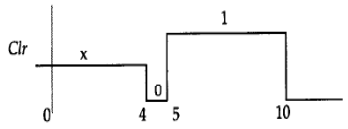


图8-7 带有语句内部时延的非阻塞性过程赋值

8.4.4 连续赋值与过程赋值的比较

连续赋值与过程赋值有什么不同之处？表8-1列举了它们的差异。

表8-1 过程赋值与连续赋值间的差异

过程赋值	连续赋值
在always语句或initial语句内出现	在一个模块内出现
执行与周围其它语句有关	与其它语句并行执行；在右端操作数的值发生变化时执行
驱动寄存器	驱动线网
使用“=”或“=”赋值符号	使用“=”赋值符号
无assign关键词(在过程性连续赋值中除外，参见第8章第8节)	有assign关键词

下例进一步解释了这些差别。

```
module Procedural;
reg A,B,Z;

always
  @(B) begin
    Z = A;
    A = B;
  end
endmodule
```

```
module Continuous
  wire A,B,Z;

  assign Z = A;
  assign A = B;
endmodule
```

假定B在10 ns时有一个事件。在过程性赋值模块中，两条过程语句被依序执行，A在10 ns时得到B的新值。Z没有得到B的值，因为赋值给Z发生在赋值给A之前。在连续性赋值语句模块中，第二个连续赋值被触发，因为这里有一个关于B的事件。这引起了关于A的事件，A引发第一个连续赋值被执行，这相应引起Z得到了A的值。Z的新值为A而不是B。然而，如果事件发生在A上，过程性模块中的always语句不执行，因为A不在那个always语句的实时控制事件清单中。然而连续赋值语句中的第一个连续赋值执行，并且Z得到A的新值。

8.5 if 语句

if语句的语法如下：

```
if(condition_1)
  procedural_statement_1
{else if(condition_2)
  procedural_statement_2}
{else
  procedural_statement_3}
```

```
if(Sum < 60)
  begin
    Grade = C;
    Total_C = Total _c + 1;
  end
else if(Sum < 75)
  begin
    Grade = B;
    Total_B = Total_B + 1;
  end
else
  begin
    Grade = A;
    Total_A = Total_A + 1;
  end
```

8.6 case语句

case语句是一个多路条件分支形式，其语法如下：

```
case(case_expr)
  case_item_expr{,case_item_expr}:procedural_statement
. . .
. . .
  [default:procedural_statement]
endcase
```

```
module ALU (A, B, OpCode, Z);
  input [3:0] A, B;
  input [1:2] OpCode;
  output [7:0] Z;
  reg [7:0] Z;
parameter
    ADD_INSTR = 2'b10,
    SUB_INSTR = 2'b11,
    MULT_INSTR = 2'b01,
    DIV_INSTR = 2'b00;
always
  @ (A or B or OpCode)
    case (OpCode)
      ADD_INSTR:    Z = A + B;
      SUB_INSTR:    Z = A -B;
      MULT_INSTR:   Z = A * B;
      DIV_INSTR:    Z = A / B;
    endcase
endmodule
```


while 循环语句

- **while(BY>0)**
- **begin**
- *Acc = Acc << 1;*
- *By = By - 1;*
- **end**

for 循环语句

```
integer K;
for (K=0 ; K ＜MAXRANGE ; K = K + 1)
  begin
    if(Abus[K] == 0)
        Abus[K] = 1;
    else if(Abus[k] == 1)
        Abus[K] = 0;
    else
      $display( "Abus[K] is an x or a z");
end
```