



- | | | | |
|---|-------|-------------------------|-----------------|
| - | SUB : | $Y = A - B;$ | OP = "010"
; |
| - | A : | $Y = A;$ | OP = "011"
; |
| - | 或 : | $Y = A \text{ OR } B;$ | OP = "100"
; |
| - | 和 | $Y = A \text{ 和 } B;$ | OP = "101"
; |
| - | XOR : | $Y = A \text{ XOR } B;$ | OP = "110"
; |
| - | 不是 : | $Y = \text{NOT } A;$ | OP = "111"
; |

每次操作结束时：

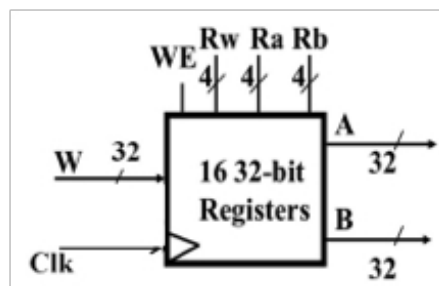
- 如果结果是严格负值，N 标志的值为 1，否则为 0。
- 如果结果为零，Z 标志的值为 1，否则为 0。
- 有 Carry 时，C 标志的值为 1，否则为 0。
- 当出现容量溢出（oVerflow）时，V 标志的值为 1，否则为 0。当两个符号相同的数字之间的算术运算结果符号相反时，就会出现溢出。

提示：ieee.numeric_std.all 库允许您使用 "+" 和 "-" 运算符。

注册银行

工作台有 16 个 32 位寄存器。它有以下输入/输出：

- CLK：时钟、
- **复位**：异步复位（高电平状态有效），图中未显示
- W：32 位写入数据总线
- RA：端口 A 4 位读取地址总线
- RB：端口 B 4 位读取地址总线
- RW：4 位写地址总线
- WE：1 位上的写入启用
- A：端口 A 读取数据总线
- B：端口 B 读取数据总线



以**组合和同步**方式**读取**寄存器：

- A 输出总线携带寄存器编号 RA 的值。

输出总线 B 保存寄存器编号 RB 的值

寄存器在时钟上升沿**同步写入**。它由 WE 信号控制：

- 如果 $WE = 1$ ，则在上升沿将 W 总线的值复制到寄存器 RW 中。

如果 $WE = 0$ ，则上升沿没有写入。

1.1 将开展的工作

1.1.1 用行为VHDL 描述和模拟这些模块

寄存器组将声明为 `std_logic_vectors` 数组。将使用一个函数对其进行初始化（见下面的代码）

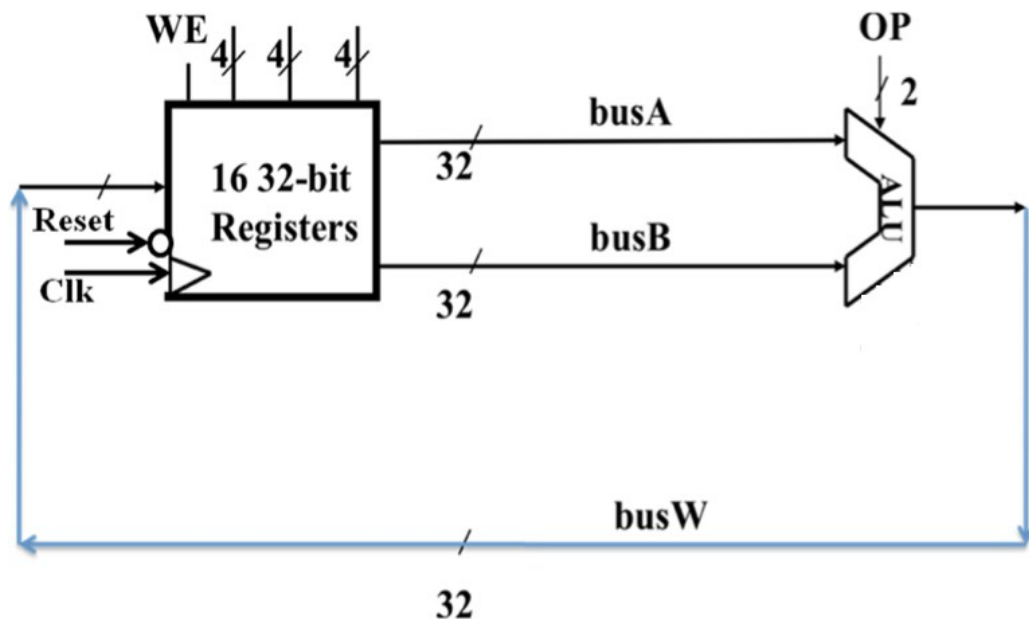
```
-- 声明内存表类型
类型表是 std_logic_vector(31 下至 0) 的数组 (15 下至 0) ;

-- 函数 init_banc 返回表为
variable result : table;
begin
    for i in 14 downto 0 loop
        result(i) := (others=>'0');
    结束循环;
    result(15) := X"00000030";
    返回结果;
end init_banc;

-- 16x32 位寄存器库的声明和初始化
信号银行: table:=init bank;
```

提示：必须避免数组索引变量为负整数、未初始化或大于数组大小。如果出现上述情况之一，模拟将以致命错误停止。

1.1.2 按下图所示组装 ALU 和工作台，以验证处理单元。



1.1.3 编写一个测试台，通过模拟验证下列操作的正确性：

- $R(1) = R(15)$
- $r(1) = r(1) + r(15)$
- $r(2) = r(1) + r(15)$
- $R(3) = R(1) - R(15)$
- $R(5) = R(7) - R(15)$

2 对 1 多路复用器

该多路复用器有一个通用参数 N，用于设置输入和输出数据的大小

- A、B：有 N 个输入位的两条总线
- COM：1 位选择命令
- S：N 位输出总线

COM 信号的值决定了输出的行为

标志扩展

COM	已执行的操作
0	$S = A$
1	$S = B$

该模块可将 N 位编码输入的符号扩展到 32 位输出。因此，它有一个设置 N 值的通用参数。

- E：N 个输入位上的总线
- S：32 位输出总线

数据存储器

该存储器可加载和存储 64 个 32 位字。它有以下输入和输出：

CLK：时钟

复位：异步复位（高电平状态有效），图中未显示。

数据输入：32 位写数据总线 **数据输出**：32 位读

数据总线 **Addr**：6 位读/写地址总线 **WrEn**：1 位

写入使能

以**组合和同步**方式**读取**寄存器：

数据输出（DataOut）输出总线携带数据值。

寄存器在时钟上升沿**同步写入**。它由 WE 信号控制：

如果 $WE = 1$ ，则在上升沿将**数据输入**总线的值复制到寄存器编号 $Addr$ 中 如果 WE

$= 0$ ，则在上升沿不进行写入操作

建议

- 该模块与寄存器库非常接近，可以在寄存器库的基础上进行设计。
- 由于来自 Alu 的信号为 32 位，因此可以在**端口映射级别**对 $Addr$ 的 6 位进行缩减（ $Addr \Rightarrow SignalAluOut(5 \text{ downto } 0), \dots$ ）。

1.2 将开展的工作

1.2.1 给出这些模块的框图

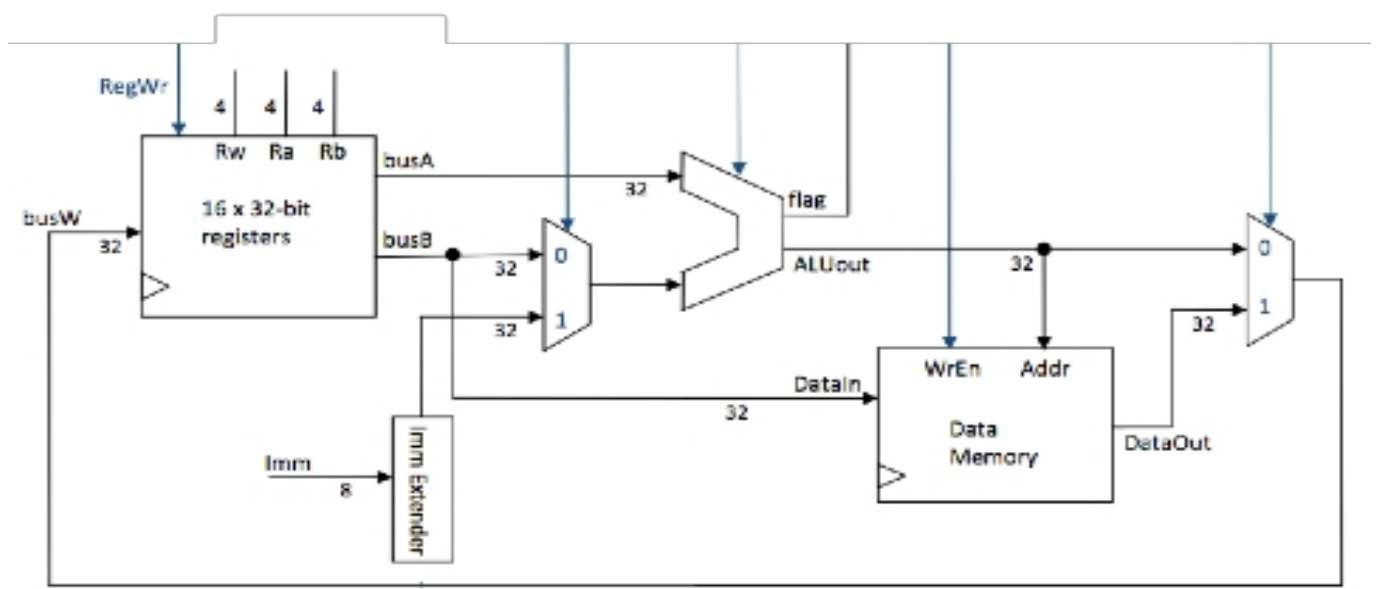
1.2.2 用行为 VHDL 描述和模拟这些模块

1.3 处理装置组件

上述模块必须按下图所示组装：

1.3.1 编写一个 VHDL 模块来组装处理单元。

1.3.2 编写一个测试平台，以验证.NET Framework 3.0 的正确运行：



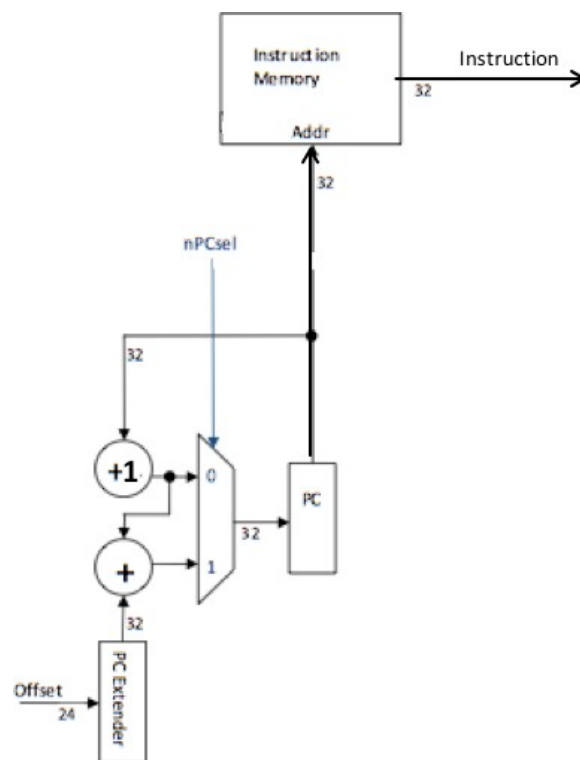
- 增加 2 个寄存器
- 将 1 个寄存器的数值立即相加
- 减去 2 个寄存器

- 从 1 个寄存器中减去 1 个直接值
- 将数值从一个寄存器复制到另一个寄存器
- 将寄存器写入内存中的一个字。
- 将一个字从内存读入寄存器

第 2 部分--指令管理股

下图所示的 32 位指令管理单元具有：

- 与处理单元类似的 64 字 32 位指令存储器，作为 `instruction_memory.vhd` 文件的附录提供。
- 没有写入数据总线，也没有写入使能。
- 32 位寄存器（PC 寄存器）。该寄存器有一个时钟和一个异步复位（高电平时有效），图中未显示。
- 与上述模块类似的 24 至 32 位带符号扩展单元
- 根据 `nPCsel` 控制信号更新 PC 程序计数器的装置：
 - 如果 `nPCsel = 0`，则 $PC = PC + 1$
 - 如果 `nPCsel = 1`，则 $PC = PC + 1 + \text{SignExt}(\text{偏移})$



您可以以完全结构化的方式描述该组件，也可以选择对生成新地址的部分使用行为描述。

2.1 描述并模拟指令管理单元

第 3 部分 - 控制单元

控制单元由一个 32 位寄存器和一个组合解码器组成。

带加载命令的 32 位寄存器

该寄存器用于存储处理器的状态（处理器状态寄存器或 PSR），并用于在 STR 指令期间在 RegAff 上存储 LED 或 7 段显示器上显示的值。

它有以下输入/输出：

- 数据输入：32 位负载总线
- RST：异步复位，高电平状态有效
- CLK 时钟
- WE：负载控制
- 数据输出：32 位输出总线

如果 WE=1，寄存器存储 DATAIN 总线上的值。如果 WE=0，寄存器保留先前的值（存储状态）。

指令解码器

这个组合模块为处理单元、指令管理单元和 PSR 寄存器生成控制信号，所有这些都已在上文说明。

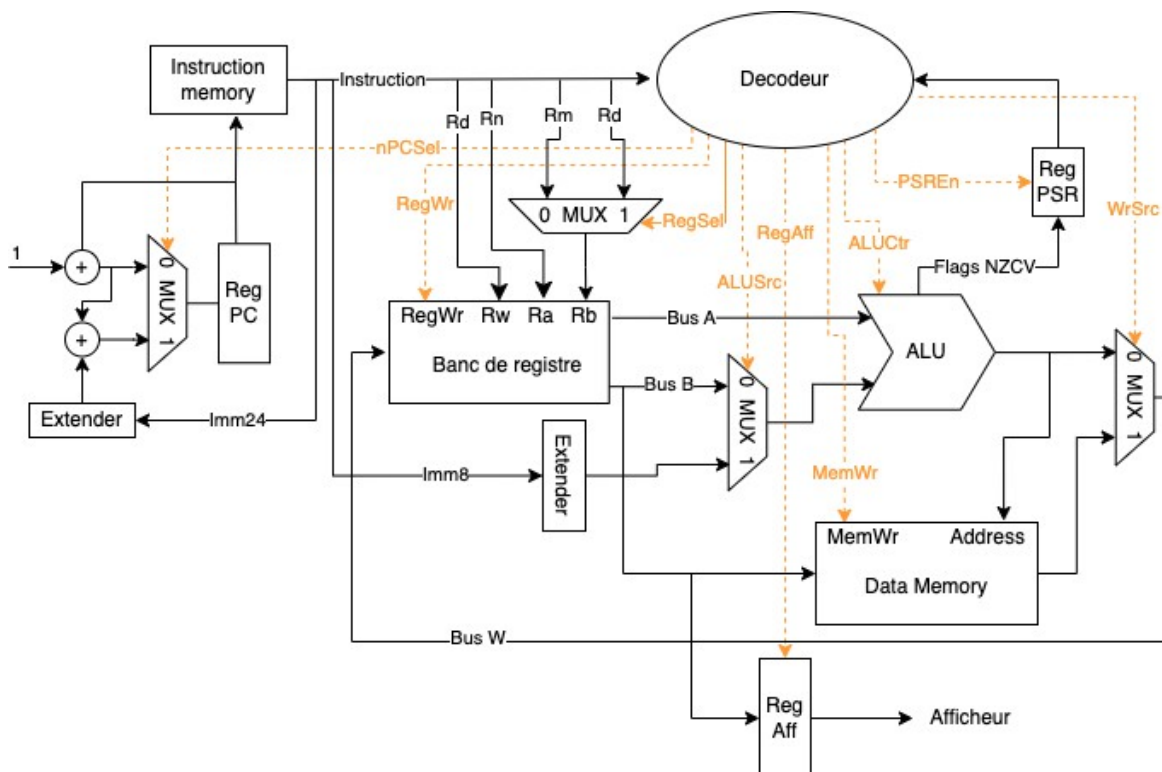
这些指令的值取决于从指令存储器获取的指令，也可能取决于 PSR 寄存器的状态。附录中描述了各种指令的二进制结构。

将生成的订单涉及：

- PC 寄存器输入多路复用器 (**nPC_SEL**)
- PSR 寄存器加载命令 (**PSREn**)
- 地址总线 Rn、Rm、Rd 和寄存器组 **WE (RegWr)**
- 位于该工作台 RB 地址总线上游的多路复用器的 **RegSel** 命令。
- ALU OP 命令 (**ALUCtrl**)

- 多路复用器指令 (MemToReg)，用于选择要写入寄存器组的信号。
- 来自 ALU 输入端 B 上游多路复用器的 **ALUSrc** 命令。
- ALU 和数据存储器输出端多路复用器的 WSrc 命令。
- 数据存储器的写入功能 (**MemWr**)
- 用于在 STR 运行期间显示输出值的信号 (**RegAff**)

处理器总图：



该解码器的设计将基于上图所示的处理器总图、控制面板（见下页）和附录中的指令集说明。

为了描述这个指令解码器，我们首先声明以下枚举类型：

枚举指令类型为 (MOV、ADDi、ADDr、CMP、LDR、STR、BAL、BLT) ；
信号 `instr_current`：枚举指令；

然后，在架构中可以使用两个流程

- 对指令存储器（附录图表中的指令）输出敏感的进程，其作用是设置 `instr_current` 信号的值。
- 对指令和 `instr_current` 信号进行敏感处理，将指令值传递给处理器寄存器和运算器。

3.1 完成下表中的 "指令值"，该表概括了解码器根据指令进行的操作。

3.2 用 VHDL 描述控制单元的两个模块：指令解码器和 32 位 PSR 寄存器，如果收到加载指令，将在 4 个最有效位上获取 ALU 的 N、C、Z 和 V 标志。

订购值

教学	nPCSel	RegWr	ALUSrc	ALUCtr	PSREn	MemWr	WrSrc	RegSel	RegAff
ADDi									
地址									
BAL									
BLT									
中医									
LDR									
MOV									
STR									

第 4 部分 - 组装和验证处理器

现在，我们需要将处理器的三个主要单元组装起来，最终完成处理器的建模工作。

- 指令管理单元
- 处理装置
- 控制单元

4.1 添加一个由控制单元产生的 RegSel 控制信号驱动的 2 输入 4 位多路复用器，完成之前设计的处理单元。如处理器图所示，该多路复用器将被置于寄存器组 RB 地址的输入端。

4.2 将处理器的三个单元组装起来。

4.3 模拟处理器执行测试程序并检查其工作是否正常。可能需要初始化数据存储器中的某些数据（从 0x20 到 0x2A）。

测试计划

```

0x0 _main : MOV R1,#0x20          ; --R1 <= 0x20
          0x1MOV R2,#0           ; --R2 <= 0
0x2 _loop : LDR R0,0(R1)          ; --R0 <= DATA_MEM[R1]
          0x3ADD R2,R2,R0        ; --R2 <= R2 + R0
          0x4ADD R1,R1,#1        ; --R1 <= R1 + 1
          0x5CMP R1,0x2A         ; -- R1 - 0x2A, 更新 N
0x6      BLT loop               --connect to _loop if R1 less than 0x1A 0x7
_end: str r2,0(r1)              --data_mem[r1] <= r2
          0x8BAL main           ; -- 连接到 _main

```

第 5 部分：测试 Fpga 卡上的处理器

在本节中，您将在 FPGA 板上实现独轮车处理器。

要检查测试程序是否运行无误，可在 7 段显示器上显示 **Reg Aff** 寄存器输出端的值。

为此，我们需要增加 4 个 7 段解码器，对 **Reg Aff** 寄存器输出的第 0 至 15 位进行解码，以显示 4 位十六进制数值。

将开展的工作

5.1 在 Quartus 上创建一个项目，在 **top_level.vhd** 文件中实例化处理器和 4 个 7 段解码器。分配引脚、运行综合并对电路板编程。

5.2 检查是否在 7 段显示屏上找到预期值。

第 6 部分：管理外部中断

要管理外部中断，需要添加一个矢量化中断控制器，通常称为 VIC（矢量中断控制器）。

指令解码器也必须进行修改，以便在指令子程序结束时解码 BX 指令，表示必须返回被中断的程序。

接下来，需要修改指令管理单元，以便将中断考虑在内。

6.1 VIC：矢量化中断控制器

参赛作品：

- CLK：时钟
- 复位：异步复位，高电平状态有效
- IRQ_SERV：指令管理单元发出的中断确认信息
- IRQ0、IRQ1：外部中断请求

外出：

- IRQ：向指令管理单元发送的中断请求
- VICPC：中断子程序的起始地址（32 位）

工作原理：

- 该模块对 IRQ0 和 IRQ1 的值进行采样，以找出这两个信号的最后（IRQi(n)）和倒数第二（IRQi(n-1)）个值。
- 如果在其中一个信号上检测到上升转换（IRQi(n)=1 和 IRQi(n-1)=0），IRQ0_memo 或 IRQ1_memo 信号将被强制置高，发出中断请求信号。
- IRQ1_memo 和 IRQ0_memo 始终处于高电平状态，直到 IRQ_SERV 输入收到确认。
- 如果没有中断请求，VICPC 将被强制置 0。
- 如果 IRQ0 = 1，则强制 VICPC 输出值为 0x9（该中断子程序的起始地址）。
- 如果 IRQ1_memo = 1，VICPC 输出将被强制设置为 0x15（该中断子程序的起始地址）。

- IRQ0 优先于 IRQ1。
- 发送到 MAE 的 IRQ 输出是 IRQ1_memo 和 IRQ0_memo 之间的逻辑 OR。

发送给指令管理单元的信号：

IRQ

指令管理单元生成的指令：

IRQ_SERV.

要做

6.1.1 创建一个新的 VHDL 源来描述 VIC。

6.1.2 使用测试台和模拟脚本模拟该组件。

6.2 修改指令解码器

BX 汇编指令用于管理中断子程序的退出。该指令用于退出中断子程序并返回主程序。

为此，必须在解码器单元中添加输出控制信号：

- **IRQ_END**：执行 BX 指令时设置为 "1"。
- 下面是 BX 指令的二进制代码：0xEB00 0000；

6.3 修改指令管理单元

必须接管指令管理单元，并将以下信号添加到其实体中：

入场：

- **IRQ**：如果激活，表示由 VIC 产生新的中断。
- **VICPC**：传送中断子程序的地址（向量）。
- **IRQ_END**：如果激活，表示中断子程序结束。

输出：

- **IRQ_SERV**：中断处理后的确认。

工作原理：

- 当中断激活（IRQ = 1）时，必须将 VICPC 值复制到 PC 中，同时将 PC 寄存器保存到名为 LR（链接寄存器）的寄存器中。
- 当中断被确认时，将 IRQ_SERV 信号设置为 "1"，否则在其余时间将其设置为 "0"。
- 当 IRQ_END 信号激活时，将 PC 设置为 LR + 1，以便主程序继续运行。

- 您需要注意信号发出的时间，以便每条指令只执行一次！

6.4 在 FPGA 上测试 VIC

在 FPGA 上使用 VIC 实现处理器，并检查其工作是否正常。

第 7 部分：为处理器添加 UART 外设

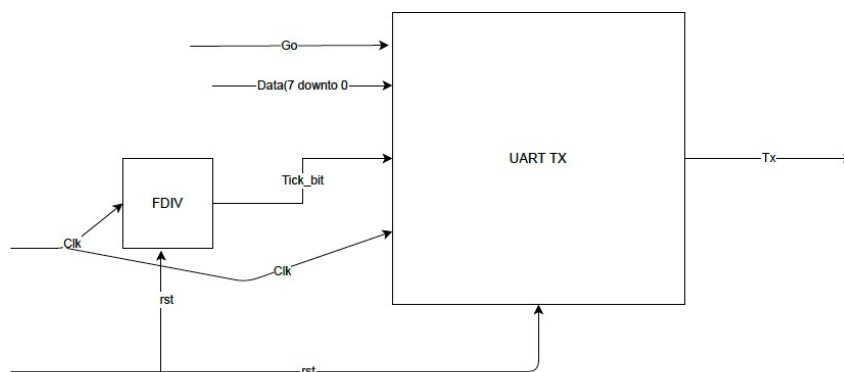
在本节中，我们将了解如何为处理器添加 UART 外设。首先将介绍 UART 的操作部分，然后了解如何将其连接到处理器。

要测试该 UART，需要执行一小段汇编代码，向 PC 终端发送 "HELLO WORLD!" 短语。

。

UART 的传输部分：

下面是 UART 传输部分的示意图和 MAE：

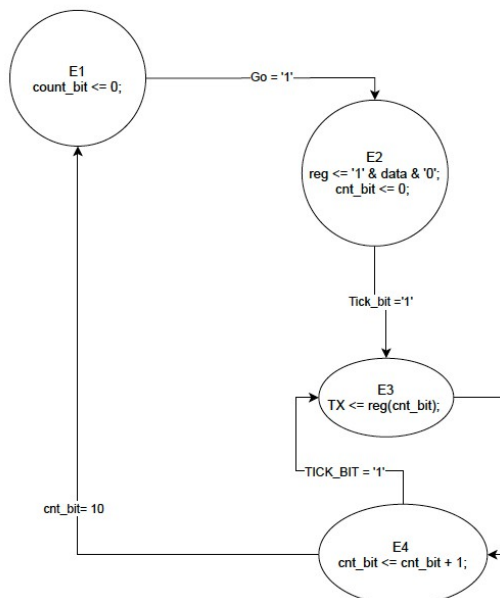


```

signal reg : std_logic_vector(9 downto 0);
signal cnt_bit : integer range 0 to 15;
  
```

```

Reset :
  Tx <= '1';
  cnt_bit <= 0;
  reg <= (others => '0');
  
```

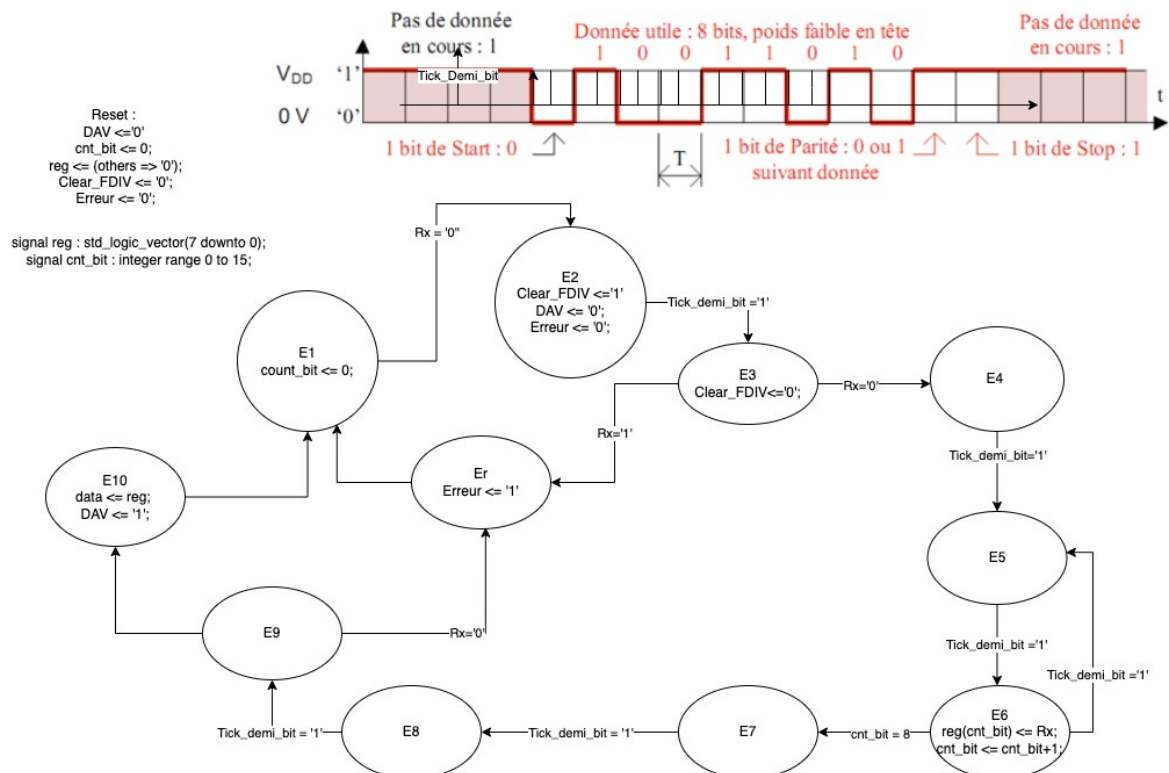
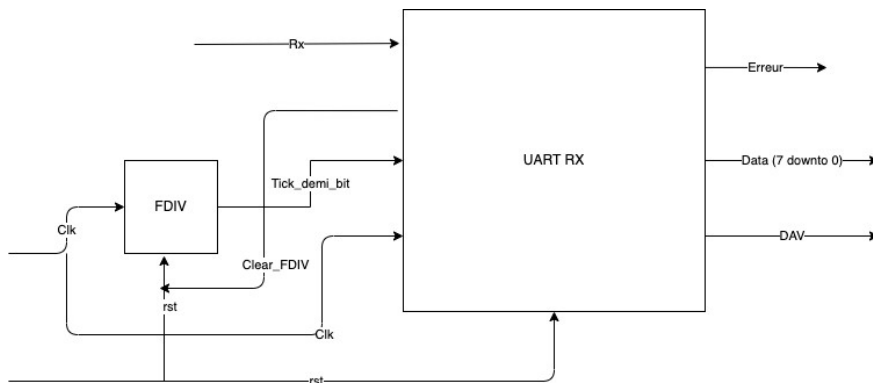


要做

- 1) 创建一个新的 VHDL 源，以描述 UART 的发送器部分
- 2) 使用测试台和模拟脚本模拟该组件。
- 3) 将 Tx 输出连接到 GPIO 端口的一个引脚（例如 GPIO(0)），将数据输入连接到 SW(7.0)，将 Go 输入连接到 KEY(1)，从而测试 FPGA 板上 UART 的工作部分。然后使用串行/USB 转换器在 PC 终端（Windows 上的 Puty 或 Linux 和 Mac 上的 Screen）上接收字符。将 Tx 输出（GPIO(0)）连接到串行/USB 转换器的 Rx 引脚（黄色）。
- 4) 为外设添加一个 32 位 UART_Conf 寄存器。每次向寄存器写入数据时，必须将 Go 设置为 1。使用地址为 0x40 的 STR 指令，必须能够写入 UART_Conf 寄存器。为此，您需要执行地址解码，只有当地址等于 0x40 时，才能写入寄存器。
- 5) 例如，在每次输入 IRQ1 时发送 0x31（ASCII 码中的 "1"）值，利用这一新外设模拟处理器。
- 6) 使用 DE10-Lite 卡上的 UART 外设测试处理器。要与电脑通信，您需要在电脑上使用串行/USB 转换器和终端（Windows 上的 Puty 或 Mac 上的 Screen）。
- 7) 发送一个字符需要一些时间。波特率为 115200 时，需要等待大约 87 秒。您可以使用中断机制来处理这个问题。当发送完一个字符后，可以产生一个中断，并在中断子程序中发送下一个字符。为此，您需要在字符发送结束时在 UART 中产生一个 TxIrq 信号。您还需要更改 VIC 以处理这一新的中断源。
- 8) 修改 UART 设备和 VIC，使其可以通过中断进行管理。修改要执行的汇编代码，以便向中断子程序发送 HELLO WORLD 字符串。在 FPGA 板上进行模拟和测试。

UART 的接收部分：

以下图和 MAE 为指导，对接收端进行同样的操作。



附录：

第 4 部分的汇编代码：instruction_memory.vhd

```

IEEE 图书馆;
使用 IEEE.std_logic_1164.all;
使用 IEEE.numeric_std.all;

实体 instruction_memory 是
    port(
        PC: std_logic_vector (31 下至 0) 中;
        指令: out std_logic_vector (31 downto 0)
    );
终端实体;

指令存储器的架构 RTL 为

    RAM64x32 类型是 std_logic_vector 的数组 (0 至 63) (31 至 0) ;

函数 init_mem 返回 RAM64x32 变量 result
    : RAM64x32;
兴办
for i in 63 downto 0 loop
    result (i):=(others=>'0');
end loop;
-- PC --INSTRUCTION -- COMMENT
result (0):=x "E3A01020";-- 0x0 _main -- MOV R1,#0x20 -- R1 = 0x20
result (1):=x "E3A02000";-- 0x1 -- MOV R2,#0x00 -- R2 = 0
result (2):=x "E6110000";-- 0x2 _loop -- LDR R0,0(R1) -- R0 =[R1] result
(3):=x "E0822000";-- 0x3 -- ADD R2,R2,R0 -- R2 = R2 + R0 result
(4):=x "E2811001";-- 0x4 -- ADD R1,R1,#1
result (5):=x "E351002A";-- 0x5 --CMP R1,0x2A -- Flag = R1-0x2A, si R1 <=result
(6):=x "BAFFFFFFB";-- 0x6 -- BLT 循环 -- PC =PC+1+(-5) si N = 1
result (7):=x "E6012000";-- 0x7 --STR R2,0(R1) -- DATAMEM[R1] =R2
result (8):=x "EAF7FFFF";-- 0x8 -- BAL main -- PC=PC+1+(-9)
    return result;
end init_mem;

信号 mem: RAM64x32 := init_mem; begin
    指令 <= mem(to_integer(unsigned (PC))); 结束架构;

```


第 6 部分的汇编代码：

instruction_memory_IRQ.vhd

IEEE 图书馆;

使用 IEEE.std_logic_1164.all;

使用 IEEE.numeric_std.all;

实体 instruction_memory_IRQ 为 port(

 PC: std_logic_vector (31 下至 0) 中;

 指令: out std_logic_vector (31 downto 0)

);

终端实体;

指令_内存_IRQ 的架构 RTL 为

 RAM64x32 类型是 std_logic_vector 的数组 (0 至 63) (31 至 0) ;

函数 init_mem 返回 RAM64x32 变量

 ram_block : RAM64x32;

兴办

```

--          PC -- 指令 -- 注释
ram_block(0 ) := x "E3A01010";-- _main :    MOV R1,#0x10    --R1 <=
0x10ram_block(1 ) := x  " E3A02000";        MOV R2,#0      ;--R2 <=

Oram_block(2 ) := x "E6110000";-- _loop :    LDR R0,0(R1)    --R0 <= MEM[R1]
ram_block(3 ) := x "E0822000";--            ADD R2,R2,R0    --R2 <= R2 + R0
ram_block(4 ) := x "E2811001";--            ADD R1,R1,#1     ; --R1 <= R1 +
1ram_block(5 ) := x "E351001A";--            R1,0x1A      ?R1 = 0x1A

ram_block(6 ) := x "BAFFFFFFB";--            BLT 循环    如果 R1 小于 0x1A 则连接到 _loop ram_block(7 )
:= x "E6012000";--                        STR R2,0(R1)    --MEM[R1] <= R2
ram_block(8 ) := x "EAFFFFFF7";--            --BAL main    ; --connect to _main

-- ISR 0: 中断 0

--保存上下文

ram_block(9 ) := x "E60F1000" --            STR R1,0(R15)        ; --mem[r15] <= r1
;
ram_block(10) := x "E28FF001" --            ADD R15,R15,1    ; --r15 <= r15 + 1
;
ram_block(11) := x "E60F3000" --            STR R3,0(R15)        ; --mem[r15] <= r3
;

--处理

ram_block(12) := x "E3A03010" --            MOV R3,0x10        ; --R3 <= 0x10
;

```

```

ram_block(13) := x "E6131000" --      LDR R1,0(R3)      ; --r1 <= mem[r3]
                ;
ram_block(14) := x "E2811001" --      ADD R1,R1,1      ; --r1 <= r1 + 1
                ;
ram_block(15) := x "E6031000" --      STR R1,0(R3)      ; --mem[r3] <= r1
                ;

-- 恢复背景
ram_block(16) := x "E61F3000";--      R3,0(R15);      --R3      <=      MEM[R15]
ram_block(17) := x "E28FF0FF";--      ADDR15,R15,-1 ; --R15 <= R15 - 1
ram_block(18) := x "E61F1000";      --LDR R1,0(R15) ; --R1 <= MEM[R15]
ram_block(19) := x "EB000000"; --BX      ; -- 中断指令结束 ram_block(20) := x "00000000";

-- ISR1: 中断 1

--保存上下文 - R15 与堆栈指针相对应
ram_block(21) := x "E60F4000"; --      STR R4,0(R15)      ; --mem[r15] <= r4
ram_block(22) := x "E28FF001"; --      ADD R15,R15,1      ; --r15 <= r15 + 1

```

```

ram_block(23) := x "E60F5000";           --STR R5,0(R15) ; --MEM[R15] <= R5
                                           --处理
ram_block(24) := x "E3A05010"; --      MOV R5,0x10           ; --R5 <= 0x10
ram_block(25) := x "E6154000"; --      LDR R4,0(R5)           ; --r4 <= mem[r5]
ram_block(26) := x "E2844002"; --      ADD R4,R4,2           ; --r4 <= r1 + 2
ram_block(27) := x "E6054000"; --      STR R4,0(R5)           ; --mem[r5] <= r4
                                           -- 恢复背景
ram_block(28) := x "E61F5000";--      ldr r5,0(r15)         ; --r5 <= mem[r15]
ram_block(29) := x "E28FF0FF"; --      add r15,r15,-1        ; --r15 <= r15 - 1
ram_block(30) := x "E61F4000"; --      ldr r4,0(r15)         ; --r4 <= mem[r15]
ram_block(31) := x "EB000000";--      BX                     ; -- 中断结束指令
ram_block(32) := x "00000001";
ram_block(33) := x "00000002";
ram_block(34) := x "00000003";
ram_block(35) := x "00000004";
ram_block(36) := x "00000005";
ram_block(37) := x "00000006";
ram_block(38) := x "00000007";
ram_block(39) := x "00000008";
ram_block(40) := x "00000009";
ram_block(41) := x "0000000A";

ram_block(42 至 63) := (others=> x "00000000");

return ram_block;

end init_mem;

signal mem: RAM64x32 := init_mem;

兴办

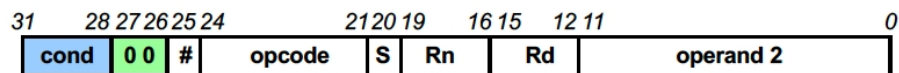
      指令 <= mem(to_integer (unsigned (PC)));

终端结构;

```

处理指令的二进制编码：

■ Instructions de traitement (1)



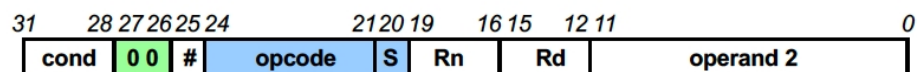
- *Cond*: l'instruction est exécutée si le registre d'état CPSR vérifie la condition spécifiée

Asm	Cond
EQ	0000
NE	0001
CS/HS	0010
CC/LO	0011
MI	0100
PL	0101

Asm	Cond
VS	0110
VC	0111
HI	1000
LS	1001
GE	1010
LT	1011

Asm	Cond
GT	1100
LE	1101
AL	1110
NV	1111

■ Instructions de traitement (2)



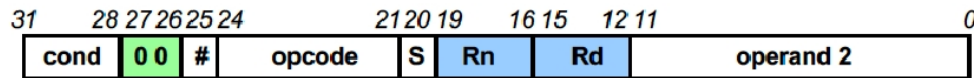
- *S = 1* : affecte CPSR
- *Opcode* : code de l'opération à effectuer

Asm	Opcode
AND	0000
EOR	0001
SUB	0010
RSB	0011
ADD	0100
ADC	0101

Asm	Opcode
SBC	0110
RSC	0111
TST	1000
TEQ	1001
CMP	1010
CMN	1011

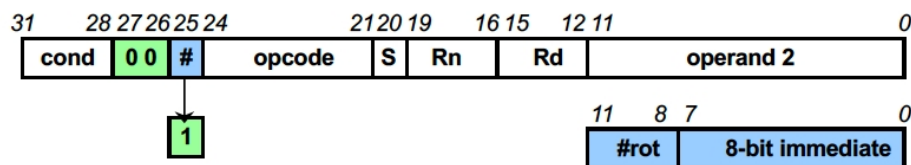
Asm	Opcode
ORR	1100
MOV	1101
BIC	1110
MVN	1111

■ Instructions de traitement (3)



- $Rd \rightarrow$ numéro du registre de destinations
 - 4 bits pour sélection parmi les 16 registres possibles
- $Rn \rightarrow$ numéro du registre qui sert de premier opérande
 - 4 bits pour sélection parmi les 16 registres possibles
- $operand2 \rightarrow$ relié à l'entrée B
 - possibilité de rotation/décalage

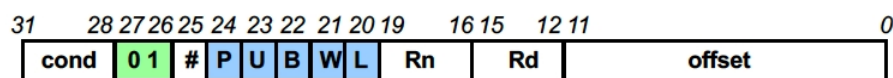
■ Instructions de traitement (4)



- Si le bit # est à 1, $operand2$ est une valeur littérale sur 32 bits.
 - Une instruction est codée sur 32 bits, il n'est donc pas possible de coder n'importe quelle valeur littérale 32 bits.
 - Cette valeur est construite par rotation vers la droite d'une valeur sur 8 bits.

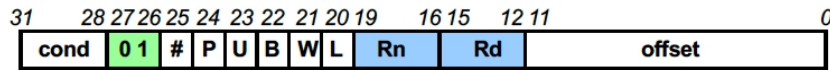
- Instructions de transfert
 - Lecture/écriture
 - Données accédées
 - Mots, demi-mots, octets
 - Signées/non signées
 - Mode d'accès (pré/post indexé, +/- offset, write back)
 - Transfert simple/multiple
 - Registre source/destination, liste de registres
 - Registre de base
 - Offset
 - Condition
 - Exécution conditionnelle d'un transfert

- Instructions de transfert de mots ou d'octets non-signés (LDR, STR, LDRB, STRB)

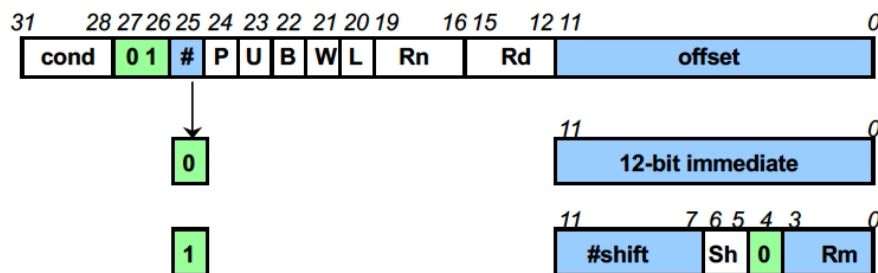


- *P* : pre/post index
 - 1 pré-indexé, 0 post-indexé
- *U* : up/down
 - 1 + offset, 0 -offset
- *B* : byte/word
 - M1 accès 8 bits, 0 accès 32 bits
- *W* : write-back
 - Si P=1, W=1 adressage pré-indexé automatique
- *L* : load/store
 - 1 load, 0 store

- Instructions de transfert de mots ou d'octets non-signés (LDR, STR, LDRB, STRB)



- $Rd \rightarrow$ registre source (si $L=0$, store) ou destination (si $L=1$, load)
- $Rn \rightarrow$ registre de base
- Instructions de transfert de mots ou d'octets non-signés (LDR, STR, LDRB, STRB)



- Offset: soit un littéral non signé sur 12 bits, soit un registre d'index (Rm) éventuellement décalé sur un nombre constant de bits ($\#shift$)

Instructions de branchement relatif

- Branch (B)
- Branch and Link (BL)



- Offset : déplacement signé sur 24 bits
- L : Link (0 branch, 1 branch and link)
- Effets:
 - B: $PC \leftarrow PC + \text{offset}$
 - BL: $r14 \leftarrow PC - 4$; $PC = PC + \text{offset}$
 - r14 : Link Register (contient l'adresse de retour)

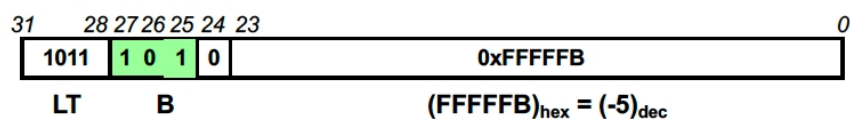
Instructions de branchement relatif

- Exemple traduction d'une boucle *for*
- Utilisation de labels en langage d'assemblage: le déplacement est calculé automatiquement par l'assembleur

```

MOV      r4, #0           @ tmp=0
MOV      r5, #0           @ i=0
loop:    ADD      r4, r4, r5   @ tmp+=i
         ADD      r5, r5, #1   @ i++
         CMP      r5, #5
         BLT      loop        @ i<5 : réitérer
         ...

```



Représentation binaire: BAFFFFFFB