西安电子科技大学
XIDIAN UNIVERSITY

基于 FPGA 的数字系统设计

实验报告

实验名称：PARWAN 设计综述及其 ALU/control section 内部状态仿真与分析

任课教师：沈沛意老师

学号姓名：

提交日期：

一、 PARWAN 简介

PARWAN CPU 是一个 8 位简单的微处理器，能完成最基本的处理器功能，包括取指令、分析指令、执行指令，能完成对 RAM 的读写功能和对外部 IO 的简单的控制功能。支持中断，能够以中断的方式与外围设备通信。

在 PARWAN 内部也支持简单的算术运算和逻辑运算，其拥有指令和数据寄存器，以支持简单的运算功能。有简单的指令集，包括算术运算指令、逻辑运算指令、跳转指令、子程序调用指令，并且有些指令有直接寻址和简介寻址两种寻址模式。同时还有一个标志寄存器，记录当前数据运算状态。

二、 总体设计

(一) 系统结构

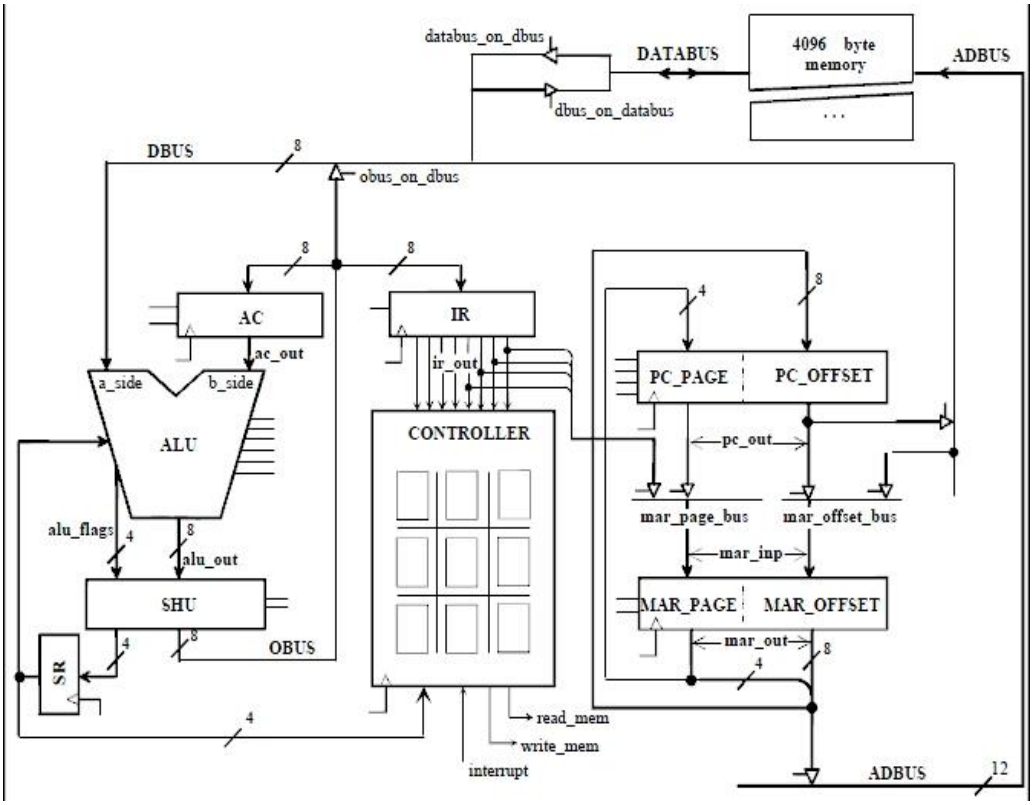

图 2- 1

从上图 2-1 可以看出，PARWAN 的内部结构主要由 ALU、移位单元 (SHU)、

控制器、累加器（AC)、指令寄存器（IR）等和 PC 控制部分组成。从 DBUS 或 AC 而来的的输入，在经过 ALU 算数逻辑单元运算后，结果放到移位寄存器中，移位寄存器可以对数据进行算术移位和逻辑移位操作，最后的运算结果可以放到累加器中、指令寄存器中或是总线内部总线上。

ALU、移位寄存器和标志寄存器构成了一个循环。ALU 读取标志寄存器中的数据，对数据计算，并输出更新后的标志信息；然后移位寄存器读取更新后的标志信息，同时对数据做移位，再一次更新标志信息，并将最终的标志信息保存到标志寄存器中。

IR 只是对读取到的指令数据寄存，以便于对指令译码，当有新的指令到来时，就更新指令寄存器的值。

CONTROLLER 负责解析 IR 中的指令，生成控制信号，通过状态机（9 个状态）协调各部件时序，解决内存访问等待，处理间接寻址等问题

PC（程序计数器）的 PC_PAGE（高 4 位）为指令页地址，PC_OFFSET（低 8 位）为页内偏移，当前指令地址通过 ADBUS 传给 MAR。MAR（内存地址寄存器）支持间接寻址，MAR_PAGE 来自 PC 页地址（PC_PAGE） 或 IR 低 4 位，MAR_OFFSET 来自 PC 偏移地址（PC_OFFSET） 或 DBUS 数据，MAR 向 ADBUS 输出当前访问的内存地址，查询 memory。

(二) 外部引脚设计

如下图 2-2

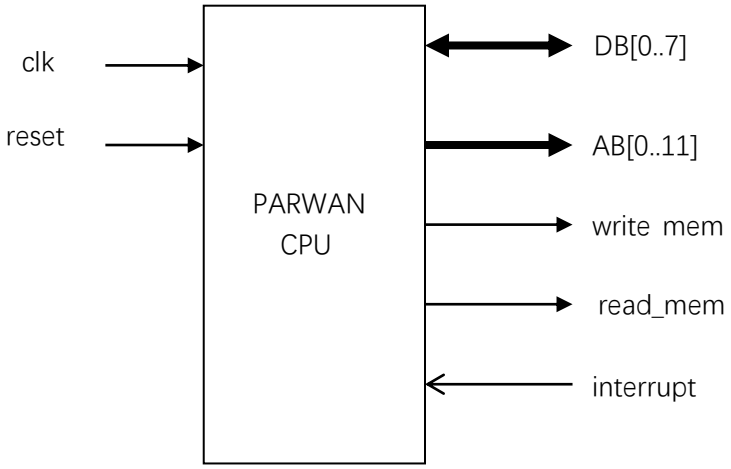

图 2- 2

其中 read_mem 和 write_mem 可以同时对内存和 IO 设备读写数据，clk 是外部输入的 CPU 工作时钟，reset 是复位信号，databus 时 8 位的数据总线，可输入、输出。adbus 是 12 位的地址总线，用于访问内存和 IO 设备用。interrupt 是外部中断信号输入，CPU 可检测外部中断，然后执行中断子程序。

从外部接口看，该 CPU 主要提供了两个功能：对内存的读写和利用中断的方式与外部 IO 通信。只要设计好时序，就可以和外围器件连接起来工作。

不足之处：中断信号只有检测引脚，没有响应信号，这样外设难以判断中断是否已被响应并执行了，并及时将中断有效信号复位，否则很容易产生二次中断的现象。

(三) 指令集设计

1. 指令集：

PARWAN CPU 的的地址线共有 12 位，将内存分为不同的页，其中的低 8 位表示页内地址，高 4 位表示页地址。每页有 256 个字节，共有 16 页。设计的 PARWAN 的指令集如下：

Instruction Mnemonic	Brief Description	Address Bits	Address Scheme	Indirect Address	Flags Use	Flags Set
LDA loc	Load AC w/(loc)	12	FULL	YES	----	--zn
AND loc	AND AC w/(loc)	12	FULL	YES	----	--zn
ADD loc	Add (loc) to AC	12	FULL	YES	-c--	vczn
SUB loc	Sub (loc) from AC	12	FULL	YES	-c--	vczn
JMP adr	Jump to adr	12	FULL	YES	----	----
STA loc	Store AC in loc	12	FULL	YES	----	----
JSR tos	Subroutine to tos	8	PAGE	NO	----	----
BRA_V adr	Branch to adr if V	8	PAGE	NO	v---	----
BRA_C adr	Branch to adr if C	8	PAGE	NO	-c--	----
BRA_Z adr	Branch to adr if Z	8	PAGE	NO	--z-	----
BRA_N adr	Branch to adr if N	8	PAGE	NO	---n	----
NOP	No operation	-	NONE	NO	----	----
CLA	Clear AC	-	NONE	NO	----	----
CMA	Complement AC	-	NONE	NO	----	--zn
CMC	Complement carry	-	NONE	NO	-c--	-c--
ASL	Arith shift left	-	NONE	NO	----	vczn
ASR	Arith shift right	-	NONE	NO	----	--zn

图 2- 3

按寻址方式将以上指令分为三类：（从各指令的 Address Bits 可以印证）

全地址指令：LDA、AND、ADD、SUB、JMP、STA。

页内地址指令：JSR、BRA_V、BRA_C、BRA_Z、BRA_N。

默认地址指令：NOP、CLA、CMA、CMC、ASL、ASR。

标志位：

N ： 符号位，标志当前的运算结果是否是一个负数；

V ： 溢出位，标志当前的运算结果是否产生了溢出；

C ： 进位位，标志当前的运算结果是否产生了进位（加法）或借位（减法）；

Z ： 非零位，标志当前的运算结果是否为零。

2. 指令集编码

下图 2-4 为指令集编码

Instruction Mnemonic	Opcode Bits 7 6 5	D/I Bit 4	Bits 3 2 1 0
LDA loc	0 0 0	0/1	Page adr
AND loc	0 0 1	0/1	Page adr
ADD loc	0 1 0	0/1	Page adr
SUB loc	0 1 1	0/1	Page adr
JMP adr	1 0 0	0/1	Page adr
STA loc	1 0 1	0/1	Page adr
JSR tos	1 1 0	-	- - - -
BRA_V adr	1 1 1	1	1 0 0 0
BRA_C adr	1 1 1	1	0 1 0 0
BRA_Z adr	1 1 1	1	0 0 1 0
BRA_N adr	1 1 1	1	0 0 0 1
NOP	1 1 1	0	0 0 0 0
CLA	1 1 1	0	0 0 0 1
CMA	1 1 1	0	0 0 1 0
CMC	1 1 1	0	0 1 0 0
ASL	1 1 1	0	1 0 0 0
ASR	1 1 1	0	1 0 0 1

图 2- 4

在编码时，按全地址指令、页内地址指令、默认地址指令三种指令分类，然后对每一类的指令分别编码。全地址指令占用两个字节，其中高字节的高三位为操作码，第四位为直接/简介寻址标识位，低十二位标识操作数的地址；页内地址指令占用两个字节，其中第八位表示页内地址，高八位标识操作码。默认地址指令占用一个字节，用八位标识操作码。

全地址指令和页内地址指令、默认地址指令的高三位不相同，后两种指令的高三位为全 1，而全地址指令不是，故可以区分开；而页内地址指令和默认地址指令的高四位不同，业内地址指令为 1111，默认地址指令为 1110，也可以区分开来。而 JSR 的高三位为 110，也可以和其他的指令区分开来。故可以先根据第一个字节的高三位判断是全地址指令、JSR 还是其他指令，再根据第四位可以判断出为页内地址指令还是默认地址指令。

(四) 层次化架构图

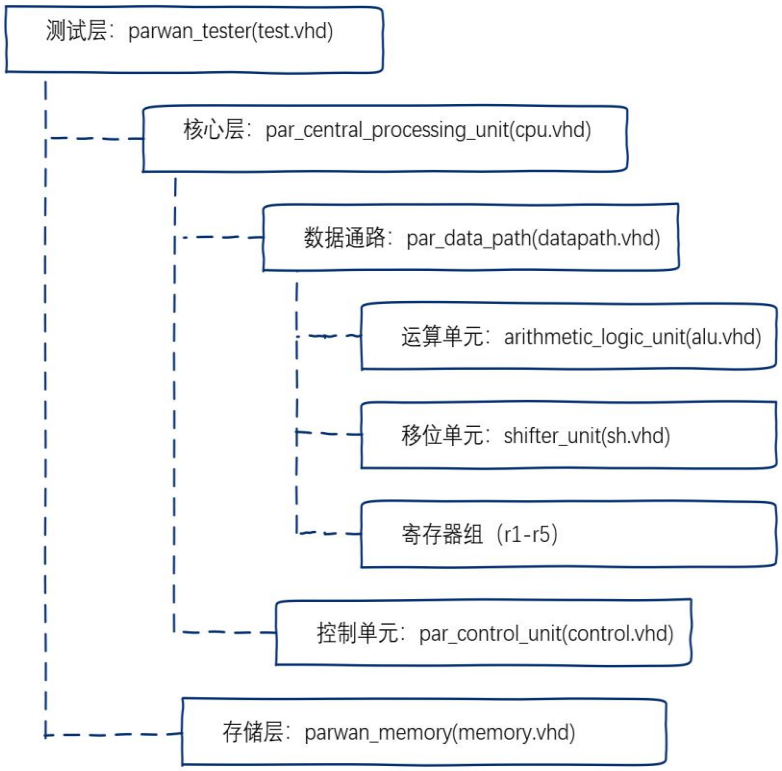

图 2- 5

测试层：

parwan_tester(test.vhd)：模拟时钟/中断信号，监控 CPU 运行状态；通过 wait_state 参数测试不同等待周期的影响

核心层：

par_central_processing_unit(cpu.vhd)：

在其子系统数据通路（par_data_path(datapath.vhd)）中存在三阶段处理：

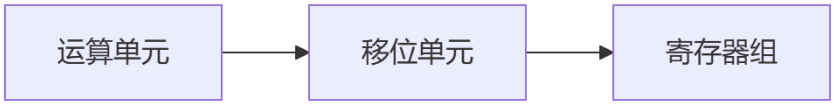

图 2- 6

另一子系统控制单元（par_control_unit(control.vhd)）：拥有 9 状态有限状态机，控制 58 个控制信号的生成

存储层：

parwan_memory(memory.vhd)：可加载外部程序，支持从 tmp.asb 文件初始化内存内容；通过 ready 信号实现与核心层的握手协议；

三、 核心模块分析

(一) ALU

ALU 在进行运算的过程中需要对标志位（n、z、c、v）进行更新，n 随着结果的最高位而变化；当结果全为 0 时把 z 置 1；加法溢出/减法借位/移位操作移出位为 1 时置 c 为 1；有符号运算溢出时把 v 置 1。

加减法影响 n、z、c、v；AND/OR/XOR/NOT 影响 n，z；移位/循环移位影响 n，z，c。

1. 外部接口分析

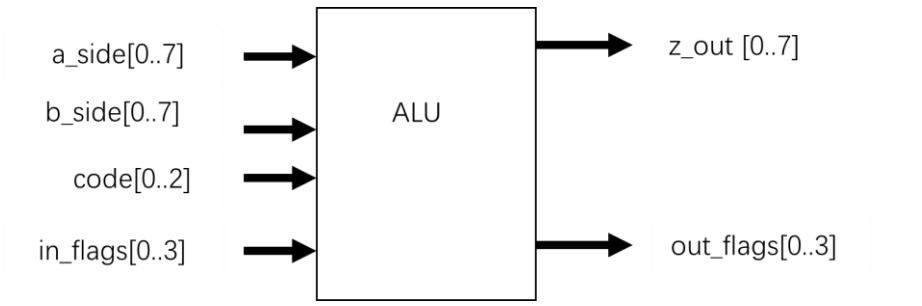

图 3- 1

在上图中，a_side 和 b_side 是 ALU 的两个输入运算数，都是八位的二进制数。code 是运算编码，每一种编码都对应着一种运算，用于控制 ALU 的运算类型。in_flags 是四位标志位，z_out 是运算的输出结果。out_flags 是更新后的标志位。

2. ALU 功能分析

ALU 的运算种类如下：

操作码	操作	功能
000	a_and_b	a 和 b 按位与
001	b_compl	b 按位取反
100	a_input	传输 a
101	a_add_b	a+b
110	b_input	传输 b
111	a_sub_b	a-b

图 3- 2

ALU 主要完成了两个八位二进制数的加、减和与运算。其中减法运算可以转换成补码的加法运算，而补码就是将原数的各位取反，然后在最低位加 1。

八位二进制加法器设计:

八位二进制的加法器可以由八个一位二进制的加法器串联而成，其中每一个一位加法器利用前一级的计算结果得出下一级的计算结果，如下图:

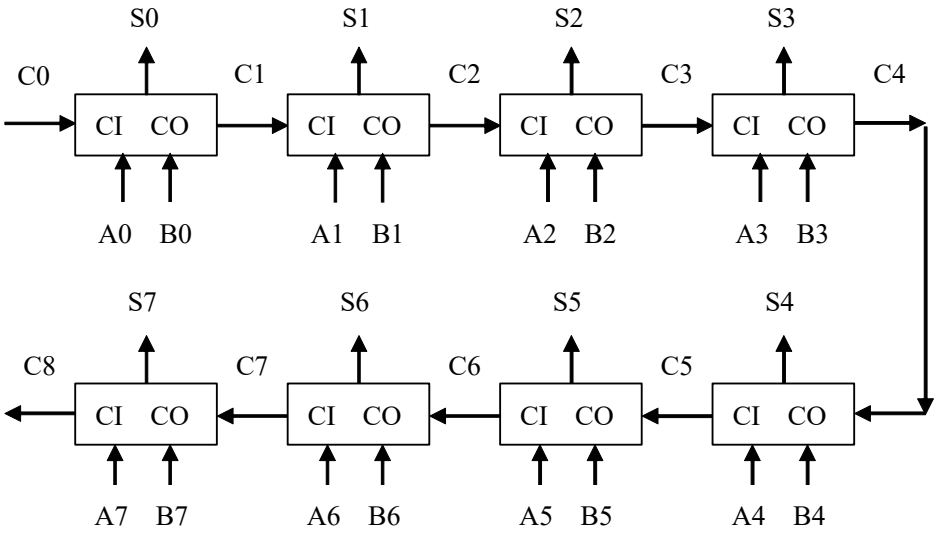

图 3- 3

如上图，八个一位加法器分别对八位的二进制数的每一位计算。第一个加法

器依据最初的进位对第零为求和并产生高位的进位，然后第二个加法器根据低位的进位对第二位计算，一次类推，知道计算出第八位的值。这样经过八次计算，就可以得出最终的计算结果。

一位加法器的逻辑表达式可以根据真值表的出来，如下：

S=A XOR B XOR CI

COUT =(A XOR B) CI + AB

这样一来，就可以完成八位二进制数的加法与减法运算。

3. ALU 代码解析

--所有操作不再继承输入标志位，强制使用新生成的标志位

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;              -- 标准逻辑库
USE work.synthesis_parameters.ALL;  -- 自定义参数包（包含字节/半字节定义）
USE work.synthesis_utilities.ALL;  -- 自定义工具函数
USE work.alu_operations.ALL;            -- ALU 操作码定义

ENTITY arithmetic_logic_unit IS
PORT (
    a_side      : IN byte;     -- 操作数 A（8 位）
    b_side      : IN byte;     -- 操作数 B（8 位）
    code        : IN std_logic_vector(2 DOWNTO 0);  -- 3 位操作码
    in_flags    : IN nibble; -- 输入标志位（4 位，实际未使用）
    z_out       : OUT byte;   -- 运算结果输出
    out_flags : OUT nibble --  新标志位输出  [V C Z N]
```

```
);
END arithmetic_logic_unit;

ARCHITECTURE synthesizable_behavioral OF arithmetic_logic_unit IS
BEGIN
coding: PROCESS (a_side, b_side, code, in_flags)
    VARIABLE t : std_logic_vector(9 DOWNTO 0);   -- 临时存储: | 溢出 | 进位 | 结果（8 位） |
    VARIABLE v, c, z, n : std_logic;   -- 标志位变量: V(溢出), C(进位), Z(零), N(负数)
BEGIN
    CASE code IS
        -- 加减法操作（唯一更新 C/V 的情况）
        WHEN a_add_b | a_sub_b =>
            -- 调用自定义函数处理加减法的进位和溢出
            -- 参数: (B, A, 进位输入, 减法标志)
            -- 输出: t(9)=溢出 V, t(8)=进位 C, t(7:0)=结果
            t := addsub_cv(b_side, a_side, in_flags(2), code(1));
            c := t(8);   -- 进位来自第 8 位
            v := t(9);   -- 溢出来自第 9 位

        -- 逻辑与操作（AND）
        WHEN a_and_b =>
            t := "00" & (a_side AND b_side);   -- 高位补 0
            c := '0'; v := '0';   --强制清零进位和溢出

        -- A 输入直通（MOV A）
        WHEN a_input =>
            t := "00" & a_side;       -- 直接输出 A
```

```
            c := '0'; v := '0';        -- 清零标志

        -- B 输入直通（MOV B）
        WHEN b_input =>
            t := "00" & b_side;        -- 直接输出 B
            c := '0'; v := '0';

        -- B 取反操作（NOT B）
        WHEN b_compl =>
            t := "00" & (NOT b_side);   -- 按位取反
            c := '0'; v := '0';

        -- 默认操作（异常处理）
        WHEN OTHERS =>
            t := (OTHERS => '0');   -- 清零所有输出
            c := '0'; v := '0';
    END CASE;

    -- 通用标志位计算（适用于所有操作）
    n := t(7);                  -- 负标志：取结果的最高位
    z := NOT all_or(t(7 DOWNTO 0)); -- 零标志：结果全 0 时置 1（all_or 为自定义函数）

    -- 输出连接
    z_out <= t(7 DOWNTO 0);   -- 取低 8 位作为结果
    out_flags <= v & c & z & n; -- 标志位打包顺序：[V C Z N]

END PROCESS coding;
END synthesizable_behavioral;
```

4. 仿真波形以及分析

以下是对 ALU 的单独仿真，其中包含 10 个测试用例，分别是对 ALU 可能出现场景的测试，测试场景分别为：基本加法，有符号溢出的运算，逻辑 AND 操作，B 输入直通，B 取反操作，减法运算（正常），减法下溢（负数结果），A 输入直通，边界值加法（最大正数加 1），零值检测。

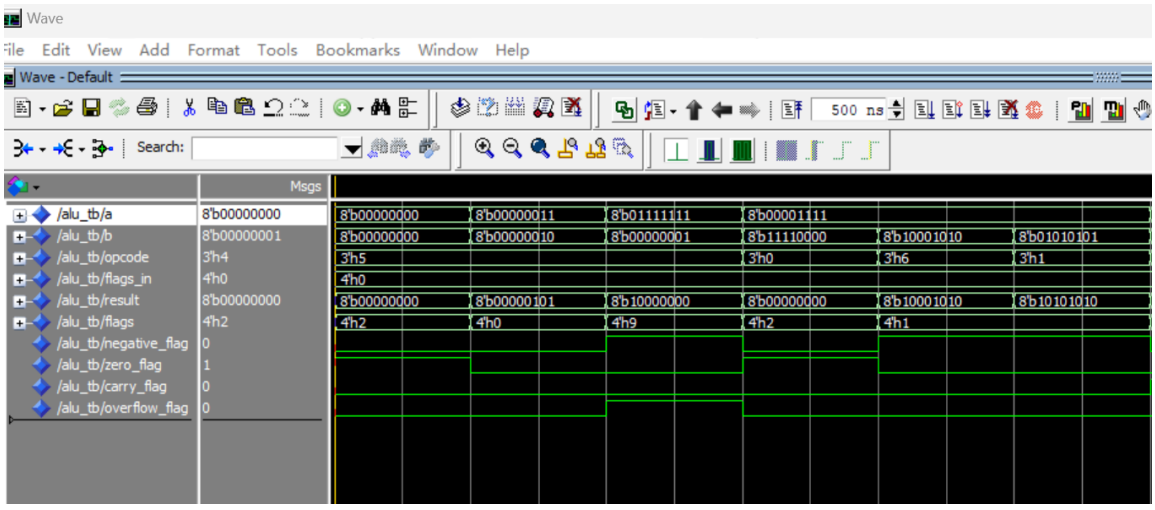

图 3- 4

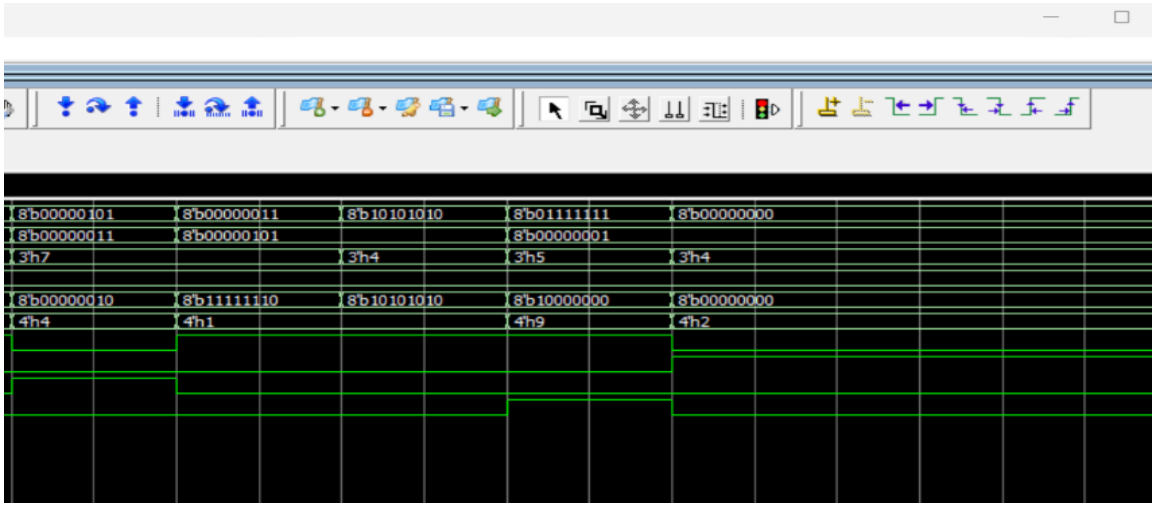

图 3- 5

在图 3-4 中是前五个测试用例，图 3-5 为后五个测试用例

测试用例 1：加法 3+2——正确结果 0x05（00000101）

测试用例 2：加法溢出——正确结果 0x80（10000000）

测试用例 3：逻辑与——正确结果 0x00（00000000）

测试用例 4: B 直通——正确结果 0x8A（10001010）

测试用例 5: B 取反——正确结果 0xAA（10101010）

测试用例 6: 减法运算 5-3（正常）——正确结果 0x02（00000010）

测试用例 7: 减法下溢 3-5（负数结果）——正确结果 0xFE（11111110）

测试用例 8: A 输入直通——正确结果 0xAA（10101010）

测试用例 9: 边界值加法（最大正数+1）——正确结果 0x80（10000000）

测试用例 10: 零值检测——正确结果 0x00（00000000）

所有测试用例均通过，ALU 测试成功

(二) CONTROL SECTION

1. 状态转换图

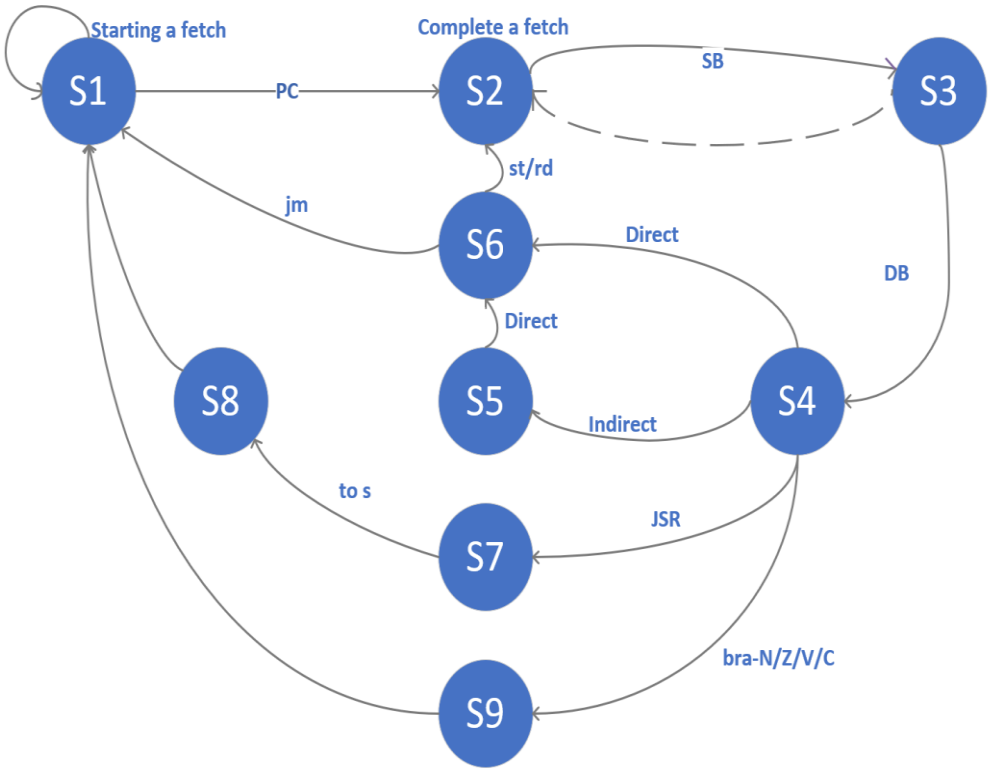

图 3- 6

状态映射关系如下:

状态	代码对应状态	对应操作
S1	instr_fetch	PC→MAR，启动取指周期
S2	opnd_fetch	完成指令获取后进入
S3	do_one_bytes	单字节指令执行
S4	do_indirect	间接寻址处理
S5	do_two_bytes	双字节指令操作数处理
S6	do_two_bytes	直接寻址执行
S7	do_jsr	JSR 调用第一阶段
S8	continue_jsr	JSR 第二阶段
S9	do_branch	条件分支执行

图 3- 7

可以看出，控制器主要是对读取的指令分析，然后对每一种指令产生对应的控制信号，从而使各个模块协调工作，完成特定的功能，但这需要我们分析出具体的指令类型，然后才能发出控制信号。

从 S1 的 Starting a fetch （将 PC 的值放到 MAR 寄存器中）转到 S2 的 Complete a fetch（读取 PC 所指向的地址的一个字节的数据，并放到指令寄存器中）后，判断读取的指令是否为双字节指令。

若为单字节指令，则判断具体的指令类型，然后发送控制信号，并转移到 S2。否则转移到 S4。接下来读取第二个字节，放到 MAR 中的页地址当中，同时根据第一个字节判断当前的指令是全地址指令还是页内地址指令。

若为全地址指令，则把 IR 中的页地址放到 MAR 中的页地址中，同时判断是否为直接寻址，若为直接寻址则转向 S6（direct 直接寻址），否则转向 S5（indirect

处理双字节指令操作数)。在 S5 读取一个字节,并存放在 MAR 中的页地址中后,转向 S6;

若为页内地址指令,则判断是 JSR 指令还是分支指令,若是 JSR,则转向 S7;若是分支指令,则根据操作码判断具体指令类型,然后转向 S9。

在 S6 先判断当前的指令是否为 JMP 指令,若是,则设置 PC 的值,并转移到 S1。判断是否为 STA 指令,若是,则这行该指令,向内存中写入数据,并转移到 S2。若是全地址指令中的其他指令,则读入操作数,然后执行算术运算,并转移到 S2。

S7 为 JSR 调用第一阶段,执行 JSR 指令,接着把当前 PC 的值写入内存地址便宜地址为 tos 处,并转移到(JSR 第二阶段)

S8 完成 PC 的地址增加运算,并转向到 S1。S9 执行具体的默认地址指令,并转向到 S1。

2. 指令实现

1) 默认地址指令:

在状态 S1,将 PC 的值放到 MAR 寄存器中,转移到 S2;在 S2 状态,读取一个字节,转移到 S3;在 S3,判断为默认地址指令,则执行指令,并转移到 S1,重新读取下一条指令。即转移状态为:S1→S2→S3→S1。

2) 页内地址指令:

在状态 S1,将 PC 的值放到 MAR 寄存器中,转移到 S2;在 S2 状态,读取一个字节,转移到 S3;在 S3,判断为双字节指令,转移到 S4;在 S4,判断为页内地址指令。对 JSR 指令,转移到 S7,保存 PC 的值,并转移到 S8;在 S8,设置 PC 的新值,并转移到 S1;对非 JSR 的指令,转移到 S9,执行程序跳转,然后

转移到 S1。

即：对 JSR 转移路径为：S1→S2→S3→S4→S7→S8→S1

对非 JSR 转移路径为：S1→S2→S3→S4→S9→S1

3) 全地址指令

状态 S1，将 PC 的值放到 MAR 寄存器中，转移到 S2；在 S2 状态，读取一个字节，转移到 S3；在 S3，判断为双字节指令，转移到 S4；在 S4，判断为全地址指令，对直接寻址，转移到 S6，在 S6，执行指令，若非 JMP 指令，则转移到 S2，否则转移到 S1；对间接寻址，转移到 S5，在 S5，读取数据的内存地址，然后转到 S6，在 S6 执行指令，若非 JMP 指令，则转移到 S2，否则转移到 S1。即：

直接 JMP 指令：S1→S2→S3→S4→S6→S1

直接非 JMP 指令：S1→S2→S3→S4→S6→S2

间接 JMP 指令：S1→S2→S3→S4→S5→S6→S1

间接非 JMP 指令：S1→S2→S3→S4→S5→S6→S2

3. Control Section 代码解析

```
-- ========== 状态映射 ========== --
-- S1(Starting a fetch) → initial
-- S2(Complete a fetch) → instr_fetch
-- S3 → do_one_bytes
-- S4 → opnd_fetch
-- S5 → do_indirect
-- S6 → do_two_bytes
-- S7 → do_jsr
-- S8 → continue_jsr
-- S9 → do_branch

-- ========== 端口声明 ========== --
```

```
ENTITY par_control_unit IS
PORT (
    clk : IN std_logic;
    -- 寄存器控制信号（对应数据路径操作）
    load_ac, zero_ac,              -- 累加器控制
    load_ir,                       -- 指令寄存器加载
    increment_pc,                  -- PC 自增
    load_page_pc, load_offset_pc, reset_pc, -- PC 管理
    load_page_mar, load_offset_mar, -- MAR 加载
    -- ... (其他端口与状态图信号对应)
);

-- ========== 状态机核心逻辑 ========== --
ARCHITECTURE dataflow_synthesizable OF par_control_unit IS
    -- 定义 9 个状态（与图中 S1-S9 对应）
    TYPE cpu_states IS (
        initial,          -- S1: 初始/取指启动
        instr_fetch,   -- S2: 完成取指
        do_one_bytes, -- S3: 单字节指令执行
        opnd_fetch,    -- S4: 操作数获取（双字节指令）
        do_indirect,   -- S5: 间接寻址处理
        do_two_bytes,   -- S6: 双字节指令执行
        do_jsr,            -- S7: JSR 调用阶段 1
        continue_jsr,   -- S8: JSR 调用阶段 2
        do_branch        -- S9: 分支指令处理
    );
```

```
    SIGNAL present_state, next_state : cpu_states;

BEGIN
    -- 时钟驱动状态转换（对应状态图箭头）
    clocking : PROCESS (clk, interrupt)
    BEGIN
        IF (interrupt = '1') THEN    -- 中断强制返回 S1
            present_state <= initial;
        ELSIF clk'EVENT AND clk = '0' THEN -- 下降沿触发
            present_state <= next_state;      -- 状态转移
        END IF;
    END PROCESS;

    -- 组合逻辑：状态转换和信号生成
    sequencing : PROCESS (present_state, ir_lines, status, ready)
    BEGIN
        -- 默认信号初始化（避免锁存器）
        load_ac <= '0'; zero_ac <= '0';    -- 清零累加器信号
        read_mem <= '0'; write_mem <= '0'; --  内存控制

        CASE present_state IS
            -- ===== S1：初始/取指启动 ===== --
            WHEN initial =>    -- 对应状态图 S1
                IF interrupt = '1' THEN
                    reset_pc <= '1';          -- PC 复位（中断处理）
                    next_state <= initial;
                ELSE
                    -- 启动取指周期：PC→MAR（图中 S1→S2）
                    pc_on_mar_page_bus <= '1';    -- PC 页地址送 MAR
                    pc_on_mar_offset_bus <= '1';-- PC 偏移送 MAR
```

```
                load_page_mar <= '1';            -- 锁存 MAR 页
                load_offset_mar <= '1';          -- 锁存 MAR 偏移
                next_state <= instr_fetch;    -- 转 S2
            END IF;

        -- ===== S2: 完成取指 ===== --
        WHEN instr_fetch =>    -- 对应状态图 S2
            mar_on_adbus <= '1';    -- MAR 输出到地址总线
            read_mem <= '1';          -- 启动内存读
            IF ready = '1' THEN      -- 内存操作完成
                databus_on_dbus <= '1';    -- 数据总线→内部总线
                load_ir <= '1';                -- 加载 IR（图中指令解码）
                increment_pc <= '1';         -- PC 自增
                -- 根据指令长度选择路径
                IF ir_lines(7 DOWNTO 4) = single_byte_instructions
THEN
                    next_state <= do_one_bytes; -- S3（单字节）
                ELSE
                    next_state <= opnd_fetch;     -- S4（双字节）
                END IF;
            ELSE
                next_state <= instr_fetch;    -- 等待内存就绪
            END IF;

        -- ===== S3: 单字节指令执行 ===== --
        WHEN do_one_bytes =>    -- 对应状态图 S3
            CASE ir_lines(3 DOWNTO 0) IS
```

```
            WHEN cla =>        -- CLA 指令
                zero_ac <= '1';    --  清零 AC
                load_ac <= '1';      --  锁存清零结果
            WHEN asr =>        --  算术右移
                arith_shift_right <= '1'; --  激活移位
                alu_code <= b_input;          -- ALU 模式设置
                load_ac <= '1';                  --  更新 AC
            -- ...  其他单字节指令
        END CASE;
        next_state <= initial;    --  返回 S1（完成执行）

-- ===== S4:  操作数获取（双字节）  ===== --
WHEN opnd_fetch =>    --  对应状态图 S4
        mar_on_adbus <= '1';    -- MAR 输出地址
        read_mem <= '1';           --  读操作数
        IF ready = '1' THEN
            dbus_on_mar_offset_bus <= '1';--  总线→MAR 偏移
            load_offset_mar <= '1';             --  锁存偏移
            --  判断指令类型（JSR/分支/其他）
            IF ir_lines(7 DOWNTO 6) = jsr_or_bra THEN
                IF ir_lines(5) = '0' THEN
                    next_state <= do_jsr;        -- S7（JSR）
                ELSE
                    next_state <= do_branch; -- S9（分支）
                END IF;
            ELSE    --  非 JSR/分支指令
```

```
                IF ir_lines(4) = indirect THEN
                    next_state <= do_indirect; -- S5（间接）
                ELSE
                    next_state <= do_two_bytes;-- S6（直接）
                END IF;
            END IF;
        ELSE
            next_state <= opnd_fetch;   -- 等待内存
        END IF;

-- ===== S5: 间接寻址处理 ===== --
WHEN do_indirect =>   -- 对应状态图 S5
        mar_on_adbus <= '1';   -- 输出间接地址
        read_mem <= '1';          -- 读取实际操作数地址
        IF ready = '1' THEN
            dbus_on_mar_offset_bus <= '1';-- 更新 MAR 偏移
            load_offset_mar <= '1';
            next_state <= do_two_bytes;   -- 转 S6 执行
        ELSE
            next_state <= do_indirect;       -- 等待内存
        END IF;

-- ===== S6: 双字节指令执行 ===== --
WHEN do_two_bytes =>   -- 对应状态图 S6
        CASE ir_lines(7 DOWNTO 5) IS
            WHEN jmp =>       -- 跳转指令
                load_page_pc <= '1';     -- 加载 PC 页
                load_offset_pc <= '1'; -- 加载 PC 偏移
                next_state <= initial; -- 返回 S1
```

```
            WHEN sta =>        -- 存储指令
                mar_on_adbus <= '1';      -- 输出目标地址
                obus_on_dbus <= '1';      -- AC→数据总线
                write_mem <= '1';          -- 启动写操作
                IF ready = '1' THEN
                    next_state <= initial; -- 完成
                ELSE
                    next_state <= do_two_bytes; -- 等待
                END IF;
            -- ... 其他双字节指令（LDA/ADD 等）
        END CASE;

-- ===== S7: JSR 阶段 1 ===== --
WHEN do_jsr =>    -- 对应状态图 S7
        mar_on_adbus <= '1';          -- 输出返回地址位置
        pc_offset_on_dbus <= '1';-- PC 偏移→数据总线
        write_mem <= '1';              -- 保存返回地址
        IF ready = '1' THEN
            load_offset_pc <= '1';    -- 加载子程序偏移
            next_state <= continue_jsr; -- 转 S8
        ELSE
            next_state <= do_jsr;       -- 等待写入完成
        END IF;

-- ===== S8: JSR 阶段 2 ===== --
WHEN continue_jsr =>    -- 对应状态图 S8
        increment_pc <= '1';    -- PC 自增（完成跳转）
```

```
                next_state <= initial;--  返回 S1


            -- ===== S9:  分支处理  ===== --

            WHEN do_branch =>    --  对应状态图 S9
                IF (status AND ir_lines(3 DOWNTO 0)) /= "0000" THEN
                    load_offset_pc <= '1';    --  满足条件时更新 PC
                END IF;
                next_state <= initial;            --  返回 S1
        END CASE;
    END PROCESS;
END dataflow_synthesizable;
```

4. 仿真波形以及分析

==========

测试用例 1：初始状态 → 指令取指

输入信号：

- interrupt = 0

- ready = 1

预期波形现象：

1. 第一个 CLK 下降沿后：

- pc_on_mar_page_bus = 1

- pc_on_mar_offset_bus = 1

- load_page_mar = 1

- load_offset_mar = 1

- 状态机迁移：initial → instr_fetch

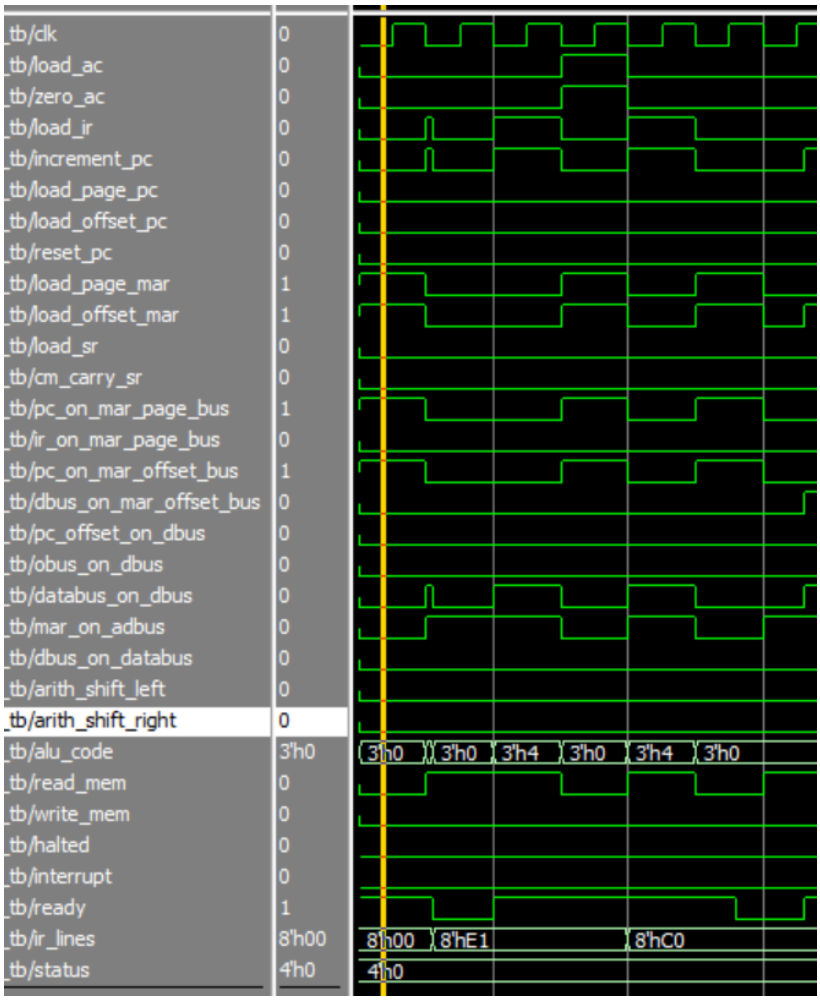

图 3- 8

==========

测试用例 2：CLA 累加器清零

输入信号：

- ir_lines = 11100001

预期波形现象：

1. 执行周期：

    - zero_ac = 1

    - load_ac = 1

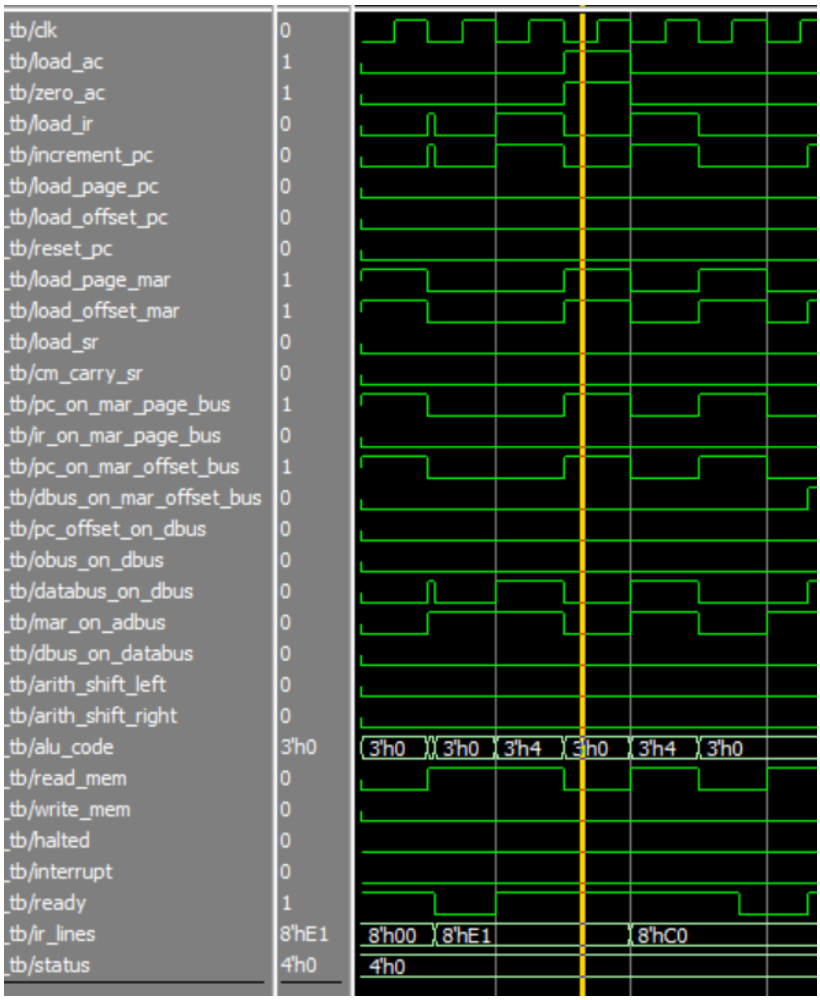

图 3- 9

==========

测试用例 3：JSR 子程序调用

输入信号：

- ir_lines = 11000000

预期波形现象：

阶段 1（地址存储）：

- pc_offset_on_dbus = 1

- write_mem = 1

- mar_on_adbus = 1

dbus_on_databus = 1

阶段 2（跳转执行）:

- ready=0 时维持 do_jsr 状态

- ready=1 时 load_offset_pc = 1

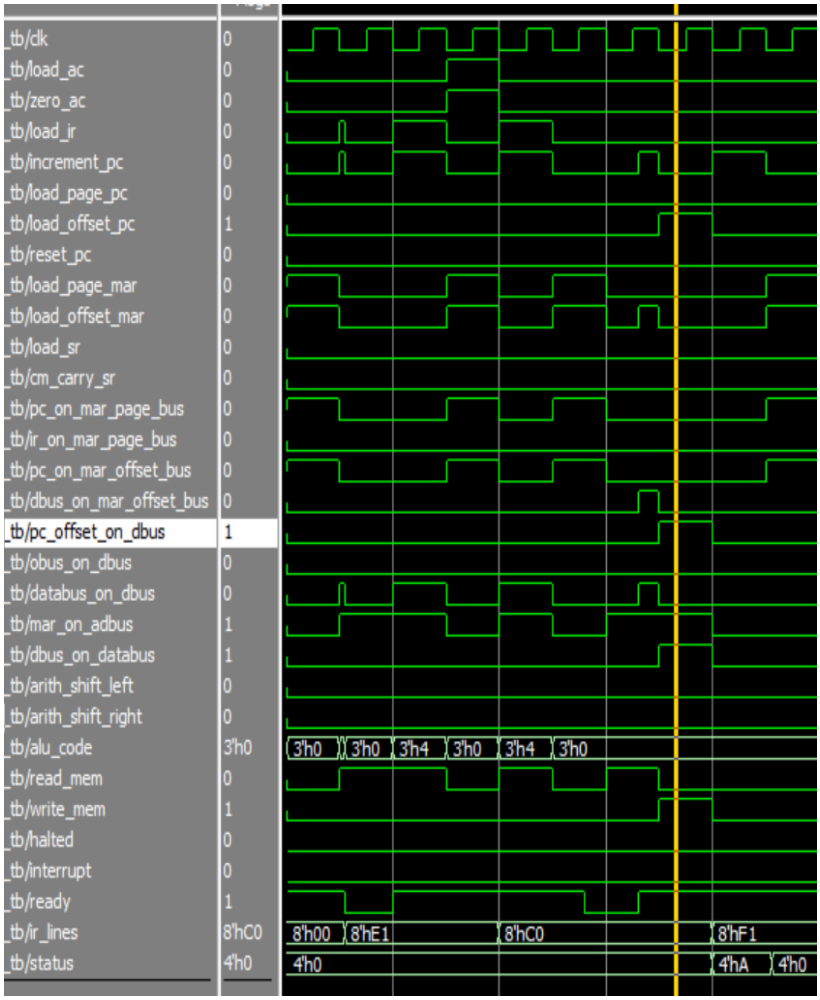

图 3- 10

==========

测试用例 4：条件分支（不满足）

输入信号:

- status = 0000

- ir_lines = 11110001

预期波形现象:

- load_offset_pc = 0

- 状态直接回迁 initial

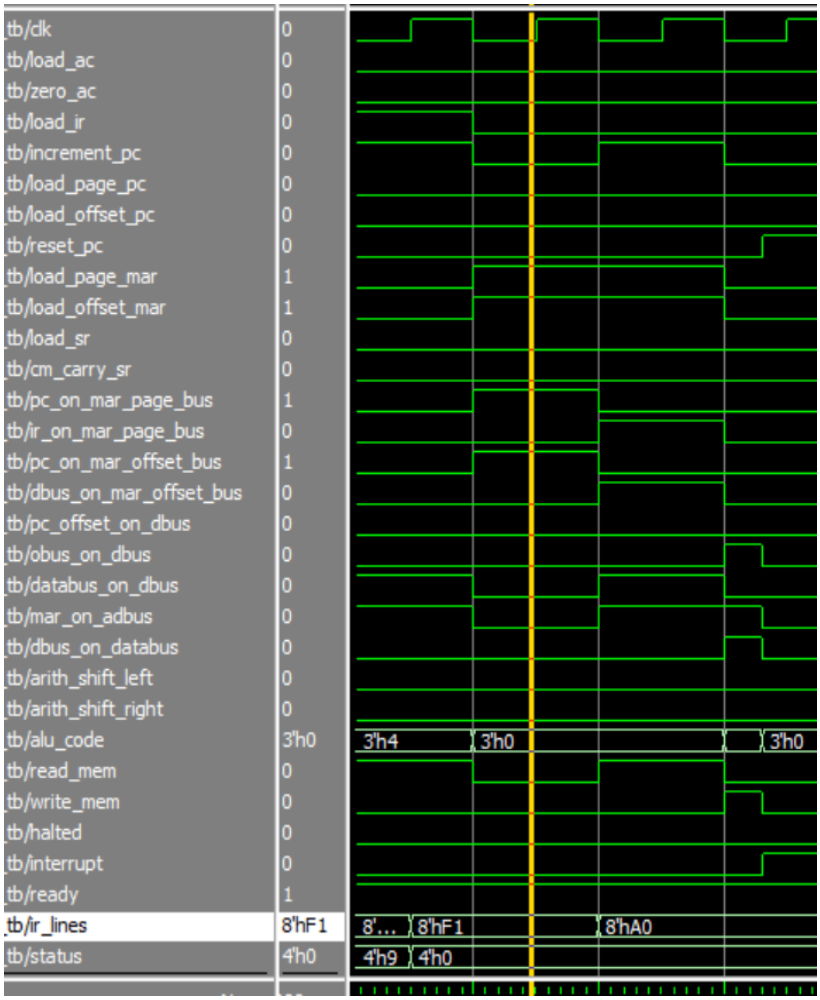

图 3- 11

==========

测试用例 5：STA 存储操作

输入信号：

- ir_lines = 10100000

预期波形现象：

- mar_on_adbus = 1

- write_mem = 1

- obus_on_dbus = 1

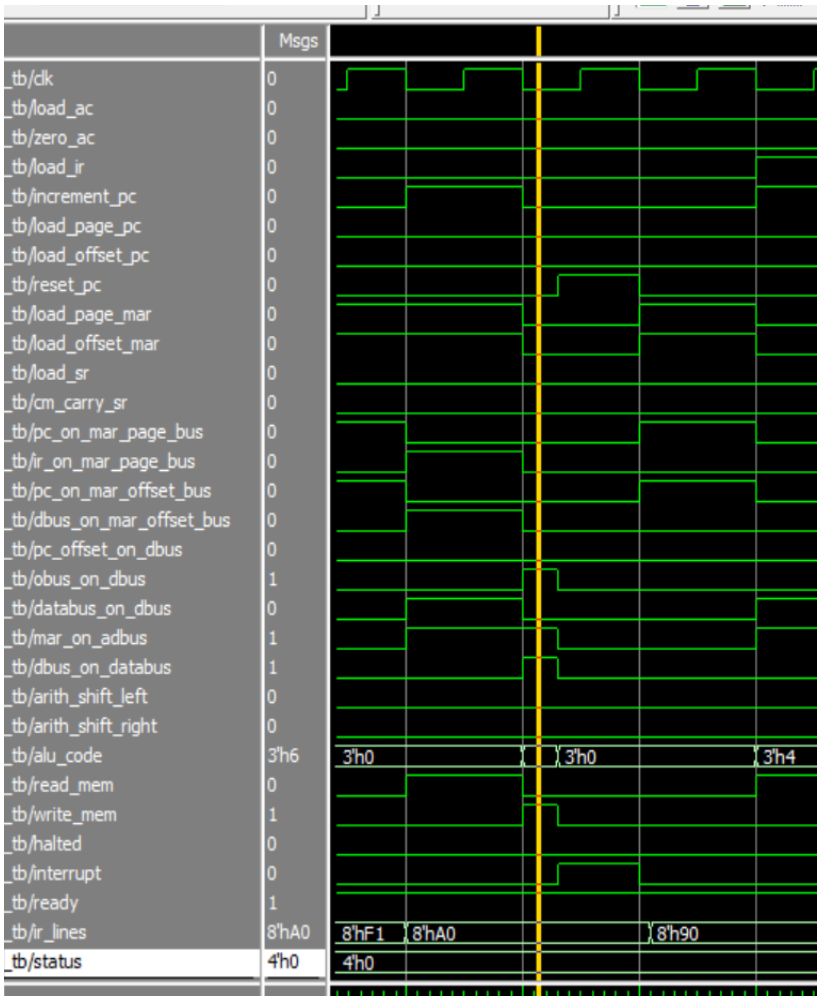

图 3- 12

==========

测试用例 6：中断响应

输入信号：

- interrupt = 1

预期波形现象：

- reset_pc = 1

- 所有总线信号归零

- 状态强制回 initial

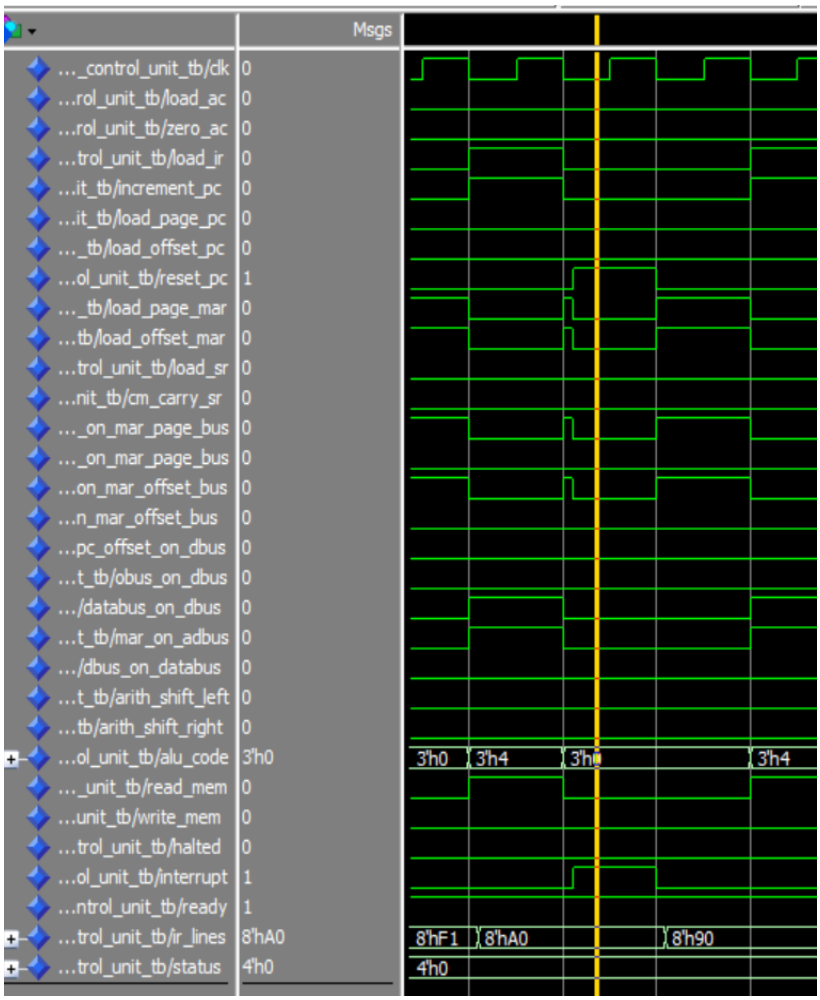

图 3- 13

==========

测试用例 7：间接寻址

输入信号：

- ir_lines = 10010000

预期波形现象：

阶段 1（取地址）：

- dbus_on_mar_offset_bus = 1

- load_offset_mar = 1

阶段 2（取操作数）：

- read_mem = 1

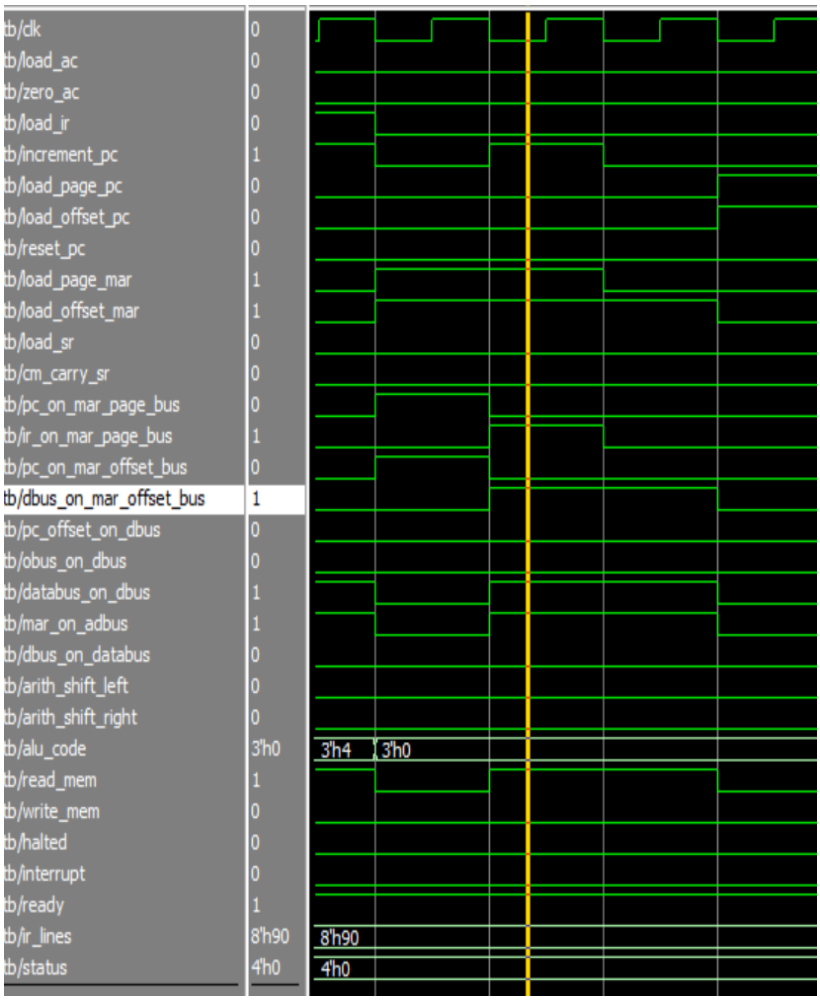

图 3- 14

四、 整体综合与测试

PARWAN CPU 是存储程序然后执行的，执行程序之前，必须先把可执行代码写入到内存中。在 PARWAN CPU 仿真过程中，PARWAN CPU 的内存并不是物理上存在的，而是 testbench 代码中的一个变量。

所以需要事先写一个简单的汇编测试程序 simple.asm 来测试 CPU 的功能（如下图 4-1），此程序需要转换为仿真器可读的内存文件，这个过程跟我们常说的“编译”类似。通过已经写好的编译 PARWAN 汇编代码的“编译器”——par_asm.c，我们将 simple.asm 编译成 tmp.asb 文件。这就是我们所需的内存文件，这个文件对于 VHDL 的编译器来说是可读的，并用来初始化 parwan 的内存。将 tmp.asb 文件拷贝到 ISE 相应工程目录，开始仿真，即可得到仿真波形图 4-2

```
1 lda x20
2 add x3
3 sta y
4 lda x20
5 lda y
6 label end
7 jmp end
8 int x1 1
9 int x3 3
10 int x20 20
11 int y 100
12 int x255 255
13 int A 7
14 int B 34
15 int C 64
16 int D 23
17 int E 56
18 int F 98
19 int G 45
20 int H 23
```

图 4- 1

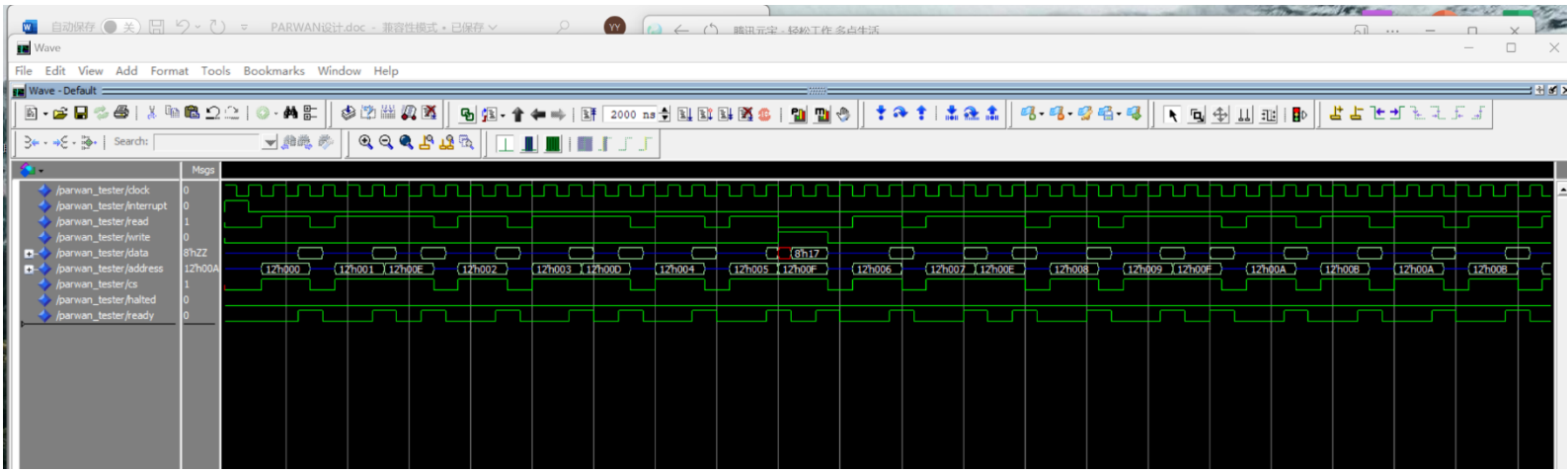

图 4- 2