单周期CPU实验报告

518030910408 周韧

2020/05/08

目录

1 实验环境

本次实验在Quartus II 13.0(64-bit)开发软件中进行。

2 实验任务

★ 采用 Verilog 硬件描述语言在 Quartus II EDA 设计平台中，基于 Intel cyclone II 系列FPGA 完成具有执行 20 条 MIPS 基本指令的单周期 CPU 模块的设计。根据提供的单周期 CPU 示例程序的 Verilg 代码文件，将设计代码补充完整，实现该模块的电路设计。

★ 利用实验提供的标准测试程序代码，完成单周期 CPU 模块的功能仿真测试，验证 CPU 执行所设计的 20 条 RISC 指令功能的正确性。从而理解计算机五大组成部分的协调工作原理，理解存储程序自动执行的原理和掌握运算器、存储器、控制器的设计和实现原理。

3 设计思路分析

3.1 完善Real_Value_Table_to student.xls文件

这份表格中左半部分包含MIPS中及主要的R型、I型和J型指令，以及每个指令对应的指令格式和指令字段，右半部分包含该指令对应的控制信号。我们根据课上所学的单周期CPU的知识完善这个表格，作为后续修改核心代码的依据。

对其中控制信号的说明如表1所示:

我们列举出每个控制信号的含义，之后对每个指令的控制信号数据进行相应的完善。展示完善的Excel表格如图1所示:

表 1: 控制信号表格

信号名称	作用
pcsource	下一条指令来源
aluc	运算控制信号
shift	跳转指令信号
aluimm	立即数参与运算信号
sext	符号扩展信号
wmem	写存储器信号
wreg	写寄存器信号
m2reg	存储器数据写入寄存器信号
regrt	写入Rt/Rd寄存器信号
call/jar	调用函数/跳转信号

图 1: 完整指令表格

	A	B	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1	输入																			
2	指令	指令格式	op	rs	rt	rd	sa	func		z	pcsource	aluc	shift	aluimm	sext	wmem	wreg	m2reg	regrt	call
3											[1..0]	[3..0]								jal
4	add	add rd, rs, rt	000000	rs	rt	rd	00000	100000		x	0 0	0 0 0 0	0	0	x	0	1	0	0	0
5	sub	sub rd, rs, rt	000000	rs	rt	rd	00000	100010		x	0 0	x 1 0 0	0	0	x	0	1	0	0	0
6	and	and rd, rs, rt	000000	rs	rt	rd	00000	100100		x	0 0	x 0 0 1	0	0	x	0	1	0	0	0
7	or	or rd, rs, rt	000000	rs	rt	rd	00000	100101		x	0 0	x 1 0 1	0	0	x	0	1	0	0	0
8	xor	xor rd, rs, rt	000000	rs	rt	rd	00000	100110		x	0 0	x 0 1 0	0	0	x	0	1	0	0	0
9	sll	sll rd, rt, sa	000000	00000	rt	rd	sa	000000		x	0 0	0 0 1 1	1	0	x	0	1	0	0	0
10	srl	srl rd, rt, sa	000000	00000	rt	rd	sa	000010		x	0 0	0 1 1 1	1	0	x	0	1	0	0	0
11	sra	sra rd, rt, sa	000000	00000	rt	rd	sa	000011		x	0 0	1 1 1 1	1	0	x	0	1	0	0	0
12	jr	jr rs	000000	rs	00000	00000	00000	001000		x	1 0	x x x x	x	x	x	0	0	x	x	x
13																				
14	指令	指令格式	op	rs	rt	rd	sa	func			pcsource	aluc	shift	aluimm	sext	wmem	wreg	m2reg	regrt	call
15											[1..0]	[3..0]								jal
16																				
17	addi	addi rt, rs, im	001000	rs	rt	imm				x	0 0	0 0 0 0	0	1	1	0	1	0	1	0
18	andi	andi rt, rs, im	001100	rs	rt	imm				x	0 0	x 0 0 1	0	1	0	0	1	0	1	0
19	ori	ori rt, rs, im	001101	rs	rt	imm				x	0 0	x 1 0 1	0	1	0	0	1	0	1	0
20	xori	xori rt, rs, im	001110	rs	rt	imm				x	0 0	x 0 1 0	0	1	0	0	1	0	1	0
21	lw	lw rt, imm(rs	100011	rs	rt	imm				x	0 0	0 0 0 0	0	1	1	0	1	1	1	0
22	sw	sw rt, imm(rs	101011	rs	rt	imm				x	0 0	0 0 0 0	0	1	1	1	1	x	1	0
23	beq	beq rs, rt, im	000100	rs	rt	imm				0	0 0	x 1 0 0	0	0	1	0	0	0	0	0
24										1	0 1									
25	bne	bne rs, rt, im	000101	rs	rt	imm				0	0 1	x 1 0 0	0	0	1	0	0	0	0	0
26										1	0 0									
27	lui	lui rt, imm	001111	00000	rt	imm				x	0 0	x 1 1 0	0	1	1	0	1	0	1	0
28																				
29	j	j addr	000010	addr							1 1	x x x x	x	x	x	0	0	x	x	x
30	jal	jal addr	000011	addr							1 1	x x x x	x	x	x	0	1	x	x	1
31																				
32	指令	指令格式	op	rs	rt	rd	sa	func			pcsource	aluc	shift	aluimm	sext	wmem	wreg	m2reg	regrt	call
33											[1..0]	[3..0]								jal

3.2 完善代码部分

将指令的完整表格填写完毕检查无误之后，我们开始对代码部分的完善。Quartus的使用和文件的导入，课程上陈老师已经进行了非常详细地讲解和指导，因此在这里我们不做赘述。本次实验中需要补充的代码在alu.v和sc_cu.v两个文件中，因此我们主要说明实验中对核心代码部分的补充。

3.2.1 alu.v

alu.v中缺少的是在一个选择结构casex中，aluc控制信号对应的ALU的各种运算操作。因此我们在Real_Value_Table_to student.xls中找到aluc这一栏，例如SUB的aluc 控制信号是x100，我们则在分支结构中补上

$$4'bx100 : s = a - b;$$

其他的运算符号也是如此。值得一提的是SLL和SRL运算，这两个位移运算在计算之前首先要对a操作数进行无符号扩展，因此我们在对a进行$unsigned处理:

左移

$$4'b0011 : s = \$unsigned(b) << a;$$

右移

$$4'b0111 : s = \$unsigned(b) >> a;$$

至此我们完成了对alu.v文件的补充。

3.2.2 sc_cu.v

sc_cu.v文件中缺少的是各种运算操作对应的标记值，wire型变量r_type决定该信号是否为R型指令，若是则使用不同function来标记指令，如果不是R型指令，则通过op来标记指令。因此我们只需观察Real_Value_Table_to student.xls文件中op和function这两栏，填入相应的值即可。

例如and是一个R型指令，function值是100100，于是代码中赋值则为:

$$wire\ i\_and = r\_type \& func[5] \& \sim func[4] \& \sim func[3] \& func[2] \& \sim func[1] \& \sim func[0];$$

又如addi是一个I型指令, opcode值是001000，于是代码中赋值则为:

$$wire\ i\_addi = \sim op[5] \& \sim op[4] \& op[3] \& \sim op[2] \& \sim op[1] \& \sim op[0];$$

将该文件中所有相应的操作值补全，我们就完成了对sc_cu.v的补充。

4 核心代码

图2: alu.v核心代码

source/sc_cu.v source/alu.v

```
1  module alu (a,b,aluc,s,z);
2     input [31:0] a,b;
3     input [3:0] aluc;
4     output [31:0] s;
5     output        z;
6     reg [31:0] s;
7     reg        z;
8     always @ (a or b or aluc)
9        begin                                  // event
10          casex (aluc)
11              4'bx000: s = a + b;             //x000 ADD
12              4'bx100: s = a - b;             //x100 SUB
13              4'bx001: s = a & b;             //x001 AND
14              4'bx101: s = a | b;             //x101 OR
15              4'bx010: s = a ^ b;             //x010 XOR
16              4'bx110: s = a << 16;           //x110 LUI: imm << 16bit
17              4'b0011: s = $unsigned(b) << a; //0011 SLL: rd <- (rt << sa)
18              4'b0111: s = $unsigned(b) >> a; //0111 SRL: rd <- (rt >> sa) (logical)
19              4'b1111: s = $signed(b) >>> a;  //1111 SRA: rd <- (rt >> sa) (arithmetic)
20              default: s = 0;
21          endcase
22          if (s == 0 )  z = 1;
23             else z = 0;
24       end
25 endmodule
```

图3: sc_cu.v核心代码

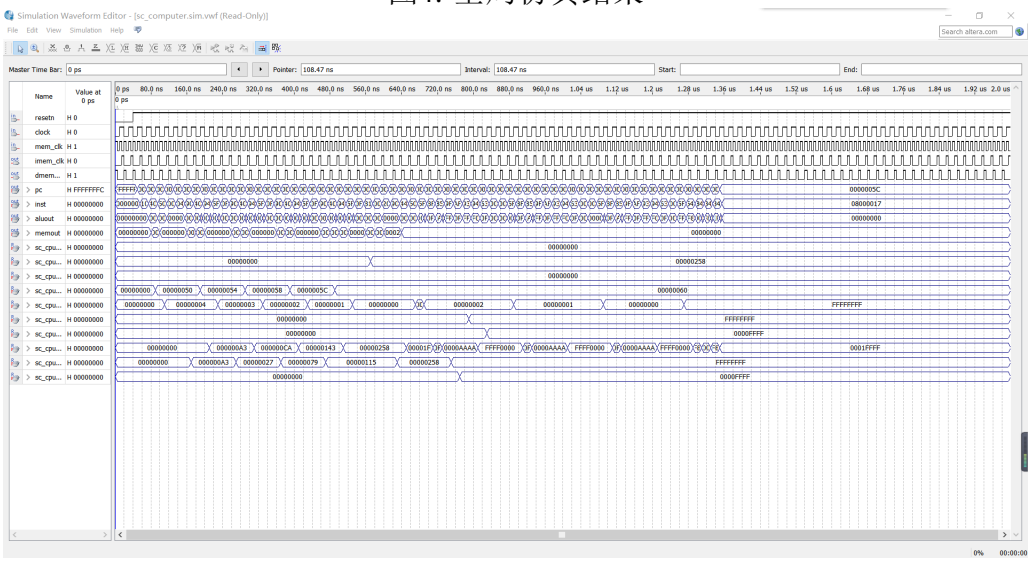
source/sc_cu.v

```
1  module sc_cu (op, func, z, wmem, wreg, regrt, m2reg, aluc, shift,
2                aluimm, pcsource, jal, sext);
3      input  [5:0] op,func;
4      input        z;
5      output       wreg,regrt,jal,m2reg,shift,aluimm,sext,wmem;
6      output [3:0] aluc;
7      output [1:0] pcsource;
8      wire r_type = ~|op;
9      wire i_add = r_type & func[5] & ~func[4] & ~func[3] &
10                  ~func[2] & ~func[1] & ~func[0];          //100000
11     wire i_sub = r_type & func[5] & ~func[4] & ~func[3] &
12                  ~func[2] &  func[1] & ~func[0];          //100010
13
14     wire i_and = r_type & func[5] & ~func[4] & ~func[3] &
15                  func[2] & ~func[1] & ~func[0];           //100100
16     wire i_or  = r_type & func[5] & ~func[4] & ~func[3] &
17                  func[2] & ~func[1] & func[0];            //100101
18     wire i_xor = r_type & func[5] & ~func[4] & ~func[3] &
19                  func[2] & func[1] & ~func[0];            //100110
20     wire i_sll = r_type & ~func[5] & ~func[4] & ~func[3] &
21                  ~func[2] & ~func[1] & ~func[0];          //000000
22     wire i_srl = r_type & ~func[5] & ~func[4] & ~func[3] &
23                  ~func[2] & func[1] & ~func[0];           //000010
24     wire i_sra = r_type & ~func[5] & ~func[4] & ~func[3] &
25                  ~func[2] & func[1] & func[0];          //000011
26     wire i_jr  = r_type & ~func[5] & ~func[4] & func[3] &
27                  ~func[2] & ~func[1] & ~func[0];          //001000
28
29     wire i_addi = ~op[5] & ~op[4] &  op[3] & ~op[2] & ~op[1] & ~op[0]; //001000
30     wire i_andi = ~op[5] & ~op[4] &  op[3] &  op[2] & ~op[1] & ~op[0]; //001100
31
32     wire i_ori  = ~op[5] & ~op[4] &  op[3] &  op[2] & ~op[1] &  op[0]; //001101
33     wire i_xori = ~op[5] & ~op[4] &  op[3] &  op[2] &  op[1] & ~op[0]; //001110
34     wire i_lw   =  op[5] & ~op[4] & ~op[3] & ~op[2] &  op[1] &  op[0]; //100011
35     wire i_sw   =  op[5] & ~op[4] &  op[3] & ~op[2] &  op[1] &  op[0]; //101011
36     wire i_beq  = ~op[5] & ~op[4] & ~op[3] &  op[2] & ~op[1] & ~op[0]; //000100
37     wire i_bne  = ~op[5] & ~op[4] & ~op[3] &  op[2] & ~op[1] &  op[0]; //000101
38     wire i_lui  = ~op[5] & ~op[4] &  op[3] &  op[2] &  op[1] &  op[0]; //001111
39     wire i_j    = ~op[5] & ~op[4] & ~op[3] & ~op[2] &  op[1] & ~op[0]; //000010
40     wire i_jal  = ~op[5] & ~op[4] & ~op[3] & ~op[2] &  op[1] &  op[0]; //000011
41
42
43     assign pcsource[1] = i_jr | i_j | i_jal;
44     assign pcsource[0] = ( i_beq & z ) | (i_bne & ~z) | i_j | i_jal ;
45
46     assign wreg = i_add | i_sub | i_and | i_or   | i_xor  |
47                   i_sll | i_srl | i_sra | i_addi | i_andi |
48                   i_ori | i_xori | i_lw | i_lui  | i_jal;
49
50     assign aluc[3] = i_sra;
51     assign aluc[2] = i_sub | i_or | i_srl | i_sra | i_ori | i_lui | i_bne | i_beq;
52     assign aluc[1] = i_xor | i_sll | i_srl | i_sra | i_xori | i_lui;
53     assign aluc[0] = i_and | i_or | i_sll | i_srl | i_sra | i_andi | i_ori;
54     assign shift   = i_sll | i_srl | i_sra ;
55
56     assign aluimm  = i_addi | i_andi | i_ori | i_xori | i_lw | i_sw | i_lui;
57     assign sext    = i_addi | i_lw | i_sw | i_beq | i_bne | i_lui;
58     assign wmem    = i_sw;
59     assign m2reg   = i_lw;
60     assign regrt   = i_addi | i_andi | i_ori | i_xori | i_lw | i_sw | i_lui;
61     assign jal     = i_jal;
62
63 endmodule
```

5 整体仿真结果

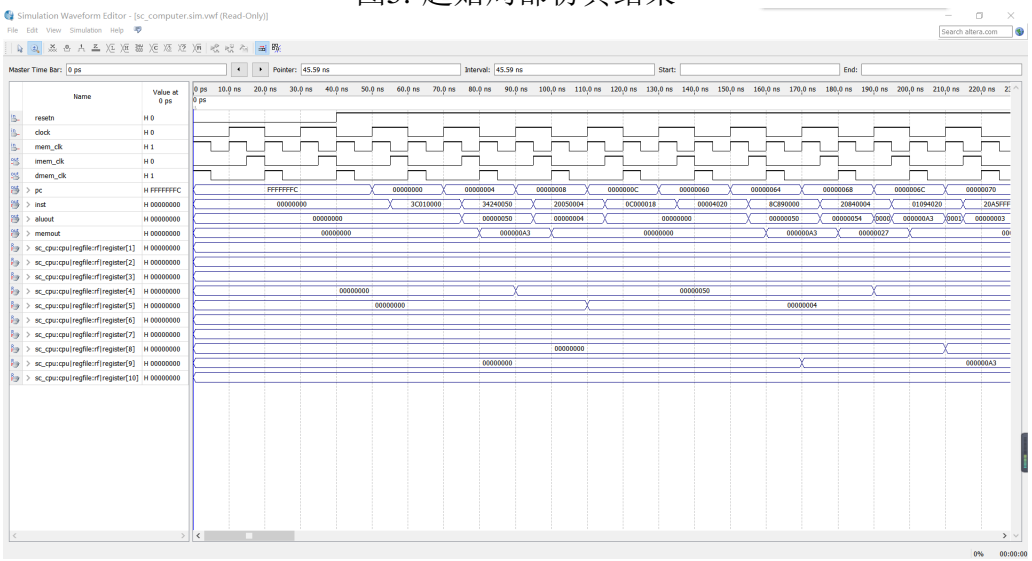
我们完成了对代码的补充之后，就可以使用RTL_simulation工具对结果进行模拟仿真了。先从看全局仿真结果:

图4: 全局仿真结果



从图中我们可以观察到全局的结果在起点与终点都与sc_computer实验要求中所给的图片结果保持一致。但是波形太密没有办法仔细观察，因此我们放大图片挑出一些细节进行分析说明:

图5: 起始局部仿真结果



当resetn信号置为高电平后，单周期CPU在时钟沿的控制下开始工作。可以看到pc指向位置0000 0000，之后instmem中的指令3C10 0000被取出，从sc_instmem.mif文件中可以看出，该指令是lui $1, 0，将立即数0放入$1寄存器的高16位，低16位补0，因此可以看到$1中的值仍然为0000 0000; 之后pc指向位置0000 0004，取出指令34240050，该指令是ori $4, $1, 80，将寄存器$1中的值与立即数80按位或后放入$4寄存器中，可以看到执行该操作时，aluout的值变为0000 0050，这个16进制数转成10进制就等于80，也就是$1(里面的值为0000 0000)和80按位或的结果。下一个时钟周期，寄存器$4的结果果然变成了aluout的值。

这样的指令无法在这里一一讲解，但是我们通过对几个指令的验后说明了我们仿真结果的正确性。我们同时列举出几张其他时段的仿真结果来进一步验证结果，这几张图均与实验要求中的结果一致。

图6: 局部仿真结果图片

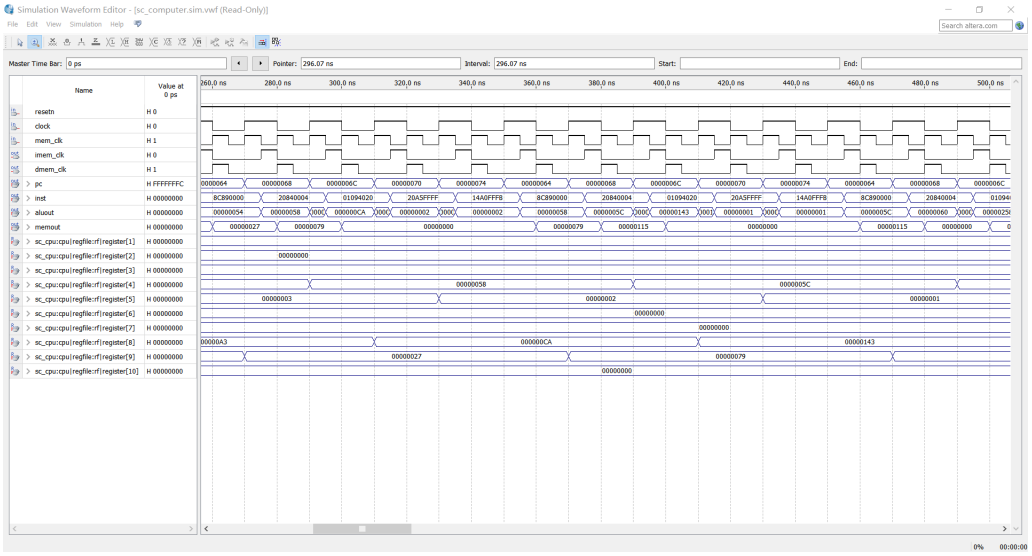


图7: 局部仿真结果图片

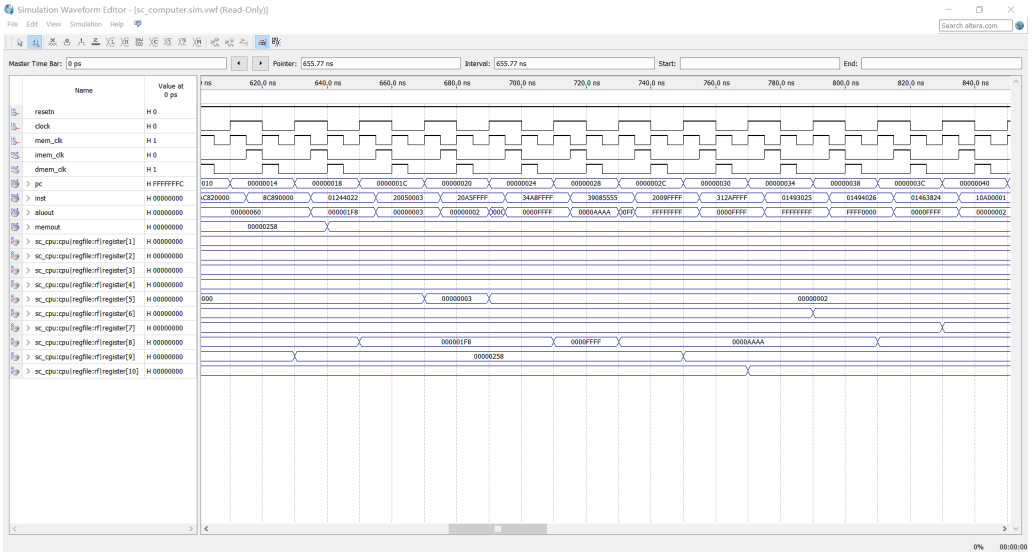
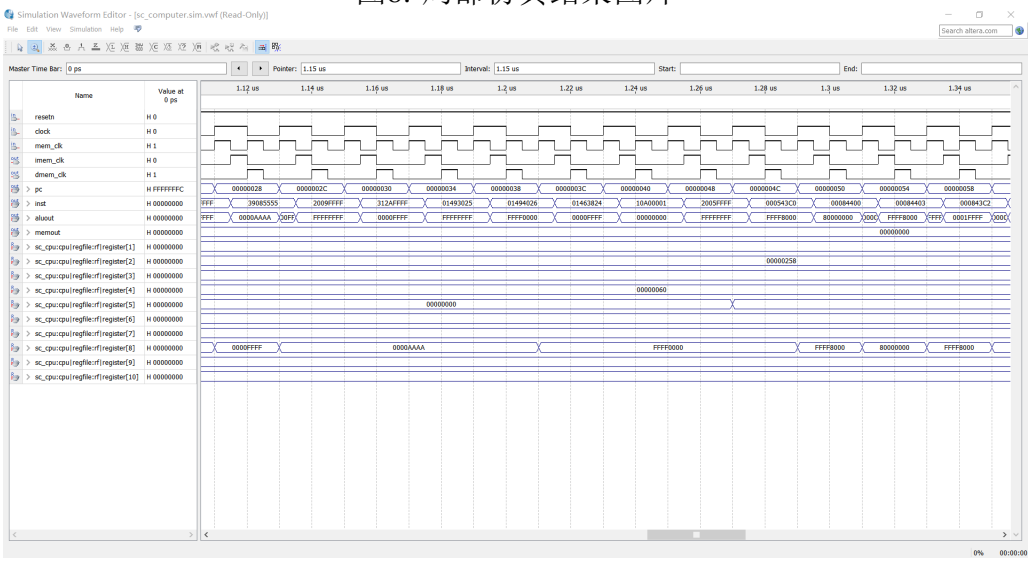


图8: 局部仿真结果图片



6 实验总结

在本次实验中，我们熟悉了Quartus仿真软件的一些基本操作，学习了Verilog的一些基础语法和编程方法，详细了解了实验中使用的每种MIPS指令的字段结构以及其对应的控制信号的值，这进一步加深了我们对于MIPS指令在CPU中是如何发挥作用的理解。此外我们再次从全局的层次上分析了CPU几大模块之间的工作过程和他们相互之间的联系，使我们对计算机

组成的学习有了更深刻更直观的印象。

非常感谢本次实验中各位老师和各位助教对我的指导和帮助!