计算机组成原理
THINPAD 大实验
实验报告

涂珂 2011011273 计 14　　傅左右 2011011264 计 14

December 12, 2013

Contents

1 实验目标

基于 THINPAD 教学计算机，设计：

- 基于 MIPS16 指令集的流水线 CPU
- 使用基本存储、扩展存储、Flash、IO 设备
- 能够运行 kernel、监控程序、project1 程序

2 指令集任务

2.1 THCO MIPS 基本指令集

序号	指令	序号	指令
1	ADDIU	14	LW_SP
2	ADDIU3	15	MFIH
3	ADDSP	16	MFPC
4	ADDU	17	MTIH
5	AND	18	MTSP
6	B	19	NOP
7	BEQZ	20	OR
8	BNEZ	21	SLL
9	BTEQZ	22	SRA
10	CMP	23	SUBU
11	JR	24	SW
12	LI	25	SW_SP
13	LW		

2.2 扩展指令集 (402)

- JRRA
- SLTI
- ADDSP3

- NOT
- SLT

3 实验成果指标

- CPU 主频为 6.25MHz（12.5MHz 有时会出一些问题，所以只能二分之，6.25MHz 是稳定频率）
- VGA 分辨率为 640*480
- **继续加啊**

4 实验成果简列

- 清晰的模块分工
- 指令集改进，指令集汇编工具
- 数据旁路
- 冒险检测
- 完整 VGA 调试工具
- FLASH 自启动
- 地址映射统一管理 IO 设备
- 串口通信
- VGA、键盘交互
  - VGA 等宽 ASCII 字符集显示
  - VGA 双端 FIFO 显存
  - 键盘输入、支持换行、发送串口与 VGA 的记事本程序

5 整体设计图

加图

6 重新设计的指令集

6.1 指令设计

- 用前 5 位表示 op。共 30 条。
- 加 * 为扩展指令。
- XXX，YYY，ZZZ 为寄存器标号。
- III 为立即数。
- 把类型相近的 op 连续起来，这样写代码就可以用大于小于判断了。

6.2 R 型指令

R	指令结构
MFIH	00001XXX00000000
MFPC	00010XXX00000000
MTIH	00011XXX00000000
MTSP	00100XXX00000000
AND	00101XXXYYY00000
OR	00110XXXYYY00000
*NOT	00111XXXYYY00000
*SLT	01000XXXYYY00000
CMP	01001XXXYYY00000
SLL	01010XXXYYYIII00
SRA	01011XXXYYYIII00
ADDU	01100XXXYYYZZZ00
SUBU	01101XXXYYYZZZ00

6.3 I 型指令

I	指令结构
ADDSP	01110IIIIIIII000
LW_SP	01111XXX00000000

ADDIU	10000XXXIIIIIIII
*SLTI	10001XXXIIIIIIII
*ADDSP3	10010XXXIIIIIIII
LI	10011XXXIIIIIIII
ADDIU3	10100XXXYYY0IIII
LW	10101XXXYYYIIIII
SW	10110XXXYYYIIIII
SW_SP	10111XXXYYYIIIII

6.4 B 型指令

B	指令结构
B	11000IIIIIIIIIII
BTEQZ	11001IIIIIIII000
BEQZ	11010XXXIIIIIIII
BNEZ	11011XXXIIIIIIII

6.5 J 型指令

J	指令结构
*JRRA	1110000000000000
JR	11101XXX00000000

6.6 NOP 指令

NOP	0000000000000000

6.7 指令流水细节

关于每一条指令在流水的五个步骤中具体做了什么。表格无法正常显示下。请见相关设计文档 `instruction.xlsx`。

7 统一的信号及编码

7.1 控制信号

每一级流水阶段的寄存器均会储存相应信号。(显然的，当前指令与当前流水信号相对应)

Table 7: 控制信号表

控制信号	发生阶段	置 0 时	置 1 时	置 10 时
PCWrite	IF	null	写 PC	
Branch(2 位)	ID	PC+4	PC+4+immediate	Reg1
ForwardA(2 位)	EX	来自寄存器堆的输出 Reg1	转发写回的值	转发上一次 ALU 运算结果
ForwardB(2 位)	EX	来自寄存器堆的输出 Reg2	转发写回的值	转发上一次 ALU 运算结果
ALUsrc	EX	来自寄存器堆的输出	来自符号扩展的立即数	
ALUop(3 位)	EX	加	001: 减, 010: 与, 011: 或, 100: 非, 101: 左移,110: 右移	
Ttype	EX	小于 T 为 1	不等于 T 为 1	
Twrite	EX	null	写入 T	
MemRead	MEM	null	读内存	
MemWrite	MEM	null	写内存	
MemtoReg	WB	写入 ALU 输出值	写入内存输入值	
RegWrite	WB	null	写入寄存器堆	

7.2 寄存器编址

我们将特殊寄存器也看作普通的寄存器。将寄存器编码长度（3 位）扩长一位，与特殊寄存器统一编址（4 位）。这样在 CPU 内部处理的时候就能简化流程。

我们设置了 SP、PC、RA、IH、Zero 四个特殊寄存器。8 个普通寄存器。特殊寄存器高位均为 1。

Table 8: 寄存器编址表

编码 4 位	寄存器
0+XXX	8 个普通寄存器
1000	Zero
1010	PC
1011	IH
1100	RA
1101	SP

7.3 字符编码

我们将 95 个 ASCII 可打印字符的等宽字符写进了 VGA 控制模块的存储里。我们以空格（0x20）为偏移量，将每一个字符对应的 ASCII 码减去 0x20 作为内部字符编码。这样就有两个好处：一是减少地址量，二是可以用连接两个 std_logic_vector 的方法（`std_logic_vector & std_logic_vector`）计算字符地址。

7.4 指令集与控制信号关系表

Table 9: 指令与控制信号关系

指令	Branch	ALU-op	ALU-src	T-type	T-write	Mem-Read	Mem-Write	Mem-to-Reg	Reg-Write
NOP	00	000	0	0	0	0	0	0	0
MFIH	00	000	0	0	0	0	0	0	1
MFPC	00	000	0	0	0	0	0	0	1
MTIH	00	000	0	0	0	0	0	0	1
MTSP	00	000	0	0	0	0	0	0	1
AND	00	010	0	0	0	0	0	0	1
OR	00	011	0	0	0	0	0	0	1
NOT*	00	100	0	0	0	0	0	0	1
SLT*	00	001	0	0	1	0	0	0	0
CMP	00	001	0	1	1	0	0	0	0
SLL	00	101	1	0	0	0	0	0	1
SRA	00	110	1	0	0	0	0	0	1
ADDU	00	000	0	0	0	0	0	0	1
SUBU	00	001	0	0	0	0	0	0	1
ADDSP	00	000	1	0	0	0	0	0	1
LW_SP	00	000	1	0	0	1	0	1	1
SW_SP	00	000	1	0	0	0	1	0	0
ADDIU	00	000	1	0	0	0	0	0	1
SLTI*	00	001	1	0	1	0	0	0	0
ADDSP3*	00	000	1	0	0	0	0	0	1
LI	00	000	1	0	0	0	0	0	1
ADDIU3	00	000	1	0	0	0	0	0	1
LW	00	000	1	0	0	1	0	1	1
SW	00	000	1	0	0	0	1	0	0
B	01	000	0	0	0	0	0	0	0
BTEQZ	01	000	0	0	0	0	0	0	0
BEQZ	01	000	0	0	0	0	0	0	0
BNEZ	01	000	0	0	0	0	0	0	0
JRRA*	10	000	0	0	0	0	0	0	0
JR	10	000	0	0	0	0	0	0	0

8 主要模块设计

8.1 硬件

8.1.1 ALU

运算逻辑单元，实现了一下 7 种运算：

ALUop	运算
000	a+b
001	a-b
010	a&b
011	a|b
100	~a
101	a << b
110	a >> b

8.1.2 BranchSelector

跳转选择器，根据不同的跳转指令及跳转条件决定 PC 是否跳转。

8.1.3 Controller

控制器，输入 Op，输出 Op 对应的控制信号（对应表见 signal.xlsx）。由于自定义了指令集使得不同的指令对应的 Op 不同且类型相近的指令相邻，让这一部分得以简化。

8.1.4 Decoder

译码器, 将 16 位指令译码为 5 位指令编号 Op, 3 个寄存器 reg1,reg2,reg3 和 16 位立即数 imm。在译码器中根据 Op 进行了立即数的扩展。3 个寄存器中 reg1，reg2 为读寄存器，reg3 为写寄存器。

8.1.5 MemoryTop

TODO

8.1.6 Passer

数据冲突检测单元，看寄存器堆取出的值是否是 EXE 或 MEM 阶段未写回的值，若是，则通过旁路引回。共引入了 4 条旁路。

8.1.7 RegFile

TODO

8.1.8 RiskChecker

控制冲突检查单元，主要是检测跳转指令所判断寄存器未写回的冲突，通过旁路引回。

8.1.9 TReg

T 寄存器，T 寄存器的值需要额外的判断写入，故将其从寄存器堆中分离出来。置于 EXE 阶段，读取 alu 的输出。
所有指令中涉及到的 T 的写入分为 2 种，不等于和大于，故我们通过 alu 做减法，不等于即为 alu 的输出不为 0，大于即为 alu 输出符号位为 0. 故可根据控制信号 Ttype,Twrite 及 alu 的输出算出 T 寄存器的值输出。

8.1.10 VGA_top

TODO

8.2 软件

8.2.1 assenmbler(thcoas.py)

在武祥晋 (2011011278) 用 python 实现的编译器的基础上进行了更改，实现了对我们自己定义的指令集的编译工作。
使用方法：`python binToHex.py a.s`, 会将 mips 汇编代码 a.s 编译输出 a.s.bin 的 2 进制文件，并在命令行上输出调试信息。

8.2.2 简单记事本 (代码见 test_keyboard.s)

将 PS2 键盘的输入直接显示到 VGA 上。

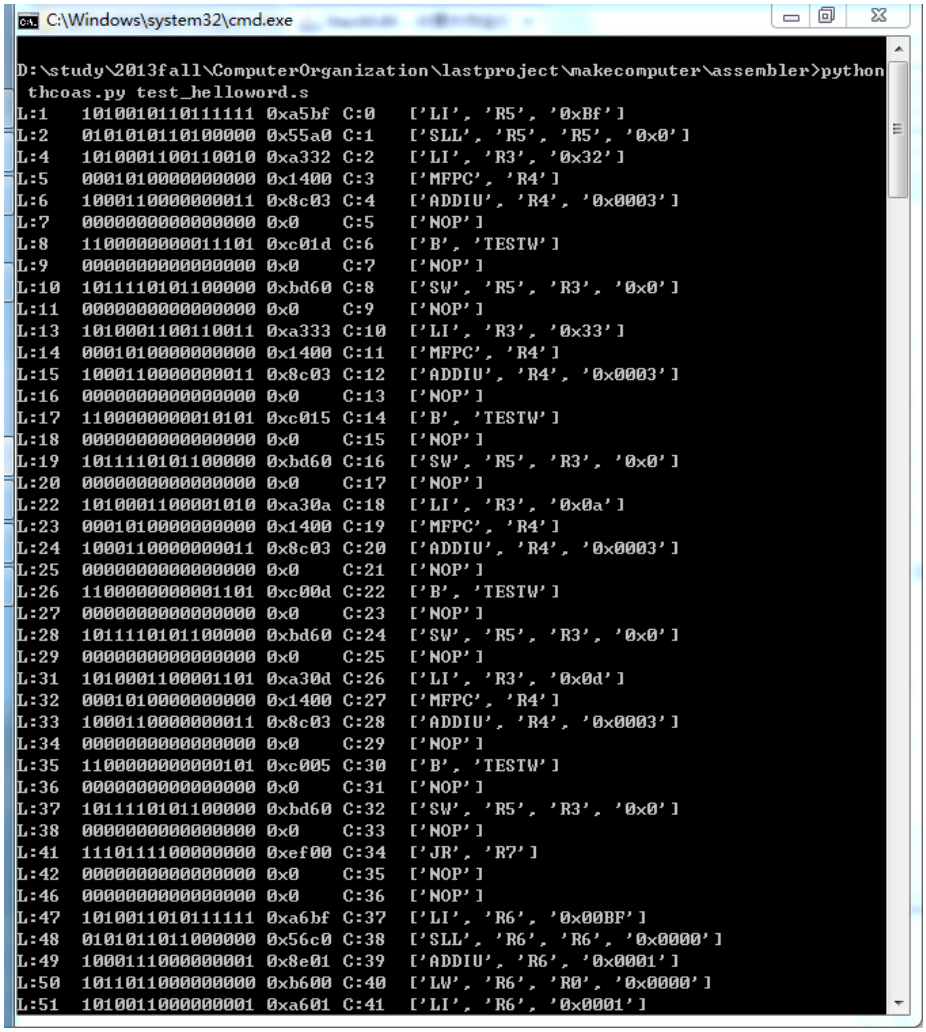

Figure 1: assembler 截图

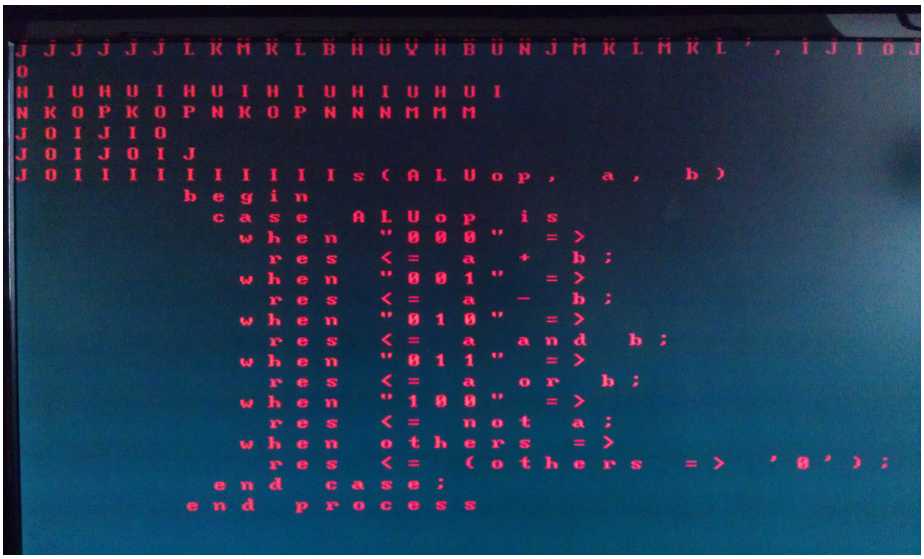

Figure 2: 记事本截图

8.3 主要模块实现

设计见上面的主要模块设计。

8.3.1 ALU

通过 ALUop 信号，通过 case…when…语句计算结果。

8.3.2 BranchSelector

对不同 Op 表示的不同的跳转指令所需的条件进行判断，输出选择器 Mux_PC 的控制信号 Branch.

8.3.3 Decoder

通过 Case 语句判断高五位的值得到指令类型，根据指令类型不同决定读写的寄存器，及立即数扩展的方式, 输出完成要读或写的寄存器 (没有则输出零寄存器)，及扩展后的立即数。

8.3.4 MemoryTop

TODO

8.3.5 Passer

根据流水寄存器的值和译码阶段的寄存器的值判断是否有数据冲突，输出控制信号。

```
if (EXMEM_RegWrite = '1' and EXMEM_W /= Zero_Reg and EXMEM_W = IDEX_R1) then
        ForwardA <= "10";
    end if;
   if (EXMEM_RegWrite = '1' and EXMEM_W /= Zero_Reg and EXMEM_W = IDEX_R2) then
        if IDEX_alusrc = '0' then
            ForwardB <= "10";
        else
            ForwardC <= "10";
        end if;
    end if;
   if (MEMWB_RegWrite = '1' and MEMWB_W /= Zero_Reg and EXMEM_W /
= IDEX_R1 and MEMWB_W = IDEX_R1) then
        ForwardA <= "01";
    end if;
   if (MEMWB_RegWrite = '1' and MEMWB_W /= Zero_Reg and EXMEM_W /
= IDEX_R2 and MEMWB_W = IDEX_R2) then
        if IDEX_alusrc = '0' then
            ForwardB <= "01";
        else
            ForwardC <= "01";
        end if;
    end if;
```

8.3.6 RegFile

TODO

8.3.7 RiskChecker

同 Passer 写法相同。

8.3.8 VGA_top

TODO

9 实验心得和体会

9.1 自定义指令集

课本上的指令集为了实现 44 条指令，不仅通过前 5 位作为 Op 判断指令类型，还需要通过其他的位才能判断具体是什么指令。这在 decoder 时，在一层 case 中要需要嵌套一层 case 语句来判断具体指令，而且在后面需要根据指令类型的运算中需要传入整个 16 位指令，不能只通过前 5 位 op 来判断。这很不方便。而我们得 cpu 只要求实现 30 条指令 (25 条基本 +5 条扩展)，因次前 5 位 op 完全足够判断指令类型，因此我们自定义了指令集。定义指令集的过程中遵守相似的指令相邻。这样就可以少写很多 case。如 controller 中对控制信号 alusrc 选择可以如下写：

```
if Op = "01010" or Op = "01011" or (Op >= "01110" and Op <= "10111") then
    ALUsrc <= '1';
else
    ALUsrc <= '0';
end if;
```

否则要通过大量 case 语句来实现，这里得到了大大的简化。

9.2 数据通路的设计

数据通路完全由一个人设计，保证整个 cpu 设计的一致性。其中数据通路共修改了 10 多次。在写代码之前就改了 7,8 次。开始写 cpu 的模块后又发现了一些问题又改了数次。后来的修改主要在旁路的设计部分，刚开始写的时候考虑不周。写代码时完全按照数据通路来写，保证程序一致。写的时候深深感受到数据通路一定要先设计好，这样才有一个好的思路，写起来也比较快。

9.3 代码同步与管理

由于是 2 个人同时写代码，所以需要代码的同步，我们使用了 git 进行同步，能很快的同步对方的代码。在修改错误时还能回滚到上一正确的版本。

9.4 debug

硬件的 debug 真是一件困难的事，它不能像软件一样输出一堆调试信息。我们只能利用有限的 LED 灯来 debug，于是我们在 RegFile 中实现了一个 debug 模块，他能实时的通过改变开关的输入来改变显示的寄存器。后来我们写好了 vga，就能输出信息在 vga 上来调试了，效率更高。
整个实现过程中遇到的最大的 bug 就是我们通过 Term 读寄存器的值时老是可能有一个寄存器里出现了随机的结果，但是我们用 led 看到寄存器里的确是正

确的。开始我们认为是我们串口写得不对，后来用仔细检查了一下发现我们串口应该没有错，那应该是执行 kernel 代码时的错，于是我们把 kennel 中读寄存器的部分单独拿出来写在一个文件了，通过手按 clk 来 debug。这里涉及到手按与自动时钟的切换，我们写了一个模块来通过开关来快速切换 2 种时钟。最后单步执行观察寄存器的值发现时 BTEQZ 的一个冲突没解决好导致多循环了一次。我们马上在 riskchecker 中多加入一条旁路程序运行正常。