

CPU 大实验 实验报告

一、实验概述

在本次 CPU 大实验，我们通过编写 VHDL 代码，在实验室提供的教学机上实现了支持五级指令流水的计算机系统，完成了包括软中断、Flash 和 VGA 在内的拓展功能，所设计、实现的 CPU 主频达到 25MHZ，性能高效且运行稳定。

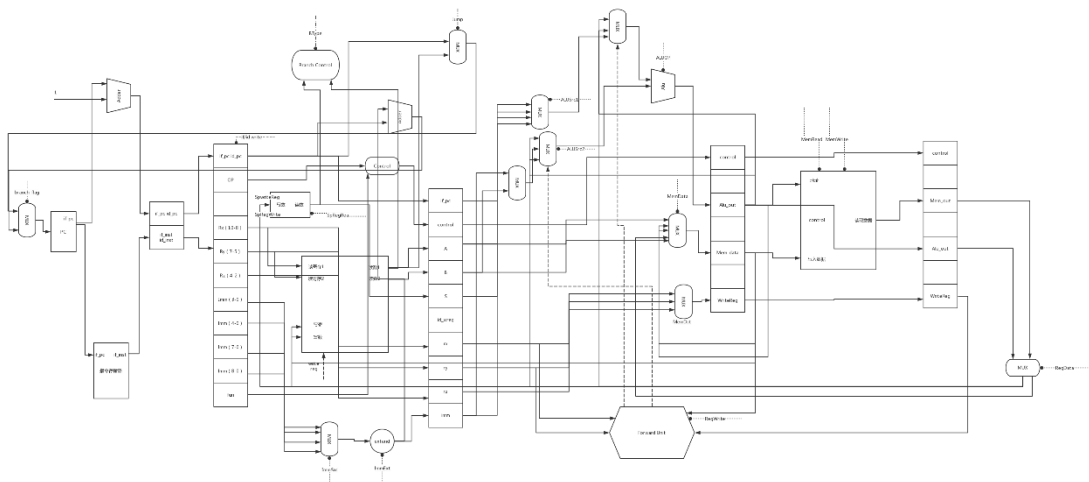
实现指令说明

我们在本次 CPU 大实验中实现了给定的 25 条基本指令和 5 条扩展指令。

其中 SRL、SRAV、CMPI、NOT 和 SLTU 为拓展指令。此外我们还实现了 INT 指令以支持软中断。

二、CPU 的设计与实现

1. 数据通路设计



2. 五级流水线设计

a) 基本结构

根据五级流水线的思想，将指令分为 IF、ID、EXE、MEM、WB 五个阶段，每个阶段通过中间寄存器 PC、IF/ID、ID/EXE、EXE/MEM、MEM/WB 获取控制信号和数据信息，每个中间寄存器分为两个状态，第一个状态更新锁存信息并传递给下一个阶段，第二个状态等待上一阶段计算好下一阶段所需要的控制信号和数据信息。各个阶段会在下一个第一个状态到来前准备好要往下一级传递的控制信号和数据信息。

b) 寄存器读写和内存访问

我们设计了专门的寄存器堆模块（Registers.vhd），内存模块（MEM.vhd）来控制寄存器和内存的读写。在内存模块中，我们使用 RAM1 来完成数据及串口的读写，RAM2 来完成指令的读写。由于设计上考虑不周的地方，我们不得不将 RAM1 的读写分为 4 个小状态来完成，因此凡涉及 LW 和 SW 的指令，CPU 主频会暂时掉到 12.5MHZ，而对于其他的指令，CPU

主频为 25MHZ

c) 冲突处理

结构冲突

在五级流水结构中，当一条指令进行访存时，最新一条指令可能正在取指。对于只有一片 RAM 的系统，此时两条指令都要访存，占用总线，结构冲突就发生了。为了处理结构冲突，我们选择使用两片 RAM 分别来读写指令（RAM2）和数据（RAM1），这样访存和取指就可以同时进行，避免了结构冲突。

控制冲突

对于跳转指令（J 型指令和 B 型指令）我们在译码阶段计算出目的地址和是否跳转的控制信号，对于跳转发生时可以选择锁存 IF/ID 寄存器以不执行延迟槽，或者不锁存 IF/ID 以执行延迟槽。根据标准 MIPS 的规范，我们选择在跳转发生时执行延迟槽内的指令。

数据冲突

由于将跳转指令目的地址的计算提前到 ID 段，因此在 ID 段和 EXE 段都会发生数据冲突。我们设计了专门的数据转发单元，来生成数据前推的控制信号。经过分析，只有当 ID 段指令需要用到上条指令的访存结果时（比如 LW 指令后紧跟一条 JR 指令）需要插入气泡，暂停流水线，其他的数据冲突均可以通过数据前推解决，基本消除了数据冲突带来性能损失。具体数据冲突处理如下：

1.ID 段用上条指令的 ALU 结果

解决方法：从 EXE 的 ALU 引旁路到 ID 段对应的位置，根据控制信号进行选择

2.ID 段用上条指令的 MEM 读取结果

解决办法:插入气泡（仅需锁存 IF/ID 寄存器一个 CPU 周期），从 MEM 的读取数据引旁路到对应位置，根据控制信号进行选择

3.EXE 段用上条指令的 ALU 结果

解决办法：从 EXE/MEM 的 ALUOUT 引旁路到对应位置，根据控制信号进行选择

4.EXE 段用上条指令的 MEM 读取结果

解决办法：从 MEM 的读取数据引旁路到对应位置，根据控制信号进行选择

5.ID 段用上上条指令的 ALU 结果

解决办法：从 EXE/MEM 的 ALUOUT 引旁路到对应位置，根据控制信号进行选择

6.ID 段用上上条指令的 MEM 读取结果

解决办法：从 MEM 的读取数据引旁路到对应位置，根据控制信号进行选择

7.EXE 段用上上条指令的 MEM 读取结果或 EXE 段用上上条指令的 ALU 结果

解决办法：均从 WB 的多路选择结果引旁路到对应位置，根据控制信号进行选择

其中数据冲突相关的控制信号的产生由 ForwardUnit 来完成(数据转发模块 ForwardUnit.vhd)

d) 各个阶段、中间寄存器及功能模块的具体实现

中间寄存器 PC

状态 0：等待 ID 段完成指令控制信号和目的地址的计算

状态 1：通过控制信号准备好下一条指令的地址和生成是否恢复中断的控制信号。若译码阶段指令为 JR R6，且当前处于软件中断状态，则生成软中断恢复信号。若需要暂停则保持 PC 不变。若需要 B 型或者 J 型跳转，则把 PC 置于计算好的目的地址。若 id 段为 INT 指令，则将 PC 置于中断处理函数的入口地址。

IF 段（在 MEM.vhd 中通过 RAM2 的读取实现）

状态 0: 将 PC 在状态 2 准备好的指令地址传给 RAM2 的地址总线, 并置总线为相应的状态以读取指令。

状态 1: 准备好读取的指令供下一阶段使用。

中间寄存器 IF/ID

状态 0: 将锁存的指令和下一条指令地址更新为 IF 段准备的指令和 PC 段准备的下一条指令地址, 供 ID 段使用。若需要暂停则保持指令和下一条指令地址不变。

状态 1: 等待。

ID 段

通过 IF/ID 寄存器提供的指令译码, 产生每条指令相应的控制信号。结合数据转发单元 (ForwardUnit) 的控制信号和该指令译码所得的控制信号对要读取的通用寄存器和特殊寄存器的值进行多路选择, 选择出该阶段所需的最新寄存器值。将准备好的控制信号和数据传给 ID/EXE 中间寄存器。

中间寄存器 ID/EXE

状态 0: 将锁存的控制信号和数据信息更新为 ID 段准备的控制信号和数据信息, 供 EXE 段使用。若需要暂停则保持控制信号和数据信息不变。

状态 1: 等待

EXE 段

根据转发单元和 ID/EXE 段的控制信号进行 ALU 操作数 A、ALU 操作数 B 的选择以及 ALU 运算, 写入内存数据选择、写回寄存器地址选择。

中间寄存器 EXE/MEM

状态 0: 将锁存的控制信号和数据信息更新为 EXE 段准备的控制信号和数据信息, 供 MEM 段使用。若需要暂停则保持控制信号和数据信息不变。

状态 1: 等待。

MEM 段

对于内存数据的读写操作, MEM 段分为 4 状态来操作 RAM1, 从而需要插入一个气泡暂停流水线 (将所有中间寄存器的值保持一个 CPU 周期)。

对于指令的读写操作, 需要在取指完成后, 新增两个状态来操作 RAM2 以读写指令, 因此同样需要插入一个气泡暂停流水线 (将所有中间寄存器的值保持一个 CPU 周期)。

根据指令系统的设计, 和地址划分, 可以通过地址段来判断所进行的访存操作是读写指令 (0x4000-0x7FFF)、内存数据(0xBF04-0xFFFF 或 0x8000-0xBEFF)、串口(0xBF00 或 0xBF02)还是查看串口状态(0xBF01 或 0xBF03)。

具体实现如下:

RAM1 的读写

状态 0: 将总线信号置于初始状态, 并等待访存地址的更新。

状态 1: 若是读、写串口, 则将数据总线至为高阻, OE、WE、EN、rdn、wrn 均置 1; 若是要读取串口状态, 则将串口的各个状态量置于对应的位中; 若是读内存数据, 则置 OE、EN 为 0, WE 为 1, 数据总线为高阻, 并将目的地址赋给地址总线; 若是写内存数据, 则将地址和数据赋值给地址总线 and 数据总线, 并将 WE 拉低, 以完成写入。

状态 2: 若是读串口, 将 `rdn` 拉低至 0, 以将数据读取到数据总线; 若是读取内存数据, 此时数据总线的的数据已为要读取的值, 输出即可; 若是写串口, 则将 `wrn` 拉至 0, 并将要写的数据赋值给数据总线。

状态 3: 对于读指令, 此时数据总线已经是读取好的数据, 将其赋值给指定变量以准备给下一阶段使用; 对于写指令, 此时写操作也已经完成。

RAM2 的指令读写

状态 0: 等待取指阶段的指令完成取指。

状态 1: 等待取指阶段的指令完成取指。

状态 2: 若是读取指令, 则将 `oe` 置 0, 数据总线置为高阻, 将读取的指令地址赋给地址总线; 若是写指令, 则将地址和指令赋值给地址总线 and 数据总线, 并将 `we` 拉低, 以完成写入。

状态 3: 若是读取指令, 此时数据总线便是读取的指令, 将其赋值给指定变量以准备给下一阶段使用; 对于写入指令, 此时写操作也已经完成。

此外由于访存需要 4 个状态, 所以需要在状态 1 根据该条指令是否有内存读写操作生成流水线暂停信号, 决定是否将所有中间寄存器的值保持一个 CPU 周期。

中间寄存器 MEM/WB

状态 1: 将锁存的控制信号和数据信息更新为 MEM 段准备的控制信号和数据信息, 并根据控制信号选择要写回的数据, 供 WB 段使用。若需要暂停则保持控制信号和数据信息不变。

状态 2: 等待。

WB 段 (在 Registers.vhd 中实现)

状态 1: 等待 MEM/WB 中间寄存器更新好要写回的数据和控制信号。

状态 2: 通过控制信号更新寄存器值。若要恢复软中断, 则将之前保存的 `r0`, `r1` 寄存器值写回。若要进入软中断, 则保存 `r0`, `r1` 寄存器的值并将中断号和中断指令地址传入 `r0`, `r1`。

ForwardUnit (数据转发单元 ForwardUnit.vhd)

EXE 段对操作数 A 的选择控制

若 ALU 所用的第一个操作数来自 `RX` 且 EXE/MEM 段的指令要写回 `RX`, 则选择 ALU 结果或者 MEM 读取结果。

若 ALU 所用的第一个操作数来自 `RX` 且 MEM/WB 段的指令要写回 `RX`, 则选择 WB 段的写回数据。

若 ALU 所用的第一个操作数来自特殊寄存器 `SP` 且 EXE/MEM 段的指令要写回 `SP`, 则选择 ALU 结果或者 MEM 读取结果。

若 ALU 所用的第一个操作数来自特殊寄存器 `SP` 且 MEM/WB 段的指令要写回特殊寄存器 `SP`, 则选择 WB 段的写回数据。

其余情况不前推。

EXE 段对操作数 B 的选择控制

若 ALU 所用的第二个操作数来自 `RY` 且 EXE/MEM 段的指令要写回 `RY`, 则选择 ALU 结果或者 MEM 读取结果。

若 ALU 所用的第二个操作数来自 `RY` 且 MEM/WB 段的指令要写回 `RY`, 则选择 WB 段的

写回数据。

EXE 段对写入内存数据的选择控制

若选择的数据来自 RY，且 EXE/MEM 段的指令要写回 RY，则选择 ALU 结果或者 MEM 读取结果。

若选择的数据来自 RY 且 MEM/WB 段的指令要写回 RY，则选择 WB 段的写回数据。

若选择的数据来自 RX，且 EXE/MEM 段的指令要写回 RX，则选择 ALU 结果或者 MEM 读取结果。

若选择的数据来自 RX 且 MEM/WB 段的指令要写回 RX，则选择 WB 段的写回数据。

ID 段跳转指令对 RX 寄存器值的选择

指令为 J 型指令或者 B 型指令，且 EXE 段要写回的通用寄存器 RX 与 ID 段要访问的通用寄存器 RX 相同。若此时要写回的数据来自 ALU 计算结果，则选择计算结果；若此时要写回的数据来自 ALU 计算出的地址所存储的数据，则需要产生暂停流水线信号。

若 ID 段指令为 J 型指令或者 B 型指令，且 EXE/MEM 段要写回的目的地寄存器与 ID 段要访问的寄存器相同。则选择 ALU 结果或者 MEM 读取结果。

ID 段跳转指令对特殊寄存器 SP 值的选择

若 ID 段指令为 J 型指令或者 B 型指令，且 EXE 段要写回的特殊寄存器 SP 与 ID 段要访问的特殊寄存器 SP 相同。若此时要写回的数据来自 ALU 计算结果，则选择计算结果；若此时要写回的数据来自 ALU 计算出的地址所存储的数据，则需要产生暂停流水线信号。

若 ID 段指令为 J 型指令或者 B 型指令，且 EXE/MEM 段要写回的目的地寄存器与 ID 段要访问的寄存器相同。则选择 ALU 结果或者 MEM 读取结果。

ImmExtend(立即数扩展单元 Immextend.vhd)

根据 ID 段产生的立即数控制信号，从指令上选取对应的位数，并根据控制信号进行符号扩展或者无符号扩展。

三、控制信号说明

如下，(具体可见“信号表.xlsx”)

[illegible]

A	B	C
ALUOp	ALUOutput	对应指令
0000	ALUInput1 + ALUInput2	ADDU / ADDIU3 / ADDIU / ADDSP/LW / SW / LW_SP / SW_SP
0001	ALUInput1 - ALUInput2	SUBU
0010	ALUInput1 & ALUInput2	AND
0011	ALUInput1 ALUInput2	OR
0100	~ALUInput2	NOT
0101	ALUInput1	MTSP LI MFPC MFIH MTIH
0110	ALUInput2	
0111	(ALUInput1 != ALUInput2) ? 1 : 0 ;	CMP CMPI
1000	(无符号比较)(ALUInput1 < ALUInput2) ? 1 : 0 ;	SLTU
1001	(ALUInput1(4-2) == 0 ?) ALUInput2 << 8 : ALUInput2 << ALUInput1(4-2)	SLL
1010	(ALUInput1(4-2) == 0 ?) ALUInput2 >> 8 : ALUInput2 >> ALUInput1(4-2)	SRL
1011	(ALUInput1(4-2) == 0 ?) ALUInput2 >> 8 : ALUInput2 >> ALUInput1(4-2)(算数右移动)	SRA
1100	ALUInput2 >> ALUInput1 (算数右移)	SRAV

四、指令分析

如下图，(具体可见“信号表.xlsx”)

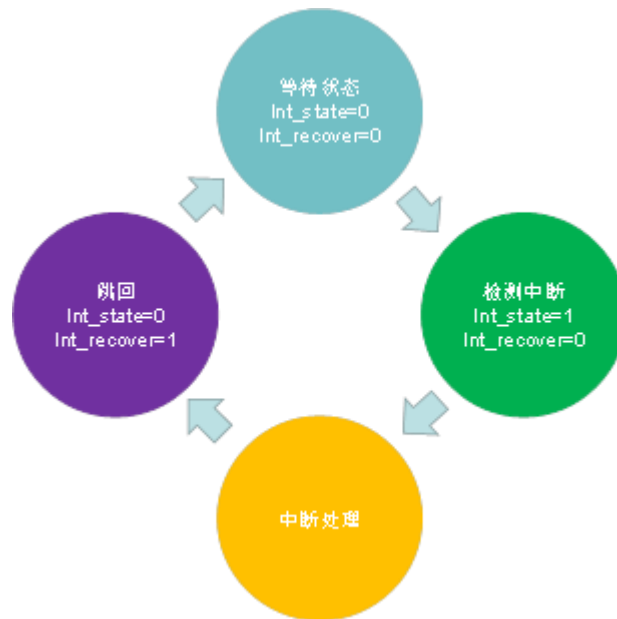
#	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1	指令	操作码	功能码	Branch	BType	Jump	InnSrc	InnExt	ALUSrcA	ALUSrcE	MemRead	MemWrite	MemData	RegWrite	WriteSpReg	RegData	RegDst	SpRegWrite	SpRegRead	ALUOP
2	NOP	00001	XX	0	XX	0	XX	X	XX	X	0	0	X	0	0	X	XX	XX	XX	XXXX
3	B	00010	XX	1	00	0	XX	X	XX	X	0	0	X	0	0	X	XX	XX	XX	XXXX
4	BEQZ	00100	XX	1	01	0	10	1	XX	X	0	0	X	0	0	X	XX	XX	XX	XXXX
5	BNEZ	00101	XX	1	10	0	10	1	XX	X	0	0	X	0	0	X	XX	XX	XX	XXXX
6	SLL	00110	00(1 0)	0	XX	0	01	X	10	0	0	0	X	1	0	0	00	XX	XX	1001
7	SRA	00110	11	0	XX	0	01	X	10	0	0	0	X	1	0	0	00	XX	XX	1011
8	SRL	00110	10	0	XX	0	01	X	10	0	0	0	X	1	0	0	00	XX	XX	1010
9	ADDIU3	01000	XX	0	XX	0	00	1	00	1	0	0	X	1	0	0	01	XX	XX	0000
10	ADDIU	01001	XX	0	XX	0	10	1	00	1	0	0	X	1	0	0	00	XX	XX	0000
11	ADDSP	01100	011(10 8)	0	XX	0	10	1	01	1	0	0	X	0	1	0	XX	00	00	0000
12	BTBZ	01100	000	1	11	0	10	1	XX	X	0	0	X	0	1	0	XX	XX	XX	XXXX
13	MTSP	01100	100	0	XX	0	XX	X	XX	X	0	0	X	0	1	0	XX	10	XX	0101
14	LI	01101	XX	0	XX	0	10	0	00	X	0	0	X	1	0	0	00	XX	XX	0101
15	CMPI	01110	XX	0	XX	0	10	1	01	1	0	0	X	0	1	0	XX	00	XX	0111
16	LW_SP	10010	XX	0	XX	0	10	1	01	1	1	0	X	1	0	1	00	XX	10	0000
17	LW	10011	XX	0	XX	0	01	1	00	1	1	0	X	1	0	1	01	XX	XX	0000
18	SW_SP	11010	XX	0	XX	0	10	1	01	1	1	0	0	0	0	X	XX	XX	10	0000
19	SW	11011	XX	0	XX	0	01	1	00	1	1	0	0	0	0	X	XX	XX	XX	0000
20	ADDU	11100	01(1 0)	0	XX	0	XX	X	00	0	0	0	X	1	0	0	10	XX	XX	0000
21	SUBU	11100	11	0	XX	0	XX	X	00	0	0	0	X	1	0	0	10	XX	XX	0001
22	AND	11101	01100(4 0)	0	XX	0	XX	X	00	0	0	0	X	1	0	0	00	XX	XX	0010
23	CMP	11101	01010	0	XX	0	XX	X	00	0	0	0	X	0	1	0	XX	01	XX	0111
24	MOT	11101	01111	0	XX	0	XX	X	XX	0	0	0	X	1	0	0	00	XX	XX	0100
25	OR	11101	01101	0	XX	0	XX	X	00	0	0	0	X	1	0	0	00	XX	XX	0011
26	SLTU	11101	00011	0	XX	0	XX	X	00	0	0	0	X	0	1	0	XX	01	XX	1000
27	SRAV	11101	00011	0	XX	0	XX	X	00	0	0	0	X	1	0	0	01	XX	XX	1100
28	TR	11101	00000000(7 0)	0	XX	1	XX	X	XX	X	0	0	X	0	0	0	XX	XX	XX	XXXX
29	MFPC	11101	01000000	0	XX	0	XX	X	11	X	0	0	X	1	0	0	00	XX	XX	0101
30	MFIH	11110	00000000	0	XX	0	XX	X	01	X	0	0	X	1	0	0	00	XX	10	0101
31	MTIH	11110	00000001	0	XX	0	XX	X	00	X	0	0	X	0	1	0	XX	01	XX	0101
32																				
33																				

五、拓展功能实现

软件中断（INT 指令）

软件中断的实现机理为，通过 INT 指令将 PC 置于软件中定义的中断处理函数的第一条指令的地址，从而进入中断处理函数，中断处理函数在执行前会先将所有寄存器的值保存在特定的内存块中，并根据进入 INT 指令的中断号进行相应操作，中断退出时，中断处理函数会将内存数据读入相应寄存器以恢复寄存器状态，并跳转到进入中断的指令的下一条指令继续执行。kernel 中的中断处理函数根据 R0、R1 来获取中断号和中断恢复时的下一条指令地址，所以在进入中断前，要先通过硬件缓存 R0 与 R1 的值，并将中断号和恢复地址赋值给 R0、R1，通过译码来判断是否从中断中恢复，从而将缓存的 R0、R1 值复原，以继续执行程序。

具体状态机如下（取自计 23 班谢晓晖组）：



具体实现：

在中间寄存器 PC 的状态 1，通过 ID 段得到的控制信号判断这是否为一个中断指令(INT)，如果是中断指令，则取出中断号，将指令地址 PC 赋值为中断处理程序的开始地址“0000000000000101”（根据 kernel 所得），并将 int_state 置为 1 以表示进入中断状态。

硬件代码如下：

```

elseif IntModule_i = '1' then
  --标志 int 状态
  int_state <= '1';
  --获取中断信号
  int_signal_o <= "000000000000" & if_id_ins_i(3 downto 0);
  --缓存 pc
  int_buffer_pc <= pc ;--+ "0000000000000001";
  int_pc_o      <= pc ;--+ "0000000000000001";
  --准备好中断指令地址
  pc <= "0000000000000110";
  if_pc <= "0000000000000101";

```

else..

在中断指令的写回阶段，先将 R0 与 R1 寄存器值缓存，再将中断号与恢复时的指令地址存入 R0、R1。并将 SP 置为“1011111100010000”（观察 kernel 所得）以保证各个寄存器的值的恢复。

硬件代码如下：

```

--进入中断保存现场
if mem_wb_int_module_i = '1' then
  int_buffer_r0 <= r0;
  int_buffer_r1 <= r1;
  SP <= "1011111100010000";
  r0 <= int_signal_i;
  r1 <= int_buffer_pc_i;

```

```
end if;
```

若 ID 段指令为 JR R6，则说明中断处理函数执行完毕，需要跳转到 INT 指令的下一条指令。此时将 int_recover 置 1 以提醒寄存器堆恢复 R0、R1 的值，并将 int_state 置 0 以表示退出中断。至此，软中断处理完毕。

硬件代码如下：

```
-- INT 恢复，ID 段为 JR R6
if jump_i = '1' and if_id_ins_i(10 downto 8) = "110" and int_state = '1' then
    int_recover_o <= '1';
    int_state <= '0';
else
    int_recover_o <= '0';
end if;
--恢复中断时要写回 r0,r1
if int_recover_i = '1' then
    r0 <= int_buffer_r0;
    r1 <= int_buffer_r1;
end if;
```

FLASH 实现 kernel 自动启动

由于 FLASH 存储的数据在断电后不会被擦除，所以可以先通过课程提供的烧入软件将 Kernel 的 2 进制代码烧入 FLASH 的指定地址块。当 CPU 的硬件代码被烧入 FPGA 并启动后，硬件代码逻辑会先从 FLASH 将指令读入 RAM2（指令存储器），当读入完成后，生成一个完成信号，以初始化 CPU 的各个寄存器和流水线寄存器以及控制状态，并从 RAM2 的第一条指令开始逐条执行。由此，将断电后需要先将指令烧入 RAM2 再将 CPU 的硬件代码烧入 FPGA 的两次烧入操作缩减为 1 次。

具体 FLASH BOOT 实现：

```
elsif flash_complete = '0' then
    --boot
    case flash_now_s is
        --从 flash 读取数据
        when s0 =>
            flash_CE <= '0';
            flash_WE <= '1';
            flash_OE <= '1';
            flash_byte <= '1';
            flash_vpen <= '1';
            flash_rp <= '1';
            flash_addr <= "000000" & pro_addr & '0';
            flash_data <= "ZZZZZZZZZZZZZZZZ";

            flash_now_s <= s1;
        --准备写 ram2
        when s1 =>
            ram2_en <= '0';
```



```

        ram2_oe <= '1';
        ram2_we <= '1';
        ram2_addr <= pro_addr;

        flash_now_s <= s2;
        --等待 flash 读取完毕
    when s2 =>
        flash_OE <= '0';

        flash_now_s <= s3;
        --将 flash 读取数据传给 ram2 数据总线
    when s3 =>
        ram2_data <= flash_data;

        flash_now_s <= s4;
        --等待
    when s4 =>
        flash_OE <= '1';

        flash_now_s <= s5;
        --ram2 拉低 WE 以完成写入
    when s5 =>
        ram2_we <= '0';

        flash_now_s <= s6;
        --ram2 复位 WE
    when s6 =>
        ram2_we <= '1';

        flash_now_s <= s7;
        --计算下一条要存取的指令地址
    when s7 =>
        pro_addr <= pro_addr + '1';
        pro_addr_o <= pro_addr;
        flash_now_s <= s0;
    when others =>
end case;
--读取地址超过指定值则说明 boot 完成
if pro_addr > x"0220" then
    flash_complete <= '1';
    flash_complete_o <= '1';
else
    flash_complete <= '0';
    flash_complete_o <= '0';

```

```
        end if;  
    end if;
```

VGA 显示寄存器状态

功能：通过显示器显示出各个寄存器的值以便调试。

实现：每 16*16 像素存放一个字符 以如下形式先将需要的字符画出来（1 表示有颜色，0 表示无），在扫描时通过 x,y 坐标定位，将对应的二维数组投影到对应的位置。并根据 0，1 值将 RGB 赋值。

```
type matrix IS array (15 downto 0) of std_logic_vector (15 downto 0);  
signal zero : matrix := (  
    "0000000000000000",  
    "0000000000000000",  
    "0000011111100000",  
    "0000010000100000",  
    "0000010000100000",  
    "0000010000100000",  
    "0000010000100000",  
    "0000010000100000",  
    "0000010000100000",  
    "0000010000100000",  
    "0000011111100000",  
    "0000000000000000",  
    "0000000000000000",  
    "0000000000000000",  
    "0000000000000000",  
    "0000000000000000"  
);
```

以下是 VGA 模块的输入输出 r0 到 pc 为输入的寄存器值，RGB 为颜色分量，Hs 与 Vs 为行同步信号与场同步信号

```

entity vga is
  Port(
    -- common port
    CLK: in std_logic;
    RST: in std_logic;
    -- data
    r0_i      : in  STD_LOGIC_VECTOR (15 downto 0);
    r1_i      : in  STD_LOGIC_VECTOR (15 downto 0);
    r2_i      : in  STD_LOGIC_VECTOR (15 downto 0);
    r3_i      : in  STD_LOGIC_VECTOR (15 downto 0);
    r4_i      : in  STD_LOGIC_VECTOR (15 downto 0);
    r5_i      : in  STD_LOGIC_VECTOR (15 downto 0);
    r6_i      : in  STD_LOGIC_VECTOR (15 downto 0);
    r7_i      : in  STD_LOGIC_VECTOR (15 downto 0);
    T_i       : in  STD_LOGIC_VECTOR (15 downto 0);
    IH_i      : in  STD_LOGIC_VECTOR (15 downto 0);
    SP_i      : in  STD_LOGIC_VECTOR (15 downto 0);
    PC_i      : in  STD_LOGIC_VECTOR (15 downto 0);
    -- vga port
    R: out std_logic_vector(2 downto 0) := "000";
    G: out std_logic_vector(2 downto 0) := "000";
    B: out std_logic_vector(2 downto 0) := "000";
    Hs: out std_logic := '0';
    Vs: out std_logic := '0'
  );
end vga;

```

具体效果见 VGA.mp4 文件

六、实验成果

性能测试

1、性能标定（运行实验提供的执行延迟槽版本的测试代码）

这段程序一般没有数据冲突和结构冲突，可作为性能标定。共 1.50 亿条指令。

运行时间：6s

2、运算数据冲突的效率测试

从这一节起，假设正确处理了数据冲突，有数据冲突的地方不再加 NOP。共 2.25 亿条指令。

运行时间：9s

3、控制指令冲突测试

从这一节起，假设正确处理了延迟槽，行为与模拟器一样，延迟槽里可能填充语句。共 1.00 亿条指令。

运行时间：4s

4、访存数据冲突性能测试

共 1.50 亿条指令。

运行时间：10s

5、读写指令存储器测试

共 0.75 亿条指令。

运行时间：4s

根据测试代码的运行结果可以计算出，对于非访存指令，CPU 主频稳定在 25MHZ，对于连续访存指令，CPU 主频会介于 12.5MHZ（访存暂停一个 CPU 周期）与 25MHZ 之间，与设计预期相符。

拓展指令

本次实验，我们实现了 5 条扩展指令 SRL、SRAV、CMPI、NOT 和 SLTU，我们为每条扩展指令编写了相应的测试代码并计算了预期的测试结果，通过与实验结果比对证明所有指令均正确执行。

1. SRL

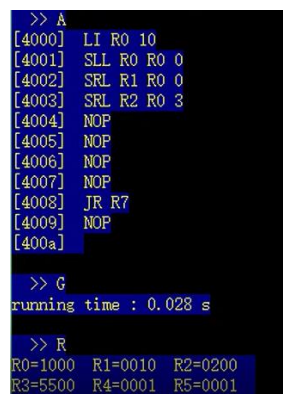
测试代码：

```
LI R0 10
SLL R0 R0 0
SRL R1 R0 0
SRL R2 R0 3
NOP
NOP
NOP
NOP
JR R7
NOP
```

代码功能：R0=0xFF00，逻辑右移动 8 位、0 位分别存入 R1，R2

预期结果 R0=0x1000、R1=0x0010、R2=0x0200

实验结果：



```
>> A
[4000] LI R0 10
[4001] SLL R0 R0 0
[4002] SRL R1 R0 0
[4003] SRL R2 R0 3
[4004] NOP
[4005] NOP
[4006] NOP
[4007] NOP
[4008] JR R7
[4009] NOP
[400a]

>> G
running time : 0.028 s

>> R
R0=1000 R1=0010 R2=0200
R3=5500 R4=0001 R5=0001
```

2. SRAV

测试代码：

```
LI R1 FF
SLL R1 R1 0
LI R2 2
SLL R2 R2 0
LI R0 1
ADDIU R0 1
```

```

LI R0 1
SRAV R0 R1
SRAV R0 R2
NOP
NOP
NOP
NOP
NOP
NOP
NOP
JR R7
NOP

```

代码功能：R1=0xFF00、R2=0x0200 算数右移动 1 位

预期结果：R1=0xFF80、R2=0x0100

实验结果：



```

[4000] LI R1 FF
[4001] SLL R1 R1 0
[4002] LI R2 2
[4003] SLL R2 R2 0
[4004] LI R0 1
[4005] ADDIU R0 1
[4006] LI R0 1
[4007] SRAV R0 R1
[4008] SRAV R0 R2
[4009] NOP
[400a] NOP
[400b] NOP
[400c] NOP
[400d] NOP
[400e] NOP
[400f] JR R7
[4010] NOP
[4011]

>> G
running time : 0.028 s

>> R
R0=0001 R1=ff80 R2=0100
R3=5500 R4=0001 R5=0001

```

3. CMPI

测试代码：

```

LI R0 FF
SLL R0 R0 0
LI R1 1
NOP
CMPI R1 2
BTEQZ 2
LI R2 FF
LI R2 00
NOP
NOP
NOP
JR R7
NOP

```

代码功能：判断 R1 是否为 2，若为 2 则跳转 R2=0xFF，否则不跳转 R2=0x00

预期结果：R1=0x1、R2=0x00

实验结果:

```
[4000] LI R0 FF
[4001] SLL R0 R0 0
[4002] LI R1 1
[4003] NOP
[4004] CMPI R1 2
[4005] BTEQZ 2
[4006] LI R2 FF
[4007] LI R2 00
[4008] NOP
[4009] NOP
[400a] NOP
[400b] JR R7
[400c] NOP
[400d]

>> G
running time : 0.028 s

>> R
R0=ff00 R1=0001 R2=0000
R3=5500 R4=0001 R5=0001
```

4. NOT

测试代码:

```
LI R0 5
NOT R1 R0
NOP
NOP
NOP
NOP
JR R7
NOP
```

代码功能: R0=0x5, 取反存入 R1

预期结果: R0=0x5 、R1=0xFFFA

实验结果:

```
>> A
[4000] LI R0 5
[4001] NOT R1 R0
[4002] NOP
[4003] NOP
[4004] NOP
[4005] NOP
[4006] JR R7
[4007] NOP
[4008]

>> G
running time : 0.028 s

>> R
R0=0005 R1=fffa R2=0000
R3=5500 R4=0001 R5=0001
```

5. SLTU

测试代码:

```
LI R0 FF
SLL R0 R0 0
LI R1 2
NOP
```

```

SLTU R0 R1
BTEQZ 2
LI R2 FF
LI R2 00
NOP
NOP
NOP
JR R7
NOP

```

代码功能: 若 R0=0xFF00 小于 R1=0x2(无符号比较), 则不跳转 R2=0x00, 否则跳转 R2=0xFF

预期结果: R0=0xFF00、R1=0x0002、R2=0x00FF

实验结果:

```

>> A
[4000] LI R0 FF
[4001] SLL R0 R0 0
[4002] LI R1 2
[4003] NOP
[4004] SLTU R0 R1
[4005] BTEQZ 2
[4006] LI R2 FF
[4007] LI R2 00
[4008] NOP
[4009] NOP
[400a] NOP
[400b] JR R7
[400c] NOP
[400d]

>> G
running time : 0.028 s

>> R
R0=ff00 R1=0002 R2=00ff
R3=5500 R4=0001 R5=0001

```

软件中断测试

本次实验, 我们实现了 INT 指令以完成软件中断, 我们编写了相应的测试代码, 通过验证 R0、R1 在中断处理前后没有变化和 Term 输出中断提醒, 来验证软件中断正确执行, 并在执行完毕后正确恢复寄存器状态。

测试代码:

```

LI R0 1
LI R1 2
NOP
NOP
INT 0
NOP
NOP
NOP
ADDU R0 R1 R2
NOP
NOP
NOP
INT 0
NOP

```

NOP
NOP
JR R7
NOP

代码功能：第一次中断前 R0=0x1、R1=0x2，第二次中断前 R2=R1+R2=0x3，两次中断处理完毕后 R0=0x1、R1=0x2、R2=0x3

预期结果：R0=0x1、R1=0x2、R2=0x3

实验结果：

```
[4000] LI R0 1
[4001] LI R1 2
[4002] NOP
[4003] NOP
[4004] INT 0
[4005] NOP
[4006] NOP
[4007] NOP
[4008] ADDU R0 R1 R2
[4009] NOP
[400a] NOP
[400b] INT 0
[400c] NOP
[400d] NOP
[400e] NOP
[400f] JR R7
[4010] NOP
[4011]

>> G
int 指令中断
int 指令中断
running time : 0.041 s

>> R
R0=0001 R1=0002 R2=0003
R3=5500 R4=0001 R5=0001
>>
```

Flash boot 展示

我们通过 FLASH 实现了自动 boot kernel 的功能，但是该功能不易于截图展示，可以通过视频，或者直接烧入 FPGA 感受实现效果。

VGA 展示

我们通过 VGA 实现了一个简单的寄存器状态显示功能。能够在显示器上实时显示当前各个寄存器的状态值，方便调试。

效果展示：见 VGA 展示.mp4

七、实验总结

通过本次 CPU 大实验，我们加深了对流水线 CPU 的理解，对于课堂与书本的知识有了更加深刻的认识，对如何实现流水线 CPU 形成了较为完整的知识体系。

本次实验，我们以课堂知识为基础，结合《自己动手写 CPU》的流水线设计框架，并参考了往年学长的代码和经验（计 23 鲁逸沁，12.5MHZ、计 33 古裔正，12.5MHZ、计 31 刘志峰，8.34MHZ），在较短时间内实现了 25MHZ 的流水线 CPU。通过实验，我们认识到实现 CPU 的关键在于数据通路（数据流图）的设计，而要设计出合理、高效的数据通路则要对 CPU 的各方面原理、瓶颈有着充分的理解，以流水线 CPU 为例，这其中包括中间寄存器的设计（如

何锁存、更新)、控制信号的产生、冲突处理(结构冲突时如何暂停、控制冲突时如何处理、数据冲突时如何进行数据前推)和访存管理(如何读写内存数据、如何读写指令、如何读写串口、如何查看串口状态)等。在设计过程中,我们借鉴前几届学长的经验并结合自己的思考,通过将各个流水段划分成几个小的状态机来保证中间寄存器的正确锁存和更新;通过详细的指令分析,设计了完整的“指令-控制信号表”来生成控制信号;通过使用两片 RAM(RAM1 和 RAM2)来避免“取指”和“访存”存在的结构冲突;通过将跳转指令的计算提前至 ID 段和执行延迟槽,来减小控制冲突带来的性能损失;通过详细分析各个指令的组合,列出所有可能的数据冲突和解决方式,基本避免了数据冲突会带来的性能损失。在这个过程中,我们的思维和硬件编程的思想都得到了极大的提高。

此外,在本次实验中我们还实现了一些简单的扩展功能。比如,我们利用 FLASH 实现了 Kernel 的自动引导;实现了 INT 软件中断指令,对软件中断的机理和实现方式有了更加深刻的认识;使用了 VGA 进行调试,对 VGA 的显像原理有了更深的理解。

当然,我们在实验过程中也遇到过不少困难。在尚未实现 VGA 时,我们只能通过 LED 灯列进行调试,调试很困难;实验初期,由于对 VHDL 的理解不够深入,导致逻辑上没有问题的代码在实现中出现时序问题,花费了很长的调试时间,最后借鉴学长的经验,通过状态机来同步时钟,终于解决了问题;我们在很早就实现了 12.5MHZ 的 CPU,但为了提高性能,我们通过减半各个段的状态数以提高主频,在实现过程中,由于 12.5MHZ 的各个流水段的各个状态与 25MHZ 有略微但是影响重大的不同,导致升级工作花费了远超预算的时间(一个通宵);在实现软件中断时,我们先用提供的 kernel.bin 进行测试,但是无论如何也通过不了,最后我们反汇编了 kernel.bin,发现其反汇编代码中含一条不要求的扩展指令 SLT,导致软件中断测试异常,最后我们重写并编译了 kernel 原码,立即解决了问题。

我们认为本次实验中的收获远远大于造计算机本身。本次实验让我们真正体验到合作带来的效率增益;学会了如何设计硬件、如何进行硬件编程、如何调试硬件、如何测试硬件等一整套较为方法论;我们提升了在学习、实验和研究上的能力和思维,甚至对我们的个人品质(耐心等)有一定陶冶作用。最后,我们得感谢刘卫东、李山山老师的指导以及助教在小班课上对 RAM 访问和数据冲突处理方面的建议,同时我们也要感谢实验过程给我们提供经验的 3 字班、2 字班学长和互相传授经验的同年级同学!