

清华大学计算机系《计算机组成原理》课程

MIPS 32 流水线 CPU 设计

实验报告

计 14 赵一开 2011011262

高俊杨 2011011277

一、设计目标

- (1) 实现支持 MIPS32 指令集的、采用五段流水线结构的 CPU
- (2) CPU 支持中断/异常处理，包括 TLB 缺失及系统调用
- (3) 编写并整理 32 位 THINPAD 教学计算机相关的测试工具，包括串口测试、SRAM 测试等。
- (4) 针对 32 位 CPU 的新特性对于原有的监控程序进行了小部分的修改，并用 Python 对原有的终端程序进行了重写，使其可以在多平台上使用。

二、运行效果

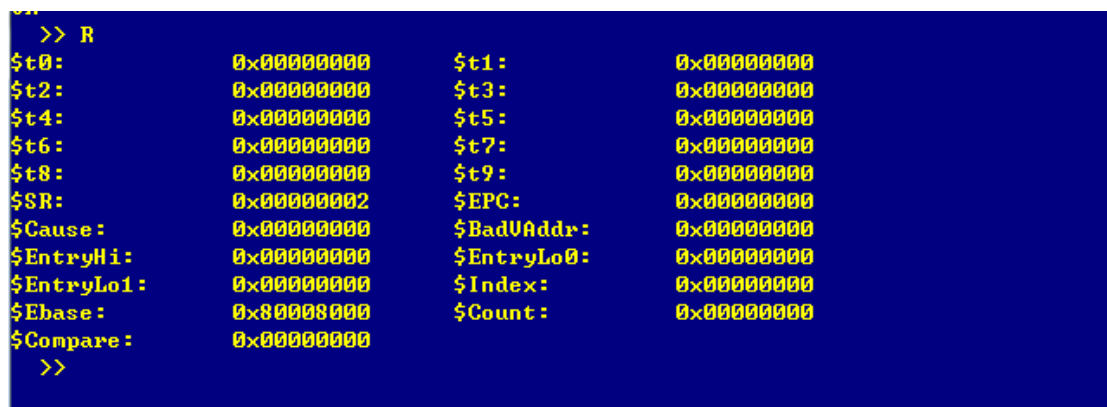
完成 CPU 的启动之后，运行 Term.exe，输入对应的端口号，点击实验平台的复位键。若终端显示 OK，则说明终端已经与 CPU 完成通讯。



```
D:\Term115200.exe
>>COM 3

    Ok.. Connected with com...
OK
>>
```

输入 R 指令，查看当前的通用寄存器的值



```
>> R
$t0:      0x00000000    $t1:      0x00000000
$t2:      0x00000000    $t3:      0x00000000
$t4:      0x00000000    $t5:      0x00000000
$t6:      0x00000000    $t7:      0x00000000
$t8:      0x00000000    $t9:      0x00000000
$SR:      0x00000002    $EPC:     0x00000000
$Cause:    0x00000000    $BadVAddr: 0x00000000
$EntryHi:  0x00000000    $EntryLo0: 0x00000000
$EntryLo1: 0x00000000    $Index:    0x00000000
$Ebase:    0x80008000    $Count:    0x00000000
$Compare:  0x00000000
>>
```

输入 D 指令，分别查看地址 0x80000000 和 0x80008000 处的值（此处为事先烧入实验平台的监控程序代码和中断处理代码）

```
>> D 0x80000000
[0x80000000]: 0x10000002
[0x80000004]: 0x00000000
[0x80000008]: 0x00000000
[0x8000000c]: 0x3c1d807f
[0x80000010]: 0x3411ff00
[0x80000014]: 0x03b1e821
[0x80000018]: 0x3c118000
[0x8000001c]: 0x36318000
[0x80000020]: 0x40917800
[0x80000024]: 0x24110002
```

```
>> D 0x80008000
[0x80008000]: 0x27bdfdc
[0x80008004]: 0xafbf0000
[0x80008008]: 0xafb00004
[0x8000800c]: 0xafb10008
[0x80008010]: 0xafb2000c
[0x80008014]: 0xafb30010
[0x80008018]: 0xafb40014
[0x8000801c]: 0xafb50018
[0x80008020]: 0xafb6001c
[0x80008024]: 0xafb70020
```

使用 LOAD 指令，将用户编辑好的二进制文件写入内存，并使用 G 指令运行这段写入的程序

```
>> LOAD ULTIMATE.BIN 0x80400000
complete!

>> G 0x80400000
5
12345
```

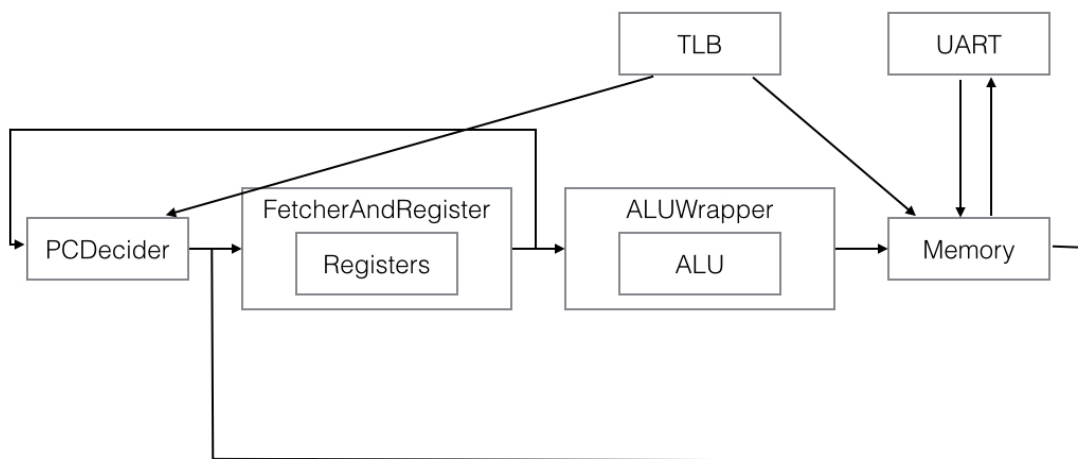
使用 D 指令可查看写入的二进制文件内容

```
>> D 0x80400000 20
[0x80400000]: 0x24020003
[0x80400004]: 0x0000000c
[0x80400008]: 0x00022820
[0x8040000c]: 0x00002025
[0x80400010]: 0x3c07001f
[0x80400014]: 0x00003025
[0x80400018]: 0x24c60001
[0x8040001c]: 0x14c7fffe
[0x80400020]: 0x00000000
[0x80400024]: 0x00000000
[0x80400028]: 0x24840001
[0x8040002c]: 0x24020005
[0x80400030]: 0x0000000c
[0x80400034]: 0x3c02bfd0
[0x80400038]: 0x1485fff6
[0x8040003c]: 0xac440000
[0x80400040]: 0x03e00008
[0x80400044]: 0x00000000
[0x80400048]: 0x00000000
[0x8040004c]: 0x09e47101
```

使用 U 指令可对这段二进制码进行反汇编

```
>> U 0x80400000 20
00100100000000100000000000000011 [0x80400000]: addiu    $v0, $zero, 3
0000000000000000000000000000001100 [0x80400004]: syscall  0
0000000000000000100010100000100000 [0x80400008]:
0000000000000000000010000000100101 [0x8040000c]: or      $a0, $zero,$zero
00111100000000111000000000001111 [0x80400010]: lui     $a3, 31
0000000000000000000011000000100101 [0x80400014]: or      $a2, $zero,$zero
00100100110001100000000000000001 [0x80400018]: addiu   $a2, $a2, 1
00010100110001111111111111111110 [0x8040001c]: bne     $a2, $a3, -2
00000000000000000000000000000000 [0x80400020]: nop
00000000000000000000000000000000 [0x80400024]: nop
00100100100001000000000000000001 [0x80400028]: addiu   $a0, $a0, 1
00100100000000100000000000000101 [0x8040002c]: addiu   $v0, $zero, 5
000000000000000000000000000001100 [0x80400030]: syscall  0
0011110000000010101111111010000 [0x80400034]: lui     $v0, 49104
0001010010000101111111111110110 [0x80400038]: bne     $a0, $a1, -10
10101100010001000000000000000000 [0x8040003c]: sw      $a0, 0($v0)
00000011111000000000000000001000 [0x80400040]: jr      $ra
00000000000000000000000000000000 [0x80400044]: nop
00000000000000000000000000000000 [0x80400048]: nop
00001001111001000111000100000001 [0x8040004c]: j       31748353
```

三、 总体设计



CPU 设计包含以下部分：

1、ALU (ALUWrapper)

ALUWrapper 中封装了 ALU 运算单元，该模块与前一级 FetcherAndRegister 以及后一级 Memory 相连。支持算术、逻辑运算但不支持乘除法运算。完成的工作包括：

- 根据前一级的信号完成算术逻辑运算
- 将其他信号继续传递给下一级
- 将为内存地址的计算结果通过 TLB 转换后得到物理内存地址，若 TLB 未找到则传递信号给 FetcherAndRegister 以产生异常

关于该模块更为详细的说明会在下一章节“模块分析”中进行。

2、Register (FetcherAndRegister)

该模块与 PCDecider、ALUWrapper 相连，包含一个 Register 子模块，并与 SRAM 数据总线相连。该模块是 CPU 工作的核心部分，CPU 大部分的工作都在其中完成，包括：

- 从 RAM 的数据线得到指令并译码
- 读取指令需要的寄存器的值
- 从 Memory 模块得到写寄存器信号并执行
- 判断跳转、等待、PC 异常、TLB 异常等并给出信号
- 将信号传给下一级

其中，Register 子模块中包含了通用寄存器组和 CP0 寄存器组，寄存器均使用 signal 实现。这样的设计可以使对寄存器的存取与译码阶段在同一个 CPU 周期内完成。实验简单并且节省时间。

关于该模块更为详细的说明会在下一章节“模块分析”中进行。

3、Memory

该模块与前一级 **ALUWrapper** 以及 **FetcherAndRegister** 相连,与 **SRAM** 的地址、内存总线、**UART** 模块相连,是 **CPU** 的存储控制器,封装了 **SRAM**、**ROM** 和串口的读写。完成的功能包括:

- 根据前一级的信号完成内存读写
- 读写特定地址时改为从 **UART** 模块读写
- 将写寄存器信号传回至 **FetcherAndRegister**

关于该模块更为详细的说明会在下一章节“模块分析”中进行。

4、PCDecider

该模块 **FecherAndRegister** 相连,并与 **SRAM** 的地址总线相连。
该模块的功能是根据输入的信号判断跳转、等待(气泡)、顺序执行等,并给 **RAM** 以地址信号,输出现在的 **PC** 值。

关于该模块更为详细的说明会在下一章节“模块分析”中进行。

5、TLB

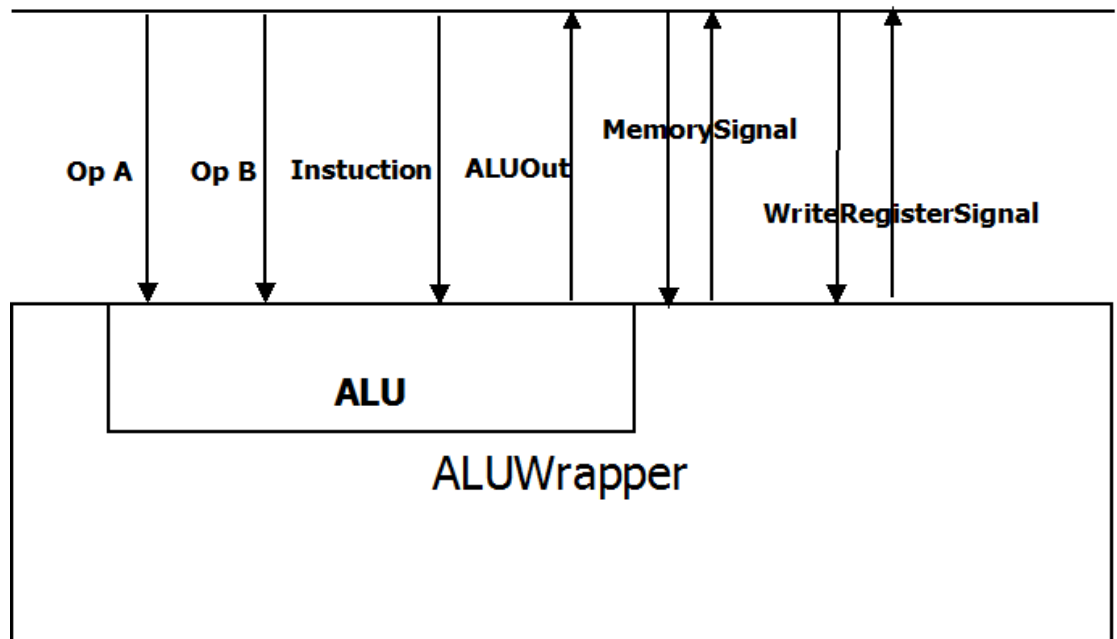
该模块主要完成从虚拟地址到实际物理地址的转化,提供 **TLB** 表,映射给 **Memory** 模块。

当取值、访存的地址发生 **TLB** 未找到异常时,这个异常信号会传递给译码模块 (**FetcherAndRegister**),其根据不同情况暂停流水并跳刀中断处理地址。

关于该模块更为详细的说明会在下一章节“模块分析”中进行。

四、 模块分析

1. ALU



instruction(hex)	result
0	C=A+B, signed
1	C=A+B, unsigned
2	C=A-B, signed
3	C=A-B, unsigned
4	C=A and B
5	C=A or B
6	C=A xor B
7	C=A nor B
8	C=A<B?1:0
9	C=A << B
A	C=A >> B (unsigned)
B	C=A >> B (signed)

结构说明：

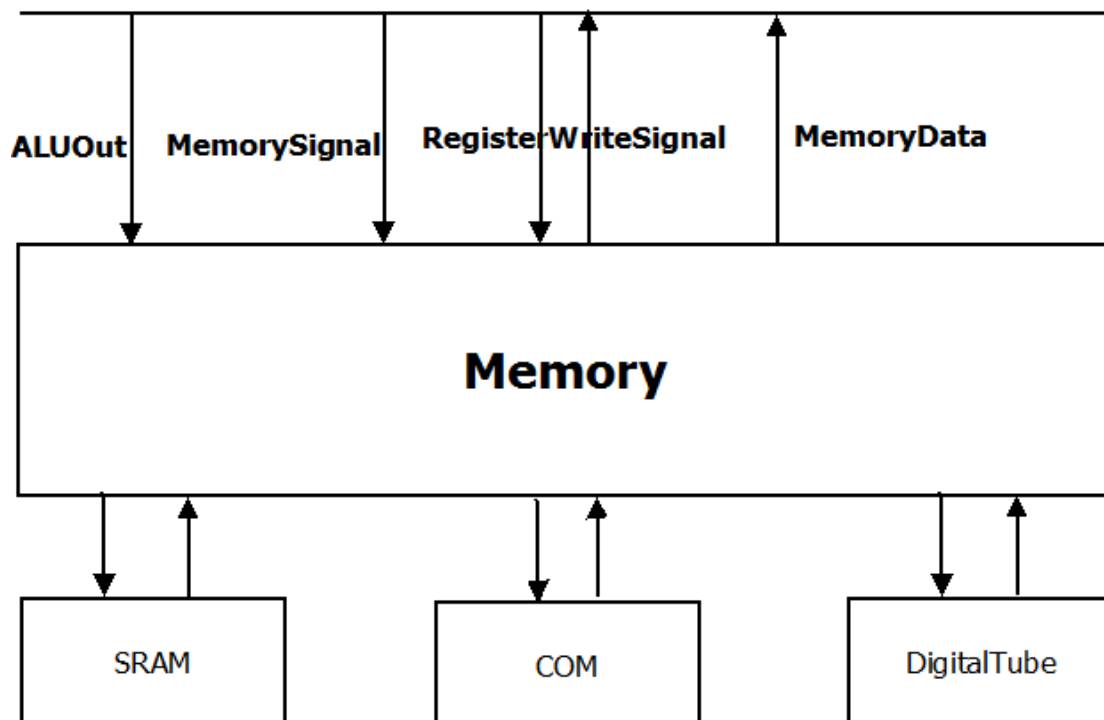
- 主要输入信号：ALU 信号，内存信号，寄存器写信号
- 主要输出信号：ALU 输出，内存信号，寄存器写信号
- 相连模块：TLB

ALU 模块主要完成算术运算和逻辑运算，具体的算术逻辑运算及其指令参见上表。我们将 ALU 模块封装至 ALUWrapper 中，在接收 ALU 模块所需要的 ALU 输入信号之外，同时接收内存信号和寄存器写信号。

ALUWrapper 对于内存信号的处理如下：将内存地址的计算结果通过 TLB 转化后得到物理内存地址。若地址合法，则将转化后的地址传送给下一级。若发生了 TLB 缺失异常，则传递信号给 FetcherAndRegister 模块由它来抛出异常。

ALUWrapper 对于寄存器写信号的处理只是做一个单纯的传递，此处并不对其进行修改。

2. Memory



结构说明：

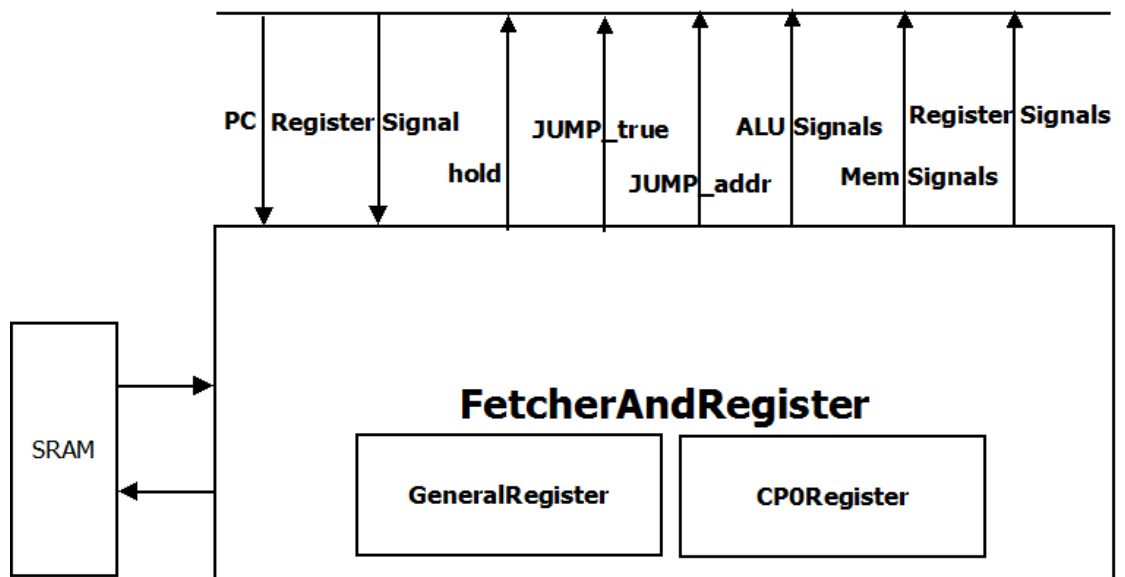
- 主要输入信号：ALU 输出，内存信号，寄存器写信号
- 主要输出信号：内存读取输出，寄存器写信号
- 主要相连模块：UART，数码管
- 相连外部接口：SRAM 的地址、数据总线

Memory 模块位于五级流水线结构的第四个阶段—访存阶段，在该模块中完成对于内存的读写。该模块与前一级 ALUWrapper 以及 FetcherAndRegister 相连，根据前一级的信号完成内存读写。在对输入的地址进行访问时，读写特定地址时改为从 UART 模块或数码管模块读写。在该模块完成了对于内存的之后会将写寄存器信号传回至 FetcherAndRegister。

其中，特殊的内存地址如下：

- UART 数据地址：0xBFDD003F8 (TLB 转化后为 0x1FD003F8)
- UART 控制地址：0xBFDD003FC (TLB 转化后为 0x1FD003FC)，可写时最低位置 1，可读时第二低位置 1
- 数码管 0、数码管 1 分别为：0xBFDD00000, 0xBFDD00004

3. Register



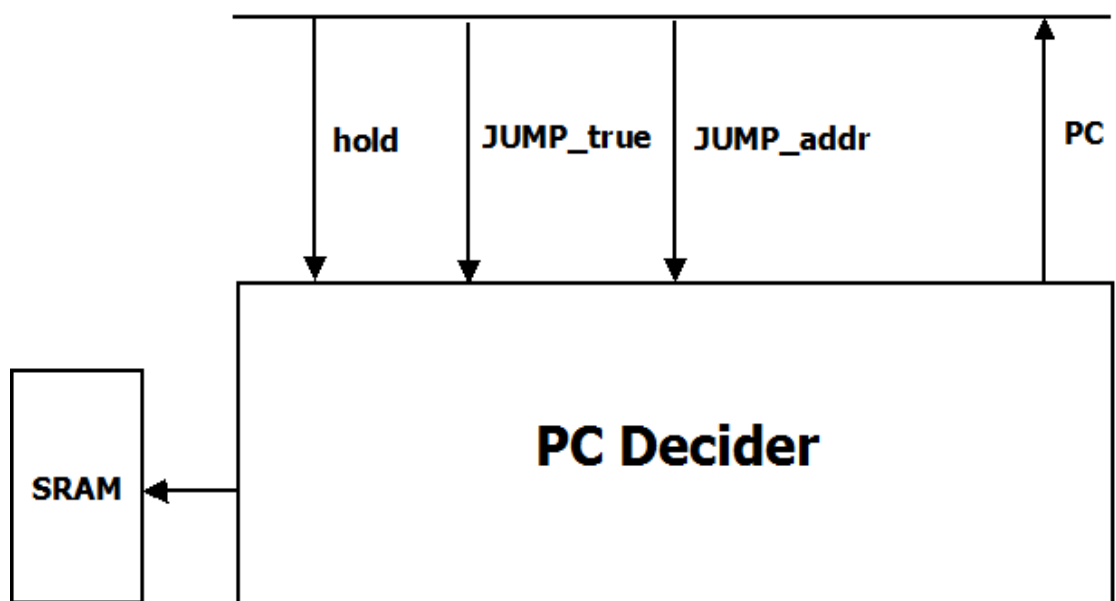
结构说明:

- 主要输入信号: PC, 寄存器写回相关
- 主要输出信号: hold, JUMP_true, JUMP_addr, ALU 信号, 内存信号, 寄存器写信号
- 相连模块: Registers
- 相连外部接口: SRAM 数据总线

该模块完成五级流水线的取指和译码阶段。从 SRAM 的数据总线获得指令并译码, 相应的给出指令需要的 ALU、内存、寄存器读写信号。对于读取寄存器的操作, 将直接通过 Register 子模块读取数据并将数据输出。

在译码的过程中, 该模块会判断跳转、hold (nop 气泡)、PC 异常、TLB 异常等情况并给出 JUMP_addr、JUMP_true 和 hold 等控制信号传递给下一级。

4. PCDecider



结构说明:

- 主要输入信号: hold, JUMP_true, JUMP_addr
- 主要输出信号: PC
- 相连模块: TLB
- 相连外部接口: 两片 SRAM 的地址总线

该模块的主要工作即产生新的 PC 值。若信号 hold 为真 (nop 指令) 则不改变当前 PC 值, 否则根据 JUMP_true 是否为真判断跳转从而产生新的 PC 值。该模块与 TLB 模块进行通讯获得该指令地址的实际物理地址, 并给 SRAM 赋以地址, 输出新的 PC 值。

5. TLB

当虚拟地址没有在 TLB 表项中查找到相应表项时, 会触发 TLB 读/写异常。当取指、访存的地址发生 TLB 未找到异常时, 信号会传递给译码模块, 其根据不同情况暂停流水并跳到中断处理地址:

- 若取指时发生异常: 产生一个周期的气泡后跳转至异常处理地址, EPC 值为当前 PC 值
- 若数据访存时发生异常: 产生一个周期的气泡后跳转至异常处理地址, EPC 值为两个周期前的 PC 值

当 CPU 执行以上工作后, 状态重新转入取指状态。

6. UART

考虑到 CPLD 和 FPGA 共享了 Base SRAM 的数据地址总线, 并且串口连接至 CPLD, 如果在 CPLD 上实现串口的功能将比较复杂。因此在我们的实现中, 将串口模块的功能完全搬移到了 FPGA 中, CPLD 仅仅将串口与 FPGA 相连当做导线使用, 经实践没有发现任何问题。另外, 我们使用的是波特率为 115200 的串口通讯, 大大提高了传输效率。

五、 流水线时序分析

为了保证运行 CPU 流水线的性能以及解决其中的数据、结构和控制冲突, 每一个 CPU 周期内部划分为四个状态 (时钟周期), 各个模块在流水线的时序工作流程如下。

1、PCDecider

状态 S0 => 判断此时是否需要跳转, 若需要, 则将 PC 值赋值为跳转到的地址, 否则 PC 值加 4。

状态 S1 => SRAM 数据总线置高阻态, 将当前 PC 位置传送给 TLB 进行虚地址转化。

状态 S2 => 从 TLB 中获得虚拟地址所对应的真实物理地址

状态 S3 => 根据真实物理地址进行 SRAM 的片选, 对 SRAM 数据总线赋值为由上一状态获得的地址, 完成 PC 值的更新。

2、FetcherAndRegister

状态 S0 => 译码阶段。在此处完成如特权指令、系统调用、分支

跳转指令、J 型、I 型和 R 型指令的分析译码，根据每个指令的特点为以后的流水线阶段准备信号。同时需要记录各条指令对于寄存器的读写访问情况。

状态 S1 => 在此阶段，CPU 根据上一阶段记录的各指令对于寄存器的读写访问情况来判断是否发生了数据冲突，如果发生了冲突则暂停流水。否则，进行正常的寄存器读写。

状态 S2 => 在这一阶段主要完成对异常和中断的处理。若发生了异常或中断，则对 CPO 协处理器的相关寄存器进行赋值并暂停流水线。对于异常和中断的处理会在第七章进行详细阐述。

状态 S3 => 将前几个阶段准备好的信号传递给下一个阶段。

3、ALUWrapper

状态 S0 => 对 ALU 子模块所需的操作数、运算指令进行赋值。对寄存器读写、内存读写的相关信号赋值。

状态 S1 => 将通过 ALU 运算后的地址进行虚拟内存地址的转化。

状态 S2 => 空闲，等待一个时钟周期。

状态 S3 => 判断是否发生了 TLB 缺失异常，如果发生异常，则发出 TLB 异常信号，并将之前的所有运算结果置零（包括 ALU 运算结果和内存读写信号），暂停工作。否则，则输出 ALU 运算结果以及转换后的真实物理地址，并传递上一阶段的内存读写信号。

4、Memory

状态 S0 => 将 SRAM 数据总线置高阻态。

状态 S1 => 为 SRAM 数据总线赋值地址，根据地址的不同，来选择是从 SRAM 还是串口中读取数据。

状态 S2 => 将数据从存储器指定的位置读出。

状态 S3 => 将 SRAM 数据总线置高阻态。

四个主要模块的四个状态都是同步进行的，为了向读者直观的显示出流水线工作时各个阶段各个状态之间的关系，特将上述内容总结为下表。

状态 模块	S0	S1	S2	S3
PCDecider	判断此时是否需要跳转，若需要，则将 PC 值赋值为跳转到的地址，否则 PC 值加 4	SRAM 数据总线置高阻态，将当前 PC 位置传送给 TLB 进行虚地址转化	从 TLB 中获得虚拟地址所对应的真实物理地址	根据真实物理地址进行 SRAM 的片选，对 SRAM 数据总线赋值为由上一状态获得的地址，完成 PC 值的更新

FetcherAndRegister	译码阶段。在此处完成如特权指令、系统调用、分支跳转指令、J 型、I 型和 R 型指令的分析译码，根据每个指令的特点为以后的流水线阶段准备信号。同时需要记录各条指令对于寄存器的读写访问情况	在此阶段，CPU 根据上一阶段记录的各指令对于寄存器的读写访问情况来判断是否发生了数据冲突，如果发生了冲突则暂停流水。否则，进行正常的寄存器读写	在这一阶段主要完成对异常和中断的处理。若发生了异常或中断，则对 CPO 协处理器的相关寄存器进行赋值并暂停流水线。	将前几个阶段准备好的信号传递给下一个阶段
ALUWrapper	对 ALU 子模块所需的操作数、运算指令进行赋值。对寄存器读写、内存读写的相关信号赋值	将通过 ALU 运算后的地址进行虚拟内存地址的转化	空闲，等待一个时钟周期	判断是否发生了 TLB 缺失异常，如果发生异常，则发出 TLB 异常信号，并将之前的所有运算结果置零（包括 ALU 运算结果和内存读写信号），暂停工作。否则，则输出 ALU 运算结果以及转换后的真实物理地址，并传递上一阶段的内存读写信号
Memory	将 SRAM 数据总线置高阻态	为 SRAM 数据总线赋值地址，根据地址的不同，来选择是从 SRAM 还是串口中读取数据	将数据从存储器指定的位置读出	将 SRAM 数据总线置高阻态

通过上表可以清晰的看出，流水线使得指令可以在 CPU 工作流程中折叠进行，相比于多周期 CPU，大大的提升 CPU 处理性能。虽然流水线的工作形式极大程度上改善了 CPU 的工作效率，但是由于指令的重叠运行也不可避免的带来了各种冲突。下面，我们将就解决三种典型冲突的方法进行详细说明。

六、冲突处理

1、数据冲突的处理

当写寄存器指令执行后的三个时钟周期内，若遇到对同一个寄存器的读操作，由于流水线结构的特性会产生数据冲突，我们的解决方法如下：

- 使寄存器模块正确处理同时读写同一个寄存器的情况，保证发生同时读写寄存器时读出的值等于写入的值，从而解决相距三个周期的冲突
- 在译码阶段记录前两条指令是否写寄存器以及写寄存器的值，当发生数据冲突时暂停流水

2、控制冲突的处理

流水线的控制冲突是因为程序执行转移类指令而引起的冲突。转移类指令如无条件转移、条件转移、子程序调用、中断等，它们属于分支指令，执行中可能改变程序执行的方向，从而造成流水线断流。对于控制冲突，我们的解决方法如下：

- 对于 **jump**、**branch** 指令，由于 MIPS 存在延迟槽机制，故不存在冲突
- 对于 **syscall**、中断等，由译码阶段判断后暂停流水一个周期

3、结构冲突的处理

指令在流水线中重叠执行时，当硬件资源满足不了指令重叠执行的要求时，就会发生因为硬件冲突而产生的结构冲突。而解决结构冲突的基本方法也就是考虑采用资源充分重复设置的方法。教学实验平台上包含两片以上的 **SRAM**，若是可以实现指令和数据的分别存放就不会发生存储器访存的结构冲突。但是由于实验环境配置的关系，我们必须实现对同一个地址空间的写指令和运行指令，因此无法采用该方法。

为了达到同样的处理效果，我们将每个 CPU 周期分为了 4 个时钟周期，对于存储器在第一和第四两个周期进行取指令，在中间两个周期进行数据读写。（可以在第五章节的图表中得以体现）

七、中断、异常处理

CPU 实现的异常处理包括 TLB 读/写异常，中断主要实现了系统调用 **SysCall**。当中断或异常发生时，如果条件满足，则需要立即跳到中断、异常处理代码阶段交由软件进行处理。（此处为事先写入内存 **0x80008000** 的异常处理代码）。当指令执行到 **ERET** 的时候，要返回到触发中断或异常的指令，对该条指令重新执行。由于处理中断和异常的过程十分相似，为节省篇幅，下面仅就由 **TLB Missing** 触发的异常为例，简述异常发生后的处理和返回的过程。

1. TLB Missing

当虚拟地址没有在 TLB 表项中查找到相应表项的时候，会触发 TLB 读/写异常，异常信号会传递给译码模块，其根据不同情况暂停流水并跳到中断异常处理地址。一旦发生了 TLB 异常，CPU 则执行以下工作：

- 记录当前指令地址，写入 **EPC** 寄存器。在此需要特别注意，若是取指时发生异常，需要产生一个周期的气泡后跳转至异常处理地址，**EPC** 值为当前 **PC** 值。若数据访存时发生异常，需要产生一个周期的气泡后跳转至异常处理地址，**EPC** 值为两个周期前的 **PC** 值。
- 记录访问内存的地址，写入 **BadAddr** 寄存器
- 记录是 TLB 读异常还是 TLB 写异常，写入 **Cause** 寄存器
- 将全局中断关闭，即 **SR** 寄存器的 **EXL** 置为 1
- 将访问的地址扩展到 32 位写入 **EntryHi** 寄存器
- 将 **EBase** 寄存器中的值，偏移 **0x180** 后写入 **PC** 寄存器

2. 异常、中断返回

当执行到 ERET 指令时，将全局中断 EXL 重新设置为 ‘0’，并且将 EPC 写入 PC，回归到原来被中断的程序继续执行。

八、 心得体会

1. 写一个 CPU 还是很容易的
2. 写一个高性能 CPU 还是很难的
3. 写一个符合行业标准的鲁棒的高性能 CPU 真心很难

参考文献

- [1] D.Sweetman. See Mips Run Linux [M]. 2th ed. San Francisco: Denise E.M. Penrose, 2010.
- [2] David A. Patterson. Computer Organization and Design, The Hardware/Software Interface, Third Edition.
- [3] 薛枫. THINPAD 教学计算机实验环境设计与实现. 2013

附录 1: MIPS 32 指令集

MIPS32 指令表

(1) addiu

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	9	s	d	signed const		
指令格式	ADDIU d, s, const					
指令功能	GPR[d] = GPR[s] + SEXT[const]					
功能说明	32 位立即数加法,无溢出检测。					

(2) addu

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	0	s	t	d	0	33
指令格式	ADDU d, s, t					
指令功能	$GPR[d] = GPR[s] + GPR[t]$					
功能说明	32 位寄存器加法，无溢出检测。					

(3)subu

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	0	s	t	d	0	35

指令格式	SUBU d, s, t
指令功能	$GPR[d] = GPR[s] - GPR[t]$
功能说明	32 位寄存器减法，无溢出检测。

(4)slt

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	0	s	t	d	0	42
指令格式	SLT d, s, t					
指令功能	$GPR[d] = ((signed)GPR[s] < (signed)GPR[t]) ? 1 : 0$					
功能说明	若寄存器 s 的值小于寄存器 t 的值，置 d 为 1，否则 d 的值置为 0。					

(5)slti

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	10	s	d	signed const		
指令格式	SLTI d, s, const					
指令功能	GPR[d] = ((signed)GPR[s]<signed const)?1:0					
功能说明	若寄存器 s 的值小于立即数的值，置 d 为 1，否则 d 的值置为 0。					

(6)sltiu

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	11	s	d	unsigned const		
指令格式	SLTI d, s, const					
指令功能	GPR[d] = ((unsigned)GPR[s]<signed const)?1:0					
功能说明	若寄存器 s 的值小于立即数的值，置 d 为 1，否则 d 的值置为 0。					

(7) sltu

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	0	s	t	d	0	43
指令格式	SLTU d, s, t					
指令功能	$GPR[d] = ((unsigned)GPR[s] < (unsigned)GPR[t]) ? 1 : 0$					
功能说明	若寄存器 s 的值小于寄存器 t 的值，置 d 为 1，否则 d 的值置为 0。					

(8)mult

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	0	s	t	0	0	24
指令格式	MULT s, t					

指令功能	$HILO = (\text{signed})GPR[s] * (\text{signed})GPR[t]$
功能说明	32 位有符号乘法，结果的高 32 位存在 HI 寄存器，低 32 位存在 LO 寄存器中。

(9)mflo

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	0	0	0	d	0	18
指令格式	MFLO d					
指令功能	$GPR[d] = LO$					
功能说明	将 LO 寄存器的值赋给通用寄存器。					

(10)mfhi

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	0	0	0	d	0	16
指令格式	MFHI d					
指令功能	$GPR[d] = HI$					
功能说明	将 HI 寄存器的值赋给通用寄存器。					

(11)mtlo

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	0	s	0	0	0	19
指令格式	MTLO s					
指令功能	$LO = GPR[s]$					
功能说明	将通用寄存器的值赋给 LO 寄存器。					

(12)mthi

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	0	s	0	0	0	17
指令格式	MTHI s					
指令功能	$HI = GPR[s]$					
功能说明	将通用寄存器的值赋给 HI 寄存器。					

(13)beq

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	4	s	t	broffset		
指令格式	BEQ s, t, label					

指令功能	if (GPR[s]==GPR[t]) then PC = PC + broffset
功能说明	如果寄存器 s、t 的值相等则跳转。

(14) bgez

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	1	s	1	broffset		
指令格式	BGEZ s, label					
指令功能	if (GPR[s]>=0) then PC = PC + broffset					
功能说明	如果寄存器 s 的值大于等于 0 则跳转。					

(15) bgtz

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	7	s	0	broffset		
指令格式	BGTZ s, label					
指令功能	if (GPR[s]>0) then PC = PC + broffset					
功能说明	如果寄存器 s 的值大于 0 则跳转。					

(16) blez

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	6	s	0	broffset		
指令格式	BLEZ s, label					
指令功能	if (GPR[s]<=0) then PC = PC + broffset					
功能说明	如果寄存器 s 的值小于等于 0 则跳转。					

(17) bltz

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	1	s	0	broffset		
指令格式	BLTZ s, label					
指令功能	if (GPR[s]<0) then PC = PC + broffset					
功能说明	如果寄存器 s 的值小于 0 则跳转。					

(18)bne

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	5	s	t	broffset		
指令格式	BNE s, t, label					
指令功能	if (GPR[s]!=GPR[t]) then PC = PC + broffset					

功能说明	如果寄存器 s 的值小于 0 则跳转。					
------	---------------------	--	--	--	--	--

(19) j

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	2	Target				
指令格式	J label					
指令功能	PC = PC[31-28] (target<<2)					
功能说明	跳转到标号。					

(20) jal

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	3	Target				
指令格式	JAL label					
指令功能	GPR[31] = PC PC = PC[31-28] (target<<2)					
功能说明	将当前 PC 存到 31 号通用寄存器中，跳转到标号。					

(21) jalr s

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	0	s	0	31	0	9
指令格式	JALR s					
指令功能	$GPR[31] = PC$ $PC = GPR[s]$					
功能说明	将当前 PC 存到 31 号通用寄存器中，跳转到寄存器 s 中的地址。					

(22) jr s

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	0	s	0	0	0	8
指令格式	JR s					
指令功能	$PC = GPR[s]$					
功能说明	跳转到寄存器 s 中的地址。					

(23) lw

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	35	b	t	offset		
指令格式	LW t, offset(b)					

指令功能	$GPR[t] = Mem[GPR[b]+offset]$					
功能说明	32 位加载。读入一个字，存到寄存器 t 中。					

(24)sw

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	43	b	t	offset		
指令格式	SW t, offset(b)					
指令功能	Mem[GPR[b]+offset] = GPR[t]					
功能说明	写一个字，将寄存器 t 的值写入内存。					

(25) lb

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	32	b	t	offset		
指令格式	LB t, offset(b)					
指令功能	GPR[t] = SEXT(Mem[GPR[b]+offset])					
功能说明	8 位加载。读入一个字节，符号扩展存到寄存器 t 中。					

(26) lbu

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	36	b	t	offset		
指令格式	LBU t, offset(b)					
指令功能	GPR[t] = ZEXT(Mem[GPR[b]+offset])					
功能说明	8 位加载。读入一个字节，零扩展存到寄存器 t 中。					

(27) sb

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	40	b	t	offset		
指令格式	SB t, offset(b)					
指令功能	Mem[GPR[b]+offset] = GPR[t]					
功能说明	写一个字节，将寄存器 t 的值写入到内存。					

(28)and

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	0	s	t	d	0	36
指令格式	AND d, s, t					
指令功能	$GPR[d] = GPR[s] \& GPR[t]$					

功能说明	32 位寄存器与运算。
------	-------------

(29) andi

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	12	s	d	unsigned const		
指令格式	ANDI d, s, const					
指令功能	GPR[d] = GPR[s] & unsigned const					
功能说明	寄存器与立即数与运算。					

(30) lui

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	15	0	d	unsigned const		
指令格式	LUI d, const					
指令功能	GPR[d] = unsigned const << 16					
功能说明	上位加载立即数。					

(31) nor

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	0	s	t	d	0	39
指令格式	NOR d, s, t					
指令功能	GPR[d] = ~(GPR[s] GPR[t])					
功能说明	32 位寄存器或非运算。					

(32) or

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	0	s	t	d	0	37
指令格式	OR d, s, t					
指令功能	GPR[d] = GPR[s] GPR[t]					
功能说明	32 位寄存器或运算。					

(33) ori

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	13	s	d	unsigned const		
指令格式	ORI d, s, const					
指令功能	GPR[d] = GPR[s] unsigned const					
功能说明	寄存器与立即数或运算。					

(34) xor

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	0	s	t	d	0	38
指令格式	XOR d, s, t					
指令功能	$GPR[d] = GPR[s] \wedge GPR[t]$					
功能说明	32 位寄存器异或运算。					

(35) xori

指令编码	14	25-21	20-16	15-11	10-6	5-0
	9	s	d	unsigned const		
指令格式	XOR d, s, const					
指令功能	GPR[d] = GPR[s] ^ unsigned const					
功能说明	32 位寄存器异或运算。					

(36) sll

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	0	0	s	d	shf	0
指令格式	SLL d, s, shf					
指令功能	$GPR[d] = (\text{unsigned})GPR[t] \ll \text{shf}$					
功能说明	逻辑左移。					

(37) sllv

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	0	s	t	d	0	4
指令格式	SLLV d, t, s					
指令功能	$GPR[d] = (\text{unsigned})GPR[t] \ll (GPR[s] \% 32)$					
功能说明	逻辑左移。					

(38) sra

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	0	0	s	d	shf	3
指令格式	SRA d, s, shf					
指令功能	$GPR[d] = (\text{signed})GPR[t] \gg \text{shf}$					
功能说明	算术右移。					

(39) srav

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	0	s	t	d	0	7
指令格式	SRAV d, t, s					
指令功能	$GPR[d] = (\text{signed})GPR[t] \gg (GPR[s]\%32)$					
功能说明	算术右移。					

(40) srl

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	0	0	t	d	shf	2
指令格式	SRL d, t, shf					
指令功能	$GPR[d] = (\text{unsigned})GPR[t] \gg \text{shf}$					
功能说明	逻辑右移。					

(41) srlv

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	0	s	t	d	0	6
指令格式	SRLV d, t, s					
指令功能	$GPR[d] = (\text{unsigned})GPR[t] \gg (GPR[s]\%32)$					
功能说明	逻辑右移。					

(42) syscall

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	0	code				12
指令格式	SYSCALL					
指令功能	CPR[14] = PC CPR[13][6-2] = 01000 CPR[12][1] = 1 PC = CPR[15] + 0x180					
功能说明	触发系统调用中断。					

(43) eret

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	16	16	0	0	0	24
指令格式	ERET					
指令功能	$CPR[12][1] = 0$					

	PC = CPR[14]
功能说明	从中断返回。清除 SR(EXL)位并且跳转到 EPC 保存的位置去。

(44) mfc0

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	16	0	t	cs	0	0
指令格式	MFC0 t, cs					
指令功能	GPR[t] = CPR[cs]					
功能说明	将协处理器寄存器 cs 的值赋给通用寄存器 t。					

(45) mtc0

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	16	4	t	cd	0	0
指令格式	MTC0 t, cd					
指令功能	CPR[cd] = GPR[t]					
功能说明	将通用寄存器 t 的值赋给协处理器寄存器 cd。					

(46) tlbwi

指令编码	31-26	25-21	20-16	15-11	10-6	5-0
	16	16	0	0	0	2
指令格式	TLBWI					
指令功能						
功能说明	写 TLB 表项，详见 2.1.4 节。					

附录 2：MIPS 32 体系结构寄存器介绍

MIPS 体系中寄存器分为两种，通用寄存器和协处理器寄存器。通用寄存器有 32 个，\$0-\$31，具体介绍参加下表

MIPS32 通用寄存器介绍

寄存器编号	习惯命名	用法
0	zero	永远返回 0
1	at	保留给汇编器使用
2-3	v0,v1	子程序返回值
4-7	a0-a3	子程序调用的前几个参数
8-15	t0-t7	临时变量，子程序使用时无需保存
24-25	t8-t9	

16-23	s0-s7	子程序寄存器变量。子程序写入时必须保存其值并在返回前恢复其值。
26,27	k0,k1	供中断或自陷处理程序使用
28	gp	全局指针
29	sp	堆栈指针
30	s8/fp	第九个寄存器变量，可用于帧指针
31	ra	子程序的返回地址

协处理器寄存器也有 32 个，\$0-31，用于 CPU 的控制，教学实验中用到的部分寄存器参见下表

IPS32 协处理器寄存器介绍

寄存器编号	助记符	用法
12	SR	状态寄存器。确定 CPU 特权级，中断使能等。
13	Cause	记录导致中断、异常的原因。
14	EPC	异常/中断结束后的返回地址。
8	BadVAddr	触发最近的地址相关异常的程序地址。
10	EntryHi	TLB 相关。详见 2.1.4。
2	EntryLo0	
3	EntryLo1	
0	Index	
9	Count	计数器，定时触发时间中断。
11	Compare	
15	EBase	异常入口地址基址。

附录 3：开发过程中的测试工具

1. ghdl，一个开源工具，用于仿真 VHDL 代码
2. Scansion，一个开源工具，用于查看 ghdl 生成的.vcd 波形
3. Python 的 pyserial 库，用于调试串口通信
4. Mips-gcc/objcopy/ld 工具链，用于编译 C 程序、汇编程序

附录 4：监控程序及终端

我们沿用了薛枫学长编写的监控程序以及终端，但是由于他们是为多周期的 CPU 所设计，我们在使用之前做了一些必要的改动，包括：

1. 在监控程序中插入延时槽
2. 将中断处理程序放到了固定的内存地址（0x80008000）（因为存在延迟槽，并且是由编译器自动产生的，因此很难在编译时判断代码的入口点，所以我们将监控程序的中断处理代码单独出来）
3. 更改了终端的串口参数（波特率等）
4. 更改了终端对输入命令中地址的大小判断（原程序允许的最大内存地址为 1M）