

MIPS-16E 流水 CPU 实验报告

钱迪晨 计 35 2013011402

叶子鹏 计 35 2013011404

朱俸民 计 35 2012011894

2015 年 12 月 10 日

1 概述

我们实现了一个无延迟槽的带动态分支预测的流水线 CPU。我们实现了一个通用性极佳（支持软硬件中断，支持绘图）的计算机。

我们最初的目的是尽可能不插，少插气泡，由于某些冲突必须暂停流水线，我们最终仅在 3 种情况插 1 个气泡，而绝大多数时候我们都无需插气泡。我们的 CPU 的主频最高可以达到 21M（我们使用了 ISE 自带的模拟器件 IP 核 DCM 来分频）对于老师给的 5 个测例，我们花费的时间都非常少，因为我们几乎不用插气泡，所以花费的时间约等于指令数除以主频。

为了提高运行效率，我们增加了分支预测功能，branch 指令和 jump 指令均在 decode 阶段进行，从而使得因为跳转引入的气泡尽可能的少。分支预测采取的是一个大小为 3 的查询表，每次使用 pc 进行查询，如果出现一次错误则更新，即记录上一次的结果。

除了 CPU 的核心以外，我们做了硬件中断，软件中断，像素映射的 VGA 接口的显示器（由于片内的 RAM 容量不大，不足以存下 RGB，我们的显示是蓝白的）。

由于我们是像素映射，硬件的接口不仅单一而且不利于编程（非常繁琐，由于屏幕非常大，不足以用 16 位表示坐标，我们得传 2 次参数才能确定一个点），我们用软件实现了如下几个画图的接口：（详见第六节）

1. 画一个点
2. 画一条细线段
3. 画一条粗线段
4. 画一个等腰直角三角形
5. 从数据 RAM 中读取一个形状（用于实现字符集，比如 Unicode 或者 ASCII）

2 具体实现

2.1 CPU 模块

CPU 模块无可厚非是我们本次实现中最重要的一个模块，这个模块里面包含了非常多的原件，我们使用了如下组件来实现我们整个 CPU。下面会一一列举。

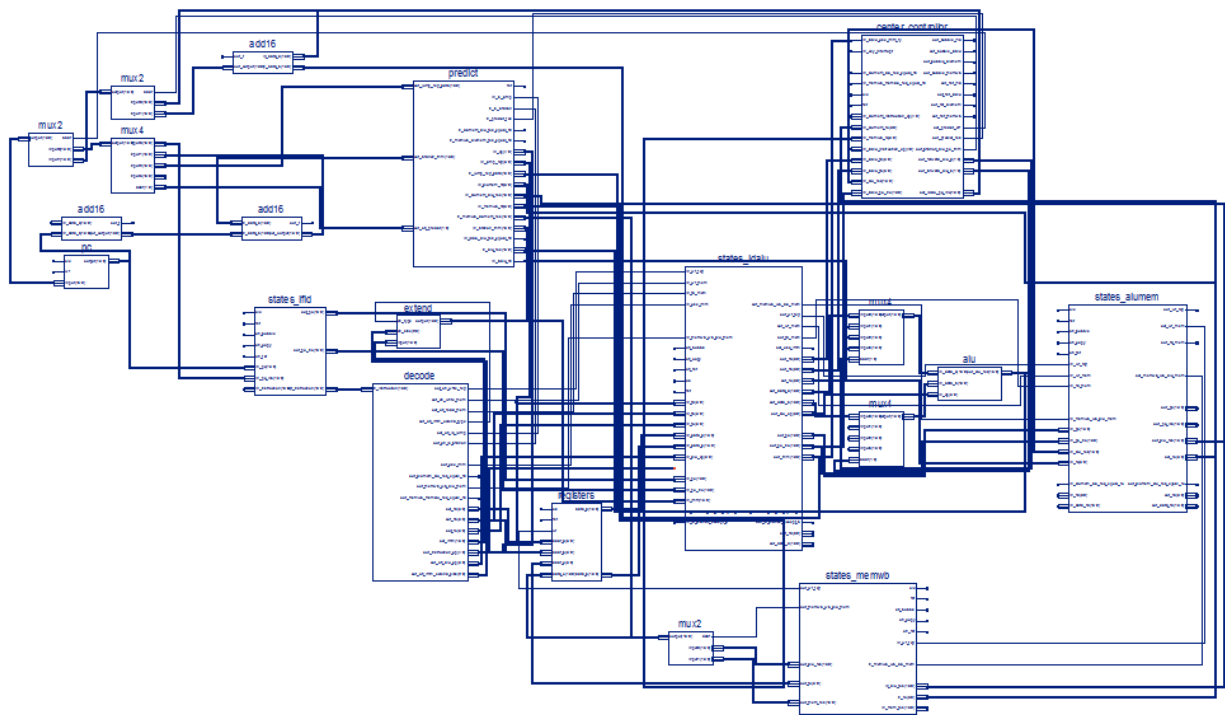


图 1: CPU 结构图

2.1.1 锁存单元

锁存单元包含 if/id 阶段，id/alu 阶段，alu/mem 阶段，mem/wb 阶段四个大的锁存器，在上升沿触发。这几个锁存器的行为都受到中央控制单元的控制，中央控制单元可以命令其进行气泡的插入，以及重置功能。

这四个部件的图如图 2-图 3 所示，具体信号如表 1-表 3 所示。

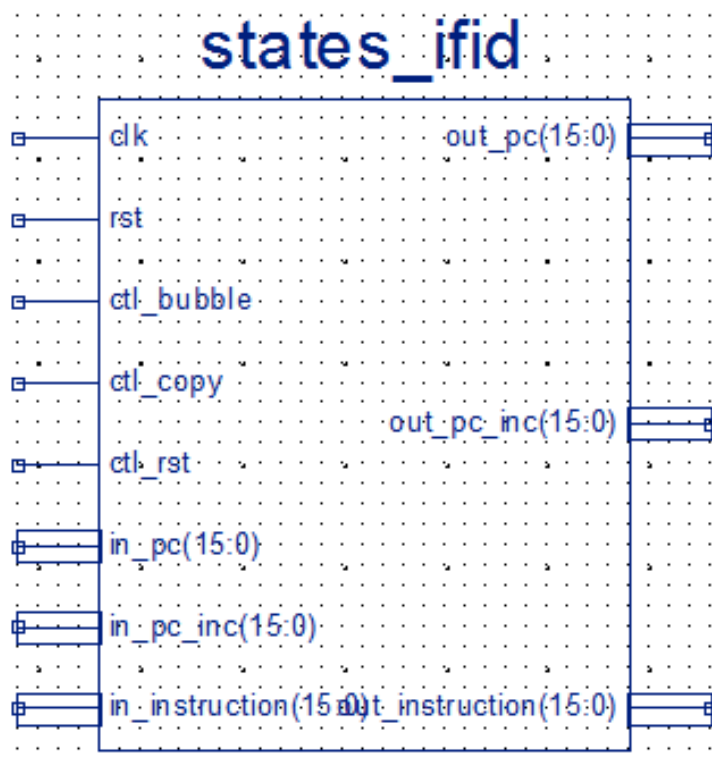


图 2: IF/ID 阶段锁存器设计图

表 1: IF/ID 阶段锁存器信号

信号	信号描述
clk	cpu 的时钟信号，上升沿的时候根据 ctl_bubble 和 ctl_rst 进行控制。如果 ctl_bubble 和 ctl_rst 均为低电平则进行锁存，将 in_pc, in_pc_inc, in_instruction 进行锁存并输出。
rst	异步清空信号，由外部控制开关接入。
ctl_bubble	气泡控制信号，由中央控制单元给出，如果该信号为高电平则表示下一个时钟上升沿，输出数据保持不变，低电平则该控制无效。
ctl_copy	由中央控制单元给出，用来进行数据拷贝。
ctl_rst	重置控制信号，由中央控制单元给出，如果如果该信号为高电平则表示下一个时钟输出清空即为一条 NOP 指令，低电平则该控制无效。
in_pc	表示下一条将要锁存的指令的 pc。
in_pc_inc	表示下一条将要锁存的指令的 pc+1。
in_instruction	表示下一条将要锁存的指令内容
out_pc	表示已经锁存的指令的 pc。
out_pc_inc	表示已经锁存的指令的 pc+1。
out_instruction	表示已经锁存的指令。

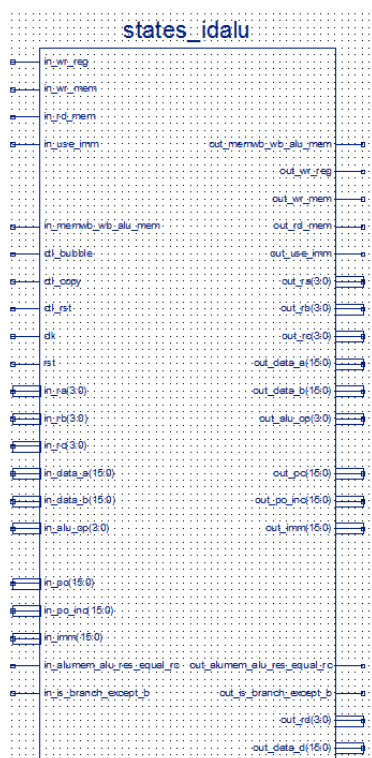


图 3: ID/ALU 阶段锁存器设计图

表 3: ID/ALU 阶段锁存器信号

信号	信号描述
in_ra	这是下一条指令 decode 出来的 alu 操作数 a 的寄存器值，注意不是数值，会传递给中央控制单元进行旁路选择。
in_rb	这是下一条指令 decode 出来的 alu 操作数 b 的寄存器值，注意不是数值，会传递给中央控制单元进行旁路选择。
in_rc	这是下一条指令 decode 出来的寄存器 c 的值，注意不是数值，会传递给中央控制单元进行旁路选择，也会传递给 alumem 锁存器。c 的寄存器表示的是写回的寄存器，非常重要，所以要一直往后传。
in_data_a	这是下一条指令 decode 出来的 alu 操作数 a 的值，用来传输给选择器，（选择器可能会选择旁路），最后送到 alu 进行计算。
in_data_b	这是下一条指令 decode 出来的 alu 操作数 b 的值，用来传输给选择器，（选择器可能会选择旁路），最后送到 alu 进行计算。
in_alu_op	这是下一条指令 alu 的操作码，会传输给三个 alu，具体内容请看 alu 部分。

in_pc	表示下一条将要锁存的指令的 pc。
in_pc_inc	表示下一条将要锁存的指令的 pc+1。
in_imm	表示下一条将要锁存的 decode 出来的立即数。
in_wr_reg	表示下一条指令是否需要在 writeback 阶段写回寄存器。
in_wr_mem	表示下一条指令是否需要在 memory 阶段写内存。
in_rd_mem	表示下一条指令是否需要在 memory 阶段读内存。
in_use_imm	表示下一条指令在 alu 阶段是否需要使用立即数，这个信号会帮助中央控制单元进行 alu_data_b 旁路的控制。
in_alumem_alu_ res_equal_rc	表示下一条指令到 memory 阶段的时候，alu 的出的结果是否会在 writeback 阶段写回寄存器。这个信号也是为了帮助中央控制单元进行旁路控制。
in_memwb_wb_ alu_mem	表示下一条指令在 writeback 阶段写回的数据是 memory 阶段读出的数据，还是在 alu 阶段算出的结果。
in_is_branch_ except_b	表示下一条指令是否是 branch 指令，除了 b 指令以外的 branch 指令都是高电平，这个信号是帮助中央控制单元进行分支预测的检验使用的信号。
ctl_bubble	气泡控制信号，由中央控制单元给出，如果该信号为高电平则表示下一个时钟上升沿，输出数据保持不变，低电平则该控制无效。
ctl_copy	由中央控制单元给出，用来进行数据拷贝。
ctl_rst	重置控制信号，由中央控制单元给出，如果如果该信号为高电平则表示下一个时钟输出清空即为一条 NOP 指令，低电平则该控制无效。
clk	cpu 的时钟信号，上升沿的时候根据 ctl_bubble 和 ctl_rst 进行控制。如果 ctl_bubble 和 ctl_rst 均为低电平则进行锁存，将所有带有 out 前轴的信号对应的 in 信号锁存然后输出。
rst	异步清空信号，由外部控制开关接入。
out_ra	表示 alu 阶段运算的寄存器 a 的值，这个值会送到中央控制单元进行旁路的控制。
out_rb	表示 alu 阶段运算的寄存器 b 的值，这个值会送到中央控制单元进行旁路的控制。
out_rc	表示 writeback 阶段写回的寄存器 c 的值。
out_rd	rd 是一个特殊的输出，只会在 sw 这条指令进行使用，rd 也需要送到中央控制单元进行旁路的控制，他的内容和 rb 完全一致。
out_data_a	表示 alu 阶段运算 a 的值，这个值是从一个四选一的选择器得到的，这个选择器的控制由中央控制单元给出。

out_data_b	表示 alu 阶段运算 b 的值，这个值是从一个四选一的选择器得到的，这个选择器的控制由中央控制单元给出。
out_data_d	表示 memory 阶段 sw 指令可能用到的值，这个值是从一个四选一的选择器得到的，这个选择器的控制由中央控制单元给出。
out_alu_op	表示当前 alu 进行的运算符号。
out_pc	表示已经锁存的指令的 pc。
out_pc_inc	表示已经锁存的指令的 pc+1。
out_imm	表示已经锁存的指令的立即数，这个数字会连接到 alu_data_b 的选择器。
out_alumem_alu_res_equal_rc	表示当前指令到 memory 阶段的时候，alu 的出的结果是否会在 writeback 阶段写回寄存器。这个信号也是为了帮助中央控制单元进行旁路控制。
out_memwb_wb_alu_mem	表示当前指令在 writeback 阶段写回的数据是 memory 阶段读出的数据，还是在 alu 阶段算出的结果。
out_is_branch_except_b	表示当前指令是否是 branch 指令，除了 b 指令以外的 branch 指令都是高电平，这个信号是帮助中央控制单元进行分支预测的检验使用的信号。
out_wr_reg	表示当前指令是否需要在 writeback 阶段写回寄存器。
out_wr_mem	表示当前指令是否需要在 memory 阶段写内存。
out_rd_mem	表示当前指令是否需要在 memory 阶段读内存。
out_use_imm	表示当前指令在 alu 阶段是否需要使用立即数，这个信号会帮助中央控制单元进行 alu_data_b 旁路的控制。

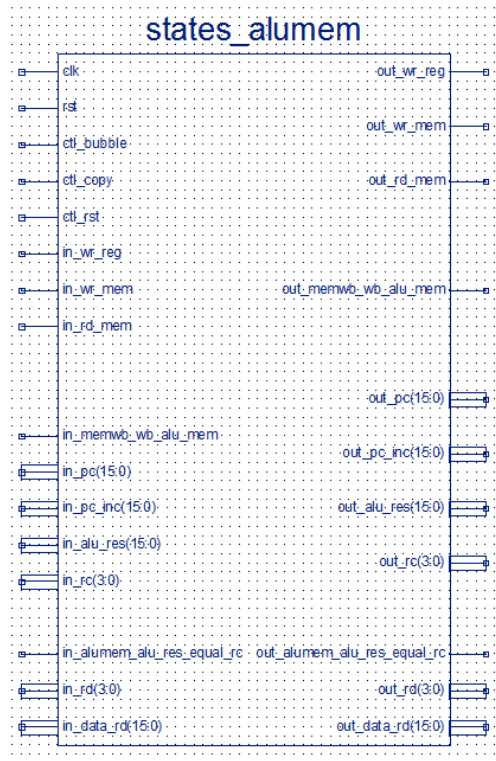


图 4: ALU/MEM 阶段锁存器设计图

表 5: ALU/MEM 阶段锁存器信号

信号	信号描述
in_rc	这是下一条指令 decode 出来的寄存器 c 的值，注意不是数值，会传递给中央控制单元进行旁路选择，也会传递给 memwb 锁存器。c 的寄存器表示的是写回的寄存器，非常重要，所以要一直往后传。
in_rd	这是专门为了 sw 指令设计的寄存器的值。
in_data_rd	这是专门为了 sw 指令设计的寄存器的值，通过一个 4 选 1 选择可以确保数据的正确性。由于我们的设计问题，alu 在进行 sw 指令的时候无法将 sw 的源寄存器值进行旁路计算解决数据冲突，所以使用了单独的一条旁路来解决这个问题。
in_pc	表示下一条将要锁存的指令的 pc。
in_pc_inc	表示下一条将要锁存的指令的 pc+1。
in_wr_reg	表示下一条指令是否需要在 writeback 阶段写回寄存器。
in_wr_mem	表示下一条指令是否需要在 memory 阶段写内存。
in_rd_mem	表示下一条指令是否需要在 memory 阶段读内存。
in_alu_res	表示下一条指令 alu 得出的结果。

<code>in_alumem_alu_</code>	表示下一条指令 alu 计算出的结果是否会在 writeback 阶段写回寄存器。这个信号也是为了帮助中央控制单元进行旁路控制。
<code>res_equal_rc</code>	
<code>in_memwb_wb_</code>	表示下一条指令在 writeback 阶段写回的数据是 memory 阶段读出的数据，还是在 alu 阶段算出的结果。
<code>alu_mem</code>	
<code>clk</code>	cpu 的时钟信号，上升沿的时候根据 <code>ctl_bubble</code> 和 <code>ctl_rst</code> 进行控制。如果 <code>ctl_bubble</code> 和 <code>ctl_rst</code> 均为低电平则进行锁存，将所有带有 out 前轴的信号对应的 in 信号锁存然后输出。
<code>rst</code>	异步清空信号，由外部控制开关接入。
<code>ctl_bubble</code>	气泡控制信号，由中央控制单元给出，如果该信号为高电平则表示下一个时钟上升沿，输出数据保持不变，低电平则该控制无效。
<code>ctl_copy</code>	由中央控制单元给出，用来进行数据拷贝。
<code>ctl_rst</code>	重置控制信号，由中央控制单元给出，如果该信号为高电平则表示下一个时钟输出清空即为一条 NOP 指令，低电平则该控制无效。
<code>out_pc</code>	表示已经锁存的指令的 pc。
<code>out_pc_inc</code>	表示已经锁存的指令的 pc+1。
<code>out_alu_res</code>	表示当前指令在 alu 阶段通过 alu 计算出来的值。
<code>out_rc</code>	表示当前指令 decode 出来的目的寄存器 c 的值。
<code>out_wr_reg</code>	表示当前指令是否需要在 writeback 阶段写回寄存器。
<code>out_wr_mem</code>	表示当前指令是否需要在 memory 阶段写内存。
<code>out_rd_mem</code>	表示当前指令是否需要在 memory 阶段读内存。
<code>out_memwb_wb_</code>	表示当前指令在 writeback 阶段写回的数据是 memory 阶段读出的数据，还是在 alu 阶段算出的结果。
<code>alu_mem</code>	

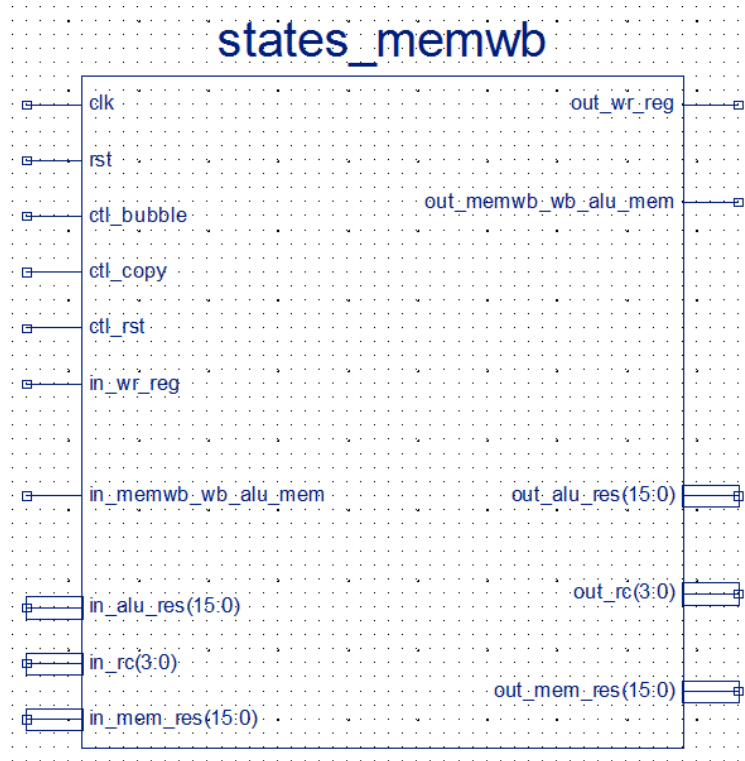


图 5: MEM/WB 阶段锁存器设计图

表 7: MEM/WB 阶段锁存器信号

信号	信号描述
clk	cpu 的时钟信号，上升沿的时候根据 ctl_bubble 和 ctl_rst 进行控制。如果 ctl_bubble 和 ctl_rst 均为低电平则进行锁存，将所有带有 out 前轴的信号对应的 in 信号锁存然后输出。
rst	异步清空信号，由外部控制开关接入。
ctl_bubble	气泡控制信号，由中央控制单元给出，如果该信号为高电平则表示下一个时钟上升沿，输出数据保持不变，低电平则该控制无效。
ctl_copy	由中央控制单元给出，用来进行数据拷贝。
ctl_rst	重置控制信号，由中央控制单元给出，如果如果该信号为高电平则表示下一个时钟输出清空即为一条 NOP 指令，低电平则该控制无效。
in_alu_res	表示下一条指令 alu 阶段计算出来的结果，可能用于写回。
in_rc	表示下一条指令写回的寄存器值。
in_wr_reg	表示下一条指令是否写回寄存器。
in_mem_res	表示下一条指令 memory 阶段得到的结果。
in_memwb_wb_alu_mem	表示下一条指令写回寄存器堆的是 alu 计算的结果还是 memory 访存的结果。

out_wr_reg	当前指令用来控制寄存器堆的写使能信号，使其可以在下降沿的时候更新。
out_memwb_wb_	当前指令写回内容 2 选 1 的控制信号，是选择 alu 计算的结果还是
alu_mem	memory 访存的结果。

2.1.2 PC 锁存单元

PC 锁存器，用来锁存 PC，保证 PC 的改写受到中央控制部分的控制。我们规定在下降沿写入，组合逻辑输出。具体信号请看下表。

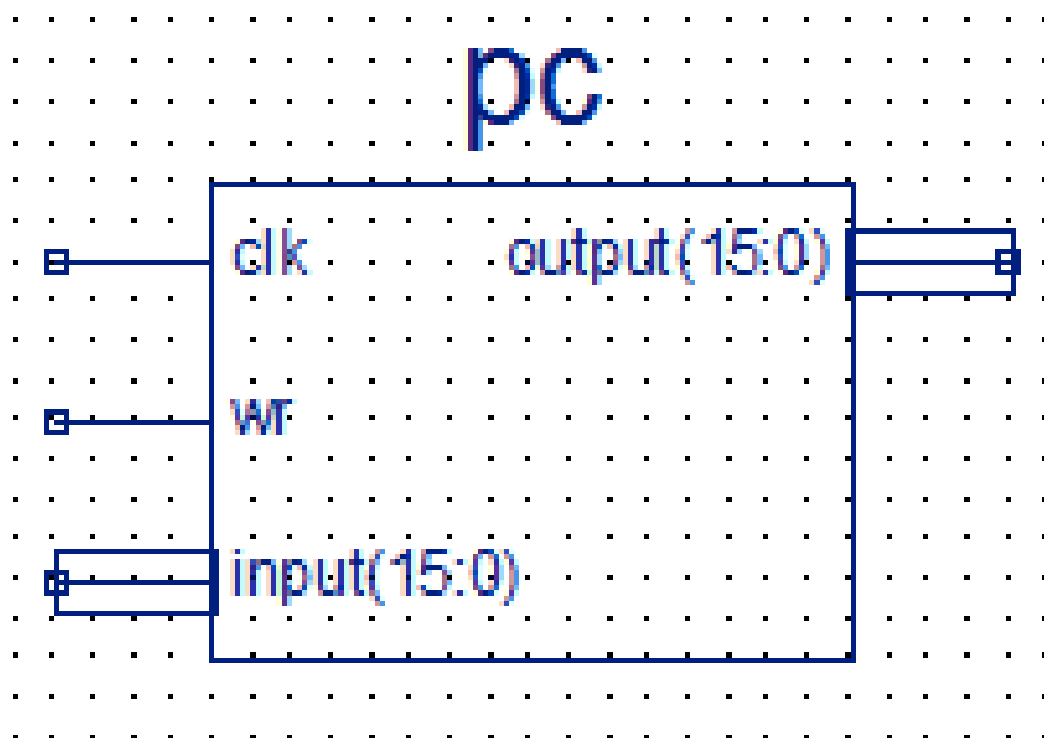


图 6: PC 锁存器

表 9: PC 锁存器

信号	信号描述
input	表示修改信号的输入，即修改的值。
clk	CPU 时钟。
wr	修改使能，如果为高电平则在下降沿修改 pc 的锁存器的值。
rst	异步清空信号，由外部控制开关接入。
output	当前 pc 锁存的值，组合逻辑。

下一条 PC 的选择是一个非常复杂的结构，具体的原理图如下。

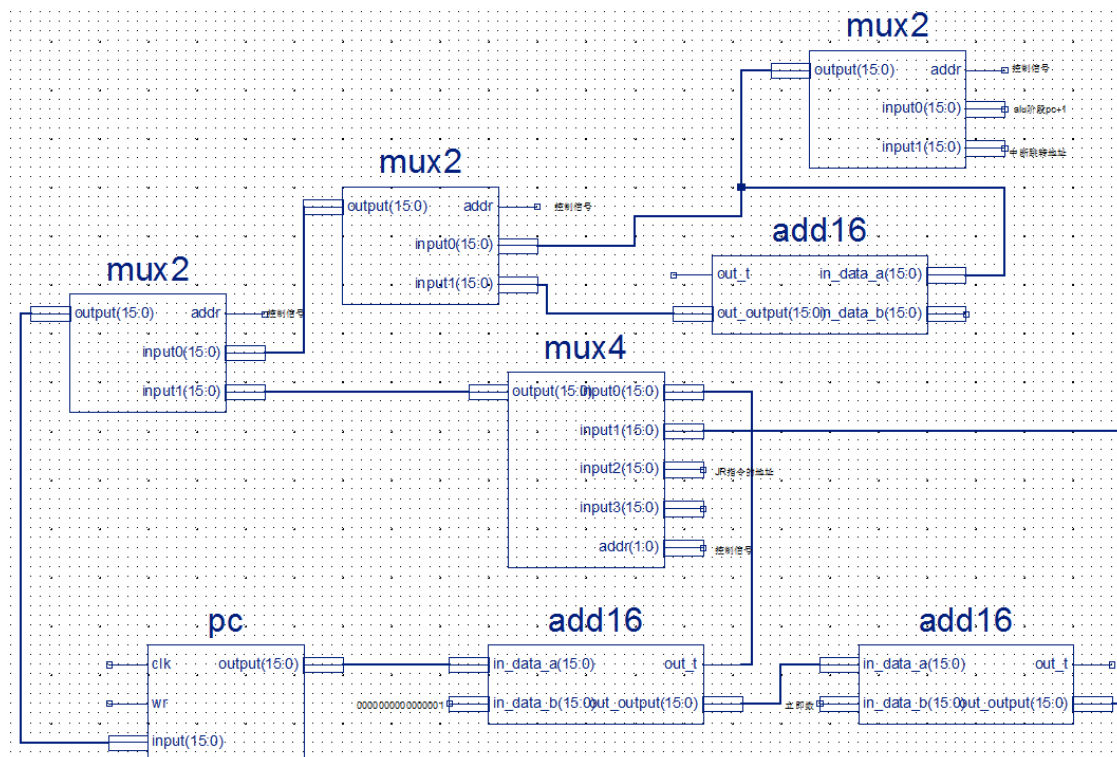


图 7: PC 结构图

具体而言，PC 的选择由两条路线决定。

1. 在 decode 阶段进行的跳转，包含 jr 指令以及分支预测。这里使用了图中下部分的部件，4 选 1 的选择器可以选择 pc+1, pc+1+imm, reg 的值进行跳转。控制信号都是中央控制单元给出的。
2. 在 alu 发现分支预测错误或者进行中断的跳转，则在上部分，最右边是一个 2 选 1，选择进行跳转的基础地址是 alu 阶段的记录下来的 pc 地址，或者是中断的指令的地址，左边的 2 选 1 则是是否加上立即数的选择器，最后连向最左边的 2 选 1 选择器。

综合上面的结构，构成了我们的 pc 整个模块，是的 cpu 的跳转可以正常的工作。由于大部分控制信号都是有中央控制单元给出，控制信号将统一在中央控制器里面描述。

2.1.3 跳转单元

跳转单元，用来在 decode 阶段进行计算和控制跳转的目的地址，这里面有旁路来处理 JR 指令的数据冲突。具体信号请看下表。

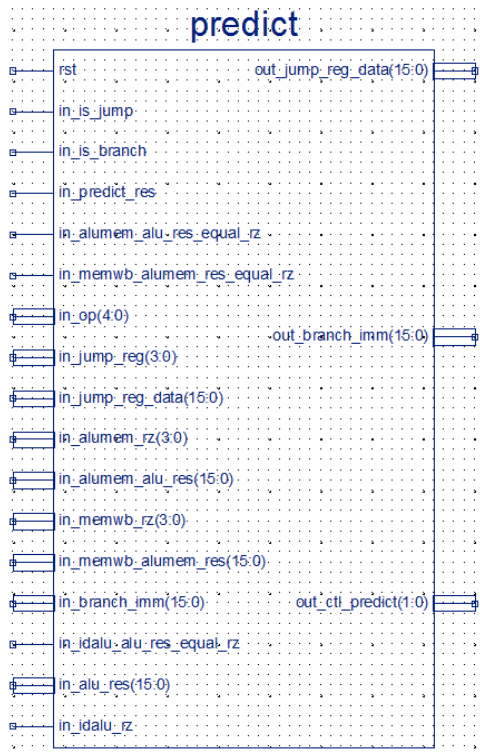


图 8: 跳转单元

表 11: 跳转单元

信号	信号描述
rst	异步清空信号，由外部控制开关接入。
in_is_jump	由 decode 给出的当前指令是不是 jump 指令。
in_is_b	由 decode 给出的当前指令是不是 b 指令。
in_is_branch__except_b	由 decode 给出的当前指令是不是除了 b 的 branch 指令。
in_predict_res	由中央控制单元给出的分支预测的结果。
in_jump_reg	由 decode 给出的 jr 指令的寄存器的值。
in_jump_reg_data	由寄存器堆给出的 jr 指令的寄存器的数据。
in_idalu_alu_res_equal_rc	这是为旁路设计的，用来计算 alu 阶段出现的数据冲突，这个信号表示 alu 阶段 alu 的值会在最后写回 c 寄存器。
in_idalu_rc	当前 alu 阶段 c 寄存器的值。
in_alu_res	当前 alu 阶段 alu 的值。
in_alumem_rc	当前 alu 阶段 c 寄存器的值。

in_alumem_alu__	这是为旁路设计的，用来计算 mem 阶段出现的数据冲突，这个信号
res_equal_rc	表示 mem 阶段 alu 的值会在最后写回 c 寄存器。
in_alumem_alu__	当前 memory 阶段 alu 的值。
res	
in_memwb_rc	当前 writeback 写回的寄存器 c 的值。
in_memwb_alumem__	这是为旁路设计的，用来计算 mem 阶段出现的数据冲突，这个信号
__res_equal_rc	表示 writeback 阶段 alu 或者 mem 的值会在最后写回 c 寄存器。
in_memwb_alumem__	writeback 阶段写回的值。
__res	
in_branch_imm	这个数据表示 branch 指令立即数的值，将会给 pc 模块用来进行加法。
out_jump_reg__data	这个输出用来连接到 pc 模块里面的 4 选 1 选择器，表示 jr 指令的目的地址。
out_branch_imm	这个数据用来输出表示 branch 指令立即数的值，将会给 pc 模块用来进行加法。
out_ctl_predict	这个数据用来输出表示分支预测的结果，连接到选择器上。

2.1.4 寄存器堆

寄存器堆用来输出寄存器的值，以及在时钟下降沿提供修改寄存器值的功能。具体信号请看下表。

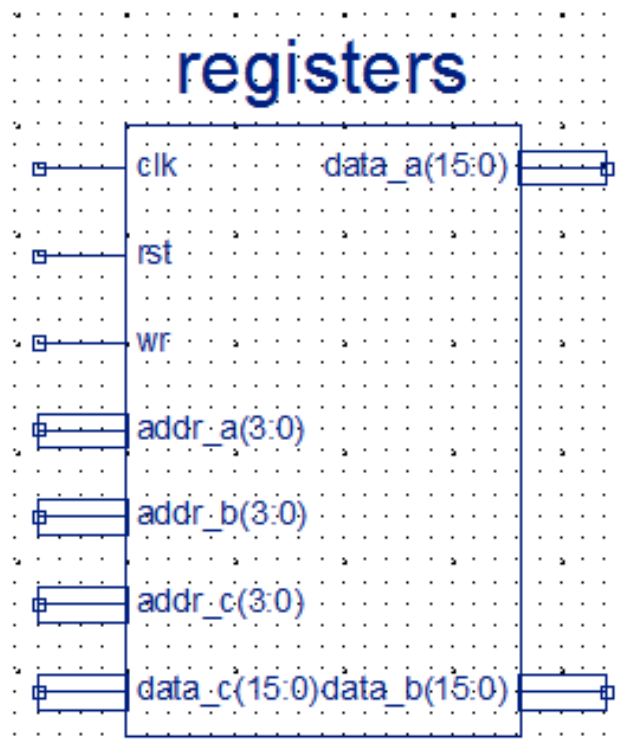


图 9: 寄存器堆

表 13: 寄存器堆

信号	信号描述
clk	CPU 时钟信号。
rst	异步清空信号，由外部控制开关接入。
wr	寄存器堆写使能，如果为高电平则在 clk 下降沿的时候将 data_c 写入对应的 addr_c 里面。
addr_a	接受 decode 出来的寄存器 a。
addr_b	接受 decode 出来的寄存器 b。
addr_c	接受 writeback 将要写回的寄存器 c。
data_a	输出 decode 出来的寄存器 a 的数值。
data_b	输出 decode 出来的寄存器 b 的数值。
data_c	写回阶段修改寄存器的值。

3 外设和中断

3.1 概述

3.1.1 VGA 像素映射的屏幕

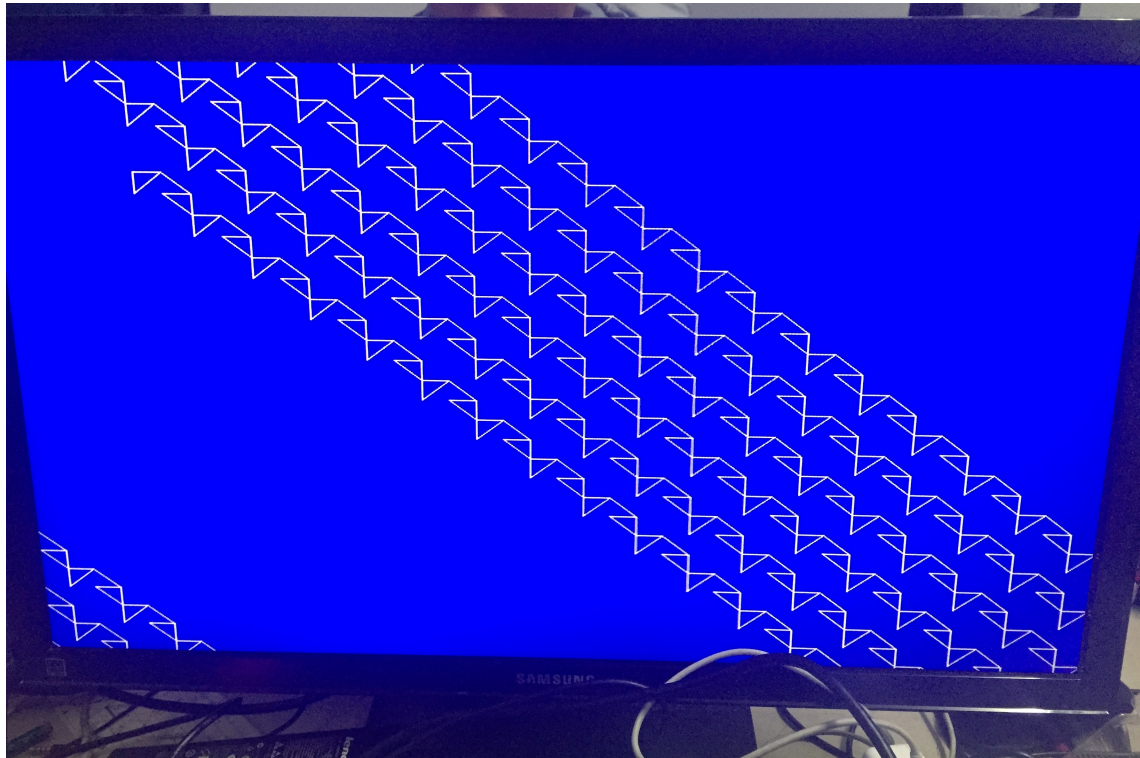


图 10: 漂亮的三角形阵

屏幕为像素映射。我们在片内开了一块等同于屏幕可显示像素数目大小的 RAM 作为我们的显存（使用 ISE 的 IP 核）。我们只需修改显存即可改变屏幕上的内容，由于片内存储空间不大，所以只能显示蓝白两色。

像素映射为我们的计算机增添了通用性。

3.1.2 键盘

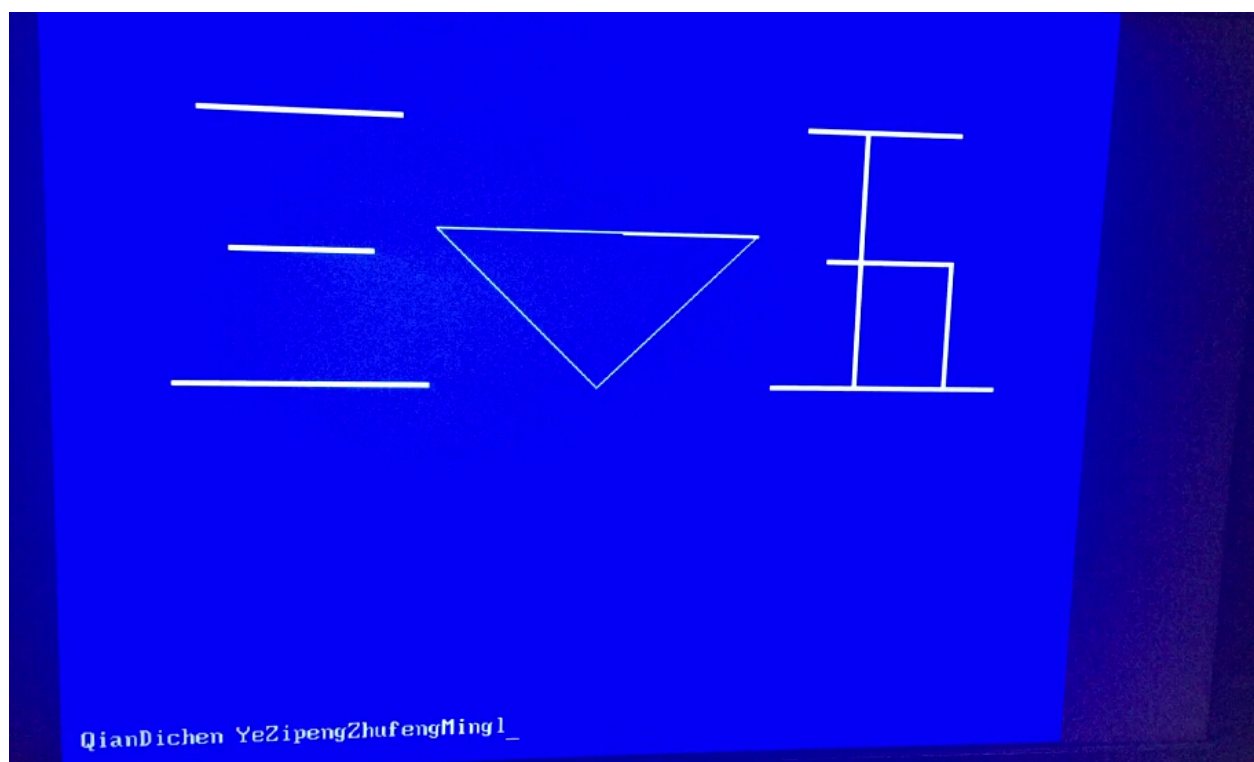


图 11：软件中断和硬件中断的结合

键盘拥有 2 个队列：一个叫做键盘队列，给硬件中断；一个叫做软中断队列，给软件中断。键盘输入一个字符，由硬件转译成 ASCII 码输送给键盘队列，当队列非空并且当时不在执行中断时，发送一个硬件中断信号，执行硬件中断。由硬件中断负责把信息转译到软中断队列。

1. 实现 0 9, a z, 退格键, 空格键, 回车键。硬件将其转化为 ASCII 码传入键盘队列
2. 实现 shift 组合键, 可以发送大写的字母, 以及 0 9 上方的特殊字符
3. 手写实现键盘队列 (长度为 8), 软中断队列 (长度为 16)

3.1.3 硬件中断

硬件中断是结合键盘和中央控制器实现的。每当键盘被按下, 硬件中断被触发, 我们会等待 MEM, 和 WB 段执行完, 清空 ID 和 ALU 段。之后记录返回 PC, 改变 PC 到中断处理程序。

3.2 细节

3.2.1 VGA 像素映射的屏幕

我们将大小为 640×480 的整个屏幕映射到我们的显存上，我们用 1 位（0 或 1）来表示颜色，所以我们的显存空间总共为 19 位。

我们的显存使用的是 ISE 自带的 IP 核，由于片内存储空间仅稍大于屏幕的可显示像素数目，我们只映射 2 种颜色。我们的 RAM 是读写端口分开的，由 CPU 进行写数据，由 VGA 控制器读出数据显示到屏幕上。

VGA 控制器，我们使用了上学期老师发给我们的 VGA 刷屏代码，我们在其之上进行修改。由于我们的显存只够显示两种颜色（蓝色和白色），我们用 0 表示蓝色，1 表示白色，刷屏的时候通过判断值来给 RGB 的线赋值。这样我们即可从我们的显存中读出屏幕的值。

CPU 通过总线来写显存。由于 16 位的字长不足以表示整个屏幕，我们使用 2 个字（BF08 和 BF09）来向显存控制器写一个像素点，第一个字表示地址的高 16 位，第二个字包括地址的地位和写入的数据的 RGB（这里的 RGB 仅仅是留作对软件的接口，由于我们只能显示 2 种颜色，所以我们只用 1 位）。

由于对于我们的这种表示来说，写一个模式（一个字母或者一个图形）对软件的要求极高，我们的工作有一部分也体现在软件上。这体现了我们的硬件的通用性，也考验我们硬件的鲁棒性。

3.2.2 键盘，键盘队列以及软中断队列

键盘的控制器是由上学期老师发的键盘代码修改而成的。

我们的键盘控制器能把通码转成 ASCII 码，滤去键盘不停发送的通码，直到接收到断码为止，也就是说我们的键盘不会导致失误。除此以外，我们支持组合键（shift + 按键可以产生大写的 ASCII 码），空格键，回车键，退格键 · · · · ·

键盘队列和软中断队列都是由我们自己实现。

这个队列是我们定义的一个模块，有 2 个端口，一个端口用于入队（写队列），一个端口用于出队（读队列）。每个端口都有各自的时钟（异步读写）和各自的使能端（如果不使能则队列保持不变）。队列内使用触发器存储数据，使用两个计数器来计数，分别表示队首和队尾。如果队首等于队尾则队列为空，否则队列非空，此时中断队列会发出中断信号，中央控制器会响应中断。

入队过程由上升沿触发，出队过程也是上升沿触发。是否入队/出队由入队/出队使能决定。键盘队列的容量是 8 个字，软中断队列的容量是 16 个字。

3.2.3 硬中断

我们每次按下键盘，键盘会将通码转换成 ASCII 码入队。这时队列的非空信号会触发一个硬件中断（由中央控制器实现）。

此时中央控制器会响应硬件中断，这是一个复杂的过程，整个流程分为如下几步：

1. 锁住 PC 的写使能
2. 存下 EPC
3. 清空流水线
4. 跳转到中断处理程序
5. 执行中断
6. 中断返回

我们详细说明如何清空流水线以及如何中断返回。

确定 EPC：我们会先检查 ID/ALU 阶段寄存器的 PC 值是否为全 0，如果不全 0，我们将选择它。否则选择 IF/ID 段的 PC 值。为什么这样取呢？因为 ID/ALU 段可能是气泡，如果全 0 肯定是气泡，我们不能使用它。而我们整个流水线中不可能有连续 2 个气泡存在，所以选择 IF/ID 是安全的。

清空流水线：由于我们有分支预测，有插气泡的可能，我们认为流水线全部清空是不合适的。因为这时如果有分支预测错误，将不能复原。我们认为让流水线流完也是不合适的，此时 PC 的使能被锁住，如果有跳转将不能正常跳转。我们选择了一个折衷的方法，也是 MIPS 提倡的方法，把 MEM 和 WB 段的指令执行完，而 ID 和 ALU 段的指令清空。这样在我们的架构下不会产生冲突，中断得以正常进行。

中断返回：我们新增了一条 ERET 指令，代码是 FFFF，这条指令是中断执行程序的特权指令，只有在中断正在执行的时候才有效。这条指令会相当于一块 B EPC，表明中断执行结束，需要继续执行指令。

3.2.4 软中断

由软件实现，其过程需要等待硬中断的触发以及处理，通过硬中断把键盘队列中的数据转移到软中断队列中，软中断得以结束。

4 软件方面

4.1 针对我们像素映射实现的接口

1. 给定坐标画一个点

2. 给定起始点，长度，类型（总共 8 个类型，每 45 度角为一类）画一条细线段
3. 给定起始点，长度，类型（同上），粗细，画一条粗线段
4. 给定直角顶点，大小，类型（总共 8 个类型，每 45 度角为一类）画一个等腰直角三角形
5. 从数据 RAM 中读取一个形状（用于实现字符集，比如汉字或是 ASCII 码）

4.2 针对键盘实现的硬件中断和软件中断的中断程序

4.2.1 硬件中断

1. 从键盘输入队列中取出一个 ASCII 码输出到串口
2. 从键盘输入队列中取出一个 ASCII 码输入到软中断队列
3. 从键盘输入队列中取出一个 ASCII 码，通过调用字符显示的函数显示到屏幕
4. 实现退格键

4.2.2 软件中断

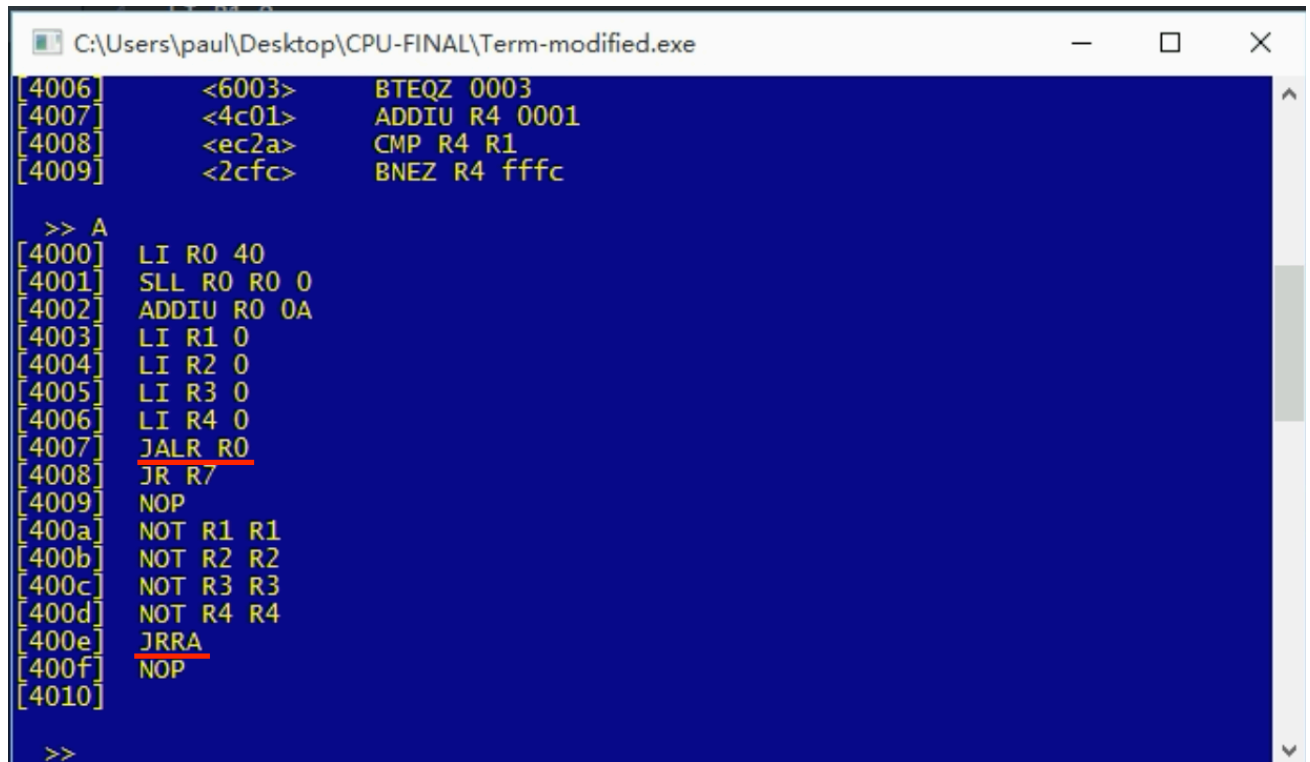
（类似于 scanf()，支持 char 和 int）

1. 从软件中断队列中读取一个 ASCII 码
2. 从软件中断队列中读取一串数字，直到空格，并解析成一个 INT
3. （暂未完成）完成一个画图程序，读入画图指令，解析成参数，传给画图函数实现画图的功能，由于时间紧张，汇编代码（1000+ 行）已经完成，然后没有调试完全，还不能工作。

4.3 定制的 term

4.3.1 概述

由于老师提供的 term 只支持基础指令，而我们需要调试也需要使用扩展的指令。



```
C:\Users\paul\Desktop\CPU-FINAL\Term-modified.exe
[4006]      <6003>      BTEQZ 0003
[4007]      <4c01>      ADDIU R4 0001
[4008]      <ec2a>      CMP R4 R1
[4009]      <2cfc>      BNEZ R4 fffc

>> A
[4000]  LI R0 40
[4001]  SLL R0 R0 0
[4002]  ADDIU R0 0A
[4003]  LI R1 0
[4004]  LI R2 0
[4005]  LI R3 0
[4006]  LI R4 0
[4007]  JALR R0
[4008]  JR R7
[4009]  NOP
[400a]  NOT R1 R1
[400b]  NOT R2 R2
[400c]  NOT R3 R3
[400d]  NOT R4 R4
[400e]  JRRA
[400f]  NOP
[4010]

>>
```

图 12: 扩展的 Term

4.3.2 具体内容及实现

我们自己的 Term 具备老师的 Term 的全部功能，除此以外，我们为它增添了新的特性，包括：

1. 可以识别拓展的 5 条指令，包括 NOT,BTNEZ,SLT,JALR,JRRA
2. 可以向 RAM 中写入拓展的 5 条指令，包括 NOT,BTNEZ,SLT,JALR,JRRA
3. 中断的支持, 对 ERET 的写入

实现这个 term 的过程并不难，因为老师提供了 term 的源码，我们只需要在其 A 指令和 U 指令的模块中增加我们的指令即可。

5 越过的千难万险

5.1 奇偶校验

在单独调试串口的时候，助教早已告诉了我们，要开奇校验。由于我们一直没遇到问题，所以一直都没开奇校验。突然某一天，我们的串口突然不能正常工作了，连续两条串口发来的消息总会有一条出错。我们百思不得其解，我们仔细研究了其出错的情形，寻找出错数据的共同点，最后终于发现了原因。

5.2 奇妙的 A 指令

刚开始的时候，我们的 A 指令总是只能写一条指令，每次写多条的时候，后面的指令都写不进去。

于是我们便开始了单步调试，单步的 Term 根本无法运行，我们只好使用串口模拟 Term 的行为，Term 给串口发什么，我们就给串口发什么。我们按照源码自己解读收到的串口信息。经历了千辛万苦之后，终于尝到了甘泉。我们发现有一个地方的控制出了差错，原来是在修改指令 RAM 的时候，有一种情况没有插气泡。发现了这个错误，我们欣喜若狂，立即修改了这个 BUG，于是我们的 Term 算是可以运行了。

5.3 虚妄的加速

我们的 CPU 不能运行在 25M 主频下。这一点很令我们头疼。

我们研究了 ISE 给我们的时间分析和时间报告，找到了关键路径。

我们开始报告 24M，我们想办法增加到了 29M，又提高到了 36M，最后报告达到了 46M。我们加了各种时间约束，各种约束都通过了。然而最后我们的 CPU 依然承受不住 25M 的洗礼。我认为可能是我们的板子延迟略高于标准值，或者是其它的什么原因导致的。

我们最后只能跑在 21M 的主频下。老师说 21M 和 25M 差不太多，我们对于旁路和分支预测的处理很好，这已经足够了。