THCO MIPS 大作业

设计报告

计32

陈文潇 何轩 李奕昕

目录

[一、 实验概述 3](#_Toc437633233)

[1. 实验目的 3](#_Toc437633234)

[2. 实验环境 3](#_Toc437633235)

[3. 整体设计 3](#_Toc437633236)

[二、IF 15](#_Toc437633237)

[1. 基本架构 15](#_Toc437633238)

[三、主控单元 16](#_Toc437633239)

[1.基本架构 16](#_Toc437633240)

[2.控制信号 17](#_Toc437633241)

[四、寄存器 18](#_Toc437633242)

[1.基本架构 18](#_Toc437633243)

[2.模块接口 19](#_Toc437633244)

[五、EX 21](#_Toc437633245)

[1.基本架构 21](#_Toc437633246)

[2. 模块接口 22](#_Toc437633247)

[六、WR 25](#_Toc437633248)

[1.基本架构 25](#_Toc437633249)

[七、旁路控制 26](#_Toc437633250)

[1.基本架构 26](#_Toc437633251)

[2.模块接口 26](#_Toc437633252)

[八、LoadUse 27](#_Toc437633253)

[1.基本架构 27](#_Toc437633254)

[2.模块接口 27](#_Toc437633255)

[九、内存储器访存 29](#_Toc437633256)

[3. 基本架构 29](#_Toc437633257)

[4. 存储器控制单元 AddrMem 与 DataMem 29](#_Toc437633258)

[5. 存储器单元 Ram1Ctrl, Ram2Ctrl 29](#_Toc437633259)

[6. 访存冲突处理 33](#_Toc437633260)

[十、外接设备 34](#_Toc437633261)

[1. VGA端口模块 34](#_Toc437633262)

[2. PS2键盘模块 35](#_Toc437633263)

[3. Term组合模块 36](#_Toc437633264)

[4. Flash模块 37](#_Toc437633265)

# 实验概述

## 实验目的

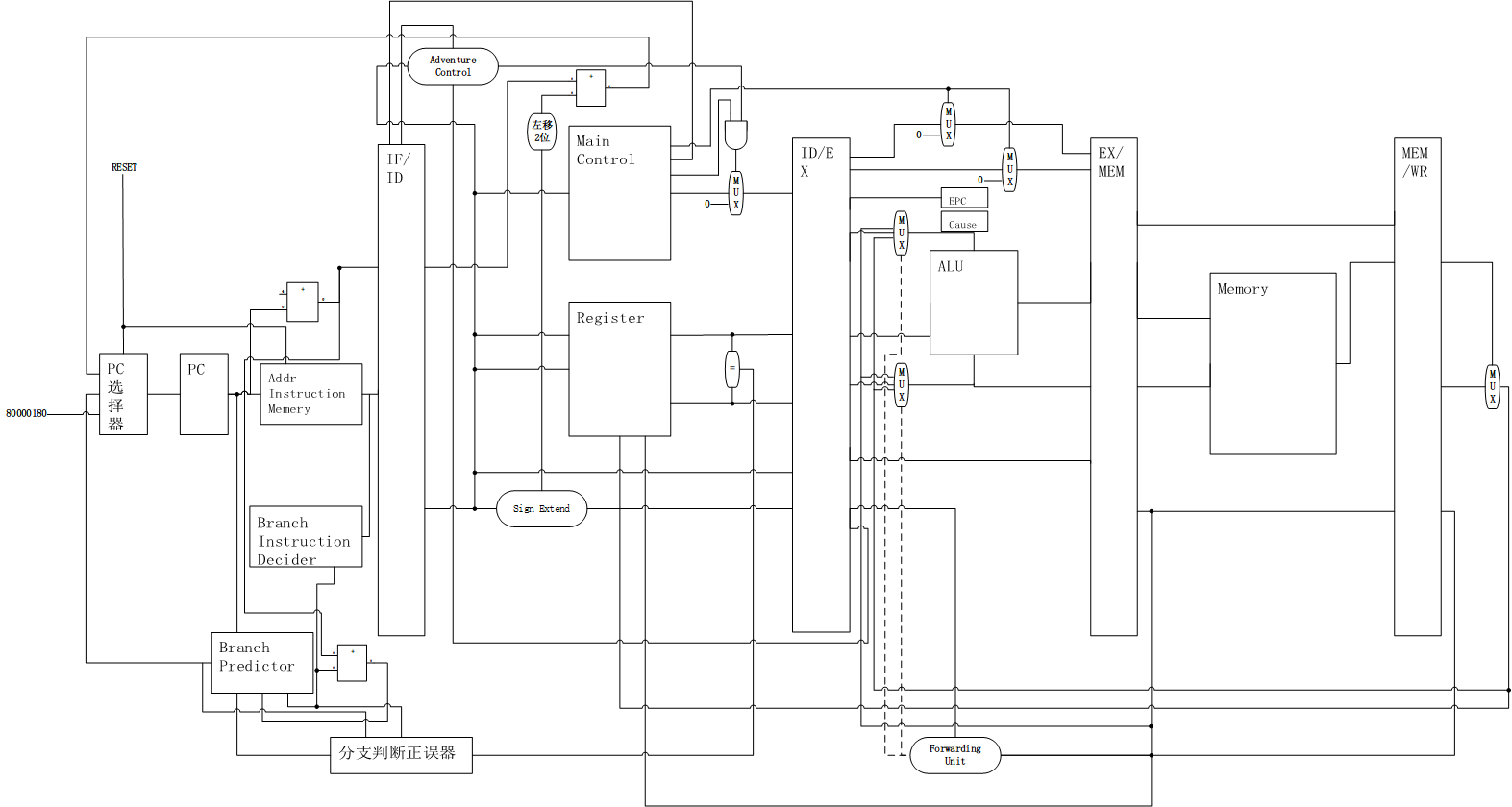
1. 加深对计算机系统知识的理解
2. 进一步理解和掌握流水线结构计算机各个部件组成及内部工作原理
3. 掌握计算机外部输入输出的设计
4. 培养硬件设计和调试的能力

## 实验环境

1. 硬件环境： 安装有Windows 10操作系统的微机，THINPAD教学计算机。其中，THINPAD教学计算机包含：FPGA CPU, RAM Memory, FLASH Memory, UART/USB/PS2/VGA
2. 软件环境： FPGA开发工具软件Xilinx ISE 14.7, THCO MIPS 指令系统，THINPAD教学计算机软件包，包括监控程序、终端程序、数据通信程序、汇编语言等。

## 整体设计

在实验开始初期，我们针对所有的指令设计了一套MIPS五层流水CPU框架，其中包括分支预测，异常处理，数据冲突，结构冲突，控制冲突的处理等。



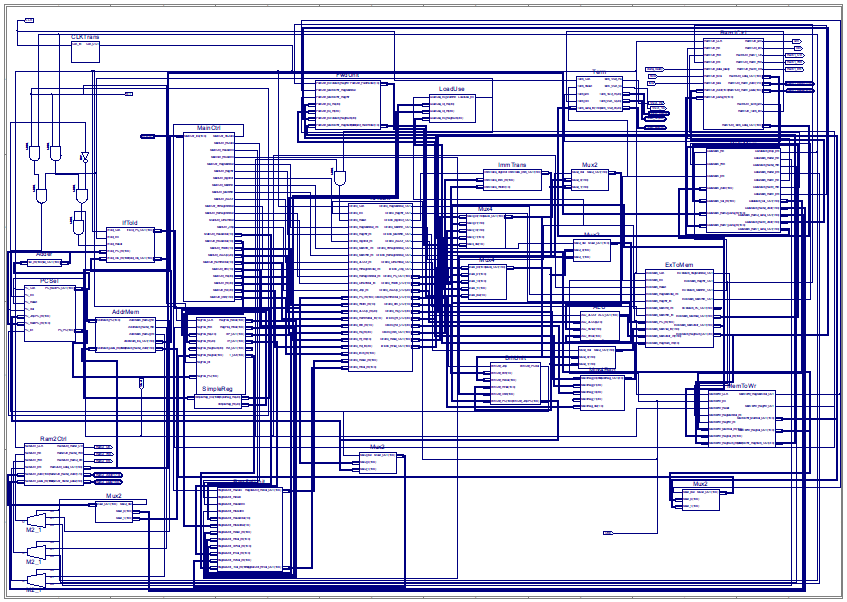
但是在实际的实验中，我们发现有一些在实验中无法调试通过的部分，于是将框架进行了修改：

分支预测部分被删除，采用默认将PC+1代替。

跳转判断转移到Ex，方便进行旁路控制。

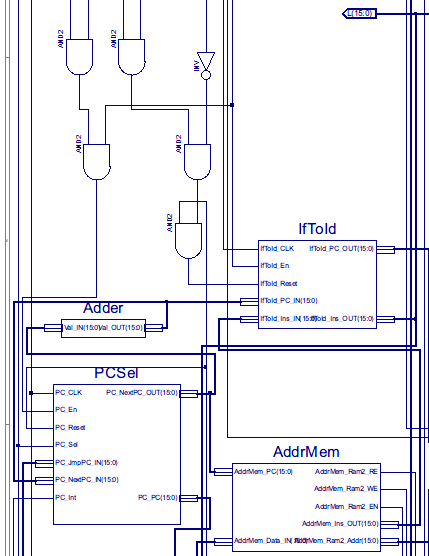
对于结构冲突，将控制IF/ID和PC的使能从指令存储器转移到数据存储器，这样方便控制，同时添加了层次结构：在查询指令的单元中，不涉及Ram2的访问；同样在查询数据的单元中，不涉及Ram1的访问。而是通过使能和数据传输将这些操作传入Ram1Ctrl和Ram2Ctrl后在进行查询或者修改。这样就更加系统和方便扩展了。

将信号具体化，比如上面图中的控制信号只有一根线，但是实际上有十多个信号，为了更加具体，做了更加详细的设计：



为了更加准确的介绍各个部分，以及提高上述图片的分辨率，下面对于各个模块进行介绍。

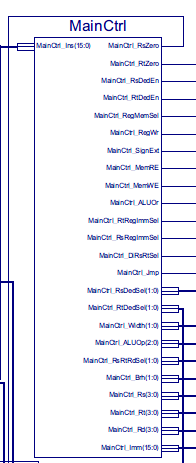
取指令和IF/ID:



这里图片上方的逻辑and和not部分是为了将各个模块的控制信号转换为一个唯一的使能。这里上方的控制信号分别来自中断信号，数据存储器冲突信号，中断信号，数据存储器冲突信号（和之前一样的线）和跳转信号和Load冲突信号（最右）。En用作暂停流水，Reset用作产生气泡。

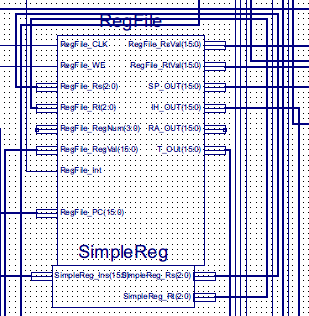
其余的部分就是PCSel选择下一个PC的值，并取指令，再将PC+4传给IF/ID。

主控单元：



根据指令产生控制信号，详细的控制信号表参见主控单元。

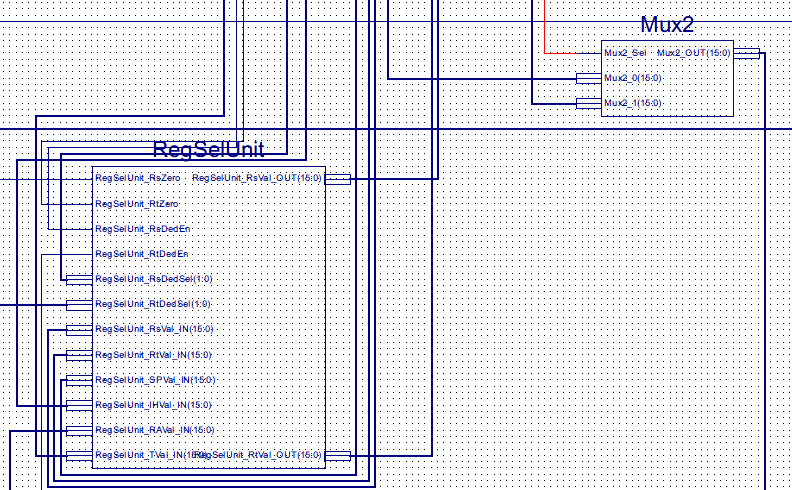
寄存器：



SimpleReg将指令的(10 downto 8)和(7 dwonto 5)分别解析为Rs和Rt，但是由于16位指令并不严格遵守这一条，部分指令这些位置根本不是寄存器编号，所以之后还需要寄存器选择器来控制。

RegFile用作存储和读写寄存器，为了处理中断，这里添加了PC，在中断来临时将PC存储下来，以便恢复现场。对于特殊寄存器SP，IH，RA，T直接输出，由于本组指令中没有涉及RA，可以直接悬空输出。对于通用寄存器，直接输出对应编号的寄存器即可。

寄存器选择器：

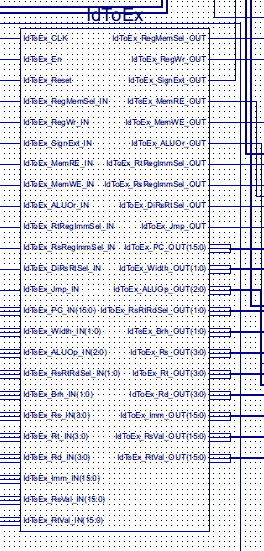


由于指令中有部分指令需要获取PC的值（MFPC），而且没有涉及RA的指令，所以这里可以将RAVal替换为PC的值（不替换也可以，但是需要多一个接口）。PC的值如果上一条是跳转指令，那么使用跳转后的值也就是EX中BrhUnit传给PCSel的值；否则直接使用IF/ID中的PC。

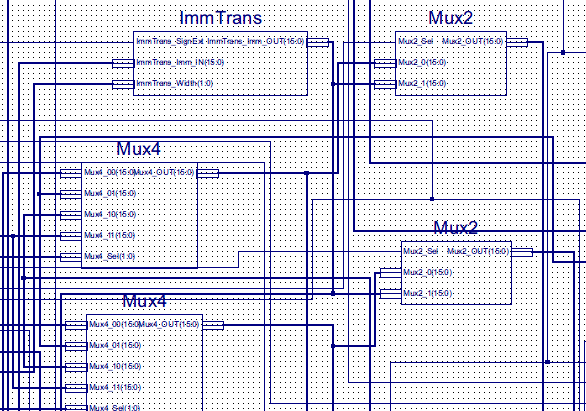
RegSelUnit会针对控制信号，返回正确的寄存器的值，也可能控制信号要求这个值应该是0，那么RsZero或者RtZero将会是1。

更加详细的介绍，参见寄存器部分。

ID/EX：



Ex阶段的旁路选择器，立即数/寄存器选择器，和立即数符号扩展模块：

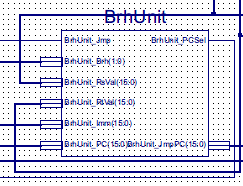


左侧的两个Mux4，分别对于Rs和Rt进行旁路选择，旁路分别来自EX/MEM的ALU计算结果（也是存储地址），MEM/WR的内存值和寄存器值。

ImmTrans会根据控制信号对于立即数进行改变，如符号扩展和将0000变为1000（对于SLL等命令）。

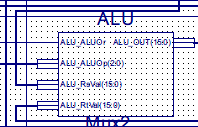
右侧的两个Mux2会根据控制信号选择传入ALU的值是来自寄存器还是立即数。

Ex阶段的跳转控制单元：



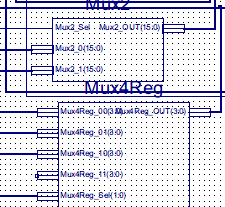
BrhUnit接受当前的PC和转化后的立即数，根据控制信号选择是否跳转以及跳转的位置。

简单的ALU单元：



根据传入的操作数来选择对于两个输入RsVal，RtVal进行哪种操作，并输出到OUT中。

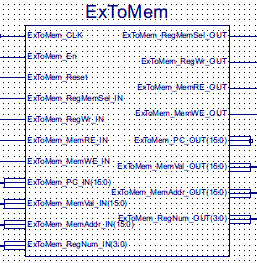
分别是存入内存的值选择器和目标寄存器选择器：



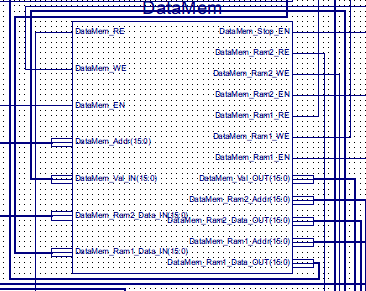
由于LW和LW\_SP存入的值分别是Rx和Ry的值，所以对于存入MEM的值也需要选择。

由于不同指令改变的目标寄存器不一样，需要选择。

EX/MEM：

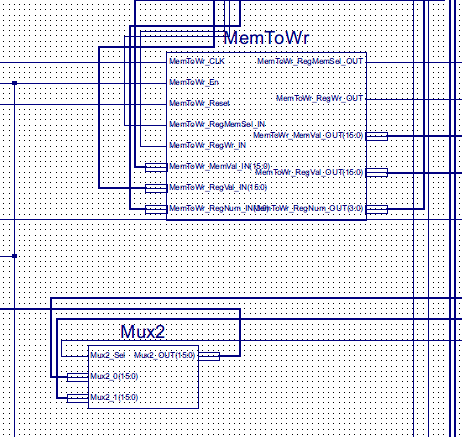


数据存储器：

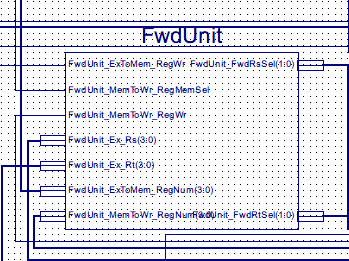


比指令存储器更加复杂，因为他需要和Ram1Ctrl，Ram2Ctrl沟通，并在产生冲突的时候发出控制信号，这个控制信号需要暂停流水并产生NOP，也要控制传入Ram2的各种信号是来自于数据存储器还是指令存储器。

MEM/WR和写回值选择器：

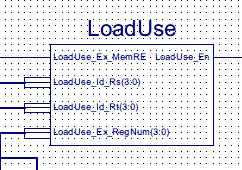


旁路控制：



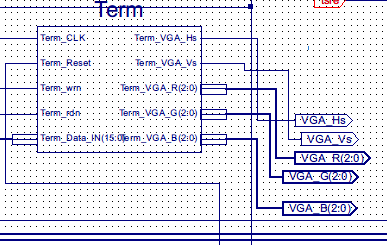
旁路控制单元，对于Ex的Mux4提供选择信号，使得MEM和WR的数据能够通过旁路传到EX处。这里对于Rs和Rt一起进行了处理，同样也可以分开。

Load冲突控制：



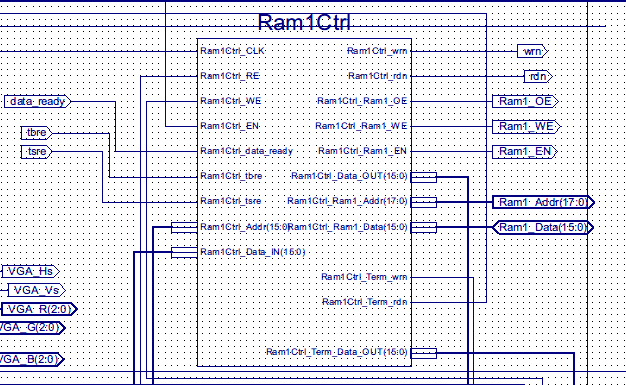
如果EX是访问存储器并且准备写回的寄存器是ID处需要的Rs，Rt中的某一个，那么应该暂停ID处的流水。具体地，在EX处插入一个NOP，并且保持PCSel，IF/ID不变。

VGA控制：



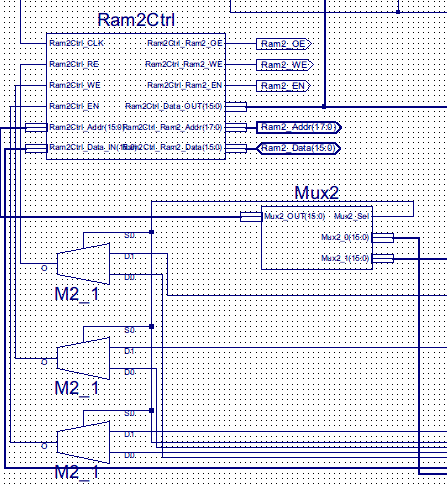
控制在VGA上显示字符的单元，由Ram1Ctrl控制传入信号。

Ram1：



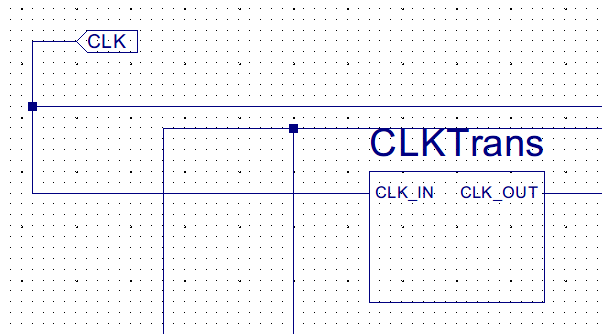
与串口，VGA，PS2，Ram1进行交互的单元。

Ram2：



与Ram2交互的单元。

CLK分频器：



对于50MHz的时钟进行分频处理，使得能够调整CPU主频。

之后会对各个模块进行更加详细的介绍，详见下文。

# 二、IF

## 基本架构

IF由PCSel，Adder和AddrMem组成：

PCSel的接口如下：

|  |
| --- |
|  |
| entity PCSel is |
| Port ( PC\_CLK : in STD\_LOGIC; | |
| PC\_En : in STD\_LOGIC; | |
| PC\_Reset : in STD\_LOGIC; | |
| PC\_Sel : in STD\_LOGIC; | |
| PC\_Int : in STD\_LOGIC; | |
| PC\_JmpPC\_IN : in STD\_LOGIC\_VECTOR(15 downto 0); | |
| PC\_NextPC\_IN : in STD\_LOGIC\_VECTOR (15 downto 0); | |
| PC\_NextPC\_OUT : out STD\_LOGIC\_VECTOR (15 downto 0); | |
| PC\_PC : out STD\_LOGIC\_VECTOR(15 downto 0)); | |
| end PCSel; | |

当PC\_En为0时，PCSel将会暂停，既保持不接受下一个时钟沿来临时的下一个PC。

当PC\_Reset为0时，PC会在下一个时钟沿来临时被清空为0。

当PC\_Int为0时，PC会在下一个时钟沿来临时被设为中断处理的地址。

当PC\_Sel为1时，PC会在下一个时钟沿来临时被设为PC\_JmpPC\_IN否则被设为PC\_NextPC\_IN。

PC\_NextPC\_OUT是当前输出的PC的值。

PC\_PC是PC寄存器的值，与上者不一定相等，原因在于上者在上升沿才会改变，而这个是直接连线的。

Adder是一个基础的加法器，将PC+1并输出。

AddrMem可以参见内存储器访存部分。

# 三、主控单元

## 1.基本架构

MainCtrl的接口如下：

|  |
| --- |
| Entity MainCtrl is |
| Port ( MainCtrl\_Ins : in STD\_LOGIC\_VECTOR (15 downto 0); | |
| MainCtrl\_RsZero : out STD\_LOGIC; | |
| MainCtrl\_RtZero : out STD\_LOGIC; | |
| MainCtrl\_RsDedEn : out STD\_LOGIC; | |
| MainCtrl\_RsDedSel : out STD\_LOGIC\_VECTOR(1 downto 0); | |
| MainCtrl\_RtDedEn : out STD\_LOGIC; | |
| MainCtrl\_RtDedSel : out STD\_LOGIC\_VECTOR(1 downto 0); | |
| MainCtrl\_RegMemSel : out STD\_LOGIC; | |
| MainCtrl\_RegWr : out STD\_LOGIC; | |
| MainCtrl\_SignExt : out STD\_LOGIC; | |
| MainCtrl\_Width : out STD\_LOGIC\_VECTOR(1 downto 0); | |
| MainCtrl\_MemRE : out STD\_LOGIC; | |
| MainCtrl\_MemWE : out STD\_LOGIC; | |
| MainCtrl\_ALUOp : out STD\_LOGIC\_VECTOR(2 downto 0); | |
| MainCtrl\_ALUOr : out STD\_LOGIC; | |
| MainCtrl\_RtRegImmSel : out STD\_LOGIC; | |
| MainCtrl\_RsRegImmSel : out STD\_LOGIC; | |
| MainCtrl\_RsRtRdSel : out STD\_LOGIC\_VECTOR(1 downto 0); | |
| MainCtrl\_DiRsRtSel : out STD\_LOGIC; | |
| MainCtrl\_Brh : out STD\_LOGIC\_VECTOR(1 downto 0); | |
| MainCtrl\_Jmp : out STD\_LOGIC; | |
| MainCtrl\_Rs : out STD\_LOGIC\_VECTOR(3 downto 0); | |
| MainCtrl\_Rt : out STD\_LOGIC\_VECTOR(3 downto 0); | |
| MainCtrl\_Rd : out STD\_LOGIC\_VECTOR(3 downto 0); | |
| MainCtrl\_Imm : out STD\_LOGIC\_VECTOR(15 downto 0)); | |
|  | |
| end MainCtrl; | |

MainCtrl的功能是对于一个指令Ins：

* + 找到所需要的寄存器编号Rs，Rt，Rd。
  + 对于立即数进行初步的译码。
  + 设置控制信号。

## 2.控制信号

我们设置了如下的控制信号；

RsZero，表示选择的Rs是否是0。

RtZero，表示选择的Rt是否是0。

RsDedEn，表示Rs是否是特殊寄存器。

RsDedSel，表示Rs的特殊寄存器编号。

RtDedEn，表示Rt是否是特殊寄存器。

RtDedSel，表示Rt的特殊寄存器编号。

RegMemSel，表示选择写回寄存器的是访存的值还是ALU计算后的值。

RegWr，表示是否写回寄存器。

SignExt，表示立即数是否进行符号扩展。

Width，表示立即数的宽度，如果是符号扩展，那么总共有4,5,8,10这4种，如果不是符号扩展，那么有3和8两种。

MemRE，表示是否读内存。

MemWE，表示是否写内存。

ALUOp，表示ALU的操作数。

ALUOr，表示ALU是否进行Or操作（Or操作比较特别，因为一共9种操作，其余刚好占满000~111）。

RtRegImmSel，表示Rt是选择寄存器的值还是立即数的值。

RsRegImmSel，表示Rs是选择寄存器的值还是立即数的值。

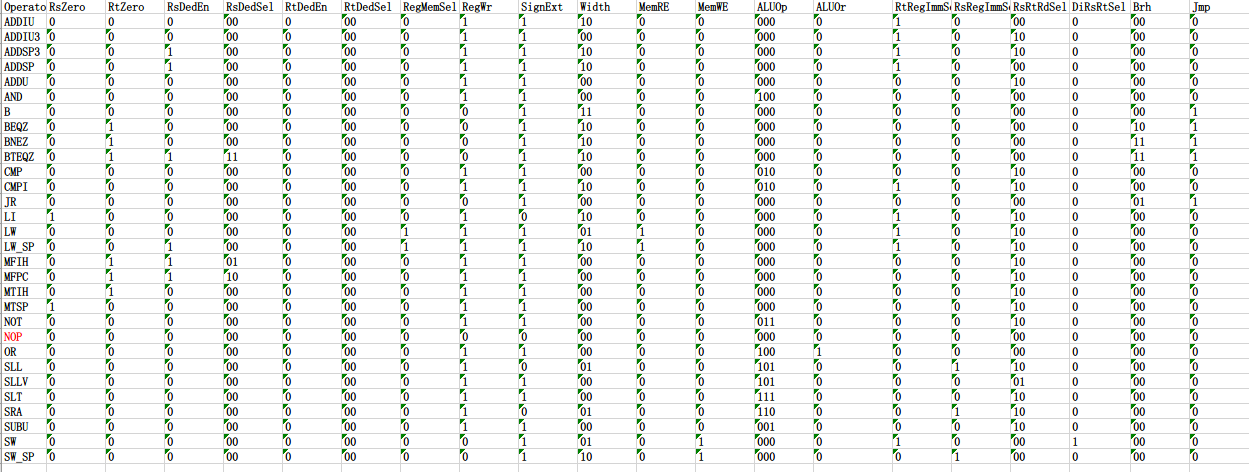
RsRtRdSel，表示选择的目的寄存器是Rs，Rt还是Rd。

DiRsRtSel，表示存入内存的是Rs还是Rt。

Brh，跳转的种类，总共有4种跳转。

Jmp，是否可以进行跳转。

这里给出一张对应指令的控制信号的表：



# 四、寄存器

## 1.基本架构

寄存器由三个部分组成SimpleReg，RegFile，RegSelUnit。

SimpleReg是将指令中的Rs和Rt部分提取出来方便RegFile进行查询。RegFile进行寄存器的查询和读写。RegSelUnit主要负责对于寄存器的值根据控制信号进行选择。

这样设计的目的是在主控单元还未完成时也可以进行寄存器的读写，然后根据控制信号选择正确的值，在时间上大部分和主控单元是并行的，效率上会更快速。

顺序上是逐层递进的：

## 2.模块接口

SimpleReg过于简单所以不在此累述，对于RegFile：

|  |
| --- |
| entity RegFile is |
| Port ( RegFile\_CLK : in STD\_LOGIC; | |
| RegFile\_WE : in STD\_LOGIC; | |
| RegFile\_Rs : in STD\_LOGIC\_VECTOR (2 downto 0); | |
| RegFile\_Rt : in STD\_LOGIC\_VECTOR (2 downto 0); | |
| RegFile\_Int : in STD\_LOGIC; | |
| RegFile\_PC : in STD\_LOGIC\_VECTOR (15 downto 0); | |
| RegFile\_RegNum : in STD\_LOGIC\_VECTOR (3 downto 0);--For write --000~111 gr, 1000~1011 SP IH RA T | |
| RegFile\_RegVal : in STD\_LOGIC\_VECTOR (15 downto 0);--For write | |
| RegFile\_RsVal : out STD\_LOGIC\_VECTOR (15 downto 0); | |
| RegFile\_RtVal : out STD\_LOGIC\_VECTOR (15 downto 0); | |
| SP\_OUT : out STD\_LOGIC\_VECTOR (15 downto 0); | |
| IH\_OUT : out STD\_LOGIC\_VECTOR (15 downto 0); | |
| RA\_OUT : out STD\_LOGIC\_VECTOR (15 downto 0); | |
| T\_OUt : out STD\_LOGIC\_VECTOR (15 downto 0)); | |
| end RegFile; | |

RegFile在读取时是一个组合逻辑，将对应的寄存器的值返回即可，这里会同时返回Rs和Rt的值，还有4个特殊寄存器的值。

但是RegFile在写入时要特别注意，由于写入的值在上升沿时还不稳定，所以要调整到时钟下降沿写入。具体地，当时钟是下降沿，并且WE是1时，将RegVal写入RegNum对应的寄存器。寄存器对应表如下：

0000：0号寄存器

0001：1号寄存器

0010：2号寄存器

0011：3号寄存器

0100：4号寄存器

0101：5号寄存器

0110：6号寄存器

0111：7号寄存器

1000：SP

1001：IH

1010：RA

1011：T

1111：无意义寄存器，当寄存器是常数值时防止写回或旁路调用的。

特别的，为了使得寄存器能保存中断前的值，当中断信号来临时，也就是Int的下降沿存储PC，并在下一个时钟上升沿来临时将对应的寄存器改变。具体来说，信号的转移是PC->PC\_tmp(Int的下降沿)，PC\_tmp->R2(时钟上升沿)。

也就是说，终端信号至少要维持一个时钟周期的低电平，实际上是5个周期，因为中断还要等待流水完成。

RegSelUnit的接口如下：

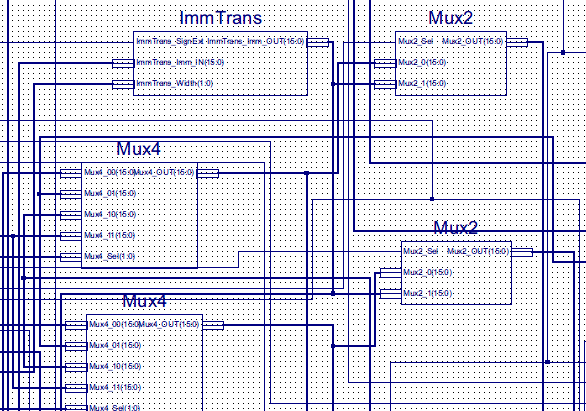
|  |
| --- |
| entity RegSelUnit is |
| Port ( RegSelUnit\_RsZero : in STD\_LOGIC; | |
| RegSelUnit\_RtZero : in STD\_LOGIC; | |
| RegSelUnit\_RsDedEn : in STD\_LOGIC; | |
| RegSelUnit\_RsDedSel : in STD\_LOGIC\_VECTOR (1 downto 0); | |
| RegSelUnit\_RtDedEn : in STD\_LOGIC; | |
| RegSelUnit\_RtDedSel : in STD\_LOGIC\_VECTOR (1 downto 0); | |
| RegSelUnit\_RsVal\_IN : in STD\_LOGIC\_VECTOR (15 downto 0); | |
| RegSelUnit\_RtVal\_IN : in STD\_LOGIC\_VECTOR (15 downto 0); | |
| RegSelUnit\_SPVal\_IN : in STD\_LOGIC\_VECTOR (15 downto 0); | |
| RegSelUnit\_IHVal\_IN : in STD\_LOGIC\_VECTOR (15 downto 0); | |
| RegSelUnit\_RAVal\_IN : in STD\_LOGIC\_VECTOR (15 downto 0); | |
| RegSelUnit\_TVal\_IN : in STD\_LOGIC\_VECTOR (15 downto 0); | |
| RegSelUnit\_RsVal\_OUT : out STD\_LOGIC\_VECTOR (15 downto 0); | |
| RegSelUnit\_RtVal\_OUT : out STD\_LOGIC\_VECTOR (15 downto 0)); | |
| end RegSelUnit; | |

其中RsZero，RtZero，RsDedEn，RtDedEn，RsDedSel，RtDedSel是来自于主控单元。特别的，当RsZero，RtZero为1时，对应的Rs，Rt输出为0，其余情况下进行特殊寄存器判断，输出通用或者特殊寄存器。

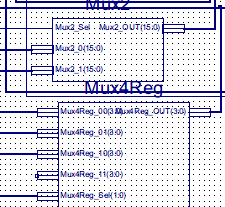
# 五、EX

## 1.基本架构

EX段由若干选择器，ALU，译码，跳转单元组成。左侧两个Mux4是旁路选择器，右侧两个是ALU输入是来自立即数还是寄存器的选择器。



这两个选择器是选择目的寄存器和存入内存的值。



ALU接受经过转换的RsVal和RtVla进行运算，并输出到下一个阶段。

而BrhUnit负责跳转，接受经过转换的立即数，RsVal，RtVal和PC，计算应该跳转到的位置。

## 模块接口

ALU部分的计算会非常简单明确：

|  |
| --- |
| entity ALU is |
| Port ( ALU\_ALUOp : in STD\_LOGIC\_VECTOR (2 downto 0); | |
| ALU\_ALUOr : in STD\_LOGIC; | |
| ALU\_RsVal : in STD\_LOGIC\_VECTOR (15 downto 0); | |
| ALU\_RtVal : in STD\_LOGIC\_VECTOR (15 downto 0); | |
| ALU\_OUT : out STD\_LOGIC\_VECTOR (15 downto 0)); | |
| end ALU; | |
|  | |
|  | |
|  | architecture Behavioral of ALU is |
|  |  |
|  | begin |
|  |  |
|  | ALU\_OUT <= ALU\_RsVal + ALU\_RtVal when ALU\_ALUOp = "000" else |
|  | ALU\_RsVal - ALU\_RtVal when ALU\_ALUOp = "001" else |
|  | "0000000000000001" when ALU\_ALUOp = "010" and ALU\_RsVal /= ALU\_RtVal else |
|  | NOT ALU\_RtVal when ALU\_ALUOp = "011" else |
|  | ALU\_RsVal OR ALU\_RtVal when ALU\_ALUOp = "100" and ALU\_ALUOr = '1' else |
|  | ALU\_RsVal AND ALU\_RtVal when ALU\_ALUOp = "100" and ALU\_ALUOr = '0' else |
|  | to\_stdlogicvector(to\_bitvector(ALU\_RtVal) SLL conv\_integer(ALU\_RsVal)) when ALU\_ALUOp = "101" else |
|  | to\_stdlogicvector(to\_bitvector(ALU\_RtVal) SRA conv\_integer(ALU\_RsVal)) when ALU\_ALUOp = "110" else |
|  | "0000000000000001" when ALU\_RsVal < ALU\_RtVal else |
|  | "0000000000000000"; |
|  |  |
|  | end Behavioral; |

由于ALU部分非常简单，可以直接使用代码描述，支持9种操作：

* + 加法
  + 减法
  + 判断不相等
  + 取非
  + 或操作
  + 与操作
  + 左移
  + 右移
  + 判断小于

BrhUnit与ALU其实类似：

|  |
| --- |
|  |
| entity BrhUnit is |
| Port ( BrhUnit\_Brh : in STD\_LOGIC\_VECTOR (1 downto 0); | |
| BrhUnit\_Jmp : in STD\_LOGIC; | |
| BrhUnit\_RsVal : in STD\_LOGIC\_VECTOR (15 downto 0); | |
| BrhUnit\_RtVal : in STD\_LOGIC\_VECTOR (15 downto 0); | |
| BrhUnit\_Imm : in STD\_LOGIC\_VECTOR (15 downto 0); | |
| BrhUnit\_PC : in STD\_LOGIC\_VECTOR (15 downto 0); | |
| BrhUnit\_JmpPC : out STD\_LOGIC\_VECTOR (15 downto 0); | |
| BrhUnit\_PCSel : out STD\_LOGIC);--PCSel's NOT connneted to IfToId.Reset | |
| end BrhUnit; | |
|  |  |
|  | architecture Behavioral of BrhUnit is |
|  |  |
|  | begin |
|  |  |
|  | BrhUnit\_JmpPC <= BrhUnit\_PC + BrhUnit\_Imm when BrhUnit\_Brh = "00" else |
|  | BrhUnit\_RsVal when BrhUnit\_Brh = "01" else |
|  | BrhUnit\_PC + BrhUnit\_Imm when BrhUnit\_Brh = "10" else |
|  | BrhUnit\_PC + BrhUnit\_Imm when BrhUnit\_Brh = "11"; |
|  |  |
|  | BrhUnit\_PCSel <= BrhUnit\_Jmp when BrhUnit\_Brh = "00" else |
|  | BrhUnit\_Jmp when BrhUnit\_Brh = "01" else |
|  | BrhUnit\_Jmp when BrhUnit\_Brh = "10" and BrhUnit\_RsVal = BrhUnit\_RtVal else |
|  | BrhUnit\_Jmp when BrhUnit\_Brh = "11" and BrhUnit\_RsVal /= BrhUnit\_RtVal else |
|  | '0'; |
|  |  |

设计有4种跳转：

* + 无条件跳转到PC+IMM
  + 无条件跳转到Rs
  + 如果Rs=Rt，跳转到PC+IMM
  + 如果Rs!=Rt，跳转到PC+IMM

特别的，当确定要跳转时，需要在IF/ID处插入一个NOP，否则延迟槽将会是2个。

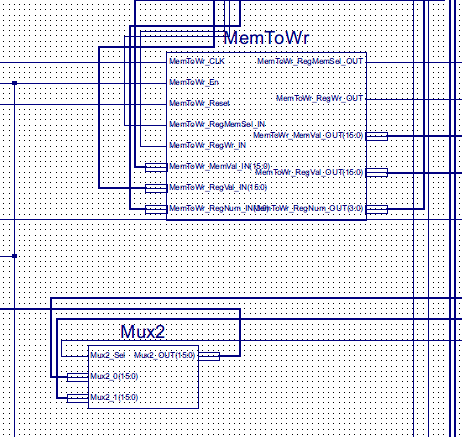
# 六、WR

## 1.基本架构

这个模块十分简单，仅仅只包含一个写回选择器。

对于数据存储器返回的MemVal和ALU计算出来的RegVal进行写回，先要进行一个次选择，如果RegMemSel是0，那么选择RegVal，否则选择MemVal。

然后将RegWr传回给RegFile的写使能，这样RegFile在时钟的下降沿会根据RegNum和对应的写入的值进行一次写入，由于这个部分仅有一个选择器的延迟，可以保证数据是稳定的。



# 七、旁路控制

## 1.基本架构

旁路控制由于处理数据冲突，当Ex处需要的寄存器的值在上一步或者上上一步被修改过时，需要通过旁路重传回来进行选择，这样才能保证计算的正确性。

旁路单元接受EX/MEM的目的寄存器和是否写回信号，MEM/WR的是否写回信号和目的寄存器以及RegMemSel的值。旁路的输出使能控制着EX处Rs和Rt的值是从哪个旁路来的还是从ID/EX锁存器来的。

## 2.模块接口

|  |
| --- |
| entity FwdUnit is |
| Port ( | |
| FwdUnit\_Ex\_Rs : in STD\_LOGIC\_VECTOR (3 downto 0); | |
| FwdUnit\_Ex\_Rt : in STD\_LOGIC\_VECTOR (3 downto 0); | |
| FwdUnit\_ExToMem\_RegWr : in STD\_LOGIC; | |
| FwdUnit\_ExToMem\_RegNum : in STD\_LOGIC\_VECTOR (3 downto 0); | |
| FwdUnit\_MemToWr\_RegMemSel : in STD\_LOGIC; | |
| FwdUnit\_MemToWr\_RegWr : in STD\_LOGIC; | |
| FwdUnit\_MemToWr\_RegNum : in STD\_LOGIC\_VECTOR (3 downto 0); | |
| FwdUnit\_FwdRsSel : out STD\_LOGIC\_VECTOR (1 downto 0); | |
| FwdUnit\_FwdRtSel : out STD\_LOGIC\_VECTOR (1 downto 0)); | |
| --"00" : RsVal "01" ExToMem "10" WrReg "11" WrMem | |
| end FwdUnit; | |

* 由于可能在EX/MEM和MEM/WR的写回寄存器是一样的，所以应该优先判断EX/MEM处的写回寄存器和EX的Rs，Rt是否相同并且确定要写回，那么旁路选择设置为01。
* 如果上者不成立，并且在MEM/WR处的写回寄存器和EX的Rs，Rt是否相同并且确定要写回，那么根据MEM/WR的RegMemSel来选择，如果要写回的是Reg的值，那么旁路选择设置为10。
* 如果要写回的是Mem的值，那么旁路选择设置11。
* 如果上述都不满足，设置为00，表示采用ID/EX锁存器传来的值。

这里需要对于Rs和Rt分别判断，可以使用将两个并行的思路。

# 八、LoadUse

## 1.基本架构

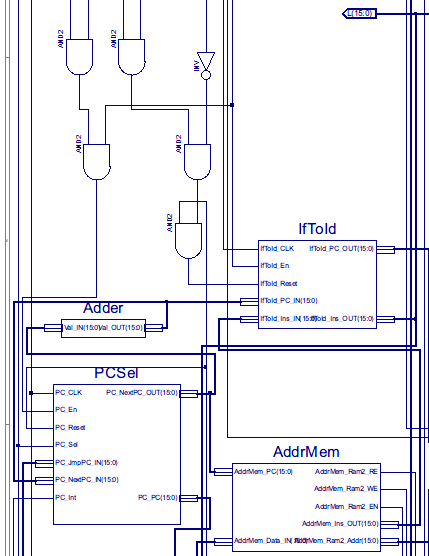
LoadUse是解决当前一个指令是从内存中取值，但是下一个指令立即要使用这个寄存器的冲突。方法是当LW到达EX而下一条指令到达ID时，下一个周期来临在EX处插入一个NOP，并让PCSel和IF/ID保持原值。

## 2.模块接口

|  |
| --- |
| entity LoadUse is |
| Port ( LoadUse\_Id\_Rs : in STD\_LOGIC\_VECTOR (3 downto 0); | |
| LoadUse\_Id\_Rt : in STD\_LOGIC\_VECTOR (3 downto 0); | |
| LoadUse\_Ex\_RegNum : in STD\_LOGIC\_VECTOR (3 downto 0); | |
| LoadUse\_Ex\_MemRE : in STD\_LOGIC; | |
| LoadUse\_En : out STD\_LOGIC);--Connnected to PC.En, IfToId.En, IdToEx.Reset | |
| end LoadUse; | |

当确认EX处是要访问存储器，并且要写回的寄存器和ID处的Rs或者Rt一致时，将输出的使能置为0。

这个使能需要接到IF处，可以回顾一下上图：



其中接到IF/ID的使能处的便是LoadUse的输出使能，可以控制PCSel和IF/ID不变。

另外还需要接到ID/EX处的Reset位置，使得下一个EX是NOP。

# 九、内存储器访存

## 基本架构

在数据通路的设计中,我们采用了将指令存储器（AddrMem）与数据存储器(DataMem)分离共两个单元的设计。基于THINKPAD教学计算机的基本设计，AddrMem与DataMem功能相分离，AddrMem单元负责对Ram2的读操作，主要用于实现读出Ram2中存储的指令。DataMem单元负责对Ram1,Ram2的读与写操作。可以发现，这二者在对Ram2的操作中存在一定冲突，二者可能会同时对Ram2进行操作，因此在此处访存的冲突处理应当被考虑在内。

## 存储器控制单元 AddrMem 与 DataMem

* 1. AddrMem指令存储器控制单元

AddrMem单元的主要功能是实现对Ram2存储器中指令的读取，即其只涉及对Ram2存储器的读操作。

由于AddrMem功能相对简单，只负责对Ram2存储器的读操作，因而其作为访问Ram2的入口，内部为组合逻辑，直接将其输入进行判断操作后与其输出相接。

* 1. DataMem数据存储器控制单元

DataMem单元的主要功能是与其他单元协作，实现系统及用户数据的存取，以及在需要时对Ram2指令存储器的读取与修改。功能提取可以了解到，相关功能涉及对Ram1与Ram2的读与写。与此同时，由于串口以及VGA与键盘相关的读写基本等同于对Ram1的读写，DataMem同时需支持对串口与VGA+键盘的控制。

DataMem相较AddrMem而言较为复杂，涉及对Ram1Ctrl, Ram2Ctrl 两个单元进行读写控制，其内部依然如同AddrMem单元，采用组合逻辑，效率相比时序逻辑更高。由于需要对输入数据进行判断，并根据访存地址的不同将输入转接到不同存储单元的对应接口，我们这里采用效率较高的when else 语句，如下：

DataMem\_Ram1\_EN <= '1' when DataMem\_Addr >= "1011111100000000" and DataMem\_EN = '1' else '0';

## 存储器单元 Ram1Ctrl, Ram2Ctrl

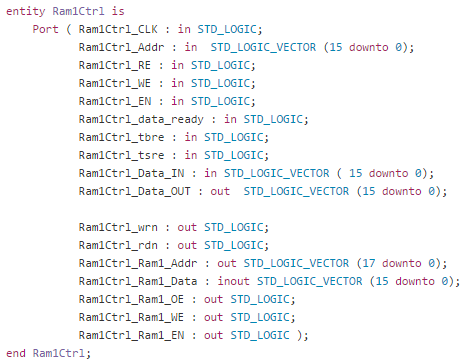
1）内存访问时序及信号

内存使用实验板上的Ram1与Ram2两个存储单元，其中地址Addr为18位，Data为16位。本次CPU设计中暂不使用Addr的高2位，默认置为0。

2）Ram1Ctrl 单元设计

Ram1与Ram2的地址段划分不同，Ram1的地址范围为： 0xBF00~0xFFFF。其中0xBF00~0xBF03看似为Ram1中的地址，事实上为串口的访问地址。

Ram1Ctrl单元的相关接口设计如下：



其中，Ram1Ctrl\_EN, Ram1Ctrl\_RE, Ram1Ctrl\_WE分别为总使能，读使能，写使能。Ram1Ctrl\_Data\_IN为写入时需写入的数据，Ram1Ctrl\_Data\_OUT为读取时读到的数据。Ram1Ctrl\_data\_ready, Ram1Ctrl\_tbre,Ram1\_Ctrl\_tsre,Ram1\_wrn,Ram1\_rdn都为与串口交互相关的接口。至于下方5个接口则是直接通过总线与Ram1单元交互的接口。之所以单独设计Ram1Ctrl单元而不是将其功能综合进DataMem单元，是由于AddrMem与DataMem可能会同时读写Ram2单元，这其中，Data是inout类型的接口，对其作选择器较为困难，故将Ram的读写单元单独设计，以实现对inout接口的分离。

关于内存部分的读写时序如下图：

读时序：



写时序：



考虑到流水线运行的效率，我们摒弃了在之前做的内存储器实验中采用的状态机转移方法来控制读写逻辑，而是采用了高低电平判断的方式来使得一个周期内可以包含两种状态。每个时钟周期默认所有使能为关。

数据读写部分： Addr > 0xBF03

读状态：读使能开启写使能关闭即Ram1Ctrl\_RE = ‘1’ and Ram1Ctrl\_WE = ‘0’。此时若为时钟高电平则读入地址Addr并开启Ram1的总使能Ram1\_EN，若为低电平则开启Ram1的读使能Ram1\_OE。

写状态：读使能关闭写使能开启即Ram1Ctrl\_RE = ‘0’ and Ram1Ctrl\_WE = ‘1’。此时若为时钟高电平则读入地址Addr并开启Ram1的总使能Ram1\_EN，若为低电平则开启Ram1的写使能Ram1\_WE。

串口读写部分：Addr<=0xBF03

读状态：读使能开启写使能关闭即Ram1Ctrl\_RE = ‘1’ and Ram1Ctrl\_WE = ‘0’。此时若为时钟高电平则读开启Ram1的总使能Ram1\_EN，若为低电平则开启串口的读使能Ram1Ctrl\_rdn。

写状态：读使能关闭写使能开启即Ram1Ctrl\_RE = ‘0’ and Ram1Ctrl\_WE = ‘1’。此时若为时钟高电平则开启Ram1的总使能Ram1\_EN，若为低电平则开启串口的写使能Ram1\_Ctrl\_wrn。

VGA与键盘读写部分：

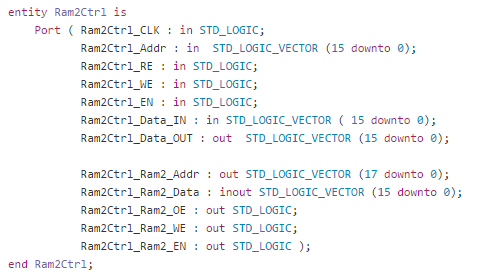
VGA与键盘方面完全组合成了Term单元，仿照串口工作，因此将所有的与串口的通信复制一份与Term单元即可。

数据写入方面，为了避免可能的冲突，统一在时钟下降沿写入。读状态时将Ram1\_Data 设置为高阻态，写状态时将Ram1\_Data设置为输入的Data\_IN。

3）Ram2Ctrl 单元设计

Ram1与Ram2的地址段划分不同，Ram2的地址范围为： 0x0000~0xBEFF

Ram2Ctrl单元的相关接口设计如下：



其中，Ram2Ctrl\_EN, Ram2Ctrl\_RE, Ram2Ctrl\_WE分别为总使能，读使能，写使能。Ram2Ctrl\_Data\_IN为写入时需写入的数据，Ram2Ctrl\_Data\_OUT为读取时读到的数据。至于下方5个接口则是直接通过总线与Ram2单元交互的接口。之所以单独设计Ram2Ctrl单元而不是将其功能综合进AddrMem单元，是由于AddrMem与DataMem可能会同时读写Ram2单元，这其中，Data是inout类型的接口，对其作选择器较为困难，故将Ram的读写单元单独设计。

关于内存部分的读写时序如下图：

读时序：



写时序：



考虑到流水线运行的效率，我们摒弃了在之前做的内存储器实验中采用的状态机转移方法来控制读写逻辑，而是采用了高低电平判断的方式来使得一个周期内可以包含两种状态。每个时钟周期默认所有使能为关。

读状态：读使能开启写使能关闭即Ram2Ctrl\_RE = ‘1’ and Ram2Ctrl\_WE = ‘0’。此时若为时钟高电平则读入地址Addr并开启Ram2的总使能Ram2\_EN，若为低电平则开启Ram2的读使能Ram2\_OE。

写状态：读使能关闭写使能开启即Ram2Ctrl\_RE = ‘0’ and Ram2Ctrl\_WE = ‘1’。此时若为时钟高电平则读入地址Addr并开启Ram2的总使能Ram2\_EN，若为低电平则开启Ram2的写使能Ram2\_WE。

数据写入方面，为了避免可能的冲突，统一在时钟下降沿写入。读状态时将Ram2\_Data 设置为高阻态，写状态时将Ram2\_Data设置为输入的Data\_IN。

## 访存冲突处理

在设计中，读取指令的指令存储器控制单元AddrMem与数据存储器控制单元DataMem同时操作Ram2单元，于是这里便存在了冲突。我们采用了插气泡让其暂停一个周期的方法来解决该问题。

DataMem中存在有一输出接口DataMem\_Stop\_EN，使用组合逻辑来判断，每当DataMem的输入地址Addr经过判断为Ram2的Addr时，DataMem\_Stop\_EN输出为’1’，以表示此时为DataMem对Ram2访问优先，需要暂停AddrMem对Ram2的操作，同时暂停一个周期。否则DataMem\_Stop\_EN = ‘0’。

冲突的控制选择，我们采用了二选一选择器来进行选择，控制信号为DataMem\_Stop\_EN，当DataMem\_Stop\_EN = ‘1’时，选择来自DataMem的数据与控制信号连接到Ram2Ctrl的对应接口上，当DataMem\_Stop\_EN = ‘0’时，选择来自AddrMem的数据与控制信号连接到Ram2Ctrl的对应接口上。

# 十、外接设备

## VGA端口模块

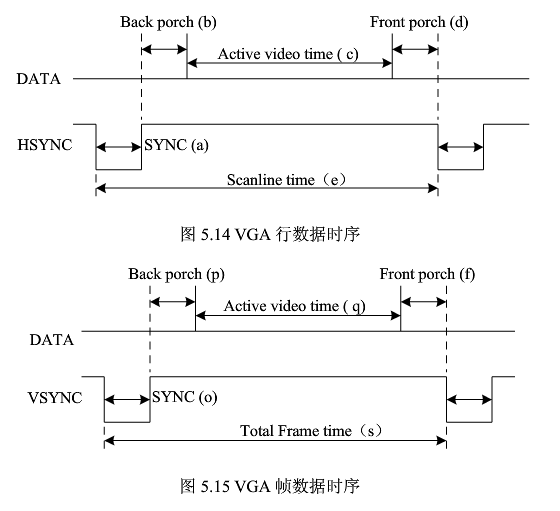
VGA模块主要由两部分组成，VGA时序逻辑部分与VGA显示控制部分。VGA时序逻辑部分主要负责将按照给定的逻辑刷新VGA屏幕，VGA显示控制部分负责决定每个像素点显示什么颜色的数值。

1. VGA时序逻辑

VGA时序如图所示。它分为行数据时序和帧数据时序。

行数据时序，顾名思义，也就是显示一行数据的时序。从图1中可以看出，显示一行数据需要处理两件事情。第一：产生行同步HSYNC。不难看出，HSYNC是一个脉冲信号，次信号的周期为： e = a+b+c+d，低电平时间为a。其中a、b、c、d均为时间信号，这些信号根据需要显示的分辨率不同而不同。第二：产生显示的数据（DATA）号，此信号Wie模拟信号，当在显示有效数据（Active video）内，DATA信号为0~0.714Vpp的模拟电压（R、G、B），根据分辨率的不同，DATA的采样率、点数也皆不相同。

帧数据时序与行时序类似，也就是显示一屏数据的时序。只是这里的基本单位为每行数据，二行数据里面的最基本单位为每个点。



1. VGA显示控制

VGA显示控制部分决定的是每一个像素应当显示的内容。

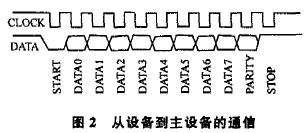
为了显示字符，我们需要知道每个需要显示的字符其在显示器上的像素数据应当是如何。为了简化数据量，我们采用了黑白点阵而不是彩色点阵来显示数据。由此我们需要有一份点阵字符表。这份字符表可见文件中的ASCII.vhd（见附表）。文件中具体内容为存储了一个constant数组，数组的内容为256个字符的点阵数据，每个字符占用8\*16个像素。

我们采用的VGA的显示分辨率为640\*480。按照每个字符占用的像素空间为8\*16计算，总计可以显示80\*30共2400个字符。由于VGA的显示设置，需要在每次刷新屏幕时给出对应像素点的RGB值，即其并没有暂存功能。由于这个原因，我们需要构建一个存储单元用于存储这其中每个位置的字符。考虑到每个字符只需要存储其ASCII码，则对每个字符存储大小为8位，总存储空间为2400\*8 = 19600 。在这里我们采用MemBlock的IP核来存储这部分数据。

对于数据的写入并显示于VGA，我们采用的方法是每次有新数据来临时，将该ASCII码存储进当前存储表的下一个位置。至于具体字符的显示，则是每次在屏幕上显示时从存储表中读取到当前位置的字符，之后按图索骥再从ASCII码点阵表上获取该处的像素的值，并将该值刷新到屏幕上对应的像素点的位置。

## PS2键盘模块

 当从设备向主设备发送数据时,首先检查时钟线,以确认时钟线是否为高电平.如果是高电平,从设备就可以开始传输数据;反之,从设备要等待获得总线的控制权,才能开始传输数据.传输的每一帧由11位组成,发送时序及每一位的含义如图2所示：



每一帧数据中开始位总是为0,数据校验采用奇校验方式,停止位始终为1.从设备到主设备通信时,从设备总是在时钟线为高时改变数据线状态,主设备在时钟下降沿读人数据线状态。

然而获取到了数据后我们并不能直接得到传输过来的数据内容是什么，是因为PS2键盘采用的编码与命令集。

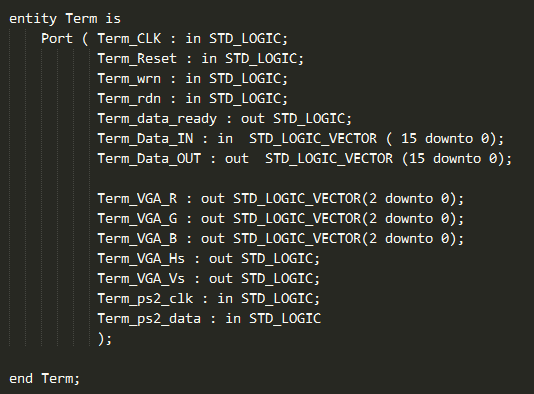
目前,PC机使用的PS/2键盘都默认采用第2套扫描码集.扫描码有两种不同的类型:“通码(make code)”和“断码(break code)”.当一个键被按下或持续按住时,键盘会将该键的通码发送给主机;而当一个键被释放时,键盘会将该键的断码发送给主机.根据键盘按键扫描码的不同,可将按键分为3类：  
第1类按键      通码为一个字节,断码为0xF0+通码形式.如A键,其通码为0x1C;断码为0xF0 0x1C.  
第2类按键      通码为两字节0xE0+0xXX形式,断码为0xE0+0xF0+0xXX形式.如Right Ctrl键,其通码为0xE0 0x14;断码为0xE0 0xF0 0x14.  
第3类特殊按键  有两个,Print Screen键,其通码为0xE0 0x12 0xE0 0x7C;断码为0xE0 0xF0 0x7C 0xE0 0xF0 0x12.Pause键,其通码为0xE1 0x14 0x77 0xE1 0xF0 0xl4 0xF0 0x77;断码为空.  
     组合按键扫描码的发送是按照按键发生的次序,如按下面顺序按左Shift十A键:① 按下左Shift键;② 按下A键;③ 释放A键;④ 释放左Shift键,那么计算机上接收到的一串数据为0x12 0x1C 0xF0 0x1C 0xF0 0x12.

在我们的设计之中，我们考虑到Term只需要使用到ASCII码表中规定的字符，所以忽略了其他的特殊案件。这样设计的好处有利于数据的传输与转化，可以方便地将数据转换成其对应的ASCII码传输给需要输出的模块。

## Term组合模块

Term模块主要仿照串口的设计，内部封装了以上的VGA模块以及PS2模块，实现的功能是代替串口部分实现脱离计算机只使用VGA与PS2键盘来做到数据的输入以及输出。

其单元接口如下：



其中，Term\_CLK与Term\_Reset不做过多解释。Term\_wrn，Term\_rdn，Term\_data\_ready这三个接口皆为仿照串口部分，实现的功能为写使能，读使能，数据准备信号。具体功能及使用方法详见本文档的串口部分，内容基本一致。Data\_IN与Data\_OUT负责VGA输出的数据写入以及PS2键盘读到的数据输出。下方的7个接口主要负责VGA与PS2键盘与管脚的交互，这里略过不表。

## Flash模块

Flash模块与CPU独立，当FPGA烧入时，需要保证CPU的使能是0，Flash模块实现从Flash读入并且将读入的内容输出到Ram2。

使用状态机实现：  
状态0：初始化，转1

状态1：FlashWE置0，转2

状态2：数据线置00FF，WE置1，转3

状态3：OE和CE置1，转4

状态4：设置ADDR为Iter，数据线置高阻，转5

状态5：读入数据，OE和CE置0，转6

状态6：Ram\_RE和Ram\_WE置1，Ram\_En置0，转7

状态7：Ram\_Data置为从Flash读入的数据，转8

状态8：拉低Ram\_WE，转9

状态9：拉高Ram\_WE，转10

状态10：如果Iter未达到监控程序的长度，那么将Iter+1，转0；否则转10

需要注意，Flash操作时，使用的时钟频率需要更低一些，否则会发现Ram2中有一部分是空，另一部分是正确的值。