Wishbone 内存/串口/总线实验

2022年秋

内容概要

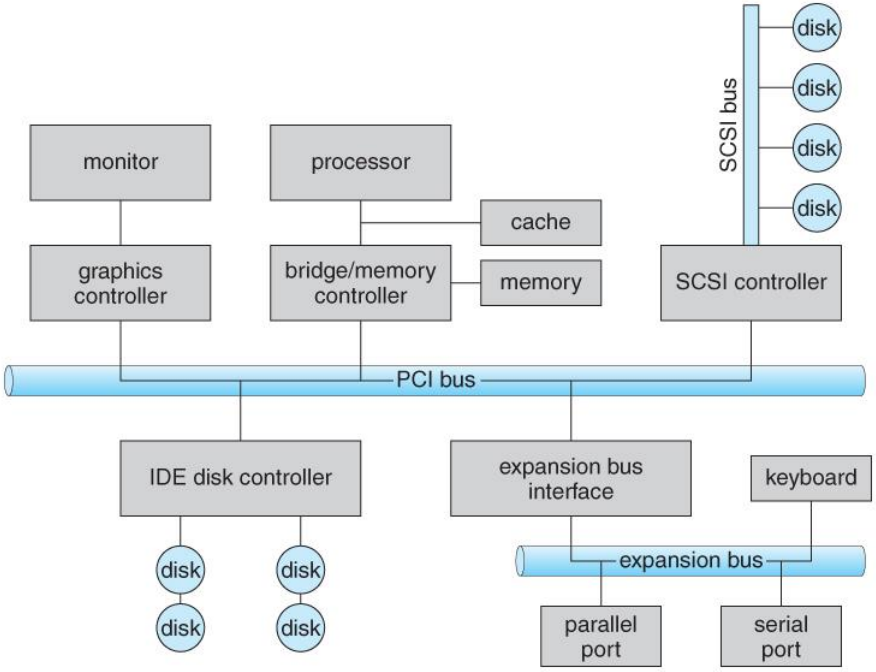
- 为什么需要总线协议
- 设计一个总线协议
- Wishbone 总线
- 内存实验
- 串口实验

为什么需要总线协议

- CPU 是如何操作外设的？
  - USB：键盘，鼠标，U盘
  - PCIe：显卡，网卡，硬盘
- 设备种类很多，但是协议却很少
- 好处：统一协议和接口，CPU 和外设可以独立更新
  - 比如：插 U 盘不需要换 CPU，换 CPU 不需要换显卡
- 把 USB、PCIe 等都称为总线协议
- CPU 内部也有总线！

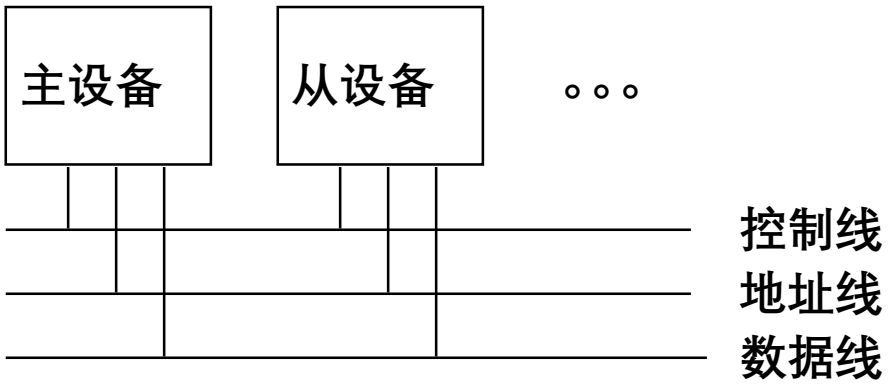
总线 Bus

- CPU与外部硬件间的桥梁
  - 访问内存数据
  - 通过外设进行输入/输出
- 为什么总线叫做 Bus？
  - 总线就像是巴士，在CPU 和外设之间来回“搬运”数据
  - 让 CPU 使用同样的接口访问各类设备



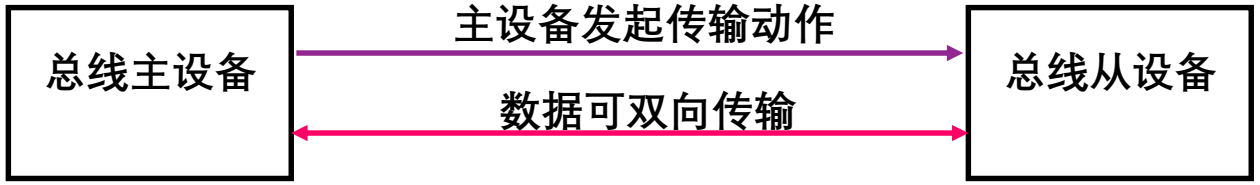
总线 Bus

- 主设备：有能力控制总线，发起总线事务
- 从设备：响应主设备请求
- 协议：定义总线传输中的事件顺序和时序要求
- 异步： 控制信号（请求，应答）作为总控信号
- 同步：使用共同的时钟信号



总线 Bus

- 总线事务包括两个部分：
  - 发起命令 （和地址）
  - 传输数据
- 主设备是总线事务的发起者：
  - 发出命令（和地址）
- 从设备是总线事务的响应者：
  - 若主设备发出的是读命令，则将数据发送到主设备
  - 否则，接收主设备发来的写入数据



总线协议需要什么

- CPU 是如何读写内存的？如下面的 C 代码：

```c
int *ptr;
int read_data = *ptr; // read
*ptr = write_data; // write
```

- 编译到 RISC-V 指令：

```
# ptr: a0
# read_data: a1
# write_data: a2

# int read_data = *ptr;
lw a1, 0(a0)
# *ptr = write_data;
sw a2, 0(a0)
```

总线协议需要什么

- 执行 lw 指令，会发生什么：
  - CPU 向内存发送**地址 a0**，告诉内存要进行**读**操作
  - 内存读取完成，向 CPU 回复**地址 a0 的数据**

```
# int read_data = *ptr;
lw a1, 0(a0)
```

- 执行 sw 命令，会发送什么：
  - CPU 向内存发送**地址 a0** 和**数据 a2**，告诉内存要进行**写**操作
  - 内存写入完成，向 CPU 回复**写入完毕**

```
# *ptr = write_data;
sw a2, 0(a0)
```

总线协议需要什么

- CPU 需要向内存传输的信息：
  - 读取或者写入的地址，记为 addr
  - 如果是写入操作，要附带写入的数据，记为 w_data
  - 本次操作是读还是写，记为 we，write enable
- 内存需要向 CPU 回复的信息：
  - 如果是读取操作，要附带读取的结果，记为 r_data
- 这样就足够了吗？

总线协议需要什么

- CPU 并不是一直在访问内存的
  - CPU 闲置的时候，让内存休息，减少能耗
  - 添加一个 CPU 向内存传输的信息：让 CPU 告诉内存，什么时候去进行读写操作
  - 记为 valid，即是一个合法操作
- 内存相比 CPU 是非常慢的
  - 内存读取要几十到一百多 ns，而 CPU 一个时钟周期只要零点几 ns
  - 相比 CPU，内存需要很长时间来完成一次读写操作
  - 添加一个内存向 CPU 回复的信息：内存是否完成了当前的读写操作
  - 记为 ready，即内存已经准备好了要读取的数据，或准备好了写入，也就是操作完成

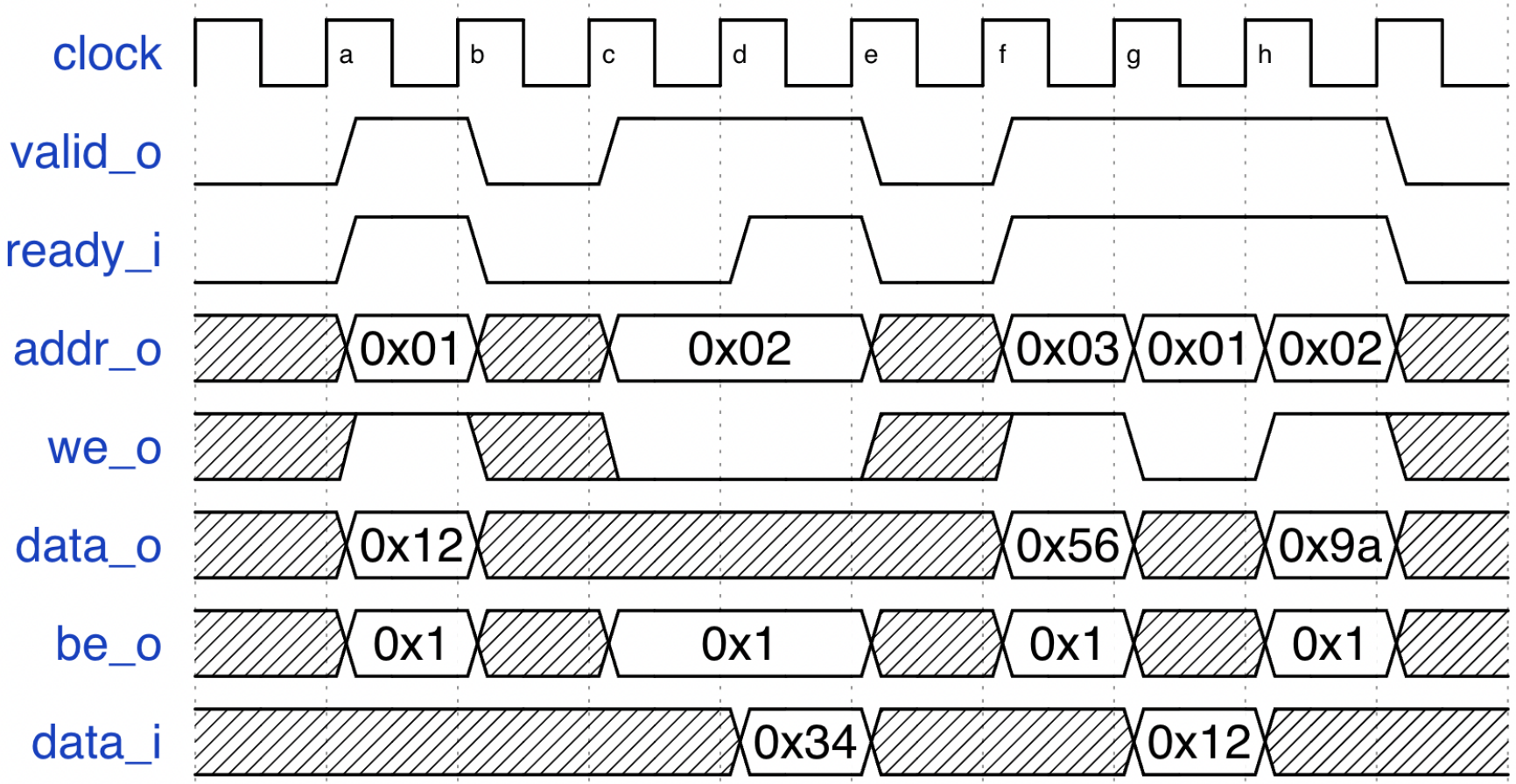
总线协议需要什么

- 总结一下 CPU 进行读写操作经历的步骤：
- CPU 向内存传输：
  - 读取或者写入的地址（addr）
  - 要写入的数据（w_data）
  - 本次操作是读还是写（we）
  - CPU 要进行一次操作（valid=1）
- 内存观测到 valid=1 后，进行实际的操作
- 内存完成操作后，向 CPU 回复：
  - 读取的数据（r_data）
  - 操作完成（ready=1）
- CPU 观测到 ready=1后，设置 valid=0，表示不继续操作
- 循环这个过程

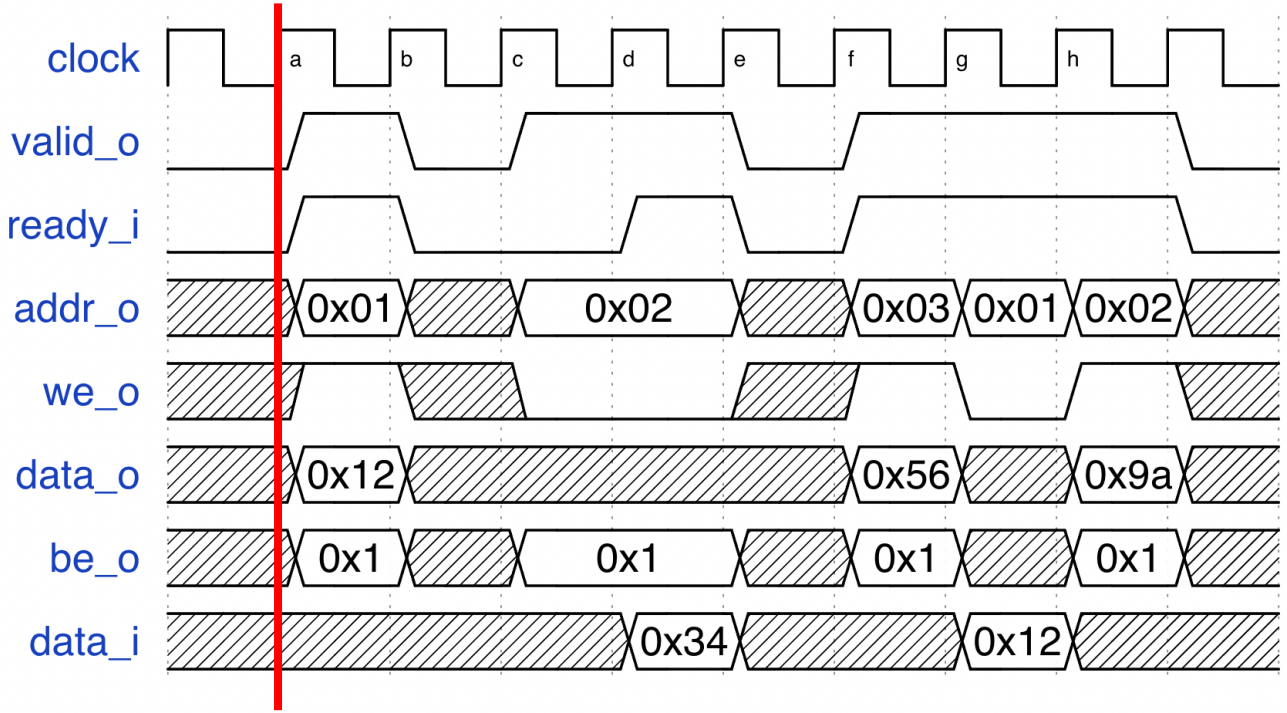
总线协议需要什么

- 那么，怎么在硬件上实现？
- 约定 _i 表示输入，_o 表示输出
- 从 CPU 角度来看，有如下的硬件信号：
  - valid_o：CPU 告诉内存要开始进行操作
  - ready_i：内存告诉 CPU 当前操作已经完成
  - addr_o：CPU 告诉内存要读写的地址
  - we_o：CPU 告诉内存本次操作是读还是写
  - data_o：CPU 告诉内存想要写入的数据
  - be_o：CPU 告诉内存要写入哪几个字节（字节使能）
  - data_i：内存告诉 CPU 读取的数据
- 最后，同步时序电路还需要一个时钟 clock_i

总线协议需要什么

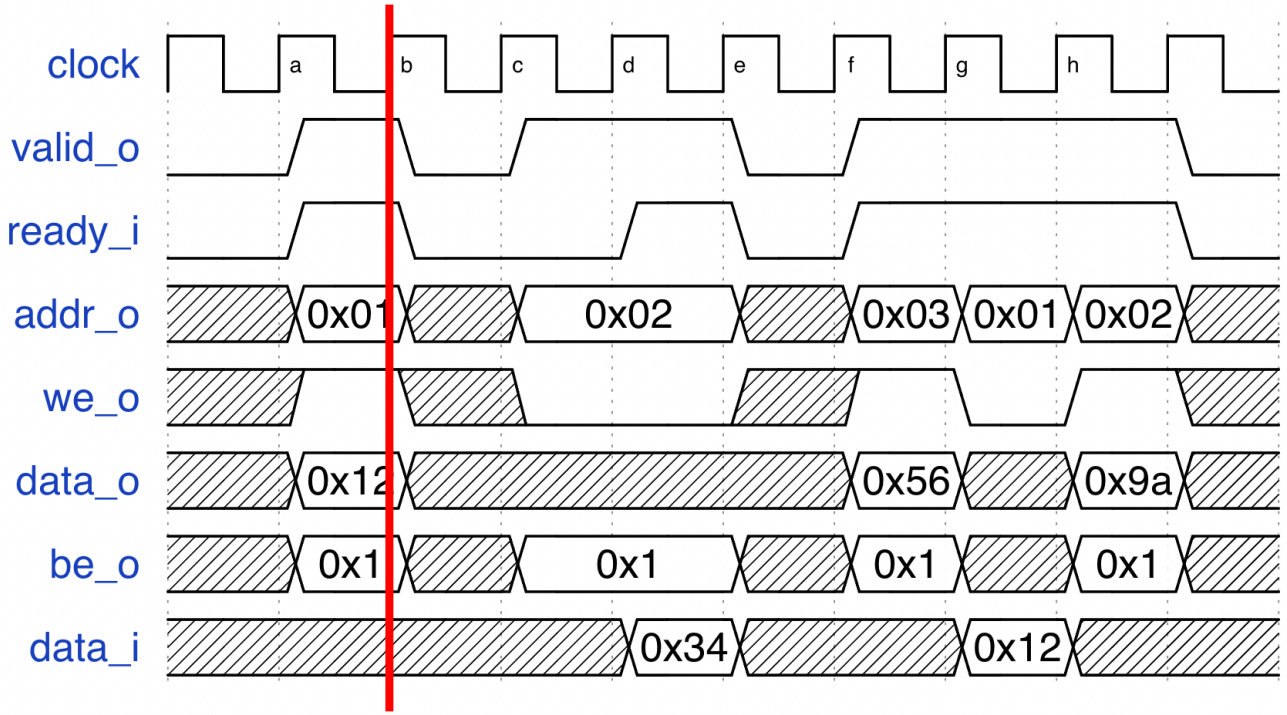

clock
valid_o
ready_i
addr_o
we_o
data_o
be_o
data_i
a
b
c
d
e
f
g
h
0x01
0x02
0x03
0x01
0x02
0x12
0x56
0x9a
0x1
0x1
0x1
0x1
0x34
0x12

总线协议需要什么



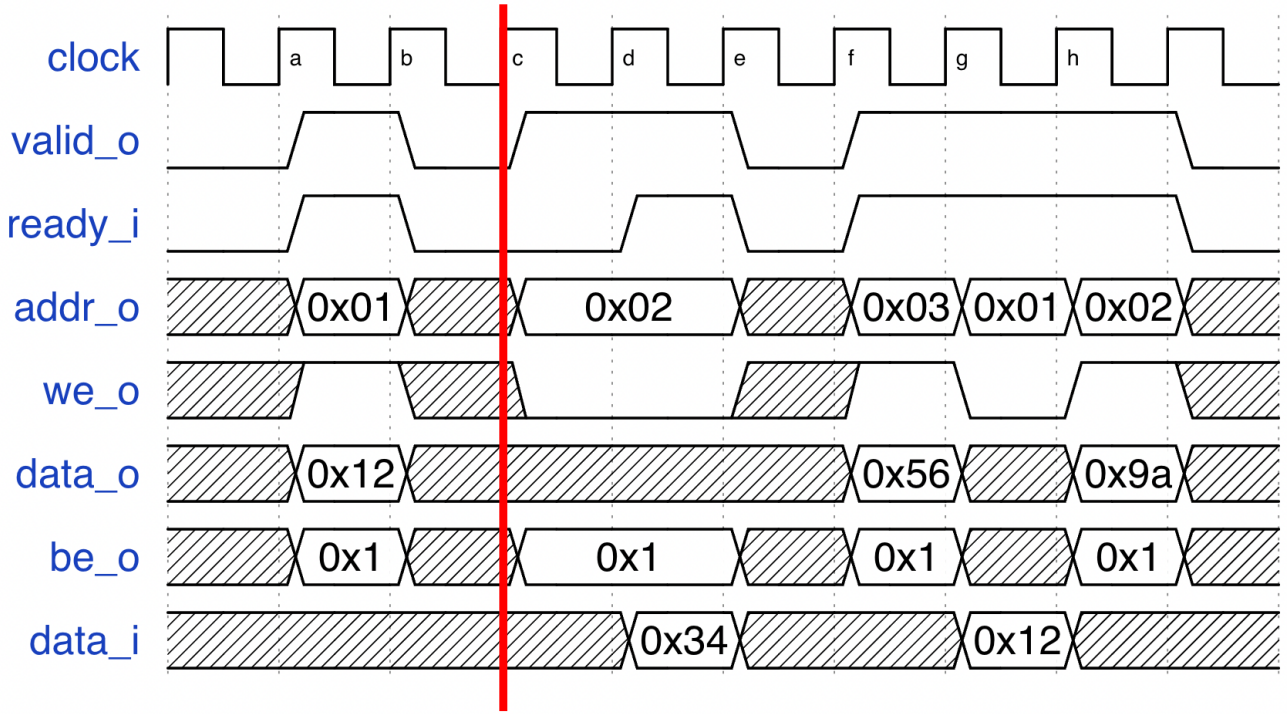
- a 周期：此时 valid_o=1 && ready_i=1 说明有请求发生，此时 we_o=1 说明是一个写操作，并且写入地址是 addr_o=0x01，写入的数据是 data_o=0x12
- 这里假设写操作立即完成

总线协议需要什么



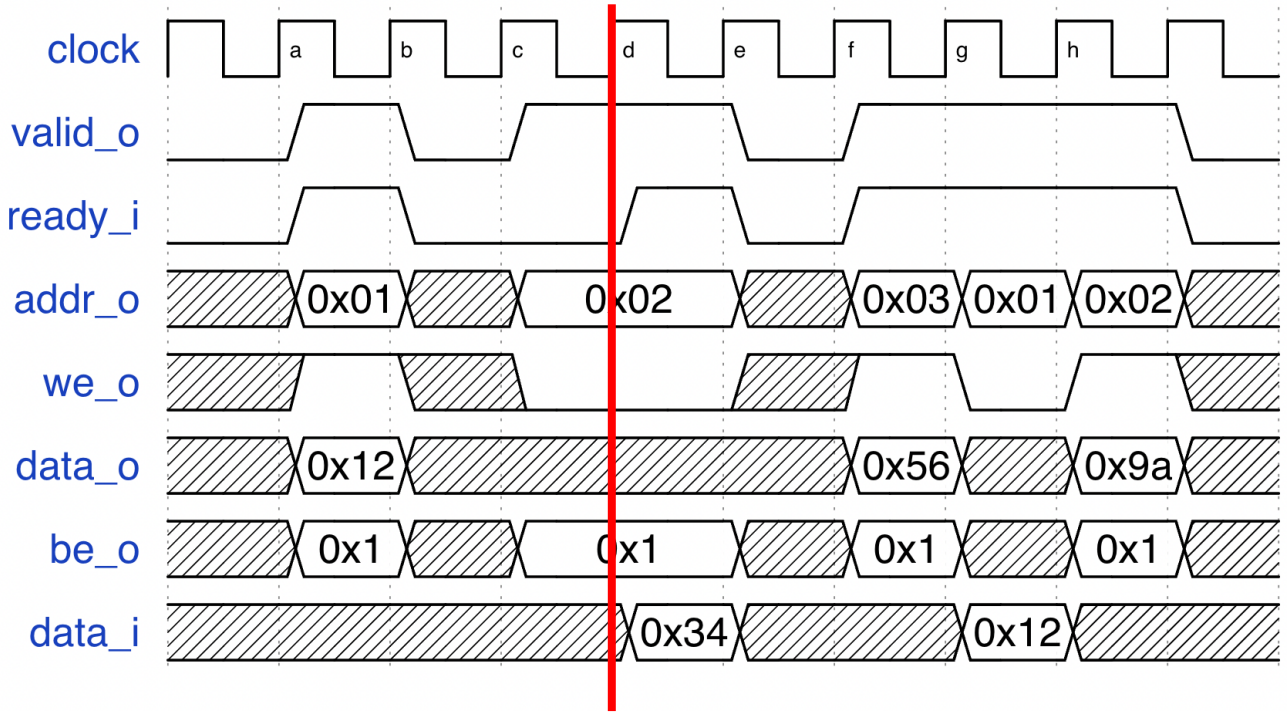
- b 周期：此时 valid_o=0 && ready_i=0 说明无事发生

总线协议需要什么



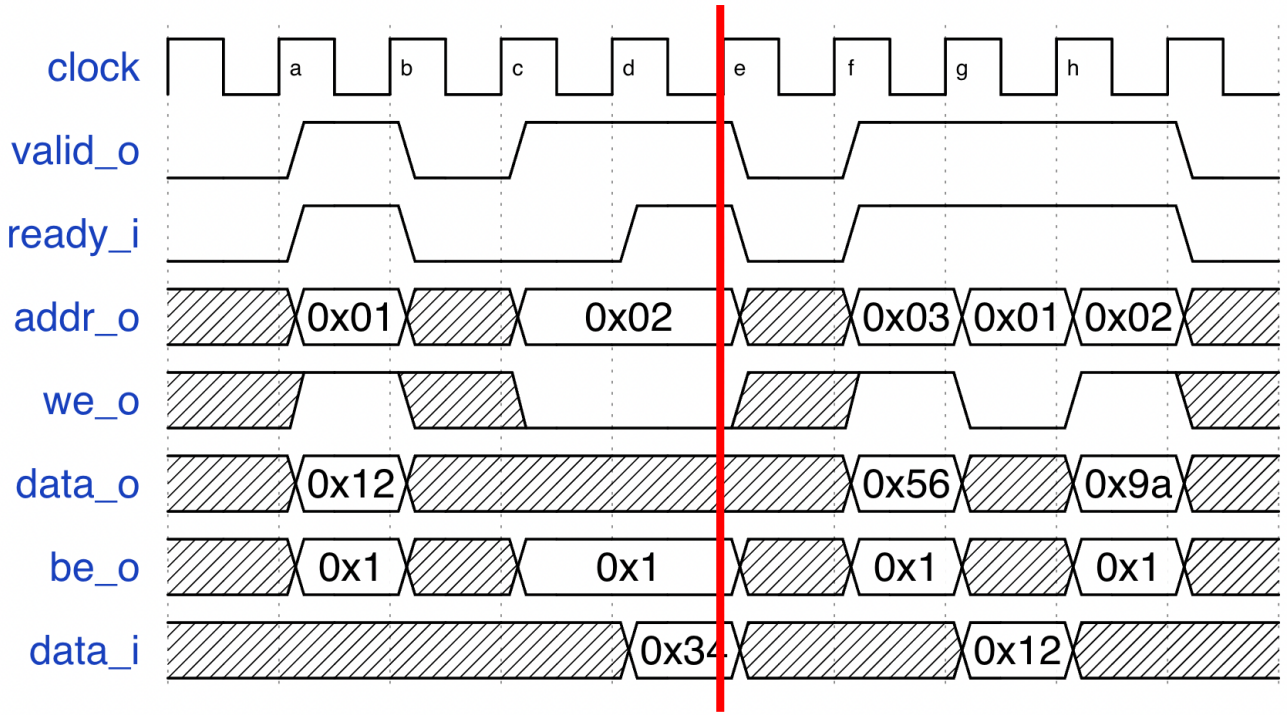
- c 周期：此时 valid_o=1 && ready_i=0 说明 CPU 想要从地址 0x02（addr_o=0x02）读取数据（we_o=0），但是内存没有来得及完成（ready_i=0）

总线协议需要什么



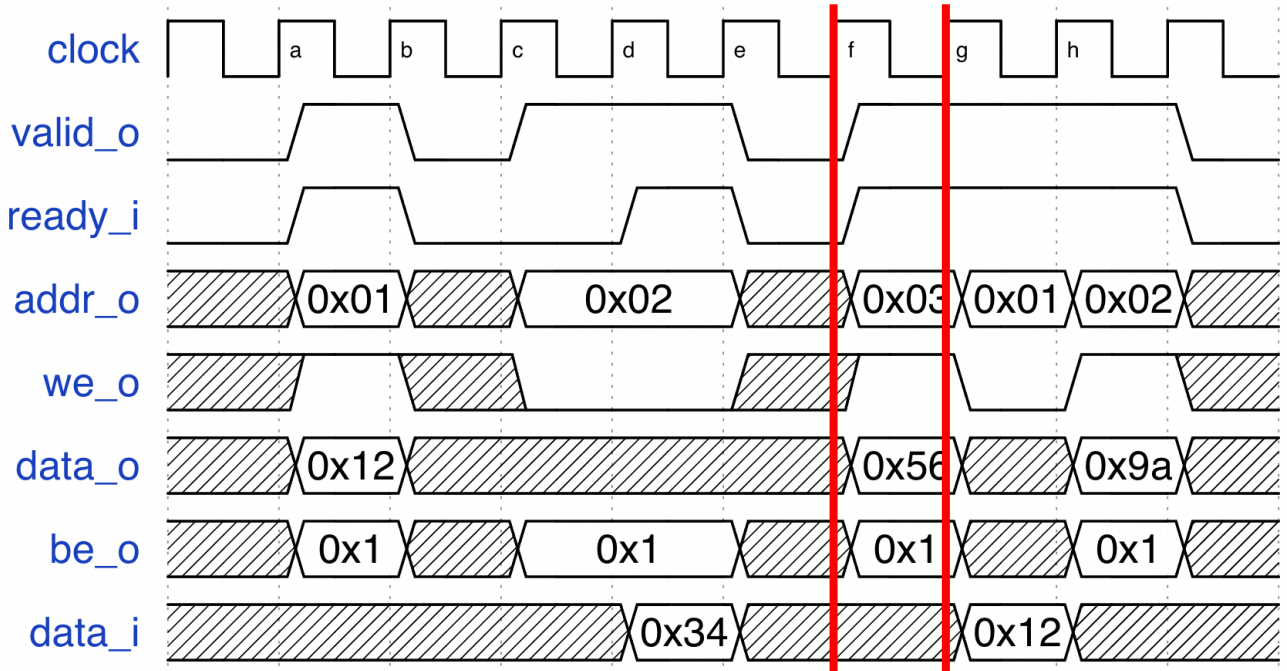
- d 周期：此时 valid_o=1 && ready_i=1 说明有请求发生，master 从地址 0x02（addr_o=0x02）读取数据（we_o=0），读取的数据为 0x34（data_i=0x34）
- 此时一共花了 c 和 d 两个周期来实现一次读操作

总线协议需要什么



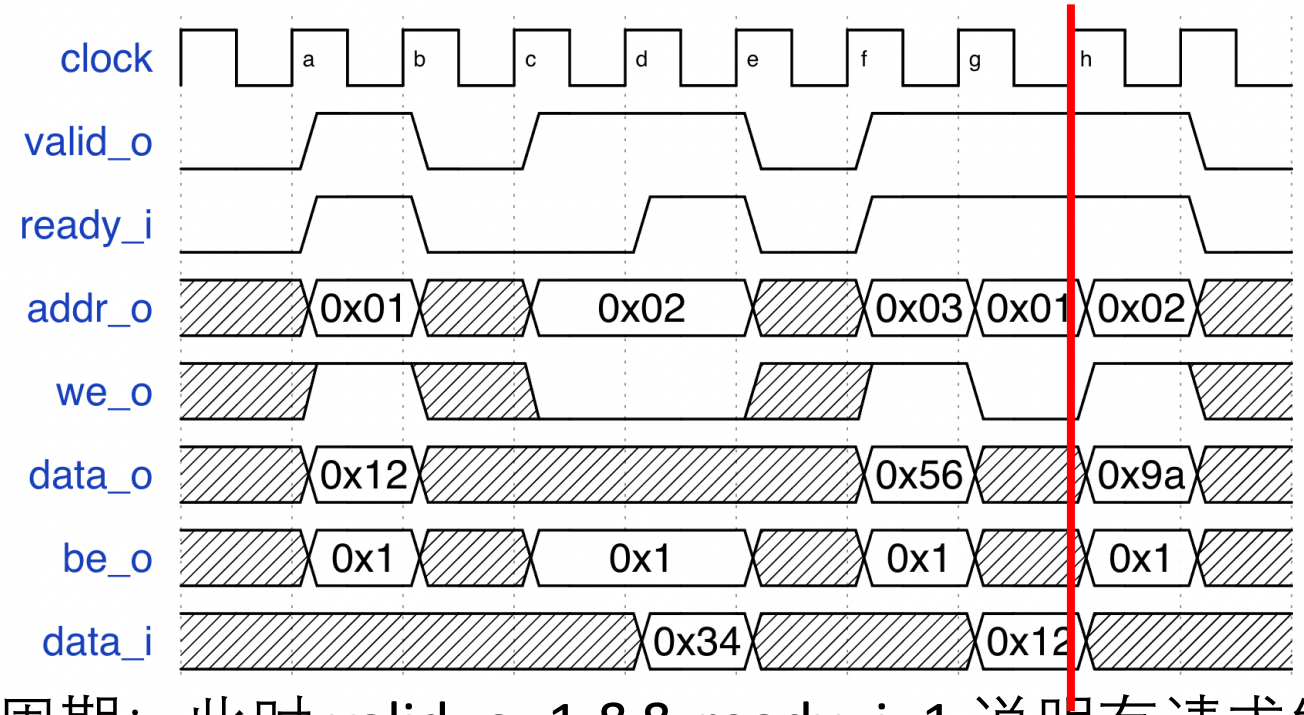
- e 周期：此时 valid_o=0 && ready_i=0 说明无事发生

总线协议需要什么



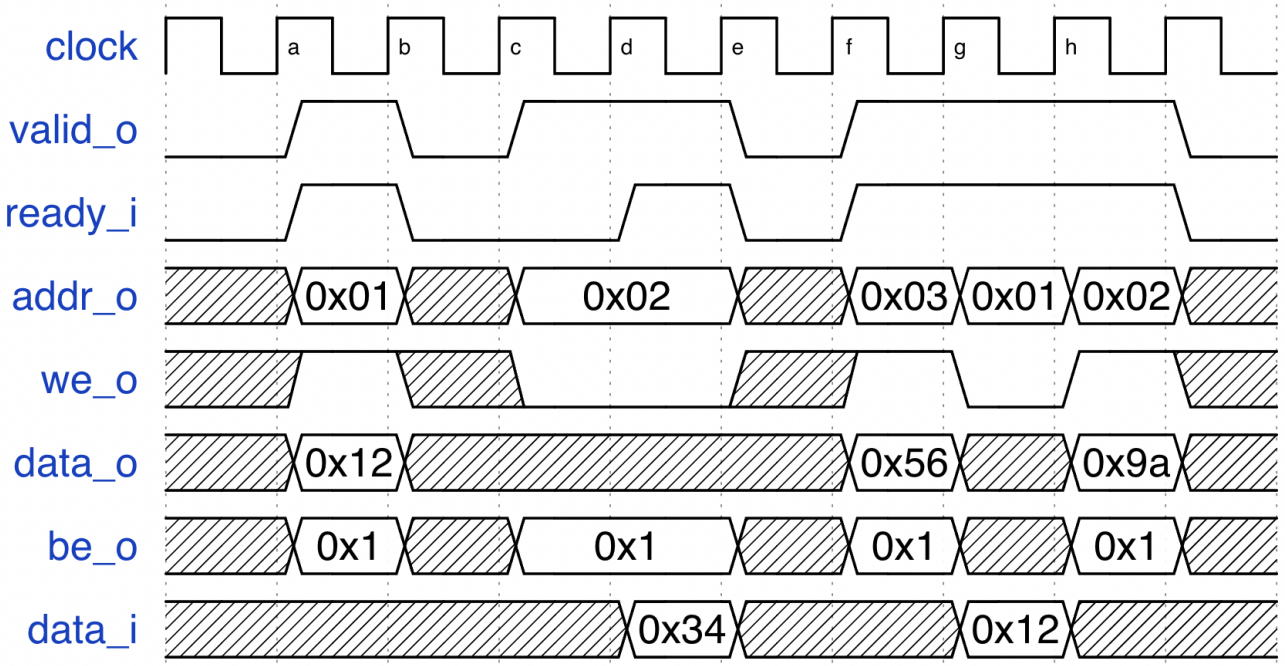
- f 周期：此时 valid_o=1 && ready_i=1 说明有请求发生，CPU 向地址 0x03（addr_o=0x03）写入数据（we_o=1），写入的数据为 0x56（data_i=0x56）
- g 周期：此时 valid_o=1 && ready_i=1 说明有请求发生，CPU 从地址 0x01（addr_o=0x01）读取数据（we_o=0），读取的数据为 0x12（data_i=0x12）

总线协议需要什么



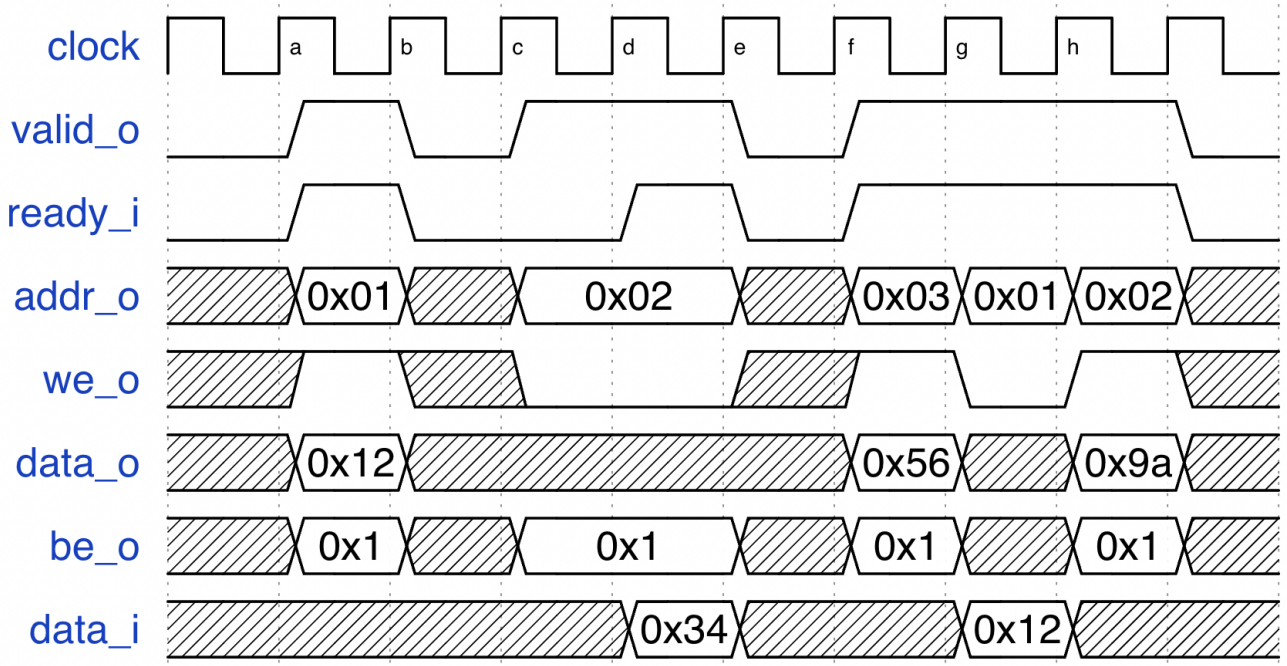
- h 周期：此时 valid_o=1 && ready_i=1 说明有请求发生，master 向地址 0x02（addr_o=0x02）写入数据（we_o=1），写入的数据为 0x9a（data_i=0x9a）
- 如果内存足够快，那么每个周期都可以完成一个读或者写操作

总线协议需要什么



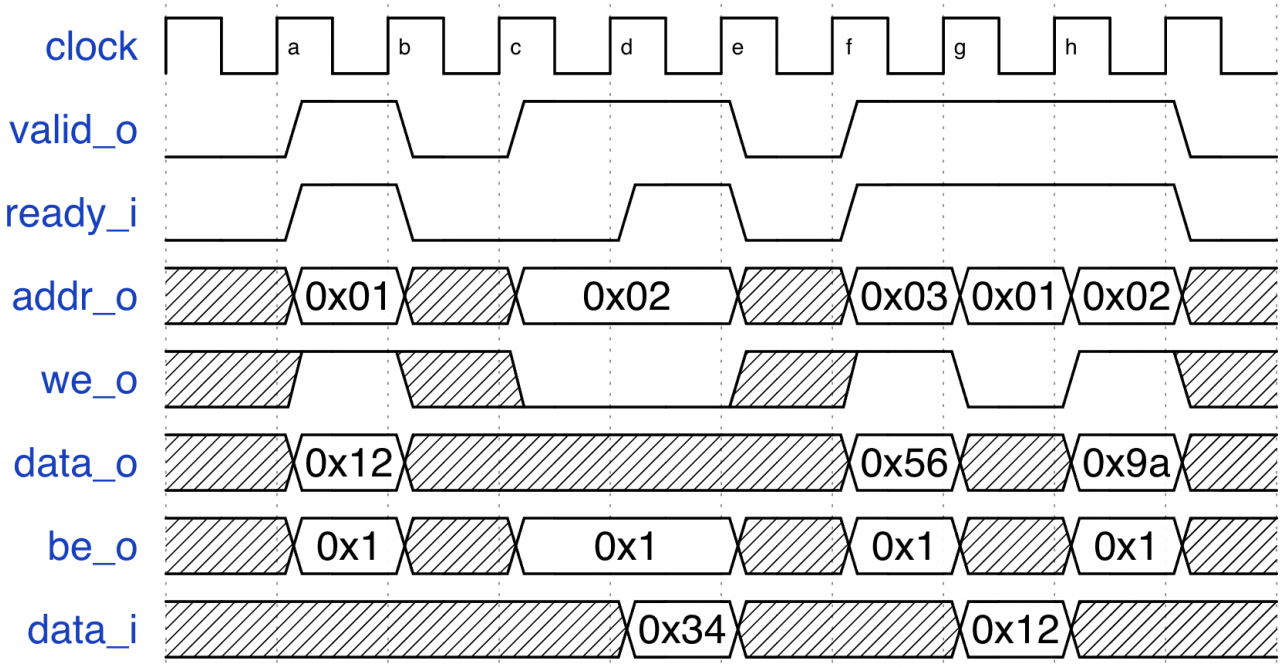
- 观察发现，CPU 想要发起请求的时候，就设置 valid_o=1；当内存可以完成请求的时候，就设置 ready_i=1；在 valid_o=1 && ready_i=1 时请求完成，可以进行下一个请求

总线协议需要什么



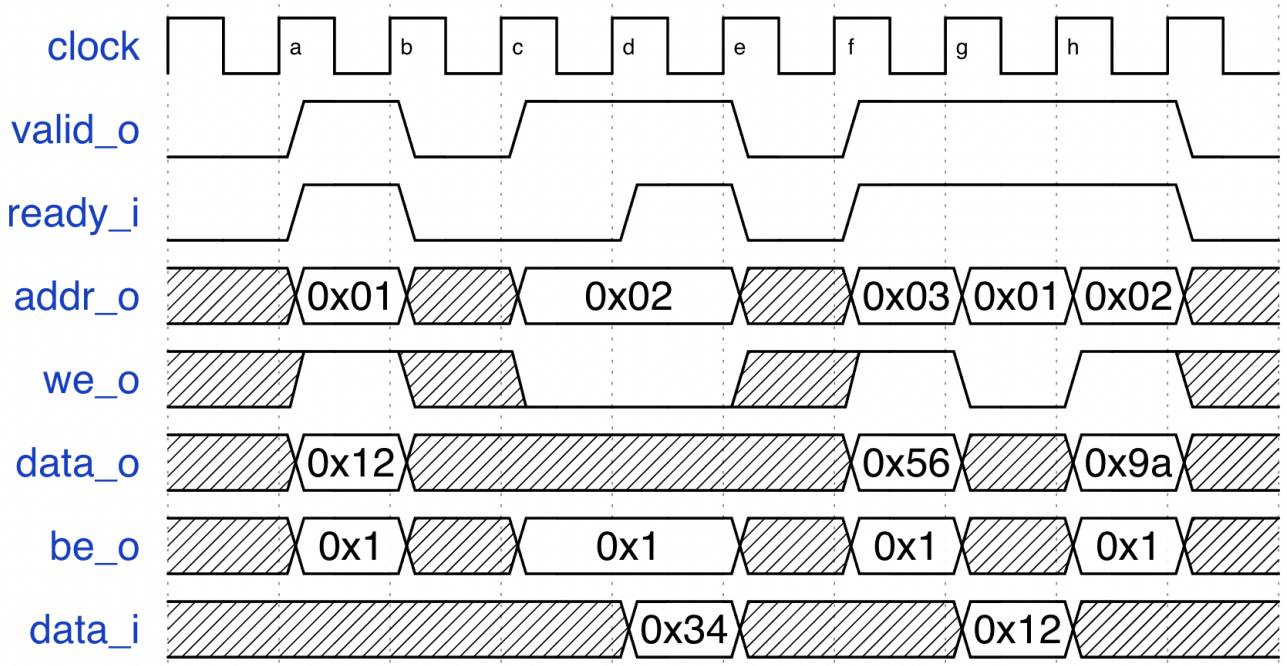
- 继续寻找规律，如果 CPU 发起请求，同时内存不能立即完成请求，即 valid_o=1 && ready_i=0，此时 CPU 要保持 addr_o we_o data_o 和 be_o 不变，直到请求结束
-

总线协议需要什么



- 当 CPU 不发起请求的时候，即 valid_o=0，此时总线上的信号都视为无效数据，不应该进行处理；对于读操作，只有在 valid_o=1 && ready_i=1 时 data_i 上的数据是有效的，其他时候都是无效数据

总线协议需要什么



- 可以连续多个周期发生请求，即 valid_o=1 && ready_i=1 连续多个周期等于一，此时是理想情况，可以达到总线最高的传输速度

总线协议小结

- 简单总结一下总线协议
- 信号：
  - CPU 到内存的信号：地址，写入的数据，读或写，字节使能，是否合法
  - 内存到 CPU 的信号：是否完成，读取的数据
- CPU 想要进行读写操作：valid_o=1
- 内存完成了 CPU 的操作：valid_o=1 && ready_i=1
  - 同时读取的数据可以在总线上得到

总线协议的用途

- 总线协议不仅可以用来访问内存，也可以用来访问外设！
- 例如网卡：
  - 规定：
  - CPU 从 0x12 地址开始读取，可以读取到网卡接收到的数据
  - CPU 向 0x34 地址逐个字节写入的数据，会通过网卡发送出去
  - 网卡并不是内存，但是通过人为规定地址背后的行为，可以让 CPU 以同样的方式操作外设
- 例如声卡：
  - 规定 CPU 向 0x56 地址写入音频文件，就会播放声音

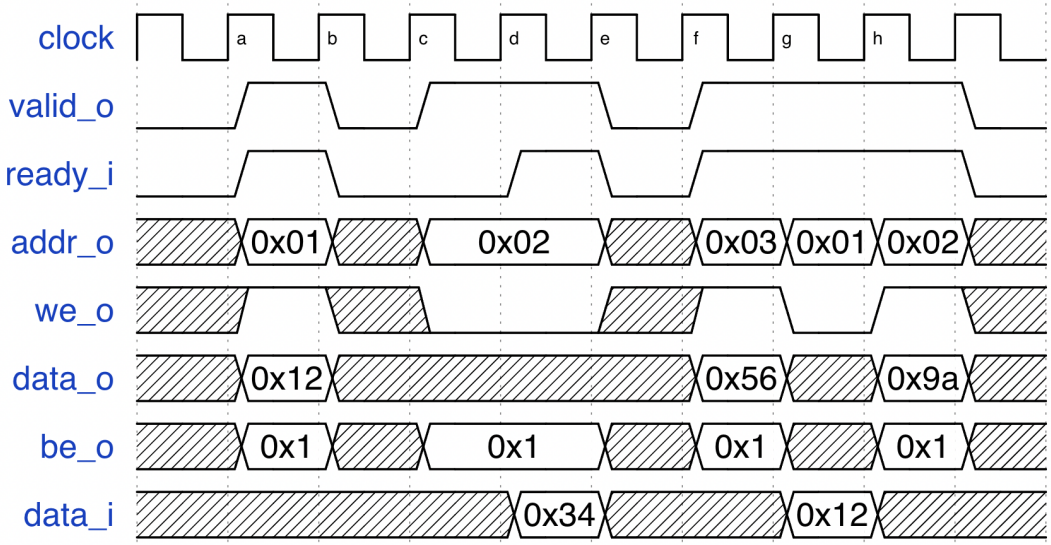
总线协议

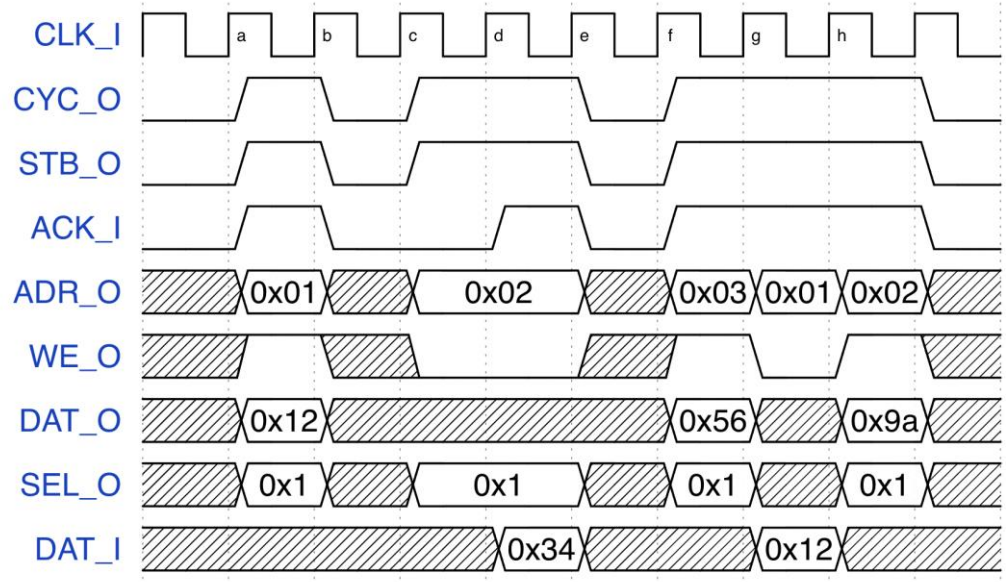
- 既然总线上被访问的可以是内存，也可以是外设。访问内存的也不一定是 CPU，可能是集成显卡。
- 把总线的概念推广：
  - Master：发送请求的一方，如 CPU，集成显卡
  - Slave：接收和处理请求的一方，如内存，外设
- 下面都用 Master 和 Slave 来表示总线上的发送方和接收方：
  - Master 设置 valid_o=1，等待 Slave 设置 ready_i=1
  - Slave 完成操作后，设置 ready_i=1
  - valid_o=1 && ready_i=1 表示完成了一次请求

Wishbone 总线协议

- 总线协议是有各种标准的
  - 例如：USB、PCIe
- 实践中常用的总线协议：Wishbone
- 与刚才我们从 CPU 访问内存需要的信息中推导出来的总线十分类似，只不过换了一个名字：
  - CLK_I: 时钟输入，即自研总线中的 clock_i
  - STB_O：高表示 master 要发送请求，即自研总线中的 valid_o
  - ACK_I：高表示 slave 完成请求，即自研总线中的 ready_i
  - ADR_O：master 想要读写的地址，即自研总线中的 addr_o
  - WE_O：master 想要读还是写，即自研总线中的 we_o
  - DAT_O：master 想要写入的数据，即自研总线中的 data_o
  - SEL_O：master 读写的字节使能，即自研总线中的 be_o
  - DAT_I：master 从 slave 读取的数据，即自研总线中的 data_i
  - CYC_O：总线的使能信号，无对应的自研总线信号

Wishbone 总线协议





小结

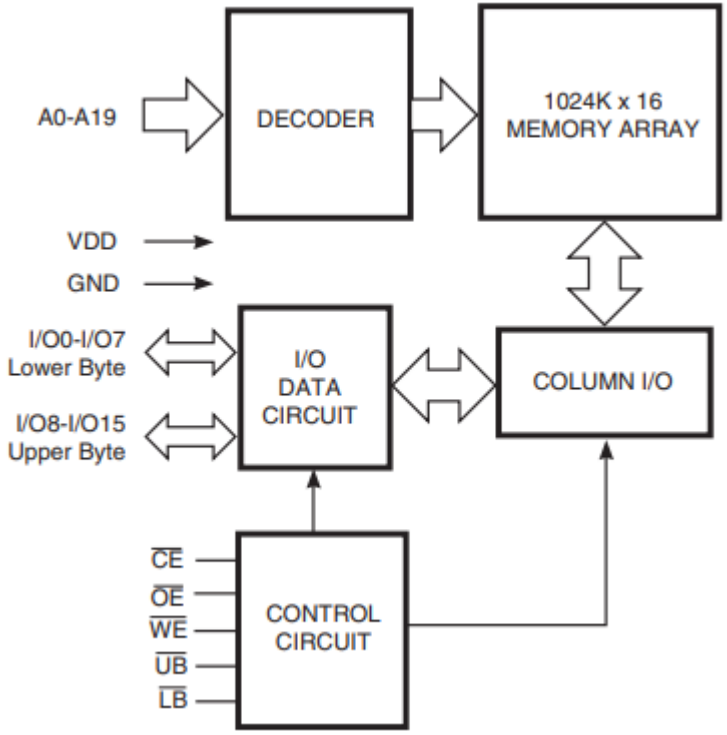
- 为什么需要总线协议
- 设计一个总线协议
- Wishbone 总线

内存实验

- 使用教学计算机上的 FPGA 芯片，设计一个 Wishbone 协议的 SRAM 控制器，完成对存储器 SRAM 的访问。
- 对着文档和提供的框架代码进行编写，你只需要编写 sram_controller 模块，无需修改 lab4_top 或 sram_tester，但强烈建议阅读它们，以便熟悉本实验的整体结构。
- 实现驱动SRAM的wishbone_slave

内存实验

- SRAM静态随机存取存储器（Static Random Access Memory）
- IS61WV102416



内存实验

- 想象成一个数组：uint32_t sram[1048576]
- 1M 个 32 位整数，一共是4 MB 的数据

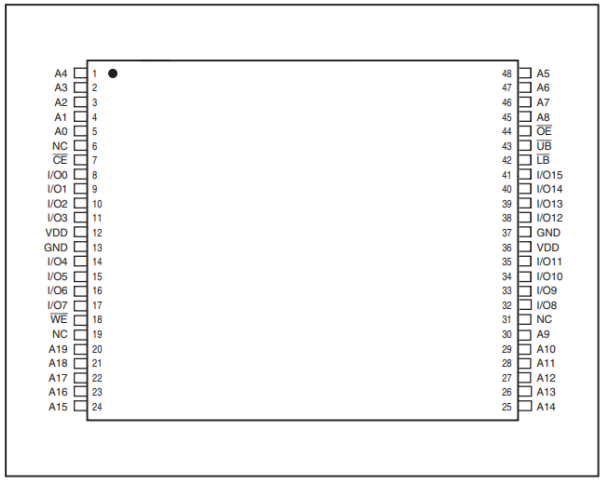
```
uint32_t sram[1048576];
// read
uint32_t read_data = sram[addr];
// write
sram[addr] = write_data;
```

内存实验

PIN DESCRIPTIONS

A0-A19	Address Inputs
I/O0-I/O15	Data Inputs/Outputs
$\overline{CE}$	Chip Enable Input
$\overline{OE}$	Output Enable Input
$\overline{WE}$	Write Enable Input
$\overline{LB}$	Lower-byte Control (I/O0-I/O7)
$\overline{UB}$	Upper-byte Control (I/O8-I/O15)
NC	No Connection
V_{DD}	Power
GND	Ground

48-pin TSOP-I (12mm x 20mm)



TRUTH TABLE

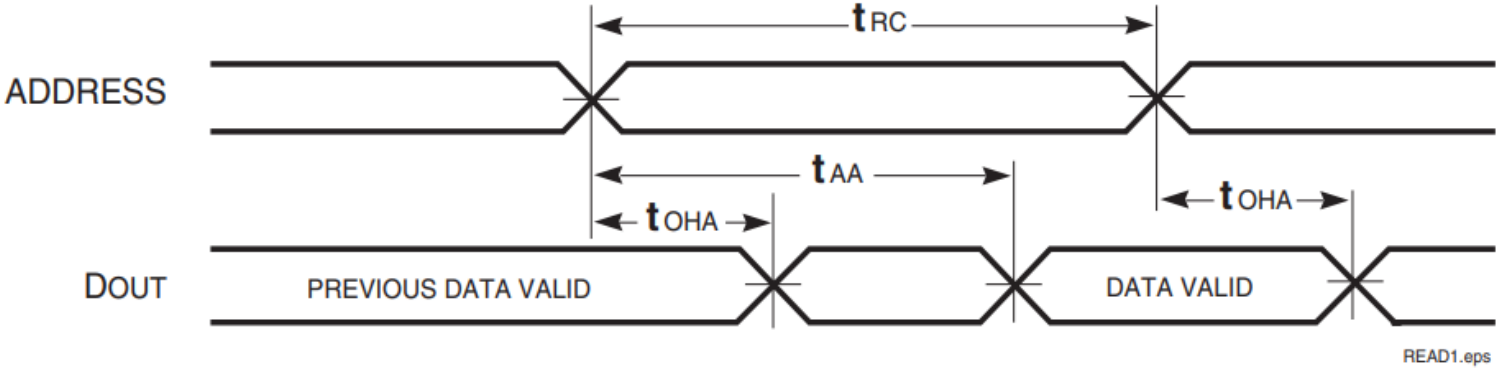
Mode	$\overline{WE}$	$\overline{CE}$	$\overline{OE}$	$\overline{LB}$	$\overline{UB}$	I/O PIN I/O0-I/O7	I/O PIN I/O8-I/O15	V_{DD} Current
Not Selected	X	H	X	X	X	High-Z	High-Z	I_{SB1}, I_{SB2}
Output Disabled	H	L	H	X	X	High-Z	High-Z	I_{CC}
	X	L	X	H	H	High-Z	High-Z	
Read	H	L	L	L	H	D_{OUT}	High-Z	I_{CC}
	H	L	L	H	L	High-Z	D_{OUT}	
	H	L	L	L	L	D_{OUT}	D_{OUT}	
Write	L	L	X	L	H	D_{IN}	High-Z	I_{CC}
	L	L	X	H	L	High-Z	D_{IN}	
	L	L	X	L	L	D_{IN}	D_{IN}	

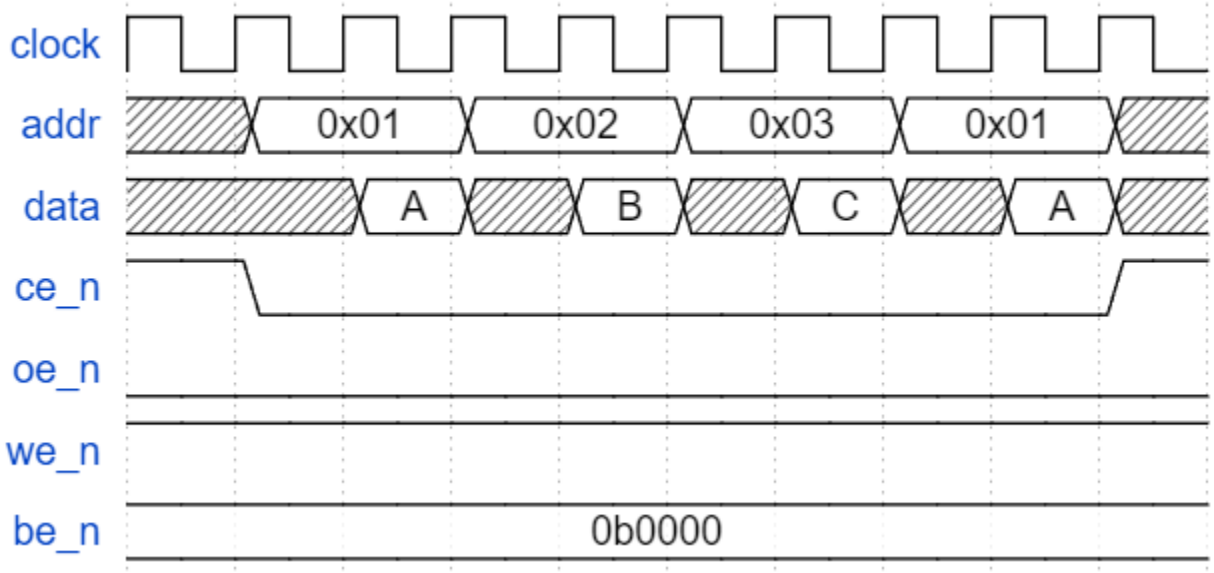
内存实验

```
//BaseRAM信号
inout wire[31:0] base_ram_data,   //BaseRAM数据
output wire[19:0] base_ram_addr, //BaseRAM地址
output wire[3:0] base_ram_be_n,   //BaseRAM字节使能，低有效。如果不使用字节使能，请保持为0
output wire base_ram_ce_n,        //BaseRAM片选，低有效
output wire base_ram_oe_n,        //BaseRAM读使能，低有效
output wire base_ram_we_n,        //BaseRAM写使能，低有效
```

内存实验

读时序

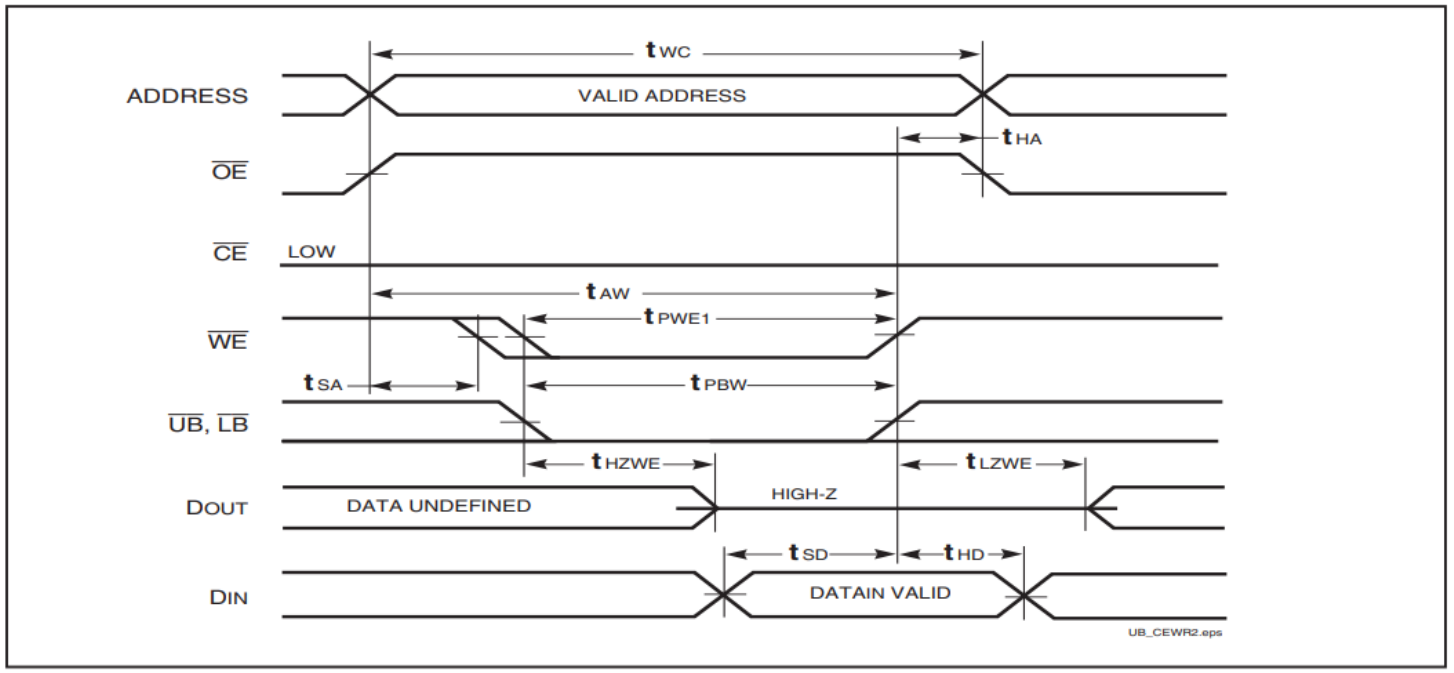

tRC
ADDRESS
tAA
tOHA
tOHA
DOUT
PREVIOUS DATA VALID
DATA VALID
READ1.eps


clock
addr
0x01
0x02
0x03
0x01
data
A
B
C
A
ce_n
oe_n
we_n
be_n
0b0000

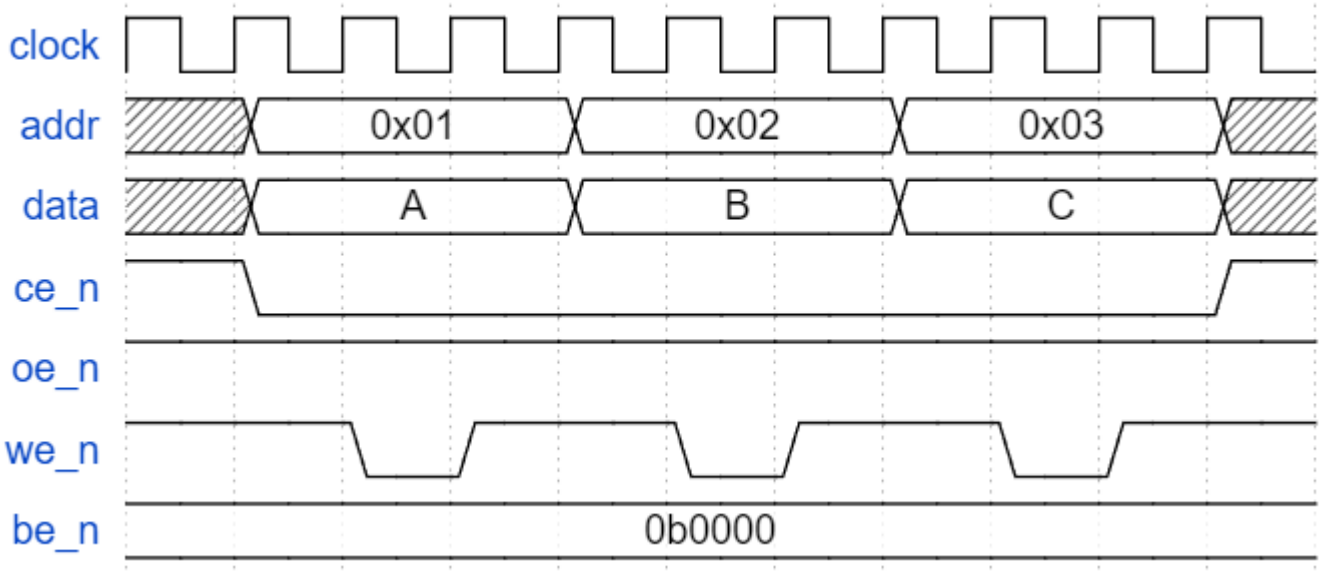
内存实验

写时序

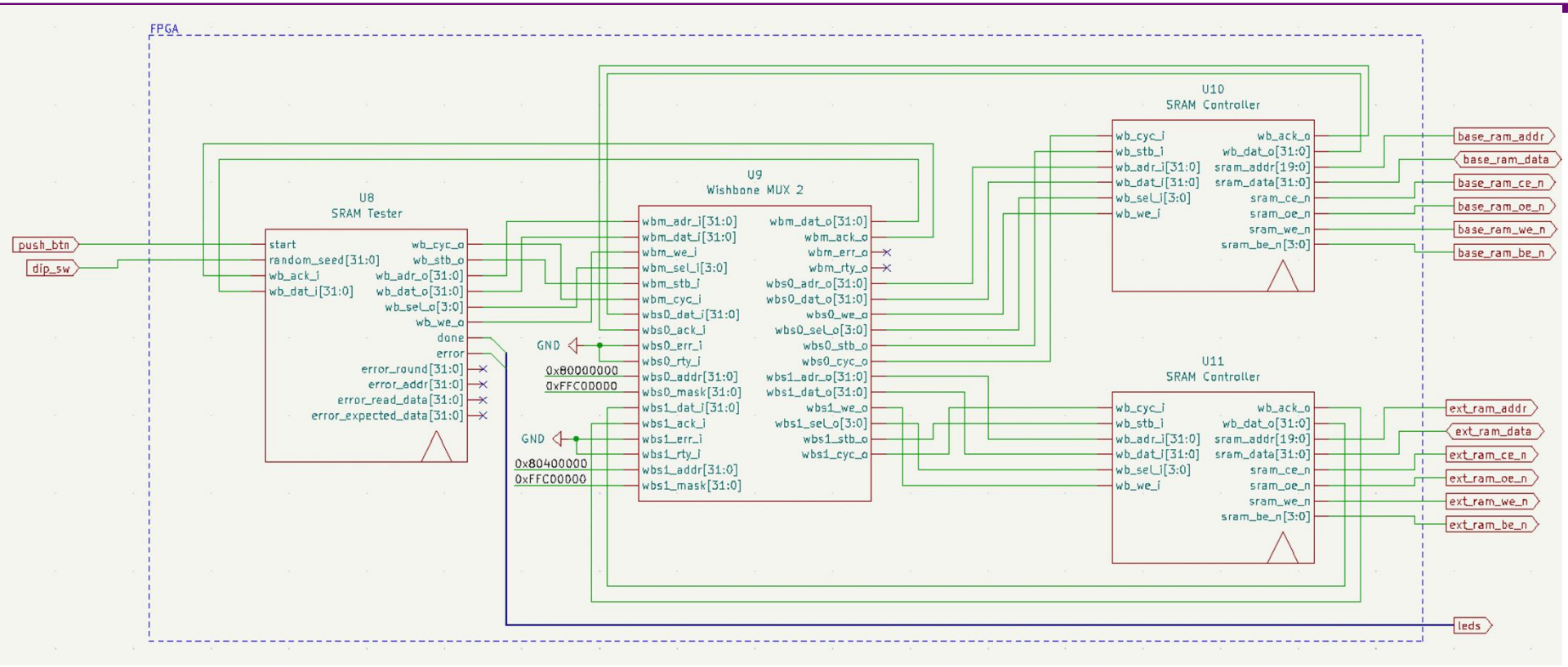
WRITE CYCLE NO. 2 ($\overline{WE}$ Controlled. $\overline{OE}$ is HIGH During Write Cycle) (1,2)

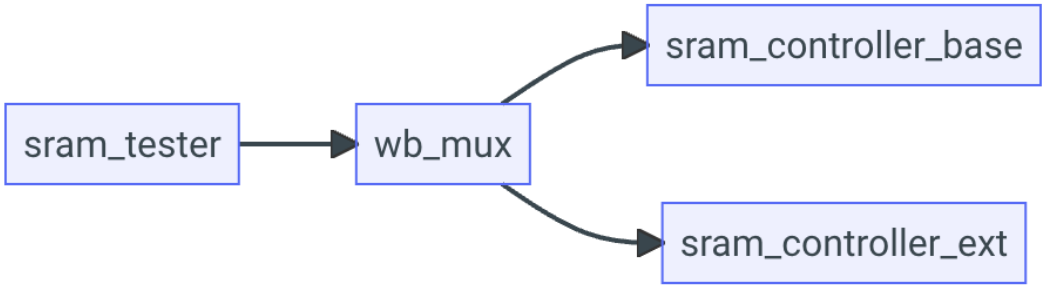


内存实验



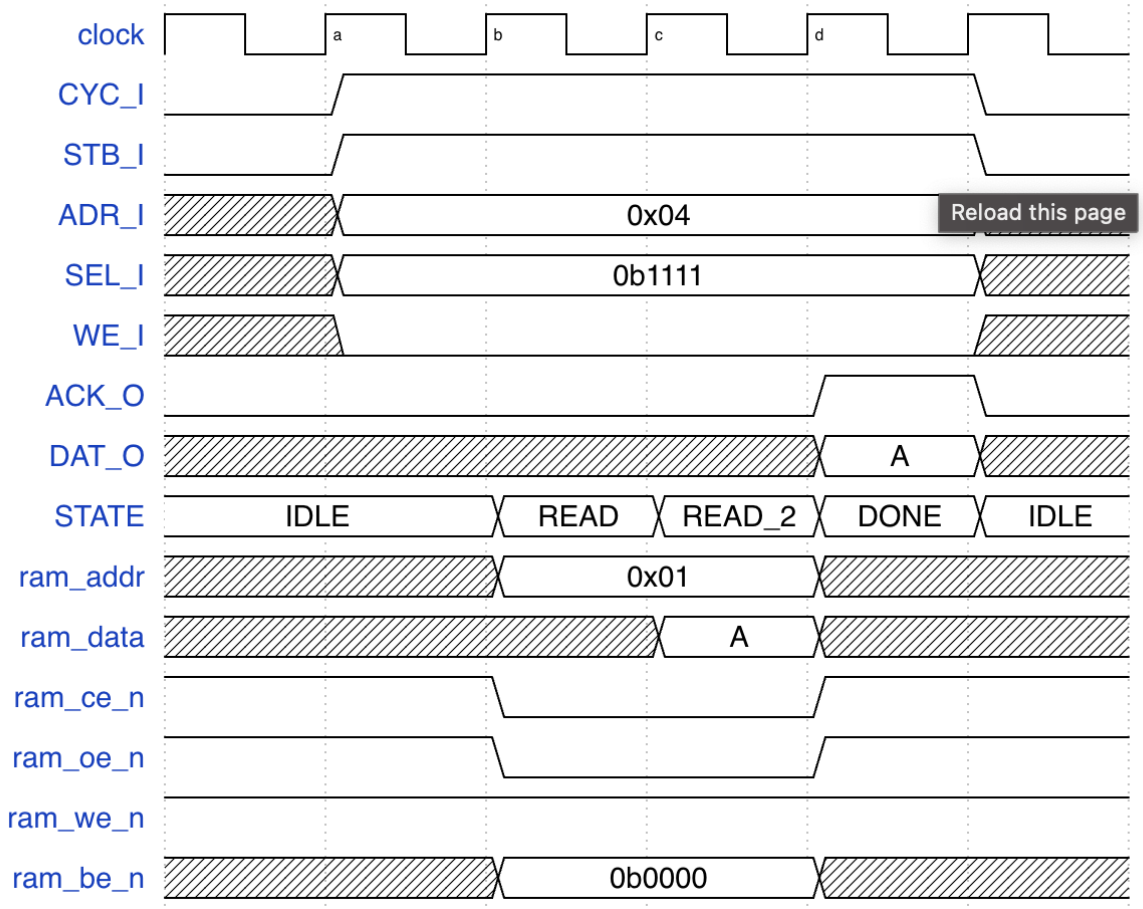
实验电路


FPGA
U8
SRAM Tester
start
random_seed[31:0]
wb_ack_i
wb_dat_i[31:0]
wb_cyc_o
wb_stb_o
wb_adr_o[31:0]
wb_dat_o[31:0]
wb_sel_o[3:0]
wb_we_o
done
error
error_round[31:0]
error_addr[31:0]
error_read_data[31:0]
error_expected_data[31:0]
U9
Wishbone MUX 2
wbm_adr_i[31:0]
wbm_dat_i[31:0]
wbm_we_i
wbm_sel_i[3:0]
wbm_stb_i
wbm_cyc_i
wbs0_dat_i[31:0]
wbs0_ack_i
wbs0_err_i
wbs0_rty_i
wbs0_addr[31:0]
wbs0_mask[31:0]
wbs1_dat_i[31:0]
wbs1_ack_i
wbs1_err_i
wbs1_rty_i
wbs1_addr[31:0]
wbs1_mask[31:0]
wbm_dat_o[31:0]
wbm_ack_o
wbm_err_o
wbm_rty_o
wbs0_adr_o[31:0]
wbs0_dat_o[31:0]
wbs0_we_o
wbs0_sel_o[3:0]
wbs0_stb_o
wbs0_cyc_o
wbs1_adr_o[31:0]
wbs1_dat_o[31:0]
wbs1_we_o
wbs1_sel_o[3:0]
wbs1_stb_o
wbs1_cyc_o
GND
0x80000000
0xFFC00000
GND
0x80400000
0xFFC00000
U10
SRAM Controller
wb_cyc_i
wb_stb_i
wb_adr_i[31:0]
wb_dat_i[31:0]
wb_sel_i[3:0]
wb_we_i
wb_ack_o
wb_dat_o[31:0]
sram_addr[19:0]
sram_data[31:0]
sram_ce_n
sram_oe_n
sram_we_n
sram_be_n[3:0]
U11
SRAM Controller
wb_cyc_i
wb_stb_i
wb_adr_i[31:0]
wb_dat_i[31:0]
wb_sel_i[3:0]
wb_we_i
wb_ack_o
wb_dat_o[31:0]
sram_addr[19:0]
sram_data[31:0]
sram_ce_n
sram_oe_n
sram_we_n
sram_be_n[3:0]
push_btn
dip_sw
base_ram_addr
base_ram_data
base_ram_ce_n
base_ram_oe_n
base_ram_we_n
base_ram_be_n
ext_ram_addr
ext_ram_data
ext_ram_ce_n
ext_ram_oe_n
ext_ram_we_n
ext_ram_be_n
leds


sram_tester
wb_mux
sram_controller_base
sram_controller_ext

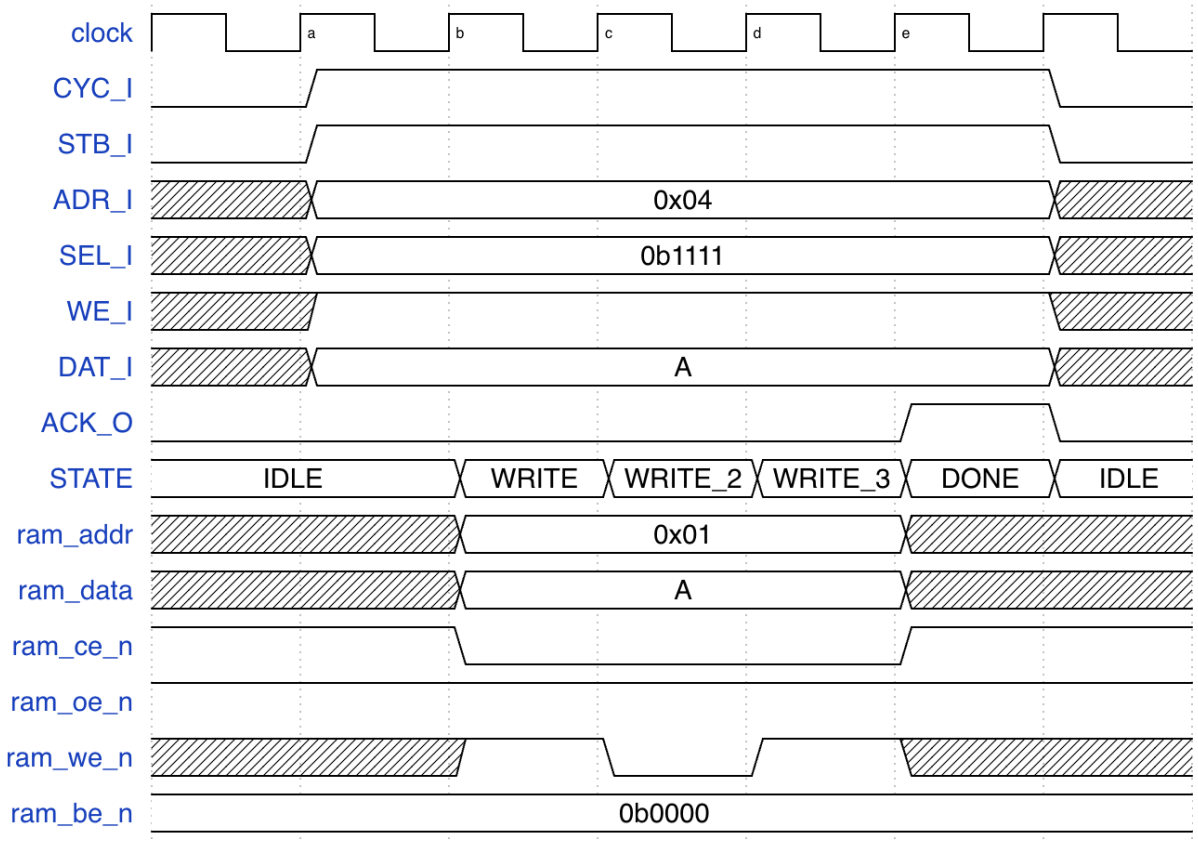
内存实验

读


clock
CYC_I
STB_I
ADR_I
SEL_I
WE_I
ACK_O
DAT_O
STATE
ram_addr
ram_data
ram_ce_n
ram_oe_n
ram_we_n
ram_be_n
a
b
c
d
0x04
Reload this page
0b1111
A
IDLE
READ
READ_2
DONE
IDLE
0x01
A
0b0000

内存实验

写


clock
a
b
c
d
e
CYC_I
STB_I
ADR_I
0x04
SEL_I
0b1111
WE_I
DAT_I
A
ACK_O
STATE
IDLE
WRITE
WRITE_2
WRITE_3
DONE
IDLE
ram_addr
0x01
ram_data
A
ram_ce_n
ram_oe_n
ram_we_n
ram_be_n
0b0000

三态门

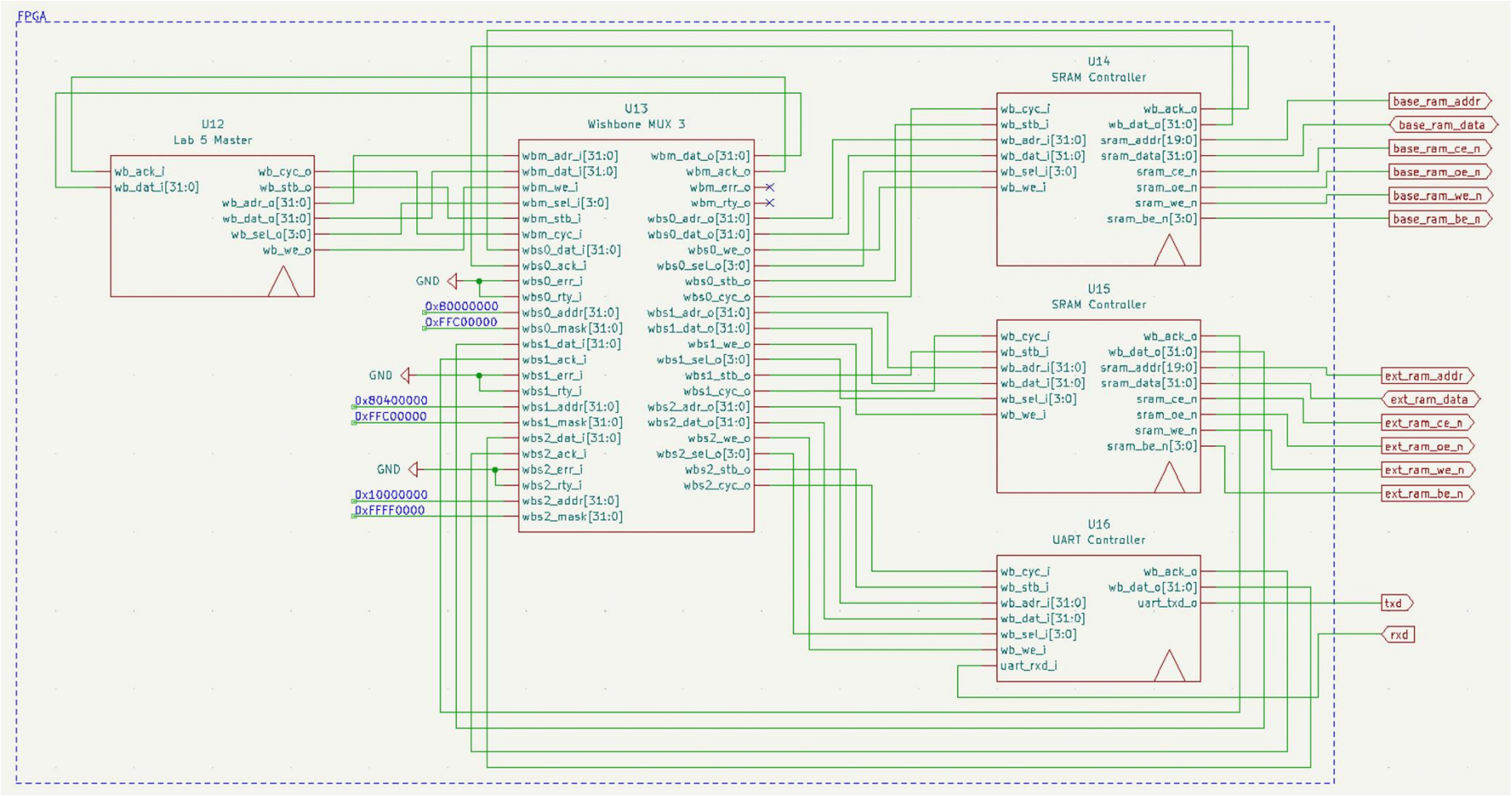
- 在 SystemVerilog 代码中，我们通常会将三态门 signal_io 拆分成三个信号：signal_i、signal_o 和 signal_t，分别表示输入、输出和高阻态

```
module sram_controller (
    inout [31:0] sram_data
);

    reg [31:0] sram_data_i_comb;
    reg [31:0] sram_data_o_comb;
    wire sram_data_t_comb;

    assign sram_data = sram_data_t_comb ? 32'bz : sram_data_o_comb;
    assign sram_data_i_comb = sram_data;

    always_comb begin
        sram_data_t_comb = 1'b0;
        sram_data_o_comb = 32'b0;

        // ...
    end
endmodule
```

实验5：UART实验

- 通过 Wishbone 协议及给出的 Wishbone UART 控制器、之前编写的 Wishbone SRAM 控制器，同时完成对教学机 UART 和 SRAM 的访问。实现和外部设备（实验中为 ThinPAD-Cloud 平台）的通信。要求分别实现如下功能：
  1. 使用云平台向 FPGA 的串口发送 10 个字节的随机数，FPGA 将随机数存储到 BaseRAM 地址从 Addr 开始的连续 10 个 word 上（存储进最低 8 位，高 24 位不使用）。
  2. FPGA 将上述 10 个随机数发送回串口，云平台从 FPGA 的串口接收 10 个字节。
  3. 程序读取 BaseRAM 中对应位置的数据，并将 SRAM 中数据、串口接收到的数据与初始随机数进行对比，对比时只考虑 SRAM 数据的最低 8 位。
- 同学需要在顶层模块中实现 Wishbone Master 状态机，根据拨码开关的输入，生成总线上的请求，从而实现实验的要求。

实验5电路图



重点实现Wishbone Master

UART协议

- UART,通用异步收发传输器(Universal Asynchronous Receiver/Transmitter)
- 全双工，单工
- 原理：
  - 输出的信号只有 0 和 1
  - 固定一个时间间隔，即把 1s 分成很多份，每个时间间隔内，信号一直是 0 或者 1 保持不变
  - 如果要传输一个字节，8 个时间间隔，每个时间间隔传一个 bit
  - 为了把传输的每个字节区分开，设定：不传输数据的时候一直输出 1，传输字节前输出一个时间间隔的 0，然后是字节的 8 个位，最后输出一个时间间隔的 1
- 我们通常把 UART 协议中 1s 时间间隔的个数记为波特率（Baud Rate）。常见的波特率有 9600 和 115200。

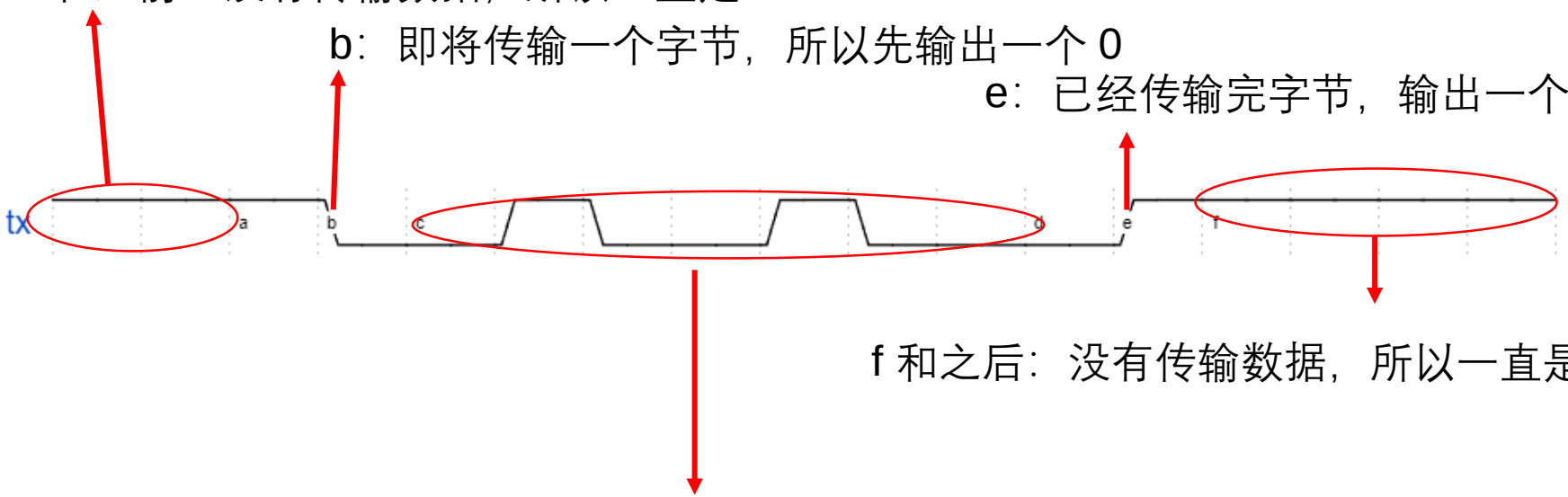
UART协议

- 要传输 0x12 字节，转换为二进制，就是 0b00010010，然后分成 8 次，一个一个位传输

a 和之前：没有传输数据，所以一直是 1

b：即将传输一个字节，所以先输出一个 0

e：已经传输完字节，输出一个 1

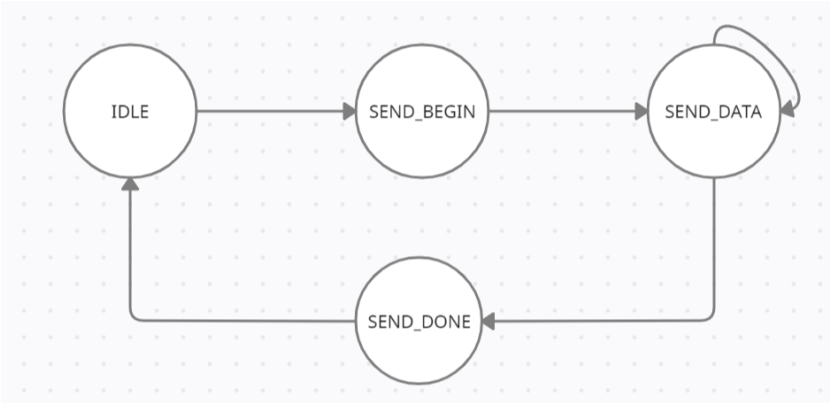


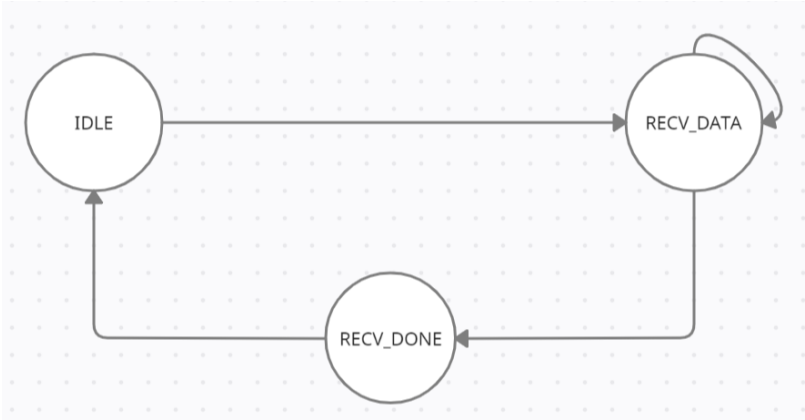
f 和之后：没有传输数据，所以一直是 1

c 到 d：8 个时间间隔，输出 0, 1, 0, 0, 1, 0, 0, 0，即 0x12 从最低位到最高位

Wishbone 串口控制器

- 发送：向串口控制器写入一个个字节，然后串口控制器负责按照 UART 协议的标准，把字节发送出去
- 接收：从信号中恢复出要传输的字节，然后再从串口控制器读取出一个个字节



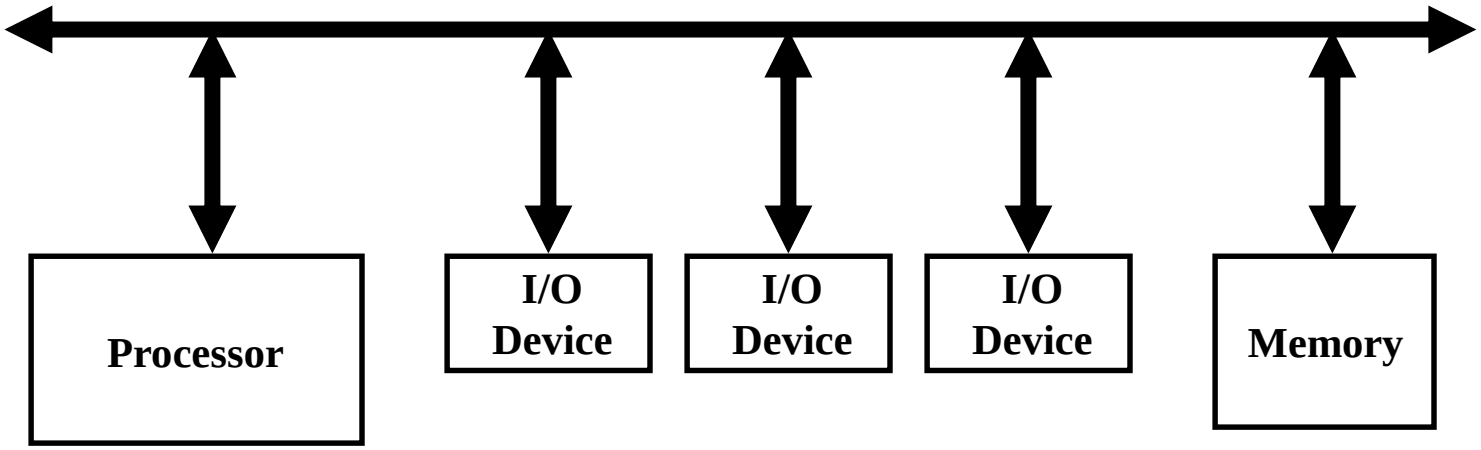


细节问题

- 时钟频率和波特率不一样，例如时钟周期是 50MHz，波特率只有 115200，中间差距很大；解决方法是分频，也就是数拍子，发送的时候，要等待许多个周期，直到经过了一个串口的时间间隔
- 接收数据的时候，由于波特率相比时钟频率很低，所以每次接收一个 bit 都会经历很多个周期，所以接收的时候也要等待，选择合适的时间把数据存下来
- 相比 CPU，串口控制器的处理是十分慢的，所以需要告诉 CPU 什么时候可以发送下一个字节、什么时候可以读取一个字节。发送的时候，只要状态机转移就算完成，不需要等到一次完整的发送完成，那样会浪费很多 CPU 时间。

MMIO

- Memory mapping I/O，即内存映射I/O



串口控制器总线接口

- 发送→写数据寄存器
  - 0x10000000：向串口发送一个字节的数据
- 发送太慢→读状态寄存器：
  - 0x10000001：不为零表示现在串口控制器可以发送数据

```
    // 假设要发送的数据在 a0
LOOP:
    // 读取 0x10000001
    li a1, 0x10000001
    lb a2, 0(a1)
    beqz a2, LOOP

    // 可以发送数据了
    li a1, 0x10000000
    // 发送
    sb a0, 0(a1)
```

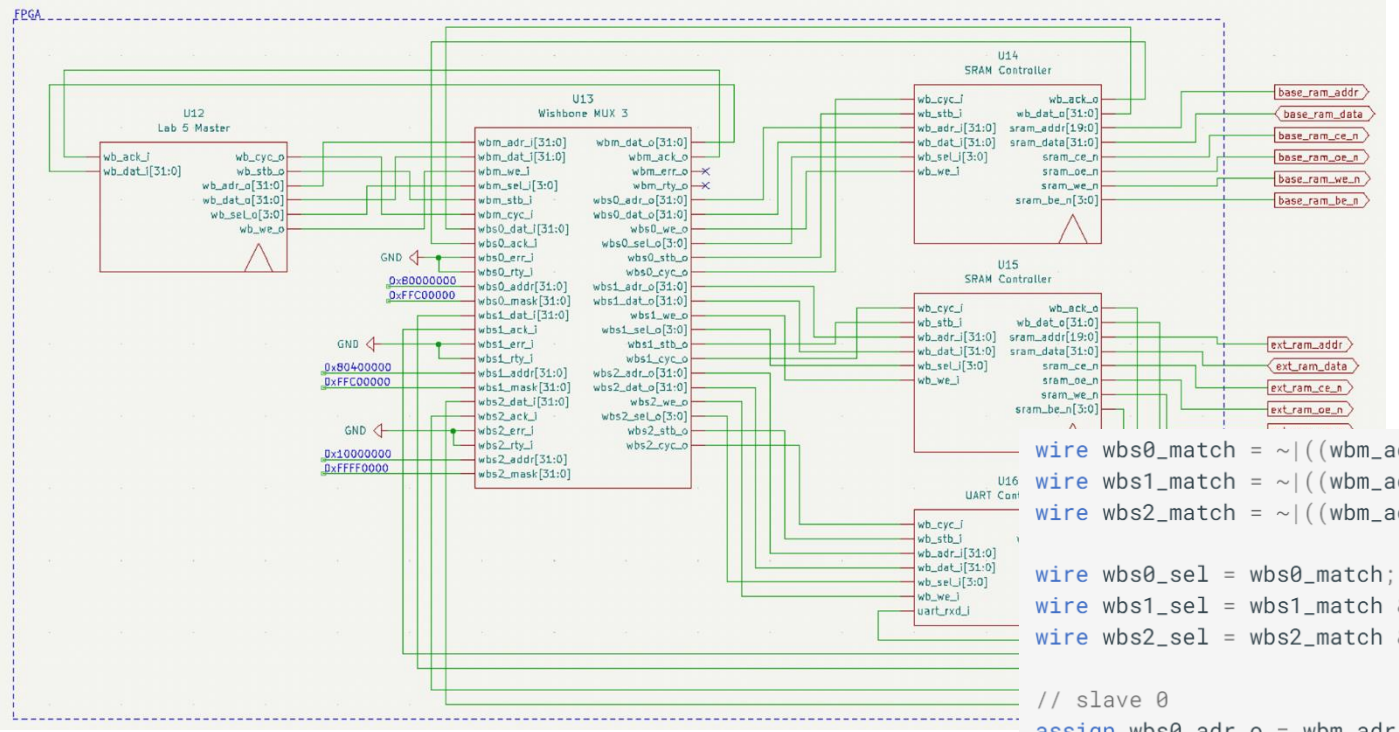
串口控制器总线接口

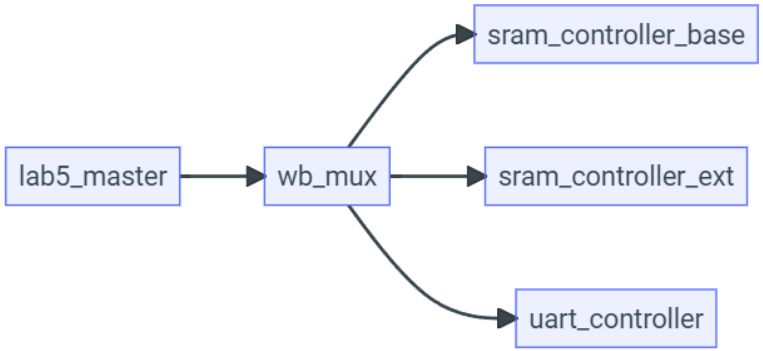
- 接收→读数据寄存器
  - 0x10000003：如果 CPU 读取这个地址，就会读取出串口控制器当前收到的数据
- 是否有数据
  - 0x10000002：不为零表示现在串口控制器已经接受到了数据，CPU 可以读取

地址	位	说明
0x10000000	[7:0]	串口数据，读、写地址分别表示串口接收、发送一个字节
0x10000005	[5]	只读，为 1 时表示串口空闲，可发送数据
0x10000005	[0]	只读，为 1 时表示串口收到数据

Wishbone MUX

- 根据地址来动态“连接”CPU 和外设





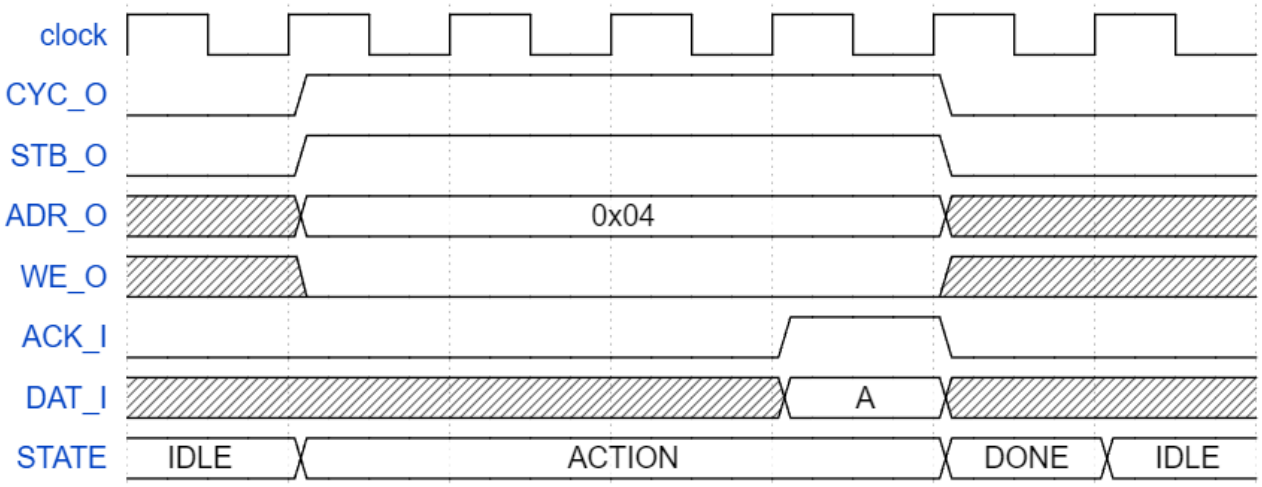

```
wire wbs0_match = ~|((wbm_adr_i ^ wbs0_addr) & wbs0_addr_msk);
wire wbs1_match = ~|((wbm_adr_i ^ wbs1_addr) & wbs1_addr_msk);
wire wbs2_match = ~|((wbm_adr_i ^ wbs2_addr) & wbs2_addr_msk);

wire wbs0_sel = wbs0_match;
wire wbs1_sel = wbs1_match & ~(wbs0_match);
wire wbs2_sel = wbs2_match & ~(wbs0_match | wbs1_match);

// slave 0
assign wbs0_adr_o = wbm_adr_i;
assign wbs0_dat_o = wbm_dat_i;
assign wbs0_we_o = wbm_we_i & wbs0_sel;
assign wbs0_sel_o = wbm_sel_i;
assign wbs0_stb_o = wbm_stb_i & wbs0_sel;
assign wbs0_cyc_o = wbm_cyc_i & wbs0_sel;

// master
assign wbm_dat_o = wbs0_sel ? wbs0_dat_i :
                   wbs1_sel ? wbs1_dat_i :
                   wbs2_sel ? wbs2_dat_i :
                   {DATA_WIDTH{1'b0}};
```

Wishbone Master 实现

- 状态机
  - IDLE ：如果想要发送一次请求，就把请求的地址等信息保存在寄存器中，并设置 STB_O=1, CYC_O=1，转移到 ACTION 状态
  - ACTION：等待 ACK_I=1；如果发现 ACK_I=1，把返回的数据 DAT_I 记录下来，转移到 DONE 状态
  - DONE：根据需求对结果做一些处理，然后转移到 IDLE 状态

Wishbone Master 实现

```
// wishbone master
output reg wb_cyc_o,
output reg wb_stb_o,
input wire wb_ack_i,
output reg [ADDR_WIDTH-1:0] wb_adr_o,
output reg [DATA_WIDTH-1:0] wb_dat_o,
input wire [DATA_WIDTH-1:0] wb_dat_i,
output reg [DATA_WIDTH/8-1:0] wb_sel_o,
output reg wb_we_o
```



谢谢