实验十 CPU 数据通路

2022 年春季学期

我梦见自己慢慢地升了起来，穿过数据平面，穿过数据网，进入并穿过万方网，最后来到了一个不认识的地方，我从没梦见过的地方……这个地方，空间无限，颜色悠闲、难以形容，没有地平线，没有天，没有地或者人类称为地面的实体区。

–《海伯利安的陨落》,丹 · 西蒙斯

本实验的目标是利在单周期 CPU 的实现之前，先完成 CPU 数据通路中的三个重要部分：寄存器堆、ALU 和数据存储器，并通过功能仿真测试。

10.1 寄存器堆

寄存器堆是 CPU 中用于存放指令执行过程中临时数据的存储单元。我们将要实现的 RISC-V 的基础版本 CPU RV32I 具有 32 个寄存器。RV32I 采用Load Store架构，即所有数据都需要先用 Load 语句从内存中读取到寄存器里才可以进行算术和逻辑操作。因此，RV32I 有 32 个通用寄存器，且每条算术运算可能要同时读取两个源寄存器并写入一个目标寄存器。为支持高速，多端口并行存取的寄存器堆，我们不能直接调用通用的 RAM，而需要用 Verilog 语言单独编写寄存器堆。

10.1.1 RV32I 中的通用寄存器

RV32I 共 32 个 32bit 的通用寄存器 x0~x31(寄存器地址为 5bit 编码），其中寄存器 x0 中的内容总是 0，无法改变。其他寄存器的别名和寄存器使用约定参见表 10–1。需要注意的是，部分寄存器在函数调用时是由调用方（Caller）来负责保存的，部分寄存器是由被调用方（Callee）来保存的。在进行 C 语言和汇编混合编程时需要注意。

表 10-1: RV32I 中通用寄存器的定义与用法

Register	Name	Use	Saver
x0	zero	Constant 0	–
x1	ra	Return Address	Caller
x2	sp	Stack Pointer	Callee
x3	gp	Global Pointer	–
x4	tp	Thread Pointer	–
x5~x7	t0~t2	Temp	Caller
x8	s0/fp	Saved/Frame pointer	Callee
x9	s1	Saved	Callee
x10~x11	a0~a1	Arguments/Return Value	Caller
x12~x17	a2~a7	Arguments	Caller
x18~x27	s2~s11	Saved	Callee
x28~x31	t3~t6	Temp	Caller

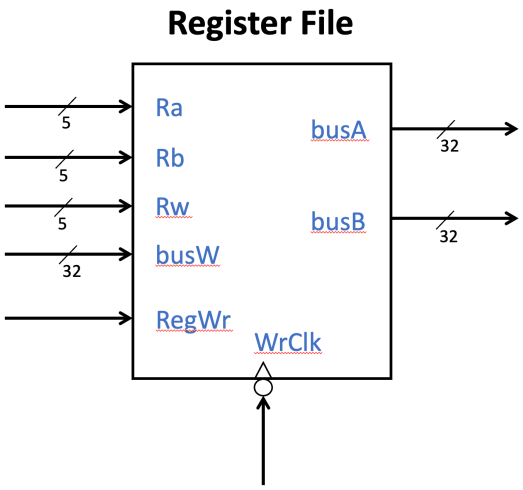

图 10-1: 寄存器堆接口示意图

10.1.2 寄存器堆实现

图 10-1描述了寄存器堆的接口，该寄存器堆中有 32 个 32bit 的寄存器。寄存器堆需要支持同时两个读操作和一个写操作。因此需要有 2 个读地址 Ra 和 Rb，分别对应 RISC-V 汇编中的 rs1 和 rs2。写地址为 Rw，对应 rd。地址均是 5 位。写入数据 busW 为 32 位，写入有效控制为一位高电平有效的 RegWr 信

表 10-2: 控制信号 ALUctr 的含义

ALUctr[3]	ALUctr[2:0]	ALU 操作
0	000	选择加法器输出，做加法
1	000	选择加法器输出，做减法
×	001	选择移位器输出，左移
0	010	做减法，选择带符号小于置位结果输出,Less 按带符号结果设置
1	010	做减法，选择无符号小于置位结果输出,Less 按无符号结果设置
×	011	选择 ALU 输入 B 的结果直接输出
×	100	选择异或输出
0	101	选择移位器输出，逻辑右移
1	101	选择移位器输出，算术右移
×	110	选择逻辑或输出
×	111	选择逻辑与输出

号。寄存器堆的输出是 2 个 32 位的寄存器数据，分别是 busA 和 busB。寄存器堆有一个控制写入的时钟 WrClk。在时序上我们可以让读取是非同步的，即地址改变立刻输出。写入可以在时钟下降沿写入。注意，寄存器 x0 需要特殊处理，不论何时都是全零。请自行思考如何实现 x0 寄存器。

10.2 ALU

ALU 是 CPU 中的核心数据通路部件之一，它主要完成 CPU 中需要进行的算术逻辑运算，我们在前面的实验中已经实现了一个简单的 ALU。在本实验中只需要对该 ALU 稍加改造即可。针对 RV32I 的运算需求，我们对 ALU 的控制信号进行了重新定义，如表 10-2所示。该 ALU 的逻辑图如图 10-2所示。ALU 对输入数据并行地进行加减法、移位、比较大小、异或等操作。最终 ALUout 输出是通过一个八选一选择器选择不同运算部件的结果，选择器的控制端可以用 ALUctr[2:0] 直接生成。ALU 其他部件的控制信号需要的控制信号包括：A/L 控制移位器进行算术移位还是逻辑移位，L/R 控制是左移还是右移，U/S 控制比较大小是带符号比较还是无符号比较，S/A 控制是加法还是减法。这些控制信号需要按照所需进行的操作对应设置，请同学们自行设计。注意：比较大小或判断相等时应使用减法操作。

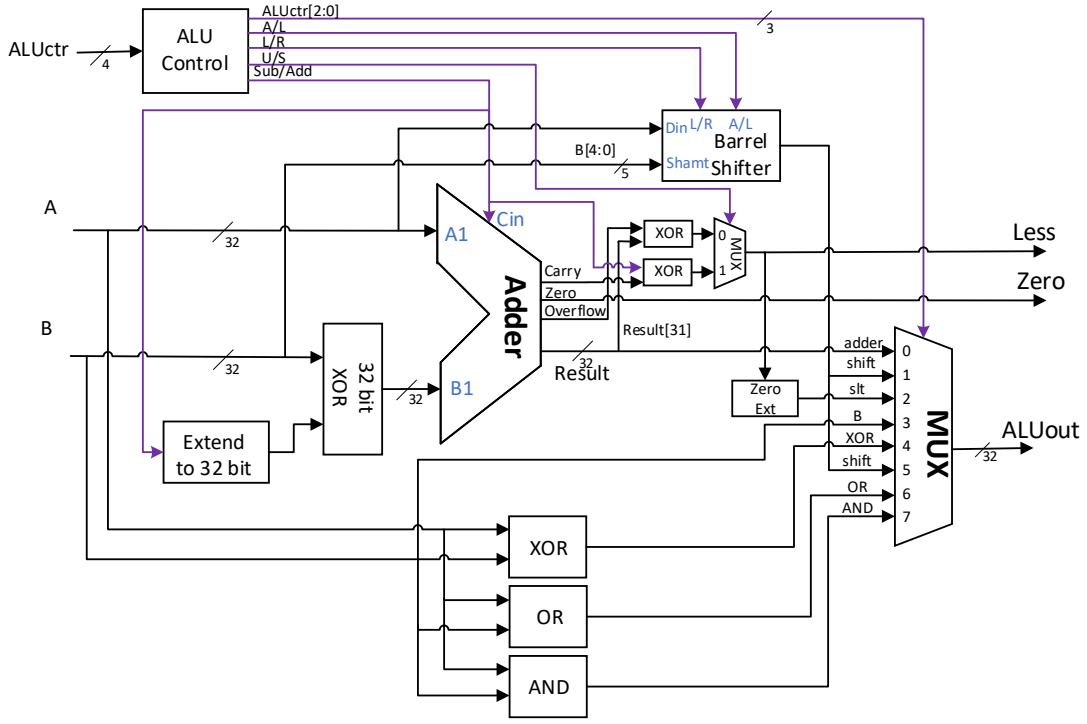

图 10–2: ALU 电路示意图

10.3 数据存储器

数据存储器在 CPU 运行中存储全局变量、堆栈等数据。我们建议大家实现至少 128kB 大小的数据存储器容量。并且，该数据存储器需要支持在沿上进行读取和写入操作。RV32I 的字长是 32bit，但是，数据存储器不仅要支持 32bit 数据的存取，同时也需要支持按字节（8bit) 或半字 (16bit) 大小的读取。由于单周期 CPU 需要在一个周期内完成一条指令的所有操作，我们需要数据 RAM 有独立的读时钟和写时钟。其中读取操作在系统时钟的上升沿进行（即一个时钟周期的一半时刻），写操作在系统时钟的下降沿进行（即一个时钟周期的结束时刻）。建议使用双端口 RAM（RAM 2 PORT）来实现数据存储器。DE10-Standard 开发板上的大容量 SRAM 是支持独立的读写时钟的。一般可以支持 128KB 以上的数据存储容量。

要实现按字节（8 bit) 或按半字 (16 bit) 大小的读写，需要同学们对 IP 核生成的存储器进行一定程度的改造。在实现过程中不需要考虑 4 字节读写或 2 字节读写时地址未对齐的情况。默认 4 字节读写时地址低 2 位为 00，2 字节读写

时地址最低位为 0。

具体地，MemOP 信号定义如下：宽度为 3bit，控制数据存储器读写格式，为 010 时为 4 字节读写，为 001 时为 2 字节读写带符号扩展，为 000 时为 1 字节读写带符号扩展为 101 时为 2 字节读写无符号扩展为 100 时为 1 字节读写无符号扩展。

MemOP 与 RV32 中的存储器操作对应关系如下：

表 10-3: 存储访问指令与 Memop 对应关系

指令		MemOP	操作
`lb`	`rd,imm12(rs1)`	000	R[rd] ← SEXT(M_{1B}[R[rs1] + SEXT(imm12)])
`lh`	`rd,imm12(rs1)`	001	R[rd] ← SEXT(M_{2B}[R[rs1] + SEXT(imm12)])
`lw`	`rd,imm12(rs1)`	010	R[rd] ← M_{4B}[R[rs1] + SEXT(imm12)]
`lbu`	`rd,imm12(rs1)`	100	R[rd] ← {24'b0, M_{1B}[R[rs1] + SEXT(imm12)]}
`lhu`	`rd,imm12(rs1)`	101	R[rd] ← {16'b0, M_{2B}[R[rs1] + SEXT(imm12)] }
`sb`	`rs2,imm12(rs1)`	000	M_{1B}[R[rs1] + SEXT(imm12)] ← R[rs2][7:0]
`sh`	`rs2,imm12(rs1)`	001	M_{2B}[R[rs1] + SEXT(imm12)] ← R[rs2][15:0]
`sw`	`rs2,imm12(rs1)`	010	M_{4B}[R[rs1] + SEXT(imm12)] ← R[rs2]

对于读取操作，我们可以每次均读取 32bit 的数据，然后根据 MemOP 来判断是需要 8bit，16bit 还是 32bit 的数据，再根据地址的低 2 位选择合适的数据拼接扩展成读取的结果即可。

对于写入操作，由于需要对 32bit 中特定的 8bit 或 16bit 数据进行写入，而不能破坏其他比特。因此在实现上需要慎重思考。我们提供以下三种解决思路，供大家参考。

- 利用 IP 核中支持单字节写入使能信号的双口 RAM 来实现。
  这种方式是我们推荐的方式。如图 10-3所示，在 Quartus 中配置双端口 RAM 的第 3 步时，我们可以选择生成单字节写入使能信号。例如，我们生成了位宽为 32bit 的 RAM 后，系统会对应生成 byteena_a[3:0] 的单字节写入使能信号，该信号是高有效。如果需要对 32 位四个 Byte 同时写入，可以将这个信号置为 4'b1111。如果只需要写入低 8 位，可以将这个信号置为 4'b0001。所以，在进行字节或半字写入时，我们只需要对应设置单字节写使能信号，并将写入的数据按正确的方式组成 32bit 一次性写入即可。
- 在写入之前先读取原有数据，修改后一次性写入 32 比特。

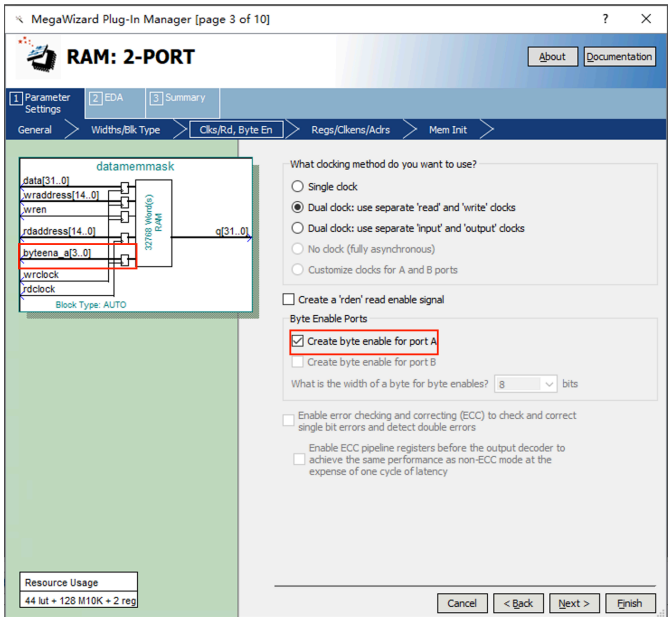

图 10-3: 双端口 RAM 中的单字节写入使能配置

IP 核生成的 RAM 不支持在仿真过程中对数据进行多次初始化，我们也可以用自己改写的 RAM 替代上述 IP 核中的带单字节写使能的 RAM 用于仿真。我们观察到，在单周期 CPU 中，CPU 在一个周期内只可能对内存执行读操作或写操作中的一种。因此，如果我们要写入 8bit 的数据又不想改变与它相邻的其他比特，我们可以在时钟上升沿读取将要写入的单元而不是读地址对应的单元。对读取的数据进行修改后，在下降沿将数据再次写回去即可。注意，此时要求写使能信号和写地址要在时钟上升沿就准备好。我们提供了如下的实现示例，其接口和 IP 核生成的双端口 RAM 是一致的。

```
1  module testdmem(
2      byteena_a,
3      data,
4      rdaddress,
5      rdclock,
6      wraddress,
7      wrclock,
8      wren,
9      q);
10
11     input    [3:0]    byteena_a;
```

```
12      input   [31:0]  data;
13      input   [14:0]  rdaddress;
14      input   rdclock;
15      input   [14:0]  wraddress;
16      input   wrclock;
17      input   wren;
18      output reg  [31:0]  q;
19
20      reg  [31:0] tempout;
21      wire [31:0] tempin;
22      reg [31:0] ram [32767:0];
23
24      always@(posedge rdclock)
25      begin
26         if(wren)
27              tempout<=ram[wraddress];
28          else
29              q <= ram[rdaddress];
30      end
31
32      assign tempin[7:0]   = (byteena_a[0])? data[7:0]  : tempout[7:0];
33      assign tempin[15:8]  = (byteena_a[1])? data[15:8] : tempout[15:8];
34      assign tempin[23:16] = (byteena_a[2])? data[23:16]: tempout[23:16];
35      assign tempin[31:24] = (byteena_a[3])? data[31:24]: tempout[31:24];
36
37      always@(posedge wrclock)
38      begin
39          if(wren)
40          begin
41              ram[wraddress]<=tempin;
42          end
43      end
44  endmodule
```

注意，这类自己写的 RAM 模块 Quartus 很有可能不会将其映射到 M10K 内存模块上来实现，直接导致系统资源不够或编译时间较长。在实际上板的代码中建议对于容量大于 64k 的存储都采用 IP 核生成的 RAM。

- 利用 4 片 8bit RAM 拼接成一个 32bit 的存储器。

  这种方法用 4 片 8bit 的 RAM 拼接成一个 32bit 的 RAM，每个 8bitRAM 负责 32 比特的给定部分。例如，RAM0 负责提供地址低两位为 00 的数据，RAM1 负责提供地址低两位为 01 的数据，以此类推。如果需要一次性读写 32bit 数据，我们将对应的数据和地址前 30 位连接到 4 片 RAM 上即可同时对 4 片 RAM 进行操作，一次读写 4×8 =32bit 数据。如果只需要写入 8bit 数据，可以根据地址低两位来控制 RAM 写使能端口，只对一片 RAM 进行写入。这种方法主要的问题是内存初始化的时候需要对四片 RAM 分别进行初始化，有一些麻烦。

10.4 实验验收要求

10.4.1 在线测试

请单独完成 CPU 的寄存器堆、ALU 和数据存储器的实现，并能够顺利通过课程在头歌系统上的两个测试。

必做 数据通路之 ALU 实现

必做 数据通路之存储实现

☞ **注意** 课程在线测试系统对时序和实现要求较高，如果无法通过在线测试，可以自行编写 test bench 由助教现场验收通过也可以完成验收。