

RISC-V子集实现的抽象视图，显示了主要功能单元和它们之间的主要连接。

所有指令首先使用程序计数器将指令地址提供给指令存储器。取指令后，指令使用的寄存器操作数由该指令的字段指定。一旦寄存器操作数被蚀刻，就可以操作它们来计算存储器地址（用于加载或存储），计算算术结果（用于整数算术逻辑指令），或者等式检查（用于分支） 。

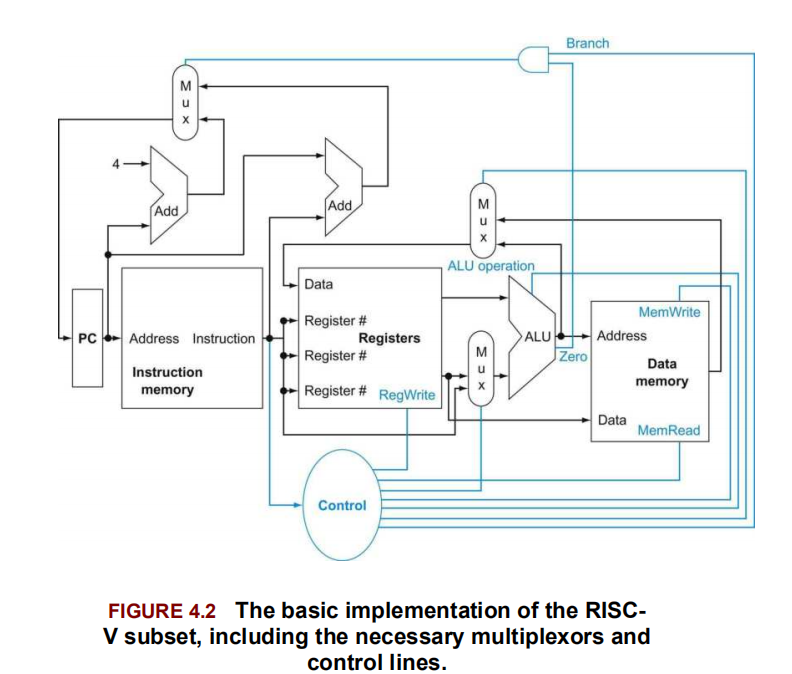
如果指令是算术逻辑指令，则必须将ALU的结果写入寄存器。

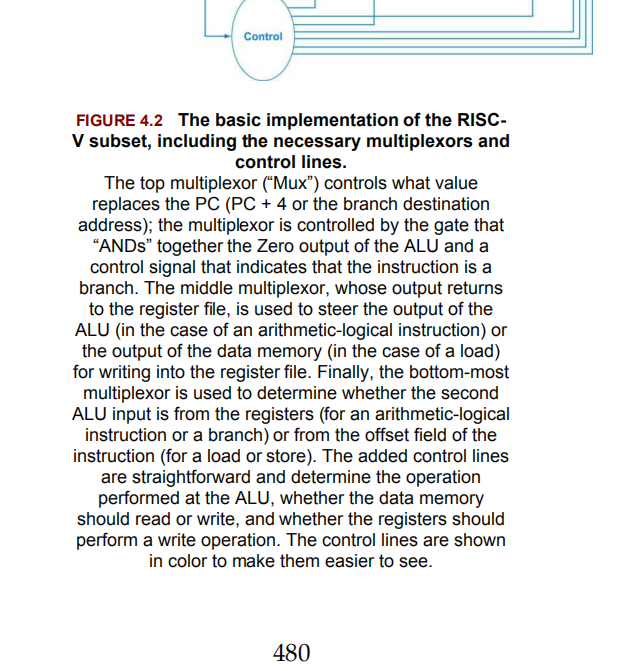
如果操作是加载或存储，则ALU结果用作地址，以存储寄存器中的值或将值从存储器加载到寄存器中。 ALU或内存的结果将写回寄存器文件。

分支需要使用ALU输出来确定下一个指令地址，该地址来自加法器（PC和分支偏移量相加）或来自将当前PC增加4的加法器。互连功能单元的粗线表示由多个信号组成的总线。箭头用于指导读者了解信息如何流动。由于信号线可以交叉，我们明确地显示当交叉线通过线交叉的点的存在而连接时。

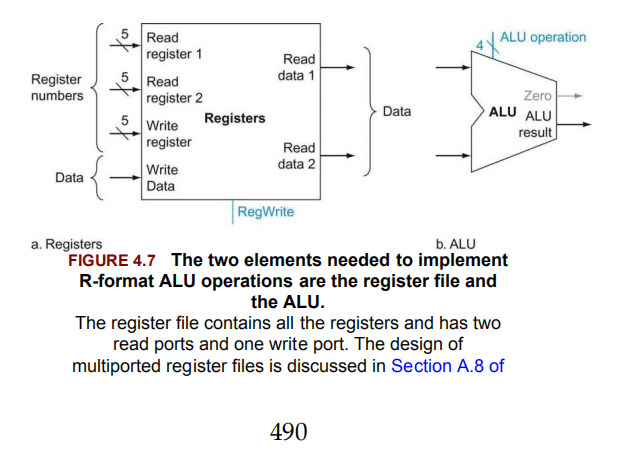
首先，在几个地方，图4.1显示了来自两个不同来源的特定单位的数据。例如，写入PC的值可以来自两个加法器中的一个，写入寄存器文件的数据可以来自ALU或数据存储器，ALU的第二个输入可以来自寄存器或直接指令领域。实际上，这些数据线不能简单地连接在一起;我们必须添加一个从多个源中选择的逻辑元素，并将其中一个源引导到其目的地。此选择通常使用称为多路复用器的设备完成，尽管此设备最好称为数据选择器。附录A描述了多路复用器，它根据其控制线的设置从多个输入中进行选择。控制线主要基于从正在执行的指令中获取的信息来设置。图4.1中的第二个省略是必须根据指令类型控制几个单元。例如，数据存储器必须读取加载并在商店上写入。寄存器文件只能在加载或算术逻辑指令上写入。当然，ALU必须执行多个操作之一。 （附录A描述了ALU的详细设计。）与多路复用器一样，基于指令中各个字段设置的控制线指导这些操作。

图4.2显示了图4.1的数据路径，其中添加了三个所需的多路复用器，以及主要功能单元的控制线。具有作为输入的指令的控制单元用于确定如何为功能单元和两个多路复用器设置控制线。确定PC +4或分支目的地地址是否写入PC的顶部多路复用器是基于ALU的零输出设置的，ALU用于执行beq指令的比较。 RISC-V指令集的规律性和简单性意味着可以使用简单的解码过程来确定如何设置控制线。

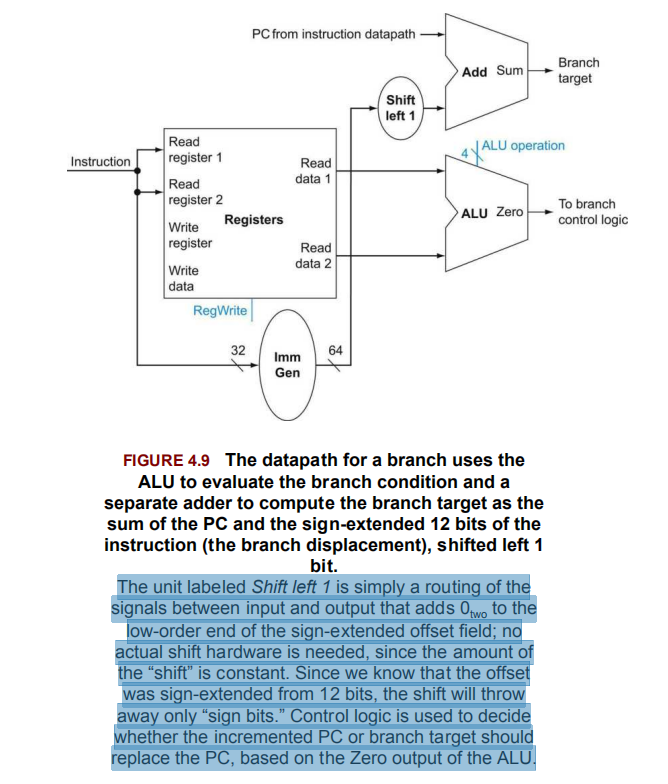




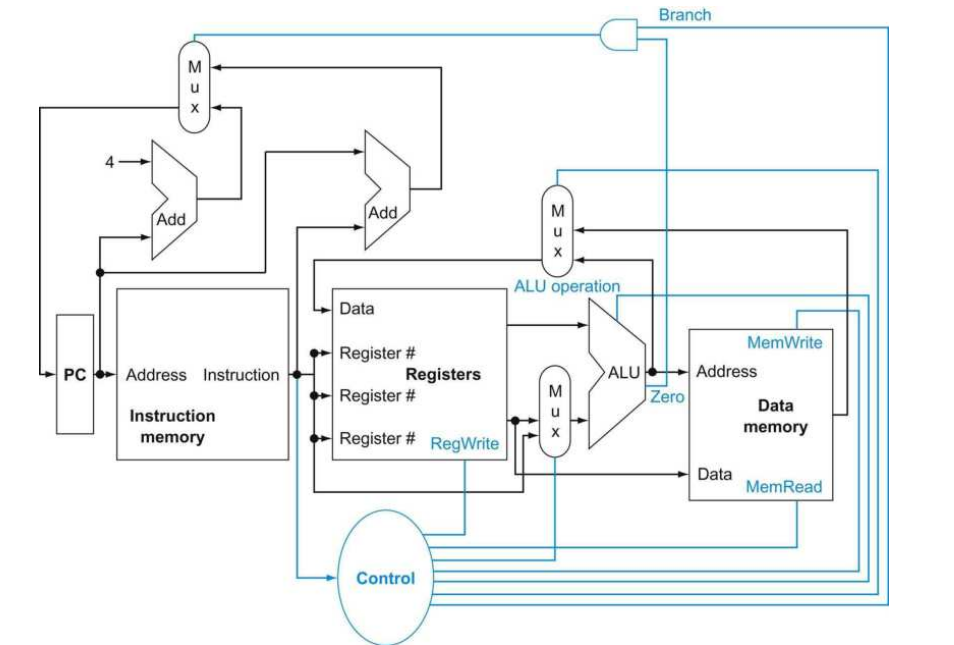
顶部多路复用器（“Mux”）控制取代PC的值（PC + 4或分支目标地址）;多路复用器由门控制，该门将ALU的零输出“和”在一起，并且控制信号指示该指令是分支。中间多路复用器，其输出返回寄存器文件，用于控制ALU的输出（在算术逻辑指令的情况下）或数据存储器的输出（在负载的情况下）用于写入寄存器文件。最后，最底部的多路复用器用于确定第二个ALU输入是来自寄存器（用于算术逻辑指令还是分支），还是来自指令的偏移字段（用于加载或存储）。添加的控制线很简单，并确定在ALU执行的操作，数据存储器是应该读还是写，以及寄存器是否应该执行写操作。控制线以彩色显示，以便于查看。



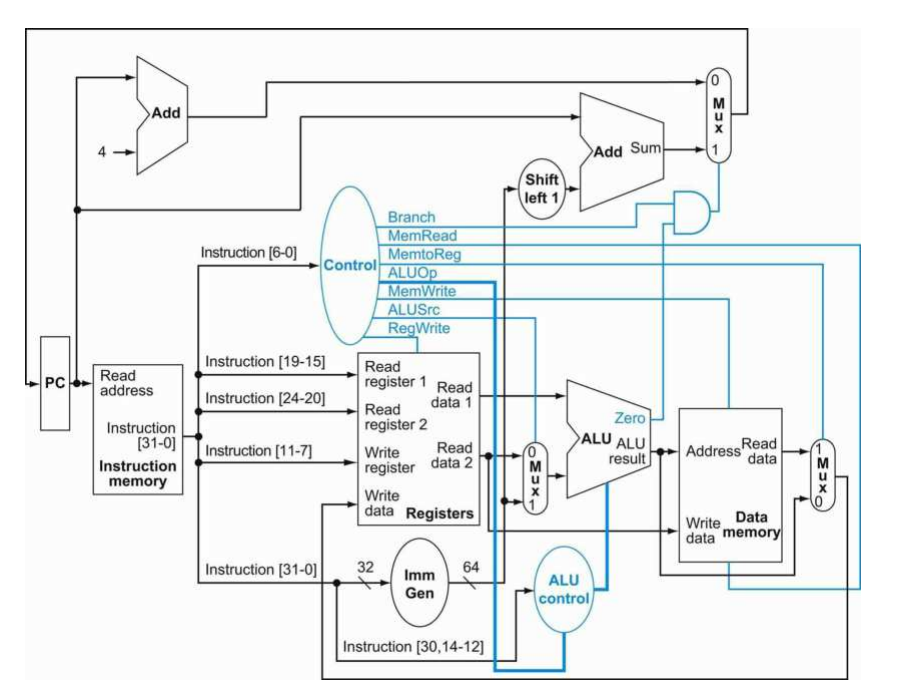
附录A.寄存器文件总是输出与输出上的读寄存器输入相对应的寄存器的内容;不需要其他控制输入。相反，必须通过断言写控制信号来明确指示寄存器写。请记住，写操作是边沿触发的，因此所有写输入（即要写入的值，寄存器编号和写控制信号）必须在时钟边沿有效。由于对寄存器文件的写入是边沿触发的，因此我们的设计可以在一个时钟周期内合法地读取和写入相同的寄存器：读取将获得在较早的时钟周期中写入的值，而写入的值将可用于读取随后的时钟周期。携带寄存器号到寄存器文件的输入都是5位宽，而携带数据值的行是64位宽。 ALU执行的操作由ALU操作信号控制，该信号为4位宽，使用附录A中设计的ALU。我们将很快使用ALU的零检测输出来实现条件分支。图4.7b显示了ALU，它采用两个64位输入并产生64位结果，如果结果为0，则为1位信号.ALU的4位控制信号详细描述于附录A;我们将在需要知道如何设置时立即查看ALU控件。接下来，考虑RISC-V加载寄存器和存储寄存器指令，其一般形式为ld x1，offset（x2）或sd x1，offset（x2）。这些指令通过将基址寄存器（x2）添加到指令中包含的12位有符号偏移字段来计算存储器地址。如果指令是存储，则还必须从寄存器文件中读取要存储的值，该寄存器文件位于x1中。如果指令是加载，则必须将从存储器读取的值写入指定寄存器中的寄存器文件，即x1。因此，我们需要寄存器文件和图4.7中的ALU。



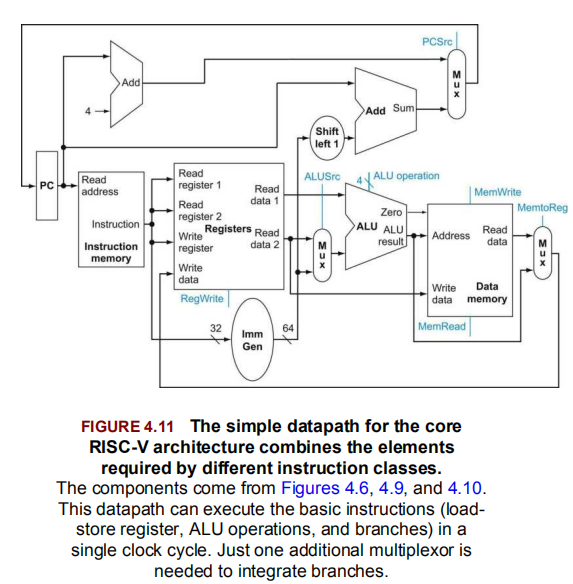
标记为Shift left 1的单位只是输入和输出之间信号的路由，它将0两倍加到符号扩展偏移字段的低阶端;不需要实际的换档硬件，因为“换档”的量是恒定的。由于我们知道偏移量是从12位进行符号扩展，因此移位将仅丢弃“符号位”。控制逻辑用于决定递增的PC或分支目标是否应该替换PC，基于零输出ALU。



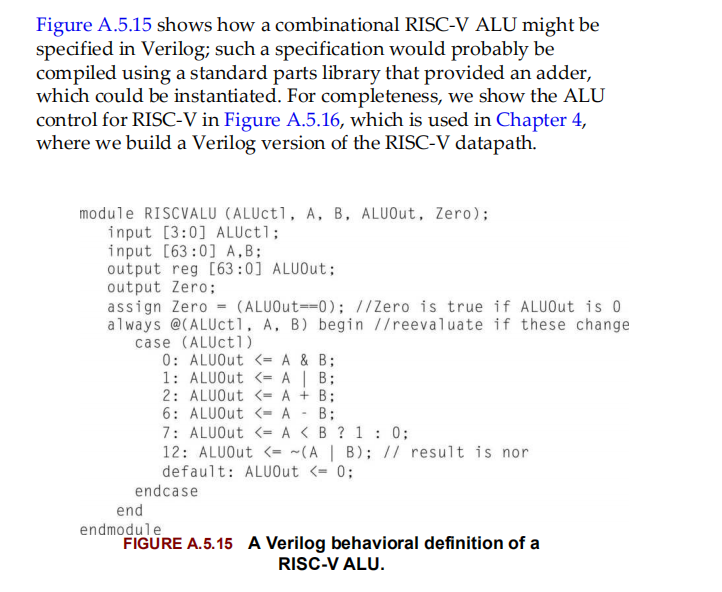
顶部多路复用器（“Mux”）控制取代PC的值（PC + 4或分支目标地址）;多路复用器由门控制，该门将ALU的零输出“和”在一起，并且控制信号指示该指令是分支。中间多路复用器，其输出返回寄存器文件，用于控制ALU的输出（在算术逻辑指令的情况下）或数据存储器的输出（在负载的情况下）用于写入寄存器文件。最后，最底部的多路复用器用于确定第二个ALU输入是来自寄存器（用于算术逻辑指令还是分支），还是来自指令的偏移字段（用于加载或存储）。添加的控制线很简单，并确定在ALU执行的操作，数据存储器是应该读还是写，以及寄存器是否应该执行写操作。控制线以彩色显示，以便于查看。

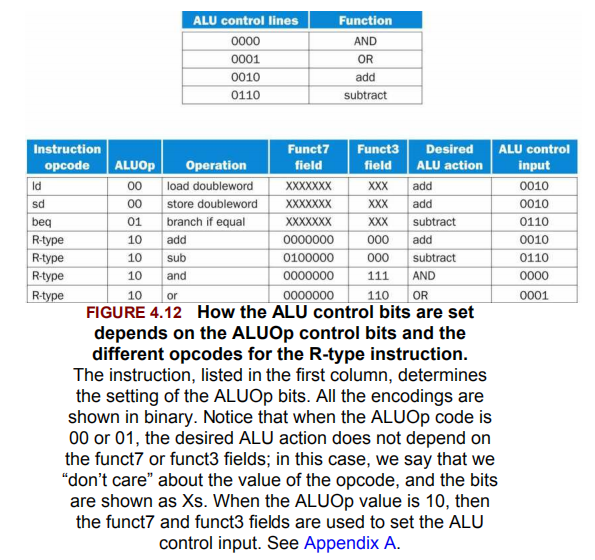


控制单元的输入是指令中的7位操作码字段。 控制单元的输出包括两个1位信号，用于控制多路复用器（ALUSrc和MemtoReg），三个信号用于控制寄存器文件和数据存储器中的读写（RegWrite，MemRead和MemWrite），1 用于确定是否可能分支（分支）的比特信号，以及用于ALU（ALUOp）的2比特控制信号。 AND门用于组合分支控制信号和来自ALU的零输出; AND门输出控制下一台PC的选择。 请注意，PCSrc现在是一个派生信号，而不是直接来自控制单元的信号。 因此，我们在后续数字中删除信号名称。



Page.1194

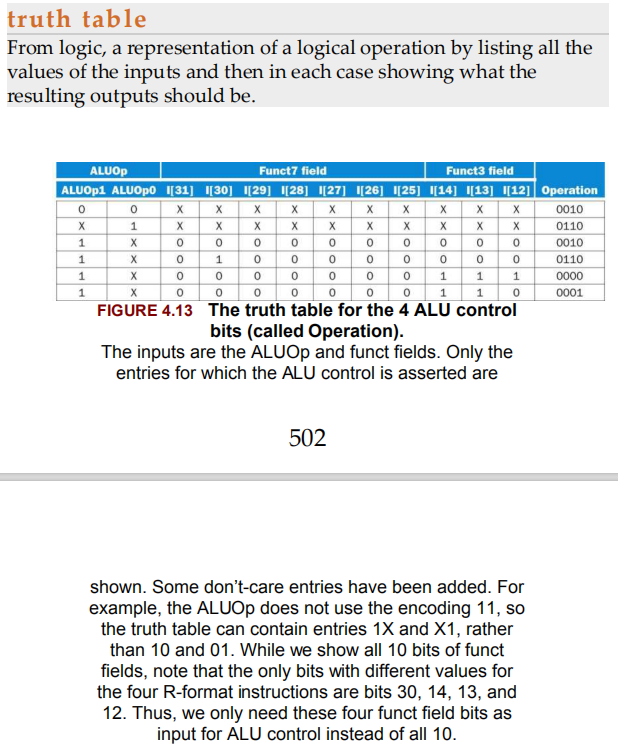




Aluop

根据指令类，ALU将需要执行这四个功能之一。对于加载和存储指令，我们使用ALU通过添加来计算内存地址。对于R类型指令，ALU需要执行四个动作之一（AND，OR，add或subtract），具体取决于7位funct7字段（位31:25）和3位funct3的值指令中的字段（位14:12）（参见第2章）。对于条件分支，如果是相等指令，则ALU减去两个操作数并测试结果是否为0.我们可以使用一个小控制单元生成4位ALU控制输入，该控制单元具有指令的funct7和funct3字段作为输入和一个2位控制字段，我们称之为ALUOp。 ALUOp指示要执行的操作是加载（00）加载和存储，减去并测试beq为零（01），还是由funct7和funct3字段（10）中编码的操作确定。ALU控制单元的输出是一个4位信号，它通过产生前面所示的4位组合之一直接控制ALU。在图4.12中，我们展示了如何基于2位ALUOp控件，funct7和funct3字段设置ALU控件输入。在本章的后面，我们将看到如何从主控制单元生成ALUOp位。

第一列中列出的指令确定ALUOp位的设置。所有编码都以二进制显示。请注意，当ALUOp代码为00或01时，所需的ALU操作不依赖于funct7或funct3字段;在这种情况下，我们说我们“不关心”操作码的值，并且位显示为Xs。当ALUOp值为10时，funct7和funct3字段用于设置ALU控制输入。见附录A.



因为在许多情况下我们并不关心某些输入的值，并且因为我们希望保持表格紧凑，所以我们还包括无关紧要的术语。 此真值表中的无关项（由输入列中的X表示）表示输出不依赖于与该列对应的输入值。 例如，当ALUOp位为00时，如图4.13的第一行，我们总是将ALU控制设置为0010，与功能字段无关。 在这种情况下，函数输入将不关心真值表的这一行。 稍后，我们将看到另一种不关心术语的例子。 如果您不熟悉无关护理术语的概念，请参阅附录A以获取更多信息。 一旦真值表被构建，它就可以被优化然后变成门。 这个过程完全是机械的。 因此，我们不是在这里展示最后的步骤，而是在附录C的C.2节中描述过程和结果。

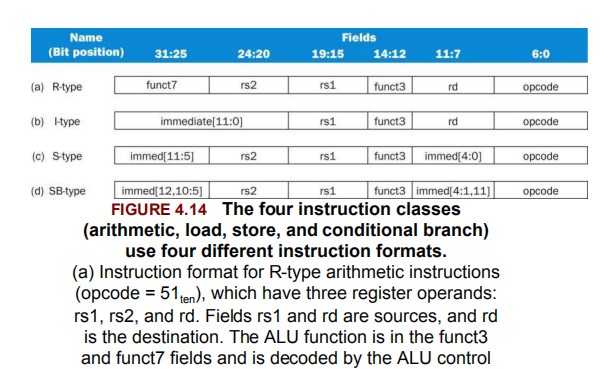


图4.14四个指令类（算术，加载，存储和条件分支）使用四种不同的指令格式。

（a）R型算术指令（opcode = 51ten）的指令格式，它有三个寄存器操作数：rs1，rs2和rd。字段rs1和rd是源，rd是目标。 ALU功能位于funct3和funct7字段中，并由上一节中的ALU控件设计解码。我们实现的R类型指令是add，sub，和，或。

（b）I型加载指令的指令格式（opcode = 3ten）。寄存器rs1是添加到12位立即数字段以形成存储器地址的基址寄存器。 Field rd是加载值的目标寄存器。

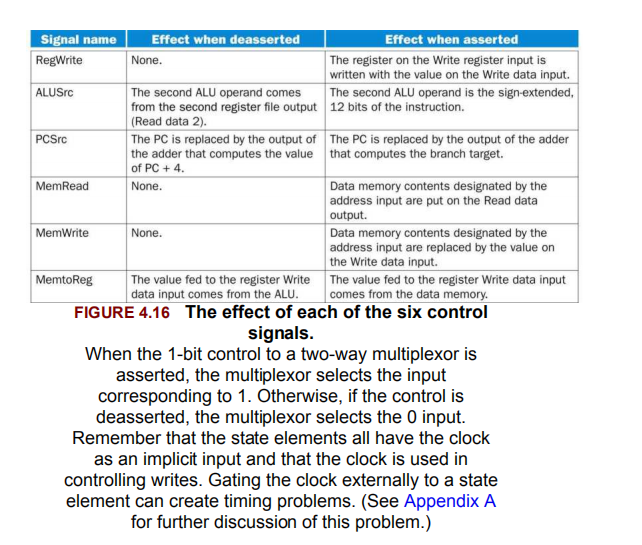
（c）S型存储指令的指令格式（操作码= 35ten）。寄存器rs1是添加到12位立即数字段以形成存储器地址的基址寄存器。 （立即数字段分为7位和5位。）字段rs2是源寄存器，其值应存储在存储器中。

（d）SB型条件分支指令的指令格式（opcode = 99ten）。寄存器rs1和rs2进行了比较。 12位立即数地址字段进行符号扩展，向左移位1位，并添加到PC以计算分支目标地址。

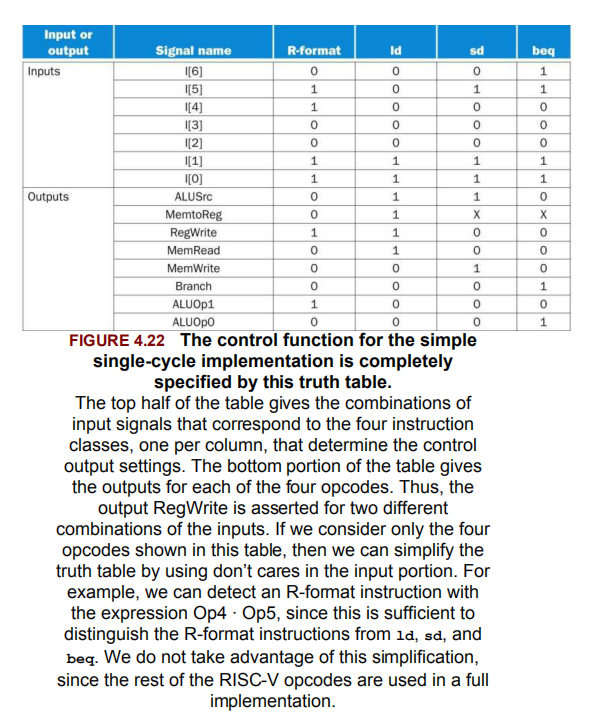
（page505）

正如我们在第2章中看到的那样，操作码字段总是位为6：0。根据操作码，funct3字段（位14:12）和funct7字段（位31:25）用作扩展操作码字段。对于R型指令和分支指令，第一个寄存器操作数始终位于位19:15（rs1）。该字段还指定加载和存储指令的基址寄存器。对于R型指令和分支指令，第二个寄存器操作数总是位于位24:20（rs2）。该字段还指定了将存储指令复制到存储器的寄存器操作数。另一个操作数也可以是分支或加载存储指令的12位偏移量。对于R型指令和加载指令，目标寄存器始终位于位11：7（rd）。第2章中的第一个设计原则 - 简单性有利于规律性 - 在指定控制时付出代价。使用此信息，我们可以将指令标签添加到简单数据路径中。图4.15显示了这些新增功能以及ALU控制模块，状态元素的写入信号，数据存储器的读取信号以及多路复用器的控制信号。由于所有多路复用器都有两个输入，因此每个输入都需要一条控制线。

（page507）



（page516，下面的译码中beq输入应该是1100111）



表的上半部分给出了与四个指令类相对应的输入信号组合，每列一个，用于确定控制输出设置。 表格的底部给出了四个操作码中每一个的输出。 因此，输出RegWrite被断言用于输入的两种不同组合。 如果我们只考虑此表中显示的四个操作码，那么我们可以通过在输入部分使用不关心来简化真值表。 例如，我们可以使用表达式Op4∙Op5检测R格式指令，因为这足以区分R格式指令与ld，sd和beq。 我们没有利用这种简化，因为其余的RISC-V操作码用于完整实现。

