

Example

Sketch a stick diagram for a CMOS gate computing $Y = \overline{(A + B + C)} \cdot \overline{D}$ (see Figure 1.18) and estimate the cell width and height.

Solution. Figure 1.47 shows a stick diagram. Counting horizontal and vertical pitches gives an estimated cell size of 40 by 48 λ .

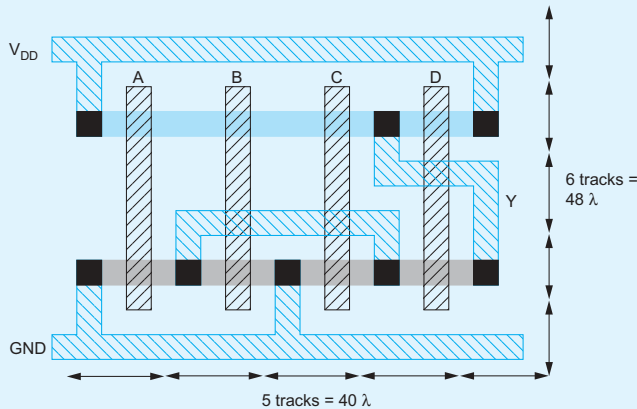


FIG 1.47 CMOS compound gate for function $Y = (A + B + C) \cdot D$

1.6 Design Partitioning

By this point, you know that MOS transistors behave as voltage-controlled switches. You know how to build logic gates out of transistors. And you know how transistors are fabricated and how to draw a layout that specifies how transistors should be placed and connected together. You know enough to start building your own simple chips.

The greatest challenge in modern VLSI design is not in designing the individual transistors but rather in managing system complexity. Modern *System-On-Chip* (SOC) designs combine memories, processors, high speed I/O interfaces, and dedicated application-specific logic on a single chip. They use hundreds of millions (soon billions) of transistors. The implementation must be divided among large teams of engineers and each engineer must be highly productive. If the implementation is too rigidly partitioned, each block can be optimized without regard to its neighbors, leading to poor system results. Conversely, if every task is interdependent with every other task, design will progress too

slowly. Design managers face the challenge of choosing a suitable tradeoff between these extremes. There is no substitute for practical experience in making these choices, and talented engineers who have experience with multiple designs are very important to the success of a large project. The notion of structured design, which is also used in large software projects, will be introduced in Chapter 8. Structured design uses the principles of hierarchy, regularity, modularity, and locality to manage the complexity.

Digital VLSI design is often partitioned into five interrelated tasks: *architecture* design, *microarchitecture* design, *logic* design, *circuit* design, and *physical* design. Architecture describes the functions of the system. For example, the x86 microprocessor architecture specifies the instruction set, register set, and memory model. Microarchitecture describes how the architecture is partitioned into registers and functional units. The 80386, 80486, Pentium, Pentium II, Pentium III, Pentium 4, Celeron, Cyrix MII, AMD K5, and Athlon are all microarchitectures offering different performance / transistor count tradeoffs for the x86 architecture. Logic describes how functional units are constructed. For example, various logic designs for a 32-bit adder in the x86 integer unit include ripple carry, carry lookahead, and carry select. Circuit design describes how transistors are used to implement the logic. For example, a carry lookahead adder can use static CMOS circuits, domino circuits, or pass transistors. The circuits can be tailored to emphasize high performance or low power. Physical design describes the layout of the chip.

These elements are inherently interdependent. For example, choices of microarchitecture and logic are strongly dependent on the number of transistors that can be placed on the chip, which depends on the physical design and process technology. Similarly, innovative circuit design that reduces a cache access from two cycles to one can influence which microarchitecture is most desirable. The choice of clock frequency depends on a complex interplay of microarchitecture and logic, circuit design, and physical design. Deeper pipelines allow higher frequencies but lead to greater performance penalties when operations early in the pipeline are dependent on those late in the pipeline. Many functions have various logic and circuit designs trading speed for area, power, and design effort. Custom physical design allows more compact, faster circuits and lower manufacturing costs, but involves an enormous labor cost. Automatic layout with CAD systems reduces the labor and achieves faster times to market.

To deal with these interdependencies, microarchitecture, logic, circuit, and physical design must occur, at least in part, in parallel. Microarchitects depend on circuit and physical design studies to understand the cost of proposed microarchitectural features. Engineers are sometimes categorized as “short and fat” or “tall and skinny.” Tall, skinny engineers understand something about a broad range of topics. Short, fat engineers understand a large amount about a narrow field. Digital VLSI design favors the tall, skinny engineer who can evaluate how choices in one part of the system impact other parts of the system.

A critical tool for managing complex designs is *hierarchy*. A large system can be partitioned into many *units*. Each unit in turn is composed of multiple *functional blocks*¹. These blocks in turn are built from *cells*, which ultimately are constructed from transistors. The

¹Some designers refer to both units and functional blocks as *modules*.

system can be more easily understood at the top level by viewing units as black boxes with well-defined interfaces and functions rather than looking at each individual transistor. Hierarchy also facilitates design reuse; a block can be designed and verified once, then used in many places. Logic, circuit, and physical views of the design should share the same hierarchy for ease of verification. A design hierarchy can be viewed as a tree structure with the overall chip as the *root* and the primitive cells as *leafs*.

An alternative way of viewing design partitioning is shown with the Y-chart in Figure 1.48 [Gajski83, Kang03]. The radial lines on the Y-chart represent three distinct design domains: behavioral, structural, and physical. These domains can be used to describe the design of almost any artifact and thus form a very general taxonomy for describing the design process. Within each domain there are a number of levels of design abstraction that start at a very high level and descend eventually to the individual elements that need to be aggregated to yield the top level function (i.e., in the case of chip design and transistors).

The behavioral domain describes what a particular system does. For instance, at the highest level we might state that we desire to build a chip that can generate audio tones of specific frequencies (i.e., a touch-tone generator for a telephone). This behavior can be successively refined to more precisely describe what needs to be done in order to build the tone generator (i.e., the frequencies desired, output levels, distortion allowed, etc.).

At each abstraction level, a corresponding structural description can be described. The structural domain describes the interconnection of modules necessary to achieve a particular behavior. For instance, at the highest level, the touch-tone generator might consist of a keyboard, a tone generator, an audio amplifier, a battery, and a speaker. Eventually at lower levels of abstraction, the individual gate and then transistor connections required to build the tone generator are described.

For each level of abstraction, the physical domain description explains how to physically construct that level of abstraction. At high levels this might consist of an engineering drawing showing how to put together the keyboard, tone generator chip, battery, and speaker in the associated housing. At the top chip level, this might consist of a floorplan, and at lower levels, the actual geometry of individual transistors.

The design process can be viewed as making transformations from one domain to another while maintaining the equivalency of the domains. Behavioral descriptions are transformed to structural descriptions, which in turn are transformed to physical descriptions. These transformations can be manual or automatic. In either case, it is normal design practice to verify the transformation of one domain to the other by some checking process. This ensures that the design intent is carried across the domain boundaries. Hierarchically specifying each domain at successively detailed levels of abstraction allows us to design very large systems.

The reason for strictly describing the domains and levels of abstraction is to define a precise design process in which the final function of the system can be traced all the way back to the initial behavioral description. There should be no opportunity to produce an incorrect design. If anomalies arise, the design process is corrected so that those anomalies will not reoccur in the future. A designer should acquire a rigid discipline with respect to the design process, and be aware of each transformation and how and why it is failproof.

Normally, these steps are fully automated in a modern design process, but it is important to be aware of the basis for these steps in order to debug them if they go astray.

The Y diagram can be used to illustrate each domain and the transformations between domains at varying levels of design abstraction. As the design process winds its way from the outer to inner rings, it proceeds from higher to lower levels of abstraction and hierarchy.

Most of the remainder of this chapter is a case study in the design of a simple microprocessor to illustrate the various aspects of VLSI design applied to a nontrivial system. We begin by describing the architecture and microarchitecture of the processor. We then consider logic design and discuss hardware description languages. The processor is built with static CMOS circuits, which have been examined in Section 1.4 already; transistor level design and netlist formats are discussed. We continue exploring the physical design of the processor including floorplanning and area estimation. Design verification is very important and happens at each level of the hierarchy for each element of the design. Finally, the layout is converted into masks so the chip can be manufactured, packaged, and tested.

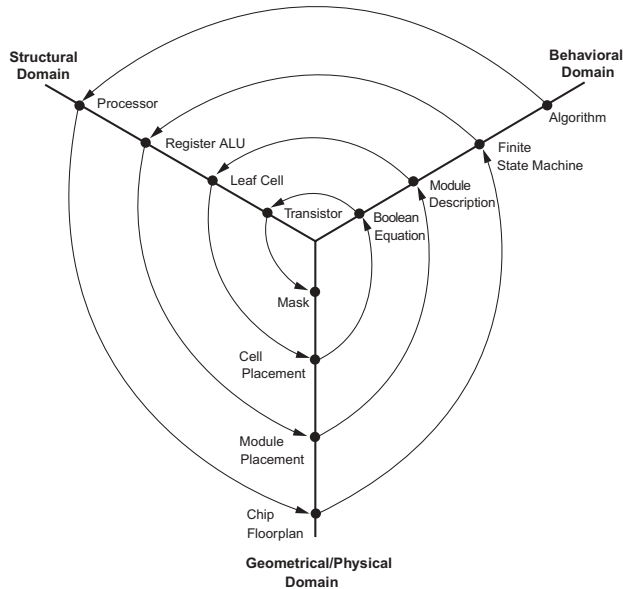


FIG 1.48 Y Diagram

1.7 Example: A Simple MIPS Microprocessor

We consider an 8-bit subset of the Patterson & Hennessy MIPS microprocessor architecture [Patterson04] because it is widely studied and is relatively simple, while still being large enough to illustrate hierarchical design. This section describes the architecture and the multicycle microarchitecture we will be implementing. If you are not familiar with computer architecture, you can regard the MIPS processor as a black box and skip to Section 1.8.

A set of laboratory exercises are available online in which you can learn VLSI design by building the microprocessor yourself using a free open-source CAD tool called *Electric*.

1.7.1 MIPS Architecture

The MIPS32 architecture is a simple 32-bit RISC architecture with relatively few idiosyncrasies. Our subset of the architecture uses 32-bit instruction encodings but only eight 8-bit general-purpose registers named \$0–\$7. We also use an 8-bit program counter (PC). Register \$0 is hardwired to contain the number 0. The instructions are ADD, SUB, AND, OR, SLT, ADDI, BEQ, J, LB, and SB.

The function and encoding of each instruction is given in Table 1.7. Each instruction is encoded using one of three templates: R, I, and J. R-type instructions (*register*-based) are used for arithmetic and specify two source registers and a destination register. I-type

Table 1.7 MIPS instruction set (subset supported)

Instruction	Function	Encoding	op	funct
add \$1, \$2, \$3	addition: \$1 → \$2 + \$3	R	000000	100000
sub \$1, \$2, \$3	subtraction: \$1 → \$2 – \$3	R	000000	100010
and \$1, \$2, \$3	bitwise and: \$1 → \$2 and \$3	R	000000	100100
or \$1, \$2, \$3	bitwise or: \$1 → \$2 or \$3	R	000000	100101
slt \$1, \$2, \$3	set less than: \$1 → 1 if \$2 < \$3 \$1 → 0 otherwise	R	000000	101010
addi \$1, \$2, imm	add immediate: \$1 → \$2 + imm	I	001000	n/a
beq \$1, \$2, imm	branch if equal: PC → PC + imm ^a	I	000100	n/a
j destination	jump: PC_destination ^a	J	000010	n/a
lb \$1, imm(\$2)	load byte: \$1 → mem[\$2 + imm]	I	100000	n/a
sb \$1, imm(\$2)	store byte: mem[\$2 + imm] → \$1	I	110000	n/a

a. Technically, MIPS addresses specify bytes. Instructions require a four-byte word and must begin at addresses that are a multiple of four. To most effectively use instruction bits in the full 32-bit MIPS architecture, branch and jump constants are specified in words and must be multiplied by four (shifted left two bits) to be converted to byte addresses.

instructions are used when a 16-bit constant (also known as an *immediate*) and two registers must be specified. J-type instructions (*jumps*) dedicate most of the instruction word to a 26-bit jump destination. The format of each encoding is defined in Figure 1.49. The six most significant bits of all formats are the operation code (*op*). R-type instructions all share *op* = 000000 and use six more *funct* bits to differentiate the functions.

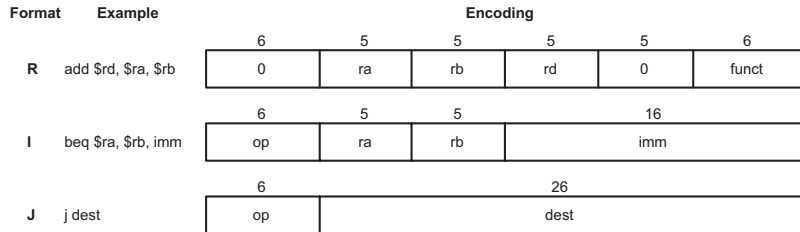


FIG 1.49 Instruction encoding formats

We can write programs for the MIPS processor in *assembly language*, where each line of the program contains one instruction such as **ADD** or **BEQ**. However, the MIPS hardware ultimately must read the program as a series of 32-bit numbers called *machine language*. An *assembler* automates the tedious process of translating from assembly language to machine language using the encodings defined in Table 1.7 and Figure 1.49. Writing nontrivial programs in assembly language is also tedious, so programmers usually work in a *high-level language* such as C or FORTRAN. A *compiler* translates a program from high-level language *source code* into the appropriate machine language *object code*.

Example

Figure 1.50 shows a simple C program that computes the n th Fibonacci number f_n defined recursively for $n > 0$ as $f_n = f_{n-1} + f_{n-2}$, $f_{-1} = -1$, $f_0 = 1$. Translate the program into MIPS assembly language and machine language.

Solution: Figure 1.51 gives a commented assembly language program. Figure 1.52 translates the assembly language to machine language.

continues

```

int fib(void)
{
    int n = 8;          /* compute nth Fibonacci number */
    int f1 = 1, f2 = -1; /* last two Fibonacci numbers */

    while (n != 0) {   /* count down to n = 0 */
        f1 = f1 + f2;
        f2 = f1 - f2;
        n = n - 1;
    }
    return f1;
}

```

FIG 1.50 C code for Fibonacci program

```

# fib.asm
# Register usage: $3: n $4: f1 $5: f2
# return value written to address 255
fib:  addi $3, $0, 8      # initialize n=8
      addi $4, $0, 1     # initialize f1 = 1
      addi $5, $0, -1    # initialize f2 = -1
loop: beq $3, $0, end    # Done with loop if n = 0
      add $4, $4, $5     # f1 = f1 + f2
      sub $5, $4, $5     # f2 = f1 - f2
      addi $3, $3, -1    # n = n - 1
      j loop             # repeat until done
end:  sb $4, 255($0)     # store result in address 255

```

FIG 1.51 Assembly language code for Fibonacci program

Instruction	Binary Encoding	Hexadecimal Encoding
addi \$3, \$0, 8	001000 00000 00011 0000000000001000	20030008
addi \$4, \$0, 1	001000 00000 00100 0000000000000001	20040001
addi \$5, \$0, -1	001000 00000 00101 1111111111111111	2005ffff
beq \$3, \$0, end	000100 00011 00000 00000000000000101	10600005
add \$4, \$4, \$5	000000 00100 00101 00100 00000 100000	00852020
sub \$5, \$4, \$5	000000 00100 00101 00101 00000 100010	00852822
addi \$3, \$3, -1	001000 00011 00011 1111111111111111	2063ffff
j loop	000010 000000000000000000000000000011	08000003
sb \$4, 255(\$0)	110000 00000 00100 0000000011111111	a00400ff

FIG 1.52 Machine language code for Fibonacci program

1.7.2 Multicycle MIPS Microarchitecture

We will implement the multicycle MIPS microarchitecture given in Chapter 5 of [Patterson98, Patterson04] modified to process 8-bit data. The microarchitecture is illustrated in Figure 1.53. The rectangles represent registers or memory. The rounded rectangles represent multiplexers. The ovals represent control logic. Light lines indicate individual signals while heavy lines indicate busses. The control logic and signals are highlighted in blue while the datapath is shown in black. Control signals generally drive multiplexer select signals and register enables to tell the datapath how to execute an instruction.

Instruction execution generally flows from left to right. The program counter (PC) specifies the address of the instruction. The instruction is loaded one byte at a time over four cycles from an off-chip memory into the 32-bit instruction register (IR). The `op` field (bits 31:26 of the instruction) is sent to the controller, which sequences the datapath through the correct operations to execute the instruction. For example, in an `ADD` instruction, the two source registers are read from the register file into temporary registers `a` and `b`. On the next cycle, the `alucontrol` unit commands the Arithmetic/Logic Unit (ALU) to add the inputs. The result is captured in the `aluout` register. On the third cycle, the result is written back to the appropriate destination register in the register file.

The controller is a finite state machine that generates multiplexer select signals and register enables to sequence the datapath. A state transition diagram for the FSM is shown in Figure 1.54. As discussed, the first four states fetch the instruction from memory. The FSM then is dispatched based on `op` to execute the particular instruction. The FSM states for `ADDI` are missing and left as an exercise for the reader.

Observe that the controller produces a 2-bit `aluop` output. The `alucontrol` unit uses combinational logic to compute a 3-bit `alucontrol` signal from the `aluop` and `funct` fields, as specified in Table 1.8. `alucontrol` drives multiplexers in the ALU to select the appropriate computation.

Table 1.8 ALUControl determination			
<code>aluop</code>	<code>funct</code>	<code>alucontrol</code>	Meaning
00	x	010	ADD
01	x	110	SUB
10	100000	010	ADD
10	100010	110	SUB
10	100100	000	AND
10	100101	001	OR
10	101010	111	SLT
11	x	x	undefined

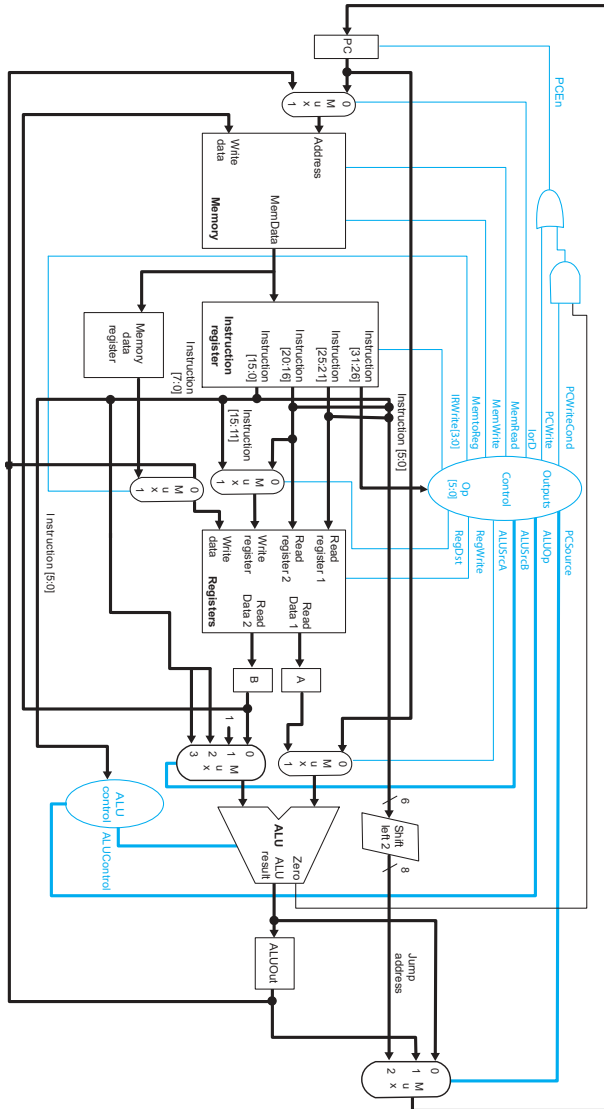


FIG 1.53 Multicycle MIPS microarchitecture

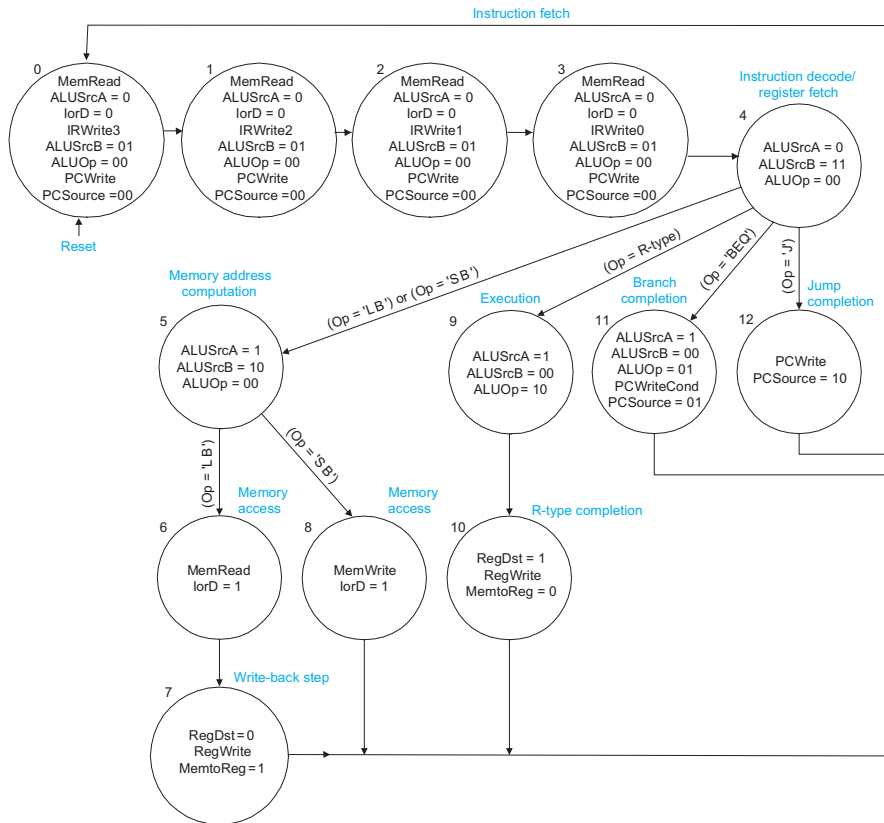


FIG 1.54 Multicycle MIPS control FSM

Example

Referring to Figure 1.53 and Figure 1.54, explain how the MIPS processor fetches and executes the **SUB** instruction.

Solution: The first step is to fetch the 32-bit instruction. This takes four cycles because the instruction must come over an 8-bit memory interface. On each cycle, we want to fetch a byte from the address in memory specified by the program counter, then increment the program counter by one to point to the next byte.

The fetch is performed by states 0–3 of the FSM in Figure 1.54. Let us start with state 0. The program counter (PC) contains the address of the first byte of the instruction. The controller must select $iord = 0$ so that the multiplexer sends this address to the memory. $memread$ must also be asserted so the memory reads the byte onto the $memdata$ bus. Finally, $irwrite3$ should be asserted to enable writing $memdata$ into the most significant byte of the instruction register (IR).

Meanwhile, we need to increment the program counter. We can do this with the ALU by specifying PC as one input, '1' as the other input, and ADD as the operation. To select PC as the first input, $alusrca = 0$. To select '1' as the other input, $alusrcb = 01$. To perform an addition, $aluop = 00$, according to Table 1.8. To write this result back into the program counter at the end of the cycle, $pcsource = 00$ and $pcen = 1$ (done by setting $pcwritecond = 1$).

All of these control signals are indicated in state 0 of Figure 1.54. The other register enables are assumed to be 0 if not explicitly asserted and the other multiplexer selects are don't cares. The next three states are identical except that they write bytes 2, 1, and 0 of the IR, respectively.

The next step is to read the source registers, done in state 4. The two source registers are specified in bits 25:21 and 20:16 of the IR. The register file reads these registers and puts the values into the A and B registers. No control signals are necessary for **SUB** (although state 4 performs a branch address computation in case the instruction is **BEQ**).

The next step is to perform the subtraction. Based on the op field (IR bits 31:26), the FSM jumps to state 9 because **SUB** is an R-type instruction. The two source registers are selected as input to the ALU by setting $alusrca = 1$ and $alusrcb = 00$. Choosing $aluop = 10$ directs the ALU Control decoder to select the $alucontrol$ signal as 110, subtraction. Other R-type instructions are executed identically except that the decoder receives a different funct code (IR bits 5:0) and thus generates a different $alucontrol$ signal. The result is placed in the ALUOut register.

Finally, the result must be written back to the register file in state 10. The data comes from the ALUOut register so $memtoreg = 0$. The destination register is specified in bits 15:11 of the instruction so $regdst = 1$. $regwrite$ must be asserted to perform the write. Then the control FSM returns to state 0 to fetch the next instruction.