Figure A.2.1 Standard drawing for an AND gate, OR gate, and an inverter, shown from left to right. The signals to the left of each symbol are the inputs, while the output appears on the right. The AND and OR gates both have two inputs. Inverters have a single input.

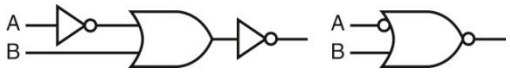
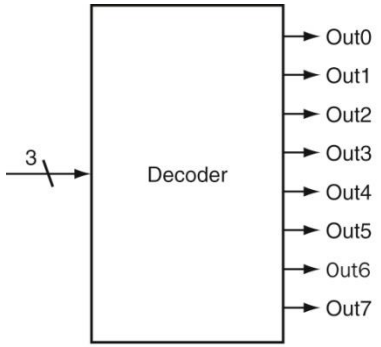


Figure A.2.2 Logic gate implementation of *A + B using explicit inverts on the left and* bubbled inputs and outputs on the right. This logic function can be simplified to *A B or in Verilog*, A & ~ B.



a. A 3-bit decoder

Inputs			Outputs							
12	11	10	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

b. The truth table for a 3-bit decoder

Figure A.3.1 A 3-bit decoder has three inputs, called 12, 11, and 10, and 23 = 8 outputs, called Out0 to Out7. Only the output corresponding to the binary value of the input is true, as shown in the truth table. The label 3 on the input to the decoder says that the input signal is 3 bits wide.

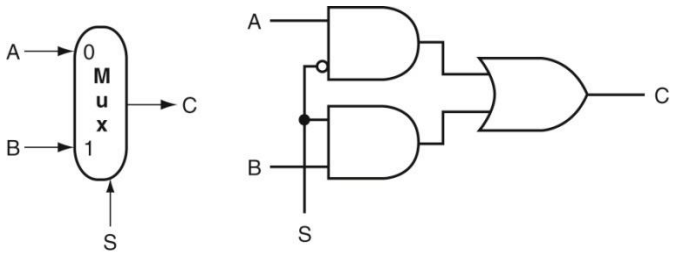


Figure A.3.2 A two-input multiplexor on the left and its implementation with gates on the right. The multiplexor has two data inputs (*A and B), which are labeled 0 and 1, and one selector input* (*S), as well as an output C. Implementing multiplexors in Verilog requires a little more work, especially when* they are wider than two inputs. We show how to do this beginning on page A-23.

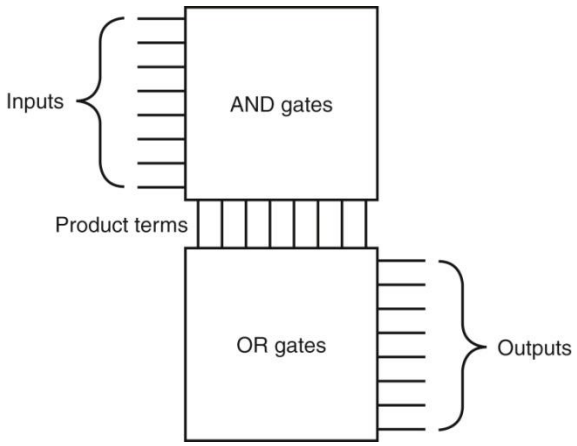


Figure A.3.3 The basic form of a PLA consists of an array of AND gates followed by an array of OR gates. Each entry in the AND gate array is a product term consisting of any number of inputs or inverted inputs. Each entry in the OR gate array is a sum term consisting of any number of these product terms.

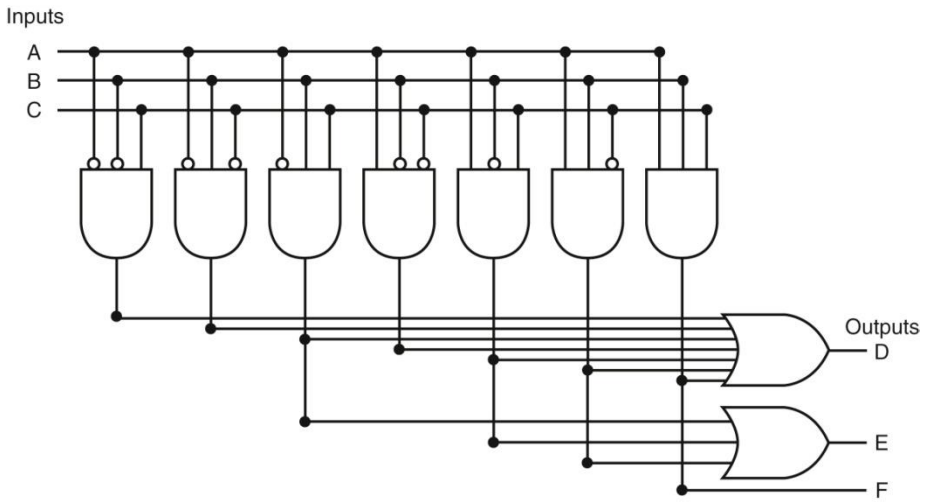


Figure A.3.4 The PLA for implementing the logic function described in the example.

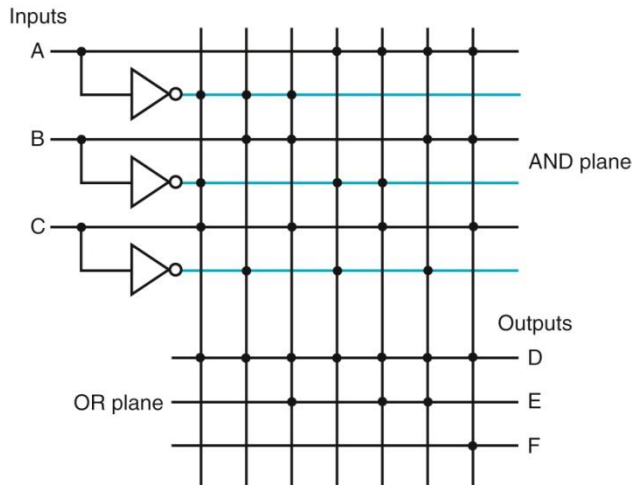
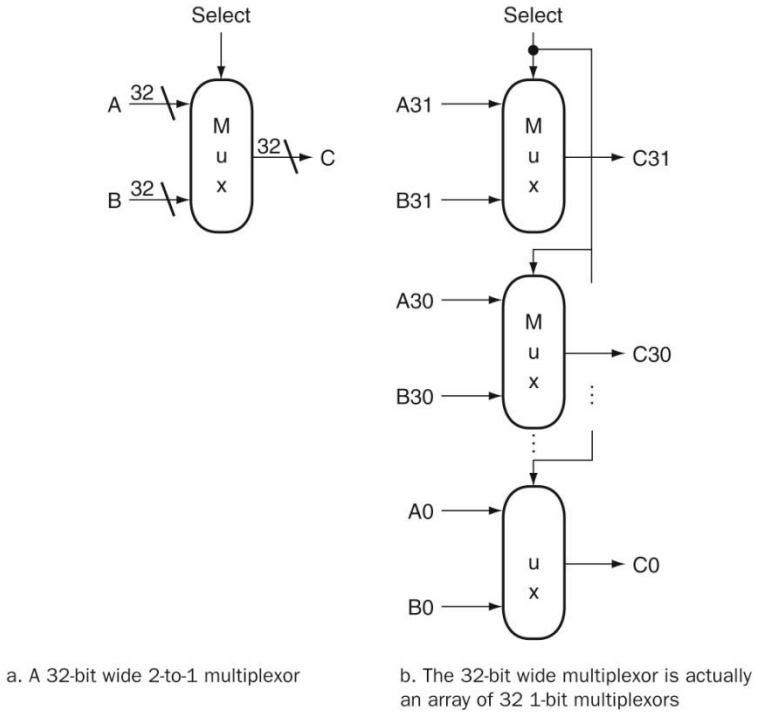


Figure A.3.5 A PLA drawn using dots to indicate the components of the product terms and sum terms in the array. Rather than use inverters on the gates, usually all the inputs are run the width of the AND plane in both true and complement forms. A dot in the AND plane indicates that the input, or its inverse, occurs in the product term. A dot in the OR plane indicates that the corresponding product term appears in the corresponding output.



a. A 32-bit wide 2-to-1 multiplexor

b. The 32-bit wide multiplexor is actually an array of 32 1-bit multiplexors

Figure A.3.6 A multiplexor is arrayed 64 times to perform a selection between two 64-bit inputs. Note that there is still only one data selection signal used for all 32 1-bit multiplexors.

```
module half_adder (A,B,Sum,Carry);
   input A,B; //two 1-bit inputs
   output Sum, Carry; //two 1-bit outputs
   assign Sum = A ^ B; //sum is A xor B
   assign Carry = A & B; //Carry is A and B
endmodule
```

Figure A.4.1 A Verilog module that defines a half-adder using continuous assignments.

```
module Mult4to1 (In1,In2,In3,In4,Sel,Out);
   input [31:0] In1, In2, In3, In4; //four 32-bit inputs
   input [1:0] Sel; //selector signal
   output reg [31:0] Out; //32-bit output
   always @(In1, In2, In3, In4, Sel)
   case (Sel) // a 4->1 multiplexor
      0: Out <= In1;
      1: Out <= In2;
      2: Out <= In3;
      default: Out <= In4;
   endcase
endmodule
```

Figure A.4.2 A Verilog definition of a 4-to-1 multiplexor with 32-bit inputs, using a case **statement.** The case statement acts like a C switch statement, except that in Verilog only the code associated with the selected case is executed (as if each case state had a break at the end) and there is no fall-through to the next statement.

```
module RISCVALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [31:0] A,B;
   output reg [31:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0; goes anywhere
   always @(ALUctl, A, B) //reevaluate if these change
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1:0;
         12: ALUOut <= ~(A | B); // result is nor
         default: ALUOut <= 0; //default to 0, should not happen;
      endcase
endmodule
```

Figure A.4.3 A Verilog behavioral definition of a RISC-V ALU. This could be synthesized using a module library containing basic arithmetic and logical operations.

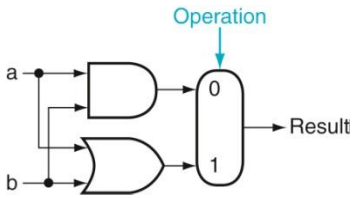


Figure A.5.1 The 1-bit logical unit for AND and OR.

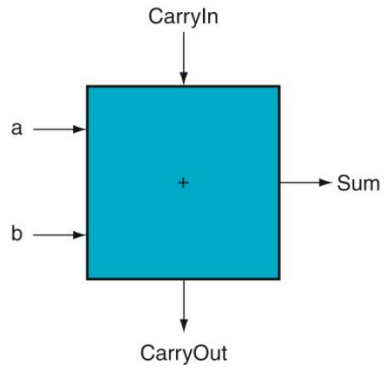


Figure A.5.2 A 1-bit adder. This adder is called a full adder; it is also called a (3,2) adder because it has three inputs and two outputs. An adder with only the a and b inputs is called a (2,2) adder or half-adder.

Inputs			Outputs		
a	b	CarryIn	CarryOut	Sum	Comments
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

Figure A.5.3 Input and output specification for a 1-bit adder.

Inputs		
a	b	CarryIn
0	1	1
1	0	1
1	1	0
1	1	1

Figure A.5.4 Values of the inputs when CarryOut is a 1.

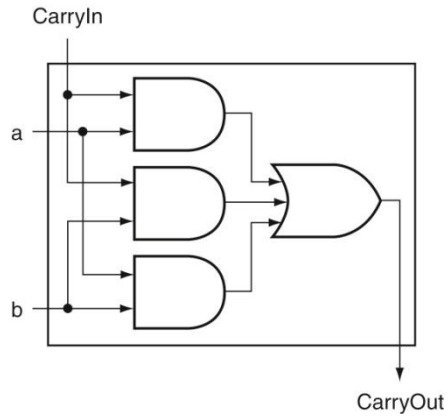


Figure A.5.5 Adder hardware for the CarryOut signal. The rest of the adder hardware is the logic for the Sum output given in the equation on this page.

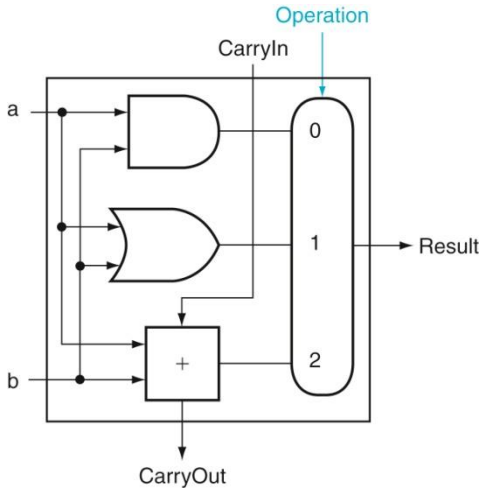


Figure A.5.6 A 1-bit ALU that performs AND, OR, and addition (see Figure A.5.5).

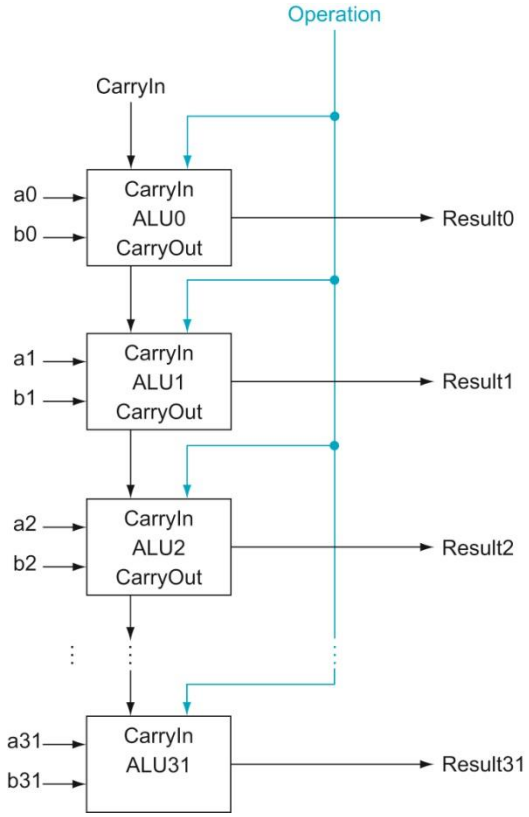


Figure A.5.7 A 32-bit ALU constructed from 32 1-bit ALUs. CarryOut of the less significant bit is connected to the CarryIn of the more significant bit. This organization is called ripple carry.

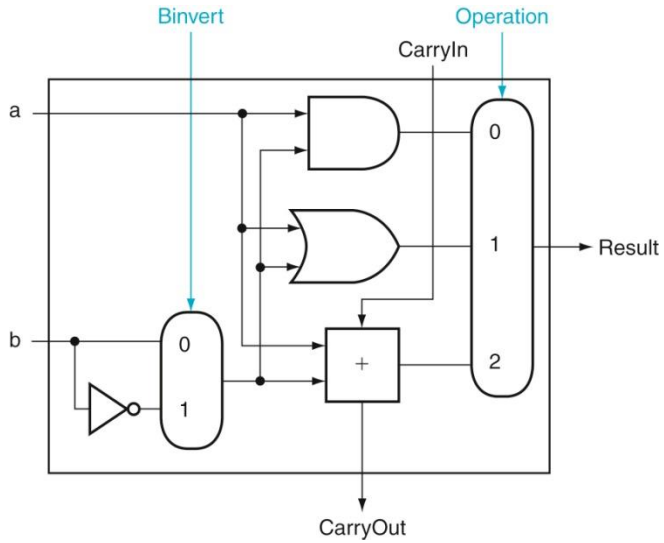


Figure A.5.8 A 1-bit ALU that performs AND, OR, and addition on a and b or a and b. By selecting b (Binvert = 1) and setting CarryIn to 1 in the least significant bit of the ALU, we get two's complement subtraction of b from a instead of addition of b to a.

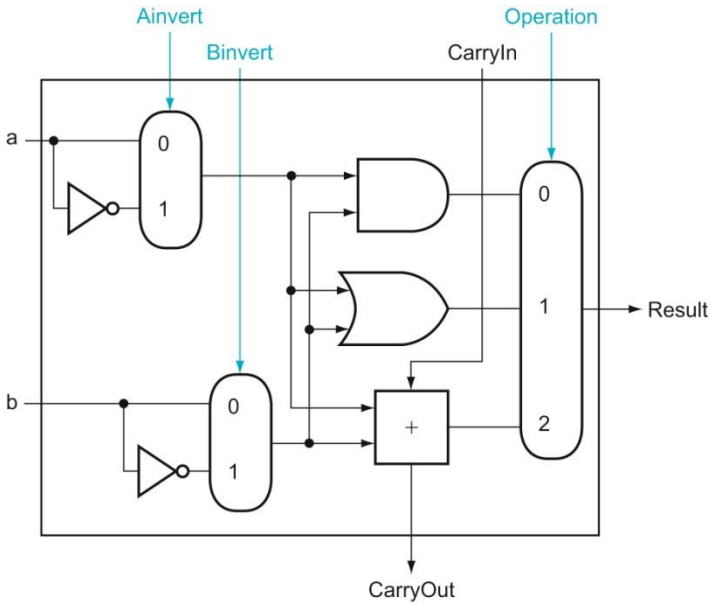


Figure A.5.9 A 1-bit ALU that performs AND, OR, and addition on a and b or a and b. By selecting a (Ainvert = 1) and b (Binvert = 1), we get a NOR b instead of a AND b.

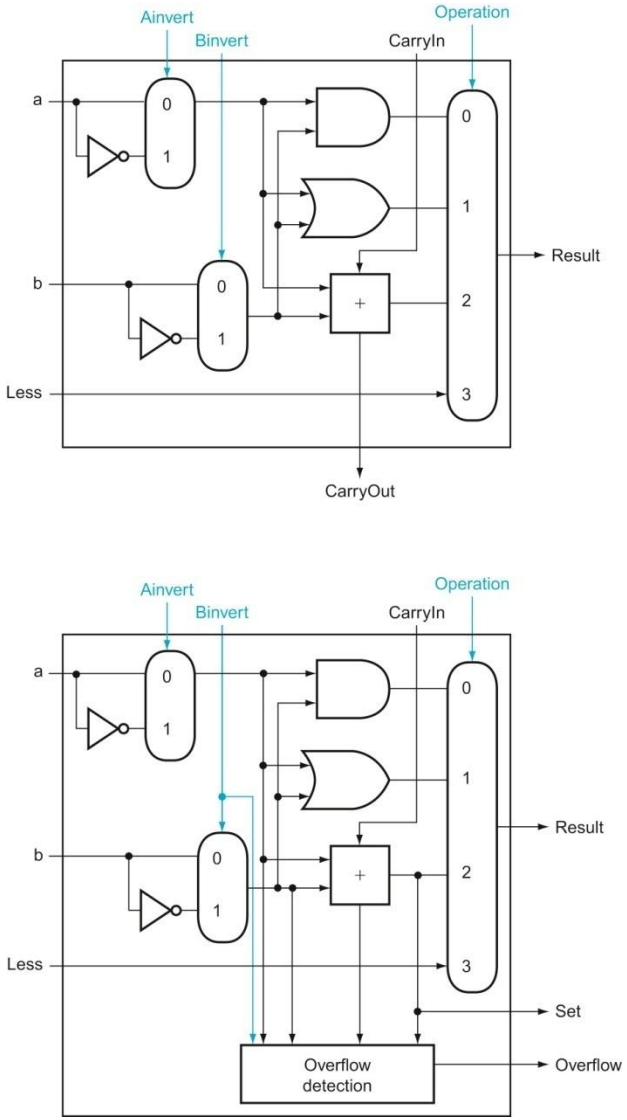


Figure A.5.10 (Top) A 1-bit ALU that performs AND, OR, and addition on a and b or b, and (bottom) a 1-bit ALU for the most significant bit. The top drawing includes a direct input that is connected to perform the set on less than operation (see Figure A.5.11); the bottom has a direct output from the adder for the less than comparison called Set. (See Exercise A.24 at the end of this appendix to see how to calculate overflow with fewer inputs.)

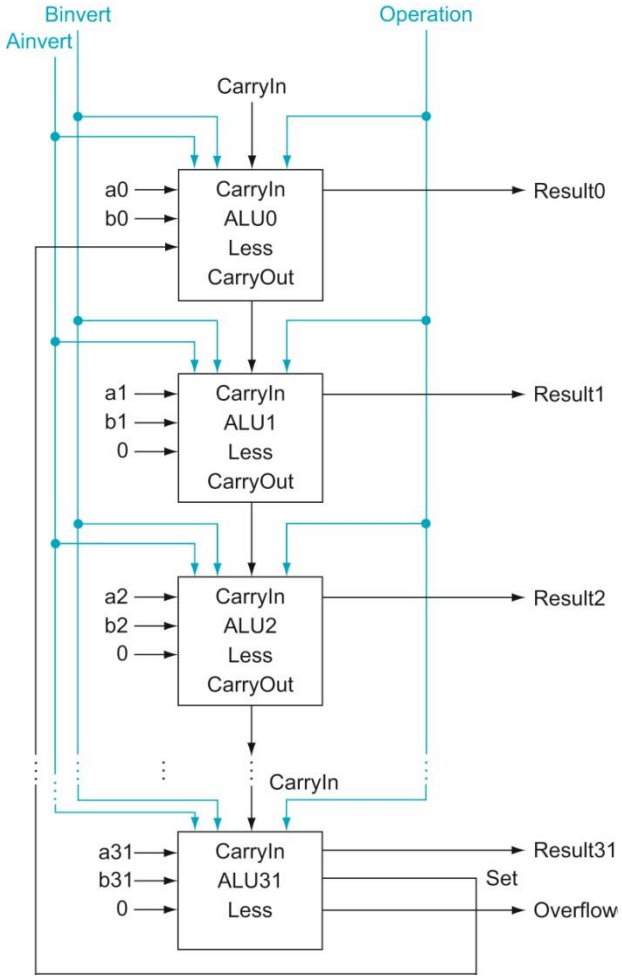


Figure A.5.11 A 32-bit ALU constructed from the 31 copies of the 1-bit ALU in the top of Figure A.5.10 and one 1-bit ALU in the bottom of that figure. The Less inputs are connected to 0 except for the least significant bit, which is connected to the Set output of the most significant bit. If the ALU performs a − b and we select the input 3 in the multiplexor in Figure A.5.10, then Result = 0 … 001 if a < b, and Result = 0 … 000 otherwise.

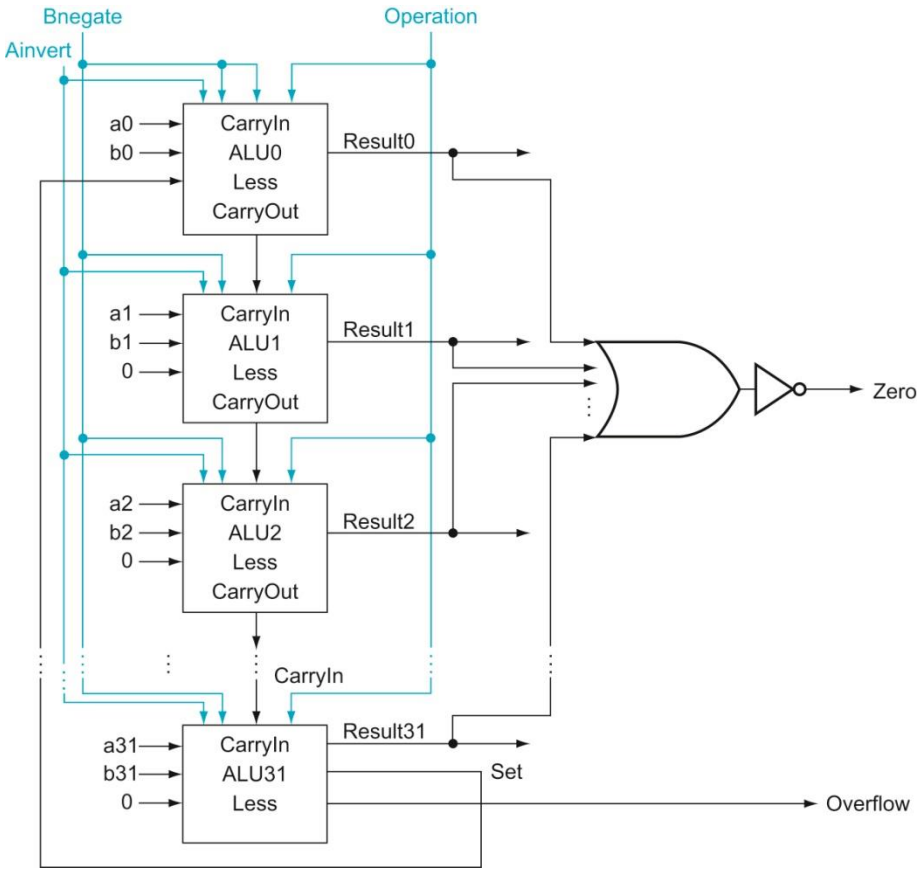


Figure A.5.12 The final 32-bit ALU. This adds a Zero detector to Figure A.5.11.

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

Figure A.5.13 The values of the three ALU control lines, Ainvert, Bnegate, and Operation, and the corresponding ALU operations.

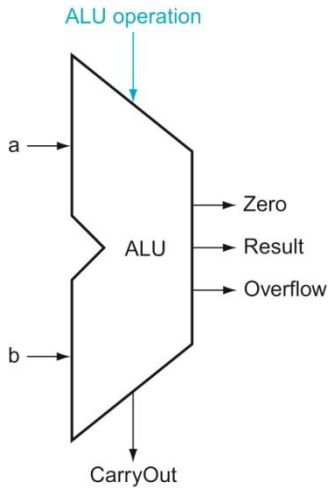


Figure A.5.14 The symbol commonly used to represent an ALU, as shown in Figure A.5.12. This symbol is also used to represent an adder, so it is normally labeled either with ALU or Adder..

```
module RISCVALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [31:0] A,B;
   output reg [31:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0
   always @(ALUctl, A, B) begin //reevaluate if these change
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1 : 0;
         12: ALUOut <= ~(A | B); // result is nor
         default: ALUOut <= 0;
      endcase
    end
endmodule
```

Figure A.5.15 A Verilog behavioral definition of a RISC-V ALU.

```
module ALUControl (ALUOp, FuncCode, ALUCtl);
   input [1:0] ALUOp;
   input [5:0] FuncCode;
   output [3:0] reg ALUCtl;
   always case (FuncCode)
   32: ALUOp<=2; // add
   34: ALUOp<=6; // subtract
   36: ALUOP<=0; // and
   37: ALUOp<=1; // or
   default: ALUOp<=15; // should not happen
   endcase
endmodule
```

Figure A.5.16 The RISC-V ALU control: a simple piece of combinational control logic.

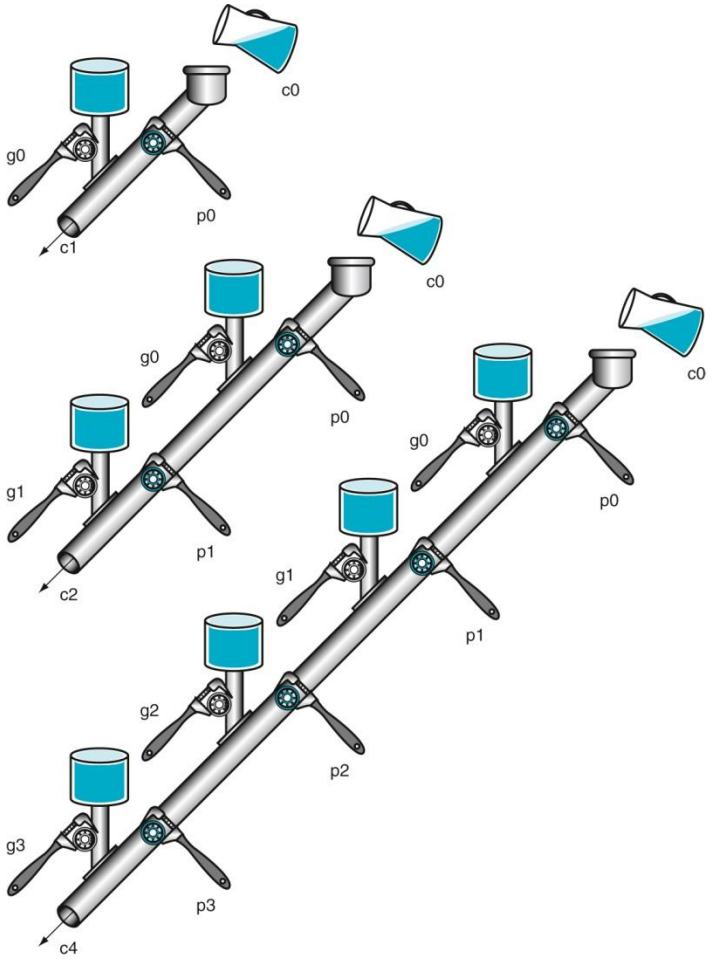


Figure A.6.1 A plumbing analogy for carry lookahead for 1 bit, 2 bits, and 4 bits using water pipes and valves. The wrenches are turned to open and close valves. Water is shown in color. The output of the pipe (c*i + 1) will be full if either the nearest generate value (gi) is turned on or if the i propagate* value (p*i) is on and there is water further upstream, either from an earlier generate or a propagate with water* behind it. CarryIn (c0) can result in a carry out without the help of any generates, but with the help of *all* propagates.

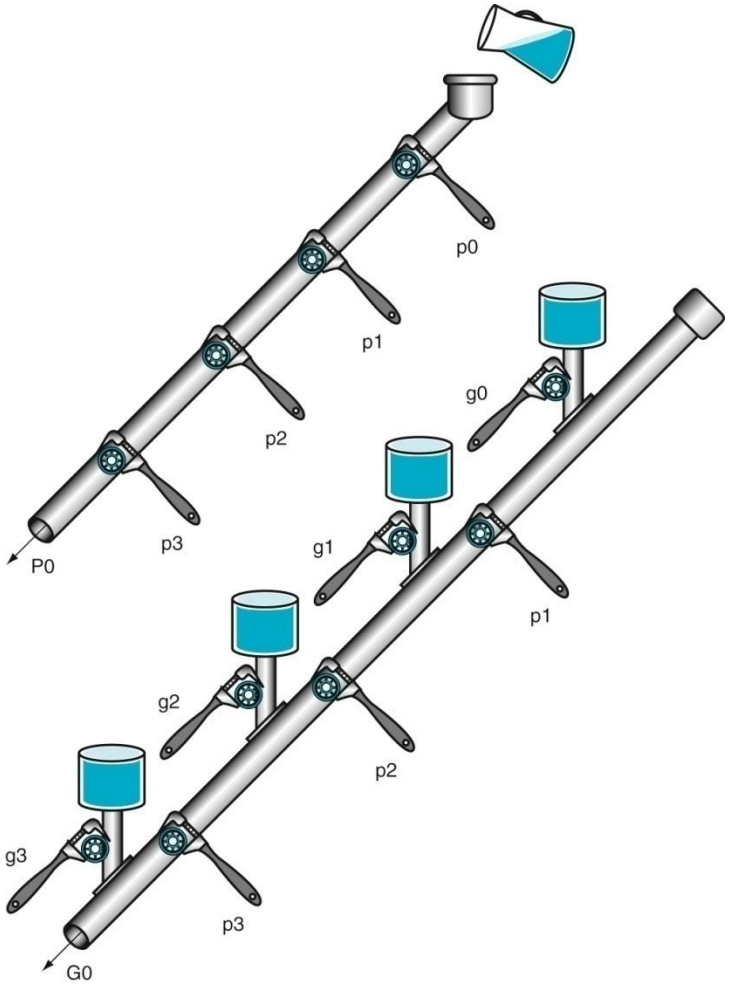


Figure A.6.2 A plumbing analogy for the next-level carry-lookahead signals P0 and G0.
P0 is open only if all four propagates (p*i) are open, while water flows in G0 only if at least one generate (gi) is* open and all the propagates downstream from that generate are open.

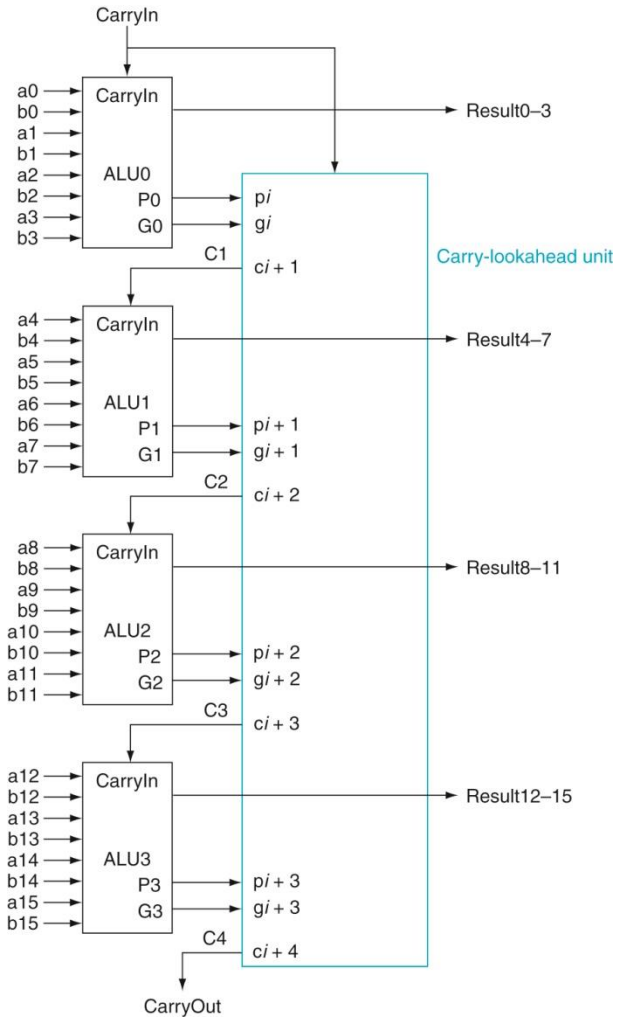


Figure A.6.3 Four 4-bit ALUs using carry lookahead to form a 16-bit adder. Note that the carries come from the carry-lookahead unit, not from the 4-bit ALUs.

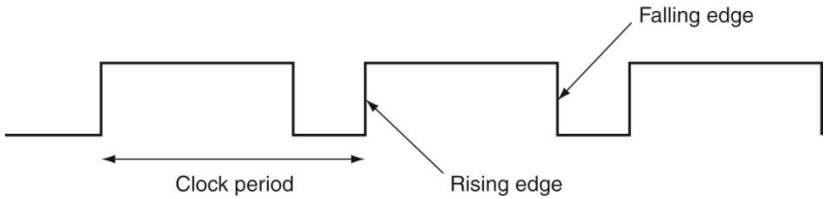


Figure A.7.1 A clock signal oscillates between high and low values. The clock period is the time for one full cycle. In an edge-triggered design, either the rising or falling edge of the clock is active and causes state to be changed.

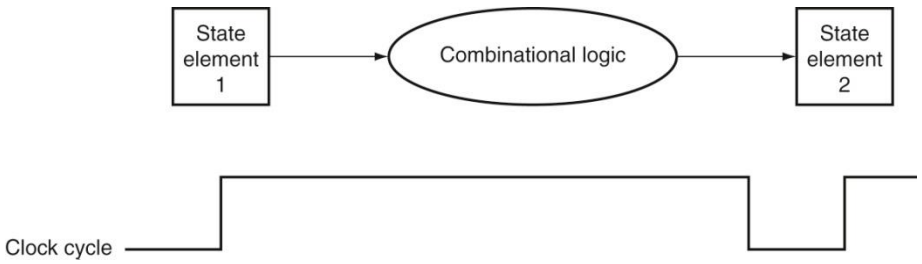


Figure A.7.2 The inputs to a combinational logic block come from a state element, and the outputs are written into a state element. The clock edge determines when the contents of the state elements are updated.

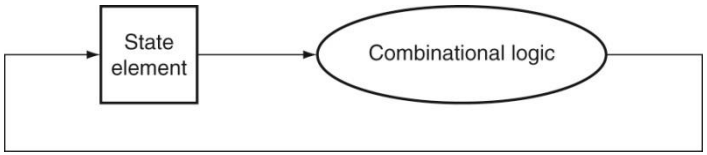


Figure A.7.3 An edge-triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could lead to undetermined data values. Of course, the clock cycle must still be long enough so that the input values are stable when the active clock edge occurs.

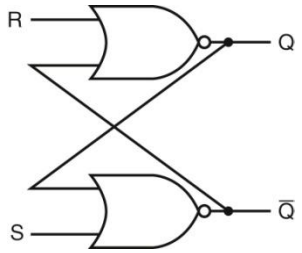


Figure A.8.1 A pair of cross-coupled NOR gates can store an internal value. The value stored on the output *Q is recycled by inverting it to obtain Q and then inverting Q to obtain Q. If either R or Q is asserted, Q will be deasserted and vice versa.*

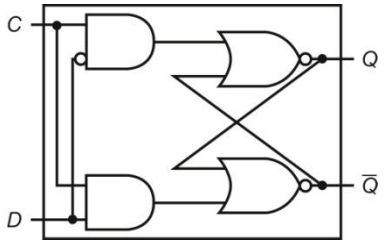


Figure A.8.2 A D latch implemented with NOR gates. A NOR gate acts as an inverter if the other input is 0. Thus, the cross-coupled pair of NOR gates acts to store the state value unless the clock input, *C, is* asserted, in which case the value of input *D replaces the value of Q and is stored. The value of input D must* be stable when the clock signal *C changes from asserted to deasserted.*

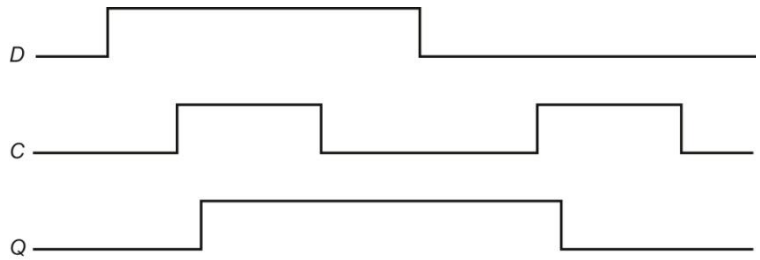


Figure A.8.3 Operation of a D latch, assuming the output is initially deasserted. When the clock, *C, is asserted, the latch is open and the Q output immediately assumes the value of the D input.*

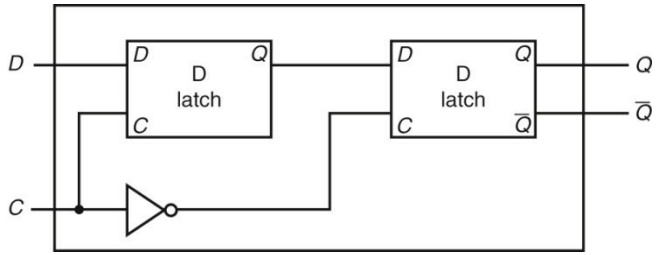


Figure A.8.4 A D flip-flop with a falling-edge trigger. The first latch, called the master, is open and follows the input *D when the clock input, C, is asserted. When the clock input, C, falls, the first latch is closed, but* the second latch, called the slave, is open and gets its input from the output of the master latch.

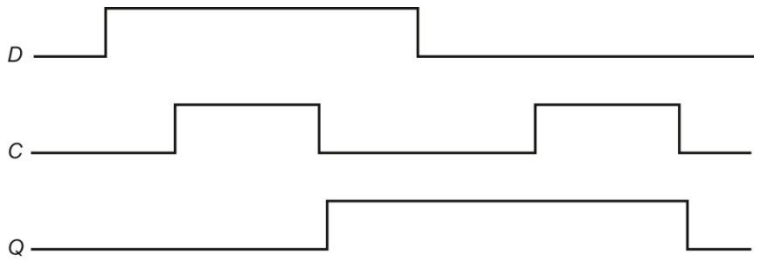


Figure A.8.5 Operation of a D flip-flop with a falling-edge trigger, assuming the output is initially deasserted. When the clock input (*C) changes from asserted to deasserted, the Q output stores* the value of the *D input. Compare this behavior to that of the clocked D latch shown in Figure A.8.3. In a* clocked latch, the stored value and the output, *Q, both change whenever C is high, as opposed to only when C transitions.*

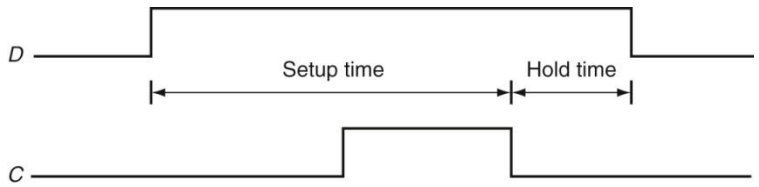


Figure A.8.6 Setup and hold time requirements for a D flip-flop with a falling-edge trigger. The input must be stable for a period of time before the clock edge, as well as after the clock edge. The minimum time the signal must be stable before the clock edge is called the setup time, while the minimum time the signal must be stable after the clock edge is called the hold time. Failure to meet these minimum requirements can result in a situation where the output of the flip-flop may not be predictable, as described in Section A.11. Hold times are usually either 0 or very small and thus not a cause of worry.

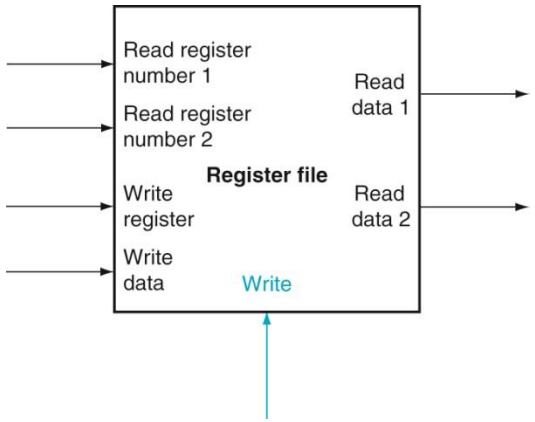


Figure A.8.7 A register file with two read ports and one write port has five inputs and two outputs. The control input Write is shown in color.

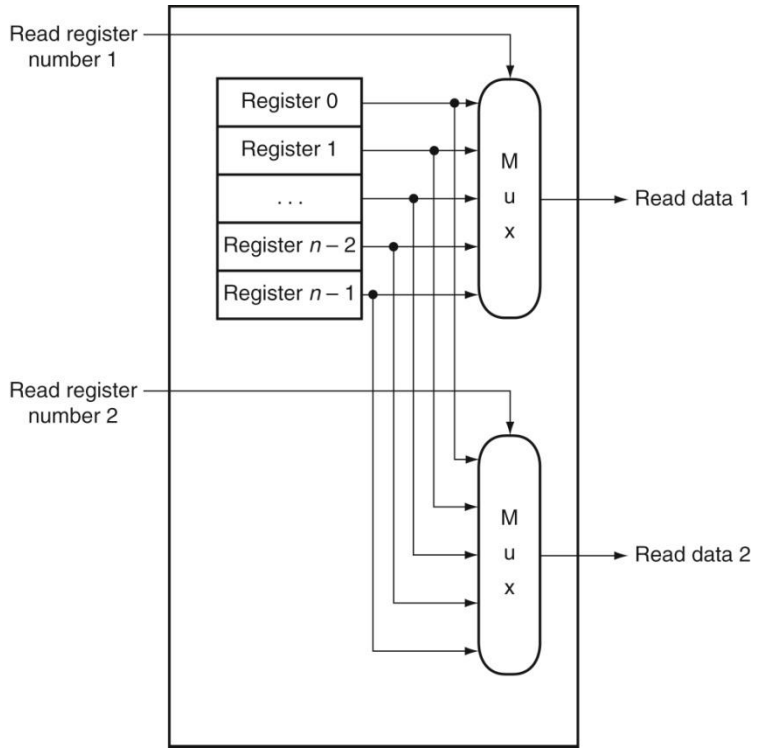


Figure A.8.8 The implementation of two read ports for a register file with *n registers* can be done with a pair of *n-to-1 multiplexors, each 32 bits wide*. *The register read number* signal is used as the multiplexor selector signal. Figure A.8.9 shows how the write port is implemented.

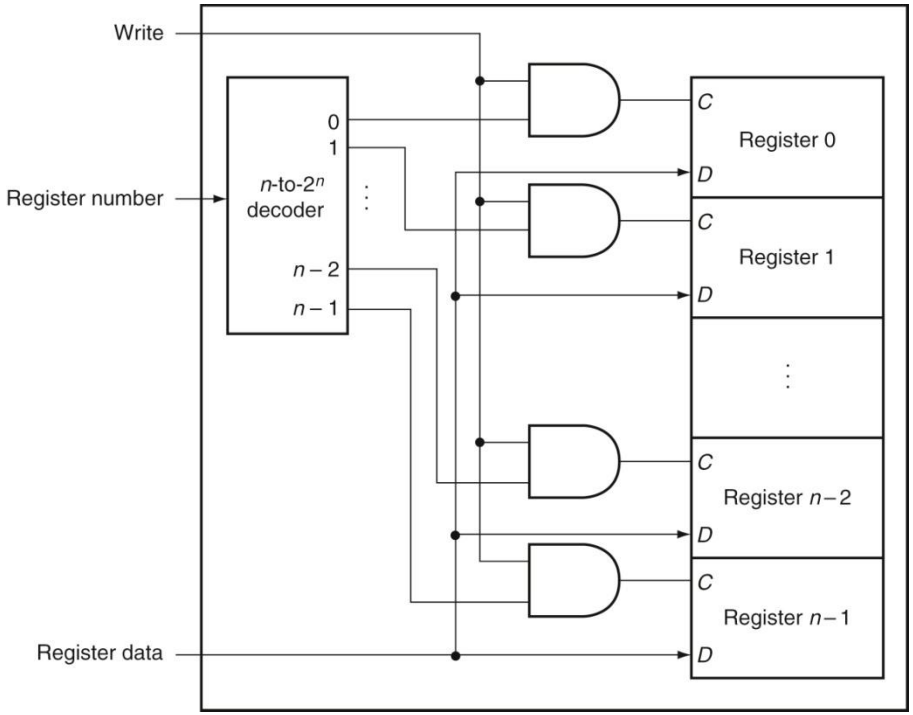


Figure A.8.9 The write port for a register file is implemented with a decoder that is used with the write signal to generate the C input to the registers. All three inputs (the register number, the data, and the write signal) will have setup and hold-time constraints that ensure that the correct data are written into the register file.

```
reg clock;
always #1 clock = ~clock;
```

Figure A.8.10 A specification of a clock.

```
    reg [31:0] A;
    wire [31:0] b;

    always @(posedge clock) A <= b;

module registerfile (Read1,Read2,WriteReg,WriteData,RegWrite,
Data1,Data2,clock);

   input [5:0] Read1,Read2,WriteReg; // the register numbers
to read or write
   input [31:0] WriteData; // data to write
   input RegWrite, // the write control
     clock; // the clock to trigger write
   output [31:0] Data1, Data2; // the register values read
   reg [31:0] RF [31:0]; // 32 registers each 32 bits long

   assign Data1 = RF[Read1];
   assign Data2 = RF[Read2];

   always begin
      // write the register with new value if Regwrite is
high
      @(posedge clock) if (RegWrite) RF[WriteReg] <=
WriteData;
   end
endmodule
```

Figure A.8.11 A RISC-V register file written in behavioral Verilog. This register file writes on the rising clock edge.

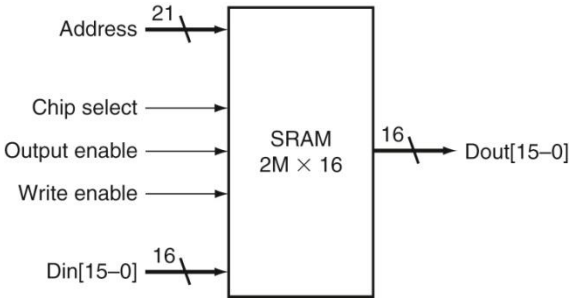


Figure A.9.1 A 32K × 8 SRAM showing the 21 address lines (32K = 215) and 16 data inputs, the three control lines, and the 16 data outputs.

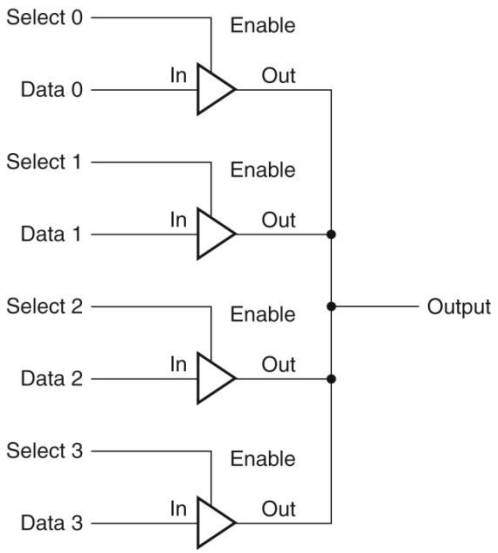


Figure A.9.2 Four three-state buffers are used to form a multiplexor. Only one of the four Select inputs can be asserted. A three-state buffer with a deasserted Output enable has a high-impedance output that allows a three-state buffer whose Output enable is asserted to drive the shared output line.

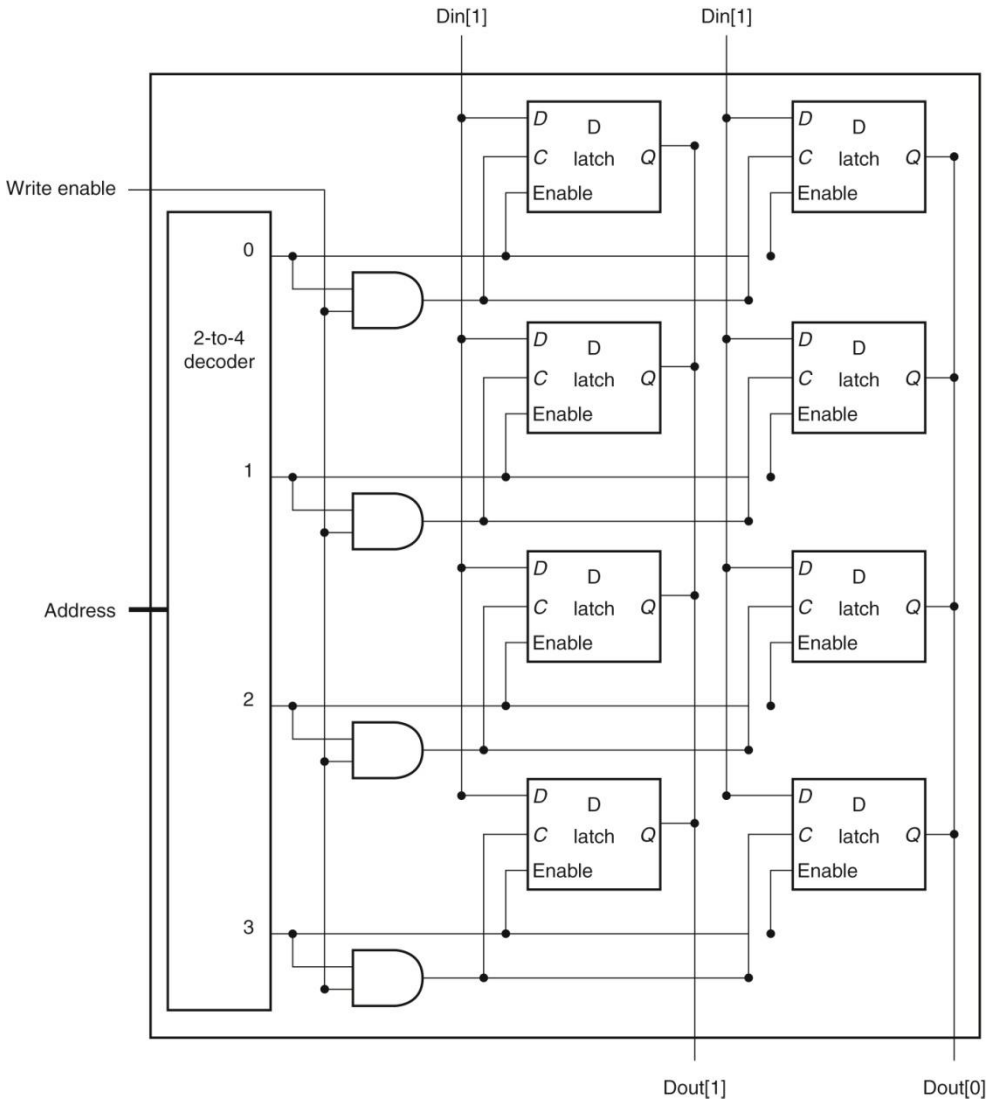


Figure A.9.3 The basic structure of a 4 × 2 SRAM consists of a decoder that selects which pair of cells to activate. The activated cells use a three-state output connected to the vertical bit lines that supply the requested data. The address that selects the cell is sent on one of a set of horizontal address lines, called word lines. For simplicity, the Output enable and Chip select signals have been omitted, but they could easily be added with a few AND gates.

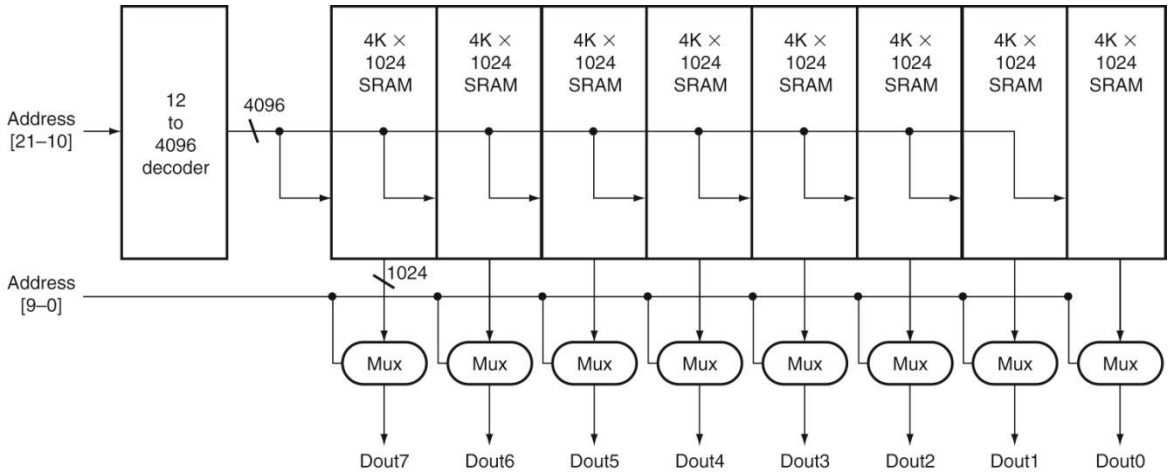


Figure A.9.4 Typical organization of a 4M × 8 SRAM as an array of 4K × 1024 arrays. The first decoder generates the addresses for eight 4K × 1024 arrays; then a set of multiplexors is used to select 1 bit from each 1024-bit-wide array. This is a much easier design than a single-level decode that would need either an enormous decoder or a gigantic multiplexor. In practice, a modern SRAM of this size would probably use an even larger number of blocks, each somewhat smaller.

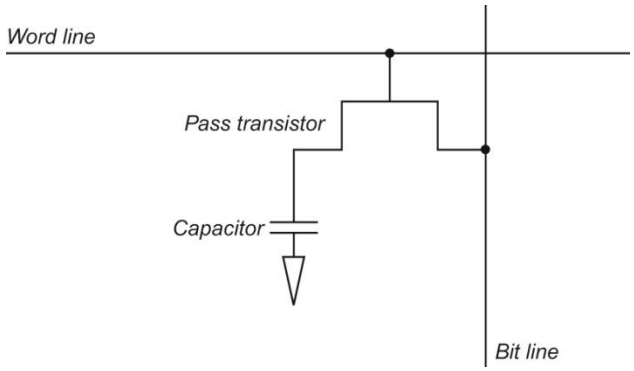


Figure A.9.5 A single-transistor DRAM cell contains a capacitor that stores the cell contents and a transistor used to access the cell.

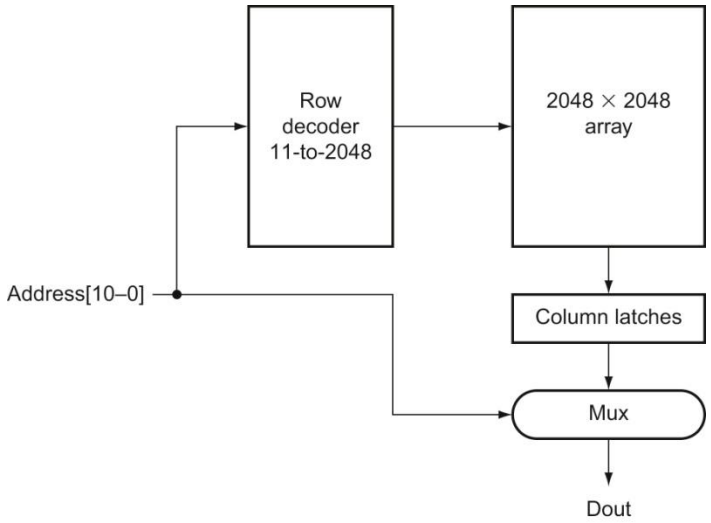


Figure A.9.6 A 4M × 1 DRAM is built with a 2048 × 2048 array. The row access uses 11 bits to select a row, which is then latched in 2048 1-bit latches. A multiplexor chooses the output bit from these 2048 latches. The RAS and CAS signals control whether the address lines are sent to the row decoder or column multiplexor.

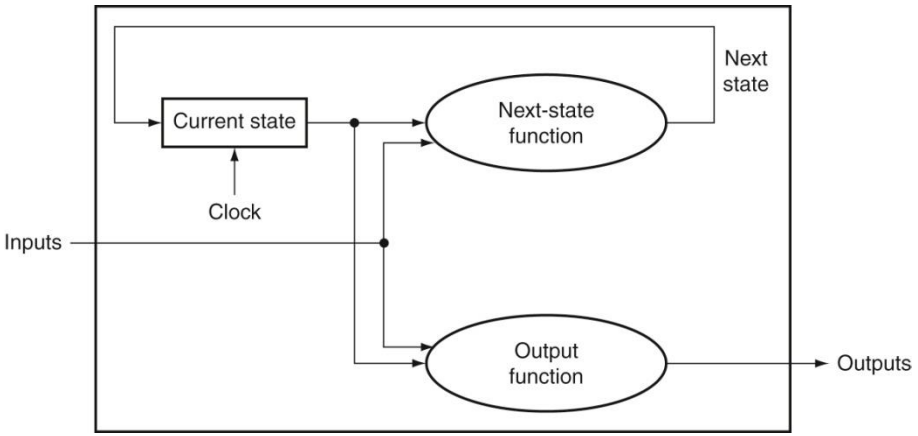


Figure A.10.1 A state machine consists of internal storage that contains the state and two combinational functions: the next-state function and the output function. Often, the output function is restricted to take only the current state as its input; this does not change the capability of a sequential machine, but does affect its internals.

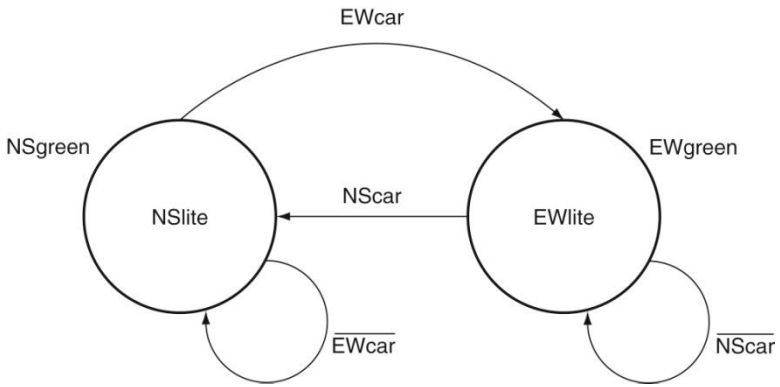


Figure A.10.2 The graphical representation of the two-state traffic light controller. We simplified the logic functions on the state transitions. For example, the transition from NSgreen to EWgreen in the next-state table is (NScar EWcar) (NScar EWcar), which is equivalent to EWcar.

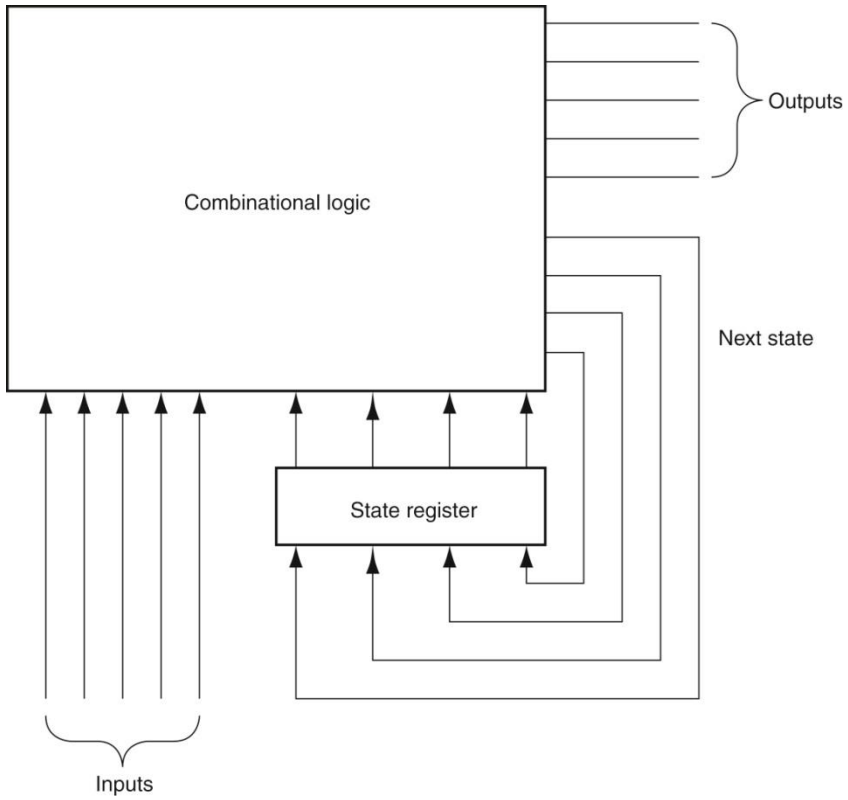


Figure A.10.3 A finite-state machine is implemented with a state register that holds the current state and a combinational logic block to compute the next state and output functions. The latter two functions are often split apart and implemented with two separate blocks of logic, which may require fewer gates.

```
module TrafficLite (EWCar,NSCar,EWLite,NSLite,clock);

   input EWCar, NSCar,clock;
output EWLite,NSLite;

reg state;

initial state=0;  //set initial state

//following two assignments set the output, which is based
only on the state variable
assign NSLite = ~ state; //NSLite on if state = 0;
assign EWLite = state; //EWLite on if state = 1

always @(posedge clock) // all state updates on a positive
clock edge
   case (state)
      0: state = EWCar; //change state only if EWCar

      1: state = ~ NSCar; // change state only if NSCar

   endcase
endmodule
```

Figure A.10.4 A Verilog version of the traffic light controller.

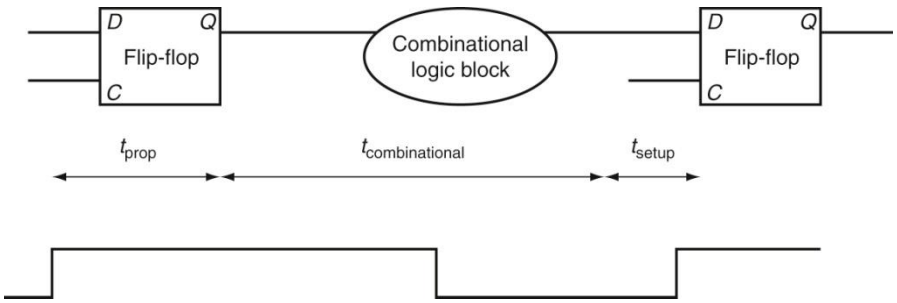


Figure A.11.1 In an edge-triggered design, the clock must be long enough to allow signals to be valid for the required setup time before the next clock edge. The time for a flip-flop input to propagate to the flip-flip outputs is *tprop; the signal then takes tcombinational to travel through the* combinational logic and must be valid *tsetup before the next clock edge.*

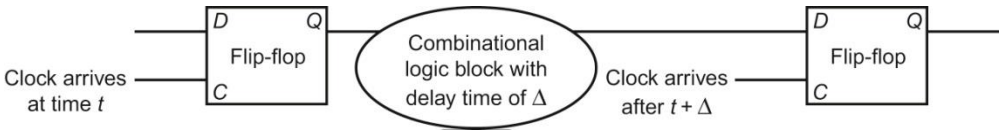


Figure A.11.2 Illustration of how clock skew can cause a race, leading to incorrect operation. Because of the difference in when the two flip-flops see the clock, the signal that is stored into the first flip-flop can race forward and change the input to the second flipflop before the clock arrives at the second flip-flop.

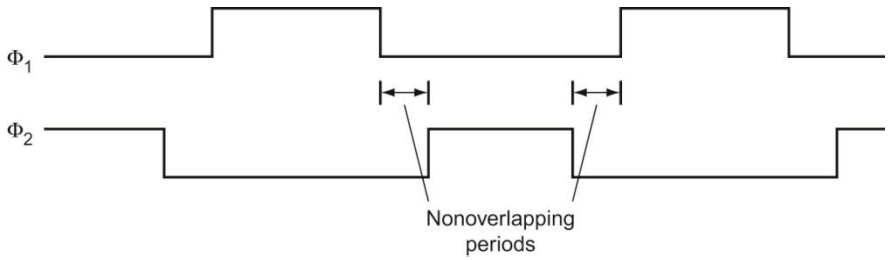


Figure A.11.3 A two-phase clocking scheme showing the cycle of each clock and the nonoverlapping periods.

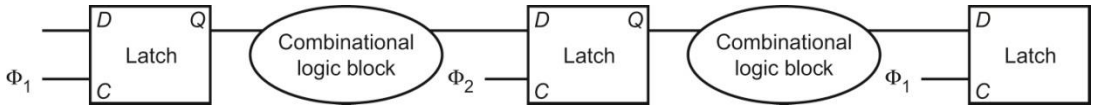


Figure A.11.4 A two-phase timing scheme with alternating latches showing how the system operates on both clock phases. The output of a latch is stable on the opposite phase from its C input. Thus, the first block of combinational inputs has a stable input during ϕ2, and its output is latched by ϕ2. The second (rightmost) combinational block operates in just the opposite fashion, with stable inputs during ϕ1. Thus, the delays through the combinational blocks determine the minimum time that the respective clocks must be asserted. The size of the nonoverlapping period is determined by the maximum clock skew and the minimum delay of any logic block.

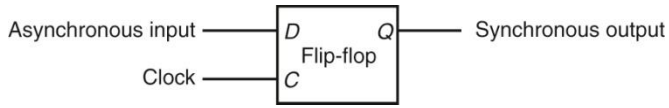


Figure A.11.5 A synchronizer built from a D flip-flop is used to sample an asynchronous signal to produce an output that is synchronous with the clock. This "synchronizer" will *not* work properly!

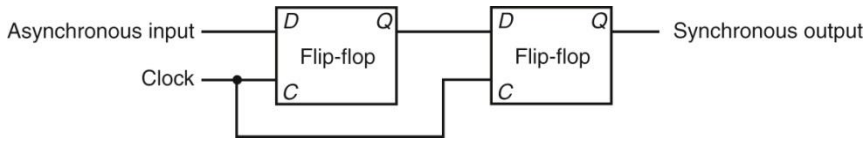
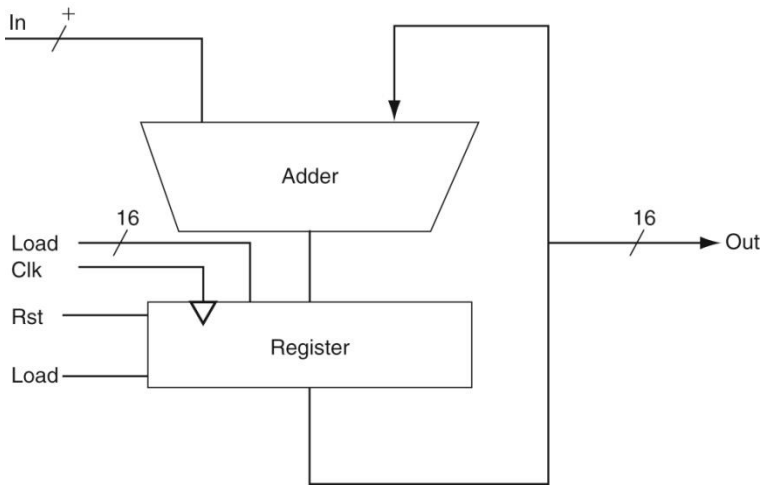


Figure A.11.6 This synchronizer will work correctly if the period of metastability that we wish to guard against is less than the clock period. Although the output of the first flip-flop may be metastable, it will not be seen by any other logic element until the second clock, when the second D flip-flop samples the signal, which by that time should no longer be in a metastable state.



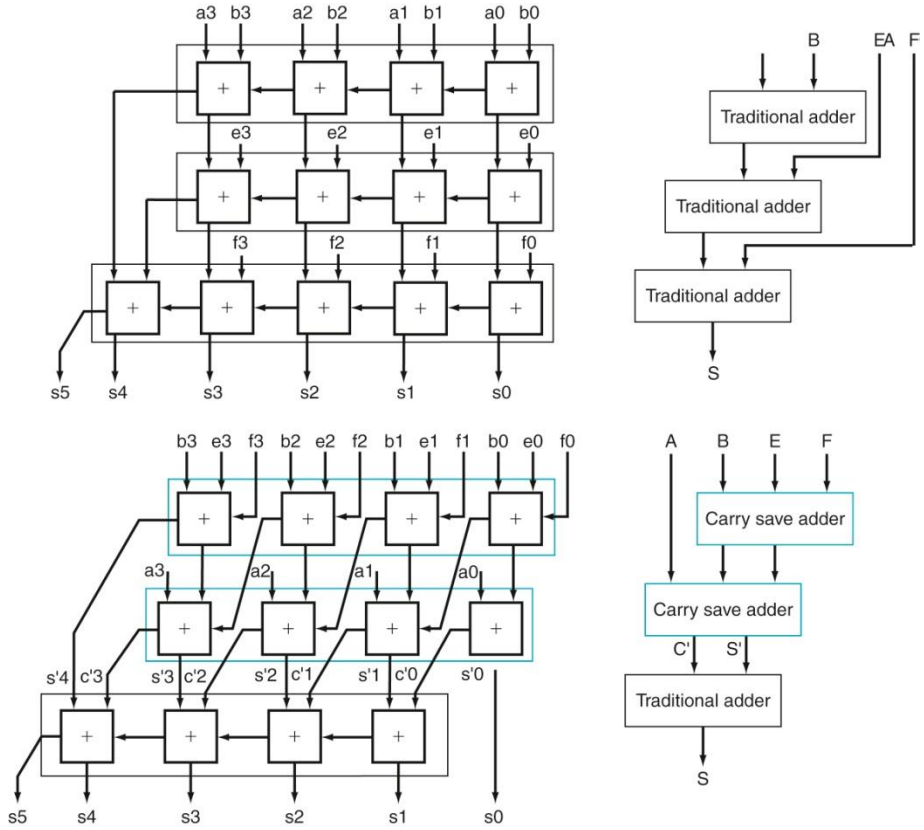


Figure A.14.1 Traditional ripple carry and carry save addition of four 4-bit numbers. The details are shown on the left, with the individual signals in lowercase, and the corresponding higher-level blocks are on the right, with collective signals in upper case. Note that the sum of four *n-bit numbers can* take *n + 2 bits.*.