

1.8 Logic Design

We begin the logic design by defining the top-level chip interface and block diagram. We then hierarchically decompose the units until we reach leaf cells. We specify the logic with a Hardware Description Language (HDL), which provides a higher level of abstraction than schematics or layout.

1.8.1 Top-level Interface

The top-level inputs and outputs are listed in Table 1.9. This example uses a two-phase clocking system to avoid hold-time problems. **Reset** initializes the PC to 0 and the control FSM to the start state. The remainder of the signals are used for an asynchronous 8-bit memory interface (assuming the memory is located off chip). The processor sends an 8-bit address **adr** and either asserts **memread** or **memwrite**. On a read cycle, the memory returns a value on the **memdata** lines while on a write cycle, the memory accepts input from **writedata**. In many systems, **memdata** and **writedata** can be combined onto a single bidirectional bus, but for this example we preserve the interface of Figure 1.53. Table 1.9 shows a very simple computer system built from the MIPS processor, external memory, reset switch, and clock generator.

Table 1.9 Top-level inputs and outputs

Inputs	Outputs
ph1	adr[7:0]
ph2	writedata[7:0]
reset	memread
memdata[7:0]	memwrite

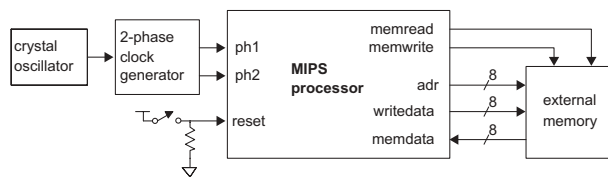


FIG 1.55 MIPS computer system

1.8.2 Block Diagram

The chip is partitioned into three top-level units: the controller, alucontrol, and datapath, as shown in the block diagram in Figure 1.56. The controller comprises the control FSM and the two gates used to compute $pcen$. The alucontrol consists of combinational logic to drive the ALU. The datapath contains the remainder of the chip, organized as eight identical *bitslices*. This partitioning is influenced by the intended physical design. The datapath contains most of the transistors and is very regular in structure. We can achieve high density with moderate design effort by handcrafting a single bitslice of the datapath, then replicating that bitslice eight times. The controller has much less structure. It is tedious to translate an FSM into gates by hand, and in a new design, the controller is the most likely portion to have bugs and last-minute changes. Therefore, we will specify the controller more abstractly with a hardware description language and automatically generate it using synthesis and place & route tools.

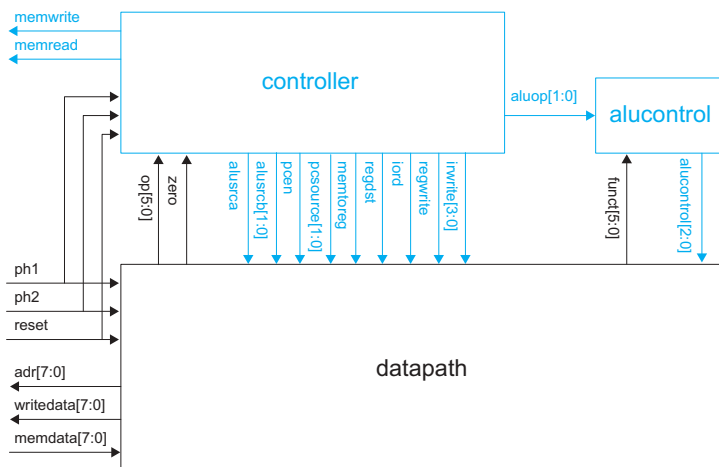


FIG 1.56 Top-level MIPS block diagram

1.8.3 Hierarchy

The best way to design complex systems is to decompose them into simpler pieces. Figure 1.57 shows the design hierarchy for the MIPS processor. The controller and alucontrol are built from a library of standard cells such as NANDs, NORs, and latches. The datapath is

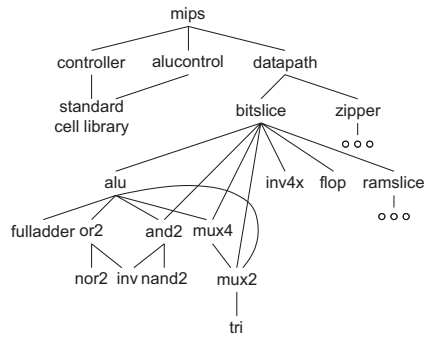


FIG 1.57 MIPS design hierarchy

composed of eight bitslices and a *zipper* that ties the datapath together by driving control signals and register enables to the various bits. The bitslice in turn is composed of the alu, register file ramslice, flip-flops, and gates. Some of these gates are reused in multiple places. The pieces composing the zipper and ramslice are left out for brevity.

The design hierarchy does not necessarily have to be identical in the logic, circuit, and physical designs. For example, in the logic view, a memory may be best treated as a black box, while in the circuit implementation, it may have a decoder, cell array, column multiplexers, and so forth. Different hierarchies complicate verification, however, because they must be *flattened* until the point that they agree. As a matter of practice, it is best to make logic, circuit, and physical design hierarchies agree as far as possible.

1.8.4 Hardware Description Languages

Designers need rapid feedback on whether a logic design is reasonable. Translating a block diagram and finite state machine state transition diagrams into circuit schematics is time-consuming and prone to error; before going through this entire process it is wise to know if the top-level design has major bugs that will require complete redesign. HDLs provide a way to specify the design at a higher level of abstraction to raise designer productivity. They were originally intended for documentation and simulation, but are now used to synthesize gates directly from the HDL.

The two most popular HDLs are *Verilog* and *VHDL*. Verilog was developed by Advanced Integrated Design Systems (later renamed Gateway Design Automation) in 1984 and became a *de facto* industry open standard by 1991. VHDL, which stands for VHSIC Hardware Description Language, where VHSIC in turn was a Department of Defense project on Very High Speed Integrated Circuits, was developed by committee under government sponsorship. As one might expect from their pedigrees, Verilog is less verbose and closer in syntax to C, while VHDL supports some abstractions useful for large team projects. Many Silicon Valley companies use Verilog while defense and telecommunications companies often use VHDL. Neither language offers a compelling advantage over the other so the industry is saddled with supporting both. Appendices A and B offer brief tutorials on Verilog and VHDL. Examples in this book are given in Verilog for the sake of brevity.

When coding in a HDL, it is important to remember that you are specifying hardware that executes in parallel rather than software that executes sequentially. There are two general coding styles. *Structural* HDL specifies how a cell is composed of other cells or primitive gates and transistors. *Behavioral* HDL specifies what a cell does.

A *logic simulator* simulates both behavioral and structural HDL. A *logic synthesis* tool maps behavioral HDL code onto a *library* of gates called *standard cells* to minimize area while meeting some timing constraint. Only a subset of HDL constructs are synthesiz-

able; this subset is emphasized in the appendices. For example, file I/O commands used in testbenches are obviously not synthesizable. Logic synthesis generally produces circuits that are neither as dense nor as fast as those handcrafted by a skilled designer. Nevertheless, integrated circuit processes are now so advanced that synthesized circuits are good enough for the great majority of application-specific integrated circuits (ASICs) built today. Layout may be automatically generated using *place & route* tools.

Verilog and VHDL models for the MIPS processor are listed in Appendix A.10 and B.7. In Verilog, each cell is called a *module*. The inputs and outputs are declared much as in a C program and bit widths are given for busses. Internal signals must also be declared in a way analogous to local variables. The processor is described hierarchically using structural Verilog at the upper levels and behavioral Verilog for the leaf cells. For example, the controller module shows how a finite state machine is specified in behavioral Verilog and the *alucontrol* module shows how complex combinational logic is specified. The datapath is specified structurally, containing bitslices, which in turn contain an ALU, which in turn contains a full adder.

The full adder could be expressed structurally as a sum and a carry subcircuit. In turn, the sum and carry subcircuits could be expressed behaviorally. The full adder block is shown in Figure 1.58 while the carry subcircuit is explored further in Section 1.9.

```

module fulladder(input a, b, c,
                 output s, cout);

    sum    s1(a, b, c, s);
    carry  c1(a, b, c, cout);
endmodule

module carry(input a, b, c,
             output cout)

    assign cout = (a&b) | (a&c) | (b&c);
endmodule

```

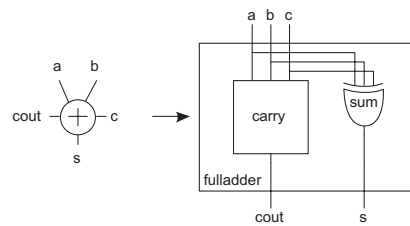


FIG 1.58 Full adder

1.9 Circuit Design

A particular logic function can be implemented in many ways. Should the function be built with ANDs, ORs, NANDs, or NORs? What should the fan-in and fan-out of each gate be? How wide should the transistors be on each gate? These and other choices influence the speed, power, and area of the system and are in the domain of circuit design.

As mentioned earlier, in many design methodologies, logic synthesis tools automatically make these choices, searching through the standard cells for the best implementation. For many applications, synthesis is good enough. When a system has critical requirements of high speed or low power or will be manufactured in large enough volume

to justify extra engineering to reduce die area, custom circuit design becomes important for critical portions of the chip.

Circuit designers often draw schematics at the transistor and/or gate level. For example, Figure 1.59 shows two alternative circuit designs for the carry circuit in a full adder. The gate-level design in Figure 1.59(a) requires 26 transistors and four stages of gate delays (recall that ANDs and ORs are built from NANDs and NORs followed by inverters). The transistor-level design in Figure 1.59(b) requires only 12 transistors and two stages of gate delays, illustrating the benefits of optimizing circuit designs to take advantage of CMOS technology.

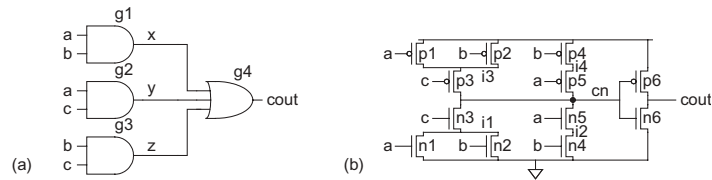


FIG 1.59 Carry subcircuit

These schematics are then *netlisted* for simulation and verification. One common netlist format is structural HDL. The gate-level design can be netlisted as shown below.

```

module carry(input a, b, c,
             output cout)

    wire x, y, z;

    and g1(x, a, b);
    and g2(y, a, c);
    and g3(z, b, c);
    or g4(cout, x, y, z);
endmodule
    
```

This is a technology-independent structural description, because generic gates have been used and the actual gate implementations have not been specified. The transistor-level netlist is shown below.

```

module carry(input a, b, c,
             output cout)
    
```

```

wire i1, i2, i3, i4, cn;

tranif1 n1(i1, 0, a);
tranif1 n2(i1, 0, b);
tranif1 n3(cn, i1, c);
tranif1 n4(i2, 0, b);
tranif1 n5(cn, i2, a);
tranif0 p1(i3, 1, a);
tranif0 p2(i3, 1, b);
tranif0 p3(cn, i3, c);
tranif0 p4(i4, 1, b);
tranif0 p5(cn, i4, a);
tranif1 n6(cout, 0, cn);
tranif0 p6(cout, 1, cn);
endmodule

```

Transistors are expressed as

```

Transistor-type name(drain, source, gate);

```

tranif1 corresponds to nMOS transistors that turn ON when the gate is '1' while tranif0 corresponds to pMOS transistors that turn ON when the gate is '0.'

With the description generated so far, we still do not have the information required to determine the speed of the gate. We need to specify the size of the transistors and the stray capacitance. Because the Verilog language was designed as a switch-level and gate-level language, it is poorly suited to structural descriptions at this level of detail. Hence we turn to another common structural language used by the circuit simulator SPICE. The specification of the transistor-level carry subcircuit at the circuit level might be represented as shown below.

```

.SUBCKT CARRY A B C COUT VDD GND
MN1 I1 A GND GND NMOS W=1U L=0.18U AD=0.3P AS=0.5P
MN2 I1 B GND GND NMOS W=1U L=0.18U AD=0.3P AS=0.5P
MN3 CN C I1 GND NMOS W=1U L=0.18U AD=0.5P AS=0.5P
MN4 I2 B GND GND NMOS W=1U L=0.18U AD=0.15P AS=0.5P
MN5 CN A I2 GND NMOS W=1U L=0.18U AD=0.5P AS=0.15P
MP1 I3 A VDD VDD PMOS W=2U L=0.18U AD=0.6P AS=1 P
MP2 I3 B VDD VDD PMOS W=2U L=0.18U AD=0.6P AS=1P
MP3 CN C I3 VDD PMOS W=2U L=0.18U AD=1P AS=1P
MP4 I4 B VDD VDD PMOS W=2U L=0.18U AD=0.3P AS=1P
MP5 CN A I4 VDD PMOS W=2U L=0.18U AD=1P AS=0.3P
MN6 COUT CN GND GND NMOS W=2U L=0.18U AD=1P AS=1P
MP6 COUT CN VDD VDD PMOS W=4U L=0.18U AD=2P AS=2P

```

```

CI1 I1 GND 2FF
CI3 I3 GND 3FF
CA A GND 4FF
CB B GND 4FF
CC C GND 2FF
CCN CN GND 4FF
CCOUT COUT GND 2FF
.ENDS

```

Transistors are specified by lines beginning with an M as follows:

```

Mname drain gate source body type W=width L=length
      AD=drain area AS=source area

```

The body connection is new. Although MOS switches have been masquerading as three terminal devices (gate, source, and drain) until this point, they are in fact four terminal devices with the substrate or well acting as the body terminal. The type specifies whether the transistor is a p-device or n-device. The width, length, and area parameters specify physical dimensions of the actual transistors. Capacitors are specified by lines beginning with C as follows:

```

Cname node1 node2 value

```

In this description the internal MOS model in SPICE calculates the parasitic capacitances inherent in the MOS transistor using the device dimensions specified. The extra capacitance statements in the above description designate additional routing capacitance not inherent to the device structure. This depends on the physical design of the gate. At the circuit level of structural specification, all connections are given that are necessary to fully characterize the carry gate in terms of speed, power, and connectivity.

1.10 Physical Design

1.10.1 Floorplanning

Physical design begins with a floorplan. The floorplan estimates the area of major units in the chip and defines their relative placements. The floorplan is essential to determine whether a proposed design will fit in the chip area budgeted and to estimate wiring lengths and wiring congestion, so an initial floorplan should be prepared as soon as the logic is loosely defined. As usual, this process involves feedback. The floorplan will often suggest changes to the logic (and microarchitecture), which in turn changes the floorplan. As a complex design begins to stabilize, the floorplan is often hierarchically subdivided to describe the functional blocks within the units.