1.7 Example: A Simple MIPS Microprocessor

We consider an 8-bit subset of the MIPS microprocessor architecture [Patterson04, Harris07] because it is widely studied and is relatively simple, yet still 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 is available at `www.cmosvlsi.com` in which you can learn VLSI design by building the microprocessor yourself using a free open-source CAD tool called *Electric* or with commercial design tools from Cadence and Synopsys.

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 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.

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 × 4`[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	101000	n/a

a. Technically, MIPS addresses specify bytes. Instructions require a 4-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 2 bits) to be converted to byte addresses.

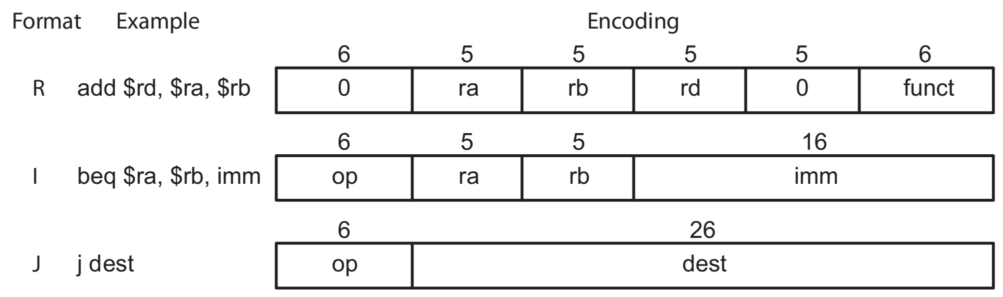

FIGURE 1.49 Instruction encoding formats

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 Java. A *compiler* translates a program from high-level language *source code* into the appropriate machine language *object code*.

Example 1.4

Figure 1.50 shows a simple C program that computes the nth 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.

```
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;
```

FIGURE 1.50 C Code for Fibonacci program

1.7.2 Multicycle MIPS Microarchitecture

We will implement the multicycle MIPS microarchitecture given in Chapter 5 of [Patterson04] and Chapter 7 of [Harris07] modified to process 8-bit data. The microarchitecture is illustrated in Figure 1.53. Light lines indicate individual signals while heavy

```
# 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
```

FIGURE 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 0000000000000100	10600004
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 00000000000000000000000011	08000003
sb $4, 255($0)	101000 00000 00100 0000000011111111	a00400ff

FIGURE 1.52 Machine language code for Fibonacci program

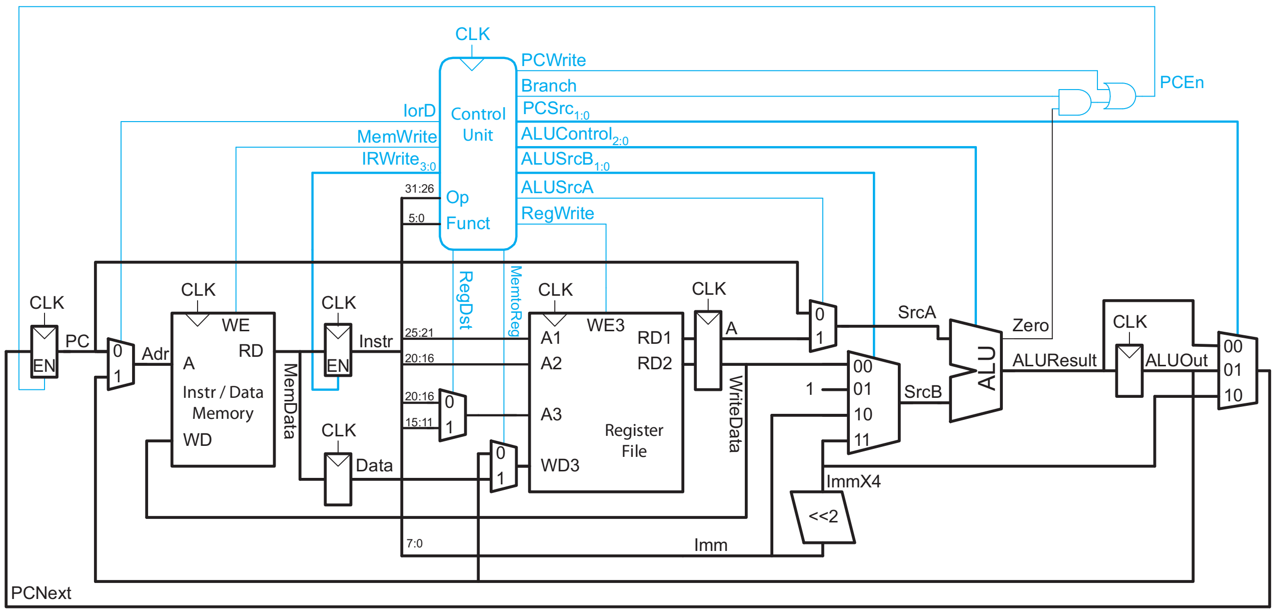

FIGURE 1.53 Multicycle MIPS microarchitecture. Adapted from [Patterson04] and [Harris07] with permission from Elsevier.

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 1 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 aludec 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 contains a finite state machine (FSM) 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

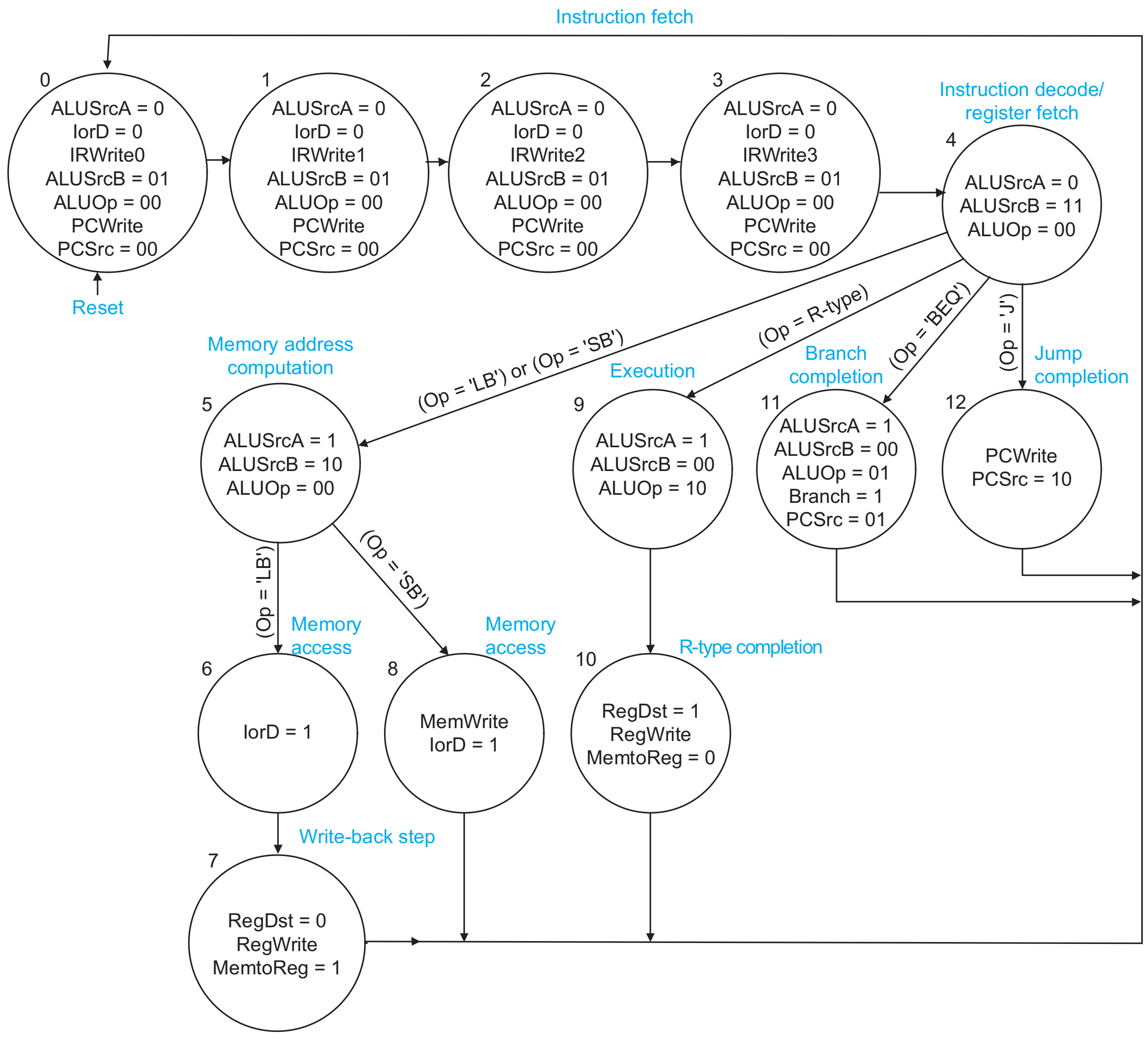

FIGURE 1.54 Multicycle MIPS control FSM (Adapted from [Patterson04] and [Harris07] with permission from Elsevier.)

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 FSM produces a 2-bit `ALUOp` output. The ALU decoder unit in the controller 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

ALUOp	Funct	ALUControl	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

Example 1.5

Referring to Figures 1.53 and 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, `IRWrite0` should be asserted to enable writing `memdata` into the least 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, `PCSrc` = 00 and `PCEn` = 1 (done by setting `PCWrite` = 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 1, 2, and 3 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.

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. This code is often called the Register Transfer Level (RTL) description.

1.8.1 Top-Level Interfaces

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.

TABLE 1.9 Top-level inputs and outputs

Inputs	Outputs
`ph1`	`MemWrite`
`ph2`	`Adr[7:0]`
`reset`	`WriteData[7:0]`
`MemData[7:0]`	

The remainder of the signals are used for an 8-bit memory interface (assuming the memory is located off chip). The processor sends an 8-bit address `Adr` and optionally asserts `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. Figure 1.55 shows a simple computer system built from the MIPS processor, external memory, reset switch, and clock generator.

1.8.2 Block Diagrams

The chip is partitioned into two top-level units: the controller and datapath, as shown in the block diagram in Figure 1.56. The controller comprises the control FSM, the ALU decoder, and the two gates used to compute `PCEn`. The ALU decoder consists of combina-

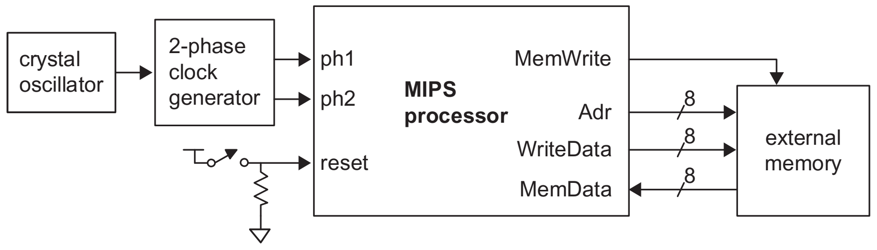

FIGURE 1.55 MIPS computer system

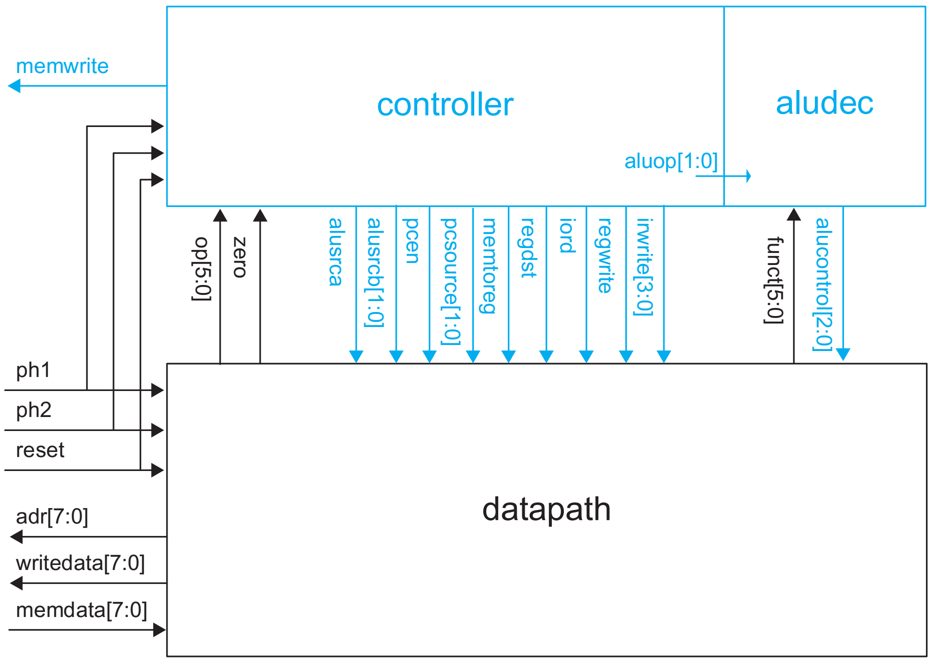

FIGURE 1.56 Top-level MIPS block diagram

tional logic to determine `ALUControl`. The 8-bit datapath contains the remainder of the chip. It can be viewed as a collection of wordslices or bitslices. A *wordslice* is a column containing an 8-bit flip-flop, adder, multiplexer, or other element. For example, Figure 1.57 shows a wordslice for an 8-bit 2:1 multiplexer. It contains eight individual 2:1 multiplexers, along with a *zipper* containing a buffer and inverter to drive the true and complementary select signals to all eight multiplexers.[6] Factoring these drivers out into the zipper saves space as compared to putting inverters in each multiplexer. Alternatively, the datapath can be viewed as eight rows of *bitslices*. Each bitslice has one bit of each component, along with the horizontal wires connecting the bits together.

The chip 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 each wordslice or bitslice and tiling the

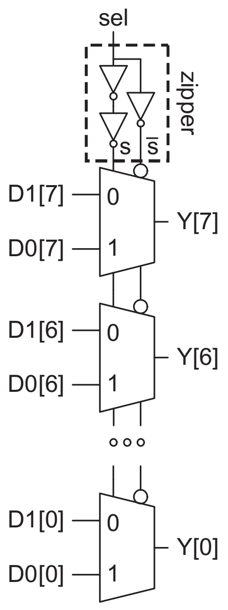

FIGURE 1.57 8-bit 2:1 multiplexer wordslice

[6]In this example, the zipper is shown at the top of the wordslice. In wider datapaths, the zipper is sometimes placed in the middle of the wordslice so that it drives shorter wires. The name comes from the way the layout resembles a plaid sweatshirt with a zipper down the middle.

circuits together. Building datapaths using wordslices is usually easier because certain structures, such as the zero detection circuit in the ALU, are not identical in each bitslice. However, thinking about bitslices is a valuable way to plan the wiring across the datapath.

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 or a programmable logic array (PLA).

1.8.3 Hierarchy

The best way to design complex systems is to decompose them into simpler pieces. Figure 1.58 shows part of the design hierarchy for the MIPS processor. The controller contains the controller_pla and aludec, which in turn is built from a library of standard cells such as NANDs, NORs, and inverters. The datapath is composed of 8-bit wordslices, each of which also is typically built from standard cells such as adders, register file bits, multiplexers, and flip-flops. Some of these cells are reused in multiple places.

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 block diagrams and FSM 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.

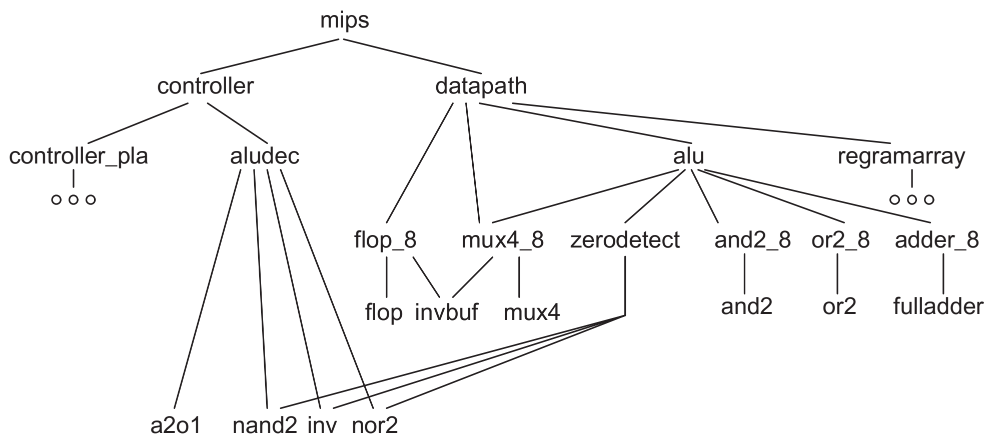

FIGURE 1.58 MIPS design hierarchy

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. In 2005, the SystemVerilog extensions were standardized, and some of these features are used in this book. 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 decisive advantage over the other so the industry is saddled with supporting both. Appendix A offers side-by-side tutorials on Verilog and VHDL. Examples in this book are given in Verilog for the sake of brevity.

When coding in an HDL, it is important to remember that you are specifying hardware that operates in parallel rather than software that executes in sequence. 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 HDL code; it can report whether results match expectations, and can display waveforms to help debug discrepancies. A *logic synthesis* tool is similar to a compiler for hardware: it maps HDL code onto a *library* of gates called *standard cells* to minimize area while meeting some timing constraints. Only a subset of HDL constructs are synthesizable; this subset is emphasized in the appendix. 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.12. 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 aludec module shows how complex combinational logic is specified. The datapath is specified structurally in terms of wordslices, which are in turn described behaviorally.

For the sake of illustration, the 8-bit adder wordslice could be described structurally as a ripple carry adder composed of eight cascaded full adders. 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 1.59 while the carry subcircuit is explored further in Section 1.9.

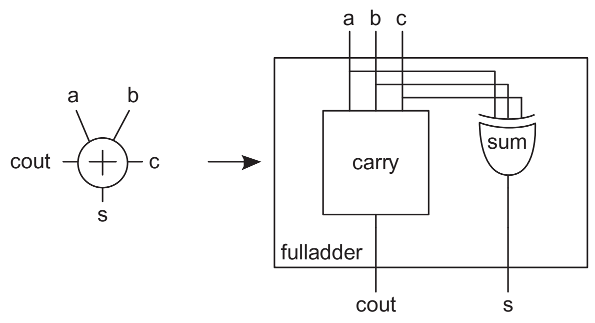

FIGURE 1.59 Full adder

```
module adder(input  logic [7:0] a, b,
             input  logic       c,
             output logic [7:0] s,
             output logic       cout);

  wire [6:0] carry;
```

```
    fulladder fa0(a[0], b[0], c,        s[0], carry[0]);
    fulladder fa1(a[1], b[1], carry[0], s[1], carry[1]);
    fulladder fa2(a[2], b[2], carry[1], s[2], carry[2]);
    ...
    fulladder fa7(a[7], b[7], carry[6], s[7], cout);
endmodule

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

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

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

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

1.9 Circuit Design

Circuit design is concerned with arranging transistors to perform a particular logic function. Given a circuit design, we can estimate the delay and power. The circuit can be represented as a schematic, or in textual form as a netlist. Common transistor level netlist formats include Verilog and SPICE. Verilog netlists are used for functional verification, while SPICE netlists have more detail necessary for delay and power simulations.

Because a transistor gate is a good insulator, it can be modeled as a capacitor, *C*. When the transistor is ON, some current *I* flows between source and drain. Both the current and capacitance are proportional to the transistor width.

The delay of a logic gate is determined by the current that it can deliver and the capacitance that it is driving, as shown in Figure 1.60 for one inverter driving another inverter. The capacitance is charged or discharged according to the constitutive equation

$$I = C\frac{dV}{dt}$$

If an average current *I* is applied, the time *t* to switch between 0 and V_{DD} is

$$t = \frac{C}{I}V_{DD}$$

Hence, the delay increases with the load capacitance and decreases with the drive current. To make these calculations, we will have to delve below the switch-level model of a transistor. Chapter 2 develops more detailed models of transistors accounting for the current and capacitance. One of the goals of circuit design is to choose transistor widths to meet delay requirements. Methods for doing so are discussed in Chapter 4.

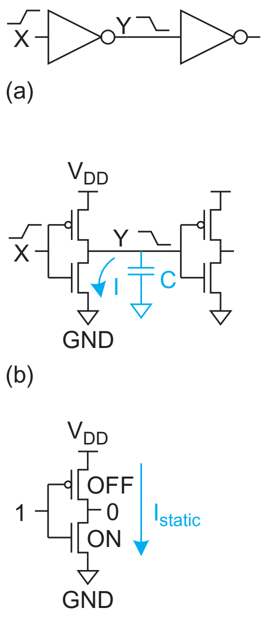

FIGURE 1.60 Circuit delay and power: (a) inverter pair, (b) transistor-level model showing capacitance and current during switching, (c) static leakage current during quiescent operation

Energy is required to charge and discharge the load capacitance. This is called dynamic power because it is consumed when the circuit is actively switching. The dynamic power consumed when a capacitor is charged and discharged at a frequency f is

$$P_{\text{dynamic}} = CV_{DD}^2 f$$

Even when the gate is not switching, it draws some static power. Because an OFF transistor is leaky, a small amount of current I_{static} flows between power and ground, resulting in a static power dissipation of

$$P_{\text{static}} = I_{\text{static}} V_{DD}$$

Chapter 5 examines power in more detail.

A particular logic function can be implemented in many ways. Should the function be built with ANDs, ORs, NANDs, or NORs? What should be the fan-in and fan-out of each gate? How wide should the transistors be on each gate? Each of these choices influences the capacitance and current and hence the speed and power of the circuit, as well as the area and cost.

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 the extra engineering, 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 1.61 shows two alternative circuit designs for the carry circuit in a full adder. The gate-level design in Figure 1.61(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 1.61(b) requires only 12 transistors and two stages of gate delays, illustrating the benefits of optimizing circuit designs to take advantage of CMOS technology.

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

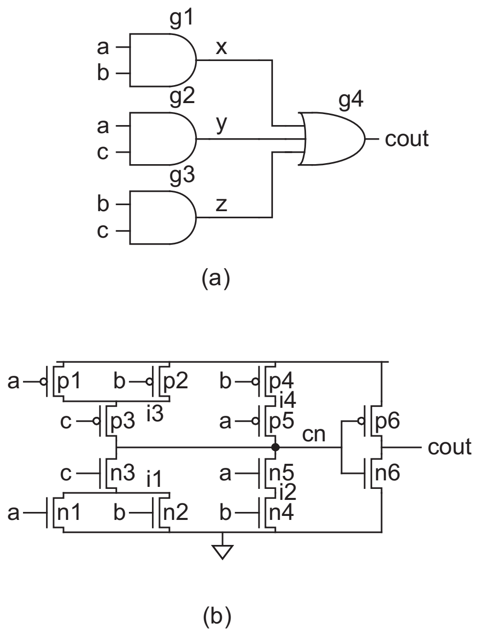

FIGURE 1.61 Carry subcircuit

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

  logic 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 follows:

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

  tri     i1, i2, i3, i4, cn;
  supply0 gnd;
  supply1 vdd;

  tranif1 n1(i1, gnd, a);
  tranif1 n2(i1, gnd, b);
  tranif1 n3(cn, i1, c);
  tranif1 n4(i2, gnd, b);
  tranif1 n5(cn, i2, a);
  tranif0 p1(i3, vdd, a);
  tranif0 p2(i3, vdd, b);
  tranif0 p3(cn, i3, c);
  tranif0 p4(i4, vdd, b);
  tranif0 p5(cn, i4, a);
  tranif1 n6(cout, gnd, cn);
  tranif0 p6(cout, vdd, 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. Appendix A.11 covers Verilog netlists in more detail.

With the description generated so far, we still do not have the information required to determine the speed or power consumption of the gate. We need to specify the size of the transistors and the stray capacitance. Because Verilog 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 follows:

```
.SUBCKT CARRY A B C COUT VDD GND
MN1 I1 A GND GND NMOS W=2U L=0.6U AD=1.8P AS=3P
MN2 I1 B GND GND NMOS W=2U L=0.6U AD=1.8P AS=3P
MN3 CN C I1 GND NMOS W=2U L=0.6U AD=3P AS=3P
MN4 I2 B GND GND NMOS W=2U L=0.6U AD=0.9P AS=3P
MN5 CN A I2 GND NMOS W=2U L=0.6U AD=3P AS=0.9P
MP1 I3 A VDD VDD PMOS W=4U L=0.6U AD=3.6P AS=6P
MP2 I3 B VDD VDD PMOS W=4U L=0.6U AD=3.6P AS=6P
MP3 CN C I3 VDD PMOS W=4U L=0.6U AD=6P AS=6P
```

```
MP4 I4 B VDD VDD PMOS W=4U L=0.6U AD=1.8P AS=6P
MP5 CN A I4 VDD PMOS W=4U L=0.6U AD=6P AS=1.8P
MN6 COUT CN GND GND NMOS W=4U L=0.6U AD=6P AS=6P
MP6 COUT CN VDD VDD PMOS W=8U L=0.6U AD=12P AS=12P
CI1 I1 GND 6FF
CI3 I3 GND 9FF
CA A GND 12FF
CB B GND 12FF
CC C GND 6FF
CCN CN GND 12FF
CCOUT COUT GND 6FF
.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
```

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 forming the *body* terminal. The body connection was not listed in Verilog but is required for SPICE. 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. Units include U (micro, 10^{-6}), P (pico, 10^{-12}), and F (femto, 10^{-15}). Capacitors are specified by lines beginning with C as follows:

```
Cname   node1   node2   value
```

In this description, the 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. Long wires also contribute resistance, which increases delay. 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. Chapter 8 describes SPICE models in more detail.

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. 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. For example, suppose microarchitects assume that a cache requires a 2-cycle access latency. If the floorplan shows that the data cache can be placed adjacent to the execution units in the datapath, the cache access time might reduce to a single cycle. This could allow the microarchitects to reduce the cache capacity while providing the same performance. Once the cache shrinks, the floorplan must be reconsidered to take advantage of the newly available space near the datapath. As a complex design begins to stabilize, the floorplan is often hierarchically subdivided to describe the functional blocks within the units.

The challenge of floorplanning is estimating the size of each unit without proceeding through a detailed design of the chip (which would depend on the floorplan and wire lengths). This section assumes that good estimates have been made and describes what a floorplan looks like. The next sections describe each of the types of components that might be in a floorplan and suggests ways to estimate the component sizes.

Figure 1.62 shows the chip floorplan for the MIPS processor including the pad frame. The top-level blocks are the controller and datapath. A wiring channel is located between the two blocks to provide room to route control signals to the datapath. The datapath is further partitioned into wordslices. The *pad frame* includes 40 I/O pads, which are wired to the pins on the chip package. There are 29 pads used for signals; the remainder are V_{DD} and GND.

The floorplan is drawn to scale and annotated with dimensions. The chip is designed in a 0.6 μm process on a 1.5 × 1.5 mm die so the die is 5000 λ on a side. Each pad is 750 λ ×

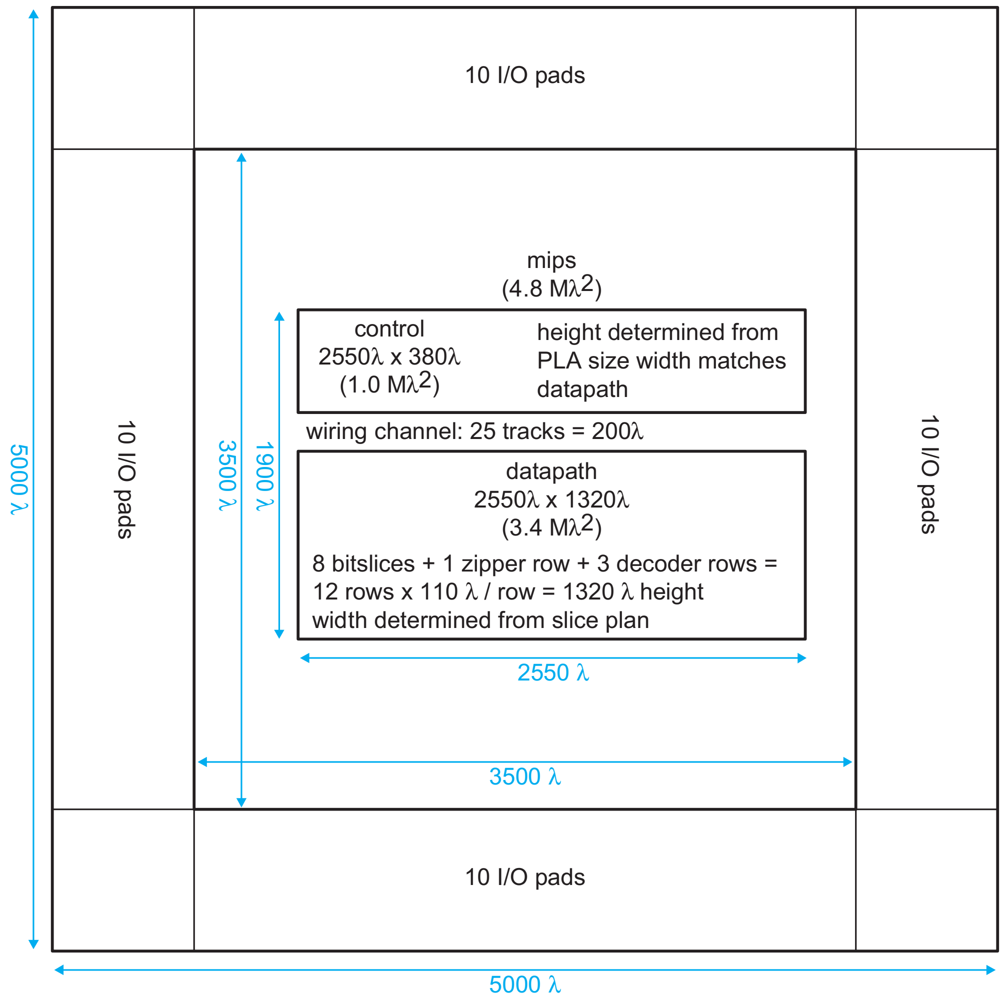
10 I/O pads
mips
(4.8 Mλ2)
control
2550λ x 380λ
(1.0 Mλ2)
height determined from PLA size width matches datapath
wiring channel: 25 tracks = 200λ
datapath
2550λ x 1320λ
(3.4 Mλ2)
8 bitslices + 1 zipper row + 3 decoder rows = 12 rows x 110 λ / row = 1320 λ height
width determined from slice plan
2550 λ
3500 λ
10 I/O pads
10 I/O pads
10 I/O pads
5000 λ
3500 λ
1900 λ
5000 λ

FIGURE 1.62 MIPS floorplan

350 λ, so the maximum possible core area inside the pad frame is 3500 $\lambda \times$ 3500 λ = 12.25 Mλ^2. Due to the wiring channel, the actual core area of 4.8 Mλ^2 is larger than the sum of the block areas. This design is said to be *pad-limited* because the I/O pads set the chip area. Most commercial chips are *core-limited* because the chip area is set by the logic excluding the pads. In general, blocks in a floorplan should be rectangular because it is difficult for a designer to stuff logic into an odd-shaped region (although some CAD tools do so just fine).

Figure 1.63 shows the actual chip layout. Notice the 40 I/O pads around the periphery. Just inside the pad frame are metal2 V_{DD} and GND rings, marked with + and −.

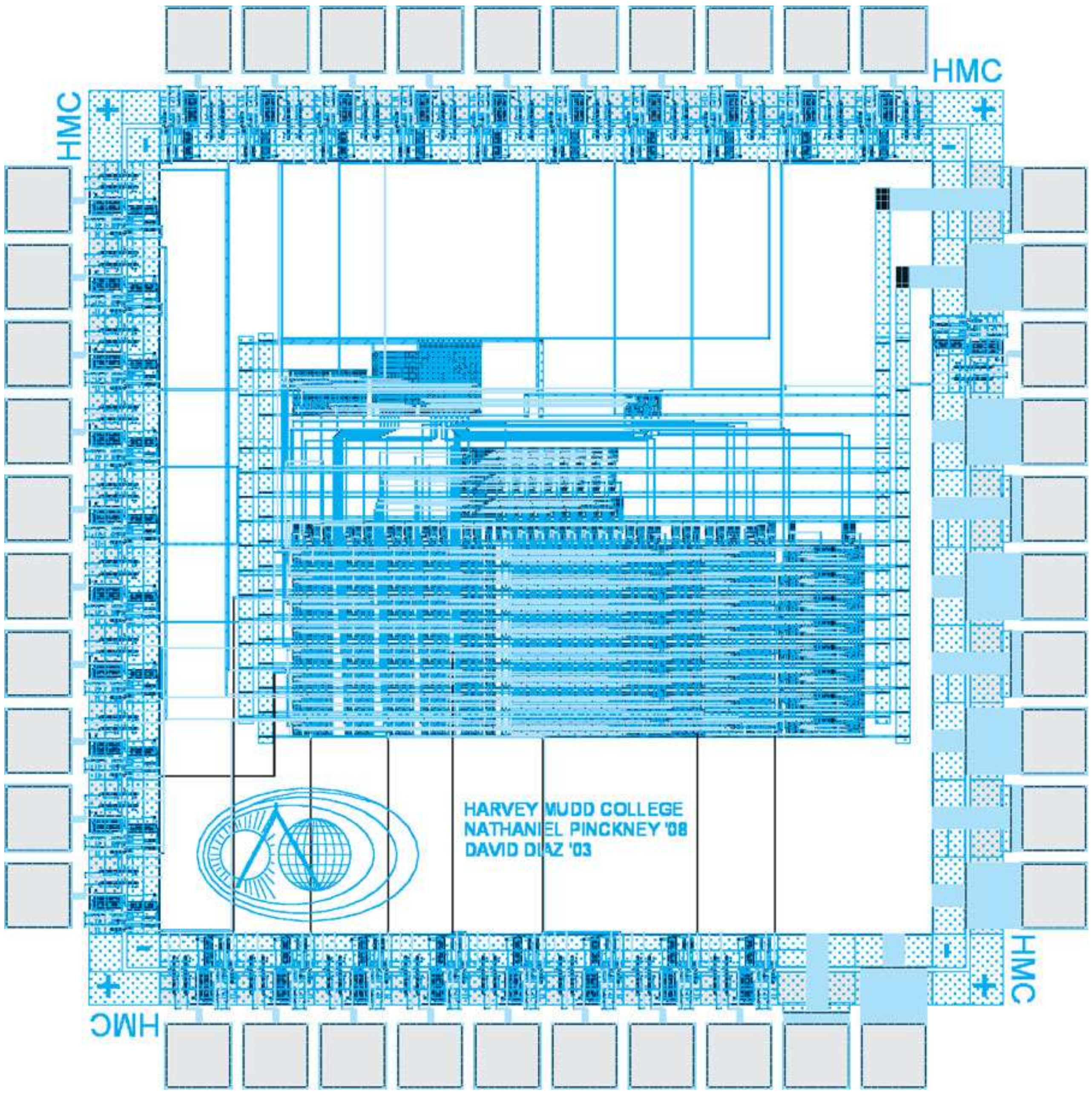

FIGURE 1.63 MIPS layout

On-chip structures can be categorized as *random logic*, *datapaths*, *arrays*, *analog*, and *input/output* (I/O). Random logic, like the aludecoder, has little structure. Datapaths operate on multi-bit data words and perform roughly the same function on each bit so they consist of multiple *N*-bit wordslices. Arrays, like RAMs, ROMs, and PLAs, consist of identical cells repeated in two dimensions. Productivity is highest if layout can be reused or automatically generated. Datapaths and arrays are good VLSI building blocks because a single carefully crafted cell is reused in one or two dimensions. Automatic layout generators exist for memory arrays and random logic but are not as mature for datapaths. Therefore, many design methodologies ignore the potential structure of datapaths and instead lay them out with random logic tools except when performance or area are vital. Analog circuits still require careful design and simulation but tend to involve only small amounts of layout because they have relatively few transistors. I/O cells are also highly tuned to each fabrication process and are often supplied by the process vendor.

Random logic and datapaths are typically built from *standard cells* such as inverters, NAND gates, and flip-flops. Standard cells increase productivity because each cell only needs to be drawn and verified once. Often, a standard cell library is purchased from a third party vendor.

Another important decision during floorplanning is to choose the metal orientation. The MIPS floorplan uses horizontal metal1 wires, vertical metal2 wires, and horizontal metal3 wires. Alternating directions between each layer makes it easy to cross wires on different layers.

1.10.2 Standard Cells

A simple standard cell library is shown on the inside front cover. Power and ground run horizontally in metal1. These supply rails are 8 λ wide (to carry more current) and are separated by 90 λ center-to-center. The nMOS transistors are placed in the bottom 40 λ of the cell and the pMOS transistors are placed in the top 50 λ. Thus, cells can be connected by abutment with the supply rails and n-well matching up. Substrate and well contacts are placed under the supply rails. Inputs and outputs are provided in metal2, which runs vertically. Each cell is a multiple of 8 λ in width so that it offers an integer number of metal2 tracks. Within the cell, poly is run vertically to form gates and diffusion and metal1 are run horizontally, though metal1 can also be run vertically to save space when it does not interfere with other connections.

Cells are tiled in rows. Each row is separated vertically by at least 110 λ from the base of the previous row. In a 2-level metal process, horizontal metal1 wires are placed in *routing channels* between the rows. The number of wires that must be routed sets the height of the routing channels. Layout is often generated with automatic place & route tools. Figure 1.64 shows the controller layout generated by such a tool. Note that in this and subsequent layouts, the n-well around the pMOS transistors will usually not be shown.

When more layers of metal are available, routing takes place over the cells and routing channels may become unnecessary. For example, in a 3-level metal process, metal3 is run horizontally on a 10 λ pitch. Thus, 11 horizontal tracks can run over each cell. If this is sufficient to accommodate all of the horizontal wires, the routing channels can be eliminated.

Automatic synthesis and place & route tools have become good enough to map entire designs onto standard cells. Figure 1.65 shows the entire 8-bit MIPS processor synthesized from the VHDL model given in Appendix A.12 onto a cell library in a 130 nm process with

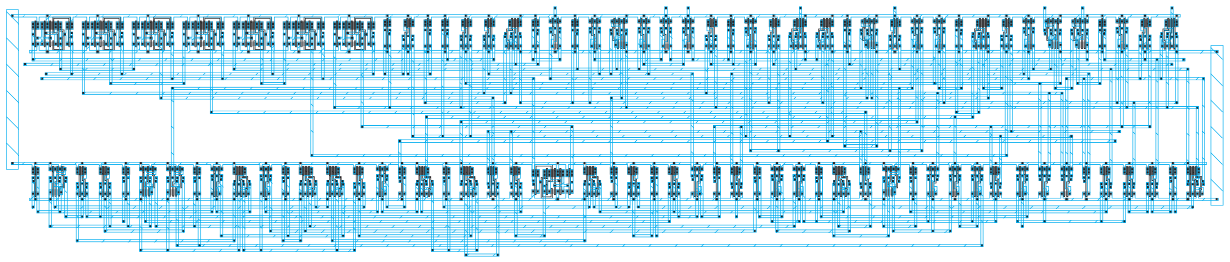
FIGURE 1.64 MIPS controller layout (synthesized)

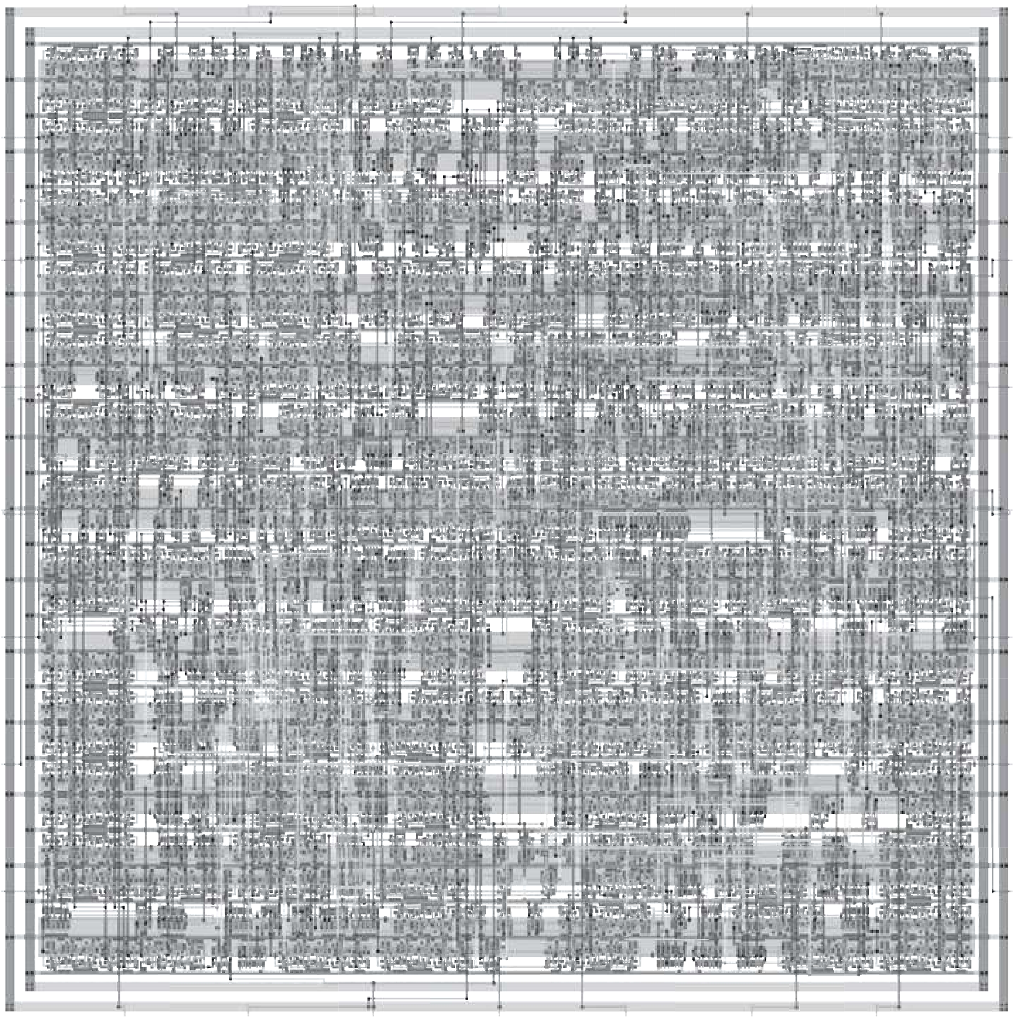
FIGURE 1.65 Synthesized MIPS processor

seven metal layers. Compared to Figure 1.63, the synthesized design shows little discernible structure except that 26 rows of standard cells can be identified beneath the wires. The area is approximately 4 Mλ^2. Synthesized designs tend to be somewhat slower than a good custom design, but they also take an order of magnitude less design effort.

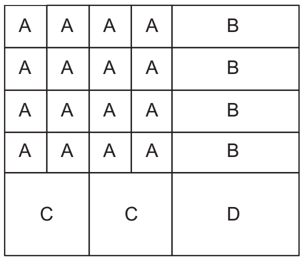

FIGURE 1.66 Pitch-matching of snap-together cells

1.10.3 Pitch Matching

The area of the controller in Figure 1.64 is dominated by the routing channels. When the logic is more regular, layout density can be improved by including the wires in cells that "snap together." Snap-together cells require more design and layout effort but lead to smaller area and shorter (i.e., faster) wires. The key issue in designing snap-together cells is *pitch-matching*. Cells that connect must have the same size along the connecting edge. Figure 1.66 shows several pitch-matched cells. Reducing the size of cell *D* does not help the layout area. On the other hand, increasing the size of cell *D* also affects the area of *B* and/or *C*.

Figure 1.67 shows the MIPS datapath in more detail. The eight horizontal bitslices are clearly visible. The zipper at the top of the layout includes three rows for the decoder that is pitch-matched to the register file in the datapath. Vertical metal2 wires are used for control, including clocks, multiplexer selects, and register enables. Horizontal metal3 wires run over the tops of cells to carry data along a bitslice.

The width of the transistors in the cells and the number of wires that must run over the datapath determines the minimum height of the datapath cells. 60–100 λ are typical heights for relatively simple datapaths. The width of the cell depends on the cell contents.

1.10.4 Slice Plans

Figure 1.68 shows a *slice plan* of the datapath. The diagram illustrates the ordering of wordslices and the allocation of wiring tracks within each bitslice. Dots indicate that a bus passes over a cell and is also used in that cell. Each cell is annotated with its type and width (in number of tracks). For example, the program counter (pc) is an output of the PC flop and is also used as an input to the srcA and address multiplexers. The slice plan

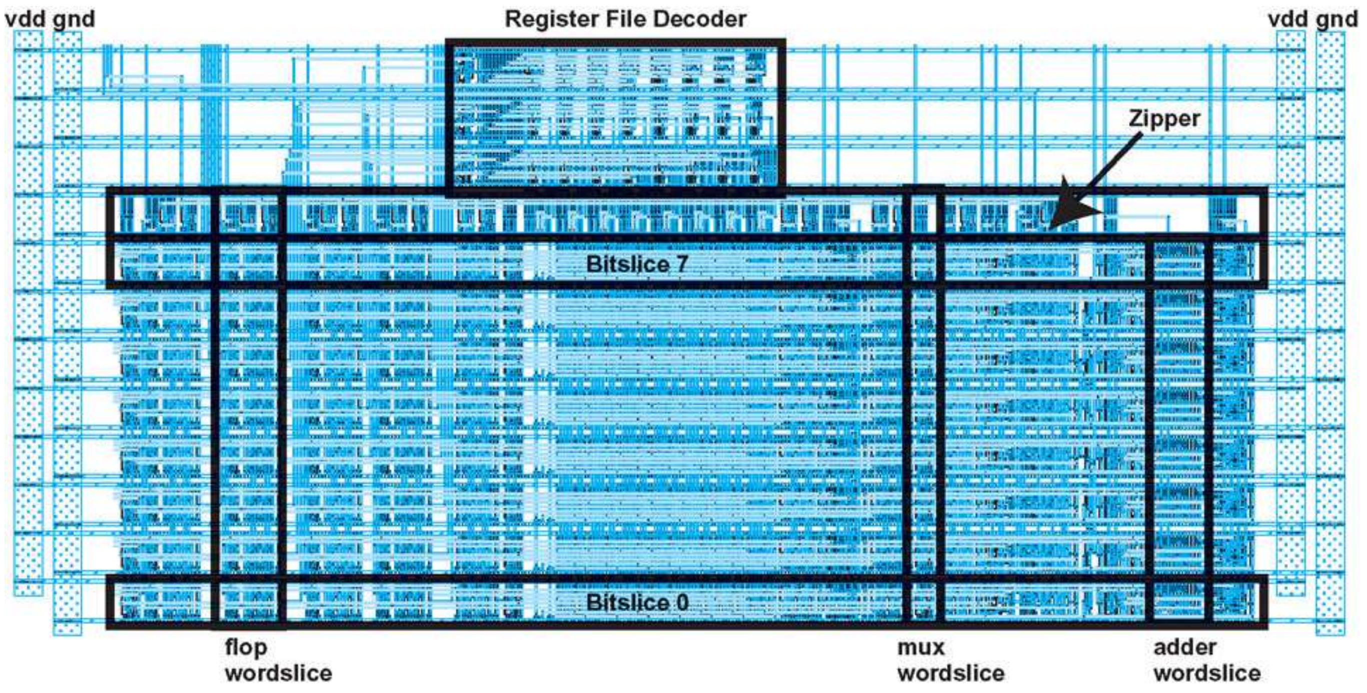

FIGURE 1.67 MIPS datapath layout

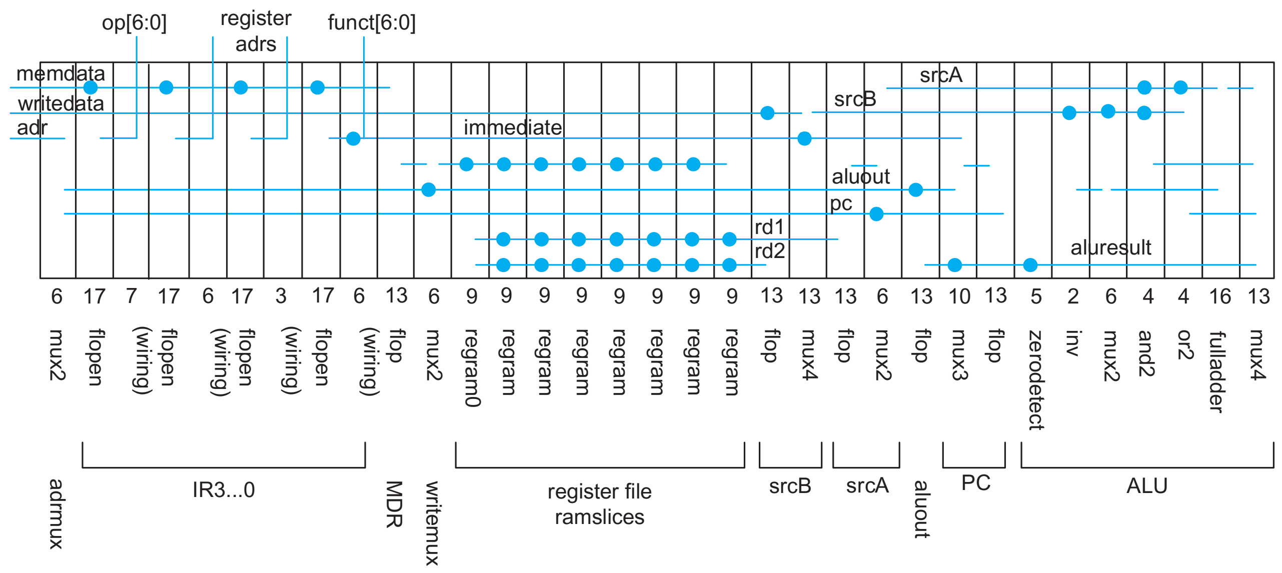

FIGURE 1.68 Datapath slice plan

makes it easy to calculate wire lengths and evaluate wiring congestion before laying out the datapath. In this case, it is evident that the greatest congestion takes place over the register file, where seven wiring tracks are required.

The slice plan is also critical for estimating area of datapaths. Each wordslice is annotated with its width, measured in tracks. This information can be obtained by looking at the cell library layouts. By adding up the widths of each element in the slice plan, we see that the datapath is 319 tracks wide, or 2552 λ wide. There are eight bitslices in the 8-bit datapath. In addition, there is one more row for the zipper and three more for the three register file address decoders, giving a total of 12 rows. At a pitch of 110 λ / row, the datapath is 1320 λ tall. The address decoders only occupy a small fraction of their rows, leaving wasted empty space. In a denser design, the controller could share some of the unused area.

1.10.5 Arrays

Figure 1.69 shows a programmable logic array (PLA) used for the control FSM next state and output logic. A PLA can compute any function expressed in sum of products form. The structure on the left is called the AND plane and the structure on the right is the OR plane. PLAs are discussed further in Section 12.7.

This PLA layout uses 2 vertical tracks for each input and 3 for each output plus about 6 for overhead. It uses 1.5 horizontal tracks for each product or minterm, plus about 14 for overhead. Hence, the size of a PLA is easy to calculate. The total PLA area is 500 $\lambda \times$ 350 λ, plus another 336 $\lambda \times$ 220 λ for the four external flip-flops needed in the control FSM. The height of the controller is dictated by the height of the PLA plus a few wiring tracks to route inputs and outputs. In comparison, the synthesized controller from Figure 1.64 has a size of 1500 $\lambda \times$ 400 λ because the wiring tracks waste so much space.

1.10.6 Area Estimation

A good floorplan depends on reasonable area estimates, which may be difficult to make before logic is finalized. An experienced designer may be able to estimate block area by

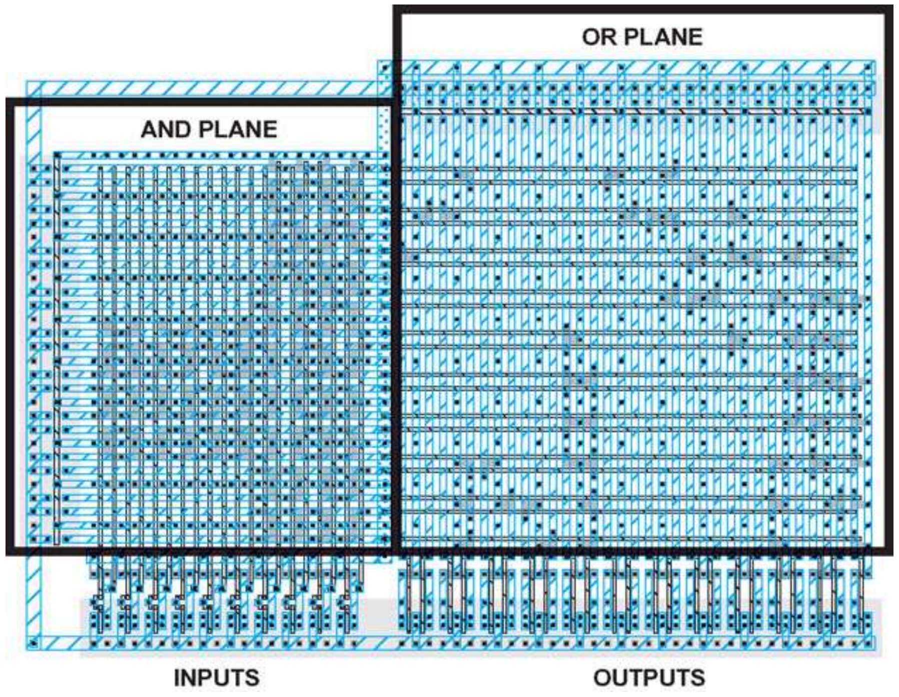

FIGURE 1.69 PLA for control FSM

comparison to the area of a comparable block drawn in the past. In the absence of data for such comparison, Table 1.10 lists some typical numbers. Be certain to account for large wiring channels at a pitch of 8 λ / track. Larger transistors clearly occupy a greater area, so this may be factored into the area estimates as a function of W and L (width and length). For memories, don't forget about the decoders and other periphery circuits, which often take as much area as the memory bits themselves. Your mileage may vary, but datapaths and arrays typically achieve higher densities than standard cells.

TABLE 1.10 Typical layout densities

Element	Area
random logic (2-level metal process)	1000 – 1500 λ^2 / transistor
datapath	250 – 750 λ^2 / transistor or 6 WL + 360 λ^2 / transistor
SRAM	1000 λ^2 / bit
DRAM (in a DRAM process)	100 λ^2 / bit
ROM	100 λ^2 / bit

Given enough time, it is nearly always possible to shave a few lambda here or there from a design. However, such efforts are seldom a good investment unless an element is repeated so often that it accounts for a major fraction of the chip area or if floorplan errors have led to too little space for a block and the block must be shrunk before the chip can be completed. It is wise to make conservative area estimates in floorplans, especially if there is risk that more functionality may be added to a block.

Some cell library vendors specify typical routed standard cell layout densities in kgates / mm^2.[7] Commonly, a gate is defined as a 3-input static CMOS NAND or NOR with six transistors. A 65 nm process ($\lambda \approx 0.03\ \mu$m) with eight metal layers may achieve a density of 160–500 kgates / mm^2 for random logic. This corresponds to about 370–1160 λ^2 / transistor. Processes with many metal layers obtain high density because routing channels are not needed.

1.11 Design Verification

Integrated circuits are complicated enough that if anything can go wrong, it probably will. Design verification is essential to catching the errors before manufacturing and commonly accounts for half or more of the effort devoted to a chip.

As design representations become more detailed, verification time increases. It is not practical to simulate an entire chip in a circuit-level simulator such as SPICE for a large number of cycles to prove that the layout is correct. Instead, the design is usually tested for functionality at the architectural level with a model in a language such as C and at the logic level by simulating the HDL description. Then, the circuits are checked to ensure that they are a faithful representation of the logic and the layout is checked to ensure it is a faithful representation of the circuits, as shown in Figure 1.70. Circuits and layout must meet timing and power specifications as well.

A *testbench* is used to verify that the logic is correct. The testbench instantiates the logic under test. It reads a file of inputs and expected outputs called *test vectors*, applies them to the module under test, and logs mismatches. Appendix A.12 provides an example of a testbench for verifying the MIPS processor logic.

A number of techniques are available for circuit verification. If the logic is synthesized onto a cell library, the postsynthesis gate-level netlist can be expressed in an HDL again and simulated using the same test vectors. Alternatively, a transistor-level netlist can be simulated against the test vector, although this can result in tricky race conditions for sequential circuits. Powerful *formal verification* tools are also available to check that a circuit performs the same Boolean function as the associated logic. Exotic circuits should be simulated thoroughly to ensure that they perform the intended logic function and have adequate noise margins; circuit pitfalls are discussed throughout this book.

Layout vs. Schematic tools (LVS) check that transistors in a layout are connected in the same way as in the circuit schematic. *Design rule checkers* (DRC) verify that the layout satisfies design rules. *Electrical rule checkers* (ERC) scan for other potential problems such as noise or premature wearout; such problems will also be discussed later in the book.

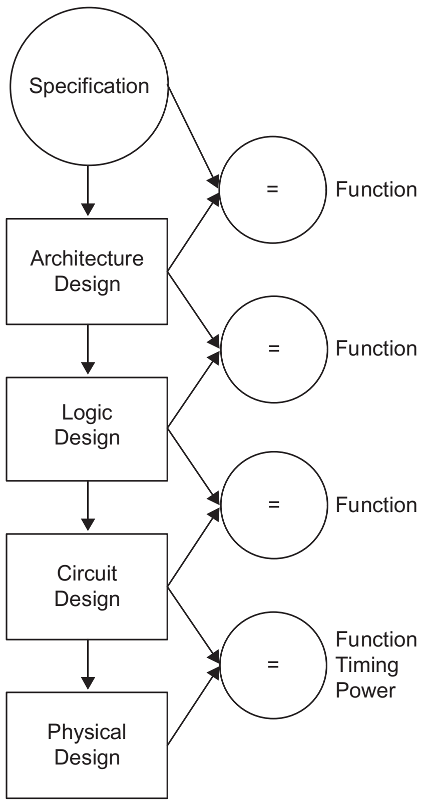

FIGURE 1.70 Design and verification sequence

[7] 1 kgate = 1000 gates.

1.12 Fabrication, Packaging, and Testing

Once a chip design is complete, it is taped out for manufacturing. *Tapeout* gets its name from the old practice of writing a specification of masks to magnetic tape; today, the mask descriptions are usually sent to the manufacturer electronically. Two common formats for mask descriptions are the Caltech Interchange Format (CIF) [Mead80] (mainly used in academia) and the Calma GDS II Stream Format (GDS) [Calma84] (used in industry).

Masks are made by etching a pattern of chrome on glass with an electron beam. A set of masks for a nanometer process can be very expensive. For example, masks for a large chip in a 180 nm process may cost on the order of a quarter of a million dollars. In a 65 nm process, the mask set costs about $3 million. The MOSIS service in the United States and its EUROPRACTICE and VDEC counterparts in Europe and Japan make a single set of masks covering multiple small designs from academia and industry to amortize the cost across many customers. With a university discount, the cost for a run of 40 small chips on a multi-project wafer can run about $10,000 in a 130 nm process down to $2000 in a 0.6 μm process. MOSIS offers certain grants to cover fabrication of class project chips.

Integrated circuit fabrication plants (fabs) now cost billions of dollars and become obsolete in a few years. Some large companies still own their own fabs, but an increasing number of fabless semiconductor companies contract out manufacturing to foundries such as TSMC, UMC, and IBM.

Multiple chips are manufactured simultaneously on a single silicon wafer, typically 150–300 mm (6″–12″) in diameter. Fabrication requires many deposition, masking, etching, and implant steps. Most fabrication plants are optimized for wafer throughput rather than latency, leading to turnaround times of up to 10 weeks. Figure 1.71 shows an engineer in a *clean room* holding a completed 300 mm wafer. Clean rooms are filtered to eliminate most dust and other particles that could damage a partially processed wafer. The engineer is wearing a "bunny suit" to avoid contaminating the clean room. Figure 1.72 is a

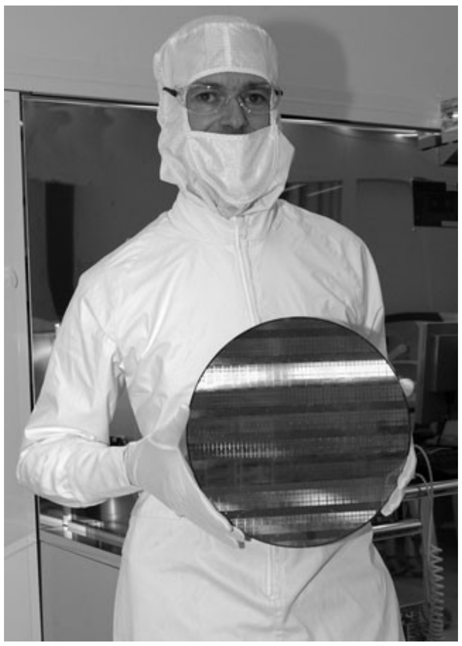
FIGURE 1.71 Engineer holding processed 12-inch wafer (Photograph courtesy of the Intel Corporation.)

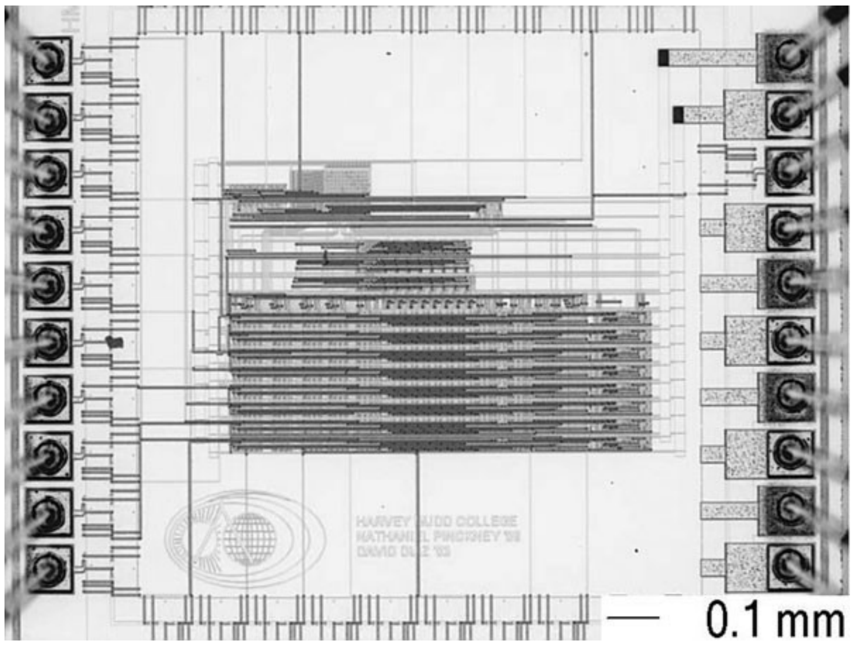

FIGURE 1.72 MIPS processor photomicrograph (only part of pad frame shown)

photomicrograph (a photograph taken under a microscope) of the 8-bit MIPS processor.

Processed wafers are sliced into dice (chips) and packaged. Figure 1.73 shows the 1.5 × 1.5 mm chip in a 40-pin *dual-inline package* (DIP). This *wire-bonded* package uses thin gold wires to connect the pads on the die to the lead frame in the center cavity of the package. These wires are visible on the pads in Figure 1.72. More advanced packages offer different trade-offs between cost, pin count, pin bandwidth, power handling, and reliability, as will be discussed in Section 13.2. Flip-chip technology places small solder balls directly onto the die, eliminating the bond wire inductance and allowing contacts over the entire chip area rather than just at the periphery.

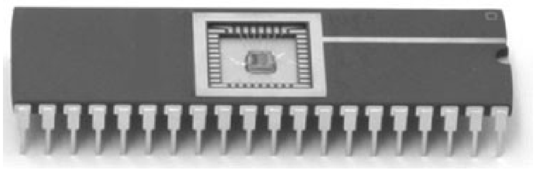

FIGURE 1.73 Chip in a 40-pin dual-inline package

Even tiny defects in a wafer or dust particles can cause a chip to fail. Chips are tested before being sold. Testers capable of handling high-speed chips cost millions of dollars, so many chips use built-in self-test features to reduce the tester time required. Chapter 15 is devoted to design verification and testing.

Summary and a Look Ahead

"If the automobile had followed the same development cycle as the computer, a Rolls-Royce would today cost $100, get one million miles to the gallon, and explode once a year . . ."

—Robert X. Cringely

CMOS technology, driven by Moore's Law, has come to dominate the semiconductor industry. This chapter examined the principles of designing a simple CMOS integrated circuit. MOS transistors can be viewed as electrically controlled switches. Static CMOS gates are built from pull-down networks of nMOS transistors and pull-up networks of pMOS transistors. Transistors and wires are fabricated on silicon wafers using a series of deposition, lithography, and etch steps. These steps are defined by a set of masks drawn as a chip layout. Design rules specify minimum width and spacing between elements in the layout. The chip design process can be divided into architecture, logic, circuit, and physical design. The performance, area, and power of the chip are influenced by interrelated decisions made at each level. Design verification plays an important role in constructing such complex systems; the reliability requirements for hardware are much greater than those typically imposed on software.

Primary design objectives include reliability, performance, power, and cost. Any chip should, with high probability, operate reliably for its intended lifetime. For example, the chip must be designed so that it does not overheat or break down from excessive voltage. Performance is influenced by many factors including clock speed and parallelism. CMOS transistors dissipate power every time they switch, so the dynamic power consumption is related to the number and size of transistors and the rate at which they switch. At feature