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

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

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

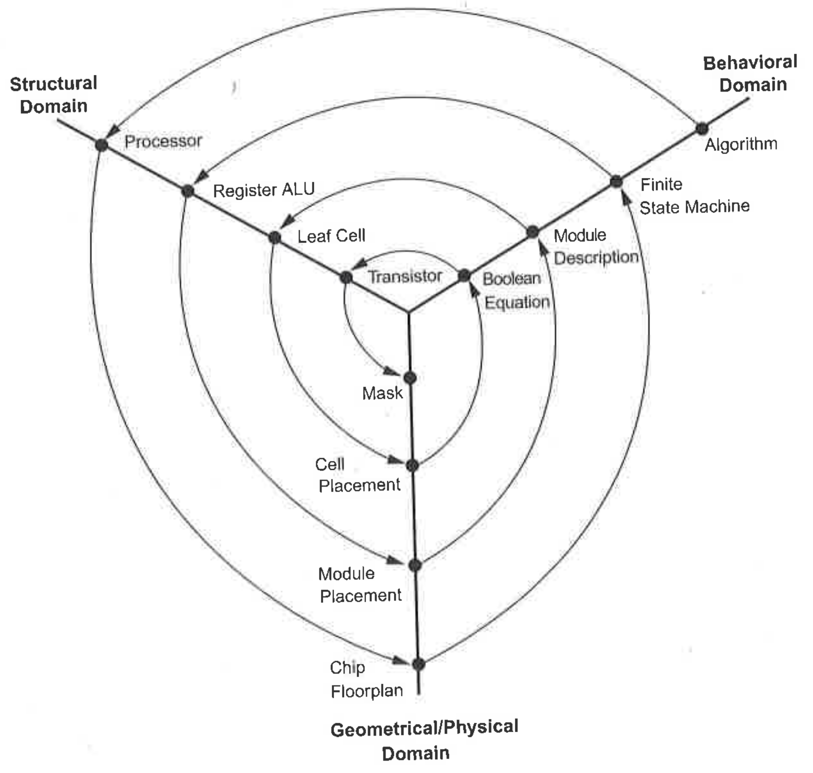


FIG 1.48 Y Diagram. Reproduced from [Kang03] with permission of The McGraw-Hill Companies.

1.7 Example: A Simple MIPS Microprocessor

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

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

1.7.1 MIPS Architecture

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

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

Table 1.7 MIPS instruction set (subset supported)

Instruction	Function		Encoding	op	funct
`add $1, $2, $3`	addition:	`$1 <- $2 + $3`	R	000000	100000
`sub $1, $2, $3`	subtraction:	`$1 <- $2 – $3`	R	000000	100010
`and $1, $2, $3`	bitwise and:	`$1 <- $2 and $3`	R	000000	100100
`or $1, $2, $3`	bitwise or:	`$1 <- $2 or $3`	R	000000	100101
`slt $1, $2, $3`	set less than:	`$1 <- 1 if $2 < $3` `$1 <- 0 otherwise`	R	000000	101010
`addi $1, $2, imm`	add immediate:	`$1 <- $2 + imm`	I	001000	n/a
`beq $1, $2, imm`	branch if equal:	`PC <- PC + imm`[a]	I	000100	n/a
`j destination`	jump:	`PC <- destination`[a]	J	000010	n/a
`lb $1, imm($2)`	load byte:	`$1 <- mem[$2 + imm]`	I	100000	n/a
`sb $1, imm($2)`	store byte:	`mem[$2 + imm] <- $1`	I	101000	n/a

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

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

Format	Example	Encoding					
		6	5	5	5	5	6
R	add $rd, $ra, $rb	0	ra	rb	rd	0	funct
		6	5	5	16		
I	beq $ra, $rb, imm	op	ra	rb	imm		
		6	26				
J	j dest	op	dest				

FIG 1.49 Instruction encoding formats

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

Example

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

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

continues

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

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

FIG 1.50 C code for Fibonacci program

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

FIG 1.51 Assembly language code for Fibonacci program

Instruction	Binary Encoding	Hexadecimal Encoding
addi $3, $0, 8	001000 00000 00011 0000000000001000	20030008
addi $4, $0, 1	001000 00000 00100 0000000000000001	20040001
addi $5, $0, -1	001000 00000 00101 1111111111111111	2005ffff
beq $3, $0, end	000100 00011 00000 0000000000000101	1060000~~5~~ 4
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)	~~110000~~ 101000 00000 00100 0000000011111111	a00400ff

FIG 1.52 Machine language code for Fibonacci program

1.7.2 Multicycle MIPS Microarchitecture

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

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

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

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

Table 1.8 ALUControl determination

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

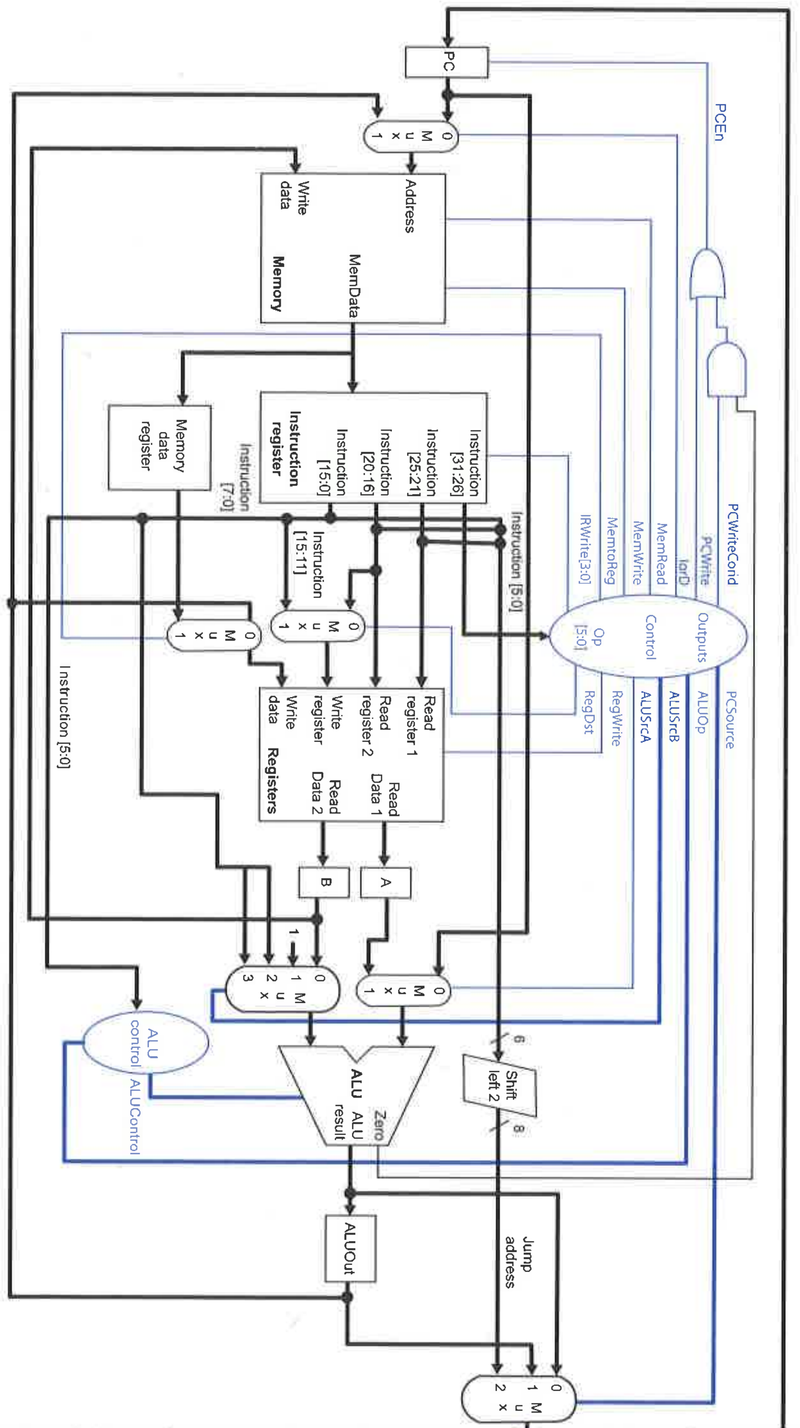


FIG 1.53 Multicycle MIPS microarchitecture. Reprinted from [Patterson04] with permission from Elsevier.

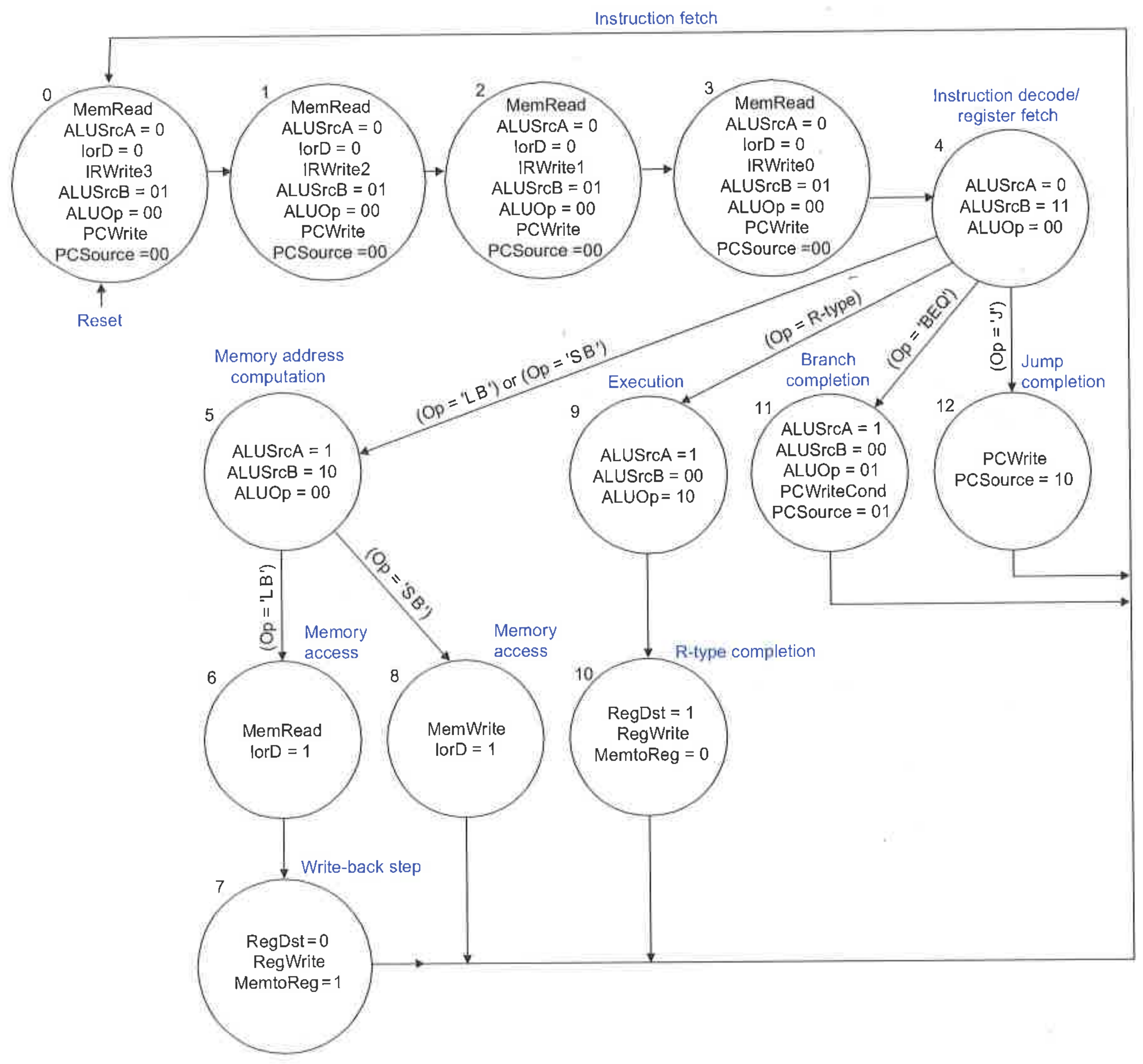


FIG 1.54 Multicycle MIPS control FSM. Reprinted from [Patterson04] with permission from Elsevier.

Example

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

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

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

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

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

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

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

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

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. Figure 1.55 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

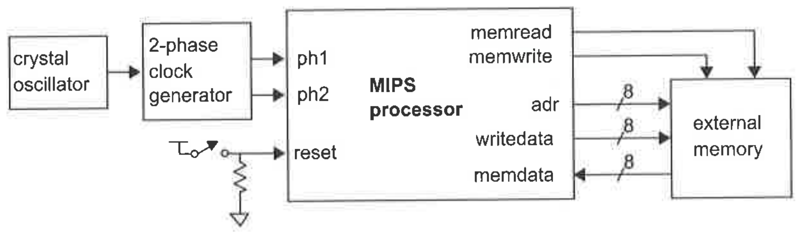


FIG 1.55 MIPS computer system

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.

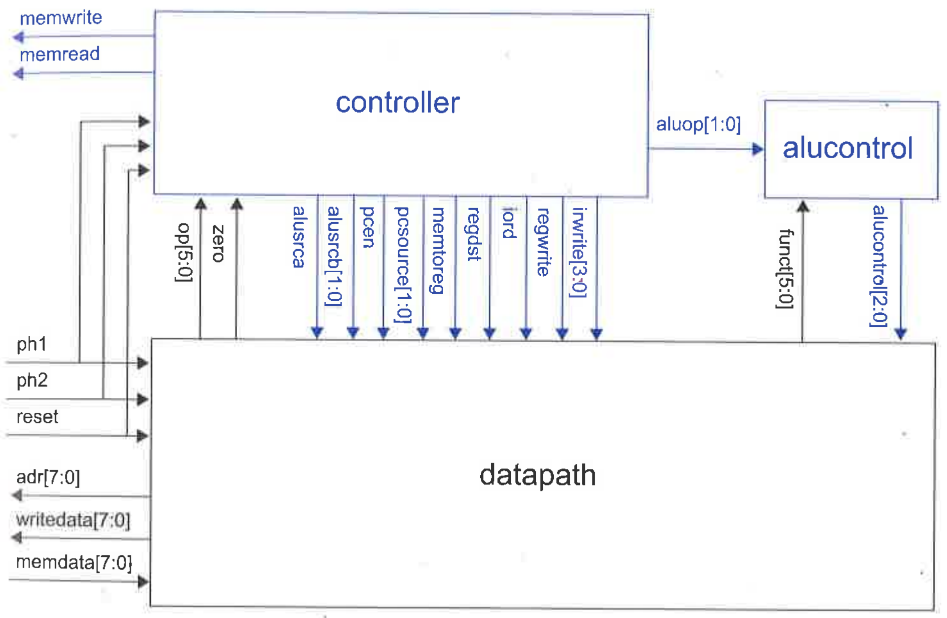
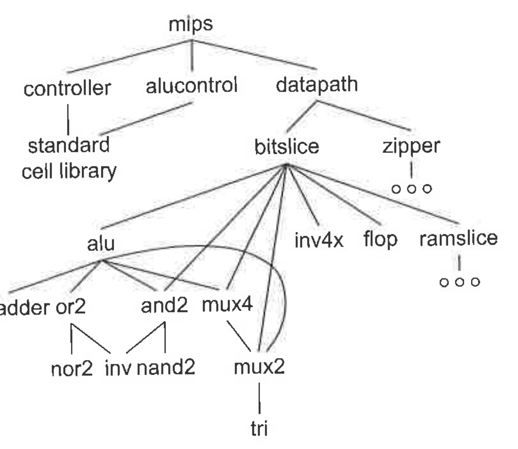


FIG 1.56 Top-level MIPS block diagram

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



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 an 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 constraints. Only a subset of HDL constructs are synthesizable; 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
```

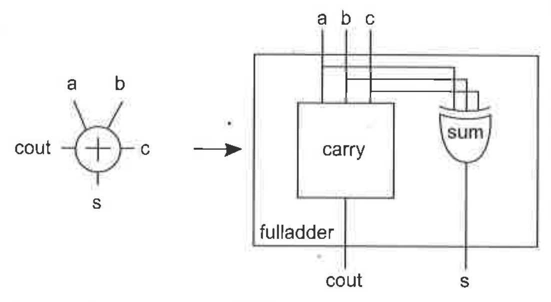


FIG 1.58 Full adder

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