Weste01.fm Page 23 Sunday, January 4, 2004 10:32 PM

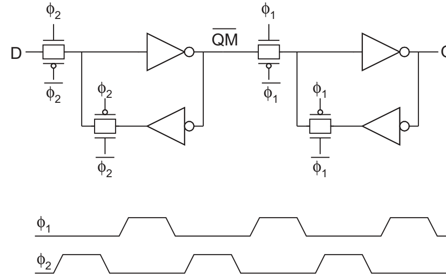

FIG 1.32 CMOS flip-flop with two-phase nonoverlapping clocks

1.5 CMOS Fabrication and Layout

Now that we can design logic gates and latches from transistors, let us consider how the transistors are built. Transistors are fabricated on thin silicon wafers that serve as both a mechanical support and an electrical common point called the *substrate*. We can understand the physical layout of transistors from two perspectives. One is the top view, obtained by looking down on a wafer. The other is the cross-section, obtained by slicing the wafer through the middle of a transistor and looking at it edgewise. We begin by looking at the cross-section of a complete CMOS inverter. We then look at the top view of the same inverter and define a set of masks used to manufacture the different parts of the inverter. The size of the transistors and wires is set by the mask dimensions and is limited by the resolution of the manufacturing process. Continual advancements in this resolution have fueled the exponential growth of the semiconductor industry.

1.5.1 Inverter Cross-section

Figure 1.33 shows a cross-section of the inverter from Section 1.4.1. In this diagram, the inverter is built on a p-type substrate. The pMOS transistor requires an n-type body region, so an n-well is diffused into the substrate in its vicinity. Note that it is also possible to design a CMOS process with an n-type substrate and p-wells to contain the nMOS transistors. As described in Section 1.3, the nMOS transistor has n-type source and drain regions and a polysilicon gate over a thin layer of silicon dioxide (SiO_2, also called *gate oxide*). The pMOS transistor is a similar structure with p-type source and drain regions. The polysilicon gates of the two transistors are tied together somewhere off the page and form the input A. The source of the nMOS transistor is connected to a metal ground line and the source of the pMOS transistor is connected to a metal V_{DD} line. The drains of the

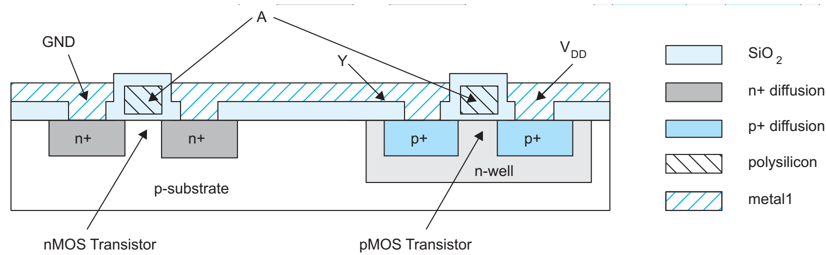

FIG 1.33 Inverter cross-section

two transistors are connected with metal to form the output *Y*. A thicker layer of SiO_2 called *field oxide* prevents metal from shorting to other layers except where contacts are explicitly etched.

The substrate must be tied to a low potential to avoid forward-biasing the p-n junction between the p-type substrate and the n+ nMOS source or drain. Likewise, the n-well must be tied to a high potential. This is generally done by adding heavily doped substrate and well contacts, or *taps*, to connect GND and V_{DD} to the substrate and n-well, respectively, as shown in Figure 1.34. The heavy doping is required to establish a good *ohmic contact* that provides low resistance for bidirectional current flow; a metal to lightly doped semiconductor junction forms a *Schottky diode*.

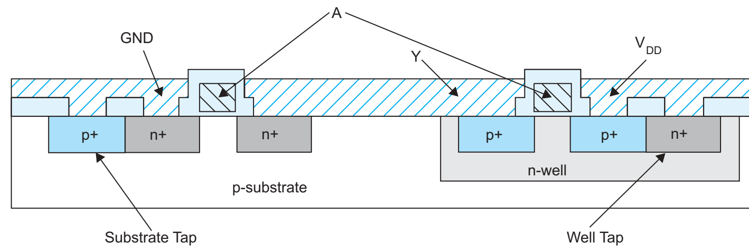

FIG 1.34 Inverter cross-section with well and substrate contacts

1.5.2 Fabrication Process

For all their complexity, chips are amazingly inexpensive because all the transistors and wires can be printed in much the same way as books. The fabrication sequence consists of

Weste01.fm Page 25 Sunday, January 4, 2004 10:32 PM

a series of steps in which layers of the chip are defined through a process called *photolithography*. Because many entire chips are printed at once, the cost of the chip is proportional to the chip area, rather than the number of transistors. As manufacturing advances allow engineers to build smaller transistors and place more transistors in the same area, each transistor gets cheaper. Smaller transistors are also faster because electrons don't have to travel as far to get from the source to the drain! This explains the remarkable trend for computers and electronics to become both cheaper and more capable with each generation.

The inverter could be defined by a hypothetical set of six masks: n-well, polysilicon, n+ diffusion, p+ diffusion, contacts, and metal (for fabrication reasons discussed in Chapter 3, the actual mask set is usually different). Masks specify where the components will be manufactured on the chip. Figure 1.35(a) shows a top view of the six masks. The cross-section of the inverter from Figure 1.34 was taken along the dashed line.

Consider a very simple fabrication process to illustrate the fundamental ideas. The process begins with the creation of an n-well on a bare p-type silicon wafer. Figure 1.36 shows cross-sections of the wafer after each processing step involved in forming the n-well; Figure 1.36(a) illustrates the bare substrate before processing. Forming the n-well requires adding enough Group V dopants into the silicon substrate to change the substrate from p-type to n-type in the region of the well. To define what regions receive n-wells, we grow a protective layer of oxide over the entire wafer, then remove it where we want the wells. We then add the n-type dopants; the dopants are blocked by the oxide, but enter the substrate and form the wells where there is no oxide. The next paragraph describes these steps in more detail.

The wafer is first *oxidized* in a high-temperature (typically 900°–1200°C) furnace that causes the Si and O_2 to react and become SiO_2 on the wafer surface (Figure 1.36(b)). The oxide must be patterned to define the n-well. An organic photoresist[1] that softens where exposed to light is spun onto the wafer (Figure 1.36(c)). The photoresist is exposed through the n-well mask (Figure 1.35(b)) that allows light to pass through only where the well should be. The softened photoresist is removed to expose the oxide (Figure 1.36(d)). The oxide is etched with hydrofluoric acid (HF) where it is not protected by the photoresist (Figure 1.36(e)), then the remaining photoresist is stripped away using a mixture of acids called *piranha etch* (Figure 1.36(f)). The well is formed where the substrate is not covered with oxide. Two ways to add dopants are diffusion and ion implantation. In the *diffusion* process, the wafer is placed in a furnace with a gas containing the dopants. When heated, dopant atoms diffuse into the substrate. Notice how the well is larger than the hole in the oxide on account of *lateral* diffusion (Figure 1.36(g)). With *ion implantation*, dopant ions are accelerated through an electric field and blasted into the substrate. In either method, the oxide layer prevents dopant atoms from entering the substrate where no well is intended. Finally, the remaining oxide is stripped with HF to leave the bare wafer with wells in the appropriate places.

[1]Engineers have experimented with many organic polymers. Brumford and Walker reported in 1958 that Jello™ could be used for masking. They did extensive testing, observing that "various Jellos™ were evaluated with lemon giving the best result."

Weste01.fm Page 26 Sunday, January 4, 2004 10:32 PM

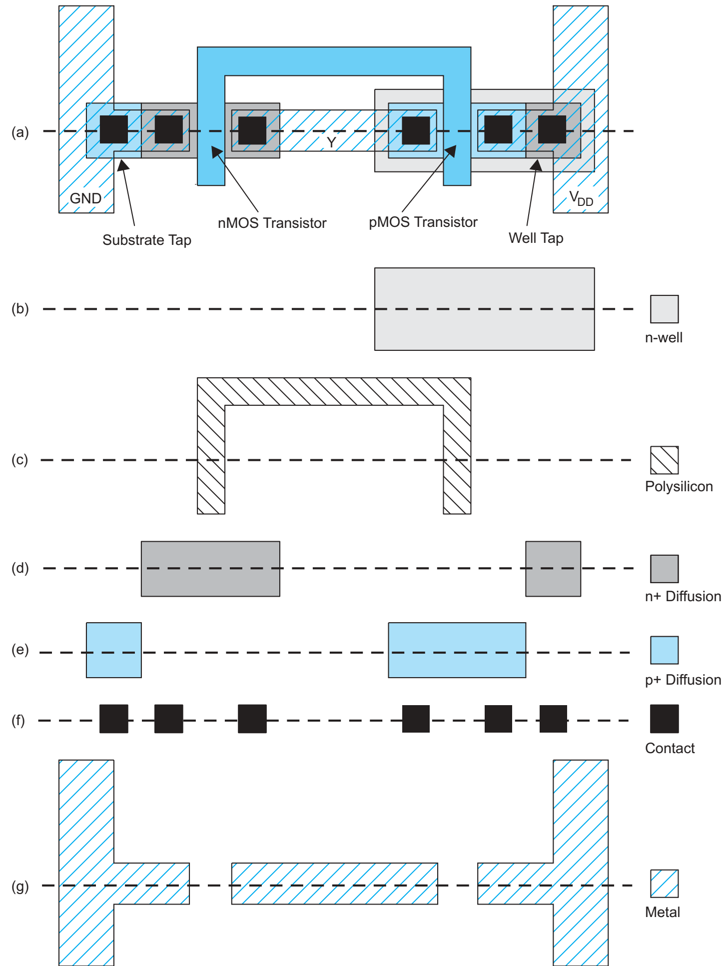

FIG 1.35 Inverter mask set

Weste01.fm Page 27 Sunday, January 4, 2004 10:32 PM

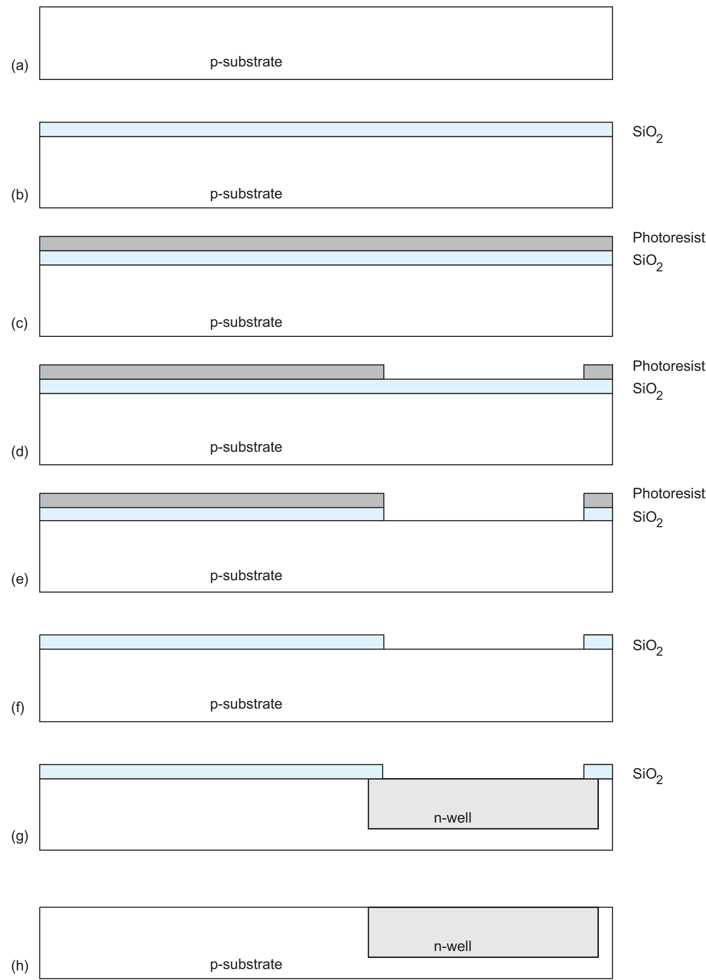

FIG 1.36 Cross-sections while manufacturing the n-well

Weste01.fm Page 28 Sunday, January 4, 2004 10:32 PM

The transistor gates are formed next. These consist of polycrystalline silicon, generally called *polysilicon*, over a thin layer of oxide. The thin oxide is grown in a furnace. Then the wafer is placed in a reactor with silane gas (SiH_4) and heated again to grow the polysilicon layer through a process called *chemical vapor deposition*. The polysilicon is heavily doped to form a reasonably good conductor. The resulting cross-section is shown in Figure 1.37(a). As before, the wafer is patterned with photoresist and the polysilicon mask (Figure 1.35(c)), leaving the polysilicon gates (Figure 1.37(b)).

The n+ regions are diffused to create the transistor active area and the well contact. As with the well, a protective layer of oxide is formed (Figure 1.37(c)) and patterned with the n-diffusion mask (Figure 1.35(d)) to expose the areas where the dopants are needed (Figure 1.37(d)). Although the n+ regions are typically formed with ion implantation (Figure 1.37(e)), they were historically diffused and thus still are often called *n-diffusion*. Notice that the polysilicon gate over the nMOS transistor blocks the diffusion so the source and drain are separated by a channel under the gate. This is called a *self-aligned* process because the source and drain of the transistor are automatically formed adjacent to the gate without the need to precisely align the masks. Finally, the protective oxide is stripped (Figure 1.37(f)).

The process is repeated for the p-diffusion mask (Figure 1.35(e)) to give the structure of Figure 1.38(a). Oxide is used for masking in the same way, and thus is not shown. The field oxide is grown to insulate the wafer from metal and patterned with the contact mask (Figure 1.35(f)) to leave contact cuts where metal should attach to diffusion or polysilicon (Figure 1.38(b)). Finally, aluminum is sputtered over the entire wafer, filling the contact cuts as well. Sputtering involves blasting aluminum into a vapor that evenly coats the wafer. The metal is patterned with the metal mask (Figure 1.35(g)) and plasma etched to remove metal everywhere except where wires should remain (Figure 1.38(c)). This completes the simple fabrication process.

Modern fabrication sequences are somewhat more elaborate because they must create complex doping profiles around the channel of the transistor and print features that are smaller than the wavelength of the light being used in lithography. However, masks for these elaborations can be automatically generated from the simple set of masks we have just examined. Modern processes may also have five or more layers of metal, so the metal and contact steps must be repeated for each layer. Chip manufacturing has become a commodity, and many different vendors will build designs from a basic set of masks.

1.5.3 Layout Design Rules

Layout design rules describe how small features can be and how closely they can be packed in a particular manufacturing process. Industrial design rules are usually specified in microns. This makes migrating from one process to a more advanced process difficult because not all rules scale in the same way. Mead and Conway [Mead80] popularized lambda-based design rules based on a single parameter, λ, which characterizes the resolution of the process. λ is generally half of the minimum drawn transistor channel length.

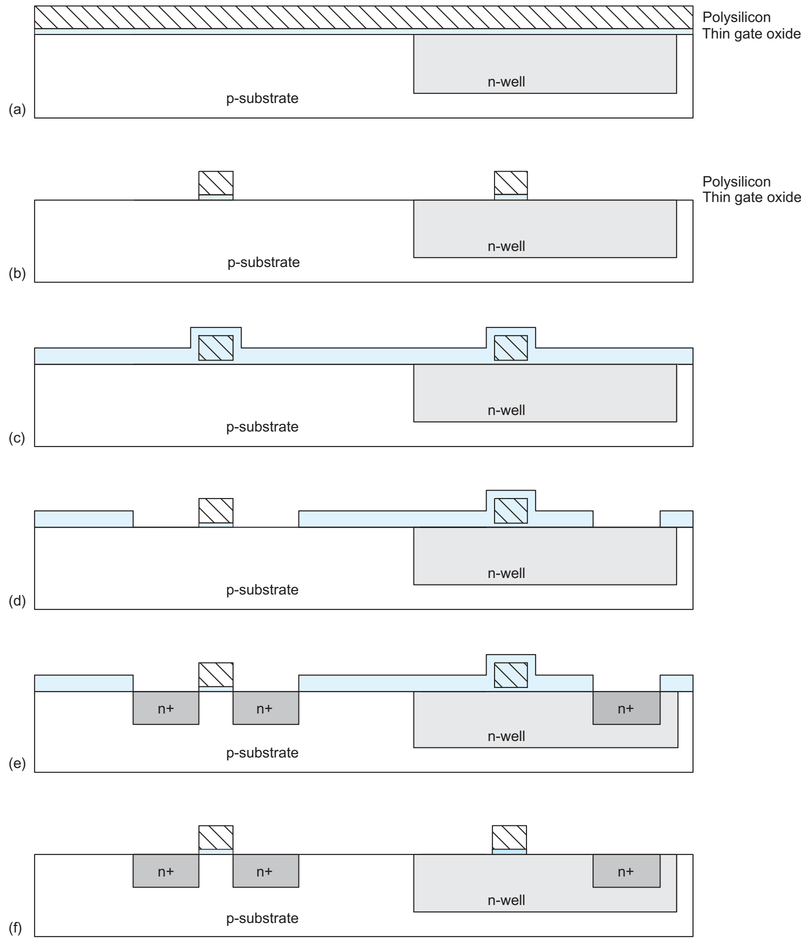

FIG 1.37 Cross-sections while manufacturing polysilicon and n-diffusion

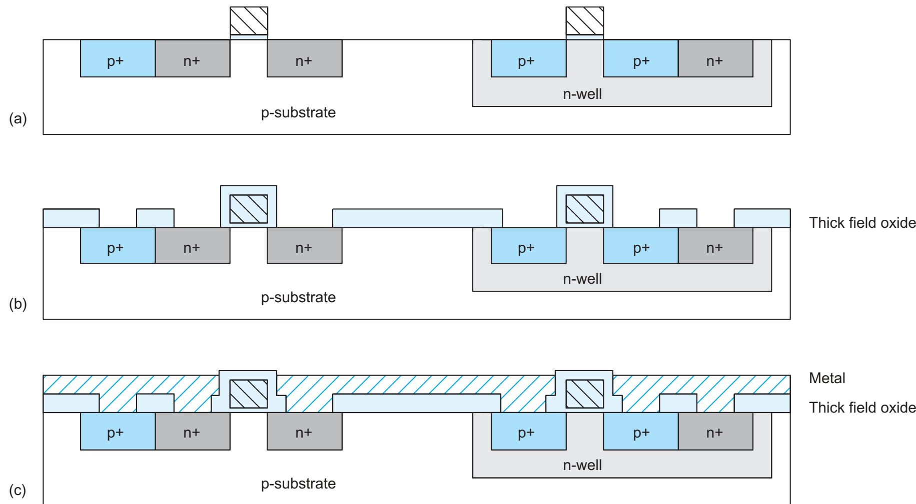

FIG 1.38 Cross-sections while manufacturing p-diffusion, contacts, and metal

This length is the distance between the source and drain of a transistor and is set by the minimum width of a polysilicon wire. This dimension is typically specified in microns for dimensions above 0.18 μm = 180 nm and in nanometers below. For example, a 180 nm process has a minimum polysilicon width (and hence transistor length) of 0.18 microns and uses design rules with λ = 0.09 μm[1]. Lambda-based rules are necessarily conservative because they round dimensions up to an integral multiple of λ. However, they make scaling layout trivial; the same layout can be moved to a new process simply by specifying a new value of λ. This chapter will present design rules in terms of λ. The potential density advantage of micron rules is sacrificed for simplicity and easy scalability of lambda rules. Designers often describe a process by its *feature size*. Feature size refers to minimum transistor length, so λ is half the feature size.

The MOSIS service [Pina02] is a low-cost prototyping service that collects designs from academic, commercial, and government customers and aggregates them onto one mask set to share overhead costs and generate production volumes sufficient to interest fabrication companies. MOSIS has developed a set of scalable lambda-based design rules

[1]Some 180 nm lambda-based rules actually set λ = 0.10 μm, then shrink the gate by 20 nm while generating masks. This keeps 180 nm gate lengths but makes all other features slightly larger.

Weste01.fm Page 31 Sunday, January 4, 2004 10:48 PM

that cover a wide range of manufacturing processes. The rules have become slightly more conservative for more advanced submicron processes, but are fundamentally very similar to those proposed by Mead and Conway in 1980 for a 6-micron process. The rules describe the minimum width to avoid breaks in a line, minimum spacing to avoid shorts between lines, and minimum overlap to ensure two layers completely overlap.

A conservative but easy-to-use set of design rules for layouts with two metal layers in an n-well process is as follows.

- Metal and diffusion have minimum width and spacing of 4 λ.
- Contacts are 2 λ × 2 λ and must be surrounded by 1 λ on the layers above and below.
- Polysilicon uses a width of 2 λ.
- Polysilicon overlaps diffusion by 2 λ where a transistor is desired and has a spacing of 1 λ away where no transistor is desired.
- Polysilicon and contacts have a spacing of 3 λ from other polysilicon or contacts.
- N-well surrounds pMOS transistors by 6 λ and avoids nMOS transistors by 6 λ.

Figure 1.39 shows the basic MOSIS design rules for a process with two metal layers. Section 3.3 elaborates on these rules and compares them with more aggressive industrial design rules.

Transistor dimensions are often specified by their Width/Length (W/L) ratio. For example, the transistors in Figure 1.39 formed where polysilicon crosses diffusion have a

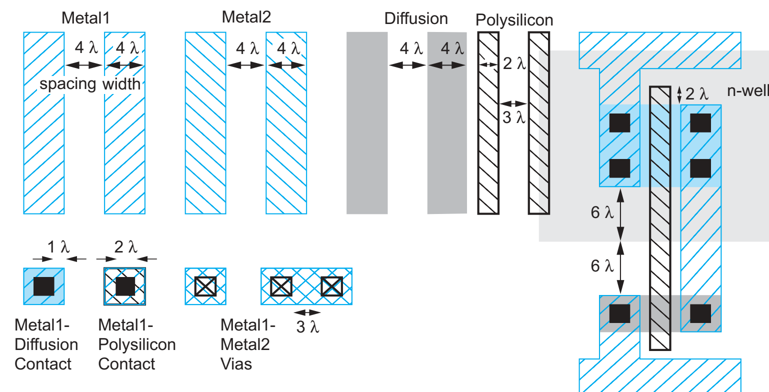

FIG 1.39 Simplified λ-based design rules

Weste01.fm Page 32 Sunday, January 4, 2004 10:32 PM

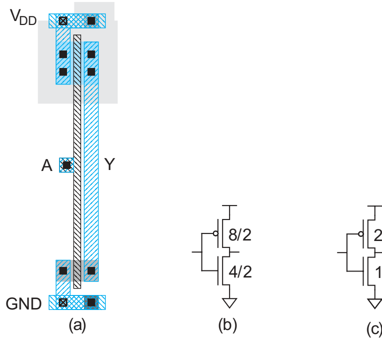

FIG 1.40 Inverter with dimensions labeled

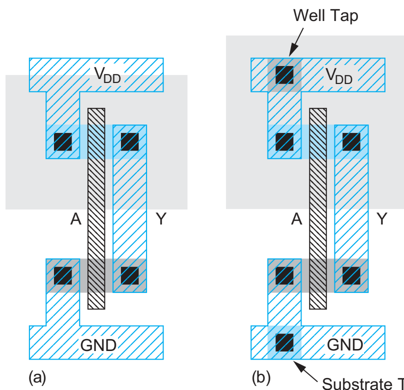

FIG 1.41 Inverter standard cell layout

W/L of 4/2. In a 0.6 μm process, this corresponds to an actual width of 1.2 μm and a length of 0.6 μm. Such a minimum-width contacted transistor is often called a unit transistor.[1] pMOS transistors are often wider than nMOS transistors because holes move more slowly than electrons so the transistor has to be wider to deliver the same current. Figure 1.40(a) shows a unit inverter layout with a unit nMOS transistor and a double-sized pMOS transistor. Figure 1.40(b) shows a schematic for the inverter annotated with Width/Length for each transistor. Figure 1.40(c) shows a shorthand we will often use, specifying multiples of unit width and assuming minimum length.

1.5.4 Gate Layout

A good deal of ingenuity can be exercised and a vast amount of time wasted exploring layout topologies to minimize the size of a gate or other *cell* such as an adder or memory element. For many applications, a straightforward layout is good enough and can be automatically generated or rapidly built by hand. This section presents a simple layout style based on a "line of diffusion" rule that is commonly used for standard cells in automated layout systems. This style consists of four horizontal strips: metal ground at the bottom of the cell, n-diffusion, p-diffusion, and metal power at the top. The power and ground lines are often called *supply rails*. Polysilicon lines run vertically to form transistor gates. Metal wires within the cell connect the transistors appropriately.

Figure 1.41(a) shows such a layout for an inverter. The input *A* can be connected from the top, bottom, or left in polysilicon. The output *Y* is available at the right side of the cell in metal. Recall that the p-substrate and n-well must be tied to ground and power, respectively. Figure 1.41(b) shows the same inverter with well and substrate taps placed under the power and ground rails, respectively. Figure 1.42 shows a 3-input NAND gate. Notice how the nMOS transistors are connected in series while the pMOS transistors are connected in parallel. Power and ground extend 2 λ on each side so if two gates were abutted the contents would be separated by 4 λ, satisfying design rules. The height of the cell is 36 λ, or 40 λ if the 4 λ space between the cell and another wire above it is counted. All these examples use transistors of width 4 λ. Choice of transistor width is addressed further in Chapter 4.

[1] Such small transistors in modern processes often behave slightly differently than their wider counterparts. Moreover, the transistor will not operate if either contact is damaged. Industrial designers often use a transistor wide enough for two contacts (9 λ) as the unit transistor to avoid these problems.

Weste01.fm Page 33 Sunday, January 4, 2004 10:32 PM

These cells were designed such that the gate connections are made from the top or bottom in polysilicon. In contemporary standard cells, polysilicon is generally not used as a routing layer so the cell must allow metal2 to metal1 and metal1 to polysilicon contacts to each gate. While this increases the size of the cell, it allows free access to all terminals on metal routing layers.

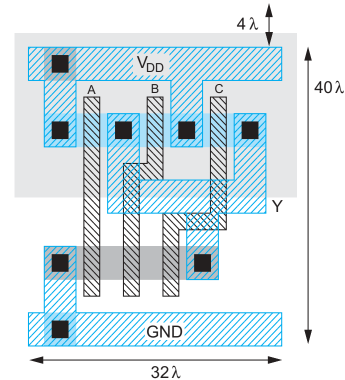

FIG 1.42 3-input NAND standard cell gate layouts

1.5.5 Stick Diagrams

As layout is time-consuming, it is important to have fast ways to plan layout and estimate area before committing to a full layout. *Stick diagrams* are easy to draw because they do not need to be drawn to scale. Figure 1.43 shows stick diagrams for an inverter and a 3-input NAND gate. While this book uses stipple patterns, layout designers use colored pencils or dry-erase markers.

With practice it is easy to estimate the area of a layout from the corresponding stick diagram even though the diagram is not to scale. Although schematics focus on transistors, layout area is usually determined by the metal wires. Transistors are merely small widgets that fit under the wires. We define a *routing track* as enough space to place a wire and the required spacing to the next wire. If our wires have a width of 4 λ and a spacing of 4 λ to the next wire, the track *pitch* is 8 λ, as shown in Figure 1.44(a). This pitch also leaves room for a transistor to be placed between the wires (Figure 1.44(b)). Therefore, as a rule of thumb, it is reasonable to estimate the height and width of a cell by counting the number of metal tracks and multiplying by 8 λ. A slight complication is the required spacing of 12 λ between nMOS and pMOS transistors set by the well, as shown in Figure 1.45(a). This space can be occupied by an additional track of wire, shown in Figure 1.45(b). Therefore, an extra track must be allocated between nMOS and pMOS transistors regardless of whether wire is actually used in that track. Figure 1.46 shows how to count tracks to estimate the size of a 3-input NAND. There are four vertical wire tracks, multiplied by 8 λ per track to give a cell width of 32 λ. There are five horizontal tracks, giving a cell height

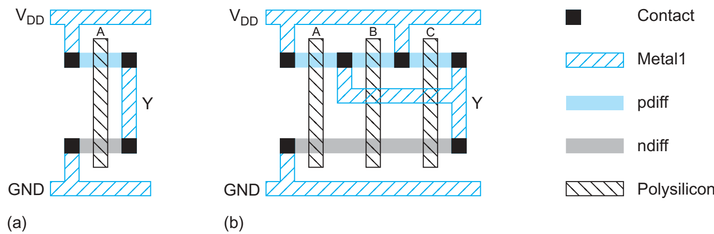

FIG 1.43 Stick diagrams of inverter and 3-input NAND gate

Weste01.fm Page 34 Sunday, January 4, 2004 10:32 PM

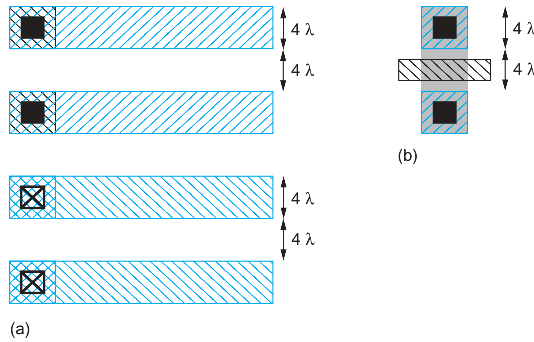

FIG 1.44 Pitch of routing tracks

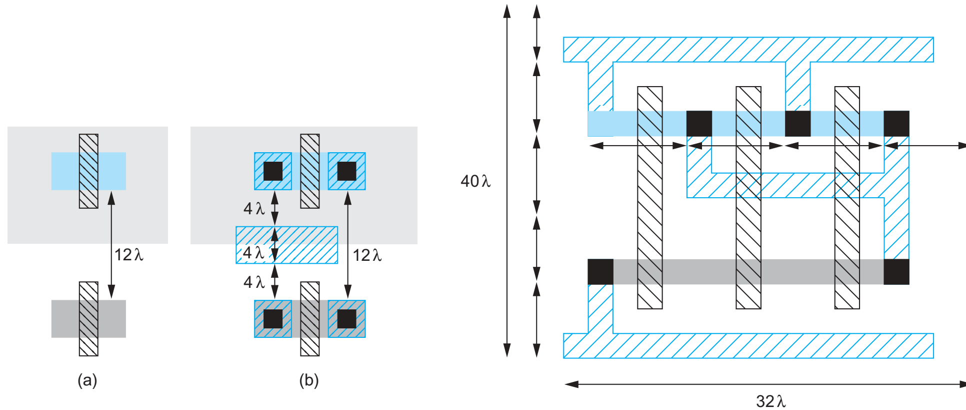

FIG 1.45 Spacing between nMOS and pMOS transistors

FIG 1.46 3-input NAND gate area estimation

of 40 λ. Even though the horizontal tracks are not drawn to scale, they are still easy to count. Figure 1.42 shows that the actual NAND gate layout agrees in size if the 4 λ spacing between the top of V_{DD} and the next place a metal line could be placed is counted. If transistors are wider than 4 λ, the extra width must be factored into the area estimate. Of course, these estimates are oversimplifications of the complete design rules and a trial layout should be performed for truly critical cells.

Weste01.fm Page 35 Sunday, January 4, 2004 10:48 PM

Example

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

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

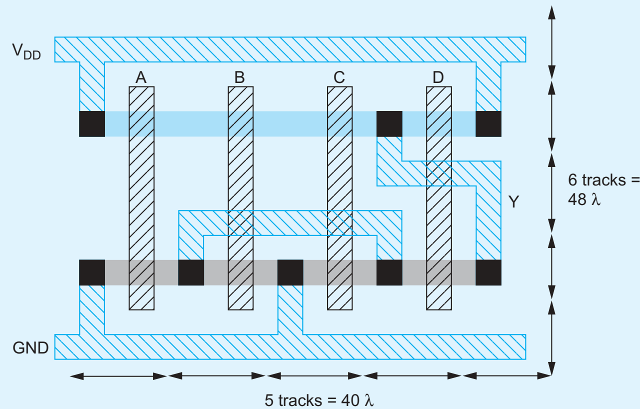

FIG 1.47 CMOS compound gate for function $Y = \overline{(A + B + C) \bullet D}$

1.6 Design Partitioning

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

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

Weste01.fm Page 36 Sunday, January 4, 2004 10:32 PM

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

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

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

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

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

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

Weste01.fm Page 37 Sunday, January 4, 2004 10:32 PM

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

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

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

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

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

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

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

Weste01.fm Page 38 Sunday, January 4, 2004 10:32 PM

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.

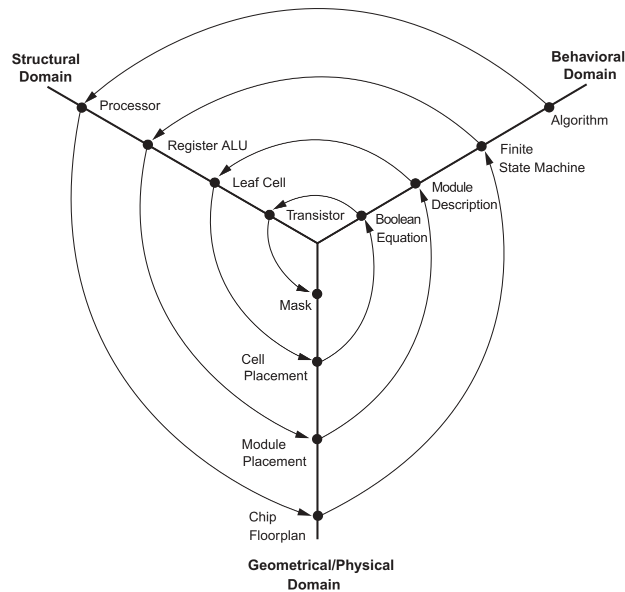

FIG 1.48 Y Diagram

Weste01.fm Page 39 Sunday, January 4, 2004 10:32 PM

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,`	add immediate:	$1→ $2 + imm	I	001000	n/a
`beq $1, $2, imm`	branch if equal:	PC → PC + imm[a]	I	000100	n/a
`j destination`	jump:	PC_destination[a]	J	000010	n/a
`lb $1, imm($2)`	load byte:	$1 → mem[$2 + imm]	I	100000	n/a
`sb $1, imm($2)`	store byte:	mem[$2 + imm] → $1	I	110000	n/a

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

Weste01.fm Page 40 Sunday, January 4, 2004 10:32 PM

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 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.

continues

Weste01.fm Page 41 Sunday, January 4, 2004 10:32 PM


```
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	10600005
add $4, $4, $5	000000 00100 00101 00100 00000 100000	00852020
sub $5, $4, $5	000000 00100 00101 00101 00000 100010	00852822
addi $3, $3, -1	001000 00011 00011 1111111111111111	2063ffff
j loop	000010 00000000000000000000000011	08000003
sb $4, 255($0)	110000 00000 00100 0000000011111111	a00400ff

FIG 1.52 Machine language code for Fibonacci program

1.7.2 Multicycle MIPS Microarchitecture

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

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

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

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

Table 1.8 ALUControl determination

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

Weste01.fm Page 43 Sunday, January 4, 2004 10:32 PM

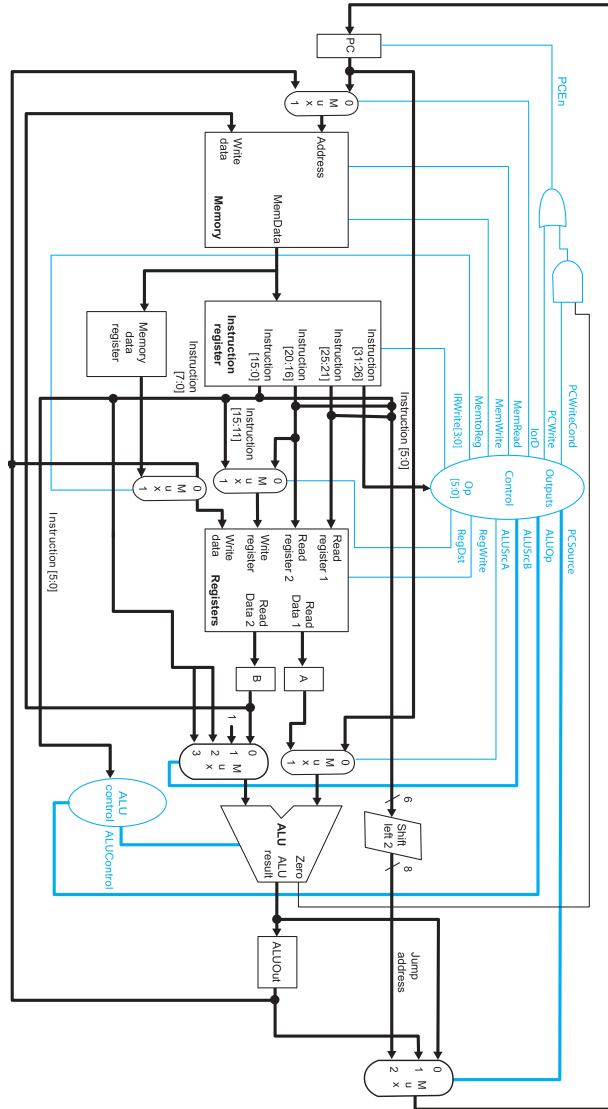

FIG 1.53 Multicycle MIPS microarchitecture

Weste01.fm Page 44 Sunday, January 4, 2004 10:32 PM

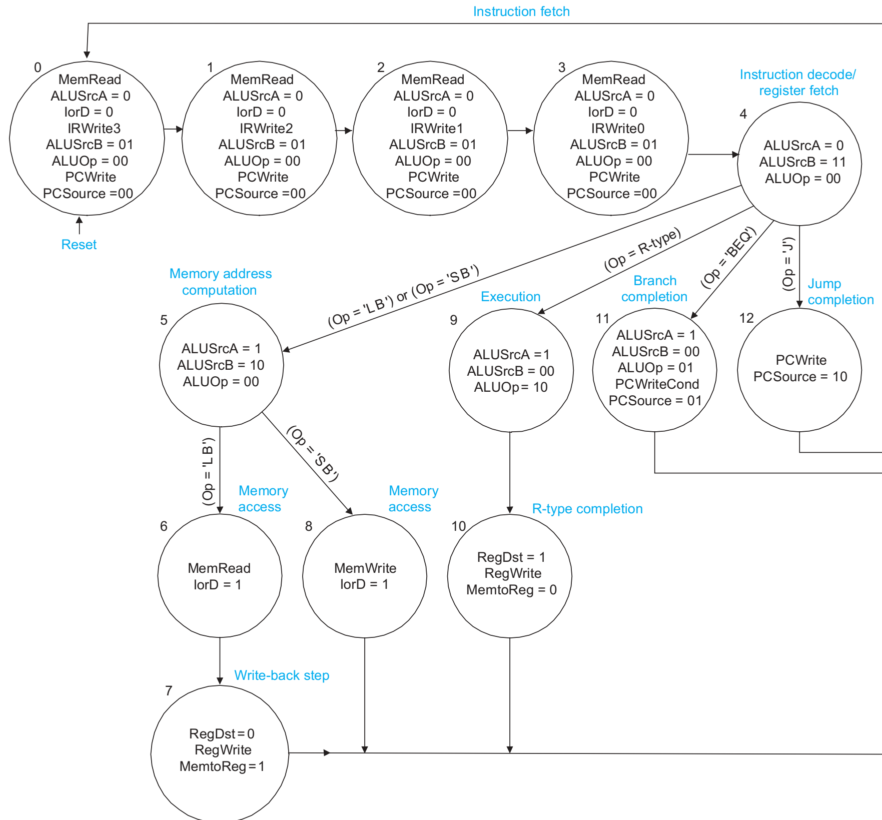

FIG 1.54 Multicycle MIPS control FSM

Weste01.fm Page 45 Sunday, January 4, 2004 10:32 PM

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.