7 PROCESSOR BASICS

In this chapter we start our focus on embedded systems with an introduction to the kinds of processors that are used. We describe the way processors operate and give examples of the instructions that make up embedded software programs. We also describe the way instructions and data are encoded in binary and stored in memory. Finally, we examine ways of connecting the processor with memory components.

7.1 EMBEDDED COMPUTER ORGANIZATION

In Section 1.5.1, we introduced the idea of an embedded system, in which one or more computers form part of the system. The computers run programs that implement the functions required of the system. Unlike a general-purpose PC, a computer in an embedded system has just those resources required to support its specialized operation. In this section, we will describe some of the general properties of embedded systems and the processing elements they contain. We won't deal with how the processing elements are designed; that is a significant field of study in its own right. Instead, we will treat them as black-box circuit components that we can use to build a digital system.

A computer embedded in a digital system generally contains the elements shown in Figure 7.1. The *central processing unit* (CPU), often called a *processor core* when it is embedded as part of an IC, is the element that processes data according to a program. The kinds of processing it can perform include the arithmetic operations that we described in Chapter 3. It can also evaluate logical conditions and select among alternate operations based on the outcomes of the conditions. We will describe the way a program is formed in more detail in Section 7.2. Meanwhile, suffice it to say that the program is encoded in binary form and stored in the *instruction memory* shown in the figure. The data upon which the program operates are also encoded in binary form and stored in the *data*

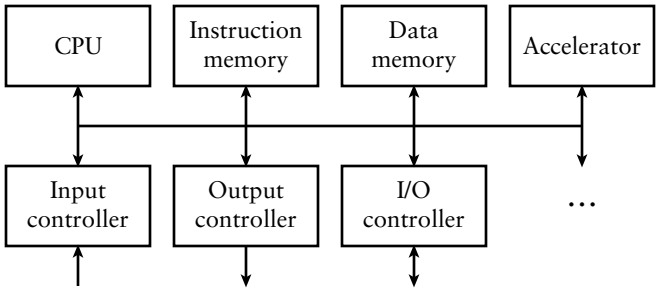


FIGURE 7.1 Elements of an embedded computer.

memory. In both cases, the memory is implemented using the kinds of memory components we described in Chapter 5. Whereas general purpose computers, such as PCs, usually store the instructions and data in the same memory, embedded computers typically separate the two. (This arrangement is often referred to as a *Harvard architecture*, named after the institution where the idea originated. The conventional approach with a single memory for instructions and data is called a *von Neumann architecture*, after the person who first described it.) The reason for the separation is that the instructions in an embedded computer are usually fixed during the manufacture of the system (or only occasionally upgraded in the field), and the amount of instruction memory required is known in advance. Hence, we usually store instructions in a ROM or flash memory component, and provide a RAM for the data memory. This differs from a general-purpose computer, in which one or more different programs need to be started at different times and run concurrently, and the amount of instruction memory is not known in advance.

The *input*, *output* and *input/output* (I/O) controllers in Figure 7.1 allow the computer to acquire data to be processed (input) and to deliver the results (output). In many embedded systems, the input data comes from sensors that sample physical properties, such as temperature, position, time, and so on. Similarly, the output data causes actuators to have a physical effect, such as moving a lever, turning a motor, heating some material, and so on. Input and output controllers can also deal with a user interface, consisting of switches, buttons and knobs for input and lights and LCD panels for outputs. For a complex user interfaces, devices such as a keyboard, mouse or display screen, as used in a general purpose computer, might also be employed. In all cases, the job of the input/output controller is to transform between a physical property or effect and a corresponding binary representation that can be processed by the CPU. We will describe how this can be done and how the CPU accesses the binary representation in Chapter 8.

The *accelerator* in Figure 7.1 is a specialized circuit designed to implement specific processing operations with higher performance than can be achieved using the CPU. Not all embedded systems include

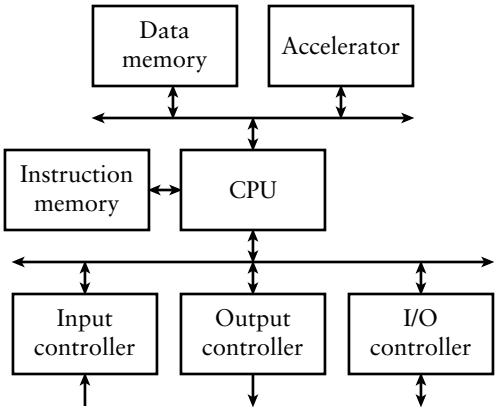


FIGURE 7.2 Organization of a high-performance embedded computer with multiple buses: one for the instruction memory, one for the data memory and an accelerator, and one for input/output controllers.

accelerators. The choice of whether to include an accelerator for any operation depends on the functional and performance requirements of the application, together with cost and other constraints that apply. We will discuss accelerators in more detail in Chapter 9, in which we include as an extended example an accelerator for detecting edges of objects in video images.

The final element in Figure 7.1 is the interconnection between the other elements. We use the term *bus* to refer to the collection of signals that form the interconnection. The figure shows just one bus connecting all of the elements. However, in more elaborate systems, there may be separate buses for connecting the memory and the input/output controllers with the CPU. There may even be separate buses for the instruction and data memories, since many high-performance processors can read further instructions concurrently with access to data by previous instructions. Accelerators, if included, might be connected to the CPU using the same bus as the memory, or using a separate dedicated bus. Figure 7.2 shows one possible organization for a high-performance embedded system with multiple buses. In this chapter, we will focus on the bus connecting the CPU and memory, and defer consideration of bus connections to input and output controllers and to accelerators until later chapters.

7.1.1 MICROCONTROLLERS AND PROCESSOR CORES

CPUs for embedded systems come in a range of sizes for different applications. Some are single-chip *microprocessors*, consisting of a CPU by itself in a package. Most CPUs used in general-purpose PCs are also available in versions suitable for embedded applications. Examples include Pentium family CPUs from Intel and the PowerPC from Freescale Semiconductor. Other microprocessors are designed specifically for embedded applications.

In both cases, we need to provide memory and I/O controllers as separate chips on a PCB. In contrast, single-chip *microcontrollers* include a CPU, instruction and data memory, and I/O controllers all in the one package. Many microcontroller vendors provide a family of chips, each with the same CPU, but varying in the amount of memory and the selection of I/O controllers. In some microcontroller families, the CPUs are relatively simple, operating just on 8-bit or 16-bit data, with relatively low performance. Other families have more complex CPUs that can operate on data up to 32 bits in length. The combination of a CPU with the on-chip memory and I/O controllers makes them suitable for a large range of cost-sensitive, low-performance applications.

An alternative to using a fixed function microprocessor or microcontroller is to include a CPU in an FPGA component. This has the advantage that the input/output controllers can be customized for an application, but still be included in the same package as the CPU. The CPU in the FPGA can be implemented as a fixed-function block embedded within the programmable fabric. The Virtex-II Pro and Virtex 4 FPGAs from Xilinx take this approach, and include one or more PowerPC processor cores. Alternatively, the CPU can be implemented as a *soft core* using the programmable resources of the FPGA. FPGA vendors provide soft core processor designs that users can include as part of their system. Examples include the MicroBlaze core from Xilinx, the Nios-II core from Altera, and the ARM core from Actel. These are all relatively high-performance CPUs that operate on data up to 32 or 64 bits in length. For simpler designs, a smaller soft core that operates on 8-bit data may suffice. It would take up less of the FPGA resources, and would fit in a smaller and cheaper FPGA component. The Xilinx PicoBlaze soft core is an example, as is the Gumnut core that we will introduce in Section 7.2.

If our design is implemented in an ASIC, we can also include a CPU and customized memory and input/output controllers. Several vendors provide processor core designs that can be included as blocks in ASICs. Among the most widely used are the ARM cores from ARM Ltd, the PowerPC cores from IBM, and the MIPS cores from MIPS Technologies. Given that we can customize the design on an ASIC, there is also opportunity to customize the CPU itself. Tensilica Inc. is a vendor that provides a customizable CPU based on the requirements of the program to be executed. Their approach involves analyzing the program and including only the CPU features needed to execute that program. They also allow extension of the CPU with customized hardware for specialized operations.

A final approach to mention is to include one or more *digital signal processors* (DSPs). These are specialized processing elements optimized for the kinds of operations involved in dealing with digitized signals, such as audio, video or other streams of data from sensors. Many signal processing applications require fixed-point or floating-point arithmetic operations to be performed at a high rate on large volumes of data. An ordinary CPU

would not be able to meet the performance requirements. Nonetheless, such applications often need a conventional CPU to perform other operations, such as interacting with the user and overall coordination of system operation. Hence, DSPs are often combined with conventional CPUs in heterogeneous *multiprocessor* systems. Modern cell phones are good examples. Another approach to providing DSP functionality is to extend a conventional CPU with additional hardware and instructions for digital signal processing. Some processor cores from ARM and MIPS include such extensions, and Tensilica processor cores can be similarly customized. Since digital signal processing is an advanced topic, we will defer consideration of DSP cores and embedded multiprocessor systems to advanced reference books.

KNOWLEDGE TEST QUIZ

1. What are the main elements of an embedded computer?
2. Why do embedded computers usually have separate instruction and data memories?
3. What is the difference between a microprocessor and a microcontroller?
4. What is meant by a *soft core* processor in an FPGA?

7.2 INSTRUCTIONS AND DATA

The function performed by a CPU is specified by a program, which consists of a sequence of instructions. Each instruction specifies one simple step in the program, such as getting a piece of data from memory, or adding two numbers. The repertoire of instructions for a given CPU is called the *instruction set* of the CPU. We also use the term *instruction set architecture* (ISA) to refer to the combination of the instruction set and other aspects of the CPU that are visible to the programmer. CPUs from different vendors have quite significantly different instruction sets, so a sequence of instructions developed for one CPU will not work on a CPU from a different vendor. When we develop the program for an application, we usually use a *high-level language*, such as C, C++ or Ada, and use a software tool called a *compiler* to translate the program into a sequence of instructions that performs the same operations. Apart from allowing us to work at a higher level of abstraction, this has the advantage that the program can be ported to work on a CPU with a different instruction set simply by using a different translator. However, when we are developing an embedded system in which the CPU interacts with circuits that we design, we often need to monitor the instruction-by-instruction operation of the CPU as we test and debug the design. At this level, it is important to understand how a CPU represents and processes individual instructions. We will just describe CPU operation at this level, and defer a discussion of programming using high-level languages to other books.

The instructions of a program are encoded in binary and stored in successive locations of the instruction memory. The CPU *executes* the program by repeatedly following these steps:

1. *Fetch* the next instruction from the instruction memory.
2. *Decode* the instruction to determine the operation to perform.
3. *Execute* the operation.

In order to keep track of which instruction to fetch next, the CPU has a special register called the program counter (PC), in which the address of the next instruction is kept. In the fetch step, the CPU uses the contents of the PC to do a read access from the instruction memory, and then increments the PC value. In the decode step, the CPU determines the resources required to perform the operation specified by the instruction. In a simple CPU, the decode step is correspondingly simple. In a larger CPU, however, decoding may involve such actions as checking for resource conflicts and availability of data, and waiting until resources are free. In the execute step, the CPU activates the appropriate internal resources to perform the operation. This involves setting control signals to make multiplexers supply the required operands and arithmetic hardware perform the required operation, and enabling registers to receive results. In a simple CPU, these steps are performed in order, and when the execute step is finished, the CPU starts again with the fetch step. More complex, high performance CPUs, however, can overlap the steps, provided they produce the same outcome as if the steps were performed in order. Techniques used within CPUs to execute several instructions in parallel include *pipelining* and *superscalar* execution, described in the reference book on computer architecture (see Section 7.5).

The data on which instructions operate is encoded in binary in fixed-size quantities. The smallest data item is usually 8 bits, called a *byte*. It is often used to represent an unsigned or a signed integer, or a character. Simple CPUs can only operate on 8-bit data, so they are referred to as 8-bit CPUs. Larger CPUs can operate on 16-bit or 32-bit *words* of data, as well as on 8-bit data, so they are referred to as 16-bit or 32-bit CPUs, respectively.

Regardless of the sizes of data that can be operated upon, the data memory is usually organized with 8-bit locations, each separately addressed. 16-bit or 32-bit data is stored in two or four successive locations. The order of the bytes within a word varies between CPUs, as shown in Figure 7.3. *Little-endian* CPUs store the byte containing the least significant bits at the lower address and the byte containing the most significant bits at the higher address. In contrast, *big-endian* CPUs store the bytes in the opposite order. (The terms “little endian” and “big endian” originated in Jonathan Swift’s *Gulliver’s Travels*, in which the people of two countries fight over which end of their breakfast eggs should be cut open.

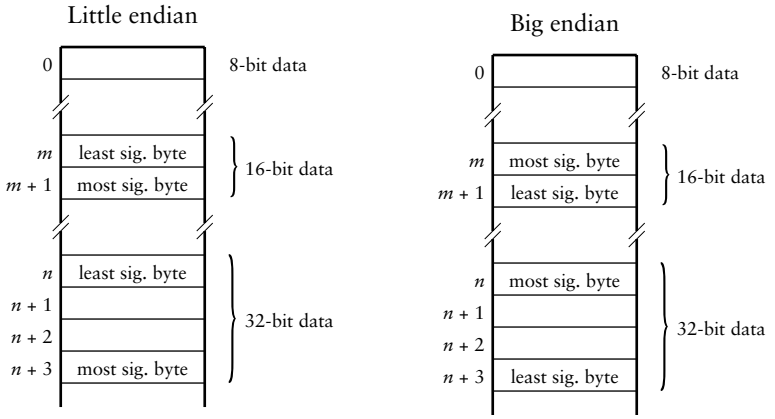


FIGURE 7.3 Little-endian (left) and big-endian (right) memory layout for data words.

The terms were adopted by Danny Cohen in an article, cited in Section 7.5, in which he argues that either byte ordering is acceptable, provided it is used consistently.) Some CPUs require that 16-bit data be stored at even addresses and that 32-bit data be stored at addresses that are a multiple of four. Others allow 16-bit and 32-bit data to be stored at any address.

7.2.1 THE GUMNUT INSTRUCTION SET

Rather than trying to describe the characteristics of the instruction sets of all CPUs, we will present one relatively simple example that embodies most of the important concepts. The CPU that we will describe is an 8-bit soft core called the *Gumnut*, developed by the author. (A gumnut is a small seedpod of an Australian eucalyptus tree. It is something small from which large things grow.) Further information and files are provided in the supplementary material for this book for use in FPGA designs. The complete Gumnut instruction set is listed in Table 7.1. We use a notation for instructions called *assembly code*. An assembly-code program can be translated by a software tool called an *assembler* into a sequence of binary-coded instructions to be loaded into the instruction memory.

The Gumnut has an instruction memory of up to 4096 instructions (using 12-bit addresses) and a data memory of 256 bytes (using 8-bit addresses). When the CPU is reset, it clears the PC to 0, and starts the fetch-decode-execute cycle, fetching the first program instruction from address 0 in the instruction memory. Within the CPU, there are eight general-purpose registers, named r0 through r7, that can hold data to be operated upon by instructions. Register r0 is special, in that it is hard-wired to have the value 0, and any updates to it are ignored. The CPU also has two single-bit *condition-code* registers called Z (zero) and

INSTRUCTION	DESCRIPTION
Arithmetic and logical instructions	
add *rd*, *rs*, *op2*	Add *rs* and *op2*, result in *rd*
addc *rd*, *rs*, *op2*	Add *rs* and *op2* with carry, result in *rd*
sub *rd*, *rs*, *op2*	Subtract *op2* from *rs*, result in *rd*
subc *rd*, *rs*, *op2*	Subtract *op2* from *rs* with carry, result in *rd*
and *rd*, *rs*, *op2*	Logical AND of *rs* and *op2*, result in *rd*
or *rd*, *rs*, *op2*	Logical OR of *rs* and *op2*, result in *rd*
xor *rd*, *rs*, *op2*	Logical XOR of *rs* and *op2*, result in *rd*
mask *rd*, *rs*, *op2*	Logical AND of *rs* and NOT *op2*, result in *rd*
Shift instructions	
shl *rd*, *rs*, *count*	Shift *rs* value left *count* places, result in *rd*
shr *rd*, *rs*, *count*	Shift *rs* value right *count* places, result in *rd*
rol *rd*, *rs*, *count*	Rotate *rs* value left *count* places, result in *rd*
ror *rd*, *rs*, *count*	Rotate *rs* value right *count* places, result in *rd*
Memory and I/O instructions	
ldm *rd*, (*rs*) ± *offset*	Load to *rd* from memory
stm *rd*, (*rs*) ± *offset*	Store to memory from *rd*
inp *rd*, (*rs*) ± *offset*	Input to *rd* from input controller register
out *rd*, (*rs*) ± *offset*	Output to output controller register from *rd*
Branch instructions	
bz ± *disp*	Branch if Z is set
bnz ± *disp*	Branch is Z is not set
bc ± *disp*	Branch if C is set
bnc ± *disp*	Branch if C is not set
Jump instructions	
jmp *addr*	Jump to *addr*
jsb *addr*	Jump to subroutine at *addr*
Miscellaneous instructions	
ret	Return from subroutine
reti	Return from interrupt
enai	Enable interrupts
disi	Disable interrupts
wait	Wait for interrupts
stby	Enter low-power standby mode

TABLE 7.1 The Gumnut instruction set. *rd* and *rs* are registers, *op2* is a register (*rs2*) or an immediate value (*immed*), *count* is count of number of places to shift or rotate, *disp* is a displacement from the next-instruction address, and *addr* is a jump target address.

C (carry). They are set to 1 or cleared to 0 depending on the result of certain instructions, and can be tested to decide among alternative courses of action in the program.

Arithmetic and Logical Instructions

The arithmetic and logical instructions operate on 8-bit data values stored in the CPU's general-purpose registers and store the result in the destination register, *rd*. For each instruction, one value is taken from a source register, *rs*. The other value, *op2*, either comes from a second source register (*rs2*) or is an *immediate value* (*immed*). An immediate value is a value that is specified as part of the instruction, rather than being stored in a register or in memory. For example, the instruction

```
add r3, r4, r1
```

adds the values currently in registers r4 and r1 and puts the result in r3. Similarly, the instruction

```
add r5, r1, 2
```

adds the immediate value 2 and the value currently in r1 and puts the result in r5. Note that the destination register can be the same as a source register. For example, the instruction

```
sub r4, r4, 1
```

updates register r4 by decrementing its value.

The addition and subtraction instructions treat the data values as 8-bit unsigned integers. The `addc` instruction includes the value of the C condition code as a carry-in bit, and the `subc` instruction includes the C value as a borrow-in bit. All of the instructions in this group modify the Z and the C bits. They set Z to 1 if the instruction result is 0, and they clear Z to 0 if the result is nonzero. The `add` and `addc` instructions set C to the carry-out bit of the addition, the `sub` and `subc` instruction set C to the borrow out of the subtraction, and the remaining logical instructions clear C to 0. We will see later in this section how the condition-code bits are used by branch instructions.

EXAMPLE 7.1 Write a sequence of instructions to evaluate the expression $2x + 1$, assuming the value of x is in register r3 and the result is to be put in r4.

SOLUTION We can multiply x by 2 by adding it to itself. The required instructions are

```
add r4, r3, r3
add r4, r4, 1
```

EXAMPLE 7.2 Write a sequence of instructions that sets the Z bit to 1 if the least significant 4 bits of r2 have the value 0101.

SOLUTION We can test whether a register value is equal to 0101 by subtracting 0101 from the value and putting the result in r0. The result value is ignored, but Z is set as a side-effect of the subtraction. However, the most significant 4 bits of r2 might contain 1s that we are not interested in, so we need to clear them to 0s before doing the subtraction. We can use an AND operation with the value 00001111 to clear the bit. The required instructions are:

```
and r1, r2, 0x0F
sub r0, r1, 0x05
```

The notation "0x" is a prefix for a hexadecimal value in the Gumnut assembly code notation. Thus, 0x0F is the value 00001111 and 0x05 is the value 00000101.

Shift Instructions

The shift instructions shift or rotate 8-bit values taken from the general purpose register *rs* and store the result in register *rd*. The number of places to shift or rotate is specified in the instruction as *count*. For example, the instruction

```
shl r4, r1, 3
```

reads the value currently in register r1, shifts it left by 3 places and puts the result in r4. The shift-left and shift-right instructions discard the bits shifted past the end of the 8-bit byte and fill the vacated bit positions with 0s. The rotate-left and rotate-right instructions copy the bits shifted past the end of the byte around to the other end. All of these instructions set Z to 1 if the

instruction result is 0, and they clear Z to 0 if the result is nonzero. They set the C bit to the value of the last bit shifted past the end of the byte.

EXAMPLE 7.3 Write instructions that multiply the value in r4 by 8, ignoring the possibility of overflow.

SOLUTION Recall from Section 3.1.2 that we can multiply an unsigned binary integer by 2^k by shifting k places to the left. Thus, since $8 = 2^3$, an instruction to multiply r4 by 8 is

```
shl r4, r4, 3
```

Memory and Input/Output Instructions

The Gumnut has separate instructions for accessing data memory and I/O controllers. We will discuss the operation of I/O controllers in detail in Chapter 8. For now, we simply point out that I/O controllers have registers that govern their operation, and that these registers can be read and written by the CPU. Just as locations in memory have addresses, each I/O controller register has an identifying address. The Gumnut uses 8-bit addresses for I/O controller registers, distinct from the 8-bit addresses it uses for locations in the data memory. We say that the Gumnut has separate *address spaces* for data memory and for I/O controller registers. This is in contrast to a number of other CPU instructions sets, in which I/O controller registers are part of the same address space as memory addresses. In those instruction sets, we say I/O registers are *memory mapped*.

For all of the Gumnut's memory and I/O instructions, the address to access is computed by adding the current value in *rs* and an offset value specified in the instruction. The load from memory instruction reads from the data memory at the computed address and puts the read value in register *rd*. The store to memory writes the value from register *rd* to the data memory at the computed address. The input and output instructions perform similar operations, but read or write to the I/O controller registers at the computed address. None of these instructions affect the values of the Z and C bits. As examples, the instruction

```
ldm r1, (r2)+5
```

calculates the memory address by adding the current value of r2 and the offset 5. It then reads from memory at that address and puts the read value in r1. Similarly, the instruction

```
stm r1, (r4)-2
```

stores the value from r1 into memory at the address 2 less than the current value of r4.

If we want to specify a particular address to access, we can use r0 as the register for *rs*. Recall that r0 always contains 0, so adding it to the offset value specified in the instruction just gives the offset value. In this case, we usually interpret the offset value as an unsigned 8-bit address. Our assembler tool allows us to imply the specification "(r0)" by omission and just write the address value, for example,

```
inp r3, 156
```

which reads from the I/O controller register at address 156 into r3. Similarly, if a register contains the address we want to access, we can use an offset of 0. Again, our assembler allows us to imply a 0 offset by omission, as in the instruction.

```
out r3, (r7)
```

EXAMPLE 7.4 Write instructions that increment a 16-bit unsigned integer stored in memory. The address of the least significant byte is in r2. The most significant byte is in the next memory location.

SOLUTION Since the Gumnut arithmetic instructions only operate on 8-bit data, we need to do two adds, with the carry from the first used in the second. The instructions are

```
ldm r1, (r2)
add r1, r1, 1
stm r1, (r2)
ldm r1, (r2)+1
addc r1, r1, 0
stm r1, (r2)+1
```

Since the load and store instructions do not affect the C bit, the C result from the first addition is preserved and used in the `addc` instruction.

Branch Instructions

The branch instructions allow us to conditionally change the normal flow of execution. We mentioned earlier that the CPU follows a fetch-decode-execute loop to execute instructions at successive addresses in the instruction memory. It uses a program counter (PC) register to keep track of the next instruction address, and increments this register after fetching each instruction. The branch instructions modify the sequential flow of execution by changing the PC value. Each form of branch tests a condition, and if the condition is true, adds a signed 8-bit displacement value to the PC. The displacement, specified in the instruction, indicates how many locations forward or backward the next instruction to execute is from the current instruction. (A displacement of 0 refers to the instruction after the branch, since the PC has already been incremented after fetching the branch instruction.) If the condition is false, the PC is unchanged, and execution continues sequentially. The different branch instructions allow us to test each of the Z and C condition code bits for being set to 1 or not set to 1. Since these bits are affected by arithmetic, logical and shift instructions, we often deliberately precede a branch instruction with one of these instructions to compare data values. In other cases, the condition code setting occurs as a serendipitous side effect of data operations that we need to perform anyway.

EXAMPLE 7.5 Suppose the value in data memory location 100 represents the number of seconds elapsed in a time interval. Write instructions to increment the value, wrapping around to 0 when the value increments above 59.

SOLUTION One possible sequence of instructions is

```
ldm r1, 100
add r1, r1, 1
sub r0, r1, 60
bnz +1
add r1, r0, 0
stm r1, 100
```

The first two instructions load the value into r1 and increment it. The `sub` instruction subtracts 60 from the new value and discards the result (by using r0 as the destination register). However, the Z condition code is updated as a side effect. If the new value is 60, the subtraction result is 0, so Z is set to 1; otherwise, it is cleared to 0. The branch instruction skips forward one instruction if Z is 0. The intervening `add` instruction, which is only executed when the incremented value was 60, overwrites the incremented value with 0. The final instruction, executed in all cases, stores the final value back to memory.

Jump and Miscellaneous Instructions

The first of the jump instructions, jmp, unconditionally breaks the sequential flow of execution by setting the PC to the address specified in the instruction.

EXAMPLE 7.6 Write instructions that test whether r1 is 0, and if so, clear the contents of memory location 100. If r1 is other than 0, the instructions should clear the contents of memory location 200 instead. Assume that the instructions start at address 10 in the instruction memory.

SOLUTION In the required sequence of instructions we have two alternative actions to perform, depending on whether r1 is 0. Since instructions are laid out in linear order in the instruction memory, we need to put the instructions for the two alternatives one after the other. We need an unconditional jump at the end of the first alternative to bypass the instructions for the second alternative. The instructions are

```
10: sub r0, r1, 0
11: bnz +2
12: stm r0, 100
13: jmp 15
14: stm r0, 200
15: ...
```

The second of the jump instructions, jsb, is somewhat more involved than the simple jump instruction. It allows us to execute a *subroutine*, that is, a collection of instructions that perform some desired operations and that we can invoke from different parts of the program. Starting execution of a subroutine is referred to as *calling* the subroutine. The jsb instruction is used in tandem with the ret instruction, which returns from the subroutine to the place of the call. The sequence of instruction execution for a subroutine is shown in Figure 7.4. Execution proceeds sequentially until the jsb is encountered. The jsb saves

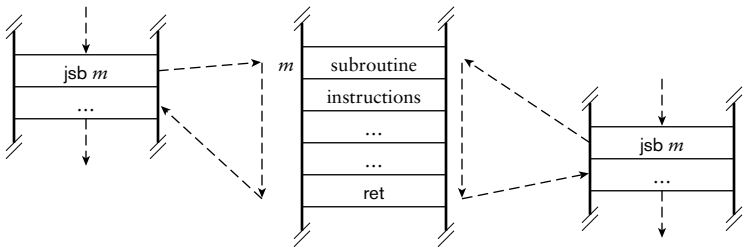


FIGURE 7.4 Flow of execution of subroutine calls. The subroutine is called from different places in the program, and in each case, returns to the instruction following the jsb.

the incremented PC value (the return address) in an internal register and then updates the PC with the subroutine address specified in the instruction. This causes instructions in the subroutine to be executed. Eventually, the subroutine executes a ret instruction, which restores the saved return address to the PC. Thus, execution continues with the instruction after the jsb. The program can include several jsb instructions that all refer to the same subroutine. In each case, the return address saved is the address of the instruction after the jsb. This allows execution to return to the right place, regardless of where the subroutine was called from.

The instructions in the subroutine can include any in the CPU's instruction set. This raises the possibility that the subroutine might include a jsb to call a sub-subroutine. The sub-subroutine might include a further jsb to call a sub-sub-subroutine, and so on. When the sub-sub-subroutine returns, execution should continue just after the jsb in the sub-subroutine, and when it returns, execution should continue just after the jsb in the subroutine. In order to achieve this effect, the CPU needs more than just a single register to save return addresses. In fact, it needs a *push-down stack* of registers, as shown in Figure 7.5. Each time a jsb is executed, the return address for that jsb is pushed onto the stack. When a ret is executed, the return address used is the top entry on the stack, and that entry is popped from the stack. The Gumnut has a return-address stack that can hold up to eight entries, which is ample for most programs.

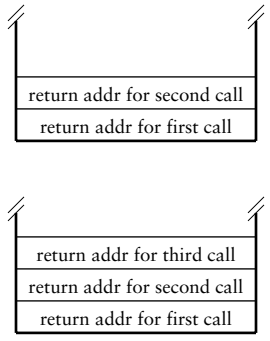


FIGURE 7.5 The push-down return-address stack after two nested calls (top) and a third nested call (bottom).

EXAMPLE 7.7 Suppose an application keeps track of a number of time intervals concurrently. Revise the sequence of instructions from Example 7.5 to form a subroutine that increments the number of seconds stored in the memory location whose address is in r2. Show how to call the subroutine to increment values in locations 100 and 102.

SOLUTION We can rewrite the instructions to form a subroutine as follows:

```
ldm r1, (r2)
add r1, r1, 1
sub r0, r1, 60
bnz +1
add r1, r0, 0
stm r1, (r2)
ret
```

Assuming the first instruction in the subroutine is at location 20 in the instruction memory, the calling instructions are

```
add r2, r0, 100
jsb 20
add r2, r0, 102
jsb 20
```

The remaining miscellaneous instructions deal with *interrupts*, which are a way of responding to events signaled by I/O controllers. The enable-interrupt instruction allows the CPU to respond to interrupt events, and the disable-interrupt instruction prevents the CPU from responding. When the CPU responds to an interrupt event, it saves the address of the instruction it is about to execute and, instead, starts executing instructions in a special subroutine called an *interrupt handler*. The interrupt handler finishes with a return-from-interrupt (reti) instruction rather than a ret instruction. The wait instruction suspends execution until an interrupt occurs, and the stby instruction enters a low-power standby mode until an interrupt occurs. The difference is that the CPU would normally be able to respond to an interrupt immediately when suspended using a wait instruction, whereas it could take some time to power up from a stby instruction. We will describe interrupt processing in more detail as part of our discussion of input/output in Chapter 8.

7.2.2 THE GUMNUT ASSEMBLER

As we mentioned earlier, programs can be written in assembly language and translated into a sequence of binary-coded instructions by an assembler. The supplementary material for this book includes a simple assembler for the Gumnut, called *gasm*. The *gas User Guide*, also included in the supplementary material, provides a detailed description of the assembly language and how to use the assembler. We will describe a few key points here, illustrated by the program in Figure 7.6.

```
; Program to determine greater of value_1 and value_2

                text
                org   0x000          ; start here on reset
                jmp   main

; Data memory layout

                data
value_1:        byte  10
value_2:        byte  20
result:         bss   1
```

(*continued*)

FIGURE 7.6 A Gumnut assembly language program to find the greater of two values.

```
; Main program
                text
                org   0x010
main:           ldm   r1, value_1    ; load values
                ldm   r2, value_2
                sub   r0, r1, r2     ; compare values
                bc    value_2_greater
                stm   r1, result     ; value_1 is greater
                jmp   finish
value_2_greater: stm  r2, result     ; value_2 is greater
finish:         jmp   finish         ; idle loop
```

FIGURE 7.6 (*continued*) A Gumnut assembly language program to find the greater of two values.

We have seen in Verilog models that we can include comments, starting with the characters "//", to describe parts of the model. We can also include comments in assembly language programs. In Figure 7.6, comments start with the ";" character and extend to the end of the line. Comments are especially important in assembly language programs, since each instruction performs only a single simple step. We use comments to describe the larger intent of a sequence of instructions.

The assembler lets us specify both the instructions to be included in the instruction memory and the contents of the data memory. We tell the assembler which memory we are specifying using the text (for instruction memory) and data (for data memory) *directives*. A directive does not represent a CPU instruction. Rather, it tells the assembler what to do when translating the program. Rather than requiring us to specify the address for each instruction and data item, the assembler adds instructions and data items at increasing addresses in each memory, starting at address 0. It automatically keeps track of where it is up to by using a *location counter* for each of the instruction and data memories. We can direct the assembler to change the location counter for the memory currently being filled by using an org (short for "origin") directive. For example, in Figure 7.6, the org 0x010 directive in the second text segment tells the assembler to continue placing instructions from location 010_{16}.

Within a data segment, we can include directives that specify the initial contents of data memory locations. The byte directive specifies the contents of an 8-bit location. The bss (short for "block starting with symbol") directive reserves a specified number of bytes of memory storage without initializing their content. We can precede each of these directives with a *label* that represents the starting address of the locations. The assembler works out the address for us. We can then refer to the label in instructions in the program. For example, the ldm instructions in Figure 7.6 refer to the labels value_1 and value_2 to load the initialized content of the data memory locations, and the stm instruction refers to the label result to store the greater value in the reserved location.

The advantage of using labels is that, when we revise the program, we don't need to revise the address values, since the assembler will work out new values when the program is reassembled.

Within a text segment, we include the instructions that form the program. Each instruction can be labeled, and the labels can be referenced in branch and jump instructions. Again, the assembler works out the instruction addresses represented by the labels, so that we don't have to work out branch displacements manually, or update references when we change the program.

One final point to note about the program in Figure 7.6 is that, once it completes its task, it doesn't stop executing. The Gumnut does not include any instructions for stopping. Instead, we include a *busy loop* at the end of the program. This just consists of an instruction that jumps back to itself, performing no useful work. Busy loops are common in embedded systems, since we usually do not want an embedded computer to stop (unless we turn the power off). An alternative is to have a CPU instruction or other facility that *suspends* operation until some activity is needed, such as responding to an I/O event. (On the Gumnut, we could use a `wait` or `stby` instruction.) This has the advantage that power consumption in the suspended state is typically much lower than in the active state. For this reason, suspending is preferred in battery-powered and other power-sensitive applications.

7.2.3 INSTRUCTION ENCODING

The instructions of a program are a form of information, and so, like any other information, can be encoded in binary. If we were to list all of the possible instructions, taking into account the operation to be performed and any registers, addresses, immediate values, and so on, we could devise an instruction coding taking up the smallest number of bits. However, decoding instructions would then be complex, leading to a large and slow decoder circuit within the CPU. Instead, instruction sets are usually encoded by separating a code word into distinct *fields*, each of which encodes one aspect of an instruction. The primary field is the *opcode*, short for operation code, that specifies the operation to be performed and, by implication, the layout of the remaining fields within the code word. By keeping the field layout simple and regular, we make the circuit for the instruction decoder simple and, hence, fast.

As an illustration, the instruction encoding for the Gumnut is shown in Figure 7.7. (The full details of the instruction encoding are described in Appendix D.) Each instruction code word is 18 bits long. The left-most bits, together with the function code (*fn*), form the opcode. Those instructions that specify register numbers have the numbers encoded in 3-bit binary form in separate fields of the instruction word. Similarly, instructions that specify immediate values, offsets, or displacements have those

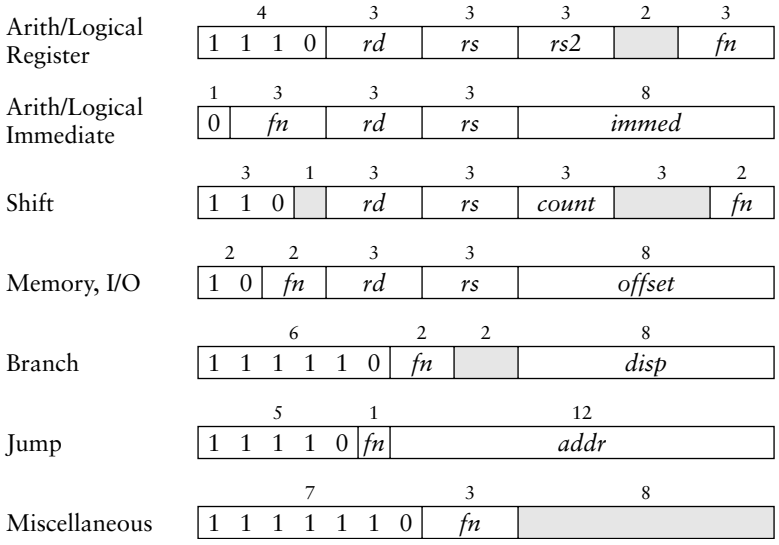


FIGURE 7.7 Instruction encoding for the Gumnut, showing the layout and size of fields within instructions.

values binary encoded in the right-most 8 bits of the instruction word. In several of the instruction formats, some bits remain unused. While this may waste some storage space within the instruction memory, the simplicity of encoding and the consequent simplicity of decoding is a trade-off worth making. As we mentioned earlier, it is the task of the assembler to translate instructions specified in textual assembly language into this binary encoding. Conversely, if we are testing a design that includes an embedded Gumnut, we may need to disassemble binary-coded instructions, that is, to determine the instructions corresponding to binary instruction code words processed by the embedded core.

EXAMPLE 7.8 Given that the function code for the addc operation is 001, what is the binary instruction word for the instruction

```
addc r3, r5, 24
```

SOLUTION This is an arithmetic/logical immediate instruction, so the left-most bit is 0, and the function code is 001. The destination register r3 is encoded as 011, the source register number as 101, and the immediate value as 00011000. So the complete instruction word is 0 001 011 101 00011000, or, in hexadecimal, 05D18.

EXAMPLE 7.9 What instruction is represented by the hexadecimal instruction word 2ECFC?

SOLUTION The binary instruction word is 111110110011111100. The left-most bits, 111110, indicate that this is a branch instruction. The function code 11 specifies a `bnc` instruction. The next two bits are 0, but are ignored in any case. The right-most 8 bits are the signed 2s-complement displacement −4. So the instruction is `bnc -4`.

7.2.4 OTHER CPU INSTRUCTION SETS

The Gumnut instruction set is relatively simple, compared to those of other CPUs. Nonetheless, it contains all the essential elements, and is quite sufficient for writing realistic embedded programs. It is similar to the instruction set of the PicoBlaze 8-bit soft core provided by Xilinx. One thing that distinguishes both of these CPUs from other commonly used 8-bit cores and microcontrollers is that all instructions are encoded in the same length. Moreover, the instruction length is not a multiple of 8 bits. (In both cases, it is 18 bits, which is one of widths to which a memory block in a Xilinx FPGA can be configured.) An example of an 8-bit microcontroller that takes a different approach is the 8051 from Intel and other vendors. It originated as a stand-alone microprocessor, and was subsequently released in microcontroller versions with various amounts of memory and I/O controllers included on chip. Its instruction set inherits from those of previous general purpose CPUs, in which a single memory address space was shared between instructions and data. Since locations in the 8051 memory are 8 bits wide, instructions are a multiple of 8-bit bytes. The opcode is included in the first byte. For some instructions the next one or two bytes contain further information to specify the instruction, such as an address and immediate data.

Another distinguishing characteristic of the 8051, compared to the Gumnut and PicoBlaze, is that the instruction set contains a much larger repertoire of operations. We call CPUs with instruction sets like this *complex instruction set computers* (CISCs), in contrast to the Gumnut and similar CPUs, which are *reduced instruction set computers* (RISCs). Many of the operations that can be expressed as one instruction on an 8051 would have to be implemented using a sequence of two or three instructions on a Gumnut. However, the complexity of the instruction set makes it much more difficult for the CPU to fetch and decode instructions. It also makes it difficult to implement a number of important CPU internal design techniques for increasing performance. For this and other reasons, RISC CPUs tend to dominate now.

The CPUs that we have mentioned thus far in this section are classified as 8-bit CPUs, as they operate only on 8-bit data. If the information to be represented in an embedded system is predominantly 16-bit, 32-bit or

64-bit data, using an 8-bit processor is very cumbersome. We may not be able to meet performance constraints, due to the number of instructions needed to implement 16-bit, 32-bit or 64-bit operations using 8-bit instructions. The alternative is to use a larger CPU whose instructions can operate on the larger data sizes directly. Most of the widely used processor cores for FPGAs and ASICs are 32-bit or 64-bit RISC CPUs. They have 32-bit or 64-bit registers and perform arithmetic and logical operations on data in those registers. They can load and store 8-bit, 16-bit, 32-bit and 64-bit data between registers and data memory. Instructions are encoded in fixed-length instruction words, usually 16 or 32 bits long. The larger, higher performance CPUs include instructions to operate on floating-point data as well as integers. Examples of this type of CPU include the PowerPC, ARM, MIPS and Tensilica cores that we mentioned earlier.

KNOWLEDGE TEST QUIZ

1. What is meant by the *instruction set* of a CPU?
2. What three steps are repeatedly performed by a CPU to execute a program?
3. How does the CPU keep track of which instruction to execute next?
4. What is meant by the terms *little endian* and *big endian*?
5. What does an assembler do?
6. What does each of the following Gumnut instructions do?

```
addc r2, r3, 25
shr  r1, r1, 3
ldm  r5, (r1)+4
bnz  -7
jsb  do_op
ret
```

7. What is the binary instruction word for the following Gumnut instruction?

```
bnc  +15
```

8. What Gumnut instruction is represented by the hexadecimal instruction word 05501?

7.3 INTERFACING WITH MEMORY

The way in which a CPU is connected to instruction and data memories depends on the implementation fabric used for both the CPU and the memories. In most embedded systems, the instruction memory is implemented with ROM, NOR flash memory, SRAM, or a combination of these. Including flash memory gives us the opportunity to upgrade the embedded software in the field. The data memory is usually implemented just with SRAM. Typically, the CPU and the memories each have a set of connection signals for the CPU/memory interface, and it is our job to join them together. If the two sets of signals are compatible, our job is relatively easy. Often, however, the sets of signals are designed in isolation, or according to different conventions. In such cases, we need to include glue logic to complete the interface.

One of the simplest cases of interfacing a CPU with memory is that of an embedded 8-bit core within an FPGA. The core includes interface signals that connect directly to those of the FPGA's memory blocks.

EXAMPLE 7.10 The memory interface signals of the Gumnut core are described in the following Verilog module definition:

```
module gumnut ( input         clk_i,
                input         rst_i,
                output        inst_cyc_o,
                output        inst_stb_o,
                input         inst_ack_i,
                output [11:0] inst_adr_o,
                input  [17:0] inst_dat_i,
                output        data_cyc_o,
                output        data_stb_o,
                output        data_we_o,
                input         data_ack_i,
                output [7:0]  data_adr_o,
                output [7:0]  data_dat_o,
                input  [7:0]  data_dat_i,
          ... );
endmodule
```

Show how to include an instance of the Gumnut core in a Verilog model of an embedded system with a 2K × 18-bit instruction memory and a 256 × 8-bit data memory.

SOLUTION The ports in the module can interface with the control signals of a flow-through SSRAM and a ROM implemented using FPGA SSRAM blocks, as described in Sections 5.2.2 and 5.2.5. In our module for our embedded system, we include the necessary nets and variables to connect to an instance

of the Gumnut entity, and use the nets and variables in always blocks for the instruction and data memories. The module is

```
module embedded_gumnut;

  reg [17:0]  inst_ROM [0:2047];
  reg [7:0]   data_RAM [0:255];

  wire         clk;
  wire         rst;
  wire         inst_cyc_o;
  wire         inst_stb_o;
  reg          inst_ack_i;
  wire [11:0]  inst_adr_o;
  reg  [17:0]  inst_dat_i;
  wire         data_cyc_o;
  wire         data_stb_o;
  wire         data_we_o;
  reg          data_ack_i;
  wire [7:0]   data_adr_o;
  wire [7:0]   data_dat_o;
  reg  [7:0]   data_dat_i;
  ...

  gumnut CPU ( .clk_i(clk_i), .rst_i(rst_i),
             .inst_cyc_o(inst_cyc_o), .inst_stb_o(inst_stb_o),
                .inst_ack_i(inst_ack_i),
             .inst_adr_o(inst_adr_o), .inst_dat_i(inst_dat_i),
             .data_cyc_o(data_cyc_o), .data_stb_o(data_stb_o),
             .data_we_o(data_we_o),   .data_ack_i(data_ack_i),
             .data_adr_o(data_adr_o), .data_dat_o(data_dat_o),
                .data_dat_i(data_dat_i), ... );

  initial $readmemh("inst_ROM.data", inst_ROM);

  always @(posedge clk)  // Instruction memory
    if (inst_cyc_o && inst_stb_o) begin
      inst_dat_i <= inst_ROM[inst_adr_o[10:0]];
      inst_ack_i <= 1'b1;
    end
    else
      inst_ack_i <= 1'b0;

  always @(posedge clk)  // Data memory
    if (data_cyc_o && data_stb_o)
      if (data_we_o) begin
        data_RAM[data_adr_o] <= data_dat_o;
        data_dat_i           <= data_dat_o;
        data_ack_i           <= 1'b1;
      end
```

(*continued*)

```
      else begin
        data_dat_i <= data_RAM[data_adr_o];
        data_ack_i <= 1'b1;
      end

  ...

endmodule
```

Note that the instruction address port of the Gumnut core is 12 bits wide, whereas the 2K × 18-bit instruction memory uses an 11-bit-wide address. In this design, we simply leave the most significant address bit of the core unconnected. Each location in the instruction memory thus appears twice in the Gumnut's instruction address space: once at an address with the most significant bit 0, and once at an address with the most significant bit 1. We would normally just use one address for the location and ignore the other alias address.

Single-chip microcontrollers, such as those based on the 8051 described in Section 7.2.4, include a small amount of instruction and data memory on the microcontroller chip. However, many of them are able to address additional off-chip memory, using a number of the chip pins for the external memory interface signals. Since using the pins for this purpose reduces the number of pins available for inputs and outputs, the memory interface pins are often multiplexed to perform different functions at different times. This complicates the connection between the microcontroller and external memory.

As an illustration, we will describe how to expand the memory of the 8051 microcontroller. The 8051 can access up to 64K bytes of instruction memory and 64K bytes of data memory, however, there are only 256 bytes of data memory and 4K to 16K bytes of instruction memory on the chip. The chip has two 8-bit input/output ports, P0 and P2, as well as a number of control signals, that can be used to connect to external memory. Figure 7.8 shows how they would be used to connect to an external 128K × 8-bit asynchronous SRAM, in which the lower 64K locations are used for instructions and the upper 64K locations for data. P2 provides the most significant address byte, and P0 is multiplexed with the least significant address byte and instruction and data bytes. Since information transfer on P0 is bidirectional, tristate drivers are used internally in the microcontroller and in the memory data pins.

The 8051 activates the address-latch enable (ALE) signal when it drives the least significant address bits on P0. We provide an 8-bit latch to hold these bits for the remainder of the memory access cycle. During an instruction read access, the 8051 activates the program-store enable ($\overline{\text{PSEN}}$) signal, driving it to a low logic level. At other times, including data

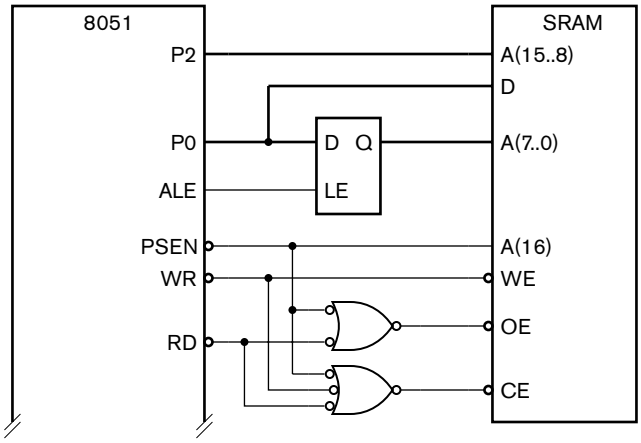


FIGURE 7.8 Connection between an 8051 microcontroller and an external combined instruction and data memory.

accesses, the signal is at a high logic level. Hence, we can use this signal directly as the most significant address bit to distinguish between instruction and data accesses to the external memory. The 8051 activates the $\overline{\text{RD}}$ signal during data read accesses and the $\overline{\text{WR}}$ signal during data write accesses. We use $\overline{\text{WR}}$ directly to control the memory's write enable ($\overline{\text{WE}}$) signal. However, we need a small amount of glue logic to derive the chip enable ($\overline{\text{CE}}$) and output enable ($\overline{\text{OE}}$) signals. We could implement this glue logic, together with the address latch, in a small PAL component.

Microcontrollers and processor cores that access 16-bit, 32-bit or 64-bit data generally need data memories that are wider than 8 bits, even though addresses correspond to 8-bit locations. This allows the CPU to access a complete data word with one read or write operation. A common approach is to make the data memory one word wide, with the byte locations arranged within the words. Figure 7.9 shows the case of byte addressing within a 32-bit-wide memory. Depending on whether the CPU is big endian or little endian, the most significant byte of a 32-bit word is stored in the byte with the lowest or highest address, respectively, of a 32-bit location. Most 32-bit CPUs ensure that 32-bit data words are stored at locations whose addresses are a multiple of four. This allows the word to be read or written with just one memory access, rather than requiring two partial memory accesses, which would be the case if the word were split over two adjacent 32-bit locations. Similarly, CPUs ensure that 16-bit halfwords are stored at locations whose addresses are a multiple of two, and that 64-bit double-words are stored at locations whose addresses are a multiple of eight, for the same reason.

Reading from data memory is quite straightforward. A 32-bit CPU, for example, reads the whole 32-bit word containing the required data item. If the required item is only a 16-bit halfword or an 8-bit byte, the CPU usually extracts the item from the appropriate memory data

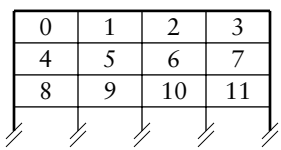


FIGURE 7.9 Arrangement of bytes within words in a 32-bit wide memory.

signals and places it in a destination register. Writing a 32-bit word is similarly straightforward. The CPU places the word on the 32 memory data signals, and the memory performs a write operation. Writing a 16-bit halfword or an 8-bit byte is more involved, since we must ensure that the other bytes in the corresponding 32-bit memory location are not affected. The CPU typically provides separate *byte write enable* control signals instead of (or in addition to) the overall write enable control signal. Alternatively, it might provide separate *byte enable* signals instead of an overall memory enable signal. To write an 8-bit byte, the CPU places the byte value on the eight memory data signals corresponding to the position of the byte within a 32-bit word and activates the associated byte enable signal. The memory then performs a write operation, updating only the enabled byte within the addressed word. Similarly, to write a 16-bit halfword, the CPU places the halfword value on the appropriate 16 memory data signals and activates the associated two byte enable signals. The memory then writes only those two bytes of the addressed word.

EXAMPLE 7.11 The Xilinx MicroBlaze 32-bit processor core has connections to a 32K × 32-bit data memory as shown in Figure 7.10. (AS stands for "address strobe." This signal is active for each memory access.) Describe how the following memory operations proceed: a word read from address 00F00; a byte read from address 00F13; a word write to address 1E010; a byte write to address 1E016; and a halfword write to address 1E020.

SOLUTION Word read from 00F00: The address is a multiple of four. Write_Strobe is 0, so all four memory components perform a read operation, providing the 32-bit data on the Data_Read signal.

Byte read from 00F13: The address is 3 more than a multiple of four, so the byte is at offset 3 within a word. However, Write_Strobe is 0, so all four memory components perform a read operation, providing the 32-bit data on the Data_Read signal. The CPU extracts the required byte from Data_Read(24:31).

Word write to 1E010: The address is a multiple of four. Write_Strobe is 1 and all four Byte_Enable signals are 1, so all four memory components perform a write operation, taking the 32-bit data from the Data_Write signal.

Byte write to 1E016: The address is 2 more than a multiple of four, so the byte is at offset 2 within a word. The CPU provides the byte data on Data_Write(16:23). Write_Strobe and Byte_Enable(2) are 1, and the remaining Byte_Enable signals are 0. The memory component connected to Data_Write(16:23) performs a write operation. The remaining components perform a read operation, but the data they supply on Data_Read(0:7), Data_Read(8:15) and Data_Read(24:31) is ignored.

Halfword write to 1E020: The address is a multiple of four, so the halfword is at offset 0 within a word. The CPU provides the halfword data on Data_Write(0:15). Write_Strobe, Byte_Enable(0) and Byte_Enable(1) are 1, and the remaining

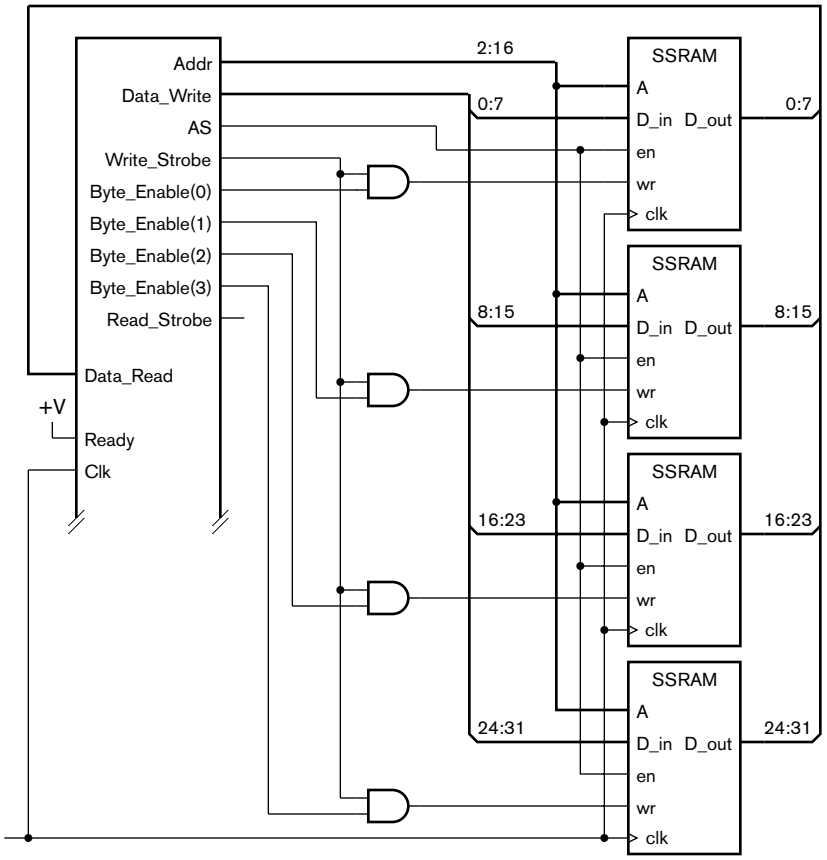


FIGURE 7.10 Connections from a Xilinx MicroBlaze core to a 32-bit data memory.

Byte_Enable signals are 0. The memory components connected to Data_Write(0:7) and Data_Write(8:15) perform a write operation. The remaining components perform a read operation, but the data they supply on Data_Read(16:23) and Data_Read(24:31) is ignored.

Some embedded systems require memory storage for large amounts of data. In such systems, it may be more appropriate to use dynamic memory (DRAMs) rather than SRAMs, given the lower cost per bit of DRAM components. As we mentioned in Section 5.2.4, controlling DRAMs is relatively complex, particularly for modern high-performance synchronous and DDR DRAMs, so we won't go into details here.

7.3.1 CACHE MEMORY

High performance embedded processors need to access instructions and data at higher rates than simple processors. For such processors,

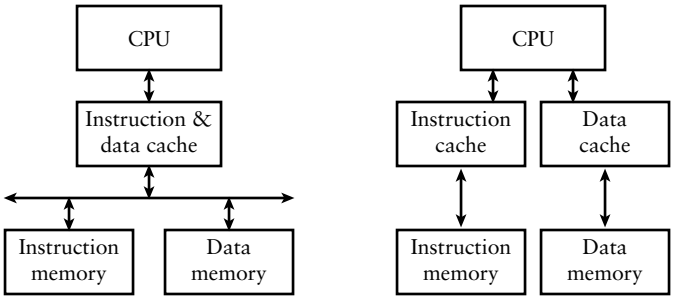


FIGURE 7.11 Processors with cache memories: a unified instruction/data cache for a single memory bus system (left), and separate instruction and data caches for a dual bus system (right).

the memory access time of a large SRAM or DRAM memory system is significantly longer than the clock cycle time of the processor, potentially making the memory a performance bottleneck. Many processors avoid the bottleneck by including a *cache* in the path between the processor and memory. A cache is a small, fast memory that stores the most frequently used items from the main memory. By making access to these items faster, we reduce the average access time experienced by the processor. Figure 7.11 shows two possible organizations: a single cache for both instructions and data, and separate caches.

Operation of a cache is predicated on the *principle of locality*, which involves two important observations about the way programs access memory. The first is that a small proportion of instructions and data account for the majority of memory accesses over a given interval of time. The second is that those items stored in locations adjacent to a recently accessed item are likely to be accessed next. To take advantage of these observations, we divide the collection of locations in main memory into fixed-sized blocks, often called *lines*, and copy whole lines at a time from main memory into the cache memory. When the processor requests access to a given memory location, the cache checks whether it already has a copy of the line containing the requested item. If so, the cache has a *hit*, and it can quickly satisfy the processor's request. If not, the cache has a *miss*, and must cause the processor to wait. The cache then copies the line containing the requested item from main memory into the cache memory. When the requested item is available in the cache, the processor can proceed with its requested access. The fact that neighboring items are also copied into the cache means that subsequent processor requests are likely to result in cache hits. As operation of the system proceeds, more and more lines are copied into the cache memory, resulting in a reduced miss rate. When the cache memory is full, some of the copied lines must be replaced by incoming lines. Ideally, the cache should replace the least recently used line. Since keeping track of usage history is complex, most caches use an approximation to determine which line to replace. In the

steady state, caches can achieve miss rates of the order of 1% of processor requests. Thus, the average access time seen by the processor is very close to the access time of the cache memory.

For a system with cache memory, most of accesses to main memory are to entire lines, rather than to single locations. Since the processor is kept waiting during a main-memory operation, it is desirable to reduce the waiting time by making cache-line accesses as fast as possible. There are a number of advanced techniques that we can use to enable a higher rate of data transfer, or *memory bandwidth*. These include:

- Wide memory: Sufficient memory chips are used so that an entire cache line can be accessed at once. The line can then be transferred back to the cache on a wide bus in one clock cycle, or over a narrower bus in several clock cycles.
- Burst transfers: The CPU issues the first address of a line to be accessed in memory. The memory then performs a sequence of accesses at successive locations, starting from the first address. This technique obviates the time required to transfer the address for locations other than the first.
- Pipelining: The memory system is organized as a pipeline so that steps of different memory operations can be overlapped. For example, the pipeline steps might be address transfer, memory access, and returning read data to the CPU. Thus, the memory system could have three memory operations in progress concurrently, with one operation completed per clock cycle.
- Double data rate (DDR) operation: Rather than transferring data items only on rising clock edges, data can be transferred on both rising and falling clock edges. This doubles the rate at which data is transferred, hence the name.

These and a number of other techniques can be used in combination to form a memory system with sufficient bandwidth to allow the processor and cache to operate with minimal waiting time. A detailed discussion is beyond the scope of this book. The topic is addressed in books on computer organization and computer architecture (see Section 7.5).

KNOWLEDGE TEST QUIZ

1. When might we need glue logic to connect a memory to a CPU?
2. In the 8051 microcontroller, why are data signals and the least significant eight address signals multiplexed onto the same set of pins?
3. How many bits wide would the data memory for a 32-bit CPU typically be?

4. Why does a 32-bit CPU provide separate byte-enable signals for its data memory?
5. What two observations about the way programs access memory define the principle of locality?
6. What is meant by the terms cache hit and cache miss?
7. During a cache miss, what happens?
8. What is meant by the term *memory bandwidth*?

7.4 CHAPTER SUMMARY

- A computer system generally contains a central processing unit (CPU), instruction and data memory, input and output (I/O) controllers, and possibly special-purpose accelerators. The elements are interconnected by one or more buses.
- A microprocessor is a single-chip CPU that can be used in a general purpose computer or an embedded computer. A microcontroller is a single-chip computer incorporating a CPU, memory and I/O controllers. A digital signal processor (DSP) is a CPU specialized for processing streams of data from digitized signals.
- Microprocessors and CPUs in microcontrollers range in scale from simple 8-bit versions to complex 32-bit and 64-bit versions, referring to the size of data that can be processed in a single operation.
- CPUs can be implemented as predesigned cores and as soft cores.
- The instruction set of a CPU is its repertoire of instructions, usually including arithmetic and logical instructions, memory and I/O instructions, branch and jump instructions, and other miscellaneous instructions.
- Little-endian CPUs store multi-byte data with the least significant byte at the lowest address and the most significant byte at the highest address. Big-endian CPUs store the bytes in the opposite order.
- Instructions are encoded in binary. However, we usually develop programs using assembly language or a high-level language and use a translator (an assembler or compiler) to translate into binary-coded instructions.
- Instruction and data memories are usually connected directly to the CPU using memory-interface signals. Memories for 8-bit, 16-bit and 32-bit CPUs are commonly 8, 16 and 32 bits wide, respectively.
- Memories for high-performance CPUs can use a number of techniques for improving the memory bandwidth, including burst transfers, pipelining and double data rate (DDR) operation.

7.5 FURTHER READING

On Holy Wars and a Plea for Peace, Danny Cohen, Internet Engineering Note 137, 1980, available at http://www.rdrop.com/~cary/html/endian_faq.html. This is the paper that originally adopted the terms "little endian" and "big endian" to refer to byte order.

Computer Architecture: A Quantitative Approach, 4th Edition, John L. Hennessy and David A. Patterson, Morgan Kaufmann Publishers, 2007. Includes a discussion of advanced memory system organization. The book also describes techniques, such as caches, used within high-performance CPUs to avoid delays due to memory accesses.

Computers as Components: Principles of Embedded Computing System Design, Wayne Wolf, Morgan Kaufmann Publishers, 2005. A more advanced reference on embedded systems design, covering CPU and DSP instruction sets, embedded systems platforms, and embedded software design.

Multiprocessor Systems-on-Chips, Ahmed Jerraya and Wayne Wolf, Morgan Kaufmann Publishers, 2004. Describes hardware, software and design methodologies for embedded systems containing multiple processor cores.

Engineering the Complex SOC: Fast, Flexible Design with Configurable Processors, Chris Rowen, Prentice Hall, 2004. Describes an approach to system-on-chip design based on extensible processors, using the Tensilica processor as an example.

ARM System-on-Chip Architecture, 2nd Edition, Steve Furber, Addison-Wesley, 2000. Describes the ARM instruction set, a number of ARM processor cores, and some examples of embedded applications using ARM cores.

Power Architecture Technology, IBM, http://www.ibm.com/developerworks/power. Resources describing the PowerPC architecture and processor cores.

See MIPS Run, 2nd Edition, Dominic Sweetman, Morgan Kaufmann Publishers, 2006. Describes the MIPS architecture, instructions set, and programming.

EXERCISES

EXERCISE 7.1 Suppose an embedded system includes two processor cores with a 32-bit wide dual-port memory for sharing data between the processors. Processor 1 is little endian, and processor 2 is big endian. Use the hexadecimal values 1234 (16 bits) and 12345678 (32 bits) to show how data is not shared correctly. How might the problem be remedied?

EXERCISE 7.2 Write Gumnut instructions to evaluate the expression $2(x + 1)$, assuming the value of x is in register r2 and the result is to be put in r7.

EXERCISE 7.3 Write Gumnut instructions to evaluate the expression $3(x - 1)$, assuming the value of x is in register r2 and the result is to be put in r7.

EXERCISE 7.4 Write Gumnut instructions to clear bits 0 and 1 of the value in register r1, leaving other bits unchanged, and to put the result in r2.

EXERCISE 7.5 Write Gumnut instructions to multiply the value in r4 by 18, ignoring the possibility of overflow. Hint: $18 = 16 + 2 = 2^4 + 2^1$.

EXERCISE 7.6 Write Gumnut instructions to increment the value in r3 modulo 60. If the result is 0, the value in r4 is to be incremented modulo 24.

EXERCISE 7.7 Write Gumnut instructions to test whether the 8-bit value in memory location 10 is equal to 99. If so, location 11 is to be set to 1; otherwise, location 11 is to be cleared to 0.

EXERCISE 7.8 Write Gumnut instructions to test whether r3 is 1 and input register 7 is also 1. If so, output register 8 is to be set to the hexadecimal value 3C.

EXERCISE 7.9 Write a Gumnut subroutine to clear a number of consecutive locations in memory to 0. The first address is provided in register r2 and the number of locations is provided in r3. Show a call to the subroutine to clear 10 locations starting from address 196.

EXERCISE 7.10 Write a complete Gumnut program to find the average of a sequence of eight 8-bit numbers stored in memory, and to write the result into a location in memory. Initialize the eight numbers to be the integers 2, 4, 6, …, 16. Use a 16-bit sum to calculate the average, and shift instructions to divide by 8.

EXERCISE 7.11 Write a complete Gumnut program that monitors the value of input controller register 10. When the value changes from 0 to a non-zero value, the program increments a 16-bit counter and writes the counter value to output controller registers 12 (least significant byte) and 13 (most significant byte). The program should not terminate.

EXERCISE 7.12 Using the information in Appendix D, determine the encoding for the following Gumnut instructions:

a) `sub r3, r1, r0`

b) `and r7, r7, 0x20`

c) `ror r1, r1, 3`

d) `ldm r4, (r3)+1`

e) `out r4, 10`

f) `bz +3`

g) `jsb 0x68`

EXERCISE 7.13 What Gumnut instructions are encoded by the following 18-bit hexadecimal values?

a) 009C0

b) 38227

c) 3353D

d) 24AFD

e) 3EA02

f) 3C580

g) 3F401

EXERCISE 7.14 Modify the design in Figure 7.8 to provide separate instruction and data memories for the 8051: a 64K × 8-bit ROM for the instruction memory and a 64K × 8-bit asynchronous SRAM for the data memory. The ROM has the same control signals as the SRAM except for the $\overline{WE}$ signal.

EXERCISE 7.15 Suppose a cache can satisfy a processor request in 5ns if it has a hit; otherwise the memory access time of 20ns must be added to the hit time. What is the average access time seen by the processor core for instructions for miss rates of 5%, 2% and 1%?

EXERCISE 7.16 Suppose a CPU with 32-bit instructions has an instruction cache with 16-byte lines. Addresses refer to bytes in memory. The cache is initially empty. Instructions are then fetched from the following addresses in order: 0, 4, 8, 92, 96, 100, 4, 8, 12, 16. For each fetch, determine whether the cache hits or misses. Assume no lines are replaced during execution of the sequence.

I/O INTERFACING

8

In the previous chapter, we introduced the notion of input/output (I/O) controllers that connect an embedded computer system with devices that sense and affect real-world physical properties. In this chapter, we will describe a range of devices that are used in embedded systems and show how they are accessed by an embedded processor and by embedded software.

8.1 I/O DEVICES

Digital systems with embedded computers are pervasive in our lives. We interact with many of them directly. Some are tools that we use in activities such as communication, entertainment, and information processing. These digital systems must incorporate human interface devices to allow us to control their operation and to receive responses. Other digital systems operate autonomously or under indirect control from us. Examples of such systems include industrial control systems, remote sensing devices and telecommunications infrastructure. These systems must incorporate devices to sense and affect the state of the physical world, as well as devices to communicate with one another, with controlling computers and with human interface devices.

Digital systems interact with the real world with *transducers*. An input transducer, or *sensor*, senses some physical property and generates an electrical signal that corresponds to the property. If the property is continuous in nature, such as temperature or pressure, the transducer may provide an analog signal that bears a continuous relationship with the physical property. Since digital systems deal with discrete representations of information, we need to convert the signal from analog to encoded digital form using a circuit called an *analog-to-digital converter*. Other forms of input transducer for continuous properties may provide discrete digital signals directly. An example is the shaft encoder for rotational position that we described in Section 3.1.3.

An output transducer, on the other hand, uses an electrical signal to cause a physical effect. Some transducers use an analog electrical signal to affect a physical property that is continuous in nature. An example is a loudspeaker that causes a continuously varying air pressure that we hear as a sound. To use such transducers in digital systems, we need a *digital-to-analog converter* circuit to convert from encoded digital form to an analog signal. Other forms of output transducer can use digital signals directly. Such transducers typically take a single-bit digital signal and cause a physical property to assume one of two values. For example, a transducer may cause a mechanical component to move to one position or another. Electromechanical transducers like this are often called *actuators*.

In the remainder of this section, we will describe a number of input and output devices that may be encountered in embedded systems. Then, in the next section, we will show how these devices can be connected to an embedded computer using input and output controllers.

8.1.1 INPUT DEVICES

Many digital systems include mechanically operated switches of various forms as input devices. These include push-button and toggle switches operated by human users, and microswitches operated by physical movement of mechanical or other objects. An example of the latter is a microswitch used to detect the presence of paper in a printer. In Section 4.4.1, we discussed ways in which switches can be connected as inputs to digital systems, and focused particularly on the problem of mechanical contact bounce and how to deal with it.

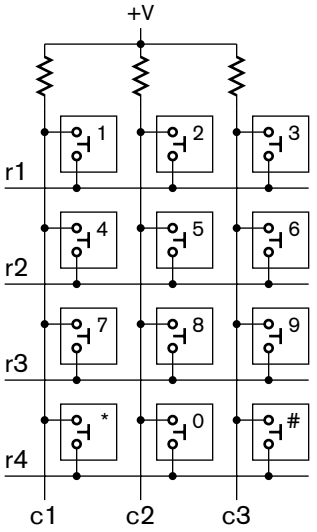


FIGURE 8.1 Keypad switches arranged in a scanned matrix.

Keypads and Keyboards

Push-button switches are also used in keypads, for example, in phones, security system consoles, automatic teller machines, and other applications. In principle, we could treat each key in a keypad as a distinct push-button switch and connect it to the digital system as we have previously described. However, that would require a large number of signals and debouncing circuits, particularly for a large keypad. A more common technique is to arrange the key switches into a matrix, as shown in Figure 8.1, and to scan the matrix for closed contacts. When all of the key switches are open, all column lines (`c1` through `c3`) are pulled high by the resistors. When a key switch is closed, one column line is connected to one row line (`r1` through `r4`). We scan the matrix by driving one row line low at a time, leaving the rest of the row lines pulled high, and seeing if any of the column lines become low. For example, if the 8 key is pressed, `c2` is pulled low when `r3` is driven low. If more than one key in a given row is pressed at the same time, all of the corresponding column lines will be

pulled low when the row line is driven low. Thus, we are able to determine the same information about which keys are pressed as we would had we used individual connections for each key switch.

This raises the question of how the row lines are driven low. We could use a counter, together with circuitry that stores the count value and the column-line values for access by the embedded software. However, that would require synchronizing the processor with changes in count value so that the software could read the values at the appropriate times. A simpler approach is to provide a register into which the processor can write values to be driven on the row lines and another register for the processor to read the values of the column lines. This is shown in Figure 8.2. (We consider how the registers are attached to the processor in Section 8.2.) Since each of the key switches is a mechanical switch, it is subject to contact bounce. Thus we need to apply techniques for debouncing similar to those that we described for individual switches. The embedded software running on the processor needs to scan the matrix repeatedly. When it detects a key closure, it must check that the same key is still closed some time (say, 10ms) later. Similarly, when it detects a key release, it must check that the same key remains released some time later. The scan must be repeated sufficiently often to debounce key presses without introducing a perceptible delay in response to key presses.

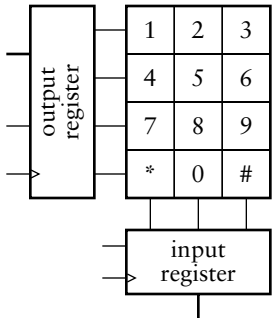


FIGURE 8.2 A keypad matrix with an output register for driving row lines and an input register for sensing column lines.

In a small digital system with a small keypad, the processing load to detect and debounce key presses would not be a significant part of the overall function of the system. The task of managing the keypad may safely be included as part of the main (or only) processor's workload. In other systems, it may be more appropriate to delegate the task, and possibly other I/O tasks, to subordinate embedded processors. The logical extension of this idea is illustrated by a keyboard for a general purpose computer. It has between 80 and 100 key switches arranged in a scanned matrix. Most keyboards include separate embedded processors whose entire workload consists of detecting key presses and dealing with roll-over (pressing a new key before the previously pressed key has been released), and communicating the information to the computer to which it is connected.

Knobs and Position Encoders

Historically, rotating knobs have been used in the user interfaces of electronic equipment to allow the user to provide information of a continuous nature. A common example is the volume control knob on audio equipment, or the brightness control on a light dimmer. In analog electronic circuits, the knob usually controls a variable resistor or potentiometer. With the introduction of digital systems, knobs were replaced by switches in many applications. For example, the volume control on audio equipment was replaced with two buttons, one to increase the volume and another

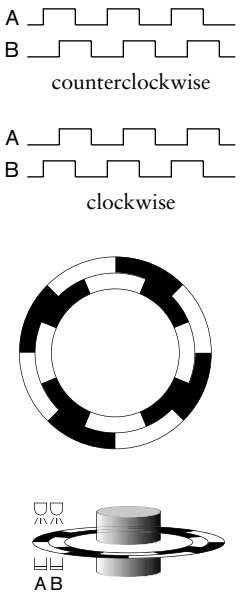


FIGURE 8.3 Operation of an incremental encoder: quadrature signals output from the encoder (top); an optical encoder disk (middle); and the disk and optical sensors attached to a shaft (bottom).

to decrease the volume. However, that form of control is not as intuitive or easy to use as a knob, so a digital form of knob is now used in many applications.

One form of digital knob input uses a shaft encoder, as we discussed in Section 3.1.3. This form has the advantage that the absolute position of the knob is provided as an input to the system. However, a simpler form of input device uses an *incremental encoder* to determine direction and speed of rotation. If the starting position or absolute position is not important, an incremental encoder is a good choice. An incremental encoder can also be used for a rotational position input in applications other than user interfaces, provided absolute positioning is not required. It can also be used for rotational speed input.

An incremental encoder operates by generating two square-wave signals that are 90° out of phase, as shown at the top of Figure 8.3. The signals can be generated either using electromechanical contacts, or using an optical encoder disk with LEDs and photo-sensitive transistors, as shown in the middle and at the bottom of Figure 8.3. As the shaft rotates counterclockwise, the A output signal leads the B output signal by 90°. For clockwise rotation, A lags B by 90°. The frequency of changes between low and high on each signal indicates the speed of rotation of the shaft.

A simple approach to using a knob attached to an incremental encoder involves detecting rising edges on one of the signals. Suppose we assume the knob is at a given position when the system starts operation. For example, we might assume a knob used as the volume control for a stereo is at the same setting as when the stereo was last used. (This would, of course, require the stereo to store the setting in a nonvolatile memory.) When we detect a rising edge on the A signal, we examine the state of the B signal. If B is low, the knob has been turned counterclockwise, so we decrement the stored value representing the knob's position. If, on the other hand, B is high, the knob has been turned clockwise, so we increment the stored value representing the knob's position. Using an incremental encoder instead of an absolute encoder in this application makes sense, since the volume might also be changed by a remote control. It is a change in the knob's position that determines the volume, not the absolute position of the knob.

Analog Inputs

Sensors for continuous physical quantities vary greatly, but they all rely on some physical effect that produces an electrical signal that depends on the physical quantity of interest. In most sensors, the signal level is small and needs to be amplified before being converted to digital form. Some sensors and the effects they rely on include:

- Microphones. These are among the most common sensors in our everyday lives, and are included in digital systems such as telephones,

voice recorders and cameras. A microphone has a diaphragm that is displaced by sound pressure waves. In an electret microphone, for example, the diaphragm forms one plate of a capacitor. The other plate is fixed and has a permanent charge embedded on it during manufacture. The movement of the plates together and apart in response to sound pressure creates a detectable voltage across the plates that varies with the sound pressure. The voltage is amplified to form the analog input signal.

- Accelerometers for measuring acceleration and deceleration. A common form of accelerometer used in automobile air bag controllers, for example, has a microscopic cantilevered beam manufactured on a silicon chip. The beam and the surface over which it is suspended form the two plates of a capacitor. As the chip accelerates (or, more important, in the air bag application, decelerates), the beam bends closer to or farther from the surface. The corresponding change in capacitance is used to derive an analog signal.
- Fluid flow sensors. There are numerous forms of sensor that rely on different effects to sense flow. One form uses temperature-dependent resistors. Two matched resistors are self heated using an electric current. One of the resistors is placed into the fluid stream which cools it by an amount dependent on the flow rate. Since the resistance depends on the temperature, the difference in resistance between the two resistors depends on the flow rate. The resistance difference is detected to derive an analog input signal. Other forms of flow-rate sensor use rotating vanes, pressure sensing in venturi restrictions, and doppler shift of ultrasonic echoes from impurities. Different forms of sensor are appropriate for different applications.
- Gas detection sensors. Again, there are numerous forms that use different effects and are appropriate for different applications. As an example, a photo-ionizing detector uses ultraviolet light to ionize a sample of atmosphere. Gas ions are attracted to plates that are held at a potential difference. A circuit path is provided for charge to flow between the plates. The current in the path depends on the concentration of the gas in the atmospheric sample. The current is sensed and amplified to form the analog input signal.

Analog-to-Digital Converters

We mentioned earlier that analog input signals from sensors need to be converted into digital form so that they can be processed by digital circuits and embedded software. The basic element of an analog-to-digital converter (ADC) is a comparator, shown in Figure 8.4, which simply senses whether an input voltage (the + terminal) is above or below a reference voltage (the − terminal) and outputs a 1 or 0 accordingly.

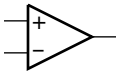


FIGURE 8.4 A symbol for a comparator.

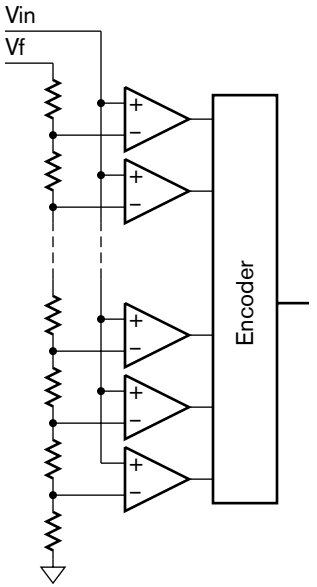


FIGURE 8.5 A flash ADC.

The simplest form of ADC is a *flash ADC*, illustrated in Figure 8.5. A converter with n output bits consists of a bank of $2^n - 1$ comparators that compare the input voltage with reference voltages derived from a voltage divider. For a given input voltage $V_{in} = kV_f$, where V_f is the full-scale voltage and k is a fraction between 0.0 and 1.0, a proportion k of the comparators have their reference voltage above V_{in} and so output 1, and the remaining comparators have their reference voltage lower than V_{in} and so output 0. The comparator outputs drive the encoder circuit that generates the fixed-point binary code for k. Flash ADCs have the advantage that they convert an input voltage to digital form very quickly. High-speed flash ADCs can perform tens or hundreds of millions of samples per second, and so are suitable for converting high bandwidth signals such as those from high-definition video cameras, radio receivers, radars, and so on. Their disadvantage is that they need large numbers of comparators. Hence, they are only practical for ADCs that encode the converted data using a relatively small number of bits. Common flash ADCs generate 8 bits of output data. We say they have a *resolution* of 8 bits, corresponding to the precision of the fixed-point format with which they represent the converted signal.

For signals that change more slowly, we can use a *successive approximation ADC*, shown in Figure 8.6. It uses a digital-to-analog converter (DAC) internally to make successively closer approximations to the input signal over several clock periods. To illustrate how the ADC works, consider a converter that produces an 8-bit output. When start input is activated, the successive approximation register (SAR) is initialized to the binary value 01111111. This value is provided to the DAC, which produces the first approximation, just less than half of the full-scale voltage. The comparator compares this approximation with the input voltage. If the input voltage is higher, the comparator output is 1, indicating that a better approximation would be above the DAC output. If the input voltage is lower, the comparator output is 0, indicating that a better

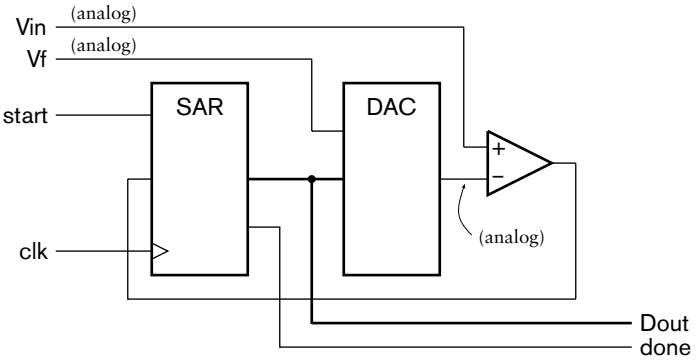


FIGURE 8.6 A successive approximation ADC. Analog signals are indicated; the remaining signals are digital.

approximation would be below the DAC output. The comparator output is stored as the most significant bit in the SAR, and remaining bits are shifted down one place. This gives the next approximation, $d_7$0111111, which is either one-quarter or three-quarters of the full-scale voltage, depending on d_7. During the next clock period, this next approximation is converted by the DAC and compared with the input voltage to yield the next most significant bit of the result and a refined approximation, $d_7d_6$011111. The process repeats over successive clock cycles, refining the approximation by one bit each cycle. When all bits of the result are determined, the SAR activates the done output, indicating that the complete result can be read.

The advantage of a successive approximation ADC over a flash ADC is that it requires significantly fewer analog components: just one comparator and a DAC. These components can be made to high precision, giving a high-precision ADC. 12-bit successive approximation ADCs, for example, are commonly available. The disadvantage, however, is that more time is required to convert a value. If the input signal changes by more than the precision of the ADC while the ADC is making successive approximation, we need to *sample and hold* the input. This requires a circuit that charges a capacitor to match the input voltage during a brief sampling interval, and then maintain the voltage on the capacitor while it is being converted. Another disadvantage of the successive approximation ADC is the amount of digital circuitry required to implement the SAR. However, that function could be implemented on an embedded processor, requiring just an output register to drive the DAC and an input bit from the comparator. The sequencing of successive approximations would then form part of the embedded software.

There are other forms of ADC apart from flash and successive approximation ADCs, each with advantages and disadvantages. Choice among them depends on the resolution, conversion speed and other factors dictated by the application. In practice, there is often a need to filter the analog input signal to ensure correct conversion to digital form. These considerations are beyond the scope of this book. More details can be found in books on digital signal processing mentioned in the Further Reading section.

8.1.2 OUTPUT DEVICES

Among the most common output devices are indicator lights that display on/off or true/false information. For example, an indicator might show whether a mode or operation is active, whether the system is busy, or whether an error condition has occurred. The simplest form of indicator is a single light-emitting diode (LED). It is low in cost, highly reliable, and easy to drive from a digital circuit, as Figure 8.7 shows. When the output from the driver is a low voltage, current flows through the LED,

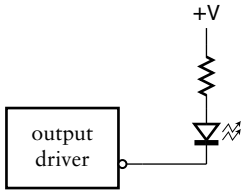


FIGURE 8.7 Output circuit for an LED indicator.

causing it to turn on. The resistor limits the current so as not to overload the output driver or the LED. We choose the resistance value to determine the current, and hence the brightness of the LED. When the output from the driver is a high voltage (near the supply voltage), the voltage drop across the LED is less than its threshold voltage, so no current flows; hence, the LED is turned off. We could, alternatively, connect the LED and resistor to ground, allowing a high output voltage to turn on the LED and a low output voltage to turn it off. However, output circuits designed to drive TTL logic levels are better able to sink current in the low state than to source current in the high state. Thus, it is more common to connect an LED as shown in Figure 8.7.

EXAMPLE 8.1 Determine the resistance for an LED pull-up resistor connected to a 3.3V power supply. The LED has a forward-biased voltage drop of 1.9V, and is sufficiently bright with a current of 2mA.

SOLUTION Assuming the output driver low voltage is close to 0V, the voltage drop across the resistor must be 3.3V − 1.9V = 1.4V. Using Ohm's Law with a current of 2mA means the resistance must be 1.4/0.002 = 700Ω. The closest standard value is 680Ω.

Displays

In Section 2.3.1, we introduced 7-segment displays and showed how we could decode a BCD value to drive the seven segments of a digit. In many applications, we have several digits to display. For example, an alarm clock typically has four digits for the hours and minutes of the time. While we could decode and drive each digit individually, that would require numerous output drivers, package pins and signals for the interconnections. Usually, it is more cost effective to connect the anodes or the cathodes of the LEDs for each digit in common, and to scan the digits. The connections for the LEDs in each digit, in this case, with common anodes, are shown in Figure 8.8. In addition to the seven LEDs for the segments, there is an LED for a decimal point (dp). The output connections for four digits are shown in Figure 8.9. Each of the outputs $\overline{A0}$ through $\overline{A3}$, when pulled low, turns on the transistor that enables a digit. We usually need these external transistors, since IC outputs cannot source enough current to drive up to eight LEDs directly.

To display four digits, we pull each of $\overline{A0}$ through $\overline{A3}$ low in turn. When $\overline{A0}$ is low, enabling the least significant digit, we drive the segment lines, $\overline{a}$ though $\overline{g}$ and $\overline{dp}$, low or high as required for the segment pattern for that digit. When $\overline{A1}$ is low, we drive the segment lines for the next digit, and so on. After driving the most significant digit, we cycle back to the least significant digit. If we cycle through the digits fast enough, our

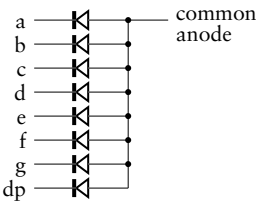


FIGURE 8.8 Connection of segment LEDs in a common anode 7-segment display.

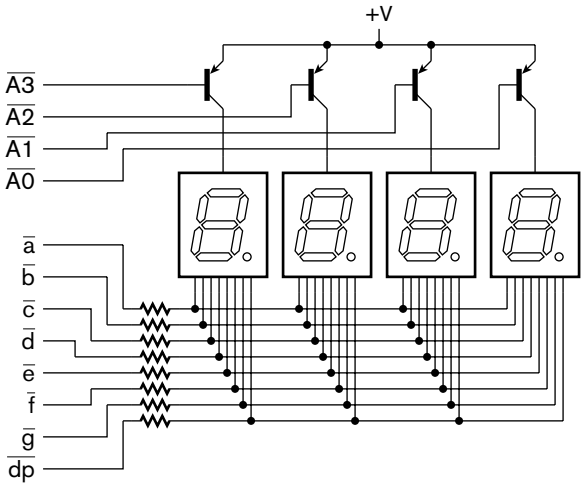


FIGURE 8.9 Connection of four 7-segment display digits.

eyes' persistence of vision smooths out any flickering due to each digit only being active 25% of the time.

The advantage of this scanned scheme is that we only need one signal for each digit plus one for each segment of a digit. For example, to drive four digits, we need 12 signals, compared with the 32 signals we would need had we driven segments individually. Depending on our application, we might use a counter or a shift register to drive the digit enable outputs and an 8-bit-wide multiplexer to select the values to drive onto the segment outputs. Often, however, the display is controlled by an embedded processor. In that case, we can simply provide output registers for the digit and segment outputs and let the embedded software manage the sequencing of output values.

EXAMPLE 8.2 Develop a Verilog model of a display multiplexer and decoder for the 4-digit 7-segment display shown in Figure 8.9. The circuit has four BCD inputs. The decimal point for the left-most digit should be lit, and the remaining decimal points not lit. The system clock has a frequency of 10MHz.

SOLUTION The module for the circuit has ports for the clock, reset and BCD inputs and for the the segment and anode outputs. Element 7 of the segment output drives the decimal point segment, and elements 6 down to 0 drive segments g through a, respectively. The outputs all use active-low logic. The circuit must include a multiplexer that selects each of the BCD inputs in turn. It decodes it to drive the 7-segment cathodes at the same time as activating the anode for the selected digit. Since we are relying on persistence of vision to avoid perceptible flicker, we need to cycle through the digits so that each is activated sufficiently frequently. A 50Hz cycle rate is acceptable. We

can achieve that rate by dividing the 10MHz clock down to 200Hz to activate a 2-bit counter for selecting digits. A module to implement these design decisions is

```
module display_mux ( output reg [3:0] anode_n,
                     output     [7:0] segment_n,
                     input      [3:0] bcd0, bcd1, bcd2, bcd3,
                     input            clk, reset );

  parameter clk_freq      = 10000000;
  parameter scan_clk_freq = 200;
  parameter clk_divisor   = clk_freq / scan_clk_freq;

  reg       scan_clk;
  reg [1:0] digit_sel;
  reg [3:0] bcd;
  reg [7:0] segment;

  integer count;

  // Divide master clock to get scan clock
  always @(posedge clk)
    if (reset) begin
      count = 0;
      scan_clk <= 1'b0;
    end
    else if (count == clk_divisor - 1) begin
      count = 0;
      scan_clk <= 1'b1;
    end
    else begin
      count = count + 1;
      scan_clk <= 1'b0;
    end

  // increment digit counter once per scan clock cycle
  always @(posedge clk)
    if      (reset)    digit_sel <= 2'b00;
    else if (scan_clk) digit_sel <= digit_sel + 1;

  // multiplexer to select a BCD digit
  always @*
    case (digit_sel)
      2'b00: bcd = bcd0;
      2'b01: bcd = bcd1;
      2'b10: bcd = bcd2;
      2'b11: bcd = bcd3;
    endcase
```

(*continued*)

```
  // activate selected digit's anode
  always @*
    case (digit_sel)
      2'b00: anode_n = 4'b1110;
      2'b01: anode_n = 4'b1101;
      2'b10: anode_n = 4'b1011;
      2'b11: anode_n = 4'b0111;
    endcase

  // 7-segment decoder for selected digit
  always @*
    case (bcd)
      4'b0000: segment[6:0] = 7'b0111111; // 0
      4'b0001: segment[6:0] = 7'b0000110; // 1
      4'b0010: segment[6:0] = 7'b1011011; // 2
      4'b0011: segment[6:0] = 7'b1001111; // 3
      4'b0100: segment[6:0] = 7'b1100110; // 4
      4'b0101: segment[6:0] = 7'b1101101; // 5
      4'b0110: segment[6:0] = 7'b1111101; // 6
      4'b0111: segment[6:0] = 7'b0000111; // 7
      4'b1000: segment[6:0] = 7'b1111111; // 8
      4'b1001: segment[6:0] = 7'b1101111; // 9
      default: segment[6:0] = 7'b1000000; // "-"
    endcase

  // decimal point is only active for digit 3
  always @* segment[7] = digit_sel == 2'b11;

  // segment outputs are negative logic
  assign segment_n = ~segment;

endmodule
```

The first always block is the clock divider that generates the 200Hz clock for selecting digits. It sets the variable scan_clk to 1 for one master clock cycle at a 200Hz rate. The second always block implements the 2-bit counter, incrementing the digit_sel variable each time scan_clk is 1. The next two always blocks use the digit_sel signal to select the BCD digit and to activate the corresponding anode. The remaining always block and assignments decode the selected digit to drive the segment cathodes.

As an alternative to using LEDs for displays, some systems use liquid crystal displays (LCDs). Each segment of an LCD consists of liquid crystal material between two optical polarizing filters. The liquid crystal also polarizes light, and, depending on the angle of polarization, can allow light to pass or be blocked by the filters. The liquid crystal is forced to twist or untwist, thus changing its axis of polarization, by application of a voltage to electrodes in front of and behind the segment. By varying the

voltage, we can make the segment appear transparent or opaque. Thus, LCDs require ambient light to be visible. In low light conditions, a back light is needed, which is one of their main disadvantages. The other disadvantages include their mechanical fragility and the smaller range of temperatures over which they can operate. They have several advantages over LEDs, including readability in bright ambient light conditions, very low power consumption, and the fact that custom display shapes can readily be manufactured.

Seven-segment displays are useful for applications that must display a small amount of numeric information. However, more complex applications often need to display alphanumeric or graphical information, and so may use LCD display panels. These can range from small panels that can display a few characters of text, to larger panels that can display text or images up to 320×240 dots, called *pixels* (short for picture elements). Beyond that size, systems would typically use the same kinds of display panels that are used in general purpose PCs. Since output for display panels is much more involved than output for simple segment-based displays, more complex control circuits are needed. We will return to control of display panels in Section 8.2.

Electromechanical Actuators and Valves

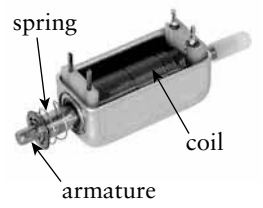


FIGURE 8.10 A solenoid actuator.

One of the simplest forms of actuator used to cause mechanical effects is a *solenoid*, shown in Figure 8.10. With no current flowing through the coil, the spring holds the steel armature out from the coil. When current flows, the coil acts as an electromagnet and draws the armature in against the spring. In a digital system, we can control the current in a small solenoid with a transistor driven by a digital output signal, as shown in Figure 8.11. The diode is required to absorb the voltage spikes that arise when the current through the inductive load is turned off.

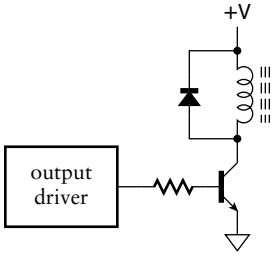


FIGURE 8.11 Solenoid controlled by a digital output.

The direct mechanical effect of activating a solenoid is a small linear movement of the armature. We can translate this into a variety of other effects by attaching rods and levers to the armature, allowing us to control the operation of mechanical systems. Hence, digitally controlled solenoids are widely used in manufacturing and other industrial applications.

There are two important classes of devices based on solenoids, the first being solenoid valves. We can attach the armature of a solenoid to a valve mechanism, allowing the solenoid to open and close the valve, thus regulating the flow of a fluid or gas. This gives us a means of controlling chemical processes and other fluid or gas based processes. Importantly, a hydraulic solenoid valve (controlling flow of hydraulic fluid) or a pneumatic solenoid valve (controlling flow of compressed air) can be used to indirectly control hydraulic or pneumatic machinery. Such machines can operate with much greater force and power than electrical machines. So solenoid valves are important components in the interface between

the disparate low-power digital electronic domain and the high-power mechanical domain.

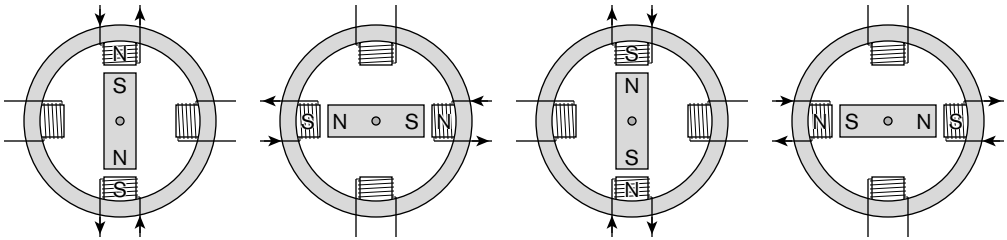
The second class of device based on solenoids is relays. In these devices, the armature is attached to a set of electrical contacts. This allows us to open or close an external circuit under digital control. The reasons for using a relay are twofold. First, the external circuit can operate with voltages and currents that exceed those of the digital domain. For example, a home automation system might use a relay to activate mains power to a mains powered appliance. Second, a relay provides electrical isolation between the controlling and the controlled circuit. This can be useful if the controlled circuit operates with a different ground potential, or is subject to significant induced noise.

Motors

Whereas solenoids allow us to control a mechanical effect with two states, many applications require mechanical movement over a range of positions and at varying speeds. For these applications, we can use electric motors of various kinds, including stepper motors and servo motors. Both can be used to drive shafts to controlled positions or speeds. The rotational position or motion can be converted to linear position or motion using gears, screws, and similar mechanical components.

A stepper motor is the simpler of the two kinds of motors that we can control with a digital system. Its operation is shown in simplified form in Figure 8.12. The motor consists of a permanent magnet rotor mounted on the shaft. Surrounding the rotor is a stator with a number of coils that can be energized to form electromagnetic poles. The figure shows that, as coils are energized in sequence, the rotor is attracted to successive angular positions, stepping around through one rotation. The magnetic attraction holds the rotor in position, provided there is not too much opposing torque from the load connected to the motor shaft. The order and rate in which the coils are energized determines the direction and speed of rotation.

FIGURE 8.12 Operation of a stepper motor.



Practical stepper motors have more poles around the stator, allowing the motor to step with finer angular resolution. They also have varying arrangements of coil connections, allowing finer control over stepping. In practical applications, current through the coils is switched in either direction using transistors controlled by digital circuit outputs. The fact that the motor is activated by the on/off switching of current makes stepper motors ideal for digital control.

A servo-motor, unlike a stepper motor, provides continuous rotation. The motor itself can be a simple DC motor, in which the applied voltage determines the motor's speed, and the polarity of the applied voltage determines the direction of rotation. The "servo" function of the motor involves the use of feedback to control the position or speed of the motor. If we are interested in controlling position, we can attach a position sensor to the motor shaft. We then use a servo controller circuit that compares the actual and desired positions, yielding a drive voltage for the motor that depends on the difference between the positions. If we are interested in controlling the speed, we can attach a tachometer (a speed sensor) to the shaft, and again use a comparator to compare actual and desired speed to yield the motor drive voltage. In both cases, we can implement the servo controller as a digital circuit or using an embedded processor. We need a digital-to-analog converter to generate the drive voltage for the motor. We can use various position or speed sensors, including the position encoders we discussed in Sections 3.1.3 and 8.1.1.

Realistic servo control involves fairly complex computations to compensate for the nonideal characteristics of the motor and any gearbox and other mechanical components, as well as dealing with the effects of the mechanical load on the system. We won't go into any detail of those effects in this book.

Digital-to-Analog Converters

Digital-to-analog converters (DACs) are the complement of analog-to-digital converters. A DAC takes a binary-encoded number and generates a voltage proportional to the number. We can use the voltage to control analog output devices, such as the servo motors we described above, loudspeakers, and so on.

One of the simplest forms of DAC is an *R-string DAC*, shown in Figure 8.13. Like the flash ADC, it contains a voltage divider formed with precision resistors. The binary-encoded digital input is used to drive a multiplexer formed from analog switches, selecting the voltage corresponding to the encoded number. The selected voltage is buffered using a unity-gain analog amplifier to drive the final output voltage. This form of DAC works well for a small number of input bits, since it is possible to match the resistances to achieve good linearity. However, for a larger number

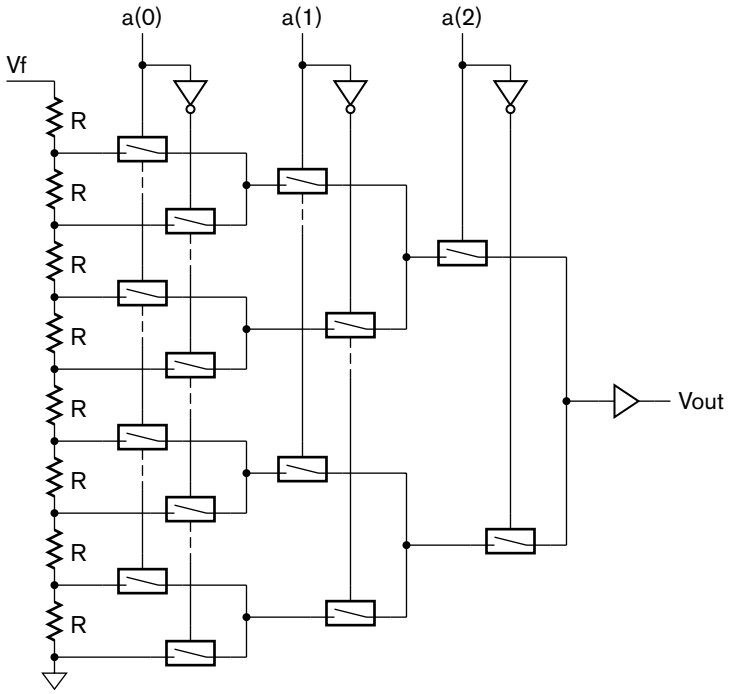


FIGURE 8.13 An R-string DAC.

of input bits, we require an exponentially larger number of resistors and switches. This scheme becomes impractical for DACs with more than eight to ten input bits.

An alternative scheme is based on summing of currents in resistor networks. One way of doing this is shown in Figure 8.14, sometimes called an *R/2R ladder DAC*. Each of the switches connected to the input bits connects the $2R$ resistance to the reference voltage Vf if the input is 1, or to ground if the input is 0. While the analysis is beyond the scope of this book, it can be shown that the currents sourced into the input node of the op-amp when the switches are in the 1 position are binary weighted. Those switches in the 0 position source no current. The superposition of the sourced currents means that the total current is proportional to the binary coded input. The op-amp voltage is thus also proportional to the binary coded input, in order to maintain the virtual ground at the op-amp input.

Just as there are numerous forms of analog-to-digital converter with various advantages and disadvantages, there are similarly numerous forms of digital-to-analog converter. We would choose an appropriate converter to meet the cost, performance and other constraints that apply to each

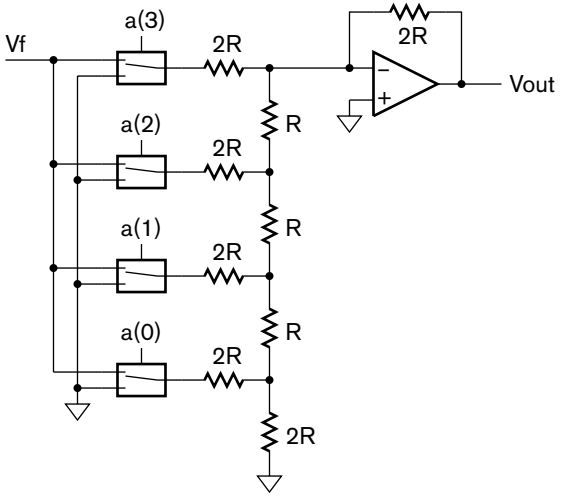


FIGURE 8.14 An R/2R ladder DAC.

application. More detail can be found in books on digital signal processing mentioned in the Further Reading section.

KNOWLEDGE TEST QUIZ

1. What is a sensor? What is an actuator?
2. Why would a digital system require a digital-to-analog converter?
3. How would we tell whether the 6 key in the keypad of Figure 8.1 is pressed?
4. Given the incremental encoder of Figure 8.3, if B is 1 when a 0 to 1 transition occurs on A, in which direction is the shaft rotated?
5. How many comparators are required in a flash ADC with a resolution of 8 bits?
6. How can we reduce the number of connections required for a multidigit 7-segment LED display?
7. What is the difference between a solenoid and a relay?
8. Identify two kinds of motor that we might control with a digital system.
9. If an application requires a 12-bit digital-to-analog converter (DAC), would we choose an R-string DAC or an R/2R ladder DAC? Why?

8.2 I/O CONTROLLERS

Given transducers, analog-to-digital converters and digital-to-analog converters, we can construct digital systems that include circuits to

process the converted input information in digital form to yield output information. However, for an embedded computer to make use of the information, we need to include components that allow the embedded software to read input information and to write output information. For dealing with input, we can provide an *input register* whose content can be loaded from the digital input data and that can be read in the same way that the processor reads a memory location. For dealing with output, we can provide an *output register* that can be written by the processor in the same way that it writes to a memory location. The output signals of the register provide the digital information to be used by the output transducer. Many embedded processors refer to input and output registers as *ports*. Since it is such a commonly used term, we will make use of it, and take care to avoid confusion with ports of Verilog modules.

In practice, both input and output registers are parts of input and output controllers that govern other aspects of dealing with transducers under software control. We will start our discussion of I/O controllers in this section with some simple controllers that just include input and output registers for transferring data. We will then move on to consider more advanced controllers.

8.2.1 SIMPLE I/O CONTROLLERS

The simplest form of controller consists just of an input register that captures the data from an input device, or just an output register to provide data to a device. Usually, there are several I/O registers, so we need to select which register to read from or write to. This is similar to selecting which memory location to access, and is solved in the same way, namely by providing each register with an address. When the embedded processor needs to access an input or output register, it provides the address of the required register. We decode the address to select the register, and only enable reading or writing of that register.

As we mentioned in Chapter 7, some processors use memory mapped I/O; that is, they just use certain memory addresses to refer to I/O registers and use the same load and store instructions for accessing both memory location and I/O registers. We can use address decoding circuits connected to the processor to identify whether memory or I/O registers are being accessed, and enable the memory chips or the appropriate register as required. Other processors, like the Gumnut that we described in Chapter 7, have separate address spaces for memory and I/O registers, and include special instructions for reading and writing I/O registers. They provide control signals that distinguish between memory and I/O register access.

EXAMPLE 8.3 The signals provided by the Gumnut core for connecting to I/O registers are described in the following Verilog module definition:

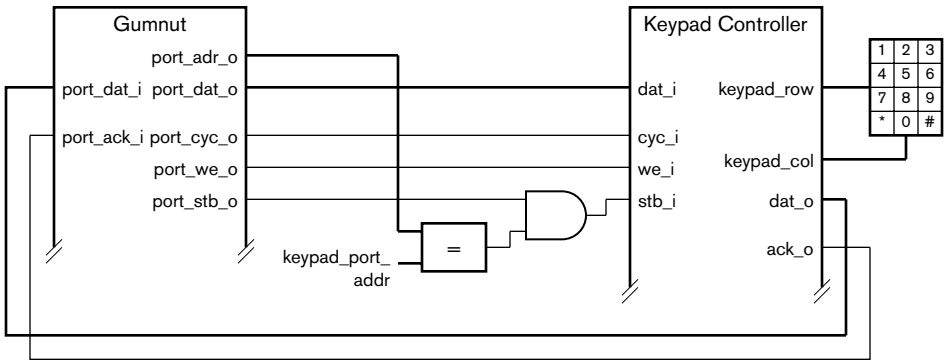
```
module gumnut ( input         clk_i,
                input         rst_i,
                ...
                output        port_cyc_o,
                output        port_stb_o,
                output        port_we_o,
                input         port_ack_i,
                output [7:0]  port_adr_o,
                output [7:0]  port_dat_o,
                input  [7:0]  port_dat_i,
                ... );
endmodule
```

The output port_adr_o is the port address, port_dat_o is the data written by an out instruction, port_dat_i is the data read by an inp instruction, port_cyc_o and port_stb_o indicate that a port read or write operation is to be performed, port_we_o indicates that the operation is a write, and port_ack_i indicates that the selected port is ready and has acknowledged completion of the read or write operation.

Develop a controller for the keypad matrix shown in Figure 8.2, and show how to connect the controller to a Gumnut core. Use output port address 4 for the matrix row output register and input port address 4 for the matrix column input register.

SOLUTION The controller connects to the Gumnut I/O signals on one side and to the keypad row and column signals on the other side, as shown in Figure 8.15. We decode the port address from the Gumnut core externally to the controller to derive the strobe control signal (stb_i) for the controller.

FIGURE 8.15 Connection of a Gumnut core to a keypad controller.



The Verilog module definition for the controller is

```
module keypad_controller ( input            clk_i,
                           input            cyc_i,
                           input            stb_i,
                           input            we_i,
                           output           ack_o,
                           input      [7:0] dat_i,
                           output reg [7:0] dat_o,
                           output reg [3:0] keypad_row,
                           input      [2:0] keypad_col );

  reg [2:0] col_synch;

  always @(posedge clk_i) // Row register
    if (cyc_i && stb_i && we_i) keypad_row <= dat_i[3:0];

  always @(posedge clk_i) begin // Column synchronizer
    dat_o     <= {5'b0, col_synch};
    col_synch <= keypad_col;
  end

  assign ack_o = cyc_i && stb_i;

endmodule
```

The first always block represents the keypad row output register, storing the value to drive on the keypad row outputs. The second always block represents the keypad column input register. Since the key switches may change at any time, we need to synchronize the input with the clock to avoid metastability failures. (We discussed this issue in Section 4.4.1.) In this design, we assume the keypad controller is the only thing driving the port_dat_o outputs, so we can assign directly to them regardless of the state of the control inputs. We will return to the topic of connecting multiple controllers in Section 8.3. The final assignment in the architecture body activates the port_ack_o output immediately on any port read or write operation, since there is no need to make the processor wait.

The controller is connected to a Gumnut core in an embedded system as shown in the following module outline:

```
module embedded_system;

  wire ...

  parameter [7:0] keypad_port_addr = 8'h04;
```

(*continued*)

```
  wire             keypad_stb_o;

  gumnut processor_core
    ( .clk_i(clk), .rst_i(rst),  ...,
      .port_cyc_o(port_cyc_o),  .port_stb_o(port_stb_o),
      .port_we_o(port_we_o),    .port_ack_i(port_ack_i),
      .port_adr_o(port_adr_o),  .port_dat_o(port_dat_o),
      .port_dat_i(port_dat_i),  ... );

  assign keypad_stb_o = port_adr_o
                        == keypad_port_addr & port_stb_o;

  keypad_controller keypad
    ( .clk_i(clk),
      .cyc_i(port_cyc_o), .stb_i(keypad_stb_o),
      .we_i(port_we_o),   .ack_o(port_ack_i),
      .dat_i(port_dat_o), .dat_o(port_dat_i),
      .keypad_row(keypad_row), .keypad_col(keypad_col) );

endmodule
```

The assignment to keypad_stb_o compares the Gumnut I/O port address with the value allocated for the keypad controller registers to derive the strobe signal for the keypad controller. The data input and output signals and the other control signals connect directly between the core and the controller.

While a simple I/O controller just has registers for input and output of data, more involved I/O controllers also have registers to allow the embedded processor to manage operation of the controller. Such registers might include *control registers*, to which a processor writes parameters governing the way transducers operate, and *status registers*, from which the processor reads the state of the controller. We often require such registers for controllers whose operation is sequential, since we need to synchronize controller operation with execution of the embedded software. As a consequence, we may have a combination of readable and writable registers used to control an input-only device or an output-only device.

EXAMPLE 8.4 In Section 8.1.1, we described a successive approximation analog-to-digital converter. It produces a binary-coded value representing the input voltage as a proportion of the full-scale reference voltage, V_f. We also mentioned that a sample-and-hold circuit can be used on the analog input if the voltage can change during the conversion process. Design a controller for a successive approximation ADC to connect to the Gumnut processor core. The controller has a control register whose contents govern operation of the converter. Bits 0 and 1 select among four alternate full-scale reference voltages.

When a 1 is written to bit 2, the analog voltage is held and a conversion is started; when a 0 is written to the bit, the analog voltage is tracked. The controller also has a status register and an input data register. Bit 0 of the status register is 1 when a conversion is complete, and 0 otherwise. Other bits of the register are read as 0. The input data register contains the converted data.

SOLUTION The controller circuit is shown in Figure 8.16. The control register is enabled when the least significant port address bit is 1 during a port write operation. The remaining port address bits are not decoded. Bits 0 and 1 of the register are decoded to control four analog switches that select the reference

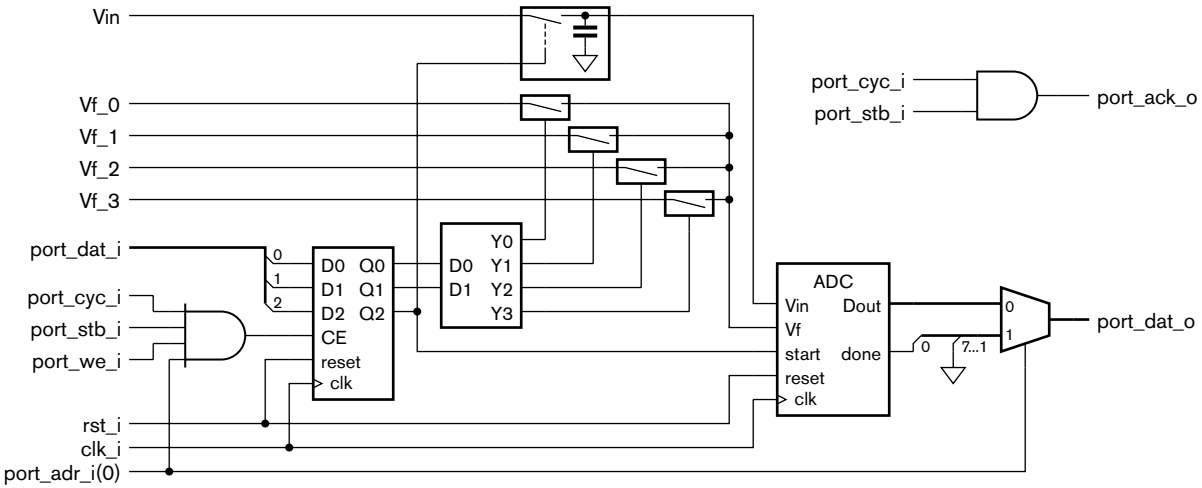


FIGURE 8.16 Circuit for a controller for a successive approximation ADC.

voltage. Bit 2 of the register controls the sample-and-hold component and the start signal of the ADC. The least significant port address bit is also used to select between the ADC data value and the ADC done status signal. Thus, when the processor performs a port read at address 0, it reads the ADC data, and when it performs a port read at address 1, it reads the done status.

8.2.2 AUTONOMOUS I/O CONTROLLERS

The simple I/O controllers in the previous section either involve no sequencing of operations, or just simple sequencing in response to accesses by a processor. More complex I/O controllers, on the other hand, operate autonomously to control the operation of an input or output device. For example, a servo-motor controller, given the desired position in an output register, might independently compute the difference between desired and actual position, compensate for mechanical lead and lag, and drive the motor accordingly. Interaction with the processor might only occur

through the processor updating the desired position in the output register and monitoring the position difference by reading an input register. In some cases, if an autonomous controller detects an event of interest to the embedded software, for example, an error condition, the controller must notify the processor. We will discuss interrupts as a means of doing this in Section 8.5.2.

One reason for providing autonomy in the controller is that it allows the processor to perform other tasks concurrently. This increases the overall performance of the system, though at the cost of the additional circuitry required for the controller. Another reason is to ensure that control operations are performed fast enough for the device. If the device needs to transfer data at high rates, or needs control operations to be performed without delay, a small embedded processor may not be able to keep up. Making the I/O controller more capable may be a better trade-off than increasing the performance or responsiveness of the processor.

As an illustration of an autonomous controller, let us return to the LCD display panels that we mentioned in Section 8.1.2 as a form of output device for complex digital systems. LCD panels consist of a rectangular array of liquid crystal pixels. The electrodes are connected in rows on one side of the panel and in columns on the other side. A voltage is applied to one row at a time, and the column electrodes are variously set to the same or a complementary voltage to activate pixels in the selected row. In this way, the panel is scanned row by row to refresh the pixel states, in much the same way that a dynamic memory must be refreshed.

Since managing and refreshing an LCD panel requires a lot of activity, manufacturers of panels typically combine a display controller with a panel to form an LCD module. The display controller is an autonomous digital subsystem that includes memory for storing the information to be displayed on the panel and circuitry for refreshing the panel. An embedded computer treats the display controller as a specialized output controller, and provides it with updates to the stored information. In a graphical LCD module, the stored information consists of the image to be displayed, represented with one bit per pixel. In a character LCD module, the stored information consists of the binary code words for the characters. The display controller is responsible for decoding the character code words and rendering the image corresponding to the characters.

A specific example of an LCD module is the ASI-D-1006A-DB-_S/W module from All Shore Industries, Inc., a 100×60 pixel LCD panel that includes an SED1560 controller chip from Seiko Epson Corp. The module is designed to connect to 8-bit microcontrollers, such as the 8051 that we mentioned in Chapter 7. Figure 8.17 shows how this might be done. The controller chip has an internal memory for storing the image to be displayed on the LCD panel. The chip provides a control register to which the microcontroller can write encoded commands, a status register,

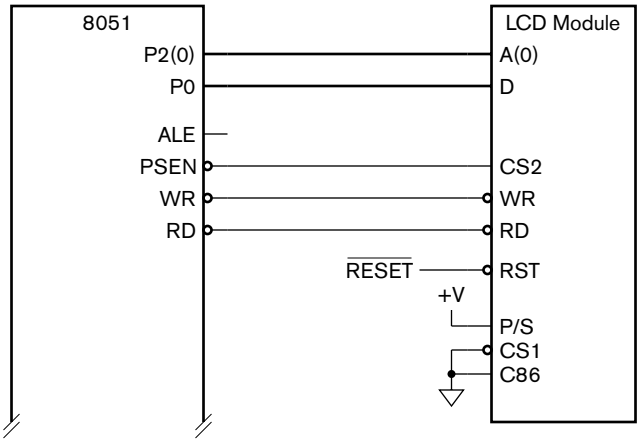


FIGURE 8.17 An LCD module connected to an 8051 microcontroller.

and a data input/output register for access to the display memory. The microcontroller issues commands to the chip to configure the display and to load pixel data into the memory. Thereafter, the chip autonomously manages scanning the display using the pixel data in its memory, leaving the microcontroller free to perform other tasks.

As we mentioned above, the use of an autonomous controller may be appropriate for a device that must transfer input or output data at high rates. Often, such data must be written to memory (in the case of input data) or read from memory (in the case of output data). If the data transfer were done by a program copying data between memory and controller registers, that activity would consume much of the processor's time. An alternative, commonly adopted in high-speed autonomous controllers, is to use *direct memory access* (DMA), in which the controller reads data from memory or writes data to memory without intervention by the processor. The processor provides the starting memory address to the controller (by writing the address to a control register), and the controller then performs the data transfer autonomously. We can think of a controller that operates in this way as an accelerator for input/output operations. Since other forms of accelerator also use DMA for data transfer, we will defer a more detailed description of DMA until Chapter 9.

KNOWLEDGE TEST QUIZ

1. What is the purpose of an input register in an I/O controller? What is the purpose of an output register?
2. What is the purpose of a control register in an I/O controller? What is the purpose of a status register?
3. If an embedded processor uses memory mapped I/O, how do we distinguish accesses to memory from accesses to I/O registers?

4. Why might a controller for an input device have registers to which a processor can write?
5. What advantages do autonomous I/O controllers have over simple controllers?

8.3 PARALLEL BUSES

As we have seen, digital circuits consist of various interconnected components. Each component performs some operation or stores data. The interconnections are used to move data between the components. Where the data is binary coded, several signals are connected in parallel, one per bit of the encoding. Many of the interconnections we have seen thus far have been simple point-to-point connections, with one component as the source of data and a single separate component as the destination. In other cases, connections fan out from a single source to multiple destinations, allowing each of the destination components to receive data from the source.

In some systems, especially embedded systems containing processor cores, parallel connections carry encoded data from multiple sources to several alternate destinations. Such connection structures, shown conceptually in Figure 8.18, are called *buses*. In the simplest case, a bus is just the collection of signals carrying the data, and control remains in a separate control section that sequences operation of the data sources and destinations. In more elaborate buses, data sources and destinations are autonomous, each with its own control section. In such cases, the control sections must communicate to synchronize the transfer of data. They do so using control signals that form part of the bus structure.

While the bus structure shown in Figure 8.18 shows the general idea of bus connection structures, it is not realizable directly as shown. Since the bus signals are shared between the data sources, only one of them should provide data at once. Most of the circuit components that we have considered so far always drive either a low or a high logic level at their outputs. If one data source drives a low level while another drives a high level, the resulting conflict would cause large currents to flow between the two components, possibly damaging them. There are several solutions to this problem, and we will look at them in turn.

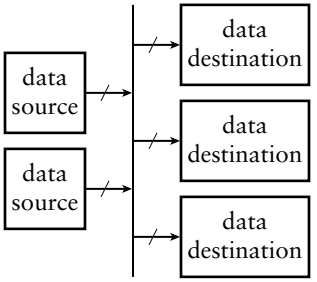


FIGURE 8.18 Conceptual connection structure for a bus.

8.3.1 MULTIPLEXED BUSES

One solution is to use a multiplexer to select among the data sources, as shown in Figure 8.19. The multiplexer selects the value to drive the bus signals based on a control signal generated by a control section. If the bus has n data sources, an n-input multiplexer is required for each bit of

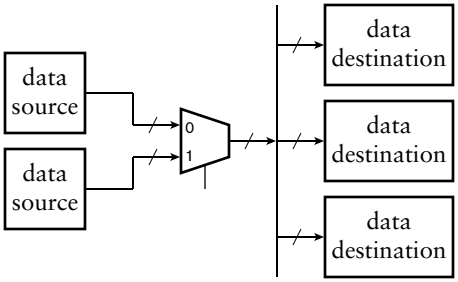


FIGURE 8.19 A bus using a multiplexer to select among data sources.

the encoded data transmitted over the bus. Depending on the number of sources and the arrangement of the components and signals on the integrated circuit chip, the multiplexer may be implemented as a single *n*-input multiplexer, or it may be subdivided into sections distributed around the chip. For example, if a bus has five data sources, two of which are on one side of a chip and the remaining three are on the other side, the bus wiring may be simplified by using a 2-input multiplexer adjacent to the two data sources and a 3-input multiplexer adjacent to the three data sources. The outputs of the multiplexers would then be connected to a 2-input multiplexer adjacent to the data destinations.

One extreme form of subdivision of bus multiplexers is the fully distributed structure shown in Figure 8.20. The data signals are connected in a chain going past all of the sources and then routed to the destinations. Each multiplexer either connects its associated data source to the chain (when the multiplexer's select input is 1) or forwards data from a preceding source (when the select input is 0). The advantage of this form of distributed multiplexer is the reduction in wiring complexity. It is often easier to route a set of signals in a chain past circuit blocks rather than trying to connect several data sources to a central hub.

One example of a bus designed to use multiplexers is the Wishbone bus. The signals in the bus and their timing are specified in a standard document, referenced in the Further Reading section. The Gumnut core

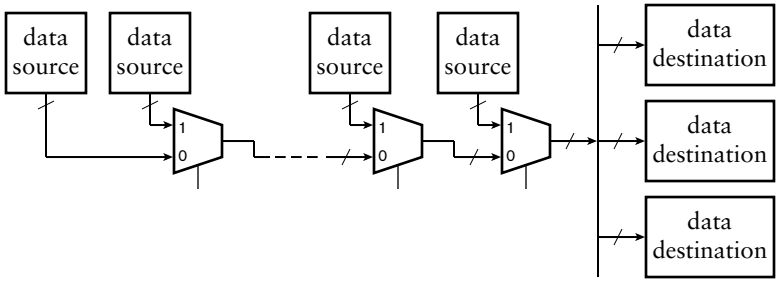


FIGURE 8.20 A distributed-multiplexer bus structure.

uses a simple form of Wishbone bus for each of the instruction, data and I/O port connections. The signals with a "_o" suffix are outputs from a component, and the signals with a "_i" suffix are inputs. Where multiple "_o" signals are to be connected to a "_i" signal, a multiplexer is required.

EXAMPLE 8.5 Show how, in an embedded system using a Gumnut core, the keypad controller of Example 8.3 and two instances of the ADC controller of Example 8.4, the components are interconnected using distributed multiplexers.

SOLUTION The Gumnut core is the single source for the port address and control signals and for the output data signals, so no multiplexer is needed for those signals. The controllers each provide input data and ack signals, so distributed multiplexers are needed for them. We can decode the port address to derive the controller strobe signals and multiplexer select signals. We choose the first ADC controller when the port address is 0 or 1, the second ADC when the port address is 2 or 3, and the keypad controller when the port address is 4. The connections are shown in Figure 8.21.

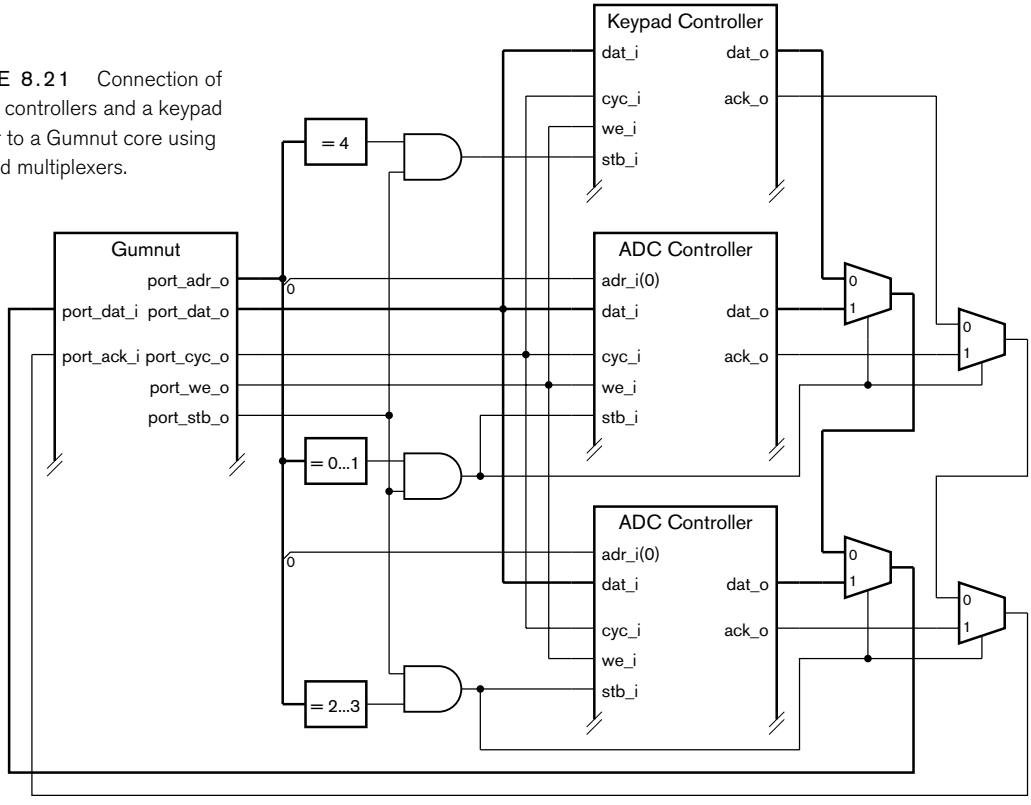


FIGURE 8.21 Connection of two ADC controllers and a keypad controller to a Gumnut core using distributed multiplexers.

EXAMPLE 8.6 Develop a Verilog model for the embedded system of Example 8.5.

SOLUTION The module definition is

```
module embedded_system_ADC_keypad;

  wire ...

  parameter [7:0] ADC0_port_addr   = 8'h00,
                  ADC1_port_addr   = 8'h02,
                  keypad_port_addr = 8'h04;

  wire       ADC0_stb_o, ADC1_stb_o, keypad_stb_o;
  wire [7:0] ADC0_dat_o, ADC1_dat_o, keypad_dat_o,
             ADC0_dat_fwd, ADC1_dat_fwd;
  wire       ADC0_ack_o, ADC1_ack_o, keypad_ack_o,
             ADC0_ack_fwd, ADC1_ack_fwd;

  gumnut processor_core
    ( .clk_i(clk),               .rst_i(rst), ...,
      .port_cyc_o(port_cyc_o),   .port_stb_o(port_stb_o),
      .port_we_o(port_we_o),     .port_ack_i(ADC1_ack_fwd),
      .port_adr_o(port_adr_o),   .port_dat_o(port_dat_o),
      .port_dat_i(ADC1_dat_fwd), ... );

  assign ADC0_stb_o = (port_adr_o & 8'hFE)
                      == ADC0_port_addr & port_stb_o;
  assign ADC1_stb_o = (port_adr_o & 8'hFE)
                      == ADC1_port_addr & port_stb_o;
  assign keypad_stb_o = port_adr_o
                        == keypad_port_addr & port_stb_o;

  keypad_controller keypad ( .clk_i(clk),
                             .cyc_i(port_cyc_o),
                             .stb_i(keypad_stb_o),
                             .we_i(port_we_o),
                             .ack_o(keypad_ack_o),
                             .dat_i(port_dat_o),
                             .dat_o(keypad_dat_o), ... );

  ADC_controller ADC0 ( .clk_i(clk),      .rst_i(rst),
                  .cyc_i(port_cyc_o),     .stb_i(ADC0_stb_o),
                  .we_i(port_we_o),       .ack_o(ADC0_ack_o),
                  .adr_i(port_adr_o[0]), .dat_i(port_dat_o),
                  .dat_o(ADC0_dat_o),     ... );

  assign ADC0_dat_fwd = ADC0_stb_o ? ADC0_dat_o : keypad_dat_o;
  assign ADC0_ack_fwd = ADC0_stb_o ? ADC0_ack_o : keypad_ack_o;
```

(*continued*)

```
  ADC_controller ADC1 ( .clk_i(clk),      .rst_i(rst),
                        .cyc_i(port_cyc_o),   .stb_i(ADC1_stb_o),
                        .we_i(port_we_o),     .ack_o(ADC1_ack_o),
                        .adr_i(port_adr_o[0]), .dat_i(port_dat_o),
                        .dat_o(ADC1_dat_o),   ... );

  assign ADC1_dat_fwd = ADC1_stb_o ? ADC1_dat_o : ADC0_dat_fwd;
  assign ADC1_ack_fwd = ADC1_stb_o ? ADC1_ack_o : ADC0_ack_fwd;

endmodule
```

The first group of assignments, after the Gumnut core instance, represent the port address decoders. They compare the port address from the processor core with the base addresses of the ADC controllers and the keypad controllers. For the ADC controllers, the port address is ANDed with the hexadecimal value FE to clear the least significant bit.

The instances of the ADC controllers are followed by assignments that represent the distributed multiplexers. The outputs of the multiplexers for the second ADC connect back to the Gumnut core port_dat_i and port_ack_i inputs.

8.3.2 TRISTATE BUSES

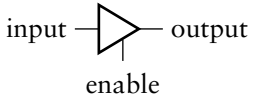


FIGURE 8.22 Symbol for a tristate driver.

A second solution to avoiding contention on a bus is to use *tristate* bus drivers. We introduced tristate drivers in Chapter 5 as part of our discussion of connecting multiple memory components. We said that the outputs of a tristate driver can be turned off by placing it in a *high-impedance*, or *hi-Z*, state. The symbol for a tristate driver is shown in Figure 8.22. When the enable input is 1, the driver behaves like an ordinary output, driving either a low or a high logic level on the output. When the enable input is 0, the driver enters the high-impedance state by turning its output-stage transistors off.

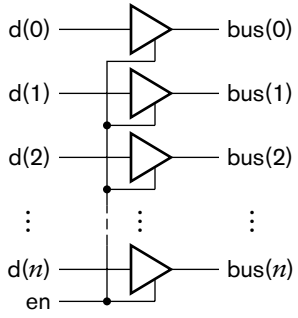


FIGURE 8.23 Parallel connection of tristate drivers.

We can implement a bus with multiple data sources by using tristate drivers on the outputs of each data source. We use one driver for each bit of encoded data provided by the source, and connect the enable inputs of the drivers for a given source together, as shown in Figure 8.23. That way, a source either drives a data value onto the bus, or has all bits in the high-impedance state. The control section selects a particular source to provide data by setting the enable input of that source's drivers to 1, and all other enable inputs to 0.

One of the main advantages of tristate buses is the reduction in wiring that they afford. For each bit of the encoded data on the bus, one signal wire is connected between all of the data sources and destinations. However, there are some issues to consider. First, since bus wires connect all of the sources and destinations, they are generally long and heavily loaded

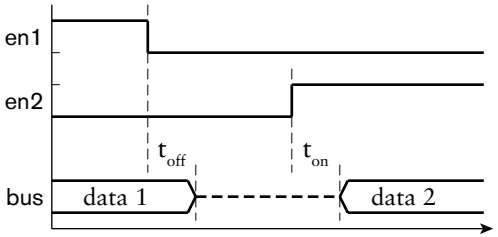


FIGURE 8.24 Tristate disable and enable timing.

with the capacitance of the drivers and inputs. As a consequence, the wire delay may be large, making high-speed data transfer difficult. Moreover, the large capacitance means we need more powerful output-stage circuits, increasing the area and power consumption of the chip.

A second issue is difficulty in designing the control that selects among data sources. The control section must ensure that one source's drivers are disabled before any other source's drivers are enabled. When we design the control section, we need to take into account the timing involved in disabling and enabling drivers. This is shown in Figure 8.24. When the enable input of a driver changes to 0, there is a delay, t_{off}, before the driver disconnects from the bus. Similarly, when the enable input changes to 1, there is a delay, t_{on}, before the driver delivers a valid low or high logic level on the bus. In the intervening time, the bus *floats*, indicated on the timing diagram by a dashed line midway between the low and high logic levels. Since there is no output driving a low or high logic level on the bus signals, each signal drifts to an unspecified voltage.

Letting the bus float to an unspecified logic level can cause switching problems in some designs. The bus signal might float to a voltage around the switching threshold of the bus destination inputs. Small amounts of noise voltage induced onto the bus wire can cause the inputs to switch state frequently, causing spurious data changes within the data destination and consuming power unnecessarily. We can avoid floating logic levels on the bus signals by attaching a *weak keeper* to the signal, as shown in Figure 8.25. The keeper consists of two inverters providing positive feedback to the bus signal. When the bus is forced to a low or high logic level by a bus driver, the positive feedback keeps it at that level, even if the forcing driver is disabled. The transistors in the output circuit of the inverter driving the bus are small, with relatively high on-state resistance, and so cannot source or sink much current. They are easily overridden by the output stages of the bus drivers.

When we need to change from one data source to another, it might seem reasonable to disable one driver at the same time as enabling the next driver. However, this can cause driver contention. If the t_{off} delay of the disabled driver is at the maximum end of its range and the t_{on} delay of the enabled driver is at the minimum end, there will be a period of overlap

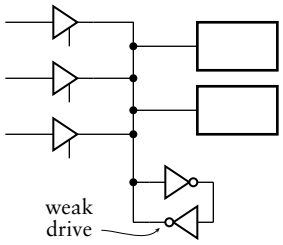


FIGURE 8.25 A bus keeper for maintaining valid logic levels.

where some bits of the enabled driver may be driving opposite logic levels to those of the disabled driver. The overlap will be short-lived and is unlikely to destroy the circuit. However, it does contribute extra power consumption and heat dissipation and ultimately will reduce the operating life of the circuit. The overlap effect can be exacerbated by clock skew in the control section. If the flip-flop that generates the enabling signal receives its clock earlier than the flip-flop that generates the disabling signal, there will be an increased chance of overlap, even if the on and off delays of the tristate drivers are near their nominal values. Given these considerations, the safest approach when designing control for tristate buses is to include a margin of dead time between different data sources driving the bus. A conservative approach is to defer enabling the next driver until the clock cycle after that in which the previous driver is disabled. A more aggressive approach is to delay the rising edges of the enable signals, for example, using the circuit of Figure 8.26, to avoid overlap between drivers. As many pairs of inverters are included as give the required delay. However, this approach requires very careful attention to timing analysis to ensure that it works effectively across the expected range of operating conditions.

A third issue relating to design of tristate buses is the support provided by CAD tools. Not all physical design tools provide the kinds of timing and static loading analyses needed to design tristate buses effectively. Similarly, tools that automatically incorporate circuit structures to enable testing of circuits after their manufacture don't always deal with tristate buses correctly. If the tools we use don't support tristate buses, we must resort to manual methods to complete and verify our design.

A final issue is that not all implementation fabrics provide tristate drivers. For example, many FPGA devices do not provide tristate drivers for internal connections, and only provide them for external connections with other chips. If we want to design a circuit that can be implemented in different fabrics with minimal change, it is best to avoid tristate buses.

In summary, tristate buses allow us to trade off significantly reduced wiring complexity against performance and design complexity, provided that our chosen implementation fabric allows tristate drivers and our CAD tool suite supports design and analysis of tristate buses. For designs that don't have stringent performance requirements, tristate buses can be a good choice. In the case of bus connections between chips on a printed circuit board, tristate buses are usually preferred. For that reason, fabrics such as FPGAs provide tristate drivers that can be used to drive output pins.

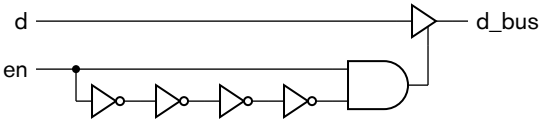


FIGURE 8.26 A circuit to delay the rising edge of a bus enable signal.

Modeling Tristate Drivers in Verilog

There are two aspects to modeling tristate drivers: representing the high-impedance state, and representing the enabling and disabling of drivers. In previous chapters, we have used single-bit Verilog net and variable values to represent single-bit logic levels. Nets and variables can also take on the value Z for representing the high-impedance state. In a Verilog model for a circuit, we can assign Z to an output to represent disabling the output. Subsequently, assigning 0 or 1 to the output represents enabling it again.

There are several additional points we should make about modeling tristate drivers in Verilog. First, we can write a Z value using either an uppercase or lowercase letter. Thus, `1'bZ` and `1'bz` are the same. Second, we can only write literal Z values as part of a binary, octal or hexadecimal number, such as `1'bZ`, `3'oZ` and `4'hZ`. In an octal number, a Z represents three high-impedance bits, and in a hexadecimal number, a Z represents four high-impedance bits. Third, Verilog allows us to use the keyword `tri` instead of `wire` for a net connected to the output of a tristate driver. Thus, we might write the following declaration in a module:

```
tri d_out;
```

or the following port declaration:

```
module m ( output tri a, ... );
```

Apart from the use of the different keyword, a `tri` net behaves exactly the same as a `wire` net. The `tri` keyword simply provides documentation of our design intent. Note that there is no corresponding keyword for a variable that is assigned a Z value; we continue to use the `reg` keyword for that purpose.

EXAMPLE 8.7 Write a Verilog statement to model a tristate driver for an output net `d_out`. The driver is controlled by a net `d_en`, and when enabled, drives the value of an input `d_in` onto the output net.

SOLUTION We can use an assignment statement, as follows:

```
assign d_out = d_en ? d_in : 1'bZ;
```

For multibit buses, we can use vectors whose elements include Z values. While we can assign 0, 1 and Z values individually to elements of vectors, we usually assign either a vector containing just 0 and 1 values to represent an enabled driver or a vector of all Z values to represent a disabled driver. Verilog's implicit resizing rules for vector values involve extending with Z elements if the leftmost bit of the value to be extended is Z. So we can write 8'bz to get an 8-element vector of Z values.

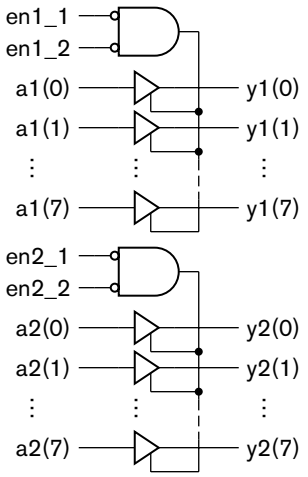


FIGURE 8.27 Internal circuit of the 16541 component.

EXAMPLE 8.8 The SN74x16541 component manufactured by Texas Instruments is a dual 8-bit bus buffer/driver in a package for use in a printed circuit board system. The internal circuit of the component is shown in Figure 8.27. Develop a Verilog model of the component.

SOLUTION We can use vector ports for each of the 8-bit inputs and outputs, and single-bit ports for the enable inputs. The module definition is:

```
module sn74x16541 ( output tri [7:0] y1, y2,
                    input      [7:0] a1, a2,
                    input            en1_1, en1_2, en2_1, en2_2 );

  assign y1 = (~en1_1 & ~en1_2) ? a1 : 8'bz;
  assign y2 = (~en2_1 & ~en2_2) ? a2 : 8'bz;

endmodule
```

Each assignment within the module represents one of the 8-bit sections of the component. The condition in the assignment determines whether the 8-bit tristate driver is enabled or disabled. The driver is disabled by assigning a vector value consisting of all Z elements. Note the use of the tri keyword in the declaration of the output ports to indicate that they can be assigned Z values.

When we have multiple data sources for a tristate bus, our Verilog model includes multiple assignment statements that assign values to the bus. Verilog must *resolve* the values contributed by the separate assignments to determine the final value for the bus. If one assignment contributes 0 or 1 to a bus and all of the others contribute Z, the 0 or 1 value overrides the others and becomes the bus value. This corresponds to the normal case of one driver being enabled and the rest disabled. If one assignment contributes 0 and another contributes 1, we have a conflict. Verilog then uses the special value X, called *unknown*, as the final bus value, since it is unknown whether a real circuit would produce a low, high or invalid logic level on the bus. Depending on how the Verilog model of a data destination receiving an X value is written, it might propagate the unknown value to its outputs, or produce arbitrary

0 or 1 values. Ideally, it would include a verification test statement that would detect unknown input values. If all assignments to a bus contribute Z, the final signal value is Z. This corresponds to the bus floating. Again, since this does not represent a valid logic level, a Verilog model of a data destination receiving a Z input should propagate an X output and detect the error condition.

An important point to realize about the Z and X values is that they do not represent real logic levels in a physical circuit. Rather, assignment of Z to an output is a notational device interpreted by synthesis CAD tools as implying a tristate driver for the output. Assignment of X to an output is a notational device used in simulation to propagate error conditions in cases where we cannot determine a valid output value. We can write Verilog statements that test whether a bus has the value Z or X, but it only makes sense to do so in testbench models, for example, in an if statement to verify that all drivers of a bus have been disabled or that there is no bus conflict. Since, according to our digital abstraction, signals in a physical circuit are only ever 0 or 1, a real digital component cannot sense any other level.

If we need to test for Z or X values in a testbench model, we should use different equality and inequality operators from those we have used so far. The == operator in Verilog, known as the *logical equality* operator, represents a hardware equivalence operation. If either operand is Z or X, the result is X, since it is unknown whether the values in a real circuit are equivalent or not. Similarly, the != operator, *logical inequality*, represents a hardware unequivalence operation, and returns X if either operand is Z or X. Thus, for example, the expressions `1'b0 == 1'bX` and `1'bZ != 1'b1` both yield X. If we want to test for Z and X values, we must use the === and !== operators, known as the *case equality* and *case inequality* operators, respectively. These perform an exact comparison, including X and Z values. Thus, `1'b0 === 1'bX` yields 0 (false), and `1'bZ !=== 1'b1` yields 1 (true). Note that, like the Z value, we can use an uppercase or lowercase letter, and we can only write literal X values in binary, octal, or hexadecimal numbers.

EXAMPLE 8.9 Suppose a Verilog module includes the following declarations and assignments

```
tri [11:0]  data_1, data_2, data_bus;
wire sel_1, sel_2;
...
assign data_bus = sel_1 ? data_1 : 12'hz;
assign data_bus = sel_2 ? data_2 : 12'hz;
```

Write a test to verify that the values of all elements of the bus signal are all valid logic levels, or that all drivers are disabled.

SOLUTION Unfortunately, Verilog does not provide an operation expressly for testing for X or Z values within a vector. However, we can make use of a property of the reduction XOR operator, ^. This operator can be applied to a vector to form the XOR of all of the bits of the vector, yielding a single-bit result. If all of the bits are 0 or 1, the result is 0 or 1, but if any bit is X or Z, the result is X. Thus, our test can be written as:

```
if ((^data_bus) === 1'bx && data_bus !== 12'hz)
  $display("Invalid value on data_bus");
```

Note that the first part of the condition includes the case of all elements being Z, so we need to check for that case separately.

8.3.3 OPEN-DRAIN BUSES

A third solution to avoid bus contention is to use *open-drain* drivers, as shown in Figure 8.28. Each driver connects the drain terminal of a transistor to the bus signal. When any of the transistors is turned on, it pulls the bus signal to a low logic level. When all of the transistors are turned off, the termination resistor pulls the bus signal up to a high logic level. If multiple drivers try to drive a low logic level, their transistors simply share the current load. If there is a conflict, with one or more drivers trying to drive a low level and others letting the bus be pulled up, the low-level drivers win. Sometimes, this kind of bus is called a *wired-AND* bus, since the bus signal is only 1 if all of the drivers output 1. If any driver outputs 0, the bus signal goes to 0. The AND function arises from the wiring together of the transistor drains. We can also use this form of bus with drivers that use bipolar transistors instead of MOSFET transistors. In that case, we connect the collector terminal of a transistor to the bus signal, as shown in Figure 8.29. Such a driver is called an *open-collector* driver.

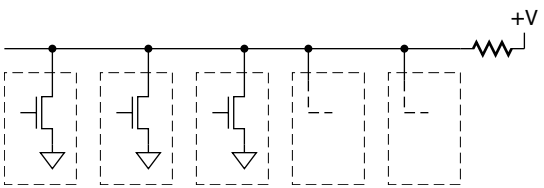


FIGURE 8.28 Open-drain bus structure.

Given the need for a pull-up resistor on each bus signal, open-drain or open-collector buses are usually found outside integrated circuits. For example, they may be used for a bus that connects a number of integrated circuits together, or for the signals in a backplane bus that connects a number of printed circuit boards together. Implementing pull-up resistors within an integrated circuit takes up significant area and consumes power. Hence, we usually use multiplexed or tristate buses within an integrated circuit chip. If we need the AND function that would be formed by open-drain connection, we can implement it with active gates.

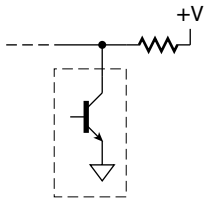


FIGURE 8.29 Open-collector bus driver.

Modeling Open-Drain and Open-Collector Connections in Verilog

We can model open-drain and open-collector drivers using a different kind of net, declared with the keyword wand (short for wired-AND). For example:

```
wand bus_sig;
```

We assign 0 to a wand net to represent a driver whose output transistor is turned on, pulling the net low. We assign 1 to the net to represent a driver whose output transistor is turned off. When a wand net is resolved, any 0 values override all other values. However, if all of the drivers are turned off, contributing 1 values, the final value of the net is 1. Note that the pull-up resistor for the bus is not explicitly represented in the model; rather, it's effect is implicit in the declaration of the net as wand instead of wire.

8.3.4 BUS PROTOCOLS

In most design projects, subsystems are often designed by different team members. Some subsystems may also be procured from external providers, or be implemented using off-the-shelf components. If the subsystems are to be interconnected using buses, it would be preferable for them to use the same bus signals with the same timing requirements; otherwise, interface glue logic is required. In order to facilitate connection of separately designed components, a number of common *bus protocols* have been specified. Some of the specifications are embodied in industry and international standards, whereas others are simply specifications agreed upon or promoted by component vendors. The specification of a bus protocol includes a list of the signals that interconnect compliant components, and a description of the sequences and timing of values on the signals to implement various bus operations.

Bus specifications and protocols vary, depending on their intended use. Some, intended for connecting separate chips on a circuit board or separate boards in a system, use tristate drivers for signals that have multiple data sources. Examples include the PCI bus used to connect add-on cards to personal computer systems, and the VXI bus used to connect measurement instruments to controlling computers. Others are intended for connecting subsystems within an IC. They have separate input and output signals, allowing for connection using multiplexers or switching circuits. Examples include the AMBA buses specified by ARM, the CoreConnect buses specified by IBM, and the Wishbone bus specified by the OpenCores Organization. Buses also vary in the number of parallel signals for transferring addresses and data, and in the speed of operation. Some, intended for high-speed data transfer, provide for the kinds of techniques we mention in Chapter 7, such as burst transfers and pipelining.

In this section, we will describe the relatively simple I/O bus protocol used by the Gumnut core. We have already introduced several aspects of the bus specification in preceding examples in this chapter. We will draw all of the aspects of the specification together here.

The Wishbone I/O bus signals for the Gumnut are described in the Verilog module definition in Example 8.3 and are shown as part of the Gumnut schematic symbol in Figure 8.21. To summarize, the signals are:

- port_cyc_o: a "cycle" control signal that indicates that a sequence of I/O port operations is in progress.
- port_stb_o: a "strobe" control signal that indicates an I/O port operation is in progress.
- port_we_o: a "write enable" control signal that indicates the operation is an I/O port write.
- port_ack_i: a status signal that indicates that the I/O port acknowledges completion of the operation.
- port_adr_o: the 8-bit I/O port address.
- port_dat_o: The 8-bit data written to the addressed I/O port by an out instruction.
- port_dat_i: the 8-bit data read from the addressed I/O port by an inp instruction.

When the Gumnut core executes an out instruction, it performs a port write operation. The timing of the operation is shown in Figure 8.30. Transitions are synchronized by the system clock. The Gumnut starts a write operation by driving the port_adr_o signals with the address computed by the out instruction and the port_dat_o signals with the data from the source register of the out instruction. It sets the port_cyc_o, port_stb_o

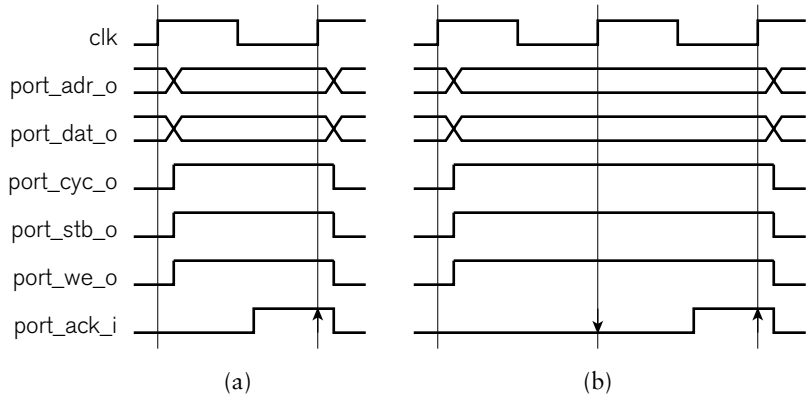


FIGURE 8.30 Timing for Gumnut I/O write operations: without wait cycles (a), and with one wait cycle (b).

and port_we_o control signals to 1 to indicate commencement of the write operation. The system in which the Gumnut is embedded decodes the port address to select an I/O controller and to enable the addressed output register to store the data. If the addressed controller is able to update the register within the first clock cycle, it sets the port_ack_i signal to 1 in that cycle, as shown in Figure 8.30(a). On the next rising clock edge, the Gumnut sees port_ack_i at 1 and completes the operation by driving port_cyc_o, port_stb_o and port_we_o back to 0. If, on the other hand, the addressed controller is slow and is not able to update the output register within the cycle, it leaves port_ack_i at 0, as shown in Figure 8.30(b). The Gumnut sees port_ack_i at 0 on the rising clock edge, and extends the operation for a further cycle. The controller can keep port_ack_i at 0 for as long as it needs to update the register. Eventually, when it is ready, it drives port_ack_i to 1 to complete the operation. This form of synchronization, involving strobe and acknowledgment signals, is often called *handshaking*.

The Gumnut performs a port read operation when it executes an inp instruction. The timing for the operation, shown in Figure 8.31, is similar

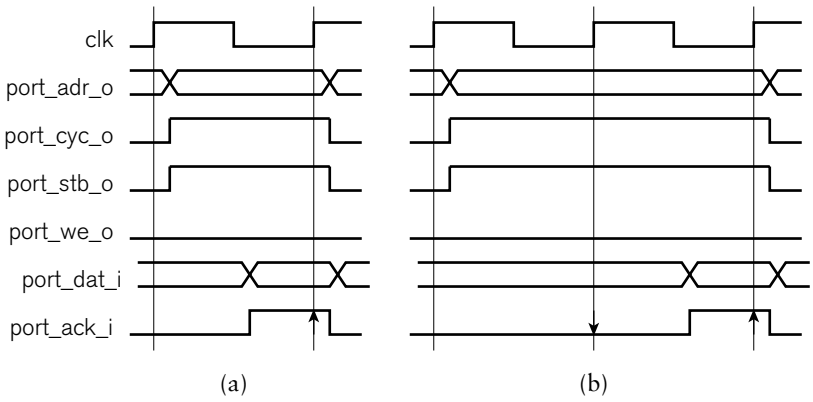


FIGURE 8.31 Timing for Gumnut I/O read operations: without wait cycles (a), and with one wait cycle (b).

to that for a port write. The Gumnut starts the port read operation by driving the port_adr_o signals with the computed address, driving the port_cyc_o and port_stb_o signals to 1, and leaving port_we_o at 0. Again, the system decodes the address to select an I/O controller and enable the addressed input register onto the port_dat_i signals. The controller drives the port_ack_i signal to 1 as soon as it has supplied the data, either during the first cycle, as in Figure 8.31(a), or in a subsequent cycle, as in Figure 8.31(b). On seeing port_ack_i at 1, the Gumnut transfers the data from the port_dat_i signals to the destination register identified in the inp instruction. It then completes the port read operation by driving port_cyc_o and port_stb_o back to 0.

At first sight, it might appear that the port_cyc_o and port_stb_o signals are duplicates of each other. However, the Wishbone bus specification defines other more involved operations in which the two control signals serve distinct purposes. While the Gumnut does not use those operations, it includes the signals in order to maintain compatibility with the Wishbone specification. The additional signal is a small cost to pay for compatibility with a large pool of third-party components.

KNOWLEDGE TEST QUIZ

1. If a system requires connection of multiple data sources and destinations, why can we not just connect them directly as shown in Figure 8.18?
2. In a multiplexed bus system, why might it be desirable to subdivide the multiplexers and distribute them around the chip?
3. How does a tristate bus avoid logic-level contention on bus signals?
4. Why should we avoid floating bus signals?
5. What is a weak keeper?
6. What problems can arise if we disable one tristate bus driver at the same time as enabling the next driver? How can we avoid the problems?
7. Write a Verilog assignment that represents a tri-state bus driver for an 8-bit bus.
8. What value results on a Verilog wire net when two tristate drivers are enabled and driving opposite logic levels?
9. Why is a signal connecting several open-drain drivers called a wired-AND connection?
10. Write a Verilog declaration that represents an open-drain bus.
11. What is a bus protocol?

8.4 SERIAL TRANSMISSION

Throughout this book, we have described transfer of binary-encoded data using *parallel transmission*, in which we dedicate one signal wire per bit of encoded data. While this might appear to give us the fastest possible rate of data transfer, there are some disadvantages. The most obvious is that we require one signal wire per bit. For wide encodings, the wiring takes up significant circuit area, and makes layout and routing of the circuit more complex. For connections that extend between chips, parallel transmission requires more pad drivers and receivers, more pins, and more PCB traces. These all add cost to the system. Moreover, there are secondary effects, such as increased delay due to the extra space required for the connections, problems with crosstalk between wires routed in parallel, and problems with skew between signals. Dealing with these problems adds cost and complexity to the system. In this section, we will describe an alternative scheme for transferring binary-encoded data. The scheme is called *serial transmission*, since bits are transmitted one bit at a time in series over a single signal wire.

8.4.1 SERIAL TRANSMISSION TECHNIQUES

In order to transform data between parallel and serial form, we can use shift registers, introduced in Section 4.1.2. At the transmitting end, we load the parallel data into a shift register and use the output bit at one end of the register to drive the signal. We shift the content of the register one place at a time to drive successive bits of data onto the signal. At the receiving end, as each bit value arrives on the signal, we shift it into a shift register. When all the bits have arrived, the complete data code word is available in parallel form in the shift register. We sometimes use the term *serializer/deserializer*, or *serdes*, for shift registers used in this way. The advantage of serial transmission is that we only need one signal wire to transfer the data. Thus, we reduce the circuit area and cost for the connection. Moreover, if necessary, we can afford to optimize the signal path so that bits can be transferred at a very high rate. Some serial transmission standards in use today allow for rates exceeding 10 gigabits per second.

EXAMPLE 8.10 Show how a 64-bit data word can be transmitted serially between two parts of a system. Assume that the transmitter and the receiver are both within the same clock domain, and that the signal start is set to 1 on a clock cycle in which data is ready to be transmitted.

SOLUTION At the transmitting end, we need a 64-bit shift register with parallel load control and an output from the least significant bit. At the receiving end, we also need a 64-bit shift register, but with a single-bit input and

parallel data output. The connections are shown in Figure 8.32. The figure also shows the control section that sequences the serial transmission. When a start pulse occurs, the control section activates the receiver clock enable, rx_ce, for 64 cycles to shift the serial data in. The control section then pulses rx_rdy to indicate that the received data is ready. A timing diagram for one transmission is shown in Figure 8.33. We can implement the control logic with a counter and a simple finite-state machine.

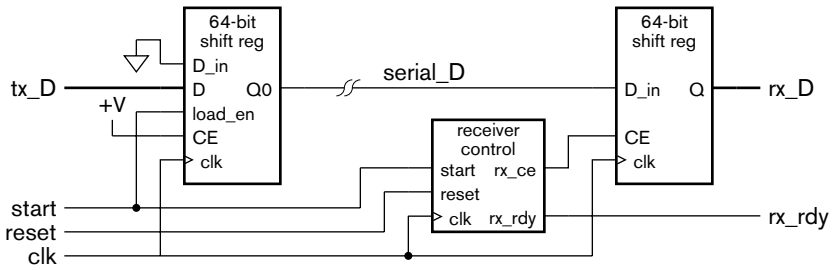


FIGURE 8.32 Serial transmission of 64-bit data within a clock domain.

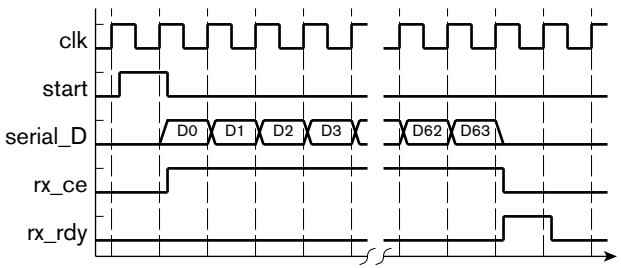


FIGURE 8.33 Timing diagram for the serial receiver control.

One important issue that we need to address when transferring data serially is the order in which we transmit the bits. In principle, the order is arbitrary, so long as the transmitter and receiver agree. Otherwise, the receiver will end up with the bits in reverse order. In Example 8.10, we transmit the least significant bit first, and so shift bits into the receiver shift register at the most significant end, shifting them down to the least significant end. Fortunately, serial transmission in a system is often governed by a standard that specifies the order. This absolves us of the need to decide.

Another important issue is synchronization of the transmitter and the receiver. If we just drive the signal with the data bit values, there is no indication of when the time for one bit ends and the time for the next bit starts. This form of serial transmission is called *non-return to zero* (NRZ), and is illustrated in Figure 8.34, which shows the logic levels on a signal for NRZ serial transmission of the value 11001111, with the most significant bit being transmitted first. We assume in this case that the value on the signal when no bit is being transmitted is 0. In the figure,

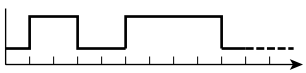


FIGURE 8.34 Serial transmission of the value 11001111.

we have drawn a timescale showing the interval in which each bit occurs. However, that information is implicit, rather than being explicitly transmitted to the receiver along with the data. If the receiver, for some reason, assumed intervals twice as long for each bit, it would receive the value 10110000. To avoid this problem, we need to synchronize the transmitter and receiver, so that the receiver samples each bit value on the signal at some time during the interval when the transmitter drives the signal with the bit value.

There are three basic ways in which we can synchronize the transmitter and receiver. The first is by transmitting a clock on a separate signal wire. We saw this scheme in Example 8.10. The second is by signaling the start of a serial code word and relying on the receiver to keep track of the individual bit intervals. A common way of doing this originated with teletypes, which were computer terminals consisting of a keyboard and a printer connected to a remote computer using serial transmission. A refined version of such serial transmission is still used to connect some devices to serial communications ports on modern PCs.

In this second scheme, the signal is held at a high logic level when there is no data to transmit. When data is ready to be transmitted, transmission proceeds as shown in Figure 8.35, again with the most significant bit transmitted first. The signal is brought to a low logic level for one bit time to indicate the start of transmission. We call this the start bit. After that, the bits of data are transmitted, each for one bit time. We might also transmit a parity bit after the data bits, in case the signal wire is subject to induced noise, though this is not shown in the figure. This would allow us to detect some errors that might occur during transmission. Finally, we drive the signal high for one further bit time to indicate the end of transmission of the data. We call this the stop bit. We can then transmit the next piece of data, starting with a start bit, or leave the signal high if there is no data ready to transmit.

At the receiving end, the receiver monitors the logic level on the signal. While it remains at a high logic level, the receiver is idle. When the receiver detects a low logic level of the start bit, it prepares to receive the data. It waits until the middle of the first bit time and shifts the value on the signal into the receiving shift register. It then waits for further successive bit times, shifting each bit into the shift register. The complete data is available after the last bit is received. The receiver uses the stop-bit time to return to the idle state.

Note that the transmitter and the receiver must agree on the duration of the bit times on the signal. Usually, this is fixed in advance, either during manufacture or by programming. The transmitter and receiver typically have independent clocks, each several times faster than the serial bit rate. The sender uses its clock to transmit the data, and the receiver uses its clock to determine when to sense the data, synchronized by occurrence

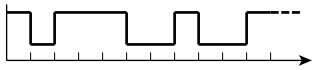


FIGURE 8.35 Serial transmission of the value 11100100 with start and stop bits.

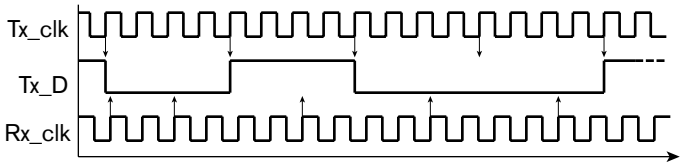


FIGURE 8.36 Generation and sampling of serial data using transmitter and receiver clocks.

of the start bit. This is illustrated in Figure 8.36, in which the transmit clock and receive clock have slightly different frequencies and are not related in phase. Provided the difference is not too extreme, the drift from the nominal sampling time does not affect correct reception of the transmitted data.

Historically, computer component manufacturers provided a component called a *universal asynchronous receiver/transmitter*, or *UART*, for serial communications ports. The software on the computer could program the bit rate and other parameters. UARTs are still useful in some applications for connecting remote devices to digital systems via serial communications links. For example, an instrumentation system with remote sensors that transmit data at relatively low bit rates can use serial transmission managed by UARTs.

The third scheme for synchronizing a serial transmitter and receiver involves combining a clock with the data on the same signal wire. This avoids the need for tight clock synchronization, since there is an indication of when each bit arrives. As an example of such a scheme, we will describe *Manchester encoding*. As with NRZ transmission, Manchester encoding transmits each bit of data in a given interval. However, rather than representing each bit using one or other logic level, it represents a 0 with a transition from low to high in the middle of the bit interval, and a 1 with a transition from high to low. (We could equally well choose the opposite assignment of transmissions, so long as transmitter and receiver agree.) At the beginning of the bit interval, a transition may be necessary to set the signal to the right logic level for the transition in the middle of the interval. Manchester encoding of the value 11100100 is shown in Figure 8.37, with the most significant bit transmitted first and with bit intervals defined by the transmitter's clock.

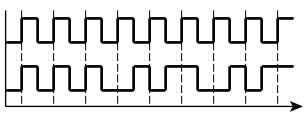


FIGURE 8.37 Manchester encoding of the value 11100100.

Since Manchester encoding of data is synchronized with the transmitter's clock and that clock is combined with the data, the receiver must be able to recover the transmitted clock and data from the signal. It does so using a circuit called a *phase-locked loop* (PLL), which is an oscillator whose phase can be adjusted to line up with a reference clock signal. A system using Manchester encoding usually transmits a continuous sequence of encoded 1 bits before transmitting one or more data words. The encoding of such a sequence gives a signal that matches the transmitter's clock.

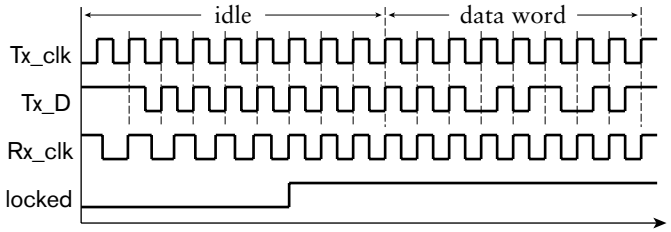


FIGURE 8.38 Synchronization of transmit and receive clocks by a PLL.

The receiver's PLL locks onto the signal to give a clock that can be used to determine the bit intervals for the transmitted data. This is shown in Figure 8.38.

The main advantage of Manchester encoding over NRZ transmission is that it contains sufficient transitions to allow clock synchronization without the need for separate signal wires. The disadvantage is that the bandwidth of the transmission is double that of NRZ transmission. However, for many applications, that is not an overriding disadvantage. Manchester encoding has been used in numerous serial transmission standards, including the original Ethernet standard. Other serial encoding schemes that are similar in concept but more involved are now becoming widely used.

8.4.2 SERIAL INTERFACE STANDARDS

Given the advantages of serial transmission over parallel transmission for applications where distance and cost are significant considerations, numerous standards have been developed. These standards cover two broad areas of serial interfaces: connection of I/O devices to computers, and connection of computers together to form a network. Since most digital systems contain embedded computers, they can include standard interfaces for connecting components. The benefits of doing so include avoiding the need to design the connection from scratch, and being able to use off-the-shelf devices that adhere to standards. As a consequence, we can reduce the cost of developing and building systems, as well reducing the risk of designs not meeting requirements.

Some examples of serial interface standards for connecting I/O devices include:

- RS-232: This standard was originally defined in the 1960s for connecting teletype computer terminals with modems, devices for serial communication with remote computers via phone lines. Subsequently, the standard was adopted for direct connection of terminals to computers. Since most computers included RS232 connection ports, RS232 connections were incorporated in I/O devices

other than terminals as a convenient way to connect to computers. Examples included user-interface devices such as mice, and various measurement devices. Serial transmission in RS232 interfaces uses NRZ encoding with start and stop bits for synchronization. Data is usually transmitted with the least significant bit first and most significant bit last. While RS232 interfaces have now largely been supplanted by more recent standards, they are still used in some equipment, for example, bar code readers in point-of-sale terminals, and industrial measurement devices.

- I^2C: The Inter-Integrated Circuit bus specification is defined by Philips Semiconductors, and is widely adopted. It specifies a serial bus protocol for low-bandwidth transmission between chips in a system (10kbit/sec to 3.4Mbit/sec, depending on the mode of operation). It requires two signals, one for NRZ-coded serial data and the other for a clock. The signals are driven by open-drain drivers, allowing any of the chips connected to the bus to take charge by driving the clock and data signals. The specification defines particular sequences of logic levels to be driven on the signals to arbitrate to see which device takes charge and to perform various bus operations. The advantage of the I^2C bus is its simplicity and low implementation cost in applications that do not have high performance requirements. It is used in many off-the-shelf consumer and industrial control chips as the means for an embedded microcontroller to control operation of the chip. Philips Semiconductor has also developed a related bus specification, I^2S, or Inter-IC Sound, for serial transmission of digitally encoded audio signals between chips, for example, within a CD player.

- USB: The Universal Serial Bus is specified by the USB Implementers Forum, Inc., a nonprofit consortium of companies founded by the original developers of the bus specification. USB has become commonplace for connecting I/O devices to computers. It uses differential signaling (see Section 6.4.1) on a pair of wires, with a modified form of NRZ encoding. Different configurations support serial transfer at 1.5Mbit/sec, 12Mbit/sec or 480Mbit/sec. The USB specification defines a rich set of features for devices to communicate with host controllers. Since there is such a diversity of devices with USB interfaces, application-specific digital systems can benefit from inclusion of a USB host controller to enable connection of off-the-shelf devices. USB interface designs for inclusion in ASIC and FPGA designs are available in component libraries from vendors.

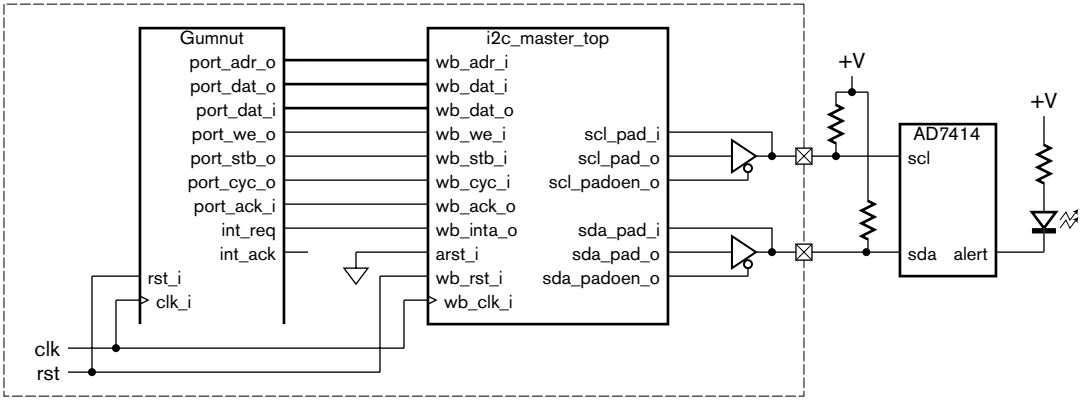
- FireWire: This is another high-speed bus defined by IEEE Standard 1394. Whereas USB was originally developed for lower bandwidth devices and subsequently revised to provide higher bandwidth,

FireWire started out as a high-speed (400Mbit/sec) bus. There is also a revision of the standard defining transfer at rates up to 3.2Gbit/sec. FireWire connections use two differential signaling pairs, one for data and the other for synchronization. As with USB, there is a rich set of bus operations that can be performed to transmit information among devices on the bus. FireWire assumes that any device connected to the bus can take charge of operation, whereas USB requires a single host controller. Thus, there are some differences in the operations provided by FireWire and USB, and some differences in the applications for which they are suitable. FireWire has been most successful in applications requiring high-speed transfer of bulk data, for example, digital video streams from cameras.

EXAMPLE 8.11 Design an interface to connect an embedded Gumnut core to a remote temperature sensor. The temperature sensor is an Analog Devices AD7414 with an I^2C connection and an alert output that can be connected to a warning indicator.

SOLUTION The OpenCores repository (see Section 8.7, Further Reading) contains an I^2C controller component that is Wishbone compliant. We can use it rather than designing a new I^2C controller from scratch. We connect the controller to the Gumnut core's Wishbone I/O bus, and provide pad connections to an external I^2C bus for connecting the temperature sensor. We connect the alert output of the sensor to an LED indicator. The sensor allows the embedded software to program threshold temperatures, beyond which the alert indicator is activated. The system design is shown in Figure 8.39. The use of the serial I^2C bus allows connection to the temperature sensor with only two wires, resulting in a substantial reduction in system cost compared to connection using a parallel bus.

FIGURE 8.39 A temperature sensing system using an I^2C serial bus.



KNOWLEDGE TEST QUIZ

1. What advantages does serial transmission of data have over parallel transmission?
2. How do we convert between parallel and serial form for serial data transmission or reception?
3. What determines the order in which we transmit bits of data?
4. What is meant by non-return to zero (NRZ) transmission?
5. What is the purpose of a start bit and a stop bit in serial transmission?
6. How does Manchester encoding represent 0 and 1 bits?
7. Why would we adopt a standard serial interface specification rather than developing a custom interface?
8. Which of I^2C or FireWire would be most appropriate for connecting a motor controller and a digital video camera, respectively, to an embedded system?

8.5 I/O SOFTWARE

Now that we have described the hardware aspects of input and output, we turn our attention to the corresponding embedded software. We have seen that an `out` instruction in the Gumnut core invokes a port write operation to update an output register in an I/O controller, and an `inp` instruction invokes a port read operation to get the value from an input register. The embedded software running on the core needs to use `out` and `inp` instructions as part of the task of managing input and output devices to implement the functionality required of the system.

Since I/O devices interact with the real physical world, the embedded software needs to be able to respond to events when they occur, or to cause events at the right time. Dealing with *real time* behavior is one of the main differences between embedded software and programs for general purpose computers. Embedded software needs to be able to detect when events occur so that it can react. It also needs to be able to keep track of time so that it can perform actions at specific times or at regular intervals. In this section, we will introduce the basic mechanisms for synchronizing embedded software with I/O events.

8.5.1 POLLING

The simplest I/O synchronization mechanism is called *polling*. It involves the software repeatedly checking a status input from a controller to see if an event has occurred. If it has, the software performs the necessary task to react to the event. If there are multiple controllers, or multiple events to

which the software must respond, the software checks each of the status inputs in turn, reacting to events as they occur, as part of a busy loop.

EXAMPLE 8.12 A factory automation system includes a safety monitoring subsystem based on an embedded Gumnut core. The core has alarm inputs from a number of machines that indicate various abnormal operating conditions. These are connected through a controller that has two input registers at addresses 16 and 17. Each bit of each register represents one alarm input, with the bit being 0 for normal operation and 1 for an alarm condition. The core also has a temperature sensor connected to an ADC. The converted value is available in an input register at address 20, represented as an 8-bit unsigned integer in °C. A temperature above 50°C is abnormal. The core has an output register at address 40. Writing a 1 to the least significant bit of the output register activates an alarm bell, and writing 0 deactivates it. Develop a polling loop for the embedded software to monitor the inputs and activate the alarm bell when any abnormal condition arises.

SOLUTION The polling loop must repeatedly read the input registers. If any alarm input bit is 1, or if the temperature value is greater than 50°C, the alarm bell output bit must be set to 1; otherwise, it must be cleared to 0. The code is

```
alarm_in_1: equ 16   ; address of alarm_in_1 input register
alarm_in_2: equ 17   ; address of alarm_in_2 input register
temp_in:    equ 20   ; address of temp_in input register
alarm_out:  equ 40   ; address of alarm_out output register

max_temp:   equ 50   ; maximum permissible temperature

poll_loop:  inp  r1, alarm_in_1
            sub  r0, r1, 0
            bnz  set_alarm ; one or more alarm_in_1 bits set
            inp  r1, alarm_in_2
            sub  r0, r1, 0
            bnz  set_alarm ; one or more alarm_in_2 bits set
            inp  r1, temp_in
            sub  r0, r1, max_temp
            bnc  set_alarm ; temp_in > max_temp
            out  r0, alarm_out ; clear alarm_out
            jmp  poll_loop
set_alarm:  add  r1, r0, 1
            out  r1, alarm_out ; set alarm_out bit 1 to 1
            jmp  poll_loop
```

Polling has the advantage that it is very simple to implement, and requires no additional circuitry beyond the input and output registers of the I/O controllers. However, it requires that the processor core be

continually active, consuming power even when there is no event to react to. It also prevents the processor from reacting immediately to one event if it is busy dealing with another event. For these reasons, polling is usually only used in very simple control applications where there is no need for fast reaction times.

8.5.2 INTERRUPTS

Probably the most common way to synchronize embedded software with I/O events is through use of *interrupts*. The processor executes some background tasks, and when an event occurs, the I/O controller that detects the event interrupts the processor. The processor then stops what it was doing, saving the program counter so that it can resume later, and starts executing an *interrupt handler*, or *interrupt service routine*, to respond to the event. When it has completed the handler, it restores the saved program counter and resumes the interrupted program. In some systems, if there is no background task to run, the processor may enter a low-power standby state from which it emerges in response to an interrupt. This has the benefit of avoiding power consumption due to busy-waiting, though it may add delay to the interrupt response time if the processor requires some time to resume full-power operation.

Different processors provide different mechanisms for I/O controllers to request an interrupt. Some provide very simple mechanisms, such as that of the Gumnut core that we will describe shortly. Others provide more elaborate mechanisms, for example, allowing different controllers to be assigned different priorities, so that a higher-priority event can interrupt service of a lower-priority event, but not *vice versa*. Some provide a way for the controller to select the interrupt handler to be executed by the processor. However, there are some aspects that are common to most systems.

First, the processor must have an input signal to which controllers can connect to request interrupts. For older microprocessors and microcontrollers, the interrupt request signal is often an active-low signal pulled up with an external resistor. Each controller connects to the signal with an open-drain or open-collector driver, pulling the signal low to request an interrupt. Thus, the signal value is a wired-OR function of the individual controllers' requests. For processor cores that are designed to connect to on-chip I/O controllers, the interrupt request input is typically driven by active gates forming the OR of the controllers' requests.

Second, the processor must be able to prevent interruption while it is executing certain sequences of instructions, often called *critical regions*. Examples are instructions that update information shared between an interrupt handler and other parts of the embedded software. If the processor is part way through updating such information and is interrupted, the interrupt handler will see the partially updated information, which may not

correctly represent a valid value. So processors generally have instructions or other means of *disabling interrupts* and *enabling interrupts*.

Third, the processor must be able to save sufficient information about the program it was executing when interrupted so that it can resume the program on completion of the interrupt handler. At the least, this includes saving the program counter value. Since the processor responds to an interrupt after completing one instruction and before starting the next, the program counter contains the address of the next instruction in the program. That is the instruction to be resumed after the interrupt handler. The processor must provide a register or some other storage in which to save the program counter. If there is other state information in the processor that might be modified by the interrupt handler, such as condition code bits, they must also be saved and restored.

Fourth, when the processor responds to an interrupt, it must disable further interrupts. Since response to an interrupt involves saving the interrupted program's state in registers, if the interrupt handler is itself interrupted, the saved state would be overwritten. Thus, the handler needs to prevent interruption, at least during the initial stages of responding to an interrupt.

Some processors allow the storage containing the saved state information to be read by a program. That allows a handler to copy the saved state into memory. The handler can then re-enable interrupts, allowing the interrupt handler itself to be interrupted to deal with another event. We call this *nested interrupt* handing. The handler must disable interrupts again when it has completed its operation so that it can restore the saved state before resuming the interrupted program.

Fifth, the processor must be able to locate the first instruction of the interrupt handler. The simplest way of doing this is for the handler to start at a fixed or predetermined address in the instruction memory. Alternative schemes involve the interrupting controller providing a *vector*: either a value used to form the address of the handler, or an index into a table of addresses in memory.

Finally, the processor needs an instruction for the interrupt handler to return to the interrupted program. Such a return from interrupt instruction restores the saved program counter and any other saved state.

The Gumnut processor core has all of these features, with the exception of nested interrupt handing. It has an input signal, int_req, that controllers can drive to 1 to request an interrupt. It includes two instructions in its instruction set: disi, for disabling interrupts; and enai, for enabling interrupts. When the core responds to an interrupt, it saves the program counter and the values of the Z and C condition codes in special internal registers, and disables further interrupts. The first instruction of the interrupt handler is located at address 1 in the instruction memory, so the processor simply loads that address into the program counter to start

executing the handler. Finally, the Gumnut instruction set includes the reti instruction to return from an interrupt handler. It restores the saved values to the program counter and the Z and C condition code bits, and re-enables interrupts. Program execution then resumes from where it left off.

There are also requirements on I/O controllers that make interrupt requests. When an event occurs, the controller must activate the processor's interrupt request signal. However, the processor may not respond immediately. The requesting controller must keep the request signal active, otherwise the request may go unnoticed. Failure to respond to an event may be a critical error in some systems. Processors typically have a mechanism to *acknowledge* an interrupt request, that is, to indicate that the event has been noticed and that the interrupt handler as been activated. If there are multiple I/O controllers that can request interrupts, the processor needs to acknowledge each request individually, so that none are overlooked. Once a request has been acknowledged, the controller must deactivate the interrupt request signal. Otherwise, multiple responses might occur for the one event. In some cases, that can be as bad as missing an event.

The Gumnut core provides a simple interrupt acknowledgment mechanism. It has an output signal, int_ack, that it drives to 1 for one cycle when it responds to an interrupt request. If there is only one controller that can request interrupts in a Gumnut system, the controller can use the int_ack signal to clear its interrupt request state.

EXAMPLE 8.13 Design an input controller that has 8-bit binary-coded input from a sensor. The value can be read from an 8-bit input register. The controller should interrupt the embedded Gumnut core when the input value changes. The controller is the only interrupt source in the system.

SOLUTION The controller contains a register for the input value. Since we need to detect changes in the value, we also need a register for the previous value, that is, the value on the previous clock cycle. When the current and previous values change, we set an interrupt-request state bit. Since there is only one interrupt source, we can use the int_ack signal from the processor core to clear the state bit. The controller circuit is shown in Figure 8.40.

EXAMPLE 8.14 Develop a Verilog model of the input controller of Example 8.13.

SOLUTION The module definition includes ports for the I/O bus, plus the interrupt request and acknowledge connections:

```
module sensor_controller ( input           clk_i, rst_i,
                           input           cyc_i, stb_i,
                           output          ack_o,
```

(*continued*)

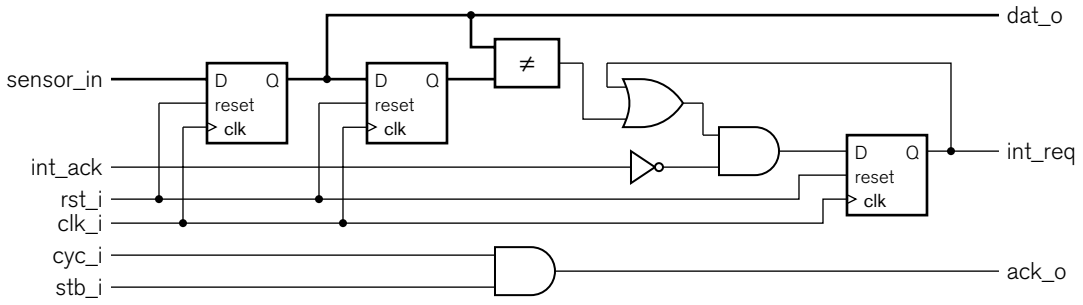


FIGURE 8.40 Circuit for an input controller with interrupt request logic.

```
                          output reg [7:0] dat_o,
                          output           reg int_req,
                          input            int_ack,
                          input      [7:0] sensor_in );

  reg [7:0] prev_data;

  always @(posedge clk_i) // Data registers
    if (rst_i) begin
      prev_data <= 8'b0;
      dat_o     <= 8'b0;
    end
    else begin
      prev_data <= dat_o;
      dat_o     <= sensor_in;
    end

  always @(posedge clk_i) // Interrupt state
    if (rst_i) int_req <= 1'b0;
    else
      case (int_req)
        1'b0: if (dat_o != prev_data) int_req <= 1'b1;
        1'b1: if (int_ack)            int_req <= 1'b0;
      endcase

  assign ack_o = cyc_i & stb_i;

endmodule
```

The first always block represents the two data registers, one for the current sensor data value and one for the previous value. The second always block represents the interrupt request and acknowledge logic. It is essentially a small finite state machine, with int_req encoding the state. In the state where int_req is 0, there is no interrupt request. However, if the current value changes from the previous value, int_req is set to 1. The value of this output is used as the interrupt request signal to the processor. It stays 1, even when the current value and the previous value no longer differ. Eventually, when the processor responds to the interrupt and sets int_ack to 1, int_req is cleared back to 0.

EXAMPLE 8.15 Show the Gumnut assembler code for the interrupt handler for the sensor controller interrupt. Assume the data register is read at port address 0.

SOLUTION The interrupt handler is

```
            data
saved_r1:   bss   1

            text
sensor_data: equ  0       ; address of sensor data
                          ; input register

            org   1
            stm   r1, saved_r1
            inp   r1, sensor_data
            ...           ; process the data
            ldm   r1, saved_r1
            reti
```

Since the handler needs to use processor register r1, it must save whatever value is in there from the interrupted program. The data memory location saved_r1 is reserved for that purpose. The interrupt handler must be located at address 1 in the instruction memory. We use an org directive to ensure this. The instructions in the handler first save the contents of r1, then read the new value from the controller's input register. The handler then executes instructions that deal with the data. Finally, the handler restores the saved value to r1 and uses a reti instruction to resume the interrupted program.

If, in a Gumnut-based system, there are several controllers that can request an interrupt, the interrupt handler must be able to determine which controller requested an interrupt so that it can execute the appropriate response. In such a system, each controller must provide status information in a status register that indicates whether it has requested an interrupt. Furthermore, the int_ack signal is not sufficient to distinguish which request is acknowledged. Instead, the processor must perform some other action to acknowledge the interrupt. We could acknowledge and clear a controller's interrupt request as a side-effect of its status register being read. Alternatively, we could require a write operation to a control register to acknowledge the request.

8.5.3 TIMERS

As we mentioned earlier, many real-time embedded systems must perform actions at specific times or at periodic intervals. For these systems, we need to include some form of timer. We showed in Chapter 4 that we can

use a counter to derive a periodic signal from the system clock. We can use such a signal as a time base: each cycle represents one unit of time in the embedded system. We also showed how we can use a loadable down counter as an interval timer. A common use for interval timers in real-time embedded systems is to generate an interrupt for the processor at some programmable multiple of a time base. The interval timer acts as an I/O controller, often called a *real-time clock*, with an output register for programming the time interval. The interrupt handler for the timer can then perform any required periodic actions.

EXAMPLE 8.16 Develop a Verilog model for a real-time clock controller for the Gumnut processor. The controller has a 10μs time base derived from a 50MHz system clock, and an 8-bit output register for the value to load into the counter. A write operation to the output register causes the counter to be loaded. After the counter reaches 0, it reloads the value from the output register and requests an interrupt. The controller has an input register for reading the current count value. The counter also has a 1-bit control output register. When bit 0 of the register is 0, interrupts from the controller are masked, and when it is 1, they are enabled. The counter has a status register, in which bit 0 is 1 when the counter has reached 0 and been reloaded, or 0 otherwise. Other bits of the register are read as 0. Reading the register has the side effect of acknowledging a requested interrupt and clearing bit 0. The counter output and input registers are located at the base port address, and the control and status registers are at offset 1 from the base port address.

SOLUTION The module definition for the controller has ports for the I/O bus, and uses the stb_i port for the decoded base port address:

```
module real_time_clock ( input        clk_i, // 50MHz clock
                         input        rst_i,
                         input        cyc_i, stb_i, we_i,
                         output       ack_o,
                         input        adr_i,
                         input  [7:0] dat_i,
                         output [7:0] dat_o,
                         output       int_req );

  parameter clk_freq         = 50000000;
  parameter timebase_freq    = 100000;
  parameter timebase_divisor = clk_freq / timebase_freq;

  reg [7:0] count_value;
  reg       trigger_interrupt;
  reg       int_enabled, int_triggered;
```

(continued)

```
  integer   timebase_count;
  reg [7:0] count_start_value;

  always @(posedge clk_i)  // Counter
    if (rst_i) begin
      timebase_count    <= 0;
      count_start_value <= 8'b0;
      count_value       <= 8'b0;
      trigger_interrupt <= 1'b0;
    end
    else if (cyc_i && stb_i && !adr_i && we_i) begin
      timebase_count    <= 0;
      count_start_value <= dat_i;
      count_value       <= dat_i;
      trigger_interrupt <= 1'b0;
    end
    else if (timebase_count == timebase_divisor - 1) begin
      timebase_count <= 0;
      if (count_value == 8'b00000000) begin
        count_value       <= count_start_value;
        trigger_interrupt <= 1'b1;
      end else begin
        count_value       <= count_value - 1;
        trigger_interrupt <= 1'b0;
      end
    end
    else begin
      timebase_count    <= timebase_count + 1;
      trigger_interrupt <= 1'b0;
    end

  always @(posedge clk_i)  // Control register
    if (rst_i)
      int_enabled <= 1'b0;
    else if (cyc_i && stb_i && adr_i && we_i)
      int_enabled <= dat_i[0];

  always @(posedge clk_i) // Interrupt register
    if (rst_i || (cyc_i && stb_i && adr_i && !we_i))
      int_triggered <= 1'b0;
    else if (trigger_interrupt)
      int_triggered <= 1'b1;

  assign dat_o = !adr_i ? count_value : {7'b0, int_triggered};

  assign int_req = int_triggered & int_enabled;

  assign ack_o = cyc_i & stb_i;

endmodule
```

The first always block represents the time-base divider, interval counter and counter output register. The variable timebase_count is used to divide the 50MHz clock to derive the 100kHz time base, and the variable count_start_value stores the value for the counter output register. The count value is represented by the variable count_value. The variable trigger_interrupt is an internal control variable used to manage interrupt requests. On reset, the variables are cleared to zeros. When a port write operation is performed with the least significant address bit being 0, the written data is used to update count_start_value, and the counters are cleared to zeros again. On other clock cycles, the counters are incremented. When the time base counter reaches its terminal count, it wraps to zero, and count_value is decremented. When count_value reaches zero, it is reloaded from count_start_value, and the trigger_interrupt variable is set to 1.

The second always block represents the control register, containing the interrupt-enable bit. On reset, the bit is cleared to 0. Otherwise, when a write operation is performed with the least significant address bit being 1, the bit is updated with the written port data.

The third always block represents the one-bit state register that determines when an interrupt event has occurred. The variable int_triggered is set to 1 when the trigger_interrupt variable is 1, that is, when count_value is reloaded after having reached zero. The variable is cleared to 0 on reset, and also on a port read operation that reads the status register.

The remaining assignments implement the rest of the required functionality. The assignment to dat_o selects the value provided for a port read operation: either the count value or the interrupt status bit. The assignment to int_req causes an interrupt request when the triggering event has occurred and interrupt requests are enabled. The assignment to ack_o implements the controller's response to bus operations, indicating that the controller is ready without delay.

EXAMPLE 8.17 Suppose a Gumnut system includes the real-time clock controller of Example 8.16 with the registers located at base port address 16. Develop Gumnut code that calls the subroutine task_2ms every 2ms. In between activations, the program stands by in low-power mode. The subroutine should not be called as part of the interrupt handler, since other interrupts should be permitted during execution of the subroutine.

SOLUTION The code is

```
;;; -------------------------------------------------------
;;; Program reset: jump to main program

              text
              org      0
              jmp      main
```

(continued)

```
;;; ----------------------------------------------------------
;;; Port addresses
rtc_start_count:      equ    16 ; data output register
rtc_count_value:      equ    16 ; data input register
rtc_int_enable:       equ    17 ; control output register
rtc_int_status:       equ    17 ; status input register

;;; ----------------------------------------------------------
;;; Interrupt handler

                      data
int_r1:               bss    1 ; save location for
                                ; handler registers

                      text
                      org 1

int_handler:          stm    r1, int_r1 ; save registers
check_rtc:            inp    r1, rtc_status ; check for
                                            ; RTC interrupt
                      sub    r0, r1, 0
                      bz     check_next
                      add    r1, r0, 1
                      stm    r1, rtc_int_flag ; tell main
                                              ; program
check_next:           ...

int_end:              ldm     r1, int_r1 ; restore registers
                      reti

;;; ----------------------------------------------------------
;;; init_interrupts:  Initialize 2ms periodic interrupt, etc.

                      data
rtc_divisor:          equ    199     ; divide 100 kHz down
                                     ; to 500 Hz
rtc_int_flag:         bss    1

                      text
init_interrupts:      add    r1, r0, rtc_divisor
                      out    r1, rtc_start_count
                      add    r1, r0, 1
                      out    r1, rtc_int_enable
                      stm    r0, rtc_int_flag
                      ...    ; other initializations
                      ret

;;; ----------------------------------------------------------
```

(*continued*)

```
;;; main program

                text
main:           jsb     init_interrupts
                enai
main_loop:      stby
                ldm     r1, rtc_int_flag
                sub     r0, r1, 1
                bnz     main_next
                jsb     task_2ms
                stm     r0, rtc_int_flag
main_next:      ...
                jmp     main_loop
```

The code is structured into separate sections and subroutines, each dealing with one part of the program. The first section deals with starting the main program when the system is reset. The instructions are located at address 0, and simply jump to the main program. The second section defines symbolic labels for the real-time clock controller registers. Reference to these labels makes the code easier to understand.

The subroutine init_interrupts initializes the real-time clock controller. It loads the value 199 into the controller's output register. This makes the controller count down from 199 to 0 and then restart from 199; thus, it divides the time base by 200 to give a 2ms period. The subroutine also sets the controller's interrupt-enable bit by writing 1 to the control register, and clears the rtc_int_flag location in memory. This location is used by the interrupt handler to indicate to the main program that a 2ms interrupt has occurred. The subroutine then proceeds with other initializations before returning to the caller.

The interrupt handler is located at instruction address 1. On responding to an interrupt, it checks the controllers in the system to determine the interrupt source, starting with the real-time clock controller. If the controller's status register is nonzero, the handler sets rtc_int_flag to 1, indicating to the main program that it should perform the 2ms task. The handler then proceeds to check for other interrupt sources before returning to the interrupted program.

The main program starts by calling the subroutine to initialize controllers and interrupts, then enables receipt of interrupts. It then stands by in low-power mode until an interrupt occurs. On return from the interrupt handler, the main program checks the rtc_int_flag location. If it is 1, a real-time clock interrupt has occurred, so the main program calls the task_2ms subroutine, as required, and then clears rtc_int_flag. The main program then performs any processing required for other interrupts that might have occurred. When that it done, it loops back and stands by for the next interrupt.

The code in Example 8.17 is a basic form of *real-time executive*, that is, a control program that schedules execution of tasks in response to interrupts and timer events. Vendors of microprocessors, microcontrollers and embedded processor cores generally provide more sophisticated *real-time operating systems* (RTOSs) for their products. There are also a number of third-party vendors who provide RTOSs that run on various processors. An RTOS generally includes an executive, together with software components to manage other resources, such as storage, input/output, communication and specialized processing resources. The advantage of using a real-time executive or an RTOS is that we can focus our software development effort on the aspects of our system that are different from other systems, and reuse proven code that deals with common embedded software mechanisms. We won't go into any further detail of real-time programming in this book. Instead, we refer to sources on the topic listed in the Further Reading section.

KNOWLEDGE TEST QUIZ

1. In dealing with real-time behavior, what does embedded software need to do?
2. How does polling synchronize embedded software with I/O events?
3. Identify an advantage and a disadvantage of polling compared to other I/O synchronization schemes.
4. What action does a processor perform upon receiving an interrupt?
5. How does a processor prevent interruption while it is executing a critical region?
6. How does the processor determine where to resume program execution on completion of handling an interrupt?
7. What is an interrupt vector?
8. Why must a controller deactivate the interrupt request signal when its interrupt is acknowledged?
9. What purpose does a real-time clock serve in an embedded system?
10. What operations are performed by a real-time executive?

8.6 CHAPTER SUMMARY

- Transducers allow a digital system to interact with the physical world. Sensors generate an electrical representation of a physical property. Output transducers, including actuators, cause a physical effect.
- Input devices include switches, keypads, knobs, position encoders, and analog sensors.
- An analog-to-digital converter (ADC) produces a binary coded representation of an analog signal. ADCs include flash and successive-approximation ADCs.
- Output devices include indicator lights, 7-segment LED and LCD displays, electromechanical actuators and valves, motors, and analog output devices.
- A digital-to-analog converter (DAC) produces an analog signal proportional to a binary coded input. DACs include R-string and R/2R ladder DACs.
- An I/O controller includes input and output registers that provide an embedded processor with access to I/O data. It may also include control and status registers for managing operation of the controller.
- An autonomous controller may perform I/O operations while a processor performs other tasks concurrently.
- Buses connect multiple data sources and destinations. Parallel buses use one signal wire per bit of encoded data.
- Multiplexed buses use multiplexers to select data from one source at a time. Multiplexers can be centralized or distributed, depending on the wiring complexity of the system.
- Tristate buses allow direct connection of sources to destinations, using a high-impedance driver state to avoid contention. Tristate buses are not generally used within chips. The high-impedance state is modeled in Verilog using the Z value.
- Open-drain and open-collector drivers allow wired-AND connections, modeled in Verilog using `wand` nets.
- Bus protocols specify the signals used and the sequences and timing of values to implement bus operations.
- Serial buses transmit bits in sequence over one wire. Shift registers are used to convert between parallel and serial transmission.

- Serial transmission requires synchronization between transmitter and receiver to determine the interval during which each bit is transmitted.
- Real-time software on an embedded processor must be able to react to I/O events and to keep track of time so that it can perform scheduled or periodic operations.
- Software can poll I/O controllers to determine when events occur.
- Interrupts are a mechanism for a controller to notify a processor of an event. The processor executes an interrupt handler to respond to the event, then resumes its interrupted task. The processor includes instructions for managing interrupts.
- Timers, or real-time clocks, issue periodic interrupts, allowing an embedded system to perform scheduled and periodic tasks.

8.7 FURTHER READING

Industrial Electronics: Applications for Programmable Controllers, Instrumentation and Process Control, and Electrical Machines and Motor Controls, 3rd Edition, Thomas E. Kissell, Prentice Hall, 2003. This is a comprehensive reference describing the kinds of input and output devices encountered in industrial settings, and the transducers and electronic circuits used to interface them to digital control systems.

Standard LCD Graphic Modules, Allshore Industries, www.allshore.com/lcd_displays/lcd_graphic_modules.asp. Provides data sheets on the ASI-D-1006A-DB-_S/W LCD module and the Seiko Epson SED1560 controller IC described in Section 8.2.2.

Understanding Digital Signal Processing, Richard G. Lyons, Prentice Hall, 2001. An introduction to the theory of digital signal processing (DSP).

WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores, Revision B.3, OpenCores Organization, 2002, www.opencores.org/projects.cgi/web/wishbone/wbspec_b3.pdf. This is the specification document for the Wishbone bus used in this book.

OpenCores, www.opencores.org. From the website's FAQ, "OpenCores is a loose collection of people who are interested in developing hardware, with a similar ethos to the free software movement." The website hosts a repository of freely reusable core designs, many of which are compatible with the Wishbone bus.

Real-Time Concepts for Embedded Systems, Qing Li, Caroline Yao, CMP, 2003. A practical introduction to real-time programming for embedded systems.

EXERCISES

EXERCISE 8.1 A calculator has keys arranged as shown in Figure 8.41. Show how the key switches can be arranged in a scanned matrix.

MC	M–	√	AC
MR	M+	%	CE
7	8	9	÷
4	5	6	×
1	2	3	–
0	.	=	+

FIGURE 8.41

EXERCISE 8.2 Design a keypad controller to connect a Gumnut core to the keypad described in Exercise 8.1. The controller should include an output register for driving row lines and an input register for sensing column lines.

EXERCISE 8.3 Develop a Gumnut program that uses the keypad controller described in Exercise 8.2 to scan the calculator keypad. When a key is pressed, the program should call a subroutine labeled `do_key` to respond to the key press. (Just include the subroutine call, not the instructions in the subroutine.) Assume the output register is at port address 0 and the input register is at port address 1, and omit switch debouncing.

EXERCISE 8.4 Show how the input controller described in Example 8.13 on page 364 can be used for a volume control knob with an incremental encoder.

EXERCISE 8.5 Develop a Gumnut interrupt handler that responds to interrupts from the incremental encoder input of Exercise 8.4. The handler should increment or decrement a value stored in memory as the knob is turned clockwise or counterclockwise, respectively. The value should be limited to the range 0 to 100.

EXERCISE 8.6 Develop a Verilog model of an 8-bit successive approximation register (SAR) for use in an ADC (see Figure 8.6 on page 320).

EXERCISE 8.7 Develop a Gumnut subroutine to perform an analog-to-digital conversion using successive approximation, returning an 8-bit result in register r1. The Gumnut is connected to an output data register, an input status register, an 8-bit DAC and a comparator as shown in Figure 8.42. The output data register is written at port address 8. The input status register is read at port address 8, and provides the value of the comparator output in the least significant bit, with other bits hardwired to 0.

EXERCISE 8.8 Some digital audio applications use an LED bar display, consisting of a row of LED indicators to display the volume level of the audio signal. Assuming that the loudness is proportional to the logarithm of the signal amplitude, we can work out which LEDs to light by finding the left-most 1 bit in the unsigned binary number representing the amplitude. Design a circuit to drive an 8-LED common-anode bar display, given an 8-bit unsigned binary amplitude value.

EXERCISE 8.9 Write a Gumnut subroutine that performs the function of the circuit described in Exercise 8.8. The subroutine takes an 8-bit unsigned

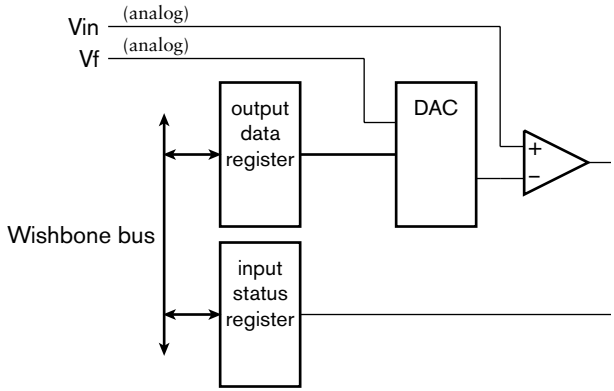


FIGURE 8.42

binary amplitude value in r2 and outputs a corresponding value to an 8-bit register at port address 28, connected to the cathodes of an 8-LED common-anode bar display.

EXERCISE 8.10 Draw a schematic of a circuit corresponding to the display multiplexer of Example 8.2 on page 323.

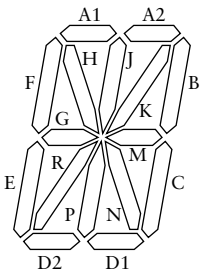


FIGURE 8.43

EXERCISE 8.11 A 16-segment LED display, shown in Figure 8.43, can display alphabetic and numeric characters. Develop a circuit schematic and a Verilog model of a display decoder and driver to drive a 16-segment common anode LED display, given a 6-bit character-code input. Use a 64×16-bit ROM to decode the input. You needn't determine the ROM content for this exercise.

EXERCISE 8.12 Modify the display multiplexer/decoder design of Example 8.2 on page 323 to provide an 8-character alphanumeric scanned display, with eight 6-bit character code inputs. Use the ROM described in Exercise 8.11 to decode the character codes.

EXERCISE 8.13 Design an output controller to drive eight solenoids. The controller should have an 8-bit output register, and should connect to the Wishbone bus used by the Gumnut core.

EXERCISE 8.14 The ST Microelectronics L298 IC is a dual full-bridge driver that can be used to drive the kind of stepper motor shown in Figure 8.12 on page 327. The connections between the L298 and the motor (in simplified form) are shown in Figure 8.44. Determine the sequences of values on the inputs to the L298 to drive the stepper motor clockwise and counterclockwise.

EXERCISE 8.15 Assume the stepper motor driver described in Exercise 8.14 is connected to a Gumnut core through a 6-bit output register at port address 8, with bits 0 to 5 of the register controlling signals `in1`, `in2`, `en_a`, `in3`, `in4` and `en_b`, respectively. Write a Gumnut subroutine to step the motor one-quarter turn, either clockwise, if r2 is 0, or counterclockwise, if r2 is 1. Hint: The subroutine will need to keep track of the current state of the stepper motor control signals. Use a location in memory to save the state.

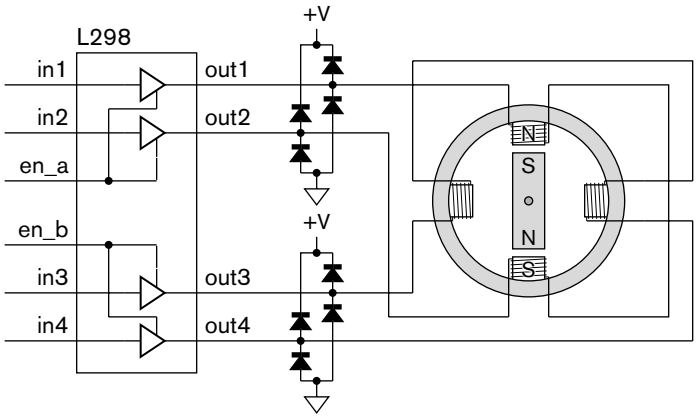


FIGURE 8.44

EXERCISE 8.16 Draw a diagram showing how the following components might be used to construct a handheld voice recorder: microphone, microphone amplifier, loudspeaker, loudspeaker amplifier, ADC, DAC, processor core, instruction memory, data memory, push-button switches. The recorder has buttons to record, play/pause, stop, skip forward, and skip backward.

EXERCISE 8.17 Draw a diagram similar to Figures 8.19 and 8.20 on page 339 showing multiplexed bus connection of two data sources, two data destinations, and two components that are both sources and destinations.

EXERCISE 8.18 Revise Figure 8.21 on page 340 to omit the second ADC controller.

EXERCISE 8.19 Revise the Verilog model in Example 8.6 on page 341 to omit the second ADC controller.

EXERCISE 8.20 Revise the Verilog model of Example 8.8 on page 346 to output X values if the enable inputs are Z or X.

EXERCISE 8.21 Design a serial output controller for connection to the Gumnut core using the Wishbone bus. The controller should transmit each 8-bit data byte written to a data register using NRZ encoding with one start bit and one stop bit, as shown in Figure 8.35 on page 355. Transmission should occur at 9600 bits per second, with a transmit timing derived from a system clock with frequency 39.321600MHz ($= 9600 \times 4096$). When the stop bit has been transmitted, the controller should set an interrupt request output. The interrupt request output should be reset when the Gumnut int_ack signal is 1.

EXERCISE 8.22 Write a Gumnut subroutine to transmit a byte of data using the serial output controller of Exercise 8.21. Assume the data register is a port address 24 and that there are no other interrupt sources in the system. The subroutine should wait in standby mode and not return until the controller interrupts to indicate that the transmission is complete.

EXERCISE 8.23 Revise the subroutine of Exercise 8.22 so that the subroutine returns after having written the byte to the data register. This allows the processor to continue with other work while the controller transmits the byte. You will need to keep track of whether the controller is busy so that a subsequent call to the subroutine does not overwrite the data register while transmission is still in progress.

EXERCISE 8.24 Develop a Verilog model of the serial output controller of Exercise 8.21.

EXERCISE 8.25 The OpenCores repository includes a UART core, uart16550, that uses the Wishbone bus. (See http://www.opencores.org/projects.cgi/web/uart16550/overview.) Develop a Verilog structural model of a system containing a Gumnut core, instruction and data memories, and an instance of the UART core.

EXERCISE 8.26 Draw a diagram similar to Figure 8.37 on page 356 showing Manchester encoding of the values 01100101 and 11110000.

EXERCISE 8.27 Design a circuit that has, as input, a transmit clock and an NRZ serial data signal (as in Figure 8.33 on page 354), and that generates a Manchester encoded serial data signal as output.

EXERCISE 8.28 Show how the system described in Example 8.11 on page 359 would be extended to connect to four AD7414 sensors.

EXERCISE 8.29 A Gumnut system includes a 4-digit 7-segment display, connected as shown in Figure 8.45. The anode data register is at port address 128, and the cathode data register is at port address 129. Write Gumnut assembly code for the task_2ms subroutine described in Example 8.17 on page 369 to scan the display. The BCD digits to display are stored in four bytes of memory labeled display_data. The subroutine should select one digit to drive each time it is called. Thus, four successive calls are required for a complete scan.

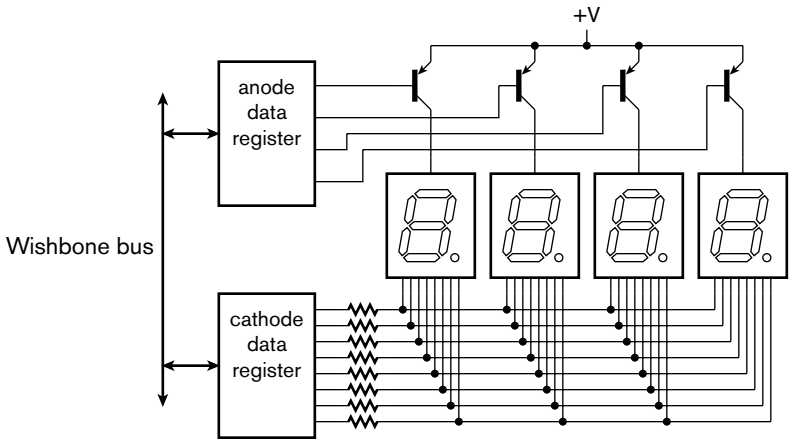


FIGURE 8.45

9 ACCELERATORS

In Section 7.1, as part of our introduction to embedded computer organization, we mentioned accelerators as optional components in embedded systems. If the system must perform some operation faster than is possible with embedded software running on a processor core, we can design custom hardware to perform the operation at the required speed. In this chapter, we will examine accelerators in more details and identify how they interact with an embedded processor.

9.1 GENERAL CONCEPTS

Many operations performed by digital systems consist of a number of steps. If a simple embedded processor core performs an operation, it performs the steps in sequence, with each step using one or more processor instructions. The rate at which the processor can execute instructions places a lower bound on the time it takes to perform the operation. The key to accelerating performance is *parallelism*: performing multiple steps at the same time, thus taking less time overall to complete the operation. The cost of parallelism is the additional components needed to perform the steps in parallel, since each component can only perform one step at a time. However, if sequential execution does not meet performance requirements, parallel hardware may be a higher-performance and lower-power alternative to using a faster (and more expensive) processor.

One place in which we can add hardware to achieve parallelism is within the processor core itself. As we saw in Chapter 7, a processor repeatedly fetches, decodes and executes instructions. Many processor cores use various techniques to perform these steps in parallel. For example, a processor might fetch a new instruction while decoding the preceding instruction and executing the instruction before that. A higher performance processor might fetch several instructions at once, decode them together, and use multiple function units to execute as many of them in parallel as it can. These and other techniques for achieving

instruction-level parallelism are described in textbooks on computer architecture (see Section 9.5, Further Reading). While they can achieve performance improvements ranging from 2 times to perhaps 20 times over a simple processor core, the improvement comes at the cost of significantly increased complexity, area and power consumption. If an application requires much greater performance, or cannot afford the area and power consumption of a high-performance processor, a custom hardware accelerator may be a better option.

The extent to which we can improve performance depends on the amount of parallelism we can achieve, that is, on the number of steps we can perform at once. Many applications involve operations on data that has a regular, repetitive structure, and in which computation steps can be performed independently. For example, data from an audio source is a regular sequence of sample values. An operation that implements a volume control simply involves multiplying each sample value by the gain value. If several sample values are available at once, they can all be multiplied by the gain value in parallel. Similarly, video data from a camera consists of a sequence of frames, each of which is a rectangular array of picture elements (pixels). Many video processing operations can be performed within a frame in parallel across multiple pixels. Applications that involve less regularly structured data, or data that arrives at irregular intervals, are much harder to accelerate.

The amount of parallelism in some operations is limited only by the amount of data available at a given time. This applies to operations where each element of data can be processed independently of the others. Audio volume control is such a case. Other operations, however, involve *dependencies* that constrain parallelism. For example, some signal processing operations on audio streams involve combining successive sample values to produce values in a result stream. Filtering, as a case in point, involves combining several successive sample values to yield a single value in the output stream. Thus, we can't complete the processing for a given output sample until all of the required input values are available. Moreover, there are intermediate results that must be computed as part of the process, and the final result cannot be computed until all of the intermediate results have been computed.

In summary, we can accelerate performance of an operation by replication of hardware resources to perform steps in parallel, up to the limits on parallelism implied by the data dependencies and the availability of data. Practical design of accelerators involves applying enough parallelism to meet performance requirements, but not more, since that would increase cost and power unnecessarily.

In order to identify opportunities for parallelism, we would typically start with an abstract description of the processing operations to be performed by the system. This might take the form of an *algorithm* description expressed

in a high-level language, such as a computer programming language or some other formal notation. The description identifies the data to be processed, how it is organized, and the sequence of processing steps to be performed. We then need to identify a *kernel* of the algorithm, that is, a part that involves the most intensive repetitive processing steps that take the most time. Such a kernel is a good candidate for an accelerator, since improving performance of the most time-consuming part of the algorithm gives the most payback. The remainder of the algorithm can then be implemented in embedded software.

We can quantify the performance gain achieved by accelerating a kernel of an algorithm. Suppose a system takes some amount of time, t, to execute the algorithm, and that a fraction, f, of that time is spent in executing the kernel. The remaining fraction, $1-f$, is spent executing code other than the kernel. Thus,

$$t = ft + (1-f)t$$

If our accelerator speeds up execution of the kernel by a factor s, the time spent in the kernel is divided by s, but the remaining time is unaffected. Thus the total execution time for the algorithm is reduced to

$$t' = \frac{ft}{s} + (1-f)t$$

The overall speedup is the ratio of the original time to the reduced time:

$$s' = \frac{t}{t'} = \frac{ft + (1-f)t}{\frac{ft}{s} + (1-f)t} = \frac{1}{\frac{f}{s} + (1-f)}$$

This formula expresses Amdahl's Law, named after Gene Amdahl, one of the pioneers of parallel computing. It indicates that the overall effect of speeding up a kernel depends strongly on the fraction of the original time taken up in executing the kernel. If that fraction is small, even a large speedup has little overall effect, since the nonaccelerated part dominates. On the other hand, if the fraction is large, accelerating the kernel has significant overall effect.

EXAMPLE 9.1 Suppose execution time is estimated for the various parts of an algorithm on an embedded processor. The algorithm has two kernels, one that consumes 80% of the execution time and another that consumes 15%. Using a hardware accelerator, we could speed up execution of the first kernel by a factor of 10 or the second kernel by a factor of 100. Which accelerator gives the best overall performance improvement?

SOLUTION The overall speedup from accelerating the first kernel is

$$\frac{1}{\frac{0.8}{10}+(1-0.8)}=\frac{1}{0.08+0.2}=3.57$$

Accelerating the second kernel gives an overall speedup of

$$\frac{1}{\frac{0.15}{100}+(1-0.15)}=\frac{1}{0.0015+0.85}=1.17$$

Thus, even though the speedup for the second kernel is ten times that for the first kernel, the lower fraction of the original execution time for the second kernel means acceleration gives less overall improvement. Accelerating the first kernel is more effective.

Within the kernel, we need to identify an order in which to perform the computational steps. We need to ensure that data can be made available to be processed in order, and that intermediate results are computed before they are needed for subsequent steps. Other than those constraints, steps can potentially be performed in parallel. We finally need to determine which steps will actually be performed in parallel to meet the performance requirements. That then leads to an *architecture* for an accelerator, that is, a description of the processing blocks and the data flow between them.

There are two main schemes for implementing parallelism in accelerators. The first of these is simply to replicate components that perform a given step so that they operate on different elements of data. The speedup achieved through replication, compared to using just a single component, is ideally equal to the number of times the component is replicated. This scheme suits applications in which steps can be performed independently on the different data elements.

The second scheme for implementing parallelism is to break a larger computational step into a sequence of simpler steps, and to perform the sequence in a pipeline, as shown in Figure 9.1. (We introduced the concept of pipelining earlier in Section 4.1.1.) The pipeline stages perform their simple steps in parallel, each operating on a different data element or an intermediate result produced by the preceding stages. The overall computation by the pipeline for a given data element takes approximately the same time as a nonpipelined chain of components. However, provided we can supply data to the pipeline input and accept data at the pipeline output on every clock cycle, the pipeline completes one computation

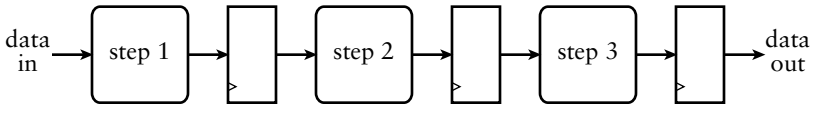


FIGURE 9.1 Pipelined organization of an accelerator.

every cycle. Thus, the speedup compared to the nonpipelined chain is ideally equal to the number of stages. This scheme suits applications that involve complex processing steps that can be broken down into simpler sequences with each step depending only on the results of earlier steps. In some applications involving independent complex computations, we can have replicated pipelines, giving the benefit of both schemes.

The analysis of systems, from algorithm description to accelerator architecture, is done early in the system design flow. It is often performed by expert system designers, drawing on their creativity and experience with previous systems. Automating this form of analysis has proven to be an extremely challenging problem, and early high-level synthesis tools have not been successful, except within very narrow application domains. More recently, a new generation of tools is starting to emerge and is showing promise in a wider range of applications, especially in audio, video and other signal-processing applications. As this technology matures, we should expect to see wider adoption in design methodologies. We will return to the topic of architecture analysis and its place in the design flow in our methodology discussion in Chapter 10.

The data for many systems involving accelerators is input or output data. In such systems, the I/O controller must transfer data between a device and the embedded system's memory at very high rates. Once the data is in memory, it can be processed by an accelerator, with the results also stored in memory. If these data memory accesses were mediated by a processor, copying data between memory and registers under software control, the rate of data transfer may be too slow. Instead, we can allow the controller and the accelerator to perform *direct memory access* (DMA), that is, to transfer data to and from memory autonomously. Instead of the processor initiating a memory access, the I/O controller or accelerator initiates an access, providing the required address and activating the memory control signals.

Since the processor and any subsystems that perform DMA must share access to the memory, and since the memory can only perform one access at a time, we need to ensure that processor and DMA accesses are interleaved. We must include an *arbiter* in the system, illustrated in Figure 9.2, that makes sure subsystems take turns to access the memory. Each *master* (the I/O controller, accelerator and processor) activates a request signal to the arbiter when it needs to access the memory. The arbiter decides among them, based on a predetermined policy, and activates a grant signal for one of the subsystems. That subsystem then proceeds with its access, with the memory responding as a *slave*. Any other master with an active request must wait. When the granted master has completed its memory access, it releases its request. The arbiter can then activate another master's grant. Different applications may use different policies for deciding among competing requests, depending on whether a

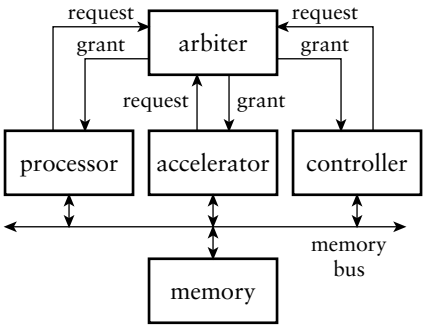


FIGURE 9.2 A multimaster system with an arbiter for the memory bus.

master can wait and for how long. Some applications use a round-robin policy, in which masters are granted access in strict turn. Other systems may require some masters to have priority over others in order to meet requirements for processing rates.

In many applications, the data to be processed by an accelerator is arranged in a regular pattern in memory, occupying *blocks* of adjacent or regularly spaced locations. The job of the accelerator is to process the data block by block. While it is processing one or more blocks, other parts of the system may be working on other blocks. As an example, several algorithms for processing still and video images divide each image into blocks of 8×8 or 16×16 pixels and process each block independently. Similarly, the MP3 format commonly used to encode audio data represents intervals of sound in frames that can be processed independently.

The datapath for a block-processing accelerator needs two main parts. The first part performs DMA to read and write data in memory. It includes circuits for generating addresses, using the starting addresses provided in registers by the processor and counters for keeping track of progress. The second part of the data path performs the required computation on the data. The control section for the accelerator sequences operation of the data path and synchronizes operation with the processor. Depending on the complexity of the operation and the bus protocol, sequencing might be done with one finite-state machine or with separate interacting machines for each activity.

Whereas a block-processing accelerator deals with blocks of data stored in continguous memory locations, other forms of accelerators deal with *streams* of data arriving in sequence from some source. Thus, the two forms of accelerator are complementary: block processing deals with sequences in space (data stored in memory), and stream processing deals with sequences in time (data arriving at intervals). The source of data for a stream-processing accelerator may be a high-speed input device or another accelerator in a processing pipeline. Alternatively, data may be fetched in a stream from memory for supply to an output device or another accelerator.

One of the most common application domains for stream-processing accelerators is *digital signal processing* (DSP). One or more signals are converted from analog to digital form, consisting of a stream of sample values at periodic intervals. Processing operations include filtering, mixing, applying gain or attenuation, and conversion between time and frequency domains. Some application areas include audio and video processing, radio and radar signal processing, and analysis of data from sensors. For details of the mathematical basis for digital signal processing and the computational techniques used, refer to Section 9.5, Further Reading.

Having provided a means for an accelerator to access data, either in memory or through a stream connection, we also need to provide a way for embedded software to control operation of the accelerator. This may include providing data, such as parameters to be used in computations. It also includes synchronizing operation of the accelerator with other activities in the system, such as arrival of data from I/O controllers or other I/O events. Generally, this is done using input and output registers within the accelerator. Embedded software can then interact with the accelerator in much the same way as it interacts with autonomous I/O controllers. For example, an accelerator might include registers for the address and length of data in memory, for control of the operation to be performed and for status. Embedded software could write to the registers to initiate an operation, and rely on an interrupt from the accelerator when the operation is complete.

In some applications, it may be possible for the processor and an accelerator to operate with less strict synchronization. For example, the processor might generate units of work for the accelerator to perform and add information describing each unit to a first-in, first-out (FIFO) queue, like that described in Section 5.2.3. The accelerator can then accept each work unit when it is ready by reading the description from the head of the FIFO queue. FIFO queues can also be used for communication between multiple processors in a large-scale embedded system.

KNOWLEDGE TEST QUIZ

1. How does parallelism improve performance?
2. What factors constrain the amount of parallelism that can be achieved?
3. What aspects are described by an algorithm?
4. Why is it best to accelerate a kernel of an algorithm?
5. If a pipeline has four stages and accepts new input data on every clock cycle, what is the speedup compared to a nonpipelined chain of components?
6. What is direct memory access (DMA)?

7. What is the task of an arbiter in a multimaster system?
8. What is the distinction between a block-processing accelerator and a stream-processing accelerator?
9. How does embedded software interact with an accelerator?

9.2 CASE STUDY: VIDEO EDGE-DETECTION

In this section, we will illustrate several aspects of accelerator design using, as an example, an accelerator for edge-detection in video images. This is somewhat of a compromise between what a real-world accelerator might do and what can be included here without overwhelming detail. Edge-detection is an important part of analyzing a scene in a video image, and has application in many areas such as security monitoring and computer vision. It involves identifying places in an image where there is an abrupt change in intensity. Those places usually occur at the boundaries of objects. Subsequent analysis of the edges can be used for recognizing what the objects are.

For this example, we will assume monochrome images of 640×480 pixels, each of 8 bits, stored row-by-row in memory with successive pixels, left to right in a row, at successive addresses. Pixel values are interpreted as unsigned integers ranging from 0 (black) to 255 (white). We will use a relatively simple algorithm, called the Sobel edge detector. It works by computing the derivatives of the intensity signal in each of the x and y directions and looking for maxima and minima in the derivatives. These are the places where the intensity is changing most rapidly. The Sobel method approximates the derivative in each direction for each pixel by a process called *convolution*. This involves adding the pixel and its eight nearest neighbors, each multiplied by a coefficient. The coefficients are often represented in a 3×3 *convolution mask*. The Sobel convolution masks, G_x and G_y, for the derivatives in the x and y directions, respectively, are shown in Figure 9.3. We can think of the derivative image being computed by centering each of the convolution masks over successive pixels in the original image. We multiply the coefficient in each mask by the intensity value of the underlying pixel and sum the nine products together to form two partial derivatives for the derivative image, D_x and D_y. Ideally, we would then compute the magnitude of the derivative image pixel as

$$|D| = \sqrt{D_x^2 + D_y^2}$$

However, since we are just interested in finding the maxima and minima in the magnitude, a sufficient approximation is

$$|D| = |D_x| + |D_y|$$

−1	0	+1
−2	0	+2
−1	0	+1

G_x

+1	+2	+1
0	0	0
−1	−2	−1

G_y

FIGURE 9.3 Sobel convolution masks.

This approximation works, because the square-root and square functions are both monotonic (that is, they increase as the operand increases and decrease as the operand decreases). Hence, the maxima and minima in the true magnitude and the approximate magnitude occur at the same places in the image. Computing the approximation involves much less hardware than computing the square and square-root functions. We repeat the computation of the approximate magnitude for each pixel position in the image. Note that the pixels around the edge of the image do not have a complete set of neighboring pixels, so we need to treat them separately. The simplest approach is to set the value of $|D|$ for the edge pixels of the derivative image to 0. Since that is a relatively straightforward process and is not time consuming, we can implement it in software.

EXAMPLE 9.2 Express the Sobel edge-detection algorithm more formally in a *pseudo-code* notation, that is, a notation like a computer programming language.

SOLUTION We will use a pseudo-code notation like Verilog. Let O[row] [col] denote pixels in the original image, and D[row] [col] denote pixels in the derivative image, where row ranges from 0 to 479 and col ranges from 0 to 639. Also, let Gx[i] [j] and Gy[i] [j] denote the convolution masks, where i and j range from –1 to +1. The algorithm is

```
for (row = 1; row <= 478; row = row + 1) begin
  for (col = 1; col <= 638; col = col + 1) begin
    sumx = 0; sumy = 0;
    for (i = -1; i <= +1; i = i + 1) begin
      for (j = -1; j <= +1; j = j + 1) begin
        sumx = sumx + O[row+i][col+j] * Gx[i][j];
        sumy = sumy + O[row+i][col+j] * Gy[i][j];
      end
    end
    D[row][col] = abs(sumx) + abs(sumy)
  end
end
```

EXAMPLE 9.3 Calculate the number of bits required to represent intermediate and final values for pixels in the Sobel convolution.

SOLUTION Each pixel is represented as an 8-bit unsigned number. Given the coefficient values in the convolution masks, the partial products range from –510 to +510. Thus, the partial products should be represented using 10-bit signed numbers. There are nine partial products to add to form each of D_x and

D_y. However, the coefficient values are such that the result values range from –1020 to +1020, which can be represented using 11 bits. We then need to add the two absolute values, giving a range of 0 to +2040 for |*D*|, which can also be represented in 11 bits. Since subsequent steps of the edge-detection operation involve determining which derivative pixels are above a certain threshold, we don't need to maintain 11 bits of accuracy for the results. Instead, it is more convenient to scale the results back to 8-bit values, since they can be packed back into memory in the same format as the original image.

EXAMPLE 9.4 Assuming a video frame rate of 30 frames per second, calculate the rate at which computations must be performed.

SOLUTION Each frame consists of $640 \times 480 = 307{,}200$ pixels. Since there are 30 frames per second, pixels must be processed at a rate of $307{,}200 \times 30 = 9{,}216{,}000$ per second, that is, approximately 10 million per second.

EXAMPLE 9.5 Identify the parallelism that can be exploited to obtain the required performance.

SOLUTION The computations required for all of the derivative pixels are independent of one another, since they only require values of the original image pixels. Thus, we could perform computations for as many derivative pixels in parallel as required. For computation of each derivative pixel, the *data dependency graph* is shown in Figure 9.4. This diagram shows the data required for each operation, starting with the pixels from the original image at the top, with intermediate results feeding through to dependent operations, yielding the derivative pixel at the bottom. We've elided partial products in which the coefficient is 0, since they don't contribute to the result. Inspection of the diagram shows that we can compute all of the partial products in parallel, since each partial product depends only on an original pixel value and a constant coefficient. We

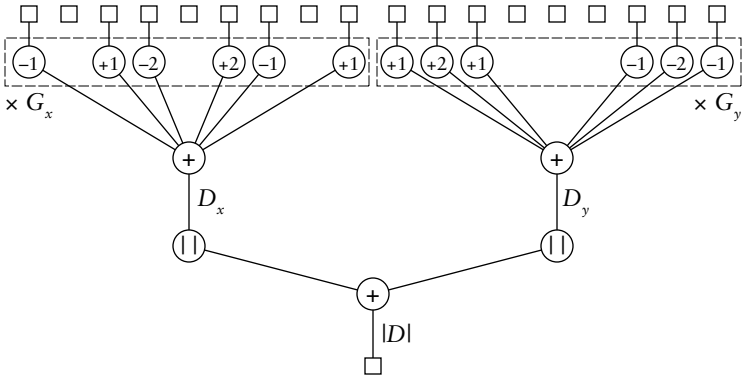


FIGURE 9.4 Data dependency graph for computation of a derivative pixel.

can then sum the two groups of six partial products in parallel, then compute the two absolute values in parallel, before summing them to produce the derivative pixel value.

The top-level view of the video system including the edge-detection accelerator is shown in Figure 9.5. Video input comes from an I/O controller for a video camera, which stores successive video frames in memory. Software on the processor directs the accelerator to operate on a given frame to produce the corresponding derivative image.

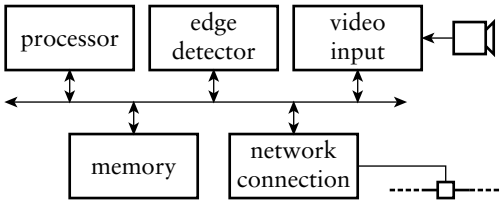


FIGURE 9.5 A video system incorporating an accelerator for edge detection.

EXAMPLE 9.6 Suppose the memory in which the original and derivative images are stored is 32 bits wide, and that each 8-bit byte is individually addressed. Video frames are stored with one byte per pixel. The pixels of a row in a frame are stored from left to right at successive addresses, and rows are stored top to bottom, one after another in memory. Each memory read or write access takes 20ns, consisting of two cycles of a 100MHz system clock. Can the memory access data fast enough?

SOLUTION Our earlier analysis showed that pixels arrive from the camera at a rate of approximately 10 million per second, or one every 100ns. If the video input controller stored each pixel to memory with a separate write access, it would consume 20% of the available memory bandwidth. A better alternative would be for the controller to aggregate four pixels and store them with a single write access, reducing its share of the memory bandwidth to 5%.

The edge-detection accelerator needs to produce a derivative pixel at the same rate at which input pixels arrive, that is, one every 100ns. Thus, writing the computed derivative pixels would consume a further 5% of the memory bandwidth, assuming groups of four derivative pixels are aggregated. Each pixel computation requires access to eight pixel values from the original image. A naive approach would involve reading each pixel with a separate read operation, and re-reading it when subsequently required to compute another derivative pixel. This approach would require eight reads per computed pixel, requiring 160% of the memory bandwidth. Clearly this is not possible.

Since each 32-bit word of memory contains four adjacent pixels in a row, we can reduce the bandwidth required for reading by using as many pixels as we can from each 32-bit read. For half the pixel positions, only three reads are needed

(when the three pixels in each of the rows fall in the same word), and for the other half of the pixel positions, six reads are needed (when the three pixels in each of the rows cross word boundaries). So on average, each pixel computation would require 4.5 reads, requiring 90% of the memory bandwidth. This is still not feasible.

A further reduction can be afforded by noting that an original image pixel, once read, is used to compute three derivative pixels in each of the following, same, and preceding columns. So rather than re-reading it for those pixels, we can store it within the accelerator for use in computing multiple derivative pixels. We can save it just for computing the pixels to the left, in the same column, and to the right. We only need to read three words for every fourth pixel being computed, requiring 15% of the memory bandwidth. This, together with the 5% for video input and 5% for writing derivative pixels, is feasible, provided the remaining 75% of the bandwidth is sufficient for other operations to be performed by the system.

If we need to further reduce the bandwidth consumed by the edge detector, we could include small memories in the accelerator to store complete rows read from the main memory. This would allow each pixel to be read only once, reducing the bandwidth required for reading pixels to just 5%. The total for video input and edge-detection would then be 15% of the available bandwidth.

In our development of the edge-detector example, we will adopt the approach of reading three rows of four adjacent pixels from the original image and storing them in registers, rather than including memories for whole rows. We will design the accelerator to process blocks of data, where a block consists of the three complete rows of the original image used to form a complete row of the derivative image. As we will see, processing a block involves a start-up phase, a repetitive sequence of computation, and a completion phase. These phases are repeated for each derivative image row.

The architecture for the Sobel accelerator datapath is shown in Figure 9.6. It is essentially a pipeline, with pixel data read from the original image entering into the registers at the top right, flowing through the 3×3 multiplier array on the left, then down through the adders to the Dx and Dy registers, then through the absolute value circuits and adder to the |D| register, and finally into the register at the bottom left. The resulting derivative pixels are then written from that register to memory. (While a right-to-left data flow is opposite to usual practice, in this case, it has the advantage of preserving the same arrangement of pixels as that in an image.) We will describe the operation of the pipeline assuming initially that it is full of data. We will then discuss how to deal with starting it up at the beginning of an image row and draining it at the end of the row.

The pipeline generates the derivative pixels for a given row in groups of four. The accelerator reads four pixels from each of the preceding,

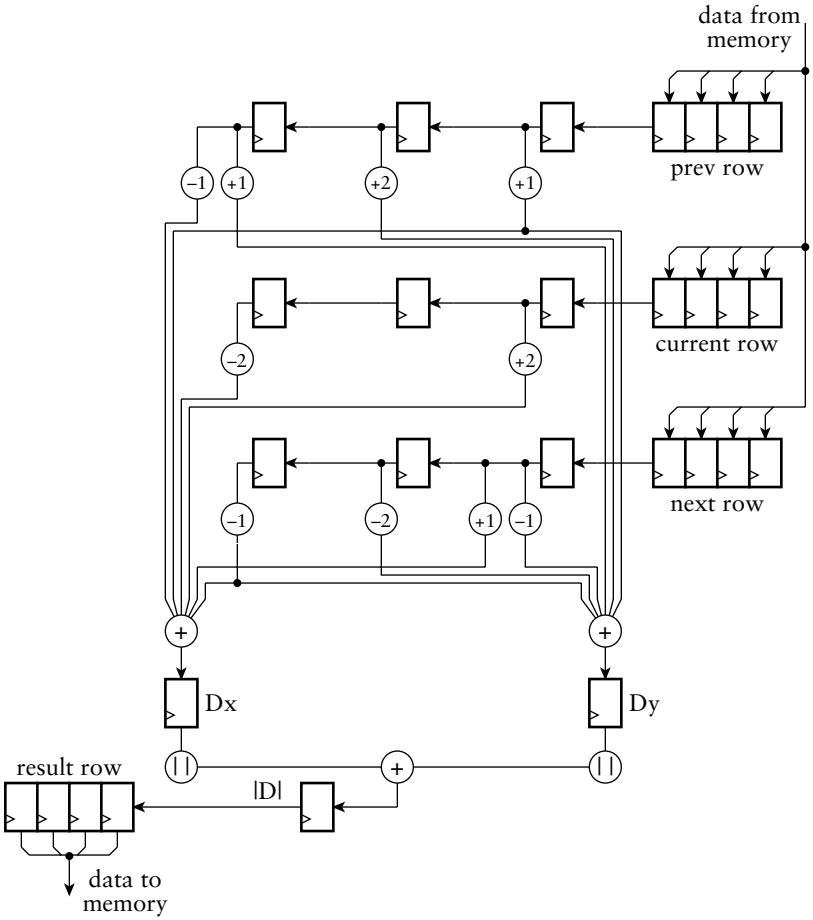


FIGURE 9.6 Architecture for the Sobel accelerator datapath.

current, and next rows in memory into the three 32-bit registers at the top right of the figure. Each register consist of four 8-bit pixel registers. Over the four subsequent clock cycles, pixels are shifted out to the left, one pixel at a time, into the multiplier array. Each cell in the array contains a pixel register and one or two circuits that multiply the stored pixel by a constant coefficient value. Since the coefficients are all -1, $+1$, -2, or $+2$, the circuits are not full-blown multipliers. Instead, multiplying by -1 is simply a negator, multiplying by $+1$ is a through connection with no circuitry, multiplying by -2 is a left shift of the result of a negator, and multiplying by $+2$ is simply a left shift. On each clock cycle, the array provides the partial products for a single derivative pixel, and the partial products are added and stored in the Dx and Dy registers. Also, on each clock cycle, the Dx and Dy values for the preceding pixel have their absolute values computed and added and stored in the |D| register. The

resulting derivative pixel values are shifted into the result row register. When four result pixels are ready in the register, they are subsequently written to memory.

In the steady state, during processing of a row, the accelerator needs to write the pixels to memory from the result register before it can shift new pixels into the multiplier array and the Dx, Dy and |D| registers. Otherwise, the result values would be overwritten. Having written four pixels, the accelerator can push four more pixels through the pipeline, thus emptying the read registers and filling the result register. It can then write those result pixels and read in three more groups of four pixels, and repeat the process. This sequence is shown in Figure 9.7, assuming a Wishbone bus connection with 32-bit-wide data signals and a 100MHz clock, as suggested earlier. Since the accelerator is one of several masters on the memory bus, it must request use of the bus for the writes and reads and wait until granted access by the bus arbiter. We assume that the arbiter gives the accelerator sufficiently high priority that it can use the memory bandwidth it needs.

Now that we have considered the steady state during processing of a row, we need to consider what happens at the beginning of a row. In that case, the registers in the pipeline contain no valid data. So we start processing a row as in the steady state, but omitting the write operation for the first two iterations. Thereafter, the result register contains valid data, so we include the write operation in each iteration. Note that after the first four computation cycles, valid data has progressed into the pipeline as far as the Dx and Dy registers. After the second four computation cycles, valid data has progressed as far as the right-most three result pixel registers. The left-most result pixel register still contains invalid data.

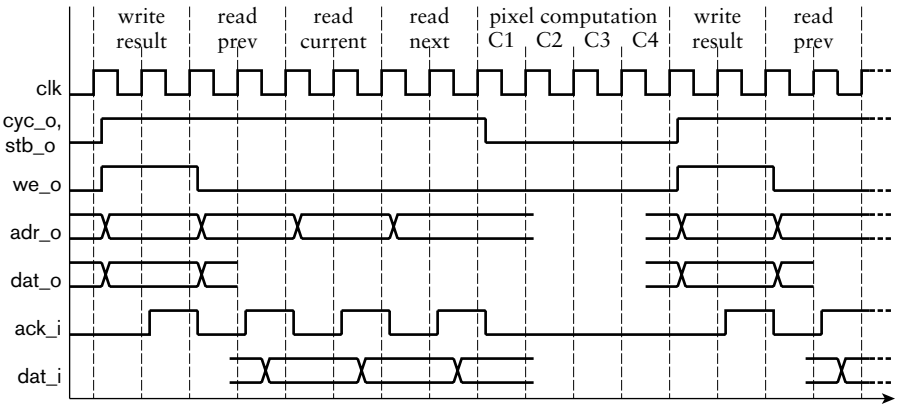


FIGURE 9.7 Timing of pixel write and read operations and computation in the pipeline.

However, this group of four pixel values is what we should write to the beginning of the derivative image row. As we mentioned earlier, the left-most position does not have a complete set of neighbors, so we don't compute a value for it. We will rely on the embedded software to clear that pixel value to 0 subsequently.

When we reach the end of a row, we need to drain the pipeline. Since the number of pixels in a row is a multiple of four ($640 = 160 \times 4$), we can always read complete groups of four pixels each. After reading the last group, we perform four computation cycles normally. This gives us four result pixels to write, plus three remaining pixel values in the pipeline. We finish the row by writing the four result pixels, omitting the reads, performing four further computation cycles to drain the pipeline and shift the last pixel values into the required positions in the result register, and performing a final write. Note that this places an invalid value in the right-most result pixel register. This corresponds to the right-most pixel of a row, which does not have a complete set of neighbors. Again, we will rely on the embedded software to clear that pixel value to 0.

EXAMPLE 9.7 Develop Verilog RTL code to describe the datapath of Figure 9.6.

SOLUTION The code in the module definition for the Sobel accelerator is

```
// Computation datapath signals

reg        [31:0] prev_row, curr_row, next_row;
reg        [7:0]  O [-1:+1][-1:+1];
reg signed [10:0] Dx, Dy, D;
reg        [7:0]  abs_D;
reg        [31:0] result_row;
...

// Computational datapath

always @(posedge clk_i) // Previous  row register
  if (prev_row_load) prev_row       <= dat_i;
  else if (shift_en) prev_row[31:8] <= prev_row[23:0];

always @(posedge clk_i) // Current  row register
  if (curr_row_load) curr_row       <= dat_i;
  else if (shift_en) curr_row[31:8] <= curr_row[23:0];

always @(posedge clk_i) // Next row  register
  if (next_row_load) next_row       <= dat_i;
```

(continued)

```
  else if (shift_en) next_row[31:8] <= next_row[23:0];

function [10:0] abs (input signed [10:0] x);
  abs = x >= 0 ? x : -x;
endfunction

always @(posedge clk_i) // Computation pipeline
  if (shift_en) begin
    D = abs(Dx) + abs(Dy);
    abs_D <= D[10:3];
    Dx <= -  $signed({3'b000, O[-1][-1]})  // - 1 * O[-1][-1]
          +  $signed({3'b000, O[-1][+1]})  // + 1 * O[-1][+1]
          - ($signed({3'b000, O[ 0][-1]})  // - 2 * O[ 0][-1]
                << 1)
          + ($signed({3'b000, O[ 0][+1]})  // + 2 * O[ 0][+1]
                << 1)
          -  $signed({3'b000, O[+1][-1]})  // - 1 * O[+1][-1]
          +  $signed({3'b000, O[+1][+1]}); // + 1 * O[+1][+1]
    Dy <=    $signed({3'b000, O[-1][-1]})  // + 1 * O[-1][-1]
          + ($signed({3'b000, O[-1][ 0]})  // + 2 * O[-1][ 0]
                << 1)
          +  $signed({3'b000, O[-1][+1]})  // + 1 * O[-1][+1]
          -  $signed({3'b000, O[+1][-1]})  // - 1 * O[+1][-1]
          - ($signed({3'b000, O[+1][ 0]})  // - 2 * O[+1][ 0]
                << 1)
          -  $signed({3'b000, O[+1][+1]});  // - 1 * O[+1][+1]
    O[-1][-1] <= O[-1][0];
    O[-1][ 0] <= O[-1][+1];
    O[-1][+1] <= prev_row[31:24];
    O[ 0][-1] <= O[0][0];
    O[ 0][ 0] <= O[0][+1];
    O[ 0][+1] <= curr_row[31:24];
    O[+1][-1] <= O[+1][ 0];
    O[+1][ 0] <= O[+1][+1];
    O[+1][+1] <= next_row[31:24];
  end

always @(posedge clk_i)  // Result row register
  if (shift_en) result_row <= {result_row[23:0], abs_D};
```

The first three always blocks in the module represent the three registers into which groups of four pixels are read from memory. Each block has a separate control signal governing loading, since the registers are loaded in successive memory read operations. They share a control signal for shifting, since they all shift a pixel out into the pipeline in parallel.

The next always block, as the comment suggests, represents the computational pipeline of the accelerator. The signals to which the block assigns, governed by

the shift_en control signal, represent the pipeline registers. The signal O is a 3×3 array of pixel values, with indices corresponding to the difference in row and column numbers from those of the derivative pixel computed from the register values. For example, the element with indices $[-1][+1]$ contains the pixel in the previous row and next column from the pixel being computed. Values are shifted into this array leftward from the left-most 8 bits of each of the input registers. The Dx and Dy values are computed from the array element values. In each case, the values are resized to 11 bits and converted to signed numbers, as we discussed earlier in our analysis of the precision requirements for the computation. Multiplying by 2 is performed with a logical shift left by one position, and multiplying by a negative coefficient is implemented by subtraction instead of addition. The absolute values of the Dx and Dy values, implemented by the abs function defined in the module, are added, and then scaled back from 11 to 8 bits to yield the final derivative pixel value.

The remaining always block represents the register that accumulates groups of four derivative pixels for writing to memory. Pixels are shifted into this register under control of the shift_en signal.

We mentioned earlier that a block-processing accelerator needs circuits for address generation, as well as for processing the data. Our Sobel accelerator needs circuits to compute the addresses for reading pixels from the original image and for writing pixels to the derivative image. We will provide a register into which the embedded software can write the base addresses for the original image and the derivative image in memory. The address generator needs to determine pixel addresses using the base addresses. We will require that all addresses are word aligned, that is, that they are all multiples of four. This means the two least significant address bits are always 00, and so do not need to be computed or explicitly stored.

EXAMPLE 9.8 Given a base address B for an image in memory, derive equations for computing the address of a pixel in row r and column c of the image. Rows and columns are numbered from 0.

SOLUTION The image size is 480 rows of 640 pixels per row. The starting address of row r is

$$B + r \times 640$$

The pixel in column c in that row is then located at address

$$B + r \times 640 + c$$

We can treat the expression $r \times 640 + c$ as an address offset from the base address.

EXAMPLE 9.9 Design the address generator datapath for the Sobel accelerator. Assume main memory is 4Mbytes in size, organized as 1M × 32 bits.

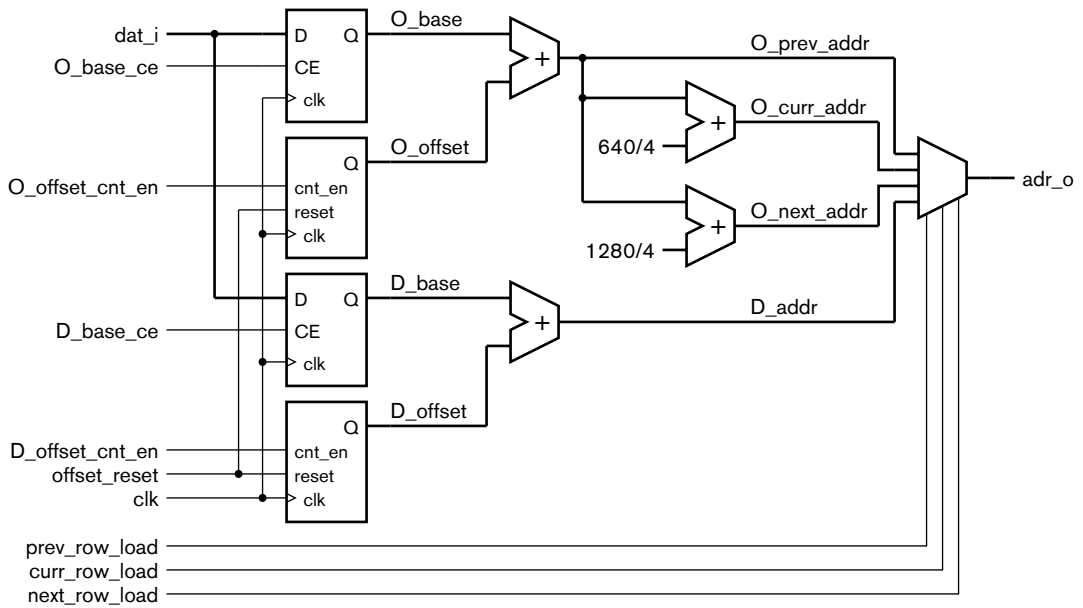
SOLUTION The address generator needs two base address registers: `O_base`, for the original image, and `D_base`, for the derivative image. Since pixels are processed in groups of four, the least significant two address bits are always 0, and so do not need to be explicitly stored in the address registers.

There are several alternatives for deriving the read and write addresses, including maintaining counters for the image rows and columns. However, we can avoid the need to multiply by 640 by counting pixel offsets from the base addresses, as shown in Figure 9.8. In the case of the original image, we start counting from an offset of 0 and increment by 1 for each group of four pixels read from memory. We add the offset to the base address to form the pixel-group address for the previous row. We add 640/4 to that to form the read address for the current row, and add 1280/4 to form the read address for the next row (assuming 00 for the least significant bits in both cases). In the case of the derivative image, we start counting from an offset of 640/4 and increment by 1 for each memory write. The multiplexer in the figure selects the appropriate computed address to drive the memory address bus.

EXAMPLE 9.10 Develop Verilog RTL code to describe the address generator of Figure 9.8.

SOLUTION The code in the module definition for the Sobel accelerator is

FIGURE 9.8 Datapath for the address generator.



```
// Address generator

always @(posedge clk_i)  // O base address register
  if (O_base_ce) O_base <= dat_i[21:2];

always @(posedge clk_i)  // O address offset counter
  if (offset_reset)         O_offset <= 0;
  else if (O_offset_cnt_en) O_offset <= O_offset + 1;

assign O_prev_addr = O_base + O_offset;
assign O_curr_addr = O_prev_addr + 640/4;
assign O_next_addr = O_prev_addr + 1280/4;

always @(posedge clk_i)  // D base address register
  if (D_base_ce) D_base <= dat_i[21:2];

always @(posedge clk_i)  // D address offset counter
  if (offset_reset)         D_offset <= 0;
  else if (D_offset_cnt_en) D_offset <= D_offset + 1;

assign D_addr = D_base + D_offset;

assign adr_o[21:2] = prev_row_load ? O_prev_addr :
                     curr_row_load ? O_curr_addr :
                     next_row_load ? O_next_addr :
                     D_addr;
assign adr_o[1:0]  = 2'b00;
```

The always blocks commented as being base address registers represent the base address registers for the original and derivative images, respectively. The always blocks commented as being address offset counters represent the counters for pixel groups read and written, respectively. The registers and counters are governed by control signals generated by the accelerator's control section. The adders are represented by the combinational assignments to the four address signals O_prev_addr, O_curr_addr, O_next_addr and D_addr. The assignment to the bus address signal adr_o represents the multiplexer that chooses among the generated addresses for memory read and write operations.

The remaining aspect of the Sobel accelerator design is control sequencing. We have touched on the sequence needed for computation of the derivative image, row-by-row and pixel-group at a time. This includes sequencing of write and read operations with the accelerator as a bus master. We also need to sequence the accelerator's response as a bus slave

when the embedded software writes to the base address registers. Finally, we need to provide for synchronization with the embedded software controlling the accelerator. That requires some additional control and status registers, as follows:

- A control register that, when written to, causes the accelerator to start processing an image. The value written is ignored.
- A control register with an interrupt enable bit in bit 0.
- A status register in which bit 0 is the done bit, set to 1 when the processor has completed processing an image. Other bits are read as 0. When the done bit is 1 and the interrupt enable bit is 1, the accelerator requests an interrupt. Reading the done bit has the side effect of acknowledging the interrupt and clearing the bit.

To keep the bus interface simple, we will map each of these registers at 32-bit aligned addresses. The complete register map is shown in Table 9.1.

REGISTER	OFFSET	READ/WRITE
Interrupt control	0	write-only
Start	4	write-only
Original image base address	8	write-only
Derivative image base address	12	write-only
Status	0	read-only

TABLE 9.1 Register map for the Sobel accelerator.

EXAMPLE 9.11 Develop Verilog model code for the accelerator's bus slave interface.

SOLUTION The timing for the bus slave operations is shown in Figure 9.9. Both write and read operations are initiated in a cycle where cyc_i and stb_i are 1.

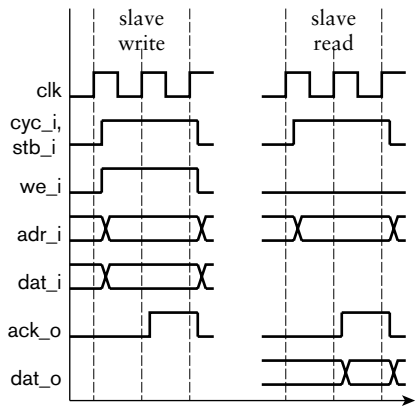


FIGURE 9.9 Timing for slave bus write and read operations.

In each case, the accelerator can respond by setting ack_o to 1 in the next cycle, then back to 0 in the following cycle. We need to decode the bus address input to derive a select signal for the accelerator, and use the less significant address bits to determine which register to read or write. For write operations, we generate clock-enable signals using combinational logic. In the case of a write to the start-register address, since there is no real register, we derive a control signal, start, that will be used by the accelerator control section to initiate a computation sequence. For read operations, we form the data value to be returned to the processor. The only real register is the status register, for which we return the value of the done bit, zero extended to 32 bits wide. For other register offsets, we just return all zeros. The read value is multiplexed with the value of the result row register to drive the accelerator's data output bus, dat_o. The model code describing these aspects is

```
// Wishbone slave interface

assign start     = cyc_i && stb_i && we_i && adr_i == 2'b01;

assign O_base_ce = cyc_i && stb_i && we_i && adr_i == 2'b10;

assign D_base_ce = cyc_i && stb_i && we_i && adr_i == 2'b11;

always @(posedge clk_i)  // Interrupt enable register
  if (rst_i)
    int_en <= 1'b0;
  else if (cyc_i && stb_i && we_i && adr_i == 2'b00)
    int_en <= dat_i[0];

always @(posedge clk_i)  // Status register
  if (rst_i)
    done <= 1'b0;
  else if (done_set)
    // This occurs when last write is acknowledged,
    // and so cannot coincide with a read of the
    // status register.
    done <= 1'b1;
  else if (cyc_i && stb_i && we_i && adr_i == 2'b00 && ack_o)
    done <= 1'b0;

assign int_req = int_en && done;

always @(posedge clk_i)  // Generate ack output
  ack_o <= cyc_i && stb_i && !ack_o;
```

(continued)

```
 // Wishbone data output multiplexer

always @*
  if (cyc_i && stb_i && !we_i)
    if (adr_i == 2'b00)
      dat_o = {31'b0, done};    // status register read
    else
      dat_o = 32'b0;            // other registers read as 0
  else
      dat_o = result_row;       // for master write
```

EXAMPLE 9.12 Develop the control section to sequence computation of the derivative image.

SOLUTION We can use a finite-state machine to sequence the computation. Since much of the sequence is repetitive, we can use counters to keep track of progress. We will use one counter to keep track of how many rows have been computed, starting from 0 and incrementing up to 477. We will use a second counter to keep track of iterations across the columns, starting from 0 and incrementing up to 159. The state transition diagram for the FSM is shown in Figure 9.10. We have only shown the states and the transition conditions to avoid cluttering the diagram. Also, we have not shown transitions from a state back to itself. We assume that if a transition condition from a given state is false, the FSM stays in that state for the next cycle.

The FSM is initially in the idle state. When the start signal is activated by a write to the start register, the FSM starts the initial sequence of reads and computations for the first row. This consists of reading the first three groups of original image pixels and then performing four computation cycles. After that, the FSM enters a loop in which it reads three more groups of original image pixels, performs four computation cycles, and then writes a group of result pixels. As we will see when we look at the output function of the FSM, the column counter is incremented after each write. At the end of the last computation cycle, the FSM either continues with the loop (if the column counter is not 158) or goes to a state to start draining the pipeline (if the column counter is 158). Draining the pipeline involves one state for writing the penultimate result group, four cycles of computation, and one last state for writing the final result group. The row counter is incremented after this final write. The FSM then goes back either to the initial sequence for the next row (if the row counter is not 477) or to the idle state (if the row counter is 477, the terminal count).

The output functions for the FSM are shown in Tables 9.2 and 9.3. To make the tables a little easier to read, we have left entries blank where the control outputs are 0, and only shown the cases where they are 1. Some of the control signals are Moore outputs, depending on the current state only. They are shown

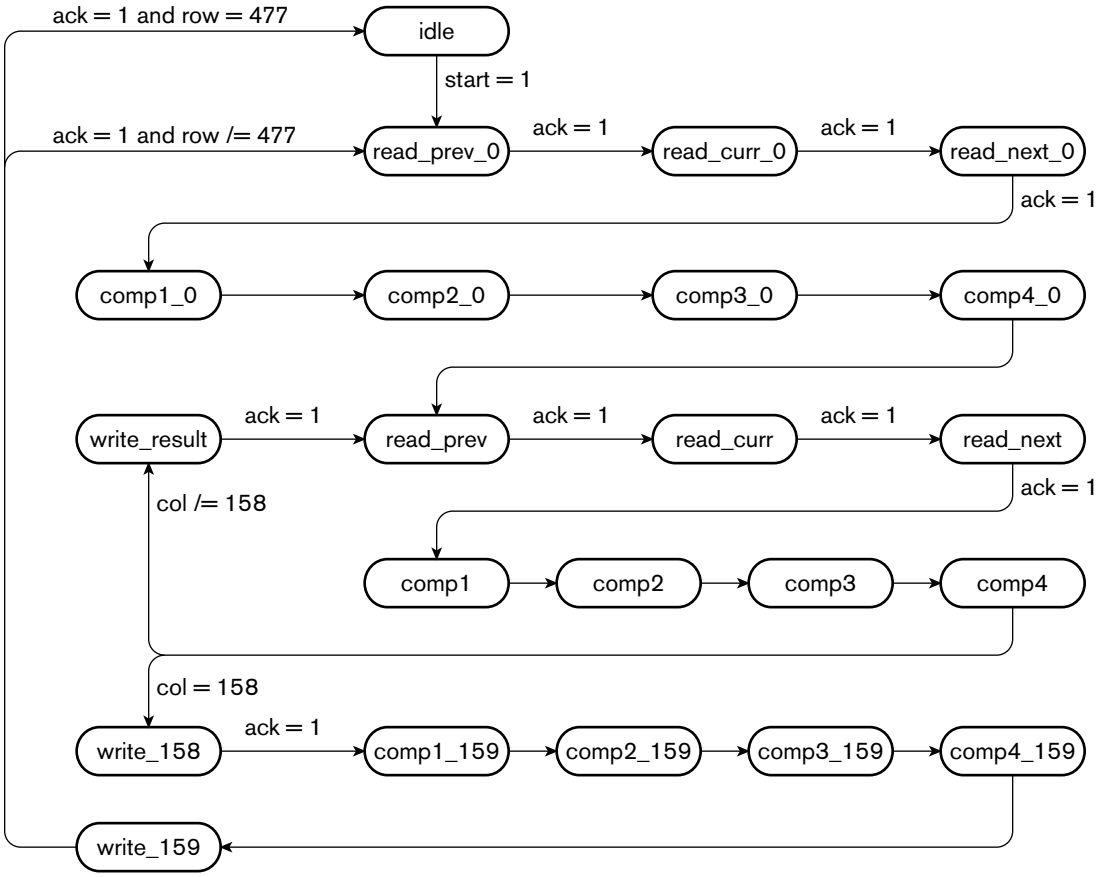


FIGURE 9.10 State transition diagram for the Sobel accelerator control section.

in Table 9.2. Other control signals are Mealy outputs. For these, in Table 9.3, we have shown the input conditions that, along with the current state, determine their values. As in the state transition diagram, we have omitted the complementary conditions. In those cases, the Mealy outputs remain 0.

EXAMPLE 9.13 Develop Verilog model code for the control section.

SOLUTION The control-section code includes declarations of internal signals for the control FSM, the row and column counters, and the control signals:

```
parameter [4:0]  idle         = 5'b00000,
                 read_prev_0  = 5'b00001,
                 read_curr_0  = 5'b00010,
                 read_next_0  = 5'b00011,
                 comp1_0      = 5'b00100,
```

(continued)

TABLE 9.2 Output functions for the Moore control outputs of the FSM.

CURRENT STATE	offset_reset	row_reset	col_reset	prev_row_load	curr_row_load	next_row_load	shift_en	cyc_o	we_o
idle	1	1	1						
read_prev_0			1	1				1	
read_curr_0					1			1	
read_next_0						1		1	
comp1_0							1		
comp2_0							1		
comp3_0							1		
comp4_0							1		
read_prev				1				1	
read_curr					1			1	
read_next						1		1	
comp1							1		
comp2							1		
comp3							1		
comp4							1		
write_result								1	1
write_158								1	1
comp1_159							1		
comp2_159							1		
comp3_159							1		
comp4_159							1		
write_159								1	1

CURRENT STATE	CONDITION	row_cnt_en	col_cnt_en	0_offset_cnt_en	D_offset_cnt_en	done_set
idle	start = 1					
read_prev_0	ack_i = 1					
read_curr_0	ack_i = 1					
read_next_0	ack_i = 1			1		
comp1_0	–					
comp2_0	–					
comp3_0	–					
comp4_0	–					
read_prev	ack_i = 1					
read_curr	ack_i = 1					
read_next	ack_i = 1			1		
comp1	–					
comp2	–					
comp3	–					
comp4	col /= 158					
comp4	col = 158					
write_result	ack_i = 1		1		1	
write_158	ack_i = 1		1		1	
comp1_159	–					
comp2_159	–					
comp3_159	–					
comp4_159	–					
write_159	ack_i = 1 and row /= 477	1			1	
write_159	ack_i = 1 and row = 477				1	1

TABLE 9.3 Output functions for the Mealy control outputs of the FSM.

```
                comp2_0      = 5'b00101,
                comp3_0      = 5'b00110,
                comp4_0      = 5'b00111,
                read_prev    = 5'b01000,
                read_curr    = 5'b01001,
                read_next    = 5'b01010,
                comp1        = 5'b01011,
                comp2        = 5'b01100,
                comp3        = 5'b01101,
                comp4        = 5'b01110,
                write_result = 5'b01111,
                write_158    = 5'b10000,
                comp1_159    = 5'b10001,
                comp2_159    = 5'b10010,
                comp3_159    = 5'b10011,
                comp4_159    = 5'b10100,
                write_159    = 5'b10101;
reg [4:0] current_state, next_state;
reg [9:0] row; // range 0 to 477;
reg [7:0] col; // range 0 to 159;

wire O_base_ce, D_base_ce;

wire start;
reg  offset_reset, row_reset, col_reset;

reg prev_row_load, curr_row_load, next_row_load;
reg shift_en;
reg row_cnt_en, col_cnt_en;
reg O_offset_cnt_en, D_offset_cnt_en;
reg int_en, done_set, done;
```

The two counters used by the control section to keep track of progress through rows and columns, respectively, are represented by the following always blocks:

```
always @(posedge clk_i)    // Row counter
  if (row_reset)       row <= 0;
  else if (row_cnt_en) row <= row + 1;

always @(posedge clk_i) // Column counter
  if (col_reset)     col   <= 0;
  else if (col_cnt_en) col   <= col + 1;
```

Next, the model includes blocks representing the finite-state machine using the techniques we have described in previous chapters. The state register is represented by the block:

```
always @(posedge clk_i)    // State register
  if (rst_i) current_state   <= idle;
  else       current_state     <= next_state;
```

A final always block combines both the state transition function and the output function into the one block. The block also includes expressions comparing the row and column counter values with their terminal count values, rather than performing the comparisons in separate combinational statements. Combining these aspects into a single block makes the Verilog model somewhat more compact and simpler to understand, since the FSM is somewhat larger than those we have previously described.

```
always @* begin     // FSM logic
  offset_reset      = 1'b0; row_reset          = 1'b0;
  col_reset         = 1'b0;
  row_cnt_en        = 1'b0; col_cnt_en         = 1'b0;
  0_offset_cnt_en   = 1'b0; D_offset_cnt_en    = 1'b0;
  prev_row_load     = 1'b0; curr_row_load      = 1'b0;
  next_row_load     = 1'b0;
  shift_en          = 1'b0; cyc_o              = 1'b0;
  we_o              = 1'b0; done_set           = 1'b0;
  case (current_state)
    idle: begin
        offset_reset = 1'b1; row_reset = 1'b1;
        col_reset = 1'b1;
        if (start) next_state = read_prev_0;
        else       next_state = idle;
      end
    read_prev_0: begin
        col_reset = 1'b1; prev_row_load = 1'b1;
        cyc_o  = 1'b1;
        if (ack_i) next_state = read_curr_0;
        else       next_state = read_prev_0;
      end
    read_curr_0: begin
        curr_row_load = 1'b1; cyc_o = 1'b1;
        if (ack_i) next_state = read_next_0;
        else       next_state = read_curr_0;
      end
    read_next_0: begin
        next_row_load = 1'b1; cyc_o = 1'b1;
        if (ack_i) begin
          0_offset_cnt_en = 1'b1;
          next_state = comp1_0;
        end
        else next_state = read_next_0;
      end
    comp1_0: begin
        shift_en = 1'b1;
        next_state = comp2_0;
      end
    ...
    comp4: begin
        shift_en = 1'b1;
```

(continued)

```
        if (col == 158) next_state = write_158;
        else            next_state = write_result;
      end
    write_result: begin
        cyc_o = 1'b1; we_o = 1'b1;
        if (ack_i) begin
          col_cnt_en = 1'b1; D_offset_cnt_en = 1'b1;
          next_state = read_prev;
        end
        else next_state = write_result;
      end
    write_158: begin
        cyc_o = 1'b1; we_o = 1'b1;
        if (ack_i) begin
          col_cnt_en = 1'b1; D_offset_cnt_en = 1'b1;
          next_state = comp1_159;
        end
        else next_state = write_158;
      end
    ...
    write_159: begin
        cyc_o = 1'b1; we_o = 1'b1;
        if (ack_i) begin
          D_offset_cnt_en = 1'b1;
          if (row == 477) begin
            done_set = 1'b1;
            next_state = idle;
          end
          else begin
            row_cnt_en = 1'b1;
            next_state = read_prev_0;
          end
        end
        else next_state = write_159;
      end
  endcase
end

assign stb_o = cyc_o;
```

Now that we have developed all of the hardware required for the Sobel accelerator, the remaining part is the embedded software that controls its operation. As we mentioned when we introduced this example, video edge-detection is used in a range of application areas. So rather than redesigning the control software for each application, it makes better sense to develop a software component that can be reused from one application to another. We can do this by developing a driver that provides a set of operations that gives application software an abstract view of the

accelerator. Each application can then use the driver as one part of a collection of software components that implements the required functionality. For example, an application that recognizes objects in video images might apply edge-detection to each image in a video stream, followed by grouping of edges and matching against a database of edge patterns. Such software development is just as important as the hardware development in a complete application. A more complete treatment can be found in books on embedded system software development (see Section 9.5, Further Reading).

KNOWLEDGE TEST QUIZ

1. If image pixels were represented using only 6 bits instead of 8, how many bits would be required for the values of Dx, Dy and |D|?
2. Can the value of |D| for a given derivative-image pixel be computed in parallel with the values of Dx and Dy? Why, or why not?
3. If the memory read and write time is increased from two cycles to four, would there be sufficient memory bandwidth for video input and edge-detection?
4. Why do we not compute values for the left-most and right-most pixels in each row of the derivative image?
5. How does the embedded software initiate processing of an image? How does it determine when processing is complete?
6. What would happen if the software attempted to initiate processing when processing of a previous image was not yet complete?
7. Is the FSM that sequences computation a Mealy, Moore, or hybrid FSM?

9.3 VERIFYING AN ACCELERATOR

Throughout this book, we have stressed the importance of verification as part of our design methodology. It is particularly important when designing accelerators, given their relative complexity. We need to ensure that the design will operate correctly with all legal data values, and that it will interact with the embedded processor correctly. Since the space of all possible data values and operational sequences is astronomically large, it is not feasible to test the design exhaustively. Rather, we need to develop a verification plan that covers a variety of operating conditions. We will return to this in more detail in our methodology discussion in Chapter 10. Meanwhile, we will illustrate a simpler approach to simulation-based verification of the Sobel accelerator described in Section 9.2.

One way to approach verification of a complex accelerator is to verify the different aspects of its operation independently. For example, we might

verify the following aspects of the Sobel accelerator one by one, adopting a "divide and conquer" approach:

- Slave bus operations
- Computation sequencing
- Master bus operations
- Address generation
- Pixel computation

Clearly all of these aspects of the accelerator must work correctly for the accelerator as a whole to work. However, verifying each in turn is much simpler than trying to verify all aspects at once. Having verified that the slave bus operations function correctly, we can then use them to initiate computation. Then we can check that computation follows the intended sequence of steps, with master bus operations proceeding correctly, ignoring the actual addresses and pixel values. We can then make sure addresses are being generated correctly, and finally check that pixel values are computed correctly. Verifying a stream-processing accelerator would proceed similarly, but we would additionally need to verify that the accelerator interacts correctly with the source of data being processed.

For this verification process, we need to construct a testbench that mimics the behavior of the embedded system containing the accelerator. If we have a verified model of the embedded processor, we can include it in the testbench and write small test programs to run on it. The test programs write to accelerator registers to set up and initiate operations. On the other hand, if no processor model is available, we can write a *bus functional model* of the processor, that is, a model that performs a predetermined sequence of bus operations without actually executing any processor instructions. Our testbench also needs to include a memory model and bus arbiter. The memory, like the processor, need not be a fully functional model. Instead, it might simply engage in write and read operations on the bus, generating read data according to a predetermined rule and discarding write data. These simplifications allow us to focus our verification effort on the accelerator, and to create test cases in a controlled manner.

EXAMPLE 9.14 Develop a testbench for the Sobel accelerator that includes a bus functional processor model. The processor should program the accelerator to operate on an original image at address 008000_{16} to generate a derivative image at 053000_{16}. It should then read the status register once every 10μs until the done bit is set. The testbench should also include a bus arbiter that gives the accelerator priority, and a bus functional memory that returns 0 for reads and discards data from writes.

SOLUTION Our testbench is modeled after the general system organization shown in Figure 9.2. The accelerator is the design under verification, and the arbiter and bus functional processor and memory form the remainder of the testbench. We also include a clock and reset generator. The outline of the testbench module definition is

```
`timescale 1ns/1ns

module testbench;

  parameter           t_c                     = 10;
  parameter [22:0]    mem_base                = 23'h000000;
  parameter [22:0]    saobel_reg_base         = 23'h400000;
  parameter           sobel_int_reg_offset    = 0;
  parameter           sobel_start_reg_offset  = 4;
  parameter           sobel_O_base_reg_offset = 8;
  parameter           sobel_D_base_reg_offset = 12;
  parameter           sobel_status_reg_offset = 0;

  reg         clk, rst;

  wire        bus_cyc, bus_stb, bus_we;
  wire  [3:0] bus_sel;
  wire [22:0] bus_adr;
  wire        bus_ack;
  wire [31:0] bus_dat;
  wire        int_req;

  wire        sobel_cyc_o, sobel_stb_o, sobel_we_o;
  wire [21:0] sobel_adr_o;
  wire        sobel_ack_i;
  wire        sobel_stb_i;
  wire        sobel_ack_o;
  wire [31:0] sobel_dat_o;
  ...

  always begin     // Clock generator
    clk = 1'b1; #(t_c/2);
    clk = 1'b0; #(t_c/2);
  end

  initial begin    // Reset generator
    rst <= 1'b1;
    #(2.5*t_c) rst = 1'b0;
  end

  sobel duv ( .clk_i(clk),           .rst_i(rst),
              .cyc_o(sobel_cyc_o),   .stb_o(sobel_stb_o),
```

(*continued*)

```
              .we_o(sobel_we_o),
              .adr_o(sobel_adr_o),   .ack_i(sobel_ack_i),
              .cyc_i(bus_cyc),       .stb_i(sobel_stb_i),
              .we_i(bus_we),         .adr_i(bus_adr[3:2]),
              .ack_o(sobel_ack_o),
              .dat_o(sobel_dat_o),   .dat_i(bus_dat),
              .int_req(int_req) );
  ...

endmodule
```

The clock generator always block uses the parameter t_c for the clock cycle time, giving a clock frequency of 100MHz. The parameters mem_base and sobel_base define the base addresses of the memory (000000_{16}) and the Sobel accelerator registers (400000_{16}). Additional parameters define the offsets from the base address for the control and status registers. Next, the testbench includes nets for the bus address, data and control signals. As we will see shortly, these are multiplexed from the various sources in the system. The testbench also declares nets for connection specifically to the Sobel accelerator. Within the module, the accelerator is instantiated as the design under verification (duv) and connected to the nets.

The testbench code for the processor bus functional model is

```
reg         cpu_cyc_o, cpu_stb_o, cpu_we_o;
reg  [3:0]  cpu_sel_o;
reg [22:0]  cpu_adr_o;
wire        cpu_ack_i;
reg [31:0]  cpu_dat_o;
wire[31:0]  cpu_dat_i;
...

task bus_write ( input [22:0] adr, input [31:0] dat );
  begin
    cpu_adr_o = adr;
    cpu_sel_o = 4'b1111;
    cpu_dat_o = dat;
    cpu_cyc_o = 1'b1; cpu_stb_o = 1'b1; cpu_we_o = 1'b1;
    @(posedge clk); while (!cpu_ack_i) @(posedge clk);
  end
endtask
...

initial begin // Processor bus-functional model
  cpu_adr_o = 23'h000000;
  cpu_sel_o = 4'b0000;
```

(*continued*)

```
  cpu_dat_o = 32'h00000000;
  cpu_cyc_o = 1'b0; cpu_stb_o = 1'b0; cpu_we_o = 1'b0;
  @(negedge rst);
  @(posedge clk);
  // Write 008000 (hex) to 0_base_addr register
  bus_write(sobel_reg_base
            + sobel_0_base_reg_offset, 32'h00008000);
  // Write 053000 + 280 (hex) to D_base_addr register
  bus_write(sobel_reg_base
            + sobel_D_base_reg_offset, 32'h00053280);
  // Write 1 to interrupt control register (enable interrupt)
  bus_write(sobel_reg_base
            + sobel_int_reg_offset, 32'h00000001);
  // Write to start register (data value ignored)
  bus_write(sobel_reg_base
            + sobel_start_reg_offset, 32'h00000000);
  // End of write operations
  cpu_cyc_o = 1'b0; cpu_stb_o = 1'b0; cpu_we_o = 1'b0;
  begin: loop
    forever begin
      #10000;
      @(posedge clk);
      // Read status register
      cpu_adr_o = sobel_reg_base + sobel_status_reg_offset;
      cpu_sel_o = 4'b1111;
      cpu_cyc_o = 1'b1; cpu_stb_o = 1'b1; cpu_we_o = 1'b0;
      @(posedge clk); while (!cpu_ack_i) @(posedge clk);
      cpu_cyc_o = 1'b0; cpu_stb_o = 1'b0; cpu_we_o = 1'b0;
      if (cpu_dat_i[0]) disable loop;
    end
  end
end
```

The processor waits for completion of system reset, then performs the required sequence of bus write operations to initialize the accelerator. For each bus operation, described by the bus_write task, the processor assigns the appropriate values to the address, data and control signals, then waits for the accelerator to acknowledge completion of the operation. After completion of the write to the start register, the processor enters a loop in which it waits for 10μs, resynchronizes with the clock, then reads the accelerator status register. When the accelerator acknowledges completion of the read operation, the processor checks whether the done bit is 1. If so, the processor exits the loop, completing the test.

The testbench code for the memory bus functional model is

```
wire       mem_stb_i;
wire [3:0] mem_sel_i;
reg        mem_ack_o;
reg [31:0] mem_dat_o;
...

always begin // Memory bus-functional model
  mem_ack_o = 1'b0;
  mem_dat_o = 32'h00000000;
  @(posedge clk);
  while (!(bus_cyc && mem_stb_i)) @(posedge clk);
  if (!bus_we)
    mem_dat_o = 32'h00000000; // in place of read data
  mem_ack_o = 1'b1;
  @(posedge clk);
end
```

The memory repeatedly waits until the bus_cyc and mem_stb_i signals are both 1, indicating that a memory operation is required. If bus_we is 0, the operation is a read, so the memory provides zeros on the data outputs. In the case of a write operation, the memory does nothing with the input data. In either case, the memory sets the acknowledge signal to 1, and then on the next clock cycle clears the signal back to 0, completing the operation.

The arbiter for the testbench is somewhat more involved than the other testbench components. It uses the sobel_cyc_o and cpu_cyc_o signals as requests from the Sobel accelerator and the processor, respectively, and generates sobel_gnt and cpu_gnt grant signals. When either of the request signals is activated, the arbiter activates the corresponding grant. If both requests are activated in the same cycle, the arbiter gives preference to the accelerator, activating its grant and leaving the processor's grant inactive until the accelerator's request is deactivated. Since the grant outputs depend not only on the values of the request inputs, but also on the preceding history of request values, the arbiter must be implemented as a sequential circuit using an FSM. The state transition diagram is shown in Figure 9.11. The FSM is a Mealy machine, since that allows us to activate a grant signal in the same cycle in which the corresponding request is activated.

The testbench code for the arbiter is

```
parameter sobel = 1'b0, cpu = 1'b1;
reg arbiter_current_state, arbiter_next_state;
```

(*continued*)

```
reg sobel_gnt, cpu_gnt;
...

always @(posedge clk) // Arbiter FSM register
  if (rst)
    arbiter_current_state <= sobel;
  else
    arbiter_current_state <= arbiter_next_state;

always @* // Arbiter logic
  case (arbiter_current_state)
    sobel: if (sobel_cyc_o) begin
             sobel_gnt <= 1'b1; cpu_gnt <= 1'b0;
             arbiter_next_state <= sobel;
           end
           else if (!sobel_cyc_o && cpu_cyc_o) begin
             sobel_gnt <= 1'b0; cpu_gnt <= 1'b1;
             arbiter_next_state <= cpu;
           end
           else begin
             sobel_gnt <= 1'b0; cpu_gnt <= 1'b0;
             arbiter_next_state <= sobel;
           end
  cpu:     if (cpu_cyc_o) begin
             sobel_gnt <= 1'b0; cpu_gnt <= 1'b1;
             arbiter_next_state <= cpu;
           end else if (sobel_cyc_o && !cpu_cyc_o) begin
             sobel_gnt <= 1'b1; cpu_gnt <= 1'b0;
             arbiter_next_state <= sobel;
           end else begin
             sobel_gnt <= 1'b0; cpu_gnt <= 1'b0;
             arbiter_next_state <= sobel;
           end
endcase
```

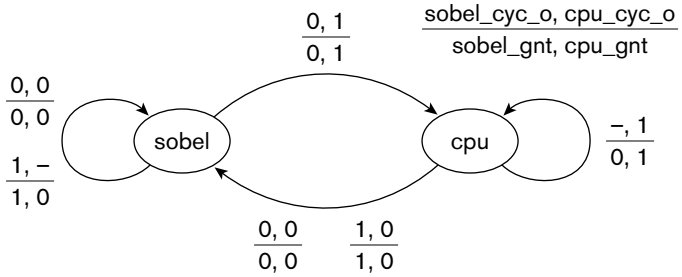


FIGURE 9.11 State transition diagram for the testbench arbiter.

The rest of the testbench code represents the bus multiplexers and slave select logic:

```
wire sobel_sel, mem_sel;
...

// Bus master multiplexers and logic

assign bus_cyc = sobel_gnt ? sobel_cyc_o : cpu_cyc_o;
assign bus_stb = sobel_gnt ? sobel_stb_o : cpu_stb_o;
assign bus_we  = sobel_gnt ? sobel_we_o  : cpu_we_o;

assign bus_sel = sobel_gnt ? 4'b1111 : cpu_sel_o;

assign bus_adr = sobel_gnt ? {1'b0, sobel_adr_o} : cpu_adr_o;

assign sobel_ack_i = bus_ack & sobel_gnt;
assign cpu_ack_i   = bus_ack & cpu_gnt;

// Bus slave logic

assign sobel_sel = (bus_adr & 23'h7FFFF0) == sobel_reg_base;
assign mem_sel  = (bus_adr & 23'h400000) == mem_base;

assign sobel_stb_i = bus_stb & sobel_sel;
assign mem_stb_i   = bus_stb & mem_sel;

assign bus_ack =   sobel_sel ? sobel_ack_o :
                   mem_sel   ? mem_ack_o :
                   1'b0;

// Bus data multiplexer

assign bus_dat =    sobel_gnt && bus_we || sobel_sel && !bus_we
                      ? sobel_dat_o :
                    cpu_gnt && bus_we
                      ? cpu_dat_o :
                    mem_dat_o;
```

The grant signals from the arbiter determine which source provides values for the bus control and address signals. They also gate the acknowledge signals back to the masters, so that a master that is waiting for the bus does not receive an acknowledgment from a slave for the active master's bus operation. The bus slave logic decodes addresses and determines which slave is selected. The select signals gate the strobe signal from the active master to the selected slave, and multiplex the selected slave's acknowledgment signal onto the bus_ack signal. The bus data multiplexer determines the source of data for the bus_dat signal,

depending on which master is active, which slave is selected, and whether the bus operation is a read or a write.

We can simulate the testbench of Example 9.14 to verify that the Sobel accelerator correctly responds to slave bus operations and performs master bus operations with correct addresses. We need to observe the values of the bus control and address signals, as well as the internal signals of the accelerator. Figure 9.12 shows a simulation waveform display of the bus signals during initialization of the accelerator by the processor bus functional model. Figures 9.13 through 9.15 show the internal signals of the accelerator during the start of processing a row (Figure 9.13), during steady state processing (Figure 9.14), and upon completion of processing a row and commencement of the next row (Figure 9.15). Finally, Figure 9.16 shows the internal signals on completion of processing an entire image.

While the verification shown here might give us confidence that the design is correct, it is by no means complete. For example, it doesn't demonstrate that the computation produces correct values according to the specification of the algorithm, and it doesn't show that the control sequencing is correct for all possible interactions between the accelerator and other bus masters. Creating test cases for simulation-based verification to cover all of these aspects is infeasible, given the number of permutations of data values and ways in which components can interact. Instead, we need to turn to more sophisticated verification techniques, such as constrained random test generation, coverage analysis, and property-based formal verification. We will return to the topic of verification again in Chapter 10, but we also refer the interested reader to advanced books on verification listed in Section 9.5, Further Reading.

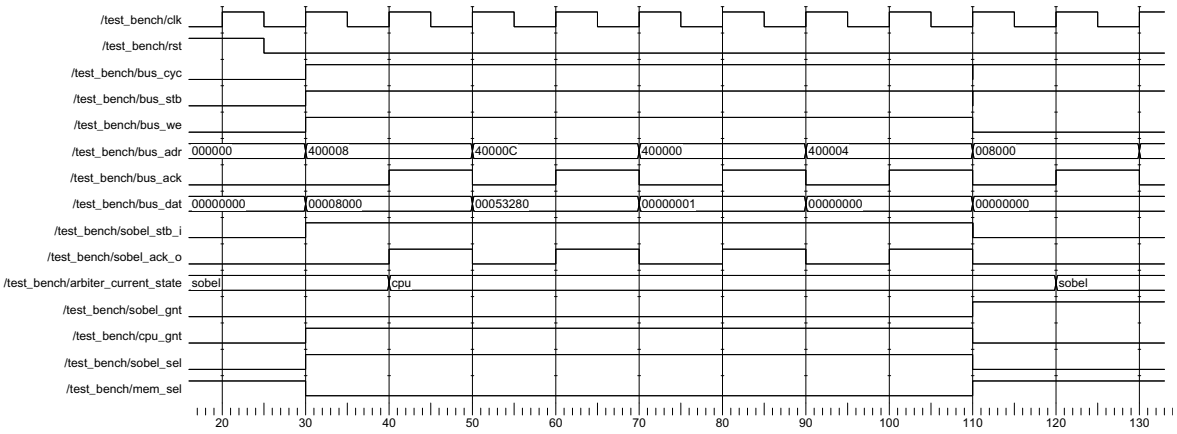


FIGURE 9.12 Waveform display of bus operations for initializing the Sobel accelerator.

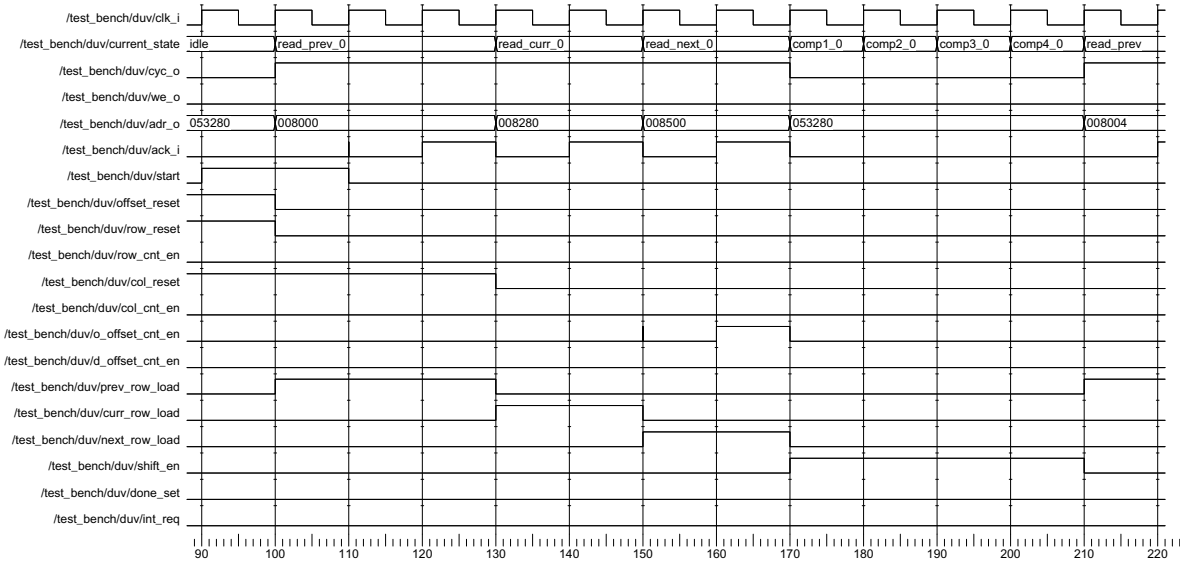


FIGURE 9.13 Waveform display of accelerator internal signals at the start of row processing.

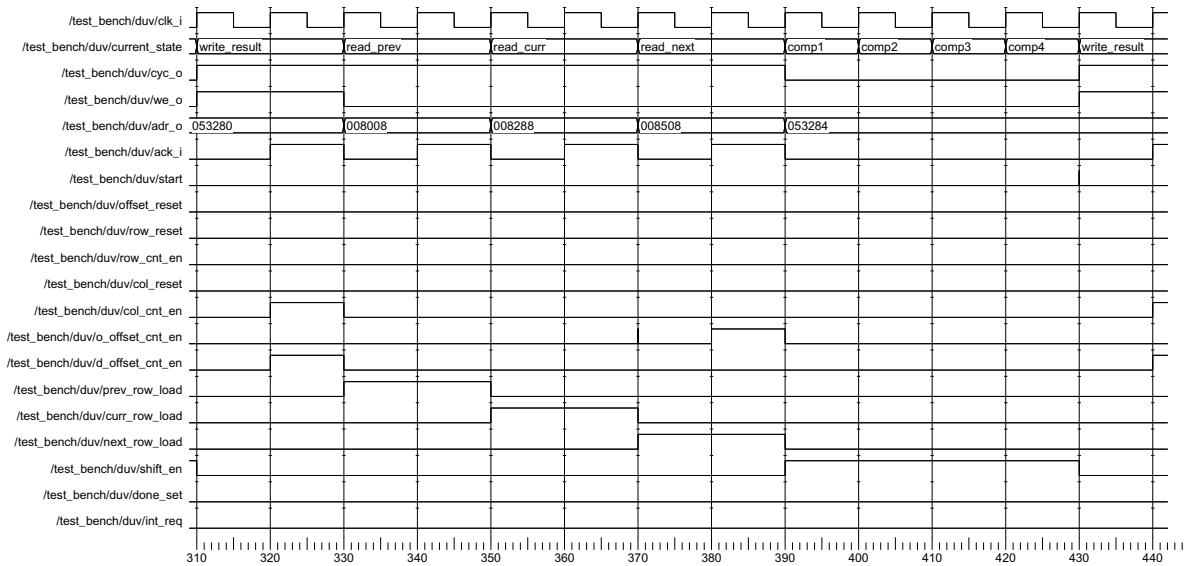


FIGURE 9.14 Waveform display showing row processing in the steady state.

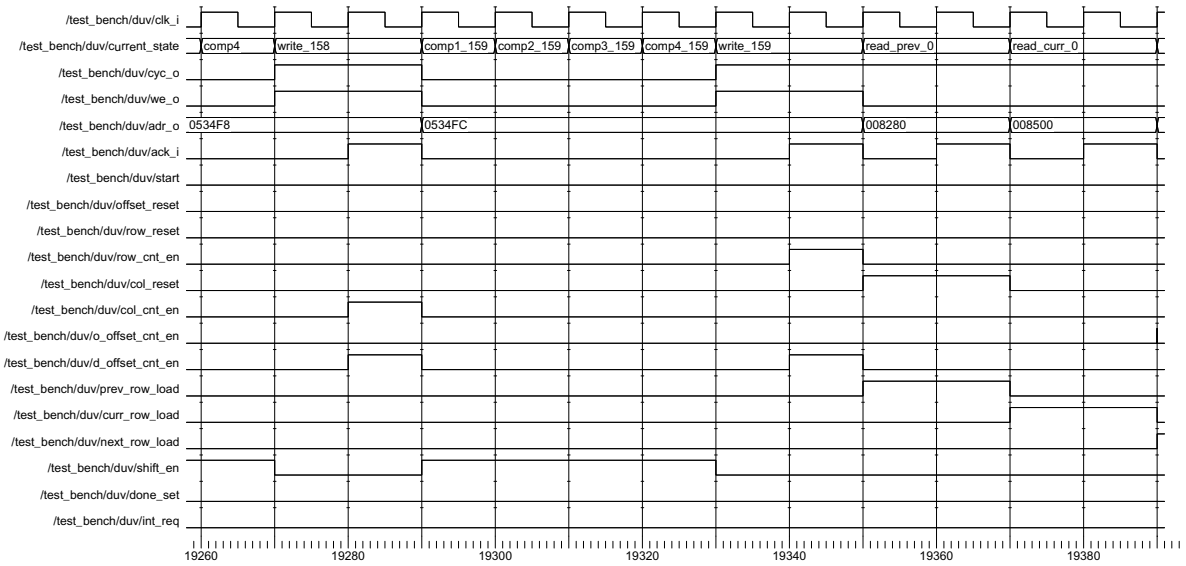


FIGURE 9.15 Waveform display showing completion of one row and commencement of the next.

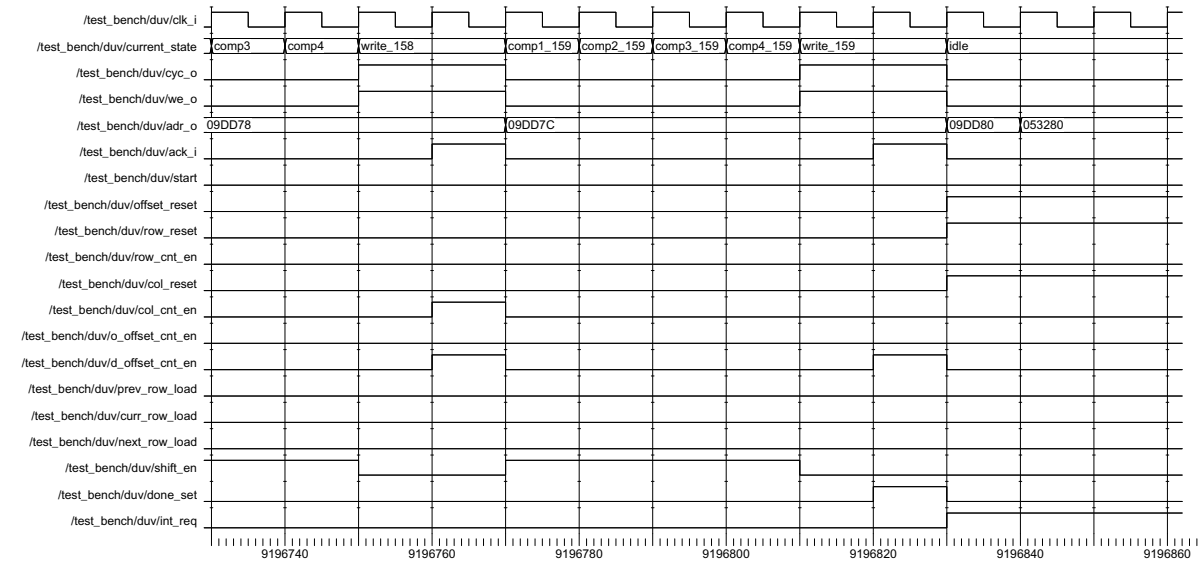


FIGURE 9.16 Waveform display showing completion of image processing.

KNOWLEDGE TEST QUIZ

1. Is it possible to verify an accelerator design using exhaustive testing? Why, or why not?
2. What is a bus functional model?
3. Given the arbiter in the testbench for the Sobel accelerator, what happens if the accelerator and the processor both request use of the bus in the same clock cycle?
4. What happens if the accelerator requests use of the bus while the processor is currently granted use?
5. Does the testbench verify correct computation of derivative pixel values?

9.4 CHAPTER SUMMARY

- Parallelism, performing multiple processing steps at once, allows accelerators to reduce the time required to complete an operation.
- An accelerator achieves parallelism by replicating hardware resources and by pipelining. This leads to cost/performance and power/performance trade-offs.
- The degree of achievable parallelism is constrained by data dependencies within a computation.
- Designing an accelerator involves analyzing an algorithm and identifying a kernel to be implemented in hardware. The remainder of the algorithm is implemented in embedded software.
- Amdahl's Law quantifies the overall speedup from accelerating a kernel of an algorithm.
- Accelerators and high-speed I/O controllers can use direct memory access (DMA) to transfer data to or from memory without processor intervention. An address generator in such a unit calculates memory addresses for DMA.
- An arbiter determines which of several bus masters can use the bus at any time to access bus slaves, such as memory and I/O controller registers.
- A block-processing accelerator processes blocks of data stored in memory. Many video and still-image processing applications are block oriented.
- A stream-processing accelerator processes data arriving from a source in a sequence of values. Digital-signal processing (DSP) is often stream oriented.
- Accelerators include control and status registers for use by embedded software.
- Verification of an accelerator using exhaustive simulation is generally not feasible. Aspects of operation can be verified independently, but a complete verification plan should include other forms of verification.

9.5 FURTHER READING

Computer Architecture: A Quantitative Approach, 4th Edition, John L. Hennessy and David A. Patterson, Morgan Kaufmann Publishers,

2007. An advanced textbook on computer architecture, covering instruction-level parallelism in depth.

Parallel Computer Architecture: A Hardware/Software Approach, David E. Culler and Jaswinder Pal Singh, Morgan Kaufmann Publishers, 1999. An in-depth treatment of parallel computing. While the book focuses on parallel computers, many of the principles can also be applied to architectures of hardware accelerators.

Understanding Digital Signal Processing, Richard G. Lyons, Prentice Hall, 2001. An introduction to the theory of digital signal processing (DSP).

Computers as Components: Principles of Embedded Computing System Design, Wayne Wolf, Morgan Kaufmann Publishers, 2005. Includes a discussion of accelerators in the context of embedded hardware and software design, with a video-processing accelerator as a case study.

Embedded Software Development with eCos, Anthony J. Massa, Prentice Hall, 2003. Describes the Embedded Configurable Operating System (eCos), including the hardware abstraction layer.

Comprehensive Functional Verification: The Complete Industry Cycle, Bruce Wile, John C. Goss and Wolfgang Roesner, Morgan Kaufmann Publishers, 2005. A detailed treatment of functional verification strategies and techniques.

EXERCISES

EXERCISE 9.1 In computer graphics applications, a three-dimensional vector representing a point's position in space can be transformed by multiplying by a 3×3 matrix:

$$\begin{bmatrix} P_x' \\ P_y' \\ P_z' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}$$

Determine the data dependencies in the computation and thus the maximum available parallelism.

EXERCISE 9.2 Devise a pipeline architecture that can perform the computation described in Exercise 9.1 using all the available parallelism. Assume a new input vector arrives and a result can be accepted on every clock cycle.

EXERCISE 9.3 If a kernel of an algorithm is accelerated by a factor of 100, and the kernel accounts for 90% of execution time before acceleration, what is the overall speedup?