Patrick R. Schaumont

A Practical Introduction to Hardware/Software Codesign

Second Edition

Springer

Contents

Part I Basic Concepts

Part II The Design Space of Custom Architectures

Part III Hardware/Software Interfaces

Part IV Applications

Part I
Basic Concepts

The first part of this book introduces important concepts in hardware-software codesign. We compare and contrast two schools of thought in electronic design: the mindset used by the hardware designer, as opposed to the mindset used by the software designer. We will demonstrate that hardware/software codesign is not just gluing together hardware and software components; instead, it's about finding the correct balance between flexibility and performance during design.

The trade-off between parallel and sequential implementations is another fundamental issue for the hardware/software co-designer; we will discuss a concurrent system model (data-flow), that can be converted into either a hardware (parallel) or else into a software (sequential) implementation.

Finally, we will show how a program in C can be analyzed and decomposed into control-flow and data-flow. This analysis is crucial to understand how a C program can be migrated into hardware. As we will discuss, a common approach to hardware/software codesign is to carry functionality from software into hardware, thereby improving the overall performance of the application.

Chapter 1
The Nature of Hardware and Software

1.1 Introducing Hardware/Software Codesign

Hardware/software codesign is a broad term to capture many different things in electronic system design. We start by providing a simple definition of *software* and *hardware*. It's by no means a universal definition, but it will help to put readers from different backgrounds on the same line. This section will also provide a small example of hardware/software codesign, and concludes with a definition of hardware/software codesign.

1.1.1 Hardware

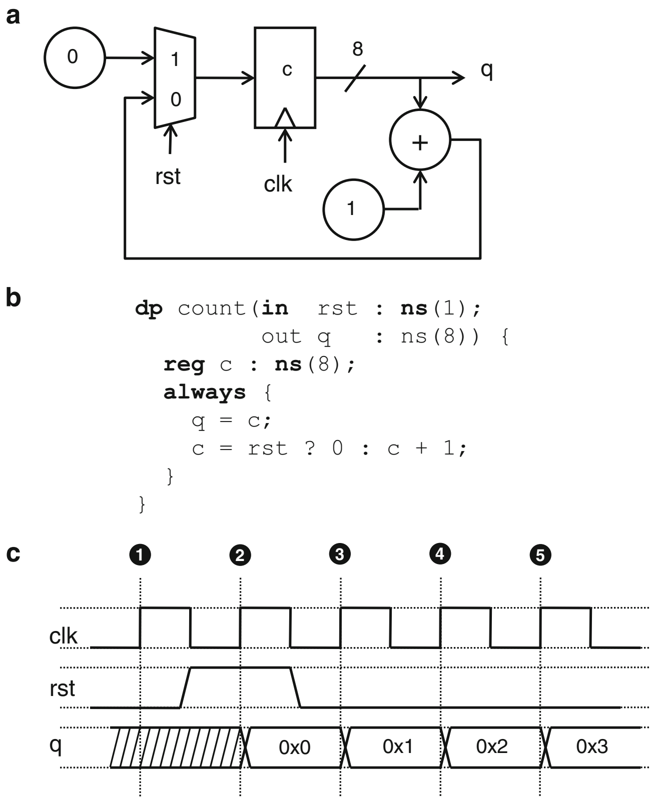
In this book, we will model hardware by means of single-clock synchronous digital circuits, created using combinational logic and flip-flops.

Such circuits can be modeled with building blocks such as for example registers, adders, and multiplexers. Cycle-based hardware modeling is often called register-transfer-level (RTL) modeling, because the behavior of a circuit can be thought of as a sequence of transfers between registers, with logic and arithmetic operations performed on the signals during the transfers.

Figure 1.1a gives an example of a hardware module captured in RTL. A register can be incremented or cleared depending on the value of the control signal `rst`. The register is updated on the up-going edges of a clock signal `clk`. The wordlength of the register is 8 bit. Even though the connections in this figure are drawn as single lines, each line represents a bundle of eight wires. Figure 1.1a uses graphics to capture the circuit; in this book, we will be using a hardware description language called GEZEL. Figure 1.1b shows the equivalent description of this circuit in GEZEL language. Chapter 5 will describe GEZEL modeling in detail.

Figure 1.1c illustrates the behavior of the circuit using a timing diagram. In such a diagram, time runs from left to right, and the rows of the diagram represent different

P.R. Schaumont, *A Practical Introduction to Hardware/Software Codesign*,
DOI 10.1007/978-1-4614-3737-6_1, © Springer Science+Business Media New York 2013



```
dp count(in  rst : ns(1);
         out q   : ns(8)) {
  reg c : ns(8);
  always {
    q = c;
    c = rst ? 0 : c + 1;
  }
}
```

Fig. 1.1 Hardware components

signals in the circuit. In this diagram, the register is cleared on clock edge 2, and it is incremented on clock edge 3, 4, and 5. Before clock edge 2, the value of the register is unknown, and the timing diagram indicates q's value as a shaded area. We will be using timing diagrams to describe the low-level behavior of hardware-software interfaces, and to describe events on on-chip buses.

The single-clock model is a very convenient abstraction for a designer who maps behavior (e.g. an algorithm) into discrete steps of one clock cycle. It enables this designer to envision how the hardware implementation of a particular algorithm should look like. The single-clock synchronous model cannot express every possible hardware circuit. For example, it cannot model events at a time resolution smaller than a clock cycle. As a result, some styles of hardware design cannot be captured with a single-clock synchronous model, including asynchronous hardware, dynamic logic, multi-phase clocked hardware, and hardware with latches. However, single-clock synchronous hardware is adequate to explain the key concepts of hardware-software co-design in this book.

Listing 1.1 C example

```
int max;

int findmax(int a[10]) {
  unsigned i;
  max = a[0];
  for (i=1; i<10; i++)
    if (a[i] > max) max = a[i];
}
```

1.1.2 *Software*

Hardware/software codesign deals with hardware/software interfaces. The low-level construction details of software are important, because they directly affect the performance and the implementation cost of the hardware/software interface. This book will discuss important implementation aspects of software, such as the organization of variables into memory, and the techniques to control this from within a high-level programming language such as C.

We will model software as single-thread sequential programs, written in C or assembly. Programs will be illustrated using listings, for example Listings 1.1 and 1.2. Most of the discussions in this book will be processor-independent. In some cases, we will assume a 32-bit architecture (e.g. ARM) or an 8-bit architecture (e.g. 8051).

A single-thread sequential C program has a surprisingly good match to the actual execution of that program on a typical micro-processor. For example, the sequential execution of C programs matches the sequential instruction fetch-and-execute cycle of micro-processors. The variables of C are stored in a single, shared-memory space, corresponding to the memory attached to the micro-processor. There is a close correspondence between the storage concepts of a micro-processor (registers, stack) and the storage types supported in C (`register int`, local variables). Furthermore, common datatypes in C (`char`, `int`) directly map into units of micro-processor storage (byte, word). Consequently, a detailed understanding of C execution is closely related to a detailed understanding of the microprocessor activity at a lower abstraction level.

Of course, there are many forms of software that do not fit the model of a single-thread sequential C program. Multi-threaded software, for example, creates the illusion of *concurrency* and lets users execute multiple programs at once. Other forms of software, such as object-oriented software and functional programming, substitute the simple machine model of the micro-processor with a more sophisticated one. Such more advanced forms of software are crucial to master the complexity of large software applications. However, they make abstraction of (i.e. hide) the activities within a micro-processor. For this reason, we will concentrate on simple, single-thread C.

Listing 1.2 ARM assembly example

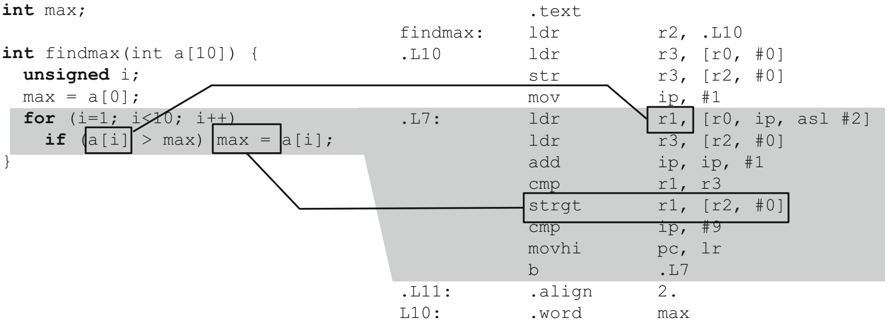
```
        .text
findmax:
        ldr     r2, .L10
        ldr     r3, [r0, #0]
        str     r3, [r2, #0]
        mov     ip, #1
.L7:
        ldr     r1, [r0, ip, asl #2]
        ldr     r3, [r2, #0]
        add     ip, ip, #1
        cmp     r1, r3
        strgt   r1, [r2, #0]
        cmp     ip, #9
        movhi   pc, lr
        b       .L7
.L11:
        .align  2
.L10:
        .word   max
```

The material in this book does not follow any specific micro-processor, and is agnostic of any particular type of assembly language. The book emphasizes the relationship between C and assembly code, and assumes that the reader is familiar with the concept of assembly code. Some optimization problems in hardware/software codesign can only be handled at the level of assembly coding. In that case, the designer needs to be able to link the software, as captured in C, with the program executing on the processor, as represented by assembly code. Most C compilers offer the possibility to generate an assembly listing of the generated code, and we will make use of that feature. Listing 1.2 for example, was generated out of Listing 1.1.

Linking the statements of a C program to the assembly instructions, is easier than you would think, even if you don't know the microprocessor targeted by the assembly program. As an example, compare Listings 1.1 and 1.2. An ideal starting point when matching a C program to an assembly program, is to look for similar structures: loops in C will be reflected through their branch statements in assembly; if-then-else statements in C will be reflected, in assembly language, as conditional branches, and labels. Even if you're unfamiliar with the assembly format of a particular micro-processor, you can often derive such structures easily.

Figure 1.2 gives an example for the programs in Listings 1.1 and 1.2. The `for`-loop in C is marked with a label and a branch instruction. All the assembly instructions in between the branch and the label are part of the body of the loop. Once the loop structure is identified, it is easy to derive the rest of the code, as the following examples show.

- The `if`-statement in C requires the evaluation of a greater-then condition. In assembly, an equivalent `cmp` (compare) instruction can be found.



```
int max;

int findmax(int a[10]) {
  unsigned i;
  max = a[0];
  for (i=1; i<10; i++)
    if (a[i] > max) max = a[i];
}
```

```
                .text
findmax:        ldr        r2, .L10
.L10            ldr        r3, [r0, #0]
                str        r3, [r2, #0]
                mov        ip, #1
.L7:            ldr        r1, [r0, ip, asl #2]
                ldr        r3, [r2, #0]
                add        ip, ip, #1
                cmp        r1, r3
                strgt      r1, [r2, #0]
                cmp        ip, #9
                movhi      pc, lr
                b          .L7
.L11:           .align     2.
L10:            .word      max
```

Fig. 1.2 Mapping C to assembly

This shows that the operands `r1` and `r3` of the compare instruction must contain `a[i]` and `max` of the C program. Both of these variables are stored in memory; `a[i]` because it's an indexed variable, and `max` because it's a global variable. Indeed, looking at the preceding instruction in the C program, you can see that both `r1` and `r3` are defined through `ldr` (load-register) instructions, which require an address.

- The address for the load of `r1` equals `[r0, ip, asl #2]`, which stands for the expression `r0 + (ip << 2)`. This may not be obvious if this is the first time you are looking at ARM assembly; but it's something you will remember quickly. In fact, the format of the expression is easy to explain. The register `ip` contains the loop counter, since `ip` is incremented once within the loop body, and the value of `ip` is compared with the loop boundary value of 9. The register `r0` is the *base address* of `a[]`, the location in memory where `a[0]` is stored. The shift-over-2 is needed because `a[]` is an array of integers. Microprocessors use byte-addressable memory, and integers are stored 4 byte-locations part.
- Finally, the conditional assignment of the `max` variable in C is not implemented using conditional branch instructions in assembly. Instead, a `strgt` (store-if-greater) instruction is used. This is a *predicated* instruction, an instruction that only executes when a given conditional flag is true.

The bottom line of this analysis is that, with a minimal amount of effort, you are able to understand a great deal on the behavior of a microprocessor simply by comparing C programs with equivalent assembly programs. In Chap. 7, you will use the same approach to analyze the quality of the assembly code generated by a compiler out of C code.

1.1.3 Hardware and Software

The objective of this book is to discuss the combination of hardware design and software design in all its forms. Hardware as well as software can be modeled using

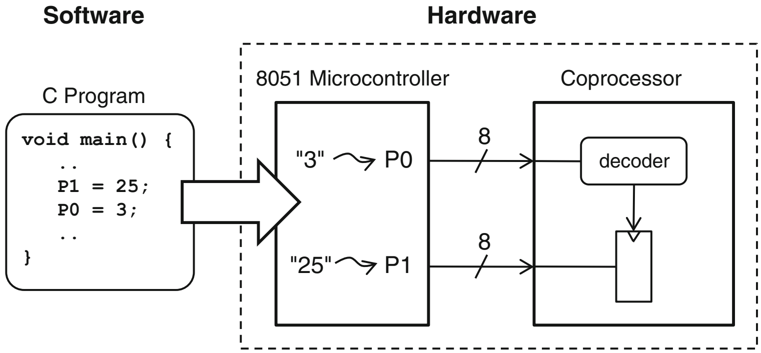


Fig. 1.3 A codesign model

RTL programs and C programs respectively. A term *model* merely indicates they are not the actual implementation, but only a representation of it. An RTL program is a *model* of a network of logic gates; a C program is a *model* of a binary image of micro-processor instructions. It is not common to talk about C programs as *models*; in fact, software designers think of C programs as actual implementations. In this book, we will therefore refer to hardware *models* and C or assembly *programs*.

Models are an essential part of the design process. They are a formal representation of a designers' intent and they are used as input for simulation tools and implementation tools. In hardware/software codesign, we are working with models that are partly written as C programs, and partly as RTL programs. We will discuss this idea by means of a simple example.

Figure 1.3 shows an 8051 micro-controller and an attached coprocessor. The coprocessor is attached to the 8051 micro-controller through two 8-bit ports `P0` and `P1`. A C program executes on the 8051 microcontroller, and this program contains instructions to write data to these two ports. When a given, pre-defined value appears on port `P0`, the coprocessor will make a copy of the value present on port `P1` into an internal register.

This very simple design can be addressed using hardware/software codesign; it includes the design of a hardware model and the design of a C program. The hardware model contains the 8051 processor, the coprocessor, and the connections between them. During execution, the 8051 processor will execute a software program written in C. Listing 1.3 shows that C program. Listing 1.4 shows an RTL hardware model for this design, written in the GEZEL language.

The C driver sends three values to port `P1`, by calling a function `sayhello`. That function also cycles the value on port `P0` between `ins_hello` and `ins_idle`, which are encoded as value `1` and `0` respectively.

The hardware model includes both the microcontroller and the coprocessor. The overall coprocessor behavior in Listing 1.4 is like this: when the `ins` input changes from 0 to 1, then the `din` input will be printed in the next clock cycle.

Listing 1.3 8051 driver program

```
#include <8051.h>

enum {ins_idle, ins_hello};

void sayhello(char d) {
  P1 = d;
  P0 = ins_hello;
  P0 = ins_idle;
}

void terminate() {
  // special command to stop simulator
  P3 = 0x55;
}

void main() {
  sayhello(3);
  sayhello(2);
  sayhello(1);
  terminate();
}
```

The coprocessor is on lines 1–19. This particular hardware model is a combination of a finite state machine (lines 10–18) and a datapath (lines 1–9), a modeling method known as FSMD (for finite-state-machine with datapath). We will discuss FSMD in detail in Chap. 5. The FSMD is quite easy to understand. The datapath contains several instructions: `decode` and `hello`. The FSM controller selects, each clock cycle, which of those instructions to execute. For example, lines 15–16 shows the following control statement.

```
@s1 if (insreg == 1) then (hello, decode) -> s2;
                     else (decode)        -> s1;
```

This means: when the value of `insreg` is 1, and the current state of the FSM controller is `s1`, then the datapath will execute instructions `hello` and `decode`, and the FSM controller next-state will become `s2`. When the value of `insreg` would be 0, the datapath would execute only instruction `decode` and the next-state of the FSM controller would be `s1`.

The 8051 microcontroller is captured in Listing 1.4 as well. However, the internals of the microcontroller are not shown; only the hardware interfaces relevant to the coprocessor are included. The 8051 microcontroller is captured with three `ipblock` (GEZEL library modules), on lines 21–38. The first `ipblock` is an `i801system`. It represents the 8051 microcontroller core, and it indicates the name of the compiled C program that will execute on this core (`driver.ihx` on line 22). The other two `ipblock` (lines 28–38) are two 8051 output ports, one to model port `P0`, and the second to model port `P1`.

Finally, the coprocessor and the 8051 ports are wired together in a top-level module, shown in lines 40–46. We can now simulate the entire model, including hardware and software, as follows. First, the 8051 C program is compiled to a binary. Next, the GEZEL simulator will combine the hardware model and the 8051 binary executable in a *cosimulation*. The output of the simulation model is shown below.

```
> sdcc driver.c
> /opt/gezel/bin/gplatform hello.fdl
i8051system: loading executable [driver.ihx]
9662 Hello! You gave me 3/3
9806 Hello! You gave me 2/2
9950 Hello! You gave me 1/1
Total Cycles: 10044
```

You can notice that the model produces output on cycles 9,662, 9,806, and 9,950, while the complete C program executes in 10,044 cycles. The evaluation and analysis of cycle-accurate behavior is a very important aspect of codesign, and we will address it throughout the book.

1.1.4 Defining Hardware/Software Codesign

The previous example motivates the following traditional definition of hardware/-software codesign.

> Hardware/Software Codesign is the design of cooperating hardware components and software components in a single design effort.

For example, if you would design the architecture of a processor and at the same time develop a program that could run on that processor, then you would be using hardware/software codesign. However, this definition does not tell precisely what *software* and *hardware* mean. In the previous example, the software was a C program, and the hardware was an 8051 microcontroller with a coprocessor. In reality, there are many forms of hardware and software, and the distinction between them easily becomes blurred. Consider the following examples.

- A Field Programmable gate Array (FPGA) is a hardware circuit that can be reconfigured to a user-specified netlist of digital gates. The program for an FPGA is a 'bitstream', and it is used to configure the netlist topology. Writing 'software' for an FPGA really looks like hardware development – even though it is software.
- A soft-core is a processor implemented in the bitstream of an FPGA. However, the soft-core itself can execute a C program as well. Thus, software can execute on top of other 'software'.

- A Digital-Signal Processor (DSP) is a processor with a specialized instruction-set, optimized for signal-processing applications. Writing efficient programs for a DSP requires detailed knowledge of these specialized instructions. Very often, this means writing assembly code, or making use of a specialized software library. Hence, there is a strong connection between the efficiency of the software and the capabilities of the hardware.
- An Application-Specific Instruction-set Processor (ASIP) is a processor with a customizable instruction set. The hardware of such a processor can be extended, and these hardware extensions can be encapsulated as new instructions for the processor. Thus, an ASIP designer will develop a hardware implementation for these custom instructions, and subsequently write software that uses those instructions.
- The CELL processor, used in the Playstation-3, contains one control processor and eight slave-processors, interconnected through a high-speed on-chip network. The software for a CELL is a set of nine concurrent communicating programs, along with configuration instructions for the on-chip network. To maximize performance, programmers have to develop CELL software by describing simultaneously the computations and the communication activities in each processor.

These examples illustrate a few of the many forms of hardware and software that designers use today. A common characteristic of all these examples is that creating the ‘software’ requires intimate familiarity with the ‘hardware’. In addition, hardware covers much more than RTL models: it also includes specialized processor datapaths, the FPGA fabric, multi-core architectures, and more.

Let us define the *application* as the overall function of a design, covering its implementation in hardware as well as in software. We can define hardware/software codesign as follows.

> Hardware/Software Codesign is the partitioning and design of an application in terms of fixed and flexible components.

We used the term ‘fixed component’ instead of hardware component, and ‘flexible component’ instead of software component. A fixed component is often hardware, but it is not restricted to it. Similarly, a flexible component is often software, but it is not restricted to it. In the next section, we clarify the balance between fixed and flexible implementations.

1.2 The Quest for Energy Efficiency

Choosing between implementing a design in hardware or in software may seem trivial, if you only consider design effort. Indeed, from a designers' point-of-view, the easiest approach is to write software, for example in C. Software is easy and flexible, software compilers are fast, there are many libraries available with source code, and development systems (personal computers and laptops) are plentiful and cheap. Furthermore, why go through the effort of designing a new hardware architecture when there is already one available that will do the job for your implementation (namely, the RISC processor)?

In reality, choosing between a hardware implementation and a software implementation is much more subtle, and it is driven by both technological as well as by economical reasons. We start with two technological arguments (performance and energy efficiency), and next provide a more balanced view on the trade-off between hardware and software.

1.2.1 Performance

Another way to compare hardware and software is to compare them in terms of their performance. Performance could be expressed as the amount of work done per unit of time. Let's define a unit of work as the processing of 1 bit of data. The unit of time can be expressed in clock cycles or in seconds.

Figure 1.4 illustrates various cryptographic implementations in software and hardware that have been proposed over the past few years (2003–2008). All of them are designs that have been proposed for embedded applications, where the trade-off between hardware and software is crucial. The graph shows performance in bits per cycle, and demonstrates that hardware crypto-architectures have, on the average, a higher performance compared to embedded processors. Of course, the clock frequency should be taken into account. A hardware implementation may execute many operations per clock cycle, but a processor may run at a much higher clock frequency. The faster processor may outdo the advantage of the parallel hardware implementation in terms of performance.

1.2.2 Energy Efficiency

A second important factor in the selection between hardware and software is the energy needed for computations. This is especially important for portable, battery-operated applications. The *energy-efficiency* is the amount of useful work done per unit of energy. A better energy-efficiency clearly implies longer battery-life.

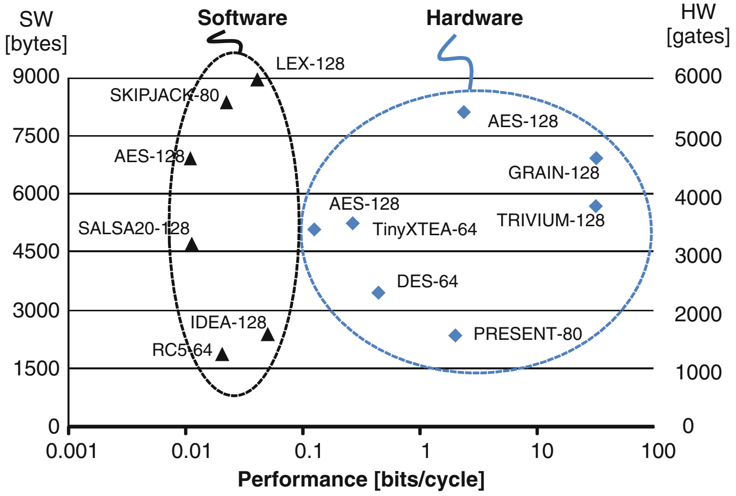


Fig. 1.4 Cryptography on small embedded platforms

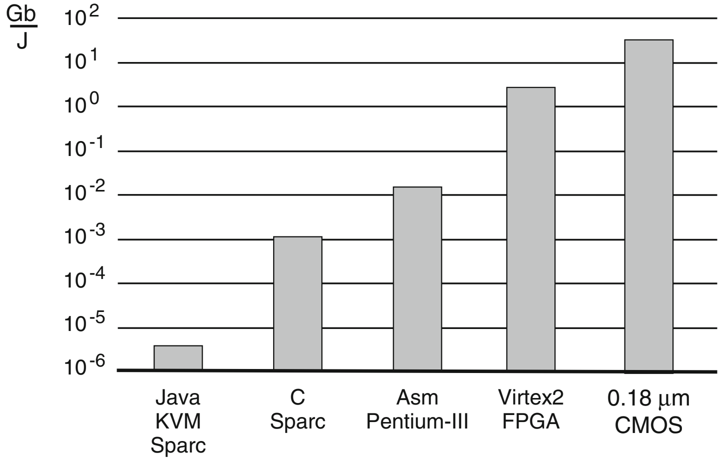


Fig. 1.5 Energy efficiency

Figure 1.5 shows the example of a particular encryption application (AES) for different target platforms. The platforms include: Java on top of a Java Virtual machine on top of an embedded processor; C on top of an embedded processor; optimized assembly-code on top of a Pentium-III processor; Verilog code on top of a Virtex-II FPGA; and an ASIC implementation using 0.18 μm CMOS standard cells. The logarithmic Y-axis shows the amount of Gigabits that can be encrypted on each of these platforms with a single Joule of energy. Keep in mind that the application is the same for all these architectures, and consists of encrypting bits. As indicated by the figure, the energy-efficiency varies over many *orders of*

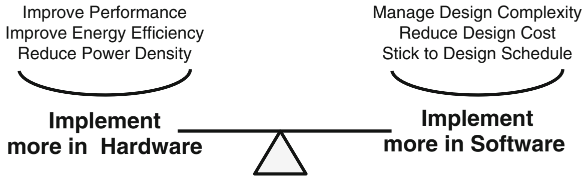


Fig. 1.6 Driving factors in hardware/software codesign

magnitude. If these architectures are being used in hand-held devices, where energy is a scarce resource, obviously there is a strong motivation to use a less flexible, more specialized architecture. For the same reason, you will never find a high-end workstation processor in a cell phone.

1.3 The Driving Factors in Hardware/Software Codesign

As pointed out in the previous section, energy-efficiency and relative performance are two important factors to prefer a (fixed, parallel) hardware implementation over a (flexible, sequential) software implementation. The complete picture, however, is more complicated. In the design of modern electronic systems, many trade-offs have to be made, often between conflicting objectives. Figure 1.6 shows that some factors argue for *more* software, while other factors argue for *more* hardware. The following are arguments in favor of increasing the amount of on-chip dedicated hardware.

- **Performance:** The classic argument in favor of dedicated hardware design has been increased performance: more work done per clock cycle. That is still one of the major factors. Increased performance is obtained by specializing the architecture used to implement the application. This can be done, for example, by introducing dedicated hardware components to accelerate part of the application.
- **Energy Efficiency:** Almost every electronic consumer product today carries a battery (iPod, PDA, mobile phone, Bluetooth device, . . .). Batteries have limited energy storage. On the other hand, these consumer devices are used for similar services and applications as traditional high-performance personal computers. By moving (part of) the flexible software of a design into fixed hardware, the energy-efficiency of the overall application can increase (See Fig. 1.5).
- **Power Density:** The power dissipation of a circuit, and the associated thermal profile, is directly proportional to their clock frequency. In modern processors, the power density is at the limits of cost-effective cooling technology; further improvement of performance can therefore not be achieved by increasing the clock frequency even more. Instead, there is a broad and fundamental shift occurring towards parallel computer architectures. At this moment, there is no dominant parallel computer architecture that has demonstrated to be effective for all applications.

  Some of the current candidates for parallel computing include symmetric multiprocessors attached to the same memory (SMP); Field Programmable Gate

Arrays used as accelerator engines for classic processors (FPGA); and multi-core and many-core architectures such as Graphics Processing Engines with general-purpose compute capabilities (GPU). It is likely that all of these architectures will coexist for a good amount of time to come.

Note that parallel architectures are not programmed in C. The current parallel architectures all come with their own style of parallel programming, and the search for a good, universal, parallel programming model is still on.

The following arguments, on the other hand, argue for flexibility and thus for increasing the amount of on-chip software.

- **Design Complexity:** Modern electronic systems are so complex, that it's not a good idea to hard-code all design decisions in fixed hardware. Instead, a common approach has been to keep the implementation as flexible as possible, typically by using programmable processors that run software. Software, and the flexibility it provides, is then used as a mechanism to develop the application at a higher level of abstraction (in software), as a mechanism to cope with future needs, and as a mechanism to resolve bugs. Thus, the flexibility of software is used to cope with the complexity of the design.
- **Design Cost:** New chips are very expensive to design. As a result, hardware designers make chips programmable so that these chips can be reused over multiple products or product generations. The SoC is a good example of this trend. However, programmability can be found in many different forms other than embedded processors: reconfigurable systems are based on the same idea of reuse-through- reprogramming.
- **Shrinking Design Schedules:** Each new generation of technology tends to replace the older one more quickly. In addition, each of these new technologies is more complex compared to the previous generation. For a design engineer, this means that each new product generation brings more work, and that the work needs to be completed in a shorter amount of time.

  Shrinking design schedules require engineering teams to work on multiple tasks at the same time: hardware and software are developed concurrently. A software development team will start software development as soon as the characteristics of the hardware platform are established, even *before* an actual hardware prototype is available.

Finding the correct balance between all these factors is obviously a very complex problem. In this book, we will restrict the optimization space to performance versus resource cost.

Adding hardware to a software solution may increase the performance of the overall application, but it will also require more resources. In terms of the balance of Fig. 1.6, this means that we will balance *Design Cost* versus *Performance*.

1.4 The Hardware-Software Codesign Space

The trade-offs discussed in the previous section need to be made in the context of a *design space*. For a given application, there are many different possible solutions. The collection of all these implementations is called the hardware-software codesign space. Figure 1.7 gives a symbolic representation of this design space, and it indicates the main design activities in this design space.

1.4.1 The Platform Design Space

The objective of the design process is to implement a specification onto a target platform. In hardware-software codesign, we are interested in programmable components. Figure 1.7 illustrates several examples: A RISC microprocessor, a Field Programmable Gate Array (FPGA), a Digital Signal Processor (DSP), an Application-Specific Instruction-set Processor (ASIP) and finally an Application-Specific Integrated Circuit (ASIC).

Mapping an application onto a platform means writing software for that platform, and, if needed, customizing the hardware of the platform. Software as well as hardware have a very different meaning depending on the platform.

- In the case of a RISC processor, software is written in C, while the hardware is a general-purpose processor.

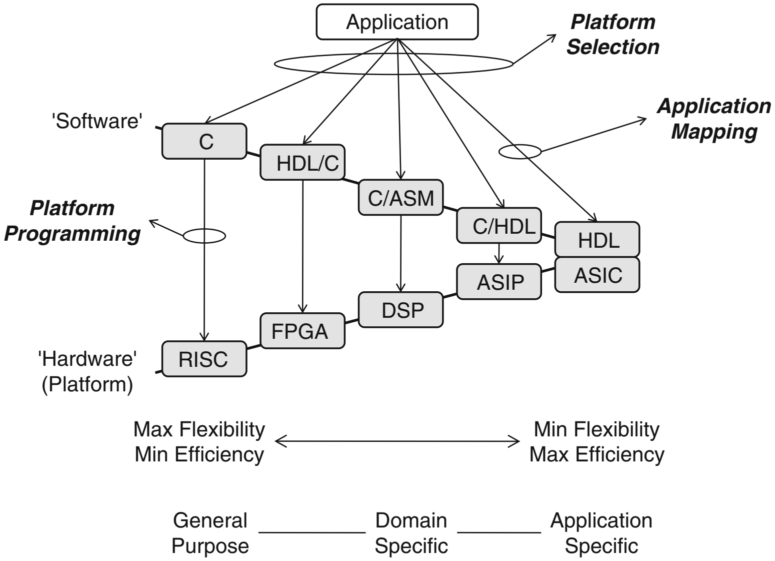


Fig. 1.7 The hardware-software codesign space

- In the case of a Field Programmable Gate Array, software is written in a Hardware Description Language (HDL). When the FPGA contains a soft-core processor, as discussed above, we will also write additional platform software in C.
- A Digital Signal Processor uses a combination of C and assembly code for software. The hardware is a specialized processor architecture, adapted to signal processing operations.
- An Application-Specific Instruction-set Processor is a processor that can be specialized to a particular application domain, for example by adding new instructions and by extending the processor datapath. The 'software' of an ASIP thus can contain C code as well as a hardware description of the processor extensions.
- Finally, in the case of an ASIC, the application is written in HDL, which is then converted into a hardcoded netlist. In contrast to other platforms, ASICs are typically non-programmable. In an ASIC, the application and the platform have merged to a single entity.

The platforms in Fig. 1.7 are organized, from left to right, according to their flexibility. General-purpose platforms, such as RISC and FPGA, are able to support a broad range of applications. Application-specific platforms, such as the ASIC, are optimized to execute a single application. In between the general purpose platform and the application-specific platform, is a third class of architectures called the *domain-specific* platform. Domain-specific platforms are optimized to execute applications from a particular domain. Signal-processing, cryptography, networking, are all examples of domains. A domain may have sub-domains. For example, one could further separate signal processing into voice-signal processing and video-signal processing and devise optimized platforms for each of these cases. The DSP and the ASIP are two examples of domain-specific platforms.

1.4.2 Application Mapping

Each of the above platforms in Fig. 1.7 presents a different trade-off between flexibility and efficiency. The wedge-shape of Fig. 1.7 expresses this idea, and it can be explained as follows.

Flexibility means how well the platform can be adapted to different applications. Flexibility in platforms is desired because it allows designers to make changes to the application after the platform is fabricated. Very flexible platforms, such as RISC and FPGA, are programmed with general purpose languages. When a platform becomes more specialized, the programming tends to become more specialized as well. We visualize this by drawing the application closer to the platform.

Different platforms may also provide different levels of efficiency. *Efficiency* can either relate to absolute performance (i.e. time-efficiency) or to the efficiency in using energy to implement computations. Under the right application, a specialized

Listing 1.5 dot product in C64x DSP processor

```
        LDDW   .D2T2    *B_n++,B_reg1:B_reg0
||      LDDW   .D1T1    *A_m++,A_reg1:A_reg0

        DOTP2  .M2X     A_reg0,B_reg0,B_prod
||      DOTP2  .M1X     A_reg1,B_reg1,A_prod

        SPKERNEL 4, 0
||      ADD    .L2      B_sum,B_prod,B_sum
||      ADD    .L1      A_sum,A_prod,A_sum
```

platform will be more efficient than a general platform, because its hardware components are optimized for that application. We can visualize this by moving the platform closer to the application in the case of specialized platforms.

The effect of the flexibility-efficiency trade-off on the source code of software can be illustrated with the following example. Consider the execution of the dot-product on a DSP processor such as TI's C64x. In C, the dot-product is a vector operation that can be expressed in single compact loop:

```
sum=0;
for (i=0; i<N; i++)
   sum += m[i]*n[i];
```

Listing 1.5 shows the body of the loop, optimized as assembly code for the TI C64x DSP processor. The TI C64x is a highly parallel processor that has two multiply-accumulate units. It can compute *two* loop iterations of the C loop at the same time. Several instructions are preceded by `||`. Those instructions will be executing in parallel with the previous instructions. Even though Listing 1.5 spans nine lines, it consists of only three instructions. Thus, Listing 1.5 has more efficiency than the original C program, but the TI assembly software is specific to the TI processor. A gain in efficiency was obtained at the cost of flexibility (or portability).

An interesting, but very difficult question is *how* one can select a platform for a given specification, and *how* one can map an application onto a selected platform. Of these two questions, the first one is the hardest. Designers typically answer it based on their previous experience with similar applications. The second question is also very challenging, but it is possible to answer it in a more systematic fasion, using a design methodology. A *design method* is a systematic sequence of steps to convert a specification into an implementation. Design methods cover many aspects of application mapping, such as optimization of memory usage, design performance, resource usage, precision and resolution of data types, and so on. A design method is a canned sequence of design steps. You can learn it in the context of one design, and next apply this design knowledge in the context of another design.

1.5 The Dualism of Hardware Design and Software Design

In the previous sections, we discussed the driving forces in hardware/software codesign, as well as its design space. Clearly, there are compelling reasons for hardware-software codesign, and there is a significant design space to explore. A key challenge in hardware-software codesign is that a designer needs to combine two radically different design paradigms. In fact, hardware and software are the dual of one another in many respects. In this section, we examine these fundamental differences. Table 1.1 provides a synopsis.

- **Design Paradigm:** In a hardware model, circuit elements operate in parallel. Thus, by using more circuit elements, more work can be done within a single clock cycle. Software, on the other hand, operates sequentially. By using more operations, a software program will take more time to complete. Designing requires the decomposition of a specification in lower level primitives, such as gates (in hardware) and instructions (in software). A hardware designer solves problems by decomposition in space, while a software designer solves problems by decomposition in time.
- **Resource Cost:** Resource cost is subject to a similar dualism between hardware and software. Decomposition in space, as used by a hardware designer, means that a more complex design requires more gates. Decomposition in time, as used by a software designer, implies that a more complex design requires more instructions to complete. Therefore, the resource cost for hardware is circuit area, while the resource cost for software is execution time.
- **Flexibility:** Software excels over hardware in the support of flexibility. Flexibility is the ease by which the application can be modified or adapted. In software, flexibility is essentially for free. In hardware on the other hand, flexibility is not trivial. Hardware flexibility requires that circuit elements can be easily reused for different activities or subfunctions in a design. A hardware designer has to think carefully about such reuse: flexibility needs to be designed into the circuit.
- **Parallelism:** A dual of flexibility can be found in the ease with which parallel implementations can be created. For hardware, parallelism comes for free as part of the design paradigm. For software, on the other hand, parallelism is a major challenge. If only a single processor is available, software can only implement concurrency, which requires the use of special programming constructs such as

Table 1.1 The dualism of hardware and software design

	Hardware	Software
Design Paradigm	Decomposition in space	Decomposition in time
Resource cost	Area (# of gates)	Time (# of instructions)
Flexibility	Must be designed-in	Implicit
Parallelism	Implicit	Must be designed-in
Modeling	Model $\neq$ implementation	Model $\sim$ implementation
Reuse	Uncommon	Common

threads. When multiple processors are available, a truly parallel software implementation can be made, but inter-processor communication and synchronization becomes a challenge.

- **Modeling:** In software, modeling and implementation are very close. Indeed, when a designer writes a C program, the compilation of that program for the appropriate target processor will also result in the implementation of the program. In hardware, the model and the implementation of a design are distinct concepts. Initially, a hardware design is modeled using a Hardware Description Language. Such a hardware description can be simulated, but it is not an implementation of the actual circuit. Hardware designers use a hardware *description* language, and their programs are models which are later transformed to implementation. Software designers use a software *programming* language, and their programs are an implementation by itself.
- **Reuse:** Finally, hardware and software are also quite different when it comes to Intellectual Property Reuse or IP-reuse. The idea of IP-reuse is that a component of a larger circuit or a program can be packaged, and later reused in the context of a different design. In software, IP-reuse has known dramatic changes in recent years due to open source software and the proliferation of open platforms. When designing a complex program these days, designers will start from a set of standard libraries that are well documented and that are available on a wide range of platforms. For hardware design, IP-reuse still in its infancy. Compared to software, IP-reuse in hardware has a long way to go.

This summary comparison indicates that in many aspects, hardware design and software design are based on dual concepts. Hence, being able to effectively transition from one world of design to the other is an important asset for a hardware/software codesigner. In this book, we will rely on this dualism, and attempt to combine the best concept of hardware design with the best concepts in software design. Our objective is not only to excel as a hardware designer or a software designer; our objective is to excel as a system designer.

1.6 Modeling Abstraction Level

The abstraction level of a model is the amount of detail that is available in a model. A lower abstraction level has more details, but constructing a design at lower abstraction level requires more effort.

This section defines the abstraction levels used in this book. We will differentiate the abstraction levels based on time-granularity. A smaller time-granularity typically implies that activities are expressed in a larger amount of (usually small) time steps. There are five abstraction levels commonly used by computer engineers for the design of electronic hardware-software systems. Starting at the lowest abstraction level, we enumerate the five levels. Figure 1.8 illustrates the hierarchy among these abstraction levels.

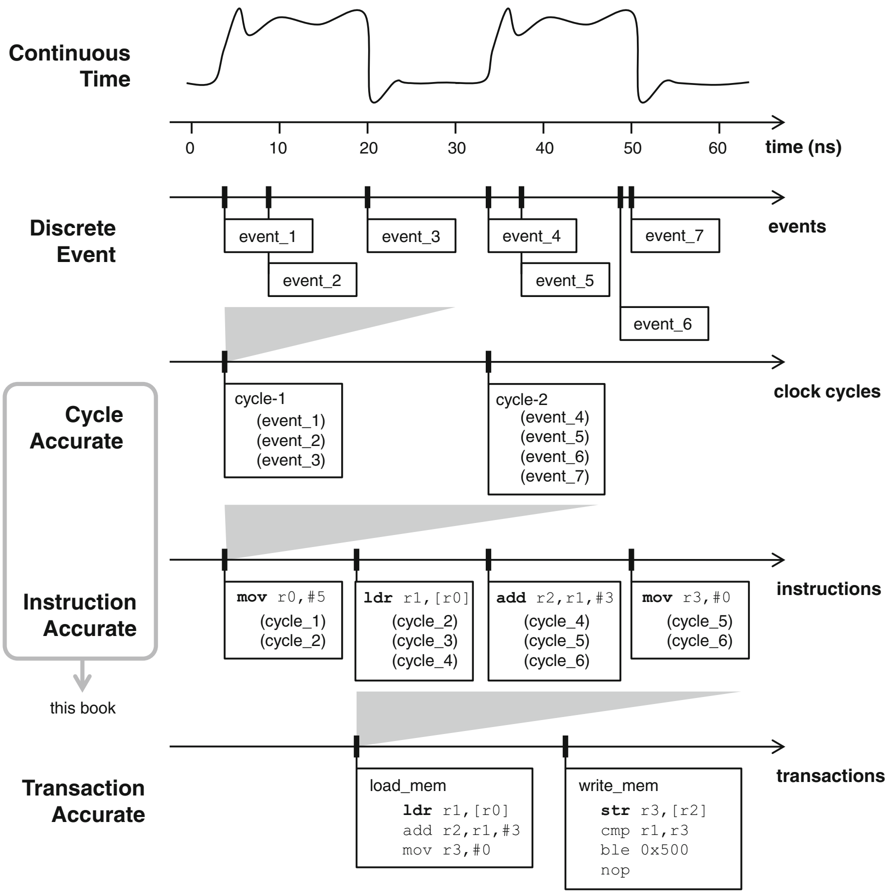

Continuous Time
0
10
20
30
40
50
60
time (ns)
Discrete Event
event_1
event_2
event_3
event_4
event_5
event_6
event_7
events
Cycle Accurate
cycle-1 (event_1) (event_2) (event_3)
cycle-2 (event_4) (event_5) (event_6) (event_7)
clock cycles
Instruction Accurate
mov r0,#5 (cycle_1) (cycle_2)
ldr r1,[r0] (cycle_2) (cycle_3) (cycle_4)
add r2,r1,#3 (cycle_4) (cycle_5) (cycle_6)
mov r3,#0 (cycle_5) (cycle_6)
instructions
this book
Transaction Accurate
load_mem ldr r1,[r0] add r2,r1,#3 mov r3,#0
write_mem str r3,[r2] cmp r1,r3 ble 0x500 nop
transactions

Fig. 1.8 Abstraction levels for hardware-software codesign models

1. **Continuous time:** The lowest abstraction level describes operations as continuous actions. For example, electric networks can be described as systems of interacting differential equations. The voltages and currents in such electric networks can be found by solving these differential equations. The continuous-time model is a very detailed level, useful to analyze analog effects. However, this level of abstraction is not used to describe typical hardware-software systems.
2. **Discrete-event:** At the next abstraction level, activities are lumped together at discrete points in time called events. Those events can be irregularly spaced. For example, when the inputs of a digital combinatorial circuit change, the effect of those changes will ripple from input to output, changing the values at intermediate circuit nodes. Each change on a node can be thought of as an event: a (value, timestamp) tuple. Discrete-event simulation is commonly used to model digital hardware at low abstraction level. Discrete-event models avoid the complexity of continuous-time simulation, yet they capture relevant information

such as glitches and clock cycle edges. Discrete-event simulation is also used to model systems at high abstraction level, to simulate abstract events with irregular spacing in time. For example, discrete-event simulation can be used to model customer queues at a bank. In the context of hardware-software system design however, we will use discrete-event modeling to refer to digital hardware models at low abstraction level.

3. **Cycle-accurate:** Single-clock synchronous hardware circuits have the important property that all interesting things happen at regularly-spaced intervals, defined by the circuit clock period. This abstraction is important enough to merit its own abstraction level, and it is called cycle-accurate modeling. A cycle-accurate model does not capture propagation delays or glitches. All activities that fall in between two clock edges are concentrated at a single point in time. In a cycle-accurate model, activities happen either immediately (for combinatorial circuits for example), or else after an integral number of clock cycles (for sequential circuits). The cycle-accurate level is very important for hardware-software system modeling, and very often serves as the 'golden reference' for a hardware-software implementation. Cycle-accurate modeling will be used extensively throughout this book. Hardware models at cycle-accurate model are frequently called register-transfer level models, or RTL models.
4. **Instruction-accurate:** For the simulation of complex systems, the cycle-accurate level may still be too slow. For example, your laptop's processor executes several billion clock cycles per second. Clearly, the simulation of even a single second of real time operation will take a significant amount of machine resources. In case when a microprocessor needs to be simulated, it is convenient to express the activities within the model in terms of one microprocessor instruction. Each instruction lumps several cycles of processing together. Instruction-accurate simulators are used extensively to verify complex software systems, such as complete operating systems. Instruction-accurate simulators keep track of an instruction count, but not of a cycle count. Thus, unless you map instructions back to clock cycles, this abstraction level may not reveal the real-time performance of a model.
5. **Transaction-accurate:** For very complex systems, even instruction-accurate models may be too slow or require too much modeling effort. For these models, yet another abstraction level is introduced: the transaction-accurate level. In this type of model, the model is expressed in terms of the interactions between the components of a system. These interactions are called transactions. For example, one could model a system with a disk drive and a user application, and create a simulation that focuses on the commands exchanged between the disk drive and the user application. A transaction-accurate model allows considerable simplification of the internals of the disk drive and of the user application. Indeed, in between two transactions, millions of instructions can be lumped together and simulated as a single, atomic function call. Transaction-accurate models are important in the exploratory phases of a design. Transaction-accurate models enable a designer to define the overall characteristics of a design without going through the effort of developing a detailed model.

In summary, there are five abstraction levels that are commonly used for hardware-software modeling: transaction-accurate, instruction-accurate, cycle-accurate, event-driven and continuous-time. In this book, the emphasis is on cycle-accurate and instruction-accurate levels.

1.7 Concurrency and Parallelism

Concurrency and parallelism are terms that often occur in the context of hardware-software codesign. They have a different meaning. Concurrency is the ability to execute simultaneous operations because these operations are completely independent. Parallelism is the ability to execute simultaneous operations because the operations can run on different processors or circuit elements. Thus, concurrency relates to an application, while parallelism relates to the implementation of that application.

Hardware is always parallel. Software on the other hand can be sequential, concurrent or parallel. Sequential and concurrent software requires a single processor, parallel software requires multiple processors. The software running on your laptop (email, WWW, word processing, and so on) is concurrent. The software running on the 65536-processor IBM Blue Gene/L is parallel.

Making efficient use of parallelism (in the architecture) requires that you have an application which contains sufficient concurrency. There is a well-known law in supercomputing, called Amdahl's law, which states that the maximal speedup for any application that contains q % sequential code is 1/(q/100). For example, if your application is 33 % of its runtime executing like a sequential process, the maximal speedup is 3. It is easy to see why. Given sufficient hardware, the concurrent part of the application can complete arbitrarily fast, by implementing it on a parallel architecture. However, the sequential part cannot benefit from parallelism. If one third of the overall code is sequential, the speedup through a parallel implementaion cannot exceed three.

Surprisingly, even algorithms that seem sequential at first can be executed (and specified) in a parallel fashion. The following examples are discussed by Hillis and Steele. They describe the 'Connection Machine' (CM), a massively parallel processor. The CM contains a network of processors, each with their own local memory, and each processor in the network is connected to each other processor. The original CM machine contained 65536 processors, each of them with 4 Kbits of local memory. Interestingly, while the CM dates from the 1980s, multiprocessor architectures recently regained a lot of interest with the modern design community. Figure 1.9 illustrates an eight-node CM.

The question relevant to our discussion is: how hard is it to write programs for such a CM? Of course, you can write individual C programs for each node in the network, but that is not easy, nor is it very scalable. Remember that the original CM had 64 K nodes! Yet, as Hillis and Steele have shown, it is possible express algorithms in a concurrent fashion, such that they can map to a CM.

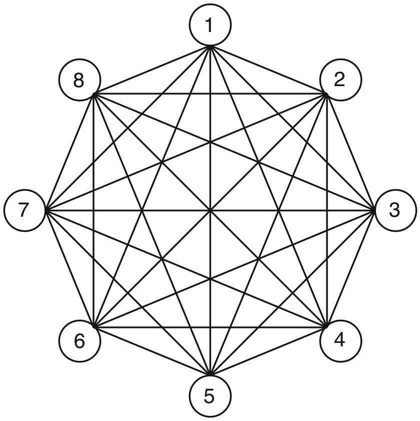


Fig. 1.9 Eight node connection machine

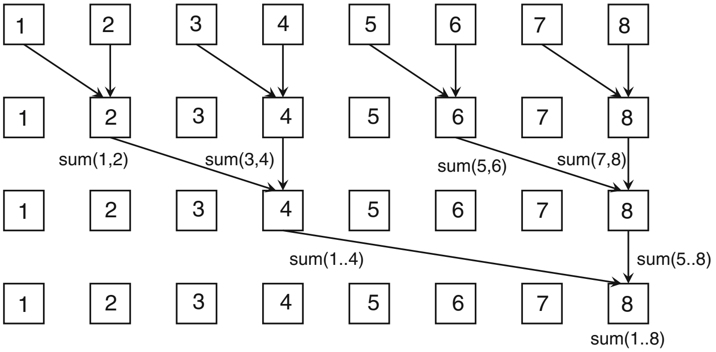


Fig. 1.10 Parallel sum

Consider taking the sum of an array of numbers, illustrated in Fig. 1.10. To take the sum, we distribute the array over the CM processors so that each processor holds one number. We can now take the sum over the entire array in $log2(n)$ steps (n being the number of processors) as follows. We perform 2^i parallel additions per time step, for i going from $log2(n-1)$ downto 0. For example, the sum of eight numbers can be computed in three time steps on a CM machine. In Fig. 1.10, time steps are taken vertically and each processor is drawn left to right. The communication activities between the processors are represented by means of arrows.

Compare the same algorithm running on a sequential processor. In that case, the array of numbers would be stored in a single memory and the processor needs to iterate over each element, requiring a minimum of eight time steps. You can also see that the parallel sum still wastes a lot of potential computing power. We have in total $3 * 8 = 24$ computation time-steps available, and we are only using seven of them. One extension of this algorithm is to evaluate all partial sums (i.e. the sum of the first two, three, four, etc numbers). A parallel algorithm that performs this in three time-steps, using 17 computation time-steps, is shown in Fig. 1.11.

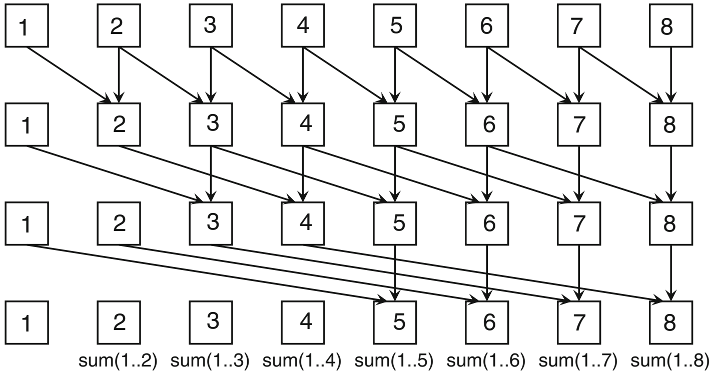


Fig. 1.11 Parallel partial sum

Concurrent specifications are important for hardware-software codesign. If you develop a concurrent specification, you will be able to make use of a parallel implementation. In contrast, if you restrict yourself to a sequential specification from the start, it will be much harder to make use of parallel hardware.

This shows a limitation of using C as a universal specification language. C is excellent to make a functional model of the application. However, to explore system architectures with a lot of parallelism, other specification models may be a better starting point. In the next chapter, we will discuss such a specification model.

1.8 Summary

We have been able to benefit from an ever growing selection of programmable components, and design using these components has become a challenge by itself. Hardware/software codesign is the collection of techniques that deals with design using programmable components. We have defined hardware/software codesign as the partitioning and design of an application in terms of fixed and flexible parts. The flexible parts run as programs on those programmable components. Traditional microprocessors are only one of the many options, and we briefly described other components including FPGA's, DSP's and ASIP's. Platform selection is the job of determining which programmable component (or combination of components) is the best choice for a given application. Application mapping is the effort of transforming a specification into a program. Platform programming is the effort of converting an application program into low-level instructions for each programmable component. We also discussed the modeling abstraction levels for hardware and software, and we highlighted cycle-based synchronous RTL for hardware, and single-thread C for software as the golden level for this book. Finally, we also made careful distinction between a parallel implementation and a concurrent program.

1.9 Further Reading

Many authors have pointed out the advantages of dedicated hardware solutions when it comes to Energy Efficiency. A comprehensive coverage of the problem can be found in (Rabaey 2009).

Figure 1.4 is based on results published between 2003 and 2008 by various authors including (Good and Benaissa 2007; Bogdanov et al. 2007; Satoh and Morioka 2003; Leander et al. 2007; Kaps 2008; Meiser et al. 2007; Ganesan et al. 2003; Karlof et al. 2004).

Further discussion on the driving forces that require chips to become programmable is found in (Keutzer et al. 2000). A nice and accessible discussion of what that means for the hardware designer is given in (Vahid 2003).

Hardware-software codesign, as a research area, is at least two decades old. Some the early works are collected in (Micheli et al. 2001), and a retrospective of the main research problems is found in (Wolf 2003). Conferences such as the *International Conference on Hardware/Software Codesign and System Synthesis* (CODES+ISSS) cover the latest evolutions in the field.

Hardware/software codesign, as an educational topic, is still evolving. A key challenge seems to be to find a good common ground to jointly discuss hardware and software. Interesting ideas are found, for example in (Madsen et al. 2002) and (Vahid 2007b).

Despite its age, the paper on data-parallel algorithms by Hillis and Steel is still a great read (Hillis and Steele 1986).

1.10 Problems

Problem 1.1. Use the World Wide Web to find a data sheet for the following components. What class of components are they (RISC/FPGA/DSP)? How does one write software for each of them? What tools are used to write that software?

- `TMS320DM6446`
- `EP4CE115`
- `SAM7S512`
- `ADSP-BF592`
- `XC5VFX100T`

Problem 1.2. Develop a sorting algorithm for an 8-node Connection Machine, which can handle up to 16 numbers. Show that an N-node Connection Machine can complete the sorting task in a time proportional to N.

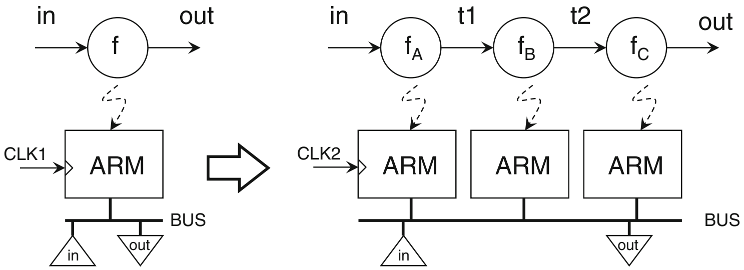


Fig. 1.12 Multiprocessor system for problem 1.3

Problem 1.3. A single-input, single-output program running on an ARM processor needs to be rewritten such that it will run on three parallel ARM processors. As shown in Fig. 1.12, each ARM has its own, independent data- and instruction memory. For this particular program, it turns out that it can be easily rewritten as a sequence of three functions fA, fB and fC which are also single-input, single-output. Each of these three functions can be executed on a separate ARM processor, so that we get an arrangement as shown below. The sub-functions fA, fB, and fC contain 40, 20, and 40 % respectively of the instructions of the original program. You can ignore the time needed for communication of variables (out, in, t1, and t2 are integers).

(a) Assume that all ARMs have the same clock frequency (CLK1 = CLK2). Find the maximal speedup that the parallel system offers over the single-ARM system. For example, a speedup of 2 would mean that the parallel system could process two times as much input data as the single-ARM system in the same amount of time.

(b) For the parallel system of three ARM described above, we can reduce the power consumption by reducing their clock frequency CLK and their operating voltage V. Assume that both these quantities scale linearly (i.e. Reducing the Voltage V by half implies that the clock frequency must be reduced by half as well). We will scale down the voltage/clock of the parallel system such that the scaled-down parallel system has the same performance as the original, single-ARM sequential system. Find the ratio of the power consumption of the original sequential system to the power consumption of the scaled-down, parallel system (i.e. find the power-savings factor of the parallel system). You only need to consider dynamic power consumption. Recall that Dynamic Power Consumption is proportional to the square of the voltage and proportional to the clock frequency.

Problem 1.4. Describe a possible implementation for each of the following C statements in hardware. You can assume that all variables are integers, and that each of them is stored in a register.

(a) `a = a + 1;`

(b) `if (a > 20) a = 20;`
(c) `while (a < 20) a = a + 1;`

Problem 1.5. The function in Listing 1.6 implements a CORDIC algorithm. It evaluates the cosine of a number with integer arithmentic and using only additions, subtractions, comparisons. The `angles[]` variable is an array of constants. Answer each of the following questions. Motivate your answer.

- Do you think it is possible to compute this function in hardware within 1,000 clock cycles?
- Do you think it is possible to compute this function in hardware within 1,000 ms?
- Do you think it is possible to compute this function in hardware within one clock cycle?
- Do you think it is possible to compute this function in hardware within 1 ms?

Listing 1.6 Listing for Problem 1.5.

```
int cordic_cos(int target) {
  int X, Y, T, current;
  unsigned step;
  X       = AG_CONST;
  Y       = 0;
  current = 0;
  for(step=0; step < 20; step++) {
    if (target > current) {
      T          =  X - (Y >> step);
      Y          = (X >> step) + Y;
      X          = T;
      current   += angles[step];
    } else {
      T          = X + (Y >> step);
      Y          = -(X >> step) + Y;
      X          = T;
      current   -= angles[step];
    }
  }
  return X;
}
```

Problem 1.6. Listing 1.7 shows a simplified version of the CORDIC algorithm in C. After compiling this code to an Intel architecture (x86), the assembly code from Listing 1.8 is generated. In this listing, arguments starting with the % sign are registers. Study the C and the assembly code, and answer the following questions.

- What register is used to store the variable `current`?
- What assembly instruction corresponds to the comparison of the variables `target` and `current` in the C code?
- What register is used to store the loop counter `step`?

Listing 1.7 C Listing for Problem 1.6.

```
extern int angles[20];

```

```
int cordic(int target) {
  int current;
  unsigned step;
  current = 0;
  for(step=0; step < 20; step++) {
    if (target > current) {
      current   += angles[step];
    } else {
      current   -= angles[step];
    }
  }
  return current;
}
```

Listing 1.8 Assembly Listing for Problem 1.6.

```
cordic:
        pushl   %ebp
        xorl    %edx, %edx
        movl    %esp, %ebp
        xorl    %eax, %eax
        movl    8(%ebp), %ecx
        jmp    .L4
.L9:
        addl    angles(,%edx,4), %eax
        addl    $1, %edx
        cmpl    $20, %edx
        je     .L8
.L4:
        cmpl    %eax, %ecx
        jg     .L9
        subl    angles(,%edx,4), %eax
        addl    $1, %edx
        cmpl    $20, %edx
        jne    .L4
.L8:
        popl    %ebp
        ret
```

Chapter 2
Data Flow Modeling and Transformation

2.1 Introducing Data Flow Graphs

By nature, hardware is parallel and software is sequential. As a result, software models (C programs) are not very well suited to capture hardware implementations, and vice versa, hardware models (RTL programs) are not a good abstraction to describe software. However, designers frequently encounter situations for which a given design may use either hardware or software as a target. Trying to do both (writing a full C program *as well as* a full hardware design) is not an option; it requires the designer to work twice as hard. An alternative is to use a high-level model, which enables the designer to express a design without committing to a hardware or a software implementation. Using a high-level model, the designer can gain further insight into the specification, and decide on the right path for implementation.

In the design of signal processing systems, the need for modeling is well known. Signal processing engineers describe complex systems, such as digital radios and radar processing units, using *block diagrams*. A block diagram is a high-level representation of the target system as a collection of smaller functions. A block diagram does not specify if a component should be implemented as hardware or software; it only expresses the operations performed on data signals. We are specifically interested in *digital* signal processing systems. Such systems represent signals as streams of discrete samples rather than continuous waveforms.

Figure 2.1a shows the block diagram for a simple digital signal processing system. It's a pulse-amplitude modulation (PAM) system, and it is used to transmit digital information over bandwidth-limited channels. A PAM signal is created from binary data in two steps. First, each word in the file needs to be mapped to PAM symbols, an alphabet of pulses of different heights. An entire file of words will thus be converted to a stream of PAM symbols or pulses. Next, the stream of pulses needs to be converted to a smooth shape using pulse-shaping. Pulse-shaping ensures that the bandwidth of the resulting signal does not exceed the PAM symbol rate. For example, if a window of 1,000 Hz transmission bandwidth is available, then we

P.R. Schaumont, *A Practical Introduction to Hardware/Software Codesign*,

DOI 10.1007/978-1-4614-3737-6_2, © Springer Science+Business Media New York 2013

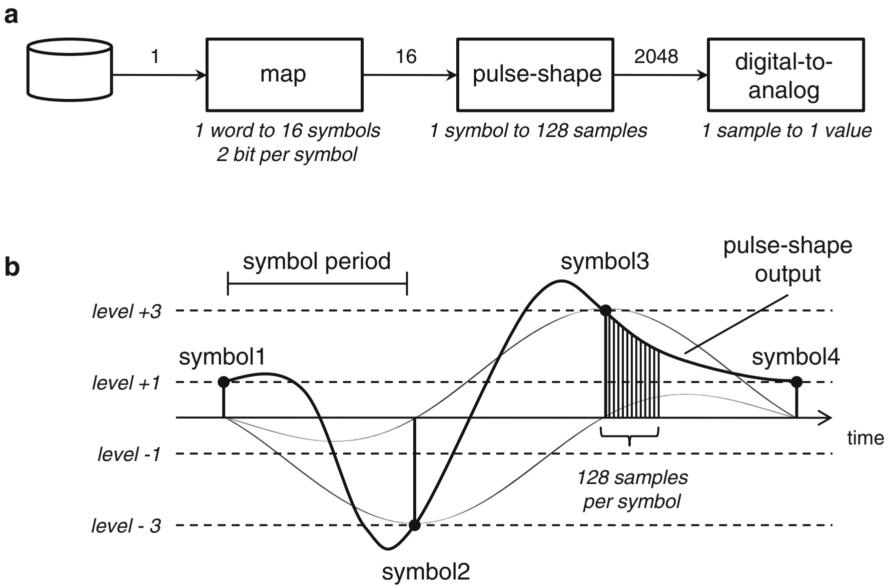


Fig. 2.1 (**a**) Pulse-amplitude modulation system. (**b**) Operation of the pulse-shaping unit

can transmit 1,000 PAM symbols per second. In a digital signal processing system, a *smooth* curve is achieved by *oversampling*: calculating many closely-spaced discrete samples. The output of the pulse-shaping unit produces many samples for each input symbol pulse, but it is still a stream of discrete samples. The final module in the block diagram is the digital-to-analog converter, which will convert the stream of discrete samples into a continuous signal.

Figure 2.1a shows a PAM-4 system, which uses four different symbols. Since there are four different symbols, each PAM symbol holds 2 bits of source information. A 32-bit word from a data source is encoded with 16 PAM-4 symbols. The first block in the PAM transmission system makes the conversion of a single word to a sequence of 16 PAM-4 symbols. Figure 2.1b shows that each PAM-4 symbol is mapped to a pulse with four possible signal levels: $\{-3,-1,1,3\}$. Once the PAM-4 signals are available, they are shaped to a smooth curve using a pulse-shape filter. The input of this filter is a stream of symbol pulses, while the output is a stream of samples at a much higher rate. In this case, we generate 128 samples for each symbol.

Figure 2.1b illustrates the operation of the pulse-shape filter. The smooth curve at the output of the pulse-shape filter connects the top of each pulse. This is achieved by an interpolation technique, which extends the influence of a single symbol pulse over many symbol periods. The figure illustrates two such interpolation curves, one for `symbol2` and one for `symbol3`. The pulse-shape filter will produce 128 samples for each symbol entered into the pulse-shape filter.

Listing 2.1 C example

```
extern int read_from_file();
extern int map_to_symbol(int, int);
extern int pulse_shape(int, int);
extern void send_to_da(int);

int main() {
  int word, symbol, sample;
  int i, j;
  while (1) {
    word = read_from_file();
    for (i=0; i<16; i++) {
       symbol = map_to_symbol(word, i);
       for (j=0; j<128; j++)
         sample = pulse_shape(symbol, j);
         send_to_da(sample);
    }
 }
}
```

Now let's consider the construction of a simulation model for this system. We focus on capturing its functionality, and start with a C program as shown in Listing 2.1. We will ignore the implementation details of the function calls for the time being, and only focus on the overall structure of the program.

The program in Listing 2.1 is fine as a system simulation. However, as a model for the implementation, this C program is too strict, since it enforces *sequential* execution of all functions. If we observe Fig. 2.1a carefully, we can see that the block diagram does not require a sequential execution of the symbol mapping function and the pulse shaping function. The block diagram *only* specifies the flow of data in the system, but not the execution order of the functions. The distinction is subtle but important. For example, in Fig. 2.1a, it is possible that the `map` module and the `pulse-shape` module work in parallel, each on a different symbol. In Listing 2.1 on the other hand, the `map_to_symbol()` function and the `pulse_shape()` function will *always* execute sequentially. In hardware-software codesign, the implementation target could be either parallel or else sequential. The program in Listing 2.1 favors a sequential implementation, but it does not encourage a parallel implementation in the same manner as a block diagram.

This illustrates how the selection of a modeling technique can constrain the solutions that may be achieved starting from that model. In general, building a sequential implementation for a parallel model (such as a block diagram) is much easier than the opposite – building a parallel implementation from a sequential model. Therefore, we favor modeling styles that enable a designer to express parallel activities at the highest level of abstraction.

In this chapter we will discuss a modeling technique, called Data Flow, which achieves the objective of a parallel model. Data Flow models closely resemble

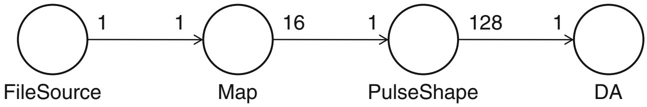


Fig. 2.2 Data flow model for the pulse-amplitude modulation System

block diagrams. The PAM-4 system, as a Data Flow model, is shown in Fig. 2.2. In this case, the different functions of the system are mapped as individual entities or *actors* such as `FileSource`, `Map`, `PulseShape` and `DA`. These actors are linked through communication channels or *queues*. The inputs and outputs of each actor are marked with the relative rate of communications. For example, there are 16 samples produced by `Map` for each input sample. Each actor is an *independent unit*, continuously checking its input for the availability of data. As soon as data appears, each actor will calculate the corresponding output, passing the result to the next actor in the chain. In the remainder of this chapter, we will discuss the precise construction details of data flow diagrams. For now, we only point out the major differences of this modeling style compared to modeling in C.

- A strong point of Data Flow models, and the main reason why signal processing engineers love to use them, is that a Data Flow model is a *concurrent* model. Indeed, the actors in Fig. 2.2 operate and execute as individual concurrent entities. A concurrent model can be mapped to a parallel or a sequential implementation, and so they can target hardware as well as software.
- Data Flow models are distributed, and there is no need for a central controller or 'conductor' in the system to keep the individual system components in pace. In Fig. 2.2, there is no central controller that tells the actors when to operate; each actor can determine for itself when it's time to work.
- Data Flow models are modular. We can develop a design library of data flow components and then use that library in a plug-and-play fashion to construct data flow systems.
- Data Flow models can be analyzed. Certain properties, such as their ability to avoid deadlock, can be determined directly from the design. Deadlock is a condition sometimes experienced by real-time computing systems, in which the system becomes unresponsive. The ability to analyze the behavior of models at high abstraction level is an important advantage. C programs, for example, do not offer such convenience. In fact, a C designer typically determines the correctness of a C program by running it, rather than analyzing it!

Data Flow has been around for a surprisingly long time, yet it has been largely overshadowed by the stored-program (Von Neumann) computing model. Data Flow concepts have been explored since the early 1960s. By 1974, Jack Dennis had developed a language for modeling data flow, and described data flow using graphs, similar to our discussion in this chapter. In the 1970s and 1980s, an active research community was building not only data flow-inspired programming languages and

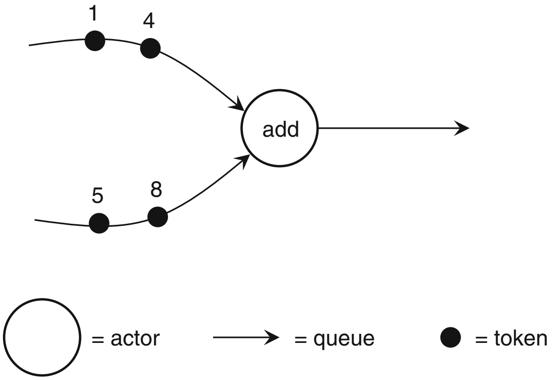


Fig. 2.3 Data flow model of an addition

tools, but also computer architectures that implement data flow computing models. The work from Arvind was seminal in this area, resulting in several different computer architectures and tools (see Further Reading at the end of this chapter).

Today, data flow remains very popular to describe signal processing systems. For example, commercial tools such as Simulink® are based on the ideas of data flow. A interesting example of an academic environment is the Ptolemy project at UC Berkeley (http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm). The Ptolemy design environment can be used for many different types of system specification, including data flow. The examples on the website can be run inside of a web browser as Java applets.

In the following sections, we will consider the elements that make up a data flow model. We will next discuss a particular class of data flow models called Synchronous Data Flow Graphs (SDF). We will show how SDF graphs can be formally analyzed. Later, we will discuss transformations on SDF graphs, and show how transformations can lead to better, faster implementations.

2.1.1 Tokens, Actors, and Queues

Figure 2.3 shows the data flow model of a simple addition. This model contains the following elements.

- *Actors* contain the actual operations. Actors have a bounded behavior (meaning that they have a precise beginning and ending), and they iterate that behavior from start to completion. One such iteration is called an actor firing. In the example above, each actor firing would perform a single addition.
- *Tokens* carry information from one actor to the other. A token has a value, such as ‘1’, ‘4’, ‘5’ and ‘8’ in Fig. 2.3.

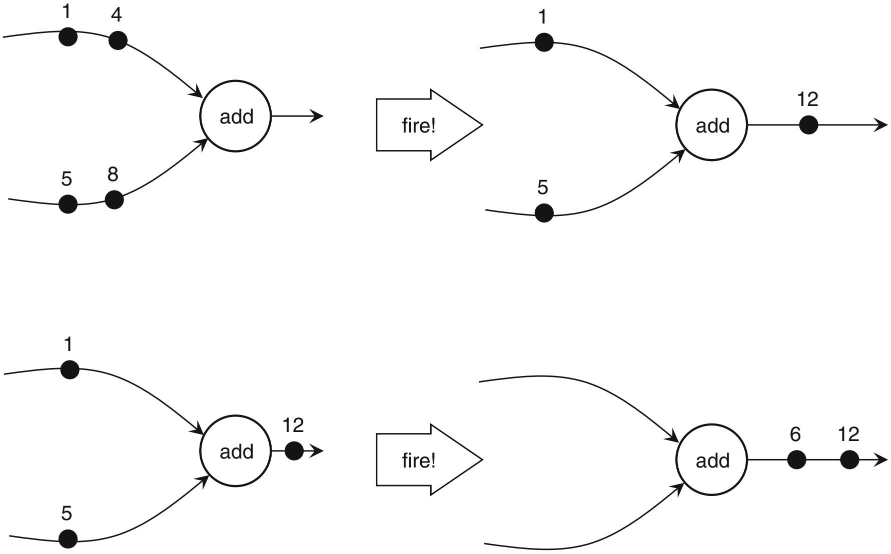


Fig. 2.4 The result of two firings of the add actor, each resulting in a different marking

- *Queues* are unidirectional communication links that transport tokens from one actor to the other. Data Flow queues have an infinite amount of storage, so that tokens will never get lost in a queue. Data Flow queues are first-in first-out. In Fig. 2.4, there are two tokens in the upper queue, one with value '1' and one with value '4'. The '4' token was entered first into the queue, the '1' token was entered after that. When the 'add' actor will read a token from that queue, the actor will first read the token with value '4' and next the token with value '1'.

When a data flow model executes, the actors will read tokens from their input queues, read the value of these tokens and calculate the corresponding output value, and generate new tokens on their output queues. Each single execution of an actor is called the firing of that actor. Data flow execution then is expressed as a sequence of (possibly concurrent) actor firings.

Conceptually, data flow models are untimed. The firing of an actor happens instantaneously, although any real implementation of an actor does require a finite amount of time. *Untimed* does not mean zero time; it only means that time is irrelevant for data flow models. Indeed, in data flow, the execution is guided by the presence of data, not by a program counter or by a clock signal. An actor will never fire if there's no input data, but instead it will wait until sufficient data is available at its inputs.

A graph with tokens distributed over queues is called a *marking* of a data flow model. When a data flow model executes, the entire graph goes through a series of markings that drive data from the inputs of the data flow model to

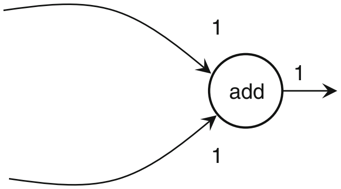


Fig. 2.5 Data flow actor with production/consumption rates

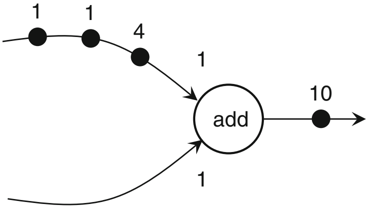


Fig. 2.6 Can this model do any useful computations?

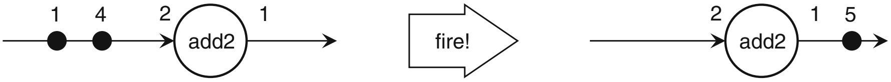


Fig. 2.7 Example of a multi-rate data flow model

the outputs. Each marking corresponds to a different state of the system, and the execution of a data flow model is defined by a sequence of markings. To an external observer, the marking (i.e., the distribution of the tokens on the queues) is the only observable state in the system. This is a crucial observation! It implies that an actor cannot use internal state variables that would affect the execution of the system, and thus the marking sequence. If we want to model system state that would affect the execution, we need to express it using tokens.

2.1.2 *Firing Rates, Firing Rules, and Schedules*

When should an actor fire? The *firing rule* describes the necessary and sufficient conditions for an actor to fire. Simple actors such as the add actor can fire when there is a single token on each of its queues. A firing rule thus involves testing the number of tokens on each of its input queues. The required number of tokens can be annotated to the actor input. Similarly, the amount of tokens that an actor produces per firing can be annotated to the actor output. These numbers are called the token consumption rate (at the actor inputs) and token production rate (at the actor outputs). The production/consumption rates of the add actor could be written such as shown in Figs. 2.5 or 2.6.

Data Flow actors may consume or produce more than one token per actor firing. Such models are called multirate data flow models. For example, the actor in Fig. 2.7

has a consumption rate of 2 and a production rate of 1. It will consume two tokens per firing from its input, add them together, and produce an output token as result.

2.1.3 Synchronous Data Flow (SDF) Graphs

When the number of tokens consumed/produced per actor firing is a fixed and constant value, the resulting class of systems is called synchronous data flow or SDF. The term synchronous means that the token production and consumption rates are known, fixed numbers. SDF semantics are not universal. For example, not every C program can be translated to an equivalent SDF graph. Data-dependent execution cannot be expressed as an SDF graph: data-dependent execution implies that actor firing is defined not only by the presence of tokens, but also by their value.

Nevertheless, SDF graphs have a significant advantage: their properties can be analyzed mathematically. The structure of an SDF graph, and the production/consumption rates of tokens on the actors, determines if a feasible schedule of actor firings is possible. We will demonstrate a technique that can analyze an SDF graph, and derive such a feasible schedule, if it exists.

2.1.4 SDF Graphs are Determinate

Assuming that each actor implements a deterministic function, then the entire SDF execution is determinate. This means that results, computed using an SDF graph, will always be the same. This property holds regardless of the firing order (or schedule) of the actors. Figure 2.8 illustrates this property. This graph contains actors with unit production/consumption rates. One actor adds tokens, the second actor increments the value of tokens. As we start firing actors, tokens are transported through the graph. After the first firing, an interesting situation occurs: both the add actor as well as the plus1 actor can fire. A first case, shown on the left of Fig. 2.8, assumes that the plus1 actor fires first. A second case, shown on the right of Fig. 2.8, assumes that the add actor fires first. However, regardless what path is taken, the graph marking eventually converges to the result shown at the bottom.

Why is this property so important? Assume that the add actor and the plus1 actor execute on two different processors, a slow one and a fast one. Depending upon which actor runs on the fast processor, the SDF execution will follow the left marking or else the right marking of the figure. Since SDF graphs are determinate, it doesn't matter which processor executes what actor: the results will be always the same. In other words, an SDF system will work as specified, regardless of the technology used to implement it. Of course, actors must be completely and correctly implemented, firing rule and all. Determinate behavior is vital in many embedded applications, especially in applications that involve risk.

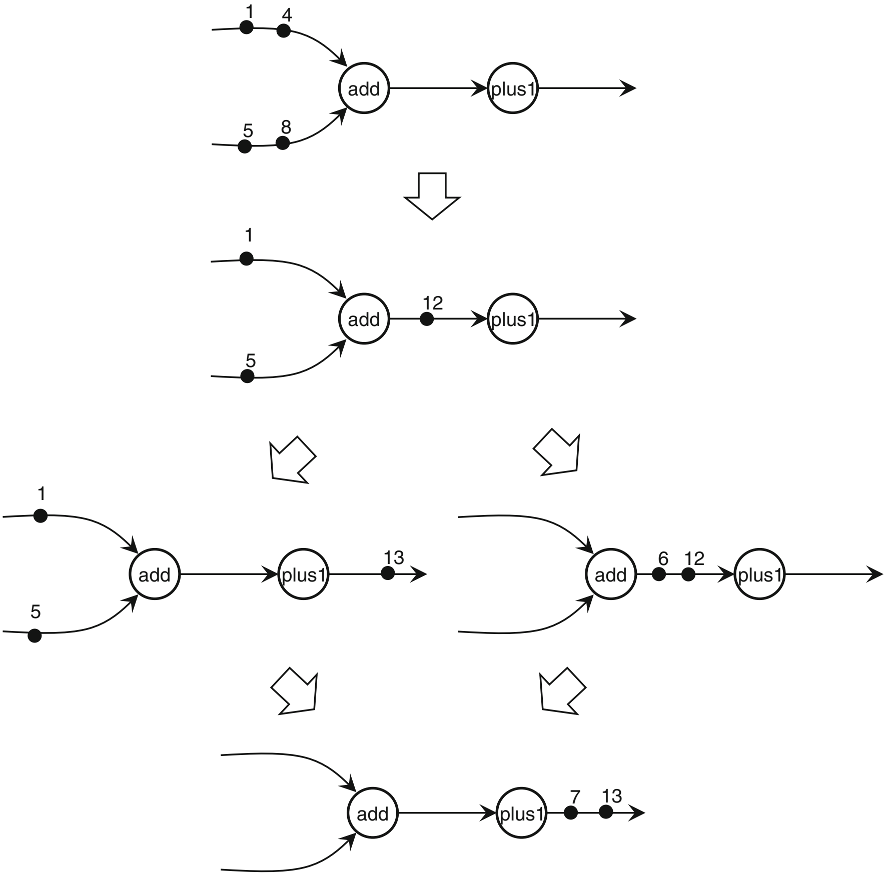


Fig. 2.8 SDF graphs are determinate

2.2 Analyzing Synchronous Data Flow Graphs

An admissible schedule for an SDF graph is one that can run forever without causing deadlock and without overflowing any of the communication queues. The term *unbounded execution* is used to indicate that a model runs forever; the term *bounded buffer* is used to indicate that no communication queue needs infinite depth. A deadlock situation occurs when the SDF graph ends up in a marking in which it is no longer possible to fire any actor.

Figure 2.9 shows two SDF graphs where these two problems are apparent. Which graph will deadlock, and which graph will result in an infinite amount of tokens?

Given an arbitrary SDF graph, it is possible to test if it is free from deadlock, and if it only needs bounded storage under unbounded execution. The nice thing about this test is that we don't need to use any simulation; the test can be done using basic

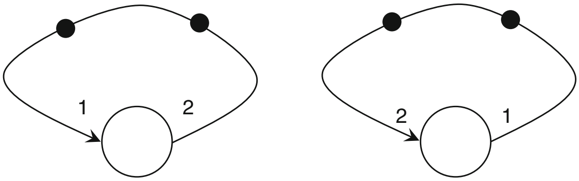


Fig. 2.9 Which SDF graph will deadlock, and which is unstable?

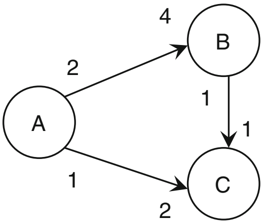


Fig. 2.10 Example SDF graph for PASS construction

matrix operations. We will study the method of Lee to create so-called Periodic Admissible Schedules (PASS). A PASS is defined as follows.

- A schedule is the order in which the actors must fire.
- An admissible schedule is a firing order that will not cause deadlock and that yields bounded storage.
- Finally, a periodic admissible schedule is a schedule that is suitable for unbounded execution, because it is periodic (meaning that after some time, the same marking sequence will repeat). We will consider Periodic Admissible Sequential Schedules, or PASSs for short. Such a sequential schedule requires only a single actor to fire at a time. A PASS would be used, for example, to execute an SDF model on a microprocessor.

2.2.1 Deriving Periodic Admissible Sequential Schedules

We can create a PASS for an SDF graph (and test if one exists) with the following four steps.

1. Create the topology matrix G of the SDF graph;
2. Verify the rank of the matrix to be one less than the number of nodes in the graph;
3. Determine a firing vector;
4. Try firing each actor in a round robin fashion, until it reaches the firing count as specified in the firing vector.

We will demonstrate each of these steps using the example of the three-node SDF graph shown in Fig. 2.10.

Step 1. Create a topology matrix for this graph. This topology matrix has as many rows as graph edges (FIFO queues) and as many columns as graph nodes. The entry (i, j) of this matrix will be positive if the node j produces tokens into graph edge i. The entry (i, j) will be negative if the node j consumes tokens from graph edge i. For the above graph, we thus can create the following topology matrix. Note that G does not have to be square – it depends on the amount of queues and actors in the system.

$$G = \begin{bmatrix} 2 & -4 & 0 \\ 1 & 0 & -2 \\ 0 & 1 & -1 \end{bmatrix} \begin{array}{l} \leftarrow edge(A,B) \\ \leftarrow edge(A,C) \\ \leftarrow edge(B,C) \end{array} \tag{2.1}$$

Step 2. The condition for a PASS to exist is that the rank of G has to be one less than the number of nodes in the graph. The proof of this theorem is beyond the scope of this book, but can be consulted in (Lee and Messerschmitt 1987). The rank of a matrix is the number of independent equations in G. It can be verified that there are only two independent equations in G. For example, multiply the first column with -2 and the second column with -1, and add those two together to find the third column. Since there are three nodes in the graph and the rank of G is 2, a PASS is possible.

Step 2 verifies that tokens cannot accumulate on any of the edges of the graph. We can find the resulting number of tokens by choosing a firing vector and making a matrix multiplication. For example, assume that A fires two times, and B and C each fire zero times. This yields the following firing vector:

$$q = \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} \tag{2.2}$$

The residual tokens left on the edges after these firings are two tokens on $edge(A,B)$ and a token on $edge(A,C)$:

$$b = Gq = \begin{bmatrix} 2 & -4 & 0 \\ 1 & 0 & -2 \\ 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} \tag{2.3}$$

Step 3. Determine a periodic firing vector. The firing vector indicated above is not a good choice to obtain a PASS: each time this firing vector executes, it adds three tokens to the system. Instead, we are interested in firing vectors that leave no additional tokens on the queues. In other words, the result must equal the zero-vector.

$$Gq_{PASS} = 0 \tag{2.4}$$

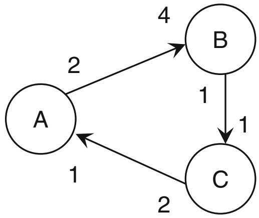


Fig. 2.11 A deadlocked graph

Since the rank of G is less than the number of nodes, this system has an infinite number of solutions. Intuitively, this is what we should expect. Assume a firing vector (a,b,c) would be a solution that can yield a PASS, then also $(2a,2b,2c)$ will be a solution, and so is $(3a,3b,3c)$, and so on. You just need to find the simplest one. One possible solution that yields a PASS is to fire A twice, and B and C each once:

$$q_{PASS} = \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} \tag{2.5}$$

The existence of a PASS firing vector does not guarantee that a PASS will also exist. For example, just by changing the direction of the (A,C) edge, you would still find the same q_{PASS}, but the resulting graph is deadlocked since all nodes are waiting for each other. Therefore, there is still a fourth step: construction of a valid PASS.

Step 4. Construct a PASS. We now try to fire each node up to the number of times specified in q_{PASS}. Each node which has the adequate number of tokens on its input queues will fire when tried. If we find that we can fire no more nodes, and the firing count of each node is less than the number specified in q_{PASS}, the resulting graph is deadlocked.

We apply this on the original graph and using the firing vector $(A = 2, B = 1, C = 1)$. First we try to fire A, which leaves two tokens on (A,B) and one on (A,C). Next, we try to fire B – which has insufficient tokens to fire. We also try to fire C but again have insufficient tokens. This completes our first round through – A has fired already one time. In the second round, we can fire A again (since it has fired less than two times), followed by B and C. At the end of the second round, all nodes have reached the firing count specified in the PASS firing vector, and the algorithm completes. The PASS we are looking for is (A,A,B,C).

The same algorithm, when applied to the deadlocked graph in Fig. 2.11, will immediately abort after the first iteration, because no node was able to fire.

Note that the determinate property of SDF graphs implies that we can try to fire actors in any order of our choosing. So, instead of trying the order (A,B,C) we can also try (B,C,A). In some SDF graphs (but not in the one discussed above), this may lead to additional PASS solutions.

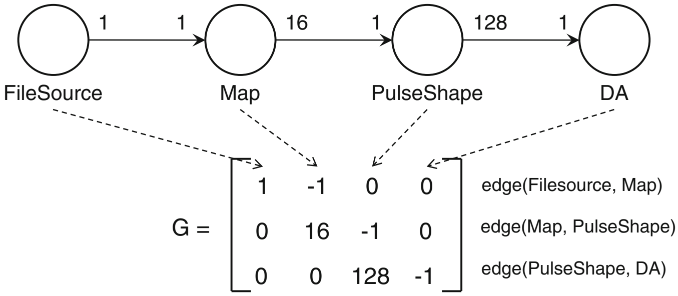


Fig. 2.12 Topology matrix for the PAM-4 system

2.2.2 *Example: Deriving a PASS for the PAM-4 System*

At the start of this chapter, we discussed the design of a digital pulse-amplitude modulation system for the generation of PAM4 signals. The system, shown in Fig. 2.12, is modeled with four data flow actors: a word source, a symbol mapper, a pulse shaper, and a digital-to-analog converter. The system is a multi-rate data flow system. For every byte modulated, $16.128 = 2048$ output samples are generated.

Our objective is to derive a PASS for this data flow system. The first step is to derive the topology matrix G. The matrix has three rows and four columns, corresponding to the three queues and four actors in the system. The second step in deriving the PASS is to verify that the rank of this topology matrix equals the number of data flow actors minus one. It's easy to demonstrate that G indeed consists of three independent equations: no row can be created as a linear combination of two others. Hence, we confirm that the condition for a PASS to exist is fulfilled. Third, we have to derive a feasible firing vector for this system. This firing vector, q_{PASS}, needs to yield a zero-vector when multiplied with the topology matrix. The solution for q_{PASS} is to fire the `Filesource` and `Map` actors one time, the `PulseShape` actor 16 times, and the `DA` actor 2,048 times.

$$G.q_{PASS} = \begin{bmatrix} +1 & -1 & 0 & 0 \\ 0 & 16 & -1 & 0 \\ 0 & 0 & 128 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 16 \\ 2{,}048 \end{bmatrix} = 0 \tag{2.6}$$

The final step is to derive a concrete schedule with the derived firing rates. We discuss two alternative solutions.

- By inspection of the graph in Fig. 2.12, we conclude that firing the actors from left to right according to their q_{PASS} firing rate will result in a feasible solution that ensures sufficient tokens in each queue. Thus, we start by firing `FileSource` once, followed by `Map`, followed by 16 firings of `PulseShape`, and finally 2,048 firings of `DA`. This particular schedule will require a FIFO of 16 positions

for the middle queue in the system, and a FIFO of 2,048 positions for the rightmost queue in the system.
- An alternative schedule is to start firing `FileSource` once, followed by `Map`. Next, the following sequence is repeated 16 times: fire `PulseShape` once, followed by 128 firings of `DA`. The end result of this alternate schedule is identical to the first schedule. However, the amount of intermediate storage is much lower: the rightmost queue in the system will use at most 128 positions.

Hence, we conclude that the concrete schedule in a PASS affects the amount of storage used by the communication queues. Deriving an optimal interleaving of the firings is a complex problem in itself (See Further Reading).

This completes our discussion of PASS. SDF has very powerful properties, which enable a designer to predict critical system behavior such as determinism, deadlock, and storage requirements. Yet, SDF is not a universal specification mechanism; it is not a good replacement for any type of application. The next part will further elaborate on the difficulty of implementing control-oriented systems using data flow modeling.

2.3 Control Flow Modeling and the Limitations of Data Flow Models

SDF systems are distributed, data-driven systems. They execute whenever there is data to process, and remain idle when there is nothing to do. However, SDF seems to have trouble to model control-related aspects. Control appears in many different forms in system design, for example:

- **Stopping and restarting**. An SDF model never terminates; it just keeps running. Stopping and re-starting is a control-flow property that cannot be addressed well with SDF graphs.
- **Mode-switching**. When a cell-phone switches from one standard to the other, the processing (which may be modeled as an SDF graph) needs to be reconfigured. However, the topology of an SDF graph is fixed and cannot be modified at runtime.
- **Exceptions**. When catastrophic events happen, processing may suddenly need to be altered. SDF cannot model exceptions that affect the entire graph topology. For example, once a token enters a queue, the only way of removing it is to read the token out of the queue. It is not possible to suddenly flush the queue on a global, exceptional condition.
- **Run-time Conditions**. A simple if-then-else statement (choice between two activities depending on an external condition) is troublesome for SDF. An SDF node cannot simply 'disappear' or become inactive – it is always there. Moreover, we cannot generate conditional tokens, as this would violate SDF rules which require fixed production/consumption rates. Thus, SDF cannot model conditional execution such as required for if-then-else statements.

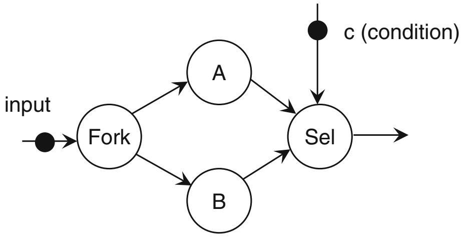


Fig. 2.13 Emulating if-then-else conditions in SDF

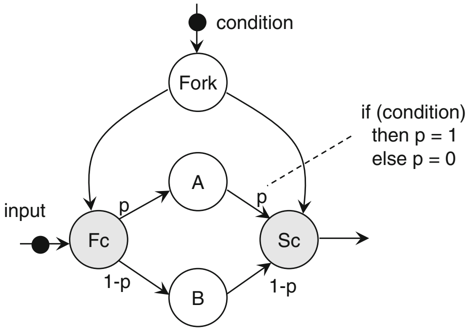


Fig. 2.14 Implementing if-then-else using Boolean Data Flow

There are two solutions to the problem of control flow modeling in SDF. The first one is to emulate control flow using SDF, at the cost of some modeling overhead. The second one is to extend the semantics of SDF. We give a short example of each strategy.

2.3.1 Emulating Control Flow with SDF Semantics

Figure 2.13 shows an example of an if-then-else statement, SDF-style. Each of the actors in the above graph are SDF actors. The last one is a selector-actor, which will transmit either the A or B input to the output depending on the value of the input condition. Note that when Sel fires, it will consume a token from each input, so both A and B have to run for each input token. This is thus not really an if-then-else in the same sense as in C programming. The approach taken by this graph is to implement both the if-leg and the else-leg and afterwards transmit only the required result. This approach may work when there is sufficient parallelism available. For example, in hardware design, the equivalent of the Sel node would be a multiplexer.

2.3.2 Extending SDF Semantics

Researchers have also proposed extensions on SDF models. One of these extensions was proposed by Joseph Buck, and is called BDF (Boolean Data Flow) (Lee and Seshia 2011). The idea of BDF is to make the production and consumption-rate of a token dependent on the value of an external control token. In Fig. 2.14, the condition token is distributed over a fork to a conditional fork and a conditional merge node. These conditional nodes are BDF.

- The conditional fork will fire when there is an input token and a condition token. Depending on the value of the condition token, it will produce an output on the upper or the lower queue. We used a conditional production rate p to indicate this. It is impossible to determine the value of p upfront – this can only be done at runtime.
- The conditional merge will fire when there is a condition token. If there is, it will accept a token from the upper input or the lower input, depending on the value of the condition. Again, we need to introduce a conditional consumption rate.

The overall effect is that either node A or else node B will fire, but never both. Even a simple extension on SDF already takes jeopardizes the basic properties which we have enumerated above. For example, a consequence of using BDF instead of SDF is that we now have data flow graphs that are only conditionally admissible. Moreover, the topology matrix now will include symbolic values (p), and becomes harder to analyze. For five conditions, we would have to either analyze a matrix with five symbols, or else enumerate all possible condition values and analyze 32 different matrices (each of which can have a different series of markings). In other words, while BDF can help solving some of practical cases of control, it quickly becomes impractical for analysis.

Besides BDF, researchers have also proposed other flavors of control-oriented data flow models, such as Dynamic Data Flow (DDF) which allows variable production and consumption rates, and Cyclo-Static Data Flow (CSDF) which allows a fixed, iterative variation on production and consumption rates. All of these extensions break down the elegance of SDF graphs to some extent. SDF remains a very popular technique for Digital Signal Processing applications. But the use of BDF, DDF and the like has been limited.

2.4 Adding Time and Resources

So far, we have treated data flow graphs as untimed: the analysis of data flow graphs was based only on their marking (distribution of tokens), and not on the time needed to complete a computation. However, we can also use the data flow model to do performance analysis. By introducing a minimal resource model (actor execution time and bounded FIFO queues), we can analyze the system performance of a data flow graph. Furthermore, we can analyze the effect of performance-enhancing transformations on the data flow graph.

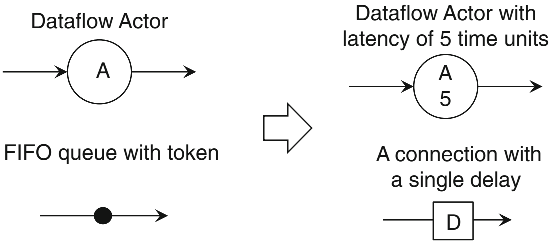


Fig. 2.15 Enhancing the SDF model with resources: execution time for actors, and delays for FIFO queues

2.4.1 Real-Time Constraints and Input/Output Sample Rate

A data flow graph is a model for a repeating activity. For example, the PAM-4 modulation system described in this chapter's introduction transforms an infinite stream of input samples (words) into an infinite stream of output samples. The model shows how one single sample is processed, and the streaming character is implicit.

The input sample rate is the time period between two adjacent input-samples from the stream. The sample rate typically depends on the application. CD audio samples, for example, are generated at 44,100 samples per second. The input sample rate thus sets a design constraint for the performance of the data flow system: it specifies how quickly the data flow graph must be computed in order to achieve real-time performance. A similar argument can be made for the output sample rate. In either case, when there's a sample-rate involved with the input or the output of a data flow graph, there is also a real-time constraint on the computation speed for the data flow graph.

We define the input *throughput* as the amount of input samples per second. Similarly, we define the output throughput as the amount of output samples per second. The *latency* is the time required to process a single token from input to output. Throughput and latency are two important system constraints.

2.4.2 Data Flow Resource Model

In this section, we are interested in performance analysis of data flow graphs. This requires the introduction of time and resources. Figure 2.15 summarizes the two enhancements needed.

- Every actor is decorated with an execution latency. This is the time needed by the actor to complete a computation. We assume that the actor's internal program requires all inputs to be available at the start of the execution. Similarly, we assume that the actor's internal program produces all outputs simultaneously after the actor latency. Latency is expressed in time units, and depending on the implementation target, a suitable unit can be chosen – clock cycles, nanoseconds, and so on.

- Every FIFO queue is replaced with a communication channel with a fixed number of delays. A delay is a storage location that can hold one token. A single delay can hold a token for a single actor execution. Replacing a FIFO queue with delay storage locations also means that the actor firing rule needs to be changed. Instead of testing the number of elements in a FIFO queue, the actor will now test for the presence of a token in a delay storage location.

The use of a data flow resource model enables us to analyze how fast a data flow graph will run. Figure 2.16 shows three single-rate data flow graphs, made with two actors A and B. Actor A needs five units of latency, while actor B requires three units of latency. This data flow graph also has an input and an output connection, through which the system can accept a stream of input samples, and deliver a stream of output samples. For our analysis, we do not define an input or output sample rate. Instead, we are interested to find out how fast these data flow graphs can run.

The easiest way to analyze this graph is to evaluate the latency of samples as they are processed through the data flow graph. Eventually, this analysis yields the time instants when the graph reads from the system input, or writes to the system output.

In the graphs of Fig. 2.16a, b, there is a single delay element in the loop. Data input/output is defined by the combined execution time of actor A and actor B. The time stamps for data production for the upper graph and the middle graph are different because of the position of the delay element: for the middle graph, actor B can start at system initialization time, since a token is available from the delay element.

In the graph of Fig. 2.16c, there are two delay elements in the loop. This enables actor A and actor B to operate in parallel. The performance of the overall system is defined by the slowest actor A. Even though actor B completes in three time units, it needs to wait for the next available input until actor A has updated the delay element at its output.

Hence, we conclude that the upper two graphs have a throughput of 1 sample per 8 time units, and that the lower graph has a throughput of 1 sample per 5 time units.

2.4.3 Limits on Throughput

The example in the previous section illustrates that the number of delays, and their distribution over the graph, affects the throughput of the data flow system. A second factor that affects the latency are the feedback links or loops in the data flow graph. Together, loops and delays determine an upper bound on the computation speed of a data flow graph.

We define two quantities to help us analyze the throughput limits of a data flow system: the *loop bound* and the *iteration bound*. The loop bound is the round-trip delay in a given loop of a data flow graph, divided by the number of delays in that loop. The iteration bound is the highest loop bound for any loop in a given data flow graph. The iteration bound sets an upper limit for the computational throughput of a data flow graph.

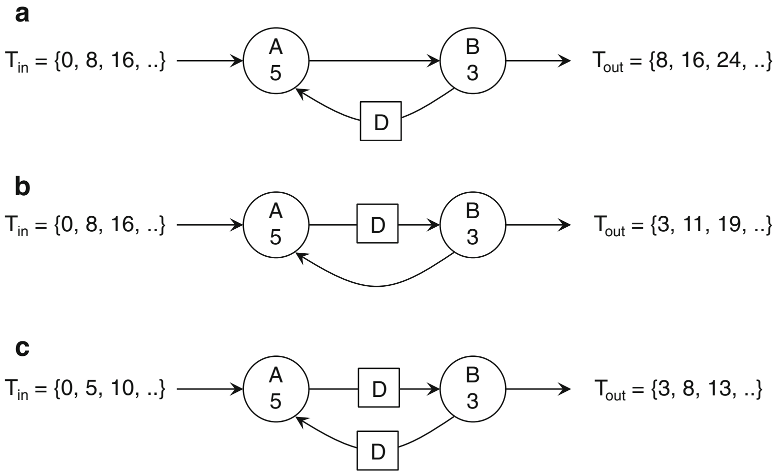

a
T_{in} = {0, 8, 16, ..}
A
5
B
3
D
T_{out} = {8, 16, 24, ..}
b
T_{in} = {0, 8, 16, ..}
A
5
D
B
3
T_{out} = {3, 11, 19, ..}
c
T_{in} = {0, 5, 10, ..}
A
5
D
B
3
D
T_{out} = {3, 8, 13, ..}

Fig. 2.16 Three data flow graphs

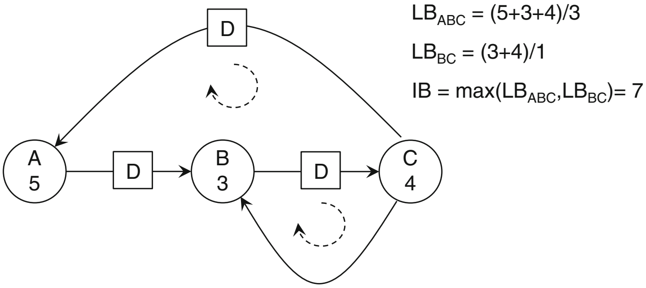


Fig. 2.17 Calculating loop bound and iteration bound

These concepts are explained with an example as shown in Fig. 2.17. This graph has three actors *A*, *B*, and *C*, each with different execution times. The graph has two loops: *BC* and *ABC*. The loop bounds are:

$$LB_{BC} = \frac{3+4}{1} = 7$$

$$LB_{ABC} = \frac{5+3+4}{3} = 4$$

The iteration bound of the system in Fig. 2.17 is the maximum of these two, or seven. This iteration bound implies that the implementation of this data flow graph will need at least 7 time units to process every iteration. The iteration bound thus sets an upper limit on throughput. If we inspect this graph closely, we conclude that loop *BC* is indeed the bottleneck of the system. Actors *A* and *C* have delays at their inputs, so that they can always execute in parallel. Actor *B* however, needs to wait

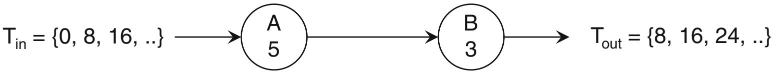


Fig. 2.18 Iteration bound for a linear graph

for a result from *C* before it can compute its output. Therefore, actor *B* and *C* can never run in parallel, and together, they define the iteration bound of the system.

It may appear that a linear graph, which has no loop, won't have an iteration bound. However, even a linear graph has an implicit iteration bound: a new sample cannot be read at the input until the output sample, corresponding to the previous input sample, has been generated. Figure 2.18 illustrates this for a linear graph with two actors *A* and *B*. When *A* reads a new sample, *B* will compute a corresponding output at time stamp 8. *A* can only start a new computation at time stamp 8; the absence of a delay between *A* and *B* prevents this any earlier. The analysis of the linear sections at the inputs and/or outputs of a data flow graph can be easily brought into account by assuming an implicit feedback from each output of a data flow graph to each input. This outermost loop is analyzed like any other loop to find the iteration bound for the overall system.

The iteration bound is an upper-limit for throughput. A given data flow graph may or may not be able to execute at its iteration bound. For example, the graph in Fig. 2.16c has an iteration bound of $(5+3)/2 = 4$ time units (i.e., a throughput of 1 sample per 4 time units), yet our analysis showed the graph to be limited at a throughput of 1 sample per 5 time units. This is because the slowest actor in the critical loop of Fig. 2.16c needs 5 time units to complete.

2.5 Transformations

Using performance analysis on data flow graphs, we can now evaluate suitable transformations to improve the performance of slow data flow graphs. We are interested in transformations that maintain the functionality of a data flow graph, but that increase the throughput and/or decrease the latency. This section will present several transformations which are frequently used to enhance system performance. Transformations don't affect the steady-state behavior of a data flow graph but, as we will illustrate, they may introduce transient effects, typically at startup. We cover the following transformations.

- **Multi-rate Expansion** is used to convert a multi-rate synchronous data flow graph to a single-rate synchronous data flow graph. This transformation is helpful because the other transformations assume single-rate SDF systems.
- **Retiming** considers the redistribution of delay elements in a data flow graph, in order to optimize the throughput of the graph. Retiming does not change the latency or the transient behavior of a data flow graph.

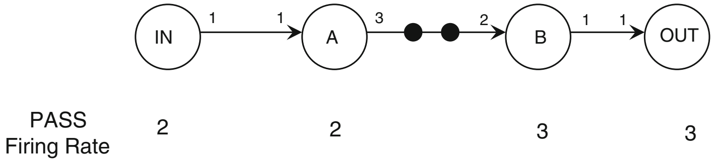


Fig. 2.19 Multi-rate data flow-graph

- **Pipeling** introduces additional delay elements in a data flow graph, with the intent of optimizing the iteration bound of the graph. Pipelining changes the throughput, and the transient behavior of a data flow graph.
- **Unfolding** increases the computational parallelism in a data flow graph by duplicating actors. Unfolding does not change the transient behavior of a data flow graph, but may modify the throughput.

2.5.1 Multirate Expansion

It is possible to transform a multi-rate SDF graph systematically to a single-rate SDF graph. The following steps to convert a multi-rate graph to a single-rate graph.

1. Determine the PASS firing rates of each actor
2. Duplicate each actor the number of times indicated by its firing rate. For example, given an actor A with a firing rate of 2, we create $A0$ and $A1$. These actors are two identical copies of the same generic actor A.
3. Convert each multi-rate actor input/output to multiple single-rate input/outputs. For example, if an actor input has a consumption rate of 3, we replace it with three single-rate inputs.
4. Re-introduce the queues in the data flow system to connect all actors. Since we are building a PASS system, the total number of actor inputs will be equal to the total number of actor outputs.
5. Re-introduce the initial tokens in the system, distributing them sequentially over the single-rate queues.

Consider the example of a multirate SDF graph in Fig. 2.19. Actor A produces three tokens per firing, actor B consumes two tokens per firing. The resulting firing rates are 2 and 3, respectively.

After completing steps 1–5 discussed above, we obtain the SDF graph shown in Fig. 2.20. The actors have duplicated according to their firing rates, and all multi-rate ports were converted to single-rate ports. The initial tokens are redistributed over the queues connecting instances of A and B. The distribution of tokens follows the sequence of queues between A,B (ie. follows the order a, b, etc.).

Multi-rate expansion is a convenient technique to generate a specification in which every actor needs to run at the same speed. For example, in a hardware

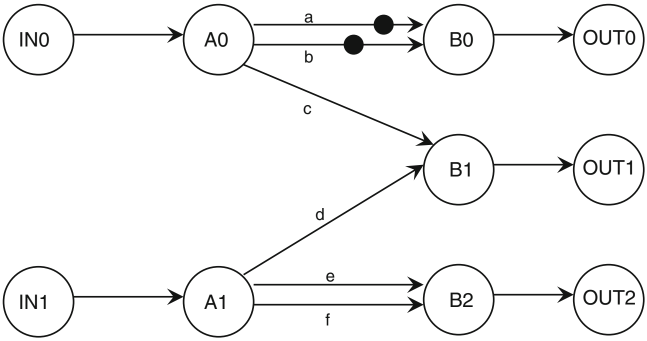


Fig. 2.20 Multi-rate SDF graph expanded to single-rate

implementation of data flow graphs, multi-rate expansion will enable all actors to run from the same clock signal.

2.5.2 Retiming

Retiming is a transformation on data flow graphs which doesn't change the total number of delays between input and output of a data flow graph. Instead, retiming is the redistribution the delays in the data flow graph. This way, the immediate dependency between actors can be broken, allowing them to operate in parallel. A retimed graph may have an increased system throughput. The retiming transformation is easy to understand. The transformation is obtained by evaluating the performance of successive markings of the data flow graph, and then selecting the one with the best performance.

Figure 2.21 illustrates retiming using an example. The top data flow graph, Fig. 2.21a, illustrates the initial system. This graph has an iteration bound of 8. However, the actual data output period of Fig. 2.21a is 16 time units, because actors *A*, *B*, and *C* need to execute as a sequence. If we imagine actor *A* to fire once, then it will consume the tokens (delays) at its inputs, and produce an output token. The resulting graph is shown in Fig. 2.21b. This time, the data output period has reduced to 11 time units. The reason is that actor *A* and the chain of actors *B* and *C*, can each operate in parallel. The graph of Fig. 2.21b is functionally identical to the graph of Fig. 2.21a: it will produce the same identical stream of output samples when given the same stream of input samples. Finally, Fig. 2.21c shows the result of moving the delay across actor *B*, to obtain yet another equivalent marking. This implementation is faster than the previous one; as a matter of fact, this implementation achieves the iteration bound of 8 time units per sample. No faster implementation exists for the given graph and the given set of actors.

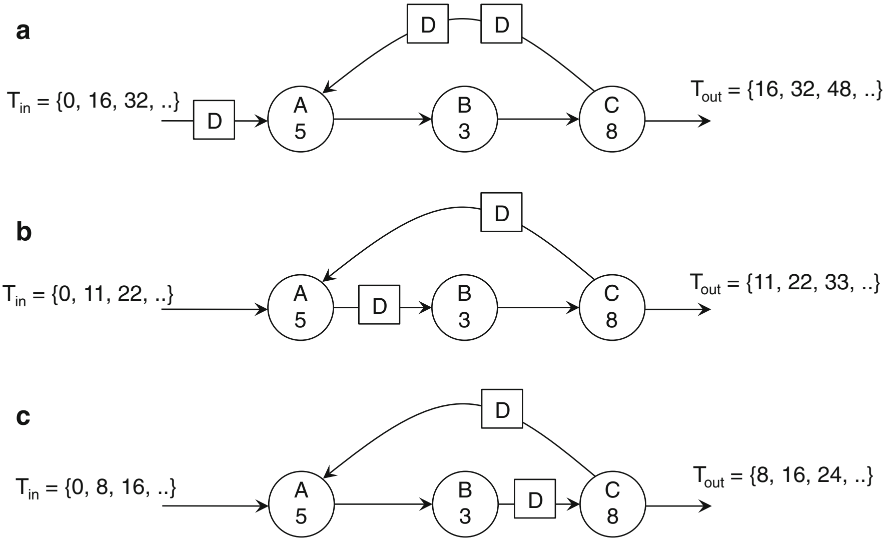

a
D
D
T_{in} = {0, 16, 32, ..}
D
A
5
B
3
C
8
T_{out} = {16, 32, 48, ..}
b
D
T_{in} = {0, 11, 22, ..}
A
5
D
B
3
C
8
T_{out} = {11, 22, 33, ..}
c
D
T_{in} = {0, 8, 16, ..}
A
5
B
3
D
C
8
T_{out} = {8, 16, 24, ..}

Fig. 2.21 Retiming: (**a**) Original Graph. (**b**) Graph after first re-timing transformation. (**c**) Graph after second re-timing transformation

Shifting the delay on the edge *BC* further would result in a delay on the outputs of actor *C*: one on the output queue, and one in the feedback loop. This final transformation illustrates an important property of retiming: it's not possible to increase the number of delays in a loop by means of retiming.

2.5.3 Pipelining

Pipelining increases the throughput of a data flow graph at the cost of increased latency. Pipelining can be easily understood as a combination of retiming and adding delays. Figure 2.22 demonstrates pipelining on an example. The orginal graph in Fig. 2.22a is extended with two pipeline delays in Fig. 2.22b. Adding delay stages at the input increases the latency of the graph. Before the delay stages, the system latency was 20 time units. After adding the delay stages, the system latency increases to 60 time units (3 samples with a latency of 20 time units each). The system throughput is 1 sample per 20 time units. We can now increase the system throughput by retiming the pipelined graph, so that we obtain Fig. 2.22c. The throughput of this graph is now 1 sample per 10 time units, and the latency is 30 time units (3 times 10 time units). This analysis points out an important property of pipelining: the slowest pipeline stage determines the throughput of the overall system.

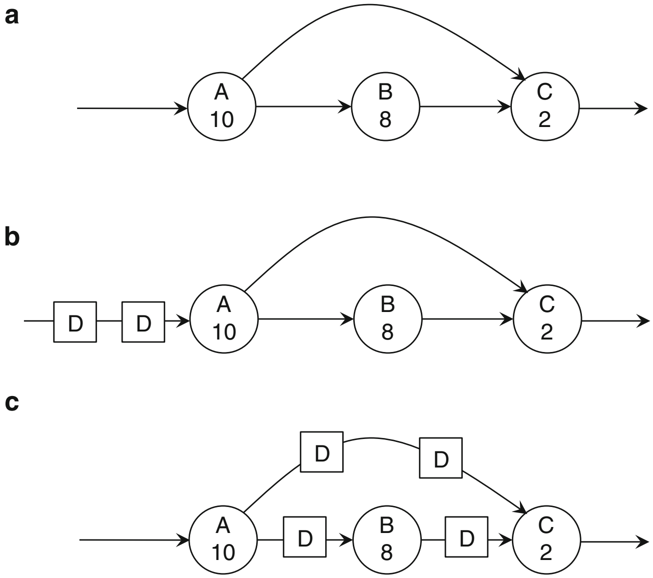


Fig. 2.22 Pipelining: (**a**) Original graph. (**b**) Graph after adding two pipeline stages. (**c**) Graph after retiming the pipeline stages

2.5.4 *Unfolding*

The final transformation we discuss is unfolding. The idea of unfolding is the parallel implementation of multiple instances of a given data flow graph. For example, assume a data flow graph G which processes a stream of samples. The two-unfolded graph $G2$ consists of two instances of G; this graph $G2$ processes two samples at a time.

The rules of unfolding are very similar to the rules of multi-rate expansion. Each actor A of the unfolded system is replicated the number of times needed for the unfolding. Next, the interconnections are made while respecting the sample sequence of the original system. Finally, the delays are redistributed over the interconnections.

The unfolding process is formalized as follows.

- Assume a graph G with an actor A and an edge AB carrying n delays.
- The v-unfolding of the graph G will replicate the actor A v times, namely A_0, A_1, .., A_{v-1}. The interconnection AB is replicated v times as well, AB_0, AB_1, .., AB_{v-1}.
- Edge AB_i connects A_i with B_k, for which $i : 0..v-1$ and $k = (i+n)\%v$.
- Edge AB_i carries $\lfloor (i+n)/v \rfloor$ delays. If $n < v$, then there will be $v-n$ edges without a delay.

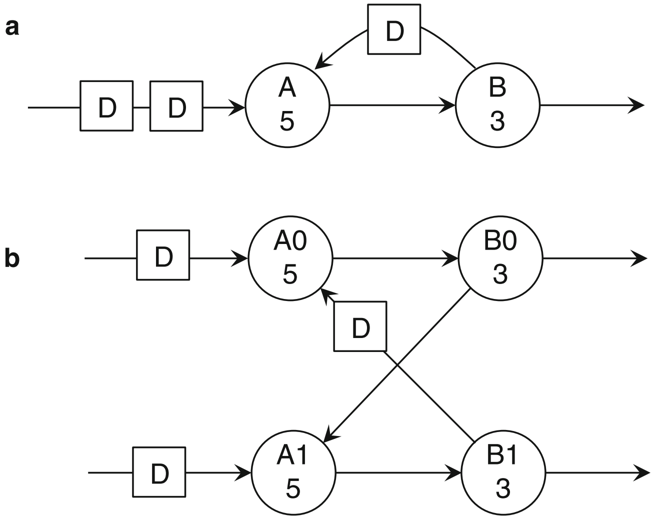


Fig. 2.23 Unfolding: (**a**) Original graph. (**b**) Graph after two-unfolding

Figure 2.23 illustrates the unfolding process with an example graph, unfolded two times. You can notice that the unfolded graph has two inputs and two outputs, and hence is able to accept twice as much data per iteration as the original data flow graph. On the other hand, unfolding the graph seems to slow it down. The critical loop now includes A_0, B_0, A_1 and B_1, but there is still only a single delay element in the overall loop. Hence, the iteration bound of a v-unfolded graph has increased v times.

Unfolding of data-flow graphs is used to process data streams with very high sample rates. In this case, the high-speed stream is expanded into v parallel streams. Stream i carries sample $s_i, s_{i+v}, s_{i+2v}, ..$ from the original stream. For example, in Fig. 2.23, the even samples would be processed by A_0 while the odd samples would be processed by A_1. However, because unfolding decreases the iteration bound, the overall computation speed of the system may be affected.

This completes our discussion on data flow graph transformations. Pipelining, retiming and unfolding are important performance-enhancing manipulations on data flow graphs, and they have a significant impact on the quality of the final implementation.

2.6 Data Flow Modeling Summary

Data flow models express concurrent systems in such as way that the models can map into hardware as well as in software. Data flow models consist of actors which communicate by means of tokens which flow over queues from one actor to the other. A data flow model can precisely and formally express the activities of a concurrent system. An interesting class of data flow systems are synchronous data flow (SDF) models. In such models, all actors can produce or consume a fixed amount of tokens per iteration (or invocation).

By converting a given SDF graph to a topology matrix, it is possible to derive stability properties of a data flow system automatically. A stable data flow system can be executed using a periodic admissible sequential schedule (PASS), a fixed period sequence of actor firings.

Next, we introduced a resource model for data flow graphs: a high-level model to analyze the performance of a data flow graph implementation. Performance is characterized by two metrics: latency and throughput. Latency is the time it takes to compute an output sample for a corresponding input sample. Throughput is the amount of samples that can be accepted per time unit at an input or an output.

We discussed four different transformations on data flow graphs. Multi-rate expansion is useful to create a single-rate SDF from a multi-rate SDF. Retiming redistributes the delays in a data flow graph in order to improve throughput. Pipelining has the same objective, but it adds delays to a data flow graph. Unfolding creates parallel instances from a data flow graph, in order to process multiple samples from a data stream in parallel.

Data flow modeling remains an important and easy-to-understand design and modeling technique. They are very popular in signal-processing application, or any application where infinite streams of signal samples can be captured as token streams. Data flow modeling is highly relevant to hardware-software codesign because of the clear and clean manner in which it captures system specifications.

2.7 Further Reading

Data flow analysis and implementation has been well researched over the past few decades, and data flow enjoys a rich body of literature.

In the early 1970s, data flow has been considered as a replacement for traditional instruction-fetch machines. Actual data flow computers were build that operate very much according to the SDF principles discussed here. Those early years of data flow have been documented very well at a retrospective conference called *Data flow to Synthesis Retrospective*. The conference honored Arvind, one of data flows' pioneers, and the online proceedings include a talk by Jack Dennis (Dennis 2007).

In the 1980s, data flow garnered attention because of its ability to describe signal processing problems well. For example, Lee and Messerschmit described SDF scheduling mechamisms (Lee and Messerschmitt 1987). Parhi and Messerschmit discussed retiming, pipelining and unfolding transformations of SDF graphs (Parhi and Messerschmitt 1989). Lee as well as Parhi have each authored an excellent textbook that includes data flow modeling and implementation as part of the material, see (Lee and Seshia 2011) and (Parhi 1999). The work from Lee eventually gave rise to the Ptolemy environment (Eker et al. 2003). Despite these successes,

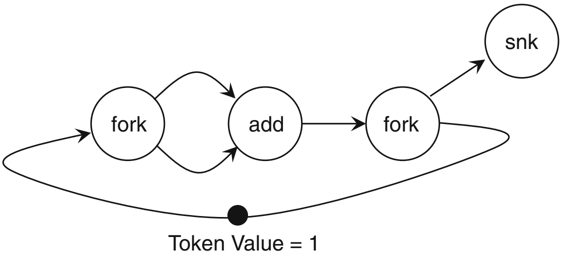


Fig. 2.24 SDF graph for Problem 2.1

data flow never became truly dominant compared to existing control-oriented paradigms. This is regrettable: a well-known, but difficult to solve, design problem that affects hardware and software designers alike is how to build parallel versions of solutions originally conceived as sequential (C) programs.

2.8 Problems

Problem 2.1. Consider the single-rate SDF graph in Fig. 2.24. The graph contains three types of actors. The fork actor reads one token and produces two copies of the input token, one on each output. The add actor adds up two tokens, producing a single token that holds the sum of the input tokens. The snk actor is a token-sink which records the sequence of tokens appearing at its input. A single initial token, with value 1, is placed in this graph. Find the value of tokens that is produced into the snk actor. Find a short-hand notation for this sequence of numbers.

Problem 2.2. The Fibonacci Number series F is defined by F(0)=0, F(1)=1, F(i)=F(i − 1) + F(i − 2) when i is greater then 1. By changing the marking of the SDF graph in Fig. 2.27, it is possible to generate the Fibonacci series into the snk actor. Find the location and the initial value of the tokens in the modified graph.

Problem 2.3. Consider the SDF graph in Fig. 2.25. Transform that graph such that it will produce the same sequence of tokens as tuples instead of as a sequence of singletons. To implement this, replace the snk actor with snk2, an actor which requires two tokens on two different inputs in order to fire. Next make additional transformations to the graph and its marking so that it will produce this double-rate sequence into snk2.

Problem 2.4. Data Flow actors cannot contain state variables. Yet, we can 'simulate' state variables with tokens. Using only an adder actor, show how you can implement an accumulator that will obtain the sum of an infinite series of input tokens.

Problem 2.5. For the SDF graph of Fig. 2.26, find a condition between x and y for a PASS to exist.

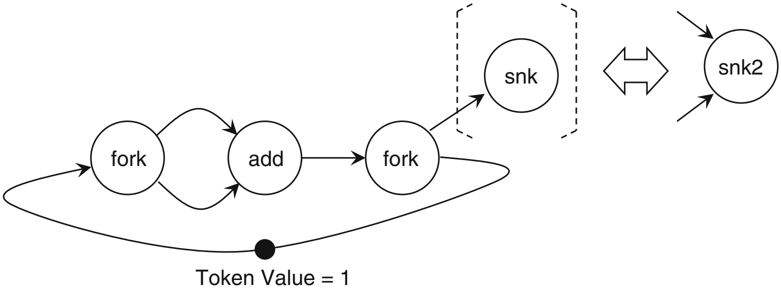


Fig. 2.25 SDF graph for Problem 2.3

Fig. 2.26 SDF graph for Problem 2.5

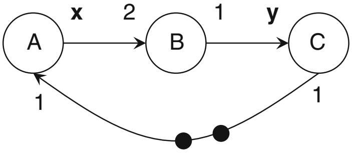


Fig. 2.27 SDF graph for Problem 2.6

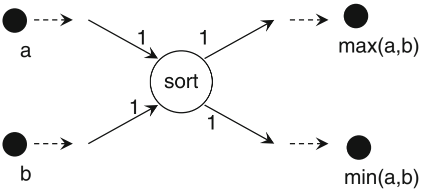
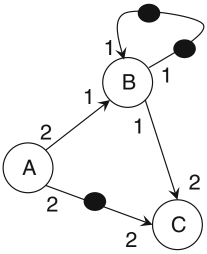


Fig. 2.28 SDF graph for Problem 2.7



Problem 2.6. Given the two-input sorting actor shown in Fig. 2.27. Using this actor, create a SDF graph of a sorting network with four inputs and four outputs.

Problem 2.7. Draw the multi-rate expansion for the multirate SDF given in Fig. 2.28. Don't forget to redistribute the initial tokens on the multirate-expanded result.

Problem 2.8. The data flow diagram in Fig. 2.29 demonstrates *reconvergent* edges: edges which go around one actor. Reconvergent edges tend to make analysis of a data flow graph a bit harder because they may imply that multiple critical loops may be laying on top of one another. This problem explores this effect.

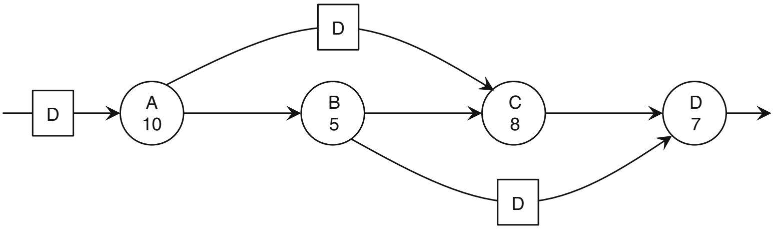
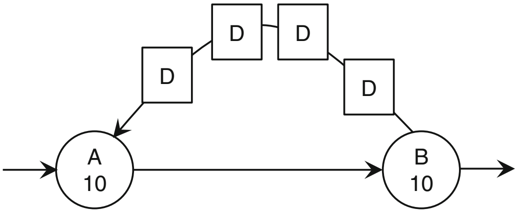


Fig. 2.29 SDF graph for Problem 2.8

Fig. 2.30 SDF graph for Problem 2.9



(a) Determine the loop bounds of all loops in the graph of Fig. 2.29. Take the input/output constraints into account by assuming an implicit loop from output to input.

(b) Using the results of the previous part, determine the iteration bound for this graph.

(c) Find the effective throughput of this graph, based on the distribution of delays as shown in Fig. 2.29.

(d) Does the system as shown in Fig. 2.29 achieve the iteration bound? If not, apply the retiming transformation and improve the effective throughput so that you get as close as possible to the iteration bound.

Problem 2.9. Unfold the graph in Fig. 2.30 three times. Determine the iteration bound before and after the unfolding operation.

Chapter 3
Data Flow Implementation in Software and Hardware

3.1 Software Implementation of Data Flow

The design space to map data flow in software is surprisingly broad. Nevertheless, a dataflow implementation will always start from the same semantics model with dataflow actors and dataflow queues.

3.1.1 Converting Queues and Actors into Software

Let's first recall the essential features of SDF graphs. SDF graphs represent concurrent systems, and they use actors which communicate over FIFO queues. Actor firing only depends on the availability of data (tokens) in the FIFO queues; the firing conditions are captured in the firing rule for that actor. The amount of tokens produced/consumed per firing at the output/input of an actor is specified by the production rate/consumption rate for that output/input. When implementing an SDF graph in software, we have to map *all* elements of the SDF graph in software: actors, queues, and firing rules. Under some conditions, the software implementation may be optimized. For example, when a fixed execution order of actors can be found (a PASS, as discussed in Chap. 2), it may be possible to skip testing of the firing rules. However, the principle remains that the implementation needs to follow the rules of dataflow semantics: optimizations (such as skipping the testing of firing rules) need to be motivated from analysis of the data flow graph.

Figure 3.1 demonstrates several different approaches to map dataflow into software. We'll distinguish the mapping of data flow to a multi-processor system from the mapping of data flow to a single processor system. When a data flow graph requires implementation in a multi-processor system, a designer will need to partition the actors in the graph over the processors. This partitioning is driven by several criteria. Typically, designers ensure that the computational load is balanced over the processors, and that the inter-processor communication is minimized.

P.R. Schaumont, *A Practical Introduction to Hardware/Software Codesign*,
DOI 10.1007/978-1-4614-3737-6_3, © Springer Science+Business Media New York 2013

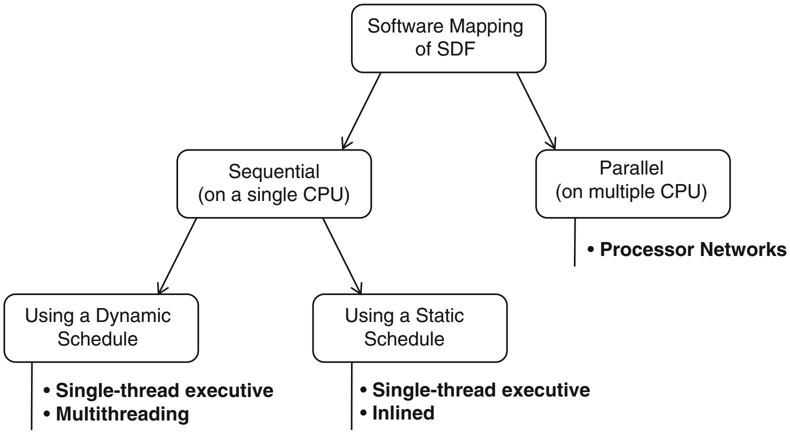


Fig. 3.1 Overview of possible approaches to map dataflow into software

However, the focus of this section will be on the single-processor implementation of a data flow graph. The key objective of a single-processor implementation of a data flow system is the efficient implementation of a sequential schedule. There are two methods to implement such a sequential schedule.

- We can use a *dynamic schedule*, which evaluates the execution order of actors during the execution of the SDF graph. Thus, at runtime, the software will evaluate the actors' firing rule and decide if the actor body should execute or not. A dynamic schedule can be implemented using a single-thread executive or else using a multi-thread executive.
- We can also use a *static schedule*, which means that the execution order of the actors is determined at design-time. A static schedule can be implemented using a single-threaded executive. However, because the static schedule fixes the execution order of the actors, there is an additional important optimization opportunity: we can treat the firing of multiple actors as a single firing. Eventually, this implies that we can inline the entire dataflow graph in a single function.

We'll start with the easy stuff: we show how to implement FIFO queues and dataflow actors in C. After that, we will be ready to implement static and dynamic schedules in software.

3.1.1.1 FIFO Queues

An SDF system requires, in principle, infinitely large FIFO queues. Such queues cannot be implemented; in practice, an implementation has to have bounded storage. If we know a PASS, we can derive a static schedule and determine the maximum number of tokens on each queue, and then appropriately choose the size for each queue.

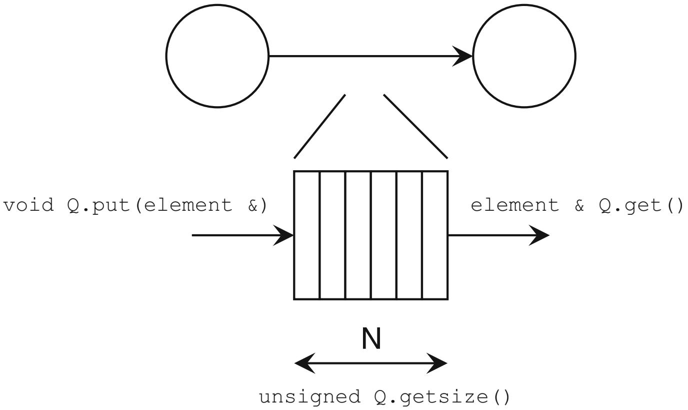


Fig. 3.2 A software queue

Figure 3.2 shows a software interface to a queue object. The software interface has two parameters and three methods.

- The number of elements N that can be stored by the queue (parameter).
- The data type element of a queue elements (parameter).
- A method to put elements into the queue.
- A method to get elements from the queue.
- A method to test the number of elements in the queue.

The storage organization can be done with a standard data structure such as a circular queue. A circular queue is a data structure consisting of an array of memory locations, a write-pointer and a read-pointer. Figure 3.3 illustrates the operation of a two-element circular queue. Such a queue uses an array of three locations. The read- and write-pointers map relative queue addresses to array addresses using modulo addressing. The head of the queue is at Rptr. Element I of the queue is at (Rptr + I) mod 3. The tail of the queue is at (Wptr − 1) mod 3.

Listing 3.1 shows the definition of a FIFO object in C. This example uses static allocation of the array, which implies that the maximum length of each queue is fixed before the simulation starts. For large systems, with many different queues, this strategy may be too pessimistic and too greedy on system resources. Another approach may be to use dynamically expanding queues. In that case, the FIFO object starts with a small amount of storage. Each time the FIFO queue would overflow, the amount of allocated storage is doubled. This technique is useful when a static schedule for the data flow system is unknown, or cannot be derived Problem 3.1.

3.1.1.2 Actors

A data flow actor can be implemented as a C function, with some additional support to interface with the FIFO queues. Designers will often differentiate between the

Listing 3.1 FIFO object in C

```
#define MAXFIFO 1024

typedef struct fifo {
  int data[MAXFIFO]; // token storage
  unsigned wptr;     // write pointer
  unsigned rptr;     // read pointer
} fifo_t;

void init_fifo(fifo_t *F) {
  F->wptr = F->rptr = 0;
}

void put_fifo(fifo_t *F, int d) {
  if (((F->wptr + 1) % MAXFIFO) != F->rptr) {
    F->data[F->wptr] = d;
    F->wptr = (F->wptr + 1) % MAXFIFO;
    assert(fifo_size(F) <= 10);
  }
}

int get_fifo(fifo_t *F) {
  int r;
  if (F->rptr != F->wptr) {
    r = F->data[F->rptr];
    F->rptr = (F->rptr + 1) % MAXFIFO;
    return r;
  }
  return -1;
}

unsigned fifo_size(fifo_t *F) {
  if (F->wptr >= F->rptr)
    return F->wptr - F->rptr;
  else
    return MAXFIFO - (F->rptr - F->wptr) + 1;
}

int main() {
  fifo_t F1;
  init_fifo(&F1); // resets wptr, rptr;
  put_fifo(&F1, 5); // enter 5
  put_fifo(&F1, 6); // enter 6
  printf("%d␣%d\n", fifo_size(&F1), get_fifo(&F1));
  // prints: 2 5
  printf("%d\n", fifo_size(&F1)); // prints: 1
}
```

internal activities of an actor, and the input-output behavior. The behavior corresponding to actor firing can be implemented as a simple C function. The firing-rule logic evaluates the firing condition, and calls the actor body when the condition is true.

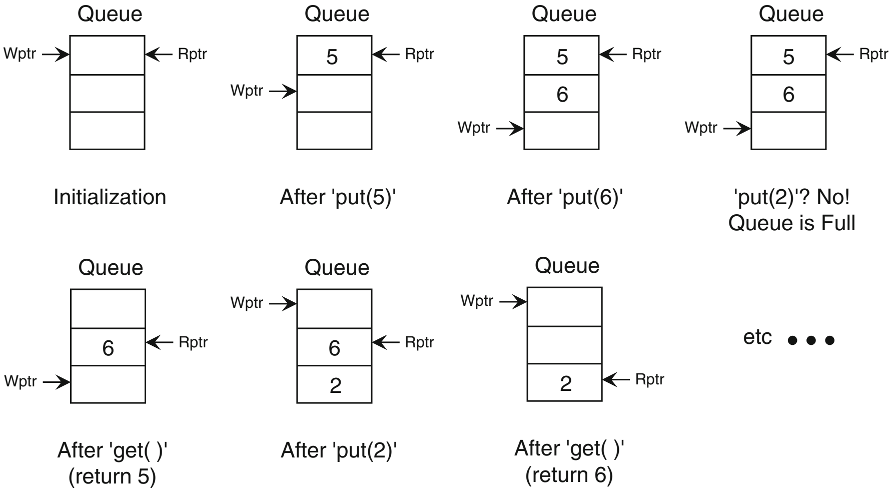


Fig. 3.3 Operation of the circular queue

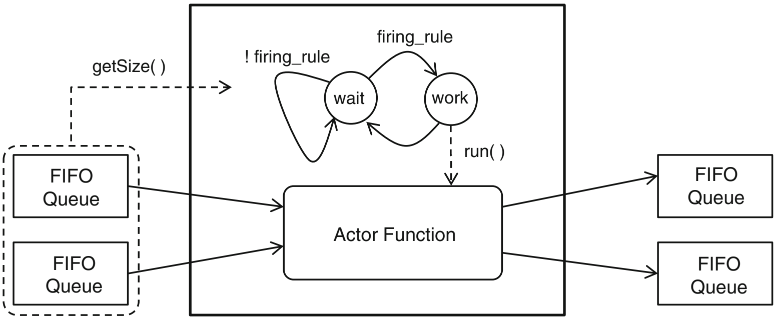


Fig. 3.4 Software implementation of the datafow actor

Figure 3.4 shows that the firing rule logic is implemented as a small, local controller inside of the actor. The local controller goes through two states. In the `wait` state the actor remains idle, but it tests the firing rule upon each invocation of the actor (an invocation is equivalent to calling the C function that implements the dataflow actor). When the firing rule evaluates true, the actor proceeds to the `work` state. In this state, the actor invocation will read tokens from the input queue(s), extract their values, and feed these to the actor body. Next, the resulting output values are written into the output queues, and the actor returns to the `wait` state. When implementing the actor, we must take care to implement the firing as specified. SDF actor firing implies that the actor has to read all input queues according to the specified consumption rates, and it has to write all output queues according to the specified production rates.

Let's say we will support up to eight inputs and outputs per actor, then we could define a `struct` to collect the input/output per actor as follows.

```
#define MAXIO 8
typedef struct actorio {
  fifo_t *in[MAXIO];
  fifo_t *out[MAXIO];
} actorio_t;
```

Next, we use `actorio_t` to model actors as functions. The following illustrates an actor with a single input and a single output. The actor reads two tokens, and produces their sum and difference.

```
void fft2(actorio_t *g) {
  int a, b;
  if (fifo_size(g->in[0]) >= 2) {
    a = get_fifo(g->in[0]);
    b = get_fifo(g->in[0]);
    put_fifo(g->out[0], a+b);
    put_fifo(g->out[0], a-b);
  }
}
```

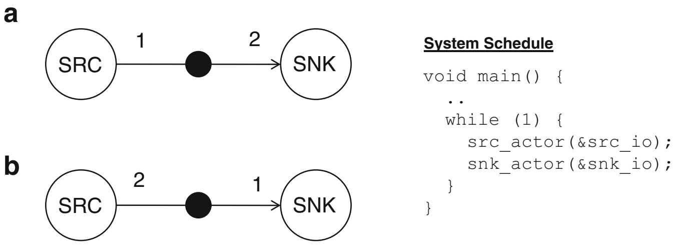
Finally, the `actorio_t` and `queue` objects can be instantiated in the main program, and the actor functions can be called using a system scheduler. We will first introduce dynamic scheduling techniques for software implementation of SDF, and next demonstrate an example that uses these techniques.

3.1.2 Software Implementation with a Dynamic Scheduler

A software implementation of SDF is obtained by combining several different actor descriptions, by interconnecting those actors using FIFO queues, and by executing the actors through a system schedule. In a *dynamic* system schedule, the firing rules of the actors will be tested at runtime; the system scheduling code consists of the firing rules, as well as the order in which the firing rules are tested.

Following the FIFO and actor modeling in C, as discussed in Sect. 3.1.1, we can implement a system schedule as a function that instantiates all actors and queues, and next calls the actors in a round-robing fashion.

```
void main() {
  fifo_t q1, q2;
  actorio_t fft2_io   = {{&q1}, {&q2}};
  ..
  init_fifo(&q1);
  init_fifo(&q2);
  ..
```



```
void main() {
  ..
  while (1) {
    src_actor(&src_io);
    snk_actor(&snk_io);
  }
}
```

Fig. 3.5 **(a)** A graph which will simulate under a single rate system schedule, **(b)** a graph which will cause extra tokens under a single rate schedule

```
  while (1) {
    fft2_actor(&fft2_io);
    // .. call other actors
  }
}
```

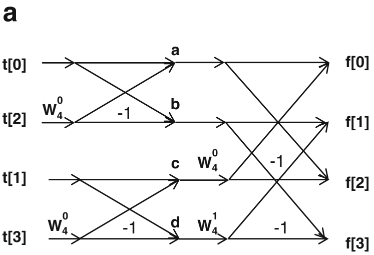
The interesting question, of course, is: what is the most appropriate invocation order of the actors in the system schedule? First, note that it is impossible to invoke the actors in the 'wrong' order, because each of them has a firing rule that prevents them from firing when there is no data available. Consider the example in Fig. 3.5a. Even though `snk` will be called as often as `src`, the firing rule of `snk` will only allow that actor to run when there is sufficient data available. This means that the `snk` actor will only fire every other time the `main` function invokes it.

While a dynamic scheduling loop can prevent actors from firing prematurely, it is still possible that some actors fire too often, resulting in the number of tokens on the interconnection queues continuously growing. This happens, for example, in Fig. 3.5b. In this case, the `src` actor will produce two tokens each time the main function invokes it, but the `snk` actor will only read one of these tokens per invocation. This means that, sooner or later, the queue between `src` and `snk` in Fig. 3.5b can overflow.

The problem of the system schedule in Fig. 3.5b is that the firing rate provided by the system schedule is different from the required firing rate for a PASS. Indeed, the PASS for this system would be `(src, snk, snk)`. However, the dynamic system schedule, given by the code on the right of Fig. 3.5, cannot make the firing rate of SNK higher than that of SRC. This problem can be addressed in several ways.

- **Solution 1:** We could adjust the system schedule to reflect the firing rate predicted by the PASS. In this case, the code for the system scheduler becomes:

```
void main() {
  ..
  while (1) {
    src_actor(&src_io);
    snk_actor(&snk_io);
```



b

```
a    = t[0] + W(0,4) * t[2] = t[0] + t[2]
b    = t[0] - W(0,4) * t[2] = t[0] - t[2]
c    = t[1] + W(0,4) * t[3] = t[0] + t[3]
d    = t[1] - W(0,4) * t[3] = t[1] - t[3]
f[0] = a    + W(0,4) * c    = a    + c
f[1] = b    + W(1,4) * d    = b    - j.d
f[2] = c    - W(0,4) * c    = a    - c
f[3] = b    - W(1,4) * d    = b    + j.d
```

Fig. 3.6 **(a)** Flow diagram for a four-point Fast Fourier Transform **(b)** Equivalent set of operations

```
    snk_actor(&snk_io);
  }
}
```

This solution is not very elegant, because it destroys the idea of a dynamic scheduler that automatically converges to the PASS firing rate. It also makes the `main` loop dependent on the topology of the SDF graph.

- **Solution 2:** We could adjust the code for the `snk` actor to continue execution as long as there are tokens present. Thus, the code for the `snk` actor becomes:

```
void snk_actor(actorio_t *g) {
  int r1, r2;
  while ((fifo_size(g->in[0]) > 0)) {
    r1 = get_fifo(g->in[0]);
    ... // do processing
  }
}
```

This is a better solution than the previous one, because it keeps the advantages of a dynamic system schedule.

3.1.3 Example: Four-Point Fast Fourier Transform as an SDF System

Figure 3.6a shows a four-point Fast Fourier Transform (FFT). It takes an array of 4 signal samples `t[0]` through `t[3]` and converts it to a spectral representation `f[0]` through `f[3]`. FFT's are extensively used in signal processing to do spectral analysis, filtering, and more. The references include a few pointers to detailed descriptions of the FFT algorithm.

In this example, we are interested in the mapping of a four-point FFT into a data flow graph. An FFT is made up out of 'butterfly' operations. An equivalent set of operations corresponding to the graph from Fig. 3.6a is shown in Fig. 3.6b. The twiddle factor `W(k,N)`, or W_N^k, is a complex number defined as $e^{-j2\pi k/N}$. Obviously, `W(0,4)=1` and `W(1,4)=-j`. The FFT thus produces complex numbers at

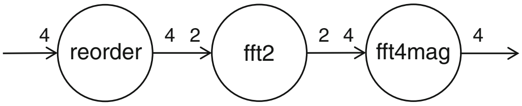


Fig. 3.7 Synchronous dataflow diagram for a four-point Fast Fourier Transform

the output. However, if the input values v[0] through v[3] are real values, then the output values V[1] and V[3] will be complex conjugate: there is thus some redundancy in the output.

Figure 3.7 shows a data flow model for the same flow diagram. It consists of three data flow actors: reorder, fft2, and fft4mag.

- reorder reads four tokens and reshuffles them according to the requirements of an FFT. In Fig. 3.6a, you can observe that the elements of the input array are not processed in linear order: t[0] and t[2] are processed by the first butterfly, while t[1] and t[3] are processed by the second butterfly. The reorder actor thus converts the sequence t[0], t[1], t[2], t[3] into the sequence t[0], t[2], t[1], t[3].
- fft2 calculates the butterflies for the left half of Fig. 3.6a. This actor reads two tokens, computes the butterfly, and produces two tokens.
- fft4mag calculates the butterflies of the right half of Fig. 3.6a. This actor reads four tokens, computes two butterflies, and produces four tokens. The fft4mag actor computes the magnitude vector real(V[0]*V[0]), real(V[1]*V[1]),real(V[2]*V[2]),real(V[3]*V[3]).

To implement the FFT as a data flow system, we would first need to compute a valid schedule for the actors. It's easy to see that a stable firing vector for this set of actors is $[q_{reorder}, q_{fft2}, q_{fft4mag}] = [1,2,1]$. Listing 3.2 shows a description for the reorder and fft4mag actors, as well as a main program to implement this schedule.

Listing 3.2 4-point FFT as an SDF system

```
void reorder(actorio_t *g) {
  int v0, v1, v2, v3;
  while (fifo_size(g->in[0]) >= 4) {
    v0 = get_fifo(g->in[0]);
    v1 = get_fifo(g->in[0]);
    v2 = get_fifo(g->in[0]);
    v3 = get_fifo(g->in[0]);
    put_fifo(g->out[0], v0);
    put_fifo(g->out[0], v2);
    put_fifo(g->out[0], v1);
    put_fifo(g->out[0], v3);
  }
}

void fft2(actorio_t *g) {
  int a, b;
  while (fifo_size(g->in[0]) >= 2) {
    a = get_fifo(g->in[0]);
    b = get_fifo(g->in[0]);
    put_fifo(g->out[0], a+b);
    put_fifo(g->out[0], a-b);
  }
}

void fft4mag(actorio_t *g) {
  int a, b, c, d;
  while (fifo_size(g->in[0]) >= 4) {
    a = get_fifo(g->in[0]);
    b = get_fifo(g->in[0]);
    c = get_fifo(g->in[0]);
    d = get_fifo(g->in[0]);
    put_fifo(g->out[0], (a+c)*(a+c));
    put_fifo(g->out[0], b*b - d*d);
    put_fifo(g->out[0], (a-c)*(a-c));
    put_fifo(g->out[0], b*b - d*d);
  }
}

int main() {
  fifo_t q1, q2, q3, q4;
  actorio_t reorder_io = {{&q1}, {&q2}};
  actorio_t fft2_io   =  {{&q2}, {&q3}};
  actorio_t fft4_io   =  {{&q3}, {&q4}};

  init_fifo(&q1);
  init_fifo(&q2);
  init_fifo(&q3);
  init_fifo(&q4);

  // test vector fft([1 1 1 1])
  put_fifo(&q1, 1);
  put_fifo(&q1, 1);
  put_fifo(&q1, 1);
  put_fifo(&q1, 1);

  // test vector fft([1 1 1 0])
  put_fifo(&q1, 1);
  put_fifo(&q1, 1);
  put_fifo(&q1, 1);
  put_fifo(&q1, 0);

  while (1) {
    reorder(&reorder_io);
    fft2   (&fft2_io);
    fft4mag(&fft4_io);
  }

  return 0;
}
```

The use of the actorio_t in the main program simplifies the interconnection of FIFO queues to actors. In this case, connections are made per actor, making it easy to follow which actor reads and writes which queue.

The actor descriptions use while loops, as discussed earlier, to ensure that the dynamic scheduler can achieve the PASS firing rate for each actor. In addition, thanks to the individual firing rules within the actors, and the deterministic property of SDF, we can write this scheduling loop in any order. For example, the following loop would yield the same results in the output queue q4.

```
int main() {
  ...
  while (1) {
    fft2   (&fft2_io);
    reorder(&reorder_io);
    fft4mag(&fft4_io);
  }
}
```

3.1.3.1 Multi-thread Dynamic Schedules

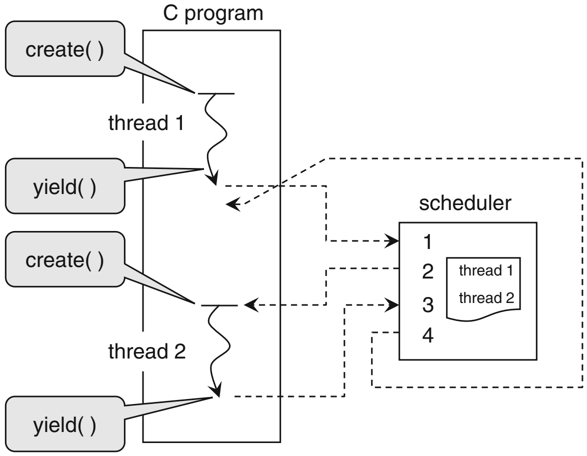
The actor functions, as described above, are captured as plain C functions. As a result, actors can maintain state using local variables in between invocations. We have to use global variables, or static variables. Another approach to create dynamic schedules is to use multi-threaded programming. We will discuss a solution based on multi-threaded programming, in which each actor executes in a separate thread.

A multithreaded C program is a program that has two concurrent threads of execution. For example, in a program with two functions, one thread could be executing the first function, while the other thread could be executing the second function. Since there is only a single processor to execute this program, we need to switch the processor back and forth between the two threads of control. This is done with a thread scheduler. Similar to a scheduler used for scheduling actors, a thread scheduler will switch the processor between threads.

We will illustrate the use of *cooperative* multithreading. In this model, the threads of control indicate at which point they release control back to the scheduler. The scheduler then decides which thread can run next.

Figure 3.8 shows an example with two threads. Initially, the user has provided the starting point of each thread using create(). Assume that the upper thread (thread1) is running and arrives at a yield( ) point. This is a point where the thread returns control to the scheduler. The scheduler maintains a list of threads under its control, and therefore knows that the lower thread (thread2) is ready to run. So it allows thread2 to run until that thread, too, comes at a yield point. Now the scheduler sees that each thread had a chance to run, so it goes back to the first thread. The first thread then will continue just after the yield point.

Fig. 3.8 Example of cooperative multi-threading



Consequently, two functions are enough to implement a threaded system: `create( )` and `yield( )`. The scheduler can apply different strategies to select each thread, but the simplest one is to let each thread run in turn – this is called a ‘round-robin’ scheduling strategy. We will use a cooperative multithreading library called Quickthreads. The Quickthreads API (Application Programmers’ Interface) consists of four function calls.

- `stp_init( )` initializes the theading system
- `stp_create(stp_userf_t *F, void *G)` creates a thread that will start execution with user function F. The function will be called with a single argument G. The thread will terminate when that function completes, or when the thread aborts.
- `stp_yield( )` releases control over the thread to the scheduler.
- `stp_abort( )` terminates a thread, so that it will be no more scheduled.

Here is a small program that uses the QuickThreads library.

Listing 3.3 Example of QuickThreads

```
#include "../qt/stp.h"
#include <stdio.h>

void hello(void *null) {
  int n = 3;
  while (n-- > 0) {
    printf("hello\n");
    stp_yield();
  }
}

void world(void *null) {
  int n = 5;
  while (n-- > 0) {
    printf("world\n");
    stp_yield();
  }
}

int main(int argc, char **argv) {
  stp_init();
  stp_create(hello, 0);
  stp_create(world, 0);
  stp_start();
  return 0;
}
```

This program creates two threads (lines 21 and 22), one which starts at function hello, and another which starts at function world. Function hello (lines 3–9) is a loop that will print "hello" three times, and yield after each iteration. After the third time, the function will return, which also terminates the thread. Function world (lines 11–17) is a loop that will print "world" five times, and yield at end of each iteration. When all threads are finished, the main function will terminate. We compile and run the program as follows. The references include a link to the source code of QuickThreads.

```
>gcc -c ex1.c -o ex1 ../qt/libstp.a ../qt/libqt.a
./ex1
hello
world
hello
world
hello
world
world
world
```

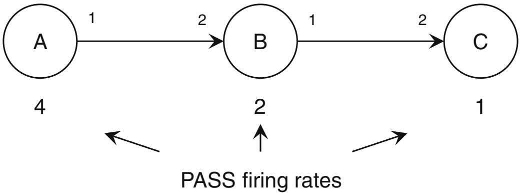
The printing of hello and world are interleaved for the first three iterations, and then the world thread runs through completion.

We can now use this multi-threading system to create a multi-thread version of the SDF scheduler. Here is the example of a fft2 actor, implemented using the cooperative threading model.

```
void fft2(actorio_t *g) {
  int a, b;
  while (1) {
    while (fifo_size(g->in[0]) >= 2) {
        a = get_fifo(g->in[0]);
        b = get_fifo(g->in[0]);
        put_fifo(g->out[0], a+b);
        put_fifo(g->out[0], a-b);
    }
    stp_yield();
}
```

The system scheduler now creates threads rather than directly invoking actors:

```
int main() {
```



```
while(1) {
  // call A four times
  A(); A(); A(); A();

  // call B two times
  B(); B();

  // call C one time
  C();
}
```

Fig. 3.9 System schedule for a multirate SDF graph

```
  fifo_t q1, q2;
  actorio_t fft2_io   = {{&q1}, {&q1}};
  ...
  stp_create(fft2, &fft2_io); // create thread
  ...
  stp_start(); // run the schedule
}
```

The execution rate of the actor code must be equal to the PASS firing rate in order to avoid unbounded growth of tokens in the system. A typical cooperative multi-threading system uses round-robin scheduling.

3.1.4 *Sequential Targets with Static Schedule*

When we have completed the PASS analysis for an SDF graph, we know at least one solution for a feasible sequential schedule. We can use this to optimize the implementation in several ways.

- First, we are able to write an exact schedule, such that every actor fires upon every invocation. Hence, we are able to remove the firing rules of the actors. This will yield a small performance advantage. Of course, we can no longer use such actors with dynamic schedulers.
- Next, we can also investigate the optimal interleaving of actors such that the storage requirements for the queues are reduced.
- Finally, we can create a fully inlined version of the SDF graph, by exploiting our knowledge on the static, periodic behavior of the system as much as possible. We will see that this not only allows us to get rid of the queues, but also allows us to create a fully inlined version of the entire SDF system.

Consider the example in Fig. 3.9. From this SDF topology, we know that the relative firing rates of A, B, and C must be 4, 2, and 1 to yield a PASS. The right side of the figure shows an example implementation of this PASS. The A, B, C actors are called in accordance to their PASS rate. For this particular interleaving, it is easy to see that in a steady state condition, the queue `AB` will carry a maximum of four tokens, while the queue `BC` will contain a maximum of two tokens. This is not the most optimal interleaving. By calling the actors in the sequence `(A,A,B,A,A,B,C)`, the maximum amount of tokens on any queue is reduced to two. Finding an optimal interleaving in an SDF graph is an optimization problem. While an in-depth discussion of this problem is beyond the scope of this book, remember that actor interleaving will affect the storage requirements for the implementation.

Implementing a truly *static* schedule means that we will no longer test firing rules when calling actors. In fact, when we call an actor, we will have to guarantee that the required input tokens are available. In a system with a static schedule, all SDF-related operations get a fixed execution order: the actor firings, and the sequences of `put` and `get` operations on the FIFO queues. This provides the opportunity to optimize the resulting SDF system.

We will discuss optimization of single-thread SDF systems with a static schedule using an example we discussed before – the four-point Fast Fourier Transform.

The system uses three actors – `reorder`, `fft2` and `fft4rev` – which have PASS firing rates of 1, 2 and 1 respectively. A feasible static and cyclic schedule could be, for example: `[reorder, fft2, fft2, fft4rev]`.

Next, let's optimize the system description from Listing 3.2 as follows.

1. Because the firing order of actors can be completely fixed, the access order on queues can be completely fixed as well. This latter fact will allow the queues themselves to be optimized out and replaced with fixed variables. Indeed, assume for example that we have determined that the access sequence on a particular FIFO queue will always be as follows:

```
loop {
  ...
  q1.put(value1);
  q1.put(value2);
  ...
  .. = q1.get();
  .. = q1.get();
}
```

   In this case, only two positions of FIFO q1 are occupied at a time. Hence, FIFO q1 can be replaced by two single variables.

```
loop {
  ...
  r1 = value1;
  r2 = value2;
  ...
  .. = r1;
  .. = r2;
}
```

2. As a second optimization, we can inline actor code inside of the main program and the main scheduling loop. In combination with the above optimization, this will allow to remove the firing rules and to collapse an entire dataflow graph

Listing 3.4 Inlined data flow system for the four-point FFT

```
void dfsystem(int   in0,   in1,   in2,   in3,
                  *out0, *out1, *out2, *out3) {
  int reorder_out0, reorder_out1, reorder_out2, reorder_out3;
  int fft2_0_out0,  fft2_0_out1,  fft2_0_out2,  fft2_0_out3;
  int fft2_1_out0,  fft2_1_out1,  fft2_1_out2,  fft2_1_out3;
  int fft4mag_out0, fft4mag_out1, fft4mag_out2, fft4mag_out3;

  reorder_out0 = in0;
  reorder_out1 = in2;
  reorder_out2 = in1;
  reorder_out3 = in3;

  fft2_0_out0  = reorder_out0 + reorder_out1;
  fft2_0_out1  = reorder_out0 - reorder_out1;

  fft2_1_out0  = reorder_out2 + reorder_out3;
  fft2_1_out1  = reorder_out2 - reorder_out3;

  fft4mag_out0  = (fft2_0_out0 + fft2_1_out0)*
                  (fft2_0_out0 + fft2_1_out0);
  fft4mag_out1  =  fft2_0_out1*fft2_0_out1 -
                   fft2_1_out1*fft2_1_out1;
  ftt4mag_out2  = (fft2_0_out0 - fft2_1_out0)*
                  (fft2_0_out0 - fft2_1_out0);
  fft4mag_out3  =  fft2_0_out1*fft2_0_out1 -
                   fft2_1_out1*fft2_1_out1;
}
```

in a single function. In case an actor would have a PASS firing rate above one, multiple instances of the actor body are needed. If needed, the multi-rate expansion technique discussed in Sect. 2.5.1 can be used to identify the correct single-rate system topology.

When we apply these optimizations to the four-point FFT example, each queue can be replaced by a series variables. The optimized system is shown in Listing 3.4. We can expect the runtime of this system to decrease significantly: there are no firing rules, no FIFO manipulations and no function boundaries. This is possible because we have determined a valid PASS for the initial data flow system, and because we have chosen a fixed schedule to implement that PASS.

3.2 Hardware Implementation of Data Flow

In this section, we are particularly interested in simple, optimized implementations. The use of hardware FIFO's will be discussed later.

3.2.1 Single-Rate SDF Graphs into Hardware

The simplest case is the mapping from a single-rate SDF graph to hardware. In a single rate schedule, the relative firing rate of all actors is equal to 1. This means that all actors will execute at the same rate. If the actors are implemented in hardware, all of them will be working at the same clock frequency.

We will be mapping such a single-rate graph into hardware using the following three rules.

1. All actors are implemented as combinational circuits.
2. All communication queues are implemented as wires (without storage).
3. Each initial token on a communication queue is replaced by a register.

The following observations can be made for circuits developed in this fashion. A combinational circuit is a circuit which can finish a computation within a single clock cycle (assuming no internal feedback loops exist). By implementing an actor as a combinational circuit, we ensure that it can do a complete firing within a single clock cycle. Now imagine two combinational circuits, implemented back-to-back. Such a combination will still operate as a single combinational circuit, and thus it will still complete computation within a single clock cycle. When two data flow actors, implemented as combinational circuits, are placed back-to-back, they will finish within a single clock cycle. Hence, a complete iteration of the schedule still takes only a single clock cycle. If we perform a scheduling analysis and identify a PASS, then the hardware circuit, created by the mapping rules described above, will be a valid implementation.

There is, however, a catch. The speed of computation of a combinational circuit is finite. Therefore, the speed of the overall hardware implementation created from a dataflow graph is limited as well. When actors need to compute sequentially, their computation times will add up. Hence, when actors are implemented as combinational circuits, their combined delay needs to remain below the clock period used for the hardware system. There is therefore a limit on the maximum clock frequency used for the system. Note, however, that a precise analysis of this problem is more complex than simply making the sum of computation times of all actors in the PASS.

We can define the *critical path* in a dataflow graph in terms of the resource model introduced in Sect. 2.4.2. First, define the latency of a path as the sum of actor latencies included in the path. Next, call a combinational path, a path which does not contain initial tokens on any of the communication queues included in the path. Finally, call the critical path of a dataflow graph the longest combinational path in the graph.

We next illustrate the hardware implementation of single-rate SDF graphs with an example: an SDF system for Euclid's Greatest Common Divisor algorithm. The SDF in Fig. 3.10 evaluates the greatest common divisor of two numbers *a* and *b*. It uses two actors: `sort` and `diff`.

The `sort` actor reads two numbers, sorts them and copies them to the output. The `diff` actor subtracts the smallest number from the largest one, as long as they

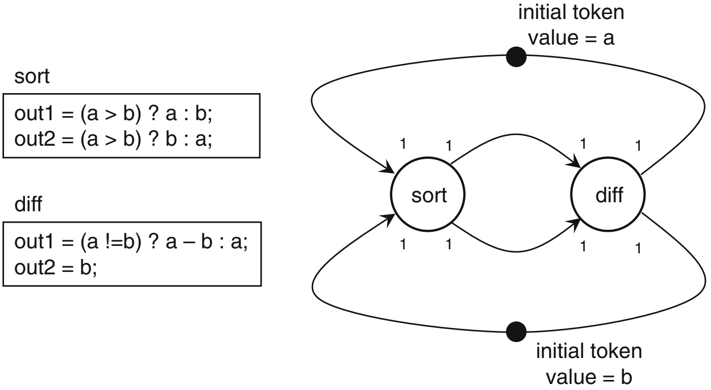


Fig. 3.10 Euclid's greatest common divisor as an SDF graph

are different. If this system runs for a while, the value of the tokens moving around converge to the greatest common divisor of the two numbers *a* and *b*. For example, assume

$$(a_0, b_0) = (16, 12) \tag{3.1}$$

then we see the following sequence of token values.

$$(a_1, b_1) = (4, 12), (a_2, b_2) = (8, 4), (a_3, b_3) = (4, 4), \ldots \tag{3.2}$$

a_i and b_i are the token values upon iteration i of the PASS. Since this sequence converges to the tuple $(4, 4)$, the greatest common divisor of 12 and 16 is 4.

We now demonstrate a PASS for this system. The topology matrix G for this graph is shown below. The columns, left to right, correspond to each node from the SDF graph, left to right.

$$G = \begin{bmatrix} +1 & -1 \\ +1 & -1 \\ -1 & +1 \\ -1 & +1 \end{bmatrix} \begin{array}{l} \leftarrow edge(sort, diff) \\ \leftarrow edge(sort, diff) \\ \leftarrow edge(diff, sort) \\ \leftarrow edge(diff, sort) \end{array} \tag{3.3}$$

The rank of this matrix is one, since the columns complement each other. There are two actors in the graph, so we conclude that the condition for PASS (i.e. rank(G) = nodes – 1) is fulfilled. A valid firing vector for this system is one in which each actor fires exactly once per iteration.

$$q_{PASS} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{3.4}$$

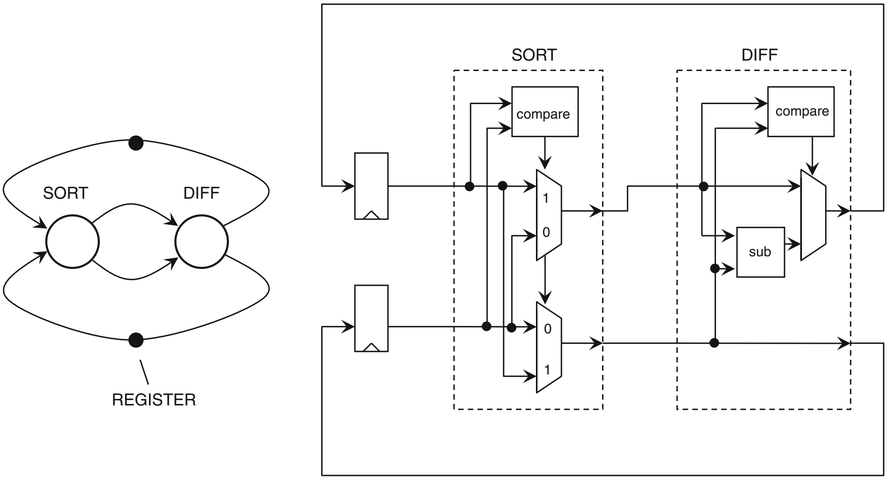


Fig. 3.11 Hardware implementation of euclid's algorithm

Based on this analysis, we can now proceed with the hardware implementation of the Euclid design. As discussed earlier, we use the following transformation.

1. Map each communication queue to a wire.
2. Map each queue containing a token to a register. The initial value of the register must equal the initial value of the token.
3. Map each actor to a combinational circuit, which completes a firing within a clock cycle. Both the `sort` and `diff` actors require no more than a comparator module, a few multiplexers and a subtractor.

Figure 3.11 illustrates how this works out for the Euclid example. In every single clock cycle, the `sort` actor and the `diff actor` are computed. The speed of computation of the overall circuit is determined by the combined computation speed of `sort` and `diff`. Indeed, the critical path of this graph is a path starting at `sort` and ending at `diff`. Assume that `sort` requires 40 ns of time to compute, and `diff` requires 60 ns of time, then the critical path of this system is 100 ns. Therefore, the maximum clock frequency of this design is 10 MHz.

3.2.2 *Pipelining*

It should be no surprise that some of the transformations, discussed earlier in Sect. 2.5, can also be used to enhance the throughput of hardware implementations. Pipelining is a very good example.

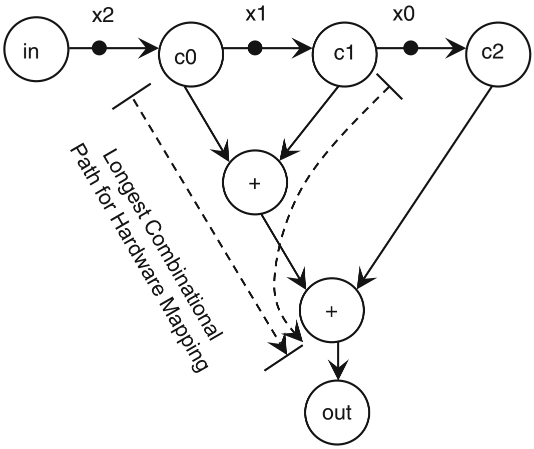


Fig. 3.12 SDF graph of a simple moving-average application

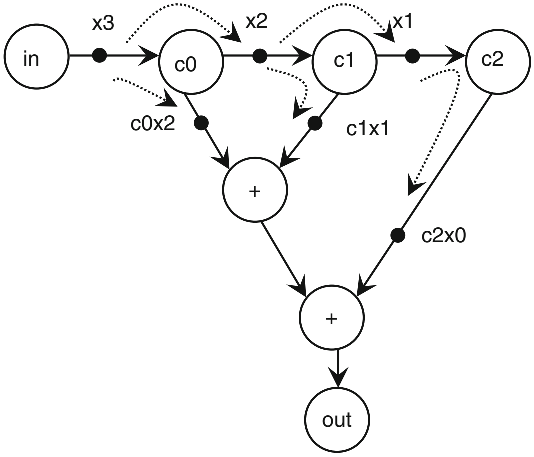


Fig. 3.13 Pipelining the moving-average filter by inserting additional tokens (1)

Figure 3.12 shows a data flow specification of a digital filter. It evaluates a weighted sum of samples of an input stream, with the sum defined as $out = x0.c2 + x1.c1 + x2.x0$.

It can be seen from this graph that the critical path is equal to a constant multiplication (with $c0$ or $c1$) and two additions. We would like to 'push down' initial tokens into the adder tree. With the rules of data flow execution, this is easy. Consider a few subsequent markings of the graph. Assume the `in` actor fires additional tokens, and the $c0$, $c1$, $c2$ and `add` actors fire as well so that additional tokens start to appear on queues that have no such tokens. For example, assume that the `in` actor produces a single additional token $x3$. Then the resulting graph looks as in Fig. 3.13.

In this graph, the critical path is reduced to only two additions. By letting the `in` actor produce another token, we will be able to reduce the critical path to a single addition, as shown in Fig. 3.14. The resulting pipelined SDF graph thus can be implemented as shown in Fig. 3.15.

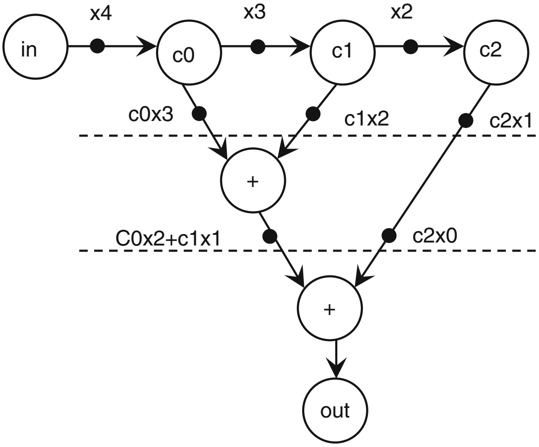


Fig. 3.14 Pipelining the moving-average filter by inserting additional tokens (2)

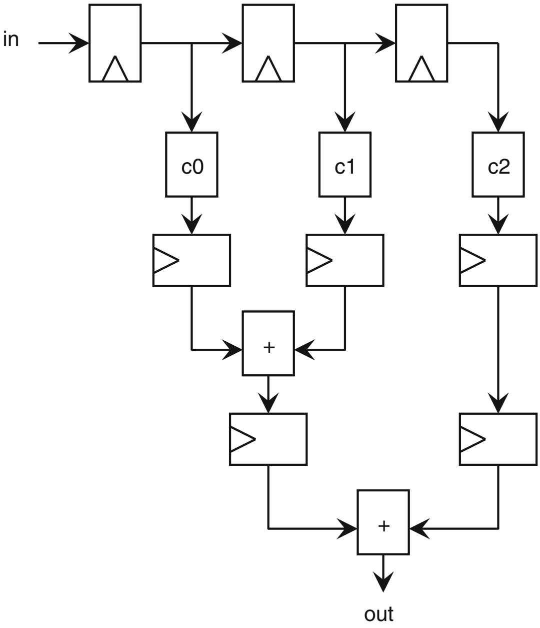


Fig. 3.15 Hardware implementation of the moving-average filter

Remember that pipelining requires you to introduce additional tokens. This may change the behavior of the dataflow graph. The change in behavior is obvious in the case of feedback loops, such as shown in the accumulator circuit in Fig. 3.16. Using a single token in the feedback loop of an add actor will accumulate all input samples. Using two tokens in the feedback loop will accumulate the odd samples and even samples separately. To avoid introducing accidental tokens in a loop, you can also perform pipelining as follows: introduce initial tokens at the input or output of the

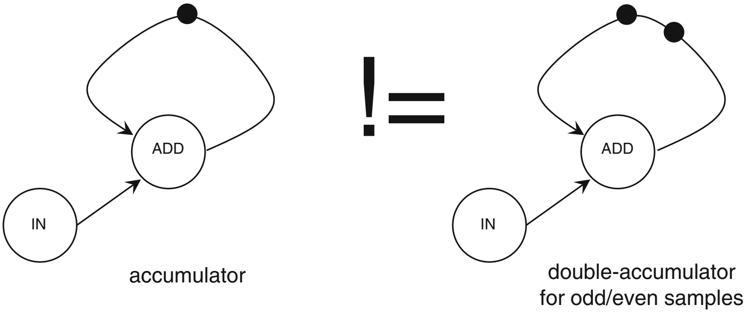


Fig. 3.16 Loops in SDF graphs cannot be pipelined

graph, outside of any loop. Next, retime the dataflow graph to reduce the critical path.

3.3 Hardware/Software Implementation of Data Flow

The implementation techniques used to map data flow graphs in hardware or software can be combined. A data flow system with multiple actors can be implemented such that part of the actors are implemented in hardware, while the other half are implemented in software. This section illustrates, by means of an example, how the interface between hardware and software can be handled.

Figure 3.17 shows a single rate data flow system with two actors and an initial token in between them. We map this system such that the first actor, `ctr`, is implemented in hardware, while the second actor, `snk`, is implement in software. We are using an 8051 microcontroller. Similar to the example of Sect. 1.1.3, we will use microcontroller ports to connect hardware and software.

The interface between hardware and software physically consists of three different connections: a data bus, a `req` connection (request) from hardware to software, and an `ack` connection (acknowledge) from software to hardware. The purpose of `req` and `ack` is to synchronize hardware and software when they communicate a token. A communication queue in data flow also needs storage; this storage is implemented on the 8051 processor as a FIFO queue.

Listing 3.5 shows a GEZEL system description of the data flow design of Fig. 3.17. The hardware actor is included on lines 1–24; the rest of the listing includes an 8051 processor, and communication ports to connect the hardware actor

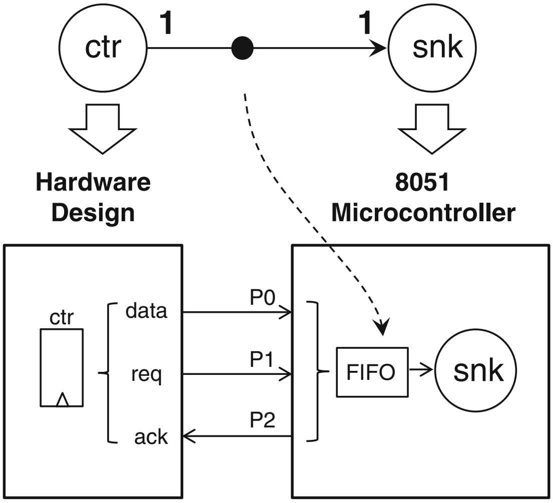


Fig. 3.17 Hybrid hardware/software implementation of a dataflow graph

Listing 3.5 GEZEL hardware description of data flow example of Fig. 3.17

```
dp send_token(out dout    : ns(8);
              out req     : ns(1);
              in  ack     : ns(1)) {
  reg ctr    : ns(8);
  reg rack   : ns(1);
  reg rreq   : ns(1);
  always {
    rack = ack;
    rreq = rack ? 0 : 1;
    ctr  = (rack & rreq) ? ctr + 1 : ctr;
    dout = ctr;
    req  = rreq;
  }
  sfg transfer {
    $display($cycle, " token ", ctr);
  }
  sfg idle {}
}
fsm ctl_send_token(send_token) {
  initial s0;
  state s1;
  @s0 if (rreq & rack) then (transfer) -> s0;
                       else (idle)     -> s0;
}

ipblock my8051 {
  iptype "i8051system";
  ipparm "exec=df.ihx";
  ipparm "verbose=1";
  ipparm "period=1";
}

ipblock my8051_data(in data : ns(8)) {
  iptype "i8051systemsink";
  ipparm "core=my8051";
  ipparm "port=P0";
}

ipblock my8051_req(in data : ns(8)) {
  iptype "i8051systemsink";
  ipparm "core=my8051";
  ipparm "port=P1";
}

ipblock my8051_ack(out data : ns(8)) {
  iptype "i8051systemsource";
  ipparm "core=my8051";
  ipparm "port=P2";
}

dp sys {
  sig data, req, ack : ns(8);
  use my8051;
  use my8051_data(data);
  use my8051_req (req);
  use my8051_ack (ack);
  use send_token (data, req, ack);
}

system S {
  sys;
}
```

Listing 3.6 Software description of data flow example of Fig. 3.17

```
#include <8051.h>
#include "fifo.c"

void collect(fifo_t *F) {
  if (P1) {            // if hardware has data
    put_fifo(F, P0); //   then accept it
    P2 = 1;            //     indicate data was taken
    while (P1 == 1); //     wait until the hardware
                               acknowledges
    P2 = 0;            //     and reset
  }
}

unsigned acc;
void snk(fifo_t *F) {
  if (fifo_size(F) >= 1)
    acc += get_fifo(F);
}

```

```
void main() {
  fifo_t F1;

  init_fifo(&F1);
  put_fifo(&F1, 0); // initial token
  acc = 0;

  while (1) {
    collect(&F1);
    snk(&F1);
  }
}
```

to the 8051. The hardware actor uses a so-called handshake protocol to ensure synchronization with the software. Synchronization protocols will be discussed in detail in Chap. 11.

Listing 3.6 shows the 8051 software to interface the hardware actor. The schedule in the `main` function invokes two functions, `collect` and `snk`. The first, `collect`, implements the synchronization with hardware. Every token received from the hardware is entered into a FIFO queue. The other function, `snk`, is a standard data flow actor.

Simulation of this design proceeds in the same way as the example in Sect. 1.1.3. First, compile the software, and next, run the cosimulation. The first 50,000 cycles of the simulation look as follows.

```
> sdcc dfsys.c
> /opt/gezel/bin/gplatform -c 50000 dfsys.fdl
i8051system: loading executable [df.ihx]
0x00    0x01    0x01    0xFF
17498 token 0/1
0x01    0x00    0x00    0xFF
0x01    0x01    0x01    0xFF
26150 token 1/2
0x02    0x00    0x00    0xFF
0x02    0x01    0x01    0xFF
34802 token 2/3
0x03    0x00    0x00    0xFF
0x03    0x01    0x01    0xFF
43454 token 3/4
0x04    0x00    0x00    0xFF
Total Cycles: 50000
```

It takes several thousand cycles to transfer a single token; this overhead can be completely attributed to software on the 8051. At 12 cycles per instruction, each token requires 712 instruction of the 8051. However, after applying software optimization techniques such as inlining, the performance of the system will improve dramatically; See Problem 3.8.

3.4 Summary

We discussed three possible target implementation for data flow systems: software, hardware, and combined hardware/software. The design space is broad and offers many alternatives.

For a sequential software implementation, we can use either threads or else static scheduling of C functions to capture the concurrent behavior of a data flow system. Optimization techniques, such as inlining of static data flow, yield compact and efficient implementations of software data flow systems.

For hardware implementation, a simple one-to-one conversion technique translates single-rate SDF graphs into hardware. The data flow transformation techniques we discussed earlier, including pipelining and retiming, can be applied to optimize the performance of graphs with a long critical path.

Finally, we discussed how both of these techniques in hardware and software can be combined into a hybrid system. In this case, we need to build synchronization between the hardware part and the software part of the data flow system.

Data flow modeling and – implementation is an important system design technique, and its ideas are present in many aspects of hardware/software design.

3.5 Further Reading

As much as you should learn the concepts, elegance, and advantages of data flow modeling, you should also understand their limitations: when to use them, and when not to use them.

An excellent place to start studying on dataflow implementations would be to look at the Ptolemy environment (Eker et al. 2003), and look at how it gets around the limitations of data flow with additional models of computation.

Dataflow excels in the description of streaming processing, and therefore it remains very popular for signal processing applications. In particular, the recent trend towards multi-processors has spurred a new interest in streaming applications. System specification is done in a dataflow-variant or language, and an automatic design environment maps this to a multiprocessor target. Some of the recent work in this area includes StreamIt (which maps to an IBM Cell Processor) (Thies 2008) and Brook (which maps to a Graphics Processor) (Stanford Graphics Lab 2003). Mature versions of data flow modeling system, with extensions for control modeling and event modeling, are also available in commercial environments such as Matlab's Simulink and National Instruments' Labview.

The Quickthreads threading system discussed in this chapter was developed by D. Keppel (Keppel 1994). I've used it for its simplicity and elegance, not for its completeness! If you're interested in threads, you'll have to consult a book on parallel and/or concurrent programming, for example (Butenhof 1997).

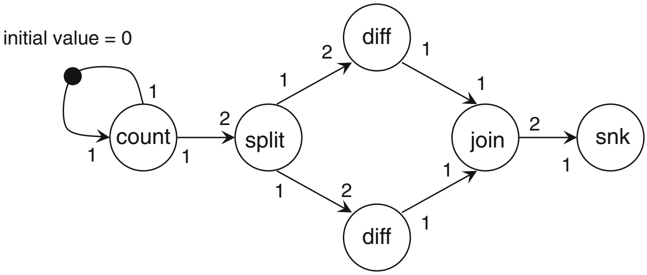


Fig. 3.18 SDF graph for Problem 3.4

3.6 Problems

Problem 3.1. Add the following improvement to the FIFO queue design of Listing 3.1. First, make the initial FIFO size very small, for example two elements. Next, implement dynamic FIFO sizing: when the FIFO overflows upon `put_fifo`, double the allocated memory for the FIFO. Use dynamic memory allocation, such as `malloc` in C. Implement changes to `put_fifo` and `get_fifo` as needed to keep track of a variable data flow queue depth.

Problem 3.2. Using an accumulator actor, as derived in Problem 2.4, implement the following C program as a SDF graph. The graph has a single input token, in, and produces a single output token out, corresponding to the return value of the function.

```
int graph(int in) {
  int i, j, k = 0;
  for (i=0; i<10; i++)
    for (j=0; j<10; j++)
      k = k + j * (i + in);
  return k;
}
```

Problem 3.3. Assume a C function with only expressions on scalar variables (no pointers) and for-loops. Show that such a C function can be translated to a SDF graph if and only if the loop-bound expressions are manifest, that is, they only depend on compile-time constant values and loop counters.

Problem 3.4. Using a PASS analysis, find a stable firing rate for each actor of the SDF graph in Fig. 3.18. The six actors in this graph have the following functionality. `count` increments a token at the input, and produces a copy of the incremented value on each of its outputs. `split` reads two tokens at the input and distributes these tokens over each output. `diff` reads two tokens at the input and produces the difference of these tokens (first minus last) at the output. `join` reads a token on each input and produces a merged stream. `join` is the complement of `split`. `snk` prints the input token.

Problem 3.5. Using the quickthreads API defined earlier, create a data flow simulation for the SDF graph shown in Fig. 3.18.

Problem 3.6. Optimize the SDF graph shown in Fig. 3.18 to a single C function by implementing the schedule at compile-time, and by optimizing the FIFO queues into single variables.

Problem 3.7. Convert the SDF graph in Fig. 3.18 to a single-clock hardware implementation. Perform first a multi-rate expansion. You can assume that the snk actor is implemented as a system-level output port.

Problem 3.8. Optimize the C program in Listing 3.6 with inlining. Implement and run the cosimulation before and after the optimization, and evaluate the gain in cycle count.

Chapter 4
Analysis of Control Flow and Data Flow

4.1 Data and Control Edges of a C Program

In the previous chapter, we discussed the data flow model of computation. Fundamental to this model is the decomposition of a system into individual nodes (*actors*), which communicate through unidirectional, point-to-point channels (*queues*). The resulting system model is represented as a graph. The data flow model of computation describes concurrent computations. We discussed techniques to create a hardware or a software implementation starting from the same data flow model.

Our objective in this chapter is to think of a C program in a similar target-independent fashion. For a software designer, a C program is software, and sequential. For a hardware-software codesigner however, a C program may be hardware or software, depending on the requirements and needs of the application. Obviously one cannot make a direct conversion of C into hardware – a major roadblock is that hardware is parallel by nature, while C is sequential.

However, there may be a different way of looking at a C program. We can think of a C program as a high-level description of the *behavior* of an implementation, without specifically pinning down the exact implementation details. Thus, we seek to understand the structure of the C program in terms of the individual operations it contains, and in terms of the relations between those operations.

We define two types of relationships between the operations of a C program: data edges and control edges. At first glance, data edges and control edges are quite similar.

> A **data edge** is a relation between two operations, such that data which is produced by one operation is consumed by the other.
>
> A **control edge** is a relation between two operations, such that one operation has to execute after the other.

P.R. Schaumont, *A Practical Introduction to Hardware/Software Codesign*,
DOI 10.1007/978-1-4614-3737-6_4, © Springer Science+Business Media New York 2013

This looks similar, but it's not identical. Let's try to identify data edges and control edges in the following C function, which finds the maximum of two variables.

```
int max(int a, b) {
  int r;
  if (a > b)
    r = a;
  else
    r = b;
  return r;
}
```

This function contains two assignment statements and an if-then-else branch. For the purpose of this analysis, we will equate statements in C with 'operations'. In addition, we define the entry points and exit points of the function as two additional operations. Therefore, the max function contains five operations:

```
int max(int a, b) {      // operation 1 - enter the function
  int r;
  if (a > b)             // operation 2 - if-then-else
    r = a;               // operation 3
  else
    r = b;               // operation 4
  return r;              // operation 5 - return max
}
```

To find the control edges in this function, we need to find the sequence of operations in this function. There may be more than one possible sequence, when the program contains if-then-else statements and loops. The control edges should capture all possible paths. In the example, operation 2 will always execute after operation 1. Therefore, there is a control edge from operations 1 to 2. An if-then-else statement introduces two control edges, one for each of the possible outcomes of the if-then-else test. If `a > b` is true, then operation 3 will follow operation 2, otherwise operation 4 will follow operation 2. There is a control edge from operation 2 to each of operations 3 and 4. Finally, operation 5 will follow either operation 3 or 4. There is a control edge from each of operations 3 and 4 to operation 5. Summing up, finding control edges corresponds to finding the possible execution paths in the C program, and linking up the operations in these execution paths with edges.

To find the data edges in this function, we examine the data production/consumption patterns of each operation.

```
int max(int a, b) {      // operation 1 - produce a, b
  int r;
  if (a > b)             // operation 2 - consume a, b
    r = a;               // operation 3 - consume a and (a>b),
                         //               produce r
  else
    r = b;               // operation 4 - consume b and (a>b),
```

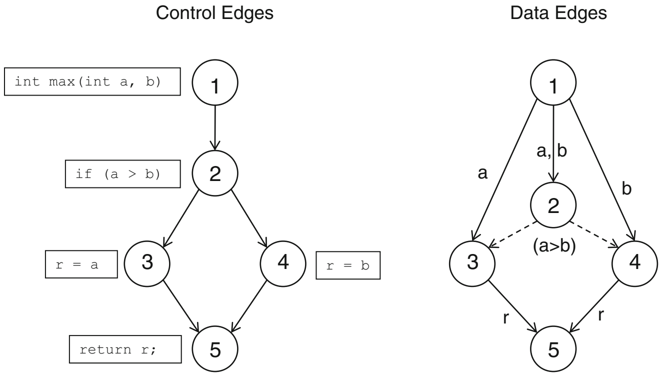


Fig. 4.1 Control edges and data edges of a simple C program

```
                        //               produce r
  return r;             // operation 5 - consume r
}
```

The data edges are defined between operations of corresponding production/consumption. For example, operation 1 defines the value of `a` and `b`. Several operations will make use of those values. The value of `a` is used by operations 2 and 3. Therefore there is a data edge from operations 1 to 2, as well as a data edge from operations 1 to 3. The same goes for the value of `b`, which is produced in operation 1 and consumed in operations 2 and 4. There is a data edge for `b` from operations 1 to 2, as well as from operations 1 to 4.

Control statements in C may produce data edges as well. In this case, the if-then-else statement evaluates a flag, and the *value* of that flag is needed before subsequent operations can execute. For example, operation 3 will only execute when the conditional expression `(a>b)` is true. We can think of a boolean flag carrying the value of `(a>b)` from operations 2 to 3. Similarly, operation 4 will only execute when the conditional expression `(a>b)` is false. There is a boolean flag carrying the value of `(a>b)` from operations 2 to 4.

The data edges and control edges of the operations from the `max` function can be arranged in a graph, where each operation represents a node. The result is shown in Fig. 4.1, and it represents the control flow graph (CFG) and the data flow graph (DFG) for the program. Control edges express a general relation between two nodes (operations), while data edges express a relation between two nodes for a specific variable. Therefore, data edges are labeled with that variable.

We will now explore the properties of control edges and data edges more carefully, and evaluate how the CFG and DFG can be created systematically for a more complex C program.

4.2 Implementing Data and Control Edges

In the context of hardware-software codesign, the implementation target of a C program may be either hardware or software. The data edges and control edges of the C program give important clues on the implementation alternatives for that C program.

- A data edge reflects a requirement on the flow of information. If you change the flow of information, you change the meaning of the algorithm. For this reason, a data edge is a fundamental property of the behavior expressed in that C program.
- A control edge, on the other hand, is a consequence of the execution semantics of the program language, but it is not fundamental to the behavior expressed in that C program.

In hardware-software codesign, we are looking to design the architecture that fits best to a given algorithm. Even though we may start from a C program, the target of this program may not be a processor. It may be a processor with a coprocessor, or a full hardware implementation. One question then inevitably arises: what are the important parts of a C program that will be present in any implementation of that program? The answer to this question is given by the control edges and data edges of the program, and it is summarized as follows.

A **data edge** must always be implemented regardless of the underlying architecture.

A **control edge** may be removed if the underlying architecture can handle the resulting concurrency.

In other words, control edges can be removed, by building sufficient parallelism into the underlying architecture. For example, modern microprocessors are able to run multiple instructions in parallel, even when they would belong to two different sequential C statements. These microprocessors are able to analyze and modify the control edges of the flow of instructions at runtime. They do this such that the data edges within that instruction flow are never broken.

Here is another example. The following function adds up three numbers using multiple operations.

```
int sum(int a, b, c) {  // operation 1
  int v1;
  v1 =  a + b;          // operation 2
  v2 = v1 + c;          // operation 3
  return v2;            // operation 4
}
```

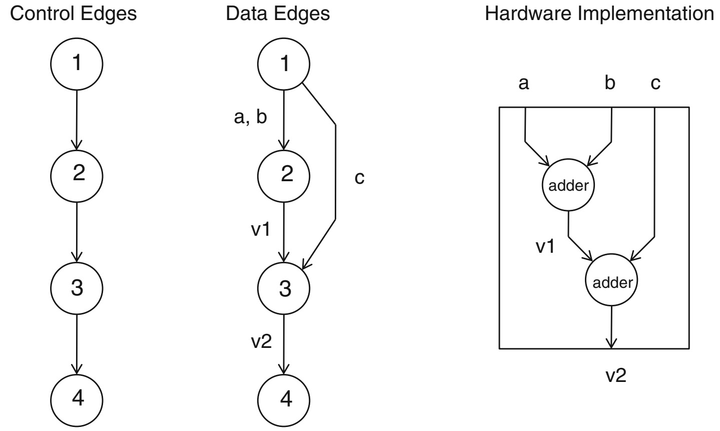


Fig. 4.2 Hardware implementation of a chained addition

It is straightforward to draw a fully parallel hardware implementation of this function. This implementation is shown, together with the data flow graph and control flow graph of the function, in Fig. 4.2. The similarity between the set of *data* edges and the interconnection pattern of the hardware is obvious. The control edges, however, carry no meaning for the hardware implementation, since hardware is parallel. The structure shown on the right of Fig. 4.2 will complete the addition in a single clock cycle.

The next section will introduce a systematic method to derive the control flow graph and the data flow graph of C programs.

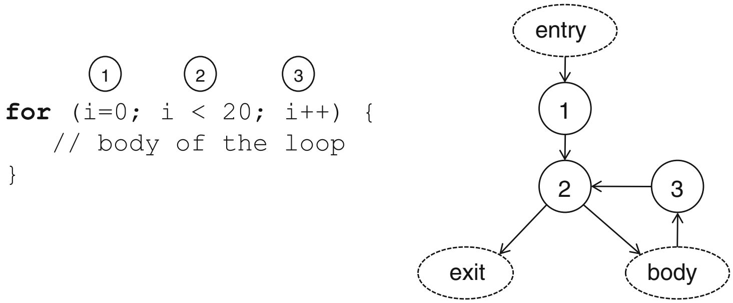
4.3 Construction of the Control Flow Graph

A C program can be systematically converted into an intermediate representation called a Control Flow Graph (CFG). A CFG is a graph that contains all the control edges of a program. Each node in the graph represents a single operation (or C statement). Each edge of the graph indicates a control edge, i.e. an execution order for the two operations connected by that edge.

Since C executes sequentially, this conversion is straightforward. However, some cases require further attention. Control statements (such as loops) may require multiple operations. In addition, when decision-making is involved, multiple control edges may originate from a single operation.

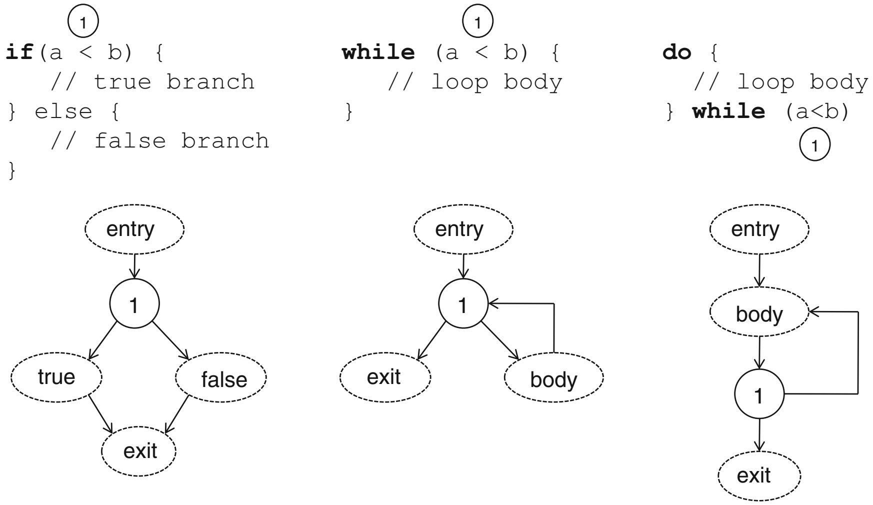
Consider the `for` loop in C, as illustrated next.

```
for (i=0; i < 20; i++) {
  // body of the loop
}
```



```
for (i=0; i < 20; i++) {
   // body of the loop
}
```

Fig. 4.3 CFG of a `for` loop



```
if(a < b) {
   // true branch
} else {
   // false branch
}
```

```
while (a < b) {
   // loop body
}
```

```
do {
  // loop body
} while (a<b)
```

Fig. 4.4 CFG of `if-then-else`, `while-do`, `do-while`

This statement includes four distinct parts: the loop initialization, the loop condition, the loop-counter increment operation, and the body of the loop. The `for` loop thus contributes three operations to the CFG, as shown in Fig. 4.3. The dashed nodes in this figure (`entry`, `exit`, `body`) represent other parts of the C program, each of which is a complete single-entry, single-exit CFG.

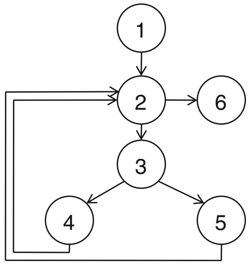
The `do-while` loop and the `while-do` loop are similar iterative structures. Figure 4.4 illustrates a template for each of them, as well as for the `if-then-else` statement.

As an example, let's create the CFG of the following C function. This function calculates the Greatest Common Divisor (GCD) using Euclid's algorithm.

```
int gcd(int a, int b) {
  while (a != b) {
    if (a > b)
      a = a - b;
```

Fig. 4.5 CFG of the CGD program

```
1: int gcd(int a, int b) {
2:   while (a != b) {
3:     if (a > b)
4:       a = a - b;
       else
5:       b = b - a;
     }
6:   return a;
   }
```



```
    else
      b = b - a;
  }
  return a;
}
```

To construct the CFG of this program, we convert each statement to one or more operations in the CFG, and then connect the operations using control edges. The result of this conversion is shown in Fig. 4.5.

In a CFG it is useful to define a *control path*, a path between two nodes in the CFG. For example, each non-terminating iteration of the `while` loop of the C program will follow either the path 2-3-4-2 or else 2-3-5-2. Control paths will be important in the construction of the data flow graph (DFG), which is discussed next.

4.4 Construction of the Data Flow Graph

A C program can be systematically converted into a data structure called a Data Flow Graph (DFG). A DFG is a graph that reflects all the data edges of a program. Each node in the graph represents a single operation (or C statement). Each edge of the graph indicates a data edge, i.e. a production/consumption relationship between two operations in the program.

Obviously, the CFG and the DFG will contain the same set of nodes. Only the edges will be different. Since a variable in C can be written-to/read-from an arbitrary number of times, it can be difficult to find matching read-write pairs in the program. The easiest way to construct a DFG is to first construct the CFG, and then use the CFG in combination with the C program to derive the DFG. The trick is to trace control paths, and at the same time identify corresponding read- and write operations of variables.

Let's assume that we're analyzing programs without array expressions and pointers; we will extend our conclusions later to those other cases as well. The procedure to recover the data edges related to assignment statements is as follows.

1. In the CFG, select a node where a variable is used as an operand in an expression. Mark that node as a read-node.
2. Find the CFG nodes that assign that variable. Mark those nodes as write-nodes.

3. If there exists a direct control path from a write-node into a read-node that does not pass through another write-node, then you have identified a data edge. The data edge originates at the write-node and ends at the read-node.
4. Repeat the previous steps for all variable-read operations in every node.

This procedure identifies all data edges related to assignment statements, but not those originating from conditional expressions in control flow statements. However, these data edges are easy to find: they originate from the condition evaluation and affect all the operations whose execution depends on that condition.

Let's derive the data flow graph of the GCD program given in Fig. 4.5. According to the procedure, we pick a node where a variable is read. For example, node 5 in the CFG reads variables a and b.

```
b = b - a;
```

First, concentrate on the b operand. We need to find all nodes that write into variable b. In the CFG, we can trace precedessor nodes for this node until we hit one that writes into variable b. The predecessors of node 5 include: node 3, node 2, node 1, node 4, and node 5. Both node 1 and 5 write into b. In addition, there is a path from node 1 to 5 (e.g. 1-2-3-5), and there is also a path from node 5 to 5 (e.g. 5-2-3-5). In each of these paths, no other nodes write into b apart from the final node 5. Thus, there is a data edge for variable b from node 1 to 5 and from node 5 to 5. Starting from the same read-node 5, we can also find all predecessors that define the value of operand a. In this case, we find that nodes 1 and 4 write into variable a, and that there is a direct path from node 1 to 5, as well as from node 4 to 5. Therefore, there is a data edge for variable a from node 1 to 5, and from node 4 to 5.

To complete the set of data edges into node 5, we also need to identify all conditional expressions that affect the outcome of node 5. Considering the control statements in this function, we see that node 5 depends on the condition evaluated in node 3 (a > b) as well as the condition evaluated in node 2 (a != b). There is a data edge from each of nodes 2 and 3 to node 5, carrying the outcome of this condition. The collection of all data edges into node 5 can now be annotated into the DFG, resulting in the partial DFG of Fig. 4.6.

We can repeat this procedure for each other node of the graph in order to construct the complete DFG. The result of this analysis is shown in Fig. 4.7. This graph does not contain the data edges originating from conditional expressions.

How do we draw a DFG of a program with pointers and arrays? There are possible several approaches, and they depend on the objectives of the analysis and the level of detail desired.

First, observe that an indexed variable is not really different from a scalar variable as long as we can exactly determine the value of the index during the data-flow analysis. Similarly, data edges resulting from pointers are easy to find if we can exactly determine the value of the pointer. However, in practice, this may be difficult or impossible. An indexed variable may have a complex index expression that depends on multiple loop counters, or the index expression may contain a variable which is unknown at compile time.

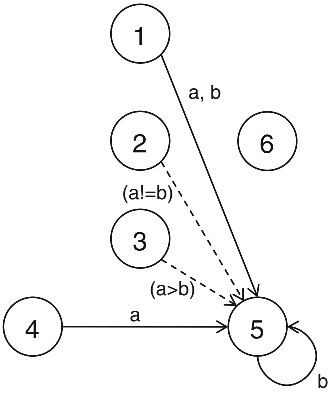


Fig. 4.6 Incoming data edges for node 5 in the CGD program

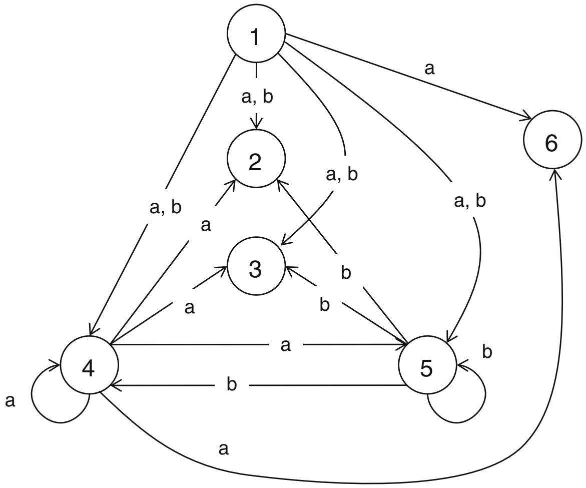


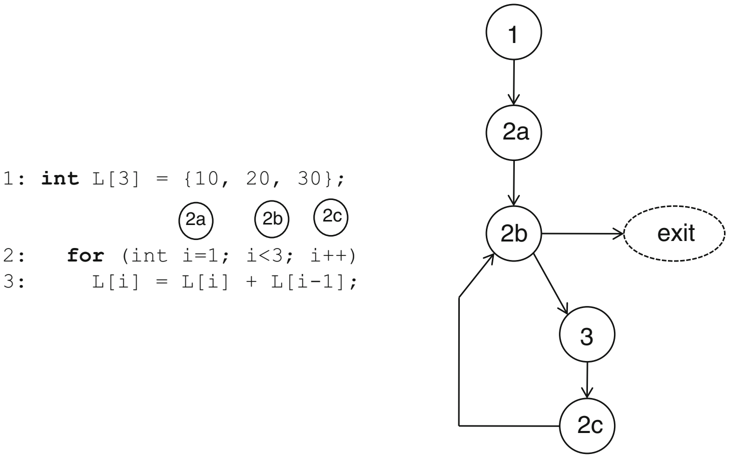
Fig. 4.7 Data edges for all nodes in the GCD program, apart from edges carrying condition variables

We may be able to relax the analysis requirements, and simplify the data-flow analysis. In many applications, the upper bound and lower bound of an index expression is known. In that case, we may consider any *write* operation into the range of indices as a single write, and any *read* operation into the range of indices as a single read. For cases when an entire range of indices would map into a single memory (a single register file, or a single-port RAM memory), this type of data-flow analysis may be adequate.

We illustrate this approach using the following example. The CFG of the following loop is shown in Fig. 4.8.

```
int L[3] = {10, 20, 30};
for (int i=1; i<3; i++)
  L[i] = L[i] + L[i-1];
```

To create a DFG for this program, proceed as before. For each node that reads from a variable, find the nodes that write into that variable over a direct path in the



```
1: int L[3] = {10, 20, 30};

2:   for (int i=1; i<3; i++)
3:     L[i] = L[i] + L[i-1];
```

Fig. 4.8 CFG for a simple loop with an indexed variable

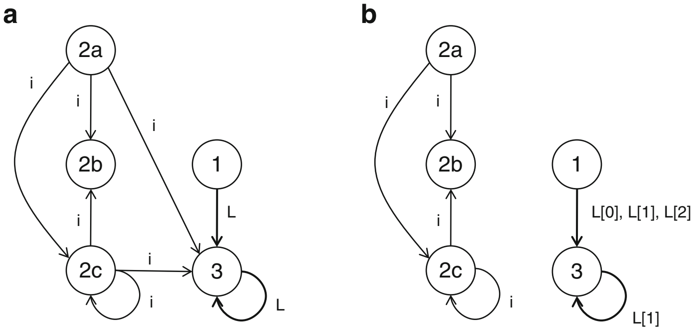


Fig. 4.9 DFG for a simple loop with an indexed variable

CFG. As discussed above, we can handle the analysis of the indexed variable `L` in two different ways. In the first approach, we look upon `L` as a single monolithic variable, such that a read from any location from `L` is treated as part of the same data edge. In the second approach, we distinguish individual locations of L, such that each location of L may contribute to a different data edge. The first approach is illustrated in Fig. 4.9a, while the second approach is illustrated in Fig. 4.9b.

When the individual locations of `L` cannot be distinguished with a data edge, additional information is needed to extract the entry of interest. For this reason, node 3 in Fig. 4.9a has an additional data edge to provide the loop counter `i`. Thus, in Fig. 4.9a, reading entry `L[i]` means: read all the entries of `L` and then select one using `i`. In Fig. 4.9b, reading entry `L[i]` means three different read operations, one for each value of `i`.

Index analysis on arbitrary C programs quickly becomes very hard to solve. Yet, hardware-software codesigners often only have a C program to start their design with. Insight into the data-flow of a complex C program is essential for a successful hardware-software codesign.

This concludes an introduction to control-flow and data-flow analysis of C programs. The next section shows an application for the techniques we've covered so far. By deriving the CFG and the DFG, we can translate simple C programs systematically into hardware. This technique is by no means a universal mechanism to translate C; its' purpose is to clarify the meaning of control edges and data edges.

4.5 Application: Translating C to Hardware

A nice application of analysis of data-flow and control-flow in a C program is the systematic translation of C into hardware. This problem is very complex, when our objective is to solve the translation of general C. Therefore, we will focus on a simplified version of this problem. We show how to translate one particular flavor of C into one particular flavor of hardware circuit. We are making the following two assumptions.

- We will translate only scalar C code (no pointers and no arrays).
- We implement each C statement in a single clock cycle.

4.5.1 Designing the Datapath

Starting from the C program, we first create the CFG and the DFG. Next, we use the data edges and control edges to implement the hardware. The data edges will help us to define the datapath components, and their connectivity. The control edges will help us to define the control signals used by the datapath. With the CFG and DFG available, the following rules will define the implementation of the hardware datapath.

1. Each variable in the C program is translated into a register with a multiplexer in front of it. The multiplexer is needed when multiple sources may update the register. By default, the register will update itself. The selection signals of the multiplexer will be driven by the controller.
2. For each C expression embedded in a node of the CFG, create an equivalent combinational circuit to implement that expression. For example, if a node in the CFG corresponds to the C statement `a = b - a`, then the C expression embedded in that statement is `b - a`. The combinational circuit required to implement this expression is a subtractor. Conditional expressions generate datapath elements, too. The outputs of these expressions become the flags used by the hardware controller of this datapath.
3. The datapath circuit and the register variables are connected based on the data edges of the DFG. Each assignment operation connects a combinational

```
int gcd(int a, int b) {
  while (a != b) {
    if (a > b)
      a = a - b;
    else
      b = b - a;
  }
  return a;
}
```

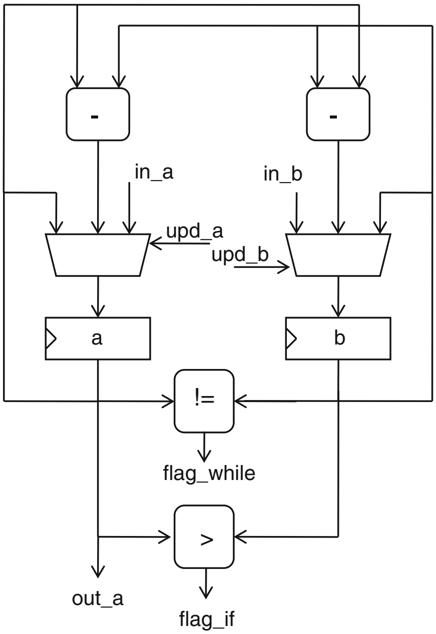


Fig. 4.10 Hardware implementation of GCD datapath

circuit with a register. Each data edge connects a register with the input of a combinational circuit. Finally, we also connect the system-inputs and system-outputs to inputs of datapath circuits and register outputs respectively.

The GCD program can now be converted into a hardware implementation as follows. We need two registers, for each of the variables `a` and `b`. The conditional expressions for the `if` and `while` statement need an equality-comparator and a bigger-then comparator. The subtractions `b-a` and `a-b` are implemented using a subtractor. The connectivity of the components is defined by the data edges of the DFG.

The resulting datapath has two data inputs (`in_a` and `in_b`), and one data output (`out_a`). The circuit requires two control variables (`upd_a` and `upd_b`) to operate, and it produces two flags (`flag_while` and `flag_if`). The control variables and the flags are the ouputs and inputs, respectively, of the controller of this datapath, see Fig. 4.10.

4.5.2 Designing the Controller

How do we create the controller for this datapath such that it implements the GCD algorithm? This control information is present in the C program, and is captured

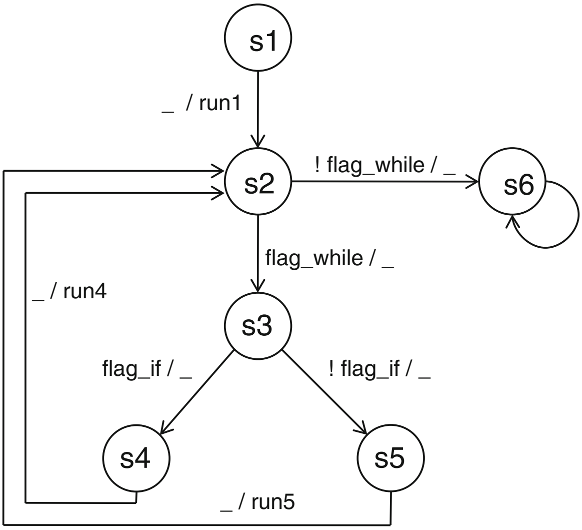


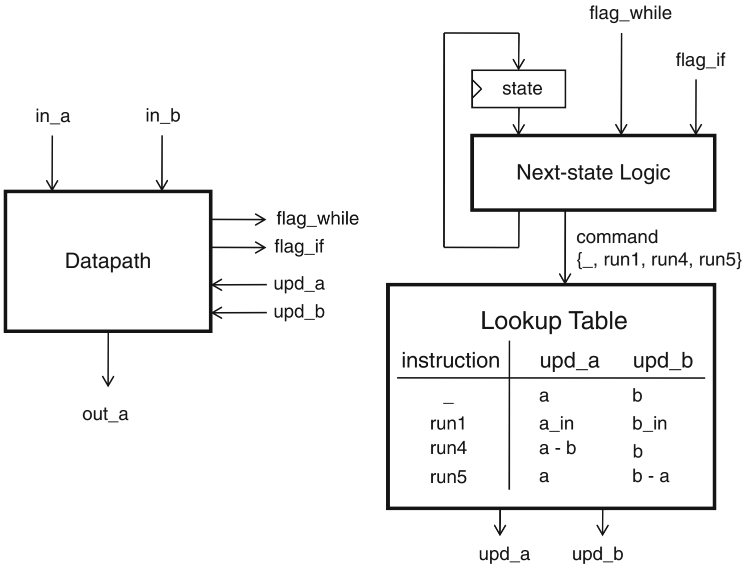
Fig. 4.11 Control specification for the GCD datapath

in the CFG. In fact, we can translate the CFG almost directly into hardware, by considering it to be a finite state machine (FSM) specification.

A finite-state machine (FSM) specification for the GCD algorithm is shown in Fig. 4.11. The correspondence with the CFG is obvious. Each of the transitions in this FSM takes one clock cycle to complete. The activities of the FSM are expressed as condition/command tuples. For example, `_/run1` means that during this clock cycle, the condition flags are don't-care, while the command for the datapath is the symbol `run1`. Similarly, `flag_while/_` means that this transition is conditional on `flag_while` being true, and that the command for the dapath is a hold operation. A *hold operation* is one which does not change the state of the datapath, including registers. The command set for this FSM includes `(_, run1, run4, run5)`. Each of these symbols represents the execution of a particular node of the CFG. The datapath control signals can be created by additional decoding of these command signals. In this case of the GCD, the datapath control signals consist of the selection signals of the datapath multiplexers.

A possible implementation of the GCD controller is shown in Fig. 4.12. Each clock cycle, the controller generates a new command based on the current state and the value of `flag_while` and `flag_if`. The commands `run1`, `run4` and `run5` are decoded into `upd_a` and `upd_b`. The table in Fig. 4.12 indicates how each command maps into these control signals. The resulting combination of datapath and finite state machine, as illustrated in Fig. 4.12 is called a Finite State Machine with Datapath (FSMD). This concept is central to custom hardware design, and we will discuss design and modeling of FSMD in detail in Chap. 5.

The operation of this hardware circuit is illustrated with an example in Table 4.1. Each row of the table corresponds to one clock cycle. It takes eight clock cycles to evaluate the greatest common divisor of 6 and 4.



instruction	upd_a	upd_b
_	a	b
run1	a_in	b_in
run4	a - b	b
run5	a	b - a

Fig. 4.12 Controller implementation for the GCD datapath

Table 4.1 Operation of the hardware to evaluate GCD(4,6)

Cycle	a	b	State	flag_if	flag_while	Next state	upd_a	upd_b
1	_	_	s1	_	_	s2	in_a	in_b
2	6	4	s2	1	1	s3	a	b
3	6	4	s3	1	1	s4	a	b
4	6	4	s4	1	1	s2	a-b	b
5	2	4	s2	0	1	s3	a	b
6	2	4	s3	0	1	s5	a	b
7	2	4	s5	0	1	s2	a	b-a
8	2	2	s2	0	0	s6	a	b
9	2	2	s6	_	_	s6	a	b

In conclusion, the DFG and CFG of a C program can be used to create and implement a hardware circuit. Of course, there are many sub-optimal elements left. First, we did not address the use of arrays and pointers. Second, the resulting implementation in hardware is not very impressive: the resulting parallelism is limited to a single C statement per clock cycle, and operations cannot be shared over operator implementations. For instance, two substractors are implemented in hardware in Fig. 4.10, but only one is used at any particular clock cycle.

4.6 Single-Assignment Programs

Converting C programs into hardware at one cycle per C-statement is not very impressive, in particular because most microprocessors can already do that. Can we do any better than this? That is, can we create a translation strategy that will allow the execution of multiple C statements per clock cycle? To answer this question, we need to understand the limitations of our current translation strategy, as described in Sect. 4.5.

In that approach, each C statement takes a single clock cycle to execute because each variable is mapped into a register. A data value takes a full clock cycle to propagate from the input of a register to the output. Therefore, each variable assignment takes a full clock cycle to take effect; it takes a full clock cycle before the value of a variable assignment is available at the corresponding register output. Thus, the mapping of each variable into a register, which by itself seems a sensible decision, also introduces a performance bottleneck. If we want to run at a faster pace then one statement per clock cycle, we will need to revise this variable-to-register mapping strategy.

The above observation triggers another question, namely: why did we map each variable into a register in the first place? The reason is, of course, to make the evaluation of expressions, and the design of control, easy. By mapping each variable into a register, it is as if we're concentrating all data-flow edges related to a given variable to go through a single, global storage location, so we always know where to find the value of a given variable. This strategy, however, hurts performance.

We can do better than that with the following technique. A C program can be translated into a *single-assignment program*. The key property of a single-assignment program is exactly what its' name refers says: each variable in that program is assigned only a single time within a single lexical instance of the program. Let's illustrate the conversion with a simple example. Assume that we have a C snippet that looks as follows.

```
a = a + 1;
a = a * 3;
a = a - 2;
```

This section of code contains three assignments on a. Using our previous strategy, we would need three clock cycles to execute this fragment. Instead, we can rewrite this program so that each variable is assigned only once. This requires the introduction of additional variables.

```
a2 = a1 + 1;
a3 = a2 * 3;
a4 = a3 - 2;
```

The difference with the previous program is that each assignment is matched by a different read operation. In other words, the single-assignment form of the program visualizes the data edges of the program in the source code: assigning a given variable indicates the start of a data edge, while reading the same variable indicates the end of the data edge.

After a program is in single-assignment form, the register-assignment strategy can be improved. For instance, in the previous example, the cycle count may be reduced by mapping a2 and a3 to a wire, while keeping a4 in a register. This would group the three C statements in a single clock cycle.

In converting C programs to single-assignment form, all assignments to a variable must be taken into account. In particular, when variables are assigned under different control conditions, or in different levels of a loop nesting structure, the single-assignment form may become ambiguous. Consider the following example: a loop which makes the sum of the numbers 1–5.

```
a = 0;
for (i = 1; i < 6; i++)
  a = a + i;
```

In the single-assignment form, the assignments to a can be made unique, but it remains unclear what version of a should be read inside of the loop.

```
a1 = 0;
for (i = 1; i < 6; i++)
  a2 = a + i;  // which version of a to read?
```

The answer is that both a1 and a2 are valid solutions for this program: it depends on the iteration within the loop. When we first enter the loop, we would write:

```
  a2 = a1 + 1;
```

After the first loop iteration, we would write instead:

```
  a2 = a2 + i;  // when i > 1
```

To resolve this ambiguity, single-assignment programs use a merge function, an operation that can merge multiple data edges into one. We can introduce a new variable a3 to hold the result of the merge function, and now formulate the program into single-assignment form as follows.

```
a1 = 0;
for (i = 1; i < 6; i++) {
  a3 = merge(a2, a1);
  a2 = a3 + i;
```

The hardware equivalent of the merge function would be a multiplexer, under control of an appropriate selection signal. In this case, (i==0) would be an appropriate selection signal. The above translation rules can be used to more complicated programs as well. For example, the GCD program can be converted as follows.

```
int gcd(int a, int b) {
  while (a != b) {
    if (a > b)
      a = a - b;
    else
      b = b - a;
  }
  return a;
}
```

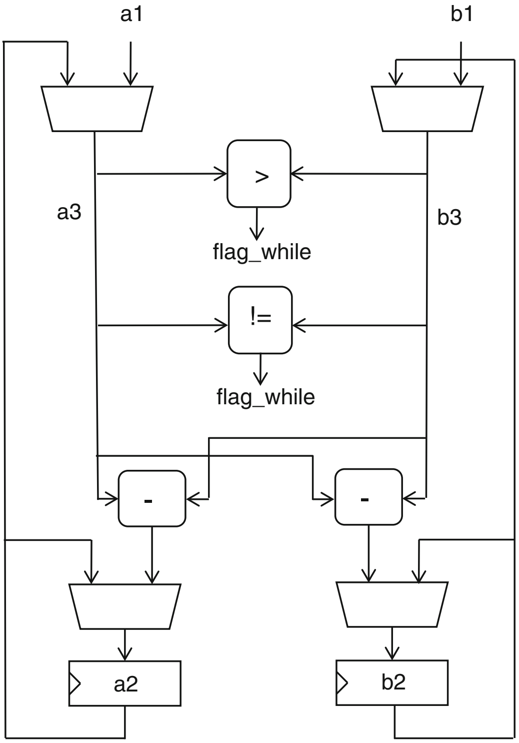


Fig. 4.13 GCD datapath from single-assignment code

The equivalent single-assignment form of the GCD is shown below. The conditional expression in the `while` statement uses variables from either the function input or else the body of the loop. For this reason, the conditional expression uses the `merge` function as an operand.

```
int gcd(int a1, int b1) {
  while (merge(a1, a2) != merge(b1, b2)) {
    a3 = merge(a1, a2);
    b3 = merge(b1, b2);
    if (a3 > b3)
      a2 = a3 - b3;
    else
      b2 = b3 - a3;
  }
  return a2;
}
```

A single assignment program such as this one is valuable because it visualizes the data edges in the source code of the program, making the connection with hardware more obvious. Furthermore, the `merge` functions can be mapped into multiplexers in hardware. A possible datapath corresponding to this single-assignment version of GCD is shown in Fig. 4.13. This datapath looks very much like the previous design (Fig. 4.10), but this design was derived from a C program with four assignments.

Only a2 and b2 were mapped into a register, while other variables are simply wires. The design in Fig. 4.13 executes multiple C statements per clock cycle.

Of course, construction of a single-assignment program is only one step in mapping a C program to hardware. Additional compiler techniques, which are outside this book's scope, are also needed. For example, software compilers use the single-assignment transformation extensively to implement advanced code optimizations (See further reading).

4.7 Summary

In this chapter we discussed the data flow and control flow properties of a C program. These properties can be modeled into two graph structures, called the DFG (data flow graph) and CFG (control flow graph). The DFG and CFG can be derived systematically starting from the C source. The data flow and control flow of a program help the designer to understand the implementation alternatives for that program. We showed that data flow is preserved over different implementations in hardware and software, while control flow may change drastically. Indeed, a sequential or a parallel implementation of a given algorithm may have a very different control flow. We used this insight to define a simple mechanism to translate C programs into hardware.

4.8 Further Reading

The material discussed in this chapter can typically be found, in expanded from, in a textbook on compiler construction such as Muchnick (1997) or Appel (1997). In particular, these books provide details on the analysis of the control flow graph and data flow graph.

High-level synthesis is a research area that investigates the automated mapping of programs written in C and other high-level languages into lower-level architectures. In contrast to compilers, which target a processor with a fixed architecture, high-level synthesis does support some freedom in the target architecture. High-level synthesis has advantages and limitations; proponents and opponents. Refer to Gupta et al. (2004) to see what can be done; read Edwards (2006) as a reminder of the pitfalls.

During our discussion on the mapping of C programs into hardware, we did explicitly rule out pointers and arrays. In high-level synthesis, the design problems related to implementation of memory elements are collected under the term *memory management*. This includes for example the systematic mapping of array variables into memory elements, and the efficient conversion of indexed expressions into memory addresses. Refer to Panda et al. (2001) for an introduction to memory management issues.

The original work on Static Single Assignment (SSA) was by Cytron et al. (1991). A discussion on how the SSA form can assist in the translation of C software into hardware may be found in Kastner et al. (2003).

4.9 Problems

Problem 4.1. Do the following for the C program in Listing 4.1.

Listing 4.1 Program for Problem 4.1

```
int addall(int a, int b, int c, int d) {
  a = a + b;
  a = a + c;
  a = a + d;
  return a;
}
```

(a) Derive and draw the CFG and the DFG.
(b) The *length* of a path in a graph is defined as the number of edges in that path. Find the longest path in the DFG.
(c) Rewrite the program in Listing 4.1 so that the maximal path length in the DFG decreases. Assume that you can do only a single arithmetic operation per C statement. Draw the resulting DFG.

Problem 4.2. Draw the CFG and the DFG of the program in Listing 4.2. Include all control dependencies in the CFG. Include the data dependencies for the variables a and b in the DFG.

Listing 4.2 Program for Problem 4.2

```
int count(int a, int b) {
  while (a < b)
    a = a * 2;
  return a + b;
}
```

Problem 4.3. Design a datapath in hardware for the program shown in Listing 4.3. Allocate registers and operators. Indicate control inputs required by the data-path, and condition flags generated by the datapath.

Listing 4.3 Program for Problem 4.3

```
unsigned char mysqrt(unsigned int N) {
  unsigned int x,j;
  x = 0;
  for(j= 1<<7; j != 0; j>>=1) {
    x = x + j;
    if( x*x > N)
```

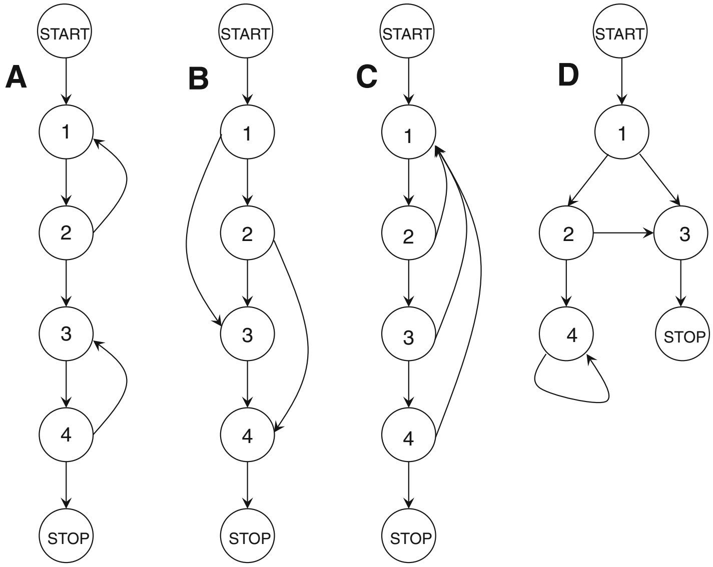


Fig. 4.14 Four CFG for Problem 4.4

```
      x = x - j;
  }
  return x;
}
```

Problem 4.4. A well-structured C program is a program that only contains the following control statements: if-then-else, while, do-while, and for. Consider the four CFG in Fig. 4.14. Which of the these CFG does correspond to a well-structured C program? Note that a single node in the CFG may contain more than a single statement, but it will never contain more than a single decision point.

Problem 4.5. Draw the DFG for the program in Listing 4.4. Assume all elements of the array a[ ] to be stored in a single resource.

Listing 4.4 Program for Problem 4.5

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int findmax() {
  int max, i;

  max = a[0];
  for (i=1; i<10; i++)
    if (max < a[i])
      max = a[i];

  return max;
}
```

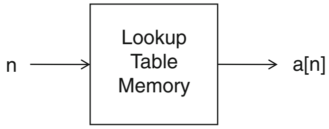


Fig. 4.15 A single-port read-only memory, used to solve Problem 4.6

Problem 4.6. Design a hardware implementation (datapath and controller) for the program in Listing 4.4. Assuming that the elements of array `a[ ]` are all stored in a memory with a single read port. Figure 4.15 illustrates such a memory. The time to lookup an element is very short; thus, you can think of this memory as a combinational element.

Problem 4.7. Convert the program in Listing 4.3 to single-assignment form.

Problem 4.8. This problem requires access to a GNU Compiler (gcc) version 4.0 or above. Start by writing up the Listing of Problem 4.3 in a file can call the file mysqrt.c.

(a) Compile this function using the following command line.

```
gcc -c -fdump-tree-cfg mysqrt.c
```

The compiler generates an object file `mysqrt.o`, as well as a file with debug information. Under `gcc 4.0.2`, the name of that file is `mysqrt.c.t13.cfg`.

Open `mysqrt.c.t13.cfg` in a text editor. This is a textual representation of a CFG as produced by the GCC compiler. Compare this CFG to one you would draw by hand. In particular, comment on the following two observations: (1) Nodes in a CFG can be grouped together when they all belong to a single path of the CFG with a single exit point. (2) `goto` and `if-then-else` are adequate to capture all control statements in C (such as `for`, `while`, and so on).

(b) Compile this function using the following command line. `O2` turns on the compiler optimizer, so that GCC will try to produce better code.

```
gcc -c -O2 -fdump-tree-ssa mysqrt.c
```

The compiler generates an object file, and a file with debug information. Under `gcc 4.0.2`, the name of that file is `mysqrt.c.t16.ssa`.

Open `mysqrt.c.t16.ssa` in a text editor. This is a textual representation of the SSA as produced by the GCC compiler. Find the `merge` functions in this file, and compare the number and location of these functions in the CFG. Did you find the same number of `merge` functions in Problem 4.7? Do they have the same location?

Part II
The Design Space of Custom Architectures

This second part of this book describes various hardware architectures, each with a varying degree of flexibility and sophistication. The objective of this part is to build insight in the nature of design problems that come with a particular architecture, and what it means to *customize* the architecture. Starting from very simple cycle-based hardware models, we gradually add control structures to increase their flexibility. We will cover FSMD (Finite State Machine with Datapath), micro-programmed architectures, general-purpose embedded cores, and finally system-on-chip architectures. We will demonstrate that each of these machines makes a trade-off between flexibility and performance. More flexible solutions, of course, will require more programming.

Chapter 5
Finite State Machine with Datapath

5.1 Cycle-Based Bit-Parallel Hardware

In this chapter, we develop a model to systematically describe custom hardware consisting of a controller and a datapath. Together with the model, we will also learn to capture hardware designs into a language, called GEZEL. This section, and the next one, describes how to create datapath modules. Further sections will explain the control model and the integration of control and datapath into an FSMD.

We will create cycle-based hardware models. In such models, the behavior of a circuit is expressed in steps of a single clock cycle. This abstraction level is very common in digital design, and it is captured with the term *synchronous* design. We will design circuits with a single, global clock signal.

5.1.1 Wires and Registers

We start with the variables that are used to describe synchronous digital hardware. The following example shows a 3-bit counter.

Listing 5.1 A three-bit counter module

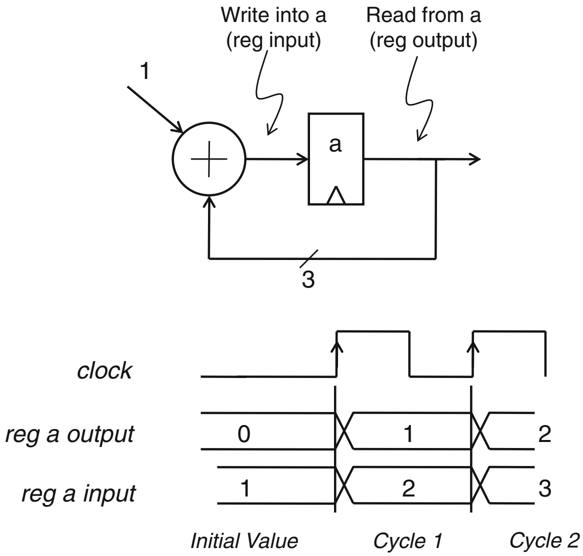
```
reg a : ns(3); // a three-bit unsigned register
always {
  a = a + 1;   // each clock cycle, increment a
}
```

This fragment represents a 3-bit register, called a. Each clock cycle, the value of a is incremented by one. Since a is a 3-bit value, the register will count from 0 to 7, and then wrap around. The initial value of a is not specified by this code. In GEZEL, all registers are initialized to zero.

Let's look closer at the expression '`a = a + 1`' and describe precisely what it means. The right-hand side of this expression, `a+1`, reads the value of the register

P.R. Schaumont, *A Practical Introduction to Hardware/Software Codesign*,
DOI 10.1007/978-1-4614-3737-6_5, © Springer Science+Business Media New York 2013

Fig. 5.1 Equivalent hardware for the 3-bit counter



a, and adds one to that value. The left-hand side of this expression assigns that value to a, and thus writes into the register. Figure 5.1 gives an equivalent circuit diagram for '`a = a + 1`', and it illustrates a key feature of a register: the input and the output of a register can have a different value. The input and the output are each connected to a different bus of wires. The timing diagram in Fig. 5.1 illustrates the circuit operation. Before the very first clock edge, the output of a is initialized to its initial value of 0. At the very first clock edge, the register will be incremented, which means that the value at the input of a is copied to the output of a.

Going back to Listing 5.1, we see that the `always` statement describes the activities in the model as a result of *updating* the registers. As shown in the diagram, this happens on the upgoing clock edge. This is the only time-related aspect of the model. The time required to execute `a+1` is unspecified, and the expression will be evaluated as soon as the output of the a register changes. Note also that Listing 5.1 does not contain an explicit clock signal: the simulation is completely defined using the semantics of a register variable.

Besides a register variable, there is another variable type, called a *signal*. A signal has the same meaning as a wire. A multi-bit signal corresponds to a bundle of wires. Listing 5.2 illustrates how a signal is created and used. A signal instantly takes up the value of the expression assigned to it. Thus, the value of b in Listing 5.2 will instantly reflect the value of `a+1`. The circuit diagram corresponding to this program looks identical to the diagram shown in Fig. 5.1. However, in this case, we have a specific name for the value at the *input* of the register a, namely the signal b.

Listing 5.2 Another 3-bit counter module

```
reg a : ns(3); // a three-bit unsigned register
sig b : ns(3); // a three-bit undersigned signal
```

```
always {
  b = a + 1;
  a = b;
}
```

A signal has no memory. When a signal value is used on the right-hand side of an expression, it will return the value assigned to the signal during that clock cycle. This has a particular effect on the program shown in Listing 5.2: the lexical order of expressions has no meaning. Only the data flow between reading/writing registers and signals is important. For example, the program in Listing 5.3 has exactly the same behavior as the program in Listing 5.2.

Listing 5.3 Yet another 3-bit counter module

```
reg a : ns(3); // a three-bit unsigned register
sig b : ns(3); // a three-bit unsigned signal
always {
  a = b;
  b = a + 1;
}
```

One can think of the difference between registers and signals also as follows. When a register is used as an operand in an expression, it will return the value assigned to that register during the *previous* clock cycle. When a signal is used as an operand in an expression, it will return the value assigned to that signal during the *current* clock cycle. Thus, registers implement communication across clock cycles, while signals implement communication within a single clock cycle.

Because a signal has no memory, it cannot have an initial value. Therefore, the value of a signal remains undefined when it is not assigned during a clock cycle. It is illegal to use an undefined signal as an operand in an expression, and the GEZEL simulator will flag this as a runtime error. Another case which is unsupported is the use of signals in a circular definition, such as for example shown in Listing 5.4. It is impossible to determine a stable value for `a` or `b` during any clock cycle. This type of code will be rejected as well by the GEZEL simulator with a runtime error message. In Sect. 5.6, we define the rules for a properly formed FSMD more precisely.

Listing 5.4 A broken 3-bit counter module

```
sig a : ns(3);
sig b : ns(3);
always {
  a = b;
  b = a + 1;  // this is not a valid GEZEL program!
}
```

5.1.2 Precision and Sign

In contrast to C, hardware registers and signals can have an arbitrary wordlength, from a single bit up to any value. It is also possible to mix multiple wordlengths in an expression. In addition, registers and signals can be unsigned or signed.

The wordlength and the sign of a register or signal are specified at the creation of that register or signal. The following example creates a 4 bit unsigned register `a` and a 3 bit signed signal `b`. The representation of `b` follows two's complement format: the weight of the most significant bit of `b` is negative.

```
reg a : ns(4);    // unsigned 4-bit value
sig b : tc(3);    // signed 3-bit value
```

In an expression, registers and signals of different wordlenghts can be combined. The rules that govern the precision of the resulting expression are as follows.

- The evaluation of an expression does not loose precision. All operands will automatically adapt their precision to a compatible wordlength.
- Assigning the result of an expression, or casting an expression type, will adjust the precision of the result.

Listing 5.5 Adding up 4 and 2 bit

```
1  reg a : ns(4);   // a four-bit unsigned number
2  sig b : ns(2);   // a two-bit unsigned number
3  always {
4    b = 3;         // assign 0b(011) to b
5    a = b + 9;     // add 0b(11) and 0b(1010)
6  }
```

As an example, the code shown in Listing 5.5 will store the value 12 in register `a`. Walking step by step through this code, the precision of each expression is evaluated as follows. First, the constant `3` needs to be assigned to `b`. A constant is always represented with sufficient bits to capture it as a two's complement number. In this case, you can express the constant `3` as a 3-bit two's complement number with the bit pattern `011`. When assigning this 3-bit value to `b`, the lower 2 bits will be copied, and the bitpattern in `b` becomes `11`. On line 5 of the code, we add the constant `9` to `b`. The bitpattern corresponding to the decimal constant `9` is `1001`. To add the bitpattern `11` and `1001` as unsigned numbers, we extend `11` to `0011` and perform the addition to find `1100`, or `12` in decimal. Finally, the bitpattern `1100` is assigned to `a`, which can accommodate all bits of the result.

When the length of an operand is extended, the rules of sign extension will apply. The additional bits beyond the position of the most significant bit are copies of the sign bit, in the case of two's complement numbers, or zeroes, in the case of unsigned numbers. In Listing 5.6, `a` is a 6-bit unsigned number, and `b` is a 2-bit *signed* number. After assigning the constant `3` to `b`, the value of `b` will be `-1`, and the bit pattern of `b` equals `11`. The result of subtracting `b` and `3` is `-4` in decimal,

Table 5.1 Operations in GEZEL and equivalent hardware implementation. `const` is a constant number

Operation	Operator	Implementation	Precedence
Addition	+	Adder	4
Subtraction	-	Subtractor	4
Unary minus	-	Subtractor	7
Multiplication	*	Multiplier	5
Right-shift	>> (variable)	Variable-shifter	0
Left-shift	<< (variable)	Variable-shifter	0
Constant right-shift	`>> const`	Wiring	4
Constant left-shift	`<< const`	Wiring	4
Lookup table	`A(n)`	Random logic	10
And	&	And-gate	2
Or	|	Or-gate	2
Xor	^	Xor-gate	3
Not	~	Not-gate	8
Smaller-then	<	Subtractor	3
Bigger-then	>	Subtractor	3
Smaller-equal-then	<=	Subtractor	3
Bigger-equal-then	>=	Subtractor	3
Equal-to	==	Comparator	3
Not-equal-to	!=	Comparator	3
Bit selection	`[const]`	Wiring	9
Bit-vector selection	`[const:const]`	Wiring	9
Bit concatenation	#	Wiring	4
Type cast	`(type)`	Wiring	6
Precedence ordering	( )		11
Selection	? :	Multiplexer	1

which is `100` as a bitpattern (with the msb counting as a sign bit). Finally, assigning `-4` to a 6-bit number will result in the bitpattern `111100` to be stored in `a`. Since `a` is an unsigned number, the final result is the decimal number `60`.

Listing 5.6 Subtracting 2 and 4 bit

```
reg a : ns(6); // a six-bit unsigned number
sig b : tc(2); // a two-bit signed number
always {
  b = 3;        // assign 0b(011) to b
  a = b - 3;    // subtract 0b(11) and 0b(011)
}
```

The effect of an assignment can also be obtained immediately by means of a cast operation, expressed by writing the desired type between brackets in front of an expression. For example, `(tc(1)) 1` has the value `-1`, while `(ns(3)) 15` has the value `8`.

5.1.3 Hardware Mapping of Expressions

For each expression involving signals and registers of a specified sign and precision, there is an equivalent hardware circuit. This circuit is easy to derive, once we know how each operator is mapped into hardware. We will discuss a list of common operations, and indicate how they map into hardware logic. Table 5.1 presents a summary.

Arithmetic Operations. Addition (+), subtraction (-), multiplication (*) are commonly used in datapath hardware design. The division (/) is not supported in GEZEL (as in most other synthesis-oriented hardware modeling languages). The modulo operation (%) is supported, but it is not synthesizable for arbitrary inputs (since that would require a division). Left-shift (<<) and right-shift (>>) are used to implement multiplication/division with powers of two. Constant-shifts are particularly advantageous for hardware implementation, since they translate to simple hardware wiring.

Bitwise Operations. All of the bitwise operations, including AND (&), OR (|), XOR (^) and NOT (∼) have a direct equivalent to logic gates. Bitwise operations are defined as bit-by-bit operations. The same precision rules as for all other operators apply: when the operands of a bitwise operation are of unequal length, they will be extended until they match. For example, if w is a word and u is a bit, then the expression

```
w & (tc(1)) u
```

will AND each bit of w with the bit in u.

Comparison Operations. All of the comparison operations return a single unsigned bit (ns(1)). These operations use a subtractor to compare two numbers, and then use the sign/overflow flags of the result to evaluate the result of the comparison. Exact comparison (== or !=) can be done by matching the bitpattern of each operand. In contrast to arithmetic operations, the comparison operations are implemented differently for signed and unsigned numbers. Indeed, the bit pattern 111 is smaller than the pattern 001 for signed numbers, but the same pattern 111 is bigger than the pattern 001 for unsigned numbers.

Bitvector Operations. Single bits, or a vector of several bits, can be extracted out of a word using the bit-selection operator.

```
reg a : ns(5);
reg b : ns(1);
reg c : ns(2);
always {
  b = a[3];   // if a = 10111, then b = 0
  c = a[4:2]; // if a = 10111, then a[4:2] = 101, so c = 01
}
```

The type of a bit-selection operation is unsigned, and just wide enough to hold all the bits. The bits in a bit vector are counted from right to left, with bit 0 holding the least significant bit. The opposite operation of bit-selection is bit-concatenation (#), which sticks bits together in a larger word.

```
reg a : ns(5);
reg b : ns(1);
reg c : ns(2);
always {
  a = c # b; // if b = 0, c = 11, then a = 00110
}
```

Selection. The ternary operator `a ? b : c` is the equivalent notation for a multiplexer. The result of the ternary operation will be `b` or `c` depending on the value of `a`. The wordlength of the result will be long enough to accommodate the largest word of either input of the multiplexer.

Indexed storage. There is no array construction in GEZEL. However, it is possible to capture lookup tables. Lookup tables can be implemented in hardware with ROMs or with random logic.

```
lookup T : ns(12) = {0x223, 0x112, 0x990};
reg a : ns(12);
always {
  a = T(2);  // a = 0x990
}
```

Organization and Precedence. Finally, brackets may be used to group expressions and change the evaluation order. The default evaluation order is determined by the precedence of each operator. The precedence is shown as a number in Table 5.1, where a higher number corresponds to a higher precedence, meaning that operator will be evaluated before others.

Each expression created using registers, signals and the operations of Table 5.1, corresponds to a hardware datapath. A few examples are shown next. The first one, in Listing 5.7, shows Euclid's Greatest Common Divisor algorithm. Two registers `m` and `n` are compared, and each clock cycle, the smallest one is subtracted from the largest one. Note that Listing 5.7 does not show how `m` and `n` are initialized.

Listing 5.7 Datapath to evaluate greatest common divisor

```
reg m,n : ns(16);
always {
  m = (m > n) ? (m - n) : m;
  n = (n > n) ? (n - m) : m;
}
```

Describing datapaths with expressions results in compact hardware descriptions. An excellent example are shift registers. Figure 5.2 illustrates a Linear Feedback Shift Register, which is a shift register with a feedback loop created by XORing bits within the shift register. The feedback pattern is specified by a polynomial, and

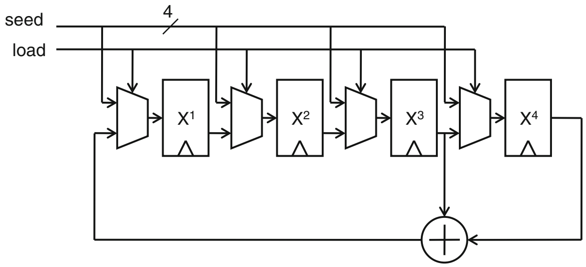


Fig. 5.2 Linear feedback shift register for $p(x) = x^4 + x^3 + 1$

the polynomial used for Fig. 5.2 is $p(x) = x^4 + x^3 + 1$. LFSRs are used for pseudorandom sequence generation. If a so-called maximum-length polynomial is chosen, the resulting sequence of pseudorandom bits has a period of $2^n - 1$, where n is the number of bits in the shift register. Thus, an LFSR is able to create a long non-repeating sequence of pseudorandom bits with a minimal amount of hardware. The shift register used to implement the LFSR must always contain at least one non-zero bit. It is easy to see in Fig. 5.2 that an all-zero pattern in the shift register will only reproduce itself. Therefore, an LFSR must be initialized with a non-zero seed value. The seed value is programmed using a multiplexer in front of each register.

Although the structure of Fig. 5.2 is complex to draw, it remains very compact when written using word-level expressions. This is shown in Listing 5.8. Line 6 of the code represents the shift-and-feedback operation. Line 7 of the code represents the loading of the seed value into the LFSR register.

Listing 5.8 Linear feedback shift register

```
1  reg shft     : ns(4);
2  sig shft_new : ns(4);
3  sig load     : ns(1);
4  sig seed     : ns(4);
5  always {
6    shft_new   = (shft << 1) | (shft[2] ^ shft[3]);
7    shft       = load ? seed : shft_new;
8  }
```

In summary, using two variable types (signals and registers), it is possible to describe synchronous hardware by means of expressions on those signals and registers. Remember that the order in which expressions are written is irrelevant: they will all execute within a single clock cycle. In the next section, we will group expressions into modules, and define input/output ports on those modules.

5.2 Hardware Modules

A hardware module defines a level of hierarchy for a hardware netlist. In order to communicate across levels of hierarchy, hardware modules define ports. Figure 5.3

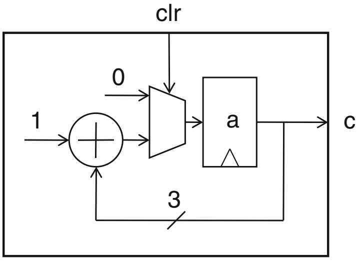


Fig. 5.3 Three-bit counter module

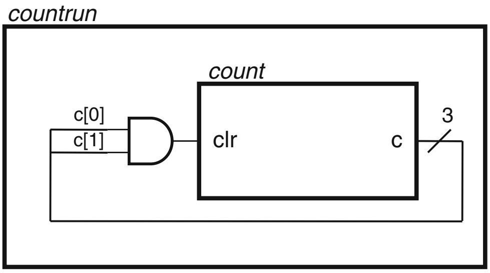


Fig. 5.4 Hardware equivalent of Listing 5.10

shows the 3-bit counter, encapsulated as a module. There is a single input port, `clr`, which synchronously clears the register. There is also a 3-bit output port `c` that holds the current count value. The equivalent description in GEZEL language of this structure is shown in Listing 5.9. The `always` block is included in a `dp` (datapath), which defines a list of `in` and `out` ports. There can be as many input and output ports as needed, and they can be created in any order. Registers and signals are local to a single module and invisible outside of the module boundary. Input ports and output ports are equivalent to wires, and therefore behave identical to signals. Input ports and output ports are subject to similar requirements as signals: it is not allowed to assign an output port more than once during a clock cycle, and each output must be assigned at least once during each clock cycle. We will further investigate this while discussing the formal properties of the FSMD model in Sect. 5.6.

Listing 5.9 Three-bit counter module with reset

```
dp count(in  clr : ns(1);
         out c   : ns(3)) {
  reg a : ns(3);
  always {
    a = clr ? 0 : a + 1;
    c = a;
  }
}
```

After hardware is encapsulated inside of a module, the module itself can be used as a component in another hardware design. This principle is called *structural hierarchy*. As an example, Listing 5.10 shows how the 3-bit counter is included in a testbench structure that clears the counter as soon as it reaches three. The module is included by the `use` keyword, which also shows how ports should be connected to local signals and registers. The equivalent hardware structure of Listing 5.10 is shown in Fig. 5.4.

The `countrun` module in Listing 5.10 has no inputs nor outputs. Such modules have no practical value for implementation, but they may be useful for simulation. The listing shows, for example, how `countrun` encapsulates `count`, and how it generates test vectors for that module. The listing also illustrates the use of a `$display` statement, which is a simulation *directive* that prints the value of a signal or register.

Listing 5.10 Encapsulated counter module

```
dp countrun {
  sig clearit : ns(1);
  sig cnt     : ns(3);
  use count(clearit, cnt);
  always {
    clearit = cnt[0] & cnt[1];
    $display("cnt = ", cnt);
  }
}
```

Once a module has been included inside of another one by means of the `use` statement, it cannot be included again: each module can be used only once. However, it is easy to create a duplicate of an existing module by means of a *cloning* statement. Listing 5.11 shows how to create three 3-bit counters, `count0`, `count1` and `count2`.

Listing 5.11 Cloning of modules

```
dp count0(in  clr : ns(1);
         out c   : ns(3)) {
  reg a : ns(3);
  always {
    a = clr ? 0 : a + 1;
    c = a;
  }
}
dp count1 : count0
dp count2 : count0
```

5.3 Finite State Machines

We will next describe a mechanism to control hardware circuits. As discussed before, the expressions that are part of an `always` block are evaluated at each clock cycle, and it is not possible to conditionally evaluate an expression. Even the selection operator (`c ? expr1 : expr2`) will evaluate the true-part as well

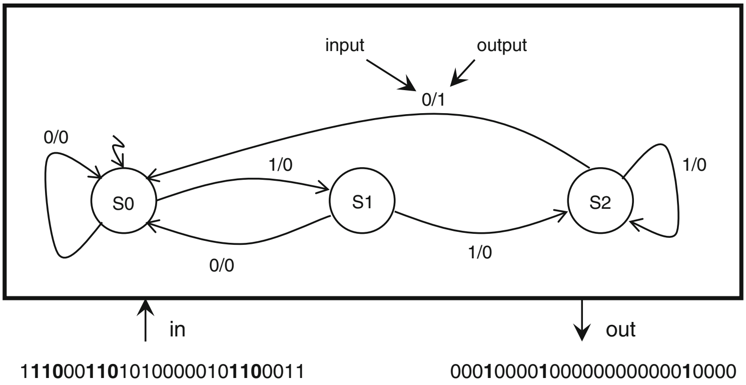


Fig. 5.5 Mealy FSM of a recognizer for the pattern '110'

as the false-part regardless of the condition value `c`. If `expr1` and `expr2` would contain an expensive operator, then we would need two copies of that operator to implement `c ? expr1 : expr2`.

A control model, on the other hand, allows us to indicate what expressions should execute during each clock cycle. Very simple control models will select a sequence of expressions to execute, over multiple clock cycles. More complex control models will also allow decision making. Advanced control models also consider exceptions, recursion, out-of-order execution, and more. In this section, we describe a common control model for hardware description, called Finite State Machine (FSM). An FSM can be used to described sequencing and decision making. In the next section, we will combine the FSM with expressions in a datapath.

An FSM is a sequential digital machine which is characterized by

- A set of states;
- A set of inputs and outputs;
- A state transition function;
- An output function

An FSM has a current state, equal to an element from the set of states. Each clock cycle, the state transition function selects the next value for the current state, and the output function selects the value on the output of the FSM. The state transition function and the output function are commonly described in terms of a graph. In that case, the set of states becomes the set of nodes of the graph, and the state transitions become edges in the graph.

The operation of an FSM is best understood by means of an example. Suppose we need to observe a (possibly infinite) sequence of bits, one at a time. We need to determine at what point the sequence contains the pattern '110'. This problem is well suited for an FSM design. We can distinguish three relevant states for the FSM, by realizing that sequential observation of the input bits transforms this pattern recognition problem into an incremental process.

Table 5.2 Conversion of Mealy to Moore FSM for Fig. 5.5

Current-state	Input = 0	Input = 1
S0	S0, 0	S1, 0
S1	S0, 0	S2, 0
S2	S0, 1	S2, 0

Table 5.3 Resulting Moore state transition table

Current-state	Input = 0	Input = 1	Output
SA = S0, 0	SA	SC	0
SB = S0, 1	SA	SC	1
SC = S1, 0	SA	SD	0
SD = S2, 0	SB	SD	0

1. State S0: We have not recognized any useful pattern.
2. State S1: We have recognized the pattern ‘1’.
3. State S2: We have recognized the pattern ‘11’.

When we consider each state and each possible input bit, we can derive all state transitions, and thus derive the state transition function. The output function can be implemented by defining a successful recognition as an input bit of ‘0’ when the FSM is in state S2. This leads to the state transition graph (or state transition diagram) shown in Fig. 5.5. The notation used for Fig. 5.5 is that of a *Mealy* FSM. The output of a Mealy FSM is defined by the present state, as well as the input.

There is a different formulation of a FSM known as a *Moore* state machine. The output of a Moore state machine is only dependent on the current state, and not on the current input. Both forms, Mealy and Moore, are equivalent formulations of FSM. A Mealy machine can be converted into an equivalent Moore machine using a simple conversion procedure.

To convert the Mealy form into a Moore form, make a state transition table for the Mealy FSM. This table contains the next-state of each state transition combined with the relevant output. For example, in state S1 of Fig. 5.5, the input ‘1’ leads to state S2 with output 0. Annotate this in the Table as: S2, 0. The entire diagram can be translated this way, leading to Table 5.2.

Using the conversion table, an equivalent Moore FSM can be constructed as follows. There is one Moore state for each unique *(next-state, output)* pattern. From Table 5.2, we can find four Moore states: (S0, 0), (S0, 1), (S1, 0) and (S2, 0). To find the Moore FSM state transitions, we replicate the corresponding Mealy transitions for each Moore state. There may be multiple Moore transitions for a single Mealy transition. For example, Mealy state S0 is replicated into Moore states (S0, 0) and (S0, 1). Thus, each of the state transitions out of S0 will be replicated two times. The resulting Moore state transition table is shown in Table 5.3, while the resulting Moore FSM graph is drawn in Fig. 5.6. A small ambiguity was removed from Fig. 5.6 by making state SA = (S0, 0) the initial state of the Moore machine. That is, the Mealy machine does not specify the initial value of the output. Since the Moore machine ties the output to the current state, an initial output value must be assumed as well.

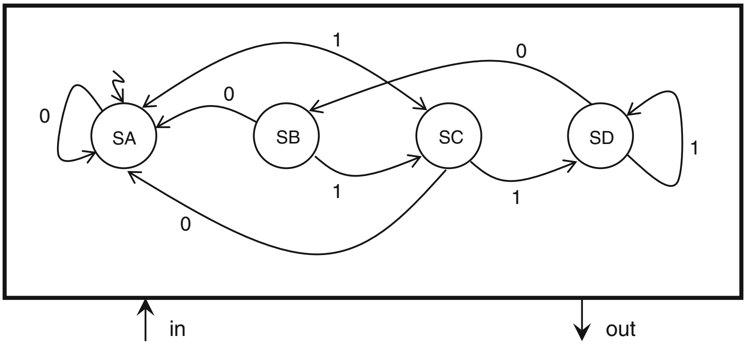


Fig. 5.6 Moore FSM of a recognizer for the pattern '110'

In summary, a Finite State Machine is a common control model for hardware design. We can use it to model the conditional execution of expressions. In that case, we will use the outputs of the FSM to control the execution of expressions. Similarly, we will use the inputs to feed runtime conditions into the FSM, so that conditional sequencing can be implemented. In the next section, we will use a modified form of a FSM, called FSMD, which is the combination of an FSM and a datapath (modeled using expressions).

5.4 Finite State Machines with Datapath

A Finite State Machine with Datapath combines a hardware control model (an FSM) with a datapath. The datapath is described as cycle-based bit-parallel hardware using expressions, as discussed in Sect. 5.1. However, in contrast to datapaths using only `always` blocks, FSMD datapaths define also one or more *instructions*: conditionally executed '`always`' blocks.

5.4.1 Modeling

Listing 5.12 shows an example of a datapath with an `always` block and three instructions: `inc`, `dec`, and `clr`. The meaning of the `always` block is the same as before: it contains expressions that will execute every clock cycle. In contrast to an `always` block, instructions will only execute when told to do so by a controller. Thus, the three instructions of the datapath in Listing 5.12 can increment, decrement, or clear register `a`, depending on what the controller tells the datapath to do.

Listing 5.12 Datapath for an up-down counter with three instructions

```
dp updown(out c    : ns(3)) {
   reg a : ns(3);
   always  { c = a; }
   sfg inc { a = a + 1; } // instruction inc
   sfg dec { a = a - 1; } // instruction dec
   sfg clr { a = 0;     } // instruction clr
 }
```

An instruction has the same meaning for expressions as an `always` block: it specifies operations in combinational logic and describes register-updates. An instruction has a name such as `inc`, `dec` and `clr` so it can be referenced by a controller. The keyword `sfg` precedes the instruction name, and this keyword is an acronym for *signal flow graph*: An instruction is a snippet of a dataflow graph of a hardware description during one clock cycle of processing.

A datapath with instructions needs a controller to select what instruction should execute in each clock cycle. The FSMD model uses an FSM for this. Listing 5.13 shows a Finite State Machine controller for the datapath of Listing 5.12. In this case, the controller will steer the datapath such that it first counts up from 0 to 3, and next counts down from 3 to 0. The finite state machine is a textual format for a state transition diagram. Going through the listing, we can identify the following features.

- Line 1: The `fsm` keyword defines a finite state machine with name `ctl_updown`, and tied to the datapath `updown`.
- Lines 2–3: The FSM contains three states. One state, called `s0`, will be the initial state. Two other states, called `s1` and `s2`, are regular states. The current state of the FSM will always be one of `s0`, `s1` or `s2`.
- Line 4: When the current state is `s0`, the FSM will unconditionally transition to state `s1`. State transitions will be taken at each clock edge. At the same time, the datapath will receive the instruction `clr` from the FSM, which will cause the register `a` to be cleared (See Listing 5.9 on Page 121).
- Lines 5–6: When the current state of the FSM equals `s1`, a conditional state transition will be taken. Depending on the value of the condition, the FSM will transition either to state `s1` (Line 5), or else to state `s2` (Line 6), and the datapath will receive either the instruction `inc` or else `dec`. This will either increment or else decrement register `a`. The state transition condition is given by the expression `a < 3`. Thus, the FSM will remain in state `s1` as long as register `a` is below three. When `a` equals three, the FSM will transition to `s2` and a decrementing sequence is initiated.
- Lines 7–8: When the current state of the FSM equals `s2`, a conditional state transition to either `s1` or else `s2` will be taken. These two state transitions will issue a decrement instruction to the datapath as long as register `a` is above zero. When the register equals zero, the controller will transition to `s1` and restart the incrementing sequence.

Listing 5.13 Controller for the up-down counter

```
1  fsm ctl_updown(updown) {
```

Table 5.4 Behavior of the FSMD in Listing 5.9

	FSM	DP	DP	
Cycle	curr/next	instr	expr	a curr/next
0	s0/s1	clr	a = 0;	0/0
1	s1/s1	inc	a = a + 1;	0/1
2	s1/s1	inc	a = a + 1;	1/2
3	s1/s1	inc	a = a + 1;	2/3
4	s1/s2	dec	a = a − 1;	3/2
5	s2/s2	dec	a = a − 1;	2/1
6	s2/s2	dec	a = a − 1;	1/0
7	s2/s1	inc	a = a + 1;	0/1
9	s1/s1	inc	a = a + 1;	1/2
9	s1/s1	inc	a = a + 1;	2/3

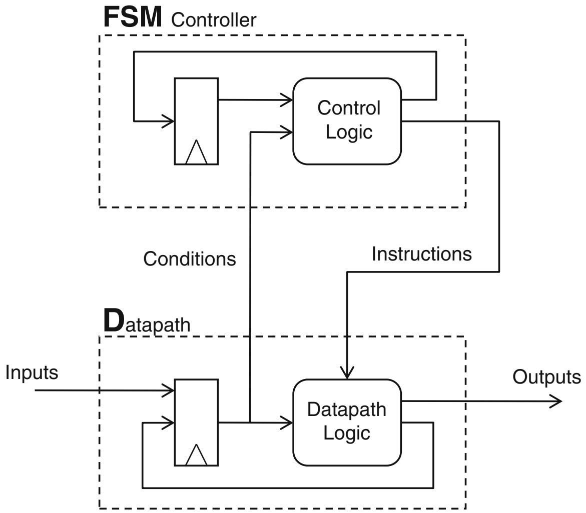
```
    initial s0;
    state s1, s2;
    @s0 (clr) -> s1;
    @s1 if (a < 3) then (inc) -> s1;
                   else (dec) -> s2;
    @s2 if (a > 0) then (dec) -> s2;
                   else (inc) -> s1;
}
```

Thus, the FSM controller determines the schedule of instructions on the datapath. There are many possible schedules, and the example shown in Listing 5.13 is one of them. However, since an FSM is not programmable, the implementation of the FSM will fix the schedule of instructions for the datapath. Table 5.4 illustrates the first ten clock cycles of operation for this FSMD. Each row shows the clock cycle, the current and next FSM state, the datapath instruction selected by the controller, the datapath expression, and the current and next value of the register `a`.

In the up/down counter example, each datapath instruction contains a single expression, and the FSM selects a single datapath instruction for execution during each clock cycle. This is not a strict requirement. An instruction may contain as many expressions as needed, and the FSM controller may select multiple instructions for execution during any clock cycle. You can think of a group of scheduled instructions and the `always` block as a single, large `always` block that is active for a single clock cycle. Of course, not all combinations will work in each case. For example, in the datapath shown in Listing 5.12, the instructions `clr`, `inc` and `dec` are all exclusive, since all of them modify register `a`. The set of expressions that execute during a given clock cycle (as a result of the `always` block and the scheduled instructions) have to be conform to the same rules as if there were only a single `always` block. We will define these rules precisely in Sect. 5.6.

Listing 5.14 shows the implementation of Euclid's algorithm as an FSMD. In this case, several datapath instructions contain multiple expressions. In addition, the controller selects multiple datapath instructions during one state transition (line 21). The body of the `reduce` instruction was presented earlier as the computational core

Fig. 5.7 An FSMD consists of two stacked FSMs



of GCD (Listing 5.14). The additional functionality provided by the controller is the initialization of the registers m and n, and the detection of the algorithm completion.

Listing 5.14 Euclid's GCD as an FSMD

```
dp euclid(in  m_in, n_in : ns(16);
          out gcd        : ns(16)) {
  reg m, n               : ns(16);
  reg done               : ns(1);
  sfg init    { m    = m_in;
                n    = n_in;
                done = 0;
                gcd  = 0; }
  sfg reduce  { m    = (m >= n) ? m - n : m;
                n    = (n >  m) ? n - m : n;   }
  sfg outidle { gcd  = 0;
                done = ((m == 0) | (n == 0)); }
  sfg complete{ gcd  = ((m > n) ? m : n);
                $display(``gcd = '', gcd); }
}
fsm euclid_ctl(euclid) {
  initial s0;
  state s1, s2;
  @s0 (init) -> s1;
  @s1 if (done) then (complete)        -> s2;
                else (reduce, outidle) -> s1;
  @s2 (outidle) -> s2;
}
```

5.4.2 *The FSMD Model As Two Stacked FSM*

Here is another way to look upon and FSMD and its associated execution model. An FSMD consists of two stacked FSMs, as illustrated in Fig. 5.7. The top FSM contains the controller, and it is specified using a state transition diagram. The bottom FSM contains the datapath, and it is specified using expressions. The top FSM send instructions to the bottom FSM, and receives status information in return. Both FSM operate synchronously, and are connected to the same clock.

Each clock cycle, the two FSM go through the following activities.

1. Just after the clock edge, the state variables of both FSM are updated. For the controller, this means that a state transition is completed and the state register holds the new current-state value. For the datapath, this means that the register variables are updated as a result of assigning expressions to them.
2. The control FSM combines the control-state and datapath-state to evaluate the new next-state for the control FSM. At the same time, it will also select what instructions should be executed by the datapath.
3. The datapath FSM will evaluate the next-state for the state variables in the datapath, using the updated datapath state as well as the instructions received from the control FSM.
4. Just before the next clock edge, both the control FSM and the datapath FSM have evaluated and prepared the next-state value for the control state as well as the datapath state.

What makes a controller FSM different from a datapath FSM? Indeed, as illustrated in Fig. 5.7, both the datapath and the controller are sequential digital machines. Yet, from a designers' viewpoint, the creation of datapath logic and control logic is very different.

- Control logic tends to have an irregular structure. Datapath logic tends to have a regular structure (especially once you work with multi-bit words).
- Control logic is easy to describe using a finite state transition diagram, and hard to describe using expressions. Datapath logic is just the opposite: easy to describe using expressions, but hard to capture in state transition diagrams.
- The registers (state) in a controller have a different purpose than those in the datapath. Datapath registers contain algorithmic state. Control registers contain sequencing state.

In conclusion, FSMDs are useful because they capture control flow as well as data flow in hardware. Recall that C programs are also a combination of control flow and dataflow (Chap. 4). We relied on this commonality, for example, to convert a C program into a hardware FSMD.

5.4.3 An FSMD Is Not Unique

Figure 5.7 demonstrated how an FSMD actually consists of two stacked FSM. This has an interesting implication. From an implementation perspective, the partitioning between control logic and datapath logic is not unique. To illustrate this on Fig. 5.7, assume that we would merge the control logic and datapath logic into a single logic module, and assume that we would combine the registers in the controller with those in the datapath. The resulting design would still look as a FSM. The implementation does not make a clear distinction between the datapath part and FSM part.

In the previous subsection, we showed that the modeling of the FSM controller and the datapath is very different: using state transition graphs and expressions respectively. Since the partitioning between a controller and the datapath is not unique, this implies that we should be able to describe an FSM using expressions. This is illustrated in Listing 5.15, which shows the datapath-version of the controller in Listing 5.13. Apart from the difference in notation (state transition diagrams versus expressions), there is another important difference between both listings. In Listing 5.15, we have chosen the encoding for controller states. In Listing 5.13, the state encoding is symbolic (`s0`, `s1`, ...).

Listing 5.15 FSM controller for updown counter using expressions

```
dp updown_ctl(in  a_sm_3, a_gt_0  : ns(1);
              out instruction     : ns(2)) {
  reg state_reg : ns(2);
  // state encoding: s0 = 0, s1 = 1, s2 = 2
  // instruction encoding: clr = 0, inc = 1, dec = 2
  always {
    state_reg   = (state_reg == 0) ? 1 :
                 ((state_reg == 1) &  a_sm_3) ? 1 :
                 ((state_reg == 1) & ~a_sm_3) ? 2 :
                 ((state_reg == 2) &  a_gt_0) ? 2 : 1;
    instruction = (state_reg == 0) ? 0 :
                 ((state_reg == 1) &  a_sm_3) ? 1 :
                 ((state_reg == 1) & ~a_sm_3) ? 2 :
                 ((state_reg == 2) &  a_gt_0) ? 2 : 1;
  }
}
```

If we can model an FSM with expressions, we can also merge it with the datapath controlled by this FSM. The resulting design for the up-down counter is shown in Listing 5.16. In this case, an entire FSMD is captured in a single datapath. How then, would you choose between developing separate FSM and datapath descriptions, versus capturing all of them in a single datapath? The following are some of the considerations you could make.

- State transition conditions of an FSM need to be stored in registers, which introduces a latency of one clock cycle for each conditional state transition. When capturing an FSM with datapath expressions, state transition conditions can be generated and evaluated in the same clock cycle.

- When capturing the FSMD in a single datapath, the expressions in the datapath include scheduling as well as data processing. On the other hand, in an FSMD with separate FSM and datapath descriptions, the datapath expressions only represent data processing. Listing 5.16 shows that, merging a controller into a datapath description tends to make the overall design more complicated to understand.
- When a datapath includes scheduling as well as data processing, it becomes harder to reuse it in a different schedule. Using an FSMD with separate FSM and datapath description on the other hand, will allow changes to the scheduling (the FSM) while reusing most of the datapath description.
- When capturing the FSMD in a single datapath, the state assignment is chosen by the designer, and may be optimized for specific applications. In an FSMD with separate FSM and datapath description, the state assignment is left to the logic synthesis tool.
- An FSMD captured as a single datapath is good for designs with simple or no control scheduling, such as designs with high-throughput requirements. An FSMD with separate FSM and datapath description is good for more complicated, structured designs.

Listing 5.16 Updown counter using expressions

```
dp updown_ctl(out c : ns(3)) {
  reg a      : ns(3);
  reg state  : ns(2);
  sig a_sm_3 : ns(1);
  sig a_gt_0 : ns(1);
  // state encoding: s0 = 0, s1 = 1, s2 = 2
  always {
    state = (state == 0) ? 1 :
            ((state == 1) &  a_sm_3) ? 1 :
            ((state == 1) & ~a_sm_3) ? 2 :
            ((state == 2) &  a_gt_0) ? 2 : 1;
    a_sm_3 = (a < 3);
    a_gt_0 = (a > 0);
    a = (state == 0) ? 0 :
        ((state == 1) &  a_sm_3) ? a + 1 :
        ((state == 1) & ~a_sm_3) ? a - 1 :
        ((state == 2) &  a_gt_0) ? a + 1 : a - 1;
    c = a;
  }
}
```

In the above examples, we showed how an FSM can be modeled using datapath expressions, and how this allowed to capture an entire FSMD in a single datapath. The opposite case (modeling a datapath as a state transition diagram) is very uncommon. Problem 5.10 investigates some of the reasons for this.

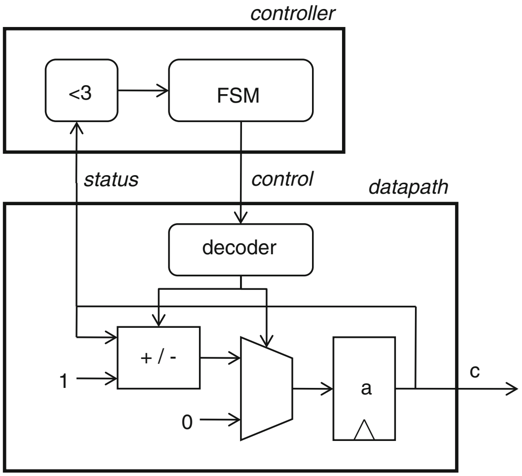


Fig. 5.8 Implementation of the up-down counter FSMD

5.4.4 Implementation

Listing 5.17 Datapath for an up-down counter with three instructions

```
dp updown(out c   : ns(3)) {
  reg a : ns(3);
  always  { c = a;     }
  sfg inc { a = a + 1; }
  sfg dec { a = a - 1; }
  sfg clr { a = 0;     }
}
fsm ctl_updown(updown) {
  initial s0;
  state s1, s2;
  @s0 (clr) -> s1;
  @s1 if (a < 3) then (inc) -> s1;
                 else (dec) -> s2;
  @s2 if (a > 0) then (dec) -> s2;
                 else (inc) -> s1;
}
```

How to determine the hardware implementation of an FSMD? The basic rules for mapping expressions on registers/signals into synchronous digital hardware are the same as with `always` blocks. However, there is also an important difference. When an expression occurs inside of a datapath instruction, it should execute only when that instruction is selected by the controller. Thus, the final datapath structure will depend on the schedule of instructions selected by the FSMD controller.

To clarify this point, consider again the up-down counter in Listing 5.17. From the FSM model, it's easy to see that `clr`, `inc`, and `dec` will always execute in different clock cycles. Therefore, the datapath operators used to implement each of `clr`, `inc`, and `dec` can be shared. A possible implementation of the FSMD is shown in Fig. 5.8. The datapath includes an adder/subtractor and a multiplexer, which implement the instruction-set (clr, inc, dec). A local decoder is needed to convert the encoding used for these instructions into local control signals for the multiplexer and the adder subtractor. The datapath is attached to a controller, which is implemented with an FSM and a local decoder for datapath status information. In this case, the value in register a is datapath status.

The resulting implementation looks as shown in Fig. 5.8. Many of the detailed implementation of Fig. 5.8 will be generated using automatic design tools (see next section). A designer who develops an FSMD will make the following overall decisions.

- The designer determines the amount of work done in a single clock cycle. This is simply the set of all datapath instructions which will execute concurrently in a single clock cycle. Hence, in order to obtain the best possible sharing, a designer must distribute similar operations over multiple clock cycles. For example, if there are 16 multiplications to perform with a clock cycle budget of four clock cycles, then an implementation with 4 multiplies each clock cycle will most likely be smaller than one which performs 16 multiplies in the first clock cycle and nothing in the next three cycles.
- The designer can also influence, indirectly, the maximum clock frequency attainable by the hardware design. This frequency is determined by the complexity of the expressions given in the datapath instructions. Obviously, small and short expressions will result in smaller and faster logic in the datapath. If the expressions used to capture the datapath become too complex, the resulting design may be too slow for the intended system clock period.

GEZEL descriptions can be simulated directly, as source code, or they can be converted into VHDL and simulated by a VHDL simulator. Refer to Appendix A for a brief overview on the installation and use of GEZEL tools.

5.5 FSMD Design Example: A Median Processor

So far, we discussed all building blocks to design a complete hardware implementation. In this section, we will apply these modeling techniques to create a full hardware implementation of a reference design in software.

5.5.1 Design Specification: Calculating the Median

The target design is a processor to compute the median of a list of numbers. The median operation selects the middle-ranked element of a list (for a list with an odd number of elements), or the average of the two middle-ranked elements (for a list with an even number of elements). For example, in the list

```
L = {4, 56, 2, 10, 32}
```

the median would be 10, since the ranked list corresponds to {2, 4, 10, 32, 56}. For our design, we will assume a fixed-length list with an odd number of elements. A median operation has an application when used as a filter in image processing applications. In that case, a stream of pixels is fed through the median operation, and each pixel is replaced with the median of the surrounding pixels. The effect of this operation is to reduce noise, which appears as speckle in the input image. Indeed, the median operation is very effective at removing outlier values within a list of pixels. This example also demonstrates the need for a high-speed implementation of the median, in particular when considering moving images.

Calculation of the median of a list requires one to sort the elements of the list, in order to find the middle element. A more efficient algorithm is the following: the median of a list L is the element e for which there are (♯L - 1)/2 elements in the list that are smaller than e. For example, to find the median of a list of five elements, we would identify the element for which there are two elements in L smaller than that element. In the example above, you can see that the value 10 meets this condition.

Listing 5.18 C function to compute the median of five numbers

```c
int median(int a1, int a2, int a3, int a4, int a5) {
  int z1, z2, z3, z4, z5, z6, z7, z8, z9, z10;
  int s1, s2, s3, s4;
  z1  = (a1 < a2);
  z2  = (a1 < a3);
  z3  = (a1 < a4);
  z4  = (a1 < a5);
  z5  = (a2 < a3);
  z6  = (a2 < a4);
  z7  = (a2 < a5);
  z8  = (a3 < a4);
  z9  = (a3 < a5);
  z10 = (a4 < a5);
  s1 = ((   z1  +    z2  +    z3  +     z4)  == 2);
  s2 = (((1-z1) +    z5  +    z6  +     z7)  == 2);
  s3 = (((1-z2) + (1-z5) +    z8  +     z9)  == 2);
  s4 = (((1-z3) + (1-z6) + (1-z8) +   z10)   == 2);
  return ( s1 ? a1 : s2 ? a2 : s3 ? a3 : s4 ? a4 : a5);
}
```

Listing 5.18 shows a reference implementation of a C function to compute the median of five inputs. The function implements the algorithm described earlier. For each input, it calculates how many other inputs are smaller than that input. Since a<b implies b>a, ten comparisons are needed for five inputs. Finally, the return value of the function is the element for which the sum of comparisons equals 2.

5.5.2 *Mapping the Median in Hardware*

Next we design an FSMD for this function, and illustrate some of the issues that one must handle when writing a hardware implementation from a software specification. We will start, at first, by designing a datapath that accepts five values, and that produces the median as the output. Based on Listing 5.18, such a design is straightforward in GEZEL. Some wordlength optimization is possible, since the result of a comparison is only a single bit. Listing 5.19 shows an equivalent datapath design in GEZEL.

Listing 5.19 GEZEL Datapath of a median calculation of five numbers

```
dp median(in a1, a2, a3, a4, a5 : ns(32); out q1 : ns(32)) {
  sig z1, z2, z3, z4, z5, z6, z7, z8, z9, z10 : ns(3);
  sig s1, s2, s3, s4, s5 : ns(1);
  always {
    z1  = (a1 < a2);
    z2  = (a1 < a3);
    z3  = (a1 < a4);
    z4  = (a1 < a5);
    z5  = (a2 < a3);
    z6  = (a2 < a4);
    z7  = (a2 < a5);
    z8  = (a3 < a4);
    z9  = (a3 < a5);
    z10 = (a4 < a5);
    s1 = ((   z1  +    z2  +    z3  +     z4)  == 2);
    s2 = (((1-z1) +    z5  +    z6  +     z7)  == 2);
    s3 = (((1-z2) + (1-z5) +    z8  +     z9)  == 2);
    s4 = (((1-z3) + (1-z6) + (1-z8) +    z10)  == 2);
    q1  = s1 ? a1 : s2 ? a2 : s3 ? a3 : s4 ? a4 : a5;
  }
}
```

Compared to the software implementation, Listing 5.19 completes in a single clock cycle. However, this function makes an assumption which is not present in the C implementation. The GEZEL version assumes that all inputs of `median` are available at the same moment! In C, variables are stored in memory, and memory access are sequential with one another. Hence, although Listing 5.18 and Listing 5.19 are syntactically almost identical, they correspond to a very different behavior. In this case, the difference in implementation affects the communication between the `median` function and the rest of the system. The hardware design requires a bus of 6 times 32 bits (5 inputs and 1 output), or 192 bits, to carry all of the data from and to the median calculation unit.

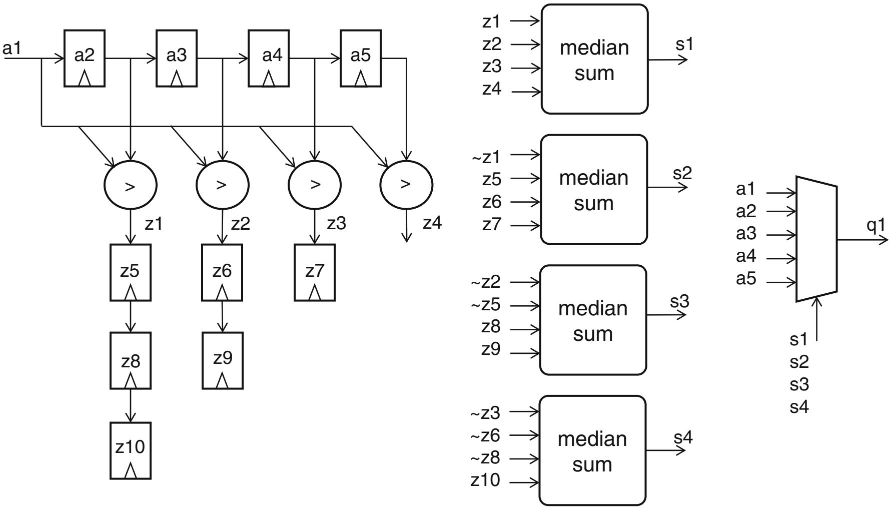


Fig. 5.9 Median-calculation datapath for a stream of values

5.5.3 *Sequentializing the Data Input*

Let's consider an alternative: a design with a single input, 32 bit wide, and a single output, 32 bit wide. For each input, the design will evaluate the median over the last five values provided to the system. For each new input, a new median can be calculated. Clearly, we can simply pipe the input into a chain of registers. This will store a window of values. The median computation requires ten comparisons for five arbitrarily chosen inputs. In this case, however, we will recompute the median over every input value. This allows reusing some of the comparisons from previous iterations. Indeed, the number of new comparisons required per median value is 4, with 6 comparisons reusable from previous iterations. Reusing comparisons is advantageous because these involve 32-bit comparators. The resulting design is illustrated in Fig. 5.9. An equivalent GEZEL description of this design is shown in Listing 5.20.

Listing 5.20 GEZEL Datapath of a median calculation of five numbers with sequentialized data input

```
dp median(in a1 : ns(32); out q1 : ns(32)) {
  reg a2, a3, a4, a5 : ns(32);
  sig z1, z2, z3, z4;
  reg z5, z6, z7, z8, z9, z10 : ns(3);
  sig s1, s2, s3, s4, s5 : ns(1);
  always {
    a2  = a1;
    a3  = a2;
    a4  = a3;
    a5  = a4;
    z1  = (a1 < a2);
    z2  = (a1 < a3);
    z3  = (a1 < a4);
    z4  = (a1 < a5);
    z5  = z1;
    z6  = z2;
    z7  = z3;
    z8  = z5;
    z9  = z6;
    z10 = z8;
    s1 = ((   z1  +    z2  +    z3  +     z4)  == 2);
    s2 = (((1-z1) +    z5  +    z6  +     z7)  == 2);
    s3 = (((1-z2) + (1-z5) +    z8  +     z9)  == 2);
    s4 = (((1-z3) + (1-z6) + (1-z8) +    z10)  == 2);
    q1  = s1 ? a1 : s2 ? a2 : s3 ? a3 : s4 ? a4 : a5;
  }
}
```

5.5.4 Fully Sequentialized Computation

In Listing 5.20, there are still four parallel 32-bit comparators required, to compute a single median value per clock cycle. In addition, the computation of the terms `s1`, `s2`, `s3` and `s4` is very similar. By spreading the computation of a median value over multiple clock cycles, the number of operators in the datapath can be reduced, at the expense of adding registers and multiplexers. This section will demonstrate this by building a fully sequentialized version of the median computation unit, implemented as an FSMD. The FSM is particularly useful to capture the controller required for such a sequentialized design.

The first step to design a sequentialized hardware module is to create a schedule – a time-sequence for the computations involved in the hardware design. This could be done directly on a C program, but in this case we’ll demonstrate the idea directly on the hardware of Fig. 5.10. The white-on-black labels indicate the clock cycles of our schedule. Each clock cycle, a different set of operations will be evaluated, until the final output can be computed, in clock cycle 12. Allocating similar operations in different clock cycles (for example, the comparison in clock cycles 2, 3, 4, and 5) will enable reuse of datapath operators.

Once the schedule is created, the resulting design can be coded. Listing 5.21 shows such a fully sequentialized version of the median computation. It is functionally identical to the previous designs, but uses 12 cycles to compute a median. It looks very different as a result. The design consists of the following major features.

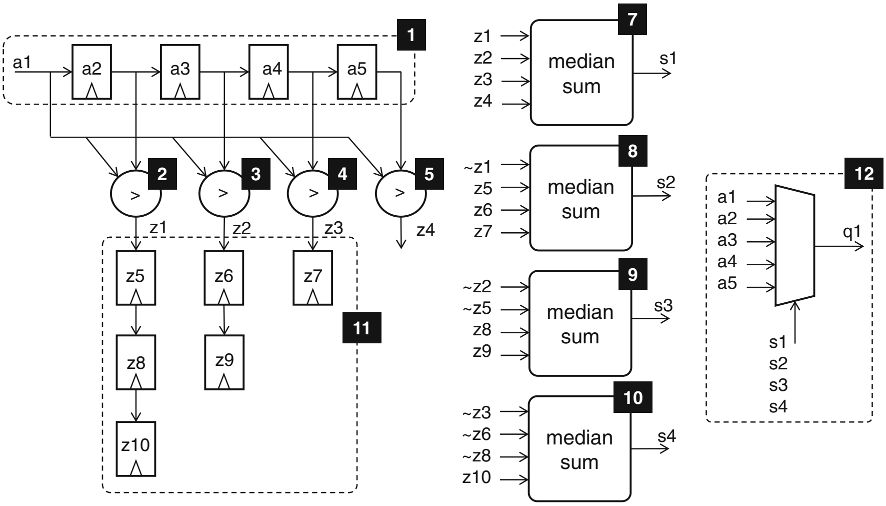


Fig. 5.10 Sequential schedule median-calculation datapath for a stream of values

- The module `hastwo` checks if exactly two input bits are one. If so, it returns true. This module is a simplified formulation of the expressions used to compute `s1`, `s2`, `s3` and `s4`. The module is instantiated in `median` (line 27), and used in four datapath instructions: `c_s1`, `c_s2`, `c_s3`, `c_s4` (lines 44–47).
- The module `comp` is a 32-bit comparator. It is instantiated in `median` (line 31), and used in four datapath instructions: `c_z1`, `c_z2`, `c_z3` and `c_z4` (lines 40–43).
- Because of the sequentialized computation, all intermediate signals from Listing 5.20 have become registers in Listing 5.21. Also, care is taken to schedule the datapath instructions in a sequence so as never to overwrite a register that would still be needed (lines 67–80).
- The datapath uses an input data-strobe and an output data-strobe. The strobe is asserted (1) when the input is read or the output is produced. Such strobes are common in multi-cycle hardware designs to support the integration of this hardware module in a larger system. In this case, the strobes are used by the testbench (lines 83–108) to decide when to provide a new input. The testbench generates a pseudorandom sequence of 16-bit values, and the sequence is advanced each time a strobe is detected.

Listing 5.21 Fully sequentialized median calculation with testbench

```
1  dp hastwo(in a, b, c, d : ns(1); out q : ns(1)) {
2    always {
3      q = ( a &  b & ~c & ~d) |
4          ( a & ~b & ~c &  d) |
5          (~a & ~b &  c &  d) |
6          ( a & ~b &  c & ~d) |
        (~a &  b & ~c &  d) |
        (~a &  b &  c & ~d);
  }
}

dp comp(in a, b : ns(32); out q : ns(1)) {
  always { q = (a > b); }
}

dp median(in  istr  : ns(1);
          in  a1_in : ns(32);
          out ostr  : ns(1);
          out q1    : ns(32)) {
  reg a1, a2, a3, a4, a5 : ns(32);
  reg z1, z2, z3, z4, z5, z6, z7, z8, z9, z10 : ns(3);
  reg s1, s2, s3, s4 : ns(1);

  reg ristr : ns(1);

  sig h1, h2, h3, h4, qh : ns(1);
  use hastwo(h1, h2, h3, h4, qh);

  sig m1, m2 : ns(32);
  sig mq     : ns(1);
  use comp(m1, m2, mq);

  always        { ristr = istr; }
  sfg getinput  { a1 = istr ? a1_in : a1;
                  a2 = istr ? a1 : a2;
                  a3 = istr ? a2 : a3;
                  a4 = istr ? a3 : a4;
                  a5 = istr ? a4 : a5;
                }
  sfg c_z1      { m1 =  a1; m2 = a2; z1 = mq; }
  sfg c_z2      { m1 =  a1; m2 = a3; z2 = mq; }
  sfg c_z3      { m1 =  a1; m2 = a4; z3 = mq; }
  sfg c_z4      { m1 =  a1; m2 = a5; z4 = mq; }
  sfg c_s1      { h1 =  z1; h2 = z2; h3 = z3; h4 =  z4;
                  s1 = qh; }
  sfg c_s2      { h1 = ~z1; h2 = z5; h3 = z6; h4 =  z7;
                  s2 = qh; }
  sfg c_s3      { h1 = ~z2; h2 =~z5; h3 = z8; h4 =  z9;
                  s3 = qh; }
  sfg c_s4      { h1 = ~z3; h2 =~z6; h3 =~z8; h4 = z10;
                  s4 = qh; }
  sfg c_z5z10   { z5 =  z1; z6 = z2; z7 = z3; z8 =  z5;
                  z9 = z6; z10 = z8; }
  sfg c_notwo   { h1   =  0;  h2 = 0; h3 =  0; h4 =   0;}
  sfg c_nocomp  { m1   =  0;  m2 = 0; }
  sfg c_noout   { ostr =  0;  q1 = 0; }
  sfg putoutput{  q1 = s1 ? a1 : s2 ? a2 : s3 ? a3 : s4 ?
                  a4 : a5;
                  ostr =  1;
```

```
                 $display($cycle, `` a1 '', a1, `` q1 '', q1
                    );
               }
}
fsm ctl_median(median) {
 initial s0;
 state s1, s2, s3, s4, s5, s6, s7, s8, s9, s10;
 state s11, s12, s13, s14, s15, s16;

 @s0 if (ristr) then (          c_notwo, c_nocomp, c_noout)
                       ->s1;
                else (getinput, c_notwo, c_nocomp, c_noout)
                       ->s0;
 @s1  (c_notwo, c_z1, c_noout) -> s2;
 @s2  (c_notwo, c_z2, c_noout) -> s3;
 @s3  (c_notwo, c_z3, c_noout) -> s4;
 @s4  (c_notwo, c_z4, c_noout) -> s5;
 @s5  (c_s1,    c_nocomp, c_noout) -> s6;
 @s6  (c_s2,    c_nocomp, c_noout) -> s7;
 @s7  (c_s3,    c_nocomp, c_noout) -> s8;
 @s8  (c_s4,    c_nocomp, c_noout) -> s9;
 @s9  (c_notwo, c_nocomp, c_z5z10, c_noout) -> s10;
 @s10 (c_notwo, c_nocomp, putoutput)        -> s0;
}

dp t_median {
  sig istr, ostr : ns(1);
  sig a1_in, q1 : ns(32);
  use median(istr, a1_in, ostr, q1);
  reg r : ns(1);
  reg c : ns(16);
  always     { r     = ostr;    }
  sfg init   { c     = 0x1234;  }
  sfg sendin { a1_in = c;
               c     = (c[0] ^ c[2] ^ c[3] ^ c[5]) #
                           c[15:1];
               istr  = 1;       }
  sfg noin   { a1_in = 0;
               istr  = 0;       }
}
fsm ctl_t_median(t_median) {
  initial s0;
  state s1, s2;
  @s0 (init, noin)       -> s1;
  @s1 (sendin)           -> s2;
  @s2 if (r) then (noin) -> s1;
             else (noin) -> s2;
}

system S {
  t_median;
}
```

The simulation output of the 200 first clock cycles of Listing 5.21 is shown below. Each line of output shows the clock cycle at which a computation completes, followed by the input arguments a1, followed by the computed median q1. The output is shown in hexadecimal notation, and register values use the notation output/input. At a given output, the value of q1 is the median of the last five values entered. For example, at cycle 116, q1 has value b848, which is the median of the list (6e12, dc24, b848, 7091, e123). Refer to Appendix A for further details on installing GEZEL on a computer and running simulations.

```
/opt/gezel/bin/fdlsim m2.fdl 200
12 a1 1234/1234 q1 0
25 a1 91a/91a q1 0
38 a1 848d/848d q1 91a
51 a1 c246/c246 q1 1234
64 a1 e123/e123 q1 848d
77 a1 7091/7091 q1 848d
90 a1 b848/b848 q1 b848
103 a1 dc24/dc24 q1 c246
116 a1 6e12/6e12 q1 b848
129 a1 3709/3709 q1 7091
142 a1 1b84/1b84 q1 6e12
155 a1 8dc2/8dc2 q1 6e12
168 a1 46e1/46e1 q1 46e1
181 a1 2370/2370 q1 3709
194 a1 91b8/91b8 q1 46e1
```

5.6 Proper FSMD

A *proper FSMD* is one which has deterministic behavior. In general, a model with deterministic behavior is one which will always show the same response given the same initial state and the same input stimuli. In Chap. 2, we discussed how determinacy ensured that SDF graphs will compute the same result regardless of the schedule. Deterministic behavior is a desirable feature of most applications, including hardware/software codesign applications.

For a hardware FSMD implementation, deterministic behavior means that the hardware is free of race conditions. Without determinacy, a hardware model may end up in an unknown state (often represented using an 'X' in multi-valued logic hardware simulation) . A proper FSMD has no such race condition. It is obtained by enforcing four properties in the FSMD model. These properties are easy to check, both by the FSMD developer as well as by the simulation tools. The four properties are the following.

1. Neither registers nor signals can be assigned more than once during a clock cycle.
2. No circular definition exists between signals (wires).
3. If a signal is used as an operand of an expression, it must have a known value in the same clock cycle.
4. All datapath outputs must be defined (assigned) during all clock cycles.

The first rule is obvious, and ensures that there will be at most a single assignment per register/signal and per clock cycle. Recall from our earlier discussion that in a synchronous hardware model, all expressions are evaluated simultaneously according to the data dependencies of the expressions. If we allow multiple assignments per register/signal, the resulting value in the register or signal will become ambiguous.

The second rule ensures that any signal will carry a single, stable value during a clock cycle. Indeed, a circular definition between signals (e.g. as shown in Listing 5.4) may result in more then a single valid value. For example, circular definitions would occur when you try to model flip-flops with combinational logic (state). A proper FSMD model forces you to use `reg` for all state variables. Another case where you would end up with circular definition between signals is when you create ring-oscillators by wiring inverters in a loop. In a cycle-based hardware description language, all events happen at the pace of the global clock, and free-running oscillators cannot be modeled.

The third rule ensures that no signal can be used as an operand when the signal value would be undefined. Indeed, when an undefined signal is used as an operand in an expression, the result of the expression may become unknown. Such unknown values propagate in the model and introduce ambiguity on the outputs.

The fourth rule deals with hierarchy, and makes sure that rules 2 and 3 will hold even across the boundaries of datapaths. Datapath inputs and outputs have the same semantics as wires. The value of a datapath input will be defined by the datapath output connected to it. Rule 4 says that this datapath output will always have a known and stable value. Rule 4 is stricter than required. For example, if we don't read a datapath input during a certain clock cycle, the corresponding connected datapath output could remain undefined without causing trouble. However, requiring all outputs to be always defined is much easier to remember for the FSMD designer.

All of the above rules are enforced by the GEZEL simulation tools, either at runtime (through an error message), or else when the model is parsed. The resulting hardware created by these modeling rules is determinate and race-free.

5.7 Language Mapping for FSMD by Example

Even though we will be using the GEZEL language throughout this book for modeling of FSMD, all concepts covered so far are equally valid in other modeling languages including Verilog, VHDL or SystemC. We use GEZEL because of the following reasons.

- It is easier to set up cosimulation experiments in GEZEL. We will cover different types of hardware-software interfaces, and all of these are directly covered using GEZEL primitives.
- More traditional modeling languages include additional concepts (such as multi-valued logic and event-driven simulation), which, even though important by themselves, are less relevant in the context of a practical introduction to hardware-software codesign.
- GEZEL designs can be expanded into Verilog, VHDL or SystemC, as will be illustrated in this section. In fact, the implementation path of GEZEL works by converting these GEZEL designs into VHDL, and then using hardware synthesis on the resulting design.

The example we will discuss is the *binary* greatest common divisor (GCD) algorithm, a lightly optimized version of the classic GCD that makes use of the odd-even parity of the GCD operands.

5.7.1 GCD in GEZEL

Listing 5.22 Binary GCD in GEZEL

```
dp euclid(in  m_in, n_in : ns(16);
          out gcd        : ns(16)) {
  reg m, n               : ns(16);
  reg done               : ns(1);
  reg factor             : ns(16);

  sfg init    { m = m_in; n = n_in; factor = 0; done = 0;
                 gcd = 0;
                $display("cycle=", $cycle, " m=", m_in,   " n
                  =", n_in); }
  sfg shiftm  { m = m >> 1; }
  sfg shiftn  { n = n >> 1; }
  sfg reduce  { m = (m >= n) ? m - n : m;
                n = (n >  m) ? n - m : n; }
  sfg shiftf  { factor = factor + 1; }
  sfg outidle { gcd = 0; done = ((m == 0) | (n == 0)); }
  sfg complete{ gcd = ((m > n) ? m : n) << factor;
                $display("cycle=", $cycle, " gcd=",
                           gcd); }
}
```

```
fsm euclid_ctl(euclid) {
  initial s0;
  state s1, s2;

  @s0 (init) -> s1;
  @s1 if (done)               then (complete)          -> s2;
      else if ( m[0] &  n[0]) then (reduce, outidle)   -> s1;
      else if ( m[0] & ~n[0]) then (shiftn, outidle)   -> s1;
      else if (~m[0] &  n[0]) then (shiftm, outidle)   -> s1;
                              else (shiftn, shiftm,
                                    shiftf, outidle)   -> s1;
  @s2 (outidle) -> s2;
}
```

5.7.2 GCD in Verilog

Listing 5.23 Binary GCD in Verilog

```
module euclid(m_in, n_in, gcd, clk, rst);
  input  [15:0] m_in;
  input  [15:0] n_in;
  output [15:0] gcd;
  reg    [15:0] gcd;
  input  clk;
  input  rst;

  reg [15:0] m, m_next;
  reg [15:0] n, n_next;
  reg done, done_next;
  reg [15:0] factor, factor_next;
  reg [1:0] state, state_next;

  parameter s0 = 2’d0, s1 = 2’d1, s2 = 2’d2;

  always @(posedge clk)
   if (rst) begin
     n      <= 16’d0;
     m      <= 16’d0;
     done   <= 1’d0;
     factor <= 16’d0;
     state  <= s0;
   end else begin
     n      <= n_next;
     m      <= m_next;
     done   <= done_next;
     factor <= factor_next;
     state  <= state_next;
   end

  always @(*) begin
    n_next      <= n;       // default reg assignment
    m_next      <= m;       // default reg assignment
```

```
      done_next   <= done;    // default reg assignment
      factor_next <= factor;  // default reg assignment
      gcd         <= 16'd0;   // default output assignment

      case (state)

        s0: begin
            m_next      <= m_in;
            n_next      <= n_in;
            factor_next <= 16'd0;
            done_next   <= 1'd0;
            gcd         <= 16'd0;
            state_next <= s1;
            end

        s1: if (done) begin
               gcd <= ((m > n) ? m : n) << factor;
               state_next <= s2;
            end else if (m[0] & n[0]) begin
               m_next    <= (m >= n) ? m - n : m;
               n_next    <= (n >  m) ? n - m : n;
               gcd       <= 16'd0;
               done_next <= ((m == 0) | (n == 0));
               state_next <= s1;
            end else if (m[0] & ~n[0]) begin
               n_next <= n >> 1;
               gcd       <= 16'd0;
               done_next <= ((m == 0) | (n == 0));
               state_next <= s1;
            end else if (~m[0] & n[0]) begin
               m_next <= m >> 1;
               gcd       <= 16'd0;
               done_next <= ((m == 0) | (n == 0));
               state_next <= s1;
            end else begin
               n_next <= n >> 1;
               m_next <= m >> 1;
               factor_next <= factor + 1;
               gcd       <= 16'd0;
               done_next <= ((m == 0) | (n == 0));
               state_next <= s1;
            end

        s2: begin
            gcd       <= 16'd0;
            done_next <= ((m == 0) | (n == 0));
            state_next<= s2;
            end

        default: begin
                 state_next <= s0;  // jump back to init
                 end
      endcase
```

```
  end

endmodule
```

5.7.3 GCD in VHDL

Listing 5.24 Binary GCD in VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity gcd is
  port( m_in, n_in : in  std_logic_vector(15 downto 0);
        gcd        : out std_logic_vector(15 downto 0);
        clk, rst   : in  std_logic
      );
end gcd;

architecture behavior of gcd is
  type statetype is (s0, s1, s2);
  signal state, state_next : statetype;
  signal m, m_next : std_logic_vector(15 downto 0);
  signal n, n_next : std_logic_vector(15 downto 0);
  signal done, done_next : std_logic;
  signal factor, factor_next : std_logic_vector(15 downto 0);
  begin

  update_regs: process(clk, rst)
  begin
    if (rst='1') then
      m      <= (others => '0');
      n      <= (others => '0');
      done   <= '0';
      factor <= (others => '0');
      state  <= s0;
    elsif (clk='1' and clk'event) then
      state  <= state_next;
      m      <= m_next;
      n      <= n_next;
      done   <= done_next;
      factor <= factor_next;
      state  <= state_next;
    end if;
  end process;

  eval_logic: process(m_in, n_in, state)
  begin
  n_next      <= n;
  m_next      <= m;
  done_next   <= done;
  factor_next<= factor;
```

```
    gcd         <= (others => ’0’);

    case state is

      when s0 =>
         m_next      <= m_in;
         n_next      <= n_in;
         factor_next <= (others => ’0’);
         done_next   <= ’0’;
         gcd         <= (others => ’0’);
         state_next  <= s1;

      when s1 =>
         if (done = ’1’) then
            if (m > n) then
              gcd <= conv_std_logic_vector(shl(unsigned(m),
                  unsigned(factor)),16);
            else
              gcd <= conv_std_logic_vector(shl(unsigned(n),
                  unsigned(factor)),16);
            end if;
            state_next  <= s2;
         elsif ((m(0) = ’1’) and (n(0) = ’1’)) then
            if (m >= n) then
                m_next <= unsigned(m) - unsigned(n);
                n_next <= n;
            else
                m_next <= m;
                n_next <= unsigned(n) - unsigned(m);
            end if;
            gcd       <= (others => ’0’);
            if ((m = "0000000000000000") or
              (n = "0000000000000000")) then
               done_next <= ’1’;
            else
               done_next <= ’0’;
            end if;
            state_next <= s1;
         elsif ((m(0) = ’1’) and (n(0) = ’0’)) then
            n_next <= ’0’ & n(15 downto 1);
            gcd       <= (others => ’0’);
            if ((m = "0000000000000000") or
              (n = "0000000000000000")) then
               done_next <= ’1’;
            else
               done_next <= ’0’;
            end if;
            state_next <= s1;
         elsif ((m(0) = ’0’) and (n(0) = ’1’)) then
            m_next <= ’0’ & m(15 downto 1);
            gcd       <= (others => ’0’);
            if ((m = "0000000000000000") or
              (n = "0000000000000000")) then
```

```
             done_next <= '1';
          else
             done_next <= '0';
          end if;
          state_next <= s1;
       else
          n_next <= '0' & n(15 downto 1);
          m_next <= '0' & m(15 downto 1);
          factor_next <= conv_std_logic_vector(unsigned
                                   (factor) +
                        conv_unsigned(1,16),16);
          gcd       <= (others => '0');
          if ((m = "0000000000000000") or
            (n = "0000000000000000")) then
             done_next <= '1';
          else
             done_next <= '0';
          end if;
          state_next <= s1;
       end if;

    when s2 =>
      gcd <= (others => '0');
      if ((m = "0000000000000000") or
        (n = "0000000000000000")) then
         done_next <= '1';
      else
         done_next <= '0';
      end if;
      state_next<= s2;

    when others =>
      state_next <= s0;

    end case;
  end process;
end behavior;
```

5.7.4 GCD in SystemC

Listing 5.25 Binary GCD in SystemC

```
#include ``systemc.h''

enum statetype {s0, s1, s2};

SC_MODULE(gcd_fsmd) {
  sc_in <bool>          clk;
  sc_in <bool>          rst;
  sc_in <sc_uint<16> >  m_in, n_in;
  sc_out <sc_uint<16> > gcd;

  sc_signal<statetype>  state, state_next;
  sc_uint<16>           m,     m_next;
  sc_uint<16>           n,     n_next;
  sc_uint<16>           factor,factor_next;
  sc_uint< 1>           done,  done_next;

  void update_regs();
  void eval_logic();
  SC_CTOR(gcd_fsmd) {
    SC_METHOD(eval_logic);
    sensitive << m_in << n_in << state;
    SC_METHOD(update_regs);
    sensitive_pos << rst << clk;
  }
};

void gcd_fsmd::update_regs() {
  if (rst.read() == 1) {
    state  = s0;
    m      = 0;
    n      = 0;
    factor = 0;
    done   = 0;
  } else {
    state  = state_next;
    m      = m_next;
    n      = n_next;
    factor = factor_next;
    done   = done_next;
  }
}

void gcd_fsmd::eval_logic() {

  n_next      = n;
  m_next      = m;
  done_next   = done;
  factor_next = factor;
  gcd         = 0;

  switch(state) {
    case s0:
      m_next      = m_in;
      n_next      = n_in;
      factor_next = 0;
      done_next   = 0;
      gcd         = 0;
      state_next  = s1;
      break;
    case s1:
      if (done == 1) {
        gcd = ((m > n) ? m : n) << factor;
        state_next = s2;
      } else if (m[0] & n[0]) {
        m_next     = (m >= n) ? m - n : m;
        n_next     = (n >  m) ? n - m : n;
        gcd        = 0;
        done_next  = ((m == 0) | (n == 0));
        state_next = s1;
      } else if (m[0] & ~n[0]) {
        n_next     = (n >> 1);
        gcd        = 0;
        done_next  = ((m == 0) | (n == 0));
        state_next = s1;
      } else if (~m[0] & n[0]) {
        m_next     = m >> 1;
        gcd        = 0;
        done_next  = ((m == 0) | (n == 0));
        state_next = s1;
      } else {
        n_next     = n >> 1;
        m_next     = m >> 1;
        factor_next= factor + 1;
        gcd        = 0;
        done_next  = ((m == 0) | (n == 0));
        state_next = s1;
      }
      break;
    case s2:
      gcd  = 0;
      done_next  = ((m == 0) | (n == 0));
      break;
    default:
      state_next = s0;
   }
}
```

5.8 Summary

In this section, we discussed a synchronous hardware modeling mechanism, consisting of a datapath in combination with an FSM controller. The resulting model is called FSMD (Finite State Machine with Datapath). An FSMD models datapath instructions with expressions, and control with a state transition graph. Datapath expressions are created in terms of register variables and signals (wires). Register variables are implicitly attached to the global clock signal. Datapath instructions (groups of datapath expressions) form the connection between the controller and the datapath.

A given FSMD design is not unique. A given design can be decomposed into many different, equivalent FSMD descriptions. It is up to designer to pick a modeling style that feels natural and that is useful for the problem at hand.

We discussed a modeling syntax for FSMD called GEZEL. GEZEL models can be simulated and converted into synthesizable VHDL code. However, the

FSMD model is generic and can be captured into any suitable hardware description language. We demonstrated equivalent synthesizable implementations of an FSMD in GEZEL, Verilog, VHDL and SystemC.

5.9 Further Reading

The FSMD model has been recognized as a universal model for RTL modeling of hardware. See Vahid (2007a) for a textbook that starts from combinational and sequential logic, and gradually works up to FSMD based design. FSMD were popularized by Gajski, and are briefly covered in Gajski et al. (2009). Earlier, one can find an excellent development of the FSMD model in Davio et al. (1983).

Further details on installation and use of the GEZEL toolset are covered in Appendix A.

By using four easy-to-remember rules, one can construct *proper* FSMD, FSMD that will not have race conditions. A mathematical proof that these four rules are sufficient can be found in Schaumont et al. (2006).

There is an ongoing discussion on how to improve the productivity of hardware design. Some researchers believe that *high-level synthesis*, the automatic generation of RTL starting from high-level descriptions, is essential. Several academic and commercial design tools that support such high level synthesis are described in Gajski et al. (2009). See Gupta et al. (2004) for a detailed description of one such an environment. On the other hand, the nature of hardware design is such that designers like to think about clock cycles when they think about architecture. Hence, abstraction should be applied with utmost care. See Bluespec (Hoe 2000) and MADL (Qin 2004) for examples of such carefully abstracted hardware design and modeling paradigms.

5.10 Problems

Problem 5.1. Which of the circuits (a, b, c, d) in Fig. 5.11 can be simulated using a cycle-based simulator?

Problem 5.2. Design a high-speed sorter for four 32-bit registers (Fig. 5.12). Show how to create a sorting network for four numbers, using only simple two-input comparator modules. The comparator modules are built with combinational logic, and have a constant critical path. Optimize the critical path of the overall design, and create a maximally parallel implementation. You may make use of comparator modules, registers, and wiring. The input of the sorter comes from four registers marked ‘input’, the output of the sorter needs to be stored in four registers marked ‘output’.

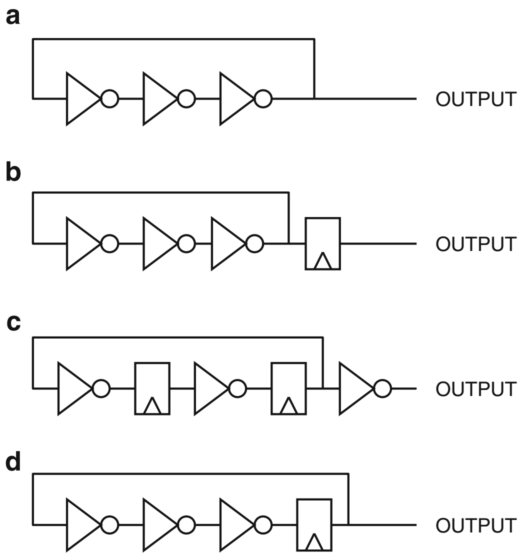


Fig. 5.11 Sample circuits for Problem 5.1

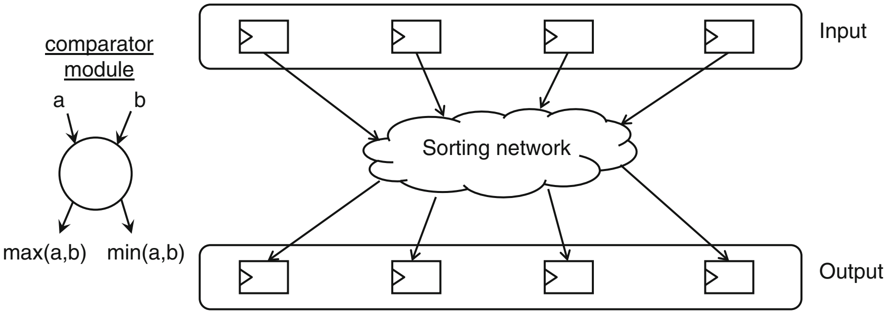


Fig. 5.12 Sorter design for Problem 5.2

Problem 5.3. Design a Finite State Machine that recognizes the pattern '10' and the pattern '01' in an infinite stream of bits. Make sure that the machine recognizes only one pattern at a time, and that it is not triggered by overlapping patterns. Figure 5.13 shows and example of the behavior of this FSM.

1. Draw a Mealy-type state diagram of this FSM.
2. Draw an RTL schematic of an implementation for this machine. Draw your implementation using registers and logic gates (AND, OR, NOT, and XOR). Make your implementation as compact as possible.

Problem 5.4. Design a Mealy-type finite state machine that recognizes either of the following two patterns: 1101 or 0111. The patterns should be read left to right (i.e. the leftmost bit is seen first), and they are to be matched into a stream of single bits.

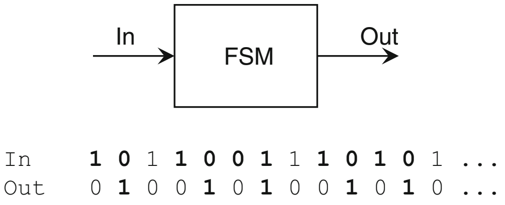

In
FSM
Out
In 1 0 1 1 0 0 1 1 1 0 1 0 1 ...
Out 0 1 0 0 1 0 1 0 0 1 0 1 0 ...

Fig. 5.13 Sorter design for Problem 5.2

Problem 5.5. Design an FSMD to divide natural numbers. The dividend and the divider each have 8 bits of resolution. The quotient must have 10 bits of resolution, and the remainder must have 8 bits of resolution. The divider has the following interface

```
dp divider(in  x     : ns(8);
           in  y     : ns(8);
           in  start : ns(1);
           out q     : ns(10);
           out r     : ns(8);
           out done  : ns(1)) {

 // Define the internals of the FSMD here ..

}
```

Given a dividend X and a divider Y, the divider will evaluate a quotient Q on p bits of precision and a remainder R such that

$$X.2^p = Q.Y + R \tag{5.1}$$

For example, if $p = 8, X = 12, Y = 15$, then a solution for Q and R is $Q = 204$ and $R = 12$ because $12 \,.\, 28 = 204.15 + 12$.

Your implementation must obtain the quotient and remainder within 32 clock cycles. To implement the divider, you can use the restoring division algorithm as follows. The basic operation evaluates a single bit of the quotient according to the following pseudo-code:

```
basic_divider(input a, b;
              output q, r) {
   z := 2 * a - b;
   if (z < 0) then
      q = 0;
      r = 2 * a;
   else
      q = 1;
      r = z;
}
```

To evaluate the quotient over p bits, you repeat the basic 1-bit divider p times as follows.

```
r(0) = X;
for i is 1 to p do
basic_divider(r(i-1), Y, q(i), r(i));
```

Each iteration creates 1 bit of the quotient, and the last iteration returns the remainder. For example, if $p = 8$, then $Q = q(0), q(1), q(2), \ldots, q(7)$ and $R = r(8)$.

Create a hardware implementation which evaluates 1 bit of the quotient per clock cycle.

Problem 5.6. How many flip-flops and how many adders do you need to implement the FSMD description in Listing 5.26? Count each single bit in each register, and assume binary encoding of the FSM state, to determine the flip-flop count.

Listing 5.26 Program for Problem 5.6

```
dp mydp(in i : ns(5); out o : ns(5)) {
  reg a1, a2, a3, a4 : ns(5);
  sfg f1 { a1 = i;
           a2 = 0;
           a3 = 0;
           a4 = 0; }
  sfg f2 { a1 = a2 ? (a1 + a3) : (a1 + a4); }
  sfg f3 { a3 = a3 + 1; }
  sfg f4 { a4 = a4 + 1; }
  sfg f5 { a2 = a2 + a1; }
}
fsm mydp_ctl(mydp) {
  initial s0;
  state s1, s2;
  @s0 (f1) -> s1;
  @s1 if (a1) then (f2, f3) -> s2;
              else (f4)     -> s1;
  @s2 if (a3) then (f2)     -> s1;
              else (f5)     -> s2;
}
```

Problem 5.7. FSMD models provide modeling of control (conditional execution) as well as data processing in hardware. Therefore, it is easy to mimic the behavior of a C program and build and FSMD that reflects the same control flow as the C program. Write an FSMD model for the C function shown in Listing 5.27. Assume that the arguments of the function are the inputs of the FSMD, and that the result of the function is the FSMD output. Develop your model so that you need no more then a single multiplier.

Listing 5.27 Program for Problem 5.7

```
int filter(int a) {
  static int taps[5];
  int c[] = {-1, 5, 10, 5, -1};
  int r;

  for (i=0; i<4; i++)
    taps[i] = taps[i+1];
  taps[4] = a;

  r = 0;
  for (i=0; i<5; i++)
    r = r + taps[i] * c[i];

  return r;
}
```

To model an array of constants in GEZEL, you can make use of the lookup table construct as follows:

```
dp lookup_example {

   lookup T : ns(8) = {5, 4, 3, 2, 1, 1, 1, 1};

   sig a, b : ns(3);

   always {
     a = 3;
     b = T[a]; // this assigns the fourth element of T to b
   }
}
```

Problem 5.8. Repeat Problem 5.7, but develop your FSMD so that the entire function completes in a single clock cycle.

Problem 5.9. Write the FSMD of Listing 5.28 in a single always block. This FSMD presents a Galois Field multiplier.

Listing 5.28 Program for Problem 5.9

```
dp D( in fp, i1, i2 : ns(4); out mul: ns(4);
      in mul_st: ns(1);
      out mul_done : ns(1)) {
  reg acc, sr2, fpr, r1 : ns(4);
  reg mul_st_cmd : ns(1);
  sfg ini { // initialization
     fpr        = fp;
     r1         = i1;
     sr2        = i2;
    acc        = 0;
    mul_st_cmd = mul_st;
  }
  sfg calc { // calculation
    sr2 = (sr2 << 1);
    acc = (acc << 1) ^ (r1 & (tc(1))  sr2[3])  // add a if b=1
            ^ (fpr & (tc(1)) acc[3]); // reduction if carry
  }
  sfg omul { // output inactive
    mul      = acc;
    mul_done = 1;
    $display(``done. mul='', mul);
  }
  sfg noout { // output active
    mul      = 0;
    mul_done = 0;
  }
}
fsm F(D) {
  state s1, s2, s3, s4, s5;
  initial   s0;
  @s0 (ini,  noout) -> s1;
  @s1 if (mul_st_cmd) then (calc, noout) -> s2;
                      else (ini, noout)  -> s1;
  @s2 (calc, noout) -> s3;
  @s3 (calc, noout) -> s4;
  @s4 (calc, noout) -> s5;
  @s5 (ini,  omul ) -> s1;
}
```

Problem 5.10. In this chapter, we discussed how FSM can be expressed as datapath expressions (See Sect. 5.4.3 and Problem 5.8). It is also possible to go the opposite way, and model datapaths in terms of finite state machines. Write an FSM for the datapath shown in Listing 5.29.

Listing 5.29 Program for Problem 5.10

```
dp tester(out o: ns(2)) {
  reg a1 : ns(1);
  reg a2 : ns(2);
  always {
    a1 = a1 + 1;
    a2 = a2 + a1;
    o  = a2;
  }
}
```

Chapter 6
Microprogrammed Architectures

6.1 Limitations of Finite State Machines

Finite State Machines are well suited to capture the control flow and decision-making of algorithms. FSM state transition diagrams even resemble control dependency graphs (CDG). Nevertheless, FSM are no universal solution for control modeling. They suffer from several modeling weaknesses, especially when dealing with complex controllers.

A key issue is that FSMs are a flat model, without any hierarchy. A *flat* control model is like a C program consisting of just one single `main` function. Real systems do not use a flat control model: they need a control hierarchy. Many of the limitations of FSMs stem from their lack of hierarchy.

6.1.1 State Explosion

A flat FSM suffers from *state explosion*, which occurs when multiple independent activities interfere in a single model. Assume that a FSM has to capture two independent activities, each of which can be in one of three states. The resulting FSM, called a *product state-machine*, needs nine states to capture the control flow of the overall model. The product state-machine needs to keep track of the current state from two independent state machines at the same time. Figure 6.1 illustrates the effect of state explosion in a product state-machine. Two state machines, `FSM1` and `FSM2`, need to be merged into a single product state-machine `FSM1xFSM2`. Due to conditional state transitions, one state machine can remain in a single state while the other state machine proceeds to the next state. This results in multiple intermediate states such as `A1`, `A2`, and `A3`. In order to represent all individual states, nine states are needed in total. The resulting number of state transitions (and state transition conditions) is even higher. Indeed, if we have *n* independent state transition conditions in the individual state machines, the resulting product

P.R. Schaumont, *A Practical Introduction to Hardware/Software Codesign*,
DOI 10.1007/978-1-4614-3737-6_6, © Springer Science+Business Media New York 2013

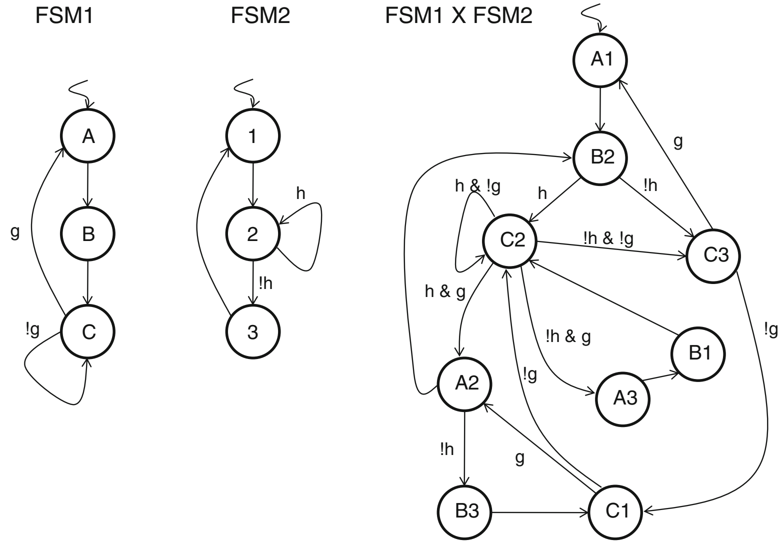


Fig. 6.1 State explosion in FSM when creating a product state-machine

state-machine can have up to 2^n state transition conditions. Of course, there have been several proposals for hierarchical modeling extensions for FSMs, such as the Statecharts from David Harel. Currently, however, none of these are widely used for hardware design.

6.1.2 *Exception Handling*

A second issue with a flat FSM is the problematic handling of exceptions. An exception is a condition which may cause an immediate state transition, regardless of the current state of the finite state machine. The purpose of an exception is to abort the regular flow of control and to transfer control to a dedicated exception-handler. An exception may have internal causes, such as an overflow condition in a datapath, or external causes, such as an interrupt. Regardless of the cause, the effect of an exception on a finite state machine model is dramatic: an additional state transition needs to be added to every state of the finite state machine. For example, assume that the product state-machine in Fig. 6.1 needs to include an exception input called `exc`, and that the assertion of that input requires immediate transition to state `A1`. The resulting FSM, shown in Fig. 6.2, shows how exceptions degrade the clarity of the FSM state transition graph.

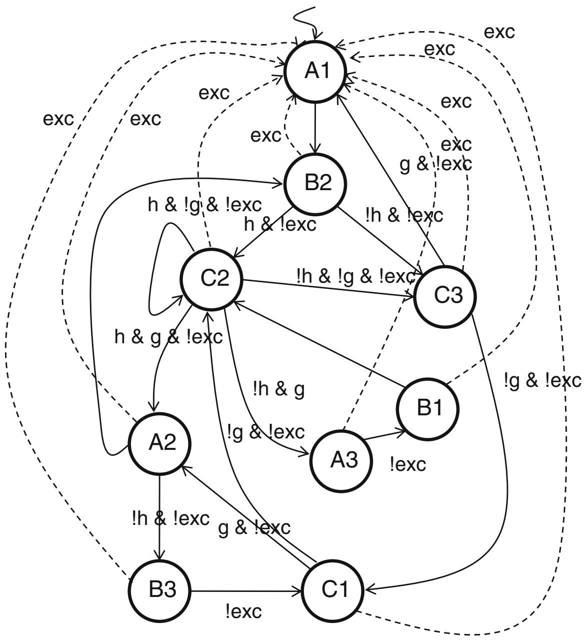


Fig. 6.2 Adding a single global exception deteriorates the readability of the FSM significantly

6.1.3 Runtime Flexibility

A major concern, from the viewpoint of hardware-software codesign, is that a finite state machine is a non-flexible model. Once the states and the state transitions are defined, the control flow of the FSM is fixed. The hardware implementation of an FSM leads to a hardwired controller that cannot be modified after implementation.

As a result, designers have proposed improved techniques for specifying and implementing control, in order to deal with flexibility, exceptions, and hierarchical modeling. Microprogramming is one such a technique. Originally introduced in the 1950s by Maurice Wilkes as a means to create a programmable instruction-set for mainframe computers, it became very popular in the 1970s and throughout the 1980s. Microprogramming was found to be very useful to develop complex microprocessors with flexible instruction-sets. Nowadays, microprogramming is no longer popular; microprocessors are so cheap that they do no longer need a reprogrammable instruction-set. However, microprogramming is still very useful as a design technique to introduce flexibility in a hardware design.

6.2 Microprogrammed Control

Figure 6.3 shows a micro-programmed machine next to an FSMD. The fundamental idea in microprogramming is to replace the next-state logic of the finite

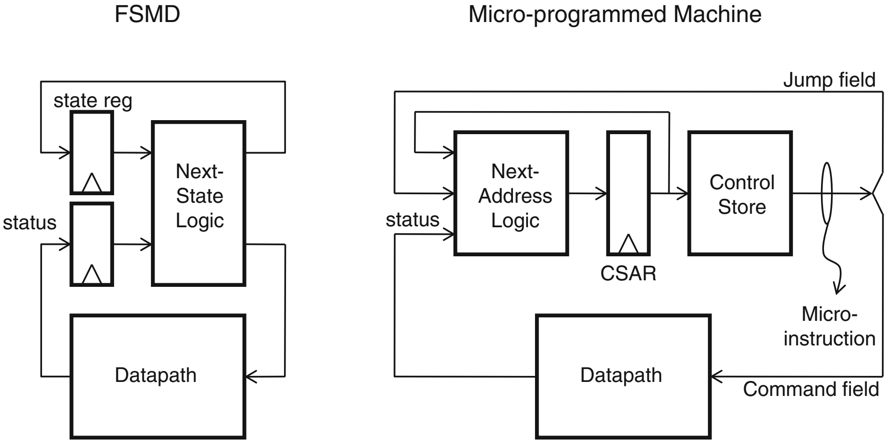


Fig. 6.3 In contrast to FSM-based control, microprogramming uses a flexible control scheme

state-machine with a *programmable* memory, called the control store. The control store holds micro-instructions, and is addressed using a register called CSAR (Control Store Address Register). The CSAR is the equivalent of a program counter in microprocessors. The next-value of CSAR is determined by the next-address logic, using the current value of CSAR, the current micro-instruction and the value of status flags evaluated by the datapath. The default next-value of the CSAR corresponds to the previous CSAR value incremented by one. However, the next-address logic can also implement conditional jumps or immediate jumps.

Thus, the next-address logic, the CSAR, and the control store implement the equivalent of an instruction-fetch cycle in a microprocessor. In the design of Fig. 6.3, each micro-instruction takes a single clock cycle to execute. Within a single clock cycle, the following activities occur.

- The CSAR provides an address to the control store which retrieves a micro-instruction. The micro-instruction is split in two parts: a command-field and a jump-field. The command-field serves as a command for the datapath. The jump-field goes to the next-address logic.
- The datapath executes the command encoded in the micro-instruction, and returns status information to the next-address logic.
- The next-address logic combines the micro-instruction jump-field, the previous value of the CSAR, and the datapath status. The next-address logic updates the CSAR. Consequently, the critical path of the micro-programmed machine in Fig. 6.3 is determined by the combined logic delay through the control store, the next-address logic, and the datapath.

While the micro-programmed controller is more complicated than a finite state machine, it also addresses the problems of FSMs:

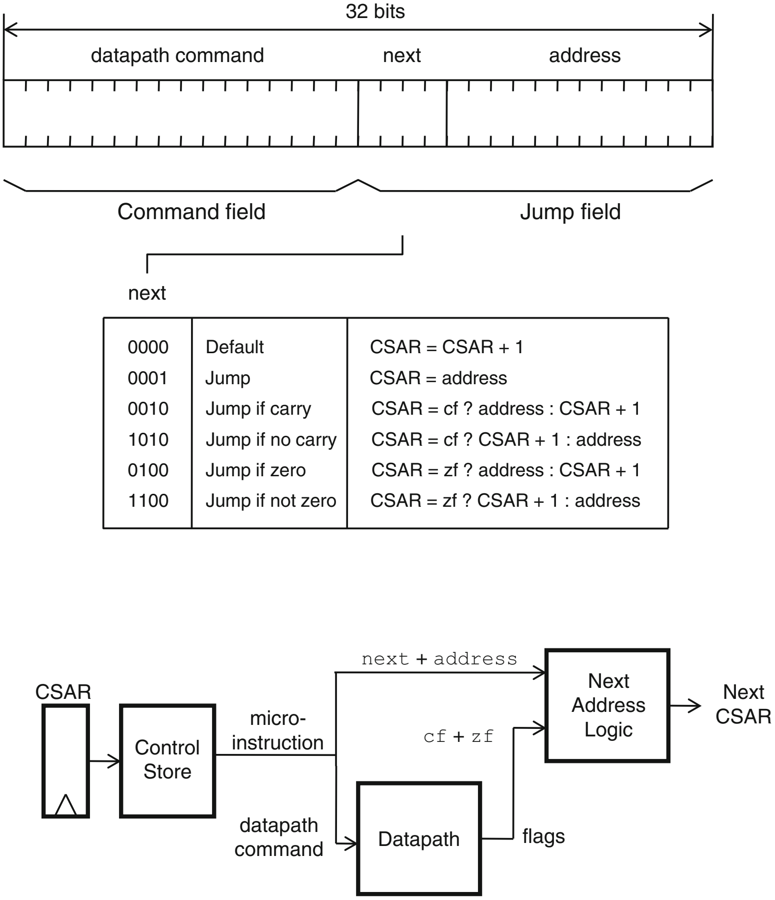
1. The micro-programmed controller scales well with complexity. For example, a 12-bit CSAR enables a control store with up the 4096 locations, and therefore a micro-program with 4096 steps. An equivalent FSM diagram with 4096 states, on the other hand, would be horrible to draw!
2. A micro-programmed machine deals very well with control hierarchy. With small modifications to the microprogrammed machine in Fig. 6.3, we can save the CSAR in a separate register or on a stack memory, and later restore it. This requires the definition of a separate micro-instruction to call a subroutine as well as a second micro-instruction to return from it.
3. A micro-programmed machine can deal efficiently with exception handling, since global exceptions are managed directly by the next-address logic, independently from the control store. For example, the presence of a global exception can feed a hard-coded value into the CSAR, immediately transferring the micro-programmed machine to an exception-handling routine. Exception handling in a micro-programmed machine is similar to a jump instruction, but it does not affect every instruction of the micro-program in the same way as it affects every state of a finite state machine.
4. Finally, micro-programs are flexible and very easy to change after the micro-programmed machine is implemented. Simply changing the contents of the control store is sufficient to change the program of the machine. In a micro-programmed machine, there is a clear distinction between the architecture of the machine and the functionality implemented using that architecture.

6.3 Micro-instruction Encoding

How should we define the encoding used by micro-instructions? In this section, we will discuss the design trade-offs that determine the micro-instruction format.

6.3.1 Jump Field

Figure 6.4 shows a 32-bit micro-instruction word, with 16 bits reserved for the datapath, and 16 bits reserved for the next-address logic. Let's first consider the part for the next-address logic. The `address` field holds an absolute target address, pointing to a location in the control store. In this case, the address is 12 bit, which means that this micro-instruction format would be suited for a control store with 4096 locations. The `next` field encodes the operation that will lead to the next value of CSAR. The default operation is, as discussed earlier, to increment CSAR. For such instructions, the address field remains unused. Otherwise, `next` will be used to encode various jump instructions. An absolute jump will transfer the value of the `address` field into CSAR. A conditional jump will use the value of a flag to conditionally update the CSAR or else increment it.



0000	Default	CSAR = CSAR + 1
0001	Jump	CSAR = address
0010	Jump if carry	CSAR = cf ? address : CSAR + 1
1010	Jump if no carry	CSAR = cf ? CSAR + 1 : address
0100	Jump if zero	CSAR = zf ? address : CSAR + 1
1100	Jump if not zero	CSAR = zf ? CSAR + 1 : address

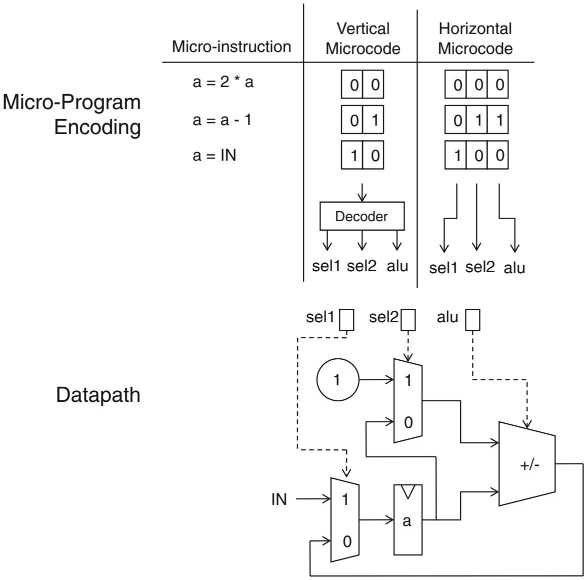
Fig. 6.4 Sample format for a 32-bit micro-instruction word

Obviously, the format as shown is quite bulky, and may consume a large amount of storage. The `address` field, for example, is only used for jump instructions. If the micro-program contains only a few jump instructions, then the storage for the address field would be wasted. To avoid this, we will need to optimize the encoding format for micro-instructions. In micro-instructions other than jumps, the bits used for the `address` field could be given a different purpose.

6.3.2 Command Field

The design of the datapath command format reveals another interesting trade-off: we can either opt for a very wide micro-instruction word, or else we can prefer a

Fig. 6.5 Example of vertical versus horizontal micro-programming



narrow micro-instruction word. A wide micro-instruction word allows each control bit of the data path to be stored separately. A narrow micro-instruction word, on the other hand, will require the creation of ‘symbolic instructions’, which are encoded groups of control-bits for the datapath. The FSMD model relies on such symbolic instructions. Each of the above approaches has a specific name. *Horizontal* micro-instructions use no encoding at all. It represents each control bit in the datapath with a separate bit in the micro-instruction format. *Vertical* micro-instructions on the other hand encode the control bits for the datapath as much as possible. A few bits of the micro-instruction can define the value of many more control bits in the data-path.

Figure 6.5 demonstrates an example of vertical and horizontal micro-instructions in the datapath. We wish to create a micro-programmed machine with three instructions on a single register a. The three instructions do one of the following: double the value in a, decrement the value in a, or initialize the value in a. The datapath shown on the bottom of Fig. 6.5 contains two multiplexers and a programmable adder/subtractor. It can be easily verified that each of the instructions enumerated above can be implemented as a combination of control bit values for each multiplexer and for the adder/subtractor. The controller on top shows two possible encodings for the three instructions: a horizontal encoding, and a vertical encoding.

- In the case of vertical microcode, the micro-instructions will be encoded. Since there are three different instructions, we can implement this machine with a 2-bit micro-instruction word. To generate the control bits for the datapath, we will have to decode each of the micro-instruction words into local control signals on the datapath.

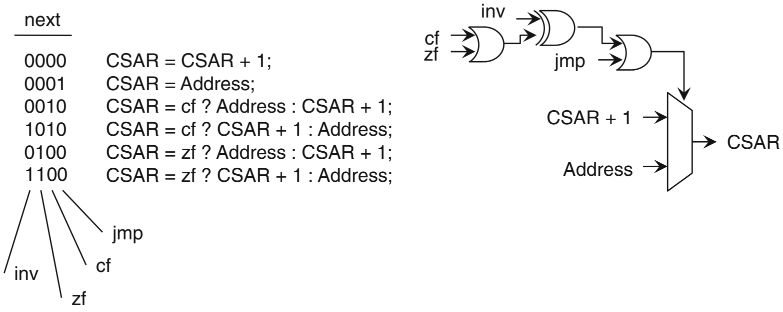

next
0000 CSAR = CSAR + 1;
0001 CSAR = Address;
0010 CSAR = cf ? Address : CSAR + 1;
1010 CSAR = cf ? CSAR + 1 : Address;
0100 CSAR = zf ? Address : CSAR + 1;
1100 CSAR = zf ? CSAR + 1 : Address;
inv
zf
cf
jmp
inv
cf
zf
jmp
CSAR + 1
Address
CSAR

Fig. 6.6 CSAR encoding

- In the case of horizontal microcode, the control store will include each of the control bits in the datapath as a bit in the micro-instruction word. Hence, in this case, the encoding of the instructions reflects exactly the required setting of datapath elements for each micro-instruction.

We can describe the design trade-off between horizontal and vertical micro-programs as follows. Vertical micro-programs have a better code density, which is beneficial for the size of the control store. In Fig. 6.5, the vertically-encoded version of the microprogram will be only 2/3 of the size of the horizontally-encoded version. On the other hand, vertical micro-programs use an additional level of encoding, so that each micro-instruction needs to be decoded before it can drive the control bits of the datapath. Thus, the machine with the vertically encoded micro-program may be slower.

Obviously, the choice between a vertical and horizontal encoding needs to be made carefully. In practice, designers use a combination of vertical and horizontal encoding concepts, so that the resulting architecture is compact yet efficient. Consider for example the value of the `next` field of the micro-instruction word in Fig. 6.4. There are six different types of jump instructions, which would imply that a vertical micro-instruction would need no more then 3 bits to encode these six jumps. Yet, 4 bits have been used, indicating that there is some redundancy left. The encoding was chosen to simplify the design of the next-address logic, which is shown in Fig. 6.6. Such spare room in the micro-instruction encoding also supports future upgrades.

6.4 The Micro-programmed Datapath

The datapath of a micro-programmed machine consists of three elements: computation units, storage, and communication buses. Each of these may contribute a few control bits to the micro-instruction word. For example, multi-function computation units have function-selection bits, storage units have address bits and read/write

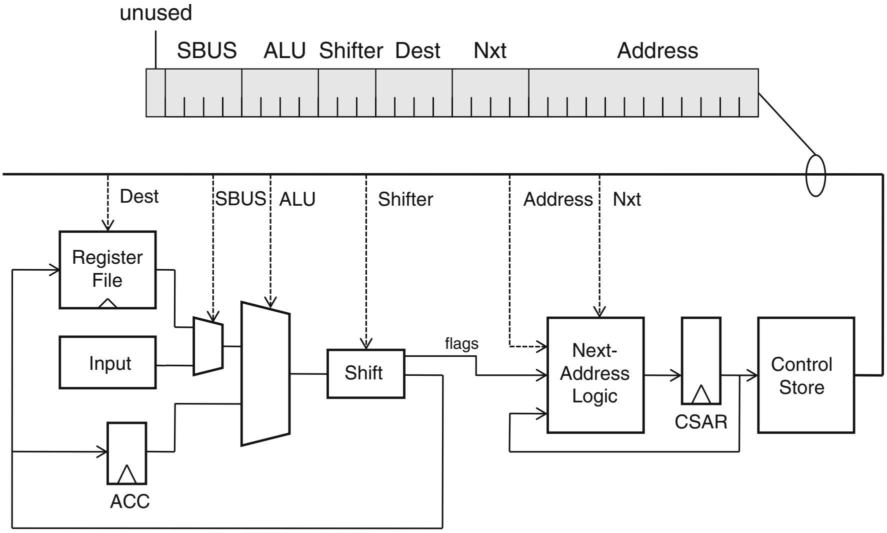


Fig. 6.7 A micro-programmed datapath

command bits, and communication busses have source/destination control bits. The datapath may also generate status flags for the micro-programmed controller.

6.4.1 Datapath Architecture

Figure 6.7 illustrates a micro-programmed controller with a datapath attached. The datapath includes an ALU with shifter unit, a register file with eight entries, an accumulator register, and an input port. The micro-instruction word is shown on top of Fig. 6.7, and contains six fields. Two fields, `Nxt` and `Address`, are used by the micro-programmed controller. The other are used by the datapath. The type of encoding is mixed horizontal/vertical. The overall machine uses a horizontal encoding: each module of the machine is controlled independently. The sub-modules within the machine on the other hand use a vertical encoding. For example, the `ALU` field contains 4 bits. In this case, the ALU component in the datapath will execute up to 16 different commands.

The machine completes a single instruction per clock cycle. The ALU combines an operand from the accumulator register with an operand from the register file or the input port. The result of the operation is returned to the register file or the accumulator register. The communication used by datapath operations is controlled by two fields in the micro-instruction word. The *SBUS* field and the *Dest* field indicate the source and destination respectively.

Table 6.1 Micro-instruction encoding of the example machine

Field	Width	Encoding			
SBUS	4	Selects the operand that will drive the S-Bus			
		0000	R0	0101	R5
		0001	R1	0110	R6
		0010	R2	0111	R7
		0011	R3	1000	Input
		0100	R4	1001	Address/Constant
ALU	4	Selects the operation performed by the ALU			
		0000	ACC	0110	ACC — S-Bus
		0001	S-Bus	0111	not S-Bus
		0010	ACC + SBus	1000	ACC + 1
		0011	ACC − SBus	1001	SBus − 1
		0100	SBus − ACC	1010	0
		0101	ACC & S-Bus	1011	1
Shifter	3	Selects the function of the programmable shifter			
		000	logical SHL(ALU)	100	arith SHL(ALU)
		001	logical SHR(ALU)	101	arith SHR(ALU)
		010	rotate left ALU	111	ALU
		011	rotate right ALU		
Dest	4	Selects the target that will store S-Bus			
		0000	R0	0101	R5
		0001	R1	0110	R6
		0010	R2	0111	R7
		0011	R3	1000	ACC
		0100	R4	1111	unconnected
Nxt	4	Selects next-value for CSAR			
		0000	CSAR + 1	1010	cf ? CSAR + 1 : Address
		0001	Address	0100	zf ? Address : CSAR + 1
		0010	cf ? Address : CSAR + 1	1100	zf ? CSAR + 1 : Address

The Shifter module also generates flags, which are used by the micro-programmed controller to implement conditional jumps. Two flags are created: a zero-flag, which is high (1) when the output of the shifter is all-zero, and a carry-flag, which contains the bit shifted-out at the most-significant position.

6.4.2 *Writing Micro-programs*

Table 6.1 illustrates the encoding used by each module of the design from Fig. 6.7. A micro-instruction can be formed by selecting a module function for each module of the micro-programmed machine, including a next-address for the Address field. When a field remains unused during a particular instruction, a don't care value can be chosen. The don't care value should be carefully selected so that unwanted state changes in the datapath are avoided.

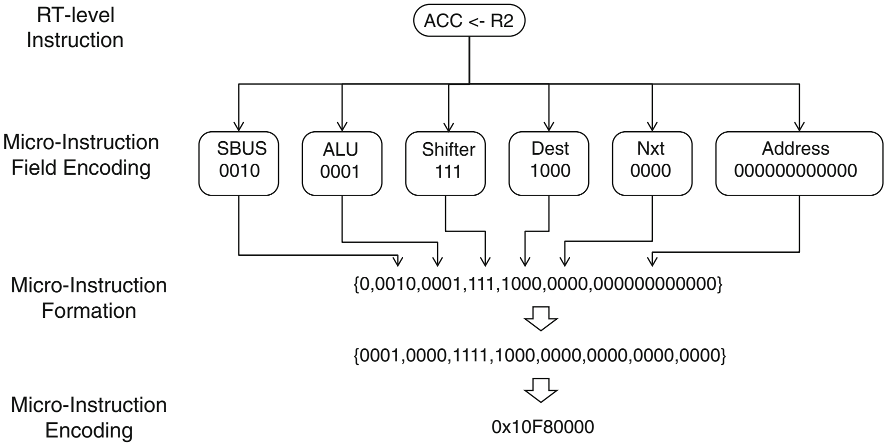


Fig. 6.8 Forming micro-instructions from register-transfer instructions

For example, an instruction to copy register R2 into the accumulator register ACC would be formed as illustrated in Fig. 6.8. The instruction should read out register R2 from the register file, pass the register contents over the SBus, through the ALU and the shifter, and write the result in the ACC register. This observation allows to determine the value of each field in the micro-instruction.

- The SBus needs to carry the value of R2. Using Table 6.1 we find SBUS equals 0010.
- The ALU needs to pass the SBus input to the output. Based on Table 6.1, ALU must equal 0001.
- The shifter passes the ALU output unmodified, hence Shifter must equal 111.
- The output of the shifter is used to update the accumulator register, so the Dest field equals 1000.
- Assuming that no jump or control transfer is executed by this instruction, the next micro-instruction will simply be one beyond the current CSAR location. This implies that Nxt should equal 0000 and Address is a don't-care, for example all-zeroes.
- Finally, we can find the overall micro-instruction code by putting all instruction fields together. Figure 6.8 illustrates this process. We conclude that a micro-instruction to copy R2 into ACC can be encoded as 0x10F80000 in the control store.

Writing a micro-program thus consists of formulating the desired behavior as a sequence of register transfers, and next encoding these register transfers as micro-instruction fields. More complex control operations, such as loops and if-then-else statements, can be expressed as a combination (or sequence) of register transfers.

As an example, let's develop a micro-program that reads two numbers from the input port and that evaluates their greatest common divisor (GCD) using Euclid's algorithm. The first step is to develop a micro-program in terms of register transfers. A possible approach is shown in Listing 6.1. Lines 2 and 3 in this program read in two values from the input port, and store these values in registers R0 and ACC. At the end of the program, the resulting GCD will be available in either ACC or R0, and the program will continue until both values are equal. The stop test is implemented in line 4, using a subtraction of two registers and a conditional jump based on the zero-flag. Assuming both registers contain different values, the program will continue to subtract the largest register from the smallest one. This requires to find which of R0 and ACC is bigger, and it is implemented with a conditional jump in Line 5. The bigger-then test is implemented using a subtraction, a left-shift and a test on the resulting carry-flag. If the carry-flag is set, then the most-significant bit of the subtraction would be one, indicating a negative result in two's complement logic. This conditional jump-if-carry will be taken if R0 is smaller then ACC. The combination of lines 5–7 shows how an if-then-else statement can be created using multiple conditional and unconditional jump instructions. When the program is complete, in Line 8, it spins in an infinite loop.

Listing 6.1 Micro-program to evaluate a GCD

```
1  ;   Command Field       || Jump Field
2          IN -> R0
3          IN -> ACC
4  Lcheck: (R0 - ACC)      || JUMP_IF_Z Ldone
5          (R0 - ACC) << 1 || JUMP_IF_C LSmall
6          R0 - ACC -> R0  || JUMP Lcheck
7  Lsmall: ACC - R0 -> ACC || JUMP Lcheck
8  Ldone:                     JUMP Ldone
```

6.5 Implementing a Micro-programmed Machine

In this section, we discuss a sample implementation of a micro-programmed machine in the GEZEL language. It can be used as a template for other implementations.

6.5.1 Micro-instruction Word Definition

A convenient starting point in the design of a micro-programmed machine is the definition of the micro-instruction. This includes the allocation of micro-instruction control bits, and their encoding.

The individual control fields are defined as subvectors of the micro-instruction. Listing 6.2 shows the GEZEL implementation of the micro-programmed design discussed in the previous section. The possible values for each micro-instruction field are shown in Lines 5–71. The use of C macro's simplifies the writing of micro-programs.

The formation of a single micro-instruction is done using a C macro as well, shown in Lines 74–81. Lines 83–136 show the micro-programmed controller, which includes a control store with a micro-program and the next-address CSAR logic. The control store is a lookup table with a sequence of micro-instructions (lines 90–105). On line 115, a micro- instruction is fetched from the control store, and broken down into individual fields which form the output of the microprogrammed controller (lines 116–122). The next-address logic uses the next-address control field to find a new value for CSAR each clock cycle (lines 124–134).

The micro-programmed machine includes several data-paths, including a register file (lines 137–168), an ALU (lines 170–192), a shifter (lines 194–221). Each of the data-paths is crafted along a similar principle: based on the control field input, the data-input is transformed into a corresponding data-output. The decoding process of control fields is visible as a sequence of ternary selection-operators.

The top-level cell for the micro-programmed machine is contained in lines 223–254. The top-level includes the controller, a register file, an ALU and a shifter. The top-level module also defines a data-input port and a data-output port, and each has a strobe control signal that indicates a data-transfer. The strobe signals are generated by the top-level module based decoding of micro-instruction fields. The input strobe is generated when the SBUS control field indicates that the SBUS will be reading an external input. The output strobe is generated by a separate, dedicated micro-instruction bit.

A simple testbench for the top-level cell is shown on lines 256–276. The testbench feeds in a sequence of data to the micro-programmed machine, and prints out each number appearing at the data output port. The micro-program for this machine evaluates the GCD of each tuple in the list of numbers shown on line 264.

Listing 6.2 Micro-programmed controller in GEZEL

```
 1  // wordlength in the datapath
 2  #define WLEN 16
 3
 4  /* encoding for data output */
 5  #define O_NIL     0    /* OT <- 0 */
 6  #define O_WR      1    /* OT <- SBUS */
 7
 8  /* encoding for SBUS multiplexer */
 9  #define SBUS_R0   0    /* SBUS <- R0  */
10  #define SBUS_R1   1    /* SBUS <- R1  */
11  #define SBUS_R2   2    /* SBUS <- R2  */
12  #define SBUS_R3   3    /* SBUS <- R3  */
13  #define SBUS_R4   4    /* SBUS <- R4  */
14  #define SBUS_R5   5    /* SBUS <- R5  */
15  #define SBUS_R6   6    /* SBUS <- R6  */
16  #define SBUS_R7   7    /* SBUS <- R7  */
17  #define SBUS_IN   8    /* SBUS <- IN  */
18  #define SBUS_X SBUS_R0 /* don't care */
19
```

```
/* encoding for ALU */
#define ALU_ACC  0    /* ALU <- ACC */
#define ALU_PASS 1    /* ALU <- SBUS */
#define ALU_ADD  2    /* ALU <- ACC + SBUS */
#define ALU_SUBA 3    /* ALU <- ACC - SBUS */
#define ALU_SUBS 4    /* ALU <- SBUS - ACC */
#define ALU_AND  5    /* ALU <- ACC and SBUS */
#define ALU_OR   6    /* ALU <- ACC or SBUS */
#define ALU_NOT  7    /* ALU <- not SBUS */
#define ALU_INCS 8    /* ALU <- ACC + 1 */
#define ALU_INCA 9    /* ALU <- SBUS - 1 */
#define ALU_CLR  10   /* ALU <- 0 */
#define ALU_SET  11   /* ALU <- 1 */
#define ALU_X ALU_ACC /* don't care */

/* encoding for shifter */
#define SHFT_SHL 1   /* Shifter <- shiftleft(alu) */
#define SHFT_SHR 2   /* Shifter <- shiftright(alu) */
#define SHFT_ROL 3   /* Shifter <- rotateleft(alu) */
#define SHFT_ROR 4   /* Shifter <- rotateright(alu) */
#define SHFT_SLA 5   /* Shifter <- shiftleftarithmetical
                         (alu) */
#define SHFT_SRA 6   /* Shifter <- shiftrightarithmetical
                         (alu) */
#define SHFT_NIL 7   /* Shifter <- ALU */
#define SHFT_X SHFT_NIL /* don't care */

/* encoding for result destination */
#define DST_R0  0       /* R0  <- Shifter */
#define DST_R1  1       /* R1  <- Shifter */
#define DST_R2  2       /* R2  <- Shifter */
#define DST_R3  3       /* R3  <- Shifter */
#define DST_R4  4       /* R4  <- Shifter */
#define DST_R5  5       /* R5  <- Shifter */
#define DST_R6  6       /* R6  <- Shifter */
#define DST_R7  7       /* R7  <- Shifter */
#define DST_ACC 8       /* IR  <- Shifter */
#define DST_NIL 15      /* not connected <- shifter */
#define DST_X   DST_NIL /* don't care instruction */

/* encoding for command field */
#define NXT_NXT  0   /* CSAR <- CSAR + 1 */
#define NXT_JMP  1   /* CSAR <- Address */
#define NXT_JC   2   /* CSAR <- (carry==1)? Address : CSAR
                              + 1 */
#define NXT_JNC 10   /* CSAR <- (carry==0)? Address : CSAR
                              + 1 */
#define NXT_JZ   4   /* CSAR <- (zero==1) ? Address : CSAR
                              + 1 */
#define NXT_JNZ 12   /* CSAR <- (zero==0) ? Address : CSAR
                              + 1 */
#define NXT_X NXT_NXT

```

```
/* encoding for the micro-instruction word */
#define MI(OUT, SBUS, ALU, SHFT, DEST, NXT, ADR) \
    (OUT   << 31) | \
    (SBUS  << 27) | \
    (ALU   << 23) | \
    (SHFT  << 20) | \
    (DEST  << 16) | \
    (NXT   << 12) | \
    (ADR)

dp control(in  carry, zero : ns(1);
           out ctl_ot      : ns(1);
           out ctl_sbus    : ns(4);
           out ctl_alu     : ns(4);
           out ctl_shft    : ns(3);
           out ctl_dest    : ns(4)) {

  lookup cstore : ns(32) = {
   // 0 Lstart: IN -> R0
   MI(O_NIL, SBUS_IN, ALU_PASS, SHFT_NIL,DST_R0, NXT_NXT,0),
   // 1         IN -> ACC
   MI(O_NIL, SBUS_IN, ALU_PASS, SHFT_NIL,DST_ACC,NXT_NXT,0),
   // 2 Lcheck: (R0 - ACC)       || JUMP_IF_Z Ldone
   MI(O_NIL, SBUS_R0, ALU_SUBS, SHFT_NIL, DST_NIL,NXT_JZ,6),
   // 3         (R0 - ACC) << 1  || JUMP_IF_C LSmall
   MI(O_NIL, SBUS_R0, ALU_SUBS, SHFT_SHL, DST_NIL,NXT_JC,5),
   // 4         R0 - ACC -> R0   || JUMP Lcheck
   MI(O_NIL, SBUS_R0, ALU_SUBS,SHFT_NIL, DST_R0, NXT_JMP,2),
   // 5 Lsmall: ACC - R0 -> ACC  || JUMP Lcheck
   MI(O_NIL, SBUS_R0, ALU_SUBA, SHFT_NIL,DST_ACC,NXT_JMP,2),
   // 6 Ldone:  R0 -> OUT        || JUMP Lstart
   MI(O_WR,  SBUS_R0, ALU_X,    SHFT_X,   DST_X, NXT_JMP,0)
  };

  reg csar        : ns(12);
  sig mir         : ns(32);
  sig ctl_nxt     : ns(4);
  sig csar_nxt    : ns(12);
  sig ctl_address : ns(12);

  always {

    mir = cstore(csar);
    ctl_ot      = mir[31];
    ctl_sbus    = mir[27:30];
    ctl_alu     = mir[23:26];
    ctl_shft    = mir[20:22];
    ctl_dest    = mir[16:19];
    ctl_nxt     = mir[12:15];
    ctl_address = mir[ 0:11];

    csar_nxt = csar + 1;
    csar = (ctl_nxt == NXT_NXT)  ? csar_nxt :
           (ctl_nxt == NXT_JMP)  ? ctl_address :
           (ctl_nxt == NXT_JC)   ? ((carry==1) ? ctl_address
                 : csar_nxt) :
           (ctl_nxt == NXT_JZ)   ? ((zero==1) ? ctl_address
                 : csar_nxt) :
           (ctl_nxt == NXT_JNC)  ? ((carry==0) ? ctl_address
                 : csar_nxt) :
           (ctl_nxt == NXT_JNZ)  ? ((zero==0) ? ctl_address
                 : csar_nxt) :
           csar;
  }
}

dp regfile (in  ctl_dest : ns(4);
            in  ctl_sbus : ns(4);
            in  data_in  : ns(WLEN);
            out data_out : ns(WLEN)) {
  reg r0 : ns(WLEN);
  reg r1 : ns(WLEN);
  reg r2 : ns(WLEN);
  reg r3 : ns(WLEN);
  reg r4 : ns(WLEN);
  reg r5 : ns(WLEN);
  reg r6 : ns(WLEN);
  reg r7 : ns(WLEN);
  always {
   r0 = (ctl_dest == DST_R0) ? data_in : r0;
   r1 = (ctl_dest == DST_R1) ? data_in : r1;
   r2 = (ctl_dest == DST_R2) ? data_in : r2;
   r3 = (ctl_dest == DST_R3) ? data_in : r3;
   r4 = (ctl_dest == DST_R4) ? data_in : r4;
   r5 = (ctl_dest == DST_R5) ? data_in : r5;
   r6 = (ctl_dest == DST_R6) ? data_in : r6;
   r7 = (ctl_dest == DST_R7) ? data_in : r7;
   data_out = (ctl_sbus == SBUS_R0) ? r0 :
              (ctl_sbus == SBUS_R1) ? r1 :
              (ctl_sbus == SBUS_R2) ? r2 :
              (ctl_sbus == SBUS_R3) ? r3 :
              (ctl_sbus == SBUS_R4) ? r4 :
              (ctl_sbus == SBUS_R5) ? r5 :
              (ctl_sbus == SBUS_R6) ? r6 :
              (ctl_sbus == SBUS_R7) ? r7 :
              r0;
 }
}

dp alu (in  ctl_dest : ns(4);
        in  ctl_alu  : ns(4);
        in  sbus     : ns(WLEN);
        in  shift    : ns(WLEN);
        out q        : ns(WLEN)) {
  reg acc : ns(WLEN);
  always {
    q = (ctl_alu == ALU_ACC)  ? acc :
        (ctl_alu == ALU_PASS) ? sbus :
        (ctl_alu == ALU_ADD)  ? acc + sbus :
        (ctl_alu == ALU_SUBA) ? acc - sbus :
        (ctl_alu == ALU_SUBS) ? sbus - acc :
        (ctl_alu == ALU_AND)  ? acc & sbus :
        (ctl_alu == ALU_OR)   ? acc | sbus :
        (ctl_alu == ALU_NOT)  ? ~ sbus     :
        (ctl_alu == ALU_INCS) ? sbus + 1   :
        (ctl_alu == ALU_INCA) ? acc + 1    :
        (ctl_alu == ALU_CLR)  ? 0          :
        (ctl_alu == ALU_SET)  ? 1          :
        0;
    acc = (ctl_dest == DST_ACC) ? shift : acc;
  }
}

dp shifter(in  ctl     : ns(3);
           out zero    : ns(1);
           out cy      : ns(1);
           in  shft_in : ns(WLEN);
           out so      : ns(WLEN)) {
  always {
    so = (ctl == SHFT_NIL) ? shft_in :
         (ctl == SHFT_SHL) ? (ns(WLEN)) (shft_in << 1) :
         (ctl == SHFT_SHR) ? (ns(WLEN)) (shft_in >> 1) :
         (ctl == SHFT_ROL) ? (ns(WLEN)) (shft_in #
               shft_in[WLEN-1]) :
         (ctl == SHFT_ROR) ? (ns(WLEN)) (shft_in[0] #
               (shft_in >> 1)):
         (ctl == SHFT_SLA) ? (ns(WLEN)) (shft_in << 1) :
         (ctl == SHFT_SRA) ? (ns(WLEN))
               (((tc(WLEN)) shft_in) >> 1) :
         0;
   zero = (shft_out == 0);
   cy   = (ctl == SHFT_NIL) ? 0 :
         (ctl == SHFT_SHL) ? shft_in[WLEN-1] :
         (ctl == SHFT_SHR) ? 0 :
         (ctl == SHFT_ROL) ? shft_in[WLEN-1] :
         (ctl == SHFT_ROR) ? shft_in[0] :
         (ctl == SHFT_SLA) ? shft_in[WLEN-1] :
         (ctl == SHFT_SRA) ? 0 :
         0;
  }
}

dp hmm(in  din   : ns(WLEN); out din_strb  : ns(1);
       out dout  : ns(WLEN); out dout_strb : ns(1)) {
  sig carry, zero : ns(1);
  sig ctl_ot      : ns(1);
  sig ctl_sbus    : ns(4);
  sig ctl_alu     : ns(4);
  sig ctl_shft    : ns(3);
  sig ctl_acc      : ns(1);
  sig ctl_dest     : ns(4);

  sig rf_out, rf_in : ns(WLEN);
  sig sbus          : ns(WLEN);
  sig alu_in        : ns(WLEN);
  sig alu_out       : ns(WLEN);
  sig shft_in       : ns(WLEN);
  sig shft_out      : ns(WLEN);
  use control(carry,    zero,
              ctl_ot, ctl_sbus, ctl_alu, ctl_shft, ctl_dest)
                    ;
  use regfile(ctl_dest, ctl_sbus, rf_in,    rf_out);
  use alu    (ctl_dest, ctl_alu,  sbus,   alu_in,  alu_out);
  use shifter(ctl_shft, zero, carry, shft_in, shft_out);

  always {
   sbus      = (ctl_sbus == SBUS_IN) ? din  : rf_out;
   din_strb  = (ctl_sbus == SBUS_IN) ? 1    : 0;
   dout      = sbus;
   dout_strb = (ctl_ot == O_WR) ? 1 : 0;
   rf_in     = shft_out;
   alu_in    = shft_out;
   shft_in   = alu_out;
  }
}

dp hmmtest {
  sig din       : ns(WLEN);
  sig din_strb  : ns(1);
  sig dout      : ns(WLEN);
  sig dout_strb : ns(1);
  use hmm(din, din_strb, dout, dout_strb);

  reg dcnt      : ns(5);
  lookup stim   : ns(WLEN) = {14,32,87, 12, 23, 99, 32, 22};

  always {
    dcnt = (din_strb) ? dcnt + 1 : dcnt;
    din  = stim(dcnt & 7);
    $display($cycle, " IO ", din_strb, " ", dout_strb, " ",
    $dec, din, " ", dout);
  }
}

system S {
 hmmtest;
}
```

This design can be simulated with the fdlsim GEZEL simulator. Because of the C macro's included in the source, the program first needs to be processed using the C preprocessor. The following command line illustrates how to simulate the first 100 cycles of this design.

```
>cpp -P hmm2.fdl | fdlsim 100
```

The first few lines of output look as follows.

```
0 IO 1 0 14 14
1 IO 1 0 32 32
2 IO 0 0 87 14
3 IO 0 0 87 14
4 IO 0 0 87 14
...
```

The micro-programmed machine reads the numbers 14 and 32 in clock cycles 0 and 1 respectively, and starts the GCD calculation. To find the corresponding GCD, we look for a '1' in the fourth column (output strobe). Around cycle 21, the first one appears. We can find that GCD(32,14) = 2. The testbench then proceeds with the next two inputs in cycles 23 and 24.

```
...
18 IO 0 0 87 2
19 IO 0 0 87 2
20 IO 0 0 87 2
21 IO 0 1 87 2
22 IO 1 0 87 87
23 IO 1 0 12 12
24 IO 0 0 23 87
...
```

A quick command to filter out the valid outputs during simulation is the following.

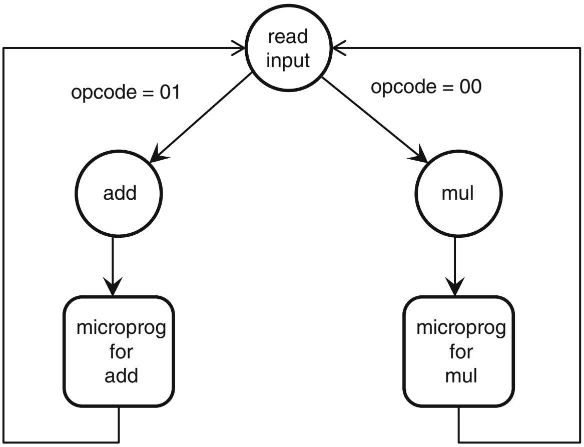
```
> cpp -P hmm2.fdl | fdlsim  200 | awk '{if ($4 == "1") print $0}'
21 IO 0 1 87 2
55 IO 0 1 23 3
92 IO 0 1 32 1
117 IO 0 1 14 2
139 IO 0 1 87 2
173 IO 0 1 23 3
```

This design demonstrates how the FSMD model can be applied to create a more complex, micro-programmed machine. In the following, we show how this can be used to create programming concepts at even higher levels of abstraction, using micro-program interpreters.

6.6 Micro-program Interpreters

A micro-program is a highly-optimized sequence of commands for a datapath. The sequence of register transfers is optimized for parallelism. Writing efficient micro-programs is not easy, and requires in-depth understanding of the machine architecture. An obvious question is if a programming language, such as a pseudo-assembly language, would be of help in developing micro-programs. Certainly, the writing process itself could be made more convenient.

Fig. 6.9 A micro-program interpreter implements a more abstract language



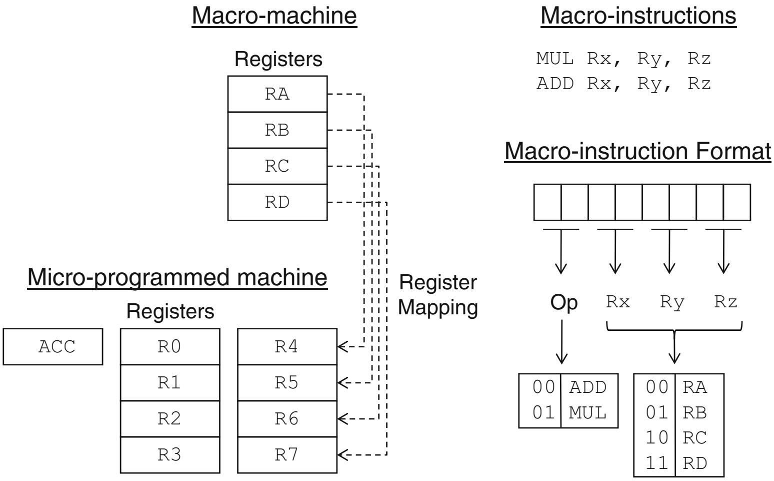


Fig. 6.10 Programmer's model for the macro-machine example

A common use for micro-programs is therefore not to encode complete applications, but instead to work as *interpreters* for other programs, developed at a high abstraction level. An interpreter is a machine that decodes and executes instruction sequences of an abstract high-level machine, which we will call the macro-machine. The instructions from the macro-machine will be implemented in terms of micro-programs for a micro-programmed machine. Such a construct is illustrated in Fig. 6.9, and is called a *micro-program interpreter*. We create a micro-program in the form of an infinite loop, which reads a macro-instruction byte and breaks down a byte in opcode and operand fields. It then takes specific actions depending on the values of the opcode.

We discuss, by means of an example, how to implement such a macro-machine. Figure 6.10 shows the programmers' model of the macro-machine. It is a very simple machine, with four registers RA through RD, and two instructions for adding and multiplying those registers. The macro-machine will have the same wordlength as the micro-programmed machine, but it has fewer register than the original micro-programmed machine. To implement the macro-machine, we will map the macro-register set directly onto the micro-register set. In this case, we will map register RA to RD onto register R4 to R7 respectively. This leaves register R0 to R3, as well as the accumulator, available to implement macro-instructions. The macro-machine has two instructions: add and mul. Each of these instructions takes two source operands and generates one destination operand. The operands are macro-machine registers. Because the micro-machine has to decode the macro-instructions, we also need to choose the instruction-encoding of the macro-instructions. This is illustrated on the right of Fig. 6.10. Each macro-instruction is a single byte, with 2 bits for the macro-opcode, and 2 bits for each of the macro-instruction operands.

Listing 6.3 shows a sample implementation for each of the ADD and MUL instructions. We have assumed that single-level subroutines are supported at the level of the micro-machine. See Problem 6.3 how such a subroutine call can be implemented in the micro-programmed machine.

The micro-interpreter loop, on line 21–29, reads one macro-instruction from the input, and determines the macro-instruction opcode with a couple of shift instructions. Depending on the value of the opcode field, the micro-program will then jump to a routine to implement the appropriate macro-instruction, add or mul.

The implementation of ADD is shown in lines 35–40. The micro-instructions use fixed source operands and a fixed destination operand. Since the macro-instructions can use one of four possible operand registers, an additional register-move operation is needed to prepare the micro-instruction operands. This is done by the putarg and getarg subroutines, starting on line 62. The getarg subroutine copies data from the macro-machine source registers (RA through RD) to the micro-machine source working registers (R1 and R2). The putarg subroutine moves data from the micro-machine destination working register (R1) back to the destination macro-machine register (RA through RD).

The implementation of the add instruction starts on line 35. At the start of this section of code, the accumulator contains the macro-instruction. The accumulator value is passed on to the getarg routine, which decodes the two source operand registers can copies them into micro-machine register R1 and R2. Next, the add macro-instruction performs the addition, and stores the result in R1 (line 36–39). The putarg and getarg routines assume that the opcode of the macro-instruction is stored in the accumulator. Since the body of the add instruction changes the accumulator, it needs to be preserved before putarg is called. This is the purpose of the register-copy instructions on lines 36 and 39.

The implementation of the mul macro-instruction starts on line 46, and follows the same principles as the add instruction. In this case, the body of the instruction is more complex since the multiply operation needs to be performed using an add-and-shift loop. A loop counter is created in register R3 to perform eight iterations of add-and-shift. Because the accumulator register is only 8 bit, the multiply instruction cannot capture all 16 output bits of an 8-by-8 bit multiply. The implementation of mul preserves only the least significant byte.

Listing 6.3 Implementation of the macro-instructions ADD and MUL

```
//-------------------------------------------------
// Macro-machine for the instructions
//
//    ADD Rx, Ry, Rz
//    MUL Rx, Ry, Rz
//
//    Macro-instruction encoding:
//    +----+----+----+----+
//    | ii + Rx + Ry + Rz +
//    +----+----+----+----+
//
//    where ii = 00 for ADD
//               01 for MUL
//    where Rx, Ry and Rz are encoded as follows:
//               00 for RA (mapped to R4)
//               01 for RB (mapped to R5)
//               10 for RC (mapped to R6)
//               11 for RD (mapped to R7)
//
// Interpreter loop reads instructions from input
macro:   IN -> ACC
         (ACC & 0xC0) >> 1 -> R0
         R0 >> 1 -> R0
         R0 >> 1 -> R0
         R0 >> 1 -> R0
         R0 >> 1 -> R0
         R0 >> 1 -> R0            || JUMP_IF_NZ mul
         (no_op)                  || JUMP add
macro_done: (no_op)               || JUMP macro

//-------------------------------------------------
//
// Rx = Ry + Rz
//
add:      (no_op)                 || CALL getarg
          ACC -> R0
          R2  -> ACC
          (R1 + ACC) -> R1
          R0 -> ACC               || CALL putarg
          (no_op)                 || JUMP macro_done

//-------------------------------------------------
//
// Rx = Ry * Rz
//
mul:       (no_op)                || CALL getarg
           ACC -> R0
           0   -> ACC
           8   -> R3
loopmul:   (R1 << 1) -> R1        || JUMP_IF_NC nopartial
           (ACC << 1) -> ACC
           (R2 + ACC) -> ACC
nopartial: (R3 - 1) -> R3         || JUMP_IF_NZ loopmul
           ACC -> R1
           R0 -> ACC              || CALL putarg
           (no_op)                || JUMP macro_done

//-------------------------------------------------
//
// GETARG
//
getarg:   (ACC & 0x03) -> R0      || JUMP_IF_Z Rz_is_R4
          (R0 - 0x1)              || JUMP_IF_Z Rz_is_R5
          (R0 - 0x2)              || JUMP_IF_Z Rz_is_R6
Rz_is_R7:  R7 -> R1               || JUMP get_Ry
Rz_is_R6:  R6 -> R1               || JUMP get_Ry
Rz_is_R5:  R5 -> R1               || JUMP get_Ry
Rz_is_R4:  R4 -> R1               || JUMP get_Ry
get_Ry:   (ACC & 0x0C) >> 1 -> R0
           R0 >> 1 -> R0          || JUMP_IF_Z Ry_is_R4
          (R0 - 0x1)              || JUMP_IF_Z Ry_is_R5
          (R0 - 0x2)              || JUMP_IF_Z Ry_is_R6
Ry_is_R7:  R7 -> R2               || RETURN
Ry_is_R6:  R6 -> R2               || RETURN
Ry_is_R5:  R5 -> R2               || RETURN
Ry_is_R4:  R4 -> R2               || RETURN

//-------------------------------------------------
//
// PUTARG
//
putarg:   (ACC & 0x30) >> 1  -> R0
           R0 >> 1 -> R0
           R0 >> 1 -> R0
           R0 >> 1 -> R0          || JUMP_IF_Z Rx_is_R4
          (R0 - 0x1)              || JUMP_IF_Z Rx_is_R5
          (R0 - 0x2)              || JUMP_IF_Z Rx_is_R6
Rx_is_R7:  R1 -> R7               || RETURN
Rx_is_R6:  R1 -> R6               || RETURN
Rx_is_R5:  R1 -> R5               || RETURN
Rx_is_R4:  R1 -> R4               || RETURN
```

A micro-programmed interpreter creates the illusion of a machine that has more powerful instructions than the original micro-programmed architecture. The trade-off made with such an interpreter is ease-of-programming versus performance: each instruction of the macro-machine may need many micro-machine instructions. The concept of microprogram interpreters has been used extensively to design processors with configurable instruction sets, and was originally used to enhance the flexibility of expensive hardware. Today, the technique of micro-program interpreter design is still very useful to create an additional level of abstraction on top of a micro-programmed architecture.

6.7 Micro-program Pipelining

As can be observed from Fig. 6.11, the micro-program controller may be part of a long chain of combinational logic. Pipeline registers can be used to break these long chains. However, the introduction of pipeline registers has a large impact on the design of micro-programs. This section will study these effects in more detail.

First, observe in Fig. 6.11 that the CSAR register is part of possibly three loops with logic. The first loop runs through the next-address logic. The second loop runs through the control store and the next-address logic. The third loop runs through the control store, the data path, and the next-address logic. These combinational paths may limit the maximum clock frequency of the micro-programmed machine. There are three common places where additional pipeline registers may be inserted in the design of this machine, and they are marked with shaded boxes in Fig. 6.11.

- A common location to insert a pipeline register is at the output of the control store. A register at that location is called the micro-instruction register, and it enables overlap of the datapath evaluation, the next address evaluation, and the micro-instruction fetch.

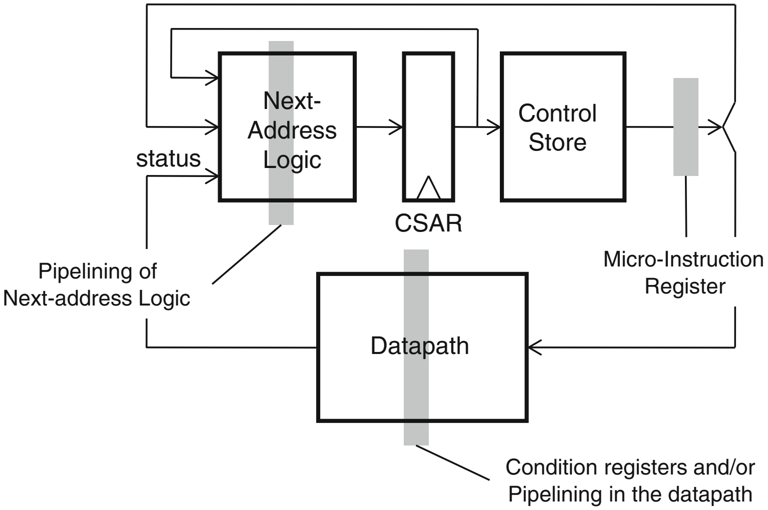

Next-Address Logic
status
CSAR
Control Store
Micro-Instruction Register
Pipelining of Next-address Logic
Datapath
Condition registers and/or Pipelining in the datapath

Fig. 6.11 Typical placement of pipeline registers in a micro-program interpreter

Table 6.2 Effect of the micro-instruction register on jump instructions

Cycle	CSAR	Micro-instruction register
N	4	
N+1	5	CSTORE(4) = JUMP 10
N+2	10	CSTORE(5) *need to cancel*
N+3	11	CSTORE(10) *execute*

- Another location for pipeline registers is the datapath. Besides pipeline register inside of the data path, additional condition-code registers can be placed at the datapath outputs.
- Finally, the next-address logic may be pipelined as well, in case high-speed operation is required and the target CSAR address cannot be evaluated within a single clock cycle.

Each of these register cuts through a different update-loop of the `CSAR` register, and therefore each of them has a different effect on the micro-program.

6.7.1 Micro-instruction Register

Let's first consider the effect of adding the micro-instruction register. Due to this register, the micro-instruction fetch (i.e. addressing the CSTORE and retrieving the next micro-instruction) is offset by one cycle from the evaluation of that micro-instruction. For example, when the CSAR is fetching instruction i from a sequence of instructions, the datapath and next-address logic will be executing instruction $i-1$.

Table 6.2 illustrates the effect of this offset on the instruction stream, when that stream contains a jump instruction. The micro-programmer entered a `JUMP 10` instruction in CSTORE location 4, and that instruction will be fetched in clock cycle `N`. In clock cycle `N+1`, the micro-instruction will appear at the output of the micro-instruction register. The execution of that instruction will complete in cycle `N+2`. For a JUMP, this means that the value of CSAR will be affected in cycle `N+2`. As a result, a JUMP instruction cannot modify the value of CSAR within a single clock cycle. If CSTORE(4) contains a JUMP, then the instruction located in CSTORE(5) will be fetched as well. The micro-programmer needs to be aware of this. The possible strategies are (a) take into account that a JUMP will be executed with one cycle of delay (so-called 'delayed branch'), or (b) include support in the micro-programmed machine to cancel the execution of an instruction in case of a jump.

Table 6.3 Effect of the micro-instruction register and condition-code register on conditional jump instructions

Cycle	CSAR	Micro-instruction register
N	3	
N+1	4	CSTORE(3) = TEST R0 sets Z-flag
N+2	5	CSTORE(4) = JZ 10
N+3	10	CSTORE(5) *need to cancel*
N+4	11	CSTORE(10) *execute*

Table 6.4 Effect of additional pipeline registers in the CSAR update loop

Cycle	CSAR_pipe	CSAR	Micro-instruction register
0	0	0	CSTORE(0)
1	1	0	CSTORE(0) twice?
2	1	1	CSTORE(1)
3	2	1	CSTORE(1) twice?

6.7.2 Datapath Condition-Code Register

As a second case, assume that we have a condition-code register in the data-path, in addition to a micro-instruction register. The net effect of a condition code register is that a condition value will only be available one clock cycle after the corresponding datapath operation. As a result, a conditional-jump instruction can only operate on datapath conditions from the previous clock cycle. Table 6.3 illustrates this effect. The branch instruction in CSTORE(4) is a conditional jump. When the condition is true, the jump will be executed with one clock cycle delay, as was discussed before. However, the `JZ` is evaluated in cycle N+2 based on a condition code generated in cycle N+1. Thus, the micro-programmer needs to be aware that conditions need to be available one clock cycle before they will be used in conditional jumps.

6.7.3 Pipelined Next-Address Logic

Finally, assume that there is a third level of pipelining available inside of the next-address update loop. For simplicity, we will assume there are two CSAR registers back-to-back in the next-address loop. The output of the next-address-logic is fed into a register CSAR_pipe, and the output of CSAR_pipe is connected to CSAR. Table 6.4 shows the operation of this micro-programmed machine, assuming all registers are initially zero. As shown in the table, the two CSAR registers in the next-address loop result in two (independent) address sequences. When all registers start out at 0, then each instruction of the micro-program will be executed twice. Solving this problem is not easy. While one can do a careful initialization of CSAR_pipe and CSAR such that they start out at different values (e.g. 1 and 0), this re-initialization will need to be done at each jump instruction. This makes the design and the programming of pipelined next-address logic very hard.

These examples show that a micro-programmer must be aware of the implementation details of the micro-architecture, and in particular of all the delay effects present in the controller. This can significantly increase the complexity of the development of micro-programs.

6.8 Microprogramming with Microcontrollers

Although microprogramming originated many years ago, its ideas are still very useful. When complex systems are created in hardware, the design of an adequate control architecture often becomes a challenge. In this section, we illustrate a possible solution based on the use of a micro-controller.

6.8.1 System Architecture

A micro-controller has minimal computational capabilities, such as an ALU with basic logical and arithmetic operations. However, they are pervasively used for all sorts of control applications. In this section, we will discuss using them as micro-program controllers.

Figure 6.12 shows an example of a micro-programmed architecture that uses a microcontroller. We assume a device which has 8-bit digital I/O ports. The microcontroller has dedicated instructions to read from, and write to, such ports.

In the design of Fig. 6.12, the I/O ports are used to control a microcoded datapath. There are three ports involved.

- A digital input port is used to carry data or status from the microcoded datapath to the microcontroller.
- A first digital output port is used to carry data from the microcontroller to the microcoded datapath.
- A second digital output port is used to carry control or micro-instructions from the microcontroller to the microcoded datapath.

The machine works as follows. For each micro-instruction, the microcontroller will combine a micro-instruction with an optional argument, and send that to the microcoded datapath. The microcoded datapath will then return any result to the microcontroller. Care needs to be taken to keep the microcoded datapath and the microcontroller synchronized. If the execution of a microcoded instruction takes multiple cycles, the microcontroller will need to delay reading data output and/or status over an appropriate amount of cycles. The next subsection demonstrates an example of this technique.

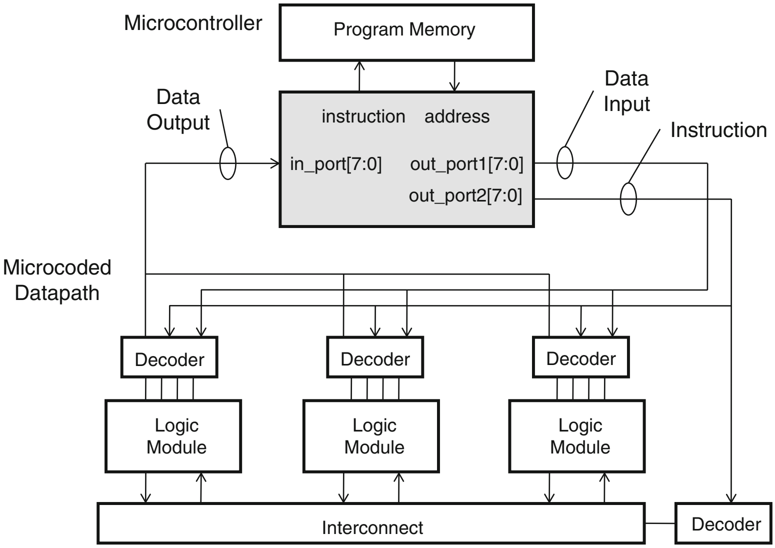


Fig. 6.12 Using a microcontroller as a microprogram controller

6.8.2 *Example: Bresenham Line Drawing*

We will discuss an example microcoded machine, designed using an 8051 microcontroller. The application is that of Line Drawing on a grid. Figure 6.13 illustrates the concept. A continuous line is defined between `(0,0)` and `(tx,ty)`. The line is drawn using a discrete grid of pixels, and the problem to solve is to decide what pixels should be turned on. We assume that the grid has unit spacing, so that `tx` and `ty` are positive integers.

An algorithm to solve this problem was proposed by Bresenham. His solution starts with the following observation. If a line segment lies in the first octant (or, `tx > ty > 0`), then the line can be drawn, pixel per pixel, by taking only horizontal and diagonal steps. Thus, if the pixel `(x1, y1)` is turned on, then the next pixel to turn on will be either `(x1+1,y1)` or else `(x1+1,y1+1)`. The pixel that should be selected is the one that lies ‘closest’ to the true line. Bresenham’s insight was to show that the closest pixel can be obtained using integer arithmetic only.

Indeed, assume that pixel `(x1, y1)` is already turned on, and that its center has a distance `e` from the true line. If the center of pixel `(x1,y1)` is above the true line, then `e` is negative. If the center of pixel `(x1, y1)` is below the true line, then `e` is positive. The line drawing algorithm can be controlled based on the sign of `e`. As long as `e` is positive, we need to take diagonal steps. When `e` becomes negative, we should take a horizontal step. In this manner, the error `e` is minimized over the entire line.

To compute `e` at every pixel, we proceed as follows. When taking a horizontal step, the true line will move up following the slope `ty/tx`. The error will increase

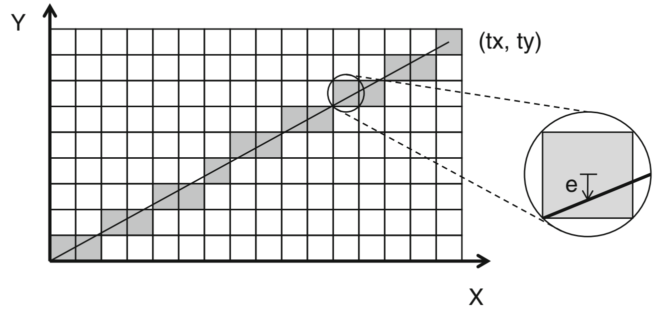


Fig. 6.13 Bresenham line drawing algorithm

by `es = ty/tx`. If we would take a diagonal step, then the error will increase by `ed = ty/tx - 1`. The factor 1 represents the step in the y direction, assuming a unit-spaced grid. Both quantities, `es` and `ed`, can be scaled to integer values by multiplying them with `tx`, which is a constant over the entire line.

Listing 6.4 shows a C function to draw pixels using the Bresenham algorithm. This function can draw lines in the first quadrant, and hence distinguishes between the cases `ty>tx` and `tx>ty`. The algorithm is written using `unsigned char`, and is limited in precision to byte-size coordinates. The sign test for the error accumulator is written as a test on the most significant bit of `e`, i.e. `(e & 0x80)`. The algorithm can be compiled and executed on an 8-bit microcontroller.

Listing 6.4 C program for line drawing in the first quadrant

```
void bresen(unsigned tx, unsigned ty) {
  unsigned char  x, y;   // pixel coordinates
  unsigned char  e;      // error accumulator
  unsigned char  es, ed; // error inc for straight/diag
  unsigned char  xs, xd; // x inc for straight/diag
  unsigned char  ys, yd; // y inc for straight/diag
  unsigned i;

  x  = 0; y = 0;
  e  = 0;
  ed = (tx > ty) ? (ty - tx) : (tx - ty);
  es = (tx > ty) ? ty        : tx;
  xd = 1;
  xs = (tx > ty) ? 1 : 0;
  yd = 1;
  ys = (tx > ty) ? 0 : 1;

  for (i=0; i<64; i++) {  // plot 64 pixels
    // plot(x, y);
    x = (e & 0x80) ? x + xs : x + xd;
    y = (e & 0x80) ? y + ys : y + yd;
    e = (e & 0x80) ? e + es : e + ed;
  }
}
```

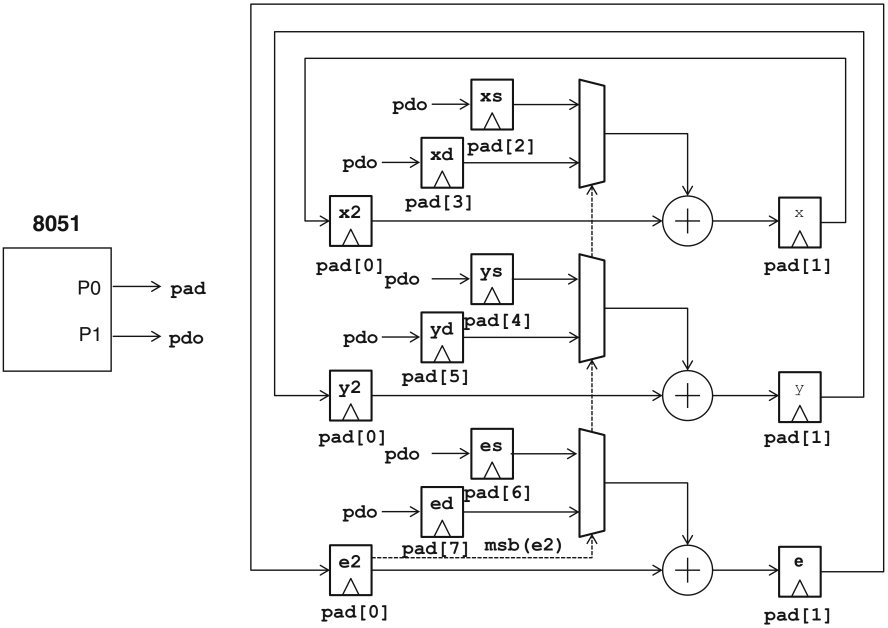


Fig. 6.14 Bresenham line drawing microcoded datapath

Next, we design a microcoded datapath for the Bresenham line drawing algorithm. The most obvious candidate for such a datapath is the most intensively executed part of the code, the loop body of Listing 6.4 (lines 20–22). Figure 6.14 shows a hardware implementation for the microcoded datapath. Three adders work in parallel to update the x and y registers holding the current pixel coordinate, as well as the e register, holding the error accumulator. The values to add are stored in additional registers within the microcoded datapath: xs, xd, ys, yd, es, and ed. The microcoded datapath is fully controlled from the 8051 microcontroller through two 8-bit ports. These ports define the value of pad and pdo.

Listing 6.5 shows the GEZEL version of the microcoded datapath in Fig. 6.14. The listing also shows the inclusion of the 8051 microcontroller, and two ports. The C driver code for the microcoded datapath is shown in Listing 6.6.

The microinstructions generated from the 8051 will look as combinations of pad and pdo. For example, let's see how to program the step registers in the datapath. There are four step registers (xd, xs, yd, ys). Their update is tied to pad[3], pad[2], pad[5], pad[4] respectively. To program xd, the 8051 needs to write the desired value to pdo, then toggle bit pad[3]. Thus, in the GEZEL description (Listing 6.5) on line 40, we find:

```
xd = pad[3] ? pdo : xd;
```

The corresponding programming command is shown on Line 27 of the C driver (Listing 6.6):

```
P1 = xd; P0 = 0x08; P0 = 0x0;
```

As a second example, let's see how the 8051 will implement loop iterations. In Fig. 6.14, each of the loops that contains register x, y and e, also contains a second register x2, y2 and e2. The control bits in pad can now be steered to update either the set x, y and e, or else the set x2, y2 and e2. The update of x, y and e is tied to pad[1], while the update of x2, y2 and e2 is tied to pad[0]. Hence, in the GEZEL description (Listing 6.5) around line 30, we find:

```
xi = e2[7] ? xs : xd;
x2 = pad[0] ? x : x2;
x  = pad[1] ? x2 + xi : x;
```

To implement one iteration of the loop, the 8051 controller first toggles pad[1], and next toggles pad[2]. This can be done simultaneously, so that the 8051 controller writes the values 0x1 and 0x2 to pad. The corresponding loop iteration command is shown on Lines 30 and 31 of the C driver (Listing 6.6):

```
P0 = 0x1;
P0 = 0x2;
```

Listing 6.5 GEZEL Microcoded Datapath

```
 1  ipblock my8051 {
 2    iptype "i8051system";
 3    ipparm "exec=bresen.ihx";
 4  }
 5
 6  ipblock mi_8051(out data : ns(8)) {
 7    iptype "i8051systemsource";
 8    ipparm "core=my8051";
 9    ipparm "port=P0";
10  }
11
12  ipblock dout_8051(out data : ns(8)) {
13    iptype "i8051systemsource";
14    ipparm "core=my8051";
15    ipparm "port=P1";
16  }
17
18  dp microdp(in  pad  : ns(8);
19             in  pdo  : ns(8)) {
20    reg xd, yd : ns(1);
21    reg xs, ys : ns(1);
22    reg x,  y  : ns(8);
23    reg ed, es : ns(8);
24    reg e      : ns(8);
25    reg x2, y2 : ns(8);
26    reg e2     : ns(8);
```

```
  sig xi, yi : ns(8);
  sig ei     : ns(8);
  always {
    xi  = e2[7]  ? xs      : xd;
    yi  = e2[7]  ? ys      : yd;
    ei  = e2[7]  ? es      : ed;
    x2  = pad[0] ? x       : x2;
    y2  = pad[0] ? y       : y2;
    e2  = pad[0] ? e       : e2;
    x   = pad[1] ? x2 + xi :  x;
    y   = pad[1] ? y2 + yi :  y;
    e   = pad[1] ? e2 + ei : e;
    xs  = pad[2] ? pdo     : xs;
    xd  = pad[3] ? pdo     : xd;
    ys  = pad[4] ? pdo     : ys;
    yd  = pad[5] ? pdo     : yd;
    es  = pad[6] ? pdo     : es;
    ed  = pad[7] ? pdo     : ed;
  }
}

dp top {
  sig pad, pdo : ns(8);
  use my8051;
  use mi_8051(pad);
  use dout_8051(pdo);
  use microdp(pad, pdo);
}

system S {
  top;
}
```

Listing 6.6 8051 Driver Code for GEZEL Microcoded Datapath

```
#include <8051.h>

void bresen_hw(unsigned tx, unsigned ty) {
  unsigned char  x, y;   // pixel coordinates
  unsigned char  e;      // error accumulator
  unsigned char  es, ed; // error inc for straight/diag
  unsigned char  xs, xd; // x inc for straight/diag
  unsigned char  ys, yd; // y inc for straight/diag
  unsigned i;

  x  = 0;
  y  = 0;
  e  = 0;
  ed = (tx > ty) ? (ty - tx) : (tx - ty);
  es = (tx > ty) ? ty        : tx;
  xd = 1;
  xs = (tx > ty) ? 1 : 0;
  yd = 1;
  ys = (tx > ty) ? 0 : 1;
  // P0 - MI   [ed es yd ys xd xs xye x2y2e2]
  //             7  6  5  4  3  2   1      0
  // P1 - DOUT
  P1 = ed;  P0 = 0x80;  P0 = 0x0;
  P1 = es;  P0 = 0x40;  P0 = 0x0;
  P1 = yd;  P0 = 0x20;  P0 = 0x0;
  P1 = ys;  P0 = 0x10;  P0 = 0x0;
  P1 = xd;  P0 =  0x8;  P0 = 0x0;
  P1 = xs;  P0 =  0x4;  P0 = 0x0;
  for (i=0; i<64; i++) {
    P0 = 0x1;
    P0 = 0x2;
  }
}
```

This concludes the example of Bresenham line drawing using a microcoded datapath and a microcontroller. This technique is very useful if one is coping with the design of control functions in hardware design.

6.9 Summary

In this section we introduced Microprogramming as means to deal with control design problems in hardware. Finite State Machines are good for small, compact specifications, but they result in a few issues. Finite State Machines cannot easily express hierarchy (FSMs calling other FSMs). Therefore control problems can easily blow up when specified as a finite-state-machine, yielding so-called ‘state explosion’.

In a micro-programmed architecture, the hardcoded next-state logic of a finite state machine is replaced with a programmable control store and a program control (called CSAR or Control Store Address Register). This takes care of most problems: micro-programs can call other micro-programs using jump instructions or using the equivalent of subroutines. Micro-programs have a much higher scalability than finite state machines.

Writing micro-programs is more difficult than writing assembly code or C code. Therefore, instead of mapping a full application directly into micro-code, it may be easier to develop a micro-programmed interpreter. Such an interpreter implements an instruction-set for a language at a higher level of abstraction. Still, even single micro-instructions may be hard to write, and in particular the programmer has to be aware of all pipelining and delay effects inside of the micro-programmed architecture.

6.10 Further Reading

The limitations of FSM as a mechanism for control modeling have long been recognized in literature. While it has not been the topic of the chapter, there are several alternate control modeling mechanisms available. A key contribution to hierarchical modeling of FSM was defined by Harel in StateCharts (Harel 1987). Additionally, the development of so-called synchronous languages have support the specification of control as event-driven programs. See for example Esterel by Berry (2000) and Potop-Butucaru et al. (2007).

A nice introduction and historical review of Microprogramming can be found online on the pages of Smotherman (2009). Most of the work on micro-programming was done in the late 1980s and early 1990s. Conference proceedings and computer-architecture books from that period are an excellent source of design ideas. For example, a extensive description of micro-programming is found in the textbook by Lynch (1993). Control optimization issues of micro-programming are discussed by Davio et al. (1983).

6.11 Problems

Problem 6.1. Figure 6.15 shows a micro-programmed datapath. There are six control bits for the datapath: 2 bits for each of the multiplexers M1 and M2, and 2 bits for the ALU. The encoding of the control bits is indicated in the figure.

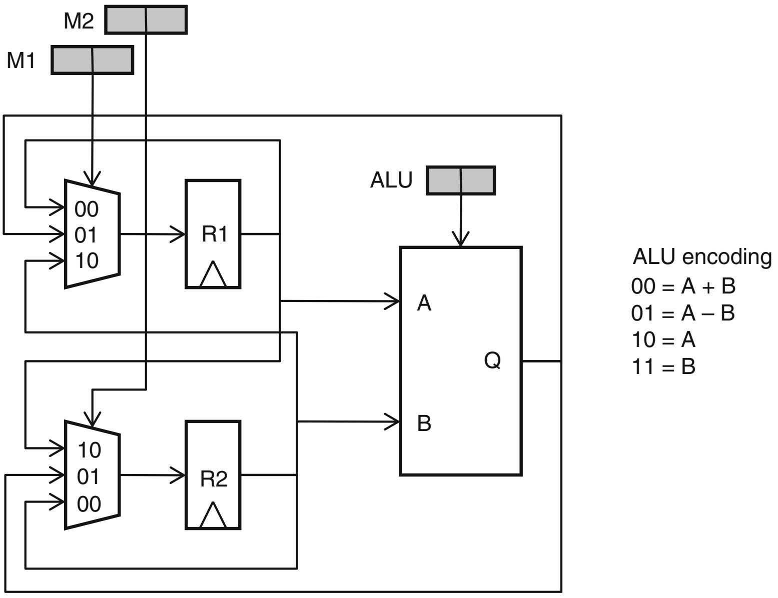


Fig. 6.15 Datapath for Problem 6.1

Table 6.5 Micro-instructions for Problem 6.1

`SWAP`	Interchange the content of R1 and R2.
`ADD Rx`	Add the contents of R1 and R2 and store the contents in `Rx`, which is equal to R1 or R2. There are two variants of this instruction depending on Rx.
`COPY Rx`	Copy the contents of Ry into Rx. (Rx, Ry) is either (R1, R2) or (R2, R1). There are two variants of this instruction depending on Rx.
`NOP`	Do nothing.

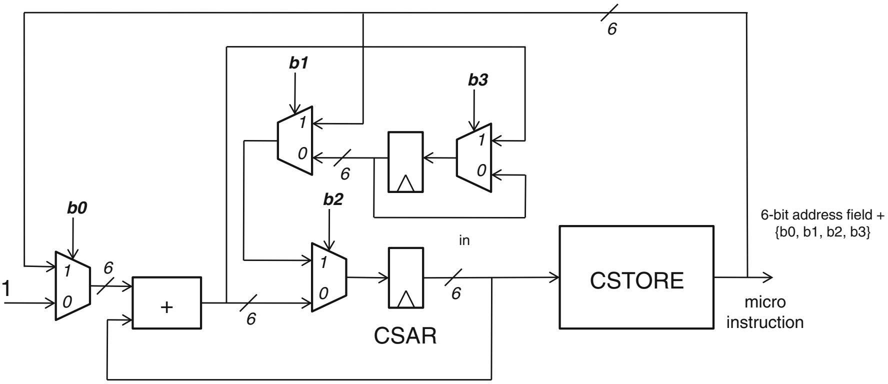


Fig. 6.16 Datapath for Problem 6.3

(a) Develop a horizontal micro-instruction encoding for the list of micro-instructions shown in Table 6.5.

(b) Develop a vertical micro-instruction encoding for the same list of instructions. Use a reasonable encoding that results in a compact and efficient decoder for the datapath.

Problem 6.2. Using the micro-programmed machine discussed in Sect. 6.5, create a program that reads in a number from the input and that counts the number of non-zero bits in that number. The resulting bitcount must be stored in register R7.

Problem 6.3. Figure 6.16 shows the implementation of a next-address decoder. A total of 10 bits from the micro-instruction are used to control the next-address logic: a 6-bit address field, and four control bits, `b0`, `b1`, `b2`, and `b3`.

For each of the combinations of control bits shown in Table 6.6, find a good description of the instruction corresponding to the control bit values shown. Don't write generic descriptions (like 'CSAR register is incremented by one'), but give a high-level description of the instruction they implement. Use terms that a software programmer can understand

Problem 6.4. Design a next-address instruction decoder based on the set of micro-instructions shown in Table 6.7. Design your implementation in GEZEL or Verilog. An example IO definition is shown next. The CSAR has to be 10 bit wide, the

Table 6.6 Next-address instructions for Problem 6.3

Combination	b0	b1	b2	b3
Instruction 1	1	X	0	0
Instruction 2	X	1	1	0
Instruction 3	0	1	1	1
Instruction 4	X	0	1	0

Table 6.7 Next-address instructions for Problem 6.4

`NXT`	`CSAR = CSAR + 1;`
`JUMP k`	`CSAR = k;`
`GOSUB k`	`RET = CSAR + 1; CSAR = k;`
`RETURN`	`CSAR = RET;`
`SWITCH k`	`RET = CSAR + 1; CSAR = RET;`

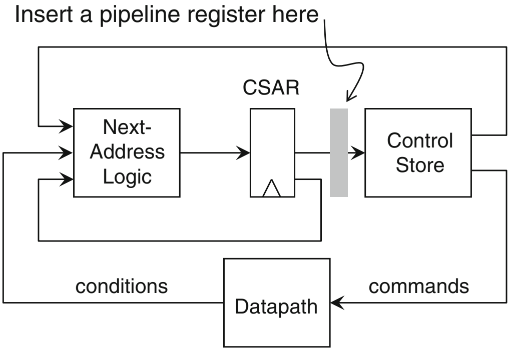


Fig. 6.17 Datapath for Problem 6.5

width of the Address field and the width of the next-address field must be chosen accordingly.

```
dp nextaddress_decoder(in csar      : ns(10);
                       out address : ns(x);
                       out next    : ns(y)) {
  // ...
}
```

Problem 6.5. Your colleague asks you to evaluate an enhancement for a microprogrammed architecture, as illustrated in Fig. 6.17. The enhancement is to insert a pipeline register just in from of the control store.

(a) Does this additional register reduce the critical path of the overall architecture?
(b) Your colleague calls this a dual-thread architecture, and claims this enhancement allows the micro-control engine to run two completely independent programs in an interleaved fashion. Do you agree with this or not?

Chapter 7
General-Purpose Embedded Cores

7.1 Processors

The most successful programmable component over the past decades is, without doubt, the microprocessor. The microprocessor is truly ubiquitous. Any modern electronic device more complicated than a pushbutton seems fitted with a microprocessor; a modern car contains 50–100 embedded microprocessors. There are several reasons for the universal success of the microprocessor.

- Microprocessors, or the stored-program concept in general, separate software from hardware through the definition of an instruction-set. No other hardware development technique has ever been able to uncouple hardware and software in a similar way. Think about microprogramming. Microprograms are really shorthand notations for the control specification of a specialized datapath. The notion of a micro-program as an architecture-independent concept makes no sense: micro-instructions are architecture-specific.
- Microprocessors come with tools (compilers and assemblers), that help a designer to create applications. The availability of a compiler to translate a programming language into a program for a microprocessor is an enormous advantage. An embedded software designer can write applications in a high-level programming language, independently from the specific microprocessor. Compilers help embedded software designers to quickly migrate from one processor family to the next one.
- No other device has been able to cope as efficiently with reuse as microprocessors did. *Reuse*, in general, is the ability to save design effort over multiple applications. A general-purpose embedded core is an excellent example of reuse in itself. However, microprocessors have also a large impact on reuse in electronic system design. Microprocessors come with bus interfaces that support the physical integration of an electronic system consisting of multiple components. Their compilers have enabled the development of standard software libraries as well as the logical integration of a system.

P.R. Schaumont, *A Practical Introduction to Hardware/Software Codesign*,
DOI 10.1007/978-1-4614-3737-6_7, © Springer Science+Business Media New York 2013

- Fourth, no other programmable component has the same scalability as a microprocessor. The concept of the stored-program computer has been implemented across a wide range of word-lengths (4-bit …64-bit). Microprocessors have also gained significant traction as central unit in complex integrated circuits. In this approach, called System-on-Chip (SoC), a microprocessor controls the collaboration of one or more complex peripherals. We will discuss SoC in the next chapter.

In summary, the combination of instruction-set, tools, reuse, and scalability have turned the microprocessor into a dominant component in electronic systems. In fact, very often hardware/software codesign starts with a program on a general-purpose microprocessor, and specializes that design by adding dedicated hardware to the microprocessor.

7.1.1 The Toolchain of a Typical Micro-processor

Figure 7.1 illustrates the typical design flow to convert software source code into instructions for a processor. The Figure introduces the following terminology used in this Chapter. A compiler or an assembler is used to convert source code into object code. An object code file contains opcodes (instructions) and constants, along with supporting information to organize these instructions and constants in memory. A linker is used to combine several object code files into a single, standalone executable file. A linker will also resolve all unknown elements in the object code, such as the address of external variables or the entry point of library routines. Finally, a loader program determines how the information in an executable file is organized in memory. Typically, there will be a part of the memory space reserved for instructions, another part for constant data, another part for global data with read/write access, and so on. A very simple microprocessor system requires at least two elements: a processor, and a memory holding instructions for the processor. The memory is initialized with processor instructions by the loader. The processor will fetch these instructions from memory and execute them on the processor datapath.

7.1.2 From C to Assembly Instructions

Listing 7.1 A C program to find the maximum of greatest common divisors

```c
int gcd(int a[5], int b[5]) {
  int i, m, n, max;
  max = 0;
  for (i=0; i<5; i++) {
    m = a[i];
    n = b[i];
```

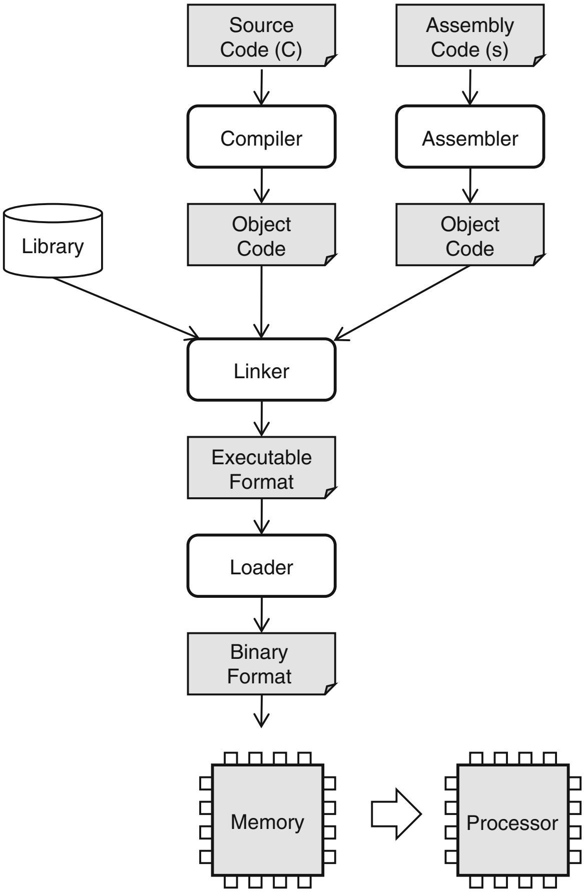


Fig. 7.1 Standard design flow of software source code to processor instruction

```
    while (m != n) {
      if (m > n)
        m = m - n;
      else
        n = n - m;
    }
    if (max > m)
      max = m;
  }
  return max;
}

int a[] = {26, 3,33,56,11};
int b[] = {87,12,23,45,17};

int main() {
  return gcd(a, b);
}
```

The steps that turn C code into machine instructions and, eventually, opcodes in memory can be demonstrated with an example. Listing 7.1 shows a C program that evaluates the largest among the common divisors of five pairs of numbers. The program illustrates a few interesting features, such as function calls, arrays, and global variables. We will inspect the C program at two levels of abstraction. First, at the level of assembly code generated by the compiler, and next, at the level of the machine code stored in the executable generated by the linker.

Before creating object code, we need to pick a microprocessor target. In this book, we will use several different targets, but for this example, we choose the ARM microprocessor. If we are working on a personal computer, we need a *cross-compiler* to generate the executable for an ARM microprocessor. A cross-compiler generates an executable for a processor different from the processor used to run the compiler.

We will make use of the GNU compiler toolchain. The command to generate the ARM assembly program is as follows.

```
> /usr/local/arm/bin/arm-linux-gcc -c -S -O2 gcd.c -o gcd.s
```

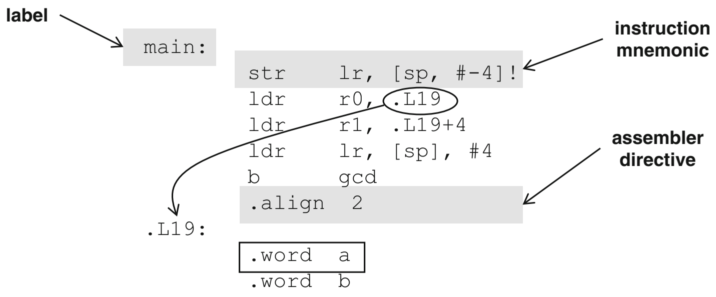
The command to generate the ARM ELF executable is as follows.

```
> /usr/local/arm/bin/arm-linux-gcc -O2 gcd.c -o gcd
```

Both commands run the same program, `arm-linux-gcc`, but the specific function is selected through the use of command-line flags. The default behavior covers both compiling and linking. The `-c` flag is used to end the compilation before the linking process. The `-S` flag is used to create assembly code. The `-O2` flag selects the optimization level.

Listing 7.2 Assembly dump of Listing 7.1

```
gcd:
        str     lr, [sp, #-4]!
        mov     lr, #0
        mov     ip, lr
.L13:
        ldr     r3, [r0, ip, asl #2]
        ldr     r2, [r1, ip, asl #2]
        cmp     r3, r2
        beq     .L17
.L11:
        cmp     r3, r2
        rsbgt   r3, r2, r3
        rsble   r2, r3, r2
        cmp     r3, r2
        bne     .L11
.L17:
        add     ip, ip, #1
        cmp     lr, r3
        movge   lr, r3
        cmp     ip, #4
        movgt   r0, lr
```



```
main:
          str    lr, [sp, #-4]!
          ldr    r0, .L19
          ldr    r1, .L19+4
          ldr    lr, [sp], #4
          b      gcd
          .align  2
.L19:
          .word  a
          .word  b
```

Fig. 7.2 Elements of an assembly program produced by `gcc`

```
22          ldrgt   pc, [sp], #4
23          b       .L13
24  a:
25          .word   26, 3, 33, 56, 11
26  b:
27          .word   87, 12, 23, 45, 17
28  main:
29          str     lr, [sp, #-4]!
30          ldr     r0, .L19
31          ldr     r1, .L19+4
32          ldr     lr, [sp], #4
33          b       gcd
34          .align  2
35  .L19:
36          .word   a
37          .word   b
```

Listing 7.2 is the assembly program generated out of the C program in Listing 7.1. Figure 7.2 illustrates several noteworthy features of an assembly program. The program contains three elements: labels, instructions, and assembler directives. Labels are symbolic addresses. They are used as target locations for branch instructions, and as symbolic locations for variables. In Fig. 7.2, for example, variables `a` and `b` are addressed by the label `.L19` and `.L19+4` respectively. Assembler directives start with a dot; they do not make part of the assembled program, but are used by the assembler. The style of assembly source code shown in Listing 7.2 is common for `gcc`; only the instruction set will change from one cross-compiler to the next.

Understanding assembly programs is vital to understand the performance issues for many kinds of hardware/software codesign problems. In this book, we make use of C programs, and the `gcc` compiler, to create those assembly programs. You will find that it is easy to find a strong correspondence between a C program and its assembly version, even for an unknown processor. Let's compare the C program of Listing 7.1 with the assembly version of it in Listing 7.2.

- The overall *structure* of the assembly program preserves the structure of the C program. The `gcd` function is on lines 1–23, the `main` function is on lines 28–34.

The loop structure of the C program can be identified in the assembly program by inspection of the labels and the corresponding branch instructions. In the gcd function, the inner for loop is on lines 10–15, and the outer while loop is on lines 5–23.

- The constant arrays a and b are directly encoded as constants in the assembly, on lines 24–27. The assembly code does not directly work with these constant arrays, but instead with a *pointer* to these arrays. The storage location at label .L19 will hold a pointer to array a followed by a pointer to array b.
- *Function calls* in assembly code implement the semantics of the C function call, including the passing of parameters and the retrieval of results. Lines 30–32 of the assembly program show how this C function call is implemented. The assembly program copies the starting address of these arrays into r0 and r1. The gcd function in the assembly can make use of r0 and r1 as a pointer to array a and b respectively.

The micro-processor works with object code, binary opcodes generated out of assembly programs. Compiler tools can re-create the assembly code out of the executable format. This is achieved by the objdump program, another program in the gcc toolchain. The following command shows how to retrieve the opcodes for the gcd program.

```
> /usr/local/arm/bin/arm-linux-objdump -d gcd
```

Listing 7.3 shows the object code dump for the gcd program. The instructions are mapped to *sections* of memory, and the .text section holds the instructions of the program. Each function has a particular starting address, measured as an offset from the start of the executable. In this case, the gcd function starts at 0x8380 and the main functions starts at 0x83cc. Listing 7.3 also shows the *opcode* of each instruction, the binary representation of instructions handled by the microprocessor. As part of generating the executable, the address value of each label is encoded into each instruction. For example, the b .L13 instruction on line 23 of Listing 7.2 is encoded as a branch to address 0x838c on line 22 of Listing 7.3.

Listing 7.3 Object dump of Listing 7.2

```
 1  Disassembly of section .text:
 2
 3  00008380 <gcd>:
 4      8380:   e52de004    str lr, [sp, -#4]!
 5      8384:   e3a0e000    mov lr, #0   ; 0x0
 6      8388:   e1a0c00e    mov ip, lr
 7      838c:   e790310c    ldr r3, [r0, ip, lsl #2]
 8      8390:   e791210c    ldr r2, [r1, ip, lsl #2]
 9      8394:   e1530002    cmp r3, r2
10      8398:   0a000004    beq 83b0 <gcd+0x30>
11      839c:   e1530002    cmp r3, r2
12      83a0:   c0623003    rsbgt   r3, r2, r3
13      83a4:   d0632002    rsble   r2, r3, r2
14      83a8:   e1530002    cmp r3, r2
```

```
    83ac:   1afffffa    bne 839c <gcd+0x1c>
    83b0:   e28cc001    add ip, ip, #1  ; 0x1
    83b4:   e15e0003    cmp lr, r3
    83b8:   a1a0e003    movge   lr, r3
    83bc:   e35c0004    cmp ip, #4  ; 0x4
    83c0:   c1a0000e    movgt   r0, lr
    83c4:   c49df004    ldrgt   pc, [sp], #4
    83c8:   eaffffef    b   838c <gcd+0xc>
000083cc <main>:
    83cc:   e52de004    str lr, [sp, -#4]!
    83d0:   e59f0008    ldr r0, [pc, #8]     ;
                              83e0 <main+0x14>
    83d4:   e59f1008    ldr r1, [pc, #8]     ;
                              83e4 <main+0x18>
    83d8:   e49de004    ldr lr, [sp], #4
    83dc:   eaffffe7    b   8380 <gcd>
    83e0:   00010444    andeq   r0, r1, r4, asr #8
    83e4:   00010458    andeq   r0, r1, r8, asr r4
```

We will discuss the use of compiler tools further in Sect. 7.4. First, we will take a closer look at the process of executing microprocessor instructions. In particular, we will discuss the factors that affect the execution time of an instruction.

7.2 The RISC Pipeline

This section describes the internal architecture of a very common type of microprocessor, the Reduced Instruction Set Computer (RISC). We will review the basic ideas in RISC architecture design. The material in this section is typically covered, in far greater depth, in a computer-architecture course.

In a RISC processor, the execution of a single instruction is split in different stages, which are chained together as a pipeline. Each instruction operates on a set of registers contained within the processor. Processor registers are used as operands or as targets for the processor instructions, and for control. For example, the ARM processor contains 17 registers: data register `r0` to `r14`, a program counter register `pc`, and a processor status register `cpsr`. The Microblaze processor has 32 general-purpose registers (`r0` to `r31`) and up to 18 special-purpose registers (such as the program counter, the status register, and more).

Each stage of a RISC pipeline takes one clock cycle to complete. A typical RISC pipeline has three or five stages, and Fig. 7.3 illustrates a five-stage pipeline. The five stages of the pipeline are called Instruction Fetch, Instruction Decode, Execute, Buffer, and Write-back. As an instruction is executed, each of the stages performs the following activities.

- *Instruction Fetch*: The processor retrieves the instruction addressed by the program counter register from the instruction memory.
- *Instruction Decode*: The processor examines the instruction opcode. For the case of a branch-instruction, the program counter will be updated. For the case of a

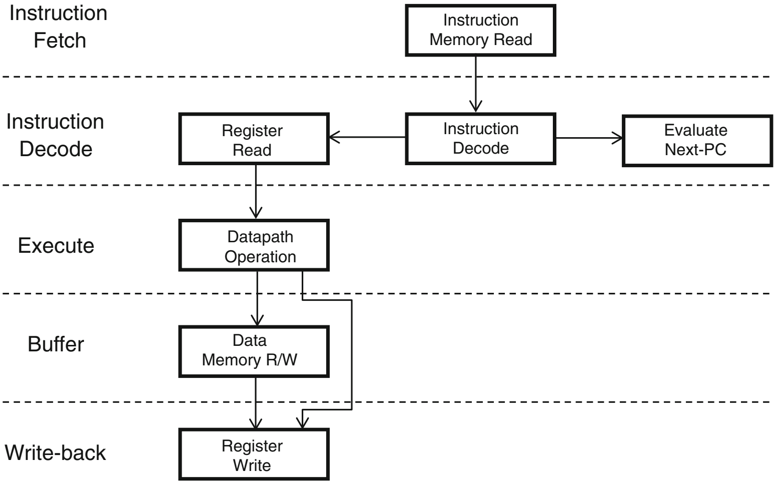


Fig. 7.3 A five-stage RISC pipeline

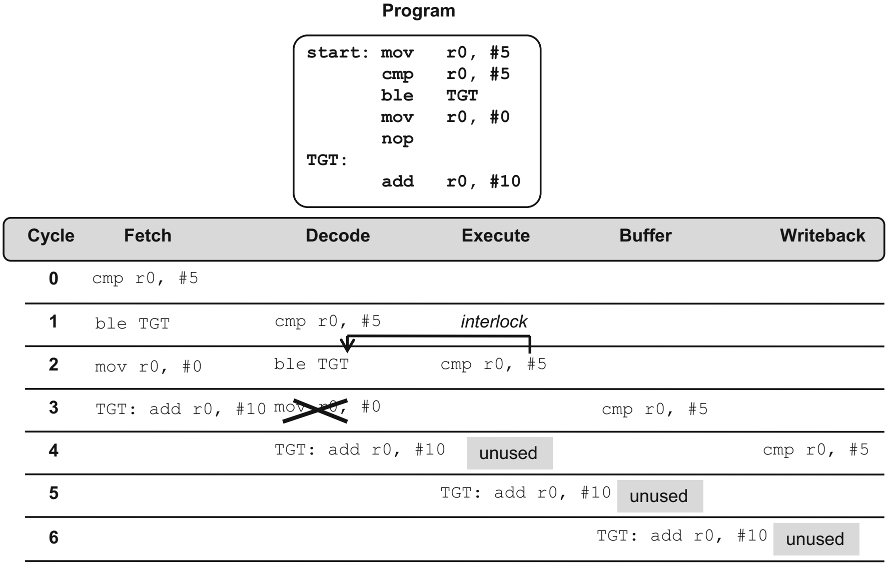
compute-instruction, the processor will retrieve the processor data registers that are used as operands.

- *Execute*: The processor executes the computational part of the instruction on a datapath. In case the instruction will need to access data memory, the execute stage will prepare the address for the data memory.
- *Buffer*: In this stage, the processor may access the data memory, for reading or for writing. In case the instruction does not need to access data memory, the data will be forwarded to the next pipeline stage.
- *Write Back*: In the final stage of the pipeline, the processor registers are updated.

A three-stage RISC pipeline is similar to a five-stage RISC pipeline, but the Execute, Buffer, and Write-back stages are collapsed into a single stage.

Under ideal circumstances, the five-stage RISC pipeline is able to accept a new instruction every clock cycle. Thus, the instruction throughput in a RISC processor may be as high as one instruction every clock cycle. Because of the pipelining, each instruction may take up to five clock cycles to complete. The instruction latency therefore can be up to five clock cycles. A RISC pipeline improves instruction throughput at the expense of instruction latency. However, the increased instruction latency of a RISC processor is usually not a problem because the clock frequency of a pipelined processor is higher than that of a non-pipelined processor.

In some cases it is not possible for an instruction to finish within five clock cycles. A pipeline stall occurs when the progress of instructions through the pipeline is temporarily halted. The cause of such a stall is a pipeline hazard. In advanced RISC processors, pipeline interlock hardware can detect and resolve pipeline hazards automatically. Even when interlock hardware is present, pipeline hazards may still occur. We discuss three different categories of pipeline hazards, along with examples for an ARMv6 processor. The three categories are the following.



```
start: mov   r0, #5
       cmp   r0, #5
       ble   TGT
       mov   r0, #0
       nop
TGT:
       add   r0, #10
```

Cycle	Fetch	Decode	Execute	Buffer	Writeback
0	cmp r0, #5				
1	ble TGT	cmp r0, #5	interlock		
2	mov r0, #0	ble TGT	cmp r0, #5		
3	TGT: add r0, #10	~~mov r0,~~ #0		cmp r0, #5	
4		TGT: add r0, #10	unused		cmp r0, #5
5			TGT: add r0, #10	unused	
6				TGT: add r0, #10	unused

Fig. 7.4 Example of a control hazard

- *Control hazards* are pipeline hazards caused by branches.
- *Data hazards* are pipeline hazards caused by unfulfilled data dependencies.
- *Structural hazards* are caused by resource conflicts and cache misses.

7.2.1 Control Hazards

Branch instructions are the most common form of pipeline stalls. As indicated in Fig. 7.3, a branch is only executed (i.e. it modifies the program counter register) in stage two of the pipeline. At that moment, another instruction has already entered the pipeline. As this instruction is located *after* the branch instruction, that instruction should be thrown away in order to preserve sequential execution semantics.

Figure 7.4 illustrates a control hazard. The pipeline is shown drawn on its side, running from left to right. Time runs down across the rows. A control hazard occurs because of the branch instruction `ble TGT`. In cycle 2, the new program counter value evaluates to the target address of the branch, `TGT`. Note that even though `ble` is a conditional branch that uses the result of the instruction just before that (`cmp r0, #5`), the branch condition is available in cycle 2 because of the interlock hardware in the pipeline. Starting in cycle 3, instructions from the target address `TGT` enter the pipeline. At the same time, the instruction just after the branch is canceled in the decode stage. This results in an unused instruction slot just after the branch instruction.

Some RISC processors include a delayed-branch instruction. The purpose of this instruction is to allow the instruction just after the branch instruction to complete even when the branch is taken. This will prevent ‘unused’ pipeline slots as shown in Fig. 7.4.

For example, the following C function:

```
int accumulate() {
  int i,j;
  for (i=0; i<100; i++)
    j += i;
  return j;
}
```

leads to the following assembly code for a Microblaze processor:

```
      addk r4,r0,r0    ; clear r4 (holds i)
      addk r3,r3,r4    ; j = j + i
$L9:
      addik r4,r4,1    ; i = i + 1
      addik r18,r0,99  ; r18 <- 99
      cmp r18,r4,r18   ; compare i with 99
      bgeid r18,$L9    ; delayed branch if equal
      addk r3,r3,r4    ; j = j + i -> branch delay slot
```

The delayed-branch instruction is `bgeid`, which is a ‘branch if-greater-or-equal delayed’. The instruction just after the branch corresponds to the loop body `j = j + i`. Because it is a delayed-branch instruction, it will be executed regardless if the conditional branch is taken or not.

7.2.2 Data Hazards

A second cause of pipeline stalls are data hazards: pipeline delays caused by the unavailability of data. Processor registers are updated at the end of each instruction, during the write-back phase. But what if the data is required before it has updated a processor register? After all, as indicated in the pipeline diagram in Fig. 7.3, the write-back stage comes two cycles behind the execute stage. An instruction that reaches the write-back stage is two instructions after the instruction that is currently executing. In the following snippet, the `add` instruction will be in the buffer stage by the time the `mov` instruction reaches the write-back stage, and the addition would have already completed.

```
mov  r0, #5
add  r1, r0, r1
```

In a RISC pipeline, this is handled by pipeline interlock hardware. The pipeline interlock hardware observes the read/write patterns of all instructions currently flowing in the RISC pipeline, and makes sure they take data from the right source.

Program

```
start: mov   r0, #5
       ldr   r1, [r0]
       add   r2, r1, #3
       mov   r3, #0
       nop
```

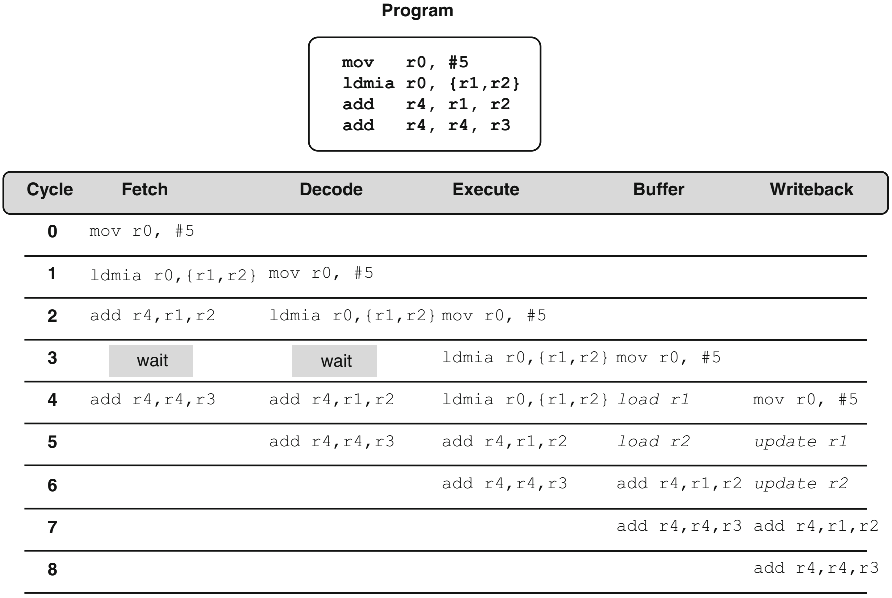
Cycle	Fetch	Decode	Execute	Buffer	Writeback
0	`mov r0, #5`				
1	`ldr r1,[r0]`	`mov r0, #5`			
2	`add r2,r1,#3`	`ldr r1,[r0]`	`mov r0, #5`		
3	`mov r3,#0`	`add r2,r1,#3`	`ldr r1,[r0]`	`mov r0, #5`	
4	wait	wait	wait	`ldr r1,[r0]`	`mov r0, #5`
5		`mov r3,#0`	`add r2,r1,#3`	unused	`ldr r1,[r0]`
6			`mov r3,#0`	`add r2,r1,#3`	unused
7				`mov r3,#0`	`add r2,r1,#3`
8					`mov r3,#0`

Fig. 7.5 Example of a data hazard

Consider the previous example again. When the `add` instruction is in the execute stage, it will use the result of the `mov` instruction as if flows through the buffer stage of the pipeline. This activity is called *forwarding*, and it is handled automatically by the processor.

In some cases, *forwarding* is not possible because the data is simply not yet available. This happens when a read-from-memory instruction is followed by an instruction that uses the data coming from memory. An example of this case is shown in Fig. 7.5. The second instruction fetches data from memory and stores it in register `r1`. The following `add` instruction uses the data from that register as an operand. In cycle 4, the `add` instruction reaches the execute stage. However, during the same clock cycle, the `ldr` instruction is still accessing the data memory. The new value of `r1` is only available at the start of cycle 5. Therefore, the interlock hardware will stall all stages preceding the buffer stage in cycle 4. Starting in cycle 5, the entire pipeline moves forward again, but due to the stall in cycle 4, an unused pipeline slot flushes out in cycles 5 and 6.

Data hazards may lengthen the execution time of an instruction that would normally finish in just five clock cycles. For classic RISC processors, data hazards can be predicted statically, by examining the assembly program. When the execution time of a program needs to be estimated exactly, a programmer will need to identify all data hazards and their effects.



Program

```
mov   r0, #5
ldmia r0, {r1,r2}
add   r4, r1, r2
add   r4, r4, r3
```

Cycle	Fetch	Decode	Execute	Buffer	Writeback
0	`mov r0, #5`				
1	`ldmia r0,{r1,r2}`	`mov r0, #5`			
2	`add r4,r1,r2`	`ldmia r0,{r1,r2}`	`mov r0, #5`		
3	wait	wait	`ldmia r0,{r1,r2}`	`mov r0, #5`	
4	`add r4,r4,r3`	`add r4,r1,r2`	`ldmia r0,{r1,r2}`	*load r1*	`mov r0, #5`
5		`add r4,r4,r3`	`add r4,r1,r2`	*load r2*	*update r1*
6			`add r4,r4,r3`	`add r4,r1,r2`	*update r2*
7				`add r4,r4,r3`	`add r4,r1,r2`
8					`add r4,r4,r3`

Fig. 7.6 Example of a structural hazard

7.2.3 Structural Hazards

The third class of hazards are structural hazards. These are hazards caused by instructions that require more resources from a processor than available. For example, a given instruction may require five concurrent additions while there is only a single ALU available. To implement such an instruction, the execution phase of the instruction will need to be extended over multiple clock cycles, while the pipeline stages before that will be stalled.

Another example of a structural hazard is illustrated in Fig. 7.6. The `ldmia` instruction (ARM) is a load-multiple instruction that reads consecutive memory locations and that stores the resulting values in memory. In the example shown, the value stored in address `r0` will be copied to `r1`, while the value stored in address `r0+4` will be copied to `r2`. When the `ldmia` instruction reaches the execute stage, the execute stage will be busy for two clock cycles in order to evaluate the memory addresses `r0` and `r0+4`. Therefore, all pipeline stages before the execute stage are halted for a single clock cycle. After that, the pipeline proceeds normally.

A structural hazard is caused by the processor architecture, but it may have a broad number of causes: the width of memory ports, the number of execution units in the data-path, or the restrictions on communication busses. A programmer can only predict structural hazards through a solid understanding of the processor architecture. Furthermore, memory latency effects can also cause the execution time of the buffer stage to vary. A cache miss can extend the latency of a load-memory instruction with hundreds of cycles. While the memory-load instruction is waiting for data to be returned from memory, it will stall the pipeline in a manner similar to a structural hazard.

7.3 Program Organization

Efficient hardware/software codesign requires a simultaneous understanding of system architecture and software. This is different from traditional computer science, where a programmer is typically interested in running a C program 'as fast as possible', but without much concern for the computer hardware that runs the C program.

In this section, we consider the relation between the structure of a C program and its' implementation on a RISC processor. We cover the storage of C variables in memory, and the implementation of function calls. We are assuming a 32-bit architecture, and will provide examples based on ARM, Microblaze, and Nios-II RISC processors.

7.3.1 Data Types

A good starting point to discuss the mapping of C programs to RISC processors are the data types used by C programs. Table 7.1 shows how C maps to the native data types supported by 32-bit processors such as ARM, Nios-II, and Microblaze. All C data types, apart from `char`, are treated as signed (two's complement) numbers.

The hardware required for operations on two's complement signed numbers and unsigned numbers is almost identical. One exception is that signed-number arithmetic uses sign extension. Converting, for example, a signed integer to a signed long (64-bit) integer will replicate the most significant bit of the source integer into the upper 32 bits of the long integer. The second difference between signed and unsigned operations is that the comparison operation for signed and unsigned numbers has a different hardware implementation. Indeed, when comparing unsigned bytes, `0xff` is bigger then `0x01`. But, when comparing signed bytes, `0xff` is smaller then `0x01`.

Table 7.1 C compiler data types

C data type	
Char	8-bit
Short	Signed 16-bit
Int	Signed 32-bit
Long	Signed 32-bit
Long long	Signed 64-bit

The mapping of C data types to physical memory locations is affected by several factors.

The first is the *aligment* of data types in the memory. A typical memory organization for 32-bit RISC processors is word-based. Thus, a single memory transfer may be able to access any of the 4 bytes in a word, but a group of 4 bytes across a word boundary cannot be accessed in a single memory transfer. A word-oriented memory organization affects the mapping of data types in logical address space (Fig. 7.7a). A 32-bit integer, for example, cannot straddle a word-boundary address. The RISC processor uses only a single load instruction for an integer, and hence can use only a single memory-access.

A second factor to affect the mapping of data types is the storage order, illustrated in Fig. 7.7b. A little-endian storage order will map the lower-significant bytes of a word into lower memory locations. A big-endian storage order, on the other hand, will map the higher-significant bytes to lower memory locations. In a typical C program, the endianess is of no concern. In hardware/software codesign however, the physical representation of data types is important in the transition of software to hardware and back. Therefore, the endianess of a processor (and in some cases even the bit-ordering) is important. It is easy to check the endianess of a given processor using a small C program such as the following one.

```
int main() {
   char j[4];
   volatile int *pj;
   pj = (int *) j;

   j[0] = 0x12;
   j[1] = 0x34;
   j[2] = 0x56;
   j[3] = 0x78;

   printf("\%x\n", *pj);
}
```

For this program, a little-endian processor will print 78563412, while a big-endian processor will print 12345678. A Microblaze processor is big-endian, a Nios-II processor is little-endian, and an ARM is bi-endian (meaning it can work either way, and the endianess is a configuration of the processor).

7.3.2 Variables in the Memory Hierarchy

A second important implementation aspect of C programs is the relationship between the variables of a C program and the memory locations used to store those variables. The *memory hierarchy* creates the illusion of a uniform, very fast memory space. As illustrated in Fig. 7.8, a memory hierarchy includes the processor registers, the cache memory and the main memory. In embedded processors, cache memory is

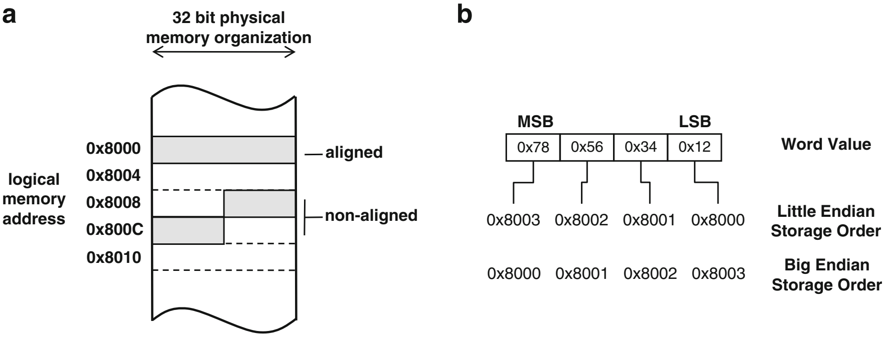


Fig. 7.7 (**a**) Alignment of data types. (**b**) Little-endian and Big-endian storage order

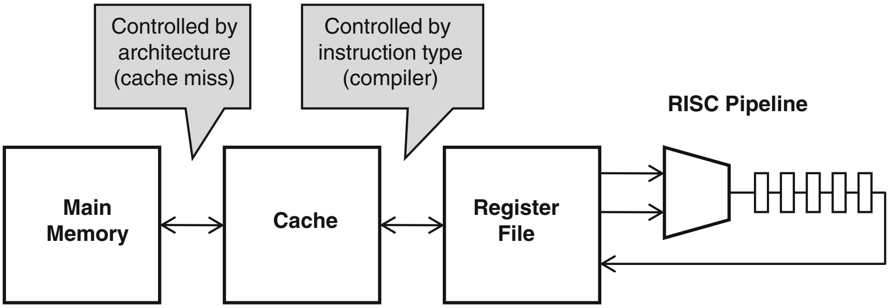

Controlled by architecture (cache miss)
Controlled by instruction type (compiler)
RISC Pipeline
Main Memory
Cache
Register File

Fig. 7.8 Memory hierarchy

optional; in high-end processors on the other hand, multiple levels of cache memory are used.

The cache operates as a fast local memory which holds the most-frequently used main-memory locations. Whenever the processor requests data from a memory location, the cache may report a hit and return a locally-stored copy of the desired memory location. The cache may also report a miss and instead first fetch the requested memory location from main memory. A cache memory improves program efficiency when data (or instructions) are used more than once by a C program. The instructions of the body of a loop, for example, are good candidates for caching. Conversely, when every data element needs to be retrieved only once, a cache memory is of little help.

The memory hierarchy is transparent to a C programmer. Under normal circumstances, a C programmer will not worry what type of memory is used to store the variables from a program. It could be the processors' registers, the cache, or the main-memory. In reality, data travels up and down the memory hierarchy during program execution. This is illustrated by the following example, which shows a C function accumulating an array.

```
void accumulate(int *c, int a[10]) {
  int i;
  *c = 0;
  for (i=0; i<10; i++)
    *c += a[i];
}
```

The ARM assembly code for this program can be created using the ARM cross-compiler. We use optimization-level two for this.

```
/usr/local/arm/bin/arm-linux-gcc -O2 -c -S accumulate.c
```

This generates the following code in accumulate.s:

```
1          mov     r3, #0
2          str     r3, [r0, #0]
3          mov     ip, r3
4   .L6:
5          ldr     r2, [r1, ip, asl #2]  ; r2 <- a[i]
6          ldr     r3, [r0, #0]          ; r3 <- *c (memory)
7          add     ip, ip, #1            ; increment loop ctr
8          add     r3, r3, r2
9          cmp     ip, #9
10         str     r3, [r0, #0]          ; r3 -> *c (memory)
11         movgt   pc, lr
12         b       .L6
```

Let us consider how the accumulator variable is implemented. In the C program, there is only a *single* placeholder for the accumulator, namely *c. In terms of physical memory, there are at three different locations where a copy of *c may be found: the processor registers, the cache, and the main memory. In the assembly implementation, the *value* of the accumulator travels up in the memory hierarchy. According to the C function, the accumulator is provided through a pointer. This implies that the accumulator will be stored in main memory. On line 6 of the previous Listing, that variable is read from memory and stored in processor register r3. On line 10, the processor register r3 is written back to memory. Thus, depending on the nature and state of the cache memory, reading/writing processor registers from/to memory may trigger additional data transfers between the cache memory and the main memory. In the context of hardware/software codesign, the difference between the physical implementation of a C program and its logical design is important. For example, the physical mapping of variables is important when a communication link needs to be created between software and hardware.

A C programmer has a limited amount of control over the mapping of variables in the memory hierarchy. This control is offered through the use of *storage class specifiers* and *type qualifiers*. The most important ones are enumerated in Table 7.2. A few example declarations are shown below.

```
volatile int *c;    // c is a pointer to a volatile int
int * const y;      // y is a constant pointer to an int
register int x;     // x is preferably mapped into a register
```

Table 7.2 C storage class specifiers and type qualifiers

Keyword	Function
Storage specifier	
`register`	Indicates that the preferred mapping of the data type is in processor registers. This will keep the variable as class as possible to the RISC pipeline
`static`	Limits the scope (visibility) of the variable over multiple files. This specifier does not relate to the memory hierarchy, but to the functions where the variable may be accessed
`extern`	Extends the scope (visibility) of the variable to all files. This specifier does not relate to the memory hierarchy, but to the functions where the variable may be accessed
Type qualifier	
`const`	Indicates that the qualified variable cannot be changed
`volatile`	Indicates that the qualified variable can change its value at any time, even outside of the operations in the C program. As a result, the compiler will make sure to write the value always back to main memory after modification, and maintain a copy of it inside of the processor registers

Type qualifiers will be important to access memory-mapped interfaces, which are hardware/software interfaces that appear as memory locations to software. We will discuss this in Chap. 11.

7.3.3 Function Calls

Behavioral hiearchy – C functions calling other functions – is key to mastering complexity with C programs. We briefly describe the concepts of C function calls in the context of RISC processors. We use the example C program in Listing 7.4.

Listing 7.4 Sample program

```

int accumulate(int a[10]) {
  int i;
  int c = 0;
  for (i=0; i<10; i++)
    c += a[i];
  return c;
}

int a[10];
int one = 1;

int main() {
  return one + accumulate(a);
}
```

Let us assume that we have compiled this program for an ARM processor using the `arm-linux-gcc` cross compiler. It is possible to re-create the assembly listing corresponding to the object file by *dis-assembling* the object code. The utility `arm-linux-objdump` takes care of that. The `-d` flag on the command line selects the dis-assembler functionality. The utility supports many other functions (See Sect. 7.4 and Problem 7.8).

```
/usr/local/arm/bin/arm-linux-objdump -O2 -c accumulate.c -o
   accumulate.o
/usr/local/arm/bin/arm-linux-objdump -d accumulate
```

The ARM assembly listing of this program is shown in Listing 7.5.

Listing 7.5 Sample program

```
00000000 <accumulate>:
   0:   e3a01000        mov     r1, #0
   4:   e1a02001        mov     r2, r1
   8:   e7903102        ldr     r3, [r0, r2, lsl #2]
   c:   e2822001        add     r2, r2, #1
  10:   e3520009        cmp     r2, #9
  14:   e0811003        add     r1, r1, r3
  18:   c1a00001        movgt   r0, r1
  1c:   c1a0f00e        movgt   pc, lr
  20:   ea000000        b       8 <accumulate+0x8>

00000024 <main>:
  24:   e52de004        str     lr, [sp, -#4]!
  28:   e59f0014        ldr     r0, [pc, #20]    ; 44 <main+0x20>
  2c:   ebfffffe        bl      0 <accumulate>
  30:   e59f2010        ldr     r2, [pc, #16]    ; 48 <main+0x24>
  34:   e5923000        ldr     r3, [r2]
  38:   e0833000        add     r3, r3, r0
  3c:   e1a00003        mov     r0, r3
  40:   e49df004        ldr     pc, [sp], #4
        ...
```

Close inspection of the instructions reveals many practical aspects of the runtime layout of this program, and in particular of the implementation of function calls. The instruction that branches into `accumulate` is implemented at address `0x2c` with a `bl` instruction – *branch with link*. This instruction will copy the program counter in a separate link register `lr`, and load the address of the branch target into the program counter. A return-from-subroutine can now be implemented by copying the link register back into the program counter. This is shown at address `0x1c` in `accumulate`. Obviously, care must be taken when making nested subroutine calls so that `lr` is not overwritten. In the `main` function, this is solved at the entry, at address `0x24`. There is an instruction that copies the current contents of `lr` into a local area within the stack, and at the end of the main function the program counter is directly read from the same location.

The arguments and return value of the `accumulate` function are passed through register `r0` rather than main memory. This is obviously much faster when only a few data elements need to be copied. The input argument of `accumulate` is the base address from the array a. Indeed, the instruction on address 8 uses r0 as a base address and adds the loop counter multiplied by 4. This expression thus results in the effective address of element a[i] as shown on line 5 of the C program (Listing 7.4). The return argument from accumulate is register r0 as well. On address 0x18 of the assembly program, the accumulator value is passed from r1 to r0. For ARM processors, the full details of the procedure-calling convention are defined in the *ARM Procedure Call Standard* (APCS), a document used by compiler writers and software library developers.

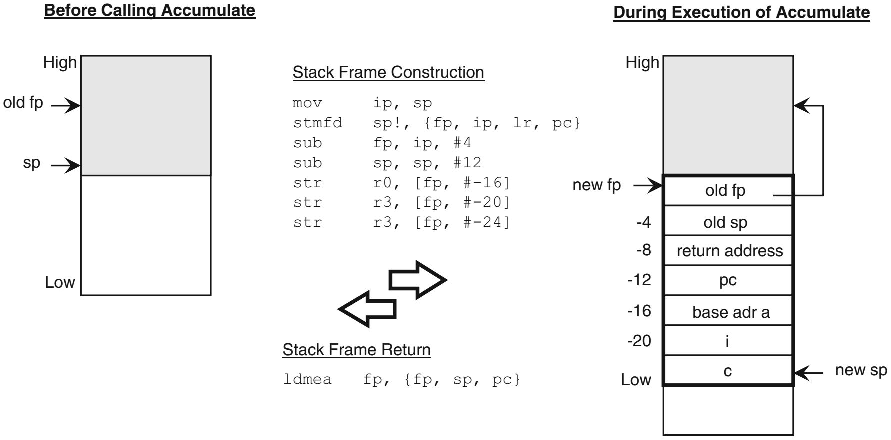
In general, arguments are passed from function to function through a data structure known as a *stack frame*. A stack frame holds the return address, the local variables, the input and output arguments of the function, and the location of the calling stack frame. A full-fledged stack frame can be found when the accumulate function described earlier is compiled *without* optimizations. In that case, the C compiler takes a conservative approach and keeps all local variables in main memory, rather than in registers.

```
/usr/local/arm/bin/arm-linux-gcc -c -S accumulate.c
```

Listing 7.6 shows the resulting non-optimized assembly code of accumulate. Figure 7.9 illustrates the construction of the stack frame.

Listing 7.6 Accumulate without compiler optimizations

```
accumulate:
    mov     ip, sp
    stmfd   sp!, {fp, ip, lr, pc}
    sub     fp, ip, #4
    sub     sp, sp, #12
    str     r0, [fp, #-16]    ; base address a
    mov     r3, #0
    str     r3, [fp, #-24]    ; c
    mov     r3, #0
    str     r3, [fp, #-20]    ; i
.L2:
    ldr     r3, [fp, #-20]
    cmp     r3, #9            ; i<10 ?
    ble     .L5
    b       .L3
.L5:
    ldr     r3, [fp, #-20]    ; i * 4
    mov     r2, r3, asl #2
    ldr     r3, [fp, #-16]
    add     r3, r2, r3        ; *a + 4 * i
    ldr     r2, [fp, #-24]
    ldr     r3, [r3, #0]
    add     r3, r2, r3        ; c = c + a[i]
    str     r3, [fp, #-24]    ; update c
    ldr     r3, [fp, #-20]
    add     r3, r3, #1
    str     r3, [fp, #-20]    ; i = i + 1
    b       .L2
```



Stack Frame Construction

```
mov      ip, sp
stmfd    sp!, {fp, ip, lr, pc}
sub      fp, ip, #4
sub      sp, sp, #12
str      r0, [fp, #-16]
str      r3, [fp, #-20]
str      r3, [fp, #-24]
```

Stack Frame Return

```
ldmea    fp, {fp, sp, pc}
```

Fig. 7.9 Stack frame construction

```
29  .L3:
30          ldr     r3, [fp, #-24]     ; return arg
31          mov     r0, r3
32          ldmea   fp, {fp, sp, pc}
```

The instructions on lines 2 and 3 are used to create the stack frame. On line 3, the frame pointer (fp), stack pointer (sp), link register or return address (lr) and current program counter (pc) are pushed onto the stack. The single instruction stmfd is able to perform multiple transfers (m), and it grows the stack downward (fd). These four elements take up 16 bytes of stack memory.

On line 3, the frame pointer is made to point to the first word of the stack frame. All variables stored in the stack frame will now be referenced based on the frame pointer fp. Since the first four words in the stack frame are already occupied, the first free word is at address fp - 16, the next free word is at address fp - 20, and so on. These addresses may be found back in Listing 7.6.

The following local variables of the function accumulate are stored within the stack frame: the base address of a, the variable i, and the variable c. Finally, on line 31, a return instruction is shown. With a single instruction, the frame pointer fp, the stack pointer sp, and the program counter pc are restored to the values just before calling the accumulate function.

7.3.4 Program Layout

Another aspect of C program implementation is the physical representation of the program and its data structures in the memory hierarchy. This leads to the *program layout*, the template that is used to organize instructions and data. A distinction must be made between the organization of a compiled C program in an executable file (or a program ROM), and the memory organization of that C program during

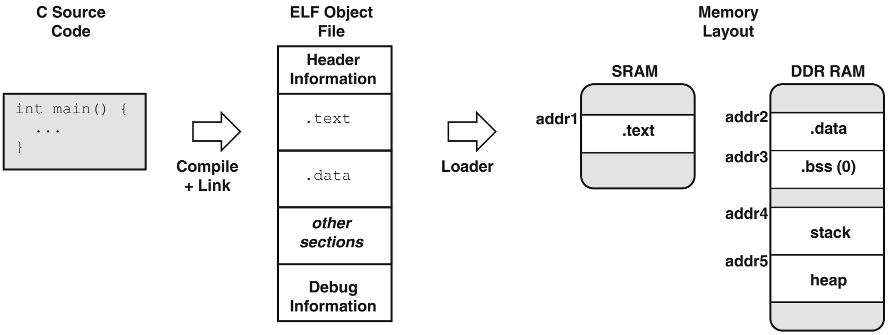


Fig. 7.10 Static and dynamic program layout

execution. The former case is a static representation of all the instructions and constants defined by the C program. The latter case is a dynamic representation of all the instructions and the runtime data structures such as the stack and the heap.

Figure 7.10 shows how a C program is compiled into an executable file, which in turn is mapped into memory. There are several standards available for the organization of executable files. In the figure, the example of ELF (Executable Linkable Format) is shown. An ELF file is organized into sections, and each of these can take up a variable amount of space in the file. The sections commonly found in an ELF file are the `.text` section which contains the binary instructions of the C program and the `.data` section which contains initialized data (constants). The ELF file may also contain debugging information, such as the names of the variables in the C program. This debugging information is utilized by source level debuggers to relate the execution of a binary program to actions of the C source code.

When a compiled C program is first loaded into memory for execution, the ELF file is analyzed by the loader and the sections with relevant information are copied into memory locations. In contrast to a file, the resulting organization of instructions and data into memory does not need to be contiguous or even occupy the same physical memory. Each section of an ELF file can be mapped at a different address, and possibly map into a different memory module. The example in Fig. 7.10 shows how the `.text` segment maps into fast static RAM memory (SRAM) while the `.data`, stack and heap segments map into DDR RAM memory. During program execution, there may be sections of memory which do not appear in the ELF file, or which do not occupy any area within the ELF file. These sections include data storage areas: dynamic data (heap), local data (stack), and global data (bss).

7.4 Compiler Tools

A C compiler provides utilities to inspect the organization of an executable file or an object file. We can use those utilities to gain deeper insight into the low-level characteristics of the executable: the size of the code segment, the entry points of function calls, and the assembly code. In this section, we show several examples for the case of the GNU compiler toolchain.

Listing 7.7 shows the example program. The core function of this program, `modulo`, computes the input modulo 179. An interesting aspect is that it computes the output without dividing by the number 179. Instead, the function uses repeated calls to `modk` and `divk`, which are divisions and modulo-operations for powers of 2. A function like `modulo` would be useful, for example, on a processor that does not have a divide instruction.

We compile an object file and executable for Listing 7.7 using the `nios2-elf-gcc` compiler, with optimization level two (`-O2`), and with full debug information. This compiler targets the Nios-II core used in Altera FPGA. The following examples illustrate some of the output formats produced by the GNU Compiler Toolchain. The examples are not comprehensive, and you'll need to consult the manual pages of these tools for further information. We will show how to find the size, the symbols, the organization, and eventually the assembly instructions for this program.

Listing 7.7 Modulo-179 function

```c
unsigned modk(unsigned x, unsigned k) {
  return (x & ((1 << k) - 1));
}

unsigned divk(unsigned x, unsigned k) {
  return (x >> k);
}

unsigned modulo(unsigned x) {
  unsigned r, q, k, a, m, z;
  m = 0xB3; // 179
  k = 8;
  a = (1 << k) - m;
  r = modk(x, k);
  q = divk(x, k);
  do {
    do {
      r = r + modk(q * a, k);
      q = divk(q * a, k);
    } while (q != 0);
    q = divk(r, k);
    r = modk(r, k);
  } while (q != 0);
  if (r >= m)
    z = r - m;
  else
    z = r;
  return z;
}

```

```
int num[4] = {221, 322, 768, 121};
int res[4];

int main() {
  unsigned i;
  for (i=0; i<4; i++)
    res[i] = modulo(num[i]);
  return 0;
}
```

In the following, we assume that we have compiled `modulo.c` using the `nios2-elf-gcc` design flow. We are concerned with examining the output of the compilation process.

7.4.1 Examining Size

A basic question of interest is: how large is this program? In resource-constrained applications, knowing the size of a program helps a designer to estimate the memory requirements. The `size` utility shows the static size of a program, that is, the amount of memory required to store instructions, constants, and global variables. The `size` utility is specific to the cross-compiler environment being used. For example, we use `arm-linux-size`, `nios2-elf-size`, and `mb-size` to analyze binary code compiler for ARM, Nios-II, and Microblaze, respectively.

The `nios2-elf-size` utility gives a summary printout of the amount of memory required for a given C program.

```
> nios2-elf-size modulo.o
   text    data     bss     dec     hex filename
    300      16       0     316     13c modulo.o
```

The output of the program on `modulo.o` shows that there are 300 bytes in the `text` section, 16 bytes in the initialized data-section `data`, and 0 bytes in the non-initialized data-section `bss`. The total amount of bytes required for this program is 316 $(300 + 16)$ in decimal notation or 0xf4 in hexadecimal notation. Since Nios-II uses word-size instructions, a text segment of 300 bytes implies 75 instructions (228/4). These include the instructions contained within `modk`, `divk`, `modulo` and `main`. There are also 16 bytes of initialized data, used to store the global array `num`. After linking the compiler `sections.o`, we obtain an executable file `modulo.elf`. The utility `nios2-elf-size` can handle the executable as well. In this case, the amount of code and data increases significantly, due to the inclusion of C libraries into the program. The linked program uses 10,472 bytes for the code, 5,252 bytes of uninitialized data, and 488 bytes of constants. Libraries have great influence on the overall size of the program. In resource-constrained environments, a great deal of effort is done to reduce the size of the libraries.

```
> nios2-elf-size modulo.elf
   text    data     bss     dec     hex filename
  10472    5252     448   16172    3f2c modulo.elf
```

The `size` utility does not show the dynamic memory usage of a program. In fact, it cannot predict the amount of stack or the amount of heap that will be needed. The size of the stack and heap are not the result of compiling code and counting instructions or data bytes. Instead, they are a design parameter selected by the programmer.

7.4.2 *Examining Sections*

Code coming out of a compiler is organized in chunks of continuous memory, called sections. The `text`, `data`, and `bss` sections listed by size are the three most important ones, but a program can define additional sections, for example to store debugging information.

The `objdump` utility allows you to examine the relative size and position of each section, as well as the names of all the functions in it. The `-h` flag of `objdump` generates a printout of section headers. This reveals the following information for the `modulo.elf` program (with only partial information being shown).

```
modulo.elf:     file format elf32-littlenios2

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .entry        00000020  09000000  09000000  00001000  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .exceptions   00000198  09000020  09000020  00001020  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .text         00002670  090001b8  090001b8  000011b8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .rodata       000000c0  09002828  09002828  00003828  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .rwdata       00001484  090028e8  09003d6c  000038e8  2**2
                  CONTENTS, ALLOC, LOAD, DATA, SMALL_DATA
  5 .bss          000001c0  090051f0  090051f0  000051f0  2**2
                  ALLOC, SMALL_DATA
  ...
```

This listing illustrates the name of the sections, their size, the starting address (VMA and LMA), the offset within the ELF file and the alignment in bytes as a power of 2. VMA stands for *virtual memory address* and it reflects the address of the section during execution. LMA stands for *load memory address* and it reflects the address of the section when the program is first loaded into memory. In this case both numbers have the same value. They would be different in cases where the program is stored in a different memory than the one that holds the program at runtime. For example, a Flash memory can store the program sections at the address reflected by LMA. When the program executes, it is copied into RAM memory at the address reflected by VMA.

Another feature of the objdump program is to print the names of all the symbols included in an object file or executable. Symbols are the elements with external visibility, such as functions and variables. The -t flag on nios2-elf-objdump shows the symbol table. The sorted output of this command looks as follows.

```
> nios2-elf-objdump -t modulo.elf | sort

09000000 g       *ABS*     00000000 __alt_mem_Onchip_memory
...
090001b8 l     d .text     00000000 .text
090001b8 g     F .text     0000003c _start
090001f4 g     F .text     00000014 modk
09000208 g     F .text     00000008 divk
09000210 g     F .text     000000c0 modulo
090002d0 g     F .text     0000005c main
...
090028e8 l     d .rwdata   00000000 .rwdata
090028e8 g     O .rwdata   00000010 num
...
090051f0 l     d .bss      00000000 .bss
09005214 g     O .bss      00000010 res
```

The program indicates that the .text segment starts at address 0x90001b8. Next, it lists several symbols. The main function, for example, starts at address 0x90002d0. The symbol table also shows the variables num and res, and their position in the rwdata section and bss section.

Finally, it is also possible to extract symbol table information directly out the linker by inspecting the linker map file. The linker map can be generated as a byproduct of the compilation process. In the GNU C Compiler, a linker map file can be generated with the -Wl,-Map=.. command line option. For example, nios2-elf-gcc will generate the linker map file for modulo.elf through the following options.

```
nios2-elf-gcc  -Wl,-Map=modulo.map -O2 -g -o modulo.elf modulo.c
```

This linker map file comes the closest to the actual memory-map of the implementation; the linker map file will show the size and type of memories defined in the system, and how the different sections of the program map into these memories. The linker map will also show exactly what object files contribute to the final executable. The following listing shows the partial content of a linker map file produced for the modulo program.

```
...
 .text          0x090001b8        0x3c hal_bsp/obj/HAL/src/crt0.o
                0x090001b8                 _start
 .text          0x090001f4       0x138 obj/modulo.o
                0x090001f4                 modk
```

```
0x09000208              divk
0x09000210              modulo
0x090002d0              main
```

7.4.3 Examining Assembly Code

The most detailed inspection of the compiler output is done by looking at the assembly output of the compiler. A straightforward way to do this is to run `gcc` with the -S option. The assembly output for `modulo.c` can be created as follows.

```
nios2-elf-gcc -S -O2 modulo.c
```

This leads to the file `modulo.s`. The `modk` function, for example, looks as follows.

```
modk:
    movi     r3,1
    sll      r2,r3,r5
    sub      r2,r2,r3
    and      r2,r2,r4
    ret
    .size    modk, .-modk
    .align   2
    .global  divk
    .type    divk, @function
```

Assembly code can also be recreated from an object file or an executable using the `objdump` function with the `-D` flag. For an executable, we will see the address as well as the opcode with each instruction. The same `modk` function will look as follows when generated by `nios2-elf-objdump`:

```
090001f4 <modk>:
 90001f4:   00c00044    movi    r3,1
 90001f8:   1944983a    sll     r2,r3,r5
 90001fc:   10c5c83a    sub     r2,r2,r3
 9000200:   1104703a    and     r2,r2,r4
 9000204:   f800283a    ret
```

There are many different ways to query the output of the compiler, using the compiler itself, the linker, and utilities such as `size` and `objdump`. All these utilities are very useful to analyze the detailed, low-level construction of a program. The next section gives an examples of such analysis.

7.5 Low-Level Program Analysis

Because the interaction of hardware and software is so fundamental to hardware/software codesign, understanding software at the lowest level is essential. Optimizing performance requires insight into the low-level construction of the program. For example, what would be the difference between using optimization level `-O2` and `-O3`? How does the compiler generate address expressions from array indices? How does the compiler implement loop counters? These are all questions that can be solved by analyzing the assembly code.

We call this concept *Program Analysis*: interpreting and understanding the performance of a program based on observing the assembly code of that program. Program Analysis is useful for addressing many design activitities, such as the following examples.

- Program analysis provides insight into the optimizations that a C compiler can or cannot offer. Many C programmers are hopeful about the capabilities of a C compiler to produce efficient assembly code. While this is generally true for a high-quality compiler, there are many cases where a compiler cannot help. For example, a compiler is unable to make optimizations that require specific knowledge in the statistical properties of the program data input. A compiler will not transform a program with `double` or `float` variables into one with `int` variables, even if an `int` would give sufficient precision for a particular application. A compiler cannot perform optimizations at high abstraction levels, such converting one type of sorting algorithm into another, equivalent, but more efficient, sorting algorithm. To understand what a compiler can and cannot do, it is helpful to compare C code and the corresponding assembly code.
- Program analysis enables a programmer to make quite accurate predictions on the execution time of a given program. In cases where you are addressing low-level issues, such as controlling hardware modules or controlling hardware interface signals from within software, these timing predictions may be very important.

Listing 7.8 A simple convolution function

```c
int array[256];
int c[256];
int main() {
  int i, a;
  a = 0;
  for (i=0; i<256; i++)
    a += array[i] * c[256 - i];
  return a;
}
```

We will discuss program analysis of Listing 7.8. This program illustrates the C implementation of a convolution operation: the cross-product of a data-array with a reversed data-array. We are interested in the efficiency of this program on a Microblaze processor. For this purpose, we generate the assembly listing of this program using the Microblaze GNU compiler. The optimization level is set at O2. The resulting assembly listing is shown in Listing 7.9.

Listing 7.9 Microblaze assembly for the convolution program

```
        .text
        .align 2
        .globl main
        .ent main
main:
        .frame r1,44,r15
        .mask 0x01c88000
        addik r1,r1,-44
        swi   r22,r1,32
        swi   r23,r1,36
        addik r22,r0,array
        addik r23,r0,c+1024
        swi   r19,r1,28
        swi   r24,r1,40
        swi   r15,r1,0
        addk  r24,r0,r0
        addik r19,r0,255
$L5:
        lwi   r5,r22,0
        lwi   r6,r23,0
        brlid r15,__mulsi3
        addik r19,r19,-1
        addk  r24,r24,r3
        addik r22,r22,4
        bgeid r19,$L5
        addik r23,r23,-4
        addk  r3,r24,r0
        lwi   r15,r1,0
        lwi   r19,r1,28
        lwi   r22,r1,32
        lwi   r23,r1,36
        lwi   r24,r1,40
        rtsd  r15,8
        addik r1,r1,44
        .end  main
$Lfe1:
        .size main,$Lfe1-main
        .bss
        .comm array,1024,4
        .type array, @object
        .comm c,1024,4
        .type c, @object
```

Did the C compiler a good job while compiling the C program in Listing 7.8? In previous sections, we already discussed several of the elements to help us answer this question, including the *stack frame*. Another concept is the *Application Binary Interface* (ABI). The ABI defines how a processor will use its registers to implement a C program. For the case of a Microblaze processor and the example in Listing 7.8, the following aspects are relevant.

- The Microblaze has 32 registers
- Register r0 is always zero and used as a zero-constant.
- Register r1 is the stack pointer.
- Registers r19 through r31 are callee-saved registers: a function that whishes to use these registers must preserve their contents before returning to the caller.

Therefore, we can make the following observations on lines 8–17 from Listing 7.9.

- Line 8 grows the stack pointer by 44 bytes (11 words). Note that the Microblaze stack grows downwards.
- Lines 9, 10, 13, 14 save registers on the stack. These registers (r22, r23, r19, r24) will be used as temporary variables by the program. They are restored just before the main function terminates.

From the values loaded into the working registers, we can infer what they actually represent.

- Register r22 is initialized with array, the starting address of the array.
- Register r23 is initialized with c+1024, which is the start of the c variable plus 1,024. Since the c variable is an array with 256 integers, we conclude that r23 points to the end of the c variable.
- Register r19 is initialized to 255, which is the loop count minus 1.
- Register r24 is initialized to 0, and could be the loop counter or the accumulator.

We now know enough to the tackle the loop body in lines 18–26. Loops in assembly code are easy to find since they always start with a label (like $L5), and terminate with a branch instruction to that label. In this case, the last instruction of the loop body is on line 26 because the branch on line 25 is a delayed-branch (ends with a 'd'). The loop body reads an element from the variables array and c (lines 18 and 19) and stores the result in r5 and r6. The next instruction is a function call. It is implemented in a RISC processor as a branch which saves the return address on the stack. r15 is used to hold the return address. The function is called __mulsi3. From its name, this function hints to be a multiplication, indicating that the compiler generated code for a micro-processor without a built-in multiplier. The multiplication will support the implementation of the following C code.

```
a += array[i] * c[256 - i];
```

The result of the function __mulsi3 is provided in registers r3 and r4 (this is another convention of the ABI). Indeed we see that r3 is accumulated to r24 on line 21. This clears up the meaning of register r24: it is the accumulator. Note that there are three adjustments to counter values in the loop body: Register r19 is decremented by 1, register r22 is incremented by 4, and register r23 is decremented by 4. The adjustment to r23 is still part of the loop because this instruction is located in the branch delay slot after the bgeid branch. We already know that register r22 and r32 are pointers pointing to the variables array and c. Register r19 is the loop counter. Thus, we conclude that the compiler was able to find out that the address expressions for `c` and `array` are sequential, just as the loop counter `i`. It is as if the compiler has automatically performed the following very effective optimization:

```
int array[256];
int c[256];
int main() {
  int i, a;
  int *p1, *p2;
  p1 = array;
  p2 = &(c[255]);
  a = 0;
  for (i=0; i<256; i++)
    a += (*(p1++)) * (*(p2--));
  return a;
}
```

The C compiler is thus able to do fairly advanced dataflow analysis and optimization (when the optimization flag is turned on). Static program analysis does not reveal cycle counts and performance numbers. Rather, it provides a qualitative appreciation of a program. Being able to investigate assembly code, even for processors foreign to you, enables you to make accurate decisions on potential software performance.

7.6 Processor Simulation

When working with embedded processors, it is frequently needed to simulate a model of the processor before the actual design is made. The difference with the hardware simulations we discussed so far, is that embedded processor simulation needs a cross-compiled software binary to run.

7.6.1 Instruction-Set Simulation

Simulations with processor models are very common in hardware-software codesign; they are meant to test the executables created with a cross-compiler, and to evaluate the performance of the resulting program. Micro-processors such as ARM can be simulated with an instruction-set simulator, a simulation engine specialized at simulating the instruction-set for a particular micro-processor.

The GEZEL cosimulation environment integrates several instruction-simulation engines, including one for the ARM processor, one for the 8051 micro-controller, one for the picoblaze micro-controller, and one for AVR micro-controller. These simulation engines are open-source software projects by themselves. SimIt-ARM was developed by Wei Qin, the Dalton 8051 simulator was developed by the team

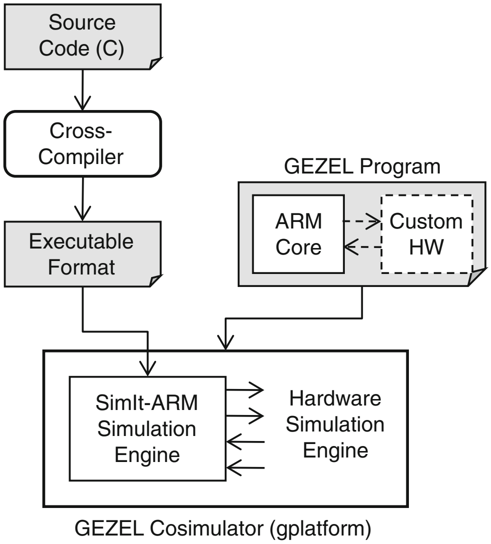

Source Code (C)
Cross-Compiler
Executable Format
GEZEL Program
ARM Core
Custom HW
SimIt-ARM Simulation Engine
Hardware Simulation Engine
GEZEL Cosimulator (gplatform)

Fig. 7.11 An instruction-set simulator, integrated into GEZEL, can simulate cross-compiled executables

of Frank Vahid, the Picoblaze simulator was developed by Mark Six, and the AVR simulator is derived from the SimulAVR project.

Figure 7.11 shows how instruction-set simulators are integrated into the GEZEL cosimulation engine, `gplatform`. In this figure, an instruction set simulator is used to combine two pieces of a cosimulation: a hardware description written in GEZEL, and a software application for an embedded core. The software application for the embedded core is written in C and compiled into executable code using a cross compiler. The hardware part of the application is written in GEZEL, and it specifies the platform architecture: the microprocessor, and its interaction with other hardware modules. The combination of the GEZEL program and the cross-compiled executable format is the input for the cosimulation.

The GEZEL cosimulator is cycle-accurate, including all the instruction-set simulators integrated within `gplatform`. Listing 7.10 shows a GEZEL program that simulates a stand-alone ARM core to execute the `gcd` program of Listing 7.1. Lines 1–4 define an ARM core which runs an executable program called `gcd`. The `ipblock` is a special type of GEZEL module which represents a black-box simulation model, a simulation model without internal details. This particular module does not have any input/output ports. We will introduce such input/output ports while discussing the various hardware/software interfaces (Chap. 11). Lines 6–12 of the GEZEL program simply configure the `myarm` module for execution.

Listing 7.10 A GEZEL top-level module with a single ARM core

```
1  ipblock myarm {
2    iptype "armsystem";
3    ipparm "exec = gcd";
4  }
```

```

dp top {
  use myarm;
}

system S {
  top;
}
```

To simulate the program, we will need to cross-compile the C application software for the ARM instruction-set simulator. Next, we run the instruction-set simulator. To generate output through the co-simulation, we modified the `main` function of the C program as follows:

```
int main() {
  printf("gcd(a,b)=%d\n", gcd(a,b));
  return 0;
}
```

The compilation and co-simulation is now done through the following commands. The output of the simulation shows that the program takes 14,338 cycles to execute.

```
> /usr/local/arm/bin/arm-linux-gcc -static gcd.c -o gcd
> gplatform top.fdl
core myarm
armsystem: loading executable [gcd]
gcd(a,b)=3
Total Cycles: 14338
```

Cycle-accurate simulation provides a detailed view on the execution of an embedded software program. Some instruction-set simulators provide feedback on the execution of individual instructions in the processor pipeline. The next subsection shows an example.

7.6.2 Analysis Based on Execution of Object Code

Processor features such as pipeline stalls and cache misses are not easy to determine using static program analysis alone. Using processor simulation, a designer can observe the dynamic effects of program execution.

SimIt-ARM, one of the instruction simulators integrated in GEZEL, is able to report the activities of each instruction as it flows through the processor pipeline. This includes quantities such as the value of the program counter, the simulation cycle-count, and the instruction completion time. Obviously, collecting this type of information will generate huge amounts of data, and a programmer needs to trace instructions selectively. SimIt-ARM provides the means to turn the instruction-tracing feature on or off, so that a designer can focus on a particular program area of interest.

Listing 7.11 shows the listing of a GCD function. Lines 11–13 illustrate a pseudo-systemcall that is used to turn the instruction-tracing feature of SimIt-ARM on and off. This pseudo-systemcall is simulator-specific, and will be implemented differently when a different processor or simulation environment is used. As shown in the `main` function on lines 17–19, the `gcd` function is called after turning the tracing feature on, and it is turned-off again after that.

Listing 7.11 Microblaze assembly for the convolution program

```
 1  int gcd (int a, int b) {
 2    while (a != b) {
 3      if (a > b)
 4        a = a - b;
 5      else
 6        b = b - a;
 7    }
 8    return a;
 9  }
10
11  void instructiontrace(unsigned a) {
12    asm("swi␣514");
13  }
14
15  int main() {
16    int a, i;
17    instructiontrace(1);
18    a = gcd(6, 8);
19    instructiontrace(0);
20    printf("GCD␣=␣\%d\n", a);
21    return 0;
22  }
```

We will also generate the assembly code for the gcd function, which is useful as a guide during instruction tracing. Listing 7.12 shows the resulting code, annotated with the corresponding C statements in Listing 7.11. First, let's look at the execution without compiler optimization.

```
/usr/local/arm/bin/arm-linux-gcc -static -S gcd.c -o gcd.S
```

Listing 7.12 ARM assembly code for gcd function

```
gcd:
    mov ip, sp                  ; set up stack frame
    stmfd   sp!, {fp, ip, lr, pc}
    sub fp, ip, #4
    sub sp, sp, #8
    str r0, [fp, #-16]          ; storage for var_a
    str r1, [fp, #-20]          ; storage for var_b
.L2:
    ldr r2, [fp, #-16]
    ldr r3, [fp, #-20]
    cmp r2, r3                  ; while (var_a != var_b)
```

```
        bne .L4
        b   .L3
    .L4:
        ldr r2, [fp, #-16]          ;  if (var_a > var_b)
        ldr r3, [fp, #-20]
        cmp r2, r3
        ble .L5
        ldr r3, [fp, #-16]
        ldr r2, [fp, #-20]
        rsb r3, r2, r3              ;   var_a = var_a - var_b;
        str r3, [fp, #-16]
        b   .L2
    .L5:                                    ;  else
        ldr r3, [fp, #-20]
        ldr r2, [fp, #-16]
        rsb r3, r2, r3              ;   var_b = var_b - var_a;
        str r3, [fp, #-20]
        b   .L2
    .L3:
        mov r0, r3
        ldmea   fp, {fp, sp, pc}
```

The Simit-ARM processor configuration uses the following parameters.

- D-cache of 16 KB, organized as a 32-set associative cache with a line size of 32-bytes.
- I-cache of 16 KB, organized as a 32-set associative cache with a line size of 32-bytes.
- Sixty-four-cycle memory-access latency, one-cycle cache-access latency.

The operation of a set-associative cache is as follows. Consider the address mapping used by a 16 KB set-associative cache with 32 sets and a line size of 32 bytes. Since the entire cache is 16 KB, each of the 32 sets in the cache contains 512 bytes or 16 lines. If we number the cache lines from 0 to 15, then address n from the address space will map into line $(n/32) mod\, 16$. For example, assume that the processor performs an instruction fetch from address `0x8524`. Figure 7.12 shows how this address maps into the second word of the tenth line of the cache. The cache will thus check each tenth line in each of the 32 sets before declaring a cache-miss.

We can now perform the simulation with instruction tracing on. The output is shown, in part, below. The columns in this listing have the following meaning.

- **Cycle**: The simulation cycle count at the instruction fetch
- **Addr**: The location of that instruction in program memory
- **Opcode**: The instruction opcode
- **P**: Pipeline mis-speculation. If a 1 appears in this column, then the instruction is not completed but removed from the pipeline.
- **I**: Instruction-cache miss. If a 1 appears in this column, then there is a cache miss when this instruction is fetched.

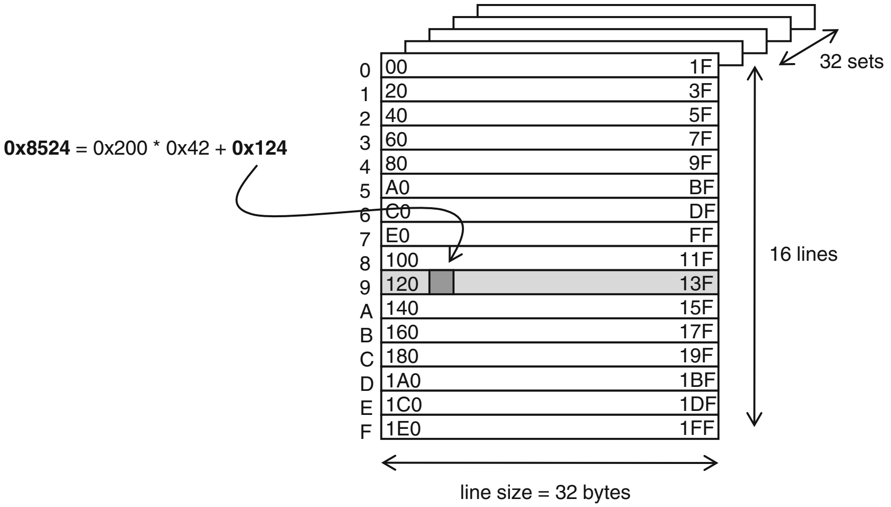


Fig. 7.12 Mapping of address 0x8524 in a 32-set, 16-line, 32-bytes-per-line set-associative cache

- **D**: Data-cache miss. If a 1 appears in this column, then there is a data cache miss when this instruction executes.
- **Time**: The total time that this instruction is active in the pipeline, from the cycle it is fetched to the cycle it is retired.
- **Mnemonic**: Assembly code for this instruction.

```
Cycle     Addr   Opcode   P I D Time  Mnemonic
30601     81e4   e1a0c00d 0 1 0   70  mov   ip, sp;
30667     81e8   e92dd800 0 0 0    8  stmdb sp!, {fp, ip, lr, pc};
30668     81ec   e24cb004 0 0 0    8  sub   fp, ip, #4;
30672     81f0   e24dd008 0 0 0    5  sub   sp, sp, #8;
30673     81f4   e50b0010 0 0 0    5  str   r0, [fp, #-16];
30674     81f8   e50b1014 0 0 0    5  str   r1, [fp, #-20];
30675     81fc   e51b2010 0 0 0    5  ldr   r2, [fp, #-16];
30676     8200   e51b3014 0 1 0   70  ldr   r3, [fp, #-20];
30742     8204   e1520003 0 0 0    6  cmp   r2, r3;
30743     8208   1a000000 0 0 0    3  bne   0x8210;
30745     820c   ea00000d 1 0 0    1  b     0x8248;
30746     8210   e51b2010 0 0 0    5  ldr   r2, [fp, #-16];
30747     8214   e51b3014 0 0 0    5  ldr   r3, [fp, #-20];
30748     8218   e1520003 0 0 0    6  cmp   r2, r3;
30749     821c   da000004 0 0 0    3  ble   0x8234;
30751     8220   e51b3010 1 1 0    1  ldr   r3, [fp, #-16];
30752     8234   e51b3014 0 1 0   69  ldr   r3, [fp, #-20];
30817     8238   e51b2010 0 0 0    5  ldr   r2, [fp, #-16];
30818     823c   e0623003 0 0 0    6  rsb   r3, r2, r3;
30819     8240   e50b3014 0 0 0    6  str   r3, [fp, #-20];
30821     8244   eaffffec 0 0 0    2  b     0x81fc;
30822     8248   e1a00003 1 0 0    1  mov   r0, r3;
```

```
30823     81fc    e51b2010 0 0 0    5  ldr   r2, [fp, #-16];
30824     8200    e51b3014 0 0 0    5  ldr   r3, [fp, #-20];
30826     8208    1a000000 0 0 0    3  bne   0x8210;
30828     820c    ea00000d 1 0 0    1  b     0x8248;
```

First, find a few instructions in the table that have a ‘1’ in the P column. These are pipeline mis-speculations. They happen, for example, at cycle 30745 and cycle 30751. You can see that these instructions come just *after* a branch instruction, and thus they are caused by a control hazard. Next observe the execution time of the instructions. Most instructions take less than 6 clock cycles, but a few take over 50 clock cycles. As indicated in the I and D column, these instructions are slowed down by cache misses. For example, the instruction at cycle 30676, address 0x8200, is an instruction-cache miss, and so is the instruction at cycle 30752, address 0x8234.

It is possible to explain why an instruction causes an I-cache miss? Indeed, this is possible, and there are two cases to consider. The first case is when a linear sequence of instructions is executing. In that case, I-cache misses will occur at the boundary of the cache-lines. In a cache organization with a line size of 32 bytes, cache misses will thus occur at multiples of 32 bytes (0x20 in hex). The instruction at address 0x8200 is an example of this case. This is the first instruction of a cache line which is not in the cache. Therefore, the instruction-fetch stage stalls for 64 clock cycles in order to update the cache. The second case is when a jump instruction executes and moves to a program location which is not in the cache. In that case, the target address may be in the middle of a cache line, and a cache miss may still occur. The instruction at address 0x8234 is an example of that case. That instruction is executed as a result of jump. In fact, the instruction just before that (at address 0x8220) is also cache miss. That instruction does not complete, however, because it is part of a control hazard.

Finally, observe also that some regular instructions take five clock cycles to complete, while others take six clock cycles. Relevant examples are the instructions on addresses 0x8214 and 0x8218. The first of these instructions is a memory-fetch that loads the value of a local variable (`b`) into register `r3`. The following instruction is a compare instruction that uses the value of `r3`. As discussed earlier, this is an example of a data hazard, where the value of a register is only available after the buffer stage of the RISC pipeline. The compare-instruction at address 0x8218 cannot benefit from pipeline interlock hardware and it must be stalled for one clock cycle until the result is available from data-memory.

Using processor simulation at low abstraction level is therefore useful to understand the precise behavior of an embedded software program.

7.6.3 Simulation at Low Abstraction Level

Even more simulation detail is available at the level of a hardware simulation. Figure 7.13 illustrates how a cosimulation can be created using a hardware description language. By replacing the instruction-set simulator with a hardware

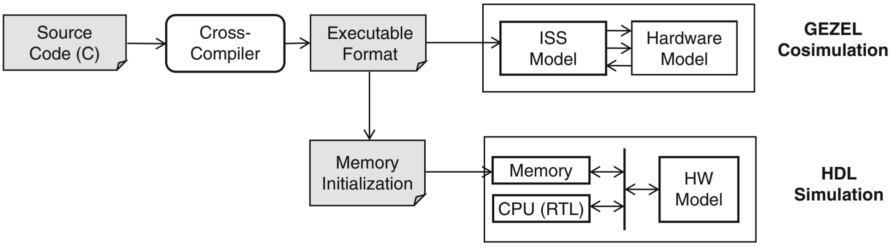


Fig. 7.13 Processor simulation using a hardware simulator

model (in a HDL) of the actual processor, a standard hardware simulator can be used. This type of simulation is very detailed, but it may have a few caveats as opposed to the use of an instruction-set simulator.

- One needs a processor HDL model before such a simulation can be done. However, not all processors have a freely available Verilog or VHDL implementation. Although several processors (OpenRISC and Leon, for example) have freely available HDL, the models for typical commercial processors are closed. Indeed, a very detailed simulation model is considered to be intellectual property of the processor designer.
- The speed of a simulation decreases with the level of simulation detail. HDL-level simulations of full processor models are typically much slower than ISS based processor models. Furthermore, a co-design puts emphasis on the interaction between hardware and software. It may therefore be useful to abstract the internal details of the processor during simulation.
- The HDL based simulation of a processor needs special preparation. In particular, before the simulation can start, the program memory of the processor needs to be properly initialized. This is done by using a memory initialization file.
- The high level of detail may implies the loss of abstraction. For example, in an HDL based cosimulation, the distinction between hardware and software is effectively lost: everything is expressed in terms of Verilog or VHDL. In particular when analyzing the behavior of software, the loss of abstraction is problematic. Using only the HDL simulation, a designer can trace the program counter value and the instruction op-codes. The interpretation of those opcodes is still the task of the designer.

Hence, although HDL based simulation of processor models is attractive in the later stages of a design, it is not a panacea. Instruction-set simulation plays a very important role in the initial development of embedded software.

7.7 Summary

In this chapter, we discussed the organization and operation of typical RISC processors, using the ARM, the Nios-II and the Microblaze as examples. In hardware-software codesign, processors are the entry-point of software into the hardware world. Hence, to analyze the operation of a low-level hardware-software interface, it is very useful to understand the link between a C program, its' assembly instructions, and the behavior of these instructions in the processor pipeline. The execution of software by a RISC processor is affected by the behavior of the RISC pipeline, its memory hierarchy, and the organization of instructions and data into memory. Through the understanding of a limited set of concepts in C, these complex interactions can be understood and controlled to a fairly detailed level. For example, the mapping of data types to memory can be influenced with storage qualifiers, and detailed performance optimization is possible through careful rewriting of C code in combination with study of the resulting program through static and dynamic analysis. This chapter has prepared us for the next big step in a hardware/software codesigned system: the extension of a simple RISC processor into a system-on-chip architecture that integrates software, processors, and custom hardware functions. Clearly, the RISC processor will play a pivotal role in this story.

7.8 Further Reading

The classic work on RISC processor architectures is by Hennessy and Patterson (2006). It is essential reading if you want to delve into the internals of RISC processors. Good documentation on the low-level software tools such as `size` and `objdump` is not easy to find; the manual pages unfortunately are rather specialized. Books on Embedded Linux Programming, such as (Yaghmour et al. 2008), are the right place to start if the man pages do not help. The ELF format is described in detail in the Tool Interface Standard ELF format (ELF Committee 1995). Processor documentation can be found with the processor designers or processor vendors. For example, ARM has an extensive on-line library documenting all the features of ARM processors (ARM 2009b), and Xilinx provides a detailed specification of the Microblaze instruction-set (Xilinx 2009a).

An effective method to learn about the low-level implementation of a RISC core is to implement one, for example starting from open source code. The LEON series of processors by Gaisler Aeroflex, for example, provides a complete collection of HDL source code, compilers, and debuggers (Aeroflex 2009). The internals of a processor simulator, and of the SimIt-ARM instruction-set simulator, are described by Qin in several articles (D'Errico and Qin 2006; Qin and Malik 2003).

7.9 Problems

Problem 7.1. Write a short C program that helps you to determine if the stack grows upwards or downwards.

Problem 7.2. Write a short C program that helps you to determine the position of the stack segment, the text segment, the heap, and data segment (global variables).

Problem 7.3. Explain the difference between the following terms:

- Control hazard and data hazard
- Delayed branch and conditional branch
- Little Endian and Big Endian
- `volatile int * a` and `int * const a`
- Virtual Memory Address (VMA) and Load Memory Address (LMA)

Problem 7.4. This problem considers C Qualifiers and Specifiers.

(a) Correct or not: The `volatile` qualifier will prevent a processor from storing that variable in the cache?

(b) When writing a C program, you can create an integer variable a as follows: `register int a`. This specifier tells the compiler that `a` should be preferably kept in a register as much as possible, in the interest of program execution speed. Explain why this specifier cannot be used for the memory-mapped registers in a hardware coprocessor.

Problem 7.5. The following C function was compiled for Microblaze with optimization-level O2. It results in a sequence of four assembly instructions. Carefully examine the C code (Listing 7.13) and the assembly code (Listing 7.14), and answer the following questions. Note that register `r5` holds the function argument and register `r3` holds the function return value.

(a) Explain why the assembly code does not have a loop?

(b) Suppose line 5 of the C program reads `a = a - 1` instead of `a = a + 1`. Determine how the assembly code would change.

Listing 7.13 C listing for Problem 7.5

```
1  int dummy(int a) {
2    int i, j = a;
3    for (i=0; i<3; i++) {
4      j += a;
5      a  = a + 1;
6    }
7    return a + j;
8  }
```

Listing 7.14 Assembly listing for Problem 7.5

```
    muli  r3, r5,  4
    addk  r4, r3, r5
    rtsd  r15, 8
    addik r3, r3, 6
```

Problem 7.6. The following C statement implements a pseudorandom generator. It translates to the sequence of assembly instructions as shown in Listing 7.15. The assembly instructions are those of a five-stage pipelined StrongARM processor.

```
unsigned rnstate;
rnstate = (rnstate >> 1) ^ (-(signed int)(rnstate &1)
                                 & 0xd0000001u);
```

Answer the following questions:

(a) What is the purpose of line 5 in Listing 7.15 (the `rsb` instruction) in the StrongArm Code? Point out exactly what part of the C expression it will implement.
(b) What types of hazard can be caused by line 3 in Listing 7.15?

Listing 7.15 Assembly listing for Problem 7.6

```
1       ldr     r3, [fp, #-16];  // load-register
2       mov     r2, r3, lsr #1;  // lsr = shift-right
3       ldr     r3, [fp, #-16];
4       and     r3, r3, #1;
5       rsb     r3, r3, #0;
6       and     r3, r3, #-805306367;
7       eor     r3, r2, r3;
8       str     r3, [fp, #-16];  // store-register
```

Problem 7.7. The C in Listing 7.16 was compiled for StrongARM using the following command:

```
/usr/local/arm/bin/arm-linux-gcc -O -S -static loop.c -o loop.s
```

The resulting assembly code is shown in listing 7.17.

(a) Draw a dataflow diagram of the assembly code.
(b) Identify all instructions in this listing that are directly involved in the address calculation of a data memory read.

Listing 7.16 C program for Problem 7.7

```
int a[100]
int b[100];
int i;

for (i=0; i<100; ++i) {
  a[b[i]] = i + 2;
}
return 0;
```

Listing 7.17 Assembly listing for Problem 7.7

```
        mov     r1, #0
.L6:
        add     r0, sp, #800
        add     r3, r0, r1, asl #2
        ldr     r3, [r3, #-800]
```

```
    add     r2, r0, r3, asl #2
    add     r3, r1, #2
    str     r3, [r2, #-400]
    add     r1, r1, #1
    cmp     r1, #99
    ble     .L6
```

Problem 7.8. Listing 7.18 shows a routine to evaluate the CORDIC transformation (Coordinate Digital Transformation). CORDIC procedures are used to approximate trigonometric operations using simple integer arithmetic. In this case, we are interested in the inner loop of the program, on lines 16–28. This program will be compiled with a single level of optimization as follows:

```
arm-linux-gcc -O1 -g -c cordic.c
```

Next, the assembly code of the program is created using the `objdump` utility. The command line flags are chosen to generate the assembly code, interleaved with the C code. This is possible if the object code was generated using *debug information* (`-g` flag above). The resulting file is shown in Listing 7.19.

```
arm-linux-objdump -S -d cordic.o
```

(a) Study the Listing 7.19 and explain the difference between an `addgt` and an `addle` instruction on the ARM processor.
(b) Use `objdump` find the size of the `text` segment and the `data` segment.
(c) Study the Listing 7.19 and point out what are the *callee-saved* registers in this routine.
(d) Estimate the execution time for the cordic routine, ignoring the cache misses.

Listing 7.18 C listing for Problem 7.8

```
 1  #define AG_CONST 163008218
 2
 3  static const int angles[] = {
 4    210828714, 124459457,  65760959,  33381289,
 5     16755421,   8385878,   4193962,   2097109,
 6      1048570,    524287,    262143,    131071,
 7        65535,     32767,     16383,      8191,
 8         4095,      2047,      1024,       511 };
 9
10  void cordic(int target, int *rX, int *rY) {
11    int X, Y, T, current;
12    unsigned step;
13    X       = AG_CONST;
14    Y       = 0;
15    current = 0;
16    for(step=0; step < 20; step++) {
17      if (target > current) {
18        T          =  X - (Y >> step);
```

```
      Y          = (X >> step) + Y;
      X          = T;
      current   += angles[step];
    } else {
      T          = X + (Y >> step);
      Y          = -(X >> step) + Y;
      X          = T;
      current   -= angles[step];
    }
  }
  *rX = X;
  *rY = Y;
}
```

Listing 7.19 Mixed C-assembly listing for Problem 7.8

```
void cordic(int target, int *rX, int *rY) {
   0:   e92d40f0    stmdb   sp!, {r4, r5, r6, r7, lr}
   4:   e1a06001    mov r6, r1
   8:   e1a07002    mov r7, r2
  int X, Y, T, current;
  unsigned step;
  X       = AG_CONST;
   c:   e59fe054    ldr lr, [pc, #84]
  Y       = 0;
  10:   e3a02000    mov r2, #0  ; 0x0
  current = 0;
  14:   e1a01002    mov r1, r2
  for(step=0; step < 20; step++) {
  18:   e1a0c002    mov ip, r2
  1c:   e59f5048    ldr r5, [pc, #72]
  20:   e1a04005    mov r4, r5
    if (target > current) {
  24:   e1500001    cmp r0, r1
      T          =  X - (Y >> step);
  28:   c04e3c52    subgt   r3, lr, r2, asr ip
      Y          = (X >> step) + Y;
  2c:   c0822c5e    addgt   r2, r2, lr, asr ip
      X          = T;
  30:   c1a0e003    movgt   lr, r3
      current   += angles[step];
  34:   c795310c    ldrgt   r3, [r5, ip, lsl #2]
  38:   c0811003    addgt   r1, r1, r3
    } else {
      T          = X + (Y >> step);
  3c:   d08e3c52    addle   r3, lr, r2, asr ip
      Y          = -(X >> step) + Y;
  40:   d0422c5e    suble   r2, r2, lr, asr ip
      X          = T;
  44:   d1a0e003    movle   lr, r3
      current   -= angles[step];
  48:   d794310c    ldrle   r3, [r4, ip, lsl #2]
  4c:   d0631001    rsble   r1, r3, r1
  50:   e28cc001    add ip, ip, #1
  54:   e35c0013    cmp ip, #19
  58:   8586e000    strhi   lr, [r6]
    }
  }
  *rX = X;
  *rY = Y;
  5c:   85872000    strhi   r2, [r7]
}
  60:   88bd80f0    ldmhiia sp!, {r4, r5, r6, r7, pc}
  64:   ea000007    b   24 <cordic+0x24>
  68:   09b74eda    ldmeqib r7!, {r1, r3, r4, r6, r7, r9, sl,
                                    fp, lr}
  6c:   00000000    andeq   r0, r0, r0
```

Problem 7.9. We discussed the modulo-179 program in Listing 7.7, Sect. 7.4. We compiled the C program for two different processors, ARM and Microblaze, at the highest optimization level. The ARM assembly is shown in Listing 7.20, and the Microblaze assembly is shown in Listing 7.21. Study the assembly listings and answer the questions below.

(a) One of the assembly programs uses delayed branch instructions. Which processor is it?
(b) One of the assembly programs uses predicated instructions, which are instructions that execute conditionally based on the value of a processor flag. Which processor is it?
(c) Estimate the number of clock cycles required on each processor to compute the result of `modulo(54)`.

Listing 7.20 ARM assembly program for Problem 7.9

```
modk:
    mov       ip, #1
    mov       r2, ip, asl r1
    sub       r1, r2, #1
    and       r0, r0, r1
    mov       pc, lr
divk:
    mov       r0, r0, lsr r1
    mov       pc, lr
modulo:
    mov       r2, r0, lsr #8
    mov       r1, #77
    and       r0, r0, #255
.L9:
    mul       r3, r1, r2
    and       ip, r3, #255
    movs      r2, r3, lsr #8
    add       r0, r0, ip
    bne       .L9
    mov       r2, r0, lsr #8
    cmp      r2, #0
    and      r0, r0, #255
    bne      .L9
    cmp      r0, #179
    subcs    r0, r0, #179
    mov      pc, lr
```

Listing 7.21 Microblaze assembly program for Problem 8.9

```
modk:
    addik    r3,r0,1 # 0x1
    bsll     r3,r3,r6
    addik    r3,r3,-1
    rtsd     r15,8
    and      r3,r3,r5
divk:
    rtsd     r15,8
    bsrl     r3,r5,r6
modulo:
    bsrli    r4,r5,8
    andi     r5,r5,255  #and1
$L18:
    muli     r3,r4,77
    bsrli    r4,r3,8
    andi     r3,r3,255 #and1
    bneid    r4,$L18
    addk     r5,r5,r3
    bsrli    r4,r5,8
    bneid    r4,$L18
    andi     r5,r5,255 #and1
    addi     r18,r0,178
    cmpu     r18,r5,r18
    bgeid    r18,$L20
    addk     r3,r5,r0
    addik    r3,r5,-179
$L20:
    rtsd     r15, 8
    nop
```

Chapter 8
System on Chip

8.1 The System-on-Chip Concept

Figure 8.1 illustrates a typical System-on-chip. It combines several components on a bus system. One of these components is a microprocessor (typically a RISC) which acts as a central controller in the SoC. Other components include on-chip memory, off-chip-memory interfaces, dedicated peripherals, hardware coprocessors, and component-to-component communication links.

The application domain greatly affects the type of hardware peripherals, the size of memories and the nature of on-chip communications. A particular configuration of these elements is called a *platform*. Just like a personal computer is a platform for general-purpose computing, a system-on-chip is a platform for domain-specialized computing, i.e. for an ensemble of applications that are typical for a given application domain. Examples of application domains are mobile telephony, video processing, or high-speed networking. The set of applications in the video-processing domain for example could include image transcoding, image compression and decompression, image color transformations, and so forth. Domain specialization in a System-on-Chip is advantageous for several reasons.

- The *specialization* of the platform ensures that its processing efficiency is higher compared to that of general-purpose solutions. Increased processing efficiency means lower power consumption (longer battery lifetime) or higher absolute performance.
- The *flexibility* of the platform ensures that it is a reusable solution that works over multiple applications. As a result, the design cost per-application decreases, applications can be developed faster, and the SoC itself becomes cheaper because it can be manufactured for a larger market.

P.R. Schaumont, *A Practical Introduction to Hardware/Software Codesign*,
DOI 10.1007/978-1-4614-3737-6_8, © Springer Science+Business Media New York 2013

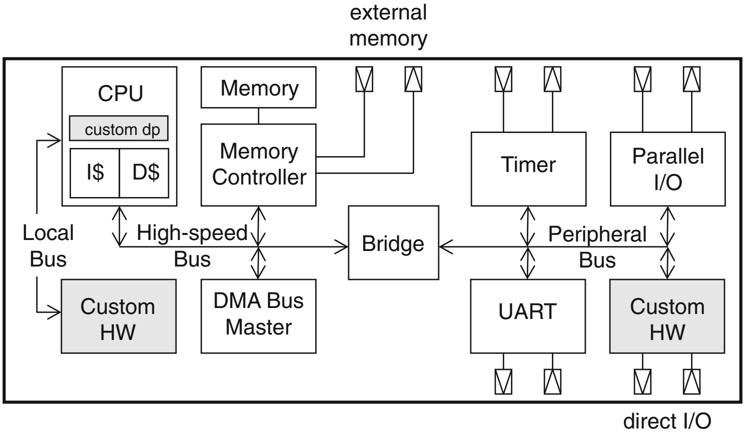


Fig. 8.1 Generic template for a system-on-chip

8.1.1 The Cast of Players

An architecture such as in Fig. 8.1 can be analyzed along four orthogonal dimensions: control, communication, computation, and storage. The role of central controller is given to the microprocessor, who is responsible of issuing control signals to, and collecting status signals from, the various components in the system. The microprocessor may or may not have a local instruction memory. In case it does not have a local instruction memory, caches may be utilized to improve instruction-memory bandwidth. The `I$` and `D$` symbols in Fig. 8.1 represent the instruction- and data-caches in the microprocessor. In the context of the SoC architecture, these caches will only be of use to the software executing on the microprocessor. This obvious fact has an often-ignored consequence: whenever the microprocessor needs to interact with a peripheral, the data moves outside of the cache. The memory-hierarchy, which makes the CPU so fast and powerful, does not work for data-movement. A microprocessor in an SoC is therefore useful as central controller, but it is not very efficient to carry data around in the system. We will come back to this point in detail further in this Chapter.

The SoC implements communication using system-wide buses. Each bus is a bundle of signals including address, data, control, and synchronization signals. The data transfers on a bus are expressed as read- and write-operations with a particular memory address. The bus control lines indicate the nature of the transfer (read/write, size, source, destination), while the synchronization signals ensure that the sender and receiver on the bus are aligned in time during a data transfer. Each component connected to a bus will respond to a particular range of memory addresses. The ensemble of components can thus be represented in an *address map*, a list of all relevant system-bus addresses.

It is common to split SoC buses into segments. Each segment connects a limited number of components, grouped according to their communication needs.

In the example, a high-speed communication bus is used to interconnect the microprocessor, a high-speed memory interface, and a Direct Memory Access (DMA) controller. A DMA is a device specialized in performing block-transfers on the bus, for example to copy one memory region to another. Next to a high-speed communication bus, you may also find a peripheral bus, intended for lower-speed components such as a timer and input-output peripherals. Segmented buses are interconnected with a *bus bridge*, a component that translates bus transfers from one segment to another segment. A bus bridge will only selectively translate transfers from one bus segment to the other. This selection is done based on the address map.

The bus control lines of each bus segment are under command of the *bus master*, the component that decides the nature of a given bus transfer. The *bus slaves* will follow the directions of the bus master. Each bus segment can contain one or more bus masters. In case there are multiple masters, the identity of the bus master can be multiplexed among bus-master components at run time. In that case, a *bus arbiter* will be needed to decide which component can become a bus master for a given bus transfer. A bus bridge can be either a master or a slave, depending on the direction of the transfers. For example, when going from the high-speed bus to the peripheral bus, the bus bridge will act as a bus slave on the high-speed bus and as a bus master on the peripheral bus. Each of the transfers on the high-speed bus and on the peripheral bus will be handled independently. Therefore the segmentation of buses using bus bridges leads to a dual advantage. First, bus segments can group components with matching read- and write-speed together, thus providing optimal usage of the available bus bandwidth. Second, the bus segments enable communication parallelism.

8.1.2 SoC Interfaces for Custom Hardware

Let's consider the opportunities to attach custom hardware modules in the context of an SoC architecture. In the context of this chapter, a 'custom hardware module' means a dedicated digital machine in the form of an FSMD or a micro-programmed machine. Eventually, all custom hardware will be under control of the central processor in the SoC. The SoC architecture offers several possible hardware-software interfaces to attach custom hardware modules. Three approaches can be distinguished in Fig. 8.1 as shaded blocks.

- The most general approach is to integrate a custom hardware module as a standard peripheral on a system bus. The microprocessor communicates with the custom hardware module by means of read/write memory accesses. Of course, the memory addresses occupied by the custom hardware module cannot be used for other purposes (i.e. as addressable memory). For the memory addresses occupied by the custom hardware module, the microprocessors' cache has no meaning, and the caching effect is unwanted. Microcontroller chips with many different peripherals typically use this memory-mapped strategy to attach peripherals. The strong point of this approach is that a universal communication mechanism (memory read/write operations) can be used for a wide range of custom hardware modules. The corresponding disadvantage, of course, is that such a bus-based approach to integrate hardware is not very scalable in terms of performance: the system bus quickly becomes a bottleneck when intense communication between a microprocessor and the attached hardware modules is needed.

- A second mechanism is to attach custom hardware through a local bus system or coprocessor interface provided by the microprocessor. In this case, the communication between the hardware module and the microprocessor will follow a dedicated protocol, defined by the local bus system or coprocessor interface. In comparison to system-bus interfaces, coprocessor interfaces have a high-bandwidth and a low latency. The microprocessor may also provide a dedicated set of instructions to communicate over this interface. Typical coprocessor interfaces do not involve a memory addresses. This type of coprocessor obviously requires a microprocessor with a coprocessor- or local-bus interface.
- Microprocessors may also provide a means to integrate a custom-hardware datapath inside of the microprocessor. The instruction set of the microprocessor is then extended with additional, new instructions to drive this custom hardware. The communication channel between the custom data-path and the processor is typically through the processor register file, resulting in a very high communication bandwidth. However, the very tight integration of custom hardware with a microprocessor also means that the traditional bottlenecks of the microprocessor are a bottleneck for the custom-hardware modules as well. If the microprocessor is stalled because of external events (such as memory-access bandwidth), the custom data-datapath is stalled also.

These observations show that, in the context of SoC, there is no single *best* way to integrate hardware and software. There are many possible solutions, each with their advantages and disadvantages. Selecting the right approach involves trading-off many factors, including the required communication bandwidth, the design complexity of the custom hardware interface, the software, the available design time, and the overall cost budget. The following chapters will cover some of the design aspects of hardware-software interfaces. In the end, however, it is the hardware-software codesigner who must identify the integration opportunities of a given System-on-Chip architecture, and who must realize their potential.

8.2 Four Design Principles in SoC Architecture

A SoC is specific to a given application domain. Are there any guiding design principles that are relevant to the design of *any* SoC? This section addresses this question in more detail. The objective is to clarify four design principles that govern the majority of modern SoC architectures. These four principles include heterogeneous and distributed communications, heterogeneous and distributed data processing, heterogeneous and distributed storage, and hierarchical control. We will review each of these four points in detail. This will demonstrate the huge potential of the SoC, and in particular their technological advantage over non-integrated circuits.

8.2.1 Heterogeneous and Distributed Data Processing

A first prominent characteristic of an SoC architecture is heterogeneous and distributed data processing. An SoC may contain multiple independent (distributed) computational units. Moreover, these units can be heterogenous. They can include FSMD, micro-programmed engines, or microprocessors.

One can distinguish three forms of data-processing parallelism. The first is word-level parallelism, which enables the parallel processing of multiple bits in a word. The second is operation-level parallelism, which allows multiple instructions to be executed simultaneously. The third is task-level parallelism, which allows multiple independent threads of control to be executing independently. Word-level parallelism and operation-level parallelism are available on all of the machine architectures we discussed so far: FSMD, Micro-programmed machines, RISC, and also SoC. However, only an SoC supports true task-level parallelism. Note that multi-threading in a RISC is not task-level parallelism; it is task-level concurrency on top of a sequential machine (See Sect. 1.7).

Each of the computational units in an SoC can be specialized to a particular function. The overall SoC therefore includes a collection of *heterogeneous* computational units. For example, a digital signal processing chip in a camera may contain specialized units to perform image-processing. Computational specialization is a key ingredient to high-performance. In addition, the presence of all forms of parallelism (word-level, operation-level, task-level) ensures that an SoC can fully exploit the technology.

In fact, integrated circuit technology is *extremely* effective to provide computational parallelism. Consider the following numerical example. A 1-bit full-adder cell can be implemented in about 28 transistors. The Intel Core 2 processor contains 291 million transistors in 65 nm CMOS technology. This is sufficient to implement 325,000 32-bit adders. Assuming a core clock frequency of 2.1 GHz, we thus find that the silicon used to create a Core 2 can theoretically implement 682,000 GOPS. We call this number the *intrinsic computational efficiency* of silicon. Of course, we don't know how to build a machine that would have this efficiency, let alone that such a machine would be able to cope with the resulting power density. The intrinsic computational efficiency merely represents an upperbound for the most efficient use of silicon real-estate.

$$Eff_{intrinsic} = \frac{291.10^6}{28.32}.2.1 \approx 682{,}000\ Gops \tag{8.1}$$

The actual Core2 architecture handles around 9.24 instructions per clock cycle, in a single core and in the most optimal case. The actual efficiency of the 2.1 GHz Core 2 therefore is 19.4 GOPS. We make the (strong) approximation that these 9.24 instructions each correspond to a 32-bit addition, and call the resulting throughput the actual Core2 efficiency. The ratio of the intrinsic Core2 efficiency over the actual Core2 efficiency illustrates the efficiency of silicon technology compared to the efficiency of a processor core architectures.

$$Efficiency = \frac{Eff_{intrinsic}}{Eff_{actual}} \approx \frac{682{,}000}{19.4} = 35{,}150 \tag{8.2}$$

Therefore, bare silicon can implement computations 35,000 times more efficient than a Core2! While this is a very simple and crude approximation, it demonstrates why specialization of silicon using multiple, independent computational units is so attractive.

8.2.2 Heterogeneous and Distributed Communications

The central bus in a system-on-chip is a critical resource. It is shared by many components in an SoC. One approach to prevent this resource from becoming a bottleneck is to split the bus into multiple bus segments using bus bridges. The bus bridge is a mechanism to create distributed on-chip communications. The on-chip communication requirements typically show large variations over an SoC. Therefore, the SoC interconnection mechanisms should be heterogeneous as well. There may be shared busses, point-to-point connections, serial connections and parallel connections.

Heterogeneous and distributed SoC communications enable a designer to exploit the on-chip communication bandwidth. In modern technology, this bandwidth is extremely high. An illustrative example by Chris Rowen mentions the following numbers. In a 90 nm six-layer metal processor, we can reasonably assume that metal layers can be used as follows (Fig. 8.2): two metal layers are used for power and ground, two metal layers are used to route wires in the X direction, and two metal layers are used to route wires in the Y direction. The density of wires in a 90 nm process is 4 wires per micron (a micron is one thousandth of a millimeter), and the bit frequency is at 500 MHz. Consequently, in a chip of 10 mm on the side, we will have 40,000 wires per layer on a side. Such a chip can shuffle and transport 80,000 bits at a frequency of 500 MHz. This corresponds to 40 Tbits per second! Consequently, on-chip communications have a high bandwidth – the real challenge is how to organize it efficiently.

The same efficiency is not available for *off-chip* communication bandwidth. In fact, off-chip bandwidth is very expensive compared to on-chip bandwidth. For example, consider the latest Hypertransport 3.1 standard, a serial link developed for high-speed processor interconnect. Usually, a (high-end) processor will have one to

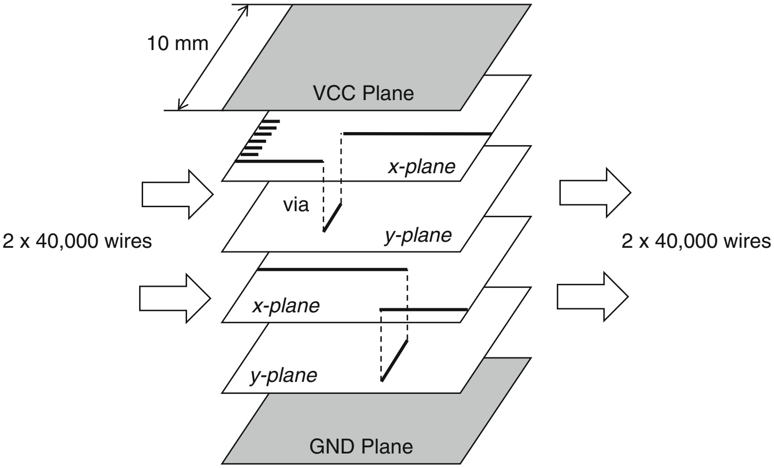


Fig. 8.2 Demonstration of the routing density in a six-layer metal 90 nm CMOS chip

four of such ports. The maximum aggregate data bandwidth for such a port is around 51.2 Gb per second. Thus we will find less than 204.8 Gb per second input/output bandwidth on a state-of-the-art processor today. That is still 195 times less than the 40 Tb/s on-chip bandwidth in an (older) 90 nm CMOS process! On-chip integration therefore allows very cheap, high-speed interconnect between components. Of course, a similar remark as with the intrinsic computational efficiency applies: the 40 Tb/s number is an ideal estimate, an upperbound. We have no means to build an operational chip with this bandwidth: signal crosstalk, and power density (each wire needs a driver) will prevent this.

8.2.3 Heterogeneous and Distributed Storage

A third characteristic of System-on-Chip architectures is a distributed and heterogeneous storage architecture. Instead of a single, central memory, an SoC will use a collection of dedicated memories. Processors and micro-coded engines may contain local instruction memories. Processors may also use cache memories to maintain local copies of data and instructions. Coprocessors and other active components will use local register files. Specialized accelerators can use dedicated memories for specific applications such as for video frame buffering, or as local scratchpad.

This storage is implemented with a collection of different memory technologies. There are five broad categories of silicon-based storage available today.

- *Registers* are the fastest type of memory available. Registers are also called *foreground memory*. They reside the closest to the computation elements of an architecture. A register does not have the concept of addressing unless it is organized in a *register file*.

- *Dynamic Random Access Memory* (DRAM) provides cheap storage at very high densities. Today (2009), one in four memory chips sold is a DRAM (or a related category such as SDRAM, DDR, DDR2). DRAM, and all the following categories are called *background memory*. Unlike registers, DRAM cells need a high-density circuit style. DRAM silicon processing is very different than the processing used for logic. Therefore, DRAM memories cannot be integrated on the same die as the SoC logic. DRAM memories usually come in a separate package. More advanced packaging technologies (stacked-die, multi-chip packaging) may still be able to produce a single physical package that contains processing as well as DRAM.
- *Static Random Access Memory* (SRAM) is used where fast read-write storage is required. SRAM has lower density and higher power consumption than DRAM. It is not used for the bulk of computer storage, but rather for specialized tasks such as caches, video buffers, and so on. On the plus side, SRAM can be implemented with the same process technology as normal logic gates. It is therefore easy to mix SRAM and computational elements in an SoC.
- *Non-volatile Read-Only Memory* (NVROM) is used for applications that only require read access on a memory, such as for example to store the instructions of a program. Non-volatile memories have a higher density than SRAM. There is a range of technologies that can be used to implement a NVROM (maskprogrammed ROM, PROM, EPROM, EEPROM).
- *Non-volatile Random Access Memory* (NVRAM) is used for applications that need read-write memories that do not loose data when power is removed. The read- and write-speed in a NVRAM can be asymmetrical (write being slower), so that in the limit the distinction between NVROM and NVRAM is not sharp.

Table 8.1 summarizes the key characteristics of these different types of memory. The entries in the table have the following meaning.

- The *cell size* is the silicon area required to store a single bit. Storage cells are only one aspect of a complete memory – additional hardware is needed for address decoding, multiplexing bits from the data bus, and so on. High-density storage technologies use only a single transistor per bit, and make use of low-level physical properties of that transistor (parasitic capacitance, floating gate, etc.) to hold the bit.
- The *retention time* expresses how long a bit can be held in a non-powered memory.
- The *addressing* mechanism shows how bits are retrieved from memory. In multiplexed addressing, such as used by DRAM, the address is cut in two parts which are provided sequentially to the memory.
- The *access time* is the time required to fetch a data element from memory. Note that the access time is a coarse number, as it does not capture the detailed behavior of a memory. For example, in NVRAM technologies, the read and write access time is asymmetrical: write takes longer than read. Modern DRAM

Table 8.1 Types of memories

Type	Register Register file	DRAM	SRAM	NVROM (ROM, PROM, EPROM)	NVRAM (Flash, EEPROM)
Cell size (bit)	10 transistors	1 transistor	4 transistors	1 transistor	1 transistor
Retention	0	Tens of ms	0	∞	10 years
Addressing	Implicit	Multiplexed	Non-muxed	Non-muxed	Non-muxed
Access time	Less then 1ns	Less then 20 ns	Less then 10 ns	20 ns	20 ns (read) 100 µs (write)
Power consumption	High	Low	High	Very low	Very low
Write durability	∞	∞	∞	1-time	One million times

memories are very efficient in accessing consecutive memory locations (burst access), but individual random locations take longer. Finally, modern memories can be internally pipelined, such that they can process more than one read or write command at the same time.

- The *power consumption* is a qualitative appreciation for the power consumption of a memory, as measured per access and per bit. Fast read/write storage is much more power-hungry than slow read-only storage.

The presence of distributed storage significantly complicates the concept of a centralized memory address space, which is so useful in SoC. As long as the data within these distributed memories is local to a single component, this does not cause any problem. However, it becomes troublesome when data needs to be shared among components. First, when multiple copies of a single data item exist in different memories, all these copies need to be kept consistent. Second, updating of a shared data item needs to be implemented in a way that will not violate data dependencies among the components that share the data item. It is easy to find a few examples where either of these two requirements will be violated (see Problem 8.1).

In 1994, Wulf and McKee wrote a paper entitled *Hitting the Memory Wall: Implications of the Obvious'*. The authors used the term *memory wall* to indicate the point at which the performance of a (computer) system is determined by the speed of memory, and is no longer dependent on processor speed. While the authors' conclusions were made for general-purpose computing architectures, their insights are also valid for mixed hardware/software systems such as those found in System-on-Chip. Wulf and McKee observed that the performance improvement of processors over time was higher than the performance improvement of memories. They assumed 60% performance improvement per year for processors, and 10% for memories – valid numbers for systems around the turn of the century. The memory wall describes the specific moment when the performance of a computer system becomes performance-constrained by the memory subsystem. It is derived as follows.

In general-purpose computer architectures with a cache, the memory access time of a processor with cycle time t_c, cache hit-rate p, and memory access time t_m, is given by

$$t_{avg} = p \times t_c + (1 - p) \times t_m \tag{8.3}$$

Assume that on the average, one in five processor instructions will require a memory reference. Under this assumption, the system becomes memory-access constrained when t_{avg} reaches five times the cycle time t_c. Indeed, no matter how good the processor is, it will spend more time waiting for memory than executing instructions. This point is called the memory wall.

How likely is it for a computer to hit the memory wall? To answer this question, we should observe that the cache hit rate p cannot be 100%. Data-elements stored in memory have to be fetched at least once from memory before they can be stored in a cache. Let us assume a factor of $p = 0.99$ (rather pessimistic), a cycle time t_c of 1, and a cycle time t_m of 10.

Under this assumption,

$$(t_{avg})_{now} = 0.99 \times 1 + 0.01 \times 10 = 1.09 \tag{8.4}$$

One year from now, memory is 1.1 times faster and the processor is 1.6 times faster. Thus, 1 year from now, t_{avg} will change to

$$(t_{avg})_{now+1} = 0.99 \times 1 + 0.01 \times 10 \times \frac{1.6}{.1} = 1.135 \tag{8.5}$$

And after N years it will be

$$(t_{avg})_{now+N} = 0.99 \times 1 + 0.01 \times 10 \times \frac{1.6^N}{1.1} \tag{8.6}$$

The memory wall will be hit when t_{avg} equals 5, which can be solved according to the above equation to be $N = 9.8$ years. Now, more than a decade after this 1994 paper, it is unclear if the doomsday scenario has really materialized. Many other factors have changed in the meantime, making the formula an unreliable predictor. For example, current processor workloads are very different than those from 10 years ago. Multimedia, gaming and internet have become a major factor. In addition, current processor scaling is no longer done by increasing clock frequency but by more drastic changes at the architecture-level (multiprocessors). Finally, new limiting factors, such as power consumption density and technological variability, have changed the quest for performance into one for efficiency.

Despite this, memory remains a crucial element in SoC design, and it still has a major impact on system performance. In many applications, the selection of memory elements, their configuration and layout, and their programming is a crucial design task.

8.2.4 Hierarchical Control

The final concept in the architecture of an SoC is the hierarchy of control among components. A hierarchy of control means that the entire SoC operates as a single logical entity. This implies that the components in an SoC receive commands from a central point of control. They can operate loosely, almost independently, from one another, but at some point they need to synchronize and report back to the central point of control. For example, consider a C program (on a RISC) that uses a co-processor implemented as a peripheral. The C program needs to send arguments to the coprocessor, wait for the co-processor to finish execution, and finally retrieve the result from the co-processor. From the perspective of the co-processor, the custom hardware will first wait for operands from the peripheral bus, next it will process them, and finally it will signal completion of the operation (for example, by setting a status flag). The software on the RISC processor and the activities on the co-processor therefore are not independent. The local controller in the co-processor can be developed using an FSM or a micro-programming technique. The RISC processor will maintain overall control in the system, and distribute commands to custom hardware.

The design of a good control hierarchy is a challenging problem. On the one hand, it should exploit the distributed nature of the SoC as good as possible – this implies doing many things in parallel. On the other hand, it should minimize the number of conflicts that arise as a result of running things in parallel. Such conflicts can be the result of overloading the available bus-system or memory bandwidth, or of over-scheduling a coprocessor. Due to the control hierarchy, all components of an SoC are logically connected to each other, and each of them may cause a system bottleneck. The challenge for the SoC designer (or platform programmer) is to be aware of the location of such system bottlenecks, and to control them.

8.3 Example: Portable Multimedia System

In this section we illustrate the four key characteristics discussed earlier (distributed and heterogeneous memory, interconnect, computing, and hierarchical control) by means of an example. Figure 8.3 shows the block diagram of a digital media processor by Texas Instruments. The chip is used for the processing of still images, video, and audio in portable, battery-operated devices. It is manufactured in 130 nm CMOS, and the entire chip consumes no more than 250 mW in default-preview mode, and 400 mW when video encoding and decoding is operational.

The chip supports a number of device modes. Each mode corresponds to typical user activity. The modes include the following.

- Live preview of images (coming from the CMOS imager) on the video display.
- Live-video conversion to a compressed format (MPEG, MJPEG) and streaming of the result into an external memory.

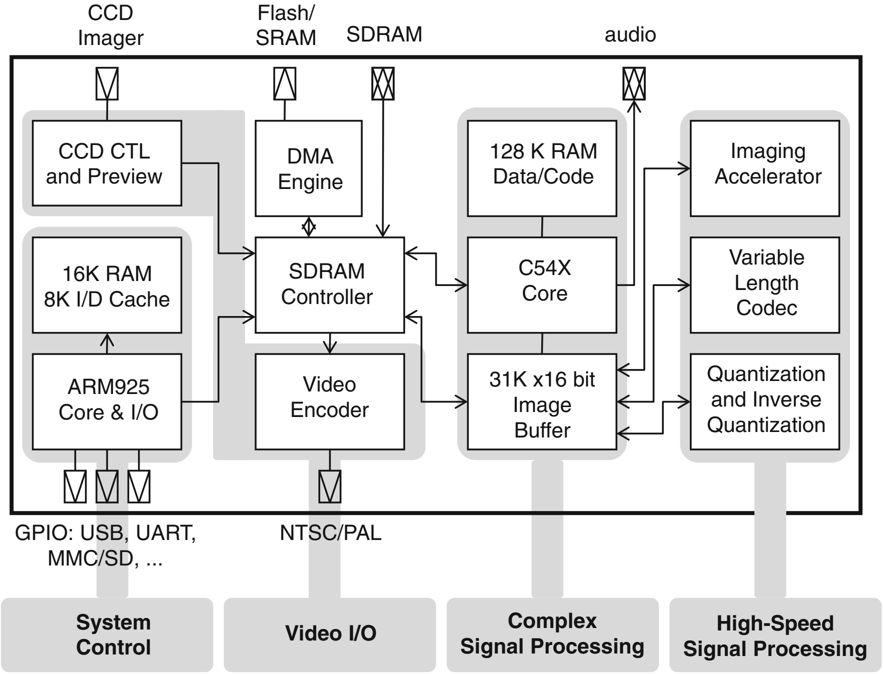


Fig. 8.3 Block diagram of portable multi-media system

- Still-image capturing of a high-resolution image and conversion to JPEG.
- Live audio capturing and audio compression to MP3, WMA, or AAC.
- Video decode and playback of a recorded stream onto the video display.
- Still image decode and playback of a stored image onto the video display.
- Audio decode and playback.
- Photo printing of a stored image into a format suitable for a photo printer.

The central component of the block diagram in Fig. 8.3 is the SDRAM memory controller. During operation, image data is stored in off-chip SDRAM memory. The SDRAM controller organizes memory traffic to this large off-chip memory. Around the memory controller, four different subsystems are organized. They deal with video input/output, complex signal processing, high-speed signal processing, and system control, respectively.

The video input/output subsystem includes the CCD sensor interface, and a video encoder. The CCD interface is capable of sampling up to 40 MHz at 12 bits per pixel, and it needs to provide high-resolution still images (2–5 Mpixels) as well as moving images (up to 30 frames/s at 640 × 480 pixels). Most CCD sensors record only a single color per pixel. Typically there are 25% red pixels, 25% blue pixels and 50% green pixels, which are arranged in a so-called Bayer pattern. Before images can be processed, the missing pixels need to be filled in (interpolated). This task is a typical example of streaming and dedicated processing. The video subsystem also contains a video encoder, capable of merging two video streams on screen, and capable of providing picture-in-picture functionality. The video coder also includes on-screen menu subsystem functionality. The output of the video coder goes to an attached LCD or a TV. The video coder computes around 100 operations per pixel, while the power budget of the entire video subsystem is less than 100 mW. Hence, we require an energy efficiency on the order of 40 GOPS per W (100 operations per pixel at 40 MHz pixel rate for 100 mW). This energy efficiency is out of range for embedded software on contemporary processors.

The complex signal-processing subsystem is created on top of a C54x digital signal processor (DSP) with 128 KB of RAM and operating at 72 MHz. The DSP processor performs the main processing and control logic for the wide range of signal processing algorithms that the device has to perform (MPEG-1, MPEG-2, MPEG-4, WMV, H.263, H.264, JPEG, JPEG2K, M-JPEG, MP3, AAC, WMA).

The third subsystem is the high-speed signal processing subsystem, needed for encoding and decoding of moving images. Three coprocessors deliver additional computing muscle for the cases where the DSP falls short. There is a DMA engine that helps moving data back and forth between the memory attached to the DSP and the coprocessors. The three coprocessors implement the following functions. The first one is a SIMD-type of coprocessor to provide vector-processing for image processing algorithms. The second is a quantization coprocessor to perform quantization in image encoding algorithms. The third coprocessor performs Huffman encoding for those image encoding standards. The coprocessor subsystem increases to overall processing parallelism of the chip, as they can work concurrently with the DSP processor. This allows the system clock to be decreased.

Finally, the system ARM subsystem is the overall system manager. It synchronizes and controls the different subcomponents of the system. It also provides interfaces for data input/output, and user interface support.

Each of the four properties discussed in the previous section can be identified in this chip.

- The SoC contains *heterogeneous and distributed processing*. There is hardwired processing (video subsystem), signal processing (DSP), and general-purpose processing on an ARM processor. All of this processing can have overlapped activity.
- The SoC contains *heterogeneous and distributed interconnect*. Instead of a single central bus, there is a central 'switchbox' that multiplexes accesses to the off-chip memory. Where needed, additional dedicated interconnections are implemented. Some examples of dedicated interconnections include the bus between the DSP and its instruction memory, the bus between the ARM and its instruction memory, and the bus between the coprocessors and their image buffers.
- The SoC contains *heterogeneous and distributed storage*. The bulk of the memory is contained within an off-chip SDRAM module, but there are also dedicated instruction memories attached to the TI DSP and the ARM, and there are dedicated data memories acting as small dedicated buffers.

- Finally, there is a hierarchy of control to ensures that the overall parallelism in the architecture is optimal. The ARM will start/stop components and control data streams depending on the mode of the device.

The DM310 chip is an excellent example of the effort it takes to support real-time video and audio in a portable device. The architects (hardware and software people) of this chip have worked closely together to come up with the right balance between flexibility and energy-efficiency.

8.4 SoC Modeling in GEZEL

In the last section of this chapter, we consider how a System-on-Chip can be modeled in GEZEL, building on our previous experience with FSMD design, micro-programmed design, and general-purpose processors. GEZEL models and simulates microprocessors as well as SoC hardware. A typical example configuration is shown in Fig. 8.4. It includes several components. Custom hardware modules are captured as FSMD models. The microprocessor cores are captured as custom library modules, called `ipblock`. Each microprocessor core offers different types of interfaces. Each of these interfaces is captured using a different `ipblock`. The software executed by the microprocessor core is developed in C or assembly, and converted to binary format using a cross-compiler or cross-assembler. The binary is used to initialize the instruction-set simulator, embedded in an `ipblock`. The entire system simulation is executed by the GEZEL platform simulator `gplatform`.

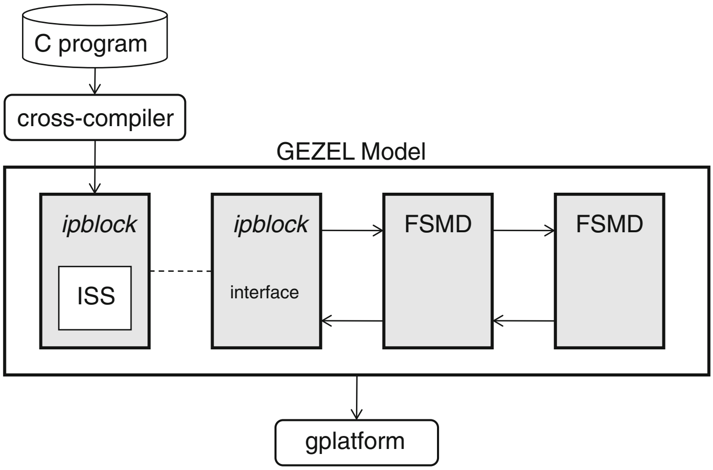


Fig. 8.4 A GEZEL system-on-chip model

Listing 8.1 A GEZEL top-level module with a single ARM core

```
1  ipblock myarm {
2    iptype "armsystem";
3    ipparm "exec = hello";
4  }
5
6  system S {
7    myarm;
8  }
```

8.4.1 An SoC with a StrongARM Core

We describe some of the features of SoC modeling using the cores included in the `gplatform` simulator. The first core is a StrongARM core, modeled with the Simit-ARM v2.1 simulator developed by W. Qin at Boston University. Listing 8.1 shows a simple, standalone ARM core. Line 2 of this listing tells that this module is an ARM core with attached instruction memory (`armsystem`). Line 3 names the ELF executable that must be loaded into the ARM simulator when the simulation starts. The syntax of an `ipblock` is generic and is used for many different types of cosimulation entities.

The model shown in Listing 8.1 is not very exciting since it does not show any interaction between hardware and software. We will extended this model with a *memory-mapped* interface, as shown in Listing 8.2. Figure 8.5 illustrates how this model corresponds to a System-on-Chip architecture. In Listing 8.2, a hardware-to-software interface is defined on lines 6–10. This particular example shows a memory-mapped interface. The interface has a single output port `data`. The port can be thought of as a register that is written by the software. The software can update the value of the register by writing to memory address `0x80000000`. After each update, the output port `data` will hold this value until the software writes to the register again. Note that the association between the memory-mapped interface and the ARM core is established using the name of the core (line 8 in Listing 8.2). Lines 12–28 show a custom hardware module, modeled as an FSMD, which is attached to this memory-mapped interface. The FSM uses the least-significant bit from the memory-mapped register as a state transition condition. Whenever this bit changes from 0 to 1, the FSMD will print the value of the memory-mapped register.

To cosimulate this model, we proceed as follows. First, we cross-compile a C program to run on the ARM. Next, we execute the cosimulation. The following is an example C program that we will run on top of this system architecture.

```
#include <stdio.h>

int main() {
  int y;
  volatile int * a = (int *) 0x80000000;

  *a = 25;
```

Listing 8.2 A GEZEL module with an ARM core and a memory-mapped interface on the ARM

```
 1  ipblock myarm {
 2    iptype "armsystem";
 3    ipparm "exec=hello";
 4  }
 5
 6  ipblock port1(out data    : ns(32)) {
 7    iptype "armsystemsource";
 8    ipparm "core=myarm";
 9    ipparm "address = 0x80000000";
10  }
11
12  dp portreader {
13    sig data : ns(32);
14    use myarm;
15    use port1(data);
16    reg changed : ns(1);
17    always   { changed = data[0]; }
18    sfg show { $display($cycle,": The MM interface is now ",
                                $dec, data); }
19    sfg nil  { }
20  }
21  fsm f_portreader(portreader) {
22    initial s0;
23    state   s1;
24    @s0 if (~changed) then (nil)   -> s0;
25                      else (show)  -> s1;
26    @s1 if (changed)  then (nil)   -> s1;
27                      else (nil)   -> s0;
28  }
29
30  system S {
31    portreader;
32  }
```

```
  *a = 0;
  *a = 39;
  *a = 0;

  return 0;
}
```

This program creates a pointer to the absolute memory address `0x80000000`, which corresponds to the memory-mapped port of the custom hardware module in Listing 8.2. The C program then writes a sequence of values to this address. The nonzero values will trigger the `$display` statement shown on line 18 of Listing 8.2. Compilation of this program, and execution of the cosimulation, is done through the following commands.

```
> arm-linux-gcc -static  hello.c -o hello
```

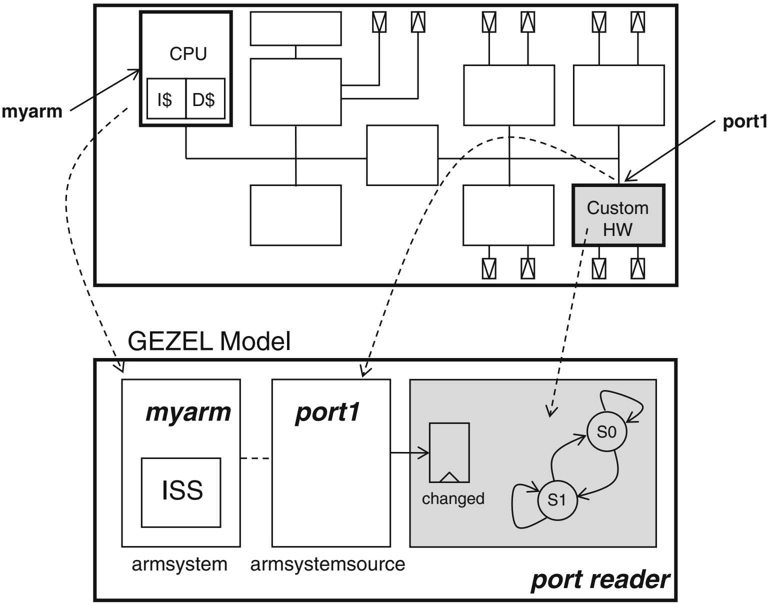


Fig. 8.5 Correspondence of Listing 8.2 to SoC architecture

```
> gplatform  armex.fdl
core myarm
armsystem: loading executable [hello]
7063: The MM interface is now 25
7069: The MM interface is now 39
Total Cycles: 7595
```

The cosimulation verifies that data is passed correctly from software to hardware. The first print statement executes at cycle 7063. This startup delay is used to set up the C runtime environment on the ARM; changing the C runtime environment to a faster, leaner library may reduce this delay significantly.

The relation between the GEZEL model and the System-on-Chip architecture, as illustrated in Fig. 8.5, shows that the FSMD captures the internals of a shaded 'custom hardware' module in a System-on-Chip architecture. The memory-mapped register captured by `port1` is located at the input of this custom hardware module. Thus, the GEZEL model in Listing 8.2 does not capture the bus infrastructure (the peripheral bus, the bus bridge, the high-speed bus) of the SoC. This has an advantage as well as a disadvantage. On the plus side, the resulting simulation model is easy to build, and will have a high simulation speed. On the down side, the resulting simulation does not capture the bus conflicts that occur in the real SoC architecture, and therefore the simulation results may show a difference with the real chip. Ultimately, the choice of modeling accuracy is with the designer. A more detailed GEZEL model could capture the transactions on an SoC bus as well, but this would cost an additional effort, and the resulting model may simulate at a lower

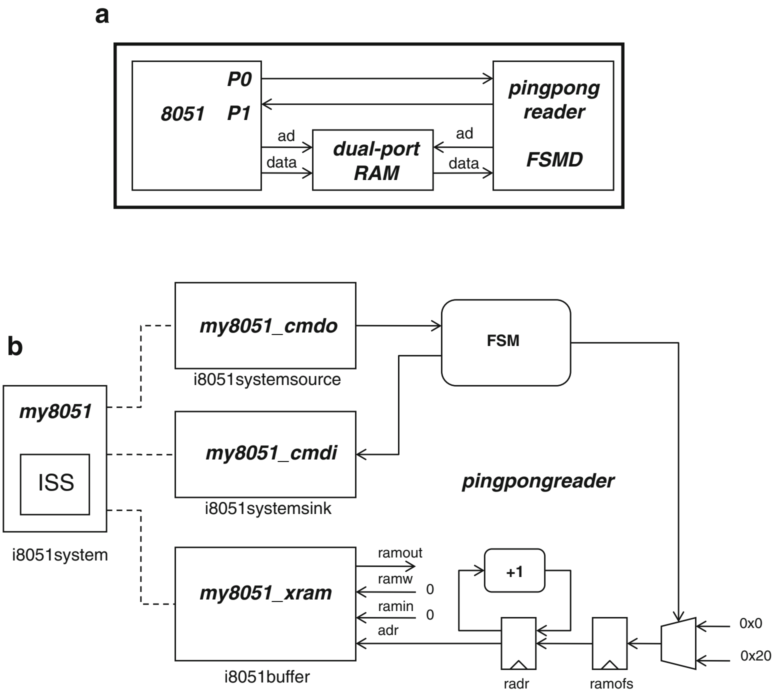


Fig. 8.6 (**a**) 8051 microcontroller with a coprocessor; (**b**) Corresponding GEZEL model structure

speed. For a cosimulation that focuses on verifying the hardware/software modules, a model such as shown on Listing 8.2 is adequate.

8.4.2 *Ping-Pong Buffer with an 8051*

As a second example, we show how an 8051 microcontroller core can be cosimulated in a GEZEL system model. Figure 8.6a shows a system with an 8-bit 8051 micro-controller, a dual-port RAM with 64 locations, and a hardware module. The micro-controller, as well as the hardware module, can access the RAM. The 8051 microcontroller has several 8-bit I/O ports, and two of them are used in this design. Port P0 is used to send a data byte to the hardware, while port P1 is used to retrieve a data byte from the hardware.

The idea of this design is a *ping-pong buffer*. The RAM is split up in two sections of 32 locations each. When the 8051 controller is writing into the lower section of the RAM, the hardware will read out the upper section of the RAM. Next, the 8051 will switch to the higher section of the RAM, while the hardware module will scan out the lower section of the RAM. This double-buffering technique is frequently used to emulate a dual-port shared RAM with single-port RAM modules. Switching between the two operational modes of the system is implemented using a two-way handshake between the 8051 controller and the hardware. The two ports on the 8051 are used for this purpose.

Figure 8.6b and Listing 8.3 show the GEZEL design that implements this model. The 8051 microcontroller is captured with three different `ipblock`: one for the microcontroller (lines 1–6), a second one for port `P0` configured as input port (lines 8–12), and a third one for port `P1` configured as output port (lines 14–18). Similar to the StrongARM simulation model, the 8051 microcontroller is captured with an instruction-set simulator, in this case the Dalton ISS from the University of California at Riverside. The shared buffer is captured in an `ipblock` as well, starting on line 20. The shared buffer is specific to the 8051 microcontroller, and it is attached to the 8051's xbus (expansion bus). The buffer provides one read/write port for the hardware, while the other port is only accessible from within the 8051 software. The hardware module that accesses the ping-pong buffer is listed starting at line 30. The FSMD will first read locations 0 through $0 \times 1F$, and next locations 0×20 through $0 \times 3F$. The handshake protocol is implemented through the 8051's `P0` and `P1` port.

Listing 8.3 GEZEL model of a ping-pong buffer between an 8051 microcontroller and an FSMD

```
 1  ipblock my8051 {
 2    iptype "i8051system";
 3    ipparm "exec=ramrw.ihx";
 4    ipparm "verbose=1";
 5    ipparm "period=1";
 6  }
 7
 8  ipblock my8051_cmdo(out data : ns(8)) {
 9    iptype "i8051systemsource";
10    ipparm "core=my8051";
11    ipparm "port=P0";
12  }
13
14  ipblock my8051_cmdi(in data : ns(8)) {
15    iptype "i8051systemsink";
16    ipparm "core=my8051";
17    ipparm "port=P1";
18  }
19
20  ipblock my8051_xram(in  idata   : ns(8);
21                      out odata   : ns(8);
22                      in  address : ns(6);
23                      in  wr      : ns(1)) {
24    iptype "i8051buffer";
25    ipparm "core=my8051";
26    ipparm "xbus=0x4000";
27    ipparm "xrange=0x40"; // 64 locations at address 0x4000
28  }
29
30  dp pingpongreader {
   reg rreq, rack, rid : ns(1);
   reg radr            : ns(6);
   reg ramofs          : ns(6);
   sig adr             : ns(6);
   sig ramin, ramout   : ns(8);
   sig ramw            : ns(1);
   sig P0o, P0i        : ns(8);
   use my8051;
   use my8051_cmdo(P0o);
   use my8051_cmdi(P0i);
   use my8051_xram(ramin, ramout, adr, ramw);
   always         { rreq   = P0o[0];
                    adr    = radr;
                    ramw   = 0;
                    ramin  = 0;       }
   sfg noack      { P0i    = 0;       }
   sfg doack      { P0i    = 1;       }
   sfg getramofs0 { ramofs = 0x0;     }
   sfg getramofs2 { ramofs = 0x20;    }
   sfg readram0   { radr   = ramofs;  }
   sfg readram1   { radr   = radr + 1;
                    $display($cycle, " ram radr ", radr, "
                       data ", ramout);
                  }
}

fsm fpingpongreader(pingpongreader) {
  initial s0;
  state s1, s2, s3, s4, s5, s6;
  @s0 if (~rreq) then (noack)                  -> s1;
                 else (noack)                  -> s0;

  @s1 if (rreq) then (doack, getramofs0)       -> s2;
                else (noack)                   -> s1;

  @s2 (readram0, doack)                        -> s3;
  @s3 if (radr == 0x5) then (doack)            -> s4;
                       else (readram1, doack)  -> s3;

  @s4 if (~rreq) then (noack, getramofs2)      -> s5;
                 else (doack)                  -> s4;

  @s5 (readram0, noack)                        -> s6;
  @s6 if (radr == 0x25) then (doack)           -> s1;
                        else (readram1, doack) -> s6;
}

system S {
  pingpongreader;
}
```

Listing 8.4 shows the driver software for the 8051 microcontroller. This software was written for the Small Devices C Compiler (sdcc), a C compiler that supports a

Listing 8.4 8051 software driver for the ping-point buffer

```
#include <8051.h>

void main() {
  int i;

  volatile xdata unsigned char *shared =
             (volatile xdata unsigned char *) 0x4000;

  for (i=0; i<64; i++) {
    shared[i] = 64 - i;
  }

  P0 = 0x0;
  while (1) {

    P0 = 0x1;
    while (P1 != 0x1) ;

    // hw is accessing section 0 here.
    // we can access section 1
    for (i = 0x20; i < 0x3F; i++)
      shared[i] = 0xff - i;

    P0 = 0x0;
    while ((P1 & 0x1)) ;

    // hw is accessing section 1 here
    // we can access section 0
    for (i = 0x00; i < 0x1F; i++)
      shared[i] = 0x80 - i;
  }
}
```

broad range of microcontrollers. This compiler directly supports 8051 port access through symbolic names (`P0`, `P1`, and so on). In addition, the shared memory accesses can be modeled through an initialized pointer.

To cosimulate the 8051 and the hardware, we start by cross-compiling the 8051 C code to binary format. Next, we use the gplatform cosimulator to run the simulation. Because the microcontroller will execute an infinite program, the cosimulation is terminated after 60,000 clock cycles. The program output shows that the GEZEL model scans out the lower part of the ping-pong buffer starting at cycle 36952, and the upper part starting at cycle 50152. The cycle count is relatively high because the instruction length of a traditional 8051 microcontroller is high: each instruction takes 12 clock cycles to execute.

```
> sdcc --model-large ram.c
> gplatfrom -c 60000 block8051.fdl
```

```
i8051system: loading executable [ramrw.ihx]
0x00    0x00    0xFF    0xFF
0x01    0x00    0xFF    0xFF
36952 ram radr 0/1 data 40
36953 ram radr 1/2 data 3f
36954 ram radr 2/3 data 3e
36955 ram radr 3/4 data 3d
36956 ram radr 4/5 data 3c
0x00    0x01    0xFF    0xFF
50152 ram radr 20/21 data df
50153 ram radr 21/22 data de
50154 ram radr 22/23 data dd
50155 ram radr 23/24 data dc
50156 ram radr 24/25 data db
Total Cycles: 60000
```

8.4.3 UART on the AVR ATMega128

A third example shows how GEZEL can be used to report low-level behavior in hardware and how that can be correlated to high-level (embedded) software behavior. This application makes use of an ATmega128 AVR microcontroller, integrated in GEZEL through the SimulAVR ISS. The AVR application software will send characters through a serial communications link, and the GEZEL model will reveal the physical communications format of bits flowing over the serial link. The ATmega128 core contains several dedicated peripherals, including a Universal Asynchronous Receiver/Transmitter (UART). The use of the UART peripheral simplifies development of communications software on the ATmega128 microcontroller. Specifically, the UART peripheral supports serial-to-parallel conversion (for reception) and the parallel-to-serial conversion (for transmission) directly in hardware. In the ATmega128 software application, communication using a UART is implemented by means of reading from, and writing to, special-special purpose registers that control the UART hardware.

Listing 8.5 shows the system architecture, which instantiates an ATmega128 core (lines 1–5), and an input/output port (lines 7–13). The UART is included within the ATmega128 core; the input/output port only serves to provide access to the input/output pins of the UART peripheral. The ATmega128 has six general-purpose input/output ports, named A through F. Special peripherals, including the UART, are multiplexed on those ports. For UART device 0, used in this example, the least significant bits of port E are used for input (rxd) and output (txd) respectively. The parameter `port=E` on line 10 of Listing 8.5 configures the interface as port E of the Atmega128 core. The parameter `pindir=xxxxxx10` on line 11 configures the direction of the port bits. Bit 0 will be used for serial-input, while bit 1 will be used as serial-output. The pins are thus configured as input and output respectively.

The hardware attached to the transmit and receive pins of the UART is contained within lines 15–32 of Listing 8.5. The `baudmeter_ctl` module implements

Listing 8.5 GEZEL UART monitor

```
ipblock avrcore {
  iptype "atm128core";
  ipparm "exec=avruart.elf";
  ipparm "fcpuMhz=8";
}

ipblock avr_E_port (in   rxd : ns(1);
                    out  txd : ns(1)) {
  iptype "atm128port";
  ipparm "core=avrcore";
  ipparm "port=E";
  ipparm "pindir=xxxxxx10";
}

dp baudmeter(out rxd : ns(1);
             in  txd : ns(1)) {
  reg b : ns(1);
  always    { b = txd;
              rxd = b;
            }
  sfg bit10 { $display("@", $cycle, ": ->", txd); }
  sfg idle  { }
}
fsm baudmeter_ctl(baudmeter) {
  initial s0;
  state s1;
  @s0 if ( b) then (bit10) -> s1;
              else (idle)  -> s0;
  @s1 if (~b) then (bit10) -> s0;
              else (idle)  -> s1;
}

dp top {
  sig txd, rxd : ns(1);
  use avrcore;
  use avr_E_port(rxd, txd);
  use baudmeter(rxd, txd);
}

system S {
  top;
}
```

Listing 8.6 UART driver program

```
#include <avr/io.h>
#include <stdio.h>

#define F_CPU 8000000UL
#define UART_BAUD 38400

void uart_init(void) {
  UBRR0L = (F_CPU / (16UL * UART_BAUD)) - 1;
  UCSR0B = _BV(TXEN0) | _BV(RXEN0); /* tx/rx enable */
}

int uart_putchar(char c, FILE *stream) {
  loop_until_bit_is_set(UCSR0A, UDRE0);
  UDR0 = c;
  return 0;
}

int uart_getchar(FILE *stream) {
  uint8_t c;
  char *cp, *cp2;
  static char *rxp;
  loop_until_bit_is_set(UCSR0A, RXC0);
  if (UCSR0A & _BV(FE0))
    return _FDEV_EOF;
  if (UCSR0A & _BV(DOR0))
    return _FDEV_ERR;
  c = UDR0;
  return c;
}

FILE uart_stream = FDEV_SETUP_STREAM(uart_putchar,
                                     uart_getchar,
                                     _FDEV_SETUP_RW);

int main() {
  uint8_t c;
  uart_init();
  stdout = stdin = &uart_stream;
  putchar('/');  // ascii 0x2F
  c = getchar();
  putchar(c);
}
```

a loopback: it connects the transmit pin to the receive pin, so that every byte transmitted by the ATmega128 application will also be received by it. The module also detects every level transition on the serial line, and prints the direction of the transition as well as the clock cycle. By analyzing this debug output, we will reconstruct the waveform generated on the serial line.

The driver software for the ATmega128 is shown in Listing 8.6. This software is written for the avr-gcc compiler and uses the definitions of avr libc. Three UART functions (uart_init, uart_putchar, and uart_getchar) support initialization, character transmission, and character reception, respectively. The selected baudrate is 38,400 for a CPU clock rate of 8 MHz. Therefore, every UART symbol (a single bit) takes 208 CPU clock cycles. The uart_putchar function transmits a character. The function first busy-waits until the UART hardware is able to accept a character, and next initiates transmission by writing the character ASCII code in the transmitter data register. The uart_getchar function receives a character. The function busy-waits until the UART signals that a character has been received, then checks the UART error flags, and finally returns the received character. The FDEV_SETUP_STREAM macro is specific to the AVR C library. This macro creates a streaming input/output structure for streaming data to/from the UART. The main function of the test program transmits a single character, then receives one, and transmits it again. If the loopback is operating correctly, we can expect that every character will be received twice.

To simulate this design, we compile the AVR application with the AVR cross compiler. Next, we use gplatform to simulate it. We selected a run of 8,000 clock cycles.

```
> avr-gcc -mmcu=atmega128  avruart.c -o avruart.elf
> gplatform -c 8000 avruart.fdl
atm128core: Load program avruart.elf
atm128core: Set clock frequency 8 MHz
@238: ->1
@337: ->0
@545: ->1
@1377: ->0
@1585: ->1
@1793: ->0
@2209: ->1
@2625: ->0
@2833: ->1
@3665: ->0
@3873: ->1
@4081: ->0
@4497: ->1
Total Cycles: 8000
```

The transitions can be explained using Fig. 8.7. The application software transmits the slash character, which corresponds to ASCII code 0x2F, or 00101111 in binary. A UART serial transmission line sends data bytes starting with the least-significant-bit, so that we expect to find the pattern 11110100 as part of the serial transmission. The UART also transmits a start bit (always 0) and a stop bit (always 1) to mark the beginning and end of the transmission. The waveform for a

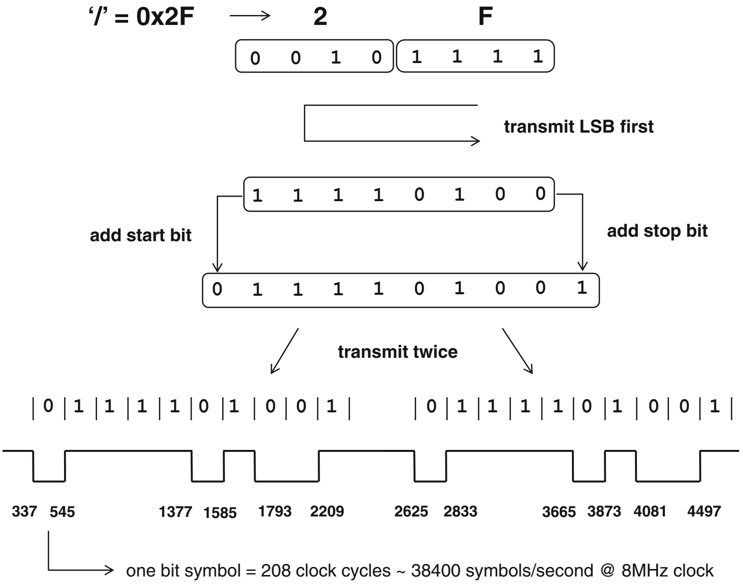


Fig. 8.7 Serial transmission of '//' as observed by GEZEL simulation

single slash character therefore corresponds to the pattern `0111101001`, and the waveform for the application will be a repetition of two times this character.

While the example discussed here is very basic, it illustrates a situation which occurs often in hardware/software codesign: activities in hardware need to be interpreted in terms of activities in software, and vice versa. This requires a designer to think across multiple, and possibly very different, abstraction levels. Just compare Listing 8.6 and Fig. 8.7, for example.

8.5 Summary

System-on-chip architecture is a balance between flexibility and specialization. The RISC core, the champion of flexibility in embedded designs, takes care of general-purpose processing, and acts as a central controller in SoC. Multiple additional specialized components, including memories, peripherals, and coprocessors, assist the RISC processor to handle specialized tasks. The interconnect infrastructure, consisting of on-chip bus segments, bus bridges, and specialized connections, help integrating everything together.

All of this makes the SoC a wide-spread paradigm that will be around for some years to come. It is a pragmatic solution that addresses several problems of modern electronic design at the same time. First, an SoC maintains flexibility and is applicable as a platform for several applications within an application domain. This reusability makes the SoC economically advantageous. Compared to a dedicated hardware design, the SoC chip is more general, and a given application can be created quicker. Second, an SoC contains specialized processing capabilities where needed, and this allows it to be energy-efficient. This greatly expands to potential applications of SoC.

In this chapter, we have reached the end of the second part in this book. The key objective of our journey was to investigate how dedicated hardware becomes flexible and programmable. We started from custom-hardware models coded as FSMD models. Next, we replaced the fixed finite state machine of an FSMD with a flexible micro-coded controller, and obtained a micro-programmed architecture. Third, we turned to RISC processors, which are greatly improved micro-programmed architectures that shield of software from hardware. Finally, we used the RISC as a central controller in the System-on-Chip architecture.

8.6 Further Reading

System-on-chip is a broad concept with many different dimensions. One of these dimensions is easier and faster design through reuse (Saleh et al. 2006). Another is that SoC technology is critical for modern consumer applications because of the optimal balance between energy-efficiency and flexibility (Claasen 2006). In recent years, alternative visions on SoC architectures have been given, and an interesting one is given in the book of Chris Rowen (2004). The example on the efficiency of on-chip interconnect comes from the same book.

The definition of *intrinsic computational power* of silicon is elaborated in the ISSCC99 article by Claasen (1999). The paper by Wulf and McKee on the Memory Wall can be found online (Wulf and McKee 1995). In 2004, one of the authors provided an interesting retrospective (McKee 2004).

The digital media processor discussed in this chapter is described in more detail by Talla and colleagues in (2004).

The instruction simulators integrated in GEZEL include a StrongARM core (based on Simit-ARM), an 8051 core (based on Dalton (Vahid 2009)), an AVR core (based on `simulavr`), and a Picoblaze core (based on `kpicosim`). Appendix A provides a summary on the use of each of those, including a few guidelines that show how to integrate your own instruction-set simulator.

8.7 Problems

Problem 8.1. Consider Fig. 8.1 again.

(a) Explain why the memory area occupied by the UART peripheral cannot be cached by the RISC processor.

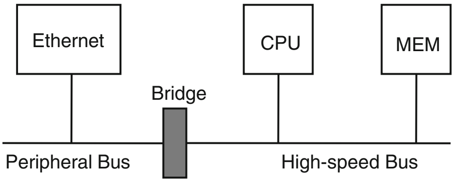


Fig. 8.8 System-on-chip model for Problem 8.2

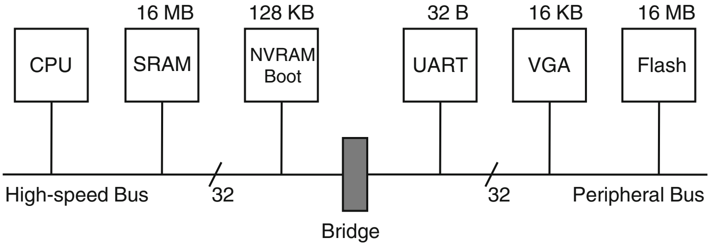


Fig. 8.9 System-on-chip model for Problem 8.3

(b) Assume that the high-speed bus would include a second RISC core, which also has an instruction-cache and a data-cache. Explain why, without special precautions, caching can cause problems with the stable operation of the system.
(c) A quick fix for the problem described in (b) could be obtained by dropping one of the caches in each processor. Which cache must be dropped: the instruction-cache or the data-cache?

Problem 8.2. Consider the simple SoC model in Fig. 8.8. Assume that the high-speed bus can carry 200 MWord/s, and the peripheral bus can carry 30 MWord/s. The CPU has no cache and requests the following data streams from the system: 80 MWord/s of read-only bandwidth for instructions, 40 Mword/s of read/write bandwidth for data, and 2 MWord/s for Ethernet packet input/output.

(a) What is the data bandwidth through the bus bridge?
(b) Assume you have to convert this architecture into a dual-core architecture, where the second core has the same data stream requirements as the first core. Discuss how you will modify the SoC. Keep in mind that you can add components and busses, but that you cannot change their specifications. Don't forget to add bus arbitration units, if you need them.

Problem 8.3. You have to design a memory map for the SoC shown in Fig. 8.9. The system contains a high-speed bus and a peripheral bus, both of them with a 32-bit address space and both of them carrying words (32 bit). The components of the system include a RISC, a 16 MB RAM memory, 128 KB of non-volatile program memory, a 16 MB Flash memory. In addition, there is a VGA peripheral and a UART peripheral. The VGA has a 16 KB video buffer memory, and the UART contains 32 bytes of transmit/receive registers.

(a) Draw a possible memory map for the processor. Keep in mind that the Bus Bridge can only convert bus transfers within a single, continuous address space.
(b) Define what address range can be cached by the processor. A ‘cached address range’ means that a memory-read to an address in that range will result in a backup copy stored in the cache.

Problem 8.4. Consider Listings 8.3 and 8.4 again. Modify the GEZEL program and the C program so that the FSMD *writes* into the shared memory, and the C program *reads* from the shared memory. Co-simulate the result to verify the solution is correct.

Part III
Hardware/Software Interfaces

The third part of this book makes a walkthrough of all the elements involved in connecting software and custom-hardware. We will discuss the general design principles of a hardware/software interface, and describe all the building blocks needed to implement it. This includes on-chip busses, microprocessor interfaces, and hardware interfaces.

Chapter 9
Principles of Hardware/Software Communication

9.1 Connecting Hardware and Software

Over the next few chapters, we will discuss various forms of interconnection between hardware components and software drivers. Figure 9.1 presents a synopsis of the elements in a hardware/software interface. The objective of the hardware/-software interface is to connect the software application to the custom-hardware module. There are five elements involved.

1. The microprocessor and the coprocessor are both connected to an on-chip communication mechanism, such as an on-chip bus. The *on-chip bus* transports data from the microprocessor module to the custom-hardware module. While typical on-chip buses are shared among several masters and slaves, they can also be implemented as dedicated point-to-point connections. For example, coprocessors are often attached to a dedicated link.
2. Both the microprocessor and the coprocessor need an interface to the on-chip communication bus. The *microprocessor interface* includes the hardware and low-level firmware to allow a software program to get 'out' of the microprocessor. A microprocessor can use several different mechanisms for this, such as coprocessor instructions, or memory load/store instructions.
3. The *hardware interface* includes the hardware needed to attach the coprocessor to the on-chip communication subsystem. For example, in the case of an on-chip bus, the hardware interface handles data coming from, and going to, the on-chip bus. The hardware interface will decode the on-chip bus protocol, and make the data available to the custom-hardware module through a register or a dedicated memory.
4. The software application is connected to the microprocessor interface through a *software driver*, a small module that wraps transactions between hardware and software into software function calls. This driver converts software-centric paradigms (pointers, arrays, variable-length data structures) into structures that are suited for communication with hardware. To achieve this goal, the software driver may require the introduction of additional data structures and commands.

P.R. Schaumont, *A Practical Introduction to Hardware/Software Codesign*,
DOI 10.1007/978-1-4614-3737-6_9, © Springer Science+Business Media New York 2013

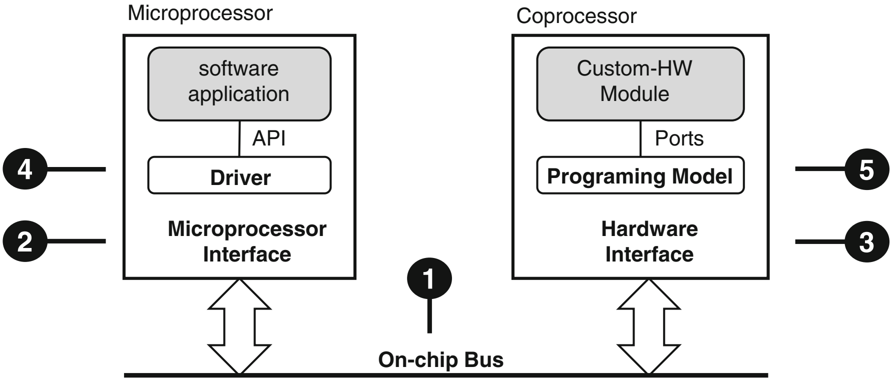


Fig. 9.1 The hardware/software interface

5. The custom-hardware module is connected to the custom-hardware interface through a *programming model*, a structure that presents an abstraction of the hardware to the software application. The hardware interface encapsulates the custom-hardware module, which may have an arbitrary number of ports, parameters. The programming model, on the other hand, should be easy to handle by the microprocessor: it uses memory locations, coprocessor instructions, and so on. To implement this conversion, the hardware interface may require the introduction of additional storage and controls.

Clearly, the link from hardware to software represents a rich and complex design space. In any given codesign problem, you'll find that there are many different ways to implement a link from hardware to software. However, there may be just a few of them that have acceptable overhead in the application.

In this chapter, we discuss the principles of hardware/software interface design. In the chapter after that, we will describe on-chip busses. Further, there will be one chapter on the microprocessor interface (with driver), and one on the custom-hardware interface (with hardware interface).

9.2 Synchronization Schemes

How can we guarantee that the software application and the custom-hardware module will remain synchronized, given that they are independently executing entities? How does a hardware module know that a software program wishes to communicate with it? Answering these questions requires us to select a synchronization scheme.

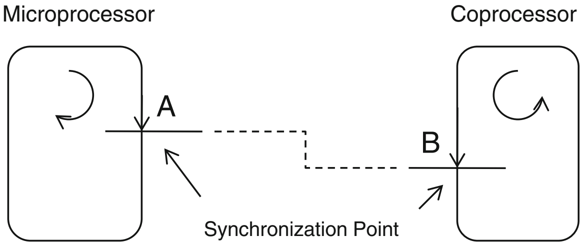


Fig. 9.2 Synchronization point

9.2.1 Synchronization Concepts

We define *synchronization* as the structured interaction of two otherwise independent and parallel entities. Figure 9.2 illustrates the key idea of synchronization. Two entities, in this case a micro-processor and a coprocessor, each have an independent thread of execution. Through synchronization, one point in the execution thread of the microprocessor is tied to one point in the control flow of the coprocessor. This is the synchronization point. Synchronization must guarantee that, when the microprocessor is at point A, then the coprocessor will be at point B.

Synchronization is needed to support communication between parallel subsystems: every *talker* needs to have a *listener* to be heard. Obviously, if parallel components never interact, there's no point in keeping them synchronized. We discussed communication within parallel systems before: recall our discussion on the implementation of data-flow (Chap. 2). In data-flow, different actors communicate with one another through the exchange of tokens. Assume that one actor is implemented in software and another one is implemented as a custom-hardware module. Also, assume that the software actor sends tokens to the hardware actor. According to the rules of data-flow, each token produced must eventually be consumed, and this implies that the hardware actor must know when the software actor is sending that token. In other words: the hardware and software actors will need to synchronize when communicating a token. Of course, there are many different ways to realize a data-flow communication, depending on how we realize the data-flow edge. But, regardless of the realization, the requirement to synchronize does not go away. For example, one may argue that a FIFO memory could be used to buffer the tokens going from software to hardware, thereby allowing hardware and software actors to run 'more independently'. Well, FIFO memories do not remove the requirement to synchronize. When the FIFO is empty, the hardware actor will need to wait until a token appears, and when the FIFO is full, the software actor will need to wait until a free space appears.

Synchronization is an interesting problem because it has several dimensions, each with several levels of abstraction. Figure 9.3 shows the three dimensions of interest: time, data, and control. In this section, we explain the meaning of these dimensions. In further sections, we discuss several examples of synchronization mechanisms.

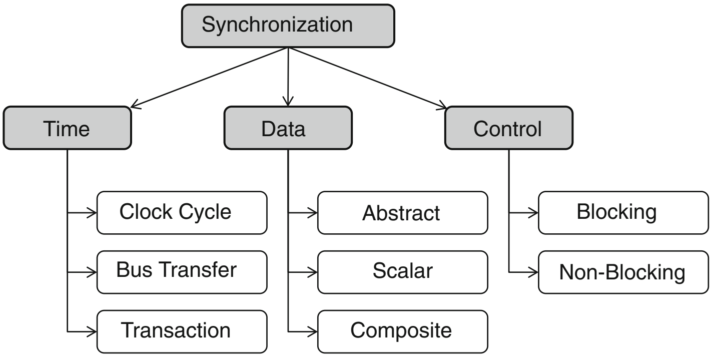


Fig. 9.3 Dimensions of the synchronization problem

The *dimension of time* expresses the granularity at which two parallel entities synchronize. Clock-cycle accuracy is needed when we interface two hardware components with each other. Bus-transfer accuracy is needed when the granularity of synchronization is expressed in terms of a specific bus protocol, such as a data transfer from a master to a slave. Finally, transaction accuracy is needed when the granularity of synchronization is a logical transaction from one entity to the next. Note that the meaning of *time* varies with the abstraction level, and it does not always have to correspond to wall-clock time. Instead, time synchronization may also refer to clock cycles, bus transfers, and logical transfers. Time synchronization may be even limited to a partial ordering, as in *A does not happen before B*, for example.

In practice, a synchronization scheme between hardware and software covers all abstraction levels in time: A high-level, logical synchronization implemented with bus transfers will only work if the bus transfers themselves are synchronized. In turn, each bus transfer may require a protocol that takes several clock cycles, and the communicating subsystems will synchronize at each clock edge. However, for a hardware-software codesigner, being able to think about synchronization problems at higher levels of abstraction is fundamental, especially in the earlier phases of a design.

The *data dimension of synchronization* determines the size of the container involved in the synchronization. When no data is involved at all, the synchronization between two entities is abstract. Abstract synchronization is useful to handle access to a shared resource. On the other hand, when two parallel entities communicate data values, they will need a shared data container. The communication scheme works as follows: one entity dumps information in the data container, and synchronizes with the second entity. Next, the second entity retrieves information of the data container. The first entity should not overwrite the value in the data container until it can be sure that the second entity has synchronized. In this scheme, synchronization is used to indicate when there is something of interest in the shared data container. Under scalar data synchronization, the two entities will synchronize for every single data item transferred. Under composite data synchronization, the two entities will synchronize over a group of data items.

The *control dimension of synchronization* indicates how the local behavior in each entity will implement synchronization. In a blocking scheme, the synchronization can stall the local thread of control. In a non-blocking scheme, the local behavior will not be stalled, but instead a status signal is issued to indicated that the synchronization primitive did not succeed.

A hardware-software co-designer is able to make decisions along each of these three dimensions separately. In the following sections, several examples of synchronization will be described.

9.2.2 Semaphore

A *semaphore* is a synchronization primitive which does not involve the transfer of data, but instead controls access over an abstract, shared resource. A semaphore S is a shared resource that supports two operations: grabbing the semaphore ($P(S)$) and releasing the semaphore ($V(S)$). These operations can be executed by several concurrent entities. In this case, we will assume there are two entities competing for the semaphore. The P and V are the first letters of the Dutch verbs 'proberen' and 'verhogen', chosen by the scientist who proposed using semaphores in system software, Edgser Dijkstra.

The meaning of $P(S)$ and $V(S)$ is as follows. $P(S)$ and $V(S)$ are indivisible operations that manipulate the value of a semaphore. Initially, the value of the semaphore is 1. The operation $P(S)$ will decrement the semaphore by one. If an entity tries to $P(S)$ the semaphore while it is zero, then $P(S)$ will stall further execution of that entity until the semaphore is nonzero. Meanwhile, another entity can increment the semaphore by calling $V(S)$. When the value of the semaphore is non-zero, any entity which was stalled on a $P(S)$ operation will decrement the semaphore and proceed. In case multiple entities are blocked on a semaphore, one of them, chosen at random, will be able to proceed. The maximum value of the basic binary semaphore is 1. Calling $V(S)$ several times will not increase the semaphore above 1, but it will not stall either. There are more elaborate semaphore implementations as well, such as counting semaphores. For our purpose however, binary semaphores are sufficient.

Using semaphore operations, it is possible to describe the synchronization of two concurrent entities. The pseudocode in Listing 9.1 is an example using a single semaphore. The first of two concurrent entities needs to send data to the second entity through a shared variable `shared_data`. When the first entity starts, it immediately decrements the semaphore. Entity two, on the other hand, waits for a short while, and then will stall on the semaphore. Meanwhile, entity one will write into the shared variable, and increment the semaphore. This will unlock the second entity, which can now read the shared variable. The moment when entity

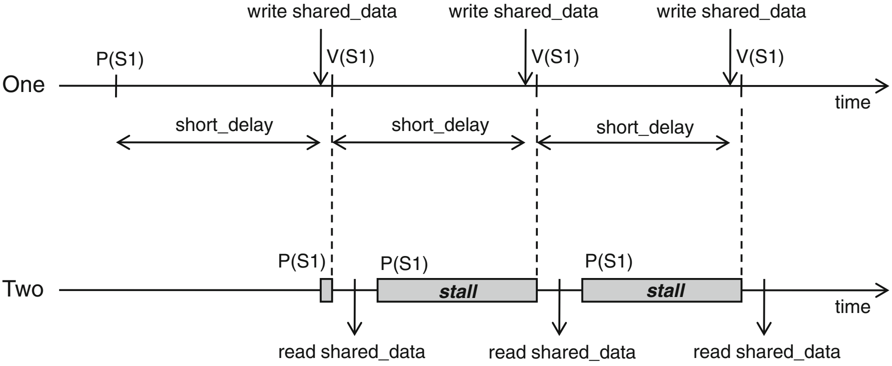


Fig. 9.4 Synchronization with a single semaphore

Listing 9.1 One-way synchronization with a semaphore

```
int shared_data;
semaphore S1;

entity one {
  P(S1);
  while (1) {
    short_delay();
    shared_data = ...;
    V(S1);              // synchronization point
  }
}

entity two {
  short_delay();
  while (1) {
    P(S1);              // synchronization point
    received_data = shared_data;
  }
}
```

one calls $V(S1)$ and entity two is stalled on $P(S1)$ is of particular interest: it is the synchronization point between entities one and two.

Figure 9.4 illustrates the interaction between entities one and two. The dashed lines indicate the synchronization points. Because entity two keeps on decrementing the semaphore faster than entity one can increment it, entity two will always stall. As a result, each write of shared_data by entity one is followed by a matching read in entity two.

Yet, this synchronization scheme is not perfect, because it assumes that entity two will always arrive first at the synchronization point. Now assume that the slowest entity would be entity two instead of entity one. Under this assumption, it is possible

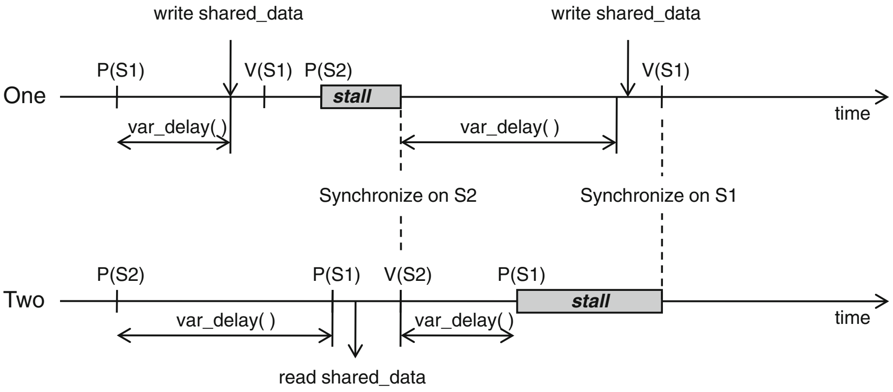


Fig. 9.5 Synchronization with two semaphores

that entity one will write `shared_data` several times before entity two can read a single item. Indeed, $V(S1)$ will not stall even if it is called several times in sequence. Such a scenario is not hard to envisage: just move the *short_delay*() function call from the while-loop in entity one to the while-loop in entity two.

This observation leads to the conclusion that the general synchronization of two concurrent entities needs to work in two directions: one entity needs to be able to wait on the other, and vice versa. In the *producer/consumer* scenario explained above, the producer will need to wait for the consumer if that consumer is slow. Conversely, the consumer will need to wait for the producer if the producer is slow. We can address the situation of unknown delays with a two-semaphore scheme, as shown in Listing 9.2. At the start, each entity decrements a semaphore. $S1$ is used to synchronize entity two, while $S2$ is used to synchronize entity one. Each entity will release its semaphore only after the read-operation (or write-operation) is complete.

Figure 9.5 illustrates the case where two semaphores are used. On the first synchronization, entity one is quicker than entity two, and the synchronization is done using semaphore $S2$. On the second synchronization, entity two is faster, and in this case the synchronization is done using semaphore $S1$.

9.2.3 One-Way and Two-Way Handshake

In parallel systems, concurrent entities may be physically distinct, and implementing a centralized semaphore may not be feasible. To handle this situation, we will use a *handshake*: a signaling protocol based on signal levels. The concepts of semaphore-based synchronization still apply: we implement a synchronization point by making one entity wait for another one.

The most simple implementation of a handshake is a *one-way handshake*, which needs only one wire. Figure 9.6 clarifies the implementation of this handshake

Listing 9.2 Two-way synchronization with two semaphores

```
int shared_data;
semaphore S1, S2;

entity one {
  P(S1);
  while (1) {
    variable_delay();
    shared_data = ...;
    V(S1);  // synchronization point 1
    P(S2);  // synchronization point 2
  }
}

entity two {
  P(S2);
  while (1) {
    variable_delay();
    P(S1);  // synchronization point 1
    received_data = shared_data;
    V(S2);  // synchronization point 2
  }
}
```

for the case of two hardware modules. When we will discuss hardware/software interfaces, we will also consider handshakes between hardware and software. In this figure, entity one transmits a query signal to entity two. Entity two captures this signal in a register, and uses its value as a state transition condition. The synchronization point is the transition of `S0` to `S1` in entity one, with the transition of `S2` to `S3` in entity two. Entity two will wait for entity one until both of them can make these transitions in the same clock cycle. Entity one needs to set of acknowledge signal to high one cycle *before* the actual synchronization point, because the request input in entity two is captured in a register.

The limitation of a one-way handshake is similar to the limitation of a one-semaphore synchronization scheme: it only enables a single entity to stall. To accommodate arbitrary execution orderings, we need a two-way handshake as shown in Fig. 9.7. In this case, two symmetrical handshake activities are implemented. Each time, the query signal is asserted during the transition preceding the synchronization point. Then, the entities wait until they receive a matching response. In the timing diagram of Fig. 9.7, entity one arrives first in state `S0` and waits. Two clock cycles later, entity two arrives in state `S2`. The following clock cycle is the synchronization point: as entity one proceeds from `S0` to `S1`, entity two makes a corresponding transition from `S2` to `S3`. Because the handshake process is bidirectional, the synchronization point is executed correctly regardless of which entity arrives first at that point.

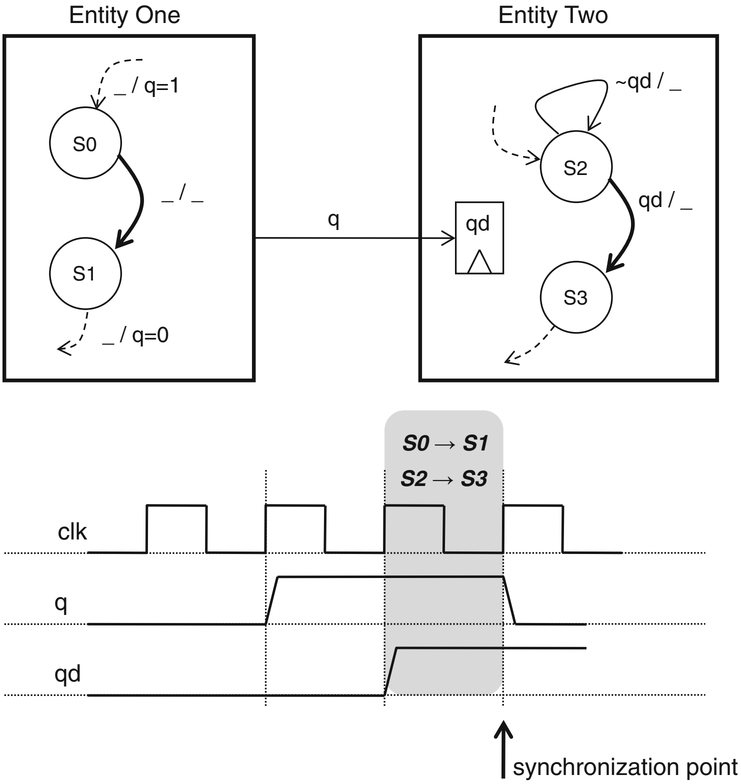


Fig. 9.6 One-way handshake

There are still some opportunities for optimization. For example, we can de-assert the response signal already during the synchronization point, which will make the complete handshake cycle faster to complete. We can also design the protocol such that it uses level transitions rather than absolute signal levels. Some of these optimizations are explored in the Problems at the end of this Chapter.

9.2.4 Blocking and Non-blocking Data-Transfer

Semaphores and handshakes are different ways to implement a synchronization point. A hardware/software interface uses a synchronization point to transfer data. The actual data transfer is implemented using a suitable hardware/software interface, as will be described later in this chapter.

An interesting aspect of the data transfer is how a synchronization point should be implemented in terms of the execution flow of the sender or receiver. If a sender or receiver arrives too early at a synchronization point, should it wait idle until the proper condition comes along, or should it go off and do something else? In

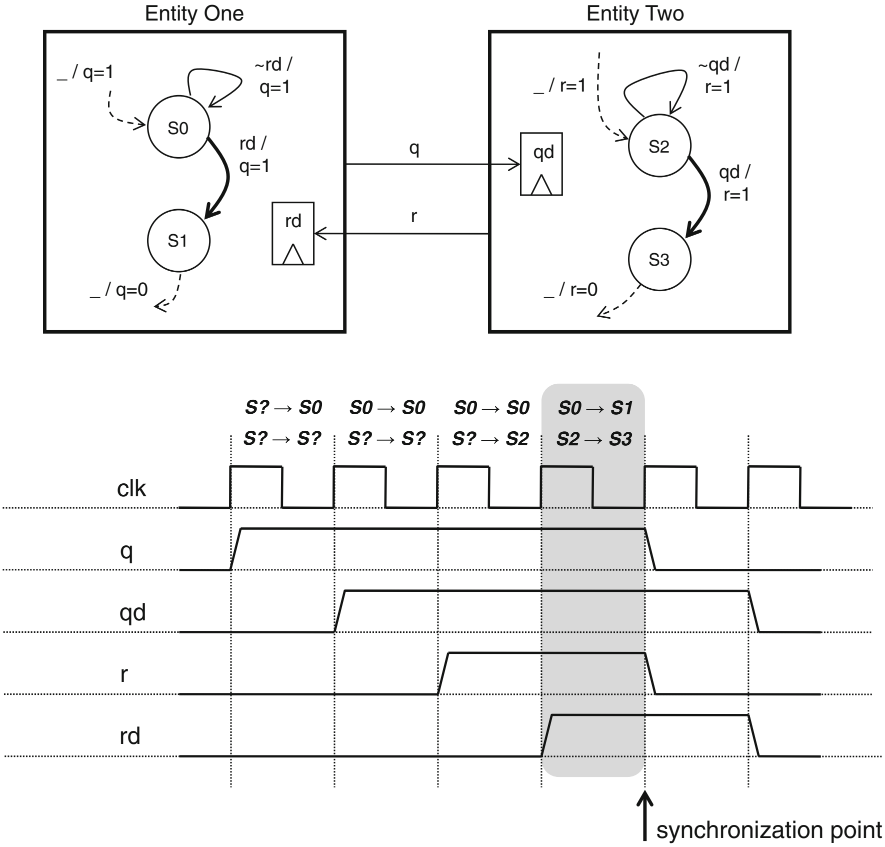


Fig. 9.7 Two-way handshake

terms of the send/receive operations in hardware and software, these two cases are distinguished as *blocking* data transfers and *non-blocking* data transfers.

A blocking data transfer will stall the execution flow of the software or hardware until the data-transfer completes. For example, if software has implemented the data transfer using function calls, then a blocking transfer would mean that these functions do not return until the data transfer has completed. From the perspective of the programmer, blocking primitives are the easiest to work with. However, they may stall the rest of the program.

A non-blocking data transfer will not stall the execution flow of software or hardware, but the data transfer may be unsuccessful. So, a software function that implements a non-blocking data transfer will need to introduce an additional status flag that can be tested. Non-blocking data transfers will not stall, but they require additional attention of the programmer to deal with exception cases.

Both of the semaphore and handshake schemes discussed earlier implement a blocking data-transfer. To use these primitives for a non-blocking data transfer, a sender or receiver should be able to test the outcome of the synchronization operation, without actually engaging in it.

9.3 Communication-Constrained Versus Computation-Constrained

In the selection of a suitable hardware/software communication interface, the resulting performance of the system is of crucial importance. Very often, the main reason for designing a custom hardware module is that the designer hopes to increase overall system performance with it. The argument of hardware acceleration is very often made in terms of computational performance. Consider the following example.

> A function *XYZ* executes on a slow software processor and takes 100 ms to complete. By writing a hardware implementation of *XYZ*, the execution time decreases to 1 ms. Hence, the system can be accelerated by a factor of 100.

There is an important pitfall in this reasoning. The overall application still runs on the slow software processor. Having *XYZ* execute in fast hardware does not help, if the software application cannot efficiently make use of the hardware module. For example, let's say that, due to an inefficient hardware/software interface, invoking the hardware version of *XYZ* takes 20 ms. Then, the system speedup is only a factor of 5, not 100!

In practical situations, we may analyze such performance limits. An example is given in Fig. 9.8. This hardware module has three input ports and one output port. Thus, each time we invoke the hardware coprocessor,we need to transfer $128 + 128 + 32 + 32 = 320$ bits. Let's assume that this custom hardware module takes

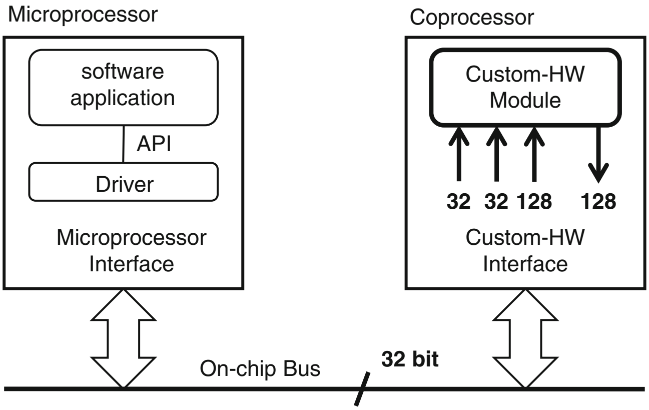


Fig. 9.8 Communication constraints of a coprocessor

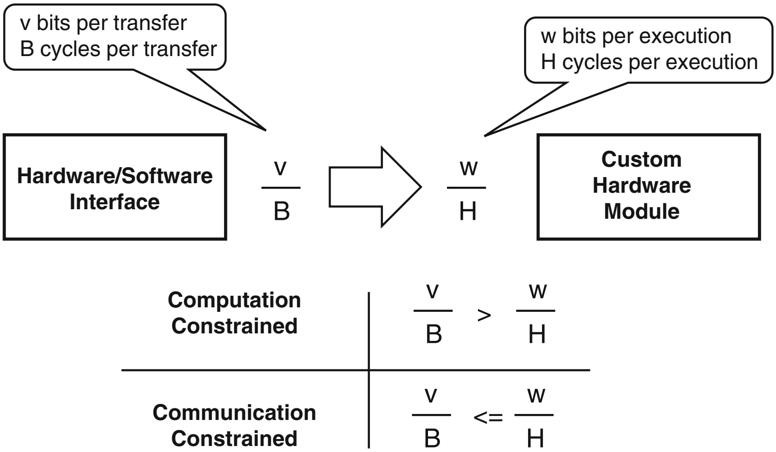


Fig. 9.9 Communication-constrained system vs. computation-constrained system

five cycles to compute a result. Hence, when this module is connected to software, and we wish to run the module at full performance, we will need to support a data bandwidth of $320/5 = 64$ bits per cycle. This data bandwidth needs to be delivered through a hardware/software interface. As illustrated in Fig. 9.8, a 32-bit bus is used to deliver data to the coprocessor. Since each bus transfers requires at least one clock cycle, the bus cannot provide more than 32 bits per cycle. Clearly, for full utilization, the coprocessor needs a larger data bandwidth than can be provided through the hardware/software interface. In this case, the system is *communication-constrained*.

Now, let's assume that the hardware coprocessor takes 50 cycles (instead of 5) to complete the operation. In this case, full utilization of the hardware will require $320/50 = 6.4$ bits per cycle. A 32-bit bus may be able to deliver the data that will keep the hardware module fully utilized. The system thus is *computation-constrained*.

Figure 9.9 summarizes these observations. The distinction between a communication-constrained system and a computation-constrained system is important, since it tells the designer where to put design effort. In a communication-constrained system, it does not make sense to implement a more powerful coprocessor, since it will remain under-utilized. Conversely, in a computation-constrained system, we don't need to look for a faster hardware/software interface. Even if the exact performance limits in a hardware-software codesign problem may be very hard to determine, it is often feasible to do a back-of-the-envelope calculation, and find out if a system is computation-constrained or communication-constrained.

An additional insight can be gained from the number of clock cycles needed per execution of the custom hardware module. This number is called the *hardware sharing factor* or HSF. The HSF is defined as the number of clock cycles that are available in between each input/output event. For example, an HSF of 10 would

Table 9.1 Hardware sharing factor

Architecture	HSF
Systolic array processor	1
Bit-parallel processor	1–10
Bit-serial processor	10–100
Micro-coded Processor	>100

mean that a given hardware architecture has a cycle budget of 10 clock cycles between successive input/output events. Thus, if this architecture would contain two multiplier operators, then these 10 clock cycles are adequate to support up to 20 multiplications. The HSF is an indication if a given architecture is powerful enough to sustain a computational requirement. Indeed, there is a strong correlation between the internal architecture of a hardware module and its HSF. This is illustrated in Table 9.1.

- A systolic-array processor is a multi-dimensional arrangement of computation units (datapaths or dataflow-actor-like processors) that operate on one or more parallel streams of data items.
- A bit-parallel processor is a processor with bit-parallel operators such as adders and multipliers that operates under control of a simple engine (such as a FSMD or a micro-programmed controller).
- A bit-serial processor is a processor with bit-serial operators, i.e. operators that compute on a single bit at a time, under control of a simple engine (such as a FSMD or a micro-programmed controller).
- A micro-coded processor is a processor with an instruction-fetch, similar to a general purpose processor.

This means that knowledge of the HSF, at the start of a design, may help a designer to select the right architecture style for the coprocessor.

9.4 Tight and Loose Coupling

The third generic concept in hardware/software interfaces is that of *coupling*. Coupling indicates the level of interaction between the execution flow in software and the execution flow in custom-hardware. In a tight coupling scheme, custom-hardware and software synchronize often, and exchange data often, for example at the granularity of a few instructions in software. In a loose coupling scheme, hardware and software synchronize infrequently, for example at the granularity of a function or a task in software. Thus, *coupling* relates the ideas of synchronization with performance.

We don't give a formal description of coupling, but instead describe it in terms of an example. First, let's compare the difference between a hardware module attached to the memory bus of a processor, and a hardware module that is attached directly to a dedicated port on the processor. These two interfaces – which we will

Table 9.2 Comparing a coprocessor interface with a memory-mapped interface

Factor	Coprocessor interface	Memory-mapped interface
Addressing	Processor-specific	On-chip bus address
Connection	Point-to-point	Shared
Latency	Fixed	Variable
Throughput	Higher	Lower

describe in detail in the next Chapters – are called memory-mapped interface and coprocessor interface respectively. Table 9.2 compares the key features of these two interfaces.

These two interfaces each take a different approach to synchronization. In the case of a coprocessor interface, synchronization between a hardware module and software is at the level of a single instruction. Such a coprocessor instruction typically carries both operands (from software driver to hardware coprocessor) as well as result (from hardware coprocessor to software driver). In the case of a memory-mapped interface, synchronization between a hardware module and software is at the level of a single bus transfer. Such a bus transfer is unidirectional (read or write), and either carries operands from the software driver to the hardware coprocessor, or else results from the hardware coprocessor to the software driver. Clearly, the use of a coprocessor interface versus a memory-mapped interface imply a different style of synchronization between hardware and software.

A given application can use either tight-coupling or loose-coupling. Figure 9.10 shows how the choice for loose-coupling of tight-coupling can affect the latencies of the application. The left side of the figure illustrates a tight-coupling scheme. The software will send four separate data items to the custom hardware, each time collecting the result. The figure assumes a single synchronization point which sends the operand and retrieves the result. This is the scheme that could be used by a coprocessor interface. The synchronization point corresponds to the execution of a coprocessor instruction in the software.

The right side of the figure illustrates a loosely coupled scheme. In this case, the software provides a large block of data to the custom hardware, synchronizes with the hardware, and then waits for the custom hardware to complete processing and return the result. This scheme would be used by a memory-mapped interface, for example using a shared-memory.

Loosely-coupled schemes tend to yield slightly more complex hardware designs because the hardware needs to deal more extensively with data movement between hardware and software. On the other hand, tightly-coupled schemes lean more on the software to manage overall system execution. Achieving a high degree of parallelism in the overall design may be easier to achieve with a loosely-coupled scheme than with a tightly-coupled scheme.

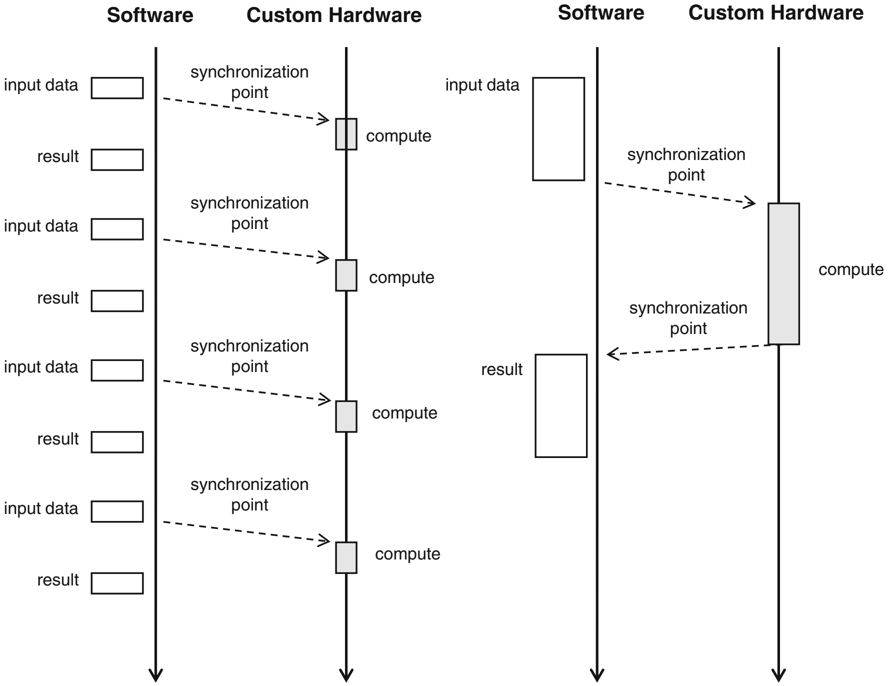


Fig. 9.10 Tight coupling versus loose coupling

9.5 Summary

In this Chapter, we discussed three concepts in hardware/software interface design. The first concept is that of synchronization, the structured interaction of parallel and independent entities. The second concept is the difference between communication-constrained and computation-constrained systems. The final concept is the difference between loose coupling and tight coupling.

All hardware/software interfaces need to deal with synchronization, and often you will need to build your own, application-level synchronization using the low-level primitives provided by a hardware/software interface. For example, the implementation of an internet-protocol component may require to process a full packet of data at a time.

Hardware coprocessors only make sense in computation-constrained systems. When a system becomes communication-constrained, one must carefully examine the overall design to determine if the system architecture (i.e. the application software *and* the hardware architecture) is still adequate for the problem at hand.

Coupling is arguably the most complex concept in this chapter, because it is the most abstract one. Often, by defining the system operation as a set of collaborating tasks, one can also establish the required coupling between components.

9.6 Further Reading

The theory of synchronization is typically discussed in depth in textbooks on Parallel or Concurrent Programming, such as Taubenfeld (2006) or Moderchai (2006). The original documents by Dijkstra are available from the E. W. Dijkstra Archive at the University of Texas (Dijkstra 2009). They're in Dutch! Een uitstekende gelegenheid dus, om Nederlands te leren.

9.7 Problems

Problem 9.1. Find the maximum communication speed from CPU1 to CPU2 in the system architecture shown in Fig. 9.11. Assume that the CPUs have a dedicated synchronization channel available so that they will be able to choose the most optimal moment to perform a read- or a write-transaction. Use the following design constants.

- Each bus transaction on the high-speed bus takes 50 ns.
- Each bus transaction on the low-speed bus takes 200 ns.
- Each memory access (read or write) takes 80 ns.
- Each bridge transfer takes 100 ns.
- The CPU's are much faster than the bus system, and can read/write data on the bus at any chosen data rate.

Problem 9.2. Consider the two-way handshake in Fig. 9.12. A sender synchronizes with a receiver and transmits a sequence of data tokens through a data register.

(a) Describe under what conditions register `r1` can be removed without hurting the integrity of the communication. Assume that, after taking `r1` away, the `req` input of the receiver is tied to logic-1.
(b) Describe under what conditions register `r3` can be removed without hurting the integrity of the communication. Assume that, after taking `r3` away, the `req` input of the sender is tied to logic-1.

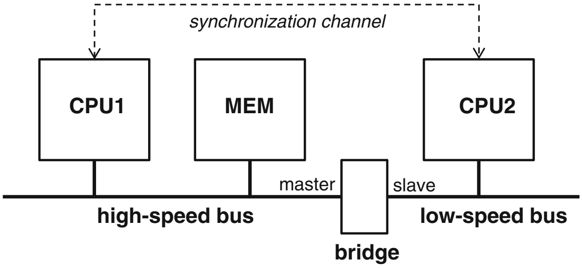


Fig. 9.11 System topology for Problem 9.1

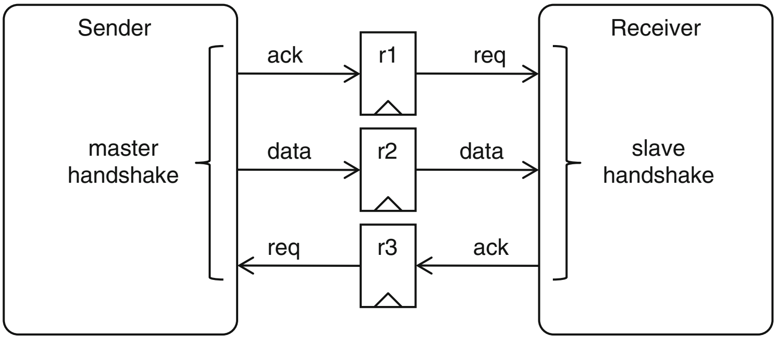


Fig. 9.12 Two-way handshake for Problems 9.1 and 9.4

(c) Assume that you would substitute register `r1` by two registers in series, so that the entire transition from sender-ack to receiver-req now takes two clock cycles instead of one. Describe the effect of this change on the throughput of the communication, and describe the effect of this change on the latency of the communication.

Problem 9.3. A C function has ten inputs and ten outputs, all of them integers. The function takes 1,000 cycles to execute in software. You need to evaluate if it makes sense to build a coprocessor for this function. Assume that the function takes K cycles to execute in hardware, and that you need Q cycles to transfer a word between the software and the coprocessor over a system bus. Draw a chart that plots Q in terms of K, and indicate what regions in this chart justify a coprocessor.

Problem 9.4. Consider the two-way handshake in Fig. 9.12. Implement this two-way handshake by developing an FSMD for the sender and the receiver. Next, optimize the two-way handshake so that *two* tokens have been transferred each time `req` and `ack` have make a complete handshake and returned to the logic-0 state.

Chapter 10
On-Chip Busses

10.1 On-Chip Bus Systems

This section describes the basic structure of an on-chip bus, and defines common terminology and notations.

10.1.1 A Few Existing On-Chip Bus Standards

The discussions in this chapter are based on four different bus systems: the AMBA bus, the CoreConnect bus, the Avalon bus, and the Wishbone bus. There are many other bus systems, but these are commonly used in contemporary System-on-Chip design.

- **AMBA** (Advanced Microcontroller Bus Architecture) is the bus system used by ARM processors. Originally introduced in 1995, the AMBA bus is now in its fourth generation, and it has evolved into a general on-chip interconnect mechanism. The fourth generation of AMBA provides five variants of interconnect: A general-purpose, low-bandwidth bus called APB (Advanced Peripheral Bus), a high-speed single-frequency bus called AHB (Advanced High-performance Bus), a high-speed multi-frequency bus called AXI (AMBA Advanced Extensible Interface), a system control bus called ASB (Advanced System Bus) and a debug control bus called ATB (Advanced Trace Bus). We will focus on the peripheral bus (APB) and, for selected cases, on the high-speed bus (AHB).
- **CoreConnect** is a bus system proposed by IBM for its PowerPC line of processors. Similar the AMBA bus, the CoreConnect bus comes in several variants. The main components include a general-purpose, low-bandwidth bus called OPB (On-chip Peripheral Bus) and a high-speed single-frequency bus called PLB (Processor Local Bus).

P.R. Schaumont, *A Practical Introduction to Hardware/Software Codesign*,
DOI 10.1007/978-1-4614-3737-6_10, © Springer Science+Business Media New York 2013

Table 10.1 Bus configurations for existing bus standards

Bus	High-performance shared bus	Peripheral shared bus	Point-to-point bus
AMBA v3	AHB	APB	
AMBA v4	AXI4	AXI4-lite	AXI4-stream
Coreconnect	PLB	OPB	
Wishbone	Crossbar topology	Shared topology	Point to point topology
Avalon	Avalon-MM	Avalon-MM	Avalon-ST

AHB AMBA highspeed bus, *APB* AMBA peripheral bus, *AXI* advanced extensible interface, *PLB* processor local bus, *OPB* onchip peripheral bus, *MM* memory-mapped, *ST* streaming

- **Avalon** is a bus system developed by Altera for use in SoC applications of its Nios processor. Avalon is defined in terms of the different types of interfaces it provides to SoC components. The most important of these are Avalon-MM for a memory-mapped interface, and Avalon-ST for a streaming interface. These two interfaces are used on shared busses and point-to-point busses, respectively.
- **Wishbone** is an open-source bus system proposed by SiliCore Corporation. The bus is used by many open-source hardware components, for example those in the OpenCores project (http://www.opencores.org). The Wishbone bus is simpler than the previous standards. The specification defines two interfaces (a master-interface and a slave-interface) from which various bus topologies can be derived.

Rather than describing each bus separately, we will unify them in a generic bus that reflects the common characteristics of all of them. We will occasionally point how each of AMBA, CoreConnect, Avalon and Wishbone implement the features of this generic bus.

We distinguish two bus configurations: the shared bus, and the point-to-point bus. The difference between these two is simply the number of components sharing the bus. Shared busses are most common, and several standards (AMBA, CoreConnect) define multiple versions of them, differing in complexity and performance. Table 10.1 shows the configurations defined for four different bus interface standards. The following two subsections provide a generic definition of a shared bus and of a point-to-point bus.

10.1.2 Elements in a Shared Bus

An on-chip bus system implements a bus protocol: a sequence of steps to transfer data in an orderly manner. A typical on-chip bus system will consist of one or more bus segments, as shown in Fig. 10.1. Each bus segment groups one or more bus *masters* with bus *slaves*. Bus bridges are directional components to connect bus segments. A bus-bridge acts as a slave at the input, and as a master at the output. At any particular moment, a bus segment is under control of either a bus master or a bus arbiter. A bus arbiter's role is to decide which bus master is allowed to control

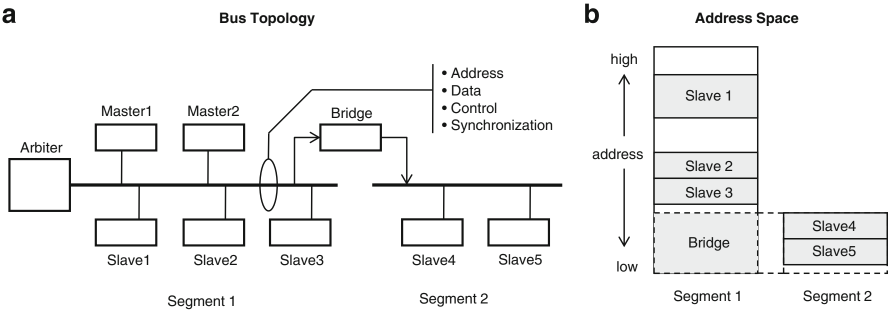


Fig. 10.1 (**a**) Example of a multi-master segmented bus system. (**b**) Address space for the same bus

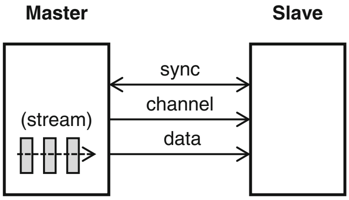
the bus. The arbitration is done for each bus transaction. It should be done in a fair manner such that no bus masters gets permanently locked out of bus access. The bus slaves can never obtain control over a bus segment, but instead have to follow the directions of the bus master that owns the bus.

A bus system uses an *address space* to organize the communication between components. A sample address space is shown on the right of Fig. 10.1. Usually, the smallest addressable entity in an address space is 1 byte. Each data transfer over the bus is associated with a given destination address. The destination address determines what component should pick up the data. Bus bridges are address-transparent: they will merge the slave address spaces from their output, and transform it to a single slave address space at their input.

An on-chip bus physically consists of a bundle of wires, which includes the following four categories: address wires, data wires, command wires, and synchronization wires.

- Data wires transfer data items between components. As discussed in the previous chapter, on-chip wiring is very dense, and data wires do not have to be multiplexed. Masters, slaves and bridges will have separate data inputs and data outputs.
- Address wires carry the address that goes with a given data item. The process of recognizing the destination address is called *address decoding*. One approach to implement address decoding is to implement it inside of the bus slave. Another approach is to perform address decoding centrally, and to distribute the decoded address signals directly to the slaves.
- Command wires describe the nature of the transfer to be performed. Simple commands include *read* and *write*, but larger on-chip bus systems may contain a wide variety of commands, that qualify a given read or write command. Several examples will be discussed later for actual on-chip buses.
- Synchronization wires ensure that bus masters and bus slaves are synchronized during data transfer. Common on-chip bus systems today are synchronous. They use a single clock signal per bus segment: all data, address, and command

Fig. 10.2 Point-to-point bus



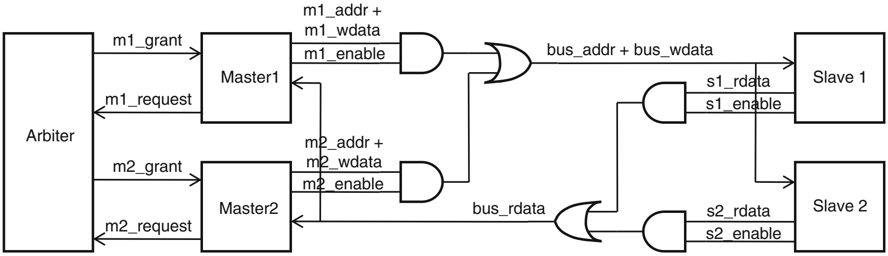


Fig. 10.3 Physical interconnection of a bus. The *_addr, *_wdata, *_sdata signals are signal vectors. The *_enable, *_grant, *_request signals are single-bit signals

wires are referenced to the edges of the bus clock. Besides the clock signal, additional control signals can be used to synchronize a bus master and bus slave, for example to indicate time-outs and to support request-acknowledge signalling.

10.1.3 Elements in a Point-to-Point Bus

For dedicated connections, bus systems may also support a point-to-point communication mode. Figure 10.2 illustrates such a point to point connection. Like a shared bus, a point-to-point bus also identifies a master and a slave. There is no concept of address space. Instead, the data is looked upon as an infinite stream of items. Data items may still be assigned to logical *channels*. This enables multiple streams to be multiplexed over the same physical channel. Each data transfer will then consist of a tuple (datavalue, channel).

The point-to-point bus has synchronization wires to handle the interaction between master and slave. The synchronization mechanism is very similar to the one we described in the previous chapter (Sect. 9.2.3).

10.1.4 Physical Implementation of On-Chip Busses

Figure 10.3 shows the physical layout of a typical on-chip bus segment with two masters and two slaves. The AND and OR gates in the center of the diagram serve as multiplexers. Several signals are merged this way into bus-wide address and data signals. For example, the address generated by the bus masters is merged into a single bus address, and this bus address is distributed to the bus slaves. Similarly, the data transfer from the masters to the slaves is merged into a bus-wide write-data signal, and the data transfer from the slaves to the masters is merged into a bus-wide read-data signal. This implementation is typical for on-chip bus systems. Due to the low cost of an on-chip wire, multiplexing is not done for the purpose of saving silicon real-estate, but for the purpose of logical merging of signals. Notice, for example, how the bus system implements a separate bundle of wires for reading data and for writing data. This is common for each of the four bus systems discussed earlier.

The convention that associates the direction of data with *reading* and *writing* the data is as follows. Writing data means: sending it from a master to a slave. Reading data means: sending it from a slave to a master. This convention affects the input/output direction of bus signals on slave components and master components.

In Fig. 10.3, each master generates its own bus-enable signal in order to drive a data item or an address onto the bus. For example, when `Master1` will write data, `m1_enable` will be high while `m2_enable` will be low. If both enable signals would be high, the resulting bus address and write-data will be undefined. Thus, the bus protocol will only work when the components collaborate and follow the rules of the protocol. The bus arbiter has to ensure that only one bus master at a time takes control of the bus. Bus arbitration can be done globally, for an entire bus segment, or it can be done per slave component. We will describe bus arbitration further in the context of Multi-Master Systems (Sect. 10.3).

10.1.5 Bus Naming Convention

Since a bus segment can group a potentially large amount of signals, bus systems will follow a *naming convention*. The objective of a naming convention is to infer the functionality and connectivity of a wire based on its name. For example, a naming convention is very helpful to read a timing diagram. A naming convention can also help engineers to visualize the connectivity in a (textual) netlist of a circuit.

A component pin name will reflect the functionality of that pin. Each bus standard has its own favorite names for address signals, data signals, control signals and synchronization signals. For example, the IBM/Coreconnect bus uses `ABUS` for the address bus, AMBA uses `PADDR` or `HADDR`, Avalon uses `address` and Wishbone uses `AD`. Using common pin names allows a designer to easily recognize the interfaces on a hardware module.

Bus signals are created by interconnecting component pins. Bus signals follow a convention, too. The key issue is to avoid confusion between similar signals. For example, in Fig. 10.3, there are two master components, each with a `wdata` signal. To distinguish these signals, the component instance name is included as a prefix in bus signal name, such as `m2_wdata`. Other bus systems use this technique as well. In some cases, both the master instance name and the slave instance name will be included as part of the bus signal name. This results in very long identifiers,

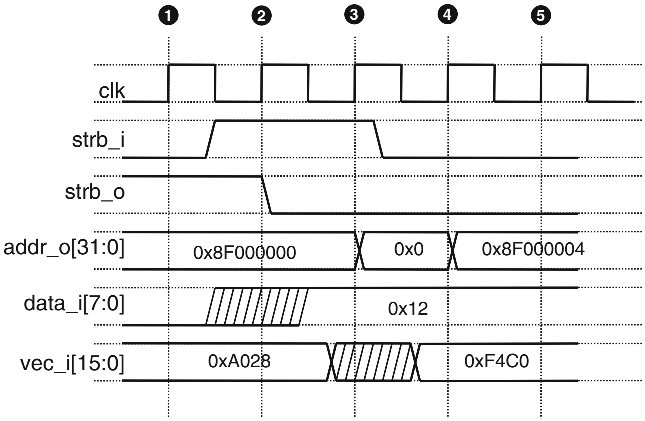


Fig. 10.4 Bus timing diagram notation

such as `cpu1_read_data_valid_sdram0`. While such identifiers are tedious to type, they precisely reflect the function of the wire.

Bus systems may use additional naming conventions to help a designer. For example, AMBA prefixes all signals on the high-speed bus (AHB) with the letter `H`, and all signals on the peripheral bus (APB) with the letter `P`. Wishbone suffixes all input pins with the letter `I`, and all output pins with the letter `O`. Obviously, for a designer, it will greatly help to adopt and follow the bus naming conventions of the particular bus in use: it avoids coding mistakes and connection errors, it makes code reusable, and it simplifies debugging and verification.

10.1.6 Bus Timing Diagram

Because a bus system reflects a complex, highly parallel entity, timing diagrams are extensively used to describe the timing relationships of one signal to the other. Figure 10.4 illustrates a timing diagram of the activities in a generic bus over five clock cycles. The clock signal is shown on top, and all signals are referenced to the upgoing clock edge. Dashed vertical lines indicate the timing reference.

Signal buses of several wires can be collapsed into a single trace in the timing diagram. Examples in Fig. 10.4 are `addr_o`, `data_i`, and `vec_i`. The label indicates when the bus changes value. For example, `addr_o` changes from `0x8F000000` to `0x00000000` at the third clock edge, and it changes back to `0x8F000004` one clock cycle later. Various schemes exist to indicate that a signal or a bus has an unknown or don't care value. The value of `data_i` at the second clock edge, and the value of `vec_i` at the third clock edge are all unknown.

When discussing timing diagrams, one must make a distinction between clock edges and clock cycles. The difference between them is subtle, but often causes confusion. The term *clock cycle* is ambiguous, because it does not indicate a singular point in time: a clock cycle has a beginning and an end.

A *clock edge*, on the other hand, is an atomic event that cannot be partitioned further (at least not under a single-clock synchronous paradigm). A consequence of the ambiguous term *clock cycle* is that the meaning of the term changes with the direction of the signals. When discussing the value of input signals, designers usually mean to say that these signals must be stable at the start of the clock cycle, just *before* a clock edge. When discussing the value of output signals, designers usually talk about signals that are stable at the end of the clock cycle, so *after* a clock edge. Consider for example signal `strb_o` in Fig. 10.4. The signal goes down just after the clock edge labeled `2`. As `strb_o` is an output signal, a designer would say that the signal is low in clock cycle 2: the output should reach a stable value after clock edge `2`. In contrast, consider the signal `strb_i`. This input signal is high at the clock edge labeled 2. Therefore, a designer would say this input is high in clock cycle 2. This means that the signal should reach a stable value before clock edge `2`. To avoid this ambiguity, we will discuss timing diagrams in terms of clock edges rather than clock cycles.

Bus timing diagrams are very useful to describe the activities on a bus as a function of time. They are also a central piece of documentation for the design of a hardware-software interface.

10.1.7 Definition of the Generic Bus

Even though different bus systems may use a different naming convention for signals, they have a similar functionality. This chapter is not written towards any specific bus system, but instead emphasizes the common concepts between them. We will therefore define a generic bus. Where appropriate, we will indicate the relationship of this generic bus with a specific implementation on CoreConnect, AMBA, Avalon or Wishbone. The signals that make up the generic bus are listed in Table 10.2. The exact meaning of these signals will be explained throughout this chapter.

Table 10.3 makes a comparison between the signal names of the generic bus, and equivalent signals on the CoreConnect/OPB bus, the AMBA/APB bus, the Avalon-MM bus, and the Wishbone bus.

10.2 Bus Transfers

In this section, we will discuss several examples of data transfers between a bus master and a bus slave. We will also discuss common strategies used by on-chip busses to improve the overall system performance.

Table 10.2 Signals on the generic bus

Signal name	Meaning
clk	Clock signal. All other bus signals are references to the upgoing clock edge
m_addr	Master address bus
m_data	Data bus from master to slave (write operation)
s_data	Data bus from slave to master (read operation)
m_rnw	Read-not-Write. Control line to distinguish read from write operations
m_sel	Master select signal, indicates that this master takes control of the bus
s_ack	Slave acknowledge signal, indicates transfer completion
m_addr_valid	Used in place of m_sel in split-transfers
s_addr_ack	Used for the address in place of s_ack in split-transfers
s_wr_ack	Used for the write-data in place of s_ack in split-transfers
s_rd_ack	Used for the read-data in place of s_ack in split-transfers
m_burst	Indicates the burst type of the current transfer
m_lock	Indicates that the bus is locked for the current transfer
m_req	Requests bus access to the bus arbiter
m_grant	Indicates bus access is granted

Table 10.3 Bus signals for simple read/write on Coreconnect/OPB, ARM/APB, Avalon-MM and Wishbone busses

generic	CoreConnect/OPB	AMBA/APB	Avalon-MM	Wishbone
clk	OPB_CLK	PCLK	clk	CLK_I (master/slave)
m_addr	Mn_ABUS	PADDR	Mn_address	ADDR_O (master) ADDR_I (slave)
m_rnw	Mn_RNW	PWRITE	Mn_write_n	WE_O (master)
m_sel	Mn_Select	PSEL		STB_O (master)
m_data	OPB_DBUS	PWDATA	Mn_writedata	DAT_O (master) DAT_I (slave)
s_data	OPB_DBUS	PRDATA	Mb_readdata	DAT_I (master) DAT_O (slave)
s_ack	Sl_XferAck	PREADY	Sl_waitrequest	ACK_O (slave)

10.2.1 Simple Read and Write Transfers

Figure 10.5 illustrates a write transfer on a generic peripheral bus. A bus master will write the value 0xF000 to address 0x8B800040. We assume that this bus only has a single bus master and that it does not need arbitration. On clock edge 2, the master takes control of the bus by driving the master select line m_sel high. This indicates to the bus slave that a bus transaction has started. Further details on the nature of the bus transaction are reflected in the state of the bus address m_addr and the bus read/write control signal m_rnw. In this case, the transfer is a write, so the *read-not-write* (m_rnw) signal goes low.

Bus requests from the master are acknowledged by the slave. A slave can extend the duration of a transfer in case the slave cannot immediately respond to the request of a master. In Fig. 10.5, the slave issues an acknowledge signal s_ack on clock

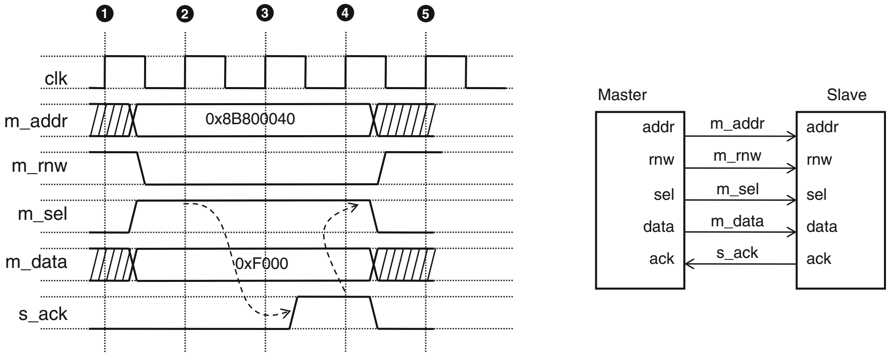


Fig. 10.5 Write transfer with one wait state on a generic peripheral bus

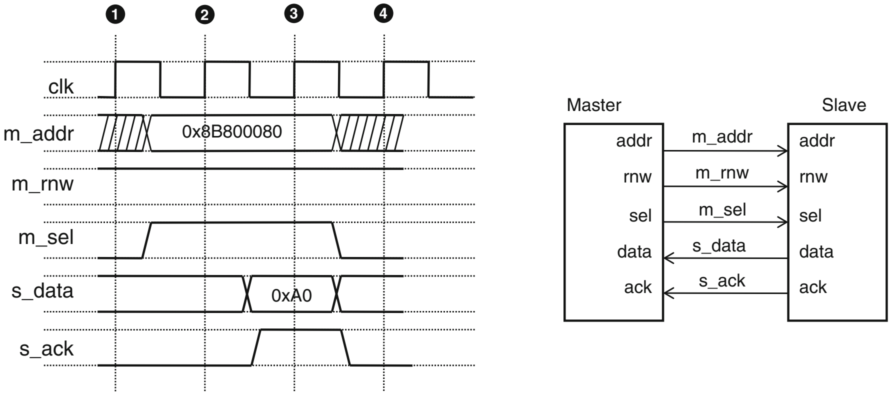


Fig. 10.6 Read transfer with no wait state on a generic peripheral bus

edge 4. This is one clock cycle later than the earliest possible clock edge 3. Such a cycle of delay is called a *wait state*: the bus transaction is extended for one clock cycle. Wait states enable communication between bus components of very different speed. However, wait states are also a disadvantage. During a wait state, the bus is tied-up and inaccessible to other masters. In a system with many slow slaves, this will significantly affect the overall system performance. A *bus timeout* can be used to avoid that a slave completely takes over a bus. If, after a given amount of clock cycles, no response is obtained from the bus slave, the bus arbiter can declare a timeout condition. The timeout condition will alert the bus master to give up the bus and abort the transfer.

Figure 10.6 shows a read transfer with no wait states. The protocol is almost identical as a write transfer. Only the direction of data is reversed (from slave to master), and the `m_rnw` control line remains high to indicate a read transfer. The bus protocols for read and write described here are typical for peripheral buses.

Table 10.4 Signal naming and numbering for a bus slave input

Signal part	Offset	CoreConnect/OPB	AMBA/APB
Word		`Sl_DBUS[0..31]`	`PWDATA[31..0]`
Most significant bit		`Sl_DBUS[0]`	`PWDATA[31]`
Little endian byte	0	`Sl_DBUS[24..31]`	`PWDATA[7..0]`
Big endian byte	0	`Sl_DBUS[0..7]`	`PWDATA[31..24]`
Little endian byte	3	`Sl_DBUS[0..7]`	`PWDATA[31..24]`
Big endian byte	3	`Sl_DBUS[24..31]`	`PWDATA[7..0]`
Signal part	**Offset**	**Avalon-MM**	**Wishbone**
Word		`writedata[31..0]`	`DAT_I[31..0]`
Most significant bit		`writedata[31]`	`DAT_I[31]`
Little endian byte	0	`writedata[7..0]`	`DAT_I[7..0]`
Big endian byte	0	`writedata[31..24]`	`DAT_I[31..24]`
Little endian byte	3	`writedata[31..24]`	`DAT_I[31..24]`
Big endian byte	3	`writedata[7..0]`	`DAT_I[7..0]`

10.2.2 Transfer Sizing and Endianess

By default, all masters and slaves on an on-chip bus will use a uniform wordlength and a uniform endianess. For example, the masters, the slaves, and the bus could be using 32-bit little-endian words. This would mean that each data transfer transports 32 bits, and that the least significant byte would be found in the lower byte of the 32-bit word. As long as the master, the bus, and the slave make identical assumptions on the data format, a single request and a single acknowledge signal will be adequate to control the transfer of data.

The numbering of wires within a signal vector depends on the bus. The documentation of the bus should be consulted to determine the name of the least significant bit of a word. As an example, Table 10.4 illustrates the signal naming for a bus slave input under various bus schemes.

While endianess can be configured on most buses, it is a static selection, and dynamic switching of endianess is not supported by AMBA, Coreconnect, Avalon-MM or Wishbone. The additional hardware and complexity introduced in the bus system does not justify the benefit. Indeed, as illustrated in Fig. 10.7, heterogeneous endianess can be resolved while interconnecting bus components to the bus system.

A bus system will also need to provide a mechanism for *transfer sizing*: selecting what part of a given word belongs to the actual data transfer. In a 32-bit data-bus for example, it is useful to be able to transfer a single byte or a halfword (16 bit). For example, this would allow a C program to write a single char (8 bit) to memory. Transfer sizing is expressed using byte-enable signals, or else by directly encoding the size of the transfer as part of the bus control signals. The former method, using byte-enable signals, is slightly more general than the latter, because it allows one to cope with un-aligned transfers.

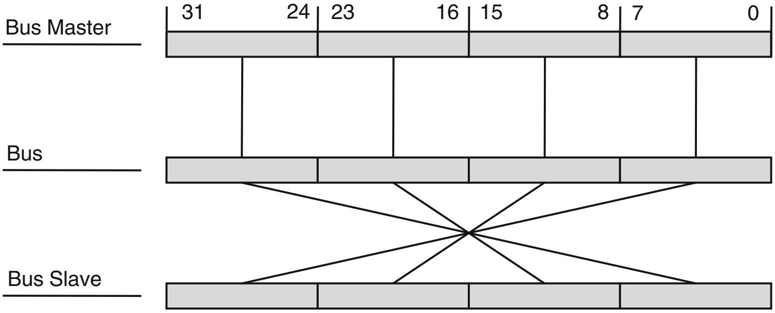


Fig. 10.7 Connecting a big-endian slave to little-endian master

To see the difference between the two, consider the difference in performance for a processor running the following C program.

```
int main() {
  unsigned i;
  char a[32], *p = a;

  for (i=0; i<32; i++)
    *p++ = (char) (i + 4);

  return 0;
}
```

As the processor moves through all iterations of the i-loop, it will generate *byte-aligned* write operations to all addresses occupied by the a array. Assume that this happens in a system with a 32-bit data bus, and that the third byte of a 32-bit word needs to be written during a particular iteration. When the bus does not support unaligned data transfers, the processor will first need to *read* the word that contains the byte, update the word by modifying a single byte, and write it back to memory. On the other hand, when the bus does support unaligned data transfers, the processor can directly write to the third byte in a word. Therefore, the example program will complete quicker on systems that support unaligned transfers. Note that unaligned transfers can also lead to exceptions. For example, processors with a word-level memory organization do not support transfer of unaligned words. If a programmer attempts to perform such a transfer, an exception will result, which usually halts the execution of the program.

Endianess and byte-transfer sizing help bus components to deal with the ordering of individual bytes within a bus word. In some cases, we may run into the situation where masters and slaves of different physical width need to be interconnected. For example, a bus slave could have an 8-bit data bus but needs to be connected to a 32-bit bus. Or, a bus master could have a 64-bit data bus but needs to be connected to a 32-bit bus. These cases require additional hardware.

Figure 10.8 shows how a 64-bit bus slave and a 16-bit bus slave can be connected to a 32-bit bus. In the case of the 64-bit bus slave, a data-write will transfer only

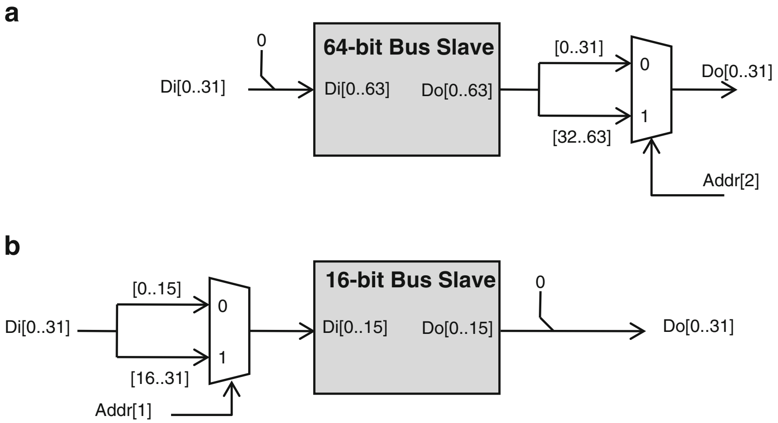


Fig. 10.8 (**a**) Connecting a 64-bit slave to a 32-bit bus. (**b**) Connecting a 16-bit slave to a 32-bit bus

32 bits at a time; the upper 32 bits are wired to zero. Additionally, the interconnect may use the byte-enable signals to mask off non-written bytes to the slave. In the case of a data-read, one of the address lines, `Addr[2]`, will be used to multiplex the 64 bits of data produced by the bus slave. The net effect of the multiplexing is that the bus slave appears as a continuous memory region when data is read.

The case of the 16-bit bus slave is opposite: the 32-bit bus system can deliver more data than the 16-bit bus slave can handle, and an additional address bit, `Addr[1]` is used to determine which part of the 32-bit bus will be transferred to the 16-bit bus slave. If the interconnect supports *dynamic* transfer sizing, it will expand each master-side transaction (read or write) into multiple slave-side transactions.

In summary, busses are able to deal with varying wordlength requirements by the introduction of additional control signals (byte-select signals), and by adding additional multiplexing hardware around the bus slaves or bus masters. A designer also needs to be aware of the endianess assumptions made by the on-chip bus, the bus master, and the bus slave.

10.2.3 Improved Bus Transfers

Each bus data transfer includes multiple phases. First, the bus master has to negotiate a bus access with the bus arbiter. Next, the bus master has to issue a bus address and a bus command. Third, the bus slave has to acknowledge the data transfer. Finally, the bus master has to terminate the bus transfer and release control over the bus. Each of these activities takes a finite amount of time to complete. Moreover, all of these activities are sequential, so that the overall system is limited by the speed of the slowest component. For high-speed buses, this is too slow.

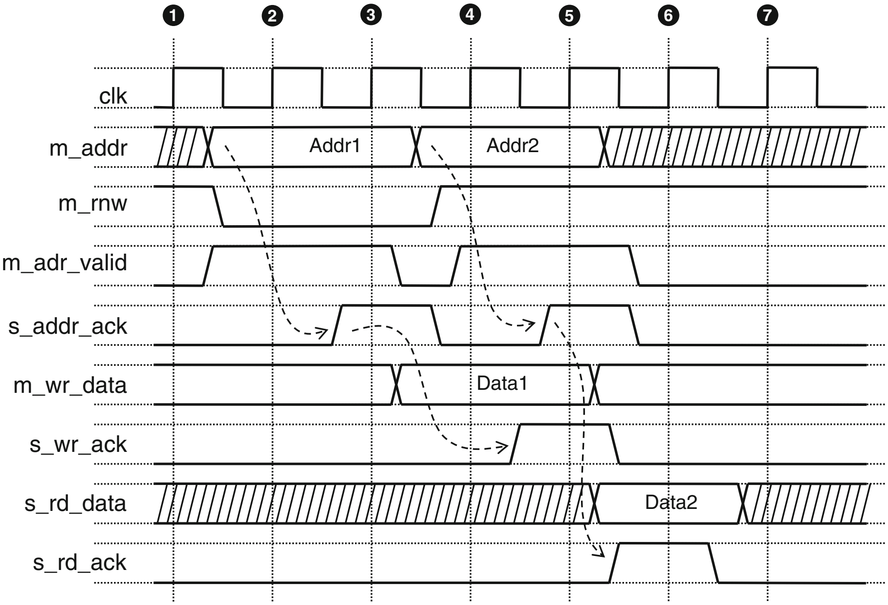


Fig. 10.9 Example of pipelined read/write on a generic bus

On-chip buses use three mechanisms to speed up these transfers. The first mechanism, *transaction-splitting*, separates each bus transaction in multiple phases, and allows each phase to complete separately. This prevents locking up the bus over an extended period of time. The second mechanism, *pipelining*, introduces overlap in the execution of bus transfer phases. The third mechanism, *burstmode operation*, enables transfer of multiple data items, located at closely related addresses, during a single bus transaction.

A bus may use one or several of these mechanisms at the same time. Two of them, transaction splitting and pipelined transfers, often occur together. We will discuss them simultaneously.

10.2.3.1 Transaction Splitting and Pipelined Transfers

In *transaction splitting*, a single bus transfer is decomposed into separate phases. For example, the high-speed bus implementation of AMBA (AHB) and CoreConnect (PLB) treat the transfer of an address as a separate transaction from the transfer of data. Each of these transfers has a different acknowledge signal. The rationale is that a bus slave will need some time after the reception of an address in order to prepare for the data transfer. Hence, after the transfer of an address, the bus master should be released from waiting for the bus slave, and carry on.

Transaction pipelining occurs when multiple transactions can proceed simultaneously, each at a different level of progress. Split transactions are good candidates for overlapped execution. Figure 10.9 gives an example of overlapped read/write transfers for a generic bus. Two transfers are shown in the figure: a write followed by a read. The bus used in this figure is slightly different from the one used in Figs. 10.5 and 10.6. The difference is that there are *two* acknowledge signals rather than a single one. On clock edge 2, the bus master indicates a write to address `Addr1`. The bus slave acknowledges this address on clock edge 3. However, at that moment the data transfer is not yet completed. By acknowledging the address, the slave merely indicates it is ready to accept data. From clock edge 4, the bus master executes two activities. First, it sends the data to be written, `Data1`, to the bus slave. Then, it initiates the next transfer by driving a new address `Addr2` on the bus. On clock edge 5, two events take place: the bus slave accepts `Data1`, and it also acknowledges the read-address `Addr 2`. Finally, on clock edge 6, the bus slave returns the data resulting from that read operation, `Data2`.

Thus, through multiple control/status signals, the bus masters and bus slaves are able to implement bus transfers in an overlapped fashion. Obviously, this will require additional hardware in the bus interface for the master and the slave. Both the AMBA/AHB and the Coreconnect/PLB support overlapped and pipelined bus transfers, although the detailed implementation of the protocol on each bus system is different. The Avalon bus and the Wishbone bus support pipelined transfers, but there are no separate acknowledge signals for address and data transfer.

10.2.3.2 Burstmode Transfers

The third technique to improve bus transaction performance is to use *burstmode transfers*. This will transfer multiple data items from closely related addresses in one bus transaction.

Burstmode transfers are useful when transfers of closely-related data items are required. The main memory accesses made by a processor with cache memory are an example. When there is cache miss on the processor, an entire cache *line* needs to be replaced. Assume that there would be 32 bytes in a cache line, then a cache miss implies reading 32 consecutive bytes from memory.

Burst-mode transfers can have a fixed or a variable length. In a fixed-length burst-mode scheme, the bus master will negotiate the burst properties at the start of the burst transfer, and next perform each transfer within the burst. In a variable-length scheme, the bus master (or the bus slave) has the option of terminating the burst after every transfer. The addresses within a burst are usually incremental, although there are also applications where the address needs to remain constant, or where the address increments with a modulo operation. Thus, as part of the burst specification, a bus may allow the user to specify the nature of the burst address sequence.

Finally, the address step will depend on the size of the data within the burst: bytes, halfwords and words will increment addresses by 1, 2, and 4 respectively. Obviously, all of these options involve adding extra control-signals on the bus, at

Table 10.5 Burst transfer schemes

Burst property	CoreConnect/OPB	AMBA/APB
Burst length	Fixed (2 .. 16) or variable	Fixed (4, 8, 16) or variable
Address sequence	Incr	Incr/mod/const
Transfer size	Fixed by bus	Byte/halfword/word
Burst property Avalon-MM	Wishbone	
Burst length	Fixed (1 .. 2^{11})	Variable
Address sequence	Incr/mod/const	Incr/const
Transfer size	Byte/halfword/word	Byte/halfword/word

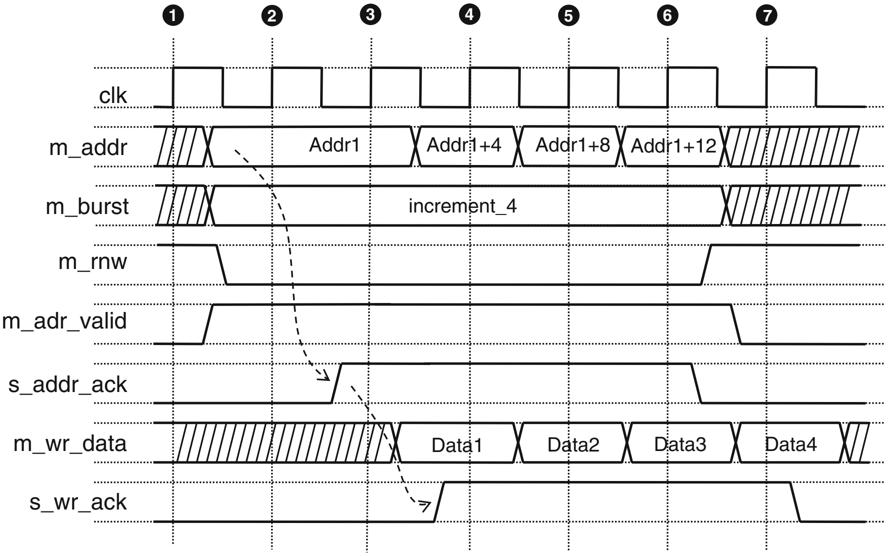


Fig. 10.10 A four-beat incrementing write burst

the side of the master as well as the slave. Table 10.5 shows the main features for burst-mode support on Coreconnect, AMBA, Avalon, and Wishbone.

An example of a burst-mode transfer is shown in Fig. 10.10. This transfer illustrates a burst transfer of four words in adjacent address locations. Besides the commands discussed before (`m_addr`, `m_rnw`, `m_adr_valid`), a new command `m_burst` is used to indicate the type of burst transfer performed by the master. In this generic bus, we assume that one of the burst types is encoded as `increment_4`, meaning a burst of four consecutive transfers with incrementing address. On clock edge 3, the slave accepts this transfer, and after that the master will provide four data words in sequence. The address information provided by the master after the first address is, in principle, redundant. The addresses are implied from the burst type (`increment_4`) and the address of the first transfer. Figure 10.10 assumes that the wordlength of each transfer is 4 bytes (one word).

Therefore, the address sequence increments by 4. The scheme in this figure is similar to the scheme used by AMBA/AHB. The Coreconnect/PLB system is slightly more general (and as a consequence, more complicated), although the ideas of burst transfers are similar to those explained above.

This completes our discussion on improved bus data transfers. As demonstrated in this section, there are many variations and enhancements possible for data transfer over a bus. Optimal bus performance requires both the master as well as the slave to be aware of all features provided by a bus protocol. For the hardware-software codesigner, understanding the bus protocols is useful to observe the hardware-software communication at its lowest abstraction level. For example, it is very well possible to associate the behavior of a C program with the data transfers observed on a bus. The Problems will explore this issue in further detail.

So far, we made the implicit assumption that there is only a single master on the bus. In the next section, we will discuss the concepts of bus arbitration, when there are multiple masters on the bus.

10.3 Multi-master Bus Systems

When there is more than a single master on a bus, each bus transfer requires negotiation. A *bus arbiter* will control this negotiation process and allocate each transfer slot to a bus master.

Figure 10.11 shows the topology of a (generic) multi-master bus with two masters and an arbiter circuit. The slaves are not shown in the figure. Of the regular bus features, only the address bus, and a transfer-acknowledge signal are visible. Each master can request access to the bus through the request connection. The arbiter uses grant to indicate the master that it can access the bus. Once a master has

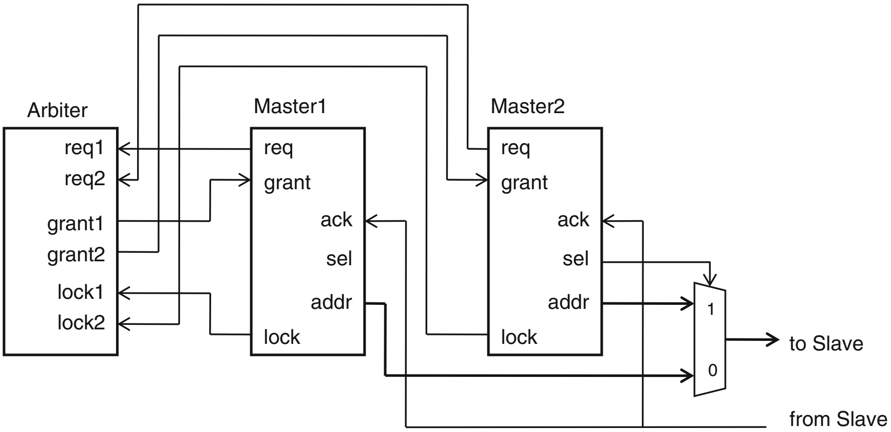


Fig. 10.11 Multi-master arbitration

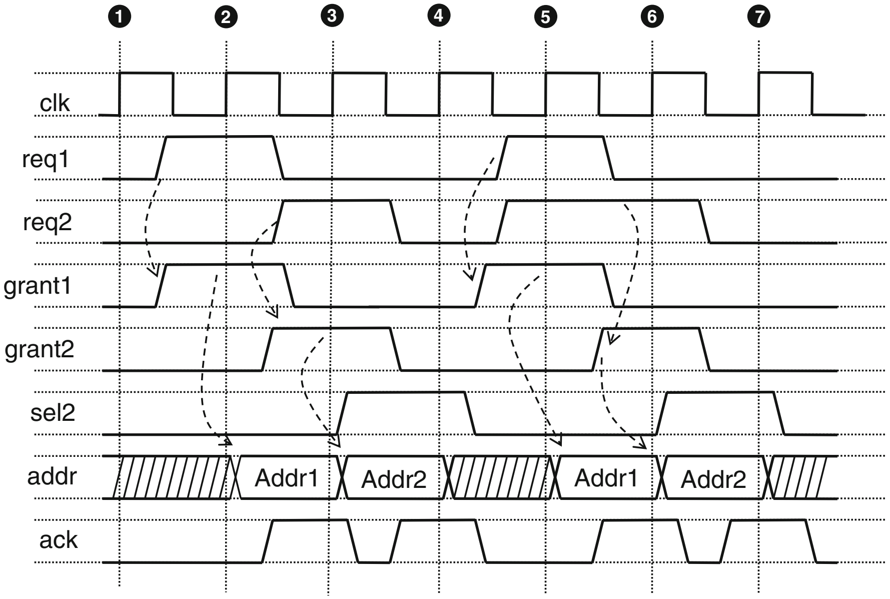


Fig. 10.12 Multi-master arbitration timing

control over the bus, it will proceed through one of the regular bus transfer schemes as discussed before. The `lock` signals are used by a master to grab exclusive control over the bus, and will be clarified later.

Figure 10.12 shows how two masters compete for the bus over several clock cycles. On clock edge 2, master 1 requests the bus through `req1`. Since master 2 does not need the bus at that moment, the arbiter will grant the bus to master 1. Note that the `grant` signal comes as an immediate response to the `request` signal. This means that the bus negotiation process can complete within a single clock cycle. In addition, it implies that the arbiter will need to use combinational logic to generate the `grant` signal based on the `request` signal.

After clock edge 2, master 1 drives an address onto the address bus and completes a regular bus transfer. We assume that the slave acknowledges the completion of this transfer on clock edge 3, by pulling `ack` high. The earliest time when the next arbitration for a bus transfer takes place is clock edge 3. This is called an *overlapping* arbitration cycle, because the arbitration of the next transfer happens at the same moment as the completion of the current transfer. The second transfer is granted to master 2, and completes on clock edge 4.

Between clock edges 4 and 5, the bus sits idle for one cycle, because no master has requested access to the bus. On clock edge 5, both masters 1 and 2 request access to the bus. Only one master is allowed to proceed, and this means that there is a *priority resolution* implemented among the masters. In this case, master 1 has fixed priority over master 2, which means that master 1 will always get access to the

Table 10.6 Arbitration signals on CoreConnect/OPB and AMBA/AHB

Signal	CoreConnect/PLB	AMBA/AHB	Avalon-MM
reqx	Mx_request	HBUSREQ	Mx_transfer_request
grantx	PLB_PAValid	HGRANT	Mx_waitrequest
lock	Mx_Buslock	HLOCK	lock
	PLB_Buslock	HMASTLOCK	
	Mx_priority[..]		
sel		HMASTER[..]	

bus, and master 2 will get access to the bus only when master 1 does not need it. The transfer of master 1 completes at clock edge 6. Since master 2 is still waiting for access to be granted, it can proceed at clock edge 6 because master 1 no longer needs to bus. The fourth and final transfer then completes on clock edge 7.

10.3.1 Bus Priority

The timing diagram in Fig. 10.12 reveals the interesting concept of priority. When multiple masters attempt to access the bus at the same time, only a single master is allowed to proceed based on priority resolution. The simplest priority scheme is to allocate a fixed priority, strictly increasing, to every master. While this is easy to implement, it is not necessarily the best solution. When a high-priority master continuously accesses the bus, other low-priority masters can be denied bus transfers for extended amounts of time.

Often, multiple masters on a bus should have equal priority. For example, when symmetrical multi-processors processors on a single bus access the same memory, no processor should have priority over the other. In this situation, priority resolution is implemented using *round-robin* scheme. Each master takes turns to access to the bus. When two masters request the bus continuously, then the bus transfers of master 1 and master 2 will be interleaved. Another possible solution is a *least-recently-used* scheme, in which the master who was waiting for the bus for the longest time will get access first. Equal-priority schemes such as round-robin or least-recently-used avoid *starvation* of the bus masters, but, they also make the performance of the bus unpredictable. When working with latency-critical applications, this can be a problem. To address this, designers can use a mixed scheme that combines multiple levels of priority with an equal-priority scheme to allow several masters to share the same priority level.

The priority algorithm used by the bus arbiter is not part of the definition of the bus transfer protocol. Therefore, Coreconnect/PLB, AMBA/AHB and Avalon-MM only describe the arbitration connections, but not the priority schemes. The Wishbone bus is special in that it does not define special bus request/grant signals. Instead, Wishbone leaves the design of the bus topology to the designer.

Table 10.6 makes a comparison between the generic bus arbtriation signals defined above, and those of CoreConnect, AMBA and Avalon. The table also lists a few arbitration signals that are unique to each individual bus protocol. The bus locking signals will be explained shortly. The other signals have the following meaning.

- Mx_priority[..] allows a PLB master to select its priority for each transfer. This scheme allows the master to change its priority level dynamically depending on the needs of the bus transfer.
- HMASTER[..] is an encoding of the identity of the master that was granted bus access by the arbiter. The signal is used to drive the bus address multiplexer.

10.3.2 Bus Locking

The final concept in multi-master bus schemes is *bus locking*: the exclusive allocation of a bus to a single master for the duration of multiple transfers. There are several reasons why bus locking may be needed. First, when large blocks of data need to be transferred with strict latency requirements, exclusive access to the bus may be required. While burst-mode transfers can help a master to complete these transfers quickly, these transfers can still be interrupted by another master with higher priority. By locking the bus, the master can be sure this will not happen.

The second need for bus locking is when a master needs to have guaranteed, exclusive access to consecutive transfers, typically a read transfer followed by a write transfer. This is needed, for example when implementing a *test-and-set* instruction. The test-and-set instruction is used to create a mutex, a well-known software primitive to implement mutual exclusion. A mutex is similar to a semaphore (discussed in Sect. 9.2.2), but it's not identical to it. A mutex implies ownership: once a mutex is locked, it can only be unlocked by the same entity that locked it. An example where a mutex can be used is to control the access of two bus masters to a single, shared region of memory. Bus masters will control access using a mutex, implemented through a test-and-set instruction as discussed next.

An example implementation of test-and-set is shown below. This C program runs on each of two processors (bus masters) attached to the same bus. They share a memory location at address 0x8000. By calling testandset, a processor will try to read this memory location, and write into it during a single locked-bus operation. This means that the function test_and_set() cannot be interrupted: only one processor will be able to read the value of the mutex when its value is low. The two processors use this function as follows. Before accessing the shared resource, the processors will call enter(), while they will call leave(). The shared resource can be anything that needs exclusive access by one of the processors.

```
int *mutex = (int *) 0x8000; // location of mutex
```

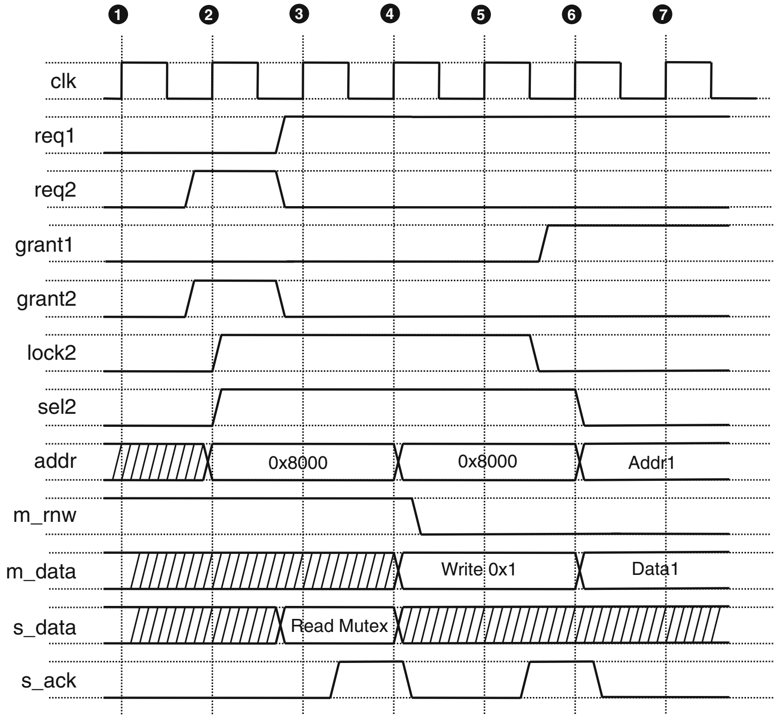


Fig. 10.13 Test-and-set operation with bus locking

```
int test_and_set() {
  int a;
  lock_bus();
  a = *mutex;
  *mutex = 1;
  unlock_bus();
  return a;
}

void leave() {
  *mutex = 0;
}

void enter() {
  while (test_and_set()) ;
}
```

Figure 10.13 shows an example of `test-and-set` with bus-locking. On clock edge 2, master 2 requests access to the bus using `req2`. This access is granted by the arbiter through `grant2`. After clock edge 2, this master grabs the bus using `sel2` and locks it using `lock2`. Master 2 will now perform a test-and-set operation, which involves a read of a memory address immediately followed by a write to the same memory address. The read operation starts on clock edge 3 and completes on clock edge 4. On clock edge 3, the master drives an address onto the bus and signals a read operation (m_rnw). On clock edge 4, the slave delivers the data stored at this address, and completes the transfer using s_ack.

Meanwhile, another master requested bus access starting on clock edge 3 (using req1). However, because master 2 has locked the bus, the arbiter will ignore this request. Master 2 will be granted further use of the bus until it release the lock. This access is guaranteed even if master 2 has a lower priority than other masters requesting the bus.

After performing the reading part of the test-and-set instruction, master 2 will now write a '1' into the same location. At clock edge 5, master 2 puts the address and the data on the bus, and at clock edge 6 the slave accepts the data. The lock can be released after clock edge 5. Note that, should the write operation to the slave fail, then the complete test-and-set instruction has failed. We assume however that the write operation completes correctly. As soon as master 2 releases lock2, control will go to master 1 because of the pending request on req1. As a result, starting with clock edge 6, a new bus transfer can start, which is allocated to master 1.

Wrapping up, when multiple masters are attached to a single bus, individual bus transfers need to be arbitrated. In addition, a priority scheme may be used among masters to ensure latency requirements for particular masters. Finally, bus-locking can be used to implement guaranteed access for an extended amount of time. Since all of these techniques are implemented in hardware, at the level of a bus transfer, they are very fast and efficient. As such, bus systems play an important role in building efficient hardware-software communication.

10.4 Bus Topologies

The bus topology is the logical and physical organization of bus components in a network. So far, we have used linear bus topologies: busses that consist of bus segments interconnected by bus bridges.

The bus topology has a major impact on system performance. Let's first consider a single bus segment. All modules connected to the same bus segment are sharing the same communication resource. This means that the communication among these modules needs to be sequentialized. Two bus masters on the same bus segment cannot initiate parallel bus transfers; they need bus arbitration to sequentialize their access.

Bus bridges can split busses in multiple segments, and are used to group bus components of similar performance. You can think of them as partitioning a road in multiple lanes, to accommodate fast traffic as well as slow traffic. However, bus bridges do not solve the issue of parallel bus transfers. Bus bridges introduce an

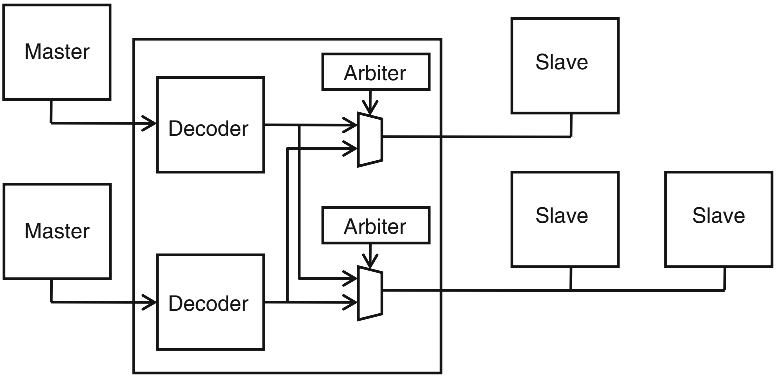


Fig. 10.14 A multi-layer bus

implicit hierarchy among bus segments: a bus bridge is a master on one side and a slave on the other. In many cases, for example in multi-processor architectures, a hierarchy among the processors is not wanted or even counter-productive.

Besides the logical constraints of a bus segments, there are also significant technological issues. Implementing very long wires on a chip is hard, and distributing high-frequency signals and clocks using such wires is even harder. The power consumption of a wire is proportional to the length of the wire and the switching frequency of the signals on that wire. Hence, long wires will consume significantly more power than small, local wires. Chips which are organized using a linear bus topology will consume more energy for the same task than chips organized using a distributed topology.

Clearly, building on-chip bus communications with segmented busses has its limitations. Customizing the bus topology, and matching it to the application, is a logical improvement to this.

10.4.1 Bus Switches

A bus segment creates a static association between the bus masters and bus slaves attached to that segment. Let's see why a static assignment of bus masters to bus segments can be a problem. Assume that a master is connected to bus segment *A*. All slaves that need to communicate with this master will need to attach to *A* as well, or else to a segment which is directly bridged from *A*. Furthermore, all masters that need to talk to *any* of the slaves attached to *A* or any bridged segment on it, need to connect to bus segment *A* as well. Hence, all masters and all slaves in the system will tend to cluster to the same bus segment *A*. In simple, single-master systems, this is no problem. In heterogeneous architectures or architectures with multiple cores, however, a single segmented bus quickly becomes a performance bottleneck.

The solution to this problem is to use *bus switching* or *bus layering*, which makes the association between masters and slaves flexible. Figure 10.14 is an example of a two-layer bus. The two bus masters each connect to a bus switch with two output segments. There are three slaves connect to the output segments. The masters can connect to either of the slaves on the output segments. A master transfer is initially decoded to determine what output segment should receive it. All requests for the same output segment are then merged and arbitrated. Simultaneous transfers to the same output segment will be sequentialized: the master at the input segment will not be able to complete the transfer until the master at the other input segment is done.

Multiple masters can still share the same input segment on a bus switch, which makes bus switching compatible with multi-master bus arbitration. Bus systems that have implemented bus switching are for example AMBA/AHB and an improved version of Avalon, Merlin.

The most evolved form of bus switch is an implementation in which each master has its own input segment on the switch, and each slave has its own output segment. Such an implementation is called a *cross-bar*. The cross-bar is a highly parallel, but very expensive implementation of on-chip interconnect. And, while it addresses the logical limits of bus segments, it does not address the electrical issues of them. A cross-bar is a global interconnect system, and it is not scalable.

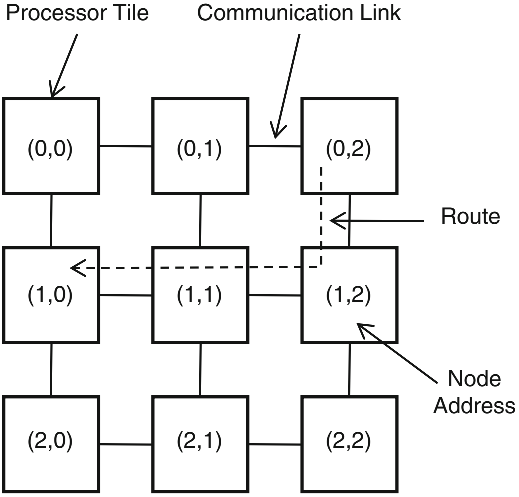
10.4.2 Network On Chip

Bus switches support a dynamic association of masters to slaves, but they have limited scalability. The fundamental issue with bus switches is that they maintain a tight association between a master and a slave. Indeed, each bus transaction is directly between a master and a slave, and the implementation of a bus transaction requires the interconnection network to create a pathway to complete it. Early telephone systems used the same concept: to connect a caller to a callee, a permanent circuit was established between the two parties for the duration of their telephone call.

The notion of a fixed and permanent pathway is however not scalable. Instead, the bus transfer itself must be implemented dynamically, in several steps. First, a master wraps a request into a packet. The master delivers the packet to the interconnect, and the interconnect finds a path from the master to the slave. Finally, the slave accepts the packet from the interconnect. When the slave is ready to issue a response, the opposite will happen: the slave delivers a packet to the interconnect, the interconnect returns the packet to the master, and the master accepts the response packet from the interconnect. Note the change of terminology from transaction to request/response and packet. The communication paradigm has changed from transactions to sending and receiving of packets. This new communication paradigm is called *Network-on-Chip*.

Figure 10.15 presents the concept of a Network-on-Chip. The computational elements of the chip and their interconnections are organized in a geometrical pattern, such as a matrix or a ring. The interconnections are made between the *tiles*

Fig. 10.15 A generic network-on-chip



of the network on chip. In Fig. 10.15, every tile can directly communicate with its neighbouring tiles. In addition, every tile has an address, symbolically indicated by the matrix indices. This allows any tile to communicate with any other tile. A *route* for the communication is selected, and a data packet travels through a number of *hops*, from a source tile to a destination tile over a number of intermediate tiles. Figure 10.15 illustrates a route from tile (0,2) to tile (1,0).

The design of a network on-chip, and its operation, introduces a challenging set of problems. At the basic level, the communication and data representation is very different from the approach used in on-chip buses. In a network-on-chip, data items are encapsulated in a *packet* before being transmitted. Once a packet leaves a source tile and travels to a destination tile, it needs to find a route. As can be seen in Fig. 10.15, a route is not unique. Between a given source tile and destination tile, many different routes are possible. Hence, one needs a routing algorithm that is able to select interconnection segments so that the overall level of congestion will remain small. The design of Network-on-Chip has been an intensive area of research during the past decade.

The CELL processor is a well known multiprocessor device that relies on network-on-chip technology to implement on-chip communications. The CELL combines eight regular processing components called *SPE* (synergistic processing element). In addition, there is a control processor called *PPE* (power processor element) and an off-chip interface unit. All of these components are connected to the same network on chip, called the *EIB* (element interconnect bus). As illustrated in Fig. 10.16, the EIB consists of four ring structures, each 16 bytes wide. The communication model of the CELL processors assumes that processors will work using local memory, and that communication is implemented by moving blocks of data from one local memory location to the other. A *Direct Memory Access* (DMA) unit is used to handle communication between the bus interface and the local memory.

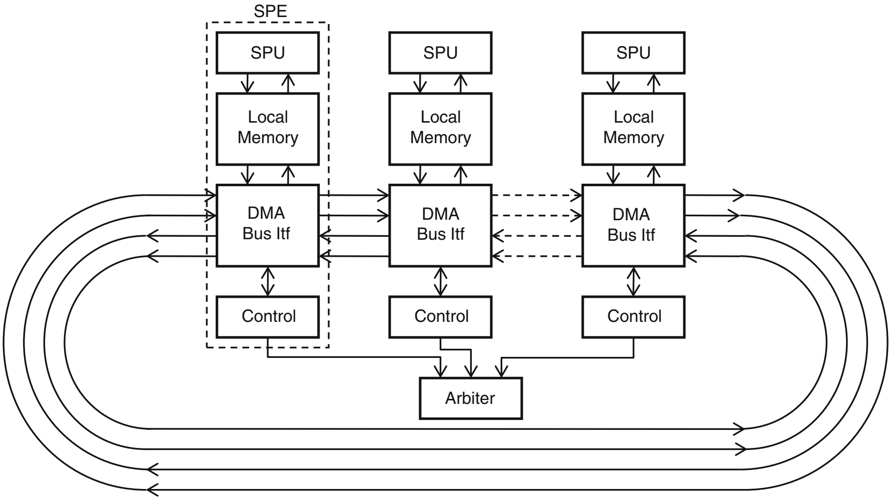


Fig. 10.16 On-chip network in the CELL processor

The four rings run in opposite directions, so that each SPE can directly talk to its neighbors. Communication with other SPE is possible by taking several hops over the communication bus. Each time an SPE wants to transmit a block of data over EIB, it will send an appropriate request to the central on-chip arbiter. The arbiter will schedule all outstanding requests over the four rings. Up to three transfers can be concurrently scheduled over each ring, provided that these transfers use different segments on the ring.

The resulting data bandwidth on the CELL processor is impressive. In a 3.2 GHz CELL chip, the interface from the SPE to the rest of the chip supports 25.6 GB/s data bandwidth in each direction. The downside is that the CELL must be programmed in a very particular manner, as a set of concurrent programs that pass messages to one another. Optimizing the performance of a parallel CELL program is complex, and requires attention to a large collection of details, such as the granularity of tasks and message blocks, the synchronization mechanisms between processors, and the locality of data.

10.5 Summary

In this chapter, we discussed the concepts of on-chip interconnection buses, the lowest abstraction level where software and hardware meet. On-chip buses are shared communication resources to connect bus master components with bus slave components.

We discussed several examples, including the AMBA, CoreConnect, Avalon, and Wishbone bus. These busses support basic read/write transfers between bus masters and slaves. They use enhancements to improve the performance of on-chip communication, including pipelining, split-transfers, and burstmode transfers. On multi-master buses, each transfer needs to be arbitrated. These transfers may need to be prioritized based on the bus master.

Finally, more recent developments in on-chip communication with System-on-Chip emphasize distributed solutions in the form of switched busses or network-on-chip.

The net effect of this evolution is that the design of on-chip communication has become a design challenge on its own, with many different abstraction layers to tackle. For the hardware-software codesigner, it's important to be aware of new evolutions in this field. The communication latency between hardware and software remains a fundamental issue that determines if a solution is communication-constrained and computation-constrained.

10.6 Further Reading

The best reference to study on-chip bus systems is obviously the documentation from the vendors themselves. The AMBA bus specification can be obtained online from ARM (2009a). Likewise, the CoreConnect bus specification can be obtained online from IBM (2009), and the Avalon specification is available from Altera (2011). An in-depth discussion of contemporary on-chip bus systems, including AMBA and CoreConnect, is available from Pasricha and Dutt (2008). The same book also reviews ongoing research topics for on-chip bus systems.

Recently research efforts have focused on network-on-chip. An overview of the design principles may be found in De Micheli's book (Micheli and Benini 2006). A recent special issue of IEEE Design and Test Magazine has reviewed several proposals and open research issues (Ivanov and De Micheli 2005).

10.7 Problems

Problem 10.1. The timing diagram in Fig. 10.17 illustrates a write operation on the AMBA peripheral bus, AMBA APB. A *memory-mapped* register is a register which is able to intercept bus transfers from a specific address. In this case, we wish to create logic which will write PWDATA into a register when a write to address 0x2000 occurs. Develop a logic expression for the logic module shown in Fig. 10.17. Assume a 16-bit address.

Listing 10.1 Program for Problem 10.2

```
#include <stdio.h>
void main() {
  int i, a[0x40];
```

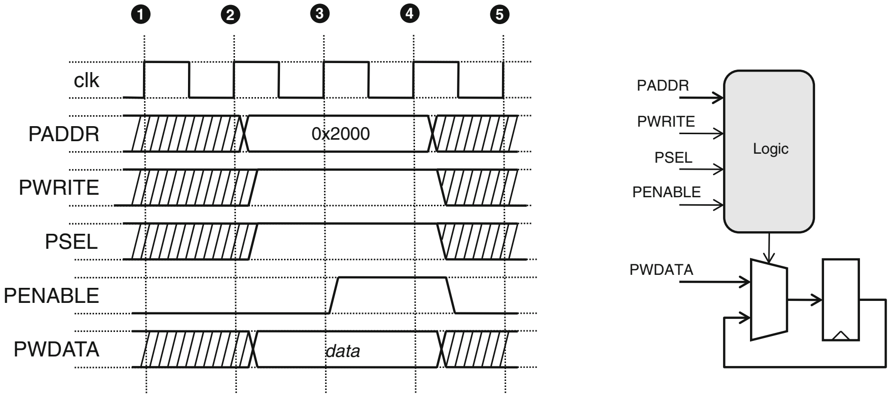


Fig. 10.17 Timing diagram and schematic for Problem 10.1

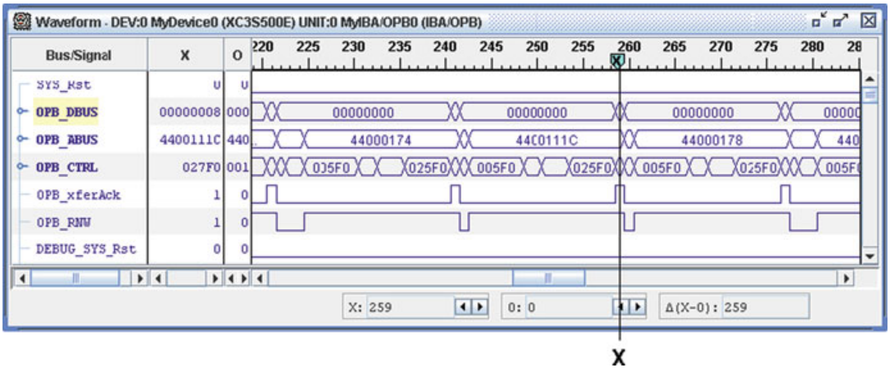

Waveform - DEV:0 MyDevice0 (XC3S500E) UNIT:0 MyIBA/OPB0 (IBA/OPB)
Bus/Signal
X
O
220 225 230 235 240 245 250 255 260 265 270 275 280 28
SYS_Rst U U
OPB_DBUS 00000008 000 00000000 00000000 00000000 0000
OPB_ABUS 4400111C 440 44000174 4400111C 44000178 440
OPB_CTRL 027F0 001 035F0 025F0 005F0 025F0 005F0 025F0 005F
OPB_xferAck 1 0
OPB_RNW 1 0
DEBUG_SYS_Rst 0 0
X: 259
O: 0
Δ(X-O): 259
X

Fig. 10.18 Timing diagram for Problem 10.2

```
  for  (i=0;  i<  0x40;  i++)
     if  (i  >  0x23)
           a[i]  =  a[i-1]  +  1;
     else
       a[i]  =  0x5;
}
```

Problem 10.2. While debugging a C program on a 32-bit microprocessor (shown in Listing 10.1), you capture the bus transfer shown in Fig. 10.18. The microprocessor is attached to an off-chip memory that holds the program and the data. The text and data segment both are stored in an off-chip memory starting at address `0x44000000`. The array `a[]` starts at address `0x44001084`. The instructions from the body of the loop start at address `0x44000170`. Observe closely the timing diagram in Fig. 10.18 and answer the questions below.

Table 10.7 List of bus transactions for Problem 10.3

Transaction	m_addr	m_data	m_rnw
0	0x3F68	1	0
1	0x3F78	1	0
2	0x3F68	2	0
3	0x3F74	2	0
4	0x3F68	3	0
5	0x3F70	3	0
6	0x3F68	4	0
7	0x3F68	5	0
8	0x3F68	6	0

(a) The cursor X in Fig. 10.18 is positioned at a point for which the address bus contains 0x4400111C and the data bus contains 0x8. Is this a memory read of a memory write?

(b) For the same cursor position 'X', is this memory access for an instruction-fetch or for a data-memory read?

(c) For the same cursor position 'X', what is the value of the loop counter i from the C program?

Listing 10.2 Program for Problem 10.3

```
int main() {
  int i, a[32];
  for (i=1; i<32; i++)
     if (i<4)
        a[4-i] = i;
  return 0;
}
```

Problem 10.3. You are debugging an embedded system by recording bus transactions between a microprocessor and an on-chip memory. The program you're analyzing is shown in Listing 10.2. Table 10.7 lists a sequence of bus transactions you observe when `m_rnw` is low. The signals in the Table follow the naming convention of the generic bus (defined in Table 10.2). Analyze the transactions and answer the questions below.

(a) What is the memory address of variable `i`?

(b) What is the memory address of variable `a[0]`?

Problem 10.4. The timing diagram in Fig. 10.19 shows the arbitration process of two masters, M1 and M2, requesting access to a shared bus. Answer the questions below using the information provided in the timing diagram.

(a) Based on the timing diagram, which master has the highest priority for bus transfers: M1, M2 or impossible to tell?

(b) Which master has control over the address bus between clock edge 3 and clock edge 4: M1, M2, or impossible to tell?

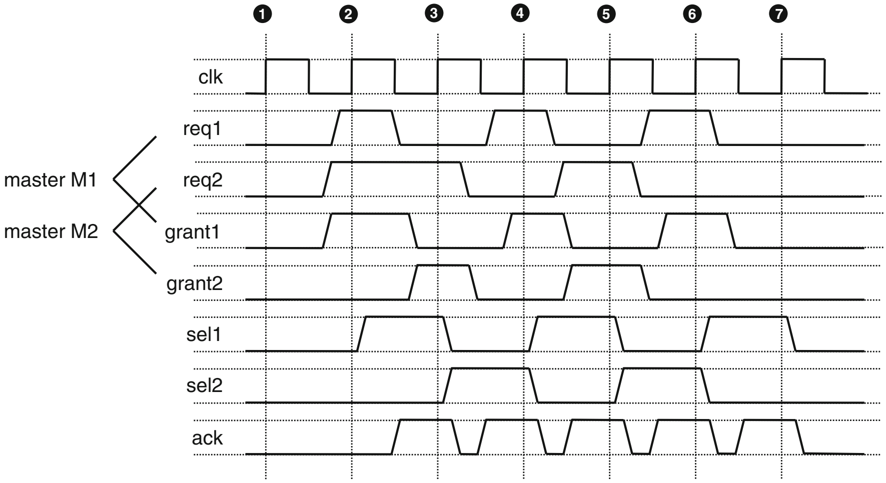
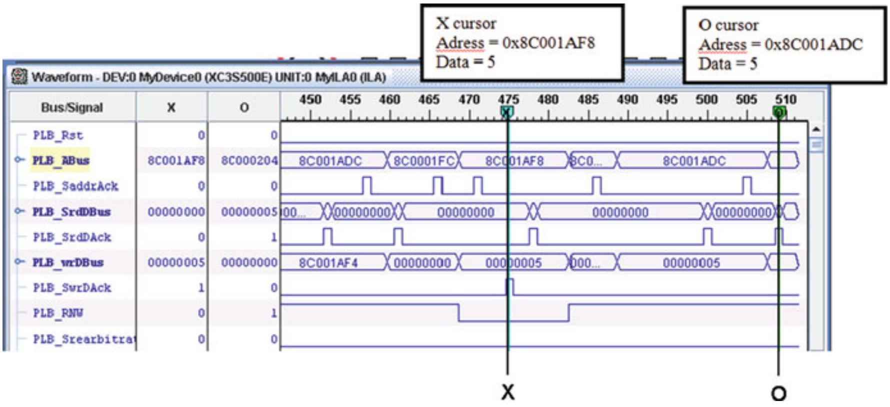


Fig. 10.19 Timing diagram for Problem 10.4



X cursor
Adress = 0x8C001AF8
Data = 5

O cursor
Adress = 0x8C001ADC
Data = 5

Fig. 10.20 Timing diagram for Problem 10.5

(c) What type of component determines the value of the `grant`*x* signals: a bus master, a bus arbiter, or a bus slave?

(d) What type of component determines the value of the `ack` signal: a bus master, an bus arbiter, or a bus slave?

Problem 10.5. The C program in Listing 10.3 is running on a processor attached to a PLB bus (Fig. 10.20). The program and data are stored in an off-chip memory starting at `0x8C00000`. During program execution, a PLB trace is captured in Chipscope, and the result is shown in Fig. 10.20. Observe closely the 'X' and 'O' cursor. Both cursors are positioned on top of the data-acknowledge phase of a

PLB bus transfer. The cursors indicate data memory accesses to variables in the C program.

(a) What memory operation is performed under the X cursor: read or write?
(b) What variable from the C program is accessed under the O cursor ?
(c) What is the base address of the array `a`?

Listing 10.3 Program for Problem 10.5

```
int main () {
  int i, a[32];
  for (i = 1; i < 32; i++)
     if (i < 8)
      a[i+1] = i;
  return 0;
}
```

Chapter 11
Microprocessor Interfaces

11.1 Memory-Mapped Interfaces

A memory-mapped interface allocates part of the address space of a processor for communication between hardware and software. The memory-mapped interface is the most general, most wide-spread type of hardware/software interface. This is no surprise: memory is a central concept in software, and it's supported at the level of the programming language through the use of pointers. In this section, we look into the operation of memory-mapped registers, and into extended concepts such as mailboxes, queues, and shared memory. We also show how memory-mapped interfaces are modeled in GEZEL.

11.1.1 The Memory-Mapped Register

A memory-mapped interface can be as simple as a register which can be read and written through bus transfers on an on-chip bus. Figure 11.1 illustrates the generic setup of a memory-mapped register.

The register will be accessed when a bus transfer is made to a specific memory address, or within a specific memory address range. The memory address, and the related bus command, is analyzed by an address decoder. This decoder will generate a read pulse or a write pulse for the register. A full decoder will generate these pulses for a single address value. However, the complexity of the decoder is proportional to the number of bits that must be decoded. Therefore, it may be cheaper to build a decoder for a range of memory addresses (See Problem Section). The result of such a multi-address decoder is that a register is *aliased* at multiple memory locations. That is, the register will be read or written upon a transfer from or to any address within the range of addresses captured by the address decoder.

A memory-mapped register works as a shared resource between software and hardware. A write-conflict may occur if the hardware and the software attempt to

P.R. Schaumont, *A Practical Introduction to Hardware/Software Codesign*,
DOI 10.1007/978-1-4614-3737-6_11, © Springer Science+Business Media New York 2013

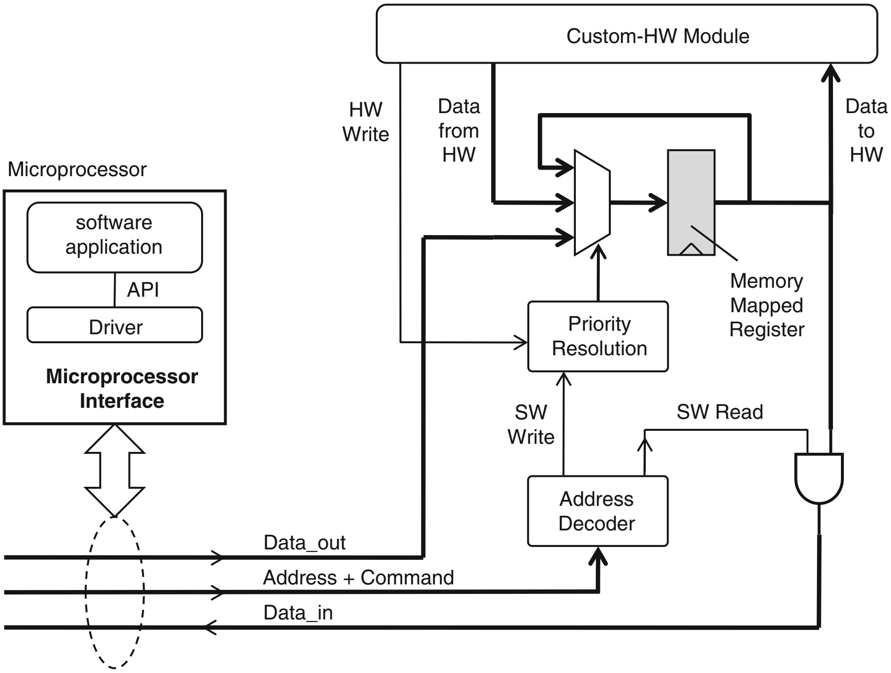


Fig. 11.1 A memory-mapped register

write into the register during the same clock cycle. To resolve this case, a priority decoder will allow hardware or software to proceed. Note that it seldom makes sense to sequentialize the write operations into the register, since one value would overwrite the other.

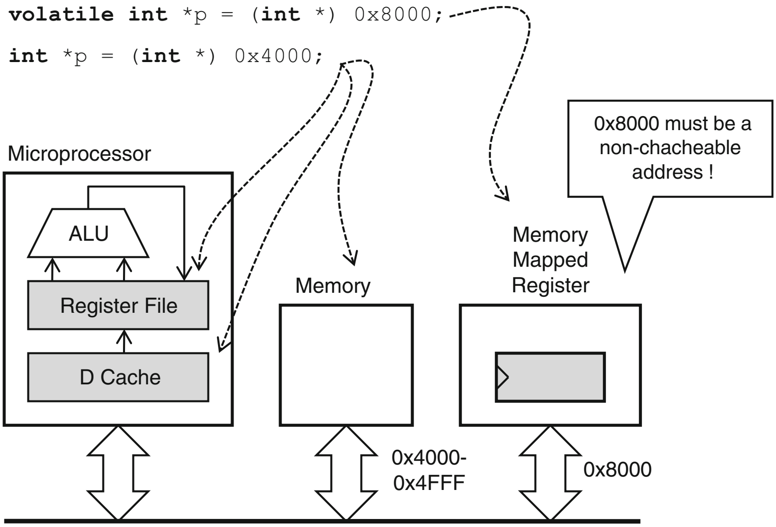
In software, a memory-mapped register can be accessed using an initialized pointer, as shown below. The hardware abstraction layer of a particular microprocessor may provide wrapper functions to write into and read from memory-mapped registers.

```
volatile unsigned int *MMRegister = (unsigned int *) 0x8000;

// write the value '0xFF' into the register
*MMRegister = 0xFF;

// read the register
int value = *MMRegister;
```

Figure 11.2 explains why the pointer must be a `volatile` pointer. A memory-mapped register is integrated into the memory hierarchy of a processor, at the level of main-memory. When a processor instruction will read from or write into that register, it will do so through a memory-load or memory-store operation. Through the use of the `volatile` qualifier, the C compiler will treat the memory hierarchy slightly different.



```
volatile int *p = (int *) 0x8000;
int *p = (int *) 0x4000;
```

Fig. 11.2 Integrating a memory-mapped register in a memory hierarchy

- When using normal pointer operations, the processor and the compiler will attempt to minimize the number of operations to the main memory. This means that the value stored at an `int *` can appear in three different locations in the memory hierarchy: in main memory, in the cache memory, and in a processor register.
- By defining a register as a `volatile int *`, the compiler will avoid maintaining a copy of the memory-mapped register in the processor registers. This is essential for a memory-mapped register, because it can be updated by a custom-hardware module, outside of the control of a microprocessor.

However, defining a memory-mapped register with a `volatile` pointer will not prevent that memory address from being cached. There are two approaches to deal with this situation. First, the memory addresses that include a memory-mapped register could be allocated into a non-cacheable memory area of a processor. This requires a processor with a configurable cache. For example, the Microblaze processor uses this technique. A second approach is to use specific cache-bypass instructions on the microprocessor. Such instructions are memory-access operations on the same address as normal load/store instructions, but they do not relay on the processor data cache. The Nios-II processor, for example, has `ldio` and `stio` instructions that serve as cache-bypass counterparts to the normal load (`ld`) and store (`st`) instructions.

Building on the principle of a memory-mapped register, we can create communication structures to tie hardware and software together.

11.1.2 Mailboxes

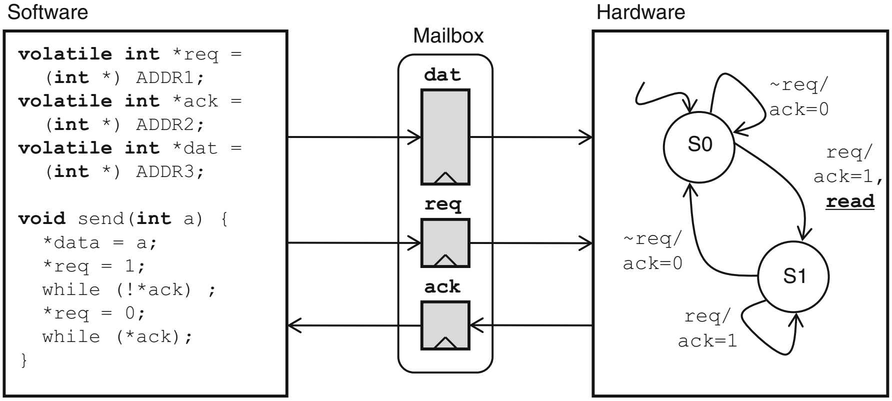
A mailbox is a simple extension of a memory-mapped register with a handshake mechanism. The obvious problem with a memory-mapped register is that the hardware cannot tell when the software has written or read the register, and vice versa. Thus, we need the equivalent of a mailbox: a box with a status flag to signal its state. Suppose that we are sending data from software to hardware, then the software writes into the register, and next sets a ‘mailbox full’ flag. The hardware will read the value from the register after it sees the mailbox flag being set. After reading, the hardware will clear the ‘mailbox full’ flag so that the software can proceed with the next transfer.

This construct is easy to build using three memory mapped registers, as illustrated in Fig. 11.3. In this case, the sender is the software program on the left of the figure, and the receiver is the hardware module on the right of the figure. After writing fresh data into the `data` memory-mapped register, the `req` flag is raised. The hardware component is a finite state machine that scans the state of the `req` flag and, as soon as the flag goes high, will capture the data, and raise the `ack` flag in response. Meanwhile, the software program is waiting for the `ack` flag to go high. Once both `ack` and `req` are high, a similar sequence is followed to reset them again.

The entire protocol thus goes through four phases: `req` up, `ack` up, `req` down, `ack` down. Because both parties work through the protocol in an interlocked fashion, the communication automatically adapts to the speed of the slowest component. The protocol has two synchronization points: once just after both `ack` and `req` have transitioned high, and a second time just after both `ack` and `req` are low. This means that it is quite easy to double the throughput of the protocol in Fig. 11.3.

A mailbox based on memory-mapped registers has a high overhead, in terms of design cost as well as in terms of performance. The frequent synchronization of hardware and software through handshakes has two disadvantages. First it requires a fine-grained interaction in the execution flow of hardware and software. Keep in mind that each four-phase handshake implies two synchronization points. The second disadvantage is that frequent synchronization implies additional bus transfers. For example, consider the `while` statements in the C program in Fig. 11.3. Each iteration in the while loop generates one read from a `volatile` pointer, resulting in one bus transfer.

Both of these problems – tight coupling and extra bus transfers – can be solved by improving the buffer mechanism between hardware and software. We will discuss two examples. The first is to use *FIFO queues* instead of mailboxes to uncouple hardware and software and to remove the need for interlocked read/write of the memory-mapped interface. The second is to use *shared memory*. This can reduce the need for synchronization by increasing the granularity of the data transfer, from a single word to an entire memory block.



```
volatile int *req =
   (int *) ADDR1;
volatile int *ack =
   (int *) ADDR2;
volatile int *dat =
   (int *) ADDR3;

void send(int a) {
  *data = a;
  *req = 1;
  while (!*ack) ;
  *req = 0;
  while (*ack);
}
```

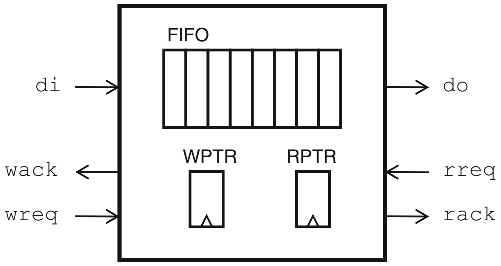
Fig. 11.3 A mailbox register between hardware and software

11.1.3 First-In First-Out Queues

When a handshake protocol is used to implement a mailbox, the write and read operations are interleaved. This is inconvenient when the nature of write-operations and read-operations is very different. For example, writing into the FIFO could be bursty, so that several tokens are written in rapid succession, while reading from the FIFO could be very regular, so that tokens are read at a steady pace. The role of the FIFO is to store the extra tokens during write operations, and to gradually release them during read operations. Of course, in the long term, the average rate of writing into the FIFO must be equal to the average reading rate.

FIFO-based communications can be implemented using handshake signals too. Instead of having a single pair of `request/acknowledge` signals, we will now have *two* pairs. One pair controls the write operations into the FIFO, while the second pair controls the read operations into the FIFO. Figure 11.4 illustrates a FIFO queue with individual handshakes for read and write operations into the queue. In this case, we have assumed a FIFO with eight positions. The GEZEL code on the left of Fig. 11.4 shows the register-transfer level operations. In this case, the handshake operations are tied to incrementing a read-pointer and a write-pointer on a dual-port memory. The increment operations are conditional on the state of the FIFO, and the level of the `request` inputs. The state of the FIFO can be empty, full or non-empty, and this condition is evaluated based on comparing the read and write pointer values. Encoding the status of the FIFO using only the value of the read and write pointer values has a negative side-effect: the code shown in Fig. 11.4 requires the FIFO buffer to maintain at least one empty space at all times. By introducing a separate `full` flag, all spaces in the buffer can be utilized (See Problem Section).

Fig. 11.4 A FIFO with handshakes on the read and write ports



```
dp fifo(in  di   : ns(8);
        in  wreq : ns(1);
        out wack : ns(1);
        out do   : ns(8);
        in  rreq : ns(1);
        out rack : ns(1)) {
  sig read, write : ns(1);
  reg rptr, wptr : ns(3);
  use dualport_mem(di, wptr, write,  // write port
                   do, rptr, read);  // read port
  always {
    read  = (rreq & (wptr != rptr)) ? 1 : 0;
    write = (wreq & ((wptr + 1) != rptr) ? 1 : 0;
    wptr  = write ? wptr + 1 : wptr;
    rptr  = read ? rptr + 1 : rptr;
    wack  = write;
    rack  = read;
  }
}
```

11.1.4 Slave and Master Handshakes

The FIFO shown in Fig. 11.4 has two *slave* interfaces: one for writing and one for reading. A slave interface waits for a control signal and reacts to it. Thus, the acknowledge signals will be set in response to the request signals. There is a matching master protocol required for a slave protocol. In the hardware/software interface of Fig. 11.3, the software interface uses a master-protocol and the hardware interface uses a slave-protocol.

By building a FIFO section with a slave input and a master output, multiple sections of FIFO can be connected together to build a larger FIFO. An example of this scheme is shown in Fig. 11.5. In this implementation, we use a FIFO with a single storage location, implemented as a register. The updates of this register are under control of a finite state machine, which uses the request/acknowledge handshake signals as inputs. Note the direction of the request/acknowledge arrows on the input port and the output port of the FIFO. At the input port, the request signal is an input, while at the output port, the request signal is an output.

A master handshake interface can only be connected to a slave handshake interface, otherwise the handshake protocol does not work. The FIFO in Fig. 11.5

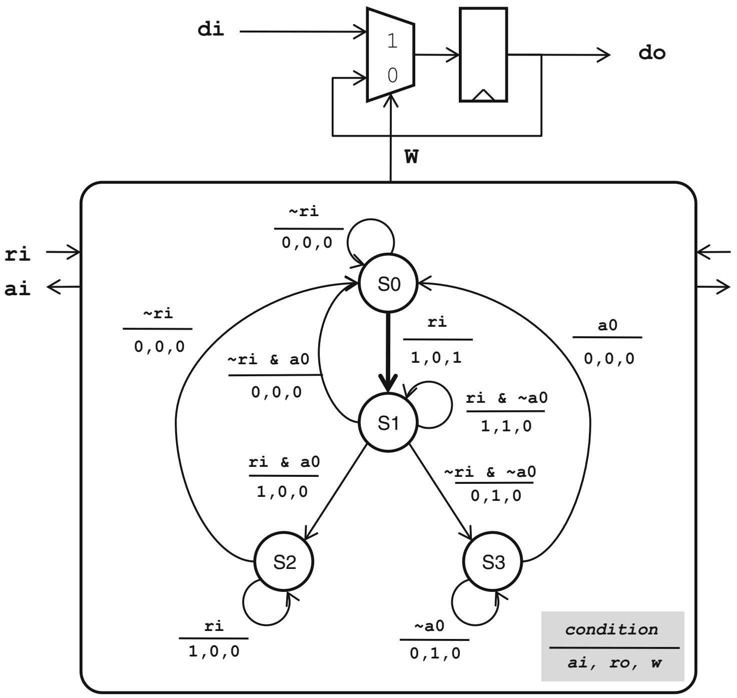


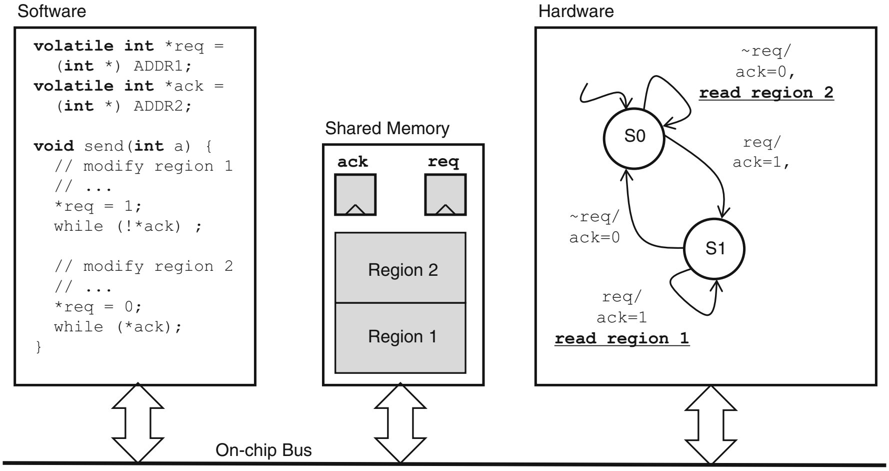
Fig. 11.5 A one-place FIFO with a slave input handshake and a master output handshake

operates as follows. Initially, the FSM is in state `S0`, waiting for the input request signal to be set. Once it is set, it will write the value on the input port into the register and transition to state `S1`. In state `S1`, the FSM sets the request signal for the output port, indicating that the FIFO stage is non-empty. From state `S1`, three things can happen, depending upon which handshake (input or output) completes first. If the input handshake completes (`ri` falls low), the FSM goes to state `S3`. If the output handshake responds (`a0` raises high), the FSM goes to state `S2`. If both handshakes complete at the same time, the FSM directly goes back to `S0`.

11.1.5 Shared Memory

Instead of controlling the access on a single register, a single handshake can also be used to control access to a region of memory. In that case, a shared-memory scheme is obtained (Fig. 11.6).

In this example, a memory module is combined with two memory-mapped registers. The registers are used to implement a two-way handshake. The memory is split up in two different regions. The handshake protocol is used to control access to these



```
volatile int *req =
  (int *) ADDR1;
volatile int *ack =
  (int *) ADDR2;

void send(int a) {
  // modify region 1
  // ...
  *req = 1;
  while (!*ack) ;

  // modify region 2
  // ...
  *req = 0;
  while (*ack);
}
```

Fig. 11.6 A double-buffered shared memory with a memory-mapped request/acknowledge handshake

regions. In one phase of the protocol, changes are allowed to region 1 of the memory. In the second phase of the protocol, changes are allowed in region 2 of the memory. This way, the protocol ensures that all data values in a given region of memory are consistent. This is useful to exchange large data records such as images, internet packets, file headers, and so on. Distributed memory organizations in System-on-Chip (Chap. 8) often make use of the shared-memory concept to implement communication channels. In addition, multiprocessor systems often make use of shared-memory structures to support inter-processor communications. A second observation is that, in some applications, clever organization of the read/write access patterns into a shared memory may lead to substantial performance improvements. This is especially true for applications that require data-reorganization in between processing stages.

11.1.6 GEZEL Modeling of Memory-Mapped Interfaces

To conclude our discussion on memory-mapped interfaces. We describe the modeling of memory-mapped interfaces in GEZEL. A memory-mapped interface is represented using a dedicated simulation primitive called an `ipblock`. There is a separate primitive for a read-interface and a write-interface, and each is mapped to a user-specified memory address.

The following example shows the modeling of a memory-mapped interface for a coprocessor that evaluates the Greatest Common Divisor Algorithm. The coprocessor uses a single input and a single output, and is controlled using memory-mapped registers. Listing 11.1 illustrates the design of the coprocessor. Lines 1–28 contain five `ipblock`, modeling the software processor and the memory-mapped hardware/software interfaces. These interfaces are not modeled as actual registers, but as modules capable of decoding read operations and write operations to a given memory bus. The decoded addresses are given as a parameter to the `ipblock`. For example, the request-signal of the handshake is mapped to address `0x80000000`, as shown on lines 6–10. The coprocessor kernel is shown on lines 30–43, and is a standard implementation of the greatest-common-divisor algorithm similar to the one used in earlier examples in this book. The hardware-software interface logic is embedded in the interface module, included on lines 45–101, which links the `ipblock` with this datapath. This module is easiest to understand by inspecting the FSM description. For each GCD computation, the hardware will go through two complete two-way handshakes. The first handshake (lines 84–89) provides the two operands to the GCD hardware. These operands are provided sequentially, over a single input port. After the first handshake, the computation starts (line 92). The second handshake (lines 93–100) is used to retrieve the result. This approach of tightly coupling the execution of the algorithm with the hardware/software interface logic has advantages and disadvantages: it results in a compact design, but it also reduces the flexibility of the interface.

Listing 11.1 A memory-mapped coprocessor for GCD

```
 1  ipblock my_arm {
 2    iptype "armsystem";
 3    ipparm "exec=gcddrive";
 4  }
 5
 6  ipblock m_req(out data : ns(32)) {
 7    iptype "armsystemsource";
 8    ipparm "core=my_arm";
 9    ipparm "address=0x80000000";
10  }
11
12  ipblock m_ack(in data : ns(32)) {
13    iptype "armsystemsink";
14    ipparm "core=my_arm";
15    ipparm "address=0x80000004";
16  }
17
18  ipblock m_data_out(out data : ns(32)) {
19    iptype "armsystemsource";
20    ipparm "core=my_arm";
21    ipparm "address=0x80000008";
22  }
23
24  ipblock m_data_in(in data : ns(32)) {
25    iptype "armsystemsink";
26    ipparm "core=my_arm";
27    ipparm "address=0x8000000C";
28  }
```

```

dp euclid(in  m_in, n_in : ns(32);
          in  go         : ns( 1);
          out ready      : ns( 1);
          out gcd        : ns(32)) {
  reg m, n : ns(32);
  sig done : ns(1);

  always { done  = ((m==0) | (n==0));
           ready = done;
           gcd   = (m > n) ? m : n;
           m     = go ? m_in : ((m >  n) ? m - n : m);
           n     = go ? n_in : ((n >= m) ? n - m : n);
         }
}

dp tb_euclid {
  sig m, n  : ns(32);
  sig ready : ns(1);
  sig go    : ns(1);
  sig gcd   : ns(32);
  use euclid(m, n, go, ready, gcd);
  use my_arm;
  sig req, ack, data_out, data_in : ns(32);

  use m_req(req);

  use m_ack(ack);
  use m_data_out(data_out);
  use m_data_in(data_in);

  reg r_req    : ns(1);
  reg r_done   : ns(1);
  reg r_m, r_n : ns(32);

  always   { r_req   = req;
             r_done  = ready;
             data_in = gcd;
             m       = r_m;
             n       = r_n;
           }
  sfg ack1 { ack = 1;        }
  sfg ack0 { ack = 0;        }
  sfg getm { r_m = data_out; }
  sfg getn { r_n = data_out; }
  sfg start{ go  = 1;        }
  sfg wait { go  = 0;        }
}

fsm ctl_tb_euclid(tb_euclid) {
  initial s0;
  state s1, s2, s3, s4, s5, s6;

  @s0 (ack0, wait) -> s1;

  // read m
  @s1 if (r_req)  then (getm, ack1, wait) -> s2;
                  else (ack0, wait)       -> s1;
  // read n
  @s2 if (~r_req) then (getn, ack0, wait) -> s3;
                  else (ack1, wait)       -> s2;

  // compute
  @s3 (start, ack0) -> s4;
  @s4 if (r_done) then (ack0, wait)       -> s5;
                  else (ack0, wait)       -> s4;

  // output result
  @s5 if (r_req)  then (ack1, wait)       -> s6;
                  else (ack0, wait)       -> s5;
  @s6 if (~r_req) then (ack0, wait)       -> s1;
                  else (ack1, wait)       -> s6;
}

system S {
  tb_euclid;
}
```

Listing 11.2 shows a C driver program that matches the coprocessor design of Listing 11.1. The program evaluates the GCD operation of the numbers 80 and 12, followed by the GCD of the numbers 80 and 13. Note the difference between a master handshake protocol, as shown in the functions `sync1()` and `sync0()`, and a slave handshake protocol, as illustrated in the FSM transitions in Listing 11.1. In a master handshake, the request signals are written first, and followed by reading and testing of the acknowledge signals. In a slave handshake, the request signals are tested first, followed by a response on the acknowledge signals.

Executing this cosimulation is easy. We first cross-compile the C program to an executable. Next, we run the cosimulator on the executable and the GEZEL program.

```
> arm-linux-gcc -static  gcddrive.c -o gcddrive
> gplatform  gcd.fdl
core my_arm
armsystem: loading executable [gcddrive]
armsystemsink: set address 2147483652
armsystemsink: set address 2147483660
gcd(80,12) = 4
gcd(80,13) = 1
Total Cycles: 11814
```

In conclusion, memory-mapped interfaces are a general-purpose mechanism to create hardware/software interfaces. The principle of a two-way handshake is applicable to many different situations. In this case, hardware and software are synchronized using simple shared storage locations. Because memory-mapped interfaces rely on a general-purpose on-chip bus, they easily become a bottleneck

Listing 11.2 A C driver for the GCD memory-mapped coprocessor

```
#include <stdio.h>
volatile unsigned int *req = (unsigned int *) 0x80000000;
volatile unsigned int *ack = (unsigned int *) 0x80000004;

void sync1() {
  *req = 1;  while (*ack == 0) ;
}

void sync0() {
  *req = 0;  while (*ack == 1) ;
}

int main() {
  volatile unsigned int *di  = (unsigned int *) 0x80000008;
  volatile unsigned int *ot  = (unsigned int *) 0x8000000C;

  *di = 80;
  sync1();
  *di = 12;
  sync0();
  sync1();
  printf("gcd(80,12)␣=␣%d\n", *ot);
  sync0();

  *di = 80;
  sync1();
  *di = 13;
  sync0();
  sync1();
  printf("gcd(80,13)␣=␣%d\n", *ot);
  sync0();

  return 0;
}
```

when throughput requirements increase. In addition, because the on-chip bus is shared with other components, a memory-mapped interface will also show a variable latency. For cases that require a dedicated, tightly-controlled link, we will need a more efficient hardware/software interface, such as the one discussed next.

11.2 Coprocessor Interfaces

In cases where high data-throughput between the software and the custom hardware is needed, the memory-mapped interface may become inadequate. Instead, we can use a coprocessor interface, a dedicated processor interface specifically created to attach custom hardware modules. As demonstrated in Fig. 11.7, a coprocessor

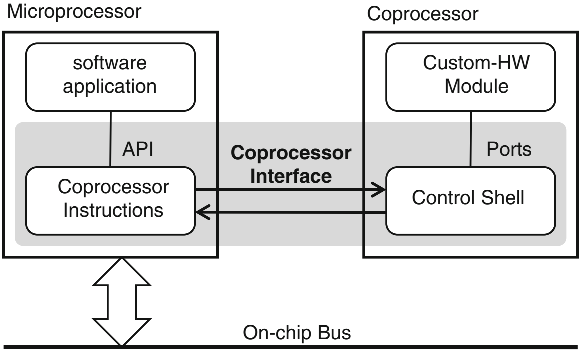


Fig. 11.7 Coprocessor interface

interface does not make use of the on-chip bus. Instead, it uses a dedicated port on the processor. This port is driven using a set of specialized instructions, the coprocessor instructions. The coprocessor instruction set, as well as the specific coprocessor interface, depends on the type of processor. A classic coprocessor example is a floating-point calculation unit, which is attached to an integer core. Not all processors have a coprocessor interface.

The choice of specializing a custom hardware module towards a specific coprocessor interface is an important design decision. Indeed, it locks the custom hardware module into a particular processor, and therefore it limits the reusability of that custom hardware module to systems that also use the same processor. In comparison, a custom hardware module with a memory-mapped interface is specific to a bus system, and not to a particular processor type. For example, a hardware module for a CoreConnect bus can be used in any system that uses a CoreConnect bus. Hence, the use case of the coprocessor model is more specific than the use case of the memory-mapped module model.

The main advantages of a coprocessor interface over an on-chip bus are higher throughput and fixed latency. We consider each of these aspects.

- Coprocessor interfaces have a higher throughput than memory-mapped interfaces because they are not constrained by the wordlength of the bus. For example, coprocessor ports on 32-bit CPUs may support 64- or 128-bit interfaces, allowing them to transport four words per coprocessor instruction. Hardware/software interfaces based on coprocessor instructions may also be implemented more efficiently than load/store instructions. A coprocessor instruction typically uses two source operands and one destination operand. In contrast, a load/store instruction will specify only a single operand. A complete hardware/software interaction over a coprocessor interface may thus be specified with fewer instructions as the same interaction over an on-chip bus.
- A coprocessor interface can also maintain fixed latency, so that the execution timing of software and hardware is precisely known. Indeed, a coprocessor bus is a dedicated connection between hardware and software, and it has a stable, predictable timing. This, in turn, simplifies the implementation of hardware/soft-

ware synchronization mechanisms. In contrast, an on-chip bus interface uses a communication medium which is shared between several components, and which may include unknown factors such as bus bridges. This leads to unpredictable timing for the hardware/software interaction.

We will illustrate the coprocessor interface model by means of two examples: the Fast Simplex Link used in the Microblaze processor, and the Floating Point Coprocessor Interface used by the LEON-3 processor.

11.2.1 The Fast Simplex Link

The Microblaze processor, a soft-core processor that can be configured in a Xilinx FPGA, is configurable with up to eight Fast Simplex Link (FSL) interfaces. An FSL link consists of an output port with a master-type handshake, and an input port with a slave-type handshake. The *simplex* part of FSL refers to the direction of data, which is either output or input. The Microblaze processor has separate instructions to write to the output port and read from the input port.

Figure 11.8 shows an FSL interface. In between the hardware coprocessor and the Microblaze, FIFO memories can be added to loosen the coupling between hardware and software. Data going from the Microblaze to the FIFO goes over a master interface consisting of the signals `data`, `write`, and `full`. Data going from the FIFO to the Microblaze goes over a slave interface which includes the signals `data`, `exists`, and `read`. Each of the master interface and slave interface implements a two-way handshake protocol. However, the labeling of handshake signals is slightly different compared to what we discussed before: `write` and `exists` correspond to `req`, while `full` and `read` correspond to `ack`.

Figure 11.9 shows the operation of the FSL interface for a FIFO with two positions. The timing diagram demonstrates the activities of writing three tokens into the FIFO, and reading them out again. The operation will be familiar because of the two-way handshake implemented in the FSL protocol. On clock edge 2, the first data item is written into the FIFO. The `exists` flag goes up because the FIFO is non-empty. On clock edge 3, a second data item is written into the FIFO. We assume that the FIFO holds no more than two places, so that this second write will fill the FIFO completely, and the `full` flag is raised as a result. In order to write more tokens into the FIFO, at least one read operation needs to complete first. This happens on clock edge 5. The third data item is written into the FIFO on clock edge 6, which completely fills the FIFO again. The FIFO is emptied by read operations on clock edges 7 and 8. From clock edge 9 on, the FIFO is empty.

The read and write operations on an FSL interface are controlled using dedicated instructions on the Microblaze processor. The basic read and write operations are of the form:

```
put  rD, FLSx   // copy register rD to FSL interface FSLx
get  rD, FSLx   // copy FSL interface FSLx to register rD
```

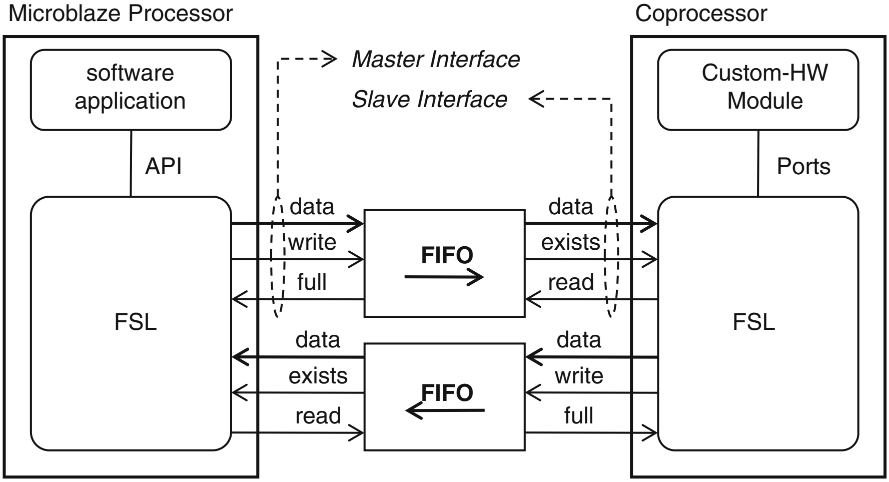


Fig. 11.8 The fast simplex link interface

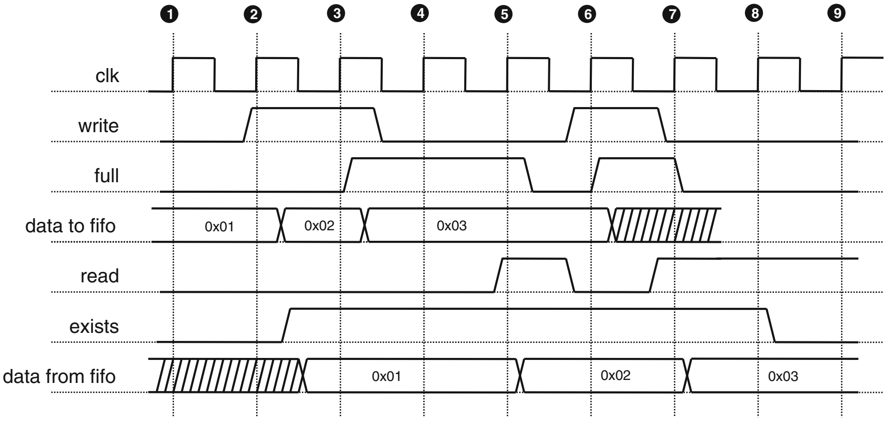


Fig. 11.9 The FSL handshake protocol

There are many variants of these instructions, and we only discuss the main features.

- The instruction can be configured as a blocking as well as a non-blocking operation. When a non-blocking instruction is unable to complete a read operation or a write operation, it will reset the carry flag in the processor. This way, a conditional jump instruction can be used to distinguish a successful transfer from a failed one.
- The FSL I/O instructions can also read a control status flag directly from the hardware interface: The *data* bus shown in Fig. 11.8 includes a 32-bit data word and a single-bit control flag. An exception can be raised if the control bit if

different from the expected value. This allows the hardware to influence the control flow in the software.
- The FSL I/O instructions can be configured as *atomic* operations. In that case, a group of consecutive FSL instructions will run as a single set, without any interrupts. This is useful when the interface to a hardware module is created using several parallel FSL interfaces. By disallowing interrupts, the designer can be sure that all FSL interfaces are jointly updated.

The FSL interface is a popular coprocessor interface in the context of FPGA designs. It uses a simple hardware protocol, and is supported with a configurable, yet specialized instruction set on the processor. However, it's only a data-moving interface. In the next section, we will discuss a floating-point coprocessor interface. Such an interface has a richer execution semantics.

11.2.2 The LEON-3 Floating Point Coprocessor Interface

The interface in this section is a tightly-coupled interface to attach a floating-point unit (FPU) to a processor. While a high-end desktop processor has the FPU built-in, embedded processors often configure this module as an optional extension. The FPU interface discussed in this section is the one used in the LEON-3 processor, designed by Gaisler Research. It is a tightly-coupled interface: instructions executed on the FPU remain synchronized to the instructions executing on the main processor.

Figure 11.10 illustrates the main signals in the FPU interface. The LEON-3 32-bit microprocessor includes an integer-instruction pipeline, a set of floating-point registers, and an instruction-fetch unit. When the microprocessor fetches a floating-point instruction, it will dispatch that instruction to the floating-point coprocessor. After the result of the floating point operation is returned to the microprocessor, it is merged with the flow of instructions in the integer pipeline. There are several interesting issues with this scheme, and the signals on the coprocessor interface can best be understood by examining the interaction between the FPU and the microprocessor in detail.

The FPU contains two different datapaths. One is a *linear* pipeline with three pipeline stages. The second is a *non-linear* pipeline, and it consists of a pipeline with feedback, so that one pipeline stage remains in use for several cycles. FPU operations such as add, subtract, and multiply are handled by the linear pipeline, while operations such as divide and square-root are handled by the non-linear pipeline. FPU instructions through the linear pipeline have a latency of three clock cycles and a throughput of one instruction per clock cycle. However, FPU instructions through the non-linear pipeline can have a latency of up to 24 clock cycles, and their throughput can be as low as one instruction every 24 clock cycles.

The challenge of the coprocessor interface is to maintain the instruction sequence in the FPU synchronized with the microprocessor. This is non-trivial because the latency and throughput of instructions in the FPU is irregular. Indeed, results must

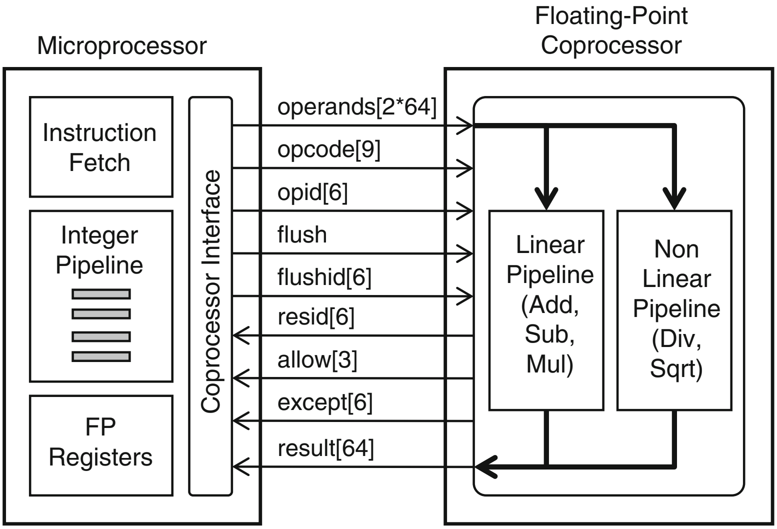
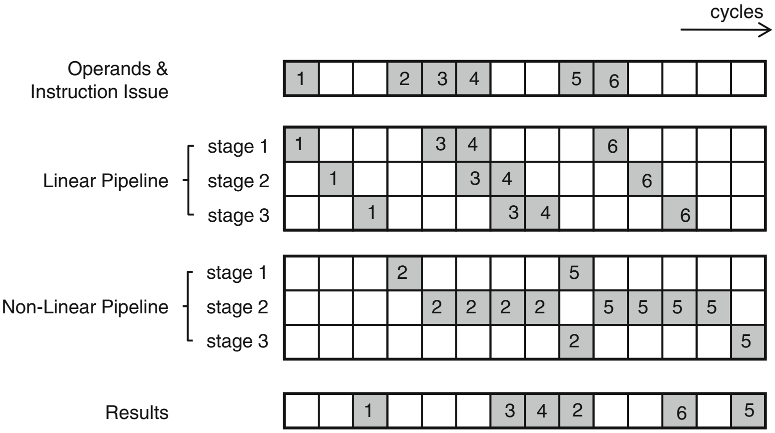


Fig. 11.10 GRFPU floating point coprocessor interface



Operands & Instruction Issue		1			2	3	4			5	6				
Linear Pipeline	stage 1	1				3	4				6				
	stage 2		1				3	4				6			
	stage 3			1				3	4				6		
Non-Linear Pipeline	stage 1				2					5					
	stage 2					2	2	2	2		5	5	5	5	
	stage 3									2					5
Results				1				3	4	2			6		5

Fig. 11.11 GRFPU instruction issue and result return

be merged in the microprocessor in the same order as their operands are dispatched to the FPU. Figure 11.11 demonstrates that, due to the complex pipeline architecture of the FPU however, results may arrive out-of-order. The GRFPU interface handles this problem.

- Each operation for the FPU is provided as a set of `operands`, with a given `opcode`, and an instruction identifier `opid`. When the FPU operation finishes, it returns the `result`, together with a corresponding instruction identifier `resid`. By examining the value of `resid`, the processor can determine what result corresponds to what set of operands. The instruction identifier is generated by the processor, but is typically a simple counter (similar to the labels in the grids on Fig. 11.11). For each result, the coprocessor will also generate an exception code `except`, which allows the detection of overflow, divide-by-zero, and so on.

- When a floating-point instruction appears just after a conditional branch, the floating-point instruction may need to be cancelled when the conditional branch would be taken. Through the control signal `flush`, the microprocessor indicates that the coprocessor should cancel an instruction. The instruction identifier is provided through `flushid`.
- Because of the non-linear pipeline in the coprocessor, not all types of FPU instructions can be accepted every clock cycle. The `allow` signal indicates to the microprocessor what instructions can start at a given clock cycle.

Clearly, this interface definition is specialized towards floating-point coprocessors. In addition, the interface is tightly-coupled with the micro-processor. Instead of handshake signals, a series of control signals is defined to ensure that the execution flow of the coprocessor and the microprocessor will stay synchronized.

11.3 Custom-Instruction Interfaces

The integration of hardware and software can be considerably accelerated with the following idea.

1. Reserve a portion of the opcodes from a microprocessor for new instructions
2. Integrate the custom-hardware modules directly into the micro-architecture of the micro-processor
3. Control the custom-hardware modules using new instructions derived from the reserved opcodes.

Since this solution does not make use of an on-chip bus interface or a coprocessor interface, it is highly processor-specific. The resulting design is called an *Application-Specific Instruction-set Processor* or ASIP for short. The difference between the use of an ASIP and the use of a coprocessor is that, in the former case, the design of the instruction set is defined by the application, while in the latter case, the coprocessor instruction set is defined as part of the micro-processor.

Design with ASIP is experienced by designers as an easier form of hardware-software codesign, because it automates some of the more difficult aspects of hardware/software codesign. First, the instruction-fetch and dispatch mechanism in the micro-processor ensures that custom-hardware and software remain synchronized. Second, design of an ASIP proceeds in an incremental fashion. A solution can be made with one custom instruction, two custom instructions, or more. The incremental nature of the design avoids drastic changes to the system architecture. In contrast, the traditional hardware design process is bottom-up, exact and rigorous.

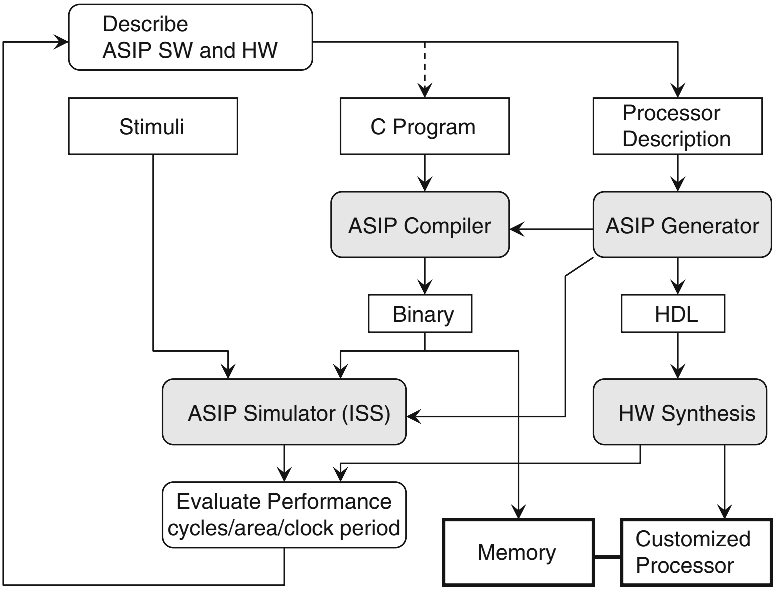


Fig. 11.12 ASIP design flow

11.3.1 ASIP Design Flow

When creating an ASIP, a designer captures the application initially as a C program. After the performance of this program on the processor is evaluated, adjustments to the program and the processor architecture can be made. Such adjustments include, for example, new hardware in the processor datapath, and new processor instructions. After the processor hardware is adjusted, the C program can be tuned as well to make use of these instructions. This leads to a design flow as illustrated in Fig. 11.12.

The design starts with a C program and a description of the processor. The *processor description* is not a hardware description in terms of FSMD, but a specialized description of processor resources. It enumerates the instructions supported by the processor, the configuration and size of register files, and the memory architecture. Using the processor description, an *ASIP generator* will create design components for the ASIP. This includes a software development toolkit (compiler, assembler, linker, debugger) as well as a synthesizable hardware description of the processor. Using the software development toolkit, the application program in C can be compiled, simulated and evaluated. The hardware description can be technology-mapped onto gates or FPGA, which yields the processor implementation as well as technology metrics such as area and achievable processor clock. Based on the performance evaluation, the processor description can be reworked to obtain a better performance on a given application. This may also require rework of the application in C.

Compared to SoC design based on custom-hardware modules, will the ASIP design flow in Fig. 11.12 always deliver better performance? Not in the general case. Keep in mind that the basic architecture template of an ASIP is a sequential processor. The fundamental bottlenecks of the sequential processor (memory-access, sequential execution of code) are also fundamental bottlenecks for an ASIP design.

Compared to SoC design based on custom-hardware modules, can the ASIP design flow deliver less error-prone results? Yes, it can for the following two reasons. First, the design process is incremental. A functional error can be detected very quickly, in the early phases of the design. Second, it works at a higher level of abstraction. The application is modeled in C. The processor description language is also at a higher level of abstraction, compared to hardware description languages.

In the past few years, a tremendous progress has been made on design tools that support the ASIP design flow. All of the shaded boxes in Fig. 11.12 can be obtained as commercial tools.

11.3.2 Example: Endianness Byte-Ordering Processor

We describe an example of ASIP design, as well as GEZEL modeling of the custom-instruction. The application is an endianness byte-ordering processor.

Figure 11.13 illustrates the design problem. Processors have a chosen byte-ordering or endianness. When a processor transmits data words over a network, the byte order of the transmitted packets is converted from the processor byte-order into network byte-order, which is big-endian in practice. For example, when a processor is little endian, it will store the word `0x12345678` with the lowest significant byte in the lowest memory address. However, when that word is transmitted over a network, the packet byte-order must follow a big-endian convention that will send `0x78` first and `0x12` last. The communication protocol stack on the processor will therefore convert each word from little-endian format to big-endian format before handing it off to the network buffer on the Ethernet card.

For a 32-bit processor, endianness conversion involves byte-level manipulation using shifting and masking. The following is an example of a function that converts little-endian to big-endian (or vice versa) in C.

```
for (i=0; i<4096; i++)
   ot[i] = ( ((in[i] & 0x000000ff) << 24) |
             ((in[i] & 0x0000ff00) <<  8) |
             ((in[i] & 0x00ff0000) >>  8) |
             ((in[i] & 0xff000000) >> 24));
```

On a StrongARM processor (which is little-endian), this loop requires 13 cycles per iteration (assuming no cache misses). Examining the assembly, we would find that there are 11 instructions inside of the loop, leaving two cycles of pipeline stalls per iteration – one for the branch and one data-dependency (`ldr` instruction).

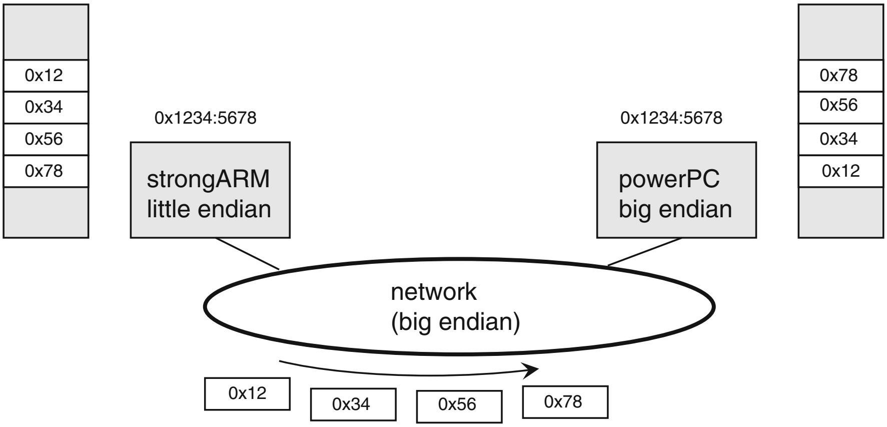


Fig. 11.13 Endianness byte-ordering problem

```
.L6:
    ldr r3, [r4, ip, asl #2]     ; read in[i]
    and r2, r3, #65280           ; evaluate ot[i]
    mov r2, r2, asl #8           ;   ...
    orr r2, r2, r3, asl #24      ;   ...
    and r1, r3, #16711680        ;   ...
    orr r2, r2, r1, lsr #8       ;   ...
    orr r2, r2, r3, lsr #24      ;   ...
    str r2, [r0, ip, asl #2]     ;   and write back
    add ip, ip, #1               ; next iteration
    cmp ip, lr                   ;
    ble .L6
```

Let’s consider possible design improvements. In hardware, the endianness-conversion is obviously very simple: it is a simple wiring pattern. The hardware-software codesign problem, therefore, essentially revolves around the problem of moving data around. Before designing an ASIP, let’s try to solve this problem with a memory-mapped coprocessor.

The memory-mapped coprocessor looks as follows. It uses two memory-mapped registers, one for input and one for output.

```
ipblock myarm {
  iptype "armsystem";
  ipparm "exec=byteswap";
}

ipblock port1(out data    : ns(32)) {
  iptype "armsystemsource";
  ipparm "core=myarm";
  ipparm "address = 0x80000000";
}

ipblock port2(in data    : ns(32)) {
  iptype "armsystemsink";
  ipparm "core=myarm";
  ipparm "address = 0x80000004";
}

dp mmswap(in d1 : ns(32); out q1 : ns(32)) {
  always {
    q1 = d1[ 7: 0] #
         d1[15: 8] #
         d1[23:16] #
         d1[31:24];
  }
}

dp top {
  sig d, q : ns(32);
  use myarm;
  use port1(d);
  use port2(q);
  use mmswap(d, q);
}

system S {
  top;
}
```

The C driver program for this memory-mapped coprocessor looks as follows.

```
volatile unsigned int * mmin = (unsigned int *) 0x80000000;
volatile unsigned int * mmot = (unsigned int *) 0x80000004;
for (i=0; i<4096; i++) {
  *mmin = in[i];
  ot[i] = *mmot;
}
```

The equivalent StrongARM assembly program for this loop looks as follows.

```
.L21:
    ldr r3, [r0, ip, asl #2]    ; load in[i]
    str r3, [r4, #0]            ; send to coprocessor
    ldr r2, [r5, #0]            ; read from coprocessor
    str r2, [r1, ip, asl #2]    ; write ot[i]
    add ip, ip, #1              ; next iteration
    cmp ip, lr
    ble .L21
```

The execution time of the loop body now takes ten clock cycles per iteration (seven instructions, one stall for branching, two stall for data-dependencies). This is a gain of three cycles. We assume that we have single-cycle access to the memory-mapped coprocessor, which is hard to achieve in practice. Hence, the gain of three cycles will probably be lost in a real design. The problem with this design is apparent from the assembly program: each data element travels *four* times over the memory bus in order to be converted.

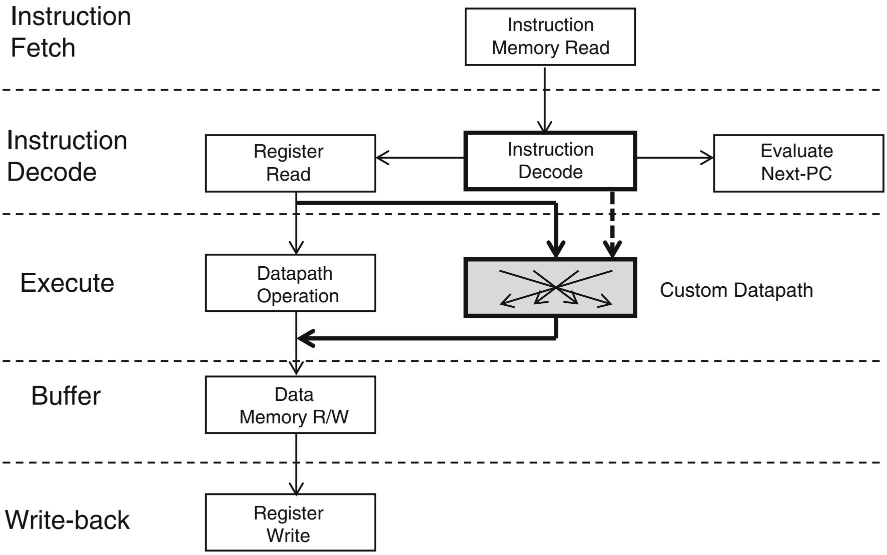


Fig. 11.14 Single-argument custom-datapath in an ASIP

Using an ASIP, this wasteful copying over the memory bus can be avoided: we can retrieve `in[i]` once, convert it inside of the processor, and write back the converted result to `ot[i]`. In order to do this as part of an instruction, we need to modify (extend) the datapath of the processor with a new operation. Figure 11.14 illustrates how this works: the execution stage of the pipeline is extended with a new datapath (endianness conversion), and a new instruction is integrated into the instruction decoder.

GEZEL supports experiments with custom datapaths in a StrongARM processor, by using several unused opcodes of that processor. In particular, GEZEL supports 2-by-2 and 3-by-1 custom datapaths in a StrongARM. A 2-by-2 custom datapath reads two register operands and produces two register operands. A 3-by-1 custom datapath reads three register operands and produces a single register operand. The following GEZEL program shows a 2-by-2 custom datapath for endianness conversion. An `ipblock` of type `armsfu2x2` is used to represent a custom instruction.

```
ipblock myarm {
  iptype "armsystem";
  ipparm "exec=nettohost_sfu";
}

ipblock armsfu(out d1, d2 : ns(32);
               in  q1, q2 : ns(32)) {
  iptype "armsfu2x2";
  ipparm "core = myarm";
  ipparm "device=0";
}
```

```
dp swap(in  d1, d2 : ns(32);
        out q1, q2 : ns(32)) {
  always {
    q1 = d1[ 7: 0] #
         d1[15: 8] #
         d1[23:16] #
         d1[31:24];
    q2 = d2[ 7: 0] #
         d2[15: 8] #
         d2[23:16] #
         d2[31:24];
  }
}

dp top {
  sig d1, d2, q1, q2: ns(32);
  use armsfu(d1, d2, q1, q2);
  use swap (d1, d2, q1, q2);
  use myarm;
}
```

We will now write a C program to use this custom datapath. We need to make use of a custom opcode to trigger execution of the custom datapath. The GEZEL `armsfu` interface relies on the `smullnv` opcode, which is unused by the C compiler (`arm-linux-gcc 3.2`). Since `smullnv` cannot be written directly in C, its opcode is inserted in a regular program by making use of inline assembly. The following C snippet shows how to define an inline assembly macro, and how to call this instruction for a single-argument and a double-argument instruction.

```
#define OP2x2_1(D1, D2, S1, S2) \
  asm volatile ("smullnv␣%0,␣%1,␣%2,␣%3" : \
                "=&r"(D1), "=&r"(D2) : \
                "r"(S1), "r"(S2));

// use as a single-argument instruction
for (i=0; i<4096; i++) {
  OP2x2_1(ot[i], dummy1, in[i], dummy2);
}

// use as a dual-argument instruction
for (i=0; i<2048; i++) {
  OP2x2_1(ot[2*i], ot[2*i+1], in[2*i], in[2*i+1]);
}
```

The resulting assembly for the single-argument case now looks as follows. The loop requires eight cycles, used by six instructions and two stalls. This is a gain of two clock cycles compared to the previous case. Equally important is that the program now only needs half the bus transfers, since the coprocessor is integrated *inside* of the StrongARM.

Table 11.1 Performance summary for the endianness byte-ordering processor

Type	Software	Memory-mapped	ASIP (1x1)	ASIP (2x2)
Instructions	11	7	6	7
Pipeline stalls	2	3	2	2
Cycles/iteration	13	10	8	9
Cycles/word	13	10	8	4.5

```
.L38:
    ldr      r3, [r4, lr, asl #2]    ; load in[i]
    smullnv  r2, r7, r3, ip          ; perform conversion
    str      r2, [r1, lr, asl #2]    ; write ot[i]
    add      lr, lr, #1              ; next iteration
    cmp      lr, r0
    ble      .L38
```

The dual-argument design is even more efficient, because the loop management code is now shared over two endianness conversions. We have nine cycles per loop iteration: seven instructions and two stalls. However, each iteration of the loop performs two conversions, so that the effective cycle cost is 4.5 cycles per endianness conversion (compared to eight cycles in the previous case).

```
.L53:
    ldr      r1, [r5], #4      ; read in[2*i], adjust pointer
    ldr      r2, [r5], #4      ; read in[2*i+1], adjust pointer
    smullnv  r0, ip, r1, r2    ; perform conversion
    str      r0, [r4], #4      ; store ot[2*i], adjust pointer
    subs     lr, lr, #1        ; next iteration
    str      ip, [r4], #4      ; store ot[2*i+1], adjust pointer
    bne      .L53
```

Summing up, by converting an all-software design to an ASIP-type design, the cycle cost for endianness conversion on a StrongARM reduces from 13 cycles per word to 4.5 cycles per word, an improvement of 2.9 times. Table 11.1 summarizes the performance of different configurations.

What is the limiting factor of the final design (ASIP 2x2)? Based on the code listing, 4 of the 7 instructions are load and store instructions. In other words, the majority of the execution time is spent in moving data between memory and the processor. This illustrates a point we made earlier: the strength of an ASIP design is also its weakness. An instruction-set architecture is convenient to build extensions, but bottlenecks in the instruction-set architecture will also be bottlenecks in the resulting hardware/software codesign.

11.3.3 Example: The Nios-II Custom-Instruction Interface

The Nios-II processor has an custom-instruction interface to attach hardware modules. Following our earlier definition of ASIP vs. coprocessor interface, we

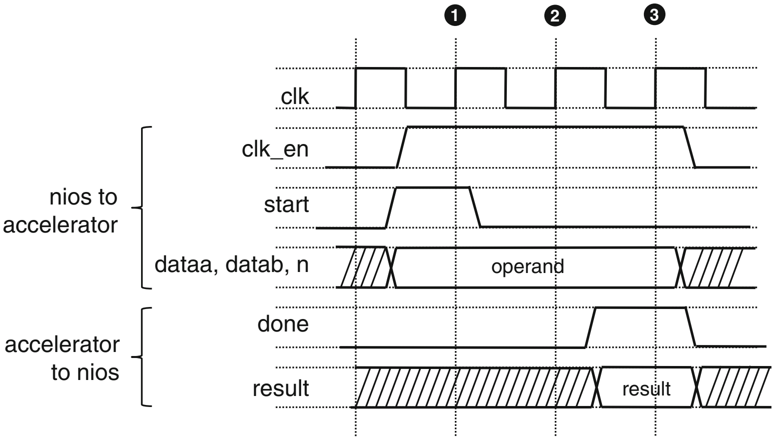


Fig. 11.15 Nios-II custom-instruction interface timing

should call this interface a coprocessor interface, since the Nios-II has predefined instructions to move operands through it. However, the look and feel of the interface closely resembles that of a custom-instruction in an ASIP. For this reason, the interface is introduced as a custom-instruction interface.

Figure 11.15 illustrates the timing of the interface. A custom instruction has three input operands: two 32-bit inputs `dataa` and `datab`, and an 8-bit input `n`. The result of the custom instruction is captured in a 32-bit output `result`. The interface supports variable-length execution of custom instructions through a two-way handshake mechanism. As shown in Fig. 11.15, the execution of the custom instruction starts with the Nios processor sending arguments to the custom hardware module at clock edge 1. The custom hardware uses a `done` output pin to indicate completion of the custom instruction at clock edge 3. The `clk_en` input is used to mask off the clock to the custom hardware when the instruction is not active.

In software, the custom-instruction is executed through a dedicated instruction, `custom`. For example, `custom 0x5, r2, r3, r5` would assign the value `0x5` to the `n` input, and associate the ports `dataa`, `datab`, and `result` with registers `r2`, `r3` and `r5` respectively. Figure 11.16a shows the result.

The `n` input is part of the instruction opcode, and is not a value in a register file. The purpose of the `n` input is to multi-plex different custom instruction implementations within the custom hardware module.

The Nios custom instruction interface also supports the use of local register files in the custom hardware module. This can be used to store local parameters for the hardware, and it can reduce the need for processor registers. Figure 11.16b illustrates this case, and shows how the custom-instruction interface needs to be extended. Only the case for the first input operand (a) is shown. An additional control signal, `reada`, selects if the custom hardware needs to accept data from the Nios register file or else from the local register file. In the former case, the input `dataa` contains the value of the processor register to be used as operand. In the latter case, the input

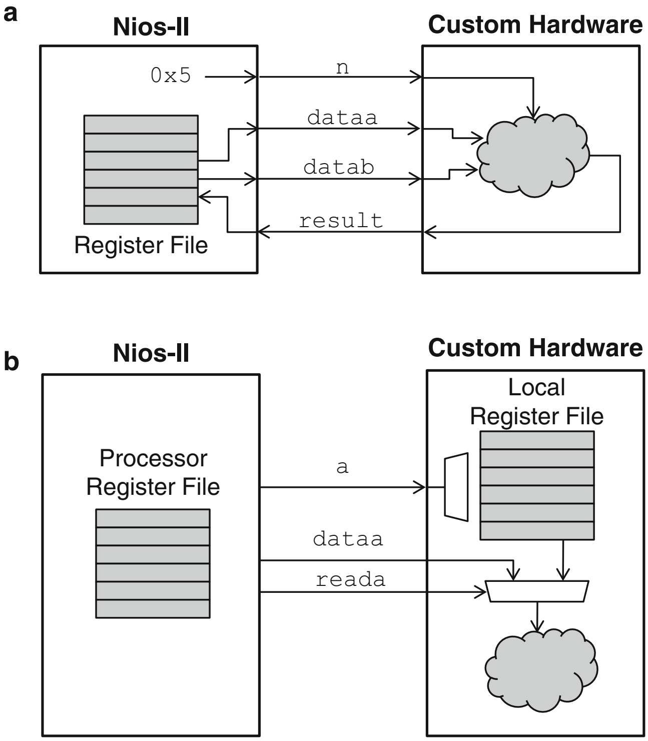


Fig. 11.16 Nios-II custom-instruction integration. (**a**) With processor register file. (**b**) With local register file

`a` indicates the address of the register field in the local register file to use as operand. To the software programmer, selection between these two cases is seamless, and the custom-instruction mnemonic integrates both. Registers prefixed with `r` are located in the processor, while registers prefixed with `c` are located in the custom hardware. Thus, an opcode such as `custom 0x5, c2, c3, r5` would take operand a and b from the local register file, and store the result in the processor register file.

11.3.4 Finding Good ASIP Instructions

How do we identify good ASIP instructions? Most likely, the application domain itself will suggest what type of primitives are most frequently needed. For example, image processing often works with 8-bit pixels. Hence, instructions that support efficient 8-bit operations will be useful for an image-processing ASIP. Another example, coding and cryptography make use of modular arithmetic. Hence, in an ASIP for coding algorithms, it makes sense to have support for modular addition and multiplication.

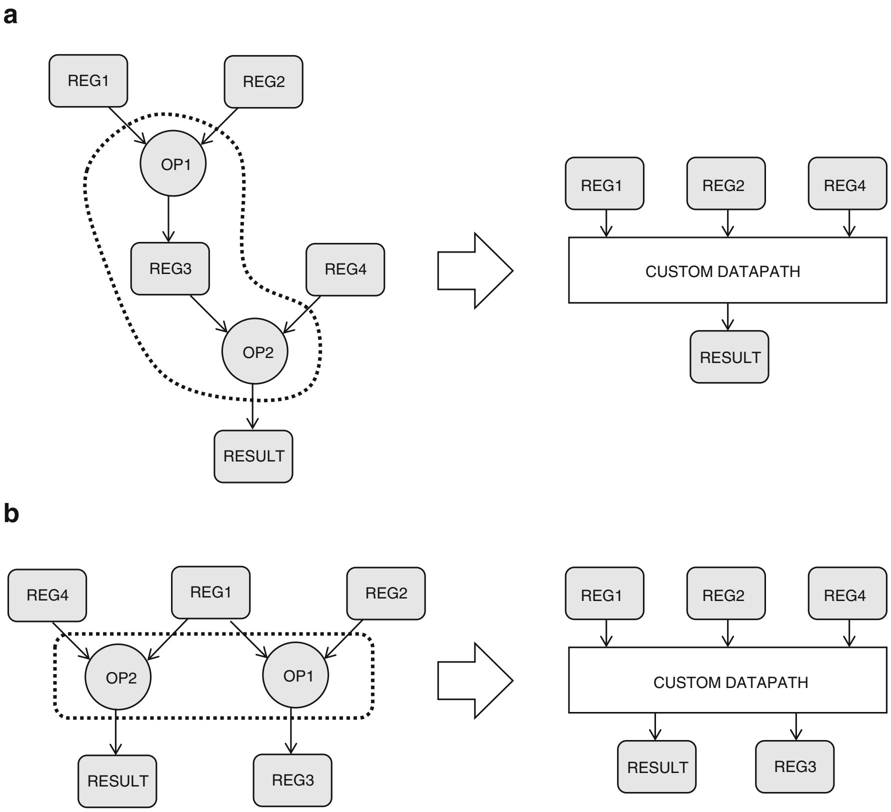


Fig. 11.17 (**a**) Operator fusion. (**b**) Operator compounding

The study of automatic instruction definition is a research field on its own, and is of particular interest to compiler developers. We will describe two basic techniques that work directly at the level of assembly language, and that are not specific to a given application domain. The first technique is called *operator fusion*, and the second technique is called *operator compounding*.

In operator fusion, we define custom instructions as a combination of dependent operations. The dependencies are found by means of data flow analysis of the code. After data flow analysis, we can cluster assembly operations together. When we group two operations together, the intermediate register storage required for the individual operations disappears. Figure 11.17a illustrates operator fusion. There are obviously a few limitations to the clustering process.

- All operations that are fused are jointly executed. Hence, they must be at the same loop hierarchy level, and they must be within the same branch of an if-then-else statement. Note that it is still possible to integrate an entire if-then-else statement as a custom instruction; see Problem 11.5.

- The number of input and output operands must be limited to ensure that the register-file bandwidth stays within bounds. Indeed, as the new custom instruction executes, it will require all the input arguments to be available at the same clock cycle, and it will produce all output arguments at the same clock cycle.
- The length of the chained operations must not be too long, since this adversely affects the critical path of the processor.

Figure 11.17b shows an example of operator compounding. In this case, we are combining multiple possibly unrelated operations together, especially when they share common inputs. The operator compounds are identified using data-flow analysis of the assembly code, and they have similar limitations as fused operators.

We'll now consider the endianness-conversion example once more and consider how much more it can be optimized beyond a single `OP2x2` ASIP instruction for endianness conversion. In this case, we consider a complete C function as follows. Notice how this code has been optimized with incremental pointer arithmetic.

```
void byteswap(unsigned *in, unsigned *ot) {
  int i;
  int d1, d2, d3, d4;
  unsigned *q1 = in;
  unsigned *q2 = ot;
  for (i=0; i<2048; i++) {
    d1 = *(q1++);
    d2 = *(q1++);
    OP2x2_1(d3, d4, d1, d2);
    *(q2++) = d3;
    *(q2++) = d4;
  }
}
```

The assembly code for this function looks as follows. The loop body contains nine instructions. Only a single instruction performs the actual byte swap operation! Four other instructions are related to moving data into and out of the processor (`ldr`, `str`), two instructions do loop counter management (`cmp`, `add`), one instruction implements a conditional return (`ldmgtfd`) and one instruction is a branch. There will be two pipeline stalls: one for the branch, and one for the second memory-load (`ldr`).

```
byteswap:
    stmfd     sp!, {r4, r5, lr}      ; preserve registers
    mov       ip, #0                 ; init loop counter
    mov       lr, #2032              ; loop counter limit
    add       lr, lr, #15
.L76:
    ldr       r2, [r0], #4           ; load in[i]
    ldr       r3, [r0], #4           ; load in[i+1]
    smullnv   r4, r5, r2, r3         ; endianness conversion
    str       r4, [r1], #4           ; store ot[i]
    str       r5, [r1], #4           ; store ot[i+1]
    add       ip, ip, #1             ; increment loop counter
```

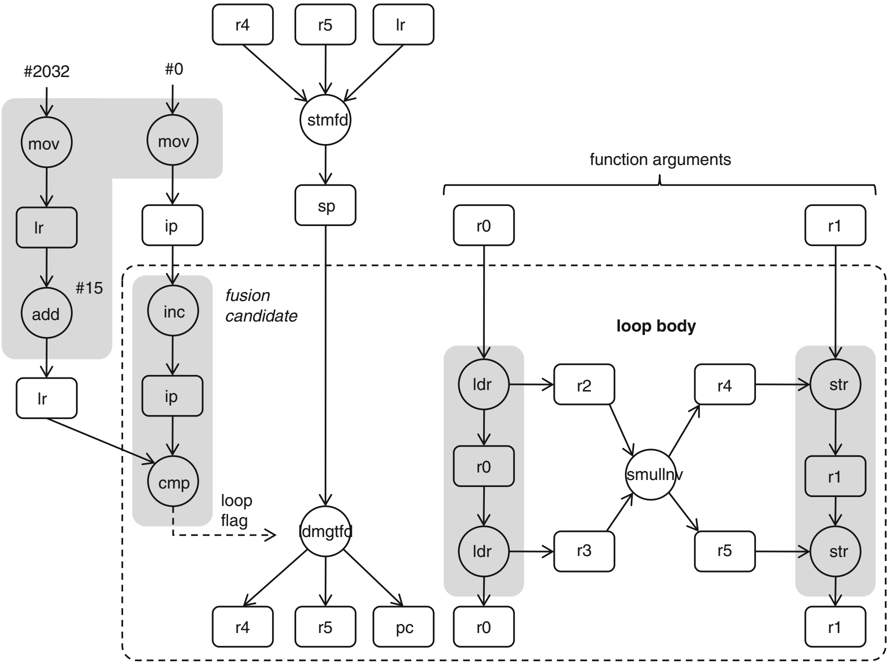


Fig. 11.18 Analysis of the `byteswap` function for ASIP instructions

```
cmp        ip, lr               ; test
ldmgtfd    sp!, {r4, r5, pc}    ; and conditionally return
b          .L76
```

To optimize this assembly code with additional ASIP instructions, we can construct a data-flow diagram for the assembly code, and investigate the opportunities for operation fusion and compounding. Figure 11.18 shows the dataflow analysis diagram. The boxes represent registers, while the circles represent operations. A distinction is made between operations inside of the loop and those outside of it; recall that fusion and compounding only work within a single loop level. The shaded clusters are examples of possible operation fusion.

- A set of fused operations could merge the loop counter logic into two new operations. The first operation initializes two registers. The second operation increments one register, compares it to the next, and sets a loop flag as a result of it. We will call these new operations `initloop` and `incloop` respectively.
- The second set of fused operations is trickier: they involve memory access (`str` and `ldr`), so they cannot be implemented with modifications to the execution stage of the RISC pipeline alone. In addition, since the StrongARM has only a single memory port, fusing these operations into a single operation will not provide performance improvement if there is not a similar modification to the memory architecture as well. Thus, we cannot define new instructions for these

fusion candidates. However, we may still reduce the footprint of the loop body by collapsing the store and load instructions in store-multiple and load-multiple instructions.

The resulting assembly code after these transformations would therefore look as follows. The loop body contains six instructions, but will have three stalls (one for `ldm`, one for the data dependency from `ldm` to `smullnv`, and one for the branch). A loop round-trip now costs 9 cycles, or 4.5 cycles per word. This is as fast as we found before, but in this version of the code, function call overhead is included.

```
byteswap:
       stmfd      sp!, {r4, r5, lr}  ; preserve registers
       initloop   ip, lr, #0, #2047  ; loop-counter instruction 1
.L76:
       ldm        r0!, {r2, r3}      ; load in[i], in[i+1]
       smullnv    r4, r5, r2, r3     ; endianness conversion
       stm        r1!, {r4, r5}      ; store ot[i], ot[i+1]
       incloop    ip, lr             ; loop counter instruction 2
       ldmgtfd    sp!, {r4, r5, pc}  ; and conditionally return
       b          .L76
```

This concludes our discussion of the third type of hardware/software interface, the custom instruction. The design of customized processors is an important research topic, and optimizations have been investigated far beyond the examples we discussed above. For the hardware/software codesigner, it's important to understand that ASIP design takes a top-down view on the codesign problem: one starts with a C program, and next investigates how to improve its performance. In a classic coprocessor design, the view is bottom-up: one starts with a given kernel in hardware, and next investigates how to integrate it efficiently into a system. Both approaches are viable, and in both cases, a codesigner has to think about interfacing hardware and software efficiently.

11.4 Summary

Microprocessor interfaces are at the core of hardware/software codesign. We made a distinction between three classes of interfaces, each with a different integration within the System on Chip architecture.

A memory-mapped interface reuses the addressing mechanism of an on-chip bus to reserve a few slots in the address space for hardware/software communication. Single memory locations can be implemented using memory-mapped registers. A range of memory locations can be implemented using a shared-memory. Specialized communication mechanisms, such as FIFO queues, further improve the characteristics of the hardware/software communication channel.

The coprocessor interface is a second type of hardware/software interface. It requires a dedicated port on a microprocessor, as well as a few predefined instructions to move data through that port. We discussed two examples of this interface, including the Fast Simplex Link interface, and a floating-point coprocessor interface for the LEON-3 processor.

The final hardware/software interface is the custom-instruction, created by modifying the instruction-set architecture of a micro-processor. This interface requires a rather detailed understanding of micro-processor architecture, and is often supported with a dedicated toolchain.

As a hardware/software codesigner, it is useful to spend time thinking about the breadth of this design space, which is enormous. Probably the most important point is to realize that there is no single silver bullet to capture all the variations of interconnections for hardware and software. There are many trade-offs to make for each variation, and different solutions can tackle different bottlenecks: computational power, data bandwidth, power consumption, design cost, and so on.

Also, keep in mind that no system is free of bottlenecks: the objective of hardware/software codesign is not to remove bottlenecks, but rather to locate and understand them. In the next chapter, we will focus on the hardware-side of custom-hardware modules, and describe how hardware can be controlled from within software through a hardware/software interface.

11.5 Further Reading

Early research in hardware/software codesign suggested that much of the hardware/-software interface problem can be automated. Chapter 4 in Micheli et al. (2001) describes some of the work in this area. To date however, no standards for creating hardware/software interfaces exist, and the design of such interfaces largely remains an ad-hoc process.

Yet, design support is critical to ensure error-free design. Designers often build virtual platforms of a chip during the implementation. A virtual platform is a complete simulation of the entire chip, emphasizing a detailed representation of the hardware/software interactions.

Memory-mapped interfaces are ubiquitous in the design of System-on-Chip architectures. One can consult the datasheet of a typical micro-controller and find that all peripherals are programmed or configured using memory-mapped input/output. For example, check the datasheet of Atmel AT91SAM7L128, an ARM-based microcontroller with numerous on-chip peripherals (Atmel 2008).

In contrast to general hardware/software interfaces, the literature on custom processor design is rich and detailed. In fact, the ASIP approach is one of the most successful models for hardware/software codesign when practical implementations are considered. A comprehensive treatment of the ASIP design process is provided by Rowen in (2004). Leupers and Ienne discuss customized processor architectures in Leupers and Ienne (2006). There are numerous publications on design applications based on ASIP, and as many conferences that cover them (a major event is Embedded Systems Week, grouping three conferences together on compiler design, on system design, and on embedded software implementation).

11.6 Problems

Problem 11.1. Build a memory-mapped register for the address bus described in Table 10.2. Evaluate the complexity of two different designs. What is the general recommendation for the design of memory-mapped address decoders you can make?

(a) A decoder that maps the register to any address of the range `0x3F000000 - 0x3F00FFFF`.
(b) A decoder that maps the register to the single address `0x3F000000`.

Problem 11.2. Modify the design of Fig. 11.3 so two integers can be transferred for each full request/acknowledge cycle. Thus, show that you can transfer twice as much data without changing the number of phases in the two-way handshake protocol.

Problem 11.3. Modify the design of Fig. 11.4 so that all positions of the FIFO are used before the `full` flag is set high.

Problem 11.4. Design a GEZEL implementation for a loopback interface on an Fast Simplex Link. Figure 11.19 illustrates the problem. The module has a single register, `loopreg`, that can hold the output of a single `putfsl` instruction. The module has a master interface for writing into the register, and a slave interface for reading from the register. Design the GEZEL program and a suitable testbench (in GEZEL).

Problem 11.5. Consider the C program in Listing 11.3, and the corresponding ARM assembly code in Listing 11.4.

(a) Study the following C program and the corresponding assembly code out of it.
(b) Define a custom instruction `max rx, ry`, which compares `ry` to the current value of `rx` and replaces that value if `ry` is bigger than `rx`. Assume that `rx` and `ry` are unsigned, 32-bit values.

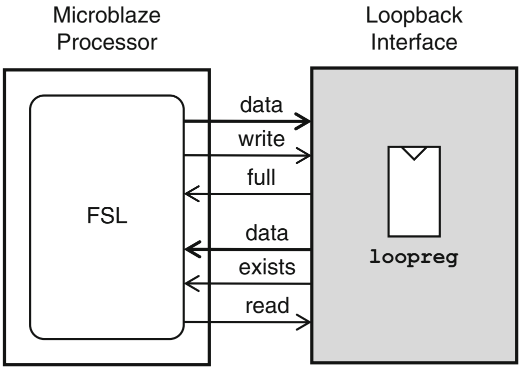


Fig. 11.19 The fast simplex link interface loopback interface for Problem 11.4

Listing 11.3 C program for Problem 11.5

```
int findmax(unsigned int data[1024]) {
  unsigned int max = 0;
  int i;
  for (i=0; i<1024; i++)
    if (max < data[i])
      max = data[i];
  return max;
}
```

Listing 11.4 Assembly program for Problem 11.5

```
findmax:
    mov         r2, #0
    mov         r1, #1020
    mov         ip, r2
    add         r1, r1, #3
.L7:
    ldr         r3, [r0, r2, asl #2]
    cmp         ip, r3
    strcc       ip, [r0, r2, asl #2]
    add         r2, r2, #1
    cmp         r2, r1
    movgt       r0, #0
    movgt       pc, lr
    b           .L7
```

Listing 11.5 C program for Problem 11.6

```
int absmax(int v, int w) {
  return (v * 6 + w * 4);
}
```

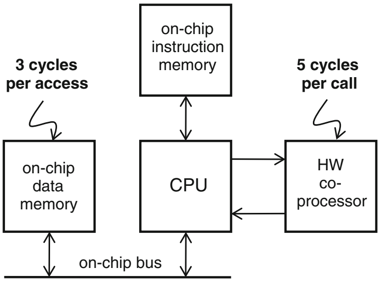
(c) Design a GEZEL datapath for this custom instruction, following the example in Sect. 11.3.2.

Problem 11.6. The C function of Listing 11.5 was compiled to a Microblaze processor and results in the assembly code of Listing 11.6. The input arguments of the assembly code are `r5` and `r6`; the return argument is `r3`; the return instruction was left out of the assembly.

(a) Perform dataflow analysis on the assembly code and draw the data dependency diagram below. Use rectangles to indicate registers and circles to indicate operations. Label the operations 'op1' to 'op6'.
(b) When you have drawn the dataflow dependency diagram, define several ASIP candidate instructions (at least 2) using operation fusion. Indicate clearly which operations you could merge into ASIP instructions.

Listing 11.6 Assembly program for Problem 11.6

```
// input arg: r5, r6
// return arg: r3
addk    r3,r5,r5    // op1
addk    r3,r3,r5    // op2
addk    r3,r3,r3    // op3
addk    r6,r6,r6    // op4
addk    r6,r6,r6    // op5
addk    r3,r3,r6    // op6
```



```
int a[1000], r[1000];
void transform() {
  int i;
  int *ra = a;
  int *rr = r;
  for (i=0; i<1000; i++)
    *rr++ = call_coproc(*ra++);
}
```

Fig. 11.20 Architecture and C program for Problem 11.7

Problem 11.7. The system-on-chip in Fig. 11.20 combines a coprocessor, a processor, and on-chip data- and instruction-memory. The processor will copy each element of an array `a[]` to the coprocessor, each time storing the result as an element of an array `r[]`. The C program that achieves this is shown on the right of Fig. 11.20. All the operations in this architecture take *zero* time to execute, apart from the following two: accessing the on-chip data memory takes three cycles, and processing a data item on the coprocessor takes five cycles.

(a) Find the resulting execution time of the C program.

(b) Show how you can rewrite the C program so that the resulting execution time becomes smaller than 8,000 cycles. As a hint, assume that you can rewrite the `y = call_coproc(x);` function call as two separate calls. The first call is `send_coproc(x);`, and it issues an argument from software to the coprocessor. The second call is `y = get_coproc();`, and it takes a result from the coprocessor.

Problem 11.8. An ASIP processor performs operations on a stream of samples. The samples appear at fixed rate. The samples are processed using an algorithm `A`. Which of the following optimizations will reduce the energy consumption `E` needed to process a single sample? The answer for each question is one of: yes, no, impossible to decide.

Chapter 12
Hardware Interfaces

12.1 The Coprocessor Hardware Interface

A hardware interface connects a custom hardware module to a coprocessor bus or an on-chip bus. The hardware interface steers the input- and output ports of the custom hardware module. This may affect many different activities of the custom hardware module, including data transfer as well as control. Figure 12.1 shows the location of the hardware interface in the overall integration of a microprocessor with a custom-hardware module.

12.1.1 Functions of the Coprocessor Hardware Interface

The design of the hardware interface is a classic hardware/software codesign problem, that matches the flexibility of custom-hardware design to the realities of the hardware/software interface. For example, the hardware interface typically takes are of any of the following.

- **Data Transfer**: The hardware interface handles data transfer between software and hardware. This includes read/write transactions on the on-chip bus, using a master-protocol or a slave-protocol. In other cases, the hardware interface implements handshake sequences for a coprocessor bus. A hardware interface can be optimized for high-throughput and bursty data transfers, for example with a dedicated Direct Memory Address Controller.
- **Wordlength Conversion**: The hardware interface converts data operands of the custom-hardware module, which can be arbitrary in size and number, into data structures suitable for on-chip bus communication. For example, in a 32-bit system, software can deliver no more than 32-bits of data at a time, while the custom hardware module may be using a much wider 1,024-bit input bus. In that case, the hardware interface needs to support the conversion of a single

P.R. Schaumont, *A Practical Introduction to Hardware/Software Codesign*,
DOI 10.1007/978-1-4614-3737-6_12, © Springer Science+Business Media New York 2013

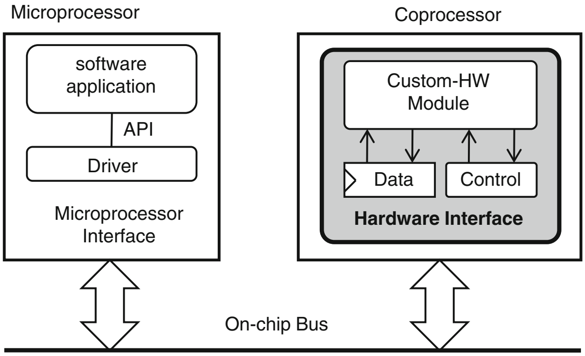


Fig. 12.1 The hardware interface maps a custom-hardware module to a hardware-software interface

1,024-bit operand into an array of 32-bit words. Furthermore, the hardware interface takes care of the bit- and byte-level organization of the data in case conversions between software and hardware are needed.

- **Operand Storage**: The hardware interface provides local and/or temporary storage for arguments and parameters for the custom-hardware component. Besides arguments and parameters, the hardware interface can also include local buffering to support the on-chip interface. In fact, the distinction between arguments and parameters is very important. Arguments are updated *every time* the custom-hardware module executes. Parameters, on the other hand, may be updated only infrequently. Hence, to minimize the hardware-software communication bandwidth, parameters are transmitted only once, and then held in a local memory in the hardware interface.
- **Instruction Set**: The hardware interface defines the software-view of a custom-hardware component in terms of instructions and data structures. The design of instruction-sets for custom hardware modules is a particularly interesting and important problem. A carefully-designed custom instruction-set can make the difference between an efficient coprocessor, and a confusing blob of logic.
- **Local Control**: The hardware interface implements local control interactions with the custom hardware component, such as sequencing a series of micro-operations in response to a single software command.

12.1.2 Layout of the Coprocessor Hardware Interface

Figure 12.2 illustrates the layout of a generic coprocessor hardware interface, which connects a custom hardware module to an on-chip bus interface or coprocessor interface. Since the hardware interface itself is a user-defined hardware component, it can have an arbitrary architecture. The following components are commonly found in a hardware interface.

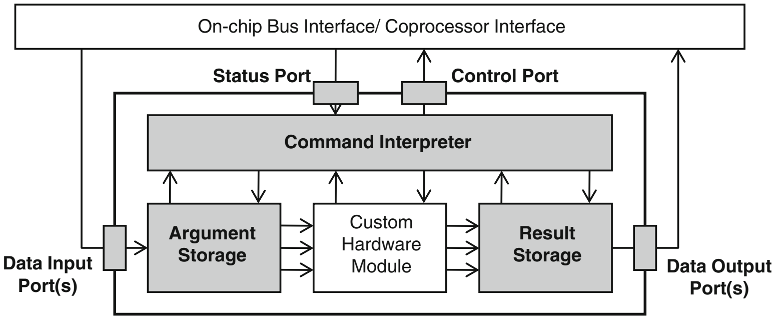


Fig. 12.2 Layout of a coprocessor hardware interface

- A data input buffer for *Argument Storage*.
- A data output buffer for *Result Storage*.
- A *Command Interpreter* to control the data buffers and the custom hardware module based on commands from software.

The hardware interface has several *ports*, addressable inputs/outputs of the coprocessor. For example, an on-chip bus interface uses an address decoder, while a coprocessor interface may have dedicated ports. From the perspective of the custom hardware module, it is common to partition the collection of ports into data input/output ports and control/status ports.

The separation of control and data is an important aspect in the design of coprocessors. Indeed, in a software program on a micro-processor, control and data are tightly connected through the instruction-set of the micro-processor. In custom-hardware however, the granularity of interaction between data and control is chosen by the designer. Observe that in Fig. 12.2, control signals and data signals are orthogonal: control flows vertically, and data flows horizontally.

Despite the relatively simple organization of the hardware interface, the design space of the data buffers and the command interpreter is rich and deep. In the following, we will discuss mechanisms to help us design the hardware interface efficiently. We will differentiate between *data-design* and *control-design*. Data-design implements data dependencies between software and hardware. Control-design implements control dependencies between software and hardware. We will discuss both of them.

12.2 Data Design

Data Design is the implementation of a mapping between the hardware interface ports and the custom hardware ports. Typically, this includes the introduction of buffers and registers, as well as the creation of an address map.

12.2.1 Flexible Addressing Mechanisms

A data port on a coprocessor has a *wordlength*, a *direction* and an *update rate*. For example, the wordlength and direction could be a 32-bit input. The update rate expresses how frequently a port changes value. The two extremes are a parameter, which needs to be set only once (after module reset), and a function argument, which changes value upon each execution of the custom hardware module.

To make a good mapping of actual hardware ports to custom interface ports, we start from the wordlength, direction, and update rate of the actual hardware ports. For example, consider a coprocessor for the greatest-common-divisor function, `gcd`. The high-level specification of this function would be:

```
int gcd(int m, int n);
```

The hardware module equivalent of `gcd` has two input ports `m` and `n`, which are 32-bit wide. The module also has a single output port of 32-bit. These three ports are the actual ports of the hardware module. When we implement this module as a memory-mapped coprocessor, the ports of the hardware interface will be implemented as memory-mapped registers.

A straightforward approach is to map each actual hardware port to a different memory-mapped register. This makes each port of the hardware module independently addressable from software. However, it may not always be possible to allocate an arbitrary number of memory-mapped ports in the hardware interface. In that case, we need to *multiplex* the custom-hardware module ports over the hardware interface ports.

12.2.2 Multiplexing and Masking

There are several occasions when the ports of the hardware module need to be multiplexed over the ports of the hardware interface.

- The hardware interface may have insufficient ports available to implement a one-to-one mapping between hardware-module ports and control-shell ports.
- Some hardware-module ports need to be programmed only once (or very infrequently), so that it is inefficient to allocate a separate hardware interface port for each of them.

Multiplexing will increase the control complexity of the hardware interface slightly. In addition, careless multiplexing will reduce the available input/output bandwidth of the hardware module. Thus, there is a risk that the module becomes communication-constrained because of the multiplexing process.

Multiplexing can be implemented in different ways. The first is *time-multiplexing* of the hardware module ports. The second to use an *index register* in the hardware interface. Figure 12.3 shows an example of a time-multiplexed port for the GCD

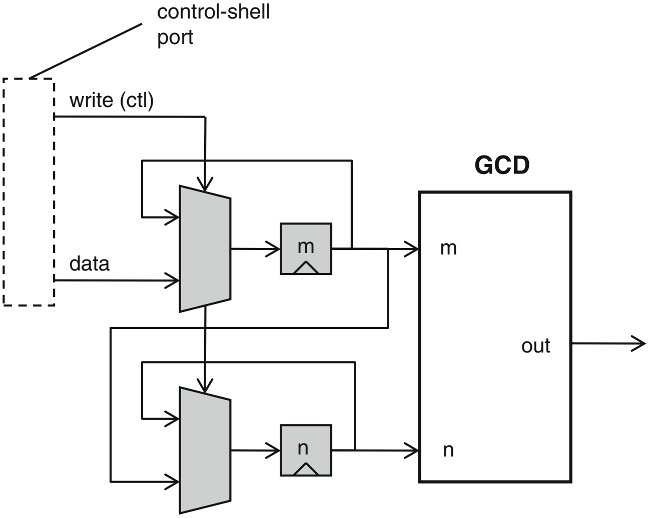


Fig. 12.3 Time-multiplexing of two hardware-module ports over a single control-shell port

coprocessor. In this case, the arguments need to be provided by writing the value of m and n sequentially to the hardware interface port.

The index-register technique works as follows. Several ports (say N) on the hardware module are mapped into two ports on the hardware interface. One port on the hardware interface is a data port of sufficient width to hold any single hardware module port. The second port is an index port and has width log_2N bits. The index port controls the mapping of the data port of the hardware interface to one of the ports on the hardware module. Figure 12.4 shows an example of the index-register technique to merge eight outputs to a single data output. The index register technique is more flexible than time-multiplexing, because the interface can freely choose the readout order of the hardware-module output ports. At the same time, it also requires double the amount of interface operations. Hence, multiplexing with index-registers is most useful for ports that update very infrequently, such as parameters.

Multiplexing is also useful to handle operands with very long wordlengths. For example, if the hardware module uses 128-bit operands, while the hardware-interface ports are only 32-bit, then the operands can be provided one word at-a-time by means of time-multiplexing.

Finally, here's a technique to work with very short operands, such as single-bit arguments. In this case, it is expensive to allocate a single hardware interface port for each single-bit hardware-module port. Instead, several hardware-module ports can be grouped together in a hardware interface port. To enable sharing of the hardware interface port among logically independent hardware-module port, an additional *mask register* is used. The mask register indicates which bits of a hardware interface port should be taken into account when updating the

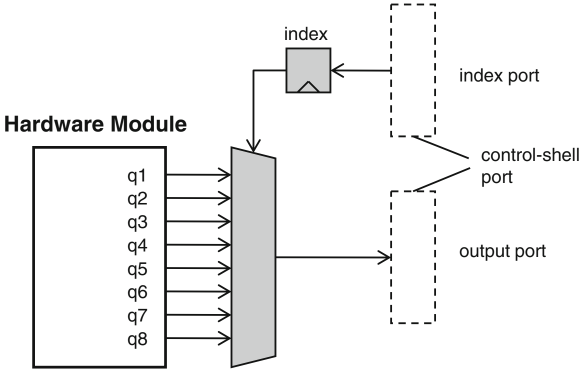


Fig. 12.4 Index-register to select one of eight output ports

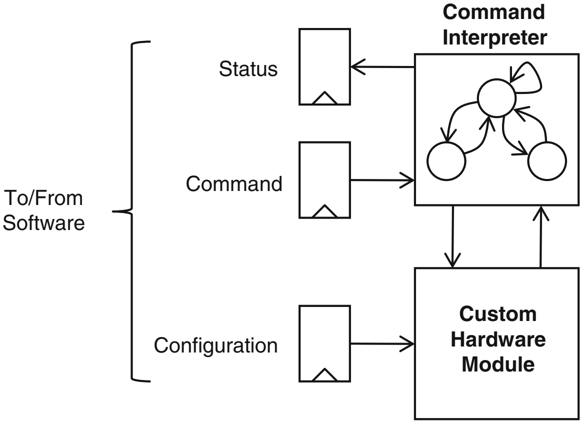


Fig. 12.5 Command design of a hardware interface

hardware-module ports. The updated value is obtained by simple bit-masking of the previous value on the hardware ports with the new value of the hardware interface port.

```
new_hardware_port = (old_hardware_port & ~mask_value) |
                    (control_shell_value & mask_value);
```

12.3 Control Design

Control design in a coprocessor is the collection of activities to generate control signals and to capture status signals. The result of control design is a set of *commands* or instructions that can be executed by the coprocessor. These commands are custom-tailored for the design.

Figure 12.5 shows a generic architecture to control a custom hardware module through software. It includes a *command interpreter* which accepts commands from software and which returns status information. The command interpreter is the

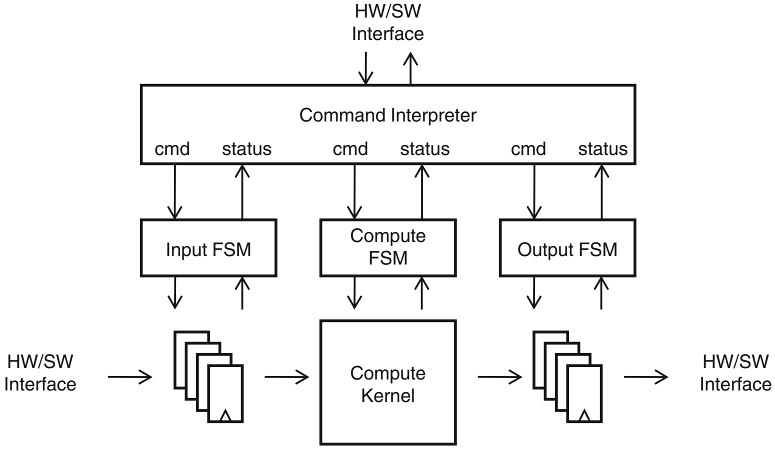


Fig. 12.6 Hierarchical control in a coprocessor

top-level controller in the coprocessor, and it communicates directly with software. We also make a distinction between a *command* and a *configuration*. A command is a one-time control operation. A configuration is a value which will affect the execution of the coprocessor over an extended period of time, possibly over multiple commands.

The following sections discuss several architectural techniques that can be used to optimize the performance of the coprocessor.

12.3.1 Hierarchical Control

Figure 12.6 shows the architecture of a coprocessor that can achieve communication/computation overlap. The coprocessor has a hierarchy of controllers, which allow independent control of the input buffering, the computations, and output buffering. The command interpreter analyzes each command from software and splits it up into commands for the lower-level FSMs. In the simplest form, these commands are simple start/done handshakes. Thus, for each command of software, the command interpreter runs a combination of lower-level FSM sequences. Often, a single level of command decoding is insufficient. For example, we may want to use a coprocessor which has an addressable register set in the input or output buffer. In that case, we can embed the register address into the command coming from software. To implement these more complicated forms of subcontrol, the start/done handshakes need to be replaced with more complex command/status pairs.

A control hierarchy simplifies the design of control, as is shown in Fig. 12.7. The command interpreter can easily adapt to the individual schedules from the input FSM, compute FSM and output FSM. On the other hand, the method of

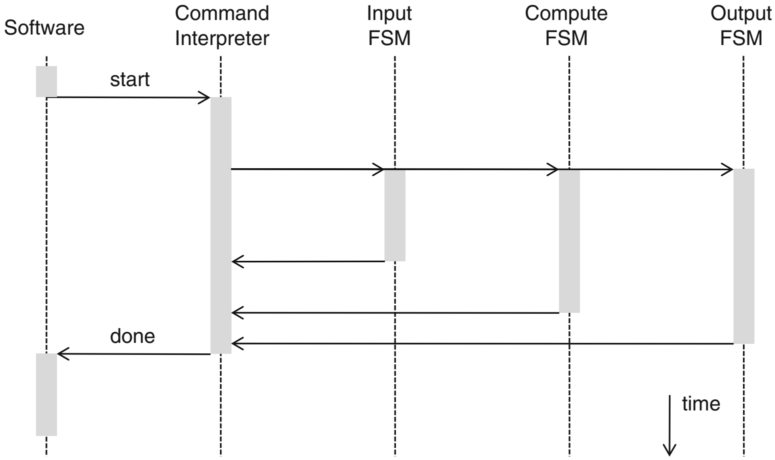


Fig. 12.7 Execution overlap using hierarchical control

Table 12.1 Command Set to control block-level pipelining

Command	Input FSM	Compute FSM	Output FSM
`pipe_load1`	Start		
`pipe_load2`	Start	Start	
`pipe_continue`	Start	Start	Start
`pipe_flush1`		Start	start
`pipe_flush2`			Start

using start/done pulses is inconvenient when working with pipelined submodules, since a done pulse only appears after the pipeline latency of the submodule. This will require a modification to the start/done scheme, which will be discussed later.

First, we examine how a hierarchical controller as illustrated in Fig. 12.6 can help in achieving computation/communication overlap. The basic principle, of course, is well known: we need to pipeline the input/compute/output operations within the coprocessor. Table 12.1 illustrates a set of five software commands to achieve pipelined execution within the coprocessor. This scheme is called *block-level pipelining*. The commands obtain precise pipeline startup and shutdown. The first three commands (`pipe_load1`, `pipe_load2`, `pipe_continue`) require the software to send an argument to the coprocessor. The last three commands (`pipe_continue`, `pipe_flush1`, `pipe_flush2`) require the software to retrieve a result from the coprocessor. Once the pipeline is filled up through the sequence of `pipe_load1` and `pipe_load2`, the software can repeat the command `pipe_continue` as often as needed.

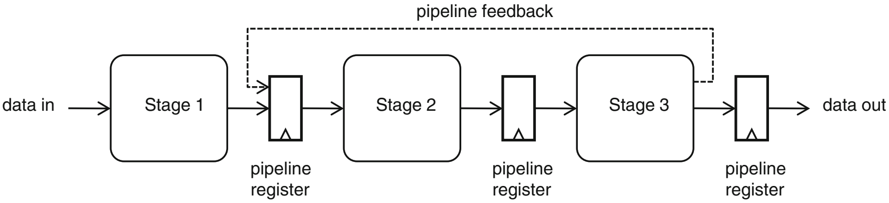


Fig. 12.8 Pipeline terminology

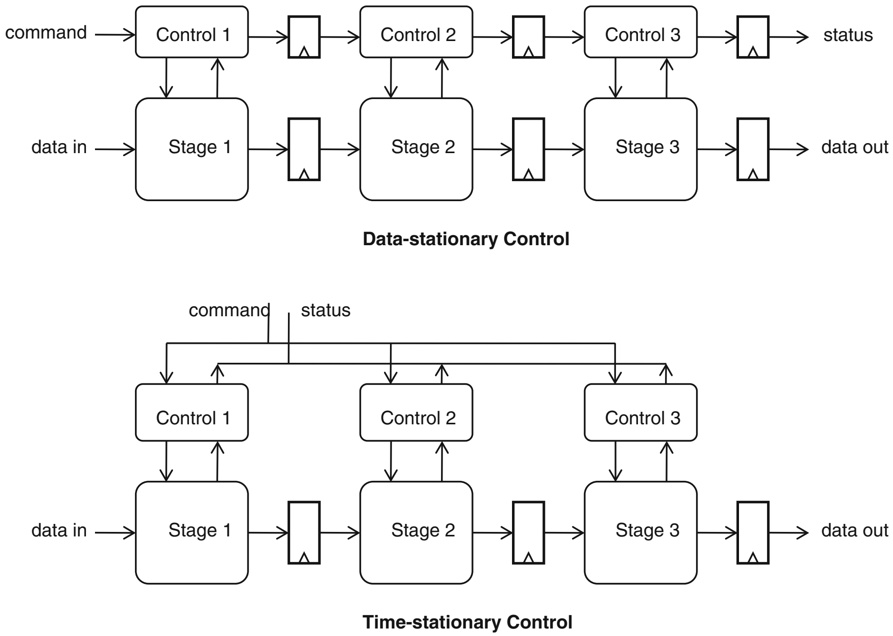
12.3.2 Control of Internal Pipelining

When a custom hardware module has internal pipeline stages, hierarchical control becomes more intricate. There are two issues of concern in the design of a hardware interface. First, we need to find a way to generate control signals for the pipeline stages. Next, we need to define a proper mechanism to interface these control signals to the higher layers of control. Indeed, a simple start/done handshake is insufficient for a pipeline, because it does not reflect the pipeline effect. In this section we will address both of these aspects.

Figure 12.8 introduces some terminology on pipeline architectures. A pipeline consists of a number of pipeline stages separated by pipeline registers. The latency of a pipeline is the number of clock cycles it takes for an operand to move from the input of the pipeline to the output. The throughput of a pipeline measures the number of results produced per clock cycle. If a pipeline accepts a new operand each clock cycle, its throughput is one (per cycle). If, on the other hand, a pipeline accepts a new operand every N cycles, its throughput is $1/N$. In a *linear pipeline* architecture, there are no feedback connections. For such a pipeline, the latency equals the number of pipeline stages, and the throughput equals 1. In a *non-linear pipeline* architecture, there are feedback connections. This happens when certain stages of a pipeline are reused more than a single time for each data token that enters the pipeline. In a non-linear pipeline, the latency can be higher than the number of pipeline stages, and the throughput can be lower than 1.

In any pipelined coprocessor, the pipeline control signals are eventually under the control of a higher-level controller. Pipeline control signals will be required in two cases.

- When a pipeline needs to perform more than a single operation, there needs to be a way to send control information into the pipeline. This control information will determine the operation of individual pipeline stages.
- In a non-linear pipeline architecture, multiplexers may be needed between the pipeline stages in order to feed operands from multiple locations within the pipeline. These multiplexers need additional control signals.



Data-stationary Control

Time-stationary Control

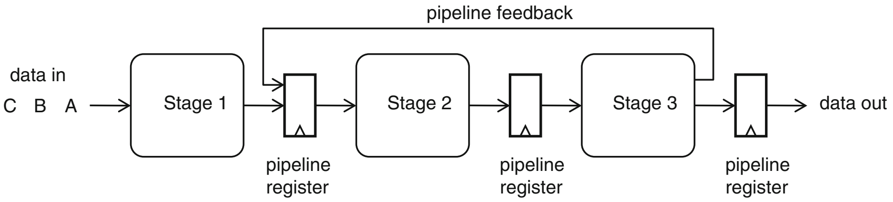
Fig. 12.9 Data-stationary and time-stationary pipeline control

12.3.2.1 Control of Linear Pipelines

We first discuss two techniques to control multi-function pipelines. The first is called *data-stationary* control, while the second is called *time-stationary* control. Figure 12.9 illustrates the differences between these two schemes.

- In a data-stationary scheme, control signals will travel along with the data through the pipeline. At each pipeline stage, the control word is decoded and transformed into appropriate control signals for that stage.
- In a time-stationary scheme, a single control word will control the entire pipeline for a single clock cycle. Because the pipeline contains fragments of different data items, each in a different stage of processing, a time-stationary control scheme will specify the operations to be performed on several data elements at the same time.

From a programmer's perspective, a data-stationary approach is more convenient because it hides the underlying pipeline structure in the program. A RISC instruction-set, for example, uses data-stationary encoding. On the other hand, time-stationary control makes the underlying machine structure explicitly visible in the control signals. Time-stationary control is therefore suitable for tasks that require access to the entire pipeline at once, such as exception handling. In addition,



Cycle	1	2	3	4	5	6	7	8	9
Stage1	A	B			C				
Stage 2		A	B	A	B	C		C	
Stage 3			A	B	A	B	C		C

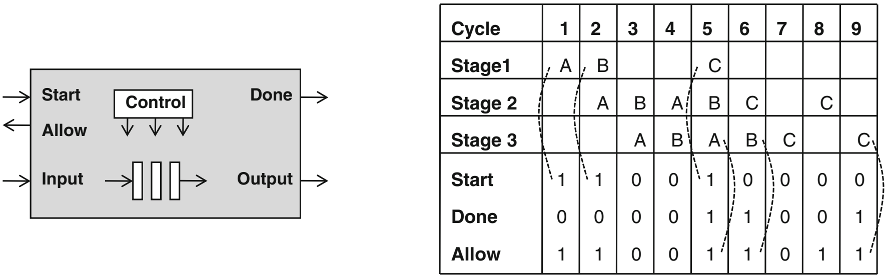
Fig. 12.10 A reservation table for a non-linear pipeline

non-linear pipeline architectures may be easier to control with a time-stationary approach then with a data-stationary approach. Indeed, generating the control signals for multiplexers between pipeline stages requires an overview of the entire pipeline.

12.3.2.2 Control of Non-linear Pipelines

When a pipeline has feedback connections, the pipeline stages are reused multiple times per data item that enters the pipeline. As a result, the throughput of a pipeline can no longer be 1.

Figure 12.10 illustrates the operation of a non-linear pipeline structure with three stages. Each data item that enters the pipeline will use stages 2 and 3 two times. When a new data item enters the pipeline, it will occupy stage 1 in the first cycle, stage 2 in the second, and stage 3 in the third cycle. The data item is then routed back to stage 2 for the fourth cycle of processing, and into stage 3 for the fifth and final cycle of processing. We can thus conclude that this non-linear, three-stage pipeline has a latency of 5. The diagram below Fig. 12.10 is a *reservation table*, a systematic representation of data items as they flow through the pipeline, with stages corresponding to rows and clock cycles corresponding to columns. The table demonstrates the pipeline processing of three data items `A`, `B` and `C`. From the diagram, we can see that data items `A` and `B` are processed in subsequent clock cycles. However, item `C` cannot immediately follow item `B`: the pipeline is busy and will occupy stages 2 and 3. Hence, item `C` will need to wait. This situation is called a pipeline *conflict*. In cycle 5, the pipeline is available again and item `C` can start. We conclude that the pipeline is able to process two elements (`A` and `B`) in four clock cycles. Therefore, the throughput of this pipeline is 0.5.



Cycle	1	2	3	4	5	6	7	8	9
Stage1	A	B			C				
Stage 2		A	B	A	B	C		C	
Stage 3			A	B	A	B	C		C
Start	1	1	0	0	1	0	0	0	0
Done	0	0	0	0	1	1	0	0	1
Allow	1	1	0	0	1	1	0	1	1

Fig. 12.11 Control handshakes for a non-linear pipeline

The operation of non-linear pipelines has been studied in detail, for example in the seminal work of Peter Kogge, but this material is beyond the scope of this chapter. Instead, we will consider the impact of pipelining on the generation of control handshake signals.

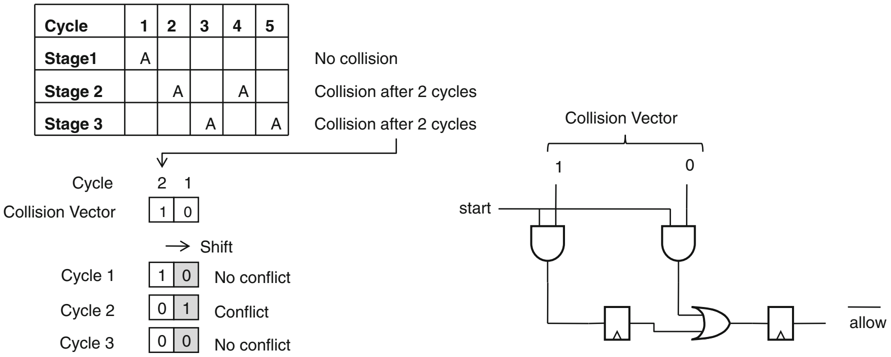
12.3.2.3 Control Handshakes for Pipelines

Earlier in this section we used a simple start/done handshake to implement hierarchical control for an iterated component. How do these handshake signals have to be modified for pipeline architectures?

In an iterated structure, a single `done` signal is sufficient to mark two distinguished but coinciding events. The first event is when a result is available at the output. The second event is when a new argument can be accepted at the input. In a pipeline structure however, input and output activities do not coincide. Indeed, in a pipeline, the latency does not have to be the reciprocal of the throughput. The case of a linear pipeline is still easy: the latency is equal to the number of pipeline stages, and the throughput is equal to 1. The case of a non-linear pipeline is more complex, and the latency as well as the throughput can both be different from 1.

To distinguish input events from output events, we will use *two* handshake-acknowledge signals. The first one, `done`, indicates when the pipeline produces valid data. The second one, `allow`, indicates when the input is able to accept new arguments.

Figure 12.11 illustrates the relation of the handshake signals to the operation of the pipeline. The left side of the figure illustrates the interface to the pipeline, while the right side shows the values for `start`, `done` and `allow` over several clock cycles. The beginning and end of a pipeline instruction are marked through `start` and `done`. The `allow` signal indicates if a new instruction can be started at the *next* clock cycle. If `allow` is zero, this means that starting an instruction will cause a pipeline conflict. You can observe that `done` is a delayed version of `start`, with the delay equal to the pipeline latency. The format of the `allow` signal is more



Cycle	1	2	3	4	5	
Stage1	A					No collision
Stage 2		A		A		Collision after 2 cycles
Stage 3			A		A	Collision after 2 cycles

Fig. 12.12 Generation of the `done` signal

complex, because it depends on the exact pattern of pipeline interconnections. For example, `allow` must be zero in cycles 3 and 4 because the second pipeline stage is occupied by instruction `A` and `B`.

Nevertheless the `allow` signal is easy to generate, as demonstrated in Fig. 12.12. The `allow` signal indicates when the pipeline is occupied and cannot accept a new instruction. For the reservation table shown in Fig. 12.12, this happens two clock cycles after a pipeline instruction starts. This duration of two clock cycles is called a *forbidden latency*. A *collision vector* is a bit-vector where forbidden latencies are marked by means of a bit. The index of this bit corresponds to the forbidden latency. For the reservation table shown, the collision vector equals `10`, since the only forbidden latency is two clock cycles. The `allow` signal can now be generated using the collision vector and a shift register, as shown on the right of Fig. 12.12. Each time a new instance of a pipeline instruction starts, a copy of the collision vector is added to the shift register. The last bit of this shift register indicates if the current clock cycle coincides with a forbidden latency. Hence, this bit is the inverse of the `allow` bit.

12.4 Programmer's Model = Control Design + Data Design

The previous two sections highlighted two aspects that affect control shell design. *Data design* is concerned with moving data from software to the encapsulated hardware module and back. *Control design* is concerned with generating control signals for the encapsulated hardware module.

In this section, we consider the impact of these design decisions on the software driver. The software view of a hardware module is defined as the *programmer's model*. This includes a collection of the memory areas used by the custom hardware module, and a definition of the commands (or instructions) understood by the module.

12.4.1 Address Map

The *address map* reflects the organization of software-readable and software-writable storage elements of the hardware module, as seen from software. The address map is part of the design of a memory-mapped coprocessor, and its design should consider the viewpoint of the software designer rather than the hardware designer. Here are some of the considerations that affect the design of the address map.

- To a software designer, *read* and *write* operations commonly refer to the *same* memory location. For a hardware designer, it is easy to route read and write operations to different registers, since read strobes and write strobes are available on a bus as separate registers. This practice should be avoided because it goes against the expectations of the software designer. A given memory-mapped address should always affect the same hardware registers.
- In the same spirit, a hardware designer can create memory-mapped registers that are read-only, write-only or read-write registers. By default all memory-mapped registers should be read/write. This matches the expectations of the software designer. *Read/write* memory-mapped registers also allow a software designer to conveniently implement bit-masking operations (such as flipping a single bit in a memory-mapped register). In some cases, *read-only* registers are justified, such as for example to implement registers that reflect hardware status information or sampled-data signals. However, there are very few cases that justify a *write-only* register.
- In software, read/write operations always handle aligned data. For example, extracting bits number 5–12 out of a 32-bit word is more complicated than extracting the second byte of the same word. While a hardware designer may have a tendency to make everything as compact as possible, this practice may result in an address map that is very hard to handle for a software designer. Hence, the address map should respect the alignment of the processor.

12.4.2 Instruction Set

The *instruction set* of a custom-hardware module defines how software controls the module. The design of a good instruction-set is a hard problem; it requires the codesigner to make the proper trade-off between flexibility and efficiency. Instructions that trigger complex activities in the hardware module may be very efficient, but they are difficult to use and understand for a software designer. The design of an instruction-set strongly depends on the function of the custom-hardware module, and therefore the number of generic guidelines is limited.

- One can distinguish *three classes* of instructions: one-time commands, on-off commands, and configurations. One-time commands trigger a single activity in the hardware module (which may take multiple clock cycles to complete). Pipeline control commands, such as discussed in Sect. 12.3.1, fall in the same category. On-Off commands come in pairs, and they control a continuous activity on the hardware module. Finally, configurations provide a parameter to an algorithm. They affect the general behavior of the hardware module. Making the proper choice between these is important in order to minimize the amount of control interaction between the software driver and the hardware module.

- *Synchronization* between software and hardware is typically implemented at multiple levels of abstraction. At the lowest level, the hardware/software interfaces will ensure that data items are transferred correctly from hardware to software and vice versa. However, additional synchronization may be needed at the algorithm level. For example, a hardware module with a data-dependent execution time could indicate completion to the driver software through a status flag. In this case, a status flag can support this additional layer of synchronization.
- Another synchronization problem is present when multiple software users share a single hardware module. In this case, the different users all see the same hardware registers, which may be undesirable. This synchronization issue can be handled in several ways. Coprocessor usage could be serialised (allowing only a single user at a time), or else a context switch can be implemented in the hardware module.
- Finally, *reset design* must be carefully considered. An example of flawed reset design is when a hardware module can only be initialized by means of full system reset. It makes sense to define one or several instructions for the hardware module to handle module initialization and reset.

12.5 Summary

In this chapter we discussed design techniques to encapsulate hardware modules onto a predefined hardware/software interface. These techniques are collected under the term *hardware interface design*. A hardware interface implements the connection mechanisms between low-level software and a hardware module, including the transfer of operands to and from the hardware module, and the generation of control signals for the hardware module. This requires, in general, the addition of data input/output buffers, and the addition of a controller.

Optimizing the communication between hardware and software is an important objective. One optimization technique is to improve the overlap between hardware/software communication and computation. This can be achieved by means of block-level pipelining and/or internal pipelining. Both forms of pipelining provide improved system-level performance, at the expense of additional hardware and increased complexity in system control.

Design of a good hardware interface is a challenging task; the application section of this book includes several in-depth examples.

12.6 Further Reading

Similar to hardware/software interface design, the implementation of coprocessor hardware interfaces is an ad-hoc design process.

The classic work on optimization of pipelined architectures is by Kogge (1981), and its ideas on scheduling of pipelined architectures are still relevant.

12.7 Problems

Problem 12.1. A C function requires 8 programmable constants that change very infrequently, and 1 data input that changes upon each call. Design a memory-mapped interface that minimizes the amount of memory locations required, while at the same time introducing minimal impact on the runtime performance of the design. Be precise: show a CPU bus on one side, and 8 + 1 registers on the other side, with your memory-mapped interface design in between. Assume the interface is mapped starting at address 0x100.

Problem 12.2. The Motorola DSP56000 processor is a pipelined processor. One of its assembly instructions look as follows.

```
MPY  x0, Y0, A     X: (R0)+, X0   Y: (R4)+, Y0
```

There are no comments in the above line – everything on that line is part of the instruction! This instruction multiplies register X0 with Y0 and places the product in the A accumulator. At the same time, the value of register X0 is updated with the memory location pointed to by registers R0, and Y0 is updated with the memory location pointed to by register R4. Does the Motorola DSP56000 use time-stationary control or does it use data-stationary control?

Problem 12.3. Listing 12.1 is a design of a control shell for a `median` module, which evaluates the median of three values. Study the operation of the coprocessor by studying the GEZEL code listing, and answer the questions below.

(a) How many data-input and data-output ports does the coprocessor have?
(b) Is the `median` module communication-constrained or computation-constrained with respect to this hardware/software interface?
(c) Describe how software should operate the coprocessor (write values to it, and retrieve the result from it).
(d) Write a C program that uses this coprocessor to evaluate the median value of the numbers 36, 99, and 58.

Problem 12.4. Listing 12.2 is a hardware implementation of a vector generator module. Given a set of input coordinates `(ix, iy)`, the module will generate all integer coordinates lying on the straight line between `(0,0)` and `(ix, iy)`. The algorithm implemented by the module, the Bresenham algorithm, is used to draw lines on raster-scan displays. The Listing 12.2 assumes that `(ix, iy)` lies

Listing 12.1 A hardware interface for a median module (Problem 12.3)

```
dp median(in  a, b, c : ns(32);
          out q       : ns(32)) {
  sig q1, q2 : ns(32);
  always {
   q1 = (a > b) ? a : b;
   q2 = (a > b) ? b : a;
   q  = (c > q1) ? q1 : (c < q2) ? q2 : c;
  }
}

ipblock myarm {
  iptype "armsystem";
  ipparm "exec=median_driver";
}

ipblock b_datain(out data : ns(32)) {
  iptype "armsystemsource";
  ipparm "core=myarm";
  ipparm "address=0x80000000";
}

ipblock b_dataout(in data : ns(32)) {
  iptype "armsystemsink";
  ipparm "core=myarm";
  ipparm "address=0x80000004";
}

dp medianshell {
  reg a1, a2, a3 : ns(32);
  sig q : ns(32);
  use median(a1, a2, a3, q);

  sig v_in, v_out : ns(32);
  use b_datain(v_in);
  use b_dataout(v_out);

  use myarm;

  reg old_v : ns(32);
  always {
    old_v = v_in;
    a1    = ((old_v == 0) & (v_in > 0)) ? v_in : a1;
    a2    = ((old_v == 0) & (v_in > 0)) ? a1   : a2;
    a3    = ((old_v == 0) & (v_in > 0)) ? a2   : a3;
    v_out = q;
  }
}

system S {
  medianshell;
}
```

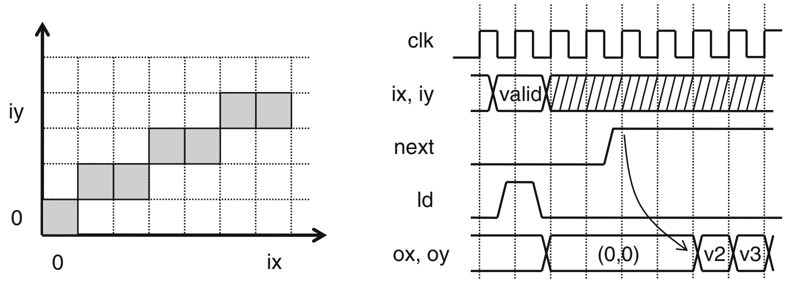


Fig. 12.13 Bresenham vector generation module (Problem 12.4)

in the first quadrant, so that `ix` and `iy` are always positive. The timing diagram on the right of Fig. 12.13 shows how to operate the module. New target coordinates can be entered using the `ld` control input. Loading new coordinates also resets the output coordinates to `(0,0)`. After that, the `next` control input can be used to retrieve new output points from the vector. The ouput will be refreshed on the second clock edge after `next` is high.

(a) Design a hardware interface for this module implemented as a coprocessor with a single 32-bit input port, and a single 32-bit output port. Optimize your design to take advantage of the fact that the coordinates of the module are 12-bit. Assume a memory-mapped interface. The design of the hardware interface includes: definition of the coprocessor instruction set, design of the hardware interface hardware, and design of sample driver software.

(b) How would you modify the design of (a) when the coprocessor needs to be connected through an 8-bit bus? Describe the required modifications to the instruction-set, the control-shell hardware, and the sample driver software.

Listing 12.2 A vector generator

```
dp bresen(in  ix, iy    : ns(12);
          in  ld, next  : ns( 1);
          out ox, oy    : ns(12)) {
  reg x, y              : ns(12);  // current position
  sig nx, ny            : ns(12);
  reg e                 : tc(12);
  reg eol               : ns(1);
  reg rix, riy          : ns(12);  // current target
  reg einc1, einc2      : tc(12);
  reg xinc, yinc        : ns(1);
  reg ldr, nextr        : ns(1);

  always {
    ldr   = ld;
    nextr = next;
    rix   = ld ? ix : rix;
    riy   = ld ? iy : riy;
    ox    = x;
    oy    = y;
  }

  sfg init {
    einc1 = (rix > riy) ? (riy - rix) : (rix - riy);
    einc2 = (rix > riy) ? riy         : rix;
    xinc  = (rix > riy) ? 1 : 0;
    yinc  = (rix > riy) ? 0 : 1;
    e     = (rix > riy) ? 2 * riy - rix : 2 * rix - riy;
    x     = 0;
    y     = 0;
  }

  sfg loop {
    nx    = (e >= 0) ? x + 1     : x + xinc;
    ny    = (e >= 0) ? y + 1     : y + yinc;
    e     = (e >= 0) ? e + einc1 : e + einc2;
    x     = nx;
    y     = ny;
    eol   = ((nx == rix) & (ny == riy));
  }

  sfg idle { }
}

fsm f_bresen(bresen) {
  initial s0;
  state s1, s2;
  @s0 (init)                         -> s1;
  @s1 if (ldr) then (init)           -> s1;
      else if (eol)   then (idle) -> s2;
      else if (nextr) then (loop) -> s1;
      else (idle)                    -> s1;

  @s2 if (ldr) then (init)           -> s1;
      else (idle)                    -> s2;
}

// testbench
dp test_bresen {
  reg ix, iy   : ns(12);
  reg ld, next : ns(1);
  sig ox, oy   : ns(12);
  use bresen(ix, iy, ld, next, ox, oy);

  always   { $display("<",$cycle, ">", $dec, " ox ", ox, " oy ",
               oy); }
  sfg init { ld = 1; next = 0; ix = 11; iy = 7; }
  sfg step { ld = 0; next = 1; ix =  0; iy = 0; }
}
fsm ctl_test_bresen(test_bresen) {
  initial s0;
```

Cycle	1	2	3	4
Stage1	A			
Stage 2		A		A
Stage 3			A	

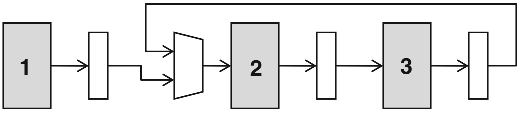


Fig. 12.14 A non-linear pipeline (Problem 12.5)

```
  state s1;
  @s0 (init) -> s1;
  @s1 (step) -> s1;
}

system S {
  test_bresen;
}
```

Problem 12.5. Figure 12.14 shows a non-linear pipeline architecture with three stages. The shaded blocks labeled 1, 2, and 3 represent combinational logic. Pipeline stage 2 iterates two times over each data item entered. As a result, this architecture can only process one data item every clock cycle.

(a) Find the forbidden latencies for this pipeline.

(b) Can this pipeline accept new data inputs at regularly spaced interfaces? If not, how could you modify this architecture so that this becomes possible?

Index

P.R. Schaumont, *A Practical Introduction to Hardware/Software Codesign*,
DOI 10.1007/978-1-4614-3737-6, © Springer Science+Business Media New York 2013