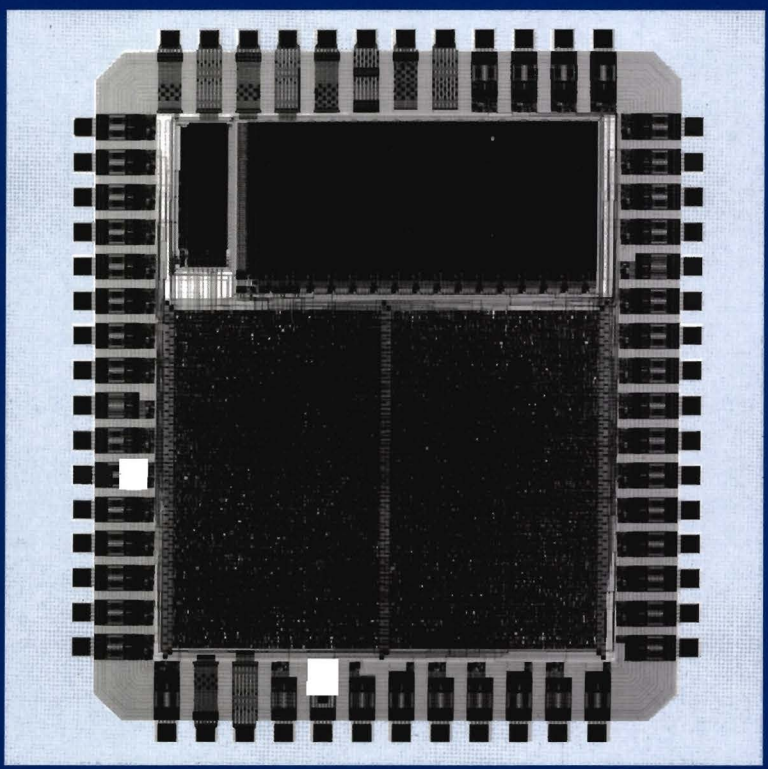
An asynchronous low-power 80C51 microcontroller

An Asynchronous Low-Power 80C51 Microcontroller

Hans van Gageldonk



An Asynchronous Low-Power 80C51 Microcontroller

Hans van Gageldonk

Cover: Layout of an asynchronous low-power 80C51 microcontroller.

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

van Gageldonk, Johan Sebastiaan Henri.

An Asynchronous Low-Power 80C51 Microcontroller / Hans van Gageldonk. -

Proefschrift Technische Universiteit Eindhoven. -

Met lit. opg. - Met samenvatting in het Nederlands.

ISBN 90-74445-42-X

Trefw.: asynchronous circuits, microcontrollers, low-power, IC-design, VLSI.

An Asynchronous Low-Power 80C51 Microcontroller

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR
AAN DE TECHNISCHE UNIVERSITEIT EINDHOVEN,
OP GEZAG VAN DE RECTOR MAGNIFICUS,
PROF. DR. M. REM,
VOOR EEN COMMISSIE AANGEWEZEN
DOOR HET COLLEGE VOOR PROMOTIES
IN HET OPENBAAR TE VERDEDIGEN
OP DINSDAG 1 SEPTEMBER 1998 OM 16.00 UUR

DOOR

JOHAN SEBASTIAAN HENRI VAN GAGELDONK

GEBOREN TE HEERLEN

Dit proefschrift is goedgekeurd door de promotoren:

prof. dr. M. Rem

en

prof. dr. ir. C. H. van Berkel

The work described in this thesis has been carried out at Philips Research Laboratories Eindhoven under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Contents

Acknowledgements

Over the last four years I had the opportunity to work in an excellent and inspiring environment, which eventually resulted in this thesis. I experienced this environment at Philips Research in the first place, as I spent most of my time as a member of the Tangram team. I would like to thank Kees van Berkel, the leader of this team, for teaching me that I should always try to improve the quality and presentation of my work. Together with Joep Kessels, Ad Peeters, Marly Roncken, Frits Schalij, and Rik van de Wiel, the Tangram team has been a very exciting team to work in. Eric van Utteren and Cees Niessen are gratefully acknowledged for giving me the opportunity to work in their group.

I was in the fortunate position to have another inspiring environment as well: the Parallelism group at Eindhoven University. I am especially grateful to Martin Rem for always being a source of inspiration and enthusiasm. Also the members of the VLSI-Club are gratefully acknowledged for providing a critical yet pleasant forum. Especially the comments of Peter Hilbers, Johan Lukkien, Tom Verhoeff, and Rudolf Mak helped me to improve the presentation of this thesis.

Steve Furber and Jochen Jess read the first draft of this thesis as members of the core committee, for which I would like to thank them.

The asynchronous 80C51 microcontroller that is described in this thesis became the result of a joint project with our colleagues from Philips Semiconductors in Zürich. I would specially like to thank Daniel Gloor, Daniel Baumann, Gerhard Stegmann, and Andreas Mettler for the lively discussions. Paul Gradenwitz and Thomas Meyer are gratefully acknowledged for making the layout that appears on the cover of this thesis.

I would like to thank Peter Klapproth, Pierre de Greef, and Eric Seelen for providing the data of the synchronous 80C51 that we compare the asynchronous version with. Furthermore, Victor Zieren and Harry van Herten are gratefully acknowledged for providing the photon emission pictures in Chapter 7 of this thesis.

ESPRIT Working Group 21949 (ACiD-WG) is gratefully acknowledged for funding my visits to workshops and conferences.

I am also very grateful to the fellow-PhD students both at Philips and at Eindhoven University: Ramon, Rik, Bart, Robert, Bob, Paul, and John: many thanks!

The support of my parents, family, and friends has always been very important to me, both in enjoying my work and my time off, for which I am very grateful to them. Finally, and most importantly, I would like to thank Mariken for all her support and love.

Chapter 1

Introduction

The power consumption of consumer-electronic products has become increasingly important over the last decade. Especially for products that rely on a limited source of power, for example battery-powered products, it is essential to keep the power dissipation to a minimum. This not only results in a longer battery-life but makes it also possible to use cheaper and smaller batteries, resulting in more appealing products.

Over recent years, increasingly many consumer products have been designed for portable use, and are hence battery powered. Examples of such products are cellular phones, pagers, personal digital assistants, and notebook computers. In a cellular phone, for example, the batteries run out after, say, 80 hours of standby time. The phone contacts its base station a few times per second to check whether a phone call has come in. This is not visible to the user who only notices the batteries running empty.

In these products the power dissipated by the ICs is a substantial part of the total power dissipation. Nowadays many chips are produced using CMOS IC-technology. CMOS has the nice property that no power is dissipated when there is no switching activity. Zooming in on digital ICs we see that much of the power (up to 50%) is dissipated by the *clock* in the chip [21]. The clock is used to drive the operation of the IC, and it determines the speed at which the circuit operates. In most ICs nowadays the clock is *global* to all register elements of the chip. This implies that the clock signal is distributed over the entire IC.

To reduce the power dissipated by the clock one alternative is to get rid of the clock in the first place, and use *distributed* and *selective* steering of the registers in the chip. One then speaks of an *asynchronous* as opposed to a *synchronous* (i.e.

clocked) IC.

Asynchronous circuits have for long been believed difficult to design, compared to synchronous ICs. It is especially difficult to design these circuits at the gate-level, where problems like timing of data validity and hazards arise. To solve these problems, at Philips Research a high-level programming language, Tangram, was defined; a compiler can translate a Tangram program in a syntax-directed fashion into the netlist of an asynchronous circuit. This translation is done *transparently*, and this makes it possible to reason about the circuit at the level of the Tangram text. The approach of translating a high-level description into an asynchronous circuit has also been described by Martin [23] and Brunvand [6].

This dissertation takes Tangram and the transparent compilation scheme as a starting point, and investigates techniques to reason about the resulting circuits. One can write various Tangram programs with the same functionality, but with different sizes, performances, and power consumptions. The programming techniques are explained using the 8-bit 80C51 microcontroller as example.

Parts of this work were reported during the 1998 Conference on Asynchronous Design Methodologies [52]. At Eindhoven University of Technology, a bibliography on asynchronous-related literature is maintained [28].

1.1 Tangram

Philips Research Eindhoven started in 1986 with the project "VLSI Programming and Silicon Compilation". In this project the design of a circuit is seen as a programming activity. To this end a simple, yet expressive programming language called Tangram was developed, together with a set of tools. Tangram is a language like Pascal or C, but offers extra constructs to express communication, parallelism, and reuse (*sharing*) of hardware. Tangram is based on Hoare's Communicating Sequential Processes CSP [16]. The tools are built around a compiler that translates a Tangram VLSI-program into the netlist of a circuit. This translation is done in two steps with *handshake circuits* as the intermediate representation. Handshake circuits were introduced by Van Berkel, and form the central part of the Tangram system [41].

A handshake circuit is a connected graph of so-called *handshake components*. Handshake components communicate and synchronize with each other using *handshake channels*. The communication between handshake components is based on a handshake (i.e. request-acknowledge) protocol. Each handshake component corre-

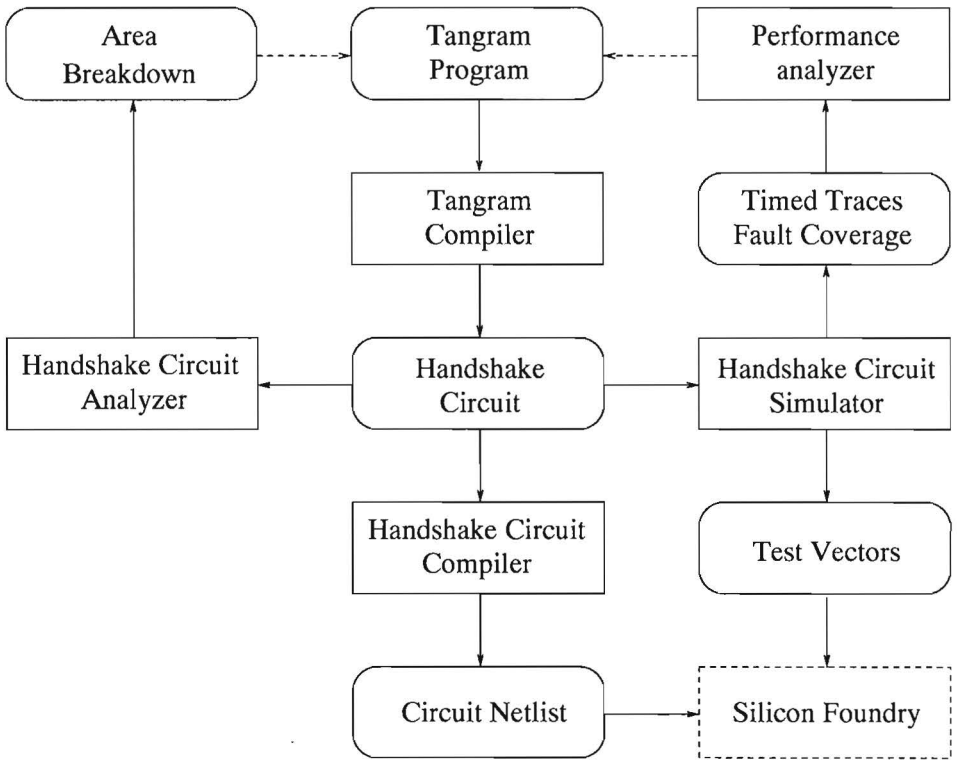


Figure 1.1: Tangram toolbox. The boxes denote tools; the ovals denote representations.

sponds to a construct in the Tangram language. A handshake circuit is an abstract view of a gate-level circuit. The next section introduces handshake circuits in more detail.

Together with the compiler various tools were designed and implemented to assist the VLSI-programmer in the design process. An overview of these tools is shown in Figure 1.1. In this design flow the VLSI-programmer (circuit designer) starts with a Tangram program which is first compiled to a handshake circuit. There are two tools that give feedback on the handshake circuit corresponding to the Tangram program:

Handshake Circuit Analyzer. This tool gives information about the *area* charac-

teristics of the circuit. Not only the numbers of transistors, cells, or gate-equivalents can be obtained, but also a breakdown into various classes of components: control, communication, logic, and memory. This helps the designer to get insight into the area characteristics of the circuit, and what to pay attention to when reducing the total area.

Handshake Circuit Simulator. This tool provides information about the *functionality* but also about the *timing* and *energy dissipation* of the circuit. The results can be shown to the designer using a viewer. The handshake circuit simulator also gives information about the *fault coverage* of the circuit for a given test. The designer generates the test vectors by hand, and the simulator shows data on coverage, as well as those parts of the circuit that are not covered by the test vectors. This data is presented to the designer on the level of the programming language. The *automatic* generation of test vectors for the circuit is subject of further research and implementation.

With the information obtained from the handshake circuit analyzer and the simulator, the VLSI-programmer can modify the Tangram program to obtain a "better" circuit in some respect. For example, the designer may want to reduce the area, improve the operating speed, or lower the power consumption. Compilation to a handshake circuit and simulation at this level is fast and allows for rapid feedback to the designer.

As the a next step in the design, VLSI-programmer compiles the handshake circuit to a netlist. The netlist can be simulated using a *gate-level simulator* to obtain accurate numbers on speed and energy dissipation. The translation of a handshake circuit to a netlist, and simulation at the gate-level is more accurate and thus more time-consuming than simulation at the handshake-circuit level. Therefore this design cycle is longer. When the VLSI-programmer is satisfied with the circuit, the netlist can be used for layout (using commercially available tools, for example) and sent to a silicon foundry for fabrication.

The compilation from a handshake circuit into a netlist is based on component-by-component substitution of handshake components by pieces of circuitry. The compilation also contains many optimizations at the gate-level, so-called *peephole* optimizations, as described by Peeters in his thesis [30]. These optimizations replace combinations of circuit elements by simpler ones that are smaller, faster, and more energy-efficient.

The translation step from a handshake circuit into a netlist can be done in various ways, resulting in circuits with different characteristics. One classification of these circuits is based on *timing assumptions* of the circuit implementation. One of the

first Tangram compilers assumed only the *isochronic fork*: this is a branch in a wire to inputs of gates of which it is assumed that the difference in delays between the branches is less than the delays through the gates to which the fork is an input [23]. This results in so-called *quasi delay insensitive* (QDI) circuits. QDI implementation of handshake circuits implies that for the encoding of data, a delay-insensitive encoding should be used [53]. This is satisfied by using *double-rail* encoding of data, in which two wires are used for communicating one bit. Communicating a "0" involves a signal on one wire; communicating a "1" is established by a signal on the other wire. The feasibility of this compilation was shown with a working IC already in 1987 [40]. A demonstrator of significant complexity is the error decoder for the DCC (Digital Compact Cassette) player as reported in [43, 44]. This error decoder consumes five times less power than its synchronous counterpart.

Though some QDI implementations show a remarkable advantage in power compared to their synchronous counterparts, their drawback is two-fold [43, 44]. First, the area overhead, typically between 70% and 100%, is in general very large for the ICs to go into production. Second, using the double-rail implementation, special cells for layout are necessary. This is a major drawback for industrial acceptance of the design method. To solve these two problems, a *single-rail* (bundled data) mapping from handshake circuits to asynchronous netlists was designed and implemented [29, 30]. The single-rail bundled data implementation uses only one wire per bit in the data communication but makes more assumptions on timing than the double-rail translation does. For each combinatoric part of the circuit, a matching *delay path* in the control is added. However, the advantage is two-fold: the area overhead is reduced considerably, and a generic cell-library can be used for layout. The feasibility of the single-rail compiler was demonstrated with a re-implementation of the error corrector for the DCC-player [30, 45]. This IC shows a power advantage of a factor 6 at the cost of 20% area overhead, compared to a synchronous solution.

The communication between handshake components is based on the handshake (request-acknowledge) protocol and therefore an asynchronous implementation of handshake circuits is a natural mapping to an IC. Within these asynchronous implementations there is a lot of freedom, of which the choice between double-rail and single-rail implementation is an example. There has also been an experiment to use one wire instead of two to establish the request-acknowledge protocol between components; one then obtains a *single-track* implementation [42]. The potential merit of this approach is high performance. A disadvantage is that a dedicated cell-library is needed to implement single-track. The mapping from handshake circuits to netlists need not result in an asynchronous circuit: one can even think of a mapping to synchronous circuits.

1.2 VLSI-programming

This thesis is about *VLSI-programming*, which is the activity of specifying a VLSI-circuit in a programming language, for example Tangram. The Tangram tool-set makes it possible to reason about circuits at the level of the Tangram-language, without knowing much detail of the gate-level implementation. The compiler translates Tangram programs *transparently* and this is the key notion that makes reasoning about circuits at the level of Tangram possible. The tools at the handshake-circuit level provide feedback to investigate the design space in terms of area, execution time, energy dissipation, and testability.

To see how the compilation from Tangram to handshake circuits works, we consider the example of a 1-place buffer. In Tangram we can describe this buffer by the procedure

```
buffer: proc(a?T & b!T) .
        begin x : var T
        |     forever do a?x ; b!x od
        end
```

The *header* of this procedure declares two external *channels* (`a` and `b`) of a certain type `T`. Channel `a` is an input channel (denoted by the question mark) and `b` is an output channel (exclamation mark). The `begin-end` construct contains the body of the Tangram procedure. It first declares a variable `x`. After the bar ("|") we see the statements of the Tangram program, in this case an endless loop (`forever do ... od`). The body of this loop is the sequence of two statements: an input along `a` into `x`, followed by an output of `x` along channel `b`. The corresponding handshake circuit is shown in Figure 1.2.

In this figure we see the various handshake components that correspond to Tangram constructs. Each handshake component has *passive* ports (denoted by open circles), *active* ports (denoted by a filled circles), or some combination of the two. In handshake circuits, a passive port is always connected to an active port, with the exception of one unique external passive port (denoted by "▷"). A handshake channel consists of a *request* wire, an *acknowledge* wire, and (possibly zero) *data* wires (Figure 1.3). Handshake channels without data wires (i.e. only consisting of a request and an acknowledge wire) are called *nonput channels*. Handshake channels that carry data are denoted by an arrow, indicating the direction of the data transport. Components communicate with each other using a *handshake protocol*. An example of such a protocol is shown in Figure 1.3.

In handshake circuits, the *active* port takes the initiative and raises the request. The passive port will eventually respond by raising the acknowledge wire. In the *2-*

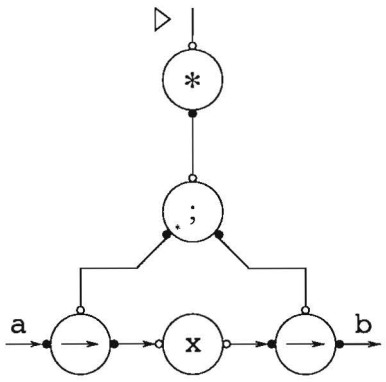


Figure 1.2: Handshake circuit for 1-place buffer.

phase handshake protocol the handshake is now complete, and the next handshake can only be initiated by the active port lowering the request after which the passive port will lower the acknowledge. In the *4-phase* handshake protocol, however, a handshake consists of both the up-going as well as the down-going of the request and the acknowledge wires. Many possible schemes of *data validity* with respect to the handshake-protocol are described by Peeters [30]. The 4-phase protocol allows for a wide variety of these schemes with many possible implementations of the handshake components.

For the operation of a handshake circuit we take the circuit of the 1-place buffer as an example (Figure 1.2). A handshake circuit is activated along its startup channel (denoted by the ▷), connected to its single passive external port. At the top we have the *repeater* (denoted by "*") connected to the startup channel: activated along its passive port it generates handshakes along its active port indefinitely. It corresponds to the Tangram `forever do ... od` statement. The *sequencer*, denoted by the ";", corresponds to the semicolon in the Tangram program. When activated along its passive port, it performs a complete handshake along its left and after that on its right active port. The first active port it performs a handshake on, is indicated by the "*". When the "*" is omitted, we assume that the sequencer always performs the first active handshake along its left handshake port in the diagram. When the two handshakes along the active ports have completed, the sequencer completes the handshake along its passive port.

The repeater and the sequencer are examples of handshake *control* components. The other components in Figure 1.2 are *datapath* components. Variable `x` corre-

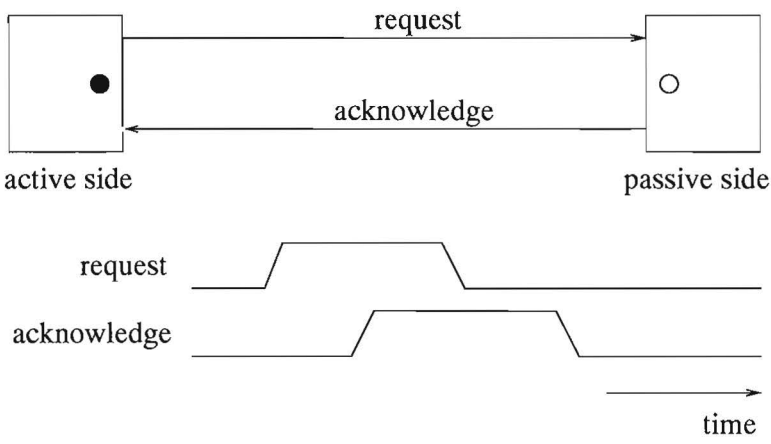


Figure 1.3: Handshake protocol with request and acknowledge wires. The active side is denoted by a filled circle and the passive side by an open circle. The active side always takes the initiative to start a handshake. The timing diagram shows the up-going and the down-going transitions on the wires.

sponds to the handshake variable component with two passive ports: one read port and one write port. Communication is established by the *transferrers* (denoted by "→"). The transferrer, once activated along its passive port, collects data from one active port and transports it to the other active port, in the direction of the arrow. The left transferrer in Figure 1.2 collects data from channel `a` and stores it in `x`; then (controlled by the sequencer) the data in `x` is copied to output channel `b` by the right transferrer.

Making a two-place buffer out of one-place buffers is straightforward. Suppose we have a Tangram procedure for a one-place buffer. A two-place buffer consists of two one-place buffers in parallel:

```
buffer(a,b) || buffer(b,c)
```

As the active port of the output transferrer of the first buffer is to be connected to the active port of the input transferrer of the second buffer, there has to be a special component in between, the *passivator*. The passivator in the datapath synchronizes two communications on its passive ports. The resulting handshake circuit is shown in Figure 1.4. (Remark: a 2-place buffer with synchronization between control components rather than between datapath components is described in [30]).

In this handshake circuit we see two instantiations of the one-place buffer circuit

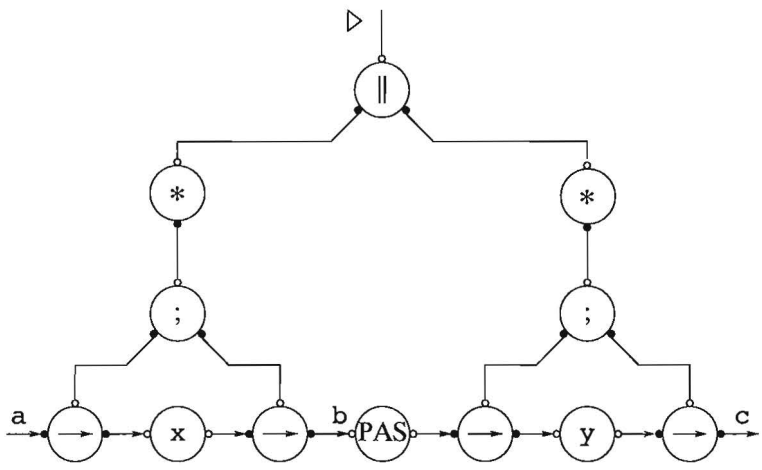


Figure 1.4: A handshake circuit for 2-place buffer.

of Figure 1.2. They are connected by the passivator in the datapath and the PAR-component in the control, denoted by "| |". This component corresponds to the Tangram "| |"-construct and takes care of parallel activity of pieces of circuitry. When activated along its passive port, it simultaneously initiates handshakes along its left and right active handshake channel. Once *both* of these handshakes have finished, the PAR-component completes the handshake along its passive port. In the case of the two-place buffer, the repeaters will never complete the handshakes on their passive ports and therefore the buffer circuits will continue their operation indefinitely.

The programming language Tangram is designed to describe hardware structures. A conventional programming language like C or Pascal lacks three constructs that are necessary to express a circuit design: *communication*, *parallelism*, and *sharing* of hardware, which will be described in Chapter 2. Tangram has these constructs available. On the other hand, concepts such as recursion, process creation, and dynamic data types are not necessary for hardware design, and therefore they are not present in Tangram.

1.3 Low-power

The total power dissipation of a CMOS circuit can be split into two components: the *static* and the *dynamic* part [54]. Static power dissipation is due to leakage current and other current drawn continuously from the power supply. Dynamic power dissipation is due to the switching transient current through resistive MOS-transistor channels, and the charging and discharging of load capacitances.

For a synchronous CMOS circuit in operation the dynamic power dissipation is dominant over the static dissipation and can be expressed by the formula [7, 54]

$$P = V_{\mathrm{dd}}^2 * f_{\mathrm{clk}} * C_L * \frac{1}{2}\alpha$$

where V_{dd} is the supply voltage of the circuit, f_{clk} the frequency of the clock, C_L the physical capacitance of the circuit, and α the activity factor of the circuit. To reduce the power dissipation of a circuit we have the following options.

First, we can reduce the *supply voltage* V_{dd}. Power scales down quadratically with the supply voltage, and therefore the supply voltage should be as low as possible for a low-power circuit. However, the *operating speed* of the circuit scales down as well, linearly with the supply voltage [12]. Introducing *parallelism* in the circuit's operation increases the operating speed, as a result of which the supply voltage can be lowered, resulting in lower power dissipation. On the other hand, introducing parallelism often leads to a larger area of the circuit. This technique is exploited, for example, on a superscalar asynchronous microprocessor, SCALP [8]. Superscalar processors have more than one execution unit for various types of instructions. This makes it possible to execute instructions in parallel.

Second, we have the *clock frequency* f_{clk}. Reducing the clock frequency reduces the power dissipation (though not the total energy that is dissipated for a given task!), but also implies lower performance in terms of execution time. On the other hand, it is often possible to reduce the clock frequency to zero for some parts of the circuit during operation. This technique, called *clock gating*, disconnects the clock from (a part of) the circuit that needs not to be active at a given point in time. There are two types of clock gating: *control dependent* and *data dependent*. Control dependent clock gating stops the clock at some part of the circuit when it is known that this part needs not be active. For example, in a microcontroller the timers can be switched off when they are not needed; in the circuit the clock is then switched off in the timer-block. Data dependent clock-gating occurs on a more fine-grained scale. For example, suppose that in a microprocessor an offset

has to be added to the program counter. When this offset is small, often only a few bits will change while a large part will not change. This can be exploited by not clocking the registers of the higher-order part of the addition when this is not necessary. The penalty is some extra control to detect whether this kind of clock-gating can be applied. Asynchronous circuits can be seen as an extreme in clock gating: only those registers are activated that need to be activated. The distributed control of asynchronous circuits makes this possible as we will see in Chapter 5.

Third, we can reduce the *capacitance* of the circuit. This can be done by resorting to a smaller feature size in the layout. Scaling down the transistor size and the wiring capacitance of a circuit reduces its total capacitance. When making a fair comparison between two circuits in terms of power dissipation it is important to compare them implemented in the same technology.

Fourth, and most relevant to VLSI-programming, we have the *activity factor* α. Power in CMOS circuits is only dissipated when there is switching *activity*, expressed by α. Minimizing activity lowers the power dissipation. The design used throughout this thesis, the 80C51 microcontroller, shows that there can be plenty of opportunity to minimize the activity of the circuit at the architecture level. The synchronous implementation of this microcontroller shows many *redundant actions* that can be filtered out due to the asynchronous distributed control.

Many of the observations above applied to energy-efficient microprocessor design can be found in literature [7, 12, 13]. An example of an asynchronous microprocessor family designed for low-power is the Amulet series that implement the ARM instruction set [27, 11].

VLSI-programming is about specifying the *architecture* of a circuit. Tangram and the Tangram-tools form a framework in which a *design space exploration* can be performed. The next section describes the parameters that are of importance in the design space, and motivates the choices for metrics for these parameters.

1.4 Metrics

There are four parameters that are important for the design space: area, operating speed, energy dissipation, and testability of a circuit.

The *area* of an IC can be measured in number of transistors, number of standard cells in the layout, number of gate-equivalents and die size. All depend on the implementation medium: does one map to a cell library or does one opt for full custom layout? Generally, full custom layout will result in less area, for it can

be optimized for the application that has to be implemented. Cell libraries allow for a quicker and cheaper design trajectory, because each design makes of the same standard-cell library. To compare one IC to another one should compare them when implemented in the same technology, using the same layout style. We choose to express the area of an IC in the number of transistors, for that is a precise measure when using a fixed style of layout.

The *operating speed* of an IC can be expressed in various ways. For microprocessors and microcontrollers with a fixed instruction set, one often chooses for *Millions of Instructions Per Second* (MIPS). This measure is good for comparing implementations of the *same* instruction set in the *same* technology. Some instruction sets make it possible to express a program in fewer instructions than other instruction sets. On the other hand, some instruction sets contain more powerful instructions than others. This makes it difficult to compare the MIPS metric for one microprocessor to another with a different instruction set.

The *energy dissipation* of an IC is expressed in Joules dissipated for a specific task. In the case of a microprocessor we can take the instruction set again and use the measure *Joules per instruction*. With the present technologies available this is in the order of a few nJ per instruction. Again this is a cumbersome metric when comparing different instruction sets. In literature one often encounters the equivalent measure MIPS/Watt as comparing metric. For the same reasons as above this is a poor measure for straightforward comparison of different microprocessor architectures. In this thesis, several implementations of the 80C51 microcontroller are described. When comparing them, we compare the same instruction set implemented differently but in the same technology, assuming comparable conditions (supply voltage, temperature, etc). Therefore we use the metric energy per instruction for comparison.

For operating speed in combination with energy dissipation one can make another observation. When one design is better in both aspects than another, we can consider both aspects separately. However, when there is a trade-off between the two, the situation is different. Suppose we have a design A and a design B that implements the same function. B dissipates half of A's energy but is twice as slow. Which design is the better (not considering the area of A and B for the moment)? When we would reduce the supply voltage for A with a factor of 2, the operating speed would scale down with a factor of 2. However the energy dissipation would, since it scales quadratically with the supply voltage, scale down with a factor of 4! Here we see clearly that A is the better design.

To have a combined measure for speed and energy, independent of the supply voltage, one has to use the *energy-delay-delay product* ET^2. For example, for the

above mentioned designs A and B, ET^2 is constant for any supply voltage that the circuit can operate in. When taking $ET^2 = m$ for design A, we would have $ET^2 = 2m$ for B, immediately demonstrating that A is the better design.

Last but not least, the *testability* of a circuit is an aspect that should be considered when designing a circuit. When a circuit has been fabricated it has to be tested for fabrication faults. The complexity of a circuit demands a thorough thinking of how to design a proper test for it. This testing issue is often considered only after the design, which makes it more difficult to create a proper test for it. Investigating the testability can have the beneficial effect of removing *redundancy* from the design. Redundant hardware is often not covered by a test, and can therefore be removed to reduce the area and improve the test coverage. This thesis will not deal with testability, though we will come back to it in Appendix A.

1.5 Challenges

At the beginning of this project, much attention had been paid to Tangram and the compiler, with the single-rail backend as one of the results. The feasibility of the approach was demonstrated by the single-rail asynchronous implementation of the DCC error-decoder.

Also the aspect of VLSI-programming had been paid attention to, in the context of the Reed-Solomon decoder for the DCC-player [19]. Other aspects of VLSI-programming for low-power are explained in [46]. As the single-rail compiler produces area-competitive circuits implemented in standard-cell layout it is interesting to research other areas of application of asynchronous low-power circuits.

The transparent compilation of Tangram into silicon enables the designer to reason about the circuit at the level of the Tangram-language. In other words, it should be possible to describe some *techniques* for VLSI-programming. As low-power is believed to be an important benefit of asynchronous circuits, these techniques for VLSI-programming should be tailored to obtain low-power circuits.

To demonstrate the rules for VLSI-programming a vehicle of industrial relevance was selected: the 80C51 microcontroller. Its architecture and implementation are reasonably old: the instruction set architecture was defined by Intel in 1980. Many derivatives have been implemented and for Philips it is a widely used architecture [4]. Some synchronous implementations are already tailored for low-power to make them suitable for applications where low power consumption is important.

The 80C51 instruction set shows a lot of irregularity in its many addressing modes

and non-uniform register structure. This makes it not straightforward to choose the better solutions in the design spectrum, and therefore a *design space exploration* is interesting. The Tangram VLSI-programming approach makes such an exploration possible, benefiting from the quick design cycle.

The 80C51 microcontroller is widely used in many products, because of its flexibility as programmable device and because it is cheap to produce. It has a general programmable CPU and therefore its application area is large. For the Tangram project it is important to demonstrate the feasibility and the advantages of the VLSI-programming approach on such vehicles of industrial importance.

1.6 Overview and contributions

This thesis is organized as follows.

Chapter 2 introduces VLSI-programming in more depth, and uses a small processor, the Move Machine, as an example. Several optimizations, of which *sharing* of hardware is important, are introduced and a small exploration of the design space is performed.

Chapter 3 describes the synchronous 80C51 microcontroller. It is an 8-bit microcontroller and can be considered a CISC (Complex Instruction Set Computer). The synchronous implementation is analyzed and six observations concerning the energy dissipation are made.

Chapter 4 outlines an asynchronous implementation of the 80C51. This microcontroller can be divided into three parts: a handshake-CPU, the peripherals and a Synchronizer-block. This block takes care of synchronizing the asynchronous IC in a synchronous environment. External memory access, where there is a timing protocol between the IC and its environment, serves as an example.

Chapter 5 describes the design space for the 80C51 CPU, the part that fetches and executes instructions. Various alternative Tangram programs result in different datapaths and control structures for the CPU. Furthermore, some local optimizations are discussed.

Chapter 6 deals with the implementation of the 80C51 *peripherals*. These additional pieces of hardware give the 80C51 system its flexibility, and interface between the CPU and the environment of the microcontroller. The interface

between CPU and peripherals is to adhere to certain constraints. A UART (Universal Asynchronous Receiver and Transmitter) serves as case study at the end of this chapter.

Chapter 7 reviews the six low-power observations made at the end of chapter 3, and discusses some alternative design methods that save power. A low-power asynchronous implementation of the 80C51 was fabricated and in this chapter the manufactured IC is analyzed. It is compared with its synchronous counterpart as well with other low-power microprocessors and microcontrollers.

Chapter 8 reviews and concludes this thesis.

The aim of this thesis is to learn to reason about circuit properties at the level of a programming language, Tangram. The main contributions of the research described in this thesis are:

- Identification of techniques for VLSI-programming;
- Design space exploration for the design of an asynchronous 80C51 microcontroller architecture;
- Demonstration of the feasibility of the Tangram VLSI-programming approach to this architecture;
- A low-power asynchronous implementation of the 80C51 microcontroller, showing a power benefit at the cost of some overhead in area.

The development of the asynchronous low-power 80C51 microcontroller was done in cooperation with Philips Semiconductors. The design of the CPU (Chapter 5), the interfaces to the environment and to the peripherals (Chapters 4 and 6), and the design of the UART as described in this thesis, were carried out by the author. The design of the other peripherals, as well as the final layout and the measurements of the test chip were done at Philips Semiconductors and Philips Research. The pictures of the photon emission, which appear in Chapter 7, were produced and interpreted at Philips Research.

Chapter 2

VLSI-Programming

Conventional programming is the activity of writing a program in some language and compiling it to a list of instructions that can be executed on a processor. Important aspects of the resulting programs are the execution time, the usage of memory, and the size of the generated code. In most cases, the program is executed in a sequential fashion, instruction by instruction. It is often possible to classify various programs according to a parameter, for example speed or memory usage.

VLSI-programming involves the activity of specifying a VLSI-circuit in a programming language, for example Tangram. Important aspects of VLSI-programs are the speed, the power dissipation, the size, and testability of the resulting circuit. Also fine-grained timing is an aspect of VLSI-circuits. To express the functionality of a circuit it is important that the language has constructs for *parallelism* and *communication.*

It is often the case that a VLSI-program for a specific function results in a smaller but, for example, less energy-efficient or slower circuit than another VLSI-program. Therefore it is difficult to use one parameter to compare VLSI-programs; often a *trade-off* between various VLSI-programs can be made. Therefore it is hard for a compiler to decide which translation is the best for a given program; the designer has to make these decisions. Thus it is important that the compiler translates *transparently*; only then the designer can reason about the circuit at the level of the programming language, and rewrite the program in such a way that the requirements are met.

The Tangram compiler translates a VLSI-program into a netlist of a circuit transparently. Tools that perform simulations at the level of the handshake circuit give feedback with reference to the structure of the Tangram program. Handshake cir-

cuits describe the circuit at a level of abstraction that is low enough to obtain accurate data about the resulting circuit; on the other hand this level is high enough to allow for fast simulation.

In the previous chapter we have identified four aspects that are of importance for VLSI-programs: area, operating speed, energy dissipation, and testability. Normally, when started from an initial design, one writes other VLSI-programs that are better with respect to all of these aspects, but when pushing one aspect to the extreme, another might get worse. It is then possible to make a trade-off, for example to choose for lower power dissipation at the cost of a larger area. In this chapter some of these aspects are demonstrated on a small example, the *Move Machine*, which is a small and easy-to-understand processor.

The goal of this chapter is to understand some basics of VLSI-programming. To this end, we have to understand the translation scheme from Tangram to handshake circuits. Therefore we first take a closer look at handshake circuits.

2.1 Handshake circuits

A *handshake circuit* is a connected graph that consists of so-called *handshake components* [41]. A classification of these handshake components is described by Peeters in his thesis [30]. In this classification he only considers handshake components with handshake interfaces, both internally and externally.

In general, a handshake circuit has the form as shown in Figure 2.1. First there are the *control components*; these components only have control (i.e. nonput) handshake channels. Second, *data components* can be distinguished; these only have data handshake channels. In between there are the *interface components*; these can communicate with other components using both nonput and data channels. The data components can furthermore be divided into three categories: the *pull*, the *push*, and the *passive* components.

An example of a passive nonput channel ("pn" in Figure 2.1) is the startup channel of the handshake circuit. After initialization of the circuit, a handshake initiated on this channel will start the circuit's operation.

Passive components are handshake components that only have passive handshake ports. Examples are the *variable* and the *passivator* as shown in Figure 2.2. The variable is used for storing values of some type. In the single-rail implementation of handshake circuits, it has one write port and and possibly more than one read port [30]. When the variable is implemented as a latch, then it is not possible to

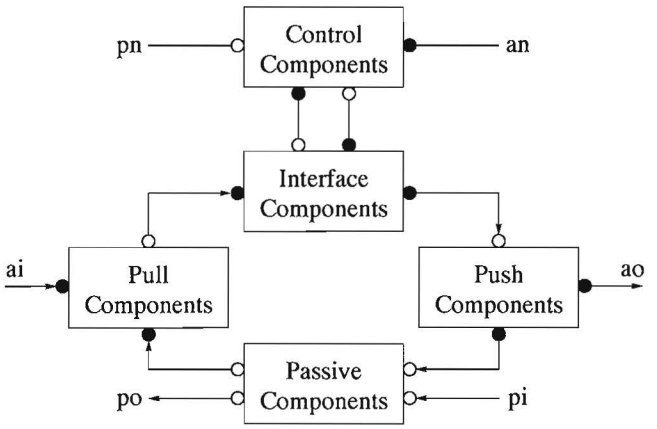


Figure 2.1: General structure of a handshake circuits [30].

read and write the variable simultaneously. However, multiple simultaneous reads are possible. The passivator that has two passive ports; it synchronizes handshakes along these ports.

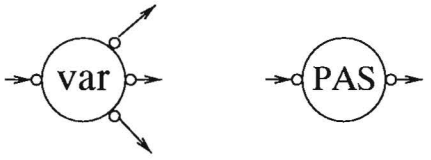


Figure 2.2: Handshake passive components: the variable and the passivator. The variable in this picture has three read ports and one write port.

Pull components collect, modify, and output data upon request. An example of such a pull component is a binary operator, such as addition. This operator is shown in Figure 2.3. Once activated along its passive handshake port, it collects two operands along the active channels, performs the addition, and outputs the result along the passive channel.

Push components take care of the communication of data to the environment of the handshake circuit, either via an active output, or indirectly though a passivator via a passive output. The multiplexer, which merges streams of data onto one

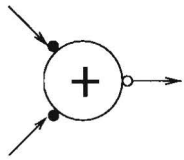


Figure 2.3: Handshake pull-component: binary addition.

channel, is an example of a push component (Figure 2.4). Multiplexers have two or more passive inputs and one active output. Handshakes on the input channels are required to be mutually exclusive. Upon activation along one of the inputs, the multiplexer performs a handshake along its active output, after which the handshake on the passive input is completed. Multiplexers that carry no data are called *mixers*. Mixers also appear in the control of a handshake circuit.

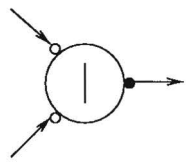


Figure 2.4: Handshake push-component: the multiplexer.

The *interface* components as shown in Figure 2.5 have both data channels and nonput channels as handshake interface to other components. Examples are the transferrer, the `case`-component, and the `do`-component. The latter two form the interface between *control* and *datapath* of a handshake circuit. The binary `case`-component takes a boolean input on its passive port and, depending on the value of this input, decides which active port to activate. The `do`-component also takes a boolean input on its passive port and activates its active port until the boolean input is false. The transferrer collects data from one active port and transfers it to the other active port. The direction of the transport of data is indicated by the arrow.

Control components are used to steer the interface components. They decide in what order the various other components are activated. Examples are the *sequencer*, the `PAR`-component, and the *repeater* (Figure 2.6) . The sequencer, once activated along its passive channel, completes handshakes first along its left (in-

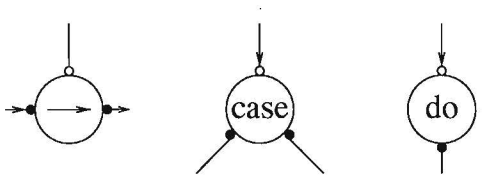


Figure 2.5: Handshake interface components: transferrer, `case`, and `do`.

dicated by "*") and then along its right active channel. It then completes the handshake along its passive channel. It thus sequences the handshakes along its active channels. When the "*" is omitted in the symbol of the sequencer we assume that the first handshake will take place along the left handshake channel. The PAR-component performs handshakes along its active channels simultaneously, and completes the handshake along its passive channel only when both handshakes along the active channels have finished. The repeater, once activated along its passive port, continues generating handshakes on its active port forever.

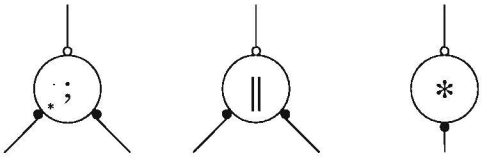


Figure 2.6: Handshake control components: the sequencer, the PAR-component, and the repeater.

In microprocessor design there is the distinction between the *control* and the *datapath*. The datapath is the part of the circuit that contains all registers and combinatorics to do calculations on the values stored in these registers. Furthermore, all communication paths including multiplexers and demultiplexers are part of the datapath. The control literally controls the datapath; it takes care of the proper steering of all elements in the datapath.

In what follows we adopt this view for handshake circuits. In order to make the connection with the "microprocessor" control and datapath more explicit, we make a slightly different, somewhat more abstract classification than the one above.

The *datapath* of a handshake circuit contains the following components:

- all passive components (the variables, corresponding to the registers, the passivators used for synchronization, but also register files and memories);
- the pull components, such as all arithmetic operations;
- the push components, such as the data multiplexers;
- the transferrers.

The *control* of a handshake circuit consists of

- the control components (the sequencer, the PAR-component, and the repeater);
- all interface components except the transferrer, such as the do and the case-component.

The only handshake components whose functionality is data-dependent, are the do and the case-component, which reside in the control. The processing of this data takes place in the datapath, and therefore there has to be some communication between datapath and control.

The channels that are the interface to the datapath are

- the active (ai) and passive (pi) inputs from the environment;
- the active (ao) and passive (po) outputs to the environment;
- the activation (nonput) channels from all transferrers in the datapath;
- the inputs for the conditional control components (i.e. the case and the do-components).

Using this classification we obtain the general view of a handshake circuit as shown in Figure 2.7.

Now that we have introduced the handshake components, it is time to look at some design issues. Area, speed, power dissipation, and testability are important parameters for a circuit. For the first three aspects, the next sections explain how we can reason about them at the level of the VLSI-programming language and on the level of handshake circuits. To this end, we use a small processor, the *Move Machine*, as running example.

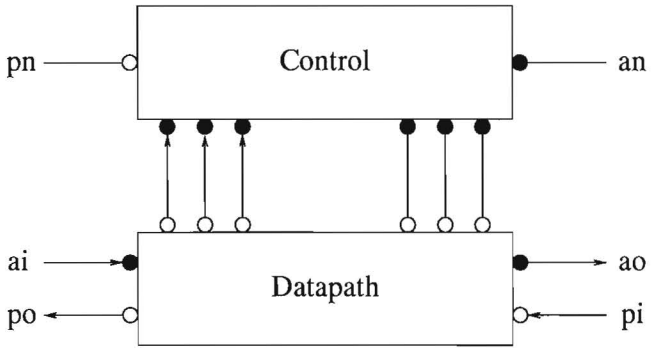


Figure 2.7: General structure of microprocessor handshake circuits.

2.2 The Move Machine

The Move Machine is a small processor with a limited instruction set. It was proposed by Sutherland in the 1970's [5]. Sutherland observed that in a computer system the main processor spends most of its time moving data back and forth in memory, while not doing any "useful" tasks it was designed for. The task of the Move Machine is to assist the main processor by moving blocks of data around in memory.

Birtwistle et.al. reported on a specification and an implementation of the Move Machine [5]. An extension of its instruction set can be found in [50]. This is the instruction set that is going to be used as running example throughout this chapter. This work was also reported on in [51].

A processor is made to execute instructions from a given instruction set. The design of the instruction set determines the flexibility and purpose of the processor. Since the Move Machine is meant for moving blocks of data, its instruction set is tailored for that task.

The Move Machine is built around a register file of 16 eight-bit wide registers, and can access an external memory M. Both the program and the data are stored in the same memory. The instruction set contains ten instructions and is shown in Table 2.1. The operands for the various instructions are indices in the register file (r_1, r_2 and r_3 in Table 2.1).

We distinguish several classes of instructions in this set:

Code	Acronym	Arguments	Action	Description
0	LOD	r_1, r_2	$r_2 := M[r_1]$	load
1	STO	r_1, r_2	$M[r_1] := r_2$	store
2	LDI	r_1, r_2, r_3	$r_2 := M[r_1 + r_3]$	load with offset
3	STI	r_1, r_2, r_3	$M[r_1 + r_3] := r_2$	store with offset
4	MOV	r_1, r_2	$r_1 := r_2$	move
5	SCC	r_1, r_2	$cc := (r_1 = r_2)$	set condition code
6	INC	r_1	$r_1 := r_1 + 1$	increment
7	ADD	r_1, r_2	$r_1 := r_1 + r_2$	addition
8	JCC	r_1	if cc then $pc := r_1$ fi	jump condition code
9	NOP			no operation

Table 2.1: Move Machine instruction set.

- instructions for loading and storing values in memory (LOD, LDI, STO, and STI);
- instructions for manipulating values in the register file (MOV, INC, and ADD);
- instructions to control the program flow: SCC (Set Condition Code) and JCC (Jump on Condition Code);
- A miscellaneous instruction: NOP (No OPeration).

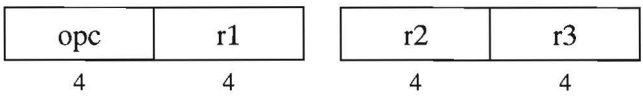


Figure 2.8: Instruction encoding.

The length of the encoding of instructions (Figure 2.8) is fixed: each instruction is encoded using two bytes. The first byte contains the opcode (4-bits) and register file index r_1 (also 4 bits); the second byte contains indices r_2 and r_3. Since not all instructions use all these register-file indices, there is some redundancy in this encoding, which (as we will see later on) can be used to our advantage in the implementation of the Move Machine.

The Move Machine is to operate with an environment that contains the memory. This memory contains both the program to execute and the data to operate on. Schematically the interface between the Move Machine and its environment is shown in Figure 2.9.

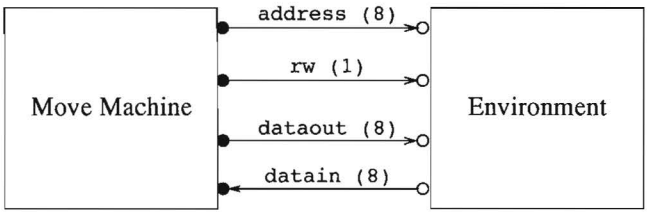


Figure 2.9: The Move Machine and its environment. The numbers denote the widths of the channels, in terms of number of bits.

The Move Machine can read and write data in memory by adhering to the following protocol:

- First, the Move Machine sends an address to the environment (along `address`), together with a signal along channel `rw` indicating whether it wants to read data from, or write data to the memory;
- then, the Move Machine either
  - reads data from memory (along `datain`),
  - or writes data to memory (along `dataout`).

2.3 An initial VLSI-program

With the instruction set and the interface to the outside world as starting points, the main structure of a Tangram program for the Move Machine can be designed. This main part is given in Figure 2.10.

The Tangram program starts with the definitions of types. The interface to the outside world is described in the header of the Tangram program, in which the input ("`?`") and output ("`!`") channels and their types are specified. The header clearly follows the structure as shown in Figure 2.9.

```
  int8 = type[0..255]
& int1 = type[0..1]
| ( address!int8
  & rw!int1
  & dataout!int8
  & datain?int8
  ) .
  begin /* Declarations of variables and procedures */
  |     main()
  end
```

Figure 2.10: Structure of the Tangram program for the Move Machine.

Procedure `main()` contains the sequence of statements that constitute the VLSI-program: it describes the implementation of the Move Machine. The processor executes an endless loop in which it fetches an instruction and then executes the appropriate statements. For reasons of simplicity we choose to fetch and execute the instructions sequentially, which can be expressed by the semicolon (";") in Tangram. Thus we obtain the following program for `main()`:

```
main : proc(). forever
               do FetchOp()
               ;  Execute()
               od
```

Fetching an instruction involves sending an address to memory, together with a `read`-signal (which is coded as a boolean value); then the Move Machine collects the opcode of the corresponding instruction and increments the program counter `pc`. Each Move Machine instruction is coded in two bytes, so we have to do two memory accesses per instruction:

```
FetchOp : proc(). dataout!pc          || rw!read
                ; datain?<<opc,r1>> || pc:=pc+1
                ; dataout!pc          || rw!read
                ; datain?<<r2,r3>>  || pc:=pc+1
```

Here we see the use of parallelism between statements by the Tangram "||"-construct. Statements put in parallel are executed simultaneously.

Having received the opcode of the instruction, procedure `Execute()` executes the necessary statements that implement the instruction. The opcode is simply a number between 0 and 9 (Table 2.1); decoding can be done using the Tangram `case`-statement, which selects the right instruction:

```
Execute : proc() .
          case opc
          is 0 then /* LOD */ lod()
          or 1 then /* STO */ sto()
          or 2 then /* LDI */ ldi()
          ...
          or 9 then /* NOP */ nop()
          si
```

The implementation of the various instructions is implemented in procedures `lod()`, `sto()`, ..., `nop()`. In fact, they implement the actions as given in Table 2.1. For example, instruction `inc()` can be implemented by

```
inc : proc() . RF[r1]:=RF[r1]+1
```

Similarly, we can implement the ADD instruction by the procedure

```
add : proc() . RF[r1]:=RF[r1] + RF[r2]
```

Some of the instructions in the Move Machine access the memory and therefore have to communicate with the environment. Note that there is only one memory that contains both program code and data. The protocol for access is the same for both for the environment does not make any difference between program and data memory. Take instruction LOD as an example. This instruction collects a value from memory and stores it in the register file. Tangram procedure `lod()` reads

```
lod : proc() . address!RF[r1] || rw!read
             ; datain?RF[r2]
```

First the address of the data sent along `address` together with the read-signal. Next, following the data-access protocol, the data is collected from memory and stored in the register file. Instruction STO stores a data value in memory and is implemented by procedure `sto()`:

```
sto : proc() . address!RF[r1] || rw!write
             ; dataout!RF[r2]
```

The instructions that control the flow of the program are also straightforwardly encoded in Tangram. The SCC instruction, for example, sets a condition code:

```
scc : proc() . cc:=(RF[r1]=RF[r2])
```

Instruction JCC tests the condition code to see whether a jump has to be made:

```
jcc : proc() . if cc then pc:=RF[r1] fi
```

Finally, the NOP instruction does nothing useful:

```
nop : proc() . skip
```

As we have seen, all instructions take a few steps to be executed. Generally, an

instruction is implemented using a sequence of statements

```
S1 ; S2 ; ... ; Sn
```

where the statements `Si` are assignments, communications, or conditional statements.

2.4 Implementation issues

Some implementation issues of a silicon compiler that translates transparently, can be reflected in the programming language. The Tangram compiler, for example, uses *latches* to implement Tangram variables. Because latches cannot be written and read simultaneously, the compiler demands that the statements that are in parallel do not read and write the same variable (though more than one simultaneous read-action is permitted). A special case is found in in the `FetchOp()` procedure of the Move Machine, where we encounter the assignment `pc:=pc+1`. For these so-called *auto-assignments* the compiler will introduce an auxiliary variable in the handshake circuit. Making this variable explicit in Tangram results in the same handshake circuit:

```
pcaux:=pc+1 ; pc:=pcaux
```

The same goes for the implementation of the register file. When this file is implemented using latches, the auto-assignments are resolved by doing the assignments in two steps, as is shown in the procedure of the `INC` instruction:

```
inc : proc() . x:=RF[r1]+1 ; RF[r1]:=x
```

The programmer can make the auxiliary variable explicit and share it amongst other (auto-)assignments, as we will see later on.

It depends on the number of read ports and write ports of the register file whether the addition in the `ADD` instruction can be implemented as a single statement. When, for example, the register file has only one read port and one write port, auxiliary variables `x` and `y` have to be used and the addition is done in three steps:

```
add : proc() . x:=RF[r1]
             ; y:=RF[r2]
             ; RF[r1]:=x+y
```

The same arguments go for the accesses to the register file in instruction `SCC`.

In the next sections we will demonstrate the impact of altering the Tangram program on the characteristics of the resulting circuit. We take three parameters as identified in the previous chapter, viz. area, speed and energy dissipation, as clas-

sification. Testability is left outside the scope of this chapter.

2.5 Area

In general, a smaller chip is cheaper to produce. Therefore, reducing the area of a chip is very important. One method to reduce area is to reuse hardware for similar tasks when possible. This is what we refer to as *sharing* of hardware.

In Tangram we can express sharing of hardware by sharing of statements in procedures. Each time the piece of hardware is to be used, the corresponding procedure is invoked. We can distinguish two kinds of sharing: sharing in the *datapath* and sharing in the *control*. First the principle behind both is explained and then an example is given that combines both.

Sharing in the datapath can be applied when there are two occurrences of an assignment `x:=y` in the Tangram program text:

```
    ...
; x:=y
    ...
; x:=y
    ...
```

In the handshake circuit this will introduce two paths from variable `y` to `x`, and a multiplexer on the write port of `x`. This multiplexer has the width of variable `x`. The resulting datapath is shown on top in Figure 2.11. To create just one path from `y` to `x` in the circuit, we introduce a procedure

```
xy : proc() . x:=y
```

and replace all occurrences of the assignment `x:=y` in the main Tangram text by invocations of this procedure:

```
    ...
; xy()
    ...
; xy()
    ...
```

This will result in the handshake circuit as shown in the lower half of Figure 2.11. In the datapath there is now only one path from `y` to `x`, saving one transferrer and the multiplexer. However, an extra mixer in the control is added to take care of the two invocations of procedure `xy()`. The mixer is a multiplexer that carries no data, and is therefore cheaper to implement than the data-multiplexer. In effect we have lifted the expensive multiplexer in the datapath to a cheaper mixer in the control.

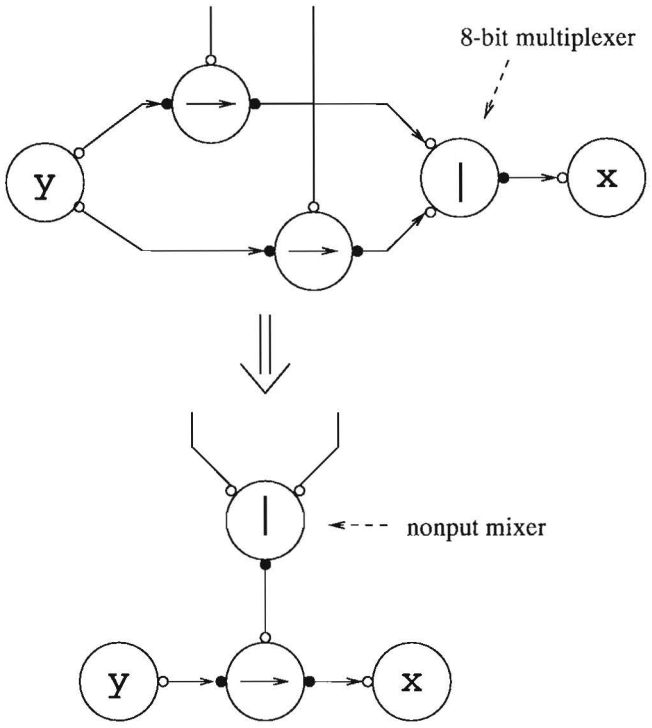


Figure 2.11: Sharing in the datapath.

Sharing of common statements can also be applied to communication statements, which occur often in our Move Machine program. For example, in the `lod()` and `sto()` procedures we have the common statement

```
address!RF[r1]
```

that can be shared in a procedure. It is straightforward to find the other common statements in the datapath. Generally speaking, sharing in the datapath results in smaller circuits.

Sharing of hardware is not restricted to the datapath; also *control structures* can be shared. Semicolons in the program text (i.e. sequencers in the handshake circuit) are an example. Suppose we have a Tangram program that contains the fragment

```
...
S1 ; S2
...
S1 ; S2
```

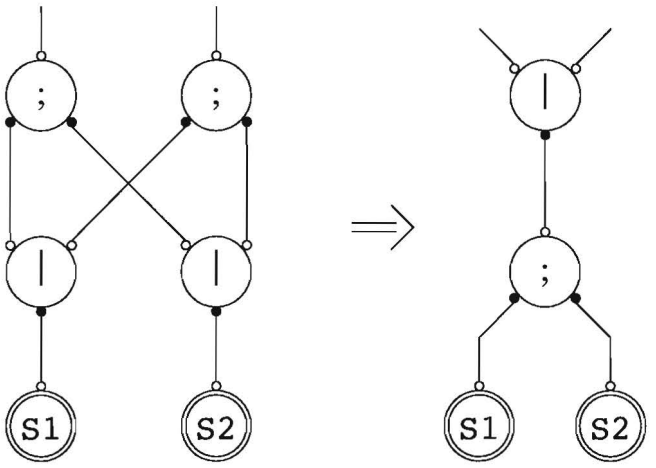


Figure 2.12: Sharing in the control.

where `S1` and `S2` are Tangram statements. The corresponding handshake circuit is shown at the left of Figure 2.12. By introducing a procedure

```
S1S2 : proc() . S1 ; S2
```

and replacing of the above fragment by

```
...
S1S2()
...
S1S2()
```

the handshake circuit on the right of Figure 2.12 is obtained. This circuit contains one sequencer instead of two, and also saves one mixer.

Sharing in the control is not restricted to reducing the number of semicolons in the Tangram program. In our Move Machine, for example, we see in procedure `FetchOp()` two occurrences of

```
dataout!pc || rw!read .
```

that can easily be shared saving one `PAR` component and one mixer.

An interesting case where sharing of datapath and control structures go together is shown by the two occurrences of the auto-assignment

```
pc:=pc+1 .
```

As auto-assignments cannot be implemented straightforwardly in the Tangram compiler, it introduces for each of these two statements a separate auxiliary variable. This results in the same handshake circuit as when we would have written the Tangram fragment

```
pc2:=pc+1 ; pc:=pc2
...
pc3:=pc+1 ; pc:=pc3
```

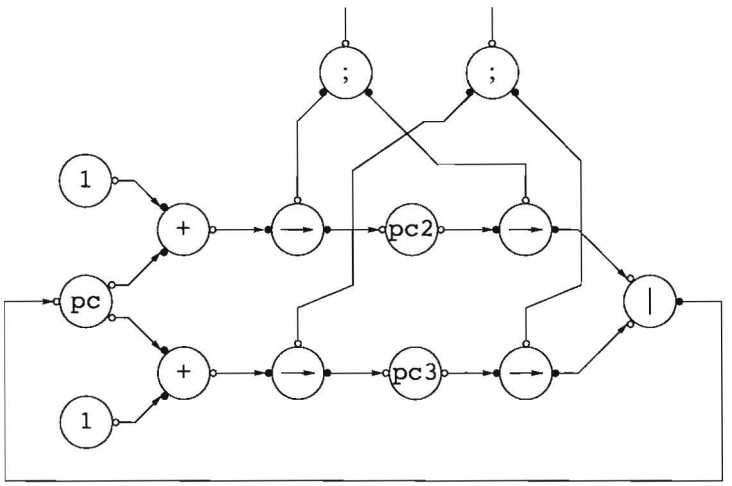


Figure 2.13: Two increments on the same variable pc.

This handshake circuit is shown in Figure 2.13. By explicitly introducing an auxiliary variable in the Tangram text we can share it among these assignments. We can then write

```
pcaux:=pc+1 ; pc:=pcaux
...
pcaux:=pc+1 ; pc:=pcaux
```

and also share the semicolons between the now identical statements. The resulting handshake circuit is shown in Figure 2.14. It saves one variable, one adder, one constant, one multiplexer, two transferrers, and one sequencer, at the cost of an extra mixer in the control. We would obtain the same handshake circuit when first sharing the auto-assignments `pc:=pc+1` into one procedure.

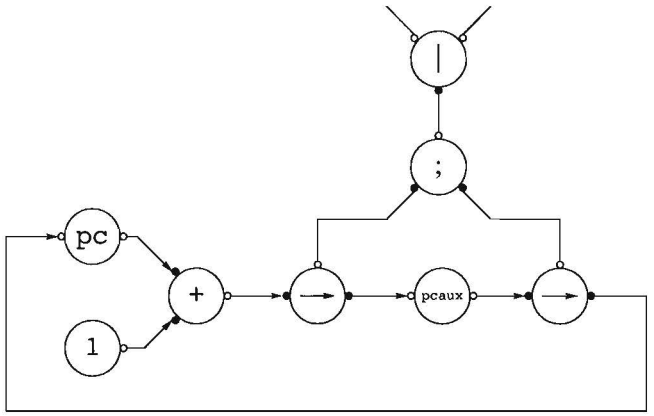


Figure 2.14: Making the auxiliary variable explicit and sharing hardware.

2.6 Energy

In CMOS circuits no energy is dissipated when there is no switching. For low power consumption it is therefore important to keep the switching activity of the circuit to a minimum. It is important to be aware of the switching activity of a circuit at the architecture level (i.e. at the Tangram level) to exploit the potential advantages of a distributed and asynchronous control. The *control* of a circuit determines the activity of the *datapath* by steering the appropriate transferrers in the handshake circuit. Thus we have to investigate where actions take place in the datapath and where activity can be reduced.

The Move Machine encodes instructions in two bytes. Though for most instructions these two bytes are necessary, for some only one byte will suffice. For example, the second byte in the code for the `INC` instruction is superfluous. It makes the control simple if we fetch two bytes in all cases. However, for the `INC` instruction fetching the second byte is wasting energy; we can instead just increment the program counter. This will complicate the control a bit: after fetching the first byte we have to check whether we have to fetch a second byte or not. Here we can trade larger area for lower energy dissipation.

When we wish to minimize the area even further, we can try to reduce the number of variables. It is often possible to communicate values by using existing communication paths in the datapath. When introducing some extra variables, i.e. extra communication paths, it is often possible to reduce the number of communication

steps during the execution of an instruction. This saves energy, for there is less activity in the datapath.

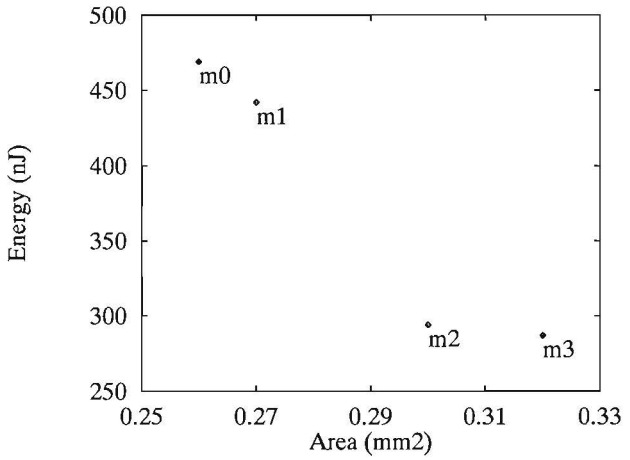


Figure 2.15: Handshake simulation: area vs energy.

For the Move Machine, these two aspects of reducing energy dissipation are illustrated using a small benchmark program. This program copies blocks of data in memory, and uses all instructions in the instruction set. Using the techniques as described in the previous section, we obtain Move Machine `m0` that reduces the area by keeping the control and datapath small. Figure 2.15 shows that `m0` occupies 0.26 mm^2 dissipates approximately 470 nJ to execute the benchmark program. Move Machine `m1` reduces the number of memory fetches for the instructions: it is slightly larger but also more energy efficient.

Taking `m0` as starting point, Move Machine `m2` has more variables and communication paths in its datapath. It is larger but significantly more energy-efficient than `m0`. Reducing the number of memory fetches in `m2` results in `m3`, which is larger and only marginally more energy-efficient.

Pictures like in Figure 2.15 visualize the trade-off between area and energy dissipation. The same can be done for area versus execution time, as illustrated in the next section.

2.7 Execution time

To increase the operating speed of a circuit there are two things that can can done at the Tangram level: reducing the number of actions and introducing parallelism.

First, reducing the number of steps that the chip has to do to perform its task, makes the chip faster. For the Move Machine, the actions taken to lower the energy dissipation, also lower the number of steps taken. Therefore they are also good for speed. Figure 2.16 takes the same designs of the Move Machine as in the previous section and shows a similar picture as for the energy dissipation. One can now trade area for operating speed.

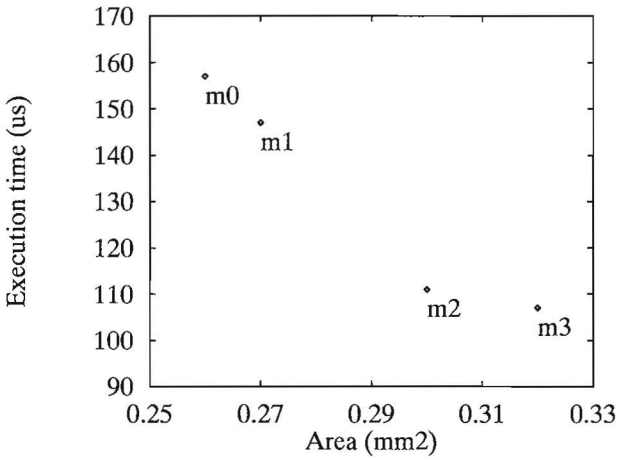


Figure 2.16: Handshake simulation: area vs execution time.

Second, there is parallelism. Executing actions in parallel clearly reduces the time necessary to complete a given task. In the Move Machine, we already did so in the `FetchOp()` instruction by putting the collection of a byte from memory and incrementing the program counter in parallel. When one wants to introduce more processes running in parallel one often has to introduce extra registers (variables) and some overhead in the control of the circuit. *Pipelining* is a typical example of introducing parallelism in instruction execution. It overlaps the execution of several consecutive instructions. By doing so, one has to transport information from one stage to another; this information would be global to the design in a non-pipelined version. For example, the program counter and instruction opcode have to be transported along the stages. This costs extra registers and communication

paths, and thus area and energy.

Increasing the operating speed of a chip is also interesting for energy dissipation. When a chip is faster than demanded by the specification, one can lower the supply voltage. An asynchronous circuit runs freely and as fast as possible (i.e. not hampered by a clock): therefore it will have a certain speed at a given supply voltage. By reducing the supply voltage, the speed will go down as well, but the energy dissipation will even go down quadratically with respect to the supply voltage.

2.8 Review

The transparent compilation from Tangram into handshake circuits, and then into netlists, implies that what is expressed in Tangram, is implemented exactly in the circuit. When there are two adders in the Tangram program, for example, also two adders will be implemented in the circuit. For small examples, like the sharing of hardware in Figures 2.13 and 2.14, the compiler could be constructed in such a way that the *parallel transfer paths* (the two paths from pc to itself) are automatically shared into one path. There are also transformations that result in a smaller but slower circuit, or in a circuit that is larger but consumes less energy. The compiler cannot choose the better design; this is a task of the designer. Therefore, it is important that the designer can reason about the resulting circuit at the level of the programming language. This property can be used to the full extent when the compiler translates a VLSI-program completely transparently into a circuit.

Sharing of small and local Tangram constructs can be done in any design. Some optimizations are good for area, speed, and energy dissipation; obviously, these optimizations result in a better circuit in any respect and should therefore be applied. The interesting parts of the design space are those areas where a trade-off between area, speed, and energy can be made. The Move Machines in Figures 2.15 and 2.16 show this part of the design space.

All aspects of optimizing VLSI-programs discussed in this chapter have a *local* character; they involve small Tangram constructs in datapath and control. The *global* structure of the circuit is determined by the way the Tangram program is written. The remainder of this thesis deals with the design of datapaths and controls for a *sequential* processor architecture (i.e. without pipelining): the 80C51 microcontroller. The next chapter introduces this microcontroller and analyzes the power aspects of the synchronous architecture.

Chapter 3

The 80C51 Microcontroller

Microcontrollers are used in various products, like VCRs, television sets, and portable telephones, because of their flexibility as programmable devices. They are often based on somewhat older architectures, having the advantage that existing software and software development environments can be used. With the aid of some extra hardware, microcontrollers can be used in many applications where there has to be some central unit to keep control of the system. Their programmability and therefore their flexibility makes them often more popular than a dedicated hardware solution.

Microcontrollers are often used in hand-held and battery-powered applications like portable CD-players, mobile phones, and pagers. Power consumption of the chips in these products is often a considerable part of the total power dissipation of the product; reducing the power has the product survive longer on one battery charge. For some applications that involve radio transmission and reception (like pagers and mobile phones) the electro-magnetic emission of the ICs is also of importance, as it can interfere with the radio.

In this chapter we first characterize a microcontroller. Then we zoom in on one microcontroller architecture, the 80C51. Analyzing the synchronous architecture we observe what the characteristics of this implementation are and where the bulk of the power is dissipated. These observations are the basis for the next chapters where a low-power asynchronous version of the 80C51 and its design aspects are discussed.

3.1 Characterization

A microcontroller consists of a general-purpose programmable *CPU* (Central Processing Unit) with program memory and data memory, and a number of *peripherals* connecting the CPU to the environment. The CPU executes instructions from a fixed instruction set; therefore the CPU is fixed. The peripherals make the difference in the microcontroller world. By adding peripherals the system obtains the functionality that is needed for the environment where the microcontroller works in. Microcontrollers are often used in *embedded* applications, integrated with other blocks of hardware. These other blocks can be other processors (a digital signal processor, DSP, for example) or blocks of dedicated hardware.

A micro*processor* is a stand-alone chip that is often tuned for high-performance. Microprocessors are used in computer systems like PCs and workstations. In a system the microprocessor is visible to the user, and in principle, programmable by the user. Microprocessors do usually not contain large blocks of memory other than caches that help the processor to improve its performance. They communicate with the main memory and other chips in the system using, for example, a bus. A micro*controller* is often used in embedded systems with blocks of hardware such as timers and an interrupt controller, integrated onto one chip. An embedded program memory contains the program that is executed by the microcontroller. Microcontrollers are also available as stand-alone devices for manufacturers that do not have the facilities to produce their own ICs. These manufacturers can integrate the microcontroller into their (embedded) systems using their own software. Microcontrollers that are embedded in a system are not visible or programmable by the user. The use of microcontrollers is as widespread as the use of embedded systems: from a simple remote control unit for a television to the complicated ICs in a cellular phone.

In the world microcontroller market, the 8-bit microcontrollers take a substantial part: in 1995 worldwide more than 1200 million units of 8-bit microcontrollers per year were produced [1]. Predictions say that this amount will certainly not decrease over the next few years. Even 4-bit microcontrollers take a substantial part in the microcontroller world market. Some bicycle-computers, which keep track of average speed, time, and distance, contain a 4-bit microcontroller, for example. Microcontrollers can be made cheap to produce, when produced in quantity (some derivatives sell for less than US$1 per packaged device).

The 80C51 is one of the most widely produced 8-bit microcontroller in the world [22]. The 80C51 instruction set originates from Intel (1980). Philips uses the 80C51 for embedding in many of its products, and produces many derivatives of the standard

80C51. In 1995 a 16-bit extension extension (the 80C51 XA, eXtended Architecture) was introduced to comply with the market needs for 16-bit architectures [3].

For this thesis, we concentrate on the standard 8-bit 80C51 microcontroller. An analysis of the synchronous architecture shows where the bulk of the power is dissipated. In the next few chapters the design spectrum of the 80C51 architecture will be explored with a low-power 80C51 implementation as a goal.

3.2 Synchronous architecture

Most of the material of this section is based on the Philips 80C51 Data Handbook [4]. This book contains about 100 pages of general architecture description; the other 1250 pages (!) contain information about the numerous derivatives of the 80C51. We focus on the general architecture description.

3.2.1 80C51 system

The 80C51 microcontroller consists of several parts; the *CPU* with its memories, and various *peripheral* blocks. The CPU fetches, decodes, and executes instructions. The peripherals comprise blocks as timers and counters, the interrupt controller, and the port logic. An overview of the 80C51 "system" is shown in Figure 3.1.

The CPU and the peripherals run in parallel and communicate with each other where and when necessary. The 80C51 derivatives are all based on the same CPU-architecture but they differ in the sizes and the implementation of the memories (ROM, OTP, Flash, etc), and in the peripherals. A derivative can have more functionality implemented in its peripherals, for example an extra timer or an interrupt controller that can handle more interrupts. But it is also possible that a derivative has extra peripherals, like a UART (Universal Asynchronous Receiver and Transmitter). For this chapter we first zoom in on the 80C51 CPU and then we take a look at some peripherals.

3.2.2 80C51 CPU

The CPU is the part of the microcontroller that fetches, decodes, and executes instructions. Instruction memory and data memory are separated: the 80C51 is a *Harvard* architecture. The program memory (usually implemented as ROM) can

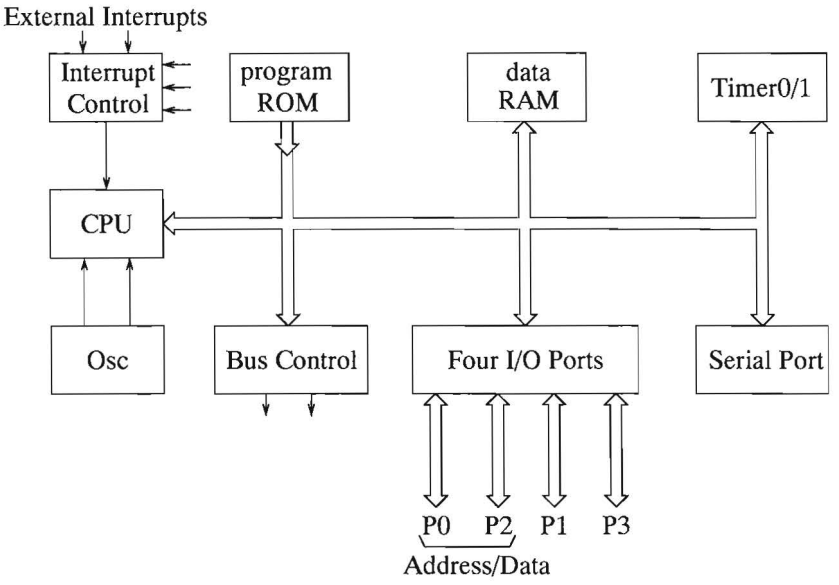


Figure 3.1: 80C51 block diagram.

be split into an internal and an external part. External memory is accessed using the I/O-ports: first a 16-bit address is sent along ports 0 and 2, and then the external memory puts the data on port 0. Most embedded 80C51 microcontrollers will only fetch from the internal program memory. The data memory (implemented as RAM) can also be split into an internal and an external part. The internal data RAM contains four *register banks* of eight registers each. Furthermore, a small part of the data memory is reserved as bit-addressable space. A special part of the memory is known as the space for the *Special Function Registers* (SFRs). These registers are readable and writable by the CPU, and interface between the CPU and the peripherals.

The 80C51 instruction set contains 255 instructions, of which the opcode is encoded in eight bits. An instruction can carry addresses of source and destination registers, making the instruction length encoding variable (1, 2, or 3 bytes). The complete 80C51 instruction set can be found in the table in Appendix B. In this table the format of the instruction opcode is P_i $++Z_j$ (i and j in hexadecimal notation). Note that columns 8 to F are taken together; the last three bits of the instruction code specify a register (0..7) in a register bank. The same goes for columns 6 and 7 that involve the instructions using indirect addressing: the last

bit represents the register (0 or 1) that contains the address of the register to be operated on. The instruction set can be partitioned into five classes:

Arithmetic instructions: ADD (addition), ADDC (add with carry), INC (increment), DEC (decrement), MUL (multiply), DIV (divide), and DA (Decimal Adjust);

Logical instructions: ANL (logic AND on bit patterns), ORL (logic OR), XRL (logic exclusive OR), CLR (clear), CPL (complement), SWAP and several instructions that rotate bit patterns;

Data transfer: moving data from and to internal (MOV) as well as external data memory (MOVX, MOVC);

Boolean instructions: instructions that operate on individual bits of registers;

Jump instructions: instructions that can conditionally or unconditionally change the contents of the program counter.

The 80C51 instruction set supports six addressing modes:

Direct addressing: the operand is specified by an 8-bit address field in the instruction code. Only internal data RAM and SFRs (special function registers) can be directly addressed. Example: INC e0h (operation: $A := A + 1$, e0h is the direct address of the accumulator A);

Indirect addressing: the instruction specifies a register that contains the address of the operand. Both internal and external RAM can be indirectly addressed. Example: INC @R_0 (operation: $(R_0) := (R_0) + 1$);

Register instructions: the register banks, containing registers $R_0 \ldots R_7$ can be accessed by instructions that carry a 3-bit register specification within the opcode of the instruction. Example: DEC R_3 (operation: $R_3 := R_3 - 1$);

Register-specific instructions: some instructions operate on specific registers. Example: RR A, which rotates the bit pattern in the accumulator A;

Immediate constants: the value of a constant is part of the instruction code. Example: MOV A,#100 (operation: $A := 100$);

Indexed addressing: only program memory can be accessed with indexed addressing, and it can only be read. This addressing mode is intended for reading look-up tables in the program memory where the address of in the table is formed by adding the accumulator to a base pointer.

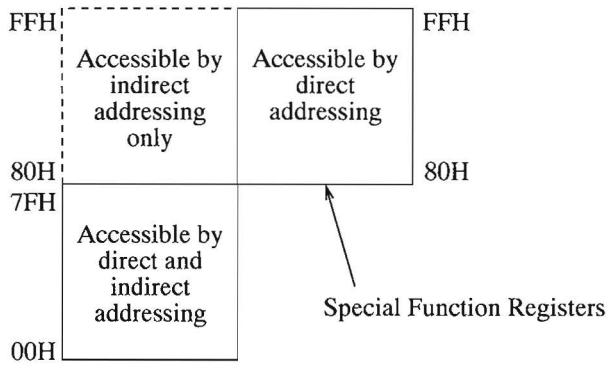


Figure 3.2: Internal data memory.

The memory structure of the 80C51 is not uniform. This goes for the registers first; the system is not built around a uniform register file. All registers have separate addresses that can be part of the instruction encoding. Also the structure of the internal data RAM is not uniform; it is even so that some addresses are shared with the space for the Special Function Registers (SFRs), as shown in Figure 3.2. The SFRs are the registers that interface between the CPU and the peripherals; they contain both control information as well as data, as described in the next section. The SFRs are accessible only by direct addressing. The upper half of the internal data memory has the same address-space, but is only addressable by indirect addressing. In the standard 80C51 the SFR-space is not completely filled: the data handbook shows that this standard version has only 21 of the 128 available places in the address space occupied by SFRs [4].

The synchronous architecture that implements the instruction set is shown in Figure 3.3. It is built around the *internal bus* IB, to which all registers can write and from which all registers can read. In the picture we see all registers, an ALU, the SFR-space, and the four bidirectional ports. A separate bus ("B" in Figure 3.3) is used for modifying the program counter PC.

All communications between registers use the IB-bus, except the communications for modifying the program counter PC. Having only one bus for these communications makes a compact implementation of the datapath possible. However, it also implies *sequential* execution of instructions. These executions take place in a number of steps, each of which communicates a value from one register to another or does some calculation. As all communications use the bus, the steps in each in-

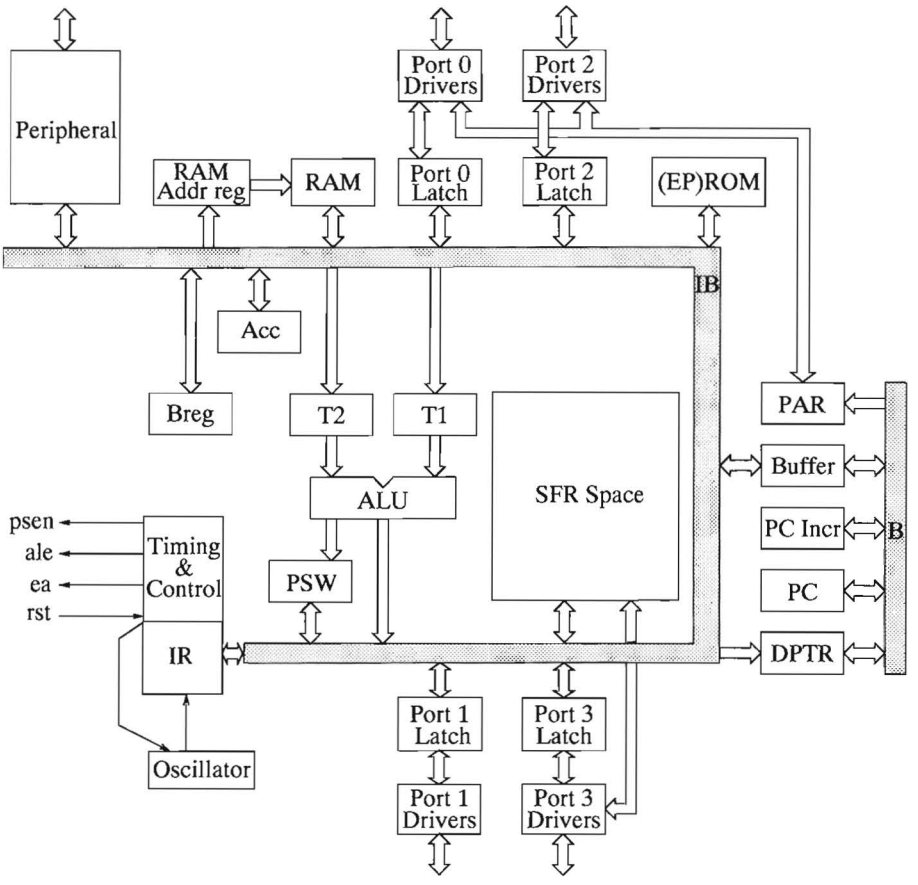


Figure 3.3: 80C51 synchronous architecture.

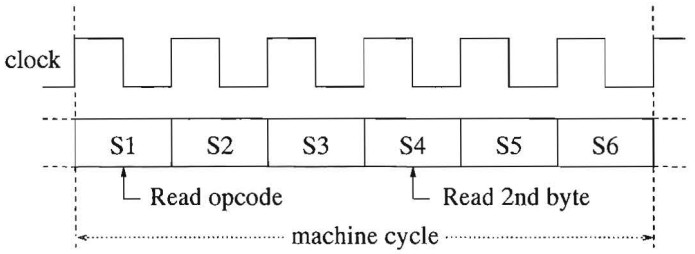


Figure 3.4: Clocking scheme for instruction MOV A,#data.

struction execution have to be done sequentially. When one would want to overlap the execution of instructions (i.e. implement *pipelining*) then one would have to separate pieces of datapath for each stage in the pipeline. Therefore it is difficult to implement pipelining using this architecture. (**Remark:** The 80C51XA (eXtended Architecture) 16-bit microcontrollers are implemented using a three-stage pipeline [3]). In the 80C51 we see that there are two separate pieces of datapath: the internal bus IB and the program-counter bus B. Therefore we can do two steps in parallel per step in the instruction execution: one communication using bus IB and one communication using bus B. It is, for example, possible to fetch a byte from the program ROM while incrementing the program counter PC.

The 80C51 instructions require a number of steps to execute, and therefore it is important to look for a scheme in which all executions fit. The instructions are executed with respect to a clocking scheme. Each instruction takes one, two or four *machine cycles* to execute. Each machine cycle consists of six *slots*, and each slot takes a clock cycle. Only the divide (`DIV`) and the multiply (`MUL`) instructions take four machine cycles; the other instructions take one or two machine cycles. (Remark: the 80C51 Data Handbook specifies that each slot in the instruction execution takes two clock cycles [4]. Recent synchronous 80C51 implementations, however, use internally only one clock cycle per slot.)

The clocking scheme of instruction MOV A,#data, which is a 2-byte 1-cycle instruction, is shown in Figure 3.4. The execution steps that take place in the six slots are listed in Table 3.1. In this table we see the required control signals and the data transfer actions that take place in the synchronous implementation to execute this instruction. Note that the IB-bus is used in all slots except slot 5. When we consider all 80C51 instructions we see that the IB-bus is used in $\frac{3}{4}$ of all slots. The actions in upper case letters in Table 3.1 denote required actions for this instruc-

S1	S2	S3	S4	S5	S6
ROM→IB	acc→ib	ram→ib	ROM→IB	QA=000	ALU→IB
IB→IR	ib→t2	ib→buffer	IB→T2	(ADD)	IB→ACC
0 →T1					
INCR PC			INCR PC		

Table 3.1: Instruction execution scheme of MOV A,# data.

	S1	S2	S3	S4	S5	S6
C1	ROM access	ACC→ T2	RAM access	ROM access	OP→ T1/T2	ALU→ destination

	S1	S2	S3	S4	S5	S6
C2	ROM access	calculate jump address		PC incr.	OP→ T1/T2	ALU→ destination

Table 3.2: General 80C51 instruction execution scheme.

tion; the lower case letters denote *redundant actions*. Execution of these redundant actions simplify the implementation of the CPU, but they have no impact on the state of the controller after completion. In the above instruction, for example, the instruction code consists of two bytes that are fetched from the program ROM in slots 1 and 4. In these slots also the program counter is incremented. In slot 6 data is transferred to the accumulator. In slots 2 and 3 redundant actions take place: the contents from the accumulator is copied into T2 and an item from the data RAM is placed in the Buffer.

The general execution scheme in which all 80C51 instructions except the divide and multiply instructions (DIV and MUL) fit is shown in Table 3.2. The DIV and MUL instructions are implemented using the shift-and-add algorithm as described in [15, 20]. An instruction consists of one, two, or three bytes; an opcode and two bytes containing operand addresses or immediate data. These bytes are fetched in the first and fourth slot of the first machine cycle and the first slot of the second machine cycle. Slot 2 of the first machine cycle copies the contents of the accumulator into register `T2`. For many instructions this is a redundant action. Slot 3 does a RAM access (which also includes access to one of the four register banks). Slots 5 and 6 of the machine first cycle take care of the ALU operation to be performed

and the write-back to the destination register. Machine cycle 2 starts with another ROM-fetch, after which the jump instructions calculate their offset (slots 2 and 3). For 2-cycle non-jump instructions, the actions in these slots are redundant. The fourth slot increments the program counter, and the 5th and 6th slot take care of an ALU operation. It turns out that approximately $\frac{1}{3}$ of all slots contain redundant actions.

When drawing a complete instruction execution scheme of the 2-cycle instructions it turns out that in the second cycle not much "useful" work is done. This is one of the major aspects of the synchronous implementation where we can save on power and on execution time; leave out the redundant actions and execute only those statements that are required for that instruction. In terms of the power equation $P = V_{dd}^2 * f_{clk} * C * \frac{1}{2}\alpha$, reduce the activity factor α to save power.

3.2.3 80C51 peripherals

Peripherals assist the CPU in its task and take care of the communication between the CPU and the outside world. Examples of peripherals are timers, the interrupt control, the input/output block, and the UART.

Timers and counters provide for timing references; they can be configured to either count multiples of clock-ticks (in timer mode) or events on external pins (in counter mode). In all of these events the timers or counters are incremented. Their values can be inspected and they generate an interrupt when they overflow. Timers and counters can serve as a timing reference to either the CPU or to another peripheral.

The **Interrupt controller** takes care of the proper dealing with (possibly external) interrupts. It decides which interrupt has priority over another and it supplies the CPU with an *interrupt address vector*. This vector points to the piece of program code (the *interrupt service routine*) that the CPU executes upon occurrence of an interrupt.

The **Input/Output** (I/O) peripheral takes care of the communication between the CPU and the outside world. It is involved in executing the proper protocol when external memory access is required. The protocol is an agreement between the microcontroller and the environment on the validity of data on external ports. In order for a new design to work in an existing environment, it is essential that the new design implements these protocols. An example of this issue of *timing compatibility* with an existing environment is discussed in Chapter 4 of this thesis.

The **UART** (Universal Asynchronous Receiver and Transmitter) is a peripheral

that takes care of bit-serial transmission and reception of bit-patterns. A standard 80C51 UART can be configured in four modes with various baud-rates that depend either on the clock or on a timer-overflow. The configuration of the UART is done in software, by the program running on the 80C51 CPU that reads and writes the SFRs for the UART. An asynchronous implementation of the UART for the 80C51 is discussed as a case-study at the end of Chapter 6.

The bridge between CPU and peripherals is formed by the Special Function Registers. These registers are readable and writable by the CPU and by the peripherals. They can be divided in *control* registers and *data* registers. The control registers specify the configuration of the peripherals and contain the interrupt bits. The data registers are used for communication between the CPU and the environment. Chapter 6 describes an implementation of the peripherals and their communication with the CPU, as well as with the environment.

3.3 CISC-nature of 80C51

The literature of microprocessor design makes a distinction between two classes of processors: *RISC* (Reduced Instructions Set Computer) and *CISC* (Complex Instruction Set Computer) [15, 11]. These classes differ in the complexity of the functionality of the instructions, the style of encoding of the instructions, and the uniformity of the register structure in the implementation. The 80C51 microcontroller can be considered a *CISC* because of the following characteristics.

1. There are *various addressing modes*: direct addressing, indirect addressing, register bank addressing, and so on. It is even the case that the same addresses are mapped onto different register spaces, as was shown for the SFRs and the data RAM.

2. The instructions are *encoded in variable length*, either in one, two, or three bytes, of which the first byte contains the opcode of the instruction.

3. The *register structure is not uniform*. There are four register banks that reside in memory. Furthermore, there is the SFR-space that is only partially filled.

4. It takes a *variable number of clock cycles to execute an instruction*. We have seen that each instruction execution fits into the execution scheme as shown in Table 3.2. Each instruction takes either one, two, or even four machine cycles (i.e. 6, 12, or 24 clock cycles) to execute.

In contrast, the basis of a *RISC* (Reduced Instruction Set Computer) processor is its uniform register structure. For example, a *load-store* architecture like the DLX [15] is built around a *register file*. All instructions access registers in this file and perform operations on them. Therefore there is only one addressing mode in a RISC machine: the one that addresses a register in the register file by its index.

In a RISC instruction set the instructions have a fixed length. For example, the DLX has 32-bit wide instructions, as have the MIPS R3000 [18] and the ARM6 [11]. In these machines the instruction opcode can be split into several *fields*, for example an opcode field, three indices for registers (two source registers and one destination), and an immediate field. The main advantage of fixed-length instruction encoding is the simplicity of the decoding in the processor, which is good for speed and power dissipation. Furthermore, for each instruction only one access to the program memory is needed. On the other hand, fixed-length instruction encoding implies *redundancy*; the instruction words are larger (in number of bits) than is strictly necessary. In contrast, the 80C51 instruction set contains 255 instruction opcodes encoded in 8 bits, which keeps redundancy to a minimum. However, as instructions may carry immediate values or addresses of registers, several accesses to program memory are needed in the execution of some 80C51 instructions.

Instructions in a RISC instruction set tend to be simpler in functionality than in a CISC instruction set. This is a result of the uniform register space and uniform addressing. Therefore, a RISC instruction can in general be executed in fewer steps than a CISC instruction. These steps are called "stages" in a RISC instruction execution. A DLX instruction, for example, can be executed in only 5 stages: an instruction fetch; the instruction decode and register file access; an ALU operation; a memory access; and finally a write-back into the register file. The ARM6 and ARM7 instructions can be executed in only three stages (fetch, decode and execute) [11].

In a RISC machine the datapath of the processor and the execution schemes of the instructions are designed simultaneously. Each stage in the execution scheme is designed in such a way that it uses a separate part of the processor's datapath. Also, each stage can generally be executed in one clock cycle. This makes it possible to overlap execution of consecutive instructions, which is implemented by *pipelining*. Pipelining makes it possible complete one instruction execution per clock cycle, i.e. achieve a CPI close to 1 (CPI = Clock Cycles per Instruction). The 80C51, however, is a sequential machine in which each slot (stage) of the instruction execution uses the bus in the datapath. Therefore, instruction execution cannot be overlapped unless a stage does not use the bus (i.e. when there is a communication path that *bypasses* the bus). When a slot takes one clock cycle, the CPI of the

80C51 is approximately 9, depending on the program that the CPU runs. This is a higher number than the CPI of a many RISC machines.

On the other hand, an instruction in a CISC-machine is more powerful and does more work, in general, than a RISC instruction. Indirect addressing, for example, is possible in various instructions in the 80C51, but it would take at least two separate instructions in a RISC (one for fetching the address, the second for collecting the data). Another example is the 80C51 instruction DJNZ (Decrement and Jump if Not Zero). This instruction executes the following steps:

- First, the value of a register is decremented;
- Then, if the result is not equal to zero, the jump is taken.

In a RISC instruction set the functionality of the DJNZ instruction would take at least two instructions: one for decrementing the register, and the second to take the jump when the result would not equal zero. In general, when we compile the same program to RISC instructions and to CISC instructions, the CISC program will contain fewer instructions.

3.4 Power analysis

In this section we describe a model of the power dissipation of a synchronous 80C51 to gain insight in where the power is dissipated. This model is used only for getting a feel of where we should pay attention to when reducing the power dissipation.

We take a netlist of a synchronous 80C51 including peripherals, as starting point. This 80C51 is a recent VHDL-synthesized implementation by Philips [35]. The netlist is used for simulation in which we can find activities of the nets. The benchmark for simulation is a program that contains all instructions of the 80C51 instruction set.

The synchronous 80C51 implementation can be partitioned into the *control* and the *datapath*, as shown in Figure 3.5. An important part of the datapath is the *bus*. Both the control and the datapath consist of *flipflops* and *combinatorics*. The flipflops are steered by the *clock*. The flipflops that are connected to the bus are multiplexed, so that they keep their value when they are not addressed in a bus-communication.

For the power we take the formula as given in Section 1.3:

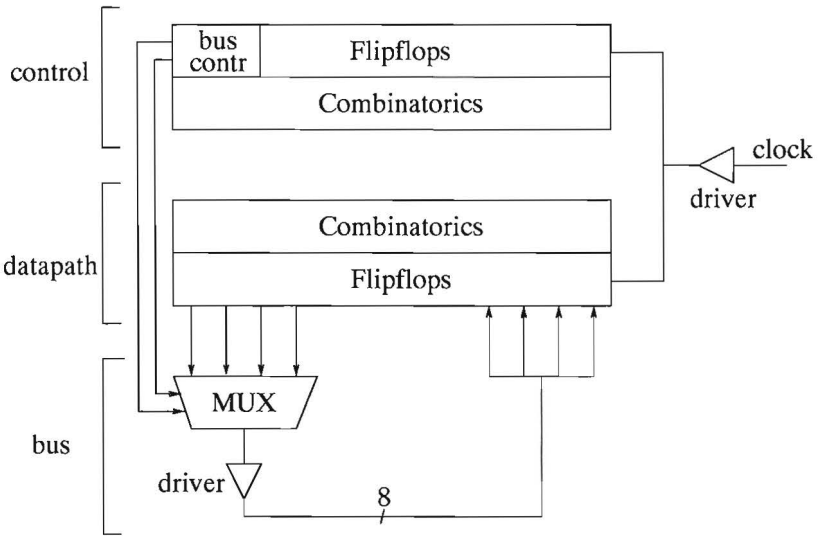


Figure 3.5: Model of a synchronous 80C51 implementation.

$$P = \frac{1}{2}\alpha * C * f_{\text{clk}} * V_{\text{dd}}^2$$

f_{clk} and V_{dd}^2 are constants in a simulation. The variable components in this formula are α and C. For α we take $\alpha_{\text{clock}} = 2$, implying that a gate has $\alpha = 1$ when the output of that gate switches once at each clock cycle.

- For the flipflops, we distinguish between the datapath and the control of the circuit, as they show a different activity α. Simulation shows that for the datapath we have $\alpha = 0.06$, and for the control we have $\alpha = 0.2$, which is due to the compact encoding of the state space. For the capacitance C_{ff} of the flipflops, we take the following:

$$C_{\text{ff}} = C_{\text{ff,clock}} + C_{\text{ff,data}}$$

$$C_{\text{ff,clock}} = C_{\text{clock, internal}} + C_{\text{clock, input}}$$

$$C_{\text{ff,data}} = C_{\text{data, internal}} + C_{\text{data, input}} + C_{\text{wire}}$$

For the wire capacity of the data part we multiply the average fanout with a standard capacity number: $C_{\text{fanout}} * \text{fanout}_{\text{average}}$.

- We take the combinatoric circuitry of the control and the datapath together. We adopt the following model: we count the number of gate equivalents (NAND-2) and observe from simulation that the average activity $\alpha = 0.06$. For the capacitance we take

$$C_{\text{comb}} = C_{\text{comb, input}} + C_{\text{comb, internal}} + C_{\text{wire}}$$

For the wire capacity we take the same formula as used for the wire capacity of the flipflops: $C_{\text{fanout}} * \text{fanout}_{\text{average}}$.

- For the bus, we observe that $\alpha = 0.43$ from simulation. This is also explained by the fact that the bus is used in approximately 75% of all slots for communicating uncorrelated data, which means that on average half of the wires will switch per bus communication. For the capacitance we take

$$C_{\text{bus}} = C_{\text{wire}} + C_{\text{driver}}$$

For the wires we estimate the length of the wires in the layout, which is 1.5 by 1.5 mm. A bus wire has 30 flipflops connected to it. We estimate that the length of a bus wire is approximately 12 mm. This number is multiplied with a capacity per mm, which is known from the technology in which the synchronous chip was produced. The load capacitance is counted with the the flipflops.

- For the clock, we observe that $\alpha = 2$. The capacitance is estimated in the same way as for the bus:

$$C_{\text{clock}} = C_{\text{wire}} + C_{\text{driver}}$$

For the length of a clock wire, we observe that there are in total 732 flipflops to be controlled. Suppose that these are distributed in a matrix of 27 by 27 flipflops, then we have a clock wire length of approximately 41 mm. The load capacitance is counted with the flipflops.

For the bus we have to make a correction. A number of flipflops in the control steer the bus communication; the energy dissipated by these flipflops is added to the bus-energy. Furthermore, the multiplexer to the bus consists of complex gates

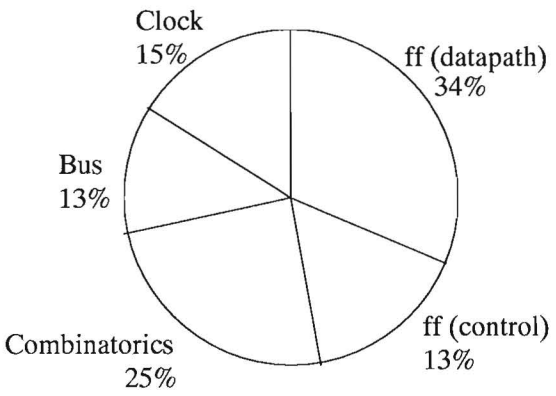


Figure 3.6: Distribution of the energy dissipation in the synchronous 80C51. A considerable part of the energy dissipated by the flipflops is due to clocking. This makes the total clock energy about 50%.

in the netlist implementation; for this combinatoric part of the circuit we estimate the energy dissipation and add it to the bus-energy. We then obtain the energy-dissipation distribution as shown in Figure 3.6.

From this distribution we see two main sources of energy dissipation. First we have the *control* of the circuit: when the clock energy dissipated *in* the flipflops is added to the total clock energy, then this part contributes about 50% to the total dissipation. This coincides with numbers in literature [21]. Second, the *bus* energy is a source of possible savings.

3.5 Low-power opportunities

Taking the architecture of the synchronous implementation and the analysis of the previous section in mind we observe the following low-power opportunities.

1. The *centralized control* with a *compact encoding* of the state space implies a high switching activity in the control of the synchronous implementation. A more redundant encoding of the state space would result in a larger control structure, but the changes between states could be encoded more locally. This would result in less activity in the control. However, it would require more flipflops for the state encoding, and therefore more clock power. The

centralized control fits nicely with the sequential execution of instructions using the slot-structure. In each slot some action (i.e. communication in the datapath) is taken, whether that is necessary or not. For the instruction execution scheme of the 80C51 it turns out that approximately $\frac{1}{3}$ of the actions in the slots are redundant. *Distributed control* enables the designer to leave out a communication when it is redundant. This will cost some extra area for the control, but it can also save activity in the datapath *and* in the control.

2. The *clock* enables all registers (flipflops) in the synchronous implementation. Some implementations have auxiliary clocks running at one tick per machine cycle, e.g. to support ROM access in the first slot, or to reduce the activity in the timer-peripheral. However, for the CPU, the sequentiality of instruction execution makes it necessary for the clock to generate at least six and sometimes twelve ticks per instruction. It turns out that less than 10% of all registers is updated in a slot. When all registers are clocked, many of these clock cycles are thus not necessary for the majority of the registers.

3. The *IB-bus* stands central in the synchronous architecture. All registers are connected to this bus, making its wires in the circuit layout relatively long, resulting in a high switching *capacitance*. Furthermore, it turns out that the IB-bus is used in approximately $\frac{3}{4}$ of all slots for communicating *uncorrelated* data. Therefore, in each slot of an execution of an instruction, on average half of the wires of the bus will switch, implying a high switching *activity* on the bus.

4. The datapath of some recent synchronous implementations is based on master-slave *flipflops* for each bit in a register. It is often possible to re-arrange the structure of communication in the chip in such a way that latches can be used instead of flipflops.

5. For the *peripherals* we can make the observation that their switching activity is generally lower than the clock frequency to the CPU. Furthermore, the activity can be quite irregular. An interrupt controller is a good example; it only needs to be activated when interrupts occur, which can be rare events that are not evenly spread in time. Making the peripherals *clock-driven* burns power unnecessarily. Even when there is some regularity in the activity of a peripheral, it is hardly necessary to keep up with the pace of the clock. An example of this is a timer which counts the number of executed instructions: it needs not to be triggered with the frequency of the clock to the CPU.

6. The synchronous 80C51 has two power-saving modes: *Idle mode* and *power-down mode*. Idle mode gates the clock off the CPU but keeps the peripherals

clocked, to make immediate response to an interrupt possible. This interrupt controller is then clock-driven, burning power unnecessarily. *Power-down* mode stops the oscillator from running with the disadvantage that it takes time (a few ms) to restart the system.

There are *five* low-power opportunities of asynchronous circuits described in [47]. These opportunities are: reduced clock power, distributed control, architectural freedom, elimination of standby power, and adaptive scaling of the supply voltage. This article shows two applications where these low-power opportunities of asynchronous circuits are exploited: the DCC error corrector and standby circuits for pagers. The six low-power opportunities for the 80C51 coincide with the first four opportunities in [47]. Adaptive scaling of the supply voltage is left outside the scope of this thesis.

With the six low-power opportunities for the 80C51 in mind, we develop a Tangram program of the 80C51 microcontroller in the next chapters. First a global overview of an asynchronous 80C51 is given, and the design decisions and their implications at this global level are discussed.

Chapter 4

An Asynchronous 80C51 Microcontroller Architecture

This chapter gives a global overview of the asynchronous 80C51 microcontroller that we will describe in detail in the next chapters. The synchronous implementation is divided into several blocks, viz. the CPU and the peripherals. In principle, we wish to make an asynchronous implementation of the 80C51 that is *compatible* with the synchronous implementation. The situation we then aim for is that the asynchronous version is a *plug-and-play* substitute for the synchronous one, i.e. showing the same external behaviour. The issue of compatibility is explained in more detail in Section 4.3.

Asynchronous ICs have no clock, i.e. there is no global timing reference to the system. The synchronous 80C51 implementation has the notion of a global timing reference, which is used for correct functioning. An asynchronous implementation of the microcontroller should therefore have facilities to mimic the synchronous timing behaviour when and where necessary, without burdening the potential advantages of asynchronous implementation, i.e. low-power, average case execution time, and low electro-magnetic emission.

This chapter explains the partition of the various blocks and their interfaces in the asynchronous 80C51 microcontroller. Furthermore, the issue of compatibility is discussed.

4.1 Partition of the 80C51 microcontroller

In the previous chapter we have seen that the synchronous 80C51 microcontroller is divided into the *CPU* with embedded memories, and the *peripherals* (Figure 3.1). They communicate with each other using the IB-*bus*, which is controlled by the global clock signal.

The *CPU* is the part of the microcontroller that fetches and executes instructions. To this end, it makes use of two memories: one for the program code and one for the data. The specification of the CPU is the *instruction set*, which is *fixed*. The instruction set specifies the registers in the CPU, the various addressing modes, and the operations that the CPU can perform. For most derivatives of the microcontroller, the CPU is fixed (as the instruction set is fixed), but the sizes and kinds of the memories ((E)PROM, FLASH, OTP etc.) may differ.

A *peripheral* is a small block of hardware that can perform a specialized and usually well-defined task. An example of a peripheral is the timer block that contains a number of timers to be configured to count events or units of time. Peripherals also form the boundary of the chip to the environment. For example, the port logic can be considered to be a peripheral. Peripherals communicate with the CPU using shared memory, viz. the *Special Function Registers*. These registers contain the control and data information for the peripheral.

Peripherals make the difference in functionality between the derivatives of a microcontroller. For example, a derivative may contain an extra timer, which is then an extension of the timer-peripheral. But a derivative may also contain a complete new peripheral like a UART, which takes care of serial transmission and reception of data. The operation of peripherals is often *demand-driven* as opposed to *clock-driven*. For example, the interrupt controller checks if an interrupt has occurred. An interrupt is an asynchronous event: it is not known on beforehand when an interrupt will occur. An asynchronous circuit is demand-driven by nature: a piece of circuitry only starts operating when there has been a request to do so. Therefore the demand-driven operation of peripherals fits nicely with the demand-driven operation of an asynchronous circuit.

To allow for fast generation of a new derivative of a microcontroller it is important that new peripherals can be added to an existing microcontroller. Therefore we choose to design the microcontroller in a *modular* fashion, containing a CPU and peripherals. Furthermore we have a Synchronizer unit that deals with the implementation of *timing compatibility*, as we will discuss later on this chapter. The asynchronous 80C51 is shown in Figure 4.1.

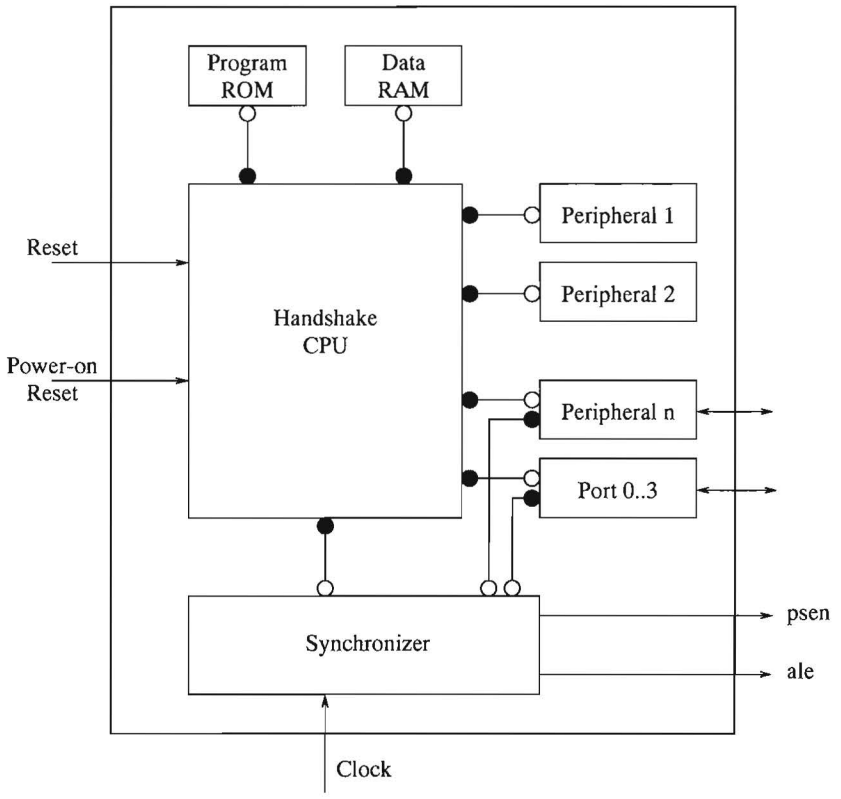


Figure 4.1: Global structure of an asynchronous 80C51 microcontroller. The microcontroller consists of a handshake CPU, a number of peripherals and the Synchronizer-unit, which serves as a timing reference to some peripherals and to the environment. This chapter discusses the compatibility issues and the Synchronizer. Chapter 5 of this thesis deals with the handshake CPU, and Chapter 6 discusses the peripherals and their interfaces to the CPU and to the environment.

4.2 Communication

For the internal communication between CPU and peripherals on the asynchronous 80C51 we choose *handshake communication*. This has the advantage that it is possible to communicate between CPU and peripherals only when and where necessary, which potentially keeps the energy dissipation to a minimum. Also, handshake communication between the various blocks makes it possible to design the system within the Tangram framework.

The 80C51 uses the Special Function Registers as communication medium between the CPU and the peripherals. The SFRs are part of the register space as specified by the instruction set. Therefore we adopt the same space for the SFRs in the asynchronous 80C51. We also would like to exploit the advantages offered by the asynchronous implementation as much as possible. For example, none of the components, CPU and peripherals, should be blocked from operating when that is not necessary. Both the CPU and the peripherals should be able to instantly access and modify the SFRs whenever necessary. In other words, we would like to *decouple* their operation and communicate between them only when and where necessary. Furthermore, no component should dissipate energy unnecessarily. These constraints make it necessary to locate the SFRs in between in the CPU and the peripherals, using a so-called *SFR-interface*. This interface implements the above mentioned constraints. The design of the peripherals and their interfaces is discussed in detail in Chapter 6.

The handshake CPU communicates with its memories using handshake interfaces. The functionality of the program memory can be described in Tangram by receiving the address and sending the data at that address in the program ROM:

```
ROM : proc() . forever
               do ROMaddr?addr
               ;  ROMdata!ROM[addr]
               od
```

The data RAM is different from the program ROM in that it can be read and written. The difference between the two is encoded by an extra bit that is supplied with the address (as we have seen in Chapter 2 with the Move Machine):

```
RAM : proc() . forever
               do RAMaddr?<<rw,addr>>
               ;  if rw
                  then /* Read  */ RAMdataout!RAM[addr]
                  else /* Write */ RAMdatain?RAM[addr]
                  fi
               od
```

The only interface channels to the CPU that are not handshake channels, are the Reset and Power-On-Reset lines. A transition on these lines forces the microcontroller to stop its current activity and replace the state by the initial state. This so-called *special condition*, together with the handling of interrupts is discussed in Chapter 5 about the design of the CPU.

4.3 Synchronization: compatibility

A chip is *fully compatible* with another chip when their packages are replaceable without making any adjustments to the environment. Put differently, two ICs are fully compatible when the environment cannot distinguish between the two. Compatibility can be divided into two classes:

- *Bit-compatibility*: two ICs are bit-compatible when they can execute the same program *code* and produce the same data;
- *Timing-compatibility:* both ICs implement the same assumptions on timing. For example, when the 80C51 accesses external memory using two external ports, there is a prescribed protocol where signals denote the validity of data on these ports. The environment inspects these signals and acts accordingly.

Pin-compatibility combines both bit and timing-compatibility and demands that a pin on one chip has the same functionality as the pin at the same location on the other chip. Therefore, also the pins for power and ground on both chips have to be at the same location. Two chips are pin-compatible when their pins show the same bit and timing behaviour when communicating with the same environment.

Bit-compatibility is ensured by taking the same instruction set and the same encoding as starting point. That is what we will do with the asynchronous 80C51: it will run the same program code as its synchronous counterpart.

The issue of timing-compatibility has to be paid special attention to. Asynchronous circuits have the property that they run *freely*, i.e. as quickly as possible and without a global clock to control their operation. Asynchronous circuits therefore show *average-case* execution time: some tasks take a short time and some take a longer time. An asynchronous CPU executes some instructions in shorter time than others. A simple MOV-instruction, for example, is finished earlier than a complicated multiply (MUL) instruction. In the execution schemes of the synchronous 80C51 instructions we also see variance in the instruction execution time: an instruction

may take one, two, or four machine cycles to complete. However, an asynchronous CPU shows a more fine-grained variation in instruction execution times.

It is possible, in principle, to execute instructions in an asynchronous chip at the same pace as in a synchronous chip. The asynchronous chip would then have to synchronize with a timing reference (i.e. a clock) to mimic the synchronous timing behaviour. Suppose that all timing requirements are met, i.e. all actions between two clock ticks in the synchronous chip can also take place between two clock ticks in the asynchronous chip. Then the asynchronous chip would implement *worst-case* execution time, just as the synchronous chip. Therefore the asynchronous IC would not show the potential advantage of average-case execution time anymore. Therefore we choose to synchronize the asynchronous chip with a clock only when necessary. We want to exploit the advantage of average-case execution time and choose not to synchronize the CPU with the clock during the execution of each instruction. Instead we choose to start the execution of the next instruction immediately after finishing the previous one. Therefore our implementation of the asynchronous 80C51 will not be *fully* timing compatible with its synchronous counterpart.

In the 80C51, various peripherals need a timing reference to perform their tasks. A timer, for example, can be configured in such a way that it counts machine cycles (i.e. blocks of 6 clock ticks) for which it needs a timing reference. Another example is the UART as introduced in the previous chapter. It operates at a certain *baud-rate*, which can be derived from a clock signal, for example.

Therefore we need a timing reference (i.e. a clock) to the asynchronous 80C51 to be able to make it timing-compatible with its synchronous counterpart where necessary. However, we want to distribute this clock signal (or a derivative of the clock) only to those blocks of the circuit that need it, avoiding unnecessary energy dissipation. The various blocks should run as autonomously as possible, only synchronizing with each other and with the clock when necessary.

To this end, we have a separate block in the asynchronous microcontroller: the *Synchronizer* (Figure 4.1). This block takes an external clock signal as input and communicates with peripherals that need a timing reference signal. These communications can in principle be implemented using handshakes, but that would result in the use of arbiters in the peripherals to choose between a communication from the CPU or from the Synchronizer [30]. Another possibility is to use internal *direct* channels. These channels are implemented using a single wire without handshake protocol. In Tangram, a direct channel `d` can be inspected for its value (by the statement `sample(d)`), for up-going (`edge(d/)`), down-going (`edge(d\)`), or any transition (`edge(d)`). The use of direct channels saves the implementation of

arbiters, which is good for area, speed, and execution time. On the other hand, the use of direct channels also implies that the designer has to verify that timing constraints are met in the implementation.

The Synchronizer also takes care of *external* timing signals that are also implemented as direct channels. Consider the *external memory access* mode of the 80C51 [4]. External memory access takes place in two steps:

- first, a 16-bit address is sent along ports 0 and 2;
- then, the environment provides the microcontroller with the 8-bit data (the instruction byte) along port 0.

The environment has to determine whether the microcontroller wants to access the external memory, when the data on ports 0 and 2 (the address) is valid, and when the data from the environment has to be valid on port 0. To this end, the microcontroller provides two extra signals, `psen` and `ale`, to determine the data-validity. These signals depend on the global clock signal, and the protocol is shown in Figure 4.2 [4].

The protocol is as follows. Halfway through slot 1, both `psen` and `ale` go high. At the beginning of slot 2, the microcontroller puts the 16-bit address on port 0 (the low-order eight bits) and port 2 (the high-order eight bits). Halfway through slot 2 `ale` goes low, to indicate that the environment can latch the the data on port 0 in an external latch (address latch enable). At the beginning of slot 3, `psen` goes low, indicating that port 0 is now free to put data on. The environment now gets the time to put the data on port 0. At the beginning of slot 4 this data is supposed to be valid on this port in such a way that the microcontroller can read it.

The dotted arrows indicate that the transitions on the wires `psen` and `ale` and the data-validity on port 0 and 2 are dictated by the clock. In Tangram, we can mimic the protocol by using direct channels. The Synchronizer inspects transitions on the clock, and synchronizes with the CPU, which takes care of the data on the ports. The two corresponding pieces of Tangram text for the Synchronizer and for the CPU are shown in Figure 4.3. At slot 4, 5, and 6 of the machine cycle the same protocol can be repeated. In this way it is possible to do two external memory accesses per machine cycle, viz. in slot 1 and 4 of the synchronous instruction execution scheme (Table 3.2).

This implementation imposes some timing requirements on the implementation of the Synchronizer and the CPU. For example, signal `ale` going low indicates to the environment that the data on Port 0 (and Port 2) is valid. The CPU is signaled one

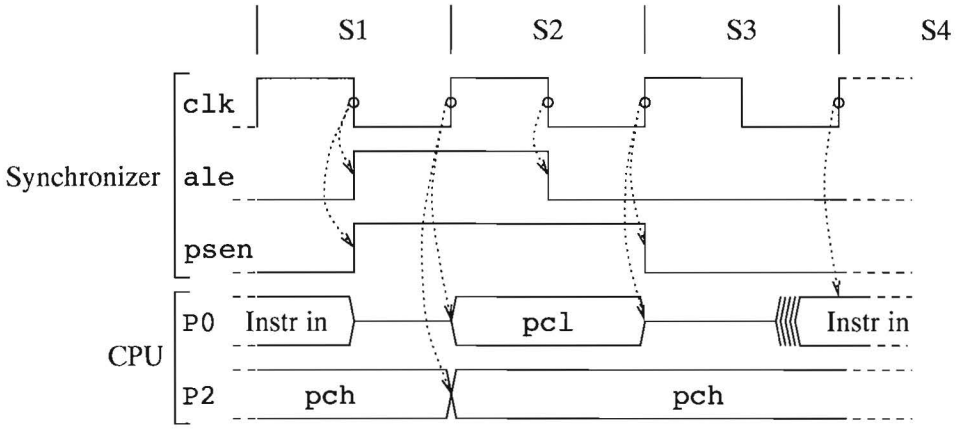


Figure 4.2: External access protocol in the synchronous 80C51: external signals `psen` and `ale` denote the validity of the data on ports 0 and 2.

Synchronizer		CPU-fragment
`forever`		
`do sync~`	←	`; sync~`
`; edge(clk) || ale:=1 || psen:=1`		
`; edge(clk) ; sync~`	→	`; sync~`
`; edge(clk) ; ale:=0`		`; ( p0out!pcl`
`; edge(clk) ; psen:=0`		`|| p2out!pch`
`; edge(clk)`		`)`
`; edge(clk) ; sync~`	→	`; sync~`
		`; p0in?data`
`od`		

Figure 4.3: Tangram program for the Synchronizer (left) and the corresponding Tangram program fragment for the CPU (right), implementing the timing protocol of Figure 4.2. The arrows indicate how to read through the combined program.

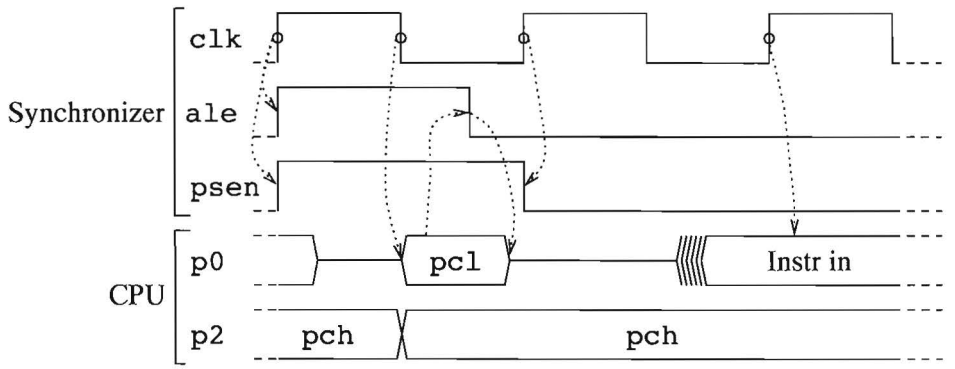


Figure 4.4: When the completion of the output of `pcl` along `p0` is used to signal the 1-to-0 transition on `ale`, one clock tick can be saved (five instead of six clock ticks) to implement the protocol (cf. Figure 4.2).

Synchronizer		CPU - fragment
`forever`		
`do sync~`	←	`; sync~`
`; edge(clk) || ale:=1 || psen:=1`		
`; edge(clk) ; sync~`	→	`; sync~`
		`; ( p0out!pcl`
		`|| p2out!pch`
		`)`
`; sync~`	←	`; sync~`
`; ale:=0`		
`; edge(clk) ; psen:=0`		
`; edge(clk)`		
`; edge(clk) ; sync~`	→	`; sync~`
		`; p0in?data`
`od`		

Figure 4.5: Tangram program for the Synchronizer (left) and the corresponding Tangram program fragment for the CPU (right), implementing the timing protocol of Figure 4.4.

clock transition earlier to put this data on these ports. Therefore, the timing interval between the two clock edges must be large enough for the Synchronizer and the CPU to synchronize and output the data along the ports.

We can do with a slightly slower clock in this case by noticing that `ale` may go low when the low-order part of the address is sent along Port 0. We then derive this fact from these actions being completed, rather than from the clock signal. We can then do the protocol with one clock tick less (i.e. five ticks instead of six), as shown in Figures 4.4 and 4.5. In an asynchronous implementation we can derive the timing of an event (1-to-0 transition of `ale`) from the completion of another event (`p0out!pcl`), as opposed to deriving it from the clock. This protocol can be repeated, but the difference with the protocol in Figure 4.2 is that the new execution of the protocol starts with a *down-going* edge instead of an up-going edge of the clock.

This scheme imposes another timing requirement on the implementation. The interval between the second and the third clock edge of the protocol must be large enough to synchronize between the Synchronizer and the CPU, put the data on the ports, establish a 1-to-0 transition on `ale`, and release the data on port 0.

In this way we can, in principle, build an interface between an asynchronous IC and a synchronous environment within the Tangram framework. This is possible under the assumption that the synchronization between the handshake modules and the actions these modules have to take, take place fast enough to meet the timing constraints.

In the asynchronous implementation of the 80C51 that we will develop over the next chapters, we make the chip bit-compatible in that it can run the same program code as its synchronous counterpart. For timing compatibility we implement the above scheme for external memory access.

4.4 Modular design

In terms of Tangram, the modular design as shown in Figure 4.1 can be expressed by writing procedures for the CPU, the Synchronizer, and the peripherals. These procedures implement both the functionality of the blocks as well as their interfaces. The procedures run in parallel and constitute the microcontroller system:

```
microcontroller
  : proc() .    CPU()
              || Synchronizer()
              || peripheral_1()
              || ...
              || peripheral_n()
```

The CPU is the most complex of these blocks, and the most interesting part of the microcontroller for an exploration of the design space. Furthermore, simulations of a synchronous implementation of the 80C51 show that in normal operation mode the CPU accounts for 50-60% of the total power dissipation. Therefore we first concentrate on the design of a low-power CPU in the next chapter. Chapter 6 discusses the design of the peripherals.

Chapter 5

An Asynchronous 80C51 CPU

This chapter addresses the design of a 80C51 handshake-CPU in Tangram. The CPU fetches and executes instructions, and is specified by the instruction set. Therefore we first focus on the instruction set and show what resources we need to implement the CPU. It turns out that the CPU can be split into two parts: the *datapath* and the *control*. The datapath contains the registers, communication paths and arithmetic circuitry; the control decodes the instructions and steers the datapath to execute them. For both parts we have various design alternatives that we discuss in this chapter. It turns out that both for the datapath and the control, the best design in terms of area, energy dissipation, and execution time, uses a mixture of these various design alternatives.

5.1 CPU and instruction set

The specification of the functionality of the CPU is given by the instruction set of the processor. An instruction set takes a set of *registers* and a set of *values* that the registers can assume. The *state* can then be viewed as a set of pairs, of which each pair represents a register and an associated value. Formally, when the instruction set assumes N registers, we have the following formula:

$$STATE : \{(R_i, v_i) | 0 \leq i < N\}.$$

An instruction can be seen as a mapping from states to states:

$$instr : STATE \rightarrow STATE$$

Instruction Class	Specification
ALU, Logical, Boolean	$PC := PC + c$; $R_{\text{dest}} := ALU(R_{\text{src1}}, R_{\text{src2}}, opc)$
Data transfer	$PC := PC + c$; $R_{\text{dest}} := R_{\text{src}}$
Jump and Branch	$PC := PC + c$; **if** $ALU(R_{\text{src1}}, R_{\text{src2}}, opc)$ **then** $PC := ALU(PC, offset, add)$ **else** skip **fi**

Table 5.1: Instruction classes and their specification. R_{dest} denotes a destination register, and $R_{\text{src1}}, R_{\text{src2}}$, and R_{src} denote source registers. ALU is a function of which the functionality is determined by opcode *opc*. PC is a special register, viz. the Program Counter, to which in each instruction class (usually small) constant c is added. The *offset* can be a positive or a negative value.

There is one special register, the *Program Counter PC*. The program counter is used to point to an instruction opcode. When an instruction is executed, the Program Counter is adjusted in such a way that it points at the next instruction. An instruction specifies which registers it operates on and which register values are changed by the instruction. This specification can be expressed by *assignments*. In the 80C51 we have various classes of instructions, as discussed in Chapter 3. The associated assignments that specify the instructions are shown in Table 5.1.

From Table 5.1 we see that we need two kinds of resources to implement instructions:

- Registers and communication paths between them;
- Arithmetic to perform operations on the values in registers (as denoted by the ALU function).

These two components constitute the *datapath* of the CPU. A datapath is capable of performing operations on register values and communicating these values. A *control* structure uses these resources and determines what parts of the datapath are used in what order in time. The control of a CPU has the following tasks:

- fetch an instruction from memory and add the constant c to the program counter;
- decode the instruction and determine the actions that the datapath has to take;
- control the datapath in such a way that the specification of the instruction is satisfied.

It depends on the structure of the datapath how a suitable control structure can be implemented. The control steers the various parts of the datapath, and determines what actions are taken in what order. Thus, the control also determines the *concurrency* and *sequentiality* in the operation of the circuit.

In the CPU we can implement a separate piece of datapath for each instruction, implying that we do not need to reuse pieces of hardware for different instructions. However, when we look at the specification of the various instructions in an instruction set, we can think of splitting each instruction execution into various steps: we then have an *instruction execution scheme*. We can look for overlap between the schemes for all instructions and try to reuse hardware for these steps. The circuits can then be produced cheaper because they implement fewer transistors.

Take the ALU-instructions in Table 5.1 as an example. Once the instruction is fetched and the program counter is incremented, the remainder of its specification is the assignment

$$R_{\text{dest}} := ALU(R_{\text{src1}}, R_{\text{src2}}, opc).$$

We wish to implement the ALU-function independent of the source and destination registers, and therefore we introduce *auxiliary registers* X, Y, and Z. We can then satisfy the specification by three steps as shown in Table 5.2.

Step a and step c deal with the *communication* of values between registers. In step b the ALU-function is performed on the two dedicated registers X and Y, and the result is stored in dedicated register Z. In hardware, this execution scheme makes it possible to *share* the ALU-*function* among all ALU-*instructions*.

$X, Y := R_{\text{src1}}, R_{\text{src2}}$	(step a)
$Z := ALU(X, Y, opc)$	(step b)
$R_{\text{dest}} := Z$	(step c)

Table 5.2: Execution scheme of an ALU instruction in three steps.

$X, Y := R_{\text{src1}}, R_{\text{src2}}$	(step 1a)
$Z := ALU(X, Y, opc)$	(step 1b)
$cc := Z$	(step 1c)
if cc **then**	
$\quad X, Y := PC, offset$	(step 2a)
$\quad Z := ALU(X, Y, add)$	(step 2b)
$\quad PC := Z$	(step 2c)
else skip	
fi	

Table 5.3: Execution scheme of a jump instruction.

The same can be done to find similarity in the execution of, for example, ALU-instructions and jump-instructions. A jump instruction, after increment of the program counter, is specified in Table 5.1 by

$$\textbf{if } ALU(R_{\text{src1}}, R_{\text{src2}}, opc) \textbf{ then } PC := ALU(PC, offset, add) \textbf{ else skip fi}\,.$$

This (conditional) assignment can be split into

$cc := ALU(R_{\text{src1}}, R_{\text{src2}}, opc)$	(step 1)
if cc **then** $PC := ALU(PC, offset, add)$ **else** skip **fi**	(step 2).

Step 1 can now be split into three new steps, as can be done with step 2. We then obtain the execution scheme as shown in Table 5.3. Steps 1a, 1b, and 1c as shown in this table are similar to the three steps in Table 5.2, as are steps 2a, 2b, and 2c. In terms of hardware it is possible to *share* pieces of datapath between the various instructions.

Parallel execution of instructions or steps in an instruction execution is a method to reduce the execution time of instructions. However, for parallel execution it is often

necessary to have separate pieces of datapath. In the jump-instruction in Table 5.3, for example, we could want to execute the two ALU-functions in parallel, when the jump is taken. In hardware this is only possible when there are two ALU circuits instead of one. This will reduce the execution time, but it will cost area.

Instructions are executed *sequentially* when an instruction is fetched only after the execution of the previous instruction has finished. When all instructions are executed sequentially, we speak of *sequential* instruction execution. Sequential execution allows for the reuse of pieces of hardware in the datapath for each step. In the synchronous 80C51 all instructions are executed sequentially. Each step in the instruction execution uses the bus IB for communication between registers. This makes a compact implementation of the datapath possible, as we have seen in Chapter 3.

Building an asynchronous datapath is not necessarily much different from building a synchronous datapath. We still need the registers, the communication paths, and the arithmetic circuitry to perform the operations. As in the synchronous case, we still have freedom in how to establish the communication between registers, as we shall see in the next section about datapath design.

However, the *control* of an asynchronous circuit is different from the *centralized* control in a synchronous circuit. Synchronous control is global: all registers are clocked by a global clock signal. Asynchronous control using handshake circuits, on the other hand, is *distributed* and makes selective steering of the elements of the datapath possible.

In Chapter 3 we have seen that the execution schemes of the synchronous 80C51 CPU show many *redundant actions*. The instructions show that there is a wide variety in the number of *necessary*, i.e. *non-redundant* actions that have to take place during execution. Take three instructions in the synchronous 80C51 as example. Their non-redundant actions are shown in Table 5.4. This table shows three instructions, that take 4, 8, and 10 actions to complete, taking the datapath of the synchronous implementation as starting point (Figure 3.3). The synchronous control makes *worst-case* assumptions: take the instruction that takes the most steps to complete, and take this number for *all* instruction executions to complete. In case of the three instructions in Table 5.4 we would let all instructions take 10 steps to complete. For the complete 80C51 instruction set we then obtain a scheme as shown in Table 3.2. This approach keeps the control simple, which is beneficial for the area of the circuit. The resulting redundant actions are not good for execution time and energy dissipation.

A distributed control structure can filter the redundant actions and control the data-

`MOV A,#data`	`ADD A,#data`	`ADD A,@Ri`
ROM → IB	ROM → IB	ROM → IB
IB → IR	IB → IR	IB → IR
ROM → IB	ACC → IB	ACC → IB
IB → ACC	IB → T2	IB → T2
	ROM → IB	RAM → IB
	IB → T1	IB → RAR
	T1+T2 → IB	RAM → IB
	IB → ACC	IB → T1
		T1+T2 → IB
		IB → ACC

Table 5.4: Non-redundant actions for three 80C51 instructions, taking the synchronous datapath as starting point. Instructions `MOV A,#data`, `ADD A,#data`, and `ADD A,@Ri` need 4, 8, and 10 communications in the datapath, respectively.

path in such a way that only the *non-redundant* actions are performed. In case of the instructions as shown in Table 5.4, the three instructions would take 4, 8, and 10 actions respectively, instead of 10 for each instruction. This saves energy and reduces the execution time, as instructions are executed with *average* speed and power. The drawback is that the control is more complicated than the synchronous control, which is disadvantageous for the area of the circuit.

The synchronous 80C51 CPU implementation implements sequential execution of instructions, making the area for both the datapath and the control small. We wish to make an asynchronous version of the 80C51 microcontroller by VLSI-programming in Tangram, and aim to save power where possible. On the other hand, we also wish to keep the area of the circuit small. Therefore we decide to assume sequential execution of the instructions, making it possible to reuse the pieces of the datapath.

Sequential execution of an instruction can be expressed in Tangram by

```
FetchOp() ; Execute()
```

Fetching an instruction is straightforward, using the handshake model of the memories as introduced in the previous chapter: sending an address (the program counter `PC`) to the program memory and waiting for the data to arrive. The program counter

value PC can be incremented in parallel with the arrival of the data (cf. the statement $PC := PC + c$ in Table 5.1):

```
    ROMaddr!PC
; ROMdata?ir || IncPc()
```

For the Tangram procedure `Execute()` we have various possibilities for implementation. We make a distinction between the *datapath* and the *control* of the CPU, as we have done for handshake circuits in Figure 2.7.

5.2 Datapath

The datapath of a processor contains all registers, communication paths between them, and arithmetic circuits. It constitutes the part of the processor where the actual computations take place and where the data is stored and moved. In the 80C51 the variety of addressing modes and the non-uniform address space demand a wide flexibility in data traffic, as it is possible to move data from any register to any other. To implement this flexibility of data traffic there are two "extremes" in the design spectrum. First we can choose to introduce channels between any two registers; we then implement so-called *point-to-point* communication. The other extreme minimizes the number of communication paths by introducing the notion of a *bus*. In this datapath, communication between two registers always takes place in two steps via the bus: first the source register is copied to the bus; then the value of the bus is transferred to the destination register.

5.2.1 Point-to-point communication

Registers in a microcontroller CPU can be implemented with variables in Tangram. A variable is implemented by a component VAR in the handshake circuit. Each variable has one or more read ports but exactly one write port. In the single-rail implementation of a datapath, the number of read ports of a variable is unbounded; no extra de-multiplexer is needed [30]. However, this is not the case for the write-port; all paths to the variable are multiplexed to the write port. The number of inputs to this multiplexer is equal to the number of textual assignments to the variable in the Tangram text. For example, if the Tangram program contains two assignments to a variable x:

```
  ...
; x:=y
; ...
; x:=z
; ...
```

then a 2-way multiplexer on the write port of x is introduced, as shown in Figure 5.1.

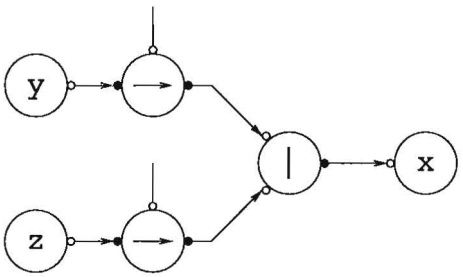


Figure 5.1: A multiplexer provides two alternative access paths to variable x.

When starting the design of a VLSI-program for a microprocessor's datapath, point-to-point communication is quite an intuitive thing to do. Suppose we have n registers, x1 to xn, and we wish to establish a communication path between all pairs of registers apart from auto-assignments (i.e. communication from xi to xi). This can be described in Tangram by

```
& x1x2  : proc() . x1:=x2
& ...
& x1xn  : proc() . x1:=xn
& ...
& xnx1  : proc() . xn:=x1
& ...
& xnxn1 : proc() . xn:=xn1
```

In the corresponding handshake circuit there are $n(n-1)$ communication paths between registers (i.e. there are $n(n-1)$ transferrers); each register can be assigned values from $n-1$ source registers. In the handshake circuit this introduces a multiplexer with $n-1$ inputs on the write port of each variable, as shown in Figure 5.2.

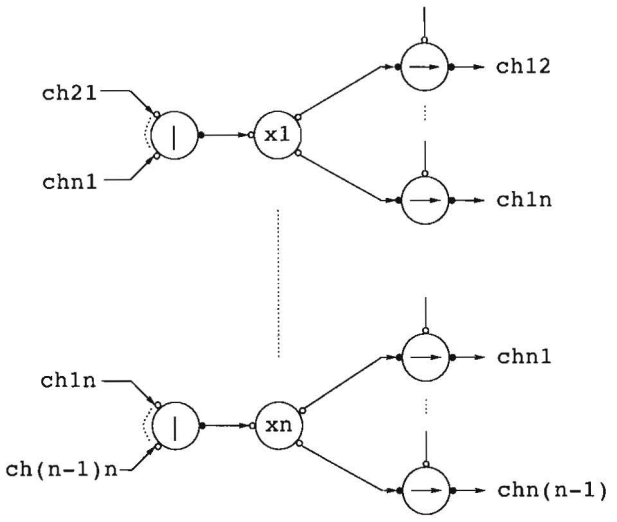


Figure 5.2: Datapath with point-to-point structure.

The introduction of multiplexers in the datapath is implicit in the Tangram text. For dense communication networks the multiplexing can have a considerable impact on the area, execution time, and energy dissipation of the resulting circuit. Therefore it can be worthwhile to reduce the number of multiplexers.

5.2.2 Bus structure

To reduce the number of multiplexers we split each communication $x_i := x_j$ into two parts: first the value of x_j is assigned to a new variable `bus`; then the value in `bus` is assigned to x_i. Thus, in the Tangram text for the point-to-point structure each occurrence of

```
xi:=xj
```

is replaced by

```
bus:=xj ; xi:=bus        .
```

In this way we obtain the Tangram text

```
& x1x2  : proc() . bus:=x2  ; x1:=bus
& ...
& x1xn  : proc() . bus:=xn  ; x1:=bus
& x2x1  : proc() . bus:=x1  ; x2:=bus
& x2x3  : proc() . bus:=x3  ; x2:=bus
& ...
& x2xn  : proc() . bus:=xn  ; x2:=bus
& ...
& xnx1  : proc() . bus:=x1  ; xn:=bus
& ...
& xnxn1 : proc() . bus:=xn1 ; xn:=bus
```

In order to avoid unnecessary multiplexing on write ports of variables we can share common assignments in procedures. The resulting handshake circuit is shown in Figure 5.3.

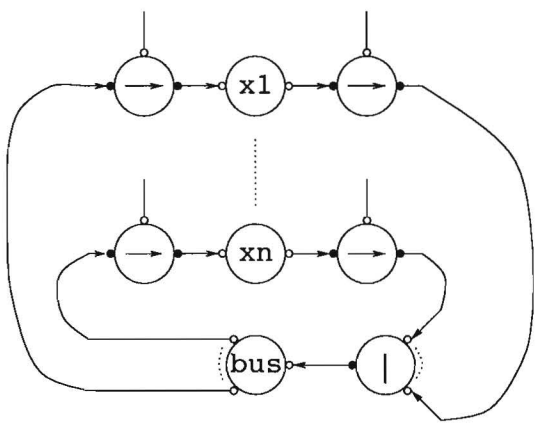


Figure 5.3: Datapath with bus structure.

This datapath is cheaper in area, in that it implements $n + 1$ variables, only one n-input multiplexer and $2n$ transferrers (i.e. communication paths). As described in [30] a multiplexer consists of a control part and a data part. It is beneficial to replace a tree of binary multiplexers by one multi-input multiplexer. Extending the number of inputs of a multiplexer involves extending the data part while the control part stays more or less the same. Therefore extending a multiplexer with an extra input is cheaper than introducing a new multiplexer.

	Point-to-Point	Bus
VAR (Variables)	n	$n+1$
MUX (Multiplexers)	$n_{(n-1)}$-input	1_n-input
TRF (Transferrers)	$n^2 - n$	$2n$
SEQ (Sequencers)	-	n^2
MIX (Mixers)	-	$2n_{(n-1)}$-input

Table 5.5: Number of handshake components for point-to-point network and bus-structure with n registers, assuming a full network without auto-assignments.

For a complete network, the costs of the point-to-point network and of the bus-implementation in terms of handshake components are compared in Table 5.5. The bus-structure is the cheaper in number of handshake components for the *datapath*, for we have only one multiplexer in front of `bus`. The drawback is that the *control* of the datapath becomes more complex. In the case of point-to-point communication only one transferrer per assignment has to be steered, whereas using the bus-structure two transferrers per assignment are controlled, by a sequencer. For a full network, assignments `bus:=xi` and `xi:=bus` occur more than once and therefore extra mixers in the control are introduced.

This sequencer-mixer control structure maps the $n(n-1)$ steering channels (i.e. the number of communication paths) to the $2n$ steering channels of the transferrers in the bus-network. Each sequencer has two active ports, and therefore the mixers have to combine $2n(n-1)$ handshake channels into $2n$ channels; each mixer thus has $n-1$ inputs. The control overhead for the 3-variable bus-network is shown in Figure 5.4. In this figure channels `cij` establish communication from register `xi` to `xj`. Similarly, channels `cib` steer the transferrer from `xi` to `bus`; channels `cbj` steer the communication from `bus` to `xj`.

It is interesting to see where the overhead in the control for the bus-structure starts to dominate the gain in the area for the datapath. For a full network of 8-bit wide variables, without auto-assignments, this is shown in Figure 5.5. For these networks, a point-to-point implementation is smaller for fewer than four variables; when more than four variables are implemented, the bus-network is smaller.

For various numbers of variables (of various widths) in the circuit we can generate Tangram programs that implement both point-to-point and bus networks. To compare them, we calculate the difference in transistor count between the two. The

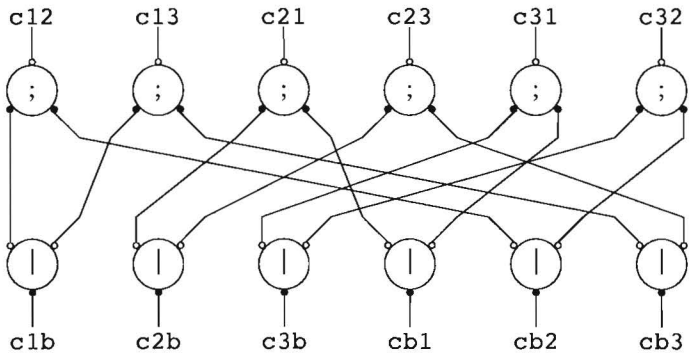


Figure 5.4: Control overhead for bus-structure.

results are shown in Figure 5.6. There is only a small area where point-to-point is smaller than the bus, viz. the light-gray area in the left-bottom corner in the figure. As soon as the number of variables and their width increase the bus-network is smaller than point-to-point.

For execution time and energy dissipation we can simulate the Tangram designs at handshake level. It appears that, disregarding the width and number of variables, energy dissipation of the bus-structure is about twice as high as the dissipation of the point-to-point network. This is explained by the fact that for each communication we need two assignments instead of one. For the complete network, one of these assignments goes through a multiplexer. For execution time we see a similar result; the bus is roughly twice as slow as the point-to-point communication.

In practice one hardly ever encounters a microprocessor datapath in which the graph of communication paths is full. It depends on the "fullness" of this graph whether the gain in area in the datapath will compensate for the overhead in control.

5.2.3 An 80C51 CPU datapath

Having seen two possibilities for the design of a datapath we consider the datapath of the 80C51. With its non-uniform register structure we can nicely exploit the design space and see what is the best solution in terms of area, execution time, and energy dissipation.

Many of the 80C51's registers are Special Function Registers. Therefore, we first

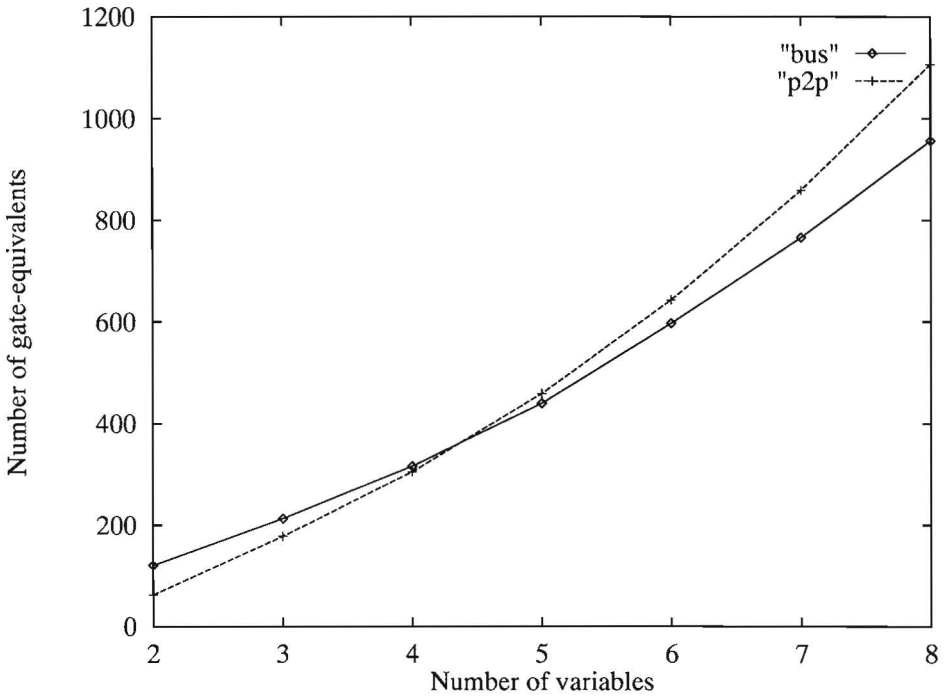


Figure 5.5: Comparison in area (in number of gate-equivalents) between a point-to-point network and a bus-network for 8-bit wide variables. The numbers are for a 0.5μ generic cell-library. The bus-network is smaller for networks with more than 4 variables.

discuss the implementation of the SFR space. The SFRs all have addresses, and they can be accessed by instructions by means of direct addressing. The address space of the SFRs is shown in Figure 3.2, and one possibility is to declare a *register file* for them. Addressing of a SFR is then straightforward: we access the register file by using the address of the SFR. It is also possible not to use a register file, but to declare the registers as separate variables. Decoding the address of the SFR has then to be programmed explicitly (by means of a Tangram `case`-statement). Adding new SFRs to a microcontroller is straightforward in case of a register file: the register is already present and has the appropriate address. In case of separate variables, we must add a line to the `case`-statement that takes care of the address decoding.

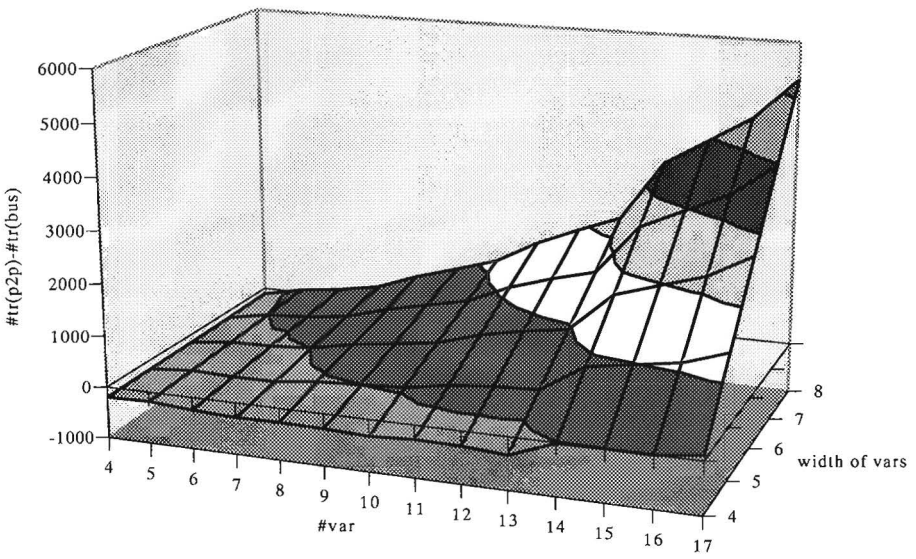


Figure 5.6: Area comparison between point-to-point and bus.

In the definition of the standard 80C51, the SFR memory map is sparse: only 21 of the 128 addresses are in use for SFRs [4]. Declaring a register file for the SFRs implies that the majority of the register space is not used. In other words, declaring separate variables for the registers and dealing with the decoding of the addresses explicitly, will result in a smaller solution.

The datapaths shown in the previous sections can all just copy values from one variable into another. Of course this is not the total picture of a useful datapath; in fact we left all combinatorics out. For the 80C51 we mimic the synchronous architecture and introduce an ALU with associated input registers `T1` and `T2`. Operands to the ALU are first assigned to these registers, after which the ALU can perform an operation. The result is written back to the destination register. Registers `T1` and `T2` implement the registers X and Y as used in Tables 5.2 and 5.3.

In case of a point-to-point datapath, there have to be communication paths between any register to `T1` and `T2`, for any register can contain a source operand. Furthermore, the result of the ALU can be written back to any register (all registers can be destination). In the datapath this results in large multiplexers on the write ports of `T1` and `T2`, and extra inputs on the write-port-multiplexers of the destination

registers.

In Tangram this is easy to implement. Suppose we have instruction `ADD A,R5` which adds the value of register `R5` in a register bank to the accumulator. When we implement a point-to-point datapath we simply write

```
  T1 := R5
; T2 := A
; A  := T1 + T2
```

Because the communication paths from `R5` to `T1`, and form `A` to `T2` are separate, the first two communications can be done in parallel:

```
  T1 := R5 || T2 := A
; A  := T1+T2
```

which reduces the execution time.

When we adopt the bus-structure in the datapath, we see that each communication between registers passes through variable `bus`. Therefore this variable is an obvious place to read values into variables `T1` and `T2`. The result of the ALU can be communicated to `bus`, which then has the same function as register Z in Tables 5.2 and 5.3. We thus obtain the datapath as shown in Figure 5.7.

How can we implement instruction `ADD A,R5` in Tangram in such a way that we obtain this datapath? Using the bus will take more steps in the program execution than in the point-to-point datapath:

```
  bus := A
; T1  := bus
; bus := R5
; T2  := bus
; bus := T1 + T2
; A   := bus
```

The datapath with bus-structure is smaller, but the penalty is clear from this example: it takes more steps to execute the instruction and the control is more complicated, as there are more semicolons in the program text. There are no parallel paths anymore from the source registers to `T1` and `T2`, and therefore the two com-

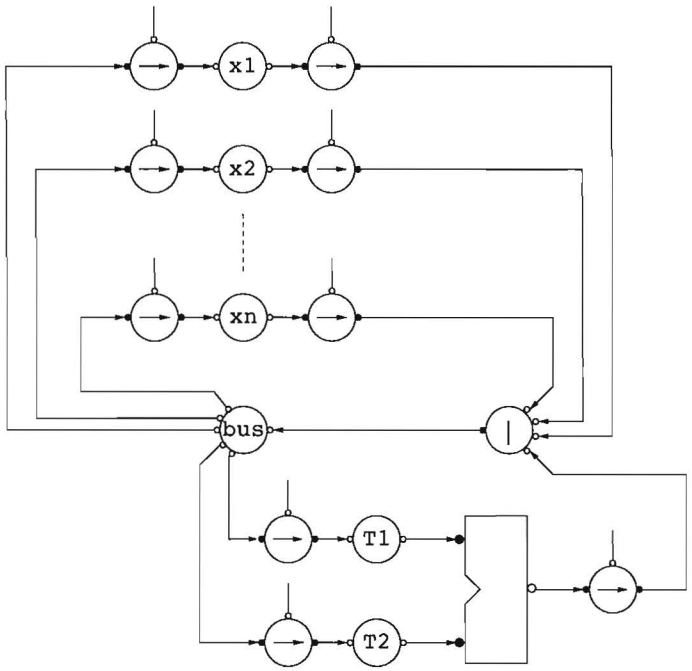


Figure 5.7: Handshake circuit of datapath with bus structure.

munications have to be done in sequential order, resulting in a longer execution time.

For the 80C51 CPU we have implemented the two datapaths, each using the same control structure. The results are shown in Table 5.6. The bus-datapath turns out to be smaller than the point-to-point datapath, but it is also a lot slower and less energy-efficient.

We wish to combine the advantages of both the bus scheme (small area), and the point-to-point scheme (low energy dissipation, low execution time). In other words, we want direct communication paths that are used frequently, and a bus-network for communications that are not used frequently. The direct communications paths then *bypass* the bus and are therefore called *bypasses*. An example of such a hybrid datapath is shown in Figure 5.8: this datapath contains one bypass, from register `x2` to `x1`. Compared to a pure bus-datapath, this handshake circuit contains one more multiplexer (on `x1`) and an extra transferrer.

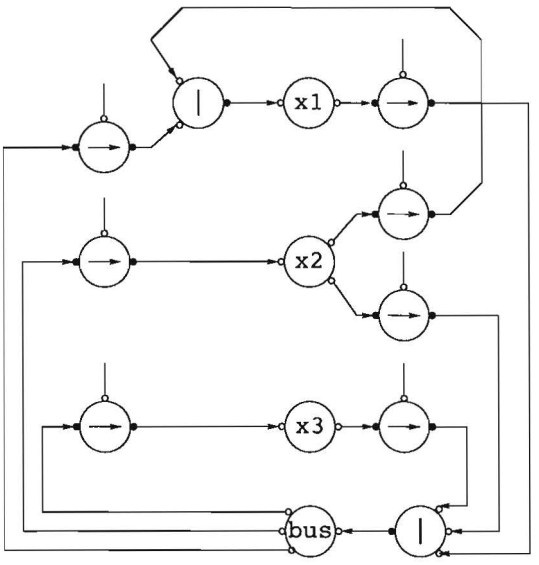


Figure 5.8: Datapath with bus structure and one bypass from `x2` to `x1`.

To investigate what paths in the 80C51 are used frequently, we take a benchmark program, and count the number of uses of eight communication paths. The results are shown in Figure 5.9. Note that the assignment `T2:=SFR` is not a single assignment: it stands for all assignments from special function registers to register `T2`. In this benchmark, the special function registers are not read frequently into register `T2`.

We have implemented bypasses for the top-4 of frequently-used communications paths. A comparison between point-to-point implementation, a full bus-network, a bus where the program counter is bypassed to the program ROM, and a bus with four bypasses is shown in Table 5.6. The design with the full bus is the smallest, but also the slowest and the least energy-efficient of the four. In fact, in the synchronous implementation there is also a path from the PC to the program ROM that bypasses the bus (Figure 3.3). Introducing a few bypasses results in a circuit that is only marginally larger, but because of the frequency with which the bypasses are used, it also results in the fastest and most energy-efficient circuit of the four.

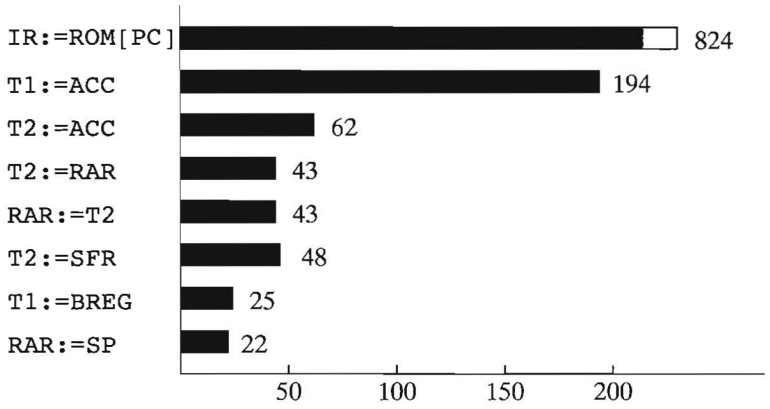


Figure 5.9: The number of times that certain communication paths are used in the 80C51. The benchmark used for these numbers is a program in which almost all 80C51 instructions are executed. It turns out that only a few communication paths are used very frequently.

5.3 Control

When we have designed a datapath for a processor, we see that the main task of a control structure for this datapath is the steering of the various transferrers in the datapath. These transferrers control the communications that take place along the communication paths in the datapath. In this section we consider various control structures, assuming sequential execution of instructions.

The global step of executing an instruction is first fetching it from program memory

	Area (trans.)	Speed (MIPS)	E/instr (nJ)
Point-to-point	31374	1.92	1.28
Full bus-network	27306	1.86	1.81
Bus with PC-bypass	27320	1.95	1.31
Bus with 4 bypasses	27482	2.10	1.06

Table 5.6: Comparison between various datapath implementations.

and then executing it. The global semicolon for the separation of fetch and execute appears in the Tangram fragment

```
FetchOp() ; Execute()
```

Executing an instruction consists of two major steps: decoding and execution of the decoded instruction. As with the design of a datapath we can distinguish two "extremes" in designing a control structure. The first decodes instructions completely, after which the proper actions in the datapath are taken; the other approach decodes instructions while executing them, thereby attempting to share common actions between various instruction executions. It is also possible to implement a combination of these two schemes.

5.3.1 Centralized decoding

The first approach of executing an instruction starts with completely decoding the instruction. Once we have decoded an instruction opcode we know exactly which actions have to be taken (i.e. which statements in the Tangram program have to be executed). Decoding in a handshake circuit can be done by using the `case` component with the instruction opcode as input. This is established by the following Tangram fragment, in which we assume that variable `ir`, the instruction register, contains an 8-bit instruction opcode:

```
case ir
is 0   then instr0()
or 1   then instr1()
or 2   then instr2()
...
or 255 then instr255()
si
```

The various procedures `instri()` contain the statements that control the datapath. Usually, an instruction takes several steps to be executed, for example fetching two operands from registers, then adding the two, and finally storing the result in a register. As we consider only sequential execution in this section, these steps can be described by the Tangram fragment

```
instri : proc() . S0 ; S1 ; S2
```

As we have seen in the previous section, the communications in the statements `S0`, `S1`, `S2`, ... determine the structure of the datapath. We obtain a control structure as depicted in Figure 5.10.

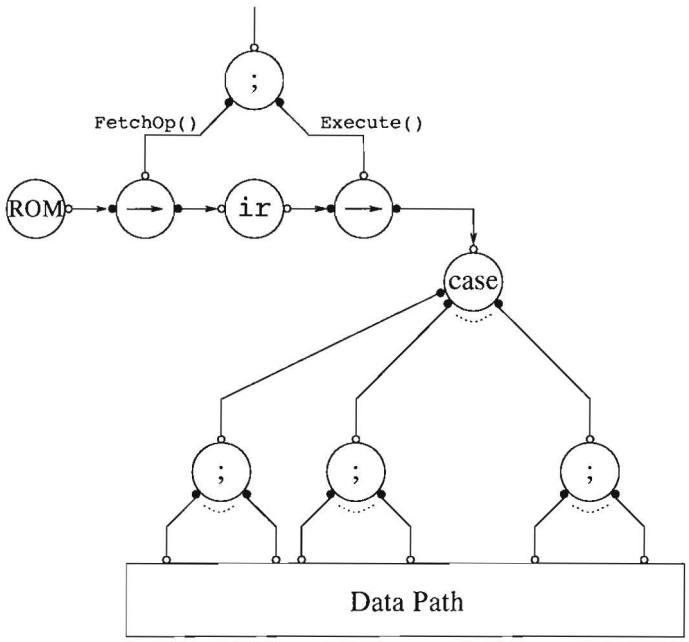


Figure 5.10: Handshake circuit of control structure with centralized decoding.

In this figure we view the program ROM as a handshake component that (after sending an address) delivers an opcode of an instruction. This opcode is copied into instruction register `ir`. In `Execute()` the opcode is input to the central `case` component that first determines which instruction is to be executed. It steers the corresponding multi-sequencer that controls the datapath. Upon completion of the execution of the instruction, the handshake protocol between the various control components is completed. Finally, upon request, the sequencer on top will start fetching the next instruction.

5.3.2 Distributed decoding

The second approach looks for overlap in execution schemes of all instructions, as was demonstrated in Section 5.1. For example, when we have two instructions

```
ADD R3,R1,R2   : R3:=R1+R2
SUB R3,R1,R2   : R3:=R1-R2
```

the corresponding instruction execution schemes might look like

```
ADD :  T1:=R1 ; T2:=R2 ; bus:=T1+T2 ; R3:=bus
SUB :  T1:=R1 ; T2:=R2 ; bus:=T1-T2 ; R3:=bus
```

We see a lot of overlap in instruction execution steps; in fact only the third step (addition or subtraction) differs. Therefore we can use simple decode steps for the first, the second and the last assignment (i.e. we do not have to distinguish between ADD and SUB) and have a more complex decoding for the third step (in which we do have to distinguish between the two).

Following this scheme we first try to fit all instructions into one execution scheme, in such a way that for one stage in this scheme most instructions demand the same action. This keeps the decoding per stage simple. Thus, in Tangram we have for the stages he fragment

```
stage1() ; stage2() ; ... ; stagen()
```

where the stages are programmed as

```
stagei : proc(). case expr0(ir)
                 is 1 then S0
                 or 0 then case expr1(ir)
                           is 1 then S1
                           or 0 then ...
                           si
                 si
```

The corresponding handshake circuit is depicted in Figure 5.11.

Fetching the instruction is done in exactly the same manner as with centralized decoding. The decode step is different: there are various stages, each with its own (small) decoding done by a `case` component. Instruction register `ir` is input to the various `case` components and the complete structure is controlled by a multi-sequencer. After completion of the instruction execution the handshake protocol between the various control components is finished and the next instruction can be fetched.

In the scheme with distributed decoding, the actual decoding for a stage takes place in two steps:

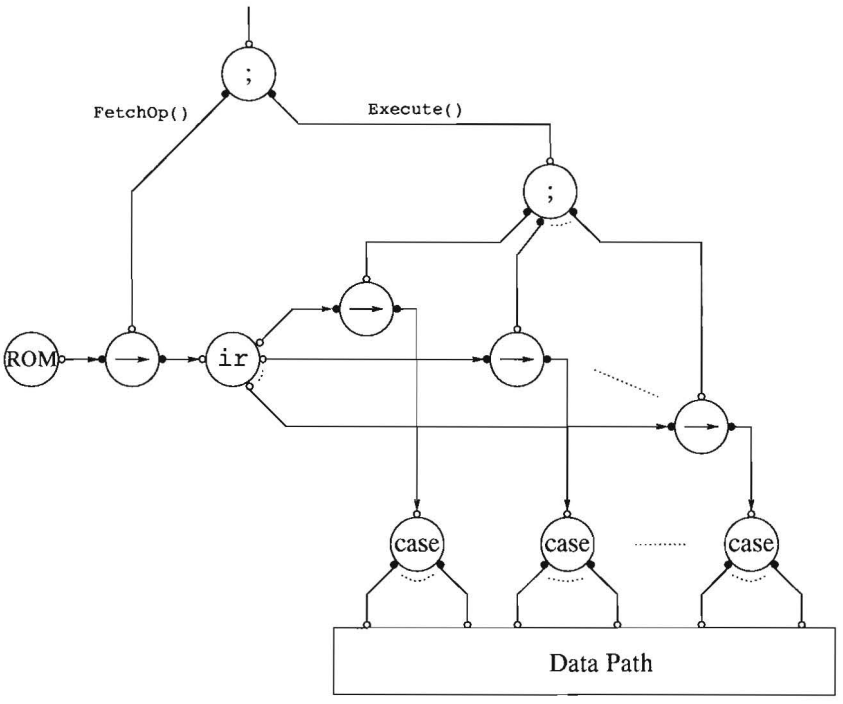


Figure 5.11: Handshake circuit of control structure with distributed decoding.

- first, the expressions are evaluated;
- then their value is inspected in the `case`-statement for that stage.

This introduces an overhead in the area of the handshake circuit. The expressions contain the information what transferrers in the datapath for a stage have to be steered. The `case`-statement decodes this binary value: either the expression is true, or false. If it is true then the associated transferrer is activated, otherwise not. In other words, the `case`-statement translates a boolean value into a handshake. In the handshake circuit, one would want to "skip" the `case` component, and connect the result of an expression directly to the passive handshake channel of a transferrer. Though this is in principle possible at the handshake level, it is not possible to express this construction in Tangram. Therefore, an automatic translation from Tangram into such a handshake circuit is not possible. Tangram demands to use the `case`-statement in this case, resulting in a larger, slower and less energy-efficient

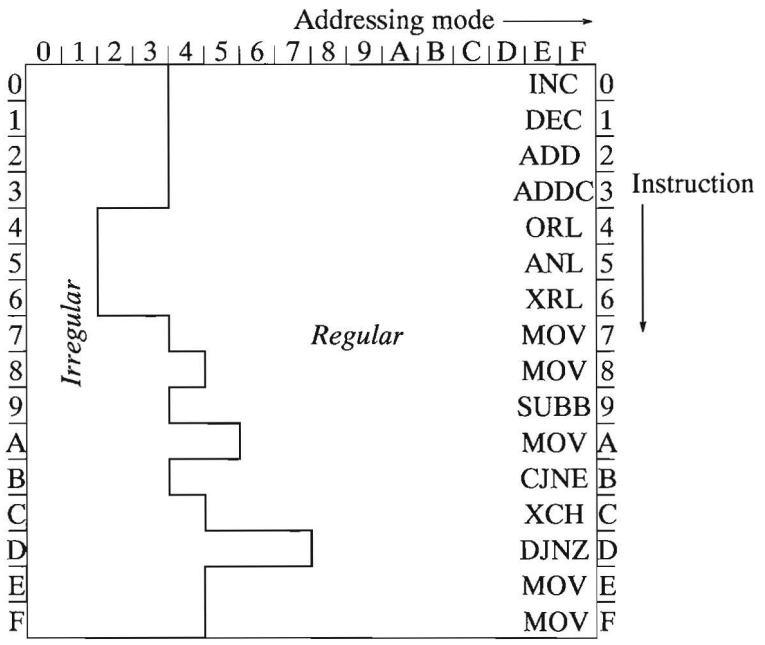


Table 5.7: Regular (Right) and Irregular (Left) part of the 80C51 instruction set.

circuit than necessary.

5.3.3 An 80C51 control structure

For the 80C51 we can apply all of the above schemes. In fact, it turns out that the cheapest solution is the hybrid scheme of centralized and distributed decoding. To this end, we split the instruction set into two parts: *regular()* and *irregular()*. To see where the split can be made, we go back to the table of the instruction set (Appendix B). The regularity shows itself best in the *rows* on the instruction set table. For each row, only one type of instruction occupies the largest part. For example, we see the INC instruction in row 0, the DEC instruction in row 1, etc. The difference per row is in the various addressing modes that are encoded in columns: columns 8 to F access registers, columns 6 and 7 access registers indirectly, column 5 implements direct addressing, etc. Taking all the same instructions per row together, we obtain the partition as shown in Table 5.7.

The instructions located at the right of the split belong to the *regular* part of the instruction set; the instructions at the left constitute the *irregular* part. The regular part can be implemented by the Tangram text

```
ReadOperands() ; Operation() ; WriteOperands()
```

where `ReadOperands()` and `WriteOperands()` decode in columns and procedure `Operation()` decodes in rows. In this fragment we have a three-way sequencer on top and (small) decoding steps at the leaves of the sequencer. For example, `operation()` can be implemented by

```
case row
is 0 then INC(T2)
or 1 then DEC(T2)
or 2 then ADD(T1,T2)
...
si
```

One can separate regular and irregular in various ways. In this implementation we have chosen to draw the separation such that the regular part is as large as possible to exploit the regularity in instruction execution as much as possible. As can be seen from Table 5.7 the separation is not a straight line and one could investigate whether moving the separation line has an impact on the performance and area of the resulting circuit.

A boolean function `f`, expressed in the bits of the instruction opcode, determines whether an instruction opcode belongs to the regular or the irregular part of the instruction set:

```
if f(ir)
then regular()
else irregular()
fi
```

The resulting handshake circuit for the control of the 80C51 is shown in Figure 5.12.

The irregular part of the instruction set contains clusters of similar instructions. For example, all `AJMP` and `ACALL` instruction are clustered in column 1, the jump instructions appear in column 0 (rows 1 to 7), and the rotate instructions are in column 3 (rows 0 to 3). Each of these clusters of instructions shows *regularity* in the execution scheme. For example, the jump-instructions follow the scheme

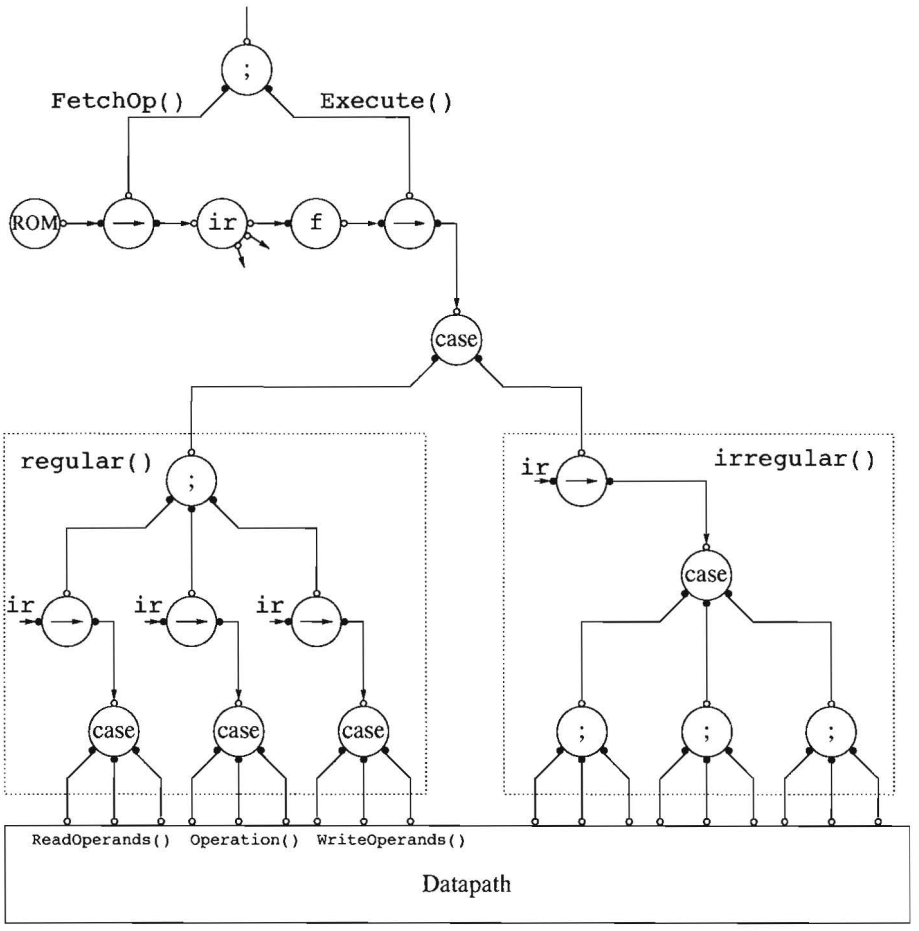


Figure 5.12: Handshake circuit for 80C51 control: hybrid scheme with both centralized and distributed decoding.

in Table 5.3. The jump-instructions differ in their calculation of the condition to jump; the second step (adjusting the PC if necessary) is for all jump-instructions the same. The rotate-instructions follow the scheme

```
  bus:=ACC
; T1 :=bus
; bus:=ROTATE(T1,CARRY)
; ACC:=bus
```

when using a bus-datapath. The calculation of the ROTATE function is different for rotate instructions; the other communications are identical.

In other words, the *irregular* part of the instruction set shows clusters of *regular* instructions. For procedure `irregular()` we can therefore first decode into these clusters, and then exploit the regularity in the same way as was done for the first instruction-decode step. In the handshake circuit of Figure 5.12 the `case`-component on top of `irregular()` decodes into these clusters and the sequencers beneath control the regular parts in the clusters.

Note that `regular()` is implemented in such a way that the instructions are executed in a minimal number of steps. Therefore this implementation filters the redundant actions as implemented by the synchronous 80C51. The distributed control of handshake circuits makes such an implementation an attractive solution in terms of energy dissipation. Furthermore, the *asynchronous* character of the control saves energy in the control as well, because only one path from the root of the control tree to a leaf (i.e. a transferrer in the datapath) is active during the execution of an instruction.

5.4 Local optimizations

The previous sections describe global approaches to the design of datapath and control of a sequential architecture. Zooming in on smaller pieces of Tangram text one can often identify local constructs that can be optimized. Some of these optimizations are discussed in this section.

Some straightforward optimizations that apply to any kind of circuit are the ones mentioned in Chapter 2: *sharing* of statements in the datapath and sharing of control structures. Sharing of common statements in the datapath results in moving expensive multiplexers in the datapath to cheaper mixers in the control. This not only saves area, but is also better for energy dissipation. Sharing in the control

saves area, but it often results in a circuit that is slightly slower and less energy-efficient.

The optimizations in this section go further than sharing of identical structures; the goal is to optimize by sharing *nearly* identical structures in the datapath. For the control it is often possible to rewrite the program text in such a way that the compiled circuit is cheaper in some sense (area, energy, or execution time). We first consider the datapath and then look at the control.

5.4.1 Datapath

We distinguish two kinds of optimizations in the datapath: the ones that reduce the execution time and the ones that reduce the area.

Execution time: faster adders

An adder is an example of a combinatoric circuit. For each occurrence of a "+" in the Tangram text a separate adder is implemented in the circuit. These adders are implemented using *ripple-carry* adder circuits. For large additions this is not the best solution for performance; it takes quite some time for the carry to ripple through all full-adder cells. Other schemes result in faster, but less area-efficient and energy-efficient circuits. An example of such a scheme is the *carry-select* adder [20, 15], which is shown schematically in Figure 5.13. It splits the addition into separate parts, for example the low-order half and the high-order half. Separate additions are calculated for both parts. For the high-order part in fact two additions are done, one assuming the carry-out of the low-order part is 0 and one addition using a carry-out of 1. All three additions are done parallel and can be implemented using ordinary ripple-carry addition or any other scheme. The carry-out of the low-order part then selects the correct value of the high-order part. This can be expressed in Tangram by the function shown in Figure 5.14 [30].

This technique is easily generalized to a carry-select adder that splits the arguments into more than two parts. The carry-select adder saves time, because the additions are done in parallel. On the other hand it is larger and less energy-efficient because for the high-order part two additions are done instead of one.

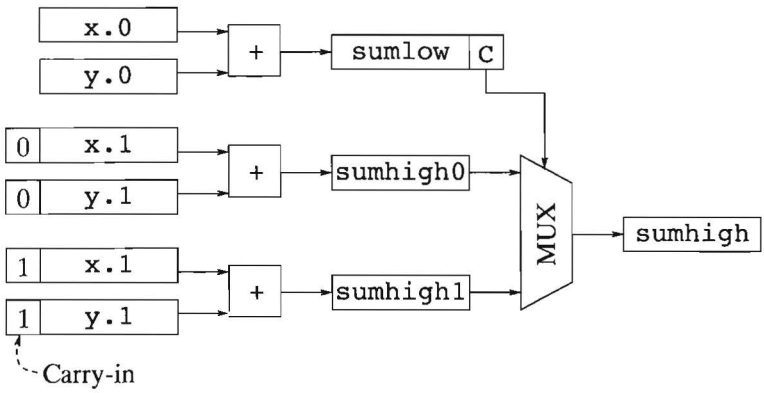


Figure 5.13: Carry-select adder scheme. Three additions are performed in parallel: one for the low-order sum, and two for the high-order sum: one assuming the carry-out of the low-order addition is 0, and one assuming that the carry-out is 1. The carry-out of the low-order addition, C, determines which high-order sum is chosen for the result (Note: this figure assumes the Tangram notation, in which the least significant bit is written on the left hand side).

Reducing area: sharing functions

Suppose we have three functions f, g, and h, that all take variables x and y as input, and produce output in variable z. Examples of such functions are addition, subtraction, and boolean bit-wise operations like logical AND and logical OR. A straightforward implementation of these function is represented by the following three Tangram procedures:

```
  calcf : proc() . z:=f(x,y)
& calcg : proc() . z:=g(x,y)
& calch : proc() . z:=h(x,y)
```

The corresponding handshake circuit is shown in Figure 5.15.

Suppose we can construct a function F that generalizes f, g, and h: this function uses x and y, and an opcode opc as input, and produces output in z. The opcode determines whether the result of F will be the result of f, g, or h. The corresponding handshake circuit is shown in Figure 5.16. This handshake circuit saves on the

```
/* carry-select implementation of z := x+y */
<<z,cout>> :=
  begin
    sumlow   = val (x.0 + y.0) cast <<int8,bool>>
  & sumhigh0 = val (x.1 + y.1)
  & sumhigh1 = val ( <<1,x.1>> + <<1,y.1>> )
                   cast <<bool,int8>>.1
  & sgn      = val sumlow.1
  & sumhigh  = val MUX(sumhigh0,sumhigh1,sgn)
  | <<sumlow.0,sumhigh >>
  end
end
```

Figure 5.14: Sketch of a Tangram program for a 16-bit carry-select addition, assuming a multiplexer function MUX.

multiplexer on the write port on z, and saves on the capacitive load on the variables x and y by reducing their number of read ports. The corresponding Tangram program fragment is

```
  F   : func(x,y : T1 & opc : T2) : T1 .
        /* function text */
& alu : proc() . z:=F(x,y,opc)
```

The circuit for function F implements the ALU (Arithmetic Logic Unit) function in Table 5.2. To generate the circuitry for F in the circuit only once, it is essential that the function is invoked only once in the program text. This is possible when the function uses only one source per input and one destination per output, as expressed by the *rule of invocation*:

Rule of invocation: Reduce the number of invocations of functions to one.

In a CPU that executes instructions, it is best to express the opcode for the function, opc, in terms of bits of the instruction opcode. This saves the introduction of an extra variable for opc and an extra assignment to this variable.

We consider a few examples of functions F that combine increment and decrement, combine addition and subtraction, and combine bitwise boolean operations.

Example: Increment and decrement

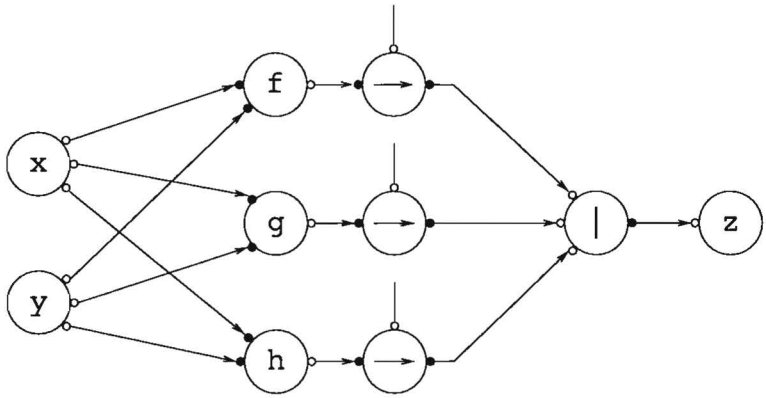


Figure 5.15: Three functions take input `x` and `y` and produce output in `z`. The handshake circuit contains a 3-input multiplexer in the datapath.

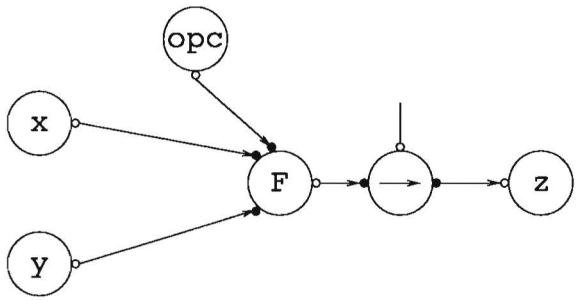


Figure 5.16: Function `F` combines the function `f`, `g`, and `h`. `F` takes an opcode `opc` to determine the result.

It is possible to take the increment and the decrement functions together into one new function that uses an extra parameter. This parameter determines the function to be calculated. Using that

$$x - 1 = x + (-1)$$

one can write a function that optionally negates "1" and performs the addition. This saves one subtracter. The function that implements this technique in shown in Figure 5.17. Negative values are represented in Tangram using 2's complement notation: negating the value of a register is done by inverting all bits and adding 1 to the result.

```
incdec = func( b : bool  &  x : int8 ) : int9 .
         begin /* if b=0 then x+1 else x-1 fi */
               w = val <<true,b,b,b,b,b,b,b>> cast int8
         |     << x + w >> cast int9
         end
```

Figure 5.17: Function for increment and decrement.

Example: Addition and subtraction

In a more general sense we can write a function in Tangram that combines subtraction and addition using only one adder. This function is based on the following observation.

$$x - y = x + (-y)$$

The function optionally negates the value of `y` and performs the addition, using only one adder. The Tangram text is shown in Figure 5.18. In the `addsub` function in Figure 5.18 the inversion of all bits is done by function `inv`; adding 1 can be done by function `f1` by invoking this function with value 1 for parameter `c` that represents the carry-in. The parameters used in this function can be derived from the instruction opcode in the 80C51; in this way it is possible to invoke the function only once in the program text.

Example: Boolean operations

Another example merges bitwise logical functions into one new parameterized function. The functions to combine are logical AND (*), OR (+), and XOR (#) (exclusive OR). For this, we observe that

```
addsub :
   func( f : bool  &  c : bool
       & x : int8  &  y : int8
       ) : int9 .
   begin /* case <<f,c>>
            is <<0,0>> then x+y
            or <<0,1>> then x+y+1
            or <<1,0>> then x-y
            or <<1,1>> then x-y-1
            si
         */
         bib = type <<bool,int8,bool>>
   &     inv = func( f : bool  &  y : byte ) : int8 .
               << f#y.0 , f#y.1 , f#y.2 , f#y.3 ,
                  f#y.4 , f#y.5 , f#y.6 , f#y.7
               >>
   &     f1  = func( c : bool
                   &  x : int8  &  y : int8
                   ) : bib .
               (( <<c,x>> cast int9
                + <<c,y>> cast int9
               ))  cast bib
   &     v   = val f1( f#c , x , inv(f,y cast byte) )
   |     <<v.1 , v.2 # f>> cast int9
   end
```

Figure 5.18: Tangram function for addition and subtraction, with or without carry.

$$x\#y = (x + y) * -(x * y)$$

This formula expresses the exclusive OR in terms of the logical AND and logical OR. By parameterizing this formula to

$$E = (p + (x + y)) * -(q * x * y)$$

we obtain the following table:

p	q	E
0	0	$x+y$
0	1	$x\#y$
1	0	**true**
1	1	$-(x*y)$

In the 80C51 we can use this function for taking bitwise logical functions (in the ANL, ORL, and XRL instructions) together. The parameters (p and q in the previous formula) can be expressed in terms of the bits of the instruction opcode, for the logical instructions appear in separate rows in the instruction set table (Appendix B).

When expressing the three bitwise operations AND, ORL, and XRL separately, the resulting handshake circuit will contain a 3-input multiplexer on the result register. This multiplexer is saved by using formula E as introduced above. On the other hand, formula E uses three AND-operations, two OR-operations, and one inverter per bit, instead of one AND, one OR, and one XOR-operation. Furthermore, formula E has the disadvantage that when $(p,q) = (1,1)$, $-(x*y)$ is calculated instead of $(x*y)$. Therefore we need another function to invert all the bits. Suppose we have this function already available (for example function `inv` in the `addsub` function in Figure 5.18!), then we obtain Table 5.8 from handshake simulation. This table shows that the combined function ("blu") is smaller but also slower and more energy consuming than the three separate functions.

5.4.2 Control

The second class of local optimizations concentrates on the control circuitry. Here we show two examples.

Multiply and Divide

This example involves the 80C51 instructions `DIV` and `MUL`. Both use the classic shift-and-add approach as described by Koren [20], Hennessy and Patterson [15], and other textbooks on computer arithmetic. The algorithms for division and multiplication have similarities and they can be combined into one algorithm as expressed in the Tangram fragment in Figure 5.19.

Width of variables (bits)	Type	Area (transistors)	E/operation (nJ)	T/operation (ns)
4	blu	598	0.35	40
	separate	726	0.15	22
8	blu	1030	0.62	44
	separate	1174	0.25	24
16	blu	1894	1.15	44
	separate	2070	0.45	24
32	blu	3622	2.22	45
	separate	3862	0.85	24

Table 5.8: Implementation of separate boolean functions ("separate") vs one boolean function ("blu") combining these functions, for various widths of the variables. The combined function results in a smaller, but also slower and more energy-consuming circuit. The numbers are obtained from handshake circuit simulation, assuming a 0.8μ generic standard-cell library.

Semicolon-sweeping

The second example is called *semicolon-sweeping*: reduce the number of semicolons in the Tangram text (i.e. sequencers in the control of the handshake circuit) by rewriting pieces of Tangram text. Suppose we have the Tangram fragment

```
case <<b0,b1>>
is <<0,0>> then S0 ; P0 ; S1
or <<0,1>> then S0 ; P1 ; S1
or <<1,0>> then S2 ; P2 ; S3
or <<1,1>> then S2 ; P3 ; S3
si
```

In the 80C51 we encounter such fragments, for example in the decoding of instructions: in the above Tangram `case`-statement, `S0` and `S1` address registers directly, and `S2` and `S3` address registers indirectly; `P0` ... `P3` stand for operations like INC, DEC, etc. A part of the corresponding handshake circuit is shown on the left-hand side of Figure 5.20. This figure shows one half of the complete handshake circuit, only for two alternatives of the `case`-statement; the other half is identical. The handshake circuit contains in total eight sequencers and four mixers, for statements `S0`...`S3` are each invoked twice in the Tangram text.

```
/* A = acc  = <<A.7,...,A.0>> and
   B = breg = <<B.7,...,B.0>> and
   P        = <<  0,...,0  >>
*/

carry:=0 ;
for 8 do if DIV then shiftleft() fi
      ;  if A.0 then P := P + B
                 else if DIV  then P := P - B
                                 else /* MUL */
                                      carry := 0
                                 fi
         fi
      ;  if DIV then A.0 := -carry
                 else /* MUL */
                      shiftright()
                 ;    carry := 0
         fi
      od ;
if DIV
then if (P<0) then P:=P+B fi
fi
```

Figure 5.19: Tangram text for multiply and division, assuming procedures `shiftleft()` and `shiftright()` that shift bit patterns.

By replacing the Tangram fragment by

```
case b0
is 0 then S0
     ;    case b1
          is 0 then P0
          or 1 then P1
          si
     ;    S1
or 1 then S2
     ;    case b1
          is 0 then P2
          or 1 then P3
          si
     ;    S3
si
```

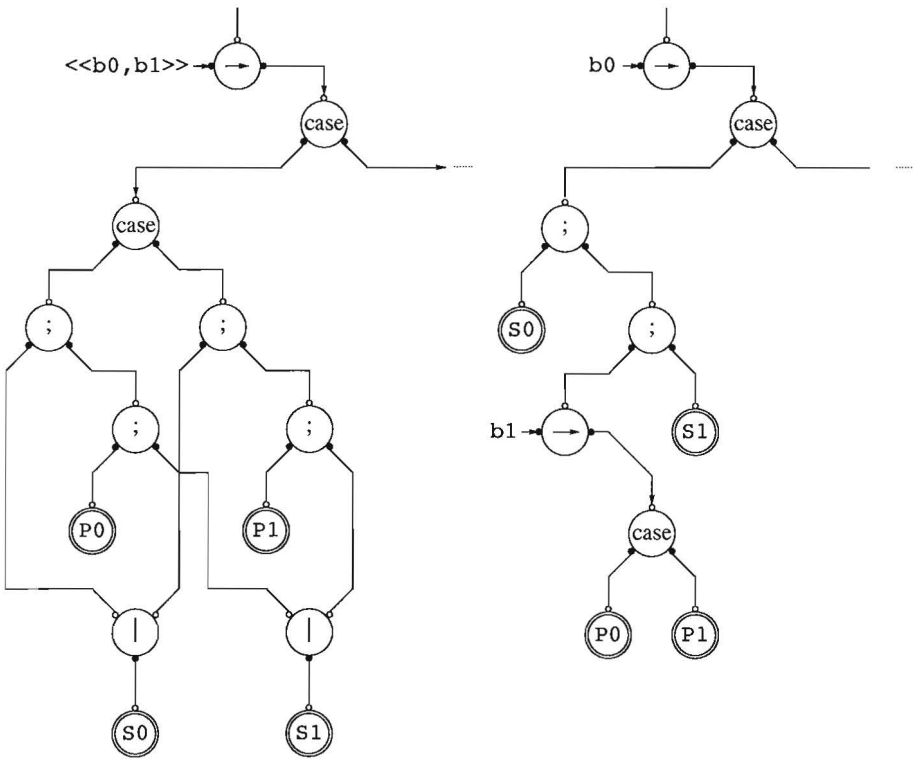


Figure 5.20: Handshake circuits for case-statements: optimizations by rewriting the Tangram program.

we obtain the handshake circuit as shown on the right-hand side of Figure 5.20. This handshake circuit saves four semicolons and the mixers on the invocations of `S0...S3`, because these statements are invoked only once. The `case` components in the two handshake circuits are slightly different; in the left circuit decoding on two booleans is done, whereas in the right circuit decodes on only one boolean, twice. The right-hand side handshake circuit results in a smaller but also faster and less energy-consuming circuit than the circuit on the left.

5.5 Exception-handling

The previous sections describe the part of the CPU that fetches and executes instructions. The microcontroller system also has to handle so-called *exceptions*. An exception can be either

- an interrupt,
- or a special condition.

An *interrupt* is an internal or external event which causes the CPU to postpone execution of the next instruction until some *interrupt service routine* has been executed. In general an internal interrupt can be generated in the CPU itself (for example when an addition of two numbers causes an overflow) or by another block connected to the CPU (for example when a timer has overflowed). An external interrupt is generated by the environment of the microcontroller. In the 80C51 some lines of Port 3 are reserved as external interrupt lines. In all of these cases the CPU adjusts the program counter to point to the first instruction of the interrupt service routine (which is part of the program that runs on the 80C51). After completion of this routine, and when no other interrupt or special condition has occurred, the CPU will fetch and execute the next instruction of the normal program flow.

A *special condition* forces the CPU to stop the execution of the program and turn into some special mode. An example of such a mode is the *reset* mode which forces the microcontroller to initialize the complete 80C51 system and start fetching the first instruction of the program. Other examples of special conditions are the *idle* and *power-down* modes. In the synchronous implementation, in idle mode the clock is disconnected from the CPU, but timers, interrupt controller, and the serial port functions are still clocked. In power-down mode the on-chip oscillator is stopped and all blocks stop executing. The only exit from this mode is a hardware reset; this resets the contents of all registers in the 80C51 while the contents of the internal data RAM is maintained. Table 5.9 shows the exceptions that occur in the 80C51, and describes how the exceptions are dealt with.

The synchronous 80C51 checks for an exception at the end of each machine cycle, but an exception is serviced only after the instruction has been executed. As we have seen in Chapter 3, all instructions except `DIV` and `MUL` execute in one or two machine cycles. The asynchronous CPU does not have the notion of machine cycles. Therefore we choose to check for an exception at the beginning of the execution of each instruction. The frequency at which exceptions are then checked is not exactly the same compared to the synchronous implementation. It is possible

Exception Name	Function	Escape
Interrupt (internal or external)	Postpone execution of the next instruction until an interrupt service routine has been executed	-
Power-on-reset	Resets the circuit when the power is turned on	-
Reset	Resets and initializes the microcontroller while the system is running	-
Idle	The clock is disconnected from the CPU; timers, interrupt controller, and serial port functions are still clocked	interrupt or reset
Power-down	All blocks stop executing	Hardware reset

Table 5.9: Exceptions in the 80C51 microcontroller.

to design the Synchronizer-block in Chapter 4 in such a way that it synchronizes with the CPU at the beginning of each "machine cycle", but this would make the CPU more complicated, because of the extra synchronization.

If neither an interrupt nor a special condition has occurred, the 80C51 can safely fetch the next instruction and execute it. The order in which the exceptions are checked has to do with their priority. A reset replaces the current state of the processor by the initial state, and has therefore the highest priority. Idle mode and power-down mode stop the CPU's activity, but the state of the CPU is kept. An interrupt saves the current state of the processor, executes an interrupt service routine, and restores the saved state. The order of priority is reflected by the nesting of the `if`-statement in the Tangram program for the main loop of `CPU()`:

```
forever
do CheckInterrupt() || CheckSpecialCondition()
;  if -Interrupt * -SpecialCondition
   then FetchOp() ; Execute()
   else if PORline
        then PowerOnReset() /* special condition */
        else if ResetLine
             then Reset()   /* special condition */
             else if IdleBit
```

```
                    then Idle()  /* special condition */
                    else if PowerDownBit
                         then /* special condition */
                              PowerDown()
                         else /* interrupt */
                              JumpToInterruptVector()
                         fi
                    fi
               fi
          fi
     fi
od
```

5.6 Review

This chapter has presented a design space exploration for a sequential handshake 80C51 CPU that was isolated in Chapter 4. It can be split into the datapath and the control.

For the datapath we observed that it is best for power to have a bus-structure for rarely-used communication paths, while having a direct path for the frequently used communications. The bus-structure reduces the area, and the point-to-point communication is better for execution time and energy dissipation when there is not much multiplexing involved. In the 80C51 only a few communication paths are used very frequently, and this makes it possible to combine the advantages of both the bus-structure and the point-to-point communication into one datapath.

For the control we observed that a distributed control structure enables the designer to reduce the number of redundant actions in the execution of instructions. This saves execution time, and energy dissipated in the datapath. Furthermore, an *asynchronous* distributed control makes it possible to save power in the control as well, because only those parts of the control are active that do useful work, at a given point in time.

Chapter 6

Asynchronous 80C51 Peripherals

Thus far we have concentrated on the CPU of the 80C51: the part that fetches and executes instructions of the program. However, there are other blocks, such as the timer block and the interrupt controller, that make the 80C51 a microcontroller: the *peripherals*.

The derivatives of a microcontroller use the same CPU, for the instruction set is fixed. The derivatives differ in the sizes and the implementation of the memories and in the number and functionality of the peripherals. Peripherals make the microcontroller a modular system.

In this chapter we describe an interface between the CPU and the peripherals. This description abstracts from the functionality of a peripheral; it solely describes the communication protocol between CPU and peripheral. The interface implements some constraints that we first identify. At the end of this chapter we discuss the UART as an example of a peripheral for the 80C51 microcontroller.

6.1 Characterization

6.1.1 What is a peripheral?

A peripheral is a piece of hardware that is capable of performing a specialized (and usually small) task. It assists the CPU in doing its job. They communicate when necessary, but run as autonomously as possible. Some peripherals take care of the communication between the CPU and the outside world, the *environment*. A peripheral and the CPU operate concurrently. Section 3.2.3 gives some examples

```
forever
do wait(start_condition)
;  execute_task()
;  set_interrupt()
od
```

Figure 6.1: General Tangram program for the operation of a peripheral.

of peripherals: timers and counters, the interrupt controller, the Input-Output (I/O) peripheral, and the UART.

6.1.2 Peripherals and power

A peripheral is capable of doing a specific task upon request. Its general functionality can be described by the pseudo-Tangram fragment in Figure 6.1.

After enabling, the peripheral performs its task; an interrupt to the CPU is set upon completion. It then starts the cycle again. From the program fragment in Figure 6.1 we observe that the peripheral works *demand-driven*. The CPU sets the start condition, and the peripheral starts executing its task only when this condition is set.

The activity of a peripheral occurs generally

- less frequent than the clock frequency of the CPU;
- not evenly spread in time.

A timer that counts machine cycles in the 80C51, for example, does so only at 1/6th of the clock frequency to the CPU and is therefore less active than the CPU. The interrupt controller, waiting for an external interrupt, is waiting for an event of which it cannot be predicted when it will happen. Even so, it might never happen.

Synchronous implementations of the 80C51 keep the peripherals clocked at the clock frequency to the CPU, or a division of that frequency (for example, a timer that counts machine cycles can be clocked at 1/6th of the clock frequency). All registers in the peripheral are clocked, even when the peripheral is not enabled, in which case energy is dissipated unnecessarily. One can say that the peripherals in the synchronous implementation are *clock-driven* while the nature of their operation is *demand-driven*.

Asynchronous systems are demand-driven by nature; only the piece of circuitry that needs to be active is doing useful work *upon request*, while the inactive parts of the circuit are not dissipating energy. Asynchronous circuits are demand-driven by nature, and therefore an asynchronous design style fits nicely with the demand-driven operation of peripherals.

6.1.3 Constraints on the implementation

Before embarking on design issues of peripherals we first compile some constraints that an implementation must adhere to.

1. To ensure maximum progress of program execution by the CPU, the peripheral should run as *autonomously* as possible. It should synchronize with the CPU only when necessary.
2. In addition to this, a request from the CPU should be granted by the peripheral *instantly*, to enable maximum progress by the CPU. In other words, a CPU that is waiting is not acceptable in the implementation.
3. As some peripherals cater for the communication between the CPU and the outside world they should comply with the environment's protocols. In other words, they must ensure *bit-compatibility* as well as *timing-compatibility* as discussed in Chapter 4. This is of importance when a redesign of an existing microcontroller is made; it must fit in any existing environment that works correctly with the old design. In the asynchronous version, this means that we have to build an interface between an asynchronous system and a synchronous environment. For timing compatibility of external memory access this was shown in Chapter 4. For the UART, the timing-compatibility issue is discussed at the end of this chapter.
4. Power consumption must be kept to a minimum.
5. The microcontroller system should be modular to make it possible to add extra peripherals to extend the system's functionality. To enable testability of the microcontroller system, peripherals as blocks should be made testable.

6.2 Implementation

In this section we specify an interface between the CPU and any peripheral. This interface must satisfy the constraints as described in the previous section.

6.2.1 General architecture

In the 80C51, the CPU and the peripherals communicate with each other using the Special Function Registers (SFRs). These registers are either *control* or *data* registers. For example, for the timer-peripheral we have two control registers: TMOD (the Timer/Counter Mode Control) and TCON (Timer/Counter Control Register). Furthermore, there are the data registers TL0, TL1, TH0, and TH1 that contain the counted values.

Both the CPU and the peripherals are able to access these SFRs; therefore we have to decide where (i.e. in which Tangram procedure) to locate them. Suppose we have a sequential implementation of the peripheral as shown in Figure 6.1. One possibility is to locate the SFRs in the CPU. This makes reading and writing the SFRs by the CPU straightforward, for they are local to the CPU. On the other hand, what happens when the peripheral wants to access an SFR? It then has to issue a communication to the CPU with the proper request. The CPU has to monitor the communication channels between itself and the peripherals regularly, say each instruction execution. This violates constraint 1, viz. maximum progress of the CPU and of the peripherals.

Another possibility is to position the SFRs in the peripheral itself. This makes it possible for the peripheral to access the SFRs at any time, i.e. the peripheral needs not to wait. With some communication channels the CPU can request access to the SFRs for reading and writing, and the initiative for communication lies with the CPU. But when the peripheral is implemented as a sequential process, and busy performing its task, it is not ready for answering any request from the CPU, which blocks the CPU in its progress. This form of waiting by the CPU is not acceptable (constraint 2).

Therefore we opt for another solution: we put the SFRs in a separate process in between the CPU and the peripheral, i.e. we create a *Special Function Register Interface* (SFRI), as shown in Figure 6.2.

6.2.2 SFR-interface

The SFR-interface[1] implements the constraints mentioned in Section 6.1.3. The SFRI decouples the operation of the CPU and of the peripheral, and takes care of

[1] Coined by Joep Kessels

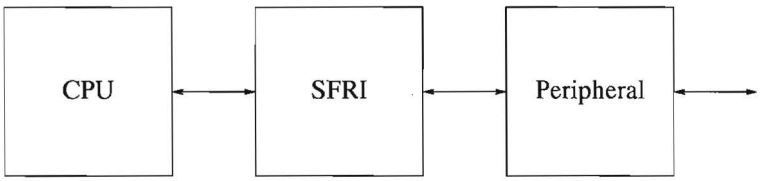


Figure 6.2: SFRI as interface between CPU and peripheral.

1. autonomy of execution: both the CPU and the peripheral must be able to make progress reducing delay (constraint 1) and waiting (constraint 2);

2. the shared-data problem: the SFRs are shared between CPU and peripherals. When both want to access an SFR at the same time, access must take place in mutual exclusion.

In addition to this, the SFRI should

1. make sure that no process overwrites important information that another processor has written in an SFR. We will explain this below, by means of the *Read-Modify-Write* problem;

2. be a generic design, in such a way that it is possible to instantiate this description and have an SFRI for a new peripheral.

We will now explain what read-and-write actions in the 80C51 are to be implemented as atomic actions. When the CPU wants to change a bit in an SFR, by executing a bit-instruction, it first reads the complete SFR, then modifies the single bit, and writes the value of the SFR. Suppose that CPU has read the SFR but not yet written the modified SFR. Furthermore, suppose that the peripheral has just finished its task and wants to set an interrupt, which happens to be a bit in the same SFR. The peripheral then accesses the SFRI and changes the bit. Just after that, the CPU overwrites the same SFR, replacing the interrupt bit with the old value. The interrupt is then lost, and the CPU will not be notified that the peripheral has completed its task. This is called the *Read-Modify-Write* problem: the peripheral should never write a SFR when the CPU has initiated but not yet finished its Read-Modify-Write cycle. Put differently, the Read-Modify-Write cycle is to implemented as an atomic action.

In the 80C51 instruction set all instructions that potentially alter the contents of an SFR are "RMW-dangerous". All instructions that have a directly addressed register as destination are RMW-dangerous, for this register might be an SFR. Furthermore, SFRs can be bit-addressable; therefore also bit-instructions that alter a bit are RMW-dangerous. An overview of RMW-dangerous instructions in the 80C51 is shown in Table 6.1.

	Z0	Z1	Z2	Z3	Z4	Z5	Z67	Z8F
P0						INC dir		
P1	JBC bit,ad8					DEC dir		
P2								
P3								
P4			ORL dir,A	ORL dir,#data				
P4			ANL dir,A	ANL dir,#data				
P4			XRL dir,A	XRL dir,#data				
P7								
P8								
P9			MOV bit,C					
PA								
PB			CPL bit					
PB			CLR bit			XCH A,dir		
PC			CLR bit			DJNZ dir,ad8		
PE								
PF								

Table 6.1: Read-modify-write instructions in the 80C51 instruction set.

6.2.3 Communication between CPU, SFRI, and peripheral

Let us take a look at the communication interface between the CPU, the SFRI, and the peripheral as shown in Figure 6.2. The values that are communicated along these channels are the values in the SFRs. Suppose we have a peripheral with two SFRs: `SFR0` and `SFR1`. These registers can be read and written both by the CPU and by the peripheral. One way of implementing this is by introducing separate read channels and write channels for all SFRs, as shown in Figure 6.3.

The SFRI can now watch all channels for a pending communication by using arbitration (the Tangram `sel...les`-statement):

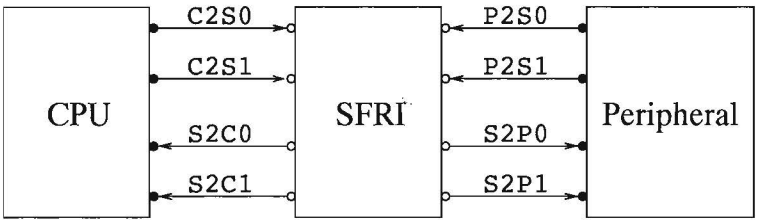


Figure 6.3: An implementation of the SFR-Interface between the CPU and a peripheral.

```
forever
do sel C2S0?SFR0  or  C2S1?SFR1
   or  S2C0!SFR0  or  S2C1!SFR1
   or  P2S0?SFR0  or  P2S1?SFR1
   or  S2P0!SFR0  or  S2P1!SFR1
   les
od
```

This solution is expensive in both the number of channels and in the arbitration used. Accesses to an SFR are done strictly sequentially, according to the sequential nature of instruction execution by the CPU. This implies that at most one communication channel between SFRI and CPU is use at some point in time. The same goes for the communication channels between SFRI and peripheral. Therefore we introduce an extra channel, code, making a number of other channels redundant. This channel indicates which SFR needs to be accessed and whether that SFR should be read or written. Using the code-channel we need only one read channel and only one write channel. Arbitration is now cheaper, as the SFRI has to arbitrate between only one request from the CPU and only one request from the peripheral. In this way we obtain the solution as shown in Figure 6.4.

A possible Tangram fragment for the SFRI is

```
forever
do sel codeC?<<b0,b1>>
       ;   case <<b0,b1>>
           is   <<0,0>> then readC!SFR0
           or   <<0,1>> then writeC?SFR0
           or   <<1,0>> then readC!SFR1
           or   <<1,1>> then writeC?SFR1
```

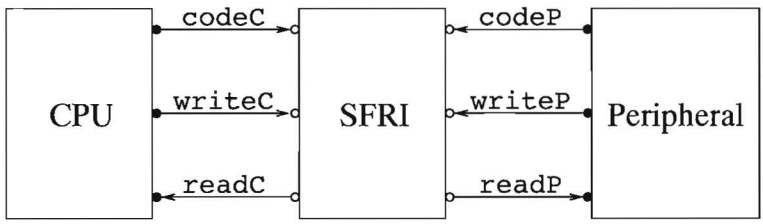


Figure 6.4: A cheaper implementation of interface CPU, SFRI and Peripheral.

```
            si
   or   codeP?<<b0,b1>>
        ;    case <<b0,b1>>
             is   <<0,0>> then readP!SFR0
             or   <<0,1>> then writeP?SFR0
             or   <<1,0>> then readP!SFR1
             or   <<1,1>> then writeP?SFR1
             si
   les
od
```

This implementation saves on the arbitration at the cost of multiplexing in the channels for reading and writing the SFRs (`readC`, `readP`, `writeC`, and `writeP`). Extending the SFRI with an extra SFR involves the extension of the `case`-statements with a new alternative, and possibly the extension of the width of channel `code`.

Suppose that the CPU executes instructions that are Read-Modify-Write dangerous. The CPU can indicate this by means of a boolean expression in the bits of the instruction opcode (Table 6.1). The SFRI can block the peripheral from writing that SFR by first waiting for the correct value to arrive from the CPU:

```
forever
do sel codeC?<<b0,b1,rmw>>
   ;    case <<b0,b1>>
        is   <<0,0>> then readC!SFR0
                     ;    if rmw then writeC?SFR0 fi
        or   <<0,1>> then writeC?SFR0
        or   <<1,0>> then readC!SFR1
                     ;    if rmw then writeC?SFR1 fi
        or   <<1,1>> then writeC?SFR1
        si
```

```
    or   codeP?<<b0,b1>>
    ;    case <<b0,b1>>
         is   <<0,0>> then readP!SFR0
         or   <<0,1>> then writeP?SFR0
         or   <<1,0>> then readP!SFR1
         or   <<1,1>> then writeP?SFR1
         si
    les
od
```

This solution still has the property that the SFRI has to arbitrate between only two channels. When the peripheral itself is able to cause a RMW-problem then a similar solution can be applied in the second alternative of the select-statement.

6.3 Case study: the UART

In this section the observations made in the previous sections are applied to the design of a UART for the 80C51. The UART (Universal Asynchronous Receiver and Transmitter) takes care of serial data communication with the environment. Upon request by the CPU, it transmits a bit pattern to, or receives a bit pattern from the environment. When the UART is enabled to receive a bit-pattern, the environment may decide when the reception starts, by means of sending a start-bit. This explains the *asynchronous* character of the UART: it is not determined on beforehand, when the bit-pattern will arrive. A timing protocol describes the validity of the data with respect to a timing reference (for example a clock or a timer-overflow rate). Thus, the UART has to implement an asynchronous-to-synchronous interface (constraint 3 in Section 6.1.3). It also demonstrates how to implement an SFR-interface.

6.3.1 Specification

The specification of the UART is taken from the data-handbook [4]. The serial port of the 80C51 is full duplex, meaning that it can receive and transmit simultaneously. It is also receive-buffered, meaning that it can commence reception of a second bit pattern before a previously received bit pattern has been read from the receive-register.

There are two Special Function Registers that are of importance for the UART. The first is SBUF, which is used for transmitting and receiving data. When the CPU writes data into SBUF, this bit pattern will be transmitted by the UART; when

a bit pattern was received by the UART, it is stored in SBUF and the CPU can collect the data by reading this register.

7	6	5	4	3	2	1	0
SM0	SM1	SM2	REN	TB8	RB8	TI	RI

Figure 6.5: Special Function Register S0CON.

The other SFR involved is status-register S0CON, which is shown in Figure 6.5. In this register the bits have the following interpretation:

- SM0 and SM1 denote the mode in which the UART operates (see below);
- SM2 enables multiprocessor communication in mode 2 and 3, which will not be dealt with in this section;
- REN indicates that reception of data is enabled;
- TB8 is the 9th data bit that is transmitted in modes 2 and 3;
- RB8 is the 9th data bit that is received in modes 2 and 3;
- TI is the transmit interrupt;
- RI is the receive interrupt.

The UART can operate in four different modes as shown in the Table 6.2. The baud rate in modes 0 and 2 depends on the clock frequency. Thus, the UART has to communicate with the Synchronizer (Chapter 4). The baud rate in mode 1 and 3 depends on the overflow rate of a timer. Therefore, the UART also has to communicate with the timer-peripheral. Loading the timer with different values will result in variable baud rates for the UART in modes 1 and 3.

6.3.2 Architecture

The architecture consists of the UART itself and the Special Function Register Interface. The UART transmits and receives bit patterns, while the SFRI contains the special function registers SBUF and S0CON, which are readable and writable by the CPU. The UART only reads these registers, and therefore it cannot cause

Mode	SM0	SM1	Description	Baud rate
0	0	0	shift register	$f_{\text{clk}} / 12$
1	0	1	8-bit UART	$\frac{2^{\texttt{PCON.SMOD}}}{32} \times f_{\text{tov}}$
2	1	0	9-bit UART	$\frac{2^{\texttt{PCON.SMOD}}}{64} \times f_{\text{clk}}$
3	1	1	9-bit UART	$\frac{2^{\texttt{PCON.SMOD}}}{32} \times f_{\text{tov}}$

Table 6.2: UART modes of operation. Bits SM0 and SM1 are bits in the SFR `S0CON` (Figure 6.5). The baud rates in modes 0 and 2 depend on the clock frequency f_{clk}. The baud rates for modes 1, 2 and 3 depend on bit `PCON.SMOD` (the 7th bit in the Power Control special function register). Modes 1 and 3 have variable baud rates, dependent on the timer overflow rate f_{tov}.

a Read-Modify-Write problem. It communicates the received bit patterns to the SFRI. The UART should of course never deadlock: it must always return to the initial state when an error in data reception or transmission has occurred.

The UART contains two processes, Transmit and Receive, that run in parallel. Both processes can run independently, and therefore we need two sets of communication channels between UART and SFRI, instead of one set of channels as shown in Figure 6.4. The architecture is shown in Figure 6.6. The communication channels between the UART and the CPU follow the same structure as in Figure 6.4.

In this Figure we see three main blocks: the processor CPU, the interface containing the special functions registers (SFRI) and the UART itself. The set of interface channels of the UART can be divided into three categories:

1. The interface channels to the environment of the microcontroller: `RxD` and `TxD`, for reception and transmission of serial data from and to the outside world. These channels are implemented as direct (i.e. non-handshake) channels;

2. The clock (`clk`), the timer overflow (`tov`) and the baud-rate bit `PCON.SMOD`. When this last bit is set to 1, transmission and reception are performed at double baud rate.

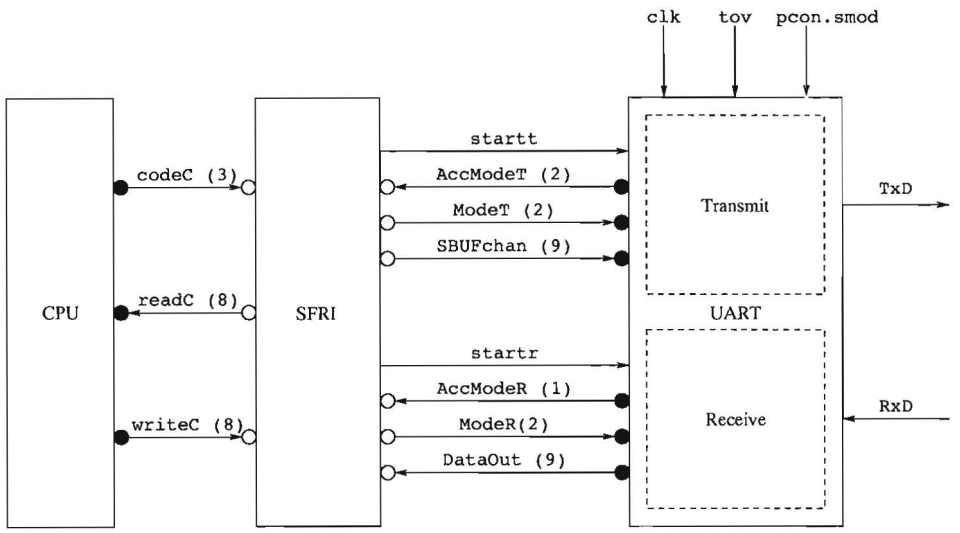


Figure 6.6: The SFRI as interface between the CPU and the UART. The *handshake* channels between SFR-interface (SFRI) and the UART are depicted by arrows with circles; their width is denoted by the numbers. The *direct* channels are depicted by arrows without circles.

3. The channels to the SFR-Interface:

   (a) For *transmission* we have a direct start-channel `startt` to enable transmission (when the CPU writes in `SBUF`), and three handshake channels. `AccmodeT` indicates what data the UART wants to exchange with the SFRI (cf. channel `CodeP` in Figure 6.4):

   - the mode (communicated along `ModeT`) in which the UART has to transmit data;
   - the data to be transmitted (communicated along `SBUFchan`);
   - whether the interrupt flag (`TI`) in S0CON should be raised (upon completed transmission of data).

   (b) For *reception* we have a similar communication interface:

   - the direct channel `startr` to initiate reception. This channel is connected directly to `S0CON.REN` (Figure 6.5);

- handshake channel AccModeR to indicate that the UART requests for receiving the mode from the SFRI;
- ModeR to communicate the mode in which reception has to take place;
- handshake channel DataOut to transfer the received bit pattern upon completion of the reception.

The specification of the UART in the data handbook describes the functionality of the various modes separately, but does not give any constraints as to when a mode may change [4]. It is, for instance, possible to receive a pattern in mode 2 while transmitting in mode 3. In mode 0 both the serial bit lines TxD and RxD are used for either transmission or reception, but not for both simultaneously, as TxD is used to generate a clock signal in this mode. Therefore, when the UART starts transmitting or receiving in mode 0, it should be in the initial state, i.e. not busy receiving or transmitting in any other mode. This constraint is not mentioned in the data handbook [4].

6.3.3 Design of the UART

The UART consists of two processes in parallel, i.e. the *Transmit* process and the *Receive* process. Both processes communicate with the SFRI by means of the direct channels and the handshake channels as introduced in the previous section. After initialization, the main part of the Tangram program consists of the parallel composition of the transmit process and the receive process:

```
  initialize()
; Transmit() || Receive()
```

Transmission

The transmission of serial data is initiated by the CPU writing new data in SFR SBUF. Once the CPU has written into SBUF, the SFRI responds with a transition from 0 to 1 on channel startt, initiating the transmission. The transmission-block of the UART detects this transition and asks the SFRI in which mode the data has to be transmitted. The data is collected from the SFRI and transmitted (in procedures tr0() and tr123() respectively) as described in [4].

A sketch of the Tangram program for the transmission looks like

```
forever
do wait(startt)
;  AccModeT!0
;  ModeT?m
;  if (m.0 + m.1)
   then tr123()     /* transmission in mode 1,2 or 3 */
   else tr0()       /* transmission in mode 0        */
   fi
od
```

Procedures `tr0()` and `tr123()` take care of the transmission interrupt upon transmission of a bit pattern. The value communicated along channel `AccModeT` indicates which data has to be transferred between UART and SFRI. We distinguish the following possibilities:

1. `AccModeT!0`: communicate (along `ModeT`) the mode in which the UART is to transmit a bit pattern;
2. `AccModeT!1`: communicate the bit pattern to be transmitted (along channel `SBUFchan`);
3. `AccModeT!2`: indicate that the transmission interrupt (`TI`) in the `S0CON` register should be set by the SFRI. The UART signals that the interrupt has to be set, but the SFRI actually sets the bit.

The implementation of the actual transmit procedures (`tr0()` and `tr123()`) is straightforward and follows the specification in the Data Handbook [4]. After sending a start bit, the data bits are transmitted; finally a stop bit is sent.

Reception

The Tangram procedure that takes care of the reception of bit patterns has a similar structure. A sketch of this procedure:

```
forever
do TxD0rbit:=true    /* reception clock in mode 0 */
;  wait (startr)
;  AccModeR!0
;  if (m.0 + m.1)
   then rec123()     /* reception in mode 1,2 or 3 */
   else rec0()       /* reception in mode 0        */
```

```
    fi
od
```

A value communicated along channel AccModeR indicates that the UART wants to read the mode from the SFRI. Upon completion of reception the UART communicates the received pattern with the SFRI along DataOut. The SFRI then checks the conditions to write the received pattern into SBUF and raises the interrupt flag (S0CON.RI). As with the transmission-section, the SFRI sets the interrupt bit upon request by the UART.

During reception in mode 0 a clock signal is generated along TxD; we refer to [4] for a complete description of this functionality. The clock is used by the environment for detection of data-validity. In modes 1, 2, and 3 we have the following situation: reception of serial data is enabled by S0CON.REN=1 and a down-going transition on RxD. First the start bit is checked; if it is not a 0 then the UART starts checking for another down-going edge on RxD. When the start bit was correct, 8 or 9 bits of data (depending on the mode) are collected, followed by the stop bit, which should be 1. If it is not a 1 then the data is thrown away, and the UART starts checking for another down-going edge on RxD repeatedly. If the stop bit was correct then the received data is communicated to the SFRI that checks whether the data can be accepted and whether the receive interrupt S0CON.RI can be raised. Each data bit is determined to be the majority of three sampled values, during states 7, 8, and 9 of each bit period. This bit period is derived from the clock (clk) or the timer overflow rate (tov), and the value of PCON.SMOD. While the SFRI deals with the received byte, the UART can commence receiving a second byte.

6.3.4 Design of the SFRI

The SFRI is to be designed in such a way that waiting of any other module (UART or CPU) is avoided. Its main procedure checks for any request from the other components to read or write data, and acts accordingly. Variable starttbit is connected directly to channel startt while channel startr is connected to bit S0CON.REN (reception enable). A sketch of the Tangram program for the SFRI is shown in Figure 6.7.

A handshake-circuit simulation of the operation of the UART is shown in Figure 6.8. In mode 0 we see that during transmission and reception a correct clock signal is generated by the UART (signals TxD0tbit and TxD0rbit, respectively), as specified in the Data Handbook [4].

```
forever
do sel codeC?<<b0,b1,rmw>>
       ;  case <<b0,b1>>
          is <<0,0>> then readC!S0CON
                     ;    if rmw then writeC?S0CON
                                 ;    startrbit:=S0CON.4
                                      /* reception enable */
                          fi
          or <<0,1>> then writeC?S0CON
                     ;    startrbit:=S0CON.4
                          /* reception enable */
          or <<1,0>> then readC!SBUF
                     ;    if rmw then writeC?SBUF fi
          or <<1,1>> then writeC?SBUF
                     ;    if startbit then /* uC too early */
                                           skip
                                      else starttbit:=true
                          fi
          si
   or  AccModeT?accmt
       ;   case accmt
           is 0 then ModeT!mode
           or 1 then SBUFchan!SBUF ; startbit:=false
           or 2 then S0CON.TI:=1 /* transmission interrupt */
           si
   or  AccModeR?accmr
       ;   ModeR!mode
   or  DataOut?x
       ;   if "conditions" then SBUF:=x || S0CON.RI:=1
           fi                        /* receive interrupt      */
   les
od
```

Figure 6.7: Tangram fragment for the Special Function Register Interface (SFRI).

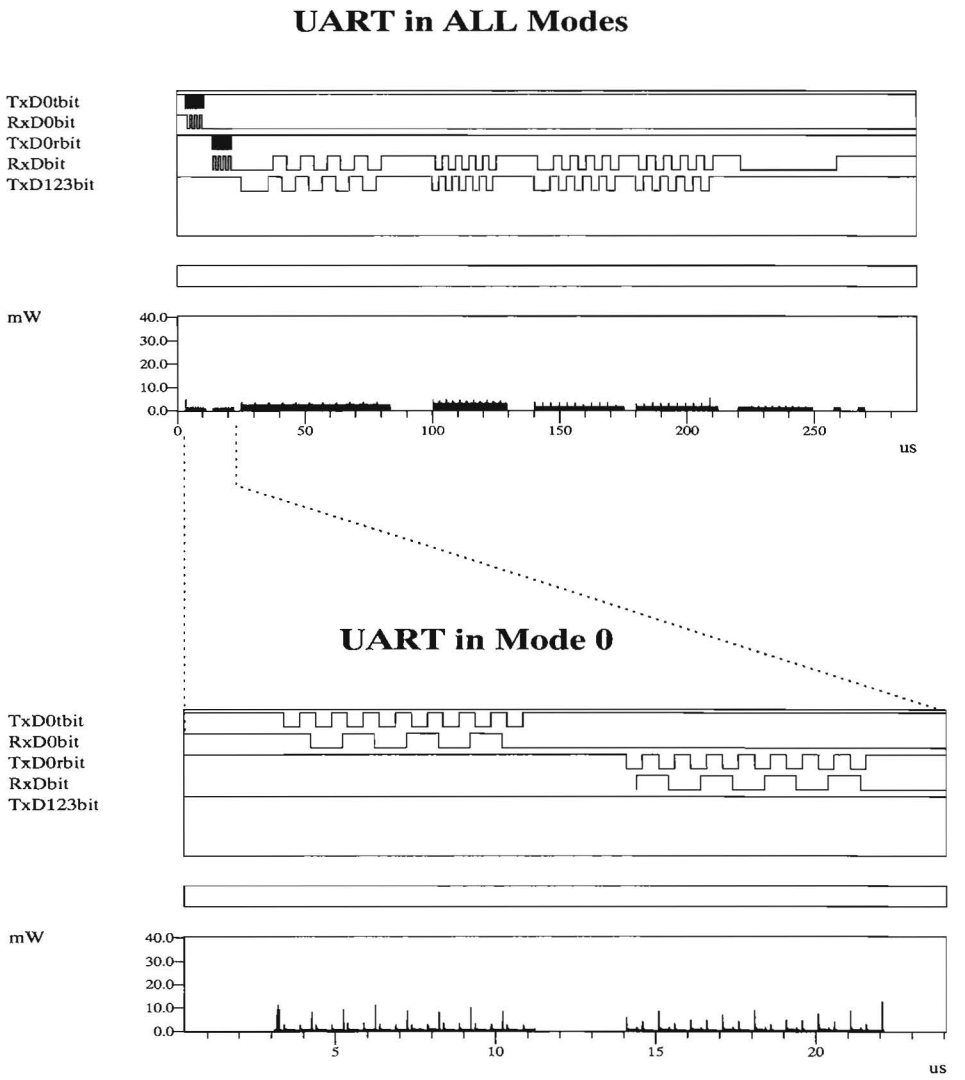


Figure 6.8: Handshake-Circuit Simulation of the UART. A 12 MHz clock frequency and a 5 MHz timer overflow rate are assumed.

6.4 Review

This chapter has introduced a design alternative for the peripherals of an asynchronous 80C51 microcontroller. The separation between the CPU and the peripherals by means of a Special Function Register Interface enables us to construct a modular microcontroller system. New peripherals can be added by defining some special function registers and programming another interface to the new peripheral.

The separation between the CPU and a peripheral enables both units to run as autonomously as possible. They can run in parallel, only communicating when necessary, but not blocking each other. For the UART, even the peripheral itself consists of two parallel processes in (transmit and receive) that run independently of each other.

The UART also shows that no energy is dissipated when the peripheral is not doing anything useful. It just waits to be started and no clock energy is dissipated. The functionality of a peripheral can easily be extended or shrunk by rewriting the Tangram program, which is typical advantage of the VLSI-programming approach. For example, it is straightforward to implement a UART that can only operate in two modes with one baud-rate. Ease of design results in quick generation of derivatives of the microcontroller family.

Chapter 7

Low-Power Implementation

Chapter 3 of this thesis outlines the basic function of the 80C51 microcontroller and identifies some low-power opportunities of the synchronous implementation. Chapters 4, 5, and 6 describe an exploration of the design space of an implementation of the 80C51, using Tangram as a VLSI-programming language.

To demonstrate the feasibility of the Tangram approach to building a competitive asynchronous microcontroller, demonstrator silicon was produced early in the project. The test chip was reported in [52], and was designed in cooperation with Philips Research Eindhoven and Philips Semiconductors Zürich. In this respect, it also served for supporting the transfer from the Tangram tools and the Tangram approach from Research to a product division. The chip has the complete functionality of the 80C51 CPU, with the peripherals that implement the Input/Output, the interrupt controller, and the timers and counters. It shows a power benefit with respect to the synchronous implementation with an overhead in area. A comparison with the synchronous 80C51, as well as with other low-power microcontrollers, is discussed in this chapter.

Before embarking on the chip itself we first review the low-power opportunities as mentioned in Chapter 3. We then describe the asynchronous 80C51 test chip and compare it to a synchronous version of the 80C51, as well as to other microcontrollers and microprocessors.

7.1 Low-power contributions

In summary, we made the following low-power observations based on the synchronous implementation. For the control, we observed that the distribution of the *clock* together with the *centralized control* has two implications: energy is dissipated in the control, but also in the datapath by the redundant actions in the instruction execution scheme. For the datapath, the internal *bus* IB is central in the synchronous implementation. This bus is used in each step of the execution of instructions for communicating uncorrelated data, and therefore shows a high switching activity. Furthermore, some synchronous 80C51 microcontrollers implement master-slave *flipflops* instead of latches. The *peripherals* are clocked at the clock frequency of the CPU (or a division on that frequency), whilst they show less activity, which is not evenly spread in time. Finally, *idle power* can take a substantial part of the total power dissipation, especially in embedded applications.

7.1.1 Distributed control

The distributed control of the asynchronous 80C51 offers a natural way to leave out the redundant actions in the synchronous instruction execution. We have seen examples of such redundant actions in Chapter 3 (Table 3.1). Leaving these actions out can reduce the total energy dissipation significantly.

Distributed control is more complex and occupies more area than centralized control. This is a clear point where area and energy dissipation can be traded for each other. Larger area can result in more energy-efficient implementations. For the 80C51, the hybrid control structure using a mixture of centralized and distributed decoding exploits the regularity in the instruction set execution, and offers a compromise between area and energy efficiency.

7.1.2 Asynchronous control

The asynchronous character of the distributed control in handshake circuits not only reduces activity in the datapath, but also in the control itself. The control of a handshake circuit takes the structure of a tree, in which the leaves are the handshake channels to the transferrers in the datapath. Due to the sequential nature of the instruction execution, there is only one path from the root to a leaf active at any point in time. All others paths remain inactive and thus dissipate no energy.In the synchronous implementation the state machine keeps being clocked, dissipating power.

A clock distributed over the entire circuit triggers each register (flipflop or latch) at every clock cycle. When no clock gating is applied, all registers are clocked while for many of them this is not necessary. For these registers the clock-power is in essence wasted power.

Some studies, like the one in Chapter 3, show that often half of the total power dissipation of a circuit is directly related to the clock [21]. In an asynchronous implementation latches are enabled selectively, thereby avoiding redundant switching of latch-enable signals. The consequence of selective steering of latches is that such a control structure is more complex than a clock-network.

7.1.3 Bus with bypasses

The synchronous implementation has two buses: the internal bus IB and a bus for program-counter traffic. For each step in the execution the same part of the datapath can be used. Therefore, the use of the buses in combination with the sequential nature of instruction execution makes a small implementation possible. Connecting all registers to the bus makes its wires in the circuit layout relatively long, and thus the associated switching *capacitance* high. Since the bus is used for communicating uncorrelated data, on average half of the wires of the bus switches during each slot of the execution. Therefore the switching *activity* on the bus is high.

To reduce the switching activity, point-to-point communication can be introduced, as shown in Chapter 5. Assembler programs for the 80C51 typically show an uneven spread in use of communication paths: some paths are used much more than others. Using this observation makes a construction with bus and *bypasses* attractive: we use direct paths (bypasses) for frequent data traffic, saving long wires and thus switching capacitance. For the less frequent data traffic we opt for the *bus* in Tangram, with its associated smaller area (fewer multiplexers) but poorer performance and higher energy dissipation.

A trade-off can be made to obtain a solution which combines favorable speed, low energy-dissipation and competitive area. One may observe that bypassing the bus for frequent data traffic can also be applied to the synchronous implementation, but this has not been done yet.

A bypass can be seen as a separate piece of datapath, which can be used in parallel with the rest of the datapath. Therefore it allows us to introduce parallelism in the execution of instructions. For example: suppose we have a bypass from the program ROM to the instruction register. This makes it possible to fetch the next byte

from the program memory, while executing the current instruction, thus building a 2-stage pipeline. It heavily depends on the encoding of the instructions whether this scheme can be implemented easily. In the 80C51, a byte in the program ROM can be interpreted as an instruction opcode, immediate data, a relative offset, or an address of some kind. This makes prefetching instructions more cumbersome compared to a RISC with fixed-length instruction encoding, as we will discuss briefly in Chapter 8.

7.1.4 Latches

Some synchronous implementations of the 80C51 are based on flipflops. The sequential nature of instruction execution makes it possible, however, to use latches instead, as reading and writing the same variable in one clock cycle do not occur. The asynchronous implementation is latch-based, but also a synchronous implementation can be made using latches. Avoiding flipflops makes the implementation more energy-efficient because there is less switching power. On the other hand, using flipflops makes a design easier to test because flipflops can easier be incorporated in a scan-chain.

7.1.5 Peripherals

We have seen that peripherals are *demand-driven* by nature; they start their activity upon request. The synchronous implementation clocks peripherals at the clock frequency of the CPU, or a division of that, thereby dissipating energy unnecessarily. The asynchronous nature of the Tangram-compiled circuits makes it possible to activate the peripheral circuit only when necessary. The SFR-interface as discussed in Chapter 6 decouples the CPU and the peripherals, making such an implementation possible. Furthermore, the SFR-interface ensures maximum progress of both the CPU and the peripherals.

7.1.6 Idle power

In embedded applications, idle power can be an important issue. For example, a cellular phone in standby-mode has to contact its base station a couple of times per second. The circuitry for this function only has to be activated at this frequency, which is orders of magnitude lower than the frequency of the clock of the CPU. In embedded applications the circuits are often in idle mode for a large portion of the

total execution time. Therefore, the power dissipated in idle mode is important for the total power dissipation in these applications.

In the synchronous implementation there are several *modes* for power saving, in each of which the clock is switched off in some parts of the circuit. Idle mode switches the clock off the CPU, but keeps some peripherals active. Sleep mode stops the oscillator completely. Stopping the oscillator has the disadvantage that resuming the activity of the circuit takes a long time (several ms) to start the oscillator.

The asynchronous 80C51 has no distinction between idle mode and sleep mode. Compared to the synchronous 80C51, the asynchronous version has the advantage that in idle mode the peripherals are active only when necessary, i.e. they are potentially lower-power. In sleep mode, the asynchronous 80C51 has the advantage of *instantaneous wake-up*; the circuit can instantly resume its activity.

We have seen six measures that were taken to reduce the energy dissipation in an 80C51 microcontroller. Schematically, these measures are shown in Figure 7.1.

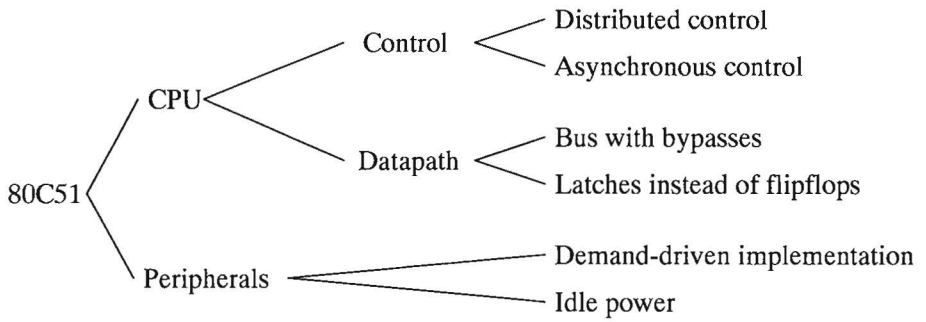


Figure 7.1: Parts of the 80C51 microcontroller and measures to save power.

7.1.7 Evaluation

The components in a handshake circuit can be divided into several classes, of which the energy dissipation per class can be obtained by simulation on the gate-level. The VLSI-programmer can make any such classification. For the 80C51 CPU, we distinguish between the bus (variable, drivers, and multiplexer), control (the control handshake components), latches (the other variables), combinatorics (for example

the binary adder), and multiplexing. Using this classification for the asynchronous 80C51 CPU without bypasses, apart from a direct path from the program counter to the program memory (Table 5.6), we obtain the distribution of the energy dissipation as shown in Figure 7.2.

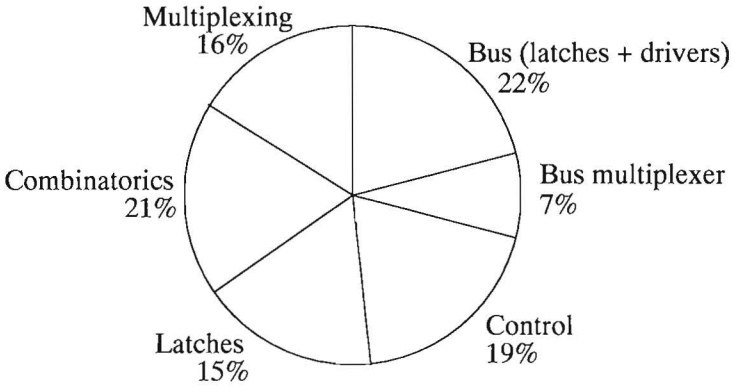


Figure 7.2: Distribution of the energy dissipation in the asynchronous 80C51 CPU with one bypass (the path from the program counter to the program memory).

The energy dissipated by the control is, percentage-wise, considerably less than in the synchronous case. This is due to the distributed control and its asynchronous character. We see that the energy dissipation by the bus is rather large, of which a considerable part is dissipated by its multiplexer. As far as energy dissipation is concerned, there is not much difference between the synchronous and the asynchronous bus. The synchronous 80C51 bus consumes about 0.6 nJ per instruction, and the asynchronous bus 0.4 nJ. However, the asynchronous 80C51 reduces on the redundant actions that account for $\frac{1}{3}$ of the total number of actions. Therefore, energy-wise, there is not much difference between the asynchronous bus and the synchronous bus.

It is now worthwhile to reduce the energy dissipation of the bus by introducing four bypasses as discussed in Chapter 5. We then obtain Figure 7.3. The energy in the final 80C51 CPU is evenly distributed over the various classes of components.

From these charts we conclude the following. First, the energy dissipation by the control in the asynchronous 80C51 is substantially lower than in the synchronous implementation. About $\frac{3}{4}$ of the energy in the asynchronous 80C51 is dissipated in the datapath, in contrast to the energy dissipated by the clock and in the control-

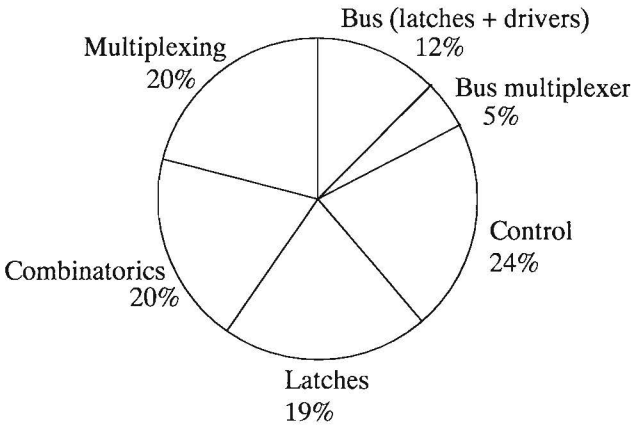


Figure 7.3: Energy distribution of the asynchronous 80C51 CPU with four bypasses.

flipflops in the synchronous 80C51 (Figure 3.6).

The latches in the datapath take a smaller part in the asynchronous 80C51 than the flipflops in the synchronous case. This is the result of less clock energy as the latches are only enabled when necessary.

What remains is an energy distribution where the bus takes a substantial part. Bypassing the bus is an attractive alternative, especially in the 80C51 where only a few communication paths are used frequently.

7.2 Demonstrator chip

The history of the project goes back to January 1995. Taking the hardware description in the 80C51 Data Handbook [4] as starting point, a prototype of the CPU was designed in Tangram. This prototype could execute all but two instructions (multiply and divide). The control of this prototype already eliminated most of the redundant actions by distinguishing between the Regular and the Irregular part of the instruction set (Section 5.3.3). It was completed in two months time.

Simulations of the prototype showed a significant power advantage compared to the best-known synchronous 80C51 CPU at that time. It was then decided to complete the design, including peripherals such as timers and counters, an interrupt

controller, and port logic. The chip also implements the timing-compatibility constraints for external memory access, as discussed in Chapter 4. The completion was done in four months time, as a joint project between Philips Semiconductors Zürich and Philips Research Eindhoven. The design of the CPU was completed at Philips Research by the author, and the design of most of the peripherals was done by Philips Semiconductors. The general architecture of the SFR-interface was developed at Philips Research, as was the design of the UART (by the author), as discussed in Chapter 6. This UART was not yet incorporated in implementation of the test chip.

While many optimizations at the Tangram level were not yet incorporated, the layout of a netlist was sent for fabrication in July 1995. This chip was meant to be a demonstrator of the feasibility of the Tangram design flow to design and implement a low-power version of the 80C51 microcontroller. It was fabricated in a 0.5μ 3-metal layer CMOS process using a generic standard-cell library. The on-board program ROM and data RAM are the same as used in synchronous implementations in the same technology. The only difference is that they have a handshake wrapper: data-valid signals in the memories are connected to the appropriate handshake-signals in the standard-cell block. In that way it is possible to describe the communication with the memories on the level of Tangram, as shown in Section 4.2. Simulations were done at the gate-level, using Verilog[1] (for functional correctness and timing behaviour) and Diesel (for power estimation at the gate-level) [2]. Three months later, silicon arrived and the testing of the chip started. It proved almost first-time right: available tests revealed an error in the DA-instruction that could easily be fixed at the Tangram level.

7.2.1 Chip

The layout of the chip is shown in Figure 7.4. It can be divided into three parts: a program ROM of 16KByte (the block on top right), a data RAM of 256 bytes (top left) and a standard-cell block. This block implements the complete 80C51, including CPU, peripherals, and the Synchronizer, as shown schematically in Figure 4.1.

Area of the chip had no priority in the first place; the test chip is about twice as large as the synchronous implementation in the same cell library. In the first place this is due to the introduction of debug-registers on the chip for testing purposes. They have no use for the functionality. Second, the chip implements point-to-point communication between the registers. As we have seen in Chapter 5, communi-

[1] Verilog is a trademark of Cadence Design Systems, Inc.

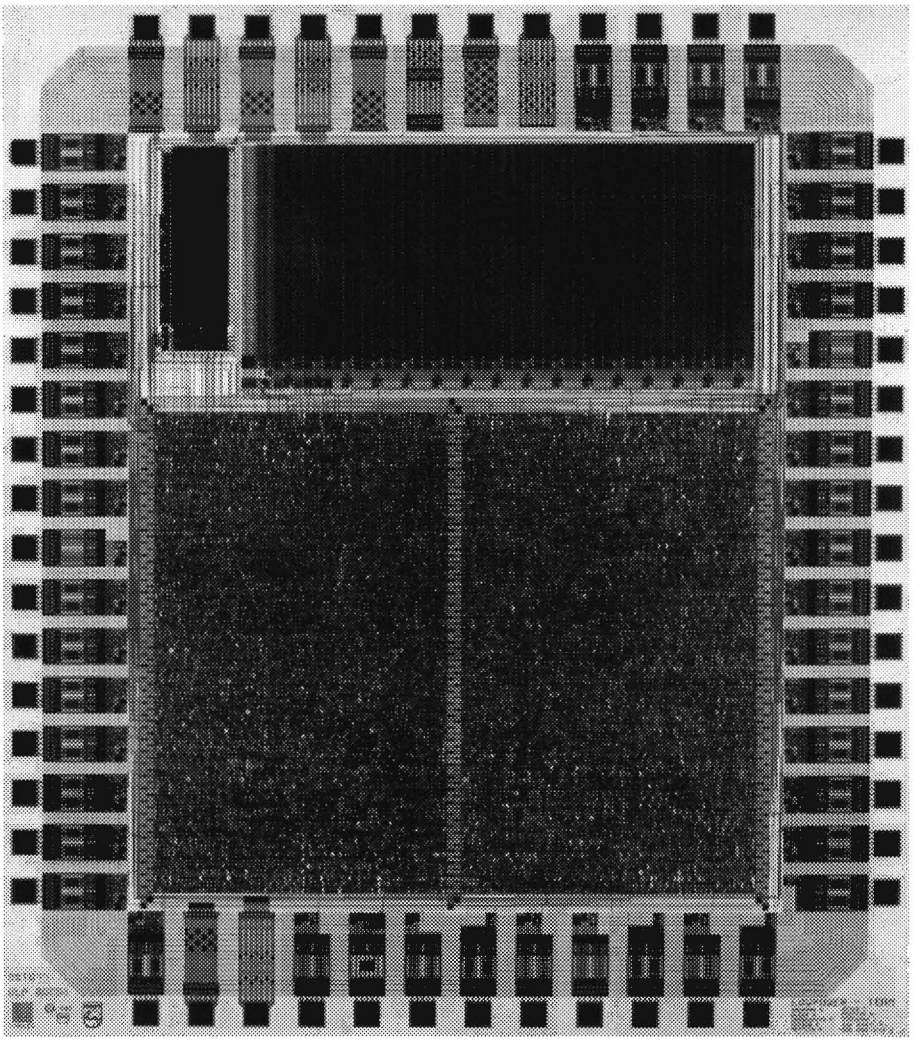


Figure 7.4: Layout of an asynchronous 80C51 microcontroller.

cation using a bus with a few bypasses results in a circuit that is smaller, while saving power and increasing speed. This was learnt after the chip was fabricated. Furthermore, many local optimizations, also described in Chapter 5, can be applied on the test chip's design, resulting in a smaller area.

7.3 Evaluation

7.3.1 Comparison with synchronous 80C51

We can draw a comparison with a synchronous version of the 80C51, implemented in the same cell library, but with a few more functions (a UART for example). This is the same implementation of the 80C51 that we used in Section 3.4 for the power analysis. The asynchronous version runs 4 MIPS when running freely. To make a fair comparison we let the synchronous version run on a 36 MHz internal clock to achieve the same speed, at the same supply voltage ($V_{dd} = 3.3V$). The synchronous 80C51 is a VHDL design that is synthesized to run a maximum internal clock frequency of 40 MHz (i.e. one clock cycle per slot in the instruction execution). Thus, the asynchronous 80C51 is slightly slower than its synchronous counterpart. Table 7.1 shows that the asynchronous 80C51 is 4 times more power efficient than its synchronous counterpart when running under similar conditions. The values are measurements of the chips, including CPU, peripherals, and memories. When a correction is made for the different functionality of the ICs, the power advantage of the asynchronous version is reduced to a factor 3.

Version	MIPS	Power (mW)	MIPS/W
Sync 80C51	4.0	40.0	100
Async 80C51 (with CPU V1)	4.0	9.00	444

Table 7.1: *Measurements* of synchronous and asynchronous 80C51 ICs (CPU, peripherals, and memories) under *typical* conditions. When a correction for different functionality is made, the power advantage of the asynchronous version is reduced to a factor 3.

Another way to compare the synchronous and asynchronous 80C51 is by visualizing *photon emission*. When a circuit is in operation, photons will be emitted from the places on the chip where circuitry is switching. These photons can be registered

	Area (trans.)	Speed (MIPS)	MIPS/W
CPU V1 (on-chip: point-to-point)	39174	1.86	685
CPU V2 (local optimizations)	31374	1.92	781
CPU V3 (bus and bypasses)	27482	2.10	943

Table 7.2: Results of gate-level *simulations* of three asynchronous 80C51 CPUs, excluding memories, assuming *worst-case* conditions. All CPUs execute instructions from *external* memory.

with a camera when their energy intensity, integrated over some period of time, is above a certain threshold. This can be made visible by white spots on a picture. Figures 7.5 and 7.6 show the synchronous and the asynchronous versions of the 80C51 under similar conditions running similar code. As can be seen from the picture the synchronous version releases more photons than the asynchronous one, indicating higher power consumption. Furthermore, activity on the asynchronous chip is more local than on the synchronous one.

7.3.2 Improvements

The test chip was designed and implemented early in the project. It showed the feasibility of the VLSI-programming approach for making a redesign of an existing microcontroller architecture, with an interesting power advantage. The area of the test chip had no priority, but the overhead of almost a factor 2 should be brought down. The first step in reducing the area was to remove the redundant debug registers that were implemented on the test chip. Other reductions in area had to do with restructuring the VLSI-program.

Chapter 5 of this thesis describes how we can reduce area of the CPU, by rewriting the original Tangram program. For the 80C51 CPU, the results of these improvements are shown in Table 7.2.

There are two major steps to reduce the area. First, local optimizations can be done, like sharing combinatoric functions and reducing the sequencers in the control (Section 5.4). This yields CPU V2 in Table 7.2, which still implements point-to-point communication. Second, going from point-to-point to a bus-structure including four bypasses (Section 5.2), we were able to save even more area (CPU V3). The improved speed and energy-efficiency are due to the fact that the by-

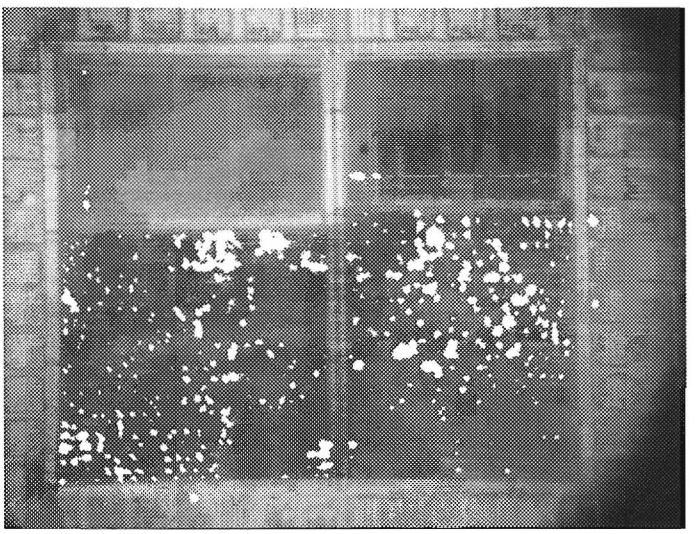


Figure 7.5: Photon emission of the synchronous 80C51.

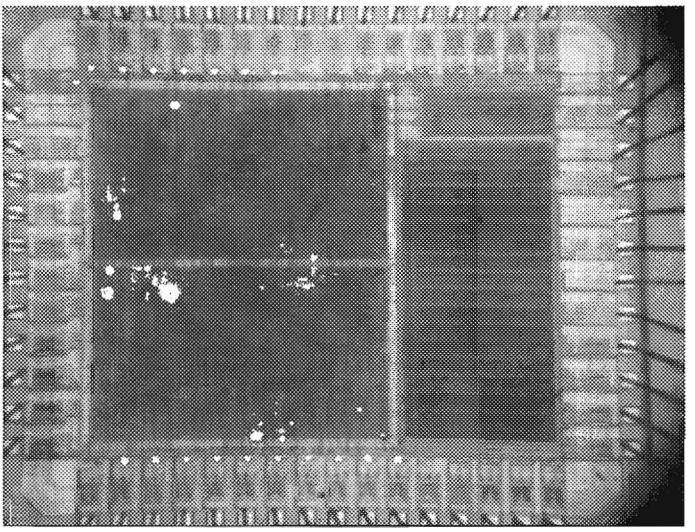


Figure 7.6: Photon emission of the asynchronous 80C51, showing less activity, which is more localized.

passes are used frequently in the 80C51 CPU.

With exception of RAM and ROM, all handshake components are mapped onto a generic standard-cell library, which contains no dedicated asynchronous cells. This does contribute to the area overhead of single-rail handshake circuits: with the addition of about 4 asynchronous cells (various C-elements) the area can be reduced by approximately 10%.

Handshake circuits are implemented using a four-phase handshake protocol and single-rail encoding of data. The standard-cell implementation makes that special attention has to be paid to delay matching and the verification (after layout) of the timing assumptions that have been made [30]. In order to minimize the verification effort, delay-matching is done conservatively. For the chip, a safety margin of 100% was chosen, but a margin of 30% for delays in the datapath is feasible, even in standard-cell implementation. These delays have not been corrected for the additional safety margin that results from the delay in the control path.

The Tangram programs for the *peripherals* also give room for reducing the area. These improvements at the Tangram-level, together with the few extra cells in the cell-library, make an area overhead of 30% over the synchronous implementation possible. The asynchronous 80C51 is slightly slower than its synchronous counterpart, but offers a power advantage of a factor 4.

7.3.3 Comparison with other designs

In designing a low-power asynchronous 80C51 IC we took the 80C51 instruction set and its sequential execution scheme as starting point. Comparing the chip with its synchronous counterpart with the same functionality and implemented in the same technology, we were able to save a factor 4 in power. However, starting from scratch one could go even further. The CoolRisc 8-bit microcontroller designed by Piguet et al. is an example [31]. The instruction set of this RISC microcontroller was designed from scratch, and allows for efficient pipelining, thereby introducing parallelism during the execution of instructions. The result is an impressive 3000 MIPS/Watt figure. The CoolRisc's RISC instruction set is different from the 80C51 CISC instruction set, with less expressive power per instruction, and fewer addressing modes. Therefore the MIPS/Watt-figures of the CoolRisc and the 80C51 are not directly comparable.

Reducing activity in a circuit is a key strategy to save energy dissipation. This approach was also used in an asynchronous implementation of a 16-bit DSP by Cogency [39]. The architecture of this DSP consists of several functional units

(an ALU, a multiplier, etc.). The decoder fetches instructions and translates them into an intermediate format, adding information about which functional units are involved in the execution of the instruction. Each unit that is not involved, sends an acknowledgement immediately, thus dissipating no energy on calculation. The units that have to do the work send an acknowledge upon completion. The next instruction is fetched as soon as all units have signalled completion of the previous instruction. The decoder in this architecture is more complex (i.e. larger in area) than the synchronous equivalent without clock gating. On the other hand, it saves energy compared to the synchronous implementation by selectively invoking the necessary parts of the datapath.

Researchers at the University of Manchester made several asynchronous implementations of the ARM microprocessor [27, 11, 9, 36]. Amulet1 shows that asynchronous logic in a large design is feasible, but it does not demonstrate a power advantage or speed advantage compared to the ARM6 [27]. The lessons learnt from Amulet1 were incorporated in the design of Amulet2e, which shows a performance and power efficiency comparable to ARM7 and ARM8 [9]. The ARM is a RISC, in which most steps of the instruction execution are useful and necessary for that instruction, i.e., there are not as many redundant actions as in the case of the synchronous 80C51. However, the advantage of less electromagnetic emission could be present in the Amulet processors as well, making them attractive for portable applications.

Martin et al. reported on an asynchronous implementation of the MIPS R3000 RISC microprocessor [24]. Their "MiniMIPS" is laid out using full custom cells, and the architecture aims for high throughput of instructions. They take the power-delay product ET^2 as measure for comparison and claim to be very competitive in this respect. Compared to Amulet2e the MiniMIPS is an order of magnitude better, as the ET^2-figures suggest. Measurements from silicon, however, are not yet available.

At Tokyo Institute of Technology and the University of Tokyo, researchers have designed and implemented a 32-bit asynchronous microprocessor, the TITAC-2 [38, 25]. This processor is based on the MIPS R2000, but the instruction set is modified. This makes a comparison with a synchronous MIPS R2000 implementation not straightforward. The TITAC-2 processor uses double-rail encoding of data, and is based on the so-called *Scalable Delay Insensitive* delay model. Using this model, the design is split into small parts, and the design of each part is based on a more relaxed (and more optimistic) delay model than the QDI (quasi-delay insensitive) model. The interconnection between the various blocks is based on the DI (delay-insensitive) model.

7.4 Review

One of the main conclusions of this chapter is that it is possible to make a competitive design using Tangram as VLSI-programming language and the Tangram compiler to map the program to an asynchronous netlist. The tools offer the VLSI-programmer the possibility to redesign the Tangram program in such a way that the resulting circuit is better in area, execution time and energy dissipation. The demonstrator chip was produced in short time, and proved to be functionally correct.

Making a design and making a *competitive* design in terms of area, execution time, and energy dissipation are two different activities. Phrased differently, *programming in Tangram* is rather straightforward, but *VLSI-programming in Tangram* is more difficult. Tangram and the transparent compiler offer a large freedom to investigate the design space. In a way the 80C51's instruction set is suitable for research purposes; its mixture of regular and irregular parts forces the designer to inspect the design space and to make a compromise between area on the one hand, and speed and power efficiency on the other. For a good VLSI-program it is necessary to have insight in the compile function and handshake circuits. It is not necessary, however, to have detailed knowledge about the low-level netlist implementation.

VLSI-programming in Tangram offers the designer a natural way to use the asynchronous distributed control structure of handshake circuits. This design style is different from the synchronous design style where there is a centralized control. Therefore VLSI-programming in Tangram can result in new architectures that exploit the distributed control to save power.

Chapter 8

Concluding Remarks

The main targets of the research described in this thesis is to describe guidelines for VLSI-programming and to illustrate them on a vehicle of industrial relevance: the 80C51 microcontroller. The process of VLSI-programming has been demonstrated by an exploration of the design space of a low-power asynchronous version of this microcontroller.

The approach taken is to partition the microcontroller into a CPU with memories, peripherals, and a Synchronizer block, as shown in Figure 4.1. The blocks communicate with each other using handshake channels and in this way a handshake CPU is isolated. The partition makes modular design of a microcontroller family possible, as derivatives differ in the memories and in the peripherals.

The Synchronizer takes care of supplying timing information to the CPU and to the peripherals, as well as to the environment of the chip. It is the only process that has a clock as input, and therefore the clock is not distributed over the entire circuit. The Synchronizer caters for *timing-compatibility* with an (existing) synchronous environment. Though timing-compatibility of an asynchronous chip with a synchronous environment is possible in principle, it can also reduce the advantages of an asynchronous system. Therefore it is important to decide what degree of compatibility is required. For the asynchronous 80C51, timing-compatibility for external program memory access has been implemented.

The handshake CPU can be split into a datapath and the control. For the datapath, it turns out that a combination of a bus with bypasses for frequently used communication paths, offers the best compromise in area on the one hand, and execution time and energy dissipation on the other hand. For the control the regularity of the instruction set has been exploited, reducing the number of actions per instruc-

tion compared to the synchronous version. The distributed character of the control saves on the redundant actions and saves power in the datapath. Furthermore, the *asynchronous* character of the control also saves power in the control as only a few handshake control components are active at a given point in time.

The peripherals are demand-driven (rather than clock-driven) by nature, and this fits nicely with an asynchronous design style. They dissipate energy only when necessary. The CPU and the peripherals communicate with each other using the Special Function Registers. The Special Function Register Interface contains these registers, and caters for the communication to the CPU and to the peripheral. The SFRI also decouples the CPU and the peripheral, in such a way that maximum progress by all components is established.

These techniques result in an asynchronous 80C51 that, compared to its synchronous counterpart, is slightly slower and has 30% more area, but dissipates four times less energy.

Philips Semiconductors has implemented the asynchronous 80C51 in a pager chip. This pager is available on the world market today.

Tangram as a programming language offers the possibility to express microcontroller architectures. The tool set allows for quick feedback of the characteristics of the circuit (a handshake simulation of the 80C51 CPU will simulate around 250 instructions per second CPU time) and enables the designer to do a design space exploration. The transparent compilation scheme of Tangram to the netlist of a circuit enables the designer to reason about the circuit on the level of the programming language. However, some circuit characteristics are hidden in Tangram but visible on the handshake circuit level (e.g. multiplexers). Therefore it is important for the designer to have knowledge about the structure of handshake circuits, and of the compilation scheme from Tangram to handshake circuits. It is, however, not necessary for the designer to have a detailed knowledge of the mapping from a handshake circuit to the netlist of a gate-level circuit.

8.1 Other processor architectures

This thesis has discussed various alternatives for architectures in the design space of the 80C51 microcontroller. In this section we consider the applicability of these alternatives in the context of other processor architectures, in particular RISC architectures.

The 80C51 is a typical example of a sequential processor architecture. This keeps

the datapath small, but also results in relatively poor performance in terms of execution time, compared to other microcontrollers and microprocessors. Introducing parallelism in instruction execution can significantly improve the processor's speed. RISC processors aim at this kind of parallelism by overlapping instruction execution, i.e. by implementing pipelining. First we show how how this can be accomplished in Tangram. Then we make some power observations concerning an asynchronous RISC processor.

8.1.1 Pipelining in Tangram

The execution of an instruction can often be split into several components, that are more or less of the same kind for all instructions. For example, each instruction execution can be split into a *fetch* and an *execute* part, where the fetch part is the same for all instructions. When each stage uses a separate part of the datapath of the circuit, then overlapping the execution of instructions is possible. Pipelining is described extensively in many books on computer architecture (for example by Hennessy and Patterson [14, 15, 26]).

From an abstract point of view, pipelining can be seen as function decomposition. Taking this viewpoint, we can express pipelining in Tangram in the following way. Suppose we have the Tangram fragment

```
forever
do a?x
;  y:=f(x)
;  c!g(y)
od
```

The corresponding handshake circuit is shown in Figure 8.1.

Using the possible overlap in instructions, we can split the calculation of `f(x)` and `g(y)` and put them in parallel for two consecutive instructions:

```
   ( forever do a?x ; b!f(x) od )
|| ( forever do b?y ; c!g(y) od )
```

We then obtain the handshake circuit as shown in Figure 8.2, which has the same structure as the 2-place buffer in Figure 1.4.

In this handshake circuit an extra channel `b` has been introduced to forward the data produced by the first "stage". For synchronization between the stages a passivator is added (Peeters describes a handshake circuit with the same functionality in which the synchronization is done in the control [30]). This decoupling of the calculation of `f` and `g` makes it possible to overlap their execution in time. For

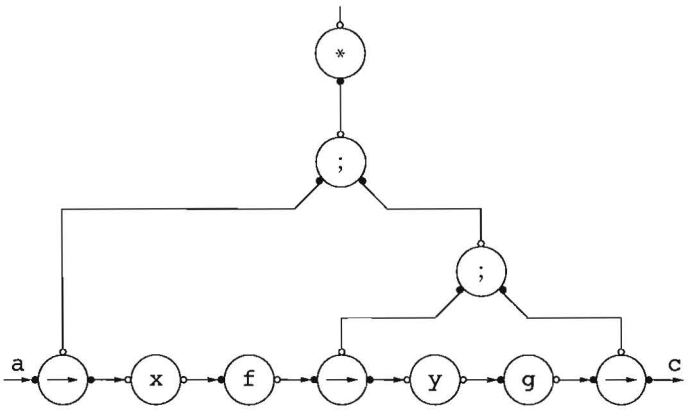


Figure 8.1: Sequential execution.

example, suppose that `f` is the fetch-part and `g` implements the execute-part of an instruction, then these two actions can be overlapped for two consecutive instructions. The handshake circuit of the two-stage pipeline shows that pipelining yields more complex circuits in terms of number of handshake components (i.e. area). This is a penalty for the higher throughput.

8.1.2 Asynchronous RISC and power

In Chapter 3 we saw four CISC-characteristics of the 80C51 instruction set:

- The various addressing modes;
- the variable length encoding of the instruction set;
- the non-uniform register structure;
- the variable number of clock cycles in which an instruction is executed.

In the design of a RISC processor, the instruction set and the datapath are designed simultaneously. A RISC instruction set is defined around a *register file*. There is only one addressing mode, viz. the one that addresses registers in the register file. RISC instructions are usually encoded using a fixed length for all instructions, with redundancy in the encoding. This makes it possible to have separate *fields*

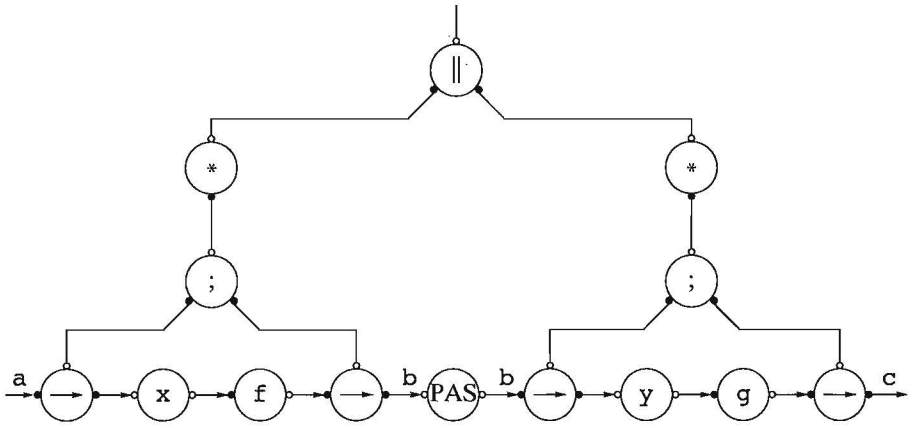


Figure 8.2: Two-stage pipeline with synchronization in the datapath.

in the instruction encoding, for example an opcode field, and fields that specify registers in the register file. The instructions are executed in a fixed number of steps. Examples of RISC machines are the ARM6 [11], the MIPS R3000 [18], the DLX [15], and the StrongARM [11]. Let us now look at the datapath and the control of a RISC processor to determine the power-saving options.

The *datapath* of a RISC is built around a register file. Instructions read values from this register file, operate on them and store the result in the register file. There is a limited number of communication paths outside the register file. The multiplexing in the datapath that we encounter in the 80C51, is hidden in the register file. Therefore, the discussion of point-to-point communication and the bus-communication that we encountered in Chapter 5, does not apply to such RISC architectures.

For the *control* of a RISC machine, we consider the instruction execution scheme of a RISC. The synchronous 80C51 executes each instruction in six or twelve steps per instruction, but a RISC instruction takes in general fewer steps to execute. An example of such an execution scheme is shown in Table 8.1. This table shows instructions that execute in five steps: first the instruction is fetched, then register values are read from the register file, an ALU-operation is performed, the data memory can be accessed, and finally the result is written back into the register file. Each instruction in the instruction set fits into this scheme. Five-stage RISC machines like in Table 8.1 are implemented in the MIPS R3000 [18], the DLX [15],

Step	Name	Action
1	IF (Instruction Fetch)	Fetch instruction opcode from progr. memory
2	ID (Instruction Decode)	Decode instruction; read register file
3	EX (Execute)	ALU-operation
4	MEM (Memory Access)	Read or write data-memory, if necessary
5	WB (Write Back)	Write result back into register file

Table 8.1: The instruction execution scheme of the DLX RISC instruction set in five stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory access (MEM) and Write-Back(WB) [15].

and the StrongARM [11]. The ARM6 [11] implements a three-stage execution scheme: Fetch, Decode and Execute, in which Execute combines the stages 3, 4, and 5 in Table 8.1. We observe from this scheme that there are not as many *redundant actions* that can be removed by using a distributed control, as we encounter in the 80C51.

For the decoding of a RISC instruction, we observe that the decoding per stage is relatively simple, compared to the 80C51 CISC. The decoding is enhanced by the fact that instructions are encoded in fixed length using fields; usually a stage only needs information in a few fields to determine its task. This suggests to use a decode structure with *distributed decoding*, as shown in the handshake circuit of Figure 5.11.

Furthermore, the stages in the execution scheme use *separate pieces of datapath.* IF uses channels to the program ROM and uses the program counter; ID contains the register file; EX contains the ALU; MEM caters for the communication to the data memory, and WB uses the register file to write the result. The only two stages that use one resource (the register file) are ID and WB. The use of separate pieces of datapath per stage enables *overlapping* the execution of instructions, i.e. introducing *pipelining*. Pipelining does not reduce the latency but it does improve the throughput of instructions. The price is an overhead in area, as information about the instructions has to be transported along the pipeline. Another problem with pipelining is the introduction of *data hazards*, i.e. the situation that one instruction needs the result of a previous instruction that has not yet been made available. This problem can be solved by stalling the pipeline, which reduces the performance advantage, or by forwarding results between the pipeline stages, which costs some area (extra communication paths and multiplexers). Hennessy and Patterson describe these issues extensively in [15].

The goal of pipelining is to reduce the execution time by having the stages being active at any point in time. Asynchronous and distributed control does not offer an obvious power advantage, as all parts of the circuit show more activity.

Researchers at Manchester University have developed the AMULET-series asynchronous microprocessors [27, 11, 9, 36]. These processors implement the ARM RISC instructions set. AMULET takes a different approach to RISC than the pipelines described in this section; in their implementation *micropipelines* are used, as introduced by Sutherland [37]. The AMULET chips show that asynchronous RISC microprocessors can be designed and implemented with similar power and speed figures as their synchronous counterparts. They offer, however, another advantage of asynchronous circuits: the absence of a clock leads to lower *electromagnetic radiation* with the additional advantages that the electromagnetic spectrum does not show the peaks caused by the clock. The asynchronous 80C51 in this thesis offers the same advantage of reduced electromagnetic emission, compared to its synchronous counterpart.

In summary, the simplicity of the *datapath* of an *asynchronous* RISC machine does not leave much room for power saving. Also the *control* of an asynchronous RISC machine has no particular power advantages: first, there are few redundant actions, and second, a RISC circuit shows more activity. However, asynchronous RISC processors do not have a clock, resulting in lower electromagnetic emission compared to their synchronous counterparts.

8.2 Typically asynchronous?

We have seen various techniques that were implemented to save power in the 80C51 microcontroller. A legitimate question is whether these techniques could have been applied to a synchronous implementation as well. We take our six low-power opportunities of Chapter 3 and low-power solutions of Chapter 7 as starting point.

The *bus with bypasses* can be implemented synchronously as well. Both in the synchronous and in the asynchronous case, the penalty will be some overhead in area for the control.

Using *latches* instead of master-slave *flipflops* saves energy in a synchronous solution as well. The sequential nature of instruction execution makes a straightforward implementation using latches possible. On the other hand, using master-slave flipflops instead of latches makes it easier to test the circuit.

Clock-gating can reduce the activity of the clock in the circuit. However, clock-gating makes a more complicated control necessary, and is not as fine-grained as the asynchronous control. In asynchronous design, the starting point is to steer the latches only when and where necessary. Synchronous design starts from the opposite side: all latches or flip-flops are clocked at all clock cycles. Clock-gating reduces the clock-activity at a global level: it switches the clock off in a part of the circuit that needs not to be active. Seelen describes a tool that automatically implements clock-gating [35]. For the synchronous 80C51 this results in a reduction of the clock power of 24%.

In connection with this, *distributed control* makes it possible to reduce the capacitance of the state machine. In other words, also in the *control circuitry* power is saved. Distributed control makes it possible to remove the redundant actions in the 80C51 instruction execution. Implementation of distributed control would in the synchronous case come with an overhead in area, because it is more complex. This overhead can also be found in the asynchronous solution. However, the *asynchronous character* of the distributed control also reduces the energy dissipated in the *control* as only one path in the handshake control tree shows activity at a given point in time. This is a typical advantage of an asynchronously operating control structure.

The operation of *peripherals* is by nature asynchronous, though clock-gating can help to reduce the power dissipated in synchronous implementations. But that takes more design effort than in an asynchronous implementation, where it comes naturally with the design style. Peripherals are by nature *demand-driven* and not *clock-driven*. Take the UART that operates in reception-mode as an example: it waits for an external start-bit to arrive. Energy is only dissipated in the asynchronous version once this start-bit has arrived. The synchronous version has to be clocked, even during the period that it waits for the start-bit to arrive. Thus, in this case, the combination of *immediate response* and *low-power* is harder to accomplish in synchronous design than in the asynchronous design.

Finally, the synchronous implementation has two power-saving modes: idle mode and power-down mode. Idle mode stops the clock in the CPU but keeps the peripherals clocked; power-down mode stops the oscillator, in which case it takes a few ms to re-start the system. The asynchronous implementation does not make a distinction between idle mode and power-down mode: there is no activity when not necessary, and the circuit can respond immediately to resume activity. It is not possible to implement this combination in a synchronous solution without the clock running.

Reviewing the six low-power opportunities, we can say that the bus with bypasses

and the use of latches, can be straightforwardly implemented in a synchronous design. The cost in terms of area and reduced testability is roughly the same as for the asynchronous case. The absence of a global clock in connection with the distributed control results in a *fine-grained control structure* in the asynchronous design. This results in reduced power dissipation in both the datapath and the control that is harder to accomplish in synchronous design at reasonable costs. Finally, asynchronous design allows for *fine-grained behaviour in time* of the chip, which is advantageous in designing low-power peripherals, and in designs where idle-power is an important part of the total power.

8.3 Remaining issues

Testability is one of the main open fields for research in asynchronous circuits at the moment. The distributed control makes stopping the circuit during a test more cumbersome than in globally clocked circuits. Therefore the control of a scan-chain in the datapath, for example, is more difficult to design. Work in this area has been done, but a push-button test method in the VLSI-programming context is not implemented yet [32, 34, 33]. Scan-test in the context of micropipeline-based asynchronous circuits is discussed in [10].

The stuck-at fault model at the gate-level has been lifted to the handshake circuit level [48, 49]. A tool gives feedback on the test coverage for a given test at the level of Tangram. To obtain a higher coverage the VLSI-programmer can then adjust the test. For the 80C51 this method can be followed to obtain a test, which is discussed in Appendix A.

The current Tangram compiler delivers circuits that have a relatively large *execution time*. This is due to the fact that the compiler was designed to deliver low-power circuits. In the case of the DCC error decoder, speed was not an issue, as the synchronous solution did not use a high clock frequency either. In the case of the 80C51 we can make two observations. First, the 80C51 is not a high-performance machine; the standard version runs below 1 MIPS. Second, removing the redundant actions in the asynchronous 80C51 CPU, we obtain a variable execution time per instruction. This improves the speed compared to an asynchronous solution in which we would fully mimic the synchronous slot-scheme. There are opportunities to improve the speed of the circuits that are produced by the Tangram compiler. The matching of delays, for example, is done conservatively resulting in very robust circuits [30]. However, these delays can be tightened when we know more about the implementation and the layout, improving the speed of the circuit.

Appendix A

Testability

In Chapters 1 and 2 of this thesis we have seen that the parameters of the design space of an IC are area, execution time, energy dissipation, and testability. This thesis mainly concentrates on the first three parameters. In this appendix some aspects of testability are discussed. Testability is an essential requirement for an IC to be accepted for industrial production.

A.1 Background

Since the resurgence of interest in asynchronous design over the last decade, there has also been an interest in the testability of asynchronous circuits. An overview of these activities is given by Hulgaard et. al. [17]. In the Tangram project there have been two main activities in the area of testing.

The first activity aims at high-level development and evaluation of tests and is described in [48]. In this approach the stuck-at fault model as used in synchronous design is "lifted" to the level of handshake circuits. In this new model, a faulty circuit can be modeled by replacing a handshake-circuit component by another component that shows erroneous behaviour. The one-to-one correspondence between handshake circuits and the Tangram language makes it possible to reason about the test on the level of Tangram. Fault-coverage simulation can be done at the handshake level using the handshake fault-model, and a tool views the coverage results to the VLSI-programmer. The test is described as a Tangram program of the environment of the chip. The designer can then adjust the test to achieve a higher fault coverage. The handshake circuit simulator needs to determine whether a fault in a

data-component is observable on an output of the circuit. For the single-rail implementation of handshake circuits the defined handshake fault model includes stuck-at-output faults in the control logic and all stuck-at faults in the datapath [48, 49]. With the handshake fault-model it is possible to design a test for a circuit generated from a Tangram program.

The second testability-activity in the Tangram project aims at improving the testability of asynchronous Tangram-compiled circuits even further by implementing *Design-for-Testability* (DfT) . In this approach extra hardware is added to enhance the effect of test-methods. One of these methods is *scan test* in which all register-elements (latches or flipflops) are in test mode connected into one chain that can be read and written by the environment. *Partial scan* was used in the DCC error corrector chip and offers a trade-off between the cost for testing and the cost associated with scan-design [32]. The approach taken is to implement the extra hardware for the test enhancement in Tangram. This gives the designer the opportunity to implement the test facilities on the level of the VLSI-programming language.

Partial scan was implemented in the double-rail implementation of the DCC error corrector chip. Observability was for free in this chip, as incorrect functioning of the IC would manifest itself by deadlock. In the single-rail implementation of handshake circuits, however, observability is no longer for free as single-rail circuits are no longer quasi-delay-insensitive with respect to datapath operation. Put differently, a fault in the datapath needs no longer result in deadlock of the IC. Asynchronous scan facilities can then be used to take snapshots of the system states from time to time and thereby compensate for the lost observability-by-deadlock [32].

To increase the testability of single-rail handshake circuits further, the applicability of other existing test-methods for synchronous ICs on asynchronous circuits were investigated [34]. One of these methods stops the circuit at some points in time and measures the quiescent supply current I_{DDQ}. Stopping a synchronous circuit is straightforward as one can stop the clock. In asynchronous circuits the situation is different: there is no global clock and therefore it is more difficult to stop the operation of the circuit. To control the operation of the asynchronous circuit on a more fine-grained level to enhance the effect of the I_{DDQ} method, extra DfT can be added: the HOLD components as introduced in [34]. HOLD components make it possible to stop the operation of the circuit during testing to enhance the observability of the state of the circuit. This approach and its applications are described in [34].

The development of a test for an IC is a task of the designer. When an IC is developed using the Tangram design flow, the VLSI-programmer should design a

test for that IC. Therefore it is important that the VLSI-programmer can design a test on the level of Tangram, as described in the first testability activity in the Tangram project [48, 49]. Therefore, in the design of a test for the 80C51 we concentrate on the first approach, viz. the development of a test on the level of the Tangram language.

A.2 Approach

Currently there is no automated procedure for generating a test for an asynchronous Tangram-compiled circuit. Therefore we have to develop the test manually, though the method described in [48, 49] makes an automatic path possible in principle.

The model used in [48, 49] defines for each handshake component a *replacement set*. The elements in this set represent handshake components that show erroneous behaviour. A replacement component can be *restricted:* this handshake component shows a behaviour-trace that is a prefix of the behaviour of the original (non-faulty) handshake component, causing the circuit to deadlock. *Non-restricted* replacements show a different (and non-blocking) behaviour and are much harder to test, as the circuit does not deadlock and faults may not be externally visible.

Because the replacement sets of *control* components only contain restricted replacements, all faults can be tested by doing handshakes on *every* control channel. In other words: the test of the circuit should activate all paths on the handshake circuit control tree.

For the *datapath*, we distinguish between combinatoric circuitry and registers. Combinatoric circuitry in asynchronous single-rail circuits is not different from synchronous implementations, and therefore existing tools for test pattern generation can be used. For the registers we have to make sure that all register elements (flipflops or latches in the implementation) have assumed all possible values, 0 and 1, and that this has been made visible on an external output.

The 80C51 is a programmable architecture and therefore the test for an 80C51 can be described in terms of an assembly program containing instructions. The VLSI-programmer has to design a test in the form of an *environment* to the circuit, delivering the test-vectors. The environment of the 80C51 can be modeled as a *memory* that contains the test program.

The 80C51 is a modular system that consists of various blocks: CPU with memories, peripherals, and the synchronizer (Chapter 4). In the same modular fashion, a test can be designed for each block separately. We can design and implement a test

for the 80C51 blocks in the following way:

- For the *control*, we have to create a test that generates handshakes on all paths of the decode tree. For the CPU this implies that the test has to contain all 80C51 instructions.

- For the combinatoric circuitry test vectors have to be generated. As the structure of combinatoric circuitry in a synchronous circuit is similar to that of an asynchronous single-rail circuit, the same tools for generation of the test vectors can be used.

- For the registers in the datapath we have to make sure that all bits have assumed 0 and 1 during the test, and that this has been made visible to an external output. For example, we can "mimic" a scan-chain through the registers and route values through the "chain", making the result visible on an external port (for example Port 1 (`P1`)). This is expressed by the assembly program

```
MOV   A,#11111111
MOV   REG1,A
MOV   REG2,REG1
...
MOV   REGn,REGn1
MOV   P1,REGn

MOV   A,#00000000
MOV   REG1,A
MOV   REG2,REG1
...
MOV   REGn,REGn1
MOV   P1,REGn
```

In principle, the bus-construct as described in Chapter 5 enhances observability compared to point-to-point communication as all values pass through the same variable, viz. `bus`. This variable can be made observable to the environment using an external port. Using this approach of high-level test design, a test for the asynchronous 80C51 microcontroller was designed by Philips Semiconductors, with a coverage of 98% based on the handshake circuit fault model.

A.3 Example: a test for the UART

This section describes a test for the UART as described in Section 6.3, but it does not provide a test for the Special Function Register Interface.

For the *datapath* of the UART we observe that there are a few registers present (i.e. the shift-registers). There is no combinatoric circuitry present. For the registers we have to make sure that all bits assume the values 0 and 1 and that these values are made visible to the environment.

For the *control* we have to make sure that the test follows all paths of the control circuitry. There is quite some similarity between modes 1,2, and 3 of the UART: they only differ in the baud rate and the number of bits that is transmitted and received. In the Tangram program this is exploited by sharing the procedures for modes 1,2, and 3. In the test for the UART we can exploit this by taking the test for modes 1,2, and 3 together into one section. Mode 0 is different: there is a specific section in the Tangram program that deals transmission and reception in this mode. Therefore, also the test program for the UART contains a separate section for mode 0.

Transmission and reception follow the same protocol: the start-bit and the stop-bit are for both actions the same. Therefore we can test transmission and reception simultaneously, by connecting the serial transmission line `TxD` directly to the serial reception line `RxD`.

The reception part of the UART in modes 1,2, and 3 contains some extra circuitry to detect a start-bit and a stop-bit. This circuitry also has to be covered by the test.

The following test for the UART contains three parts:

1. Test mode 0:
   - transmit pattern `SBUF=<<10101010>>` (the CPU makes the UART to send this pattern);
   - receive pattern `<<01010101>>` (the environment makes the UART to receive this pattern).
2. Test mode 1, 2, and 3: output `TxD` is connected directly to input `RxD`. Patterns that are transmitted, are then received simultaneously:
   - send (and receive) pattern `<<SBUF,TB8>>=<<01010101,0>>` in mode 2 at slow baud rate;

- send (and receive) pattern <<SBUF,TB8>>=<<10101010,1>> in mode 2 at fast baud rate;
- send (and receive) pattern <<SBUF,TB8>>=<<01010101,0>> in mode 3 at the baud rate determined by the timer overflow;
- send (and receive) pattern <<SBUF,TB8>>=<<10101010,1>> in mode 1 at the baud rate determined by the timer overflow; note that in mode 1 the value of bit TB8 does not matter as mode 1 specifies an eight-bit UART.

3. Finally, the environment generates wrong start bits and stop bits to test the hardware that detects these faults in serial data communication:

   - generate the wrong stop bit, which should be 1, in mode 2 at fast baud rate;
   - generate two wrong start bits in mode 2 at fast baud rate:
     - offer bits 0,1,1 at counter states 7, 8, and 9 respectively;
     - offer bits 1,0,1 at counter states 7, 8, and 9 respectively.

The simulation of this test on the level of handshake circuit is shown in Figure A.1. The test yields 100% coverage according to the handshake fault-model. In the complete handshake simulation we see the parts of the test described above.

A.4 Review

Testability is a problem that can be addressed at the level of the VLSI-programming language: the test is seen as a Tangram description of the environment of the circuit. For the datapath of a circuit, test patterns have to be offered by the environment and the results have to be observable by the environment. As the datapath of an asynchronous system is comparable to a datapath in a synchronous system, a synchronous testing approach can be chosen. The control of an asynchronous system is different from that of a synchronous system. The control of a handshake circuit takes the structure of a tree, in which each path from root to any leaf must be activated in the test, to achieve a higher coverage. The VLSI-programmer must therefore have insight in the structure of the control to design a high-coverage test for the IC that is compiled from the VLSI-program.

At this moment high-level test generation has to be done by hand: the VLSI-programmer designs a test and obtains observability data from the test-tool [48, 49].

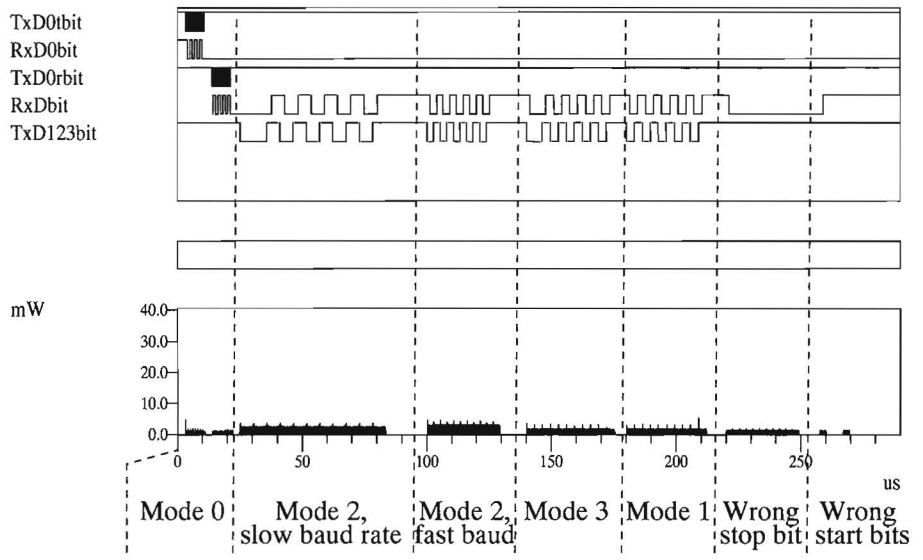


Figure A.1: Handshake simulation results of a test for the UART.

This path can in principle be automated to obtain an automatic generation of test patterns for a VLSI-design in Tangram. When this test does not yield a high enough coverage the design-for-test methods as described in [32, 34, 33] have to be applied to increase the coverage.

Appendix B

80C51 Instruction Set

In this table the opcode of the instructions at entry $P_i + +Z_j$, is ij where i and j are in hexadecimal format. Columns Z8 through ZF are taken together into one column, are are columns Z6 and Z7. The specification of the instructions can be found in the 80C51 Data Handbook (IC20) [4].

	Z0	Z1	Z2	Z3
P0	NOP	AJMP ad11	LJMP ad16	RR A
P1	JBC bit,ad8	ACALL ad11	LCALL ad16	RRC A
P2	JB bit,ad8	AJMP ad11	RET	RL A
P3	JNB bit,ad8	ACALL ad11	RETI	RLC A
P4	JC ad8	AJMP ad11	ORL dir,A	ORL dir,#data
P5	JNC ad8	ACALL ad11	ANL dir,A	ANL dir,#data
P6	JZ ad8	AJMP ad11	XRL dir,A	XRL dir,#data
P7	JNZ ad8	ACALL ad11	ORL C,bit	JMP @A+DPTR
P8	SJMP ad8	AJMP ad11	ANL C,bit	MOVC A,@A+PC
P9	MOV DPTR,#data16	ACALL ad11	MOV bit,C	MOVC A,@A+DPTR
PA	ORL C,-bit	AJMP ad11	MOV C,bit	INC DPTR
PB	ANL C,-bit	ACALL ad11	CPL bit	CPL C
PC	PUSH dir	AJMP ad11	CLR bit	CLR C
PD	POP dir	ACALL ad11	SETB bit	SETB C
PE	MOVX A,@DPTR	AJMP ad11	MOVX A,@R_i	
PF	MOVX @DPTR,A	ACALL ad11	MOVX @R_i,A	

	Z4	Z5	Z67	Z8F
P0	INC A	INC dir	INC @R_i	INC R_n
P1	DEC A	DEC dir	DEC @R_i	DEC R_n
P2	ADD A,#data	ADD A,dir	ADD A,@R_i	ADD A,R_n
P3	ADDC A,#data	ADDC A,dir	ADDC A,@R_i	ADDC A,R_n
P4	ORL A,#data	ORL A,dir	ORL A,@R_i	ORL A,R_n
P5	ANL A,#data	ANL A,dir	ANL A,@R_i	ANL A,R_n
P6	XRL A,#data	XRL A,dir	XRL A,@R_i	XRL A,R_n
P7	MOV A,#data	MOV dir,#data	MOV @R_i,#data	MOV R_n,#data
P8	DIV A,B	MOV dir,dir	MOV dir,@R_i	MOV dir,R_n
P9	SUBB A,#data	SUBB A,dir	SUBB A,@R_i	SUBB A,R_n
PA	MUL A,B		MOV @R_i,dir	MOV R_n,dir
PB	CJNE A,#data,ad8	CJNE A,dir,ad8	CJNE @R_i,#data,ad8	CJNE R_n,#data,ad8
PC	SWAP A	XCH A,dir	XCH A,@R_i	XCH A,R_n
PD	DA A	DJNZ dir,ad8	XCHD A,@R_i	DJNZ R_n,ad8
PE	CLR A	MOV A,dir	MOV A,@R_i	MOV A,R_n
PF	CPL A	MOV dir,A	MOV @R_i,A	MOV R_n,A

Bibliography

[1] Electronic Engineering Times, March 1995.

[2] Diesel User Manual - Version 1.0.1, 1.0.2. Technical report, Philips Electronic Design & Tools, 1996.

[3] 16-bit 80C51XA (eXtended Architecture) Micrcontrollers Data Handbook (IC25). Philips Semiconductors, 1997.

[4] 80C51-Based 8-bit Microcontrollers: Data Handbook (IC20). Philips Semiconductors, 1997.

[5] G. Birtwistle, Y. Liu, D. Spooner, John Aldwinckle, Ken Stevens, and Wanzhen Yu. Case Studies in Asynchronous Design. Part 1: AMM Architecture. Technical report, University of Calgary, 1993.

[6] Erik Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. PhD thesis, Carnegie Mellon University, 1991.

[7] Thomas D. Burd and Robert W. Brodersen. Energy Efficient CMOS Microprocessor Design. In *Proceedings of the 28th Annual HICSS Conference*, pages 288–297, January 1995.

[8] Philip B. Endecott. *SCALP: A Superscalar Asynchronous Low-Power Microprocessor*. PhD thesis, Department of Computer Science, University of Manchester, 1995.

[9] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N.C. Paver. AMULET2e: An Asynchronous Embedded Controller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, April 1997.

[10] S. B. Furber and O. A. Petlin. Scan Testing of Micropipelines. In *Proc. IEEE VLSI Test Symposium*, pages 296–301, May 1995.

[11] Steve Furber. *ARM System Architecture*. Addison-Wesley, 1996.

[12] Sonya Gary. Low-Power Microprocessor Design. In Jan M. Rabaey and Massoud Pedram, editors, *Low-Power Design Methodologies*, pages 255–288. Kluwer Academic Publishers, 1996.

[13] Ricardo Gonzalez and Mark Horowitz. Energy Dissipation in General Purpose Microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1283, 1996.

[14] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kauffman, 1990.

[15] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kauffman, second edition, 1996.

[16] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[17] Henrik Hulgaard, Steven M. Burns, and Gaetano Borriello. Testing Asynchronous Circuits: A Survey. *Integration, the VLSI journal*, 19(3):111–131, November 1995.

[18] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall PTR, 1992.

[19] Joep Kessels. VLSI-Programming of a Low-Power Asynchronous Reed-Solomon Decoder for the DCC Player. In *Asynchronous Design Methodologies*, pages 44–52. IEEE Computer Society Press, May 1995.

[20] Israel Koren. *Computer Arithmetic Algorithms*. Prentice Hall, 1993.

[21] Dake Liu and Christer Svensson. Power Consumption Estimation in CMOS VLSI Chips. *IEEE Journal of Solid-State Circuits*, 29(6):663–670, June 1994.

[22] Michael S. Malone. *The Microprocessor A Biography*. Springer-Verlag, 1995.

[23] Alain J. Martin. Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.

[24] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nystroem, Paul Penzes, Robert Southworth, and Uri Cummings. The Design of an Asynchronous MIPS R3000 Microprocessor. In *Advanced Research in VLSI*, September 1997.

[25] T. Nanya, A. Takamura, M. Kuwako, M. Imai, T. Fujii, M. Ozawa, I. Fukasaku, Y. Ueno, F. Okamoto, H. Fujimoto, O. Fujita, M. Yamashina, and M. Fukuma. TITAC-2: A 32-bit Scalable-Delay-Insensitive Microprocessor. In *Symposium Record of HOT Chips IX*, pages 19–32, August 1997.

[26] David A. Patterson and John L. Hennessy. *Computer Organization & Design - The Hardware / Software Interface.* Morgan Kauffman, 1994.

[27] Nigel Paver. *The Design and Implementation of an Asynchronous Microprocessor.* PhD thesis, Department of Computer Science, University of Manchester, 1994.

[28] Ad Peeters. The 'Asynchronous' Bibliography (BIBTEX) database file `async.bib`. `ftp://ftp.win.tue.nl/pub/tex/async.bib.Z`. Corresponding e-mail address: `async-bib@win.tue.nl`.

[29] Ad Peeters and Kees van Berkel. Single-Rail Handshake Circuits. In *Asynchronous Design Methodologies*, pages 53–62. IEEE Computer Society Press, May 1995.

[30] Ad M.G. Peeters. *Single-Rail Handshake Circuits.* PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1996.

[31] Christian Piguet et al. Low-Power Design of 8-b Embedded CoolRisc Microcontroller Cores. *IEEE Journal of Solid-State Circuits*, 32(7):1067–1078, 1997.

[32] Marly Roncken. Partial Scan Test for Asynchronous Circuits Illustrated on a DCC Error Corrector. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 247–256, November 1994.

[33] Marly Roncken, Emile Aarts, and Wim Verhaegh. Optimal Scan for Pipelined Testing: An Asynchronous Foundation. In *Proc. International Test Conference*, pages 215–224, October 1996.

[34] Marly Roncken and Erik Bruls. Test Quality of Asynchronous Circuits: A Defect-Oriented Evaluation. In *Proc. International Test Conference*, pages 205–214, October 1996.

[35] H.A.J.M. Seelen. Automatic Synthesis of Gated Clocks for Low Power. Master's thesis, Dept. of Electrical Engineering, Eindhoven University of Technology, October 1996.

[36] Peter Song. Asynchronous Design Shows Promise. *Microprocessor Report*, October 6 1997.

[37] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[38] Akihiro Takamura, Masashi Kuwako, Masashi Ima, Taro Fujii, Motokazu Ozawa, Izumi Fukasaku, Yoichiro Ueno, and Takashi Nanya. TITAC-2: An asynchronous 32-bit microprocessor based on Scalable-Delay-Insensitive model. In *Proc. International Conf. Computer Design (ICCD)*, pages 288–294, October 1997.

[39] Jim Turley. Cogency Pushes Asynchronous Logic. *Microprocessor Report*, October 6 1997.

[40] C. H. (Kees) van Berkel and Ronald W. J. J. Saeijs. Compilation of Communicating Processes into Delay-Insensitive Circuits. In *Proc. International Conf. Computer Design (ICCD)*, pages 157–162. IEEE Computer Society Press, 1988.

[41] Kees van Berkel. *Handshake Circuits. An Asynchronous Architecture for VLSI Programming*. Cambridge University Press, 1993.

[42] Kees van Berkel and Arjan Bink. Single-Track Handshaking Signaling with Application to Micropipelines and Handshake Circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 122–133. IEEE Computer Society Press, March 1996.

[43] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, and Frits Schalij. A Fully-Asynchronous Low-Power Error Corrector for the DCC player. In *International Solid State Circuits Conference*, pages 88–89, February 1994.

[44] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, and Frits Schalij. Asynchronous Circuits for Low Power: A DCC Error Corrector. *IEEE Design & Test of Computers*, 11(2):22–32, Summer 1994.

[45] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, Frits Schalij, and Rik van de Wiel. A Single-Rail Re-implementation of a DCC Error Detector Using a Generic Standard-Cell Library. In *Asynchronous Design Methodologies*, pages 72–79. IEEE Computer Society Press, May 1995.

[46] Kees van Berkel and Martin Rem. VLSI Programming of Asynchronous Circuits for Low Power. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 152–210. Springer-Verlag, 1995.

[47] Kees van Berkel, Hans van Gageldonk, Joep Kessels, Cees Niessen, Ad Peeters, Marly Roncken, and Rik van de Wiel. Asynchronous Does Not Imply Low power, But ... In A. Chandrakasan, editor, *IEEE Low Power Book (Reader)*, 1998.

[48] Rik van de Wiel. High-Level Test Evaluation of Asynchronous Circuits. In *Asynchronous Design Methodologies*, pages 63–71. IEEE Computer Society Press, May 1995.

[49] Rik van de Wiel. *Testing Handshake Circuits*. PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, To appear in 1998.

[50] Hans van Gageldonk. The Asynchronous Move Machine: Verification using CCS. Master's thesis, Dept. of Mathematics and Computing Science, Eindhoven University of Technology, August 1994.

[51] Hans van Gageldonk. VLSI-Programming for Low-Power Applications. In *ProRISC/IEEE Workshop on Circuits, Systems and Signal Processing*, pages 137–142, Mierlo, The Netherlands, 1996. STW, Technology Foundation.

[52] Hans van Gageldonk, Daniel Baumann, Kees van Berkel, Daniel Gloor, Ad Peeters, and Gergard Stegmann. An Asynchronous Low-Power 80C51 Microcontroller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1998.

[53] Tom Verhoeff. Delay-Insensitive Codes—An Overview. *Distributed Computing*, 3(1):1–8, 1988.

[54] Neil H.E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*. Addison-Wesley, second edition, 1993.

Index

Summary

Power consumption has become increasingly important, especially for hand-held battery-powered products. A large part of the power in these products is dissipated by the digital circuitry. Microcontrollers are examples of digital circuits that are widely used in embedded applications where low-power is an issue.

The main contribution of this thesis is a low-power implementation of one such microcontroller: the 8-bit 80C51 microcontroller. Another contribution is a demonstration of the use of a method with which low-power circuits can be designed: silicon compilation from a high-level programming language. This compilation is transparent, which makes it possible to reason about the characteristics of the circuit at the level of the programming language: area, execution time, energy dissipation, and testability. The method makes it possible to do a design-space exploration. An important aspect of the compilation is that it results in an *asynchronous* circuit, i.e. a circuit without a global *clock signal.*

Clocked circuits are also called *synchronous* circuits. The clock is a global control signal that causes activity to take place anywhere in the circuit, at specific points in time. *Asynchronous* circuits take a different approach: circuit elements activate each other by using local *handshaking* instead of global clocking. In principle, asynchronous circuits show activity only when and where necessary.

At Philips Research there have been investigations into the methodology to design asynchronous circuits using the approach of silicon compilation. A description in a programming language is translated into the netlist of an asynchronous circuit. The programming language is called Tangram, and has similarities to C and Pascal, with additional constructs to express communication, parallelism, and sharing of hardware. The process of writing programs in Tangram with the aim to generate efficient circuits in terms of area, execution time, energy dissipation, and testability is called *VLSI-programming.* The designer is assisted by a set of tools that give feedback on these aspects at the level of the programming language.

This thesis starts when the Tangram compiler and the associated toolset are available. The potential advantages of the Tangram approach were demonstrated on an error decoder for the DCC (Digital Compact Cassette) player, which shows a power advantage of a factor 6 at the penalty of 20% overhead in area. The asynchronous 80C51 microcontroller as described in this thesis served as vehicle to learn to exploit the transparent compiler, but also to transfer the Tangram technology from Research to Philips Semiconductors.

The synchronous 80C51 consists of two parts: the CPU and the peripherals. The CPU fetches and executes instructions from its program memory. The peripherals perform specific and small tasks, and cater for the communication between the CPU and the environment. Timers, counters, and the interrupt controller are examples of peripherals. *Derivatives* of the 80C51 architecture have the same CPU, but differ in the sizes and implementations of the memories, as well as in the peripherals. The synchronous 80C51 is analyzed for low-power opportunities, of which six are identified. With these in mind we develop an asynchronous low-power 80C51.

To enable modular design of the asynchronous 80C51, we adopt the same partition into CPU and peripherals. There is, however, an extra block: the *Synchronizer*. The CPU and the peripherals are designed asynchronously to maximally exploit the advantages of an asynchronous implementation for low-power. The Synchronizer is the only process that has a clock as input; it synchronizes with the CPU and the peripherals when they need a timing reference for correct functioning. An example of such a situation is *external memory access*, where a protocol is executed in which data validity with respect to a timing reference (i.e. a clock) is specified. In effect, the Synchronizer enables the designer to establish *timing compatibility* with synchronous environments when necessary. The CPU, the peripherals, and the Synchronizer communicate with each other using handshake channels.

The CPU can be split into the *datapath* and the *control*. The datapath contains the registers and arithmetic circuitry, and is steered by the control. In the datapath we can choose between point-to-point communication between the registers, or a bus-structure, in which there is an extra variable that all registers can write to and read from. In the bus-structure each communication takes place via the bus. The bus-structure is cheaper in area, but a point-to-point path can be more energy-efficient and faster. In the 80C51 only a few communication paths are used very frequently, and therefore a hybrid scheme is best for area, execution time, and energy dissipation of the datapath: a point-to-point path for frequent traffic (faster and more energy-efficient) and the bus for less-frequent traffic (cheaper in area). For the control we can also choose between two design alternatives: either decode the instruction completely and then execute the appropriate actions (centralized

decoding), or split the decoding into a few steps (distributed decoding). The 80C51 instruction set can be split into two parts that we call *regular* and *irregular*. It turns out that for the *regular* part distributed decoding is the best solution, while for the *irregular* part centralized decoding is. The distributed control structure enables the designer to remove the *redundant actions* that are present in the synchronous execution scheme of instructions, saving power in the datapath. Furthermore, the *asynchronous* character of the handshake control circuits also saves on power in the control, as only a few components are active at any point in time.

Peripherals are small blocks that communicate with the CPU and with the environment. Their activity is usually *demand-driven*, i.e. they only have to become active upon request, which is issued either by the CPU or by the environment. The CPU and the peripherals can be decoupled to ensure maximum progress and minimum power dissipation by the blocks. They communicate with each other using shared memory, viz. the Special Function Registers. To establish the decoupling, a new interface structure containing these Special Function Registers is introduced, and exemplified by the design of a peripheral for the 80C51: the UART (Universal Asynchronous Receiver and Transmitter), which takes care of serial communication of data between the CPU and the environment .

Early in the project the cooperation with Philips Semiconductors resulted in a demonstrator chip, containing the 80C51's functionality implemented using the Tangram silicon compiler. Since this chip the design space exploration for the 80C51 has been carried out, as described in this thesis. The chip is compared to a recent synchronous counterpart. With all improvements we show that it is possible to build an asynchronous 80C51 that is slightly slower and 30% larger, but uses four times less energy than its synchronous counterpart. This design is used in Philips pagers that are available on the world market today.

The design of an asynchronous Tangram-compiled 80C51 microcontroller shows how the transparent compilation scheme can be used to reason about the characteristics of the resulting circuit. It also demonstrates how a new architecture can be described in Tangram to exploit the potential power benefits of an asynchronous implementation.

Samenvatting

Het energieverbruik wordt vooral in draagbare producten die op een batterij werken, steeds belangrijker. Een groot deel van de energie die deze producten gebruiken wordt gedissipeerd door de digitale ICs in die producten. Microcontrollers zijn voorbeelden van digitale ICs die veel toegepast worden in producten waar een laag energieverbruik belangrijk is.

De belangrijkste bijdrage van dit proefschrift is een energie-zuinige implementatie van zo'n microcontroller: de 80C51 8-bit microcontroller. Een andere bijdrage is een demonstratie van het gebruik van een methode waarmee energie-zuinige circuits kunnen worden ontworpen: silicium compilatie vanuit een hoog-niveau programmeertaal. Deze vertaling is transparant, hetgeen het mogelijk maakt te redeneren over de eigenschappen van het circuit op het niveau van de programmeertaal: oppervlakte, executietijd, energieverbruik en testbaarheid. De methode maakt het mogelijk een exploratie van de ontwerpruimte uit te voeren. Een belangrijke eigenschap van de vertaling is dat zij resulteert in een *asynchroon* circuit, d.w.z. een circuit zonder centrale *klok*.

Circuits met een centrale klok worden ook wel *synchrone* circuits genoemd. De klok is een globaal signaal dat ervoor zorgt dat er overal in het circuit activiteit is op bepaalde tijdstippen. Asynchrone circuits gaan van een ander standpunt uit: elementen in het circuit activeren elkaar door het gebruik van locale *handshaking* in plaats van een globale klok. In principe vertonen asynchrone circuits alleen activiteit waar en wanneer dat nodig is.

Op het Natuurkundig Laboratorium van Philips wordt onderzoek gedaan naar het ontwerp van asynchrone schakelingen met behulp van silicium compilatie. Een beschrijving in een programmeertaal wordt vertaald naar de netlijst van een asynchroon circuit. De programmeertaal die hierbij gebruikt wordt heet Tangram en vertoont gelijkenissen met talen als C en Pascal, met extra constructen om communicatie, parallellisme, en gemeenschappelijk gebruik van hardware (*sharing*) te

kunnen uitdrukken. Het proces om programma's in Tangram te schrijven met als doel om efficiënte circuits te genereren in termen van oppervlakte, executietijd, energieverbruik en testbaarheid, heet *VLSI-programmeren*. De ontwerper wordt hierbij geassisteerd door diverse gereedschappen die informatie over deze aspecten geven in termen van de constructen van de programmeertaal.

Dit proefschrift begint als de Tangram vertaler met de gereedschappen die erbij horen, beschikbaar is. De potentiële voordelen van de Tangram benadering waren aangetoond met verscheidene proefontwerpen, waaronder een error decoder voor de Digitale Compact Cassette (DCC) speler, die 20% groter is maar zes keer minder energie verbruikt dan zijn synchrone tegenhanger. De asynchrone 80C51 microcontroller zoals beschreven in dit proefschrift diende als vehikel om te leren om de transparante vertaling uit te buiten, maar ook als medium om de Tangram technologie te introduceren bij Philips Semiconductors.

De synchrone 80C51 bestaat uit twee delen: de CPU (Central Processing Unit, de centrale verwerkingseenheid) en de periferie. De CPU haalt instructies op uit het programma geheugen en voert ze uit. De periferie bestaat uit een aantal blokjes die ieder een kleine en goed omschreven taak uitvoeren. Bovendien nemen de perifere blokken de communicatie met de omgeving van de chip voor hun rekening. Voorbeelden van perifere blokken zijn timers en counters, en de interrupt controller. *Afgeleiden* van de 80C51 architectuur hebben allen dezelfde CPU, maar zij verschillen in de afmetingen en implementatie van de geheugens en in de perifere blokken. In dit proefschrift wordt eerst een analyse gedaan van de synchrone 80C51 waarbij er zes punten worden geïdentificeerd waar energie zou kunnen worden bespaard. Deze zes punten dienen als uitgangspunt bij het ontwerpen van een asynchrone energie-zuinige 80C51 architectuur.

Om modulair ontwerpen van de asynchrone 80C51 mogelijk te maken gaan we uit van dezelfde opdeling als in het synchrone geval: CPU en periferie. Er is echter een extra blok: de *Synchronizer*. De CPU en de periferie worden asynchroon ontworpen om de voordelen van een asynchrone implementatie voor energie-zuinige chips zoveel mogelijk uit te kunnen buiten. De Synchronizer is het enige proces dat een klok als invoer heeft; zij synchroniseert met de CPU en de perifere blokken wanneer die een tijds-referentie nodig hebben. Een voorbeeld van zo'n situatie is *extern geheugen acces*, waar een protocol uitgevoerd moet worden dat data-geldigheid op externe poorten in relatie tot een kloksignaal aangeeft. De Synchronizer maakt het de ontwerper mogelijk om tijd-compatibiliteit met (bestaande) synchrone omgevingen te bewerkstelligen. De CPU, de perifere blokken en de Synchronizer communiceren met elkaar door middel van handshake communicatie.

De CPU bestaat uit een *datapad* en een *control*. Het *datapad* bevat de registers en de schakelingen die rekenkundige operaties kunnen uitvoeren, en wordt bestuurd door de control. In het datapad kunnen we kiezen uit punt-tot-punt communicatie tussen de registers, of een bus-structuur waarbij er een extra register is waar alle registers naar toe kunnen schrijven en van kunnen lezen. Iedere communicatie in de bus-structuur gebruikt de bus. De bus-structuur is goedkoper in oppervlakte, maar punt-tot-punt communicatie kan energie-zuiniger en sneller zijn. In de 80C51 worden slechts enkele communicatie paden zeer frequent gebruikt, terwijl de andere paden weinig gebruikt worden. Dit maakt een tussenoplossing aantrekkelijk: punt-tot-punt paden voor frequent data-verkeer (snel en energie-zuinig) en de bus voor de andere paden (kleine oppervlakte). Ook voor de *control* kunnen we kiezen uit twee ontwerp varianten: de instructie eerst decoderen en dan acties uitvoeren in het datapad (centrale decodering), of de decodering opsplitsen in kleine stapjes (gedistribueerde decodering). De 80C51 instructieset kan gesplitst worden in twee delen die we *regulier* en *irregulier* noemen. Het reguliere deel kan het beste met een gedistribueerde decodering gedecodeerd worden, en het irreguliere deel met een gecentraliseerde decodering. De gedistribueerde structuur van de control maakt het zo mogelijk om redundante acties in de synchrone implementatie te elimineren zodat er energie in het datapad wordt bespaard. Bovendien wordt energie bespaard door het *asynchrone* karakter van de control, doordat er slechts enkele delen van de control actief zijn op een bepaald tijdstip.

Perifere blokken communiceren met de CPU en met de omgeving van het IC. Hun activiteit wordt meestal bepaald door een gebeurtenis in de omgeving of in de CPU. De CPU en de periferie kunnen ontkoppeld worden om zo maximale voortgang van de blokken alsmede minimaal energieverbruik mogelijk te maken. Zij communiceren met elkaar door middel van een gemeenschappelijk geheugen, namelijk de Speciale Functie Registers (SFRs). Om de ontkoppeling mogelijk te maken wordt er in dit proefschrift een nieuwe interface-structuur tussen CPU en perifere blokken voorgesteld die de SFRs bevat. Dit wordt toegelicht door middel van het ontwerp van een perifeer blok: de UART (Universal Asynchronous Receiver and Transmitter), een blok dat zorg draagt voor serieel data verkeer tussen de CPU en de omgeving.

Vroeg in het project is er een testchip gefabriceerd, in samenwerking met Philips Semiconductors. Deze chip implementeert de functionaliteit van de 80C51 met behulp van de Tangram silicium compiler. Na de fabricage van de chip is de exploratie van de ontwerpruimte uitgevoerd, zoals beschreven in dit proefschrift. Met de gevonden verbeteringen is het mogelijk om een asynchrone energie-zuinige 80C51 te maken die iets trager en 30% groter is, maar vier keer minder energie gebruikt dan zijn synchrone tijdgenoot. Dit ontwerp wordt gebruikt in Philips pagers die op

dit moment op de wereldmarkt verkrijgbaar zijn.

Het ontwerp van de asynchrone 80C51 in Tangram demonstreert het gebruik van de transparante vertaling naar silicium om over de eigenschappen van het circuit te redeneren. Het laat ook zien hoe een nieuwe architectuur in Tangram is te beschrijven zodat de potentiële voordelen van een asynchrone implementatie kunnen worden uitgebuit.

Curriculum Vitae

Hans van Gageldonk was born on September 3rd, 1971 in Heerlen, The Netherlands. After attending the Rombouts College Atheneum in Brunssum, he started his study Computing Science at Eindhoven University of Technology in September 1989. In 1993 he was co-organizer of a study tour to Japan. His M.Sc. thesis was about formal verification of asynchronous circuits using the process algebra CCS. The work for this thesis was carried out at the University of Manchester, U.K., in the first half of 1994.

After gradation he started to work towards a Ph.D. at September 1st, 1994. As a Ph.D.-student at Eindhoven University, he spent most of his time at Philips Research Eindhoven. This research resulted in this thesis, on which he expects to receive the degree at September 1st, 1998.

From October 1st, 1998, Hans will be working at Philips Research Laboratories Eindhoven as a research scientist.

Titles in the IPA Dissertation Series

The State Operator in Process Algebra
J. O. Blanco
Faculty of Mathematics and Computing Science, TUE, 1996-1

Transformational Development of Data-Parallel Algorithms
A. M. Geerling
Faculty of Mathematics and Computer Science, KUN, 1996-2

Interactive Functional Programs: Models, Methods, and Implementation
P. M. Achten
Faculty of Mathematics and Computer Science, KUN, 1996-3

Parallel Local Search
M. G. A. Verhoeven
Faculty of Mathematics and Computing Science, TUE, 1996-4

The Implementation of Functional Languages on Parallel Machines with Distrib. Memory
M. H. G. K. Kesseler
Faculty of Mathematics and Computer Science, KUN, 1996-5

Distributed Algorithms for Hard Real-Time Systems
D. Alstein
Faculty of Mathematics and Computing Science, TUE, 1996-6

Communication, Synchronization, and Fault-Tolerance
J. H. Hoepman
Faculty of Mathematics and Computer Science, UvA, 1996-7

Reductivity Arguments and Program Construction
H. Doornbos
Faculty of Mathematics and Computing Science, TUE, 1996-8

Functorial Operational Semantics and its Denotational Dual

D. Turi
Faculty of Mathematics and Computer Science, VUA, 1996-9

Single-Rail Handshake Circuits
A. M. G. Peeters
Faculty of Mathematics and Computing Science, TUE, 1996-10

A Systems Engineering Specification Formalism
N. W. A. Arends
Faculty of Mechanical Engineering, TUE, 1996-11

Normalisation in Lambda Calculus and its Relation to Type Inference
P. Severi de Santiago
Faculty of Mathematics and Computing Science, TUE, 1996-12

Abstract Interpretation and Partition Refinement for Model Checking
D. R. Dams
Faculty of Mathematics and Computing Science, TUE, 1996-13

Topological Dualities in Semantics
M. M. Bonsangue
Faculty of Mathematics and Computer Science, VUA, 1996-14

Algorithms for Graphs of Small Treewidth
B. L. E. de Fluiter
Faculty of Mathematics and Computer Science, UU, 1997-01

Process-algebraic Transformations in Context
W. T. M. Kars
Faculty of Computer Science, UT, 1997-02

A Generic Theory of Data Types
P. F. Hoogendijk
Faculty of Mathematics and Computing Science, TUE, 1997-03

The Evolution of Type Theory in Logic and Mathematics
T. D. L. Laan
Faculty of Mathematics and Computing Science, TUE, 1997-04

Preservation of Termination for Explicit Substitution
C. J. Bloo
Faculty of Mathematics and Computing Science, TUE, 1997-05

Discrete-Time Process Algebra
J. J. Vereijken
Faculty of Mathematics and Computing Science, TUE, 1997-06

A Functional Approach to Syntax and Typing
F. A. M. van den Beuken
Faculty of Mathematics and Informatics, KUN, 1997-07

Ins and Outs in Refusal Testing
Lex Heerink
Faculty of Computer Science, UT, 1998-01

A Discrete-Event Simulator for Systems Engineering
G. Naumoski and W. Alberts
Faculty of Mechanical Engineering, TUE, 1998-02

Scheduling with Communication for Multiprocessor Computation
Jacques Verriet
Faculty of Mathematics and Computer Science, UU, 1998-03

An Asynchronous Low-Power 80C51 Microcontroller
Hans van Gageldonk
Faculty of Mathematics and Computing Science, TUE, 1998-04

Stellingen

behorende bij het proefschrift

An Asynchronous Low-Power 80C51 Microcontroller

van

Hans van Gageldonk

Technische Universiteit Eindhoven
September 1998

1. Een krachtige VLSI-programmeertaal in combinatie met een transparante vertaling maakt een gestructureerde exploratie van de ontwerpruimte van een digitaal IC mogelijk.

   [Literatuur] Dit proefschrift, hoofdstuk 5 en 6.

2. Het is mogelijk om een asynchrone 80C51 microcontroller zodanig te ontwerpen dat hij, qua functionaliteit en timing, door een omgeving niet van een synchrone tegenhanger te onderscheiden is.

   [Literatuur] Dit proefschrift, hoofdstuk 4.

3. Het asynchroon ontwerpen van een CISC CPU nodigt uit tot het elimineren van redundante acties in de executie van instructies, ten gunste van het energieverbruik.

   [Literatuur] Dit proefschrift, hoofdstuk 5.

4. Handshaking maakt het mogelijk om circuit-blokken te ontkoppelen in de tijd, ten gunste van het energieverbruik.

   [Literatuur] Dit proefschrift, hoofdstuk 6.

5. Asynchroon ontwerpen levert vaak nieuwe inzichten op die synchroon ook toegepast kunnen worden, maar waar het synchrone ontwerptraject niet op een natuurlijke wijze langs loopt.

6. Het is mogelijk om een pipelined RISC processor compact in Tangram te beschrijven; de huidige compiler die Tangram naar een circuit vertaalt is echter niet gericht op het genereren van snelle circuits, en daarom is het op dit moment niet mogelijk om een snelle RISC processor te maken in Tangram.

7. De 8051 architectuur is waarschijnlijk de op een na meest gebruikte microprocessor architectuur tot op heden.

   [Literatuur] Michael S. Malone: The Microprocessor A Biography. Springer Verlag, 1995.

8. Gereedschappen voor de formele verificatie van hardware kunnen steeds grotere ontwerpen verifiëren en zullen daardoor in een industriële omgeving steeds meer gebruikt gaan worden.

9. In een tijd waarin steeds meer ontwerpen gemaakt worden waarin goede keuzes gemaakt moeten worden voor zowel hardware als software zullen universiteiten aan beide onderwerpen aandacht moeten besteden in de onderwijsprogramma's.

10. Als bij kinderen met leesproblemen geruime tijd wordt getraind in woordlezen dan ontstaat er automatische woordherkenning; hierdoor kunnen zij sneller gelijkende woorden herkennen.

    [Literatuur] Mariken de Wolf: Afstudeerscriptie.
    Vrije Universiteit Amsterdam, 1998.

11. Een promotie-onderzoek als samenwerking tussen een universiteit en een bedrijf vereist dat er goede afspraken tussen de beide partijen gemaakt worden, die dan ook nageleefd moeten worden.

12. Een goedkope piano wordt door handelaren vaak "studie-piano" genoemd; de kwaliteit van zo'n piano belemmert de studievoortgang echter zodanig dat "studie-piano" daarmee een twijfelachtig begrip is.