POWER-EFFICIENT EMBEDDED PROCESSING

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2005

By
Yijun Liu
School of Computer Science

Contents

List of Tables

List of Figures

Abstract

As more and more transistors and functionality are integrated in single chips, power consumption has become one of the most important design parameters in modern embedded circuits. The purpose of the work described in this dissertation is to identify ways to reduce the power consumption of embedded systems. Low-power design is a complex task requiring care at all levels of the design hierarchy. In this dissertation, the focus is mainly on the following low-power techniques:

- Exploring asynchronous logic design for its low-power potential. The power-efficiencies of asynchronous and synchronous designs are compared. Different asynchronous design issues are also discussed in terms of their power-efficiency.

- Circuit-level optimizations to reduce the power consumption of function units, including adders and multipliers, and memory. An asynchronous carry-lookahead adder and a pipelined iterative multiplier are presented, both of which are designed based on analyses of their input data characteristics. The circuit-level design issues of a low-power embedded SRAM macro are also presented.

- Architecture-level optimizations to reduce the execution overheads of soft-programmable processors. A hierarchical processing architecture is proposed based on an analysis of embedded processing programs. A RISC-like coprocessor has been designed to demonstrate the power-efficiency of a hierarchical processing architecture. A dataflow coprocessor has also been designed which is more power-efficient and faster than the RISC-like coprocessor.

Together these results demonstrate that there is scope for improvements in power-efficiency at several different levels in the design hierarchy, underlining the need to treat low-power design as a holistic process.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Acknowledgements

The work leading to this dissertation was done during my three years as a post-graduate student in the APT group in the School of Computer Science at the University of Manchester, UK. These years have been very stimulating and instructive. The friendly people and atmosphere in the group has greatly helped the work described in the thesis.

First of all I would like to give my deepest thanks to my supervisor, Professor Steve Furber, whose support and guidance over the post few years have been invaluable for this work.

Special thanks are due to Dr. Jim Garside, my advisor, for his very beneficial advice on many different problems. Special thanks are also due to Dr. Steve Temple and Dr. Viv Woods not only for their patience in proofreading the draft of the thesis but also for their kind help with CAD tools and my English.

Many thanks to everyone else in the APT group for their support, encouragement and friendship.

I would also like to acknowledge with gratitude the support of an ORS grant from Universities UK and a research scholarship from the School of Computer Science.

to my family

Chapter 1

Introduction

The exponential development of CMOS technology [7] makes it possible to embed high-performance data processing units in portable and wireless devices such as cell phones, personal digital assistants (PDAs), multi-media players and sensor network applications. The high data processing ability allows these devices to support a wide range of functions and wireless communications, making these devices popular and necessary for daily life. The worldwide market for digital signal processing (DSP) has grown to 7.8 billion dollars per annum and is developing at an annual growth rate of 30%. Of these DSP chips, 71.5% are used in wireless applications, mostly in cell phones [4]. The rapid growth of the wireless market also inspires the growth in the functionality of PDAs, digital cameras and multi-media players. The increasing prominence of portable and wireless devices has become one of the major impetuses that drives CMOS VLSI design.

Although performance is still a very important issue in the design of wireless and portable devices, it is not the only concern. Battery life, package size, device weight and, of course, cost are also important metrics. The power consumption of the CMOS circuits in these devices has a direct relation to these factors. Wireless and portable applications call for low-power design.

Figure 1.1 shows the power consumption growth of battery-powered chips predicted by the International Technology Roadmap for Semiconductors (ITRS) [8]. ITRS predicts that after 2012, the power consumption of battery-powered chips will be maintained at 3 watts. However, this prediction is not reliable if the power consumption of VLSI chips is maintained only by CMOS technology scaling (although it is the major contributor) without addressing low-power design techniques. Moreover, the demand to prolong the operation of portable devices

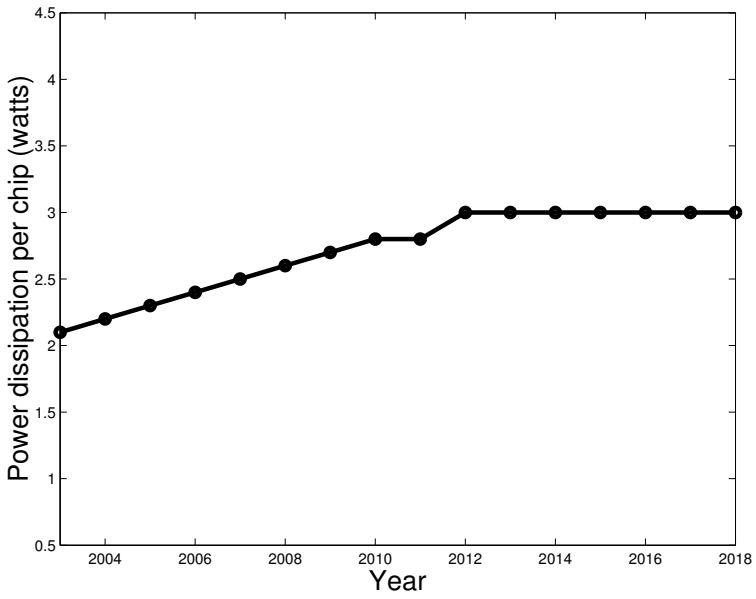


Figure 1.1: Trends in the power consumption of battery-powered chips [4]

between charges will never stop. Some new applications bring forward even more 'critical' requirements for circuit power-efficiency. For example, sensor network designers suggest that a small battery (maybe an "AA"-size battery) should power the whole device life of a sensor node [9]. Low-power techniques are extremely important in the design of the CMOS circuits in these applications. Low-power techniques also allow devices to operate without thermal management, resulting in a cheaper package and smaller size. High power consumption affects the reliability of CMOS circuits, so low-power techniques improve the robustness of devices as well.

As the requirements of portable and wireless devices shift from high performance to high portability and long battery life, a rethinking of design flow and techniques is necessary. For embedded data processing circuits, the 'performance first' CMOS design philosophy needs to be changed. Power consumption concern should be considered as one of the most important metrics in CMOS circuit design together with performance and area considerations. This thesis will address low-power techniques for data processing especially in battery-powered embedded applications.

The remainder of this chapter presents the research goals and major contributions. It concludes with an overview of the rest of the thesis.

1.1 Research goals and contributions

It would be impossible to include all low-power processing design issues in a single thesis. Instead, low-power asynchronous logic, power-efficient arithmetic units, low-power memory design and a coprocessor architecture based on the characteristics of embedded processing applications are the emphases of this thesis. The key research contributions include:

- Evaluating the contribution of asynchronous logic to the design of low-power data processing;
- The development of a set of power-efficient arithmetic function units, including an iterative pipelined multiplier and an asynchronous carry-lookahead adder. The design of these functional units begins with the analysis of the characteristics of input vectors; these guide the power-saving techniques used in the arithmetic units.
- The design of a low-power embedded SRAM macro, which adopts a novel dual-rail row decoder and a low-swing write-voltage scheme.
- The development of a hierarchical architecture which aims to reduce the power consumption of most commonly executed short code segments, such as small loops and long equation evaluations. Several architectures are evaluated in terms of power-efficiency, including conventional RISC architectures and a dataflow architecture using a novel synchronization scheme. The experimental results demonstrate an order of magnitude of improvement in power-efficiency over current general-purpose processors.

The research described in this thesis has led to the following publications:

- "The design of an asynchronous carry-lookahead adder based on data characteristics", Proceedings of PATMOS 2005, Springer, Lecture Notes in Computer Science.
- "A Low Power Embedded Dataflow Coprocessor", Proceedings of the 2005 International Symposium on VLSI System Design.
- "The Design of a Low-Power Asynchronous Multiplier", Proceedings of the 2004 International Symposium on Low Power Electronics and Design.

- "Minimizing the Power Consumption of an Asynchronous Multiplier", Proceedings of PATMOS 2004, Springer, Lecture Notes in Computer Science.
- "A Transistor-Level Delay-Insensitive Register File for Deep Sub-micron SoC", Proceedings of the Embedded Systems Show, October, 2004.
- "The Design of a DI adder using 1-of-4 code", Proceedings of the 15th UK Asynchronous Forum, Cambridge, January, 2004.
- "A Low-Power Asynchronous multiplier", Proceedings of the IEE Seminar on SoC Design, Test and Technology, September, 2003.

1.2 Thesis overview

Chapter 2 reviews the fundamentals of low-power CMOS design. It analyses the power consumption of conventional processors and gives an overview of low-power processing techniques.

Chapters 3 to 5 focus on more detailed low-power techniques for low-level CMOS circuit implementations, including:

Chapter 3 addresses low-power techniques in a specific field — asynchronous design. It briefly introduces basic asynchronous logic design; it then compares the power-efficiency of synchronous and asynchronous design; finally, low-power asynchronous logic techniques are discussed.

Chapter 4 presents the designs of two power-efficient arithmetic units, an adder and a multiplier, based on the analysis of the input vector characteristics of arithmetic units.

Chapter 5 addresses the design issues of a low-power embedded RAM macro.

Chapters 6 and 7 discuss architecture-level design issues aiming at reducing the power consumption of soft programmable embedded processors, including:

Chapter 6 presents an analysis of embedded processing programs and proposes a hierarchical processing scheme based on a CPU-coprocessor architecture. The coprocessor can do a lot of the work of a general-purpose processor and is more flexible than a conventional coprocessor which supports only a limited set of functions, such as floating-point calculations. The coprocessor need not support those complex functions of a general-purpose processor, such as a wide range of addressing modes and instructions, and in this way power overheads due to complex control and instruction decoding can be avoided. Chapter 6 analyzes

the power savings that can be achieved by using different RISC-like architectures and their performance implications.

Chapter 7 studies the power-efficiency of another less-studied strategy — a dataflow architecture. Detailed issues of a low-power dataflow architecture are addressed in this chapter and the advantages and disadvantages of dataflow and conventional RISC architectures are compared.

Chapter 8 provides the conclusions of the thesis and directions for future work.

Chapter 2

Low power design

This chapter presents a review of circuit-level low-power CMOS design techniques. The sources of power dissipation in CMOS circuits, fundamental dynamic power saving techniques and power-efficiency metrics are briefly described. The power dissipation of conventional data processing units — microprocessors — is analyzed which gives the direction for further power saving. The chapter also gives a survey of low-power processing techniques.

2.1 Power dissipation sources in CMOS circuits

There are four sources of power consumption in a CMOS circuit, as described in the following equation [10]:

$$P_{total} = P_{dynamic} + P_{leakage} + P_{short} + P_{static}$$

where P_{total} is the total power consumption of the circuit, being the sum of four factors: $P_{dynamic}$ — the dynamic power consumption, $P_{leakage}$ — the leakage power consumption, P_{short} — the short-circuit power consumption and, P_{static} — the static power consumption.

- Dynamic power consumption

  The dynamic power consumption of a CMOS circuit results from the charging and discharging of the capacitances of wires and transistors. The dynamic power consumption of a CMOS gate is equal to:

$$P_{dynamic} = \frac{1}{2} \cdot C_{load} \cdot V_{dd}^2 \cdot N \cdot f \tag{2.1}$$

where C_{load} is the overall capacitance of its output node and the gate(s) it drives; V_{dd} is the supply voltage; N is the average number of transitions in one clock cycle; f represents the switching frequency of the clock. The dynamic power consumption of a CMOS gate depends on how frequently the gate is switched. If a gate does nothing, there is no dynamic power dissipation. Since CMOS circuits normally run at very high clock frequencies, the dynamic power consumption is the dominant factor in the overall power consumption of CMOS circuits (This situation may change when using deep sub-micron CMOS technologies). Methods to reduce dynamic power consumption are the emphasis of this thesis.

- Leakage power consumption

  Leakage power dissipation has two components: the reverse-bias diode leakage current at transistor drains and the sub-threshold current through a turned-off transistor channel. Leakage power dissipation is one of the proportions of overall power dissipation not caused by switching activity. Leakage current exists even when a circuit is idle. Therefore, although the leakage current of a single transistor is very small, the total effect of the leakage current can be significant in a big chip. Moreover, the leakage increases rapidly when the threshold of the devices drops. Deep sub-micron CMOS circuits normally have very low threshold voltages. Therefore, leakage is becoming another significant factor of power dissipation in these circuits. Although minimizing leakage power dissipation is an important aspect of CMOS power saving, it is a general problem, normally not depending on the different data being processed. Reducing leakage power consumption is outside the scope of this thesis. More detailed information on leakage power reduction can be found elsewhere [11] [12].

- Short-circuit power consumption

  Ideally, in a complementary CMOS gate, the pull-up pMOS network and the pull-down nMOS network should not conduct at the same time. However, in practice, there is a short period during each switching transition when both the pMOS and nMOS networks conduct, allowing a current to flow directly from V_{dd} to ground. This current is called the short-circuit current. If the input transition is slow, the duration of the short-circuit current becomes correspondingly longer, resulting in significant short-current power

dissipation. However, good design will avoid slow edges and can keep short-circuit power dissipation to a small fraction of the dynamic power.

- Static power consumption

  For traditional complementary static circuits using full voltage swing, there is no static power consumption caused by a constant static current flow because, during any operation, either the pMOS or the nMOS network is closed. However, in circuits using other CMOS logic styles, a constant static current may exist; pass-transistor logic and pseudo-nMOS logic are two examples. For a pass-transistor circuit, the low output voltage-swing of a pass-transistor gate (such as a weak 1 after an n-pass-transistor) may cause both the p-transistor network and the n-transistor network after the pass-transistor gate to conduct, allowing a static current from Vdd to ground. A pseudo-nMOS logic gate contains a single p-transistor, whose gate is always connected to ground; when the pull-down nMOS network conducts, a static current exists between Vdd and ground.

2.2 Dynamic power-saving techniques

Equation 2.1 revealed the three degrees of freedom inherent in dynamic power-saving techniques: supply voltage, physical capacitance and switching activity.

- Voltage reduction:

  Because of its quadratic relationship to power, voltage reduction offers the most effective means of minimizing power consumption. The power-saving techniques due to supply voltage reduction are as follows:

  - More advanced CMOS technologies need lower supply voltages and hence using these CMOS technologies contributes to low-power VLSI design.
  - If the speed of a circuit using a standard CMOS supply voltage is fast enough, the circuit can be powered by a voltage lower than the standard supply voltage and may still meet the throughput and peak performance requirements of applications. If the supply voltage is too low to meet the performance requirements, some dedicated techniques

can be used to speed up the circuit, such as parallelism and pipelining [13].

- Capacitance reduction:

  Dynamic power consumption depends on the switched capacitance, thus minimizing the effective capacitance which is charged and discharged also results in power saving.

  CMOS technology scaling greatly contributes to reducing the power consumption of CMOS circuits. Technology scaling reduces the capacitance of transistors and wires. It also allows devices to have a low supply voltage while maintaining their performance because devices have a low threshold voltage.

  Using the same CMOS technology, capacitance can be kept at a minimum by using simple circuit implementations, more efficient logic styles, fewer transistors, or shorter interconnect wires.

- Switching activity reduction:

  The goal of reducing switching activity is to reduce the average number of transitions in a clock cycle (N in Equation 2.1). Switching activity reduction techniques can be separated into two categories: minimizing unwanted activity and minimizing the required activity.

  Ideally, the gates of a CMOS circuit should only switch when performing useful tasks. However, in real designs, unwanted transitions unavoidably exist. For example, in a synchronous circuit, all registers are linked to a global clock signal and their gate loads are forced to charge and discharge in every clock cycle even when they have no valid inputs. These unnecessary transitions contribute to make the global clock one of the largest loads in synchronous circuits. One efficient way to minimize the unnecessary transitions is by gating off sub-blocks which perform no required activity [14].

  Another source of unwanted switching activity is glitches; these are spurious transitions which occur before a node settles down to its final steady-state value. One example of a glitch is the sum signals of a carry-propagate adder. Because of the chain of carry signals, some intermediate states occur at the outputs of the sum signals and cause a lot of unnecessary switching activity inside the adder. A more serious situation is when these output

glitches are propagated through the whole pipeline and cause a sequence of unwanted activity. It is difficult to get rid of all glitches in a circuit, but the number and their effects can be minimized. An efficient way to minimize the impact of glitches is to isolate them within a pipeline stage by opening pipeline registers for a short period only when data is to be admitted.

Using a simpler and more efficient implementation to achieve the same logic function is the basic idea in minimizing the required activity. There are often many ways to implement a logic function and an efficient way should be found for power saving. To minimize the power consumption of an existing CMOS circuit, data representations, signal encoding schemes and algorithms need to be reconsidered in terms of power-efficiency.

2.3 Power breakdown of microprocessors

Analyzing the power breakdown of processors indicates directions which potentially lead to power savings. Figure 2.1 shows the power breakdown of an embedded low power RISC microprocessor — a StrongARM microprocessor [5], which is a general-purpose, 32-bit microprocessor with a 16KB instruction cache, a 16KB write-back data cache, a write buffer, and a memory management unit combined in a single chip. It implements the ARM V4 instruction set and is designed for low-power and low-cost applications.

Although Figure 2.1 shows the power breakdown of one particular processor, it displays a general characteristic of processor power breakdown. There is little difference in power breakdown between the StrongARM processor and some other processors, for example, a CISC Pentium-Pro processor [15], a high-performance RISC processor — Alpha 21264 [16] and an asynchronous RISC ARM — the Amulet3 processor [17]. The actual power consumptions of the various processors differs, but their power breakdowns are similar. As can be seen from Figure 2.1, four parts contribute most to the overall processor power dissipation and may have the greatest power-saving potential; these are as follows:

- Timing control:

  For a synchronous processor, the power consumption of the timing control is the power dissipated in its global clock. Figure 2.1 shows that the global clock uses as much as 10% of the overall power, but this includes only the

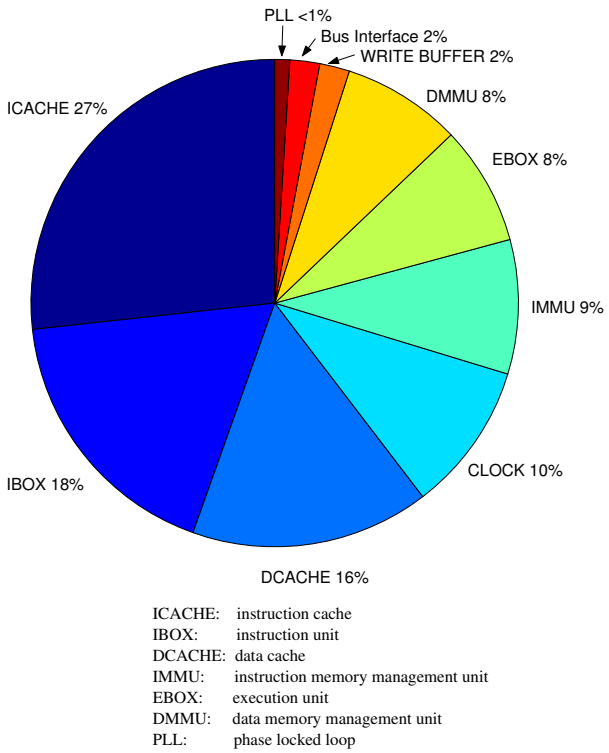


Figure 2.1: The power breakdown of a StrongARM processor [5]

power dissipated in the main clock distribution network and its drivers. If the local clock power for every sub-block is extracted and summed, it is as high as 25% of the total chip power [5]. For some Alpha processors, the timing control power contributes 40% to the overall power consumption [16].

For asynchronous processors, timing control is accomplished by 'local handshakes'. The contribution of handshakes and delay-matching circuits to overall processor power consumption is still very high as discussed in the next chapter.

- Memory access:

  Memory stores instructions and data and, ideally, a processor should have a large memory to store a lot of instructions and data. A larger memory implies more power consumption and a longer access latency. Modern processors usually have large memories which use significant power. The StrongARM processor with only 32KB on-chip cache uses 43% of the overall chip power in its caches, most of which is used by the instruction cache. A further 17% of the overall power is indirectly used by memory — the

memory management units for data and instructions (DMMU and IMMU).

- Instruction operations:

  A major part of the hardware involved in the execution of instructions is the instruction decoder. Modern processors have a wide range of instructions to make them more 'friendly' to programmers and to support these, the instruction decoder becomes a complex circuit implementation to translate instructions to control signals. Many instructions are rarely used, but design complexities are still needed to support them. Instruction operation hardware also includes some other parts, including program counters and address incrementers. The power used by instruction operations of the StrongARM is included in the IBOX sector of Figure 2.1, which contributes 18% to the overall power consumption.

- Register file:

  Processors refer to register files in almost every clock cycle, reading operands and writing results, resulting in a high switching activity. Moreover, to support high performance, register files normally have multiple read and write ports, resulting in a larger switching capacitance than a similar-sized single-port SRAM. These two factors make the register file a power-hungry component. The power used by the register file in Figure 2.1 is included in the EBOX sector. According to the power data from the Amulet3, 15% of the chip power is used by its register file [17].

It can also be seen from power analyses of microprocessors that the "real work" — data processing — uses only a very small proportion of the overall processor power. This is a common problem in all "soft-programmable" data processors. In these processors, data processing is guided by a set of 'commands' — instructions, so that every data processing operation is accompanied by at least one instruction. Instructions indicate not only data values but also the addresses of the data values. Therefore, much more power is dissipated in control flow than data processing. A pure RISC processor has a fixed instruction-length and a load-store architecture, which require a larger code size to describe a given task (poor code-density) and more power for instruction fetching and data transfer than many CISC processors.

2.4 Low power processing techniques

As can be seen from the previous analysis, the most effective way to design a low power data processor is to minimize execution overheads, including timing control power, memory (including the register file) power and the number of instructions needed to describe programs.

2.4.1 Minimizing timing control power

Reducing the clock frequency can minimize timing control power but is not the choice of many CMOS designers since this approach slows down the processor. Another way to minimize timing control power is to reduce the switching capacitance attached to clock signals.

Clock switching capacitance can be reduced by using more efficient registers [18] (for example, the registers for PowerPC and StrongARM), or by using both edges of clock signals to control dual-edge-triggered flip-flops (DETFFs) [19]. Reducing clock switching power can also be done by using a half-swing clocking scheme at the cost of a longer flip-flop delay [20].

Minimizing clock capacitance can also be achieved by using clock gating [21], here gating is used to separate a circuit into several blocks, each of which has its own local timing regenerator with a gating signal. The gating signal can gate the local timing regenerator off when the corresponding block has no required activity, so the clock capacitance of the block does not contribute to the overall capacitance when inactive. The global clock controls only the local timing generator drivers and thus has a much smaller capacitance than when used to drive the whole chip. However, clock gating also has some disadvantages which will be discussed in Section 3.2.1. Asynchronous logic is claimed to have the advantage of providing natural fine-grain clock gating. Asynchronous and synchronous clock gating schemes are compared in Section 3.2.1.

2.4.2 Minimizing memory power consumption

A number of low-power memory techniques have been proposed. Among them, the most important may be the 'principle of locality' [3], which directly leads to memory hierarchy design. Locality is a fundamental property of programs and is of two types as follows:

- Temporal locality — Recently accessed data and instructions are likely to be accessed again in the near future.
- Spatial locality — Instructions fetched by processors are likely to have adjacent addresses and data are also adjacent to each other.

Locality observations influence memory hierarchy architectures. Modern computers employ a large amount of storage memory and the larger the storage memory, the slower it is and the more power is needed to access it. The access latency of a hard disc, the largest capacity memory, is several milliseconds, which is clearly not compatible with the few nanoseconds occupied by a processor in executing an instruction. Clearly, a unified memory architecture does not meet the requirement for high performance and hence a multi-level memory hierarchy must be used to close the processor-memory performance gap.

A register file can be regarded as the lowest level of a memory hierarchy, which stores current operands and some temporary variables. A register file is usually very fast — few hundred picoseconds latency, while its size is normally relatively small — for example 32 words or 1K bits. The next hierarchy level may be a small on-chip memory block called a 'cache' used to store recently used instructions and data; this is also fast — several nanoseconds access latency— and small — several dozens of kilobits. Other levels of memory hierarchy may be an off-chip cache, a main memory and a hard disc. The latencies of adjacent levels change by factors of about 10 and the sizes of adjacent levels change by factors of about 1000.

A memory hierarchy not only increases performance but also reduces power consumption. Since accessing a smaller memory needs less power, if most of the data processing operations of a processor are done within a small memory, significant power can be saved.

Since the concept of memory hierarchy was introduced in the 1940s, improving the efficiency of memory hierarchy architecture has attracted a lot of researchers. Bajwa et al. [22] and Lea et al. [23] proposed a hardware addition called a 'loop cache' or 'loop buffer' to minimize the power used by refetching instructions in small loops. The idea of the loop cache scheme was based on two observations as follows:

- For embedded applications, power consumption due to instruction access is typically higher than that due to data access because one data access may

be accompanied by 3-4 instruction accesses. Hence, reducing instruction fetching can result in power saving.

- The dynamic execution traces of embedded programs are dominated by small program loops containing a small number of instructions (32 or fewer).

Based on the principle of locality, embedding a small instruction cache to hold the small program loops can greatly reduce overall processor power consumption, since a loop cache can be of a very small size and be situated near the processor. Loop cache schemes have been reported to reduce by 37.9% the main instruction cache accesses and to save the same proportion of instruction power without any delay penalty [23].

Bajwa and Lea's scheme handles only the innermost small program loops which do not contain if-else statements. Wu and Hwang improved Lea's scheme by proposing a loop cache with a stack-based architecture, which supported nested loops containing if-else statements [24]. A further 40% reduction in instruction power was reported compared to Lea's scheme.

To summarise, a loop cache is another level of memory hierarchy, possibly below the on-chip instruction cache. The rationale for loop caches is based on a characteristic of embedded programs — the domination of program loops with a small number of instructions.

2.4.3 Increasing code-density

Processors with a high code-density use fewer or smaller instructions to describe a given program, thus the power used by instruction fetching is reduced. Increasing processor code-density can be achieved in several ways, such as:

- Minimize redundant space and reduce instruction lengths, such as the ARM Thumb-1, Thumb-2 and MIPS16 instruction sets [25];

- Support more addressing modes and instruction set architectures, such as memory-memory and register-memory architectures [26], so separate address calculation instructions in a load-store (register-register) architecture can be saved.

However, increasing code-density means more complex instruction decoders, resulting in more hardware, lower decoding speed and higher decoder power.

Therefore, changing an instruction set needs careful consideration of the trade-offs between these factors.

2.4.4 'Hard-controlled' processing

A carefully designed 'hard-controlled' circuit is usually much more power-efficient than a soft programmable processor. A specifically-tailored ASIC can get rid of execution overheads such as temporary variable transfer, instruction fetching and decoding. It executes only useful data processing operations and thus has the highest power-efficiency. Even in general-purpose processors, the idea of using a direct hardware implementation to replace a set of instructions to perform a logic function is also used, an example is the 'count leading zeros' (CLZ) instruction in the ARM V5 instruction set [1]. The problem with ASICs is that, once they are fabricated, there is little possibility to change functionality.

Reconfigurable processors [27] offer an intermediate approach between ASICs and general-purpose processors, potentially achieving power efficiency by direct hardware mapping, while maintaining a higher flexibility than an application-specific circuit. Although reconfigurable processors are claimed to offer greater flexibility than ASICs, it is very difficult to design a reconfigurable hardware architecture having the flexibility and general-purpose processing ability close to that of a soft programmable processor.

2.5 Metrics for power efficiency

Energy- or power- efficiency is always linked to performance, so a brief introduction to performance metrics is necessary. Two circuits (or processors), say A and B, are appointed the same task. If A finishes the task before B, A is regarded as a faster processor. However, since A and B may have different instruction sets and are designed for different applications, it may be difficult to find identical programs that can be executed in both A and B. *MIPS* (million instructions per second) is used as a metric to compare the performance of processors; it is the rate of operations per unit time and is defined as follows:

$$MIPS = \frac{Number\ of\ Instructions\ executed}{Execution\ time} \times 10^{-6}$$

Some metrics can be used to measure the energy-efficiency of processors. One

popular metric is *MIPS/watt* (MIPS/W) which means how many instructions a processor can execute with the energy supply of 1 joule. Since

$$MIPS/watt = \frac{10^{-6} \cdot instructions/seconds}{joules/seconds} = \frac{10^{-6}}{joules/instructions}$$

a reciprocal metric of *MIPS/watt* is *energy per instruction* which means on average how much energy a processor needs to execute an instruction. Since

$$energy\,per\,instruction = \frac{joules}{instructions} = \frac{joules/seconds}{instructions/seconds} = power \times delay$$

an equivalent metric of *energy per instruction* is *power delay product*.

MIPS/W and its equivalent metrics are very important in battery-powered applications and they define with a given battery energy how many tasks processors can execute. However, these metrics define only energy-efficiency. Two processors, A and B, may have the same energy-efficiency but A is faster than B; these metrics cannot differentiate between the two processors. Another metric—$MIPS^2/watt$ is used to measure the compromise between energy efficiency and performance. Since

$$MIPS^2/watt = \frac{instructions^2/seconds^2}{joules/seconds}$$

$$= \frac{1}{seconds/instructions \cdot joules/instructions} = \frac{1}{energy \times delay}$$

Energy delay product (EDP) is a reciprocal metric of $MIPS^2/watt$.

$MIPS^3/watt$ is another important metric to measure energy efficiency.

$$MIPS^3/watt = \frac{MIPS}{energy \cdot delay} = \frac{1}{energy \cdot delay^2}$$

Since $energy \propto power \cdot delay \propto C \cdot V_{dd}^2 \cdot (1/delay) \cdot delay \propto V_{dd}^2$ and $delay \propto 1/V_{dd}$, $MIPS^3/watt$ is independent of the supply voltage V_{dd}. Thus it can be used to evaluate CMOS processors when voltage-scaling is taken into account and to compare processors using different supply voltages.

The similar metric $E \times D^n (n \geq 3)$ is also used depending on the importance of the delay increase caused by an energy reduction technique.

As an alternative to using these metrics, a fairer way to compare the power-efficiencies of two processors is to define several 'standard programs' (benchmarks), execute these benchmarks in both processors (which may have different instructions, so reprogramming may be needed), and then compare the energy used. Choosing 'typical' and 'fair' benchmarks is another difficulty. The benchmarks should represent the common characteristics of the applications for which the processors are designed.

2.6 Summary

This chapter has presented the sources of power dissipation in CMOS circuits and techniques to reduce dynamic power consumption. Following the power breakdown of a StrongARM and other processors, this chapter pointed out the four functions (or components) that contribute most to overall processor power consumption: timing control, memory access, instruction operations and the register file. The analysis provides the directions leading to the low-power processing techniques to be introduced in the remainder of this thesis — Chapter 4 addresses how to minimize timing control power; Chapter 5 presents low-power memory techniques and Chapters 6-7 discuss low-power instruction and register file techniques. This chapter also reviewed existing low-power processing techniques and addressed the metrics for power-efficiency.

The next chapter will discuss low-power techniques in a specific field — asynchronous logic design.

Chapter 3

Power-efficient asynchronous design

This chapter explores asynchronous logic design for its low-power potential. A brief introduction to asynchronous logic design is given in Section 3.1. The power-efficiencies of synchronous and asynchronous designs are compared in Section 3.2. Different low-power asynchronous logic techniques are addressed in Section 3.3. It is impossible to cover the whole field of asynchronous logic design and the chapter will focus on low-power asynchronous design. More detailed information on asynchronous design can be found elsewhere [28] [29].

3.1 Introduction to asynchronous logic design

Synchronous logic has dominated digital circuit design for decades because a global clock assumption makes circuits easier to understand and simpler to design. As an alternative design methodology, there has been a resurgence of interest in asynchronous logic design in both industrial and academic circles since the 1990s. Synchronous circuits use a globally-distributed clock signal to provide timing references and to synchronize data transfers. Asynchronous circuits, on the other hand, utilise no global signal; dataflow and synchronization are controlled using locally-generated 'handshake' signals. The difference in timing control gives asynchronous logic inherent properties which may be exploited to advantage:

- Low power consumption:

Asynchronous logic provides a natural fine-grain clock gating, which dissipates power 'on demand'. When idle, asynchronous circuits can achieve zero standby dynamic power consumption.

- High operating speed:

  Theoretically, the run-time latency of an asynchronous circuit is determined by the sum of the total local latencies rather than the global worst-case latency. However, for pipelined designs, the throughput of an asynchronous circuit depends on the slowest pipeline stage(s). In some operation cycles, however, it may achieve a lower latency than its synchronous counterparts.

- Avoidance of clock skew problems:

  Asynchronous logic replaces a global distributed clock network with multiple locally generated signals which have shorter wire lengths. This reduces the area and design complexity (as well as power) and avoids the need to provide the required stability and low skew of a global clock distribution network.

- Low emission of electro-magnetic noise:

  All components in a synchronous circuit switch simultaneously with the edges of the global clock. This phenomenon concentrates the radiated energy emission of a circuit at the harmonic frequencies of the clock, thus maximising electro-magnetic noise. Asynchronous design works in a different way. Different parts of a circuit are controlled by discrete local timing signals and, as a result, asynchronous circuits produce broadband distributed interference spread across the electromagnetic spectrum at a much lower level.

In the remainder of this section, a brief introduction to asynchronous logic is presented, including: asynchronous timing strategies, asynchronous delay models, data encodings, handshake protocols, and asynchronous pipeline latch controllers.

3.1.1 Asynchronous timing strategy

It takes some time for a gate to generate valid outputs and for a wire to propagate a digital signal. The period of transfer and evaluation is referred to as 'delay' and the phenomenon of unpredictable results due to the existence of delays is called

a 'hazard'. A 'glitch' is another phenomenon that should be avoided. Between two stable valid outputs, a combinatorial circuit may produce many spurious transient states. These unexpected transient states are referred to as 'glitches'. To avoid hazards and glitches, there must be some dedicated timing signals to indicate when signals are stable and valid. Synchronous design and asynchronous design differ in the ways they define these timing signals.

A global clock is used as a timing reference in a synchronous design to define the validity of data items. A synchronous circuit can be viewed as a finite-state machine with registers holding the current states; it changes from one state to another on the edges of the global clock. A typical synchronous pipeline circuit is illustrated in Figure 3.1. The transitions of the global clock signal (the falling or rising edges) indicate the validity of the outputs from the combinatorial logic blocks. The clock period must be no shorter than the delay of the slowest logic block to ensure valid outputs.

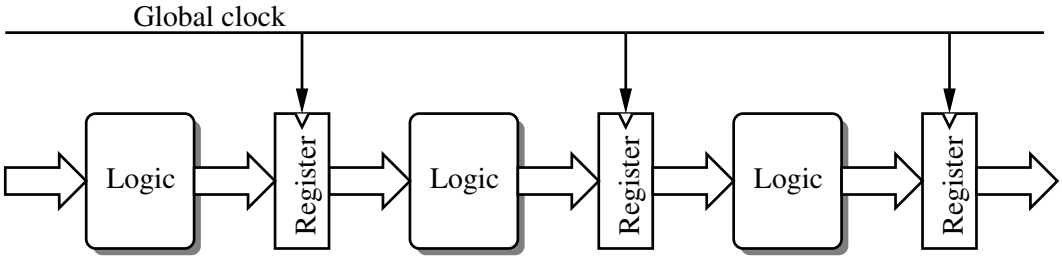


Figure 3.1: A synchronous pipeline circuit

Asynchronous logic, on the other hand, normally uses *Request-Acknowledge* handshake signals to indicate the validity and acceptance of data items. Figure 3.2 illustrates an asynchronous pipeline circuit, in which several pipeline stages are separated by pipeline registers or latches. The latch controllers (LT) control the states of the pipeline registers so that a data item can be propagated through the datapath as soon as possible without overwriting preceding items. The synchronization between two pipeline stages utilises a handshake protocol, one commonly used, the 4-phase bundled-data protocol, is shown in the figure. Other asynchronous handshake protocols are presented in the remainder of this section.

Using a 4-phase handshake protocol, the communication between the 'sender' and the 'receiver' proceeds through a sequence of actions as follows:

- The sender puts a valid data item on a bus and issues a logic high on the

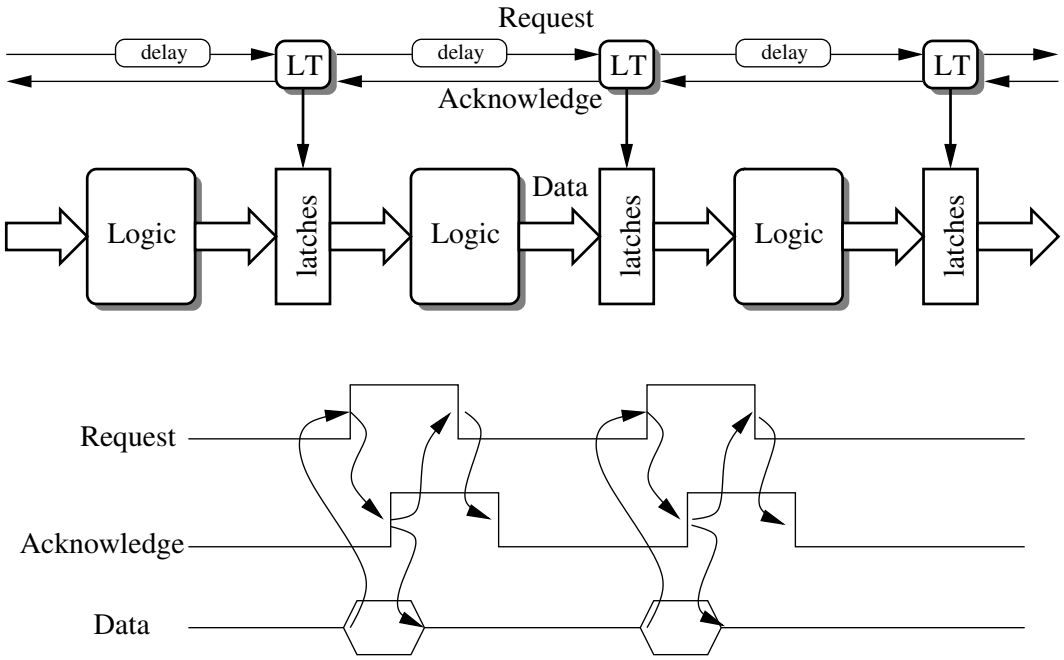


Figure 3.2: An asynchronous pipeline circuit

Request wire;

- The receiver accepts the data item when it is ready to do so, then it issues a logic high on the *Acknowledge* wire;
- The sender removes the data item and pulls the *Request* low;
- The receiver then initializes the next communication by pulling low the *Acknowledge* signal.

3.1.2 Delay models

To obtain valid data in digital circuits, assumptions must be made regarding wire and gate delays. All synchronous circuits use a *bounded-delay model*, where the delays in both the gates and wires are assumed to be within a bounded range. However, it becomes more and more difficult to set a bounded-delay range for a modern VLSI CMOS circuit, where the wire delays are no longer ignorable compared with gate delays. Asynchronous logic offers more flexible delay models than synchronous logic and allows *unbounded* delay assumption on gate or/and wire delays. In an *unbounded-delay model*, the delays can be any finite value.

Asynchronous circuits using an unbounded-delay model can be classed as being speed-independent, quasi-delay-insensitive or delay-insensitive depending on the delay assumptions made:

- Delay-insensitive (DI) [30]

  A delay-insensitive circuit operates correctly in spite of unknown delays in wires and gates. Although delay-insensitive circuits are the most robust, the constraints on delay-insensitive circuits are very restrictive [31], and the strict timing assumptions make their implementation the most expensive in terms of in hardware and power consumption.

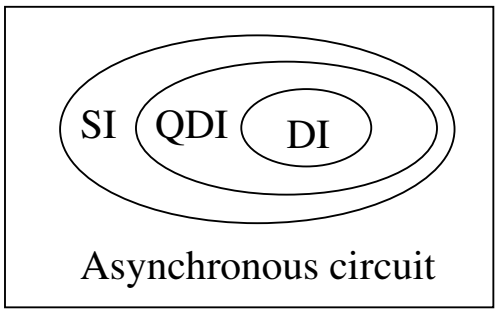
- Quasi-delay-insensitive (QDI) [31]

  Quasi-delay-insensitive circuits are delay-insensitive circuits augmented with isochronic forks [31]. Isochronic forks are sets of interconnecting wires that have delays whose differences are negligible compared with gate delays.

- Speed-independent (SI) [32]

  A speed-independent circuit is an asynchronous circuit that operates correctly regardless of unknown gate delays, while wire delays are assumed to be negligible compared with gate delays.

The relation and distinction between DI, QDI and SI circuits are illustrated in Figure 3.3.



Relation

	gate delay	wire delay
DI	unbounded	unbounded
QDI	unbounded	unbounded + isochronic fork
SI	unbounded	small

Delay assumptions

Figure 3.3: The relations and delay assumptions of asynchronous circuits using unbounded-delay models

3.1.3 Data encodings

In asynchronous design, timing can either be indicated by some dedicated delay-matching blocks (such as an inverter chain) or be extracted from data items that hold extra timing information by using specific data-encoding schemes. Asynchronous logic uses different data-encoding schemes as follows:

- Bundled-data encoding

  A bundled-data encoding is also called a single-rail encoding because it uses a single wire per bit, just as in synchronous circuits. The data items of a single-rail circuit cannot indicate any timing information themselves; a delay-matching block is needed to indicate the validity of the outputs from a bundled-data circuit. A typical asynchronous circuit using a bundled-data encoding was introduced in Figure 3.2, where the delay in the control matches the logic delay.

- Code-data encoding

  A codeword using a code-data encoding contains both a data value and timing information. An asynchronous circuit using a code-data encoding may need no delay-matching blocks and can be delay-insensitive. Examples of code-data encodings are [30]: Berger encoding and N-of-M encoding. This section describes two kinds of N-of-M encoding: dual-rail encoding and 1-of-4 encoding.

  With a code-data encoding, timing information can be obtained by inserting an 'empty value' between two 'valid values'. A completion detector is used to detect whether data items are 'empty' or 'valid'. A circuit using a code-data encoding is shown in Figure 3.4, where $valid = 1$ means the outputs are valid and, $valid = 0$ means the outputs are empty. An empty value separates the current value from the previous one. As can be seen from a comparison of Figure 3.2 and 3.4, no delay blocks are needed to match the delays of the combinatorial logic blocks in Figure 3.4; the speed of the asynchronous circuit is not fixed but self-adaptive and input-relative.

  The most commonly used code-data encoding is dual-rail encoding. A dual-rail encoding uses two wires ($d.f$ and $d.t$) to represent one bit of data. Figure 3.5 (a) shows a dual-rail encoding scheme. $d.f$ is used for signaling logic 0 and $d.t$ is used for logic 1. Figure 3.5 (b) shows the schematic of a completion

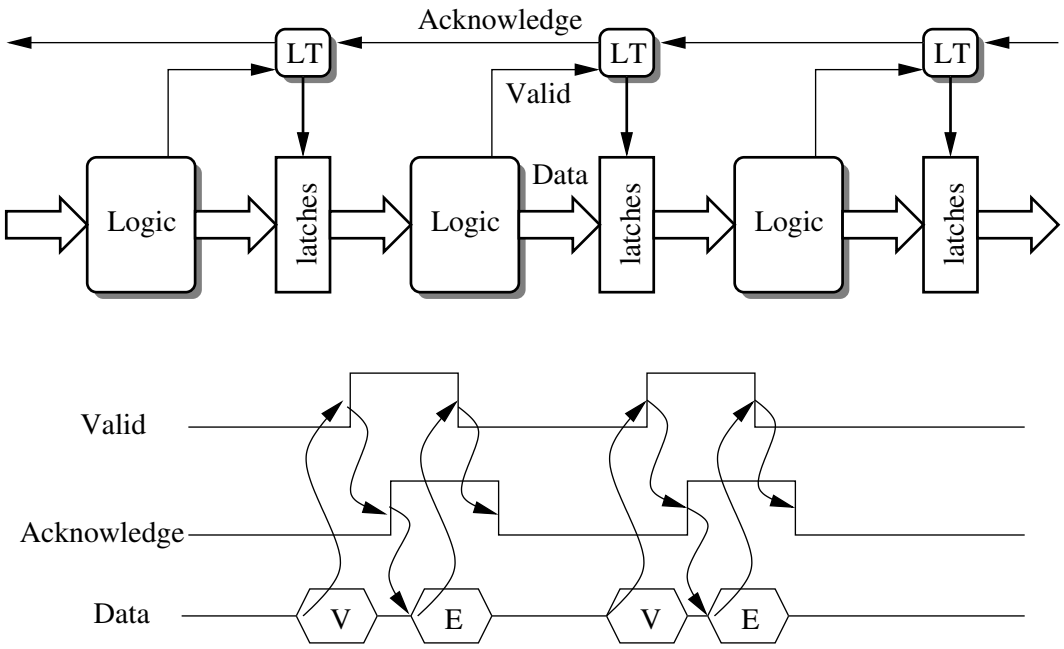


Figure 3.4: An asynchronous pipeline using a code-data coding

detector for 2-bit dual-rail data items which uses a 2-input symmetric C-gate [33], an important element in asynchronous design, whose schematic and truth table are shown in Figure 3.6. The output of 1 indicates that both of its inputs are 1s and the output of 0 indicates both of its inputs are 0s. If the two inputs are not equal, the C-gate will not change the output — its 'state'. N-input OR-gates can be used for detecting the states of a 1-of-N encoding. OR-gates and C-gates are usually used together to construct completion detector circuits. The C-gate in Figure 3.6 (a) is symmetric because inputs a and b control both the rising and falling transitions of the output. A C-gate can also be asymmetric if one or more of its inputs controls only one transition of its output, as shown in Figure 3.6 (b).

1-of-4 encoding is another N-of-M encoding; it uses four wires ($d0$, $d1$, $d2$, $d3$) to represent a two-bit data item. Figure 3.7 shows the encoding scheme of 1-of-4 encoding and its completion detector. The completion detector is a 4-input OR-gate.

3.1.4 Handshake protocols

Asynchronous handshake protocols differ in their ways of interpreting handshake events. Two commonly used handshake protocols are presented here.

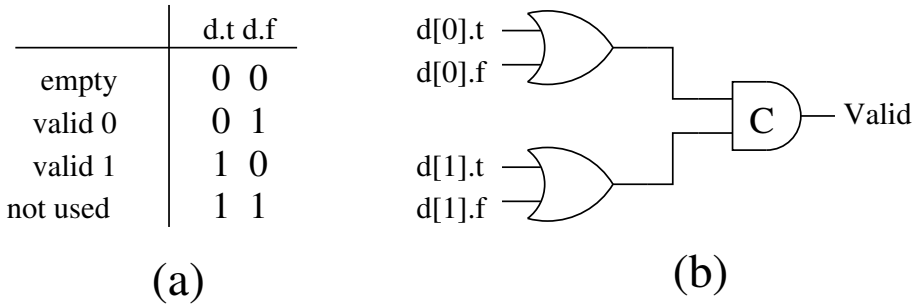


Figure 3.5: A dual-rail encoding scheme and its completion detector

- 2-phase protocol:

  A 2-phase protocol treats rising and falling transitions as having the same meaning. A 2-phase protocol does not return to zero to initialize another handshake, so it is also called a 'non-return-to-zero' protocol. Figure 3.8 shows the data validity scheme of a 2-phase protocol where a transition on the *Request* line indicates the validity of data. After receiving the data, the receiver sends a transition on the *Acknowledge* line telling the sender to release the data and send another data item. Since level-sensitive CMOS circuits are much easier to implement than transition-based ones, 2-phase protocols are not very commonly used in asynchronous design.

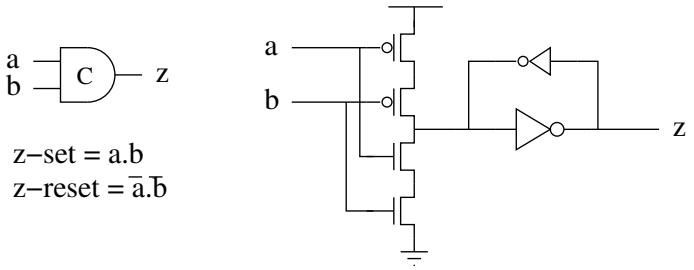
- 4-phase protocol:

  A 4-phase protocol uses levels rather transitions to denote events as shown in Figure 3.2. 4-phase protocols have return-to-zero transitions, so they are also called 'return-to-zero protocols'.

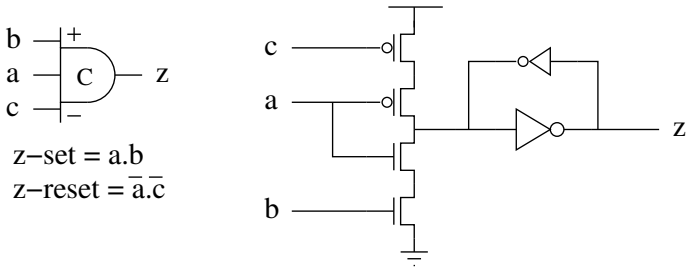
  With a 2-phase protocol, data are valid between two adjacent transitions. In a 4-phase protocol, however, the return-to-zero operations result in several different data validity schemes — early, broad, late and reduced broad; these are shown in Figure 3.9.

- Push and pull protocols:

  Asynchronous handshake protocols are either push or pull depending on whether the sender or the receiver initiates the communication. If the sender initiates the communication in a handshake protocol, the protocol is called a push protocol; otherwise, it is called a pull protocol.



(a) a symmetric C−gate



(b) an asymmetric C−gate

Figure 3.6: Example logic symbols and schematics of symmetric and asymmetric C-gates

To sum up, three characteristics define an asynchronous protocol: data encoding, 2- or 4-phase, push or pull. For example, a protocol can be a 4-phase bundled-data push or a 2-phase dual-rail push protocol. For 4-phase protocols, another parameter must be added — data validity; it can be early, broad, late or reduced broad.

3.1.5 Asynchronous pipeline latch controllers

In asynchronous design, pipeline latch controllers play a very important role in controlling the handshakes between two consecutive pipeline stages. 2- and 4-phase handshake protocols were introduced in the last section. These handshake protocols need to be supported by different pipeline latch controllers.

- 2-phase latch controller:

  A 2-phase handshake protocol uses transitions to identify events as illustrated in Figure 3.10 which shows a 2-phase latch controller as given by Sutherland [34]. The 2-phase latch controller is constructed with a C-gate,

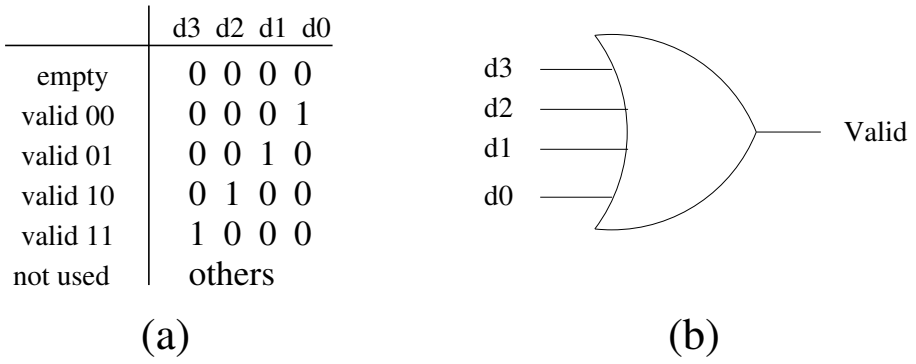


Figure 3.7: A 1-of-4 encoding scheme and its completion detector

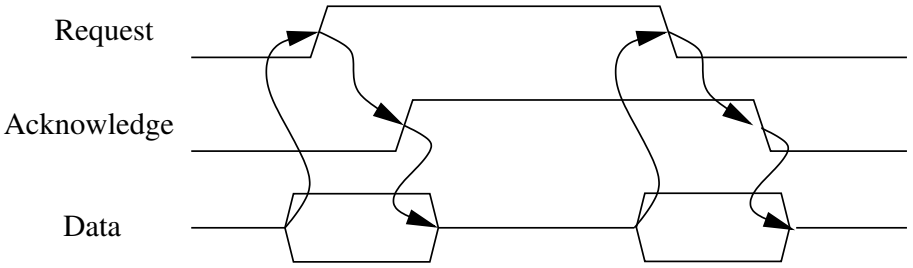


Figure 3.8: The validity scheme of a 2-phase protocol

an XOR-gate and a 'toggle'. The toggle steers events (transitions) to its outputs alternatively starting with the dot. The XOR-gate and the toggle together form a 2- to 4-phase protocol converter. The schematic is very neat and clearly identifies the concept of 2-phase logic. However, the 'toggle' is large and dominates the hardware cost of the controller.

- 4-phase latch controller

  4-phase level-sensitive circuits are commonly used in asynchronous designs and need 4-phase latch controllers. A 4-phase latch controller proposed by Furber and Day [35] is shown in Figure 3.11 (a) and its STG [36] description is shown in Figure 3.11 (b). The dashed arrows indicate orderings which are maintained by the environment; the solid ones represent orderings which the circuit itself ensures.

  One advantage of this controller is that the control signal 'LT' can be used to latch input data in level-sensitive latches which present only half the capacitive load and use only half the power of their edge-triggered equivalents. Hence the 4-phase latch controller is good for low power.

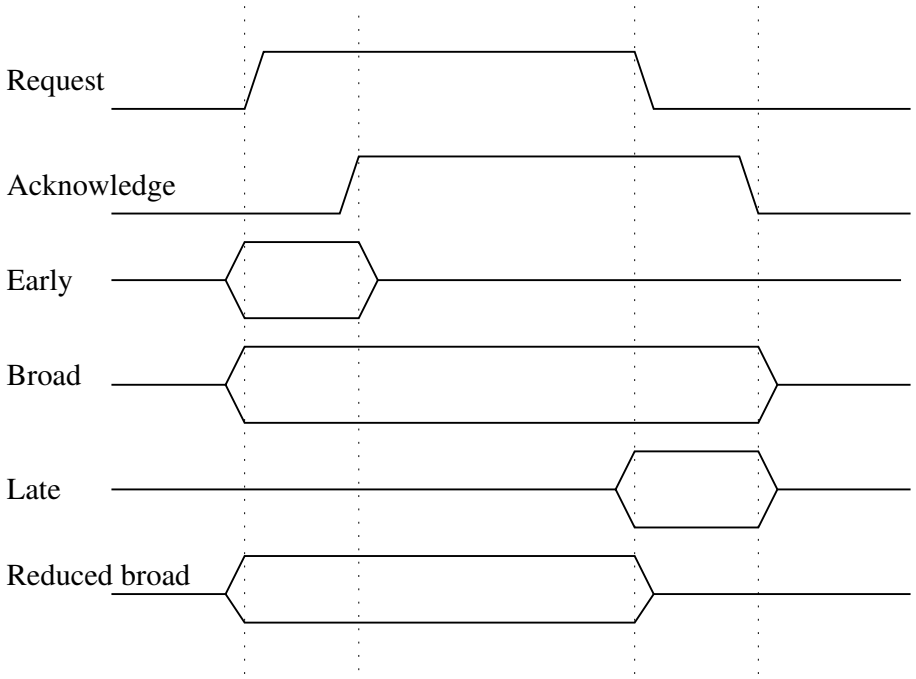


Figure 3.9: The validity schemes of 4-phase protocols

Another interesting characteristic of this latch controller is that it is fully decoupled [35]. This means that two adjacent pipeline stages can be processing in parallel and normal level-sensitive latches can be used to construct an asynchronous pipeline with 100% occupancy. Since no bubbles are in the logic, fully-decoupled latch controllers can be used to construct asynchronous pipelined circuits with high throughput.

The focus of the next two sections is to use asynchronous design to achieve power saving: Section 3.2 compares the power-efficiency of synchronous and asynchronous logic design; Section 3.3 addresses low-power asynchronous design. The remainder of this chapter presents experimental results on the power consumption of some test circuits to demonstrate and compare the power-efficiency of different design approaches.

3.2 Comparison of the power consumption of synchronous and asynchronous designs

The potential advantage of low power consumption is one of the main reasons why asynchronous logic attracts attention from both industrial and academic

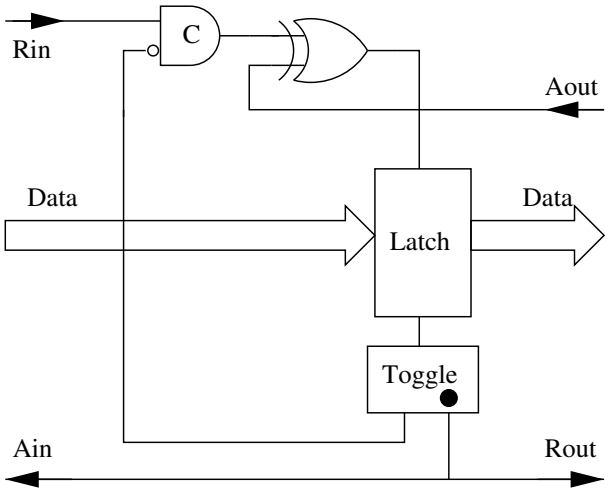


Figure 3.10: The schematic of a 2-phase latch controller

circles. A number of low-power asynchronous designs have been proposed since the 1990s [37], [38], [39], [40]. However, very few publications [41] [42] have evaluated the power-efficiency of asynchronous logic design. In this and the next section, an attempt is made to compare the power-efficiency of asynchronous and synchronous designs, and to evaluate low-power asynchronous techniques.

Asynchronous circuits using a code-data (or delay-insensitive) encoding may give an impression of large size, low speed and high power because of the high logic overheads needed to meet more rigorous timing assumptions. However, not all asynchronous circuits use code-data encoding schemes; an ARM-compatible processor — the Amulet3 [39] — demonstrates that asynchronous microprocessors using a bundled-data encoding can achieve speed and power-efficiencies comparable to their synchronous counterparts. Amulet3 is functionally compatible with the ARM9TDMI microprocessor, which supports ARM architecture version 4T, the 16-bit Thumb instruction set, interrupts and memory faults. The circuit characteristics of the processor are shown in Table 3.1 [1]. Amulet3 achieves roughly the same performance (120 MIPS) as the ARM9TDMI using the same process technology and supply voltage. Its power-efficiency (780 MIPS/watt) is also equal to or marginally better than the ARM9TDMI.

As shown above, an asynchronous design can achieve a similar power-efficiency to its synchronous counterpart. In the remainder of this section, the power-efficiencies of asynchronous and synchronous circuits are compared.

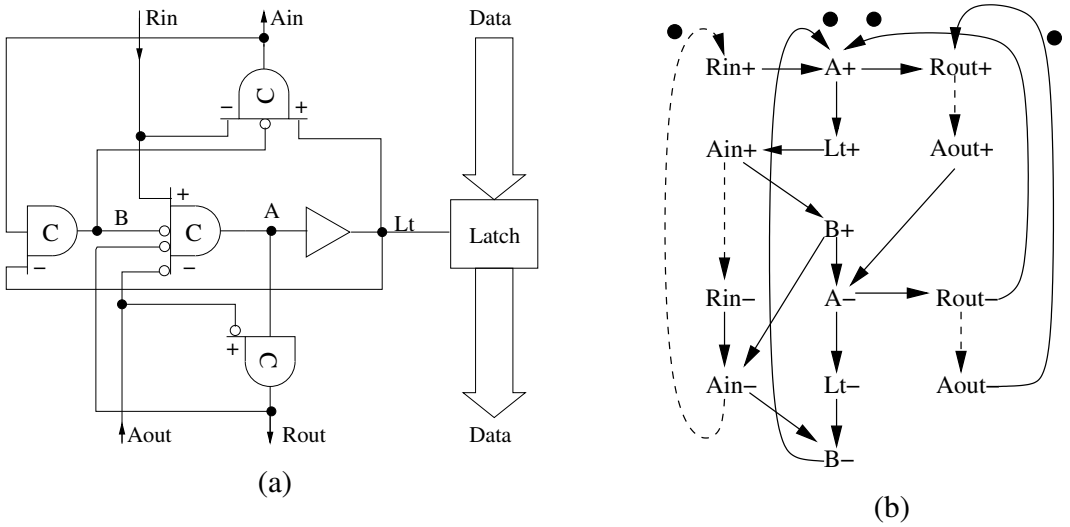


Figure 3.11: A 4-phase latch controller and its STG description

Table 3.1: The characteristics of the Amulet3 processor [1]

Technology	$0.35\mu m$	**Transistors**	113,000	**MIPS**	120
Metal layers	3	**Core area**	3 mm^2	**Power**	154 mW
Voltage	3.3 V	**Clock**	none	**MIPS/W**	780

3.2.1 Global clock and handshakes

The global clock consumes a large proportion of the overall power in synchronous processors as discussed in Section 2.3. The reasons are as follows:

- Much effort must be expended in the design of clock systems, including the drivers and distribution network, to provide a fast and low-skew global timing system distributed to every corner of a silicon die.
- The global clock signal is the most frequently switched signal in a microprocessor; it has two transitions per clock cycle.
- A large proportion of the hardware in a microprocessor is attached to the global clock signal, such as pipeline registers and precharge circuits; these contribute to a large load on the global clock.

In summary the global clock has the largest active switching capacitance and is the most power-hungry part of a synchronous microprocessor.

Using asynchronous logic, the global clock signal is replaced by multiple locally-generated handshake signals. Since each local wire is shorter, and has

smaller capacitance than the global signal, the overall power used by the timing control of an asynchronous circuit may be lower than that of its synchronous counterpart. However, asynchronous timing control also has some unavoidable power overheads.

- The most commonly used 4-phase protocols need four transitions to transfer a data item (two transitions on the *Request* signal and two transitions on the *Acknowledge* signal), while a global clock signal needs only two transitions.
- Handshake circuits, including latch controllers and delay-matching blocks, themselves consume extra power.
- Since no global clock signal exists, there are some unavoidable problems in asynchronous design. For example, it is very difficult for one pipeline stage to know the current timing point of another. Dedicated circuits are needed to exchange information between pipeline stages. The power overhead caused by these circuits can be quite significant.

According to measurements on Amulet3 [17], the overall power contribution of latch controllers, delay-matching blocks and timing drivers is 10.5% of the core power consumption. The proportion consumed by dedicated circuits used to solve the problems of asynchronous design (mainly due to bypassing techniques to prevent pipeline stalls caused by control- and data-dependence) is about 29% (for a synchronous design, the proportion is about 15%). So the overall power consumption due to timing control and asynchronous logic overhead is roughly 25% of the core power consumption of Amulet3. This proportion is not much different from the clock power proportion of a well-designed synchronous microprocessor, such as the StrongARM introduced in Section 2.3.

From the analysis above, asynchronous designs are not necessarily more power-efficient than synchronous designs because they replace the global clock with locally-generated handshakes.

3.2.2 Average latency leading to a simple implementation

Since, in a synchronous pipelined circuit, the clock rate is determined by the slowest stage under worst-case conditions and assuming worst-case inputs, a lot of design effort and hardware additions are needed to speed up the worst-case latency even though such conditions rarely occur.

For an asynchronous pipelined circuit, the run-time latency is data-dependent, and the average throughput depends on the average latency in all conditions. Therefore, it is possible for an asynchronous circuit to allow rare worst-case conditions to have a longer processing delay. If the worst-case conditions do happen, there will be some performance loss, but as long as the worst-case conditions are rare, their impact on overall performance will be small. An example is an asynchronous ripple-carry adder which will be introduced in Section 4.4.2. The hardware additions designed especially for speeding up the worst-case delay are avoided. The power overheads due to these additions can also be saved by replacing a global clock scheme with asynchronous pipeline techniques.

3.2.3 Registers and latches

One data transfer needs 4 transitions in a 4-phase asynchronous design because of the superfluous return-to-zero transitions; the redundant transitions can be used to save power, however.

The fully-decoupled asynchronous latch controllers, introduced in the last section, can use level-sensitive latches to achieve 100% pipeline occupancy. In a synchronous design, 100% pipeline occupation can be achieved only by using edge-triggered registers. Since level-sensitive latches are half the size and have half the active capacitance of edge-triggered registers, asynchronous logic saves significant power in the design of register-based circuits. According to an experiment on improving the power efficiency of a synchronous-style multiplier [43], about 50% of the total power is dissipated in the pipeline registers. Using asynchronous design, 20% of the multiplier power was saved by replacing edge-triggered registers with level-sensitive latches [43]. Section 4.6 will give more detailed information on the multiplier design.

Therefore, asynchronous logic may be more power-efficient in the design of register-dominated circuits.

3.2.4 Zero standby dynamic power consumption

"The Amulet processors are especially good at doing nothing!" — Professor S. B. Furber from the APT group in the University of Manchester uses this sentence to describe the zero standby power consumption in both the Amulet2 [44] and the Amulet3 processors, both designed using an asynchronous bundled-data encoding

scheme.

When it runs out of useful work, the Amulet2 processor is set to an idle state by a 'halt' instruction which stalls a signal in the asynchronous control network. The stall rapidly propagates throughout the control circuit and brings the whole processor to an inactive state. Since there is no global clock, the standby power consumption of the Amulet2 processor is zero except for leakage. When an interrupt occurs, it will activate the blocked control signal and reactivate the whole processor immediately.

A synchronous microprocessor can also enter a low-power idle state, but only with considerable effort. The global clock must be gated off to all parts of the system, except for the interrupt circuit. An interrupt must gate the global clock back on. Since the global clock still ticks in the idle state and the power overheads of clock gating are not avoided, the standby power consumption of a synchronous microprocessor can be quite significant compared to that of an asynchronous microprocessor.

It can be argued that a synchronous microprocessor can also switch off its oscillators and phase-locked loops (PLLs). However, stopped oscillators and PLLs take a considerable time to stabilize when they are turned back on, compromising response time when an interrupt occurs. A poor interrupt response is not acceptable in real-time systems. Some synchronous microprocessors require as much as 10 ms to enter and exit standby state and even fast synchronous microprocessors have wake-up times of circa 6 μs [45]. Asynchronous microprocessors need only a few tens of nanoseconds to wake up — several orders of magnitude faster than synchronous microprocessors.

Asynchronous design can thus achieve near zero standby dynamic power consumption quickly and efficiently with very little overhead.

In a system that spends most of its time in idle state, the standby power consumption is especially important. An example of this is in ultra low-power sensor network applications. A microprocessor with a 200 μA standby current will have a maximum lifetime of 1 year when powered by an AA-size battery even if it never leaves the standby state. In contrast, the lifetime of a microprocessor that burns only few μA of leakage current will be completely dominated by battery self-discharging and the active work to be done. So, 'being good at doing nothing' is a very useful contribution for low-power battery-powered systems.

3.2.5 Fine-grain clock gating

Clock gating is an efficient technique to minimize the power consumption of the global clock. However, in a synchronous microprocessor, clock gating is only 'block-based' or very 'coarse-grain' due to the following reasons:

- The hardware overhead for fine-grain clock gating is high since many small circuit blocks must be gated and the power used by these dedicated clock-gating circuits becomes significant as the granularity of clock gating becomes smaller.

- It is very difficult for a complex clock-gating network to ensure that all disabled blocks power up in time. The modified clocks may generate glitches and this imposes strict timing constraints on the gating signals and requires careful timing verification. Avoiding clock skew and glitches is becoming a greater challenge with the increasing performance of processors, and clock gating makes the problem even more difficult. Thus, the granularity of clock gating is a trade-off between power saving and the complexity of the clock network.

Clock gating is a natural property of asynchronous logic design however. With reference to the asynchronous pipeline using a code-data encoding as shown in Figure 3.4, a pipeline stage sends a *Request* signal to its next stage(s) only when it finishes its evaluation and generates a result. When the next stage of a pipeline detects a *Request* signal, it accepts the inputs and sends an *Acknowledge* back to the stage(s) making the *Request*; it then processes the inputs and relays the *Request* signal to the succeeding pipeline stage. The *Request* signals are sent forward signaling the availability of input data while the *Acknowledge* signals are sent back indicating the availability of 'spaces' to hold the input data. Under the control of *Request* and *Acknowledge* signals, data flows automatically in the asynchronous pipeline; the local handshake scheme ensures that the local clock signals are generated as needed. This is a natural way to implement fine-grain clock gating.

Section 4.6 will present the design of an asynchronous iterative multiplier which achieves significant power saving by using fine-grain clock gating [46]. The multiplier is shown in Figure 3.12 (a) while its synchronous counterpart is shown in Figure 3.12 (b). The 32×32-bit iterative multiplier has two pipeline stages, the first adds 8 partial products together while the second iterates the result, adds

the value from the first stage, then shifts the sum right 8 bits. The multiplier uses an iterative architecture to save hardware and normally needs four cycles to finish a 32×32-bit multiplication. An 'early-termination' scheme is used which increases the speed of the multiplier: if the 8 first most-significant bits are all 0 or all 1, one cycle of the multiplication can be reduced by eliminating the partial products corresponding to these bits; similarly two or three cycles can be reduced if the next one or two groups of 8 bits are all the same as the first group.

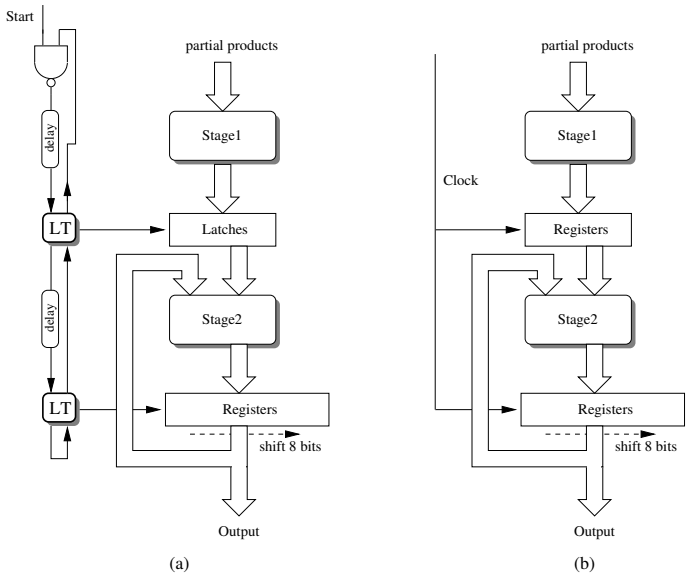


Figure 3.12: An asynchronous multiplier datapath and its synchronous counterpart

Figure 3.13 shows the activities of the asynchronous multiplier and its synchronous counterpart when they execute without early termination cycles and with 3 early termination cycles. Without early termination, as shown in Figure 3.13 (a), the synchronous multiplier has some unnecessary switching activity in the first and last cycles: In the first cycle, only pipeline stage1 needs to do useful work, while stage2 should be idle and wait for the data from stage1. In the last cycle, pipeline stage1 has finished its work, and only stage2 needs to calculate the final result. However, because in the synchronous multiplier both pipeline register and shift register are connected to the global clock signal, both stages operate simultaneously, thus causing unnecessary switching activity. With 3 early termination cycles, the synchronous multiplier wastes even more energy as shown in Figure 3.13 (b).

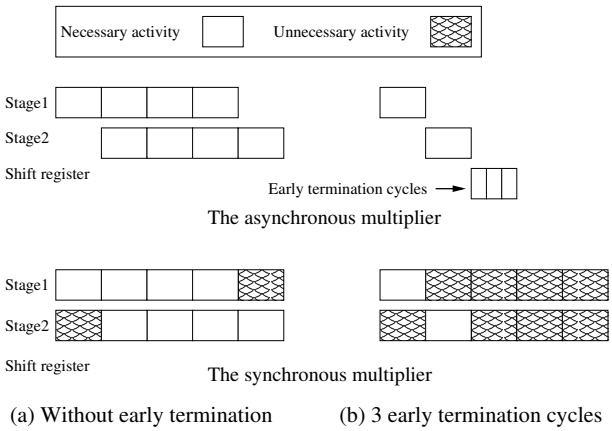


Figure 3.13: The pipeline activities of two multipliers

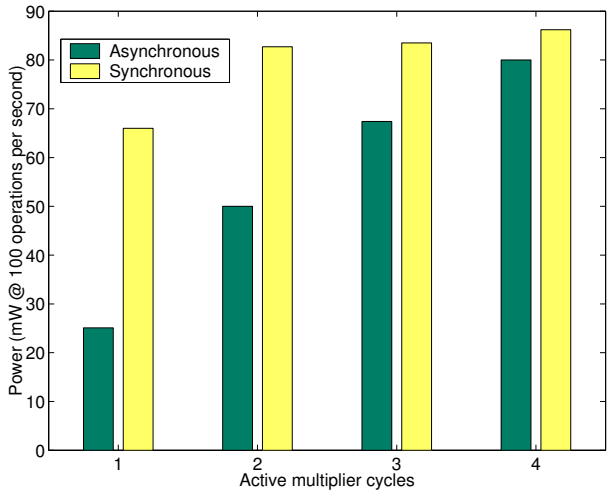


Figure 3.14: The power consumptions of two multipliers

Figure 3.14 shows the power comparison between the two multipliers. As can be seen, the fine-grain clock gating of asynchronous logic greatly reduces the power consumption of the multiplier. More detailed information on the multiplier design is given in Section 4.6.

To sum up, the power consumption of asynchronous and synchronous designs was compared in this section. The comparison shows that asynchronous handshake timing control can be as power-efficient as the global clock timing control of synchronous design. The comparison also shows that asynchronous design may have potential advantages for power saving in several areas including saving hardware for speeding up the rare worst-case delays, using level-sensitive latches, zero standby power consumption and fine-grain clock gating.

3.3 Low-power asynchronous design

In the previous section, the power consumption of asynchronous and synchronous designs was compared. Here, the power consumption of asynchronous designs using different handshake protocols, different data representations and indications are discussed.

3.3.1 Latch controller selection

As discussed earlier, 2-phase logic is rarely used, so only 4-phase latch controllers are discussed in this section. 4-phase latch controllers can be either "normally-closed" or "normally-open" depending on when and for how long the latches they control are transparent (open). A normally-closed (opaque) latch controller opens the latches only for a short period of time (after receiving $Rin+$ events and before sending $Ain+$ events assuming "early" data validity). A normally-open latch controller opens the latches when the output side is free.

The selection of a latch controller is a trade-off between performance and power consumption. If latches are normally open, data items can pass through the pipeline as soon as they become valid. However, any spurious transitions or glitches are also propagated through the whole datapath, causing significant power wastage especially when they occur in the early pipeline stages. If the latches are normally closed, glitches are isolated from the downstream pipeline. However, normally-closed latches suffer from the extra delay of opening the gates once a data item is ready.

To allow a real-time tradeoff between performance and power consumption, Lewis et al. proposed a reconfigurable latch controller which can be switched between 'normally-closed' and 'normally-open' modes at run time [47]. Power savings of 1.8% and 20% are reported when the normally-closed mode is used at maximum speed and at low speed respectively. The speed penalty of the normally-closed mode is from 6.8% to 7.8% depending on the different data validity schemes of the 4-phase protocols.

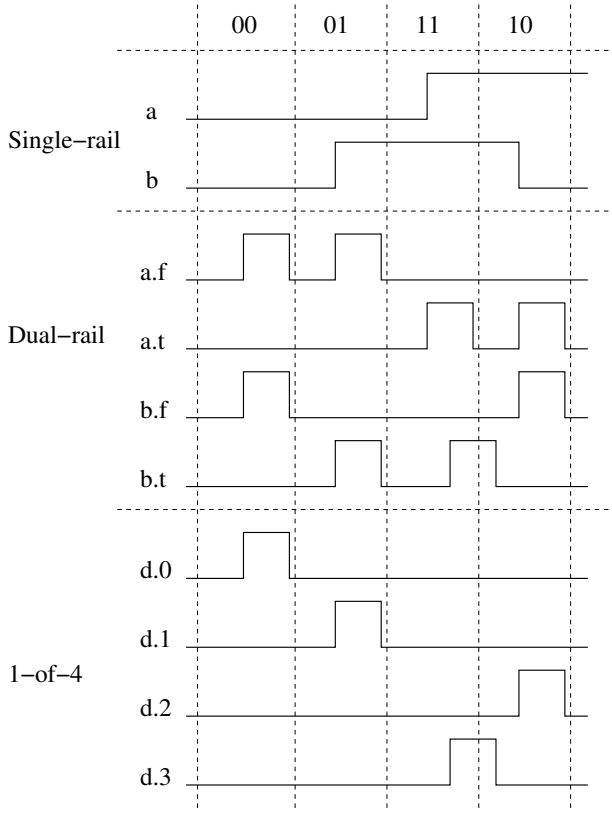


Figure 3.15: Transition numbers in different data representations

3.3.2 Data representation

Data representation is an important asynchronous design issue which has a great impact on power consumption. In this section, three commonly used data representations are discussed — single-rail, dual-rail and 1-of-4 encodings.

Consider the average numbers of transitions of these three encoding schemes. Figure 3.15 shows the transitions of $00 \rightarrow 01 \rightarrow 11 \rightarrow 10$ using the three data representations. The number of transitions in single-rail circuits depends on the historical bit series, but the average number of transitions per bit transfer is 0.5. For dual-rail and 1-of-4 encodings, the number of transitions per bit transfer is fixed — 1 for 1-of-4 encoding and 2 for dual-rail encoding. This analysis gives the impression that a single-rail circuit uses half the power of its 1-of-4 counterpart and a 1-of-4 circuit uses half the power of its dual-rail counterpart. However, this estimated ratio of power consumptions is not accurate in real designs.

To compare the power-efficiencies of the three data representations, three array multipliers were designed and tested. Table 3.2 shows the power consumption

Table 3.2: Power comparison of multipliers using three different data representations

Data representation	single-rail	1-of-4	dual-rail
Transistor number	1456	2593	2216
Power consumption (mW)	0.247	0.696	0.936
Delay — evaluation (ns)	1.21	1.22	1.01
Delay — RTZ (ns)	–	0.85	0.67

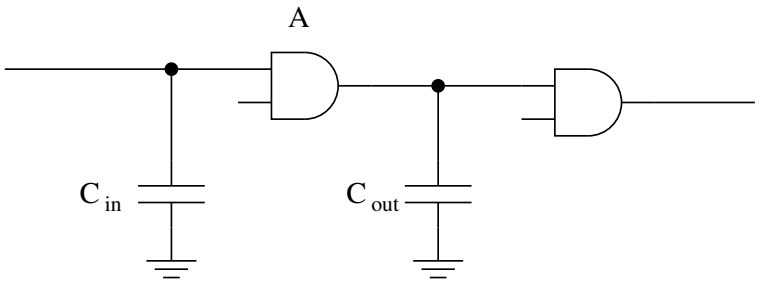


Figure 3.16: The capacitance distribution of a CMOS gate

and the delay of the three multipliers which were tested using the same $0.18\mu m$ CMOS technology and supply voltage of 1.8 volts. The throughput of the multipliers is controlled at 100 million operations per second.

As can be seen from the table, the single-rail multiplier uses only 1/3 to 1/4 of the power of the code-data encoding ones, and the power consumption difference between the 1-of-4 multiplier and the dual-rail multiplier is only 25.6%, which is less than expected. To explain the difference, the analysis of a basic gate, as shown in Figure 3.16, is helpful. If the supply voltage and operating frequency are constant, the dynamic power consumption of a CMOS circuit depends on two factors: N — the transitions per time unit and C_{load} — the overall capacitance of the circuit. The ratio of transitions of single-rail, 1-of-4 code and dual-rail is $1 : 2 : 4$. However, the ratio of overall capacitances is not fixed and depends on the real circuit implementation.

The overall capacitance of a CMOS gate comprises: C_{in} and C_{out}. C_{in} is the capacitance of the input node of the gate and C_{out} is that due to its load and its output node. Based on the differing proportions of C_{in} and C_{out}, these two capacitances have different impacts on the total power consumption. For example, if a gate drives a large load, C_{out} has a significant impact on the total power consumption; otherwise, the influence of C_{in} dominates. Since normally a

single-rail encoding is the simplest of the three encodings, single-rail gates have the smallest gate capacitance (C_{in}). A single-rail circuit also has the smallest number of transitions. These two factors result in a 70% power saving of a single-rail circuit compared to a code-data one.

The power comparison between a dual-rail circuit and a 1-of-4 circuit is more complex. Dual-rail encoding is more straightforward than 1-of-4 encoding, so a dual-rail gate normally contains a smaller number of transistors and uses a smaller silicon area than its 1-of-4 counterpart. Both these factors result in a smaller gate capacitance, which partly compensates for the larger number of transitions. As discussed before, the different sizes of C_{in} and C_{out} cause differing impacts on the total power consumption. When driving large loads, such as powerful inverters and long wires, the power dissipated via C_{out} dominates the total power consumption, so a 1-of-4 gate can save almost half the power when driving large loads. When driving small loads, the power dissipated via C_{in} dominates the total power consumption. Because a dual-rail gate has a smaller gate capacitance and a larger number of transitions than a 1-of-4 gate, the power saved by using a 1-of-4 encoding becomes less significant.

Several basic gates were tested to compare the power consumption of 1-of-4 and dual-rail encoding schemes. These gates were tested under two circumstances: normal load and no load conditions. In the normal load case, each gate drives a load of 4 inverters having the same input capacitance as the gate. These gates are controlled to run at the speed of 500 million operations per second. The experimental power consumptions and delays are shown in Table 3.3.

Inversion is 'transistor-free' and can be implemented by crossing wires in dual-rail and 1-of-4 encodings. AND-gates and OR-gates have the same circuit implementation except for the orders of inputs. XOR-gates and XNOR-gates also have the same implementation. As can be seen from the table, for a 1-of-4 gate, the large gate capacitance counteracts the small number of transitions. Overall, 1-of-4 circuits are more power efficient (0% — 100%) than dual-rail circuits, but how much power saving a 1-of-4 gate can achieve depends on the load capacitance it drives.

The power efficiency advantage of 1-of-4 encoding is especially significant when wire capacitance dominates the overall capacitance, for example, in a long distance interconnection. Figure 3.17 shows two interconnection schemes. One scheme uses dual-rail encoding; the other has dual-rail data inputs and outputs,

Table 3.3: Power and delay comparisons

AND/OR-gates			
	1-of-4	dual-rail	Ratio
Power (normal load) (μW)	126.5	181.6	1/1.44
Power (without load) (μW)	86.0	89.9	1/1.04
Delay (evaluate) (ns)	0.27	0.24	1/0.89
Delay (RTZ) (ns)	0.32	0.26	1/0.81
XOR/XNOR-gates			
	1-of-4	dual-rail	Ratio
Power (normal load) (μW)	132.0	185.8	1/1.41
Power (without load) (μW)	88.4	100.1	1/1.11
Delay (evaluate) (ns)	0.27	0.24	1/0.89
Delay (RTZ) (ns)	0.33	0.26	1/0.79

but uses an internal dual-rail to 1-of-4 code encoder and decoder. Based on experimental data, the relation between the power increase and the wire capacitance increase is shown in Figure 3.18. The power overhead of the encoder and decoder is 99.4 μW at the speed of 500 operations per second. As can be seen, when wire capacitance is greater than 35 fF (about 150 μm of metal in a 0.18 μm CMOS technology — not very long), even with encoding and decoding overheads, 1-of-4 encoding is more power efficient than dual-rail encoding. So, even in dual-rail circuits, it is worth thinking about transferring the dual-rail encoding to 1-of-4 encoding when driving long distance interconnections.

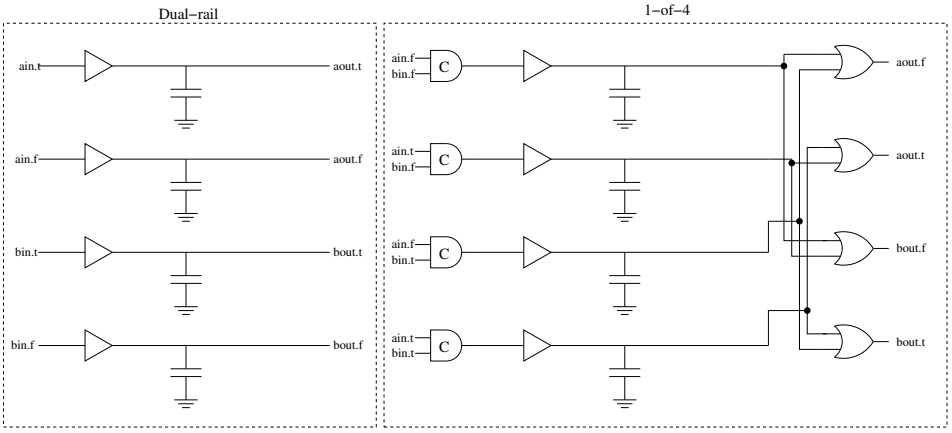


Figure 3.17: Two schemes for long distance interconnection

To conclude, 1-of-4 encoding is more power efficient than dual-rail encoding

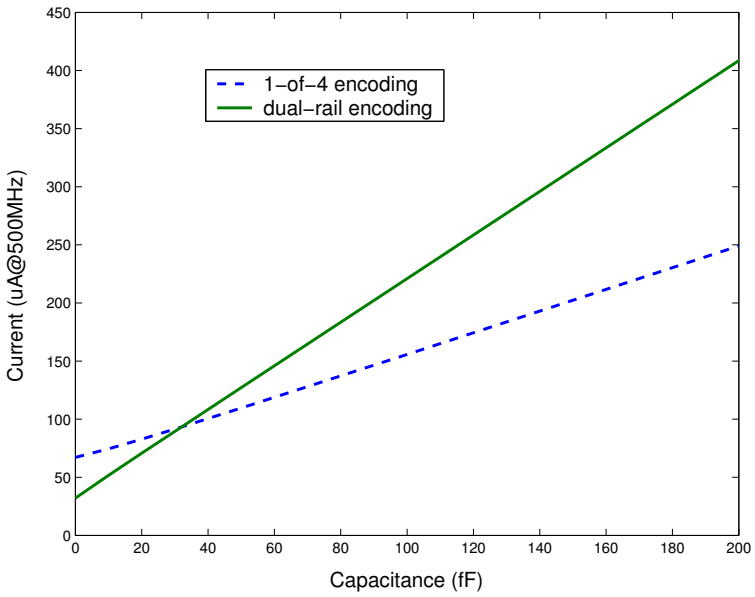


Figure 3.18: Power vs. wire capacitance

but the power saving depends on the ratio of gate capacitance and load capacitance. 1-of-4 encoding is about 100% more power-efficient than dual-rail encoding in long distance interconnections.

3.3.3 Indication selection

Indication is a very important concept in speed-independent and delay-insensitive asynchronous design. An asynchronous design can be either strongly indicating or weakly indicating [28] defined as follows:

- An asynchronous circuit is strongly indicating if (1) it waits for all inputs to become valid before it starts to generate valid outputs, and (2) it waits for all inputs to become empty (normally return-to-zero) before it starts to issue an empty output.

- An asynchronous circuit is weakly indicating if (1) it starts to generate valid output before all of the inputs are valid, although it will not generate a complete valid output until all the inputs are valid, and (2) it starts to generate empty output before all of the inputs are empty, although it will not generate a complete empty output until all the inputs are empty.

Strongly indicating circuits are usually slow and power hungry because their indication constraint is too strict. The rules of weakly indicating circuits are more

flexible as long as their valid/empty outputs indicate their valid/empty inputs, so weakly indicating circuits are normally faster and more power-efficient than strongly indicating ones.

3.4 Summary

In this chapter, the power-efficiencies of asynchronous and synchronous logic designs were compared. Replacing a global clock signal by locally generated handshake signals does not necessarily result in a lower power consumption. However, asynchronous design does have low-power advantages over synchronous design in some areas, especially in low-performance circuits. This power-efficiency comes from fine-grain clock gating, avoidance of hardware complexity because of worst-case delay, level-sensitive latches, zero standby dynamic power consumption and efficient switching between active and idle states.

Different asynchronous design styles were also compared in terms of power consumption. Because normally-closed asynchronous pipeline latch controllers can efficiently block glitches, they are more power-efficient than normally-open ones at the cost of speed. Single-rail circuits are much lower power than code-data encoded circuits. Within code-data representations, a 1-of-4 coding circuit is normally more power-efficient than a dual-rail one, but the power saving depends on the ratio of gate capacitance to load capacitance. Weakly-indicating circuits are normally more power-efficient than strongly-indicating ones because of their small hardware requirements.

The next chapter will present low power techniques for some of the main components of data processing — arithmetic unit designs.

Chapter 4

Low-power arithmetic unit design

Arithmetic and logic units are the main components that perform data processing. Although arithmetic and logic units contribute only a small proportion of the overall power consumption of soft-programmable applications, they can use a significant proportion in stream-based and 'hard'-controlled applications such as multimedia and stream-based encryption/decryption circuits. Arithmetic units also perform other calculations required in control circuits, such as address and index calculations, thus increasing the power-efficiency of arithmetic and logic units is very important. Since logic operations such as AND, OR and XOR are quite simple, the potential for power saving is small and not discussed here. Floating-point computations and integer divisions can be implemented using integer addition and multiplication and many embedded data processing systems do not support floating-point computations or integer division. Hence, this chapter discusses low-power techniques for integer addition and multiplication only.

Since arithmetic units perform data processing and are normally in the critical paths, their speeds are extremely important and low-power techniques must have little or no impact on their performance. The trade-off between speed and power saving should be very carefully studied before adopting a low-power technique. Power saving achieved by a significant performance sacrifice is not acceptable. In this chapter, the *energy delay product* (EDP) is used to evaluate the power-efficiency of arithmetic units together with the *energy per operation* metric.

The remainder of this chapter is organized as follows: Section 4.1 gives a brief introduction to integer adder and multiplier design; Sections 4.2 and 4.3 discuss two implementation issues important to power consumption — logic styles and

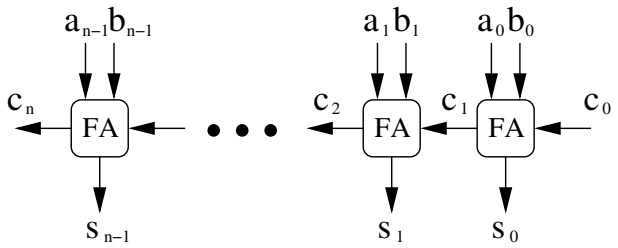


Figure 4.1: A ripple carry adder

data representations; Section 4.4 presents low-power adders and proposes a carry-lookahead adder based on input data characteristics; Section 4.5 describes low-power multiplier techniques; Section 4.6 proposes a low-power integer multiplier; and Section 4.7 sums up the chapter.

4.1 Introduction

A multi-bit adder can be implemented using a 'basic building block' — a single-bit full adder. The logic function of such a full adder is very simple — it adds two bits of the same order (a and b) and a carry bit (c) from a lower-order adder together to generate one sum bit (s) and one carry bit (C_{out}). The carry bit is propagated to a higher-order adder. The logic equations of s and C_{out} are as follows:

$$s = a\overline{b}\overline{c} + \overline{a}b\overline{c} + \overline{a}\overline{b}c + abc$$

$$C_{out} = ab + ac + bc$$

The fundamental problem in constructing a multi-bit adder from a number of single-bit adders is propagating the carry from the low-order adders to the high-order ones. The most straightforward approach to solve carry propagation is with a ripple-carry adder as shown in Figure 4.1. In the figure, $s = a + b$; a_0, b_0 and s_0 are the least significant bits; a_{n-1}, b_{n-1} and s_{n-1} the most significant. The lowest-order adder normally has a carry input of 0; the carry-out bit of m-th adder is fed to the carry-in of $(m + 1)$-th adder.

The critical path of a ripple-carry adder is from a_0, b_0 or c_0 through the whole carry chain to c_n or s_{n-1}, so the worst-case delay of a ripple-carry adder is $n \times carrydelay_{fulladder}$. Although the slowest adder structure, a ripple-carry adder uses the least hardware and has a regular architecture, which make it the cheapest and most power-efficient adder. An n-bit ripple-carry adder has a delay

of O(n). However, since the constant delay factor of a single-bit full adder is small, short ripple-carry adders are often used as building blocks to construct larger adders.

The basic principle of multiplication is also very simple. Given a multiplicand $a = a_{n-1}a_{n-2}...a_0$ and a multiplier $b = b_{n-1}b_{n-2}...b_0$, we have $c = a \times b$ and

$$c = \sum_{i=0}^{n} a \cdot b_i \cdot 2^i$$

where $a \cdot b_i \cdot 2^i$ are called partial products. If the current bit of b (b_i) is 1, the corresponding partial product equals the multiplicand shifted to the left by i bits ($a \cdot 2^i$); otherwise, the partial product is 0. The function of multiplication is to sum the partial products. To improve speed, carry-save adders [48] are used to add partial products to yield a multi-bit partial sum and partial carry. A carry-propagate adder [48] finally adds the partial sum and the partial carry together to produce the final result. A multiplier using carry-save adders is shown in Figure 4.2.

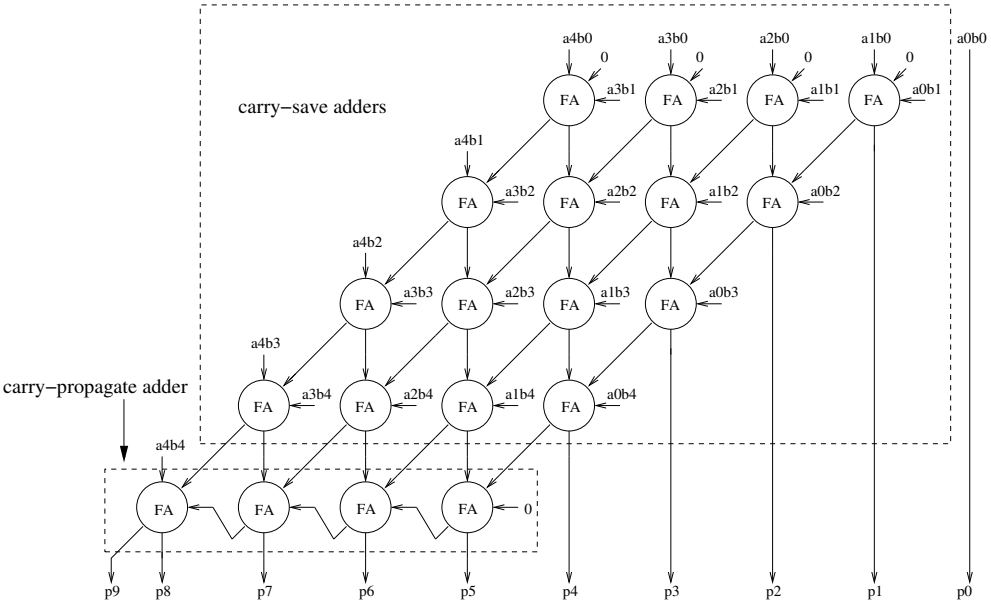


Figure 4.2: A 5×5-bit array multiplier

4.2 Logic style selection

The selection of logic style has a significant impact on the overall speed and power consumption of a circuit. As the most commonly-used technology, conventional static CMOS logic is often used in arithmetic designs. Complementary static logic (CSL) has advantages in both speed and power consumption. It promises that all signals within circuits are fully switched. This characteristic not only minimises static power consumption but also ensures that CSL circuits have 'sharp' transitions, which minimises short-circuit current. Another advantage of CSL is that the layout for a CSL gate is regular and easy to design because of its complementary structures. Moreover, CSL has high immunity to electrical noise and environmental variations. This characteristic is extremely important in submicron VLSI design and is attractive to VLSI designers.

Pass-transistor logic (PTL) is an alternative logic style in arithmetic circuit design which uses a small number of transistors to construct XOR-gates and multiplexers, the basic gates for building arithmetic units.

A pure PTL gate is area-efficient because it uses only n-transistors which are normally half the size of p-transistors. However, n-transistors have a problem in passing logic 1, which results in a low voltage swing [49]. Low signal swing not only affects speed but also results in poor noise immunity which deters designers from using only n-pass transistor logic.

Complementary pass-transistor logic (CPL) uses two n-transistor networks to increase noise immunity which makes full use of the advantages of n-transistors, which are faster and smaller than p-transistors. However, CPL has disadvantages in terms of performance and power-efficiency as follows:

- Since CPL circuits use only n-transistors, the outputs of CPL gates do not exhibit a full voltage swing. The degraded output signals increase static power dissipation.
- The low signal swings decrease circuit speed.
- A swing restoration circuit is needed on every output node, which causes extra delay and power consumption.

The most serious problem of CPL is evident when several gates are cascaded as shown in Figure 4.3. The voltage swing on X is $V_{dd} - V_{tn1}$ and on Y is $V_{dd} - V_{tn1} - V_{tn2}$, where V_{dd} is the supply voltage; V_{tn1} is the threshold voltage

of transistor $T1$ and V_{tn2} is the threshold voltage of transistor $T2$. If the signal swing of cascaded pass transistors is too low, the circuits become very slow, and may even fail to switch the subsequent gate (the inverter in the figure).

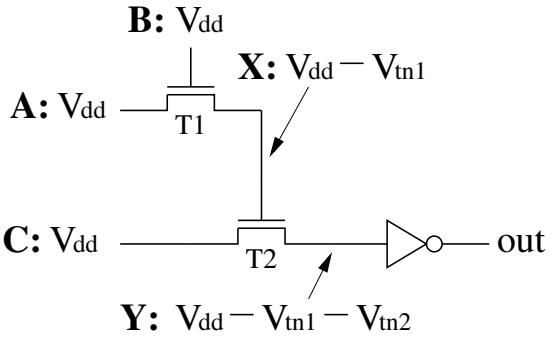


Figure 4.3: The low voltage swing of a cascaded CPL circuit

Double pass-transistor logic (DPL) avoids the low signal voltage swings of CPL by adding p-transistors in parallel with n-transistors. Thus, the problems of low noise immunity and speed degradation caused by reduced voltage swing can be solved. However, the additional p-transistors result in increased gate capacitances.

To compare the speeds and power efficiencies of CSL, CPL and DPL circuits, three 8×8-bit multiplier arrays using CSL, CPL and DPL logic styles were designed and the comparison results are shown in Figure 4.4 (EDP represents *energy delay product*). As can be seen from the figure, the CSL multiplier performs worst in terms of speed and power consumption because it has the biggest active capacitance. The CPL multiplier has the lowest power consumption. The DPL multiplier has the best speed. The *energy delay product* of the CPL multiplier is almost the same as that of DPL.

However, since the experiments do not measure leakage and static power consumption, DPL may be a better choice in low-power and low-performance embedded systems than CPL because DPL normally has lower leakage and static power consumption than CPL. The conclusions drawn from these experiments are compatible with the result of experiments elsewhere [50].

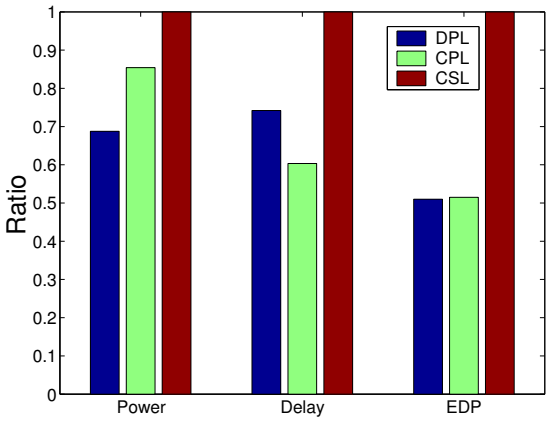


Figure 4.4: Comparison of DPL, CPL and CSL 8×8-bit multiplier

4.3 Data representations

A power consumption comparison of different asynchronous data encoding schemes was made in Section 3.3.3 which showed that data encoding selection has a significant impact on the overall power consumption of circuits. The choice of data representations also has a significant impact on the overall power consumption of an arithmetic unit.

The two's complement data representation scheme is the most commonly-used because it is easy to perform arithmetic operations (such as add and subtract) and has a unique zero. The less significant bits of a two's complement number represent actual data bits; the most significant bits are used to represent the sign. The most significant series of 0 and 1 bits are called the sign-extension bits. The sign-extension scheme is the major factor which causes significant transitions when data switches from positive to negative or vice-versa. For example, changing from -1(11...1) to 0(00...0) toggles all bits through the entire word-width. Therefore, the use of a two's complement representation results in significant switching activity in applications where most data values are small signed numbers and the values frequently switch between positive and negative.

A one's complement data representation has sign-extension problems similar to the two's complement approach. Another representation is "sign and magnitude" where an n-bit sign and magnitude number uses the least significant $n-1$ bits to represent the data, and the most significant bit to represent the sign. Since only one bit indicates the sign, there is no sign-extension problem when using this system.

Chandrakasan and Brodersen analyze and compare the power consumption of a 16-bit bus when it transfers Gaussian data using two's complement and sign and magnitude representations [51]. The results show that sign and magnitude is much more power-efficient than a two's complement representation, and the greatest power saving is when the signal dynamic range is smallest (which means that data are small and sign bits dominate the word-width) and the signal is highly anti-correlated (which implies that the signal changes frequently between positive and negative).

However, arithmetic units to support sign and magnitude representation are difficult to design and adapt to conventional processing systems. The additional hardware and delay overhead may outweigh the power saving of replacing a two's complement representation by a sign and magnitude one. An alternative approach for power saving may be to design a circuit which supports a two's complement interface, while changing the internal data representation to reduce the number of transitions due to the two's complement representation. A sign-exchange algorithm with low hardware and speed overheads is proposed in Section 4.6.

4.4 Adder design

4.4.1 Architecture selection

Selecting an adder architecture is a tradeoff between speed, power consumption and hardware cost. Ripple-carry adders are efficient in terms of hardware and power consumption but are slow; however a ripple-carry scheme is often used to construct small basic blocks for fast and complex adders.

Carry-select adders [52] use hardware duplication to increase speed. A carry-select adder is subdivided into several blocks each containing two small ripple-carry adders. One pre-calculates the result with a carry input of 1 and the other simultaneously pre-calculates the result with a carry input of 0. The carry bit from a lower-order block controls a multi-bit multiplexer to select the correct output. Therefore the worst carry chain delay of an n-bit ripple-carry adder is reduced by parallel calculation. In a carry-select adder, each block pre-calculates two results, but only one is useful and the discarded result causes power wastage.

Carry-lookahead adders [53] also comprise several blocks, each containing one small ripple-carry adder and carry lookahead logic. The small ripple-carry adder

performs normally. The carry lookahead logic generates a carry bit much faster than the carry rippling from the least significant bit of the block to the most significant bit. Thus the worst-case carry chain delay of two blocks is reduced to the overall delay of a small ripple-carry adder and the carry lookahead logic. Since the ripple-carry adder will eventually generate the carry bit, the carry lookahead circuit is redundant and uses extra power.

To further improve adder speed, more complex architectures have been proposed, such as carry-skip adders and carry arbitration adders [54].

A lot of research has been done to compare the power consumption of adders using different architectures [50] [2]. Table 4.1 [2] shows the worst-case delay, area, power consumption and *energy delay product* (EDP) of 6 different adder architectures (using a 2-micron CMOS technology and a 5 volt supply voltage).

Table 4.1: Comparison of different adder architectures [2]

Architectures	Delay (ns)	Power (mW@2MHz)	EDP (ratio)	Area (mm^2)
Ripple carry	51.4	0.43	1	0.26
Constant carry skip	28.6	0.49	0.63	0.33
Variable carry skip	22.8	0.50	0.52	0.49
Carry lookahead	22.5	0.58	0.59	0.53
Carry select	18.6	0.69	0.58	0.88
Conditional sum	21.2	0.84	0.81	1.14

As can be seen from the table, generally, the faster an adder is, the greater the power and area usage. If a given performance constraint can be met, the simplest architecture is the best choice for power saving. Moreover, under specific circumstances, custom-designed architectures are needed to achieve the best power efficiency. For example, if an adder is used to increment a current number by a small offset, half adders can be used to replace full adders in the high-order bits.

4.4.2 The design of an asynchronous carry-lookahead adder based on data characteristics

Asynchronous logic design is claimed to have speed and power advantages over synchronous logic design because an asynchronous circuit can use simple hardware to achieve an 'average latency' which is smaller than the worst-case delay of

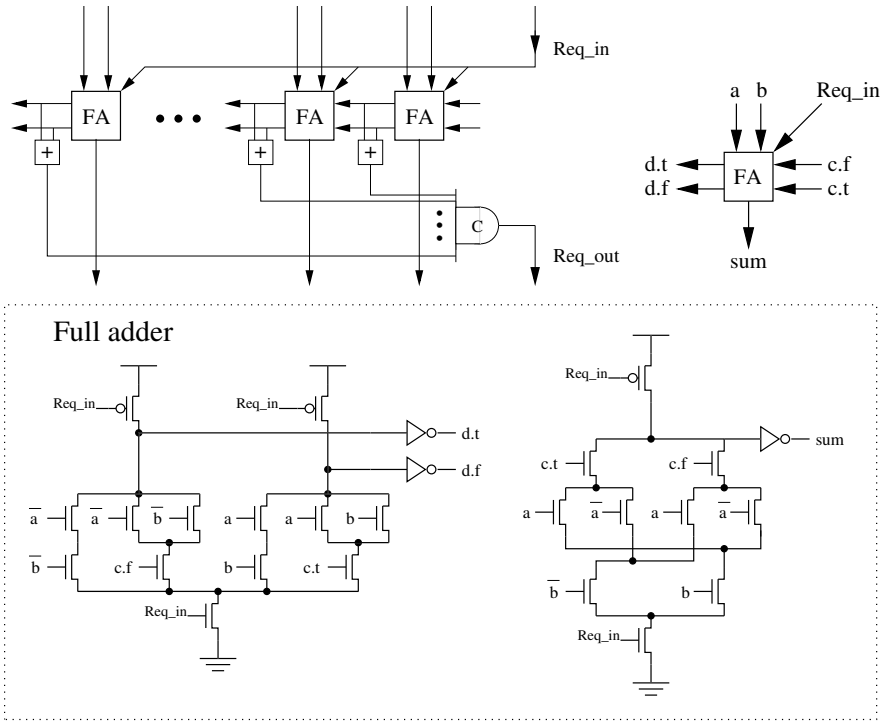


Figure 4.5: A hybrid asynchronous adder which displays average-case latency

its synchronous counterparts. Asynchronous ripple-carry adders are commonly-used examples to demonstrate the average latency of asynchronous design. The schematic of a precharged asynchronous ripple-carry adder is shown in Figure 4.5 [28].

This is a hybrid asynchronous circuit, which has single-rail inputs and outputs, but whose carry chain uses dual-rail 4-phase signaling. The carry bits of this adder are weakly indicating. Therefore, if the two inputs (a and b) of the adder are equal, the carry-generating circuit can generate a carry output and propagate it to the high-order adder without waiting for a valid carry input from the low-order adder. Only when $a \neq b$ does the adder need to wait for a low-order carry input. So there is only a 50% probability that the adder needs to propagate carries. The carry chain of the asynchronous ripple-carry adder is shown in Figure 4.6. The weak indication of the asynchronous full adders makes the carry chain discontinuous — every node has a 50% chance to break the chain. Thus the worst-case latency of an addition depends on the longest carry chain segment in the addition. Figure 4.7 plots the average latency of a weakly-indicating asynchronous ripple-carry adder as a function of its word-length when it is fed with randomly generated numbers [6]. For a weakly-indicating 32-bit ripple-carry asynchronous adder, the

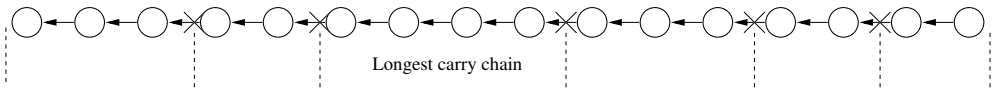


Figure 4.6: A weakly indicating asynchronous carry chain

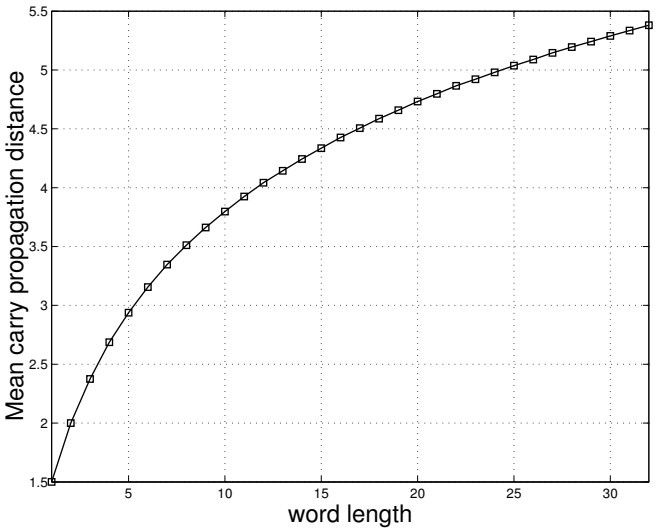


Figure 4.7: Average size of the longest carry chain for different word lengths assuming random data distribution [6]

average latency is only 5.34 full adder delays, which is smaller than the worst-case delay of a much more complex synchronous adder.

Unfortunately, this simulation does not reflect what is encountered in practice; here the low average latency of asynchronous adders is based on the assumption that all input vectors are random numbers. However, in real applications, the input operands fed to an asynchronous adder are not random and this 'average latency' of asynchronous adders is not generally achieved. Garside demonstrated the unbalanced distribution of both data processing and address calculations in the Dhrystone benchmark [6]. In this work, ten sets of input vectors were used to test the practical average latency of asynchronous adders. The characteristics of the input vectors and the average latencies of a weakly-indicating asynchronous adder when using these vectors are shown in Table 4.2.

The vectors are taken from the ALU and address incrementer (discarding sequential calculations) of an ARM microprocessor running several benchmarks. Figure 4.8 and Figure 4.9 show the different distributions of the longest carry propagate distance. For *rand*, the carry chain lengths congregate in the range 2 to 7, so the average carry propagate distance is about 5.4. For $jpeg(d)$, the carry

Table 4.2: Input vectors

Vectors	Description	Average carry propagate distance (full adders)
Rand	Randomly generated inputs	5.34
Gauss	Gaussian samples	14.78
adp(a)	Branch calculations of a media program	12.25
adp(d)	Data processing of a media program	12.20
espr(a)	Branch calculations of an Espresso algorithm	14.90
espr(d)	Data processing of an Espresso algorithm	16.96
jpeg(a)	Branch calculations of a JPEG program	10.40
jpeg(d)	Data processing of a JPEG program	11.83
qsort(a)	Branch calculations of a quick sort program	10.44
qsort(d)	Data processing of a quick sort program	16.04

chain lengths separate onto the two ends of the X-axis. Many carry chains are smaller than 8 full adder delays, while many of them exceed 24. This yields a mean carry propagate distance of 11.8. For $jpeg(a)$, a significant extra peak exists at the 16 point and a mean carry propagate distance of 10.4 full adder delays. As can be seen from the comparison, the actual average carry propagate distance of an asynchronous adder is much bigger than the average carry propagate distance of 5.3 based on random numbers.

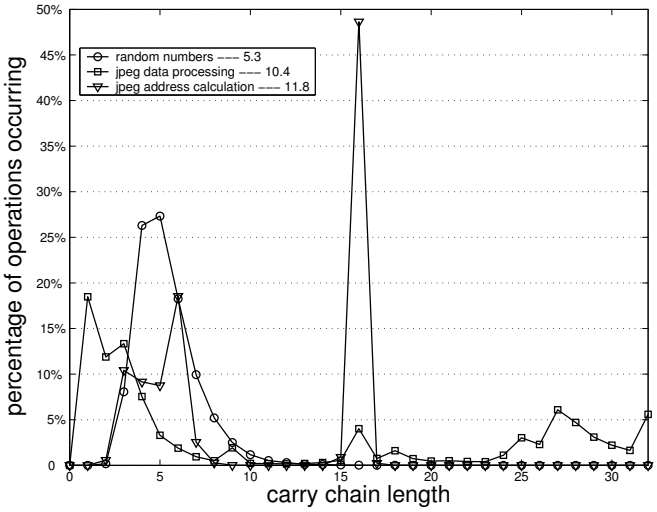


Figure 4.8: Longest carry propagate distance distribution

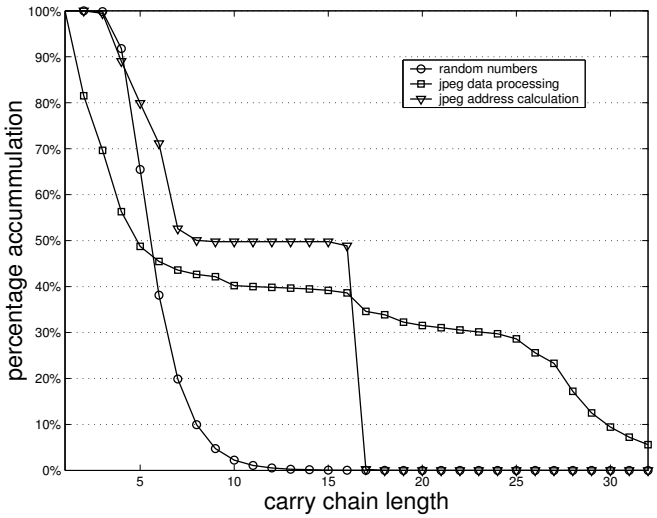


Figure 4.9: Proportion of longest carry chains exceeding given length

An interesting characteristic of Figure 4.8 is that the high percentage distributions gather at the two ends — the longest carry propagate distances are either very small or very big. Through further analysis of data processing operations it was found that the short carry chains represent the additions of two inputs having the same signs (they are both negative or positive). The long carry chains represent the additions of a small negative number and a positive number. For example, $0-1$ (0xFFFFFFFF+0x00000000) has the longest carry chain containing 32 full adder delays. If it is assumed that there is a 50% chance of adding positive numbers and negative numbers, the average latency of an n-bit asynchronous ripple-carry adder is about $n/2$ full adder delays.

The reason why a significant proportion of address calculations have a longest carry propagate distance equal to half the word length is due to compilation characteristics and the specific CPU architectures. In the tests, the ARM programs are loaded at the address 0x8000. As is well known, branches dominate the trace of a program and among branches, jump back instructions having a small jump distance are the most common. The specific loaded addresses and small negative offset result in a longest carry chain segment. For example, 0x8010 - 9 (0x00008010+0xFFFFFFF7) has a carry chain that contains 16 full adder delays. Thus more than 50% of address calculations gather in the area of half of the word length.

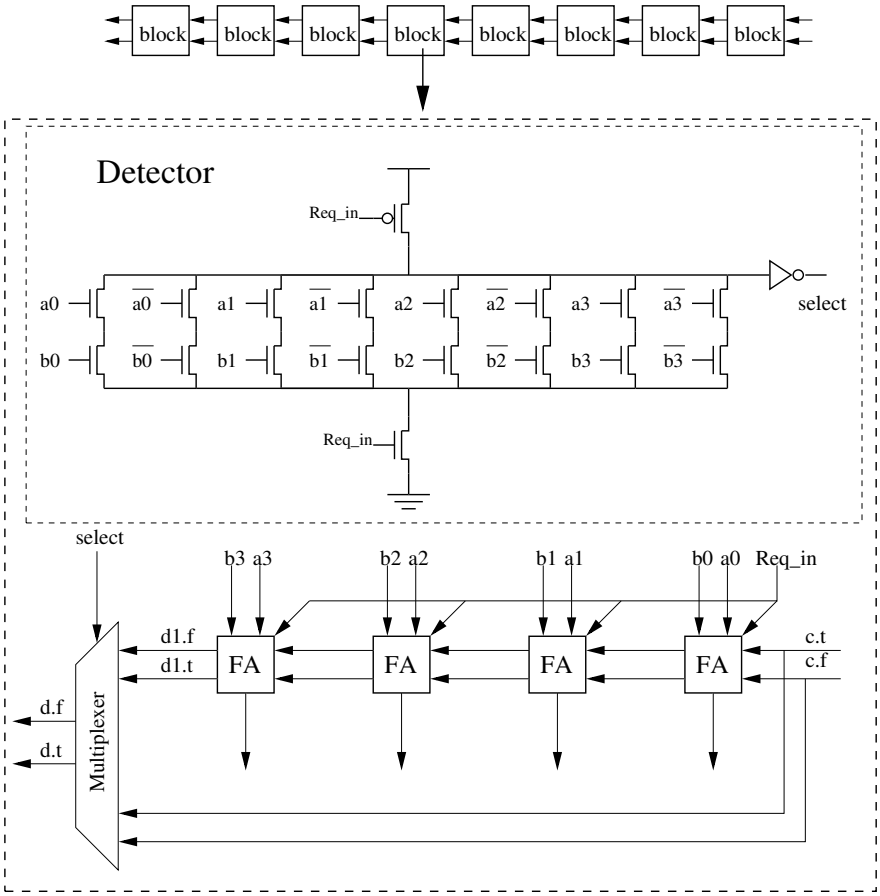


Figure 4.10: The proposed adder

4.4.3 An asynchronous carry-lookahead adder

Because long carry delays usually occur when adding a negative and a positive number, for applications where most operands are positive numbers, asynchronous adders still have a low average latency advantage. However, if an application contains many additions of small positive and negative numbers and the average latency of an asynchronous adder cannot meet a given performance requirement, hardware additions are needed to speed up the asynchronous adder. High performance techniques used in synchronous adders can also be used for asynchronous adders. However, another method can be deduced from the observation mentioned above. If there is a pair of same-order input bits equal to each other, a carry output can be generated before the arrival of a carry input and the carry chain is broken at this node.

A fast asynchronous adder is shown in Figure 4.10. A 32-bit adder is subdivided into 8 blocks, each containing a 4-bit asynchronous ripple-carry adder

and a detector. The detector checks if there is a bit pair that are equivalent. If not, the carry from the lower-order block is propagated over 4 bits directly to the high-order block; otherwise, the carry bit is propagated in a conventional way. This adder is similar to a synchronous carry-lookahead adder; it is efficient and has low hardware-overhead. It reduces the worst-case delay by a factor of 8 (ignoring the multiplexer delay) and the hardware overhead includes only 7 detectors and multiplexers. Using a precharge logic style, the detector is small and fast. The hardware overhead is about 25% of the overall transistors in the adder. The delay of the detector is not on the critical path of the adder since all the detectors execute in parallel at the beginning of an addition. The delay of a multiplexer is 0.8 of a full adder delay. The worst-case delay of the 4-4-4-4-4-4-4-4 (8,4) scheme (8 blocks and each block has 4 full adders) is $7 \times 1 + 7 \times 0.8 = 12.6$ full adder delays. The worst-case delay can be minimized by reorganizing the full adders and a 5-5-5-5-5-7 scheme is tested for comparison. Figure 4.11 shows the evaluation speeds (without including the delay of the completion detector) of the different schemes. As can be seen, the (8,4) scheme is good in terms of speed and it keeps the detectors at a reasonable size. Using the proposed technique, the average latency of address calculation is reduced by 44% and the average latency of data processing is reduced by 53% with a 25% hardware overhead compared to the ripple-carry adder shown in Figure 4.5.

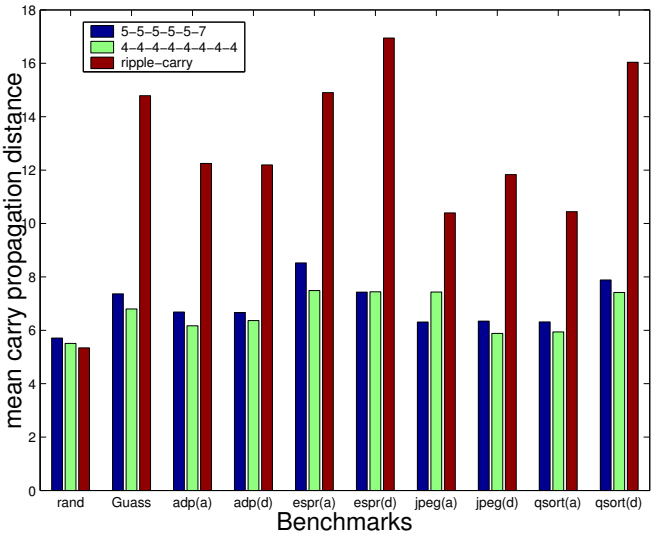


Figure 4.11: Delay comparison of three asynchronous adders

4.4.4 Completion detector design

The delays in Figure 4.11 do not include the delays of an asynchronous detector which indicates the completion of additions. A completion detector can greatly affect the overall speed because it has a C-gate with a fan-in of n (see Figure 4.5). Nielsen and Sparsø [55] combined strongly indicating and weakly indicating full adders to minimize the number of nodes which need to be checked. However, this scheme also increases the possibility of carry dependence, which affects the average latency of asynchronous adders (a 24.6% performance loss based on the benchmarks).

A tree-style detector is used in the proposed adder as shown in Figure 4.12 (a); its delay can be reduced by several inverter delays by interlacing n-pass-transistor gates and p-pass-transistor gates as shown in Figure 4.12 (b). This tree detector is fast because it detects the completion of carries in parallel, so for n-bit adders, it has a $log_2 n$-level logic delay. Moreover, the completion signal is propagated concurrently with the carries, so the delays mostly overlap and the detector's contribution to the overall delay is small.

To improve the speed of the completion detector, drivers are added for long wires. Because the pass transistors in the completion detectors are closed after return-to-zero phases, the internal nodes after the pass transistors need to be precharged before the next addition. When using n-pass-transistor gates and p-pass-transistor gates together, the nodes after the n-transistors are precharged to 1 and those after the p-transistors to 0.

4.4.5 Experimental results

The proposed adder was designed using a $0.18\mu m$ SGS-Thomson CMOS technology. An asynchronous ripple-carry adder and a synchronous carry-select adder were also designed for comparison as shown in Table 4.3. The simulations are schematic-based and wire capacitances are ignored; the supply voltage is 1.8 volts.

The use of a dual-rail internal carry chain not only results in a low average latency but also reduces unnecessary switching activity caused by glitches from the carry chain, because carries are not propagated until they are valid. However, the dual-rail carry chain does not yield low power consumption although it minimizes glitches, because after each operation, the carry chain needs to return to zero, introducing extra transitions. A similar problem occurs with the completion

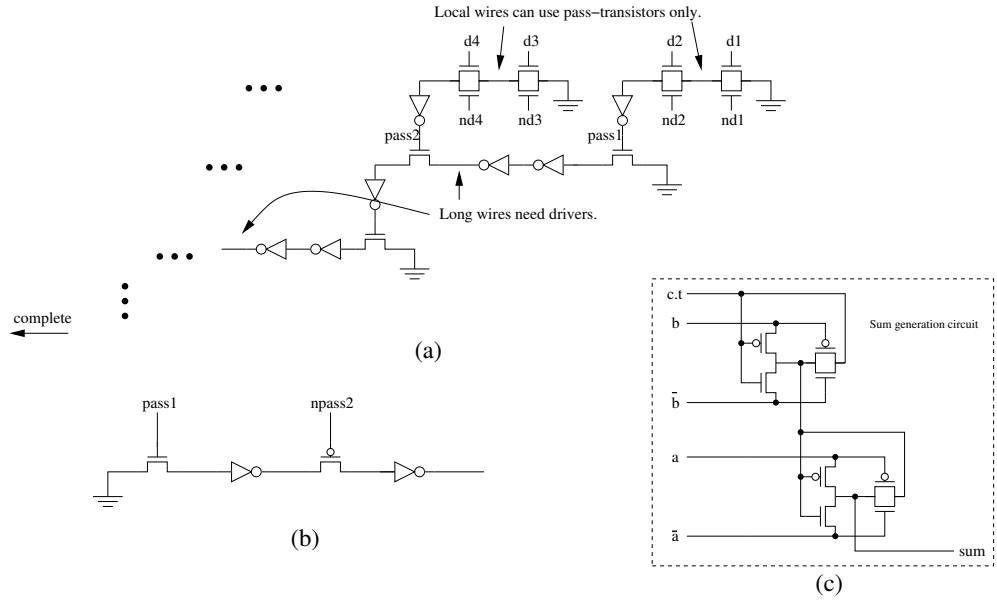


Figure 4.12: The pass-transistor tree completion detector and sum generation circuit

detector. This is the reason why the asynchronous ripple-carry adder does not halve the power consumption of the synchronous carry-select adder as expected, although it uses less than half of the hardware. For this reason, the sum generation circuit does not use a precharge logic style but instead uses pass-transistor logic as shown in Figure 4.12 (c).

As can be seen from Table 4.3, the proposed adder is 27% faster than the asynchronous ripple-carry adder in data processing and 19% faster than the ripple-carry adder in address calculation at the cost of 25% hardware overhead and 15% power overhead when running these benchmarks. For the worst-case latency, the proposed adder is more than twice as fast as the ripple carry adder. The proposed adder is also faster than a synchronous carry-select adder in both data processing and address calculation. For power-efficiency, the proposed adder using pass-transistor sum circuits is only slightly better than the carry-select adder. This is due to the return-to-zero operations of the dual-rail carry chain and the completion detector. Without return-to-zero, the synchronous adder also saves transitions when the current operand pair is similar to the one it has just finished.

Table 4.3: The comparison of different adders

Adders		proposed precharged	proposed pass-transistor	asynchronous ripple-carry	synchronous carry-select
worst-case delay		1.68	1.68	3.48	1.38
transistor number		1,730	1,690	1,332	3,144
area (ratio)		0.5	0.5	0.4	1
rand	delay	1.18	1.18	1.11	1.38
	power	0.51	0.45	0.40	0.64
Gauss	delay	1.23	1.23	1.69	1.38
	power	0.46	0.43	0.37	0.56
data processing	delay	1.28	1.28	1.75	1.38
	power	0.48	0.42	0.37	0.45
address calculation	delay	1.18	1.18	1.45	1.38
	power	0.45	0.42	0.35	0.45

4.5 Multiplier design

As described in the first section, a multiplier often has a tree architecture which contains an array of one-bit full adders. Low-power multiplier techniques usually focus on two aspects: reducing the overall capacitance and reducing the number of transitions within the full-adder array. Reducing capacitance is normally achieved by low-level circuit optimizations and reducing transition numbers usually employs high-level optimizations, for example, using a more efficient data representation and multiplication algorithm. The remainder of this chapter will address these two aspects of low-power design.

4.5.1 Basic building blocks and architectures

Multipliers can be constructed using a number of of basic building blocks such as 1-bit full adders. Such a full adder has three inputs and two outputs and hence it is also called a 3-2 adder. Figure 4.2 shows the architecture of an array multiplier. It is simple but slow because it adds partial products serially and has a processing delay of O(n).

The long latency of an array multiplier can be reduced to O($logN$) using a tree architecture which adds n partial products in parallel and generates $n/2$ 'middle variables' (partial sums). These $n/2$ partial sums would then be added to generate $n/4$ partial sums, and so on; a tree multiplier therefore has a processing

delay of O($log_2 N$). A Wallace tree architecture [56] uses 3-2 adders to achieve high performance as shown in Figure 4.13 (a). However, because a Wallace tree uses 3-2 adders as basic building blocks and a 3-2 adder consumes 3 inputs and generates 2 outputs, it reduces partial products only at the rate of $log_{1.5} N$. Moreover, a Wallace tree is difficult to implement because, as can be seen from the figure, the full adders and the interconnections are arranged irregularly.

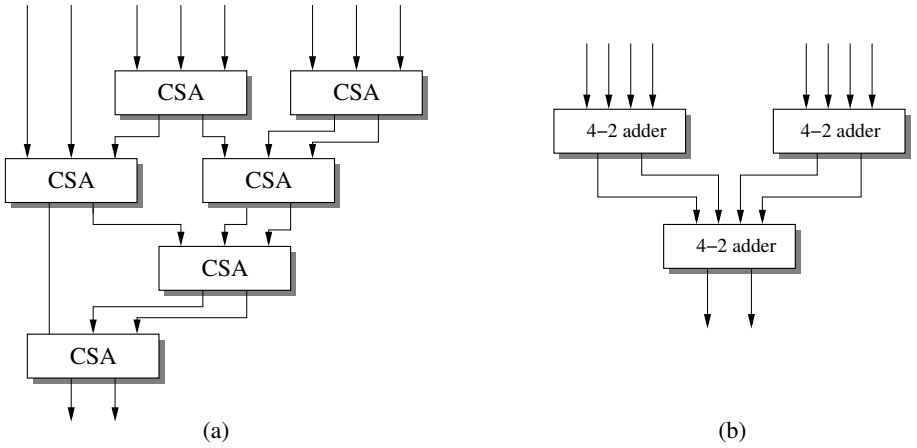


Figure 4.13: Two 8-2 tree adders using 3-2 adders and 4-2 adders

With the intention of reducing architectural complexity, more efficient adder elements should be created to support more regular tree structures. A 4-2 adder, having 4 inputs and 2 outputs, can be used to achieve this goal. A binary tree architecture using 4-2 adders is shown in Figure 4.13 (b). The binary tree architecture reduces partial products at the rate of $log_2 N$, so it is superior to a Wallace tree in terms of speed. The binary tree architecture also has the advantage of regularity, which contributes to reductions in area, delay and switching capacitance of internal wires.

A 4-2 adder can be viewed as a logic component constructed from two 3-2 adders as shown in Figure 4.14 (a). As can be seen from the figure, C_{out} is not a function of C_{in}, so there is no carry propagation along adder rows with the same order. This is the key idea behind a 4-2 adder. By considering C_{in} and C_{out} as two intermediate variables, the adder consumes 4 inputs and produces 2 outputs.

Although a 4-2 adder can be synthesized by using two 3-2 adders, a faster and smaller implementation is possible. A DPL (double pass-transistor logic) 4-2 adder as used in the Amulet3 multiplier [57] is shown in Figure 4.14 (b); this is faster than the one shown in Figure 4.14 (a) and its delay is only 1.5 (instead of

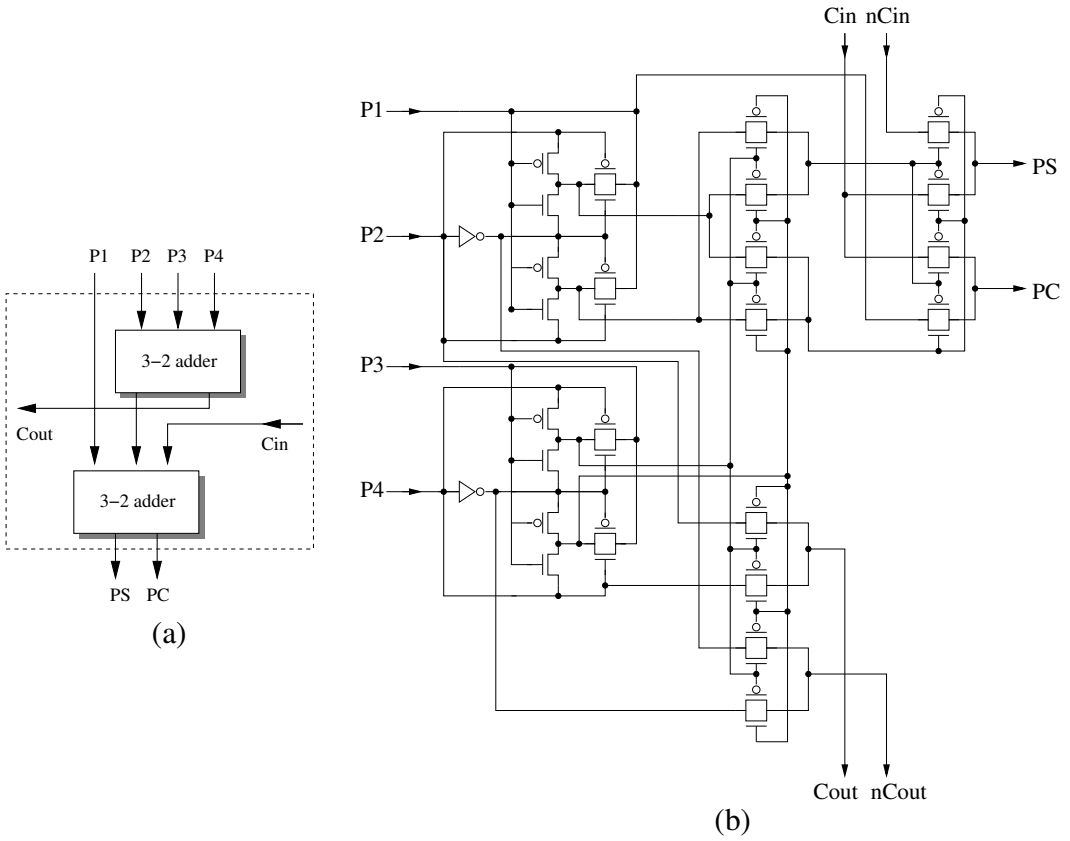


Figure 4.14: Two ways of implementing a 4-2 adder

2) full adder delays.

More complex basic adders can be used, for example a 5-3 adder [58], but because of their complexity and the large circuit size, these adders are not discussed here.

4.5.2 Commonly-used algorithms

Modified Booth's algorithm

Since adder arrays only combine partial products, specific algorithms are needed to generate these partial products. The simplest algorithm, already mentioned, scans a multiplier bit by bit and if the current bit is 0, the corresponding partial product is 0, otherwise it equals the multiplicand. This algorithm generates n partial products where n is the word length of the multiplier. A more efficient algorithm called the "modified Booth's algorithm" [59] generates only $n/2$ partial products thus reducing partial product additions by a factor of two.

The mathematical principle behind the modified Booth's algorithm using a two's complement representation is illustrated in equations as follows:

$$A = B \times C$$

$$C = -c_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} c_i \cdot 2^i \tag{4.1}$$

$$C = \sum_{i=0}^{n/2-1} (s_{2i} + s_{2i+1} - 2 \cdot s_{2i+2}) \cdot 2^{2i+1} + c_0 \tag{4.2}$$

where B is the multiplicand, C is the multiplier, $s_0 = 0$; $s_n = c_{n-1}$ and $s_1 ... s_{n-1} = c_1 ... c_{n-1}$.

Let $k_i = s_{2i} + s_{2i+1} - 2 \cdot s_{2i+2}$, so k_i is in the set of $\{-2, -1, 0, 1, 2\}$. Different operations on the multiplicand B are based on the different values of k_i as shown in Table 4.4.

Table 4.4: The modified Booth's algorithm scheme

Bits group (k_i)	Operation	V
000	+0	0
001	+B	0
010	+B	0
011	+2B	0
100	-2B	1
101	-B	1
110	-B	1
111	+0	0

In the table, '+0' indicates the corresponding partial product is 0; '+B' indicates the partial product is B or the multiplicand; '+2B', '-B' and '-2B' indicate multiplicand weightings. V is a 'forced carry-in' to allow the negative ($-B$, $-2B$x) partial products to be implemented with bitwise inversion ($-B = \overline{B} + 1$). Hence, V is added to the least significant bit of the partial product.

Improved sign extension algorithm

When a partial product is negative, it is necessary to extend its sign bits to the most significant bit (MSB). This requires more adders in a multiplier array and consumes extra power. An improved sign extension algorithm [60] was proposed

to reduce the number of the sign bits needing to be extended. This improved sign extension algorithm is based on the observation of the following equations:

Positive numbers: 00...0 0xxxxx = 11...11 100000 + 1xxxxx

Negative numbers: 11...1 1xxxxx = 11...11 100000 + 0xxxxx

If a 4-2 architecture is used, the 'triangle' of 1s (most significant bits) of the four partial products can be pre-added to a string of '10101011' as shown in Figure 4.15. The sum of the rest of the 31 least significant bits of the four partial products is processed as normal. The string of '10101011' can be distributed into the four partial products. Since the third, the fifth and the seventh bits of the string are zero, the inverted most significant bits of the second, the third and the fourth partial products (pp2, pp3 and pp4) are just inserted into the string. For the first partial product (pp1), if the inversion of its most significant bit is 0, its sign extension bits are set to '11'; otherwise, they are set to '100'. Therefore, the four partial products can be replaced using the following steps:

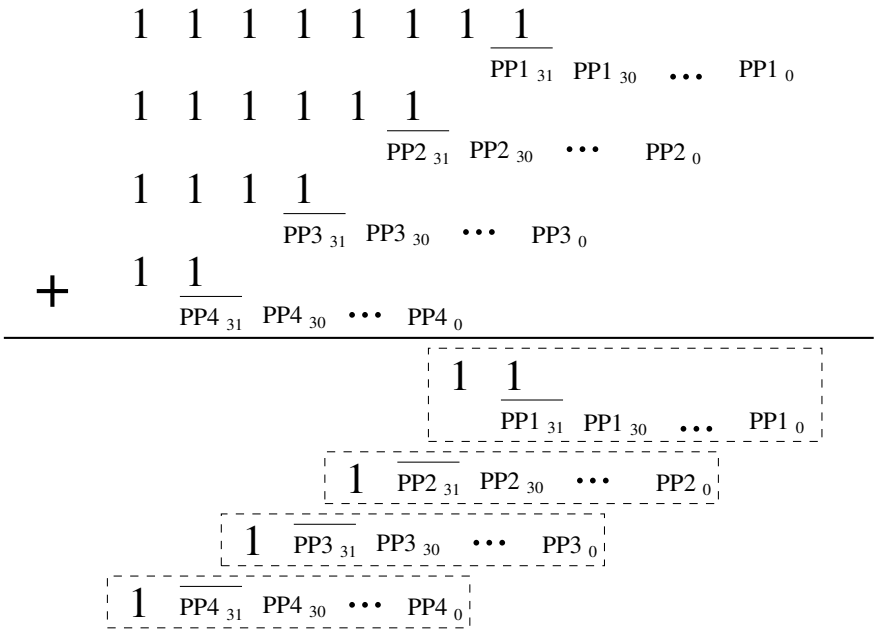


Figure 4.15: The principle of an improved sign extension algorithm

- If the MSB of the first partial products is 1, put '01' to the left of the first partial product as extended sign bits; otherwise, put '10' to the left of the first product as extended sign bits.
- Invert the MSB of the second, the third and the fourth partial products; put a 1 to the left of the three partial products as their extended sign bits.

With the improved sign extension algorithm, only one sign extension bit (for pp1, 2 bits) is needed rather than a long sign extension all the way up to the most significant bit. This scheme saves both hardware and power.

Signed and unsigned algorithm

An unsigned number representation is common in digital signal processing applications. It is often necessary for a multiplier to support both a two's complement representation and an unsigned representation. If a multiplier C is an unsigned number, its value is defined by the following equation:

$$C = c_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} c_i \cdot 2^i \tag{4.3}$$

This is different from the definition used by a two's complement representation as in Equation 4.1. Fortunately, Equation 4.2 can be modified by only one bit to support both signed and unsigned numbers. Equation 4.4 shows a modification of Equation 4.2 by defining s_n differently:

$$C = \sum_{i=0}^{n/2-1} (s_{2i} + s_{2i+1} - 2 \cdot s_{2i+2}) \cdot 2^{2i+1} + c_0 \tag{4.4}$$

where $s_n = c_{n-1}$ when C is a signed number and $s_n = 0$ when C is an unsigned number. $s_0, s_1 ... s_{n-1}$ are left unchanged. In a practical multiplier, a signal $Sign$ indicates the number format of the operands (1 for a two's complement format and 0 for an unsigned format), so $s_n = c_{n-1} \cdot Sign$. By doing this, the multiplier supports two data representations by simply controlling a sign bit.

4.5.3 Architecture selection

Callaway et al.[2] compared 4 different multiplier architectures with the results shown in Table 4.5. As can be seen, an array multiplier is the worst in terms of both speed and power consumption. A modified Booth's multiplier is good for speed but not for power-efficiency. A radix-2 Wallace multiplier is the best among them in energy delay product (EDP).

Table 4.5: Characteristics of multiplier architectures

Multiplier	Delay (ns)	Power (mW@10MHz)	EDP (ratio)	Area (mm^2)
Array	92.6	43.5	1	4.2
Split array	62.9	38.0	0.59	6.0
Wallace	54.1	32.0	0.43	8.1
Modified Booth's	45.4	41.3	0.47	8.5

4.5.4 Input vector characteristics

Most published work on low-power multipliers uses randomly-generated vectors to test the power consumption of the designs (for example [50] and [2]). However, in a practical multiplier, operands are far from random, so these results are not representative of the performance with typical data. The best way to test the power consumption of a multiplier is to base the test operands on real applications. As this is difficult to do during the design period, an alternative is to use benchmarks.

To investigate the characteristics of typical multiplier operands, four benchmark programs are employed: "go" (an internationally-ranked go-playing program), "ijpeg" (a standard JPEG image compression and decompression program), "compress" (a file compression program) and "vortex" (an object-oriented database). A total of 33 million multiplication pairs were taken from these four benchmarks and it was found that the distribution of inputs was unbalanced in two respects:

- The distribution of positive and negative operands is highly imbalanced; table 4.6 shows the statistical results. As can be seen from the table, the great majority of inputs are positive.

- The 'Significant Bit Count' (SBC) is the number of bits at the bottom of a binary number ignoring all of the high order bits which comprise a

Table 4.6: Operand distribution between positive and negative

Benchmark	**go**	**ijpeg**	**compress**	**vortex**
% positive	87.6%	97.8%	100%	100%
% negative	12.4%	2.2%	0%	0%

series of 1s or 0s. These high order bits represent the sign of the number in 2's complement format. For example: the SBC of 0x00006380 is 15 and the SBC of -5 (0xFFFFFFFB) is 3. Figure 4.16 and Figure 4.17 show the SBC distributions for the benchmark programs. As can be seen, the SBC distribution is not balanced. Most inputs have an SBC between 0 and 16 bits.

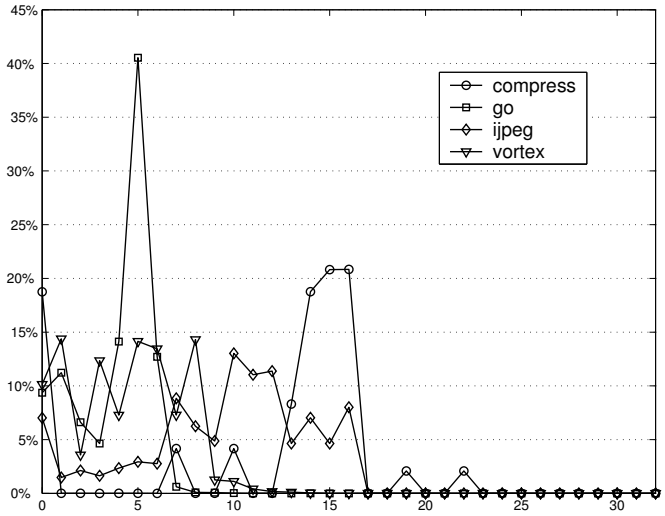


Figure 4.16: SBC distributions for the benchmark programs

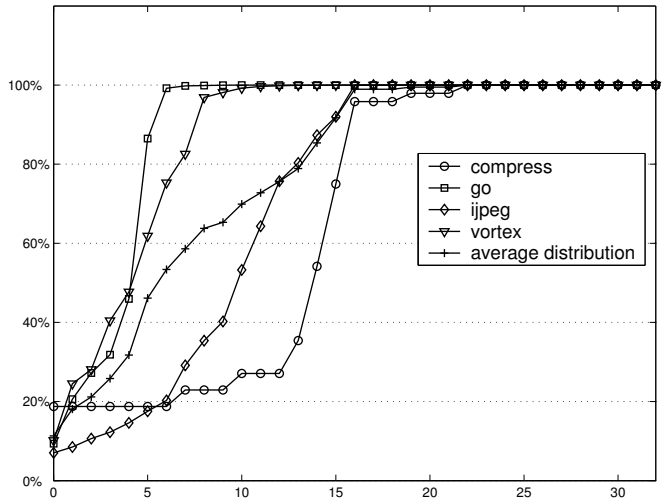


Figure 4.17: Proportion of operands having SBC below given number

These two imbalances in the operand distributions are very important factors in the design of a low-power multiplier. The following section introduces a low-power multiplier designed to exploit these unbalanced distributions.

4.6 A low power iterative multiplier

From the statistics of multiplier input vectors, on average about 60% of multiplication operands have SBCs below 8 bits and more than 90% of operands have SBCs below 16 bits. These characteristics can be exploited to increase speed as well as to reduce power consumption in multipliers. This goal can be achieved using an asynchronous 'early termination' algorithm.

The principle of an early termination algorithm is simple — For a $m \times m$-bit multiplication, if the most significant n bits of the m-bit multiplier are all 1s or 0s, the corresponding n partial products can be discarded and the multiplication speeds up by n/m. According to the statistical results mentioned above, more than 90% of multiplications can have their times cut by 50% using the early termination algorithm.

Tree architectures, however, cannot use an early termination algorithm, since all partial products are fed in at the same time; a sequential architecture [48] supports early termination but loses the speed advantage of a tree architecture. Based on these tradeoffs between speed, area and power consumption, a shift-iterative architecture is a compromise solution.

4.6.1 A shift-iterative architecture

Figure 4.18 shows a 32-bit multiplier using a shift-iterative architecture. The multiplier compresses 8 partial products in one cycle so, in the worst case, the multiplier needs 4 cycles to complete a 32×32-bit multiplication. The multiplier has a two-stage pipelined datapath, the first is an 8-2 tree adder; the second pipeline stage contains a 4-2 tree adder and a shift register. After each calculation cycle, the shift register shifts the results to the right by 8 bits and feeds the shifted results back to the 4-2 tree adder. With pipeline control, the first stage and the second stage execute in parallel, increasing speed by a factor of two for the same hardware cost. If the delay of the shift register can be ignored, the multiplier has the same speed as a 32-2 tree multiplier.

The early termination algorithm can be implemented using the shift-iterative architecture as follows: if the most significant 8 bits of a multiplier are all 1s or 0s, one cycle of the multiplication can be sped up and the multiplier moves an 'early termination cycle'. Early termination cycles do not process data but only shift the results to the correct position, so they are faster and consume less power

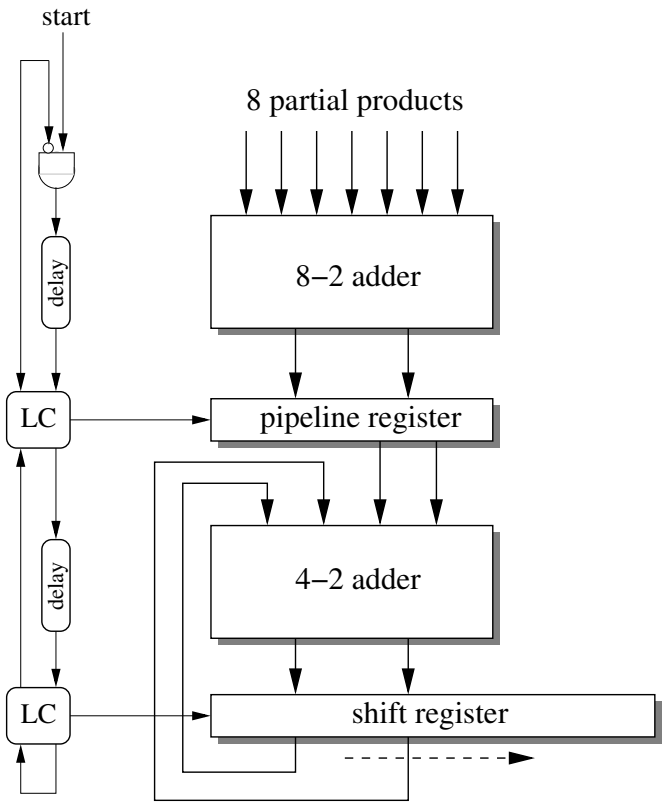


Figure 4.18: A shift-iterative architecture

than normal calculation cycles. Two or three cycles can be sped up if the next one or two groups of eight bits have the same values as the top group.

4.6.2 A radix-2 algorithm

High-radix algorithms are often used to speed up multipliers by reducing the number of partial products; an example is the commonly-used modified Booth's algorithm as previously described. The modified Booth's algorithm scans the multiplier bits in pairs and generates one partial product per pair, thus reducing the number of partial products by a factor of two. However, previous experiments have shown that a modified Booth's multiplier is less power-efficient than a non-Booth's multiplier [50][2] as shown in Table 4.5.

Why is the modified Booth's algorithm not power-efficient even though it reduces the number of partial products and one-bit full adders? Through analysis, one reason is identified as follows: with the modified Booth's algorithm, partial products have a high probability of being $-1\times$ or $-2\times$ the multiplicand. These

inverting operations introduce many transitions into adder trees (similar to the phenomenon mentioned in Section 4.3). For example, if the multiplicand MD is equal to 1, $-MD$ and $-2 \times MD$ switch all of the bits up to the most-significant bit, causing 32 transitions in each operation. Based on the statistics presented previously, it is known that most operands are small positive numbers. The modified Booth's algorithm causes a lot of switching activity even for a small multiplication, so it may not be a good algorithm for use in low-power multipliers.

A second reason for questioning its use is that, because of their long output wires and big fan-out, Booth's encoders and partial-product generators consume a lot of power themselves and generate glitches which propagate through the whole multiplier datapath. Race-free encoding schemes were proposed to ameliorate this problem [60].

Consequently Booth's algorithm may not be a good choice in the design of a low-power multiplier. Although Booth's encoders and partial-product generators reduce the number of partial products by a factor of 2, the same result can be achieved by adding another row of 4-2 adders to an adder tree. As shown in Figure 4.19, two partial product generators can be replaced by a 4-2 adder and four AND-gates (or 2-1 multiplexers).

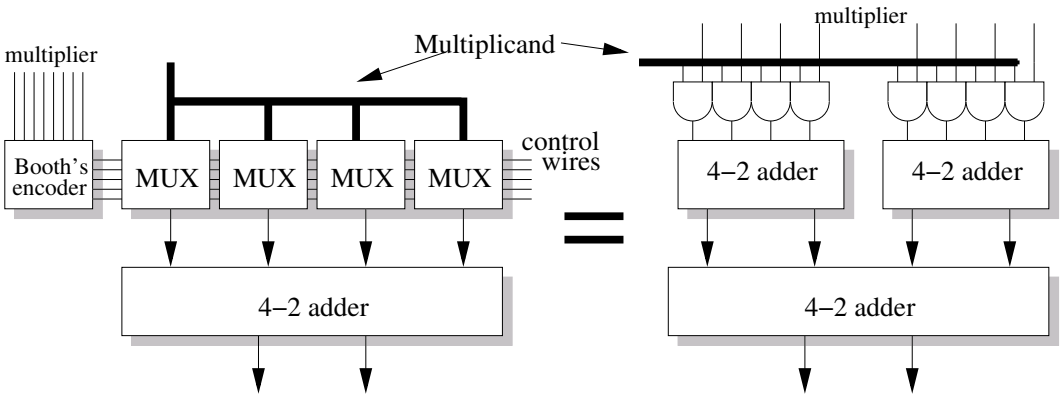


Figure 4.19: Two kinds of 8-2 adder trees

The speed and area requirements of these two circuits can be compared, taking a race-free encoding multiplier [60] as an example. A race-free partial-product generator [60] requires 24 transistors. A compact 4-2 adder using double pass-transistor logic (Figure 4.14(b)) requires 44 transistors. The 2-1 pass-transistor multiplexer requires 3 transistors. It appears that the non-Booth's circuit is slightly bigger than the race-free encoding circuit ($44 + 3 \times 4 > 24 \times 2$). However,

the race-free encoding circuit has four control wires — NEG, $\times 1$, $\times 2$, Z — spreading through the whole partial product generator. If the wires are taken into account, the non-Booth's circuit is smaller than the race-free encoding circuit. Even if the non-Booth's circuit is compared with a small Booth's encoding circuit [60], the non-Booth's circuit incurs a small hardware overhead. Through my experiments, the non-Booth's encoding circuit was found to be 18% faster than the Booth's encoding circuit because the long control wires put heavy loads on their drivers and make the Booth's encoding circuit slow. The regular structure of the non-Booth's encoding reduces the possibility of glitches which is also good for power-efficiency.

A software model was developed in this design to compare the switching activities of non-Booth's and Booth's tree multipliers. The model was used to calculate the numbers of transitions in 4-2 adders with 10000 positive input numbers having different SBCs. The results are shown in Table 4.7.

Table 4.7: Numbers of transitions in Booth's and non-Booth's multipliers

SBC	**8**	**16**	**24**	**32**
Non-Booth's	140978	634928	1362437	2221972
Booth's	490987	1243814	1874260	2511356
Ratio	1/3.50	1/1.96	1/1.38	1/1.13

As can be seen from Table 4.7, even for random operands, non-Booth's multipliers are somewhat more power-efficient than Booth's multipliers. However, for inputs having a small SBC, non-Booth's multipliers are significantly better.

4.6.3 Sign-changing Algorithm

The results in Table 4.7 are based on the assumption of positive inputs. Although most inputs are positive for the four benchmark programs previously described, some applications may have a higher incidence of negative numbers. The low power advantage of non-Booth's multipliers would be more convincing if the power-efficiency could be maintained for negative inputs. To achieve this goal, an algorithm is needed which can change the signs of both multiplicands and multipliers if they are negative. A sign-changing algorithm is proposed to achieve this objective.

The sign-changing algorithm for multipliers is based on the following functions:

$$a \times b = \overline{a} \times \overline{b} + \overline{a} + \overline{b} + 1 \tag{4.5}$$

$$a \times b = \overline{\overline{a} \times b + b - 1} \tag{4.6}$$

$$a \times b = \overline{a \times \overline{b} + a - 1} \tag{4.7}$$

The sign of an input can be tested by checking its most significant bit (bit 31). If both multiplicand and multiplier are negative Equation 4.5 is used: invert both multiplicand and multiplier and send them to a non-Booth's multiplier. For array or tree multipliers, another row of 4-2 adders is needed to add $\overline{a}$ and $\overline{b}$. Then the partial sum and the partial carry are generated. A carry-propagate adder is used to add the partial sum and carry with a carry in of 1 to generate a final result.

If only one of the operands is negative Equation 4.6 or 4.7 can be used: the negative operand is inverted. The inverted operand and the positive operand are sent to a non-Booth's multiplier. An additional row of 4-2 adders is used to add the positive operand and 0xFFFFFFFF (the least significant 32 bits of '-1'). However, we are left with the problem of adding 0xFFFFFFFF00000000 (the most significant 32 bits of '-1'). We could, of course, have a long 4-2 adder row consisting 64 adders to do this, but this would need a lot of hardware and consume a lot of power. Instead, the 1s in 0xFFFFFFFF00000000 are distributed across the significant bits of the 32 partial products. Figure 4.20 illustrates the principle. This scheme is similar to the sign extension algorithm mentioned before with only one sign extension bit.

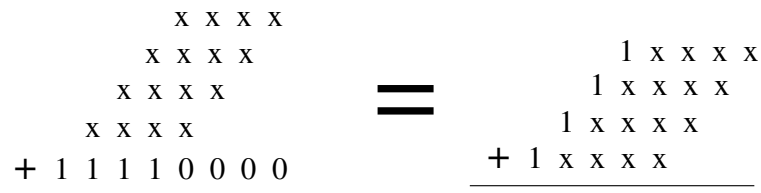


Figure 4.20: Distributing the higher-order 1s

Using the sign-changing algorithm, the non-Booth's multiplier retains its low power advantage for negative operands but another row of 4-2 adders is needed to add the two extra numbers. Fortunately, for the pipelined iterative architecture shown in Figure 4.18, this overhead can be avoided by putting these two extra

numbers in the shift registers (there are two 62-bit shift registers which hold the partial products to be sent to a carry-propagate adder) in Figure 4.18 at the start of a multiplication. For positive inputs, the shift registers are set to zero. In the first cycle of a multiplication, the 4-2 adder just propagates the two partial products from the pipeline registers to the shift registers. For negative inputs, in the first cycle, the 4-2 adder adds the two partial products with the two extra numbers. Therefore, the implementation of the sign-changing algorithm in the pipelined iterative architecture needs only two rows of inverters and multiplexers, which represents a very small hardware overhead.

4.6.4 Circuit implementation

The proposed multiplier was implemented using 4-phase bundled-data asynchronous logic. The asynchronous control circuit uses two latch controllers proposed in [46]. The latch controllers use level-sensitive latches as pipeline registers and they are normally closed. Both of these characteristics are good for low power design.

In the multiplier, the PowerPC603 master-slave register [18] is used for the shift registers. Although this register is power-efficient, and small level-sensitive latches are used as pipeline registers, the registers and their drivers consume about 58% of the total power when a multiplication needs only one normal calculation cycle. This is because large registers (about 200 register bits) put a heavy load on the control wires. The long control wires themselves constitute a large capacitance. If the capacitance of the registers and wires can be decreased, a power saving can be achieved. Because of the unbalanced SBC distribution, in most cases, the top 16 bits of pipeline and shift registers stay at 0, so there is no need to clock them and a 'split register scheme' can be used to reduce the register power. The registers are separated into 3 groups. The first group is $Bit_{32} - Bit_{16}$, the second group is $Bit_{15} - Bit_{8}$ and the third group is $Bit_{7} - Bit_{0}$ as shown in Figure 4.21.

Two control wires — $Pass0$ and $Pass1$ — are used to control how many bits should be clocked. Because the circuit already has a cycle detector to detect how many normal calculation cycles a multiplication needs, the only overhead incurred for splitting the registers is 2 multiplexers. Based on testing the split registers by simulating the schematic a 12% energy saving is achieved by using the split register scheme.

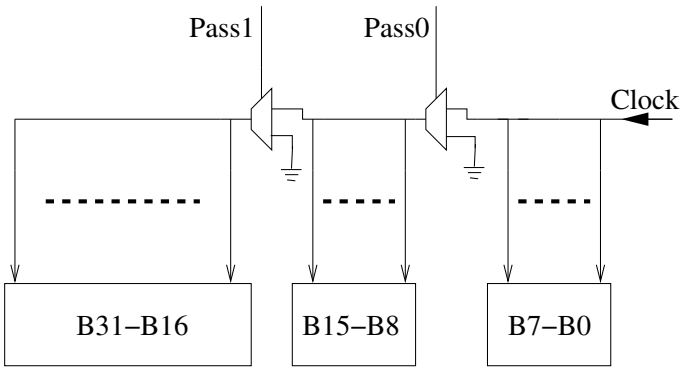


Figure 4.21: 32-bit split register organization

The split-register technique is similar to a technique used previously in a low power register bank [61]. However, it has not previously been applied to the design of latch- or register-based multipliers.

4.6.5 Experimental results

Four different multipliers were implemented for comparison. They are: a synchronous radix-2 multiplier without split registers (S2N), an asynchronous radix-4 Booth's multiplier without split registers (A4N), an asynchronous radix-2 multiplier without split registers (A2N) and an asynchronous radix-2 multiplier with split registers (A2S). The multipliers were simulated using HSPICE on a 0.18 micron CMOS technology at 1.8 V and 27^oC. The input vectors were 1000 pairs of numbers taken randomly from the 4 benchmarks as introduced before. The throughput was controlled at 100 million multiplications per second. The results on power consumption based on a schematics-level simulation are shown in Table 4.8.

Table 4.8: Power comparison of 4 multipliers

Multipliers	**S2N**	**A4N**	**A2N**	**A2S**
Power (mW)	16.17	9.56	7.35	6.47
Ratio	2.2	1.3	1	0.88

As can be seen from the table, asynchronous multipliers consume less than half the power of their synchronous counterpart. Radix-2 multipliers save 23% of the power of those using the radix-4 modified Booth's algorithm, and split registers save another 12%.

The A2S multiplier completes a multiplication requiring 4 normal cycles in 6.5 ns, resulting in a throughput of 150 million multiplications per second, this is the worst case however. It completes a multiplication requiring 3 normal cycles in 6.0 ns, one requiring 2 normal cycles in 5.2 ns, and one requiring 1 normal cycle in 4.3 ns. Based on the unbalanced SBC distribution, A2S has an average throughput of more than 200 million multiplications per second.

To summarise, a new design for a low-power multiplier is proposed based upon the observations of imbalanced input operands. It uses an area-efficient pipelined iterative architecture employing asynchronous control and an early-termination scheme. The new multiplier dissipates power 'on demand' — for small operands, it consumes less power; for large operands, it consumes more power. The experiments show that asynchronous control reduces the multiplier's power consumption by more than half.

A radix-2 non-Booth's algorithm avoids the inverting operations necessary in multipliers using the modified Booth's algorithm, thereby avoiding signal transitions. A sign-changing algorithm is used to ensure that the non-Booth's multiplier retains its low-power advantage for negative operands. Compared to a Booth's multiplier, the non-Booth's multiplier has a 23% reduced power dissipation.

The split-register scheme helps the multiplier to save a further 12% of its power dissipation. The total switching capacitance is decreased by splitting a large register into several small segments. In most cases, only a small part of the register is driven by the clock signal on each cycle, thus saving power.

The new asynchronous multiplier demonstrates the low-power advantage of asynchronous logic resulting from its fine-grain control. Although the multiplier is an iterative asynchronous multiplier, the non-Booth's algorithm used here is equally suited to synchronous multipliers, including array and tree multipliers. The split datapath scheme is also valid for any register- or latch-based multiplier with an unbalanced distribution of input operands.

4.7 Summary

The basic design issues of low-power arithmetic unit design, such as logic styles, data representations and different arithmetic architectures have been discussed in the chapter. Moreover, the chapter focuses on the designs of two low-power arithmetic units — an asynchronous carry-lookahead adder and an asynchronous

iterative multiplier using an early-termination scheme. Both of the circuits are designed to achieve low power consumption by exploiting the specific data characteristics found from the evaluation of benchmarks.

The next chapter will present some low-level design issues of low-power memories.

Chapter 5

A low-power embedded SRAM macro design

To meet the requirement for high speed and low power consumption, static random-access memories (SRAMs) are integrated into data processing circuits. This avoids long latencies and high power consumption due to input/output (I/O) pins and long wires for off-chip memory interconnections. On-chip SRAM sometimes dominates the silicon die area and power consumption of an embedded SoC chip. Therefore, a power-efficient SRAM is extremely important in the design of a low-power embedded processing circuit.

Low-power SRAM techniques are applied either at the circuit level or at the architecture level. At the architecture level, the basic principle for memory power saving is 'the principle of locality' as introduced in Section 2.4.2. A low-power SRAM is usually organized as a hierarchy. Since memory blocks at lower hierarchy levels are smaller, faster and more power-efficient, high-level low-power memory techniques focus on more efficient memory hierarchy architectures, which have a low access rate at high levels in the memory hierarchy (a high 'hit' rate for low levels in the memory hierarchy). This is the basic idea behind many high-level low-power memory architectures. The low-power SRAM techniques in this chapter focus on the low circuit level.

This chapter presents the design of a low-power embedded SRAM macro: Section 5.1 overviews the basic design issues for low-power SRAMs; Section 5.2 describes low-power bit-line write techniques; Section 5.3 proposes a dual-rail row decoder; Section 5.4 gives the architecture and timing control of the proposed SRAM macro; Section 5.5 presents layout and experimental results; Section 5.6

summarises the chapter.

5.1 SRAM design overview

5.1.1 Conceptual SRAM structure

A basic SRAM architecture is shown in Figure 5.1. The SRAM contains a large number of one-bit memory elements, called 'memory cells'. Due to its high density and stability in state holding, a 6-transistor memory cell [49] is most commonly used in on-chip SRAMs. Each memory cell contains a pair of cross-coupled inverters which hold state. Two n-transistors are connected to the inverters for read and write accesses under the control of a $word-line$ signal. The values to be sent to and read from a memory cell go via the access n-transistors to two bit-lines — bit and $\overline{bit}$.

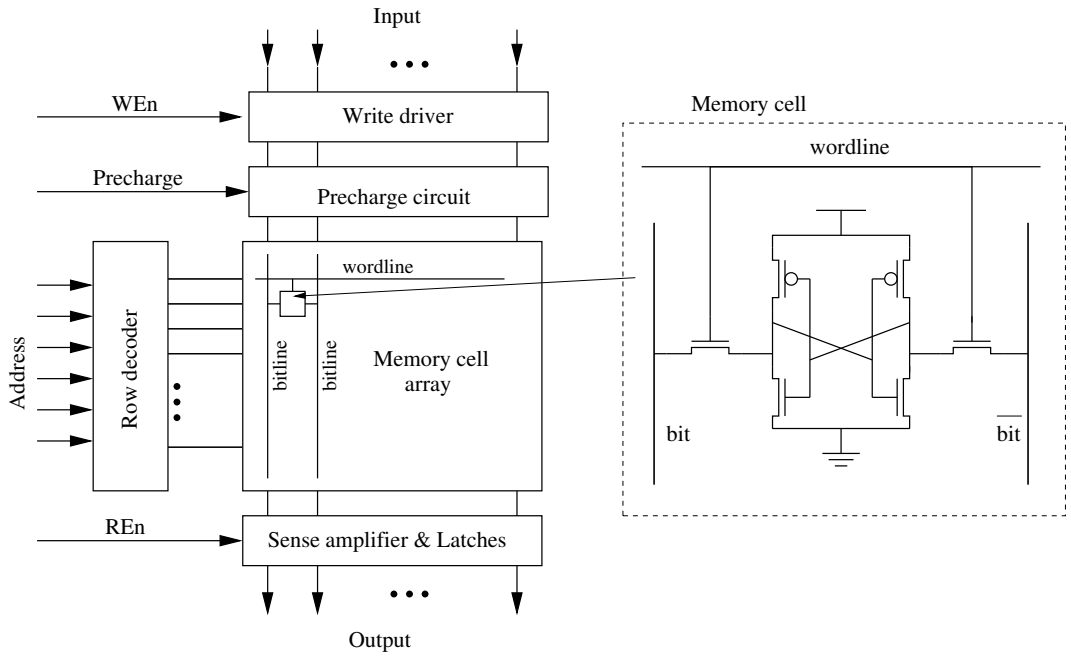


Figure 5.1: A conceptual SRAM architecture

During a write, the two bit-lines are driven by a complementary new input value. When the word-line is activated, the new input value is written into the memory cell by overpowering the previous value held in the cross-coupled inverters. After the write, the word-line returns to 0, isolating the memory cell from the bit-lines.

Since n-transistors are good at passing a logic low ('0') and weak at passing a logic high ('1'), a precharge bit-line scheme is used for reads. Before each read, the bit-lines are precharged to 1 and during the read, the word-line becomes 1 to allow read accesses. Only one of the bit-lines is pulled down by the memory cell; the other stays high.

The memory cells of an SRAM are organized as an array. Each row of the array contains m (m is the word-length of the SRAM) memory cells. The $word-line$ connections of the memory cells in a row are connected to make the memory cells accessible simultaneously, forming a word of the SRAM. The words of the SRAM are arranged in a column; the bit-lines of the memory cells in the same column are connected together to allow them to be accessed via one read and one write port. At any time, only one word can be accessed under the control of the $1-of-2^n$ word-lines (2^n is the number of words of the SRAM) generated by a row decoder.

Specific peripheral circuits for read and write operations are needed to connect the SRAM to its environment. Since memory cells are optimized to minimize their size, their drive abilities are weak, resulting in a slow discharge rate for reads. To speed up the read operation, sense amplifiers are often used to amplify the low voltage swing of bit-lines.

5.1.2 Low power SRAM design techniques

Various methods [62][63] have been proposed to minimize the power consumption of SRAMs; many of them are based on a few basic principles: minimizing the active capacitance and reducing voltage swing.

Minimizing the active capacitance of an SRAM can be achieved by using high-density memory cells and smaller and more efficient decoders, write drivers, sense amplifiers and output latches. The capacitances of word-lines and bit-lines can be minimized by partitioning a unified memory block into a number of smaller sub-blocks as will be introduced later in this section. A dual-rail decoder having a smaller active capacitance than conventional SRAM decoders will be described in Section 5.3.

Equation 2.1 in Section 2.1 defined the dynamic power consumption of a CMOS circuit, which is directly proportional to $C \cdot V^2$ where V is its voltage swing and C the active capacitance. Therefore, a reduction of the voltage swing can achieve a significant power saving.

For conventional SRAMs, during a read operation the bit-lines do not range over the full voltage swing because the driving ability of RAM cells is too weak to charge the high-capacitance bit-lines. Sense amplifiers are used to amplify the voltage difference of the bit-lines to obtain a full voltage swing; these not only increase speed but also reduce the power consumption of read operations. For write operations, the bit-lines normally switch fully because the signal on the bit-lines must overpower the memory cell flip-flop with new values. Thus, a write operation usually consumes more energy than a read in conventional SRAMs if the number of rows is large. The size of an SRAM also has an effect on performance since the capacitance of the bit-lines increases with the number of rows in it; the drive strength of the write drivers must be increased to maintain a given write speed. This further increases the power consumption of write operations; for example, the write drivers use 90% of the write power when the number of words in a column reaches 256 [64]. Therefore minimizing write power is critical in the design of power-efficient SRAM. Low-power write techniques are presented in the next section.

5.1.3 Block partitioning

Increasing the SRAM size results in corresponding wire-length increases in both word- and bit-lines. The RC delay of a long wire grows as the square of the number of memory cells it connects, and its power consumption grows linearly. Therefore, long word- and bit-lines result in an increase in physical capacitance and power consumption and a degradation in speed. An efficient way to minimize the capacitance of long word- and bit-lines is to partition a large memory block into smaller sub-blocks which will have shorter word- and bit-lines. Since an SRAM block is a two-dimensional array, it can be divided horizontally and vertically.

A vertical block partitioning scheme is also known as a Divided WordLine approach (DWL) [65] which divides a long word-line of a unified SRAM into a number of segments. The segments are independently activated under the control of block select signals. Figure 5.2 shows the architecture of a DWL approach where a w-wordlength SRAM is horizontally divided into k sub-blocks, each of which has a wordlength of w/k. The row decoder is designed to have two stages — one stage generates the global word-lines and the other one generates the block select signals. During each memory access, the active word-line and block select

signal are ANDed to activate the corresponding local word-line of a sub-block. Since the local word-lines are only $1/k$ of the length and connect to $1/k$ of the memory cells of the global word-lines without the DWL approach, their RC delay is reduced to $1/k^2$ and power consumption is reduced to $1/k$.

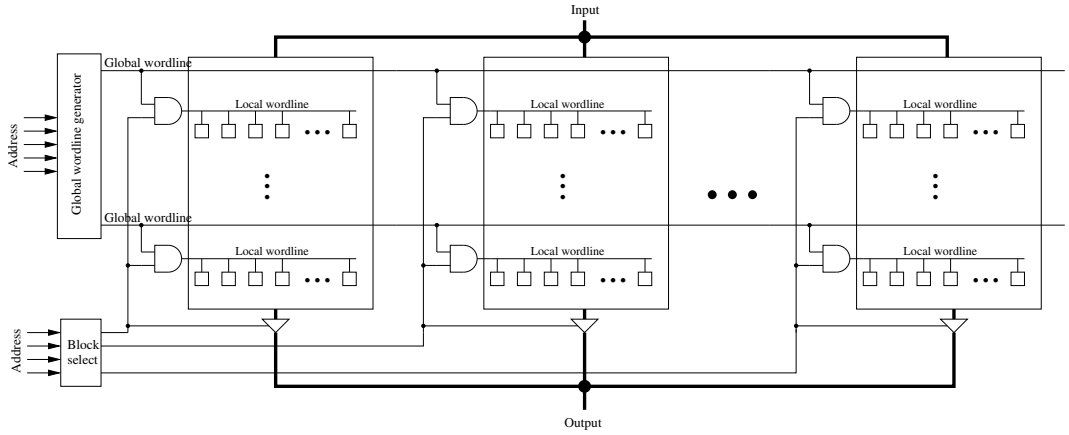


Figure 5.2: A divided word-line approach

Although the length of a global word-line is nearly as long as that in a unified SRAM, it only connects to k local word-line drivers instead of w memory cells. Normally, k is much smaller than w, so the global word-lines have a much smaller capacitance, resulting in lower RC delay and power consumption. To reduce the RC delay further, the global word-line decoder is normally located in the middle of the sub-blocks, and the global word-lines use low-capacitance high-level metal wires.

The concept of DWL can be extended to sub-divide global word-lines and block select signals when w is very big. A Hierarchical Word Decoding (HWD) scheme has been proposed [66] to cope with the problem of very high-capacitance word-lines. The HWD approach divides a long word-line into multi-level shorter word-line segments, forming a word-line hierarchy. Each wire segment has a reduced load and wire capacitance, thus improved speed and power-efficiency.

Block partitioning can also be applied in a vertical direction; this approach is called a Divided Bit-line scheme (DBL), aiming at minimizing the power consumption and increasing the speed of an SRAM by dividing its bit-lines into a number of shorter segment groups. Figure 5.3 shows a DBL architecture, where a long bit-line is divided into shorter local bit-lines. The local bit-lines are connected to the global bit-lines via pass-transistor gates, allowing the memory cells in one column to share the same input/output port. The concept of hierarchy

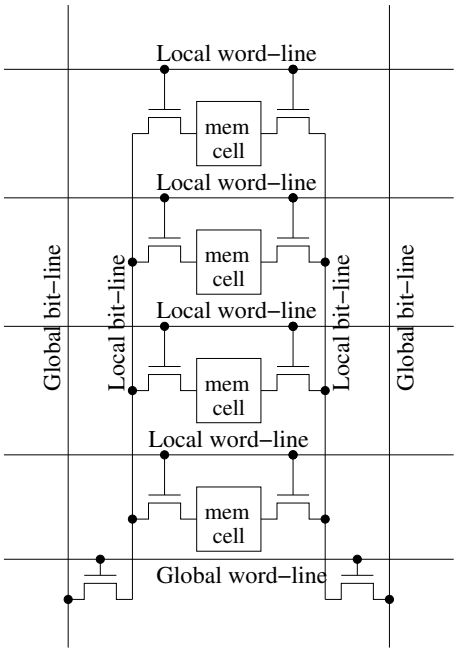


Figure 5.3: A divided bit-line approach

can also be used to form a hierarchical divided bit-line architecture.

Partitioning an SRAM incurs an area overhead at the boundary of the sub-blocks in terms of block-select drivers and pass-transistor multiplexers. This area overhead results in extra length and capacitance in word-lines and bit-lines, which degrades the performance and power-efficiency gained by partitioning. Therefore, the sub-blocks must not be too small.

5.2 Low-swing write techniques

The critical problem in low-swing writes is overwriting the memory cell flip-flop while maintaining the stability of the memory cell during reads. Two general techniques have been proposed to achieve a low-swing write operation:

- Reduce the bit-line reference voltage;
- Use memory cells which are more 'input-sensitive'.

For conventional embedded SRAM design, bit-lines are referenced to V_{dd} by precharging and, during writes, the write drivers discharge half of the bit-lines to ground. Thus, the write power can be minimized by reducing the bit-line reference voltage. Allowersson and Andersson [67] proposed a technique where

they used a low bit-line reference voltage of 0.5 volts with a supply voltage of 5 volts. When a word-line is activated, a small bit-line differential is propagated to the internal cell nodes through the access n-transistors. When the word-line is turned off, the small voltage differential is amplified by the positive feedback of the cross-coupled inverters. To maintain a stored Boolean value, a lower word-line voltage is used during reads, thus weakening the access n-transistors and preventing a spurious discharge of the internal cell nodes. The penalty of this technique is that when the access n-transistors are gated by a lower voltage, their read accesses become slower. Mai et al. use the same principle except that they use a different bit-line reference voltage [68]. They proposed a half-swing pulse-mode technique which can easily generate the voltage of $V_{dd}/2$, so they use $V_{dd}/2$ as the bit-line reference voltage. During writes, half of the bit-lines are discharged to ground and since the bit-lines use only a half-rail swing, 75% of the bit-line write power can be saved compared to conventional techniques. However, using a $V_{dd}/2$ reference for the bit-lines potentially leads to cell instability during reads because of the leakage current from logic high internal nodes. This problem can be solved by an increased internal cell voltage.

An advantage of low-swing write techniques based on reducing the bit-line reference voltage is that they use RAM cells that are the same as those used in conventional SRAMs, so no area penalty is incurred in the memory cell arrays. However, these techniques have disadvantages as follows:

- Special techniques must be used to generate the bit-line reference voltage and the internal cell voltage.
- Since the bit-line reference voltage is reduced, the differential voltage between the cell internal nodes is smaller than usual, which inevitably results in a longer write delay.

An alternative method has been proposed to achieve a low-swing write, using memory cells with greater input sensitivity. Wang et al. [69] proposed a current-mode RAM cell as shown in Figure 5.4 (a). This RAM cell is similar to the conventional 6-transistor RAM cell, except for an equalizing n-transistor — M_{eq}. During read, the cell-equalization signal *weq* is kept low, and the RAM cell acts like a conventional RAM cell. The equalizing n-transistor clears the contents of the memory cell prior to a write, making it easy for the bit-lines to drive the internal cell nodes even with a small voltage differential. P-transistors are used

as access transistors, which results in a very compact cell area. However, since the gain factor of p-transistors (β_p) is smaller than that of n-transistors (β_n), the RAM cell has a long access delay during read and write. If n-type access transistors are used, the area will be much bigger since each RAM cell would have 5 n-transistors and this makes regular layout difficult to design.

Another input-sensitive RAM cell is proposed by Kanda et al. [64] as shown in Figure 5.4 (b). This RAM cell is called a sense-amplifying RAM cell (SAC). During read, SLC is kept high, making the RAM cell act like a conventional RAM cell. During writes, SLC is made low and the V_{SS} switch is turned off before the word-line turns on. Even with a very small voltage differential between the bit-lines, the cell can switch state because the n-transistor drivers do not draw current. After the word-line goes low, the V_{SS} switch is turned on and the small voltage differential is amplified to full swing inside the RAM cell.

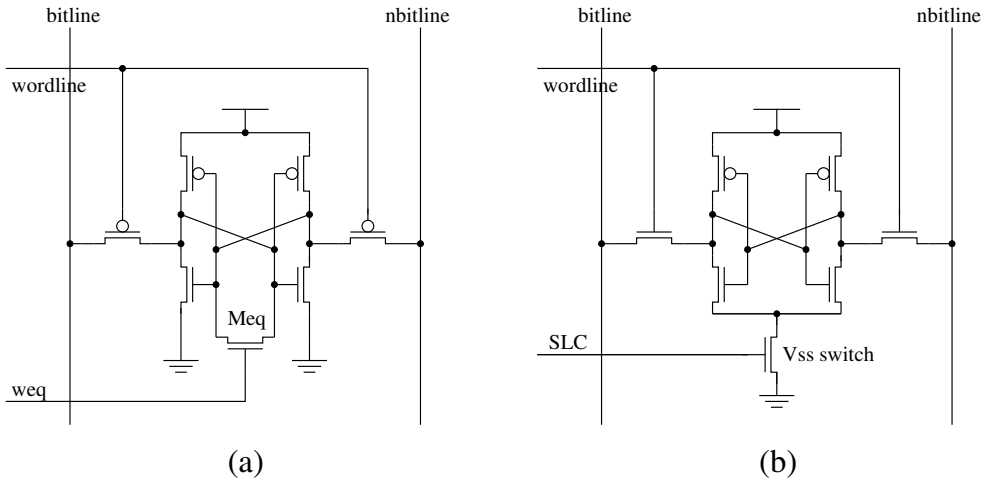


Figure 5.4: 'Input-sensitive' RAM cells

The SAC technique is similar to another technique proposed by Amrutur [70] as shown in Figure 5.5. All the cells in a given row share a virtual ground (vgnd), which is driven by an AND-gate. During read, the virtual ground is driven low, making the SRAM act like a conventional SRAM. Before write, the virtual ground is first driven high to reset the contents of the cells. Then the word-line is pulsed high, transferring the small swing bit-line signal to the internal cell nodes. After that, the virtual ground is driven low, causing the cells to complete the full swing. However, this technique has problems during read. Since all the cells in the row share the virtual ground, the read current of the cells gathers on the virtual ground and flows to ground through an n-transistor inside the AND-gate. This

causes two problems:

- The current is so big that it needs a wide metal wire for the virtual ground, wider than the RAM cell height;
- The current raises the voltage of the virtual ground, resulting in read instability. One solution for this technique is to distribute the n-transistor of the AND-gate to each cell in the row, as is done in the SAC scheme.

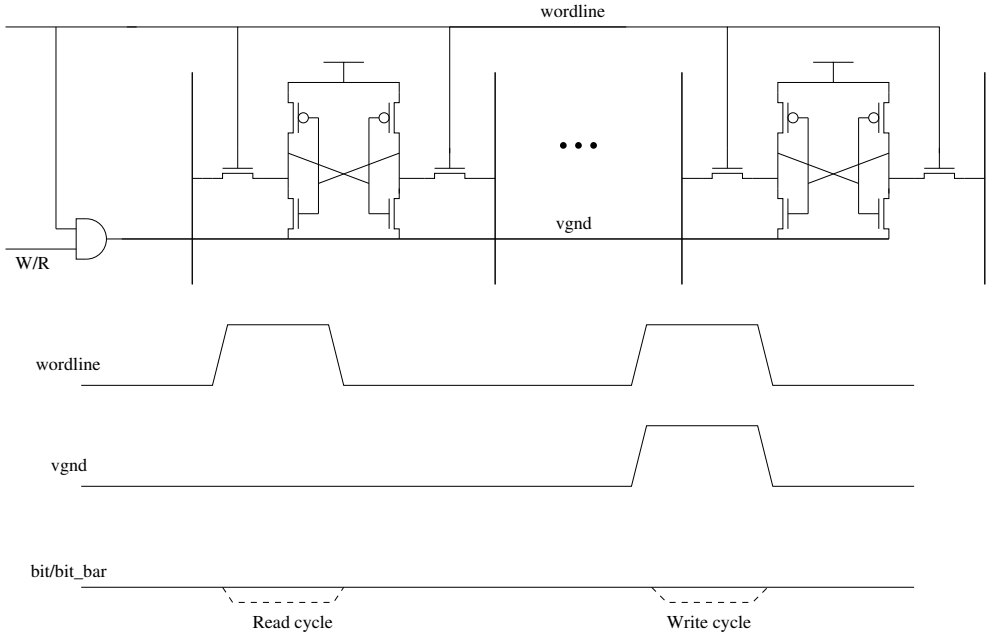


Figure 5.5: Amrutur's low write scheme

Comparing these techniques, the SAC scheme is used in the design presented here for low-swing writes because:

- The RAM cell is similar to the conventional 6-transistor RAM cell;
- The write and read delays are relatively short compared to the other techniques;
- The RAM cell maintains a reasonable static noise margin.

Sense-amplifying RAM cell design is very important using the SAC scheme, since memory cell design is a trade-off between silicon area, read speed and noise margin. The basic requirement for RAM cells is safe reading. Figure 5.6 (a) illustrates the read current path of a sense-amplifying cell. Let us assume that

internal node A is low, B is high and the bit-lines are precharged to $V_{dd} - V_T$ (V_T is the n-transistor threshold voltage.). If T_2 stays off during read, the state holding inside the RAM cell will not be affected and read operations are safe.

It is easy to figure out that all the three n-transistors — T_{access}, T_1, T_{ss} — are in their linear regions, since they all satisfy the condition: $0 < V_{ds} < V_{gs} - V_T$. The n-transistors — T_{access}, T_1, T_{ss} — are fabricated using the same nMOS technology and have the same channel lengths, so their channel impedances are inversely proportional to their widths. The equivalent circuit of that in Figure 5.6 (a) is shown in Figure 5.6 (b). From Figure 5.6 (b),

$$V_{dsT1} = (V_{dd} - V_T) \times \frac{\alpha \frac{1}{W_1}}{\alpha \frac{1}{W_{access}} + \alpha \frac{1}{W_1} + \alpha \frac{1}{W_{ss}}}$$

$$= (V_{dd} - V_T) \times \frac{1}{\frac{W_1}{W_{access}} + \frac{W_1}{W_{ss}} + 1} \tag{5.1}$$

$$V_{gsT2} = V_{dsT1} \tag{5.2}$$

where α is a constant defined by the CMOS technology. Transistors normally have the minimum channel length.

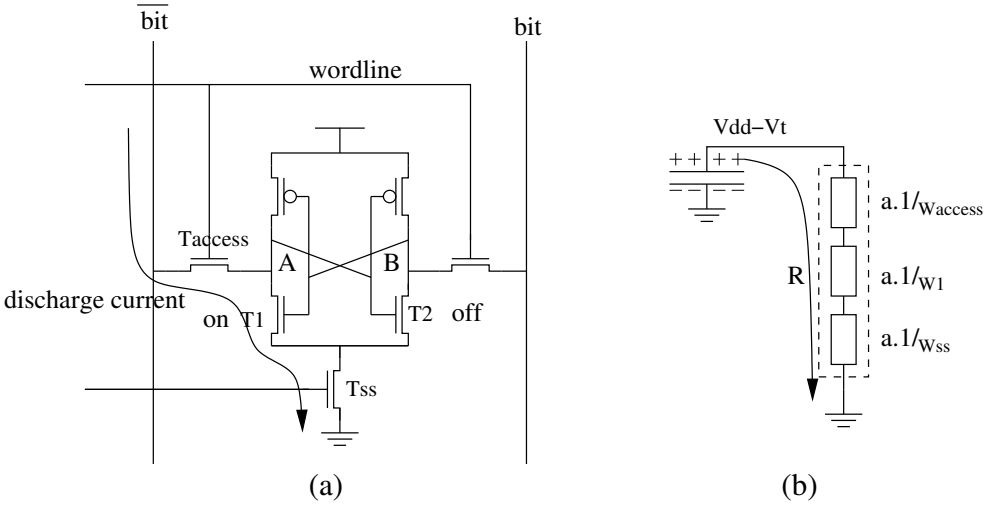


Figure 5.6: Discharge current during read

The property that guarantees T_2 stays off is $V_{gsT2} < V_T$. Since the width of T_{access} is normally designed to be the minimum, from Equation 5.1, increasing the width of T_1 and reducing the size of T_{ss} can reduce V_{gsT2}, thus increasing the read margin. However, the contribution to the increased read margin coming

from reducing the size of T_{ss} is relatively less, and a small size of T_{ss} also results in a small discharge current and big read delay. This can also be deduced from Figure 5.6 (b).

$$T_{discharge} \propto C_{bit-line} \times \Delta V \times R$$

$$T_{discharge} \propto \alpha \times (\frac{1}{W_{access}} + \frac{1}{W_1} + \frac{1}{W_{ss}}) \tag{5.3}$$

So the selection of the channel width of $T_{ss}$is also a trade-off between speed and size.

Another concern in the SAC RAM cell design is how to fit T_{ss}. If each cell contains a T_{ss} transistor, the area overhead of the RAM cell array is large. Kanda et al. put one big T_{ss} transistor for every 4 RAM cells. The same scheme is used in the proposed SRAM macro, but the transistor sizes are changed. Figure 5.7 illustrates the architecture and transistor sizes. The p-transistors and access n-transistors are set to minimum size. W_1 and W_2 are set to $2 \times W_{access}$. W_{ss} are set to $8 \times W_{access}$. From experiments and calculation, the penalty of the SAC RAM cell includes a 12% area increase, a 25% noise margin decrease and a 7% read latency increase.

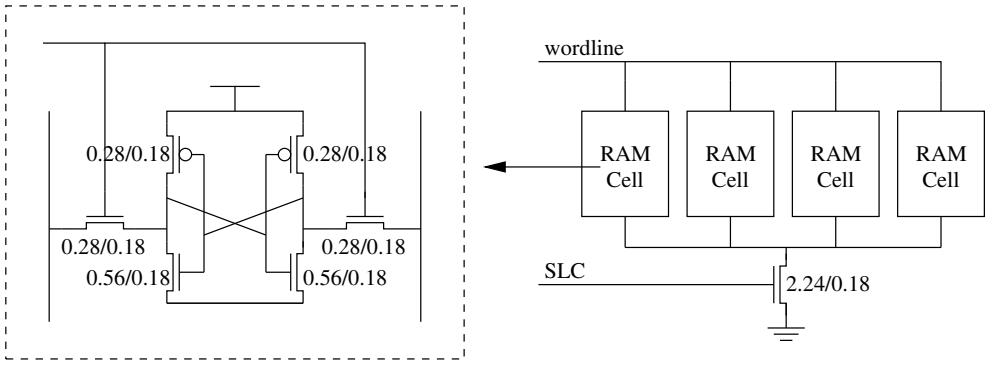


Figure 5.7: Shared T_{ss} scheme to reduce area overhead

5.3 A dual-rail decoder

In large SRAMs, there are two kinds of decoders — row decoders and column decoders. The row decoder activates one of the word-lines in a block, which connects the RAM cells of this row to the bit-lines. The column decoder controls a large multiplexer which connects one bit-line column to the peripheral interface. A small embedded SRAM macro contains only a row decoder since all the RAM

cells are put inside a single column. Figure 5.8 illustrates a conventional two-level SRAM row decoder, which has an n-bit input address and activates one of 2^n word-lines. To improve speed, two first-level predecoders are used, which decode $n/2$ address bits to drive one of $2^{n/2}$ predecode lines. The second-level decoder is one column of 2-input AND gates. Each of them ANDs two predecode lines to generate a word-line signal.

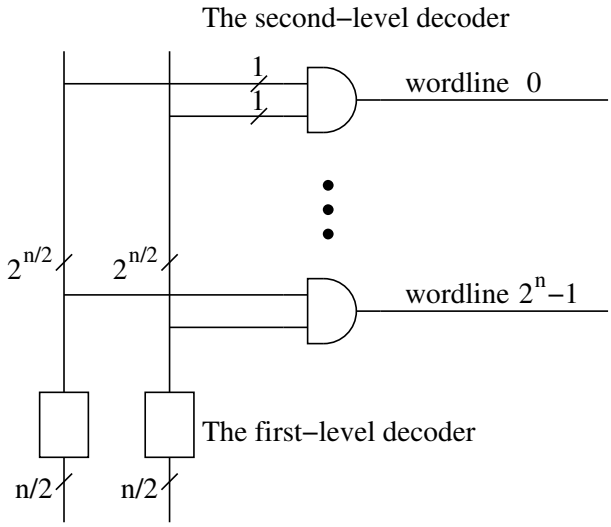


Figure 5.8: A two-level decoder

However, the decoder shown in Figure 5.8 has two problems:

- One of the word-lines is always active;
- Two word-lines may be active simultaneously for a short period when the address bits change.

To shorten the active duty-cycle of the word-lines and to prevent the word-lines overlapping, a pulsing word-line technique is needed. A pulsing word-line can be achieved either by gating the second level AND gates with a pulsed enable signal as shown in Figure 5.9 (a) or by putting an address transition detection (ATD) pulse generator on each word-line as shown in Figure 5.9 (b). The drawbacks of using a pulsed enable signal are:

- The enable signal is a long wire with a big capacitance and its fan-out is 2^n;
- The second level AND gates have three n-transistors stacked, which increases the delay.

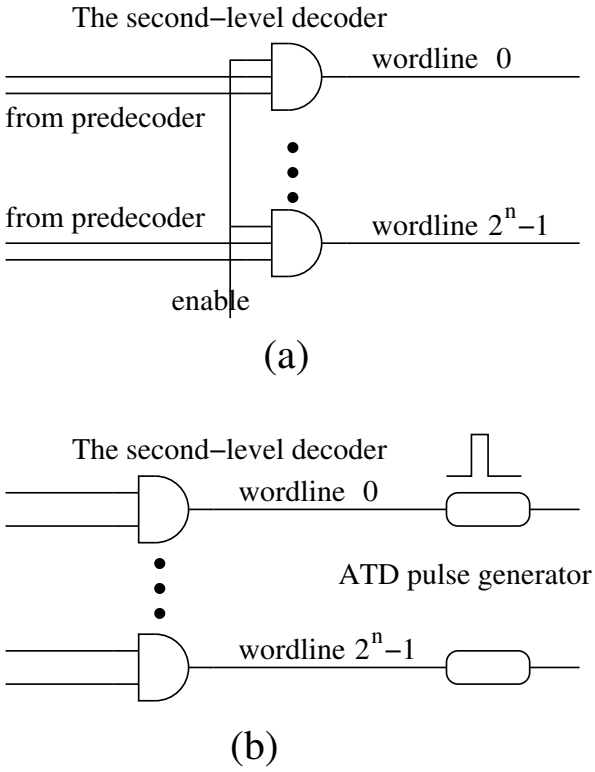


Figure 5.9: Pulsing word-line techniques

Using the second technique, the pulse generators increase the size of the decoder. To overcome the problems mentioned above, a novel dual-rail row decoder is proposed as shown in Figure 5.10 (a). Here, the pulsed enable signal does not gate the last-level AND gates but gates the inputs of the address bits and their complementary bits, giving the address bits a dual-rail format. The fan-out of the enable signal is therefore $2n$, which is much smaller than 2^n when $n > 4$. The enable signal is generated by an ATD pulse generator under the control of an asynchronous *Request* signal. The schematic of an ATD pulse generator is illustrated in Figure 5.10 (b). The pulse generator detects the rising edges of the *Request* signal, and then generates a positive pulse whose width is decided by the delay element. The predicating circuit in the dashed box prevents the pulse generator from generating several pulses from one input rising edge.

Since all the signals inside the dual-rail logic are pulsed, it is safe to use dynamic AND gates inside the decoder as shown in Figure 5.10 (c). Each dynamic AND gate contains only one p-transistor, which minimizes the size and the power consumption of the decoder. The circuit in Figure 5.10 (c) generates a word-line signal and an SLC signal which is gated by the *Read* signal, since SLC must

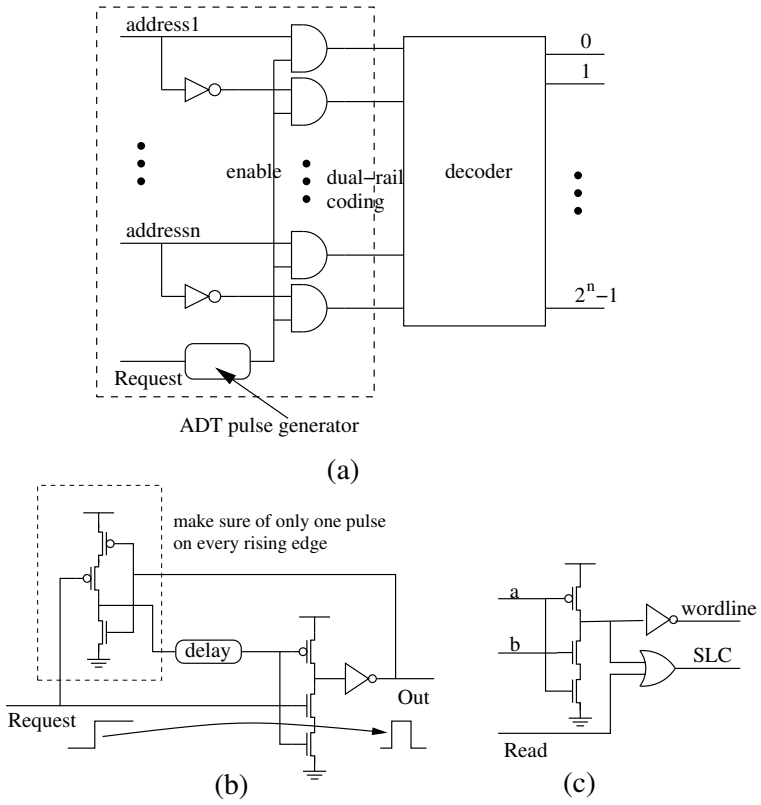


Figure 5.10: The proposed dual-rail decoder

stay low during read. The proposed dual-rail decoder is smaller and more power-efficient than conventional decoders and there is no speed penalty incurred by the new scheme.

5.4 Architecture and timing

The proposed SRAM is designed to be embedded with an asynchronous microprocessor, so its interface is designed to communicate with its environment using a normal 4-phase handshake protocol with a delay-matching scheme. The proposed SRAM is 2Kb (64×32 bits), and its architecture is illustrated in Figure 5.11 (a). Delay1 and delay2 are ADT pulse generators with asymmetric delays. They control the timing of the write enable signal (WEn) and the sense-amplifier enable signal (REn) respectively. The write and read timing diagrams are shown in Figure 5.11 (b) and (c) respectively. The acknowledge signal (ACK) is generated by a dummy column on the critical path of the SRAM. The ACK signal matches the read and write delays; its rising edges indicate that the current read

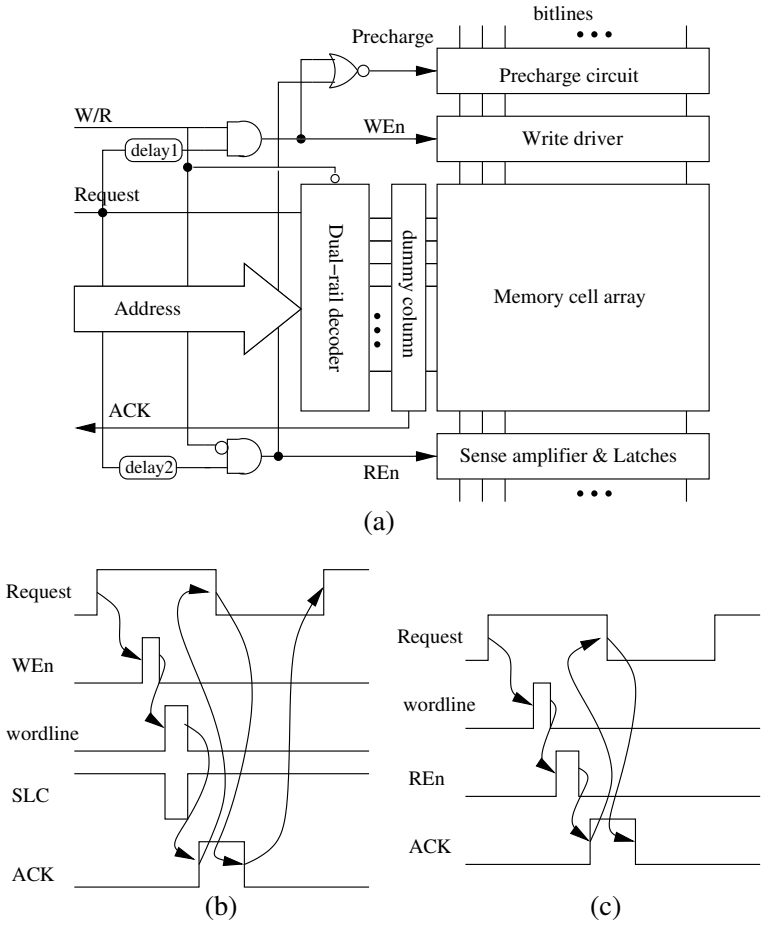


Figure 5.11: The architecture and timing of the proposed SRAM

or write operation is finished and its falling edges mean that precharging has completed.

One scheme [64] uses a special voltage to generate the bit-line voltage differential during write but this results in extra design complexity. To avoid a voltage converter, only one voltage is used in this SRAM macro. Since the discharging write driver n-transistors are operated in their linear region, the bit-line voltage changes are directly proportional to the discharge time and the width of the discharging n-transistors (assuming the transistor channel lengths are the minimum size). We have:

$$\Delta V \propto T_{discharge} \cdot W$$

The voltage differential can be controlled by adjusting the size of the n-transistors and the pulse width of WEn. Using a 0.18 μm CMOS technology, the capacitance attached to the bit-lines is 75 fF. To obtain a voltage differential of 0.2 volt, the

size of the discharge n-transistors is $W/L : 1.00\mu m/0.18\mu m$, and the WEn pulse width is 0.12 ns. Since the voltage differential only needs to be driven to 0.2 volts, the write driver size is small compared to conventional SRAM write drivers, which partly compensates for the area overhead of the sense-amplifying RAM cell array.

5.5 Layout and experimental results

The 2Kb embedded SRAM was designed using SGS-Thomson (ST) $0.18\mu m$ technology and a plot of the SRAM macrocell is shown in Figure 5.12. The circuit was simulated using HSPICE under typical operating conditions (1.8 V, 27^oC) and the experimental results and comparisons with a commercial SRAM macro (an ST SRAM macro [71]) are illustrated in Table 5.1. The ST SRAM macrocell uses the same CMOS technology and supply voltage as the proposed SRAM.

Table 5.1: Comparisons between the new SRAM and ST macrocell

	Proposed SRAM	ST macrocell	Ratio
Area (μm^2)	245 × 126	186 × 160	1/0.97
Delay/Write	1ns	2ns	1/2
Delay/Read	1.5ns	2ns	1/1.33
Power/Write	4.62μA/MHz	21.0μA/MHz	1/4.55
Power/Read	4.70μA/MHz	16.1μA/MHz	1/3.42

As can be seen from the table, the new SRAM improves on the power consumption of the ST SRAM macrocell by a factor of almost 4 with only 3% area overhead. With an asynchronous control circuit, the new SRAM can have different read and write delays. During write the new SRAM is 50% faster than the ST RAM, and during read the new SRAM is 25% faster. The ST RAM uses a more compact memory cell which is 11% smaller than the RAM cell used in this paper. The T_{SS} n-transistors result in a further 12% area overhead, so the proposed SRAM has a memory cell array 23% bigger than the ST memory cell array. However, these two RAMs have almost the same total size because the proposed dual-rail decoder and the small write drivers needed by the low-swing write compensate for the area overhead of the memory cell array. The decoder and the peripheral circuits in the proposed SRAM are about 34% smaller than those in the ST RAM.

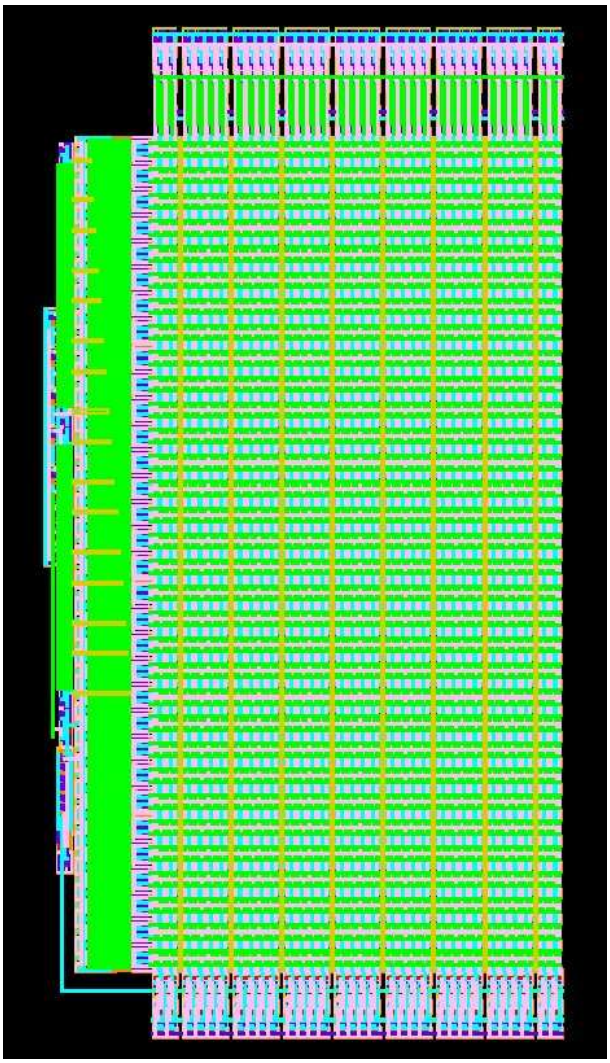


Figure 5.12: The layout of the proposed SRAM

5.6 Summary

This chapter has presented a 64 × 32-bit SRAM for embedded applications using a novel dual-rail decoder which operates efficiently with a low-swing write scheme. A factor of 4 improvement in power-efficiency was demonstrated over a commercial RAM macrocell. The proposed SRAM incurs little area overhead because the cost of the sense-amplifying cell array is compensated for by the small decoder and write driver. Different read and write delays are supported using asynchronous control logic, making the proposed SRAM about 30% faster than the commercial SRAM macro.

The SRAM macrocell achieves a low-voltage swing write by using the *SAC*

scheme. However, the SAC scheme has a limitation. As can be seen from Figure 5.4 (b), the content of a SAC RAM cell becomes unstable (or may change) once SAC becomes 1. Therefore, the RAM cell has to be rewritten by a new value once SAC becomes 1, thus all RAM cells in a word must be simultaneously written and partitioned write operations (for example, only writing one byte) are not allowed. This problem can be solved by the DWL approaches introduced in Section 5.1.3 at the cost of area overhead due to local word-line gating circuits.

The next chapter will present architecture-level optimizations for low-power processing in conventional RISC microprocessors. A hierarchical processing scheme will be proposed based on analyses of embedded processing characteristics. A CPU-coprocessor architecture will be implemented and tested to demonstrate the power-efficiency of hierarchical processing.

Chapter 6

Low-power hierarchical processing

The low-power processing techniques presented in the previous chapters are mainly focused on low-level circuits. This chapter presents a high-level architecture to minimize the power consumption of embedded processing. The chapter starts with an analysis of embedded processing characteristics; a hierarchical processing concept is then described. To demonstrate the power-efficiency of hierarchical processing, a hierarchical processing architecture and a RISC coprocessor are designed and analysed in this chapter.

6.1 Hierarchical processing

The purpose of this thesis is to explore techniques to minimize the power consumption of low-performance and cost-efficient embedded data processing systems which have a very low power budget. Although, as discussed in Chapter 2, high-performance processors have a similar power breakdown to that of conventional embedded microprocessors, the latter have characteristics which give potential for specific power-saving techniques.

The success of a general-purpose processor is determined by the commercial market, for which performance, binary compatibility with existing software and programmer friendly interfaces which attract software designers are the most

important design issues. Consequently, the main design philosophy of general-purpose processors is to maintain existing instruction sets while using the abundance of transistors provided by Moore's Law to increase the processors' performance. For example, Intel uses hardware to translate external x86 instructions to internal RISC-like instructions for performance while maintaining compatibility with x86 software. This approach achieves commercial success, but it is a power-inefficient approach.

The design considerations for an embedded processor are different because the characteristics of embedded applications are different from those of PC applications. Embedded processing systems often deal with continuous data streams for which average-performance is no longer the most important factor. Instead, a worst-case latency constraint for real-time processing is more important. Therefore, if the overall requirements for average throughput and worst-case latency can be met, efforts which further increase the processor's performance are not necessary.

Another difference between PC processors and embedded processors is in the support for programs. A PC processor is much more general-purpose than an embedded one, because it is impossible to know exactly what kinds of programs will be executed in a PC. The processor should therefore have similar processing abilities for all kinds of programs, requiring a lot of design effort as well as execution overhead. Embedded processors usually deal only with a very small number of applications, and among these applications, CPU occupation rates are highly unbalanced. Embedded processors may spend most of their execution time in executing only a few loops of a few programs. Consequently, a small number of important program kernels (program segments which may be small loops or function calls) have the greatest impact on the success of an embedded processor. Moreover, since the execution and power-efficiency of these important kernels are critical for overall performance and power consumption, the kernels are normally hand-optimized.

Figure 6.1 shows a program segment taken from the main part of an embedded JPEG program. The JPEG program contains 148,040 instructions and needs 13,112,711 clock cycles to finish when running on an ARM9 processor. The program segment contains several patterns having a similar shape. The X-axis of the figure shows the number of clock cycles, and the Y-axis shows the address of the instructions. A point means an instruction is executing in the corresponding

clock cycle. If a horizontal line contains more points, the corresponding instruction is executing more frequently. As can be seen from the figure, the lines in the red rectangle are the most dense (containing the most points), so the corresponding instructions are the most frequently executed. It can be seen from the enlarged figure that the number of most frequently executed instruction is only 14. The 14 instructions dominate about 40% of the overall execution time, and they have the greatest impact on the overall power-efficiency. Figure 6.2 gives the percentage of execution time the ARM processor spends on each instruction of the JPEG program. As can be seen from the figure, the instructions have highly imbalanced executing percentages. Most of the instructions are never executed or just executed once. It can be deduced from the analysis above that an embedded processor spends most of its time only in small instruction segments, which can be either a small loop or a function call.

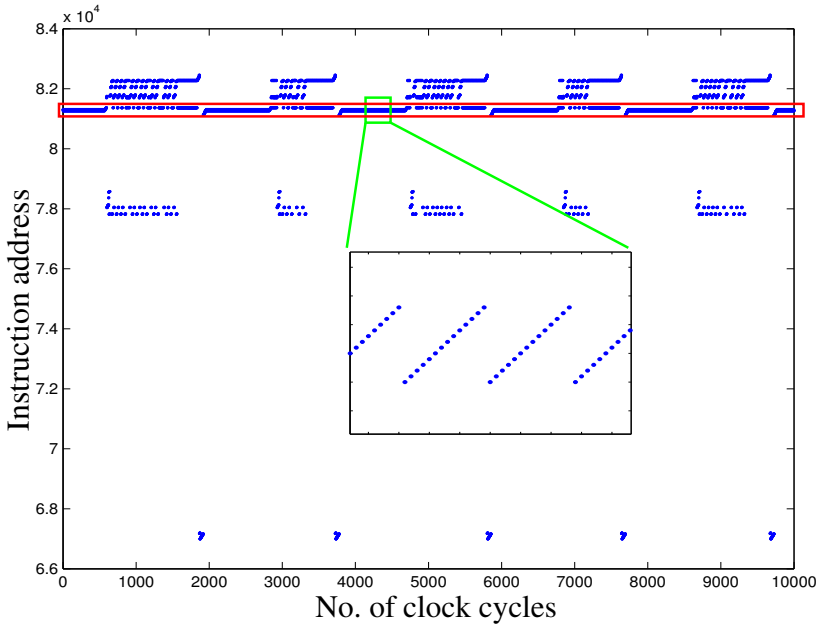


Figure 6.1: The running trace segment of a JPEG program

This phenomenon can be further revealed by the analysis of another media (audio/video) encoding/decoding program as shown in Figure 6.3. The media processing program contains 14,780 instructions and needs 9,105,576 clocks to complete when running in an ARM9 processor. The ARM processor spends 76% of its time in executing only 52 instructions.

Other research work also shows the execution time of an embedded processor is

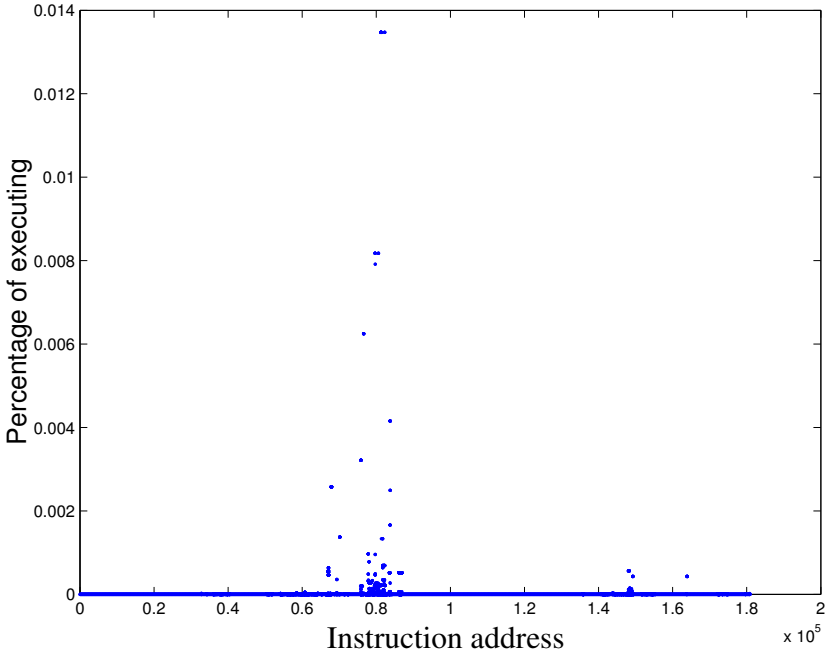


Figure 6.2: The distribution of execution time for instructions in a JPEG program

dominated by small number of instructions. As discussed in Chapter 2, the design of the 'loop buffer' began with the observation that dynamic execution traces of embedded programs are dominated by program loops containing a small number of instructions. Lee et al. observed that about 46% of all taken instructions belong to small loops with a backward jump distance of 32 instructions or fewer [23].

It can be concluded from the analysis above that embedded processing has three specific characteristics:

- An embedded processor can be less general-purpose than a PC processor. Although an embedded processor can be used in many applications, once it is integrated in an embedded system, it may execute only a few hand-optimized programs, so the complexity of embedded processing instruction sets can be simplified. However, some features that can significantly improve the performance and power-efficiency for the targeted kernels should be added. Moreover, an embedded processor needs to support fewer functions than a PC processor, for example, floating-point units are rarely used in embedded programs so they can be omitted.

- Since the CPU usage rates for different program segments and instructions are highly unbalanced, an embedded processor should be biased towards

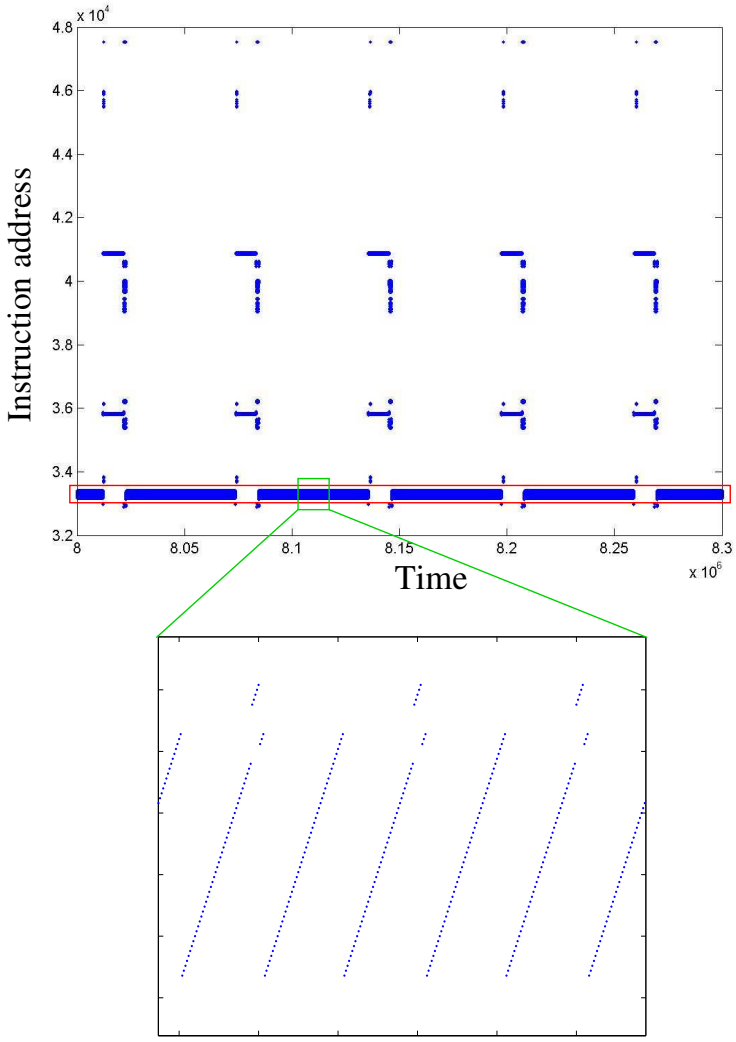


Figure 6.3: The running trace segment of a media processing program

the most commonly-executed segments and instructions.

- The principle of locality becomes more apparent in embedded processing applications, especially for instructions.

These characteristics can be employed to achieve power saving by designing an embedded processor which supports only the most commonly-used instructions and has a very small instruction memory holding only commonly-executed program segments. However, this approach is not practical for two reasons:

- Compiler support is still a very important design issue for an embedded processor. To assist embedded system programmers, an embedded processor

should provide an instruction set with abundant instructions. The requirement to support a wide range of embedded applications also calls for an abundant instruction set.

- A embedded program is more than just the set of the most commonly executed instruction segments. The instructions that are executed only once still need to be stored in memory, fetched, decoded and executed — they require the same treatment as the more commonly-executed instructions.

Therefore, there exists a contradiction between execution complexity and general-purpose functionality and this contradiction cannot be eliminated by a conventional unified processor architecture in which all tasks are executed in a single processor. The concept of 'hierarchical processing' is introduced to alleviate this contradiction.

The basic idea of hierarchical processing is to include several levels of processing units (processors) in one embedded system where the processors are not identical but have hierarchical functionalities. More complex processors support more general-purpose functionality but have more execution overheads, thus consuming more power. Simpler processors have a small memory and support only some commonly used instructions and addressing models, so they can be designed to be faster and more power-efficient by eliminating execution overhead. The infrequently used programs and instructions are executed in complex processors to allow the embedded system to support general-purpose functionality. The frequently executed program segments and instructions are executed in simple processors to minimize overheads.

Although a hierarchical processing scheme can minimize execution overheads while maintaining general-purpose functionality, it introduces another overhead, which is the communication between processors. Therefore, efficient and low-overhead communication is critical in the design of a hierarchical processing architecture. A multi-level hierarchical processing architecture with a lot of (instruction-level) communications and data transfers between processors is undesirable. In this research, the hierarchical processing architecture is based on a two-level CPU-coprocessor architecture, and the communications are not instruction-level but coarse-grained — a number of instructions are executed between two communications.

6.2 A hierarchical processing architecture

6.2.1 The overall architecture

Figure 6.4 shows the proposed hierarchical processing architecture employing two data processing units in the embedded processing system. One is a conventional general-purpose embedded processor acting as a main CPU, the other is a simple processing unit acting as a coprocessor. The main CPU provides abundant instructions, supporting various address modes, interrupts, memory faults and off-chip control. The coprocessor supports a simple instruction set, containing only the most frequently-used instructions but may include some dedicated hardware which increases the performance and power-efficiency of some specific embedded processing applications.

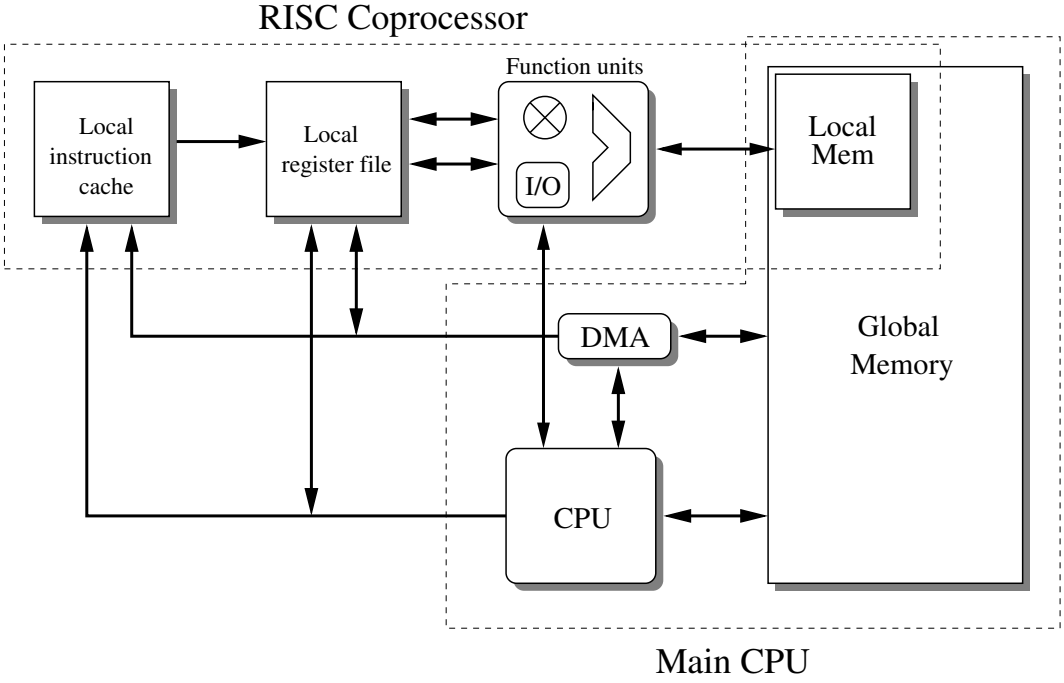


Figure 6.4: The proposed hierarchical processing architecture

Although the coprocessor is simple, it has a powerful processing ability when executing the instructions that it supports. Because the coprocessor no longer needs to support a full instruction set, it can avoid much of the execution overhead of a general-purpose processor, thus being power-efficient. The frequently-executed program segments, such as small loops, long equation evaluations and

function calls, will be executed in the coprocessor. The infrequently-executed program segments and those instructions that cannot be executed in the coprocessor will be handled by the main CPU.

To maintain functional simplicity and minimize power consumption, the coprocessor has only a very small instruction buffer and cannot fetch instructions from the main instruction memory when it finishes processing the current task. Instead, the main CPU will feed program segments into the small instruction buffer when it wants them to be executed in the coprocessor.

As discussed before, infinite and continuous data stream computing is very common in embedded processing applications. Normally, for data stream computing, two large memory arrays are processed and the results are put into another array. To support stream-based computing efficiently, the coprocessor contains a small local memory, which may be partitioned into several blocks. Local memory-memory calculation can be executed within the coprocessor without the assistance of the main CPU. However, the coprocessor cannot access the memory space outside the local memory. To the main CPU, the local memory, the instruction buffer and the register file of the coprocessor are just parts of its main memory.

The concept of coprocessor in this architecture is a bit different from that of conventional ones because, on the one hand, the coprocessor may dominate the actual execution time — it can execute a program segment for a long time without the assistance of the main CPU and it has a more powerful processing ability than conventional coprocessors. On the other hand, to support efficient communication, the coprocessor should be placed close to the main CPU. In this sense, the coprocessor is more like a powerful function unit of the main CPU.

6.2.2 Coupling the CPU and the coprocessor

Theoretically, the main CPU can process in parallel with the coprocessor, but the real-time synchronization and resource sharing introduce a lot of execution complexity and overheads. In the proposed architecture, the main CPU and the coprocessor execute serially.

As described above, the coprocessor does not have the ability to fetch instructions from the main memory by itself. There are two ways to initialize the instructions of the coprocessor. One is a static approach — all program segments to be executed in the coprocessor can be loaded into the local instruction buffer in advance before executing a program. The other is a dynamic approach — all

instructions are loaded in the main memory and the main CPU starts to execute a program. When the main CPU finds that a program segment should be executed in the coprocessor, it will send all the instructions of the program segment to the instruction buffer in the coprocessor at run-time.

Each of these approaches has advantages and disadvantages. For the static approach, if a lot of program segments need to be processed in the coprocessor, the coprocessor needs to have a large instruction buffer to hold these segments. A large instruction buffer means low speed and high power. The dynamic approach requires a smaller instruction buffer because, at one time, only one or a few program segments are held in the instruction buffer, thus saving power. However, if the program segments keep on changing, the overheads due to switching contexts become significant; fortunately, based on the principle of locality, these situations are rare. The proposed architecture supports both approaches, but the dynamic instruction feeding scheme is used more frequently.

Figure 6.5 illustrates the execution of a program in a hierarchical processing architecture. The program contains some segments to be executed in the main CPU as well as some to be executed in the coprocessor. Because of the advantages of asynchronous logic design mentioned in Chapter 3, the two processing units are proposed to be asynchronous.

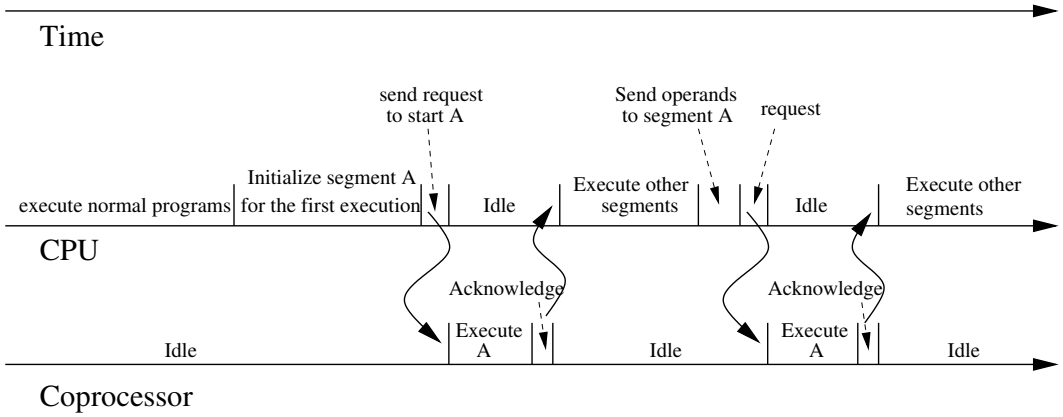


Figure 6.5: The cooperation between the CPU and the coprocessor

Before execution, the PC of the main CPU is set to the start address of the program. The program is then executed in a normal way in the main CPU. After some time, the main CPU encounters an instruction telling it to initialize the instruction buffer of the coprocessor. Actually, the instruction is no different from other memory data transfer instructions. The main CPU then enters a loop to

send coprocessor instructions to the local instruction buffer and initial operands of the local memory and the local register file. It is very inefficient if every data transfer is handled by the CPU. A more efficient way is to use a Direct Memory Access (DMA) module to handle the lowest-level memory block movement, initialization can thus be done by using a few DMA control instructions.

After initialization, the main CPU sends a *Request* signal to wake up the coprocessor and enters an idle state waiting for the results from the coprocessor. The coprocessor then starts executing the program segment. When the coprocessor finishes the program segment, it sends an *Acknowledge* signal to wake up the CPU and then puts itself into an idle state.

The main CPU continues executing the program until it meets another program segment to be executed in the coprocessor. If the program segment is one previously initialized (the local instruction buffer of the coprocessor may contain more than one program segments), the CPU does not need to initialize the program segment again; it needs only to send the new operands to the register file or the local memory. Otherwise, it will send the new instructions to the local instruction buffer of the coprocessor.

The approaches for data transfers between the two processing units are very flexible. Data transfers can be done either by an I/O handshake module in the function block or by sharing memory and can be either fine- or coarse-grained. An example of fine-grained data transfer is an encryption algorithm in which one data item is processed through a complex mathematic equation where one output is generated. Examples of coarse-grained data transfer are stream-based processing applications, which have a lot of input data and output results.

As can be seen from Figure 6.6, the tasks to be processed in the coprocessor are coarse-grained, which means the coprocessor needs to execute tasks for many cycles before communicating with the main CPU. If the coprocessor needs to contact the main CPU every time it finishes one instruction, the communication overheads will overwhelm the power saving by hierarchical processing, but for coarse-grained communication, the overheads are assumed to be very low (just like putting an instruction into cache in a conventional memory hierarchy architecture).

6.3 RISC coprocessor design

In the two-level hierarchical processing architecture presented in the last section, the main CPU is just an embedded general-purpose processor. Low-power techniques for general-purpose embedded microprocessors are too broad to be addressed in this thesis. This section discusses the design of a simple RISC-like coprocessor.

6.3.1 Instruction set design

The design of an instruction set needs very careful evaluation. Moreover, for embedded processing applications, a successful instruction set design also depends on the statistics of the behaviours of the targeted program kernels that are executed frequently. The motivation of the coprocessor design is to minimize execution overheads and to increase power-efficiency. So 'simple is good' is the philosophy used in the coprocessor design.

The prototype instruction set of the coprocessor includes only three kinds of instructions: data processing, branch and load/store instructions.

A typical data processing instruction has a 'three-address' format with two source addresses for operands and one destination address for results. These instructions normally do not change the condition code bits (to be introduced later). Comparison instructions are also included in the processing instructions, these differ from normally data processing instructions in that they do not produce a result but set condition code bits. A comparison instruction is normally put before a branch instruction. The coprocessor supports only a very simple immediate instruction which sends a 10-bit immediate number to a given register.

A branch instruction changes the program counter (PC) depending on the current condition code bits. The prototype coprocessor has only two condition code bits — Negative (N) and Zero (Z), ignoring the carry and overflow that are included in most general-purpose processors, thus the prototype coprocessor normally is not used to handle arithmetic calculations with a carry input.

The load instruction loads a data item from the local memory to the local register file. The store instruction, on the other hand, stores a data value from the local register file to the local memory. Stream-based computations normally load and store data sequentially from a start address. Autoincrement instructions improve the code-density if an embedded processing application has a lot of these

kinds of computations. The prototype coprocessor supports only postincrement load and store instructions.

All the instructions have an instruction-length of 20 bits. The first 5 bits represent what kind operation the processor should execute. Since the total number of instructions is larger than 16, the instruction opcodes need 5 bits. The unused opcodes can be left for specific data processing applications. Instructions for division, Hamming distance, Counting Leading Zeros (CLZ) and S-box for DES algorithm are possible candidates [72]. The last 15 bits of an instruction are divided into 3 segments with different definitions for different instructions. The structure of the instruction set is shown in Figure 6.6 and the prototype instruction set of the coprocessor is shown in Table 6.1.

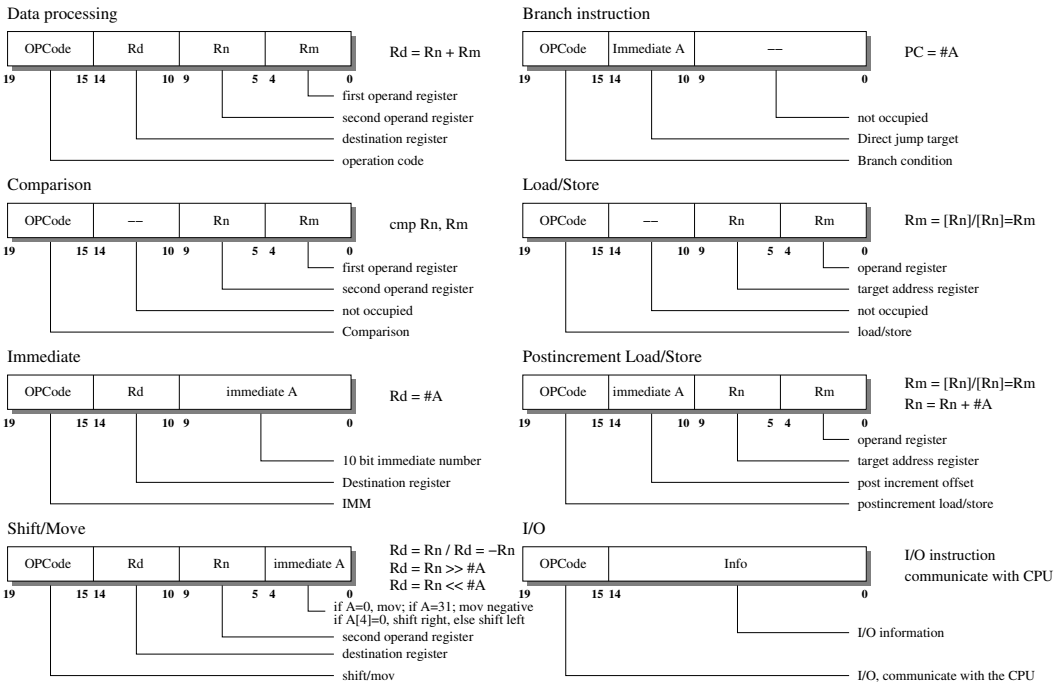


Figure 6.6: The structure of the instruction set

As can be seen from Figure 6.6 and Table 6.1, the coprocessor has a very simple and regular instruction set. ARM processors are known to have a relatively simple and easily decoded instruction set. Even so, the instruction decoder of the ARM7TDMI processor has 12 inputs, 91 minterms and 42 outputs and it uses 15% of the overall core power [73]. In the coprocessor here, the power consumption due to instruction decoding is much smaller than that of the ARM (in Table 6.3, it is included in 'Others', about 20% of 'Others' and 5% of the overall coprocessor

Table 6.1: The coprocessor instruction set

No.	Mnemonic	Meaning	No.	Mnemonic	Meaning
0	AND	logic AND	16	BAL	branch always
1	OR	logic OR	17	BEQ	branch if equal
2	XOR	logic XOR	18	BNE	branch if not equal
3	ADD	addition	19	BGT	greater than
4	SUB	subtract	20	BLT	less than
5	CMP	comparison	21	BGE	greater than or equal
6	SM	shift/move	22	BLE	less than or equal
7	MUL	multiplication	23		
8	IO	CPU communication	24		
9	IMM	move immediate	25		
10	LDR	load	26		
11	PLDR	postincrement load	27		
12	STR	store	28		
13	PSTR	postincrement store	29		
14			30		
15			31		

power consumption).

6.3.2 The cost of pipelining

Pipelining is the simplest form of concurrency to improve the performance of a processor. The simple instruction set of a RISC processor greatly simplifies its pipeline design and makes it possible to have a high clock rate with single-cycle execution. Even so, a RISC processor still spends a significant proportion of its overall core power consumption in pipelining. The power overhead due to pipelining is mainly caused by the efforts to reduce 'pipeline hazards'.

Pipeline hazards are situations that prevent the next instruction in an instruction stream from executing at a designated clock cycle. Hazards have a significant impact on the performance obtainable from pipelining. There are three kinds of hazards:

- Structural hazards:

Structural hazards arise from hardware conflicts when two or more instructions attempt to access hardware which allow only mutually-exclusive access.

- Data hazards:

  Data hazards arise when an instruction needs the result of a previous instruction that is still being processed in the pipeline (data dependence).

- Control hazards:

  Control hazards arise when a branch instruction has been fetched but the previous instruction that sets the condition code is still being processed.

A number of techniques have been proposed to avoid or reduce pipeline hazards. These techniques belong to three categories:

- To minimize structural hazards, more hardware must be added to reduce the chance of hardware conflicts. The most commonly accessed part of a processor is its register file. To avoid structural hazards due to register file access, register files are normally built to have several input and output ports. Because of this multi-port architecture, register files are more power hungry than single-port SRAMs.

- The basic way to minimize data hazards is by using a 'forwarding' or 'bypassing' technique. Data hazard stalls normally arise when one instruction tries to read operands from a register file or memory, but the previous instruction has not put the result back to the register file or memory because of the pipeline architecture. Of course, the instruction can be blocked until the previous instruction sends the result back. If the previous instruction can forward its result to the instruction that needs the result, the pipeline operation will not stall. However, a forwarding technique introduces extra execution overheads in control and forward path.

- The simplest way to handle branch instructions is to stall instruction prefetch until the destination address is known. This scheme is simple but loses performance. A 'branch prediction' scheme is normally used to minimize pipeline branch penalties. The 'branch prediction' scheme holds a record of the former program trace and uses this to predict the branch target of a

branch instruction. The execution overheads of a branch prediction scheme come from two factors:

1. Keeping the program trace record and implementing a prediction algorithm;
2. If the previous prediction is wrong, the processor needs to discard the mistakenly-fetched instructions, introducing extra complexity in control. This also wastes processing power.

Let us analyze the impact on performance if we discard these techniques. 4 simulators were built to simulate the performance of 4 processors listed as follows:

- P1: a single-port register file, without bypassing, stall branch instructions.
- P2: a three-port register file (two for read, one for write), without bypassing, stall branch instructions.
- P3: a three-port register file, bypassing, stall branch instructions.
- P4: a three-port register file, bypassing, branch prediction.

Consider the hazards of the following program segment. There is a data dependence between Instruction1 and Instruction2. A control hazard arises between Instruction3 and Instruction4. The architecture of the RISC coprocessor is shown in Figure 6.9 (more detailed discussion on the architecture is presented in the next section). The pipeline operations when the program segment is executed in the four processors are illustrated in Figure 6.7. As can be seen from the figure P4 is faster than P1.

```
/* A C program for sum = 0+1+2+3+...+100                    */
            for(i=0; i<101; i++)
               sum = sum + i

/*           Assemble codes                                  */
/*           i->r0, sum->r1                                  */
    loop:
       ADD r0, r0, #1;    -- Instruction1:
       ADD r1, r1, r0;    -- Instruction2: data dependence
       CMP r0, #101;      -- Instruction3:
       BNZ loop;          -- Instruction4: control hazard
```

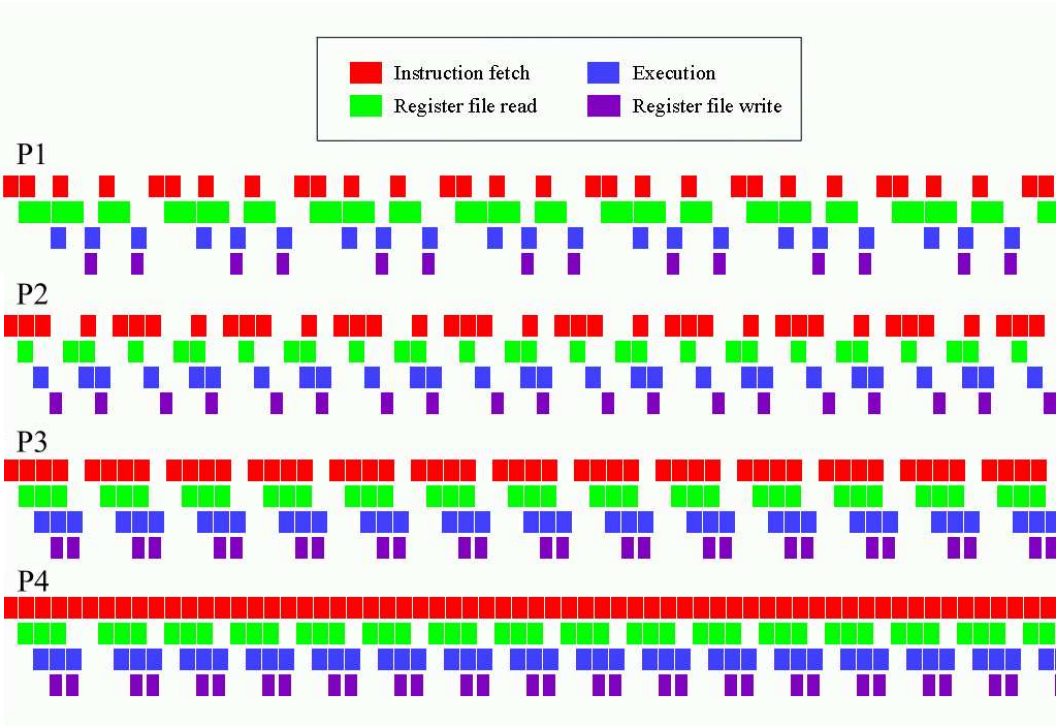


Figure 6.7: The pipeline operations of the four processors

From the experimental results, replacing a single-port register file with a 3-port register file increases the performance by 35%. Using a bypassing scheme increases the performance by another 37%. Employing a branch prediction strategy (assuming a 100% correct rate) further increases the performance by 16%. Therefore, if the three pipeline hazard reduction techniques are all be used, the speed of a processor can be improved by $1.35 \times 1.37 \times 1.16 = 2.1$ times. These performance impacts differ for different programs and architectures. The analysis is made using a RISC architecture with only 4 pipeline stages and a program segment having very dense data dependencies. It is estimated that for real programs, the impact on performance by avoiding these techniques will be smaller than 50%.

Pipeline hazard reduction techniques increase the performance of a pipelined RISC processor, but introduce extra power overheads. The following analysis is based on the power consumption of the Amulet3 processor where the total power consumption of the register file, branch prediction circuit and bypassing circuit is about 32% of the whole core power consumption (7% for branch prediction, 13% for the register file and 12% for bypassing) [17]. These data are based on

the power breakdown of a general-purpose processor. For the coprocessor, if the overall power consumption is reduced and instruction fetching and decoding are no longer the most power hungry parts, the power consumption of pipeline hazard reduction techniques is estimated to be greater than 32% of the overall processor power consumption.

If these pipeline hazard reduction techniques are skipped and a single-port register file is used, overall power consumption can be significantly reduced: through experiments, using a single-port register file reduces the power consumption of a register file by 1/3; changing a bypassing scheme with a register locking scheme reduces the power consumption of a bypassing technique by 70% (this number is based on testing an asynchronous processor, since there is no global clock timing, an asynchronous processor needs to use a more complex bypassing scheme than that of a synchronous processor). Avoidance of branch prediction can save almost all the power used by branch prediction. The total power saving by avoiding pipeline hazard reduction techniques is about $0.07+0.13\times0.33+0.12\times0.7 = 20\%$ of the overall core power consumption of Amulet3. For the coprocessor, the power saving percentage is estimated to be about 33%.

To sum up, pipeline hazards impact on the performance of a pipelined processor. The performance penalty can be minimized by using pipeline hazard reduction techniques and more hardware. However, these techniques and hardware additions introduce extra power overheads. For the coprocessor, avoidance of these techniques and hardware can achieve a power saving of about 33% of the overall power consumption at the cost of less than 50% performance loss. The total power consumption can be further reduced by totally abandoning pipelining [74], but the power saving is only 5% (mainly due to the drivers of the pipeline registers) based on simulation and the performance penalty is about another 30%. Therefore, the proposed coprocessor still uses a pipeline architecture but omits pipeline hazard reduction techniques (the P1 architecture).

6.3.3 The proposed RISC coprocessor architecture

Figure 6.8 shows a classic 5-stage RISC pipeline architecture. With this architecture, the execution of an instruction in the processor normally includes five steps:

- The processor fetches the instruction from the instruction cache.

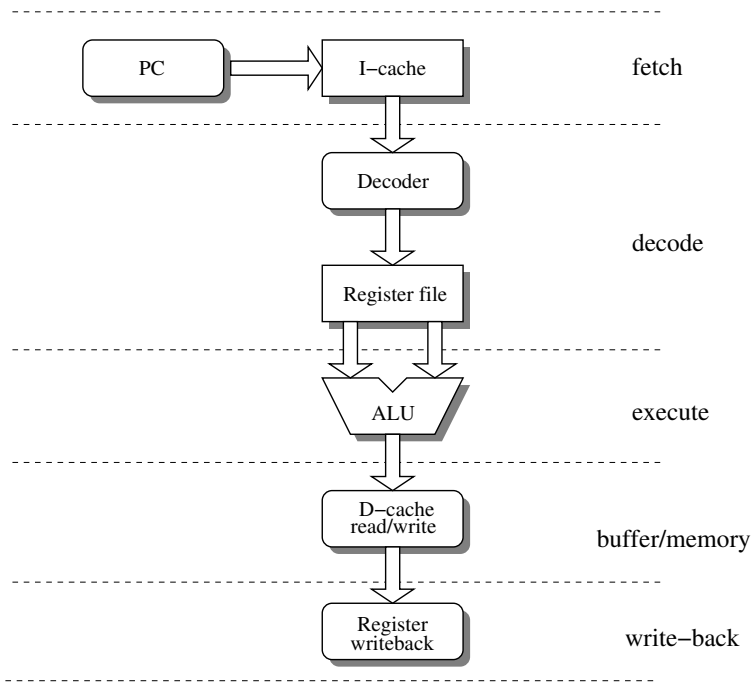


Figure 6.8: A conventional 5-stage RISC pipeline architecture

- The instruction is decoded and the operands are read from the register file.
- The operands are processed in the ALU and the result is generated.
- The result is stored to the data memory if necessary. Otherwise, the result is just buffered for one clock cycle to balance the number of pipeline stages.
- The result generated by the instruction is written back to the register file and any required data is loaded from the data memory.

To improve throughput, these five steps are pipelined and executed in parallel. In ARM processors, the respective pipeline stages are called instruction fetch (IF), instruction decode (ID), execute (EXE), buffer/memory (MEM) and write-back (WB).

Figure 6.9 (a) and (b) show the proposed RISC architecture of the coprocessor and its pipeline stages. Since the instructions of the coprocessor are very simple, there is no need to include a dedicated instruction decoding stage. Therefore, the pipeline architecture includes only 4 pipeline stages: instruction fetch (IF), register file reading (REG), processing/memory (EXE) and result writing back (WB). Since the load and store instructions in the coprocessor do not need to calculate an operand memory address, the memory fetch/store circuits are put

in parallel with the function units which process data. Both the load and store instructions can complete in one cycle. The postincrement load and store instructions can also complete in one cycle by allowing a memory load/store operation to execute in parallel with an address calculation. The postincrement load may cause a slight longer latency than other instructions because two results need to be written back to the register file (one data item from the memory and the post incremented address), but the latency can be tolerated by an asynchronous pipeline.

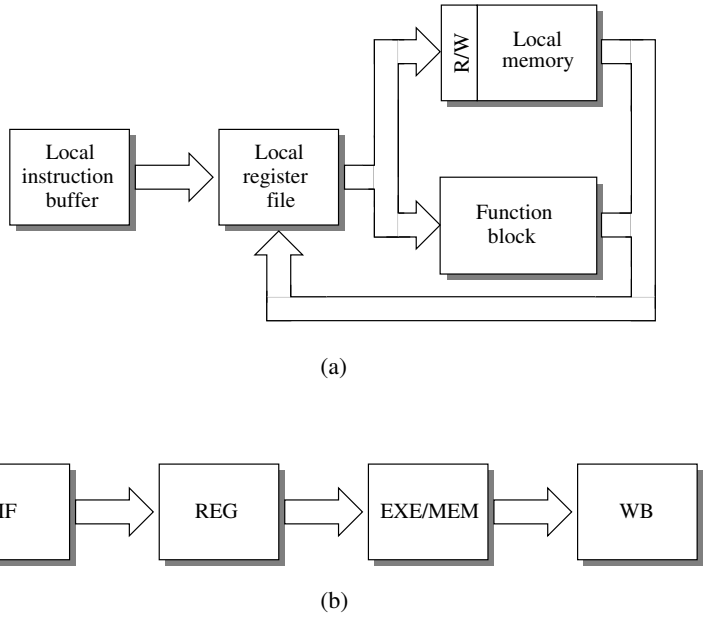


Figure 6.9: The organization and pipeline architecture of the coprocessor

Unlike the classic 5-stage pipeline architecture, which puts the memory load and store operations after the ALU, the coprocessor puts it in parallel with the function units. Since the coprocessor does not allow load/store instructions to have address calculations, putting the memory load/store circuit in an early pipeline stage helps to reduce the power consumption used by propagating opcodes and addresses through more stages in the datapath.

The function block contains the most commonly used function units, including a shifter, an arithmetic/logic unit and a multiplier, to support the instruction set. Unlike the ARM processors, in which operands are first shifted then processed, the coprocessor puts the shifter and other data processing units in parallel. An instruction cannot execute a shift operation and a data processing operation together; this approach reduces the complexity of the datapath.

As mentioned before, embedded processors usually contain some special instructions to support efficient execution of important kernels. Some special embedded operations, for example the S-box of DES, cannot be executed efficiently using conventional instructions. Therefore, the function block may include some special function units to support these operations.

Word length is another factor in the coprocessor design. A short word length imposes constraints on embedded applications, such as address space size and data precision, a long word length may cause a significant waste of datapath bits. Table 6.2 [3] shows the distribution of data size for the TMS320C540xDSP, about 90% of operands are 16 bits. Figure 4.17 revealed the same distribution characteristic, where more than 90% of the operands have a SBC below 16 bits. An 8-bit datapath offers another choice, but its small address space may make some applications difficult to be implemented. A 16-bit word length is chosen for the coprocessor.

Table 6.2: Data size distribution [3]

Data size	Memory operand in operation	Memory operation in data transfer
16 bits	89.3%	89.0%
32 bits	10.7%	11.0%

6.3.4 Primary experimental results

To demonstrate the power efficiency of the hierarchical processing architecture, a prototype RISC coprocessor was designed and simulated in a 0.18 μm CMOS technology. The coprocessor has a 16-bit datapath and a local instruction buffer, which stores up to 32 instructions. The register file is a single-port SRAM containing 32 16-bit words. Using a supply voltage of 1.8 volts and at a temperature of 25^{o}C, the energy and delay characteristics of each component are shown in Table 6.3; the delays of pipeline controllers and registers are included in the delays of other components.

Based on the data shown in Table 6.3, if the execution of an instruction contains one instruction fetch, two register file reads, one register write and one multiplication, the instruction uses $3.3 + 3.1 \times 2 + 3.1 + 7.51 + 34.2 = 54.31$ pJ energy, which equals 18,400 MIPS/watt. The energy efficiency is based on

Table 6.3: The characteristics of the proposed RISC coprocessor components

Component	Energy per operation (pJ)	Delay (ns)
Instruction buffer (read)	3.3	1.5
Register file (read)	3.1	1.5
Register file (write)	3.1	1.2
16x16-bit Multiplier	34.2	2.61
16-bit Adder/Logic Units	3.17	0.90
Others (including decoder, pipeline latches and controllers drivers, incrementer, PC, function block multiplexers.	7.51	–

schematic simulation, which does not fully include wire capacitances. The post-layout energy efficiency is estimated to be reduced by 2 to 9,200 MIPS/watt. Compared to the energy efficiency of conventional embedded RISC microprocessors (normally around 1,000 MIPS/watt using the same technology and supply voltage), the power efficiency is very high even when power-hungry multiplication instructions are considered. Consequently, the coprocessor can at least improve the power efficiency of a conventional general-purpose embedded processor by a factor of 5 (2x difference between 16-bit word length and 32-bit word length). If 50% of the instructions of a program can be executed in the coprocessor, 40% of the overall power can be saved if the hierarchical processing scheme is used.

The primary experimental result demonstrates the hierarchical processing architecture proposed in this chapter is power efficient when executing those embedded applications whose computations are contained in several small instruction segments.

6.4 Summary

In this chapter, a hierarchical processing architecture is proposed based on the analysis of embedded data processing characteristics. When running a typical embedded data processing program, a processor spends a large proportion of its time executing a few program kernels containing only a very small number of instructions. A hierarchical processing scheme copes with these frequently executed instructions differently from other instructions. The most frequently used instructions are executed efficiently in a low-level processing unit which supports a

small number of instructions, has a simple datapath and takes instructions from a small instruction buffer. The high-level processing unit is a general-purpose embedded processor which has a conventional instruction set and functionality. The instructions that cannot be executed (efficiently) in the coprocessor will be executed in the high-level CPU, the low-level processing unit therefore improves power efficiency when executing the most frequently used instructions, while the high-level processing unit preserves general functionality.

To demonstrate the power-efficiency of the hierarchical processing architecture, a prototype RISC coprocessor was designed and simulated. The coprocessor is demonstrated to be at least 5 times more power-efficient than a general-purpose embedded processor. The cost of RISC pipelining techniques in power consumption is also studied in this chapter.

The next chapter will present the design of another coprocessor — one using an asynchronous dataflow scheme.

Chapter 7

The design of a dataflow coprocessor

Since Patterson and Ditzel first clearly brought forward the concept of the Reduced Instruction Set Computer (RISC) in 1980 [75], considerable research has been carried out in RISC microprocessors, including a lot of low-power techniques. Since a mature architecture may have little potential for further power saving, this chapter makes an attempt to explore the power-efficiency of an older but less-studied (and maybe more interesting) approach — a dataflow architecture [76] [77]. A dataflow coprocessor is designed and studied in this chapter.

The chapter is organized as follows: Section 7.1 gives a brief introduction to dataflow machines; Section 7.2 proposes a dataflow model used in the coprocessor; Section 7.3 presents the dataflow architecture; Section 7.4 describes the circuit implementation of the coprocessor; Section 7.5 proposes an automatic mapping algorithm translating conventional RISC programs to dataflow ones; Section 7.6 presents the experimental results, and also compares the power and performance of the RISC coprocessor with the dataflow coprocessor; Section 7.7 concludes the chapter.

7.1 Introduction to dataflow machines

Computing machines can be referred to as 'control flow machines' or 'dataflow machines' depending on the way in which instruction orders are defined:

- The commonly used von Neumann machines belong to 'control flow' machine, in which programs are executed in an order that is specified by programmers. The execution order of a program in a von Neumann machine is sequential, in which instructions are fetched from consecutive locations, unless this sequentiality is changed by some dedicated control instructions (branches).

- In contrast to control flow machines, a 'dataflow' machine executes programs in an order implied by data interdependencies and space availability.

Dataflow execution offers natural properties for fine-grained instruction-level parallel processing. The parallelism of a dataflow program is self-scheduling. Except for data dependencies, there is no constrained sequentiality, so a dataflow program allows all forms of instruction parallelism. Synchronization of different parallel threads is implicit in the form of data interdependencies. With a dataflow scheme, parallelism and synchronization eliminate the need for programmers to use explicit control instructions which manage parallel execution. These properties have attracted many researchers aiming for high performance. In this chapter, however, dataflow computing is explored specifically to improve the power-efficiency of conventional microprocessors. As will be seen later, a dataflow coprocessor can achieve higher performance than the RISC coprocessor introduced in the last chapter.

7.1.1 Dataflow graphs

In a dataflow machine, programs are normally described in the form of dataflow graphs. The evaluation of a complex computation can be regarded as a sequence of data flowing through some data processing units (or function units), such as multipliers and adders, and the evaluation can be described by a graph. Figure 7.1 shows a dataflow graph that illustrates the equation $x = a+(b\times c)+(d\times e)$. As can be seen from the figure, if a, b, c, d and e are set by initialization, the output x will be calculated after some time depending on the speed of the multipliers and adders used. The sequence of the operations is not important. As long as the function units have valid inputs, they can process the operations. However, if a stream of data needs to be processed, function units must synchronize with others to prevent overwriting the data values (data tokens) which are being calculated. Thus a dataflow operation should satisfy two requirements:

- Each input port of a function unit should have a valid data token.
- Each output port of a function unit (destination) should be empty, which means the function unit has finished its former calculation (if any) and is ready to process new tokens.

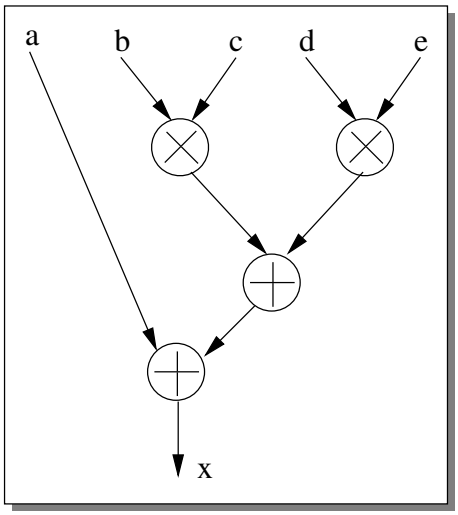


Figure 7.1: A dataflow graph for a long equation

From the example mentioned above, a dataflow graph is constructed from some basic dataflow 'nodes', each representing a basic data processing function. The input arcs of a node represent the inputs for the respective data processing function and the output arcs represent the outputs from the respective data processing function. Normally, a node has two input arcs and one output arc, but the number of input and output arcs can be changed for different purposes. A number of basic nodes are connected together to implement different programs and data flowing through a dataflow graph are referred to as 'tokens'. If two nodes are linked together, the node that issues data tokens is called a 'sender' and that accepting data tokens is called a 'receiver'. Two kinds of dataflow computations are defined depending on who initializes dataflow operations:

- Data-driven computation — operations are executed in an order determined by the availability of input data.
- Demand-driven computation — operations are executed in an order determined by the requirements for data.

In data-driven computation, senders start dataflow operations, similar to a 'push' channel in asynchronous logic. In data-demand computation, receivers start dataflow operations, similar to a 'pull' channel in asynchronous logic.

Figure 7.1 includes only basic data processing nodes with two input arcs and one output arc. The basic data processing node normally has two input arcs and one output arc, as shown in Figure 7.2(a). However, an output is often sent to several different locations and hence a 'fork' node is also needed, as shown in Figure 7.2(b). A data processing and a fork node can form a unified node which has two input arcs and two output arcs as shown in Figure 7.2(c). Since one node corresponds to an instruction and the duplication of an output is very common, the combination of a data processing node and a fork node can greatly reduce the number of instructions required for creating temporary variables.

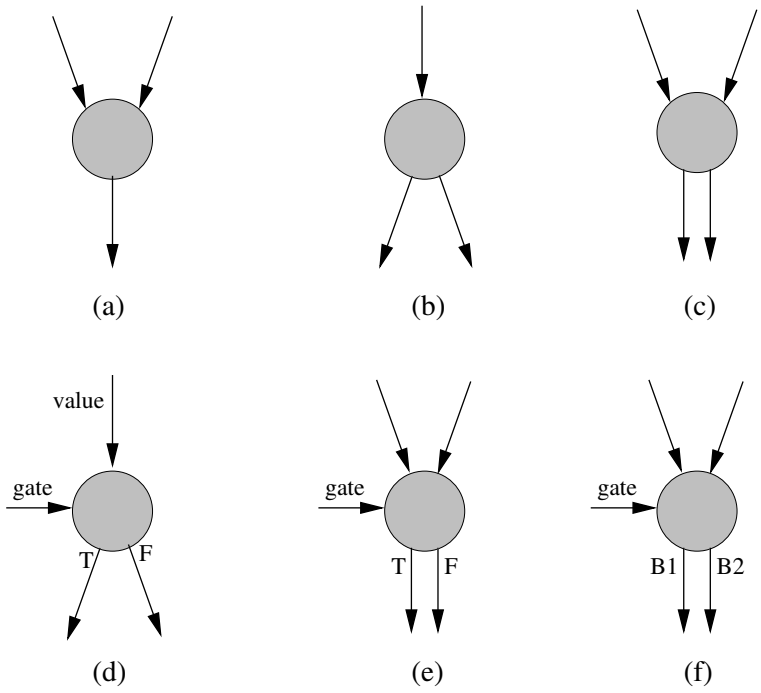


Figure 7.2: Basic dataflow nodes

Conditional and repetition executions are indispensable for basic program segments such as loops and branches and these operations call for a condition node called a 'branch' as shown in Figure 7.2(d) which contains two input arcs and two output arcs. The input tokens from the 'value' arc are normal data tokens and those of the 'gate' arc are Boolean tokens which control the output direction of data tokens. If the value of the gate token is 1, the data token is sent

to the 'true' output arc; otherwise, it is sent to the 'false' output arc. The branch node can also be combined with a data processing node as shown in Figure 7.2(e).

If the two output arcs of Figure 7.2(e) are not fixed but are configurable, a unified node can be constructed as shown in Figure 7.2(f). This node represents a unified format which combines the functions of data processing, duplication and conditional control. The unified node can be configured to behave like the other nodes shown in Figure 7.2. In the unified node, $B1$ and $B2$ are Boolean values that can be set as 'true' or 'false'. A result will be sent to the output arc(s) whose Boolean value is equal to the gate value. If $B1 \neq B2$, the node is a branch. If $B1 = B2 = gate\ value$, the node is a fork. If $B1 = B2 \neq gate\ value$, the node is a 'sink', which only consumes tokens but never generates tokens. As will be described in the remainder of this chapter, the unified dataflow node format results in a very regular dataflow circuit implementation.

The three types of basic nodes — data processing nodes, the branch node and the fork node — can be used together to describe any deterministic program. Figure 7.3 shows a dataflow graph of

$$sum = \sum_{i=0}^{n} i$$

.

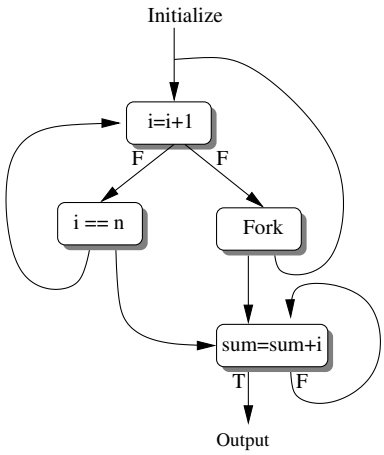


Figure 7.3: A dataflow graph for $sum = 1 + 2 + ... + n$

7.2 The proposed dataflow model

Clearly, it is inefficient for a general-purpose data processing unit to have a specific datapath for every possible computation such as that shown in Figure 7.3; a more flexible architecture is needed which can support many different computations. Figure 7.4(a) shows a basic static dataflow architecture [78] and Figure 7.4(b) illustrates the mapping of the equation $x = a + (b \times c) + (d \times e)$ onto this architecture. There are three main components in this architecture — a matching store, a function block and an interconnect network. The matching store is used to store instructions and data. The function block contains several function units where data are processed and an Input/Output module to access main memory or communicate with a CPU. The interconnect network sends data tokens to the function units and results back to the specified positions in the matching store.

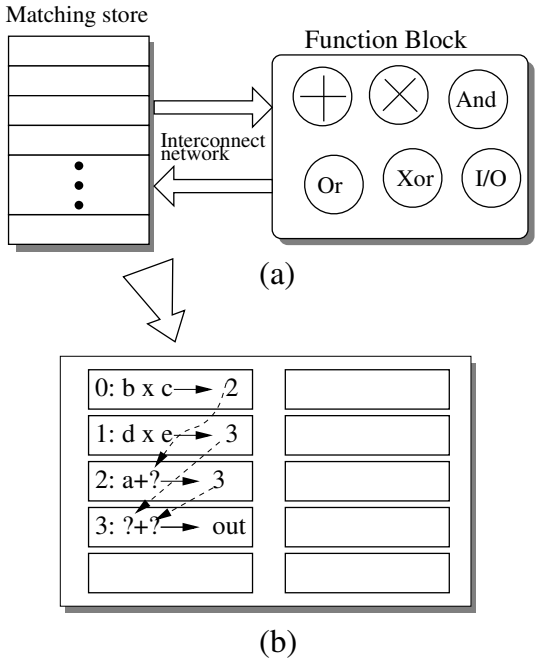


Figure 7.4: A static dataflow architecture

One instruction comprises four segments — an operation code, two operands and the destination addresses of the result. If an instruction satisfies the two requirements introduced above (each input arc has a valid token and each output arc is empty), the instruction becomes 'active'. However, not every active instruction can be processed immediately. The number of active instructions that need the same processing function, say multiplication, may exceed the number of

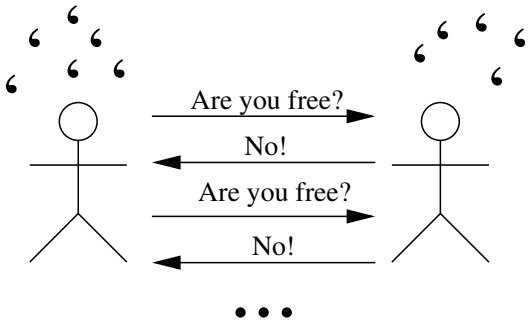
'free' function units. Thus some instructions must wait for the execution of other instructions before they can get a free function unit. The interconnect network takes charge of selecting the 'active' instruction; this is called 'arbitration'. If an instruction is selected by the interconnect network, it is said to be 'fired'.

Examining the dataflow operation in Figure 7.4(b), the question marks indicate empty places for input tokens. At the beginning, both instruction0 and instruction1 are active because they satisfy the two requirements, while instruction2 and instruction3 wait for the inputs indicated by question marks. If the interconnect network selects instruction0 to execute, the output of $b \times c$ will be sent to the empty place in instruction2 and change instruction2 to active. Then both instruction1 and instruction2 will be sent to the function block and the outputs will be sent back to the two empty places in instruction3. Finally, instruction3 becomes active and will be sent to the function block which generates the result x. The sequence of instructions may be 0123, 1023 or 0213; this does not matter and is not detected outside the matching store. The outside environment only issues the operands and receives the result. The mapping of the equation $x = a + (b \times c) + (d \times e)$ in Figure 7.4(b) differs from that in Figure 7.1; this illustrates that the mapping of a computation is not unique.

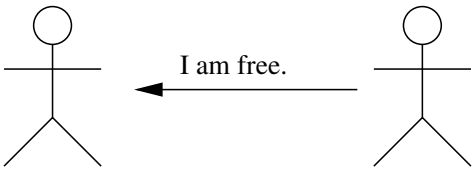
When a stream of data needs to be processed, two important details covered in the dataflow operations mentioned above have to be addressed— how to define the validity of a data token and an empty place and how to detect the state of the destination. Defining the validity of a data token and an empty place is very simple. If the interconnect network sends a result to an instruction, it also sets a valid bit corresponding to the operand to 1, indicating a valid data token. When the instruction is fired, it sets the valid bit back to 0, indicating an empty place.

Detecting the states of destinations is a problem. A straightforward solution is that the instruction requiring to send a result actively polls its destinations for their states. However, this is very inefficient because the instruction may have to repeatedly ask 'are you empty?' If the destinations are always occupied and cannot reply 'yes', the instruction will ask forever. This is a power-hungry dynamic action. A more efficient approach is proposed called a 'negative solution'. With a 'negative solution', receivers automatically send 'empty tokens' or 'data demands' to their senders when they are fired. Senders become active depending on the data demands from their receivers. Thus, in the 'negative' solution, two kinds of tokens flow inside the processor — data tokens and empty tokens. As

can be seen from the above, operations in the negative scheme are executed in an order determined not only by the availability of input data but also by the requirements for data. Figure 7.5 illustrates the different situations for these two synchronization schemes, the negative solution is more efficient because only one communication is needed to indicate an empty space.



(a) the positive scheme



(b) the negative scheme

Figure 7.5: The comparison of two different synchronization schemes

To support the 'negative' scheme and the unified format as shown in Figure 7.2 (f), the structure of the coprocessor instructions is as shown in Table 7.1. The number in the bracket after each field name indicates how many bits are needed in the respective field. The numbers are based on a 16-bit operand length and a matching store with 32 instructions.

The function of each field is interpreted as follows:

Table 7.1: The structure of an instruction

OPCode (7)	Const1 (1)	V1 (1)	OP1 (16)
DorG (0)	Const2 (1)	V2 (1)	OP2 (16)
—	ConstG (1)	VG (1)	GV (1)
Source1 (5)	SF1 (0)	Dest1 (6)	DV1 (1)
Source2 (5)	SF2 (1)	Dest2 (6)	DV2 (1)
SourceG (5)	SFG (1)	ETReq (1)	ETGot (2)

- OPCode defines what kind of operation should be processed. It also defines the types of operands in an instruction — immediate data or memory addresses for the operands. This will be discussed in Section 7.3. The operations of the dataflow coprocessor are as shown in Table 6.1, but move immediate and branch instructions are no longer needed.

- DorG defines the output as a data value or a Boolean gate value. 1 indicates a data value and 0 a Boolean value. This bit does not physically exist in the coprocessor and is implicit in the opcodes.

- OP1, OP2 and GV are three input values — operand1, operand2 and gate-value.

- Const1, Const2 and ConstG are three flags (const flags) defining whether or not the respective input values are constants. If an input has a 'const' bit of 1, this input will always be valid and never be set to empty.

- V1, V2 and VG are three flags (operand valid bits) determining whether the respective input tokens are empty or not. Their const bits are Const1, Const2 and ConstG respectively.

- Dest1 and Dest2 are two destination addresses for 'fork' or 'branch'. For a register bank with 32 registers where each register has two data value positions, these addresses need 6 bits each.

- DV1 and DV2 are two flags attached to the output arcs. The output is only sent to the address(s) having DV equal to the gate value. If $DV1 = DV2 = GV$, the instruction is a fork. If $DV1 \neq DV2$, the instruction has only one output. If $DV1 = DV2$ and $DV1 \neq GV$, the instruction is a 'sink', which only absorbs input data and does not generate output. 'Sink' instructions are not very common in sequential execution, but they happen quite often at the end of a loop.

- Source1, Source2 and SourceG illustrate the respective addresses of sources where the empty tokens should be sent.

- SF1, SF2 and SFG are three flags (source flags) determining whether the respective source addresses are valid or not. For example, if $SF1 = 0, SF2 = 1, SFG = 0$, the instruction only needs to send an empty token to the

address indicated by Source2. In a real program, an instruction should at least accept an input token (which means at least one empty token should be sent), so SF1 is always set to 1 and thus can be eliminated. Actually, the source flags are equal to the inversion of the const flags. These bits are duplicated to implement a regular architecture to be introduced in Section 7.3.

- ETReq indicates how many empty tokens an instruction needs to become active. It equals the number of its output arc(s). 0 means an instruction needs one empty token to become active; 1 means an instruction needs to two empty tokens to become active.

- The two-bit number ETGot indicates how many empty tokens one instruction has. If an instruction satisfies $ETGot = ETReq + 1$, it has received enough empty token(s) to become active.

Among these bits, only a few determine the state of an instruction — these are the 'condition bits' (Const1, V1, Const2, V2, ConstG, VG, ETReq and ETGot). Other bits indicate data values and the destinations of outputs and empty tokens. If an instruction is to be active, it should satisfy two conditions: $V1 = V2 = VG = 1$ and $ETGot = ETReq + 1$. When an instruction is fired, it clears ETGot and those operand valid bits whose const bits are 0.

Figure 7.6 illustrates how the state of an instruction progresses by changing the values of condition bits. Instruction0 specifies $a \times b$ with conditional control; instruction1 specifies the addition of c and the output of $a \times b$; instruction2 specifies the addition of d and the output of $a \times b$. Assuming that a is a variable and b is a constant number, the diagram of the state transition of instruction0 is triggered by the events shown in the figure. The order of the numbers in this figure is <Const1, V1, Const2, V2, ConstG, VG, ETReq, ETGot>.

7.3 The dataflow coprocessor architecture

The dataflow machine is designed to be a coprocessor to handle repeatedly-used and power-hungry operations such as those introduced in the last chapter. The dataflow coprocessor has no ability to fetch instructions, the main processor decides what kinds of instructions should be fed to it. The architecture of the dataflow coprocessor is illustrated in Figure 7.7.

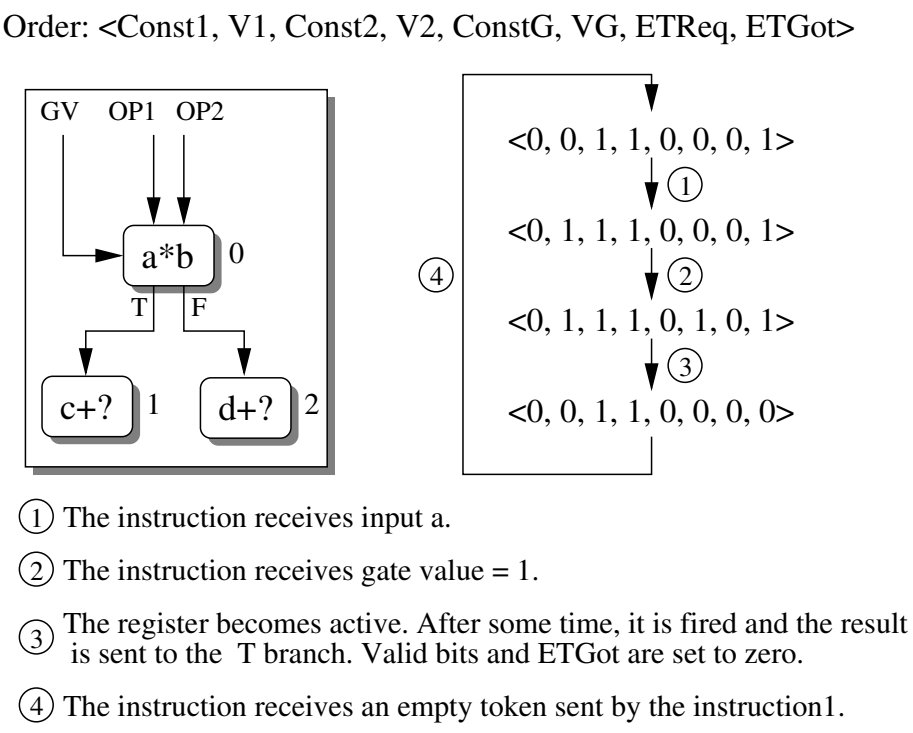


Figure 7.6: The state transition diagram of an instruction

This architecture differs from that of Figure 7.4(a). The matching store is separated into two parts — the Controller and the matching store RAM. Only the 'condition bits' are stored in the Controller because these bits determine the changes in the instruction states. The other 71 bits of an instruction, which have no relation to the change of its state, are put in a single 32×71-bit RAM block. The Control FSMs (finite state machines) are the basic elements of the Control and determine the state of an instruction, their logic function is very simple. If an FSM receives a data token it sets the respective operand valid bit (V1, V2 or VG) to 1. If an FSM receives an empty token it increases ETGot by 1. If $V1 = V2 = VG = 1$ and $ETGot = ETReq + 1$ the FSM becomes active and sends a *Request* signal to a 32-to-1 arbiter in the interconnect network which fires the FSM by sending back an *Acknowledge* signal. This *Acknowledge* signal clears the operand valid bits (if the respective const flag is 0) and ETGot, thus preparing for another execution. The number of Control FSMs in the Controller depends on the capability of the instructions, in this dataflow prototype, there are 32 control FSMs. The *Acknowledge* signals connect to the 32 wordlines of the matching store RAM and the wordline of the 'fired' instruction will be selected.

The scheme of separating the control block and the matching store SRAM not

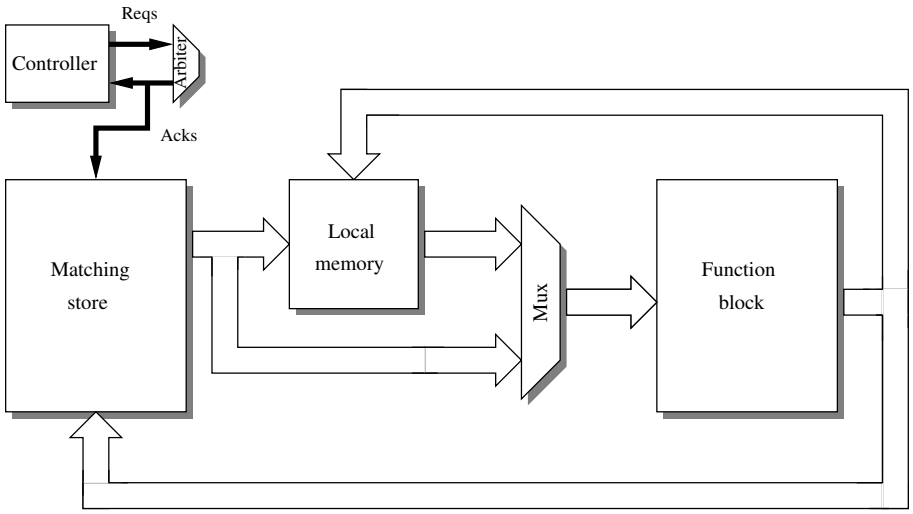


Figure 7.7: The proposed dataflow architecture

only minimizes the size but also reduces the power consumption because most bits are in the RAM block which has a compact structure and is very power-efficient.

Supporting only a load-store instruction architecture is one of the major reasons why RISC processors usually have poor code-density. To overcome this shortcoming, the dataflow coprocessor supports a rich instruction architecture, including register-register, register-memory and memory-memory operations. The current instruction architecture is determined by setting three bits of the OPCode — IA1, IA2 and IAresult. $IA1 = 0$ means OP1 is an immediate value; otherwise, it represents the local memory address of the operand. IA2 also has the same definition. $IAresult = 0$ means the result is sent to the matching store; otherwise, the result is sent to the local memory. The result address is defined by Dest1, DV1, Dest2 and DV1 (totally 14 bits), so the local memory space for results is 16K. The number of pipeline stages can be either 3 or 4. If a local memory read is needed, the 4 pipeline stages are fetching, memory reading, executing and result writing; otherwise, the number of pipeline stages is 3 — fetching, executing and result writing. As will be presented in the next section, using asynchronous control, the dataflow coprocessor can flexibly switch the number of pipeline stages. Consequently, the dataflow coprocessor supports rich instruction architectures by setting only three indicating bits and the execution overhead for supporting these architectures is very small.

7.4 Circuit implementation

7.4.1 Controller design

The interface of the control FSM is illustrated in Figure 7.8:

Figure 7.8: The interface of the control FSM

After initialization, the values of Const1, Const2, ConstG and ETReq will not change until the CPU outputs another instruction. Once the function block sends a data/Boolean token (OP1, OP2 or GV) to an instruction, it will set the corresponding valid bit (V1, V2 or VG) to 1 by sending a pulse on the respective set wire (set_V1, set_V2 or set_VG). If the three data tokens are all valid ($V1 = V2 = VG = 1$), the instruction becomes active and it will set its *Request* high. After arbitration, the interconnect network fires the instruction by pulling *Acknowledge* high. The *Acknowledge* signal clears the valid bit(s) which have the const bit(s) of 0. The matching store reads the respective instruction out and sends it to be executed. After the return to zero phase, the instruction waits for the next set of inputs.

Notice that there are three set wires (set_V1, set_V2 and set_VG) for each register. So, for 32 instructions, the Controller has 96 set wires — a large number. In fact, it does not matter which data token is valid, only is how many. Provided a FSM receives 3 data tokens, it will set the respective instruction active, thus the three set wires can be reduced to a single wire — *Datatoken*. The schematic of the counter circuit for the number of data tokens is illustrated in Figure 7.9.

One pulse on *Datatoken* shifts the data of the three-bit register right by one bit. If the signal $DTValid$ becomes 1, all of the three data tokens are valid. The empty token counter circuit uses a similar design.

The 32 *Request* signals are sent to a 32-to-1 arbiter which answers only one request at one time by issuing a logic high on the respective *Acknowledge* wire; thus only one of the *Acknowledge* wires is active at a time. The 32 *Acknowledge* wires can be used to control the wordlines of the matching store SRAM and the content of the respective SRAM row (an instruction) is sent to be executed.

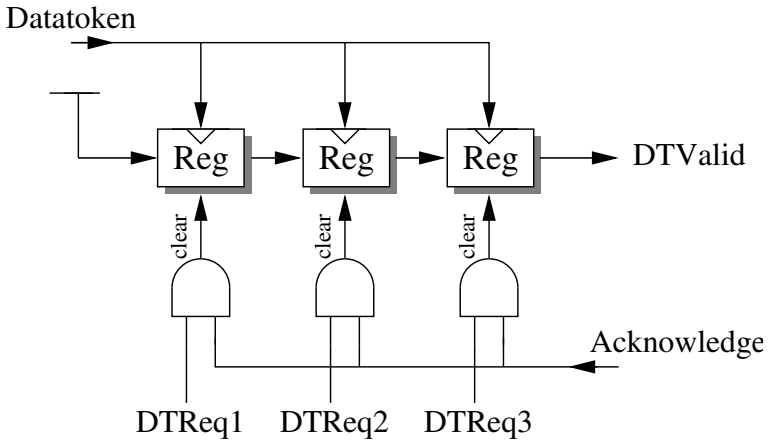


Figure 7.9: The schematic of the data token counter

7.4.2 Pipeline control for various stage numbers

In synchronous circuits, pipelined datapaths usually have a fixed number of pipeline stages, because, otherwise, it is very difficult to define a proper clock period that satisfies each pipeline stage. In asynchronous circuits, the operation of a pipeline is self-scheduling and the number of stages of an asynchronous pipeline can be changed to meet different operating requirements.

To support rich instruction architectures and result duplication, the number of pipeline stages is variable in the dataflow coprocessor. For example, the result of an instruction may be written to one or two positions in the matching store, and memory-memory operations needs an extra pipeline stage to read operands from the local memory. The control circuit for a pipeline having variable stages can be implemented by an asynchronous component called a 'Select-Box'.

A schematic for the Select-Box is shown in Figure 7.10 (a). The signal marked by a diamond sign indicates whether the corresponding pipeline state needs to be executed. The Select-Box just passes Request and Acknowledge signals to and from the next stage if the diamond signal is 0; otherwise, it waits for the execution in the corresponding stage and then relays the Request and Acknowledge signals.

Figure 7.10 (b) shows a circuit that uses B_rm to decide if a local memory read is needed before a data processing operation. $B\_rm = 0$ indicates that the operands sent to the data processing block are immediate numbers and no memory access is needed. If $B\_rm = 1$, the data processing block will wait for the operands taken from the local memory.

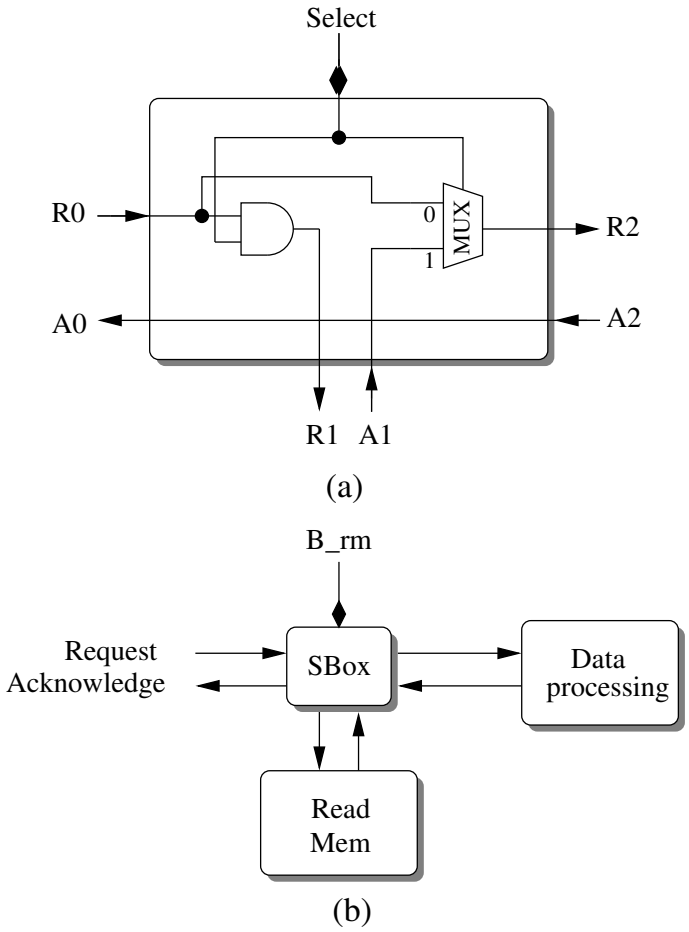


Figure 7.10: An implementation of a pipeline with various stage number

7.5 An automatic mapping algorithm

Since most programmers are familiar only with conventional sequential instructions, an automatic translation program that changes conventional sequential programs to the dataflow programs will greatly simplify the programming of the dataflow coprocessor. An automatic mapping algorithm is proposed in this section to achieve this goal.

The main difference between the instructions of the data coprocessor and conventional RISC processors is that the coprocessor's instruction contains operand values instead of register addresses and the results are sent to the corresponding position in the matching store. The translation from normal RISC codes to dataflow codes requires changing register addresses to the actual operand addresses in the matching store. Figure 7.11 shows an example of changing a conventional ARM code segment to a dataflow code segment that can be executed

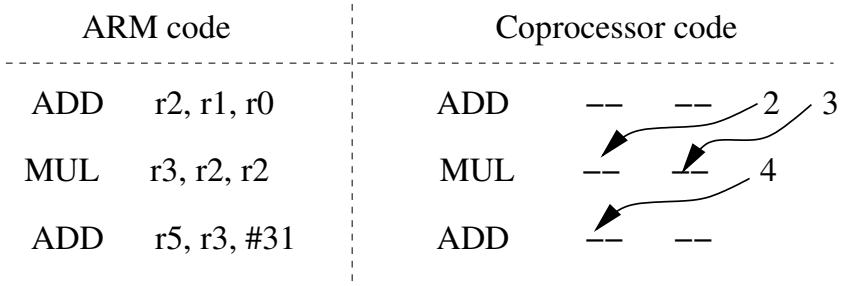


Figure 7.11: An example of mapping RISC codes to dataflow codes

in the coprocessor.

The second ARM instruction uses the result ($r2$) from the first instruction, so when changed to a dataflow code, the first instruction should set its destinations to the addresses of OP1 and OP2 in the second instruction (2 and 3). Following the same principle, the destination of the second instruction sets its destination to the address of OP2 in the third instruction (4).

From the example mentioned above, an automatic mapping algorithm can be implemented by using three steps:

1. Copy the OPCodes and the initialized operand values of RISC instructions line by line to dataflow instructions. Since the coprocessor supports only a simple subset of a general-purpose processor instruction set, the OPCodes of the coprocessor are simpler than those of a microprocessor. Small changes may be needed to generate the coprocessor's OPCodes.

2. Find the destination addresses of each instruction by using register addresses. This can be done by an algorithm called 'converse search'. Take the code segment of Figure 7.11 as an example, in the second instruction, the value in register $r2$ is used for OP1 and OP2, so we can conversely search for the instructions putting their results in $r2$, and set the destinations of the corresponding instructions to the addresses of OP1 and OP2. However, the converse search has two problems as shown in Figure 7.12.

   - As shown in Figure 7.12 (a), the operands of an instruction may be changed by an instruction behind it if there is a backward branch.
   - Some destinations may be superfluous. The only useful instructions that can change the content of a register are the latest ones. As shown in Figure 7.12 (b), Instruction1 does not need to send its result to

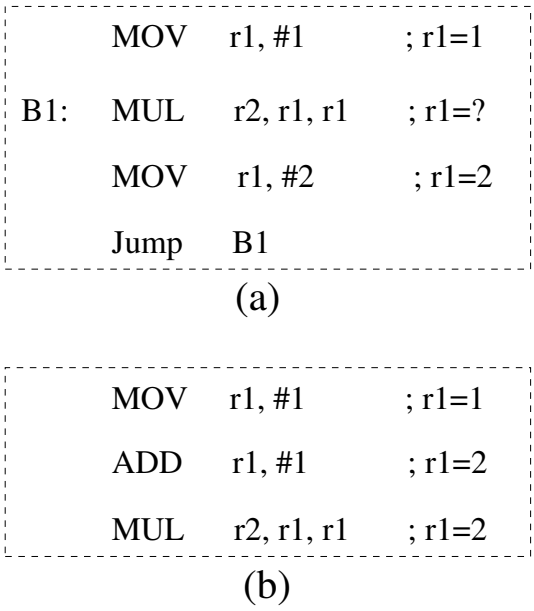


Figure 7.12: Two problems of automatic mapping

Instruction3 because Instruction2 is the one that changes the operands of Instruction3 most recently.

When branches exist, the converse search scheme becomes more complex. Figure 7.13 illustrates a search trace where an attempt is made to find all instructions which send results to a register address (say $r2$). The structure of the search trace is a tree. Tree nodes indicate data processing instructions, trunks indicate sequential program segments and forks indicate branches. The automatic mapping problem amounts to conversely searching all the trunks to find the nearest nodes which send their results to a given register address. Once a node is found in a trunk, this trunk does not need to be searched further. The black trunks indicate the trace being searched, and the grey trunks indicate unnecessary trunk searches. If one trunk leads outside the program segment, the respective operand needs to be initialized by the outside environment.

A recursive algorithm for searching the instructions that send results to a given operand address is as follows:

```
/*AddDest function is used to find all the instructions that
send results to a given Destination. The register address of
this destination in RISC codes is registerNo. The InstNo is
```

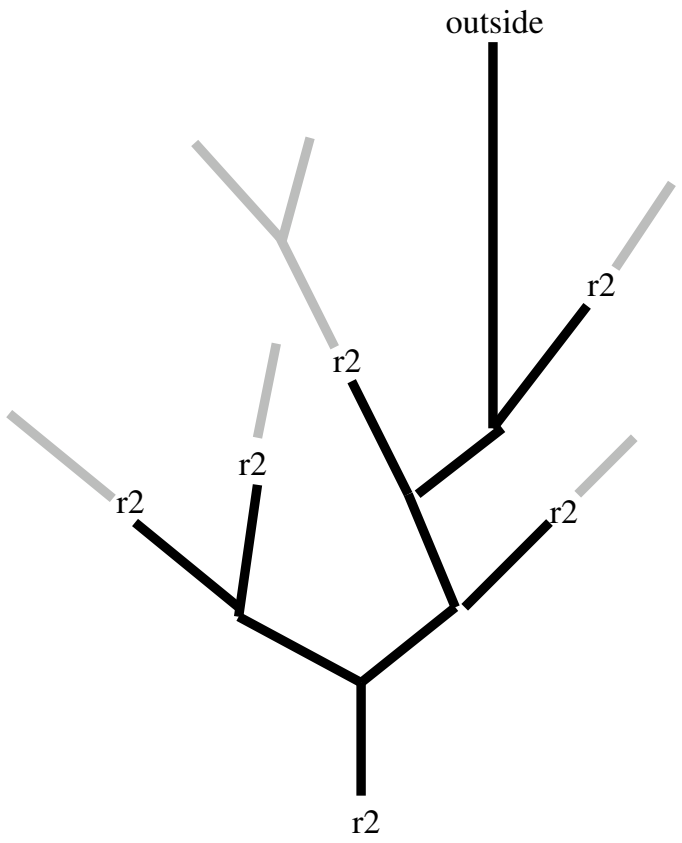


Figure 7.13: The actual trace of a converse search

```
the number of the current instruction that is searched. All
searched instructions are put in a VisitedInst list, which
is used to avoid infinite recursion*/

AddDest(Destination, InstNo, registerNO){
    /*if the search is out of the program segment, add the
    destination in the initialization list*/

    if(InstNo == -1){
        Add Destination in the initialization list;
        return;
    }

    if(InstNo in VisitedInst list)
        return;
```

```
        put InstNo in the VisitedInst list;

        if(the result of (InstNo-1) instruction is sent to
        registerNo){
            Add the Destination to the destination list of
              (InstNo-1) instruction;

            return;
        }

        AddDest(Destination, InstNo-1, registerNo);

        for(all branches (i) which jump to InstNo)
            AddDest(Destination, i, registerNo);

    }
```

For example, in the code segment of Figure 7.11, to find which instructions send results to OP1 of the second instruction, AddDest(3, 2, r2) can be used, where 3 is the address of OP1 in the matching store, 2 means the second instruction, $r2$ means that OP1 is stored in the register address $r2$ in the microprocessor. The result of this function call is that 3 is added to the destination list of the first instruction.

3. Rearrange instructions and destinations

   Sometimes, there are more than two destinations of an instruction, a fork instruction must then be added below it to duplicate results. In this circumstance, the instructions need to be rearranged and the destinations need to be updated.

By using the automatic mapping algorithm proposed above, RISC programs can be automatically translated to programs which can be executed in the dataflow coprocessor, greatly increasing programming efficiency.

Table 7.2: The characteristics of the dataflow coprocessor

Characteristic	Description
Technology	ST 0.18μm
Supply voltage	1.8 volts
Design style	Asynchronous logic using bundled-data scheme
Datapath width	16-bit
Matching store size	32×71-bit
Local memory	1K×16-bit dividing into 4 identical blocks
Function units	An ALU with a 16-bit carry-select adder, a barrel shifter, a 16×16 array multiplier,
Inst architectures	Load-store, memory-register, register-register

Table 7.3: The characteristics of the components

Component	Energy per operation (pJ)	Delay (ns)
Controller	1.36	< 0.3
Interconnect network	7.32	–
Matching store RAM (1 Read and 1.5 writes)	12.8	1.40 (read) 1.40 (write)
16x16-bit Multiplier	34.2	2.61
16-bit Adder/Logic Units	3.17	0.90
Local memory read	4.18	1.50
Local memory write	9.86	1.50

7.6 Experimental results and comparisons

7.6.1 Experimental results

The dataflow coprocessor was implemented at a schematic level using a SGS-Thomson 0.18μm CMOS technology, its characteristics are summarized in Table 7.2. The energy and speed characteristics of each component in the coprocessor are shown in Table 7.3, the delay of the interconnect network is included in the read and write delays of the matching store SRAM.

Five benchmarks are used to evaluate the power efficiency of the dataflow coprocessor. These benchmarks are:

- $SUM = \sum_{i=1}^{100} i$;
- A 5-tap finite impulse response (FIR) filter with 500 inputs;

- A 5-tap infinite impulse response (IIR) filter with 500 inputs;
- The IDEA encryption algorithm with 500 inputs;
- A 4-point fast Fourier transform (FFT) algorithm with 500 inputs.

Table 7.4 shows the number of instructions needed to execute these benchmarks, the numbers of activations of different function units and the energy per instruction of the benchmarks.

Table 7.4: The statistic of energy of the benchmarks

Bench-mark	Inst count	Match store	Read mem	Write mem	Add/ Logic	Mult	Energy/ Inst
SUM	4	400	0	0	400	0	24.65
FIR	15	7500	500	500	5000	2500	35.93
IIR	17	8500	500	500	6000	2500	34.60
IDEA	29	14500	3000	0	12500	2000	29.29
FFT	22	11000	2000	2000	9000	2000	32.84
Average	–	–	–	–	–	–	31.46

As can be seen from Table 7.4, the dataflow coprocessor consumes an average of 31.46 pJ for each instruction, which corresponds to about 31,800 MIPS/W. This power efficiency is based on a schematic-level simulation and a 16-bit datapath, the post-fabrication result of a 32-bit dataflow coprocessor is estimated to be about 10,000 MIPS/W. Using the same supply voltage and under the same test temperature, the power consumption of a low-power commercial ARM processor — AT91R40008 [79] — with all peripheral clocks deactivated is 0.73 mW/MHz, which equals 1,370 MIPS/W. Therefore, the dataflow coprocessor can improve the power-efficiency of AT91R40008 by a factor of 7.3.

The main reason for the power-efficiency of the dataflow coprocessor is due to it minimizing the execution overheads compared with a general purpose CPU. For the FIR benchmark, data processing uses as much as 40% of the overall processor power consumption — much more than in a general-purpose microprocessor.

The power-saving strategy of the coprocessor design includes four aspects as follows:

- Principle of locality

  The principle of locality is exhibited in embedded processing applications and the coprocessor exploits it to achieve low power consumption. Firstly,

the matching store contains only frequently-used code segments assumed to dominate the dynamic execution trace of embedded programs. Thus, most of the time, the coprocessor does not need to access a large instruction cache, instead, it fetches instructions from a 32-word SRAM. Secondly, the local memory makes the stream-based data processing more power-efficient.

- Specific coprocessor architecture

  The coprocessor does not use a conventional RISC-like approach but employs a dataflow architecture. As will be discussed later, the dataflow architecture is more power-efficient than a RISC one under certain circumstances. A few specific low-power circuit implementations are also used in the dataflow coprocessor, such as splitting the controller and the matching store, a negative synchronization scheme and a single-port matching store.

- A simple instruction set

  The coprocessor supports only a small number of the most commonly used instructions. Therefore, many execution overheads due to fetching and decoding of an abundant instruction set are saved. Supporting only a small number of instructions also results in simpler datapath design.

- Asynchronous logic

  The coprocessor is designed using asynchronous logic, which offers very flexible control to support various function unit delays and number of pipeline stages and also provides a fine-grain clock gating. The matching store, the function units and the local memory are executed in a form of 'pay as you go'. Only the useful parts use energy in the execution of an instruction.

To sum up, the dataflow coprocessor proposed in this chapter can minimize the power consumption of a general-purpose microprocessor by a factor of 5-10. The power saving comes from minimizing execution overheads, employing a dataflow scheme, and using a few power efficient circuit implementations.

7.6.2 Comparisons

Two kinds of coprocessors have been presented in the thesis — one uses a conventional RISC architecture but employs a single-port SRAM as the register file and avoids pipeline hazard reduction overheads (P1 introduced in the last chapter),

the other one uses a dataflow architecture with simple pipeline control. There is a natural question — is the dataflow coprocessor more power efficient than a RISC coprocessor which has a very small loop buffer and supports only a very simple instruction set? The question is addressed in this section, looking at power consumption and performance.

Power comparison

Let us review the architectural characteristics of the two coprocessors. The dataflow coprocessor, 'DACO' (DAtaflow COprocessor), has a 16-bit datapath and a matching store which stores 32 71-bit instructions. The RISC coprocessor, 'RICO' (RIsc COprocessor), also has a 16-bit datapath and an instruction buffer which stores 32 instructions. RICO's instruction length is 20-bit. Let us assume that DACO and RICO have exactly the same function blocks, and use the same amount of energy in data (e.g. arithmetic/logic) processing, memory references and pipeline control.

Let us first compare how many bits these two coprocessors need to fetch and store in a typical instruction execution. To execute one instruction, RICO needs to fetch a 20-bit instruction, access the register file (SRAM) and read two 16-bit operands. The operands are propagated to the function block and a 16-bit result is sent back to the register file. So, for a typical instruction, RICO reads 52 bits and writes 16 bits. For DACO to execute one instruction, it reads a 71-bit instruction and writes a 16-bit result to the matching store. It seems that DACO uses more energy in fetching and storing than RICO does; however, energy is not simply proportional to the number of bits read and written because the coprocessors have two differences:

1. DACO reads a 71-bit word from the matching store SRAM, but RICO reads a 20-bit word from the instruction buffer and two 16-bit words from the register file. The power consumption is not directly proportional to the number of bits read because reading three L-length words uses more energy than reading one $3L$-length word. The power consumption of one reading operation from a 32×71 SRAM is 7.8 pJ (0.18 μm, 1.8 V), whilst the power consumption of one read from a 32×16 SRAM is 3.1 pJ (3.3 pJ for a 32×20 SRAM). The reason is that, in a small SRAM, the decoder power dominates the overall power consumption. Read three L-length words needs

three decoder operations while read one $3L$-length words needs only one decoder operation.

2. In RICO, the operand values in the register file can be referred to several times by different instructions. However, DACO's instructions normally contain only immediate values, and the result of an instruction may need to be duplicated and sent to several different addresses in the matching store, thus DACO may write more bits than RICO. Fortunately, the duplication operations do not occur frequently in normal data processing algorithms. Based on benchmark experiment, a rough estimate is that DACO writes 1.5x the number of bits written by RICO and uses 1.5x the write power. The power consumptions of writing a 16-bit word to the matching store and the register file are similar using a DWL approach of Figure 5.2 at 3.3 pJ.

Consequently, DACO uses $7.8+3.3\times1.5 = 12.75$ pJ in instruction fetching and result storing for every instruction execution, whilst RICO uses $3.3+3.1\times2+3.3 = 12.8$ pJ, the two energy consumptions are similar.

The problem of result duplication can be alleviated by using a 'store pool'. If the result of a DACO instruction needs to be duplicated many times, the result can be put into in a 'store pool' which is very small (4 words is enough) and is placed at the bottom of the local memory as shown in Figure 7.14. The store pool acts like the register file of RICO and because of its small size, the store pool is accessed quickly and power efficiently. Every instruction can refer the store pool by setting $OP1$ or $OP2$ as a local address, and so the duplications can be avoided.

A significant difference between DACO and RICO instruction sets is that DACO has a very large instruction-length which allows a more freedom for different low-power techniques, one of which is to use these bits to support abundant instruction set architectures. As discussed in Section 7.3, by setting three bits of OPCode, DACO supports different instruction set architectures, including memory-memory, register-memory and register-register (load-store) architectures. A processor supporting a memory-memory has a high code-density. For example:

```
                C=A+B

Memory-memory:   ADD   [C], [A], [B]  ;[C] means the address of C
```

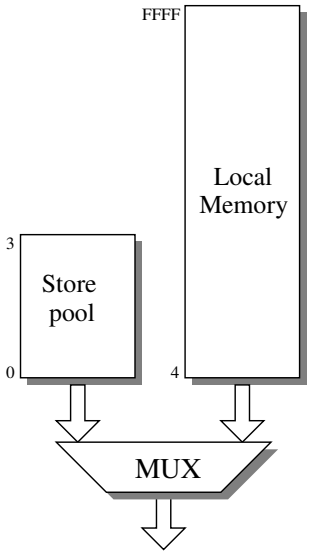


Figure 7.14: An architecture for reducing duplications

```
Load-store:     Load  r1, [A]
                Load  r2, [B]
                ADD   r3, r1, r2
                Store [C], r3
```

From the example above, a memory-memory architecture needs only one instruction to describe $C = A + B$, while a load-store architecture needs 4 instructions. Supporting only load-store architectures usually makes RISC processors fetch more instructions to execute a given program than CISC processors, i.e., a RISC processor has a low code-density. When executing general programs, the dynamic usage of load and store instructions is about 43% for ARM processors [1]. For signal processing applications, stream-based computations occupy most of the execution time. These computations refer to memory very frequently, so the percentage of load and store instruction is even greater than 43%. For this reason, DACO greatly reduces the number of instruction fetches. Based on the test of a matrix multiplication, a processor with a memory-memory architecture reduces the number of instruction fetches by about 40% compared to a processor with a load-store architecture.

A long instruction has another advantage — good support for immediate values. For RICO, since an immediate value is coded in the 20 bits of instruction, it is not possible to support 16-bit immediate numbers (from Figure 6.6, RICO

supports 10-bit immediate numbers). Large immediate values must be assembled by several instructions. For example:

```
r2=1111

MOV r3, #87
MOV r2, #1
LSL r2, #10
ADD r2, r2, r3
```

Normally, RICO will only occasionally use an immediate value larger than 1024, so DACO has only a small advantage in this respect. However, for applications using a lot of 16-bit immediate numbers, DACO may save a number of instruction references.

Another difference between RICO and DACO is that DACO does not need branch instructions thus saving power due to branch instruction fetches. However, DACO needs to write the results of arithmetic comparisons back to the matching store, counteracting this power saving. In this respect, RICO and DACO are estimated to use the same power.

To sum up, DACO uses about $(3.3+3.1\times2+3.3)\times1.4-(7.8+3.3\times1.5) = 5.17$ pJ less than RICO in one instruction execution (based on an assumption that the memory reference of RICO is 40% bigger than that of DACO). The power saving of DACO mainly comes from supporting rich instruction set architectures. Executing applications with a low data reference rate, DACO and RICO achieve similar power-efficiency.

Performance comparison

Figure 6.7 compares the performances of different RISC architectures. To save 1/3 of the overall power consumption, RICO uses a single-port SRAM as its register file and avoids pipeline hazard reduction techniques. The impact of these approaches is that the performance is reduced by 50%.

DACO also uses a single-port SRAM as its matching store and has data dependence problems, the impact of data dependence problems can be alleviated however by the dataflow scheme it uses. Figure 7.15 shows a dataflow diagram of the FIR program introduced before. As can be seen from the figure, there is a lot of parallelism in this program and DACO can automatically choose the active

instructions to execute. If there is an instruction waiting for the results of the executing instructions, DACO will avoid a pipeline stall by selecting other active instructions. As long as there are three independent program threads (these situations are very common in data processing algorithms), DACO will not suffer from data dependent problems, because DACO has at most four pipeline stages.

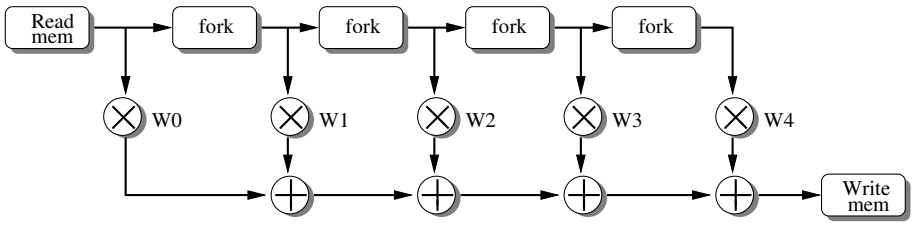


Figure 7.15: A dataflow diagram for FIR

Figure 7.16 shows the executing segments of DACO and RICO when they execute the FIR program. As can be seen from the figure, the executing segment of RICO is highly regular because of its sequential control. For DACO, however, the executing segment is irregular because the arbiter randomly selects active instructions and the executing sequence of instructions is ambiguous. The experiment shows that DACO is 1.6 times faster than RICO, which can be seen from the figure. The first reason for this is due to DACO's dataflow scheme, the second reason is that DACO reads two operands at the same time but RICO needs to read the register file twice. Although DACO may need to write results more than once, this situation is very rare. Therefore, DACO is still faster than RICO in this scenario.

To sum up, DACO is faster than RICO because DACO can alleviate pipeline stalls due to the data dependence of one instruction by selecting other active instructions. How much faster DACO is than RICO depends on the actual programs. If a program contains more than two independent threads and each thread has data dependence, DACO can be much faster than RICO. However, DACO is still not as fast as a RISC coprocessor using pipeline hazard reduction techniques (such as P4 introduced in the last chapter) because it uses a single-port matching store and cannot totally avoid data dependence problems.

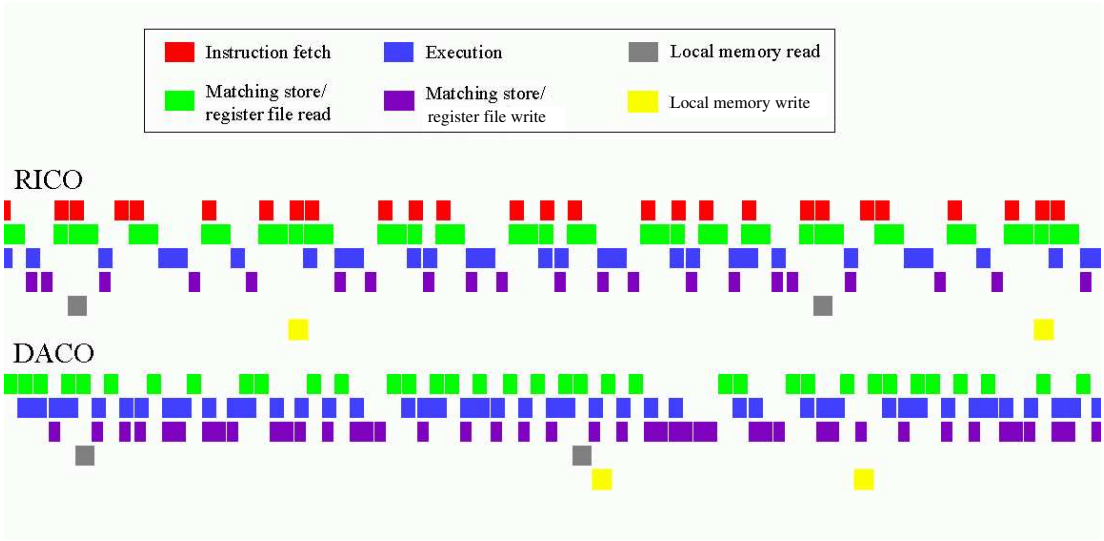


Figure 7.16: The pipeline operations of FIR in RICO and DACO

7.7 Summary

The performance and power consumption of the RISC coprocessor and the dataflow coprocessor have been compared in this chapter. The advantages of the RISC architecture and dataflow architecture are as follows:

- Advantages of the RISC architecture
  - Easy programming
  - Good scalability
- Advantages of the dataflow architecture
  - Low power to some extent
  - Fast to some extent
  - Simple datapath control

Since most people are familiar with conventional programming, the RISC coprocessor is easier to program. The automatic mapping algorithm proposed in this chapter should help to alleviate the difficulty of programming the dataflow coprocessor.

From the analysis presented here, the dataflow coprocessor is more power-efficient than the RISC coprocessor, the analysis however is based on the assumption that only 32 instructions can be stored in the coprocessors. If more

frequently-used instructions need to be stored in the dataflow coprocessor, the size of the matching store must be enlarged and the overall power consumption may be dominated by the matching store. At the same time, if the RISC coprocessor also enlarges its instruction buffer, the power efficiency of using a register file will be significant. Although the impact on power consumption due to a large matching store can be reduced by using a shifting window scheme [80], it is estimated that the dataflow coprocessor will be power inefficient when the matching store becomes large enough to store 128 instructions. Fortunately, a large instruction store is not encouraged in a hierarchical processing architecture.

A prominent advantage of the dataflow architecture is that it can achieve a reasonable performance using a simple datapath control. For a RISC processor to improve performance, sophisticated branch prediction techniques and complex bypass paths are needed, which greatly increase design complexity for datapath control. For a dataflow processor, good performance is gained by using a dataflow scheme, there is no need to bypass results from one block to another. Thus, each block has only one input port and one output port, which requires a simple datapath control.

A low-power dataflow coprocessor has been designed and tested in this chapter. The coprocessor is shown to improve the power-efficiency over a general-purpose RISC processor by an order of magnitude. Therefore, in a hierarchical processing architecture, if 50% of the dynamic instructions are executed in the coprocessor, the overall power saving will be about 45%. An automatic mapping algorithm is also proposed in the chapter, which makes it easy to program the coprocessor.

The next chapter is a conclusion chapter which summarises the low power techniques which have been presented in the thesis.

Chapter 8

Conclusions

Future embedded processing systems call for low power consumption. The work discussed in this dissertation has presented and evaluated a range of techniques for power-efficient embedded processing. The low-power embedded processing techniques presented in the dissertation focus on three main aspects:

- Clock power consumption is becoming a serious design concern as the size of embedded processing chips increases. Asynchronous logic design is an alternative approach which may offer low power consumption. Chapter 3 compares the power-efficiencies of asynchronous and synchronous circuits. Low-power asynchronous designs are also summarized in the chapter.

- The low power designs presented in Chapter 4 and Chapter 5 are circuit-level optimizations minimizing the power consumption of essential processing units, including adders, multipliers and memory. Based on a sign-changing algorithm and an early-termination algorithm, an asynchronous pipelined iterative multiplier is proposed in Chapter 4. The multiplier is shown to increase the power-efficiency of a conventional synchronous multiplier by a factor of 2.5. Chapter 4 also presents the design of an asynchronous adder. Using a carry-lookahead scheme, the adder greatly reduces the critical delay of a ripple-carry asynchronous adder. The proposed adder is 27% faster than a ripple-carry asynchronous adder at the cost of 15% power overhead. The proposed adder is also shown to be faster and lower power than a conventional synchronous carry-select adder.

  Chapter 5 presents low-power techniques for an embedded SRAM. The

techniques include low voltage swing bit-line write techniques and a dual-rail decoder. A 64×32-bit SRAM macro was designed and tested in this chapter. The layout-based simulation results show the proposed SRAM macro is 1.6x faster and 4x more power-efficient than a commercial SRAM for only a 3% area overhead cost.

- The low-power techniques presented in Chapters 6 and 7 are architecture-level optimizations to minimize execution overheads in programmable architectures. Based on the analysis of embedded processing programs, a hierarchical processing scheme is proposed. The basic idea behind this scheme is that an embedded processing system should include more than one processing unit arranged in a hierarchy. Complex processing units are used to maintain general-purpose functionality and simple processing units are used to execute the most commonly used operations and to increase the power efficiency of the overall system. A two-level hierarchical processing architecture and a specific RISC-like coprocessor are presented in Chapter 6. The primary experimental results show that hierarchical processing is promising in terms of power-efficiency.

  Chapter 7 explores the power saving potential of a less-studied approach — a dataflow architecture. A dataflow coprocessor is designed and tested in this chapter. Specific low-power techniques are used in the dataflow coprocessor, including an efficient synchronization scheme and some circuit implementations. The power consumption and performance of the dataflow coprocessor and a RISC-like coprocessor are compared. The experimental results show that the proposed dataflow coprocessor exceeds the power efficiency of an ARM processor by a factor of 7.3. The power saving mainly comes from the simple functionality of the coprocessor in a hierarchical architecture.

Although low-power embedded processing is a very broad research field and it is impossible to address all power-saving techniques in a single thesis, two points are clear from the research as follows:

- Low power design is a complex task calling for effort at all levels of the design hierarchy. A power-efficient design results from a lot of comparisons and evaluations of various different implementations and architectures. It

also requires a lot of trade-offs among many aspects, including performance, hardware and power consumption.

- Analyzing the specific characteristics of a targeted design is an efficient approach to find low-power techniques. Through analysis, the most power hungry components are found and these components point to the directions that will potentially lead to power savings. Low-power techniques should be evaluated using real applications or benchmarks which can represent the characteristics of these applications.

8.1 Future work

Firstly, hierarchical processing gives a promising direction for low-power embedded processing but it also introduces a lot of design complexity, especially for software support. Although as discussed before, some very important embedded program segments are hand-optimized, an architecture with software support is more practical and more easily commercialized. The work presented in this dissertation includes only a hardware architecture and a few software considerations. However, the design flow shown in Chapter 7 reveals the basic steps and components for an automatic software environment:

- A simulator and a compiler for the CPU:

  The compiler compiles high-level programs into machine languages. The compiled programs can be functionally simulated in the simulator.

- An analyzer

  The analyzer analyzes the dynamic execution trace of the compiled programs and the most commonly executed program segments are found. The power consumption of these segments is evaluated using data stored in a power library as in Table 7.3. The communication overheads are also analyzed. Then the decision is made about which segments should be executed in the coprocessor.

- An automatic translator

  This translates the program segments found in the second step into a program which can be executed in the coprocessor, such as the automatic mapping algorithm proposed in Section 7.5.

Designing a software environment to support the hierarchical processing architecture proposed in the dissertation is one direction to extend this work.

Secondly, the motivation behind the coprocessor designs in the dissertation is mainly to demonstrate the power-efficiency of the hierarchical architecture, so the instruction sets are designed in a straightforward way. Effort has not been made to optimize them. Another direction to extend this work is to optimize and evaluate the instruction set of the coprocessor.

Bibliography

[1] S. Furber. *ARM system-on-chip architecture*. Addison-Wesley, 2000.

[2] T.K. Callaway and E.S. Swartzlander. The power consumption of CMOS adders and multipliers. *in [10]*.

[3] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach. 3rd Edition*. Morgan Kaufmann Publishers, 2003.

[4] Forward Concepts. DSP/wireless market bulletin. *http://www.fwdconcepts.com/*, January 2005.

[5] J. Montanaro, R.T. Witek, K. Anne, and et. al. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *IEEE Journal of Solid-State Circuits*, 31:1703–1714, November 1996.

[6] J. D. Garside. A CMOS VLSI implementation of an asynchronous ALU. *Proceedings of the IFIP Working Conference on Asynchronous Design Methodologies, Manchester, England*, pages 181–192, 1993.

[7] G.E. Moore. Cramming more components onto integrated circuits. *Electronics*, pages 114–117, April 1965.

[8] International technology roadmap for semiconductors report, 2004 update. *http://www.itrs.net/*, 2004.

[9] J. Rabaey. Ultra low-power computation and communication enables ambient intelligence. *Keynote at 2003 Smart Objects Conference in Grenoble*, April 2003.

[10] A. Chandrakasan and R. Brodersen. *Low Power CMOS Design*. Wiley-IEEE Press, 1998.

[11] Y. Lin, C. Wu, and et. al. Leakage scaling in deep submicron CMOS for SoC. *IEEE Transactions on Electron Devices*, 49:1034–1041, June 2002.

[12] N.S. Kim, T. Austin, and et. al. Leakage current: Moore's law meets static power. *IEEE Transactions on Electron Devices*, 36:68–74, December 2003.

[13] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31:1277–1283, September 1996.

[14] Q. Wu, M. Pedram, and X. Wu. Clock-gating and its application to low power design of sequential circuits. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 47:479–482, March 2000.

[15] P. Bose, M. Martonosi, and D. Brooks. Modeling and analyzing CPU power and performance: metrics, methods and abstractions. *http://www.princeton.edu/~mrm/tutorial/Sigmetrics2001_tutorial.pdf*.

[16] M.K. Gowan, L.L Biro, and D.B. Jackson. Power considerations in the design of the Alpha 21264 microprocessor. *Proceedings of 1998 Design Automation Conference*, pages 726–731, June 1998.

[17] A. Efthymiou. Asynchronous techniques for power-adaptive processing. *PhD Thesis, School of Computer Science, University of Manchester*, 2002.

[18] V. Stojanovic and V.G. Oklobdzija. Comparative analysis of master-slave latches and flip-flops for high-performance and low-power systems. *IEEE Journal of Solid-State Circuits*, page 536–548, April 1999.

[19] W.M. Chuang and M. Sachdev. A comparative analysis of dual edge triggered flip-flop. *Proceedings of 2000 Conference of Electrical and Computer Engineering*, pages 1034–1041, 2000.

[20] H. Kojima, S. Tanaka, and K. Sasaki. Half-swing clocking scheme for 75% power saving in clocking circuitry. *IEEE Journal of Solid-State Circuits*, pages 432–435, April 1995.

[21] V. Tiwari, D. Singh, S. Rajgopai, G. Mehta, R. Patel, and F. Baez. Reducing power in high-performance microprocessors. *Proceedings of 1998 DAC*, pages 1034–1041, 1998.

[22] R. S. Bajwa. et. al. Instruction buffering to reduce power in processors for signal processing. *IEEE transition on VLSI*, page 417C424, 1997.

[23] L. H. Lee, W. Moyer, and J. Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. *Proceedings of 1999 International Symposium on Low Power Electronics and Design*, pages 267–269, 1999.

[24] C. Wu and T. Hwang. Instruction buffering for nested loops in low power design. *Proceedings of 2002 IEEE International Symposium on Circuits and Systems*, pages 81–84, May 2002.

[25] ARM Co. Improving ARM code density and performance. *www.arm.co.uk*, 2003.

[26] C. Pigue, J. Masgonty, and et. al. Low-power design of 8-b embedded CoolRISC microcontroller cores. *IEEE Journal of Solid-State Circuits*, pages 1067–1078, July 1997.

[27] K. Compton. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys*, page 171–210, June 2002.

[28] J. Sparsøand S. Furber (eds). *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001.

[29] S. Hauck. Asynchronous design methodologies, an overview. *Proceeding of IEEE*, pages 69–93, January 1995.

[30] T. Verhoeff. Delay-insensitive codes — an overview. *Distributed Computing*, pages 1–8, 1988.

[31] A.J. Martin. The limitations to delay-insensitivity in asynchronous circuit. *Advanced Research in VLSI: Proceedings of the sixth MIT Conference*, pages 263–278, 1990.

[32] D.E. Muller and W.S. Bartky. A theory of asynchronous circuits. *Proceedings of 1957 International Symposium on the Theory of Switching*, pages 204–243, April 1957.

[33] D.E. Muller. Asynchronous logics and application to information processing. *Proceedings of 1963 Symposium on Application of Switching Theory in Space Technology*, pages 289–297, 1963.

[34] I.E. Sutherland. Micropipelines. *Communications of the ACM*, pages 720–738, June 1989.

[35] S. Furber and P. Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 247–253, June 1996.

[36] T.A. Chu. Synthesis of self-timed VLSI circuits from graph-theoretic specification. *PhD thesis, MIT*, 1987.

[37] V. Ekanayake, C. Kelly, and et. al. Bitsnap: Dynamic significance compression for a low-energy sensor network asynchronous processor. *Proceeding of 11th International Symposium on Asynchronous Circuits and Systems*, pages 144–154, March 2005.

[38] A. Martin, S.M. Nystrm, K. Papadantonakis, and et. al. The lutonium: a sub-nanojoule asynchronous 8051 microcontroller. *Proceeding of Ninth International Symposium on Asynchronous Circuits and Systems*, pages 14–23, May 2003.

[39] S. Furber, D. Edwards, and J. Garside. Amulet3: a 100 MIPS asynchronous embedded processor. *2000 International Conference on Computer Design*, pages 329–334, September 2000.

[40] L. Nielsen and J. Sparso. Designing asynchronous circuits for low power: an IFIR filter bank for a digital hearing aid. *Proceedings of the IEEE*, pages 268–281, February 1999.

[41] L. Nielsen. Low-power asynchronous VLSI design. *PhD thesis, Department of Information Technology, Technical University of Denmark*, 1997.

[42] N. Paver and D. Edwards. Is asynchronous logic good for low-power? *IEE Colloquium on Low Power Analogue and Digital VLSI: ASICS, Techniques and Applications*, pages 1–5, June 1995.

[43] Y. Liu and S. Furber. The design of a low-power asynchronous multiplier. *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, pages 301–306, August 2004.

[44] S. Furber, J. Garside, and et. al. Amulet2e: An asynchronous embedded controller. *Proceedings of the IEEE*, pages 243–256, February 1999.

[45] J.L. Hill. System architecture for wireless sensor networks. *PhD thesis, University of California, Berkeley*, 2003.

[46] Y. Liu and S. Furber. Minimizing the power consumption of an asynchronous multiplier. *Proceedings of 2004 PATMOS, Springer*, pages 289–300, September 2004.

[47] M. Lewis, J.D. Garside, and L.E.M. Brackenbury. Reconfigurable latch controllers for low power asynchronous circuits. *Proceedings of 1999 International Symposium on Asynchronous Circuits and Systems*, pages 27–35, April 1999.

[48] A.R. Omondi. *Computer Arithmetic Systems: Algorithms, Architectures and Implementations*. Prentice Hall, 1994.

[49] N.H. Weste and K. Eshraghian. *Principles of CMOS VLSI design: A System Perspective*. Addison-Wesley Publisher, 1992.

[50] L. Bisdounis. Circuit techniques for reducing power consumption in adders and multipliers. *in "Designing CMOS Circuit for Low Power" (D. Soudris, C. Piguet and C. Goutis eds.)*, 2002.

[51] A.P. Chandrakasan and R.W. Brodersen. *Low Power Digital CMOS Design*. Kluwer Academic Publishers, 1995.

[52] O.J. Bedrij. Carry-select adder. *IRE Transactions on Electronic Computers*, pages 340–346, June 1962.

[53] B. Gilchrist, J.H. Pomerene, and S.Y. Wong. Fast carry logic for digital computers. *IRE Transactions on Electronic Computers*, EC-4(4):133–136, November 1955.

[54] D. Goldberg. Computer arithmetic. *in [3]*.

[55] L.S. Nielsen and J. Sparsø. A low-power asynchronous data-path of a FIR filter bank. *Proceedings of 1996 International Symposium on Asynchronous Circuits and Systems*, pages 197–207, 1996.

[56] C.S. Wallace. A suggestion for a fast multiplier. *IEEE Transitions on Electronic Computer*, EC-13:14–17, 1964.

[57] J. Liu. Arithmetic and control components for an asynchronous system. *PhD thesis, Department of Computer Science, The University of Manchester*, 1997.

[58] J. Gu and C. Chang. Low voltage, low power (5:3) compressor cell for fast arithmetic circuits. *Proceedings of 2003 IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2003.

[59] O.L. Mac Sorley. High-speed arithmetic in binary computers. *IRE Proceedings*, 49:67–91, January 1961.

[60] R. Fried. Minimizing energy dissipation in high-speed multipliers. *Proceedings of 1997 IEEE International Symposium on Low Power Electronics and Design*, pages 214–219, 1997.

[61] V. Zyuban and P. Kogge. Split register file architectures for inherently low power microprocessor. *Proceedings of Power Driven Microarchitecture Workshop at ISC98*, pages 32–37, 1998.

[62] K. Itoh, K. Sasaki, and Y. Nakagome. Trends in low-power ram circuit technologies. *Proceedings of the IEEE*, page 524–543, April 1995.

[63] M. Margala. Low-power sram circuit design. *Proceedings of 1999 IEEE International Workshop on Memory Technology, Design and Testing*, pages 115–122, August 1999.

[64] K. Kanda, H. Sadaaki, and T. Sakurai. 90% write power-saving SRAM using sense-amplifying memory cell. *IEEE Journal of Solid-State Circuits*, pages 927–933, June 2004.

[65] M. Yoshimito, K. Anami, and et. al. A divided word-line structure in the static RAM and its application to a 64K full CMOS RAM. *IEEE Journal of Solid-State Circuits*, pages 479–485, October 1983.

[66] T. Hirose, H. Kuriyama, and et. al. A 20-ns 4-Mb CMOS SRAM with hierarchical word decoding architecture. *IEEE Journal of Solid-State Circuits*, pages 1068–1077, October 1990.

[67] J. Alowersson and P. Andersson. SRAM cells for low-power write in buffer memories. *Proceedings of 1995 IEEE Symposium on Low Power Electronics*, pages 60–61, October 1995.

[68] K.W. Mai, T. Mori, and et. al. Low-power SRAM design using half-swing pulse-mode techniques. *IEEE Journal of Solid-State Circuits*, pages 1659–1671, November 1998.

[69] J. Wang, P. Yang, and W. Tseng. Low-power embedded SRAM macros with current-mode read/write operations. *Proceedings of 1998 International Symposium on Low Power Electronics and Design*, pages 282–287, August 1998.

[70] B.S. Anrytyr. Design and analysis of fast low power SRAMs. *PhD thesis, Stanford University*, 1999.

[71] ST Co. Hemos8dsps4 datasheet. 2002.

[72] J. G. Proakis and D. Manolakis. *Digital Signal Processing: Principles, Algorithms and Applications (3rd Edition)*. Prentice-Hall Engineering/Science/Mathematics, 1995.

[73] S. Segars. ARM7TDMI power consumption. *IEEE Journal of Micro*, pages 12–19, July 1997.

[74] B.A. Warneke and K. Pister. An ultra-low energy microcontroller for smart dust wireless sensor networks. *Proceedings of 2004 IEEE International Solid-State Circuits Conference*, pages 16–18, 2004.

[75] D.A. Patterson and D.R. Ditzel. The case for the reduced instruction set computer. *ACM SIGARCH Computer Architecture News*, pages 25–33, October 15 1980.

[76] A.H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, pages 365–396, December 1986.

[77] J. Gurd and I. Watson. A data driven system for high speed parallel computer. *Computer Design*, pages 97–106, June 1980.

[78] J.B. Dennis and D.P. Misunas. A preliminary architecture for a basic data-flow processor. *Proceedings of the 2nd Annual Symposium on Computer Architecture*, pages 126–132, December 1974.

[79] Atmel Co. AT91R40008 electrical characteristics. *http://www.atmel.com/dyn/resources/prod_documents/doc1795.pdf*, 2002.

[80] D.A. Patterson and C.H. Sequin. RISC I: A Reduced Instruction Set VLSI Computer. *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 443–457, 1981.