ASYNCHRONOUS TECHNIQUES FOR POWER-ADAPTIVE PROCESSING

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

December 2002

By
Aristeidis Efthymiou
Department of Computer Science

Contents

List of Tables

List of Figures

Abstract

Power consumption has become a significant concern in the design of digital integrated circuits. A solution to the need for both low-power and high-performance systems is power-adaptivity, the capability of a system to dynamically scale its power consumption according to the demand for processing.

This thesis examines micro-architectural techniques that can be used to build an asynchronous, power-adaptive microprocessor. Two main techniques are presented using AMULET3, an asynchronous processor designed at the University of Manchester, as a basis for development and evaluation. The first controls the pipeline occupancy using a token mechanism, while the second enables adjacent pipeline stages to be merged, thus altering the processor's micro-architecture. These techniques manage the processor's power consumption by controlling its speculation depth. The execution time may be increased but, if the method is applied to programs with slack time, the user-perceived performance will not be degraded. Hardware-based methods for controlling the speculation depth dynamically are also investigated.

A large proportion of a system's power budget is also attributable to its memory. As a step towards the design of a power-efficient memory system, a self-timed, adaptive, Content-Addressable Memory (CAM), for use in associative caches, is developed that consumes almost a quarter of the power of a standard CAM.

Together these techniques exploit asynchronous design in a way which would be difficult in a clocked system. Results are presented which show that such techniques are of significant benefit, but only in certain classes of programs; thus for a general-purpose processor the ability to control the micro-architecture dynamically adds more operating flexibility and potentially greater energy and power savings.

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

The author

Aristides Efthymiou received the B.Sc. and M.Sc. degrees in Computer Science from the university of Crete in Greece in 1993 and 1995 respectively. Uppon completing his first undergraduate year he joined the Institute of Computer Science of the Foundation of Research and Technology Hellas (ICS-FORTH) as a trainee working on medical image compression and later on VLSI design. His Masters thesis concerned the design, implementation, and testing of a 25 Gb/s pipelined memory switch buffer in full-custom CMOS which was fabricated and operated successfully with minor errors. After finishing his military service he joined the Group for Chacterization and Design (GCD), Institute of Microelectronics, National Centre of Scientific Research "Demokritos", Athens, Greece in 1997, where he worked on low-voltage, low-power circuit design. He joinned GCD's spin-off company, ISD S.A. in 1998, where he worked on low-voltage DSP processors and ADSL modems. He started his pursuit of the Ph.D. degree with the AMULET group of the Department of Computer Science at the University of Manchester in October 1999.

Acknowledgements

The person who deserves the most credit is my wife, Christina. Without her love and support I would not have started and, definitely, would not have finished this work. For this and for her wholehearted devotion, I am profoundly grateful.

My work was made considerably easier by the encouragement of my friends and parents. I deeply thank them all for their affective presence.

This research was completed under the supervision of Dr. Jim Garside to whom I am obliged. Throughout these years I valued greatly his insights, guidance and support. I would also like to thank my advisor Prof. Steve Furber for his effective counsel on a number of occasions.

Special thanks go to Peter Riocreux and Dr. Steve Temple for efficiently proofreading this thesis in record time. I would also like to thank Dr. John Bainbridge for his useful and detailed comments on the thesis and Tomaz Felicijan for reading some preliminary parts. Thanks to Jeff Pepper and Will Toms for their help with STM technology, Cadence and other tools and to Steve Temple who helped with Compass and Powermill.

This research was supported by a teaching assistantship and an Atlas scholarship from the Department of Computer Science whom I sincerely thank.

Chapter 1

Introduction

Power consumption in digital electronics is becoming an ever-increasing concern for circuit designers. Initially the fast-growing demand for portable electronic devices, such as mobile phones, was the main driving force for low power design. Recently, low power has become important even for high-speed desktop and server processors, as the high temperatures they reach start not only to increase the unit cost, because of the more expensive packaging required, but also limit their performance and their lifetime.

Ideally, low-power design techniques improve the power consumption considerably for no, or very little, speed degradation. These are usually new, improved algorithms for implementing functional units or new, power-efficient circuits. Such a technique can be 'hard-wired' in a system almost without a second thought.

Unfortunately, in most cases, power savings come at the cost of speed reduction, so incorporating such power-saving techniques into a system cannot be hard-wired. A better approach is to have the option of enabling them while the system is operating, so that the decision to use them may be made when the system is being used rather than when it is manufactured. This is not useful if the system under consideration is a busy, high-performance processor, which always has computationally demanding tasks to perform as, in this case, the extra delay imposed by those methods cannot be hidden. Instead, these methods can be applied successfully in systems that have idle time, either while waiting for user input or when running applications with soft real time deadlines (e.g. some multimedia applications or communication protocols) [Bur01].

In the latter systems the *slack* time may be better exploited for energy efficiency by slowing down the application rather than running the system at normal

(high) speed and putting it to sleep for the remaining time. In a typical system there are considerable energy and delay penalties associated with changing processor activity state [BBdM00] so, if the slack time is too short for these techniques to be effective, the processor will be busy-waiting and some energy will be wasted during the slack time. Even in an ideal system with no dynamic energy consumption when idling or zero-overhead sleep/restart transitions, allowing the application to use up all of the available time can have energy benefits; for example if the slow-down is combined with a lower supply voltage or lower switched capacitance than normal, e.g. by (micro)architectural modifications.

Currently a number of processors use dynamic voltage scaling (e.g. [BPSB00], [CHM+01], [Fle00]) and a number of architectural adaptations have been proposed (e.g. [MKG98], [BM01b], [KSB02]). All these methods were developed for synchronous processors. Since adaptation is a key issue in this class of low-power techniques, this thesis investigates the use of asynchronous design [SF01] for power saving architectural modifications. Asynchronous circuits do not synchronise all activities with a global clock, so they can be more flexible. This flexibility is exploited by new techniques, such as pipeline occupancy control [EG02a], described here.

1.1 Research goals and contribution

The goal of this research is to develop micro-architectural techniques for asynchronous processors that will enable them to adapt their power consumption efficiently according to their processing workload. The key research contributions are:

- Development of two techniques for controlling the occupancy of an asynchronous pipeline. The first uses an external feedback mechanism for controlling the occupancy, while the other changes the pipeline structure dynamically by collapsing the pipeline latches while the system is operating.

- Development of a new, adaptive, low-power CAM organization, which makes CAM-based caches more power efficient than way-predicting or pseudo associative caches.

- A detailed power analysis of an asynchronous processor and comparison of the results against an equivalent synchronous processor.

- Development of a fast energy estimation method based on a commercial simulator.
- An analysis of the effect of speculation on the energy consumption of an asynchronous single-issue processor.

The following papers have been published presenting some aspects of the work described in this thesis:

A. Efthymiou, J. Garside, "Adaptive Pipeline Depth Control for Processor Power-Management", *Proc. of the International Conference on Computer Design (ICCD)*, pp. 454–457, September 2002.

A. Efthymiou, J. Garside, "An adaptive serial-parallel CAM architecture for low-power cache blocks", *Proc. of the 2002 International Symposium for Low Power Electronics and Design (ISLPED)*, pp. 136–141, August 2002.

A. Efthymiou, J. Garside, "A Comparative Power Analysis of an Asynchronous Processor", *Workshop on Power And Timing Modelling Optimization Simulation (PATMOS)*, September 2001.

S. Furber, A. Efthymiou, J.D. Garside, M.J.G. Lewis, D.W. Lloyd and S. Temple, "Power Management in the AMULET Microprocessors" *IEEE Design and Test of Computers* 18(2):42-52, March-April 2001 (special issue Ed. E. Macii)

S. Furber, A. Efthymiou, M. Singh, "A power-efficient duplex communication system", *Workshop on Asynchronous Interfaces: tools, techniques, and implementations*, pp. 145-150, July 2000.

Some preliminary work was also published in local fora in the UK:

A. Efthymiou and J.D. Garside "Adaptive Pipeline Depth for Asynchronous Systems using Collapsible Latch Controllers". *13th UK Asynchronous Circuits Forum*, Cambridge, Dec. 2002.

A. Efthymiou, "Pipeline Occupancy Control for Power Adaptive Processors". *12th UK Asynchronous Circuits Forum*, London, June 2002.

A. Efthymiou, "Power Analysis of AMULET3". *Future Directions in Low-Power Design Forum*, Manchester, Oct. 2000.

A. Efthymiou, "The Design of a Low-Power Asynchronous Communication System". *9th UK Asynchronous Circuits Forum*, Cambridge, Dec. 2000.

1.2 Thesis organisation

Chapter 2 provides some background material on low-power design. The sources of power consumption in CMOS circuits are described and some of the well-known techniques for low-power design are presented. A brief introduction to asynchronous design and its low power aspects is also included.

The concept of power-adaptive processing is introduced in chapter 3 and the differences compared to power management in conventional processors are identified. Some of the current techniques for adaptive processors are presented, namely dynamic voltage scaling (DVS) and micro-architectural adaptation, with emphasis on the latter since this is where this thesis aims to make a contribution.

Chapter 4 presents the power analysis of an asynchronous processor, namely AMULET3 [GFC99], which was developed by other members of the AMULET research group, at the University of Manchester. The same processor is used as a basis for the techniques presented later, so its organisation is briefly described. A detailed breakdown of the power consumption in the processor core is presented.

A number of techniques that can enable an asynchronous processor to be power-adaptive, by dynamically changing key parts of its micro-architecture are presented in chapter 5. The emphasis is on controlling the pipeline occupancy as a way to control the processor speculation. The experimental setup that supported the simulations of these techniques is also described in this chapter.

Chapter 6 describes in more detail the techniques presented in chapter 5, providing circuit implementations for them.

Chapter 7 builds on the techniques presented in chapter 5 to adapt dynamically the processor pipeline. Two hardware methods are developed and evaluated which improve the speed of the adaptive processor compared to an unpipelined configuration using the techniques of chapter 5 statically.

As an initial attempt to provide power adaptation for the memory system, chapter 8 describes a low power cache implementation based on an adaptive CAM organisation. The CAM is capable of operating in two modes allowing energy-speed trade-offs. The detailed, transistor-level circuits are presented and an evaluation based on Spice-like simulation is given.

Finally, chapter 9 provides the concluding remarks and suggestions for future work.

Chapter 2

Low-power design overview

This chapter presents an overview of power dissipation in digital CMOS circuits and of the basic low-power design techniques. In addition the relationship of power dissipation to energy consumption and circuit delay is presented, to provide metrics for evaluating the efficiency of circuits in a low-power environment. At the end of the chapter a brief introduction to asynchronous design is given and the advantages it offers for low-power design are discussed.

2.1 Power dissipation in CMOS circuits

There are four sources of power dissipation in CMOS circuits [CB95a]:

static Static power is consumed when there is a continuous flow of current from the supply terminal to ground.

leakage Leakage power is due to leakage currents from reverse-biased diode junctions and transistor subthreshold conduction.

short circuit Short circuit power is due to current flowing from the supply to the ground for the short time when both the pull-up and pull-down transistor stacks are conducting.

switching Switching power is due to the current used for charging and discharging capacitances in the circuit when nodes switch logic states.

Switching and short circuit power are collectively called dynamic power, because they are related to currents that flow when transitions are made.

2.1.1 Static power

Static power is consumed in analogue circuits and in ratioed circuits (e.g. pseudo NMOS), where some transistors are not completely turned off. Design styles that draw static current are avoided in circuits intended for low-power applications, therefore this source of power consumption is regarded as a design error in the context of this work.

2.1.2 Leakage power

In the most common CMOS technology (bulk CMOS) transistors form diodes at their source and drain terminals with the well in which they are made, or the substrate below them. These diodes operate in reverse bias and have a leakage current that is proportional to their area. The reverse biased diode leakage current depends heavily on the temperature and, as a rule of thumb, it doubles for every 10°C degree increase in temperature [Tsi87]. Similarly, the wells also form reverse-biased diodes with the substrate causing further current leakage.

The other component of leakage power is due to current through transistors that are supposed to be 'turned off', since the gate to source voltage, V_{gs}, is below the threshold voltage, V_t, but in reality they are still conducting when V_{gs} is above the weak inversion point [Tsi87].

Although currently leakage power is not a significant part of the total power dissipation when the system is operating, it is very significant when the system is in a 'stand-by' state. The problem is that the system is consuming energy even though it does not appear to do any useful work. This mainly affects the usability of portable, battery operated devices.

In future technologies, as the supply voltage is scaled down, transistor threshold voltages are also scaled down in order to retain the circuit speed. This leads to increased leakage power, which becomes significant even when a system is operating.

2.1.3 Short circuit power

Short circuit power is similar to static power, but it is considered a dynamic effect because it occurs only when an output switches. It does not appear in all types of circuits: carefully designed dynamic circuits, without feed-back elements, do not consume any short circuit power, because the P and N transistor stacks are never

conducting at the same time. The same is true for static circuits operating at a supply voltage lower than the sum of the two transistor type threshold voltages.

Short circuit power becomes important when the input rise/fall times are longer than the output transition times. Designing a gate with slow output transition times, to minimise its short circuit power consumption, does not solve the problem as it will cause higher short-circuit power consumption in the fanout gates. The best solution is to try to have equal transition times for both input and output transitions. When this condition is observed, the short circuit power is typically less than 10% of the total [Vee84].

2.1.4 Switching power

Switching power is the major contributor to the power consumption of digital CMOS circuits. The switching power formula for a gate driving its output at full voltage rail is

$$P = \alpha_{0\to1} C_L V_{dd}^2 f_{clk} \tag{2.1}$$

where C_L is the switched capacitance, V_{dd} is the supply voltage, f_{clk} is the clock frequency, and $\alpha_{0\to1}$ is the output node transition activity factor, which is the average number of low to high transitions of the output node in a cycle. The total switching power of a system is found by calculating the above formula for each node and summing the results:

$$P = \sum \alpha_{0\to1} C_L V_{dd}^2 f_{clk} = C_{eff} V_{dd}^2 f_{clk} \tag{2.2}$$

where C_{eff}, the effective capacitance, is the sum of the products of transition activity and capacitance for every node in the circuit.

The activity factor $\alpha_{0\to1}$ depends on a number of parameters. The logic function implemented by the gate is one of them. For example the output of a two input NOR gate can only be 1 when both inputs are 0, so, assuming all possible input values are of the same probability, the 0 to 1 activity factor is $3/4 \times 1/4 = 3/16$. For an XOR gate it would be 1/4. Another factor is the probability of the inputs taking specific values, as this would affect the output 0 to 1 transition probability as computed above. Finally, the logic style (static, dynamic logic, dual-rail) in which the gate is implemented plays an important role in determining the activity factor. The above examples of the NOR and XOR gate assume an implementation in static logic. For an N-tree domino logic,

the probability of an output transition of 0 to 1 is equal to the probability of the output evaluating to 0, because the output will be set to 1 anyway, during the precharge phase. Thus a two input dynamic NOR gate will have an activity factor of 3/4. Dual-rail dynamic logic (e.g. CVSL [HGDT84]) generates both the required logic function and its inverse, so one of the two outputs will always be 0 when the gate is evaluated and thus the activity factor will be 1 for all the logic gates of the family.

The above discussion assumes that all inputs arrive at the same time at a gate. In practice this does not always happen. Consequently the gate, upon receiving the first input change, may switch its output only to switch it back again when another input arrives. These unwanted transitions, due to the difference in the arrival times of the inputs, are called glitches. They waste energy which can be a significant part of the total, 15% - 20% reported by [BFR94]. Glitches cannot happen in dynamic logic circuits because they can only have one transition when they are evaluated.

The rest of the thesis ignores all other sources of power consumption except switching power, as this is currently the dominant factor in a CMOS digital system.

2.2 Energy, power and speed

Even in a low-power system, performance is still important. There is no practical use for a processor that has a very low power consumption, but is too slow to run any useful application. Furthermore, in most systems, it is low energy rather than low power that is the design target. Portable electronic devices operate on batteries, which have a limited energy (or charge) storage. For such devices, the aim is to be able to do as much work as possible with the available *energy*.

In order to assess different circuit implementations, it is useful to have a metric, so that power/energy efficiency can be measured quantitatively. Obviously, using power as a metric is not adequate because it does not take delay into account. Similarly using energy as a metric, usually measured in Joules/instruction or its inverse MIPS/W, also has drawbacks. Energy is the product of the average power dissipation and the time to complete an operation. So a processor A, that dissipates half the power of B but completes a task in double the time, consumes the same energy. Thus, although the faster processor B is better in this case,

using energy as a metric shows that they have the same efficiency.

Burd *et al.* [BB96] have proposed *energy to throughput ratio* (ETR), which can be extended to include the processor idle energy. This is useful for processors operating in "burst mode" which are mostly idle and spend only a fraction of their time performing computations, but when computation is demanded, the faster they complete it, the better. Unfortunately this metric requires many parameters so it has not been adopted by the low-power design community.

One of the most widely used metrics is *energy delay product* [GH96], which is usually expressed (inversed) in MIPS^2/W. This metric correctly shows processor B from the example above to be the most efficient. A similar metric of $E \times D^2$ has been proposed [PM02] and later generalised as $E \times D^n$ in a number of independent publications [ZS02], [Hof02]. This metric allows flexibility in setting energy-delay trade offs; depending on the importance of the delay increase caused by an energy reduction technique, an appropriate value of n is selected.

2.3 Basic low-power design techniques

From the switching power equation 2.2, it is clear that a low-power design should try to minimise the supply voltage, the effective switched capacitance (i.e. the switching activity and the node capacitances) and the operating frequency. Reducing the operating frequency without affecting any of the other parameters does not produce any energy savings; it just delays the execution by the same factor as it reduces the power dissipation. Since the focus here is on low-energy, operating frequency reduction on its own is not considered.

The following sub-sections describe the most common design techniques, organised according to the parameter minimised.

2.3.1 Voltage scaling

From the switching power equation 2.2 it is evident that the best way to make CMOS digital circuits dissipate less power is to lower their supply voltage, because of its quadratic contribution. The drawback is that a lower supply voltage means slower circuits. An approximation of the delay through a logic gate is given by [CB95b]:

$$T_{delay} = \frac{C_L \cdot V_{dd}}{I_{avg}} \approx \frac{C_L \cdot V_{dd}}{k(W/L)(V_{dd} - V_t)^2} \tag{2.3}$$

where k is a technology constant, W, L are the transistor dimensions and V_t is the transistor threshold. From the equation, it is evident that as the supply voltage approaches V_t, the delay rises rapidly.

Equation 2.3 is not totally accurate for contemporary, sub-micron, velocity-saturated technologies, but it is sufficient to show the trend. Using high-accuracy HSpice [Met96] simulations of a fanout-four inverter, figure 2.1 shows the normalised delay and power with a range of supply voltages. It is clear that as the supply voltage approaches V_t, the delay increases rapidly. The graph is produced by simulating an inverter with a fan-out of four identical inverters in a 0.35μm bulk CMOS technology with a nominal supply voltage of 3.3V.

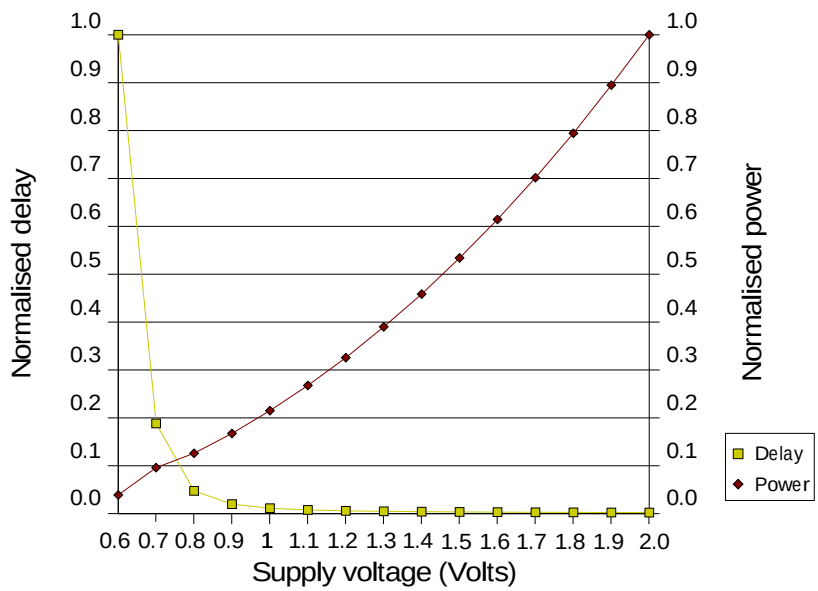


Figure 2.1: Normalised delay and power versus supply voltage.

Technology considerations for voltage scaling

From the discussion above it can be concluded that in order to save power and not suffer from a large increase in delay, supply voltage scaling should be combined with threshold voltage (V_t) scaling. Unfortunately this is not as straightforward as it appears. Firstly a small decrease in threshold voltage causes a significant increase in leakage current through the transistors [GGH97]. Secondly the transistor thresholds are not easily controlled during fabrication, so the actual values

have significant variation. This variation in turn causes a variation in their driving currents and speed, which is greatly exaggerated as the supply voltage is lowered [GGH97]. As a result the supply voltage is usually set no lower than 4–5 times the nominal threshold voltage.

The semiconductor technology imposes limits to the supply voltage [CB95a]. A relatively high supply voltage creates a high electric field under the transistor gate. This attracts more carriers in the channel, increasing the transistor drive and thus the circuit speed. Unfortunately, the high electric field degrades the devices possibly leading to failure. As a result an upper-limit to the supply voltage is set for reasons of reliability. With shrinking device sizes, the gate oxide is becoming thinner, lowering this upper-limit.

Another supply voltage *practical* upper limit is due to velocity saturation in sub-micron technologies. Velocity saturation limits the maximum current drive of the transistors and thus sets a 'minimum' supply voltage above which any increase does not affect the delay significantly [CB95a].

With the continued shrinking in CMOS technologies, the nominal supply voltage is getting lower because of the above limits. Thus some power savings come without any effort from the designer! The remainder of this section describes voltage scaling methods that lower the supply voltage to achieve further power and energy reduction.

Architecture-driven voltage scaling

As explained previously, a low supply voltage reduces the speed, so architectural modifications can be employed to win back the lost performance. This is generally achieved through parallelism: either by deeper pipelining or by using (more) functional units in parallel. The performance loss because of the lower throughput of each unit is recovered by having more of them operating in parallel. This technique, termed "architecture-driven voltage scaling" [CB95a] has been very successful in the digital signal processing (DSP) world [CBB94] due to the fixed throughput of DSP computations. Naturally there is a significant area cost in this solution, but area is often not as important as performance and power consumption.

In ideal conditions[1], a single unit operating at voltage V_r can be replaced with

[1] $V_t = 0$, no leakage power, and no energy consumption in the circuits distributing the data to the parallel units.

N units operating at V_r/N. The aggregate throughput will be the same and the parallel implementation will be consuming $1/N^2$ times the power of the original [CB95a]. These power savings would come at a cost of increasing the area N times.

In systems with irregular throughput, like microprocessors, architecture-driven voltage scaling cannot be used effectively. Using multiple processors, each operating at a low voltage, has a high synchronisation overhead. Moreover, not all tasks can be made to work in parallel. Those that cannot will be the speed bottleneck of the whole system.

A potential problem arises from taking architecture-driven voltage scaling to the extreme in systems composed of several modules which have different optimal supply voltages, as in a "system on a chip". If each module has a different 'optimal' supply voltage, a very high cost in area, both on-chip and on the printed circuit board, has to be paid to supply and distribute the different voltages. For practical reasons only a few, if not just one, supply voltages will be provided. As a consequence most of the components will operate at a higher than their optimal voltage and thus the system's power consumption will be higher than optimal.

Reduced voltage swing

The switching power equation 2.2 only applies to circuits that drive their signals at full swing from the supply voltage to the ground. When the voltage swing of a signal is V_s the power consumption of the gate driving it is $P = \alpha_{0\to1} C_L V_{dd} V_s f_{clk}$. Thus by reducing the voltage swing, power and energy can be saved in a linear fashion.

Using reduced voltage swings for signals reduces the noise margins which could lead to low yield and reliability issues. Thus this technique is used in highly controlled, full-custom circuits, where noise is accounted for and some margins are provided.

A low-swing signal cannot be connected directly to a 'common' logic gate as it will keep the transistors it drives continuously conducting, thus causing static, short-circuit currents. Special circuits are required to 'restore' the signal to full swing, before it can be used as an input to other circuits.

Low swing is usually employed when driving long wires with high capacitance, where both power and delay can be reduced by its application. A typical example is a memory array where sense amplifiers are used to amplify the voltage difference

on the bit-lines which are weakly driven by the memory cells that are being read. Low power memories stop discharging the bit-lines when the sense amplifiers have triggered which limits the voltage swing on the bit-lines to the lowest possible [AH98]. Another common application is on main system buses as exploited by a number of designs [ZGR00].

2.3.2 Effective switched capacitance reduction

In order to reduce the effective switched capacitance, the transition activity and/or the circuit node capacitances must be reduced. Reducing the capacitance of a node also improves the time required for its driving gate to switch the node voltage and thus, in the long run, improves the circuit speed.

Capacitance reduction

Figure 2.2 shows a 'typical' node in a CMOS circuit; the inverters are shown for simplicity, each one can be any logic gate. The node capacitance in fig. 2.2 has been grouped in three lumped capacitances. C_o is the output capacitance of the gate that drives the signal. It is formed by the drain to bulk and drain to gate capacitances of the transistors connected to the output. C_w is the capacitance of the interconnect. C_i is the total capacitance of the transistor gates the signal is connected to.

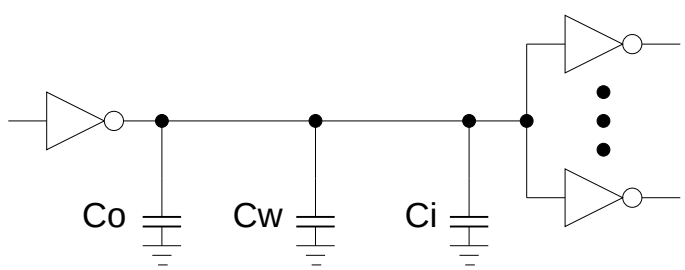


Figure 2.2: Breakdown of node capacitance.

Generally C_o is the smallest and depends on the transistor sizes of the driving gate. The latter are sized according to the driving current required to (dis)charge the node in a 'reasonable' time, which obviously depends on C_i and C_w. Thus reducing C_i and C_w will also reduce C_o.

C_i, in addition to technology (e.g. gate oxide thickness), depends on the number of logic gates driven and the sizes of the transistors controlled by the signal.

Thus circuits with low fanout generally help to reduce the power consumption. Transistor sizes depend on the load the gate drives, so minimizing them is a matter of careful engineering practice[2]. Logical effort [SSH99], [SL01] is a method for transistor sizing primarily focused on high-speed, while Borah *et al.* [BOI96] focus on power consumption — including short-circuit power — with delay constraints.

C_w depends on the actual distance between the gate generating the signal and those using it. In addition, the more gates a signal is an input to, the more interconnect capacitance it has. Careful routing is the best way to reduce C_w but, typically, thousands of nets must be routed, so the attention each net receives depends on the available design time.

In summary, the capacitance depends primarily on the technology and the ability of the designer and the CAD tools to produce a good layout. When designing the circuits the only optimisation is to keep the fanout of the gates as small as possible.

Switching activity

A large number of factors affect the switching activity of a circuit, ranging from the algorithm that is implemented and the data representation to micro-architecture and circuit style.

At the highest level, the operations that make up the algorithm are very important. For example $\cos(x)$ can be calculated approximately as $1 - x^2/2 + x^4/24$ [BdM00]. By slightly relaxing the accuracy, it can be calculated as $1 - x^2/2 + x^4/32$ where the last division is avoided, as it can be implemented with a right shift. Generally the fewer (and simpler) the operations, the less the activity, thus optimising the steps of an algorithm is very important.

One way to minimise the transition activity is to encode the data appropriately for computation, storage, or for transmission.

Various encoding techniques have been investigated especially for buses and inter-chip communication, where the wires have very high capacitances [SB97], [AS00], [HL01]. For example, in Gray code, sequential numbers cause a single bit transition so it is ideally suited for encoding address buses. In less predictable situations, bus-invert coding [SB95] can save 25% of the average power consumption on buses at the cost of an added signal called *invert*. When the Hamming

[2]When using standard cells there are gates with fixed sizes so it may not be possible to size the transistors optimally. Moreover, this could be a slow process affecting the design time.

distance between the current and next bus values is larger than half the bus bit-width, the next bus value is transmitted inverted and the *invert* line is set. One recent communication protocol in the asynchronous framework [FES00] uses optimal delay insensitive codes (3 out of 6) [Ver88] for transmitting information with as few transitions as possible.

Data representation also affects switching activity in memories. Chang *et al.* [CPK99] propose two methods for memory structures with single bitline evaluation, such as ROMs and multi-ported SRAMs (e.g. processor register files). A more advanced method [VZA00] compresses bytes containing all zeros to single bits for both reading and writing, giving a data cache energy reduction of 26% for a 9% size increase. Moshnyaga *et al.* [MIF02] show a 20% energy reduction in video memory by not writing or reading the MSBs that are not different. The advantage of this method is that it does not depend on the presence of a large number of zeros as do the other two methods.

Two's complement representation is usually used for numbers in digital systems because the basic arithmetic operations are easily performed with this representation. In digital signal processing (DSP), where the processed numbers come from sampled analogue signals that switch frequently around zero and don't utilise their entire bit width, it is well known that sign-magnitude is a better representation [CB95b]. When a two's complement number changes sign, a large number of the more significant bits switch; e.g. +1 to -1 in 32-bit representation causes 31 transitions. On the other hand a number represented in sign-magnitude undergoes only a single bit change in the above example.

Micro-architecture, i.e. resource time-sharing, sequencing of operations etc., is also one of the principal issues affecting the effective switched capacitance, i.e. both the switching activity and the node capacitances. For example, choosing to share a bus amongst a number of units saves silicon area — thus reducing capacitance — while at the same time increases the switching activity in the shared bus. All such decisions have to be made with careful consideration of all these factors.

A common issue is 'guarding' the inputs of a unit when it is not in use [DT99]. It is quite common to connect units performing different operations to the same buses which provide the input data. When the inputs of such a unit are not 'disconnected', whenever new data appear on the buses a result will be produced,

consuming energy, even though the result is not needed. Two options are generally available: inserting latches at the inputs that are only loaded when the result of the unit is needed, or inserting some simple logic gate, e.g. AND, OR, that will convert the inputs to values that will not cause internal transitions. The energy consumption of the added circuits and of the signal controlling them should be lower than the energy they save.

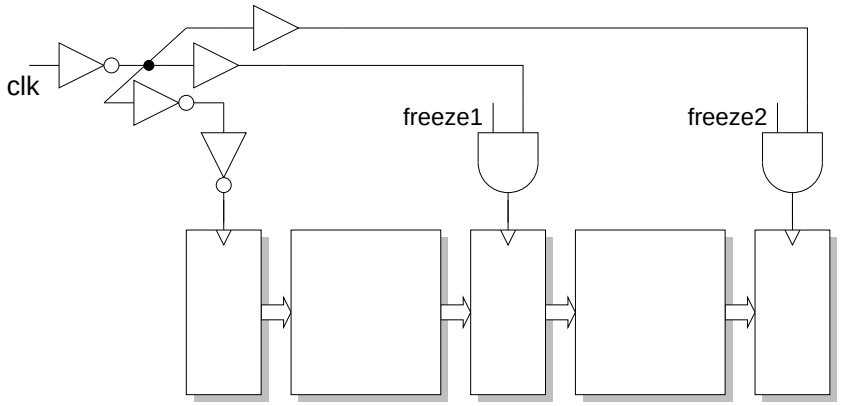
In a synchronous circuit, 'clock-gating' — conditional disabling the clock input to latches or dynamic circuits — is usually employed for the above situation at the pipeline stage level. The clock can be gated either because there is no useful operation to be performed in the current cycle (as determined by a control unit) or, more aggressively, when the data are such that they will produce no new result in all, or part of, the bit-width [BM00]. This technique poses many design challenges because it makes it harder to control the clock skew and to test and verify the circuit and it exaggerates the power supply noise problem due to large differences in current from cycle to cycle ($L \times d_i/d_t$) [BBS+00]. Despite the problems, clock-gating has been used in microprocessors for several years [GBJ98].

Another important issue in minimising the transition activity is avoiding glitches. Although there are logic families that are glitch free (e.g. domino logic), they may not be advantageous over non glitch-free families because of their higher inherent transition probabilities. In addition the constraints imposed by these logic families may lead to less power efficient circuits than static CMOS. A lot of methods have been published targeting this problem either from the CAD tools approach [RDJ99] or the circuit design [BdMM+00].

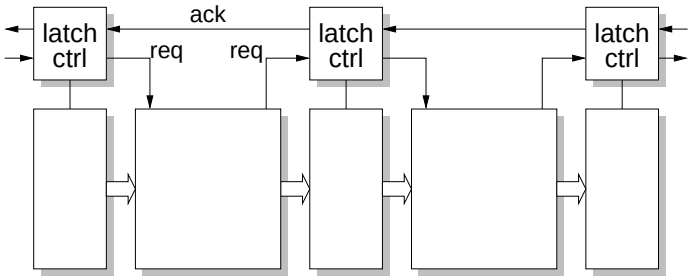
2.4 Asynchronous design

The clock signal in a synchronous circuit serves as a global timing reference for communicating data among the different units in a system. The data communication is through (pipeline) latches, which use the clock as a 'load-enable' signal.

In an asynchronous system there is no notion of a 'clock' and all communication is explicit using a *channel* that follows a *handshake protocol*: the sender, after making the data available, sends a *request* signal notifying the receiver, which processes (or stores) the data and sends an *acknowledgement* back when it has finished. When two units are separated by a pipeline latch, there is a channel



(a) Synchronous pipeline.



(b) Asynchronous pipeline.

Figure 2.3: Synchronous and asynchronous pipelines.

on both sides of the latch. The signal protocol of the channels and the latch are coordinated by a unit, usually implemented as a separate block, called a *latch controller*.

Figure 2.3 shows a simple pipeline in a synchronous and asynchronous implementation [OS02] where the equivalence of the latch controllers with the local clock drivers, using clock-gating signals (like *freeze*), is apparent.

2.4.1 Asynchronous design and power consumption

In a contemporary synchronous processor a large proportion of the power is attributed to the clock. Tiwari *et al.* [TSR+98] report that about 40% of CPU power is spent on the clock, including the generator, drivers, distribution tree and loading. StrongARM [MWA+96] — with a similar architecture to AMULET3 [FEG00] that was developed in the University of Manchester — is reported to

use 26% of its power in the clock, including the PLL, while in its successor, the XScale, this figure drops slightly to 23% [CHM+01].

The clock equivalent in an asynchronous processor is the set of latch controllers including the drivers for the large latch-enable loads and some precharge signal drivers which are controlled by handshake signals. Based on the power analysis presented in chapter 4, the contribution of all these circuits to the power consumption of AMULET3 was found to be only 10.5% of the core while running Dhrystone 2.1 [EGT01]. Comparing this with the proportion of the power taken by the clock shows that asynchronous techniques have the potential to reduce power consumption significantly.

Due to the implicit synchronisation by the clock, the units in a clocked design have to exchange data in every cycle even though there is nothing new (or valid) to send. Even with aggressive clock-gating, some information will have to be passed from one pipeline stage to the next — e.g. the information whether the data sent are valid — which will cause some activity. Asynchronous circuits, on the other hand, perform operations on request so they do not consume dynamic power when there is no computation. Depending on the specifics of the implementation, the above applies to the whole range from a processor to arbitrarily small sub-circuits.

In addition to the 'no useful work–no power consumption' property of asynchronous circuits, another related advantage becomes apparent when returning to normal operation from a 'deep sleep' mode. As there is no clock generator to restart, an asynchronous processor can 'wake-up' and start executing instructions immediately. In contrast a synchronous processor typically suffers a delay of a few microseconds [MWA+96].

The ability of an asynchronous pipeline stage to have variable delays can be exploited for power savings. In a synchronous system, a unit that is infrequently used will still have to be designed so that it meets the clock cycle target, otherwise all operations will be delayed. This could mean that an implementation which is not power efficient must be used. In an asynchronous system a power efficient, but slower, implementation of the unit can easily be incorporated: the delay through the pipeline stage will be longer only when this unit is used. If the unit is infrequently used the performance loss will be very small.

Unfortunately asynchronous design is not free of disadvantages. The requirement of explicit communication makes some operations difficult in asynchronous

design, such as communicating with non-neighbouring pipeline stages. Sometimes the communication can be avoided altogether by duplicating information in many parts of the pipeline. In any case, the duplicated data and the extra communication will increase the power consumption, of at least the control part, of the system.

Another disadvantage in power consumption of asynchronous circuit comes from the practical problem of lack of CAD tool support. This leaves more design steps to be done manually. A less automated design is much more likely to be less optimised.

2.4.2 Classification of asynchronous circuits

The above discussion presented the general advantages and disadvantages, in power consumption, of asynchronous circuits in comparison to synchronous circuits. These properties are common to every class of asynchronous circuits. In this section some types of asynchronous circuits are covered and their suitability for low power design is discussed.

Handshake signalling

The signals used in the handshake protocols are used to communicate events: e.g. the availability of new data and the acknowledgement of the reception of data. As signals use two-level Boolean values, there are two transitions for each wire that can be used as an event: the positive edge and the negative edge. When both edges are used to mean the same event, the handshake protocol is called *non-return-to-zero* (NRTZ) [Sei80]. When only one edge (usually the positive) represents the event and the other is used to return to a quiescent state, the handshake protocol is called *return-to-zero* (RTZ).

Although from the power consumption point of view NRTZ seems preferable since it reduces the number of transitions, the circuits implementing these handshake protocols are more complex than those implementing RTZ. Currently almost all asynchronous circuits use RTZ for this reason [SNNS93], [FDG+94], [FGT+97]. NRTZ signaling would be useful when the channel wires have large capacitances, for example when the two communicating parts are in different ICs, as in [FES00]. Since this is rarely the case, for the remainder of this section RTZ signaling is assumed.

Delay-insensitive codes

In a handshake protocol the indication of the validity of the data can be inherent in the data representation. Such representations are called *delay-insensitive* (DI) codes because there is no need for a separate valid or request signal to signify that the data is ready. When such a signal is used it must be guaranteed to be asserted *after* the data is ready, which is impossible in a delay-insensitive model where the delays through both gates and wires are unbounded.

Verhoeff [Ver88] describes the theory and a number of DI codes. The most common codes used are *one-hot*, usually found as 1-of-2 (*dual-rail*) or 1-of-4.

In dual-rail each bit (d) is represented by two wires: one representing logic '1' ($d.t$) and the other representing logic '0' ($d.f$). Depending on the actual value of the bit, only one of the wires is asserted at each time (table 2.1). Using return-to-zero signaling, when a series of data are transmitted, $d.t$, $d.f$ must go through the empty code in between valid data transmissions.

Table 2.1: Dual-rail code

$d.t$	$d.f$	
0	0	empty, data is not valid
0	1	valid 0
1	0	valid 1
1	1	not used, illegal.

In one-hot encoding, n-bits are encoded in 2^n wires, each representing one of the 2^n possible values. Since a very high number of wires is needed, such codes are rarely used for more than 2 bits in a datapath circuit. Groups of more bits can be split into pairs and 1-of-4 codes can be used for each pair.

In comparison both dual-rail and 1-of-4 require the same number of wires to represent the same number of bits. Considering the number of transitions though, 1-of-4 uses two transitions, including the return to zero, for sending 2 bits, while dual-rail would require four. Thus for power consumption 1-of-4 is lower, while keeping the silicon area (number of wires) the same. This was shown to be true in practice in a case study by Lloyd *et al.* [LG01].

Bundled-data

The simplest way to implement a channel is to have the data represented in the 'common' binary, or single-rail, format and *bundled* with a *request* signal. When

request is asserted, the data are ready to be used by the receiver. This means that the data must arrive at the receiver (shortly) before the request signal, which means that there is a timing dependence that must be verified/implemented by the designer. In this respect DI codes are superior, since they do not require such dependencies and the design effort implied by them.

The greatest advantage of bundled-data representation is that they require fewer wires to encode the same number of bits compared to delay-insensitive representation. This saves silicon area and, consequently, decreases the average node capacitance.

In RTZ signaling of bundled-data handshakes, only the request signal need return to zero. Thus, in comparison to DI codes where the data wires will have to return to the empty code, the bundled-data style offers a lower switching activity.

From the above discussion it is clear that bundled-data is more suitable for low-power design, at the expense of greater design effort.

Chapter 3

Power adaptive processors

The previous chapter concentrated on techniques that can be used to reduce energy consumption without sacrificing performance. This chapter discusses adaptive techniques that enable execution time to be traded for energy consumption. The increase in time does not necessarily affect performance, as perceived by the user, in many applications for example in interactive tasks or those with soft deadlines. Moreover, the user may prefer to trade some performance or quality of service for extended battery life.

The first section defines power adaptability and discusses which parameters can be adapted. Section 3.2 describes power management techniques in conventional processors, which can be used as a 'poor man's' alternative to power-adaptive processors. Dynamic Voltage Scaling (DVS), which has recently been incorporated into commercial processors, is examined in section 3.3. The remainder of this chapter considers basic architectural adaptation techniques.

3.1 Power adaptability

Power adaptability is the ability of a processor to scale its power consumption according to the 'intensity' of the work it performs and/or the operating conditions. The same concept is also called *power awareness* in part of the literature [BMC01].

Intuitively, power adaptability seems to be a natural property for a processor to have, but only recently has it started to be considered in depth. To some extent, even processors which are not designed with energy efficiency as a goal exhibit some power adaptability, especially when they are executing 'easy' instructions

(e.g. register to register move), or computing with 'easy' data (e.g. adding with zeros). But a processor designed with power adaptability in mind should be able to achieve much greater energy savings.

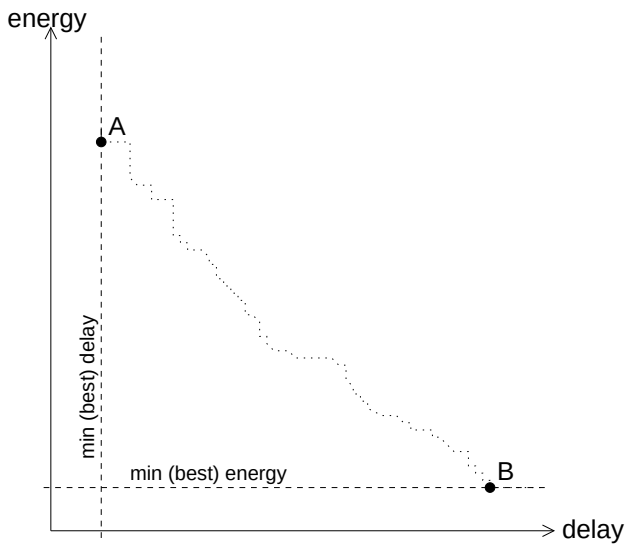


Figure 3.1: Energy-Delay plot of an adaptive processor.

Figure 3.1 shows how the energy consumption of an adaptive processor could relate to the execution delay. There is a minimum energy consumption (B) and a minimum execution delay (A) when the processor executes a given program. Increasing the delay further than B does not reduce the energy consumption, but it could reduce the *power* dissipation. On the other hand, more energy can be used, e.g. by using more parallelism, without any speed improvement above A. In between the two extremes an adaptive processor can offer energy-delay trade-offs, either continuous, as in the plot, or discrete points.

In a changing operating environment, power adaptability can also be viewed as robustness: the processor is operating as fast as it can for the current conditions. An example of such an environment is a contact-less smart-card [ABR+01]. In these smart-cards there is no physical connection to the card reader and the processor is powered from the ambient radio field generated by the reader. The available power — and supply voltage — fluctuates according to the distance from the reader, requiring the processor to be able to operate continuously in a range of voltages.

3.1.1 Motivation

Bhardwaj *et al.* [BMC01] mention two motivating views for having power awareness:

- The user is allowed to make the trade-off between energy and speed or 'quality' in general (e.g. quality of video or sound). Each user has their own preferences, which probably change even in the time between battery recharges, and the devices should be able to satisfy these preferences. A system designed to be power-adaptive is probably the best way to implement these trade-offs.

- There is significant variance in the workload of many applications which could be exploited automatically by the system for lower energy consumption. The user need not be aware of this function which should either cause no 'quality reduction', or the reduction should be within a user specified level.

In addition to the above motivating views, a power-adaptive system can be used for thermal management [BM01c]. With higher levels of integration, heat dissipation is becoming an important concern, especially for high-end processors where, in the near future, the performance may be limited because of heat dissipation [BM01c]. Power adaptivity can be used as a means to reduce the temperature of a processor when it is approaching a hazardous limit, while maintaining operation, albeit at a reduced speed [HRYT00].

Many application programs exhibit significant variance in their execution time, depending on the data being processed or, for interactive applications, the slow — for a processor — response time of the user [PBB00], [HKA+01], [HRYT00]. For example Pering *et al.* [PBB00] take advantage of the fact that a user cannot notice any screen update faster than 50ms, in their voltage scaled processor. Hughes *et al.* [HKA+01] report an execution time variability with a range of 37% to 195%, relative to the mean execution time, for a set of speech, video and audio codecs at the frame granularity. Finally, Huang *et al.* [HRYT00] argue that not only multimedia and interactive applications show variability, but also general purpose ones, especially when they depend on network operations, e.g. web-servers.

3.1.2 Categories

As explained in the previous chapter there are three 'variables' that can be changed to affect speed and energy consumption in a processor:

Time There are two cases depending on the timing paradigm used:

- **Synchronous** The execution time can easily be changed by changing the clock frequency. Power dissipation will scale with execution time, but the energy to execute a task will be largely constant.
- **Asynchronous** Since there is no clock the circuits work as fast as they can. The only way to slow down is to deliberately add delays, using long inverter chains for example. Power should scale with execution time as with the synchronous processors, but the delay elements will consume energy, so this scaling will be worse than linear: in the plot of figure 3.1, to the right of point B the curve will rise gently instead of being parallel to the axis.

Supply voltage Dynamic voltage scaling (DVS) [NNSvB94] is the most commonly used method to make an adaptive system. As the dynamic energy consumption of a system depends on the square of the supply voltage, a small decrease in voltage would yield big savings in energy consumption. On the other hand circuits are slower at a lower voltage, so the speed will drop and so will the power dissipation. Although for an asynchronous circuit the speed loss is handled automatically, for a synchronous one the clock frequency must be adapted.

Architecture Changing the architecture affects the effective switched capacitance for the dynamic power and the cycles per instruction (CPI) for speed. It can reduce the energy consumption and usually will slow down the processor, thus the power dissipation will decrease too.

Since adapting the architecture or the supply voltage offers more improvement in energy consumption than adapting execution time on its own, the latter is applied only as a final resort when all other measures have been applied and still lower *power* dissipation is required. Dynamic voltage scaling and adaptive architecture are discussed in this chapter, after a brief discussion of power management for conventional, non-adaptive processors is given in the following section.

3.2 Power management

With a little hardware support, conventional, non power-adaptive processors can also exploit idle time to save energy. Usually, the processor completes the task in its usual operating mode, and then enters a 'sleep mode', which is a condition where the processor does not execute any commands and shuts down some of its activity to save energy.

Most contemporary processors support 'sleep' or 'power-down' modes (e.g. [MWA+96], [GDE+94]). The typical offering is three modes: a normal-operation or *run* mode and two sleep modes. Figure 3.2 (from [BBdM00]) shows the state transition diagram of the power-down states of StrongARM, the power consumption at each state and the delay to switch states. Typically, in idle mode all operations are halted, but the interrupts and the PLL are still enabled, so that the processor can resume operation quickly when an interrupt happens. In StrongARM, during sleep mode the power supply of the internal circuits is turned-off. The I/O circuits are still powered so that the specified logic levels are observed on the external wires.

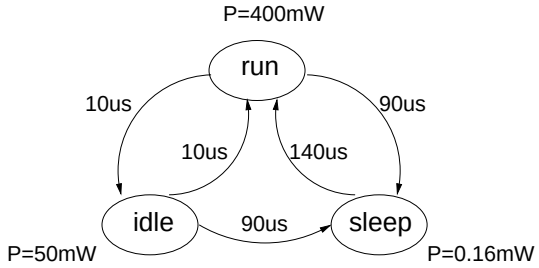


Figure 3.2: Power modes state transition diagram.

When there are no tasks to execute, selecting which power-down mode to switch to and when, is a gamble. The lowest power consuming mode is obviously best for long idle times, but it is hard to predict when a new task is going to require the use of the processor so that the idle time can be predicted efficiently. Benini *et al.* [BBdM00] define 'break-even time' of a power-down mode as the minimum idle time required to compensate the cost of switching to that mode. If the idle time is less than the break-even time, there is no energy benefit from entering the power-down mode; on the contrary, there will be an energy loss. A wide range of algorithms have been reviewed [BBdM00] for deciding when to enter a power-down mode, the simplest of which is the common timeout policy used in most personal computers.

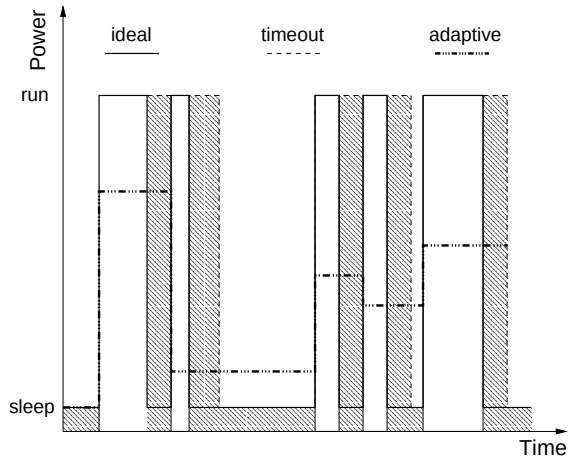


Figure 3.3: Wasted energy due to idle time.

In figure 3.3 the solid line shows the instantaneous power consumption of an ideal power management policy, where the processor can switch to a low-power mode and back with no delay. The dashed line shows how a processor with a timeout power management policy would behave for the same workload. The shaded area represents the wasted energy. The energy consumption while in sleep mode is the low limit of this wasted energy and this is common to both the ideal and the timeout policies. All gray areas above the sleep power level are the energy overhead of the timeout policy, which is quite substantial.

Finally, the possible behaviour of a power-adaptive processor is also shown. From the graphs it is clear that, although using power-management on conventional processors can save some energy, power-adaptive processors can be more efficient by slowing down the execution so that each task finishes just before the next one begins.

3.3 Dynamic voltage scaling

Dynamic voltage scaling (DVS) is a technique that dynamically varies the supply voltage and the clock frequency (for synchronous systems) in response to computational load demands [BPSB00], with the purpose of saving energy while maintaining high speed when it is needed. DVS is different to techniques like architecture driven voltage scaling (described in chapter 2), where the supply voltage is fixed at a (generally low) value, and architecture techniques are used (typically parallelism) to keep the performance at the appropriate level. The

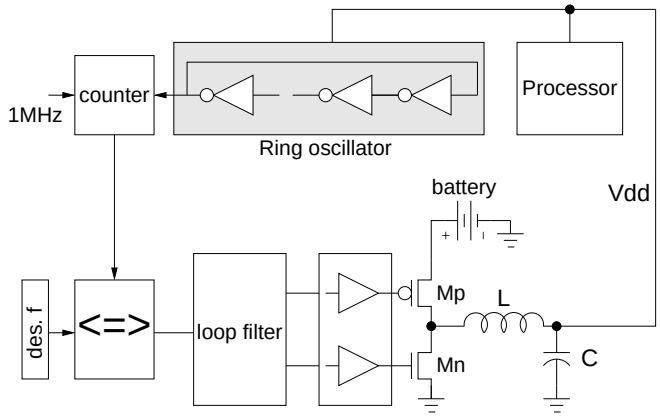


Figure 3.4: Voltage converter and processor connection for DVS.

difference is twofold: the speed and the supply voltage both change dynamically.

A number of recent processors use DVS: lpARM [PBB98], XScale [CHM+01], Crusoe [Fle00]. XScale and Crusoe are commercial processors, so not many details about how they are designed are published. For this reason the description here is based on lpARM [PBB98], [BB00], [BPSB00].

DVS requires carefully designed circuits, a voltage converter, and some support to decide when to switch voltage levels. Each of these elements are described in the following sections.

3.3.1 Hardware requirements

The central part of a DVS system is a regulator consisting of a voltage converter combined with a clock generator. When the processor wants to change its energy-speed level, it sends the requested new clock frequency to the regulator which changes the supply voltage to the lowest possible level that can achieve the clock frequency.

Figure 3.4 shows how the regulator was implemented by Burd *et al.* [BB00] and how it is connected to the processor. The regulator compares the current frequency to the desired one and changes the operating voltage accordingly by pumping charge in or out of a large capacitor (C) that provides the energy supply for the processor. During this process, as the supply voltage changes so does the processor clock frequency and the correcting action is modified until the desired frequency is reached. The ring oscillator generating the processor clock operates from the same supply voltage as the processor and it is designed to track its worst delay path.

The size of the capacitor is critical [BB00]. Increasing C reduces supply ripple and the energy loss of the conversion, but it also increases the transition time (switching between supply voltages) and the energy loss during the transition.

While the supply voltage is changing it is possible to continue the operation if the circuits are designed according to some guidelines [BB00]:

- Static CMOS circuits are very tolerant to supply voltage changing and should be used as much as possible.
- NMOS pass transistors should not be used, because they cannot operate at a supply voltage lower than twice the threshold voltage[1] (V_t).
- Stacked devices with more than 3–4 transistors should be avoided. Their delay cannot be matched easily, at low supply voltages, by an inverter chain that is commonly used in a ring oscillator. This means that the clock generator will not be able to "follow" the critical path.
- Dynamic logic could cause latch-up or false evaluation if the supply voltage changes too fast. This can be avoided by using a PMOS device on the dynamic node controlled by an inverted stage of the output. A similar problem could happen in tri-state buses that can be left undriven for a number of cycles. In this case bus-keepers, a pair of cross-coupled inverters, can be used to retain the bus value and follow any changes in the supply voltage.
- In SRAMs, sense amplifiers that are symmetrical and tolerant of supply voltage changes must be used.

Crusoe [Fle00] operates somewhat differently. When the decision is made to change the supply voltage, the processor is halted for some time (20μs) for the clock generator to switch frequency. Then the processor resumes operation, at the low speed, while the voltage is slowly ramped in a number of steps.

3.3.2 DVS in asynchronous systems

In asynchronous circuits there is no global clock. Thus an asynchronous processor with DVS capability should require a simpler regulator, since there is no need

[1] Assuming equal absolute values of NMOS and PMOS transistor thresholds.

to generate an appropriate clock frequency for the supply voltage level. When the supply voltage changes, the circuits automatically adapt and operate at the maximum speed for the new voltage. Obviously, similar restrictions to those stated above apply to the circuits for voltage scaling to work. The feasibility of DVS in an asynchronous environment was shown by Nielsen *et al.* [NNSvB94], in the early days of low-power design.

Due to the absence of a reference clock, it is harder to associate a supply voltage with a desired speed. In the early work by Nielsen *et al.* [NNSvB94] the speed requirement was to maintain a fixed throughput. The solution was to place two FIFOs around the main asynchronous unit and by measuring the occupancy in the input FIFO, they raised the voltage when it was almost full and lowered it when it was near empty. In this solution the feedback mechanism that sets the supply voltage is based on the actual delays of the circuit. The ring oscillator used in synchronous implementations must be made to match precisely the delay of the critical path, which is not easily achieved, forcing the use of safety margins which reduce the power savings.

For an asynchronous processor it is not clear how an appropriate supply voltage can be estimated from the desired speed. A table of average speeds for a large number of supply voltages could be created experimentally and then used as a reverse look-up table to set the appropriate voltage for a selected speed, but this is not precise and the average speed of the initial measurements may not match the speed of the executed application. Es Salhiene *et al.* [EFR02] have recently proposed a voltage scheduling mechanism for an asynchronous processor [RVR98] but they do not show the mechanism that converts a required speed to a supply voltage.

3.3.3 Voltage scheduling

To make effective use of DVS in a processor a 'voltage scheduler' is used to examine the current and predict the future processor workloads and set the new desired speed and voltage [PBB00]. As this task is quite complex and related to normal task scheduling it is implemented in software as part of the operating system.

The voltage scheduler need not take both time and supply voltage into consideration. In a synchronous DVS processor, setting a target clock frequency automatically generates an appropriate supply voltage. As explained earlier this

can be harder or less accurate in an asynchronous processor. Thus the voltage scheduler only has to decide the desired processor speed.

Program threads need to provide an estimate of how many instructions they are going to execute and a completion deadline. Based on the number of threads and the information they provide, the voltage scheduler calculates the minimum processor clock frequency which should allow all tasks to complete on time. Pering *et al.* [PBB00] include the threads that are not currently runnable in the calculation with the aim to reserve time for them in the future. On the contrary Es Salhiene *et al.* [EFR02] use only the runnable and ready threads for the calculation which could avoid overestimating the processing requirements. The speed setting is reviewed each time a thread is created or completed and at the deadlines.

3.4 Power adaptive micro-architecture

Adapting the (micro-)architecture is equivalent to changing the effective switched capacitance for dynamic power consumption. At the same time performance parameters are also affected, namely the (average) number of cycles per instruction (CPI) and/or the cycle time. As expected, simplifying the micro-architecture usually reduces the power consumption, but increases the CPI.

Although a number of micro-architectural features can be adapted at runtime, they can all be placed in two categories depending on which part they mainly affect: datapath or control. Common datapath techniques, discussed in section 3.4.1, exploit narrow data bit-width or trivial computations [CGS00], [YL02], [BM00], [NS99]. 'Speculation control' [MKG98], [EG02a], 'pipeline balancing' [SKO+97], [BM01b], [KSB02], and 'resource scaling' [IM01], [BM01a] in processors supporting instruction-level parallelism are the main classes of control-based micro-architecture adaptation. These classes are not disjoint, for example most pipeline balancing methods indirectly control the speculation of a processor.

Another way to classify adaptive micro-architecture methods is by how they are initiated.

Some methods can be 'automatic' or 'opportunistic', in a sense that they always try to save energy without external enabling. Methods that can detect energy-saving opportunities without affecting the speed, or when this effect is reasonable, could fall in this category.

Most methods, though, have a significant speed penalty so they cannot be permanently hardwired into the system. They can be selectively enabled by an entity that has more knowledge, such as the operating system, the user, higher-level control logic, etc.

3.4.1 Datapath-based adaptation

Although current processors typically operate on 32-bit data operands, a large number of operands actually use many fewer bits; e.g. 62% of register values are just one byte in the Mediabench suite [CGS00]. This can be exploited in storage and functional units to save energy.

Brooks *et al.* [BM00] propose extending the clock-gating technique from basing its decision on the operation type to also consider the operand values. In their method the datapath width is split into a low (significant) part and an upper part. The upper-part, when the operands are just sign extensions, is gated-off to reduce the switching activity. With this technique they exploit 'narrow-width instructions' to save energy. They claim a 45%–60% power reduction in a 64-bit integer unit, without including the overhead of the detection circuits. A second more aggressive technique they propose is to pack a number of narrow width instructions in a single functional unit to be executed in parallel. They do not give any power consumption results for this technique.

A similar technique was used in an asynchronous, interpolated finite impulse response (IFIR) filter implementation [NS99]. Taking advantage of the asynchronous design flexibility, the 16-bit adder was split into two parts and the most significant was only operated when needed.

To save energy in simple pipelined processors, Canal *et al.* [CGS00] developed 'significance compression'. The least significant byte of a 32-bit value is considered to always contain useful information, but each of the other three bytes is augmented with an extension bit. For each byte which contains just the sign extension of the previous byte, the corresponding extension bit is set. Using this data representation for the cache, the pipeline latches, the registers and the functional units, a number of pipelined implementations are presented, varying from a completely byte-serial implementation to a fully 32-bit parallel with gating for the unused datapath parts. The activity is reduced by around 30%–40% in all cases with an increase in CPI from 24% for the byte serial implementation to 2-6% for the parallel one.

Trivial computations, i.e. computations that can be simplified or where the result is 0, 1, or equal to one of the operands, are targeted by Yi *et al.* [YL02] mostly to improve performance, but it is expected that they can save energy too. They show that even when compiling a program with full optimizations, about 30% of arithmetic operations, which is 12% of all dynamic operations, are trivial. The performance improvement comes solely from the fact that instructions performing trivial computations do not have to wait for both their operands to become ready in order to be issued. This improvement is approximately 8% but, unfortunately, the potential energy savings were not investigated.

3.4.2 Pipeline balancing

One class of control-based micro-architecture adaptation techniques try to balance the processor throughput by throttling its front-end, i.e. instruction fetch and decode.

The earliest such method is called 'instruction cache throttling' [SKO+97] and was used for thermal management in PowerPC processors. In normal operation four instructions are fetched in each cycle if there is a cache hit. When the processor temperature is high, i.e. its power consumption has been high for some time, the rate of instruction delivery is restricted. It was shown that halving the fetch rate reduces the processor speed by 12% while dropping 7°C of the temperature. The power reduction due to I-cache throttling saturates at a value of 2.13W when clock power dominates the other power consumption sources in the processor.

A more recent method by Baniasadi and Moshovos [BM01b] balances the throughput of a processor's front-end (fetch, decode, issue) with that of the back-end, by throttling the instruction-fetch rate. Three methods are used to measure the throughput or 'instruction flow' at the front- and back- ends:

Decode/commit rate (DCR) The rationale is that if there are many more instructions being decoded than committed, speculative (pre)fetching is not successful, so why not stop it. The best results are achieved when fetching stops if 3 times more instructions are decoded than committed during a cycle.

Dependence based (DEP) Using this metric, if there are more dependencies than a specified number, fetching stops for 3 cycles. The idea is that if there

are too many dependencies, few instructions will be executed in parallel, so bringing more from the cache is not likely to improve the performance. The best results are achieved when the threshold is set to half the decoding width.

Adaptive DCR/DEP From the simulations it emerged that, when the commit rate is relatively low, DEP works best, otherwise DCR is better. So this method gets an indication of the commit rate and switches between DEP and DCR accordingly.

Adaptive DCR/DEP is the best of these methods and it reduces the average instruction throughput — used as a first-order energy metric — by 11% and 15% for the fetch and decode stages respectively, while slowing down the processor by about 3.6%

A similar approach is followed by Karkhanis *et al.* [KSB02] by trying to offer 'just in time (JIT) instruction delivery'. With JIT instruction delivery, instructions stay in the issue queue for only a short time and the number of mis-speculated instructions is reduced indirectly. The proposed implementation uses a simple up-down counter that is incremented for each fetched instruction and decremented for each committed one. When the counter exceeds a programmable maximum value, fetching is halted. A simple algorithm sets the maximum value by monitoring the program execution.

One of the novelties of this method is that it reduces stall cycles in the decode and issue queues. Stall cycles can expend energy (in the local clock drivers) by reloading the same values into the pipeline registers. By reducing the number of stall cycles this energy is saved. In the simulation study with only 3% performance degradation, energy savings of 10%, 12% and 40% were achieved in the fetch, decode, and issue queue respectively.

3.4.3 Resource scaling

The goal of resource scaling techniques is to "determine the changing needs of each program and tune processor resources to the program with the aim of reducing power consumption" [BM01a]. The processor resources that are scaled include the number of registers [AIC+01], the issue width [IM01] and the number of active functional units [BM01a].

Extending their earlier work [MBB01], Bahar and Manne [BM01a] target resource scaling by adjusting the pipeline issue and execution capabilities of a processor. They assume a base processor with 8 integer, 4 floating point units and 4 memory ports, split equally into two clusters. In this configuration they define two low-power modes: a 4-wide issue mode, implemented by just disabling one of the two clusters, and a 6-wide issue mode, implemented by disabling both floating point units in one cluster.

A simple finite state machine controls changes between these modes depending on the IPC measured in a sampling window. As the instruction-level parallelism in a program changes, the state machine follows the changes and adjusts the active units in the processor accordingly, saving energy by shutting down the unused units.

The simulation results of the adaptive processor running SPEC95 benchmarks show a power reduction of 10%–23% in the issue queue and 5%–12% in the execution units. The overall energy reduction for the processor is at most 8%, out of a maximum possible of 12%, while the average performance loss is 1% to 2%.

Iyer and Marculescu [IM01] use (some quite expensive) profiling hardware used to identify 'hot-spots' of a program and a power estimation method to trigger changes to the micro-architecture, specifically the issue width. The achieved energy savings per instruction are only up to 8%, while the energy overhead of the profiling hardware seems to be underestimated.

3.4.4 Speculation control

A very important aspect of high-end processors is *speculation*, as it allows the processor to overcome control dependent stalls and extract more instruction-level parallelism, especially when the behaviour of the branches cannot be predicted accurately [HP96]. Speculation is always wasteful of energy for two reasons:

- A percentage of the speculatively fetched instructions are discarded when the prediction upon which they were fetched is wrong. The energy expended on fetching these instructions is wasted.

- Energy is expended at every cycle to support speculation, for example making predictions, keeping information to restore the processor state in case the instructions are mispredicted or an exception is caused.

In an ideal processor where every functional unit is optimally energy efficient, the only energy wasted would be due to speculation. Thus all the control-based micro-architectural techniques described earlier, in effect, control the degree of speculation of a processor.

'Pipeline gating' [MKG98] is the best known speculation control technique. The branch prediction hardware is coupled with a confidence estimation mechanism, which reports on how much confidence there is in each prediction. When there are more than a set number of low confidence branches in the pipeline, the probability that the instructions currently being prefetched will actually be executed is quite low. For this reason the front-end of the processor is stalled until the combined confidence improves, which happens after resolving (some of) the branches at the write back stage. The paper [MKG98] investigates a number of parameters and the best results showed that up to 38% of extra work can be saved at the fetch and decode pipeline stages while the performance is affected by only 1%.

3.5 Comparison of DVS and micro-architectural adaptation

A brief overview of the limitations and strengths of DVS and micro-architectural adaptation is given here. The results of a comparison for thermal management [BM01c] and in multimedia applications [HSA01] are also presented.

3.5.1 Limitations of micro-architectural adaptation

There are three main limitations to micro-architectural adaptation:

- In every architecture there is a finite number of changes that can be made. Thus the most important limitation of micro-architectural adaptation is that it cannot offer a continuous range of energy-delay options, as can supply voltage scaling.

- Since each program uses a processor's resources differently, the same processor configuration could have different effects on the energy consumption and the execution delay for different programs. Thus micro-architectural adaptation must be automatically customised to the executing application.

- As the contribution of effective switched capacitance to energy consumption is linear while the supply voltage has a quadratic effect, micro-architectural adaptation does not generally achieve as much energy savings as DVS.

3.5.2 Limitations of DVS

As explained earlier DVS requires careful design and verification of the circuits that can have their supply voltage changing while they operate. This can significantly increase the design time of a system.

In modern system-on-a-chip (SoC) designs, a large number of blocks are present and each could have different speed and energy requirements. Usually the environment in which the SoC operates provides only one or two supply voltages and all others have to be generated on-chip. If each block requires a separate voltage converter, a large area will be used by these converters and the distribution of power on the chip will be a hard problem. Operating all of the blocks from the same regulator would require that they all operate at the same frequency for the whole range of supply voltages, which is hard to achieve in large ICs and probably suboptimal in most cases.

In addition to these practical problems, a limiting factor of DVS is the relatively slow transition times to change voltage (and frequency) levels. Crusoe [Fle00] requires up to 20μs to change the clock frequency and then it begins scaling the voltage, which takes another 20μs *per step*. It could take up to 300μs for a voltage change. XScale [CHM+01] and lpARM [BPSB00] both need at most 70μs to switch to a different voltage level which is over 5000 cycles for lpARM and much more for XScale. Moreover each transition expends energy which can be up to 4μJ in lpARM [BPSB00] that is equivalent to 712 full-load cycles at the maximum speed.

The high cost of each transition means that the decision to change voltage level must be carefully made. Thus the algorithms used for voltage scheduling must be quite conservative, so that they do not cause voltage switches too frequently.

3.5.3 DVS and micro-architectural adaptation

Brooks and Martonosi [BM01c] compared architectural adaptation with DVS for processor thermal management. In their simulations at a preset temperature level, a sensor causes an interrupt which triggers a 'response mechanism' to reduce

the power consumption and thus the temperature. As response mechanisms they experiment with DVS, frequency scaling, and a number of micro-architectural adaptation methods. In their findings the response delay was much longer for DVS because of the time it takes to change the voltage and resynchronise the clock. Moreover using architectural adaptation as a thermal response mechanism achieves lower performance losses than DVS while managing to keep the temperature below the critical level. From these results it is clear that the slow response times of DVS could affect its usefulness.

It is possible to combine micro-architectural adaptation with DVS for further energy savings than can be achieved by each method alone. Hughes *et al.* [HSA01] present a case study of discrete and continuous DVS combined with resource scaling adaptation using multimedia applications for benchmarks. The applications first run in a 'profiling' phase where, for each type of frame and each processor configuration, the energy per instruction is measured and the results are used to order the configurations according to energy per instruction. The remaining frames are run in the 'adaptation' phase, where the number of instructions for the next frame (of the same type) is predicted and an appropriate configuration and supply voltage is selected so that the frame can be processed at the allowed time with the minimum energy consumption. Depending on the predicted number of instructions, the appropriate configuration and, when DVS is enabled, the appropriate frequency and voltage are selected.

The simulation results show an average energy reduction of 68% to 78% for DVS only and 22% for micro-architectural adaptation only. Combining micro-architectural adaptation with DVS offers an additional improvement of 11% to 17% relative to DVS alone. This result shows that micro-architectural adaptation can be successfully combined with DVS.

3.6 This thesis

All the micro-architectural adaptation techniques published are for synchronous, superscalar processors. While there is obviously far more scope for experimentation in superscalar architectures, the majority of embedded systems still use single-issue processors. This thesis aims to exploit micro-architectural adaptation for this class of processors.

Moreover, with the possible exception of LPX [BBB+02], there are no micro-architectural adaptation techniques developed for asynchronous processors. Even LPX is not entirely asynchronous; it follows the 'locally asynchronous, globally synchronous' paradigm proposed by Athas [Ath01] and uses asynchronous circuits only for some functional units, while the global control is synchronous.

Chapter 4

Power analysis of AMULET3

In order to gain a better understanding of where the power is consumed in an asynchronous processor, a detailed power analysis of AMULET3 was undertaken. The analysis is based on simulations for the following reasons. First, there were no fabricated chips available to measure at the time; even if there were, the taped-out chip contains much more than the processor and there are no separate supply pins for each component. Second and more importantly, the focus is on the power consumption at a very fine granularity, in which case it would be impractical to build an IC with so many different supply lines and pins.

Although the main emphasis is on the core, all of the other components in the AMULET3i system were monitored, but the interest is in their total consumption, not in the breakdown into sub-blocks. This is very important because there is no point in investigating methods of improving the processor's power efficiency if its contribution to the system power is small. The information on the power consumption of the other components will keep the analysis in the right perspective.

This chapter first describes the hardware that was simulated and the software that was run on the processor. The power breakdown is then presented at the top level and at the main block level. Based on the power analysis, a comparison with the closest synchronous processor, ARM9, is presented in section 4.6. Finally the last section summarizes the chapter and sets the main direction for the main part of this thesis.

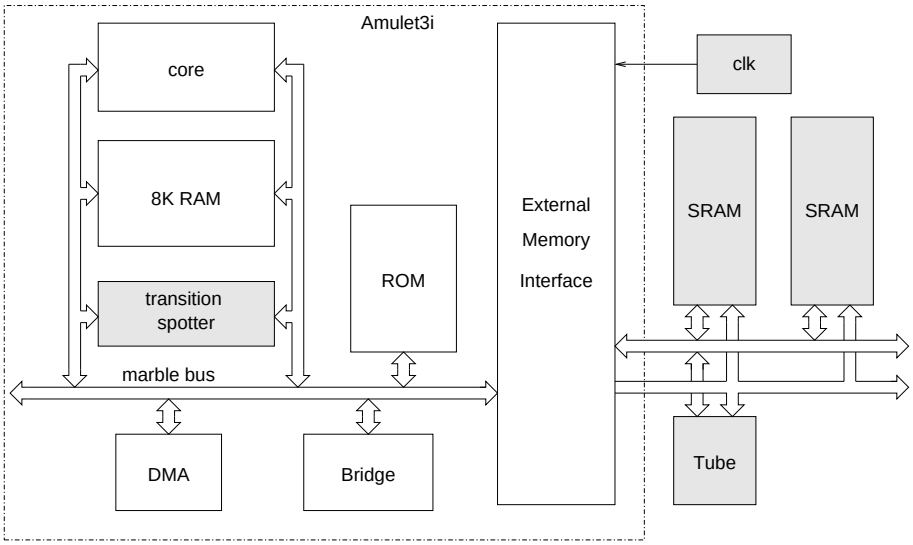


Figure 4.1: Simulated system.

4.1 Simulated hardware

The simulated system is shown in figure 4.1. It is an AMULET3i system [GBB+00], [AMU99] with small additions. There are three types of modules:

Behavioural support modules: Tube, SRAMs, clk These are included so that the system is easier to simulate. They are C functions linked with the simulator and behave like memory-mapped peripherals.

Tube is a multi-purpose 'peripheral' which can imitate external interrupts, terminate the simulation, and print characters to the screen and/or a file, including the current simulation time. All these operations are initiated by the processor writing appropriate values into the *Tube* memory-mapped registers. For these simulations the *Tube* is used as an output device only, so that the executed programs can print their results, or the elapsed time. It is also used to stop the simulation when the executed program has finished.

The two *SRAM* blocks extend the total memory available to the system by 128K bytes, in addition to the local 8K byte RAM. Finally, *clk* generates a reference clock which can be used by some of the peripherals.

All of these modules were used for the validation of AMULET3 and were previously created by other members of the AMULET group.

Flat netlists: Peripherals, buses, local RAM These are circuit netlists extracted directly from layout so there is no hierarchy within them. This is because the back-end tools of the Compass Design Automation software, that was used to design AMULET3, destroy the design hierarchy. All of them are part of the previously designed AMULET3i chip and they are described in [GBB+00].

Hierarchical netlists: processor core The processor core is the only fully hierarchical circuit netlist. It also contains the parasitic capacitances extracted from layout. This was generated by joining the flat back-end representation with the hierarchical schematic representation and its extraction was the non-automated, laborious work of a member of the AMULET group. For this reason the other parts of the system were left flat.

4.1.1 Core description

Figure 4.2 shows a block diagram of the AMULET3 asynchronous processor core [GFC99]. It implements the ARM v4T architecture [Fur97] using 4-phase (return-to-zero) handshake protocols and bundled-data representation. This architecture implements two instruction sets: the 'standard' 32-bit and a compressed, 16-bit instruction set called "Thumb". Although AMULET3 is a single-issue processor, limited out-of-order completion is supported: data transfer instructions, especially 'load multiple' (LDM), can be overtaken by subsequent data processing instructions. For this reason a four-place reorder buffer is used [GG97].

AMULET3 has two independent memory ports, one for instructions and one for data, to avoid pipeline stalls when load or store instructions are executed. The local memory in AMULET3i is shared by instructions and data; it is segmented in banks and interleaved. Thus pipeline stalls can still (seldomly) occur when the same bank is accessed by both memory ports simultaneously.

The *prefetch* unit is responsible for generating the address of the next instruction to be fetched. It contains a small, 16-entry branch target buffer (BTB) for branch prediction. Branches that are taken once are stored in the BTB and are subsequently predicted taken, until the entry is removed from the BTB to make room for a more recently taken branch. As all the information for previously predicted branches is kept in the BTB, the instruction fetch stage is bypassed when they are to be fetched.

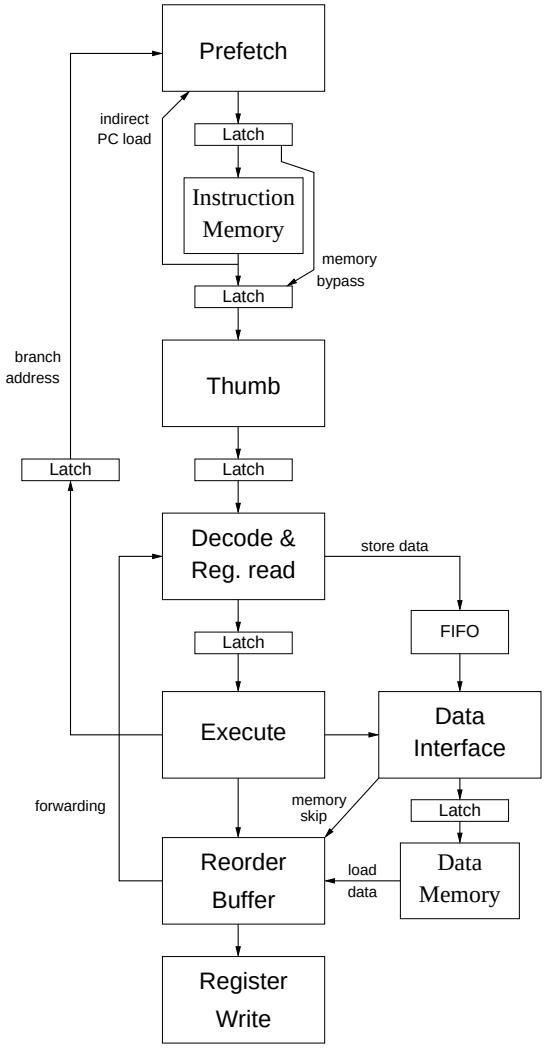


Figure 4.2: Organisation of AMULET3.

If the processor is executing Thumb instructions, the *Thumb* stage receives 32-bit words from memory, splits them into two 16-bit Thumb instructions and translates them into ARM instructions one after the other. Generating two instructions from one word is easily accommodated with the asynchronous design style; the upstream stages are slowed down by simply acknowledging their requests at a slower rate. When 32-bit instructions are being executed, *Thumb* behaves as a simple buffer.

The next stage in the processor pipeline, *decode*, decodes the instructions, fetches their operands and directs them to the appropriate unit(s) for processing. In addition, space is reserved in the reorder buffer for the instruction result and forwarding paths are set up.

Execute checks the instruction's condition code with the current processor status and then performs the data processing operations or the calculation of the memory address for data transfer instructions. Branches are also resolved in this stage and, if they are taken, the target address is sent to *prefetch* via a decoupling latch shown to the left of the pipeline in figure 4.2.

The *data interface* handles the data transfers, getting the initial addresses from the execution unit but generating subsequent addresses of multiple data transfer instructions internally. The reorder buffer gathers the execution results and the loaded data, possibly out-of-order and handles the forwarding of the results back to the decoder stage when the latest value of a register is not in the register file. Finally, the register write stage empties the reorder buffer into the register file.

4.2 Simulated software

The power consumed by a processor depends on the operations it performs. So the selection of the set of programs used to measure a processor's consumption is important. Unfortunately, there is no de facto collection of benchmarks used to measure processor power as there is for performance (e.g. SPEC). Many of the published results are based on Dhrystone [Wei84], which is a synthetic benchmark that claims to have a dynamic mix of instructions similar to "typical" application programs. In addition to Dhrystone, DES encode/decode [How92] and GSM codec [DB94] were also used.

Apart from the unavailability of power benchmarks, there are two other relevant limitations: One is that the local memory of AMULET3 is very small for most benchmarks and their data sets to be stored. An obvious solution would be to use the "second level" SRAM that is available, but since this memory is a behavioural model, no measurement can be made for its contribution to the total power. The solution chosen was to use the most critical functions of the benchmarks in place of the full benchmark. This can be done by running the complete benchmarks with a profiler on an instruction level simulator. The most critical function is usually much smaller than the full benchmark and should fit in the system's 8K RAM, together with its data set. With this method, the local memory can be considered as a perfect cache, with no misses at all, even for the first time it must fetch a line.

The second limitation is simulation time. The simulated system has about 800,000 transistors and 300,000 capacitors, excluding the behavioural parts. As full transistor level simulation (Powermill [Syn98]) was chosen, the simulation is slow. This low level of simulation is preferred because it gives much more accurate results compared to higher level simulations [Seg96],[BP94]. The increased simulation time limits the number of instructions that can be executed by the processor in reasonable time. For example, two iterations of the Dhrystone main loop take about 15 hours to simulate on a Sun Ultra 5 workstation. For this reason the number of iterations and/or the data set sizes of the benchmark programs were limited, so that the total number of instructions executed is below 5000, which keeps the simulation time to about 24 hours.

All of these benchmarks were written entirely in C and were compiled using the C compiler of the ARM development tools. In the results presented here, the programs were compiled using speed optimisation options and the 32-bit ARM instruction format was used for the target code.

Of the benchmarks used, Dhrystone and DES encode and decode were small enough to fit in the local memory. GSM codec is large and by profiling the encode and decode parts, the most time consuming functions were chosen for the simulations. The input data for the functions were gathered by encoding/decoding a speech sample in a modified version of the full benchmark, where the input data of the functions were saved to files. The set of functions selected for the encoder proved to be too slow to simulate, so only the decoder's most time consuming function, a signal filter, was finally used for this benchmark.

4.3 Configuration options

AMULET3 has a number of built in configuration options. The ones that are important for power consumption are those that control the enabling of the Branch Target Buffer (BTB), the colour counterflow mechanism, and the latch controlling power/performance signal (Turbo).

The colour counterflow mechanism is a way to flush, from the pipeline, instructions in the shadow of a taken branch at the decode stage. Following a taken branch, the execute stage counterflows the information that a branch happened to *decode*, which starts discarding instructions until those from the branch target arrive. A 1-bit colour is attached to each instruction at the prefetch stage, which

is toggled every time a branch happens, so that subsequent stages can know when a new flow of instructions arrive. Colour counterflow is an optimisation because these instructions would have been recognised as invalid at the execution stage anyway. From the power consumption point of view this optimisation seems beneficial because stopping these instructions earlier saves energy in the decode and register read pipeline stage and also the energy spent to nullify the space reserved in the reorder buffer for these instructions.

The pipeline latches of the processor core can be configured to be either normally open or normally closed [LGB99]. Normally open means that the latches become transparent as soon as the downstream stage has signalled it has processed the data, even if no new input data are available. This means that spurious transitions can propagate to the following pipeline stage consuming unnecessary energy. If the latches are normally closed they pass their inputs to their outputs only after it is known that the input data are valid. This makes the data transfer slower, but saves energy. There is clearly a power - performance trade off that is controlled by the control signal (*turbo*) that selects the operating mode of the latches.

4.4 Power consumption measurements

Table 4.1 shows the total power consumed by the system (sys), the processor core (core) and the local memory (mem), for the three benchmarks and for a number of configuration options. Note that up to 10% of the power is unaccounted for in the table. This is consumed by the bus and the peripherals. Table 4.2 shows the execution times and energy-delay products for the benchmarks and their configuration options.

From these results it is clear that the processor is the major consumer of power in this system. It is responsible for 59% to 73% of the total power in all of the benchmarks and configurations.

The local memory accounts for 21% to 31% of the system power. This corresponds to the ideal case where everything the processor needs is always in the local memory. Usually, the processor will either have to access some data at a second level of memory or there will be some form of memory management to move blocks of code and data along the memory hierarchy. In both cases the energy consumption in the memory system will be increased compared to this

Table 4.1: Power consumption and breakdown of the system (mW).

Options			Dhrystone			DES encode			GSM decoder (filter)		
B	C	T	sys	core	mem	sys	core	mem	sys	core	mem
√	√	√	223	138	64	260	165	75	238	172	50
			100%	62%	29%	100%	64%	29%	100%	73%	21%
	√	√	215	129	65	253	158	75	228	162	51
			100%	60%	30%	100%	63%	30%	100%	71%	22%
√		√	226	141	64	260	165	75	239	174	49
			100%	62%	28%	100%	64%	29%	100%	73%	21%
√	√		219	134	64	256	162	74	237	173	49
			100%	61%	29%	100%	63%	29%	100%	73%	21%
	√		211	125	65	249	155	74	226	159	51
			100%	59%	31%	100%	62%	30%	100%	71%	22%

Table 4.2: Execution time and energy delay product (EDP) (μs, nJs).

Options			Dhrystone		DES encode		GSM decoder	
BTB	C/flow	Turbo	Time	EDP	Time	EDP	Time	EDP
√	√	√	15.452	32.9	46.014	349.3	52.514	477
	√	√	15.563	31.2	46.041	335.3	56.541	517
√		√	15.604	34.3	46.087	350.5	52.608	481
√	√		15.558	32.4	46.403	348.8	52.743	480
	√		15.679	30.7	46.434	334.4	56.843	516

analysis and the average access time will be affected.

The contribution of local memory and processor to the system power presented here matches quite closely that reported by Pering *et al.* [PBB98]: 58% core, 33% cache, 7% processor bus and 2% SRAM. They also use an ARM architecture processor, but their system is fully synchronous and uses a 16KByte cache. Their results also include the effect of the external SRAM. It is not clear however if this power decomposition is valid with or without the dynamic voltage scaling that they use.

4.4.1 Effect of configuration options

The impact of the processor configuration options for each benchmark can be seen in table 4.1. Five configurations were simulated. They are represented here as strings of the letters B, C, T, where the existence of the letter in the string means that feature is enabled: B - BTB, C - colour Counterflow, T - Turbo (normally open latch controllers).

BCT All of the features are enabled in this configuration. This is considered the high performance configuration, with all the performance options turned on. Evidently, it achieves the fastest execution time in all benchmarks. At the same time, in all but GSM, the energy efficiency (EDP) of this configuration is the second worst.

CT Branch prediction is disabled in this configuration. This leads to a lower power consumption for the core but somewhat higher for the local memory. This is because the BTB avoids accesses to memory, both because the branch instruction itself is not fetched and, when the prediction is successful, only the correct instructions are fetched. In order to have these benefits the BTB table has to be consulted for every new instruction to be fetched, which has a quite significant power cost for the processor core. From table 4.2 it is clear that branch prediction always saves some execution time, although this is significant only in the GSM benchmark. This configuration achieves the second best energy efficiency for Dhrystone and DES encode, but the worst for the GSM filter.

BT Branch colour counterflowing is turned off in this configuration. From the results it is clear that colour counterflowing is beneficial for the power consumption of the processor and the system, albeit slightly. It has to be noted that the branch prediction is enabled in this configuration which means that the effects of the branch colour counterflowing will be minimised, assuming the prediction is successful. The execution times of all benchmarks increased for this configuration in comparison to *BCT* and the energy efficiency is the worst for all but GSM filter.

BC *Turbo* is disabled in this configuration, so the pipeline latches are normally closed. Having normally closed latches, makes a pipelined circuit consume less power when the pipeline is not fully occupied [LGB99]. The fact that there is some benefit from turning *Turbo* off in AMULET3 means that the processor's pipeline is not always fully occupied. This is not surprising since it is well known that branches affect the average occupancy. In addition, due to the asynchronous nature of the pipeline and the differences between the minimum and maximum times for neighbouring stages to finish, it is natural for the pipeline to have some slack. The performance is worse for all of the benchmarks compared to the *BCT* configuration but by less than

1%, so the energy efficiency is in the middle of the range for all benchmarks.

C Only branch colour counterflow is enabled in this configuration. Branch prediction and *Turbo* are independent, so the result of combining them should be aggregate. Evidently, from the results it can be seen that this is roughly true. The performance of this configuration is consistently the worst for all of the benchmarks, but the energy efficiency is the best in all but GSM filter.

From the above discussion it is concluded that Dhrystone and DES encode have similar behaviour in all of the configurations, while the GSM filter is different. The difference is based on the fact that branch prediction makes a significant difference to the execution time of GSM filter, while this is not the case for the other benchmarks. This is because the GSM filter code is a double nested loop which is executed many times, so branch prediction is very successful, while branch behaviour is more unpredictable in Dhrystone and there are almost no branches in the DES encode program.

4.5 Core power breakdown

The processor power breakdown into the top level blocks is presented in figure 4.3. The biggest consumers are the execution block and the registers[1] block, followed by the prefetch unit, when the branch prediction is enabled, and the decoder.

From these results it is clear that the increase in the absolute processor core power consumption observed in the DES and GSM benchmarks is attributed to the registers and execution blocks. Interestingly when running Dhrystone or DES encryption both blocks consume about the same power, but for the GSM filter the execution unit power consumption exceeds that of the registers block. This is attributed to the frequent use of the multiplier which is not used in the DES encryption program.

It is also clear that, apart from the significant reduction of the prefetch block's power when branch prediction is turned off, no configuration option makes a significant difference to the power consumption of a specific block.

The power breakdown in the main blocks is described in the rest of this section.

[1]The registers block comprises the register file and the reorder buffer.

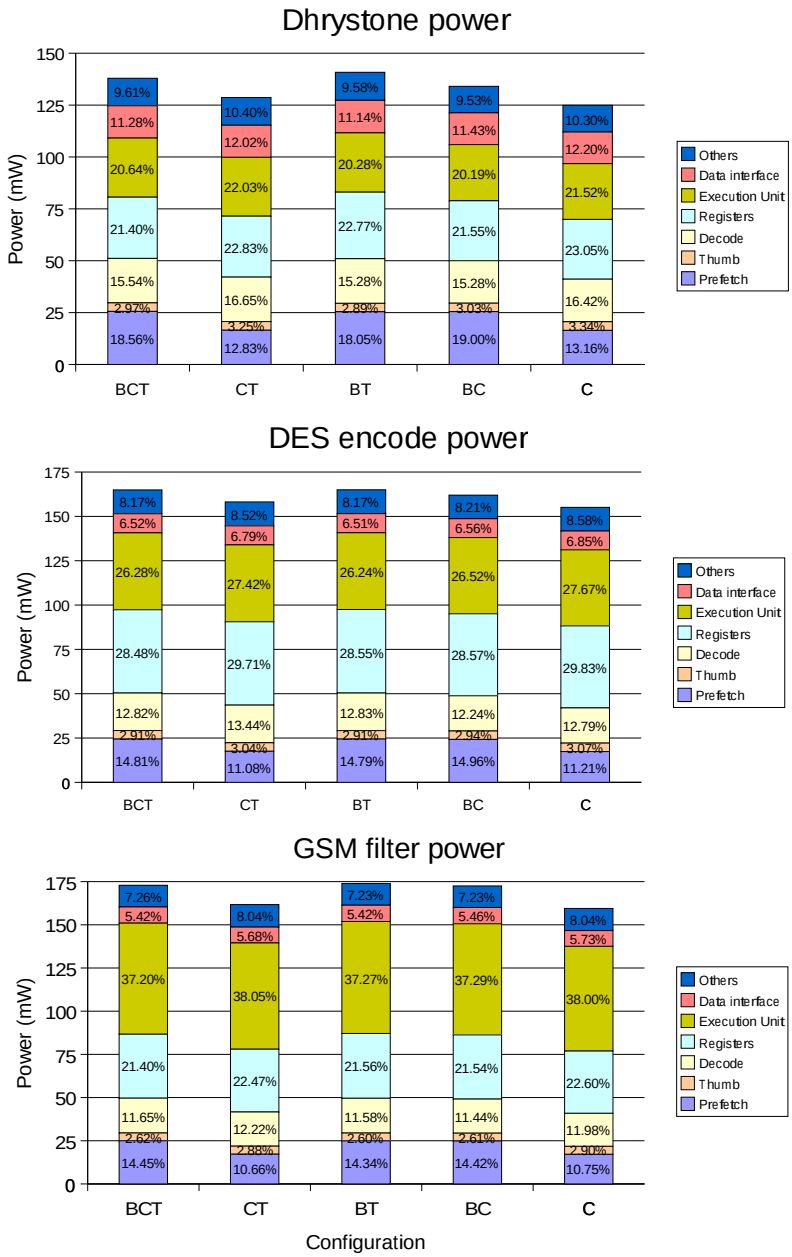


Figure 4.3: Processor core power breakdown.

4.5.1 Prefetch

In the prefetch block the power is consumed mostly in the block's control unit, the branch target buffer, when it is enabled, and the instruction memory address drivers (see figure 4.4).

When enabled, the BTB performs a 16 entry CAM look up in every cycle. The RAM is read only when a match is found, which means that the active

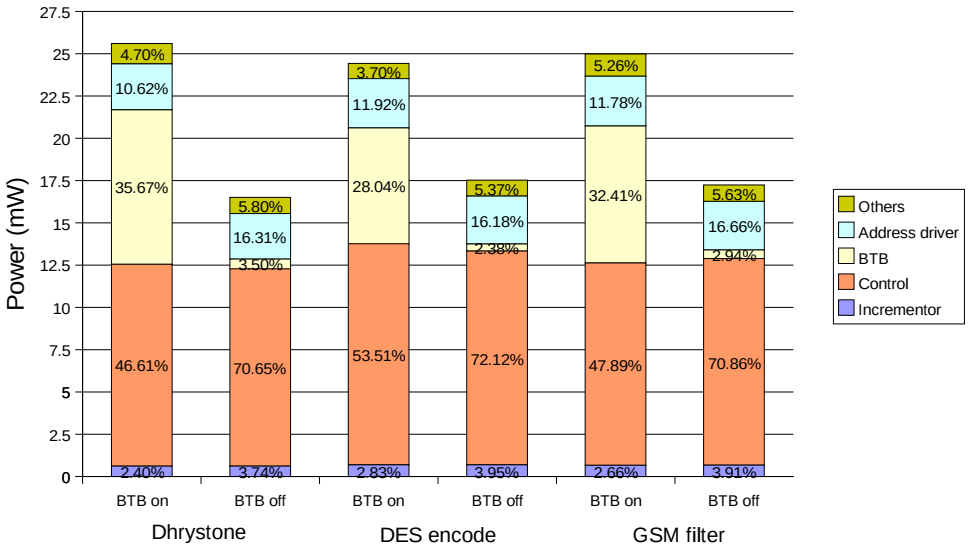


Figure 4.4: Prefetch block power breakdown.

parts of the branch prediction hardware were kept to a minimum. Even with this optimisation branch prediction is a costly — in energy consumption — feature of the processor.

The PC incrementor is implemented as a simple ripple carry adder. It is well known ([CB95a]) that the probability of a PC bit switching is halved from a bit position to the next most significant bit position. For this reason the ripple carry adder was selected and it is apparent that the incrementor's power consumption is negligible.

Finally, the address bus drivers make a non-negligible contribution to the block's power consumption. Naturally, the address bus has a high parasitic capacitance and, although on average only a few bits are switched every cycle, the contribution of the bus drivers is a sizeable part of this block.

4.5.2 Decode

Within the decode block, the sub-blocks that consume most of the power are the *sequencer* (main control), which drives high-fanout control signals, the input latch controller (from *Thumb*) mainly for driving the very high-fanout load enable, the branch target adder and the immediate generator (figure 4.5). 'Others' represents the combinatorial circuits that perform the actual decoding of the instruction opcodes.

The input latch controller drives the load enable signal for the decoder block input latches. The latch inputs originate from the *Thumb* block and the total number of bits latched is 151, so the fanout of the load enable signal is significant. In fact about 80% of the controller's power is spent driving this signal, which is about 1.4% to 1.8% of the processor power, depending on the benchmark and configuration.

Sequencer is the main controller of the decoder. As with most other control circuits in AMULET3, its high contribution to this block's power consumption is due to the high capacitance of the control signals that it drives.

The branch target adder is a fast adder, similar to that used in the ALU. This choice was made because this adder is in the critical path for branch instructions. Looking at the consumption of this block when executing the DES encode benchmark gives some idea of how much power is wasted in this adder. DES encode has virtually no branches. Nevertheless, the adder's consumption as a portion of the decoder is only approx. half of that of the other benchmarks. This means that possibly, up to half of this block's consumption is wasted. However, this accounts for less than 1% of the core power, so adding hardware (e.g. operand latches at the input) to remedy this, may result in a slower circuit and higher power in the control part.

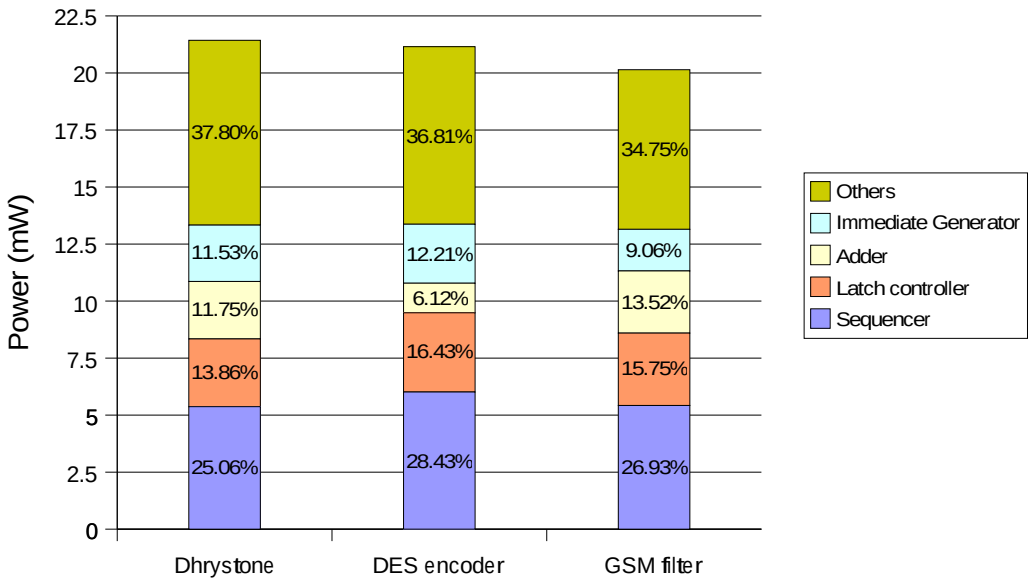


Figure 4.5: Decoder power breakdown.

The immediate generator produces various forms of immediate operands used in a number of instruction types. The inputs are selectively latched if the instruction is using an immediate operand, thus there is switching activity only when necessary.

4.5.3 Registers

Figure 4.6 shows the power breakdown in the *registers* block. As expected, the register file and the main part of the reorder buffer (*Queue*) are the highest consuming sub-blocks. It has to be noted that *Queue*, *QCAM* and some of the blocks under the *Others* category, form the reorder buffer, so its total power consumption is slightly less than half of the block's total consumption.

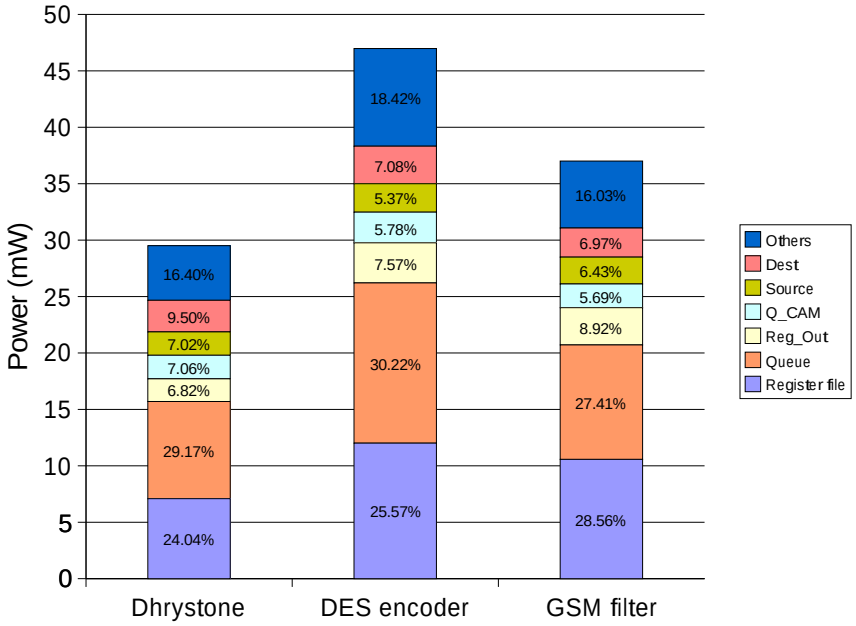


Figure 4.6: Registers block power breakdown.

In the register file most of the power is consumed by precharging the read ports. The power consumed depends on the actual values read through the ports, as a register containing mostly 1's will discharge more lines than one containing mostly 0's. This is why the DES and GSM spend more power in the register file compared to Dhrystone.

4.5.4 Execution

Figure 4.7 summarises the power consumption within the execution block. It is clear that the highest consumers are the ALU, the multiplier, the control, the result bus driver, the multiplexor for the second operand of the ALU and the shifter.

In the ALU, a significant proportion of power is consumed by the output drivers, which have a quite high capacitance to drive. A factor that increases the consumption of the drivers is that the ALU uses precharge logic causing increased transitions in most ALU signals. Another interesting observation is that although the output of 'compare' instructions is used only to generate the condition codes, the current implementation allows the ALU to drive the result bus as well. Since comparisons are a non-negligible part of the executed instructions, there is some room for improvement here.

The control block of the execution unit oversees the reception of the operands to be acted on, checks the condition codes and branch colour and handles the communication with the data interface and the reorder buffer. In addition it provides the trigger signals for the functional units contained in the execution block. Its power contribution can be attributed to the high fanout control lines that it has to drive.

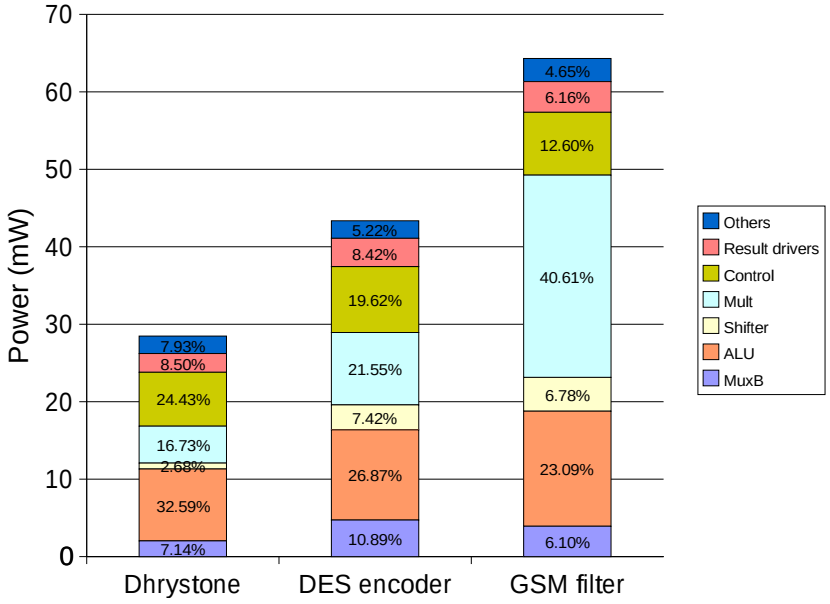


Figure 4.7: Execution unit power breakdown.

The multiplier's power consumption was a surprise. It was expected that it would consume significant power when it operated, but it proved that it also consumes considerable power even when it is not used. For example, the DES encode benchmark does not have any multiplication instructions, but still the multiplier is responsible for about 22% of the execution block's power. Clearly some parts of the multiplier are always active. An investigation showed that Booth encoding and partial product generation were performed for every data value on the operand buses that connect to the multiplier. This problem was fixed and the power consumption results presented in the remaining chapters are based on simulations using the modified multiplier.

4.5.5 Data interface

In the data interface block, control is the main power consumer, followed by the datapath and the data bus drivers (figure 4.8). The two stage FIFO that holds the decoder requests, *Instruction Pipe*, consumes under 10% of the block's power. Control is the largest part of the block and drives high fan out signals, so its proportion of the power consumption is justifiable.

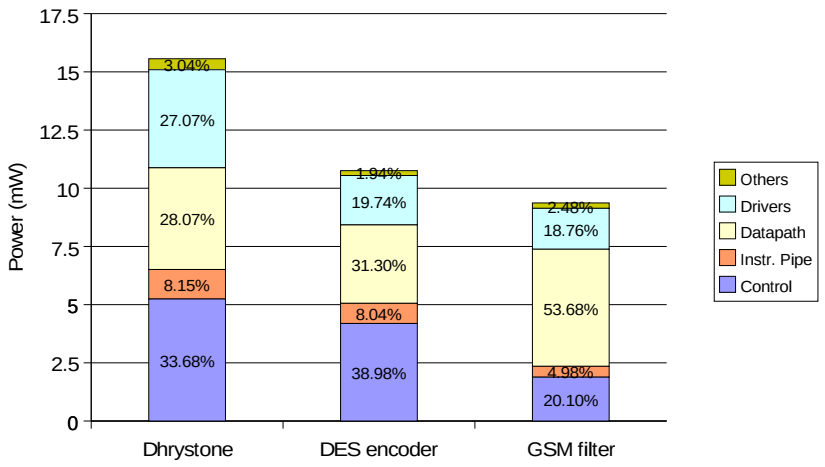


Figure 4.8: Data interface power breakdown.

The datapath consists of three basic parts. The largest is the data address multiplexor and incrementor (for the multiple loads or stores). The other two have to do with the extraction or generation of the sub-word values for loads or stores, respectively. The address multiplexor and incrementor consume about 70% to 80% of the datapath power.

Finally, the data bus drivers (*drivers*) are connected to the high capacitance external wires of the data address, outgoing data and control buses. As there are no glitches on these wires there is no power wasted here.

4.6 Comparison with synchronous systems

In a synchronous processor a large proportion of the power is attributed to the clock. Tiwari et al. [TSR+98] state that about 40% of CPU power is spent on the clock, including the generator, drivers, distribution tree and loading. StrongARM [MWA+96], with a similar architecture to AMULET3, is reported to use 26% of its power in the clock, including the PLL. Other processors state similar results.

It would be interesting to estimate the equivalent of this power in AMULET3. As there is no simple way to determine which circuits should be considered equivalent to the clock, the choice is somewhat arbitrary. The closest equivalent is the set of latch controllers of the pipeline latches between the submodules of the core; these include the drivers for the large latch enable loads. In addition some precharge signal drivers which are controlled by handshake signals were also included in this group. The contribution of all those circuits to the power consumption was found to be 10.5% of the core while running Dhrystone 2.1. Comparing this with the proportion of the power taken by a clock shows that asynchronous techniques can significantly reduce power consumption.

Naturally the benefits of the asynchronous design style come at a cost. As the different stages in the pipeline are not synchronised, state information between pipeline stages is difficult to exchange. This leads to duplication of information in several places in the pipeline. For example, in AMULET3 each pipeline stage holds the address of the instruction being processed there because of the difficulty in accessing a central PC. This is not a significant power overhead, because only a few of the PC's bits switch each time. In addition to duplicating information, the fine grain control of circuits leads to the existence of more state/sequencing information compared to a synchronous processor. This translates to an increase in control power for the processor, proportionately about 40% of the processor core power in AMULET3. This is quite high compared to other published results, although this includes most of the 'clock' power stated above. It has to be noted though that the control circuits are implemented using standard cells,

automatically placed and routed by CAD tools, whereas the datapaths are full-custom. Thus the wire capacitances tend to be higher and there is less control over the driving strength of the gates. As synchronous low-power processors are increasingly using extensive clock gating and functional unit guarding techniques their control units tend to be as complicated as their asynchronous counterparts. Moreover, the ARM architecture is quite complex for a RISC machine which makes the control logic inherently more difficult and power consuming. ARM7 [Seg97], which implements the previous version of the ARM architecture, is stated to consume 40% of its power in the control part, although the definition of control circuits may be different.

4.6.1 Comparison with ARM9

ARM9TDMI is the synchronous implementation closest to AMULET3; both execute the same instruction set and have been implemented on technologies of the same feature size and occupy the same silicon area (about 4mm^2). On this 0.35 μm process ARM9 [Seg98] operates at up to 120 MHz having a performance of 1.1 MIPS/MHz and consuming 1.8 mW/MHz. This gives an energy per instruction metric of 610 MIPS/W. Unfortunately a power breakdown has not been given.

The measured results from AMULET3i show a power consumption of 221 mW at a performance of 85 Dhrystone MIPS. According to the simulation results, the core consumes 62% of the power (137 mW). This gives an energy per instruction figure of 620 MIPS/W, effectively the same as that of ARM9.

It has to be noted that the speed of AMULET3i is slower than anticipated from simulation, which predicted a speed of about 100 MIPS for the system. This is attributed to the (unoptimised) memory system; the core alone runs at about 130 MIPS in simulation. Thus, it is safe to assume a speed of over 100 MIPS for the processor on silicon, without the limitations of the memory, which would raise the energy efficiency of AMULET3 to about 730 MIPS/W.

4.7 Summary

In this chapter a power analysis of an asynchronous processor was presented. The micro-architecture of the processor core was described and the effect of the configuration options was evaluated. A breakdown of the core's power consumption at the top-level and in each of the main components was also given.

From the power analysis it is clear that, without sacrificing speed, there is no one specific part that could significantly improve the power consumption of the processor if an improved design for it were invented. The same is possibly true for any processor, as the parts that could have a significant impact have already been optimised. Thus, unless a significant breakthrough in circuit design styles or computer architecture happens, to improve the energy efficiency of processors, adaptive techniques such as those presented in chapter 3 should be used. So the rest of the work presented in this thesis is aimed in this direction.

Chapter 5

Speculation control techniques

This chapter presents a number of new techniques to control speculation in an asynchronous processor core.

Before the speculation control techniques are described, the experimental setup with which these techniques were designed, implemented and measured is presented in section 5.1. Three speculation control techniques are described in the remaining sections, with the emphasis on pipeline occupancy, as this allows a wide range of speed/energy tradeoffs.

5.1 Experimental setup

The models, tools, scripts, etc. that are used to simulate relatively large benchmarks and to get energy estimation and performance information are described in this section. In addition the tools and benchmarks described in chapter 4 are also used here wherever they are applicable.

5.1.1 Simulation and energy estimation

Accurate energy estimation by simulation is a difficult task and a research topic on its own. One of the problems is that, to get accurate results, extracted capacitance is needed, preferably including post-layout parasitic capacitances. The simulated system has to be fully designed and mapped to a library of logic gates with known terminal capacitances for this information to be available. Experimenting with architecture changes using such an experimental setup is very difficult, as every idea has to be taken to layout before it can be evaluated.

The other problem is the unavoidable trade-off of simulation time and energy estimation accuracy. Transistor level simulators, such as Spice and its derivatives, solve the circuit equations for each component and thus produce very accurate results, at the expense of very long simulation times. With a claimed accuracy loss of around 10%, programs like Powermill speed up simulation considerably by taking shortcuts in solving the equations [Syn98]. The improvement is such that it is possible to simulate AMULET3i (core, memory and peripherals) running around one thousand instructions of Dhrystone in approximately 15 hours on a Sun Ultra-5. To be able to run more instructions in processor simulation, faster, higher-level simulation is needed. Although such simulators are gradually becoming available, none of them is universally accepted in the research and/or industrial community as being accurate enough and, a more practical aspect, none was available for this work.

The energy estimation accuracy problem can be avoided by re-using the extracted capacitance from previously designed and extracted blocks. If the previous designs are in a different technology, as is usually the case, the capacitance can be converted automatically to the new technology by appropriately scaling the dimensions of the wires and the transistors. Usually only a relatively small part of a new system is designed from scratch; in most cases the additional parts consume so little energy compared to the re-used blocks that a rough estimate of their average switched capacitance is enough to produce accurate overall results.

Having node capacitances, all that is required to estimate the energy consumption of the system are transition counts for the nodes. In CMOS circuits an energy of $E = CV^2$ is consumed in each node for every full-rail transition, so the total energy consumption of the circuit can be estimated by multiplying CV^2 by the number of these transitions and summing for all the nodes. If the high level simulator can provide the number of toggles in the nodes, the energy consumption can be easily estimated.

The simulation strategy used in this work is based on the above observations and is quite similar to that of Burd [Bur01] and of a group at IBM T.J. Watson research centre [BBS+00] [BBB+02]. Both use capacitance estimated from previous designs or analytical models of the circuits and estimate energy during simulation by measuring node transitions.

5.1.2 Simulator choice

The asynchronous design style is notoriously badly-supported by CAD tools. Thus the only available AMULET3 models before this work were a LARD [EF98] model and the netlist from the design database (annotated with parasitic capacitances).

LARD is a language/simulator developed for asynchronous design, based on "channel" communication. Although it is a high-level behavioural simulator, it is very slow, mostly because it is interpreted. In addition there are very limited debugging and waveform viewing capabilities. Counting node toggles in LARD requires the designer to use variables explicitly to hold the previous values of the signals and compare them to the current ones each time a block is evaluated. As there is limited support for extracting specific bits from a variable, after XORing the current and previous values to find the transitions, shifting and masking is needed to extract the transitions for each bit within multi-bit variables. The extra LARD instructions used for these operations add considerably to the simulation time, exacerbating the problem. There is a recently developed compiled version of LARD which is significantly faster than the original [JE01], but still slower than the simulator used in this work. Moreover the way node toggles are counted is unchanged, so energy consumption is still hard to estimate.

The other available AMULET3 model, the annotated netlist, is only available in Spice or EPIC (Powermill) format, so it can only be used directly with these simulators. As explained previously, higher-level simulation is needed, so a Perl script was developed which converted this netlist into a Verilog netlist, so that it could be used with fast Verilog simulation. Unfortunately, the library of standard cells used in AMULET3 does not have a Verilog representation and a large part of the processor is custom made. Since the technology used for AMULET3 was already two generations old by the time this work started, there was no point in investing significant time and effort in characterising and generating Verilog models for these cells in a technology that would not be used again.

Since LARD was deemed inadequate and the AMULET3 netlist could not be used directly with a high-level simulator, a significant amount of time was spent to manually create a Verilog model of AMULET3 from the processor schematics. To save some development time, the model is mixed structural/behavioural but retains most of the hierarchy of the processor; all of the important signals are

preserved, as well as most of the high capacitance intermediate signals. Simple combinatorial logic parts are modelled using continuous assignments and all sequential parts are modelled with behavioural "always blocks". Delays were inserted manually based on the delays produced from Powermill simulations of the original extracted netlist. The model was verified using the ARM validation suite and the collection of benchmarks used for the evaluation.

Verilog simulation is fast, especially in the "compiled" version, NC-Verilog from Cadence. Table 5.1 summarises the simulation times for Dhrystone using Powermill with the full extracted netlist, LARD, Verilog-XL (interpreted simulation) and NC-Verilog. Using NC-Verilog speeds up the simulation by 2000 times compared to Powermill. Most of the time in the NC-Verilog simulation was taken up in the simulator's internal initialisation, so the speed advantage increases when running larger benchmarks. From the table it is shown that Verilog-XL is 6 times faster than LARD. The real speed difference is actually higher because the time presented for Verilog-XL includes the time to parse the input files. That would have been an additional 108 seconds for LARD, pushing the difference to over 10 times.

Table 5.1: Simulation speed comparison (Dhrystone)

Simulator	Simulation time
Powermill	20583 sec
LARD	138 sec
Verilog-XL	23 sec
NC-Verilog	11 sec

5.1.3 Energy estimation

An overview of the energy estimation flow is given in figure 5.1. As explained earlier, two elements are needed for the estimation of energy consumption at each node: total capacitance and toggle count.

The capacitance information only needs to be extracted once and then it is used selectively in simulations that require energy estimation. The capacitances of these signals are extracted from the processor netlist using a Powermill configuration command that reports the total capacitance of a node. This information is kept in a file with lines containing node name and capacitance pairs. In total over three thousand nodes and their capacitances are used.

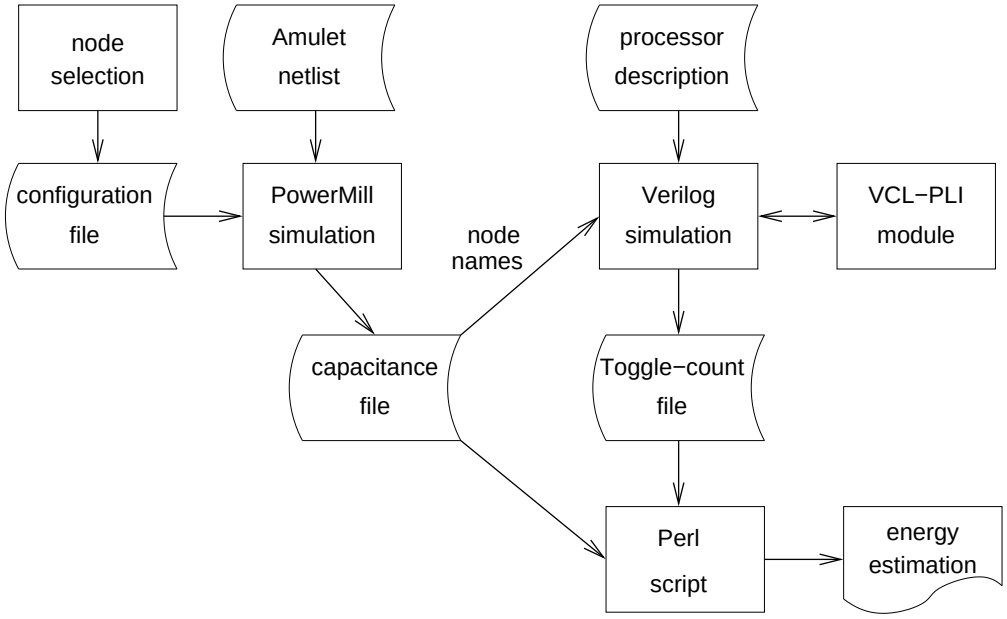


Figure 5.1: Energy estimation flow.

To count the number of node transitions, Verilog's programming language interface (PLI) is used. Specifically the value change link (VCL) is used which monitors value changes of selected nodes. At the beginning of the simulation, if energy estimation is required, the list of nodes which should be monitored is read and, for each node, VCL is set up to call a C function. The function simply increments a counter for the specific node, which is passed as a parameter. At the end of the simulation a PLI function is called to dump the collected toggle counts to a file.

After the simulation a Perl script combines the toggle counts with the capacitance and reports the energy consumption. As all the node names are hierarchical, i.e. they contain all the instance block names from the top-level, the Perl script is able to report energy consumption for each block in the design hierarchy, by summing up the consumption of the nodes in the block.

5.1.4 Benchmarks

Faster simulation allowed the use of more benchmarks (table 5.2) than the three presented in section 4. The new benchmarks are *compress* and *ijpeg* from SPEC Int95. In addition to faster simulation speed, these benchmarks also require file input/output and dynamic memory allocation; the following two sections describe

Table 5.2: Dynamic instruction count of the benchmarks

Benchmark	Dynamic instr. count
Dhrystone	1100
DES encode	3888
GSM filter	4457
compress	4103550
ijpeg	11247880

how these are handled. It is these requirements that made it difficult to add more SPECInt95 benchmarks in the list: the development of a complete run-time system is beyond the scope of this work, so it was not undertaken. Nevertheless, *compress* and *ijpeg* are the most likely of the SPECInt95 benchmarks to run on an embedded system, for which a processor like the one described here is designed.

The new benchmarks execute a few million instructions, while the previous ones executed only a few thousand. Their input data sizes were reduced compared to the provided reference set, as they would otherwise take days to simulate. This does not seem to affect their usefulness as benchmarks; the execution time was not dominated by the run-time library functions or file input/output.

Admittedly, even with the addition of the two SPECInt95 benchmarks, the total number of benchmarks used in this work is relatively low. However adapting other benchmarks to work with the simulator is a very time consuming task and it was considered that this time is better spent developing new architectural ideas than debugging benchmarks.

5.1.5 System calls, I/O emulation

To aid the simulations, some way of printing output messages on the screen must be provided. The approach used in the Verilog model, was to extend the previously developed method used in the Powermill (and LARD) simulations of AMULET3, which was capable of printing characters (or the simulation time) on the screen and of ending the simulation. This was to use a pseudo-peripheral, called *the Tube*, which is mapped to a reserved area in the memory address space. Writes and reads to specific memory locations in this area have special meaning and trigger the emulation of input/output operations in the host processor where the simulation is running. Operations that need more than one parameter are handled by writing the information in a block of memory and making the last

write trigger the operation. When there are data to be returned, a subsequent read of a location in the reserved memory space is used. The operations supported are listed in table 5.3.

Table 5.3: Supported I/O operations

Operation	Description/details
Original Tube	Print ASCII character, current time, finish simulation
fopen	Open file for read or write
fclose	Close file
getc	Read character from file
ungetc	Return read character to file buffer
putc	Write character to file
ftell	Current position in a file
fread	Read a number of bytes from a file into memory

Some of these operations are handled by Verilog built-in system functions; the rest are implemented using user-defined functions using the Verilog PLI [Spe99].

When a benchmark is compiled using the ARMtools compiler, the run-time library functions linked with it are those which work with the ARM debugger-simulator. To be used with the Verilog model, the library functions had to be changed to use the *Tube* and its services for input/output. The library functions were written in assembly code and are made up of two parts: a user-mode part and a privileged-mode part. As the *Tube* is a peripheral, the memory space it is mapped to should be protected, thus accesses to it are allowed only when the processor is in *supervisor mode*. Currently this is not strictly necessary, but it could be useful for the future if memory protection is used. Thus, the user-mode part of each library function, loads its parameters to registers and calls the software interrupt instruction (SWI) providing it with a number representing the operation it wants to perform. The SWI changes the processor mode to *supervisor* and branches to the appropriate privileged-mode routine depending on the SWI number provided. That routine just writes the parameters to the specific memory addresses and, if there is a return value to be collected, reads and stores it in a register. The routine switches back to user mode and to its caller function, which just returns to the program with the returned result if any.

Apart from the run-time library functions that are used for input/output, dynamic memory allocation (*malloc*) had to be supported as it is used by many programs. The *malloc* provided by the standard run-time library used by the

ARMtools could not be used, as it is tightly integrated with the rest of their run-time support system. Thus a custom dynamic memory allocation system was developed.

The dynamically allocated area is at the end of the program code and the global variables, which are stored at low memory addresses. The linker provides a symbol name pointing to the first memory address of the unused memory and the custom malloc function maintains a pointer to the highest unallocated memory address starting from there. As memory is allocated, this pointer moves higher. To simplify the implementation, the released memory is not recovered, so calls to *free* just return back to the caller. In addition no checking is done if the allocated area runs into the stack. As the stack starts at address 0x7000000 there is plenty of space between them and memory allocation or overrun problems were not experienced in the simulations.

5.1.6 The memory model

To allow the processor to run large programs, the memory model has to give the impression that the complete 32-bit address space is available[1]. Obviously it is impractical to declare and use a memory of that size in Verilog. Thus the equivalent of a paged virtual memory system was implemented, which uses the host processor's disk space for secondary storage when needed.

Sixteen memory blocks of 32 KBytes each are defined and a separate *translation table* keeps the *page numbers* of the pages stored in the memory blocks. For each access, the translation table is looked up to see if the page needed is already present. If it is, the appropriate memory block is accessed; otherwise, the file directory where the simulator was started is searched for a file named after the page number that needs to be loaded. If such a file exists, its contents are loaded into one of the memory blocks, using a round-robin replacement algorithm. If the page to be replaced has been modified, the data in that block are saved back to disk first.

For the memory system to work, the program code to be executed is broken up into a number of files, named after their page numbers. After reset, the processor accesses the instruction at address 0, so the file named "0" is loaded first. Thus there is no need to pre-load the program into memory before the processor starts executing.

[1]With the exception of the area reserved for the *Tube*, which is handled differently.

Using this memory system, programs of any code size can be executed. As in the case of a virtual memory system, if the required memory is much larger than that available thrashing can occur, which will slow down the simulation enormously. In the simulations performed here a page was replaced in the memory system only a handful of times, so the simulation time was barely affected by reading/writing files for the memory system.

5.1.7 Other simulation methods

In some cases testing new ideas does not require full processor simulation because a good estimate of the energy consumption or performance can be obtained using simple metrics, e.g. number of fetched instructions. In these cases to speed up the evaluation and to use benchmarks with more executed instructions, a different approach was used.

The method under test is implemented as a C program which simulates only the part of the processor that is affected, using a previously generated trace file. The fast instruction-level simulator (armsd) contained in ARMtools v2.51 is used to provide the trace file. Large benchmarks can be executed on the simulator without modification, as a full run-time library is provided. The instruction-level simulator has to run only once to produce the trace file. So the most time consuming process is not repeated for each modification to the method that is being developed.

5.2 Conditional instructions

Conditional instructions can be considered as a specialised form of speculation: they must be fetched — because it is not known in advance whether they are conditional — but they may not be executed, thus the energy spent fetching and decoding them could be wasted. Nothing can be done about the energy spent fetching failing instructions, but if their condition is checked as early as possible, some 'processing' energy can potentially be saved.

Currently, in AMULET3 (fig. 5.2), the condition is checked at the beginning of the execution stage since the processor status is kept there. This means that instructions failing their checks have already wasted energy in the previous stages, decoding, fetching operands from the register file, assigning space in the reorder buffer and generating immediates or branch targets. Most of the energy

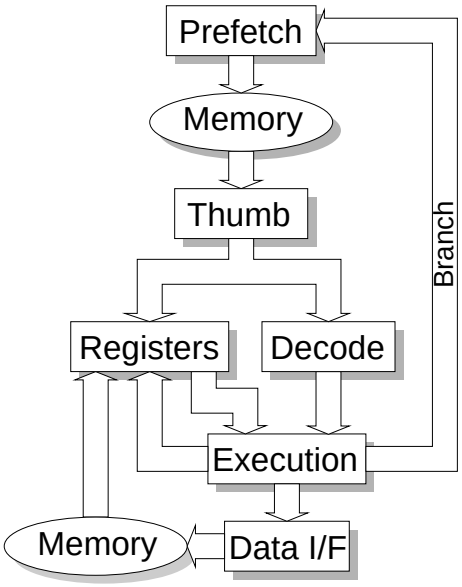


Figure 5.2: Block diagram of the AMULET3 processor core.

consumption in *decode* and *registers* could be saved for these instructions which constitute around 10% of the executed instructions [Seg98]. Based on the power analysis (chapter 4), it can be expected that around 4% of the core energy can be saved on average: 10% of both 22% (*registers*) and 16% (*decode*).

In the ARM instruction set any instruction can be conditional, the intention being to remove branches that skip a small number of following instructions [Fur97], as is commonly the case in *if* statements. The instruction format is uniform in this respect, so only the four MSBs of the opcode need to be examined to determine whether an instruction is conditional.

In the Thumb instruction set there is only one conditional instruction, branch. So there is no need to detect conditional Thumb instructions early, as there is seldom any wasted energy in decoding Thumb instructions. Moreover, since not much processing is done in the *Thumb* stage for ARM instructions, the condition can be tested at the beginning of the decode stage, where Thumb instructions have already been translated and thus all instructions can be treated in the same way.

The following section describes the design of a method that evaluates the condition at the decode stage. Before allowing any conditional instruction to proceed with decoding, the latest processor status is read from the execute unit, the condition is checked and, if it fails, the instruction is discarded without being decoded.

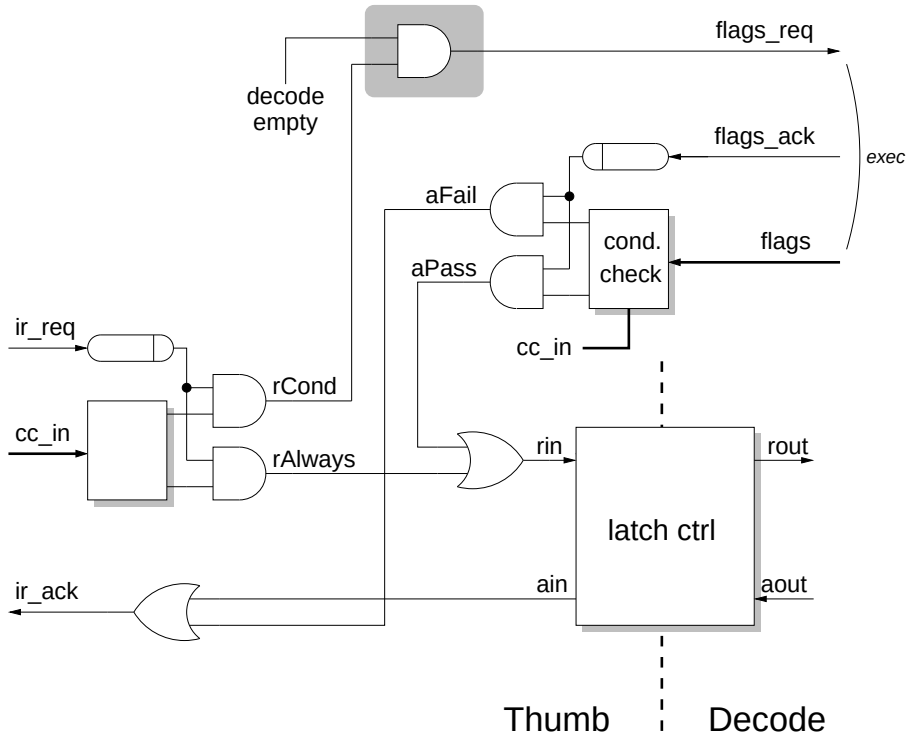


Figure 5.3: Block diagram of early condition checking.

5.2.1 Design of early condition checking

As most of *decode* is made of combinatorial logic, whenever new data are loaded to the input latch there is nothing to stop the circuits switching and consuming energy. For this reason conditional instructions should not be allowed to pass through the pipeline latch before their condition is tested. Thus the circuit that checks the conditions must be integrated with the latch controller of the pipeline latch. Figure 5.3 shows an overview of the combined latch controller – condition checking circuit.

When a new instruction is to be decoded, *ir_req* is raised. Depending on the four MSBs of the opcode which specify the instruction's condition (*cc_in*), either *rCond* or *rAlways* is asserted for conditional and unconditional instructions respectively. Unconditional instructions go through directly, raising the request input of the latch controller through an OR gate.

Conditional instructions generate a request (*flags_req*) to get the latest processor status and specifically the *flags*: overflow, negative, zero, and carry. For *flags_req* to be raised, the decode stage must be empty; there must be no instruction — which can change the flags — between that at the decode input and the

one (if any) at the execute stage. Otherwise the flags received may not be the correct ones, if the instruction in the decode stage changes them when it executes. In other words, *flags_req* always waits for any instruction currently in the decode stage to complete before it rises. The latch controller can provide this information (*decode empty*). The circuit that drives *flags_req* is thus a little more complicated than the AND gate shown.

When *flags_req* reaches the execute stage, if there is no instruction executing or if the executing instruction is not setting the flags, an acknowledgement (*flags_ack*) is sent. Otherwise the acknowledgement is sent after the instruction has finished executing and has updated the flags. Generally *flags_ack* is used as an indication that the flags are valid and it is delay-matched (*bundled*) with them.

In the condition check block, the instruction's condition is checked with the flags and raises *aFail* or *aPass* depending on the condition check result. These signals are generated by a delayed version of *flags_ack*, so that its rising edge reaches the AND gates when the check is complete. If the condition fails, the previous stage is acknowledged (*ir_ack*) but the instruction is not loaded into the decode input latch. Otherwise, the input request of the latch controller is raised and the instruction is decoded.

Extensions

Currently instructions following taken-branches can be discarded at the decode stage using a colour counterflow mechanism (described in section 4.3, page 58). This is done by comparing the incoming instruction's colour with a current colour stored in *decode* which is updated after each branch. Now that there is a mechanism which is used to discard instructions before loading the input latch, the colour can also be checked at the same time as the detection of conditional instructions. Instructions that have different colour are not loaded into the latches, saving more energy.

This idea is used for conditional instructions only: when the current flags are read from the execute stage, the new colour is also read and compared with the incoming instruction's. Thus, in the modified decode stage, an instruction can be discarded either because it fails its condition check or because it has the wrong colour, i.e. it is in the shadow of a taken branch.

5.2.2 Evaluation

The early condition checking design was implemented in Verilog as a variation of the AMULET3 model and was evaluated using the benchmark suite used throughout this work. Figure 5.4 shows the execution delay, energy consumption, and energy-delay product (EDP) for all benchmarks, normalised relative to the standard AMULET3.

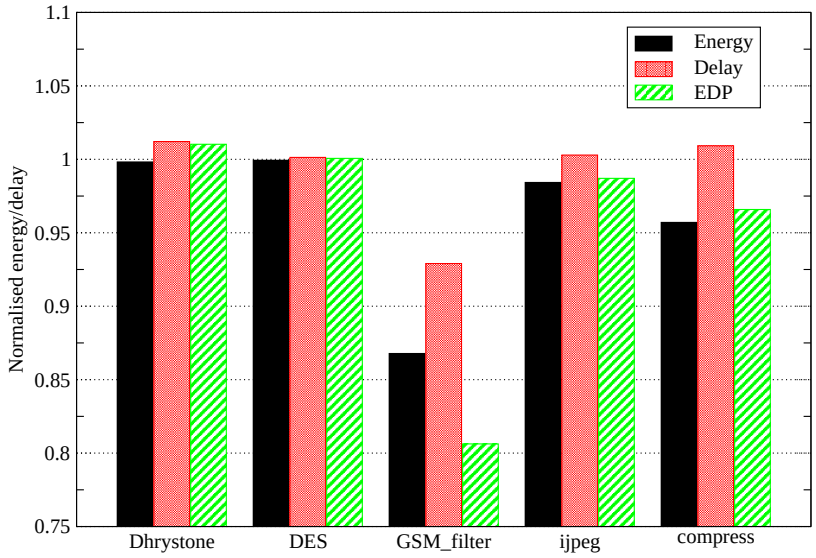


Figure 5.4: Early condition checking results.

Generally, the processor with early condition checking consumes less energy, but the execution time is increased. Of all benchmarks only compress shows the expected energy savings of 4% and the others, except GSM_filter, save less energy. This is because, for these benchmarks, most of the conditional instructions are branches. Branches don't read registers and thus expend less energy than the estimated average in the *registers* block.

Especially for GSM_filter, both the execution time and the energy consumption are lower than the original processor. The method seems to be much more successful for this benchmark than the others. The reason is that GSM_filter has a high proportion of conditional instructions that fail the condition check (about 30%). As these are discarded earlier, both time and energy are saved.

In all cases, except for Dhrystone and DES, the energy delay product (EDP) is lower for the modified processor, which means that proportionately more energy is saved than execution time is increased, so overall the technique is a positive step towards improving energy efficiency. Moreover it has considerable benefits

— in the order of 20% — for programs with a large number of conditional data-processing instructions, like GSM_filter, so it could be selectively activated only for such programs.

5.3 Energy overhead of pipelining

Ignoring the performance implications, pipelining makes a processor consume more energy than absolutely necessary to perform a task. Gonzalez and Horowitz [GH96] report a 30% energy overhead using a simplified energy estimation model. In this section some quantitative data are presented about how much this overhead is in AMULET3.

An interesting piece of information to have is how much energy the speculative instructions fetched following a (taken) branch consume, especially in the processor stages following the fetch: Thumb, decode and execute. In order to collect this information, a simple behavioural prefetch block was written in C using Synopsys' ADFMI — the C programming language interface to Powermill/Timemill simulators. The new prefetch block is linked to the rest of the processor which is still modelled as a layout-extracted netlist for high accuracy. It reads the program instructions from a previously prepared trace file which contains only the opcodes of the instructions that were executed — including those that failed their condition code test. Thus, no instruction addresses are generated and branch requests from the execution unit are acknowledged but ignored. The trace file is generated by dumping the instructions that are executed during a Verilog simulation.

With this method only the "right" instructions are fetched, so the energy consumption would be the ideal and it would provide an estimate of an upper limit of the energy that can be saved. This limit is valid if no other changes are made to the processor pipeline; some of the techniques described later do change the processor pipeline, so for these the energy limit established can only be used as an *indication* of the possible energy savings.

Table 5.4 shows the energy reported by Powermill simulating the ideal and full-processor versions running three benchmarks. In the "implementation" column, the energy consumption of the prefetch unit is subtracted from the total of the full-processor so that the comparison is fair. Branch prediction was enabled for the full-processor simulations; the energy consumption would have been significantly higher, especially for the pipeline stages following *prefetch*, as many more

Table 5.4: Energy overhead of pipelining.

Benchmark	Ideal	Implementation	Ideal / Impl.
Dhrystone	$140\mu J$	$174\mu J$	0.80
DES enc.	$604\mu J$	$647\mu J$	0.93
GSM_filter	$707\mu J$	$777\mu J$	0.91

Table 5.5: Energy overhead of branch prediction.

Benchmark	Branch prediction		disabled / enabled
	enabled	disabled	
Dhrystone	$213\mu J$	$200\mu J$	0.94
DES enc.	$759\mu J$	$728\mu J$	0.96
GSM_filter	$908\mu J$	$914\mu J$	1.01

speculative instructions would have made their way into these pipeline stages.

The results show that, even with branch prediction enabled, energy savings can be achieved. These can be as low as 7% for DES encode, which has very few branches, and as high as 20% for Dhrystone. It is clear that there is large variation in the potential savings, so it is essential for whichever techniques are developed to be dynamically controlled, so that they can be used only for programs which would really benefit from them.

In addition to the energy spent processing speculative instructions that eventually get discarded, pipelined processors use special hardware, such as branch prediction, to improve their performance. From the information collected in the power analysis of AMULET3, presented in table 5.5, it is clear that branch prediction has a negative effect on the energy consumption in some benchmarks. Since branch prediction is responsible for about 5% of the energy consumption of the processor core, the potential energy savings presented above could be extended by a similar figure when branch prediction is disabled. Thus up to around 25% of the energy consumption of the core can be attributed to the pipeline energy overhead and could be saved by the techniques presented below.

5.4 Token controlled pipeline occupancy

One way of controlling the processor's speculation due to pipelining, is to control the pipeline occupancy. In a synchronous processor this is easily achieved by incrementing a counter for each instruction fetched and decrementing for each

instruction committed. Unfortunately in an asynchronous processor due to lack of global synchronisation, other methods, like the one presented here, have to be used.

The proposed method is easy to illustrate in a simple pipeline like that of figure 5.5, where a FIFO of tokens is shown under the processing pipeline. The first stage, must acquire a token before it can proceed to send data to the next stage. At the other end of the pipeline, the token is returned to the FIFO. The number of tokens that are present in the system determines the maximum occupancy of the pipeline (assuming that there are not going to be more tokens than the actual number of pipeline stages). So this mechanism, called a *token FIFO*, gives the capability to vary the number of operations in the pipeline from none (stalled) to its maximum capacity.

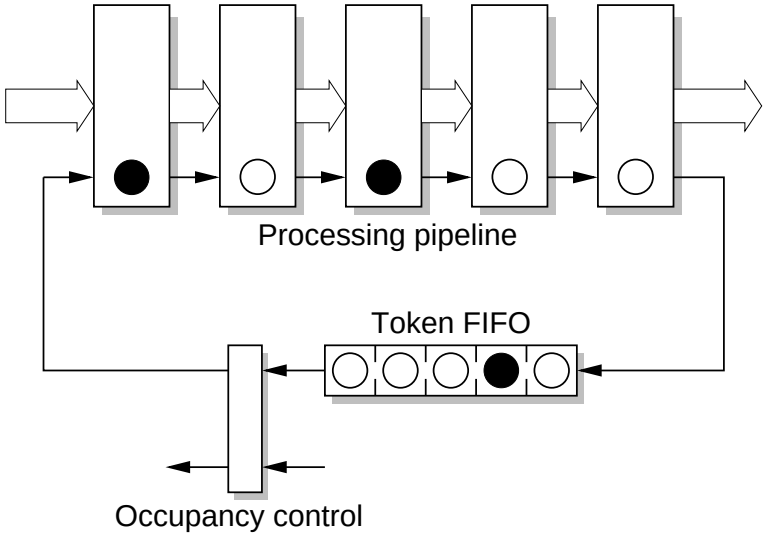


Figure 5.5: Token controlled serial pipeline.

As an alternative method of controlling the pipeline occupancy in an asynchronous pipeline, the first stage can be made to delay processing each data item by some time, controlled by a programmable delay circuit. This spaces out the data items in the pipeline, effectively controlling its occupancy. The advantage of this method is that only one stage is affected, so the implementation cost is small. On the other hand, there is no direct relation of the programmable delay in the first stage to the pipeline occupancy, because of the elasticity of the pipeline. A large delay will guarantee that the occupancy will be (at most) one, but it is

hard to estimate an appropriate delay which sets the occupancy to a fixed intermediate level. In addition, the programmable delay line required by the method must have a very large range. This will make it both expensive to build and energy consuming to run. As the disadvantages outnumber the advantages for this method, the token FIFO method was selected for implementation.

5.4.1 Incorporating token FIFO into a processor

The first stage in all processor pipelines is a (pre)fetch stage. In order to use the token FIFO technique in a processor, this stage must be modified so that it waits for a token from the FIFO before it can start operating. It is harder to identify a suitable "last" stage in a modern processor, as the pipeline splits into multiple parallel pipelines in superscalar processors. In general the token should be returned at the stage where the next instruction to be fetched is known, that is after the current instruction has been decoded and, if it is a branch, its target has been resolved.

If the token controlled pipeline method is used in a single-issue processor, like AMULET3, the stage that is used to release the tokens is *execute*. Although there is one stage following *execute* (or more, for data transfers), there is no need to delay returning the token until the very end of the pipeline. The reason is that conditional instructions, and most importantly, branches, are resolved at the beginning of the execute stage, so returning the token later than this stage will not save any more energy.

For correct operation, especially when only one instruction is permitted in the pipeline, after a taken branch returns the token, its latency through the token FIFO should be longer than the time it takes the branch request to reach the prefetch unit. The reason is that on arrival of a token, the prefetch unit fetches the sequential instruction if there is no branch request pending. Thus if the token arrives before the branch request, another instruction from the branch shadow will be fetched instead of the branch target, as it would have been expected for a processor with a pipeline occupancy of one. To ensure the correct operation, the execution stage is made to complete its handshake with the prefetch stage before the token is released.

In addition to interfacing the token FIFO with the *prefetch* and *execute* stages, the incorporation of token-based pipeline occupancy control in AMULET3 requires minor modifications to some other parts of the processor. These are needed

for handling the tokens taken by instructions that are discarded before reaching the execution stage, instructions with multiple cycles spent in the execution stage, and handling indirect PC loads.

Discarded instructions

The handling of discarded instructions is regarded as an advantage of the asynchronous design style. In a synchronous processor, the common practice is to add a "valid" bit with the instruction and pass it from each pipeline stage to the next. When the valid bit is not set, the instruction is regarded as a *nop* and it is not processed. When an instruction is to be discarded, the valid bit is cleared when the instruction is passed to the next stage. Only if the valid signal is available early enough, can the pipeline register be clock-gated to save energy. Unfortunately, in most cases, the valid signal is on the critical path, so clock-gating is rarely triggered. Conversely, in an asynchronous processor, discarded instructions just disappear so they are never passed on to the next stage; an approach which is more energy efficient.

With the introduction of the token mechanism, dropping discarded instructions at any pipeline stage is no longer possible, because the token acquired at the *prefetch* must be returned or the processor will eventually be starved of tokens. If the tokens can be returned in any order, the solution is relatively simple; just modify the token FIFO so that it has many "heads" which can collect tokens concurrently and merge the token flow into a single FIFO by the "tail" of the token FIFO. If the order must be retained, the tokens must be carried forward from each stage to the next, until they reach the one that returns them to the token FIFO. The latter solution would require modifications to the control part of each pipeline stage involved, but this overhead is minimal: just a handful of gates when it was implemented in the decode stage of AMULET3 as a test.

Strictly speaking, tokens should be returned in the order they were acquired. Otherwise there can be situations where the processor speculation is not controlled as intended. Assume, for example, that a pipeline is restricted to two tokens at a time and that of the two instructions in the pipeline the first is a branch and the second gets discarded before the branch changes the instruction flow. If the discarded instruction's token is allowed to return to the token FIFO before the branch has executed, another instruction in the branch shadow will be fetched. Clearly this behaviour is not wanted, as this latest fetch will be wasting

energy. Such situations are not common though and usually instructions can be discarded only at a few pipeline stages. Thus, depending on the processor, it can be a better solution for the implementation to allow the tokens to be returned in any order.

In AMULET3, the earliest place an instruction can be discarded is at the decode stage. The only case when this happens is after a branch is taken; the decode stage is informed with a delay of one instruction; it then starts discarding instructions until those from the branch target arrive. Thus *decode* starts returning tokens from the second instruction following a branch. By this time the prefetch unit has been notified of the branch, so there can be no fetching of instructions at the branch shadow due to *decode* returning tokens earlier than execute. So, for AMULET3, the solution of having a token FIFO with two heads is used. The arbitration between them is handled internally as described in section 6.1.

Multicycle instructions

In AMULET3 there are two instruction types that take more than one cycle to execute: long multiplications and multiple data transfers.

Multiple data transfers use the execution unit only for one cycle, to check their condition codes and calculate the first address; the data transfers and address incrementing/decrementing are handled by the data transfer unit. The PC can be one of the registers to be loaded, effectively causing a branch. This is examined in the first cycle and the appropriate address, if any, is given to the fetch unit as a special case of branch (see also indirect PC loads, below). Although the instruction may take a number of memory accesses to finish, the only time it can change the instruction flow — ignoring exceptions — is during this first cycle. So, by the end of this cycle, the speculation ends and it is safe to return the token.

It is possible for one of the addresses to cause a memory exception (e.g. a page fault), in which case the instruction(s) fetched because of the early returned token will be discarded, wasting energy. The frequency of memory exceptions is deemed low: many embedded processors don't use memory protection/translation at all, so they don't cause memory exceptions. An alternative, more conservative, implementation would require more hardware overhead, so the simple solution of returning the token at the first cycle of these instructions was selected.

Long multiplications take two cycles in the existing processor. A multiplication cannot change the instruction flow of the processor (using the PC as the destination register has an undefined behaviour) so the token can be returned at the first cycle without risking a speculative fetch. In this case the execution block is activated for the second cycle, so the implementation must not pass a second token to the FIFO.

Indirect PC loads

The implementation of memory PC load is peculiar to AMULET3. Instead of performing a standard data load and then passing the loaded value to the prefetch unit, the address is given to *prefetch* which does the load through the instruction memory port. The loaded value (the new PC) is then used as the address to load the next instruction. This was implemented mostly to speed up multiple load instructions, which are frequently used to restore the registers and the PC from the stack, when returning from a function call. The PC can be loaded concurrently with the other loads and the instruction flow from the target address can be restored quickly [GFC99].

For the token-controlled implementation, only the first cycle (PC load) acquires a token; the second cycle reuses the previous token. This is implemented by deferring the acknowledgement of the receipt of the token until the second memory cycle is complete, i.e. only when an *instruction* is loaded from the memory.

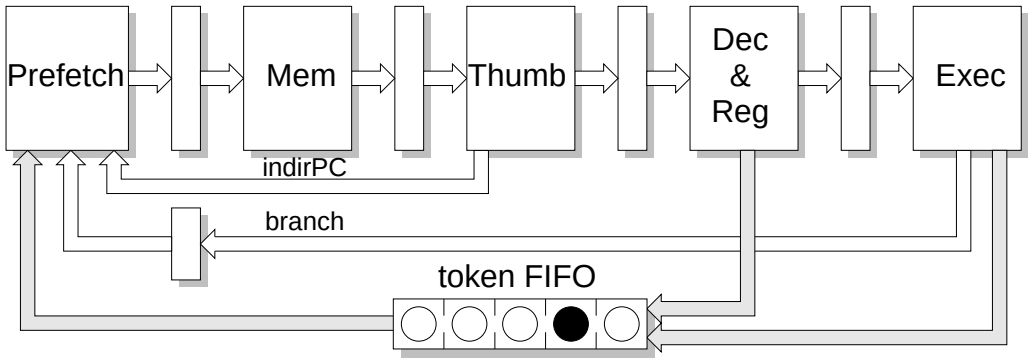


Figure 5.6: Token controlled AMULET3 processor core.

Figure 5.6 shows a simplified block diagram of the processor with the token FIFO. The block arrows are handshake channels and the direction of the arrow is the same as the direction of the request signal. The only change to the simplified

linear pipeline example of figure 5.5 on page 87, is that tokens are returned to the FIFO from two pipeline stages. Arbitration is required between these two token inputs which is implemented in the token FIFO block. The token FIFO circuits and some minor circuit changes in the control units of *prefetch* and *execute* that were made to incorporate the new handshake channels, are presented in section 6.1.

5.4.2 Evaluation of pipeline occupancy control

The modified processor was evaluated using the experimental setup described in section 5.1. Figure 5.7 shows the normalised power, energy, and execution time for each pipeline occupancy, with branch prediction disabled, for a number of benchmarks. The pipeline occupancy was kept constant for the whole duration of the execution. In addition to the usual power, energy and execution time metrics, *extra work*, as defined by Manne *et al.* [MKG98], is used as an indication of wasted energy in the various pipeline stages. Extra work is the number of extra instructions entering the pipeline stage relative to the number of instructions that are finally executed. In the evaluation, the latter category includes only the instructions that passed their condition test, so there is some extra work even at the execute stage. Figure 5.8 shows the extra work per pipeline stage, for each occupancy, as a percentage of the executed instructions.

DES encryption has only a few branches, so the extra work in any pipeline stage is insignificant; the same holds for the energy benefits through pipeline adaptation. GSM filter was optimised by the compiler so that many branches were replaced by conditional instructions, which is why there is a lot of extra work at the execute stage from instructions that fail their condition test. For this benchmark early condition checking, described in section 5.2, gives more significant energy savings than controlling the pipeline occupancy. The other benchmarks have a significant amount of extra work, which was successfully removed by lowering the pipeline occupancy and this is reflected in the energy graphs as well.

In figure 5.7 it is apparent that, as the occupancy decreases, the execution time increases and, consequently, the power consumption decreases. The token FIFO method does save energy, as can be seen in the energy curves; these curves would have been flat if the reduction of power consumption was only due to the slower execution rate.

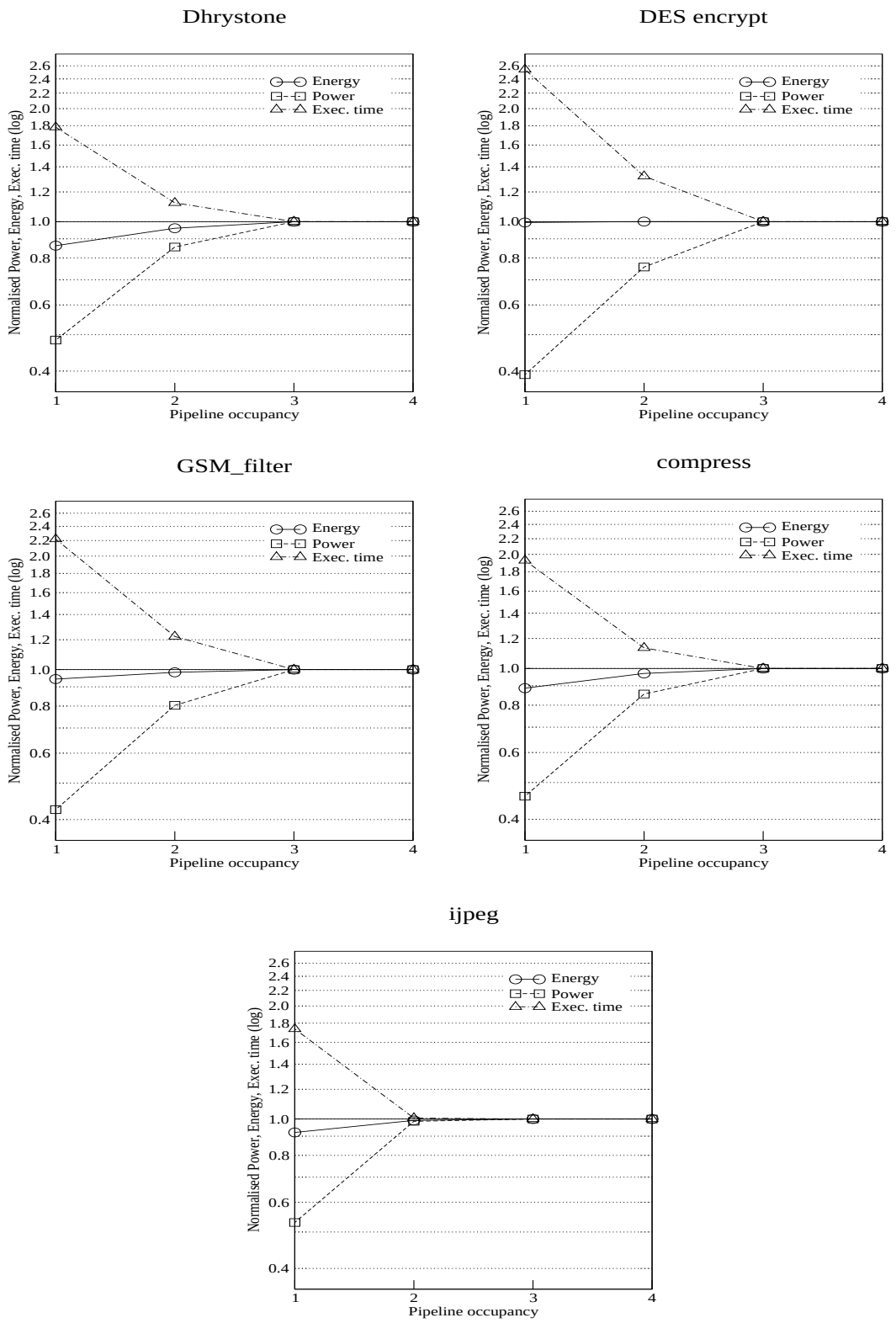


Figure 5.7: Power, energy, exec. time of token-based pipeline occupancy control

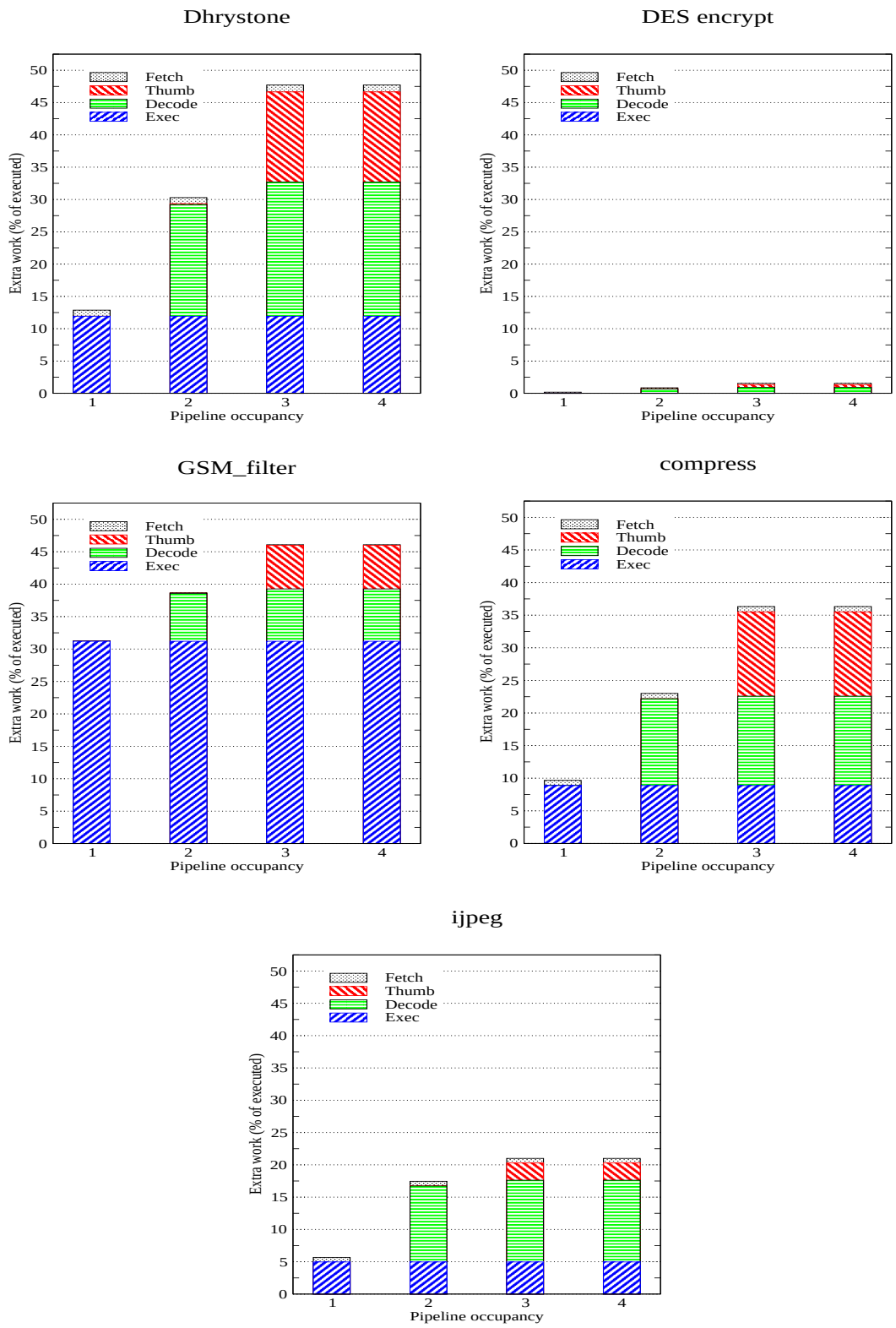


Figure 5.8: Results (extra work) of token based pipeline occupancy control

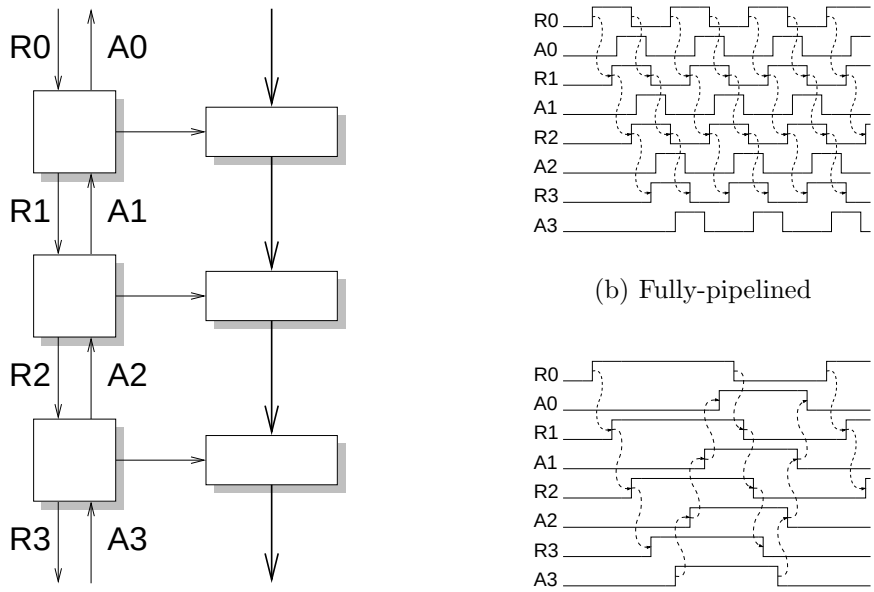
Only when the pipeline occupancy is low (1 or 2) are there reasonable energy savings. At these occupancies the execution time is roughly doubled for all benchmarks. From the energy savings achieved it is clear that this method cannot be used in performance critical applications, as the energy delay product gets worse when the method is applied. Its usefulness comes when there is idle time in the execution of a program, where the energy savings can be exploited without an impact on the user-perceived performance.

In the results presented here the pipeline occupancy was kept constant for the duration of the execution. The token FIFO mechanism could also be used as a simple way to dynamically control the pipeline occupancy at runtime. Different programs and even parts of the same program can behave differently and can use the available range of pipeline occupancy levels to reach the energy/performance level that is required. In addition, as the operating environment or the available time to complete a task changes, the pipeline occupancy can be adjusted accordingly. The time for inserting or removing tokens can be decided by a hardware monitoring block or simply by software writing the required level of pipelining into a system register. The latter approach is used in the simulations presented here because of its simplicity and generality of use. Chapter 7 investigates some dynamic pipeline controlling algorithms, but these are based on the technique presented in the following section.

5.5 Adapting the pipeline structure

The previous section described a method of controlling the occupancy of the pipeline to trade speed for energy reduction by removing unneeded, speculative instruction fetches. The same effect can also be achieved by dynamically changing the pipeline depth.

A pipeline is defined by the position of the pipeline registers/latches in the circuit. As there is no way to move these registers after the circuit is fabricated, the only alternative is to control when they are transparent or opaque. When a pipeline latch is made 'permanently' transparent, its two neighbouring pipeline stages are effectively joined in one stage. With this method the pipeline structure can be altered dynamically to suit the power and performance levels required by the system at any particular time. In a micropipeline-style asynchronous circuit the latches are controlled by latch controllers so, in order to change the



(a) Simple pipeline

(b) Fully-pipelined

(c) Collapsed pipeline

Figure 5.9: Collapsed pipeline behaviour

pipeline depth, reconfigurable latch controllers, that can be either 'permanently transparent' (*collapsed*) or 'normal', are needed.

Figure 5.9(a) shows a simple micropipeline with no processing between the pipeline latches. Figure 5.9(b) shows sample waveforms for the request-acknowledge signals, when the pipeline is operating at its maximum capacity/throughput. For comparison, figure 5.9(c) shows waveforms of the same pipeline, but this time the pipeline latches are collapsed.

In a collapsed pipeline the input is not acknowledged (*A0*) until the output acknowledgement (*A3*) is raised and propagated back to the start. Thus the whole set of pipeline stages in between behaves like a single, slow pipeline stage. The request signals of the intermediate stages are just delayed versions of the input request and similarly the intermediate acknowledgements are delayed versions of the output acknowledgement. It is clear that in the collapsed configuration there is only one operation present in the pipeline. In contrast, in normal, fully-pipelined configuration (figure 5.9(b)) there can be up to three parallel operations at the same time. All combinations between these extremes are possible, by collapsing any number of pipeline stages.

5.5.1 Collapsible latch controllers

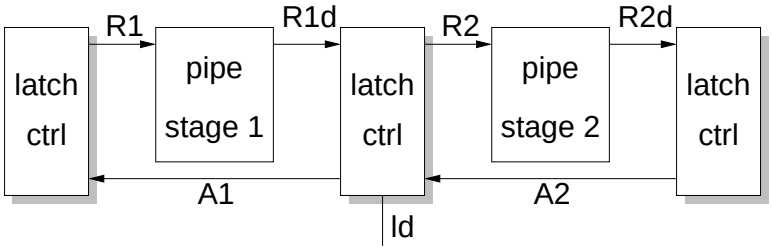
For a standard latch controller, when an input request arrives and there is no pending operation at the output (downstream), the latch loads the data and then closes, the input is acknowledged, and an output request is made. If the output is busy, the input is not acknowledged, nor are the latches opened, until the current output operation has finished. There are a number of variations, depending on how long the data should be kept valid when the signals return to zero, and if the latches should be already open before the input request arrives, if the output side is ready. These are described in more detail in section 6.2, where the circuit implementations of the collapsible latch controllers are given.

In the collapsed operating mode, the latch controller must 'pretend' that it only connects the inputs and outputs together, as if they were the same wire, while always keeping the latch transparent. So, an input request will be passed on to the output, and will only be acknowledged when the output side is acknowledged. In a series of collapsed latch controllers, the first will have to wait for the last one to be acknowledged (and propagated back) before it can give its acknowledgement.

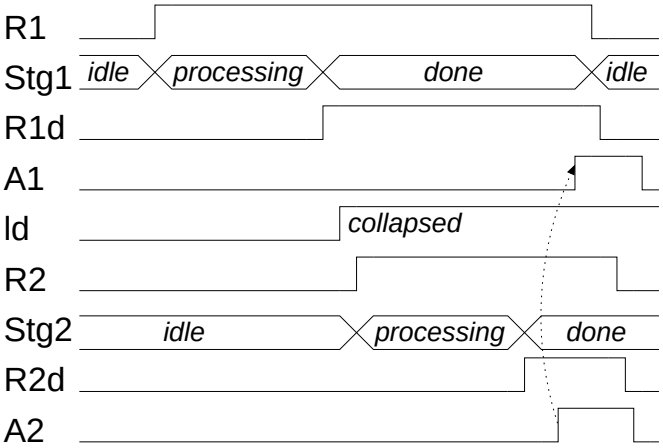
The pipeline latches cannot be collapsed at any time. Figure 5.10 shows a two stage pipeline where the latch controller in the middle becomes collapsed and the signal waveforms at the time of collapse. The correct time to change from normal mode to collapsed is when a new input request is received and the latches are about to be loaded. At this time the downstream pipeline stage (*stg2*) has finished processing the previous data so it is safe to merge the two pipeline stages. After the collapse the upstream stage is not acknowledged (*A1*) immediately, as it would normally be, because the new, joined, pipeline stage has not finished processing the current data; its second half still has work to do. When the output acknowledgement (*A2*) is received, which means that processing at the second half of the stage has finished and the output data are safely stored at the next latch, the upstream latch controller is acknowledged, notifying the upstream stage that the whole pipeline stage is finished.

Splitting a previously merged pipeline stage also happens when a new request is received. After the latches have loaded the input data, the load enable signal is deasserted, making the latches opaque and acknowledging the upstream stage. Concurrently the output request is issued as usual. The pipeline stages are now split and the first stage is free to receive the next data package for processing.

As the operating mode of a latch controller can only change at the time of



(a) A 2-stage pipeline



(b) Collapsing pipeline waveforms

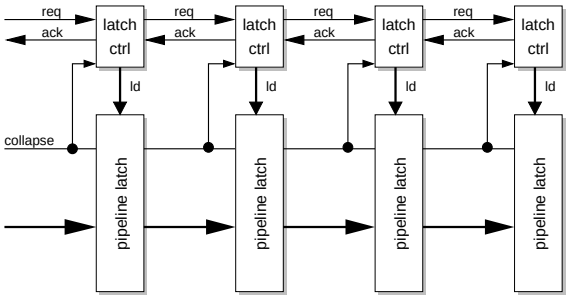
Figure 5.10: Timing of pipeline collapsing

new requests, the signal that sets the latch controller mode, *collapse*, must be locally synchronised with the input request signal. This makes it hard to have a global *collapse* signal[2]. The solution (similar to [LGB99]) is to have *collapse* bundled with the rest of the data signals that are latched at the pipeline stage.

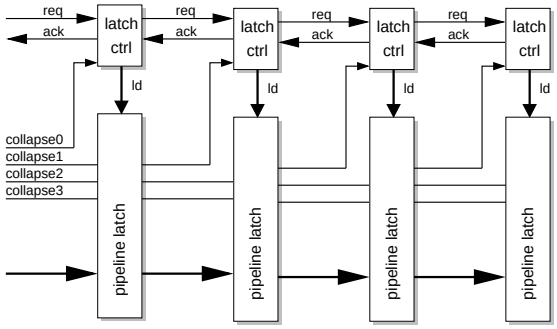
One possible implementation would be to have the first pipeline stage produce the *collapse* signal for the first latch controller, which is then propagated to subsequent controllers down the pipeline (figure 5.11(a)). En-route, depending on local conditions, the collapse signal can be changed by the pipeline stages. Alternatively, for independent control of each latch controller, the first stage can produce a *set* of collapse signals which are latched and propagated to the appropriate latch controller (figure 5.11(b)). In both cases the first pipeline stage is the most suitable to generate the signals since, in asynchronous pipelines, it is

[2]Unless the latch controllers are reconfigured while the pipeline is not operating.

easy to transmit signals following the pipeline flow but considerably harder in the other direction.



(a) Serially controlled



(b) Individually controlled

Figure 5.11: Collapsible signal distribution.

5.5.2 Integration in the processor

Figure 5.12 shows an abstract block diagram of the AMULET3 core showing which pipeline latches were changed to use collapsible controllers. Only the latches marked in the figure with grey waves need to be made collapsible; all others, including those not shown in the figure, are unaffected. Although the concept of collapsible pipeline latches may seem more complicated than the token FIFO method (section 5.4), fewer changes are required.

As can be inferred from the way the collapse signal is drawn in the block diagram, there is only one such signal propagating from the prefetch stage down the pipeline to the execute stage, in the same fashion as explained earlier in figure 5.11(a). One simple way to put *collapse* under software control, is to

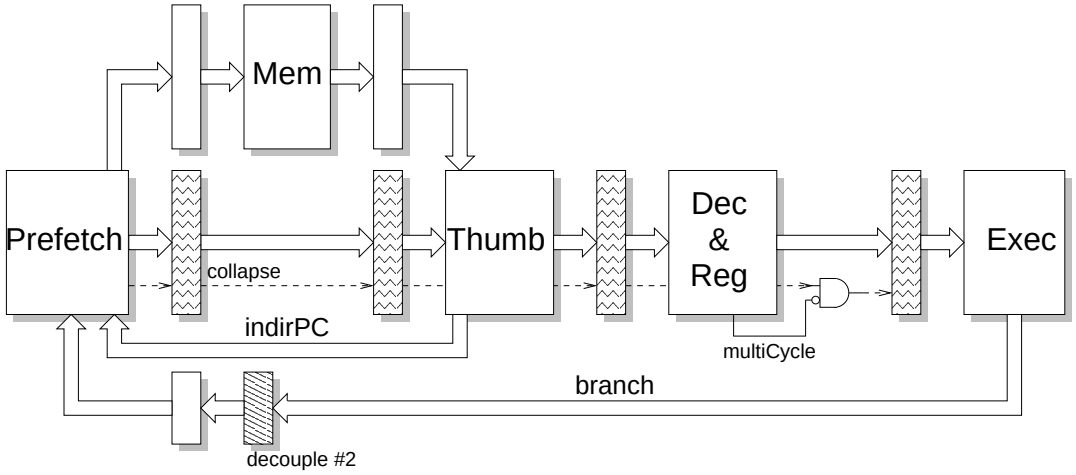


Figure 5.12: Block diagram of AMULET3 with collapsible pipeline latches.

associate it with one of the unused bit-fields of the processor's current program status register (CPSR). An MSR instruction can then be used to write to that CPSR field, setting or resetting the collapse signal. In AMULET3, the MSR, after setting the status register, behaves like a special form of branch[3] so that the new status, e.g. a processor protection level change, can start taking effect at the prefetch stage of the next instruction. Using the same mechanism the collapse signal will be set at the prefetch stage and then spread to the rest of the pipeline bundled with the next instruction.

Nearly all the pipeline latches of the processor are modified to be of the collapsible type. A notable exception is the decoupling latch, on the branch path from execute to prefetch, which must not be collapsed. If it were, in a configuration where all the latch controllers are collapsed, back-to-back branches would deadlock: the first branch would write its target address to the decoupling latch and get an acknowledgement, allowing it to finish. The prefetch unit would then fetch the next instruction which is also a branch, but it would not acknowledge the decoupling latch until the instruction has executed. When the execution unit tries to store the target address to the decoupling latch it will still be occupied by the previous branch, causing the processor to deadlock. To break the deadlock, a second decoupling latch is added in the branch channel. This latch can be a standard latch, but then the branch latency will be increased, which is undesirable. Alternatively it can be a collapsible latch, which works in the opposite way to the other latches in the pipeline: it is collapsed when there is at least one other

[3]It branches to the next instruction, flushing the pipeline.

latch in the pipeline that is not collapsed. In other words, it works as a latch only when all the other latches are collapsed, which is when the deadlock could happen.

The latches at both sides of the instruction memory/cache must also not be collapsed, to prevent a possible deadlock in the memory system. Although the system memory appears to be dual-ported, in reality it is composed of a number of single-ported SRAM blocks. If the latches around the instruction port were collapsed, the accessed SRAM block would not be released until the instruction is finished. If the instruction needed to access data in the same block through the data memory port, it would be blocked and the processor would deadlock. Although these latches cannot be collapsed, the prefetch stage can still be joined with the memory fetch stage (and/or the mem. fetch with Thumb) by collapsing the corresponding latches that hold the control signals (under the memory in figure 5.12). These parallel latches are independent, but for a stage to proceed with the next operation, it must have received acknowledgements from both of the latch controllers. Thus if one of the latches is collapsed, the processor behaves as if both were collapsed.

Instructions discarded at the decode stage require no special treatment; they just disappear from the pipeline as in the original implementation. Instructions that require multiple execution cycles temporarily disable the collapse signal for the decode-execute pipeline latch (using the AND gate in figure 5.12). This reinstates the pipeline stage between decode and execute for the duration of the multi-cycle instruction, so that multiple execution cycles can be performed as required. This demonstrates the flexibility of this pipeline control method where locally generated control signals can change the pipeline behaviour.

5.5.3 Evaluation

To evaluate the effect of changing the pipeline structure on the performance and energy consumption, the AMULET3 Verilog model was modified to support collapsible latch controllers, as outlined above. In addition, experiments with the processor having individually controlled pipeline latches were also performed. These experiments show how each pipeline latch affects the energy consumption and the performance when it is collapsed. Since the processor with a single collapse signal is just a special case of the individually controlled version, only the results from the latter are presented here. All simulations and measurements

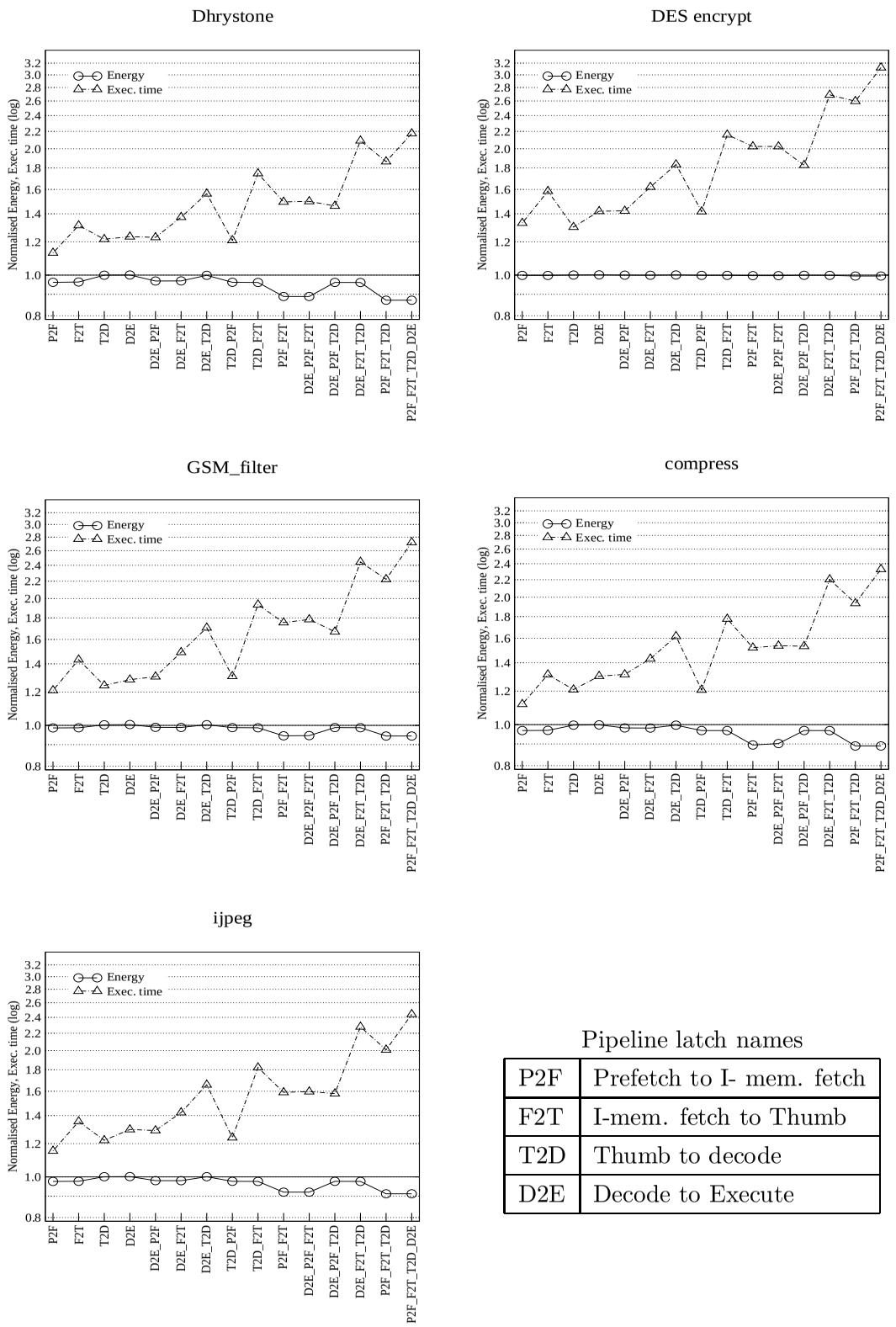
were performed as described in section 5.1.

There are four pipeline latches in the processor core that can be collapsed so there are 15 possible configurations in addition to the fully pipelined. Of these there are 4 combinations with one collapsed latch, 6 with two, 4 with three and 1 with all latches collapsed.

Figure 5.13 shows the energy and performance measurements, normalised relative to the fully-pipelined configuration (i.e. no collapsed stages), for all the configurations grouped by the number of collapsed latches. As expected, no configuration is faster than the fully-pipelined configuration and generally, as more latches are collapsed, the performance deteriorates. An interesting observation is that there are significant differences amongst the results of configurations where the same number of latches are collapsed, but at different places in the pipeline. Moreover the performance of each configuration relative to the others is roughly the same across all benchmarks.

With only one collapsed latch the best results are achieved by joining the first two stages, *prefetch* and *memory fetch*. Collapsing the memory to Thumb latch achieves a similar energy consumption, but at a lower performance. The other two combinations just increase the execution delay without saving energy.

Although these results might at first seem unexpected, there is a simple explanation. In fully-pipelined mode usually three instructions are present in the AMULET3 pipeline, meaning that for every branch the two instructions following it are wrongly fetched and waste energy. When one of the latches near the end of the pipeline becomes collapsed (e.g. D2E), the formed pipeline stage becomes the bottleneck and dictates the new pipeline throughput rate. Immediately after the collapse, the first stage will continue inserting instructions at its own rate until the pipeline becomes full. From that time the throughput of the whole pipeline will adapt to that of the slowest stage. Because the slow stage is at the end of the pipeline, although the pipeline depth is reduced, it will become more occupied during the time it takes to adjust the throughput rate. In these experiments the number of present instructions continued to be three, thus each branch is still followed by two instructions, so there is no energy saving. On the contrary, when the collapsed latch is near the start of the pipeline, the rate at which instructions *enter* the pipeline drops, decreasing the pipeline occupancy. This means that fewer instructions are discarded when branches happen, leading to energy savings.



Pipeline latch names

P2F	Prefetch to I- mem. fetch
F2T	I-mem. fetch to Thumb
T2D	Thumb to decode
D2E	Decode to Execute

Figure 5.13: Normalised energy, exec. time for all configurations of collapsible latches.

In view of the above analysis, it comes as no surprise that the best configuration for two collapsed latches is the one where the first two latches are collapsed. The same happens for configurations with three collapsed latches.

Starting from the configuration where the first three latches are collapsed, collapsing the remaining latch increases the execution delay for no energy savings. This means that when a branch is passed from the combined '*prefetch–decode*' stage to *execute*, it is processed and a request is sent to *prefetch* before the combined stage has time to start a new fetch. Examination of the signal waveforms shows that the time it takes for the acknowledgement to propagate from the *decode-execute* latch to the prefetch block, is longer than the time needed to process the branch at the execute stage and send a branch request to *prefetch*[4]. Thus the lowest energy consumption can be achieved by collapsing the first three pipeline latches only.

Each benchmark achieves different maximum energy savings by this technique. DES encode, with its low number of branches, has only marginal energy savings. The other benchmarks can save from 8% (GSM_filter) to 13% (Dhrystone).

5.6 Summary

This chapter presented three methods for controlling the speculation in an asynchronous single-issue processor. In addition the experimental setup is described and the energy overhead of pipelining is investigated.

The first method deals with conditional instructions and saves energy by not allowing them to proceed in the pipeline until their condition is checked. On average a small improvement was measured in the energy delay product of AMULET3 when this technique is enabled. In programs with a lot of conditional instructions, such as GSM filter, the technique not only manages to save energy, but it also improves the execution time.

The other two methods control the occupancy of the processor by using a token-based feedback mechanism, or by restructuring the pipeline itself by collapsing some pipeline latches. Energy savings of up to 16% were measured.

[4]The target address is calculated at *decode*, so all that needs to be done in *execute* is to check the condition, if any, and send a request to *prefetch*.

Chapter 6

Circuit-level implementation

This chapter elaborates on the speculation control techniques presented in chapter 5 by providing gate-level circuit implementations of the token FIFO and the collapsible latch controllers.

6.1 Token FIFO implementation

The token FIFO mechanism of controlling the occupancy of a pipeline was described at the architectural level in section 5.4. Figure 6.1, a slightly more detailed version of figure 5.6, shows a top-level diagram of AMULET3 modified to support this mechanism. Three stages of AMULET3, namely *prefetch*, *decode* and *execute*, are connected to the token FIFO and thus require small modifications. The detailed circuit implementations of these modifications and the gate-level design of the token FIFO itself are presented in the following pages.

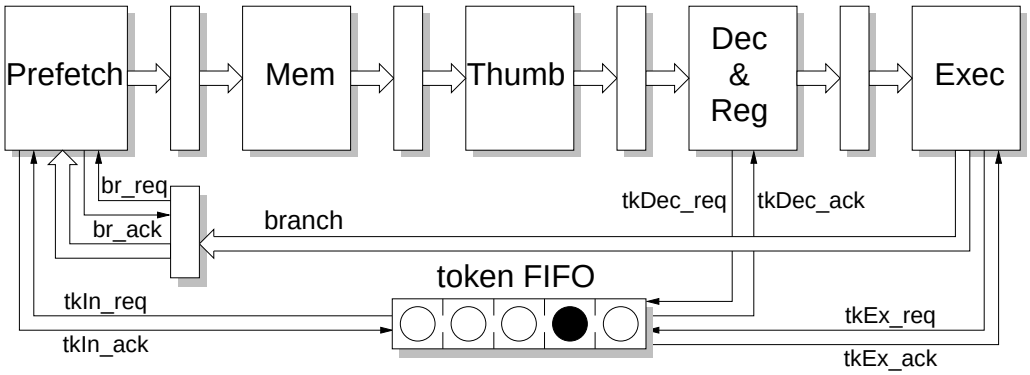


Figure 6.1: Token controlled AMULET3 processor core.

6.1.1 Prefetch

Figure 6.2 shows the implementation of the token FIFO interface in the prefetch unit. Each time the prefetch unit starts the process of selecting the next instruction to fetch, there are two possible options: the consecutive instruction (*R2*) and, sometimes, a branch target (*R1*). The latter is signalled by a request from the branch-decoupling latch, *br_req* while the consecutive instruction is 'automatically' selected if, after the current address is generated (i.e. *ack*=1) , no branch is pending.

Mutex makes the central decision if a consecutive (*seq_req*) or a branch target instruction (*branch_req*) will be fetched. Whichever input arrives earlier at the mutex, the corresponding action will be taken. If both input signals arrive simultaneously, the mutex will assert only one of the outputs, taking an unbounded time to resolve which one, but without falling into a metastable condition.

A token must already be available before the mutex operates. The choice to make both 'contestants' wait for a token before they can acquire the mutex is deliberate, so that the decision is made after the token is available and not earlier. If there are no tokens in the FIFO and a taken branch has just been executed, the branch request will reach the prefetch unit before the released token (as explained in section 5.4.1), so that the branch target will be able to win the arbitration. If, in the above situation, the token existence was checked after the mutex's decision, the consecutive instruction would have already won the arbitration and

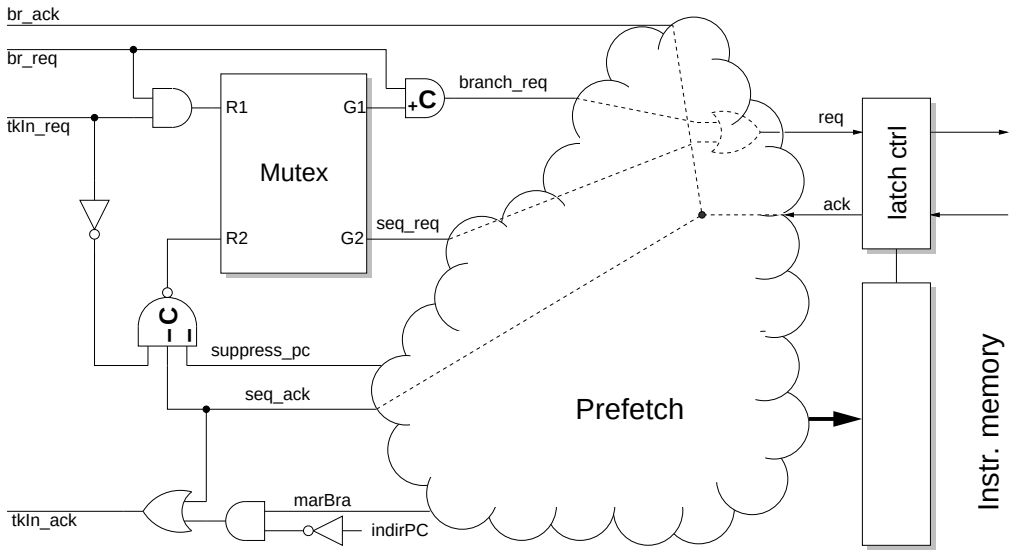


Figure 6.2: Prefetch - token FIFO interface.

would just be waiting for the token to proceed to the processor pipeline. This is clearly not the desired behaviour.

When a branch request (*br_req*) and the consecutive instruction simultaneously become ready, a just-arriving token should generate a request at the 'branch input' (*R1*) of the mutex earlier than the other input. The choice of having the token signal (*tkIn_req*) go through an AND gate for the branch target, but through a C element for the sequential instruction, was made with the intention to put a handicap on the consecutive instruction. This assumes that an AND gate has a shorter propagation delay than a C element. As this is not generally true, some extra delay could be added at the output of the C element generating the mutex request for the sequential instruction.

The token is acknowledged either when the consecutive instruction is complete (*seq_ack*) or, for branch targets, when the memory address register is loaded with the branch target (*marBra*) and the branch is not for an indirect PC load. In the latter case, a memory read is performed at the branch target address and the data is returned back to the fetch unit via the indirect PC channel[1]. At this time the branch is acknowledged and the newly loaded PC address is used for what appears to be a normal consecutive instruction. After this instruction is acknowledged, the token used for the indirect branch is acknowledged.

6.1.2 Decode

As explained in section 5.4, sometimes *decode* has to return tokens to the FIFO. This can happen to instructions discarded following a taken branch if the pipeline occupancy is relatively high. Figure 6.3 shows the circuits added to the decode stage, that make up the simple interface to the token FIFO.

Instructions discarded at *decode* raise the (internal) *bypass* signal. Normally (see inset of the same figure) this signal is used to communicate the completion of instruction processing at this stage (*decInAck*) to the pipeline latch controller at the input. For the token FIFO implementation *bypass* is used to generate a request to give a token to the FIFO (*tkDec_req*). Instead of *bypass*, the acknowledgement of the FIFO (*tkDec_ack*) is used to generate *decInAck*.

[1]Removed from figure 6.1 for clarity, see fig. 5.6 on page 91.

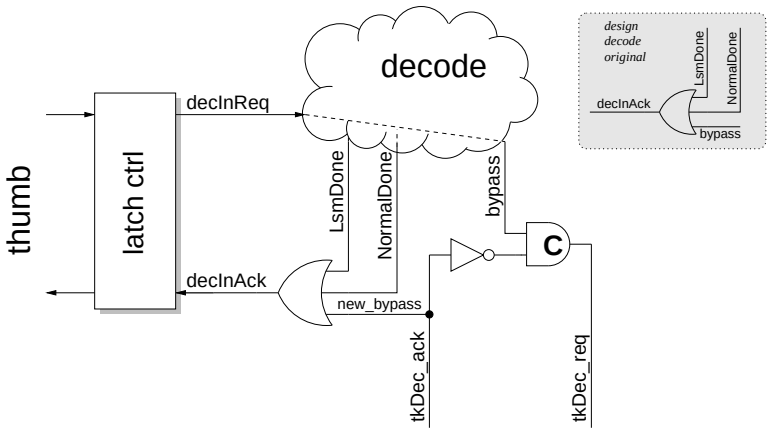


Figure 6.3: Decode - token FIFO interface.

6.1.3 Execute

Execute needs to interact with the token FIFO in order to return the tokens acquired at the beginning of the processor pipeline. Figure 6.4 shows the token-FIFO interface circuits added to the execute stage.

The appropriate time to return a token (*tkEx_req*) is when an instruction completes its execution, that is when the acknowledge signal (*realExecAck*) is raised. In order to make sure that the token is sent before the next instruction moves in, the final execution acknowledge (*execAck*) is generated after the token has been received at the FIFO.

The operation sequence is the following: the execution unit completes (*realExecAck*) and sends the token to the FIFO (*tkEx_req*). When the FIFO acknowledges (*tkEx_ack*), the final acknowledge of the execute stage (*execAck*) is raised. Shortly afterwards, in the return to zero phase, *realExecAck* becomes zero, removing the

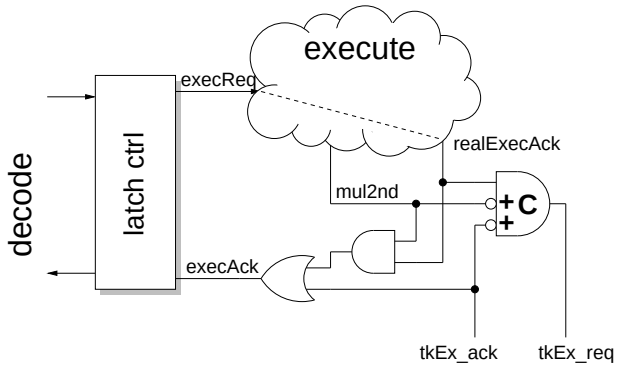


Figure 6.4: Execute - token FIFO interface.

token request, which in turn clears *execAck*. At this time the pipeline latch controller is ready to send the next instruction to be executed.

The only case when a token should not be sent is for the second cycle of a long multiplication, so this condition is added to those driving the token request, as seen in the circuit schematic (fig. 6.4). Multiple loads and stores only use the execute stage once, at the first data transfer, and return their token at that time; no extra circuits are needed to handle them.

Using the above mechanism increases the time spent in the *execute* stage by the time taken by the token handshake. Since the token FIFO does no processing, it moves the tokens fast. So at the rate at which tokens are returned (the fastest execute response time is 3.4ns), the head of the FIFO will always be ready to receive a token. Thus the handshake delay overhead is minimal, since *execute* does not get blocked waiting for an empty place in the FIFO.

6.1.4 FIFO implementation

The abstract figure 5.5 on page 87, describing the token FIFO concept, shows a simplified version of the FIFO. As explained in section 5.4, the implementation for the specific processor requires the FIFO to be able to sometimes accept two tokens simultaneously. In order to accommodate this requirement, the structure of figure 6.5 is used.

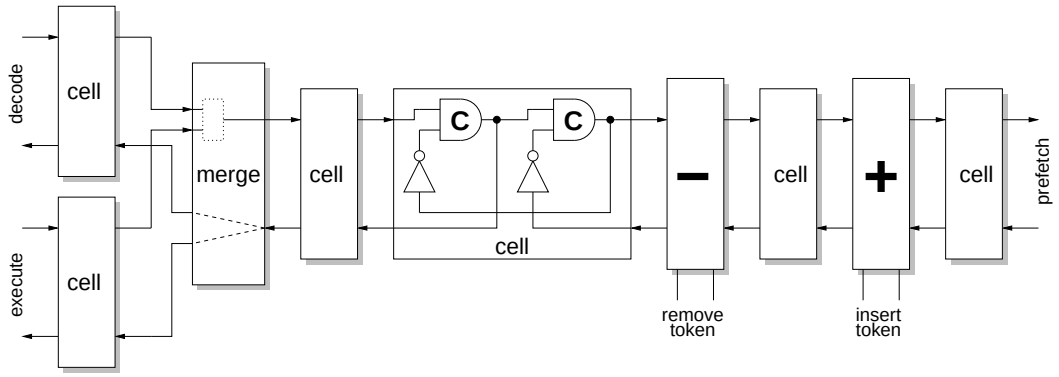


Figure 6.5: Token FIFO block diagram.

The head of the token-FIFO is split into two single-element FIFOs, one for each pipeline stage that returns tokens: *decode* and *execute* in this case. One element is deemed enough for each of the two 'heads', as the token-FIFO is able to shift tokens much faster than the processor pipeline can produce them.

Since there is no processing involved in the token FIFO, tokens move rapidly in contrast to the processor pipeline stages for which shifting the instructions is (or must be) just a small part of the stage's 'cycle time'. The intention is that the processor should never be blocked because of the unavailability of space in the token FIFO. The two single-element FIFOs are joined by an "arbitrating call" cell which merges the two token flows into one. This cell forwards the requests from either of its inputs to the output, one at a time.

Each FIFO element is a pair of Muller pipeline elements (distributors) [SF01], very similar to a latch controller, but without the load enable output. The other two cell-types in the FIFO are used for inserting and removing tokens and are described below.

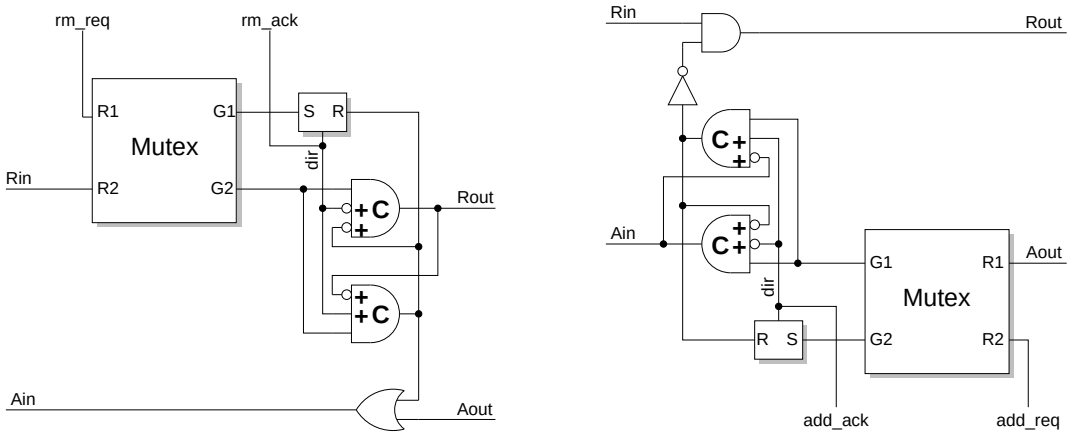
Token insertion/removal

Figures 6.6(a) and 6.6(b) show the circuits that remove and insert tokens in the FIFO, respectively.

In order to remove a token, all that is needed is to acknowledge an incoming request without passing it on to the next stage. Because the request to remove a token can coincide with the arrival of the input request, a *mutex* is included in the circuit. When the remove request wins the mutex it sets a 'direction' flip-flop (*dir*). The flip flop selects which direction the next input request will follow: either the output request or the input acknowledge. In normal operation the output request direction is followed. When a token is to be removed, the flip-flop is set and the next request will be routed to the input acknowledgement wire. When this happens the direction flip-flop is reset, so that the following requests will pass through normally.

The 'direction' is used as the acknowledgement for the remove request signal (*rm_req*). The latter should return to zero as soon as the direction is set, to release the mutex so that the next token request can pass through. The acknowledgement (*dir*) is held high, even after the remove request has dropped, until the token removal is complete.

Adding tokens (figure 6.6(b)) is very similar. Following an input request, which is forwarded to the output (*Rout*), the acknowledgement from the output side (*Aout*) is used to make the output request return to zero. As *Aout* returns to zero to complete the handshake, the input request *Rin*, which was not acknowledged, causes a second output request. This time the acknowledgement is



(a) Token removal circuit.

(b) Token insertion circuit.

Figure 6.6: Token insertion, removal circuits.

forwarded to the input side, so the input request is finally acknowledged, having produced an additional token in the meantime.

The token inserting circuit requires the existence of a token in order to produce another one. For this reason the token FIFO is initialised so that only one element has a token at start-up. If more tokens are needed they can be inserted using the token insertion circuit.

6.2 Collapsible latch controllers

Section 5.5 gave a brief description of collapsible latch controllers and how to integrate them into a processor. This section describes in detail their specification, circuit design and operation. Finally a detailed evaluation of the controllers is given.

6.2.1 Latch controller types

Latch controllers are an important element of the control part of an asynchronous processor. So it is not surprising that there is a great variety of latch controller types that can be built. This section considers the set of four-phase (return to zero) controllers implementing protocols used in AMULET3.

Depending on how long data validity is maintained after the handshake signals return to zero, there are a number of handshake protocols and corresponding

controllers. Two of the most commonly used protocols are discussed here (see figure 6.7):

Broad Data are kept valid until the (output) acknowledgement becomes low.

Broadish Data are kept valid until the (output) request becomes low.

Broadish is generally faster but assumes that the return to zero of the acknowledgement is 'dead' time for the downstream circuits. Some circuits need the data to be held for this time, in which case a broad controller/protocol must be used. As different parts of a processor have different use for the acknowledge return to zero time, generally both types of controllers are used.

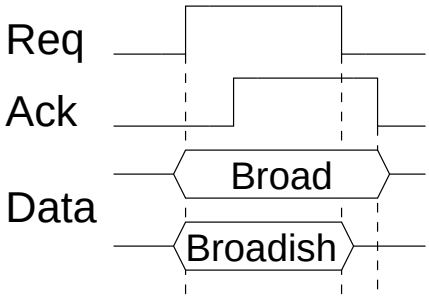


Figure 6.7: Broad, broadish protocol timing.

For each of the above protocols there can be two variations depending on when the latches become transparent:

Normally closed After the input request has been received and (obviously) before the input acknowledgement is asserted.

Normally open When the output side is not busy (which depends on the protocol) regardless of an input request.

Normally-closed latches isolate the downstream circuits from any spurious transitions while new data is expected but they suffer from the extra delay of having to open the latches after the data is ready. Thus normally-closed latch controllers lead to more energy efficient pipelines, while normally-open controllers lead to faster pipelines. To enable this trade off at run-time, reconfigurable latch controllers that can be either normally-open or closed have already been developed [LGB99] and are incorporated in the design of AMULET3.

Another type of latch controller, early-open, was later proposed [RLB00], but it requires the existence of a pre-request signal, becoming valid some time before

the input request, to open the latches just in time. As this signal might not always exist, this type of controller is not considered here. Nevertheless, when such a signal is available it is straightforward to apply the early-open concept to the latch controllers presented below.

6.2.2 Collapsed latch controller types

For a collapsed controller the input and output of both request and acknowledge signals are expected to behave as if they were the same wire. Specifically, output request should follow input request and input acknowledge should follow output acknowledge[2]. This section determines if the protocols described above are still meaningful when the latch controller is collapsed. Since the broad and broadish protocols differ only in the way the return to zero of the acknowledge signal relates to the data validity, the following discussion focusses only on this phase of the protocol.

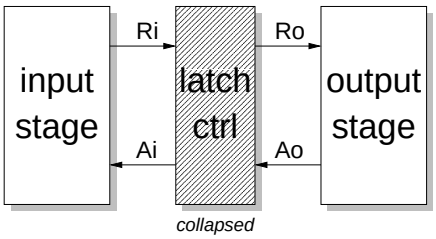


Figure 6.8: Pipeline with collapsed broad latch controller.

For broad protocol, the output stage expects the data to be kept valid until it has returned the input acknowledgement (Ao) to zero (see figure 6.8). As the latch controller is collapsed, its latch(es) will always be transparent, so it cannot directly guarantee the above condition. Thus this requirement must be guaranteed by the upstream stage. If the upstream stage adheres to the broad protocol, it will not change that data until *its* output acknowledgement (Ai) has returned to zero. Consequently, for correct operation, the collapsed latch controller must not allow its input acknowledgement (Ai) to return to zero until the output acknowledgement (Ao) has; in other words the controller behaves as if Ai and Ao were the same wire.

For the broadish controller type, the return to zero of the acknowledge signals

[2]Note that 'input' and 'output' refer to the flow of data in the pipeline, so the direction of the acknowledge signals is actually the inverse of what their names imply.

can happen at any time after the input request becomes low, as it is not related to the time when the data change. So the collapsible broadish controller can take a short-cut and allow the return to zero of the input acknowledgement immediately after the input request has fallen regardless of the state of the output acknowledgement. In this case the behaviour is different: Ai and Ao do not have to appear as if they were on the same wire.

Normally open/closed controllers

By definition a collapsed latch should be always open, to let data pass through as if it were not there. So it may seem that the normally-closed variation is of no use when the latch controller is collapsed.

In reality the unwanted energy consumption caused by glitch propagation — which triggered the idea for normally closed latches — is magnified when collapsed latches are used. Moreover it is well known that glitch-induced energy consumption increases with the logic depth between latches [BLBS+98], which is precisely what happens when collapsing a pipeline latch. So the benefit of reducing speculation by limiting the pipeline depth may backfire because of the increase in the energy consumed by glitches. Having a collapsed latch controller which only opens when the request input is ready would help, because the request is asserted only when the data are ready; all intermediate values are stopped from propagating down the pipeline. Obviously in this case the latch itself is not really collapsed, but the controller still gives this impression, so the term "collapsed" is still used here.

From the above discussion, all four types of latch controllers presented above are still meaningful when they are collapsed, thus circuits for all of them must be built. The reconfigurable normally open/closed operating mode [LGB99], is an attractive feature, so it is retained in the proposed latch controllers. The next section defines two (collapsible) latch controllers (for the broad and broadish protocols) that are configurable as collapsed or normal and as normally open or closed.

6.2.3 Circuit specification

A convenient way of describing the operation of asynchronous circuits is by using state transition graphs (STG) [SF01]. Figure 6.9 shows the STGs for normal

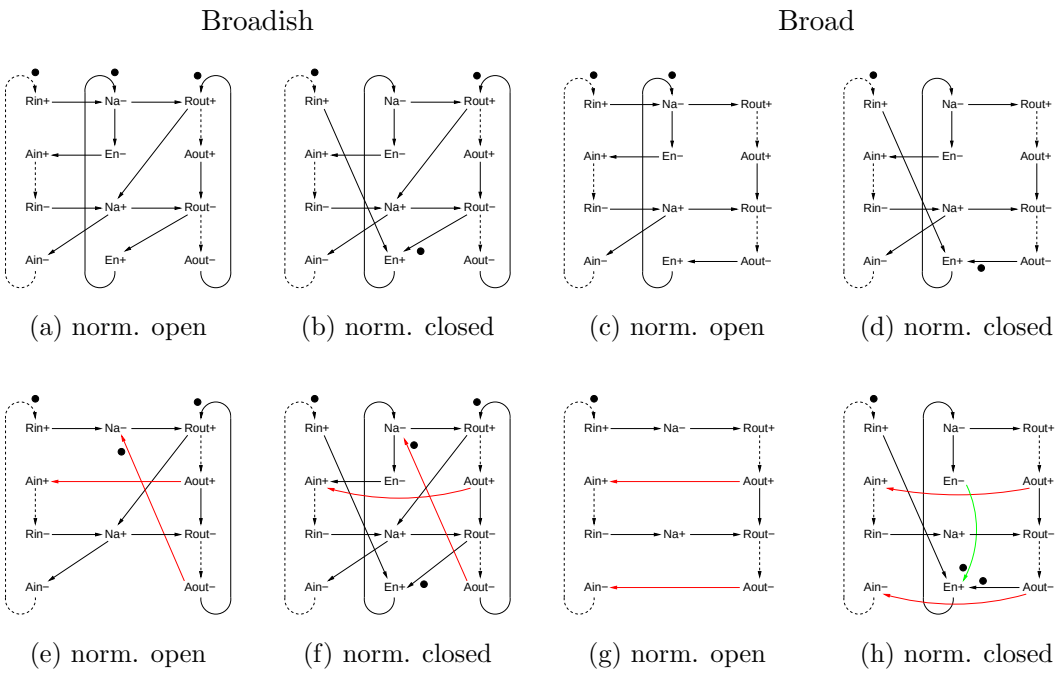


Figure 6.9: STG's of all latch controllers.

(a-d), collapsed (e-h), normally open and normally closed configurations for both broad and broadish protocols.

A brief description of the operation of the broadish latch controller is given below in both normal and collapsed operating conditions. The broad protocol is the same with the exception that the latch enable signal is allowed to rise after both output handshake signals have returned to zero.

Non-collapsed

In the quiescent state of normal, non-collapsed operating mode (fig.6.9(a,b)), all signals are low, except for *Na* which is high and *En* which depends on the normally open or closed condition. *Na* is low when a request has been received but not acknowledged yet. In normally-closed mode, an input request (*Rin*) causes *En* to rise, making the latches transparent. The rising *En* causes *Na* to fall, which turns *En* back low and makes *Rout* high, propagating the request downstream. After the latch has closed, *Ain* is asserted to acknowledge the input. *Rin* could then fall which resets *Na* to its quiescent value of 1, causing *Ain* to return to zero. At the output side, the raised *Rout* will eventually be acknowledged by a rising *Aout*. Then *Rout* falls re-enabling *En* to rise, when the next input request

arrives. The only difference in normally-open mode is that *En* is set back to 1 whenever *Rout* falls, regardless of the state of *Rin*.

Collapsed

In collapsed, normally-closed mode (fig.6.9(f)) the operation is the same as above except for the rising transition of *Ain*. This happens whenever *Aout* rises. At that time *Rin* is still held high, so *Na* is low. *Rin* can now fall which makes *Na* high, turning *Ain* low again. The high *Na* combined with *Aout* will make *Rout* low. *Na* cannot fall again in response to a new input request until *Aout* has returned back to zero. In normally-open mode (fig.6.9(e)), *En* is forced high continuously, so *Na* does not depend on it. All other operations are exactly the same.

The STGs of the collapsed controllers have only two added arcs compared to the normal controllers. By the definition of the collapsed controller, arc *Aout*+ → *Ain*+ is added, so that the input can only be acknowledged when the output is acknowledged. The corresponding arc for the falling edge is only needed when the protocol is broad. For the broadish version, an arc from *Aout*- to *Na*- is added instead. This makes sure that if a new input request is received (*Rin*+) while the output acknowledgement has not yet fallen, *Ain* will not be set high, mistaking *Aout* for a new acknowledgement. When the latch is collapsed and normally-open, the load enable is always on, so it is not shown in the STG. It should be noted that a number of arcs in the collapsed STGs (6.9(e-h)) are not necessary but are retained to show the similarities to the equivalent normal STGs (e.g. *Aout*+ → *Rout*-).

6.2.4 Collapsible latch controller circuits

The proposed collapsible broadish controller is shown in figure 6.10. For comparison, figure 6.11 shows the original broadish latch controller.

The proposed circuits are based on the existing latch controllers by Lewis *et al.* [LGB99]. The STGs in figure 6.9 were synthesised independently using Petrify [CKK+97] and the resulting equations were modified slightly to be useful in all operating modes. The final circuits are speed independent in either operating mode. During the transitions between the collapsed and normal modes though, speed independence cannot be guaranteed. For this reason a few gates are added

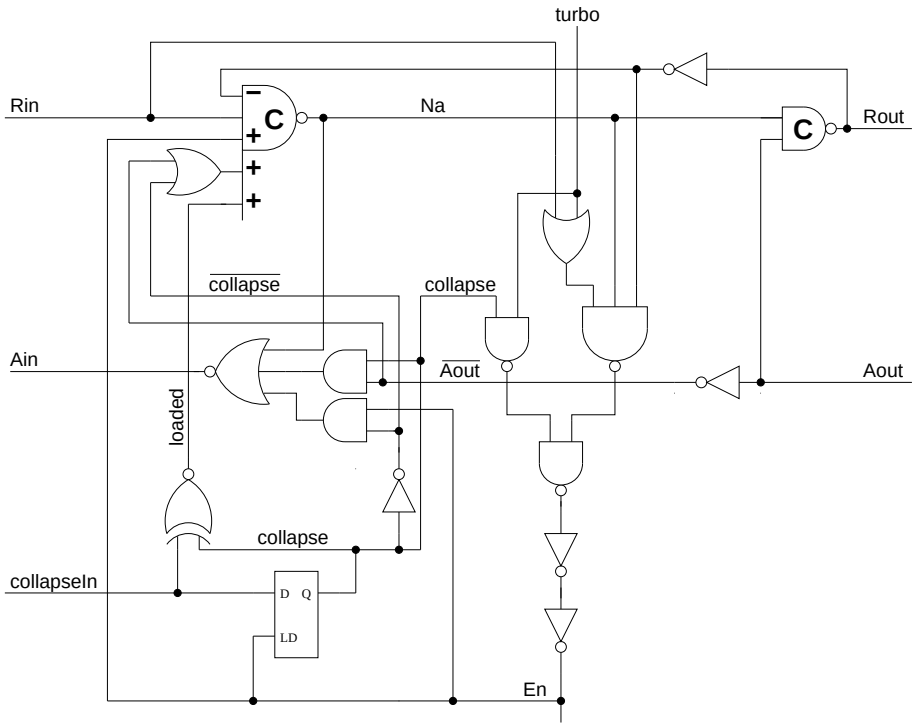


Figure 6.10: Collapsible broadish latch controller.

to ensure smooth transition between the operating modes with some timing assumptions. These proved easy to guarantee and transistor-level simulation in a 0.18um process showed that the latch controllers are operating (and switching operating modes) correctly in all process corners.

There are four main differences to the existing latch controllers:

- *Ain* is now driven by an AND-OR-invert gate instead of a NOR gate. For *Ain* to rise, *Na* must be low, meaning that a request has been received. In addition, *En* must be low, in normal mode, or *Aout* must be high, in collapsed mode, selected by the multiplexor formed by the AND gates connected to the NOR which drives *Ain*.

- The C element driving *Na* has two extra inputs controlling when it is going to fall. When the controller is collapsed, *Aout* must be low for *Na* to fall. This enforces the arc from *Aout*- to *Na*- in the STG. The OR gate makes sure that this is ignored in normal operation. The second new input (*loaded*) ensures that the new value of *collapse* is loaded before an acknowledgement can be given. It is only used when the controller switches operating condition.

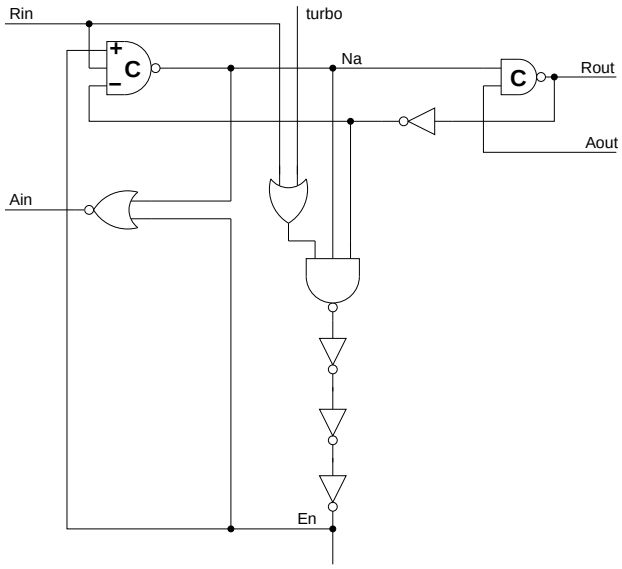


Figure 6.11: Existing broadish latch controller.

- *En* is forced high when *turbo* and *collapse* are both high by the added NAND gate.
- There is a latch to hold the *collapse* signal.

Changing to collapsed operating mode

When changing from normal to collapsed mode, the *collapseIn* signal will be high some time before *Rin* rises (bundling constraint). In the simplest case the latch is normally-open and the previous output has been acknowledged so *En* is already high, and the latches transparent, including the one that holds *collapse*. Thus *collapse* becomes high and makes the following changes:

- *En* is forced high (*turbo* is on), but it is already high so it doesn't trigger any further changes.
- The multiplexor integrated into the C element driving *Ain* selects $\neg Aout$. This can only affect the rising edge of *Ain*, which must wait for *Na* to fall before. Thus no transitions are caused in this case.
- It will let $\neg Aout$ pass through the OR gate integrated into the C element driving *Na*. Again no other transitions are triggered.

In this, the best case, everything will be set before *Rin* rises, so when it happens, the behaviour of the circuit will be as defined by the STG of the collapsed controller (6.9(e)).

The worst case is when *Rin* is already high when *En* rises, either because the controller is normally-closed, or because the output just received its acknowledgement. In this case *collapse* will rise some time after *Rin*. Because of the race between *collapse* and *Na* for both *En* and *Ain*, *Na* must not be allowed to transition until after *collapse* is loaded and it has set up the various gates it controls. This is done by the XNOR gate of *collapseIn* and *collapse*: if *collapseIn* is different to *collapse*, the XNOR output, *loaded*, is low keeping *Na* from falling at the next input request until *collapse* has been loaded. By that time, everything that *collapse* controls must be steady.

This is an additional timing assumption/requirement, which breaks the speed independence of the controller at mode transition times, but the requirements are reasonable and can be easily verified by simulation. Two minor circuit changes that can help in this case are to load the *collapse* latch from an earlier version of *En* (e.g. two inverter gates earlier) and/or to add some extra delay at the XNOR output. The later will only slow down the controller when changing operating mode, since *loaded* is constantly driven high when there is no operating condition change.

Changing to normal operating mode

When the latch is collapsed and normally-open, the latches are always transparent, so a change in *collapseIn* will immediately pass through to *collapse*. According to the broadish protocol, changes to the data — which extends to *collapseIn* — are allowed after *Rin* has fallen, without waiting for *Ain* to fall too. So, in the worst case, *collapseIn* and *Rin* will fall at about the same time, causing *Na* to rise and *collapse* to fall, respectively. The cleared *collapse* will try to make *En* drop, as it is held high only because of the *turbo-collapse* AND gate. Concurrently, as *Na* and *Aout* are both high, they will cause *Rout* to fall. The combination of high *Na* and low *Rout* will raise *En* high (normally open, so *Rin* does not have to be high). Thus *En* may exhibit a downwards glitch but it will not have any other effect on the circuit. *Ain* can only return to zero and this is triggered by rising *Na* without any other condition. Similarly, the falling transition of the glitch on *En* cannot affect *Na*, because it is connected to a '+' input of the C element, and

the rising edge of the glitch can only clear *Na* when *Rin* is also high, which is the normal operation. Thus the circuit is safe in this case.

To save the unnecessary transition on the *En* wire, which is heavily loaded with capacitance, the collapse latch can be made to load only when both *En* and *Rin* are high.

When the latch is normally closed, the rising *Rin* will open the latches and *collapse* will be cleared. If, as mentioned above, *loaded* is set late enough to give time to all other gates controlled by *collapse* to be steady, the operating mode change will work as normal: *Na* will drop when *En* is high regardless of *Aout*, and *Ain* will be acknowledged as soon as *Na* drops *En*.

Collapsible broad latch controller

The implementation of a collapsible latch controller is simpler than the broadish one, mostly because, from the specifications, *Aout* cannot be high when *En* becomes high (see broad protocol STGs in figure 6.9(g-h)). This is apparent in the circuit implementation in figure 6.12. The original broad latch controller is shown in figure 6.13, for comparison.

Because the broad protocol is simpler, the only race-condition when changing

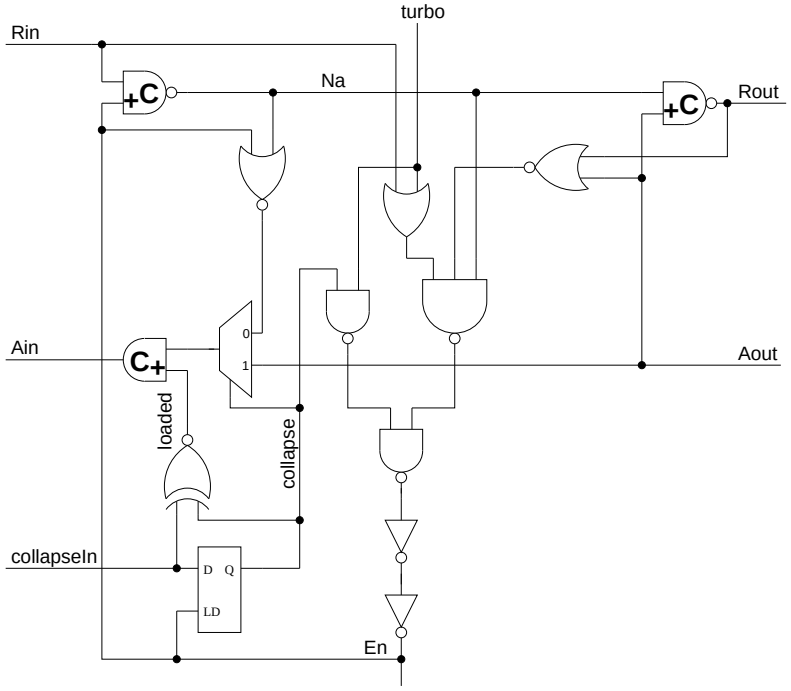


Figure 6.12: Collapsible broad latch controller.

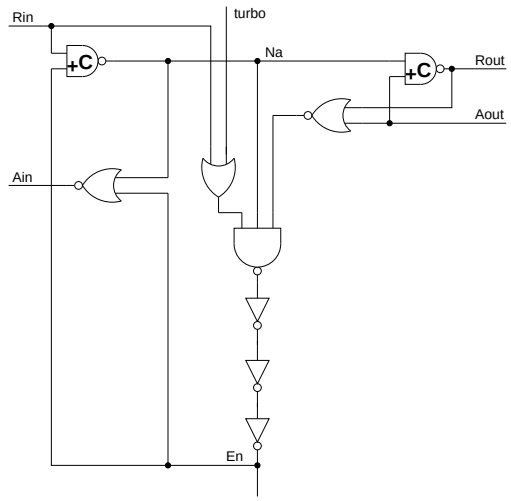


Figure 6.13: Existing broad latch controller.

operating mode concerns the input acknowledge (*Ain*). When changing from normal to collapsed operating mode, the circuit must ensure that *Ain* is not asserted before *collapse* has been allowed to set all the signals it controls. This is done using an XNOR gate of *collapseIn* and *collapse*, in the same manner as in the broadish controller circuit. The reverse operating-mode change is also smooth, without any glitches on the load enable line (*En*). Falling *collapse* will not cause *En* to drop because at the time *collapse* falls, *En* is held high by either *Rin* (normally-closed) or the NOR gate of *Rout* and *Aout* (normally-open).

6.2.5 Evaluation

For evaluation both the collapsible and the standard latch controllers were designed in STM 0.18μm technology and simulated with Spectre, Cadence's variation of Spice. The designs are at transistor level, but they have not been laid out thus the effect of the interconnect capacitance is not included. In AMULET3 the latch controllers were not custom laid out, because a great variety of them is needed; the same norm was followed here. The only wire that is likely to have a considerable capacitance load is the load enable signal *En*, which is modelled in the simulations. All the other wires are expected to be short, so their parasitic capacitance should not make a noticeable difference.

The test harness used is shown in figure 6.14. It consists of four latch controllers, without any data processing blocks between them. At the rightmost side the output request automatically generates an acknowledgement, if *ack* is one.

Table 6.1: Simulation results

Protocol	Normally	Parameter	Collapsible	Existing
Broadish	closed	Min. cycle time	1.31ns	1.13ns
		Rin+ to Rout+	0.49ns	0.44ns
		Rin+ to Ain+	0.72ns	0.65ns
	open	Min. cycle time	1.02ns	0.89ns
		Rin+ to Rout+	0.21ns	0.19ns
		Rin+ to Ain+	0.43ns	0.39ns
Broad	closed	Min. cycle time	1.82ns	1.32ns
		Rin+ to Rout+	0.48ns	0.44ns
		Rin+ to Ain+	0.96ns	0.68ns
	open	Min. cycle time	1.53ns	1.06ns
		Rin+ to Rout+	0.19ns	0.18ns
		Rin+ to Ain+	0.66ns	0.41ns

Similarly at the leftmost side an acknowledgement generates another request if *goB* is low. All the load enable wires have a 0.2pF capacitance load on them.

In all the simulations the same test harness was used and the components were replaced by the latch controller under test. Four circuits were simulated: the new collapsible latch controller (in the 'normal' operating mode) and the standard latch controller, each in both broad and broadish implementations. Each circuit was simulated in both normally-open and normally-closed operating modes.

Table 6.1 summarises the results. The cycle time of the collapsible latch controller is always slower than the standard one from 14% (for broadish normally-open) to approx. 40% (for broad normally-open). This is expected as the collapsible latch controller is significantly more complex.

When data processing functions are inserted between the pipeline latches, the cycle time will be dominated by the data processing, thus the impact of the latch controller will be minimal.

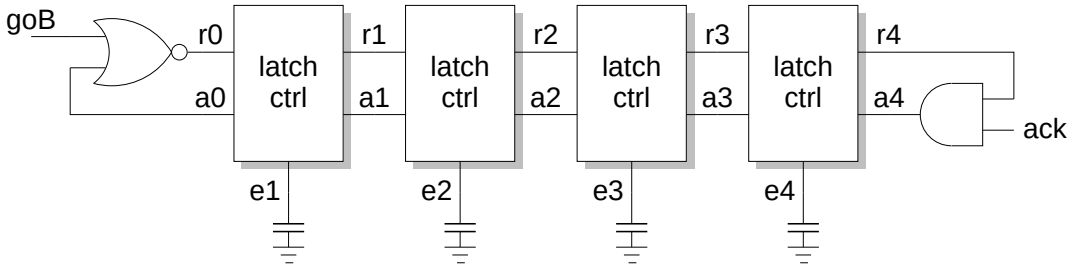


Figure 6.14: Evaluation set-up for the latch controllers.

Chapter 7

Dynamic pipeline adaptation

Chapter 5 presented two methods for controlling the speculatively fetched instructions in a processor by adjusting the pipeline occupancy. The evaluation presented there was based on a constant pipeline configuration for the duration of the program execution. This is the primary reason for the heavy performance penalty paid by all the benchmarks when the pipeline occupancy was low. This chapter investigates algorithms that change the pipeline occupancy adaptively, based on opportunities or predictions, using hardware mechanisms.

For the lowest possible power consumption a processor with a pipeline occupancy of one should be used. Conversely, if the highest possible speed is required, the processor should be configured to be fully pipelined. For an in-between trade-off, the simplest solution is to have a fixed occupancy for each program, determined by some analysis done by the compiler about how frequently branches are (expected to be) taken. However, even in different parts of a program, there is considerable variation in the number of speculative instructions fetched. Therefore dynamic techniques could be more efficient in minimising speculative instruction fetches while not increasing the execution time by as much as a static occupancy setting.

As explained earlier, the only speculatively fetched instructions in a single-issue processor are those following a taken branch. Hence a mechanism that could predict branches with a good accuracy and with a very low energy overhead is needed. Standard branch prediction was shown in chapter 5 not to be energy efficient, thus other prediction schemes are investigated in this chapter.

A conservative solution would be to stop fetching any other instructions after a branch has been fetched. There are two disadvantages for this solution. First

it slows down the processor for every branch instruction whether taken or not. Second it is usually not possible to identify that an instruction is a branch until it gets decoded which is at least after one pipeline stage. During that time the instruction following the branch is being fetched speculatively. Furthermore, in asynchronous processors, the decode unit where a branch gets identified must explicitly synchronise with all the other stages in order to inform them of its existence. Such global synchronisations are costly in time and are avoided in asynchronous design.

Compared to a standard branch predictor the branch target is not needed as the output of the prediction, since speed is not a primary objective. Therefore a simpler and more energy efficient way of predicting branches than the standard branch predictor could be designed. Two such methods are presented in this chapter. The first tries to predict the position of the branch in the instruction stream from the positions of past branches. The second 'predicts' conditional branches by identifying instructions that change the processor's condition code and assuming that a branch will follow.

7.1 Branch anticipation

For dynamic pipeline adaptation to be successful, some mechanism for predicting branches must be used so that the pipeline occupancy is adjusted at the right time. This section aims to establish if the position of the next branch in the instruction flow can be predicted based on the positions of a number of previous branches. If it is true, the branch can be *anticipated* and the pipeline occupancy adapted when the instruction at the anticipated position arrives.

The analysis here assumes a synchronous processor in that all stages take the same time (1 cycle) and the instructions move from one pipeline stage to the next at the same time. This is obviously an approximation for an asynchronous processor, but greatly simplifies the analysis.

7.1.1 Algorithm sketch

After a branch has executed, a new stream of consecutive instructions is fetched and executed from the new PC address. The distance of an instruction from the beginning of that stream, i.e. its sequence number, is called *instruction position* (α).

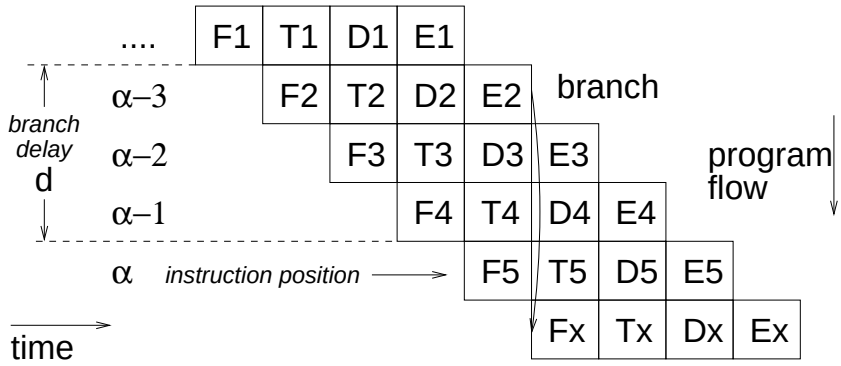


Figure 7.1: Abstract pipeline view.

With pipelining a number of instructions are present in the processor at any time. Figure 7.1 shows an abstract view of the execution of an instruction stream in a four-stage pipelined processor. In this example the second instruction is a branch which ends the current instruction stream and starts a new one. Due to pipelining, when the branch happens, a number of instructions consecutive to the branch — at the *branch shadow* — are already in the pipeline and have to be discarded. Generally, the number of instructions in the branch shadow is the same as the number of cycles it takes for the new instruction stream to arrive at the processor. *Branch delay* (d) is the common term for this number and is used here to refer to the number of instructions in the branch shadow rather than the delay.

Suppose α is the position of the next instruction to be fetched, as in figure 7.1. If any of the instructions currently in the pipeline, i.e. at positions $\alpha - 1$ to $\alpha - d$, is a branch (and is eventually taken), fetching the instruction at α would be wasting energy since it would be discarded.

Figure 7.2 shows how the branch probability distribution graph is expected to look like: very few branches at the first instruction positions, a peak somewhere in the middle, and few branches at positions which are too far. The probability of having a branch in the pipeline, when the decision for fetching instruction α has to be made, is represented by the shaded strip of the graph between $\alpha - d$ and $\alpha - 1$.

The branch anticipation algorithm works as follows: when the prefetch unit is to fetch an instruction, it first predicts if the fetched instruction is in the shadow of a branch or not. If it guesses the instruction is in a branch shadow, it defers fetching it until the pipeline is drained, otherwise it fetches normally. The decision

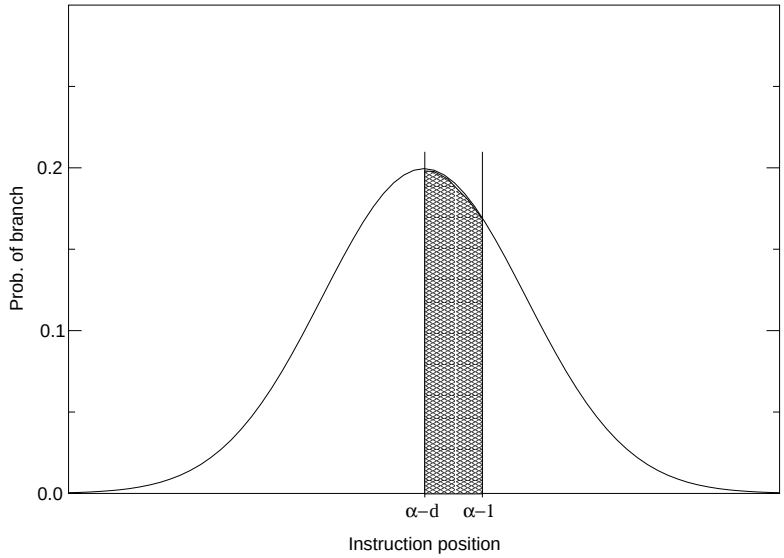


Figure 7.2: Probability of a branch in the pipeline.

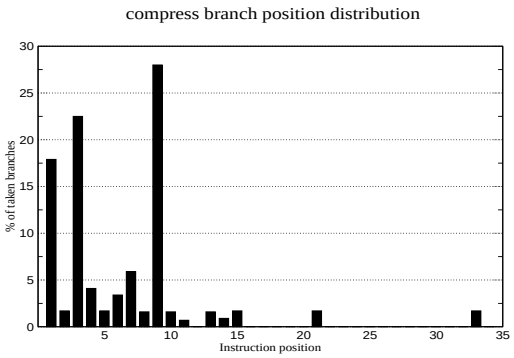
is based on the position of the instruction to be fetched (α) and a history of past behaviour of branches (the probability distribution). If no branch occurs as the pipeline drains, the processor could either start filling up the pipeline, as in its normal operation mode, or it could single-step each subsequent instruction until a branch finally occurs. The first option was chosen here as it offers more flexibility: the processor can be stopped at a later position again, while the second option is too conservative.

This behaviour can be easily implemented in a processor with collapsible latch controllers, by starting to collapse the pipeline latches when the decision to stop fetching is taken. Collapsing will start from the *prefetch* stage and spread to the other stages as the last instruction (at $\alpha - 1$) moves down the pipeline. By the time this instruction reaches the *execute* stage, if there was a branch in the pipeline, it would have send a 'branch request' to the prefetch unit. Otherwise the acknowledgement of the instruction at $\alpha - 1$ will propagate from the *execute* stage back to the *prefetch* which will start fetching the next instruction. If single-stepping is chosen, the latch controllers will be left collapsed, otherwise the first instruction fetched will revert them to their normal mode as it travels down the pipeline.

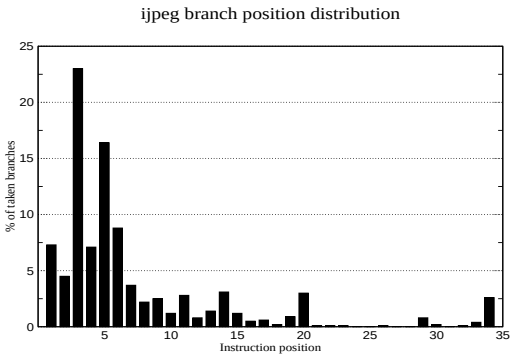
Position of branches in the instruction stream

The distribution of branch positions is presented as 'bell-shaped' above. The graphs in figure 7.3 show the actual data from three SPECInt95 benchmarks. The measurements are gathered by scanning executed instruction traces from instruction-level simulations of StrongARM running the benchmarks. The graphs are not 'bell-shaped' as the number of branches do not increase steadily, peak at a single position, and then drop. In most cases the graphs continue further to the right as there is a small number of very long instruction streams.

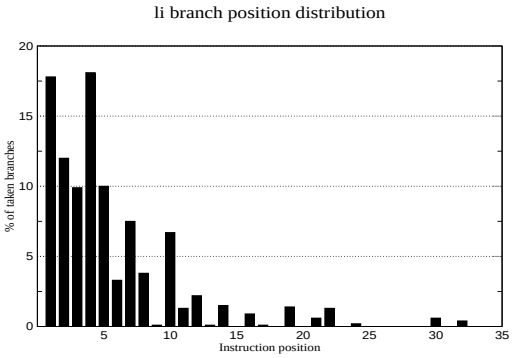
Interestingly the probability of having a branch immediately after another branch (at instruction position 1) is quite high. These are returns from subroutines which are called as the last instruction of another subroutine, or jump-tables implementing "switch" statements. On average approximately 6 instructions can be expected to be executed before another branch is taken, with blocks of more than 10 consecutive instructions being very rare.



(a) compress



(b) ijpeg



(c) li

Figure 7.3: Percentage of branches occurring at each instruction position.

Note that the graphs show the distribution of the branches for the duration of the program execution. In practice the curve shape can change as different parts of the program are executed. The prediction algorithm should be able to adapt according to this change.

Deciding when to stop fetching

The outline of the algorithm above left open the question of when the probability of a branch present in the pipeline is high enough to stop fetching. Depending on the success of the algorithm in predicting branches, execution time is traded for energy savings. If fetching stops before the branch shadow, time is wasted since instructions that will eventually execute are fetched with delay. If it stops too late in the branch shadow, energy is wasted since unwanted instructions have already been fetched.

The heuristic used here is to stop fetching when the probability of having a (to be taken) branch in the pipeline is higher than the probability of a branch expected further down the instruction stream. Using the graph of figure 7.2, prefetching stops when the shaded area is larger than the area to the right of $\alpha - 1$. In every cycle the two areas are calculated based on the current instruction position and the stored information about the branch position distribution and a decision is taken before fetching the next instruction.

Branch position distribution

To gather the information needed for the 'branch position distribution' (*BPD*) as the program is being executed, the number of taken branches at each instruction position must be counted. Naturally it is not practical to keep information about all previous branches. Moreover, since the distribution of branch positions could change as the program is running, some of the old information might be misleading the algorithm in its predictions. Thus only a number of recent branches, the *branch history* (h), is stored in the BPD structure.

In theory, counting (taken) branches at each instruction position would require an infinite number of counters. In practice, a sufficiently large number of counters — called *buckets* here — must be used. These buckets are numbered from 1 to n according to the instruction position that they keep information for. Since the average branch position observed in the simulations is 6–8, the number of buckets (n) does not have to be much larger than that. Since the heuristic needs to know

the number of 'future' branches, the last bucket (n) counts not only the number of branches that were taken at that instruction position, but also for every further position. As not all past branches are remembered, the buckets are 'leaky', losing the information about the oldest branch when a new one happens.

When the current instruction position becomes n or greater, the heuristic cannot be used as no information is kept for these positions. Two options were examined in the simulations for this situation: either continue at normal speed (`C`), i.e. fully-pipelined, or stop prefetching (`S`), i.e. single-step the remaining instructions until the branch. The first option, `C`, gives up anticipating the branch and thus will execute the remaining of the instruction stream at full speed, but incurs the full energy penalty of fetching d instructions when the branch eventually happens. The second option, `S`, assumes that the branch should be arriving at any moment so, very conservatively, sacrifices speed to save energy.

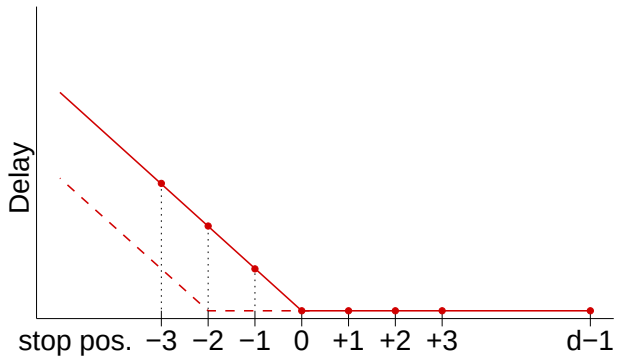
7.1.2 Evaluation setting

Prediction accuracy effect on energy, delay

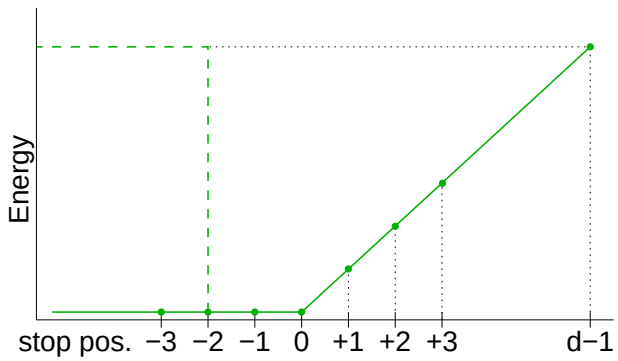
This section analyses the effect of the prediction accuracy on energy and delay. Figure 7.4 shows the delay and energy penalties in relation to the prediction accuracy, i.e. how far from the beginning of the branch shadow the algorithm decides to stop prefetching. When prefetching stops exactly at the instruction position following the branch, the prediction is successful as no energy or time is wasted. This position corresponds to the '0' on the horizontal axis. The values to its right represent the number of instruction positions beyond the beginning of the branch shadow where prefetching stopped, while those to the left represent instruction positions before.

For this evaluation, energy is wasted when instructions following a (taken) branch are fetched. For simplicity it is assumed that each such instruction consumes the same amount of energy. There are two cases where energy is wasted:

- When prefetching stops at a position to the right of '0' in figure 7.4(b). The energy penalty increases linearly as the stopping position is further from the branch, since more instructions are wrongly fetched. There is an upper limit to this penalty, since d instructions after the branch ($d-1$ after '0') the pipeline starts filling up with useful instructions again, so no more energy is wasted.



(a) Delay



(b) Energy

Figure 7.4: Branch, delay penalty for branch prediction.

- When prefetching stops before the branch, there is no wasted energy as long as the processor continues to single-step the instructions up to the branch. As described in the previous section, variant `C` of the algorithm starts operating in fully-pipelined mode again at instruction positions beyond the last 'bucket' (n). Thus if the actual branch position is beyond n, stopping early will waste the maximum energy in variant `C`. This is shown with a dashed line in figure 7.4(b).

In the processor considered, every taken branch incurs a delay penalty of d cycles as all branches are assumed 'not-taken'. Thus even if the prediction is successful, this delay will still be incurred. If prefetching stops at an instruction position beyond the branch, there will be no further delay penalty since the branch is already in the pipeline and it is progressing at the normal, one cycle per stage speed.

On the contrary if prefetching stops before (or even at) the branch position,

there will be a delay penalty. Each instruction fetched after the stop will take d cycles to complete since it is single-stepped. Thus the delay penalty increases linearly with the distance from where prefetch stops to the branch position (solid line in figure 7.4(a)).

The delay penalty of variant `C` depends on whether the branch position is before n or after. In the first case the delay penalty is as above (solid line). In the latter case the penalty depends on the actual number of instructions that are single-stepped. The graph (dashed line) is parallel to the solid line, since the same delay is incurred for each single-stepped instruction in both cases, and meets the minimum delay at the position where n would fall in the horizontal axis.

The above discussion assumes that when prefetching is stopped, the processor operates in single-step mode until either a branch occurs or, for variant `C`, position n is reached. In reality the processor could resume its full speed earlier. Thus the delay line should be considered representing the maximum delay.

Simulation

Deviating from the usual evaluation flow (described in section 5.1), branch anticipation was evaluated using a C program and instruction traces from an instruction-level simulator. This method provides a way of evaluation without having to spend too much time in implementation, as would have been required by the 'standard' methodology. Moreover, the algorithm requires significant hardware resources which should be accounted for their energy consumption, but this is hard to estimate before they have been implemented.

In order to speed-up run time, the C program does not simulate the full processor operation, but only part of the behaviour of the prefetch unit, where the branch anticipation algorithm would be incorporated. It uses a previously generated instruction trace, containing the addresses and opcodes of the executed instructions, which is produced by the fast, instruction-level simulator (armsd) contained in ARMtools v2.51.

As each instruction address is read from the trace, it is passed to the simulated prefetch unit, unless the address is not sequential. In this case a branch must have occurred so, to emulate the pipeline behaviour, a number (d) of dummy sequential addresses are generated internally, before the non-sequential one is passed to the prefetch unit.

At each cycle the simulated prefetch unit decides whether or not to fetch based on the branch anticipation algorithm. At the end of the simulation, the execution time in cycles and the total number of fetched instructions are reported based on the energy-delay analysis presented above. In addition, statistics about the accuracy of the branch predictions are reported.

Benchmarks

The benchmarks used for the evaluation are *compress*, *li*, and *ijpeg* from SPECInt. The other benchmarks used in simulations, *Dhrystone*, *DES encode*, and *GSM filter* execute too small a number of instructions to be used here. All benchmarks were simulated for 10 million instructions, after skipping the first 1–5 million instructions, depending on the benchmark. These initial parts of each program set up the data structures and consist of simple loops which are easily identified and exploited by branch anticipation. It was decided that excluding them from the simulations would make the results more indicative. The branch distributions for the simulated trace parts are selected to be the same as those for the simulation of the whole benchmark.

Simulation variations

As explained earlier, when the current instruction position exceeds the number of available buckets n, the simulator can be instructed either to stop fetching (`S`), i.e. single step until a branch is taken, or continue fetching (`C`).

In addition to these options, two further options were evaluated for the way the heuristic decides to stop fetching. The one already explained compares the probability of a branch in the pipeline (shaded area in fig. 7.2) to that of a branch further down the instruction flow. The other compares the probability of a branch existing in any instruction position up to the current one (equivalent to the area from 0 to $\alpha - 1$ in fig. 7.2) to that of a branch further down the pipeline. With the second option once prefetching stops, the processor gets into single-step mode until a branch occurs (or, combined with option `C`, until instruction position n is reached). Since the probability of stopping is higher for the second option, `H`(High) is used to show that this option is enabled, while `L` is used for the standard option.

Four variations of the branch anticipation algorithm can be simulated using the above options. The variants are represented in the following results with

two letters referring to these options. Table 7.1 is provided as a quick reference explaining these variations and the variables used in the evaluation.

Table 7.1: Symbols used in evaluation of branch anticipation.

d	Branch delay
n	Number of 'buckets'
h	Branch history size
H	Use area from 0 to $\alpha - 1$
L	Use area from $\alpha - d$ to $\alpha - 1$
S	Stop fetching beyond instruction position n
C	Continue fetching beyond n

7.1.3 Evaluation results

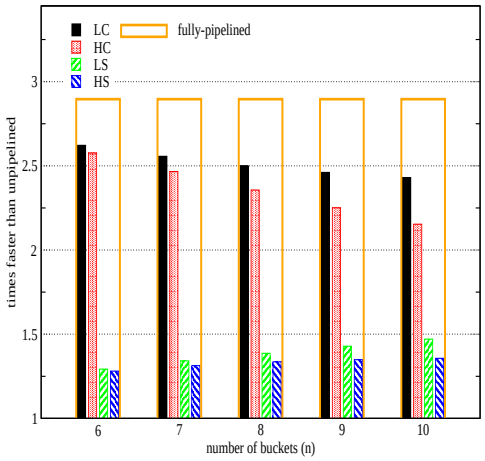
Figure 7.5 compares the delay and energy of the branch anticipation method to an unpipelined processor, i.e. a processor which only fetches an instruction after the previous one has been executed. For reference the delay/energy of the fully-pipelined processor is also included in the graphs. The data presented are based on averages (arithmetic mean) of the three benchmarks. The graphs (7.5(a), 7.5(c), 7.5(e)) show how much faster each variant is relative to the unpipelined processor (higher is better). Energy in the graphs (7.5(b), 7.5(d), 7.5(f)) shows how many extra instructions were fetched relative to the unpipelined processor (lower is better). The default values of the three variables are: $n = 8$, $d = 3$, $h = 16$.

As expected, the energy and speed of all variants for all the possible variable values lie somewhere between those of the fully-pipelined and the unpipelined processors. Unfortunately branch anticipation does not achieve a lower energy-delay product than the fully-pipelined processor, as was intended; the variants that are close in speed to the fully-pipelined processor do not achieve a number of extra prefetched instructions as low as an unpipelined processor. After the analysis of these results below, the next section reports on the prediction accuracy of branch anticipation.

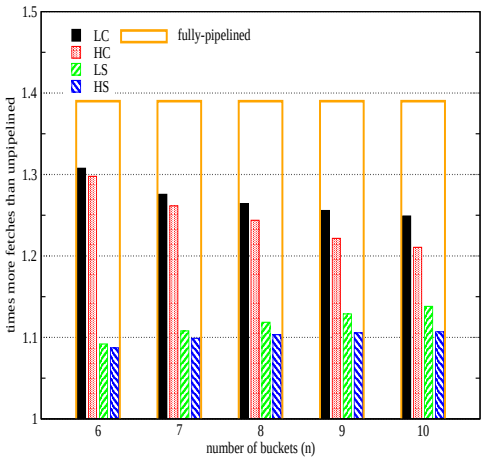
Options `H`, `L`

In many cases variants which differ in their `H` or `L` option have similar speed and energy consumption especially when the number of buckets is comparable to

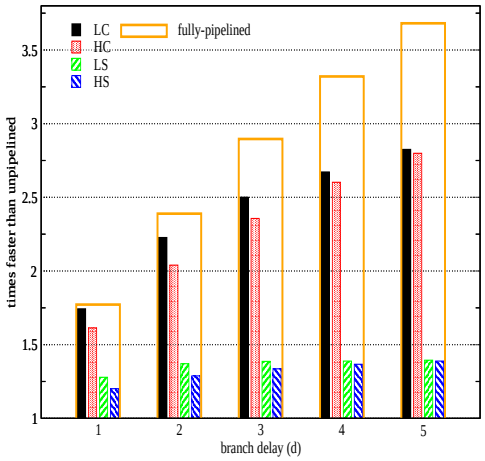
speed energy



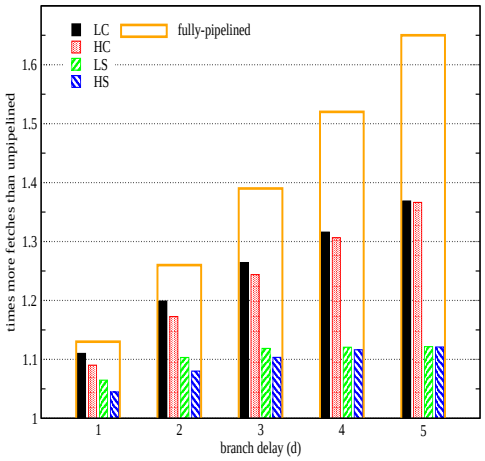
(a) $d = 3$, $n = 6 - 10$, $h = 16$



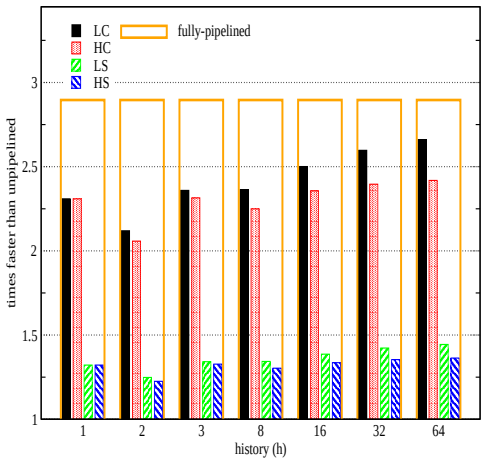
(b) $d = 3$, $n = 6 - 10$, $h = 16$



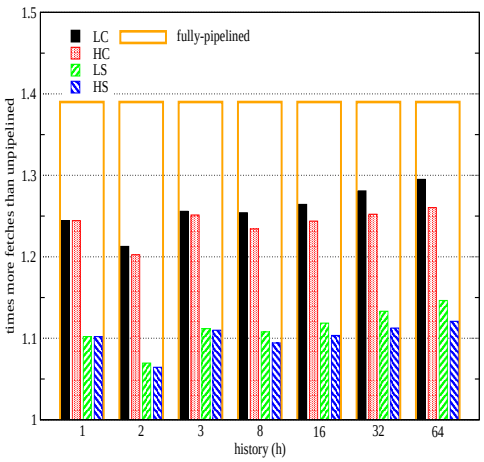
(c) $d = 1 - 5$, $n = 8$, $h = 16$



(d) $d = 1 - 5$, $n = 8$, $h = 16$



(e) $d = 3$, $n = 8$, $h = 1 - 64$



(f) $d = 3$, $n = 8$, $h = 1 - 64$

Figure 7.5: Dependence of branch anticipation energy, delay on the parameters.

the branch delay or when the number of branches in the history is low. This is because in these cases the probability of a branch already present *in* the pipeline (`L`) is similar to the probability of a branch existing *before* the current instruction position (`H`). In other words the difference between the areas of the probability distribution graph they represent is small. In general the energy delay product favours the `L` option.

Options `C, S`

Forcing the processor to single step after fetching the instruction at position n (`*S`) saves more fetches compared to the opposite variants `*C`, but it also lacks considerably in speed. The difference between these variants in both energy and delay gets smaller when a large number of *buckets* (n) is used, while the other variables seem to have the opposite effect. Generally the energy delay product favours the `C` option.

Number of buckets (n)

Figures 7.5(a) and 7.5(b) show the difference that the number of *buckets* (n) make in delay and energy. As n increases, the `*C` variants are becoming slower while their energy consumption also drops. On the contrary, `*S` variants are becoming faster and their energy consumption increases.

To explain this behaviour consider, for example, n increasing from 6 to 7 and the instruction to be fetched is at position 6. When $n = 6$ the processor would either always fetch (`*C`) or always stop (`*S`) at this instruction position, while when $n = 7$ the decision will be made based on the branch position distribution. Thus for variant `*C` there will be more pauses and consequently fewer fetches, while for `*S` there will be fewer pauses.

Branch delay (d)

As the *branch delay* (d) increases, the branch anticipation method is increasingly fast, compared to the unpipelined processor, for the `*C` variants. The speed difference with the fully-pipelined processor is increasing though. On the contrary, the `*S` variants become only slightly faster with higher d. This means that their delay increases at almost the same rate as the unpipelined processor, which is proportional to d. There is a (fixed) proportion of instructions at positions beyond

n, which are single stepped so they take $d+1$ cycles each to execute. As d increases so does the time spent in these instructions which explains this observation.

In the energy graphs, for the `*S` variants the number of extra fetches increases very slowly, while for `*C` the increase is much sharper. For instruction positions lower than n, both variants behave exactly the same, thus the difference in the number of fetched instructions must come from branches happening at positions beyond n. At these instruction positions, `*C` variants always fetch d extra instructions for each branch, while `*S` do not. As d increases so does the difference in fetched instructions between the two variations which is what is shown in the graph. The small increase in `*S` is due to the increased penalty each time a branch is not predicted.

Branch history (h)

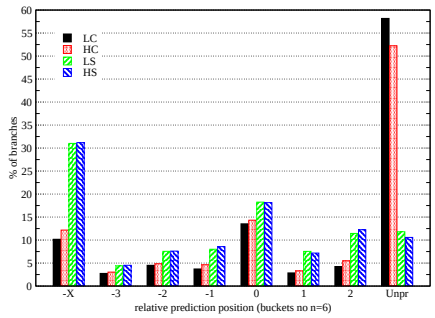
Figures 7.5(e) and 7.5(f) show that the size of the *branch history* (h) makes a relatively small difference to the performance and the energy consumption. This means that just 'remembering' the position of the last branch is enough to get most of the effects of *branch anticipation*. Consequently the hardware overhead for keeping the branch position distribution should be small.

The fact that whether keeping the last one branch or more does not make a significant difference to the algorithm, hints that the prediction accuracy is limited. This is examined in the following section.

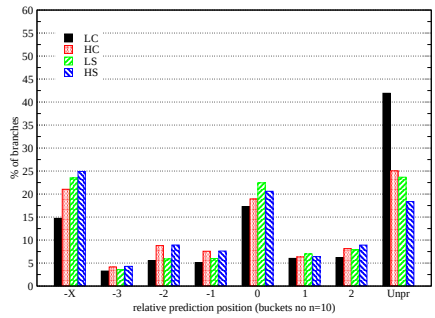
7.1.4 Evaluation of prediction accuracy

Due to the nature of *branch anticipation*, the effectiveness of the prediction depends on the distance of the predicted position to the actual branch. For this reason the evaluation presented here is based on both the proportion of predicted branches and the relative distance of the prediction to the actual branch.

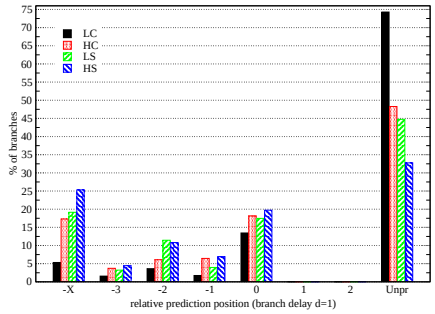
Each graph in figure 7.6 shows the percentage of predicted branches at each (relative) position, for all four variants of the algorithm. As in figure 7.4 earlier, '0' on the horizontal axis corresponds to when prefetching stops exactly at the instruction position following the branch. The values to its right represent the number of instruction positions beyond it where prefetching stopped, while those to the left represent those before. '-X' represents all prefetch stops at four or more instructions before the correct position. The percentage of unpredicted branches



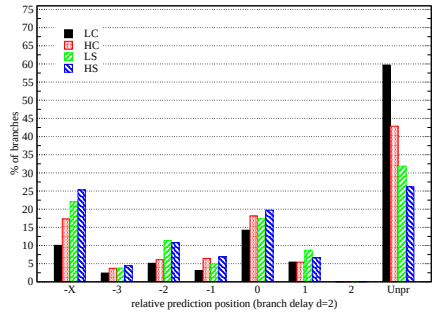
(a) $d = 3$, $n = 6$, $h = 16$



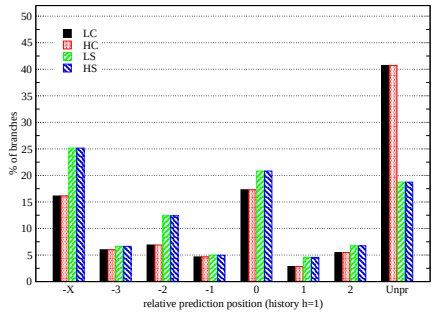
(b) $d = 3$, $n = 10$, $h = 16$



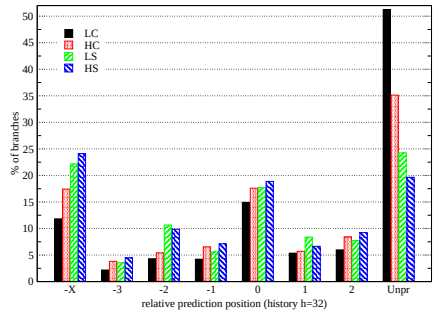
(c) $d = 1$, $n = 8$, $h = 16$



(d) $d = 2$, $n = 8$, $h = 16$



(e) $d = 3$, $n = 8$, $h = 1$



(f) $d = 3$, $n = 8$, $h = 32$

Figure 7.6: Prediction accuracy (avg. for all benchmarks).

is also included in each graph. This category accounts not only for the case where prefetching never stopped for a branch, but also those where prefetching stopped early but then restarted so the branch was missed.

Each row in figure 7.6 shows two graphs for representative values of one of the three variables: number of buckets n, branch delay d, and history h.

The first observation is that only up to 20% of the branches are predicted at the optimal instruction position ('0') and the unpredicted ones always dominate in the graphs. As expected, `*S` variants cause more stops so their 'unpredicted' bars are always the lowest, while at the same time they frequently stop too early.

As the number of buckets increases, the percentage of unpredicted branches drops as some of its share moves into the other categories, especially '0' and '-X'. The same happens when the branch delay gets higher. In this case, a part of the previously unpredicted branches moves to the new bars appearing to the right of '0'.

Finally, increasing the number of branches kept in the 'history' causes a small decrease in the percentage of branches predicted at the optimal position, while fewer stops happen early. This shows that although there is a good chance for a branch to be at the same position as the previous one (around 16%–20% in 7.6(e)), the branch position in general cannot be predicted from the positions of a number of previous branches.

7.1.5 Conclusion

The evaluation showed that the prediction of branches based on their position in the instruction stream is not successful with the algorithms presented. This is the reason for *branch anticipation* not being more energy efficient, in energy-delay product, with comparison to a fully-pipelined processor. Regardless, it could still be useful as an energy-delay configuration between the two extremes of an unpipelined and a fully-pipelined processor, which may not be expensive to build, as only the position of the last branch needs to be kept.

7.2 Condition-code setting detection

Instead of trying to predict the position in the instruction stream where a branch could happen from the behaviour and the positions of past branches, a 'hint' from the instruction stream could be used to predict that a branch is approaching. Since over 80% of the branches are conditional[1] [HP96], the instruction that generates the branch condition could be used as that hint.

The processor used here implements the ARM architecture which uses condition codes to specify the branch condition. Thus a technique that can be applied is to start single-stepping the processor when an instruction that changes the condition codes is detected, as there is a high probability that it is closely followed by a conditional branch. When the first instruction from the branch target

[1] This figure is for an architecture without conditional instruction execution. The percentage is probably lower for ARM programs.

is fetched, the processor will resume its normal operating mode. For processors that do not use condition codes, comparison-type instructions could be detected instead.

Detecting instructions that change the condition code is simple in the ARM instruction set architecture as there is a specific bit in the data-processing instruction format which controls this. It is important that the detection occurs as early as possible in the pipeline; the number of stages between *prefetch* and the detecting stage determine how many instructions following the one that sets the condition code have already been fetched and thus how much energy might be wasted. In the processor used here, the detection can be done at the first decoding stage (Thumb in figure 5.2), so only one instruction may be already fetched before the detection happens. This is very useful as over 50% of branches in the benchmarks used here, immediately follow the instruction (usually compare - CMP) that sets the condition codes.

In order to detect the condition-setting instructions as early as possible, a more general opcode pattern could be used for the detection which matches more instructions than just those setting the conditions. This will obviously put the processor into single-stepping mode for larger parts of the program execution, slowing it down but, if it allows the detection to be done at an earlier stage, it could save an extra instruction fetch per taken branch.

As an alternative to detecting the condition-setting instruction as early in the pipeline as possible, an optimising compiler could be used to insert some 'neutral' instructions between the branch and the condition-setting instruction. This would make the hardware implementation simpler and could allow enough time to detect the condition-setting instructions in instruction-sets with complex encoding. Modifying the existing compiler was not possible for this work, since the source code is not available. Thus the following design of this technique tries to detect condition-setting instructions as early as possible.

7.2.1 Design

Of the two methods to control the pipeline occupancy presented, the one which dynamically changes the pipeline using collapsible latch controllers is the most appropriate for implementing this technique. When an instruction that changes the condition code is detected at the Thumb stage, the whole processor should become just one pipeline stage, by collapsing all the pipeline latches. As explained

in chapter 5 this can only happen gradually; as the condition code setting instruction moves down the pipeline it collapses the pipeline latches on the way. That would still leave the processor a two stage pipeline, *prefetch* and the rest of the stages joined, so one instruction following the branch would be fetched and wasted if the branch is taken.

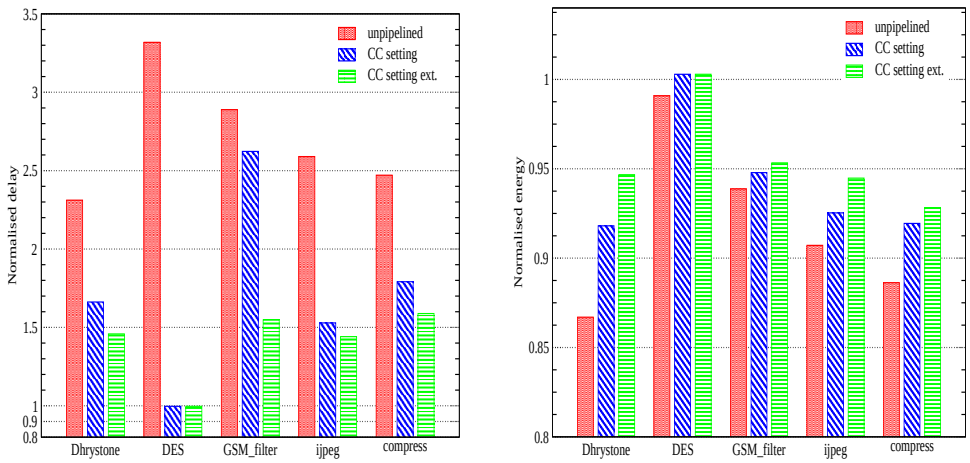
In order to save this instruction too, the information that a condition-setting instruction has been detected must be sent back (e.g. counter-flowed) to the prefetch stage, so that the remaining latch controller can be collapsed as well. The simplest and safest way would be to pass this information the next time the two stages 'synchronise', when the next instruction is passed from *prefetch* to *Thumb*. The implementation used here takes advantage of the knowledge that the memory access will take longer than the detection, and passes the result of the detection to *prefetch* before the memory acknowledges. This will collapse the *prefetch* to *Thumb* latches (when they next get loaded), making the whole processor a single pipeline stage. Thus *prefetch* will not fetch another instruction until the one following the detected is complete. Consequently, even if a branch follows immediately after the condition-setting instruction, no extra instructions will be fetched.

When the first instruction from the branch target is fetched, it resets the latches to their normal operating mode, as it travels down the pipeline, returning the processor to its normal, fully-pipelined operating condition.

If the branch is not taken or, generally, the next taken branch is quite distant, the processor will single-step for a considerable amount of time that could have a bad effect on its performance. In the ARM instruction set architecture all instructions can be conditional (not only branches) which can help to avoid some branches. Thus the design described above might be too conservative. For this reason a variant was designed and evaluated that detects instructions that are conditional but not branches and forces the processor to operate in fully-pipelined mode. Thus when the condition code is set for use by a non-branch instruction, the processor does not have to operate in single-step mode until a branch occurs. This method is expected to improve the execution delay without compromising the energy savings achieved.

7.2.2 Evaluation

The condition-code setting detection techniques require sufficiently small changes to the processor that can be evaluated using the standard evaluation methodology used in this work (section 5.1). Two Verilog models were produced and simulated running the usual set of five benchmarks. Figure 7.7(a) shows the execution delay of each benchmark for the two variations described above and the unpipelined version, all normalised relative to the fully-pipelined version. Figure 7.7(b) shows the normalised energy consumption for the same programs and processor variations.



(a) Delay (b) Energy

Figure 7.7: Evaluation results for condition code setting.

For *DES encode* there are insignificant energy savings with any speculation control technique, as there are very few branches and thus almost no speculative instruction fetches. Thus, for this benchmark, the interest is in the execution delay overhead of the method tested. In this case both condition-setting detection variants managed to keep the execution delay to the levels of the fully-pipelined version. In contrast the unpipelined version is over three times slower.

The difference in the two variations can be seen in a benchmark like *GSM_filter*. The execution delay of the first variant is very close to that of the unpipelined processor, while the execution delay of the second variant is almost half of the first. As the *GSM_filter* code has many conditional data-processing instructions, it clearly benefits from restoring the fully-pipelined mode early, as is done by the second variant.

Generally, both variants managed to reduce the execution delay compared to the unpipelined processor, for a small increase in energy consumption. The second variant is consistently faster than the first, but it also consumes more energy. It is very useful for benchmarks that behave like *GSM_filter*.

7.3 Summary

This chapter details two methods for adapting the processor pipeline dynamically, based on the collapsible latch controller technique described in chapter 5.

The first, *branch anticipation*, predicts when a branch is going to happen based on the position of branches in the instruction stream in a recent history. If a branch is predicted, the following instructions are not fetched until the branch has been resolved. The method was evaluated using high-level simulation: executed instruction traces and a C model of the prefetch mechanism. Unfortunately the evaluation results showed that this method of predicting branches is not successful as at most 20% of the branches were predicted accurately.

The second method, uses compare-type instructions as a 'warning' that a conditional branch may be approaching. Since most branches are conditional, significant energy savings can be achieved by not fetching instructions following a branch, until it has been resolved. This method was implemented and shows comparable energy savings to a unpipelined processor, but with a much lower speed penalty.

These methods are not the only possible or the best proven solutions to energy reduction by dynamic pipeline adaptation. A plethora of other techniques and variation can be designed. Software controlled methods, for example, could be used to set the number of tokens in the token-FIFO or set which pipeline stages should be joined. However, the methods described in this chapter do show that collapsible latch controllers can be successfully employed for *dynamic* power management in an asynchronous processor.

Chapter 8

Low-power cache design

As shown in chapter 4, the memory consumes a significant proportion of the total energy in the system. In a memory system where a cache is used, most of the energy consumption comes from it. For example the cache is responsible for over 40% of the power consumption in StrongARM [MWA+96]. At the same time the cache is usually performance-critical so implementations targeting low energy consumption cannot neglect the access time.

The previous chapters concentrated on saving energy in the processor core. Here, as a step towards adaptive, low-energy memory systems, a cache architecture is presented which is based on a novel CAM design [EG02b].

8.1 Cache organization

There are many issues involved in designing a cache. This work does not deal with most of the 'high-level' cache design options — total size, replacement algorithm, write policy — and how they affect the hit rate, but does have some dependence on associativity and the line size. It is more related to the organization of the data and tag storage blocks and how these are accessed in order to save energy.

A fundamental assumption throughout this chapter is that the data storage of the cache is made up of a number of small memory blocks (*banks*), because this is more energy efficient than using a single large memory [SD95]. In RAM blocks energy consumption and access delay increase as the block size increases [AH00], thus dividing a large memory into small banks gives speed and energy advantages at the expense of an increase in silicon area due to the overhead of the (constant size of the) memory periphery circuits: sense amplifiers, decoders,

precharge circuits etc. Burd [Bur01] presented a case study, where he concluded that banks of about 1 KByte give the best compromise. Banks of that size are used in almost all ARM implementations [MWA+96], [Seg98], [Bur01], [CHM+01].

Although direct-mapped caches are the fastest and simplest to build they do not achieve hit rates as high as associative caches of the same size for most programs [HP96]. High hit rates are important not only for performance but for energy consumption, since they reduce the number of accesses to larger, secondary memories via long buses. Thus another assumption is that direct-mapped caches are inadequate for the hit rates required, so some degree of associativity is needed. Generally the higher the associativity the better the hit rate, but design complexity increases with associativity and the hit rate obeys the law of diminishing returns. As a rule of thumb, an associativity of four is usually adequate, although a higher associativity — at no extra cost — would still be preferred.

Associative caches can be built using either RAM- or CAM-based tag structures. The rest of this section discusses the possible organizations of each category and their advantages and disadvantages.

A cache size of 16 KBytes with a line size of 32 bytes is used as a working example. Based on the fact that sub-banking is energy efficient, the cache is made up of sixteen memory banks of 1 KByte. The choice of the bank and cache-line sizes means that each bank contains 32 cache lines and, similarly, each tag structure holds 32 tags.

8.1.1 Caches with RAM-based tags

The simplest way a RAM-tagged cache can be organized is as a conventional n-way set-associative cache, where a number, n, of memory banks are accessed concurrently (figure 8.1). Since an associativity of four is regarded as giving an adequate hit rate, for the working example there will be four sets of four memory banks. Within each memory bank a direct-mapped organization is used, so some of the address bits are used to select a cache line. A further two bits of the address would be used to select which set of 4 banks to access, in addition to those selecting the cache line and word within the line, leaving a tag size of 20 bits for a 32-bit architecture. The conventional set-associative cache is quite wasteful of energy because, for every memory read, n bank (tags and data) reads are performed, of which at least $n - 1$ are wasted[1].

[1] Misses waste all n accesses.

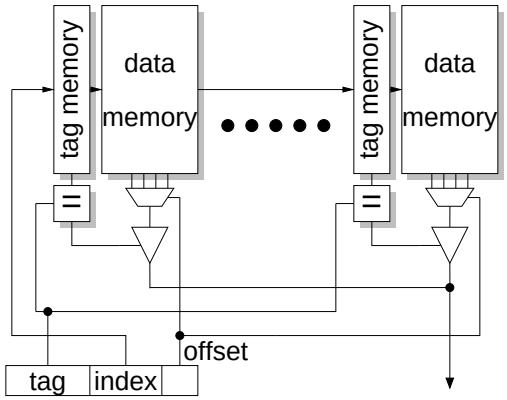


Figure 8.1: RAM-based cache organisation.

In order to improve the energy consumption of the conventional set-associative cache, "phased caches" were invented [HKY+95]. They operate in two phases. In the first phase they read and compare all the tags in parallel. In the second phase, the one data memory where a hit is found (if any), is accessed. Phased caches save considerable energy by avoiding unnecessary accesses to the data memories. However, in a synchronous system, they double the cache latency since they need two clock cycles (or phases) to read or write data. The first of these cycles is probably underutilized as the tag read and comparison takes less time than accessing the data memory. In an asynchronous system the data memory read can start immediately after the first phase, so phased caches would not suffer from doubling the latency.

Instead of 'phasing' the tag and data memory accesses, the order of accessing the cache 'ways' can be phased. This is the principle behind "pseudo-associative" caches [HP96].

Pseudo-associative caches

Pseudo-associative caches were originally invented as a cheaper way of implementing associativity in a direct-mapped cache [HP96], [CGE96]. The cache is initially accessed in the usual direct-mapped fashion. Following a miss at this access, another line in the cache is tried (usually by inverting the most significant bit of the address index field) and if this does not hit either, the access is treated as a miss. Obviously this process of 're-trying' could be repeated a number of times, to give a higher associativity. In the cache organization described here,

since the memory banks are relatively small, each attempt can happen in a different bank — to spread out the occupied cache lines, instead of trying a different address at the same bank.

Interestingly, pseudo-associative caches have variable hit (and, in some cases, miss) latencies depending on the number of bank accesses performed. Although this could be a disadvantage in a synchronous processor, since the delays must fit into fixed clock cycle intervals, an asynchronous processor can take full advantage of this property.

Considering pseudo-associative and phased caches, there are two orthogonal variations in the way an associative cache can be accessed:

- The order and degree of parallelism in accessing the memory banks of a set, e.g. all the banks in parallel for conventional set-associative caches, one bank at a time for the simplest pseudo-associative ones.

- Within the banks, the sequential or parallel accessing of the tags and the data storage for loads; stores must access them sequentially.

There is a clear speed - energy trade off in these design options. The more parallelism, the better the speed but the worse the energy consumption because the output of most parallel operations is not used. The more a sequential approach used, the less energy is consumed, but the average access time may be larger. When a sequential method is employed, if the first bank accessed hits most of the time, both the average access time and consumed energy would be low. Thus a successful method of predicting which bank to access first saves both time and energy, providing the energy consumption of the prediction itself is low. This is the rationale behind the original way-predicting cache[IIM99] and the variations proposed by Huang *et al.* [HRYT01].

A way-predicting set-associative cache [IIM99], speculatively selects one bank (way) to access first and if there is no hit, the remaining banks are accessed concurrently, as in a conventional set-associative cache. Using a most recently used algorithm, the prediction was successful on average 86% for the data cache and 96% for the instruction cache. Huang *et al.* [HRYT01] experimented with combinations of way-predicting and phased caches. Two of the most interesting ones are “fall back phased” and “predictive phased”. “Fall back phased” works as a way-predicting cache for the first cycle and if there is no hit, it accesses the remaining banks in a phased manner: first the tag blocks and then the data block

where the hit is found, if any. "Predictive phased" caches read all the tag blocks and the predicted data block in the first cycle and subsequently access a second data block if the prediction was wrong and there is a hit in another bank. They both were shown to perform slightly better than way-predicting, with predictive phased achieving the best energy-delay product.

8.1.2 Caches with CAM-based tags

Using a CAM for tag storage in a cache bank makes the bank a small, fully-associative cache. Thus every time a bank is accessed, any cache-line in the bank could be the one required. Given that the associativity within a bank is already high, the most suitable way to organize the banks is as an array, directly indexed by some bits from the memory address (figure 8.2).

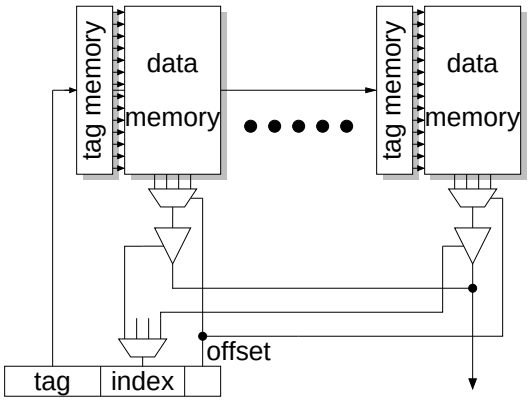


Figure 8.2: CAM-based cache organisation.

The CAM-based organization is inherently phased: the CAM must have a match before the data RAM is accessed. In addition, no other bank is accessed in parallel. This means that this organization is a good candidate for an energy efficient cache.

The associativity offered by this cache organization is high, probably more than required for a suitable hit rate. The high associativity is not intentional, but rather a side effect of the data bank and the cache line sizes. Using 1 KByte banks and 32 byte cache-lines, as in the working example, gives an associativity of 32, since 32 lines fit into one bank. For the 16 KByte cache the tag size would be 23 bits, for a 32-bit processor.

8.1.3 Comparison of cache organizations

To decide which cache organisation is the most efficient, a comparison of their energy consumptions is undertaken. The comparison is based on the number of accesses to tag and data memories for each cache access.

A variation of the "fall-back phased cache", where even the first access is phased, is selected as the most energy-efficient RAM-based cache. In this organisation only one data bank is accessed for cache hits and none for misses. Since CAM-based caches also have the above property, the comparison needs to examine only the number of accesses to the tag memories for the two organisations. In addition the comparison will have to consider the difference in the energy consumed when reading an SRAM and searching a CAM, which is presented in the following section.

The tag sizes of the two organizations can be different. For the working example used here, the CAM-tagged organization will have a tag size of 23 bits, while a direct mapped cache of the same size will have 20 bits of tag. The original pseudo-associative caches have the same tag size as a direct-mapped cache but require some extra bits to keep track of whether the cache line is at its original (direct-mapped) position or not. For simplicity it is assumed that the tag size of the pseudo-associative cache is the same as that of a direct-mapped cache; this may favour the RAM-based case slightly.

The way-predicting cache, using a most-recently-used algorithm for selecting which bank to access first, has a reported prediction rate of about 90% [IIM99]. If the other ways are checked in parallel, the average number of tag accesses will be $0.9 \times 1 + 0.1 \times 4 = 1.3$. The average number of tag accesses when checking the other ways in series, depends on the probabilities of hitting in these banks, but in general it will be somewhat lower. For the CAM-based cache there will always be exactly one tag search per memory access. So, if a CAM search consumes less than 1.3 times the energy of a similarly sized RAM, the CAM-based cache organization is more energy efficient.

Considering the overhead of the prediction for the way-predicting caches and the better hit-rates achieved by the high-associativity of the CAM-based organization, the above number is quite pessimistic. Burd [Bur01], comparing a CAM-based organization to a conventional set-associative organization, claims that a CAM-based organization is better if a tag CAM search consumes less than twice the energy of a tag RAM read. The actual number would depend on the details

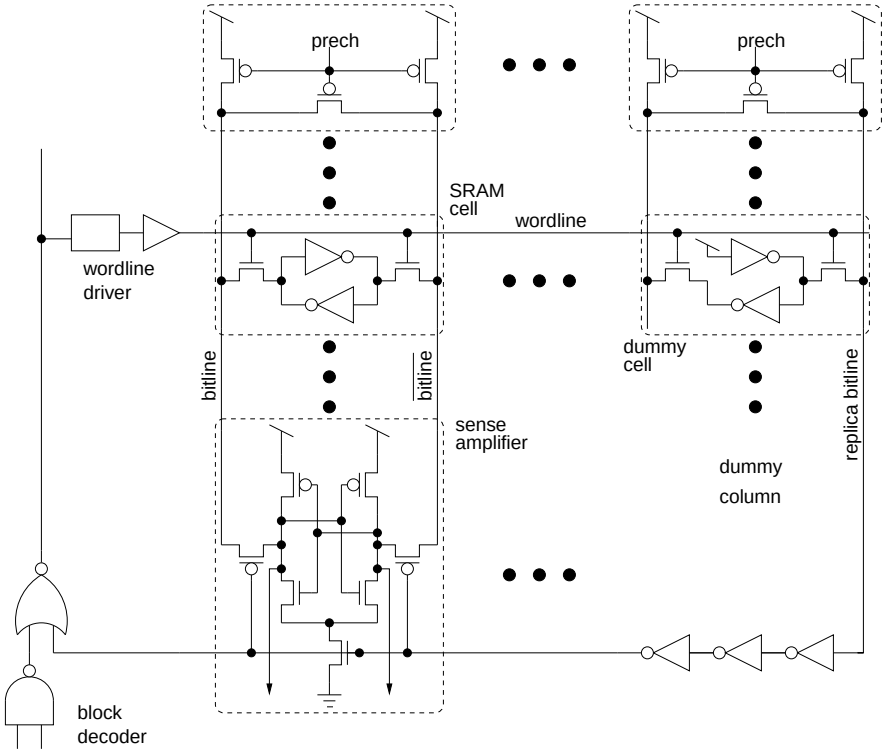


Figure 8.3: Low-power RAM block.

of the implementations of the two circuits, but it should lie somewhere in between these limits.

8.2 Comparison of RAM/CAM energy consumption

For the comparison a set of RAM, CAM and sense amplifier cells were designed and laid out in a $0.18\mu m$, 1.8V, dual V_t technology and simulated with Spectre, a Spice variant. Although the implementations could be further optimised, they were designed by the same person and a similar amount of time was spent on each of them, so this comparison should be fair.

The RAM design, shown in figure 8.3 is based on the low-power SRAM of Amrutur and Horowitz [AH98]. The standard 6-transistor static RAM cell is used for storage, arranged in an array of 32 rows of 20 cells. The block uses cross-coupled inverter-based sense amplifiers and pulsed wordlines to limit the voltage drop on the bit lines to about 15% of the supply voltage. A signal, generated

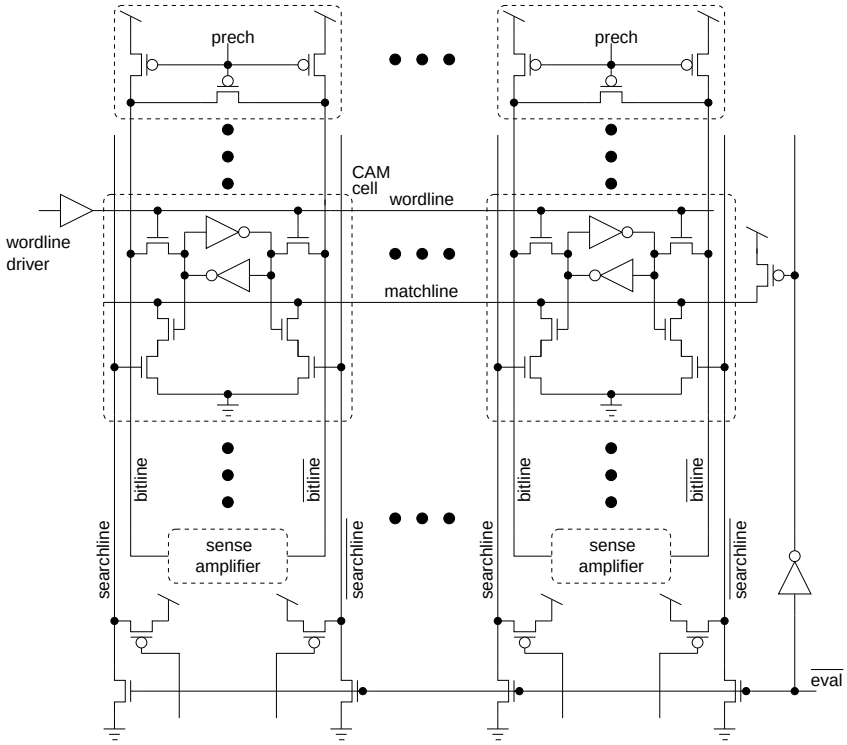


Figure 8.4: Low-power CAM block.

from a dummy cell column, is designed to match the delay of the bitline voltage difference to reach the appropriate level to be detected by the sense amplifiers. This signal activates the sense amplifiers and, immediately after, deasserts the wordlines so that the SRAM cells stop discharging the bitlines to conserve energy.

For the CAM design (fig. 8.4), separate search and bit lines were used to minimise the capacitance of the former, which are more frequently used. The CAM block is made up of 32 rows of 23 cells. Before starting a search, all match lines are precharged while the search lines are held low. During the search operation whichever cell does not match the value on the search lines, pulls down the match line in its row. No energy is consumed within the CAM cell itself when a search operation is performed; most of the energy consumption comes from the discharging of the match lines and from driving the search lines for each search.

To compare the two circuits, a series of ten reads and ten searches were performed on the RAM and CAM blocks respectively and the average delay and energy consumption were calculated. As neither the value nor the position of the row being read affects the energy consumption of the SRAM, the Spectre circuit

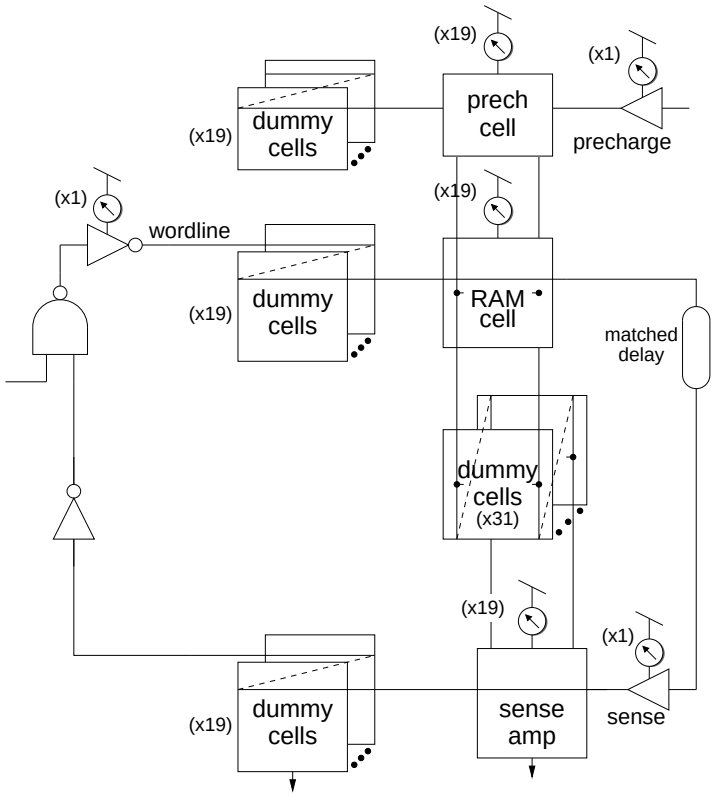


Figure 8.5: Spice model of SRAM block.

description was simplified, having only one cell active and dummy cells to capture the effect of wire loading for the columns and rows (figure 8.5). A number of probes were used to measure the supply currents which are then multiplied appropriately for the columns and the rows that would normally be activated. The CAM was modelled similarly, with two cells in a row active and dummy cells for the rest of the row and column. At most one row will match each time and that row consumes no energy, since it will not discharge its match line, but all the other rows do. All ten searches were made not to match in the simulation and the current consumed by the row is multiplied by the number of rows less one, to capture the most usual case of a cache hit.

The RAM-based tag block has an access time of 0.5ns and an average energy of 2.3pJ per read access, in typical silicon and operating conditions. These figures include the energy consumed by the drivers of the precharge and sense amplifier trigger signals, but not the address decoder consumption, which — for a 32 row memory — is not expected to contribute significantly. The wordline driver consumption is included however.

The energy consumed by the CAM is 12pJ per search at a minimum access time of 0.8ns. This energy consumption is over five times that of the RAM, although the result is slightly biased in favour of the RAM because the energy consumed by the decoder and that of the comparator needed to test the tags are not included. The CAM block is also larger than the RAM, the cell being 25% bigger and the block width is 23 bits for each of the 32 rows (3 bits more).

Based on these results a (pseudo) associative cache with a low average number of sub-block accesses per memory request is better than the CAM-based implementation. The following sections present an alternative CAM architecture that reverses this situation.

8.3 Application program behaviour in cache tag matching

Much of the energy consumed in a CAM is due to the frequent precharging and discharging of all but one of the match lines for each access. It would be useful to know the number of bits that are actually different in each comparison and their most likely positions. An analysis of the memory traces of various SPECInt95 benchmarks was undertaken to gather information to answer this question.

The applications were compiled for an ARM processor, with all the speed optimizations enabled, using the ARM Developer's toolkit compiler and debugger/simulator version 2.51. The simulator was set to emulate the StrongARM implementation which is the closest processor model to AMULET3 available in the simulator.

For each benchmark two cases were analysed:

- A unified 16K cache.
- Split instruction and data caches, each 16K in size.

The cache-line size is 32 bytes in all cases and the caches use a write-back, no write-allocate policy with round-robin line replacement.

A C program was written to model a cache with the above specifications. The match operation in the CAM was modelled so that, for every cache line, the tag is compared serially, starting from the least significant bit. Where there is a mismatch the operation ends. The program is recording the number of matches

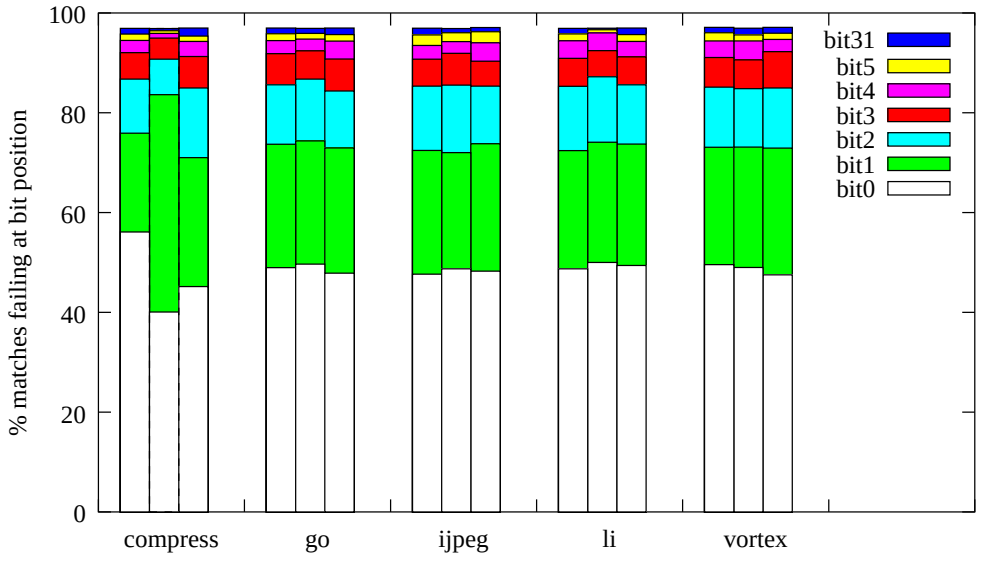


Figure 8.6: Ratio of tag checks ending at each bit position.

that stop at each bit position. The results, presented in figure 8.6, show that over 90% of the mismatches are determined within the four least significant bits of the tags. This behaviour is consistent for all the benchmarks and cache types: Unified, Instruction and Data, from left to right in the graph.

For these results virtual addressing was used for the caches. If the addresses were translated to physical addresses, the results may have been different, but for low-power operation, virtually addressed caches are preferred because they avoid address translation for most memory accesses.

8.4 Proposed CAM organization

Based on the results of the tag matching behaviour of the applications, an adaptive serial-parallel CAM (SPCAM) organisation was designed to take advantage of the high probability of mismatch in the four least significant bits of the tag. It is based on the following principles, which are the same as those of Hsiao *et al.* [HWJ01]:

Minimize the transitions on the match lines In this case, test the four LSBs first and only allow any of the other bits to discharge the match-line if the 4 LSBs match.

Use separate search and bit lines The capacitance on the search lines is lowered.

Do not force the search lines to 0 or 1 while charging the match lines The transitions on the search lines are greatly reduced. The search-line drivers are the second largest energy consumer after the match line precharge circuits.

Minimize the use of timing signals The energy spent on the signal drivers is reduced.

The proposed CAM can operate either in parallel or in serial mode. In serial mode the four least significant bits of each row are checked serially and, if they all match, the remaining bits are checked in parallel. In parallel mode the four LSBs are checked serially again, but testing the remaining bits is performed concurrently. The match results of the two parts are ANDed together in both cases to give the final result. The operating mode can be changed at any time when searches are not performed in the CAM, allowing an adaptive use of this feature.

A row of the circuit is shown in figure 8.7. Note that there are different types of CAM cells for the serial and parallel parts. The parallel CAM cell is the same as that in figure 8.4, with the ground connection of the two pull down NMOS chains testing the equivalence, replaced with *VgndMatch*. This signal runs lengthwise and is connected to all the parallel CAM cells in the row.

The serial CAM cell circuit is shown in figure 8.8. A cell that matches opens its NMOS pass transistor (N1), propagating the result from its less significant neighbour to its more significant neighbour. If there is no match, the match result from the previous stages is isolated and a one is generated at the output. Since

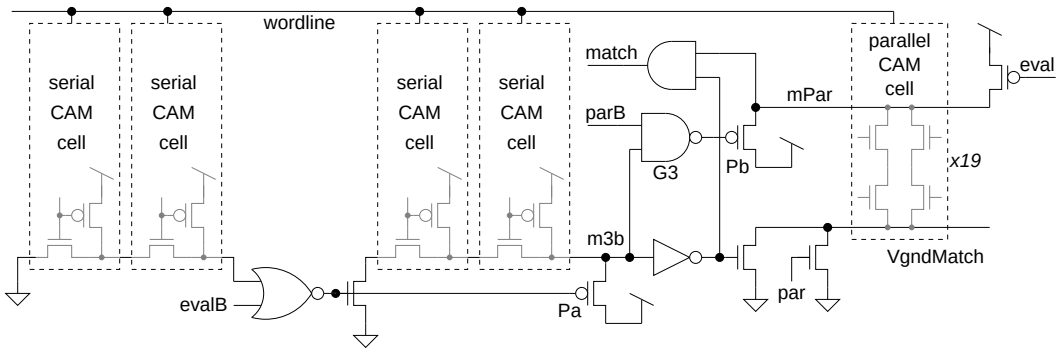


Figure 8.7: A row of the proposed CAM.

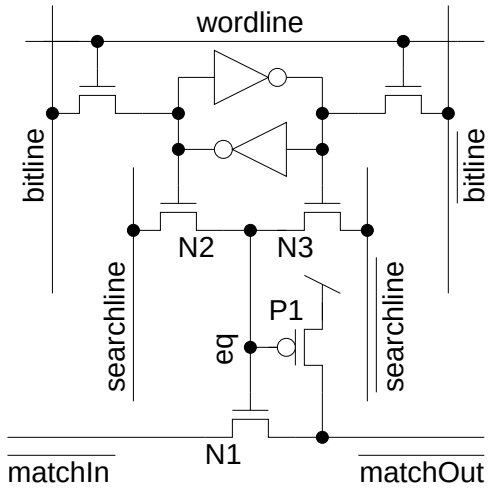


Figure 8.8: The serial CAM cell.

eq is driven high through an NMOS transistor (N2 or N3), its high state suffers from a (NMOS) threshold drop. In the current implementation, the technology offers two threshold levels for all transistors, so low threshold NMOS transistors were used for N2 and N3. In addition a high-threshold PMOS transistor (P1) is used for pulling up the output, so that it can be turned off by the weak high voltage of *eq*.

The match operation in the serial part is similar to the Manchester carry chain used in binary adders [KEA60]. A match propagates as a zero from the least significant bit to the most significant. The four serial bits are broken down into two sets of two bits to limit the maximum number of transistors in series to three. If the first cell of a set doesn't match but the second does then a one is propagated to the right through an NMOS transistor. This is allowed to propagate only through one device — another reason for separating the four bits into two sets. The gates at the output of these chains are designed so that their input threshold is lowered (with widened NMOS transistors) to compensate for the V_t voltage drop in this case. When both cells in the second set match but those in the first don't, transistor Pa (fig. 8.7) is used to pull up the serial part match signal *m3b* signifying a 'no match'.

If all of the four LSBs match, the virtual ground *VgndMatch* is pulled down allowing the rest of the row to evaluate the parallel match line *mPar*. Otherwise the match line of the parallel part is precharged, via transistor Pb. Most of the time *mPar* would not have been discharged previously, so Pb will just replace the small leakage charge since the last operation.

In serial operating mode the search lines are not pre-discharged; this saves energy as transitions only occur when the search data actually change. This means that the parallel CAM cells will be evaluating even when the match line is being precharged. If some of these cells don't match, there will be a path from the parallel match line *mPar* to the virtual ground, which is left undriven. This could leave *mPar* not fully charged but it will not affect the operation. If during the following search the LSBs match, *VgndMatch* will be pulled down and, as some of the parallel part tags don't match, *mPar* will also be discharged. If the LSBs don't match, the final match line will stay low, driven from *m3b* and the charging of *mPar* will continue, so its voltage will reach the supply voltage level. Spectre simulations indicate that the time between two accesses is sufficient to fully precharge *mPar* when this form of charge sharing happens.

An interesting situation can occur when the following operations happen consecutively: a search is started and in some row the LSBs all match but the MSBs do not. This will leave *mPar* discharged. A subsequent search is initiated which matches the whole row. This would normally fail because *mPar* was not precharged in the interim. For this reason *evalB* — the inverse of the parallel block evaluation signal *eval* — is connected to the NOR gate within the LSB match logic; this forces a 'no match' for the LSBs, imposing a precharge of the parallel match line (*mPar*) for every search. This implementation was preferred to combining the *evalB* signal with *m3b* in a logic gate to drive *VgndMatch* and *mPar* because it has less capacitive load on *evalB*.

When the CAM is operating in parallel mode, the virtual ground line, *VgndMatch*, of the parallel part of the CAM is always connected to ground through the NMOS controlled by *par*. In addition the NAND gate *G3* isolates the precharging of the parallel part's match line, *mPar*, from the match signal of the serial part, *m3b*. A separate signal, *eval*, is then used for precharging. This signal is gated off in serial mode so that its driver does not consume power.

The search lines of the parallel part need to be forced low while the (parallel) match lines are being precharged. The circuits at the periphery of the CAM ensure that the search lines are only driven during the evaluation phase.

Figure 8.9 shows the most important signals in an SPCAM row for five consecutive operations. The potential charge sharing problem mentioned earlier could occur in the third search (13–15ns), where *VgndMatch* is being charged up together with *mPar*. For the selected precharge transistor size, the problem does

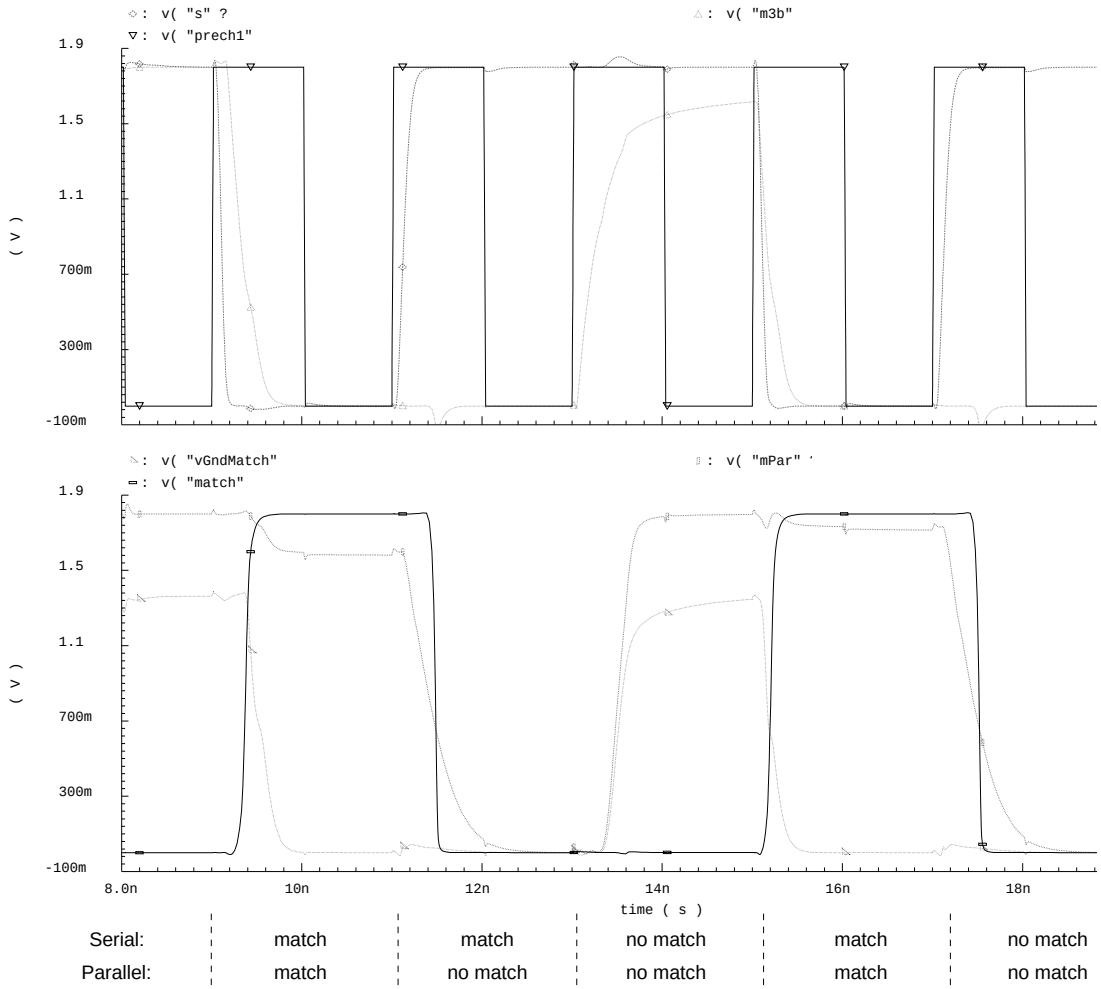


Figure 8.9: Waveforms of SPCAM in serial mode.

not appear here and *mPar* is charged to the supply level.

During the same operation *m3b* (upper group of signals) is shown not to reach V_{dd}; this is the case when it is being pulled high through an NMOS transistor, because the last cell in the serial part matched but the one before didn't. As shown in the waveforms, the circuit correctly reports a 'no match'.

The slowest case for the parallel CAM part, where the parallel match line (*mPar*) is discharged by only one non-matching bit is shown in the second and fifth searches. The discharge time is around 0.5ns and makes the *mPar* evaluation one of the most significant parts of the critical path.

8.5 Results

SPCAM was simulated using the same simulator and technology as the previous designs. The stimuli have a distribution of mismatches at bit positions which follows the findings of the previous section. The simulation results for all the designs are summarised in table 8.1. In serial mode the energy consumption of SPCAM is only 45% more than the RAM (which does not include the decoder and comparator); this is almost a quarter of the standard low-power CAM energy consumption. The cycle time is twice that of the RAM, but only 25% slower than the original CAM. Thus SPCAM is much more energy efficient than the conventional CAM. In parallel mode, the energy consumption is 3.5 times that of the RAM, still 33% better than the conventional parallel CAM, while the performance is the same as the latter.

Table 8.1: Comparison of tag implementations

	Energy per search/access	Cycle time
CAM (32x23)	12.0 pJ	0.8ns
RAM (32x20)	2.3 pJ	0.5ns
Serial SPCAM	3.3 pJ	1.0ns
Parallel SPCAM	8.0 pJ	0.8ns

With a CAM search energy consumption so close to that of a single RAM read, the fully associative cache organization becomes a much lower energy choice than any (pseudo) associative, way predicting cache. Caches with conventional CAMs are reported to have a similar access time to caches with RAM tags [ZA00], thus the effect of the decoder and the comparator must slow down the RAM-based designs to a similar speed to the CAM-based ones. With the results presented here conventional CAMs are only 25% faster than the proposed SPCAM in serial operation. Thus the performance of a cache using the SPCAM should not be significantly slower than a cache built with RAM tags.

The proposed CAM is able to switch from serial to parallel mode, trading energy for speed. In a synchronous processor this is hard to exploit, unless the cache access is made to take two cycles in serial mode and one cycle in parallel. As this design is intended for an asynchronous processor, the variation in speed can be accommodated more easily.

8.5.1 Related work

A recent CAM design by Hsiao, et. al. [HWJ01], claims to be the lowest power CAM yet reported. It evaluates the match lines serially (NAND-type) and does not require discharging of the search lines while the match line is precharged. However precharging and evaluating the match line segments requires more 'clocking' power than the design proposed here. They report a 45.5fJ/bit/search at 12ns cycle time in a .35μm, 3.3V technology. Converting their energy per bit per search result to the technology used here, suggests about 11fJ/bit/search, which is over twice that of the SPCAM in serial mode. Moreover their CAM is fully serial across the length of each line, thus slower than the proposed CAM.

Zhang and Asanovic [ZA00] argue that CAM-based caches are preferable for low-power processors. They describe a CAM design with separate bit and search lines and they precharge the match lines through NMOS transistors to reduce voltage swings. As a speed enhancement they split each match line into two segments which, in view of the analysis here, would also save energy as the most significant part will be discharged infrequently. For further speed improvement they employ single-ended sense amplifiers on both segments of the match lines. The internals of these are not described but probably consume significant power. The energy consumed in the tags is not directly compared in that paper but they showed that the total energy consumption of a cache, with the same configuration as the working example here, is similar to a 2-way associative conventional (RAM-based) cache. Their CAM-based cache has an almost identical performance to a conventional RAM-based cache, which is not phased, i.e. all tags and data are read in parallel in all ways.

Huang *et al.* [HRYT01] compared a number of different pseudo-associative policies. In their results, systems with *Fallback regular*, *Fallback phased* and *Predict phased* policies have very similar energy consumption and delays. Unfortunately they do not compare these results to CAM-based caches.

Burd [Bur01] presented a CAM which consumes twice the energy of an equivalently sized RAM. This is quite surprising because it is a conventional parallel CAM with shared bit lines and search lines. The difference is that the bit lines are pulled up when the match lines are precharged, but one of them still has to be pulled down for each search. With this modification the bit lines are more heavily loaded, with two transistor gates and a drain for each cell. Burd argues that caches using this CAM consume the same energy as a conventional 2-way

set associative cache built with RAM tags, so the CAM-based design is preferable since a higher associativity is needed for his design.

8.6 Summary

A new serial-parallel CAM (SPCAM) design is described in this chapter which consumes about a quarter of the energy of a conventional low-power CAM, when used as a cache tag store/comparator. It exploits the address patterns commonly found in application programs, where testing the four LSBs of the tag is sufficient to determine over 90% of the tag mismatches; the proposed CAM checks those bits first and evaluates the remainder of the tag only if they match. In addition the search lines do not have to be forced to '0' or '1' while precharging the match line, which accounts for almost half of the energy of a conventional CAM.

The proposed CAM is also adaptive, i.e. it can be configured to work serially as described above or it can operate as a parallel CAM with lower energy savings than in serial mode, but at the same speed as a conventional CAM. This adaptivity provides two operating modes which trade speed for lower energy consumption and can be easily exploited by an asynchronous processor.

SPCAM's energy consumption is comparable to that of reading a RAM of similar capacity. Thus using this CAM for the tag parts of a sub-blocked high-associative cache, would make this cache more energy efficient compared to way-predicting (pseudo) associative caches.

Chapter 9

Conclusions

Power-adaptive processors are perfectly suited for low power (and energy), as they are able to scale their power consumption according to their workload. Power-adaptivity by micro-architectural modification has the advantage of very fast transition times between the various power-down modes and the normal, full-speed operation compared to dynamic voltage scaling (DVS). Moreover it can be combined with DVS for even greater power savings; as much as 17% improvement according to Hughes *et al.* [HSA01].

This thesis presented a number of techniques that can enable an asynchronous processor to be power-adaptive, by dynamically changing key parts of its micro-architecture. Although this thesis describes how they are designed in AMULET3, they can be used for any asynchronous processor with minor modifications.

First a simple method for saving energy by delaying the decoding and reading of operands for conditional instructions was designed. It can save up to 13% of the processor core energy consumption and even speed-up the execution for programs with a large number of conditional data-processing instructions.

Two methods for controlling the power consumption of the processor core, by controlling the pipeline occupancy, were designed and implemented at gate level. The first restricts the number of instructions entering the processor pipeline using a FIFO of tokens. The other changes the effective pipeline depth by collapsing pipeline latches, i.e. making them 'permanently' transparent. Both methods achieve energy savings in the order of 15% with a pipeline occupancy of one.

Keeping the processor pipeline occupancy permanently low achieves low power consumption at the expense of excessive speed loss. In order to reduce the speed penalty the pipeline has to be dynamically set to a low occupancy only when

branches are expected. Since common branch prediction is shown here not to be energy efficient, two other techniques were designed. They remove almost half of the speed loss, for a small decrease in energy savings compared to the single pipeline occupancy configuration.

As a first step towards the development of an adaptive memory system, a CAM organisation has been developed for use as tag storage in caches. It consumes only 45% more energy per search compared to an equivalent-size RAM read while improving by about a factor of four the energy consumption of a standard parallel CAM.

9.1 Limitations

The work presented in this thesis is limited by some practical aspects.

The five benchmarks used are obviously not enough for a thorough evaluation of the micro-architectural techniques presented. This shortage is due to a number of reasons.

First simulation speed restricts the size of benchmarks that can be executed. Each of the two large benchmarks, *compress* and *ijpeg* take more than a day to execute with NC-Verilog on a Sun Ultra5 workstation; for example the results in figure 5.13 on page 103 took about a month of processor time to run! For similar reasons such large benchmarks could not be used in the more accurate Powermill simulations.

As the simulation environment was created from scratch there is limited run-time support. A considerable time is required to modify each benchmark in order to use only the provided run-time functions. Thus it was decided that the time spent in making more benchmarks able to execute in the simulation environment was better spent for developing new ideas.

Another practical limitation is the accuracy of the energy estimation method. Although it is adequate for comparing the relatively small differences of the processor configurations presented in this work, it obviously suffers in accuracy compared to a low-level circuit simulator and cannot be as good as a commercial energy estimation tool.

9.2 Future research directions

There are numerous ways to extend the research presented here:

- Most micro-architectural adaptations in the published literature are for superscalar processors and there is much more scope for experimentation in that class of processors. Asynchronous techniques could prove very useful in such an architecture, especially in the design of the issue queue.

- The variation in the data values has great potential for exploitation for energy savings. Although there is some existing work in this area, asynchronous techniques could be used more aggressively to exploit this variation.

- The memory system consumes a large proportion of a system's energy. Dynamically adapting the size of the memory structures while offering a wide range of access times could improve the ability of a system to adapt to varying workload demands.

- Finally, although collapsible pipeline latch controllers were used in this work for saving energy in processors, there may be other applications where they could be of use.

Bibliography

[ABR+01] A. Abrial, J. Bouvier, M. Renaudin, P. Senn, and P. Vivet. A new contactless smart card IC using an on-chip antenna and an asynchronous microcontroller. *IEEE Journal of Solid-State Circuits*, 36(7):1101–1107, July 2001.

[AH98] B. Amrutur and M. Horowitz. A Replica Technique for Wordline and Sense Control in Low-Power SRAM's. *IEEE Journal of Solid-State Circuits*, 33(8):1208–1218, August 1998.

[AH00] B. Amrutur and M. Horowitz. Speed and power scaling of SRAM's. *IEEE Journal of Solid-State Circuits*, 35(2):175–185, February 2000.

[AIC+01] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Architectural and compiler strategies for dynamic power management in the COPPER project. In *Proceedings of the International Workshop on Innovative Architecture(IWIA)*. January 2001.

[AMU99] AMULET Group. *AMULET3H, 32-bit integrated Asynchronous Microprocessor Subsystem. Internal document*, 1999.

[AS00] A. Acquaviva and R. Scarsi. A Spatially-Adaptive Bus Interface for Low-Switching Communication. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, pages 238–240. ACM Press, July 2000.

[Ath01] B. Athas. Asynchronous Design and the Pursuit of Low Power. In *Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, page 2. March 2001.

[BB96] T. Burd and R. Brodersen. Processor Design for Portable Systems. *Journal of VLSI Signal Processing*, 13(2–3):203–222, August-September 1996.

[BB00] T. Burd and R. Brodersen. Design Issues for Dynamic Voltage Scaling. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 9–14. ACM Press, August 2000.

[BBB+02] P. Bose, D. Brooks, A. Buyuktosunoglu, P. Cook, K. Das, P. Emma, M. Gschwind, H. Jacobson, T. Karkhanis, S. Schuster, J. E. Smith, V. Srinivasan, V. Zyuban, D. Albonesi, and S. Dwarkadas. Early-Stage Definition of LPX: a Low Power Issue-Execute Processor. In *Workshop on Power-Aware Computer Systems, in conjunction with HPCA-8*, pages 1–10. IEEE Computer Society, February 2002.

[BBdM00] L. Benini, A. Bogliolo, and G. de Micheli. A Survey of Design Techniques for System-Level Dynamic Power Management. *IEEE Transactions on VLSI Systems*, 8:299–316, June 2000.

[BBS+00] D. Brooks, P. Bose, S. Schuster, H. Jacobson, P. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, M. Gupta, and P. Cook. Power-Aware microarchitecture: design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6):26–44, November-December 2000.

[BdM00] L. Benini and G. de Micheli. System-level power optimization: techniques and tools. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(2):115–192, 2000.

[BdMM+00] L. Benini, G. de Micheli, A. Macii, E. Macii, M. Poncino, and R. Scarsi. Glitch Power Minimization by Selective Gate Freezing. *IEEE Transactions on VLSI Systems*, 8(3):287–298, June 2000.

[BFR94] L. Benini, M. Favalli, and B. Riccò. Analysis of hazard contribution to power dissipation in CMOS IC's. In *Proceedings of the International Workshop on Low Power Design*, pages 27–32. May 1994.

[BLBS+98] E. Boemo, S. Lopez-Buedo, C. Santos Perez, J. Jauregui, and J. Meneses. Logic Depth and Power Consumption: A comparative

Study Between Standard Cells and FPGAs. In *Proceedings of the XIII Design of Circuits and Integrated Systems (DCIS) Conference*. November 1998.

[BM00] D. Brooks and M. Martonosi. Value-based clock gating and operation packing: dynamic strategies for improving processor power and performance. *ACM Transactions on Computer Systems (TOCS)*, 18(2):89–126, May 2000.

[BM01a] R. Bahar and S. Manne. Power and Energy Reduction via Pipeline Balancing. In *Proceedings of the International Symposium on Computer Architecture*, pages 218–229. ACM Press, June 2001.

[BM01b] A. Baniasadi and A. Moshovos. Instruction Flow-Based Front-end Throttling for Power-Aware High-Performance Processors. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 16–21. ACM Press, August 2001.

[BM01c] D. Brooks and M. Martonosi. Dynamic Thermal Management for High-Performance Microprocessors. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 171–182. IEEE Computer Society, January 2001.

[BMC01] M. Bhardwaj, R. Min, and A. Chandrakasan. Quantifying and enhancing power awareness of VLSI systems. *IEEE Transactions on VLSI Systems*, 9(6):757–772, December 2001.

[BOI96] M. Borah, R. M. Owens, and M. J. Irwin. Transistor sizing for Low Power CMOS circuits. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 15(6):665–671, June 1996.

[BP94] T. Burd and B. Peters. A Power Analysis of a Microprocessor: A Study of an Implementation of the MIPS R3000 Architecture. Technical report, University of California, Berkeley, USA, May 1994.

[BPSB00] T. Burd, T. Pering, A. Stratakos, and R. Brodersen. A Dynamic Voltage Scaled Microprocessor System. In *Dig. of Technical Papers International Solid-State Circuits Conference*, pages 294–295. February 2000.

[Bur01] T. Burd. *Energy-Efficient Processor System Design*. Ph.D. thesis, Department of Electrical Engineering and Computer Sciences, May 2001.

[CB95a] A. Chandrakasan and R. Brodersen. *Low power digital CMOS design*. Kluwer Academic Publishers, 1995. ISBN 0-7923-9576-X.

[CB95b] A. Chandrakasan and R. Brodersen. Minimizing Power Consumption in Digital CMOS Circuits. *Proceedings of the IEEE*, 83(4):498–523, April 1995.

[CBB94] A. Chandrakasan, A. Burnstein, and R. Brodersen. A Low Power Chipset for Portable Multimedia Applications. *IEEE Journal of Solid-State Circuits*, 29(12):1415–1428, December 1994.

[CGE96] B. Calder, D. Grunwald, and J. Emer. Predictive Sequential Associative Cache. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture*, pages 244–253. IEEE Computer Society Press, February 1996.

[CGS00] R. Canal, A. Gonzlez, and J. E. Smith. Very low power pipelines using significance compression. In *Proceedings of the 33rd ACM/IEEE International Symposium on Microarchitecture*, pages 181–190. ACM Press, December 2000.

[CHM+01] L. Clark, E. Hoffman, J. Miller, M. Biyani, Y. Liao, S. Strazdus, M. Morrow, K. Velarde, and M. Yarch. An Embedded 32-b Microprocessor Core for Low-Power and High-Performance Applications. *IEEE Journal of Solid-State Circuits*, 36(11):1599–1608, November 2001.

[CKK+97] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, March 1997.

[CPK99] Y.-S. Chang, B.-I. Park, and C.-M. Kyung. Conforming inverted data store for low power memory. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 91–93. ACM Press, August 1999.

[DB94] J. Degener and C. Bormann. GSM 06.10 13 kbit/s RPE/LTP speech compression: C source code, 1994.

[DT99] W. E. Dougherty and D. E. Thomas. Modeling and automating selection of guarding techniques for datapath elements. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 182–187. ACM Press, August 1999.

[EF98] P. Endecott and S. Furber. Behavioural Modelling of Asynchronous Systems for Power and Performance Analysis. In A.-M. Trullemans-Anckaert and J. Sparsø, editors, *Workshop on Power and Timing Modeling, Optimization and Simulation*, pages 137–146. October 1998.

[EFR02] M. Es Salhiene, L. Fesquet, and M. Renaudin. Dynamic Voltage Scheduling for Real Time Asynchronous Systems. In *Workshop on Power And Timing Modelling Optimization Simulation*, volume 2451 of *Lecture Notes in Computer Science*, pages 390–399. Springer, September 2002.

[EG02a] A. Efthymiou and J. D. Garside. Adaptive Pipeline Depth Control for Processor Power-Management. In *Proceedings of International Conference on Computer Design*, pages 454–457. IEEE Computer Society Press, September 2002.

[EG02b] A. Efthymiou and J. D. Garside. An adaptive serial-parallel CAM architecture for low-power cache blocks. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 136–141. ACM Press, August 2002.

[EGT01] A. Efthymiou, J. D. Garside, and S. Temple. A Comparative Power Analysis of an Asynchronous Processor. In *Workshop on Power And Timing Modelling Optimization Simulation*. September 2001.

[FDG+94] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, S. Temple, and J. V. Woods. The Design and Evaluation of an Asynchronous Microprocessor. In *Proceedings of International Conference on Computer Design*, pages 217–220. IEEE Computer Society Press, October 1994.

[FEG00] S. B. Furber, D. A. Edwards, and J. D. Garside. AMULET3: A 100MIPS asynchronous embedded microprocessor. In *Proceedings of International Conference on Computer Design*, pages 329–334. IEEE Computer Society Press, September 2000.

[FES00] S. B. Furber, A. Efthymiou, and M. Singh. A power-efficient duplex communication system. In *Workshop on Asynchronous Interfaces: tools, techniques, and implementations*, pages 145–150. July 2000.

[FGT+97] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N. C. Paver. AMULET2e: An Asynchronous Embedded Controller. In *Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 290–299. IEEE Computer Society Press, April 1997.

[Fle00] M. Fleischmann. Crusoe Power Management: Cutting x86 Operating Power Through LongRun. In *Symposium Record Hot Chips 12*. August 2000.

[Fur97] S. B. Furber. *ARM System Architecture*. Addison Wesley Longman, 1997. ISBN 0-201-40352-8.

[GBB+00] J. D. Garside, W. Bainbridge, A. Bardsley, D. Edwards, S. B. Furber, J. Liu, D. Lloyd, S. Mohammadi, J. Pepper, O. Petlin, S. Temple, and J. Woods. AMULET3i — an Asynchronous System-on-Chip. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 162–175. IEEE Computer Society Press, April 2000.

[GBJ98] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the design of the Alpha 21264 microprocessor. In *Proceedings of the 35th Design Automation Conference*, pages 726–731. ACM Press, May 1998.

[GDE+94] S. Gary, C. Dietz, J. Eno, G. Gerosa, S. Park, and H. Sanchez. The PowerPC 603 Microprocessor: A Low-Power Design for Portable Applications. In *IEEE Computer Society International Conference*, pages 307–315. March 1994.

[GFC99] J. D. Garside, S. B. Furber, and S. Chung. AMULET3 Revealed. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 51–59. April 1999.

[GG97] D. A. Gilbert and J. D. Garside. A Result Forwarding Mechanism for Asynchronous Pipelined Systems. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 2–11. IEEE Computer Society Press, April 1997.

[GGH97] R. Gonzalez, B. Gordon, and M. Horowitz. Supply and threshold voltage scaling for low power CMOS. *IEEE Journal of Solid-State Circuits*, 32(8):1210–1216, August 1997.

[GH96] R. Gonzalez and M. Horowitz. Energy Dissipation In General Purpose Microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, September 1996.

[HGDT84] L. Heller, W. Griffin, J. Davies, and N. Thoma. Cascode Voltage switch logic: A differential CMOS logic family. In *Dig. of Technical Papers International Solid-State Circuits Conference*, pages 16–17. February 1984.

[HKA+01] C. Hughes, P. Kaul, S. Adve, R. Jain, C. Park, and J. Srinivasan. Variability in the Execution of Multimedia Applications and Implications for Architecture. In *Proceedings of the International Symposium on Computer Architecture*, pages 254–265. ACM Press, June 2001.

[HKY+95] A. Hasegawa, I. Kawasaki, K. Yamada, S. Yoshioka, S. Kawasaki, and P. Biswas. SH3: High Code Density, Low Power. *IEEE Micro*, 15(6):11–19, December 1995.

[HL01] J. Henkel and H. Lekatsas. A^2BC: adaptive address bus coding for low power deep sub-micron designs. In *Proceedings of the 38th Design Automation Conference*, pages 744–749. ACM Press, June 2001.

[Hof02] H. P. Hofstee. Power-constrained microprocessor design. In *Proceedings of International Conference on Computer Design*, pages 14–16. IEEE Computer Society Press, September 2002.

[How92] D. How. DES encryption & decryption: C source code, 1992.

[HP96] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, second edition, 1996. ISBN 1-55860-372-7.

[HRYT00] M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas. A framework for dynamic energy efficiency and temperature management. In *Proceedings of the 33rd ACM/IEEE International Symposium on Microarchitecture*, pages 202–213. ACM Press, December 2000.

[HRYT01] M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas. L1 Data Cache Decomposition for Energy Efficiency. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 10–15. ACM Press, August 2001.

[HSA01] C. J. Hughes, J. Srinivasan, and S. V. Adve. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture*, pages 250–261. IEEE Computer Society, December 2001.

[HWJ01] I. Y.-L. Hsiao, D.-H. Wang, and C.-W. Jen. Power Modeling and Low-Power Design of Content Addressable Memories. In *Proceedings of the International Symposium on Circuits and Systems*, pages 926–929, vol. 4. IEEE, May 2001.

[IIM99] K. Inoue, T. Ishihara, and K. Murakami. Way-Predicting Set-Associative Cache for High Performance and Low Energy Consumption. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 273–275. ACM Press, August 1999.

[IM01] A. Iyer and D. Marculescu. Power aware microarchitecture resource scaling. In *Proceedings of the Design Automation and Testing in Europe*, pages 190–196. March 2001.

[JE01] L. Janin and D. A. Edwards. Debugging Tools for Asynchronous Design. In *10th Asynchronous UK Forum*. July 2001.

[KEA60] T. Kilburn, D. B. G. Edwards, and D. Aspinall. A parallel arithmetic unit using a saturated transistor fast-carry circuit. *Proc. IEE*, 107, pt. B:573–584, November 1960.

[KSB02] T. Karkhanis, J. E. Smith, and P. Bose. Saving energy with just in time instruction delivery. In *Proceedings of the 2002 international symposium on Low power electronics and design*, pages 178–183. ACM Press, August 2002.

[LG01] D. W. Lloyd and J. D. Garside. A Practical Comparision of Asynchronous Design Styles. In *Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 36–45. IEEE Computer Society Press, March 2001.

[LGB99] M. Lewis, J. D. Garside, and L. Brackenbury. Reconfigurable Latch Controllers for Low Power Asynchronous Circuits. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 27–35. April 1999.

[MBB01] R. Maro, Y. Bai, and R. I. Bahar. Dynamically Reconfiguring Processor Resources to Reduce Power Consumption in High-Performance Processors. *Lecture Notes in Computer Science*, 2008:97–105, 2001.

[Met96] Meta-software. *HSPICE User's Manual*, 1996.

[MIF02] V. G. Moshnyaga, K. Inoue, and M. Fukagawa. Reducing energy consumption of video memory by bit-width compression. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 142–147. ACM Press, August 2002.

[MKG98] S. Manne, A. Klauser, and D. Grunwald. Pipeline Gating: Speculation Control For Energy Reduction. In *Proceedings of the International Symposium on Computer Architecture*, pages 132–141. ACM Press, June 1998.

[MWA+96] J. Montanaro, R. Witek, K. Anne, A. Black, E. Cooper, D. Dobberpuhl, P. Donahue, J. Eno, W. Hoeppner, D. Kruckemyer, T. Lee, P. Lin, L. Madden, D. Murray, M. Pearce, S. Santhanam, K. Snyder,

R. Stehpany, and S. Thierauf. A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor. *IEEE Journal of Solid-State Circuits*, 31(11):1703–1714, November 1996.

[NNSvB94] L. S. Nielsen, C. Niessen, J. Sparsø, and K. van Berkel. Low-power operation using self-timed and adaptive scaling of the supply voltage. *IEEE Transactions on VLSI Systems*, 2(4):391–397, December 1994.

[NS99] L. S. Nielsen and J. Sparsø. Designing asynchronous circuits for low-power: An IFIR filter bank for a digital hearing aid. *Proceeding of the IEEE*, 87(2):268–281, February 1999.

[OS02] V. G. Oklobdzija and J. Sparsø. Future directions in clocking multi-ghz systems. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 219–219. ACM Press, August 2002.

[PBB98] T. Pering, T. Burd, and R. Brodersen. Dynamic Voltage Scaling and the Design of a Low-Power Microprocessor System. In *Power Driven Microarchitecture Workshop, in conjunction with ISCA*. July 1998.

[PBB00] T. Pering, T. Burd, and R. Brodersen. Voltage scheduling in the lpARM microprocessor system. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 96–101. ACM Press, August 2000.

[PM02] P. I. Pénzes and A. J. Martin. Energy-delay efficiency of VLSI computations. In *Proceedings of the 12th ACM Great Lakes Symposium on VLSI*, pages 104–111. ACM Press, April 2002.

[RDJ99] A. Raghunathan, S. Dey, and N. Jha. Register Transfer Level Power Optimization with Emphasis on Glitch Analysis and Reduction. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 18(8):1114–1131, August 1999.

[RLB00] P. Riocreux, M. Lewis, and L. Brackenbury. Power reduction in self-timed circuits using early-open latch controllers. *IEE Electronics Letters*, 36(2):115–116, January 2000.

[RVR98] M. Renaudin, P. Vivet, and F. Robin. ASPRO-216: A standard-cell QDI 16-bit RISC asynchronous microprocessor. In *Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 22–31. March 1998.

[SB95] M. Stan and W. Burleson. Bus-invert coding for low-power I/O. *IEEE Transactions on VLSI Systems*, 3(1):49–58, March 1995.

[SB97] M. Stan and W. Burleson. Low Power Encodings for Global Communication in CMOS VLSI. *IEEE Transactions on VLSI Systems*, 5(4):444–455, December 1997.

[SD95] C.-L. Su and A. M. Despain. Cache design trade-offs for power and performance optimization: a case study. In *Proceedings of the International Symposium on Low Power Design*, pages 63–68. ACM Press, April 1995.

[Seg96] S. Segars. *Low Power Microprocessor Design*. Master's thesis, Department of Computer Science, University of Manchester, 1996.

[Seg97] S. Segars. ARM7TDMI Power Consumption. *IEEE Micro*, 17(4):12–19, Jul/Aug 1997.

[Seg98] S. Segars. The ARM9 family-High performance microprocessors for embedded applications. In *Proceedings of International Conference on Computer Design*, pages 230–235. IEEE Computer Society Press, October 1998.

[Sei80] C. L. Seitz. System Timing. In C. A. Mead and L. A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[SF01] J. Sparsø and S. B. Furber, editors. *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001. ISBN 0-7923-7613-7.

[SKO+97] H. Sanchez, B. Kuttanna, T. Olson, M. Alexander, G. Gerosa, R. Philip, and J. Alvarez. Thermal Management System for High Performance PowerPC Microprocessors. In *Proceedings of the IEEE*

COMPCON 97, pages 325–330. IEEE Computer Society Press, February 1997.

[SL01] I. Sutherland and J. Lexau. Designing fast asynchronous circuits. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 184–193. IEEE Computer Society Press, April 2001.

[SNNS93] J. Sparsø, C. D. Nielsen, L. S. Nielsen, and J. Staunstrup. Design of Self-Timed Multipliers: A Comparison. In S. B. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 165–179. Elsevier Science Publishers, 1993.

[Spe99] C. Spears. Verilog PLI for file input/output, 1999.

[SSH99] I. Sutherland, B. Sproull, and D. Harris. *Logical Effort: Designing Fast CMOS Circuits*. Morgan Kaufmann Publishers, Inc., 1999. ISBN 1-55860-557-6.

[Syn98] Synopsys. *PowerMill User Guide*, 1998.

[Tsi87] Y. Tsividis. *Operation and modeling of the MOS transistor*. McGraw-Hill, 1987. ISBN 0-07-100332-0.

[TSR+98] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez. Reducing Power in High-Performance Microprocessors. In *Proceedings of the 1998 Design Automation Conference*, pages 732–737. June 1998.

[Vee84] H. J. M. Veendrick. Short-Circuit Dissipation of Static CMOS Circuitry and Its Impact on the Design of Buffer Circuits. *IEEE Journal of Solid-State Circuits*, 19(8):468–473, August 1984.

[Ver88] T. Verhoeff. Delay-Insensitive Codes—An Overview. *Distributed Computing*, 3(1):1–8, 1988.

[VZA00] L. Villa, M. Zhang, and K. Asanovic. Dynamic zero compression for cache energy reduction. In *Proceedings of the 33rd ACM/IEEE International Symposium on Microarchitecture*, pages 214–220. ACM Press, December 2000.

[Wei84] R. Weicker. DHRYSTONE : A Synthetic Systems Programming Benchmark. *Communications of the ACM*, 27(10):1013–1030, October 1984.

[YL02] J. Yi and D. Lilja. Improving processor performance by simplifying and bypassing trivial computations. In *Proceedings of International Conference on Computer Design*, pages 462–465. IEEE Computer Society Press, September 2002.

[ZA00] M. Zhang and K. Asanovic. Highly-Associative Caches for Low-Power Processors. In *Kool Chips Workshop, in conjunction with MICRO-33*. December 2000.

[ZGR00] H. Zhang, V. George, and J. M. Rabaey. Low-Swing on-chip signaling techniques: effectiveness and robustness. *IEEE Transactions on VLSI Systems*, 8(3):264–272, June 2000.

[ZS02] V. Zyuban and P. Strenski. Unified methodology for resolving power-performance tradeoffs at the microarchitectural and circuit levels. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 166–171. ACM Press, August 2002.