Branch Prediction Strategies for Low Power Microprocessor Design

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF SCIENCE

Richard York
Department of Computer Science
1994

Contents

List of Figures

List of Tables

Abstract

Current interest in lower power design has arisen from two areas of application. The first is the fast emerging market for portable, battery-powered, equipment which often requires significant computing power. Secondly, for very high performance processors, there are limits to the heat that can be successfully removed from the package; this in turn puts upper limits on the number of transistors that can be fabricated in a single package. With the number of transistors on a single chip likely to rise to 100 million by the end of the decade the problems of power must be tackled now.

This thesis first examines circuit-level and architectural factors which affect power consumption, with the latter considered in more detail. Pipeline occupancy is identified as being important in many systems for both high throughput and power efficiency. Branch prediction is often used to reduce pipeline stalls; later chapters examine branch mechanisms currently in use and possible branch prediction schemes for accurate speculative execution. The architecture and design of a branch target cache for AMULET2, a low power asynchronous microprocessor, is considered. Finally possible power savings are evaluated and further schemes yielding much higher prediction accuracy are considered.

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 institute of learning.

Copyright

Acknowledgements

The Author would like to thank many people who have given a great deal of help and advice during the years research. These include all of the members of the AMULET group for their readiness to give advice and for providing useful feedback on my ideas.

My supervisor, Dr. J.V. Woods for much time spent discussing the work undertaken and for advice in preparation of this thesis.

Prof. S. B. Furber for suggesting the course of part of my research, particularly the Branch Target Cache, and for discussing the finer points of its architecture.

Much of the software used for benchmarking the BTC is from the public domain, and I would like to thank the authors for releasing the software for general public use.

Finally I would like to express gratitude for the grant I received during my MSc, in the form of an ARM scholarship jointly funded by the Department of Computer Science and Advanced RISC Machines Limited.

The Author

The Author obtained a B.Sc. (I) in Computer Engineering from the University of Manchester in 1993. This thesis reports the work carried out during the Author's year as a member of the AMULET research group at the University of Manchester.

The AMULET (Asynchronous Microprocessor Using Low Energy Techniques) group comprises a number of research projects exploring the potential of asynchronous logic for low power and high speed applications.

1. Introduction

In recent years the design of low power systems has become very important, with the demand for portable computer equipment such as PDA's (Portable Digital Assistants), notebook computers, communications devices etc., creating a large demand for low power IC's. The combination of demanding applications such as handwriting recognition, which require high processing performance, and long battery life has driven chip designers to concentrate on the power consumption aspects of microprocessors and associated peripheral chips, such as ethernet and RS232 drivers, hard and floppy drive controllers.

The need for more power-efficient desktop machines should also not be ignored. It has been estimated by the US Environmental Protection Agency (EPA) [22] that at the current time around 5% of the USA's commercial sector energy requirements is for desktop computers, rising to around 10% by the end of the decade.

For many desktop computers lower power consumption, or more accurately lower heat output, has become a necessity. Processors such as the DEC Alpha [28] dissipate around 30 watts. If this is not successfully removed, overheating and unreliability will occur. Effective removal of large amounts of heat requires careful design of the heat sinks, air flow etc, and the inclusion of one or more (noisy) fans to ensure an adequate airflow. In a portable machine, where there are many more restrictions on air flow, fans are normally impractical and so there are clearly limits to the amount of heat which can be removed. An Alpha-based notebook machine would obviously be quite a challenge!

This thesis suggests architectural improvements which will allow much lower power processors to be built.

1.1 Current Power Consumption Requirements

Research at Intel [24] has shown that around 50% of power consumed in an office is used by PC's and monitors. Of these the majority of the power used is in the monitor and the often inefficient power supply. The same often applies to portable computers, though the power supply is normally much more efficient. A breakdown of the power consumption for each component is shown below in table 1.1.

Component	Power (Watts)
CPU+2Mbytes memory	3.65
Screen backlight	1.43
LCD	0.32
Hard drive motor	1.10
Maths coprocessor	0.65
Floppy drive	0.50
Keyboard	0.49
Interfaces	0.07
Total	**8.21**

Table 1.1 : Power consumption of portable computer components

The table demonstrates how the processor and memory system constitute 44% of the power in a notebook machine. Other factors such as the back light, screen and hard drive are also significant, and in particular the provision of the back light may seem an unnecessary luxury. Removing it, however, is a problem unless display technology becomes significantly better.

An average PC at the current time may use around 1300kWH per year. Cutting this by simple techniques such as automatically shutting the machine down when not required (a PC is in use on average only 15% of the time) will provide significant savings. Further savings will require much more effort, with power regulation of individual components of the PC becoming necessary. Improvements in power consumption of the system are made more difficult by the ever **increasing** power demands of micro-chips. This trend will have to be reversed or will cancel many of the gains currently being made in the rest of the system.

1.1.1 Heat Extraction and Power Distribution

For any type of IC package there is a limit to the temperature that the die can tolerate before it fails to function correctly. This is normally around 175°C. The heat generated must be conducted through the packaging and into the surrounding fluid, normally air. This heat conduction causes a temperature gradient from the die to the outside of the package, and this must be taken into account when carrying out package and die temperature calculations.

The more conductive the package the faster the heat is removed from the die for a certain fluid temperature around the package. A plastic package is normally rated at around 1 watt; a ceramic one around 10 watts. Also the effect of temperature on the lifetime of the silicon should not be ignored. It is estimated that up to 150°C the life expectancy of a semiconductor junctions halves for every 10-15°C rise in temperature. Above 150°C the lifetime halves for every 5-10°C. It is likely therefore that any device that can claim very low heat output should also be able to claim increased long term reliability.

Active and Passive Heat Sinks

The addition of a heat sink allows the surface area and therefore the dissipation of the package to be increased. A typical IC heat sink will have a thermal resistance of around 40°C/watt, ie for every watt that it dissipates its surface temperature rises by 40°C above the ambient temperature. Heatsinks increase the bulk of the package, and need plenty of air space around them to work correctly, making highly integrated systems difficult to build.

An active heat sink employs a component such as a Peltier-effect heat pump to extract heat from the package. Many high-power processors now build a small fan into the heatsink to force air past the fins. Other technologies have been developed such as the thermosiphon employed in the DEC BIPS chip (a 150W ECL microprocessor) [23]. This consists of a sealed boiler and condenser using a volatile liquid to transport the heat efficiently to air-cooled external fins. This has a thermal resistance of around 0.32°C/watt, allowing the die to consume 150W while being maintained below 100°C. The designers have estimated that in quantity the cost excluding the die would be $150. This is not a very practical technology for high volume microprocessors, but does demonstrate what is possible at the current leading edge.

Current Supply and Distribution

The higher currents required for devices such as the BIPS chip have to be supplied and distributed around the die. This normally requires a large number of pins exclusively for power and ground, plus large power rails on the die itself. The BIPS part has 162 power connections, constituting 26% of the pins, and gold bus bars over the die to distribute the current. In the design of PCBs it is quite normal to allocate two complete layers to power supply and ground. This may also become necessary on chip as the transient currents of an IC continue to rise. These high currents are worsened by the global clocking enforced by synchronous processors. The DEC Alpha for example has to supply transient currents of approximately 80A on each clock period [28].

Cost

With all of these schemes the cost of the packaging and cooling employed rises very rapidly as soon as the power consumption rises above a few watts. For volume production of low cost electronics these techniques are too costly and normally the only available packaging is plastic or perhaps ceramic.

1.2 Battery Technology

In a portable system the source of power is often rechargeable batteries. The type and number of batteries is of major importance; too few and frequent charging is required; too many and the machine becomes large, heavy and expensive. Batteries are also difficult and expensive to dispose of due to their heavy metal content; this is compounded by their relatively fixed and often short life span.

The voltages required by the system can restrict the choice of battery voltage. There are often a number of supplies required, for example ±12V, +5V and 20V+ to drive an LCD. Either these are supplied direct from the battery or DC-DC converters are employed to generate all of the potentials from one supply. These converters have, in the past, been rather inefficient, often no better than 70%, though recently this has been pushed up to 80-90% over a large range of current loads (150:1). With the movement towards lower chip supply voltages (see section 1.3.2) this problem is likely to get worse, with systems needing a mix of 2V, 3V, 3.3V and 5V supplies!

1.3 CMOS Power Dissipation

For low power design the static consumption is very important. This is defined to be the power drawn by a circuit when its components are not switching. The dynamic consumption is the power required by circuits when they are switching. Most new VLSI design is implemented in CMOS technology since it has the useful characteristic of very low static current consumption. This generally allows a system doing no work to use zero power. Technologies such as bipolar ECL have large static currents, with much smaller dynamic effects and are therefore of less interest to the low power designer.

Energy is used every time a CMOS gate switches its output. The factors that should be considered for power consumption in a system are :-

1. The number of gates.
2. The size of the gates.
3. The track capacitances.
4. The energy consumed per gate per transition.

The energy dissipated for each gate output is easily calculated. This is given below:

$$Energy = \frac{1}{2}C_L \Delta V^2$$

Where C_L is the total switching capacitance of the gate and ΔV is the change in output voltage. This equation represents the energy required to charge or discharge the output capacitance seen by the gate. This occurs when the output switches, either from a low to a high or a high to a low level.

Other lesser factors affecting energy consumption include the short circuit and leakage current and are discussed later. In a synchronous system there is likely to be a global clock which drives many gates. If this clock runs at a frequency **f** the power is often given for the system as:

$$Power = C_L V^2 \cdot f \cdot n$$

In this case **n** is the average number of gates that switch per clock. This will be inaccurate since it makes assumptions about the average load (and voltage swing for dynamic circuits) of each gate.

To improve the energy consumption of a system there are two major factors in the power equations that can be reduced; these are the switching capacitance C_L and the supply voltage V_{dd}. The former is generally referred to as nodal capacitance and will be examined first, by looking at the circuit technology used for CMOS design.

1.3.1 Circuit Technology

The term ‘circuit technology’ generally includes the cells and components with which the design is implemented, for example the *cell library* generally supplied with a VLSI CAD package. This cell library provides a set of components with which the designer can implement the required logic. This usually consists of basic gates, eg AND and OR, but may also apply to much larger components such as register cells and ALU ‘slices’. Cells may also be custom designed by the engineer, for critical parts of the system, for example the data-path. This allows much greater control over the characteristics of the cells, for example propagation delay, input capacitance etc. which all affect the power consumed.

To study the effects of the different parameters consider the simple CMOS inverter shown in figure 1.1:

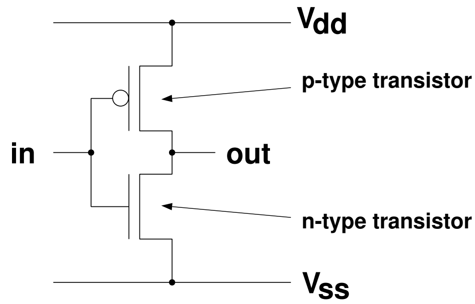

Figure 1.1 : Simple CMOS inverter

The layout of the inverter allows a number of different parameters to be varied. These include the individual transistor sizes and cell topology. Varying these parameters allows a cell to be designed with characteristics precisely tuned for the particular circumstances. Some of these parameters are now considered in more detail.

Transistor Sizing

The gain of a MOS transistor is set by the ratio of the gate to channel lengths. Normally this ratio is shown as W/L. The larger this ratio the greater the gain of the transistor, but also the greater the input capacitance due to the larger gate area. A p-channel transistor is weaker for the same W/L ratio than an n-channel one; both must be of equal gain for equal rise and fall times when driving capacitive loads, ie other gate inputs. The increased input capacitance of the wider p-channel gate, needed for equal transistor gains, presents an increased load to preceding stages. Relaxing the requirement for equal rise and fall times will therefore allow a reduction in nodal capacitance.

Short Circuit Current

As the input to a gate rises or falls both transistors in the stack may be momentarily turned on, and current flows direct from V_{dd} to V_{ss}. The duration of the short circuit current is proportional to the rise time of the input; the slower the rise time the longer both transistors are conducting and therefore the greater the energy dissipated. The rise time is affected by the nodal capacitance; the larger the capacitance the slower the rise time. Short circuit current is also affected by the threshold voltages of the transistors. This is the point at which the transistor begins to conduct, and if $V_{dd} = V_{tn}+V_{tp}$ (threshold of n and p transistors) as one transistor switches off the other switches on, resulting in no short circuit current.

Summary

The parameters considered above are clearly all interlinked with, for example, the transistor ratios affecting the output edge times, which consequently affects the short circuit current in the gate being driven, requiring an adjustment in the transistor ratios.... The development of cells must be an iterative process in which the effects of combining different cells with interconnecting tracks and gate loads is considered and simulated until the required speed-power performance is achieved.

It is clear from this that the semi-custom designer (who uses only the provided standard cells) has less control over the power consumption of the circuit than a full custom designer. What is needed is a way of allowing the possible cell parameters, such as the transistor sizing, to be optimised by the designer so that energy is expended only where necessary. A power simulator can aid this process by allowing the designer to study the areas of the circuit which are contributing most to the energy consumption.

1.3.2 Lowering the Supply Voltage

A reduction in power can be obtained by simply lowering the supply voltage. Since the power is proportional to V^2 (section 1.3) significant reductions can be made. The lower supply does however increase the propagation delay and the rise and fall times of the gate. In detail the rise time is given by:

$$T_{pRise} = \frac{2.57C_L}{\beta_p V_{dd}}$$

Where β_p is the gain of the p-channel transistor and C_L is the load capacitance of the gate. The derivation of this can be found in [46]. To evaluate T_{pFall} substitute β_n for β_p. This shows that both T_{pRise} and T_{pFall} increases as V_{dd} drops (speed inversely proportional to V_{dd}), whereas the energy decreases proportional to V_{dd}^2 (energy per transition proportional to V_{dd}^2), making a reduction in supply voltage beneficial. Lowering the supply from 5V to 3V reduces the power by approximately 2.7 times but also reduces the speed by 1.7 times, for the same capacitive load. As process technology improves the feature size will reduce; this results in a reduction in the gate capacitance which reduces the propagation delay of the cells. The speed of the interconnections does not however scale so easily [56], and as feature size falls the interconnect delays will predominate.

Currently 3.3V and 3V are popular supply voltages. Further reductions may require that the threshold voltage of the MOS transistors be decreased, something that becomes difficult at low voltages, for example less than 1V, and the corresponding drop in noise immunity will cause other problems.

Reductions in the supply voltage will continue, though only where it is acceptable to sacrifice some of the performance and noise immunity of the system for a reduction in power consumption.

1.3.3 Clock Rate Reduction

In a synchronous system, since power is directly proportional to clock frequency, **f**, halving the system clock rate will also halve the power consumed. This is a simple technique often used in portable equipment, where the processing load is constantly evaluated and the system clock rate adjusted to suit. Complete shutdown of the system is also possible, with some form of restart mechanism then required to react to external events that require processing activity. In both cases the energy used to carry out a required task does not change; the system is ensuring that, when there is no activity, the processor doesn't waste energy.

1.4 Improved System Architecture

Architecture can best be described as the high level structure and strategy selected by the designer for the system. Decisions such as the cache structure, size of register bank, the number of bits in a register and the external size of data buses all constitute the architecture of the microprocessor and can have a significant impact on its throughput and power efficiency.

1.4.1 Cache Architecture

Most processors include a cache either on or off the chip. The cache is normally situated between the processor and the main memory and decouples the two components (figure 1.2). A cache relies on the fact that at any time there is a current *working set* of data and instructions in use by the program. This is due to *temporal and spatial locality* - the former indicating that over a certain period of time there will be a number of commonly accessed elements and the latter that there is a high probability that if location x is read, location x+1 will also be read immediately. This working set is held in the cache and any requests for these data or instructions is satisfied from it without having to access the main memory. This serves to reduce the number of accesses made to the memory and, since the cache access time is normally much faster than main memory, allows the processor to run at a higher throughput. It is only forced to slow down when a request can not be satisfied by the cache and data has to be fetched from main memory.

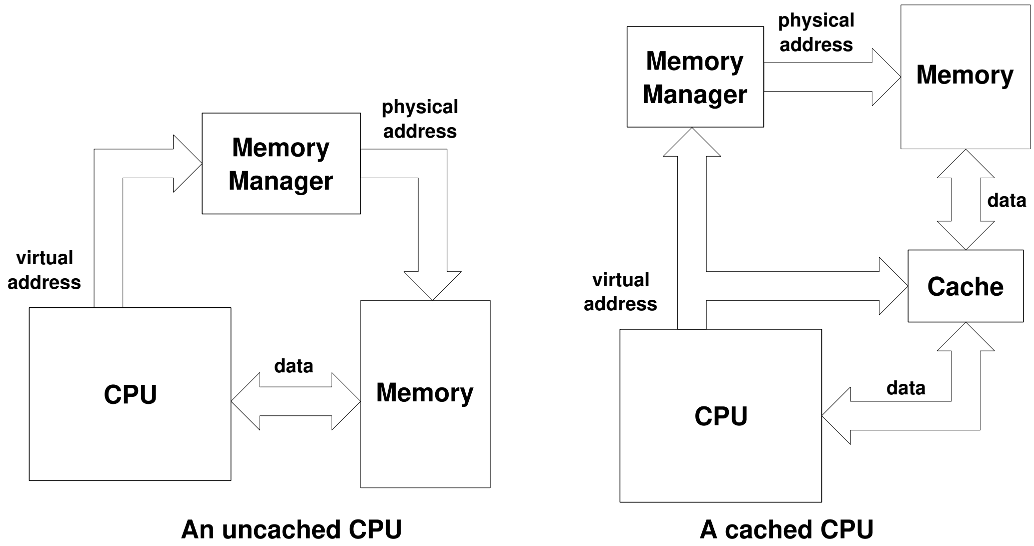

Figure 1.2 : Cached and uncached CPUs

A cache may allow power to be saved by reducing the number of (expensive) accesses to main memory, replacing them with (cheaper) local accesses. A cached processor itself will, however, often consume **more** power than its uncached counterpart. This does not take account of the increased throughput and reduced power in the rest of the system. Overall, cacheing may be used as a power saving feature, with the additional benefit of faster operation, though when low power is relatively unimportant some cache designs will often sacrifice lower power for higher performance.

1.4.2 Register and Operand Accessing

A microprocessor spends much of its time shifting data from one place to another, with some intermediate processing. The movement of data from place to place inevitably involves the use of buses, and the more global the bus the more power and time is required to drive it. To move data from the (off-chip) main memory to an (on-chip) cache costs around 10 times as much as moving data from the cache to the execution unit. This is due to the much higher capacitance of tracks on the PCB and the input load of other IC’s, compared to interconnections within the same chip. This principal also applies on chip, but with proportionally lower costs; a processor’s execution unit may fetch most of its operands from a register bank. Layout considerations result in a 64 entry register bank being further away from the execute unit than one with 16 entries.

Accesses to a larger register bank are likely to cost more in time and energy than for accessing a more compact and local 16 entry alternative.

1.4.3 Operand size

The basic unit of information in a processor is normally the word. The word size is often used to classify the processor type, for example a 16- or 32-bit processor and indicates the size of the internal registers, and functional units that operate on them. It also indicates the width of the address bus for contemporary processors since address or pointer operations require the ability to perform arithmetic on complete addresses, and therefore the register width must be at least as wide as the address bus. Older 8-bit processors utilised 16-bit address buses and had to specify memory pointers using pairs of 8-bit registers. As the word size of processors increases it becomes more important to make full use of the available information content. Using a 64-bit register to manipulate small integers that could easily fit into an 8- or 16-bit register is wasteful since the entire 64-bit data path must be activated. Either the style of programming must improve to allow fuller use of the data path width, or the instruction encoding must allow the use of particular ‘slices’ of the register width. The Motorola 680x0 [52] is one example of a processor allowing byte, half word and word operations on its registers.

1.4.4 Instruction Bandwidth and Density

Each operation a processor carries out is specified by an instruction read from memory. Once read it must be decoded to establish what operations must be carried out. The size of the instruction and the way in which the operation is encoded affects the ease of decoding. A ‘typical’ 32 bit RISC processor, such as the ARM or SPARC, uses a 32-bit fixed length instruction. The semantic content of the instruction will vary depending on the complexity of the operation required. A simple arithmetic operation such as the addition of two registers, writing the result to a third, may only need around 16 bits of information (4 bits to specify each register plus a further four for the operation); thus with a fixed 32-bit instruction approximately 50% of the instruction is unused. A more complex operation such as the restoration of a number of registers from the stack following a procedure call may require a full 32 bits.

There is always a trade off between instruction compactness and ease of decoding. For the last 10 years, with the popularity of RISC processing, the latter has been considered more important. Recently however, studies evaluating instruction set compactness have demonstrated the apparent wastefulness of 32-bit fixed length instructions [29]. Section 2.6 looks at this in detail, with suggestions as to where the trade offs should be made.

1.4.5 Uses and Effects of Pipelining

A simple processor will normally fetch an instruction, decode it and then carry out the specified operation, before beginning to fetch the next. Most of the processor therefore lies unused for much of the time. For example, while the next instruction is being fetched the decoding logic and ALU are not required.

If a number of instructions are allowed to execute concurrently, each using a different part of the processor, the total time to execute one instruction stays approximately the same but the **rate** at which instructions complete increases. This is known as *pipelining*. Problems are encountered with pipelining when one instruction affects the execution of the next, for example a compare followed by a conditional branch. This is known as a *control hazard* and has to be detected to ensure that the correct result is generated. Similarly if a register is written to by an instruction and then read by the following one, care must be taken to ensure the correct ordering of the operations. This is known as a *data hazard*.

Data and control hazards reduce the throughput of the processor from its theoretical peak. For example, on average, the instruction stream normally consists of around 14% taken branches [30]. This means that a simple three stage pipeline (fetch, decode, execute) with a peak of 1 instruction per cycle actually ends up executing around 0.75 instructions per cycle, a 25% reduction on the peak rate. This is wasteful, since the energy used for speculative execution past taken branches has generated no useful results. To enable a processor to approach its peak rate, branch prediction and other techniques must be employed to avoid pipeline stalls. These are discussed further in section 2.8 and chapter 3.

1.4.6 Parallelisation

Most silicon designers optimise their systems for speed, and one possible technique is to employ circuit parallelisation to reduce the delays through the logic. For example a multiplier may well employ a simple adder, with control circuitry to implement multiplication by repeated addition. The use of multiple adders and carry save adders to merge results together in parallel clearly costs silicon area but can result in significant gains in speed. Another example of the application of parallelism is the ripple-carry adder. This has a very slow worst case delay which must be accommodated by the clock period. If a carry look-ahead adder is implemented, which calculates some or all of the carries in parallel, the worst case delay is reduced, allowing the clock period to be reduced.

If, instead of using parallel hardware to **increase** the throughput **at the same clock rate**, it is used to maintain the **same** throughput at a **lower clock rate** power savings might be made. Under these circumstances power is reduced by dropping the supply voltage [43]. As discussed earlier this will reduce the speed of the system but will also cause a proportionally greater reduction in power.

Pipelining a functional unit can also have a similar effect on clock rate requirements. If an adder operating in one pipeline period is split into two stages, and the previous clock rate is maintained, each stage caries out half the computation in the same time period and therefore can have a lower supply voltage. There is a slight area overhead due to the pipeline registers and an increase in the number of nodes driven, but even taking these into account Chandrakasan et. al. [43] estimate a power reduction of 2.5.

These techniques are of greatest interest in applications where there is a certain throughput requirement, and once this is met there is nothing to be gained by further speed improvements. For example a CODEC (COder-DECoder) or MPEG (a data format defined by the Motion Picture Experts Group for video compression) (de)compression chip. This may also apply to many microprocessor applications. In an application such as a PDA, once the response time for a particular task drops below a fraction of a second there is no point in increasing the speed any further.

1.5 Conclusions

Both the circuit technology and architecture have major effects on the power consumed by a processor. Although they can both be considered in isolation, to achieve the best results they must both be addressed when designing a new processor and its support chips.

This thesis concentrates on the architectural impacts of processor design. The circuit level studies are not of principal interest and they will not be discussed further. The next chapter goes into more detail on the various components of a processor and how design decisions affect the power consumption of the finished product.

2. Architectural Factors

The previous chapter discussed the general factors that affect the energy requirements of a system. Both circuit and architectural levels were considered; this chapter explores the impact of various architectural features in more detail, with suggestions as to where trade-offs and improvements can and should be made.

2.1 Clock Rate

As mentioned in section 1.3.3 reducing the clock rate of a system will generally reduce the power consumption, but not the overall energy required for a task to be completed. In addition in a complex system different components can be run at differing rates depending on the processing required of them. The processor is obviously a prime candidate for varying the clock rate, as are the other related components such as the memory system, i/o controllers etc.

When a processor is running, monitoring the amount of useful processing is easy to achieve. For example, in a multi-tasking operating system, the scheduler can monitor the number of system calls or the CPU load, something already done in most systems (the CPU meter). The scheduler can then adjust the system clock rate to account for a lower CPU load (using either specific instructions, or an external programmable clock divider), or increase it if there is a sudden demand for processing. The rate at which this occurs can vary; with a fine grain scheme slowing the processor down between key presses is possible. The granularity refers to the rate at which the throughput is evaluated and the clock is adjusted.

Complete shutdown of the processor is more difficult, but obviously yields much greater savings when practical, ie where the processing occurs in bursts. Examples might include an embedded system logging data periodically or a PDA which is turned 'off' most of the time, but is actually idling waiting for the user to interact with it. The shutdown is normally instigated by the execution of a *halt* or *wait* instruction and the restart by the occurrence of an interrupt. This is very common in micro-controllers oriented towards embedded systems. The other system components are likely to be controlled using programmable i/o.

Dynamic logic will enforce a lower bound on the clock rate of a system. This is because some of the internal state of the processor is held dynamically using the capacitive properties of the internal nodes. If the clock is taken away for more than a certain time these nodes will discharge and the processor will lose its state. In this case halting the processor completely requires some form of state saving and restoration. For example the MIPS R4200 [31] employs built in clock rate reduction by 75%, plus a state-saving power down mode. In a system built with DRAM (Dynamic Random Access Memory) and other dynamic peripherals, some form of refresh must always be present and for efficient processing on restart the cache contents may also have to be maintained.

The principal difficulty encountered with saving and restoring processor state is that the overheads involved are high, precluding fine grain use of the feature. Granularity refers to the size of the time step over which a change is made; the smaller the time step the finer the granularity of the control. MIPS however claim that the R4200 [31] takes only 10us to preserve total state.

PowerPC

The PowerPC 603 microprocessor has a very aggressive power-saving architecture. A number of shutdown features are present and these are;

- Doze mode puts the 603 into a state where all activity is stopped except for Time Base/Decrementer updates and bus snooping. Whilst in this mode a number of events, including an external asynchronous interrupt, a system management interrupt, a decrementer interrupt, a reset (hard or soft), or a machine check will bring the 603 back to normal operation.

- Nap mode provides further savings, with only the Time Base operating. During the Doze and Nap modes, the PLL (Phase-Locked Loop) continues to run and the transition to the full-on state takes only a few processor cycles after an interrupt assertion.
- Sleep mode provides the lowest power consumption. Whilst in this state no functional units are operating and in addition the PLL may be shut down. To return to normal operation the system logic enables the PLL again and then asserts an interrupt. In addition a hard or soft reset may return it to normal operation.

Intel 80x86

Intel's system-management mode (SMM) provides an extra interrupt line (SMI#) to allow a hardware timer to force a trap to power-management software. Many Intel processors are designed using static logic to allow the clock to be halted easily. Extra instructions also provide varying levels of power saving, for example to allow partial shutdown of the core while still carrying out cache snooping to maintain consistency in a multiprocessing environment.

All current processors are designed so that their peak heat output can be successfully dissipated by the packaging. If the processor instead specifies a maximum time it would be allowed to run at full speed, before slowing down, it would be possible to under-rate the packaging. This is unlikely to have a major impact on the usability of the system as most processing bursts would fit into the time periods allowed for full speed running. The throughput would only be affected in the event of a long period of sustained processing.

Conclusions

In most systems containing a microprocessor the processing requirements are sporadic. During periods of relative inactivity the best strategy must be a complete or partial shutdown of the system. This should give almost zero consumption during these times and may be the main strategy employed by PC manufacturers to make their PC's 'greener'. For machines that have a much higher average load, such as a file server, more radical solutions must be employed to reduce the power.

2.2 Clock Gating and Distribution

Clock gating involves disabling the clock signal to a particular area of the chip and is employed in synchronous systems to cut the power consumed by functional units which are currently not required. A typical example might by the floating point unit, which is often idle but will continue to be clocked on every cycle. The gating is likely to be controlled by the instruction decoder, which, for each instruction cycle would only enable those blocks which require clocking.

Clock gating can be a solution to some of the problems of controlling power consumption but it does generate others. The main problem arises due to the extra skew and delay that is inserted into the clock path across the chip. Clock skew is the time difference between clock edges at different parts of the chip. This is already a significant problem for very high clock rate systems such as the DEC Alpha, where the 5ns clock period can tolerate a skew of only a fraction of a nanosecond across the whole chip. Careful simulation has been carried out to verify that the design, which has the clock driving 63000 separate loads, would function correctly. The addition of extra gate delays in the clock distribution introduces further skew, making this verification more difficult.

The PowerPC 603, as well as providing various features to control clock rate, also employs extensive clock gating on a number of units including the fixed- and floating point units, the system unit, the load/store unit and the caches.

Another factor which affects the possible savings to be made are the data buses and control lines that feed the clock-gated blocks. These large busses still have to drive the capacitive loads present, even though the blocks are not using the information being presented to them.

Conclusions

The problems noted above have meant that, until recently, clock gating has not been popular as a power saving feature. For low speed systems and designs where the clock skew can be controlled, clock gating is beneficial. It is not clear what the control overhead would be for cycle-by-cycle gating, but it is likely much of the required control information is already evaluated in the normal instruction decoding.

Local clock generation might also be possible for functional units. This is similar to clock gating, since the unit is only operational when its local clock is activated, but where it differs is in the lack of synchronisation to the global clock. This embodies some of the ideas of asynchronous design, but with the implementational efficiency of a clocked design. Pipelining these functional units may however prove to be difficult.

2.3 Cache Structure

The cache is a critical component of any processor, since its performance has a major impact on throughput [47]. In addition, it is accessed for virtually every cycle of the processor and its size (the cache often dwarfs the processor core) results in its consuming a significant proportion of the total power. Some of the factors to consider are :-

- Degree of associativity. A fully associative cache allows a data element to occupy any entry in the cache. The less associative the cache the more restrictions placed on the locations a particular entry may occupy. This also generally results in a larger cache being required to achieve the same hit rate. The hit rate is the percentage of accesses that are satisfied by the cache.
- Number of lines. A line may hold a number of sequential data words.

- The number of words per line. If there are few words per line the cache utilisation will be high but so will the proportion of silicon used for the tag, since every line requires a tag entry. Many caches have four words (16 bytes) per line. The longer the line the greater the likelihood of 'dead' or unused areas in the cache. It is normal to fetch a complete line on a cache miss; if data is fetched unnecessarily, power and memory cycles will be wasted. Long lines do however allow the exploitation of sequential transfer modes present in most DRAM devices; this saves power in comparison with the same number of 'discrete' accesses.
- Number of read and write ports. For memory-based architectures such as the Intel 80x86 the cache must normally have two read ports and one write port, to allow one instruction to read its arguments while another writes back its result. More ports may be needed if complex features such as non-blocking reads are to be supported. For RISC designs however most operands are sourced from a register bank, reducing the bandwidth required of the (data) cache and allowing fewer read and write ports.
- Write strategy. A cache is normally designed to be either write through or write back. The former directs all memory writes from the CPU direct to memory, updating the cache entry in parallel. The latter writes only to the cache, copying the contents back to memory at a later time. This significantly reduces the writes to external memory, saving bandwidth and power.
- Cache consistency. This is only an issue for multiprocessing systems that share memory. If one processor writes to memory while another holds a cached copy of the original data the latter needs to be told that its copy is no longer valid. There are many protocols suggested to allow support this, such as MESI (Modified, Exclusive, Shared, Invalid). Cache snooping will require extra accesses to check whether bus accesses refer to data present in the local cache, using more power. This may be significant where there are many processors communicating with each other via shared memory.

Cache structures vary from fully associative to direct mapped, with two or four set associativity being popular. The number of sets refers to the number of possible locations in the cache that an individual address can occupy. High degrees of associativity require some form of parallel lookup mechanism, usually implemented as a CAM (Content Addressable Memory). This allows a comparison of the address against all of the set entries in parallel. The power consumption of this block is generally affected by the hit detection mechanism. The ‘hit’ lines normally need to be precharged high on every cycle and then conditionally discharged if any bit fails to match. Lines that are not pulled low are then detected as a hit. Generally all but one of the hit/miss lines will be discharged on every cycle. It is difficult to build CAM cells whose outputs do not swing their outputs to the supply rails, and therefore use less energy to detect a hit. A fully associative cache is shown in figure 2.1.

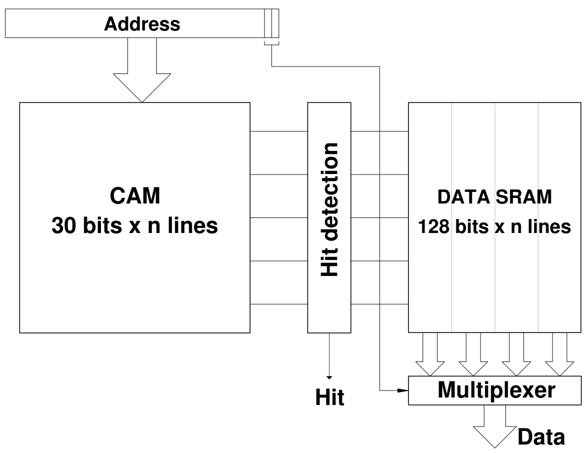

Figure 2.1 : Fully associative cache structure

For low associativity caches the construction is normally based on SRAM technology. A 4-way set associative cache is shown in figure 2.2. A lookup involves selecting a set, normally from 1 (Direct mapped) to 8 (8 way set associative) and then checking the tag to ensure validity. Although a CAM isn’t required, large blocks of SRAM are used, which have long, highly capacitive busses that need large currents to drive them on every cycle. Simple schemes detect differential voltage swings using sense amplifiers. More advanced designs detect current differences in two lines to determine the data.

These are analogue components and consume a lot of power, but reduce the voltage swings on the long hit line busses to a fraction of a volt.

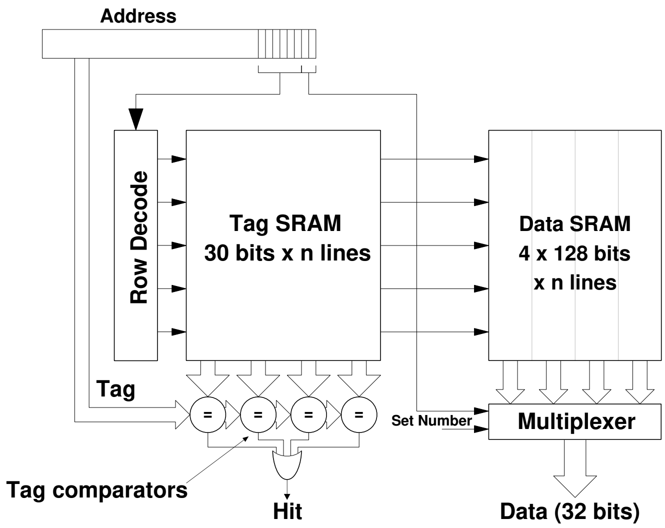

Figure 2.2 : Four-way set associative cache

The real benefit of using low associativity caches is that the access time of the cache is low. CAMs are slow due to the hit detection mechanism and the fact that the data must be accessed **after** the hit detection. A direct mapped cache allows the tag and the data to be accessed in parallel, with the hit detection being done at the same time as the reading of the data. This allows very fast access times, important for high clock rate processors. Of course, for power efficiency, the hit should be determined before the RAM is, possibly unnecessarily, accessed. If the hit rate is shown to be high enough however parallel accessing of the CAM and RAM may be acceptable. More importantly, accessing the tag RAM and data RAM in series can be slow; if the data RAM is accessed in parallel with the tag RAM however, all four words (in this example) must be read, with the tag contents selecting the required word at the end of the access. Thus four times as much data is read than is required on every read.

Multi-Level Caches

A simple single level cache is easy to build and provides reasonable hit rates, but to guarantee near 100% hit rates the cache needs to be very large. Often this can not be built as a single cache and so a secondary level of cacheing before the main memory is added to ‘catch’ accesses that miss in the primary cache. This is often built off-chip and is direct mapped due to its size (256K bytes or more). The power-saving advantages of multi-level cacheing are less clear and need to be examined.

Programming Style

The hit rate of a cache can vary a great deal depending on the access patterns of the programs being run. Cache *thrashing* is a behaviour observed by some programs whose access patterns cause most entries to be thrown out of the cache before they are used again. This is particularly common with direct mapped and low associativity caches, where a large number of addresses map to only a few entries. Thrashing causes a great deal of power to be wasted, both in unsuccessful cache lookups and the resulting very high main memory traffic.

Many architecture handbooks, such as the DEC Alpha and MIPS documents, provide detailed guidelines on how to avoid the problems of thrashing, but this is rarely implemented automatically by the compiler. Because of the effects of thrashing, direct mapped caches have taken time to become widely adopted, but because they are easy to construct are now very popular.

Conclusions

Overall a cache may be used as a power saving feature in a processor because the energy required to access the cache can be much lower than that of main memory. However the cache will also be a major power consumer in a high speed processor. A large, fully associative cache with a simple hit detection mechanism will consume many watts of power and may actually be comparable with the energy costs of accessing main memory. This can be reduced by better hit detection mechanisms to reduce the effect of the inevitable cycling of long bus lines. These include current sense amps to allow swings of a fraction of a volt to be detected. Spatial locality should also be exploited to reduce the number of full lookups on the cache. This can be effective since the line length is chosen to make the best use of the CAM, but often a full lookup is still carried out for each access in the line. If the sequential nature of program code is exploited, many of the cache lookups become unnecessary because of the knowledge that the previous access was a hit and therefore the current request may also hit the same line. The processor core can help the cache in establishing this by providing information to indicate when the address is sequential. It is not then necessary to compare the whole address, but only the lower bits to determine when line ‘wrap around’ has occurred.

If designed with low power in mind the use of a single cache can give good performance and power benefits, but there is still a large difference between the cost of a cache and a main memory access. The addition of one or more further caches allows a finer-grain memory hierarchy to be built. As an example consider a 2 level cache. Closest to the processor is a very fast, cheap cache designed to satisfy maybe 80% of accesses. This is low for many caches but at the next level out is a much larger cache, possibly with a completely different structure, designed to satisfy maybe a further 15%. The remaining 5% of accesses then have to be fetched from main memory. A problem here is that if an item is to be read from main memory a primary and secondary cache access still occurs, wasting power.

With the peak throughput of processors increasing faster than the access times of bulk memory, cache hit rates of better than 98% will become necessary if the processor performance is not to be degraded by cache misses.

2.4 Improving Pipeline Occupancy

The occupancy of a pipeline is important. On every clock period each stage in the pipeline evaluates a result based on the outputs of the previous stage. If the utilisation of the pipeline is low, with bubbles introduced due to data dependencies and taken branches, many of the results are never used. If this is the case a shorter pipeline with fewer interdependencies may give more efficient power utilisation. An example of this is the MIPS R4400 eight stage ‘super-pipeline’ [48]. The branching mechanism has had to be very carefully designed to ensure only a two stage ‘bubble’ is introduced for a taken branch. For pipelined ALU’s, interdependencies between one instruction and the next may cause the pipeline to halt whilst the previous instruction finishes. These problems are caused by the compiler’s inability to separate the generation of results and their reuse in the code. Superscalar processors put even greater requirements on the compiler writers to extract the maximum instruction-level parallelism from the code to maintain power efficiency.

2.5 Register Usage

Virtually all instructions specify a number of operands which must be fetched, processed and then stored. The operands may be simple registers, constants or memory references. By far the most frequent access for a RISC processor is to a register, followed by constants and then memory references least of all. The architecture is likely to be optimised towards the former two access types, such that irrespective of the instruction type the processor will not need to stall. Towards this aim the register bank is likely to have the required number of read and write ports to satisfy all possible instructions, normally two reads and a write per cycle. For a non load-store architecture the frequency of memory operations increases dramatically, with many instructions allowed to specify a memory-addressed operand as a parameter. This is the case for the Intel x86 architecture and here, since the register bank is less well used, it may be acceptable to reduce the number of ports, requiring some infrequent instructions to take multiple cycles to read their operands. This will make it smaller, since fewer read buses and associated hardware are needed, and this in turn is likely to reduce the length of the remaining buses and any parasitic capacitances, making it more power efficient.

In a simple architecture all the reads and writes normally occur exactly as they are specified. Although the instructions specify a particular sequence of operations it may be that a different or simpler set of transfers can be used to achieve the same effect with greater power efficiency. Reusing the last result of an arithmetic operation is one simple example. It has been shown [29] that 29% of ALU results are reused in the next operation, and 20% of produced results are only used once. Detection and use of this information provides the opportunity to reduce register bank operations, replacing them with much simpler result recycling.

Explicit result reuse, using an accumulator or similar structure would make this process simpler since the decode logic does not have to detect when reuse can occur, and allow a more compact instruction encoding. Extension of an accumulator into a stack, eg the T9000 [41], or a queue would allow a greater amount of flexibility in reuse, but with an increased cost for the simple case. The short term storage may well be no more than one or two entries, since it would be difficult to make use of these values across block boundaries especially for a queue. A stack is an example which can be used to pass parameters between code blocks, and the CRISP [32] (and the commercial version, called the Hobbit) optimises this by maintaining the top of the memory stack within a specific on-chip stack-cache.

Register renaming has been employed to increase performance by allowing speculative execution to proceed further. This is unlikely to have a significant effect on power dissipation, since it allows an increased amount of speculative execution, many of whose results will be later thrown away. The only beneficial effect this would have is to increase throughput, allowing the clock rate and therefore supply voltage to be lowered. Generally speculative execution must have a high probability of being correct or power will be wasted in generating incorrect results, ie the pipeline usage will be low.

Conclusions

Some form of result reuse is necessary for reducing accesses to the register bank. Whether the reuse is explicit, using an accumulator or set of accumulators, or detected by the operand fetching unit is perhaps not important, though explicit specification reduces the decode overhead but increases the instruction size.

2.6 Instruction Encoding

The choice of instruction encoding affects a number of parameters in the design of a processor :-

- The available memory bandwidth. The more compact the encoding the less instruction data needs to be fetched from the main memory and cache.
- The complexity of the instruction fetch unit. A compact instruction encoding invariably implies variable length instructions, which are likely to be more difficult to fetch and extract from cache lines than the fixed word-long RISC schemes.
- Instruction decoding. The more compact the encoding the more difficult it is to extract the required information. A simple 32-bit encoding often fixes the position of fields such as the source and destination register numbers, making extracting them trivial and allowing the decode stage to be very short, or non-existent.

Code size assessments have indicated that 32 bit RISC code occupies 150-200% of the space of its equivalent CISC code [30]. A simple test to measure the information content in a binary is to compress it with a standard compression tool, to remove any redundancy in the encoding. SPARC binaries show around a 55% size reduction, but this is likely to be an over estimate due the greater compressability of text embedded within the binary. A study of SPARC and 68020 code size [26] shows that for a number of Unix binaries the SPARC code was only 22% larger than its CISC equivalent. These figures are for **static** code size however, and for power and memory bandwidth it is the average **dynamic** size that is important for power consumption. A RISC instruction set is normally optimised for the frequently executing CISC instructions (the so called 20%-80% rule [30]), and therefore for a dynamic trace the semantic content and size of the average RISC instruction may be closer to that of a CISC instruction. Studies to back this up have not yet been carried out however.

An experiment conducted to measure the memory usage in a typical working environment showed that only approximately 40% of the memory usage of a Sun workstation was for program space. This was for a simple setup of processes, with no major applications running. As applications are run the proportion of code space drops, showing that there is little incentive to increase the instruction compactness, at least for the purposes of reducing memory demands. Improved cache usage is however a more interesting benefit. A compact encoding will allow a larger working set to be maintained in the cache which will help to reduce the effects of cache thrashing.

Another incentive to improve code density is to reduce the required memory bandwidth for fetching instructions. This seems unimportant in cached systems where the cache supplies most of the instructions; when multi-processor systems which share a common bus to memory are constructed, instruction bandwidth does again become important since processors are competing for a share of a fixed bandwidth. The lower the individual demands of a processor the more processors can share a single bus. For example if a processor requires 20% of the bandwidth of a bus, up to four or five could share one bus to the shared memory system.

A decoded instruction cache (DINC) is one way of reducing the problem of reading non word-aligned instructions from memory (figure 2.3). This is a cache of pre-decoded instructions, normally of a fixed width, with information such as the source and destination operands stored in fixed positions. This means that almost all the required decoding for the execution pipeline is carried out ‘in advance’; this reduces the depth of the main pipeline and removes the need to decode some instructions repeatedly. This is of benefit in loops and other repeating structures where an instruction would normally have to be decoded every time it is fetched from the cache. The reduction in pipeline depth also helps reduce the problems of pipeline latency; this causes wasted cycles when executing branches (see section 2.1.8).

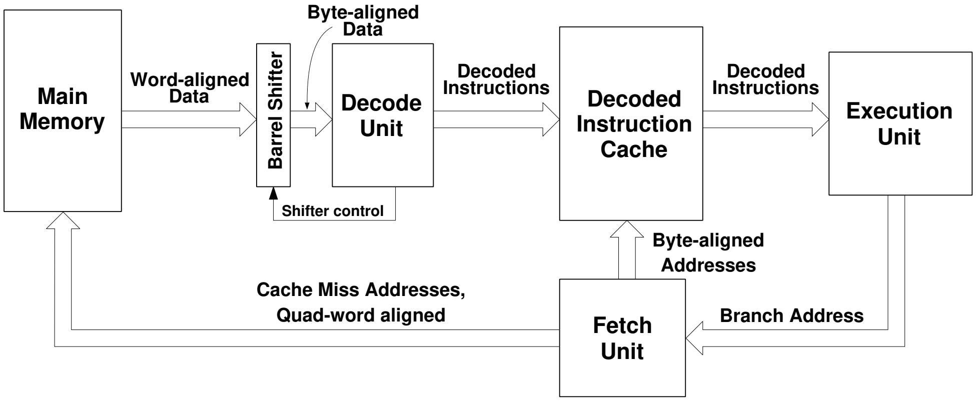

Figure 2.3 : Decoded instruction cache structure

A DINC is likely to be byte addressed, differing from the main cache which is likely to be word or quad-word addressed.

The alternative to a DINC is to decode after the cache (figure 2.4); this maybe of reduced size since the cache stores the instructions in the same compact form as main memory, but will increase the decoding requirement. This is the 'conventional' scheme with a standard cache. A barrel shifter is inserted before the decode unit to extract the instruction from the word-aligned data.

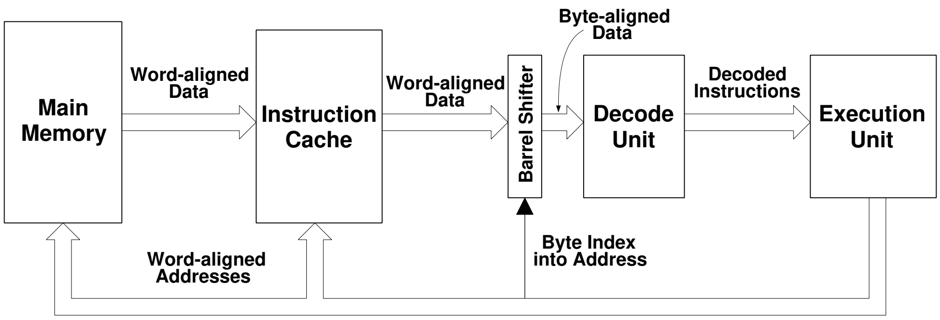

Figure 2.4 : Non-decoded instruction cache structure

The barrel shifter must take in complete cache lines and provide instructions to the decode unit, one per clock cycle. This is a difficult task to accomplish, and is likely to result in the addition of a number of extra pipeline stages to the fetch unit, increasing the latency when a branch is taken. The CRISP [32] uses a small line cache to hold a number of 16 bit packets before the barrel shifter. This is shown in figure 2.5.

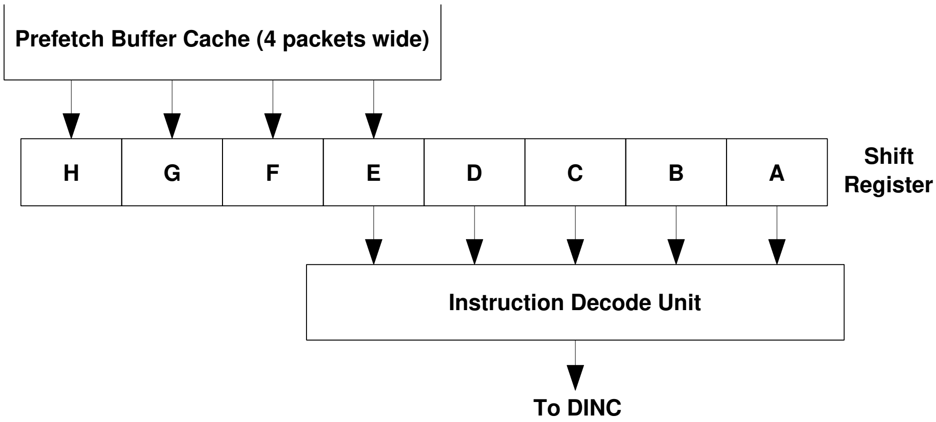

Figure 2.5 : CRISP instruction alignment mechanism

An instruction occupies either one, three or five packets, and as each instruction is clocked into the decode unit the shift register moves to the right to occupy the 'holes' created. Whenever there is space in the shift register four packets are loaded into it from the prefetch buffer cache to keep it full. This mechanism allows one instruction per cycle to be delivered to the decode unit, except where two five packet instructions follow each other; the compiler should rarely need to generate this sequence.

Asynchronous techniques may be of benefit here to build a more efficient realignment mechanism. The addition of an elastic pipeline between the shifting register structure and the decode unit decouples the instruction fetch and realignment from the execute stages. This allows increased tolerance of sequences of full length instructions or stalls due to cache misses. Of course synchronising to this pipeline would then be difficult, and flushing of the queue could cause latency problems but it is an area that is worth investigation.

Conclusions

Extracting and decoding instructions could easily become a bottleneck in a variable length instruction processor. The shifting and queueing needed to extract the instructions from a word-aligned stream adds extra pipeline stages, possibly eliminating much of the power saved by the lower instruction bandwidth. The extra pipe stages also increase the latency of non-sequential instruction fetches after a break in the flow of control. Some of these problems can be addressed by a byte-addressable cache, but an alignment mechanism would still be required.

Another limiting factor is that the lower bandwidth needed to execute a typical program may be largely offset by the extra decode overhead. In this case the instruction may be expanded into a fixed length form, which might be much larger than a standard 32 bit instruction. This is the case for the CRISP where the decoded format is 192 bits wide. The only appreciable gains then appear to be in better cache utilisation. The reduction in main memory requirements, which is largely static, do not seem significant.

A better choice may be to allow variable length instructions, but provide fewer variations, for example 2/4 bytes, to make the extraction simpler. However it may be that simplicity and regularity is best, with better use made of a fixed length instruction format.

2.7 Asynchronous Design

It has been suggested that asynchronous design offers significant advantages when building low power systems, because of its elimination of the global clock, a major power consumer in synchronous systems. The different parts of the processor can become autonomous, self-timed blocks, triggered only when required. This avoids many of the problems of global clock distribution together with the ‘built-in’ clock gating that occurs (functional units are only ‘fired’ when needed). In very high clock rate systems, clock gating (section 2.1.2) is difficult to incorporate because it increases the clock skew that is inevitably present. Asynchronous designs only ‘clock’ the required blocks when needed, so giving the same benefits of cycle-by-cycle clock gating in a synchronous system.

Asynchronous design works well where there are few interdependencies between blocks. If synchronisation is required significant time penalties are often incurred which are not easily hidden; it is therefore important to design an asynchronous pipeline with a smooth and rarely interrupted flow of information from one end to the other. It is also not clear whether the overheads of the asynchronous handshake signals outweigh the advantages gained by tolerance of infinitely variable delays.

The design of a branch predictor for an asynchronous processor is studied in depth later in this thesis.

2.8 Decision Making and Branching Strategy

Measured throughput of current simple and medium complexity processors is fairly low when compared to their theoretical peak. Many pipeline stalls occur due to program interdependencies and changes in the flow of control. The latter forces stalls and/or wasted effort due to incorrect speculative execution past the branch. A general term for a branch is a *control transfer instruction* (CTI). Efficient representation of the conditional code and the ability of the processor to evaluate, in advance, the likely flow of control, results in accurate speculative execution and therefore high power efficiency. The next chapter goes on to examine this in greater detail, exploring different ways of specifying loop and conditional structures and the possible ways of reducing the number of CTI's required in a program. Branch prediction strategies are also examined in terms of the power requirements and the improved performance obtained.

2.9 Conclusions

Clock rate reduction allows the throughput of the system to be closely matched to the processing load required. Asynchronous systems may do this automatically; if no processing is required the processor will 'stall' until the next request for an operation arrives. This stalling may result from an explicit instruction being executed; a polling loop in the program is an inefficient way of waiting for work to be done.

For asynchronous systems, although clock gating can be 'automatic', the problems of buses driving unused functional blocks remains (section 2.2). Buses in asynchronous systems may, in general be kept local, however if a resource is shared, for example a write bus into a register bank, frequently driving a result onto it will be costly. This is offset however by the reduction in the number of ‘ports’ that need to be provided, in this example to the register bank.

A more compact instruction encoding may seem a reasonable way to improve cache usage and reduce the power consumed in fetching the instructions. However the overheads in implementing a variable length instruction processor such as the CRISP seem to give no noticeable benefits over a fixed length architecture such as the ARM when the final power consumption is compared (120mW for ARM700, 210mW for CRISP, with both providing similar throughput). This is likely to because the compact encoding has to be expanded quite early into a fixed length structure, often larger than a normal 32 bit instruction, and the extra stages that this adds to the pipeline can make flushes expensive. The CRISP mechanism of turning all instructions into branches, with destination addresses accompanying each instruction, is interesting but again seems to result in an overhead which cancels out many of the gains produced by branch folding.

In a pipelined system energy must be efficiently used. Pipelines should not be allowed to evaluate incorrect or unrequired results. This restricts the use of speculative execution since energy is wasted if the decisions made turn out to be incorrect. Strictly speaking unless predictions are 100% correct, power is wasted in comparison with a scheme which always stalls until a branch is resolved; few stalling architectures provide an acceptable throughput for current applications however. The branch mechanism chosen for an architecture and the way in which it is implemented will have a major effect on the efficiency of speculative execution. This is due to the high frequency of taken branches found in almost all instruction sequences. The next chapter examines instruction branch strategies in detail, together with an analysis of a number of branch prediction schemes, both past and current.

3. A Review of Instruction Branch Strategies

Branch instructions or CTI's (Control Transfer Instructions) cause the program counter to be altered, forcing the current flow of control through the program to be altered. There are various forms of branch instruction and addressing styles, for example, a relative or absolute address might be provided, and the branch may, in addition to adjusting the PC, save the current PC for future use, such as returning from a subroutine call. The branch may also be conditional on the processor being in some state, for example a particular condition code, or may include a condition evaluation, such as a *test and branch* instruction.

3.1 Programming Style and Behaviour

In many applications the behaviour of the program will be largely driven by the data presented to it at run-time, whether it be interactively provided by the user or from data files. A 'scientific' program however, is more likely to have a specified algorithm which is applied to a data set to produce a result, for example applying a FFT (Fast Fourier Transform - an algorithm used to convert from the time to the frequency domain) to a set of data points. For this style of programming the flow of control is largely independent of the data on which it is operating and therefore a high proportion of CTI's will be deterministic at run-time and often also at compile time [6]. This makes branches much easier to predict, reducing the branch penalty. Even for data-driven scientific processing much of the work is carried out on vector quantities. This means that the instructions must simply keep the vector arithmetic units busy and instruction fetching is therefore less of a bottleneck.

A deterministic branch doesn't have to be unconditional, just that the outcome of the branch can be determined easily in advance. For example consider the following two code fragments:

```
FOR A := 1 TO 10 DO BEGIN
  ...
  ...
END
```

This outcome of the loop closing branch in this example is totally deterministic, just as long as the loop variable is not altered within the loop body. Compare it with this example:

```
REPEAT
  ...
  A := A+1;
  ...
UNTIL A>10
```

In this case it is more difficult for the compiler to establish the end condition because the calculation is buried in the loop body, even though the behaviour may be identical to the previous example.

For most current architectures both sequences are likely to execute in a similar way. The loop closing branch will be coded as a compare of the loop counter followed by a conditional branch to the beginning of the block. The deterministic nature of the block has thus been lost in the compilation process. If instructions are added to allow this information to be preserved the processor should be able to take over some or all of the loop scheduling. This is discussed further in section 3.6.4.

Branches are used in a variety of situations. Normally they are classified as either loop or non-loop branches. The examples above are of loop branches. Other loop branches would be used for coding DO-WHILE structures. A non-loop branch is used for building IF-THEN-ELSE, CASE and SWITCH statements. The predictability of these branch classes has been extensively studied, most recently in [1]. In the course of the work described here the predictability of branches in a range of programs was examined and it was established that loop branches showed a very high degree of predictability (around 90%). Unfortunately non-loop branches constitute the majority of CTI's and these tend to be much harder to predict correctly since their behaviour is much more 'random'. Consider a CASE statement:

```
CASE PollType
BEGIN
  1 : Process1;
  2 : Process2;
  3 : Process3;
END;
```

This is likely to be compiled as register-specified jump into a table of jump instructions which point to the actual code blocks. The jump address is then calculated to point to the correct entry in the table. In many cases the execution path taken is dependant on the data being processed and may be unpredictable. The same also applies to IF-THEN-ELSE structures. Ball et al. [1] showed that non-loop branches form around 50% of all branches. Clearly if these are poorly predictable, obtaining a high level of CTI prediction accuracy is difficult. In modern, highly interactive code, such as 'desktop' style applications, little deterministic data processing goes on. Even if loop branches are reliably predicted, without a non-loop branch prediction scheme the performance will be poor.

3.1.1 Conditional Instruction Skipping

Certain architectures, for example the HP Precision architecture [33] allows the next instruction to be conditionally skipped on the basis of the outcome of a comparison or arithmetic operation. The purpose of this is to try to eliminate short forward branches, typically employed for *if-then-else* type statements, where the code might normally be:

```
  CMP R0,#10      ; does R0=10?
  BEQ skip
  MOV R1,#0       ; clear R1 if it doesn't
skip:
```

This sequence can then be replaced by:

```
CMPSK R0,#10    ; skip the next instruction if R0=10
MOV   R1,#0
```

The MOV is conditionally skipped, based on the comparison in the previous instruction. This will only allow one instruction to be skipped however, restricting its use to very simple statements. A small study of SPARC code was carried out to establish the effect conditional skipping could have on the number of branches executed. The branch statistics for the sample programs is shown below in table 3.1.

Program	**Percentage of Instructions that are branches**	**% of branches forward**
Compiled Renderer	10.9%	77.7%
ls -lg	19.9%	67.8%
sort	18.4%	66.1%
gcc	19.3%	75.3%

Table 3.1 : Basic branch direction statistics

A branch refers to a PC-relative branch instruction, which does not include procedure call and returns. The table shows that forward branches predominate in all of the benchmarks. Looking at the branch offset in each case gives the following results:

	Forward Branch Offset (percentage of forward branches)							
Program	1	2	3	4	5	6	7	⩾8
Compiled Renderer	2.3%	2.2%	11.9%	7.6%	4.7%	4.4%	5.1%	61.8%
ls -lg	3.6%	1.9%	2.9%	6.1%	3.9%	9.7%	8.8%	63.1%
sort	2.3%	0.7%	2.9%	8.3%	4.2%	21%	1.6%	59.0%
gcc	2.8%	0.7%	3.3%	5.1%	4.9%	6.0%	5.8%	71.4%
Average	2.8%	1.4%	5.3%	6.8%	4.4%	10.3%	5.3%	63.8%

Table 3.2 : Size of forward branch offsets

This clearly shows that only a small percentage (2.8%) of forward branches could be eliminated by single conditional skipping, and an examination of some compiled HP-PA code [33] shows that this is often the case, with very few of this type of instruction being generated.

Table 3.1 also demonstrates the frequency of branch instructions. Hennessy and Patterson [30] state that between 1 in 4 and 1 in 5 instructions (20-25%) are branches;

this is born out both by the above figures and by results given later, in chapter 4, for ARM code.

The ARM architecture (see appendix C) allows all instructions to be conditional on various combinations of the four standard (N, V, C and Z) condition codes. This allows much greater flexibility in the elimination of forward branches, since an arbitrary number of instructions can be skipped. For example:

```
CMP   R0,#10
MOVEQ R1,#0     ; only execute if R0=10
MOVEQ R2,#4     ; only execute if R0=10
LDREQ R3,[R4]   ; only execute if R0=10
ADDNE R5,R6,R7  ; only execute if R0 <> 10
```

There is a trade off here between the number of instructions which are skipped and the forward branch distance. Conditionally skipping instructions still requires almost complete execution, with possibly only the write-back pipeline stage to the register bank being suppressed. In terms of the power consumed, very little has been saved, and if the forward branch can be executed with little cost this would be preferable. For ARM code, execution of more than four skipped instructions becomes inefficient in both performance and power.

3.2 Branch Prediction

Branch prediction is a feature provided by many processors to allow the direction of a branch to be predicted, and instructions following the predicted target to be speculatively executed until the correct branch outcome is evaluated. Branch prediction is generally only necessary in a pipelined processor. If the fetch-decode-execute stages of an instruction do not overlap those of another the decision to branch can affect the address of the next instruction to be fetched without interruption or stalling (figure 3.1). This is the case for many simple 8-bit microprocessors.

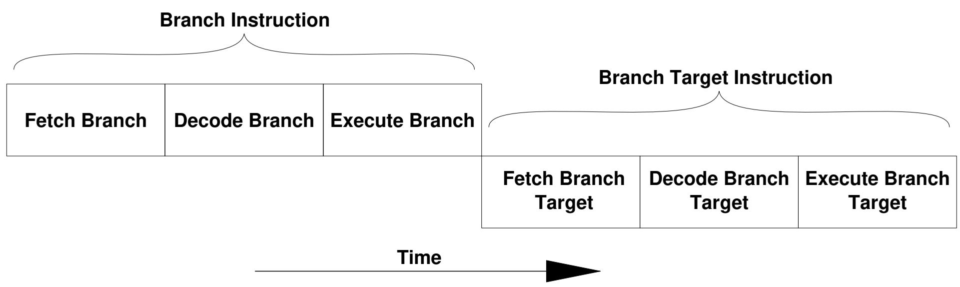

Figure 3.1 : Non-pipelined branch execution

If, however, the execution of instructions is pipelined, the evaluation of the branch condition and/or the target address occurs **after** one or more instructions following the branch have already entered the pipeline. If the MIPS five stage pipeline is examined, the effect that a branch has on the flow of instructions can be seen. The five stages are given below and illustrated in Figure 3.2.

1. Instruction fetch.
2. Register/operand fetch.
3. Execute/ALU.
4. Access memory.
5. Write result.

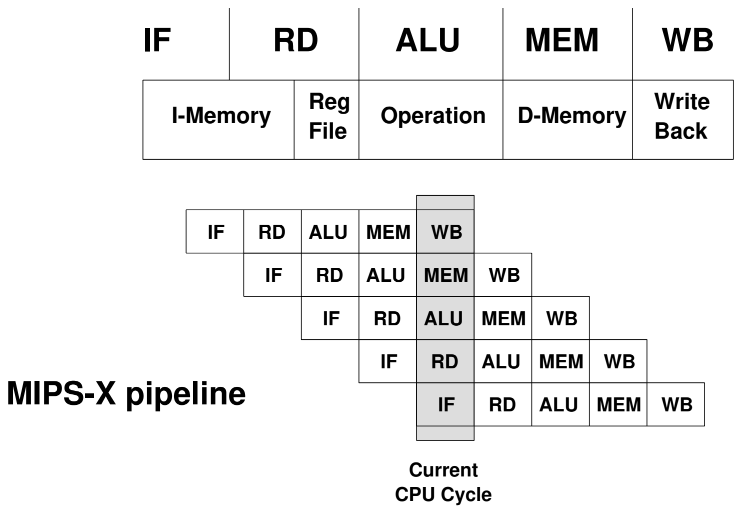

Figure 3.2 : MIPS-X pipeline structure

The branch condition is evaluated in the main ALU during the execute stage. At the end of this stage either the incremented PC or the branch target is sent out to memory. This is shown in Figure 3.3. There are two instructions that have been partially executed and may need to be abandoned if the branch is taken. In the case of the MIPS processor these are classed as branch delay slots (see section 3.5.4).

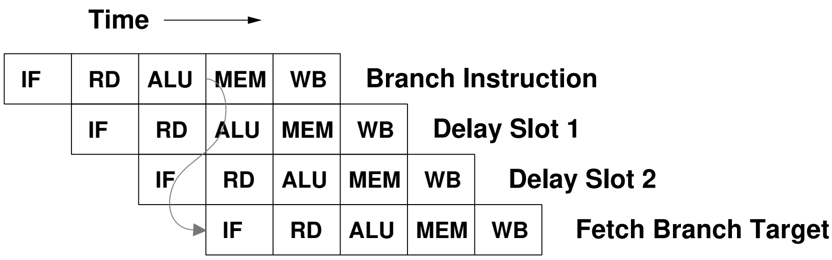

Figure 3.3 : MIPS-X branch execution

3.3 Types of Branch

For a conditional branch to be executed three things must be specified :-

1. The branch condition, to determine whether it will be taken. For an unconditional branch there will be no condition to specify.
2. The branch target address.
3. The actual point at which the flow of control changes.

These are normally provided in one or two instructions, though if the three stages are separated into individual instructions some of them may be implied. For example the omission of the condition specifier results in a non-conditional branch.

Most processors use two instructions to perform a conditional branch :-

1. Evaluate the condition, eg CMP R1,R2 storing the result either in a general purpose register or in specific condition codes.
2. Specify the target and the simple condition to indicate whether to take it or not, eg BNE *address*.

In this case the branch examines the condition codes to test whether an earlier compare resulted in a ‘not equal’ condition. If no condition codes exist, for example the DEC Alpha, a register must be specified, in which case a simple test would be carried out in the branch, such as a test for zero. In this case it seems that more work is required, since it requires a full register read plus ALU operation, whereas the use of condition codes, which require little storage, probably within the execution unit. need little effort to examine.

Early specification of the branch target is possible, using a *prepare to branch* instruction. This allows the branch target to be specified at the start of the code block, with the actual control transfer occurring at the end. The principal advantage of this is when used with non-blocking caches, allowing the branch target to be brought into cache if it is not already there. This is of most use for procedure calls; other types of branch are generally more local in destination and spatial locality is likely to ensure the target instruction is already present in the cache. The average size of a basic block (a non-interruptible code sequence, possibly terminated by a branch) has been shown to be quite small [57], of the order of 8 instructions, and therefore a *prepare to branch* may save say 7 cycles due to a cache miss. For a deeply pipelined processor it will often be the case that the basic block size will fit into the pipeline depth. In this case the prepare to branch itself will have to cause the instructions entering the pipeline to switch to the branch target immediately, if predicted.

Condition Code Usage

DeRosa and Levy [20] investigated the setting and usage of condition codes. They discovered that for a set of VAX instruction traces, 73% of instructions set the condition codes but only 22% of these instructions set them usefully, ie 78% of condition code results were not used before being overwritten. Compare and test instructions formed the majority of the condition-code setting instructions, but again only 9% of these were useful settings. One interesting result demonstrated was that there would be a 6-7% **increase** in the number of instructions executed if condition code setting was removed from the arithmetic-type instructions, resulting in the need to add compares before many conditional branches. An example of this might be a decrement instruction, followed by a branch, which would need to be replaced by decrement, compare, branch.

A similar study has been carried out for the ARM processor [26] to examine how particular features of the instruction set have been used by various optimising compilers. 14% of all instructions were observed to be non-subroutine relative branches, and of these around 80% were directly preceded by a compare. This suggests that although arithmetic operations that set the condition codes are useful (accounting for 20% of branch conditions) the majority of branches need an explicit compare, presumably of a value calculated earlier in the code block. Improved compiler technology may improve this.

In both examples presented, condition code storage could be replaced by the use of general purpose registers, but there is always a need for very short term storage of Boolean results. The first example also demonstrates that allowing too many instructions to set the condition codes results in unnecessary work since a great deal of the setting is not required. It also allows many instructions to affect the outcome of branches, which makes prediction slower, since the processor must be able to recognise many different instruction types so that it is able to wait for condition codes to become valid.

Compare and Branch

To eliminate condition codes a compare and branch instruction can be provided, as for example in the MIPS architecture. This allows the programmer to specify two registers to compare, a condition type and a 16 bit displacement. The requirement for two ALU's in this case (one for the condition test, one for the relative branch offset calculation) is fulfilled by an additional ALU in the address unit. The compare and branch instruction allows the MIPS architecture to eliminate condition codes, whilst avoiding the use of general purpose registers to hold intermediate results; This is rather power inefficient and wastes register capacity, especially for 32 and 64 bit architectures. A compare-and-branch does, however, allow the compare preceding a branch to be combined with it into a single instruction. In the case of ARM code most branches have been shown to require a compare instruction and so would be able to make good use of a compare-and-branch.

Compare-and-branch prevents the condition evaluation and the point at which the flow of control changes from being separated. This prevents early resolution of a branch, which can normally be achieved for separated compares and branches, if *branch spreading* is implemented. The term branch spreading is suggested by Ditzel and McLellan [3] and refers to the process of separating the condition evaluation and the branch instruction within the code block. The aim is to ensure that when a conditional branch is decoded there are no condition-setting instructions ahead of it in the pipeline. This allows the branch condition to be evaluated immediately. This might still be possible for a compare and branch, by recognising the compare part of the instruction, and evaluating it early whenever possible.

Branch spreading may be implemented by adding NOP's (**N**o **OP**erations) between the compare and the branch, which is similar to forcing the processor to stall on a branch fetch. This gives no performance improvements though and a better strategy is to migrate the compare instructions away from the branch. A simple example of this is within a loop. Normally the loop counter will be adjusted, compared and the branch taken at the end of the code block. If the adjust and compare are moved to the start of the code block it will reduce or eliminate the evaluation delay around the branch. This may also be possible with IF-THEN and CASE statements, though the degree to which the compare and branch can be separated may differ in these cases.

Conclusions

The elimination of condition codes is sometimes seen as an architecturally beneficial decision because it :-

- Reduces the amount of state at any one time.

- Makes testing and setting conditions simpler.
- Allows easier superscalar implementation. This is because multiple instructions are issued and complete in parallel and therefore the use of a small number of condition codes can become a bottleneck.

A better scheme would be to provide a set of generalised condition codes, rather like a set of registers. The compare instruction would then have a specified destination instead of an implied condition code flag. This is similar to using the normal registers but is :-

- More power efficient. The flags can be held local to the execution unit and are not full register widths so operations on them are faster and consume less power.
- More efficient on register usage. Complete 32/64 bit registers are no longer wasted holding what is only a single bit of information.
- Allows a more compact encoding; eight flags are likely to be ample for holding temporary condition state.

Arithmetic operations also benefit from keeping condition flags, easing the carrying over of results from one calculation to another. The condition codes could 'shadow' the normal register set, with the destination, either register or flag implied by the instruction type. For example a compare would have a destination register, such as R1. This would be interpreted as condition code 1, instead of the normal 32-bit R1 in the register bank.

The PowerPC architecture implements a condition code register containing eight independent flags, used for storing the result of compares and arithmetic carries and for evaluating the direction of conditional branches.

3.3.1 Branch Target Calculation

Calculation of the branch target varies in difficulty depending on the way it is specified in the instruction. A PC-relative address will require a separate ALU if early fetching is required, since the main ALU may be in use by an earlier instruction. This also applies to compare-and-branch instructions where the main ALU is required for evaluating the branch condition. If the target is an absolute address this is not a problem since an addition is not required, but specifying a full 32 bit address within the instruction is not particularly practical, especially as the address space increases. Absolute addressing will also make code relocatability difficult. A good combination of absolute and relative addressing may be to use concatenation. An n-bit constant field specified in the instruction can be used to replace the lower n bits of the current PC. This allows pseudo-relative addressing within the program, just as long as programs are always loaded on an appropriate modulo-n boundary. This has two useful attributes :-

- An adder is not required since the target address is specified by simple substitution.
- Faster issuing of the branch target address is possible.
- Some code relocatability is possible.

This scheme has also been suggested by Calder and Grunwald [34] who point out that for a 64-bit address space, the use of a 21-bit displacement in the branch instruction allows branching within an 8Mb address space. Thus there would be approximately 2^{33} possible 'segments' of 8Mb each. This segment size should be able to hold almost all programs currently in existence. The ARM allocates 24 bits for branch displacement, allowing branching within a 64Mb space, but this is a true relative offset from the current PC. It seems unlikely that single programs will grow to fill this space in the foreseeable future.

The use of a fixed lower address offset limits the relocatability of the program code since it enforces loading on 2^n address boundaries. To alleviate the problem the program loader can alter the branch offset at load time. Alternatively a smaller offset can be used to allow loading on more address boundaries. Katevenis [44] has also suggested schemes similar to these.

If the target is specified indirectly, for example using the contents of a register or memory location, extra restrictions are imposed since speculative prefetching of the branch target is more difficult. This is because the processor cannot evaluate condition codes without knowing if an instruction is about to alter them. Register-specified jump addresses are useful for function call returns (the return address may be stored in a link register or on a stack) or jump tables. For example:

```
ADD R1,PC,#&10  ; set R1 to point to the jump table
ADD R1,R3       ; use R3 to index into the table
LDR R2,[R1]     ; get address from jump table
JMP [R2]        ; jump to it.
```

Here a jump table is used to implement a switch-like construct, with R3 as the control variable. The branch unit must wait for the load to complete before it can start fetching from the branch target.

This also demonstrates the effect that branching may have on the cache. It is likely that either the jump table or the branch target are not in the data/instruction cache. This will cause the pipeline to stall once or twice during the execution of the code block. If more simple addressing is used the branch unit can begin prefetching from the target before the branch is resolved, so that it is ready to enter the pipeline when needed, hiding some or all of the target fetch latency (due to a cache miss) from the execution unit.

3.4 Branch Implementation

In a pipelined processor there are several ways of implementing CTI's. The mechanism chosen has a major impact on how CTI's affect the performance of the processor. A poorly designed scheme can waste a large proportion of the instruction bandwidth, and cause many CPU cycles to be lost recovering from incorrect decisions or pipeline stalls.

A branch instruction essentially does no useful work. The computational part of the program is in the data processing and manipulation, with branches linking the basic blocks of the program together in the required order. Ideally the execution unit should never need to see a branch at all, with the instruction fetch unit always providing it with the next instruction required, regardless of changes in the flow of control. In such a scheme the instruction fetch unit is logically separated from the execute unit. This is shown in figure 3.4.

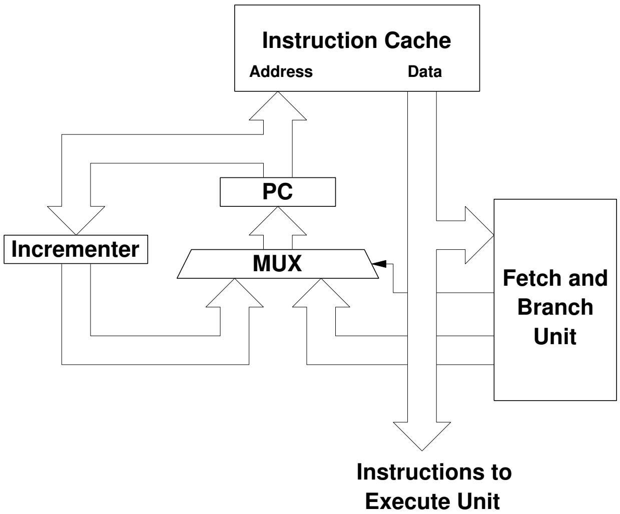

Figure 3.4 : Remote instruction fetch unit

This scheme works well for non-conditional branches, but as soon as a data-dependent branch is executed the prefetch and execute units must 'synchronise' to allow the condition test result to be passed to the fetch unit. In an asynchronous world this is likely to be implemented as a pipeline, which implements both the synchronisation and result passing. If conditional branches occur frequently, however, the resyncronisation time will become a bottle neck.

There are two simple strategies that can be used by a pipelined processor to cope with branches :-

- Whenever a branch is decoded and recognised, normally quite early in the pipeline, further instructions are prevented from beginning execution until the outcome of the branch is known. This is known as *stalling* and may be the most power-efficient technique if performance is not of primary importance. For an unconditional branch, fetching and execution from the new target can begin as soon as it is recognised, providing the target can be easily calculated.
- Execution of instructions following the branch continues until the outcome of the branch is known. The outcome will then affect any instructions following the

branch in earlier stages of the pipeline. For an unconditional branch or a conditional one that passes its condition test the instructions following the branch must be thrown away before the target of the branch can be executed. If the branch is not taken the following instructions can continue to execute as normal.

The second example is actually a simple branch prediction mechanism. In effect branches are predicted not-taken due to the fact that speculative execution of the instructions that follow the branch has begun. This is the mechanism implemented in current versions of the ARM [35].

Evaluating when the branch condition is valid varies in difficulty depending on the number of different ways that it can be set. For example, if condition codes are provided, a branch which is being executed must decide when the codes are valid. If only a small number of operations can alter the codes then it becomes easier to track instructions ahead of the branch in the pipeline. This might be used by the fetch unit to track the progress of a branch instructions to establish when the branch condition is valid.

For example consider the following code sequence:

```
CMPS  R1,R2     ; compare R1 and R2, setting the flags
MOV   R2,R4
ADD   R1,R3,R7
BEQ   label
```

In this sequence there are two non-condition-code-modifying instructions before the branch. This allows the branch unit to execute the branch earlier than would otherwise be possible. This form of optimisation is not available to single compare and branch instructions, though the two instructions before the branch could be compared to branch delay slot entries (Section 3.5.4)

The effect of branches on a pipelined processor is to cause a reduction in the throughput of instructions from the theoretical peak of one per clock. This reduction can be quantified by knowing the frequency and outcome of branches in a program and the depth and structure of the pipeline. These figures vary considerably and depend on a number of factors including the instruction set, the type of program, the compiler used and the level of optimisation employed.

3.5 Past Branch Prediction Schemes

There are many types of branch prediction that have been used in the past and these can be broadly divided into two classes, static and dynamic. Static prediction embeds prediction information generated by the compiler into the instruction encoding, the latter derives prediction information dynamically as the program executes.

Branch prediction may be made harder by other factors in the instruction set. For example, the provision of instructions that have side effects. The ARM processor provides a class of load and store multiple register instructions which provide register pre- and post-index addressing. These allow a base address register to be automatically adjusted before or after load and store operations, for example to maintain a stack pointer. Any such side effects must be fully reversible if speculative execution is to be allowed.

3.5.1 Multiple Instruction Pipelines

A simple pipeline suffers branch penalties because the decision to branch, or not, is often taken at or near the end of the pipeline. If the branch is to be taken, all of the preceding partially completed instructions in the pipeline must be discarded and the pipeline refilled. This delay often seriously degrades the amount of useful work done. A brute force approach to improving this is to follow both possible paths of the branch simultaneously by duplicating some or all of the pipeline stages. When a branch is encountered in the decode stage the target address is calculated and the parallel pipeline is started off down the alternative path. Some time later, when the branch outcome is established at the end of the first pipeline, the decision is made as to which execution pipeline continues and which one is flushed. This allows execution of the branch and its target instructions with very little interruption. While one of the pipelines executes the correct path after the branch the other is flushed, ready to accept the next branch target stream.

This approach has a number of problems, chiefly concerned with the time taken to obtain the target address of the branch. If this depends on a previous result, or an ALU operation is required, for example a PC-relative branch, the parallel pipeline may not be able to start up immediately, so causing a delay if the branch is finally taken before the branch target is fully executed. Also the duplication of hardware is substantial, with two instruction fetch, decode and operand processing units being required, plus some duplication in the register sets. This hardware is unlikely to be efficiently used, since much unnecessary speculative execution along what turns out to be the wrong branch path occurs, making the gains achieved very costly. The scheme may also put a much heavier load on the available memory bandwidth. Despite these problems both the IBM 370/168 and IBM 3033 [16] implement some sort of multiple instruction stream architecture.

The above is mainly a feature used to enhance performance and pays no heed to power-saving; it is not considered further.

3.5.2 Loop Buffers

A loop buffer is a very specific branch prediction mechanism used mainly in the scientific and super-computing environment where the code executed is very loop-based, mainly due to carrying out repetitive vector-type calculations. The strategy is to store the current loop in a small high speed buffer which is filled and maintained by the instruction fetch unit. The loop buffer is really a FIFO queue of the instructions that have entered the main execution pipeline. As a new instruction enters decode it is stored in the FIFO and the oldest is discarded.

When the target of a loop branch is detected in the buffer the contents of the buffer are frozen and the pipeline is then fed cyclically from the buffer until the loop exits. The start and end points of the loop are held so that the last instruction in the loop is immediately followed by the first. If this turns out to be a wrong decision, loop mode is terminated and the pipeline is flushed. If the loop is unable to fit completely into the buffer no use can be made of the structure and the loop-closing branch must be executed conventionally.

A loop buffer can also be used as a prefetch buffer, so that sequential instructions are fetched without the normal memory access latency of individual accesses. Prefetch buffers are common on super-computers such as the Cray1 [36], which has four that are used in a FIFO manner. In this case the prefetch buffer is more like a cache, allowing loop-buffer like access, but in a random way that can support nested loops and subroutine calls within the loop body. Many loops are also eliminated within super-computers by the use of vector instructions.

Loop buffering has not been used in microprocessor design since instruction caches provide many of the same benefits, but in a more flexible way.

3.5.3 Branch Target Prefetching

This is similar to having multiple instruction pipelines, but is simpler. The branch target is calculated as early as possible in the pipeline, and prefetching begins along this path, so that if the branch is subsequently taken the target instructions can enter the pipeline immediately without the normal memory latency overhead. Providing some buffering for this alternative path is available a number of instructions along the target path can be fetched. This may be worthwhile when the cost of fetching further sequential instructions following a non-sequential one is fairly low, though this will only work easily for non-variant branch targets because of the need to issue the target address as early as possible. The addition of an additional simple branch decoder and ALU in the fetch unit would allow earlier recognition of the branch and calculation of the target (for PC-relative branches).

Again, for low power design this may not offer any benefits since there is little intelligence in the fetching strategy, resulting in many wasted instruction fetches.

3.5.4 Delayed Branches

This scheme is conceptually easy both to imagine and build, though not so easy to program. It involves allowing a number of instructions following a branch always to execute, whether or not the branch is taken. In effect the branch at instruction x affects the fetch address of instruction x+y, where y-1 is the size of the branch delay slot. The MIPS series of processors specify a delay slot of one instruction. The ideal size for the delay slot is the number of pipeline stages between the address issue and branch condition test points. If all these slots could be filled the would be no interruption to the execution of instructions.

Although such a scheme is easy to build it can be difficult for a compiler to fill the delay slots, and if no useful instructions can be found to fill the slots, NOP's must be used. Analysis has shown [2] that the probability of being able to fill more than one delay slot is small (70% for the first slot, only 25% for the second) and so the approach is of limited benefit for deeply pipelined processors. Also it is difficult, once the number of slots is defined, to change this as the architecture develops since existing binaries are unlikely to function correctly. The unfilled slots also waste power, though if a specific NOP is provided, as opposed to using a harmless operation such as MOV R0,R0 this might be kept to a minimum.

Of course it is also not possible to add delayed branches to existing architectures without altering the behaviour of existing software. Variations of this, allowing variable sized slots are considered later.

Delayed Branches with Squashing

Normally instructions in delay slots are always executed, irrespective of the outcome of the branch. If there is no useful code to fill them NOP's must be used. This has a code space penalty that must always be allowed for, and for large delay slots this is significant. For a 20% branch density with 70% of branches taken, which is not untypical, and if the delay slot filling rate given in [2] is used (tabulated in table 3.3) 14% of fetched instructions are delay slot NOP's.

first slot filled	second slot filled	success rate
no	no	30%
yes	no	55%
yes	yes	25%
average		1.05 slots filled

Table 3.3 : Branch delay slot filling success rates

A study by Su and Despain [14] has shown that a branch delay slot of 3, which might be typical for a deeply pipelined processor, can on average be filled with only 0.88 instructions; this is even worse than the figures calculated above. The conclusion here must be that simple delay slots should be no more than two or three instructions deep, unless a way of varying the slot size can be found. One possibility is to fill the delay slots with the most likely branch target, and then allow the hardware to squash these instructions if the static prediction was incorrect. Two bits are required in the instruction; the first indicates the predicted direction, the second controls whether squashing will be used.

If the delay slot can be filled with instructions **before** the branch the squashing bit is not set. However, if set, the predicted direction bit for the branch is used to indicate from which path the delay slot instructions are taken. For a branch predicted taken the delay slot is filled with instructions from the branch target, and if the branch then turns out not to be taken these instructions are squashed. For a branch predicted not-taken the delay slot is 'filled' with the sequential instructions following the branch. These are then squashed if the branch is taken. The ability to fill the delay slot instruction with instructions from either before or after the branch is that the former will always be executed, where as the latter are speculative and therefore may need to be squashed, which wastes energy.

The results quoted by McFarling and Hennessy [2] show that this saves nearly half a cycle per branch over a standard delayed branch scheme. More detailed figures obtained by Su and Despain [14] using profiling to improve the branch prediction gives the number of cycles per branch (consisting of the branch plus the number of unused delay slots) as 1.42. This compares with 3.13 for a simple delayed branch scheme (admittedly having 3 delay slots, which is quite large).

3.5.5 Branch Target Cache

A *Branch Target Cache* (BTC), sometimes also called a *Branch Target Buffer* (BTB) or *Branch History Table*, is a small associative memory (a CAM - **C**ontent **A**ddressable **M**emory) which stores the address of a branch instruction as a tag. The data field contains a number of bits of information to allow prediction of the direction of the branch. The target address is also often included, to be used in the event of it being predicted taken and possibly one or more of the target instructions as well. The latter however is not common, but allows one or more of the target instructions to be fed direct to the pipeline, hiding the initial latency of the memory. A branch target cache is shown in figure 3.5.

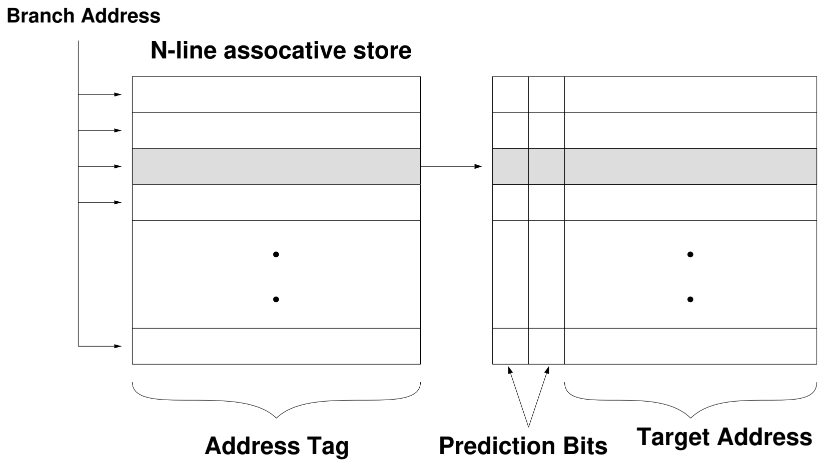

Figure 3.5 : Branch target cache structure

The initial design for a BTC was created for the MU5 computer system developed at Manchester University [5]. The system was referred to as a *Jump Trace* and implements a pure BTC with eight entries. Every address issued to the main store was compared with the jump trace, and a 'hit' resulted in the subsequent address being issued from the jump trace entry. The need for an intelligent branch mechanism was identified early on in the design of MU5, where the very high main memory latency was recognised as a performance bottleneck if jumps occurred too frequently. To simplify the design, only non-variant-target branches were entered in the jump trace. This removed the need to check whether the predicted target was indeed correct; only the prediction itself needed to be verified when the branch was executed.

It is interesting to compare the measured performance figures given by Holgate and Ibbett for MU5 [5] with current BTC implementations. The proportion of branches in the instruction stream (the *branch density*) varied from 12% to 19% (average 16%) for a mixture of ALGOL and FORTRAN programs. The measured prediction accuracy is given below:

```
ALGOL
  execution 67%
  compilation 42%
FORTRAN
  execution 65%
  compilation 46%
```
AVERAGE = 55%

These results seem quite poor, with current schemes claiming accuracy of ≈ 80%, but a number of important characteristics of MU5 must be considered:

- Branches were taken on average 77% of the time.
- The cost of a taken branch was about 19 pipeline periods, compared to an un-taken branch of one period.

These means that the cycles per branch was reduced from 15 to 10.9, which reduced the overall CPI by 20%, which was a significant improvement for a comparatively small amount of hardware.

Another early BTC implementation was present on the AM29000 RISC processor [42]. The branch target cache recorded the first four instructions fetched after a branch. The cache structure was arranged as two 16 entry associative sets with a random replacement policy. The next time the branch instruction was executed the first four target instructions were supplied from the cache, to hide the main memory latency. The effect on the instruction fetching is shown below in figure 3.6. The instruction in cycle 3 causes a branch. The BTC then provides the target instructions during cycles 4-7 from the BTC, hiding the memory latency, until finally the main memory responds in cycle 8 with the fifth instruction from the target path.

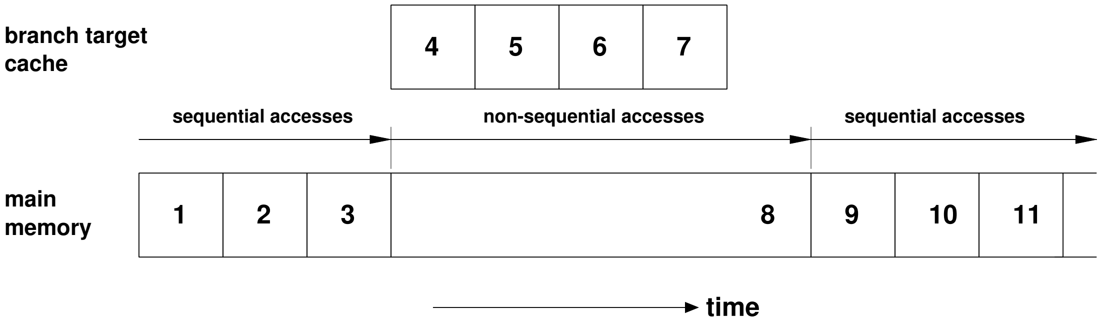

Figure 3.6 : AM29000 branch target cache operation

Current BTC Implementations

Branch Target Buffers have been widely adopted by commercial processors such as the Pentium [21] since they allow branch prediction to be added to an existing architecture without too much effort. The overhead of a BTC in silicon area is higher than a ‘normal’ cache however since a larger associative store is required. The Pentium implements a 256 entry BTC divided into 4 sets. This occupies approximately 7mm^2 of silicon, and is comparable in capacity to a 2K-bits of data cache. A ‘normal’ cache has a line length of greater than one word per tag entry. This is of no use for a branch cache since it is rarely possible to associate two or more adjacent branches with one address tag. This makes the CAM proportionally much larger than the data block compared to a conventional cache.

3.5.6 A Shared Pipeline

The normal use of a Pipeline with N stages is to run a single program with N instructions executing concurrently. Alternatively N programs could be executed in parallel, each occupying one stage of the pipeline. The processes are effectively interleaved at the instruction level. This has its problems, for example the need to have the working set of N processes available at one time, and that if there are not N processes available a certain proportion of the pipeline throughput is wasted. Also if one process stalls on a resource the others are likely to have to stall as well. The Denelcon HEP [37] implements this scheme, breaking a program down into the required number of independent processes.

It is possible to use a normal pipelined processor in this manner, by interleaving instructions from a number of tasks. This has the effect of separating the generation and reuse of results, reducing the number of pipeline interlocks and stalls. In practice however this has proved to be very difficult to achieve, probably reflecting the general difficulty experienced by superscalar processors of extracting enough instruction level parallelism to keep multiple pipelines busy.

3.5.7 Branch Removal

Since branch instructions are really instructions to the fetch unit and not to the execute unit they could be removed before they reach the execution unit. If branches are conditional the branch unit needs to verify that the correct instructions have in fact been sent to the execution pipeline, and recovery would have to take place if not. The CRISP processor [3] implements what is called *branch folding*. This makes use of a decoded instruction cache which stores both the address of each instruction plus its target on every line. This effectively turns all instructions into branches. To make use of this, as instructions are inserted into the cache it is noted where a ‘normal’ instruction is followed by a branch, and *folds* the branch into the previous instruction. Instructions are of variable width, consisting of one, three or five parcels of 16-bits each. Only certain branches are folded (one and three parcel non-branching instructions followed by one parcel branches) but these are very frequent (no figures are given) and allow most branches to be folded.

The removal of branches earlier in the pipeline using a branch unit should reduce the energy required to execute the branch since the instruction passes through fewer stages of the processor and additionally the execution unit may be simpler. This will be offset however by the overhead of recognising instruction pairs that may be combined; this is likely to add extra pipeline stages. In addition the branch unit must monitor the execution unit to verify that the direction of conditional branches was correctly evaluated. If the prediction turns out to be wrong the recovery may also be more expensive than allowing the execution unit to handle the branch.

3.5.8 Taken/Not Taken Bits

It has been shown by Ball and Larus [1] that at compile-time it is possible to establish with a high degree of accuracy the likely direction a branch will take. If this information can be encoded into the instruction set a processor can make early decisions about which of the two possible paths following the branch to follow. This still suffers from many of the problems of multiple instruction streams, for example branch target calculation delays, but requires less extra hardware to implement.

The encoding of the static information can take several forms. Normally for every branch instruction a flag, if set, indicates that the branch should be predicted taken. The CRISP implements single bit static prediction and the average prediction results for a number of benchmarks (Troff, C compiler etc) measured by Ditzel and McLellan [3] are given below :

Static branch Prediction	1 bit of dynamic prediction	2 bits of dynamic prediction	3 bits of dynamic prediction	Number of branches executed
85.4%	81.0%	85%	84.4%	63 Million

Table 3.4 : CRISP static and dynamic performance analysis

This shows that static prediction does indeed give very good prediction rates, without the large storage required for dynamic prediction. It also reflects the general RISC philosophy of passing work from the processor to the compiler.

3.6 Recent Branch Prediction Schemes

As processors become super-pipelined (a pipeline depth usually greater than eight) and superscalar (more than one instruction can be issued in parallel) the demands on branch prediction increase. Processors with an issue rate of greater than four instructions per cycle are likely to appear in the next few years, and with an average basic block size of around five instructions this implies approximately one branch per cycle. If the performance is not to be crippled by mispredicted branches, prediction rates much higher than the 70-80% currently achieved will be necessary.

The following sections look at some of the current schemes for branch prediction plus some of the author's own suggestions.

3.6.1 Prediction With Masked Squashing

As discussed in section 3.5.4 delayed branches only offer a small performance improvement due to the inability of compilers to fill more than one delay slot efficiently. In a super-pipelined processor the delay size may have to be greater than three instructions and so an extension of branch squashing has been suggested by Su and Despain, called *Branch With Masked Squashing* (BWMS) [14]. This allows part of the delay slot to be conditionally annulled depending on the outcome of the branch. The delay slot is first filled with 'safe' instructions from before the branch. The remainder of the slots are then filled with 'unsafe' instructions from the predicted branch target, and marked as such using a specific bit in the instruction encoding. Alternatively these bits are held in the branch instruction, with one bit per delay slot required.

BWMS shows 15-20% higher performance than branch with squashing, and 67% better than a simple delayed branch (with three slots). Further performance improvements were shown when profiling was used to improve the prediction information, though this tended to increase the static code size, due to more instructions being predicted taken and therefore requiring target instructions to be copied into the delay slot [14].

3.6.2 Procedure Call Stack

Procedure calls and returns form a significant proportion of current CTI's (10-30% of branches in the ARM traces) and their prediction is becoming even more important as object oriented programming styles gain in popularity [18]. This is due to the much greater reliance on procedures which results from the object oriented paradigm. The behaviour of procedure call instructions is similar to conventional branches, and in

ARM code are often unconditional, making their prediction easier. The accompanying procedure return is much more difficult to predict with current schemes however since its target will vary depending on the part of the program from where it was called.

A good solution to this is to implement a procedure call stack which maintains the return address of the last n calls. This has been shown to work very effectively in cases where the call and return mechanism is well defined. In more 'flexible' instruction sets such as ARM code there are a number of different ways of returning from a function call, and the one chosen will depend on the context. The prediction mechanism must be able to identify a return 100% of the time or the stack will become out of step, resulting in a 100% misprediction rate for the remaining stack entries.

Studies of the effectiveness of subroutine return stacks by Kaeli and Emma [18] have shown that adding a stack of around 10 entries combined with a BTC of 128 entries gives better prediction rates (33% improvement) than a single BTC of 4096 entries.

3.6.3 Branch Correlation

It has been variously reported that the direction a branch takes is dependent not only on the past history of the branch but also the path that was taken to reach it [10, 40]. To exploit this various schemes have been suggested to maintain the outcome of the last n branches in a shift register (BHR - **B**ranch **H**istory **R**egister). The contents of this register can then be used to select between a number of different prediction tables. There are a number of ways of implementing this mapping. The choice of which arrangement to use is not clear, since it is heavily dependent on the programming task and therefore the branch behaviour. If a low cost scheme is required the following mechanism (figure 3.7) has been shown to perform well.

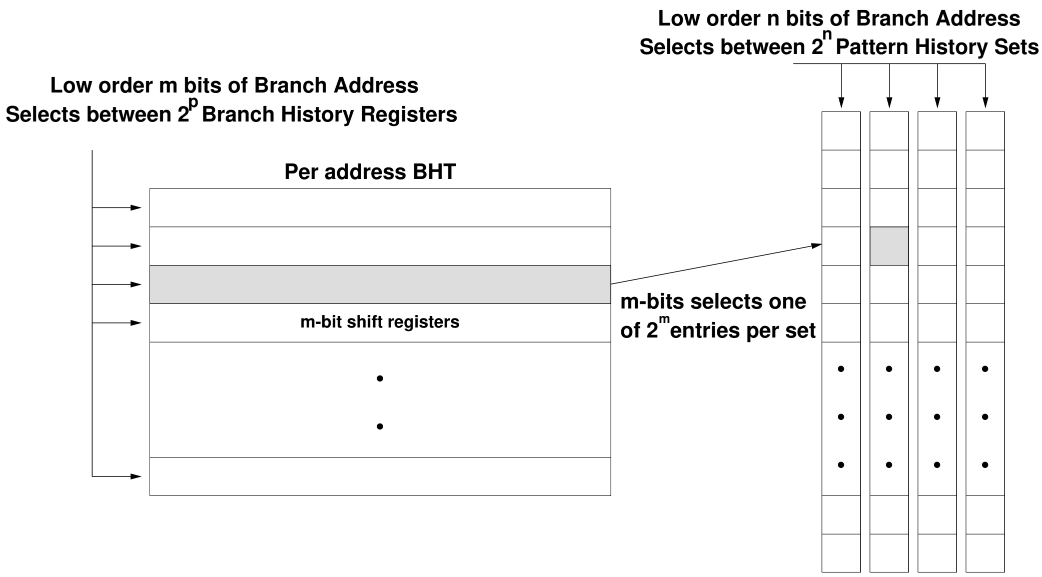

Figure 3.7 : Branch correlation implementation

The entry in the Pattern History Table consists of a number of pattern history bits. The prediction is some function of these bits, and the final outcome of the branch then causes the two tables to be updated.

The results given by Yeh and Patt [11] report a prediction accuracy above 90% for all the benchmarks. In fact for 2K bits of 'storage' they quote over 96%! These schemes are clearly very promising and more research needs to be done to see how architecture- and compiler-specific the results are.

3.6.4 Hardware Loop Support

When a program written in a high level is compiled to native machine code a great deal of semantic information is lost. This suggests that explicit support for some higher level constructs, such as loops, might be worthwhile. Dedicated processor hardware would then be able to take over the maintenance of the loop variable and would control the number of iterations of the loop body. This is only likely to be of benefit for deterministic loops without early termination since the hardware would need to identify the end condition, requiring that the instruction fetch unit have access to the register bank or condition codes. If only part of the control was passed to a branch unit it might be more flexible. In this case the loop control continually provides the loop body instructions until the execute unit jumps out of the loop. In this case there is only likely to be a branch penalty when the loop terminates, due to a number of partially fetched and executed instructions from the loop body being abandoned.

An instruction is often provided in microcontrollers called *Decrement and branch*. This takes a register value, subtracts one from it and branches if the result is not zero, combining two or three instructions into one with the register forming the loop counter. If, instead of using a general purpose register, specific 'loop count' registers are provided, the branch unit can take over the maintenance of the loop. When the generated code for a C compiler was examined however, a significant proportion of loop code was found to either adjust the loop counter within the block, or conditionally jump past the loop-closing branch to provide early termination. In these cases hardware loop support can speed up the normal case where the loop repeats, but specifying the termination case may be difficult.

By allowing a variable branch delay slot to be specified within the branch instruction more efficient loop execution is possible, since the entire branch body is then specified within the slot size. This then allows a processor with a real slot size less than or equal to the required size to execute the loop without delays. In addition, if the branch target address can be taken outside the loop body (since it will be invariant from one loop iteration to the next) the code density will increase. The resulting code might look like this;

```
  BTARGET start   ; specify the subsequent branch targets
start:
    ...           ; loop body instruction 1
    ...           ; loop body instruction 2
  LOOPEND 2,R1    ; jump back, using R1 as loop counter..
                  ; ..and with a slot size of two
```

This is likely to be compact to specify, and reduces the proportion of branch code in the loop. This may be of fairly limited use however, since for nested loops or other conditional statements the loop body size will vary, restricting the possible size of delay slot that could be specified. It does however allow a degree of implementation independence since it is up to the processor to manage the actual execution and annulling of the delay slot, allowing it to vary in size across different versions.

If hardware support can be added and used efficiently there should be useful reductions in energy required to execute the loop body, since there is more information available to the processor to optimise its execution. This allows more of the work to be done by the instruction fetch and branch units, with the execution pipe only seeing the useful data-processing operations required.

3.6.5 Storing Prediction Flags in the Cache

A form of dynamic prediction involves adding one or more bits to each entry in the instruction cache to indicate the predicted direction. These are similar to other dynamic schemes, but have the advantage that they are available slightly earlier than normal and to the decode unit look more like static prediction bits. This allows the prediction unit to update the prediction in the cache, removing the need for a separate history table. If the instruction set defines a static prediction encoding the possibility exists of dynamically changing the instruction in the cache should the prediction turn out to be incorrect. This is a form of self-modifying code, but with no dangerous side effects since if the branch is removed from the cache all that is lost is the updated prediction flag.

3.7 Summary

Generally, for any branch type, the prediction mechanism selected depends on two factors :-

1. How well the branch direction can be predicted.
2. How far ahead of the control point the condition can be evaluated.

Static prediction seems to offer real advantages for new architectures since modern compilers and profiles are able to make reliable estimates of the likely behaviour of individual branches. Dynamic schemes allow prediction to be added to existing architectures, giving reasonable performance improvements without having to make instruction set changes. Combining the two schemes allows the compiler to give good suggestions to the prediction unit while the dynamically gathered statistics can allow the former to be enhanced or corrected. For example, some versions of the DEC Alpha implement this mechanism.

Static prediction can only be designed into a new architecture, since it is usually not possible to retro-fit it to an existing instruction set, unless there are spaces in the branch instruction encoding that can be redefined as prediction bits. This is normally not the case. As a result, when designing a new instruction set it would seem sensible to allow for static prediction bits, even if in early versions of the compiler and/or processor they are ignored for design simplicity.

For architectures that are not suited to static prediction the use of dynamic schemes can be considered. The problem here is that architectures for low power design are being considered; a feature of many dynamic schemes is a history table which is updated and compared for some or all of the instructions issued or received from memory, and this uses significant amounts of energy. For example if a BTC is to be used as a power-saving feature the unavoidable cost of comparing all issued addresses with the BTC tags must be low, or the power saved in the external memory and the processor will be wasted in the BTC. Schemes that operate on the incoming instruction stream may be more power-efficient since the history tables are only examined for instruction recognised as branches. Nevertheless there will be a restriction on the size of these dynamic structures, such as the *pattern history table* used in branch correlation schemes, if reasonable power savings are to be obtained.

To examine the effect on power consumption of the branch prediction schemes considered here the AMULET2 asynchronous microprocessor has been chosen as a test bed on which to implement a prediction unit. The next chapter looks at the architectural design of this in detail; a number of prediction schemes reasonably suited to both asynchronous implementation and the AMULET architecture are compared, and analysing the possible performance improvements and power savings that might be obtained over the existing (non-branch-predicted) architecture are evaluated.

4. Requirements and Design of a Branch Predictor

The previous chapter examined in detail at some of the many branch prediction schemes that have been suggested and implemented. This chapter discusses the architectural design of a branch prediction scheme for the AMULET2 microprocessor.

The micropipelined AMULET architecture is described, the branch prediction mechanisms are considered and the reasons why a Branch Target Buffer has been adopted are given. Finally statistics are given which were used to select the optimum configuration.

4.1 AMULET1 Architecture

To investigate the possible advantages of asynchronous design for building low power systems the ARM architecture [35] has been reimplemented in a micropipelined [38] asynchronous style. The first version of this (AMULET1) has been fabricated and shown to be fully functional. A second version (AMULET2) is currently being designed, based on the experiences gained with the design of AMULET1 and evaluation of the silicon.

AMULET1 has a number of novel features, including :-

- A register bank which maintains coherent register operation while allowing concurrent read and write access with arbitrary timing and dependencies.
- An ALU whose speed of operation depends on the data being processed.
- An instruction prefetch unit which has non-deterministic but bounded prefetch depth beyond a branch.

In addition it implements many of the complex features found on modern RISC processors, such as precise exceptions, pipelined operation and maintains instruction set compatibility.

In studies of the behaviour of AMULET1 it has been noted that the branching scheme employed (bounded but non-deterministic prefetching) performs poorly where the program has a high density of taken branches. This causes a number (on average 3) of prefetched instructions to be thrown away without being executed every time a branch is taken. Clearly this wasted fetching costs power, both in unnecessary memory accesses and in the partial execution of the instructions following the branch. The recovery time of the processor after a branch is taken is also long, since the instruction pipe plus the decode and execute stages are congested with partially executed instructions that must be invalidated. This is time consuming and degrades the performance of the processor.

4.2 AMULET2 Structure

There are several issues to be addressed when considering the design of a branch prediction unit. Firstly, how a branch is detected, how the new target is evaluated and when and how this new target is fetched. Secondly, the instructions that are being executed past a branch must be verified for 'correctness' to ensure that the correct path is indeed being fetched and executed. If the prediction is found to be incorrect the processor must recover cleanly, restarting instruction fetching at the correct point. The following sections describe the operation of the major functional blocks of AMULET2, since their current design and operation will determine the design practicalities.

4.2.1 Address Interface

The instruction fetching mechanism is now presented, since this will require changes to allow branch targets to be automatically followed. Figure 4.1 shows the current *address interface*, which handles the issuing of requests, both instruction and data, to the memory system.

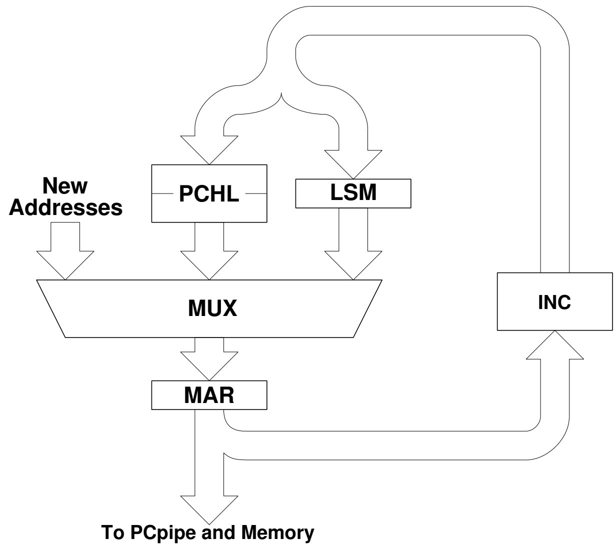

Figure 4.1 : AMULET2 address interface

The *address interface* is designed to issue sequential program counter addresses (PCs) autonomously unless interrupted by a request from the *execute unit* for a new PC or a data access. PC's normally circulate around the MAR (**M**emory **A**ddress **R**egister), incrementer and the PCHL (**PC** **H**olding **L**atch). This circulation is controlled by the arbitrating MUX which allows interruption of PC fetching when a new address should be issued. For a single data access the address is issued and PC fetching then continues. For an LSM (**L**oad or **S**tore **M**ultiple register) instruction the new address circulates around the MAR-INC-LSM until the required number of addresses have been issued. While the interface is used for data accesses the PC is held in the holding latch, ready to resume once the LSM transfer has finished.

When a branch is executed in the *execute unit* the new PC is sent to the *address interface*, where the arbiter eventually grants it access to the MAR. This address is then issued to memory whilst the old PC is thrown away, and the new PC is incremented and loaded into the PCHL. Instruction fetching now continues sequentially until it is interrupted again.

As addresses are sent out from the *address interface* the access type (Opcode or Data, sequential or non-sequential, byte or word read etc) is stored in the memory control pipe. This information then synchronises with the data returning from memory. In addition the PCs of instructions issued are saved in the PC-pipe. The PC is available as a general purpose register and therefore must be available in the register bank when the instruction fetches its operands. This structure is shown below in figure 4.2

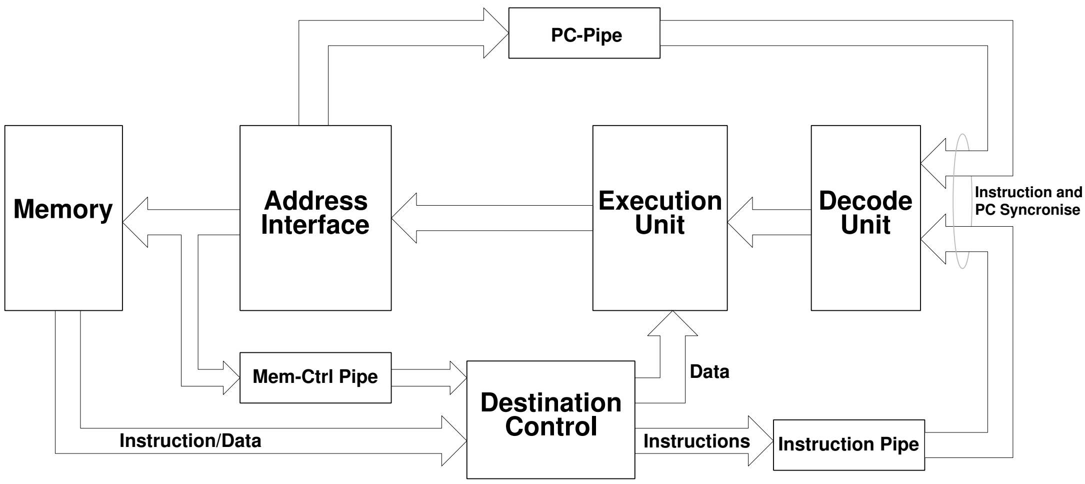

Figure 4.2 : AMULET2 instruction fetching

Because of the non-deterministic prefetching, a mechanism must be provided for identifying to which flow of control instructions belong. Each issued instruction has an associated *colour*. The colour is changed whenever the flow of control alters. This allows the instruction decode and execute stages to identify from which thread the instructions being executed belong, and to discard any that are incorrect. With the current implementation there can only ever be two instruction colours, and is represented by a single bit added to each instruction.

To illustrate how this functions imagine the current colour is *red*. The address interface issues instruction fetches, marking them as *red*. Fetched instructions enter the Instruction pipe and are eventually executed and their results written back to the register bank.

When a branch is executed the PC colour is changed in the execute unit to *blue*, and the new PC is sent to the address interface; here it eventually interrupts the sequential issuing of *red* instructions. The decode and destination control units are also told of the colour change and start to discard *red* instructions as they are seen, until they finally start to see *blue* ones. Any *red* instructions that have already passed by the decode unit will be thrown away at the end of the execute stage.

In this way only correct instructions are allowed to affect the state of the processor, even though some may have been partially executed from the incorrect stream and invalidated. Thus the operation of the processor is still deterministic.

4.2.2 Decode Pipe

As an instruction emerges from the instruction pipe it enters the *primary decode* (middle of figure 4.3). In parallel with this the *immediate field extractor* takes the instruction and separates out the immediate field if it is present. The decoding that occurs at this stage is only partial, splitting up the instruction set into classes. This is because the ARM instruction format is not as regular as some other RISC designs and a PLA to decode the instruction completely in one pass would be too slow.

The multiplexer (MUX) above the register bank selects between the *PC-pipe* and the *exception pipe*. Normally the PC for a fetched instruction is taken from the *PC-pipe* and latched in the register bank since the PC is available as an instruction operand. If a load or store instruction causes an exception however it is necessary to enter ABORT mode, by changing the current mode bits and jumping to the address of the abort handler. The PC of load and store instructions is preserved in the *exception pipe*, in case an abort occurs. The decode unit transfers the PC of the aborted instruction into the register bank for the abort handler to use. If the instruction isn't aborted the PC in the exception pipe is discarded.

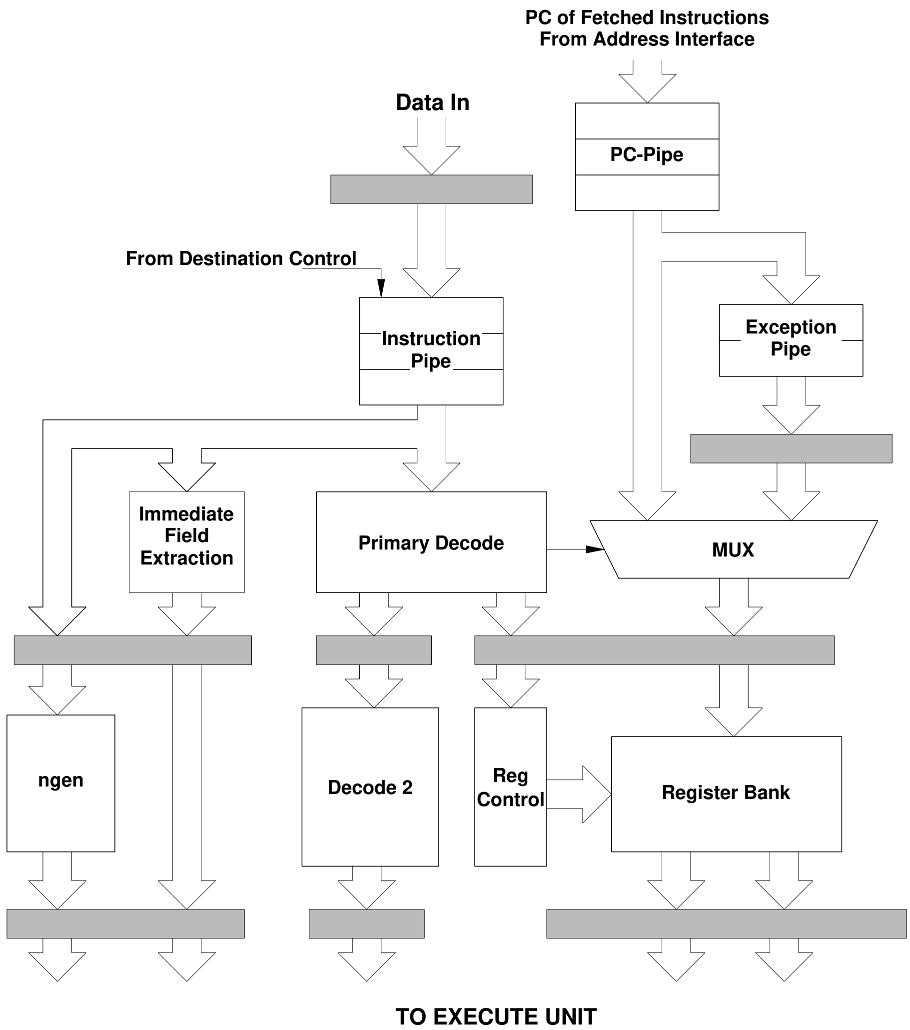

Figure 4.3 : AMULET2 decode pipeline

The *decode 2* block fully decodes the instruction, creating the signals required for the execute pipe, and at the same time accesses the register bank to fetch the register operands specified in the instruction.

The *ngen* block shown to the left of *decode 2* is used to produce any required constants needed by the execute stage; for example when executing a branch-and-link (BL) the PC must be adjusted before being saved in the link register (R14/LR). This is accomplished by adding -3 (‘plus’ carry) produced by *ngen* to the PC using the ALU.

4.2.3 Execute Pipe

The execution of the instruction occurs in the execute pipe. This consists of only one micropipeline stage and is shown below in Figure 4.4.

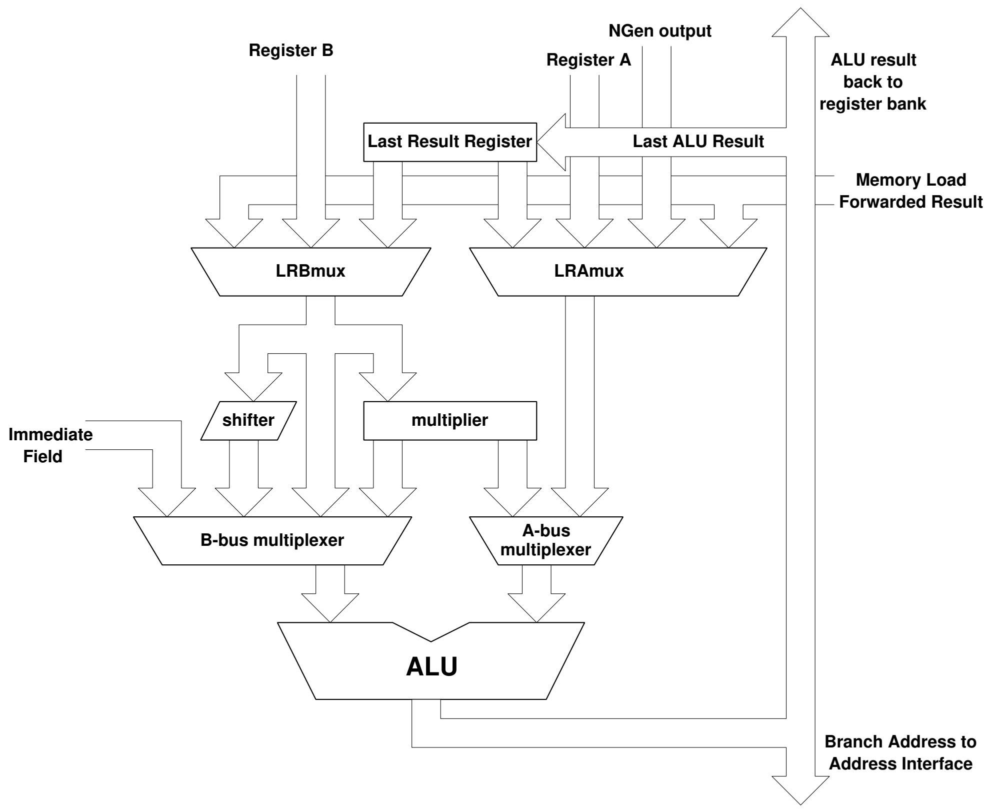

Figure 4.4 : AMULET 2 execute pipe structure

Although the *execute pipe* operates in a single stage, individual blocks within it such as the ALU and the multiplier are all self-timed, and the operation time of this stage is not fixed.

4.3 Evaluating Predictability

Studying branch prediction is not easy since there is no totally reliable model for branch behaviour that encompasses all of the many differing programming styles. Because of this any analysis must be done statistically using traces derived from 'real' program execution, and the results verified using as wide a range of software as possible.

To gather these traces an instruction-level ARM simulator (the ARMulator, [39]) was modified to produce address and opcode information for every instruction executed. The ARMulator runs 'below' ARMsd, a target-independent symbolic debugger which allows programs to be loaded and run on top of the target, in this case the ARMulator. The address traces are then processed on the fly by a prediction simulator, which can evaluate several different prediction strategies in parallel, and then produce statistics at the end of the run.

Because the ARM simulation is at the instruction level (it contains no gate-level information) the execution speed is high, enabling fairly large programs, such as compilers to be simulated in a reasonable time scale. For example it takes approximately 1 hour to simulate a C compiler compiling 20K of C-source. Much of this time is taken up compressing and storing the instruction trace file. Once the trace is generated the prediction simulator can be rerun on the trace without the ARMulator.

4.3.1 Branch Behaviour and Types

The ARM supports a number of ways to modify the PC. These are :-

- LDM (load multiple registers) with PC in the transfer list
- LDR PC, x where x is a constant
- Data operation on the PC (eg MOV PC, LR)
- Branch - B
- Branch and Link - BL (saves current PC to LR)
- SWI (SoftWare Interrupt)
- Undefined Instruction
- Interrupt

All of these can cause a modification to the PC, and a change in the flow of control which should be predicted. Unless otherwise stated, a reference to a branch refers to any PC-modifying instruction. Interrupts occur due to an external event not directly dependent on the current instruction stream, and therefore are almost totally non-deterministic.

The general prediction style simulated is to maintain history for individual branches, and to predict the direction of the branch based on its past history. No branch correlation is considered (see section 3.6.3). This is a dynamic scheme; static prediction mechanisms have also been evaluated and the results are given later in the chapter.

The first task was to discover the maximum predictability possible using the basic mechanism considered to provide a target for possible implementations. The ideal predictor maintains statistics for every branch in the program, ie it has an infinite number of entries. The degree of history to be maintained is important since, in general, the more history stored the more complex the update requirements. Lee and Smith [16] suggest that there is little benefit in maintaining more than two bits of history, ie the 'direction' the branch took the two previous times executed. In fact a single bit resulted in a very reasonable accuracy. For these studies there are advantages in restricting the number of history bits, since the history updating would require the prediction unit and the execution unit to synchronise on every branch to allow updates to occur. This may become a bottleneck since the two are likely to be operating asynchronously and therefore some form of arbitration would have to be present. A single history bit requires little or no updating and therefore was the level of information chosen to be initially evaluated.

In addition statistics were maintained not just for each instruction class listed above, but for each possible condition code (all ARM instructions are conditionally executed - see appendix C) that could be used. This facilitated studies to determine whether prediction could be guided by the branch condition code or other opcode parameters. Appendix 1 gives a sample output for run of ASim (an event-driven digital simulator).

Table 4.1 gives the initial results for the seven benchmarks. A description of the benchmarks can be found in appendix B. A branch is predicted-taken based on the direction it took when previously executed. If it was taken it is predicted-taken again. The table shows how many branches were executed, and the proportion which were taken. As can be seen around 70% of all branches are taken, and therefore without a prediction unit 30% would thus have been correctly 'predicted' . The average prediction 'limit' in this case (one history bit) is 88.9%.

Bench mark (Abbreviation)		Prediction Accuracy	Branches Taken	Branch Density	Number of Instructions
ASim (ring counter)	BM1	89.4%	66.5%	24.0%	970k
D'stone (10000 loops)	BM2	85.7%	67.0%	19.9%	3.55M
C compiler (small)	BM3	84.4%	64.8%	24.9%	192k
C compiler (large)	BM4	88.6%	66.2%	26.2%	9.53M
espresso	BM5	85.1%	66.6%	19.4%	3.48M
3d Renderer	BM6	91.9%	81.6%	11.0%	7.08M
Vi clone	BM7	97.3%	78.5%	18.0%	3.85M
Average		88.9%	70.2%	20.5%	28.7M (total)

Table 4.1 : ARMulator benchmarks, ideal prediction

These measurements show that there is a significant degree of predictability in most programs, even with the fairly simple scheme employed here. The CPI (Cycles Per Instruction) for branches gives an indication of the memory savings made by branch prediction. CPI is calculated using the assumption that a mispredicted branch takes four fetch cycles, compared to a correct case of one cycle. The calculation does not account for the recovery time of a misprediction, which will tend to increase the mispredicted cost. When no branch prediction is present the CPI will be;

$$0.702*4 + 0.298*1 = 3.11$$

When ('perfect') branch prediction is added the CPI becomes;

$$0.211*4 + 0.889*1 = 1.73$$

This is a saving of 1.38 cycles per branch. CPI is linearly proportional to the predictive accuracy.

Further studies of individual branch behaviour to evaluate whether certain branch types behave better than others was also carried out. Table 4.2 shows the percentage prediction accuracy and proportion of branches for each of the five branch types. If the number of instances of branch type, such as LDR, were very small it has not been given. Branch and 'Branch and Link' are slightly more predictable than LDM and 'data op' instructions. This is perhaps to be expected since LDMs and 'data ops' are mainly used for procedure returns, and therefore the target may vary in addition to whether it is taken or not.

Benchmark	BM1		BM2		BM3		BM4		BM5		BM6		BM7		Average	
LDM	83	8.6	100	6.0	71	9.8	81	32	74	5.7	100	3.0	92	3.9	86	10
LDR	97	1.0														
B, BL	90	82	87	66	86	81	90	80	86	92	92	91	98	91	90	83
Data op	88	7.8	78	26	83	9.2	75	6.5	82	2.1	87	5.7	94	4.6	84	8.8
SWI	99	0.2			60	0.1			97	0.1			100	0.9	89	0.2

Table 4.2 : Performance of each branch type

Tables 4.3 and 4.4 look at the predictability for *branch* and *branch and link* instructions respectively in more detail, since these form by far the highest proportion of branch instructions (83%). Each entry in the tables has seven values for the seven benchmarks used, given in the same order as table 4.1, with a breakdown of the forward and backward subsets. No overall average column is given, since for some of the values the spread across the benchmarks was wide and a single average was felt to be rather deceiving. Where averages are quoted they are simple mean values taken for the seven equivalent values for the seven benchmarks.

Benchmark	BM1		BM2		BM3		BM4		BM5		BM6		BM7	
Percent of All Branches	67.2		45.9		62.0		56.9		84.6		77.5		79.7	
Percent of Branches Taken	63.5		48.7		63.5		64.9		63.1		90.6		79.1	
Prediction Accuracy	87.9		81.2		81.9		86.1		84.6		92.5		97.3	
Taken Branches Correct	89.5		80.6		82.2		88.8		87.6		95.9		98.2	
Not Taken Branches Correct	85.2		81.8		81.4		81.1		79.6		60.4		93.9	
CPI	1.36		1.56		1.54		1.42		1.46		1.23		1.08	
Direction	**F**	**B**	**F**	**B**	**F**	**B**	**F**	**B**	**F**	**B**	**F**	**B**	**F**	**B**
Percent of Branch Type	70.4	29.6	58.3	41.7	65.7	34.3	69.7	30.3	63.7	36.3	8.3	91.7	62.2	37.8
Percent Taken	58.8	74.8	44.0	55.3	57.2	75.7	60.3	75.6	57.7	72.6	23.9	96.6	69.1	95.6
Prediction Accuracy	92.9	76.1	89.1	70.2	83.1	79.7	88.4	80.7	90.9	73.6	77.5	93.9	97.3	97.3
Taken branches correct	92.7	83.4	87.5	73.0	79.9	85.5	89.7	87.2	91.7	81.7	52.9	96.8	97.9	98.5
Not taken branches correct	93.1	54.3	90.4	66.7	87.2	61.7	86.5	60.7	89.6	52.2	85.2	9.8	96.0	69.8
CPI	1.21	1.72	1.33	1.89	1.51	1.61	1.35	1.58	1.27	1.79	1.68	1.18	1.08	1.08

Table 4.3 : Performance of Bxx instructions

This shows that simple relative branches form a significant (68%) proportion of the total, though the standard deviation is high (12.8%). The average prediction accuracy is 87%, slightly less than the average for all branches, though not significantly. More interesting is the average prediction accuracy for taken and not-taken branches, 89% and 80% respectively; this suggests that for a simple prediction scheme it is better to concentrate on predicting taken branches (or conversely more effort is required to predict not-taken ones). When forward and backward branches are considered separately (each column has two sub-columns marked F - Forward and B - Backward) the prediction accuracy for forward branches is better on average than for backward branches (88%, 82%). This is interesting since the general consensus [1] seems to be that backwards branches, mainly used for loop constructs, are easier to predict. One benchmark shows the opposite behaviour and this is the hand-coded renderer (BM6). The predominance of backward branches perhaps explains this. The text editor (BM7) shows a similar, very high predictability for both forward and backward branches. This is likely to be because of the very regular operations performed, such as screen redraw and block copying of data.

A guideline suggested by Hennessy and Patterson [30] is that 90% of backward-going branches are taken, as are 50% of forward-going branches (the 90/50 branch taken 'rule'). The results given in table 4.3 suggest that the 'rule' is closer to 75/60.

The tests for branch-and-link are made simpler since the direction of the branch is mainly dependent on the ordering of the functions within the code and the ordering of the link stage. This means there is little point in distinguishing between forward and backward branches.

Benchmark	BM1	BM2	BM3	BM4	BM5	BM6	BM7
Percent of Branches	15.0	22.1	18.9	22.7	7.4	13.8	10.9
Percent of Branches Taken	74.3	97.5	60.7	60.4	82.1	41.3	66.5
Prediction Accuracy	96.6	99.8	89.7	99.0	97.4	88.5	99.4
Taken branches correct	95.4	99.8	83.7	98.3	96.9	86.0	99.1
Not taken branches correct	99.9	99.8	98.9	100.0	99.9	90.2	100.0
CPI	1.14	1.01	1.31	1.03	1.08	1.35	1.02

Table 4.4 : Performance of BLxx instructions

Branch-and-link is normally used to call a subroutine, and although forming only 16% of branches are shown to be very predictable (96%). It is interesting that there is very little difference between the taken and not-taken predictability (unlike ‘branch’ instructions), with only the small C compiler benchmark (BM3) showing a significant difference. The percentage taken is similar to simple branches at 69.0%, indicating that simply predicting taken for branch-and-link produces mediocre results compared to a dynamic scheme. This result is slightly distorted by the renderer results. Removing this from the average gives 73% taken.

4.4 Possible Prediction Strategies

4.4.1 Branch Target Buffer

A branch target buffer (or branch target cache) as defined by Smith [16] and discussed in section 3.5.5 is an associative store which on being presented with the address of a branch instruction returns prediction information; this usually comprises the address of the branch target but can include the actual instruction or instructions found there as well. Integration of a branch target buffer into AMULET2 turns out to be quite easy and is outlined in figure 4.5. The original address interface was shown earlier in figure 4.1.

As can be seen, the BTC is placed in parallel with the address incrementer. For every PC issued to the memory system a lookup in the BTC occurs in parallel. If this ‘hits’ in the BTC, and the prediction is that the branch is taken, the multiplexer selects the target address stored in the BTC instead of the incremented PC. This is then stored in the PCHL (PC Holding Latch), ready to be issued on the next instruction fetch.

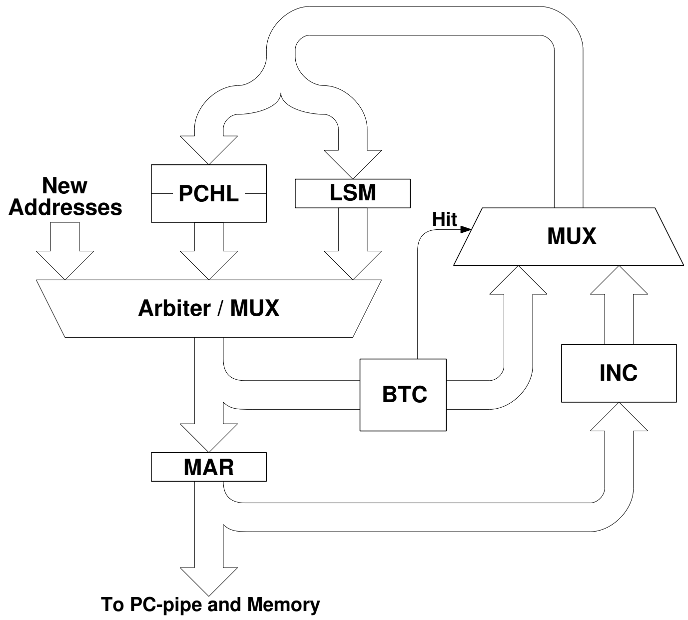

Figure 4.5 : Address interface with added BTC

If implemented in this way there is little interruption to instruction fetching, apart from the fact that the sequential flow is broken by a branch predicted-taken. This may result in the memory system supplying the branch target slower than the normal sequential case. One technique to reduce the hardware complexity might be to implement only the CAM in the BTC. If an address ‘hits’ it would then cause the fetching to stall until the branch arrives, at which time the target can be calculated (for relative branching) and the path followed. This type of scheme can, if built correctly, have no performance penalty over the full BTC; prefetching and queuing of instructions hides the latency of the branch prediction stall. This will benefit only the silicon area occupied by the BTC; the power consumed will not alter since a CAM lookup still occurs for every instruction fetch.

Another alternative to a full branch target buffer is to store only the offset in the data field, and to utilise an adder to calculate the target. This reduces the data area, since on average the offset will be small, [29] requiring around 8 bits. The size of the data store doesn’t seriously affect the power consumed however, except for the increased capacitance of busses that have to pass through the RAM, since most of the power is consumed by the CAM compare on every cycle. When the increased complexity is also considered, both for inserting and taking the branch, any area benefits become less useful.

4.4.2 Opcode-Based Prediction

A static prediction mechanism based on the opcode was investigated. This involves examining each instruction as it arrives from memory and, based on particular characteristics, predicting it either taken or not. What characteristics could be used to determine branch direction?

1. Condition code. The ARM allows all instructions to be conditionally executed. There are fourteen condition types (plus *never* and *always*), and this, in effect, means there are fourteen different conditional branch types.
2. Branch direction. A simple heuristic [49] speculates that backward branches are taken and forward ones are not. This can be easily established by looking at the sign bit in the branch offset.
3. Magnitude of the branch offset.
4. Some a combination of the above.

Condition Code

Studying the full prediction results for the benchmarks reveals that predicting branch direction based on condition code is not at all reliable with the current compiler. For example, table 4.5 shows the statistics for two condition codes from runs of a 68000 C compiler and of ASim (see appendix B for benchmark descriptions). The classes chosen are CC - Carry Clear, and HI - unsigned greater than.

This shows the proportion of branches taken for two instruction classes plus the prediction accuracy achieved for the two program runs. The number of branches executed is also given. Clearly there is a big difference between the activities of the two programs. The compiled code rarely takes BCC instructions for example, which is not true for ASim.

Program	Condition code	Prediction Accuracy	Proportion Taken	Number of Branches
68000 compiler	CC	89.1%	12.6%	6724
	HI	74.4%	90.5%	46258
ASim	CC	56.4%	62.7%	1997
	HI	69.0%	57.0%	6159

Table 4.5 : Example ARM condition code examination

There is considerable scope here for compiler optimisation work since by choosing certain instruction classes for different code structures it should be possible to give the processor clues about its likely direction. For example if loop conditions are rearranged so that most of the time BNE or BEQ is used for the loop termination test, which is likely to be true most of the time, the branch unit knows it is worth making an entry for these instruction types in its prediction table. Likewise for CASE statements, which may or may not be predictable depending on the function it is serving in the program, the choice of branch can help the branch unit to decide **not** to make an entry which is likely to be wrong much of the time, possibly throwing out a more useful entry.

Branch Direction

Prediction based on the direction of the branch is easy to evaluate, but is dependent on factors such as the strategy of the linking. The normal heuristic used is to predict backwards taken, forwards not-taken (BTFNT). If function call instructions such as branch-and-link are to be predicted using this mechanism, frequently called routines must be linked below the caller. This may prove to be impossible however since many linked modules may have conflicting link order requirements.

The results for branch direction for the benchmarks are given in table 4.6. These are for PC-relative branches, with function calls (branch & link) listed separately, due to the problems discussed above.

	Prediction Accuracy	
Benchmark	**Branch**	**Branch & Link**
ASim (ring counter)	51.2%	62.6%
D'stone (10000 loops)	55.7%	6.7%
C compiler (small)	54.1%	55.1%
C compiler (large)	50.6%	52.8%
espresso	53.3%	37.4%
3d Renderer	94.9%	35.9%
Vi clone	55.3%	49.8%
Average	59.3%	42.9%

Table 4.6 : BTFNT prediction results

As can be seen the simple static scheme performs badly, yielding results little better than a random choice. In this case predicting always-taken (see table 4.4, percentage taken) would give better though more variable results (approximately 70% correct). The only benchmark that performs well for BTFNT is the hand-coded image renderer, with near perfect results. The branch-and-link figures clearly show that although they are taken fairly often, the order in which the program is linked sets the direction of the branch. The likely path is therefore unrelated to the direction.

A combination of branch direction and condition code was also studied and the results were similarly inconclusive.

For simple static schemes such as these to work it is clear that extensive compiler optimisation would be needed. Otherwise the predictor might as well choose at random!

Possible Implementation

Since this scheme is based on the opcode of the branch instruction the prediction hardware must be situated somewhere between the *data in* register and the execution block. The instruction must be recognised as a branch, the predicted direction established and the branch target calculated and sent to the address interface. This will not prevent some instructions after the branch instruction from being fetched unless the *address interface*, memory system and data interface are unpipelined, and the address interface then stalls until the instruction is returned and examined to see if it is a branch. This would degrade the performance since the instruction fetching from memory would become a major bottleneck. Also, if the bandwidth of the instruction fetching is not high enough to support a stall whilst the fetch address is altered due to a branch predicted-taken, little performance would be gained over the existing scheme. There would be power savings however since speculative fetching is reduced.

It is likely that a number of instructions will arrive from the pipelined memory system after a branch is recognised and predicted. These instructions would normally be thrown away but will be required if the prediction turns out to be incorrect. The addition of a small buffer to hold these instructions until the branch direction is verified would speed the recovery for cases of branches incorrectly predicted. For a deeply pipelined system the possibility might exist of more than one outstanding branch, in which case it would have to stall, unless multiple recovery buffers are provided. This is unlikely to be the case with AMULET2 however.

To make this scheme work effectively an accurate prediction mechanism is needed. As shown earlier, schemes based on direction and condition-code behave poorly. Another possible parameter is the branch offset size; however if too many characteristics of the branch are combined to form the prediction the evaluation may prove too costly. The best solution to this would be the use of static prediction bits, but since there is no available space in the branch instruction encoding this doesn't seem possible.

A combined static-dynamic scheme using past history could improve the static characteristics. Some of the branch opcode bits could be combined with part of the address of the branch to index into a dynamic prediction table. This would allow many of the advantages of a BTC but with a lower storage overhead. Of course the BTC also has may advantages, principally its easy construction, simple integration into the existing design and low branch cost for a correct prediction. A great deal of hardware and compiler work would have to be carried out to make a static scheme work effectively.

4.4.3 Chosen Prediction Scheme

The three schemes outlined here have many variations and possible implementations. The choice is heavily influenced by the ease of integration into the AMULET2 implementation and in view of this the dynamic Branch Target Cache has been chosen. The following sections in this chapter consider the precise structural design for the BTC, and a more detailed study of the resulting performance and power consumption, based on figures obtained from the resulting silicon layout, is described in chapter 6.

4.5 BTC Structural Design

To establish the exact structure of the BTC requires experimentation. The parameters to consider are :-

- Degree of history maintained.
- Prediction policy.
- Update policy.
- Associativity of the prediction storage.
- The available silicon area, which limits the number of entries.

The initial prediction is based on past statistics gathered by the branch unit. It is normal to refer to the number of history bits maintained when comparing schemes such as this. If the prediction is based solely on the instruction type, this is generally referred to as 0 history bits, ie no statistics on past branch behaviour are maintained. This provides poor performance (around 60% prediction accuracy) compared to history-based prediction. Each extra bit of history represents an extra branch direction record for a particular branch being maintained. Systems employing between 0 and 5 bits of history have been examined by Lee and Smith [16]; more than 2 bits gave little or no improvement and one bit was often enough to give greater than 90% predictability. Using only one history bit also simplifies the history-based prediction unit. For example the two history bits need to be mapped onto a prediction algorithm that indicates the likely direction of the next branch. In the following example a T indicates that the branch was taken, an N that is was not. The simplest scheme examines the history bits directly to decide the next direction. For example:

Branch History	Next branch prediction
T,T	Taken
N,T	Taken
T,N	Taken
N,N	Not Taken

Table 4.7 : Two bit branch prediction rules

An alternative is to apply a mapping to the history bits so that a state is recorded which doesn’t directly correlate to the branch history. This means that at any point in time an entry has a state S, and depending on the outcome of the next branch moves to a new state or stays in the same state, ie a normal ‘Moore’ finite state machine. Figure 4.6 shows two example state networks:

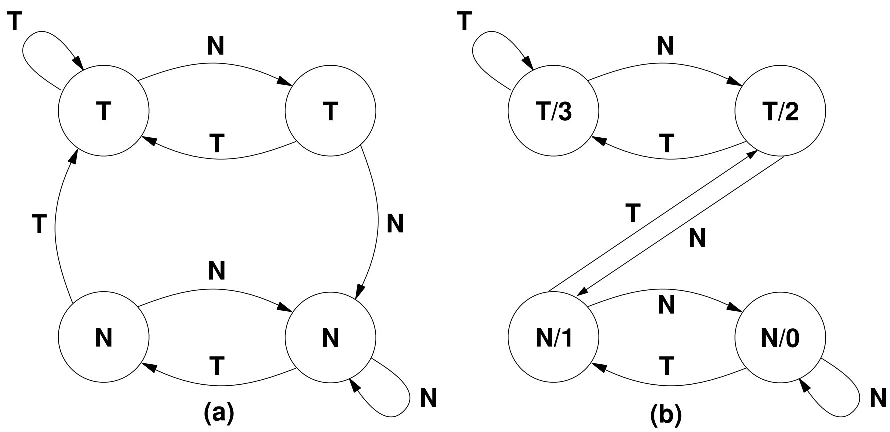

Figure 4.6 : Prediction state diagrams

Figure 4.6a, which has been used in the S1 machine from Stanford University, demonstrates one possible strategy. Each circle represents a state and the prediction that will be used when the branch is next encountered. An ‘N’ indicates predict not-taken, a ‘T’ predict taken. When the branch outcome is known the state is updated by following the appropriate ‘N’ or ‘T’ arrow to the next state. In this case two incorrect guesses are required before the prediction is changed, but also a further two to return to the previous prediction. There is an obvious pathological case (NNTTNNTT...) which will, under certain circumstances, cause every prediction to be incorrect.

An alternative mapping is the two bit saturating up/down counter, shown in figure 4.6b. The counter is incremented when the branch is taken and decremented when it is not. The branch is then predicted when the count is greater than or equal to two.

These algorithms are discussed extensively in [16] together with performance statistics for the above and other state machines. The results for these show a small improvement in prediction over a simple ‘history only’ scheme, though with little to justify the extra complexity involved in implementing them.

The above example requires two ‘history’ bits to hold the current state for each branch. If only one history bit is maintained (in practice the performance for one is only slightly less than for two) a small amount of state is saved and the updating of the history buffer is much simpler. The experiments presented here concentrate therefore on a single bit history, with modifications to try to improve performance whilst not incurring the more complex update requirements.

In the course of this research a number of algorithms used for inserting and updating entries in the BTC have been tried to evaluate the effect these have on prediction accuracy. The main feature of all strategies is the action taken when an incorrect prediction is made. The cases considered were :-

1. Predict taken incorrectly.
2. Predict taken correctly, but not to the correct location. This can only occur for instructions such as MOV PC, LR which have a dynamic target. B and BL instructions have invariant targets, providing the instruction isn’t modified.
3. Predict not-taken correctly.

For case 1 the entry could either be left in, on the assumption that it may well be taken next time (eg if this was a loop termination), or taken out. Results showed that on average leaving it in gave a small performance improvement (between 0.5 and 1%) and had the added advantage that entry invalidation/removal hardware isn't required. Also, because for a loop only the prediction of the last branch is wrong, some of the advantages of two bit history are obtained without the usual update complexity.

The possibilities for case 2 are :-

a. Leave it in.
b. Leave it in but correct the target address in the cache.
c. Take it out/invalidate the entry.

Again the best results seem to be when the entry is left as it is. Updating the target address seems to offer little performance improvement. For non-loop branches the reasoning for this is unclear and more studies are needed to understand the behaviour observed. The choice of strategy here has important implications if virtual memory is provided. Self-modifying code is often disallowed because of the problems of dealing with inconsistencies between the instruction cache and main memory. Similar effects occur however when the instruction space is re-mapped. When this occurs the instruction cache is normally flushed; this also applies to the BTC if only the outcome and not the target is verified at execute time.

For case (3) this is effectively what will happen when a branch is first encountered (if only 'taken' branches are stored in the BTC) and so all that can really be done is to insert a prediction into the BTC, on the assumption that it will be taken next time.

These tests were carried out on both an infinite and finite BTC, though for the former removing entries makes little sense, since there is a BTC entry available for every branch instruction.

Again the results confirm those given by [16], which is interesting since the traces and machine architectures differ substantially from the ARM architecture.

4.5.1 Finite BTC Sizes

For a real implementation there will be a limitation on the size of BTC that can be constructed and therefore such issues as size and configuration (direct mapped, set or fully associative) must be considered. The replacement and insertion strategies may also need to be changed.

The initial choices for simulation were as follows :-

- For reasons of complexity a full LRU algorithm is impractical (though a limited form might be possible) and so tests were carried out using a circular replacement algorithm.
- Most BTC's tend to have a high level of associativity due to their small size and therefore a fully associative tag store was used. The effects of differing degrees of associativity is considered in appendix B.
- Sizes of 8 and 16 entries were chosen since it was expected that the silicon area available would preclude a BTC much larger than this.
- The insertion and replacement algorithms used were the ones described above, ie no invalidation or updating of entries when mispredictions occur.

It is likely that different instruction classes (LDR, LDM, BRANCH etc) will have different characteristics. Table 4.8 gives the results for BTC sizes of 16 and 32 entries. Each column gives the prediction accuracy for different combinations of branches cached.

	'Branch' type			
BTC size	**All types**	**All except data ops**	**All except SWIs**	**only Bxx, BLxx**
16 entries	62.2%	58.9%	62.2%	65.6%
32 entries	75.3%	71.8%	74.9%	68.6%

Table 4.8 : BTC performance for different forms of control transfer

Initial results with all branch types cached proved to be unsuccessful, producing lower prediction rates with small BTC's (fewer than 16 entries) than with only Bxx and BLxx instructions. The poor results seem to be due to the fact that the more branch types to stored in the BTC, the less likely a particular branch entry will still be present by the time it is executed again. Circular replacement performs worse than random replacement in this case by throwing entries away just before they are needed again when the working set is larger than the number of entries. With random replacement there is at least some probability than an entry will remain in the cache until it is required again.

When the size of the BTC is increased (to 32 entries) better results were produced than for storing only B(L)xx. This is because once the cache reaches the size of the 'working set' of branches most will be predicted correctly.

The reason why switching off Data op's had such a poor effect in this case (compared to SWI's and B(L)xx) is that for ASim they are more common in the code.

It is assumed that a BTC with much more than 16 entries will not be practical in AMULET2 due to limitations on the silicon area available. The implication of this, considering the earlier results, is to store only Bxx and BLxx instruction targets in the BTC. Also, since the branch target is static, as long as self-modifying code is not allowed, the mechanism for checking the validity of the prediction is simplified.

It is worth noting at this point the effect of a different replacement policy on the size of very small BTC's. Figure 4.7 and 4.8 give a comparison of random and cyclic replacement for ASim and a C compiler.

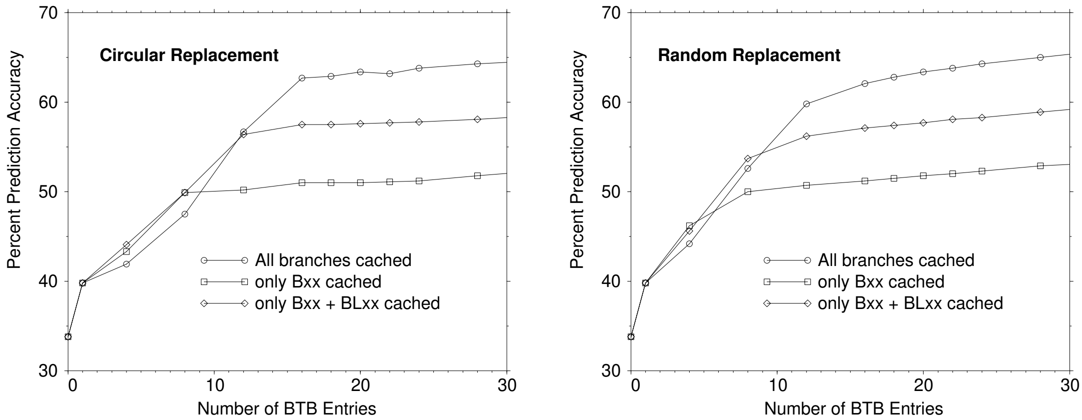

Figure 4.7 : Circular and random replacement for C compiler

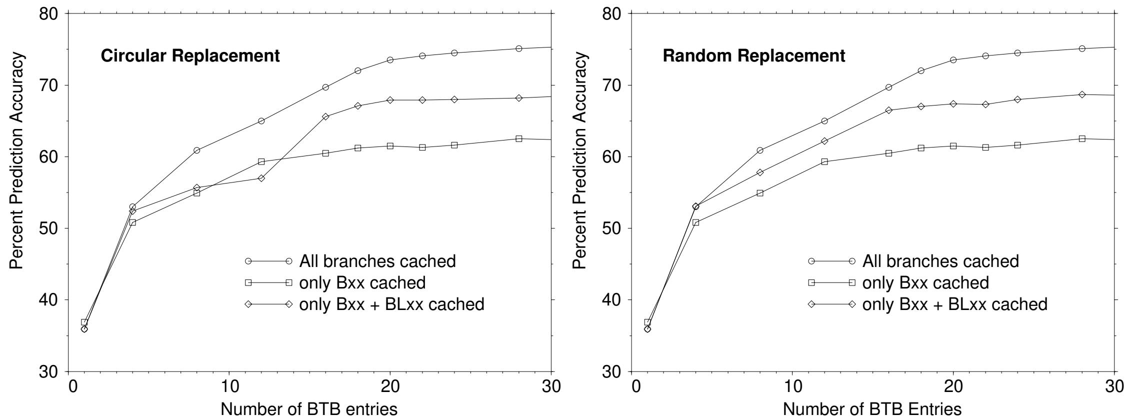

Figure 4.8 : Circular and random replacement for ASim

For both benchmarks it can be seen that a random policy has reduced the effects of thrashing for small BTC's.

A way of measuring the predictability in a program, and therefore the size of BTC required, needs to be established. One possibility is to count the number of different (taken) branches between executing a branch and executing it again. For example consider the loop closing branch of a *while* loop. If the loop contains another inner loop plus a function call there may be ten branches executed for each iteration. If the loop closing branch is to be predicted, and indeed all the other branches as well, a history buffer large enough to hold all of them at the same time is needed, in this case 11 entries (one for the closing branch and ten for the code within the loop). When considering nested loops an inner loop branch should only be counted once since if a branch 'hits' in the BTC it would be unnecessary to re-insert it.

Figures 4.9 and 4.10 show the loop distance statistics for the benchmarks. The former counts all types of branch, whereas the latter considers only Bxx (relative branch, any condition code) and BLxx (branch and link, any condition code), which are guaranteed to have invariant targets if self-modifying code is disallowed. The dotted line on all graphs shows the 16-entry point, since this is the size likely to be implemented. It also allows an easy comparison point between graphs. As can be seen the general effect of restricting the prediction to 'branches' only is to move the curves generally to the left. This means that for some programs major loops become cacheable for 16 entries (eg for ASim).

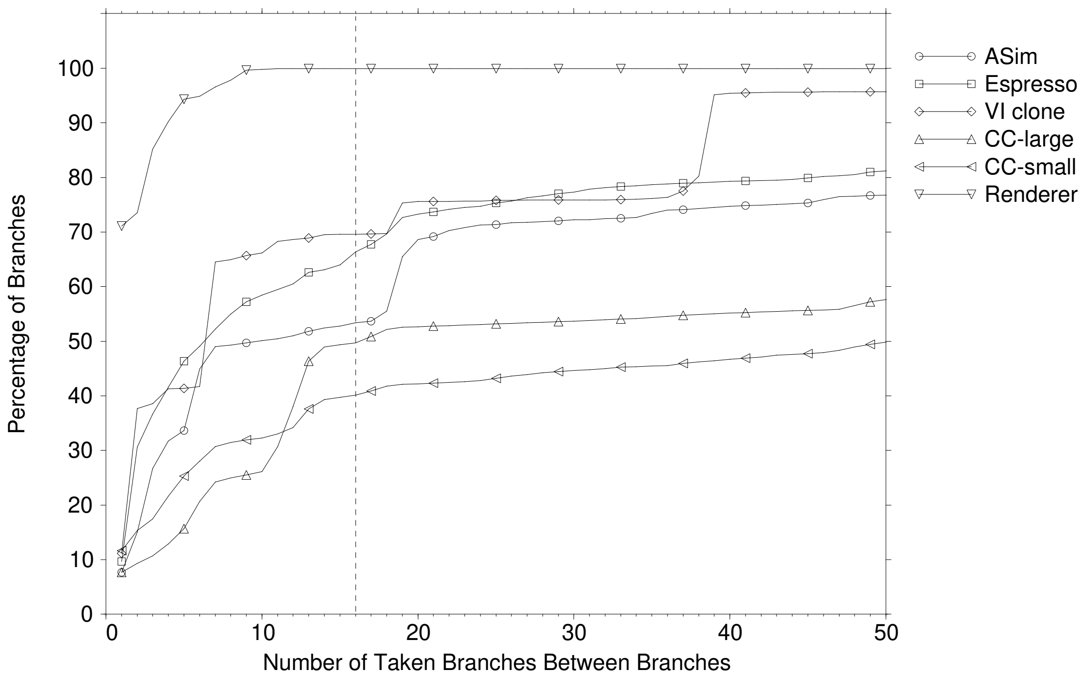

Figure 4.9 : Cumulative branch distances

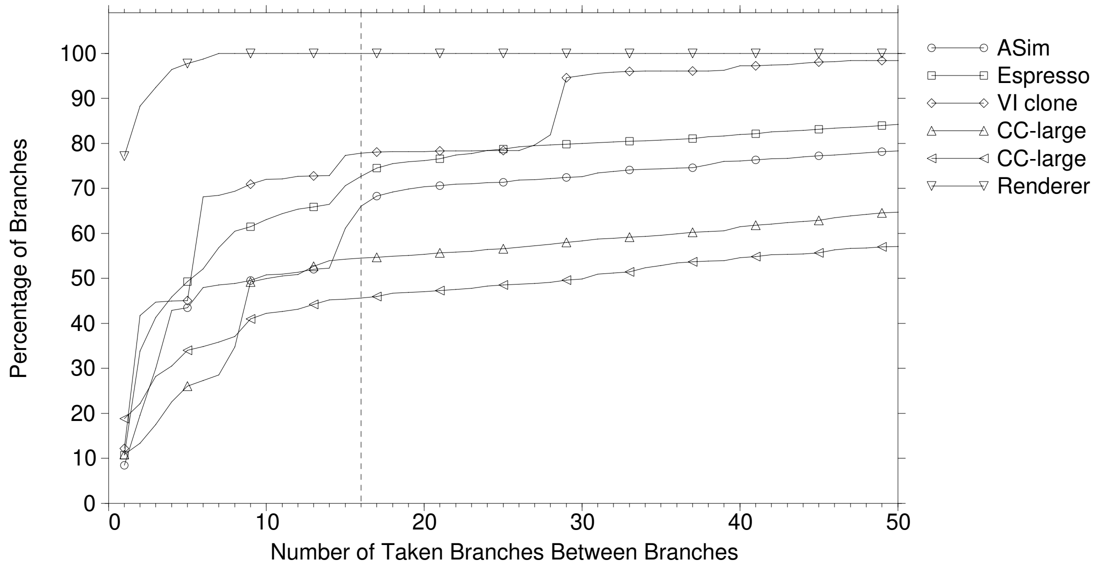

Figure 4.10 : Cumulative branch distances, B(L)xx only

Although the predictability of all branch types seems to be fairly consistent (table 4.2), the complexities of having to cope with variant targets and the slightly lower predictability suggest that restricting the studies to relative branch (Bxx) and Branch and Link (BLxx) only may be advantageous. Figure 4.11 shows the simulation results for various BTC sizes for the six benchmarks. Dhrystone has not been included since its behaviour does not seem consistent with the other benchmarks considered. A size of ‘0’ represents having no BTC, ie the AMULET1 scheme where not-taken branches are predicted correctly due to prefetching.

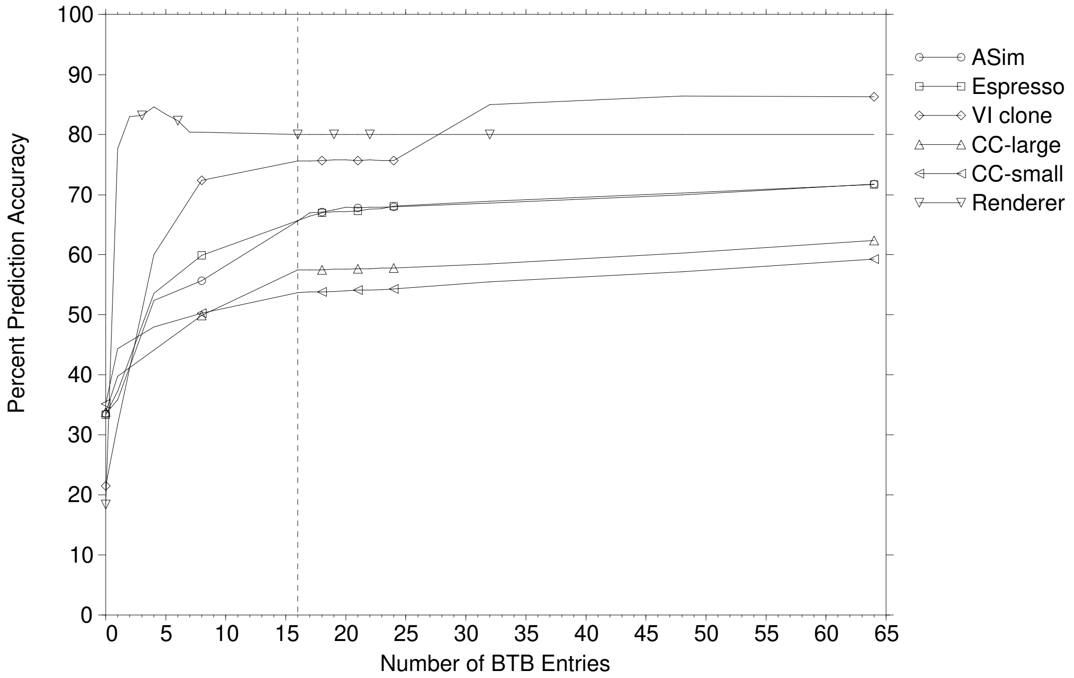

Figure 4.11 : Performance of circular replacement, B(L)xx only

It is also worth studying the effect of varying the associativity of the BTC so statistics were gathered for 16, 32 and 64 entries, varying from fully associative to direct mapped. The graphs for the six test programs are shown in appendix B, figures B.1 to B.6. Generally the performance of a fully associative BTC of size x is equivalent to a direct mapped one of size 2x. An exception to this was for the ‘vi clone’ test program, when comparing 32 and 64 entries. This may be due to the generally high results, and the fact that a text editor may spend much of its time in a small number of screen redrawing routines, whose performance will be very dependant of how the individual branch addresses map onto a direct mapped cache. This gives a more optimistic indication than the rule given by Hennessy and Patterson [30] that a direct mapped cache must have twice the number of entries than a 2-way set associative cache, which is far less associative than a fully associative cache. The studies carried out to determine the correct size of a combined data/instruction cache for the ARM2 [35] showed that a direct mapped cache four times larger than a fully associative one is necessary to obtain the same performance levels. This seems to show that a branch target cache is less sensitive to the degree of associativity than a data or instruction cache.

Another important characteristic of any cache, be it for storing normal data/instructions, or a BTC, is the replacement policy. The most common schemes are circular, random or LRU. LRU was investigated and shown to offer no significant advantage over a circular policy. This is perhaps to be expected since the behaviour of branch execution may be very much circular for the particular set that is being held. Studying a random policy (figure 4.12) together with a variation of a circular technique called GODS (**GO D**own **S**tack - the insertion point is varied to try to reduce thrashing) in figure 4.13 demonstrates no real difference when compared to the circular scheme. More significant is the size of the cache, since this directly affects the amount of history that can be stored, and therefore the number of previously taken branches that can be predicted taken again.

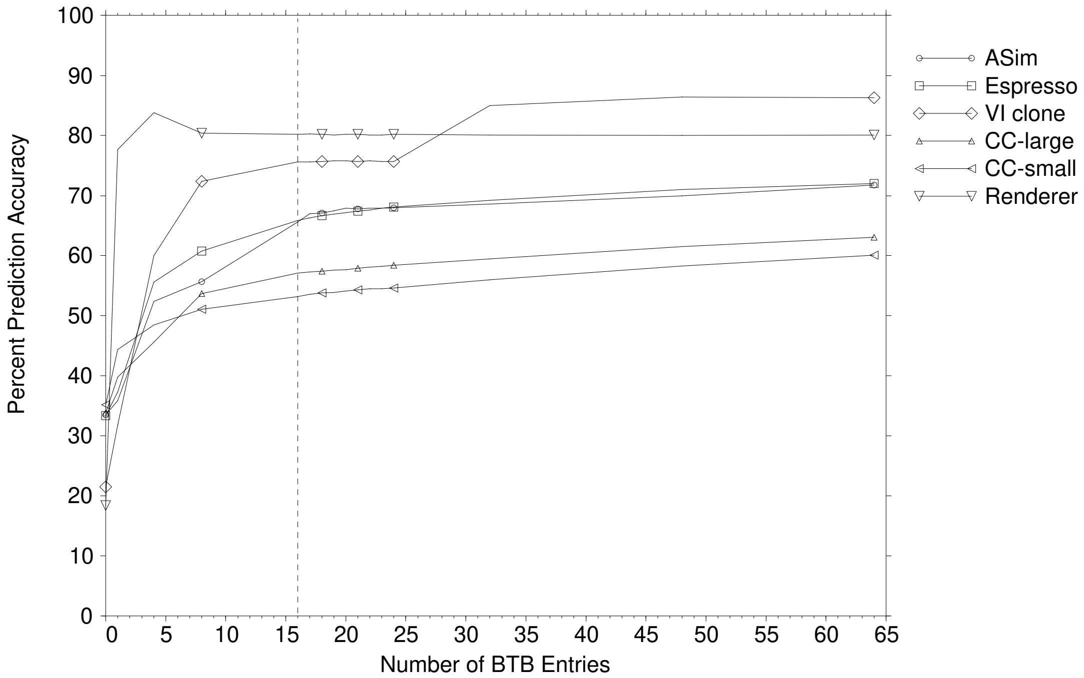

Figure 4.12 : Performance of random replacement, B(L)xx only

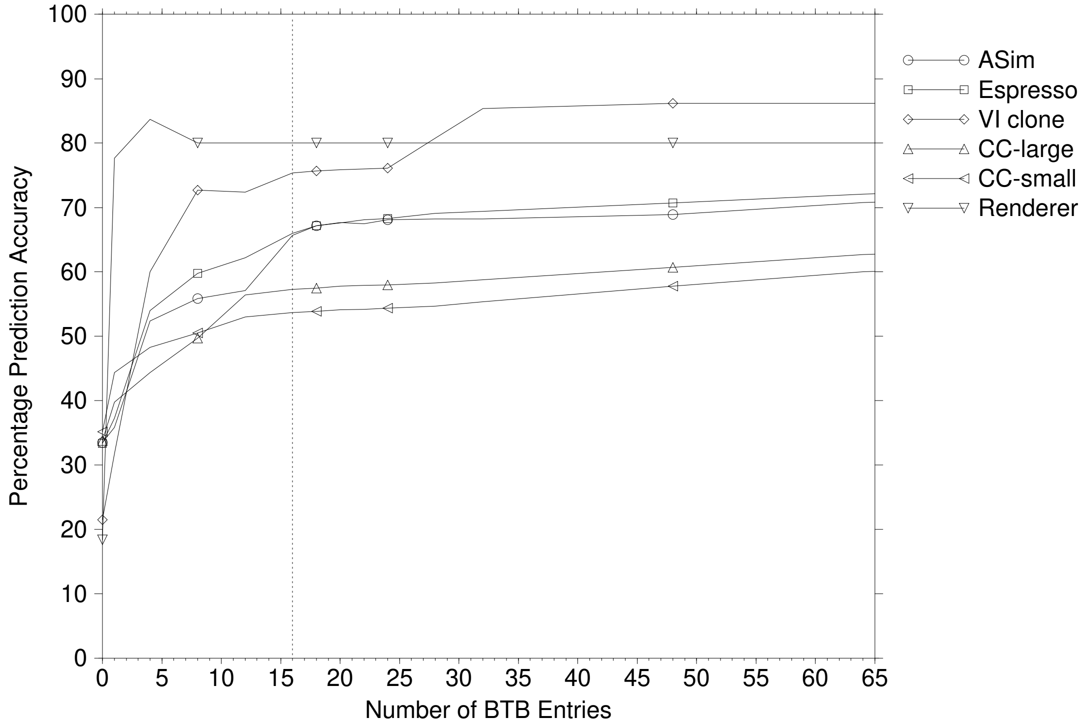

Figure 4.13 : Performance of GODS replacement, B(L)xx only

It is interesting that a random policy works so well. In fact for very small BTC's of around 8 entries it produces slightly better results than for circular replacement, presumably because of the more 'fuzzy' degradation in performance. This is particularly important where the average branch distance may be larger than the buffer size, resulting in thrashing. Also, when examining schemes with small cache sizes that attempt to cache all branch types, the performance becomes more acceptable because of the resistance to thrashing. The suggests that small, cyclically replaced caches are in general a bad idea. What 'small' is defined to be is difficult to say. It is probably safe to presume however that sixteen entries is the lower practical limit for circular replacement.

4.6 Conclusions and Summary

The final BTC design is as follows :-

- 20 fully associative entries.
- Cyclic or random replacement.
- Single history bit per branch.
- No update of entries on misprediction.
- Store only branches predicted taken (implied by the BTC mechanism).

This should allow a prediction accuracy of around 71% to be achieved, compared to around 30% with no BTC.

Many 'paper' designs concentrate on more 'intelligent' replacement algorithms, usually LRU [1,13,18,19]. This seems misguided however since the results show little or no dependence on the replacement policy, and the implementation costs of LRU are very high. This may be because of the naturally 'cyclic' nature of branches, resulting in the size of the BTC being of principal importance, not how it is used.

5. An AMULET 2 Branch Target Cache

The previous chapter describes the architectural design of a branch target cache for AMULET2. This chapter goes on to describe how this has been integrated into the current AMULET2 models and simulated to obtain performance statistics.

5.1 Asynchronous Logic

The micropipeline design style employed in AMULET was popularised by Ivan Sutherland in his Turing Award lecture of 1988 [50]. This lecture, among other things, described a library of asynchronous building blocks that can be used to construct control circuits and pipelines. The resulting circuits form logical blocks, and the communication between the blocks is implemented by a two-phase signalling convention.

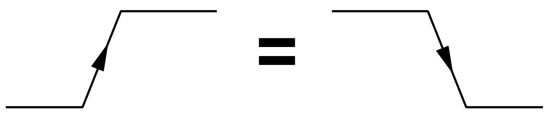

Figure 5.1 : Two phase signalling convention

The two phase convention uses a **transition** to indicate an 'event'. Most design schemes use a level sensitive protocol where the absolute voltage on a connection indicates its logical state, either active or inactive. If a signal is active it must be de-asserted before it can be re-asserted, ie two transitions occur for each event. For transition signalling there is no need for a recovery phase while a signal returns to an inactive state. This in theory is more power efficient. Figure 5.2 demonstrates how a pair of event lines can be used to form the control path between two asynchronous blocks.

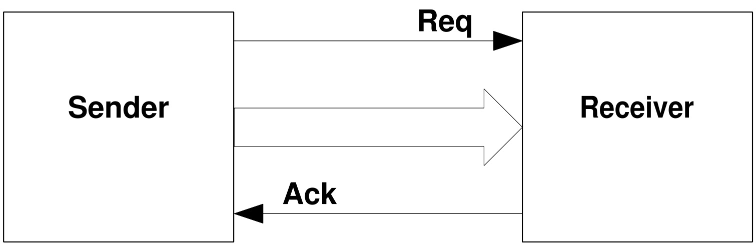

Figure 5.2 : Micropipeline handshake signals

The handshake that occurs on the Req and Ack lines is shown below in figure 5.3. Valid data provided by the sender is represented by the white areas, and the data being prepared by the grey areas.

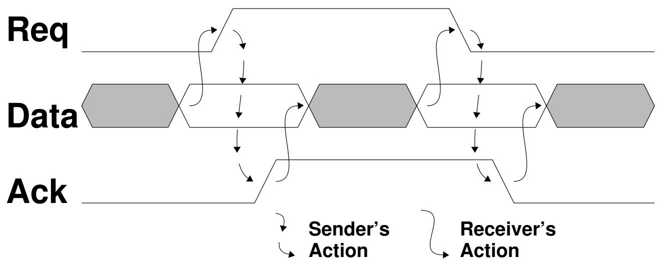

Figure 5.3 : Bundled data handshake sequence

A transition on Req and Ack brackets every data transfer. The two-phase signalling means that either a rising or a falling edge on a handshake lines indicates an event. This is conceptually quite simple, but does increase the complexity of the logic blocks used for event-based control in comparison with a more conventional level-sensitive scheme. The basic library of event blocks is shown below in figure 5.4.

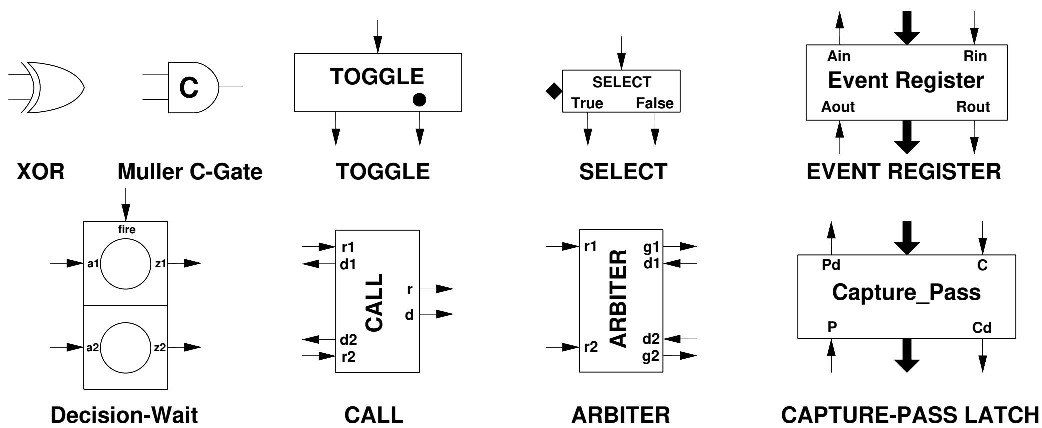

Figure 5.4 : Micropipeline building blocks

A description of the operation and function of each of the building blocks follows; when the environment is referred to it means the surrounding logic providing and using the blocks inputs and outputs.

XOR

The XOR gate serves as an ‘OR’ of events; whenever an event occurs on one of the inputs an event is generated on the output. The environment is responsible for ensuring that events on both inputs do not occur at, or near, the same time. An XOR is often used to merge two event lines into one.

Muller C-Gate

This acts as an event synchroniser. An event occurs on the output whenever an event has occurred on both inputs. It is a truly asynchronous component since there are no restrictions on the relative timing of the input events, except that when an event occurs on one input an event must occur on the other input before a new event can arrive on the first.

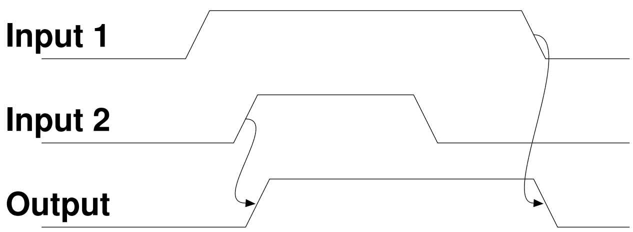

Figure 5.5 : Muller C-Gate operation

Toggle

The toggle alternates the issue of an event on one of its two outputs every time an event is received on its input. The first event is issued on the *dot* output (see figure 5.4), the next on the *blank* output, and then the cycle repeats.

Select

This block allows events to be directed to one of its two outputs depending on the state of a Boolean input, indicated by the diamond. A ‘1’ on the Boolean causes the input event to be steered to the *true* output, a ‘0’ to the *false* output. The environment is responsible for ensuring that the Boolean input is stable around the incoming event.

Decision-Wait

The ‘decision-wait’ is another event-synchronising block. An event on the *fire* input (see figure 5.4) ‘primes’ the decision wait. After this an event on either a1 or a2 is passed to z1 or z2 respectively. The relative timing of *fire* and *a1/a2* is not specified. The environment must ensure that events do not occur on both a1 **and** a2 for a single fire event. There is no restriction on the number of inputs; in practice up to four are common.

Arbiter

An arbiter is used to allow access to a shared resource by a number of blocks, all of which may make requests at arbitrary times. In the case of simultaneous requests the arbiter must make a decision as to which block will be granted the resource. In this case there are two request inputs, r1 and r2; the resource may be granted at an arbitrary time later using either g1 or g2.

Call

The call allows two or more blocks to share a common resource, similar to the use of a function within a programming environment. An event on one of the request lines causes an event on the request out; when the acknowledge is received it is then steered back to the caller. The call block requires that events are not simultaneously received on both the request lines. A call block is often combined with an arbiter to allow non-mutually exclusive blocks to share a common block.

Event Register

An event register is the basic building block of an asynchronous elastic pipeline. If the register is initially empty the environment provides data on its input and a subsequent event on Rin. This is latched and an acknowledge (Ain) is generated, indicating that the register has latched the data. At the same time an Rout event is produced to indicate to the next stage that data is now available. The provider may now present more data on the input together with an Rin event, but however will not be accepted until the next stage has latched the data currently held.

It is possible to construct elastic pipelines using event registers. A three stage pipeline is shown in figure 5.6.

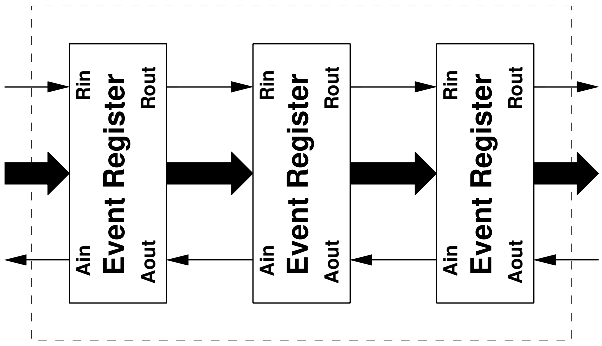

Figure 5.6 : Three element asynchronous pipeline

5.1.1 AMULET Design Style

The asynchronous model employed is a combination of the *delay-insensitive* and *bounded delay* models. Delay-insensitive logic specifies that the same result will be produced regardless of the delays through gates or wires in the system. The bounded-delay model however, requires that the delay along the wires and building blocks is known, or at least bounded within a particular range. It relies on the data being correctly bracketed by the handshake signals so that the setup and hold requirements of the asynchronous blocks are not violated. The control paths however are generally delay-insensitive. A more detailed description of the different asynchronous methodologies can be found in [25].

An example of the design style used is in the destination control block (figure 5.7). This routes the incoming data from the memory system into the required instruction or data pipeline.

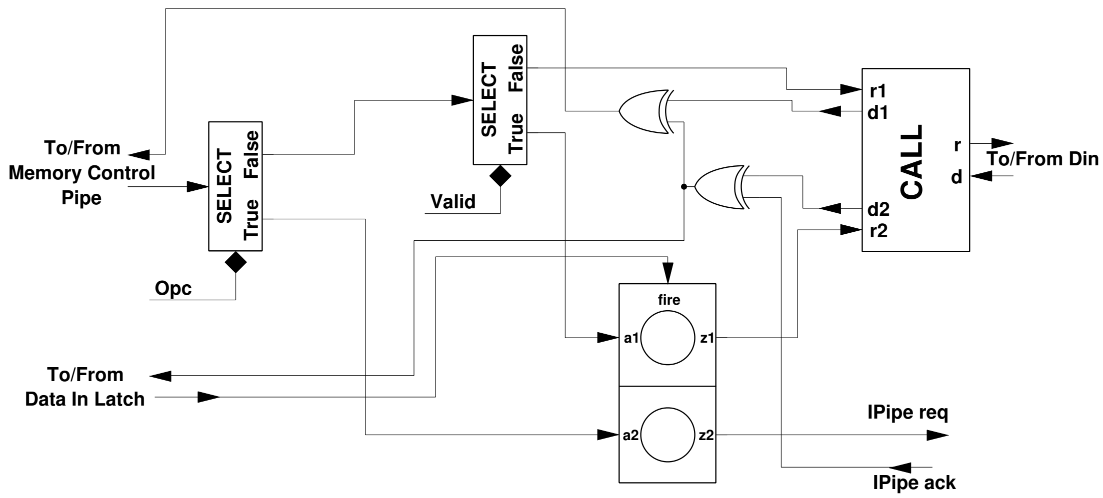

Figure 5.7 : AMULET destination control block

This block provides an example of delay-insensitive control logic, with bounded delay Boolean inputs (Valid, Opc) guaranteed valid with respect to the memory control pipe handshake signals.

When an event arrives from the memory control pipe, indicating that there is an outstanding memory request, it enters a select block. If it is an opcode fetch (Opc is true) an event is directed to the decision wait, to synchronise with the data when it arrives in the data in latch. A request is then sent to save the incoming data into the instruction pipe (*IPipe req*), which, when acknowledged (*IPipe ack*), signals to the data in and memory control pipes that the arriving data has been processed and can be removed. For a data fetch the second select block is used. The *valid* signal indicates whether data is expected to arrive from the memory system. If not the decision wait is bypassed, since there is no data that has to be dealt with, and an acknowledge is immediately sent back to the control pipe.

This demonstrates a general design style that uses select blocks to steer the incoming event along the required path, and XOR gates to merge together the events from the returning paths.

5.2 Additions to be made to AMULET2

The previous chapter presented the various parts of the processor, and each requires the addition of hardware to implement the BTC. A cache is incorporated to store the target of *branch* and *branch-and-link* instructions only. Other branch instructions, such as subroutine returns, have varying targets; if these are to be predicted also the target must be verified at the execution unit to ensure that the predicted address is correct. This was judged to be too complex to implement and could actually reduce the predictive accuracy for a small number of BTC entries (section 4.5.1)

When a branch is referred to as being ‘not-taken’, this means that an entry for it was not present in the BTC, and therefore the address interface continued to fetch sequentially. A branch predicted ‘taken’ indicates that a BTC match occurred and the PC following the branch was altered. For each case there are two possibilities; that the prediction was correct or it was not, and corrective action was required.

5.2.1 Address Interface

The structure of the modified address interface was shown in figure 4.5. Not shown are the modifications needed to the memory control pipe to store the predicted state of each instruction. The internal structure of the BTC is shown in figure 5.8. A multiplexer allows either the memory address register (for normal PC lookups) or the X-Pipe (for writing entries) to be presented to the CAM. If a match is found within the CAM the appropriate RAM select line becomes active, driving the new PC value out of the BTC. The memory address register is also available as an input to the RAM, since for writes into the BTC, this is the bus on which the branch destination arrives.

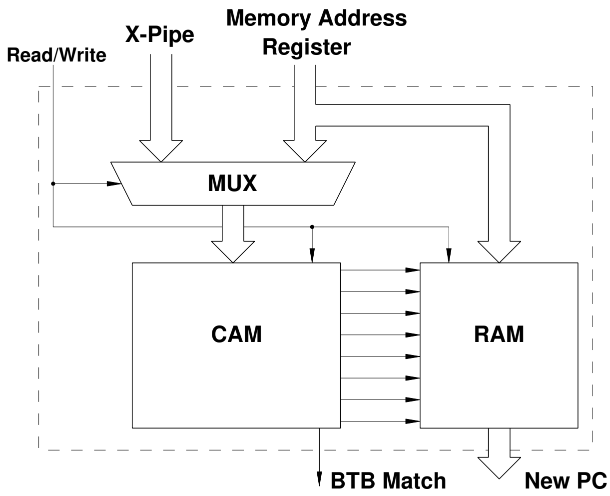

Figure 5.8 : BTC internal structure

Every memory access generated causes an entry to be made in the memory control pipe to record its type, and for branches that are predicted-taken an extra bit has been added to indicate this. If this bit is set the branch has been predicted-taken and therefore the instruction immediately following is out of sequence; if clear the branch has been 'predicted' not-taken, ie the simple sequential case.

The prediction bit accompanies a branch as it moves through the pipeline so that the execution of a branch can be modified depending on the predicted target. The execution will vary as follows :-

- Branch predicted-taken; normally the ALU operation for the branch calculates the address of the target, ie PC+offset. Instead, for a predicted branch, the operation must prepare PC+4, since if the prediction proves incorrect it must 'un-branch' to the sequential address.
- Branch predicted not-taken; effectively a normal branch but, for reasons discussed later, the instruction decode block must taken special action for a branch that has been predicted not-taken.

The need to mark the branch as predicted puts some timing constraints on the branch unit; for an issued PC, determination of whether the PC matches an entry in the BTC must be made before the memory access details can be inserted into the memory control pipe. If the matching in the BTC is too slow it will reduce the rate at which PC fetches can be issued. To correct this the memory control pipe insert may be delayed by one cycle as follows:

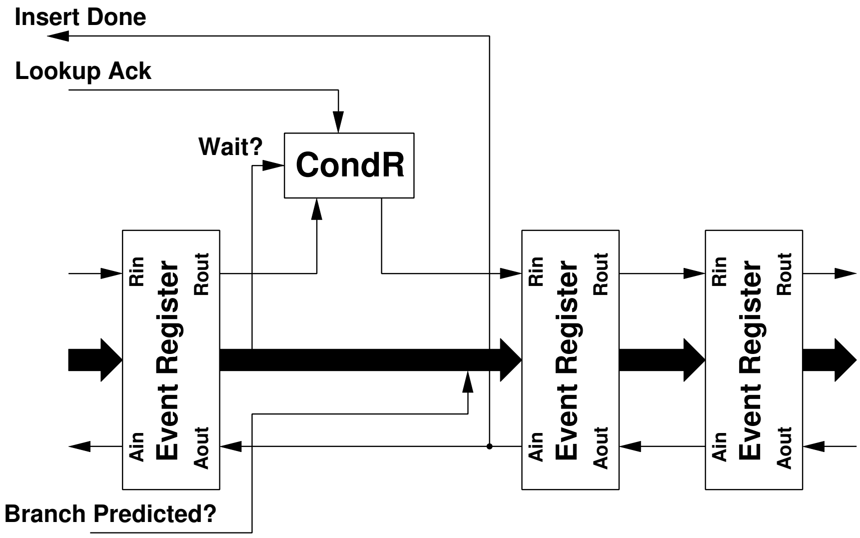

Figure 5.9 : Split three stage micropipeline

Here the three stage pipeline delays the insertion of the prediction result for one stage. The *CondR* block provides a conditional rendezvous between two event lines, controlled by a boolean input, in this case, '*Wait?*' . If this is true the Rin signal to the second stage of the pipeline is forced to synchronise with the lookup acknowledge (*Lookup Ack*). This allows the lookup in the BTC to proceed in parallel with the issue of the PC of the branch. The result (*Branch Predicted?*) then need only be ready just before the **next** PC is ready to be issued.

5.2.2 Data Interface

Data arriving from memory must be directed to either the instruction pipe or the register bank (incoming load data) depending on its type. An entry in the memory control pipe, which records the types of all outgoing memory requests, controls this process; additionally the branch prediction flag must now be considered. When an instruction arrives from memory it is stored into the *Instruction Pipe* by the *Destination Control Block*, along with a number of bits from the memory control pipe, specifically the sequential flag and the instruction ‘colour’. In addition the predicted flag is now inserted, to make it available to the decode and execute units so that the behaviour of the branch instruction can be modified.

5.2.3 Instruction Decode

As instructions emerge from the instruction pipe they enter the primary decode unit (top of figure 4.3). This block splits instructions into classes and at this point, if the branch is predicted not-taken, the PC of the instruction is stored in the exception pipe. The exception pipe’s primary purpose is to allow the PC of potentially faulting load and store instructions to be saved; should an exception occur, the PC of the faulting instruction is then made available to the exception handler. However, for branches predicted not-taken, it may be necessary to make an entry into the BTC, should the branch subsequently be taken. The exception pipe can be used for this purpose. The PC of the branch is required, since this forms the tag of the cache entry.

If an instruction requires multiple ALU cycles to complete, it resides in the primary decode until finished. The required number of operations are dispatched to the next pipeline stage to fetch the operands, which then proceed to the ALU (lower part of figure 4.3 and figure 4.4).

The ALU operation type is determined in the primary decode unit. In the case of a normal branch a signal indicates that the *immediate field extractor* must generate the branch offset from the instruction. For a branch predicted **not-taken** this is the required behaviour, but if predicted **taken** the ALU should be calculating PC+4 instead. This requires a different block (*ngen* in figure 4.3) to generate the required constant, in this case four.

5.2.4 Decode 2 and Operand Fetch

Very little needs to be altered in this stage. The adjustments to Decode 1 cause the required changes to the fetched operands to happen automatically. All that has to be added are extra pipeline bits to carry the branch prediction information to the next stage.

5.2.5 ALU Stage and Branch Recovery

At this stage the required ALU operations are carried out on the operands fetched/ generated in the previous stage (figure 4.4). The prediction information is used to adjust the ALU multiplexers to select the correct operand sources; for a predicted branch the PC is presented on the opposite bus to normal. The reason for this is that for a normal branch the branch offset arrives on the B bus from the *immediate field extractor*, but values from the constant generator (*ngen)*, arrive on the A bus (see figure 4.4).

The normal calculation for a relative branch target is PC+offset. However due to the historical effects of the three stage pipelined ARM architecture the PC is actually the PC of the branch+8. This means that the need to calculate the recovery address of a predicted branch (PC+4) actually requires the ALU to calculate (PC+8)-4. This however is the same operation as carried out by a branch and link to save the subroutine return address, and therefore generation of -4 is provided.

The above calculation is carried out in the ALU, but only if the branch passes its condition code test (all ARM instructions are conditional). This test however must be manipulated to allow for the fact that branches may have been predicted-taken. The normal, non-predicted action is :-

- If Branch passes its condition code test, take the branch.
- If it fails then continue executing sequentially.

This is made more complex when the BTC is present since there are now four cases to consider :-

- Branch predicted-not-taken and condition code passes.
- Branch predicted-taken and condition code passes.

- Branch predicted-taken and condition code fails.
- Branch predicted-not-taken and condition code fails.

Slightly different behaviour is required for each of these four cases.

The simple AMULET2 model uses a PLA to establish whether the instruction should execute. The interface to this is shown in figure 5.10a. When true *Pass0* indicates that the instruction is valid and has passed its condition test, and therefore should be executed. When branch prediction is added, additional logic, shown in figure 5.10b is required. *Colour* (the colour of the branch; see section 4.2.1) and *IColour* (the 'colour' of the execution unit) must be checked to ensure that the instruction belongs to the current instruction path, and is not in the shadow of a taken branch. *Xt[1:0]* indicates whether the instruction has been 'taken over' by the exception handler and turned into an exception entry.

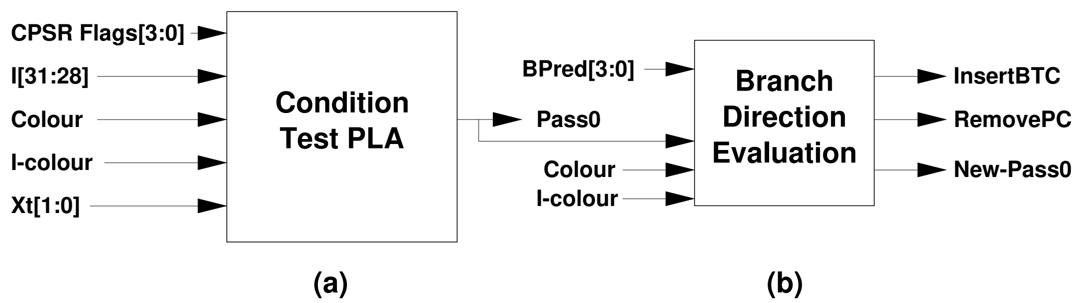

Figure 5.10 : Modified condition code evaluation

It is required that the prediction information be factored into the Condition Test PLA to produce a modified *Pass* signal, plus control for the branch recovery and BTC. This has been implemented using another PLA ('*Branch direction Evaluation*') to manipulate the original *Pass0* signal. In practice these PLAs are likely to be combined, or 'random' logic used for the extra processing.

The signals generated (*InsertBTC*, *RemovePC, NewPass0*) control the suppression of correctly predicted taken branches and the recovery when the prediction is incorrect. For each of the four possible outcomes the logic shown in figure 5.10 must provide the correct signals for the following circumstances :-

- Correctly predicted-not-taken: The branch is prevented from executing, exactly as before. In addition however decode1 has inserted the PC of the branch into the exception pipe, and this must be thrown away. This is indicated by the *RemovePC* signal.
- Correctly predicted-taken: Normally this would cause the branch to be taken, however the instruction flow has been correctly altered by the address interface so the branch execution must be suppressed.
- Branch predicted-taken but condition code fails: In this case the action should normally be to not take the branch, however the address interface has modified the instruction flow, so the execute unit must now force a branch to PC+4.
- Branch predicted-not-taken and condition code passes: This requires a branch to the target, as normal. In addition an entry must be made into the BTC so that the next time the branch is fetched it will be correctly executed. This is indicated by the *FireBTC* signal.

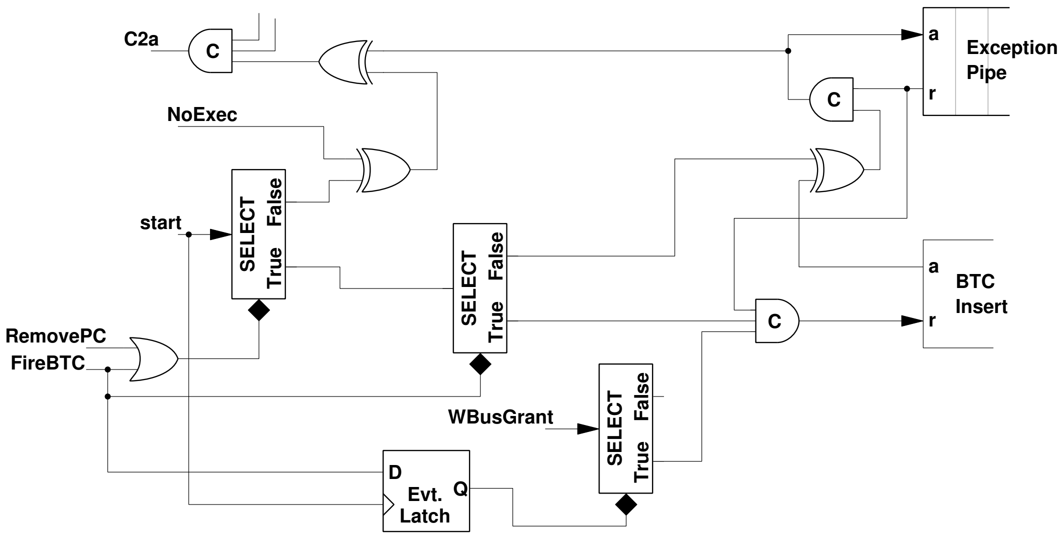

Figure 5.11 : BTC execute control logic

Inserting a new entry is the most complex of the operations. This is because the address interface and therefore the BTC are operating asynchronously to the execute unit. For a branch address to be issued the execute unit must gain access to the address interface to stop sequential PC issuing. Only at this point can the new PC be passed to it, as signalled by an event on *WBusGrant*. As the new PC is sent out from the address interface it is paired up with the PC from the exception pipe to form a ‘data + tag’ pair to be inserted into the BTC. Once the insertion has been acknowledged the PC is removed from the execute pipe and the instruction cycle is allowed to finish (*C2a* in figure 5.11).

5.3 The Effects of Self-Modifying Code on the BTC

The BTC implementation so far described requires that the processor does not modify the instruction space without then flushing the BTC. This is to avoid problems in the following circumstances :-

1. A branch is replaced by a (non-branching) instruction, causing the flow of control to change when the new instruction is executed.
2. A branch is replaced by another branch with a different target, causing the processor to branch to the wrong location.

Either case would cause the processor to fail. When might these cases arise? (2) could occur if the operating system creates jump tables dynamically, for example to handle SWIs, or if the trap-handling vectors are remapped to support extra interrupt code. This is only a problem if relative branches are used; LDR PC, *address* could be used, for example, to prevent BTC prediction problems. (1) occurs when program code is altered, either by remapping the instruction space, or by loading a new program over an existing one.

Both cases can be handled by flushing the BTC before executing the modified instructions. This should be needed infrequently and therefore should have little impact on performance.

In the case of changes to the exception vectors, one strategy might be to flush the BTC automatically whenever the execute unit takes an exception. This might appear to reduce performance, and if the handler contains few cacheable branches this would be the case. An alternative is to suppress entries for branches at the exception entry points. This does however make presumptions concerning how interrupt code is written. As with many architectural enhancements it must be decided whether placing additional restrictions on the behaviour of branches is acceptable.

If, any degree of code compatibility is to be maintained there is a requirement to allow the BTC to be flushed. The flushing might be done automatically by a memory manager using an external control signal to cause a flush, or alternatively an instruction could be defined to allow the programmer direct control. In either case the flushing must be synchronised with address comparisons since a flush must not occur midway through a CAM operation. Synchronisation could occur at the next BTC insert, since this forces the execute control and address interface to synchronise. This may be subject to delay however, depending on the current state of the BTC; a better strategy would be for the BTC to examine all data accesses for the presence of a flush condition, for example the reading of address 0.

5.4 Conclusions

An outline design for integrating a BTC into the AMULET2 architecture has been described. This has been simulated in the ASim hardware description language and functions as required. Initial results have shown approximately five percent performance improvement for the Dhrystone benchmark. This is likely to be an underestimate of improvement achieved in practice due to the high number of memory accesses in the Dhrystone benchmark, which are not affected by branch prediction. Power savings are discussed in the following chapter.

Benchmarks other than Dhrystone are difficult to run on the switch-level ASim simulation due to the lack of an operating system environment. The speed of execution (10 instructions per second) also limits the possible performance profiling. The next chapter examines these problems in more detail and attempts to evaluate the power saving and performance improvements to be expected from the BTC.

6. AMULET2 BTC Evaluation

The AMULET2 branch target cache has been successfully modelled using a hardware description language called ASim [51]. This chapter evaluates the performance of the current architecture and discusses how it might be improved.

6.1 Power consumption

The power consumed by the BTC depends on how often each of the component blocks are accessed. Table 6.1 shows, for each benchmark, the percentage of instructions that ‘hit’ in the BTC and therefore require an access to the BTC RAM, and the percentage which cause an entry to be inserted. This analysis is for a 20 entry BTC with a cyclic replacement policy, the implementation of which is described in the previous chapter.

Benchmark	Branch Density	Percent Cacheable	Percent Hits	Percent Inserts	Saved Fetches
ASim	24.0%	82.2%	41.2%	16.0%	1.03
D’stone	19.9%	68.0%	12.7%	34.3%	0.20
‘small’ C	24.9%	80.9%	25.6%	28.6%	0.56
‘large’ C	26.2%	79.6%	31.3%	23.0%	0.72
espresso	19.4%	92.0%	53.2%	16.0%	1.01
3d renderer	11.0%	91.3%	90.1%	0.1%	1.85
Vi clone	18.0%	90.6%	61.5%	12.4%	1.6
Average	**20%**	**84%**	**45%**	**19%**	**1.0**

Table 6.1 : Performance of 20 entry BTC; cyclic replacement

The ‘percent cacheable’ entry indicates the proportion of branches that can be cached in the BTC. In this case only relative branch instructions are cached (see section 5.4). The ‘percent hit’ column shows the proportion of branches that matched an entry in the BTC when they were issued. The ‘percent insert’ figure shows the proportion of executed branches that require a entry to be inserted into the BTC. As the number of entries increases the hit percentage also increases and the insert percentage decreases. With more entries available, there is an increased likelihood that an entry will remain in the BTC until it is used again, thus increasing the hit rate and reducing the number of re-insertions required. Finally the ‘saved fetches’ column shows the number of memory accesses per branch that are saved due to the presence of a BTC.

To estimate the performance and power improvements gained, the values from table 6.1 must be combined with figures for the energy consumed for a CAM lookup, BTC insert etc, together with the energy saved from memory accesses and partially executed branches after a taken branch. The energy consumed by the BTC has been ascertained by Spice simulation of the completed silicon layout. The cost of a memory access takes into account external factors such as power used by RAM, caches, busses etc. An estimate of the energy required to fetch an instruction from memory is given in section 6.2.

The CAM compare on every issued instruction address will dominate the consumption of power in the BTC. To evaluate the cost of this over the execution of a program the number of opcode fetches that are made can be totalled. This evaluation is complicated by the fact that with no prediction every branch requires, on average, three fetches (equation 6.5). However with a BTC present the number of fetches drops to around 2, reducing the total number of instruction fetches.

An alternative way of evaluating energy saved is to consider the performance improvement due to the BTC. If the throughput increases by, say, 10% the same throughput can be obtained with 10% lower energy cost. This a approximation, since it presumes that the improvement has been achieved without increasing the consumption of other parts of the processor. This is certainly not true for the address interface, which carries out a CAM compare on every address.

Establishing the exact performance improvement is difficult for the ASim simulation model, which provides an accurate switch-level simulation of the processor, since the number of programs that can be run on it is severely limited. This is due to the lack of an operating system environment, which provides support for file access which is necessary for many of the benchmarks used. In addition the simulation speed, approximately 20 instructions per second, is too slow for most benchmarks (see number of instructions executed, table 4.1). This requires that the ARMulator is used, together with the approximate power figures given above, when programs other than Dhrystone are examined. The ARMulator is an cycle-level simulator of the ARM instruction set, and does not model any of the complex asynchronous behaviour of the AMULET processor, limiting the accuracy of any results obtained.

6.2 Potential Power Savings

A silicon implementation of the BTC is currently being developed and the figures presented here are derived from the layout and further ARMulator work. Approximate calculations are also presented for the energy used by the external memory system and required to execute an instruction. These are measured from the AMULET1 evaluation card, which consists of an AMULET1 microprocessor, 128K of fast 'cache' SRAM, a 64K EPROM and a UART to provide serial communications and some limited I/O. Larger systems, such as a 'desktop' machines have not yet been investigated.

There are two factors to consider in evaluating the BTC; the overheads of adding a BTC, such as the lookup of instruction addresses in the CAM, and the power saved due to fewer external memory accesses and more accurate speculative execution.

Consider first the energy dissipated for an external memory access. The factors to consider are :-

- The power used by the pad drivers. These are the circuits which drive the external pins of the processor.
- The capacitive load on the external data and address lines.

- The power used by the external decoding logic.
- The consumption of the memory devices.

The capacitive load on the address bus will arise from the tracking to connect the upper address lines to any address decoding logic, and the lower ones direct to the memory arrays. This will be approximately 20pF per address line. The load on the data lines will depend on whether the data bus is global or is locally buffered around the memory to reduce the track lengths driven by the processor and RAM. With some gating present an estimate is 10pF per line. A modification to the ARMulator was carried out to measure the average number of address and data lines that change between one instruction fetch and another. For the address bus it was found that an average of two bits change per instruction fetch. The energy dissipated is then given by:

$$\frac{1}{2}CV^2 = \frac{1}{2}(2 \cdot 20pF) \cdot 5^2 = 0.18nJ \qquad \textit{Eqn. 6.1}$$

The data bus was shown to have a higher number of transitions, eleven on average, dissipating 0.50nJ per instruction fetch.

Consumption of a memory device is normally quoted in terms of current, but with no information as to the access patterns, supply voltage etc., under which this is measured. An approximate calculation has been made by measuring the current drawn by a 32Kx8 SRAM using a small series resistor in the power supply lead. Measuring the voltage drop across the resistor using an oscilloscope allows the energy for a read could be estimated. This gives 60nJ for a single 8-bit read. The cost of reading a 32-bit instruction is therefore approximately 240nJ, represented by *Ef* in the following equations. This shows that, in this case, when accessing external SRAM the board capacitance contributes little to the power consumption.

The average energy required per instruction can now be calculated. Firstly the energy required for an instruction without the BTC present should be calculated. For the benchmarks used the branch density is 20%. The energy per instruction can then be evaluated:

$$Energy\ per\ instruction = 0.8En + 0.2Eb \qquad \textit{Eqn. 6.2}$$

Where *En* is the average energy for a non branch instruction and *Eb* is the energy for a branch.

$$En = Ef + Ee \qquad \textit{Eqn. 6.3}$$

Ef is the energy required to fetch an instruction from memory and *Ee* the energy to execute it. Calculating this for an 'average' program is problematic since little is known about the cost of individual instructions and the way they interact. Also data transfer instructions will carry out reads and writes to external memory. In view of this the calculations presented evaluate the total energy saved in a system with a BTC, and not the percentage improvement. An estimate of the total **percentage** savings is given later.

The energy for a branch (*Eb*) is calculated below:

$$Eb = Ef \cdot CPB + Ee + Ew(1 - Accuracy) \qquad \textit{Eqn. 6.4}$$

CPB is the average number of memory cycles required for a branch, and includes the number of instructions incorrectly prefetched after a taken branch. When the BTC is added this value will drop. *Ew* represents the cost of partially executing instructions following a branch which are then thrown away due to it being taken. *Accuracy* indicates the proportion of branches that are correctly predicted. When there is no BTC present this equals the percentage of branches not-taken.

When a branch is mispredicted some instructions following the branch will have begun execution. At least one instruction will enter *primary decode*; when the branch completes, and is taken, this will proceed into execution. This instruction will fail its colour test and therefore the ALU operation will be cancelled, but the operands will however have been fetched by decode 2. There may be two other instructions to be thrown away at the primary decode, with the possibility that one of these may also enter decode before the colour change is detected, and will proceed to execution. This will then execute as well. From this simple analysis it can be seen that a correctly predicted branch will save approximately 1½ executed instructions within the processor core. Energy figures obtained for AMULET1 suggest that the cost of executing an instruction (*Ee*) is around 7nJ [55].

The CPB is calculated using the percentage of branches not-taken, from table 4.1. This is referred to as the *accuracy* since for when no BTC is present a not-taken branch is considered to be predicted correctly. When a branch is predicted correctly one memory access (the instruction fetch) is required. If the branch is mispredicted three incorrect instruction fetches following the branch occur, giving four in total. Equation 6.5 calculates CPB using these values:

$$CPB \;=\; 1 * Accuracy \;+\; 4(1 - Accuracy) \;=\; 3.1 \qquad \textit{Eqn. 6.5}$$

Eb can now be calculated by substituting into equation 6.4:

$$Eb \;=\; 240 * 3.1 \;+\; 7 \;+ 10(1 - 0.298) \;=\; 758nJ \qquad \textit{Eqn. 6.6}$$

When the BTC is added a number of factors in the above equations are modified:

1) The CPI of a branch drops to 2.1 due to one saved instruction fetch for each branch (see table 6.1).
2) The addition of the BTC alters the behaviour of the branch instruction. For example, a branch which is predicted not-taken causes the PC of the branch to be inserted into the X-pipe (section 5.2.1) and thrown away if the prediction proves to be correct.
3) The cost of an instruction fetch increases since every instruction address requires a CAM compare.
4) The BTC gives an improvement in prediction accuracy, resulting in fewer instructions incorrectly executed, on average, past a branch.

The largest single contribution to the overhead of a BTC is from the CAM. Within the CAM design considered for the BTC, each entry has a corresponding hit line which is precharged to V_{cc} and then discharged if any of the individual bits fail to match with the presented address. Therefore on each memory access all of the hit lines within the CAM will generally discharge (except where an entry matches, which is rare and will only reduce the count by one). Given an estimate of 2.5pF per hit line, and a 20 entry CAM, plus associated control logic and timing paths, this uses 0.9nJ per compare. With a branch density of 20%, there will be an average of 5 compares per branch, or 4.5nJ per executed branch.

The energy per branch (*Eb*) with a BTC present can now be calculated, by substituting into equation 6.4 and adding on the cost of the CAM (4.5nJ). A CPB of 2.1 is used, as justified in point (1) and an *accuracy* of 0.67, both derived from simulation of a 20 entry BTC.

$$Eb = 240 * 2.1 + 7 + 10(1 - 0.67) + 4.5 = 519nJ \qquad \textit{Eqn. 6.7}$$

This shows a saving of 32% for the total energy cost of a branch, as shown in equation 6.6.

6.2.1 Improved CAM Design

An interesting and novel CAM optimisation has been developed by the researcher integrating the BTC into the AMULET2 design. This makes use of the spatial locality of instruction fetches; this is often used to optimise accesses to RAM. A flag indicating that the current fetch is sequential with the previous one allows the CAM to evaluate only a small number of low order bit lines in the CAM for each access. Only when the sequentiality is broken due to a jump caused by the execution unit, or the lower bits ‘wrap around’ and cause the upper part to change, do the upper bits need to be re-evaluated. The power savings in the CAM may be significant using this scheme, though simulation is necessary to establish the optimum trade-off between the upper and lower block sizes.

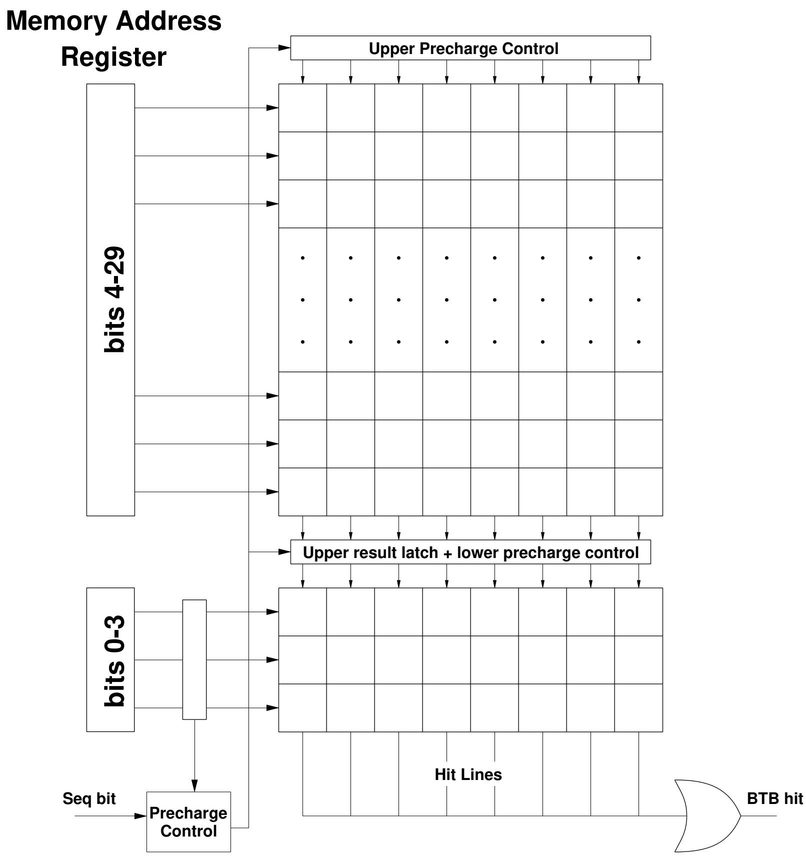

Figure 6.1 : Power-saving CAM structure (eight entries)

The relative sizes of the upper and lower sections of the CAM greatly affects the power and the performance of the structure. The pre-charging of the upper section has been shown to use approximately ten times the power of the lower section. How frequently this occurs depends on a number of factors :-

- The basic block size: The larger the basic block (a non-interruptible code sequence) the more sequential instructions are issued between branches. During sequential runs the upper part is pre-charged if the lower part of the address wraps around. A larger basic block size requires a larger number of bits in the lower part, to limit the number of times ‘wrap around’ occurs within a block. Conversely, the more frequently branches occur, the smaller the block size and therefore the smaller the lower section can be. After a branch, even if the prediction is correct, a full precharge must occur since the address is non-sequential. For a branch density of

20%, with 70% of branches taken, this implies a full precharge for at least 14% of all fetches.

- Prefetching after a branch: If a branch is incorrectly predicted not-taken, approximately three instructions after the branch will be fetched before the fetching is interrupted. This prefetching increases the effective basic block size, which also increases the probability that the lower address bits will wrap around before a branch occurs.

The distance between branches also affects the size of the lower section of the CAM. If branches are very close together a small size is optimal. This is due to the fact that if the upper part of a line fails to match, the lower part will not be precharged. The larger the upper part the less likely a line will match a particular address. Ideally, for a basic block terminated by a branch, all but one of the entries should miss, with only the matching branch entry comparing its lower bits on subsequent cycles of the block to detect when the branch is reached. This will result in only one short hit line being precharged and discharged, plus the control logic, on every sequential cycle.

Exact figures for the savings made with the modified CAM have yet to be fully quantified, but the results given in the following chapter are based on some early estimates.

6.3 Improvement in Throughput

The performance improvements have yet to be fully evaluated, due to the problems discussed earlier of carrying out accurate switch-level simulation of benchmarks. Only Dhrystone has been studied in detail; this has given approximately a five percent reduction in execution time with the BTC present. An examination of the benchmarks used has shown that around 76% of the memory activity is due to instructions. On average the BTC reduces the number of instructions fetched by 17%. This gives an overall saving of approximately 12% of memory fetches, which includes both instruction fetches and data reads and writes. The BTC will give a slightly smaller improvement than this, due to BTC overheads, such as the time required to insert entries on mispredicted branches and the degraded cycle time due to the extra control logic present.

6.4 Summary

The figures presented here calculate the energy costs and saving of executing branch instructions for AMULET2. For the AMULET1 test card the energy required to execute a branch is reduced by 32%.

7. Conclusions

7.1 Cache Technology

Cache technology has a major impact on the energy consumption of a processor. An example of this is the addition of a 4K cache to the ARM2. This increased the power from 0.1W to 1W (an order of magnitude), while increasing the performance from only 4 to 15 MIPS. This increase is partly offset however by energy savings in the rest of the system due to reduced memory and bus activity. To obtain sufficient performance from the processor core the cache must provide very high hit rates (>>90%), which in turn requires a large cache. If this cache is designed to fulfil all memory requirements, both instruction and data, its size is often of the order of 32K bytes. Using separate instruction and data caches may provide slightly lower performance for the same total size, due to poor load balancing, but each individual access will be cheaper; a mixed cache is able to balance differing instruction and data requirements [35]. Further cache splitting, into primary and secondary (or more) levels may help to reduce the cost of an access. The problem then is that an access which misses in all levels of the cache, finally triggering a main memory access, will be slow and energy is wasted due to multiple cache lookups. It might be possible to build a cache-miss 'predictor' to indicate whether a cache miss is likely to occur and therefore bypass the primary and secondary cache lookups to go direct to main memory.

7.2 Instruction Set Design

The design of recent instruction sets, such as that of the DEC Alpha [53], have stressed instruction orthogonality, for example in the use of general purpose registers to store the results of compares and the removal of function-specific registers. Eliminating these features makes high speed, multiple issue implementations possible but can restrict the ability to run at low power. This is because much of the result locality is no longer explicit in the instruction stream, resulting in many values being frequently passed between the register bank and execution unit. The use of general purpose registers for all results requires that the number of registers be larger than normal, 64 in the case of the Alpha. This is approximately double that of a processor with instruction set features designed to facilitate better result reuse, such as the PowerPC [54] or ARM [35]. These architectures have generally yielded much lower power implementations, and in the case of the PowerPC, with performance approaching that of the Alpha.

7.3 AMULET2 BTC

The design of the AMULET2 Branch Target Address Cache (BTC) has shown that it is possible to apply a traditionally mainframe technique such as a BTC to a VLSI processor, improving both power consumption and performance. The techniques used here have also recently been incorporated into commercial processors such as the Intel Pentium and the PowerPC 604, both of which include a BTC. These designs are of much higher cost in silicon area than the one chosen for AMULET2, with 256-entry 4-set associativity for the Pentium and 64-way fully associative for the PowerPC. The history and replacement policies used are complex compared to AMULET2, but use essentially the same strategy; a number of history bits are stored per branch, and dynamic is applied when the direction is known. No performance figures are available at present for these implementations so a comparison is not possible; it seems unlikely however that they would achieve much better than 80% accuracy, given the statistics for branch working set size given in chapter 4. If higher predictability (of the order of 90%) is needed there seems little substitute for a dynamic system not simply based on PC and target address.

The Branch Target Cache has been identified as a feature which, when added to a processor with a decoupled instruction fetch unit such as AMULET2, provides useful improvements in power efficiency. Table 7.1 shows an estimate of the energy saved, compared to the total energy required to execute each benchmark, for a 20 entry BTC. The energy estimates are based on measurements taken from a typical ARM-based printed circuit board. The system consists of an AMULET1 processor plus 128K bytes of 32-bit wide SRAM and 64K of 8-bit wide EPROM. AMULET1 is the first implementation of an asynchronous ARM architecture. In addition a UART is present, providing some limited I/O. The energy figures show that for accessing external memory the savings far outweigh the cost of a BTC lookup on every instruction fetch.

Benchmark	**Number of Instructions**	**Total Energy Cost of BTB**	**Total Energy Saved**
ASim	970K	1mJ	58mJ
'small' C	129K	0.1mJ	6.5mJ
'large' C	9.53M	5mJ	430mJ
espresso	3.48M	2mJ	170mJ
3d renderer	7.08M	3mJ	350mJ
vi clone	3.85M	2mJ	270mJ

Table 7.1 : Energy costs and savings for a 20 entry BTC

Total Energy Savings

The calculations presented in the previous chapter attempt to evaluate the costs and benefits gained by the incorporation of a BTC. With an uncached ARM the predominant cost is the accessing of external memory. This means that an overall estimate of the percentage of energy saved in executing a program can be made. In this case the energy saving will equal the percentage reduction in memory fetches, which is calculated in section 6.3 (12%).

The principal design aim of the Branch Target Cache is to save energy. To achieve this goal the size and configuration of this BTC differs from other implementations, which have usually been optimised towards performance. A side effect of the AMULET2 BTC is to provide some performance improvement due to the reduction in the time required to execute a correctly predicted, taken branch. The performance of an asynchronous processor can be easily tuned however (for example by adjusting the speed of the memory system) and therefore power (but not energy) can be dynamically traded off against performance. This may be required to obtain maximum efficiency from the power supply, which often has an optimum load point.

Figure 7.1 shows the variation of energy per instruction as the number of BTC entries increases. This allows the size of the BTC to be optimised purely for energy efficiency, or a combination of energy and throughput (a larger BTC generally increases the performance, see figures 4.11 to 4.13).

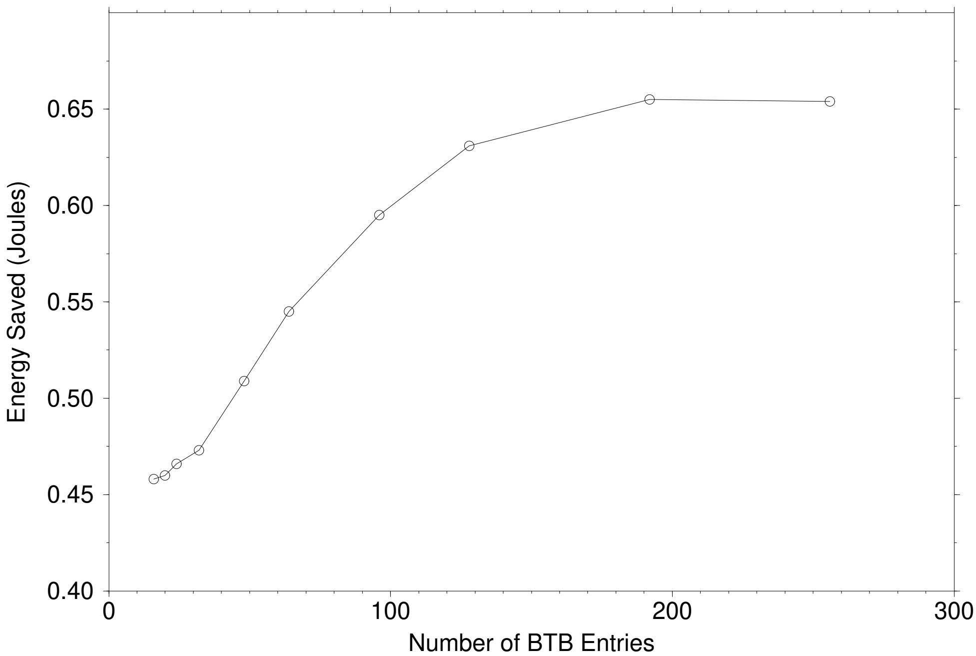

Figure 7.1 : Graph of BTC size against resulting energy savings

The various policies of BTC usage (insertion, replacement, update, removal etc) have been developed principally to give maximum predictive accuracy from the BTC. Power consumption has, however, been a factored into the decisions made; few instances have arisen where the two requirements have resulted in contradictory implications. Examining each BTC policy in turn:

Degree of History

In general, the more history bits maintained, the more accurate the prediction. However by choosing to maintain a single bit of history, and to store only branches that are predicted taken in the BTC, the history 'bit' becomes the existence, or not, of an entry for an address in the BTC. This simplifies the prediction algorithm and makes better use of the BTC.

Update and Removal

It can be observed that maintaining a history, in general, requires that the BTC is updated whenever the outcome of a branch becomes known. If the branch unit and the execute unit are loosely coupled, as with AMULET2, some form of frequent synchronisation would be required between the two. This may become a performance bottleneck and a large consumer of energy. The use of zero-bit or opcode-only prediction was shown to be perform poorly for ARM code (section 4.4.2). In general, for a single history bit, if a prediction turns out to be wrong the BTC should be updated with that new state. For example, if an entry in the BTC indicates that the branch is to be predicted taken, but the assumption is incorrect, the entry should be removed. Research showed, however, that leaving an 'incorrect' entry in the BTC produced a marginal performance improvement. This was found to be due to the ability to cope with a loop-closing branch more accurately as follows. On the last iteration of a loop the loop-closing branch is not-taken; this branch will be incorrectly predicted taken and will cause the entry to be removed from the BTC. When the loop restarts the loop-closing branch is encountered again, and will be predicted not-taken, which is incorrect. Two bit history allows tolerance of this condition (see section 4.5), but a single bit with no updates achieves the same effect (by leaving the branch in place), with a much simpler update strategy.

Insertion

The BTC functions by examining the addresses issued to the memory system and by changing the next address to be issued if it matches a previously taken branch. Before a branch can be predicted an entry must be present in the BTC. It makes little sense then to insert un-taken branches, since the default action of the address interface is to issue sequential instruction fetches until altered by the execution unit. Therefore only when a branch is incorrectly ‘predicted’ not-taken, is an entry placed in the BTC.

Replacement

If there are no available entries in the BTC and a new entry is to be made, an existing item must be overwritten. A number of algorithms were studied closely. LRU (Least Recently Used) is often the choice for BTC’s but this was found to perform no better than a simple cyclic scheme. This seems to be due to the cyclic nature of branches, and indeed when the behaviour of an LRU BTC was examined it was observed to be functioning almost identically to a cyclic scheme. Cyclic and LRU algorithms have a clear pathological failure mode, where the working set of entries is just greater than the size of the cache. A random replacement policy avoids this problem, but with slightly lower performance where the working set is close to the size of the BTC, since entries may be thrown out unnecessarily. For a small BTC, as required for AMULET2, a random policy may be optimum. Figures 4.7 and 4.8 demonstrate the advantages of a random policy for small BTC’s. Cyclic replacement may be easier to implement however, compared to a random scheme, due to the difficulty in generating a provably random entry number which is not a power of two (20 entries are likely to be implemented). It may be that a weighted random policy is be acceptable, and this will be investigated in the near future.

Final BTC Configuration

The factors discussed above resulted in a BTC design with the following configuration :-

- 20 fully associative entries.
- Cyclic or random replacement, depending on required tolerance of cache thrashing.
- Single history bit per branch.
- No update of entries on misprediction.
- Store only branches predicted taken (implied by the BTC mechanism).

This results in an energy saving of approximately 12% (section 6.3) when executing the benchmarks on the AMULET test card. A similar increase in throughput is obtained. Both improvements are due to an improvement in prediction accuracy from 30% to 71% (section 6.2). The choice of random or cyclic replacement is mainly dictated by whether or not Dhrystone is affected by the addition of a BTC. There are 24 branches in the Dhrystone loop, requiring greater than 23 BTC entries if cyclic replacement is to be used; fewer than 25 entries causes the BTC to 'thrash'. The silicon area required for 24 entries may be too large for the AMULET2 implementation however.

7.4 Asynchronous Design

The use of asynchronous design to build low power systems has been pursued in the AMULET group for several years. During this time a number of large asynchronous circuits have been designed, and working silicon produced. So far, at least for microprocessor design, it has not been shown to produce better results (either lower power or higher performance) than equivalent systems implemented synchronously. The BTC CAM design has, however, shown an advantage of asynchronous design, namely the ability of asynchronous logic to tolerate arbitrary delays through blocks. In this example, when a full and late BTC precharge occurs (due to a taken branch from the execution unit) the instruction address issue must wait longer than normal for the slower CAM lookup to finish before it can continue. In a synchronous design the worst case delay must always fit into the clock period, making a slow, but abnormal case difficult to allow for.

The two-phase design style described here has been found to facilitate the design and find faults with the resulting circuits. There are problems however, due to the slow speed and large area of the standard building blocks, such as a select block. These blocks have a large number of internal nodes and transistors, and for an event on the input, which drives two internal gates, four internal transitions occur. The use of a four-phase design style may improve both the speed and the power consumption asynchronous control logic due to the simpler building blocks. The silicon design of the BTC is being designed using four-phase logic.

7.5 Further Research

The energy savings given here for executing a branch on AMULET2 are based on accessing large external memory systems. Although a small ‘cache’-SRAM-based memory system has been studied, data on larger DRAM devices suggests similar energy consumption. Work is currently being carried out to design a cache and memory management system for the AMULET2 core, and it is likely that the cost and benefits of a BTC will change under these conditions. Studies should be carried out to estimate the energy costs of a cache, and also the differences between sequential and non-sequential fetches, both in caches and external memory devices. Sequential fetches are likely to cost considerably less that non-sequential ones, reducing the benefits of a BTC.

The branch prediction unit described has been designed as a power-saving addition to AMULET2. Although the BTC improves performance, the increase is small (around 5%, measured from the current ASim models). Low power consumption, without an acceptable level of performance, will not satisfy many of the current demands of portable computing devices. The accuracy of branch prediction achieved with a BTC is limited by the prediction unit working only on the outgoing address stream. This requires a large CAM and RAM structure to hold the prediction information for many branches, so that taken branches are automatically followed. To achieve further accuracy a more extensive prediction strategy is required. Branch correlation has been shown to give excellent results [10] (see section 3.6.3) and the adaptation of this, and other mechanisms, to an asynchronous implementation should be examined.

One problem observed with the current AMULET address and data units is that the processes of issuing PC addresses and decoding the resulting instructions from memory are decoupled. To implement a dynamic, instruction-based scheme, which predicts only the **direction** of the branch, the prediction unit in the data interface/decode unit must evaluate the branch target and synchronise with the address interface to start fetching from the new address. Currently, only the ALU (execute stage) can affect the PC in the address interface, which means that the arbiter would require another input to be added to allow interruption from third source (section 4.2.1).

Pipelining the memory system also demands a much higher prediction rate, because there will be more outstanding and invalid memory requests when a branch is found to be incorrectly predicted. For example, to maintain the number of instruction fetches per branch at 1.6 the prediction accuracy must increase from 80% to 90% when the branch cost is increased from 4 to 7 cycles, to model an extra 3 pipeline stages in the memory system. A three-stage pipelined cache and memory management system is currently being designed for AMULET2. 90% is not a practical prediction accuracy for a BTC if power savings are to be maintained, since its size would have to be of the order of 512 entries.

A Branch Target Cache has proved to be a relatively simple structure to incorporate into AMULET2. It is likely that other forms of branch prediction such as Branch Correlation will require much more work including a major reorganisation of the instruction fetch mechanism. There is a great deal of work still to be done on instruction fetch mechanisms for asynchronous processors.

Appendix A : Example Output

This is a sample of the output generated by the BTC simulator.

```
IType = LDM  (Total number = 20129)
 Percentage of branches = 8.6%
  Total taken = 65.0%
  Prediction accuracy = 82.0%
  Taken branches correct = 75.8%
  Not-taken branches correct = 93.4%
ConditionCode;
  EQ = 82.1%, 5698
  NE = 95.0%, 3873
  CC = 0.0%, 1
  MI = 50.0%, 2
  LS = 95.8%, 120
  GE = 85.7%, 7
  LT = 58.1%, 215
  LE = 98.5%, 394
  AL = 77.8%, 9819
IType = LDR  (Total number = 2259)
 Percentage of branches = 1.0%
  Total taken = 100.0%
  Prediction accuracy = 97.0%
  Taken branches correct = 97.0%
  Not-taken branches correct = NaN%
ConditionCode;
  NE = 99.9%, 2134
  AL = 52.8%, 125
IType = BRANCH  (Total number = 157818)
 Percentage of branches = 67.2%
  Total taken = 63.5%
  Prediction accuracy = 87.9%
  Taken branches correct = 89.5%
  Not-taken branches correct = 85.2%
  Percentage backward = 29.6%
   backward taken = 74.8%
   taken backward correct = 83.4%
   not-taken backward correct = 54.3%
   backward correct = 76.1%
  Percentage forward = 70.4%
   forward taken = 58.8%
   taken forward correct = 92.7%
   not-taken forward correct = 93.1%
   forward correct = 92.9%
Forward branch CPI = 1.21
Backward branch CPI = 1.72
ConditionCode;
```

```
  EQ = 85.4%, 50860
  NE = 88.7%, 40866
  CS = 88.6%, 166
  CC = 94.9%, 1935
  MI = 99.0%, 5534
  PL = 95.9%, 413
  HI = 85.9%, 6160
  LS = 80.1%, 806
  GE = 93.3%, 8543
  LT = 82.5%, 9456
  GT = 83.9%, 3132
  LE = 84.4%, 12025
  AL = 97.3%, 17922
IType = BL  (Total number = 35335)
 Percentage of branches = 15.0%
  Total taken = 74.3%
  Prediction accuracy = 96.6%
  Taken branches correct = 95.4%
  Not-taken branches correct = 99.9%
  Percentage backward = 52.5%
   backward taken = 85.2%
   taken backward correct = 95.5%
   not-taken backward correct = 100.0%
   backward correct = 96.2%
  Percentage forward = 47.5%
   forward taken = 62.3%
   taken forward correct = 95.3%
   not-taken forward correct = 99.9%
   forward correct = 97.0%
Forward branch CPI = 1.09
Backward branch CPI = 1.11
ConditionCode;
  NE = 57.7%, 26
  HI = 100.0%, 6
  LT = 100.0%, 9062
  AL = 97.1%, 26241
IType = DATAOP  (Total number = 18247)
 Percentage of branches = 7.8%
  Total taken = 73.2%
  Prediction accuracy = 87.5%
  Taken branches correct = 87.3%
  Not-taken branches correct = 88.2%
ConditionCode;
  EQ = 79.2%, 6934
  NE = 96.0%, 25
  LS = 94.1%, 17
  GE = 56.5%, 262
  LT = 95.5%, 22
  LE = 99.9%, 809
  AL = 93.5%, 10178
IType = SWI  (Total number = 1217)
 Percentage of branches = 0.5%
  Total taken = 99.8%
  Prediction accuracy = 98.0%
  Taken branches correct = 98.1%
```

```
  Not-taken branches correct = 50.0%
ConditionCode;
  NE = 66.7%, 9
  AL = 98.9%, 1208

Number of instructions = 970177
Number of branch instructions = 234922 (24.2% of I's)
Number of branch instructions taken = 156424 (66.6% of branches, 16.1%
of I's)
```

This is for an 'ideal' BTC, run on ASim, simulating a simple ring counter. The percentage for each condition code is how predictable it is, eg LDMEQ was predicted correctly 82.1% of the time, based on the direction of the previous execution of the branch. The second number is the absolute number of that type of branch executed, to allow the prediction accuracy to be put into perspective.

Appendix B : Direct-Mapped BTC Performance

The following graphs show the effects of varying degrees of associativity for the six benchmarks programs used. The benchmarks were chosen to cover a wide range of possible applications and programming styles. The benchmarks used were:

ASim. An event-driven digital simulator, written and used by Acorn Computers to design the ARM chip-set. The program was used to simulate a Johnson counter.

Dhrystone. A simple integer benchmark, often used to specify the performance of a processor (quoted as ‘Dhrystones per second’). In this case the program executes the dhrystone ‘loop’ 10000 times.

Espresso. A PLA optimiser, which is part of the OCTOOLS software suite. The input file was one of the provided example files.

3-D Renderer. A hand-coded (assembly language) 3-dimensional image renderer. The test was to render two frames from a rotating teapot sequence.

C-Compiler. A public domain C compiler for the 68000 processor family. Two compilation runs of part of the compiler source were studied, of 19 and 709 line programs (‘small’ and ‘large’ compiler benchmarks).

A Vi-clone. This is a public domain package called ‘Elvis’, and provides an clone of the standard Unix™ text editor. The test sequence was to load a file, scroll forward a few pages then insert and edit a text sequence.

The benchmarks used were compiled using the current release of the ARM C compiler, with the exception of the ‘renderer’ which was hand coded in ARM assembler. The reason this was included is because it contains many 'tricks' to improve performance, as well as extensive conditional execution; it was therefore expected to be sufficiently different from compiled code to warrant inclusion. The choice of C-compiled programs was restricted to those that required no special libraries such as those providing an interface to a 'windowed' environment. Since these 'interactive' applications now form an important part of a processor's work load a text editor was included. This used only ANSI terminal codes for controlling the display, allowing it to run correctly in a terminal window. This was unfortunately the most interactive program that could be successfully built using the ANSI compiler.

The horizontal scale of the graphs shows the degree of associativity for each result. Three BTC sizes were considered, of 16, 32 and 64 entries. For each size the tests varied from fully associative (1S/xE - one set, of x entries) to direct mapped (xS/1E - x sets, each of one entry).

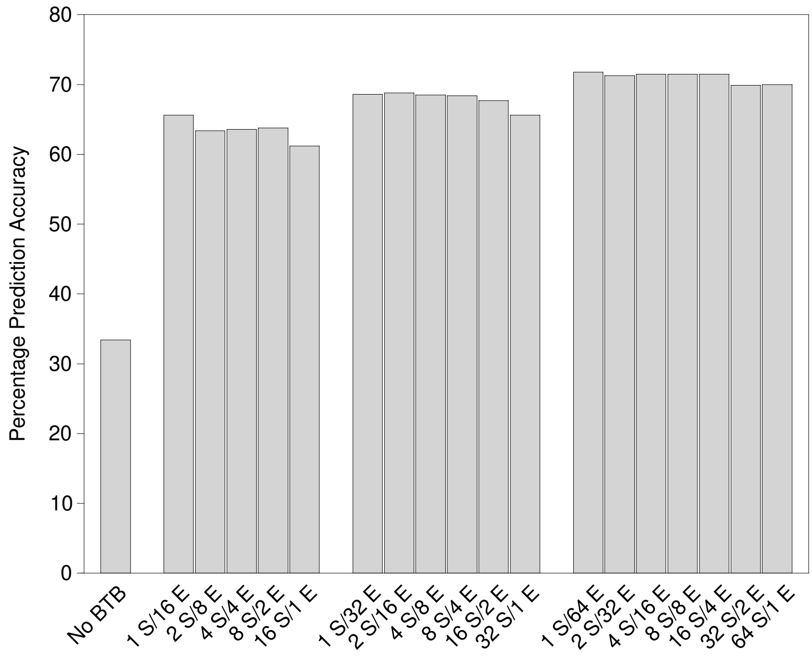

Figure B.1 : Varying associativity for ASim

The results for ASim (figure B.1) show clearly the effect of changing from a fully associative to direct mapped cache, namely that the number of entries needs to approximately double to maintain the same prediction accuracy.

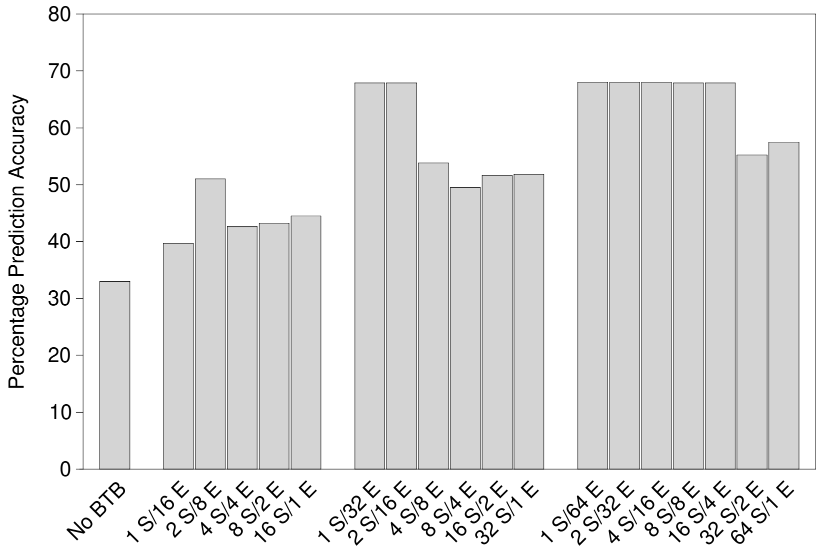

Figure B.2 : Varying associativity for Dhrystone

Dhrystone has proved to be particularly problematic when judging the performance of the BTC, and figure B.2 adequately demonstrates this. Due to the small size and particular behaviour of the Dhrystone loop the prediction accuracy is basically ‘all or nothing’ depending on how the branch addresses in the loop map to the cache structure. In this case there are 24 branches to be cached, and the alignment causes the interesting effect for a 16 entry cache that full associativity is the **worst** structure to use! Knowledge of the BTC structure would clearly be an advantage to a compiler in the placement of branch instructions in the generated code.

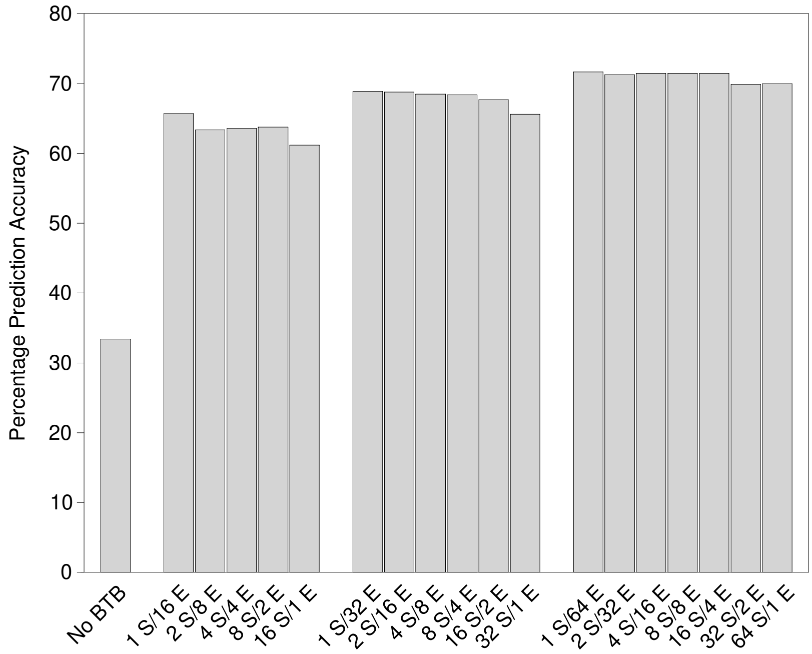

Figure B.3 : Varying associativity for Espresso

The prediction accuracy for Espresso (figure B.3) again shows that doubling the cache size allows a direct mapped cache to be used with no loss of performance.

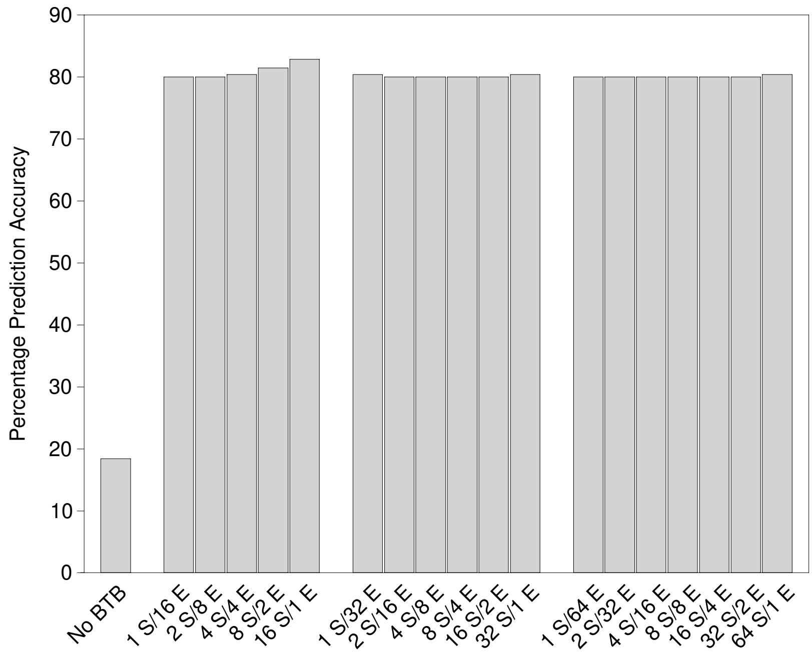

Figure B.4 : Varying associativity for 3d-Renderer

The 3-d 'renderer' (figure B.4) has a few very major loops that prove to be highly predictable, and hence almost any cache configuration gives the same results.

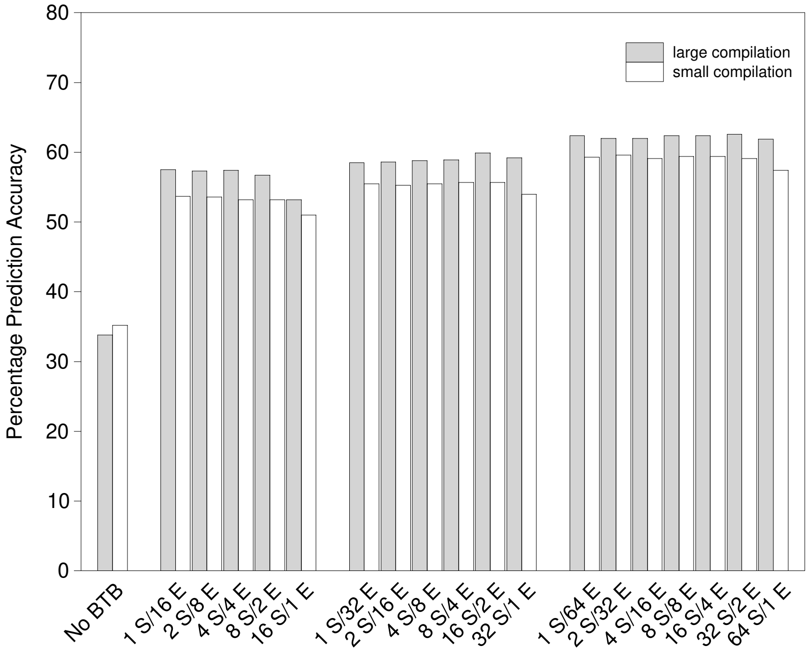

Figure B.5 : Varying associativity for C compiler

The results for the C compiler (figure B.5) are generally the worst of the benchmark set, though it again confirms the theory concerning direct mapped caches. Interestingly the small compilation was consistently worse than the large one, even though it gave better results with no BTC present.

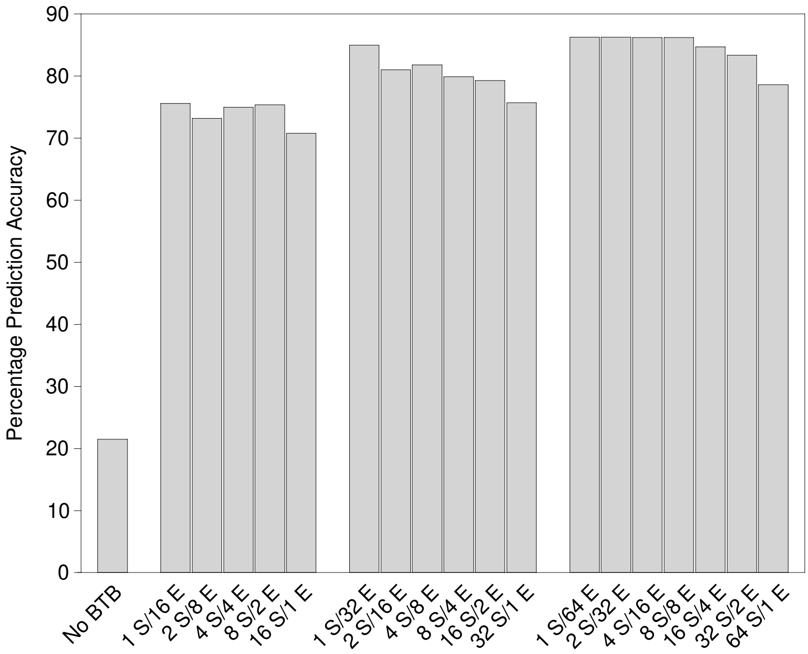

Figure B.6 : Varying associativity for Vi clone

Figure B.6 shows how Vi clone performs for differing degrees of associativity. For 16 and 32 entries the results are fairly consistent, but for a 64 entry cache the accuracy significantly. This is likely to be due to the small number of dominant branches present in the program, which are likely to perform poorly in a direct mapped cache due to thrashing of important entries.

Appendix C : The ARM Microprocessor

The ARM architecture was designed in 1984 by Steve Furber and his team at Acorn Computers [35] and was the first commercial RISC microprocessor. It has a number of interesting features such as multiple register load and stores and a fully conditional instruction set. The first commercial version of the ARM was the ARM2; this is described first followed by the modifications to create the ARM6, the current 32-bit ARM processor core.

C.1 The ARM2

The ARM2 is a 32-bit data architecture with a 26 bit address space. It has 27 registers which are 32 bits wide. At any one time there are fifteen registers visible (R0-R14) plus the program counter (R15). The current processor mode dictates which of the 27 registers map to the R0-R14. The modes are :-

- User
- Supervisor (SVC)
- Interrupt (IRQ)
- Fast Interrupt (FIQ)

Figure C.1 Shows how the registers are mapped in the various modes.

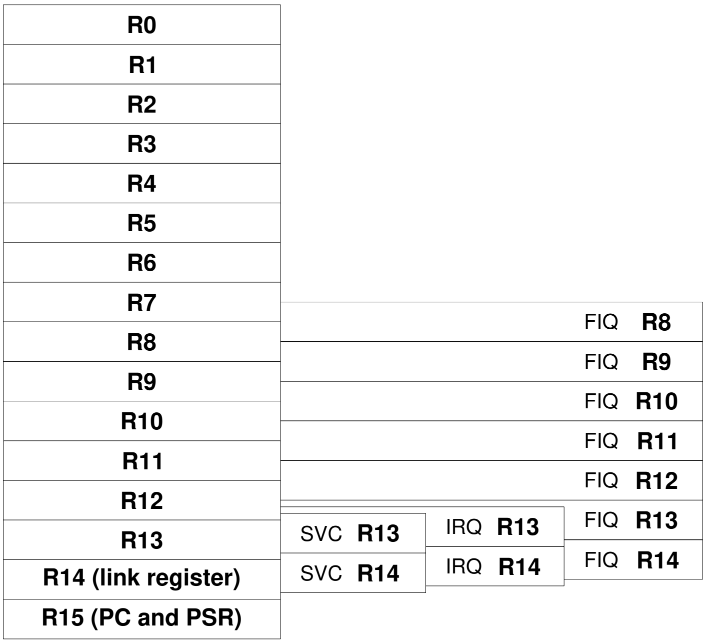

Figure C.1 : ARM2 Register allocation

Because of the 26 bit address space, and the fact that since instructions are word (four byte) aligned, there are eight 'spare' bits in R15. This are used to hold the current processor mode plus the four condition code flags.

C.2 ARM2 Instruction Set

The ARM instruction set is based on a load-store model. This means that there is a specific class of instructions which are allowed to access memory, and these comprise load and stores only. All data processing instructions operate only on registers, with no memory to memory operations supported.

There are six basic instruction types :-

- Data processing operations performed on the registers. These make use of the ALU, barrel shifter and multiplier.
- Single register load and stores, allowing a number of pre- and post-indexed addressing modes, allowing easy implementation of stacks and queues.
- Multiple register load and stores to allow fast procedure call and context switching.

- Software interrupts.
- Branches.

Figure C.2 shows the format of R15, the combined PC and PSR register.

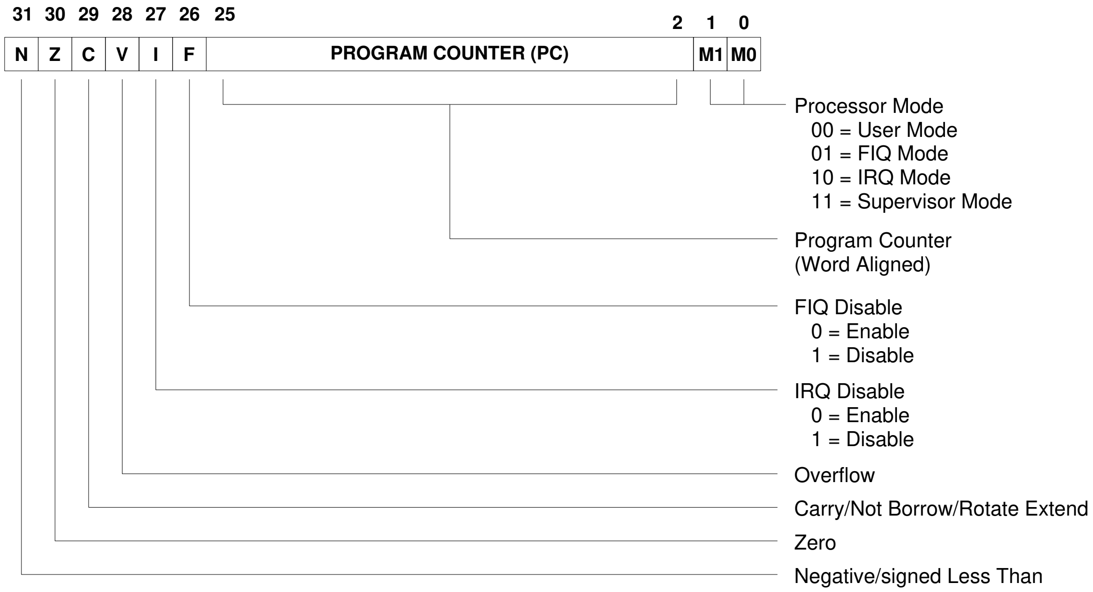

Figure C.2 : ARM2 PC and PSR format

C.3 The ARM6

The ARM6 is a ful 32-bit implementation of the ARM instruction set. In addition it implements extra operating modes to ease operating system design. The new modes each have their own R13 (stack register), R14 (link register) and SPSR (Saved Program Status Register), increasing the total number of registers to 31. The new register structure is shown below in figure C.3.

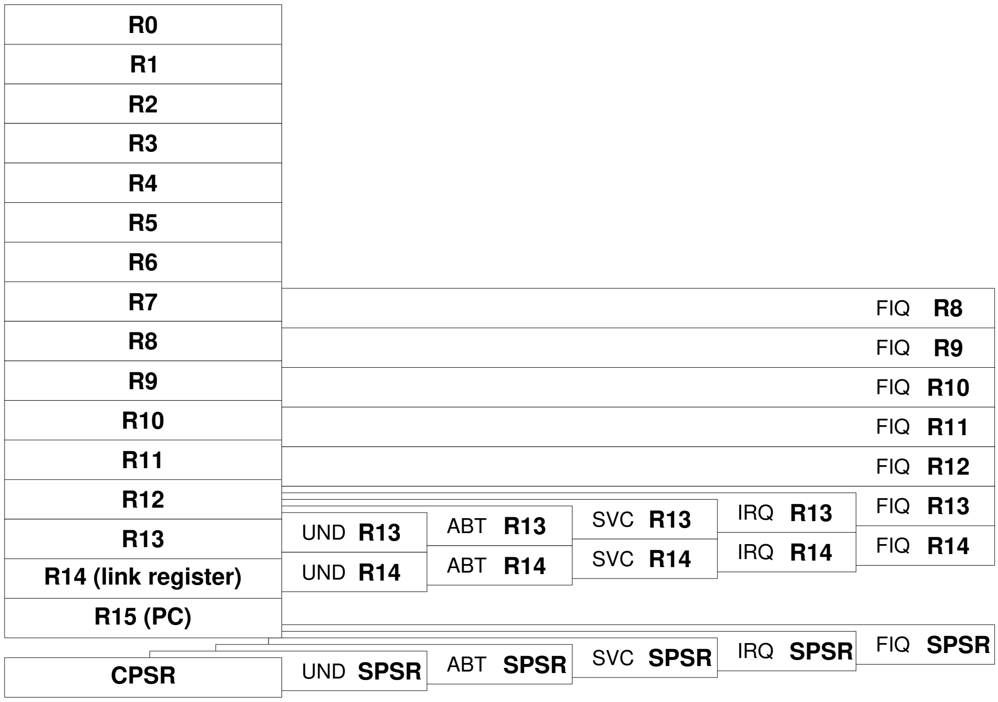

Figure C.3 : ARM6 register allocation

The move to full 32-bit addressing required the status and mode bits to be moved to a separate register, the CPSR (Current Processor Status Register). In addition to the CPSR there is one SPSR (saved Process Status Register) for each of the privileged modes. This allows the current mode and flags to be saved across mode changes. Previously this happened automatically with the saving of the PC. Extra instructions have also been added to allow access to the CPSR and SPSR registers via one of the general purpose registers.

Compatibility with the ARM2 has been ensured by the addition of 26 bit program and data configuration pins. When in 26 bit mode the ARM6 behaves exactly as an ARM2.

References and Bibliography

[1] Thomas Ball, James R. Larus, '*Branch Prediction For Free*', Technical Report #1137, Computer Sciences Department, University of Wisconsin - Madison, USA, February 9th 1993.

[2] Scott McFarling, John Hennessy, 'Reducing the Cost of Branches', *Proceedings of the 13th International Symposium on Computer Architecture*, June 1986 pp.396-403.

[3] David R. Ditzel, Hubert R. McLellan, 'Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero', *Proceedings of the 14th International Symposium on Computer Architecture*, 1987 pp.2-9.

[4] Carl O. Stjernfeldt, Edward W. Czeck, David R. Kaeli, '*Survey of Branch Prediction Strategies*', CE-TR-93-05, July 28th 1993.

[5] R. W. Holgate, R. N. Ibbett, 'An Analysis of Instruction-Fetching Strategies in Pipelined Computers', *IEEE Transactions on Computers*, Vol. C29, No.4 1980, pp.325-329.

[6] Peter L. Bird, Uwe F. Pleban, Nigel P. Topham, Henrik Scheuer, 'Semantics Driven Computer Architecture', *Parallel Computing '91*, pp.267-273.

[7] Nathalie Drach, Andre Seznec, '*MIDEE: Smoothing Branch and Instruction Cache Miss Penalties on Deep Pipelines*', report no. 2038, INRIA, France.

[8] Manolis Katevenis, Nestoras Tzartzanis, 'Reducing the Branch Penalty by Rearranging Instructions in a Double-Width Memory', *Proceedings of the 18th International Symposium on Computer Architecture*, May 1991, pp.15-27.

[9] Adam R. Talcott, Wayne Yamamoto, Mauricio J. Serrano, Roger C. Wood, Mario Nemirovsky, 'The Impact of Unresolved Branches on Branch Prediction Scheme Performance', *Proceedings of the 22th International Symposium on Computer Architecture*, 1994, pp.12-21.

[10] Tse-Yu Yeh, Yale N. Patt, 'Alternative Implementations of Two-Level Adaptive Branch Prediction', *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992, pp.124-134.

[11] Tse-Yu Yeh, Yale N. Patt, 'A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History', *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993, pp.257-266.

[12] Jose Luis Pino, Balraj Singh, '*Performance Evaluation of One and Two-Level Dynamic Branch Prediction Schemes over Comparable Hardware Costs*', EECS Department, University of California, Berkeley, USA.

[13] Brad Calder, Dirk Grunwald, 'Fast & Accurate Instruction Fetch and Branch Prediction', *Proceedings of the 22th International Symposium on Computer Architecture*, 1994, pp.2-11.

[14] Ching-Long Su, Alvin M. Despain, 'Branch With Masked Squashing in Superpipelined Processors', *Proceedings of the 22th International Symposium on Computer Architecture*, 1994, pp.130-140.

[15] Dionisios N. Pnevmatikatos, Gurindar S. Sohi, 'Guarded Execution and Branch Prediction in Dynamic ILP Processors', *Proceedings of the 22th International Symposium on Computer Architecture*, 1994, pp.120-129.

[16] Johnny K. F. Lee, Alan Jay Smith, 'Branch Prediction Strategies and Branch Target Buffer Design', *IEEE Compute*r, January 1994, pp.6-22.

[17] David J. Lilja, 'Reducing the Branch Penalty in Pipelined Processsors', *IEEE Computer*, July 1988, pp.47-55.

[18] David R. Kaeli, Philip G. Emma, 'Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns', *Proceedings of the 18th International Symposium on Computer Architecture*, May 1991, pp.34-41.

[19] Wen-mei W. Hwu, Thomas M. Conte, Pohua P. Chang, 'Comparing Software and Hardware Schemes For Reducing the Cost of Branches', *Proceedings of the 16th International Symposium on Computer Architecture*, May 1989, pp.224-233.

[20] John A. DeRosa, Henry M. Levy, 'An Evaluation of Branch Architectures', *Proceedings of the 14th International Symposium on Computer Architecture*, 1987, pp.10-16.

[21] Brian Case, 'Intel Reveals Pentium Implementation Details', *Microprocessor Report*, Volume 7, Number 4, pp.9-17.

[22] Lindley Gwennap, 'Intel Adds Low-Power Features to Every 486', *Microprocessor Report*, Volume 7, Number 8.

[23] William R. Hambergen, John S. Fitch, '*Packaging a 150 W Bipolar ECL Microprocessor*', WRL Research Report 92/1.

[24] Deo Singh, 'Prospects for Low Power Microprocessor Design', presentation at the International Workshop on Low Power Design, Napa Valley, April 1994.

[25] Scott Hauck, '*Asynchronous Design Methodologies: An Overview*', Technical report 93-05-07, Department of Computer Science and Engineering, University of Washington.

[26] David Jaggar, '*A Performance Study of the Acorn RISC Machine*', M.Sc. thesis, University of Canterbury, New Zealand, 1990.

[27] Apoorv Srivastava, Alvin M. Despain, 'Prophetic Branches: A Branch Architecture for Code Compaction and Efficient Execution', *Proceeding of MICRO-26*, 1993.

[28] Dobberpuhl et. al., ‘A 200-MHz Dual-Issue CMOS Microprocessor’, *IEEE Journal of Solid State Circuits*, Vol. 27, No. 11, November 1992, pp.1555,1565.

[29] Philip B. Endecott, ‘*Processor Architectures for Power Efficiency and Asynchronous Implementation*’, M.Sc. Thesis, Manchester University, 1993.

[30] John L. Hennessy, David A. Patterson, ‘*Computer Architecture A Quantitative Approach*’, Morgan-Kaufman, 1990, ISBN 1-55860-069-8.

[31] C. Rowen, ‘MIPS R4200, An Innovation in Low-Power Microprocessor Design’, presentation at Microprocessor Forum Europe, May 1993.

[32] David R. Ditzel, Alan D. Berenbaum, ‘The Hardware Architecture of the CRISP Microprocessor’, *Proceedings of the 14th International Symposium on Computer Architecture*, 1987 pp.309-319.

[33] Hewlett-Packard, ‘Precision Architecture and Instruction Reference Manual’, Hewlett-Packard Company, 1986.

[34] Brad Calder, Dirk Grunwald, ‘*Branch Prediction Architectures for 64-bit Address Space*’, technical report CU-CS-690-93, University of Colorado, November 1993.

[35] Stephen B. Furber, ‘*VLSI RISC Architecture and Organisation*’, Marcel Dekker Inc., New York, 1989, ISBN 0-8247-8151-1.

[36] R. M. Russell, ‘The Cray-1 Computer System’, *Comm. ACM*, Vol. 21, No.1, Jan. 1978, pp.63-72.

[37] B. J. Smith, ‘Architecture and Applications of the HEP Multiprocessor Computer System’, *Proceedings to the Real-Time Signal Processing IV, SPIE*, 1981, pp.241-248.

[38] Nigel C. Paver, ‘*The Design and Implementation of an Asynchronous Microprocessor*’, Ph.D. Thesis, University of Manchester, UK, 1994.

[39] ARM Ltd., ‘*ARM Software Development Toolkit Reference Manual*’, ARM Ltd., Cambridge UK.

[40] S-T Pan, K. So, J. T. Rahmeh, ‘Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation’, *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992, pp.76-84.

[41] INMOS, ‘*The T9000 Transputer Hardware Reference Manual*’, INMOS Ltd, UK, 1993.

[42] AMD, ‘*Am29000 Users’s Manual*’, Advanced Micro Devices, 1987.

[43] Anantha P. Chandrakasan et. al., ‘Low-Power CMOS Digital Design’, *IEEE Journal of Solid State Circuits*, Vol. 27-4, April 1992.

[44] Manolis G. H. Katevenis, '*Reduced Instruction Set Computer Architecture for VLSI*', ACM Doctoral Dissertation Award Series, MIT Press, 1985.

[45] Neil H. E. Weste and Kamran Eshraghian, '*Principals of CMOS VLSI Design, A Systems Perspective, 2nd edition*', Addison Wesley, 1992.

[46] M. J. Morant, '*Integrated Circuit Design and Technology*', Chapman and Hall, 1990.

[47] Alan Jay Smith, 'Cache Memories', *Computing Surveys*, Vol. 14, No. 3, September 1982.

[48] Integrated Device Technology, Inc., '*R4400 Data Sheet*', 1993.

[49] J. E. Smith, 'A Study of Branch Prediction Strategies', *Proceedings of the 8th International Symposium on Computer Architecture*, 1981 pp.135-148.

[50] I. E. Sutherland, 'Micropipelines', *Communications of the ACM*. 32(6), January 1989, pp.720-738.

[51] L. Smith, '*A Guide to the ASIM Simulation and Modelling Language*', ARM Ltd., 1992.

[52] Gill, Corwin and Logar, '*Assembly Language Programming for the 68000*', Prentice-Hall, 1987.

[53] Digital Equipment Corp., '*Alpha Architecture Handbook*', DEC, 1992.

[54] IBM, '*PowerPC Architecture*', October 1993.

[55] Dr. Steve Temple, Personal Communication, 1994.

[56] H. B. Bakoglu, '*Circuits, Interconnections, and Packaging for VLSI (VLSI Systems Series)*', Addison Wesley, 1990.

[57] Henrik Scheuer, Personal Communication, 1994.