SPLIT-DATAPATH EXECUTION FOR LOW POWER

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

September 2001

By
Matthew Evans
Department of Computer Science

Contents

List of Tables

List of Figures

Abstract

Since the introduction of smartcards in the mid-1980s, the technology has improved drastically with many current cards incorporating tens of kilobytes of storage and even high-performance 32-bit processors. Most smartcards rely on making electrical contact with a reader when they are accessed.

Contactless smartcard ICs are simpler because of power limitations; they are powered directly from a magnetic field with no physical connections and must operate on very low power levels. Applications and users demand ever-increasing processing power from such cards but current high-performance processors consume too much power for many low-energy applications.

A scheme is proposed that can avoid unnecessary work by slicing the datapath and inferring partial results, rather than evaluating them. Such a system is described; for each partition metadata is supplied to the execution unit which describes the operand and allows unnecessary operations to be avoided.

A software model of a 32-bit ARM processor with a partitioned datapath shows that, when real programs are executed, partial ALU operations can be avoided – or replaced with simpler operations – up to 70% of the time. This suggests that a split datapath would be advantageous in saving energy in a processor. Possible architectures are examined, and a VLSI implementation of an asynchronous byte-serial split-datapath ALU is presented. Power and timing analysis was performed on the layout, and the split-datapath ALU is compared to a contemporary, asynchronous 32-bit ALU in terms of performance, power use, energy efficiency and silicon area.

The split-datapath ALU implementation, being a prototype, showed some inefficiencies but may still have promise in the future. Further optmisations to the implementation would improve its efficiency, however real savings are likely to appear with very wide execution widths (such as 64 or 128-bit), and other power-hungry circuits such as system buses.

Declaration

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

Copyright

Acknowledgements

I would like to thank my supervisor, Dr. Jim Garside, for his advice, help and his work on the ARM emulator; my advisor, Prof. Steve Furber; ARM Ltd. and the EPSRC for their support during this work; my parents; Frank Weigel for his work on the JimSim ARM emulator; Andrew Bardsley, Peter Riocreux, John Bainbridge, Ian Rogers and Charles Brek for the many interesting conversations and ideas; Will Toms for the conversations; Clüre Hobley for everything; Jim Dixon and his introduction to academia; the Amulet group; Marge Inaverra; David StHubbins, Nigel Tufnel, Derek Smalls and Viv Savage for going up to eleven. Last but not least I would like to thank all those I have forgotten, hopefully accidentally.

Chapter 1

Introduction

1.1 Smartcards

Smartcards were first introduced in France in the mid-1980s as a public telephone payment card. They are credit card-sized plastic cards containing an integrated circuit (IC) that identifies itself to the payphone and presents a count of the 'money' stored in a small memory. The IC allows this counter to be decremented by the reader in the phone as the credit is used up. It is not possible for the IC to increment the counter, preventing fradulent 'recharging' of the credit on the card.

As with any technology, significant advances have been made in the years since its inception. The ability to store larger amounts of data in the card was deemed very useful and the storage capacities of the ICs have increased from tens of bits to tens of kilobytes. It was also realised that this stored data could be very valuable and smartcards with cryptographic data protection were developed to prevent the data from being read without authorization and (just as importantly) to prevent its alteration. It is now common to find smartcards with several kilobytes of memory (EEPROM, ROM, RAM), a microprocessor and perhaps a cryptographic coprocessor.

The advances in technology have revealed, and in some cases been driven by, many new applications for smartcards. Electronic purses, money [Mon95], and credit are now possible because of the encryption capabilities of newer cards (preventing fraud by verifying both the card and the reader to each other, for example to prevent an unscrupulous user from posing as a bank to increase the 'money' on his card). Television subscriber smartcards can ensure that particular subscribers only have access to the channels they have paid for and only allow the broadcaster to change their selections. Store loyalty smartcards can store shopping preferences and buying habits, facilitating more advanced data gathering on the shoppers. Identification cards can verify the user to control access to buildings or equipment.

More recently the processing power of smartcard CPUs has increased enormously. Some smartcards now contain powerful 32-bit CPUs, Java runtime environments [SMI01] etc. together with much larger memories. The increased computational power increases the flexibility of the card, enabling one smartcard to host many applications making the card very convenient for its user. In the future we may see a smartcard that is a personal ID card, driving license, credit/debit card, electronic cash, underground train pass, and which stores biometric data about its owner – all protected by strong cryptographic measures. The card could authorise its owner by his fingerprint and provide details of his medical history and allergies to emergency services. The benefits of large data storage, encryption and ruggedness clearly leave traditional magnetic-stripe cards behind.

The plastic cards are a standard size and shape, and contain a small IC. This IC is contained behind a small set of terminals on one side of the plastic card; the physical dimensions of a smartcard and the position of these terminals are set out in the ISO7816 standard. When a card is plugged into a reader these terminals draw power from the reader and provide a synchronous serial communication channel. ISO7816 also dictates the electrical and communications characteristics of smartcards, defining a standard protocol used on the communications link between the card and the reader; the standard includes filesystem details and defines operations that can be performed on the card. The standard also sets a limit to the maximum supply current available to the card. These factors ensure compatability between different makes and models of cards and readers.

1.2 Contactless smartcards

In addition to traditional smartcards that are plugged into a slot on a reader, there now exist contactless smartcards. These cards are completely sealed with no external electrical connections. The card contains a coil that is energised by a modulated magnetic field generated by the reader; the magnetic field supplies

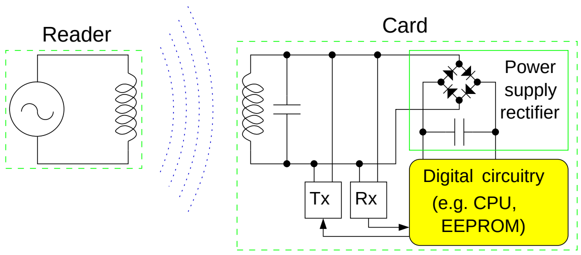

Figure 1.1: Contactless smartcard operation

power and supports communication with the card across an area of space. The most common type of contactless smartcard is the *Proximity* card which operates over a distance of 0 to about 10cm (whose specifications are defined in ISO14443). Other types include *Close-coupled* cards that operate over distances of 0-2mm (abandoned in favour of proximity cards) and *Vicinity* cards that operate over distances up to about 1m. Vicinity cards are simple circuits providing Radio Frequency Identification (RFID), storing a serial number (similar to a barcode) for tracking and identification of packages and objects in warehouses and production lines.

Figure 1.1 shows the functional parts of a typical proximity card. A loosely-coupled transformer is set up by the coil in the reader and a coil in the card. The reader coil is energised by a 13.56MHz carrier wave; when the card is within the magnetic field a current is induced in the card's coil and the tuned circuit formed by the coil and a capacitor resonates.

This varying field serves two purposes. Firstly, the energy received from the field by the coil is rectified and smoothed to provide a power supply for the circuits on the card. Secondly, by varying the field intensity as a function of time, data can be transmitted between the reader and the card. This process means the card is not required to touch or be plugged into anything, merely positioned in the proximity of the reader's field. Because the card is completely sealed and there is no physical contact, wear is reduced making the cards more reliable than contacted and magnetic-stripe cards. The fact that the card needs only be passed through a field also provides potential for increasing the speed of transactions. Due to this, contactless cards are becoming increasingly common in transit applications, among others. The Seoul Bus Association currently employs 6 million proximity contactless cards, enabling passengers to move quickly through turnstyles. This increased convenience is the main reason for the growing popularity of contactless cards.

1.2.1 Contactless smartcard communication

The ISO14443 type B standard specifies a 10% Amplitude Shift Key (ASK) communication path from the reader to the card, at 106kbit/s (1/128th of the carrier frequency); the 13.56MHz carrier is amplitude modulated with the data. 10% ASK means that the carrier amplitude when sending a '1' is 10% larger than nominal, and when sending a '0' it is 10% smaller (figure 1.2). The carrier is always present, ensuring the IC has a constant power supply. ISO14443 type A (MIFARE[Phi98]) specifies 100% ASK, whereby the carrier is switched off completely for the duration of the transmitted bits (so requires larger capacitors to smooth out the dips in the power supply). A circuit connected to the card's coil samples the field, extracts the received data from the signal and supplies it to the digital circuit.

Data is transmitted from the card to the reader by means of capacitive load modulation; modulation capacitors are shunted across the coil with an 847KHz carrier frequency when data is to be transmitted, to vary the resonance of the tuned circuit. The card reader can detect the change in its field, and extract the transmitted data. In many applications, however, the magnetic field will not be populated by only one card at a time. For instance several people at an underground station turnstyle may be trying to pass by the same reader. To ensure correct transactions for each card without interference between them, an anti-collision protocol is used to 'wake up' (or energise) all cards in the field, but select and complete transactions with cards one at a time.

Because the duration of a transaction must often be rather short (around 200ms) there may sometimes be the danger that the card has been moved away from the field before the transaction completes (called a 'tear'). This can be a problem if the data on the card is left in an incomplete state from only a part

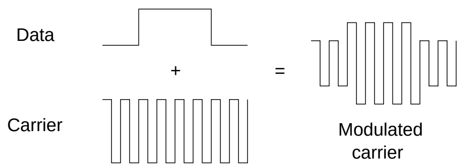

Figure 1.2: 10% Amplitude Shift Keying

of an update being performed. To avoid tears, the card must be able to process a transaction as fast as possible, reducing the chance of moving out of the field before completion. In the event of an abnormally short period of time in the field that *does* cause a tear, transactions may be rolled-back to a state before the tear, ensuring data integrity.

1.2.2 Power supply issues

A contactless card's circuits are powered directly from the current induced in the card's coil by the reader's magnetic field. The level of power available from the field is inversely proportional to the cube of the distance of the card from the reader, and most cards are designed to work several centimetres away. This would not be a problem if one could make the field as strong as possible; however, if it is too strong the field can cause interference. The interference may be magnetic, e.g. a strong field will erase magnetic-stripe cards nearby and may damage watches or other equipment. Also, because the field is modulated, carrier sidebands can cause Radio Frequency Interference (RFI). Typically the maximum range of a card and the power available for its operation will depend upon the maximum level at which the RFI is still acceptable by emission laws.

Consequently, the actual peak power available for the card's IC is low, around 10mW. This places tight demands on the circuitry on the card to minimise power consumption. Moveover, the IC cannot simply store up the energy received on a capacitor and then use it at a higher power at some other time because the IC relies on the field for its communication too; the transaction must be processed and completed whilst the card is still within the field so that the communication to finalise the transaction can still be performed. If a tear were to occur, and the card is moved from the field prematurely, the transaction would not complete and would have to be started again.

An IC that runs at low power is required, but the transaction time is also severely limited; a maximum duration of about 200ms is the goal for most commercial proximity cards. If the transaction was to overrun it may cause annoyance for a human user having to pass the card through the field again; worse, if the card is part of a moving object on an assembly line, there may be no second chance. The IC must therefore perform the operations of the transaction fast enough to complete within the maximum time; there is also an increasing demand for performance, reduced transaction times, etc. However the IC must also run from the low level of power received from the field.

The current induced in the coil of the card as it is moved through the field will vary from zero up to a peak, and back to zero; the power supply is therefore also not constant for a moving card. The regulated supply voltage for the IC will vary over a range of a couple of volts, and the IC must be resilient enough to function correctly and efficiently on a varying supply.

1.3 Contactless futures

Because the IC in contactless cards is required to run from an 'unstable' power and communications link in a very power-efficient manner, most contemporary cards are comparatively simple. Many contactless cards contain only a small amount of memory and a simple state-machine control circuit. More exotic ones may include a cryptographic processor and some contain small 8-bit CPUs.

Many current contactless cards are primitive compared with contacted counterparts, merely containing a few kilobits of memory with simple control. The family of MIFARE [Phi98] contactless smartcard ICs from Philips mostly comprises memory cards, with one high-end card utilising an 8-bit microcontroller with a cryptographic coprocessor.

As new smartcard applications emerge, more advanced cards will be sought with larger memories and higher-performance CPUs. This trend has already been seen in contacted cards that have moved from simple memory-based cards to cards containing cryptographic processors and 16 or even 32-bit CPUs. It can be expected that similar advances will be sought after in contactless cards as applications require more advanced functionality. The low power budget holds contactless card technology behind contacted cards; current 32-bit processors are too power-hungry for use with the low levels of power available on contactless cards. The challenge exists to reduce the power consumption of traditionally power-hungry high performance 32-bit processors so that one day contactless cards may provide more advanced functionality.

Chapter 2

Introduction to low power

Contactless smartcards have stringent power limitations, taking their supply from a low-power magnetic field. The field can only supply a low peak power to the card, and many high-performance CPUs are too power-hungry for the task. Here we examine where the energy goes and what can be done to reduce the power consumption and increase energy efficiency.

2.1 Power and energy

Power is measured in watts and defined as the amount of energy (in joules) used or transferred per second, i.e. the rate of energy use. A given CPU may have a low rate of energy use (it has a low draw of power), but there is the question of how much energy it requires to perform an operation (e.g. execute an instruction). Clearly a CPU that requires less total energy to perform an operation can get more useful work done for a given amount of energy. This is the energy efficiency of the CPU. The term 'low-power' is often used to refer to circuits that are energy efficient, rather than those that may use a lot of energy but over a broad stretch of time (thus technically using a low level of power).

2.1.1 Where is energy used in CMOS digital circuits?

The vast majority of digital circuits these days are implemented in Complementary Metal Oxide Semiconductor (CMOS) technology. The output of a logic gate is a function of the gate's inputs and driven to the correct state with combinations of *complementary* N-channel and P-channel Field Effect Transistors, or

FETs. Figure 2.1 shows the construction of a CMOS NAND gate. When both inputs A and B are at logic 1 (or Vdd, or 'high'), NFETs 2 and 3 are activated, 'connecting' the output, N, to the ground rail. If input A were to transition to 0 (or 0V, or 'low'), this connection is broken as NFET 3 turns off; PFET 4 turns on, connecting output N to the Vdd supply rail (thus the output goes to 1). For every combination of inputs to a static CMOS gate, the output is connected to either low or high supply rails and thus in a defined state. In this example N is connected to the supply, Vdd, when A AND B are not both 1, and N is connected to ground when both A AND B are 1.

Current does not flow through the structure from the inputs because the load of the FETs' gates is capacitive only. The field effect of the inputs being charged to 0 volts or, say, 5 volts is required to switch FETs on. If output N is driving an input of another CMOS gate and is at logic 0, no current flows between the 0V rail and the second CMOS gate because the input is already discharged to 0 volts. Likewise if the output were at logic 1, the input of the next gate would be charged to 5 volts (for a system running on a 5 volt supply), and no current would flow. In CMOS circuits a current will flow when a node is driven from one state to the other. In the NAND gate example, when the output is 0 and transitions to 1 (for example by PFET 4 conducting) node N and all of the subsequent input wires it is connected to must be raised from a potential of 0 volts up to the supply voltage. Current flows from the higher potential Vdd into node N, raising its potential to Vdd. Moving N from 1 to 0 is the same, a current flowing from the higher-potential N down to ground.

The output of a gate, the wires connecting the gate to other inputs, and the

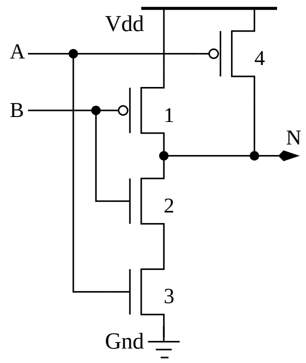

Figure 2.1: A CMOS NAND gate

inputs of subsequent gates impose a capacitive load. The amount of charge that flows into and out of a node when it is switched to 0 or 1 is given by the product of this capacitance and the voltage to which it is charged, i.e.:

$$Q = CV \tag{2.1}$$

The energy required to move this charge through a potential of V volts is given by:

$$E = QV \tag{2.2}$$

Thus it follows that the energy required to cycle a node (capacitance C) through voltage V is given by:

$$E = CV^2 \tag{2.3}$$

In CMOS circuits, current flows when a node is charged or discharged. It is also possible that when a node is between 0 and 1, overlapping thresholds of the PFET and NFET transistors may mean that the pull-up stack and pull-down stack transistors are simultaneously enabled for an instant, creating a direct path from Vdd to ground, and a current flow. Substrate leakage can also cause a small current. However the dominant factor in CMOS energy use is the energy used in transitions [CSB92]. CMOS has historically been regarded as a low-power technology compared to TTL and ECL whose logic levels are based on active current flow, because a CMOS circuit only uses appreciable amounts of energy when it is actually active and transitions are occuring.

2.1.2 Power consumption

When a CPU performs an operation, a number of transitions will occur, using a discrete amount of energy. The power consumption of the CPU (i.e. the rate of energy use) thus is dependent on the rate at which operations are performed. A measure of this is often the clock rate of a synchronous CPU, since all activity across the circuit is synchronised to happen on the edge of the clock. The power consumption, P, is

$$P \propto fCV^2 \tag{2.4}$$

where C is the total capacitance in the circuit, switched through voltage V at a frequency f. One can reduce the peak power consumption of a CPU by lowering its operating frequency. If f is halved, P is also halved. However it will still take, say, 2 million cycles for a particular program to execute, and because the cycles are now twice as long the execution time will double. Also, since the same transitions will occur for the same number of cycles, the amount of energy required per operation is unaffected and the program will require the same amount of energy to execute.

One can make a complex CPU run on 1mW of power by slowing it down enough, but it is no use using such a CPU in a contactless smartcard if it is then so slow that it takes a minute to process a transaction. The solution is to increase the energy efficiency of the CPU so that it can perform the required amount of work at an adequate rate with reduced power consumption. That is, a CPU with a greater number of MIPS per watt is sought. $1/(MIPS/W)$ is a measure of the average energy required to execute an instruction.

In the next section we will look at techniques that can reduce the energy required to execute an instruction, thus increasing the energy efficiency.

2.2 Reducing energy use in CMOS circuits

We saw in the previous section that the energy required to perform an operation is given by $E = CV^2$, where C is the combined capacitance of all of the nodes cycled by the operation. The energy per transition can be reduced by reducing C and V. Additionally, reducing the number of transitions in an operation will reduce its total energy.

2.2.1 Reduce transitions

The most obvious idea for reducing the energy of a circuit is to reduce its activity. This is a very good design policy but there may be tradeoffs. For instance, strategic placement of latches can avoid spurious transitions on the inputs of combinatorial logic blocks (which may propagate far along the circuit), but latches themselves consume significant amounts of energy in operation.

Another option is dynamic logic. In dynamic logic an output node is precharged and conditionally discharged by the function only if necessary. Dynamic logic has distinct advantages over static logic in that it can reduce spurious transitions as the nodes in the circuit do not transition between the power rails while they evaluate, as some static designs may. This can substantially reduce glitches on the outputs of a block of logic [CSB92]. Since there is no complementary transistor stack, the capacitive load of the nodes is reduced, there are fewer overall transistors, and 'short-circuit' leakage is elimintated. The disadvantages include the necessity to precharge the whole circuit which in some situations is often then immediately discharged upon evaluation. Large buffers may also be needed for the precharge phase. Because the charge will leak away with time, dynamic circuits are more sensitive to being run at different speeds; a lower speed limit is imposed, whereas the speed of static designs can be reduced without problems.

Complementary Pass transistor Logic (CPL) is an attractive option to reducing the number of transistors of many logic functions (and thus C). For example an XOR gate can be implemented with only 2 pass transistors, allowing smaller and simpler circuits. However CPL circuits provide no gain on the output; the reduced drive results in slower operation at lower supply voltages. When a CPL circuit drives a CMOS input the high state is not quite Vdd and the transistors in the CMOS stack may not be fully switched off in this stable state, which can increase the direct-path leakage current.

2.2.2 Reduce circuit size

The switched capacitance of a circuit is dependent on the number and size of transistors and connections between them. Reducing the feature size of the silicon technology will reduce the capacitance of the transistor gates being switched, and therefore the energy per transition, up to a point. Feature sizes are falling fast, with cutting edge fabrication plants approaching $0.1\mu m$ technologies. However, as transistors are getting smaller (and having lower capacitances) the wires connecting them are not scaling at the same rate. As the wires get closer together (and taller) their capacitance is becoming relatively high compared to the gates driving them, adding extra design problems. At less aggressive technologies the benefits are more straightforward; moving the technology of a design from $1\mu m$ to $0.35\mu m$ will certainly reduce the capacitance of the circuits and require proportionally less energy to switch them through a given voltage. However, a smaller capacitance in the same voltage will switch much faster, offering opportunities for increased performance for the same switching power – since $P \propto fCV^2$, if C is reduced then f can increase with no change in P. On the other hand, one may take advantage of the faster switching of the reduced capacitance to achieve the same switching speed at a lower voltage. Since a reduction in V gives a quadratic reduction in E (as $E = CV^2$), large reductions in the energy per operation are possible with smaller silicon feature sizes. Silicon technology is, predictably, getting smaller so the energy use of circuits will decrease. Although reductions in feature size (and therefore C) can be expected of the future, it should not preclude the development of other interesting schemes that can improve efficiency on whatever technology is *current*; it is no good trying to improve the efficiency of a design by waiting two years for feature sizes to shrink.

2.2.3 Reduce supply voltage

One of the strongest driving forces in reducing power consumption of processors has come from mobile computing applications. Handheld or portable devices are getting smaller and lighter but battery technology is not advancing at the same rate. Users expect small batteries to have a long life, and this has driven improvements in energy efficiency. Many high-performance CPUs in mobile computing applications have improved their power efficiency by allowing the CPU into 'sleep states' that slow or sometimes stop the clock to reduce the power consumed (for example Intel SpeedStep™). In mobile computers the CPU is not utilised fully for a large proportion of the time; if there is not enough work for the processor to do it spends the rest of its time running an idle loop. With this in mind, software can detect this and instruct the hardware to reduce the effective operating speed of the processor, for example from 600MHz to 300MHz. Halving the frequency, f, will cause a 50% reduction in the operating power P drawn.

Other CPUs (for example the Transmeta Crusoe processor [Fle01]) increase their energy efficiency further with a technique called Dynamic Voltage Scaling, or DVS. The idea is the same, to reduce the processing throughput when high performance is not needed. However, when the clock frequency is reduced, the supply voltage can also be reduced, increasing energy efficiency of the processor. The switching speed of the circuits will be reduced with lower voltage, but since the clock frequency is similarly reduced correct operation is maintained. The Transmeta Crusoe TM5600 CPU uses DVS (under the name of LongRun™) to scale the performance from 600MHz at 1.6 volts, down to 300MHz and below at 1.2 volts. By reducing both the frequency (a linear reduction in operating power) and the voltage (a quadratic reduction in operating power) at the same time, a cubic reduction in power consumption can be achieved. The voltage and clock can both be scaled in fine-grained steps to achieve the optimum performance.

The supply voltage can only be reduced so far, since there is a lower voltage limit that must be heeded for correct operation of any circuit. Also as the voltage swing narrows, the crossover between transistor thresholds gets broader and direct-path leakage current will increase. Reducing these thresholds will alleviate this problem, but noise immunity and substrate leakage will worsen.

Reducing the supply voltage and clock rate where possible can provide great benefits, but they do not come for free. A power supply capable of regulating its output voltage to varying levels is reasonably complex in itself, and for very small microprocessors may be so complex that it would use up more energy than would be saved by using DVS. Since contactless smartcards operate in a varying magnetic field, the supply voltage available is already varying over the range of perhaps one volt and the IC cannot demand whichever supply voltage it requires. The smartcard IC is required to run correctly from this varying supply voltage and, since the transaction time is important, get as much useful work done as possible and as efficiently as possible. Conventional synchronous processors do not easily run from a widely varying supply voltage. If the supply is too low for the circuits to switch fast enough to keep up with the clock the CPU cannot operate and must either wait for the supply voltage to rise, or abort the transaction. Conversely, if the available voltage is higher than usual, a synchronous CPU's circuits will switch faster but the performance will be held back by the clock. Contactless smartcards are an application where adapting to the power supply is vitally important – if any extra power is available then the processor should burn it doing useful work while it can.

2.3 Asynchronous logic

The activity in conventional synchronous circuits is synchronised across the system by a global clock pulse. Blocks of logic are given data and set to work by a clock edge, and are expected to have completed by the next edge when they pass on their result to another circuit down the line.

In contrast, asynchronous logic [BD95] blocks communicate with each other and coordinate actions by *request* and *acknowledge* handshakes. A block is passed data with a request for it to do something; when the data has been accepted it is acknowledged and the sender can continue with something else. Likewise, when a block has completed its work it presents the data to the next block and requests

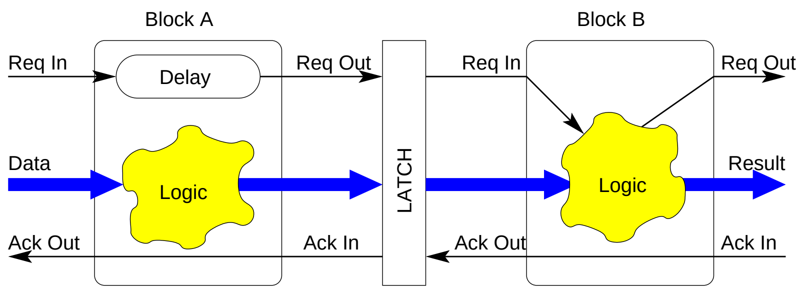

Figure 2.2: Bundled data pipeline

it to accept it, waiting for its acknowledgement (a *push* protocol is used). Or, blocks may request that they be sent data by another block which acknowledges the request when data arrives (using a *pull* protocol). The underlying notion is that each block times itself and is not dependent on any other block.

There are a number of different circuit styles that can be used in the design of asynchronous circuits, some of which complement each other and can be mixed and others that are distinct.

2.3.1 Bundled data

Bundled data communication from a sender to a receiver consists of a bus carrying data (usually in a two's-complement form) bundled with one request wire which signals that the sender functional block has completed its task and the output data bus is all valid. The request then tells the next block that it can latch the data, and the bundle is acknowledged all at once. A simple bundled-data pipeline is shown in figure 2.2. Each data block must signal its completion by passing a request on, and this is usually accomplished by a bounded delay around the block (block A); the delay is matched to the worst-case delay of data through the block. Delay-matching for complex blocks can often be difficult, relying on simulation and validation to make sure the timing constraints are not violated. Fixed delays on blocks may appear quite similar to a synchronous pipeline where the throughput is equal to the throughput of the slowest block. However the delay matching in an asynchronous pipeline is very fine-grained; each block will have a different delay and conditional forks will create different paths, varying throughput which adapts to the functional blocks in use. Complex functions such as additions or multiplies may additionally detect early completion, and generate the request onwards (thus exhibiting data-dependent delays (block B).

There are also different handshaking protocols, examples of which are depicted in figure 2.3. The request and acknowledge events are signalled by a transition on the wire in the two-phase, or Non Return To Zero (NRTZ), protocol; in the four-phase protocol the handshakes are opened by a rising edge and completed by a falling edge, or Return To Zero (RTZ).

Although 2-phase signalling requires half the number of transitions of 4-phase per handshake, 2-phase control circuits are often more complex than those using the 4-phase protocol; most recent bundled-data circuits use 4-phase signalling. Since data in bundled-data is almost always in a two's-complement form, the transmission of N bits requires $N+2$ wires (including the req/ack pair). Transmitting random binary numbers across a bundle with 4-phase signalling will require on average $N/2+4$ transitions per cycle (including the handshake). One will find, however, that buses in digital circuits do not change greatly from one cycle to the next and the actual average number of bundled-data transitions may be slightly less than this.

2.3.2 Dual-rail

Dual-rail codes represent each bit on a bus with two wires. The pair encode two states for the bit of data and one symbol meaning 'no data'. This means that the request is implicit in each bit; a block waits for every pair in the bus to move from the 'no data' state and once all data bits have arrived it can process the data (a single acknowledge is sent for the whole bus). Some logic can synchronise itself to incoming data, ensuring correct operation as fast as the gates can manage.

Threshold logic gates are usually used in dual-rail circuits, but logic blocks can get more complex than with traditional binary data. Unfortunately logic for completion detection at block boundaries can also impose a speed and area

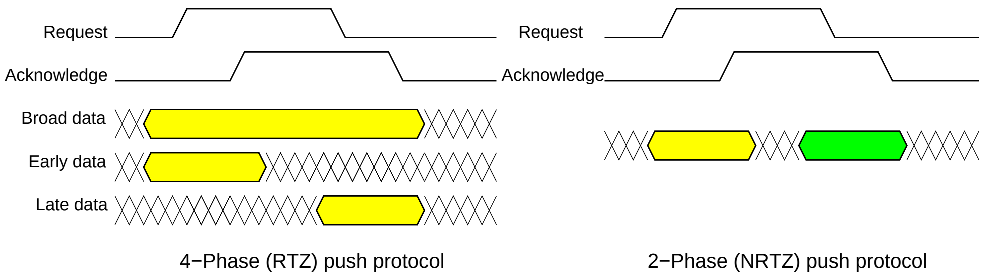

Figure 2.3: Two- and Four-phase bundled data handshake protocols

penalty.

Dual-rail circuits are usually much larger than bundled-data circuits; to transmit N bits, $2N + 1$ wires are required (including the acknowledge). Data can be transmitted with transition signalling (NRTZ), in which a transition is used on each pair to transmit a bit. Thus, $N + 1$ transitions occur. Completion detection logic is slow for transition signalling, and quicker for RTZ signalling where for each bit there is a $\texttt{0} \rightarrow \texttt{1} \rightarrow \texttt{0}$ on one wire of each pair. However in the RTZ dual-rail protocol, $2N + 1$ transitions occur transmitting N bits. This may be over four times as many transitions as a bundled-data scheme.

2.3.3 1-of-N codes

The idea of dual-rail asynchronous logic continues to 1-of-N codes in which data is represented by groups of N wires. In 1-of-4 coding, four wires encode 2 data bits. Again, data can be sent using NRTZ with one transition on one of the four wires, or with RTZ which performs a cycle on one of the wires to transmit 2 bits. Although logic functions using 1-of-4 coding are more complicated than for dual-rail (and thus much more complicated than bundled-data), 1-of-4 codes are useful because they reduce transitions compared to dual-rail. Using a RTZ signalling protocol, N bits are transmitted across $2N + 1$ wires with $N + 2$ transitions. A transition (NRTZ) protocol reduces the number of transitions to $N/2 + 1$, approximately the same number as with bundled-data.

Although 1-of-4 codes appear to have as good energy efficiency as bundled-data with the advantage of circuits being truly self-timed (avoiding possible complex validation of matched delays), they require significantly more complex logic for simple operations and have double the wiring cost. There is also the fact that in a bundled-data bus statistically the number of bits that change may be lower than the average (reducing the transitions per bit below $N/2$); with 1-of-4 NRTZ codes, transmitting N bits will *always* cause $N/2 + 1$ transitions. Indeed [LG01] shows bundled-data to use slightly less power than 1-of-4, and with significantly less complex logic.

2.3.4 Asynchronous versus synchronous design

The boundaries between blocks of asynchronous logic are very ‘pure’, with each block acting only when requested to and completing when it is ready. However, most design tools are not developed to cope with the issues of asynchronousity which may complicate the design of asynchronous circuits. Since every portion may run at a different speed to another, making sure that things happen in the right order, for example, can be a problem.

Synchronous design is often much easier in the timing sense, since a logic block knows that it will get its inputs on a clock edge and the order of events; all the block needs to do is prepare its result for the next clock edge. However this is a two-edged sword and a difficulty of synchronous design is the same fact that everything must be ready in time for the next edge, when data is to be passed on. If one block of logic is much more complex than its neighbor, it still needs to have completed by the next clock edge. Very complicated blocks of logic are a problem and much optimisation is required to make sure that the function completes on time. One cannot just extend the clock period for a complex section because that will affect the performance of the whole circuit. Blocks of logic, particularly arithmetic functions, may have varying completion times (e.g. 17×2 may complete quicker than 2189×177772) but must be heavily optimised so that the worst-case completion time meets the timing restrictions of the clock; this optimisation costs design time, chip area, and energy.

A self-timed logic block may take as long as it requires (performance permitting) to obtain a result; the recipient of the result will just wait for its handshake and this problem is avoided. For example, a multiplier will often be able to complete quickly, and the worst-case of multiplying two large numbers is statistically very rare. Consequently, asynchronous logic blocks can be designed and optimised for average-case scenarios and can often be much simpler. If, every 10,000 operations one operation takes twice the time, performance is not significantly affected. Fine-grained local timing can be achieved, with each block working at the optimum rate for any given data.

Another disadvantage with large synchronous designs is simply the fact that this global synchronising clock must theoretically arrive simultaneously at every point of the circuit. In practice this is very hard to achieve, and causes problems in the design. For example a block may be expecting data from a distant source at a particular clock edge; if the clock is skewed at the distant source, it may not have prepared the data in time (since it may have started later). This presents a challenge to designers, and significant effort has gone into minimising the effects of clock skew. Powerful clock buffers can minimise skew, but clock drivers alone can burn over 30% of a chip's energy use [GBJ98]. In the DEC Alpha 21064 [Dob92], the clock drivers sank peak currents of up to 10 amps into the clock tree. Distributing a global clock may not be in the interests of efficient energy use.

A global clock will also cause global activity across the circuit on every tick, even in parts of the circuit that may be unused or dormant. This effect can be reduced by gating the clock of disabled logic blocks; this is, however, a difficult design task and can only be achieved in a fairly coarse-grained manner. Asynchronous circuits by nature only do work if there is work to do. Activity is demand-driven by data moving through the circuit. Blocks are only active when and where they are needed, so dormant parts of the circuit have no activity and burn very little energy. This property can lead to major reductions in energy consumption in a variety of applications [vBBK$^+$94].

An inherent effect of self-timed CMOS circuits is that their speed can adapt to the supply voltage. If the supply voltage increases, the switching speed of the circuits increases; since the logic is free to complete whenever it is ready (and not restrained by a clock), the circuit will operate faster. At lower voltages there is no clock to fall foul of, so correct operation is maintained for longer as the circuit slows. Asynchronous circuits can therefore operate correctly over a wide voltage range, and adapt smoothly to their environment.

Adapting to a varying supply voltage is beneficial in contactless smartcards. An asynchronous circuit can always process data as fast as possible, getting optimum power efficiency from the supply at any one time. Philips has developed an asynchronous implementation of an 8-bit CPU for contactless smartcards [KKdBT00] which successfully adapts its performance to its varying supply. The CPU works at the fastest speed wherever possible, keeping the transaction time low. Philips' smartcard IC comprises an asynchronous 8051 microcontroller, 38KB of ROM, 2KB of RAM and 32KB EEPROM for persistent data storage. It also includes DES and RSA [oS77] [Lab01] cryptography co-processors to support encrypted transactions. The IC compared favourably with a similar synchronous 8051 implementation, and adapted well to the varying supply whilst using little average power.

In summary, asynchronous logic can provide great benefits in increasing the energy efficiency and lowering the power consumption of circuits since activity is demand-driven and only present where it is needed. Self-timed CMOS circuits can scale their own running speed to the power supply, automatically performing at the optimum rate.

2.4 High-level design considerations

The implementation of a design such as a processor is only part of the story. A badly designed and inefficient architecture will never be energy efficient no matter how many implementation tricks are used and how low a voltage it is run at.

The ARM architecture [Sea01] is widely regarded as an energy-efficient low power 32-bit embedded microprocessor, offering relatively high functionality for little energy use. ARM processors use a load/store RISC architecture, with an instruction set that is more complex than many simple RISC architectures (such as MIPS [KH92]). The instructions are generally more expressive and are all conditionally executed. This means that ARM code density is usually significantly better than that of most other RISC architectures [Jag90]. This has the effect that fewer instructions need to be fetched and decoded to execute a particular program, as each instruction can get a lot done, and reducing instruction fetches saves energy.

The architecture of the ARM7 is small and clean, with a simple 3-stage pipeline and no speculative execution, and remains the energy-efficient ARM processor core of choice. More contemporary ARM processors, such as the ARM9TDMI, have improved performance; 5-stage pipelines, branch prediction, etc. increase throughput, but at the cost of reduced energy efficiency.

The AMULET group at Manchester University have been involved with the development of a series of fully asynchronous implementations of the ARM architecture since 1991. The the latest implementation, the AMULET3 [FGG98], is a high-performance (100 MIPS) processor that couples the efficient ARM instruction set architecture with a very low-power asynchronous implementation. The AMULET3 provides very competitive power efficiency and a high ratio of MIPS per watt.

However, powerful 32-bit processors (even the AMULET3) still require too much power for use in small embedded applications such as contactless smartcards. Most of these processors already employ traditional low-power implementation tricks, so there is the question of how the energy efficiency of 32-bit processors can be further improved. One could further optimise the same CPU architecture at a low level, reducing glitches and scaling transistors so that every circuit operates quickly and efficiently; Efthymiou, Garside and Temple [EGT01] show several areas in the AMULET3 that could be improved, for instance. But, perhaps there can be other high-level design changes that may yeild energy savings. The next chapter considers some different architectures for low power.

Chapter 3

High-level design techniques

Previous chapters have shown that there are many techniques that can be used in the implementation of a design that may lower its energy consumption and power. Silicon technology and different logic styles may reduce the switched capacitance, and voltages may be tactically lowered in order to reduce the energy used to perform an operation in a circuit. When designing a very low energy system these implementation details may be assumed; the system must be optimal in every way, from the voltage it runs at and the types of logic it uses up to the high-level architecture. Low-level techniques to reduce transitions and energy use in a circuit should be used with complementary attempts to reduce complexity (in energy terms) from a higher-level architectural design view as well.

A simple example of high-level choices could be that of the implementation of a voice compression algorithm. The algorithm may require repetitive arithmetic operations on a block of data, and could easily be implemented using a general-purpose high-performance CPU. However, because a CPU is often designed with a wide range of target applications in mind it will usually have a generalised and flexible architecture. CPUs are rarely designed to run one particular type of program; instead they are designed to be efficiently adaptable to a range of programs. This means that this specific signal-processing voice compression algorithm may not match the host CPU's architecture and instruction set perfectly, and thus requires more instructions to describe the operation of the algorithm. This will be less efficient in terms of energy (transitions and complexity in the hardware) and execution throughput compared to an implementation of the system with a rather more specific Digital Signal Processor (DSP). DSPs are specifically optimised to run algorithms performing many repetitive arithmetic operations, and their instruction set may match the requirements of the compression algorithm better. However, a DSP is designed generally to run different types of signal-processing algorithms, and its architecture will reflect this required flexibility. A voice compression algorithm running on a DSP will therefore be similarly inefficient compared to an architecture specifically designed to process voice data and compress it (avoiding fetching and decoding instructions, just inherently processing one specific function).

A system's architecture should be tailored to its application or the algorithm it will implement, but it can also be tailored to the data set of the application. This partially accounts for the continuing popularity of old 8-bit CPUs. In many embedded systems, such as a washing machine or an engine management system the data the CPU will work on will be limited to, say, a handful of 6-bit analogue-to-digital readings, a few 8-bit outputs and so forth. If the application makes no use of numbers over 8-bits in magnitude, there is no sense using anything larger than an 8-bit architecture. In this way the architecture suits the data set, reducing chip size, complexity and power consumption.

However, the data in other applications is not limited to small-magnitude numbers and may often have to deal with complex worst-case scenarios. Kim, Zeisler and Papaefthymiou [KZP00] present an analysis of the MPEG-2 video compression system [PG95] whose data conversion rate depends on real-world input, the complexity and detail of the image data. They examine the Inverse Discrete Cosine Transform (IDCT) requirements (part of the MPEG decompression algorithm) and show that the worst case high bit rate requires a high IDCT throughput, although the average required throughput for real-world MPEG data is very much lower. Traditional decoder architectures that are capable of of decoding the higher MPEG standard bitrates are often heavily-pipelined; pipeline registers can dissipate a large amount of energy, making architectures designed only for high throughput inefficient at lower bit rates. Kim et al. [KZP00] present an implementation of an IDCT block whose pipeline depth can be changed dynamically depending on the data rate required, from a shallow pipeline for low-power low-bitrate decoding up to a high-performance pipeline required to decode the possible high bitrates. This improves efficiency by using a simpler pipeline the majority of the time (for low data rates where high performance is not required), removing the need to burn energy switching the pipeline registers.

In other applications the precision of the data's representation may not need

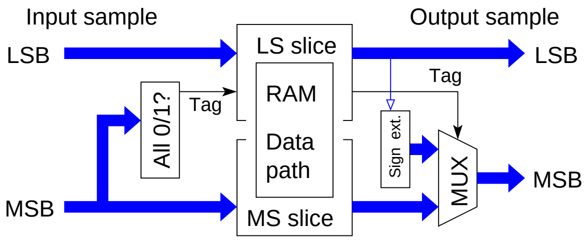

Figure 3.1: IFIR filter with conditional activation of top half [NS99]

to be as fine as the hardware expects. The IEEE-754 standard specifies formats for floating point numbers, and states that single-precision floating point numbers comprise 8 bits of exponent, 23 bits of mantissa and 1 sign bit. Floating point operations are complex and very power-hungry, and most embedded processors do not include floating-point units. Tong, Rutenbar and Nagle showed that in several benchmarks (mostly artificial intelligence-based) the accuracy of the result is affected very little by reducing the number of bits of mantissa calculated [TRN98]. The benchmarks provided similar results even when using only 5 bits of mantissa; calculating 5 instead of 23 bits represents a large energy saving. They present a digit-serial floating-point multiplier that is capable of performing multiplies 8, 16 or 24 bits wide depending on the accuracy required. As higher accuracy is required less frequently, the energy required is reduced. Energy savings are made when work that would normally be performed uselessly in traditional architectures is avoided and operations only performed when necessary.

Patterns in input data to a system can be exploited to reduce the number of operations performed, as in the previous examples. In [NS99], Nielsen and Sparsø describe an Interpolated Finite Impulse Response (IFIR) filter bank for a digital hearing aid. The system filters incoming samples in the frequency domain according to a set of coefficients describing the band-pass filter characteristics. They show that a majority of the time a hearing-aid samples low-amplitude sound (in a conversation 20%-50% of the time the filter is processing background noise due to pauses in speech) which means most of the samples processed are of a small magnitude. Since representing most of the samples requires less than half of the 16-bits designated per sample, Nielsen and Sparsø designed the filter datapath with a split so that it was partitioned into two 8-bit slices (figure 3.1. The most significant half of the datapath is conditionally disabled when the input sample is detected to have a magnitude of 8 bits with only sign-extension in the top half; this is the frequent case and activity is therefore often avoided in half of the datapath (which includes RAM and add-multiply-accumulate units). This slicing increased typical energy efficiency by 30%.

Systems working with digital signals often deal with zero-crossing data (i.e. part of a wave that moves from the positive to negative region or vice-versa); in two's-complement, negative numbers are represented with large sign-extensions of ones, so a small positive number changing to a small negative number will cause almost all of the bits in the word to transition from zeroes to ones. By storing the sign in a single bit, and the absolute magnitude separately (a sign-magnitude data representation), data that changes frequently between small positive to small negative numbers will cause many fewer transitions. A change of sign changes only one bit in sign-magnitude data, rather than inverting almost all of the bits in the word as in two's-complement data. Sign-magnitude representation is used in systems to reduce power needed to carry waveforms that frequently cross zero; however sign-magnitude often makes arithmetic functions significantly more complex making any energy efficiency advantages less clear-cut.

The dataset in systems that take sampled real-life audio and video as input is often correlated to a high degree, with many redundant operations performed in traditional fixed-size architectures. These correlations facilitate architectural optimisations that can prune the work performed in a data-dependent manner that can save energy when a full-width operation is not required. However at first sight, specific DSP filter applications and systems that sample real-world data may seem distinct from the data set seen in general programs run on general-purpose CPUs. 32-bit CPUs have a higher performance and throughput of data because they can process data in wider chunks than smaller 8-bit CPUs, with a lower instruction decoding overhead. Although many complex applications require the throughput of 32-bit CPUs, a piece of software will not *always* require 32-bit performance; consider loop counters and array offsets, whose magnitudes are rarely 32-bits wide and are used frequently in programs. For low-energy applications (and not just contactless smartcards), even simple CPU operations may use an appreciably large amount of energy. In the next chapter, we will discuss the extent to which a 32-bit processor architecture is under-utilised, and whether data-dependencies can be taken advantage of to reduce energy use.

Chapter 4

Splitting a processor's datapath

In the previous chapter we saw that programs processing real-world sampled data such a sound and video often have smaller magnitude data than the worst-case data size, and can conditionally avoid performing computations in a data-dependent manner. In this chapter we will examine the dataset used in general programs in a general-purpose 32-bit ARM processor. The ARM architecture is an efficient low-energy starting-point and ARM cores are popular in low-power embedded systems, with extensive software libraries, emulators, and compiler kits available. The AMULET3 is an asynchronous ARM implementation and may provide a good starting point for considering architectural changes; data-dependent architectures may have varying evaluation times and lend themselves to an asynchronous implementation. Asynchronous pipeline stages are 'elastic' in that they inherently accommodate variations in throughput. If one stage operates slowly, the rest of the stages will slow, waiting for it.

'Data', in a program, is reflected by the contents of memory and by the contents of the processor's registers. The registers contain the most active set of data, on which operations are performed with every instruction. A software model, written in 'C', was used to emulate an ARM processor. The state of the processor can easily be extracted at any point during the execution of a portion of code, for instance the register contents before and after the execution of a particular instruction can be examined to reveal the exact effect of its execution.

By monitoring the contents of the processor's registers, trends and patterns in the datasets of programs can be seen. If any patterns are discovered, they may suggest the use of a high-level optimisation such as the datapath split in Nielsen & Sparsø's hearing-aid filter [NS99].

Three small benchmarks were used in this study. They were all written in 'C' and compiled using the Norcroft 'C' compiler from ARM Ltd. The compiler generates good assembler, so should not have significantly worse properties from average hand-written assembler. Also, considering the fact that most commercial code is compiled from high-level languages to reduce development time it is quite reasonable to solely examine benchmarks written in 'C'. The three benchmarks used were:

Dhrystone – A small synthetic benchmark, designed to use a 'typical' spread of instructions.

DES encode – An implementation an encoder for the Data Encryption Standard [oS77] cryptography algorithm.

GSM filter – A signal-processing oriented algorithm for the decode filter part of the GSM speech compression system.

4.1 Operand Size

To examine whether data (operands of instructions) do not always make full use of the 32-bit registers and datapath, the 'length' of the operands was counted in the benchmarks. Operands of a small magnitude will be, in two's-complement, a set of 'useful' data bits at the low end of the data word followed by many bits of sign extension (i.e. all zeroes or all ones). Cumulative frequency plots of how likely it is for a particular bit to be part of the 'useful' data rather than just sign-extension are shown in figures 4.1, 4.2 and 4.3 for Dhrystone, DES and GSM respectively. One can see that the lower (least significant) bits in the operands in all three benchmarks are much more likely to be part of the data in the word, and the higher a bit is the less likely it is to be part of 'data'. Each benchmark has its own properties for the average length of operands; for instance, because the GSM filter processes sound data as 16-bit samples, one can see a sharp cut-off point at around the 16th bit.

On each plot there are three curves representing the three possible operands to an instruction. For a two-operand ARM instruction (as the majority are) operand A is always a register but operand B may be a register, optionally shifted, (e.g. `SUB r0,r1,r2`) or may be an immediate constant (e.g. `SUB r0,r1,#4`). The B operand is separated into two plots for the register form and immediate form

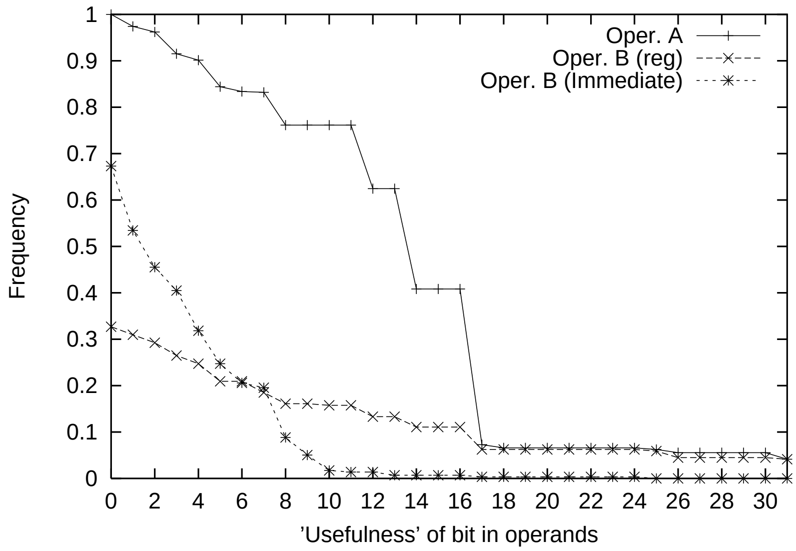

Figure 4.1: Frequency of bit 'use' in Dhrystone

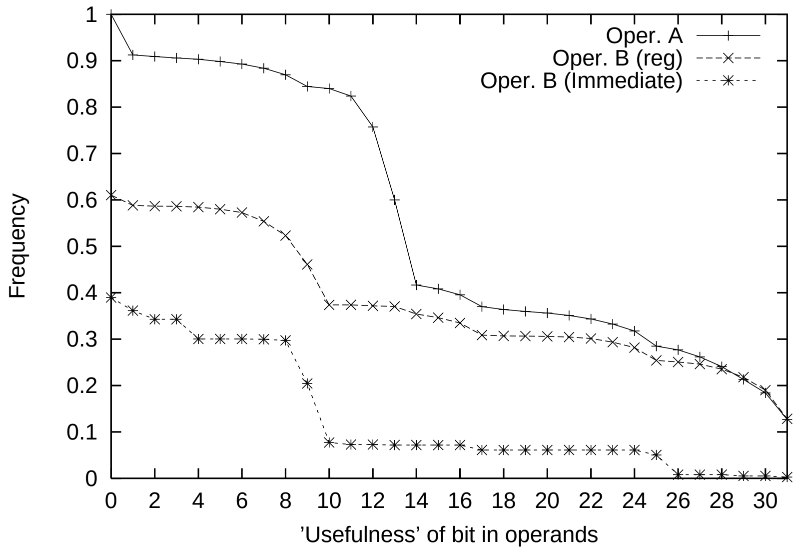

Figure 4.2: Frequency of bit 'use' in DES

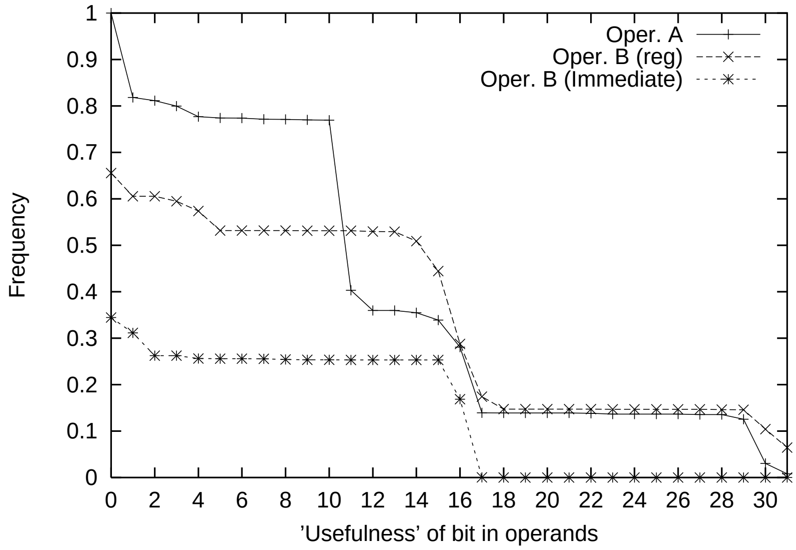

Figure 4.3: Frequency of bit 'use' in GSM filter

(the sum of these two lines represents all B operands). Immediate constants on the ARM are an 8-bit field rotated to any 2-bit boundary in the word; thus constants can have a wide range and are not limited to small numbers, but the data bits are confined to an 8-bit wide window (the rest of the word containing all zeroes). However, the graphs show that the majority of constants are in fact small. Indeed, Jaggar presents an analysis of the profiles of a suite of programs run on the ARM [Jag90] and shows that only 5% of immediates are greater than 255 (over an 8-bit magnitude). Since immediate constants make up 30–50% of instruction operands, there is more evidence of much of the width of operands not representing real data.

The height of the graphs over the lower bits suggests that most of the operands have small absolute values (the data is represented by few bits at the lower end of the word) and thus much of the execution of an instruction will be operating upon a large sign-extended portion. Since the sign extension is zeroes or ones the operations are often trivial, (e.g. `0 OR 1 = 1`) but performed on many bits causing many transitions and consuming energy unprofitably.

4.1.1 Instructions executed

The instruction breakdowns for these three benchmarks are shown in table 4.1. The DES encryption algorithm is based on logical operations and shifts, and one can see a large proportion of the benchmark's operations are logical instructions (e.g. `AND`, `EOR`, `ORR`); logical operations are comparatively rare in the other two benchmarks. Memory operations are frequent in all three benchmarks. Dhrystone and the GSM filter use many more arithmetic operations and branches than DES, and this seems to be a more typical spread of instructions; Jaggar [Jag90] showed that a variety of programs use few logical operations, about 22% branches, 20% arithmetic instructions (e.g. `ADD`, `SUB`, `CMP`), and about 36% memory loads and stores. Multiplications are usually rare, making up less than 1% of instructions; DES used no `MUL` instructions, in Dhrystone they made up 0.3% of the total. In the GSM filter about 7% of the instructions were multiplies; this is higher because the audio filter is a DSP algorithm, which comprises of many arithmetic and multiply operations. For the purposes of profiling data and operations for a low-power processor, multiplies are complex and rare so can be treated as a special case. Tong et al. [TRN98] refer to a data-dependent multiplier that may be considered if a hardware implementation is deemed necessary.

Actual addition operations are very frequent in the processor's execution unit, since they are performed when executing arithmetic (`ADD`, `SUB`, `CMP` etc.) instructions, as well as load/store and branch instructions in the calculation of the target address. Additions are more costly than logical operations in terms of both energy and delay, so saving unnecessary add operations is therefore particularly important.

Type	DES	Dhrystone	GSM
Loads/Stores	24.0%	34.3%	12.0%
ADD/SUB/CMP	2.9%	25.4%	42.3%
MOV	18.0%	22.0%	24.9%
Logical ops	54.8%	6.5%	6.9%
B, BL	0.2%	10.6%	7.1%
Other	0.1%	1.2%	6.8%

Table 4.1: Instructions breakdown of three benchmarks

4.2 Splitting the datapath

Most operands consist of right-justified data in the lower bits of the word. When this data is used in the execution of an instruction, the whole width of the register is read (perhaps for only a small number of bits), and sent to the execution unit which blindly performs the instruction's operation on the data. It would make sense for the register bank to store metadata about each register so that the execution unit can be informed of which parts of its operands contain useful data. The Arithmetic Logic Unit (ALU) can calculate what it needs to and then stop somewhere in the middle having declared that the rest of the operation is easily inferrable, e.g. `000...+ 000...= 000...`. For example, the metadata could state how many bits of real data there are in an operation; the execution unit could gate off logic past this point and provide a way of determining the length of the result data. This is, however, likely to add a large amount of extra logic and complexity to the execution unit. Alternatively, a bit-serial approach could be used whereby each operand is calculated one bit at a time starting at the bottom, and finishing somewhere in the middle once all of the real data has been calculated (the empty top of the result being inferred); this suffers the drawback that bit-serial arithmetic is extremely slow. Varying the operation's width dynamically can therefore incur significant overheads. Varying an operation's cover would be accomplished more efficiently by sectioning the datapath of register bank and ALU sideways into a fixed number of discrete partitions. The ALU tries to infer the result for each partition, conditionally performing an operation only if it cannot infer the result (similar in principle to the IFIR filter [NS99]).

Data bits are usually clustered together in a local area of the word; this often occurs at the bottom of the word, but in some cases may be clustered at some other higher point. Immediate operands on the ARM, for example, can be a cluster of up to 8 bits anywhere in the word's width (e.g. `0x02f00000`) and data pointers often index from memory areas with large addresses. Pointers such as `0xf00001a4` are 32 bits *wide* but contain a lot of empty space as holes in the middle of the word; calculating the result from the bottom of the word up to a mid-point (whether on a bit-by-bit basis or rounded to a partition boundary) will still perform unnecessary operations. It therefore makes sense to allow any partition, regardless of its place in the word, to infer from empty data if it can; the concept of allowing 'holes' in the middle of a word is likely to catch more cases of empty data, allowing more operations to be skipped by being inferred (particularly with large immediate operands which by nature are empty space low down in the word).

4.2.1 Metadata

The decision as to whether the ALU need do work will usually rest upon whether it is faced with actual data, or 'empty' data (all zeroes or ones). The register file can store metadata about a partition's contents, which can be described as being in one of these three states:

Full, F(n) – The partition contains useful data n, e.g. `0x27`.

Empty ones, E(1) – All of the bits contained in the partition are `1`.

Empty zeroes, E(0) – All of the bits contained in the partition are `0`.

Thus, for example the number `0x00ff7fdc` stored in a register file split into four equal partitions would be represented as `E(0)`, `E(1)`, `F(7f)`, `F(dc)`. When an operation is to be performed in a partition, it may be that the states of the two operands can be used to infer what the result will be without any calculation having to be performed. We have seen that most higher bits in operands are just sign extension; this means that upper partitions will be *E0* or *E1* most of the time. For example, if an `OR` operation is to be performed and the operands in a partition are *E0* and *E0* the result will obviously also be all zeroes, or *E0*.

Table 4.2 shows all of the possible combinations of operand states for an add operation (each partition may of course receive a carry-in from a lower partition, and this also affects the result). The table shows whether an ALU operation needs to be performed, and if not what the result can be inferred to be. For example, if four equal partitions are used and the operation is `0x00 00 00 23 + 0xFF FF FF 1a`, then exactly one add would need to be performed, and the other three partitions would be 'inferred':

$$\begin{aligned}
F(23) + F(1a) &\rightarrow F(3d)\ \textit{Add}\\
E(0) + E(1) &\rightarrow E(1)\ \textit{Infer}\\
E(0) + E(1) &\rightarrow E(1)\ \textit{Infer}\\
E(0) + E(1) &\rightarrow E(1)\ \textit{Infer}
\end{aligned}$$

A		B	Carry-in	Result	ALU operation
0	+	0		0	*[Inferred - none]*
0	+	0	+ 1	F(..0001)	Add
0	+	1		1	*[Inferred - none]*
1	+	0		1	*[Inferred - none]*
0	+	1	+ 1	1 + Cout	*[Inferred - none]*
1	+	0	+ 1	1 + Cout	*[Inferred - none]*
1	+	1		F(..1110)	Add
1	+	1	+ 1	1 + Cout	*[Inferred - none]*
0	+	F(n)		F(n), i.e. B	Move
F(n)	+	0		F(n), i.e. A	Move
0	+	F(n)	+ 1	F(r)	Add
F(n)	+	0	+ 1	F(r)	Add
1	+	F(n)		F(r)	Add
F(n)	+	1		F(r)	Add
1	+	F(n)	+ 1	F(n), i.e. B, + Cout	Move
F(n)	+	1	+ 1	F(n), i.e. A, + Cout	Move
F(n)	+	F(m)		F(r), poss. Cout	Add
F(n)	+	F(m)	+ 1	F(r), poss. Cout	Add

n, m arbitrary 'Full' operands, r = add result.

Table 4.2: Rules for inferring result of an ADD operation

The result, `0xFF FF FF 3d`, is produced along with metadata stating that the upper three partitions are *Empty(1)* and the bottom one is *Full*. This metadata is written back to the register file along with the result for use next time the data is referenced.

Some of the inferrer rules still perform an ALU operation, albeit a simpler one – a move. In the operation `0xf3 21 a2 d3 + 0x00 00 FF 4a`, the following operations would occur:

$$\begin{aligned}
F(d3) + F(4a) &\rightarrow F(0d) + Cout, \textit{ [Add]} \\
Cin + F(a2) + E(1) &\rightarrow F(a2) + Cout, \textit{ [Move]} \\
F(21) + E(0) &\rightarrow F(21), \textit{ [Move]} \\
F(f3) + E(0) &\rightarrow F(f3), \textit{ [Move]}
\end{aligned}$$

In this instance, although every partition has an ALU operation performed, only one is an ADD. The *move* operation implementation in most ALUs is usually much quicker and simpler than an add (normally consisting of an OR, or a directly-multiplexed result). This is useful when one larger number is being added to a much smaller one, as is often the case when an `LDR` instruction increments a pointer for instance. Adding two small numbers results in most of the word being inferred and the add avoided, but even adding one small number to one large number still results in add operations being replaced with cheaper move operations.

So far we have considered inferring the result of arithmetic operations, but the same principle can be applied even more easily to logical operations. For example, an operation of `0x00 FF d7 a3` *AND* `0xFF FF FF 00` will result in the following operations:

$$\begin{aligned}
F(a3).E(0) &\rightarrow E(0) \textit{ Infer} \\
F(d7).E(1) &\rightarrow F(d7) \textit{ Move} \\
E(1).E(1) &\rightarrow E(1) \textit{ Infer} \\
E(0).E(1) &\rightarrow E(0) \textit{ Infer}
\end{aligned}$$

The act of inferring a result entails evaluating a function on a few bits of meta-data instead of performing a full (more complex) operation. The simplicity of inferring the result may save many transitions, and therefore energy, since the ALU is often left to perform trivial operations.

Another area where a split of the datapath may bring benefits is in the register file. Although there must be some additional size overhead to store additional metadata bits for each register, there is the possibility of avoiding writes back to registers in some circumstances. In most implementations of register files, a write is performed by energising a common (thus large with a high capacitive load) bus with the data and selecting the register to write with an enable line driving every bit cell (also with a high load). This is a costly operation from the energy point of view due to the large areas and therefore high loads involved.

Many results are likely to be able to be inferred and generate *Empty* results. Depending on the implementation, this may be only two bits of data that represents the whole partition (of 4, 8 or perhaps 16 bits); writing two bits means fewer data lines on the bus need to be driven, and an efficient register implementation may only enable the metadata cells for writing (reducing the load on the enable line that would otherwise go across the entire register width). Secondly, it may be possible to avoid writing some results (be they *Full* data or just metadata) back to the register bank altogether. Many instructions just update a register value; reading from a register, performing an operation, and writing the same register back. This can occur explicitly (e.g. an instruction such as `SUB r0,r0,#4`) or implicitly (e.g. `LDR r0,[r1],#4` which increments the base register `r1` by 4). If a large operand is updated by having a small number added to it, for example, chances are that the operand's new value will not differ much from the original; that is, most of the register's partitions will remain unchanged. If the instruction decoder notifies the execution unit that such an update is taking place, it can avoid generating an unchanged result in a partition (e.g. `74 + 0 = 74`, a *move* operation), and therefore also avoid writing the result back to the register file. This avoids activity not only in the ALU, but in any result latches, register writeback buses and associated control.

There may also be scope for a register file implementation that can avoid reading the whole width of the register when only metadata is required. For instance, a register partition could be selected and the metadata bits are driven onto the bus; then, only if the partition is marked *Full* is the rest of the data width driven onto the bus. This would require a small amount of extra control, but could reduce the energy required to both enable the read and drive the read data port bus.

4.3 Software simulation

It was shown earlier that most operands are small, that there may be a portion of unnecessary work and that this work may be replaced by simpler infer operations. In order to test the hypothesis that splitting the register file and execution unit into partitions will allow operations to be inferred, the 'C' ARM simulator was augmented with code to simulate executing instructions with arbitrary numbers of partitions. The inferrer rules for *ADD* (Table 4.2), *AND*, *OR* and *EOR* were added to the simulator's execution code, modelling a split-datapath implementation. When an instruction is executed, the exact operations that need to be performed with real data for the operands are counted and statistics for the number of operations performed or inferred collected. Thus, real ARM programs were run under simulations of a partitioned datapath and the effects of the inferrer rules discovered.

4.3.1 Operations performed versus those inferred

Table 4.3 shows results from benchmarks run with four simulated partitions. It shows the average workload of additions in terms of the number of partitions that needed an add to be performed (rather than just a move or the result being inferred). From this table it can be seen that it is rare that a full-width add operation must be performed and that in fact most of the result can be inferred. Dhrystone requires a full add operation only 3.9% of the time. The vast majority

Partitions added	Freq. in Dhrystone	Freq. in GSM	Freq. in DES
0 *(all inferred)*	14.7%	8.7%	11.6%
1	58.2%	78.6%	34.9%
2	23.3%	12.7%	53.4%
3	0%	0%	0%
4 *(all added)*	3.9%	0%	0%

Table 4.3: Operations required when adding two registers with 4 partitions

Partitions EORed	Frequency in DES
0 *(all inferred)*	13.4%
1	24.1%
2	27.9%
3	18.2%
4 *(all added)*	16.4%

Table 4.4: EOR operations required in DES with 4 partitions

of its adds require 50% or less (≤ 2 partitions) of the word to actually be added, and in 14.7% of adds the entire result avoids an add operation; executing instructions whose entire result avoids an addition this frequently may seem rather peculiar. This occurs when an instruction adds zero to a register, such as `LDR r0,[r1]`; in other words an offset of `0` is given from the base address `r1`, and a *Move* occurs instead.

The results of the GSM filter show that it never requires over half of the word to be added; the implementation of the algorithm predominantly operates on 16-bit `short int` datatypes, which affects the operand size. The DES benchmark, however, also appears to never require a full 32-bit operation; one may wonder why then it needs to run on a 32-bit CPU at all. The DES algorithm consists almost entirely of bitwise exclusive-or operations and comparatively few arithmetic operations, so table 4.3 does not highlight the computation in DES. Table 4.4 shows the frequency distribution of partitions that required an *EOR* operation to be performed in DES. A full 32-bit operation was performed in 16.4% of *EOR* instructions, though one can see that the majority of instructions did not require an operation over 50% of the width, and therefore many results were still inferred.

A factor that affects the benchmarks is the address at which code is run in the memory map. A high proportion of additions are performed to calculate branch target addresses and indirect memory addresses in load/store operations – 88% of the additions in DES are part of load/store instructions. If the code and data begins at address 0, clearly when calculating an array offset the base address will be a small value (with more empty partitions) and more of the calculation will be inferred. (This explains why DES requires few partitions to be added in an add operation.) Pointers to high-addressed memory locations will require more operations in address calculations associated with them.

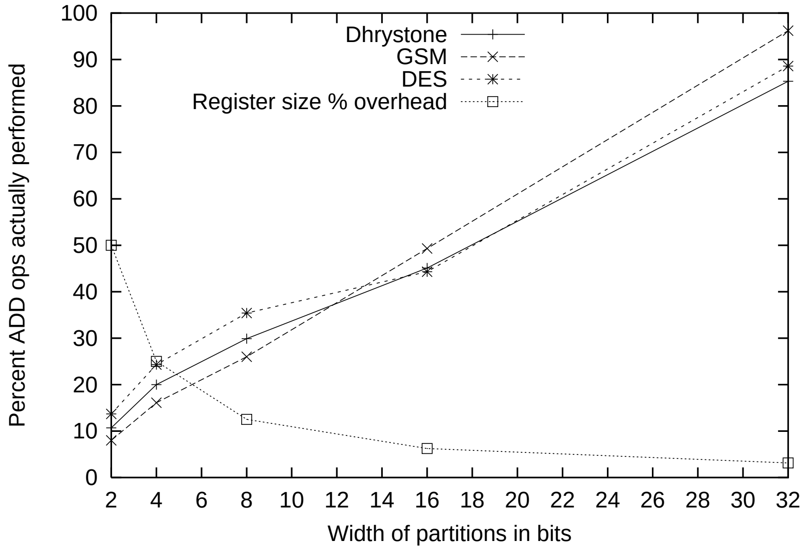

Figure 4.4: Adds performed/register overhead vs. partition width

4.3.2 Varying partition sizes

The simulations were run with different-sized partitions; for simplicity, all partitions were equally-sized. The benchmarks were run with sixteen 2-bit partitions, eight 4-bit partitions, four 8-bit partitions, two 16-bit partitions, and one 32-bit partition for comparison.

The savings gained from operations avoided by inferring the result vary depending on the sizes of the partitions. The smaller the partitions are, the finer-grained the representation of the operands and empty space will be; fewer partitions will be 'half-full' and more infer operations can take place. However there will be considerable control overhead if partitions are too small and, since an extra bit must be stored in the register file for each partition's metadata, the register file area will begin to become unruly. Large partitions, say two 16-bit partitions, will reduce the overhead but will save fewer adds since even tiny-magnitude operands will be 'rounded-up', appearing to be 16 bits long.

The tradeoff is shown with the Dhrystone, DES and GSM benchmarks in figure 4.4; the number of extra bits in the register file increases proportionally with the number of partitions, but the number of add operations that need to be performed also drops dramatically. With the datapath split into 8 or more partitions (i.e. 4-bits or narrower) the register bank is almost a quarter larger; the

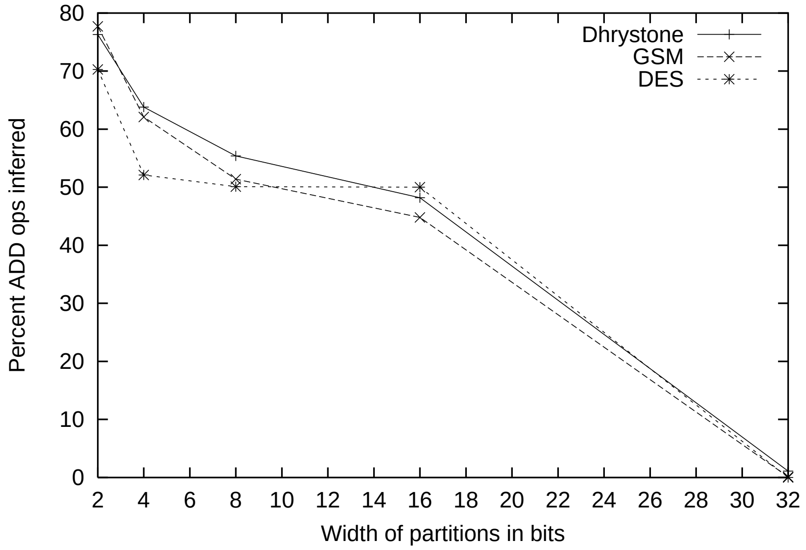

Figure 4.5: Adds operations inferred vs. partition width

overhead is more acceptable for four 8-bit partitions, and in this configuration an average of about 30% of add-partition operations are actually performed. With two 16-bit partitions, the register overhead is minimal and about 46% of the add operations are required. Other logical operations are also saved; even though DES has complex use of logicals (and therefore in theory fewest savings) it still avoided 50% of EOR operations and 80% of ANDs.

It appears that a partitioning scheme can avoid the majority of add operations in the ALU; however figure 4.4 just shows how many adds are performed and not what they are replaced with. When a partition avoids an add it may elect to perform a *Move* operation, or infer the result – clearly one wants as many operations to be inferred, avoiding datapath activity altogether. Figure 4.5 shows the proportion of 'avoided' adds that are inferred rather than performing an ALU *Move*. It shows that most of the avoided add operations do actually perform a simpler infer operation; on average over 50% of data is inferred with two or more partitions, although it also shows that four 8-bit partitions only infer an average of about 5% more data than two 16-bit partitions. However, figure 4.4 showed that two 16-bit partitions perform 16% more add operations. Thus there is still a clear advantage of 8-bit partitions over 16-bit since 8-bit partitions still replace a substantial portion of additions with moves.

Instructions that update the value of a register (e.g. `ADD r0,r0,#7`) do not

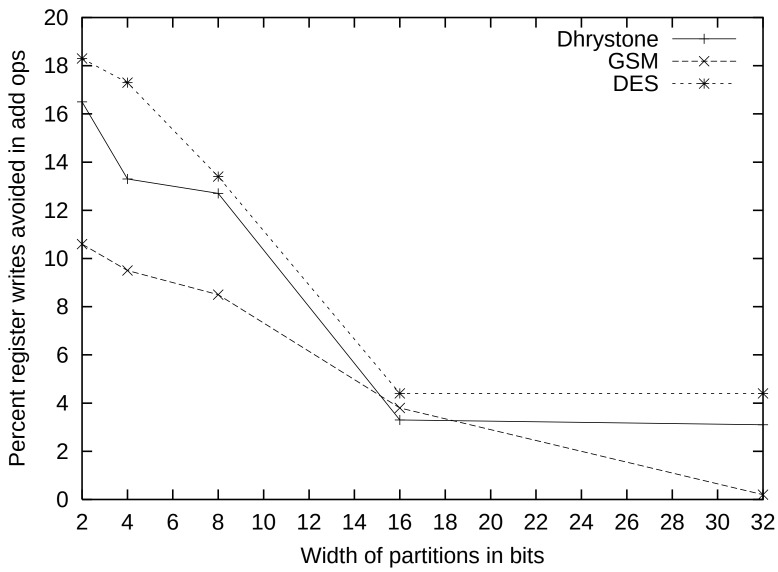

Figure 4.6: Percentages of update add ALU writes avoidable

need to write unchanged data back to the same register. A *Move* operation just copies data without any change, and in instructions updating a register, they can be avoided (saving power and time in avoiding both the register write and the ALU operation itself). It can also be inferred when metadata would be written back unchanged to also avoid these writes. Figure 4.6 plots the percentage of ALU operations in an add that are redundant *Move*s and can be skipped, against the partition width for the three benchmarks. As partition size decreases the finer-grained the decision to skip the write becomes and the savings increase, tending towards a limit. The fact that 8-bit partitions perform many more *Move*s than 16-bit partitions becomes more important because the percentage of avoidable *Move* operations and writes also increases with 8-bits. Therefore, even though four 8-bit partitions may not appear to infer significantly more of the results, fewer adds are performed by replacing them with *Move*s and many of the *Move* operations can also be avoided. The energy savings of avoiding adds and data writes to registers should be significant for four 8-bit partitions over two 16-bit partitions.

The simulation results suggest that there are data dependencies in general code on a 32-bit processor, and that a split-datapath allows a large proportion of additions to be avoided. Operations are either replaced with a cheaper *Move* operation or, the majority of the time, inferred completely (avoiding ALU activity

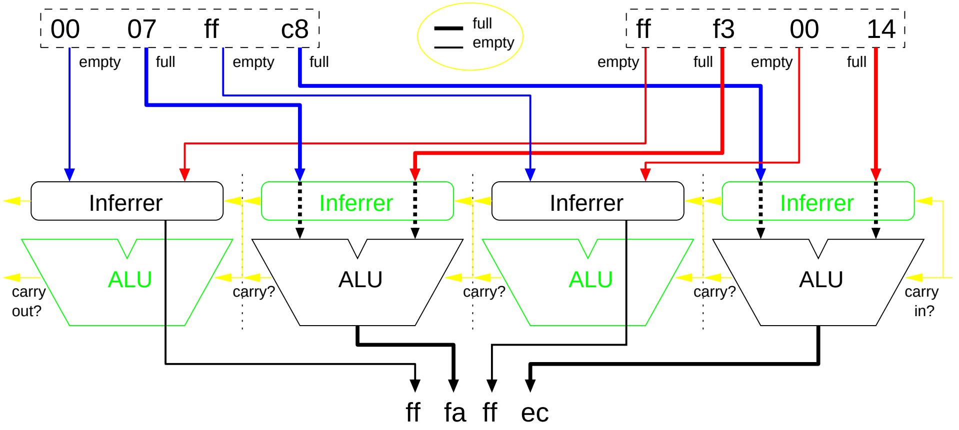

Figure 4.7: Splitting the ALU in parallel

altogether). The next section describes two ways in which a partitioning scheme could be implemented.

4.4 Architectures

The purpose of a datapath split is to make the operations performed dependent upon the data passing through the processor. Data-dependent operations will often require varying amounts of time to complete and the architecture may allow this fact to be exploited in a variable-length (or asynchronous) pipeline. If the operations required are fast, there is the potential that the rest of the pipeline can also speed up to match the execution unit, increasing throughput. There are several ways that the traditional execution datapath can be split to facilitate this 'skipping' of operations, and two are examined in this section.

4.4.1 Parallel

A normal 32-bit ALU could be partitioned sideways across the word (for example into quarters). The register bank supplies metadata to the ALU along with two 32-bit operands; each partition can look at the metadata of its operands and infer for itself whether to enable its ALU section. The operation's result is calculated in each partition in parallel. This architecture is shown in figure 4.7; each partition is represented as a 'mini ALU' and inferrer, and in each block the result is inferred or the ALU is enabled.

For example, figure 4.7 shows the operation `0x0007ffc8 + 0xfff30014`. The lowest partition (right-hand) must add `c8` and `14` so performs an ALU operation. The next partition along is just adding `ff` and `00` which it infers, and so on. Two ALU operations are performed in this example, and two infers save 50% of the operations.

An advantage of a parallel split is that the partitions need not be the same size; for example it may be found for a particular application that most of the data is not less than a 8-bit quantity; splitting as 16 bits at the bottom and two 8-bit partitions at the top would avoid unnecessary control overhead in the bottom two bytes.

The performance of evaluating partitions in parallel would be slightly less than a non-partitioned ALU due to the control overhead, although the same order of magnitude. The actual execution delay for the whole word would vary slightly, depending on the proportion of faster infer operations in a particular operation. This variation in execution time could be easily absorbed in the elasticity inherent in asynchronous pipelines which will adapt around the execution unit; however, the variation will be slight since each partition is dependent on its 'previous' (or lower) partition. The data word will skew diagonally slightly, but the pair of operands still arrives in a whole 'packet'. There could be some performance gain in pipelining each partition so that it can move onto another pair of operand partitions, but this is likely to be complex because of carry-in dependencies in additions.

Duplicating the inferrer logic for every partition makes operation simpler, but has the disadvantage that a parallel datapath split would have a significantly increased silicon area compared to a non-partitioned ALU. The extra metadata bits required would also have a size impact on the register file. Logic to save energy on reading metadata by only enabling the read on the metadata bits would also increase the size slightly.

4.4.2 Serial

Another approach is possible when execution is split into equal-sized partitions; the datapath could be made the width of one partition, and could perform several serial cycles per instruction. For example, with four 8-bit partitions the unit would do four cycles starting with the least-significant byte of the word. Each cycle would fetch the operands as either 1-byte chunks or metadata saying "This

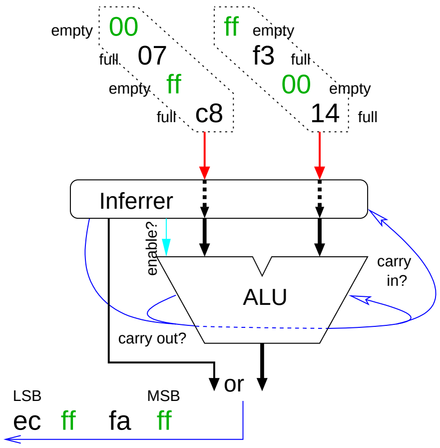

Figure 4.8: Splitting the ALU serially

partition is zeroes/ones" and an 'inferrer' would, if possible, perform the cycle and deduce the result. Otherwise, the ALU would take the cycle and perform the 8-bit operation on the data operands (figure 4.8). If an add operation is being performed, the carry-out from the previous cycle (or carry-in to the entire operation) is carried-in on the next cycle. The result would be generated one partition at a time, low byte to high byte.

A serial approach is more interesting from the asynchronous design point of view; with N partitions, N cycles would be performed executing an instruction but some of the cycles may be faster (inferrer operations) and some slower (ALU operations). Thus, not only could different instructions take different times to execute (as is the case in many asynchronous ALUs) but the same instruction would take anywhere between N short cycles to N longer cycles to execute depending on its operands. This feature would nicely exploit the elasticity inherent in asynchronous pipelines, whereas controlling timing in a synchronous implementation would be more complex and possibly less efficient.

At first sight a serial architecture may appear very slow; there will be N times as many cycles performed executing an instruction (where N is the number of partitions) plus some delay for control overhead. However this may be worthwhile if a large enough proportion of operations are avoided and implemented asynchronously the quicker inferrer cycles will speed up execution. Table 4.3 shows that few adds perform solely ALU cycles so the worst case is rare and the average case cycle time will be lower due to the high proportion of infer operations performed. Fig. 4.9 shows estimated average-case cycle times for the three

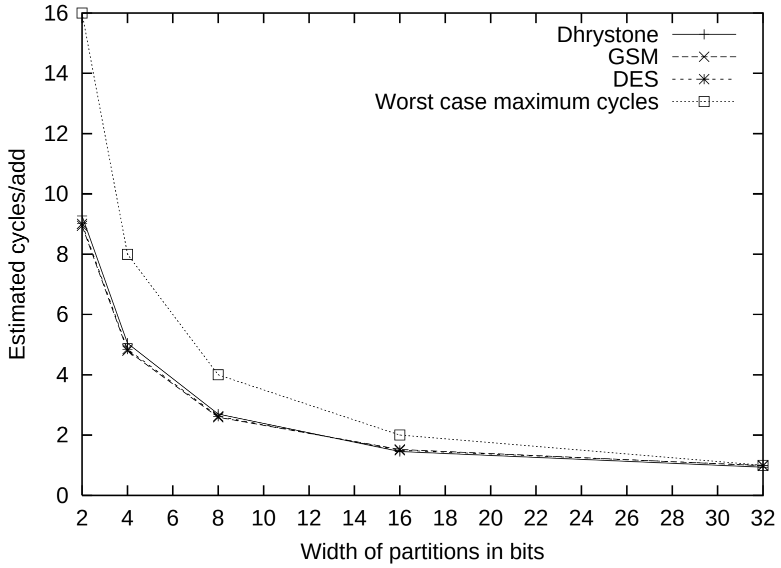

Figure 4.9: Estimated cycle times for a serially-split ALU

benchmarks, alongside the worst-case 'N slow cycles' performance. In the estimation an ALU cycle takes the same speed as a 32-bit add and an inferrer cycle is 0.5 times the speed; these are pessimistic approximations but the graph implies that average case performance is quite reasonable.

Using two 16-bit partitions means a minor decrease in performance (Fig. 4.9) over a non-partitioned ALU but only saves about 50% of additions. Four 8-bit partitions can save roughly 70% of adds, with a small performance hit and a small increase in register bank size. Eight 4-bit partitions can save many adds, but the number of extra cycles and the register bank overhead are unlikely to be justified.

There is also the possibility of pipelining the ALU and Inferrer blocks in a serial execution unit to increase throughput; the inferrer could be inferring the next few partitions whilst the ALU is busy performing an operation. The operation of the two blocks could overlap within and even between instructions (the inferrer could work ahead of the ALU as it is faster, and begin working on the next instruction along). Again, carry and operand dependencies add complexity to this approach, and the increased performance may not be justified.

It is difficult to predict the exact energy used with various partition sizes from the simulations. Figure 4.10 shows an estimation of the relative energy use of different partition sizes. This is calculated from the estimations that an n-bit

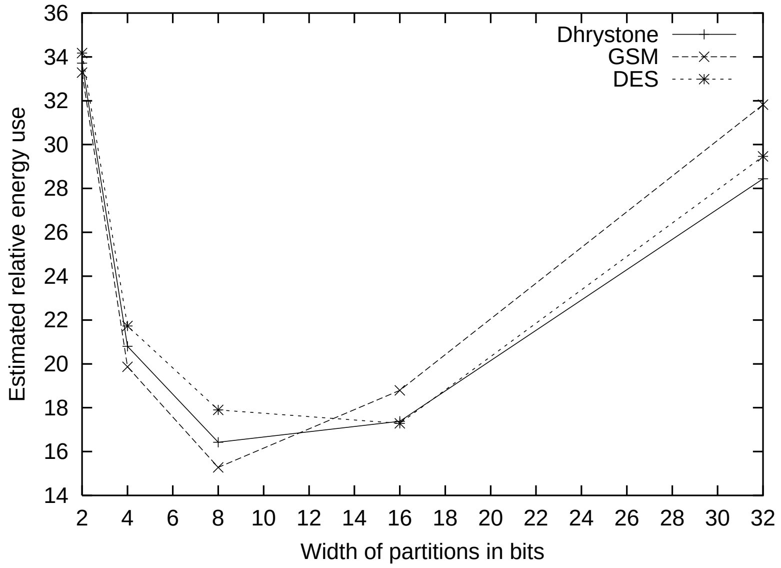

Figure 4.10: Estimated energy use for different partition widths ALU

addition may use roughly n units of energy compared to an infer operation that may use around 2 units of energy. This graph is clearly very approximated but it is useful to gain an idea of the tradeoffs present when varying partition sizes. Large partitions may use more energy performing additions (avoiding fewer in favour of inferred operations); as the partition size decreases many more cheaper inferred operations are performed and the energy use may fall. However, if partitions become too small then the energy use will rise again as the large number of inferrer operations now required will dominate.

An ALU with four 8-bit partitions appears to be a good compromise between register file size overhead and the number of add operations avoided; but, with a serial implementation it may achieve approximately 2.2 times worse performance than a normal ALU. However, lightweight energy-efficient embedded applications do not need a 100MIPS ARM processor; processing performance is important but if one achieves half of the speed for one third of the power consumption, there is a gain. A serial implementation trades performance for datapath size; an 8-bit serial ALU appears in simulations to be able to avoid a significant proportion of add operations (replacing most with a simple infer operation) and would reduce the datapath size to almost a quarter.

The graph of estimated energy use (figure 4.10) suggests that the lowest energy use may be achieved with partition widths of around 8-12 bits; of course with a parallel datapath split one may easily have two 11-bit partitions and one 10-bit partition (approximating three 11-bit partitions). Eleven-bit partitions avoid only slightly fewer additions than 8-bit partitions, however implementing an 11, 11, 10 split in a partition-serial manner would introduce extra logic to cope with a varying length partition. For this reason, using 8-bit partitions would be simpler and the graph suggests that this width is also around the lowest energy area.

To sum up, a partitioned register file and ALU scheme can avoid significant numbers of ALU add operations, replacing some with simple *Move* operations and most of the time inferring the result altogether. Four 8-bit partitions provide good savings with reasonable register file overheads; a serial implementation of an 8-bit partitioned ALU lends itself to an efficient asynchronous implementation with a moderate performance penalty but small size. There is also potential to save power by avoiding writing data back to the register file unnecessarily. However software simulations cannot give an accurate measure of energy efficiency and power savings; to quantitatively compare the actual power consumption of a split-datapath approach, an implementation of this system will be compared to a traditional 32-bit ALU. The implementation is described in the next chapter.

Chapter 5

Implementation

Software simulation of a split-datapath processor showed that many operations in the ALU could be avoided and results inferred instead in the majority of cases. It suggested that an implementation of a split-datapath would be useful in investigating energy savings from avoiding operations.

It is only appropriate to split some of the parts of the processor into partitions. The datapath in a modern ARM processor connects the register bank, the execution unit and the memory load/store unit so that data can flow between them. It is the data itself that the conditional activation of the partitions is based upon, since the data tends to contain sparsely-spread information; other parts of the processor such as those fetching and decoding instruction streams are not as appropriately split. Instruction bitmaps should be very dense, making the likelihood of fetching areas of *Empty* instruction very small; the decoding of instructions is also unlikely to have any benefit from being partitioned and the added complexity would be more of a hinderance than a help.

In order to process an instruction's data in a partitioned execution unit, the register file must be partitioned to supply metadata for each partition of the operands and accept a partitioned result. A register file will probably benefit from the small amount of extra control logic and the size increase because metadata reads will avoid reading the whole register width in most cases, and optimisations can be made when updating a register value to avoid register writes. The ideas behind a register file split into partitions are reasonably simple; these amount to the partition select circuits plus optimisations to prevent all of the bus lines from being energised when only metadata is being read from a partition. Because of this relative simplicity, the implementation of a partitioned register file is not as interesting as that of the rest of the datapath and since implementing a register file often requires a lot of hand optimisation and custom layout (a lengthy process), implementation time may be better spent elsewhere.

The memory load/store unit is the split-datapath's interface to external memory and could benefit from using partitions, especially if some sort of bespoke memory system stores the metadata alongside the data. This would allow full-width reads and writes to be avoided in a similar way to those in the register file, reducing transitions on the external buses. However this may require specialised memories and, again, the time taken to implement the control circuits involved is not justified given their relative simplicity.

The execution unit is significantly more complex in its operation than the rest of the processor. Chapter 4 showed that many ALU operations can be avoided by inferring the result, and the execution unit should therefore be the main area that would benefit from a datapath split. It is more interesting from the implementation perspective because of its complexity.

In the ARM, the execution unit comprises the shifter, the ALU and a multiplier. The multplier is used rarely (see section 4.1.1) and multipliers are often large, complex units; a split-datapath multiplier would be used too rarely to justify the effort required for its inclusion in the implementation. Tong et al. [TRN98] presented a byte-serial multiplier that may be appropriate for the byte-split datapath.

The shifter is in series with the ALU on the B operand port, and every operand on this port can be shifted. Performing shift operations on data that is coming from a partitioned register file in a byte-serial manner is a problem (for instance the scenario of shifting right, where the bottom byte will get data from higher bytes that have not been sent yet). The shifter would also have to deal with metadata in partitions, and ensure that the output metadata reflects the state of the result. It may be that a tighter coupling of the shifter and register file would solve this problem, with the operand only being serialised for the ALU upon exiting the shifter. A byte-serial shifter would clearly be more complex and perhaps slower than a standard shifter; however, shifted operands are used relatively infrequently (about 10% of operands [Jag90]). An uncommon slow case will not severely affect performance in asynchronous pipeline, as the pipeline speed will always adapt to run as fast as possible. Although a shifter that deals with metadata for a byte-serial ALU is an interesting problem, the implementation has been left for future work because of its sporadic use.

The ALU performs operations on every instruction executed, and simulation showed the attractive possibilities of avoiding operations by using an inferrer. Furthermore, operations with data-dependent completion times have promise for an efficient asynchronous implementation. In terms of power consumption, the ALU uses a significant portion (around 25% [EGT01]) of a processor's total; the effects of splitting a processor's datapath should be apparent from the effects of a split-datapath ALU.

An asynchronous byte-serial ALU was implemented, with interfaces to accept serialised operands with the decoded instruction operation and an interface to write results. This allows a comparison to be drawn between split-datapath execution and traditional monolithic execution units such as the AMULET3 ALU.

5.0.3 Asynchronous design choices

A four-phase bundled-data style was chosen for this implementation. Even though there is the problem of delay-matching with bundled-data circuits (that is avoided with dual-rail and 1-of-4), the energy efficiency and low wiring costs of bundled-data circuits make them a more attractive choice than dual-rail and 1-of-4. In addition, the logic design in bundled-data circuits can be considerably more straightforward than in dual-rail and 1-of-4; keeping things simple is a major factor in a prototype design within a limited time frame. The AMULET3 processor is implemented in the same four-phase bundled-data style, meaning that closer comparisons can be made between the AMULET3's monolithic ALU and the byte-serial split-datapath ALU.

5.0.4 Data representation

There are a number of ways to implement the metadata, encoding a partition's state into *Full*, *Empty zeroes* and *Empty ones*. The word in the register file could have two extra bits that specify the length of the data (that there is 1, 2, 3 or 4 bytes of data in the word), reflecting the amount of sign-extension present. This approach does not exploit 'holes' of empty data that may be present in the middle of the word (e.g. in `0xf200000c`) and thus may cause unnecessary work on operands that are large in magnitude but contain little data (e.g. immediates).

Alternatively one could add two extra bits to each 8-bit partition to specify its state independently from its neighbours: one bit to determine whether the partition is *Empty* or *Full*. If *Empty*, the other bit would differentiate between all zeroes and all ones. This requires two extra bits (a 25% overhead) per partition but allows 'holes' in the middle of a word to be represented as *Empty* and more work avoided. However this representation is somewhat redundant, and it was decided to represent partitions as an 8-bit two's complement binary number (D[7:0]) with one extra bit, F, per partition. When $F = 1$ the partition is marked as *Full* and D[7:0] contain useful data. When $F = 0$ the partition is *Empty*, and bit D[7] differentiates between *Empty 1* and *0* when set to 1 or 0 respectively. Thus a partition in an operand is 9 bits long (a 12.5% increase in size).

5.1 Design of the byte-serial ALU

This section will give a simple overview of the byte-serial asynchronous ALU; full schematics of the implementation are included in Appendix A.

5.1.1 Basic operation

A simple diagram of the serial ALU's structure is shown in figure 5.1. The flow of activity starts with the arrival of a decoded instruction from the instruction decode stage into the instruction latch. The decoder breaks instructions down into a smaller number of ALU operations (for example both an `ADD` and a `CMN` instruction perform an addition in the ALU, an `EOR` and a `TEQ` both perform an exclusive-or operation). The instruction decoder also arranges for the correct operands to be sent from the register file. `MOV` instructions are special in that they only require the B operand, and a bit is set in the instruction packet indicating that the instruction should start having received only the B operand. Each operand arrives in four chunks, as an 8-bit data value or an 'empty' partition (least-significant byte first). The output of the instruction latch sends a request downstream when it has been given an instruction. This single request is then turned into four requests, so that the actual execution operation is performed four times (one for each pair of operands).

For each of the four internal cycles, the following events occur:

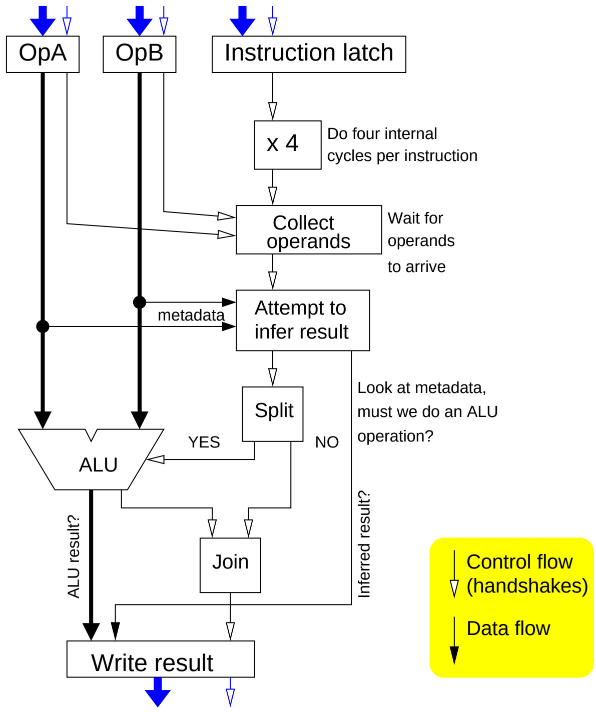

Figure 5.1: Byte-serial ALU

1. The `collect` block waits for handshakes from the two operand latches signifying that both operand chunks have arrived and the execution can continue. However the signal $HaveA$ from the instruction packet optionally may tell `collect` that it only has to wait for the B operand as A is not required in this particular instruction (i.e. it is a `MOV` instruction).

2. The metadata (and data, if *Full*) of both operand chunks is examined. A signal $IsALU$ is asserted if the result cannot be inferred from the metadata and inferrer rules and an ALU operation is performed. $IsALU$ controls a split in the path; the inferrer evaluates the result in parallel so that if $IsALU$ finds that the inferrer's result is valid it is ready to be passed on to the result latch. If $IsALU$ determines that the ALU must perform an operation, a handshake is sent to the ALU.

3. When the ALU receives a request it evaluates the required function and when finished passes a request onward with its result. The forked path with the inferrer on one side and the ALU on the other joins at the result latch, generating a shared request for the latch to accept the result (inferred,

or calculated in the ALU).

Once the result latch accepts the result, an acknowledge is passed back up and another cycle begins. Each cycle performs the same operations with the decision to enable the ALU evaluated for each partition's data (i.e. partitions operate independently from each other) with the exception of the carry-in during addition operations. When performing an addition, the first cycle takes the carry-in for the whole instruction (passed in with the instruction packet). The next three cycles take the previous cycle's carry-out as their carry-in, ensuring correct carry-propagation across the 32-bit word. Both the inferrer and ALU must accept carry-ins and generate carry-outs.

5.1.2 Control logic

Chapter 4 showed that in many cases, when an ALU operation is performed a (faster and cheaper in energy terms) *Move* operation can often be performed instead. These cases are recognised in the inferrer and whilst the execution still must be passed to the ALU, a *Move* operation is requested instead of the instruction's normal operation. The inferrer has control over the ALU's operation and carry signals. For example in the addition case where $Empty(1) + Full(n) + Cin$ is being evaluated, a move on n is performed and the inferrer forces the ALU result to include a carry-out.

The control logic also formats the real data operands input to the ALU; the data from the operand latches passes through a `TC01` block just before being fed into the ALU inputs. These blocks pass the input through as either *True*, *Complement*, *Zeroes* or *Ones* (TC01); that is, each operand may be inverted, or forced to all-zeroes or all-ones before being fed into the ALU. This is done for two reasons.

Firstly, the ALU performs a smaller number of operations (`ADD`, `OR`, `AND`, `EOR`) than there are instructions, and there needs to be a way of synthesising instructions such as subtract. To perform a subtraction of $A - B$ (in executing `SUB` or `CMP`) the ALU must perform an addition between A and $(-B)$; in 2's-complement arithmetic this is $A + \overline{B}$ with a carry-in of 1. The instruction decode sets the *Binv* control bit in the instruction packet and the `TC01` logic inverts the B operand. Logical instructions such as `BIC` (which performs $A \wedge \overline{B}$) are executed in the same way. The A operand can also be inverted for instructions such as `RSB`

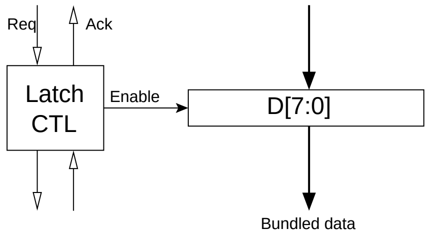

Figure 5.2: Conventional asynchronous pipeline latch

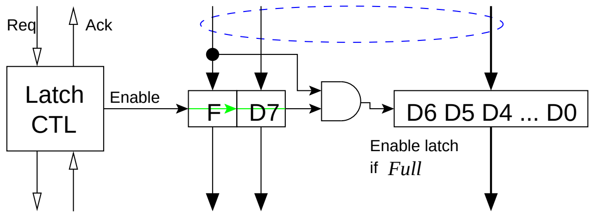

Figure 5.3: Data/metadata pipeline latches

(which perform $B - A$).

Secondly the control logic may need to alter the operands to evaluate a particular partition; for example in an `EOR` instruction, $n \oplus 1 = \overline{n}$ so the ALU is forced to perform a *Move*, and forces the `TC01` to invert n before the *Move*. The inferrer logic only generates metadata results but there are cases where a *Full* data result must be generated; *Empty(0)* + *Empty(0)* + *Carryin* must result in *Full(0x01)*, so the inferrer forces the ALU to perform an addition with two `0` operands and a carry-in to generate the correct *Full* result. Conversely, there are cases where the reverse is true, when the ALU takes two *Full* data operands and generate a result; the actual result will usually also be *Full* data, but there is the possibility that an *Empty* result will occur. For instance, `0xaa` + `0x55` = `0xff` (*Empty-ones*), or `5 - 5` = `0` (*Empty-zeroes*). The ALU thus detects whether the result is zero (which is common in ALUs for updating the Zero status flag) or all ones, and if so causes a signal to tell the result latch that only metadata needs to be latched. This is supplementary to purely inferred results.

5.1.3 Datapath latches

The main operand buses are designed to carry data represented as described in section 5.0.4, eight data bits and one metadata bit F to distinguish between

Full and *Empty* data. When an operand is *Empty*, the only valid bits are $D[7]$ and F; when it is *Full*, all nine bits ($D[7:0], F$) are valid. Since bus signals often have significant capacitance, and many such signals are switched, reducing transitions is important. Traditional (asynchronous) pipeline latches (figure 5.2) use a common *Enable* signal from the latch control logic across the whole width of the latch. This switches a significant load on the *Enable* signal, and will probably change the state of the lower seven bits which may not be required as new data will frequently just be *Empty*.

Instead, the two operand latches and the result latch have a structure shown in figure 5.3. When data is presented at the latch with a request and the latch controller accepts it, the first two bits of data are latched (F and $D[7]$). If the data to be latched is *Full*, then $F = 1$ and the rest of the width of the latch is also enabled, so all 9 bits are latched. If, however, the data is *Empty* then only the top two bits are latched and the rest are unchanged. This structure avoids activity in the lower part of the bus – since for *Empty* data the lower half is undefined (the latch inputs may be in any state) and the latch controllers hold the outputs constant – and the minimum energy is therefore used.

5.1.4 The ALU block

The ALU evaluates the four functions `ADD`, `AND`, `OR` and `EOR`. A 6:1 multiplexer selects the desired result according to the function required; two extra *Move* ports are available, whereby the result is taken directly from the A or B operand bus, missing out the ALU entirely. In many other simple ALUs a *Move* operation is performed by setting one operand to zero (using a `tc01` block) and performing an `OR` operation. However, because the inferrer replaces many ALU operations with a *Move*, they are more frequent in this design so adding the extra fast-forwarding ports to perform the *Move* speeds up their execution by taking the ALU logic out of the *Move* critical path. Accelerating frequent operations can improve performance in asynchronous design.

A ripple-carry adder is used in the ALU. Ripple-carry adders are very slow for large word widths but the completion time is adequately small for this 8-bit ALU. For small adders, the large overhead in logic complexity and energy efficiency of faster carry-lookahead adders may not be justified; a small ripple-carry design will save energy over carry-lookahead designs. The adder may include completion detection, an option in asynchronous adders that allows early termination and thus variable evaluation times.

Logical operations will have reasonably uniform completion times, and a simple bounded-delay derived from simulation will suffice to time the operation. The operation type determines whether the incoming request is delayed by the bundled delays (e.g. *delay(XOR)* or *delay(AND)*) or whether the adder completion signal determines completion.

5.1.5 Design simplifications

In the design of the split-datapath ALU, a number of simplifications had to be made to make the implementation feasible in the development time available. Creating schematics for a fully-featured ALU would have been possible, but a simpler design was chosen so that the whole ALU could then be implemented as layout. A schematic of a circuit can be simulated with loosely estimated timing properties, but in order to obtain more accurate power consumption and timing data capacitances extracted from real layout are needed. This was required to make reasonable comparisons between the energy efficiency and speed of the split-datapath ALU and a 'traditional' 32-bit ALU.

Among the simplifications that were made to the initial design are the omission of status flags (overflow, zero, negative and carry flags in the ARM) in the result, and the optimisation of skipping register writes when updating data.

The actual result latch returns the carry-out from each stage of evaluation. In a real processor's ALU this would probably be held internally and once the four cycles have completed the status flags returned. The flags would comprise the last carry-out flag, overflow (which should be detected in the ALU and returned from the last cycle), negative (which is simply the top bit of the result, and should be latched after the last cycle) and zero. Although zero is currently detected in the ALU, the overall zero flag for the entire 32-bit result must be generated from the logical AND of all of the four cycles' zero flags (i.e. the result is zero when all of the cycles generated zero). Of course this would also have to take into account inferred *Empty zeroes* partitions. Whilst the extra logic to generate the proper flags is not substantial, there would be little gain returned from the time taken to implement it.

The mechanism to avoid register writes would also be reasonably simple to add to the design. The instruction decode can tell if the instruction is updating a register's value (the destination register is the same as one of the operands) and supply information in the instruction packet. A simple function evaluated as the inferrer is inferring the result and generating *IsALU* can indicate that the result will be the same as the updated register's current data value, usually because a *Move* operation would be performed. Instead of then sending a request to the ALU, the whole cycle would then be acknowledged straight away. The cycle would be terminated (and the next cycle would commence), having avoided performing the ALU operation, latching the result and writing the partition back to the registers. If a split-datapath register file were to be implemented, this optimisation should be included to quantify what savings can be made from avoiding register writes.

5.2 Implementation

The ALU block as implemented contains some significant inefficiencies. These were the result of time limitations; simplifications were made in order to ensure that the implementation could be completed in the time available.

The ALU block was kept very simple. It uses a traditional ripple-carry adder with no completion-detection circuitry. A bounded-delay mechanism is used instead; in its current state, the ALU uses a single delay regardless of the operation performed. An obvious optimisation that should be made is to at least vary the bounded delay depending on the operation requested, to avoid every operation taking the same 'worst case' addition time. Simulation can determine the completion times required for these different operations and the actual delay varied; however matching delays is time-consuming and the emphasis was on developing a working prototype rather than low-level optimisations.

The ALU block is also missing logic to gate off inputs for unused functions to reduce activity; as it stands the block simply evaluates all functions and selects the desired result. This wastes some energy for every ALU operation performed. Again the emphasis was on achieving a working system first, and leaving optimisations until later. Furthermore inefficiencies in the ALU may have a smaller impact in this split-datapath design because most of the operations are inferred and not performed in the ALU block.

A trial byte-serial ALU was implemented from standard cells from a cell library supplied by ARM Ltd., using the Compass CAD system. Full schematics are available in Appendix A. Layout was then synthesised, targetting a 0.35μm

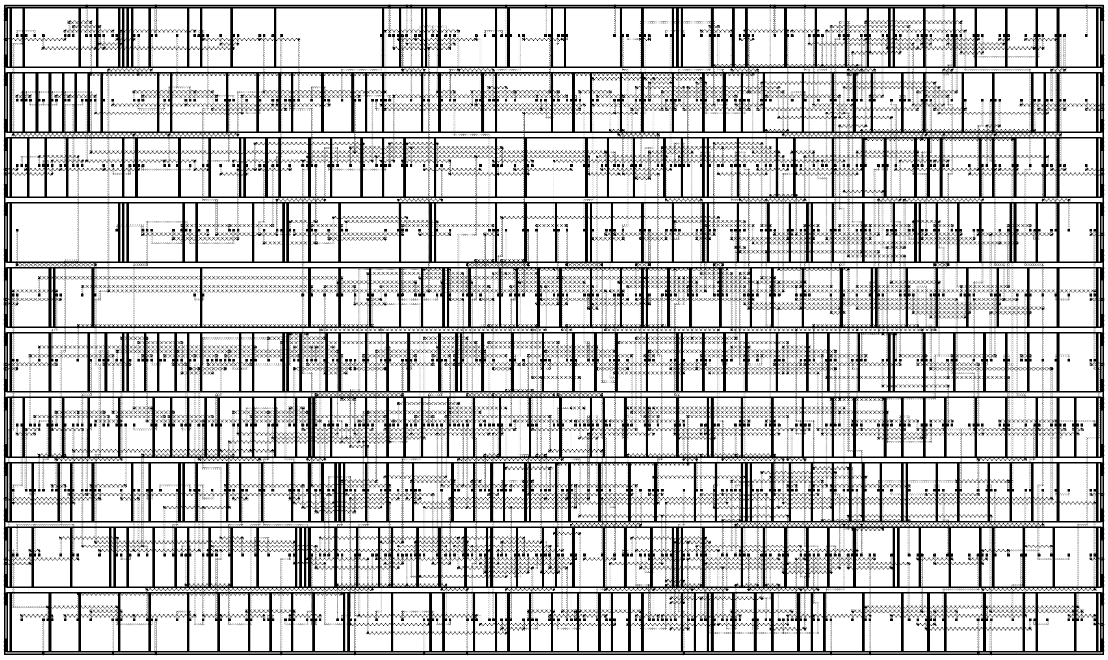

Figure 5.4: Byte-serial ALU cell layout

3-layer metal process. The cell layout is shown in figure 5.4.

The implementation is a proof-of-concept prototype, and little time was spent with hand optimisations of the circuit. Most gates were sized according to their load to improve switching speeds, but there was no time available for in-depth transistor and gate size optimisations. The design would benefit greatly from the datapath being laid out by hand and cell contents hand-optimised, but this is extremely time-consuming and prevented by the project deadlines. The results from unoptimised layout are expected to be pessimistic but this is a fairer comparison than, for example, attempting to compare only with schematic-based simulations.

5.2.1 Simulation

Once the circuit was laid out, simulations could be run to determine performance and power consumption. Both *Timemill* and *Powermill* from EPIC were used to get accurate (usually within 10% of real silicon) timing and power estimates. A software test harness was constructed in 'C', with the ADFMI interface to the EPIC simulator software. ADFMI allows software triggers on logical events in the hardware under simulation, and flexible software models of circuits can thus

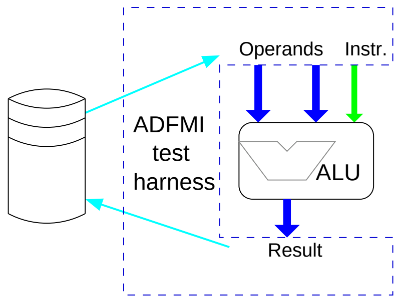

Figure 5.5: ADFMI Timemill/Powermill test harness

be built in ‘C’. A model was written (figure 5.5) to wrap around the circuit, interfacing to the input and output of the ALU. The model behaves like the upstream components in a processor’s datapath that handshake in requests to execute instructions alongside handshakes supplying register data to operate on. The model also accepts results written out the other end of the ALU, appearing like a register file accepting data to write. The model reads in a stream of pre-decoded instructions from a file, sends them through the ALU and logs the results produced back to disc.

The instruction stream and register contents supplied to the circuit are read in from a file by the model, allowing interchangable programs to be run on the ALU. The instructions and operands in the three benchmarks evaluated earlier (DES, Dhrystone and the GSM filter) were read in by the test harness; the same three datasets that software simulations were taken from were therefore run through the ALU. The fact that the same data were used in the ALU circuit allows the accuracy of the early software simulations to be investigated. There is also value in the fact that the ALU was simulated running real programs and not a suite of contrived test vectors.

5.2.2 Comparison: The AMULET3 ALU

In order to put the simuated results of the byte-serial ALU into context, a comparison was needed between a monolithic 32-bit ALU and the split-datapath system. It was decided that the ALU in the AMULET3 was a good (if tough) comparison. The AMULET3 is a reasonably current design, implemented on the same process, in the same asynchronous design style (4-phase bundled-data). These constant factors bring the two designs closer allowing more straightforward comparisons

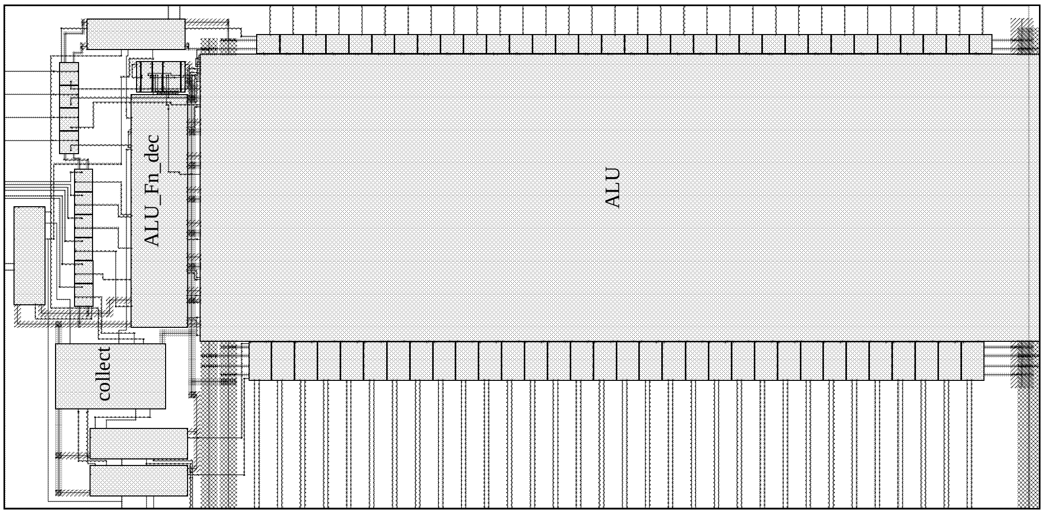

Figure 5.6: AMULET3 ALU test circuit

to be made.

The ALU in the AMULET3 was designed in 1997 by Jianwei Liu [Liu97]. It is a high-performance and low-power design, taking 1.8ns to perform a 32-bit addition with a novel carry generation scheme. The AMULET3 ALU circuit under test is well-designed with a large amount of careful hand-optimised layout, performed over a relatively long period.

The ALU was placed in a test circuit to enable it to handshake in operands and operations, and taken to layout (figure 5.6). A similar model of the 'outside world' was constructed using ADFMI, feeding in decoded instructions and logging the result in the same way as for the byte-serial ALU. The same benchmarks were run on both systems, and the performance and power consumption of both estimated using the simulator tools. The results of the simulations, and a quantative comparison between the two implementations are presented in the next chapter.

Chapter 6

Results

The previous chapter described the simulation test harness in which the byte-serial ALU (SerALU) block was run under *Timemill* and *Powermill* simulations. The simulations modelled the circuit with Vdd = 3.3V and at a temperature of 25°C. Real programs and operands from the three benchmarks (DES, Dhrystone and GSM), a total of over 7500 instructions, were run through the SerALU block and the results written back to a file. Comparing these calculated results with the results generated by the software ARM model showed that it had calculated the results correctly.

Figure 6.1 shows a *Timemill* trace of some of the activity that occurs as the SerALU block processes data. External handshakes for the instruction packet (`ireq`, `iack`), the two operands `ain` and `bin` and the result at the other end are shown on the plot together with two internal signals, `seralu.reqalu` and `seralu.reqinf`. The bottom two signals are the request lines after the select

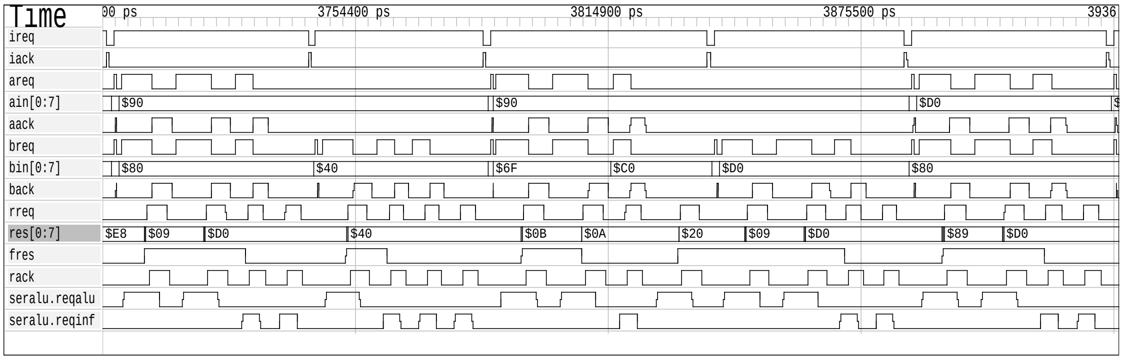

Figure 6.1: Signal trace of SerALU executing part of Dhrystone

block that splits the path between an ALU operation and an inferred operation; it is apparent from these requests which cycles were ALU operations and which were inferrer operations.

The events occuring during the execution of four instructions are shown in the plot, with time going from left to right. At the very top one can see new instructions being delivered with `ireq`; the instruction is then followed by four operand 'chunks' arriving with both `areq` and `breq`. Once a pair of operand chunks have been received, the result is calculated. The first two operations in this instruction are ALU cycles (`seralu.reqalu` is asserted), and the last two are faster inferrer cycles (`seralu.reqinf` is asserted). After each result is calculated, `rreq` goes high indicating a result is ready to be written out.

The second instruction executed is somewhat different; there is a conspicuous lack of activity on the A operand port. This instruction is a `MOV`, which requires only one operand. Consequently, to save power, only one operand is sent to the SerALU, which executes the instruction after only waiting for the B operand to arrive. This `MOV` performs one ALU cycle then three inferrer cycles, producing a *Full, Empty, Empty, Empty* result as the `fres` result 'fullness' signal indicates.

The third instruction is again interesting. It performs three ALU cycles and one inferrer cycle on its two operands, but the inferred partition is the third (i.e. a 'hole' in the middle of the word) out of the four. The waveform trace clearly shows the relative execution time for each instruction; one can see that the instructions take varying times to complete, depending on how many shorter inferrer cycles they perform. Many asynchronous ALUs have varying throughput depending on the instructions executed; the execution speed of the SerALU is dependent on the data set used by the software. Ideally the non-inferrred cycles (`reqalu` on the trace) should be shortened too, as mentioned in the previous chapter, with variable-timing in the ALU.

6.1 Execution times

6.1.1 SerALU

Table 6.1 shows the total time taken by the SerALU block to execute the three benchmarks, and a comparison of the performance in MIPS. The instruction throughput varies over 5-10%, depending on the benchmark. This variation is

Benchmark	No. of instructions	Time taken	MIPS
Dhrystone	706	32545ns	21.7
DES	3793	194760ns	19.5
GSM	3015	130371ns	23.1

Average MIPS: 21.4

Table 6.1: SerALU execution speed for three benchmarks

Benchmark	MIPS
Dhrystone	112.7
DES	131.8
GSM	123.1

Average MIPS: 122.5

Table 6.2: AMULET3 ALU execution speed for three benchmarks

not attributable to different instruction breakdowns since all ALU or inferrer operations take the same time (as the implementation of the ALU block in the SerALU circuit has a single bounded delay for an ALU operation). Referring back to figure 4.4, one can see a very similar variation; DES avoids fewer adds, and executes at 19.5 MIPS, Dhrystone avoids more adds and executes faster, at 21.7 MIPS, and GSM avoids almost 10% more than DES and executes faster still because more of its cycles are inferred.

6.1.2 AMULET3 ALU

Table 6.2 shows the same benchmarks running on the AMULET3 ALU. There is some variation in speed between the benchmarks, which is due to the different mix of operations (and the varying times taken for each); the execution speed of instructions in the AMULET3 ALU is not, however, dependent on the data operated upon. As expected, the performance is much greater on the AMULET3 ALU, and it is a factor of 5.7 faster than the split-datapath ALU. This is partly attributable to the serial nature of the split-datapath ALU; however it is about half as fast as estimated in section 4.4.2 where it was estimated to run about 2.3 times slower than a monolithic 32-bit ALU.

The difference between the implementations of the two ALUs is a major factor in this discrepancy. The SerALU layout was compiled from standard cells; its

	SerALU		AMULET3 ALU	
Benchmark	Duration (ns)	Avg. I (mA)	Duration (ns)	Avg. I (mA)
Dhrystone	32545	3.28	6265	7.61
DES	194760	3.44	28776	6.71
GSM	130371	3.51	24440	9.11

Table 6.3: Average current use of the two ALUs for three benchmarks

performance will be lower than hand-optimised layout, because of the inefficiencies in compiling from standard cells. The *Timemill* signal traces show that the 8-bit ALU requires about 2.5ns to perform an addition; this is a lot longer than the 1.8ns required by the AMULET3 ALU to perform a 32-bit addition. Each byte-wide partition is processed in about 10ns in the SerALU block; the remaining time is spent in control handshakes between the latches and logic blocks, and in the *IsALU*/inferrer circuitry. Of course, inferred cycles skip the ALU cycle and are only composed of this logic delay. The amount of control overhead is quite significant because of the inferring work that needs to be performed, and the fact that time was not available to lay out and size transistors in the logic by hand. Tighter hand-drawn layout is very time-consuming but would improve the performance of the SerALU block, both in reducing the control delays and increasing the speed of the ALU logic.

6.2 Power consumption

The *Powermill* simulator provides an average estimated current used by the circuit, over the duration of the program running through the ALU under test. Table 6.3 shows the average currents for the three benchmarks running under the two ALUs, and the duration of each benchmark. Some variation is seen between the power consumption of the three benchmarks on both ALUs. The variation is slight on the split-datapath ALU, and is caused by differences in the number of inferred operations (varying ALU and latch activity). Differences in the data also have an effect although all three benchmarks display similar trends in data, so the variation is small; within about 5-10%. Due to the implementation of the 8-bit ALU, there is little difference between the ALU operations so the overall variation is not affected significantly by the mix of instructions being executed.

The current in the AMULET3 ALU varies over a wider range across the three

Benchmark	SerALU efficiency (Inst/J)	AMULET3 ALU efficiency (Inst/J)
Dhrystone	2.0×10^9	4.4×10^9
DES	1.7×10^9	6.0×10^9
GSM	2.0×10^9	3.8×10^9
Average	1.9×10^9	4.5×10^9

Table 6.4: Efficiency in instructions per joule for both ALUs

benchmarks. This ALU was implemented in such a way that inputs to different ALU functions are gated off if another function is being performed; for example there is no activity in `OR`ing the operands if an `ADD` is being performed. The currents used by the AMULET3 ALU are indicative of the types of operations performed. Logical operations are less expensive than additions, and this is reflected by DES using the lowest current (with many logicals and few additions), Dhrystone in the middle (with few logicals and more additions), and GSM using the most (with many additions performed).

The first impression of the split-datapath ALU is that its peak current use is, on average, 2.3 times lower than that of the AMULET3 ALU. This reduced power is attractive, but the execution speed must also be considered; the AMULET3 ALU uses 2.3 times more power, but for about one-fifth of the elapsed time. A better comparison is the energy efficiency of the ALUs, which factors in the performance. The efficiency is often measured in MIPS/watt; since MIPS is $Instructions \times 10^6/seconds$ and watts are $joules/seconds$, the seconds cancel meaning efficiency is a measure of instructions per joule of energy. Given that $P = Vdd \times I$, the total energy used by the ALUs using an average of I_{avg} amps for $T_{duration}$ seconds in these benchmark runs is

$$E_{total} = 3.3 \times I_{avg} \times T_{duration}$$

So, the efficiency N of an ALU (in instructions/joule) is

$$N = \frac{Instructions}{3.3 \times I_{avg} \times T_{duration}}$$

The efficiencies of both ALUs running the three benchmarks are shown in table 6.4. Although the peak power of the AMULET3 ALU is greater than that of the split-datapath ALU, it is a factor of 2.5 times more energy-efficient; it uses less total energy to execute the same program. It would be beneficial to examine the subsections of the SerALU block (ALU, inferrer, miscellaneous control) and measure their separate power consumption to see what dominates the energy use in the SerALU design and what needs to be optimised. However, this process is not straightforward with the available design tools; the tool used to extract the capacitances from the circuit does not preserve the schematic hierarchy used, making it difficult to measure currents for specific blocks in the circuit.

6.3 Area used by SerALU and the AMULET3 ALU

The AMULET3 ALU and support circuitry (operand and result latches) occupies a chip area of roughly $4430000\lambda^2$ (0.177 mm^2). The entire SerALU block (ALU, latches and control) occupies roughly $2435000\lambda^2$ (0.0974 mm^2), an area about half the size of the 32-bit ALU. This area reduction is significant. The increase in register size from adding required metadata and control is in the region of about 13%, so the overall area may be around 70% of the area of the AMULET3 block. Note that the SerALU block is implemented with a standard-cell layout; hand-drawn datapath components will not only increase performance and efficiency, but will significantly reduce the size of the SerALU circuit.

The next chapter presents a discussion of these results, and conclusions that may be drawn from them.

Chapter 7

Conclusions

This chapter presents an analysis and summary of the results in the previous chapter, and examine conclusions that can be drawn about this split-datapath execution system.

7.1 Summary

The results in the previous chapter show that the byte-serial ALU uses approximately half as much power as the AMULET3's flat 32-bit ALU when running the same benchmarks with the same data, but it requires over five times as long to execute the same code. For most low-power applications requiring a 32-bit processor, a reduced peak power is beneficial; however, in contactless smartcards and battery-powered devices the absolute energy efficiency is more important (as the total energy available is finite and as many operations as possible must be performed with it). The serial ALU is approximately 2.5 times less energy-efficient than the AMULET3 ALU. The reduction in peak power is therefore less significant since a CPU's power can always be halved by halving its processing speed (which leaves the amount of energy per operation unchanged). So this implementation of the byte-serial ALU has no energy advantage over the AMULET3 ALU, in fact requiring *more* energy to complete a task.

The serial ALU in its unoptimised standard-cell layout occupies about half of the area of the AMULET3 ALU. With hand layout the area would be even smaller; although in a complete procesor the register file would require extra bits of metadata and thus be larger, it is likely that a reduction in the total area (serial ALU and register file) would be achieved. The ALU in a processor usually covers a significant proportion of chip area, and a reduction in chip size can reduce chip cost or provide more space for additional functionality.

7.1.1 Analysis

The performance and energy efficiency of the SerALU block are not as high as was predicted in early software simulations. This section examines some of the possible factors that contribute to the reduced energy efficiency and performance.

The ALU block in the SerALU was implemented with several known inefficiencies. The use of a ripple-carry adder in the 8-bit ALU partially accounts for the lower performance of the SerALU block, at least in addition operations. Many faster carry-propagation schemes are available, but these sacrifice energy efficiency for speed. A simple adder is most likely the best choice for energy efficiency (the prime goal), however enhancements over the simple standard-cell implementation can be made.

Completion detection circuitry would improve the performance of the adder by allowing it to be fully self-timed; instead of a bounded worst-case delay, operations can terminate 'early' if possible. The adder would then always run as fast as possible with the given data. Furthermore, other ALU operations should be timed separately from the adder (a bounded delay would suffice as there is no data-dependent delay with logical operations). In the current implementation all operations are timed by the worst-case addition delay, which slows down the potentially faster logical and *Move* operations. By varying the delay with the operation an overall speedup will be achieved (this should be particularly noticeable in the DES benchmark as it is composed from a high proportion of faster logical operations). The *Move* ports in the ALU were implemented to increase the performance of the frequent *Move* operations; however this optimisation is masked in the current implementation as the delay does not reflect the fact that *Move* operations are much faster. Operation-dependent delays in the ALU would exploit the faster *Move* ports and further increase the performance.

The implementation also contains inefficiencies from the energy standpoint. For simplicity, the 8-bit ALU performs all four of its operations in parallel and the result is selected as desired. When an operation is performed the other functions of the inputs are also evaluated, and unnecessary transitions are caused. Worse, when a simple logical operation is requested, an addition is also performed; the carry propagation causes even more unnecessary transitions. This means that energy is unnecessarily wasted in every ALU operation. Although this was known about in the implementation stage, it was considered a lower priority than getting the whole system running and there was no time available to remedy the situation. The impact of this inefficiency is difficult to quantify without more specific current measurements, but was probably underestimated in the implementation stage. It is likely that this ALU inefficiency accounts for a large proportion of the SerALU block's energy use, even though only 40-50% of cycles are ALU operations. The logical functions are derived from the first half-adder in the ALU; when an addition is performed, few extra transitions are generated, but when a logical or *Move* operation is performed an addition is also evaluated. It is estimated that this would double the average energy cost of an ALU operation, especially with the high frequency of *Move* operations. Many ALUs in other processors, including the AMULET3, avoid this activity by gating off the inputs to unused ALU logic and enabling only the section required by the operation. Thus unnecessary transitions are reduced, saving energy.

In the SerALU block, unnecessary ALU transitions usually occur when an ALU cycle is underway; however, transitions can also occur in the ALU during an inferred cycle. Usually, when both operand chunks are *Empty*, the operand latches will hold the lower seven bits of each operand in their last state, and only D[7] will change; this causes activity only at the top of the ALU (with no carries propagating). Unfortunately in some circumstances the outputs of the `tc01ctl` logic block may also change state or glitch, changing one or both of the ALU's operands and causing unwanted activity. This was not forseen and, in hindsight, extra gating should have been added to ensure that the state of the `tc01`s can only change during an ALU cycle, alongside logic to enable only the required ALU function.

Standard-cell and auto-routed layout also contributes to the reduced performance and increased energy use. The performance of the 8-bit ALU is poor compared to the AMULET3 32-bit ALU (the 8-bit ALU taking 2.5ns for an addition and the AMULET3 ALU taking only 1.8ns for an addition four times wider), and this is largely due to the less dense standard-cell implementation. Hand-drawn layout is smaller, implying lower load capacitance, so switching times are reduced; transistor sizes can be suited exactly to the circuit rather than being chosen from a small selection of standard parts. The 8-bit adder would run much faster with full custom layout, and use less energy because of the reduced capacitance.

The gate and transistor sizes in the control logic were not aggressively tuned to achieve higher speeds and lower energy, because of time constraints. Reducing the loads that gates drive will increase switching speeds and reduce the energy used; the inferrer logic could benefit significantly, increasing the energy efficiency of the SerALU circuit.

Among the changes and improvements discussed so far, hand-drawn layout and optimised transistor sizes would most likely provide the greatest improvement in performance and energy efficiency for the SerALU.

The simulation of the layout would be improved with the ability to measure separate currents for different blocks in the circuit. This would allow a better evaluation of the effects of the inefficiencies in the ALU, and the relative merit of ALU and inferrer operations. The tool used to extract capacitances does not preserve the hierarchical nature of the design making it difficult to confine measurements to specific blocks in the circuit. However, it is possible to process the extracted data to simulate with hierarchical block names, but this is not an automatic procedure and was not achieved, again due to time constraints.

7.2 Conclusions

The analysis of the high-level software simulations presented in chapter 4 suggested that a serial split-datapath ALU could be more energy efficient than a monolithic 32-bit ALU; the function to infer the result is relatively simple and the majority of addition operations will perform this infer function rather than an ALU addition.

However the implementation does not bear out this hypothesis, and the SerALU as implemented is less energy efficient than the AMULET3 ALU. There are a number of known inefficiencies in the SerALU implementation which reduce the performance and energy efficiency. The performance and power efficiency of the AMULET3 ALU are hard targets to compare against, even for most other 32-bit processors, so the effects of any inefficiencies will be more marked.

7.2.1 Energy

Although the *power* efficiency of the SerALU is better than that of the AMULET3 ALU, it is not significant since *energy* efficiency is the most important factor in most applications. The SerALU requires roughly 2.5 times more energy per operation than the AMULET3 ALU, but a significant proportion of this will be due to the implementation's inefficiencies. The standard-cell layout and sub-optimal ALU construction may account for most of this difference.

The cost of inferred operations is also higher than expected, even though it is quite a simple function; again the standard-cell layout will reduce the speed of inferred operations and increase the energy consumption.

Overall, it was perhaps not quite fair to compare the SerALU implementation to the AMULET3 ALU. The AMULET3 ALU was built over a much longer period of time, with full-custom layout. Although the the energy efficiency of the SerALU is pessimistic, it is not far from the target energy efficiency of the AMULET3 ALU; it is reasonable to expect that, with an optimised full-custom implementation that includes logic to reduce activity in the ALU and `tc01` blocks, the SerALU energy efficiency could at least match that of the AMULET3 ALU. Energy comparisons should have been made after full optimisation of the SerALU was performed, for a more accurate view of the energy consumption.

7.2.2 Speed

The high-level software simulation indicated that a byte-serial split-datapath ALU would have a throughput roughly 2.3 times slower than a traditional 32-bit ALU. However, the performance of the SerALU is just under half of that expected. This is due to the 8-bit ALU implementation and the standard-cell layout. The 8-bit ALU uses a poor bounded-delay model that does not take advantage of faster *Move* and logical operations, creating a large performance penalty. Furthermore, with custom layout the ALU performance will increase significantly; if the AMULET3 ALU can perform an addition in 1.8ns then it is clear that the 2.5ns the 8-bit ALU takes to perform an addition can be improved. It is reasonable to expect custom layout and a fine-grained ALU delay model to double the performance, bringing it to the level predicted by the simulations.

Another option for increasing performance of a byte-serial ALU is to overlap the activity in the ALU and inferrer blocks so that they can proceed in parallel. Referring back to the first instruction on the left of figure 6.1, it can be seen that two ALU operations are performed followed by two inferred operations. Instead of the inferrer lying dormant while the ALU works (and vice versa), it is possible to evaluate the ALU operations and the inferred operations at the same time (carry-in dependencies permitting, if the instruction performs an addition), providing a significant increase in throughput; the inferrer and ALU blocks could even process data from different instructions. Because of the widely varying execution times this technique would produce the parallelisation may only be efficiently implementable in an asynchronous system.

7.2.3 Size

The SerALU occupies about half of the area of the AMULET3 ALU, and this is the first clear advantage of this implementation of a byte-serial ALU. The SerALU is smaller despite its standard-cell layout, and would shrink drastically if implemented in full-custom layout; a reasonable estimation would be that hand-drawn layout would reduce the SerALU block to less than a third of the AMULET3 ALU's area.

Metadata is required for the SerALU operation and extra storage and control logic would be required in the register file; a split datapath would increase the size of the register file by around 13-15%. Therefore the overall area of a split-datapath register file and ALU would be slightly larger than one third of the AMULET3 ALU area; the area of full-custom layout of the SerALU and register file is estimated to be just under 50% of the equivalent blocks in AMULET3.

In summary, this implementation of a serial split-datapath ALU did not succeed in achieving more efficient energy use compared to the AMULET3 'traditional' monolithic ALU. However with custom layout and logic to reduce spurious ALU activity the energy efficiency is likely to at least match that of the AMULET3 ALU.

Although there are clear patterns in the data flowing through processors it is likely that ALU addition operations are less costly than they were previously considered to be, when compared to the logic required to infer the result. It is not that the inferrer logic is complex, merely that the logic required to infer a result is comparable to actually performing the addition in the first place. As the ALU operation becomes more costly (because it is wider, or implemented in, say, dual-rail which uses more energy per bit) the inferrer will become increasingly worthwhile.

7.3 Future work

A future implementation of a byte-serial split-datapath ALU could improve performance by parallelising the operation of the ALU block and inferrer so that both can process data concurrently, as described in section 7.2.2. This would allow most logical operations to execute faster; additions could use some sort of carry lookahead scheme to allow different partitions of the same (or different) words to be added at the same time. This poses an interesting design challenge, and would further improve performance.

A parallel implementation of a split-datapath ALU was dismissed earlier in favour of implementing the partition-serial version; however, a parallel split will provide much higher performance than the SerALU with a slightly smaller energy overhead from simpler control circuits, and for this reason may be worth investigating. Again, a more complicated pipelining approach could be used, whereby each partition proceeds at its own pace and the whole ALU could be executing a number of different instructions in separate partitions.

The byte-serial split-datapath ALU has shown that patterns in program data can be exploited and that it is possible to infer results. Although the benefits of this are uncertain for a two's complement 32-bit ALU, as ALU operations get more and more expensive a datapath split will become more beneficial. An ALU implemented in other asynchronous design styles such as dual-rail and 1-of-4 will occupy a much larger area than a two's-complement bundled-data ALU. A dual-rail adder may also use significantly more energy than a bundled-data implementation. By reducing the width and executing instructions byte-serially as in the SerALU block, the area can be decreased significantly. The larger the ALU is, the more area can be saved by executing instructions serially. Furthermore, as ALU operations become more and more expensive compared to inferrer logic, more energy will be saved by avoiding ALU operations. Such design styles may be more prevalent in the future, and a serial split-datapath execution style is well-poised to offset some of the disadvantages of delay-insensitive 1-of-n codes in terms of energy use and chip area. Similarly, 64-bit ALUs (with much more expensive operations) are becoming more common and a split-datapath approach to a 64-bit ALU could be useful in increasing energy efficiency and reducing area.

This work has concentrated on the execution unit within a processor; it is likely that patterns in data will extend outside the processor, to memory, across inter- and intra-chip buses. Buses are usually relatively large and consume significant amounts of energy; a system whereby metadata accompanies real data on a bus and data is only sent when it has changed or is not *Empty* could reduce bus transitions and save a large amount of energy.

Bibliography

[BD95] G. Birtwistle and A. Davis, editors. *Asynchronous Digital Circuit Design*. Workshops in computing. British Computer Society, 1995.

[CSB92] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, April 1992.

[Dob92] D. W. Dobberpuhl. A 200-MHz 64-b dual-issue CMOS microprocessor. *IEEE Journal of Solid-State Circuits*, 27(11):1555–1566, November 1992.

[EGT01] A. Efthymiou, J. D. Garside, and S. Temple. A comparative power analysis of an asynchronous processor. In *PATMOS 2001*, 2001.

[FGG98] Stephen B. Furber, James D. Garside, and David A. Gilbert. AMULET3: A high-performance self-timed ARM microprocessor. In *Proc. International Conf. Computer Design (ICCD)*, October 1998.

[Fle01] Marc Fleischmann. Longrun power management. Technical report, Transmeta Corporation, http://www.transmeta.com/, January 2001.

[GBJ98] Michael K. Gowan, Larry L. Biro, and Daniel B. Jackson. Power considerations in the design of the Alpha 21264 microprocessor. In *Design Automation Conference*, pages 726–731, 1998.

[Jag90] D.V. Jaggar. A performance study of the Acorn RISC Machine. Master's thesis, University of Canterbury, NZ, 1990.

[KH92] Gerry Kane and Joseph Heinrich. *MIPS RISC Architecture, 2/e*. Prentice Hall, 1992. ISBN 0-13-590472-2.

[KKdBT00] Joep Kessels, Torsten Kramer, Gerrit den Besten, and Volker Timm. Applying asynchronous circuits in contactless smart cards. In *Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2000)*, pages 36–44, April 2000.

[KZP00] Suhwan Kim, C. H. Ziesler, and M. C. Papaefthymiou. A reconfigurable pipelined IDCT for low-energy video processing. In *13th Annual IEEE International ASIC/SOC Conference, 2000*, pages 13–17, September 2000.

[Lab01] RSA Laboratories. RSA encryption standard, January 2001.

[LG01] David W. Lloyd and James D. Garside. A practical comparison of asynchronous design styles. In *Seventh International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2001)*, pages 36–45, March 2001.

[Liu97] Jianwei Liu. *Arithmetic and control components for an asynchronous system*. PhD thesis, University of Manchester, 1997.

[Mon95] Mondex. Mondex electronic cash system, 1995. http://www.mondex.com/.

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

[oS77] National Bureau of Standards. Data encryption standard. Federal Information Processing Standards Publication, 1977.

[PG95] William B. Pannebaker and Didier J. Le Gall. *MPEG Digital Video Compression Standard*. Wiley, John & Sons Incorporated, 1995. ISBN 0442019203.

[Phi98] Philips. MIFARE system overview, 1998.

[Sea01] David Seal, editor. *ARM Architecture Reference Manual*. Addison-Wesley, 2001. ISBN 0-201-73719-1.

[SMI01] Java Card (TM) API, 2001. http://java.sun.com/products/javacard/.

[TRN98] Y. Tong, R. Rutenbar, and D. Nagle. Minimizing floating-point power dissipation via bit-width reduction. In *Power-Driven Microarchitecture Workshop*, June 1998.

[vBBK+94] K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Schalij. Asynchronous circuits for low power: A DCC error corrector. *IEEE Design & Test of Computers*, 11(2):22–32, 1994.

Appendix A

Serial ALU schematics

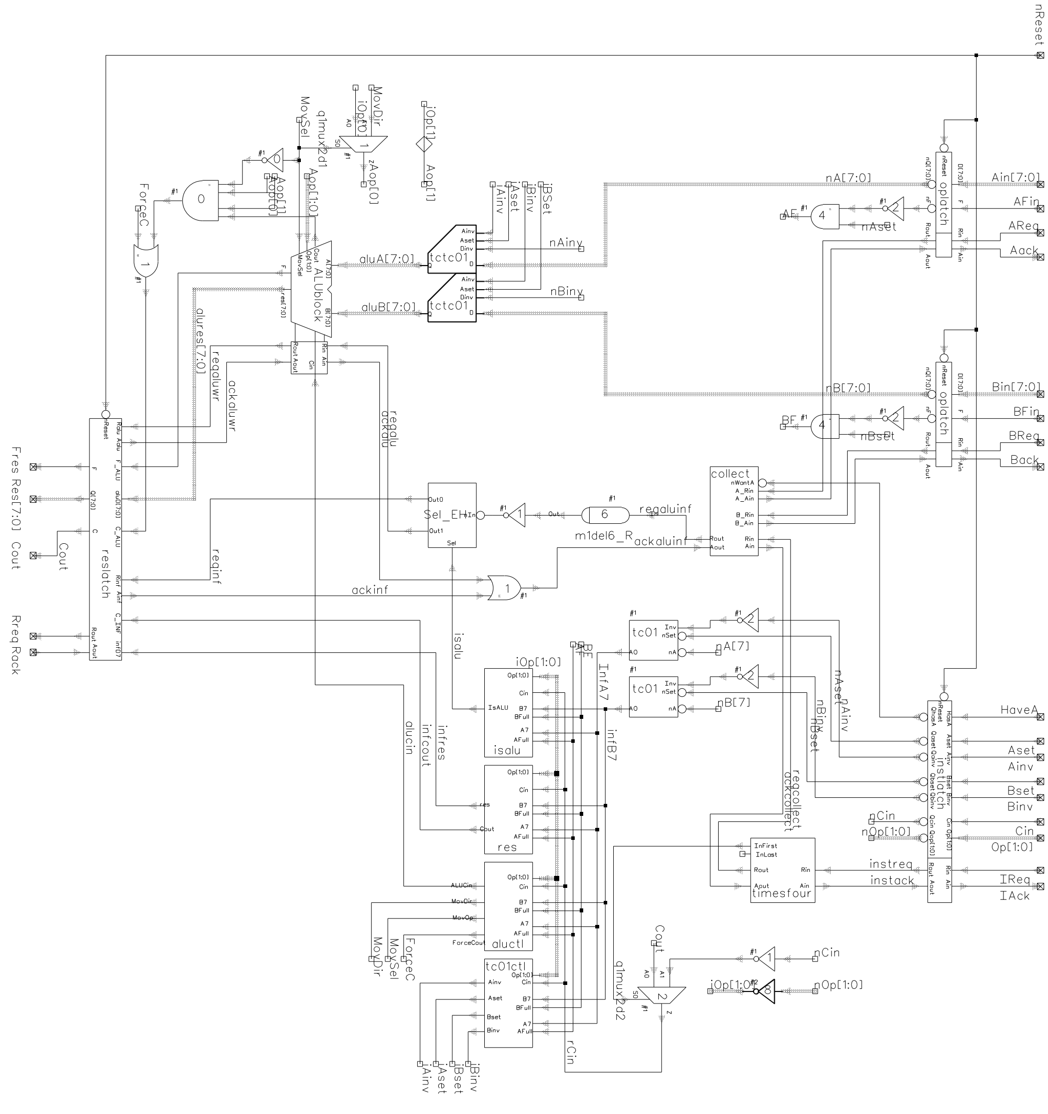

Figure A.1: Serial ALU top-level schematic

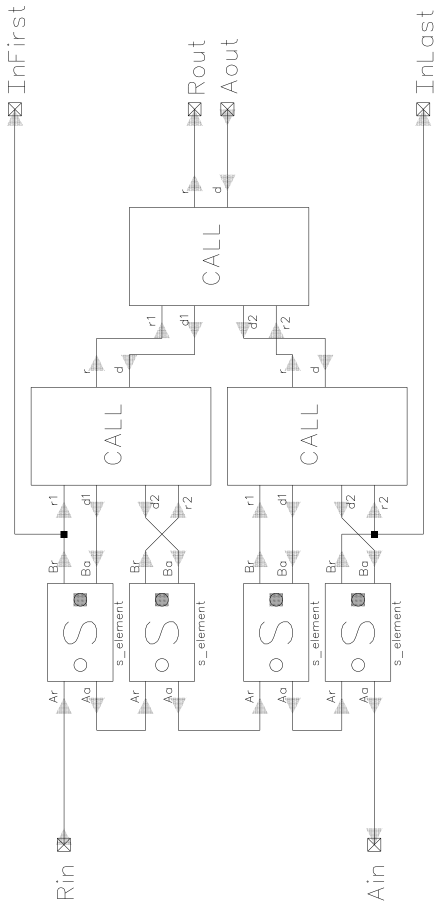

Figure A.2: `timesfour` block schematic

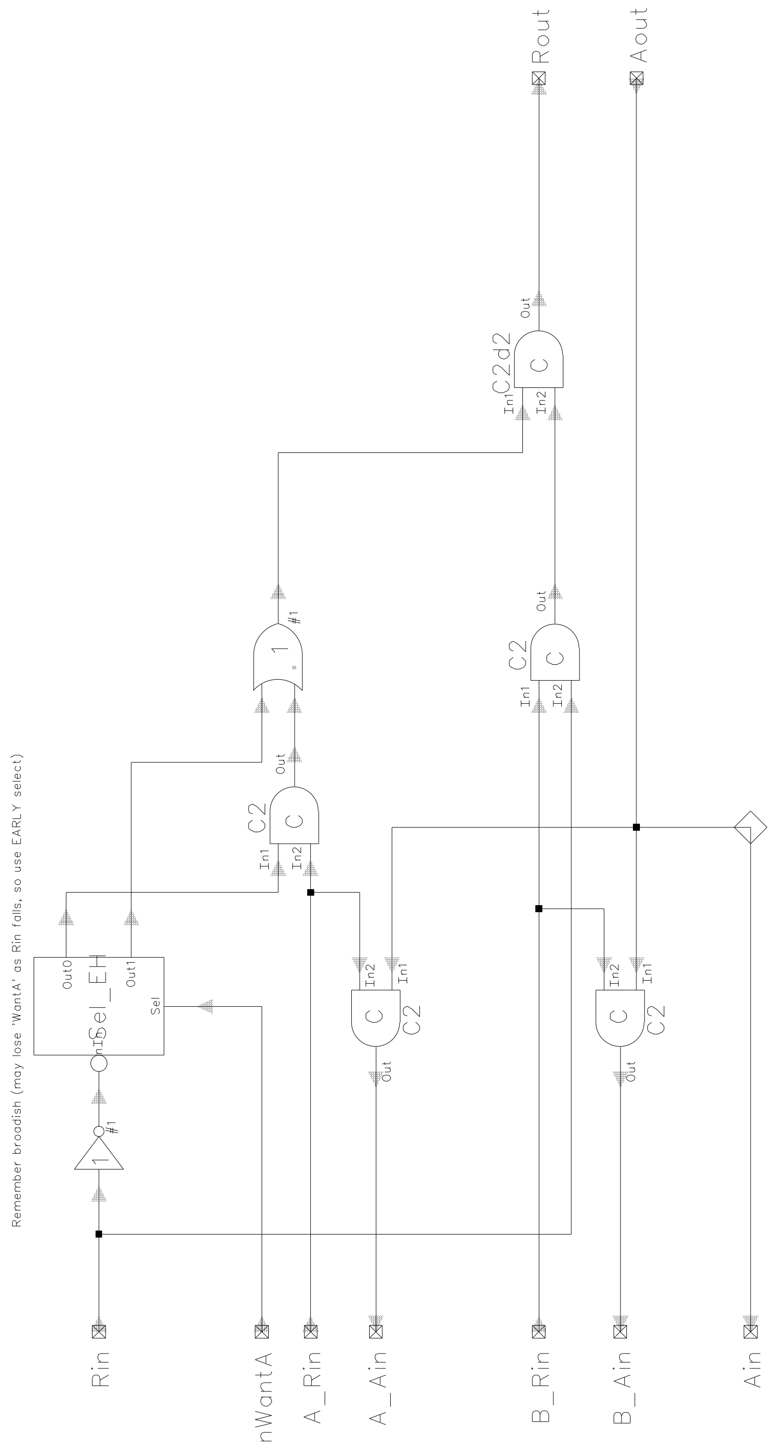

Figure A.3: `Collect` block schematic

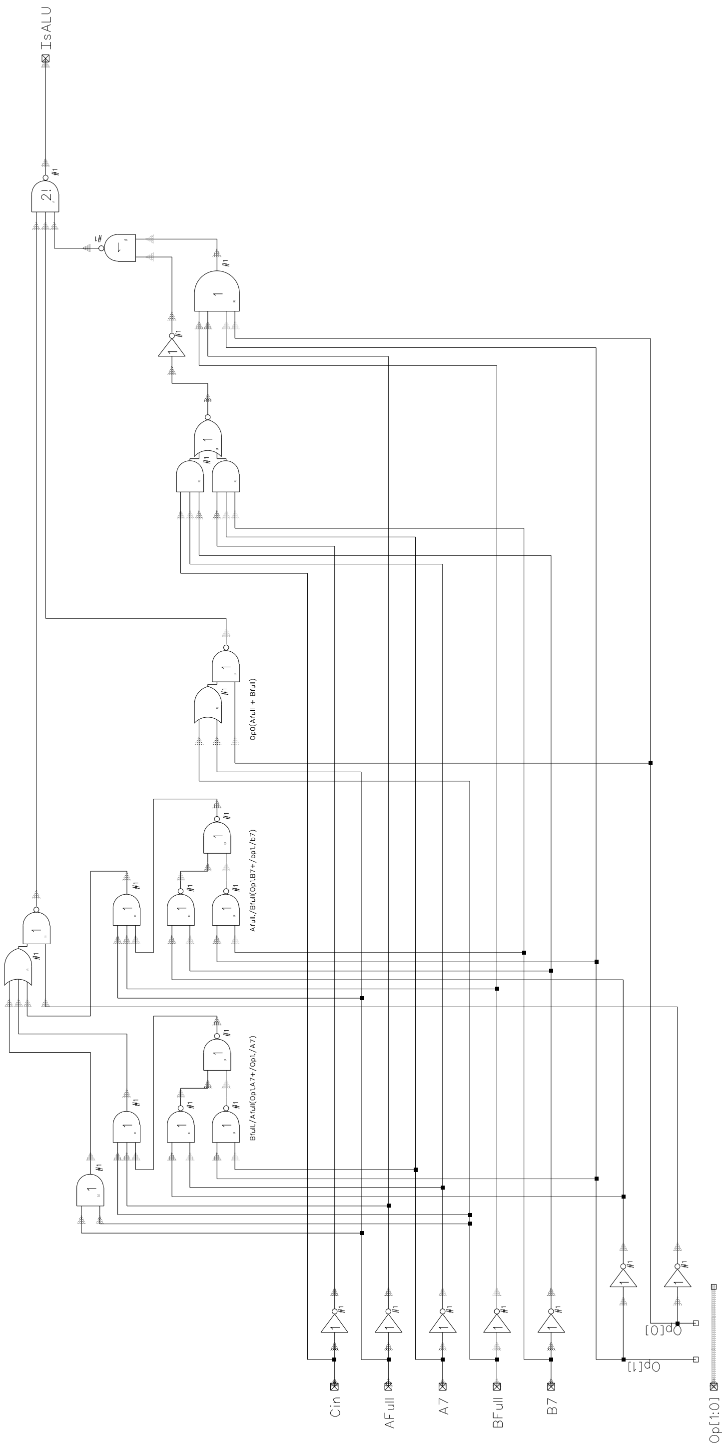

Figure A.4: `IsALU` block schematic

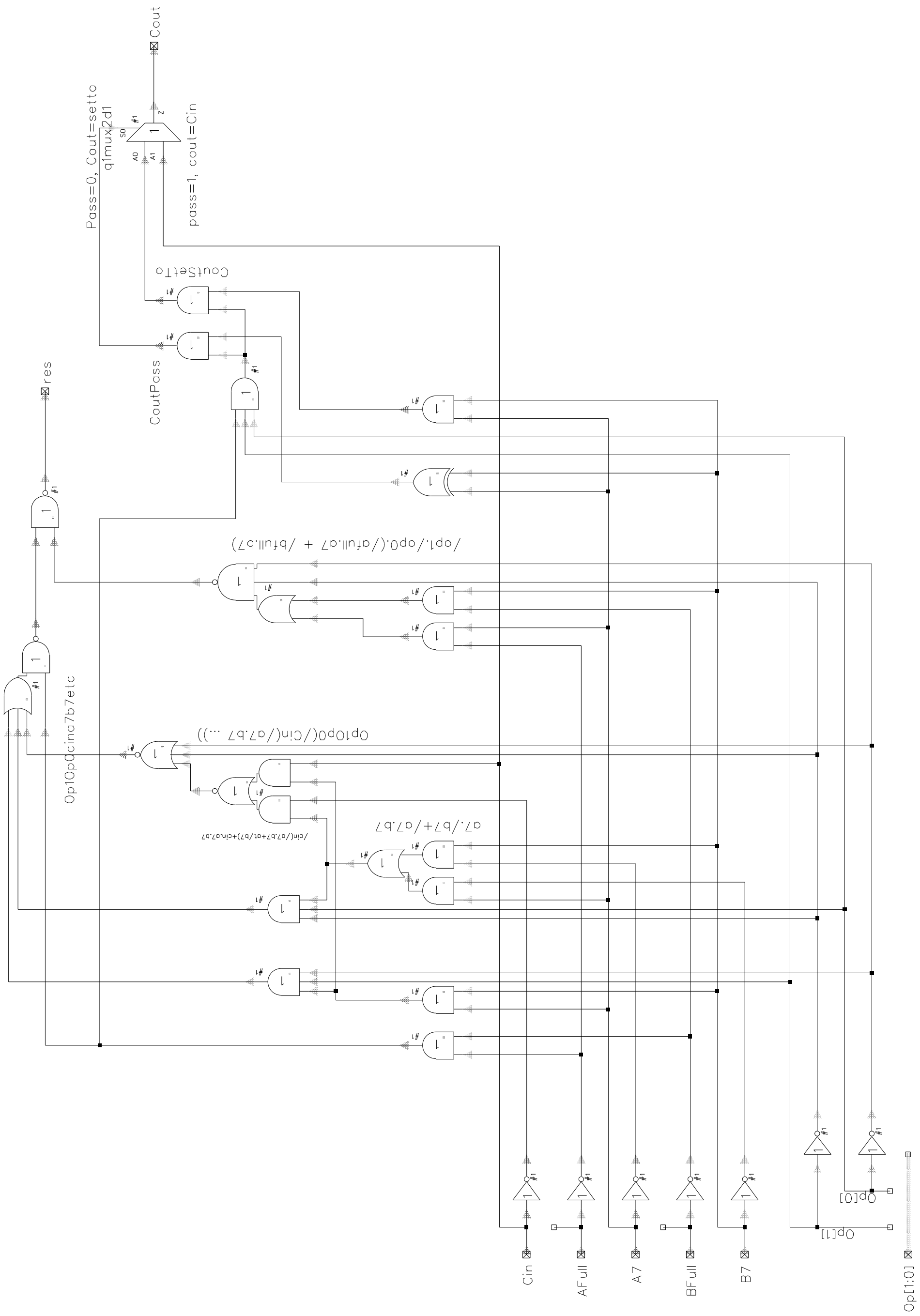

Figure A.5: `Res` block schematic

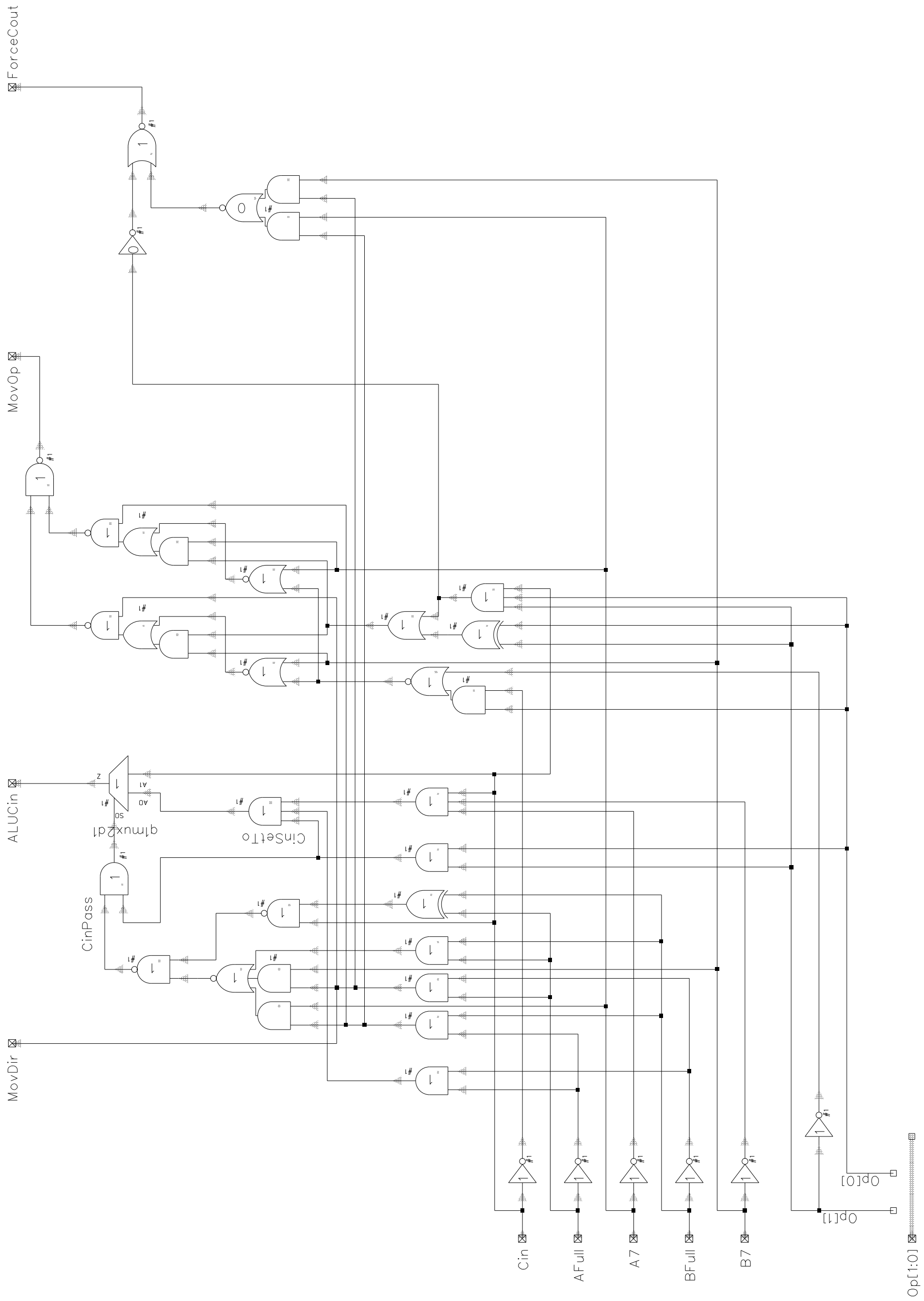

Figure A.6: AluCtl block schematic

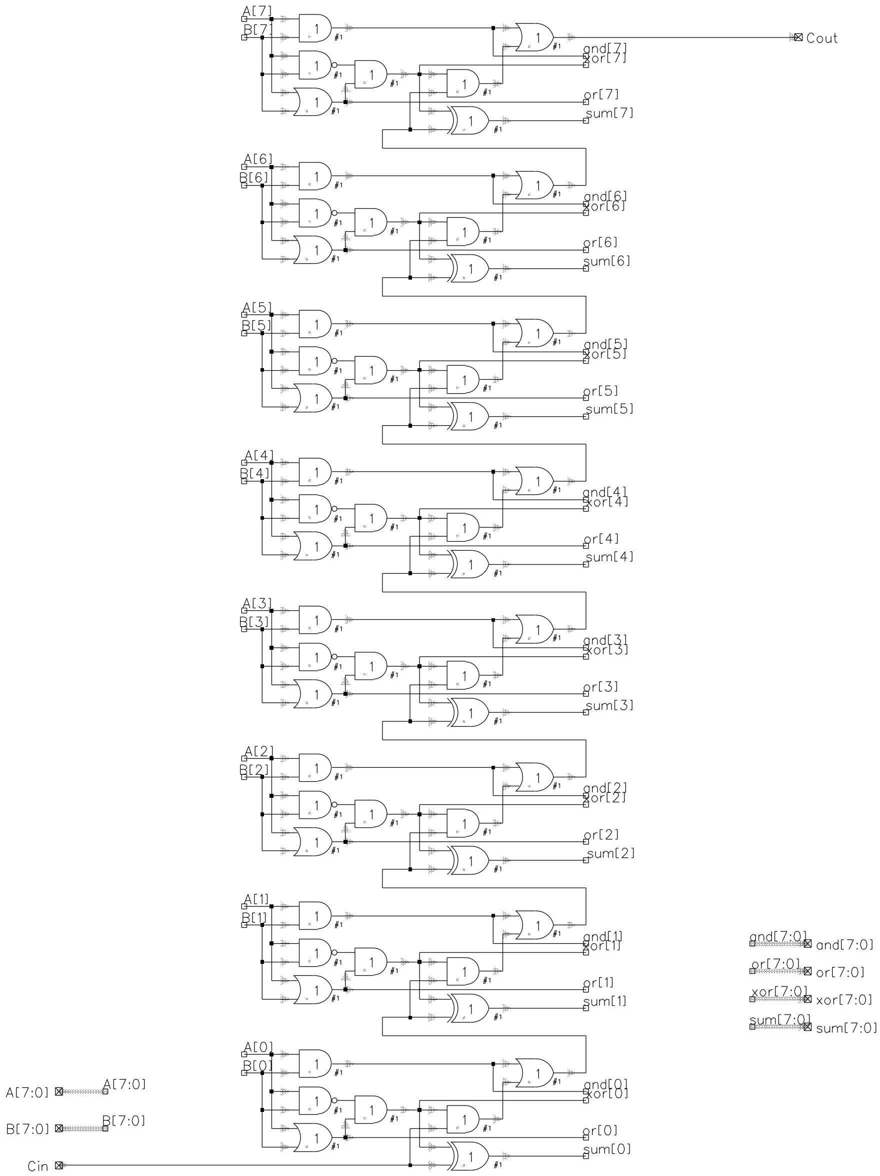

Figure A.7: `alulogic` block schematic

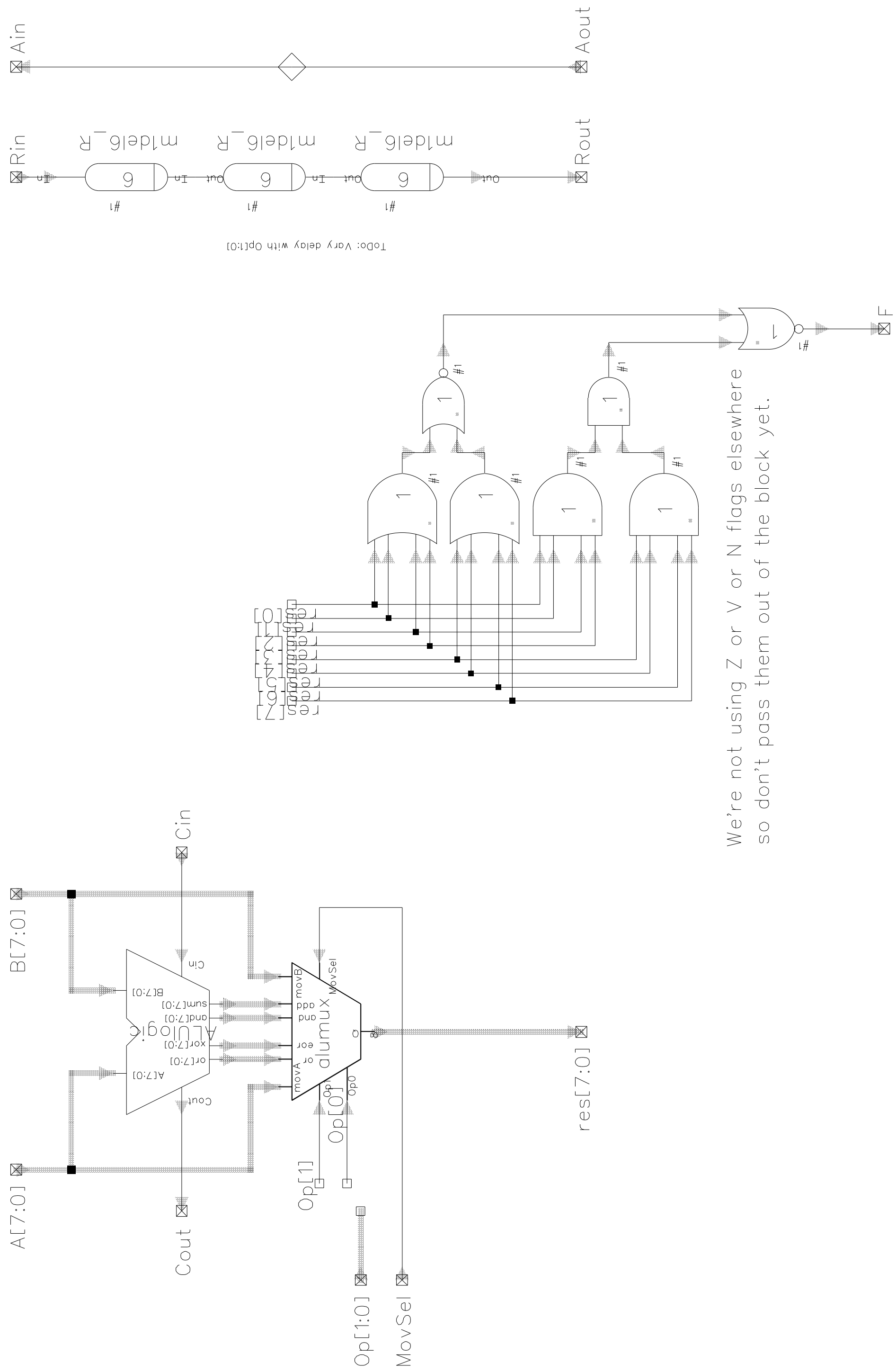

Figure A.8: ALU block schematic

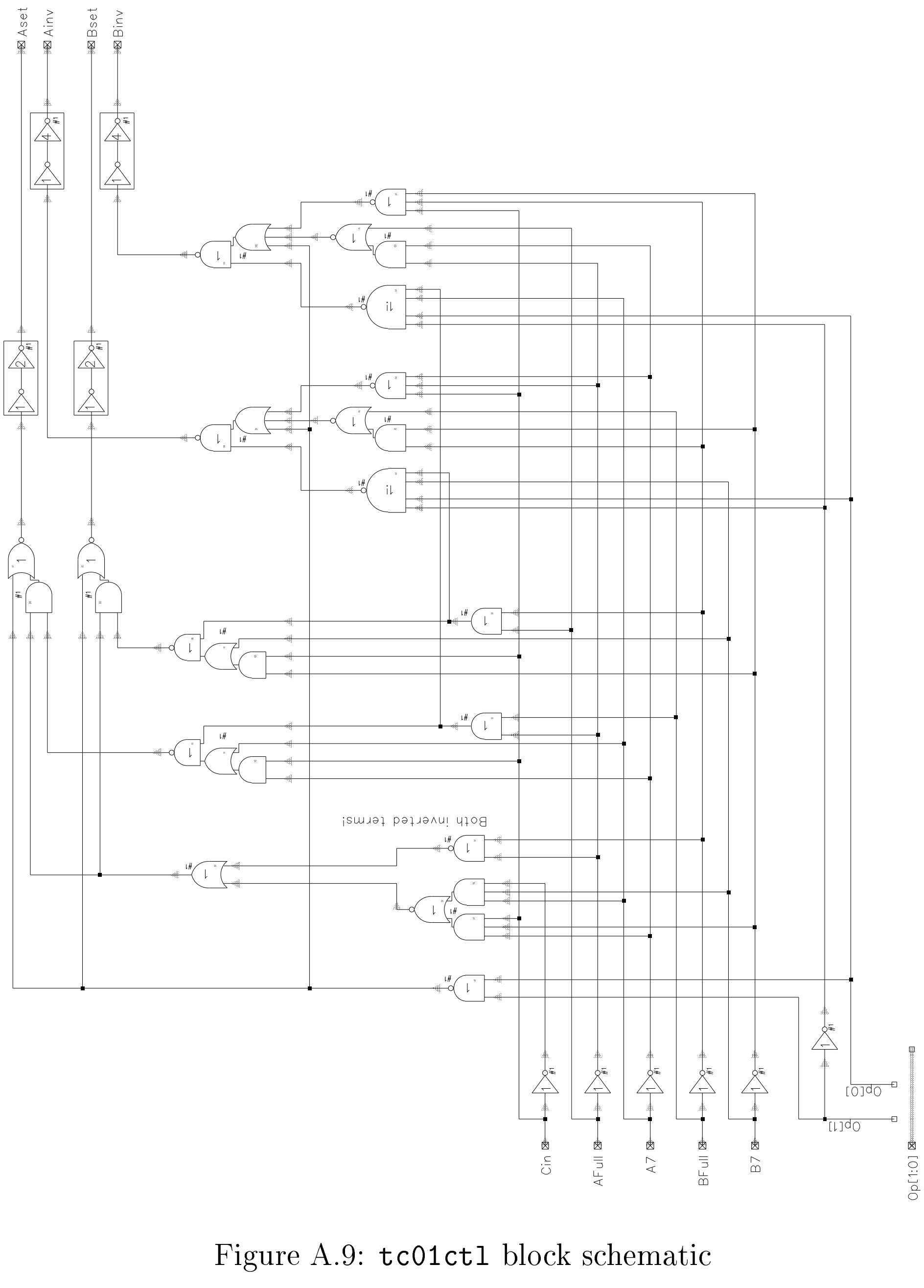

Figure A.9: `tc01ctl` block schematic

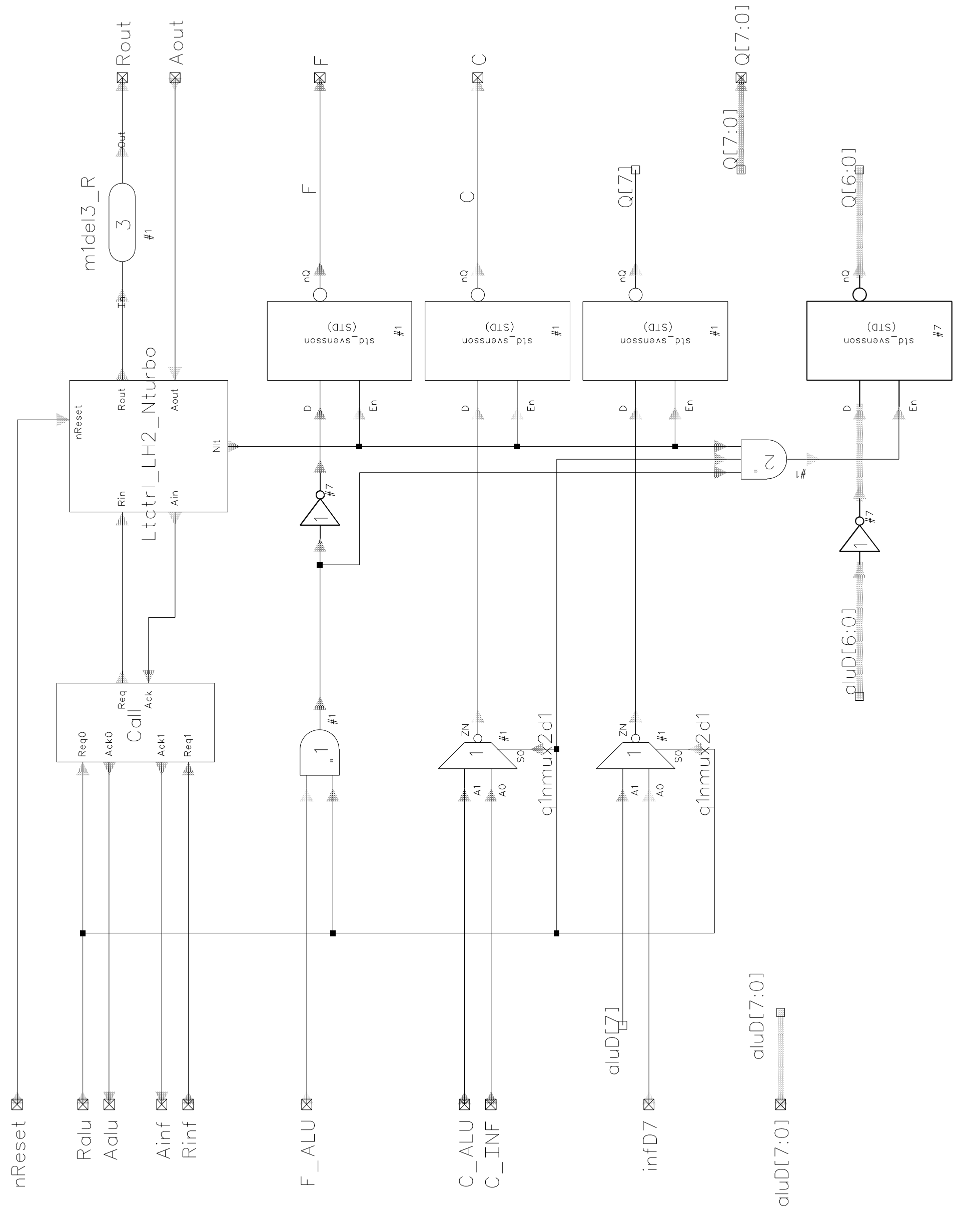

Figure A.10: Result latch schematic