Rotary Pipeline Processors

Simon Moore, Peter Robinson, Steve Wilcox
Computer Laboratory, University of Cambridge

Submitted: 15th December, 1995
Revised: 30th May, 1996

Abstract

The *rotary pipeline processor* is a new architecture for superscalar computing. It is based on a simple and regular pipeline structure which can support several ALUs for efficient dispatching of multiple instructions. Register values flow around a rotary pipeline, constrained by local data dependencies. During normal operation the control circuits are not on the critical path and performance is only limited by data rates. The architecture is particularly well suited to implementation using self-timed logic.

1 Introduction

Most current designs for processors reflect their intended implementation technology. In particular, the availability of extraordinarily large scale integration and the benefits of clocking have led to the current range of superscalar designs using multiple instruction issue into parallel pipelines to increase performance. However, the resolution of data dependencies within and between these pipelines requires many special bypasses which can only be synchronised by a single, central clock. This introduces the further problem of distributing a global clock across a large chip.

Alternative architectures that avoid central clocks are beginning to show some promise. In these, timing is controlled by completion detection or matched delays with local handshakes between stages. The counterflow pipeline processor [12] is designed around a bi-directional pipeline carrying instructions and arguments in one direction and results in the other. This can be implemented in conventional clocked logic, but it also lends itself to an asynchronous implementation using micropipelines [13]. However, the synchronisation required between the rising and falling pipelines appears to impose a severe bottleneck on performance.

The rotary pipeline processor avoids the problem by only passing data in a single direction. In principle, the processor's registers continually circulate around a ring which links various function units — ALUs, memory access and so on — as shown in Figure 1. At each stage, the values are inspected and passed on, possibly after modification, with no significant overhead for synchronisation. Instructions are dispatched from the centre to the function units which allows multiple instruction issue.

This paper introduces the concept of the rotary pipeline and explores its practical implementation. Section 2 develops the architecture and shows how it handles branches, conditional execution, speculative execution and exceptions. Section 3 considers the problems of a practical implementation and gives circuit details for the key components including physical dimensions of their layout. Section 4 presents the results of some preliminary simulations indicating the likely performance of the architecture. Section 5 discusses the relationship of this work to others. Finally conclusions are drawn in Section 6.

2 Rotary Pipeline Concept

2.1 Overview

In its simplest conceptual form, a rotary pipeline circulates all of the processor's registers around a ring (see Figure 1). As the register file flows around this ring, values are inspected by the various function units and results are inserted.

Registers are not kept in lock step although they are prevented from lapping each other. This allows unused registers to be forwarded to subsequent ALUs so that operations can occur in parallel. If there are data dependencies, however, the function units will wait for the new value. Thus it is common for a function unit to be primed with an instruction which it will start to execute as soon as the data arrives.

2.2 Basic Pipeline Construction

A rotary pipeline is constructed from banks of flip-flops and switch networks (see Figure 2). The first switch network in a stage selects which registers a function unit requires for its operands. The result is then injected back into the register file by a second switch network. Finally a set of flip-flops store

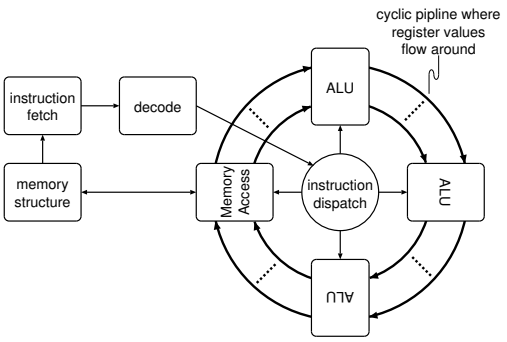

Figure 1: Rotary pipeline abstraction

and buffer the registers at this stage in the pipeline. These allow the preceeding function unit to be reset and made ready for the next operation.

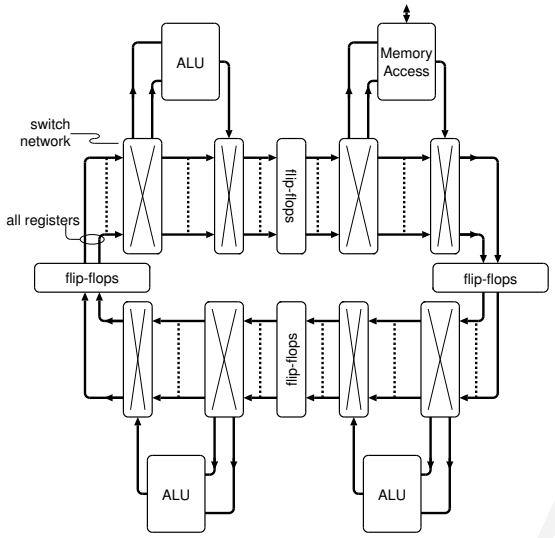

Figure 2: Wide rotary pipeline structure

2.3 Adding a register file

A simple rotary pipeline is quite a large structure if there are many registers. However, if there are only a few function units, then just a subset of the registers will be active. Only this subset need be passed around the pipeline whilst the rest remain in a register file (see Figure 3).

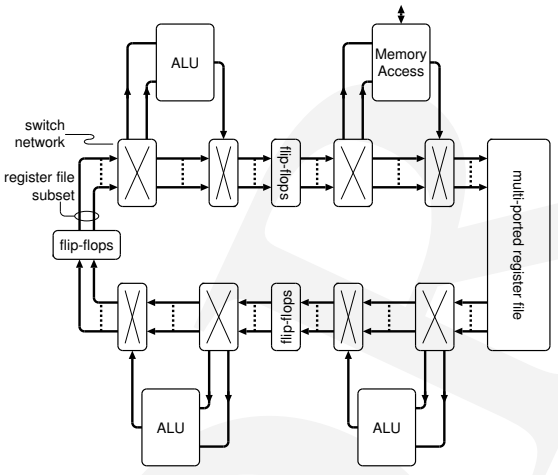

Figure 3: Narrower rotary pipeline with register file

2.4 Rotary bus allocation

Required registers can be allocated to buses in the pipeline on a first come, first served basis when instructions are dispatched. Some of the registers output by the register file may only be read by the first function unit, for example. In which case the function unit may reuse a bus for its output (see ALU 1 in Figure 4). Conversely, a register may be read by more than one ALU which also allows the number of buses to be reduced. However, too few buses will be restrictive when instructions are independent. If there are insufficient buses then instances will arrive when some ALUs will have to be issued `NOP`s to prevent deadlock from occurring. Again, this can be handled by the central dispatcher.

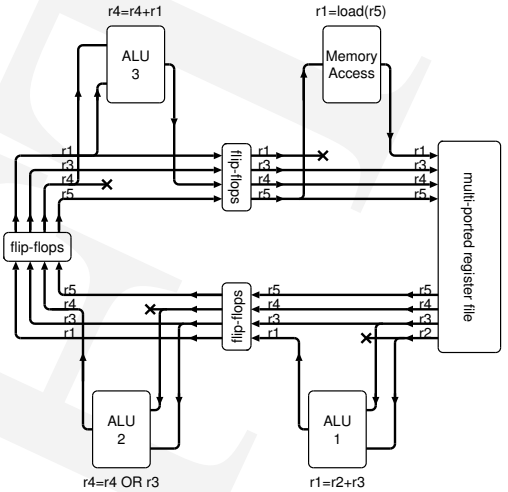

Figure 4: Example of rotary bus usage

2.5 Instruction issue

In the simple case, sequential instructions are issued in clockwise order around the ring in the same direction as the register flow. This ensures that register dependencies are resolved correctly provided registers are prevented from lapping each other. The latter is enforced by the clean-up logic at each stage which sweeps up behind the register flow and prevents registers from advancing into unclean areas.

Some function units (e.g. the ALUs) can execute many instructions but others (e.g. memory access) are more specialised. Consequently, there are times when an instruction cannot be executed by the next available function unit — a no-operation (`NOP`) instruction is issued instead and the subsequent function unit is tried. Fortunately `NOP`s execute quickly (2 to 3 gate delays) because they just pass on the register values around the ring.

Dynamic instruction reordering can be used to improve execution unit utilisation. For example, a `LOAD` followed by a independent `ADD` might be swapped so that the ALU before the memory access unit can be issued with the `ADD` rather than a `NOP`. However, our initial results (see Section 4.2) indicate that function unit utilisation is only increased by 3% on code which is not optimised for this particular architecture (where one would expect to see the most benefit). This also has to be balanced against the expense of lengthening the decode stage which will detract from pipeline refill performance, e.g. after a mispredicted branch.

Even when instructions are issued in-order, the data driven nature of rotary pipelines allows out-of-order execution and completion. Consequently the penalty for issuing an instruc-

tion in the wrong order is less critical than with clocked processors.

Figure 4 presents an example of in-order instruction issue and out-of-order completion. The two instructions (`r1=r2+r3` and `r4=r4 OR r3`) may execute in parallel, the latter completing first because there is no carry propagation. The third instruction (`r4=r4+r1`) is dependent upon the result of the first two instructions and will, therefore, wait for the results to flow around. Finally, the fourth instruction (`r1=load(r5)`) is independent from the first three and may well complete (assuming a cache hit) before the third instruction.

2.6 Condition flags

If a register is required to hold condition codes then it may be passed around the rotary pipeline as a specialised register. This runs the risk of introducing a dependency between every pair of instructions. However, some instructions will not modify the condition codes so they can be passed on quickly. Other instructions will set or clear conditions irrespective of their initial state so it is possible for some or all condition bits to be generated early. Alas, other instructions (e.g. add-with-carry) will have to wait for the appropriate condition bits to arrive and will hold some of them up until the calculation is complete.

Of course the condition codes dependency problem is not a new one and existing techniques can be used to avoid them:

1. Do not have condition codes — e.g. Digital's Alpha [10].

2. Have many copies of them — e.g. Motorola's PowerPC [11].

2.7 Conditional execution

Conditional execution of arithmetic and logical instructions (e.g. as used by the ARM [1]) may be supported by a little extra control logic at each ALU. The extra control logic determines whether the result of the ALU operation is written to the rotary pipeline by controlling the output switch network. A copy of the destination register must be passed around the rotary pipeline in case it is not updated by the conditional instruction but is consumed later on. Conditional updates of the program counter (PC) are performed by the branch unit in a similar manner (see the next section).

2.8 Branches

Branches always have the potential to disrupt pipeline efficiency, so they need to be intercepted early. Non-conditional branches and jumps are independent of the execute stage and may be successfully taken at the decode stage. Conditional branches are always a problem because they are dependent upon the execute stage. In such cases a prediction about the result of the branch may be made and instructions fetched accordingly for speculative execution (see the next section).

The branch still needs to be issued to the execute stage to ensure that the prediction was correct. This may be performed by a specialised function unit.

2.9 Speculative execution

It is essential that speculatively executed instructions can be revoked, i.e. it must be possible to restore the state of the processor after unsuccessful speculative execution. If a register file is used then one can simply ensure that registers are not written until it is known that the results are correct. Alternatively, a larger register file may be used and speculative results written to temporary registers (e.g. as used on the Power PC). Mapping register numbers to physical registers is then a process of colouring, which may be undone if the speculative execution fails.

If a wide rotary pipeline is in use without a register file, then instructions marked as speculative simply prevent the flip-flops from the previous stage from resetting. Thus, the old register values may be reintroduced if the speculatively executed instructions have to be removed.

2.10 Exceptions

Precise exceptions are easy to achieve in a pipeline without a register file because instructions being executed further around the pipeline may be cancelled by resetting the appropriate stages. Registers then revert to the snapshot of registers before the offending instruction.

If a register file is used then all instructions have to be treated as speculative until the preceeding one succeeds. Thus, registers may only be retired to the register file once it is known that the previous instruction has succeeded. This may be relaxed if an imprecise exception model is used for arithmetic operations. Precise exceptions are still required for memory accesses, however, because page and TLB faults need to be intercepted transparently. In this case, only the memory access unit needs to prevent updates reaching the register file.

3 Implementation Issues

3.1 Data encoding and completion detection

There are two principle methods for determining when a logic block has completed an evaluation [7, 9, 5]:

1. embedding a completion signal within the data,

2. localised timing using matched delays.

Whilst the latter can use conventional binary encoding, the former requires a less efficient encoding which includes a completion signal. For example, 1 of 2 or 1 of 4 encoding might be used (see Figure 5). 1 of 2 encoding has the advantage of replacing binary directly but 1 of 4 encoding (radix 4 data) has fewer transitions per cycle (per pair of logical bits) and so consumes less power but without complicating the arithmetic unit.

A completion detection circuit is also required which may be constructed using `OR` gates and a tree of Muller C-elements [2] (see Figure 6). The `OR` gates determine when

code	meaning
00	clear
01	logical 0
10	logical 1

(a) 1 of 2 encoding

code	meaning
0000	clear
0001	logical 0
0010	logical 1
0100	logical 2
1000	logical 3

(b) 1 of 4 encoding

Figure 5: Data encodings

a valid logic encoding has been reached and a tree of C-elements indicates when all logic values are set and then when all are cleared.

Localised timing uses matched delays and bundled data epitomised by Sutherland's Micropipelines [13]. Bundled data just uses binary encoding which uses half as many wires as 1 of 4 encoding. However, inverting a 1 of 4 encoded number is just a matter of swapping wires over where as binary encoding requires inverter gates. Consequently the area difference is not quite as large as one might expect.

Moreover, matched delays require a good margin for error (often at least 50%) when estimating delays due to manufacturing tolerances and any localised thermal effects. This timing margin is directly in the data path so it has a substantial effect on performance. On the other hand, completion detection need only affect when a pipeline stage can be reset for reuse. The completion signal does not need to travel forward on the critical data-path since the data has the completion information embedded in it. Consequently 1 of 4 encoded data with completion detection is faster than bundled data for rotary pipelines.

3.2 Using dynamic logic

Dynamic logic and inverted 1 of 4 encoded data dovetail rather nicely because precharging the dynamic logic corresponds to clearing a 1 of 4 encoded function before evaluation. If dynamic logic is used then the time taken to perform an evaluation is data dependent but the time to reset the circuit is not. This fact can be used to simplify completion detection: replace the tree of C-elements with `AND` gates (actually a `NAND` and `NOR` tree) to detect data completion (as used by Unger [14]), and a delay based on precharging to determine reset completion (see Figure 7).

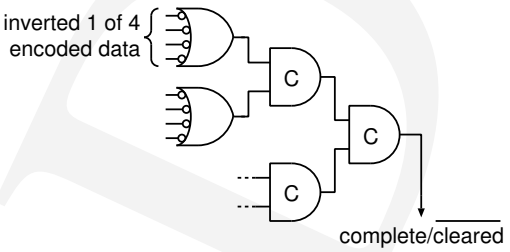

Figure 6: C-element based completion

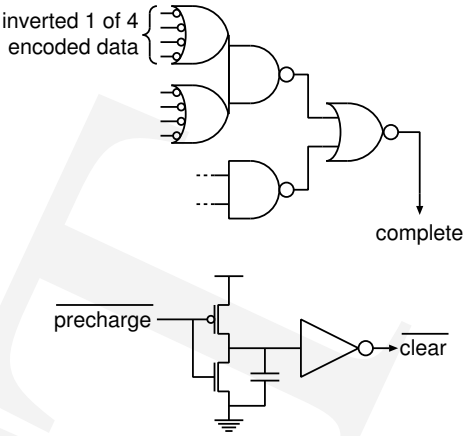

Figure 7: Combining completion detection with delayed clear signal (the capacitor is adjusted to match delays)

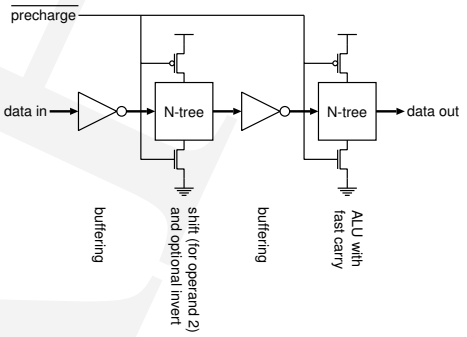

Figure 8: ALU outline

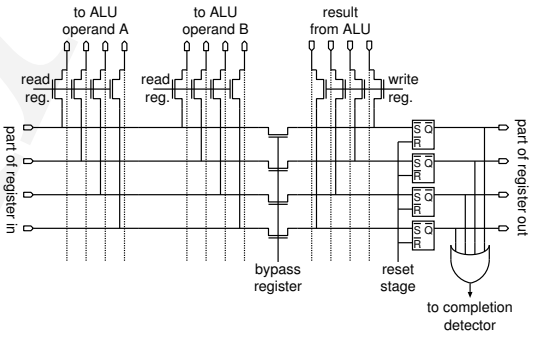

Figure 9: Pipeline stage

3.3 Outline of a stage in the pipeline

Each stage of the rotary pipeline consists of some function unit (e.g. Figure 8) and a switch network (Figure 9) to intercept the appropriate register values and inject results back into the pipeline.

The switch network consists of n-transistors which pass zeros well so an inverted 1 of 4 encoding is used. Banks of transistors select which registers are to be passed as operands to the ALU. Another bank allows register values to be passed through without modification and a further one allows the re-

sult to be injected. Then a bank of SR flip-flops are used to store the result of this stage which is followed by completion detection. The humble SR flip-flop (Figure 10) is used because it is simple and has the right functionality: capture zeros when R is high and reset when R is low.

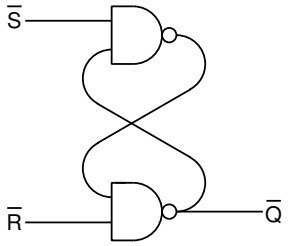

Figure 10: not-S not-R flip-flop

3.4 Controlling the pipeline

Each stage of the pipeline passes through the following states:

- empty — after the ALU is precharged and the flip-flops are reset
- waiting for data — precharge and reset are released
- latching data — SR flip-flops store the result and completion is detected
- precharge — once data has been latched, ALU precharge may commence
- reset — once the next stage has indicated completion (i.e. the data has been consumed) then the latches of this stage may be reset.
- empty — to complete the cycle

A simple implementation of the control structure which embodies the above is shown in Figure 11. This does not include waiting for the instruction to arrive and then being cleared during precharge. However, this may be added by a simple extension to the precharge logic.

3.5 Synthesising complex functions

Complex functions, for example multiply or divide, may be issued as micro-instructions to the multiple ALUs in the rotary pipeline. In this instance the rotary pipeline acts in an analogous manner to Williams' divider [17] — intermediate results flow around as quickly as they are produced.

Multiplies may use a 2-bit Booth's algorithm to multiply two bits per ALU [3]. The only extra hardware required is a two bit shift register which needs to be loaded with the multiplicand and forms part of the control logic. The control logic then issues the correct micro-instruction for each pair of bits in the multiplicand.

Division may be tackled using conventional ALUs by implementing nonrestoring division [6]. A modicum of extra hardware is required for accumulating the quotient bits and

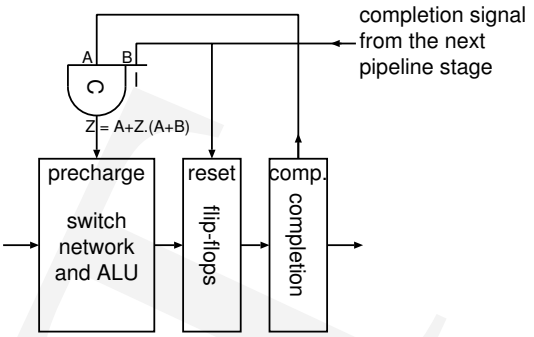

Figure 11: Simplistic pipeline control

extending the control logic to allow the ALUs to make the local decision to add or subtract the quotient depending upon the sign of the previous intermediate result.

3.6 Sizes

The switch network (Figure 12) and banks of SR flip-flops (Figure 13) were laid out using a double metal 1.0μm CMOS process in order to estimate silicon real estate usage. The switch network to intercept A and B operands from one 32 bit 1 of 4 encoded register measures 408μm by 246μm. Thus, if the rotary pipeline transported six registers then it would be 1476μm wide. There are 128 SR flip-flops in Figure 13 with a pitch which matches the switch network. The length is 526μm. Once all of the switch network is complete and joined to the flip-flops the structure is expected to be less than 2mm by 1.5mm. A better CMOS process with more layers of metal would improve these figures.

Figure 12: Switch network excerpt

Figure 13: 128 SR flip-flops

4 Simulation

4.1 Instruction set choice — ARM

Currently our investigations have pivoted around the ARM instruction set [1, 15] because it serves as a good basis for comparison with existing clocked (from Advanced RISC Machines Ltd) and self-timed (from the Amulet group [4, 8]) implementations.

There are three important characteristics of this instruction set which are pertinent to this paper:

1. conditionals — every instruction can be conditionally executed;

2. PC — the program counter is one of the general purpose registers and may be written to, thereby causing a branch;

3. load and store multiple registers in one instruction.

The first characteristic may be efficiently implemented by allowing functions to evaluate unconditionally but only switch the result out onto the rotary pipeline when the condition code has been checked.

The second problem may be resolved by making instructions with PC as a destination a special case. These may be dealt with by the unit that performs conditional branches. Branch prediction is difficult in such situations and it is probably best to simply wait until the correct path has been determined.

Multiple memory accesses can be handled by the decode logic which turns them into a stream of single memory operations. The performance will be limited by memory bandwidth and no use will be made of most of the ALUs in the rotary pipeline.

4.2 Initial results

A discrete event simulation of the rotary pipeline has been undertaken using the ARM instruction set for the programmer's model. *Dhrystone* and *compress* benchmarks were used to test performance. The standard ARM C compiler and GNU CC were also compared because the ARM compiler is targeted towards their commercial single issue clocked processor where as GNU CC is more general purpose and can unroll ARM code to assist multiple issue.

A number of rotary pipelines were tested. Initially just three function units were used: an ALU, a memory access unit and a branch unit. Then further ALUs were added. The results in Figures 14 and 15 indicate the percentage of time when each function unit can be issued with an instruction. As one would expect, adding more ALUs increases performance but with diminishing returns. Like any RISC processor, memory accesses soon limit performance.

Dynamic instruction reordering improved ALU utilisation by around 3% but at the expense of complicating instruction dispatch which slows pipeline refills (e.g. due to exceptions). Performance could be improved by using inlined functions, hardware branch prediction and an architecture with a larger register file [16]. This is left to future studies.

Carry propagation was also measured. For 1 of 4 encoded data the carry had to propagate around 7 2-bit stages for *compress* and 9.8 stages for *Dhrystone*. This compares with the binary (or 1 of 2 encoded) case of 14 stages for *compress* and 20.6 stages for *Dhrystone*. These figures are above the average for random data so fast carry propagation circuits are desirable, although self-timed circuits are not limited by worst-case performance.

5 Relation to other approaches

Rotary pipelines contrast with current superscalar processors by avoiding global communication. We believe that this is important if the designs are to scale.

Rotary pipelines make heavy use of the fact that data can be passed through latches between pipeline stages at any time. This differs from clocked design where latches only pass data in lockstep with the clock. Other asynchronous processors (e.g. Amulet [4, 8] and the CFPP [12]) also exploit the self-timed nature of latches.

Amulet [4, 8] is a single issue processor designed using matched delays. The pipeline structure closely resembles clocked counterparts. However, in situations like pipeline refills the latches between stages can initially be left transparent which allows rapid data transfer.

The counterflow pipeline processor (CFPP) [12] has an unusual pipeline structure where instructions traverse up the pipeline and register values filter down. Instructions pick up, or "garner", operands from the register pipeline and may be executed once all operands have been garnered. The result is kept with the instruction and a copy is also sent down the register pipeline so that it may be garnered by following instructions. Instructions reaching the top of the pipeline have their results retired to a register file.

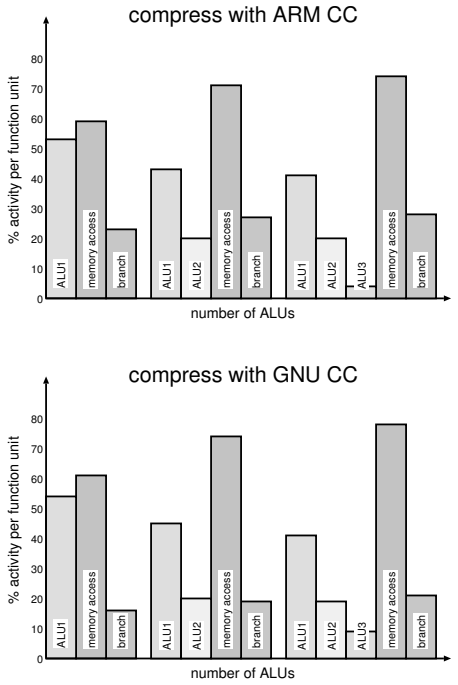

Figure 14: Function unit utilisation for compress

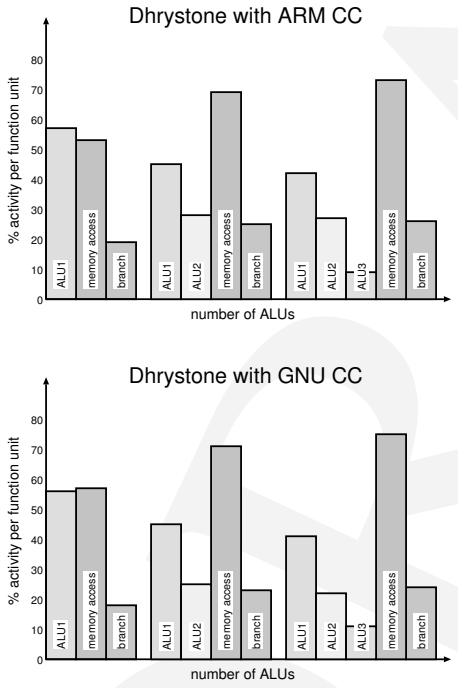

Figure 15: Function unit utilisation for Dhrystone 2.1 (1000 iterations)

The CFPP provides a conceptually elegant method for localising instruction dependency — results are passed quickly down the register pipeline and can be garnered by instructions below. However, flow control in counterflow pipelines is complex, requiring arbiters or "traffic cops". On average at least two arbitrations are required to move an instruction up one stage in the pipeline, which severely limits performance. Furthermore, multiple issue of instructions is difficult, if not impossible.

6 Conclusions

Rotary pipelines have been introduced as a simple and regular self-timed structure to allow efficient multiple instruction issue. Two variations have been investigated: one passes the complete register file around the rotary pipeline and the other only passes the active registers and caches the rest in a register file.

The emphasis was on performance rather than small size or low power. The large number of buses and 1 of 4 encoded data do make the structure fairly large. However, this should be compared with clocked superscalar processors which employ large collections of buses to forward data via bypasses.

Preliminary designs and simulation have been presented to demonstrate the architectural feasibility. Further work to elaborate these is being undertaken.

The design was motivated by the data driven nature of self-timed circuits and would, therefore, be unsuitable for a clocked implementation. Perhaps the future of self-timed circuits lies with new architectural possibilities.

Acknowledgements

This work was funded by EPSRC grant GR/J62708.

References

[1] ARM. *ARM 7TDMI Data Sheet*. Advanced RISC Machines Ltd, 1995.

[2] I. David, R. Ginosar, and M. Yoeli. An efficient implementation of boolean functions as self-timed circuits. *IEEE Transactions on Computers*, 41(1):2–11, 1992.

[3] S.B. Furber. *VLSI RISC architecture and organization*. Marcel Dekker, 1989.

[4] S.B. Furber, P. Day, J.D. Garside, N.C. Paver, S. Temple, and J.V. Woods. The design and evaluation of an asynchronous microprocessor. In *Proceedings of ICCD 94*, pages 217–220, Boston, Massachusetts, October 1994.

[5] S. Hauck. Asynchronous design methodologies. *Proceedings of the IEEE*, 83(1):69–93, 1995.

[6] J.L. Hennessy and D.A. Patterson. *Computer Architecture — A Quantitative Approach*. Morgan Kaufmann, 1996. Appendix A, page A-5.

[7] R.E. Miller. Sequential circuits. In *Switching Theory*, volume 2. Wiley, NY, 1965. In Chapter 10 David Muller's work on speed-independent circuits is reviewed.

[8] N.C. Paver. *The Design and Implementation of an Asynchronous Microprocessor*. PhD thesis, Dept. of Computer Science, University of Manchester, 1994.

[9] C.L. Seitz. System timing. In C.A. Mead and L. Conway, editors, *Introduction to VLSI systems*. Addison-Wesley, 1992.

[10] R. Sites. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[11] J.E. Smith and S. Weiss. PowerPC 601 and Alpha 21064: a tale of two RISCs. *Computer*, 27(6):46–58, 1994.

[12] R.F. Sproull, I.E. Sutherland, and C.E. Molnar. The counterflow pipeline processor architecture. *IEEE Design & Test of Computers*, 11(3):48–59, 1994.

[13] I.E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[14] S.H. Unger. *Asynchronous Sequential Switching Circuits*. John Wiley & Sons, 1969.

[15] A. Van Someren and C. Attack. *ARM RISC chip: A Programmer's Guide*. Addison-Wesley, 1993.

[16] D.W. Wall. Limits of instruction-level parallelism. Technical Report 93/6, Digital, Western Research Laboratory, 1993.

[17] T.E. Williams and M.A. Horowitz. A zero-overhead self-timed 160-ns 54-b CMOS divider. *IEEE Journal of Solid-State Circuits*, 26(11):1651–1661, 1991.