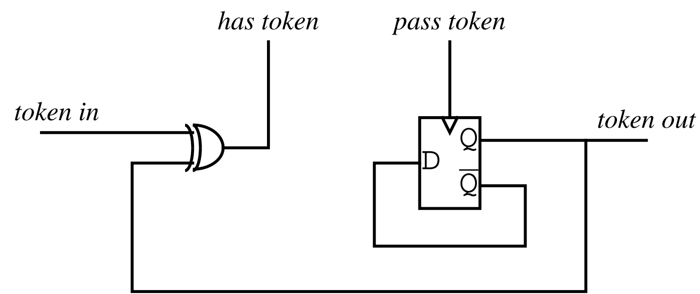
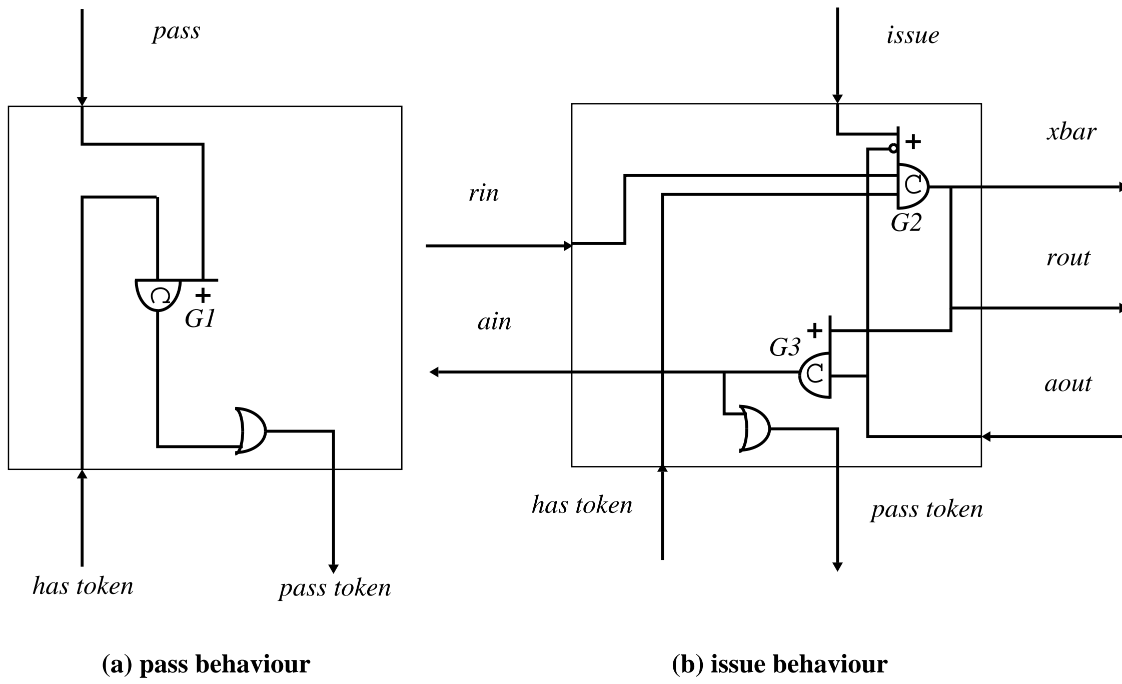


Figure 13: Token Logic Implementation

Block	Cycle Time
Instruction group fetch	117
Instruction issuer (per instruction group)	54
Instruction issuer (per instruction per functional unit)	31
Functional Unit 1 (alu)	54
Functional Unit 2 (move)	70
Functional Unit 3 (register bank)	37
Functional Unit 4 (load/store)	79
Functional Unit 5 (branch)	26

Table 2: SCALP Pipeline Performance



(a) pass behaviour **(b) issue behaviour**

Figure 11: Sequencer Implementation sub-circuits

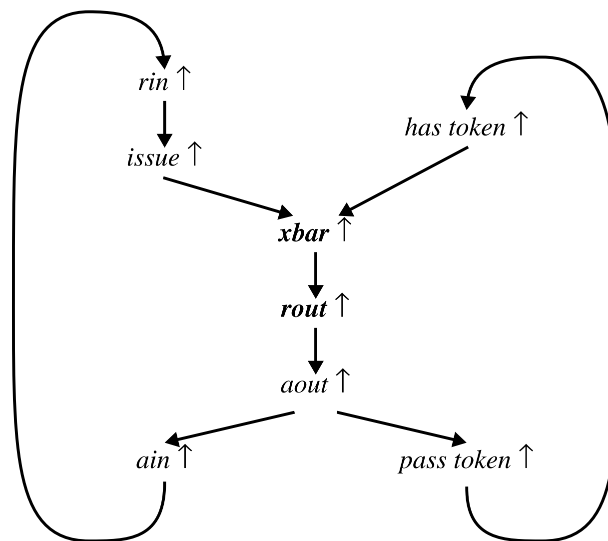
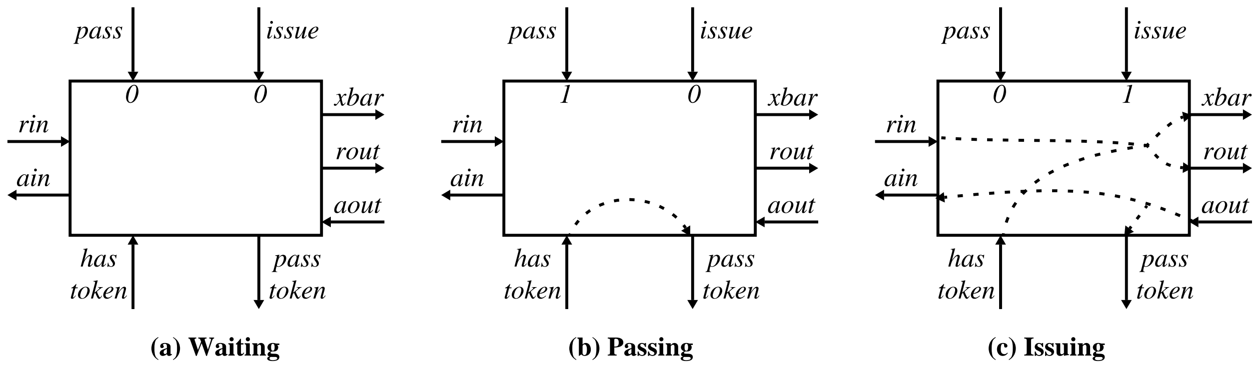
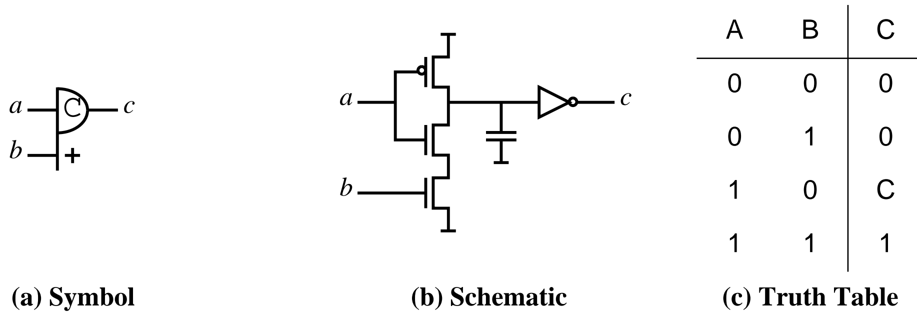


Figure 12: STG for active transitions in issue circuit



(a) Waiting (b) Passing (c) Issuing

Figure 8: Behaviour of the Sequencer



A	B	C
0	0	0
0	1	0
1	0	C
1	1	1

(a) Symbol (b) Schematic (c) Truth Table

Figure 9: Example Asymmetric Muller C Gate

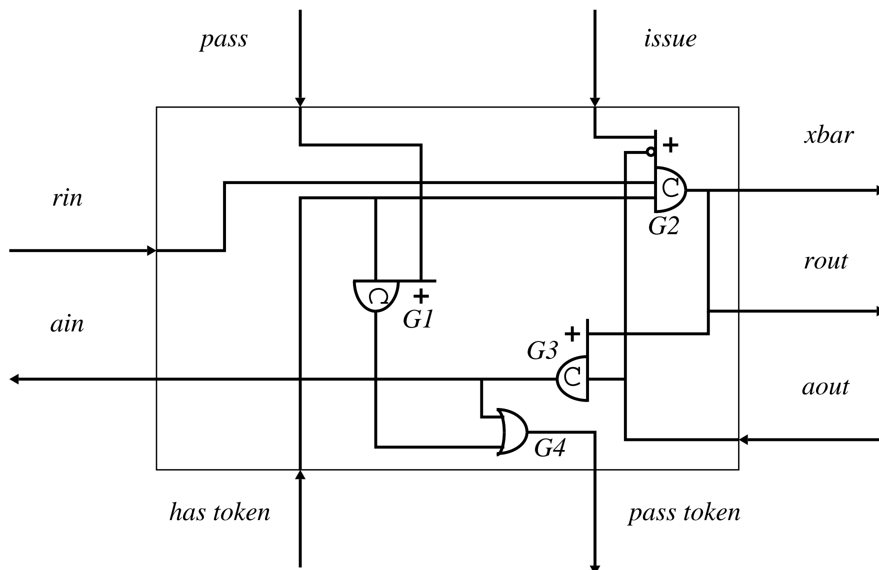


Figure 10: Sequencer Implementation

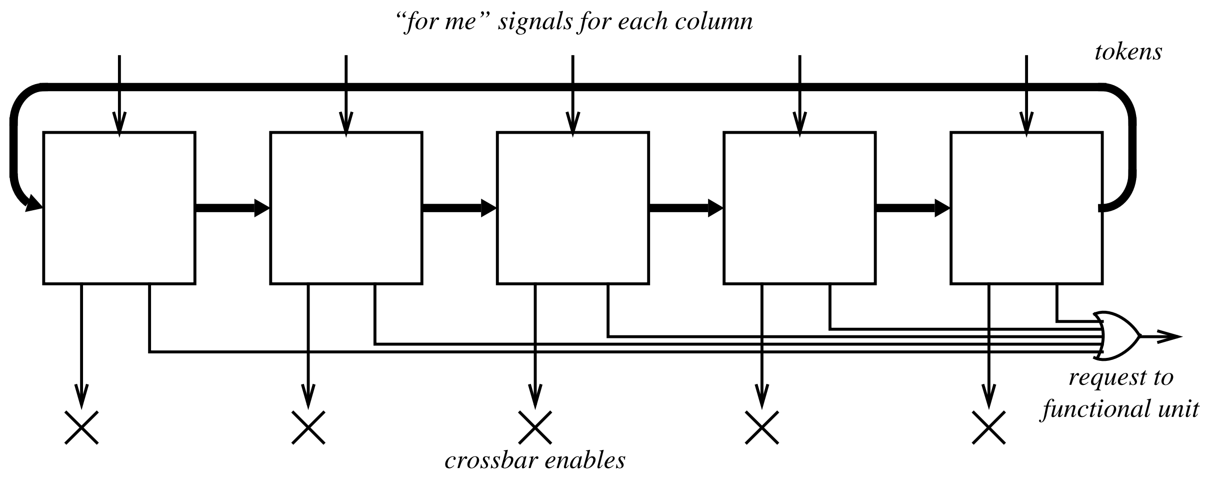


Figure 6: Controller Organisation

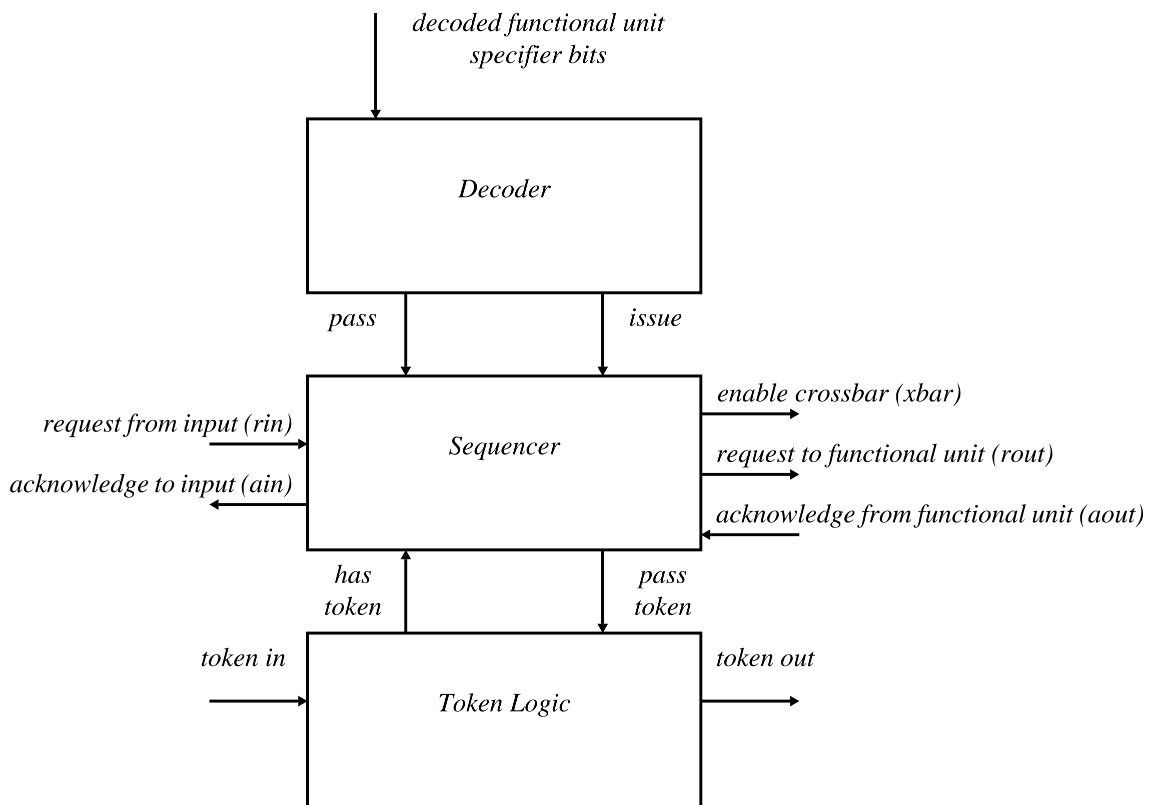


Figure 7: Controller Cell Organisation

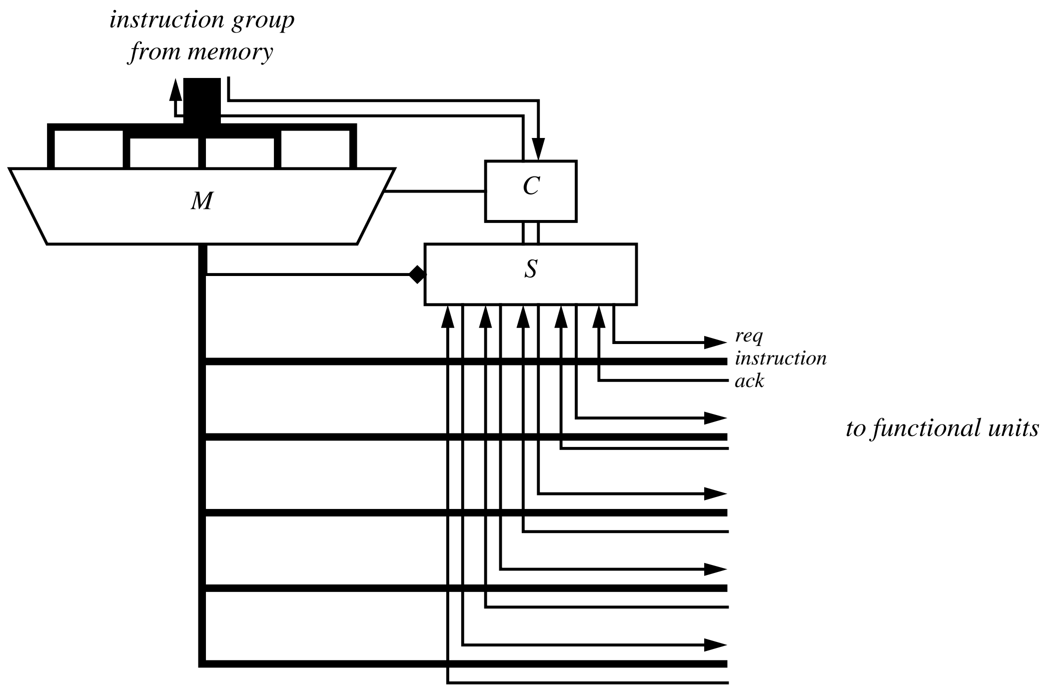


Figure 4: Sequential Instruction Issuer

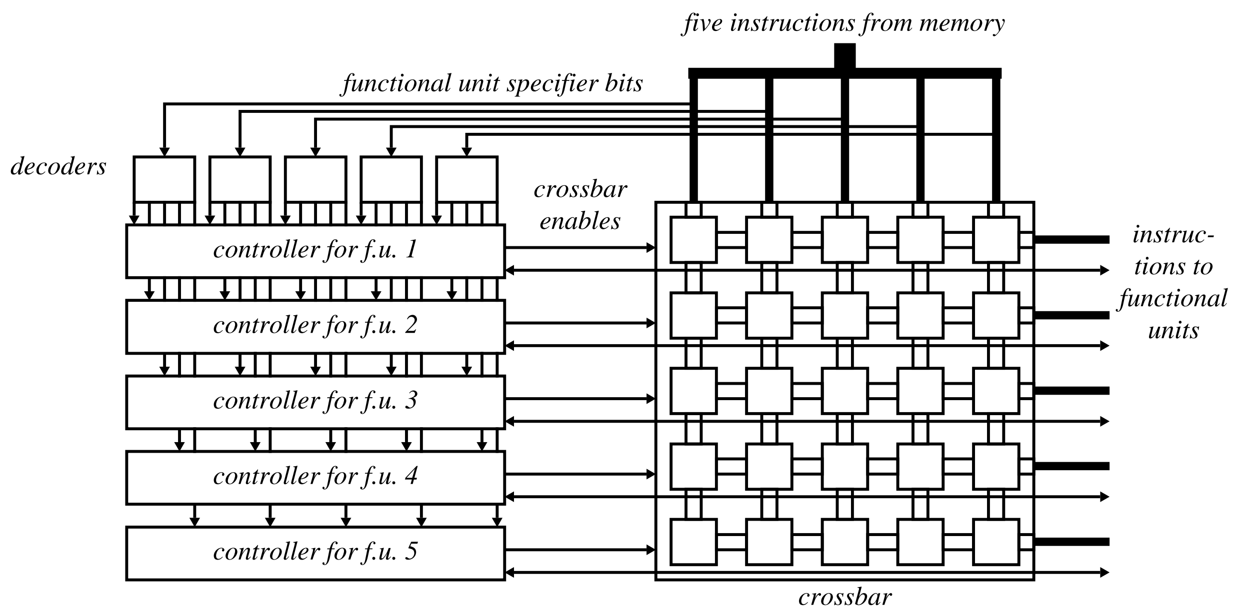
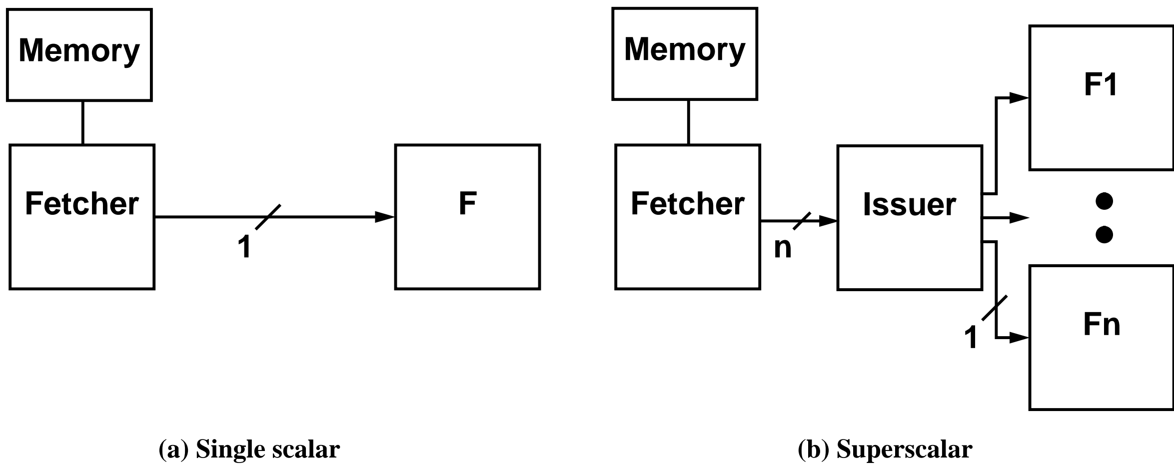


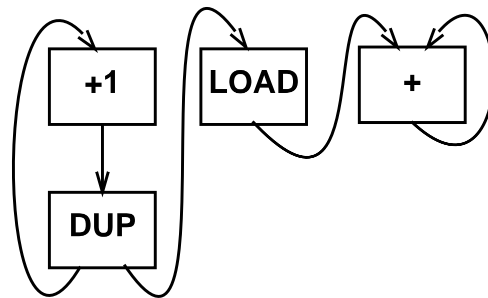
Figure 5: Crossbar Issuer Implementation



(a) Single scalar **(b) Superscalar**

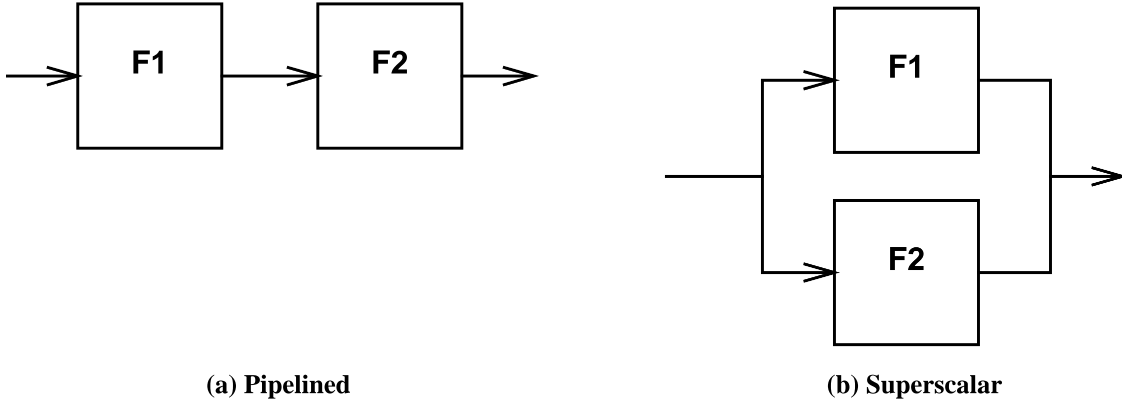
Figure 2: The role of the instruction issuer

```
L1:     ADD 1 -> duplicator
        DUPLICATE -> alu_a, mem_addr
        LOAD -> alu_b
        ADD -> alu_a
        BR L1
```



(a) Listing **(b) Data Flow Graph**

Figure 3: Example SCALP Program



(a) Pipelined **(b) Superscalar**

Figure 1: Pipelined and Superscalar Structures

		Synchronous	Asynchronous
Pipelined functional units	T^{-1}	$= \max(t_1, t_2)$	$\geq \max(t_1, t_2)$
	L	$= 2\max(t_1, t_2)$	$\geq t_1 + t_2$
Superscalar functional units	T^{-1}	$= \frac{1}{2}\max(t_1, t_2)$	$\geq \left(t_1^{-1} + t_2^{-1}\right)^{-1}$
	L	$= \max(t_1, t_2)$	$\geq 2\left(t_1^{-1} + t_2^{-1}\right)^{-1}$

Table 1: Performance of synchronous and asynchronous pipelined and superscalar systems

[5] P. B. Endecott, “Parallel Structures for Asynchronous Microprocessors”, IEEE Technical Committee for Computer Architecture Newsletter, October 1995, pages 11-16. http://www.cs.man.ac.uk/amulet/publications/papers/para_structs.html

[6] P. B. Endecott, “SCALP: A Superscalar Asynchronous Low-Power Processor”, Ph.D. thesis, University of Manchester, 1996, to be submitted. http://www.cs.man.ac.uk/~endecotp/research/phd.html

[7] S. Furber, “Computing Without Clocks: Micropiplining the ARM Processor”, in G. Birtwistle, A. Davis, “Asynchronous Digital Circuit Design”, Springer-Verlag Workshops in Computing Series, ISBN 3-540-19901-2 or 0-387-19901-2, 1995, pages 211-262.

[8] D. Kearney, N. W. Bergman, “Performance Evaluation of Asynchronous Logic Pipelines with Data Dependent Processing Delays”, Proceedings of the Second Working Conference on Asynchronous Design Methodologies, London, U.K., 1995, pages 4-13.

[9] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, P. J. Hazewindus, “The Design of an Asynchronous Microprocessor”, Advanced Research in VLSI; Proceedings of the Decennial Caltech Conference on VLSI, MIT Press, pages 351-373, 1989; also as technical report Caltech-CS-TR-89-02, Computer Science Department, California Institute of Technology, 1989.

[10] N. C. Paver, “The Design and Implementation of an Asynchronous Microprocessor”, Ph.D. Thesis, Department of Computer Science, University of Manchester, U.K., 1994. http://www.cs.man.ac.uk/amulet/publications/thesis/paver94_phd.html

[11] W. F. Richardson, E. Brunvand, “Fred: An Architecture for a Self-Timed Decoupled Computer”, Technical Report UUCS-95-008, University of Utah, 1995. ftp://ftp.cs.utah.edu/techreports/1995/UUCS-95-008.ps.Z

[12] R. F. Sproull, I. E. Sutherland, C. E. Molnar, “Counterflow Pipeline Processor Architecture”, IEEE Design and Test of Computers, Volume 11, No 3, pages 48-59, 1994. Also as Sun Microsystems Laboratories Inc. Technical Report SMLI TR-94-25.

7. Evaluation

A gate level VHDL representation of the entire processor was evaluated by simulation based on relative gate delays approximately matching those in the AMULET1 and AMULET2 designs. The speed of the instruction issuer should be considered in the context of the SCALP processor pipeline; for efficient operation the issuer must be able to maintain the same throughput as the instruction fetch block and the functional units. The speed of each pipeline stage was measured in isolation by simulating it connected between two neighbours with zero delays. Table 2 shows the cycle times (in arbitrary time units) of various parts of the pipeline for typical representative instructions.

From the top part of the table it can be seen that the instruction issuer can keep up with the memory and instruction fetch system. From the lower part it can be seen that the issuer operates faster than all but one of the functional units.

This level of performance comes at a cost: the issuer contains a total of over 4000 gates, making up about 40% of the processor as a whole. This size also contributes significantly to the power consumption of the processor. In comparison an implementation based on the sequential design of figure 4 would use a few hundred gates but would form a performance bottleneck.

Importantly the design has very regular structure; there are many identical cells each containing a few dozen gates. This fact would make a compact VLSI implementation much easier as the individual cells could be carefully designed and then abutted in a rectangular arrangement.

8. Conclusions

The instruction issuer is the key part of the SCALP processor. Without an efficient design, simultaneous execution of instructions in multiple functional units would be hard to achieve. This instruction issuer design is more than fast enough to supply instructions to SCALP's five functional units, ensuring that the speed of the functional units themselves limit the processor's performance.

The design of the SCALP processor has proved a valuable exercise in asynchronous design. In particular the nature of the instruction issuer made its implementation very challenging and interesting. The design trade-offs leading to the token passing scheme described here were quite different from those that would apply in a synchronous design.

The resulting design is robust as it is based on the constrained design styles described in section 4. It performs well and offers the potential for an efficient VLSI implementation.

Automatic synthesis tools were not used for this design. Though these tools do offer fast and reliable synthesis from specifications such as STGs the hard part of the process is deriving this specification. This problem may be used in future work as a benchmark example for evaluation of such tools.

Finally it is interesting to compare this design with the equivalent synchronous problem. Much of the complexity of this design is a result of the fact that the block has a large number of inputs and outputs operate that all operate asynchronously with respect to each other. The task of the instruction issuer's control is therefore to enforce the necessary temporary synchronisation between inputs and outputs so that data can be transferred. This problem is totally absent in a synchronous processor.

9. References

[1] K. van Berkel, R. Burgess, J. Kessels, F. Schalij, A. Peeters, "Asynchronous Circuits for Low Power: A DCC Error Corrector", IEEE Design and Test of Computers, Volume 11, No 2, 1994, pages 22-32.

[2] E. Brunvand, "The NSR Processor", Proceeding of the 26th Annual Hawaii International Conference on System Sciences, pages 428-435, Maui, Hawaii, 1993.

[3] A. P. Chandrakasan, A. Burnstein, R. W. Brodersen, "A Low-Power Chipset for a Portable Multimedia I/O Terminal", IEEE Journal of Solid State Circuits, Volume 29, Number 12, December 1994, pages 1415-1428.

[4] P. Day, J. V. Woods, "Investigation into Micropipeline Latch Design Styles", IEEE Transactions on VLSI, Volume 3, Number 2, June 1995, pages 264-272.

- If *pass* is asserted then as soon as *has token* becomes asserted then *pass token* is asserted.
- If *issue* is asserted then when *has token* and *rin* are asserted the crossbar enable and functional unit request signals are asserted. When the functional unit has accepted the instruction and asserted its acknowledge the acknowledge to the input is asserted and the token is passed on.

The implementation of the sequencer makes use of asymmetric Muller C gates. An example C gate symbol, simplified transistor implementation and truth table are shown in figure 9. The behaviour of these devices is as follows: when all of the inputs connected to the main body of the gate and any inputs connected to the arm labelled + are high the output becomes high. When all of the inputs connected to the main body of the gate and any inputs connected to the arm labelled - are low the output becomes low. At other times the output is unchanged.

The sequencer circuit is shown in figure 10. It is made up from two subcircuits as shown in figure 11; G1 implements the behaviour for pass as shown in figure 8(b), and G2 and G3 implement the behaviour for issue as shown in figure 8(c). G4 simply merges the two possible sources of the *pass token* signal.

The pass subcircuit of figure 11(a) is straightforward. When both *has token* and *pass* are asserted, *pass token* is asserted. When *has token* is deasserted, *pass token* is deasserted. It is not possible to simply use an AND gate for G1 because the decoder logic does not guarantee that *pass* will remain asserted after *pass token* has been asserted.

The issue subcircuit of figure 11(b) is more complex. Figure 12 shows a signal transition graph (STG) for part of its behaviour. This STG has been simplified by omitting the return-to-zero transitions. From this STG the three non-inverting inputs of G2 can be explained: *xbar* and *rout* must be asserted when *rin*, *issue* and *has token* are all asserted.

The inverting input to G2 from *aout* is required to prevent this cell from starting to issue an instruction when some other cell in the same controller is still in the process of issuing some previous instruction. G3 is required to prevent *ain* from being asserted when another cell issues an instruction: *ain* is only asserted when this cell's *rout* caused *aout* to be asserted.

Implementation of the token logic was interesting. The initial implementation used the four phase protocol to transfer the token between blocks, but it was subsequently found that unusually the two phase protocol leads to a simpler implementation. In the two phase protocol both rising and falling transitions on a signal indicate an event, so there is no need for return to zero phases. The implementation of the token logic is shown in figure 13. When a new token arrives the input is inverted and the output of the exclusive OR gate becomes high. When *pass token* is asserted the D flip-flop inverts the value of the output, sending the token to the next block and also causing *has token* to become low again by inverting the other input to the exclusive OR gate.

6. Additional Complexity

For simplicity two features of the instruction issuer are not presented in detail here:

- **Branch instructions:** the issuer must treat branch instruction specially by waiting for the branch unit to confirm whether they were taken or not before proceeding. When the branch is taken following instructions up to a specially marked target instruction must be discarded by the issuer. To accomplish this, tokens are labelled as "normal" or "delete" tokens. Taken branches transform tokens from normal to delete mode and branch targets switch back again. When a delete token is received the corresponding instruction is acknowledged without being issued.

  Note that instructions are prefetched beyond branches and untaken branches do not disrupt the progress of execution. Although the present SCALP implementation does not implement branch prediction this is not a restriction imposed by the instruction issuer. This issuer design could support speculative prefetching of predicted branch target instructions, with transfer to functional units delayed until correct prediction was confirmed.

- **Variable length instructions:** SCALP has two lengths of instructions, those with and those without immediate values. Those with immediate values occupy two "columns" of the issuer. The issuer has to skip over those columns containing immediate values, and does so by extending the token logic to allow tokens to be passed to either the following cell or the cell after next. The decoding a sequencing blocks are also extended to support this.

Dealing with these complexities adds to the size of the issuer and reduces its speed. More details of the implementation can be found in [6].

The choice of design style for a particular system must be a trade off between the extra design effort required by the less constrained styles and the potential inefficiency of the more constrained styles. The extra effort is most likely to be justified where small elements are replicated in a regular structure, for example in a datapath. In blocks of random logic such as control circuitry the simplicity of the more constrained styles is more appropriate. Consequently this design uses a bounded delay datapath with primarily speed-independent control logic.

5. Instruction Issuer Design

Despite the simplifications introduced by the architectural ideas mentioned in section 3, the design of the asynchronous instruction issuer is not straightforward: simple implementations cannot match the throughput of other stages in the processor pipeline. Consider the design shown in figure 4. Instructions are fetched from memory in groups of five at a time (five 12-bit instructions are stored in a 64-bit memory word). In this scheme each of these instructions is selected in turn by means of the multiplexer M and counter C. The select block S is controlled by the functional unit specifier bits of the selected instruction, and sends a 'request' to the appropriate functional unit. Once that functional unit has accepted the instruction and asserted the acknowledge signal the next instruction can be considered. The time taken to issue one group of five instructions using this circuit is at least five times greater than the time for a functional unit to accept one instruction. This makes the instruction issuer a bottleneck, outstripped by the functional units themselves.

The alternative is to use a more highly connected scheme. One option is a crossbar switch, allowing all possible instructions from the five instruction group to be simultaneously connected to all possible functional unit inputs. In this way several instructions may be issued to different functional units concurrently. The VLSI implementation of a crossbar structure can be very regular and hence area-efficient, making it more attractive than less highly connected intermediate schemes.

The use of a crossbar allows for much higher performance than the sequential issuer, but in order to use the available performance a complex control system is required. The overall scheme is shown in figure 5. Each column of the crossbar is connected to one instruction from the input group and each row is connected to one functional unit. The control is divided into one controller for each functional unit, responsible for controlling one row of the crossbar. Other logic decodes the functional unit specifier bits for each instruction and indicates to each controller whether or not each instruction is for that functional unit. This allows the five controllers to be internally identical.

The design of the controller is made more complex by the possibility that an instruction group may contain more than one instruction for each functional unit, and that in this case they must be issued in program order. This suggests a sequential implementation for the controller, but a fully sequential implementation would be too slow. Instead a hybrid implementation is used.

The internal organisation of each controller is as shown in figure 6. For each column the controller decides what action it would take if that column were the 'current' column. The possibilities are

- to issue that instruction to this functional unit,
- to skip over that instruction and consider the next because this instruction is for another functional unit,
- to wait because the next instruction for this column has not arrived.

The controller then makes each column in turn the 'current' column by passing a token between them. Since the action to be taken on arrival of the token is precomputed, in the common cases the token can be passed very quickly.

Figure 7 shows the internal structure of the controller cells. The cell is divided into three parts:

- The decoder decides which of the three actions mentioned above (issue, pass, wait) will be carried out when this becomes the current cell.
- The sequencer controls the actions that occur when the token does arrive.
- The token logic is responsible for receiving and transmitting tokens to and from neighbouring cells.

The decoder logic is a hazard free implementation of a simple combinational function and is not discussed further here.

The sequencer behaves in one of three ways under control of the *issue* and *pass* signals as shown in figure 8:

- When neither is asserted, nothing happens i.e. the cell waits for some condition to change, even if the token arrives.

The alternative is to shift the responsibility for this task from the hardware to the compiler. SCALP does this by means of an instruction encoding with three fields:

- **The functional unit specifier** which indicates which functional unit will execute this instruction.
- **The opcode** which indicates which operation will be carried out by that functional unit.
- **The destination specifier** which indicates to which other functions unit's input the result of the operation should be sent.

Instructions then execute as follows:

- The issuer distributes the instructions to the appropriate functional units according to their functional unit specifier bits. It need not consider any other part of the instruction.
- The individual functional units consider the opcode of the instruction to identify which operation to carry out. They wait for any necessary operands to arrive at their operand inputs.
- Once a result has been computed it is sent to its destination as indicated by the destination specifier part of the instruction. One possible destination is a global register bank which is used for longer term storage.

Note that by separating the functional unit specifier bits from the opcode the amount of decoding that has to be carried out by the instruction issuer is minimised, the task being moved to the individual functional units.

As an example of this style of programming, consider the code fragment shown in figure 3(a). The program is also represented by a data flow graph in figure 3(b). The function of the program is to sum the values in an array. On each iteration of the loop the address pointer is incremented (first line of the listing, and left hand loop of the graph), and one copy of the new address is sent to the memory unit. This performs a load, sending the result to the ALU where it is added to the running total.

SCALP uses a 12 bit instruction format with 2, 7 and 3 bits for the functional unit specifier, opcode and destination specifier respectively. This is a significant saving over a conventional architecture with perhaps three 5-bit register specifiers, and helps to increase SCALP's code density. This reduces the amount of instruction fetch activity, increasing power efficiency.

One potential area of concern is that of deterministic order of execution; specifically, if two instructions executing concurrently send their results to the same destination, which will arrive first? The solution adopted is to provide a basic sequencing operation and to require the compiler to introduce it as and when required. Further details of the SCALP instruction set architecture and its properties can be found in [6].

4. Style of Implementation

The term 'asynchronous' is a very broad one, encompassing any type of design that does not use a global clock. The totally unconstrained design style that this implies is prone to problems such as hazards and races and rightly leads to a suspicion of asynchronous design by traditional synchronous designers. In contrast the design styles used here are highly constrained. For example the standard inter-block communication protocol is the four-phase bundled data protocol as earlier work [4] for the AMULET designs identified this as the best protocol in terms of power, speed and area.

There is a range of possible circuit level design styles that constrain the designer in different ways. The fewer assumptions are made about the behaviour of the circuit the more constraints must be imposed on the designer. For example in a delay insensitive design delays through gates and wires are assumed to be unbounded but finite. Circuits designed to work under these assumptions will function correctly irrespective of real gate and wire delays and operating conditions, but may be more complex than equivalent designs using less constrained styles.

Speed independent designs are a superset of delay insensitive designs that assume that wires act as equipotentials, i.e. all points change simultaneously. This assumption may be reasonable for smaller circuits, and leads to less complex implementations than delay insensitive design. In SCALP, control circuits are generally built from delay insensitive interconnections of 'macromodules', each of which is internally speed independent.

A still less constrained design style is the bounded delay design. A bounded delay design functions correctly only under certain specified assumptions about upper and lower bounds on the delays through gates and wires. Bounded delay design requires less logical overhead, but the delay assumptions have to be verified by means of detailed timing analysis and simulation.

Other forms of parallelism such as pipelining have the same potential benefits, but in the case of an asynchronous implementation the benefit of superscalar parallelism exceeds that of pipelined parallelism. This advantage is the subject of section 2.

In conventional architectures the advantages of superscalar operation come at the cost of significantly increased complexity. SCALP has an unusual architecture, described in section 3, that simplifies this problem.

The remaining sections of this paper describe the design, implementation and evaluation of the key component of the SCALP processor, the instruction issuer.

2. Superscalar versus Pipelined Parallelism

It has been observed that pipelined parallelism extracts less of the potential benefit of asynchronous implementation than superscalar parallelism [5]. Figure 1 shows how a function can be divided into two sub-functions F1 and F2 either as (a) a pipeline or (b) with a superscalar arrangement. The latency L (time from one item entering the block to leaving) and throughput T (rate at which items are processed) of the two arrangements for synchronous and asynchronous operation are summarised in table 1 (t_1 and t_2 refer to the delays in F1 and F2 respectively). In all synchronous cases the performance is determined by the clock speed which has to match the greater of t_1 and t_2.

For an asynchronous pipeline the latency is determined by the sum of the delays. On the other hand all data has to pass through all blocks including the slowest block, meaning that the throughput is still limited by the slowest block. With the superscalar arrangement this is not the case as slow operations in one block can be overtaken by faster operations in the other block, leading to both latency and throughput determined by typical performance. In this sense superscalar parallelism is preferable to pipelined parallelism for asynchronous implementation.

Note that in table 1 the expressions for the asynchronous systems are expressed as upper bounds on the performance. This is because these expressions do not take into account the effects of *starvation* and *blocking*, that is the situations when one block cannot proceed because the preceding stage cannot provide data to it (starvation) or the following stage cannot accept data from it (blocking). This effect is described in [8] and can be significant.

3. The SCALP Architecture

The main additional requirement for a superscalar processor is a superscalar instruction issuer (figure 2). Upstream from the issuer the instruction fetch logic is similar to that of a conventional processor except that it fetches groups of several instructions simultaneously. Downstream from the issuer the functional units are generally similar to the execute pipeline of a conventional processor. The issuer is a new block responsible for distributing the fetched instructions to appropriate destinations.

Each group of fetched instructions may contain any arbitrary mix of instructions for the functional units. When there are several instructions for one functional unit they must be sent to the functional unit in program order. Each functional unit then requires some sort of queue to store the issued but not executed instructions.

The instruction issuers in conventional superscalar processors are made complex by the need to detect dependencies between instructions and to control the order of issue of instructions and the routing of results and operands in response. This is largely because conventional superscalar processors use the same basic execution model as single scalar processors, and the potential concurrent execution of several instructions must be hidden from the programmer's model. SCALP employs a new architecture designed to avoid this problem and hence simplify the implementation.

Conventional architectures use a global shared register bank for communication of data from one instruction to another. In general one instruction writes its result into a register from which it is subsequently read as an operand by a following instruction. In a superscalar processor this global shared register bank needs one port for each operand and results of each functional unit.

This register bank model works correctly when the production and consumption of data is separated by many instructions. On the other hand in a pipelined or superscalar processor it is often the case that an instruction will require as an operand the result of an instruction that has yet to be written to the register bank. To allow this operation *forwarding buses* are used to link the various pipeline stages in each of the functional units. One of the functions of the instruction decoding and issuing process in a conventional processor is to compare the register numbers in all pending and executing instructions and to activate any necessary forwarding buses. Finding all of these dependencies can be a very complex operation.

Superscalar Instruction Issue in an Asynchronous Microprocessor

Philip B. Endecott

Department of Computer Science, University of Manchester, Manchester, M13 9PL, U.K.

pbe@cs.man.ac.uk

Abstract

This paper describes the implementation of the instruction issuer for a superscalar asynchronous microprocessor, SCALP, as a case study in asynchronous design. The issuer accepts five instructions at a time from the memory interface and issues them out of order to five parallel functional units. SCALP's architecture is designed to reduce the complexity of the instruction issuer by removing the need to detect dependencies between instructions as they are issued. The design has a regular cellular structure suitable for VLSI implementation. Its performance is sufficient that it does not form a bottleneck in the SCALP pipeline.

1. Introduction

A number of asynchronous microprocessor designs have been proposed recently. These include the AMULET1 and AMULET2 processors [10][7], the Counterflow Pipeline Processor [12], the NSR processor [2], Fred [11], and the Caltech Asynchronous Microprocessor [9]. By operating asynchronously these processors aim for a number of benefits in comparison with conventional synchronous architectures:

- **Increased performance:** the performance of a synchronous system is bounded by the worst case delay in the slowest processing block; asynchronous systems are often governed by typical delays.
- **Power saving:** synchronous systems may consume more power than is necessary in applications with a variable computational demand as the clock signal and the logic driven by it will continue to operate even when there is no useful work to perform. Furthermore many asynchronous systems employ schemes where every transition is meaningful i.e. there are no "wasted" transitions or glitches that consume unnecessary energy [1].
- **Modular design:** asynchronous systems are generally built using well-defined local communication between blocks, making it easier to re-use modules than is the case with synchronous systems.
- **No clock problems:** the absence of a global clock signal eliminates the problems of clock generation and distribution and of clock skew.

These recent asynchronous processors incorporate many of the features of advanced synchronous architectures including cache memories, 32 bit datapaths, large register banks, pipelining, branch prediction, dependency detection and forwarding. However none of these previous designs has implemented a *superscalar* processor; that is one capable of issuing more than one instruction in parallel to multiple functional units. Both the Fred processor and the Caltech Asynchronous Processor have multiple functional units but they issue instructions serially, making the instruction issue process a potential bottleneck.

The SCALP processor (Superscalar Asynchronous Low-Power Processor) [6] is the first processor design of this type. The potential benefits of superscalar operation include increased performance and increased power efficiency. Performance is increased because the number of instructions completed per unit time increases. To increase power efficiency rather than performance the supply voltage must be reduced when the parallelism is introduced. Power efficiency is increased because the speed of the logic reduces linearly with the voltage whereas the energy per transition reduces with the square of the voltage [3].