

# **ADDRESS GENERATOR SYNTHESIS**

*by*

**Douglas M. Grant**

A thesis submitted to the Faculty of Science,  
University of Edinburgh, for the degree of  
Doctor of Philosophy

Department of Electrical Engineering,  
1991



# **Abstract of Thesis**

Increasing complexity of Application Specific Integrated Circuits (ASICs) has demanded a corresponding increase in the power of Computer Aided Design (CAD) tools, so that contemporary design tools can now synthesise an entire silicon architecture, given only a description of its functionality. Specialised automated synthesis techniques have now been applied to almost all parts of the architecture, but one area which remains unresolved is that of memory address generators.

Previously combined with other logic synthesis techniques, less than optimal solutions were often found for generating memory address sequences, and this thesis examines address generator synthesis as an individual step in the design process, as part of an investigation into high level synthesis. The synthesis techniques developed for address generators in the AG1 and AG2 tools presented, target specific architectural forms including counters, incrementors and ROM look-up tables, and the details of these are gathered within a comprehensive data structure which allows optimisation through hardware sharing to occur.

At a slightly higher level, the specification of address sequences as a stage in memory synthesis is also investigated and a behavioural to register-transfer level silicon compiler, MC<sup>2</sup> is presented. The data path and memory architectures constructed by this tool are used to produce realistic address generation requirements whose implementations are also presented, synthesised by AG2.

It is shown that both array and non-array memory can benefit from more specialised address generator synthesis over the existing, mainly logic synthesis approach.

## **Declaration of Originality**

The material contained herein was researched and composed entirely by myself in the Department of Electrical Engineering at the University of Edinburgh, between October 1988 and October 1991.

## **Acknowledgements**

I would firstly like to thank my supervisors, Peter Denyer and Peter Grant for all their help and advice throughout my time at Edinburgh, all the guys from the SARI project, BAe for their sponsorship, Fiona from the TTC and Joan Burton for their help with all things complicated, Iain Finlay, Paul Neil, Jonathan Puddicombe and Hamish Fallside for putting up with my ranting in the office, and most of all thanks to Tracy my wife-to-be for all her support during those late night brainstorming sessions.

## Contents

Abstract .....	I
Declaration of Originality .....	II
Acknowledgements .....	II
Contents .....	III
Preamble .....	1
<b>Chapter 1: Introduction to address generation .....</b>	<b>2</b>
Address Generation in Digital Systems .....	2
What is address generation? .....	5
The scope of address generation .....	5
Overview of address generation techniques .....	6
For control .....	6
For memory access .....	6
Overview of address generator synthesis techniques .....	7
For control .....	7
For memory access .....	7
Other related approaches .....	8
The case for address generators based on counters .....	8
Comments .....	10
<b>Chapter 2: Introduction to address generator architecture .....</b>	<b>11</b>
The memories .....	11
The binary counter .....	12
The ripple counter .....	12
The serial carry counter .....	12
The parallel synchronous counter .....	13
The serial/parallel synchronous counter .....	13
The pseudo-parallel synchronous counter .....	14
The modulus m counter .....	14
Other address generator elements .....	16
Address ROMs .....	16
Exclusive OR gates .....	18
Clocked-bits .....	19
Incrementors .....	19
Logic .....	20
Cost breakdown of address generator elements .....	21
Area-costs .....	21



Speed-costs .....	22
Comments .....	22
<b>Chapter 3: Requirements for an address generator .....</b>	<b>23</b>
Data-dependant addressing .....	23
Scheduled memory addressing .....	24
Array access .....	24
Control .....	26
Comments .....	26
<b>Chapter 4: Address generation based on binary counters .....</b>	<b>27</b>
Some traditional problems .....	27
Some manually designed address generators.....	31
AG1 - Address generator synthesis based on binary counters .....	33
Data entry types.....	34
Method .....	34
Logic synthesis .....	37
Output format .....	40
Address generators designed using AG1 .....	41
Comparisons .....	41
Use of the 'C' programming language .....	44
Comments .....	44
<b>Chapter 5: Introduction to behavioural synthesis .....</b>	<b>46</b>
What is behavioural synthesis? .....	46
Key steps in the high level synthesis process.....	47
Capture of behaviour .....	48
Scheduling .....	49
Resource Allocation .....	51
Data path synthesis .....	51
Controller synthesis .....	55
Resulting design format .....	55
Impact on address generation .....	56
Comments .....	57
<b>Chapter 6: A heuristic approach to memory, communication and control synthesis for scheduled algorithms .....</b>	<b>58</b>
The joy of synthesis! .....	58
Schedules and their scheduling method .....	58
Constraints on this approach .....	62
MC <sup>2</sup> - Memory, Communications and Control synthesis .....	62

Schedule data-base .....	62
Pre-assignment or not? .....	65
The Three Steps to heaven .....	65
Memory and communications synthesis .....	65
Address and control requirement analysis .....	78
Address and control sequence synthesis .....	83
Output format .....	86
Some synthesised data-path architectures .....	88
Comparisons with related results .....	91
A Standard for behavioural synthesis results presentation .....	95
Prolog for fast development .....	96
Comments .....	96
<b>Chapter 7: A general approach to address generator synthesis .....</b>	<b>99</b>
The need for generalisation .....	99
The inevitable data model .....	99
Requirements of an address generator synthesis tool .....	102
AG2 - A general address generator synthesis tool .....	102
Input format .....	102
Basic Method .....	105
A working example .....	106
Method .....	108
Finding an Incremental Sequence .....	108
Padding a Bit Sequence .....	109
Transformation to Repetition Sequence .....	110
Reducing the Repetition Sequence .....	111
The Repetition Sequence Characteristic .....	113
Matching the Characteristic to a Bit Sequence Generator...	114
Finding Clocked-type Bit Sequence Generators .....	115
Multiple Access Sequences .....	116
Optimization .....	116
Output format .....	119
Other worked examples and results .....	120
ADA - A big step .....	131
Comments .....	131
<b>Chapter 8: Address generator synthesis as part of a general behavioural</b>	
<b>    synthesis toolset .....</b>	<b>133</b>
Introduction to SAGE - Concepts and Reality .....	133
Address generation within SAGE .....	136

Scheduled memory .....	136
Array memory .....	137
Macro-generation of counters .....	138
Future plans .....	139
Comments .....	139
<b>Chapter 9: Conclusions and new directions .....</b>	<b>140</b>
References .....	144
Appendix A - Author's publications .....	160
Appendix B - Example output from AG2 .....	194
Appendix C -Workplan .....	213
Appendix D - User Guides .....	214
Appendix E - Address generators for MC <sup>2</sup> examples .....	
Also included: 1 disk containing all code.	

## Preamble

This thesis examines high level synthesis of data path architectures and in particular, address generation hardware. This is based upon the recognition of certain characteristics of binary sequences which relate directly to certain generation methods. The first chapter introduces the concept of address generation, and gives an overview of related work in this area, and then the motivation for this work is stated. Chapter 2 introduces the hardware involved in address generation, including counters, single logic gates and ROMs, and the next chapter illustrates the possible requirements for an address generator, before Chapter 4 looks at a simple address generator synthesis tool - AG1.

Next, in Chapter 5, the many and varied approaches to contemporary behavioural synthesis are documented, and then Chapter 6 presents a heuristic approach to memory, communication and control synthesis for scheduled algorithms - MC<sup>2</sup>, which has been coded in Prolog. This step was necessary in order to produce some realistic address sequences for otherwise well-known examples.

Chapter 7 returns to address generator synthesis with a more complex address generator synthesis tool - AG2, and in Chapter 8 its application within a general behavioural synthesis tool is investigated. Finally, Chapter 9 presents conclusions and directions for future work.

Appendix A contains the author's publications and then Appendix B presents some annotated output from the tool described in Chapter 7. Appendix C gives account of how time was spent on this project, and Appendix D holds the user guides to the tools presented in this thesis. Also included with this thesis is a disc which holds all the code described in the text.

# Chapter 1 Introduction to Address Generation

## 1.1 Address Generation in Digital Systems

Since the first digital systems were constructed in the mid-1930s [200], great advances have been made in both their design and physical implementation. From the earliest vacuum tube transistors used by Newman and Pinkerton [201, 202], evolved solid-state logic [203], and with that came the first great improvement in the size, power consumption and reliability of digital circuit components.

As the properties of silicon as a substrate for both transistors and interconnect were developed in the early 1950's, a second step was taken in circuit performance, so that we now have highly complex micro-computers and data-processing hardware, integrated on a single chip (IC), less than three square centimetres in area.

Figure 1.1 shows a typical Von Neumann data-processing architecture [204], in which data is guided between computational hardware and memory of some sort, under the direction of a process controller and ancillary hardware.

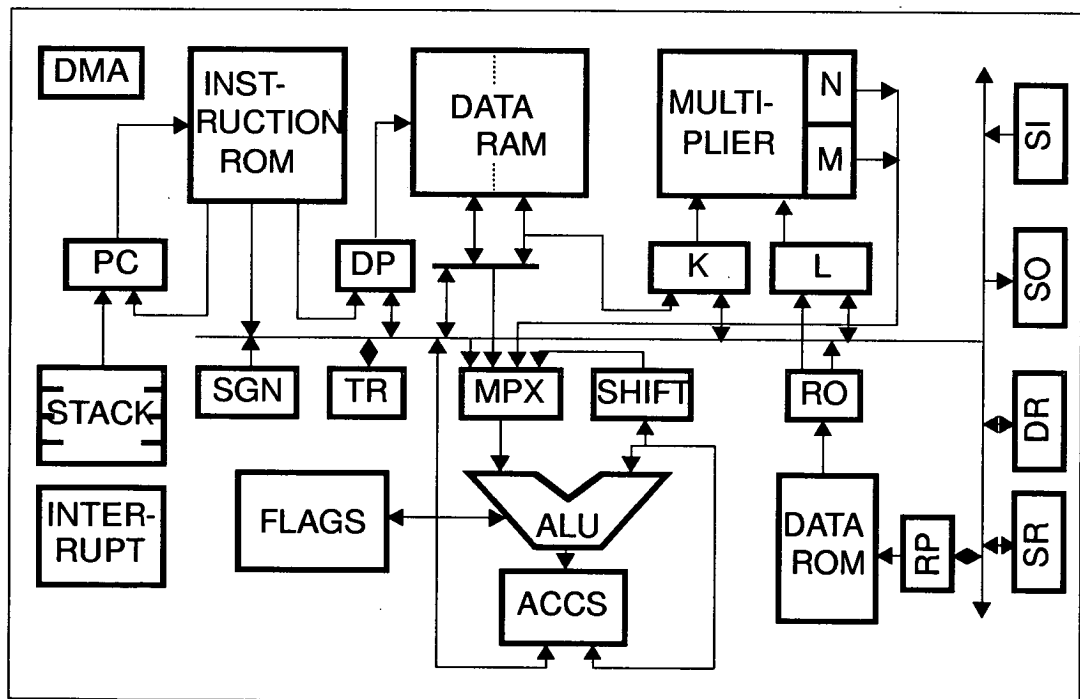


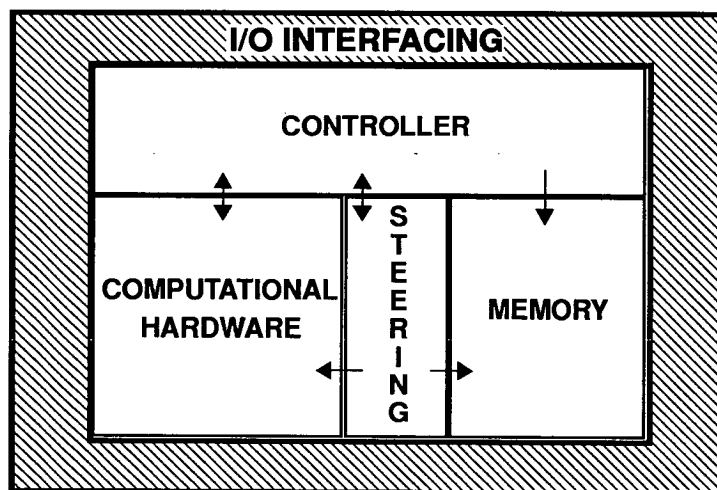
Figure 1.1 A typical data-processor architecture (NEC7720).

Computational hardware performs any *calculations* required on the data, for instance adding and multiplying, and this may be done by hardware dedicated to that

specific calculation - adders or multipliers - or by multi-function Arithmetic Logic Units (ALUs). The memory hardware implements storage for either long-term data, for fast, on-chip access, or for data being transferred between computational hardware elements.

Memories may be Read Only Memories (ROMs), Random Access Memories (RAMs), registers (latches) or register-files, shift registers, stacks, or any other data storage device, and in the case of ROMs, RAMs and register-files, these memories will need an address to be generated before data may be accessed.

The steering logic, including multiplexers and demultiplexers, guides the data between all this circuitry, and is controlled by signals generated by a controller, which may be micro-programmed [84, 85], or have some simpler implementation [193, 194]. This controller may receive feedback from computational hardware, for decision-making, as well as interpreting externally applied information, and generating any clocking signals required by synchronous circuitry. Its other main task is to generate the control necessary for the memory hardware, consisting of Read/Write enable signals, the memory addresses, and any other Shift/Load/Push/Pull control signals. A simplified description of the architecture is shown in Figure 1.2.



*Figure 1.2 Simplified data-processing architecture.*

The huge rise in integrated circuit complexity, through Large Scale Integration (LSI), to Very Large Scale Integration (VLSI), soon overwhelmed the wholly manual approach to digital system design, but *also* provided a solution, in the form of Computer Aided Design<sup>\*</sup>. As more complex and powerful computers were designed, they were used to aid the design of even higher-performance systems (The DEC MicroVax

---

<sup>\*</sup> The IEEE Transactions on Computer Aided Design was created in 1982 to handle the increasing activity in this area.



One area of high-level synthesis which has been under-developed, is that of synthesis of application-specific address generators, as separate entities to the general controller on an IC. The hardware required for address generation can make up a large percentage of the total chip area (up to 50%), so there should be at least as much work done on its synthesis as there is for the computational part of the design. Some contemporary systems treat address generation as just another computation and design the hardware as a data path, but in [141] sequence generation is identified as a distinct basic block in a functional block environment, and it is the field of sequence generator synthesis which is targeted by this thesis.

## 1.2 What is address generation?

To generalise as far as possible, address generation is the production of some sequence of binary words, of some width.

Since the primary use of these words is by memories, as addresses, the generation of them is known as address generation.

An address generator is therefore the hardware which actually produces these data, and may produce several addresses as part of its data word.

## 1.3 The scope of address generation

As stated above, an address generator does of course generate addresses for memories, but can have other uses:

- i) To generate control bit sequences for steering logic (such as multiplexers).
- ii) To generate control bit sequences for selection of function in ALUs, and for other selection requirements.
- iii) To generate test patterns for a processor, on the same IC.

In order to place some limits on the scope of this thesis however, address generation is defined as being the production of a *predetermined* sequence of binary words of some width.

This precludes applications for which a sequence is dependent on internal variables or data, for instance the output from an adder during successive uses.



## **1.4 Overview of address generation techniques**

There follows an account of related techniques in sequence generation, both for control purposes and for memory access.

### **1.4.1 For control**

To date there are very few ASIC design systems which deal with the memory addressing problem as separate entity from the more general and possibly non-deterministic control problem. So control generation has dominated address generation techniques with most address generators eventually embedded in the controller.

Conventional approaches to control generation are, like most tried and tested means, too general in their methods to produce the most optimal architectures. They will produce passable results all the time but are not ideally suited to ASIC design.

The problem is usually one of mapping state numbers to actual control signals as fast as possible, using the least area of silicon. If there are just a few simple mappings then full combinatorial logic would be used, but as the number of inputs and outputs increases, a ROM look-up table may be used in conjunction with a (state) counter to produce any deterministic sequences, or as part of a FSM. However, as the number of inputs increases further, the ROM becomes outsized and a PLA is more likely. ROMs are also used as look-up tables as the basis for micro-programmed logic. A PLA may be used in conjunction with a binary counter to implement any combinatorial logic functions which map the linear state count to the control outputs, and this is certainly the most common form of controller architecture both for deterministic and non-deterministic sequences. The area of a PLA may be reduced by folding parts of the logic array, and their ease of programmability makes them very popular, but testing them as they stand is very difficult. By adding special structures to allow selection of each crosspoint on the PLA however, some PLAs may be used to test themselves [140].

Various other sequence generators and detectors are discussed in [152].

### **1.4.2 For memory access**

Several different address generation schemes have been reported, especially for array processing, based on adders [144], and on ALUs [145] which can use three types of address arithmetic to produce eleven different addressing modes in the Motorola DSP56000 chip. Address Calculation Units are also used in the Tektronix M275 programmable array processor [146] along with pointer registers. Counters have been

used in many cases, especially where the address sequence is to be used for array access, in a regular pattern [205].

## **1.5 Overview of address generator synthesis techniques**

Now we examine the diverse techniques for automatically synthesising sequence generators, for general control, and as a separate entity for memory access.

### **1.5.1 For control**

Devadas et al.'s MUSTANG system attempts to synthesise FSMs [64] using state assignment techniques to optimise a multi-level logic (multiple PLA) implementation [62], while Amman et al.'s SUCIM tool [66] aims at both optimal state assignment and state sequences. Here too, multiple PLA/ROM-structured FSMs are targeted for use with binary counters [65].

Micro-programmed controller synthesis has been around a relatively long time, with Grass and Lipp's LOGE system being a fair example [84, 85], and the Cathedral II system [81] synthesises micro-programmed control also.

The principals of combinatorial logic synthesis are described in [63] and the optimisation or minimisation of such logic is a popular subject [67, 68].

The Yorktown Silicon Compiler [73, 74] is one of the earlier systems for controller synthesis, and others include CPC [72] for use in the SYCO compiler, the SILC compiler [75], and work reported in [76, 77, 79, 80, 82].

### **1.5.2 For memory access**

A schematic capture method is reported in [142] which can be extended to the capture of address generation hardware, and in [143] address sequences for scheduled memory are produced automatically for multi-port memory allocations, but the problem of virtual to real address conversion is ignored, and no real synthesis is attempted.

In the Cathedral II system [162], background memory is synthesised in the form of Pointer Addressed Memories (PAMs), which only require an incremental address sequence, thus avoiding the construction of complex Address Calculation Units. For high speed circuits, the PHIDEO system [61] allows exploration of the address generation costs/ memory costs domain, and can synthesise address generators based on relative addressing using pointers, for a minimal sized memory structure, or based

on counters for a possibly redundant memory, or based on otherwise minimised memory structures, which can lead to rather complex generators.

The author's own address generator synthesis systems, AG1 [149] and AG2 [148, 151], use decomposition of address sequences in trying to find a better hardware implementation than would normally be found using logic synthesis.

## **1.6 Other related approaches**

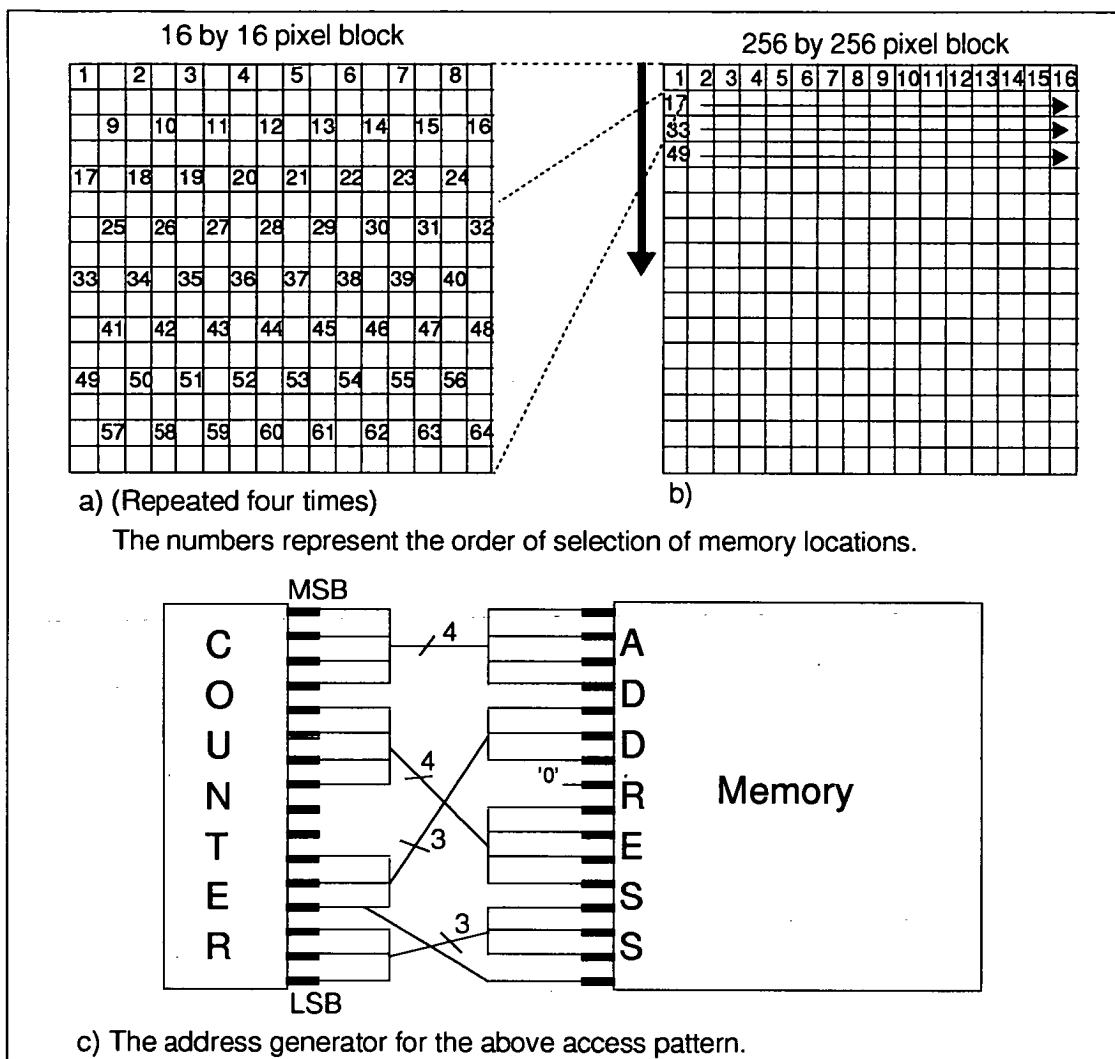
There are other applications for sequence generators. BIST (Built-In Self Test) for PLA's [138], and more general use [135, 139], as well as other design techniques [136, 137] can utilise the same sort of architecture as address generators.

## **1.7 The case for address generators based on counters**

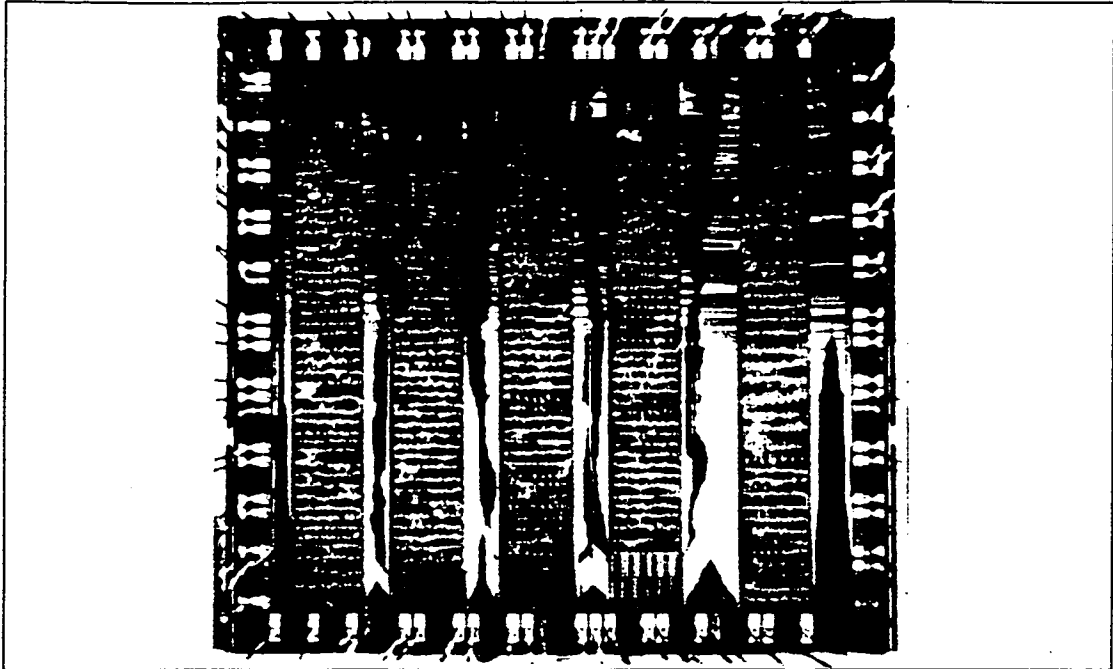
Counters are reliable, reuseable, testable and efficient, which makes such adaptable circuitry invaluable in chip design.

The use of counters in address generators is not a new concept [65], and in fact the subject matter for this thesis was originally inspired by some address generation hardware, manually designed for an image-processing application [205]. One such address generator is illustrated in Figure 1.4, along with a graphical description of the memory access sequence it produces. The gate array chip layout which includes this architecture can be seen in Figure 1.5.

Once this, and other solutions had also been produced automatically, it became apparent that counters could form a part of much more general-purpose address generators, if only the techniques for recognising their possible use could be developed.



*Figure 1.4 Example of a counter in an address generator.*



*Figure 1.5 Chip layout of which about 50% of active area is address generation circuitry.*

## **1.8 Comments**

The DTI and the SERC have recently specified a new jointly funded collaborative programme - VLSI Design Automation for Information Technology Systems - whose workplan includes a sub-section on memory architectures which calls for "Novel configurations of memory and address generation, leading to reduced hardware requirements, involving automatic generation from system requirements" [206].

The design of address generators is at present an expert's task. The intuitive decisions made in their conception are based on a collection of experience of this and similar problems, from many different angles. It is only by utilising this experience that the human designer may hope to overcome the size and complexity of some of the design problems, to realise a working solution. But then a careful and often lengthy check must be made, perhaps using simulation, to ensure that no errors have occurred during the design. The size of the problem, however, will often make exhaustive simulation prohibitively expensive, so often the expert must be recalled to intuitively check the solutions by hand.

We believe that an address generator synthesis tool should not only guarantee correct solutions, which will also be testable, but should also do so with a vast improvement in design time. We will report later on such a tool which has mirrored manual designs with an increase of between one and three *orders of magnitude* in performance.

## **2 Introduction to address generator architecture**

In Chapter 1, an address generator was defined as producing a predetermined sequence of binary words, of some width. To complicate proceedings a little we may add constraints of timing on both the address sequence as a whole, and on the speed of production of the data words. Area considerations may force re-use of all or some part of the address generator, so some outside control may be necessary, as well as the associated logic to implement that control. Power consumption may also figure in deciding on a specific address generator architecture.

Commencing with a description of the types of memories for which addresses may be generated, this chapter defines the constituent parts of successively more complex and generic address generators, starting with some different implementations of the most basic element - the binary counter. This is then generalised to a modulus  $m$  counter, before other pieces of the architectural jigsaw are examined. Finally, a breakdown of the costs of these address generator elements is presented.

### **2.1 The memories**

Before defining possible address generator architecture, we first examine the memories which can require addressing.

On the physical level, a RAM cell (bit) may be reduced to just five transistors for a slower, static approach [176], or to a single transistor in the faster, dynamic mode [178] which has refresh overheads. A ROM cell may also be reduced to a single transistor due to its simplicity [179]. Other novel approaches include magnetic bubble memories [180].

Memories may be Content-Addressable [172, 173, 177], or Content-Associative for faster access [171], or have other added features [174], and may be designed using many different styles [175] including Standard Cell, Super Integration and Structured Array to develop RAMs and Cache memory.

A highly parallel memory structure may be defined for parallel processor applications [170], and multi-phase clocking schemes may introduce other possibilities [181, 182..184]. A RAM may have one or possibly several pairs of address and data ports.

## 2.2 The binary counter

Being such a well-known piece of hardware, the binary counter can take one of several proven forms, and more specialized architectures are possible with differing IC technologies [195, 196, 198]. The basis for all counters in this thesis is a series of JK master-slave flip-flops, whose values toggle at certain times.

### 2.2.1 The ripple counter

Simplest to construct is a ripple counter, which has all of its flip-flops, or bits, set to toggle at any time, by tying both J and K inputs to logic '1'. The first flip-flop, Bit 0, is clocked by an external clock, and its  $Q(t)$  output is then used to clock the second flip-flop, Bit 1. The  $Q(t)$  output from this is used to clock Bit 2, and so on until a counter of the correct size (number of bits) is produced. The  $Q(t)$  outputs form the count word at time  $t$  - an output from the counter module - which may need to be strobed by a control signal to synchronise the signals.

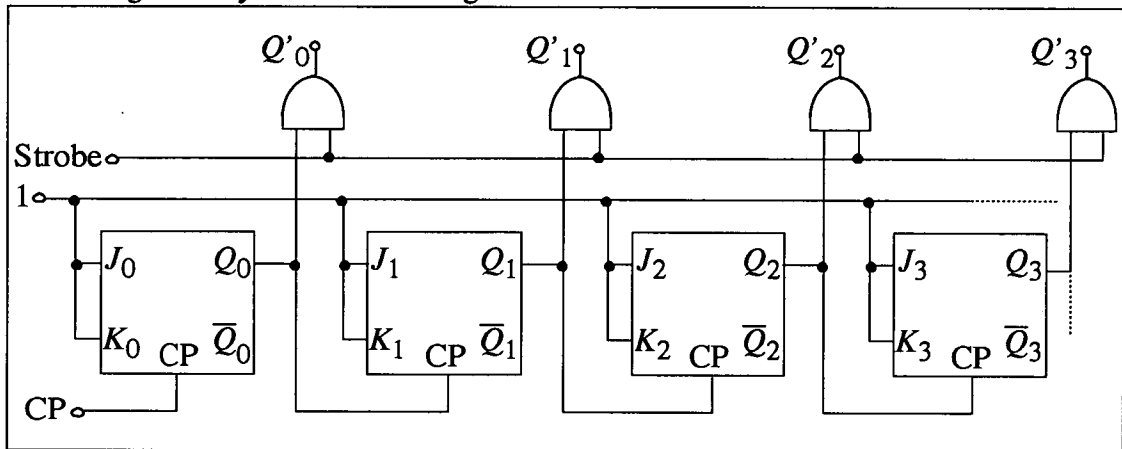
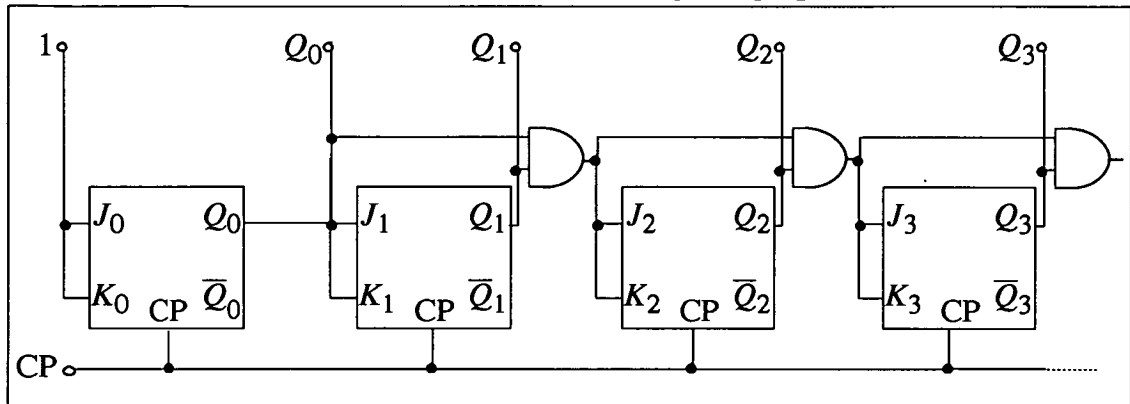


Figure 2.1 The ripple counter with strobe.

### 2.2.2 The serial carry counter

To avoid the problem of synchronisation, and so the need for a strobe, the serial carry counter is constructed a little differently to the ripple counter. The first bit of the counter is set to toggle at any time as before, but instead of using its output to clock the next bit, this is done by the external clock itself. The second bit will only toggle if the  $Q(t-1)$  output of the first bit is in the high state. All other bits of this counter are also clocked by the same clock, and the value on their J and K inputs is set by the result of

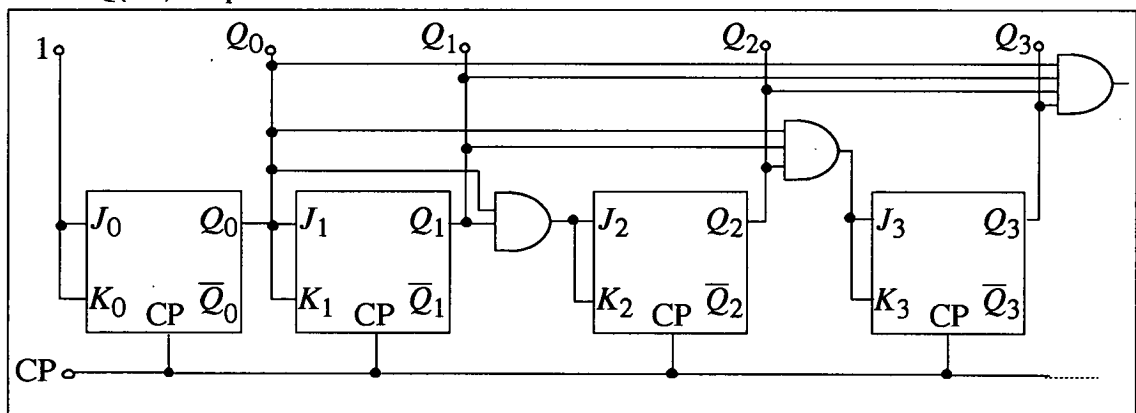
ANDing the next lower bit's  $Q(t-1)$  output with its J (and K) input. In this way, the delay between the first and last bits of the counter settling is kept quite low.



*Figure 2.2 The serial carry counter.*

### 2.2.3 The parallel synchronous counter

To increase the speed of the counter further, one must take a more parallel approach in its construction. Here, the first three bits work in the same way as for a serial carry counter, but any further bits will have their J and K inputs set by a logical AND of *all* lower Q(t-1) outputs.



*Figure 2.3 The parallel synchronous counter.*

#### 2.2.4 The serial/parallel synchronous counter

Since the parallel counter described above requires an  $(n-1)$  input AND gate to implement an  $n$ -bit counter, then as the number of bits increases so does the likelihood of AND gate fan-in rules being broken. This problem may be partially solved, at the expense of some speed, by placing an upper limit on the number of counter bits



connected in this way. If this limit is reached during the construction of the counter, then a serial carry is taken to the next bit, and parallel carrying is commenced from there.

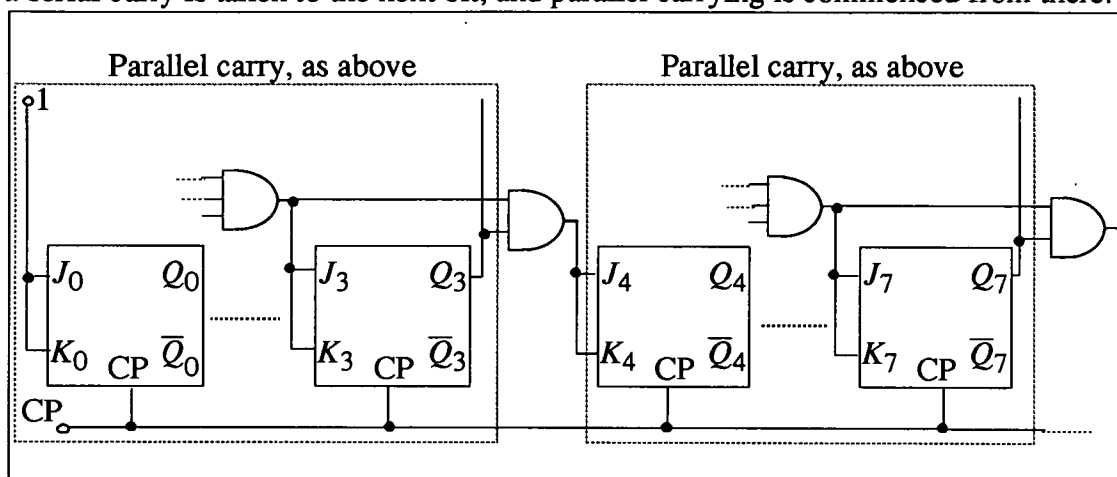


Figure 2.4 The serial/parallel synchronous counter.

## 2.2.5 The pseudo-parallel synchronous counter

Another possible conglomeration of basic counter architectures is shown in Figure 2.5, where a large (Say > 4 bits) ripple counter is divided into equal slices, and each of these slices is clocked by a parallel carry from the previous slice, but remains, internally, a ripple counter.

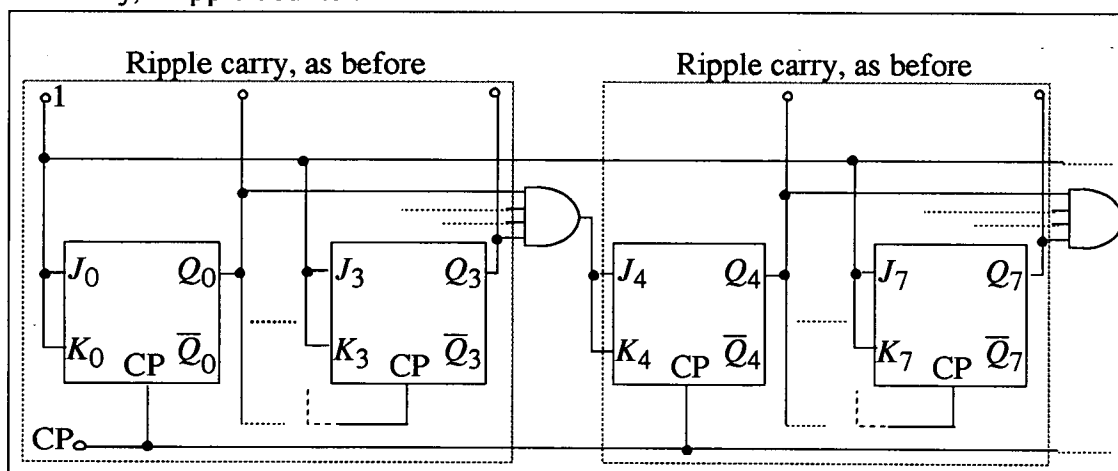


Figure 2.5 The pseudo-parallel synchronous counter.

## 2.3 The modulus m counter

A useful generalisation of the binary counter is the so-called modulus  $m$  (or modulo  $m$  [197]) counter. Instead of each counter bit producing streams of bit values which change value after some power-of-two of bits, *the modulus  $m$  counter produces*

*streams of bit values which change after  $m \cdot 2^n$  bits, where  $n$  is the bit number on the counter ( $n \geq 0$ ). Thus a binary counter is in fact a modulus 1 counter.*

A modulus  $m$  counter is basically two binary counters connected in series, with the first counter having the logic necessary to reset it after  $m$  clock ticks (or rather, when its count value would otherwise have reached  $m$ ).

The bits of this first counter make up the *lesser* bits of the modulus  $m$  counter, as illustrated in Figure 2.6, which shows the architecture of a modulus 7 counter. The reset logic shown on the outputs of the lesser bits will henceforth be omitted for simplicity, and because the cost of this logic is minimal.

The second counter, clocked by the MSB of the first counter (or a combination of bits, if a more parallel carry counter is used), implements the *upper* bits of the modulus  $m$  counter, whose bit sequences follow the  $m \cdot 2^n$  repetition rule as defined above.

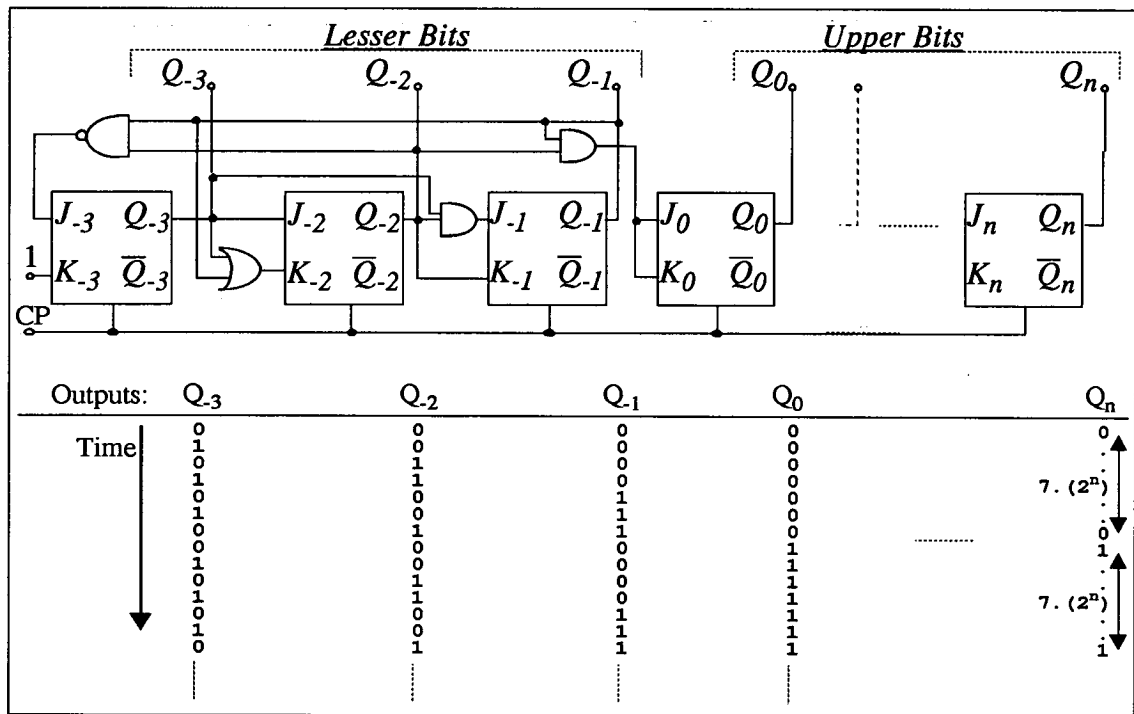


Figure 2.6 A modulus 7 counter.

By using modulus  $m$  counters prudently, it is possible to realise very cheap and simple address generation schemes which implement complicated, non-binary access patterns, for array access. It is also possible to produce some very efficient schemes for scheduled memory address generation and even general control, usually the final element of a processor to be examined, can benefit from the use of modulus  $m$  counters.

There are two, very basic address generation architectures which use a modulus  $m$  counter (Figure 2.7). The simplest is the preset-modulus counter, which has reset logic built directly onto the counter, as for the modulus 7 example above. The second address generator contains a parametrizable-modulus counter, which has an input dedicated to receiving the modulus parameter. It also contains a comparator, to generate the reset signal when needed.

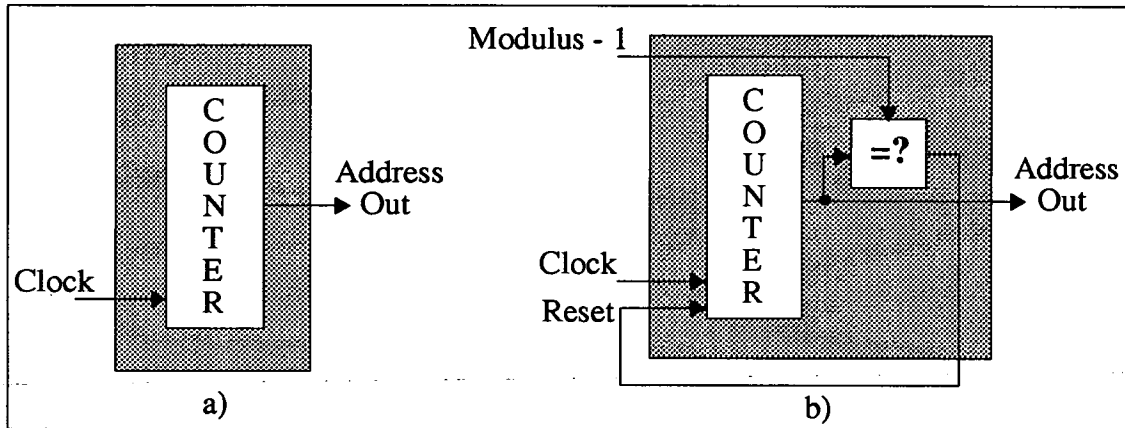


Figure 2.7 Two basic address generators:  
a) Simplest counter; b) Parametrizable-modulus counter.

## 2.4 Other address generator elements

Other address generator elements include: ROMs, dedicated to storing a sequence of addresses; incrementors (simplified adders); exclusive OR gates, and random logic.

### 2.4.1 Address ROMs

Address ROMs, and their own address generation schemes, have three distinct architectures, as illustrated in Figure 2.8:

- 1) A minimum-sized ROM, which contains only one copy of any address required.
- 2) A medium-sized ROM, which contains multiple copies of addresses needed more often.
- 3) A maximum-sized ROM, which contains an address, required or not, for every control step of a process.

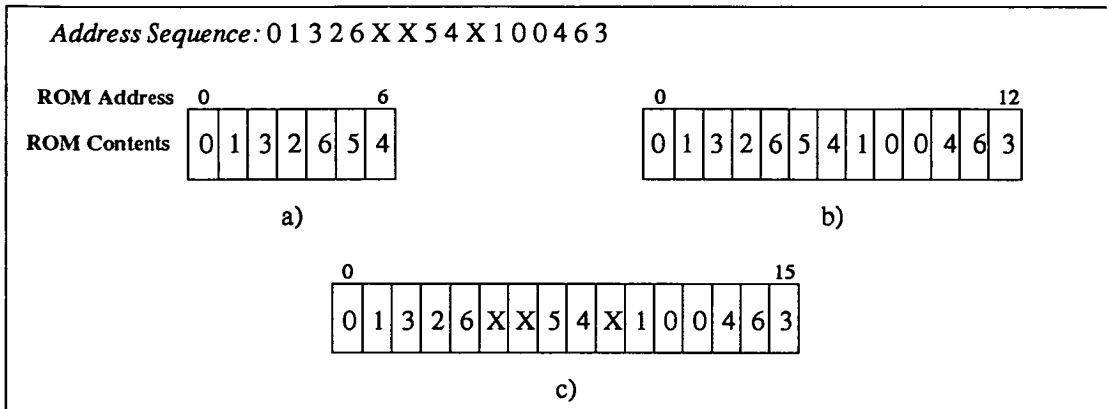


Figure 2.8 The three address ROM architectures:  
a) Minimum-sized; b) Medium-sized; c) Maximum-sized.

Often the choice of address ROM will be forced, by factors inherent in the overall address generation requirement. For example, in array access, one element of that array can be accessed in each control step, implying a maximum-sized ROM. But in the case of scheduled memory access, any form of address ROM is possible.

A maximum-sized ROM is created for scheduled memory, by filling any “Don’t care” times in the original access sequence, with real addresses. This may seem strange, but it allows us the simplest of *ROM-address* generation schemes - A simple, binary counter. The very action of filling the “Don’t care” times is a complicated one, and this should be done on a bitwise, rather than wordwise basis. This may even allow a cheaper address generation scheme than a ROM-based one. This problem is dealt with in Section 6.4.3.3.

A medium-sized address ROM can be created simply by storing in it only those addresses actually required, in that order, and allowing multiple copies of addresses. This slightly complicates matters for the *ROM-address* generator, in that the counter’s clock must be gated by a control signal, which must be generated elsewhere, and which allows the counter to be clocked only once the current address in the ROM has been read out. With some care, it is possible that the gating control sequence for the ROM-address counter could be given regularity by utilising any “Don’t care” times available, and perhaps clocking the counter a little earlier than necessary.

A minimum-sized ROM is created if only *one* copy of any address required is stored within it. This can save a lot of ROM area, but *unless* the addresses are only ever needed once, or they are required often, but in a regular order, the address generation for the

ROM access can result in yet another ROM-based address generator! This is obviously pointless, if we consider the example below.

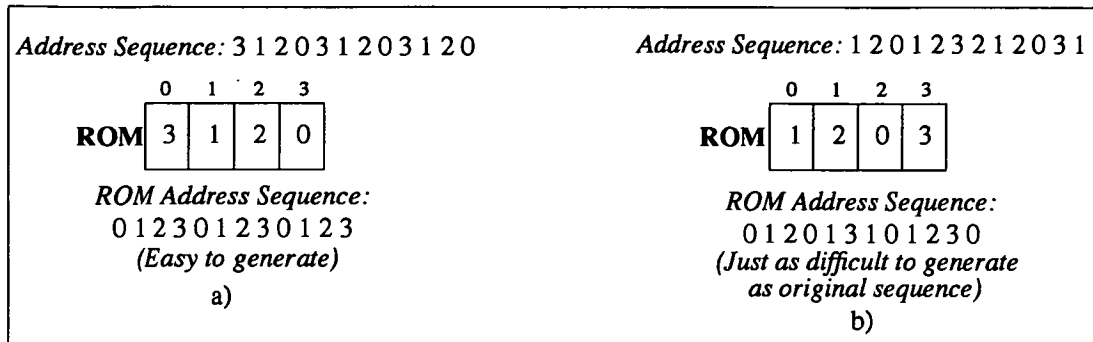


Figure 2.9 Examples of a): good and b): bad uses of Minimum-sized ROMs.

### 2.4.2 Exclusive OR gates

Exclusive OR (EXOR) gates are often essential to a good address generation solution, especially when used in conjunction with counters. An exclusive OR gate can be used to invert the polarity of some bit sequence, after some number of bits, as shown in Figure 2.10. The original bit sequence is input at A, and the second sequence, which controls the inversion of polarity, is applied at B. When the value at B is logic '0', the polarity of C follows that at A, and otherwise is the inversion of A. The bit sequence produced at C is then exclusive ORed with yet another bit sequence, and the same rules apply to that combination.

Usually the sequences fed in at B and D will be generated by a counter, causing a regular inversion of polarity, and often the sequence at A is produced by one or more bits from the same counter. In Figure 2.10, all bit sequences applied to the EXOR network are generated by a 4-bit, modulus 3 counter.

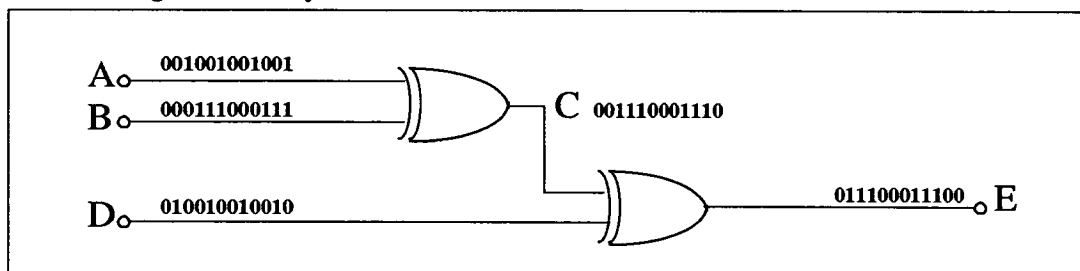


Figure 2.10 The effect of an EXOR network.

The sequence produced at E bears only a little resemblance to those used to generate it, and it is here that many problems arise, in recognising possible bit sequence generators.

### 2.4.3 Clocked-bits

It is often the case that one bit sequence within an address sequence may be generated by using another of its constituent bit sequences to clock a T-type flip-flop. This is illustrated in Figure 2.11. A ripple counter is formed in this way.

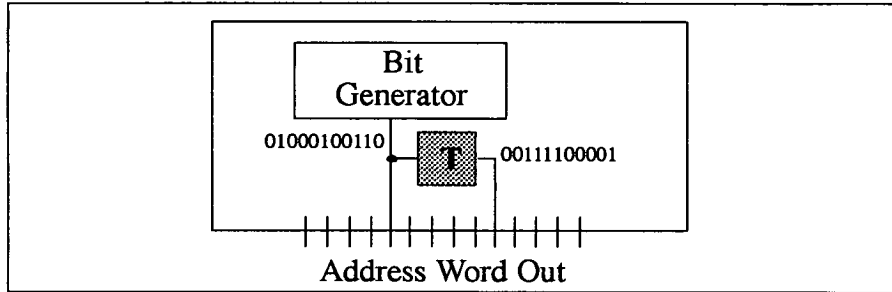


Figure 2.11 Example of Clocked-type bit sequence generator.

### 2.4.4 Incrementors

Another commonly occurring element of address generators is the adder. This is perhaps most useful when the 'sum' output is fed back to one of the adder inputs, and the other input is hardwired or set externally to a constant binary value, thus creating an Incrementor. The adder can of course take one of several hardware implementations, including lookahead-carry in bit-parallel adders [185..189], and various bit-serial approaches [198]. We will concentrate on the bit-parallel adders in this thesis, since the feedback is much simpler.

A simple incrementor is shown in Figure 2.12, where the increment is 3, and since this contains no reset circuitry, the address sequence produced does not repeat until after the third cycle, as shown.

Figure 2.13 shows a more commonly used incrementor, which has circuitry to cause a reset of the incrementor at a value which will produce a shorter cyclic address sequence. Finally Figure 2.14 illustrates a general purpose incrementor, whose increment may be set externally, along with the value at/above which to reset, and also an optional preset value. Also included is an optional gating circuit, and corresponding gating signal, which enable the incrementor to remain at the same value for several control steps.

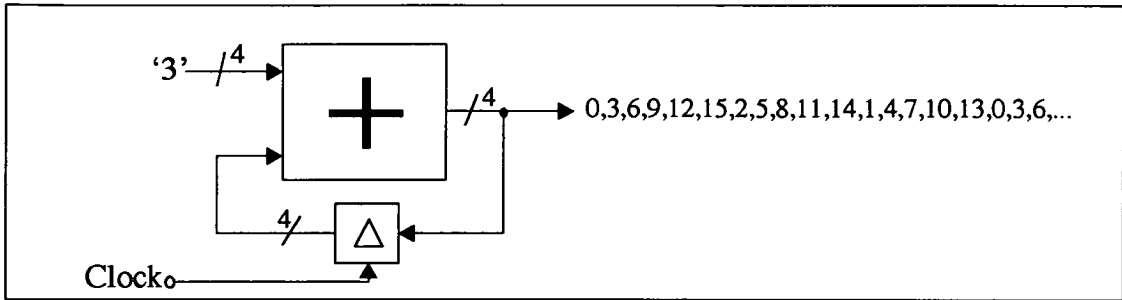


Figure 2.12 A simple incrementor.

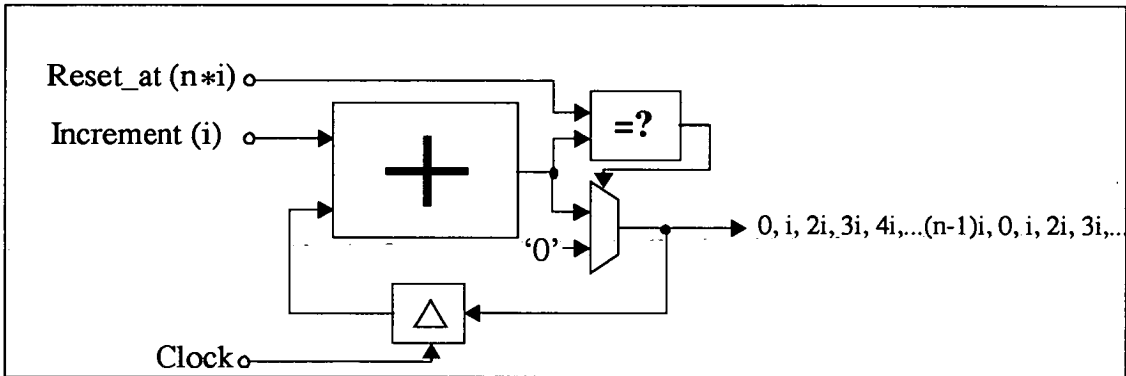


Figure 2.13 Incrementor with reset.

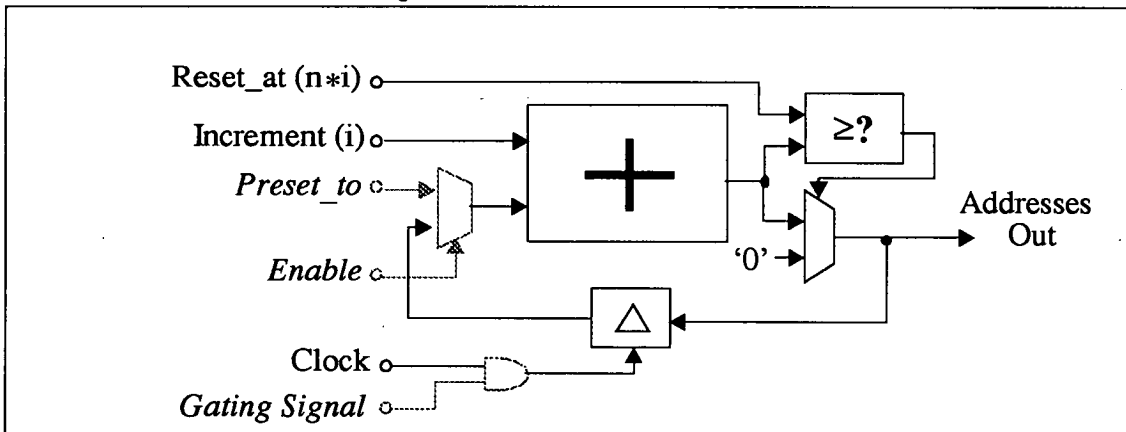


Figure 2.14 A general purpose incrementor.

## 2.4.5 Logic

The vague classification “logic”, represents here either a Boolean network of any size, which is fed by some counter bits to produce address bit sequences, or that combinatorial logic required for any very local control (eg: Reset logic for counters).

## 2.5 Cost breakdown of address generator elements

The area and speed of the hardware described are both technology-dependent, making costing a rather localised matter. For our purposes, ES2's Solo-1400 1.2 micron parts library [102] was suitable as the source of costs, where a "stage" is two transistors, or half a gate.

### 2.5.1 Area-costs

Table 2.1 below shows the approximate areas of each address generator element in terms of the number of "stages" of logic which they will require.

Component	Area cost in stages	
	Actual	Relative
Counters: Ripple (per bit)	27.4	1.00
Serial	47.5	1.73
Parallel	49	1.78
Serial/Parallel	48	1.75
Pseudo-Parallel	48	1.75
ROM (per bit, not incl. overhead)	1	NA
EXOR Gate	7	NA
JK Flip-Flop	30	1.10
Incrementors: Simple	49	1.78
General Purpose	58	2.11
Logic (per two-input gate)	2	NA

*Table 2.1 Comparative area of address generator components.*

The area of a ROM-based address generator can only be defined once the size of the ROM has been decided upon, since the ROM will have a certain overhead-cost for creation (address decode, etc.) which will be shared between all bit sequences contained in the ROM, as will the cost of the address generator for the ROM itself.

The ripple counter, although quite small, is unlikely to be used because of the problems its asynchronous output signals introduce, and because of the slow speed when used with strobing circuitry.



### 2.5.2 Speed-costs

The approximate delay of each component, in terms of the delay through a single JK flip-flop, are given in Table 2.2.

Component	Delay	
	Actual	Relative
Counters: Ripple (n bits)	n	10n
	Serial	20
	Parallel	10
	Serial/Parallel	2.5n
	Pseudo-Parallel(4)	40
ROM	3	30
EXOR Gate	0.1	1
JK Flip-Flop	1	10
Incrementors: Simple	1 (fast lookahead carry)	10
	General Purpose	30
Logic	$<1 \rightarrow \infty$	$<10 \rightarrow \infty$

Table 2.2 Comparative speeds of components.

### 2.6 Comments

An address generator is composed of a set of bit sequence generators, which may take any of the forms mentioned in this chapter. The next chapter will describe the different possible requirements for an address generator, for different memory architectures and for control purposes.

### 3 Requirements for an address generator

This chapter investigates the possible causes of requirements for an address generator.

#### 3.1 Data-dependent addressing

Data-dependent addressing requires an address generator to produce an address from variable data, either directly, as in case (a) below, or indirectly, as in case (b):

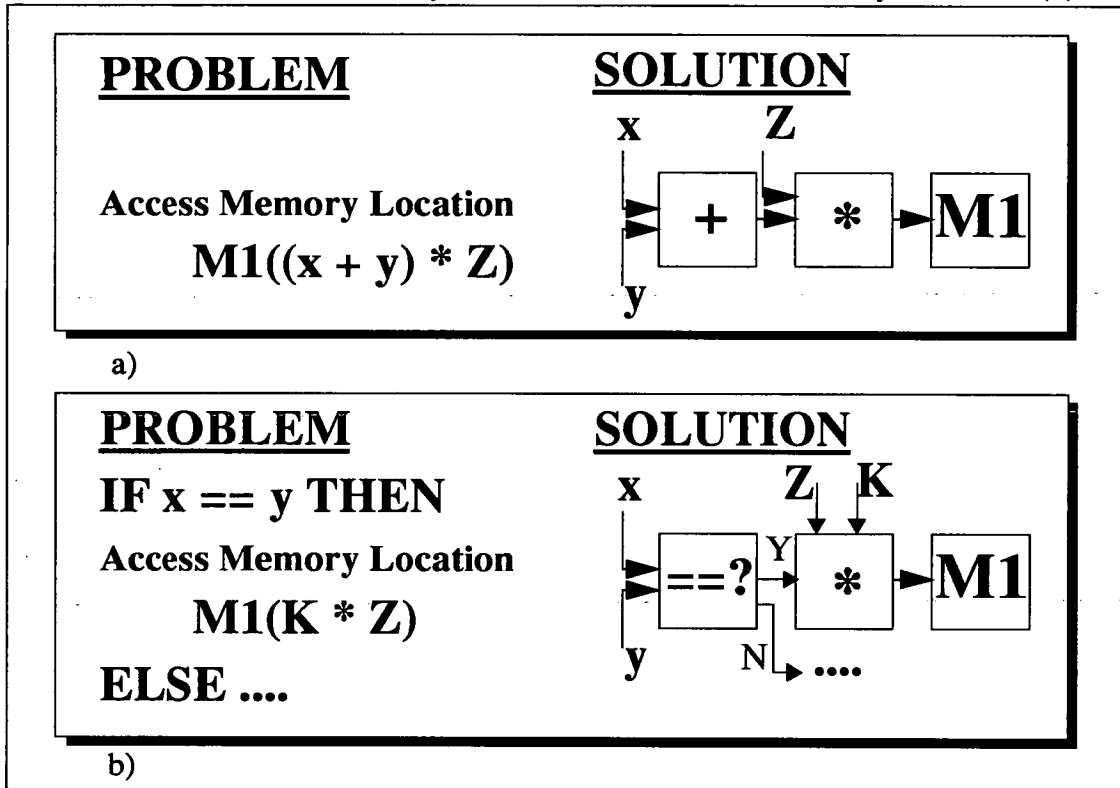


Figure 3.1 Data-dependent addressing schemes.

Neither of the cases above are applicable to the automatic address generator synthesis techniques targeted by this thesis, but in each case there is a definite direction to take for automation. In the case of direct transformation of available data, this can be recognised at a behavioural level, from the corresponding High-level description language, and address generation hardware constructed as part of the computational base [163]. Indeed, it is this approach which is taken in SAGE (See Chapter 8), a behavioural synthesis tool developed at the University of Edinburgh[150].

In the second case of data-dependent addressing - That of indirectly using available data to specify a wholly different address, or sequence of addresses (if a branch runs for

several control steps, or perhaps branches further) - no such scheme is possible. The simplest approach seems to be to use the branch conditions to select from a variety of ROM's which hold the possible address sequences, or to embed the address generator in the chip controller.

### 3.2 Scheduled memory addressing

In a design with many operations and few resources, the data produced by operations may need to be stored over more than one clock tick, perhaps accessed several times during the lifetime of that data. Instead of assigning a single register to store each group of temporally disjoint data, a RAM or register file may be used which can hold much more data but which will require an address sequence to control access. Very often there will be times when no address is actually required, so that the address produced at that time may be any of those possible, and choosing one may have a significant effect on the cost of producing the address sequence. This is a problem which must be tackled before the address generator synthesis stage-proper, as part of the memory synthesis task (See Section 6.4.3.3).

### 3.3 Array access

The final, and perhaps the most promising situation as far as synthesis is concerned, is that of array access. Here, one, two or more dimensional data arrays are written to and read from usually large memories. Commonly the access sequence will be predefined at synthesis time, most usually as a set of (nested) loops. It is a simple matter to deduce the access sequence thus required and to use decomposition methods to recognise the use of counters in its generation.

It is also possible that the synthesis stage could recognise oversized memories or redundant accesses and optimise them accordingly [166]. Figure 3.2 shows an example, where data is to be written to a memory in locations  $a[0]$  to  $a[255]$ , and then read out again in four separate passes.

```
for i = 0 to 255 loop
  get x;
  store x in a[i];
next i;

for j = 0 to 3 loop
  for i = (((j+1) * 64) - 1) downto j*64 loop
    read a[i];
  next i;
next j;

I.E.: i = 63,62,...0,127,...64,195,...128,255,...196.
```

Figure 3.2 Example of array access specification.

This obviously implies a 256-word RAM, with a simple counter to generate the write address sequence, and something a little more complex for the read address sequence, detailed in Figure 3.3a&b. Since the two most significant bits in each addressing scheme are produced by the corresponding bits from the (shared) counter, it is possible to reduce the memory size by half, and if the RAM may have two ports then the read and write sequences may be interleaved, as illustrated in Figure 3.3c.

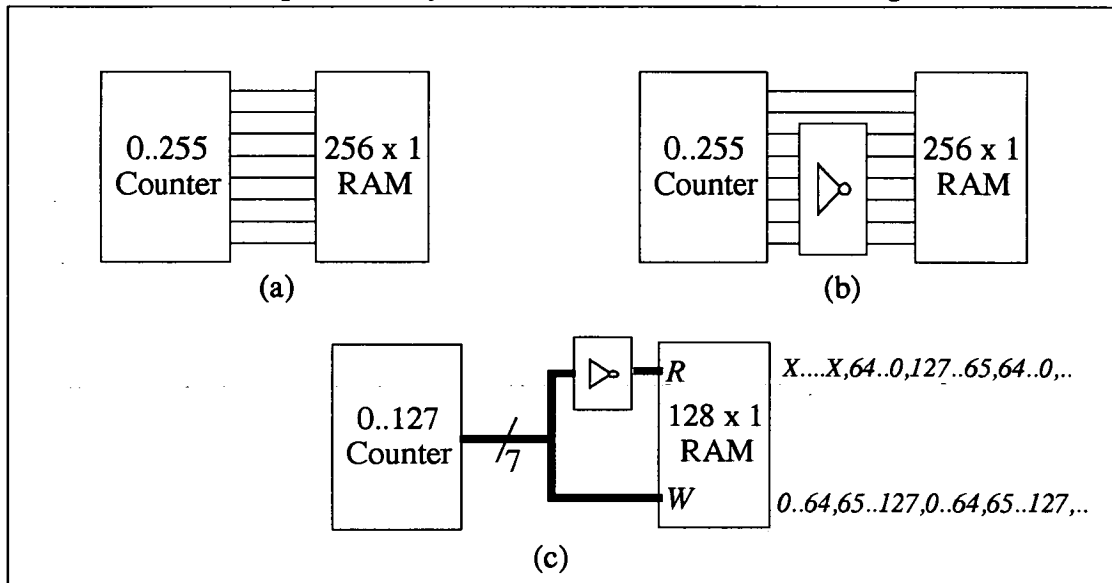


Figure 3.3 Synthesis and optimisation of array access example.

This generalises to the rule: “If the top  $n$  bits of the address word may be generated by the same counter bits for two access sequences, then as long as circumstances allow, the top  $(n-1)$  bits may be discarded, the memory shrunk by a factor of  $2^{(n-1)}$ , and the new most significant address bit inverted for one sequence. This is pictured in Figure 3.4.

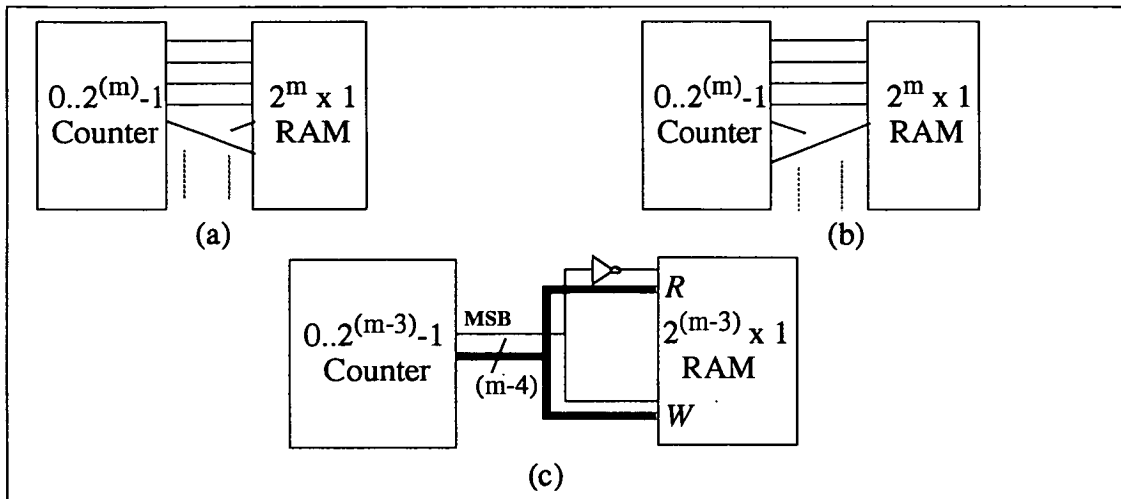


Figure 3.4 Generalisation of memory optimising transformation.

### 3.4 Control

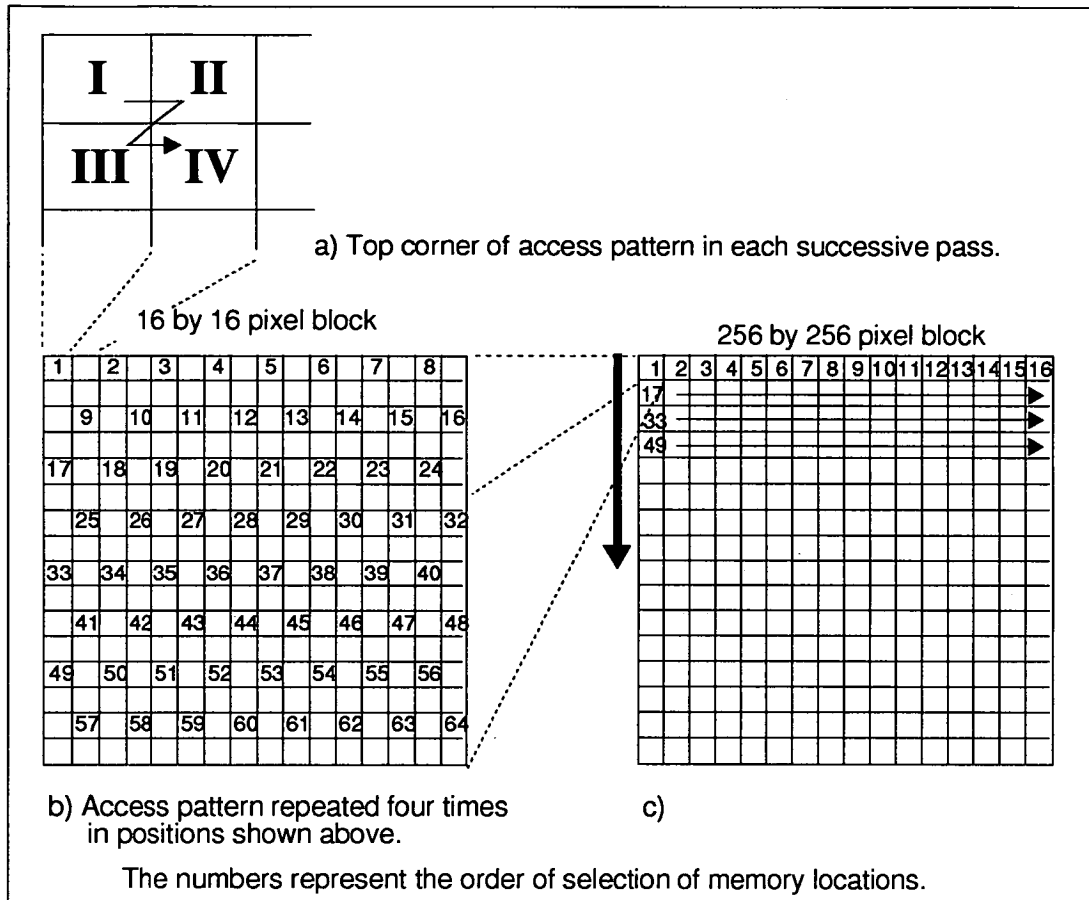
Wherever a control bit sequence can be predefined, we can use the address generator synthesis techniques to try to find a cheap hardware implementation to produce it. As for scheduled memory address sequences, the binding of Don't Care values to actual values can be critical if such a solution is to be found. Requirements targeted here are multiplexer control, write enable signals for memories and ALU function-selection, amongst others.

### 3.5 Comments

It becomes obvious that almost any deterministic sequence of binary words has the potential to be produced more cheaply than by the usual combinatorial logic methods. Not mentioned here are sequences to be produced by built-in self-test (BIST) circuitry, whose manual solutions bear a striking resemblance to those produced automatically for other sequence requirements. The next chapter will detail some of those solutions obtained using AG1 - a first attempt at an address generator synthesis tool.







*Figure 4.3 Further alternative thresholding access pattern.*

Following the determination of a local threshold for each block of data, the image will pass through a binarisation stage, producing the binary - black and white - image, and this too can have a non-linear address sequence. One such sequence is described below, and is much simpler than the thresholding patterns given before.



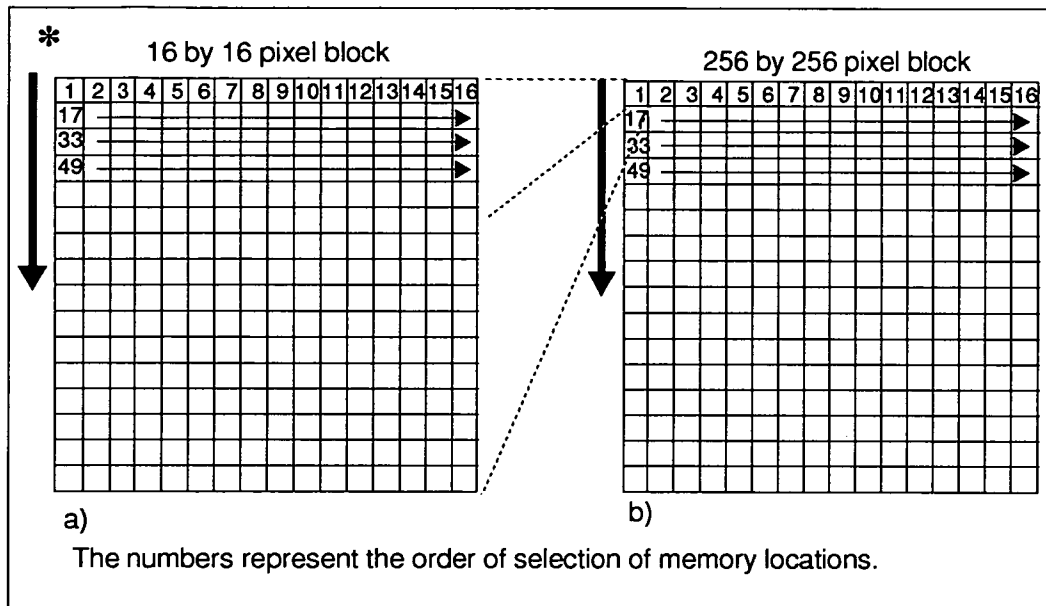


Figure 4.4 Binarisation stage access pattern.

If the image processing application is one of image recognition, then the binary image will probably be passed through a correlator, which gives a measure of how well segments of the image match the same segments of a previously stored image. This could have the access pattern shown in Figure 4.5, which samples the data in four pixel by four pixel blocks.

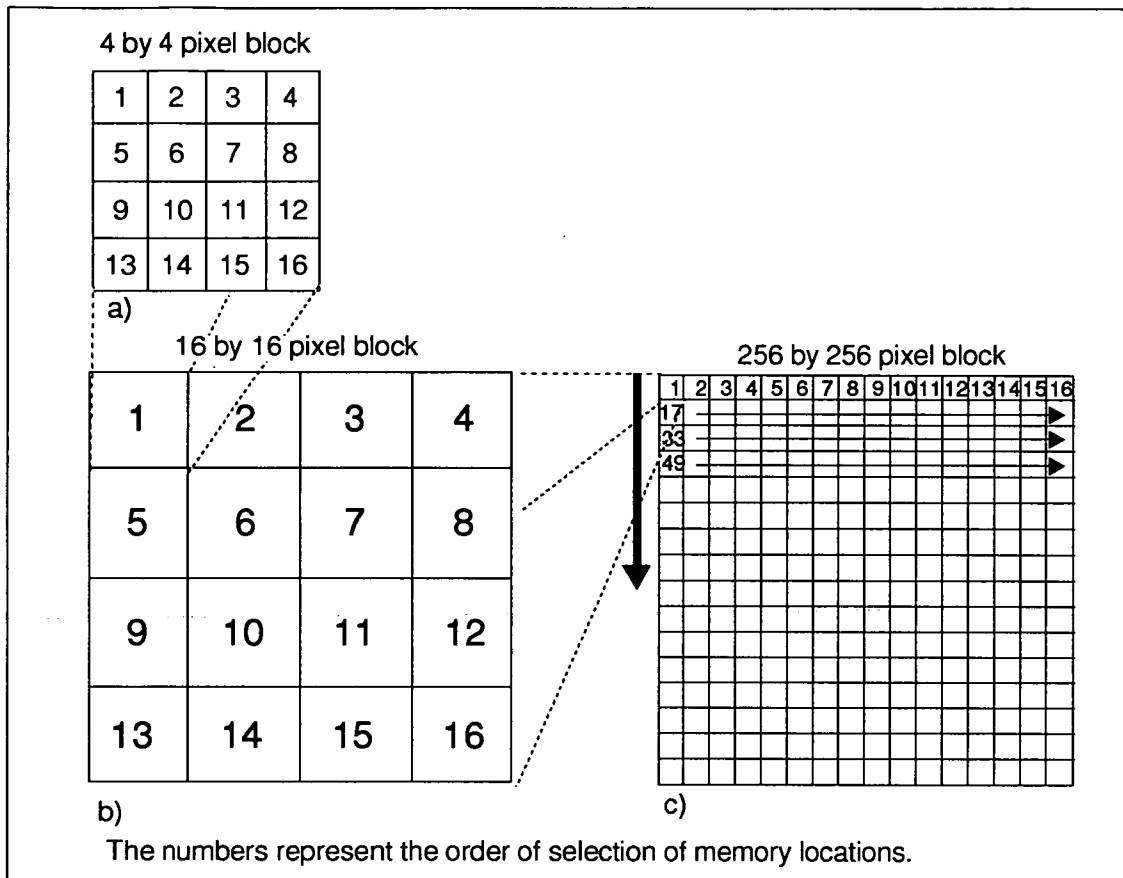


Figure 4.5 Possible correlation filter access pattern.

## 4.2 Some manually designed address generators

Shown below are the address generation solutions found manually for some of the examples given in Section 4.1. It should be said that these designs originally took a matter of man-days to produce, and considerably longer to verify by simulation (Indeed, exhaustive simulation of all 64k steps was never attempted). The address generator for the original thresholding filter consists of a 16-bit binary counter, whose output bits are “shuffled” by wiring only, before being connected to the 16-bit address port of the memory.

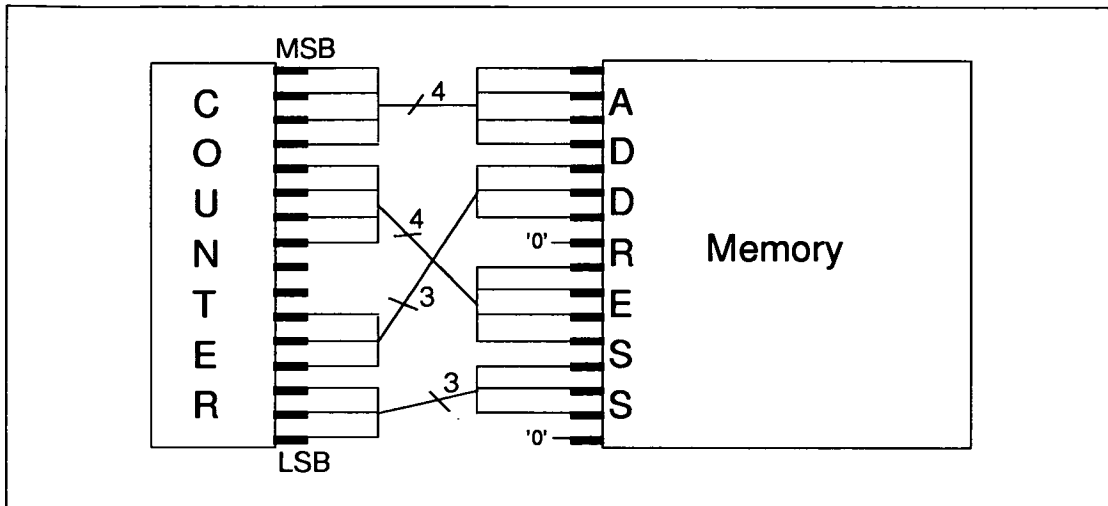


Figure 4.6 Original thresholding filter address generator.

The other two thresholding filter patterns produced the following address generators respectively.

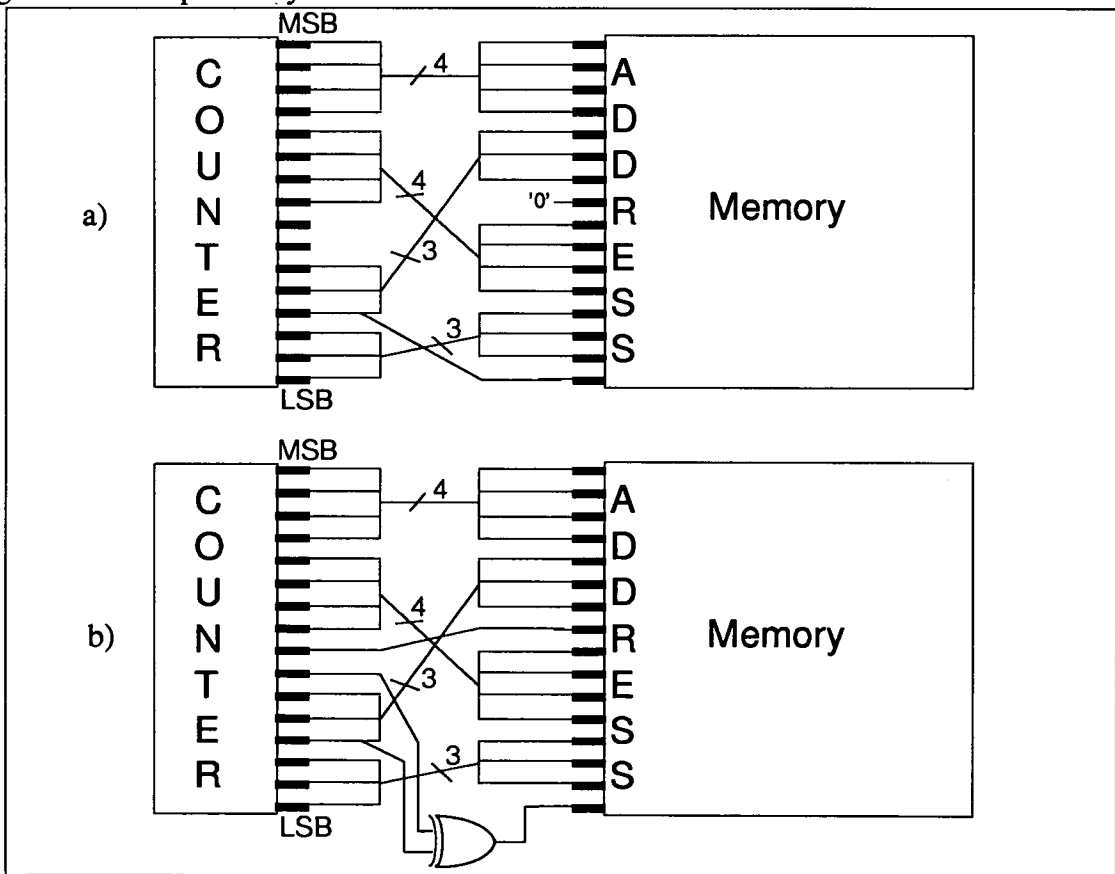


Figure 4.7 Other thresholding filter address generators.

The second real example, of the binarisation process, produced a much simpler shuffling of the counter bits, as shown in Figure 4.8.

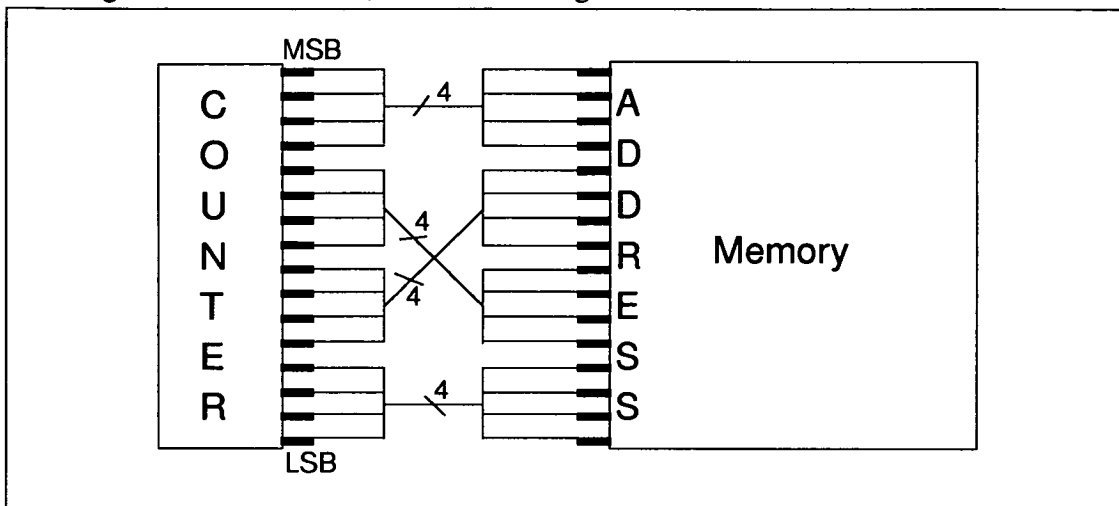


Figure 4.8 Binarisation process address generator.

From the third contrived example of a possible correlation filter access sequence, was obtained a more complex transformation of the address bus, and this demonstrates some of the rules which *could* be used to automate this design process.

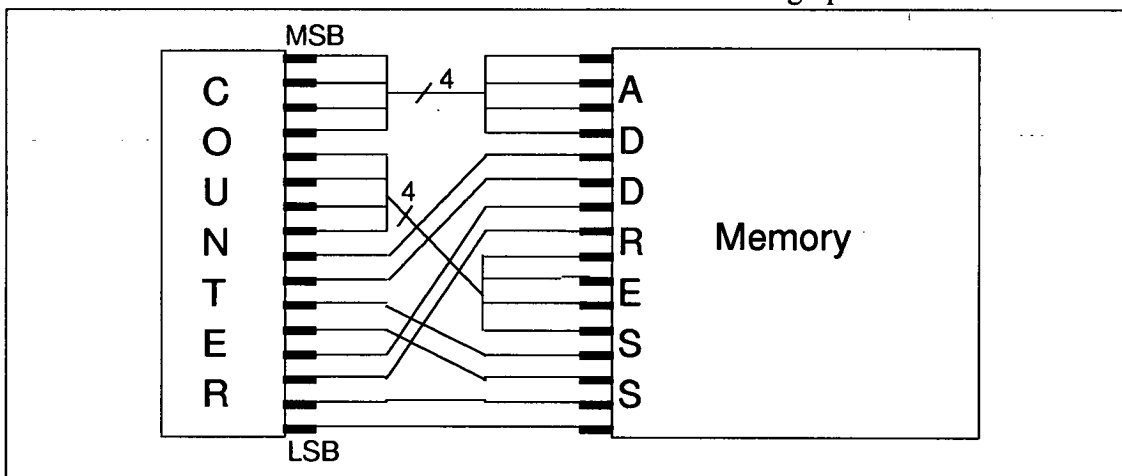


Figure 4.9 Address generator for possible correlation filter.

### 4.3 AG1 - Address generator synthesis based on binary counters

There follows a description of a tool developed to automatically design (describe) address generators of the type described above, based on the specification of the address sequence to be generated. A brief outline of modes of data entry is followed by the method behind the synthesis algorithms. A short description of how combinatorial logic

synthesis proceeds is then given, and finally the output format from the tool is explained.

For a more detailed description of the tool's functionality, and the source code itself, refer to Appendix D, and the diskette enclosed.

#### 4.3.1 Data Entry types

Four different methods exist to introduce the access sequence into AG1. The first allows generation of the sequence by software, possibly as a set of nested loops, as shown in Figure 4.10. The second data-entry method uses a built-in graphical entry tool to lay out the access pattern on a two-dimensional representation of the memory space. Like the first method, this is very amenable to the non-expert designer.

```
for Y = 0 to 65535 step 4096,
  for X = 0 to 255 step 16,
    for i = 1 to 4,
      for y = 0 to 4095 step 512,
        for x = ( y / 512 ) mod 2 to 15 step 2,
          address(time t) = x + y + X + Y,
        next x,
      next y,
    next i,
  next X,
next Y
```

(block height = 16 rows)  
(block width = 16 columns)  
(do 4 times)  
(every 2nd line)  
(every 2nd pixel, skewed)

Figure 4.10 Sequence specification by software.

The final two data-entry options deal with loading predefined bit sequences from file: Primarily sequences which follow no specific pattern, or which are not of length  $2^n$  bits. These sequences should, however, have 'padding' bits added to make the total length a power of two, since *the main pre-requisite of all input sequences is that they have length  $2^n$ .*

#### 4.3.2 Method

The basis for the synthesis method involves iteratively bisecting each bit sequence in the address sequence from the LSB to the MSB, recognising the presence of various binary counter bits in their generation. The length of the first and subsequent halves of

the sequence, which should be length  $2^n$ , give indication as to the binary counter bit(s) involved, as described by the rules listed below.

1) *Split sequence of bits, list[1..2n], into two halves, list[1..n] and list[n+1..2n].*

For example the list:

[0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1]

becomes:

[0,0,0,0,1,1,1,1] and [0,0,0,0,1,1,1,1]

and the list:

[1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1]

becomes:

[1,0,0,1,0,1,1,0] and [0,1,1,0,1,0,0,1]

(These lists are much longer in practice).

2) *If the list has a single entry ( $n = 1$ ), then force the current address bit to '0' or '1', according to that entry, and then go on to examine the next most significant bit of the addresses.*

This only happens if all the entries in the original list for the bit were identical.

3) *If the two halves of the list are identical then halve the list by discarding the second half, and return to (1).*

This controls the use of Rule 1 by allowing the first half of the list to be split further, only if both original halves are identical. For the first example given above, after the first split the two halves are identical, and so we can take the first half and split that:

[0,0,0,0,1,1,1,1]

becomes:

[0,0,0,0] and [1,1,1,1].

4) *If the two halves are not identical, nor the logical inverse of each other, then we cannot use any counter bits directly connected to this address bit, and we go on to use the logic synthesis tool.*

A list of bits must be symmetrical about the midpoint for things to proceed further.

*5) Rule 5 checks that the list has a length of  $2^m$  (which should always happen) and stores the fact that the  $(m+1)$ th counter bit,  $cbit_m$ , can be used to generate this list.*

Rule 5 is invoked with the knowledge that the two halves of the list are not identical (otherwise it would have been split again) but that they are the logical inverse of each other (otherwise Rule 4 would have been invoked). Several possibilities arise at this point, with the list having many different possible forms:

[0,0,0,0] and [1,1,1,1],

[0,0,1,1,0,0,1,1] and [1,1,0,0,1,1,0,0],

[1,0,0,1] and [0,1,1,0],

[0,1,0,0,1,1,0,0] and [1,0,1,1,0,0,1,1], etc.

*6a) If all bits in the first half, list[1..n], are equal, then the list has been reduced as far as possible, and Rule 7 is called.*

Thus lists which conform perfectly with sequences produced by a binary counter bit are identified. For example, the sequence:

[0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1]

may be generated using Bit 2 of a binary counter.

*6b) If not all bits in list[1..n] are identical, then use the  $(m + 1)$ th counter bit (from Rule (5)) XORed with whatever bit is chosen by halving the list again and returning to (2).*

Rule 6b deals with the other possibilities from Rule 5. Any list which has the two halves non-identical, but logically inverse, and not all entries in one half the same, is the XOR function of the  $(m+1)$ th counter bit, with whatever is produced by halving the list again and returning to Rule 2. The XOR gate will allow the first half of the sequence to be inverted after a constant number of bits.

*7) If the first bit in the list is a '1', then negate whatever counter bit, or combination of bits, has been chosen.*

This implies that all bit sequences produced by counters begin with a '0', which is quite natural.

8) Print out the connections from counter bit(s) to address bit, and start at (1) with the next most significant address bit.

Once this process has been completed for all address bits, we have a list of connections from counter bits to address bits - the mapping, or transform - which will produce the correct sequence of addresses with the minimum of logic.

### 4.3.3 Logic synthesis

Logic synthesis is done in two distinct steps in AG1. The first step is carried out on a bitwise basis across the address sequence, for any bit sequence which fails at Rule 4 above. The second step is one of global optimisation.

Firstly, the *minterm value* is determined as the binary value which appears *least* in the bit sequence. Then an algorithm iterates down the bit sequence and for every minterm a new logic function is produced in terms of a product of binary counter bits. Figure 4.11 shows the relationship between the binary counter which will be used to feed the logic, and the bit sequence to be produced.

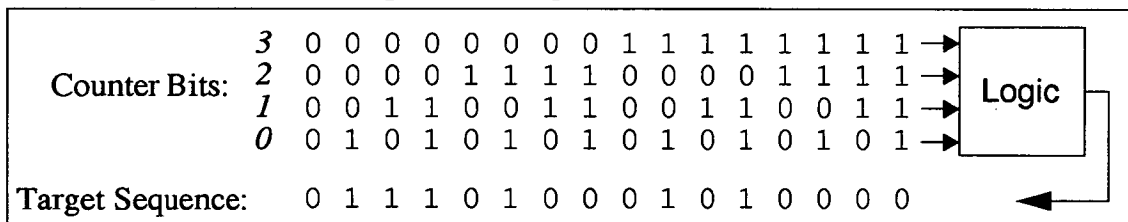


Figure 4.11 Counter output against target bit sequence.

In order to get the minimum number of counter bits required to generate each minterm, an increasing number of bits of the counter are examined, and all possible combinations of each set of bits are tried, until the selected counter bits' values only ever correspond to a minterm. This is illustrated in Figure 4.12 where a *mask* is created to select the counter bits and the bit *pattern* seen through the mask appears at several locations in the count sequence. Only if all locations correspond to minterms is the combination of, and values of unmasked counter bits accepted, and the next minterm examined. Masked counter bits are stored in the pattern as '0's.





Counter Bits:	3	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
	2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Target Sequence:		A	B	C		D				E		F		X			
		0	1	1	1	0	1	0	0	0	1	0	1	0	0	0	0
Mask:		at A gives $\overline{\text{Bit1}}.\text{Bit0}$ which corresponds to minterms at D and E but not at X, so reject this mask (0011).															
Mask:		at A gives $\overline{\text{Bit2}}.\text{Bit0}$ which corresponds to minterms at C, E and F and to no maxterms, so the mask (0101) and pattern at A (0001) define the logic required for A. (The same would be found for C,E,F).															

Figure 4.12 Masks and patterns for minterms.

Once each minterm has been defined in terms of a mask and pattern, in each random bit sequence of the same length, and attempt must be made to match all or part of each corresponding logic function to other minterms' logic, to get an optimised multi-level, multiple-output logic implementation of the sequence generator. A heuristic weighting system is employed to reduce the run-times of what is a complex problem, with the weights calculated as follows. Given that minterms A and B have masks  $m_A$  and  $m_B$  and counter bit patterns  $p_A$  and  $p_B$ , the weighting between A and B is taken as the number of '1' bits in

$$(m_A \oplus m_B) \cdot (p_A \oplus p_B).$$

In other words, we give a high weighting between two minterms whose logic functions are identical, and a low weighting between those which are not at all alike. Figure 4.13 shows some examples of this.

$m_A = 0101$	$m_B = 1111$	$m_C = 0101$	$m_D = 0101$
$p_A = 0001$	$p_B = 0010$	$p_C = 0001$	$p_D = 0101$
Weightings: $w(A,B) = 1$ , $w(A,C) = 4$ , $w(A,D) = 3$ , etc.			

Figure 4.13 Examples of minterm weighting.

Once all minterms have been compared and weighted against all other minterms and the weights stored in an adjacency matrix (Figure 4.14), the total of all weights for each minterm in turn is calculated. If the number of '1' bits in the corresponding mask (Number Of Bits In Mask -: NOBIM) is greater than one (i.e.: More than one counter bit is involved in its generation) then this total is normalised by multiplying it by (Maximum\_possible\_NOBIM / NOBIM), and the result is stored in another, one dimensional table, as the true weight for that minterm. If only a single counter bit is involved (NOBIM = 1) then the true weight should be zero, since we will use the logic for the minterm with the highest weight to help generate other minterms, and the single counter bits are already available. Figure 4.14 gives the weightings for the sequence in Figure 4.12.

Choosing the minterm with the highest true weight (A), we then give it a unique function number, and then search through the original weight table for any similar logic functions. If the weight between two minterms is equal to the Maximum\_possible\_NOBIM, then they are given the same logic function number (C, E and F), but if the weight is smaller, but still > 0, and

$$(m_A \cdot m_B) \cdot (p_A \oplus p_B) == m_A,$$

then the current function number is stored alongside the other minterm as part of a sum of products. The weights against any completely specified minterms are zeroed in the original weight table, and a new set of true weights constructed, until no weights > 0 remain. Then any remaining minterms not so far given a function number have this added, and finally the logic functions are extracted from the mask and pattern information and printed out. Figure 4.14 shows the logic generated for the example.

Minterm:	A	B	C	D	E	F		
A		1	4	3	4	4	$m_A = 0101$	$m_D = 0101$
B	1		1	0	1	1	$p_A = 0001$	$p_D = 0101$
C	4	1		3	4	4	$m_B = 1111$	$m_E = 0101$
D	3	0	3		3	3	$p_B = 0010$	$p_E = 0001$
E	4	1	4	3		4	$m_C = 0101$	$m_F = 0101$
F	4	1	4	3	4		$p_C = 0001$	$p_F = 0001$
True Weights	32	4	32	24	32	32		

**Output:**

f1 = Bit2bar.Bit0  
f2 = Bit3bar.Bit2bar.Bit1bar.Bit0bar  
f3 = Bit3bar.Bit2.Bit1bar.Bit0  
f1+f2+f3 ==> Target Sequence.

**Alternative Output (b(1) = Bit b of binary counter (modulus 1)):**

f1 = 2(1)bar.0(1)  
f2 = 3(1)bar.2(1)bar.1(1)bar.0(1)bar  
f3 = 3(1)bar.2(1).1(1)bar.0(1)  
f1+f2+f3 ==> Target Sequence.

Figure 4.14 Weights and true weights for example sequence.

#### 4.3.4 Output format

The output produced by AG1 is very simple to understand. Each bit of the address word is described in sum-of-products form, which can vary from involving a single binary counter bit to a multi-level logic description using several counter bits. The use of inversion and EXOR gates is also made perfectly clear. Figure 4.15 contains the description of a benchmark circuit, WGT(5) [69], which must produce a binary count of the number of '1' bits in a five-bit input word. Note that the modulus 32 counter is equivalent to a 5-bit binary counter.

```
f1 = -5(32).-4(32).-3(32).-2(32)
f2 = -5(32).-4(32).-3(32).-1(32)
f3 = -5(32).-4(32).-2(32).-1(32)
f4 = -5(32)bar.-4(32)bar.-3(32)bar.-2(32)bar
f5 = -5(32).-3(32).-2(32).-1(32)
f6 = -4(32)bar.-3(32)bar.-2(32)bar.-1(32)bar
f7 = -5(32)bar.-3(32)bar.-2(32)bar.-1(32)bar
f8 = -4(32).-3(32).-2(32).-1(32)
f9 = -5(32)bar.-4(32)bar.-2(32)bar.-1(32)bar
f10 = -5(32)bar.-4(32)bar.-3(32)bar.-1(32)bar

-1(32) exor(-2(32) exor(-3(32) exor(-4(32) exor(-5(32)))) ==> adbit 0

not(f4 + f6 + f7 + f9 + f10 + f1 + f2 + f3 + f5 + f8) ==> adbit 1

f1 + f2 + f3 + f5 + f8 ==> adbit 2
```

*Figure 4.15 Example of output from AG1 - Description of WGT(5).*

#### **4.4 Address generators designed using AG1**

All the address generators described in Section 4.2 were mirrored by AG1, and it is pleasing to note that the synthesised designs are identical to those produced manually. Examining the address generators in Figure 4.6 and Figure 4.8, a useful fact becomes apparent - we do not need such a large (64kword) memory. As long as the original data is available somewhere (if the image has not changed), then we need only store one sixteenth of the whole image at one time, and we can discard the top three bits of the address as unnecessary, inverting the fourth and so reducing the memory size drastically. This conclusion can be reached from the fact that all four top bits have no shuffle applied.

#### **4.5 Comparisons**

The logic synthesis tool, which was coded in a matter of weeks, nevertheless stands up to comparison with other logic synthesisers.

One benchmark example used was WGT(5), which must output the number of logic '1's in its 5-bit binary input. Its logic circuit as synthesised by Gatemap [71] is shown

in Figure 4.16, and the equivalent logic as defined by AG1 appears in Figure 4.17. The chain of exclusive OR gates was created by the core algorithm of AG1, while the other logic was synthesised by the logic synthesis part -: A joint effort.

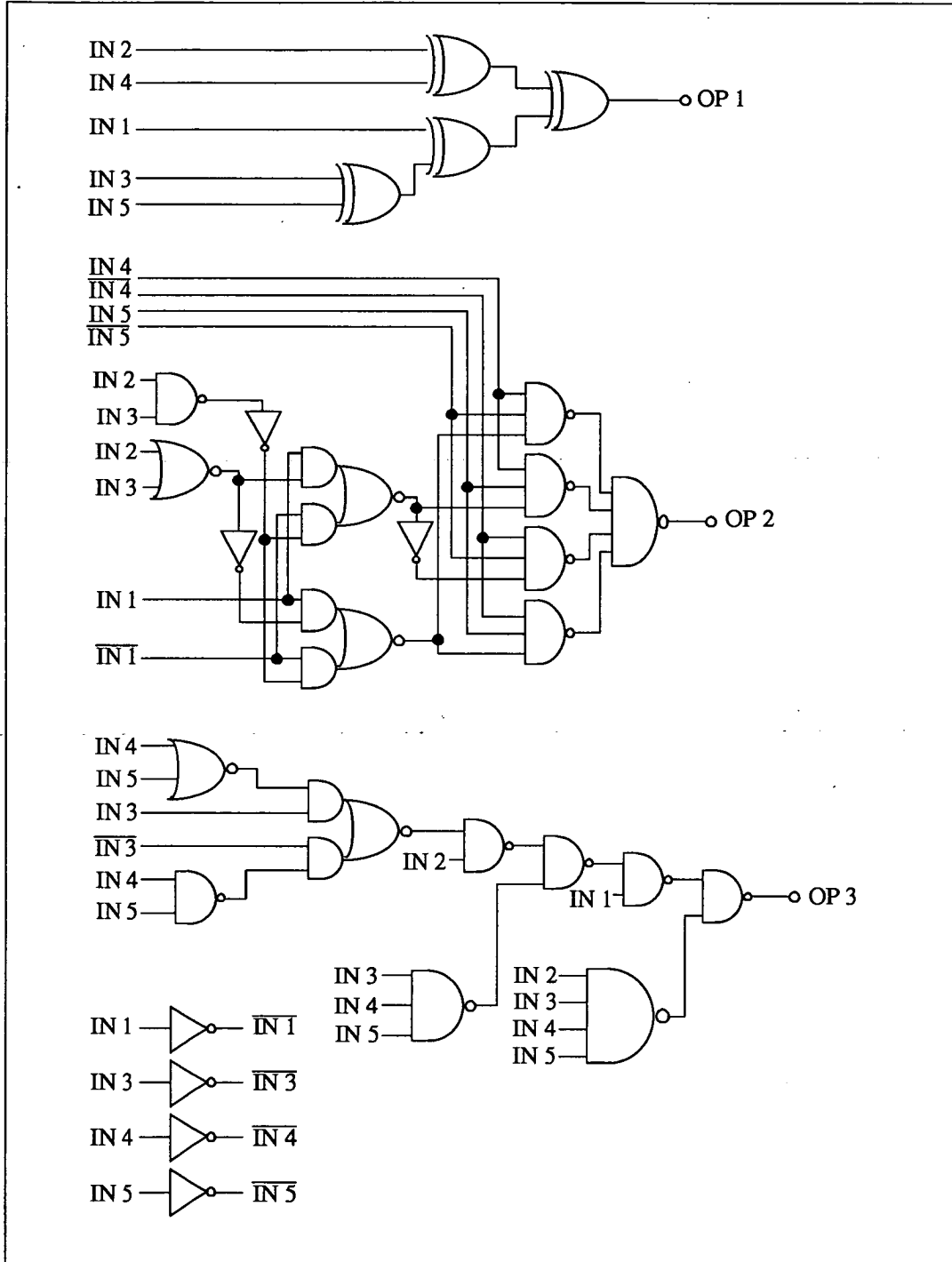


Figure 4.16 Logic Synthesis results for WGT(5) benchmark by Gatemap, which requires 96 transistor pairs in CMOS (8 per EXOR gate).

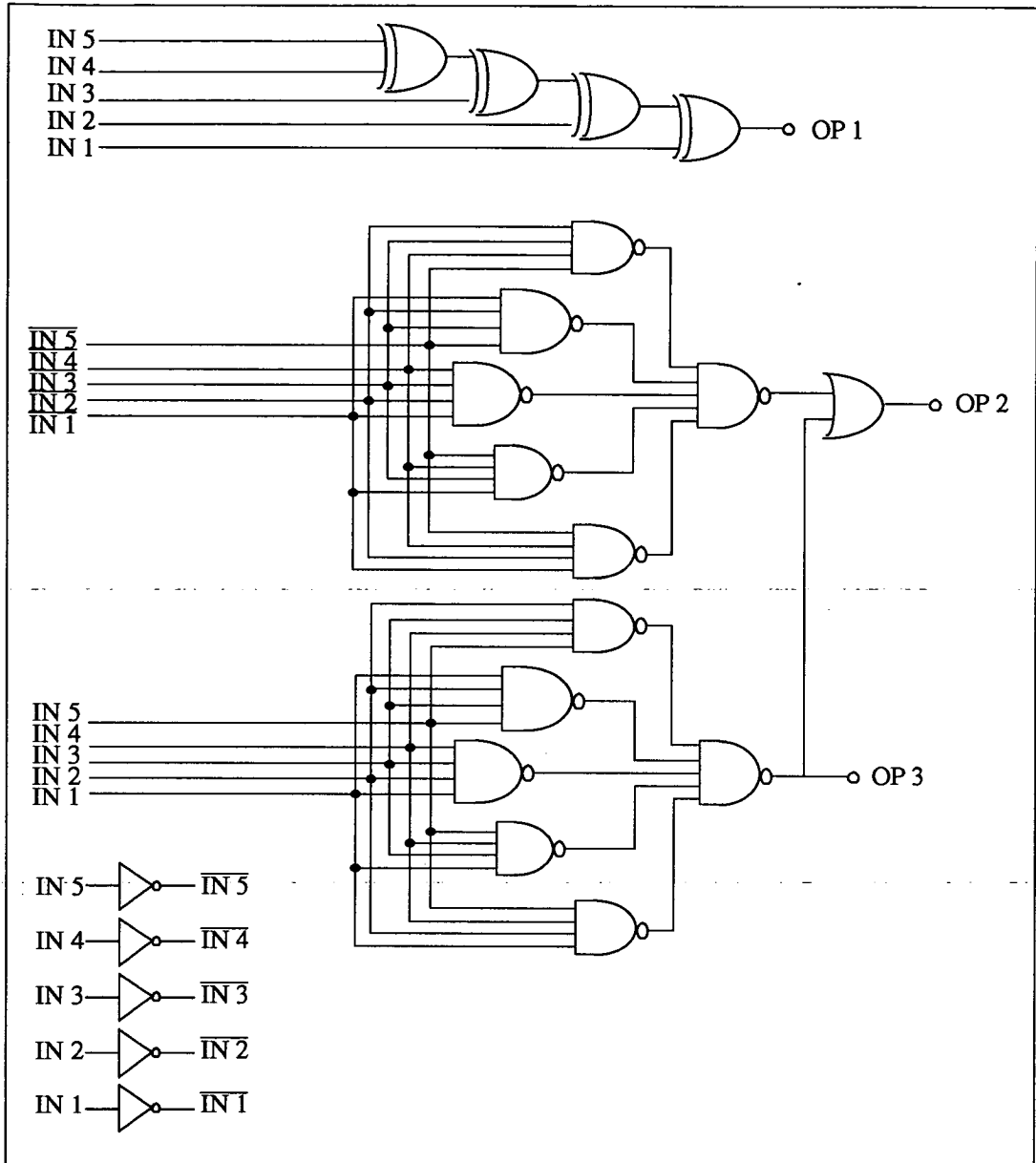


Figure 4.17 Logic synthesis result for WGT(5) by AG1, which requires 89 transistor pairs.

The address generators described in Section 4.4 matched exactly those designed manually, and those which were not originally designed manually were carefully checked. Producing the same results as a human designer is no small victory for AG1, and coupled with the marked reduction in design time, as described in Table 4.1, AG1 is, for all its limitations, a very useful tool.

Example	Manual	Gatemap	AG1
WGT(5)	10 mins (approx)	3.2 mins.	< 1 sec.
Image Filters	10 hrs (approx)	N.A.	<10 mins. (incl. recompilation)

Table 4.1 Design-time comparisons.

## 4.6 Use of the 'C' programming language

The 'C' programming language was chosen for AG1, simply because it was the language best known to us at the time of coding. Despite this, 'C' was found to be almost perfect for the job, with its built-in array pointers and useful bitwise functions, and although strong typing and therefore code security is not encouraged, the language was found to be quite amenable to algorithm development.

## 4.7 Comments

Obviously, despite all its complexity, AG1 is not a generally useful tool, in that the sequences fed to it must have length  $2^n$ , and any "good" solutions found, owe this entirely to the binary characteristics of the original access patterns. For instance, the correlation filter memory access pattern described in Section 4.1 differed from that defined for a real correlation filter [203], in that the real filter required data in three-by-three pixel blocks, instead of four-by-four. The manual design for the real filter's address generator included two long line delays, to obtain the three vertically adjacent pixels needed. This seemed a rather crude method, and prompted an investigation into address generation using non-binary counters, which is described in Chapter 7. The logic synthesis part of AG1 has proved to be useful on many occasions, and is re-used as part of the aforementioned investigation, and consequent synthesis tool.

The worst-case complexity of the various parts of AG1 are given below:

Sequence length =  $l_s = 2^n$ ,      Sequence width (bits) =  $w_s$ .

Loading sequence =  $O(l_s)$ .

Matching sequence to solution  $\approx O(w_s * l_s)$ .

Logic synthesis (m minterms):

$$\text{Minterm detection} \approx O(l_s + \frac{3l_s^3}{2}).$$

$$\text{Minterm factorisation} \approx O(\frac{19m^2}{6} + \frac{2m^3}{3}).$$

From these figures we can deduce that the logic synthesis tool will be slowed considerably for random bit sequences greater than length  $\approx 100$  bits, while the main part of AG1 will run in linear time.



## 5 Introduction to behavioural synthesis

### 5.1 What is behavioural synthesis?

A behavioural synthesis tool will accept a description of what a processor is expected to do - its *behaviour(s)* - and generate a netlist of hardware components which will exhibit that behaviour, in a design anywhere from chip-level to system-level. Note that a processor may exhibit more than one behaviour, probably under external control. For instance, an Arithmetic Logic Unit (ALU), may add two numbers, multiply them, or utilise any other built-in functionality and so has a set of behaviours describing each function in turn.

The netlist of hardware components forms the *structure* of the design and there can be only one structure for a given component, perhaps implementing several behaviours. The hardware components themselves may have behaviours and structure, and in order to generate correct and efficient hardware solutions, the simplest components should be chosen from a library on the merit of their ability to exhibit the required behaviour(s), as well as a number of other factors. The design process is summed up in Figure 5.1.

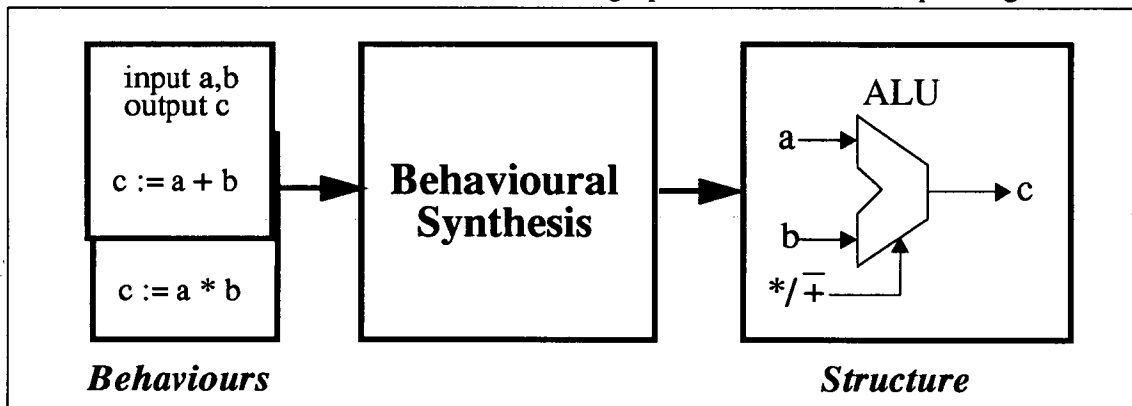


Figure 5.1 The Behavioural Synthesis Approach.

A major requirement of a behavioural synthesis system, is an ability to handle a hierarchical definition of a process [87, 93]. This allows the process to be broken down into ever-simpler processes, as in a typical manual design, and hopefully these simpler processes' behaviours will match those of some library components, which can then be used to implement those functions, as part of the overall processor architecture. A problem arises though, that decisions made at the simplified level may have major ramifications on the overall design, and the outcome of these decisions must be propagated back up through the design hierarchy - A computationally complex, and time consuming task. However, it is only with behavioural synthesis that we may

explore the design space so thoroughly, in such an efficient manner, and perhaps with parallel-processing [94], point tools [86] or some other recent development, it is possible that design time, from behavioural specification to netlist generation, could be reduced to a negligible span, by today's standards.

To get from the behavioural description to a register transfer level description, before logic and layout synthesis complete the design, is the task of high level synthesis. It is this area which will be explored in the following sections.

## **5.2 Key steps in the high level synthesis process**

Usually the behavioural description of the process will first be broken down into a simpler internal format before carrying out the tasks of allocating a number of library modules - computational resources - to be used, scheduling any computational operations onto these resources and creating the necessary memory and communications circuitry and control circuitry to support the process. These problems will be examined as part of an overview of high level synthesis systems.

Since the CMU-DA system [10] developed at Carnegie-Mellon University in the late seventies, and MACPITTS in the early eighties [109, 16], a great deal of research has gone into high level synthesis [27]. In Europe such projects include the CATHEDRAL systems of IMEC [103, 104, 105, 106], the MIMOLA project at the University of Kiel [26], EASY at Eindhoven [20], CADDY at Karlsruhe [23], SCHOLAR at Southampton [33], FIRST at Edinburgh [108], Pyramid from Philips Research Laboratories [113] and the VENUS [97] system at Siemens.

In the United States and in Canada also, much work has been carried out, especially at CMU, with their original design system and more recently with the System Architects Workbench [114]. California has also been a centre of interest, with the three universities of Southern California, Berkeley and Irvine producing the ADAM [39], LAGER [120] and HYPER [52], and VSS [116] systems respectively. IBM's Yorktown Silicon Compiler [53] and V compiler [54], and General Electric's Parsifal system [117], along with the Bridge [118] and SAM (now CHARM) [55] systems at AT & T Bell Laboratories, are typical of systems in industrial use.

The Canadian universities of Waterloo and Carleton have also been at the forefront with the SPAID [19] and HAL [15] systems respectively. Inevitably, a large Japanese effort is underway, and work is also going on in India, France and of course in Edinburgh with the University's recently ended SARI project [34], and the author's own MC<sup>2</sup> system [50].

There follows a description of the key steps in high level synthesis, and their implementations in the various synthesis systems.

### 5.2.1 Capture of behaviour

Capture of behaviour from the HDL or BDL into the internal format is normally done by some sort of parser, and several classifications of the internal control and data flow representations exist.

At IMEC a *tree based* description is derived directly from the Silage input language [159] which is actually a signal flow graph description, and the LAGER system also uses Silage. MIMOLA also uses a tree structure, parsed from its name-sake input language [160].

*Separate data and control flow graphs* are used by Camposano [22] for the synthesis of VHDL behavioural models, *ASM* (Algorithmic State Machine) *charts* in Slicer [38], and in the VSS (VHDL Synthesis System). The latter use groups of data flow operations which need no control, as *basic blocks* in the control flow graph, while the former includes every operation.

A *semi data flow graph*, which requires less analysis of the input description, is usually represented as a bipartite graph [168], and this method is used to create four separate data models in the Design Data Structure (DDS) of the ADAM system: For data flow; Timing and control; Logical structure; Physical Structure. The first two are based on bipartite directed acyclic graphs, and a multi-graph formulation is also utilised in the CADDY system, where three graphs for data flow constraints, data flow itself, and for timing constraints, all share the same nodes. The nodes correspond to the operations in the DSL input language [21]. The YSC's YIF format is another manifestation of a semi data flow graph.

Using a *combined data and control flow graph* with fork and merge nodes can lead to a more coherent data structure, and both the EASY system and the System Architects Workbench have this distinction. The combined graph allows much simpler global optimisation, by avoiding the need for basic block and their boundaries. Repetition (loops) can also be handled better, rather than being represented outside the basic blocks. The SCHOLAR system also uses a combined data and control flow graph derived from its input description, as does FLAMEL [35] which deals with memory in a global sense, over all basic blocks.

### 5.2.2 Scheduling

Scheduling of operations onto hardware resources is required whenever a maximally parallel approach is not feasible. The set of operations will have dependencies on other operations for data, and these constraints are represented by the data arcs between operations in the schedule. The schedule may be constrained in length by a target number of control steps (usually the least possible) and by a target number of each type of hardware resource available (again, the least possible). There are four main methods of scheduling, including iterative operation-by-operation techniques, self-organising methods, integer linear programming (ILP) and transformational scheduling which tries to improve on an existing schedule. An example of a schedule is shown in Figure 5.2.

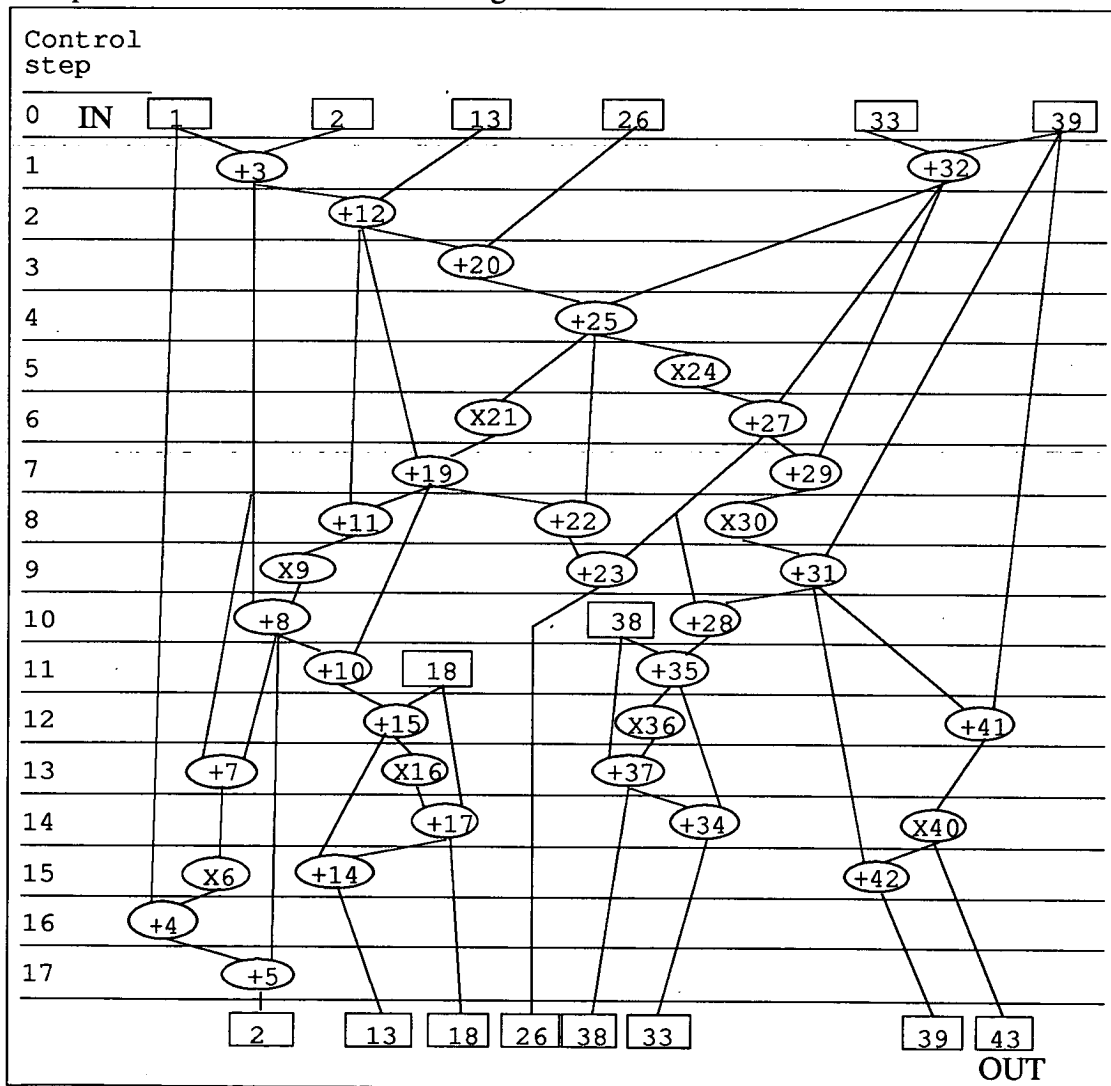


Figure 5.2 A example of a (cyclic) schedule.

Most scheduling algorithms cannot handle loops or hierarchical behaviours and so operations are commonly grouped into basic blocks, to be scheduled as possible. If these basic blocks form quite large schedules then this practice works fine, but if the blocks are small, and there are several of them in a control-dominated design, then the number of basic block boundaries will cause a global scheduling problem for a block-by-block scheduler. A hierarchical extension to this [45] schedules each basic block to get a global timing, which is tailored by successively removing expensive and little-used resources. Pipelining a design to increase the data throughput is a complex scheduling problem, and only a few pieces of work, such as SEHWA [39] and that by Hwang [43] and Mallon [44] have attempted to solve the problem.

Where a lot of chained operations are desired, for a fast (short) schedule, a *path based* scheme [32] can help tremendously. Each conditional data flow path derived from the control flow graph is scheduled separately, and loops are treated as ordinary straight line data flow which may or may not be executed.

In the methods for scheduling *within* basic blocks, iterative techniques are the most prevalent. These are divided between those that examine each control step in order and those that iterate through the operations instead. In the former group, *as soon as possible* (ASAP) scheduling, as used in the first CMU-DA system [29], which operates under no hardware constraints, and also *list scheduling* techniques, are common. List scheduling involves applying hardware constraints to delay certain operations from an ASAP implementation, using some heuristic to choose which of the operations are to be delayed. A significant improvement was made in the HAL system [41] with *force directed list scheduling*. This was a logical extension of previous work on SEHWA [39] which uses a combination of list schedulers to get some idea of the *urgency* of operations (and can handle pipelined designs), and this is similar to the approach taken in the ELF [14] and CSTEP [114] schedulers. The Slicer scheduler [8] uses both the ASAP and the ALAP (*as late as possible*) schedules to determine the *mobility* of operations in the time domain. More recent work includes a time-constrained list scheduler [57].

The iterative algorithms which examine each operation in order, include critical path schedulers and distribution based schedulers. In [17], Parker schedules all operations on the critical path(s) first, and then uses a mobility factor to assign control steps to the others, while the CATHEDRAL II scheduler, ATOMICS [101], the length of each critical path is taken into account. *Force directed scheduling* (FDS) [42] distributes all the operations not on the critical path using a parallel of spring tensions as a guide, and the CASCH scheduler [24] uses a more conservative statistical method.

An extension to FDS in PHIDEO [61] includes memory costing in the overall “tension” calculations.

Integer linear programming (ILP) methods have been applied to scheduling with some success for short schedules [58, 59]. This involves solving a linear program to minimise some cost. Recently a more efficient ILP formulation has been reported in [60].

Transformational scheduling applies either serialising techniques to a maximally parallel starting schedule, or the corresponding parallelising transformations to a serial schedule. Examples of these are found in the YSC [53] and CAMAD [78] systems respectively. A branch and bound method for optimal parallel to serial transformation is discussed in [7]. SCHOLAR uses a rule base for the same sort of transformations and FLAMEL uses a rule base to transform the *behavioural description itself*, to greatly increase possible parallelism in the eventual design.

Other scheduling methods include the application of simulated annealing algorithms [51], and a branch and bound search for an optimal solution in SCHALLOCC [6], the scheduler in the CHIPPE system [38], which uses the connectivity binder SPLICER [9] to prune the search space by costing.

### 5.2.3 Resource Allocation

Once the schedule is available, it is possible to determine the type and required multiplicity of resources needed, and this may have been a constraint during the scheduling task itself. It is possible however, that different resources’ functions may be combined, in an ALU for instance, and this may produce a better or worse design. This is an area normally left to the human designer, or ignored altogether, but in both the MIMOLA system and the ADPS [59] system, module selection is done automatically using an ILP formulation to reduce the global cost of the resources.

### 5.2.4 Data Path Synthesis

There are three specific tasks in constructing the data path to implement the scheduled operations in the correct order. Memory must be created to store temporary values, the interconnection between that and the computational resources must be added and operation must be assigned to a distinct resource where a choice is available.

These three stages may be merged, but the complexities involved make this approach infeasible even for medium-sized problems. A serial approach can apply each algorithm in turn to the whole design at once - a global approach - but inherent



interdependencies between each subproblem can cause problems here too, as can the order of application of the algorithms within the general synthesis scheme.

The algorithms themselves may be based on several different methods: Global algorithms, based on clique partitioning/covering methods; Iterative and greedy algorithms; Rule based schemes; Branch and bound search techniques; ILP formulations; Logic synthesis. A further approach, of interconnect-driven schemes, considers wiring as a first-order effect during data path synthesis.

Most *global algorithms* utilise clique covering methods to cover an undirected graph, and Tseng [29] was the first to apply clique partitioning, based on heuristics, to first complete the memory (register) assignment, then operation assignment and finally interconnect synthesis. The heuristic approach does well in general but several special cases allow exact algorithms to be used. The Left Edge algorithm (See Figure 5.3), devised originally for a channel routing problem, can be used to allocate and assign the exact minimum of registers, but cannot handle cyclic schedules [167]. An algorithm is described in [4] which deals with this as a multi-commodity network flow problem.

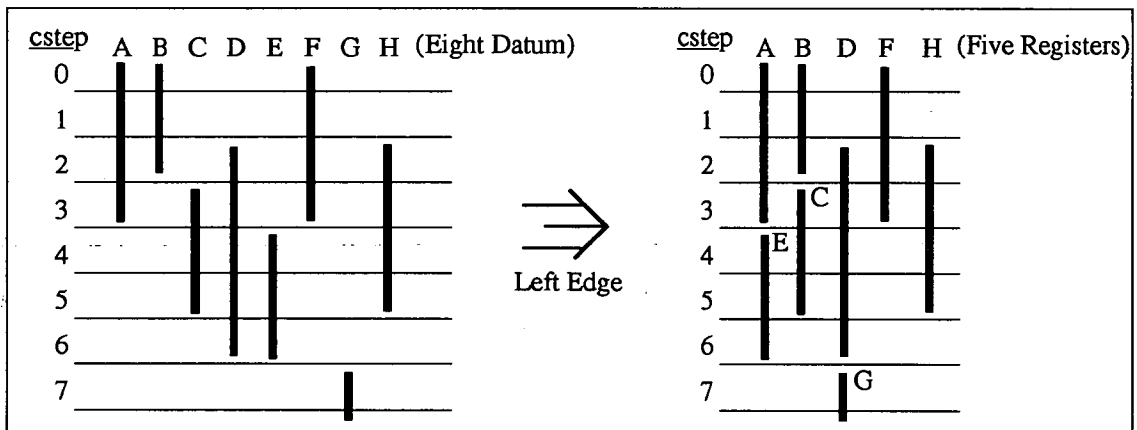


Figure 5.3 The Left Edge algorithm groups data lifetimes to registers.

To allow some communication between the synthesis tasks, a correlated clique cover approach is applicable, and is used in CADDY, HAL and EASY. In CADDY, the order of tasks is register, operation and then bus assignment, and all are based on colouring *restriction* and *preference* graphs [169]. In HAL the operation assignment is done first, using functional partitioning, and is followed by register allocation using clique partitioning weighted by interconnection patterns, whose synthesis completes the data path. For EASY the first synthesis scheme starts with operation assignment using heuristics to cover cliques with the largest weight. Then memory synthesis is attempted using an improved left edge algorithm to reduce the associated cost of

interconnect, and another clique covering algorithm as an optimisation stage. Finally the interconnect is added.

A mainly rule-based synthesis scheme is used in the DAA [13] for the original, global memory synthesis, before partitioning the design using clique partitioning, and then optimising at local and global levels, again using the knowledge-base. SCHOLAR assigns registers to variables using a rule base, and then a point to point interconnection network is added before finishing with operation assignment using a clique partitioning method similar to that in FACET [29], but extended to handle concurrency. CATHEDRAL II's *Jack-the-Mapper* [107] is another example of a rule-based memory synthesis scheme, which also examines address generation as part of its task. Operation assignment can be influenced by pragmas from the designer and is completed along with register assignment by the ATOMICS micro-code scheduling tool. MC<sup>2</sup> uses a rule base to first assign temporary variables to dual port register files, then to assign operations to specific resources as a side effect of interconnection minimisation. Finally, variables are assigned to specific locations within the register files, bearing address generation costs in mind, and the address sequences and optimised control sequences are then produced automatically.

A branch and bound scheme is used in MIMOLA, where register assignment is done first, for any straight line code. Operation assignment forms part of the communications synthesis stage using a branch and bound algorithm on one control step at a time (starting with the *busiest* cstep). SPLICER includes dynamic register allocation with operation assignment in the interconnection synthesis stage, again on one cstep at a time. Solutions are found quickly by this method, and then improved on using backtracking.

Integer linear programming models are also included in MIMOLA now [25] for register and operation assignment during scheduling, but this only works on a time-local basis (step by step).

Iterative approaches include that taken in the ADA to standard cell compiler [14] which iterates through the operations in the scheduled order to assign them to resources. A similar approach is taken in MABAL [1, 2] but limited reiteration is possible, and both registers and operations are assigned together, using the partial interconnect's cost as a guide. The EMUCS system [115] assigns operations on a step by step basis using heuristics to order them (within a control step), while CHARM [56] iteratively performs a form of graph colouring, again using heuristics, with the register and



communications synthesis built into the costing algorithm. Here sets of compatible operations are constructed which will eventually share the same resource.

Logic synthesis systems such as the YSC and HERCULES [119] can play a part in data path synthesis, especially for control-dominated designs, but lack the intimate design knowledge of most other methods.

Although many of these systems do take interconnect costs into account, it is not treated as an integral part of the synthesis scheme, and so work has been done on data path synthesis schemes driven primarily by these interconnect costs. Park [18] describes a method where short sequences of operations are assigned to the same partial structure, using a heuristic clique partitioning algorithm on pipelined designs, to reuse as much interconnect as possible. Register assignment is done at the same time, implicitly, but any remaining interconnections must be added manually. In EASY again [3], interconnect synthesis is done on two levels of hierarchy. After storage operation grouping, which groups operations' data arcs into single port register files, perhaps using a two-phase memory access scheme, register allocation takes place using a bipartite graph edge colouring algorithm. Finally a simulated annealing algorithm is used to assign operations to resources, which should reduce the costs of local interconnections between memory and computational resources. MC<sup>2</sup> adopts a very similar approach, but uses a rule base to produce control-free interconnect from resources to memories, and to minimise the cost of the rest of the communications network.

Table 5.1, shows the relative strengths and weaknesses of these different methods of data path synthesis.

Data Path Synthesis Method	+	-
Global algorithms	Formal basis; Efficient; Possibly exact	Sometimes too general
Rule based	Good for Application Specific Synthesis	Slow; Difficult to maintain
Branch and Bound	Very fast first solution	Too time-intensive
ILP	Good for small designs and for automated resource allocation	Too time-intensive
Iterative	Simple to implement	A lot of expensive look-ahead required
Logic Synthesis	Okay for control-dominated designs	Cannot synthesise efficient data operators
Interconnect Driven	Good for communication-dominated designs	Suffers from resource allocation stage

*Table 5.1 Comparison of Data Path Synthesis Methods.*

### 5.3 Controller synthesis

Once the memory and communications have been added to the design, multiplexer and register control sequences may be extracted using the schedule, and these will commonly be handed to a logic synthesis tool to be assigned to a PLA-FSM or some other controller. The SCHOLAR system is one which constructs a specialised control unit itself, as a non-deterministic FSM consisting of a Sequence Controller and some combinatorial logic. Where memory address sequences are required, first the assignment of values to actual memory locations may need to be completed, and this may have great bearing on the cost of generation of these sequences. Control steps where one does not care about a certain control value may be exploited to allow sharing and simpler generation of these also. This is examined in Section 6.4.3.2 and Section 6.4.3.3.

### 5.4 Resulting design format

With the general simplicity of data path components, it is amazing how badly reported are the results from some of the data path synthesis systems described above.

The resulting design, excluding the controller, should be a netlist of computational inputs and outputs, memory, buses and steering logic, as detailed in Figure 5.4.

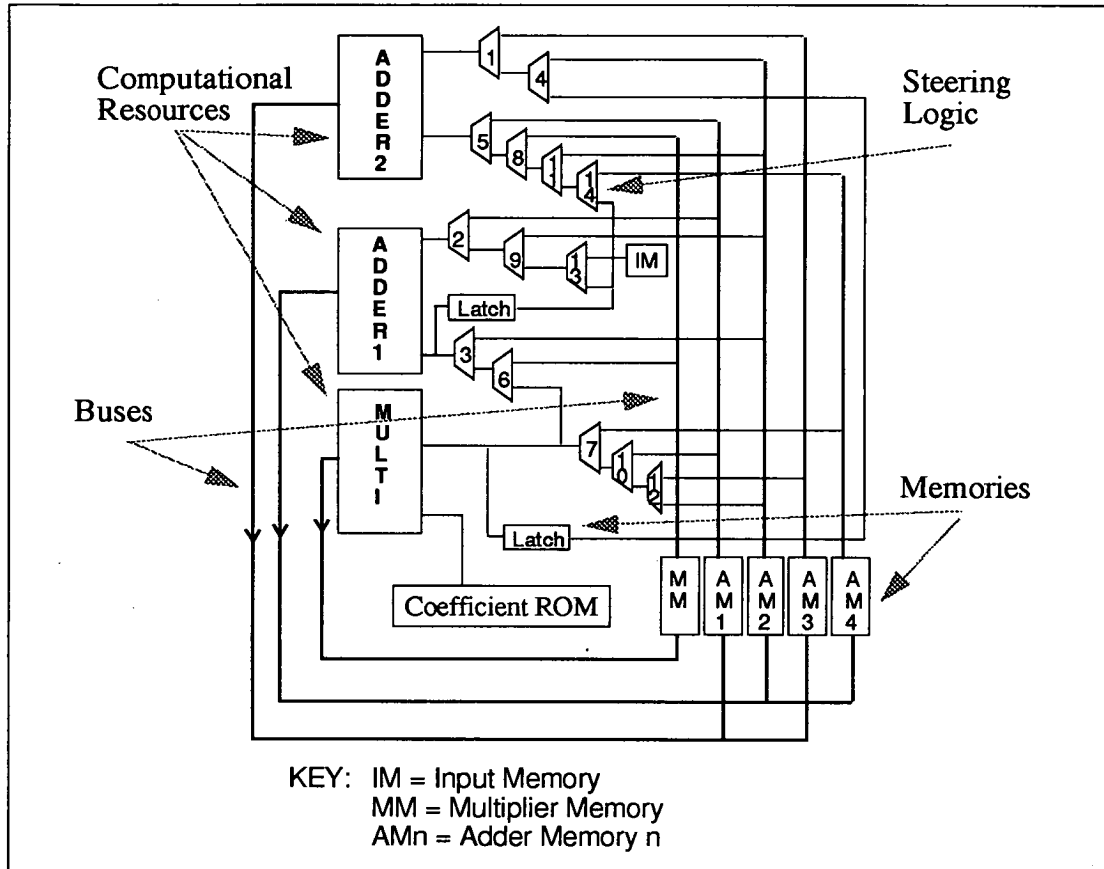


Figure 5.4 A example of a design resulting from data path synthesis.

## 5.5 Impact on address generation

Imagining now that we have synthesised a register transfer level description of the required design, and that the registers have been grouped into register files as their access times allow, let us backtrack through the synthesis process to find any stages which have effected the address generation for those register files.

Obviously the grouping and subsequent sharing of registers within the files ultimately defines the address sequence, but the stage of operation assignment may restrict this grouping due to interconnect costs. Scheduling is by far the most important stage as far as memory is concerned, since it is here that data arcs between the operations are stretched or squashed, producing different storage requirements, and perhaps it would be better to schedule the data arcs onto a predefined memory architecture, whose address generation properties are known already. The PHIDEO

system [61] attempts to use look-ahead memory costing to influence the scheduling task, but cannot attach more than a vague guess at address generation costs.

## **5.6 Comments**

All the systems described deal with bit parallel architectures, but work has also been done on bit serial high level synthesis. FIRST [108] at Edinburgh for bit-serial design, and the CATHEDRAL I system for bit serial digital filter design were the earliest systems, and since then Hartley and Jasica [48] and Cheung and Leung [49] have reported such work.

## **6 A heuristic approach to memory, control and communications synthesis, for scheduled algorithms**

### **6.1 The joy of synthesis!**

Without a doubt, the area of behavioural synthesis is an extremely interesting one, offering constant challenges to an automation designer. Once the first part of the synthesis problem has been expanded to a dozen or more equally difficult tasks, nothing seems to be “simple” any more.

The overall motivation was that to examine the possibility of using automatic synthesis techniques to construct scheduled memory address generators, one first needs some address sequences on which to test and prove the techniques. These sequences could have been compiled on a random basis, but would then bear no real relevance to scheduled memory addressing, or they could have been produced manually for several real examples, with high probability of errors, and with some difficulty. Therefore a third option was exercised: That of designing a simple, automated synthesis system, capable of accepting a schedule and some allocation information, and of producing any address sequences needed by scheduled memory.

As coding of this synthesis system progressed however, it became obvious that much needed to be done before a realistic memory address sequence could be produced, and it was decided that the synthesis system should also produce a netlist of computational resources, memories and steering logic, as well as the control bit sequences for that logic. These were also targeted as test vectors for an address generator synthesis technique.

So the synthesis system developed, as well as becoming more general in the form of schedule required, until it consisted of three major programs, the function of which are described in Section 6.4.

### **6.2 Schedules and their scheduling method**

Scheduled memory addressing becomes a problem the moment an algorithm has been scheduled, but little or no attention is paid by the literature to either memory or memory addressing costs at the scheduling stage, with few exceptions. Scheduling methods to date have been targeted at reduction of computational hardware and controlling logic, along with an optimisation of throughput, by load-balancing and tree-based methods.

The first schedule below (Figure 6.1) is for a 5th order, elliptical, digital wave filter [15], and has become a standard benchmark for several data-path synthesis systems [1, 3, 6, 9, 19]. There are 42 computational operations in all, with 26 adds and 8 multiplications. One constant factor of each multiplication is held in a ROM, and a single multiplier, pipelined in two control steps, and with a latency of one, is available. Two adders, operating in a single control step, are also available. The filter algorithm was scheduled into 20 control steps using a Force Directed Scheduling (FDS) algorithm [41, 42], which utilises load-balancing techniques. The second schedule of the same algorithm is shown in Figure 6.2, but this time the operations have been scheduled onto slightly different hardware in just 17 control steps, by a simulated annealing approach [51]. Again there are two adders and one multiplier available, but this time a fast multiplier is used which does not need pipelining.

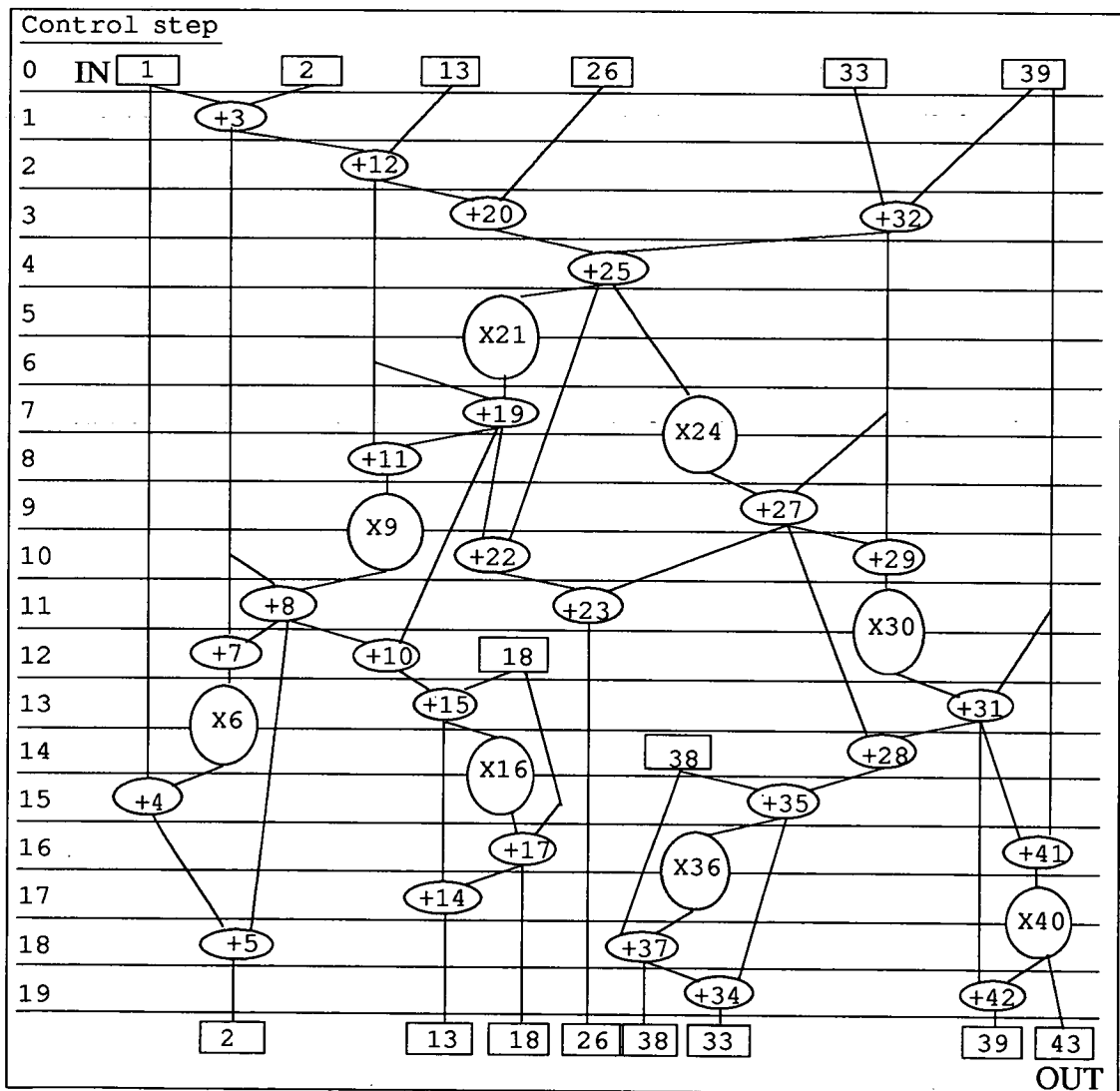


Figure 6.1 Fifth-order wave filter schedule.

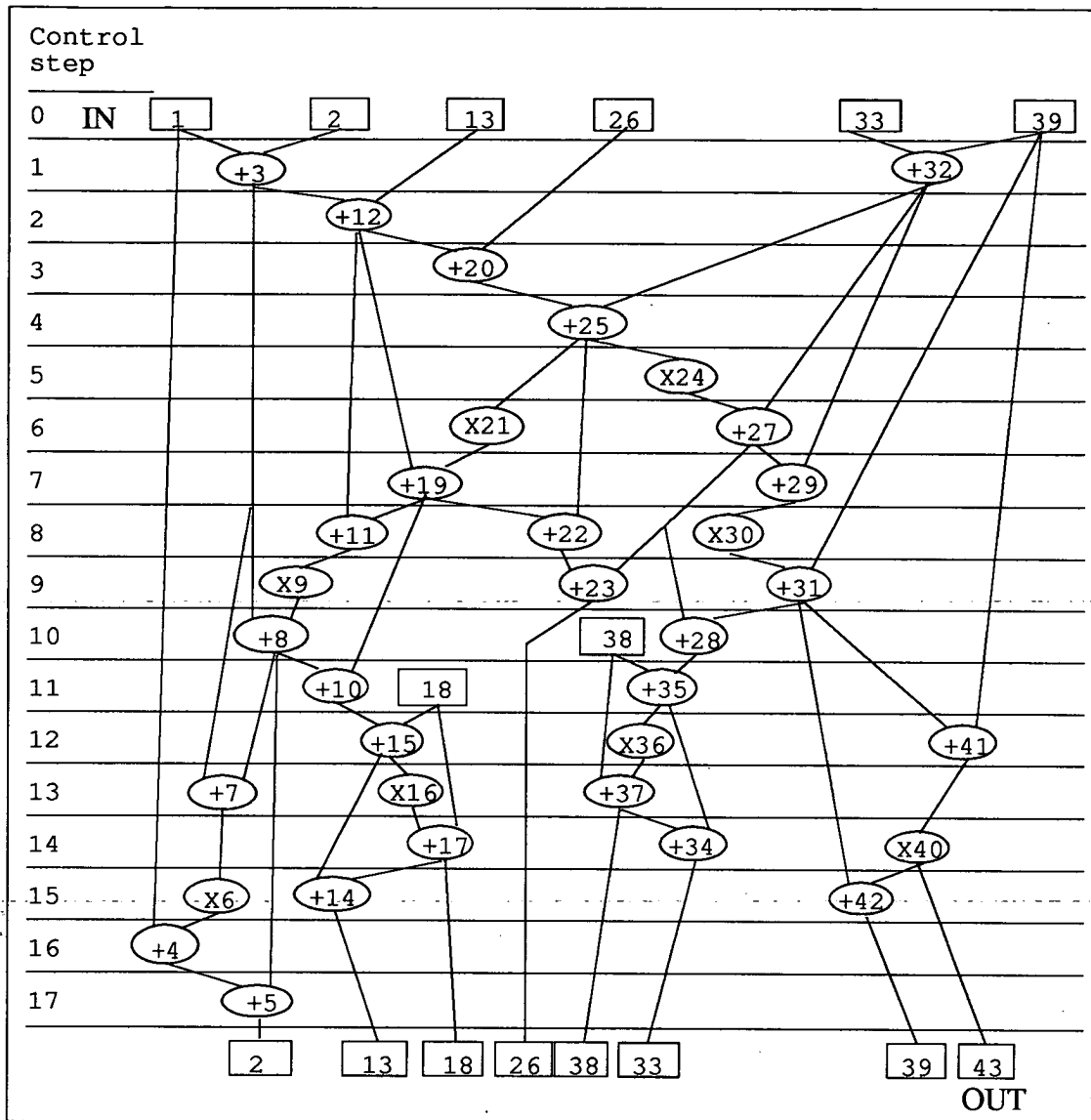


Figure 6.2 Fifth-order wave filter scheduled using Simulated Annealing.

Figure 6.3 shows a schedule of a 16-point, digital FIR filter. Here, again, the multiply operations have one input supplied by a pre-defined ROM, and in order to complete the 23 operations in 6 control steps (actually 3 control steps, through pipelining), 5 adders and 3 multipliers are needed [18].

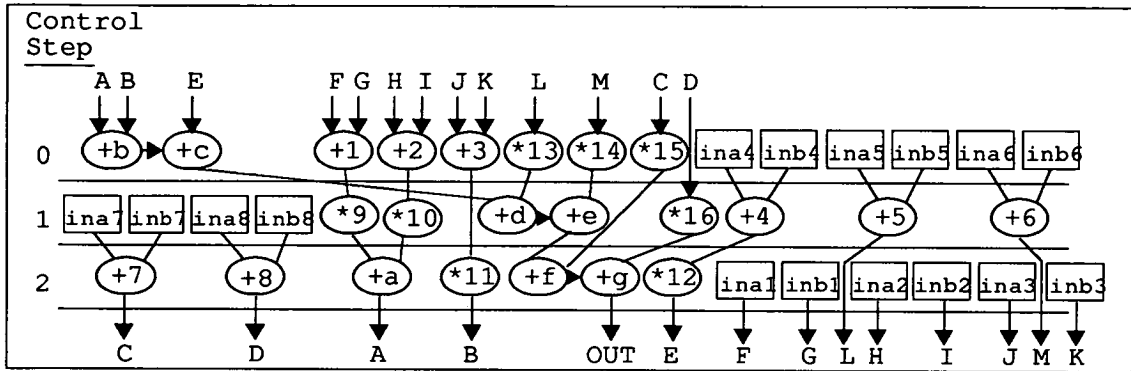


Figure 6.3 16-point FIR filter schedule (Pipelined).

A third schedule, shown in Figure 6.4 is for a Fast Discrete Cosine Transform (FDCT) algorithm, scheduled using simulated annealing into 13 control steps, and requiring two adders, two subtractors and two multipliers. All multiply operations have one input fed from a (hidden) ROM.

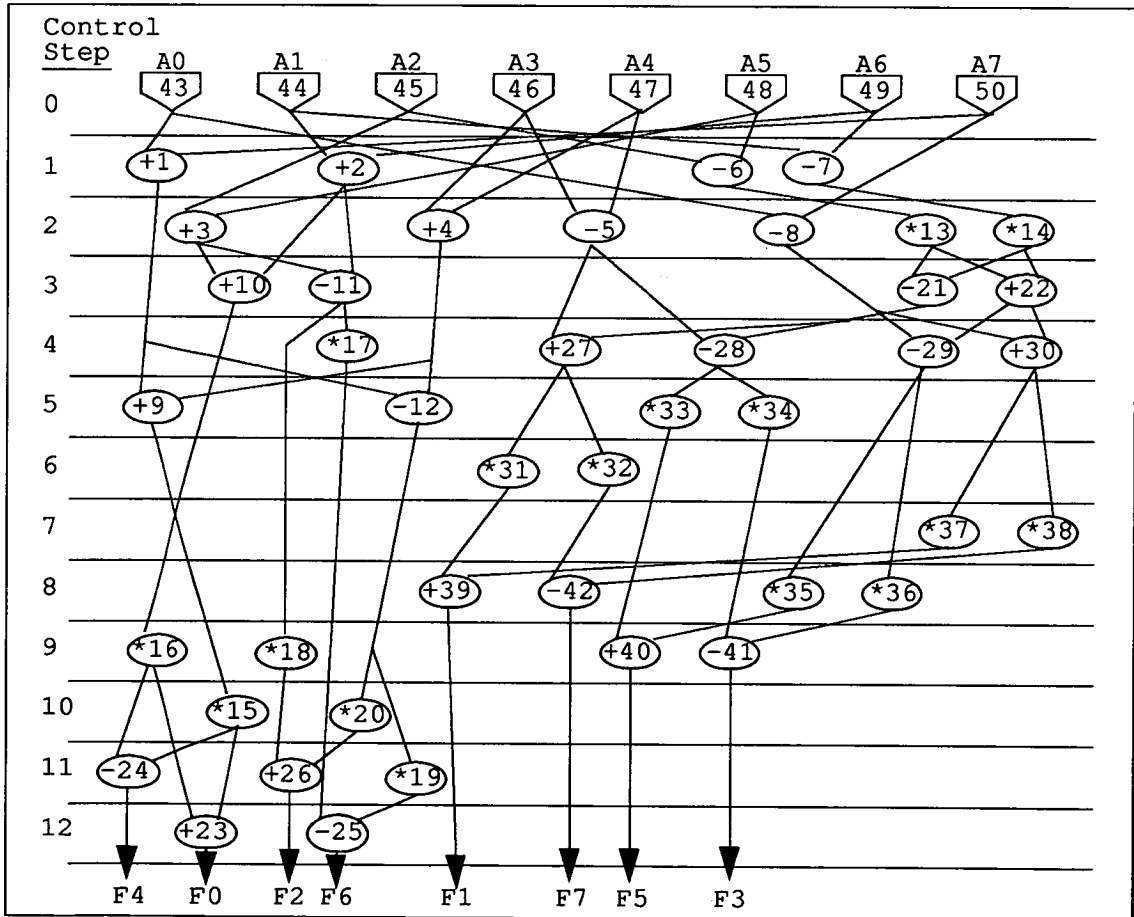


Figure 6.4 Fast Discrete Cosine Transform schedule.



The fourth example is that of a differential equation, and the schedule in four control steps is shown in Figure 6.5.

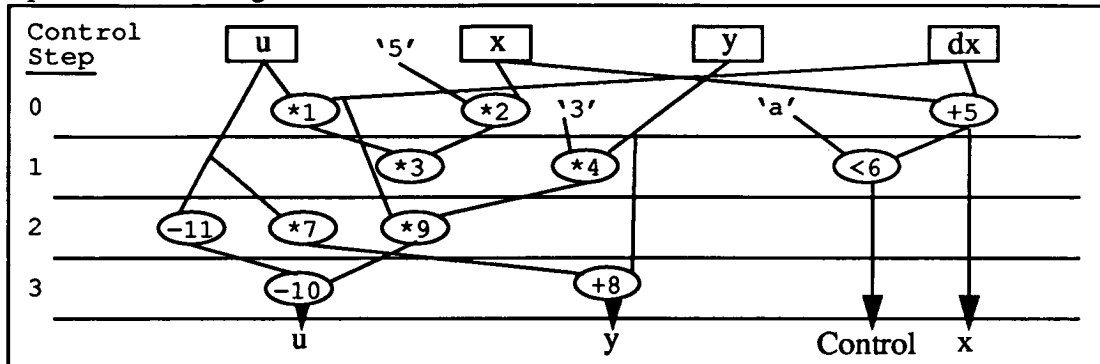


Figure 6.5 Schedule for the differential equation example.

### 6.3 Constraints on this approach

As it stands, the synthesis system described below, MC<sup>2</sup>, has certain constraints. Schedules containing fork and merge operations - the equivalent of an IF..THEN..ELSE statement - cannot be handled properly, due to restrictions in the assignment stage. These schedules' corresponding memories would need data-dependent addressing schemes, which are not considered for automatic synthesis here.

The allocation of an ALU resource cannot be handled correctly by MC<sup>2</sup>, again due to restrictions in early stages of synthesis.

Operator chaining is allowed, as are ROM definition (to hold constants), cyclic and acyclic schedules, multicyclic delays, and multicycling or pipelined computational resources.

### 6.4 MC<sup>2</sup> - Memory, Communications and Control synthesis of scheduled algorithms

There follows a description of the MC<sup>2</sup> tool, from its input format through the major steps in the synthesis process, to the output format and some synthesised examples. Given a description of an operational schedule along with some allocation information, MC<sup>2</sup> generates a netlist of hardware components as well as the control and memory address sequences required to make the design function.

#### 6.4.1 Schedule data-base

The schedule which is handed to MC<sup>2</sup> is described in terms of resources, operations, operation timings and data flow constraints.

The data-base facts:

```
cstep0(X) .  
cstepn(Y) .
```

define the first control step of the schedule to be X and the last control step to be (Y-1).

*Resource* declarations follow the format:

```
res(Restype, Nres, Ninputs, [Widths_of_ports], [Lists_of_equiv_inputs]) .
```

Restype is the type of resource ('+', 'adder', 'mult', etc.), and Nres is the number of resources of that type available. Ninputs is the number of input ports to that resource type (Number of outputs is assumed to be unary) and a ROM resource has zero inputs. The widths of these input ports are defined in bits for each port, and then lists of interchangeable inputs describe the commutativity of the resource's function.

*Operations* are defined thus:

```
opr(OpUID, Restype, Nres, Wtime, [Rtimes]) .  
start_time(OpUID, Start) .  
reads_from(OpUID, [Other_OpUIDs]) .
```

OpUID is usually a number or letter for each operation. If the operation is actually providing a constant from a ROM, then the OpUID should be of the form:

```
'c.Constant_value'.
```

Restype is the resource type on which this operation can occur, and Nres is as for that corresponding resource definition. Again, if a ROM is being used then Nres should equal the number of constants to be stored, although the actual size of the ROM may be reduced at a later date. Wtime is the clock tick during which the operation *terminates* in the schedule (i.e.: When its output data becomes available), and the list of Rtimes are the clock ticks when that data is actually required by other operations.

The starts\_at/2 fact defines the clock tick during which the operation starts, and the operations which feed data to the one in question are listed in the reads\_from/2 fact, which is also a tentative port assignment. If an operation is to receive data from another operation in the same control step - Chaining - then the supplier's UID should be preceded by a 'ch.' in the reads\_from list. Multicyclic operations are handled by providing different start and Write times for that operation. Figure 6.6 shows extracts from the FIR filter schedule description.

The schedule data-base will form the basis for synthesis to which structural information will be added to the system.

<pre>cstep0(0). cstepn(3).  res_list(*, []). res_list(+, []). res_list(in, []). res_list(rom, []).  res(+, 5, 2, [8, 8], [[1, 2]]). res(*, 3, 2, [8, 8], [[1, 2]]). res(in, 6, 1, [8], []). res(rom, 3, 0, [], [[]]).  opr(a1, in, 6, 2, [0]). opr(b1, in, 6, 2, [0]). opr(a2, in, 6, 2, [0]). opr(b2, in, 6, 2, [0])..  :  opr('c.1', rom, 3, _, [1]). opr('c.2', rom, 3, _, [1]). opr('c.3', rom, 3, _, [2])..  :  opr(1, +, 5, 0, [1]). opr(2, +, 5, 0, [1]). opr(3, +, 5, 0, [2]). opr(4, +, 5, 1, [2]). opr(5, +, 5, 1, [0]).  :  opr(9, *, 3, 1, [2]). opr(10, *, 3, 1, [2]). opr(11, *, 3, 2, [0])</pre>	<pre>starts_at(a1, 2). starts_at(b1, 2). starts_at(a2, 2). starts_at(b2, 2). starts_at(a3, 2)..  :  starts_at(1, 0). starts_at(2, 0). starts_at(3, 0). starts_at(4, 1). starts_at(5, 1).  :  reads_from(7, [a7, b7]). reads_from(8, [a8, b8]). reads_from(a, [9, 10]). reads_from(b, [11, a]). reads_from(c, ['ch.b', 12]). reads_from(d, [c, 13]). reads_from(e, ['ch.d', 14]).</pre>
---	--

Figure 6.6 Partial schedule for the FIR filter.

### 6.4.2 Pre-assignment or not?

It is possible to pre-assign operations to a resource, if desired, by specifying the resource as a singular one. For instance, if three adders are required in a schedule, then to pre-assign some operations to one of these, it would have to have a name different from the other two adders, for example “pre\_assgnd\_adder”, instead of just “adder”.

It is not possible, however, to assign any further operations to these resources, once synthesis has commenced, since they would then have to support two, distinct operation types - pre-assigned and normal - and ALUs are not handled by the system.

Also, the pre-assignment of operations severely constrains the synthesis heuristics employed, often giving sub-optimal results, and since the assignment plan in MC<sup>2</sup> is, in fact, a *direct side-effect of memory synthesis*, any tinkering with pre-assignment can cause severe down-grading of results. Pre-assignment is a useful feature, though, when it comes to defining ROM access operations in the schedule, operations whose constant values will probably be pre-grouped to specific ROMs. It is also useful for testing assignment strategies produced by other work [51].

### 6.4.3 The Three Steps to Heaven

The possible complexity of the synthesis process, and limited memory space, demand a partitioned programming approach. The first stage is to take the schedule data-base, and construct the memory and communications structure around the computational base. Then the second stage extracts control and virtual address sequences from the structure, and the final stage finds the *actual* control and address sequences, as (sets of) bit sequences.

#### 6.4.3.1 Memory and communications synthesis

A major premise in this memory synthesis method is that dual port memories are to be used wherever possible, and that, due to a single-phase clocking scheme, no memory element may be both read-from and written-to in the same control step. Because of this, the first stage in memory synthesis is to find any operations (in a cyclic schedule) whose output data is not finally read until the same control step as it was written, as for operation 3 in Figure 6.7a. In each case a dedicated latch is added, to store the data between cstep0 and the datum's final use (Figure 6.7b).

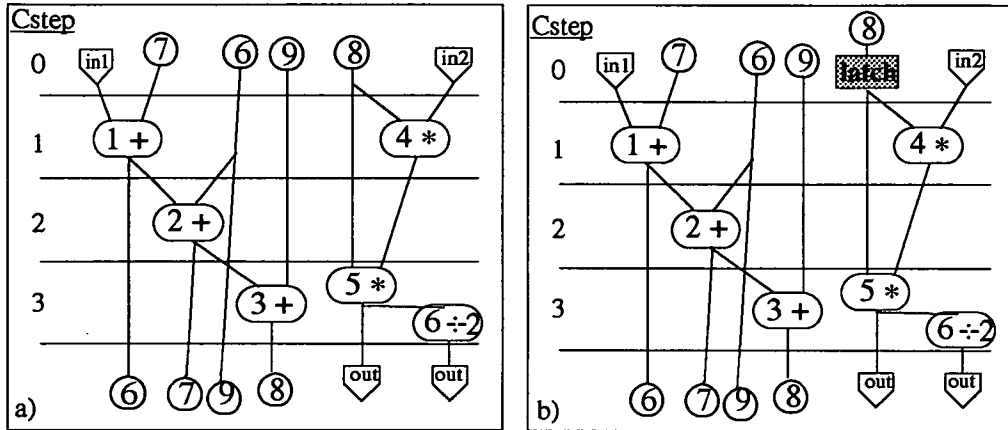


Figure 6.7 a) Simple schedule.  
b) Schedule after latch insertion.

The changes to the schedule database for this transformation are as follows:

**Added:**

```
opr(latchUID,latchUID,1,0,[1,3]).
res(latchUID,1,1,[W],[[1]]).
starts_at(latchUID,0).
reads_from(latchUID,[3]).
```

**Altered:**

```
opr(3,+,1,3,[1,3]). => opr(3,+,1,3,[0]).
reads_from(4,[3,in2]).=>reads_from(4,[latchUID,in2]).
reads_from(5,[3,4]).=>reads_from(5,[latchUID,4]).
```

Any operations which are chained within a single control step, to avoid having a latch inserted at this stage, have the marker "ch." prepended to their UID, which is discarded after this stage.  
For example: `opr(5,*,1,3,[3,4]).` and `reads_from(6,[ch.5]).`

Next, any multicyclic delays which have been declared in the schedule database, are expanded and transformed into strings of latches (shift registers). For instance, operations 2 and 3 would have been declared with:

```
reads_from(2,[1,@.1]).
reads_from(3,[1,@@.1]).
```

In other words, operation 2 reads data from operation 1, and from operation 1 in the previous cycle, while operation 3 reads data from operation 1 again, but from two cycles previously.

The same sort of transformation occurs here as for simple latch insertion, with the delay latch placed in the control step just before the operation requiring that delay, wherever possible. Figure 6.8 shows the schedule from Figure 6.7b, after delay insertion.

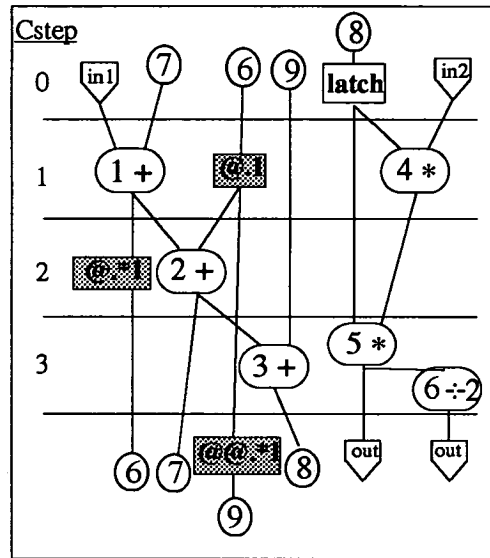


Figure 6.8 Schedule after delay insertion.

Once these steps are completed, memory synthesis-proper begins, by forming groups of operations of the same type, in two stages. If there is only one resource of a certain type, for example the multiplier in the Wave Filter example, then obviously all operations of that type *must* happen on that resource, and are grouped together. Multiple-resource operations, with a choice of resource on which to occur, are also grouped by resource type.

What we hope to achieve is illustrated in Figure 6.9. There should be a separate set of memories written to by each resource, and this allows a control-free bus structure, one for each resource. This may seem constrictive if a least-memory-locations solution is desired, but there was no serious increase in the number of memory elements found for the examples used here (See Section 6.6). We intend to concentrate on reducing the control necessary in the data path, at the expense of a few memory elements.

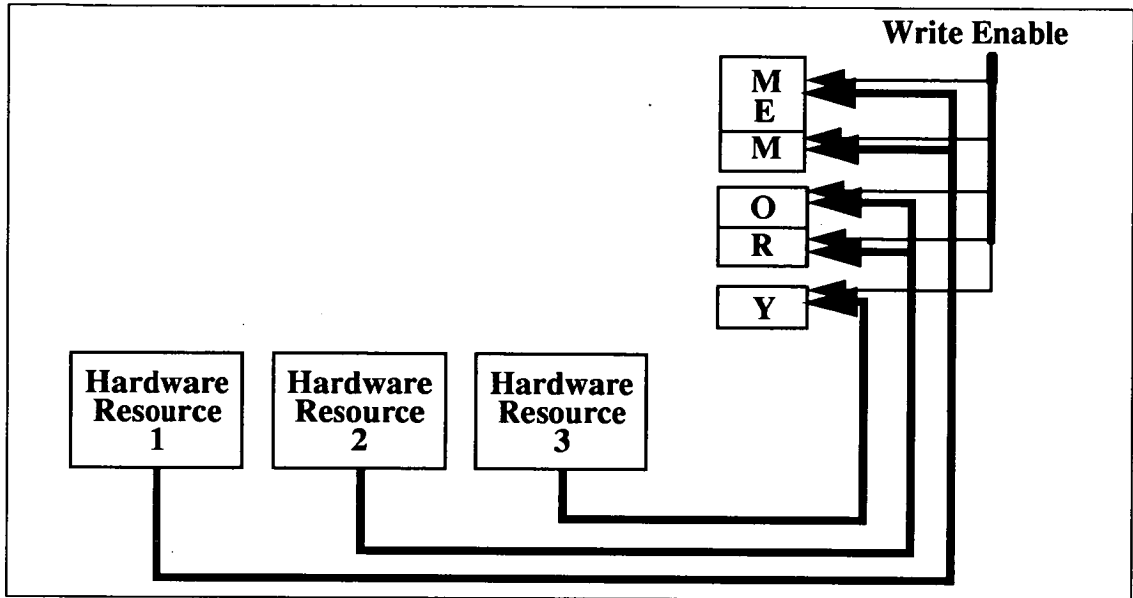


Figure 6.9 Target Write-bus architecture.

Since it is possible that not all data from a single resource may be eventually held in the same, two-port (one Write-only, one Read-only) RAM or register-file, because of access clashes between data, those data, or rather the operations which produce them, must be further grouped “intelligently” so that a near-optimal solution is found. This further grouping allocates multiple-resource operations to specific computational resources, as a side effect of forming the memory structure required to support that allocation.

The “intelligent” approach, using heuristics, was adopted over an exhaustive or iterative search, because of the complexity and accompanying run-times of those other approaches. By pre-grouping the operations by their resource type, we drastically reduce the complexity of the overall problem, and we can use a simple weighting system to implement the second grouping of operations, and their data, to specific memories.

Multiple-resource operations are given a “Write” weight, equivalent to the number of other operations of the same type which require to write data to memory at the same time. Single-resource operations can never have Write access clashes, since only one Write access can ever happen in any control step. The “Read” weight is similarly determined for *all* operations, and the two weights are combined to give a number corresponding to *the degree of difficulty of grouping one operation with others of the*

same type. Figure 6.10 repeats the Wave Filter example schedule, annotated with the weights given to each operation.

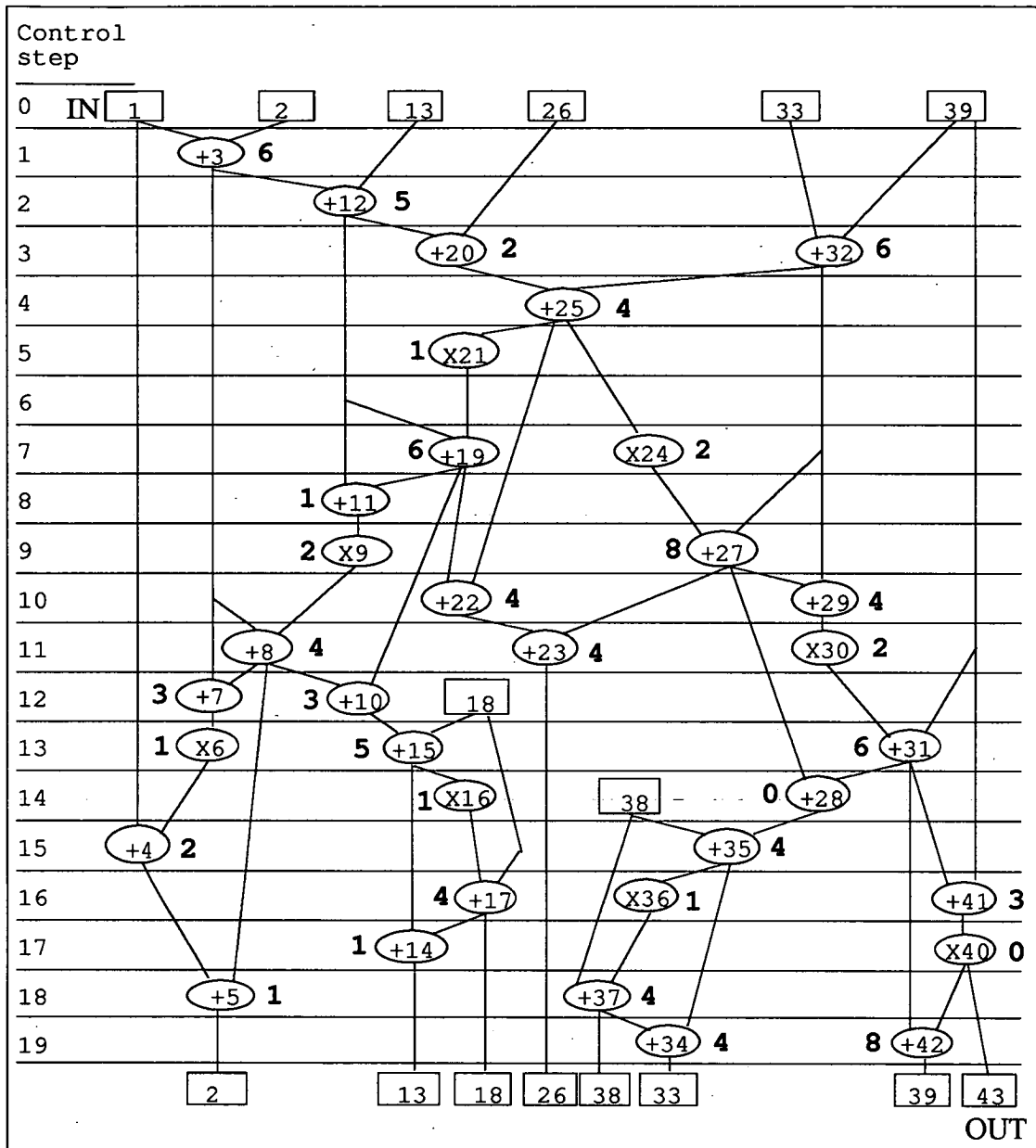


Figure 6.10 a): Wave filter schedule and weightings given to operations.

One simple way to understand the applicability of these weights is to examine an adjacency matrix, or "square graph", of operations (or their associated memory elements). Figure 6.11 shows the square graph of the add operations from Figure 6.10, with an 'r', a 'w' or an 'rw' denoting an access clash (or two) between two data. The



object of the exercise is then to reorder the data on the axes so that *the fewest possible squares* may be drawn on the diagonal, containing no clashes, and covering all data. This reordering should be done using the weights gathered previously.

Two very simple reordering schemes involve sorting the operations in each group into an increasing or decreasing list, by weight. The operation with the lowest weight is the “easiest” to group with other operations, and if we start with the easiest first, and work through the sorted group towards the hardest, we will usually end up with one large group of data, and several small groups. For all examples, the other, hardest-first approach produced the least number of groups (memory blocks), which were more balanced in size. It should be noted, however, that the size of each group, or memory block, bears little relation to the actual number of memory *locations* required to store all the data.

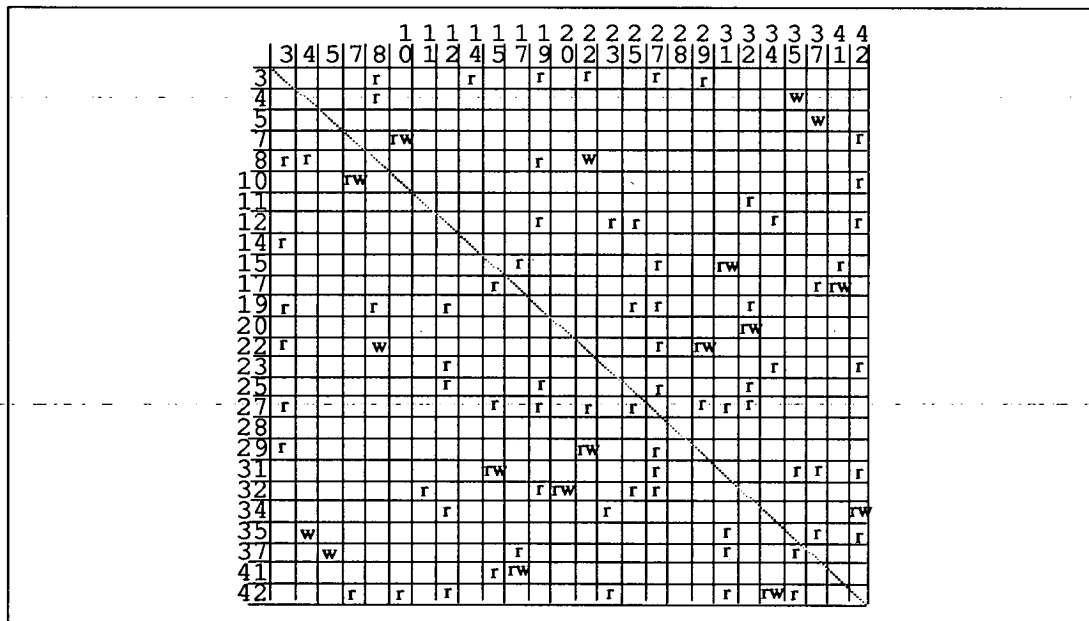


Figure 6.11 Original square graph of add operations in Wave Filter Example.

Figure 6.12 shows the square graph with operations reordered into ascending order of weight, with arbitrary groupings shown. The hardest-first approach is used to further reorder them, forming the square graph in Figure 6.13, which represents four memory blocks.

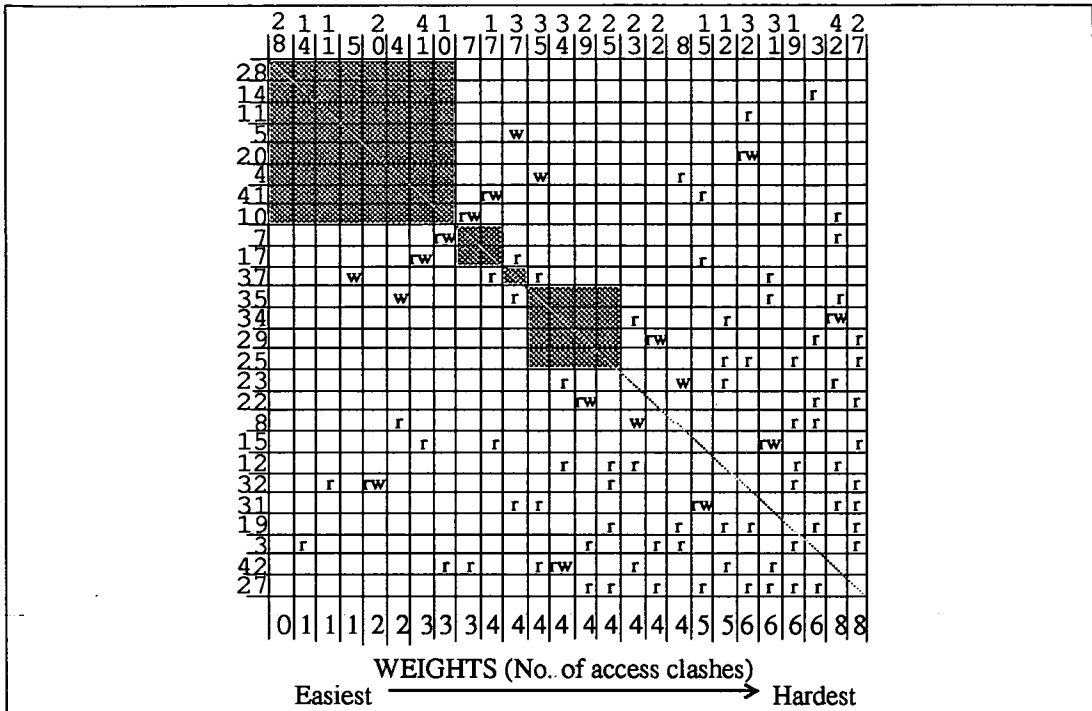


Figure 6.12 Add operations sorted on square graph axes.

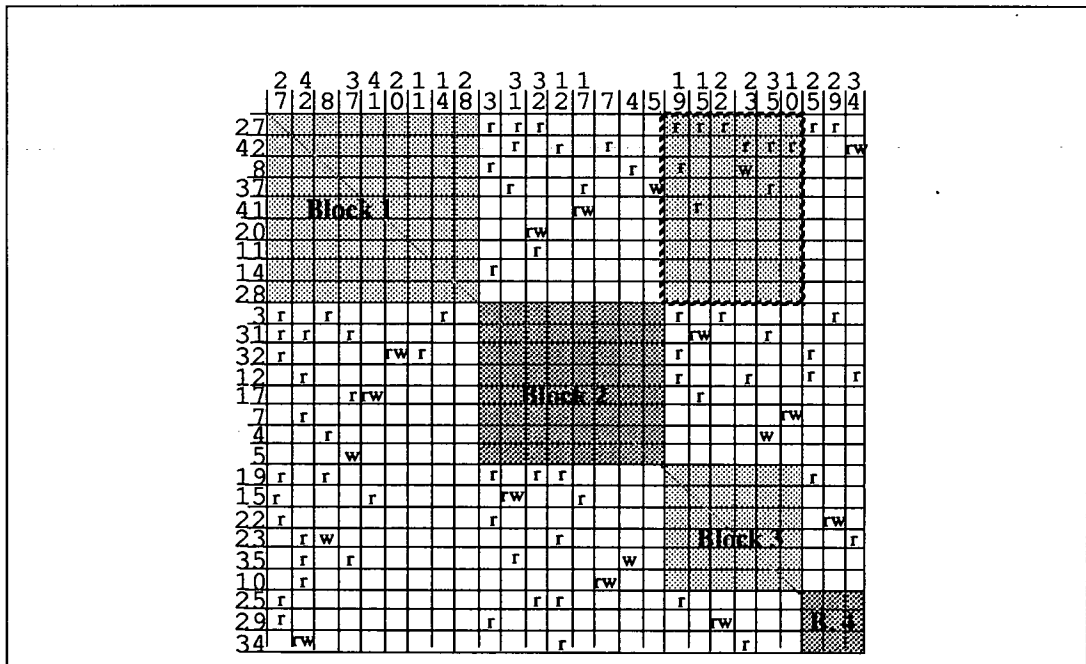


Figure 6.13 Add operations finally grouped into memory blocks.

The final touch to memory synthesis is to make sure that each memory block may be bound to a single computational resource. If the data elements in two different

memory blocks of the same resource-type are Written at the same time (for example operations 8 and 23), then those memory blocks, and their associated operations, may not be bound to the same resource. The square graph in Figure 6.14 shows only the Write access clashes between data in the example, and this information may be used in the same way as before, to group together memory blocks. *There must be as many groups of compatible memory blocks as there are resources of the associated type.* The resulting Write-bus architecture for the example is given in Figure 6.15.

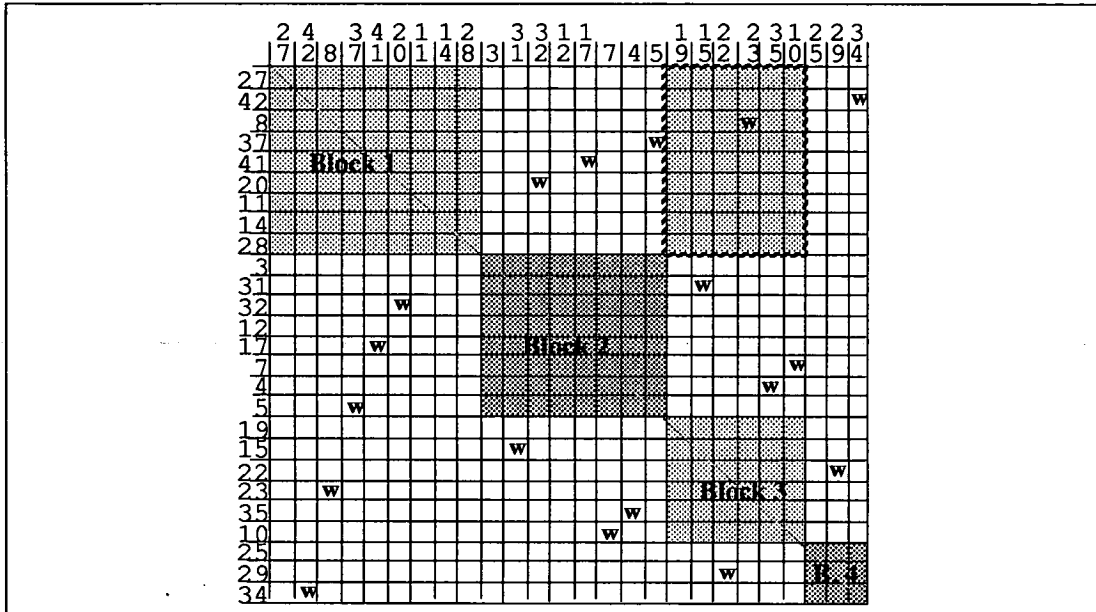


Figure 6.14 Square graph showing Write access clashes only.

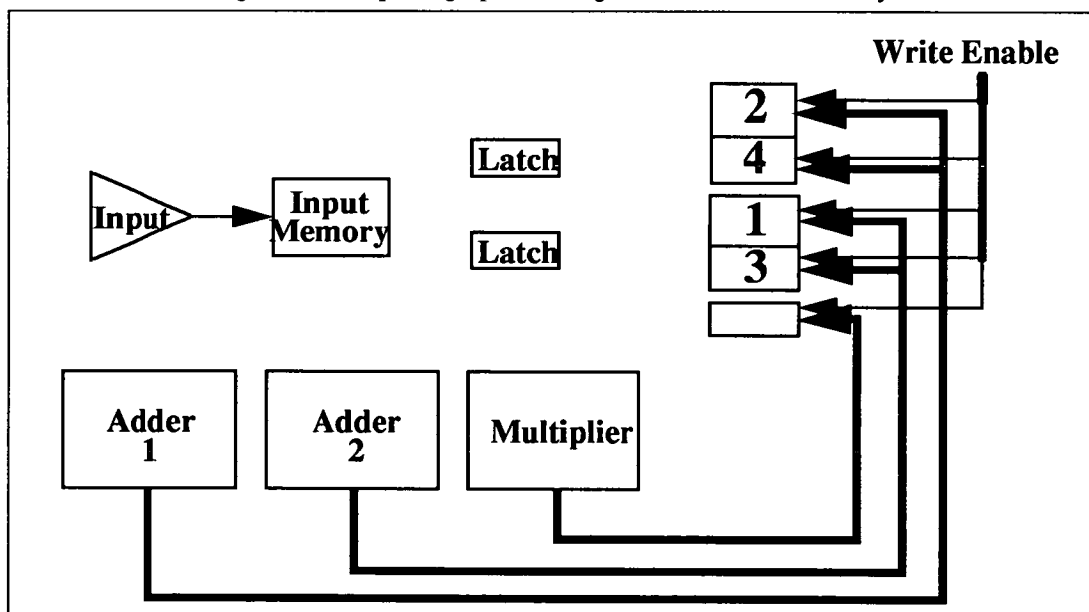


Figure 6.15 Resulting Write-bus architecture for the Wave Filter Example.

If the target number of groups of memory blocks is not reached, then a routine iteratively searches for that target, by moving the causal operations to other, compatible memory blocks, and avoiding local loops in the iteration. The two least-clashing memory blocks (1 and 3, here) are searched for the “culprit” operations (8 and 23), and the one which is easiest to move to another memory block (operation 8), determined with more heuristics, is moved there (to memory block 4). If the target number of groups of blocks is still unattainable, then a different Write access clash is moved, until the target can be reached.

Now we move on to communications synthesis. Since the Write-bus network is already defined, “all” that remains to be done is the construction of the network of multiplexers between the memories and the computational resources, completing the classical, Von Neumann architecture (Figure 6.16).

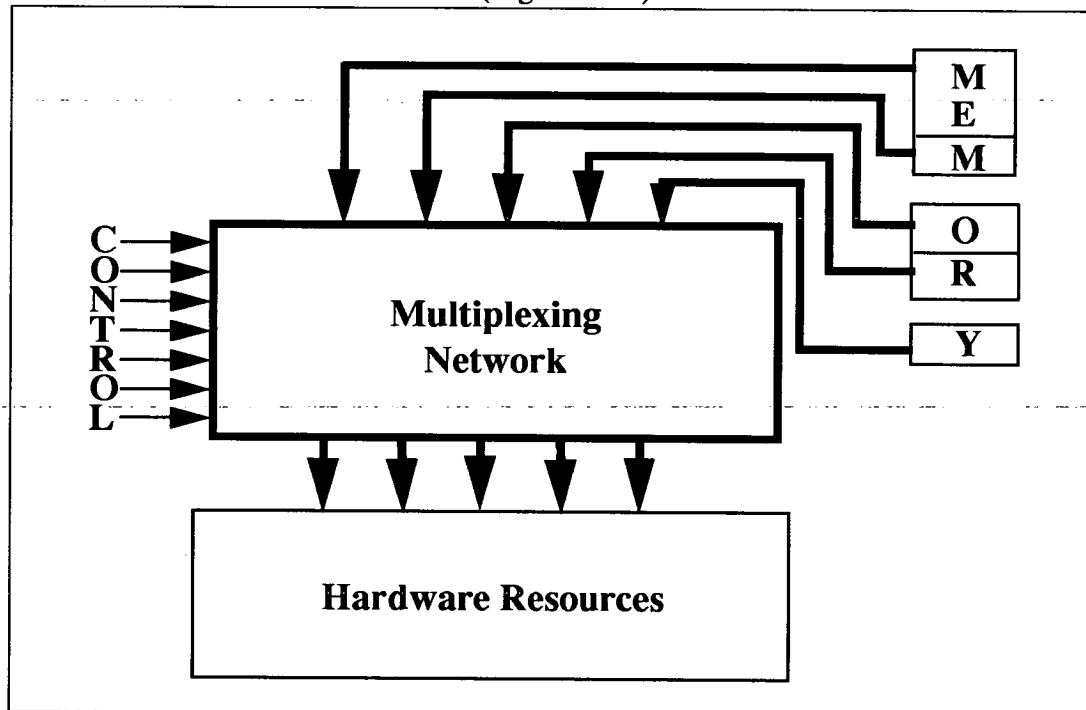


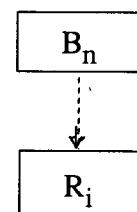
Figure 6.16 Target Read-bus architecture.

Control step:	Adder1		Adder2		Mult1	
	Port A	B	A	B	A	B
1			IPM1	AM1		
2			AM3	AM1		
3	AM3	AM1	AM2	AM4		
4	AM1	AM2				
5					AM1	
6						
7	AM3	MM1			AM1	
8	AM3	AM4				
9	MM1	AM2			AM1	
10	AM4	AM1	AM5	AM2		
11	AM1	AM5	AM3	MM1	AM2	
12	AM3	AM2	AM2	AM4		
13	MM1	AM4	AM2	AM3	AM1	
14	AM5	AM1			AM2	
15	IPM1	MM1	AM3	AM1		
16	AM1	AM4	MM1	AM3	AM2	
17	AM2	AM3			AM1	
18	AM1	AM2	AM3	MM1		
19	AM1	MM1	AM3	AM2		

Figure 6.17 Read-access table for Wave Filter Example.

This process commences with the creation of a table of Read accesses, derived from the “coloured” square graph, shown in Figure 6.17 for the Easiest-first memory configuration. Next, starting at a control step defined by the user (usually cstep 0), and working forwards in time (going back to the start of the schedule if necessary), until all control steps have been examined for each resource, “paths” are created between memory blocks and resource inputs using the following criteria:

We wish memory block  $B_n$  to be available on input  $i$  of resource  $R$ ,  $R_i$ , in cstep  $C$ , with  $I(i,j)$  meaning that inputs  $i$  and  $j$  are interchangeable on  $R$ :



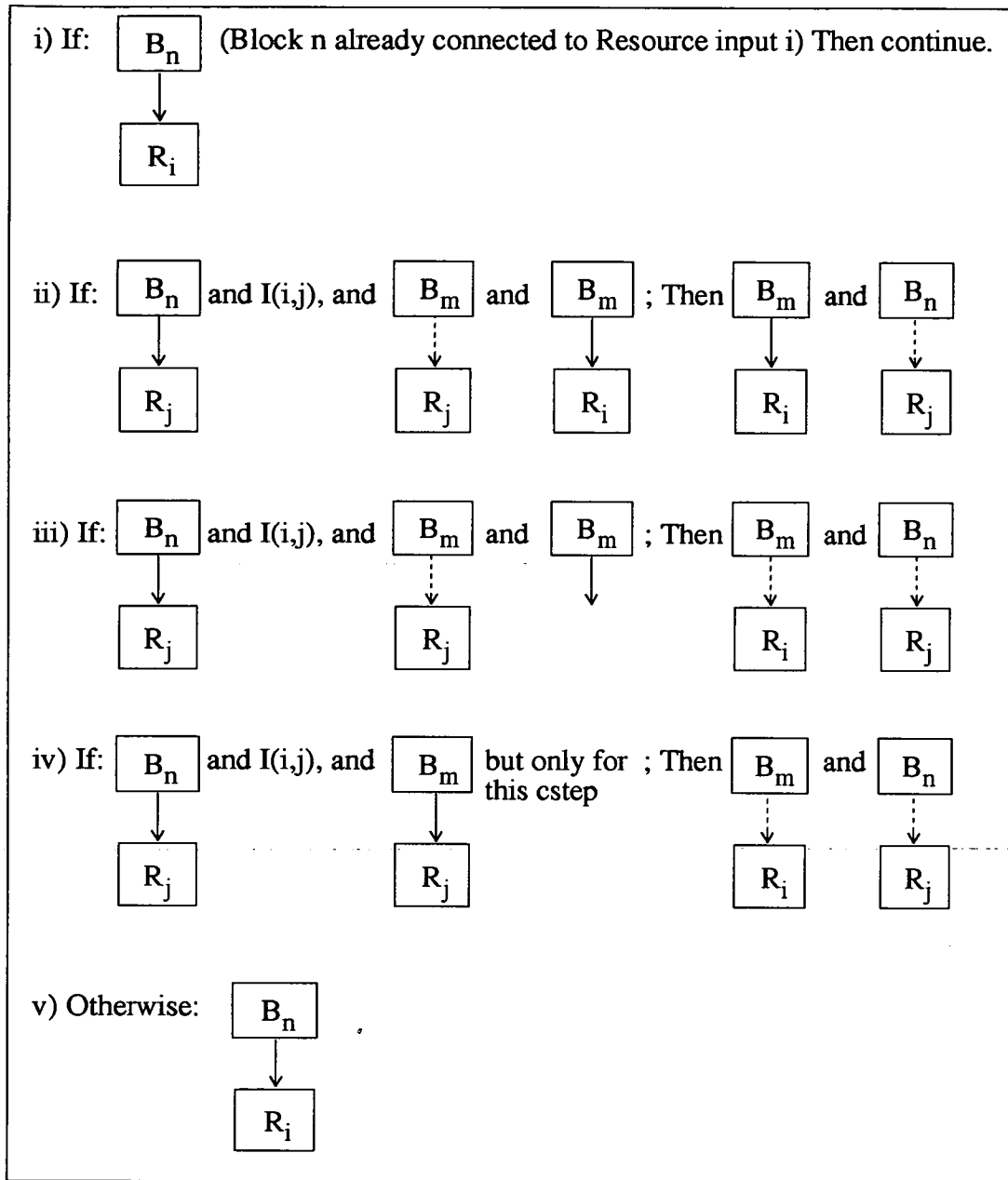


Figure 6.18 Path making criteria for construction of Read-bus network.

In (i) above, a path has already been created between the memory block and the resource input, and the present control step is added to a list associated with that path, holding the times when that path is needed.

In (ii), a path has already been created to an interchangeable input on the resource. This interchangeability is specified in the definition of the resource, as lists of

interchangeable inputs. For example, inputs 1 and 2 of the adders are interchangeable. Here there is also a path already existing from the block which was to have been connected to the interchangeable input, to the input originally (and arbitrarily) intended for the block in question. The blocks are swapped on the inputs, and we get criteria (i) for the other block, and eventually for the block in question.

In (iii), a path to an interchangeable input exists for the block in question, but no path yet exists for the other proposed block at all. The blocks are swapped on the inputs, and path-making continues for the same input.

For case (iv), there already exists a path from the other block to the other input, but which has only been created in this cstep. Here the newly created path is erased, the blocks are swapped, and we continue with the same input.

In (v), none of the above criteria have been found, and a new path is created from the block to the input, whose creation is noted for the duration of the path-making for this cstep.

This path-making continues for all the inputs of each resource, until we have the minimum number of paths from memory blocks to each resource input, which can then be rationalised into a network of multiplexers.

Now we create an optimal communications network between memory blocks and resources, consisting of buses and 2to1 muxes.

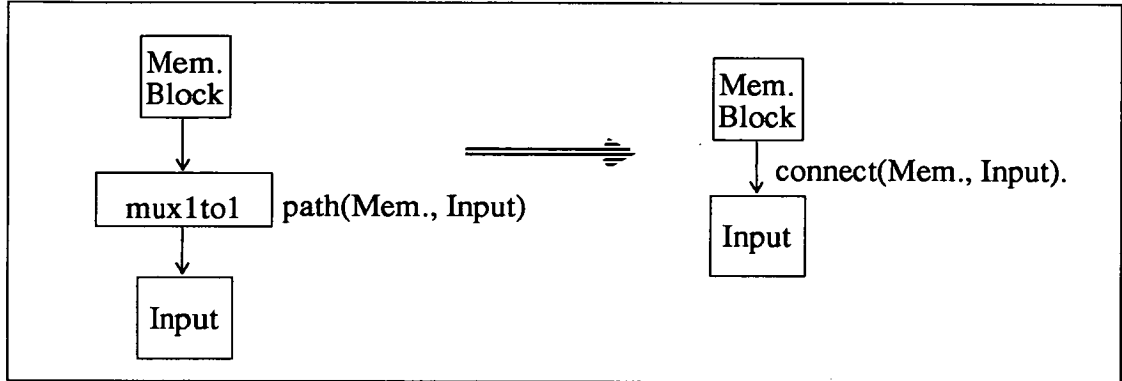
The process starts with the creation of a muxnto1 for each resource input, where  $n$  is the number of blocks which must be connected to that input. This number will have been minimised during path-making. If  $n=0$  (There is no memory block connected to the input), then the mux0to1 is erased. Along with each muxnto1 exists an associated list of csteps during which the mux is needed.

Using the path information, the muxnto1s are broken down into 2to1muxes. From this point an "input" is either to a resource or to a 2to1mux, which has inputs '1' and '2', and output '3'.

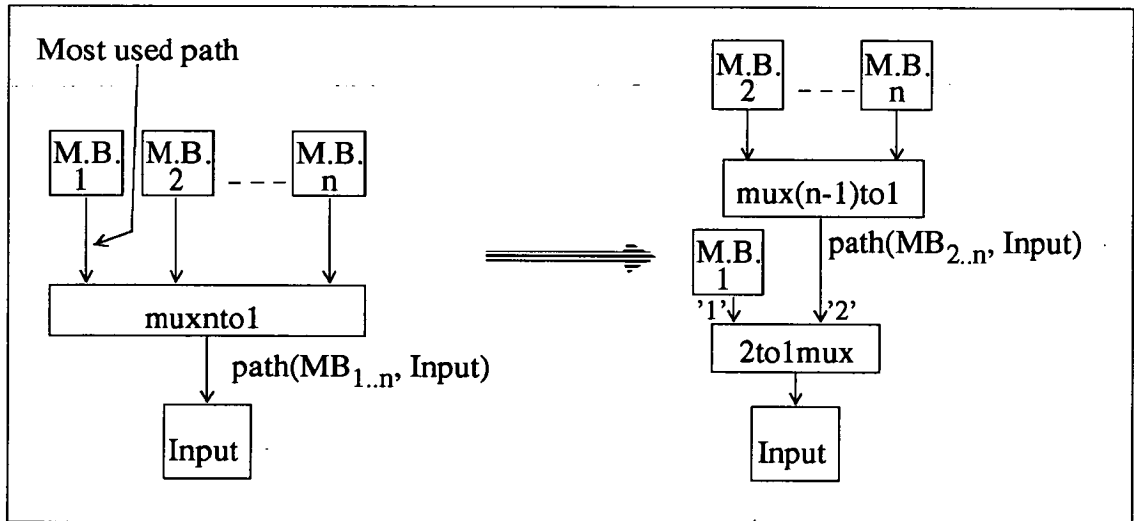
First we find any inputs which must receive data from, at least the same memory blocks, the data being identical whenever the blocks are Read in the same cstep. All but one of the common muxnto1s, and their associated paths, are erased, the one left being that with the greatest value of  $n$ . The lists of csteps when data is required are updated with any different csteps from the erased muxnto1s and paths, and a connection is made from the output of the remaining muxnto1 to the inputs whose muxnto1s were erased.

Next we find the most-used path and extract a 2to1mux from its associated muxnto1, if necessary:

a) If  $n = 1$ , then connect the memory block where the path starts, to the input, where the path ends, erasing the path and the mux1to1:



b) If  $n > 1$  then connect the memory block whose path it is, to input '1' of a new 2to1mux, and connect the output of that 2to1mux to the input expecting the data. The muxnto1 is erased, and a mux(n-1)to1 substituted, with its list of csteps updated so that it no longer includes the csteps when the chosen path was needed. The path is erased and all other paths associated with the mux(n-1)to1 are redirected to lead to input '2' of the new 2to1mux:



By choosing the most used path first, we increase the amount of “Don’t care” values in the control sequences for the 2to1muxes, and so increase the chance of being able to fold the control sequences at a later stage (Section 6.4.3.2).

Finally, any new mux1to1s are erased, along with their associated paths, and the blocks are connected to the inputs expecting the data. The whole process then iterates,



until there are no muxnto1s or paths remaining. The Read-bus network is described as a set of connections, from either memory blocks or 2to1mux outputs, to either resource inputs or 2to1mux inputs.

The resulting Read-bus network for the Wave Filter example is shown in Figure 6.19, which includes the Write-bus networks from before.

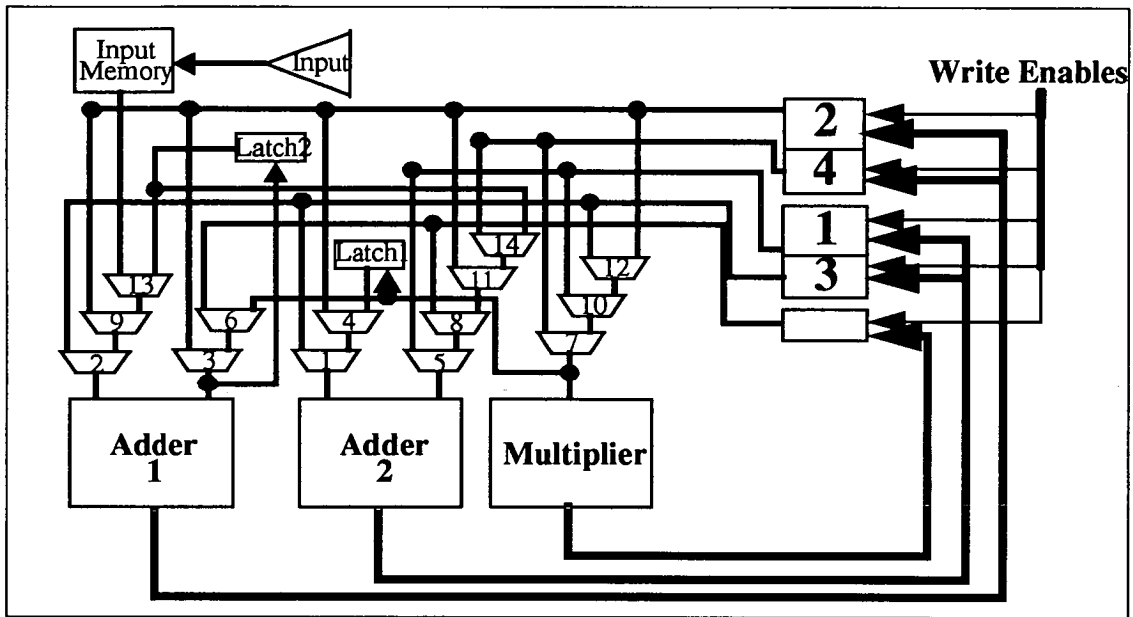


Figure 6.19 Resulting communications network for Wave Filter example.

#### 6.4.3.2 Address and control requirement analysis

The next step is to analyse and decide on virtual address and control bit sequences for the memories and multiplexers.

It is a simple matter to construct a virtual address sequence for each port of each memory, where the virtual addresses are just the UIDs of operations whose output data are being stored or accessed. Alongside the virtual address sequence construction, a boolean “Don’t Care” sequence is also built to denote the actual addressing needs, and also the lifetimes of each data item are noted. Any Write Enable control sequences are

also produced at this stage if required. A sample of the information gathered for the Wave Filter example is shown below.

```
wal('*1', [0,0,0,0,0,0,21,0,24,0,9,0,30,0,6,16,0,36,40,0]) .
ral('*1', [0,0,0,0,0,0,0,21,0,24,0,9,0,30,0,6,16,0,36,40]) .
wal('+1', [0,0,0,20,0,0,0,0,11,27,0,0,0,0,28,0,41,14,37,42]) .
wal('+2', [0,3,12,32,0,0,0,0,0,0,0,0,7,31,0,4,17,0,5,0]) .
ral('+1', [37,0,14,42,20,0,0,0,0,11,27,27,0,42,27,28,42,41,0,37]) .
ral('+2', [17,5,3,12,32,0,0,12,12,32,32,3,3,7,31,0,31,17,4,31]) .
dc_seq('*1', r, [0,0,0,0,0,0,0,1,0,1,0,1,0,1,0,1,1,0,1,1]) .
dc_seq('+1', r, [1,0,1,1,1,0,0,0,0,1,1,1,0,1,1,1,1,1,0,1]) .
dc_seq('+2', r, [1,1,1,1,1,0,0,1,1,1,1,1,1,1,1,1,0,1,1,1]) .
latch_control([latch11,latch21,in1], [save,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]) .
```

Figure 6.20 Information produced during address and control requirement analysis.

Latch control sequences, in the form of “save” operations (technology independent) within a sequence of control steps are also easily derived. It is not so simple, however, to extract the control sequences required for what may be a large number of multiplexers (say < 100). Knowing that memory M must be connected to the computational resource input, R<sub>i</sub>, during control step C, we can trace the path from R<sub>i</sub> back through the multiplexing network to M, noting the input desired on each multiplexer traversed. The control values of any muxes not needed in a control step are “Don’t care” values, which are taken to be ‘0’, whereas the muxes which *are* needed can have control value ‘1’ or ‘2’. This representation is used because the *real* control values, ‘0’ and ‘1’, have not yet been assigned, and may be bound to either of the virtual values, at a later date. Examples of multiplexer and Write Enable control bit sequences are given in Figure 6.21.

```
csig1([mux1], [0,1,1,2,1,0,0,0,0,0,1,1,2,1,2,1,1,0,2,2])
csig2([mux2], [0,0,0,2,0,0,0,1,2,1,2,2,2,1,0,1,2,2,1,1])
csig5([we,+4], [1,1,1,1,2,1,1,1,1,1,2,2,1,1,1,1,1,1,1,2])
csig10([mux10], [0,2,0,0,0,0,0,0,0,0,0,2,1,0,0,2,2,0,1,1])
csig12([mux12], [0,2,0,0,0,0,0,0,0,0,0,1,0,0,0,2,1,0,0,0])
```

Figure 6.21 Example multiplexer virtual control bit sequences.

In the controller, a PLA-FSM or COUNTER-ROM method may be used to generate the control bit sequences for the multiplexers, and the address sequences for the memories. Setting the address generation aside for the moment, we must find some way

of folding the control bit sequences together, so that many multiplexers may share the same bit sequence, reducing the controller's PLA or ROM area.

There are three distinct ways to fold the bit sequences together, each of which is tried, in order, on pairs of control sequences.

### **Overlapping fold.**

This is where the two sequences have common values '1' or '2'.

E.g.: a)                    0 0 1 2 2 1 1 0 1 2  
                                      \* \* \* \*                    \*

and b)                    2 0 1 2 2 1 0 0 0 2

have a "positive" overlap of 5 bits (marked by \*s).

These sequences would fold into:

c)                    2 0 1 2 2 1 1 0 1 2

We can easily swap '1's and '2's, simply by swapping the inputs to the muxes, so

d)                    1 1 2 1 1 2 2 2 2 1

would become:

e)                    2 2 1 2 2 1 1 1 1 2

which sequence (c) would fold into, with positive overlap of 8.

### **Non-overlapping fold.**

This is where the control bit sequences have no overlapping '1's or '2's, but do have overlapping '0's.

E.g.: a)                    2 1 0 0 0 1 0 2 1  
                                      \*                    \*

and b)                    0 0 0 1 0 0 1 0 0

have a "negative" overlap of 2.

These sequences fold into:

c)                    2 1 0 1 0 1 1 2 1

### **Shifted fold.**

Depending on the length of the control bit sequences, it may be useful to introduce delays on some bit sequences, so that they may be generated by other sequences.

a)                    a b c d e f g h ...

b)                    0 0 ... a b c d e ..

If some sequence starts with n '0's (b), and then continues with a sequence already existing, but earlier in time (a), then we can introduce a delay of up to n csteps on the control line of the first sequence. Sequences whose first bit is non-zero, i.e.: "Do care", cannot be generated using a delay, unless the first bit can be preset on the delay - a situation too dependent on other factors to be explored further here.

E.g.: a)                    2 1 0 2 1 2 0 0 1  
           b)                    0 0 2 1 0 2 1 2 2

Introducing a delay of 2 on (b) to get (b''), we get:

a)                    2 1 0 2 1 2 0 0 1  
                          \* \*       \* \* \*  
           b'')                2 1 0 2 1 2 2 0 0

which have a positive overlap of 5, and can be folded into:

c)                    2 1 0 2 1 2 2 0 1 (= (a) & (b'')).

The maximum number of delays it is viable to introduce depends on the length of the control sequence, and the respective areas of control and delay.

### **No possible fold.**

If the sequences follow none of the above patterns, then their overlap is null, and they will never be folded together.

### **Order of selection**

The selection of pairs of control sequences for possible folding is not haphazard.

The control sequences are ordered by the number of '0's in each, so that the "busiest" sequence - that with the fewest '0's - is examined first for possible folding. From the remaining sequences, the one with the largest overlap with that sequence is chosen as a partner for the first. If no overlap exists with any of the remaining sequences, then the sequence with the next fewest '0's is tried.

Once a possible fold is found, the sequences are merged, overlaps are again calculated, and folding attempted once more, until no more folding can occur. Finally, if the sequences are long enough to merit it, shifted folding is attempted, commencing with the sequence starting with the most '0's.

Figure 6.22 contains the original and folded control bit sequences for each multiplexer.

Of an original sequence count of 15 for the Easiest-first memory configuration, only 7 sequences were necessary, and from 12 sequences for the Hardest-first configuration, only 6 sequences were actually needed.

Shifted folding was not attempted, since the sequences were not considered long enough to use a delay instead of separate control bit.

In another example, a set of 40 multiplexer control sequences of length 14 bits was reduced to just 8 sequences, again without trying shifted folding.

**Original control sequences:** (0 -> *Don't Care*, 1&2 -> *Logic '0' or '1'*)

controlsig1 -> [mux1] = [0,1,1,2,1,0,0,0,0,1,1,2,1,2,1,1,0,2,2]

controlsig2 -> [mux2] = [0,0,0,2,0,0,0,1,2,1,2,2,2,1,0,1,2,2,1,1]

controlsig3 -> [mux3] = [0,2,1,1,1,0,0,0,0,1,2,2,2,1,1,2,0,2,1]

controlsig4 -> [mux4] = [0,0,0,0,0,1,0,1,0,2,0,1,0,2,1,0,1,2,0,0]

controlsig5 -> [mux5] = [0,0,0,1,0,0,0,1,1,1,2,2,1,2,0,2,2,1,2,2]

controlsig6 -> [mux6] = [0,0,0,1,0,0,0,0,1,0,1,1,2,0,0,0,1,2,0,0]

controlsig7 -> [mux7] = [0,0,0,1,0,0,0,0,0,0,0,0,1,0,2,0,0,0,1,1]

controlsig8 -> [mux8] = [0,0,0,0,0,0,0,0,0,0,1,2,0,1,0,2,1,0,1,2]

controlsig9 -> [mux9] = [0,2,0,0,0,0,0,0,0,0,1,2,2,1,0,2,2,0,2,2]

controlsig10 -> [mux10] = [0,2,0,0,0,0,0,0,0,0,0,2,1,0,0,2,2,0,1,1]

controlsig11 -> [mux11] = [0,0,0,0,0,0,0,0,0,1,0,0,0,2,0,0,0,1,0,0]

controlsig12 -> [mux12] = [0,2,0,0,0,0,0,0,0,0,1,0,0,0,2,1,0,0,0]

**Folded control sequences:**

Some of the mux inputs may have been swapped, inverting their control bits.

controlsig13 -> [mux1,mux10] = [0,1,1,2,1,0,0,0,0,1,1,2,1,2,1,1,0,2,2]

controlsig14 -> [mux2,mux12] = [0,1,0,2,0,0,0,1,2,1,2,2,2,1,0,1,2,2,1,1]

controlsig15 -> [mux3,mux8] = [0,2,1,1,1,0,0,0,0,1,2,2,2,1,1,2,0,2,1]

controlsig16 -> [mux5,mux11] = [0,0,0,1,0,0,0,1,1,1,2,2,1,2,0,2,2,1,2,2]

controlsig17 -> [mux7,mux9] = [0,1,0,1,0,0,0,0,0,2,1,1,2,2,1,1,0,1,1]

controlsig18 -> [mux4,mux6] = [0,0,0,1,0,1,0,1,1,2,1,1,2,2,1,0,1,2,0,0]

Figure 6.22 Control sequences for wave filter example before and after folding.

### 6.4.3.3 Address and control sequence synthesis

As far as the control bit sequences are concerned, all that remains to be done is to fill any remaining Don't Care times in a sequence with '1's and '2's so that each sequence is made as *regular* as possible, and then to bind these virtual control values to actual values, '0' and '1'. This also defines which input to each multiplexer is selected by which control value.

The replacement of the Don't Care values can be quite complex, but the method is based on taking as short a sequence as possible from the start of the original sequence, and making it fit repetitively onto the rest of the bit sequence, as described in Figure 6.23.

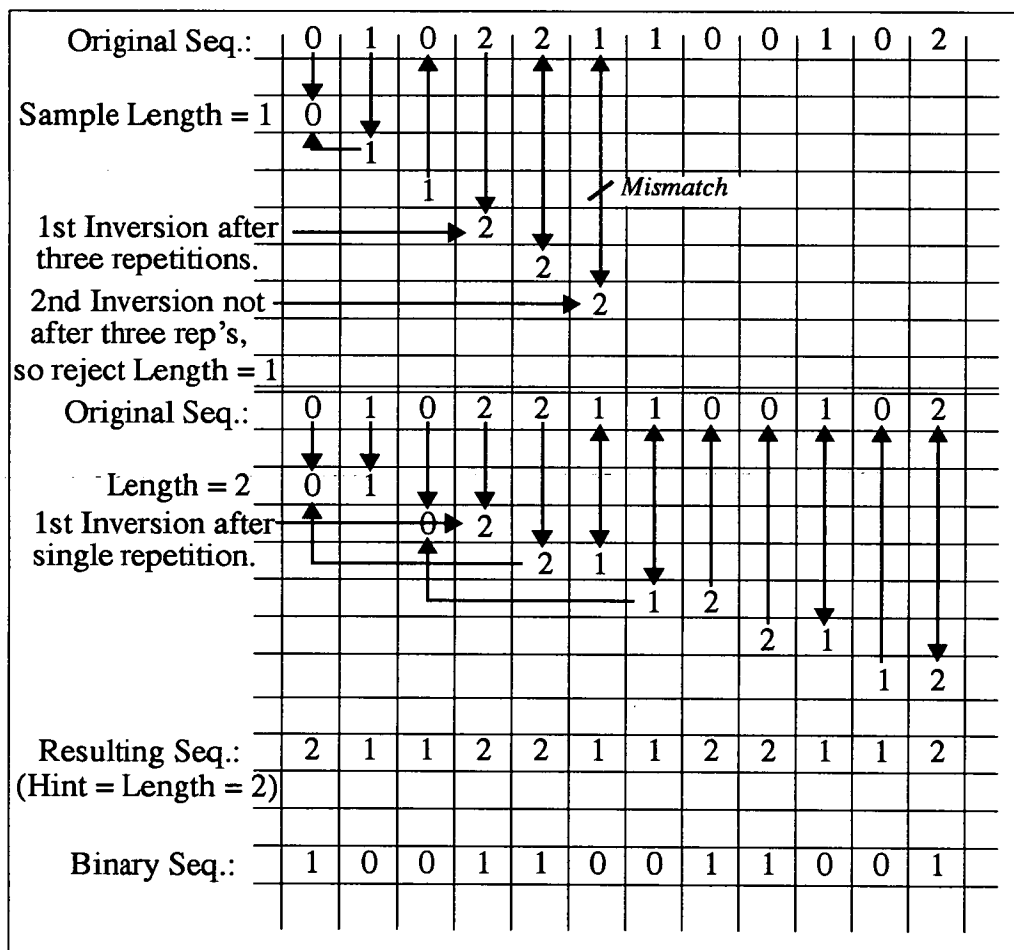


Figure 6.23 An incompletely specified control bit sequence is filled with values to capture inherent regularity.

The virtual values are then converted to real binary values on an arbitrary basis; '1' -> '0' and '2' -> '1'.

The assignment of data items to actual memory locations now commences, which defines the possibly incomplete address sequences, and their constituent bit sequences. Looking at each memory in turn, first the virtual read address sequence is examined. In an interactive mode the number of memory locations available may be set between the minimum number required, discernable from the data-lifetime information, and the maximum feasible with that number of address bits needed to support the minimum. For instance, if at least five memory locations are required, determined from the lifetimes of the data by the Left-Edge algorithm (See Section 5.2.4), then three address bits are needed, which can support up to eight memory locations. The default number of memory locations to use is the minimum required.

For each bit of the address sequence a certain set of values will be possible. An example is given below, where five memory locations exist.

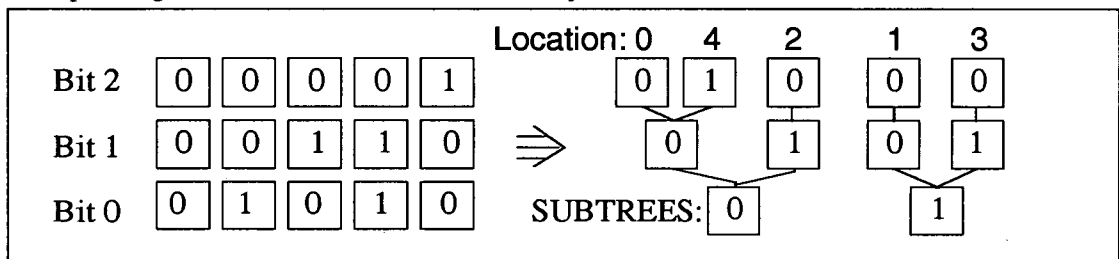


Figure 6.24 Possible address bit values for a five-word memory.

We traverse the virtual address sequence from start to end and build up the lowest significant address bit sequence first, and then all the others, in such a way that the sequences are as regular as possible. This method requires information on the lifetimes of the data, and on which data may share the same memory location. Data are assigned to one or other subtree (Figure 6.24) of possible locations depending on that information, in such a way that there will be enough room for all the data in each subtree, keeping in mind the symmetry of the resulting bit sequence, before the next bit of each real address is examined. Once all data have been assigned to specific memory locations, an algorithm similar to that used on the control bit sequences is then put to work, which takes into account all possible values of each bit of each address to come to an optimal solution. It should be noted that addresses greater than the number of locations available may appear, but only at Don't Care times. Figure 6.25 shows a virtual and the corresponding real address sequence for one of the memories in the wave filter example.

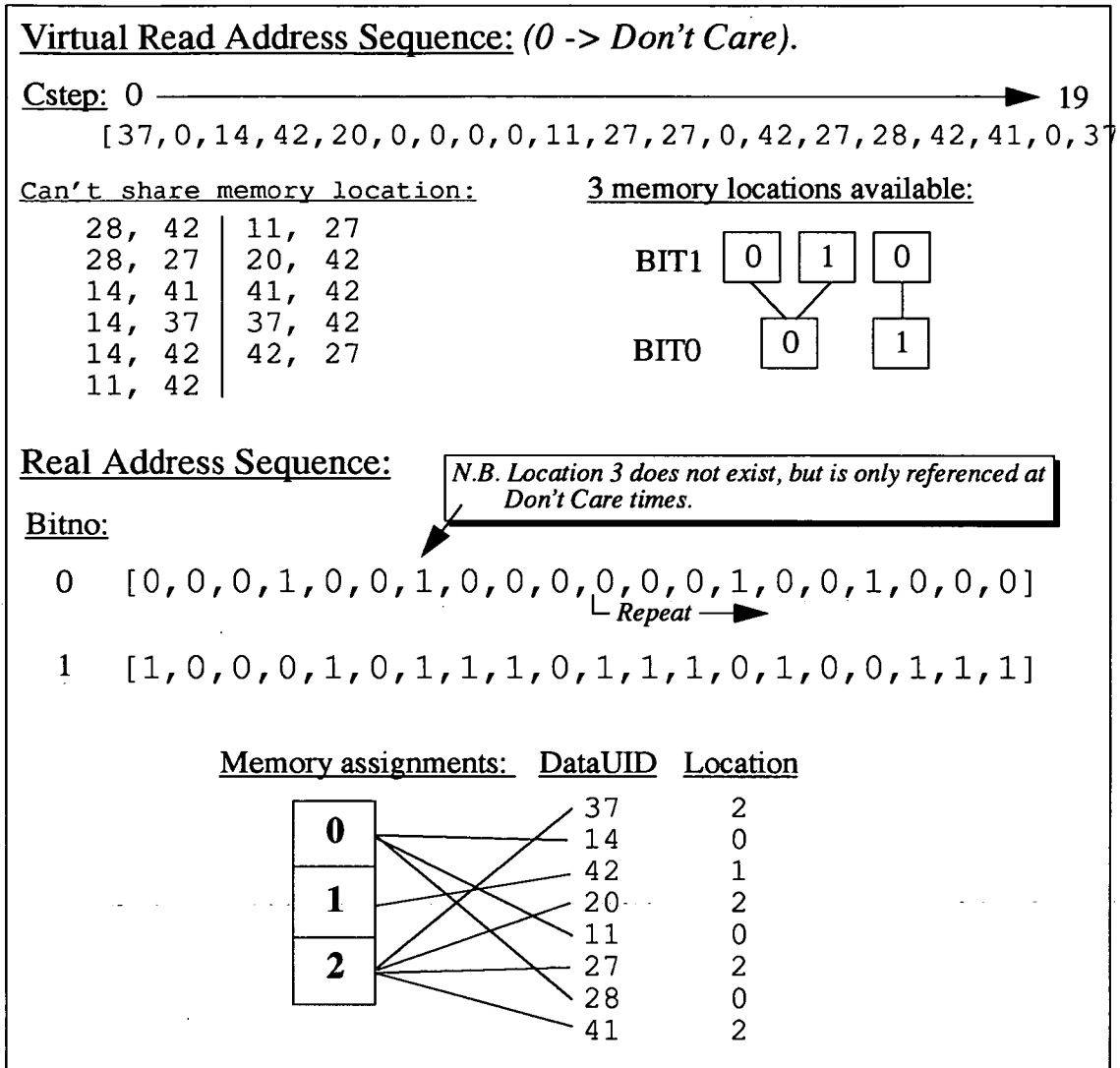


Figure 6.25 Virtual to Real read address sequence conversion.

Since a virtual write address sequence will never be less sparse (have more Don't Care times) than its corresponding read address sequence, the greater degree of freedom this allows relegates the write address sequence to a slightly cruder synthesis technique. The actual locations assigned to each datum are already known from the previous synthesis stage, and it is a simple matter to insert these locations into the write address sequence, where required. The address sequence can then be split bitwise, and each bit sequence can have any inherent regularity retained during the filling of any Don't Care times, as before (Figure 6.26).



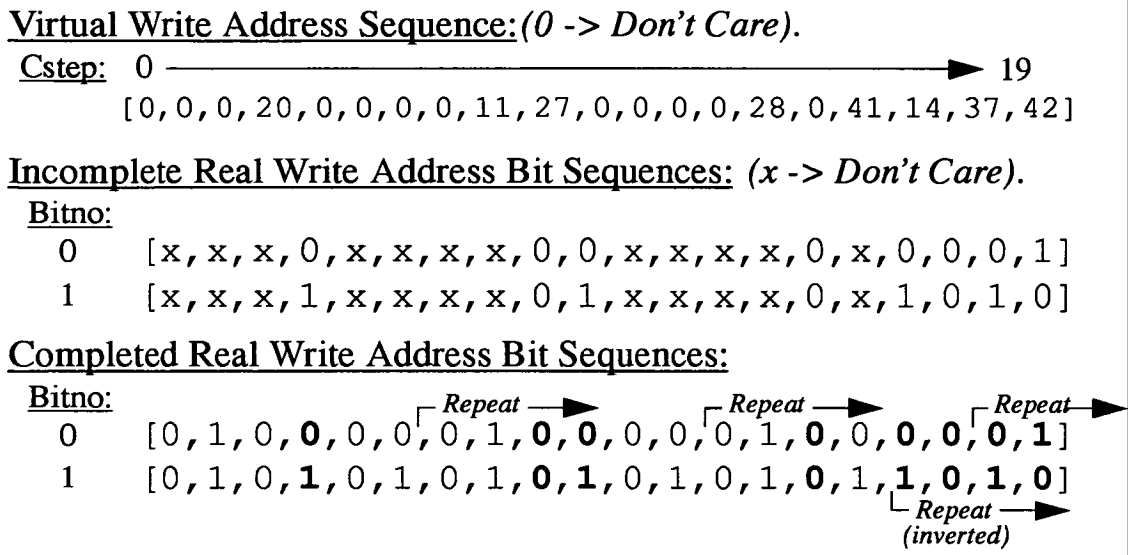


Figure 6.26 Virtual to Real write address conversion.

#### 6.4.4 Output format

The output from MC<sup>2</sup> describes two things:- The structure of the circuit, and the address/control requirements. The former is in the form of a netlist of components, with connections defined between specific ports on the computational and ancillary resources. The circuit synthesised for the FIR filter is described in part by the facts in Figure 6.27. The circuit itself appears in Section 6.5.

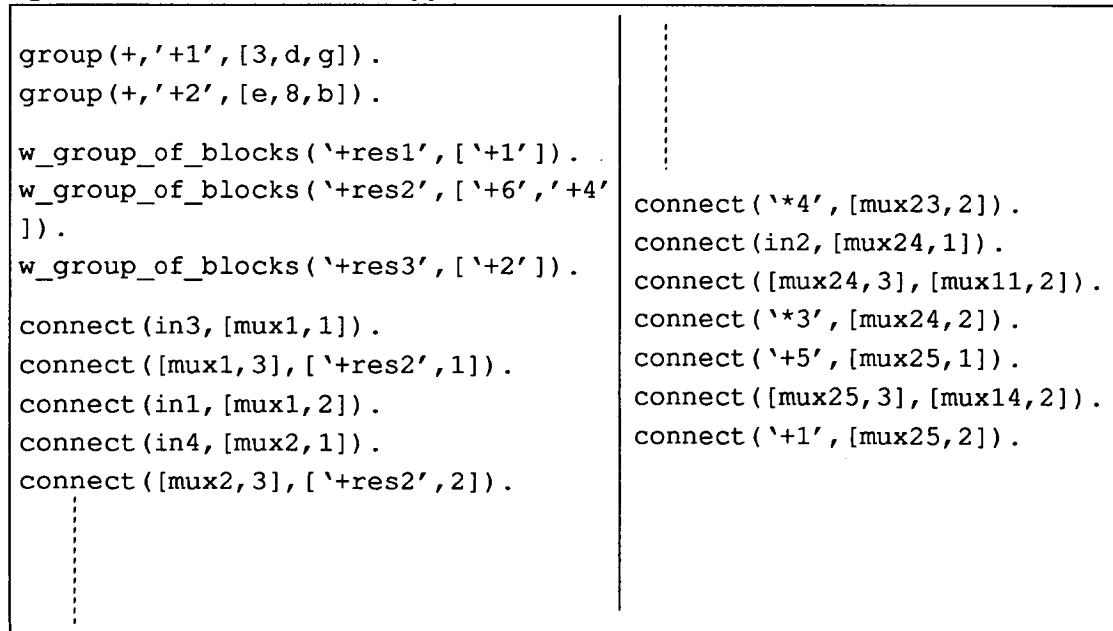


Figure 6.27 Netlist description of FIR circuit.

Multiplexers are defined to have inputs numbered 1 and 2 and an output numbered 3. The inputs will be bound to specific control values at a later date. The other information shown in Figure 6.27 specifies the assignment of operations (e.g.: 3, d, g) to resources (+res1, +res2) and the memory allocation which that introduces (Memories are '+1', '+6', etc. - Adder memory no. 1, Adder memory no. 6).

The second collection of output data specifies the address and control sequences which are required for memories, latches and multiplexers. Memories have their minimum number of memory elements calculated using the Left Edge algorithm, and this information is used by the sequence specification stage. If a memory comprises only a single memory location then the memory will be implemented as a latch, with its own latch control sequence. Address sequences for RAM or register files are described in terms of their constituent bit sequences, with arbitrary control values 1 and 2, to separate virtual values from actual ones. A binary sequence defining which addresses are actually required, for both read and write address ports is also included, if there is at least one Don't Care time.

Latch control signals are defined, technology independently, as a 'save' operation during the correct control steps, and may share the same wire if identical. Write enable signals for memories and multiplexer control signals are grouped together, and so may have several destinations, and the signals are described in terms of those destinations and of course the control values. The address and control information for the FIR filter example is shown in part in Figure 6.28. This information may be simply translated into the format used to input address sequences to AG2, an address generator synthesis tool described in the next chapter.

```
min_no_memels('*1', 3).
min_no_memels('*2', 2).

adbit(w, 0, '*2', [2, 2, 1], [2, [inv, 1]]).
adbit(r, 0, '*2', [1, 1, 2], [2, [inv, 1]]).
adbit(w, 1, '*1', [1, 2, 1], [2, [inv, 0]]).
adbit(w, 0, '*1', [2, 1, 1], [2, [inv, 1]]).
adbit(r, 1, '*1', [1, 1, 2], [2, [inv, 1]]).
adbit(r, 0, '*1', [1, 2, 1], [2, [inv, 0]]).

dc_seq('*1', r, []).
dc_seq('*2', r, [1, 0, 1]).

latch_control(['+6', '*4'], [0, save, 0].
control(csig74, [mux2, [not, [we, '*3'], [not, [mux1]]]], mux25
```

Figure 6.28 Partial address and control specification for FIR filter.

The information following each virtual address and control bit sequence specifies the shortest section from the start of the sequence, which can be used to generate the whole sequence, if repeated, perhaps with inversion in polarity after a number of repetitions. It is this information which allows us to calculate a hint as to the generation of the sequence, to also be handed to AG2. Appendix E contains the address and control generators as synthesised by AG2 for all the examples given here.

## 6.5 Some synthesized data-path architectures

Shown in the next few diagrams are the circuits synthesised by MC<sup>2</sup>. These have been drawn manually, following the netlist information from the system. Firstly, the circuit for the wave filter example is given in Figure 6.29, and Figure 6.30 shows the circuit for the FIR filter. The implementation of the differential equation example is illustrated in Figure 6.31, and finally the FDCT architecture appears in Figure 6.32.

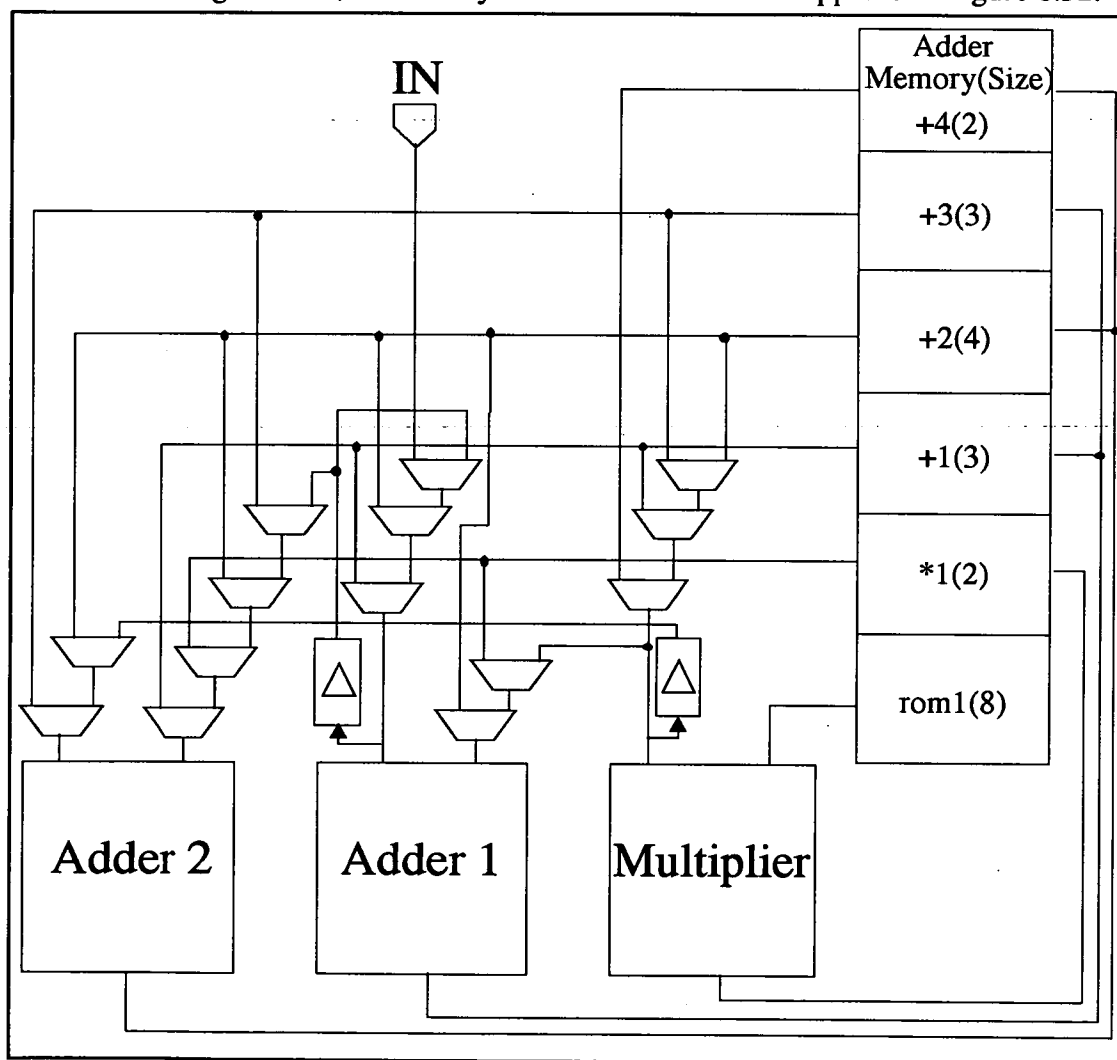


Figure 6.29 Wave Filter circuit.

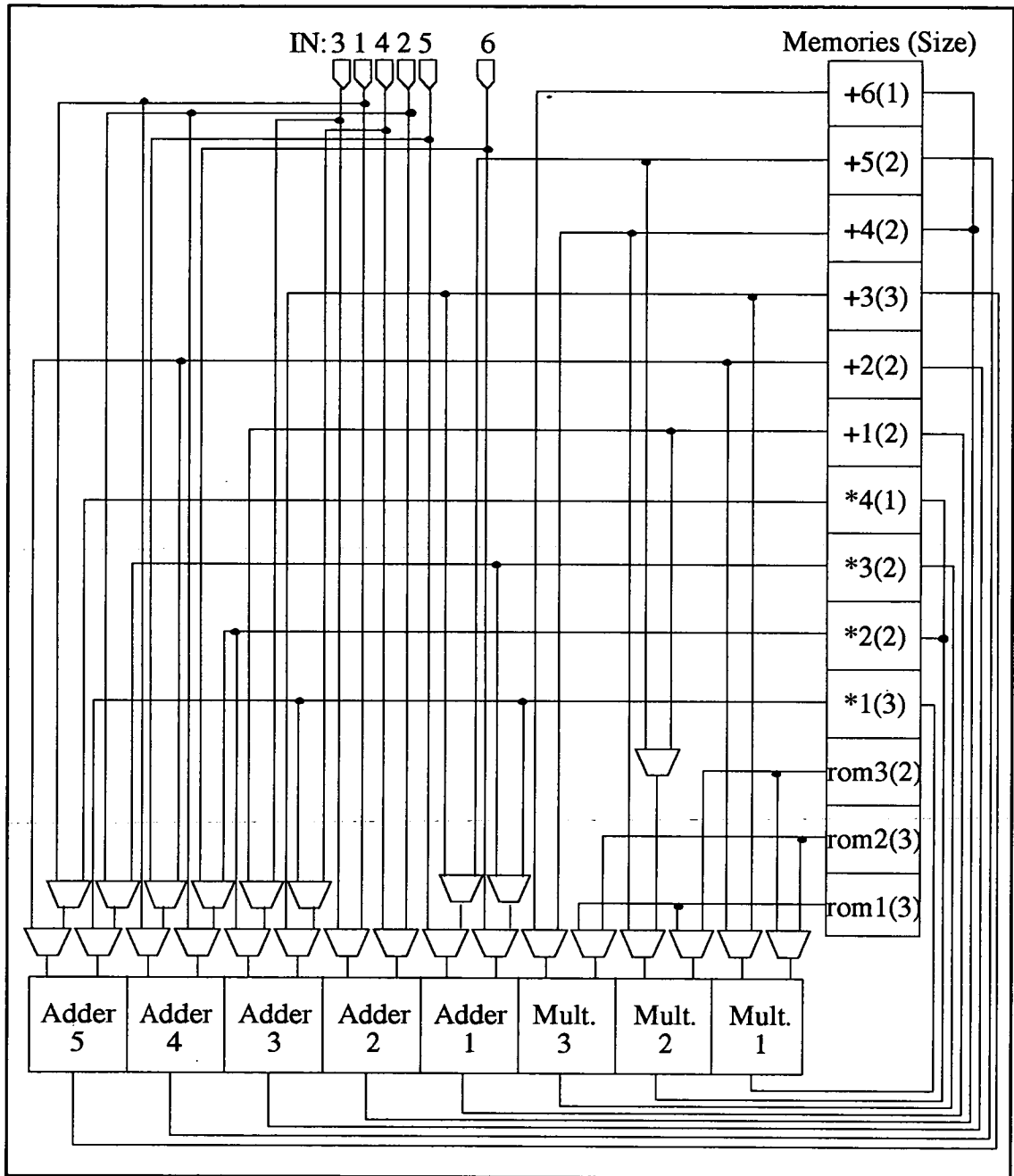


Figure 6.30 FIR Filter implementation.



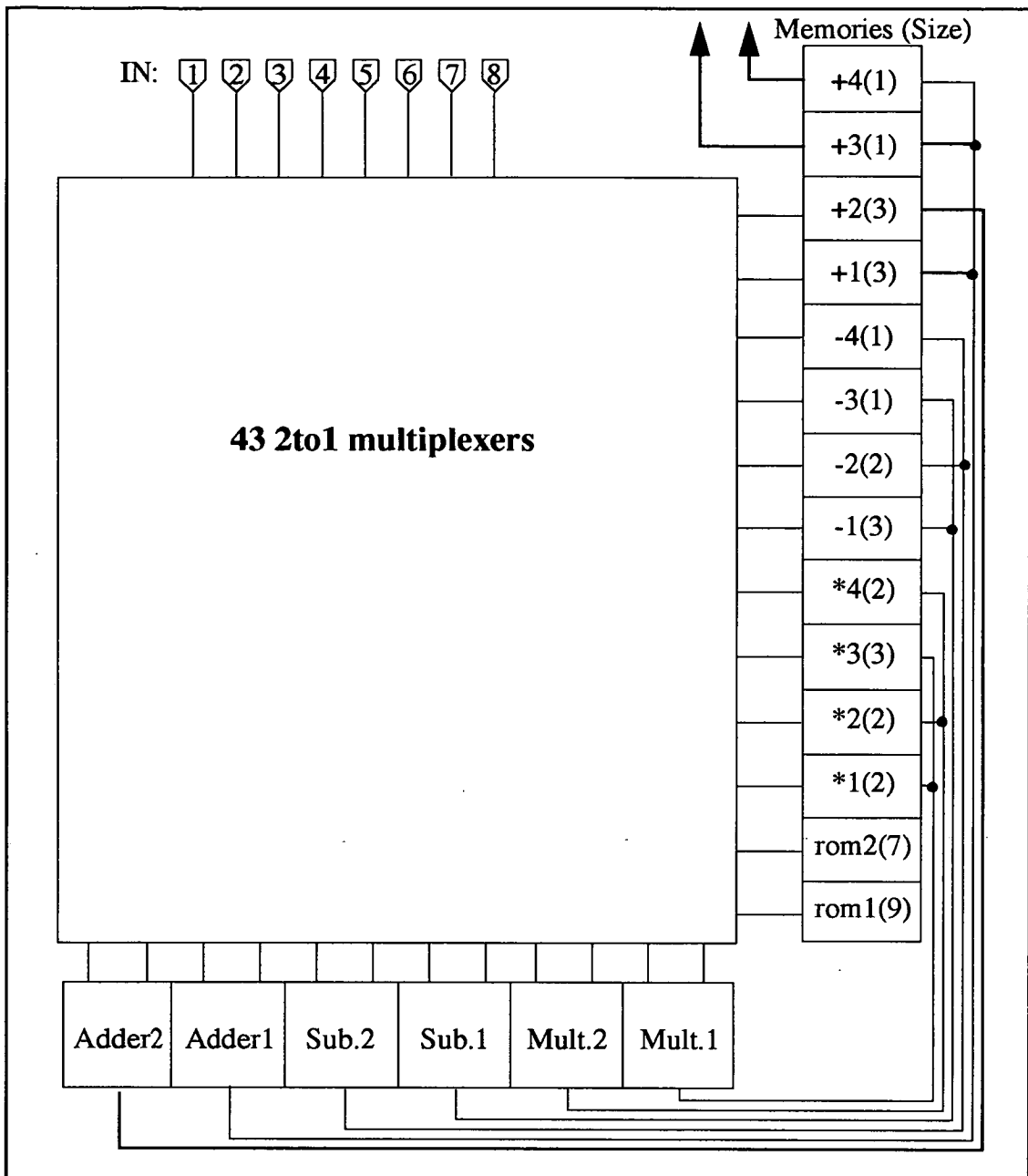


Figure 6.32 FDCT solution.

## 6.6 Comparisons with related results

It is very difficult to find a globally fair comparison between circuits synthesised by different systems, described in the literature [1, 3, 6, 9, 15, 19, 39]. This is due to a lack of standards in the reporting of results in this area, and such a standard is proposed in the following section. Table 6.1 gives some of the results from the literature alongside those from MC<sup>2</sup>. A second table shows the same results normalised to the best result in each column.

Multiplexer trees are taken to be autonomous networks of multiplexers; Multiplexer inputs are all *primary* inputs to muxes (i.e.: not from other muxes); Multiplexer equivalent is simply a count of all 2 to 1 multiplexers in a design - A multiplexer tree with  $n$  primary inputs will need  $(n-1)$  2 to 1 mux equivalents, or less if multi-level optimisation has occurred. The number of local interconnections is given as the total number of wires with only two ends - Point-to-point links - and this may include wires from computational resources to memories, from memory outputs or chip inputs to multiplexer or resource inputs and links within multiplexer trees. To find the number of buses in the designs, all connected wires (nets) with more than two ends are gathered together as a bus, and this should include any wires not labelled as local interconnect. shows the wave filter architecture with buses in grey and the local interconnect in black.

Registers are either RAM locations or single latches. A register file with a single location should be converted to a latch, so that any register file will contain more than one location. Control Bits is a count of all different bit sequences to be applied to the circuit, including latch control, mux control, memory write-enable and address wires. Any control bit sequence may have several destinations.

CPU Time is given as an approximate guide to run-times, but is only important in that all the time given are quite short, considering the complexity of the examples. The use of different hardware on which to run the software precludes any real comparison of speed between systems.

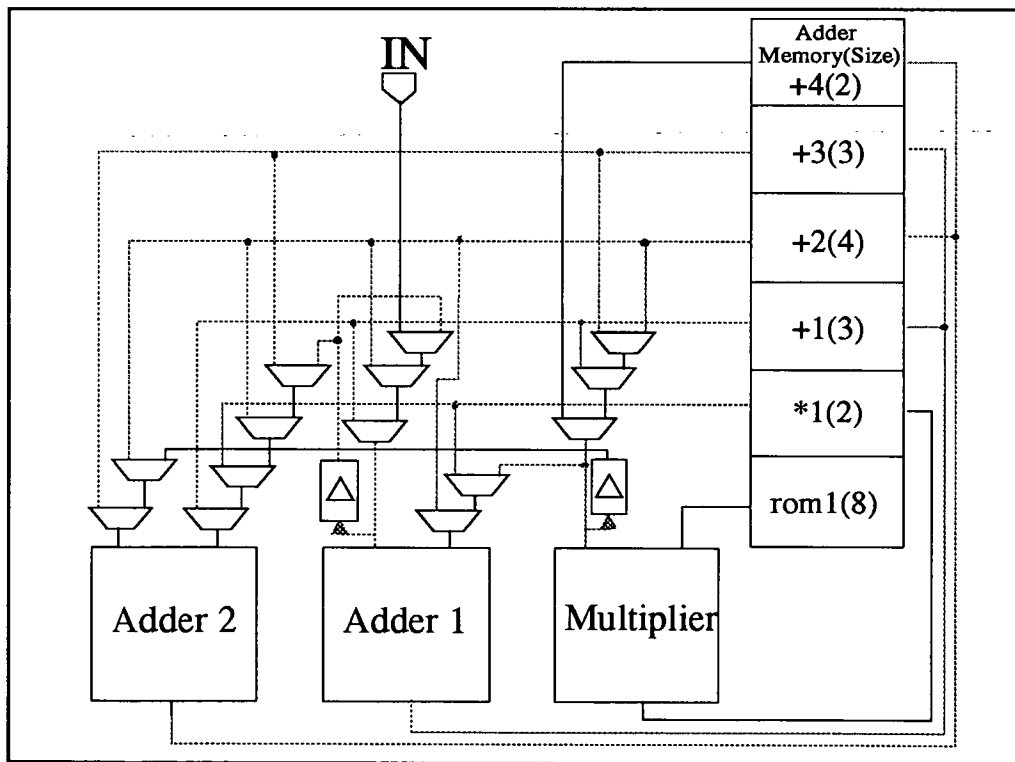


Figure 6.33 Wave filter architecture with buses highlighted in grey, and local interconnect in black.

Systems: A-Easy; B-SPAID; C-Splicer; D-Sehwa; E-SCHALLOC; F-MABAL; I-HAL; J-EMUCS; H-MC<sup>2</sup>.

**Wave Filter:**

System	Cycles	+	*	M.T.	M.I.	M.E.	L.I.	Buses	Regs	R.F.	C.B.	C.T.
F	17, 19	2	1p	13,10	45,32	32,22		-, 2	16,10			<15
A	19	2	1p		25				15	8		
C	21	2	1	9	43							55
I	19	2	1p	6	26	20	45		12			~600
J	19	2	2	14	50	36			12			
B	19, 21	2	1p		17(51)		14,13	5,4	19	5,4		~60
E	21	2	1p	13	57,53	28,27		11,9	13			~150
H	19	2	1p	5	16	14	17	8	16	5	28	~237

**FIR Filter:**

System	Cycles	+	*	M.T.	M.I.	M.E.	L.I.	Buses	Regs	R.F.	C.B.	C.T.
D	3(6)	5	3			23	34x8		18			
B	3(6)	4	3									
I	3(6)	5	3									
H	3(6)	5	3	16	41	25	33	19	20	8	4	563

**Differential Equation:**

Sys.	Cyc.	+	-	*	<	M.T.	M.I.	M.E.	L.I.	Buses	Regs	R.F.	C.B.	C.T.
F	4	1	1	2	1	6-8	13-17	7-9			5-6			<1
I	4	1	1	2	1	6	13-14	7			5			140
C	4	1	1	2	1	5	12	7			6			291
E	11,4	1	1	2	1	3-7	11-16	6-8			6			11-423
H	4	1	1	2	1	6	13	7	12	8	9	3	8	57

**FDCT:**

Sys.	Cyc.	+	-	*	M.T.	M.I.	M.E.	L.I.	Buses	Regs	R.F.	C.B.	C.T.
H	13	2	2	2	11	51	43	41	27	24	8	41	1121

Key: M.T. - Multiplexer Trees.

M.I. - Multiplexer Inputs.

M.E. - Two-to-one Multiplexer Equivalent.

L.I. - Number of local interconnection wires (not busses).

R.F. - Register files.

C.B. - Number of different address/control bits to be generated.

C.T. - Approx. CPU time in seconds.

Table 6.1 Results from literature against those from MC<sup>2</sup>.



Wave Filter:												
System	Cycles	+	*	M.T.	M.I.	M.E.	L.I.	Buses	Regs	R.F.	C.B.	C.T.
F	19	2	1p	2	2	1.5	-	1	1	-	-	1
A	19	2	1p	-	1.6	-	-	-	1.5	1.6	-	-
H	19	2	1p	1	1	1	1	4	1.6	1	28	16
I	19	2	1p	1.2	1.6	1.4	2.6	-	1.2	-	-	40
J	19	2	2	2.8	2.6	2.6	-	-	1.2	-	-	-
B	21	2	1p	-	1.3	-	13	1	1.5	4	-	1.1
E	21	2	1p	1.2	1.3	27	-	2.3	1	-	-	2.7
C	21	2	1	1	1	-	-	-	-	-	-	1

FIR Filter:												
System	Cycles	+	*	M.T.	M.I.	M.E.	L.I.	Buses	Regs	R.F.	C.B.	C.T.
D	3(6)	5	3	-	-	1	1	-	1	-	-	-
B	3(6)	4	3	-	-	-	-	-	-	-	-	-
I	3(6)	5	3	-	-	-	-	-	-	-	-	-
H	3(6)	5	3	16	41	1.1	1	19	1.1	8	4	563

Differential Equation:														
Sys.	Cyc.	+	-	*	=?	M.T.	M.I.	M.E.	L.I.	Buses	Regs	R.F.	C.B.	C.T.
F	4	1	1	2	1	1.2	1.1	1	-	-	1	-	-	1
I	4	1	1	2	1	1.2	1.1	1	-	-	1	-	-	140
C	4	1	1	2	1	1	1	1	-	-	1.2	-	-	291
E	4	1	1	2	1	1.4	1.3	1.1	-	-	1.2	-	-	11
H	4	1	1	2	1	1.2	1.1	1	12	8	1.8	3	8	57

Where no comparison could be made, the original figures have been retained.

Table 6.2 Normalised results.

From these tables we can deduce that MC<sup>2</sup> tends to produce better results for communications and control circuitry at the expense of local memory space. This is an effect of the memory-first synthesis approach taken in MC<sup>2</sup> and is not terribly detrimental to the size of a design, since RAM locations come quite cheaply.

## 6.7 A Standard for behavioural synthesis results presentation

It is always useful to know the number of cycles into which an algorithm has been scheduled, but commonly the given figure does not include any input or output control steps. This is all right because any input or output steps may be wrapped round to the end/start of the schedule, so that the number of cycles scheduled - **Cyc** - should include all computational operations. Whether the schedule is cyclic or acyclic is important, and for pipelined designs, the delay for the first output should also be given, in brackets.

Since we want to be able to compare as many different architectures as possible, all results should be technology independent. In other words, instead of giving the actual area of any computational resources, we should simply present the number and type of such hardware, for example: 2 adders, 3 multipliers, etc.

The number of multiplexer trees and primary inputs are not as important as the number of 2 to 1 multiplexer equivalents, which should be given as the total number in the design -  $N_{2m1}$ . We are also interested in the number of different multiplexer control wires required -  $N_{cm}$ . Other communications information should include the number of buses -  $N_B$  and the number of local wires as defined in the tables above -  $N_{wL}$ .

From the memory side of things, we wish to know the number and average size of any RAMs or register files -  $N_{RAM}$ ,  $S_{RAM}$ ; The number of registers (latches) -  $N_{REG}$  - and also the number of different address wires and latch control wires required -  $N_{cA}$ ,  $N_{cREG}$ .

Finally, the CPU time should be given - **CPU** - as well as a note of the hardware used in producing that run-time.

If any of the above factors are not automatically produced by a system, they must be compiled by hand and reported as such in the results table.

Example	Cyc	$N_{2m1}$	$N_{cm}$	$N_B$	$N_{wL}$	$N_{RAM}$	$S_{RAM}$	$N_{REG}$	$N_{cA}$	$N_{cREG}$	CPU
Wave Filter	19	14	12	8 <sub>h</sub>	17 <sub>h</sub>	5	2.7 <sub>h</sub>	3	11*	1	237
* Including 3 ROM address bits. <span style="float: right;"><sub>h</sub> signifies hand compilation</span>											

Table 6.3 Example of standard result.

## 6.8 Prolog for fast development

Indubitably it was Prolog that allowed such fast development of MC<sup>2</sup>. Having used the language previously, the re-learning curve was steep, and although the code can get rather unintelligible, the ease of handling the data is a boon. Any new facts can be simply added to the data-base in a meaningful format, and old data may be removed just as easily. The data structure may not be hierarchical or coherent, but is sufficient for our purposes and took very little time to develop.

## 6.9 Comments

Even at this point, there are several problems still unaddressed in this area, but sadly these have to remain so, while the over-riding task, that of automating scheduled memory address generator synthesis (and now also, control bit sequence generator synthesis) is completed.

The complexity figures for the algorithms used in MC<sup>2</sup> are almost impossible to compute due to the recursive nature of much of the code. However, Table 6.4 below shows the run-times for the various sections of MC<sup>2</sup> for the examples used, along with some information on the complexity of the original schedules.

CPU time (s)				
Data path, memory and communications synthesis	44	242	17	257
Control and Address sequence extraction and optimisation	72	281	21	748
Control and Address sequence analysis, and final memory synthesis	121	40	19	116
No. of Operations	42	39 ①	11	50 ③
No. of Resources	3	14 ②	5	14 ④
No. of Csteps	19	3	4	13
No. of Multiplexers	14	25	7	43
<div> <div> ① Including 16 input operations.  ② Including 6 input ports.  ③ Including 8 input operations.  ④ Including 8 input ports. </div> <div> Example:  Wave Filter  FIR Filter  Diff. Equation  FDCT </div> </div>				

Table 6.4 Run-times for the three sections of MC<sup>2</sup> for the examples.

From these figures we can determine that a busy schedule with a large resource set slows down data path and memory synthesis, a long schedule effects the control and address sequence analysis, and a large number of multiplexers badly effects the time for control and address sequence extraction and optimisation. Formulae obtained for the run-times of the two latter stages are:

$$\text{CPU}_{\text{STAGE2}} \approx (0.64 * N_2 m_1)^2 \text{ secs.}$$

$$\text{CPU}_{\text{STAGE3}} \approx 5(\sqrt{(\text{Ops.Csteps})} - 2.8) \text{ secs.}$$

MC<sup>2</sup> turns out to be a highly adaptable and functionally complex data-path synthesis system, as well as fulfilling its seemingly simple original purpose.

If a partial target solution to a design problem can be found quickly, then we can use exhaustive techniques to perfect this. It is the compilation of the target architecture which requires all the planning of a chess master, such that the game is almost over after the first move. The chess master will look ahead not to all possible moves in the game, but only to those *probable* in the situation reached. We have the same planning problem in behavioural synthesis, in that every decision made will effect our choice of further decisions, but unlike the chess master, our opponent is neither inscrutable nor unpredictable. We should know exactly what will happen after the very first decision has been made, since our synthesis algorithms tend to be serial and predictable.

Synthesis systems cannot hope to achieve anything near as good as a manual approach while they are based on a single, very general algorithm. If the algorithm is tailored to a specific example, then it will be better at similar problems, but worse than the general approach for different ones. To be as good as manual designers in general **a system must be able to select and alter the algorithms to be applied**. The synthesis is then targeted directly at the individual problem, for best results.

For instance, the time taken to find a solution using each individual algorithm should be known from its complexity order. If a solution is required very quickly, perhaps as an innovative suggestion or trigger, then only the fastest algorithms should be chosen from. If in a chip design problem there are a large number of operations, then we will most likely need to schedule them onto a fewer number of computational resources. If there are a large number of data arcs in the schedule then point to point communications will probably prove to be unworkable, and memory synthesis is very important.

By fine-tuning the available algorithms as best we can, very good solutions should be found quickly, and it is only this lack of interface between the synthesis algorithms and the human designer that inhibits the possible generality of our tools.

## **7 A general approach to address generator synthesis**

### **7.1 The need for generalization**

After the completion of the first part of this work, described in Chapter 4, we accepted that we were still very constrained in terms of the problems targeted. For some digital image processing filters for example, it is known that more efficient designs can result from sampling the image data in blocks not sized by a power of two [207], but by three or five or some other number. Also, much of the control of these processes should also be based on that size of sample, and any FSM built to generate that control would be more efficient when driven by a non-binary counter.

So the decision was made to attempt to construct a more general and complete tool which would seek the same sort of solutions as AG1, but which would do so using an intelligently-chosen counter modulus. The bitwise approach was adopted from AG1 but a much more comprehensive data structure in AG2 allows a whole design's address and control generators to be optimised in terms of their total area. Many other methods of address generation other than using counters are also identified in AG2.

### **7.2 The inevitable data model**

The possible complexity of the situation demands a coherent and comprehensive data structure, in which to store and process all the necessary information. The hierarchical approach taken here is both necessary [87, 93] and natural, with each memory requiring a set of address generators, which are in turn formed from a set of bit sequence generators (See Figure 7.1).

At the top level is a list of the memories required in the design. Each memory has a name and a UID, as well as a possible pointer to an actual hardware component (in a general design environment - See Chapter 8). The type of memory should be known, and if this is given as "control" then then addresses are actually to be generated as control words for use elsewhere. Information on the dimensions of the memory are available, and a list of the names of any arrays (or groups of data) to be stored in the memory, corresponds to the address generators needed to access that data.

An address generator data structure is tied to a single data-array, and as such has well defined time limits on its use, as well as a clock signal, which may be gated by a second, strobe signal. The mode of memory access for which this address generator will

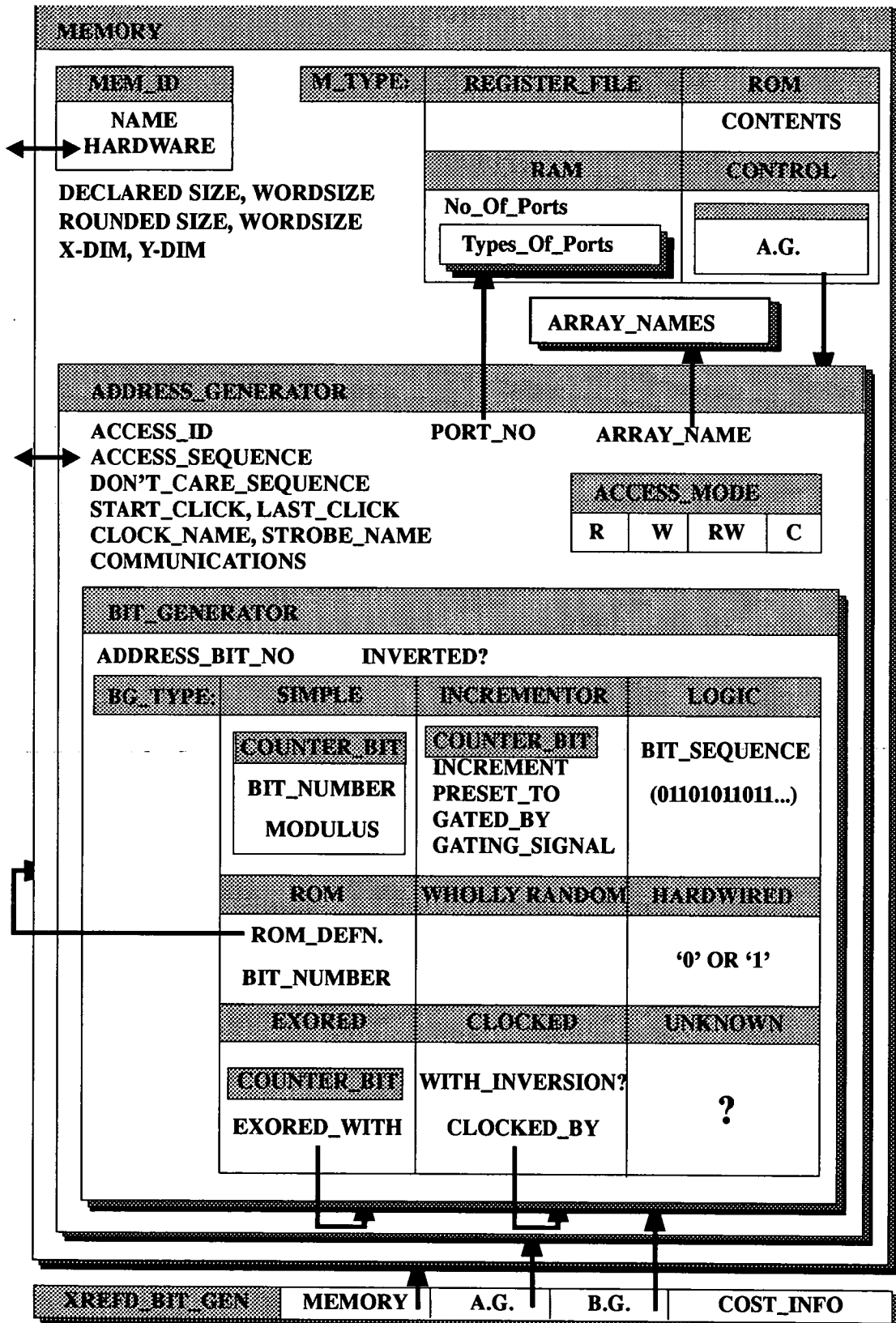


Figure 7.1 Address Generation Data Structure.

be used will be important, as will the port number on the RAM (if a multi-port RAM is involved).

The destination of stored data, suppliers of data to be stored, or in the case of control generation, the destinations of control signals, may be defined, as well as a list of behavioural function calls for addresses, and which of these are *actually* required (There may be dummy calls inserted).

The address generator also comprises a number of bit generators, which may be one of several types:

- I) A Simple bit generator is defined as the  $b$ th bit of a modulus  $m$  counter;
- II) An EXORed bit generator is some bit generator, as above, exclusive ORed with the output of some other bit generator;
- III) A Clocked bit generator is a simple flip-flop, clocked by the (possibly inverted) output of some other bit generator;
- IV) An Incrementor-type bit generator is bit  $b$  of some incrementor, which may be preset, reset and have a gated clock;
- V) A ROM bit generator is some output bit  $b$  from an address ROM, which may be constructed as part of the optimisation process;
- VI) A Logic-type bit generator is the output from some combinatorial logic or PLA, characterised by a binary sequence;
- VII) A Hardwired bit generator is simply a constant logic '0' or '1';
- VIII) A Wholly Random bit generator is one whose required bit sequence is too long and too random to consider here;
- IX) Finally, a bit sequence may have an as-yet-Unknown bit generator.

The outputs from these bit generators together form the address words to be generated, and may be inverted in polarity.

The final element of the data model, to allow bit generator costing and optimisation at a later date, is a cross-referenced list of *all* bit generators, along with space for their costs, further discussed in Section 7.4.5.

Taking a general view of this data structure, it can be seen that a RAM may have several ports of different types, each of which may be utilised to store or access some group of data in some predetermined sequence. Each group of data will have addresses



generated by a set of bit generators, and, most importantly, these may be implemented on hardware shared with other address generators.

A data ROM access sequence may be defined as for any other memory, along with its contents, and this may possibly have addresses generated by another, *address* ROM, which will in turn need addressing, most likely using an incrementor. The “Unknown” bit generator may in the future be expanded to handle other types of bit generator, for instance ring-counters or more complex structures [196, 199], with corresponding extensions to the data structure. It is felt, however, that what we have now is sufficient to handle most situations.

### **7.3 Requirements of an address generator synthesis tool**

As stated, AG2 can perform global optimisation of address and some control generation hardware. This is a major requirement of a synthesis tool since the bias towards the use of counters is not only for their simplicity, but also their re-usability. Other requirements include explicit definition of address sequences for RAMs and ROMs, and the ROM contents if applicable. Deterministic control bit sequences should be handled and any sequence must be able to express whether each address is actually needed or is just a dummy address filling an unnecessary part of the sequence.

A number of options for the generation of each sequence should be considered, and hardware sharing should be encouraged where it is effective in reducing the total area of all the generation circuitry.

### **7.4 AG2 - A general address generator synthesis tool**

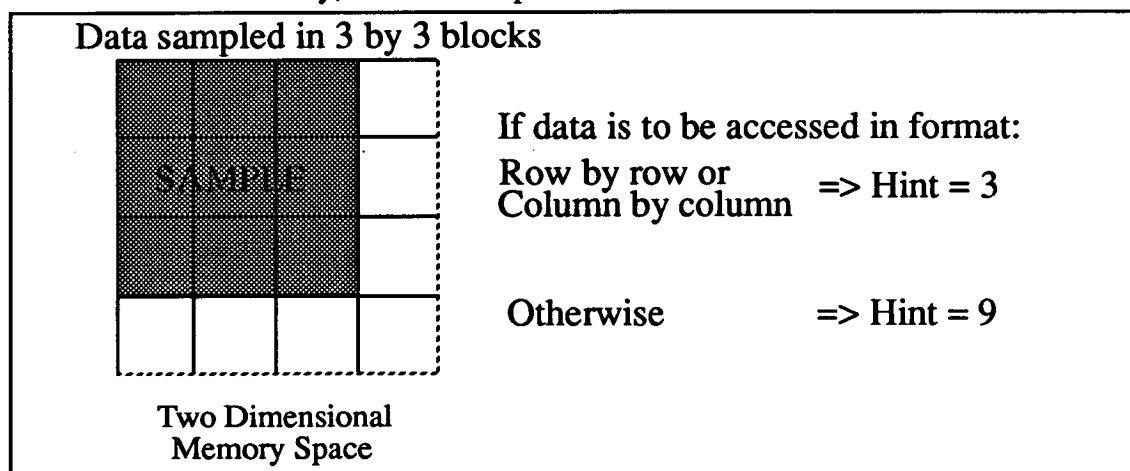
In the following sections we examine the human interfaces to AG2, as well as describing the method used to recognise various types of bit sequence generator, and then investigating the optimisation stage where the bit sequences are bound to their cheapest possible generator. A user guide for AG2 appears in Appendix D and the code is included on the disk.

#### **7.4.1 Input format**

There is a standard, file-based input format for AG2, with the input file(s) containing the following information:

- **Memory Name** - This is derived from the name of the access sequence file itself.
- **Memory Type** - May be ‘ram’, ‘rom’ or ‘control’, in which case the sequence is to be generated for control rather than memory access. The memory name should reflect this.

- **Start Click** - An integer defining the clock tick (click) on which the first address is to be generated.
- **Clock Name** - The clock which will be used to iterate through the sequence (and it is this clock which should be used to determine the Start Click).
- **Strobe Name** - This may simply be 'dummy', but if the sequence is to be generated using a gate on the aforementioned clock, then the strobe name should be defined as the UID of that gating signal. This may be done automatically if necessary.
- **Access Mode** - 'W', 'R', 'WR', 'RW' or 'C'. This defines the mode of memory access for which this sequence will be used. 'C' denotes a control sequence.
- **Communications Name** - This should be a list of all computational resources which read/write data from/to the memory, or a list of all destinations of a control sequence.
- **Hint** - A very important integer which will be used by the synthesis tool as a basis for finding counter-based address generators. Figure 7.2 describes how a designer could find a good hint, which should reflect any repetitive pattern length inherent in the problem. If given as '0', it will be found automatically, but at the expense of time.



*Figure 7.2 Manually defining a hint.*

- **Access Sequence** - This is the list of integers which must be generated, terminated by a '-1'.
- **Don't Care Sequence** - A binary sequence with the same length as the Access Sequence, again terminated by a '-1'. Where the Don't Care Sequence is '0', the corresponding address in the Access Sequence is not actually required. If a Don't Care Sequence would be all '1's (every address is required), then it may be omitted, apart from the '-1' terminator.

- ROM Contents - The list of constants to be fixed in the ROM, in order of ROM address, terminated with a '-1'. This data is only included if the Memory Type is 'rom'.
- Any other information. For example control connection details for multiplexers, in a control sequence whose destinations include multiplexers, as output by MC<sup>2</sup>.

The format of this data within the access sequence file is shown in Figure 7.3.

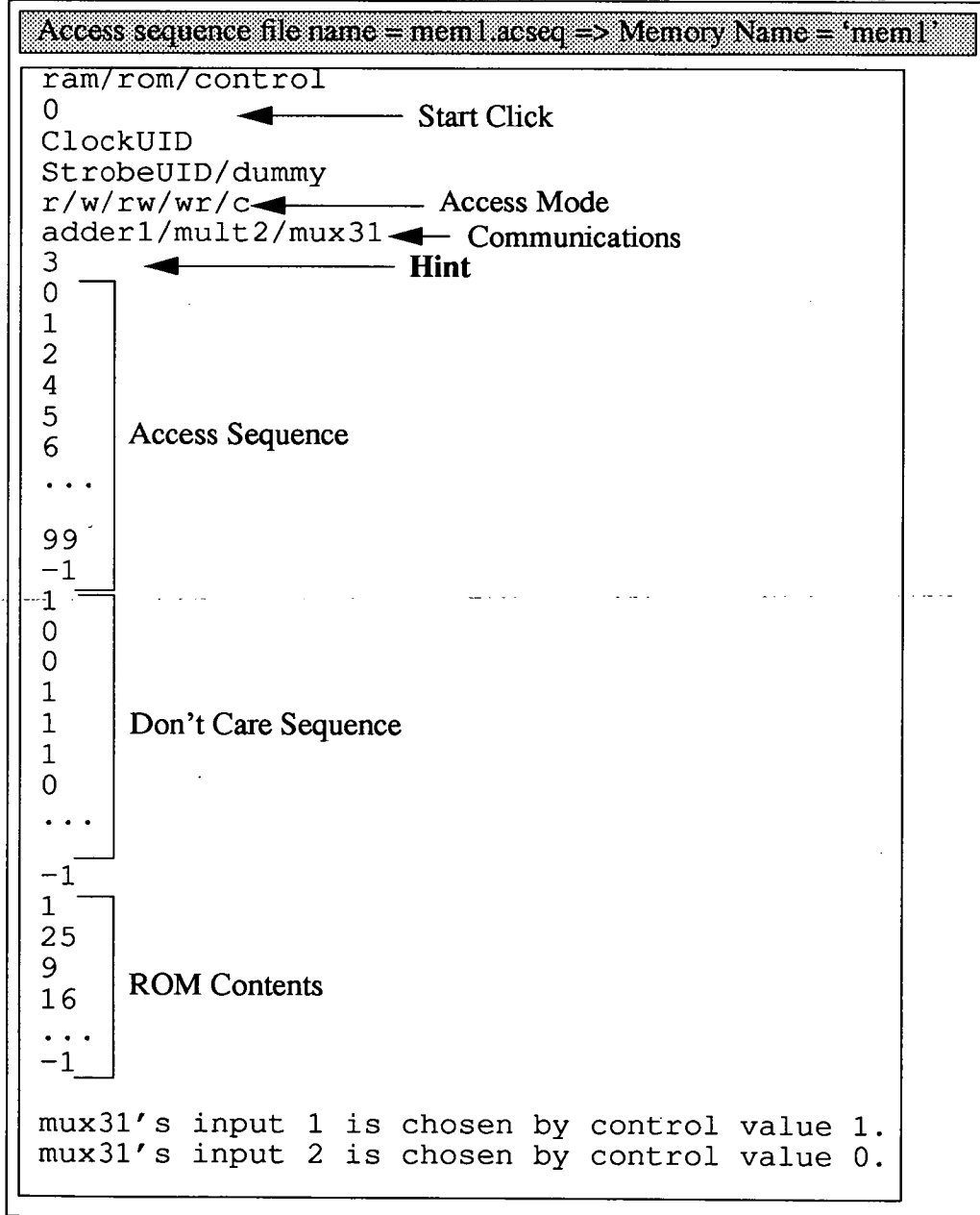


Figure 7.3 Access Sequence File Format.

This standard format may be compiled by hand, but facilities exist in both MC<sup>2</sup> and AG1 for automatically producing files in this format.

#### **7.4.2 Basic method**

This section will describe the support procedures which transfer the data from the access sequence file into the internal data structure, and also the basic method for synthesis, without getting into too much detail.

The main procedure in AG2 is run twice during the synthesis process, first to pick up all the address sequences defined externally, and then to synthesise address-ROM address generators, for any ROMs which have been created as look-up tables as part of other address generators. In the first case, the system enquires as to the data capture method: Using a drive file containing the names and memory data widths of several memory access and control sequences (width = 1); Or loading data from a single access sequence file and then manually providing the data width.

To commence the loading of information, a check is made on the total size (in bits) of the access sequence, and if this is too large then the system will load each bit sequence of the access sequence individually. The maximum address, access mode and memory name and type are also derived at this point, and are used to either add the information to an existing memory record, or to produce a new one, adding ports to each RAM as required. Then the rest of the access sequence information is loaded and stored in the corresponding memory record, before the synthesis stage-proper can begin.

The first type of address generator sought is an incrementor. The incrementor may be preset for the first clock tick, and may be reset at a fixed value after that. It should have a fixed increment, which may be a positive or negative integer, and need not increment every click.

If an incrementor cannot be used to generate the given sequence then it is split bitwise and an attempt is made to match each individual bit sequence to a counter-based bit sequence generator. This involves using the given *hint* (or derived hint, if not given) to pad out the bit sequence so that it is of length  $H \cdot 2^n$ , where  $H$  is the hint. This allows iterative bisection of a compact representation of the bit sequence, quite like the method used in AG1, to hopefully find a cheap bit generator.

Finally, for address sequences which are not too large, each constituent bit sequence is compared with others to try to find one which may be generated using another sequence to clock a flip-flop.

Once all this has been done for all bit sequences in each access sequence file, costs are derived for each possible implementation of each bit sequence generator, based on their area and possible hardware sharing. These costs are then iteratively optimised until each bit sequence has a single, definite generator, and then any sequences which have been bound to address *ROMs*, are collected together to form the contents lists of those *ROMs*, and the access information for the *ROMs* is handed back to the main procedure as a whole new set of access sequences, but which cannot be generated by *yet another* address *ROM*.

### 7.4.3 A working example

In order to show as many features of the system as possible, a set of three access sequences will be used as a working example. The first two contain memory access sequences for the same memory, and the third is a control bit sequence for a set of multiplexers. The first address sequence bears little resemblance to those normally seen, since its constituent bit sequences have been tailored to illuminate certain points, and then combined to give the address sequence. More realistic examples will be detailed in a later section. The access sequences, as handed to AG2, are given in Figure 7.5, and their file names and memory widths are contained in a drive file, shown in Figure 7.4.

```
mem1.wacseq
8
mem1.wracseq
8
csig21.cseq
1
```

Figure 7.4 Contents of drive file for working example.



#### **7.4.4 Method**

A primary stage, before loading an access sequence from file, is necessary to establish whether the size of the sequence will allow it to be loaded in its entirety, or only on a bitwise basis. Then the memory name is derived from the input file name, and an attempt is made to find this amongst existing memory records. If one is found then that memory record is returned to have another address generator added, and otherwise a new memory record of the correct type is created, with a single port of the required access mode. Then the dimensions of the memory are updated if necessary, for example if a larger address appears than in an address sequence already treated.

Finally the rest of the access sequence information is transferred to the internal data structure, changing or adding ports to the memory if required. A ROM may only have a single port of mode R (Read), while a RAM may have as many ports as required of any mode, and if a RAM port of mode R or W is available for an access sequence of the opposite mode (W or R) then the port will be extended to be of mode RW - Read and Write. Now the synthesis algorithms start in earnest to examine the sequence for possible generation methods, starting with incrementor-based generators.

##### **7.4.4.1 Finding An Incremental Sequence**

In iteratively examining the address sequence, we look for several points:

- The maximum address must be  $> 1$ .
- The very first address is the preset value for the incrementor.
- If the next address is the same as the present one then a gating signal is required on the clock, and it should be set to '0' for the present clock tick.
- If the next address = 0, for the first time in the sequence (apart from the very start) then the modulus of the incrementor is made equal to the previous address plus the increment value - the address which the incrementor will not quite reach. Also any gating signal should equal '1' for this clock tick.
- If, when the next address = 0, the present address = the modulus minus the increment, then any gating signal will equal '1', and if not then the sequence cannot be generated using an incrementor.
- If the next address is different from the present one, for the first time in the sequence, then the increment is recorded as `next_address` minus `this_address`, and also any gating signal should be '1'.

- If the next address differs from the present one by the correct increment, then any gating signal should be '1' for this clock tick but if the increment is different to the recorded one then an incrementor is not suitable for this sequence.

If the whole address sequence passes the above tests then a corresponding set of bit generators must be constructed. If the increment is unary then a counter is more efficient than an adder and such a counter is constructed, with possible preset and gating signal. The counter modulus is the maximum address plus 1, and only the lesser bits of such a counter are required. If the increment is greater than one then an adder will be used, and the output bits are numbered from 0 up to the MSB of that adder. If a gating signal is not required for the incrementor then the name of the dummy signal is set to 'clock' and otherwise takes the form 'gating\_sigUID'. Figure 7.6 shows how an incrementor type solution is found for *mem1.wrseq* in our working example.

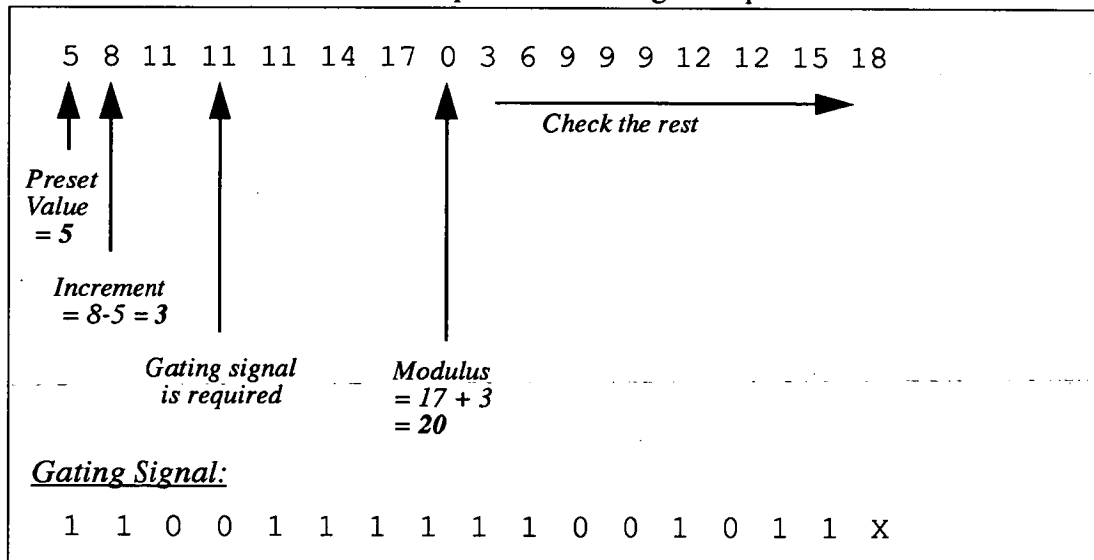


Figure 7.6 Finding An Incrementor-based Solution.

#### 7.4.4.2 Padding A Bit Sequence

Assuming that the address sequence cannot be generated using an incrementor, the next task is to try to match each constituent bit sequence to a counter bit or logical combination of bits. To allow this matching process to function correctly, the bit sequence must first be padded out so that it has length  $\text{Hint} * 2^n$ . This involves appending a copy of part of the bit sequence to its end, inverting the polarity of the copy if necessary. The hint may be predefined, and if so will be used in padding ALL bit sequences in the address sequence, but if it is not specified (ie: Hint = 0) then it must be



found automatically for each bit sequence in turn. This can be very time consuming for very long sequences since it involves taking successively longer sections from the start of the sequence, which a separate algorithm attempts to fit repetitively onto the rest of the sequence, as explained in Figure 7.7.

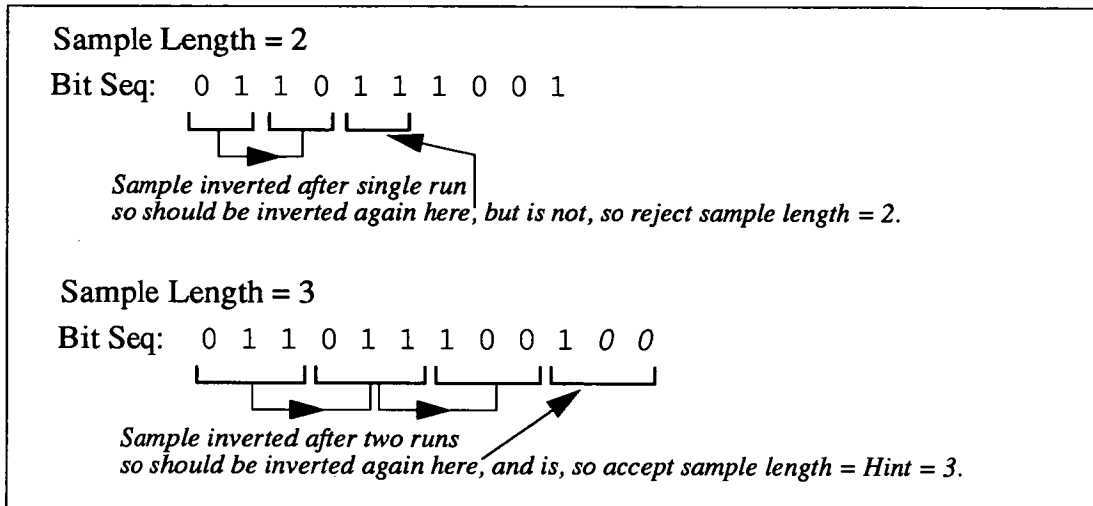


Figure 7.7 Finding a hint where none is given.

The padding routine consists of two stages, after the hint is discovered. The first stage copies a section from the first basic pattern, to get a whole number of these basic patterns, as detailed for a pair of bit sequences in Figure 7.8a, and then the second stage adds the padding to get the final length,  $H \cdot 2^n$ , as shown in Figure 7.8b

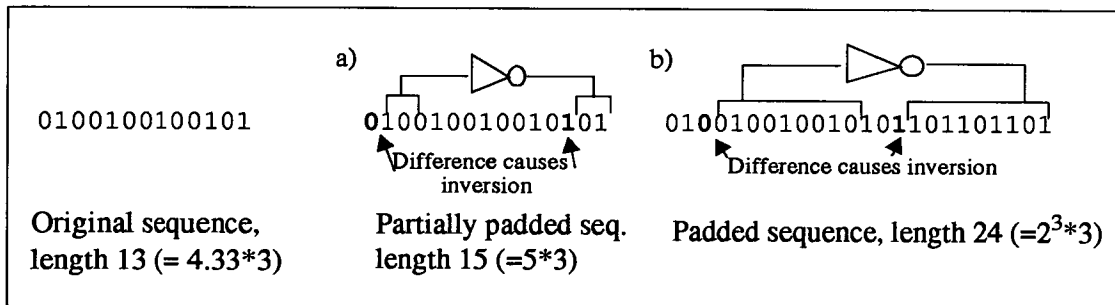


Figure 7.8 Padding a bit sequence: a) to a whole number \* Hint; b) to a power of two \* Hint.

#### 7.4.4.3 Transformation To Repetition Sequence

The previous stage may seem to be wasteful, in that it lengthens the bit sequence by up to (Original\_length - 1) bits, but the next stage, during which the *repetition sequence* is formed, aims to reduce the sequence length by a lot more than that. The repetition sequence contains a polarity value - the first bit of the corresponding bit sequence - followed by a sequence of integers describing the number of adjacent bits with the same

values. Examples of the repetition sequence construction for the *mem1.wacseq* example are given in Figure 7.9 and the final one, not from our working example, demonstrates the savings possible through this transformation, with a bit sequence of original length 32k reduced to a repetition sequence of length 2.

Bit No.	Padded bit sequence	Repetition Sequence
7	011100011100100011100011 length = 24	0, (1,3,3,3,2,1,3,3,3,2) length = 10
6	0000000111111111111110000000 length = 28	0, (7,14,7) length = 3
5	11111111111111110000000000000000 length = 32	1, (16,16) length = 2
4	10100101001010010100 10100101001010010100 length = 40	1, (1,1,1,2,1,1,1,2,1,1,1,2,1,1,1,2, 1,1,1,2,1,1,1,2,1,1,1,2,1,1,1,2) length = 32
3	000000111111000000111111 length = 24	0, (6,6,6,6) length = 4
2	0001111100000011111000 length = 22	0, (3,5,6,5,3) length = 5
1	00011100111000110001110011100011 length = 32	0, (3,3,2,3,3,2,3,3,2,3,3,2) length = 12
0	00101101001011010010110100101101 length = 32	0, (2,1,1,2,1,1,2,1,1,2,1,1,2,1,1, 2,1,1,2,1,1,2,1,1) length = 24
<div style="display: flex; align-items: center; justify-content: center;"> <div style="text-align: center; margin-right: 10px;"> <math>\longleftrightarrow</math> 16k bits 11111...00000... length = 32k </div> <div style="text-align: center; margin-right: 10px;"> <math>\longleftrightarrow</math> 16k bits 11111...00000... length = 32k </div> <div style="text-align: center;"> <math>\longrightarrow</math> </div> </div>		(1, (16384, 16384)) length = 2

Figure 7.9 Construction of repetition sequences.

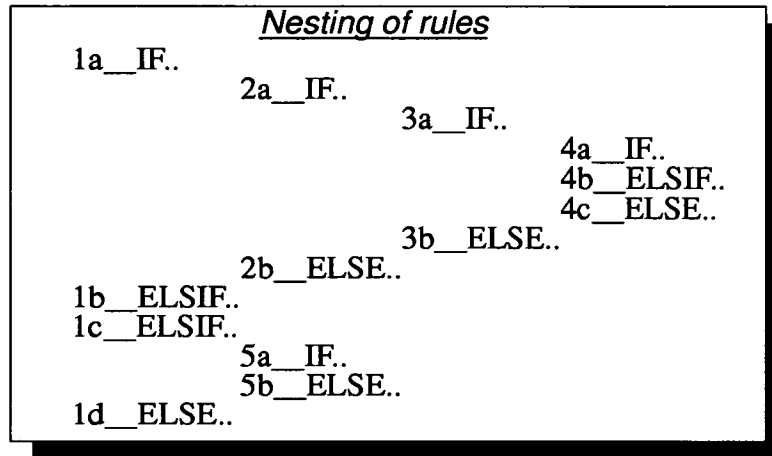
#### 7.4.4.4 Reducing The Repetition Sequence

The length of the repetition sequence (not including the polarity value) is very important in the synthesis process. We can tell straight away if the bit sequence contains only one polarity of bit, and should be hardwired to logic '0' or '1', when the repetition sequence length is unary. If this is not the case then we start to iteratively bisect the repetition sequence, to hopefully find a counter-based solution to its corresponding bit sequence's generation. Different routes are taken in reducing the repetition sequence, depending on whether its length is odd or even, and whether half of its length is odd or even, as described by the following rules (BSG - Bit Sequence Generator).

### **Rules**

**$L\_RS$  = length of the repetition sequence,  $m = L\_RS / 2$ .**

- 1a) IF  $L\_RS$  is even THEN Rule 2.
- 1b) IF  $L\_RS = 1$  THEN find\_BSG using remaining sequence.
- 1c) IF  $rep\_seq(2..(m-1)) = rep\_seq((m+1)..(L\_RS-1))$  AND  $rep\_seq(1) + rep\_seq(L\_RS) = rep\_seq(m)$  THEN Rule 5.
- 1d) find\_BSG using remaining sequence.
  
- 2a) IF  $rep\_seq(1..m) = rep\_seq((m+1)..L\_RS)$  THEN Rule 3.
- 2b) find\_BSG using remaining sequence.
  
- 3a) IF  $L\_RS/2$  is odd THEN Rule 4.
- 3b) Bisect the sequence and recurse using the first half.  
EG: Bit 1: (3,3,2,3,3,2,3,3,2,3,3,2) => (3,3,2,3,3,2).
  
- 4a) IF  $L\_RS/2 = 1$  THEN bisect the sequence and recurse.  
EG: Bit 5: (16,16) => (16).
- 4b) IF we can generate the corresponding bit sequence, as the repetition sequence stands, with a single counter bit, then find\_BSG using the current repetition sequence.
- 4c) The corresponding bit sequence is the result of EXORing a counter bit (found from  $L\_RS$ ), with whatever BSG is found by recursing using the first half of the sequence.  
EG: Bit 7: (1,3,3,3,2,1,3,3,3,2) => (1,3,3,3,2) EXOR 2(3) (Bit 2 of a modulus 3 counter).
  
- 5a) IF  $(L\_RS+1)/2$  is even THEN as for RULE 4c.
- 5b) Bisect the sequence and recurse.  
EG: Bit 2 : (3,5,6,5,3) => (3,5,3).



#### 7.4.4.5 The Repetition Sequence Characteristic

Now the repetition sequence has been collapsed as far as possible, the remaining sequence is sent to be matched to a bit sequence generator. It is first converted into another format, to ease this matching, which consists of four parameters, as shown in Figure 7.10.

## Repetition Sequence Characteristic:

Polarity (as for repetition sequence) - P	First repetition in repetition sequence - $R_1$
Repetition count of first repetition - $R_R$	Final repetition, if different - r

Some examples help to explain this:

Repetition sequence

Rep. seq. Characteristic

Corresponding bit seq.

$(0, (2, 2, 2, 1)) \Rightarrow$

0	2
3	1

$\Leftarrow (0, 0, 1, 1, 0, 0, 1)$

$(1, (4, 4, 4)) \Rightarrow$

1	4
3	0

$\Leftarrow (1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1)$

$(0, (1, 2)) \Rightarrow$

0	1
1	2

$\Leftarrow (0, 1, 1)$

$(\Rightarrow \text{bit}(-2), \text{mod } 3, \text{EXORed with bit}(-1), \text{mod } 3)$

A special case is needed to denote a random bit sequence, where all parameters = 0:

$(0, (1, 3, 2, 4, 2, 3)) \Rightarrow$

0	0
0	0

$\Leftarrow (0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1)$

sent to logic synthesis

Figure 7.10 Repetition sequence characteristic formulation.

#### 7.4.4.6 Matching the Characteristic to a Bit Sequence Generator

Now we are ready to try to find the counter bit which will produce the bit sequence, characterised as above. The counter bit is described by its bit number,  $b$ , and the modulus of the counter,  $m$ . The polarity in the characteristic determines whether a counter bit's output should be inverted by a NOT gate.

##### Finding the counter modulus, $m$

If there is a single repetition left in the repetition sequence, then its characteristic will look something like this, where,  $R_1$  is the remaining bit repetition length:

0	$R_1$
1	0

The modulus of the counter needed is found from  $R_1$  by finding the lowest odd factor of  $R_1$ , i.e.: By dividing it by 2 until an odd quotient is found. The number of times  $R_1$  can be divided is the bit number of an upper ( $\geq 0$ ) bit of the counter. For

example, a characteristic:

0	176
1	0

will be generated by bit 4 of a modulus 11 counter.

However, if there is more than one repetition left, then the modulus is calculated as:

$$\text{modulus} \left( \begin{array}{|c|c|} \hline P & R_1 \\ \hline R_R & r \\ \hline \end{array} \right) = (R_1 * R_R) + r = m$$

i.e.: The sum of the remaining repetitions. This should represent the sequence generated by a lesser ( $<0$ ) bit of a modulus  $m$  counter. For example a

characteristic:

1	2
5	1

will be generated by bit(-3) of modulus 11 counter.

If a random bit sequence is characterised, then for consistency its modulus is set to 0, as a flag. Also, if a repetition sequence remainder ( $r$ ) is greater than the first repetition, then it is possible that this may characterise a bit sequence generated by EXORing lesser bits of a counter, and these lesser bits can be found by expanding the characterised repetition sequence to its bit sequence, and then repeating the whole

synthesis process for that sequence (Padding first, using a hint of 1). For example the

characteristic: 

0	2
1	3

 represents the bit sequence 00111, which is the result of

EXORing sequences 00110 and 00001, which are produced by bits(-2) and (-1) of a modulus 5 counter, respectively.

### Finding the lesser bit number

If, once the modulus has been found, it is found to be greater than the first repetition, then a lesser bit of the counter is desired, and the correct bit is derived simply from the modulus, which determines how many lesser bits there will be, and from the first repetition, which should be a power-of-two.

#### 7.4.4.7 Finding Clocked-type Bit Sequence Generators

Whether or not a counter-based solution has been found for a bit sequence, all sequences in the address sequence are examined in the hope of finding some other bit sequence which may be used to clock a flip flop and produce the bit sequence in question. A list of repetition sequences is constructed, describing each bit sequence and this list is sorted using the first repetition in each sequence as a guide, so that the repetition sequence with the shortest first repetition is at the head of the list. This sequence is then compared to each other one in turn, comparing the sum of each successive *pair* of repetition values from the former with individual repetition values from the latter, as illustrated in Figure 7.11.

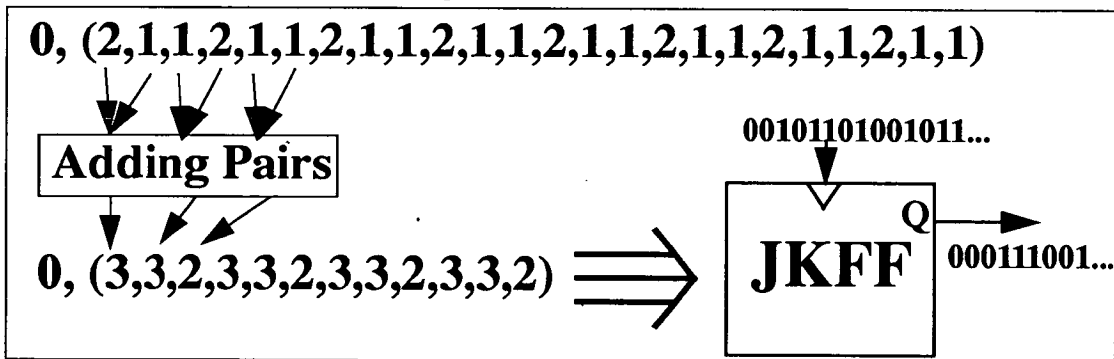


Figure 7.11 Recognising clocked-type bit sequences and their clocking bits.

Of course the clocked bit sequence may start with a '1', in which case it is said to be NOTTED. The address bit which will be used to clock the flip flop is recorded and the algorithm then goes on to examine the next repetition sequence, until all have been compared. Generating a sequence in this way introduces a skew on the timing of

address bits, but if several flip flops are to be chained together (forming a ripple-carry counter), producing a large skew, then this chain could easily be given serial or parallel carry, to reduce the skew. This is not yet implemented as an optimisation stage.

#### 7.4.4.8 Multiple Access Sequences

At this point the main procedure would reiterate, loading and analysing the next sequence named in the drive file, if that was the data entry option used, and this repeats until all sequences have been run through the synthesis algorithms. Then costs are found for each and every bit sequence generator before the optimisation stage commences. This costing will be explained in the next section.

#### 7.4.5 Optimisation

Before any optimisation of hardware in terms of area can proceed, the area costs of each bit sequence generator must be found. The generators, or parts of them, are assigned one or more possible generation methods from the following, each of which must be costed:

- Counter-based, including semi-random bit sequence generators which will by default use a counter and some combinatorial logic.
- Incrementor-based.
- Clocked-type.
- ROM-based.

Each address bit in each sequence may be generated using a ROM look-up table, accessed using a counter, so this cost is estimated for every sequence, although the different types of bit sequence generator may be given other costs as well.

*SIMPLE* bit generators are costed along with *EXORed* and *LOGIC*-type bit generators, and also with *INCREMENTOR*-based generators with a preset of zero and a unary increment. These counter-based costs are calculated in the following way. Having determined all the moduli of the counters required, for each of these moduli a list of all counter bits-used is compiled, with the least significant first. Then the cost of implementing each counter bit in turn is based on the nearest existing bit on the counter, the size of a JK flip flop, and on the number of bit sequence generators which can share this hardware, as shown in Figure 7.13 overleaf. If a bit generator is an incrementor bit with increment equal to one, then a gating signal may be required and the extra cost for this is based on the ROM area required to store the gating sequence. If a semi-random

bit sequence exists then it will require all lesser bits of a counter whose modulus equals the length of the random part of the sequence, and an extra cost for the combinatorial logic is based on that length also.

The compatability of bit generators sharing the same hardware is determined by comparing their clock names and either their strobe names or the gating signals themselves, before looking at the start and finish times of the sequences. Two sequences are compatable if their start times and their finish times coincide, or if the sequences do not overlap in time, or finally if the differences between the two start times is equal to an integer multiple of the length of repetitive bit sequence naturally produced by the counter bit in question, as shown below.

Clock Tick:	0	1	2	3	4	5	6	7	8	9...	(= 3 * 3)							
Seq. 1:	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1
Seq. 2:										1	1	1	0	0	0	1	1	1

Figure 7.12 A pair of compatable sequences.  
These two sequences can be produced by the same (modulus 3) counter.

**INCREMENTOR**-based bit sequence generators are grouped onto adders and share the cost of any support required, such as storage for the increment value and the gating sequence.

**CLOCKED**-type bit generators are grouped by their interconnectivity (ie: In chains), and each is given the area cost of a JK flip flop.

**ROM**-based bit sequence generators are grouped into ROMs by the compatability of the lengths and timings of the sequences and each generator is given a ROM area cost based on the length of the sequence, and an equal share of any ROM address generation costs and of decoder/driver hardware costs incurred during the creation of the ROM (a flat-rate cost).

Once all possible costs for each bit sequence generator have been calculated, it remains to choose the globally optimal method of generation for each one. This is obviously another NP-complete problem, in that the sharing of costs between generators is dependant on them using one or another method of generation, and randomly selecting bit generators and binding them to, or inhibiting the use of certain hardware is not feasible, unless run-times are to skyrocket.



The approach taken here is to use the total of the cheapest possible costs of every bit sequence generator as a target to aim for, and then calculate the total costs incurred using as much ROM-based bit generation as possible, and using as little as possible. Comparing the latter two extremes should show one to be higher than the other and the bit generator with the greatest extra expense by the more expensive method is forced not to adopt that method in the future. The cost functions are then run again to determine the new costs of bit generators after this binding, and if the cheapest possible cost has increased by more than a small amount then the previous decision is reversed. This continues until every bit sequence is tied to a specific bit sequence generator and it is now that the main synthesis routine is run once more, with the ROM-based generators grouped into ROMs to be fed back as internally-supplied access sequence information and then to have their own address generators synthesised. At the moment the ROM address sequences are simple counts from zero, but it would be possible to take advantage of coincident 'Don't Care' times in the individual bit sequences to reduce the ROM size, and perhaps the size of its own address generator.

**Counter-based costs:**

If we require bit  $b$  on a modulus  $m$  counter, and there already exists the lower significant bit  $(b-l)$ , then the cost of the new counter bit is:

$$l * JKFF\_SIZE$$

If we require bit  $b$  on an incrementor with unary increment, preset = 0, which requires a gating signal, and there already exists bit  $(b-l)$ , then the cost of the new counter bit is:

$$(l * JKFF\_SIZE) + (ROM\_BIT\_SIZE * Sequence\_length)$$

If we require a random bit sequence of length  $L$ , then we will need all lesser bits of a modulus  $L$  counter, and the cost is given by:

$$(\log_2(L) * JKFF\_SIZE) + (L / 4)$$

These costs should be divided equally between all bit sequence generators which can share the hardware.

**Clocked-type costs:**

If there exists a sequence which can be used as a clock for a flip-flop to generate a second sequence, then the cost of the generator of that second sequence is given as:

$$JKFF\_SIZE$$

*Figure 7.13 Cost Functions*

**Incrementor-based costs:**

If we require bit  $b$  on an incrementor with increment  $i$ , preset to  $p$ , and there already exists the lower significant bit ( $b-l$ ), then the cost is calculated as:

$$(l * \text{ADDER\_BIT\_SIZE}) + \text{Extra\_cost1} + \text{Extra\_cost2}$$

If the increment  $> 1$  and the modulus is  $m$  then the Extra\_cost1 is:

$$(\log_2(i) * \text{ROM\_BIT\_SIZE}) + (\log_2(m) * \text{LATCH\_SIZE})$$

The Extra\_cost2 must be added if a gating sequence, length  $L$  is required:

$$L * \text{ROM\_BIT\_SIZE}$$

These extra costs should be shared equally between *all* bit sequence generators using the incrementor, while the basic cost should only be divided between generators using the given bit on the incrementor.

**ROM-based costs:**

Every bit sequence is treated as if it were to be placed in a ROM.

There are three distinct costs involved in the address ROMs created from a conglomeration of compatible bit sequences.

Each bit sequence (Length =  $L$ ) is given the cost of ROM area it will use:

$$L * \text{ROM\_BIT\_SIZE}$$

If a ROM is constructed from  $N$  compatible bit sequences, of length  $L$  each bit sequence generator receives a share of the cost of support circuitry (decoders, etc.):

$$\text{ROM\_CREATION\_COST} / N$$

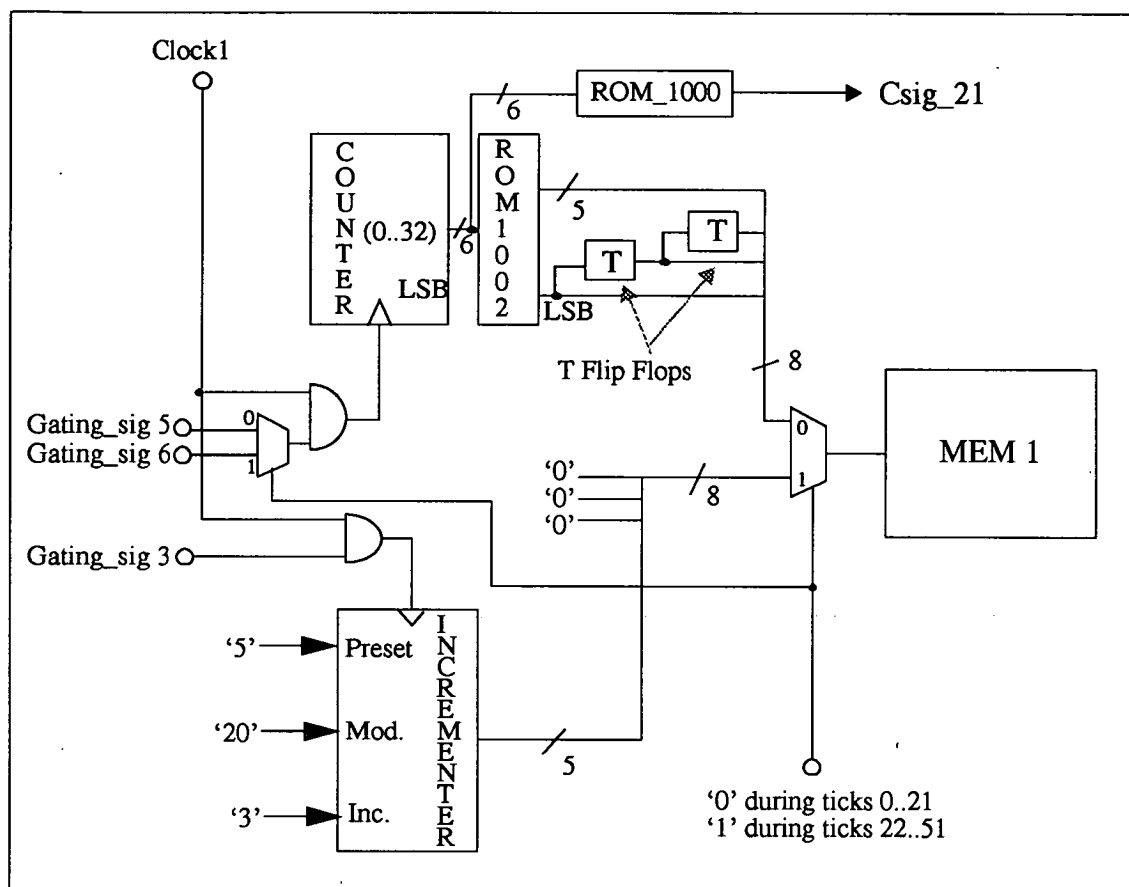
There will also be a share of the cost of addressing that ROM:

$$(\log_2(L) * \text{JKFF\_SIZE}) / N$$

*Figure 7.13 (continued) Cost Functions*

## **7.4.6 Output Format**

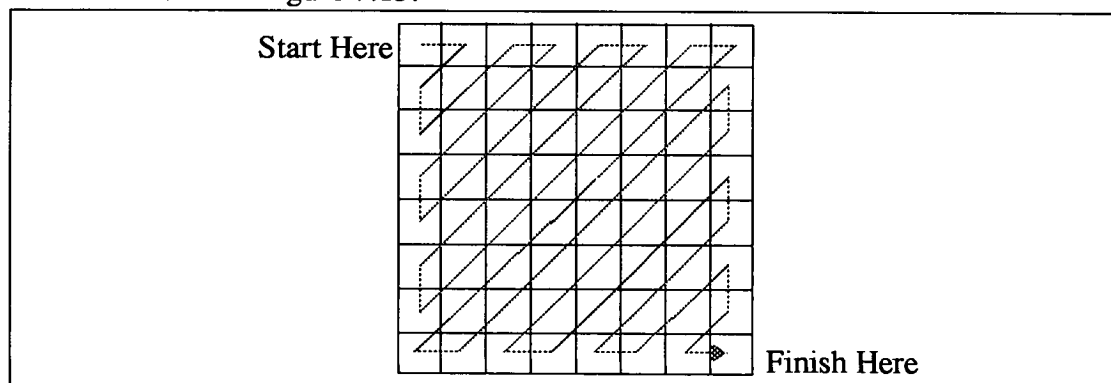
Appendix B contains annotated extracts from AG2's output for the working example. The schematic for this example is shown in Figure 7.14.



*Figure 7.14 Schematic diagram of synthesis result for working example.*

## 7.5 Other worked examples and results

The first example from the real world is that of a memory access pattern for image compression. A 256 by 256 word memory is to be accessed in 8 by 8 blocks of pixels, on a row by row or column by column basis, where the zig-zag access pattern inside the blocks is shown in Figure 7.15.



**Figure 7.15** Access pattern of 8 by 8 block of pixels.

The solution to this problem is not a simple one, but with a bit of thought a decent attempt may be made. Having input the 64k-word access sequence in row-by-row format - generated automatically using the graphical entry tool - to AG2, we get the following result.

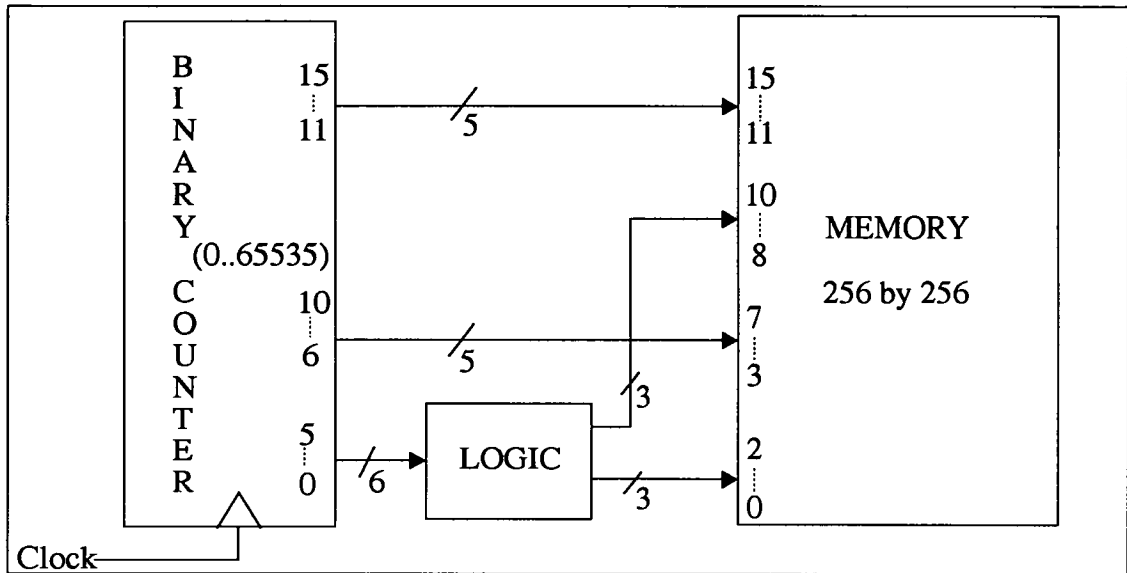


Figure 7.16 Primary solution to image compression access problem.

This is the same result as was obtained manually, for which the combinatorial logic block was synthesised automatically as about 100 gates\*. Further examination of the output from the logic block leads to the conclusion that the sequence is symmetrical about its mid-point, and the second half of the sequence is in fact the first half of the sequence, inverted in polarity and in time. To get a sequence to reverse in time is possible if the first half of the sequence is stored in a look-up table (LUT) which must be traversed in both directions. For our example, the LUT will be 32 words long, and the access sequence for it should be a count from 0 to 31 and then back. This may be generated by a 6-bit binary counter as demonstrated in Figure 7.17.

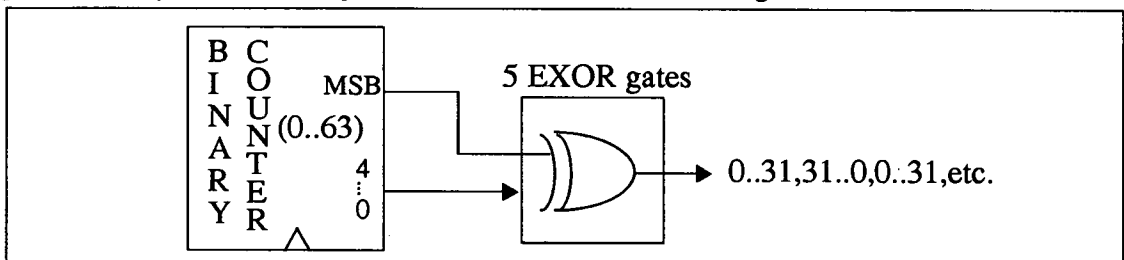


Figure 7.17 Producing an up/down count from a simple counter.

\* Thanks to Martin Bolton of SGS-Thomson for this information.



The next two real examples come from another vision application, and are best described using a set of address/time graphs. Each example has a Write and a Read address sequence, as shown in Figure 7.19 to Figure 7.22.

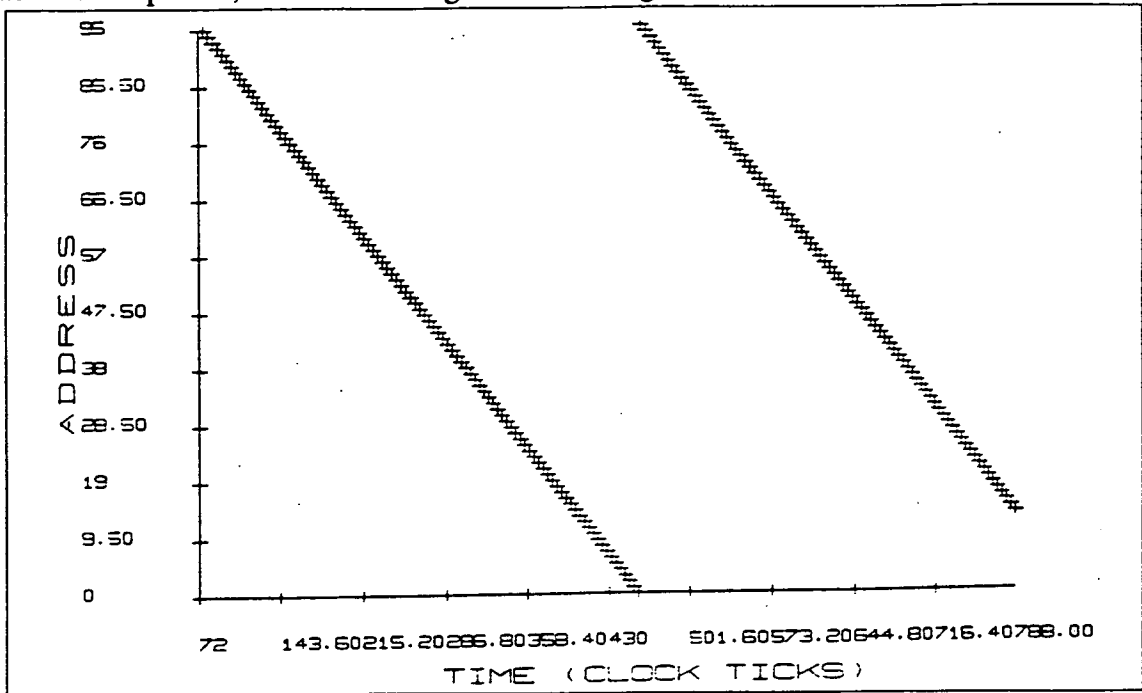


Figure 7.19 Example 1 Write address sequence plotted against time.

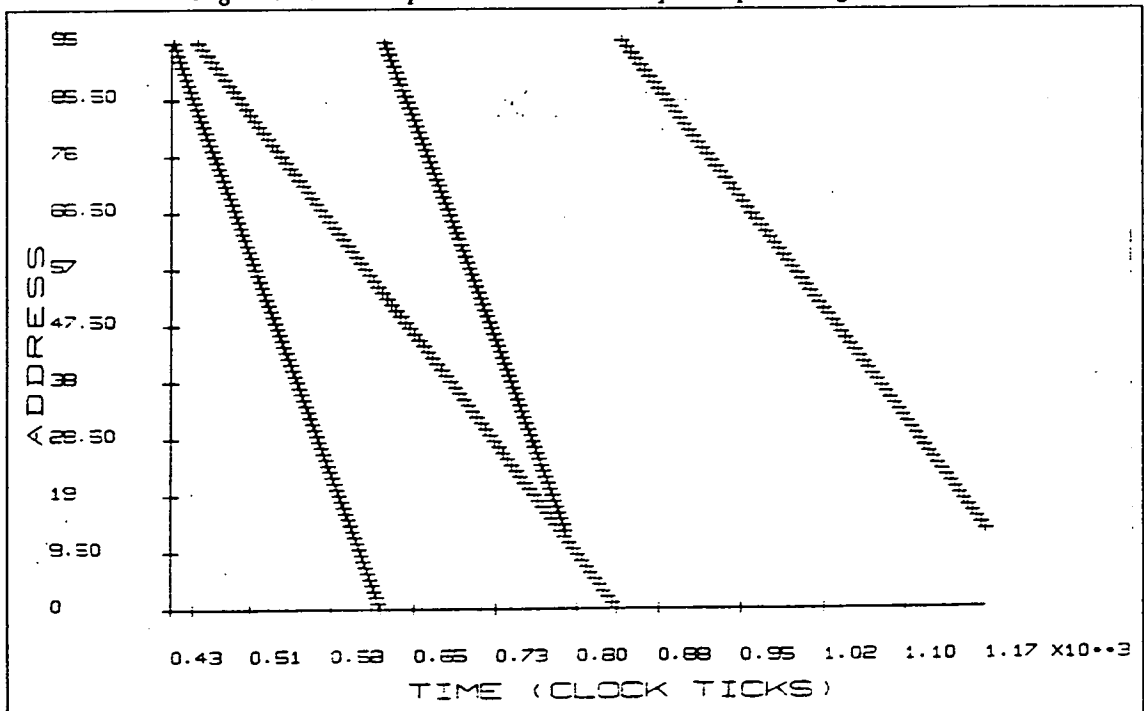


Figure 7.20 Example 1 Read address sequence plotted against time.

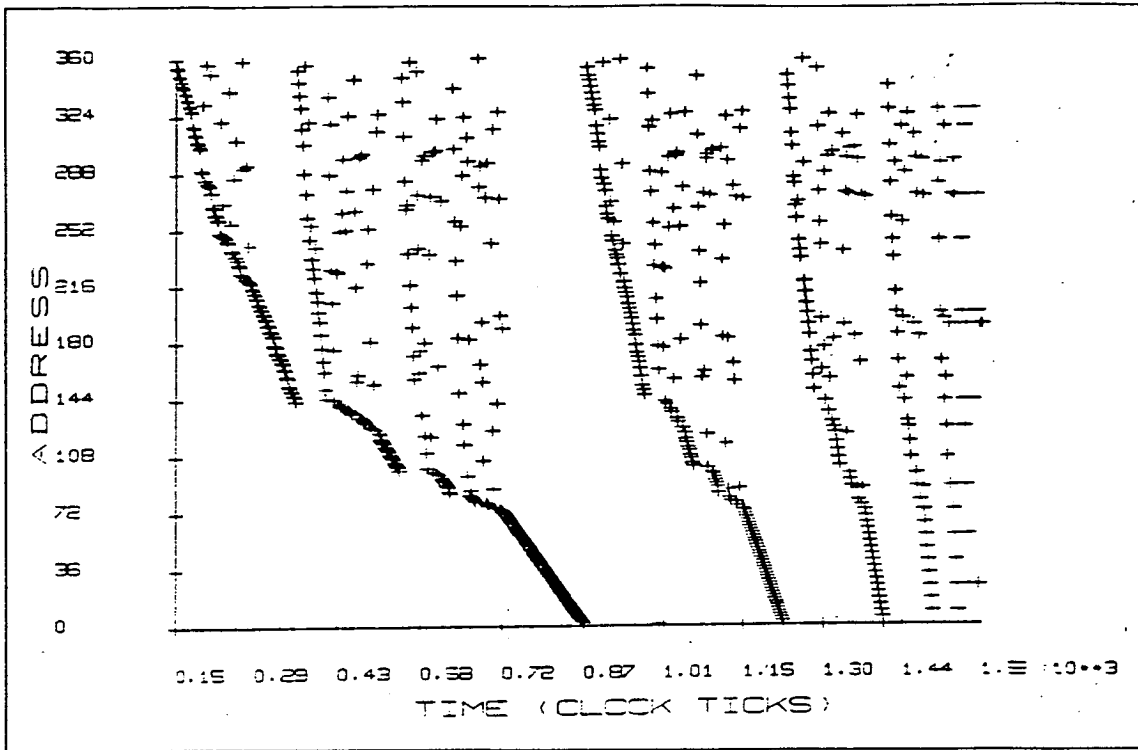


Figure 7.21 Example 2 Write address/time graph.

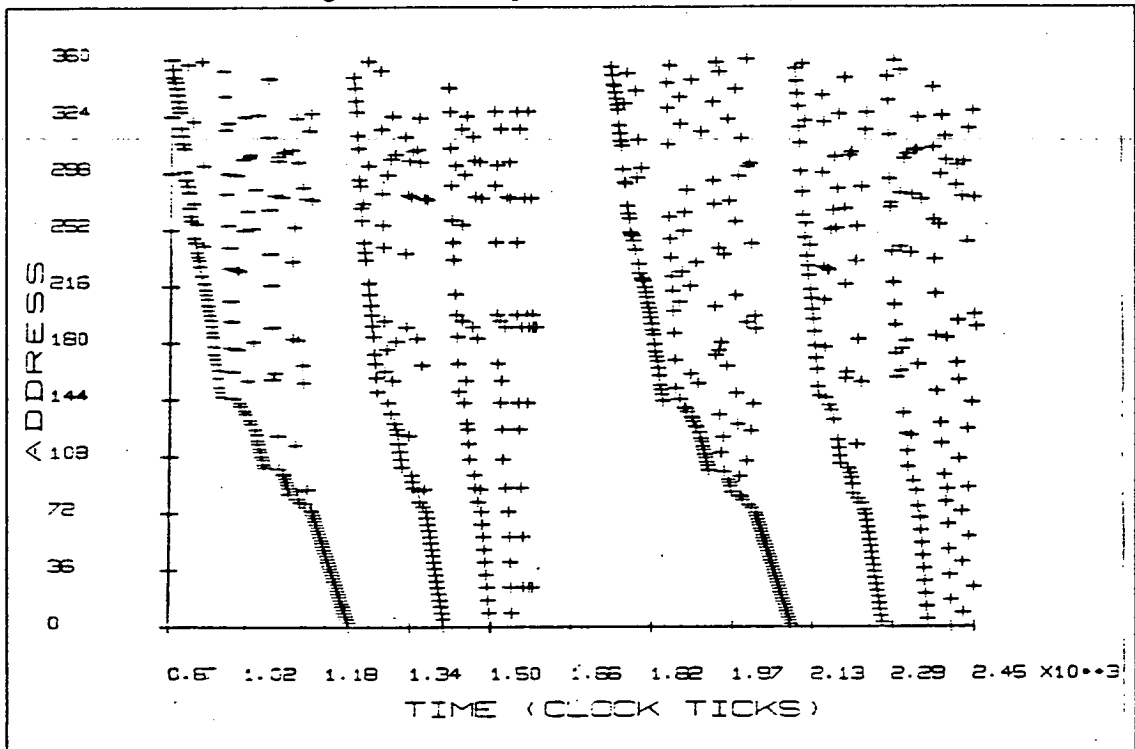


Figure 7.22 Example 2 Read address/time graph.

Examining these graphs we can see that the first example requires rather simple address sequences, while the latter requires much more complex sequences. The resulting address generators for the first example are shown below and these were derived from the output of AG2 without the temporal partitioning of the sequences, which may have helped.

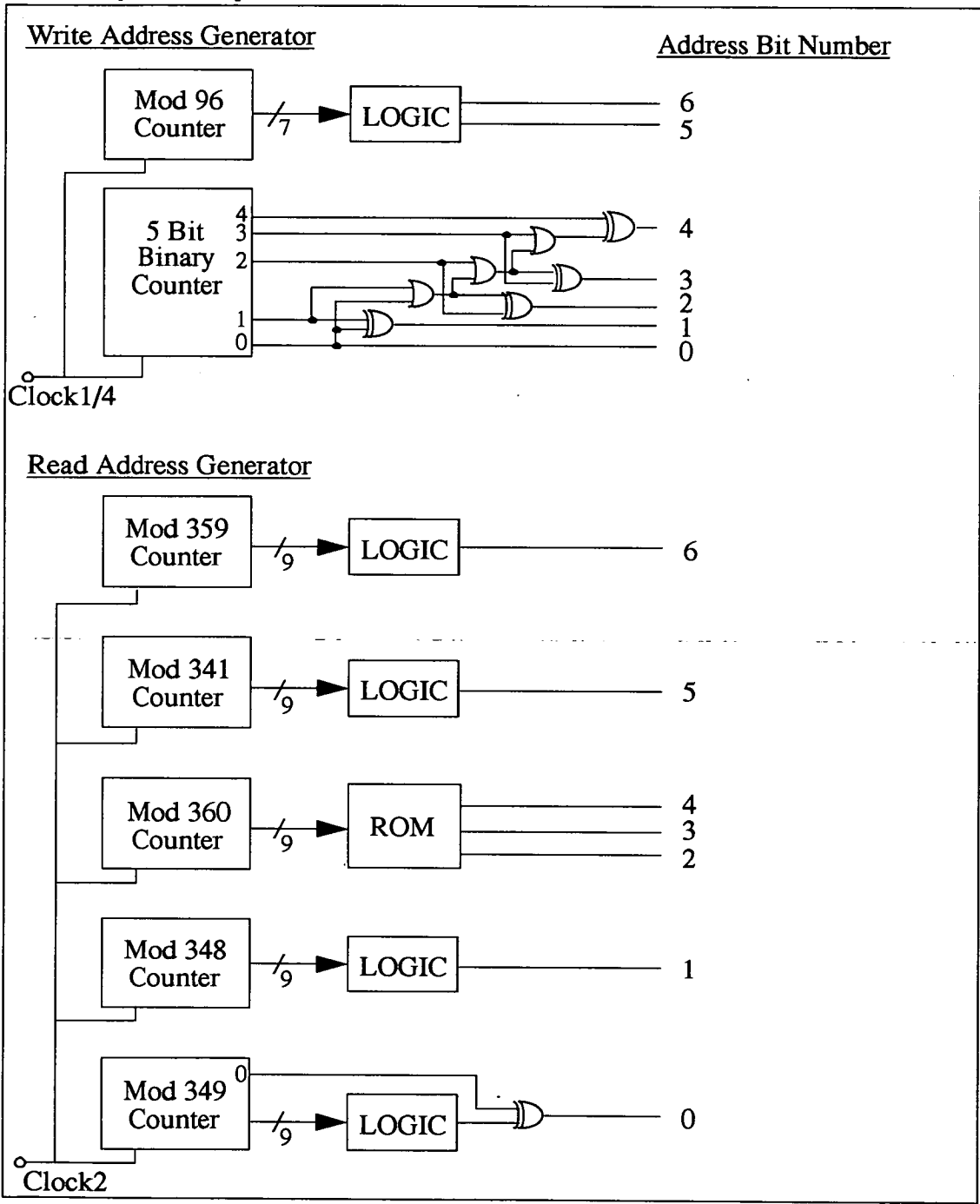


Figure 7.23 Resulting address generators for first example.



The Read address sequence, containing 360 addresses is obviously rather complex, requiring a lot of combinatorial logic to implement its generator. The specified architecture of several large counters stems from that fact that each individual address bit sequence ends with the same short (and possibly inverted) bit sequence as it starts with. This short repetition does not make the sequence much easier to generate through logic, so that it is likely that the final architecture after designer intervention would be as detailed in Figure 7.24, with the individual logic blocks combined to aid optimisation. It is hoped that this process of optimisation will be automatised at the earliest date..

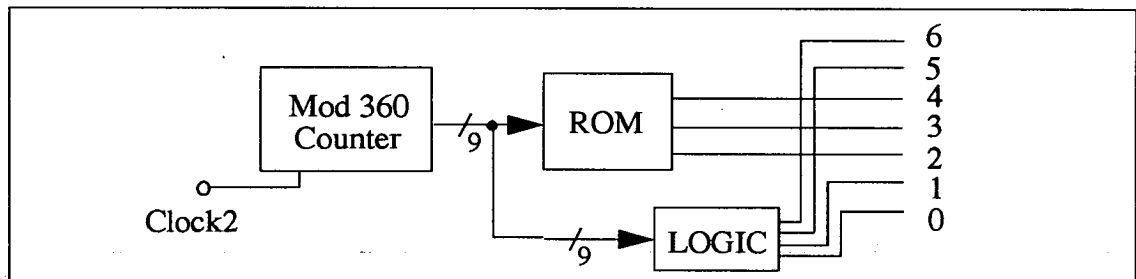


Figure 7.24 Final address generator architecture for the Read address sequence.

Another problem not yet approached is that of timing. Obviously with all the logic required, as well as ROM access delays, the output signals would be difficult to synchronise, and the critical path through a logic network may be longer than the maximum specified in the problem. This is not a problem we wish to approach at the moment because of the difficulty in technology mapping, but let it be said that the results given here are not constrained by timing considerations. In fact, the Write address generator may also take the form shown in Figure 7.25, where the knowledge that there will be an appreciable delay through the logic block is used to authorise the use of a ripple counter-type architecture.

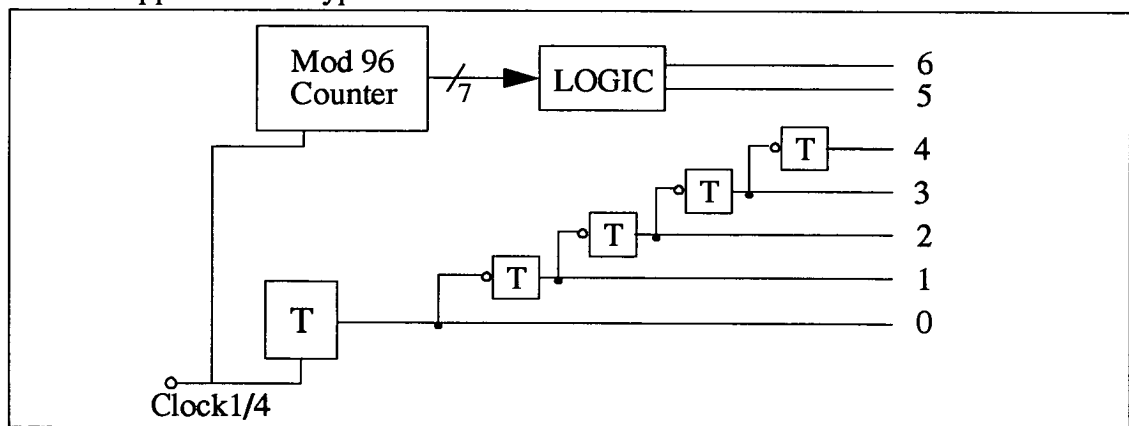


Figure 7.25 Alternative Write address generator architecture for first example.

It is probable that a more efficient solution could be found for the Read address generator if the address sequence was partitioned into sections of obvious regularity - The diagonal address/time graph sections - but because the address sequence was not continuous in time (There is an irregular clock signal required), making the optimisation unpredictable, this partitioning was not attempted.

The second, and much more irregular pair of address sequences yielded the design illustrated in Figure 7.26. This again shows the signs of a sequence, length 720 words, whose complexity merits the construction of a single pair of logic blocks of some form, although it is interesting to note from the address/time graphs that the second group of three scans of memory in the Write address sequence is repeated exactly as the first group of three scans in the Read address sequence, and could thus share hardware.

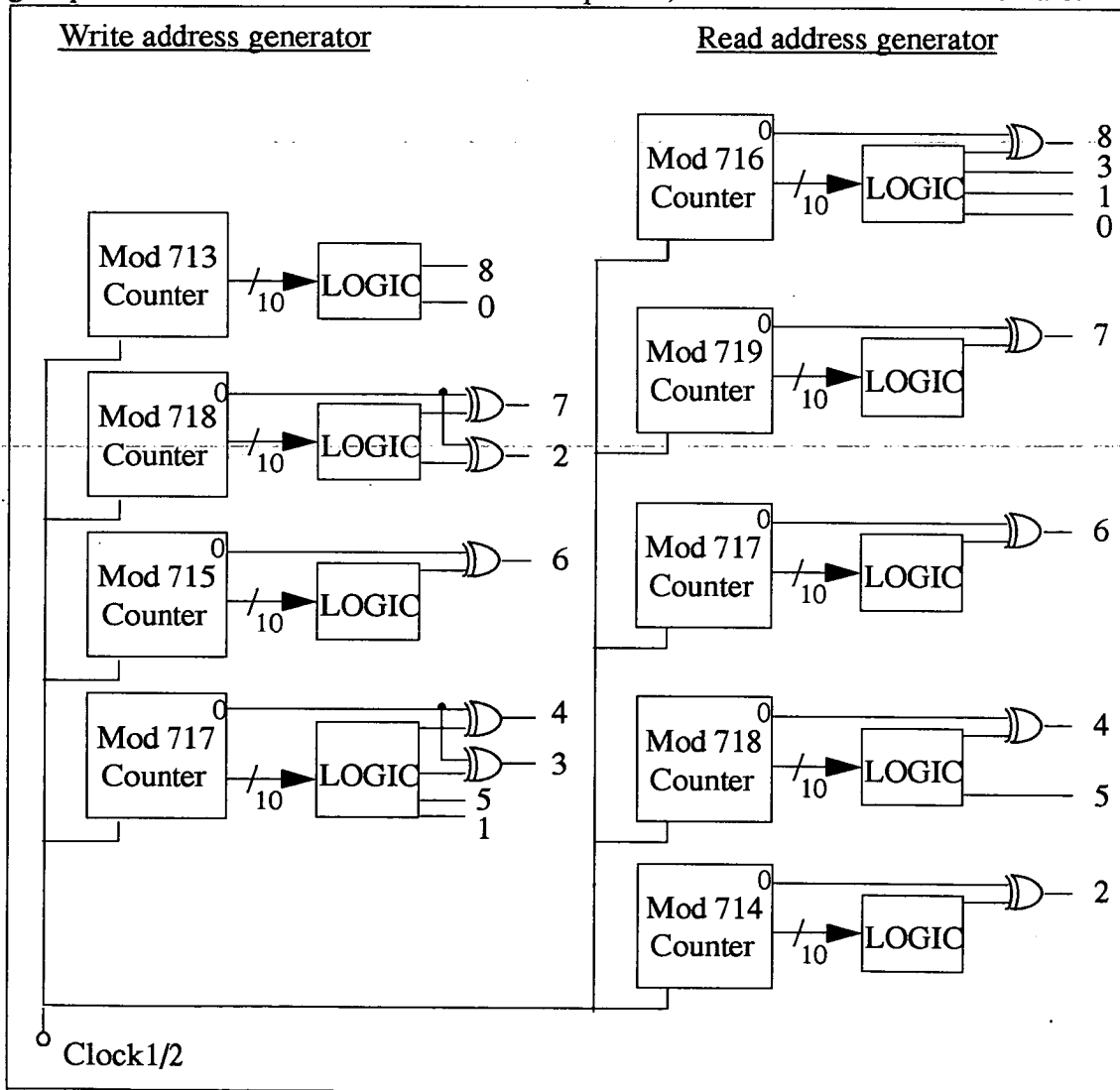


Figure 7.26 Address generators for second example.

The final real example comes again from an image processing application and is certainly the most complex of the examples presented. The address/time graphs for the Write and Read address sequences are given in Figure 7.27 and Figure 7.28. The approach to synthesis here is to partition the address sequences into sections which show some regularity, and this is especially useful for the Read address sequence which shows obvious points of interest.

Examining more closely the second half of the Read address sequence, shown in Figure 7.29a and b, the regularity is obvious, and handing this section to AG2, as a high address sequence, a low address sequence and a high-low control sequence, produces the address generation architecture shown in Figure 7.30. We have ignored the timing and clocking of this generator for simplicity, but there may be a skew problem with the long chain of flip-flops. This partitioning technique for address generator synthesis would be greatly helped by the inclusion of some graphical interface to allow easier sectioning of the sequences, but unfortunately the time and resources were not available, so this has to remain a part of future work.

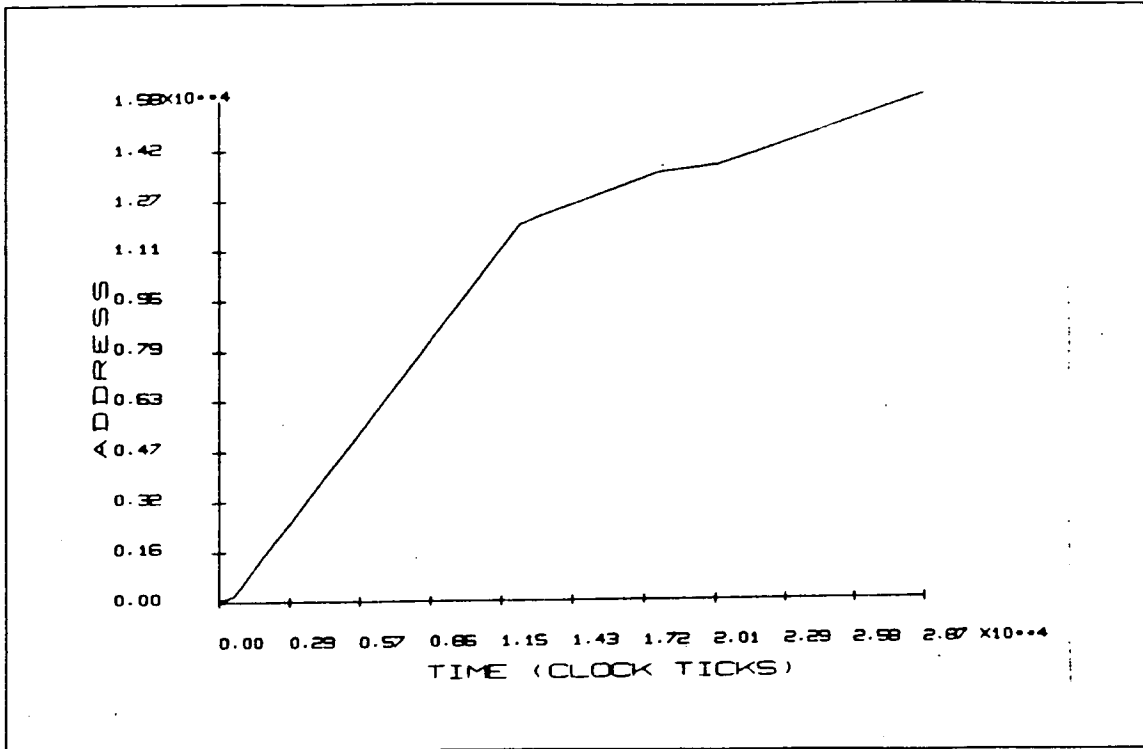


Figure 7.27 Address/time graph for Write address sequence for final example.

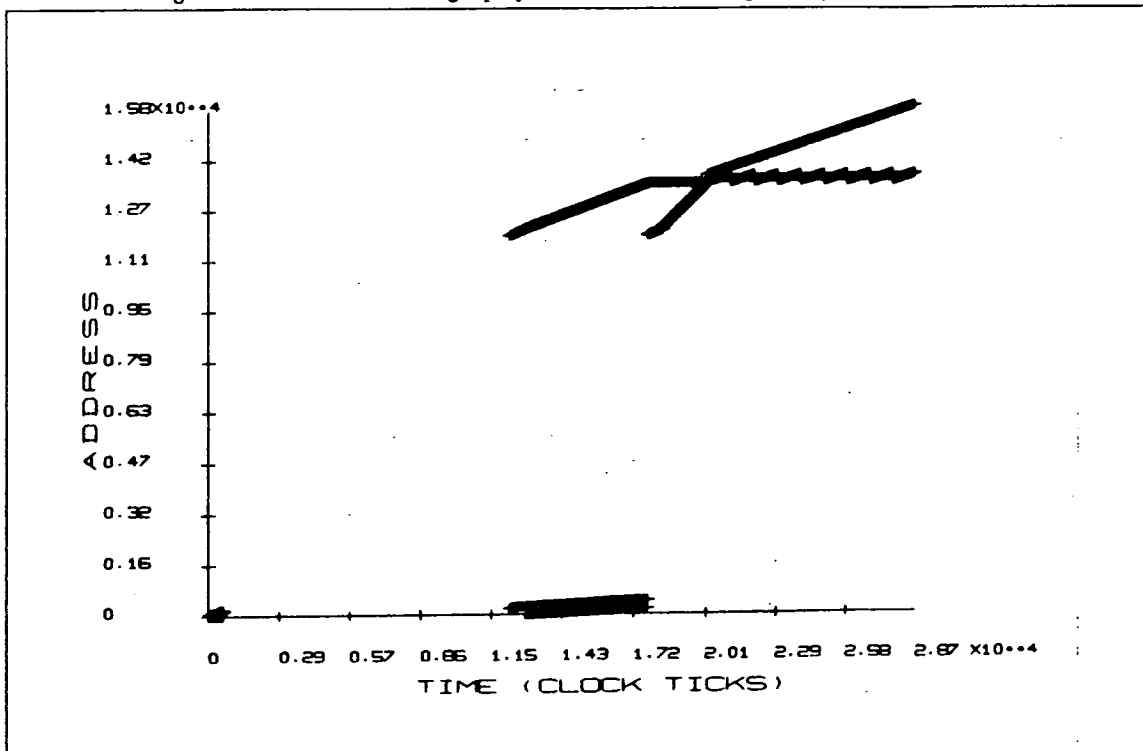


Figure 7.28 Address/time graph for Read address sequence for final example.

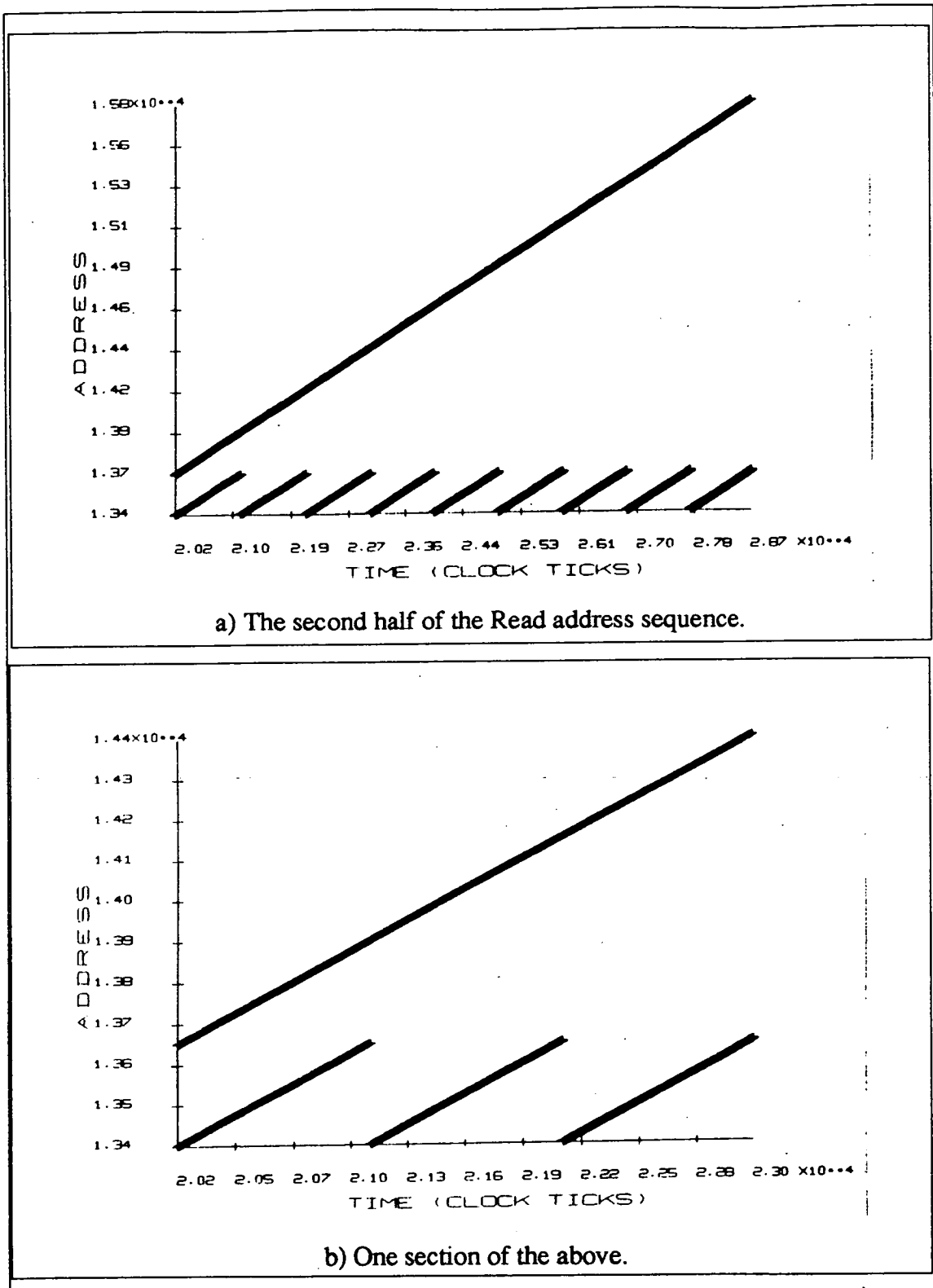


Figure 7.29 Closer examination of small section of Read address sequence for final example

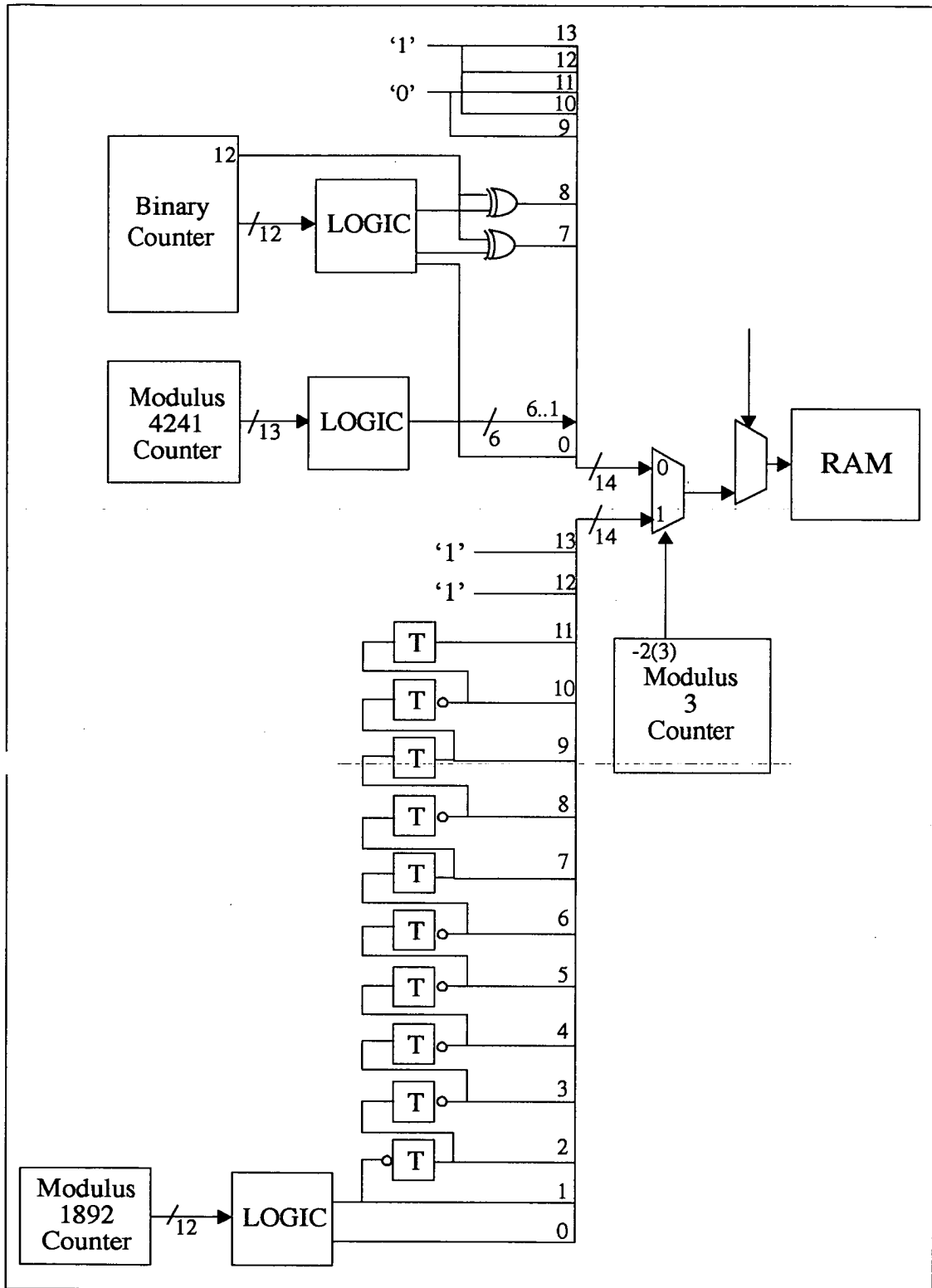


Figure 7.30 Address generation architecture for section of Read address sequence.

## 7.6 ADA - A big step

The reason for writing AG2 in ADA was one more of necessity than of choice. It was originally intended to form part of the SAGE toolset, also written in ADA and it still uses many common items in the SAGE data model.

To learn the necessary features of ADA takes weeks and it is not at all an easy matter to get even the simplest of programs running. Once the hurdles of library, family, package and procedural definition have been crossed however, ADA becomes a boon to the programmer, with its user-defined types, generic programs and overloading facilities. It is fair to say that the coding was done more efficiently in ADA than in C, after a respectable learning curve.

## 7.7 Comments

The AG2 synthesis tool is by no means a complete system. There is no guarantee that it will find a better solution than any other system for a given address sequence. AG2 is envisaged as a preprocessing stage before the more common logic synthesis stage, through which any deterministic sequence should be passed in an *attempt* to find a cheaper solution than would normally be produced.

The data structure in AG2 was designed to support memory synthesis as well as address generator synthesis and could form the basis for a more complete tool. A standard input format has been described along with interfaces to two other tools - MC<sup>2</sup> for data path synthesis and scheduled memory access sequence specification, and AG1 for its logic synthesis stage. Much of the method described here may be extended to be more clever or more complete in the recognition and synthesis of other forms of address generator, and the optimisation stage especially requires a higher degree of complexity which was not implemented in this work.

Several examples have been presented which prove the power of the tool in certain situations and also prove its applicability to scheduled memory address generation - a field previously combined with general controller synthesis. The run-times for the examples are quite satisfactory for real-time design, and so could be of help in an iterative, interactive CAD environment, for costing partial designs.

## 8 Address generator synthesis as part of a general behavioural synthesis toolset

### 8.1 Introduction to SAGE - Concepts and Reality

The approach taken in the SAGE design tool [34] was to provide a human designer with extensive design assistance in real-time VLSI architectural synthesis, but to leave the designer with overall control of the design process. This was to be done with no architectural constraints, other than that continuous-time analogue circuits would not be an option. Design-for-correctness was also proposed, to reduce simulation costs at a later date, and the whole process was to be interactive, with cost functions produced by the system as feedback to the human operator. This would encourage the designer to iteratively refine an initial architecture, devised by the expert's experience, with proposed design times of between five and twenty engineer-days for a 100,000 transistor ASIC. The process-independent output from the SAGE tool would then be passed to existing logic and layout synthesis tools for a final design specification, and all stages of its design would be automatically documented to help explain the design strategies followed. The proposed SAGE design environment is shown in Figure 8.1.

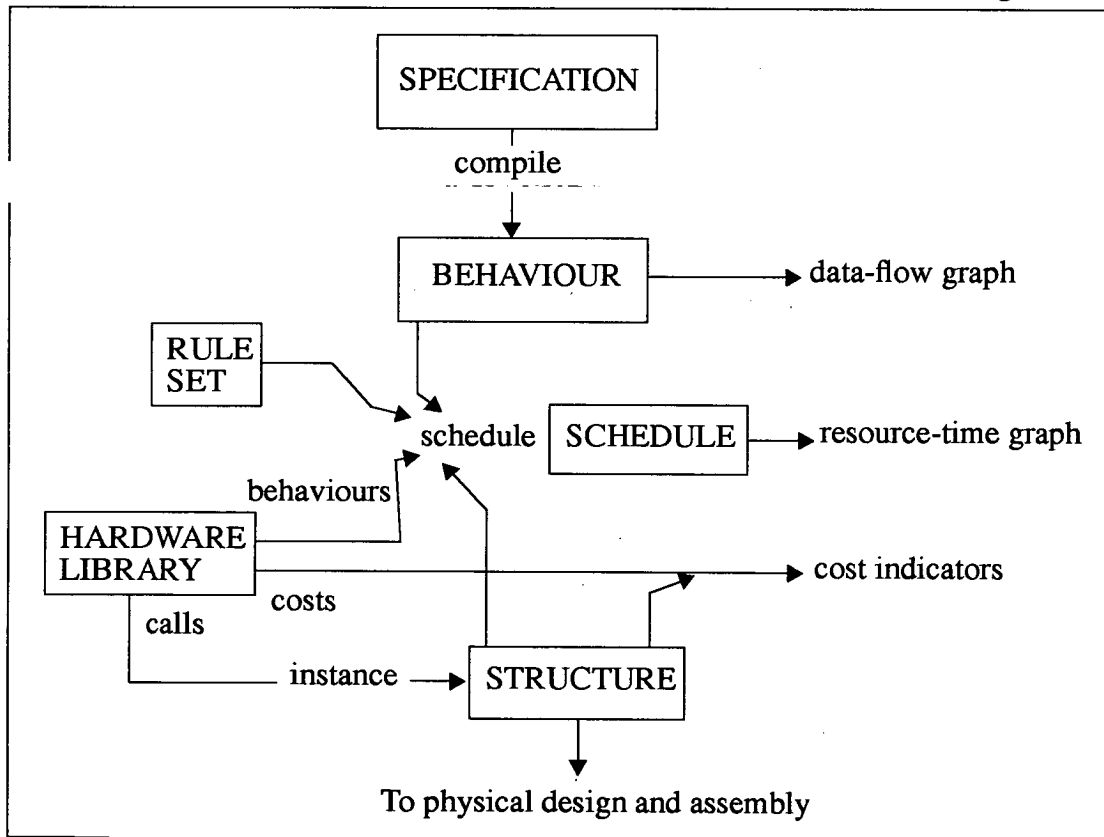


Figure 8.1 The Proposed SAGE Design Environment.



An important feature of the system is the *resource-time* graph representation of the design space, and an example is given in Figure 8.2 for the data-flow graph shown.

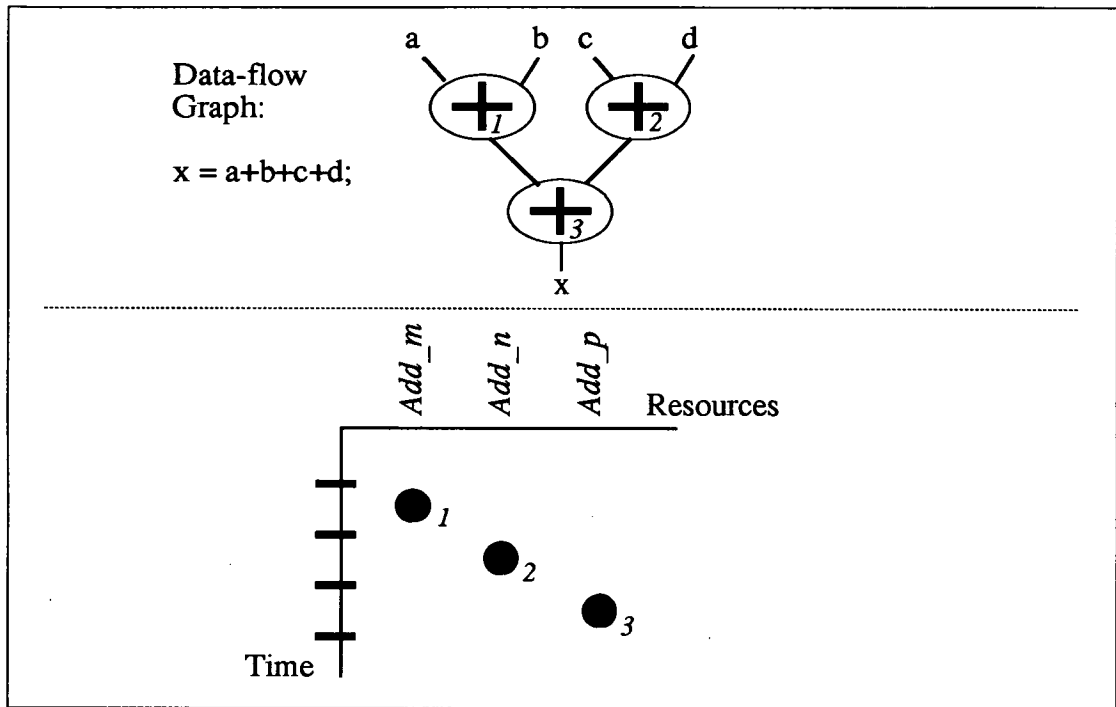


Figure 8.2 Resource-time graph for a three-adds example.

This resource-time graph represents a maximal hardware solution with no operation concurrency, and the human designer would be expected to compress the graph in time and resource numbers to realise a more optimal solution using smart, global cost functions to guide the process. This is illustrated in Figure 8.3 for the three-add example.

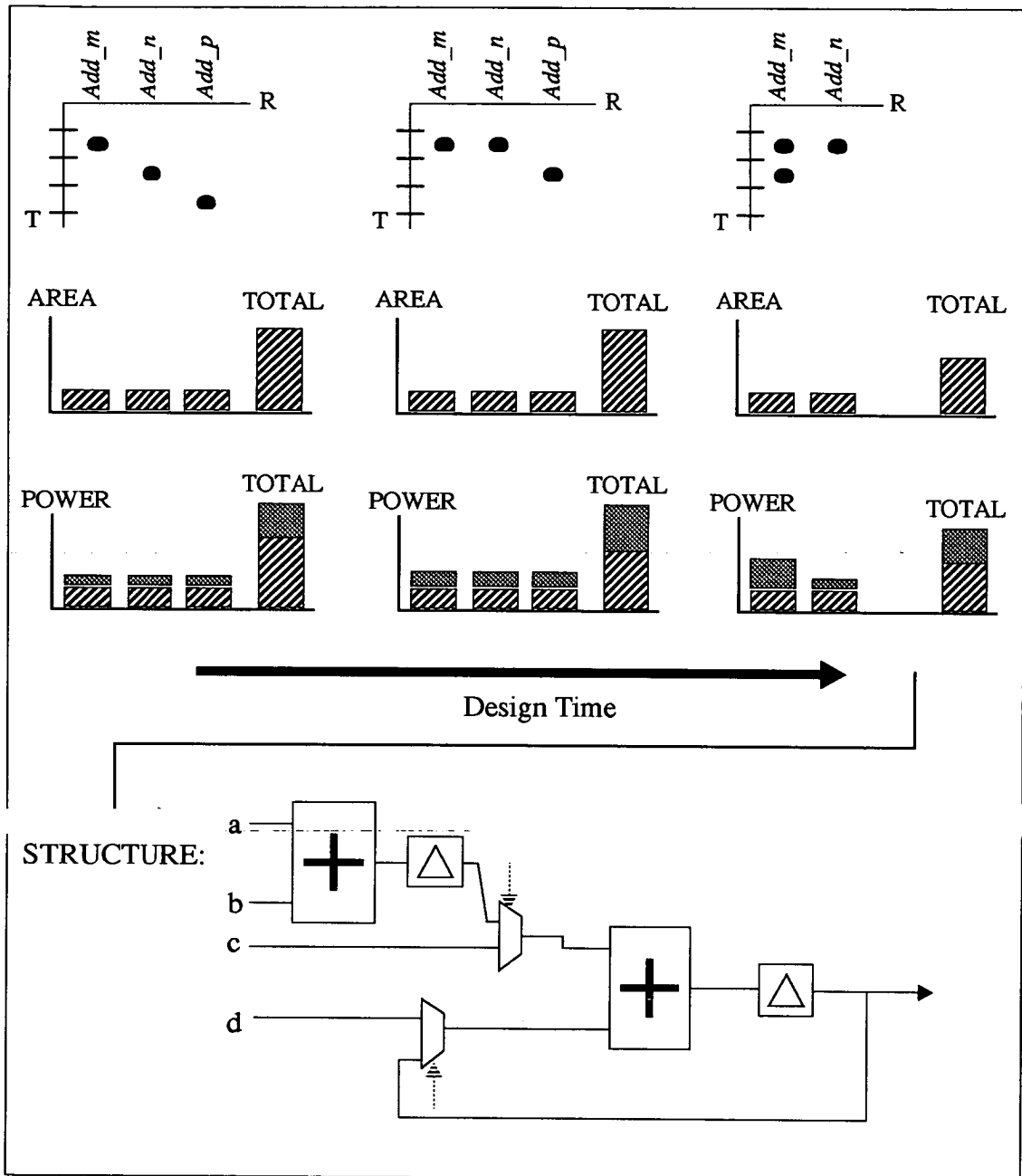


Figure 8.3 Manipulation of the resource-time space.

There is one major problem with the resource-time graph, in that for a design of any appreciable size, the graph is too cluttered with information to make much sense, and large designs must therefore be partitioned, localising any optimisation stage. Unfortunately, the proposed flexibility for SAGE introduces very complex problems in supporting this hierarchical design approach, which could not all be tackled within the

lifetime of the project. So the SAGE tool remains for now a quite comprehensive template for a general behavioural synthesis system, capable of most of the sub-tasks proposed, but lacking the clever design techniques necessary in producing more optimal data paths in individual designs.

## **8.2 Address generation within SAGE**

The address generator synthesis tool previously described has not yet been fully integrated with SAGE. Steps were taken to at least provide some sort of address generator synthesis in SAGE, and these are dealt with in the following sections.

### **8.2.1 Scheduled memory**

The detection of scheduled memory requirements and subsequent address generation requirements is not a simple matter within the SAGE data model. Very simple address generators, actually constant generators, will have already been placed in the structural and behavioural descriptions by the scheduled memory synthesis stage [147], and these must be replaced with calls to a single address generator for each (1-port) memory.

As a compromise to time, with AG2's functionality unavailable, a ROM-based address generation solution is the default, and this would probably be chosen anyway, for the short, rather random sequences. A gated-clock counter of the correct length is also specified, to generate the ROM addresses, and like the ROM, is created automatically (See Section 8.2.3). Figure 8.4 below shows the changes made to the behavioural and structural models as a result of this rather crude address generator synthesis.

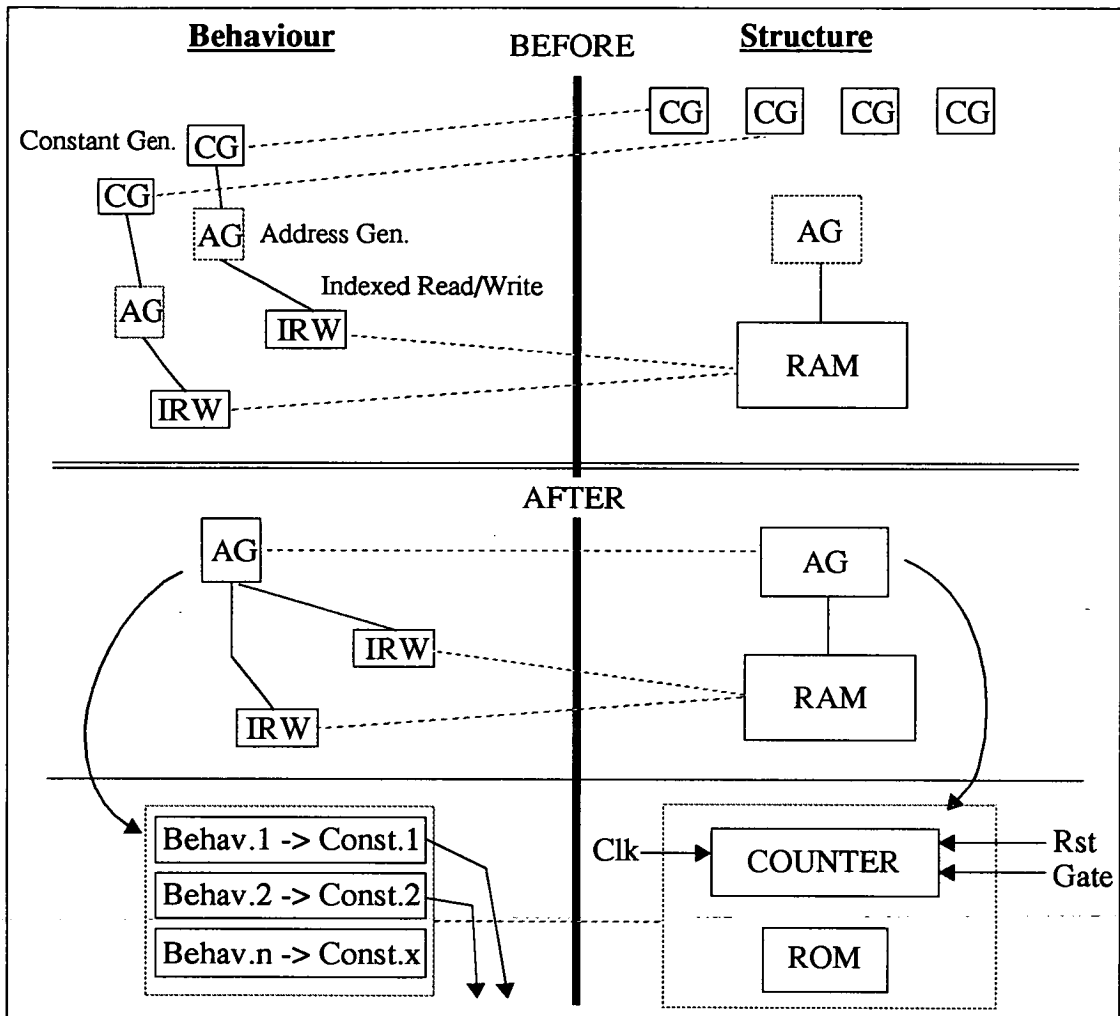


Figure 8.4 Effects of simplified address generator synthesis in SAGE.

A ROM has a single structure, of course, but must have a *separate behaviour* for each *datum* stored in it, in order to distinguish those contents from one another.

## 8.2.2 Array memory

Array memory addressing is at present handled by the VTIP front-end to SAGE, which specifies the address generator directly from the VHDL description of its requirements. The loops, or otherwise, are considered as part of the main process, and have counters, adders, comparators, etc. specified as required, as illustrated in Figure 8.5. It is possible that the VHDL description of the address generator could be

simulated, to generate the actual address sequence, to be handed to AG2, but this has not yet been attempted.

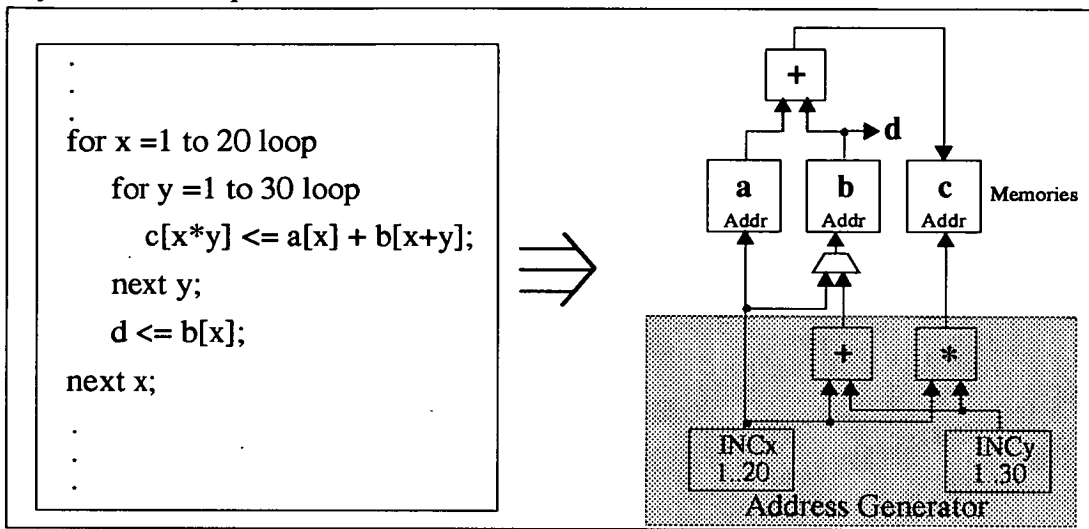


Figure 8.5 Address generator synthesis by SAGE front-end.

### 8.2.3 Macro-generation of counters

As counters play an integral part in most address generators, it makes sense to allow a macro-based approach to their construction within SAGE. The coding of the macro-generation tool proved invaluable experience in dealing with the SAGE data model.

Three different specialised counter architectures are possible: A preset modulus version where the modulus of the counter is hardwired; A parametrisable modulus counter with a port dedicated to supplying the modulus; A binary counter specifically designed for the controller synthesis stage in SAGE, with *STOP* and *CLEARBAR* ports.

In order to create these specialised counters architectures, two further macro-generators were developed. The first produces an n-input AND gate from two-input gates, in such a way that the minimum delay through the net is achieved, and the second uses this facility to produce the core counters which will be placed in the specialised architectures.

Four different counter variants are provided for: Serial carry; Parallel carry; Ripple carry; Strobed ripple carry. The two ripple carry counter types have a parallel carry for every four counter bits, reducing possible skew to a manageable level.

### **8.3 Future plans**

There are no plans at the moment to fully integrate AG2 with SAGE. This is not to say that it would be an impossible task, since the same address generation data structure is used in both, and the input/output formats are quite amenable to full integration, but simply to say that time has not allowed it.

### **8.4 Comments**

The problem of developing a general design system is at the outset one of enormous proportions. Developing a coherent data structure is the first task, but trouble appears without careful definition of the interfaces between parts of the system. The user interface can then be drawn up, so that the testing of tools is supported from the start. Only then should the coding of each tool commence, with each programmer providing example output from his own allocation of work, so that interfaces can be matched. To hand-code an entire data model for test purposes requires a second user interface, which should be developed alongside the data structure, before the actual system visuals are required.

## 9 Conclusions and new directions

### 9.1 Conclusions

In this thesis we have tried to preset a coherent explanation of address generation as a high level synthesis step. Starting with the definition of the address sequences targeted and the advantages of biasing synthesis towards the use of counters, we gave an example of just what can be achieved by this. Some possible architectures for address generators have been described, including the introduction to the modulus  $m$  counter which can play an important part in sequence generation.

Then the various requirements for address sequences were investigated, including a short description of data-dependent addressing. The areas of scheduled and array memory access were proposed as candidates for synthesis, as well as any deterministic control sequences, before the introduction to AG1 - A synthesis tool targeting the use of binary counters. It was found that by iteratively bisecting the individual bit sequences of an address sequence  $2^n$  words long, the possible use of binary counter bits in their generation could be recognised. At this stage it also became necessary to provide a graphical entry method by which memory access patterns could be manually defined in the minimum of time. Several real and possibly-real examples were given and their solutions found using AG1 were shown to exactly match those produced manually, with orders of magnitude reduction in design time. Putting this performance into practice in a much more general synthesis environment was later to prove no simple matter, but the experience gained in designing AG1 was to be invaluable.

To approach the problem of less regular access patterns and even short term or foreground memory management required a deep investigation into automatic recognition of the use of non-binary counters, but before any such system could be tested on real examples, those examples had first to be specified. Synthesising the short-term memory is normally the last stage in high level synthesis, apart from controller synthesis perhaps, and no scheduled memory address sequences were available in the literature, so that a diversion of attention was necessary to further investigate the field of high level synthesis, in the hope of constructing a very simple foreground memory synthesis tool to produce such example sequences.

After the results of that investigation were presented, there followed a description of MC<sup>2</sup>, a mainly rule-based data path synthesis suite, which can produce from an operation-priority schedule and some resource allocation information, a data path in the form of a netlist, including registers and register files, multiplexers and ROMs, whose

address and control sequences are made available to allow their generation to be investigated. MC<sup>2</sup> was shown to apply both well known and novel techniques in a pragmatic manner and to produce as a *side effect of memory synthesis*, data paths to rival those given in the literature. The memory synthesis-first approach may be a little wasteful in memory space but to the benefit of control and communications complexity. Comparisons with the literature were made with some difficulty due to the diversity of reported information, and so the outline for a standard format for reporting data path synthesis results was presented.

Next the final, main piece of work was discussed. A general address generator synthesis tool - AG2 - was presented which built on the experience of AG1 to develop ways of recognising non-binary counter-produced sequences, and was shown to allow the design of a whole system's address generation layer. This was optimised as globally as possible given the short run times required in the iterative behavioural synthesis system - SAGE - in which AG2 was envisaged to play a part.

Based on an extension of the method of iterative bisection of bit sequences, novel algorithms in AG2 control the recognition and global optimisation of sequence generators, in turn based on both binary and non-binary counters as well as several other possible address generator elements. To facilitate all this, a coherent and comprehensive data model was presented along with a textual user interface, with which both AG1 and MC<sup>2</sup> are compatible.

The three main stages of the synthesis process were all described and backed up by several examples of their use. Firstly the development of possible solutions was described. This depended mainly on the user to provide some hint as to the modulus of counter to be used if possible, but automatic methods were described which find this hint on a bitwise basis within the address word. Next, a simple transformation was applied which allowed iterative bisection techniques to work on sequences generated by non-binary counter bits. Explanations of this and other generator-recognition techniques were presented using a working example, which was construed to show most of the features of the AG2, rather than to be representative of real problems. Finally the area-cost based optimisation of the address generators was expounded, before the results of synthesis for several examples were presented. AG2's results are simple to interpret and were shown to be of excellent quality when compared to manual designs. It remains to be seen if AG2 is any better than some other synthesis tools, but it is felt that good advances have been made in the area of address generator synthesis.



As high level synthesis moves into its second decade and it struggles to keep up with the demand for more intelligent ASIC synthesis, the field is fragmenting as each new problem appears. It is unreasonable now to think that all of a designer's knowledge may be incorporated in the automata, or that every problem is foreseeable, and we are sure that as ASIC design progresses into behavioural synthesis, new problems will be encountered. However, if a pragmatic, hands-on approach is taken, as in this thesis, then it may soon be a reasonable thought after all.

The main problem in developing useful tools in the past has been a lack of consistency in design representation. VHDL aims to change that, with the authority of the I.E.E.E. and the N. American D.of D. behind it, and already CASE tool frameworks are being marketed to provide a systems-level design environment into which tools like AG2 may be designed to fit. It is possible that such frameworks may be seen as overly constrictive on future design plans and it may be some time before a framework is produced to be acceptable to all, in the knowledge that the industrials would then have to play by the same rules. There might then be a return to the competitive, secretive, in-house coding that we have seen in the past, perhaps to the detriment of technical advancement.

On a different note, many of the task in high level synthesis would benefit hugely from a multi-processor environment, and configurable processor arrays coupled with iterative synthesis tools could be used to provide instant simulation and feedback on system dynamics and design constraints. An array of processors may be programmed to act like the architecture defined by the synthesis tool and fundamental to this technique would be the coherent interface between the hardware and software. For most tasks in high level synthesis a depth-first approach is best - You don't know the best solution until you have tried them all - and hardware simulation should take the sting out of this infuriating truth.

## **9.2 Future Plans**

Further work to be undertaken on AG1 might be an improved, mouse-driven graphical interface for the definition of memory access patterns, and in MC<sup>2</sup> a graphical rather than textual description of the synthesised data path would be better.

For AG2 there is a need for another graphical interface for the analysis of long, complex address sequences, and other improvements might include the exploration of ROM contents to allow possible optimisation, automatic array memory optimisation by

the methods described in Section 4.4, and an output format compatible with the SAGE tool or even in VHDL or EDIF.

Other plans include the use of AG2 and its data structure as the basis for another project on memory synthesis in general. There is also a motive for an investigation into memory-based scheduling: Instead of scheduling computational operations onto a smaller set of resources, the chronological constraints on the order of operations - in effect their storage requirements - are mapped onto a given, but possible flexible memory architecture.

## References

### Data-Path Synth.

- 1 Küçükçakar, K. and Parker, A.C., *MABAL: A software package for Module And Bus ALlocation*, Tech. Report No. CRI-88-61, University of Southern California, 1989.
- 2 Küçükçakar, K. and Parker, A.C., *Data Path Design Tradeoffs Using MABAL*, Tech. Report No. CENG 89-21, U. of S.C., 1989.
- 3 Stok, L., *Interconnect Optimisation During Data Path Allocation*, Proc. European DAC '90, pp. 141-145.
- 4 Stok, L., *Synthesis and Optimisation of Architectures for Digital Systems*, PhD Thesis, Eindhoven University of Technology, March 1991.
- 5 McFarland, M. et al, *Tools for Architecture Synthesis*, IFIP Workshop on Fast Prototyping of VLSI, Elsevier Publ., North-Holland, 1987.
- 6 Berry, N. and Pangrle, B.M., *SCHALLOC: An Algorithm for Simultaneous Scheduling and Connectivity Binding in a Data Path Synthesis System*, Proc. European DAC '90, pp. 78-82.
- 7 Grass, W., *A Branch-and Bound Method for Optimal Transformation of Data Flow Graphs for Observing Hardware Constraints*, Proc. European DAC '90, pp. 73-77.
- 8 Pangrle, B.M. and Gajski, D.D., *Slicer: A state synthesiser for intelligent silicon compilation*, Proc. ICCAD '87, pp. 42-45.
- 9 Pangrle, B.M., *Splicer: A Heuristic Approach to Connectivity Binding*, Proc. 25th IEEE DAC, 1988, pp. 536-541.
- 10 Parker, A.C. et al, *The CMU design automation system: An example of automated data path design*, Proc. 16th DAC, June 1979.
- 11 Thomas, D.E. et al, *Methods of automatic data path synthesis*, IEEE Computer, Vol. 6, No. 12, Dec. 1983.
- 12 Kowalski, T.J. and Thomas, D.E., *The VLSI design automation assistant: What's in a knowledge base*, Proc. 22nd DAC, 1985.

- 13 Kowalski, T.J. et al, *The VLSI Design Automatin Assistant: From Algorithms to Silicon*, IEEE Design and Test of Computers, pp. 33-42, August 1985.
- 14 Girczyc, E.F. and Knight, J.P., *An ADA to standard cell hardware compiler based on graph grammars and scheduling*, Proc. ICCD '84.
- 15 Paulin, P.G., Knight, J.P. and Girczyc, E.F., *HAL: A multi-paradigm approach to automatic data path synthesis*, Proc. 23rd DAC, 1986.
- 16 Southard, J.R., *MacPitts: An approach to silicon compilation*, IEEE Computer, Dec. 1983, pp. 74-82
- 17 Parker, A.C., Pizarro, J. and Mlinar, M., *MAHA: A program for data path synthesis*, Proc. 23rd DAC, 1986.
- 18 Park, N. and Kurdahi, F.J., *Module Assignment and Interconnect Sharing in Register-Transfer Synthesis of Pipelined Data Paths*, Proc. ICCAD '89, pp. 16-19.
- 19 Haroun, B.S. and Elmasry, M.I., *Architectural Synthesis for DSP Silicon Compilers*, IEEE Trans. CAD, Vol. 8, No. 4, 1989, pp. 431-447.
- 20 Stok, L. and van der Born, R., *EASY: Multiprocessor Architecture Optimisation*, Proc. Int'l Workshop on Logic and Architecture Synthesis for Silicon Compilers, Ed. P.M.McLellan, pp. 313-328, Grenoble, May 1988.
- 21 Camposano, R. and Rosenstiel, W., *Synthesising Circuits From Behavioural Descriptions*, IEEE Trans. CAD, Vol. 8, No. 2, Feb 1989, pp. 171-180.
- 22 Camposano, R. and Tablet, R.M., *Design Representation for the Synthesis of Behavioural VHDL models*, Proc. 9th Int'l Conf. Comp. HDLs, May 1989.
- 23 Rosenstiel, W., *CADDY: The Karlsruhe Behavioural Synthesis System*, IEEE High-Level Synthesis Workshop, Orcas Island, Wa., Jan. 1988.
- 24 Krämer, H. and Rosenstiel, W., *System Synthesis using Behavioural Descriptions*, Proc. European DAC '90, pp. 277-282.
- 25 Balakrishnan, M. and Marwedel, P., *Integrated Scheduling and Binding: A Synthesis Approach for Design Space Exploration*, Proc. 26th DAC, pp. 68-74.
- 26 Marwedel, P., *The MIMOLA Design System: Tools for the Design of Digital Processors*, Proc. 23rd DAC, pp. 587-593.

- 27 Gajski, D. and Kuhn, R., *Guest Editor's Introduction: New VLSI Tools*, IEEE Computer, Vol. 16, No. 12, pp. 11-14.
- 28 Gajski, D. (Ed.), *Silicon Compilation*, Addison-Wesley, 1988.
- 29 Tseng, C.-J. and Siewiorek, D.P., *Automated Synthesis of Data Paths in Digital Systems*, IEEE Trans. CAD, Vol. 5, No. 3, pp. 379-395.
- 30 Bergstraesser, T. et al, *SMART: Tools and Methods for Synthesis of VLSI Chips with Processor Architecture*, Proc. 25th DAC, 1988, pp.654-657.
- 31 McFarland, M.C., Parker, A.C. and Camposano, R., *Tutorial on High-Level Synthesis*, Proc. 25th DAC, 1988, pp. 330-336.
- 32 Bergamaschi, R.A. and Camposano, R., *Synthesis using Path-Based Scheduling: Algorithms and Exercises*, Proc. 27th DAC 1990, pp. 450-455.
- 33 Bergamaschi, R.A. and Allerton, D.J., *A Graph-Based Silicon Compiler for Concurrent VLSI Systems*, Proc. IEEE CompEuro '88, Brussels, pp. 36-47.
- 34 Denyer, P.B. et al, *An Approach to the Synthesis of VLSI Systems from Behavioural Specifications*, SARI Internal report, No. SARI-001-B, Dec. 1987.
- 35 Trickey, H., *Flamel: A High-Level Hardware Compiler*, IEEE Trans. CAD, Vol. 6, No. 2, pp. 259-269.
- 36 Hou, P.-P., Owens, R.M. and Irwin, M.J., *DECOMPOSER: A Synthesiser for Systolic Systems*, Proc. 25th DAC, 1988, pp. 650-653.
- 37 Jerraya, A. et al, *Principles of the SYCO Compiler*, Proc. 23rd DAC, 1986.
- 38 Brewer, F.D. and Gajski, D.D., *Knowledge Based Control in Micro-Architecture Design*, Proc. 24th DAC, 1987, pp. 203-209.
- 39 Park, N. and Parker, A.C., *Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications*, IEEE Trans. CAD, Vol. 7, No. 3, pp. 356-370.
- 40 Blackman, T., Fox, J. and Roseburgh, C., *The SILC Silicon Compiler: Language and Features*, Proc. 22nd DAC, 1985, pp. 232-237.
- 41 Paulin, P.G. and Knight, J.P., *Force-Directed Scheduling for the Behavioural Synthesis of ASIC's*, IEEE Trans. CAD, Vol. 8, No. 6, 1989.

- 42 Paulin, P.G. and Knight, J.P., *Force-Directed Scheduling in Automated Data Path Synthesis*, Proc. 24th DAC, pp. 195-202.
- 43 Hwang, K et al, *Scheduling and Hardware Sharing in Pipelined Data Paths*, Proc. ICCAD '89, pp. 24-27.
- 44 Mallon, D.J. et al, *A New Approach to Pipeline Optimisation*, Proc. EDAC '90, pp. 83-87.
- 45 Potkonjak, M. and Rabaey, J., *A Scheduling and Resource Allocation Algorithm for Hierarchical Signal Flow Graphs*, Proc. 26th DAC, pp. 7-12.
- 46 Yassa, F. et al, *A Silicon Compiler for Digital Signal Processing: Methodology, Implementation and Application*, Proc. IEEE, Vol. 75, No. 9, pp. 1273-1282.
- 47 Denyer, P. and Renshaw, D., *VLSI Processing: A Bit Serial Approach*, Reading, Mass., Addison-Wesley, 1985.
- 48 Hartley, R. and Jasica, J., *Behavioural to Structural Translation in a Bit-Serial Compiler*, IEEE Trans. CAD, Vol. 7, No. 8, pp. 877-886.
- 49 Cheung, Y.S. and Leung, S.C., *A second generation compiler for bit-serial signal processing architecture*, Proc. IEEE CASSP '87, pp. 487-490.
- 50 Grant, D.M. and Denyer, P.B., *Memory, Control and Communications Synthesis for Scheduled Algorithms*, Proc 27th DAC, 1990, pp162-167.
- 51 Neil, J.P. and Denyer, P.B., *Exploring Design Space Using SAVAGE: A Simulated Annealing based VLSI Architecture GEnerator*, Proc. 33rd Mid-west Symp. on Circuits and Systems, Calgary, Aug., 1990.
- 52 Chu, C.M. et al, *HYPER: An Interactive Synthesis Environment for High Performance Real Time Applications*, Proc. Int'l Conf. on Computers, 1989, pp. 432-435.
- 53 Brayton, R.K. et al, *The Yorktown Silicon Compiler System*, Silicon Compilation, Ed. Gajski, D.D., pp 204-310, [28].
- 54 Berstis, V., *The V Compiler: Automating Hardware Design*, IEEE Desifn and Test of Computers, pp. 8-17, April 1989.
- 55 Woo, N.S., *SAM: A Data Path Allocation System*, Proc. IEEE CICC 1990, pp. 14.4.1 - 14.4.3.

- 56 Woo, N.S., *A Global, Dynamic Register Allocation and Binding for a Data Path Synthesis System*, Proc. 27th DAC, pp. 505-510.
- 57 Kumar, A.A. and Balakrishnan, M., *A Novel Integrated Scheduling and Allocation Algorithm for Data Path Synthesis*, Proc. 4th CSI/IEEE Int'l. Symp. on VLSI Design, New Delhi, Jan. 1991, pp. 212-218.
- 58 Lee, J.H., Hsu, Y.C. and Lin, Y.L., *A New Integer Linear Programming Formulation for the Scheduling Problem in Data Path Synthesis*, Proc. ICCAD '89, pp. 20-23.
- 59 Papachristou, C.A. and Konuk, H., *A Linear Program Driven Scheduling and Allocation Method*, Proc. 27th DAC 1990, pp. 77-83.
- 60 Gebotys, C.H. and Elmasry, M.I., *A Global Optimisation Approach for Architectural Synthesis*, Proc. ICCAD '90, pp. 258-261.
- 61 Lippens, P.E.R. et al., *PHIDEO: A Silicon Compiler for High Speed Algorithms*, Proc. EDAC '91, pp. 436-441.

### **Logic Synthesis**

- 62 Devadas, S. et al, *MUSTANG: State Assignment of Finite State Machines for Optimal Multi-Level Logic Implementation*, Proc. ICCAD '87, pp. 16-19.
- 63 Rosenstiel, W. and Schmid, D., *Logic Synthesis*, Advances in CAD for VLSI, Vol. 2, North-Holland, 1986, p. 37.
- 64 Devadas, S. et al, *MUSTANG: State Assignment of Finite State Machines Targeting Multilevel Logic Implementations*, IEEE Trans. CAD, Vol. 7, No. 12, pp. 1290-1299.
- 65 Amman, R. and Baitinger, U.G., *New State Assignment Algorithms for Finite State Machines using Counters and Multiple-PLA/ROM Structures*, Proc. ICCAD '87, pp. 20-23.
- 66 Amman, R. and Baitinger, U.G., *Optimal State Chains and State Codes in Finite State Machines*, IEEE Trans. CAD, Vol.8, No. 2, Feb. 1989, pp.153-170.
- 67 Brayton, R.K., Sentovich, E.M. and Somenzi, F., *Don't Cares and Global Flow Analysis of Boolean Networks*, Proc. ICCAD '88, pp. 98-101.

- 68 Gurunath, B. and Biswas, N.N., *An Algorithm for Multiple Output Minimisation*, IEEE Trans. CAD, Vol. 8, No. 9, Sept. 1989, pp 1007-1013.
- 69 Harada, Takashi, *Research Memo: Prolog based Logic Synthesis System*, University of Edinburgh, Dept. of E.E., July 1988.
- 70 Wei, R.-S., Rothweiler, S. and Jou, J.-Y., *BECOME: Behavior Level Circuit Synthesis Based on Structure Mapping*, Proc. 25th DAC, 1988, pp. 409-414.
- 71 Pitty, S. et al, *Syntactic Translation And Logic Synthesis In GATEMAP*, Plessey Research, October 1987.

### **Controller Synth**

- 72 Mhaya, N. and Jerraya, A.A., *CPC: A Control Section Synthesiser*, Proc. 8th International Custom Microelectronics Conference, 1988, pp. 22.0-22.7
  - 73 Brayton, R.K. et al, *A microprocessor design using the Yorktown silicon compiler*, Proc. ICCD '85.
  - 74 Camposano, R., *Structural synthesis in the Yorktown silicon compiler*, Proc. VLSI '87.
  - 75 Fox, J.R. and Fried, J.A., *Telecommunication circuit design using the SILC silicon compiler*, Proc. ICCD '85, pp.213-219.
- 
- 76 Joepen, H. and Glesner, M., *Architecture construction for a general silicon compiler system*, Proc. ICCD '85.
  - 77 Krekelberg, D.E., Sobelman, G.G. and Jhon, C.S., *Yet another silicon compiler*, Proc. DAC '85, pp. 176-182.
  - 78 Peng, Z., *Synthesis for VLSI systems with the CAMAD design aid*, Proc. DAC '86.
  - 79 Siskind, J.F., Southard, J.R. and Crouch, K.W., *Generation of custom high performance VLSI design from succinct algorithmic description*, Proc. 1982 Conf. on Advanced Research in VLSI, MIT, pp. 28-39.
  - 80 Southard, J.R., *MacPitts: An approach to silicon compilation*, IEEE Computer, Dec. 1983, pp. 74-82
  - 81 Zegers, J. et al, *CGE: Automatic Generation of Controllers in the CATHE-DRAL-II Silicon Compiler*, Proc. European DAC '90, pp. 617-612.



- 82 Nagle, A.W., Cloutier, R. and Parker, A.C., *Synthesis of Hardware for the Control of Digital Systems*, IEEE Trans. CAD of ICs and Systems, Vol. 1, No. 4, pp. 201-211.
- 83 Spaanenburg, L., *Structured Design of Control Specifications*, Advances in CAD for VLSI, Vol. 2, North-Holland, 1986, pp. 53-92.
- 84 Grass, W. and Lipp, H.-M., *LOGE - A highly effective system for logic design automation*, ACM SIGMA newsletter 9, No. 2, 1979.
- 85 Grass, W., Biehl, G. and Hall, S., *LOGE-MAT, a program for the synthesis of microprogrammed controllers*, Proc. CAD '80, Brighton, pp. 543-558.

### Quotes

- 86 De Man, H., "Efficient design synthesis is only possible when targeted to one particular architecture," De Man, H. et al, *A unified toolbox of CAD tools for the design of dedicated signal processing chips*, IEEE International Conf. on Computer Design: VLSI in Computers, ICCD '84, pp. 838-844.
- 87 Zimmerman, G., "... the top-down design of complex digital systems for VLSI implementations is possible and is capable of yielding much better results than bottom-up methods.", *Top-Down Design of Digital Systems*, Advances in CAD for VLSI, Vol. 2, North-Holland, 1986, p. 28.

### Test

- 88 Maly, W., Nag, P.K. and Nigh, P., *Testing Oriented Analysis of CMOS ICs with Opens*, Proc. ICCAD '88, pp. 344-347.
- 89 Goldstein, L.H., *Controllability/Observability Analysis of Digital Circuits*, IEEE Trans. Circuits and Systems, Vol. CAS-26, No. 9.
- 90 Goldstein, L.H. and Thigpen, E.L., *SCOAP: Sandia Controllability/Observability Analysis Program*, Proc. 17th DAC, 1980, pp. 190-196.
- 91 Catthoor, F. et al, *A Testability Strategy for Multiprocessor Architecture*, IEEE Design and Test of Computers, Apr. 1989, pp. 18-34.
- 92 Sutlieff, C., *Testing Time for ASIC's*, IEE Review, Jan. 1991, pp. 27-31.

## **CAD Issues**

- 93 Lang, M.H. and McCormick, P.E., *Hierarchical Design Methodologies: A VLSI Necessity*, Advances in CAD for VLSI, Vol. 6, North-Holland, 1986, pp. 123-149.
- 94 Tutorial on Parallel Processing, presented at DAC '90, Orlando, FL, June 1990.

## **CAD Systems**

- 95 Rosenstiel, W. and Schmid, D., *Logic Synthesis*, Advances in CAD for VLSI, Vol. 2, North-Holland, 1986, p. 31.
- 96 Weste, N. and Eshraghian, K., *Principles of CMOS VLSI Design, A Systems Perspective*, Reading, Mass., Addison-Wesley, 1985, p. 241.
- 97 Wecker, T., *Semi-Custom Design Systems*, Advances in CAD for VLSI, Vol. 2, North-Holland, 1986, pp. 195-228.
- 98 Koike, N. and Ohmori, K., *Design Automation Machine*, Advances in CAD for VLSI, Vol. 6, North-Holland, 1986, pp. 465-499.
- 99 Shiva, S.G., *Automated Hardware Synthesis*, Proc. IEEE, Vol. 71, No. 1, Jan. 1983, pp. 76-87.
- 100 Burrows, D.F., *SHADE: Plessey's structured hardware design environment*, Proc. 3rd Silicon Design Conference, 1986, pp. 105-114.
- 101 Goossens, G., *An efficient microcode-compiler for custom DSP-processors*, Proc. ICCAD '87, pp. 24-27.
- 102 SOLO 1400 Reference Manual (Overview), ES2 Publications Unit, January 1990.
- 103 De Man, H. et al, *CATHEDRAL II: a silicon compiler for digital signal processing*, IEEE Design and Test, Dec. 1986, pp. 13-25.
- 104 De Man, H. et al., *Cathedral II, A Computer-Aided Synthesis System for Digital Signal Processing VLSI systems*, IEE CAE Journal, April 1988, pp55-66.
- 105 Vanhoof, J., Rabaey, J. and De Man, H., *A Knowledge-Based CAD System for Synthesis of Multi-processor Digital Signal Processing Chips*, Proc. VLSI '87, Sequin, C.H. (Ed), Elsevier, New York, 1988, pp. 73-88.

- 106 De Man et al, *Architecture-Driven Synthesis Techniques for VLSI Implementation of DSP Algorithms*, Proc. IEEE, Vol. 78, No. 2, pp. 319-335, Feb. 1990.
- 107 Goossens, G. et al, *Optimisation-based synthesis of multiprocessor chips for digital signal processing, with Cathedral-II*, Proc. Int Wkshp. on Logic and Architecture Synthesis for Silicon Compilers, 1988.
- 108 Denyer, P.B., Renshaw, D.A. and Bergmann, N.W., *A silicon compiler for VLSI signal processors*, Proc. ESSCIRC 1982, pp. 215-218.
- 109 Siskind, J.F., Southard, J.R. and Crouch, K.W., *Generation of custom high performance VLSI design from succinct algorithmic description*, Proc. 1982 Conf. on Advanced Research in VLSI, MIT, pp. 28-39.
- 110 Saunders, J.E., *ELLA - A toolset for system designers*, Proc. 8th International Custom Microelectronics Conference, 1988, pp. 57.0-57.7.
- 111 Koelmans, A.M., McLauchlan, M.R. and Robson, A.P., *The STRICT language and design methodology*, Proc. 1987 Electronic Design Automation Conference, pp. 79-86.
- 112 Koelmans, A.M., McLauchlan, M.R. and Kinniment, D.J., *Asynchronous extensions to the STRICT High level design system*, Proc. 8th International Custom Microelectronics Conference, 1988, pp. 23.0-23.5.
- 113 Huiskens, J et al., *Efficient design of Systems on Silicon with Pyramid*, Proc. Int Wkshp. on Logic and Architecture Synthesis for Silicon Compilers, 1988, pp.299-311.
- 114 Thomas, D.E. et al, *The System Architects Workbench*, Proc. 25th DAC, pp. 337-343, 1988.
- 115 Thomas, D.E. et al, *Algorithmic and Register-Transfer Level Synthesis: The System Architects Workbench*, Kluwer Academic Publishers, Boston, 1990.
- 116 Lis, J.S. and Gajski, D.D., *Synthesis from VHDL*, Proc. IEEE ICCD 1988, pp. 378-381.
- 117 Casavant, A.E. et al, *A synthesis environment for designing DSP systems*, IEEE Design and Test of Computers, pp. 35-43, April 1989.
- 118 Tseng, C.J. et al, *Bridge: A versatile behavioural synthesis system*, Proc. 25th DAC, pp. 415-420, 1989.

- 119 De Micheli, G. and Ku, D.C., *HERCULES: a System for High Level Synthesis*, Proc. 25th DAC, pp. 483-488.
- 120 Shung, B.C. et al, *An Integrated CAD System for Algorithm-Specific IC Design*, IEEE Trans. CAD, Vol. 10, No. 4, pp. 447-463, 1991.

### **Circuit level Simulation**

- 121 Nagel, L.W., *SPICE2: A Computer Program to Simulate Semiconductor Circuits*, Memo ERL-M520, University of California at Berkley, 9th May 1975.
- 122 Nagel, L.W., *ADVICE for Circuit Simulation*, Proc. IEEE ISCS, Houston, 1980.

### **Timing Simulation**

- 123 Chawla, B.R., Gummel, H.K. and Kozak, P., *MOTIS - An MOS Timing Simulator*, IEEE Trans. Circuits and Systems, Vol. 22, No. 12, 1975, pp. 901-910.
- 124 Agrawal, V.D. et al, *Mixed Mode Simulation in the MOTIS System*, Journ. Digital Systems, 1981, p. 383.

### **Switch level Simulation**

- 125 Bryant, R.E., *MOSSIM: A Switch-Level Simulator for MOS LSI*, Proc. 18th DAC, July 1981, pp. 786-790.

### **Timing Verification**

- 126 Jouppi, N.P., *Timing analysis for nMOS VLSI*, Proc. 20th DAC, June 1983, pp. 411-418.

### **Schematic Capture**

- 127 Joobbani, R. et al, *Design Consultant: A Design Synthesis Tool to Enhance Design Productivity*, Proc. 8th International Custom Microelectronics Conference, 1988, pp. 54.0-54.7.
- 128 Bain, J., *STELLA - A Schematic Capture Tool for ELLA*, Proc. 8th International Custom Microelectronics Conference, 1988, pp. 36.0-36.7.

## **Netlist Comparison**

- 129 Ebeling, C. and Zajicek, O., *Validating VLSI Circuit Layout by Wirelist Comparison*, Proc. ICCAD '83, pp. 172-173.
- 130 Spickelmier, R.L. and Newton, A.R., *Wombat: A New Netlist Comparison Program*, Proc. ICCAD '83, pp. 170-171.

## **Layout**

- 131 Persky, G., Deutsch, D.N. and Schweikent, D.G., *LTX - A Minicomputer-based System for Automated LSI Layout*, Journ. DA and Fault Tolerant Computing, Vol. 1, No. 3, pp. 217-255.

## **Design Styles**

- 132 Tanaka, S. et al, *A Sub-Nanosecond 8K-Gate CMOS/SOS Gate Array*, Proc. ISSCC '84, pp. 260-261.
- 133 Takechi, M. et al, *A CMOS 12K Gate-Array with Flexible 10Kb Memory*, Proc. ISSCC '84, pp. 258-259.
- 134 Werner, J., *Custom IC Design in Europe*, VLSI Design, Jan. 1984, pp. 28-33.

## **BIST**

- 135 Scholz, H.N. et al, *ASIC Implementations of Boundary-Scan and BIST*, Proc. 8th International Custom Microelectronics Conference, 1988, pp. 43.0-43.9.
- 136 Chakradhar, S.T., Bushnell, M.L. and Agrawal, V.D., *Automatic Test Generation Using Neural Networks*, Proc. ICCAD '88, pp. 419-419.
- 137 Maly, W. and Nigh, P., *Built-In Current Testing - Feasability Study*, Proc. ICCAD '88, pp. 340-343.
- 138 Dandapani, R., Gulati, R.K. and Goel, D.K., *Built-In Self-Test for Large Embedded CMOS Folded PLAs*, Proc. ICCAD'88, pp. 236-239.
- 139 Koenemann, B., Mucha, J. and Zwiehoff, G., *Built-In Logic Block Observation Techniques*, Digest 1979 Test Conference, 79CH1509-9C, pp. 37-41.
- 140 McCluskey, E.J., *Logic Design Principles*, Prentice Hall International, pp. 450-455.

## **Address Generation**

- 141 Bateman, A., Bolton, M. and Reed, G., *Specification and Synthesis of VLSI Digital Communications Systems*, Proc. 8th International Custom Microelectronics Conference, 1988, pp. 31.0-31.7.
- 142 Joobbani, R. et al, *Design Consultant: A Design Synthesis Tool to Enhance Design Productivity*, Proc. 8th International Custom Microelectronics Conference, 1988, pp. 54.0-54.7.
- 143 Balakrishnan, M. et al, *Allocation of multi-port memories in data path synthesis*, Proc. ICCAD '87, pp. 266-269.
- 144 Kung, S.Y., Whitehouse, H.J. and Kailath, T. (Eds.), *VLSI and Modern Signal Processing*, Prentice-Hall, 1985, pp. 339-340.
- 145 Kung S.Y., Owen, R.E. and Nash, J.G., *VLSI Signal Processing II*, IEEE Press, 1986, pp. 238-239.
- 146 Kung S.Y., Owen, R.E. and Nash, J.G., *VLSI Signal Processing II*, IEEE Press, 1986, pp. 261-263.
- 147 SAGE 4.2 User Manual, Silicon Architectures Research Initiative, Oct. 1990, pp 3\_51 - 3\_54.
- 148 Grant, D.M., *Address generation for SAGE4*, SARI internal report, University of Edinburgh, June 1990.
- 149 Grant, D.M., Denyer, P.B. and Finlay, I., *Synthesis of Address Generators*, Proc. ICCAD '89, pp116-119.
- 150 Grant, D.M., *Sari Technical Note re: Address Generation in Sage4*, University of Edinburgh, June 1990.
- 151 Grant D.M. and Denyer, P.B., *Address Generation for Array Access Based on Modulus m Counters*, Proc. EDAC '91, pp. 118-122.
- 152 Chirlian, P.M., *Analysis and Design of Integrated Electronic Circuits*, Vol. 2, Harper and Row, 1982, pp. 431-432.

## **HDLs**

- 153 Hartenstein, R.W. (Ed.), *Advances in CAD for VLSI*, Vol. 7, 1987.

- 154 Campbell, R.H., Koelmans, A.M. and McLauchlan, M.R., *STRICT: a design language for Strongly Typed Recursive Integrated Circuits*, IEE Proc., Vol 132, Parts E and I, No. 2, March/April 1985, pp. 108-115.
- 155 Marshall, R.M., Blair, G.M. and Gray, J.P., *ASIC-BASIC: An Application-Description Language and Compiler*, Proc. 8th International Custom Microelectronics Conference, 1988, pp. 29.0-29.7.
- 156 Rudell, R.L., de Geus, A.J. and Miles, J., *HDL-based Synthesis Speeds Up ASIC Design*, Proc. 8th International Custom Microelectronics Conference, 1988, pp. 55.0-55.7
- 157 VHDL Tutorial for IEEE Standard 1076 VHDL, (Second Draft), CAD Language systems, Inc., 1987.
- 158 Hollingworth, Paul, *The rise of VHDL: 1076 and all that*, IEE Review, April 1991, pp 139 - 142.
- 159 Hilfinger, P.N., *SILAGE: A Language for Signal Processing*, University of California at Berkeley, 1984
- 160 Marwedel, P., *The MIMOLA Design System: A design System which spans several levels*, Methodologies of Computer System Design, Ed. Shriver, B.D., North Holland 1985, pp. 223-237.

## Memory Synthesis

- 161 Fogg, D.C., *Assisting Design Given Multiple Performance Circuits*, VLSI memo No. 88-479, Oct. 1988, pp. 36-42.
- 162 Verbauwhede, I. et al, *Background Memory Synthesis for Algebraic Algorithms on Multi-Processor DSP Chips*, Proc. VLSI '89, pp. 209-218.
- 163 Delaruelle, A. et al, *Synthesis of delay functions in DSP compilers*, Proc. European DAC '90, pp. 68-72.
- 164 Balakrishnan, M. et al, *Allocation of Multiport Memories in Data Path Synthesis*, IEEE Trans. CAD, Vol 7., No. 4, April 1988, pp. 536-540.
- 165 Kurdahi, F.J. and Parker, A.C., *REAL: A Program for REGISTER ALlocation*, Proc. 24th DAC, 1987, pp.210-214.

- 166 Fallside, H. and Denyer, P.B., *Memory Optimised Synthesis from A High Level Language: A First Year Report*, University of Edinburgh, April 1991.

### **Graph Theory**

- 167 Tucker, A., *Colouring a family of circular arc graphs*, SAIM J. Appl. Math., Vol. 29, 1975, pp. 493-502.
- 168 Golumbic, M., *Algorithmic graph theory and perfect graphs*, New York: Academic, 1980.
- 169 Gabow, H. and Kariv, O., *Algorithms for edge colouring bipartite graphs and multigraphs*, SIAM J. Comput., Vol. 11, Feb. 1982, pp. 117-129.

### **Memories**

- 170 Nakano, A., Yasuura, H. and Tamaru, K., *Functional Memory Type Parallel Architecture for Image Processing*, VLSI '89, pp. 329-338.
- 171 Hongjiang, W. and Yulin, Q., *The Design of a Content Associative Memory*, Proc. VLSI '89, pp. 319-328.
- 172 Kohonen, T., *Content-Addressable Memories*, Springer, 1980.
- 173 Chae, S.I. et al, *Content Addressable Memory for VLSI Pattern Inspection*, IEEE Solid-State Circuits, Vol. SC-23, No. 1, 1988.
- 174 Goser, K, Foelster, C. and Rueckert, U., *Intelligent Memory in VLSI*, Information Sciences 34, 1984, pp. 61-82.
- 175 Iizuka, T., *ASIC Intelligent Memory*, Proc. VLSI '89, pp. 307-318.
- 176 Dingwall, A.G.F. and Stewart, R.G., *16K CMOS/SOS Asynchronous Static RAM*, IEEEJourn. SSC, Vol. 14, No. 5, pp. 867-872.
- 177 Hou, J.C.L, *Design of a Fully Associative Cache Memory Controller*, Dept. EE, MIT, VLSI memo 83-133.
- 178 Rideout, V.L., *One-Device Cells for Dynamic Random-Access Memories*, IEEE Trans. Electron Devices, Vol. 26, June 1979, pp. 839-852.
- 179 Weste, N. and Eshraghian, K., *Principles of CMOS VLSI Design, A Systems Perspective*, Reading, Mass., Addison-Wesley, 1985, p.354.



- 180 Chirlian, P.M., *Analysis and Design of Integrated Electronic Circuits*, Vol. 2, Harper and Row, 1982, pp. 502-504.
- 181 Ohwada, N., Kimura, T. and Doken, M., *LSI's for Digital Signal Processing*, IEEE Journ. Solid-State Circuits, Vol. SC-14, No. 2, pp. 221-239.

### **Clocking**

- 182 Mead, C.A. and Conway, L.A., *Introduction to VLSI Systems*, Reading, Mass., Addison-Wesley, 1980.
- 183 Weste, N. and Eshraghian, K., *Principles of CMOS VLSI Design, A Systems Perspective*, Reading, Mass., Addison-Wesley, 1985, p. 221.
- 184 Weste, N. and Eshraghian, K., *Principles of CMOS VLSI Design, A Systems Perspective*, Reading, Mass., Addison-Wesley, 1985, p. 211.

### **Adders**

- 185 Goncalves, N.F. and De Man, H.J., *NORA: A Racefree Dynamic CMOS Technique for Pipelined Logic Structures*, IEEE Journ. Solid-State Circuits, Vol. SC-18, No. 3, pp. 261-266.
- 186 Uehara, T. and van Cleemput, W.M., *Optimal Layout of CMOS Functional Arrays*, IEEE Trans. Computers, Vol. 30, No. 5, pp. 305-311.
- 187 Pomper, M. et al, *A 32-Bit Execution Unit in an Advanced nMOS Technology*, IEEE Journ. Solid-State Circuits, Vol. SC-17, No. 3, pp. 533-538.
- 188 Brent, R.P. and Ewin, R.R., *Design of an nMOS Parallel Adder*, TR-CS-82-06, Dept. of C.S., Australian National University, 1982.
- 189 Uya, M., Kaneko, K. and Yasui, J., *A CMOS Floating Point Multiplier*, Proc. ISSCC '84, pp. 90-91.

### **Other Hardware**

- 190 Tsai, M.Y., *High Density Parity-Checking Circuits with Pass Transistors*, IBM Technical Disclosure Bulletin, Vol. 26, No. 3A, Aug. 1983, pp. 959-960.
- 191 Grifton, W.R. and Hildebeitel, J.A., *CMOS Four-Way XOR Circuit*, IBM Technical Disclosure Bulletin, Vol. 25, No. 11B, Apr. 1983, pp. 6066-6067.

- 192 Whitaker, S., *Pass-Transistor Networks Optimize n-MOS Logic*, Electronics, Sept., 1983, pp. 144-148.
- 193 Law, H-F. S. and Shoji, M., *PLA Design for the BELLMAC-32A Microprocessor*, Proc. ICCD, 1982, pp. 161-164.
- 194 Smith, K.F., *Design of Regular Arrays Using CMOS in PPL*, Proc. ICCD '83, pp. 158-161.
- 195 Krejak, M. and Lipp, R., *Logic Design with CMOS Gate Arrays*, VLSI Design, Oct. 1983, pp. 86-98.
- 196 Vergis, A. *Linear-Testable Counters for Multiple Faults*, Proc. ICCAD '87, pp. 156-159.
- 197 Chirlian, P.M., *Analysis and Design of Integrated Electronic Circuits*, Vol. 2, Harper and Row, 1982, pp. 413 - 414.
- 198 Smith, S.G., PhD Thesis, *Serial-Data Computation in VLSI*, University of Edinburgh, Dept. of E.E., 1987, pp 18-20.
- 199 Chirlian, P.M., *Analysis and Design of Integrated Electronic Circuits*, Vol. 2, Harper and Row, 1982, pp. 426-428.

### **General and Miscellany**

- 200 Evans, Christopher, *The Making of the Micro*, Victor Gollancz Ltd., 1981.
- 201 Evans, Christopher, *The Making of the Micro*, Victor Gollancz Ltd., 1981, p77.
- 202 Evans, Christopher, *The Making of the Micro*, Victor Gollancz Ltd., 1981, p87.
- 203 Evans, Christopher, *The Making of the Micro*, Victor Gollancz Ltd., 1981, p95.
- 204 Goto, S. (Ed.), *Advances in CAD for VLSI*, Vol. 6, p 309, North-Holland, 1986.
- 205 Denyer, P.B. and Bruce, W.H., *Skin Pattern Recognition Method*, ISG Report - Finger-001-C, University of Edinburgh, May 1988.
- 206 DTI/ SERC Technical Objectives Annex A, *A Research Programme in Design Automation and Architectures*, UK-DA Workshop, May 1991.
- 207 Grant, D.M., *Optimal design of median filters in hardware, for real-time image processing*, B.Sc. Hons. Project Report HSP 621, University of Edinburgh, 1988

## **Appendix A - Author's Publications**

### **At ICCAD '89:**

#### **Synthesis of Address Generators**

D.Grant                  P.B.Denyer                  I Finlay

University of Edinburgh,  
Dept. of Electrical Engineering,  
Edinburgh EH9 3JL, UK.

#### **ABSTRACT**

This paper describes an approach to address generation hardware synthesis. We present algorithms and tools that describe the hardware between a binary counter and the address port of a block of memory, which is accessed in some repetitive pattern. These tools match results produced manually for examples taken from a VLSI image processing application.

#### **Introduction**

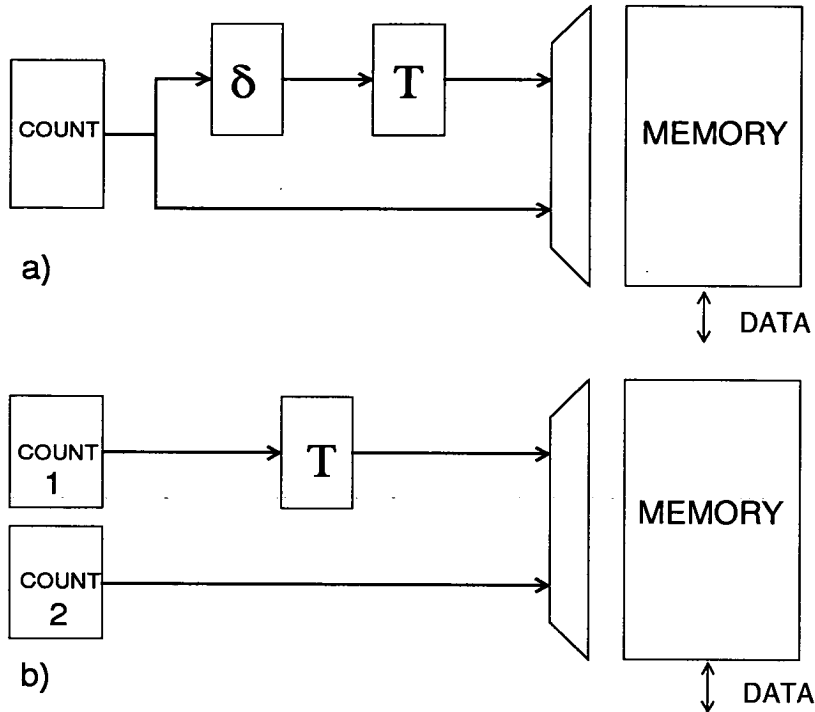
Random Access Memory, as its name implies, supports write and read access to randomly addressable locations. In many applications, however, the sequence of storage and retrieval for particular blocks of data is strongly patterned. This applies particularly in signal processing (several examples from a vision application are repeated below) and in other applications.

In these cases it is often useful to arrange memory allocation in one of the following ways:

- (a) the incoming data is written to consecutive locations and the consumed data is read in the required pattern;
- (b) the incoming data is written in a pre-determined pattern so that reading can proceed from sequential locations.

It then becomes feasible and efficient to generate the necessary address patterns either directly from a dedicated counter, or via circuit transformations (bit-shuffling and combinatorial logic operations) applied to a counter output. Figure 1(a) shows a generic model for this scheme, in which a counter output is used to provide a

consecutive address sequence, which is modified as necessary by an offset,  $\delta$ , and transform,  $T$ .



- Figure 1: Generic address-generation architectures. -

The offset  $\delta$  is additive and simply accounts for an arbitrary delay in commencing the sequence of read operations after the commencement of write operations. This offset function may be avoided if the read sequence does not overlap the write sequence, in which case the counter may simply be reset to commence the read access; or the offset can be achieved by using a second counter started, or reset, to ensure read sequence synchronisation. This arrangement is shown in Figure 1(b) and can be beneficial for high speed applications.

The class of problems we address here are characterised by partial regularity in the retrieval sequence. In particular, we exploit redundancies that are present whenever patterns occur whose length is some power of two. This situation is not so contrived as it first appears. Memory allocation in practice is commonly and advantageously partitioned into power-of-two segments, not least because of the resultant efficiency of address generation that we exploit here. Several examples below reinforce this argument. The corresponding solutions for address generation are often apparently elegant but their derivation is generally non-trivial. Again the examples below demonstrate this point.

### A First Example

As an example, we look at a particular retrieval pattern for image processing\*. This pattern is shown in Figure 2, and is such that every second memory element on every second line is addressed, and this is repeated four times for each 16 by 16 block taken from a 256 by 256 image (interpreted two-dimensionally).

Here it is not obvious how to generate the retrieval sequence, and a designer would use some intuitive knowledge and some trial-and-error to come to a potentially incorrect solution. Unfortunately, simulation is not an effective aid in these circumstances because of the exceptional run times required for even moderate counter lengths.

It is usually quite simple to represent the address sequence in software (Figure 3 represents in pseudo-code the generation of the scan-pattern of Figure 2 ), where it is generated by a series of nested loops, or to build up the address sequence using a graphical entry tool. As long as the address sequence can be generated and has power-two separation, the corresponding address generation hardware may be automatically synthesised.

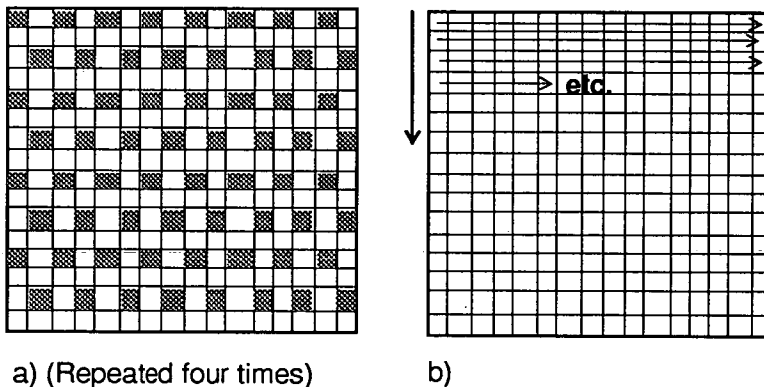


Figure 2: a) Retrieval pattern required for a 16 by 16 block of memory elements;  
b) 256 by 256 pixel array, comprising 16 by 16 blocks from (a).

```
for Y = 0 to 65535 step 4096, (block height = 16 rows)
  for X = 0 to 255 step 16, (block width = 16 columns)
    for i = 1 to 4, (do 4 times)
      for y = 0 to 4095 step 512, (every 2nd line)
        for x = ( y / 512 ) mod 2 to 15 step 2, (every 2nd pixel)
          address = x + y + X + Y,
        next x,
      next y,
    next i,
  next X,
next Y
```

Figure 3: Address generation in software.

\* This is derived from an actual VLSI image processing and pattern recognition system, under development at the University of Edinburgh. This particular example comes from an early image filtering process which samples blocks of the image to determine an appropriate threshold to be set for binarisation.

### **Synthesis algorithm for non-random address sequences**

In this section we describe an algorithm to synthesise transform circuits that operate on the linear sequence produced by the write-address counter to produce the specified retrieval sequence.

The first stage of the process is simply to build a sequential list of the addresses to be produced. (See Figure 4.) This list is the basis for synthesis, and may be generated by hand, by execution of code (Figure 3) or by other methods.

Address list = [0, 2, 4, 8, 10, 12, 14, 513, 515, 517, 519, 521, 523, 525, 527, 1024, 1026, 1028, 1030,.....].

Figure 4: List of addresses to be generated.

Now we attempt to generate this sequence from the bits of the linear sequence generated by a binary counter. Starting with the LSB of the addresses, we look down the sequence of bits, applying the following rules in order, unless otherwise specified:

1) *Split list of bits, list[1..2n], into two halves, list[1..n] and list[n+1..2n].*

For example the list:

[0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1]

becomes:

[0,0,0,0,1,1,1,1] and [0,0,0,0,1,1,1,1]

and the list:

[1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1]

becomes:

[1,0,0,1,0,1,1,0] and [0,1,1,0,1,0,0,1]

(These lists are much larger in practice).

2) *If the original list has a single entry, then force the current address bit to '0' or '1', according to the state of that entry, and then go on to examine the next most significant bit of the addresses.*

This only happens if *all* the entries in the original list for the bit were identical.

*3) If the two halves of the list are identical then reduce the list by returning to (1).*

This controls the use of Rule 1 by allowing the first half of the list to split, only if both halves are identical. For the first example given above, after the first split the two halves are identical, and so we can take the first half and split that:

[0,0,0,0,1,1,1,1]

becomes: [0,0,0,0] and [1,1,1,1].

*4) If the two halves are not identical, nor the logical inverse of each other, then we cannot use any counter bits directly connected to this address bit, and we go on to use the logic generator (section 4 below).*

*5) Rule 5 checks that the list has a length of  $2^m$  (which should always happen) and stores the fact that the  $(m+1)$ th counter bit,  $cbit_m$ , can be used to generate this list:*

Rule 5 is invoked with the knowledge that the two halves of the list are not identical (otherwise it would have been split again) but that they are the logical inverse of each other. Several possibilities arise at this point, with the list having many different possible forms:

[0,0,0,0] and [1,1,1,1],

[0,0,1,1,0,0,1,1] and [1,1,0,0,1,1,0,0],

[1,0,0,1] and [0,1,1,0],

[0,1,0,0,1,1,0,0] and [1,0,1,1,0,0,1,1].

*6a) If all bits in the first half,  $list[1..n]$ , are equal, then the list has been reduced as far as possible, and Rule 7 is called.*

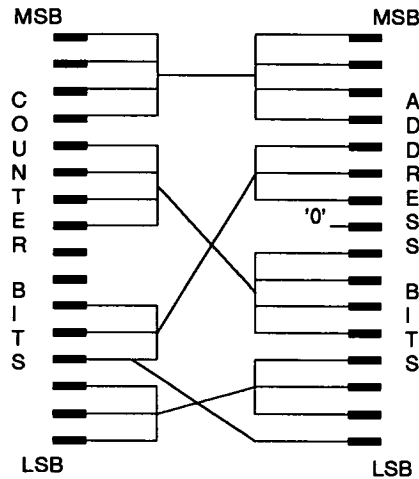
Thus lists which conform perfectly with a binary counter bit are identified.

*6b) If not all bits are equal, then use the  $(m + 1)$ th counter bit (from Rule (5)) XORed with whatever bit is chosen by halving the list again and returning to (2).*

Rule 6b deals with the other possibilities from Rule 5. Any list which has the two halves non-identical, but logically inverse, and not all entries in one half the same, is the XOR function of the  $(m+1)$ th counter bit, with whatever is produced by returning to Rule 1.

cbit 3 ==> adbit 0  
cbit 0 ==> adbit 1  
cbit 1 ==> adbit 2  
cbit 2 ==> adbit 3  
cbit 8 ==> adbit 4  
cbit 9 ==> adbit 5  
cbit 10 ==> adbit 6  
cbit 11 ==> adbit 7  
'0' ==> adbit 8  
cbit 3 ==> adbit 9  
cbit 4 ==> adbit 10  
cbit 5 ==> adbit 11  
cbit 12 ==> adbit 12  
cbit 13 ==> adbit 13  
cbit 14 ==> adbit 14  
cbit 15 ==> adbit 15

(a)



(b)

Figure 5: a) Output from synthesis tool,  
b) Bit mappings for scan-pattern.

7) If the first bit in the list is a '1', then negate whatever counter bit, or function of bits, has been chosen.

8) Print out the connections from counter bit(s) to address bit, and start at (1) with the next most significant address bit.

Once this process has been completed for all address bits, we have a list of connections from counter bits to address bits - the mapping, or transform - which will produce the correct sequence of addresses with the minimum of logic. The resulting mapping for our first example is shown in Figure 5 (Note that this requires only a set of hard-wired connections with no additional logic).

### **Logic synthesis for semi-random address sequences**

Often an address sequence may contain bit sequences which repeat after a power of two of addresses generated, but which do not map directly to a counter bit or an EXORed combination of bits. A logic generator has been written, based on sum-of-products logic, which synthesises the necessary logic for generation of these bit sequences from



a raw counter output. Once it has been decided by rule (4) in the previous section, that some logic is needed to generate the address bit currently under analysis, the logic generator is called. This consists of the following processes:

- 1) Find the minterm value ('0' or '1') by counting 1's in sequence.
- 2) Find the next minterm which has not already had logic generated for it.
- 3) Generate the logic for this (and possibly other) minterm(s) by exhaustive mask generation and pattern matching. Mask generation produces all possible combinations of counter bits, and all counter values with the same pattern of bits are checked. If this pattern matches only those counter values which correspond to minterms, then the logic to generate these minterms is easily read from the bit pattern.
- 4) Print out the logic generated and return to (2) if any minterms remain unmatched to logic.

In this way a semi-random sequence of bits, which repeats every  $2^n$  addresses, can be generated by  $n$  counter bits plus minimal logic. An example is given below.

```

rand[ ] = { 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1 },
for Y = 0 to 65535 step 4096,
  for X = 0 to 255 step 16,
    for y = 0 to 4095 step 256,
      for x = 0 to 15 step 1,
        address = rand[x]*x + y + X + Y,
      next x,
    next y,
  next X,
next Y.

```

Figure 6: Loops for a semi-random scan-pattern.

```

(cbit0.cbit1bar.cbit2 +
cbit0.cbit2.cbit3) ==> adbit 0
(cbit0bar.cbit1.cbit2bar +
cbit0.cbit1.cbit2.cbit3) ==> adbit 1
(cbit1bar.cbit2 +
cbit0.cbit2.cbit3) ==> adbit 2
(cbit0bar.cbit1.cbit2bar.cbit3 +
cbit1bar.cbit2.cbit3 +
cbit0.cbit2.cbit3) ==> adbit 3
cbit 8 ==> adbit 4
cbit 9 ==> adbit 5
cbit 10 ==> adbit 6
cbit 11 ==> adbit 7
cbit 4 ==> adbit 8
cbit 5 ==> adbit 9
cbit 6 ==> adbit 10
cbit 7 ==> adbit 11
cbit 12 ==> adbit 12
cbit 13 ==> adbit 13
cbit 14 ==> adbit 14
cbit 15 ==> adbit 15

```

(a)

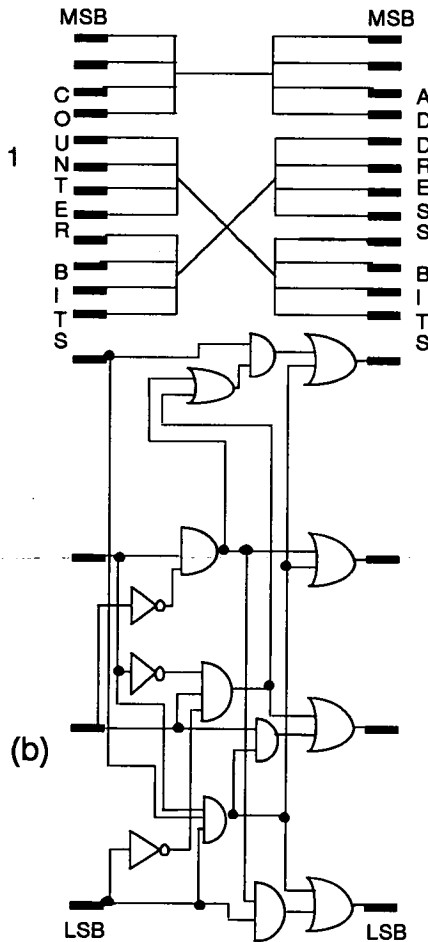


Figure 7: a) Output for semi-random example,  
b) Bit mappings & logic.

### Further work

If the semi-random sequence of address bits is longer than some number of bits, then

we consider it to be fully random, and must look to other methods to generate the addresses.

Possible solutions include the use of either local or controller ROM space to store the addresses, or some combination of the two. It may also be possible to re-allocate the memory space itself, so that the addresses can be more economically generated. The cost involved in each of the possible solutions is calculable from the relative areas of ROM and RAM bits, and of flip-flops (for the counter). The area of wires is not calculable until their lengths are known, but an n-bit bus can be taken to be n times the area of a single wire, of an arbitrary length.

Work is proceeding on a goal-directed tool which will attempt to automatically synthesise address generation hardware for any sequence of addresses of any length. This tool is targeted at address generation for memory requirements incurred during the design of a system, rather than those inherent in the system's specification.

### **Status and performance**

The synthesis procedures given here have been implemented in 'C' and used to generate the above examples. It was found that execution time varied linearly with the length of the address sequence, and depended very little on the complexity of the solution, as shown in Table 1 below.

Length of sequence (No. of addresses)	Execution time (seconds)
4096	3.5
8192	7
16384	14
32768	28
65536	56

Table 1: Performance statistics on a Sun 3/60.

### **Conclusion**

Address generation is an important element of a whole synthesis system. Memory allocations which emphasise power-of-two patterns encourage efficient address generation and we have reported general synthesis techniques to realise address generators which exploit this potential. The tool has been written and used to synthesise examples drawn from a real VLSI vision system. The results match those produced manually and have led to the adoption of this tool in a second generation design for the system.

### **Acknowledgements**

The authors acknowledge the support of the Silicon Architectures Research Initiative, and the Science and Engineering Research Council.

## **At DAC '90:**

### **Memory, Control and Communications Synthesis for Scheduled Algorithms**

Douglas M. Grant\* and Peter B. Denyer

Silicon Architectures Research Initiative  
Department of Electrical Engineering  
University of Edinburgh, Scotland, EH9 3JL

**This paper explores a method of grouping individual memory requirements from a hardware-constrained schedule of an algorithm, such that control and communications may be optimised. A new representation of memory requirements is introduced to explain the method. The technique may also be used to allocate operations to hardware resources. This, and control and communication optimisation are illustrated with an example.**

### **1. INTRODUCTION**

An important step in any ASIC synthesis system is that of memory allocation for intermediate variables, which may come before or after the operator (hardware resource) allocation step, working on a validly scheduled algorithm. This step will in turn introduce communications and control requirements, for which an optimum solution must be found.

The intractability and interdependency of each of these steps can result in an iterative synthesis method in order to obtain a good result. In order to produce real-time feedback to an interactive scheduler however, a faster, one-shot synthesis method is more useful, and this paper describes such a scheme. Section 2 sets the limits of the problems we aim to solve. Section 3 gives an overview of related work. The example, with which the method is explained, is introduced in Section 4, and Sections 5, 6 and 7 describe the memory, communications and control synthesis steps respectively. Results and comparisons with other work are presented in Section 8, and conclusions are drawn in Section 9.

### **2. LIMITS OF THIS WORK**

The schedule on which data-path synthesis depends, is passed to the tool as a database of Edinburgh-Prolog facts, which defines a directed (a)cyclic graph, as well as information about the hardware constraints in force. Pipeline delays may be declared explicitly in the schedule. The synthesis tool then extracts all memory requirements and, inserting delays if necessary, groups these requirements into memory blocks, having one Read and one Write port. A bus-based communications network is then constructed between hardware resources and the memory blocks, and its multiplexers' control requirements extracted and minimised. No attempt is made to further reduce the size of the memory blocks by register sharing, since this will be carried out by a separate tool, currently under construction, with address generation costs in mind.

The whole process may be carried out before or after operator allocation, but if done before, will return such an allocation, derived directly from the memory allocations.

The output from the system consists of a netlist of connections between operators, memory blocks and 2-to-1 multiplexers (2to1muxes), along with optimised control sequences for those muxes, and the virtual address sequences for the memory blocks.

\* Supported by the Science and Engineering Research Council, and by British Aerospace.

Conditional branches are treated as separate, since the memory requirements of each branch may be different, and so all possibilities must be allowed for.

### **3. RELATED RESEARCH**

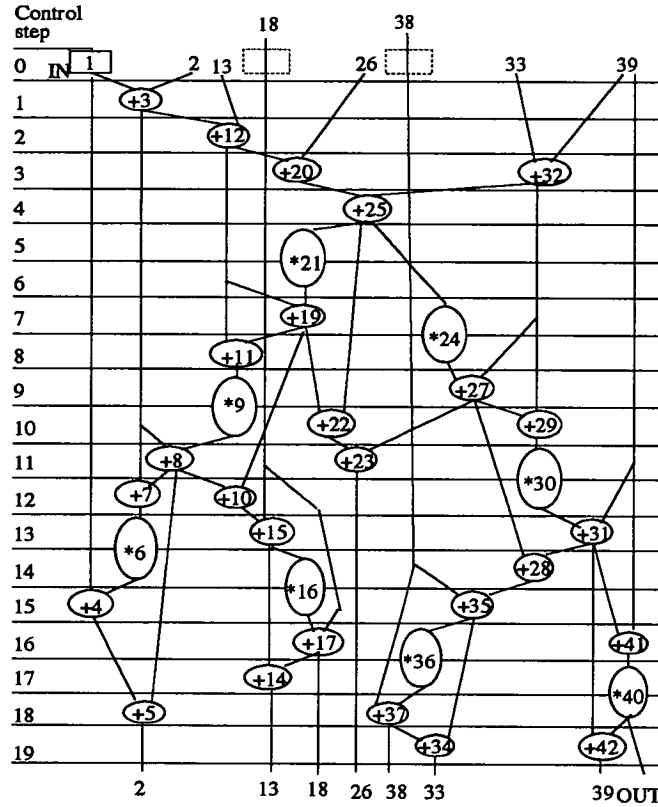
Several different approaches to the problem of register allocation have previously been attempted. In CATHEDRAL II [1], memory requirements suited to register files and FIFO's are extracted from the algorithm description by a Background memory manager, which passes them to a Foreground memory manager. CMU-DA [2] uses a linear programming approach to group registers to multiport memories such that interconnect costs are reduced by judiciously assigning registers to the ports of the memories. This may be done before or after operator allocation, but does not take into account register sharing or the functionality of the operators.

REAL [3] uses a greedy Left-edge algorithm to minimally colour a set of data lifetimes and thus find an optimal set of registers before operator allocation, and EASY [4] does the same, but after allocation, and includes interconnect costing, before attempting to group registers into register files, again with interconnect costs in mind. However, results are not presented for this. FACET [5] uses formal clique-partitioning techniques to group variables onto registers, which are then grouped into files, if possible, before operator allocation takes place. SPLICER [6] utilises a recursive synthesis method, on small sections of a schedule at a time, to construct a bus-based interconnection network, but predefines a possibly incorrect minimum number of registers on which to work, while SCHOLAR [7] synthesises a point-to point interconnection network, with single-level multiplexing. SPAID [8] groups memory requirements into register files before attempting to share registers between requirements. However, duplication of data is sometimes necessary, to reduce the interconnect costs, and a two phase clocking scheme allows simultaneous Reads and Writes to the same location.

### **4. INTRODUCTION TO THE EXAMPLE**

The example with which the synthesis method is explained, is that of the wave-digital filter, for which Paulin's [9] force-directed schedule is shown below. The available hardware consists of two adders which operate in a single control step (cstep), and a single, pipelined multiplier, operating in two csteps.

One input to the multiplier is always a constant, and has been omitted from further use of the example.



Paulin's force- directed schedule of wave filter.

## 5. MEMORY SYNTHESIS

Assuming no operator allocation has yet been attempted, memory synthesis proceeds as follows.

### 5.1 Latch Insertion

Examining the schedule above, it is noted that data produced by operations 17 and 37 are not finally used until the same cstep as they are produced in the next cycle of the system. Since data may not be read from and written to the same location in a single cstep (assuming single phase clocking) to avoid overwrite errors, two memory elements (memels) are needed to store each datum. In order that subsequent address generation may be done on a single cycle of the system, a dedicated latch is introduced in cstep0 for each datum, as shown by the dotted boxes - the latch "operations" - in the schedule.

### 5.2 Grouping memory requirements into memory blocks

We now attempt to group the individual memory requirements extracted from the schedule into memory blocks, so that there are no two simultaneous Reads or Writes to a block. Each memory requirement (operation) is given a private memory element (memel), identified by the same number as the operation which will Write to it. To increase the tractability of this step, and with communications (bus and multiplexing) costs in mind, we do this separately for each resource type in turn. This also allows us to ignore data width information at this point. A clique-partitioning approach is possible here, but we have used a simpler heuristic search technique to produce a result more quickly.

A weight is given to each memel by counting the number of simultaneous Reads or Writes with other memels of the same resource type. If there is only one resource of a type available (eg: one multiplier),

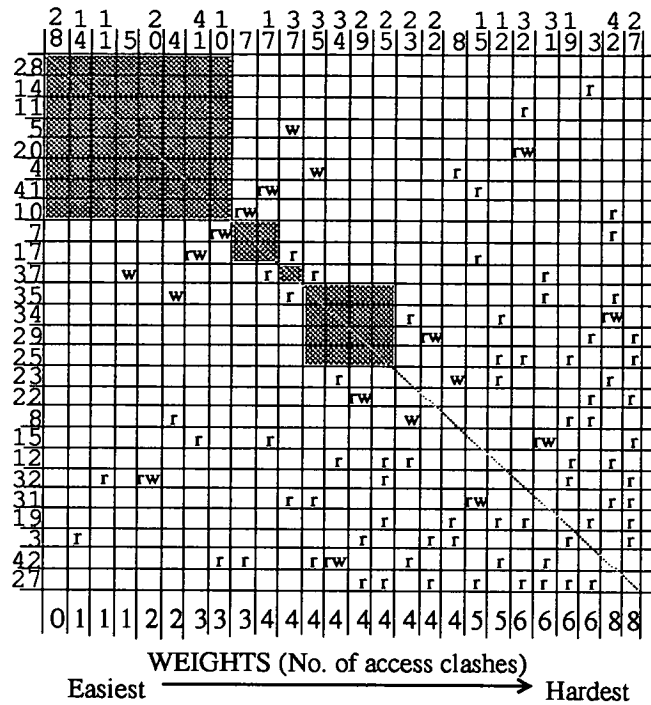


Figure 1: Square-graph of Read (r) and Write (w) access clashes between adder-type memels.

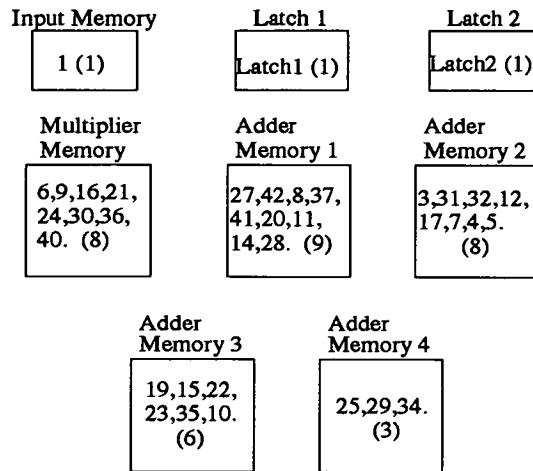
then there will be no Write clashes. These access clashes, and their bearing on the problem, may be clearly understood by examining the Square-graph below, which contains information on the add operations only, for which there are two adder resources available.

The memels on the axes of the graph have been sorted according to the number of access clashes each has. We now wish to draw the minimum number of boxes on the diagonal of this graph such that no box contains an access clash, and no boxes overlap, as shown by the example shaded boxes on the diagram. To get the minimum number of boxes, however, there will be some re-ordering of the memels on the axes. If we start by examining the memel with the least number of access clashes (the 'Easiest' memel - no. 28), then we will have the best chance of finding another memel to group with it. This tends to produce a solution with a few large groups and several small groups. If we start with the 'Hardest' memel first (no. 27), then the solution tends to have a few, similar sized groups.

### 5.3 Results of Memel Grouping

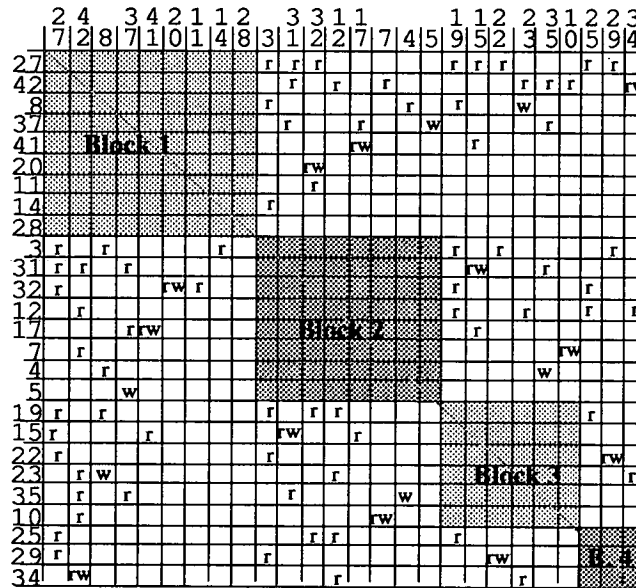
Shown below are the Square-graph, obtained using the "Hardest-first" heuristic on the adders' memels,





Memel-grouping solution using Hardest-first method.  
The numbers in the memory blocks are those of the operations whose data will be stored there, and the numbers in brackets are the number of memels in each block.

and the full solution for all resource types, using the same heuristic.



## 6. COMMUNICATIONS NETWORK SYNTHESIS

We must now construct an optimal, bus-based communications network between computational resources (eg: adders, multipliers) and these memory blocks, as well as allocating operations to any multiply-available resources.

### 6.1 Optimising the Write-bus network

As stated previously, we wish to find some groups of memory blocks which are written to by a single resource, ie: There are no simultaneous write accesses within the group of blocks.

For singly-available resources, this poses no problem, and all memory blocks dedicated to each single

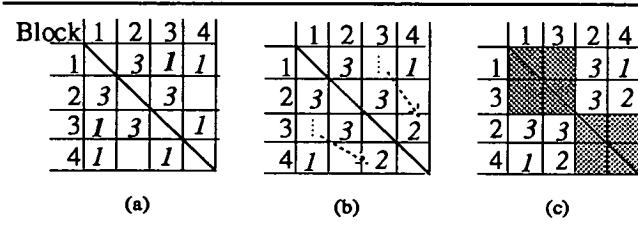


Figure 2: a) Square-graph of Write clashes between adders' memory blocks; b) after regrouping; c) after reordering a solution is found.

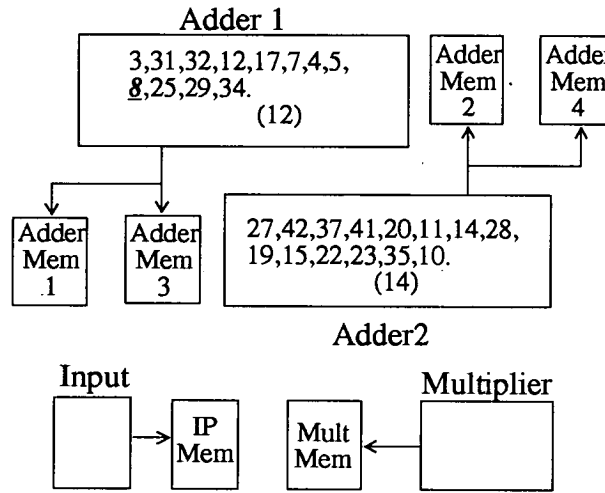


Figure 3: Write-bus architecture with adder allocation.

resource can be grouped onto a single Write bus. Where there is more than one resource of a type available (eg: two adders), we wish to find the same number of groups of memory blocks as there are resources (two).

Constructing another Square-graph (Fig. 2a) for the adder memory blocks, with a count of the number of simultaneous Write accesses between each pair of blocks, we wish to draw two boxes (since two adders) on its diagonal, as before, containing no Write access clashes. However, this is not possible as the graph stands, and we must regroup the memels into blocks to make it possible. This does not involve backtracking all the way to the original grouping algorithm, but uses simple heuristics to choose and relocate any obstructive memels as necessary. For instance, as shown in Figure 2a, there is only one Write clash between blocks 1 and 3, and identifying the clashing memels as 8 and 23 respectively, memel 8 is chosen and moved from block 1 to block 4. Now we can redraw the Square-graph (Fig 2b), and draw two boxes to cover the diagonal, after

reordering the blocks on the axes, as shown in Figure 2c. Also shown above (Fig. 3) is the corresponding Write-bus architecture for this solution, which includes the operation allocation information for the adders. The two latches have been omitted since they are written to by the memory blocks, via a Read-bus network which does not exist yet.

## 6.2 Read-bus network requirement extraction

We now look to create a bus-based communications network between the Read ports of the memory blocks and the inputs to the computational resources (and latches also).

Allowing for commutative properties of some resources (as specified in the input database, for each re-

source type), the minimum number of paths between memory blocks and resources is compiled. Each path has a source (a memory block), a destination (a resource input) and a list of control steps during which that path is used. Path making is done for each cstep in turn, starting at a user-specified cstep, which should contain the use of all resources of a type. For example, both adders and the multiplier are first used together in cstep 11, so that should be chosen as the starting point for path-making. The list of paths, some of which are shown below, constitute the Read-bus requirements of the system, and must then be rationalised into an optimum multiplexer network.

#### PATHS TO ADDERS (Start at cstep 11)

<u>From</u>	<u>To</u>	<u>Port</u>	<u>Csteps when used</u>
Latch 2	Adder 1	1	[16]
*1	Adder 1	2	[11,13,15,16]
+3	Adder 1	2	[19]
In1	Adder 1	1	[1,15]
+2	Adder 1	1	[11,12,18]
+4	Adder 1	2	[3,12,18]
+1	Adder 1	1	[2,3,4,10,13,19]
+2	Adder 1	2	[1,2,4,10]
Latch 2	Adder 2	2	[13]
+1	Adder 2	2	[11,14,15,16]
Latch 1	Adder 2	1	[15,18]
+2	Adder 2	2	[3,8,17]
+2	Adder 2	1	[7,9,14,16,19]
*1	Adder 2	2	[7,9,18,19]
+3	Adder 2	1	[3,8,10,11,12,13,17]
+4	Adder 2	2	[10,12]

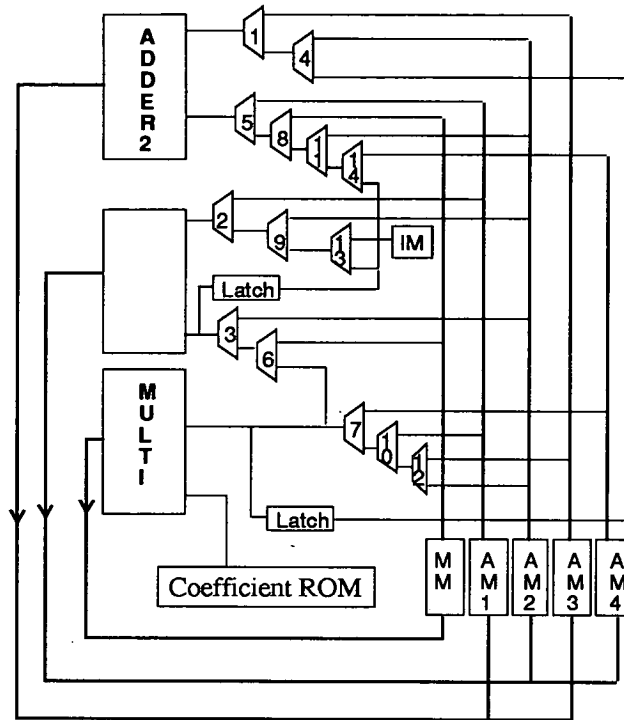
**KEY:** \*1 = Multiplier Memory  
In1 = Input Memory  
+n = Adder Memory n

### 6.3 Optimising the Read-bus network

As a starting point for optimisation, each input to each resource has bound to it an n-to-1 multiplexer (muxnto1), where n is the number of memory blocks which must feed data to that input. The value of n should have been minimised by the previous synthesis step. 2-to-1 multiplexers (2to1muxes) are then extracted from these muxnto1's, reducing the value of n by 1 each time, and erasing csteps from the respective path's list, until no muxnto1s remain. The most used path is examined at each pass of the synthesis algorithm, increasing the amount of Don't Care values in the subsequent control requirements for the 2to1muxes, which is valuable in optimising the control sequences (Section 7). The result of this step is a netlist of connections from memory blocks or 2to1muxes, to resources or other 2to1muxes. Figure 4 presents the final architecture synthesised for the example. There are 14, 2to1muxes, some of whose inputs have been swapped by the control optimiser described in the following section.

## 7. CONTROL SEQUENCE SYNTHESIS

In the controller, a PLA-FSM or Counter-ROM method may be utilised to generate the control sequences for the multiplexers, and the address sequences for the memories. Setting the address generation aside, as a subject too complex to explore here, we must find some way of reducing the area of the controller's PLA or ROM.



KEY: IM = Input Memory  
MM = Multiplier Memory  
AMn = Adder Memory n

Figure 4: Resulting Architecture for Wave Filter

### 7.1 Extraction of Control Bit Sequences

Using the interconnection netlist generated by the previous step, and the Read-bus requirements for the system, we can trace a path back through the multiplexer network, from resources to memories, noting the necessary values of the multiplexer control bits as we pass through them. This is done for each control step in turn, until we have a complete control bit sequence for each 2to1mux in the network, some of which are shown below for our example. To avoid confusion between real and virtual bit values, the control bits may have (virtual) value '1' or '2' and any Don't Care values are denoted by a '0'. The real values corresponding to '1' and '2' will be decided on later.

#### CONTROL BIT SEQUENCES

(csig1,[mux1],[0,0,0,1,0,0,0,2,1,2,1,1,1,1,2,2,2,1,2,2])  
(csig2,[mux2],[0,2,1,1,1,0,0,0,0,0,1,2,2,1,0,2,2,0,2,1])  
(csig3,[mux3],[1,1,1,2,1,0,0,0,0,0,1,2,2,2,0,2,2,0,2,2])  
(csig4,[mux4],[0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,1,2,1,0,2,1])  
(csig5, etc.

### 7.2 Optimising the Control Bit Sequences

Firstly, we calculate the "overlap" for every pair of control sequences. Three types of overlap are possible: A "straight" overlap is where two sequences have the same virtual values during some csteps, and never have different values during any others. The number of overlapping values is used as a weight for

that pair of sequences. An "inverted" overlap exists when two sequences have opposite, and never the same virtual values in some csteps, and again a weight is calculated from the amount of overlap. This inversion of the virtual values will merely cause the inputs to the corresponding 2to1 mux to be swapped, at no cost. A "negative" overlap happens when two sequences have only Don't Care values in common, and the number of overlapping Don't Cares is the weight for that pair of sequences. A fourth type of overlap is the "null" overlap, which means that two sequences may never be generated along the same control line, because their virtual values clash at some point.

For example, sequences (a) and (b) below have a straight overlap of weight 5:

(a) 0 0 1 2 2 1 1 0 1 2  
      \* \* \* \*

(b) 2 0 1 2 2 1 0 2 0 2.

The control bit sequences are ordered by the number of Don't Care values in each, and the "busiest" sequence - that with the fewest '0's - is examined first for possible folding into the others. If several other sequences may share a control line with this one, then the sequence with the largest overlap is chosen, the two sequences are merged into a new control sequence, and the whole operation is repeated, until no more possible folds are found.

### 7.3 Results of Control Bit Optimisation

Shown below are the 7 maximally-folded control bit sequences for the example, which originally numbered 14.

In another example, a set of 40 multiplexer control sequences of length 14 bits was reduced to just 8 sequences.

#### FOLDED CONTROL SIGNALS

(csig2,[mux2],[0,2,1,1,1,0,0,0,0,1,2,2,1,0,2,2,0,2,1])  
(csig5,[mux5],[0,0,0,2,0,0,0,2,2,2,1,2,2,1,1,1,2,2,2])  
(csig6,[mux6],[0,0,0,2,0,0,0,0,0,0,0,1,2,1,0,1,1,0,2,2])  
(csig16,[mux12,mux1,mux8],  
          [0,0,0,2,0,0,0,1,2,1,2,2,2,2,1,1,1,2,1,1])  
(csig17,[mux3,mux10],  
          [1,1,1,2,1,0,0,0,0,1,1,2,2,2,2,2,1,2,2])  
(csig19,[mux7,mux9,mux14],  
          [2,2,0,1,0,1,0,1,0,2,1,1,1,2,2,2,2,1,2])  
(csig21,[mux4,mux13,mux11],  
          [0,2,0,1,0,0,0,1,1,1,2,0,2,2,1,2,1,1,2,1])

Any Don't Care values remaining in the control bit sequences are then given values '1' or '2', in such a way that the sequences are maximally symmetrical. If it is possible to generate a bit sequence using a shorter, repeating sequence, then this solution will be found. For instance, it can be found that the sequence for control signal 1 (csig1) may be generated from the shorter sequence, [1 2 1 1 2], as follows:

csig1: 0,2,1,1,1,0,0,0,0,0,1,2,2,1,0,2,2,0,2,1  
      1,2,1,1,1,2|1,2,1,1,1,2|2,1,2,2,2,1|2,1  
                  (Inverted after two runs)

Control signal 6 (for mux6) may be similarly generated using the sequence [1 2 2 2] as a base, inverting its values after every two runs, and so on for the rest of the sequences.

The control for the latches will be left for now, since a choice of rising/falling edge-, or level-triggered latch is possible, and is technology dependant.

## 8. RESULTS

The following results have been found for some examples. The need for a validly scheduled algorithm, without operation chaining or allowing simultaneous Reads and Writes to the same physical location, has restricted the number of examples. Also, since this technique is targeted at algorithms with a larger number of operations (>20), scheduled with tight hardware constraints, there are very few applicable examples available. A fast discrete cosine transform (FDCT) algorithm was tried, containing 50 operations, scheduled in 13 control steps, with 2 adders, 2 multipliers and 2 subtractors available. Shown below are the results for the wave filter example, and comparisons with other work, as well as the results for the FDCT schedule. The minimum number of registers in each memory block after possible sharing, was calculated by hand using well known graph-colouring techniques [10].

The synthesis system has been coded in Edinburgh-Prolog, running on a Sun 3/60 workstation.

Example	# Mux Inputs	# Control/ Address Bits	# Regs	# Comms buses	CPU Time (secs)
Wave Filter	18 (14, 2to1 muxes)	14 / 8	17	29	35
Ditto: HAL [9]	26	NA	12	47	360 (including scheduling)
Ditto: Splicer [6]	43	NA	NA	NA	55
FDCT	53	31 / 12	33	NA	180

Results for Wave filter and FDCT examples

## **9. CONCLUSIONS**

Due to the lack of applicable benchmarks, this synthesis method has not yet been fully proven to be better than any other method, in general. However, the result obtained for the wave filter example shows a marked reduction in communications and control complexity, although there is an increase in the number of registers.

As stated previously, the original purpose of the system was to generate virtual address sequences with which to test an Address-Generator Synthesiser, which it does, so the use of the system to produce data paths is simply a useful sideline.

## **10. REFERENCES**

- [1] Verbauwhede, I. et al., "Background Memory Synthesis for Algebraic Algorithms on Multi-Processor DSP Chips," *Proc. VLSI 89*, pp. 209-218.
- [2] Balakrishnan, M. et al., "Allocation of Multiport Memories in Data Path Synthesis" *IEEE Trans. CAD*. Vol. 7, No. 4, April 1988, pp. 536-540.
- [3] Kurdahi, F.J. and Parker, A.C., "REAL: A Program for REGISTER ALlocation," *Proc. 24th Design Automation Conference*, 1987, pp. 210-215.
- [4] Stok, L. and Van Den Born, R., "EASY : Multiprocessor Architecture Optimization", in *Proc. Int. Workshop on Logic and Architecture Synthesis for Silicon Compilers*, ed. Saucier, G. and McLellan, P.M., Grenoble, May 1988, pp. 313-328.

- [5] Tseng, C. and Sewiorek, D.P., "Automated Synthesis of Data Paths in Digital Systems," *IEEE Trans. Computer-Aided Design*, Vol. CAD-5, July 1985, pp. 379-395.
- [6] Pangrie, B.M., "Splicer: A Heuristic Approach to Connectivity Binding," *Proc. 25th Design Automation Conference, 1988*, pp. 536-541.
- [7] Haroun, B.S. and Elmasry, M.I., "Architectural Synthesis for DSP Silicon Compilers", *IEEE Trans. CAD.*, Vol. 8, No. 4, April 1989, pp. 431-447.
- [8] Bergamaschi, R.A. and Allerton, D.J., "A Graph-Based Silicon Compiler for Concurrent VLSI Systems," *IEEE CompEuro.*, 1988, pp. 36-47.
- [9] Paulin, P.G., "Force-Directed Scheduling in Automatic Data Path Synthesis," *Proc. 24th Design Automation Conference, 1987*.
- [10] Tucker, A., "Applied Combinatorics," *Pub. John Wiley & Sons, 1980, ISBN 0-471-04766-X*, pp. 261-274.

## **At EDAC '91:**

### **Address Generation for Array Access Based on Modulus m Counters**

Douglas M. Grant

Peter B. Denyer

University Of Edinburgh,  
Kings Buildings,  
Mayfield Road,  
Edinburgh, EH9 3JL.

#### **Abstract**

*The necessary task of Address Generation for RAM and ROM accesses can often result in hardware taking up an appreciable fraction of the area of a data processing IC. Close examination of the address sequences can reveal symmetry which may be exploited to automatically devise small and simple address generators, based on counters. This paper will describe automated techniques used to recognise and develop symmetries in address sequences, and to synthesise the necessary address generation hardware.*

#### **Introduction**

In contemporary High Level Synthesis systems, the task of designing address generators usually comes late in the design process, after data-path, memory and communications synthesis steps. But with address generation hardware taking up to half the final chip area<sup>†</sup>, it is clear that this step is one of importance, and so deserves a closer investigation.

Address generators can be partitioned into three main types. There are those for data-dependent address generation, where an address is some function of internal variables, and specialised hardware should be constructed to perform this function. The second type of address generator cannot be constructed until memory synthesis has been completed, and must generate the addresses to access temporary storage areas, which are the result of grouping registers into register files or RAMs [1]. These generators can take the form of a ROM lookup table, although some clever assignment of values to actual memory locations may allow a significantly smaller solution to be found [2]. The third type of address generator, and the one dealt with in this paper, is that for array-type memory accesses. Here, very often a regular sequence of addresses is required, and since this sequence can often be determined directly from the behavioural description of the chip, the synthesis and optimization of the generator can precede, or run in parallel with other design stages.

\* Supported by SERC and BAe.

† Around half of the active area of an image processing chip, designed using Solo 1200, comprised Address Generation circuitry.



As the address sequence can be rather long, the definition of address generators by hand, and especially the simulation of their correctness, becomes very difficult. The description of an array-type address sequence may be a set of nested loops, whose variables are combined to give each address at successive passes of the loops, and it is possible to examine the loops' variables and increments and to build an address calculation unit (ACU) to realise the loops' function in hardware [3,4].

It may also help to map the memory addresses onto actual memory locations to allow simpler addressing schemes [5]. PLA FSMs based on counters may also be used to generate address sequences [6], and simple binary counters can be very effective in some circumstances [7]. Combinatorial logic plays a large part in many address generator designs, although the large size and low speed of the circuits can be prohibitive.

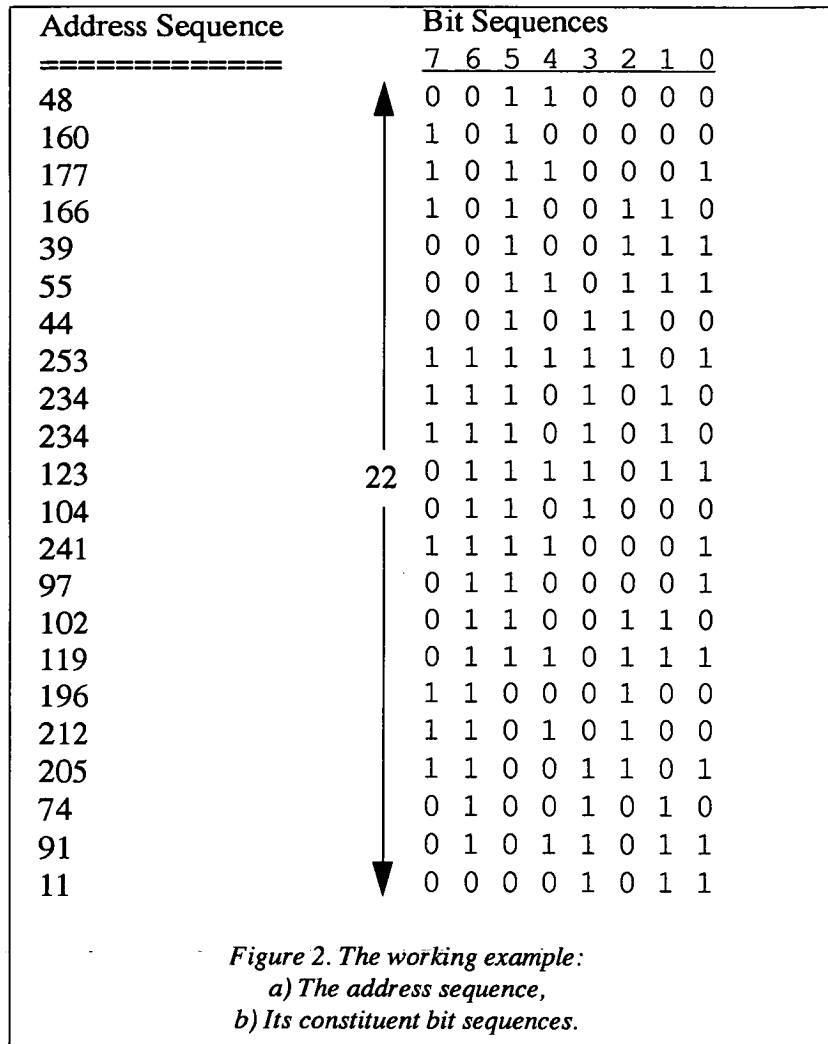
Many problems arise in address generator design when the regular addressing pattern is not based on a binary sequence, and this paper is targeted at those situations, and solutions based on non-binary counters. This is a generalisation of previous work [7], which could only deal with binary sequences.

Firstly the situation is explained more clearly, along with the introduction of a working example, and then the primary stage of synthesis, that of developing the problem to suit the algorithms, is examined. The next part of the synthesis process is then described, in which an address sequence is matched to a suitable address generator. Finally, some examples are given to demonstrate the power of the tool, before conclusions are drawn and future work laid out.

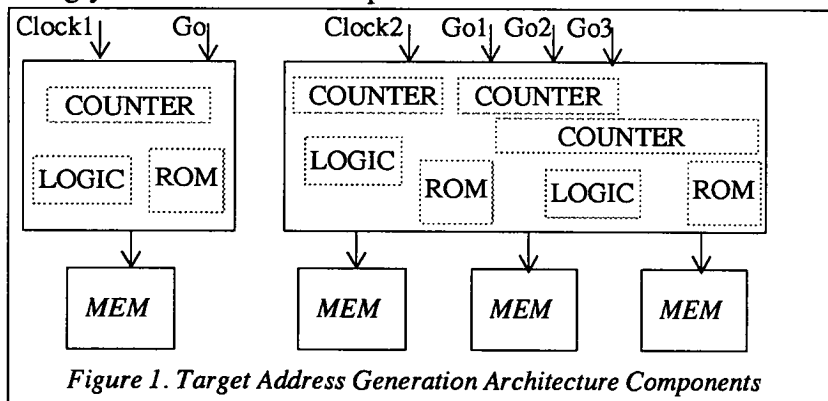
### **Problem Definition**

The task of automatically synthesising an address generator is simply defined: Given predetermined sequences of memory addresses to be generated regularly in time, synthesise the hardware which will do just that (Figure1), hopefully comprising various counter bits.

The address sequence may be extracted from a software description, usually in the form of a set of nested loops, at the centre of which the address is specified by some function of the loop variables. However, to give a more general view of the tool, the working example shown in Figure 2 does not follow this premise, but contains several different bit sequences, each included to show some feature of the tool, which together



define a seemingly random address sequence.



One major problem a designer faces with this task is the long length of the address sequence, often comprising tens of thousands of addresses, and this makes full examination of the sequence rather arduous. However a computer is ideal for processing the address sequence, if only we can give it a designer's intuitive knowledge of what to

look for in a sequence in order to match it to an address generator.

### **Developing Possible Sequence Symmetry**

Symmetry, or regularity of an address sequence, easily recognised in a graphical description of the sequence on the memory space, can be exploited to allow simpler generation of the sequence, by output bits from a counter. To automate the recognition of symmetry however, we first split the address word sequence into a set of bit sequences, and apply the algorithms to each bit sequence in turn.

#### **The Hint**

Very often the designer will be in a position to give the algorithm some indication of the solution expected, without any knowledge of the detailed solution. This "hint" is simply an integer, which should be the number of memory accesses in the most basic, repeating access pattern involved, which is repeated to cover the memory space, thus building up the entire address sequence.

If the designer is not in a position to specify this hint, as for our working example, then it may be found automatically for each bit sequence by exhaustively searching the bit sequence to find the length of the shortest repeating sequence of bits, and then defining the hint as the lowest odd factor of this length. For instance, if the shortest repeating bit sequence has length 40 bits, then the hint can be found thus:

$$40/2 = 20, /2 = 10, /2 = 5 = \text{Hint.}$$

#### **Padding the Bit Sequences**

Once the hint has been specified for a bit sequence (it will often be common to all bit sequences), it is used to develop any possible symmetry or regularity in the sequence. To allow the forthcoming algorithms to work, we "pad out" the bit sequence so that it has a length equal to some power-of-two times the hint. This is simply done by appending the correct number of bits from within that sequence, inverting their value if necessary.

##### **Padding to a Whole Number of Basic Patterns**

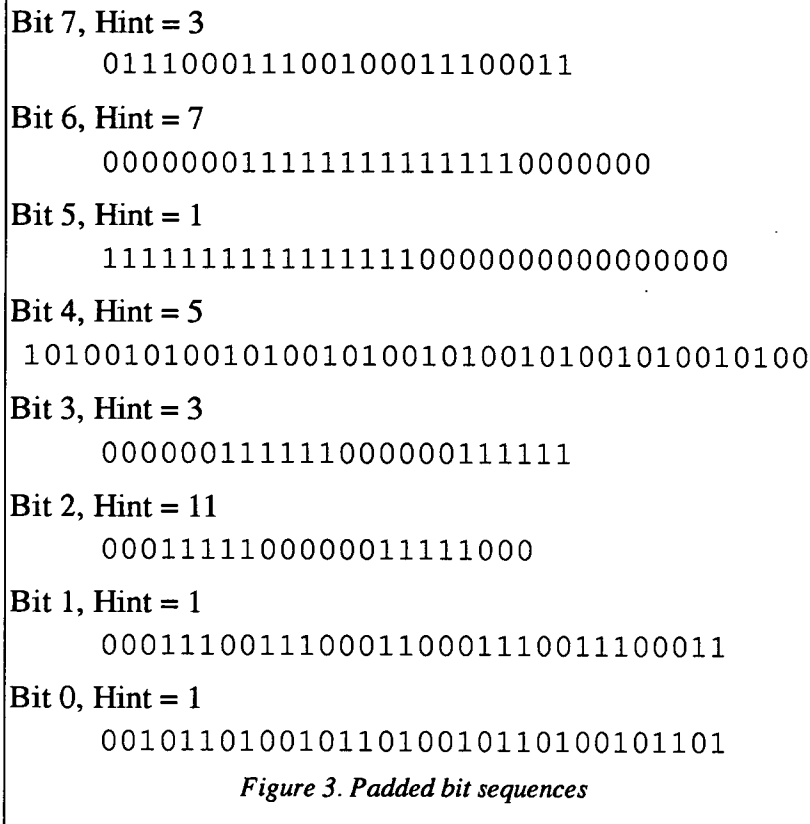
In some cases, as for the working example, the original bit sequence will not contain a whole number of basic patterns. We must first pad this sequence to a length ( $i \times \text{hint}$ ), where  $i$  is the next integer above ( $\text{Orig\_length} / \text{hint}$ ).

##### **Padding a Bit Sequence to Length $2^j \times \text{Hint}$**

Once a whole number of basic patterns is available, the bit sequence can then be padded to its final length of  $2^j \times \text{Hint}$  ( $j = \text{integer}$ ), again appending previous bits in the sequence, inverting their logic values (0 or 1) if necessary.

All this means that the resulting bit sequences will at least be symmetrical for the copied bits, allowing us to prepare for the next stage of the synthesis algorithm, by finding the Repetition Sequence for the bit sequence.

Shown in Figure 3 are the bit sequences for our working example, after padding.



### The Repetition Sequence

To reduce the complexity of the synthesis task, and allow the algorithms to function correctly, the padded bit sequence is converted into a sequence describing the repetition of similar bit values - The Repetition Sequence.

This consists of a polarity value, which is simply the first bit in the bit sequence, followed by a sequence of integers specifying the number of successive bits of each value. To clarify, the working examples are given below (Figure 4), some of which

exemplify the reduction in complexity this conversion allows.

<u>Bit number</u>	<u>Repetition Sequence</u>
7	0, (1,3,3,3,2,1,3,3,3,2)
6	0, (7,14,7)
5	1, (16,16)
4	1, (1,1,1,2,1,1,1,2,1,1,1,2,1,1,1,2, 1,1,1,2,1,1,1,2,1,1,1,2)
3	0, (6,6,6,6)
2	0, (3,5,6,5,3)
1	0, (3,3,2,3,3,2,3,3,2,3,3,2)
0	0, (2,1,1,2,1,1,2,1,1,2,1,1,2,1,1,2, 1,1,2,1,1)

*Figure 4. The Repetition Sequences*

### **Collapsing the Repetition Sequence**

Once formed, the repetition sequence may be iteratively bisected, according to a number of rules, in such a way that a bit sequence generator may be found.

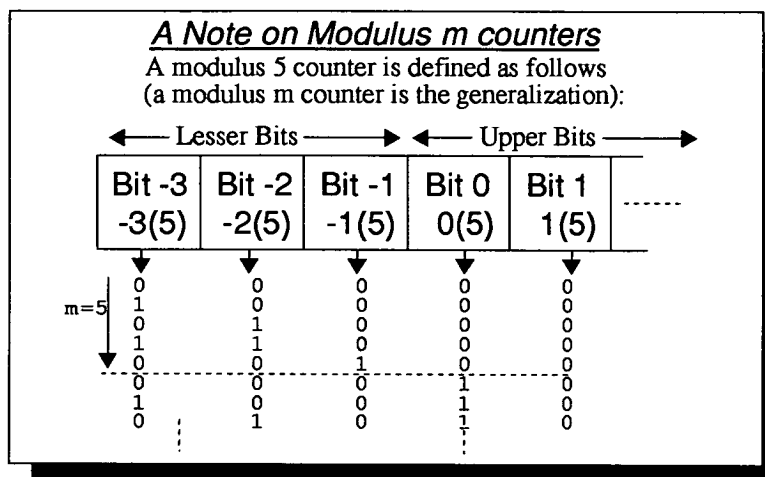
### **Bit Sequence Generator Structure**

To allow an understanding of the principles involved here, a view of the internal structure of a bit sequence generator (BSG) is necessary.

A BSG must obviously know which bit of the address word it is generating, and there is also a flag specifying whether the output of the BSG must be inverted (if the polarity bit of the repetition sequence = '1'). There are seven different types of BSG:

- a) SIMPLE: The output from bit b of a modulus m counter.
- b) EXORED: As above, but EXORED with the output from another BSG.
- c) LOGIC: Where no counter bit(s) can generate a bit sequence directly, the "random" bit sequence is saved, to be handed to a logic synthesis tool as a Truth Table output.
- d) HARDWIRED: Where a bit sequence consists of only '0's or '1's, the address bit should obviously be hardwired to logic '0' or '1'.
- e) ROM: When the type of bit sequence generator synthesised is more expensive than the cost of placing the bit sequence in the controller ROM, this type of BSG is used, which defines which of the ROM output bits produces this bit sequence. This will be used during the optimization stage.
- f) CLOCKED: When a bit sequence may be generated by using some other existing sequence to clock a JK flip-flop, then the information on this other address bit is held here.
- g) WHOLLY\_RANDOM: A BSG of this type is used if the bit sequence involved is

random and too long for the simple logic synthesis tool to handle.



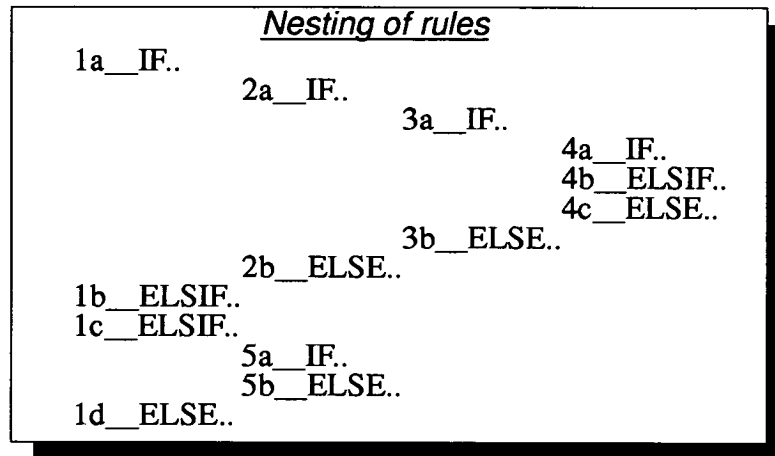
## The Rules

There follows a list of the rules which are used when collapsing the Repetition Sequence:

$L\_RS$  = length of the repetition sequence,  $m = L\_RS / 2$ .

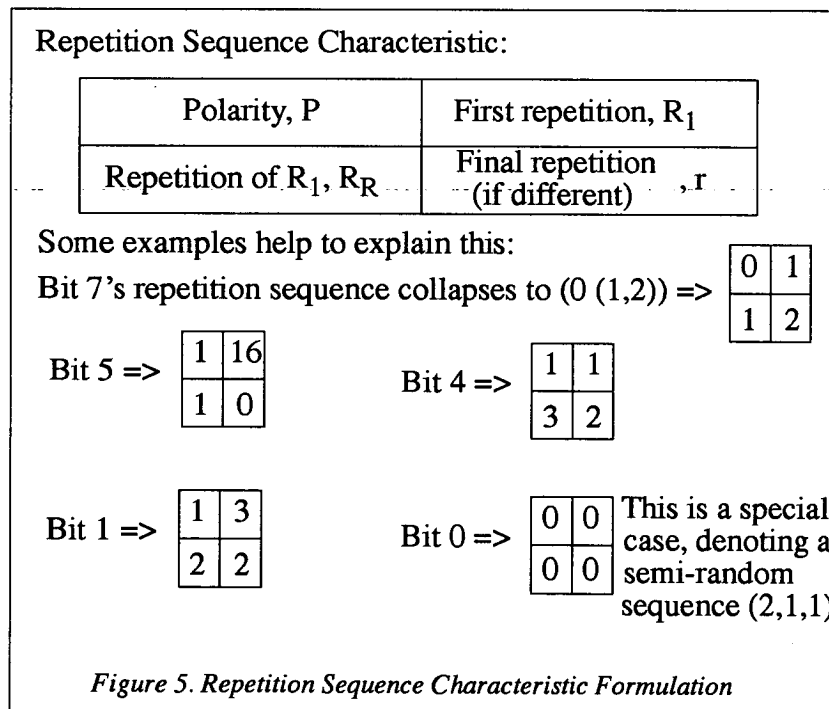
- 1a) IF  $L\_RS$  is even THEN Rule 2.
- 1b) IF  $L\_RS = 1$  THEN find\_BSG using remaining sequence.
- 1c) IF  $rep\_seq(2..(m-1)) = rep\_seq((m+1)..(L\_RS-1))$  AND  $rep\_seq(1) + rep\_seq(L\_RS) = rep\_seq(m)$  THEN Rule 5.
- 1d) find\_BSG using remaining sequence.
- 2a) IF  $rep\_seq(1..m) = rep\_seq((m+1)..L\_RS)$  THEN Rule 3.
- 2b) find\_BSG using remaining sequence.
- 3a) IF  $L\_RS/2$  is odd THEN Rule 4.
- 3b) Bisect the sequence and recurse using the first half.  
EG: Bit 1: (3,3,2,3,3,2,3,3,2,3,3,2)  $\Rightarrow$  (3,3,2,3,3,2).
- 4a) IF  $L\_RS/2 = 1$  THEN bisect the sequence and recurse.  
EG: Bit 5: (16,16)  $\Rightarrow$  (16).
- 4b) IF we can generate the corresponding bit sequence, as the repetition sequence stands, with a single counter bit, then find\_BSG using the current repetition sequence.
- 4c) The corresponding bit sequence is the result of EXORing a counter bit (found from  $L\_RS$ ), with whatever BSG is found by recursing using the first half of the sequence.  
EG: Bit 7: (1,3,3,3,2,1,3,3,3,2)  $\Rightarrow$  (1,3,3,3,2) EXOR 2(3).
- 5a) IF  $(L\_RS+1)/2$  is even THEN as for RULE 4c.
- 5b) Bisect the sequence and recurse.

EG: Bit 2 : (3,5,6,5,3) => (3,5,3).



### The Repetition Sequence Characteristic

Now the repetition sequence has been collapsed as far as possible, the remaining sequence is sent to be matched to a bit sequence generator (the find\_BSG routine mentioned above). The sequence is first converted into another format, to ease this matching, which consists of four parameters, as shown in Figure 5.



### Matching the Characteristic to a Bit Sequence Generator

Now we are ready to find the counter bit(s) which will produce the bit sequence, characterised as above. The counter bit is described by its bit number, b, and the modulus

of the counter,  $m$ . The polarity in the characteristic determines whether a counter bit's output should be inverted by a NOT gate.

### Finding the Counter Modulus

If there is a single repetition left in the repetition sequence, then its characteristic will look something like this, where,  $R_1$  is the remaining bit repetition length:

0	$R_1$
1	0

The modulus of the counter needed is found from  $R_1$  by finding the lowest odd factor of  $R_1$ , i.e.: By dividing it by 2 until an odd quotient is found. The number of times  $R_1$  can be divided is the bit number of an upper ( $\geq 0$ ) bit of the counter. For example, a

characteristic: 

0	16
1	0

 will be generated by bit 4 of a

modulus 1 counter (a simple binary counter).

However, if there is more than one repetition left, then the modulus is calculated as:

$$\text{modulus} \left( \begin{array}{|c|c|} \hline P & R_1 \\ \hline R_R & r \\ \hline \end{array} \right) = (R_1 * R_R) + r = m$$

i.e.: The sum of the remaining repetitions. This should represent the sequence generated by a lesser ( $<0$ ) bit of a modulus  $m$

counter. For example a characteristic: 

0	2
3	1

 will be

generated by bit(-2) of modulus7 counter.

If a random bit sequence is characterised, then for consistency its modulus is set to 0, as a flag. Also, if a repetition sequence remainder ( $r$ ) is greater than the first repetition, then it is possible that this may characterise a bit sequence generated by EXORing lesser bits of a counter, and these lesser bits can be found by expanding the characterised repetition sequence to its bit sequence, and then repeating the whole synthesis process for that sequence (Padding first, using a hint of 1). For example the

characteristic: 

1	1
3	2

 represents the bit sequence

10100, which is the result of EXORing sequences 10101 and 00001, which are produced by NOT.bit(-3) and bit(-1) of a modulus 5 counter, respectively.



### **Finding the Lesser Bit Number**

If, once the modulus has been found, it is greater than the first repetition, then a lesser bit of the counter is desired, and the correct bit is derived simply from the modulus, which determines how many lesser bits there will be, and from the first repetition, which should be a power-of-two.

### **Finding Clocked-type Bit Sequence Generators**

To find any address bit sequences which may be produced at the output of a JK flip-flop which is clocked using some other bit sequence of that address, a list of all repetition sequences with original length  $> 1$  is collected. This list is sorted so that the sequence with the shortest first repetition(s) comes first, and then the rest of the list is searched for a sequence which may be used as the clock.

This may find the same solutions as the main synthesis process, but can also find some surprisingly elegant solutions, which were not otherwise discovered. For example, address bits 1 and 2 in our working example, can be generated by using bit 0 to clock a pair of JK flip-flops.

### **Optimization of Address Generators**

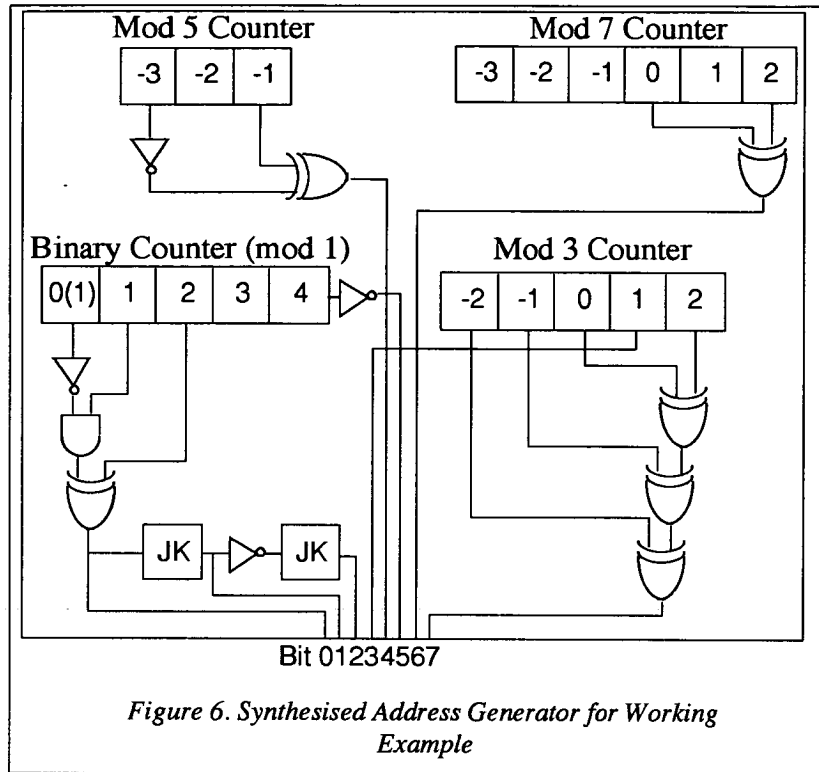
Although no optimization stage has yet been implemented, it is imagined that it will take a global list of all bit sequence generators, perhaps supplying several memories, and formulate a cost for each one. For example, if there are three users of a particular counter bit, then the individual cost of each of the three, is one-third of the cost of implementing the counter bit in hardware. These costs may be compared with the costs of embedding the bit sequences within a ROM, or perhaps just as one output from a combinatorial logic network.

Choosing the globally cheapest solution will most likely be done using some heuristics, since it is an NP-complete problem.

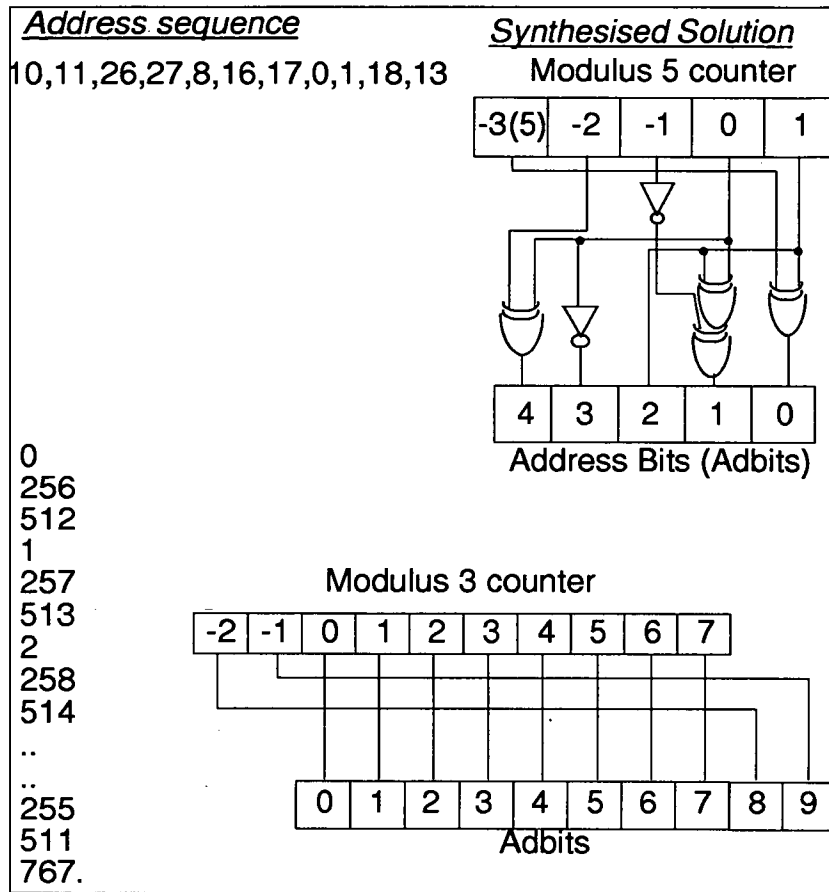
### **Solution to Working Example**

Shown below is the circuitry which was specified by the tool, when given the working

example:



### More Examples



Obviously the examples given in this paper needed to be rather simple, so that the address sequences could be fully specified in the limited space available. Real address sequences can range in length from that of the examples given here, to sequences containing tens of thousands of addresses, each with perhaps a dozen or more constituent address bits. Solutions synthesised for complex examples were found to match those constructed by hand.

### Conclusions and Further Work

This paper has presented a novel technique for the synthesis of address generation hardware from a specification of the address sequence to be generated. The method is extensible to almost any sequence generation problem where the sequence may exhibit some symmetry or regularity, for example on-chip ATPG or controller synthesis.

The results presented matched, or were better than, those designed by hand, and were completed in a fraction of the time. Typical run times range from a few seconds, for short sequences, to around one minute, for the longer sequences.

Although the tool is not suitable for all address generator synthesis problems, as part of a set of address generator synthesis tools, it can perform a much-needed and fairly complex task with the ease of an experienced designer, and perhaps better.

## Appendix B

The following pages show annotated extracts of output from AG2, sampled from the working example's bit generator synthesis process.

***Start of program:***

DRIVE FILE or SINGLE FILE (d/s) => d

DRIVE FILE NAME => Thesis\_all.drv

Information density? (Lots\_and\_lots | a\_Bit | Result\_only: l/b/r): b

***Memory and access sequence info:***

File name = Thesis\_mem1.wacseq

ADDRLISTSIZE (Bits) = 176

MODE= W

START\_AT= 0

STOP\_AT= 21

CLOCK= clock1

STROBE= strobe\_a

COMMS= input1

MAXADDRESS= 253

HINT= 0

***Start of synthesis:***

Trying Incrementor solution

***No incrementor-based solution found***

Second Pass

BIT 2( 3) EXORED\_WITH

***I.E.: Bit 2 of modulus 3 counter***

BIT 0( 3) EXORED\_WITH

BIT -1( 3) EXORED\_WITH

BIT -2( 3) => AdBit 7

Of address port 1, of memory Thesis\_mem1

BIT 1( 7) EXORED\_WITH

BIT 0( 7) => AdBit 6

Of address port 1, of memory Thesis\_mem1

NOT.BIT 4( 1) => AdBit 5

Of address port 1, of memory Thesis\_mem1

BIT -1( 5) EXORED\_WITH

NOT.BIT -3( 5) => AdBit 4

Of address port 1, of memory Thesis\_mem1

BIT 1( 3) => AdBit 3

Of address port 1, of memory Thesis\_mem1

$^{11}(0,0,0,1,1,1,1,1,0,0,0)(1,1,1,1,1)^{\wedge} \Rightarrow \text{AdBit } 2$

Of address port 1, of memory Thesis\_mem1

***I.E.: A bit sequence 11 bits long will produce the entire sequence if repeated.***

***The five extra bits are used as padding by the logic synthesis tool which requires a sequence length of a power of two.***

BIT 3( 1) EXORED\_WITH

$^8(0,0,0,1,1,1,0,0)^{\wedge} \Rightarrow \text{AdBit } 1$

Of address port 1, of memory Thesis\_mem1

BIT 2( 1) EXORED\_WITH

$^4(0,0,1,0)^{\wedge} \Rightarrow \text{AdBit } 0$

Of address port 1, of memory Thesis\_mem1

Third Pass

***To look for clocked type bit sequence generators:***

A\_BIT\_CLOCKED\_BY UNNOTTED OUTPUT FROM...

BIT 2( 1) EXORED\_WITH

(0,0,1,0)

=> AdBit 1

Of address port 1, of memory Thesis\_mem1

A\_BIT\_CLOCKED\_BY NOTTED OUTPUT FROM...  
A\_BIT\_CLOCKED\_BY UNNOTTED OUTPUT FROM...  
BIT 2( 1) EXORED\_WITH  
(0,0,1,0)  
=> AdBit 2  
Of address port 1, of memory Thesis\_mem1

***Move onto next access sequence:***

File name = Thesis\_mem1.wracseq  
ADDRLISTSIZE (Bits) = 180  
MODE= WR  
START\_AT= 22  
STOP\_AT= 57  
CLOCK= clock1

STROBE= strobe\_a

COMMS= input2/adder1/mult1

MAXADDRESS= 18  
HINT= 0

---

Trying Incrementor solution  
Incrementor type solution found

***Last access sequence to be handled:***

File name = Thesis\_csig21.cseq

ADDRLISTSIZE (Bits) = 20  
MODE= C  
START\_AT= 22  
STOP\_AT= 41  
CLOCK= clock1

STROBE= dummy

COMMS= mux1/mux5/mux10

MAXADDRESS= 1

HINT= 0

*No Incrementor-based solution possible for a single bit sequence*

Second Pass

BIT 4( 1) EXORED\_WITH

$\wedge 16(0,1,1,0,1,1,1,0,1,1,0,0,0,1,1,0)\wedge \Rightarrow$  Thesis\_csig21

Third Pass

*No clocked type bit generators found.*

*Any bit sequence may be generated by using a ROM to store the sequence in, or by a SINGLE other method. Any clocked-type bit sequence generators will have a second non-ROM based solution, which must be either discarded or enforced, to allow the costing routines to function.*

+++++

Address bit 1 of Port 1 of memory Thesis\_mem1.

From click 0 to click 21

There is a clocked type bit generator, costing 28 units of area.

This entails a skew of 100ns on this address line.

The alternative bit generator could cost 117 units of area.

Do you need to see the latter?:

n

Do you want the clocked type bit generator?:

y

+++++

Address bit 2 of Port 1 of memory Thesis\_mem1.

From click 0 to click 21

There is a clocked type bit generator, costing 28 units of area.

This entails a skew of 200ns on this address line.

The alternative bit generator could cost 117 units of area.

Do you need to see the latter?:

y

Memory = Thesis\_mem1  
. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1  
Acseq Start..Finish = 0.. 21 Mode = W  
Access ID = 1 Port 1 (Type RW)  
Bit Generator Element = (0,0,0,1,1,1,1,1,0,0,0)  
(1,1,1,1,1) => Adbit 2  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 117 0 0 0

Do you want the clocked type bit generator?:

y

+++++

***Start of initial costings:***

Thesis\_csig21  
Acseq Start..Finish = 22.. 41 Mode = C  
Access ID = 4 Port 1  
Bit Generator Element = (0,1,1,0,1,1,1,0,1,1,0,0,0,1,1,0)  
=> Thesis\_csig21  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 117 200 0 0

Thesis\_csig21  
Acseq Start..Finish = 22.. 41 Mode = C  
Access ID = 4 Port 1  
Bit Generator Element = BIT 4( 1) EXORED WITH THE ABOVE =>  
Thesis\_csig21  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 140 -1 0 0  
***I.E.: The total non-ROM based cost of producing csig21 is 257 units.***

Memory = Thesis\_mem1  
. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1  
***Sizes are: Declared size, Rounded size, Dec. width, Rnd. width, X-dim, Y-dim***  
Acseq Start..Finish = 22.. 57 Mode = WR  
Access ID = 2 Port 1 (Type RW)  
Bit Generator Element = BIT 4( 20), Preset to 5  
***I.E.: Bit 4 of a modulus 20 incrementor (one which resets at 20)***  
An incrementor, INC = 3  
With clock gated by gating\_sig3 => Adbit 4  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 0 77 0 69



Memory = Thesis\_mem1  
. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1  
Acseq Start..Finish = 22.. 57 Mode = WR  
Access ID = 2 Port 1 (Type RW)  
Bit Generator Element = BIT 3( 20), Preset to 5  
An incrementor, INC = 3  
With clock gated by gating\_sig3 => Adbit 3  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 0 77 0 69

Memory = Thesis\_mem1  
. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1  
Acseq Start..Finish = 22.. 57 Mode = WR  
Access ID = 2 Port 1 (Type RW)  
Bit Generator Element = BIT 2( 20), Preset to 5  
An incrementor, INC = 3  
With clock gated by gating\_sig3 => Adbit 2  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 0 77 0 69

Memory = Thesis\_mem1  
. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1  
Acseq Start..Finish = 22.. 57 Mode = WR  
Access ID = 2 Port 1 (Type RW)  
Bit Generator Element = BIT 1( 20), Preset to 5  
An incrementor, INC = 3  
With clock gated by gating\_sig3 => Adbit 1  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 0 77 0 69

Memory = Thesis\_mem1  
. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1  
Acseq Start..Finish = 22.. 57 Mode = WR  
Access ID = 2 Port 1 (Type RW)  
Bit Generator Element = BIT 0( 20), Preset to 5  
An incrementor, INC = 3  
With clock gated by gating\_sig3 => Adbit 0  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 0 77 0 69

Memory = Thesis\_mem1

. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1

Acseq Start..Finish = 0.. 21 Mode = W

Access ID = 1 Port 1 (Type RW)

Bit Generator Element = A\_BIT\_CLOCKED\_BY NOTTED OUTPUT FROM  
ADBIT 1's BIT GENERATOR => Adbit 2

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 0 44 28 0

Memory = Thesis\_mem1

. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1

Acseq Start..Finish = 0.. 21 Mode = W

Access ID = 1 Port 1 (Type RW)

Bit Generator Element = A\_BIT\_CLOCKED\_BY UNNOTTED OUTPUT  
FROM ADBIT 0's BIT GENERATOR => Adbit 1

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 0 44 28 0

Memory = Thesis\_mem1

. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1

Acseq Start..Finish = 0.. 21 Mode = W

Access ID = 1 Port 1 (Type RW)

Bit Generator Element = (0,0,1,0)

=> Adbit 0

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 61 44 0 0

Memory = Thesis\_mem1

. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1

Acseq Start..Finish = 0.. 21 Mode = W

Access ID = 1 Port 1 (Type RW)

Bit Generator Element = BIT 2( 1) EXORED WITH THE ABOVE => Adbit 0

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 84 -1 0 0

Memory = Thesis\_mem1

. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1

Acseq Start..Finish = 0.. 21 Mode = W

Access ID = 1 Port 1 (Type RW)

Bit Generator Element = BIT 1( 3) => Adbit 3

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 28 44 0 0

Memory = Thesis\_mem1  
. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1  
Acseq Start..Finish = 0.. 21 Mode = W  
Access ID = 1 Port 1 (Type RW)  
Bit Generator Element = NOT.BIT -3( 5) => Adbit 4  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 28 44 0 0

Memory = Thesis\_mem1  
. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1  
Acseq Start..Finish = 0.. 21 Mode = W  
Access ID = 1 Port 1 (Type RW)  
Bit Generator Element = BIT -1( 5) EXORED WITH THE ABOVE => Adbit 4  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 56 -1 0 0

Memory = Thesis\_mem1  
. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1  
Acseq Start..Finish = 0.. 21 Mode = W  
Access ID = 1 Port 1 (Type RW)  
Bit Generator Element = NOT.BIT 4( 1) => Adbit 5  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 28 44 0 0

Memory = Thesis\_mem1  
. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1  
Acseq Start..Finish = 0.. 21 Mode = W  
Access ID = 1 Port 1 (Type RW)  
Bit Generator Element = BIT 0( 7) => Adbit 6  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 112 44 0 0

Memory = Thesis\_mem1  
. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1  
Acseq Start..Finish = 0.. 21 Mode = W  
Access ID = 1 Port 1 (Type RW)  
Bit Generator Element = BIT 1( 7) EXORED WITH THE ABOVE => Adbit 6  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 28 -1 0 0

Memory = Thesis\_mem1  
. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1  
Acseq Start..Finish = 0.. 21 Mode = W  
Access ID = 1 Port 1 (Type RW)  
Bit Generator Element = BIT -2( 3) => Adbit 7  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 28 44 0 0

Memory = Thesis\_mem1  
. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1  
Acseq Start..Finish = 0.. 21 Mode = W  
Access ID = 1 Port 1 (Type RW)  
Bit Generator Element = BIT -1( 3) EXORED WITH THE ABOVE => Adbit 7  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 28 -1 0 0

Memory = Thesis\_mem1  
. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1  
Acseq Start..Finish = 0.. 21 Mode = W  
Access ID = 1 Port 1 (Type RW)  
Bit Generator Element = BIT 0( 3) EXORED WITH THE ABOVE => Adbit 7  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 28 -1 0 0

Memory = Thesis\_mem1  
. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1  
Acseq Start..Finish = 0.. 21 Mode = W  
Access ID = 1 Port 1 (Type RW)  
Bit Generator Element = BIT 2( 3) EXORED WITH THE ABOVE => Adbit 7  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 28 -1 0 0

Do you want to interact?: n

***Start of iterative cost optimisation:***

+++++

Old cost = 1110

*The sum of cheapest costs.*

ROM\_based\_cost = 937

*As ROM-based as possible.*

Other cost = 1195

*As non-ROM-based as possible.*

***ROM\_based\_cost < Other\_cost so:***

Forcing the following into ROM

Memory = Thesis\_mem1  
. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1  
Acseq Start..Finish = 0.. 21 Mode = W  
Access ID = 1 Port 1 (Type RW)  
Bit Generator Element = (0,0,1,0)  
=> Adbit 0  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 61 44 0 0

New cost = 1126 *Sum of cheapest costs.*

++++  
++++

Old cost = 1042

ROM\_based\_cost = 937

Other cost = 1122

Forcing the following into ROM

Memory = Thesis\_mem1

. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1

Acseq Start..Finish = 0.. 21 Mode = W

Access ID = 1 Port 1 (Type RW)

Bit Generator Element = BIT 0( 7) => Adbit 6

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 112 44 0 0

---

New cost = 1154

++++

.  
. .  
. .  
. .

++++

Old cost = 905

ROM\_based\_cost = 937

Other cost = 905

***ROM\_based\_cost > Other\_cost so:***

Forcing the following away from ROM

Memory = Thesis\_mem1

. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1

Acseq Start..Finish = 0.. 21 Mode = W

Access ID = 1 Port 1 (Type RW)

Bit Generator Element = A\_BIT\_CLOCKED\_BY NOTTED OUTPUT FROM  
ADBIT 1's BIT GENERATOR => Adbit 2

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 0 44 28 0

New cost = 920

++++  
++++

Old cost = 920

ROM\_based\_cost = 942

Other cost = 920

Forcing the following away from ROM

Memory = Thesis\_mem1

. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1

Acseq Start..Finish = 0.. 21 Mode = W

Access ID = 1 Port 1 (Type RW)

Bit Generator Element = A\_BIT\_CLOCKED\_BY UNNOTTED OUTPUT  
FROM ADBIT 0's BIT GENERATOR => Adbit 1

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 0 47 28 0

New cost = 940

++++  
++++

Old cost = 940

ROM\_based\_cost = 947

Other cost = 940

Forcing the following away from ROM

Memory = Thesis\_mem1

. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1

Acseq Start..Finish = 22.. 57 Mode = WR

Access ID = 2 Port 1 (Type RW)

Bit Generator Element = BIT 4( 20), Preset to 5

An incrementor, INC = 3

With clock gated by gating\_sig3 => Adbit 4

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 0 77 0 69

New cost = 940

++++

```
+++++
Old cost = 940
ROM_based_cost = 983
Other cost = 940
Forcing the following away from ROM
Memory = Thesis_mem1
. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1
Acseq Start..Finish = 22.. 57 Mode = WR
Access ID = 2 Port 1 (Type RW)
Bit Generator Element = BIT 3( 20), Preset to 5
An incrementor, INC = 3
With clock gated by gating_sig3 => Adbit 3
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 0 88 0 69
```

New cost = 940

```
+++++
```

.  
.
.  
.

```
+++++
```

```
Old cost = 907
ROM_based_cost = 907
Other cost = 907
New cost = 907
```

```
+++++
```

***End of iterative optimisation.***

***The following describes the binding to a specific form of generation:***

Thesis\_csig21

```
Acseq Start..Finish = 22.. 41 Mode = C
Access ID = 4 Port 1
Bit Generator Element = (0,1,1,0,1,1,1,0,1,1,0,0,0,1,1,0)
=> Thesis_csig21
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): -1 200 -1 -1
```

Memory = Thesis\_mem1

. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1

Acseq Start..Finish = 22.. 57 Mode = WR

Access ID = 2 Port 1 (Type RW)

Bit Generator Element = BIT 4( 20), Preset to 5

An incrementor, INC = 3

With clock gated by gating\_sig3 => Adbit 4

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 0 -1 0 69

Memory = Thesis\_mem1

. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1

Acseq Start..Finish = 22.. 57 Mode = WR

Access ID = 2 Port 1 (Type RW)

Bit Generator Element = BIT 3( 20), Preset to 5

An incrementor, INC = 3

With clock gated by gating\_sig3 => Adbit 3

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 0 -1 0 69

Memory = Thesis\_mem1

. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1

Acseq Start..Finish = 22.. 57 Mode = WR

Access ID = 2 Port 1 (Type RW)

Bit Generator Element = BIT 2( 20), Preset to 5

An incrementor, INC = 3

With clock gated by gating\_sig3 => Adbit 2

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 0 -1 0 69

Memory = Thesis\_mem1

. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1

Acseq Start..Finish = 22.. 57 Mode = WR

Access ID = 2 Port 1 (Type RW)

Bit Generator Element = BIT 1( 20), Preset to 5

An incrementor, INC = 3

With clock gated by gating\_sig3 => Adbit 1

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 0 -1 0 69



Memory = Thesis\_mem1

. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1

Acseq Start..Finish = 22.. 57 Mode = WR

Access ID = 2 Port 1 (Type RW)

Bit Generator Element = BIT 0( 20), Preset to 5

An incrementor, INC = 3

With clock gated by gating\_sig3 => Adbit 0

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 0 -1 0 69

Memory = Thesis\_mem1

. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1

Acseq Start..Finish = 0.. 21 Mode = W

Access ID = 1 Port 1 (Type RW)

Bit Generator Element = A\_BIT\_CLOCKED\_BY NOTTED OUTPUT FROM  
ADBIT 1's BIT GENERATOR => Adbit 2

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 0 -1 28 0

Memory = Thesis\_mem1

. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1

Acseq Start..Finish = 0.. 21 Mode = W

Access ID = 1 Port 1 (Type RW)

Bit Generator Element = A\_BIT\_CLOCKED\_BY UNNOTTED OUTPUT  
FROM ADBIT 0's BIT GENERATOR => Adbit 1

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 0 -1 28 0

***All remaining bit sequence generators are to be ROM-based:***

Memory = Thesis\_mem1

. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1

Acseq Start..Finish = 0.. 21 Mode = W

Access ID = 1 Port 1 (Type RW)

Bit Generator Element = (0,0,1,0)

=> Adbit 0

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): -1 51 -1 -1

Memory = Thesis\_mem1  
. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1  
Acseq Start..Finish = 0.. 21 Mode = W  
Access ID = 1 Port 1 (Type RW)  
Bit Generator Element = BIT 1( 3) => Adbit 3  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): -1 51 -1 -1

Memory = Thesis\_mem1  
. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1  
Acseq Start..Finish = 0.. 21 Mode = W  
Access ID = 1 Port 1 (Type RW)  
Bit Generator Element = NOT.BIT -3( 5) => Adbit 4  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): -1 51 -1 -1

Memory = Thesis\_mem1  
. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1  
Acseq Start..Finish = 0.. 21 Mode = W  
Access ID = 1 Port 1 (Type RW)  
Bit Generator Element = NOT.BIT 4( 1) => Adbit 5  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): -1 51 -1 -1

Memory = Thesis\_mem1  
. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1  
Acseq Start..Finish = 0.. 21 Mode = W  
Access ID = 1 Port 1 (Type RW)  
Bit Generator Element = BIT 0( 7) => Adbit 6  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): -1 51 -1 -1

Memory = Thesis\_mem1  
. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1  
Acseq Start..Finish = 0.. 21 Mode = W  
Access ID = 1 Port 1 (Type RW)  
Bit Generator Element = BIT -2( 3) => Adbit 7  
COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): -1 51 -1 -1

***Any ROM-based bit sequence generators must now have the ROMs synthesised, along with their own address generators:***

Commencing ROM extraction

***The ROMs have their contents constructed as a conglomeration of their constituent bit sequences.***

***ROM access sequence info:***

ADDRLISTSIZE (Bits) = 110

MODE= R

START\_AT= 0

STOP\_AT= 21

CLOCK= clock1

STROBE= strobe\_a

COMMS= Thesisp1/Thesisp1/Thesisp1/Thesisp1/Thesisp1/Thesisp1/

***I.E.: 6 bits of Thesis\_mem's address port 1.***

MAXADDRESS= 21

HINT= 0

Trying Incrementor solution

Incrementor type solution found

***The incrementor based solution should always be found, as the ROM access sequence is constructed as an incremental sequence.***

ADDRLISTSIZE (Bits) = 100

MODE= R

START\_AT= 22

STOP\_AT= 41

CLOCK= clock1

STROBE= dummy

COMMS= Thesis\_csig21/

MAXADDRESS= 19

HINT= 0

Trying Incrementor solution

Incrementor type solution found

+++++

***ROM address generator information:***

Memory = ADDRESS\_ROM1000

Contents:

0  
1  
1  
0  
1  
1  
1  
0  
1  
1  
0  
0  
0  
1  
1  
0  
1  
0  
0  
1

. Sizes (DS, RS, DW, RW, X, Y) = 20, 1, 1, 0, 1, 1

Acseq Start..Finish = 22.. 41 Mode = R

Access ID = 1001 Port 1

Bit Generator Element = BIT -1( 32), Preset to 0

With clock gated by gating\_sig6 => Adbit 4

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 19 -1 0 -1

.  
.  
.

Memory = ADDRESS\_ROM1000

Contents: *As above.*

. Sizes (DS, RS, DW, RW, X, Y) = 20, 1, 1, 0, 1, 1

Acseq Start..Finish = 22.. 41 Mode = R

Access ID = 1001 Port 1

Bit Generator Element = BIT -5( 32), Preset to 0

With clock gated by gating\_sig6 => Adbit 0

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 19 -1 0 -1

.  
.  
.

Memory = ADDRESS\_ROM1002

Contents:

12  
40  
45  
40  
9  
13  
10  
63  
58  
58  
31  
26  
61  
25  
24  
29  
48  
52  
51  
18  
23  
3

. Sizes (DS, RS, DW, RW, X, Y) = 22, 8, 6, 0, 8, 1

Acseq Start..Finish = 0.. 21 Mode = R

Access ID = 1003 Port 1

Bit Generator Element = BIT -1( 32), Preset to 0

With clock gated by gating\_sig5 => Adbit 4

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 21 -1 0 -1

.....

Memory = ADDRESS\_ROM1002

Contents: *As above.*

. Sizes (DS, RS, DW, RW, X, Y) = 22, 8, 6, 0, 8, 1

Acseq Start..Finish = 0.. 21 Mode = R

Access ID = 1003 Port 1

Bit Generator Element = BIT -5( 32), Preset to 0

With clock gated by gating\_sig5 => Adbit 0

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 21 -1 0 -1

***Final bindings of bit sequence generators to hardware:***

Thesis\_csig21

Acseq Start..Finish = 22.. 41 Mode = C

Access ID = 4 Port 1

Bit Generator Element = Bit 0 of some ROM called ADDRESS\_ROM1000

=> Thesis\_csig21

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 0 -1 0 0

Memory = Thesis\_mem1

. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1

Acseq Start..Finish = 22.. 57 Mode = WR

Access ID = 2 Port 1 (Type RW)

Bit Generator Element = BIT 4( 20), Preset to 5

An incrementor, INC = 3

With clock gated by gating\_sig3 => Adbit 4

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 0 -1 0 69

.

.

.

Memory = Thesis\_mem1

. Sizes (DS, RS, DW, RW, X, Y) = 254, 256, 8, 8, 256, 1

Acseq Start..Finish = 0.. 21 Mode = W

Access ID = 1 Port 1 (Type RW)

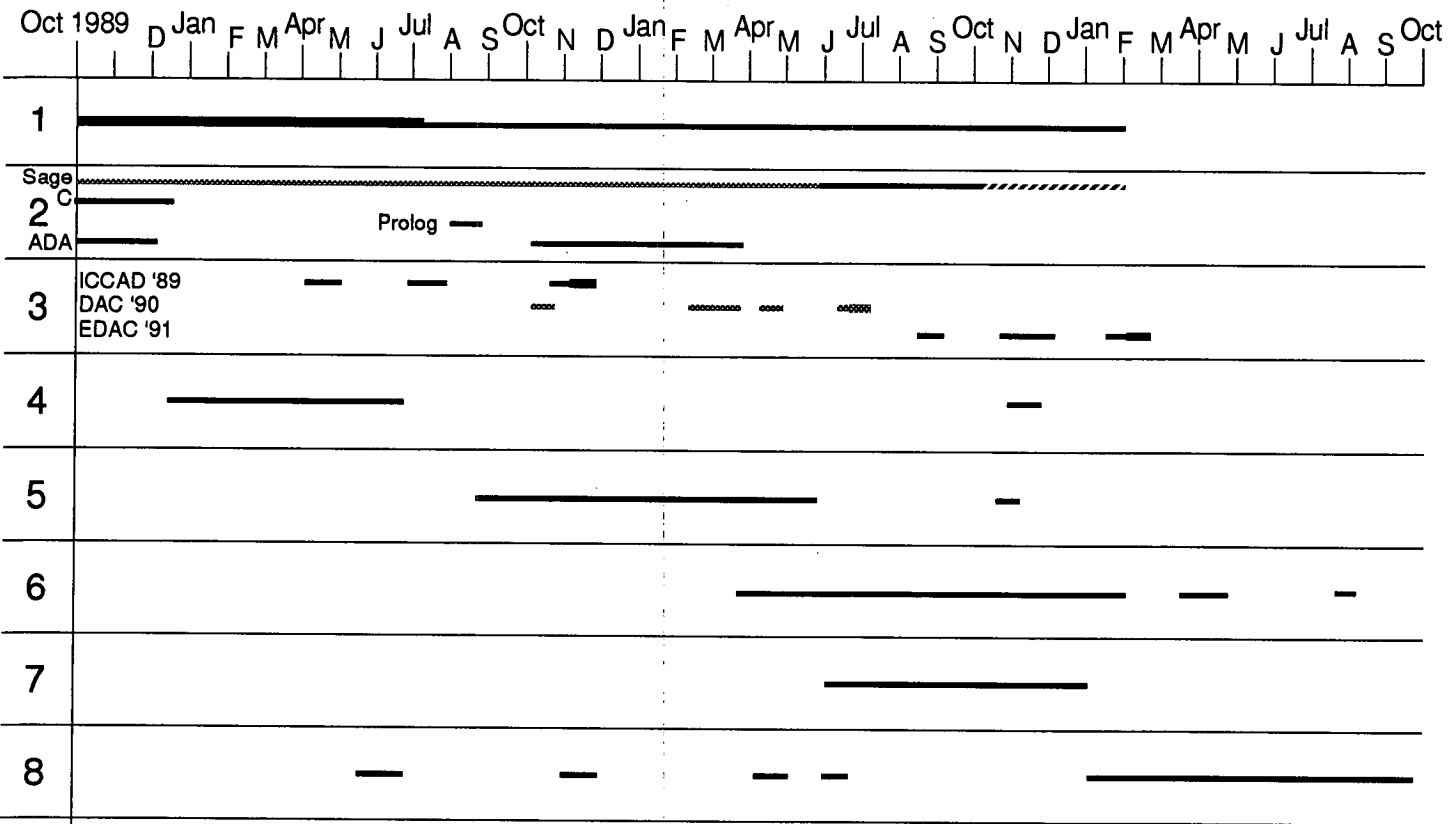
Bit Generator Element = Bit 5 of some ROM called ADDRESS\_ROM1002

=> Adbit 7

COSTS (COUNTER, ROM, CLOCKED, INCREMENTOR): 0 -1 0 0

## Appendix C

Shown below is a breakdown of the work carried out over the three year period of this thesis.



**Key:** 1. Literature Survey  
 2. Language and System Familiarisation  
 3. Conference Paper Prep. and Presentation  
 4. C Programming and Development (AG1)  
 5. Prolog Programming (MC²)  
 6. Ada Programming (AG2)  
 7. Ada Programming (SAGE)  
 8. Other Documentation

## Appendix D

### AG1 User Guide

There are four specific ways in which to introduce the address sequence into the synthesis algorithm of AG1. The first requires a software description of the address sequence, and the second method is by using a graphical entry method. The two remaining data-entry formats are specifically aimed at very random sequences, for immediate logic synthesis.

For the first data-entry method, using the '-s' command-line option, a function (doloop/0) is called, and this should fill up the global address sequence with values calculated by a set of loops, of perhaps with values explicitly declared within doloop/0. This function should also return the value equal to half the address sequence length, which *must be a power of two*. [Put eg of this in agmain4/doloop]. The function should exist within the same file as the AG1 source code, necessitating full recompilation for each different sequence, but perhaps could be compiled in a separate file in the future.

The second option (-g) is to use the built-in graphical entry tool, which allows an addressing *pattern* to be laid out, taking a macro-oriented, hierarchical approach if necessary, on a two-dimensional representation of memory space.

Sequences created in this way may be saved to file in a format compatible to AG2, described in Section 7.4.1.

The graphical entry method commences with a prompt for the unique name of the address pattern to be specified. There are three reserved names:

'\*' is the name of the *smallest* pattern to be defined in a hierarchical description, within which the order of accesses *does not matter*;

'.' is the name given to the entire memory space, as the last stage of the description;

'memel' is the third reserved name, relating to a single memory element: A pattern of dimensions  $x = 1, y = 1$ .

Next the dimensions of the pattern are required, which should be within the limits:  $1 \leq x \leq 32, 1 \leq y \leq 16$ . The user is then prompted for the name of the pattern which is to be mapped to each coordinate in the present pattern. Usually the smallest pattern, '\*', will be defined as an array of 'memel's. The system then reports the *actual* number of



memory elements in the present pattern, displays it as an array of points on the screen, and waits for commands to move around the pattern, selecting the points in the correct order. If the pattern name is '\*', then any coordinates selected during its specification (including those in any hierarchically lower patterns, used to construct '\*') are sorted into ascending order, since this should result in the simplest possible addressing scheme. Otherwise the selected coordinates are left in the order selected.

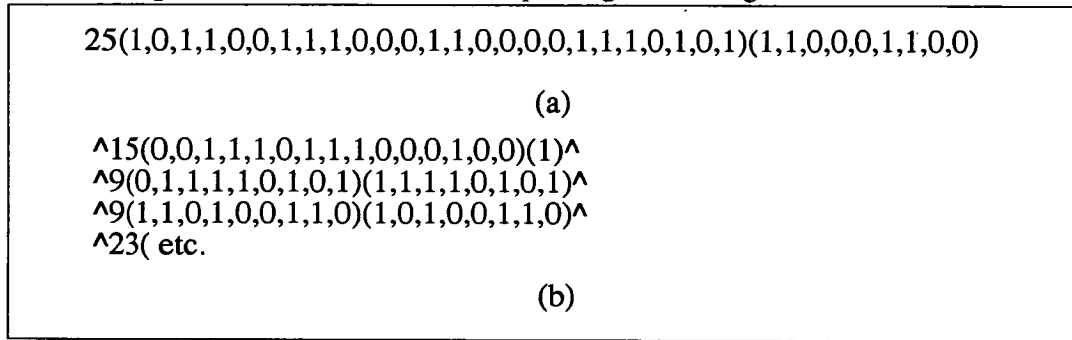
The maximum dimensions of any pattern ( $x \leq 32$ ,  $y \leq 16$ ) are necessarily limited by the ability of the user to select the desired coordinates from a large array, and by the present simplicity of the interface. Given a mouse-driven selection procedure, on a high-resolution display, far higher limits could be placed on the patterns' size, at the risk of wasted effort through loss of hierarchy.

Neither the address sequence length, nor its constituent patterns' dimensions need to be powers of two, if the sequence is simply to be saved to a file for use elsewhere. If, however, the sequence is to be handed to the main algorithm in AG1, described in Section 4.3.2, then the address sequence should definitely have length  $2^n$ , and the value  $2^{n-1}$  is then returned to AG1.

A third command-line option, -l, allows a single sequence of binary values to be loaded from file, along with the length of that part of it which should have logic generated for it, as illustrated in Figure D.1a. The sequence may contain '()'s, ','s, and carriage returns, and must have total length  $2^n$ . If this length is longer than actually required, the length actually needed corresponds to the modulus of the binary counter which will be used to drive the logic, which in turn will produce the sequence.

The final option for AG1 is -L. This allows a number of different binary sequences, of perhaps different lengths (but still some powers of two), and delimited by '^'s, to be accessed by the main algorithm. The first sequence encountered in a serial search of the

input file, will be produced by the LSB of the synthesised address generator, and so on to the last sequence - the MSB. An example is given in Figure D.1b.



*Figure D.1 a) Specification of a binary sequence for AG1.  
b) Specification of several binary sequences together.*

## **AG2 User Guide**

AG2 is a very simple tool to use. Two options are available on the specification of the file(s) containing the access sequences: “s” loads the access sequence from a single file, and later enquires as to the width of the corresponding memory’s data; “d” prompts for the name of a drive file which should contain the name of each access sequence file to be examined, followed by the corresponding memory’s data width. If a control bit sequence is to be generated, then no memory exists and the width should be set to 1. The name of the access sequence file is important also, since any characters up to the first ‘.’ will be taken as the name of the corresponding memory. A good hint is to use:

mem\_name.c/r/wacseq[n],

where a ‘c’ denotes a control bit sequence and ‘r’ and/or ‘w’ denote the access mode of an access sequence. ‘n’ would be an integer if more than one access sequence of a single mode was present for a given memory.

The next prompt allows the output information to be filtered, before the algorithms start to work in earnest, and you sit back and watch! Depending on the level of information requested the following may, or will (denoted by a ‘!’), appear.

! Firstly, the name of the current access sequence file will appear, followed by information echoed or derived from that file, including the size of the address sequence, in bits, and if this is too large (> MAXADDRLISTSIZE) then the decision to commence a bitwise investigation of the sequence is reported. An example of the information to expect is shown below.

DRIVE FILE or SINGLE FILE (d/s) => d  
DRIVE FILE NAME => all3.drv  
Information density? (Lots\_and\_lots | a\_Bit | Result\_only: l/b/r): r

File name = h0test1.acseq

ADDRLISTSIZE (Bits) = 90  
MODE= R  
START\_AT= 12  
STOP\_AT= 29  
CLOCK= clock1  
STROBE= strobe\_a  
COMMS= comms\_name  
MAXADDRESS= 30  
HINT= 0

The first algorithm to run, in a non-bitwise approach, looks for incremental/decremental sequences, which may have irregular timings for these increments/decrements, and reports back if such a situation exists.

If not, then a second pass is tried on the sequence, with a different algorithm, which looks for bit sequences generated by (a collection of) counter bits. Details on the generation of each bit sequence may then be shown, and then a third pass of the access sequence is made by an algorithm which inspects the bit sequences in the hope of finding some which may be generated by using another bit sequence to clock a flip-flop. This is how a ripple counter may be constructed, but the skew on the outputs may be too large to handle. Again, any new information may be printed out, before the final stage commences. Some annotated details of the information which will possibly appear are given below.

Trying Incrementer solution

*None was found*

Second Pass

BIT -1( 3) => AdBit 4

*Means "Bit -1 of*

Of address port 1, of memory h0test1

*modulus 3 counter"*

BIT 1( 3) EXORED\_WITH

BIT 0( 3) => AdBit 3

Of address port 1, of memory h0test1

NOT.BIT -2( 5) => AdBit 2

Of address port 1, of memory h0test1

BIT -3( 7) => AdBit 1

Of address port 1, of memory h0test1

BIT -1( 5) EXORED\_WITH

BIT -3( 5) => AdBit 0

Of address port 1, of memory h0test1

Third Pass

Firstly the details of every possible bit sequence generator for each bit sequence will be displayed, along with initialised costs for each one, determined using costing functions based on area, usage and control overheads.

***/\* Clocked bit sequence generator \*/***

Memory = inc\_test3b ***Declared + Rounded Size and Width of Memory***  
 . Sizes (DS, RS, DW, RW, X, Y) = 25, 32, 30, 32, 32, 1  
 Acseq Start..Finish = 0.. 19 Mode = W  
 Access ID = 12 Port 1 (Type W)  
 Bit Generator Element = NOT.A\_BIT\_CLOCKED\_BY NOTTED OUTPUT FROM  
 ADBIT 0's BIT GENERATOR => Adbit 1  
 COSTS (COUNTER, ROM, CLOCKED, INCREMENTER): 0 -1 28 0

***/\* Exored combination of counter bit and 'random' sequence \*/***

Memory = inc\_test3b .  
 Sizes (DS, RS, DW, RW, X, Y) = 25, 32, 30, 32, 32, 1  
 Acseq Start..Finish = 0.. 19 Mode = W  
 Access ID = 12 Port 1 (Type W)  
 Bit Generator Element = (1,0,0,1,1,0,0,0,1,1,1,1,0) (0,0,0) => Adbit 1  
 COSTS (COUNTER, ROM, CLOCKED, INCREMENTER): 61 56 0 0

***-1 prohibits  
ROM-based  
soln. for this  
bit gen'r.***

***'Random' bit sequence, to be  
generated using a counter + logic.  
(Only first section to be actually gen'd)***

Memory = inc\_test3b .

Sizes (DS, RS, DW, RW, X, Y) = 25, 32, 30, 32, 32, 1

Acseq Start..Finish = 0.. 19 Mode = W

Access ID = 12 Port 1 (Type W)

Bit Generator Element = BIT 0( 13) EXORED WITH ^ => Adbit 1

COSTS (COUNTER, ROM, CLOCKED, INCREMENTER): 28 -1 0 0

***Means Exor with  
generator above***

***/\* Incrementer type bit sequence generator with preset and reset at 27 \*/***

Memory = inc\_test3 . Sizes (DS, RS, DW, RW, X, Y) = 25, 32, 29, 32, 32, 1

Acseq Start..Finish = 0.. 20 Mode = W

Access ID = 10 Port 1 (Type W)

Bit Generator Element = BIT 4( 27), Preset to 6

An incrementer, INC = 3

With clock gated by gating\_sig11 => Adbit 4

COSTS (COUNTER, ROM, CLOCKED, INCREMENTER): 0 50 0 62

Optimisation is then based on finding the best generation method for each bit sequence, in a global context. This involves choosing between adder/counter-based and ROM-based solutions, and perhaps a clocked flip-flop approach. By grouping otherwise expensive bit sequences into a ROM, the expense of the ROM's creation may be justified, and judging from the comparative costs of a wholly ROM-based and a wholly otherwise approach, along with the cheapest possible cost overall (without taking ROM sharing into account), bit sequences are selected in turn to be bound to a single generation method, until a globally good solution is found.

Once every bit sequence has a definite generator, any ROM-type bit sequence generators then have the memory and access sequence information constructed for that ROM, before it is handed back to the mouth of the tool, where the whole process is repeated to synthesise the address ROMs' own address generators, which are reported in the same way. Finally the total cost is displayed, followed by the details of all bit sequence generators, including those for accessing address ROMs. The construction of the netlist of bit sequence generating components and memory address / control bit destinations, remains a manual task, but a simple schematic is not difficult to produce.

## **MC<sup>2</sup> User Guide**

This is probably the simplest tool to use, in that little or no interaction is required, and the output is to file, for easy browsing.

The system comes in three main parts: *msyn*, *acreqs* and *acgen*. To run the first stage, enter Prolog and consult the files *msyn*, *stdlib* and the file containing the database of schedule information. *Msyn* is then run with the goal *go(Cstep, Sortmethod, Outfile1)*, where *Cstep* is an integer within the range of the current schedule and *Sortmethod* is either 'e' or 'h' - Easiest or Hardest-first method of synthesis.

Once *msyn* has been completed, the next stage is to enter Prolog again, this time consulting *acreqs* only. This extracts all address and control generation requirements from the data path defined in *Outfile1* and using the goal *go(Outfile1, Outfile2)*, these requirements will be stored in *Outfile2*. The third part of the process necessitates entering Prolog once more, this time consulting the file *acgen* which should be run with *go(Outfile2, Outfile3)*, and this produces the final bindings of data to memory locations, perhaps using the interactive mode to specify more than the minimum required number of memory elements in any RAM.

The data in *Outfile3* explains the address and control requirements of each memory and control wire on a bitwise basis, and this may be translated to a format compatible with AG2 by using the program *tlate*. This should be consulted and run with the goal *go(Outfile3)* to produce a set of access sequence files in the correct format.

## Appendix E

This appendix presents the address generators synthesised by AG2 for all the examples described in Chapter 6. The first is for the digital wave filter, and then Figure E.2 shows the address generator for the FDCT example. Figure E.3 illustrates the generator for the FIR filter and finally Figure E.4 shows that for the differential equation example.

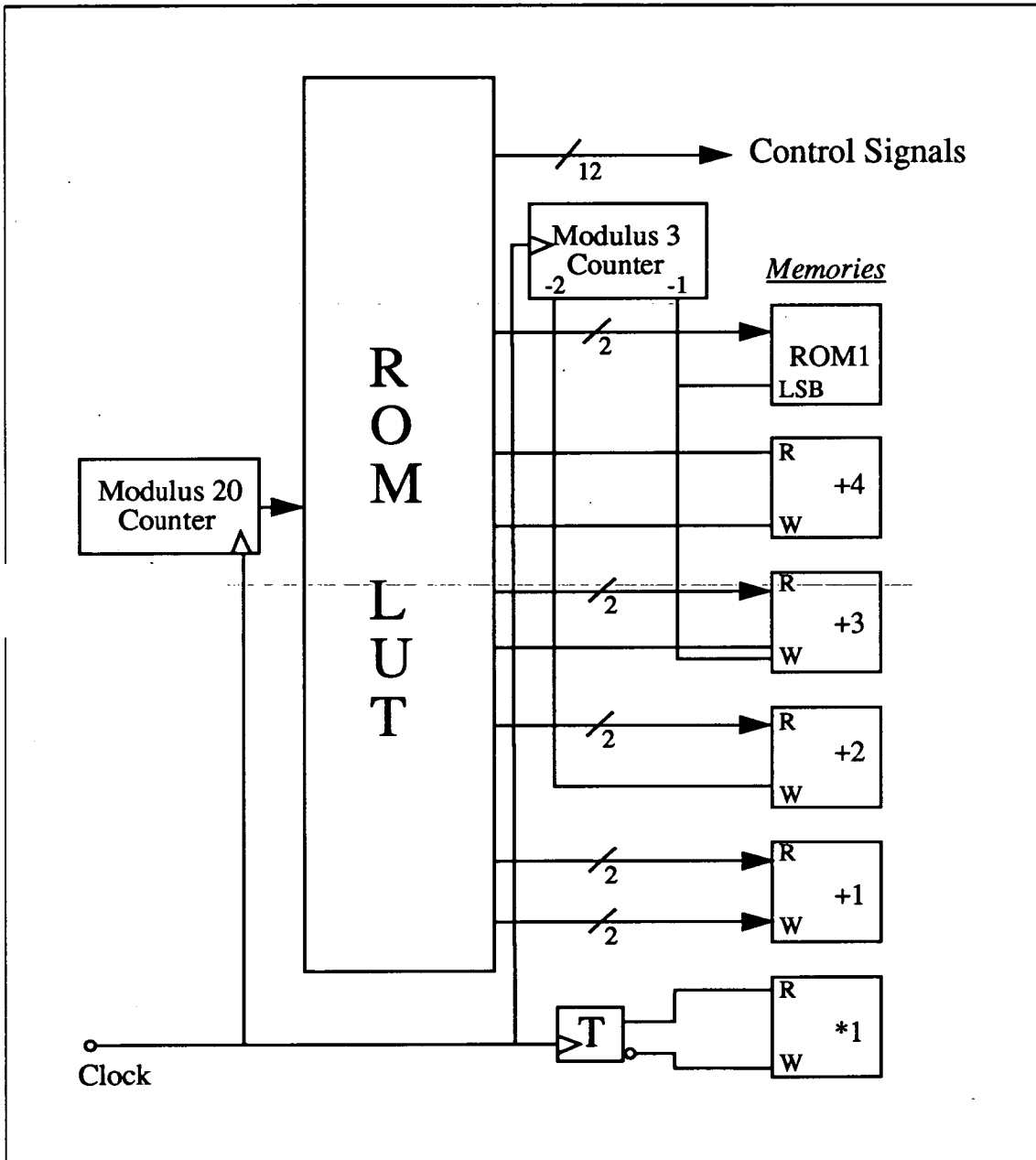


Figure E.1 Address and control generator for wave filter example.



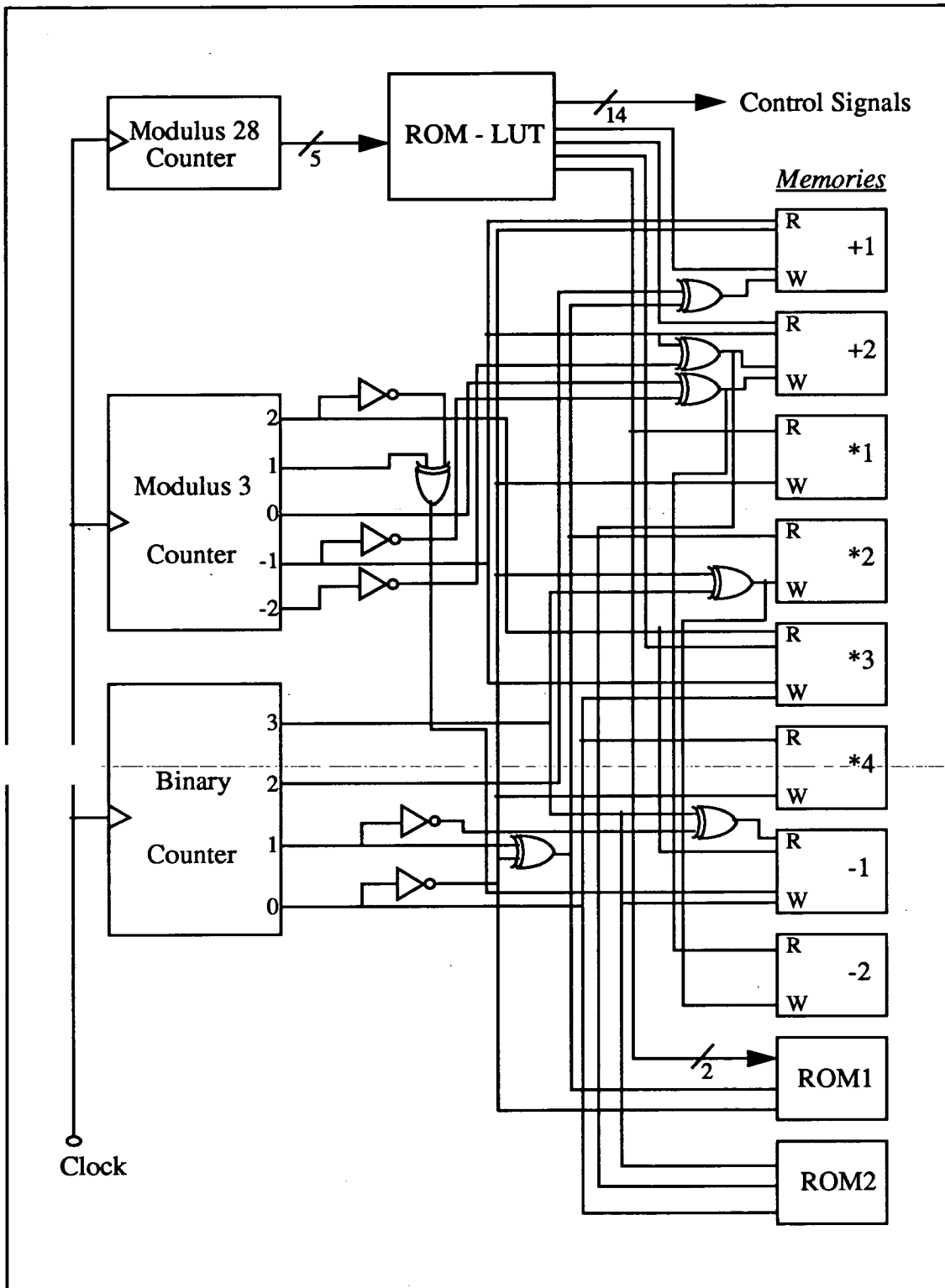


Figure E.2 Generator for the FDCT design.

