(53)

The Design of the Amorphous Computing Project's Prototype Processor

by

Han Chou

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degrees of Bachelor of Science in Electrical Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology

May 20, 1998

Author .....
Department of Electrical Engineering and Computer Science
May 20, 1998

Certified by .....
Christopher J. Terman
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

The Design of the Amorphous Computing Project's Prototype Processor

by

Han Chou

Submitted to the
Department of Electrcial Engineering and Computer Science

May 7, 1998

In Partial Fulfillment of the Requirements for the Degree of Bachelor of Science in Electrical Science and Engineering Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The time required to simulate an ultrascale computing environment in software is the core bottleneck of Amorphous Computing research. This environment is simply a multitude of almost-identical processing units connected in some communications medium. A hardware implementation of the amorphous computing microchip would greatly decrease simulation time, thereby increasing the efficiency of research in the area. This paper specifies such a design of an amorphous computing unit using contemporary VLSI techniques.

In addition to increasing the productivity of Amorphous Computing research, this hardware implementation will allow for observation of more real-world problems implicit to this computing environment. This paper is intended to serve as a reference manual into the specifications for the chip, and focuses on the test structures and glue-logic portions of the chip design.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Background - The Amorphous Computing Project

Recent technological developments will enable inexpensive production of large numbers of tiny information-processing elements with integrated sensors and microactuators. It is hoped that these sensor-rich processing elements can be distributed and embedded into structural building materials to create intelligent and responsive environments, such as bridges with load sensing capabilities or smart surfaces that monitor the weather.

If these elements are small and inexpensive enough to be mixed into materials that are produced in bulk, such as paints, gels, and concrete, then these smart materials may be able to reduce the need for strength and precision in mechanical and electrical apparatus, through the application of computational intelligence.

Imagining for the moment that we have the hardware necessary for building these kinds of structures, we then require new programming paradigms for obtaining organized, fault-tolerant, and coherent behavior in such environments. We have come to call this effort of identifying the engineering principles and languages that can be used to observe, control, organize, and exploit the behavior of programmable multitudes the study of *amorphous computing* [1].

Amorphous computing considers the following questions fundamental in determining how best to organize ultrascale computing systems:

> How do we obtain coherent behavior from the cooperation of large numbers of unreliable parts that are interconnected in unknown, irregular, and time-varying ways?

What are the methods for instructing myriads of programmable entities to cooperate to achieve particular goals?

1.2 Motivation for hardware implementation now

Software and board-level simulation of the amorphous computing environment have allowed researchers to make the first level of discoveries in these new programming methodologies. These methods have matured to such size and complexity that software and board-level environments are now too slow to be suitable for productive research. Generally speaking, many methods for organizing processors [2] into approximate hierarchies, shapes, sizes, and orientations, and variably overlapping clubs have been developed based loosely on LaPlace equations for analytical surfaces.

These methods of spontaneously collecting processors into larger groups have greatly increased the numbers of calculations needed in software simulation. As these larger blocks are now communicating with each other, interactions have become an order of magnitude more complex.

For this reason and others, a microchip implementation of the amorphous computer is now desired. The highly parallel nature of processing in this hardware environment should save orders of magnitude in simulation time. In essence, software environments necessitate that one or a few processors emulate many thousands of amorphous computing units, while this hardware environment will allow for simultaneous processing of information.

We are designing a chip capable of operating at 100MHz, but however fast our current implementation may be, it is important to note that even today's most advanced tech-

nologies would not permit a feasible processor small and inexpensive enough to fabricate in bulk and create the aforementioned smart materials. However, the silicon microfabrication processes which we are using to create the amorphous computing prototype have an excellent chance of being, or being closely related to, the medium on which these ideal processors are created in the future.

Assuming that amorphous computers will in fact be realized in some VLSI process, this first design of the amorphous computer in silicon will allow observation of many more real-world reactions and problems implicit to the amorphous computing environment.

Implementation size in today's processes also allow for large numbers of input/output pins on the chip exterior. In the future, we anticipate this luxury will no longer be available since pins on chip packages have not been scaling with transistor minimum feature sizes.

In the extreme case, we imagine the final amorphous computing unit with only three exterior connections: ones for power, ground, and communication. In this case, our only way of determining the chip's internal state for debugging purposes will be through the chip's radio module. If the radio module is faulty, this complicated matters even more.

It is therefore desirable that we have well-tested hardware cores which we can use in later implementations. At the very least, it is desirable to have a robust interconnection scheme, including test structures, through which the major modules on-chip may communicate and be thoroughly tested, perhaps easily replaced.

1.3 Goals

This paper is intended to be a reference manual for all users of the chip. This includes hardware debuggers, amorphous computing programmers, and designers of future versions of amorphous computing units. Since many of the specifications and design techniques and decisions have changed in mid-design, and still other decisions have not yet been made, this document is expected to be amended to reflect any late changes made in this previously document-sparse project.

The design of this chip is the culmination of the efforts of many. Overall functionally and all major modules are described in this paper, although the design work was not necessarily done by the author. These module design descriptions are included simply so that this document may be used as a reference guide for the chip. The author of this thesis designed the test structures and the glue-logic (module interfacing and layout) for this chip. Hopefully these structures will be reusable should any additional modules be placed in later chips, or should any changes in current module implementations be needed.

The commonalities which we feel will be valid through several iterations of this chip include a protocol and medium for communication between modules (a single-cycle processor-mastered internal bus) a debugging mechanism which will allow us to view the state of the chip from its external pins, and a self-test scheme which provides a fast means of functional verification.

Fast design time was a major goal of the design team for this chip. Thus we have made design trade-offs to keep designs simple. We assume that future iterations of this

chip will perform the necessary minimizations or other optimizations that will increase performance, area, and power metrics.

Hoping that many parts of this prototype design will be re-used in later (potentially optimized) chips, we are motivated to provide extensive and complete test structures so that we will be able to find any potential hardware bugs easily and quickly, especially since we anticipate some amount of hardware bugs in this first design. It is also a goal to provide later designs with well-tested and functional (at least at the logic level) modules. Even if these modules should change, hopefully, their I/Os will remain similar, such that they can be easily plugged into the existing design for test.

1.4 Roadmap to this thesis

This document is divided into four parts. First is a logic description of the normal operation of this chip including broad specifications for the main modules on the chip and their interfaces to the internal bus. Several high level abstractions such as the memory address space allocations and Beta processor description are described here, which may be useful to programmers, or people debugging code compiled for this processor. Signal names are given here, which may serve as a good reference point for people discussing this chip's design and possible enhancements or derivations, or other low-level discussions.

Next, test and debug structures are described. These test structures have a mode in which they do not affect (but perhaps can observe, or capture) the state of the chip. Those investigating the fabricated chips should find the description of the test interfaces here useful. Both single-step (interactive) and built-in self testing (BIST, hardware for fast func-

tionality verification) are described in this section. The hardware pieces are shown, and testing examples and methods are shown.

The third section describes the physical specifications of this design. The general floorplan is specified, the dimensions of each module are listed, and power supply and clock distribution are diagramed. Relevant information on the process technology is included.

The last section is a list of improvements for the next iteration. Possible optimizations which were identified but not implemented are noted, and areas where minimization could be performed if deemed useful are outlined. Potential units to add to the chip are described, some of which implement required functionality in the applications described in the motivation and background section of this thesis.

Chapter 2 Functional Description

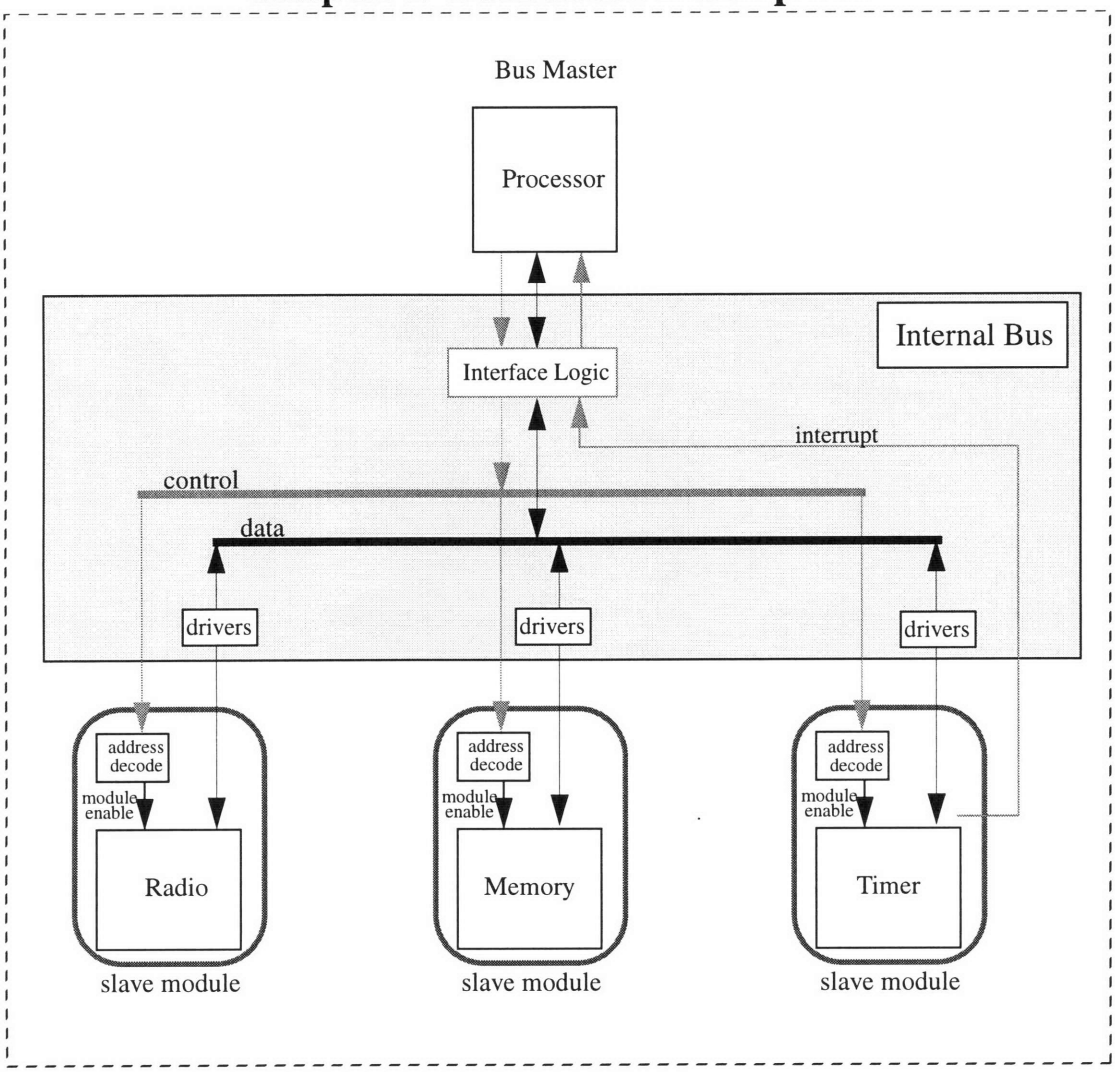

Figure 1. Functional Unit Blocks
This picture does not reflect actual or relative size, shape, or placement of modules.

2.1 Top-Level Description

This section describes operation of the chip in normal mode. In normal mode, all test structures are transparent to the functional units shown above. Normal and other modes of chip operation are determined by the command word in the Instruction Register. This register and all modes of chip operation are described in detail in Chapter 3. Besides test structures, driver controls and external pins are also not discussed in this section.

2.1.1 High Level Description of Operation

This implementation uses the basic units necessary for an amorphous computer. We need a processor to run code and control data transfer on-chip, we need memory to store data, and we need a radio unit to transmit and receive data with other amorphous computers. We also have an interrupt timer, through which we can provide the chip with some sense of time -- counting either global bus clock cycles or on-chip oscillator cycles (for real-time calculation if the global clock is irregular). There is a single data bus which will allow communication between these modules, and this structure will be called the internal bus. As we shall see later, since the internal bus interfaces to module input/outputs, it turns out to be the right place to place test structures.

2.1.2 Processor-Mastered Internal Bus

We call any agents which can drive the control (address and write-enable) lines on the internal bus the master of the bus. In our implementation, the processor is the sole master of the bus, and so we refer to operations from the point of view of the processor. We refer to all other modules connected to the internal bus as slave modules.

Slave module enable signals are encoded in the high order bits of the address lines, and specific functions (for the radio and timer module) or memory locations are encoded in the remaining (offset) address bits. Since the processor will be performing a bus transaction on every clock cycle, all slave modules are required to complete and be valid in exactly one bus cycle.

Most of these transactions will be the processor attempting to fetch its next instruction from memory. When we were previously using a Sparc DSP as our processor, there

was some interface logic needed to convert the processor's Harvard (dual bussing for simultaneous instruction and data memory access) to our architecture. The present interface logic, however, consists simply of logic to decode the memory write byte enable signals to determine the direction (read or write) for modules' bi-directional i/o drivers.

2.1.3 Internal Bus Signals

The following is a list of named signals through which the chip's functional unit blocks communicate:

TABLE 1. Internal Bus Signals

Signal Name	Description
Brst	Global Reset
Bclk	Global Clock
Birq	Interrupt Request (driven to the processor from the timer circuity)
Baddr<15:0>	Address Lines driven by the processor
Bmwe<3:0>	Memory Write Enable -- each correspond to one byte of the 32-bit data word
Bdata<31:0>	32-bit data word, driven by the processor on writes, by slave modules on reads

2.1.4 Memory Address Space Allocation

Although the processor has 32 address lines, we could only fit 40 kilobytes (KB) of memory onto this chip, so we only use 16 address lines corresponding to a total (byte-addressed) memory space of 64 KB. The processor drives all 32-bits of data when any single or combination of the 4 Memory Write Enables are active, although anywhere from 1 to 4 of the bytes are valid and intended to be written in the cycle. Slave modules infer the Read operation when all four write enables are inactive.

TABLE 2. Memory Address Space Allocation

Memory Space Range	Module	Function
0x0000 to 0x4FFF	ROM	20KB read-only memory -- see Memory Section
0x5000 to 0x7FFF	Unspecified	Unspecified
0x8000 to 0xCFFF	RAM	20KB read/write memory -- see Memory Section
0xD000 to 0xD01F	Radio	See Radio Section
0xD020 to 0xD03F	Timer	See Timer Section
0xD040 to 0xFFFF	Unspecified	Unspecified

2.2 Beta2 Processor

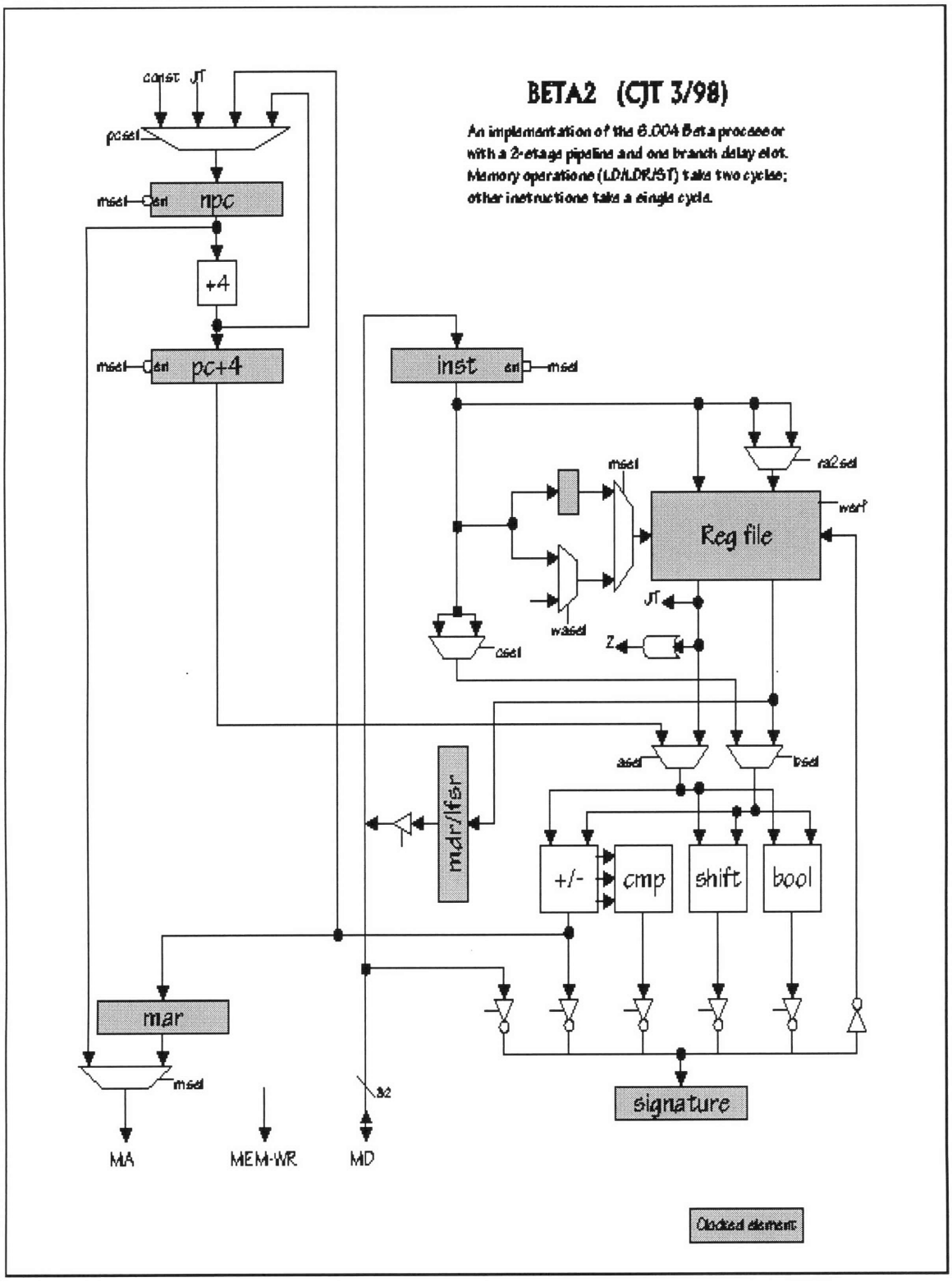

Figure 2. BETA2 Datapath

Chris Terman's design of the BETA2. Note that all clocked elements but the Register File are part of the scan chain.

2.2.1 Datapath

This processor was designed by Chris Terman (cjt@mit.edu). It is a 32-bit processor with a 2-stage pipeline with one branch delay slot. Memory accesses take an additional cycle and thus stall the pipeline. This processor has 32 general purpose registers, and no caching is implemented (as we do not have enough space for this). The simple architecture has a RISC instruction set [3] which is well-tested, and easy to learn and debug.

2.2.2 Major Control Signals and I/O

TABLE 3. Beta I/O and Major Internal Control Signals

Signal Name	type	Description
Prst	input	Processor Reset (connected to global reset BRST)
Pclk	input	Processor CLK (directly connected to global BCLK)
Pirq	input	Processor Interrupt Request (generated only by the timer circuit in this chip)
Pmode<1:0>	input	00 or 01 means normal operation, 10 means scan data, 11 means run built-in self test.
Ptdi	input	Input to (tail of) Processor's serial scan chain
Ptdo	output	Output from (head of) Processor's serial scan chain
Pmwe<3:0>	output	Processor Memory Write Byte Enables. Passed to boundary scan registers before becoming BMWE<3:0>
Paddr<31:0>	output	Least significant 16 bits are passed through tristate-able boundary scan registers to the internal bus.
Pdata<31:0>	input/output	All 32 bits are passed through tristate-able bi-directional boundary scan registers to the internal bus.
Pmsel	control	Interior control signal for processor. Active when making a data memory access. Stalls the pipeline since memory operations take two-cycles.
Pwerf	control	Interior Register File Write Enable
Pasel	control	ALU input 1 (A) select
Pbsel	control	ALU input 2 (B) select
Pcsel	control	Literal or offset (C) select
Ppcsel	control	Program Counter Select
Pwasel	control	Write Address Select
Pz	condition	Register Value = Zero?
Pjt	pointer	Jump address
Pconst	pointer	Exception pointer

2.2.3 Processor Modes

The processor operates in three modes. In normal mode, a rising edge clock (PCLK) either advances everything in the pipeline (including doing the fetch for the next instruction), or performs a data memory read/write. In Test Mode, a rising edge clock shifts read-only data out of the processor's internal registers. In BIST Mode, the LFSR provides memory access data (both instruction and data).

2.3 RF Transceiver (Radio)

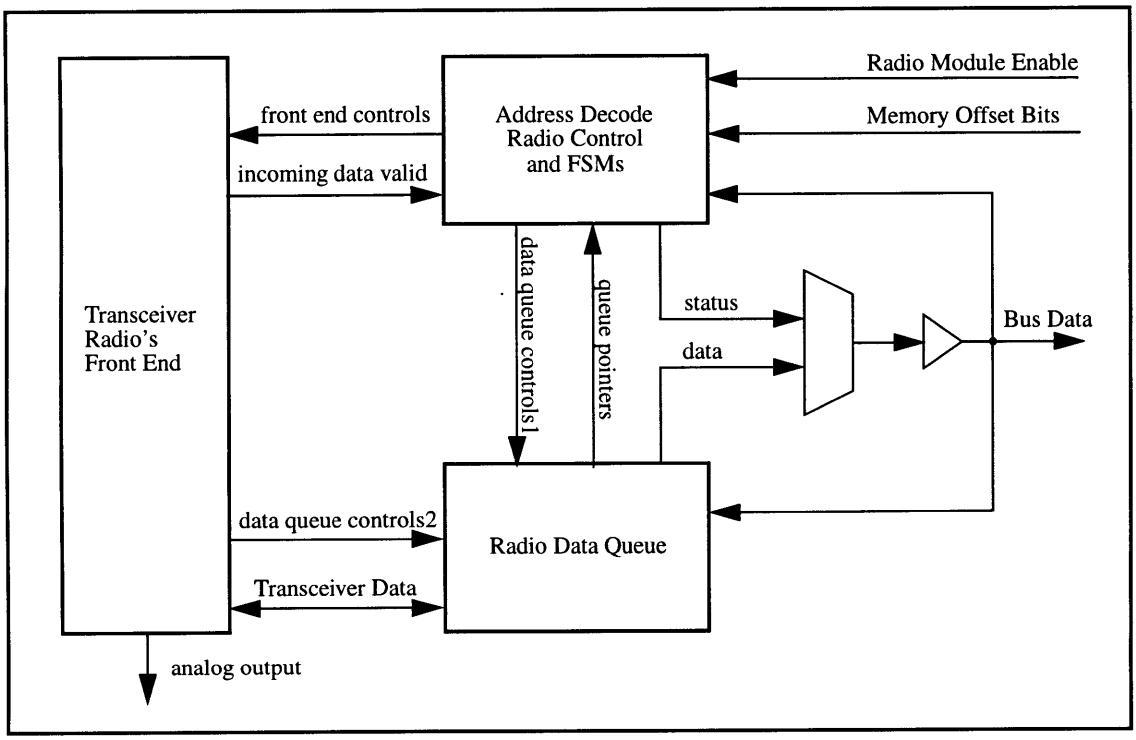

Figure 3. Radio Module Block Diagram

Data queue controls1 controls when data is read out of the incoming buffer or written into the outgoing buffer.
Data queue controls2 controls when data is read out of the outgoing buffer or written into the incoming buffer.
'Front end controls' manage shift channel codes, exchange registers, synchronize the receiver, and provide threshold parameters for the front end.

The Radio module is currently being designed by Chris Hanson. Much of the digital portions of the radio have already been fabricated and tested, with verified functionality. However, we will be making some changes to this circuitry for the amorphous

prototype. The prototype will have the first silicon implementation of the analog front-end circuitry.

2.3.1 Radio Operation - Transmitting and Receiving

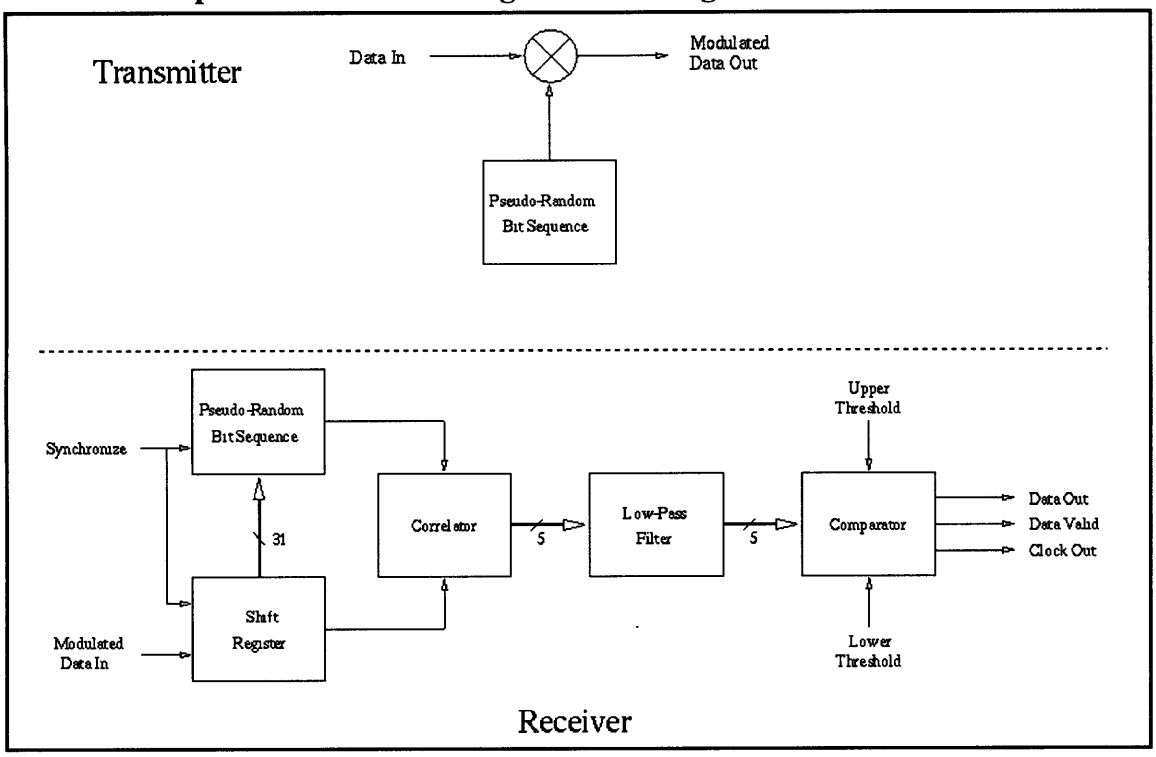

Figure 4. Block diagram of the radio (front-end) transceiver.

This block diagram shows the separate transmitter and receiver circuits on the chip. An important logic block in this circuit (and also in test structures) is the pseudo-random bit stream (PRBS) generator. This is really a shift register, where the lowest order bit is generated by properly XORing together certain bits in the 31- bit register [4]. The other bits are shifted to one higher place of significance. The XORs which compose the new (lowest order) bit to be shifted in are created in such a way such that the resulting sequence of 31-bit values cycles through the 2^{31} - 1 values encoded in the register. The zero value in this register is the single value that is not cycled through. The PRBS is unable to leave the zero state unless it is reset.

We will call the PRBS a Linear Feedback Shift Register (LFSR) in the context of testing structures. We call it a PRBS in the context of the radio because usually we want to think of this structure as generating a pseudo-random sequence of *bits* whereas we want to think of a pseudo-random cycling through of many 31-bit values when we use it for testing.

The transmitter simply mixes the input data with a high-speed pseudo-random bit stream (PRBS) to produce a modulated baseband output. That is, each data bit input controls whether the next several bits (in our case, 512 bits) generated in the PRBS are sent through the (analog) transmitter in their true or complemented version.

The receiver accepts a modulated baseband signal as input. It expects this signal to be like the transmitter output -- a single bit mixed with a sequence of 512 bits generated in the PRBS. The receiver collects the modulated baseband signal bit-stream into a 31-bit shift register. This value is fed to a correlator, which counts the number of bit-wise matches the current 31-bit sequence has with in the locally-generated copy of the PRBS.

Thus, a 5-bit value (there can be anywhere from 0 to 31 bits which matched the current PRBS) will be generated each clock cycle, each being either two more, two less, or exactly the same as the previous 5-bit value. Now we can imagine having a 512 long sequence of 5-bit values for each bit that we want to transmit. The low-pass filter block attempts to remove high frequency components in the 512 sample-long sequences.

The low-pass filtering actually implements an averaging of the many 5-bit samples into a single 5-bit value. This averaging occurs on clock cycles of powers of two, such that we finally have one 5-bit value every 5.12 μs (the data rate). If this averaged 5-bit value is

greater than an upper threshold or less than a lower threshold which we can program, then we decide that the data is valid (Data_In_Valid is asserted). The data bit itself (Data_In) is determined by whether it is above or below the threshold. The upper and lower thresholds are composed of 4 bits each. This is because we will require that the number of matches (or differences) with the PRBS be greater or less than one half (16) the total possible to consider the data to be valid.

The "synchronize" command instructs the receiver to assume that the modulated input data is valid and to synchronize (load) its internal PRBS with that data. In practice, this will not always work, so there will be a state machine responsible for iterating this process until synchronization is achieved.

The transmitter/receiver pair works with a 5.12 μsec/bit data stream (about 191 kbit/sec), and a 2^{31}-1 bit PRBS running at 100 Mbit/sec.The PRBS rate and the data rate may be varied but the data rate must be exactly 512 times slower than the PRBS rate. This PRBS has been fabricated and tested. It runs at PRBS rates up to 125 Mbit/sec. The PRBS will be clocked by the global clock pin on the amorphous prototype.

The rate ratio of 512 provides a theoretical "process gain" of about 27 dB, which will in practice be about 15-20 dB. The process gain provides a measure of immunity to other non-correlated signals in the same frequency range -- specifically, such signals must be 15-20 dB stronger than the signal of interest before they will cause significant interference. Due to the process gain, we are able to have multiple transmissions simultaneously sharing the same physical channel.

2.3.2 Processor-Directed Functions in the Radio

Below is a table of memory offset values which correspond to memory-mapped commands the processor may issue to the Radio Module.

TABLE 4. Radio Address Space Offset Decode

Mem offset	read/ write	Function/ Description	Data MSB Bdata<31:24>	Bdata<23:16>	Bdata<15:8>	Data LSB Bdata<7:0>
0x00	write	Reset				A
0x04	write	Synchronize				K
0x08	write	Enable Test				T
0x0c	write	Register Xfer				SSDD
0x10	write	Set Threshold				UUUU LLLL
0x14	write	Code Shift			RR+	NNNN NNNN
0x1c	write	Write Buffer	IIII IIII	IIII IIII	IIII IIII	IIII IIII
0x00	read	Radio Status		FFFF FVBT	PPPP PPPP	UUUU LLLL
0x10	read	Read Buffer	OOOO OOOO	OOOO OOOO	OOOO OOOO	OOOO OOOO

Note that the Memory offset values represent only the 5 least significant bits of the bus address (BADDR<4:0>). A write command to the base address in the radio module's enabled region asserts the radio **reset** signal, thereby initializing the state of the radio. This reset signal may also require the least significant data bit active high (logic 1). The effect of a reset includes presetting all PRBS registers high, clearing the radio buffer, putting the radio in operational mode (not test mode), and putting all radio FSMs in their initial states. The threshold values may also be reset to some value, as yet to be determined.

TABLE 5. Address Space Data Symbol Elaboration

code	Representation	code	Representation	code	Representation
A	Reset Data	D	Destination Register	+	Up/Down
K	Synchronize	U	Upper Threshold	N	Amount to shift
T	Test Enable/Disable Data	L	Lower Threshold	I	Data into buffer
S	Source Register	R	Register to shift	O	Data out of buffer
F	FSM state bits read out during status checks - this is how to tell when functions are done	P	Pointer values for incoming head/tail and outgoing head/tail	B	Data In
				V	Data In Valid

The **synchronize** command attempts to synchronize the receiver's PRBS to some transmitter's channel code by observing the incoming data stream, and assuming that transmission occurred smoothly, as described in section 2.3.1. An FSM is used to determine whether or when this channel code capturing has succeeded. This is done by examining the Data_In and Data_In_Valid signals.

The **radio status** command allows the processor to read various status registers in the radio module. This data includes the current values for the upper and lower thresholds, the data valid and data valid in signals, whether the radio is in test mode, the states of the channel code shifting and synchronization FSMs. These bits are yet to be assigned, and they may extend into higher order data bits on the bus than we have shown here. We will be most interested in when these FSMs have completed their tasks. The command also allows us to read the current positions of the pointers into the data buffer.

As will be explained in the section below, we have determined that four entries in each direction (incoming and outgoing data buffers) should ensure that data is not inappropriately overwritten. Each pointer (head and tail of each buffer) is therefore two-bits long. Higher order data bits correspond to high order bits for the value of the pointers. The pointers are (in order from most significant to least) incoming head, incoming tail, outgoing head, and outgoing tail.

Our present threshold values are 4 bits long each. They loosely correspond to how "off" we are willing to be and still decide that the incoming data is valid. We say loosely, since some low-pass filtering is done to smooth the correlating function. These threshold

values can be read out during the status command, and are written in the **set thresholds** command.

The **register transfer** command is currently planned to be implemented as a copy from one register to another (overwriting the destination value). Eventually, we may want it to be a parallel exchange (swapping) of the values of two (31-bit) PRBS registers.

We will have 4 pseudo-random bit stream generators, each corresponding to a specific modulation channel. One PRBS keeps track of the receiver channel code, another the transmitter channel code. The other two PRBSs keep track of channel codes which we are presently not using but may want to switch into either the transmitter or the receiver. Later radio units may have multiple transmitter or receiver units, so that multiple channels can be decoded or encoded simultaneously.

Besides swapping existing channel codes, we may want to **shift channel codes** forward or backward in time. The command takes data from the bus in a sign-magnitude representation, with the most significant bit determining the direction we want to shift the channel code. Advancing the code in time corresponds to allowing more clocks to the register than normal, and moving the code backwards just disables clocking to a particular register for the number of cycles.

Valid data received in the receiver (incoming data) and data intended to be sent out by the processor (outgoing data) are stored in the radio's data queue. The processor will perform **read incoming buffer** and **write outgoing buffer** commands to perform these queue accesses. The pointers will be moved to reflect where the new pointers of the head

and tail of the queue are. These processor accesses (and thus all entries in the data queue) are full 32-bit accesses.

To **enable radio testing**. we write to the proper address and make the least significant data bit BDATA<0> active high (logical 1). This effectively connects the transmitter output to the receiver input, allowing us to test the analog circuitry by sending data through the outgoing buffer and (hopefully) back into our own incoming buffer. To **disable radio testing** (and put the radio into normal operating mode), we write an inactive low value (logical 0) to the lowest data bit of the specified address.

We note that test structures require us to provide a read for every write command in a module's given address space. Therefore, the radio will perform a radio status read for the address offset range 0x00 - 0x0F, and it will perform a radio buffer read for any address in the offset range 0x10 - 0x1F.

2.3.3 Radio Data Queue

The radio unit defaults to actively trying to decode incoming data. It operates at the bus clock frequency. Transparently to the processor, a FIFO buffer in the radio will be filled as incoming data is successfully received. Because the radio unit is a slave module, it cannot take control of the bus and request data transactions. Therefore, we must prevent accidental overwriting of data in this buffer by ensuring that the processor moves data from the incoming buffer on a regular basis. How frequently this must happen depends on the size of the data buffer, and the maximum speed at which it can become filled.

The data rate ratio is 512 global clock cycles per bit, and we store 32-bit words in the each entry of the data buffer. Therefore we expect (a maximum of) one word to enter the buffer every 16,384 cycles. We have then decided to make the buffer four entries deep such that we have to check the buffer at a minimum period of once every 65,536 cycles. At the estimated operating frequency of 100MHz, this corresponds to checking the radio buffer approximately every 650 μsec.

Since the processor we are using only has one interrupt request line, we have chosen not to allow the radio to directly cause an interrupt to the processor. Although this could be implemented in a simple design, there are other several other routines which we probably want to run in timed intervals as well. Therefore, we have chosen to implement an on-chip timer, which will interrupt the processor relative to either a programmable, real-time period as generated by an on-chip oscillator or a programmable number of global BCLK signals.The specific routine of checking the radio's data buffers we will refer to as radio buffer polling. Radio buffer polling may also include similar measures to ensure that the processor does not write values into the outgoing radio buffer as well.

A datapath for the radio's data buffer is shown below. Several optional signals are calculated such as INfull which may be useful in generating radio-initiated interrupts to the processor in future designs.

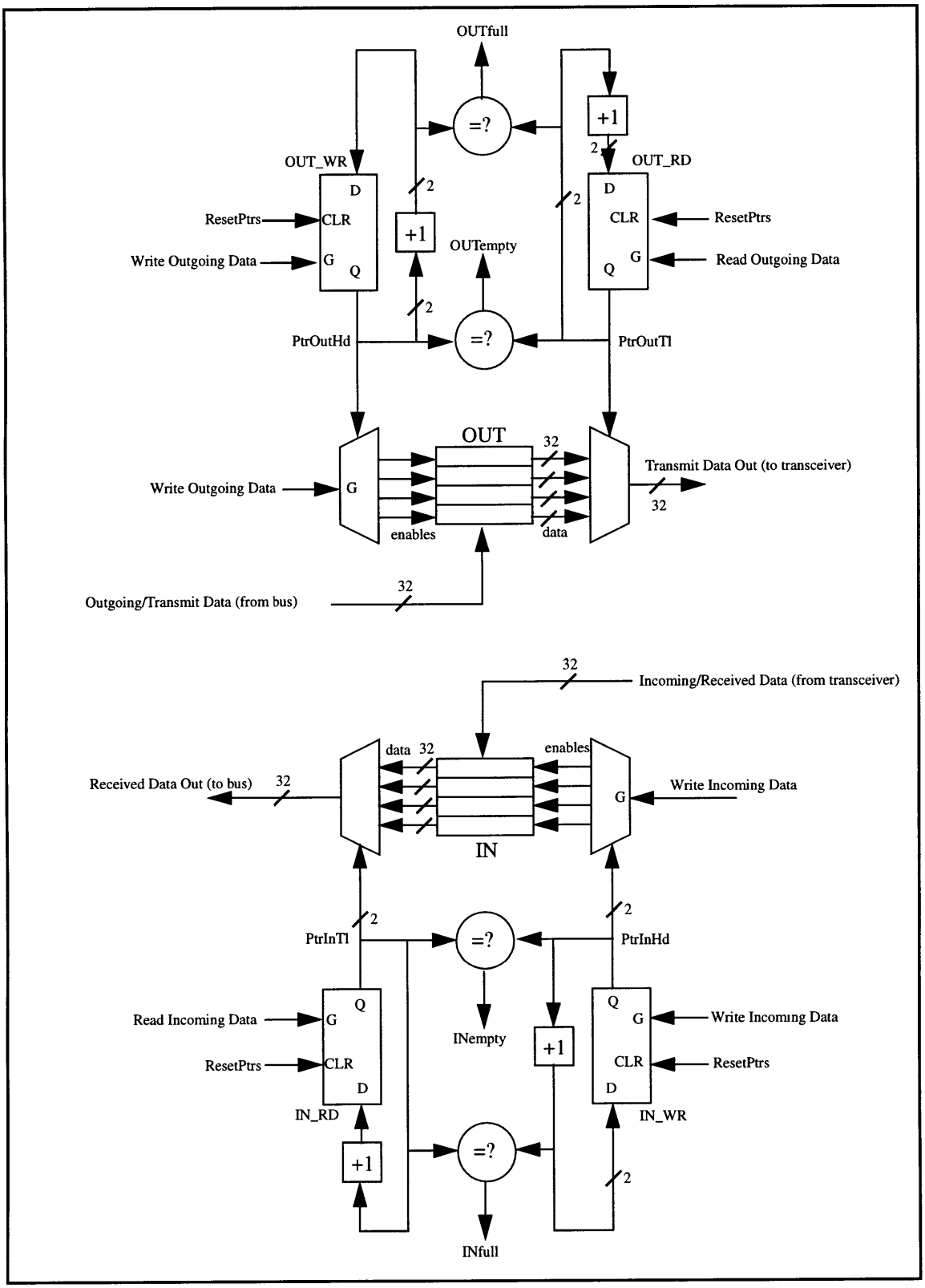

Figure 5. Radio Data Queue

2.3.4 Interrupt Timer

This module is a simple programmable counter. We will clock this timer with the global bus clock, as this will be operating at a consistent real-time period. The timeout signal from the timer goes through internal bus structures and eventually acts as the IRQ input to the processor.

As with the radio, we need to ensure active reads are possible from every address available in our allocated memory space. Therefore, the timer module reads the current value for memory offsets 0x00-0x0F, and reads the period for 0x10-0x1F.

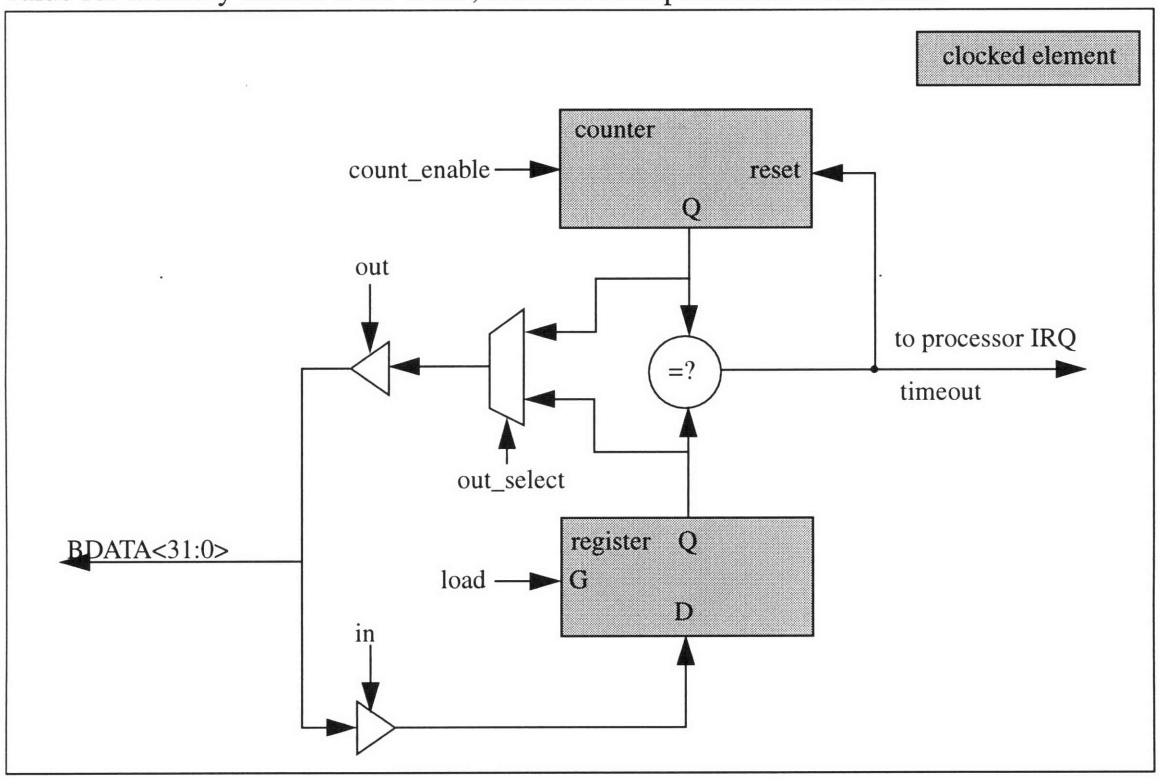

Figure 6. Interrupt Timer Block Diagram

TABLE 6. Interrupt Timer Functions

mem offset	type	function/description	data
0x00	read	readout current count value	Bdata<31:0>
0x04	write	enable count - sets or resets register which enables or disables the counting in the counter.	Bdata<0> high enables, lo disables
0x08	write	write new timeout period	Bdata<31:0>
0x0c	write	reset	Bdata<0> high resets
0x10	read	readout current timeout period	Bdata<31:0>

Note that the Memory Offset values represent only the 5 least significant bits of the bus address (BADDR<4:0>)

2.3.5 RAM and ROM (Memory)

The memory cells were designed by Tim McNearny and redesigned by Chris Hanson. These cells were designed to be operate at 100 MHz. Their layouts were made as small as possible, then used to fill the chip with memory cells after the other modules were completed (or at least, their boundary boxes restricted to an agreed size).

Half (20 kilobytes) of the total memory (40 kilobytes) is Read-Only Memory, and the other half is Read-Write memory. The density of the ROM cells is about 4 times that of the RAM cells, and so there is 4 times as much space used by RAM cells as by ROM cells. ROM cells come in 4 kilobyte blocks (1024 32-bit words) each and RAM cells come in 1 kilobyte blocks (256 32-bit words) each.

Both the RAM and ROM have 14 address pins. That means that we have 2 pins which we can "program" in the ROM, and 4 such free pins in the RAM. We group our RAM cells into groups of four to match the structure of the ROM cells, and to simplify the process of interfacing to our 16-bit address bus. We need to program 2 of the RAM pins in

order to accomplish this grouping, and we are left with two free pins to enable the correct memory grouping. This decoding is shown below, along with a rough routing picture.

These memory cells were designed to allow the routing of address and data lines vertically over them (they are wider than they are tall). In order to maximize the number of memory cells we could fit onto the chip, and still make use of the vertical routing over the cells, we eliminated I/O pins on the sides of the chip. We created vertical stacks of five memory cells, with a ROM being on top, and four RAM cells stacked below. Every vertical stack has the same base decode as shown in the selector below.

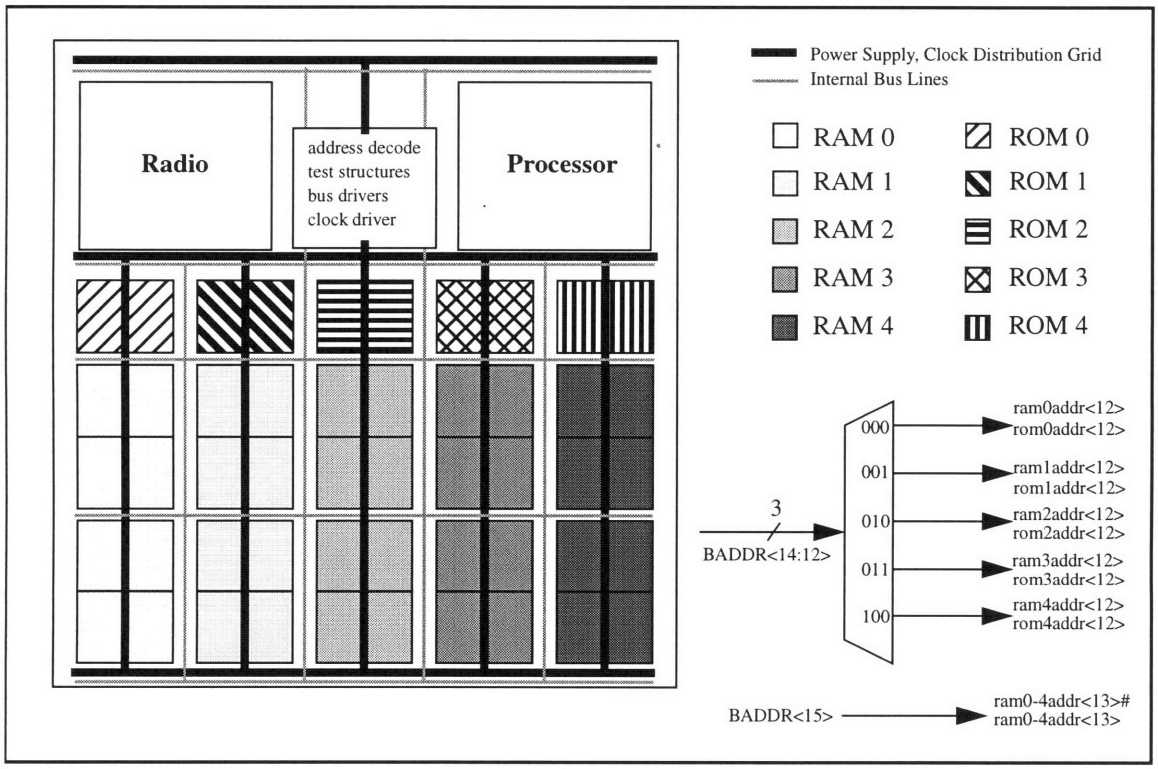

Figure 7. Memory Cell Grouping, Decode, and Signal Routing
Note - Figure not to scale (See Chapter 4)

Chapter 3 Chip Testing

The on-chip hardware structures which we have provided for observing and setting the state of our machine is based upon suggested design implementations of the IEEE Boundary Scan Architecture (standard # 1149.1) [5]. This approach focuses on transparency of test-hardware to normal operation, a small number of I/O connections to control the test-hardware, and ease of design. We have used the IEEE user interface so that existing software can be used to interact with our test hardware.

3.1 Logic Testing - Chip Observability

One of the main goals of our test structures is to provide near-complete observability of the internal state of our chip. That is, we would like to sample the state of the machine at discrete points in time, then somehow get this information to the exterior of the chip. We implement this exporting of data through a serial chain, in anticipation of insufficient pins to do this any other way in the future. Although not particularly applicable to our chip, this will become increasingly important with future designs. State observability will be crucial in isolating and debugging low-level hardware bugs or fabrication problems.

Since the majority of the state of the chip is already found in the chip's registers, we need only provide some way of getting this data to the chip exterior. Using some routing and a large selector, we create a serial sequence to the exterior as shown below. Note that this method allows transparent observation of this data, potentially during normal operation.

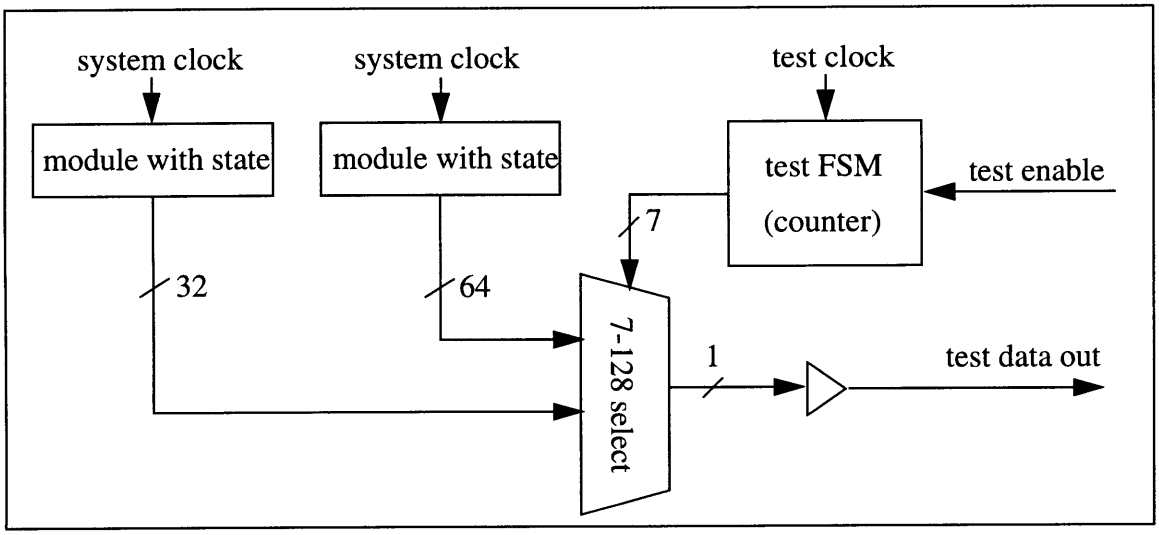

Figure 8. Observability Implementation
In this example the state bits of two modules are routed to a selector, then converted to a serial sequence of bits at the chip exterior. A low (inactive) test enable signal synchronously resets the count -- high (active) allows counting.

The radio module has some internal structures still under construction, and may have a similar observability structures implemented for its internal state. We may refer to the above read-only registers as **module-internal observable registers**.

3.2 Logic Testing -- Chip controllability

In addition to transparent observation of the state of our logic, we would like to have some control -- that is, the ability to put our machine into a known state. To implement either mode (observe or control), we place **boundary scan cells** (BSCs) at module interfaces.

Our basic BSCs consist of two registers and two multiplexors each as shown on the following diagram.

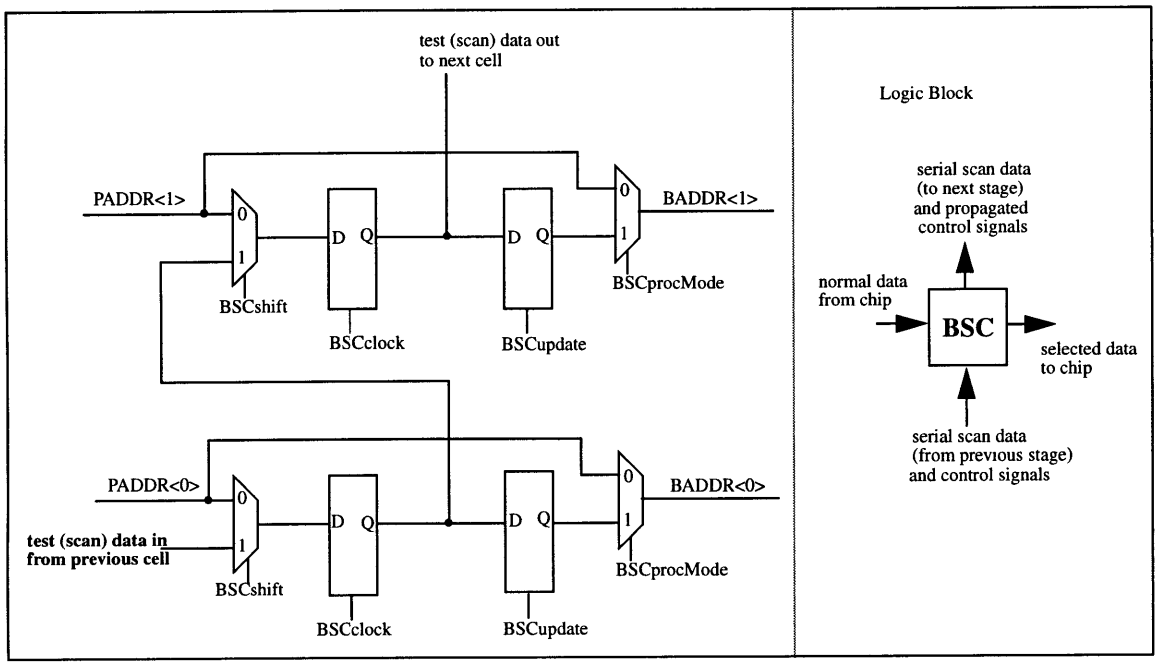

Figure 9. Basic Boundary Scan Cell

This is an example of a BSC implemented at a module's (in the case, the processor's) output. Note that if we can emulate the processor's outputs, we are in effect emulating the inputs to the slave modules. For module input BSCs, we reverse the above picture -- The Interrupt Request Line from the Timer Circuit and the Bus Reset Line (BIRQ, BRST) would be placed on the left in the above picture and PIRQ, PRST would be on the right.

Notice the test data input (in bold) which can be shifted into the register chain. BSCs can be placed in a mode where they provide inputs to the module in place of the nominal chip circuitry. We call this the ability to *emulate the chip* (from the perspective of the module), and the mode in which we do this is referred to as a module's internal test.

We may also *emulate the module* (from the perspective of the rest of the chip) by providing the module's outputs (to the rest of the chip) from the BSCs. This is known as a module's external test.

Because there is only one master of the bus (the processor), there is no need to place BSCs around both inputs and outputs of every module. The output of the processor is really the inputs to all the slave modules, and the outputs of the slave modules are really

the inputs to the processor (except for the global clock signal, and the like). Therefore, we have implemented our test strategy with BSCs completely around the processor, and a few others in special cases. Below is a picture of BSCs completely surrounding a module, allowing emulation of the module (for the rest of the chip) or emulation of the rest of the chip (for the module).

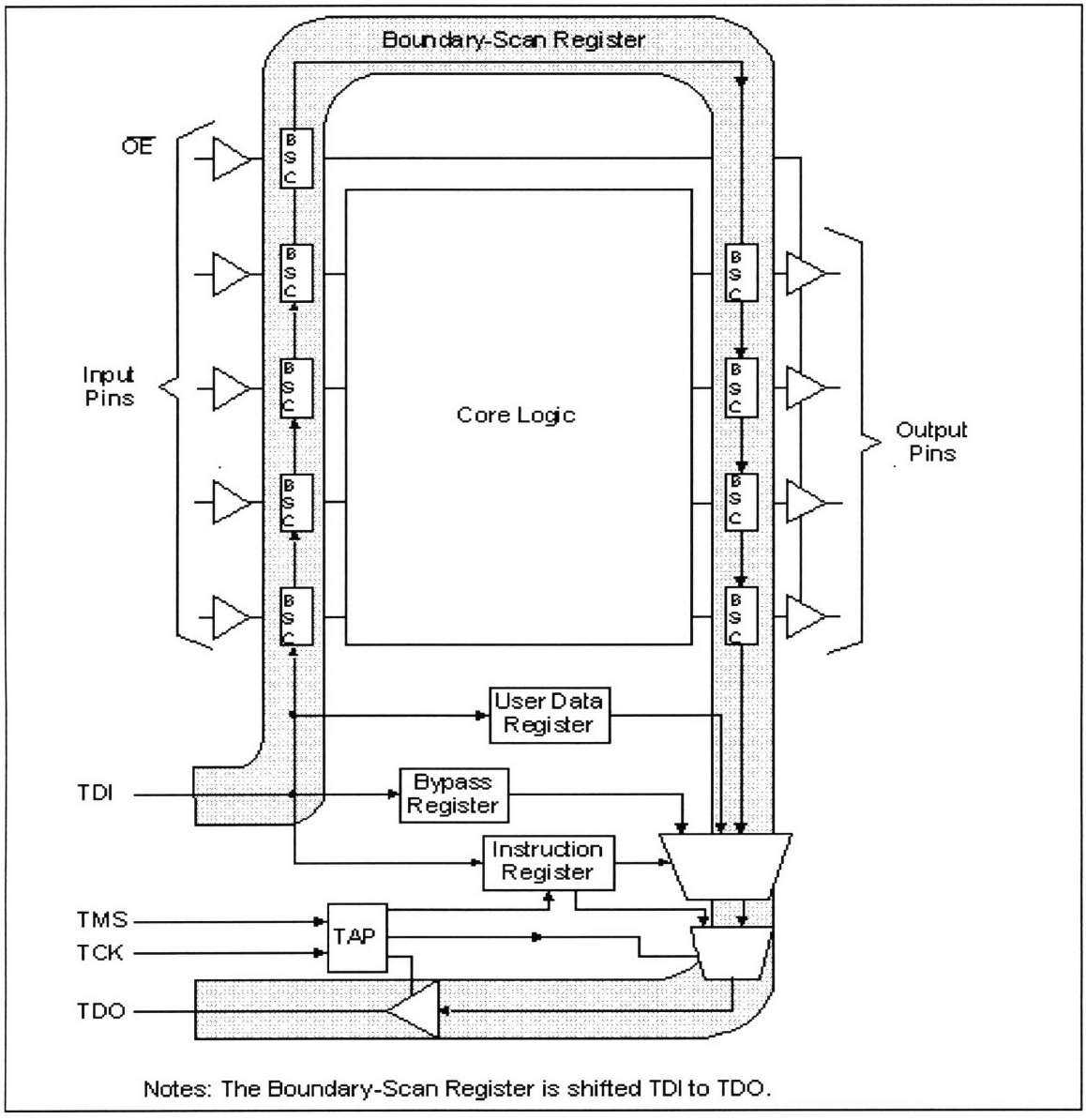

Figure 10. Boundary Scan Method
Surrounding a module with BSCs allows us to fully simulate inputs to the module under test, as well as simulate module outputs (inputs for some other module on-chip). The TAP and the instruction register are the two main structures used to control the BSCs. These are described in the next section.

Of slightly more complication is the bi-directional I/O BSC, shown below.

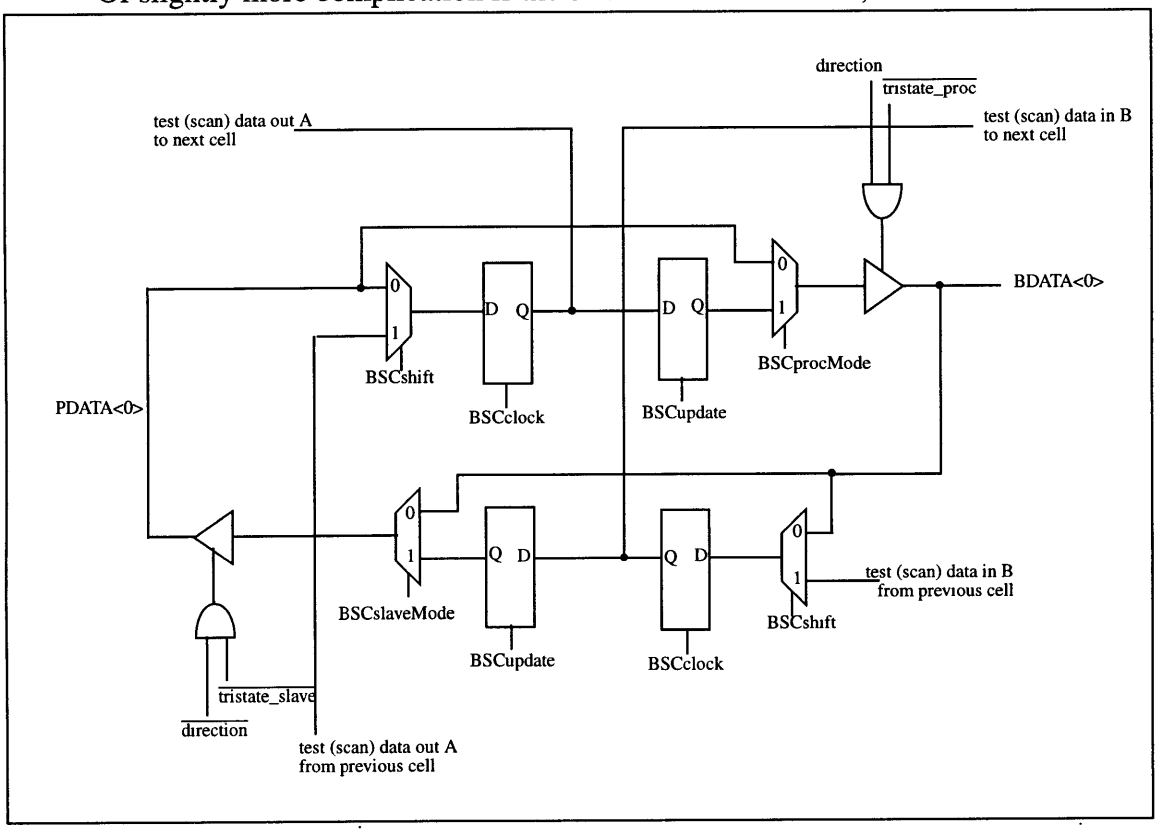

Figure 11. BSC for Bi-Directional Module I/O

One cell of a bi-directional boundary scan chain. Note that there are two paths for scan data. The desire to separate these two paths will become apparent in the following section when LFSRs and Signature Analyzing abilities are added to these cells. The need to separately tristate data also becomes apparent in BIST discussion.

The time required to shift in data and progress through functional clock cycles is too slow to be practical for a quick chip functionality test. Instead, we provide for more specialized hardware so that modules can perform at-speed tests, which will be called Built-In Self Tests (BIST). Self-test for slave modules uses hardware that is added to particular BSCs. Self-testing the processor uses hardware that is internal to the processor.

3.3 Built-in Self Test(BIST) - At-speed testing

The general approach taken towards at-speed testing is verifying the input/output behavior of modules. We provide a stream of inputs to a module at operating speed, and either observe the outputs at the chip exterior at-speed (verify them as they occur), or

observe a checksum (an accumulation the consecutive outputs) after incremental numbers of cycles.

Verification of the outputs at the externals of the chip relies on finding software and hardware interfaces which can operate at the estimated 100MHz operating speed of the chip. This is, in general, difficult. It may be possible to drive the pin externals at a rate of 100 MHz, but even if we do not connect our chip to a printed circuit board, the load of some oscilloscope probes may still be too much for the chip. In addition, having to precalculate and store what the values should be for each cycle is laborious. Verifying the matching of this value probably has such latency that makes this method highly difficult, and therefore we do not discuss the external verification method further in this document.

We have decided to design on-chip hardware to collect test data. This should then allow us to use known, tested software to retrieve the results of our tests stored as checksums in the BSCs.

The checksum technique requires that we begin BIST from a known machine state. We should then be able to calculate how this checksum should end up after a large number (on the order of millions) of cycles, and compare our calculated version with one we shifted out of the chip after applying the correct number of global clock cycles.

Although the resulting checksum is not completely assured of finding errors, it gives us a very probabilistic assurance that we will catch any errors that exist. This is qualified by our correctly placing the signature analyzer in an appropriate place (to maximize capturing the effects of our inputs) and our correct method of collecting this checksum.

These issues are discussed in the following sections which also discuss how the BSCs are manipulated to provide us with LFSR and Signature Analyzer functionality.

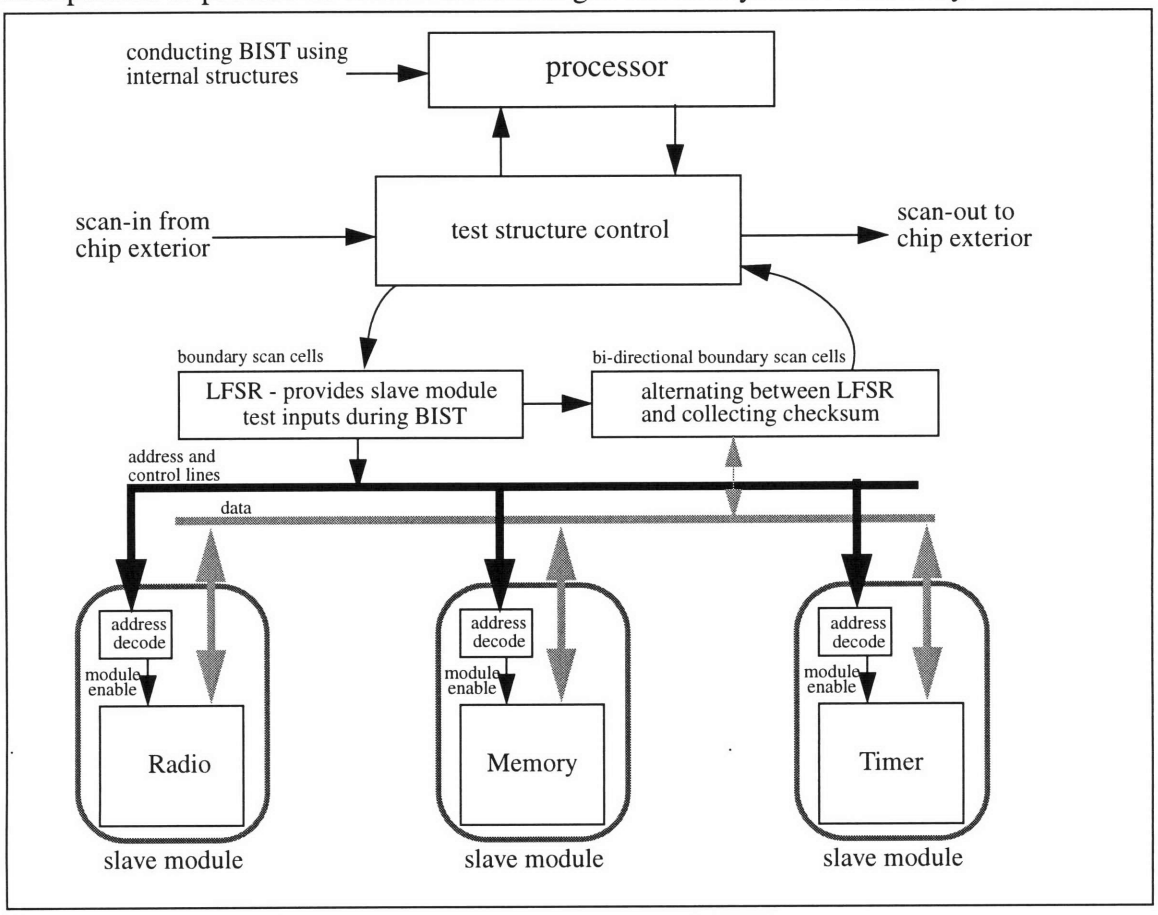

Figure 12. BIST Method

Notice we have separated testing of the processor and testing of the slave modules. Both testings require a pseudo-random number generators (LFSRs) which provide inputs to the module(s) under test, and a signature analyzer which collects the outputs of the module(s) under test and allows this data to be serially shifted to the chip exterior.

3.3.1 BIST for slave modules

The inputs which we must provide to the slave modules during BIST are the address and write-enable lines. We implement the input stream using BSCs modified to produce LFSR outputs, so we will cycle pseudo-randomly through the available addresses. Since our address space is not completely filled (about 40Kbytes out of 64Kbytes are used), some addresses will correspond to no active slave modules and therefore, nothing will drive the data bus. These data lines are the outputs of the slave modules, and thus

where we implement our signature analyzer -- the BSCs at the data I/O lines. We do not want to capture these unknown (undriven) values in our Signature Analyzer, so we disable the Signature Analyzer accumulation during reads to addresses where no module enables are active.

We allow two kinds of BIST for slave modules. In the first, we do all reads, and in the second we perform a write then a read to each pseudo-randomly calculated address. During both read and write cycles, we supply pseudo-random address line inputs. In addition, during write cycles, we supply pseudo-random data to the bus data lines, disable the accumulation in the signature analyzer, and disable the address LFSR from going to the next address. During read cycles, we accumulate the bus data in the signature analyzer (if we have enabled some slave module), and enable the LFSR to bring us to the next pseudo-random address.

3.3.2 BIST for the processor

Processor self-test can be run simultaneously with slave module BIST. Starting with the processor in a known state, we enable BIST and perform several global clocks to the chip. During each BIST-enable clock cycle, the processor takes (instruction fetch and memory) data from its internal memory bus which is driven by an internal LFSR as shown in the Beta datapath diagram. The tristate driver which buffers the signal from the data BSCs must be then disabled (hardware is implemented which performs this) during processor BIST. The signature analyzer is connected to the outputs of the processor ALU. Many of the pseudo-random instructions passed to the processor will be invalid instruc-

tions (and thus treated as no-ops), but the ALU will still provide useful test data to have accumulated in the signature analyzer.

3.3.3 Adding BIST functionality to Boundary Scan Cells

To provide the slave modules with an input stream of random addresses and data, we modify the BSCs at the processor address, write-enable, and outgoing data lines. These modified address and write-enable BSCs are shown below.

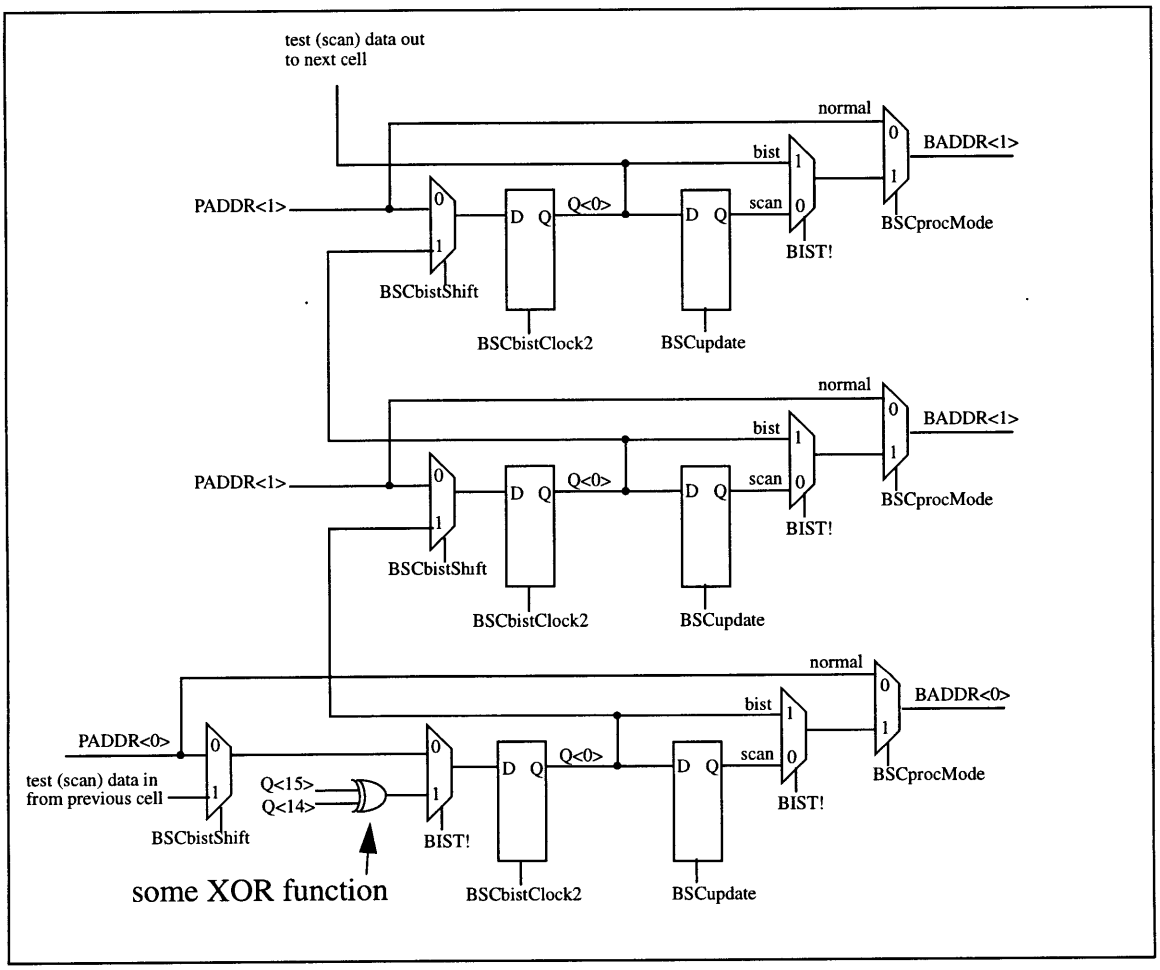

Figure 13. LFSR-enabled BSC

The XOR function is chosen so that a maximal number of values are cycled through before repetition.

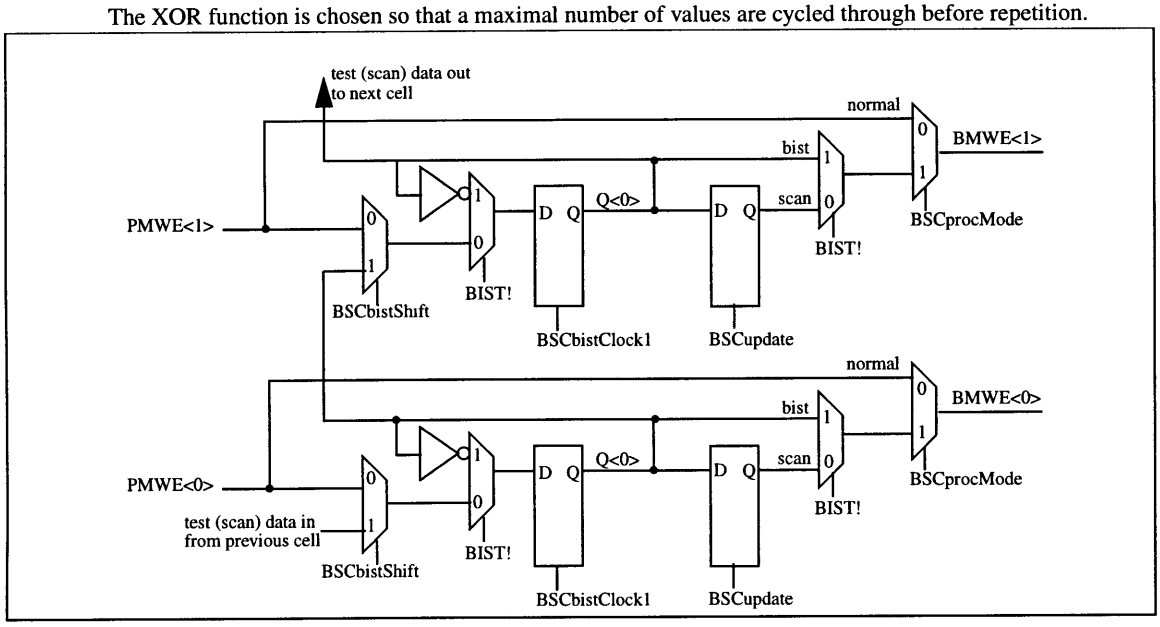

Figure 14. BSC for Processor Write-Enables
Write enables should be preloaded all high to start BIST testing. During BIST, they invert every cycle.

The sample/shift registers in the above BSCs implement a Linear Feedback Shift-Register function (as discussed in the Radio section) by allowing the first cell to take data from the XOR function, and allowing global clocks to update the sample/shift registers during BIST. The rest of the BSCs simply provide the shift function to complete the requirements for an LFSR.

Note that we additionally need to bypass the "update" registers to provide the pseudo-random sequence to the slave modules. The update register normally allows us to shift data transparently, thereby allowing discrete transitions into wanted states. During BIST, we are counting on every cycle's change to be input to the module.

Every BSC for the address lines then requires only one extra multiplexor to allow this bypassing of the update register. Additionally, another multiplexor, some routing, and a few XOR gates are needed for the first cell only.

The write-enable BSCs should be preloaded inactive for slave-module BIST2 (read-only). During this kind of BIST, the sample/shift register will still provide data to the exterior, however, it will not be clocked by the global clock. During 2-cycle slave BIST (write followed by read to the same address) the write enables toggle between all-high and all-low, and in this mode, the sample/shift registers should be clocked by the global clock. The values for the write-enables should be preloaded to an all-high or all-low values for this mode as well.

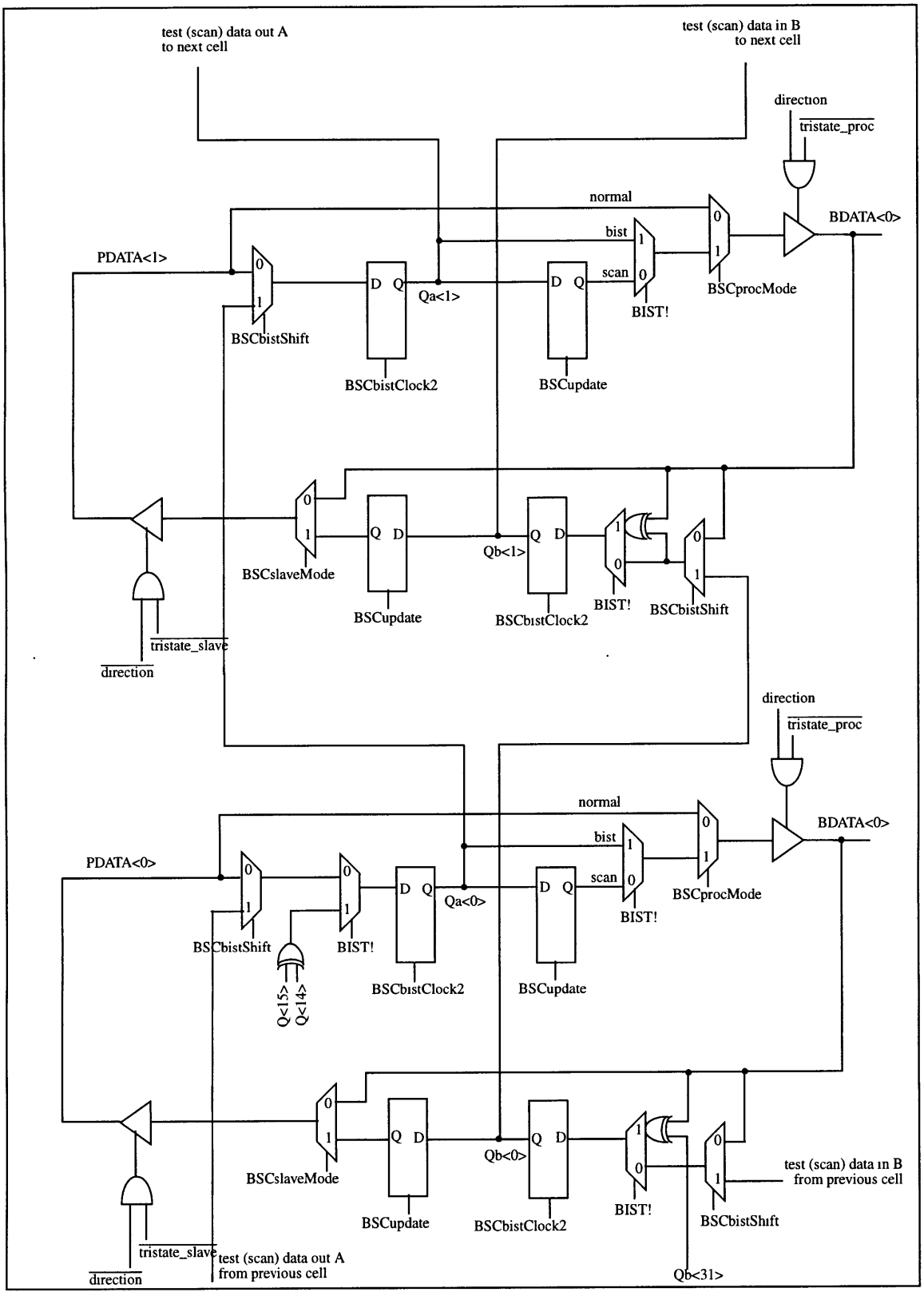

Figure 15. LFSR and Signature Analyzer at Processor Data I/O

3.4 User-Interface -- Test Structure Controls

The test structure controls are divided into two regions - user interface control (TAP pins and FSM controller) and on-chip instructions (the modes as encoded in the instruction register).

3.4.1 Test Access Port (TAP)

The Boundary-Scan registers are controlled via a Test Accessibility Port[5] (TAP). The TAP is a set of designated pins on the chip package through which the user may change the control signals to the test structures.

The IEEE implementation of the TAP focuses on using as few resources (connections to the outside world, or pins) as possible. The pins required to control these structures are listed in the following table:

TABLE 7. TAP Pins

signal	description
TRST*	Forces the TAP Controller FSM state bits to the RESET state, sets the instruction register to a default of sample/preload, and potentially presets all BSCs (LFSR and Signature Analyzers -- this is not implemented, so we are expecting that these will be preloaded to correct values, notably not the zero-value for the LFSRs)
TMS	Test Mode Select. This input determines the next state in the TAP FSM. Five rising edge clock signals with TMS asserted (high) will always result in the TAP Controller FSM going to the RESET state.
TCK	Test Clock. Clocks the TAP Controller FSM. May be clock gated in the CLOCK and the UPDATE signals passed to the BSC registers.
TDI	Test Data input - Scan in data comes from here.
TDO	Test Data output - Scan out data comes from here.

(*) denotes an optional pin

The TAP signals TCK, TRST, and TMS are inputs to an FSM (called the Tap controller FSM) which provides control signals to the rest of the machine. Each state in this FSM corresponds to an action that some test structures can be doing -- in some states, we are shifting data in the instruction register. In other states, we are capturing the parallel data into a scan chain data register.

The TAP Controller FSM is shown below.

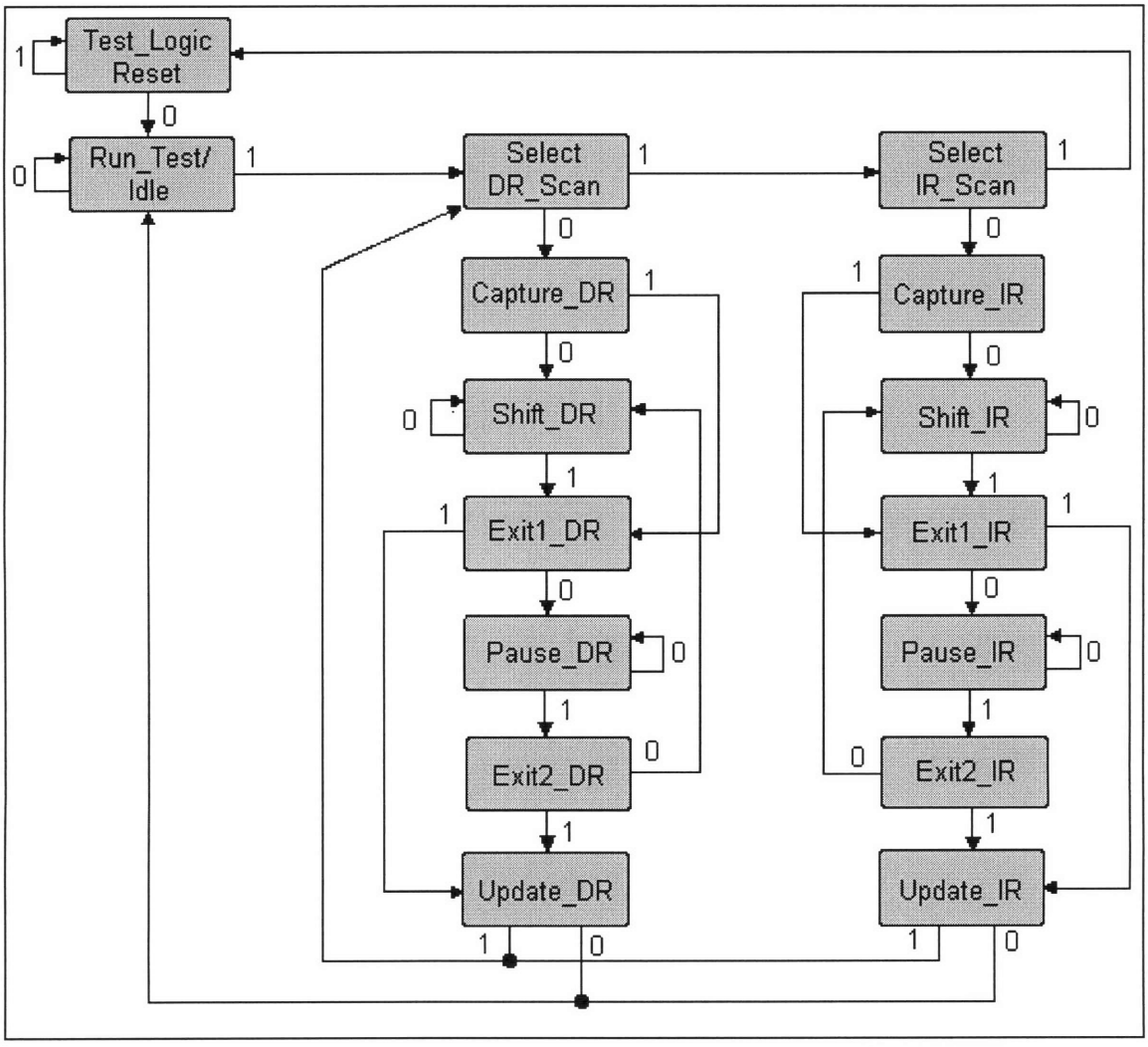

Figure 16. Tap Controller FSM
DRshift, IRshift, enable, and reset control lines are registered to avoid glitchy or hazards. This FSM is taken from a suggested design given in Weste and Eshraghian [x]

3.4.2 Instruction register

We use a suggested design for an instruction register given in Weste and Eshraghian [x]. This cell looks exactly like the basic boundary scan cells except there is no selector at the output.

The instructions in the instruction register are listed below. They differ by tristating different modules on-chip, which BSC chain to activate (that is, which BSC should be allowed to capture, shift, and update data), which modes the BSC are in, and if any self-testing mechanisms should be activated.

Either the processor or all the slave modules can be tristated so that a user may drive values at the external pins and emulate either the processor or the slaves. The two choices for active data register chains are the processor's internal test registers or the BSCs around the processor's externals. Each of these chains can be either transparent, or provide data to the chip modules, as determined by that chain's mode. BIST functions were described in the previous section.

TABLE 8. Instruction Register Chip Modes

Instruction Name	Bit Code	active data chain	tristated module	scan mode activation	Description
Sample/Preload "Normal" mode	000	processor external	--none--	--none--	Normal Operation. Transparent Observation
Read Processor's Internal Registers	001	processor internal	slave	--none--	Read Data from Processor Internal Scan Chain
Offchip Emulation of processor	010	--	processor	--none--	Processor outputs driven from offchip
Boundary Scan Emulation of processor	011	processor external	none	processor outputs	Processor outputs driven from the scan chain
Offchip Emulation of Slave Modules	100	processor external	slave	--none--	Slave outputs driven from offchip
Boundary Scan Emulation of slave modules	101	processor external	--none--	processor inputs	Slave outputs driven from the scan chain

Instruction Name	Bit Code	active data chain	tristated module	scan mode activation	Description
BIST1	110	processor external (no updating)	slave	processor in/outputs	Read-only built-in self test
BIST2	111	main	slave	processor in/outputs	2-cycle (write/read) built-in self test

The diagram below shows flow of control signal groups from the external pins to the boundary scan cells.

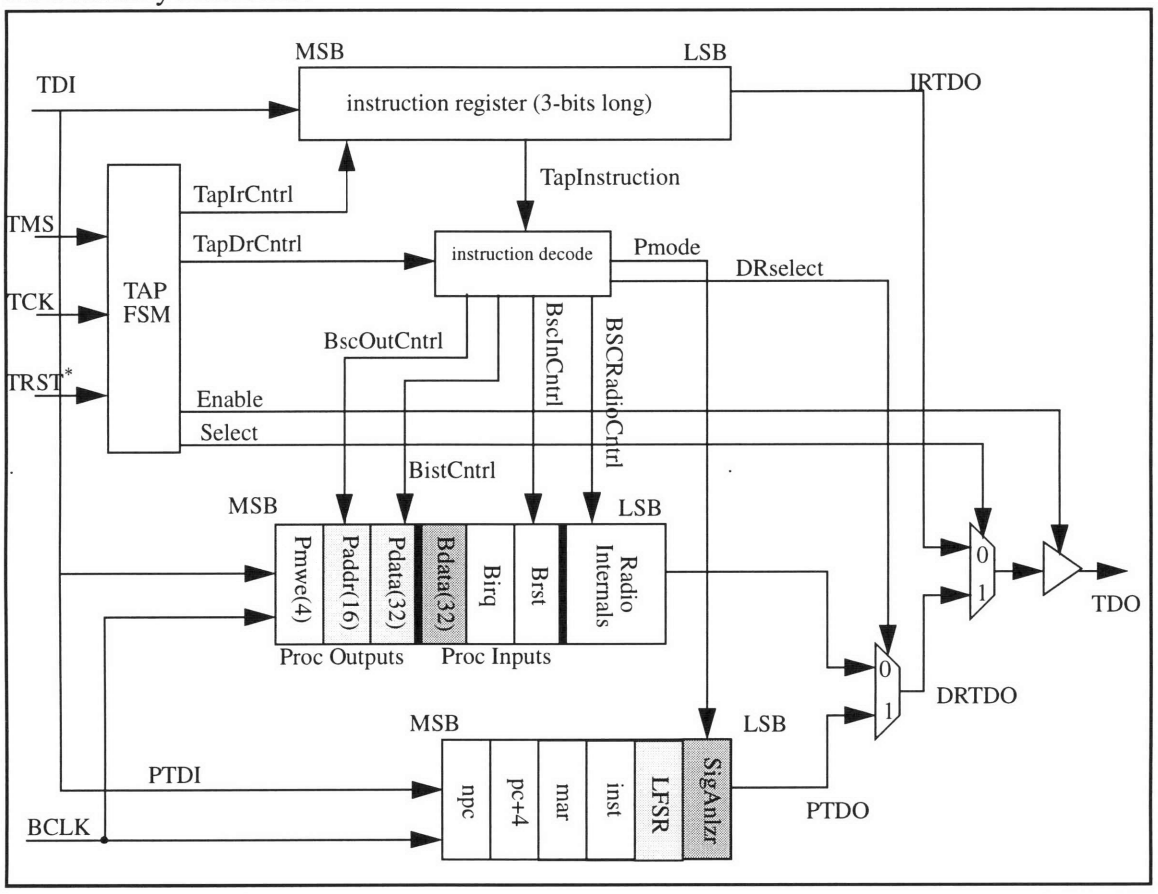

Figure 17. Test Control Signal Flow (in signal groups)
Shaded blocks are BIST-modified BSCs.

All registers in the processor's internal scan chain are 32-bits long. All registers in the Processor External Chain are 1-bit unless otherwise noted. There is presently no data accessed in an additional Radio Scan Chain.

When shifting data out of the processor's internal scan chain, the TAP controller FSM must be in the DRshift state, the instruction register must be set to Reading the processor's internal scan chain, and the Global Clock (BCLK) must be clocking.

All data in the processor scan chain is read-only except for the LFSR and the Signature Analyzer. All data in the processor external scan chain will be updated in the Update_DR (DRupdate) state if the processor external scan chain is active, so we must be careful to return the same value that we shifted out of the scan chain back in when we are done if we want to preserve the state of the machine.

Below is a table of signals which are used in testing.

TABLE 9. Test Control Signals

signal / group(s)	source / derivation
Pmode<1:0>	00 or 01 means normal operation, 10 means scan data, 11 means run built-in self test. Scan data when FSM is in DRshift state and TapIRstate is Read Processor Internal Scan. Processor expects GLOBAL clock signals in order to advance the test data in its scan chain (not the test clock). Run BIST when TapIRstate is BIST1 or BIST2 and TapFSMstate is Run-Test/Idle.
AddrOutRange	asserted when no slave modules are active. It is derived by NORing the module enable signals. This signal tells the signature analyzer not to be clocked during BIST read cycles, as there will be nothing driving the bus
direction (passed to all I/O BSCs)	asserted when any of the Bmwe<3:0> are high -- tristates one of the bi-directional drivers for processor and external I/O pins. Controls whether BIST address advances or not (advance when direction is low (processor is reading).
tristate_proc (all proc outs and I/O)	asserted when TapIRstate is Offchip Emulation of Processor - tristates processor outputs when active
tristate_slave (all proc ins and I/O)	asserted when TapIRstate is Offchip Emulation of Slaves or BIST1 or BIST2 - tristates slave module outputs when active
BSCprocMode (all proc outs and I/O)	asserted when TapIRstate is Scan Emulation of processor or BIST1 or BIST2 -- processor outputs come from scan chain when active. Controls the mode selector for the Address and write-enable pins.
BSCslaveMode (all proc ins and I/O)	asserted when TapIRstate is Scan Emulation of Slaves - processor inputs come from scan chain when active

signal / group(s)	source / derivation
BSCenable	Enables the Processor External Scan registers for shifting, sampling, and updating. Occurs when instruction register word is anything except Processor Internal Scan. This signal is the negation of the processor's internal-scan enable signal.
BSCclock	clock signal to all other boundary scan signals on the chip (the ones not enabled for BIST) -- excluding the processor's internal scan registers. This is simply DRclock AND BSCenable.
BSCshift	DRshift AND BSCenable
BSCupdate	DRupdate AND BSCenable
BIST! All BIST-modified BSCs	asserted when TapIRstate is BIST1 or BIST2 and TapFSMstate is Run-Test/Idle. controls two muxes -- one allows LFSR and Signature Analyzers to XOR and shift in new data, other brings that data to the exterior, bypassing the update register.
BSCbistClock1	clock signal to the "sampling" register in the boundary scan cells at the write-enable lines. This signal follows the global clock (BCLK) when BIST! is asserted. Otherwise this signal follows DRclock AND BSCenable AND NOT((BIST1 or BIST2) AND DRcapture).
BSCbistClock2	clock signal to the "sampling" register in the boundary scan cells which have LFSR or signature analysis capability. This signal is exactly like BSCbistClock1 but this only follows BCLK if direction is inactive (update only on reads)
BSCbistShift	asserted when DRshift AND BSCenable, or BIST!

3.5 Radio Testing

The analog portions of the chip consist of transistors so large that they are probabilistically going to be error-free (in terms of fabrication faults). The testing method we will use for this piece involves connecting the output of the chip's transmitter to the input of the receiver. The data is then placed in the outgoing buffer and a known number of clock cycles are provided. Since the group delay is constant, the arrival time of the data is known, so if the same data that we intended to send returned to the incoming buffer, then we can be somewhat satisfied that our radio is working. The memory addresses needed to perform these operations are described in the Radio Module section.

Chapter 4 Floorplan and Physical Layout

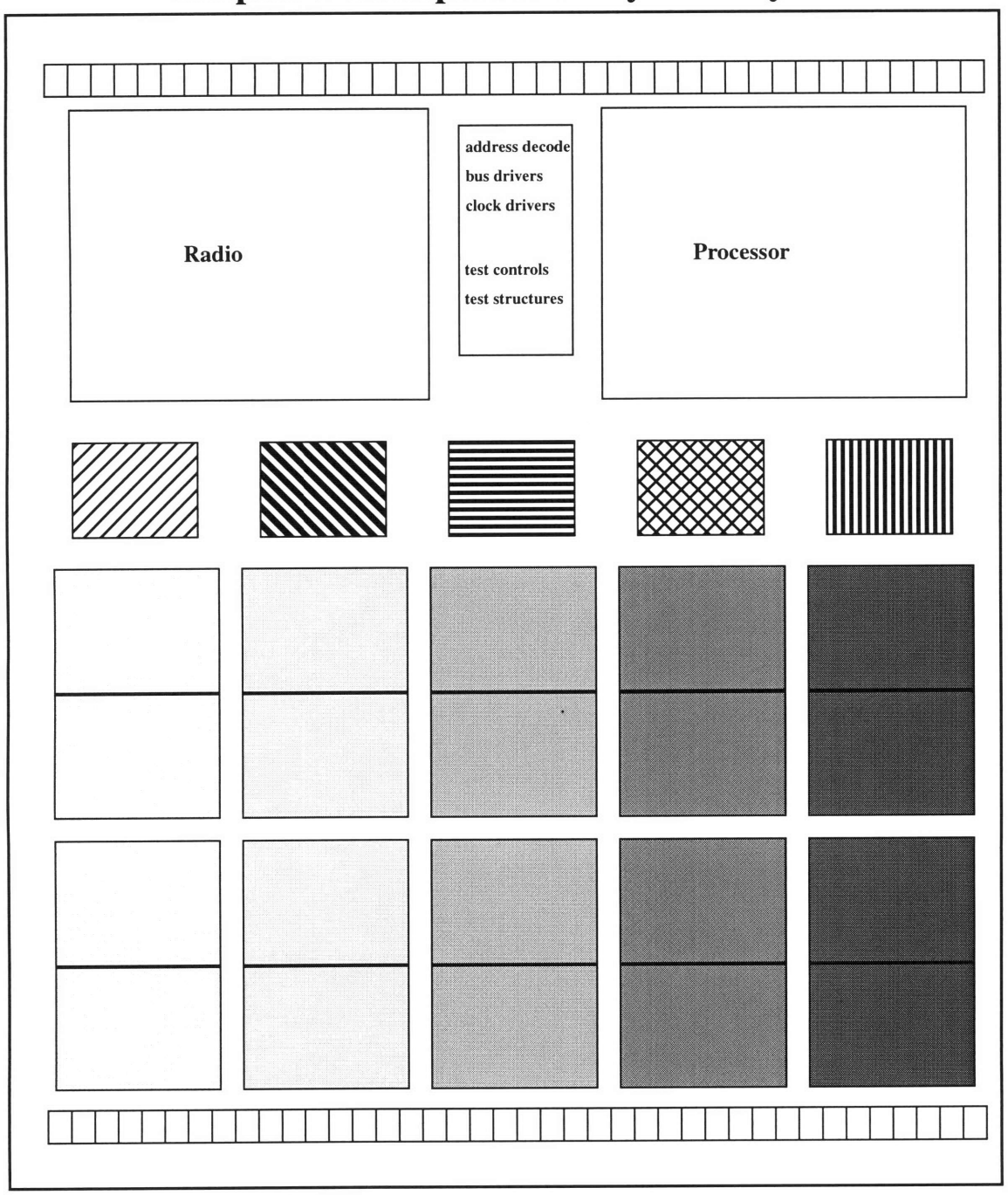

Figure 18. Fullchip integration
Note: Not to scale, but approximate relative sizes are shown

4.1 Module Sizing and Placement Strategy

The extent of fullchip design was limited by the lack of definition of the module dimension specifics. This is partly due to portions still being designed. Nevertheless, a lay-

out has been specified such that we should be able to plug-in all the modules by the time they are finished.

Floorplan design began with estimating module sizes. The following information was used to determine the layout above, which allows us 80 Input/Output pads, has a regular memory structure, and gives us enough room for guard rails, power and clock distribution, and bus routing:

TABLE 10. Module Size Specifications

Cell/Module	Dimensions: (WxH) in μm	Area (mm^2)	Dimensions: (WxH) in lambda	Area ($lambda^2$)
Die	6,800.0 x 6,800.0	46.2400	22,666 x 22,666	513 sq klambda
Beta Processor	2,500.0 x 1,600.0	4.0000	8,333 x 5,333	44 sq klambda
Radio w/ grd rng	2,500.0 x 1,600.0	4.0000	8,333 x 5,333	44 sq klambda
1 KB RAM cell	1,297.2 x 933.0	1.2100	4,324 x 3,110	13 sq klambda
4 KB ROM cell	1,133.4 x 779.1	0.8830	3,778 x 2,597	~10 sq klambda
Clock Driver	240.0 x 240.0	0.0576	800 x 800	0.64 sq klambda
TAP Controller & Instruction Register & Decode	240.0 x 240.0	0.0576	800 x 800	0.64 sq klambda
BIST-enabled scan address cell	300.0 x 480.0	0.144	1,000 x 1,600	1.60 sq klambda
BIST-enabled scan data I/O cell	300.0 x 960.0	0.288	1,000 x 3,200	3.20 sq klambda
Entire BSC chain	300.0 x 1,650.0	0.495	1,000 x 5,500	5.50 sq klambda
Corner I/O Pad	219.3 x 219.3	0.048	731 x 731	0.53 sq klambda
Other I/O Pad	167.4 x 219.3	0.037	558 x 731	0.41 sq klambda

4.1.1 Layout Specifics

We are using a 0.5 μm process, which corresponds to the minimum feature length of a transistor in this process. We are also using lambda-design rules and the Magic layout editor for this design, in which lambda represents 0.3 μm. A minimum-sized transistor is represented in this technology with a two-lambda long channel.

In the general floorplan design, an attempt was made to isolate the radio from digital noise (mainly in the processor) while at the same time maintaining structures memory arrays. Spacing was left for guard rails which surround the I/O pads to prevent latch-up. Guard rails are also used around the radio, since crosstalk can be reduced by placing supply lines around the module which will act as some kind of capacitive shielding for the analog circuitry. Internal bus routing and power/clock distribution are discussed below.

4.1.2 Layout Height Spacing

The difference between the sum of module's heights and the total height of usable space on the die is 834 lambda. Of this, 257 lambda is given both above and below the area designated for the processor and radio. The processor and the radio are assigned boundary boxes of the same size, and are placed at the same height on-chip). This amount is given in order to allow a spacing of 25 lambda for each guard rail. Approximately 70 lambda is allocated for clock, power, and ground lines, which may be in the range of 20 lambda wide each. This leaves 137 lambda to route signals to the edges of the circuit. This will allow about 10 lambda for about 10 internal bus lines which need to be routed to the corners of the chip. We shouldn't have to allocate space for routing for those pins near the middle of the I/O pads, since these should be easily accessible.

Between the RAM and ROM cells we have left 150 lambda. About 70 lambda will be dedicated to power, ground, and clock distribution as before. The rest will be used in order to convert the wiring pattern from the ROM scheme to the RAM wiring scheme.

There is a 70 lambda gap between the top portion of RAM cells and the bottom portion. This allows another line at which we can strap the supply lines into our grid.

The remaining 100 lambda is left between the bottom of the SRAM cells and the bottom I/O pads. Other than the guard rail, this space will be used to attach bus lines to the I/O pads. Each of the 5 RAM modules along the bottom should interface to 8 I/O pads, which should be straightforward with the leftover 75 lambda.

4.1.3 Layout Width Spacing

The I/O pads come in two varieties -- chip corner pads, which are 731 lambda on each side of its square layout, and regular I/O pads, which have a width of 558 lambda. These tile together to perfectly fit inside the 22,666 lambda frame, with a corner I/O pad on the end of each line, and 38 I/O pads between.

The Radio is placed along the left edge, although we may move it some distance in should that become convenient for routing some supply lines. We anticipate that the radio's analog pin may lie within the module's boundary box definition, and along the edge of the left side when it is complete. The processor is placed flush along the right edge, unless need arises to wire things on the its right side. This leaves us with approximately 6,000 lambda in the center to place test circuitry, drivers, random logic, and perhaps one more memory cell.

Each ROM is centered relative to its vertical stack of RAM cells. This leaves 273 lambda between the edge of the ROM cell and the corresponding edge of the RAM in its vertical stack on either side. In addition to this space, there is 210 lambda between each RAM cell's side. This leaves 103 between the chip's left and right edges and the closest RAM cell.

4.2 I/O - Pinout

The I/O pads are placed only along the top and bottom edges of the chip (there are none along the side edges). Power, ground, and clock have 7 pins each in order to reduce inductance and resulting oscillations.

TABLE 11. Chip Pinout

Pin #	Name	Pin #	Name	Pin #	Name	Pin #	Name
1	PWR1	21	CLK2	41	GND4	61	PWR6
2	Bdata<2>	22	Tdo	42	Bdata<31>	62	Bdata<16>
3	Bdata<1>	23	Birq	43	Bdata<30>	63	Bdata<15>
4	Bdata<0>	24	Baddr<12>	44	Bdata<29>	64	Bdata<14>
5	GND1	25	PWR3	45	CLK4	65	GND6
6	Baddr<15>	26	Baddr<11>	46	Bdata<28>	66	Bdata<13>
7	Baddr<14>	27	Baddr<10>	47	Bdata<27>	67	Bdata<12>
8	Baddr<13>	28	Baddr<9>	48	Bdata<26>	68	Bdata<11>
9	CLK1	29	GND3	49	PWR5	69	CLK6
10	Bmwe<3>	30	Baddr<8>	50	Bdata<25>	70	Bdata<10>
11	Bmwe<2>	31	Baddr<7>	51	Bdata<24>	71	Bdata<9>
12	Bmwe<1>	32	Baddr<6>	52	Bdata<23>	72	Bdata<8>
13	PWR2	33	CLK3	53	GND5	73	PWR7
14	Bmwe<0>	34	Baddr<5>	54	Bdata<22>	74	Bdata<7>
15	Brst	35	Baddr<4>	55	Bdata<21>	75	Bdata<6>
16	Trst	36	Baddr<3>	56	Bdata<20>	76	Bdata<5>
17	GND2	37	PWR4	57	CLK5	77	CLK7
18	Tms	38	Baddr<2>	58	Bdata<19>	78	Bdata<4>
19	Tck	39	Baddr<1>	59	Bdata<18>	79	Bdata<3>
20	Tdi	40	Baddr<0>	60	Bdata<17>	80	GND7

Chapter 5 Conclusion

5.1 Notes for Improvements

Since performance in the system is now limited by communication speed, we begin by looking for improvements in the radio. Using a more sophisticated modulating strategy may allow a greater symbol rate. This means that we may be able to maintain the data rate of one symbol per 5.12 msec, but we may be able to represent more than one bit of information in each symbol. However many more bits we can reliably encode, we will increase the communication rate by the same factor. The problem with this is that such transmission schemes like QAM, PAM, and PSK, complicate both the transmitting and receiving portions of the chip. Design time was an important reason why our radio was implemented with this simplified radio.

Future designs may use specialized signal processing hardware which will enable more sophisticated transmissions. These digital signal processors (DSP) will probably include hardware units which can perform multiply and accumulates in one cycle -- a typical operation in computing convolutions or fast Fourier transforms. Signal processors also tend to be able to optimize for expected tight loop operations. Currently, the amorphous computing group is looking into buying signal processing cores (i.e., the Hitachi HF-3) and building their own specialized signal processing hardware blocks. If we use the processor to implement these signal processing computations, performance may again be a design issue. Even if the processor is not used for performing DSP calculations, any increase in the communication rate will make us begin to check whether computation speed is a limiting factor in the system.

Besides going to buy intellectual property (IP) processing cores, we discuss ways in which we can increase our own processor's performance. One way to do this is to provide an instruction cache. This exploits our knowledge that again, many programs will have large numbers of loops in performing computations with certain structures. While we do not have very much memory to spend on a data cache (which typically need to be very large), an instruction cache can keep a small number of instructions and still be effective in decreasing the number of memory accesses. This will be useful in any implementation which uses our singe bus for both instruction and memory data, but it will only be useful on instructions that have memory accesses.

Other efforts to increase radio bandwidth may include the use of differential signals, which automatically filter out common mode noise. We may also add hardware for multiple transmitters or receivers per chip, and simply set them to different channel codes.

Besides increasing the performance in the radio and processor, we will always be looking to increase the amount of memory on the chip. More memory space makes programming more flexible and allows for more complex operations.

5.2 Working Towards the Amorphous Computing Goals

Amorphous computing has already discovered that true random number generation in processors will be very useful in assigning (probabilistically) unique identifiers to each processor, which is important in setting up chains of communication, and other hierarchical structures. Implementing a hardware random number generator in the amorphous computer is therefore a goal. This random number generation can be done using a zener diode,

which provides a good statistical white-noise (truly random) output. This is qualified by the need to biased the diode at the correct voltage [6].

Amorphous computers will eventually also need a clock generation unit on chip. This break from synchronicity with the rest of the chips will cause a certain amount of redesign, as the current design assumes synchronized global clock signals. The vitality of this synchronicity can be seen in the radio unit -- the transmitter and receiver pair will work much less effectively if out of synchronization.

As mentioned before, we will be working towards lower pin counts. This is most quickly solved by removing the address and bus data I/O pins. Efforts will also be made toward low power design. Amorphous chips are eventually intended to be powered by the heat of a human being, solar cells, or wind. Unused portions of the chip may be shut down, and alternative designs for shift registers may be used, since shift registering tends to cause latching of many potentially unnecessary values.

5.3 Acknowledgements

I would like to thank all the members of the amorphous computing group for their support. In particular, Chris Hanson, Don Allen, and Dr. Chris Terman were tremendous sources of encouragement, knowledge, and wonderful humor.

References

[1] Hal Abelson, Tom Knight, Gerry Sussman.
Amorphous Computing,
White Paper, October, 1995.
http://www.ai.mit.edu/abstracts.html#bioinsp

[2] Daniel Coore, Radhika Nagpal, Ron Weiss.
Paradigms for Structure in an Amorphous Computer,
MIT A.I. Memo No. 1614, October 6, 1997.
http://www.ai.mit.edu/abstracts.html#bioinsp

[3] Massachusetts Institute of Technology,
Department of Electrical Engineering and Computer Science.
6.004 Beta Instruction Set Architecture Reference. March 12, 1998.
http://www.cag.lcs.mit.edu/6.004/Beta/isa_ref/P001.html

[4] Neil H.E. Weste, Kamran Eshraghian.
Signature Analysis and BILBO, Ch.7.3.4.1.
Principles of CMOS VLSI Design. Addison-Wesley Publishing Company,
Reading, Massachusetts. Copyright by AT&T, October 1994.

[5] Texas Instruments.
Boundary-Scan and IEEE Std 1149.1, Ch. 3 of *1149.1 Testability Primer.*
http://www.is-bremen.de/~axel/b.scan/c3.htm

[5] Neil H.E. Weste, Kamran Eshraghian.
Test Accessability Port (TAP) and TAP Controllers,
Ch.7.5 of *Principles of CMOS VLSI Design.* Addison-Wesley Publishing Company,
Reading, Massachusetts. Copyright by AT&T, October 1994.

[5] LaNae Joy Avra
Synthesis Techniques for Built-in Self-Testable Designs,
CRC Technical Report No. 94-6, CSL TR 94-633, July,1994. Center for Reliable
Computing. Computer Science Laboratory, EECS Dept, Stanford University

[6] Jonathan P. How, Steven R. Hall.
Local control design methods for a hierarchic control architecture.
Guidance, Control and Dynamics, 15(3), May-June 1992.