An I/O Port Controller for the MAP Chip

by

Albert Ma

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Engineering

and

Bachelor of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1997

Author . . . .
Department of Electrical Engineering and Computer Science
May 23, 1997

Certified by . . . .
William J. Dally
Professor
Thesis Supervisor

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

An I/O Port Controller for the MAP Chip

by

Albert Ma

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 1997, in partial fulfillment of the
requirements for the degrees of
Master of Engineering
and
Bachelor of Science in Electrical Engineering and Computer Science

Abstract

This thesis describes the design and implementation of the I/O port controller for the MAP chip as part of the M–Machine project. The I/O port controller is responsible for managing the I/O port, which is used for communication between the MAP chip and external devices. The I/O port is intended to be connected to an off-chip module that then connects to other peripheral buses, such as SCSI or SBUS. The I/O port controller is implemented in Verilog HDL and synthesized to a standard cell library.

Thesis Supervisor: William J. Dally
Title: Professor

Acknowledgments

There are many without whom this thesis would not be possible. First of all I would like to thank God for the life he gave me and to give him all the glory. I would also like to thank Professor Bill Dally for giving me the opportunity to work in his group. The past year has been the most grueling but most rewarding time for me at MIT. I learned so much by working on this real world type project. Many thanks go to Steve Keckler and Andrew Chang for being so patient with me and all of my questions. Also, thank you to Keith Klayman for his work on the GCFG and help with the synthesis and timing analysis.

The greatest thanks go to David Um, Hong Min, and James Won who prayed for me throughout this time. Especially to Hong who gave me rides and brought late-night food.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis presents the design and implementation of the I/O port controller on the M–Machine[2]. The M–Machine is an experimental multicomputer being built by the Concurrent VLSI Architectures Group in the Artificial Intelligence Laboratory at the Massachusetts Institute of Technology. It is being developed to test architectural concepts motivated by the constraints of modern semiconductor technology and the demands of programming systems.

1.1 Architectural Overview

The M–Machine consists of a collection of computing nodes interconnected by a bi-directional 2–D mesh network, as shown in Figure 1-1. Each six-chip node consists of a multi–ALU (MAP) chip and 1 MW (8 MBytes) of synchronous DRAM (SDRAM) with ECC. The MAP chip includes the network interface and router, and provides bandwidth of 800 MBytes/s to the local SDRAMs and to each network channel. A user accessible message passing system yields fast communication and synchronization between nodes. Rapid access to remote memory is provided transparently to the user with a combination of hardware and software mechanisms.

As shown in Figure 1-2, a MAP contains: three execution clusters, a memory subsystem comprised of two cache banks and an external memory interface, and a communication subsystem consisting of the network interfaces and the router. The

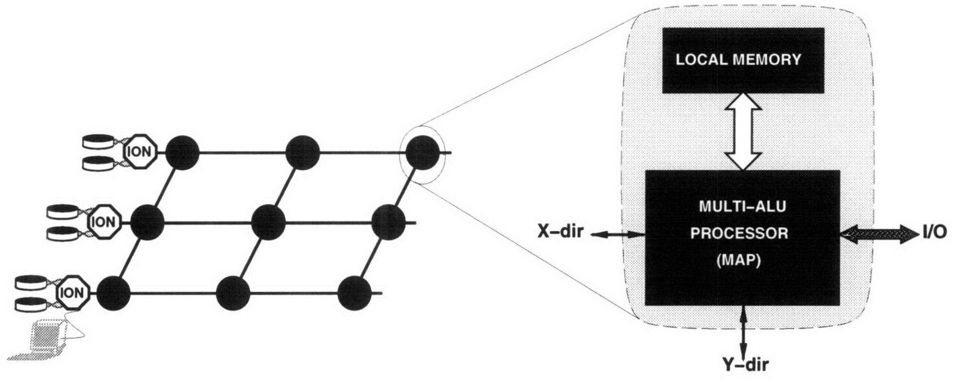


Figure 1-1: The M–Machine architecture.

multiple function units are used to exploit both instruction-level and thread-level parallelism. Two crossbar switches interconnect these components. Clusters make memory requests to the appropriate bank of the interleaved cache over the M–Switch (MSW). The C–Switch (CSW) is used for inter-cluster communication and to return data from the memory system.

Words on the M–Machine are 66-bits wide. This is comprised of 64 data bits, 1 synchronization bit, and 1 pointer bit. The synchronization and pointer bits are unique to the M–Machine. They allow it to more efficiently implement synchronization and memory protection.

The MAP architecture includes global condition code (CC) registers. Similar to data registers, each global CC register has an accompanying scoreboard bit. These values are broadcast to the three clusters. Based on the CC register values, conditional branches and assignment operations can be executed.

In order to provide communications between the M–Machine and the outside world, I/O devices will need to be connected to the M–Machine. For this purpose, a dedicated I/O port is provided on the MAP chip on each node of the M–Machine. The I/O port is intended to be connected to an off-chip slave, perhaps implemented in an FPGA, that then connects to other buses, such as SCSI or SBUS. These in turn would connect to mass storage devices such as hard drives or to host interfaces. Figure 1-3 shows the above relationships.

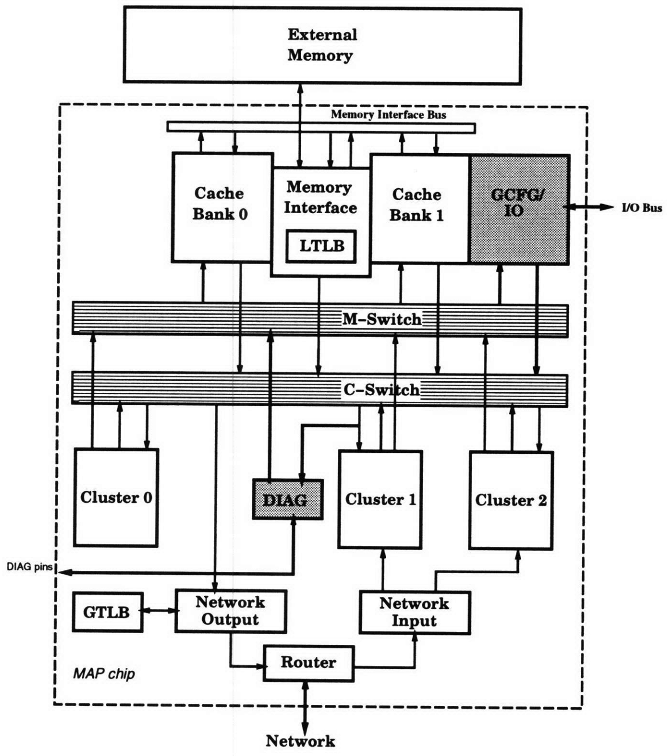


Figure 1-2: The MAP architecture.

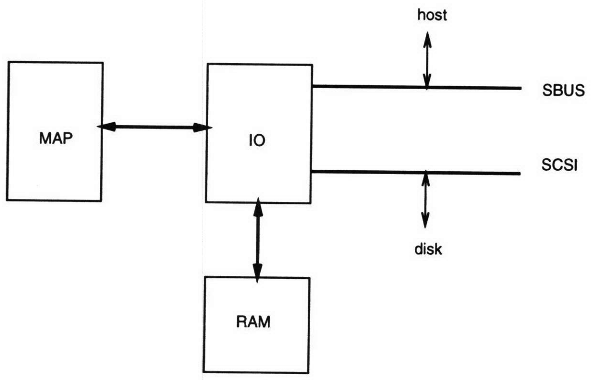


Figure 1-3: IO Port Overview

The I/O port controller is responsible for managing the I/O port. It implements the software interface that allows programs to access the I/O port and the low-level communications protocols that allow the MAP chip to communicate with the off-chip slave I/O processor.

My role in the M–Machine project was to design and implement the I/O port controller in Verilog HDL as well as verifying its correctness in all aspects.

1.2 Thesis organization

The thesis shall be organized as follows. Chapter 2 describes the architecture of the I/O port controller and its interfaces. Chapter 3 details the I/O port interfaces and presents the timing design. Chapter 4 describes the implementation and design of the I/O port controller in Verilog. Chapter 5 discusses the verification. Chapter 6 concludes the thesis.

Chapter 2

I/O Port Controller Architecture and Interfaces

This chapter discusses the architecture of the I/O Port Controller in terms of the interfaces between the controller and the off-chip slave, the MAP chip, and the software.

2.1 Design constraints and considerations

A major design consideration was the intended use of the I/O port. The port will be used primarily to communicate with two classes of devices. The first class consists of storage devices, especially disk drives. The second class consists of interfaces to the outside world, for example, a console display or a host interface. These two classes have very different requirements and properties. Disk drives require high-bandwidth and are block oriented. Thus the port needs to be able to transfer a large amount of data at a high data rate. Host interfaces require low-bandwidth and are character oriented. Thus the port ought not impose a large overhead for character transfers in terms of bus utilization and latency. In addition, the port needs to be flexible enough to handle a wide range of devices beyond these two classes and thus should not make arbitrary assumptions about what is being connected to it.

Another major consideration in the design of the architecture was to make the I/O port controller as decoupled from the rest of the MAP chip as possible. The I/O port controller was one of the last modules to be specified and designed on the MAP chip and so had to be designed in a way that would not require modifications to the rest of the MAP chip.

2.2 Off-chip Interface

The design of the off-chip interface follows a layered approach; this is similar to the way networks are designed. First, the physical link and low-level protocol are designed. These are concerned with getting information from the MAP chip to the off-chip slave and vice versa. Built upon this are the mid-level and high-level protocols. These structure and give meaning to the information transmitted. These protocols are all implemented in the I/O port controller hardware. Higher level protocols can be easily implemented on top of this in software and on the off-chip slave.

2.2.1 Physical link design

The link uses a 18-bit bi-directional bus running at half the system clock rate, that is, 50 MHz. We chose this clock rate because running the wires at 100 MHz is both difficult and unnecessary. In addition, it would be very hard to run the off-chip slave processor at 100 MHz, especially if it were to be implemented on an FPGA. Also, running the bus between 50 MHz and 100 MHz would require the bus to be run asynchronously, which adds some complexity to the design. Finally, running at anything less than 50 MHz would be a waste of pin bandwidth.

Using an 18-bit bus allows us to transfer a full 66-bit word in 4 I/O clock cycles. This gives a peak bandwidth of 12.5 MW/s or 100 MB/s. We decided this to be a good bandwidth, sufficient for its planned use but, at the same time, not overly wasteful. In comparison, ultra-wide SCSI II has a peak bandwidth of 40 MB/s, 400 MHz FireWire has a peak bandwidth of 50 MB/s, and PCI has a peak bandwidth of 132 MB/s. We expected most traffic to be one-way bursts of data, as in reading or writing from/to a drive; thus, we chose a bi-directional bus to save 18 pins.

2.2.2 Low-level transmission protocol

The low-level transmission protocol needs to guarantee three conditions to prevent data from being lost. First, it must ensure that both sides do not try to drive the bus simultaneously. Second, when one side transmits, the other side must know to grab data off the bus. Third, when one side cannot receive, as when its buffers are full, the other side must know not to transmit so that data does not get lost.

To maximize bus utilization, we chose to use a synchronous transmit/ready protocol. Transmit indicates the availability of data to be transmitted on the next I/O clock cycle. Ready indicates the availability of resources to receive data on the next I/O clock cycle. A side is able to transmit if both its own transmit line is high and the other side's ready line is high. If only one side is able to transmit, then that side goes ahead and transmits on the next I/O cycle. In addition, the other side latches the data at the end of that cycle. If both sides are able to transmit, a bit of state determines which side gets to transmit. This bit is duplicated on the MAP and on the off-chip slave processor. On reset, the bits are set so that the MAP has priority. When either side transmits, the bit is reset so that the other side gets priority. This scheme guarantees that, when both sides want the bus, both sides get the bus exactly half the time. This is important to ensure that both sides make progress and cannot be locked out. Figure 2-1 shows an example of the protocol.

Several alternative protocols were considered. The leading candidate was transmit/acknowledge. In this protocol, the acknowledge signal indicates that data was successfully received on this cycle. The advantage of this protocol over the chosen protocol is that it is simpler. It does not require that either side knows that its buffers will be free on the next cycle, just whether its buffers are free on the current cycle. The disadvantage of this protocol is that some bus bandwidth is wasted when both sides want to transmit but only one side is able to receive. We reasoned that the extra complexity of the chosen design is minor and is warranted by the improved performance.

A minor variation compresses the transmit and ready lines onto a single wire that is clocked at the system clock rate; the transmit and ready signals would be driven on

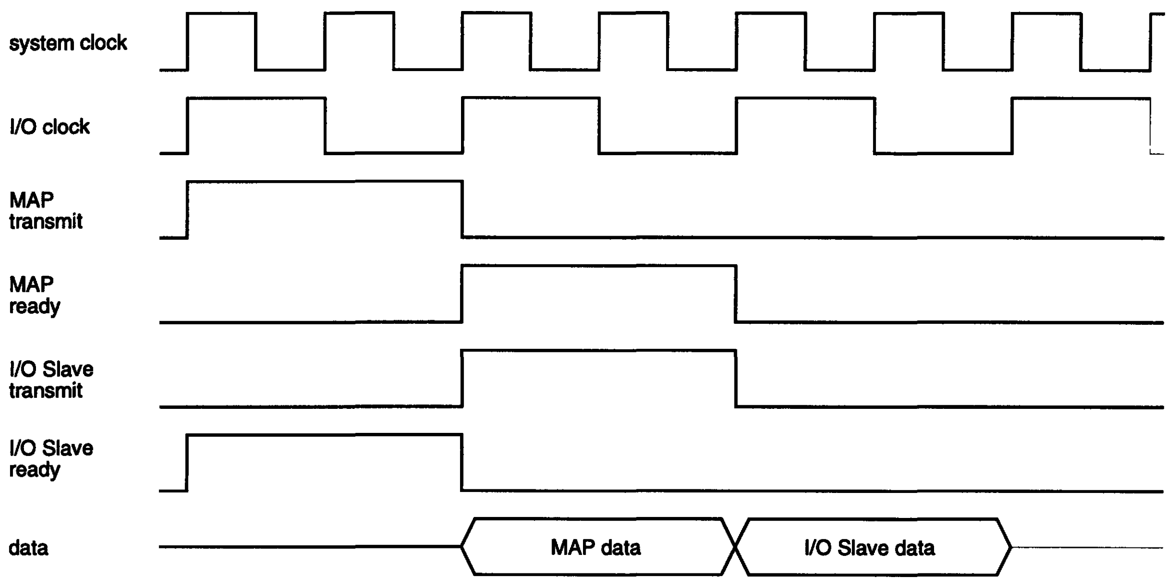


Figure 2-1: I/O Port protocol. The first transaction is sent by the MAP chip; the second transaction is sent by the I/O chip.

alternate system clock cycles. The advantage of this is that two signal pins (and up to two ground pins) are saved. The disadvantages are tighter timing constraints, extra logic complexity, and more difficult external debugging from reduced observability. The tighter timing and reduced observability favored eliminating this variation.

2.2.3 Mid-level protocol

The mid-level protocol organizes consecutive 18-bit packets into variable length words. The top two bits ([17:16]) of the first packet of a word indicates the number of packets there are in the word. This can be one to four. The mid-level protocol needs to be implemented in hardware for performance reasons. If the software was responsible for sending each packet, it would have to issue an I/O instruction on average every two system clock cycles to fully utilize the I/O bus. This is tricky to write in software and would waste much precious M–Switch and C–Switch bandwidth. By organizing the packets into words, an I/O instruction need only issue on average every eight cycles to fully utilize the I/O bus. Alternatively, we could eliminate the count bits and always send a four packet word. This would save one pin. However, this wastes bandwidth when the words are actually fewer than four packets long.

2.2.4 High-level protocol

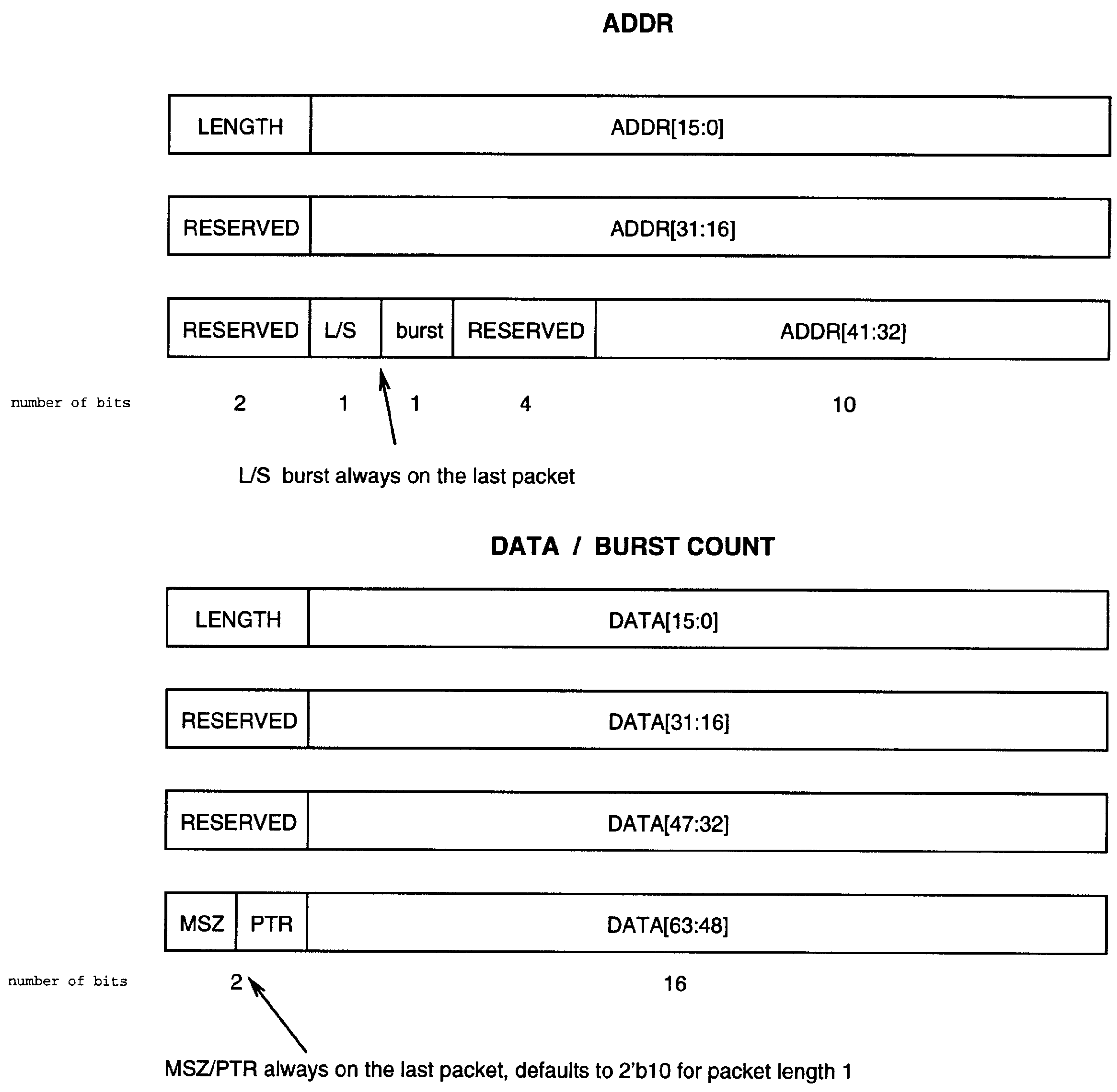
The high-level protocol implements a memory-mapped I/O abstraction with an extension for burst transfers. In this abstraction, all I/O is performed through loads or stores to memory locations in an I/O address space; all I/O devices are mapped onto that address space. For example, we could map an entire 4 GB drive onto the range of addresses 4 GB through 8 GB, console output to address 0, keyboard input to address 4, and so on. The assignment of addresses is completely up to the higher-level protocols.

To implement this, three I/O word types are defined; they are: address words, data words, and burst count words. The address word can be one, two, or three packets long. An address word contains the I/O address (up to 42 bits long because of the software layer) to load or store to. In addition, it contains 2 bits of command information. Bit 15 of the last packet of the word indicates whether the operation is a load, indicated by 1, or a store, indicated by 0. Bit 14 of the last packet of the word indicates whether the operation is a burst initiate instruction. The address is stored in a little-endian format in the packets as shown in figure 2-2.

The data word can be one to four packets long. Bits 17 and 16 of the last packet are the synchronization and pointer bits of the word. In the case of a 1-packet word, the synchronization is assumed to be 1 and the pointer bit is assumed to be 0. The data bits are stored in a little-endian format in the packets as shown in figure 2-2.

The burst count word can be one to four packets long, though rarely does it actually exceed one packet. It is similar to a data word, but without the synchronization and pointer bits.

Every transaction begins with an address word from the MAP chip; the off-chip slave cannot initiate a transaction. If the transaction is a load (non-burst initiator) instruction, the off-chip slave, at some point in the future, responds with a data word which is the result of the "load" from the address indicated by the address word. It is up to the off-chip slave processor, and thus the higher-level protocols, to decide the size of the data words that it returns. If the transaction is a store (non-burst initiator) instruction, the address word is followed by a data word from the MAP chip.



LENGTH		L/S		burst	
00	1	0	STORE	0	normal
01	2	1	LOAD	1	initiate burst
10	3				
11	4				

Figure 2-2: I/O Port Packet Structure

This data word is "stored" at the I/O address indicated by the address word.

If the transaction is a burst initiator, the address word is followed by a burst-count. In addition, if it was a load burst initiator, the off-chip slave, at some point in the future, responds with some number of data words as the result of the load. The number of data words is indicated in the burst-count transmitted by the MAP chip. If the transaction is a store burst initiator, the MAP chip, following the burst-count, sends out some number of data words as indicated by the burst-count.

The reason for the burst commands is to maximize bandwidth on block data transfers, as to a disk drive. Without burst commands, we would have to send an address word for every data word sent. The current scheme thus increases available bandwidth by 75%, assuming 3 packet address words and 4 packet data words.

2.3 MAP interface

There is no mechanism available to do DMA (direct memory access) transfers of data between memory and the I/O port. To add one would have required hooks in the external memory interface and in the cache banks and throughout the memory system. This was far too costly in terms of design time. Therefore, I/O transfers had to go through the register files. The most natural and easy way was through software I/O. A load or store has to be issued for every word of data transferred through the I/O port. Doing I/O through software avoids countless problems with synchronization and scoreboarding and makes for an extremely clean interface with the rest of the MAP chip.

For the most part, the datapath already existed for this to happen. Memory instructions (that access I/O) are issued in the clusters, go through the M–Switch, and get picked up by the I/O port controller. Data coming in through the I/O port is sent by the I/O port controller through the C–Switch, into the clusters and are written into the register files. What was missing, though, were the ports on the M–Switch and C–Switch for the I/O port to connect to. Thus we had two alternatives, to create the extra ports, or to share the ports with another module. The first alternative wasn't really an option, as it would mean changing to many things in the design including the switch datapaths. Thus we chose to share the port with the GCFG. This was a natural choice. The GCFG is very rarely accessed, so it ports are generally free. In addition, it had room in its address space to accommodate the I/O address space. Sharing the ports mean that the GCFG and the I/O controller need to be physically close to one another. Figure 2-3 shows the the relevant datapaths and interfaces

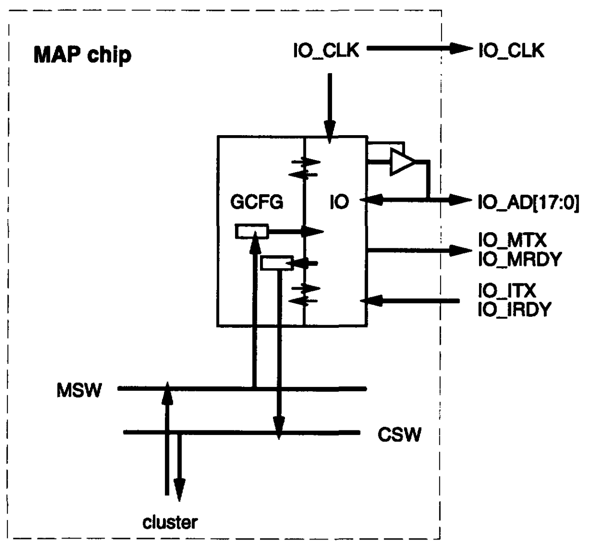


Figure 2-3: I/O Port interfaces. The I/O Port is accessed through memory operations which map onto global config space, part of which is dedicated to I/O. GCFG/IO Memory operations are issued in the cluster, sent through the MSW, and are received in the GCFG. Any results, as for loads, are returned across the C–Switch and written back into the register files or cc registers.

2.4 Software model

To minimize the impact on the memory-instruction pipeline, the I/O reuses existing instructions but in some cases uses the fields differently. There are seven MAP instructions instructions recognized by the I/O port; these are: LD, ST/FST, STSU/FSTSU, and STSCND/FSTSCND. LD, ST, and FST are the normal single word load and store operations. FST is the floating-point register version of ST. STSU/FSTSU are similar to ST/FST except that, in addition, they return a CC (condition code) value.

This is provided for software flow-control. The number of packets sent for the address and data words are determined by the upper bits of the address accessed by the instruction as will be explained in subsection 2.4.2. A special encoding indicates that no address packets are to be sent; only data packets are to be sent or received. This is the mechanism to use to send or receive each data word of a burst transfer. STSCND/FSTSCND initiate burst transactions. The postcondition bit in this case indicates whether it is a load or a store. The source data indicates the size of the transaction in words. These also return a CC value for software flow-control. The commands are summarized in table 2.1.

`memu` [cond cr]	`ld`	src1, [src2,] dest
`memu` [cond cr]	`st`	src1, [src2,] dest
`memu` [cond cr]	`fst`	src1, [src2,] dest
`memu` [cond cr]	`stsu`	pre, post, src, dest, ccdest
`memu` [cond cr]	`fstsu`	pre, post, src, dest, ccdest
`memu` [cond cr]	`stscnd`	pre, post, src, dest, ccdest
`memu` [cond cr]	`fstcnd`	pre, post, src, dest, ccdest

Table 2.1: I/O memory instruction set architecture. Src/src1 is the register that holds the source of the operation. For loads this is a pointer, for stores this is data. Dest is the target register for loads and the target pointer for stores. Src2 holds the postincrement field as described in the MAP ISA reference.[1]. The pre fields are always ignored. The post field only has meaning for the stscnd/fstscnd instructions, indicating the direction of the burst transfer. CCdest is the CC register to which is written the flow-control bit.

2.4.1 Software Flow control

As mentioned above, flow control is built into the software model to facilitate smooth operation without tying up resources. Flow control for loads come for free. Any operation on the loaded register will freeze the thread until the load completes. Flow control for stores are done using the STSU/FSTSU instructions. Since the destination CC register is marked invalid on issue, subsequent store operations can be conditionally executed on ccreg true, and will stall waiting for the CC register to be written back. The CC value is returned when the I/O port is ready to receive a new packet from the cluster. If not for the software flow-control, too many packets would be sent to the I/O port controller at once. This would fill the I/O port controller pipelines and then the GCFG C–Switch input pipeline. At that point, the GCFG is blocked from receiving either GCFG or I/O requests, though no requests are lost because of the hardware flow control.

2.4.2 I/O Address space

To access the I/O port, memory instructions need to use special pointers that point into the I/O address space. The structure of an I/O pointer is shown in figure 2-4. The memory system sees bits [53:3]. Of these, the top 5 bits are reserved by the GCFG. A special encoding indicates a pointer destined for the I/O subsystem. Of the remaining bits, 2 bits encode the address length in packets, and 2 bits encode the data or burst count length in packets; this allows more efficient utilization of the I/O bus bandwidth. This leaves a total of 42 address bits. This can still easily map entire drives. An address length of 11_2 indicates that no address packets are to be sent.

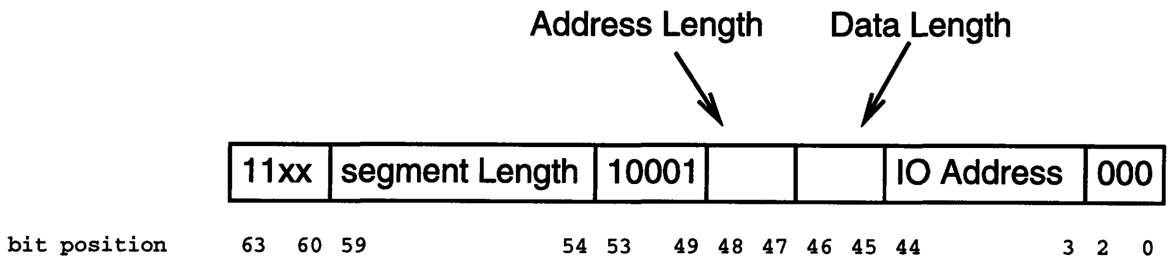


Figure 2-4: I/O Pointer structure.

2.4.3 Extensions

Note that by using the no-address-packet addresses, the high-level communication protocol can be bypassed because arbitrary words can be transmitted or received through the I/O port. This is a side benefit of the design. I do not expect the I/O port to be used in this manner because the existing protocol ought to be general enough. However, the capability exists to create a software protocol layer completely replacing the high-level protocol layer.

Chapter 3

Interfaces and Timing

This chapter details all the I/O port controller interfaces and discusses the timing of the major signals.

3.1 Interfaces

The I/O Port controller communicates with two separate modules. The I/O port controller talks with the off-chip module through the I/O bus and the protocols described in section 2.2. It also talks to the M–Switch and C–Switch through the arbitration of the GCFG. For this it uses a pair of data interfaces and a pair of handshake interfaces.

3.1.1 Off-chip interface

A total of 23 signals are used to communicate between the I/O Port and the off-chip I/O module. An additional 19 signals are are used to communicate between the I/O port and the pads. These are summarized in table 3.1. These signals are used in the implementation of the low-level protocol described in section 2.2.2.

signal	width	direction	function
IO_CLK	1	input	a 50 MHz clock obtained by dividing the system clock by 2. This becomes the clock for the off-chip I/O module
IO_MTX	1	output	MAP transmit. MAP chip has data to transmit on the next I/O clock cycle.
IO_MRDY	1	output	MAP ready. MAP chip is ready to receive data on the next I/O clock cycle.
IO_ITX	1	input	I/O chip has data to transmit on the next I/O clock cycle.
IO_IRDY	1	input	IO chip is ready to receive data on the next I/O clock cycle.
IO_AD	18	bidir	a bi-directional bus between the MAP and the off-chip module.
io_data_out	18	output	This is the data input to the IO_AD drivers. Only when oe is high is this value driven onto the bus.
oe	1	output	output enable signal controlling the direction of the IO_AD bus. This signal goes to the IO_AD pads, but not off-chip.

Table 3.1: I/O Port off-chip interfaces

3.1.2 GCFG/MSW data interface

The GCFG/MSW data interface is shown in table 3.2. These inputs come from the MSW and are p-latched in the GCFG. The GCFG holds this data until the I/O controller is ready to receive them. This p-latch is shared between the GCFG and the I/O and can be considered the first pipeline stage into both pipelines. When this input latch is full and cannot advance in the pipeline for some reason, the MSW `ready` line goes low and stays low until the pipeline advances. This blocks anybody from issuing any MSW request to the GCFG, preventing buffer overflow.

3.1.3 GCFG/MSW handshake interface

The GCFG/MSW handshake interface is shown in table 3.4 These signals form the handshake between the GCFG and I/O controller to pass MSW packets to the IO pipeline. As soon as the I/O controller indicates that it has received that data (by signalling `mswack_v2`), the GCFG can load new data into the MSW input p-latch.

signal	width	function
in_op	6	specifies the memory operation to be performed. see table 3.3 [4]
in_post	1	postcondition bit. This is normally used by memory system to set the synchronization bit after performing a synchronizing memory operation. Used by the I/O to determine the direction of burst transfers.
in_rtnclst	2	The cluster to which the result is to be returned
in_tslot	3	The V–Thread id of the thread to which the result is to be returned
in_rf	1	The register file bit. 1 indicates floating point. 0 indicates integer.
in_reg	4	The id of the register to which the data result is to be returned.
in_rtncc	4	The id of the CC register to which the CC result is to be returned.
in_addr	51	The source or target address of the memory instruction.
in_sync	1	The synchronization bit of the source data.
in_data	65	The pointer bit and source data.

Table 3.2: GCFG/MSW data interface

operation	op type
ST	100000
FST	100000
STSU	101100
FSTSU	101100
STSCND	101010
FSTSCND	101010
LD	110000

Table 3.3: List of instructions recognized by the I/O and their corresponding type identifiers.

signal	width	direction	function
iodav_v2	1	input	Data is available to be loaded on the rising edge of system clock (clk). Needs to be valid very early in the cycle.
mswack_v2	1	output	Data will be loaded on the rising edge of system clk. This signal is independent of iodav_v2 and is a don't care when iodav_v2 is low.

Table 3.4: GCFG/MSW handshake interface

3.1.4 GCFG/CSW data interface

The GCFG/CSW data interface is shown in table 3.5. These outputs are n-latched in the GCFG. The GCFG holds this data until the cluster is ready to receive it and the CSW resources are available to send it. This n-latch is shared between the GCFG and the I/O and can be considered the last pipeline stage of both pipelines. Handshaking prevents this buffer from overflowing. In the event both pipelines want to use the n-latch simultaneously, the I/O pipeline gets priority.

signal	width	function
`out_rtnclst`	2	The cluster to which the result is to be returned
`out_mb`	1	Memory barrier flag. When asserted indicates that the receiver should decrement its memory barrier counter as a side result of receiving this transfer. Always 0 from the I/O
`out_xfrtype`	3	CSW transfer type. This field indicate the type of transfer in progress. The meaning of the transfer type code varies depending which slot is addressed, as shown in table 3.6 [3]
`out_sclst`	2	sender cluster ID. Identifies the cluster that generated the transfer. Always 0 from the I/O
`out_sslot`	3	sender slot ID. Identifies the slot that generated the transfer. Always 0 from the I/O
`out_tslot`	3	The V–Thread id of the thread to which the result is to be returned
`out_rf`	1	The register file bit. 1 indicates floating point. 0 indicates integer.
`out_reg`	4	The id of the register to which the data result is to be returned.
`out_cc`	1	The CC result to be written to the condition register.
`out_rtncc`	4	The id of the CC register to which the CC result is to be returned.
`out_sync`	1	The synchronization bit of result word
`out_data`	65	The pointer bit and result data.

Table 3.5: GCFG/CSW data interface

3.1.5 GCFG/CSW handshake interface

The interface is shown in table 3.7. These signals form the handshake between the GCFG and IO to pass CSW packets from the IO pipeline.

XfrType [2:0]	Meaning	
	dstSlot 0-5 (Threads)	dstSlot 6 (IFU-Queue)
000	**abort transfer**	
001	data only transfer	Message Enqueue (unordered) / Event Enqueue
010	cc only transfer	Message Enqueue (ordered)
011	data and cc	Netout State Read/Write
100	**reserved**	GTLB Read/Probe
101	**reserved**	GTLB Write
110	**reserved**	IFU: Normal Return
111	*Command in data field*	IFU: Cancelled Return

Table 3.6: Table of CSW transfer types

signal	width	direction	function
cswdav_2	1	output	Data is available at the beginning of the current system clk cycle.
cswack_2	1	input	Data will be loaded sometime in the current system clk cycle. Must only be high when **cswdav_2** is high. Must be valid by about the middle of the system clk cycle.

Table 3.7: GCFG/CSW handshake interface

3.2 Timing

Figure 3-1 shows the timing of the I/O Port. The IO_MTX and IO_MRDY signals originate in the system clock (clk) domain and are stretched another half cycle by n-latching with clk. The outputs are then stretched another half IO_CLK cycle by n-latching with IO_CLK. The point of the first n-latch is to guarantee that the hold time of the following n-latch is met, even when IO_CLK may be significantly skewed. This scheme tolerates up to 5 ns of skew between the system clock and the IO_CLK as seen by the second n-latch. The point of the second n-latching is to more-or-less center the rising edge of IO_CLK around IO_MTX and IO_MRDY and to provide a generous skew margin between the IO_CLK and IO_MTX/IO_MRDY. As will be shown later, for IO_MTX, this gives a skew margin of about 3 ns before the setup time of IO_CLK and 10 ns after the hold time of IO_CLK. The latter margin is especially important as IO_CLK becomes skewed in the clock distribution of the off-chip slave processor.

The output enable signal (oe) is n-latched with clk to create a system clock cycle long pulse more or less centered around the rising edge of IO_CLK. This means that the IO_AD bus will be driven approximately starting 5 ns before IO_CLK rises, and stopping 5 ns after IO_CLK rises. This protects from bus conflicts and provides a generous skew margin around IO_CLK in either direction. In addition, even when the tri-state drivers are turned off, the capacitance of the transmission line maintains the voltage until it is driven to a new value. This gives about ten more nanoseconds to latch the data off the bus if IO_CLK is very skewed.

On the I/O chip side, all outputs are to be valid well before the rising edge of IO_CLK. The IO_AD bus is to be driven on the second half of the IO_CLK cycle. The reason the IO_AD bus is only driven for half an IO_CLK cycle is to prevent bus conflicts, even when there is significant IO_CLK skew.

All inputs (on either side) are to be sampled on the rising edge of IO_CLK. The reason IO_CLK is used to sample data coming onto the MAP chip is that it is skewed from the system clock. This helps relax some of the time constraints in transmitting data from the off-chip slave to the MAP chip. IO_CLK is guaranteed to go high before

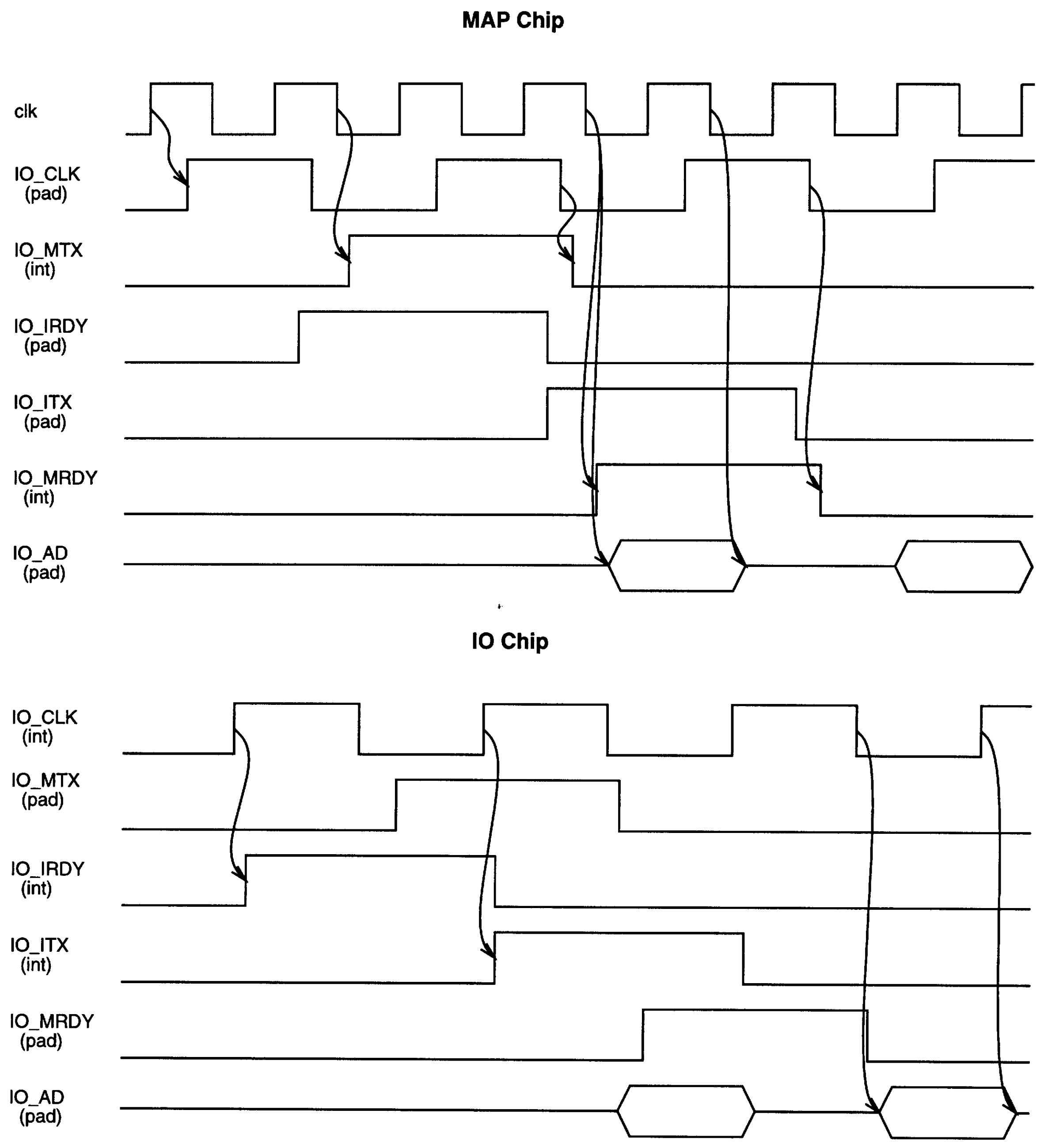


Figure 3-1: I/O Port protocol and timing. The first transaction is sent by the MAP chip; the second transaction is sent by the I/O chip.

the data from the off-chip slave goes invalid, so there are no hold time problems, though setup time can still be violated. The alternative would be to use the falling edge of the system `clk` to sample the data. This has the advantage of relaxing some time constraints but is not guaranteed to work if we stretch out the system clock.

Chapter 4

Logic and RTL Design

This chapter discusses the design and implementation of the I/O port controller in Verilog HDL.

4.1 Overview

Physically, the I/O port controller is a submodule of the GCFG. There are a total of 11 modules in the design. The majority of the modules form three major pipelines in the design. They will be called the output, the input, and the return pipelines. Each pipeline runs independently from the other pipelines. The output pipeline holds data and state required to transmit data off-chip. The input pipeline holds data and state required to receive data from off-chip. The return pipeline holds information required to generate CSW replies. It is loosely coupled with the input and output pipelines through a pair of handshakes. In addition to the modules that form the three major pipelines, there are three modules that do not go into any pipeline. These non-pipeline modules help control and coordinate the three pipelines. A top level diagram showing the interconnections is shown in figure 4-1.

4.2 Non-pipeline modules

The non-pipeline modules are IO_DEC, SYNCFSM, and IO_PORT_CTL.

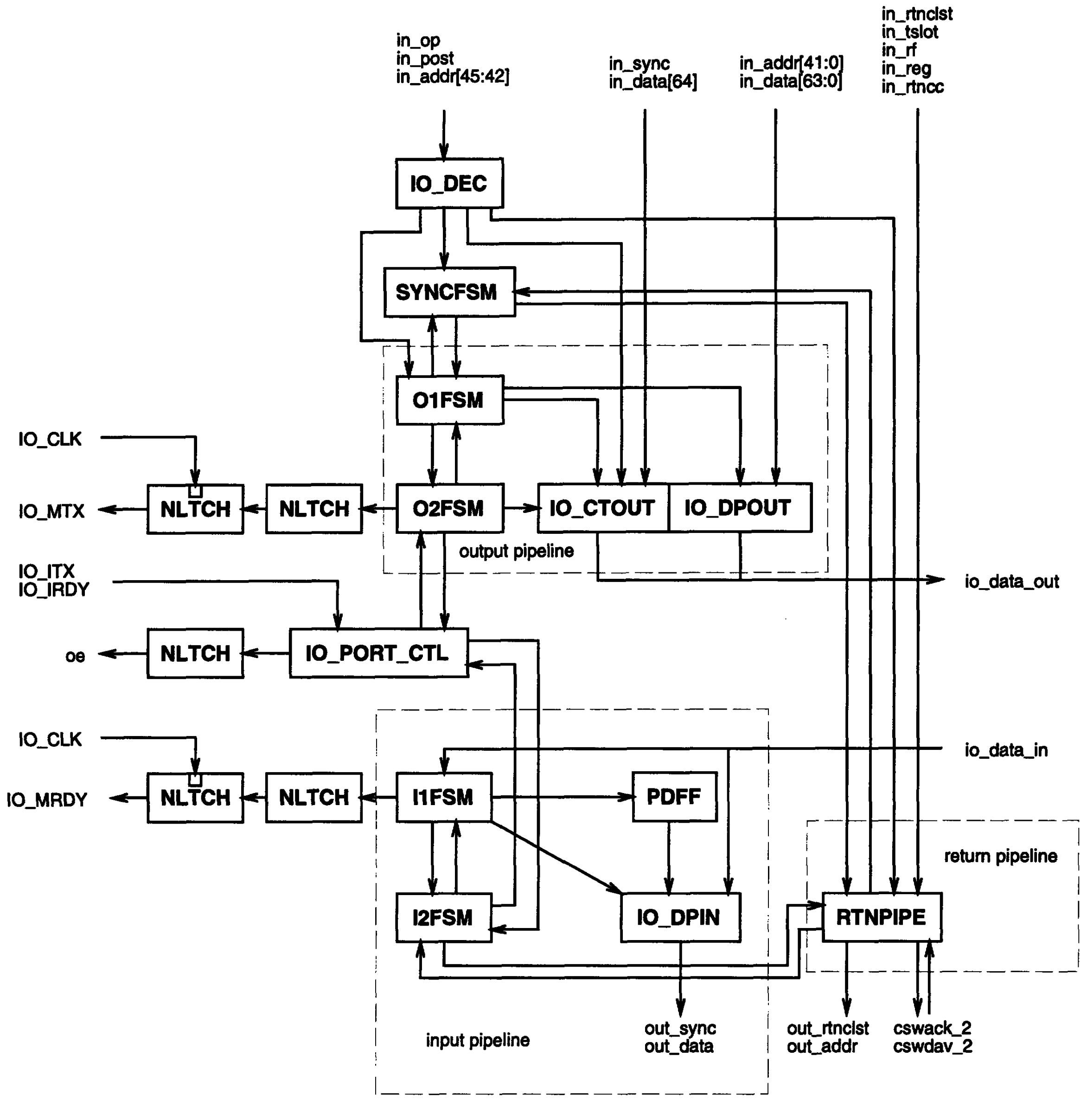


Figure 4-1: Structure of the IO Port. Except where noted, all latches and flip-flops are clocked by system clk

4.2.1 IO_DEC

This module is purely combinational. It decodes the MSW request passed to it by the GCFG and creates nine control signals which are used by the the input and control pipelines.

hasaddr This signal indicates whether this I/O instruction will generate an address word. Every instruction generates an address word unless its I/O address indicates a length of 11_2. These are the data words of burst transfers, which are not accompanied by address words.

hasdata This signal indicates whether this I/O instruction will generate a data or burst count word. Every store and burst-initiate instruction does this.

ls This signal indicates whether the transaction type is load or store. Recall that on burst-initiate instructions, the postcondition holds this information. Otherwise, the instruction type holds this information.

hasctl This signal indicates that the instruction will use the return pipeline. Everything uses the return pipeline except non-flow-controlled stores.

hasad This signal indicates that the instruction will use the output pipeline. This is the logical OR of hasaddr and hasdata.

addrlen This two bit signal indicates the length of the address word to transmit.

datalen This two bit signal indicates the length of the data word to transmit. Recall that this forms part of the I/O address.

bypass This signal indicates whether the instruction uses the special bypass path in the return pipeline. This will be discussed later in the description of the return pipeline in section 4.4. The STSCND and STSU instructions use this.

burst This signal indicates that the instruction is a burst initiate.

4.2.2 SYNCFSM

This module synchronizes the output and return pipelines. It ensures that the each of the pipelines read their respective portions of the MSW request exactly once. A new MSW request cannot be received unless all the pipelines have read any data destined to them. This is implemented as a 3-state FSM (finite-state machine) as shown in figure 4-2. The FSM keeps track of which pipeline has read their data. State 00 indicates that neither pipeline has read the current request. State 01 indicates that only the return pipeline has read the request and the output pipeline needs to read the request. State 10 indicates that only the output pipeline has read the request and the return pipeline needs to read the request. Recall that some instructions may use only one of the two pipelines.

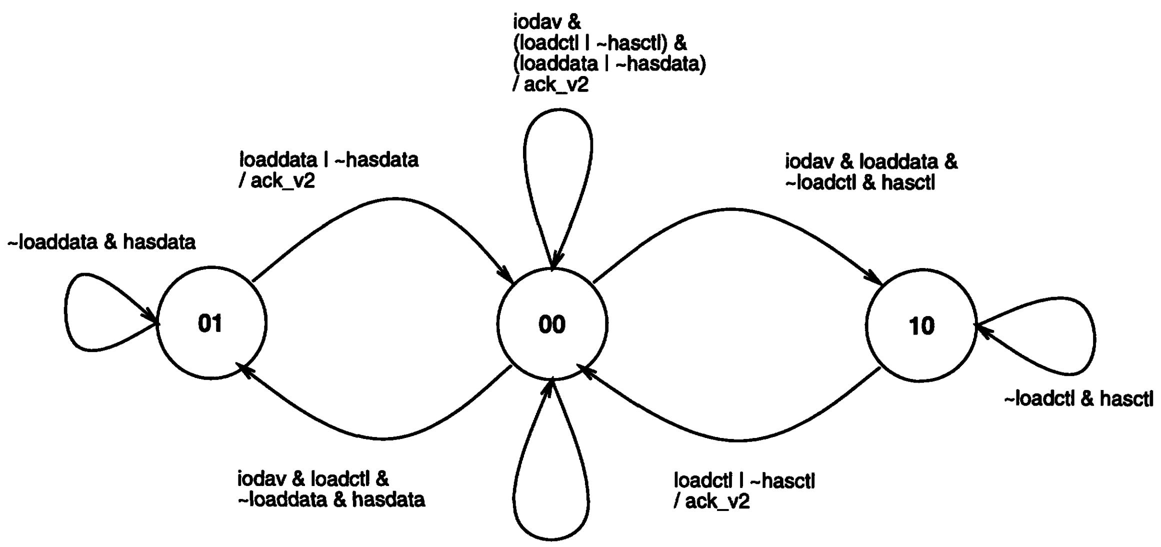


Figure 4-2: State diagram of the SYNCFSM module. `rst` is omitted for clarity. On `rst` it goes to state 00. Transitions occur on the rising edge of system `clk`.

SYNCFSM communicates with each of the GCFG, the output pipeline, and the return pipeline through handshake signals, one pair for each.

iodav_v2 This input from the GCFG indicates that there is data available in the GCFG MSW latch that needs to be read.

mswack_v2 This output to the GCFG indicates that the I/O port controller has read the data in the latch. Upon getting the ack, the GCFG either loads a new instruction into the latch or deasserts `iodav_v2`. `Mswack_2` is a don't-care when `iodav_v2` is low.

data_dav This output to the output pipeline indicates that there is data available in the GCFG MSW latch that needs to be read by the output pipeline.

loadol This input from the output pipeline indicates that it has read the data from the latch. This signal can only be high when `data_dav` is high.

ctl_dav This output to the return pipeline indicates that there is data available in the GCFG MSW latch that needs to be read by the return pipeline.

loadctl This input from the return pipeline indicates that it has read the data from the latch. This signal can only be high when `ctl_dav` is high.

4.2.3 IO_PORT_CTL

This module is the FSM that manages the I/O port bus. It directly implements the low-level protocol described in section 2.2.2. The FSM is shown in figure 4-3. The FSM keeps track of which side transmitted last. State 0 indicates that the off-chip module was the last to transmit and that the MAP will get priority. State 1 indicates that the MAP was the last to transmit and that the off-chip module will get priority.

4.3 Output pipeline

The modules of the output pipeline are organized first into two classes. The first class holds only the actual pipeline registers and muxes. This class is further divided among two modules, IO_CTOUT and IO_DPOUT. A logic diagram is shown in figure 4-4. These modules are designed in a datapath style, with very regular patterns. There is no control logic to mess up the regularity. This way, if it was desired to do so, these modules can be placed and routed manually with minimal effort for greater area efficiency and speed. The second class holds all the control logic. There are two modules, one for each pipeline stage.

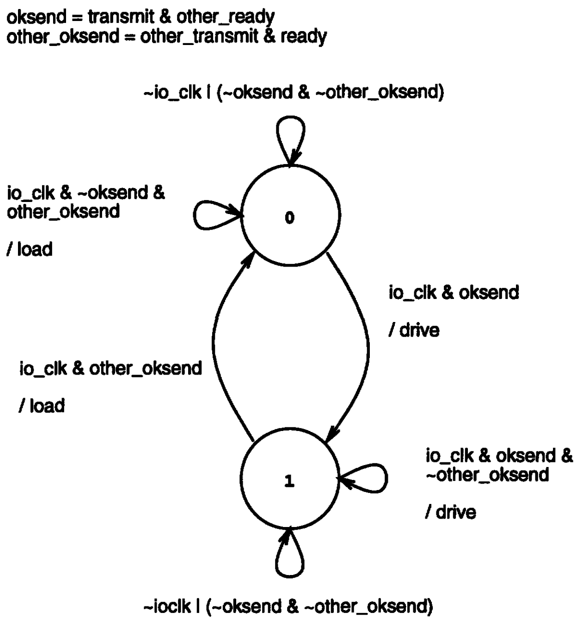

oksend = transmit & other_ready
other_oksend = other_transmit & ready
~io_clk | (~oksend & ~other_oksend)
io_clk & ~oksend & other_oksend
/ load
0
io_clk & oksend
/ drive
io_clk & other_oksend
/ load
1
io_clk & oksend & ~other_oksend
/ drive
~ioclk | (~oksend & ~other_oksend)

Figure 4-3: State diagram of the IO_PORT_CTL module. `rst` is omitted for clarity. On `rst` it goes to state 0. Transitions occur on the rising edge of the system `clk`.

4.3.1 IO_CTOUT

This module contains pipeline registers required to generate the top 2 bits of the data output, which are multiplexed between length and sync/ptr bits. Each register stage either loads new data or holds old data depending on the control signals from O1FSM and O2FSM respectively.

4.3.2 IO_DPOUT

This module holds pipeline registers required to generate the bottom 16 bits of the data output, which are multiplexed between the different packets of the address and data using a pair of shift registers and muxes. There are two pipeline stages. The second stage may require up to 14 cycles to finish transmitting the data. The first stage serves as a buffer to keep the second stage always full without needing to stall the entire GCFG for long periods of time. This allows full IO bandwidth utilization going off-chip. The first stage is controlled by O1FSM through the `loado1` signal. The second stage is controlled by O2FSM through `loaddo2, holddo2, shiftdo2,`

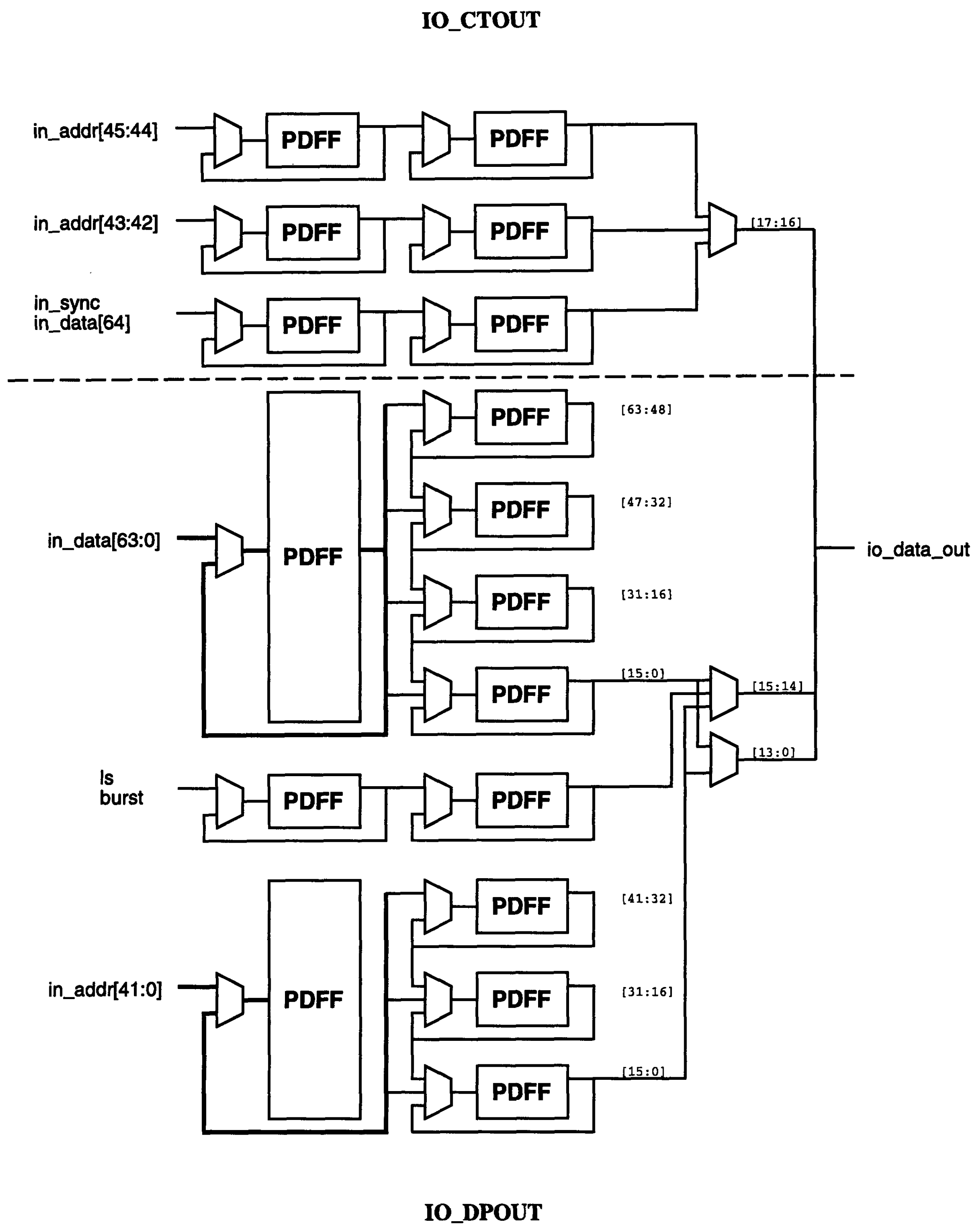


Figure 4-4: Logic diagram of the IO_CTOUT and IO_DPOUT modules. All flip-flops are clocked on the rising edge of the system clk.

`loadao2, holdao2, and shiftao2`. The first three signals control the top set of shift registers, which hold the data and burst count words. The last three signals control the bottom set of shift registers, which hold the address words.

4.3.3 O1FSM

O1FSM is a simple FSM that manages the first stage of the output pipeline. It keeps track of whether the stage is empty and whether it needs to be loaded. It communicates with SYNCFSM and with O2FSM using a pair of handshakes. The state diagram is shown in figure 4-5.

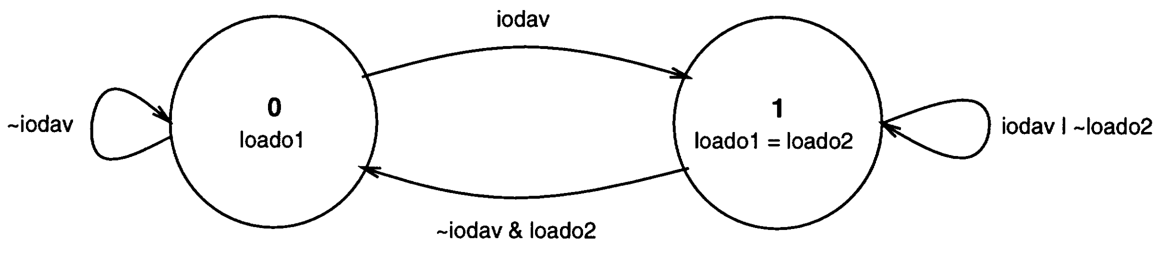


Figure 4-5: State diagram of the O1FSM module. `rst` is omitted for clarity. On `rst` it goes to state 0. Transitions occur on the rising edge of system `clk`.

4.3.4 O2FSM

O2FSM is the FSM that manages the second stage of the output pipeline. It keeps track of whether the stage is empty and whether it needs to be loaded or shifted. It communicates with O1FSM and with IO_PORT_CTL with a pair of handshakes.

o1dav This input signal from O1FSM indicates that stage 1 of the output pipeline has data.

loado2 This output signal to O1FSM indicates that stage 2 is going to load data at the end of the current cycle. This signal indicates to O1FSM to load new data or deassert `o1dav`. This signal is independent of `o1dav`.

transmit_2 This output signal to IO_PORT_CTL indicates that in two cycles it will have a data packet to transmit. This is not always known in advance. When it is unsure, `transmit_2` is low to prevent underflow.

drive This input signal from IO_PORT_CTL indicates that on the current cycle and on the next cycle, data is being driven on the bus. Data is to be stable during these two cycles. A new data packet should be loaded at the end of the next cycle.

To do this, O2FSM keeps track of two pieces of state. The first is the number of address packets it still needs to transmit. The second is the number of data (or count) packets it still needs to transmit. In the chosen encoding, three bits are used to represent each. The address state is a three-bit shift register with four legal states. This encoding wastes a register but saves on some logic and is faster compared to the compact 2-bit encoding. The data state is a two-bit decrementing counter plus a valid bit. Transmit is asserted when there will be valid data in either the data pipeline registers or the address pipeline registers. When there is data in both, the address is sent first. This occurs for store and burst initiate instructions.

4.4 Return pipeline

The entire return pipe, including both pipeline registers and control is implemented in the RTNPIPE module. Its basic logic is shown in figure 4-6. There are five pipeline stages; this is to keep track of 2 transactions going out, 2 transactions coming in, and 1 transaction in flight off-chip. The pipeline has the capability to squash pipeline bubbles. This is required since transactions do not generally arrive on each cycle and the CSW may be blocked. This is implemented in the load chain on the left side of figure 4-6. Each stage loads if the next stage loads or the current stage is empty.

There is a normal path (for loads) and a bypass path (for flow-controlled stores); non-flow-controlled stores and burst initiators do not use this pipeline at all. If there is something in the bypass path, then it is ready to advance when the first stage of the output pipeline will be free on the next cycle. This guarantees that if another I/O instruction is issued, there is space in the pipeline for it without blocking the GCFG. When a transaction can go to the CSW from either path, the normal path has priority. This shortens a very long combinational path in the design. It is much faster to tell if the normal path is ready to issue a CSW request since it is just an AND of two register outputs. The bypass path logic is much more complex. If the priority was reversed, `outnorm` would have to wait until `readybyp` was resolved, even though the `cswack_2` could come earlier. `Outnorm` feeds the load chain in RTNPIPE as well as the `IO_MRDY` signal that goes off-chip. Both paths are extremely long and tight on timing.

RTNPIPE also synchronizes with the input pipeline so that data is presented to the GCFG when both pipelines are ready. RTNPIPE communicates with SYNCFSM, the GCFG, and I2FSM through three pairs of handshakes. The SYNCFSM interface is described in section 4.2.2. The GCFG interface is described in section 3.1.5. The I2FSM interface is shown below.

i2dav This input from I2FSM indicates that there is data available in that stage.

outnorm This output to I2FSM indicates that the GCFG is going to latch the data from I2FSM at the end of the current cycle. This indicates to I2FSM that it should load new data or deassert `i2dav`. Outnorm is only high when `i2dav` is high.

4.5 Input pipeline

The input pipeline is organized much like the output pipeline. There is a datapath module and then a control module for each pipeline stage.

4.5.1 IO_DPIN

IO_DPIN holds the pipeline registers. This holds state required to generate the data portion of the CSW return packet. The four sets of Positive-edge-triggered D-Flip-Flops (PDFF's) connected to `io_data_in` load in sequence to latch in the different packets as they arrive. In addition, they also have the ability to be zeroed at the proper times to zero-pad data that is less than 4 packets wide. A logic diagram

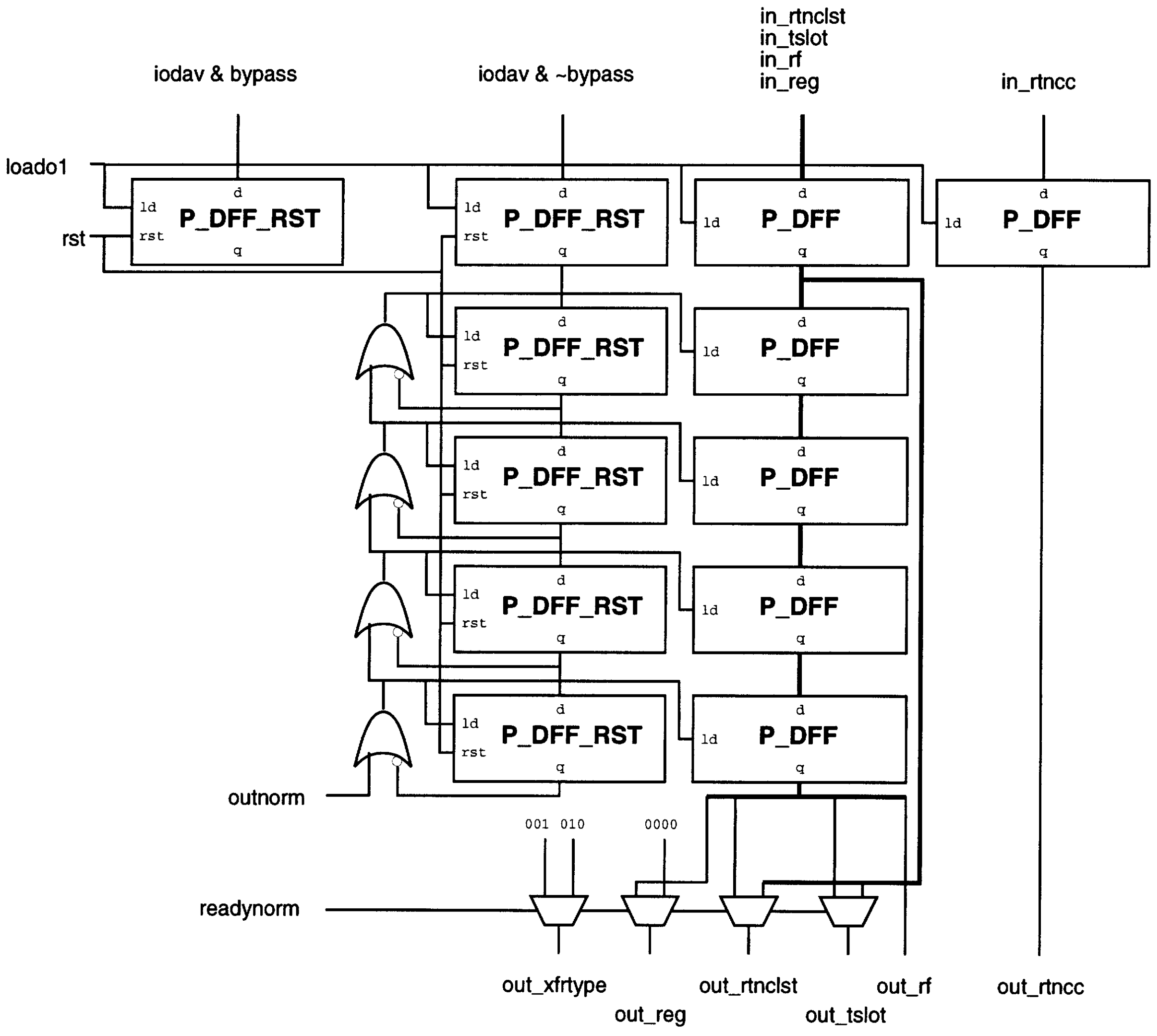


Figure 4-6: Logic diagram of the RTNPIPE module. All flip-flops are clocked on the rising edge of the system clk

is shown in figure 4-7. There are two pipeline stages; this allows the MAP chip to continue receiving data even while the CSW is blocked for several cycles. The first stage of IO_DPIN is controlled by the signals H0-H3 and L0-L3. H0 through H3 indicate that the respective register is to hold its data. L0 through L3 indicate that the respective register is to load new data. For any register, if neither is asserted, then the register is zeroed. It is illegal for both to be asserted simultaneously. The second stage is controlled with the loadi2 signal.

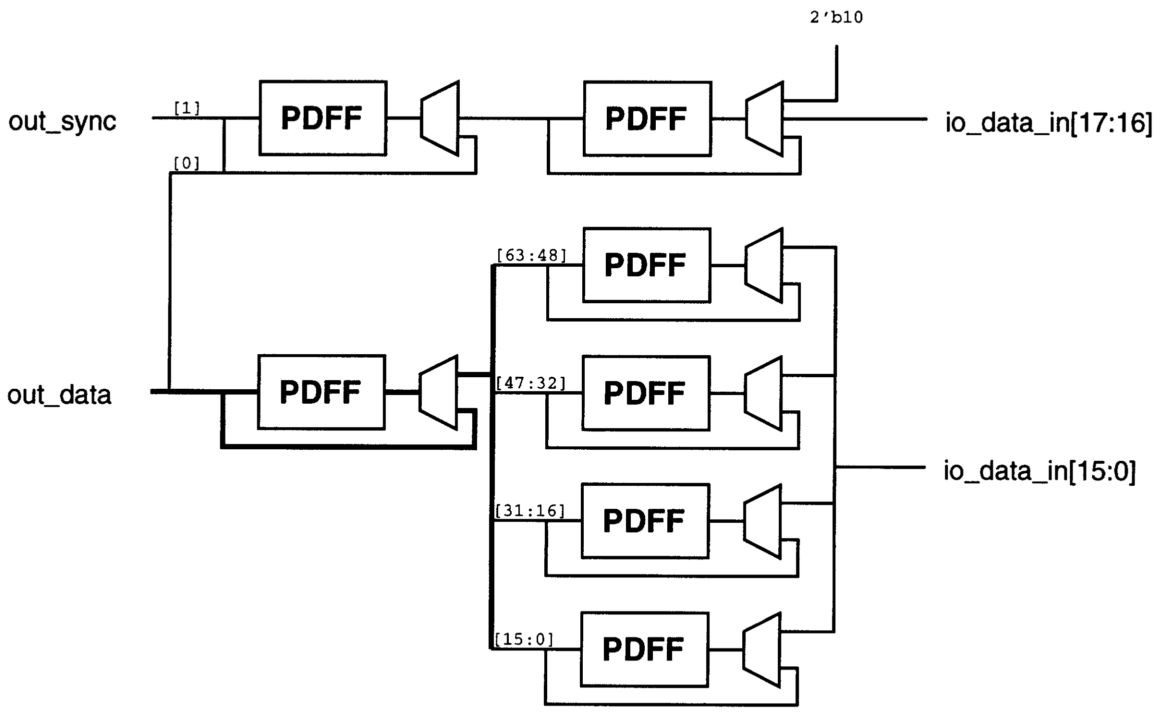


Figure 4-7: Logic diagram of the IO_DPIN module. All flip-flops are clocked on the rising edge of system clk.

4.5.2 I1FSM

I1FSM is the FSM that manages the first stage of the input pipeline. It keeps track of whether the stage is empty or full; it communicates with I2FSM and with IO_PORT_CTL with a pair of handshakes.

i1dav This output to I2FSM indicates that this stage is full.

loadi2 This input from I2FSM indicates that stage two is loading the data from stage one. This tells I1FSM to deassert i1dav. `Loadi2` is independent of `i1dav`.

ready_2 This output to IO_PORT_CTL indicates that the stage will be empty in four cycles. This is not always known in advance. When uncertain, the `ready_2` is low to prevent overflow.

loadi1 This input from IO_PORT_CTL indicates that the stage should be loaded at the end of the next cycle.

I1FSM is implemented as a 3-bit 5-state FSM as shown in figure 4-8. Based on the states, the load and hold signals are generated. The sequencing of these signals is shown in figure 4-9.

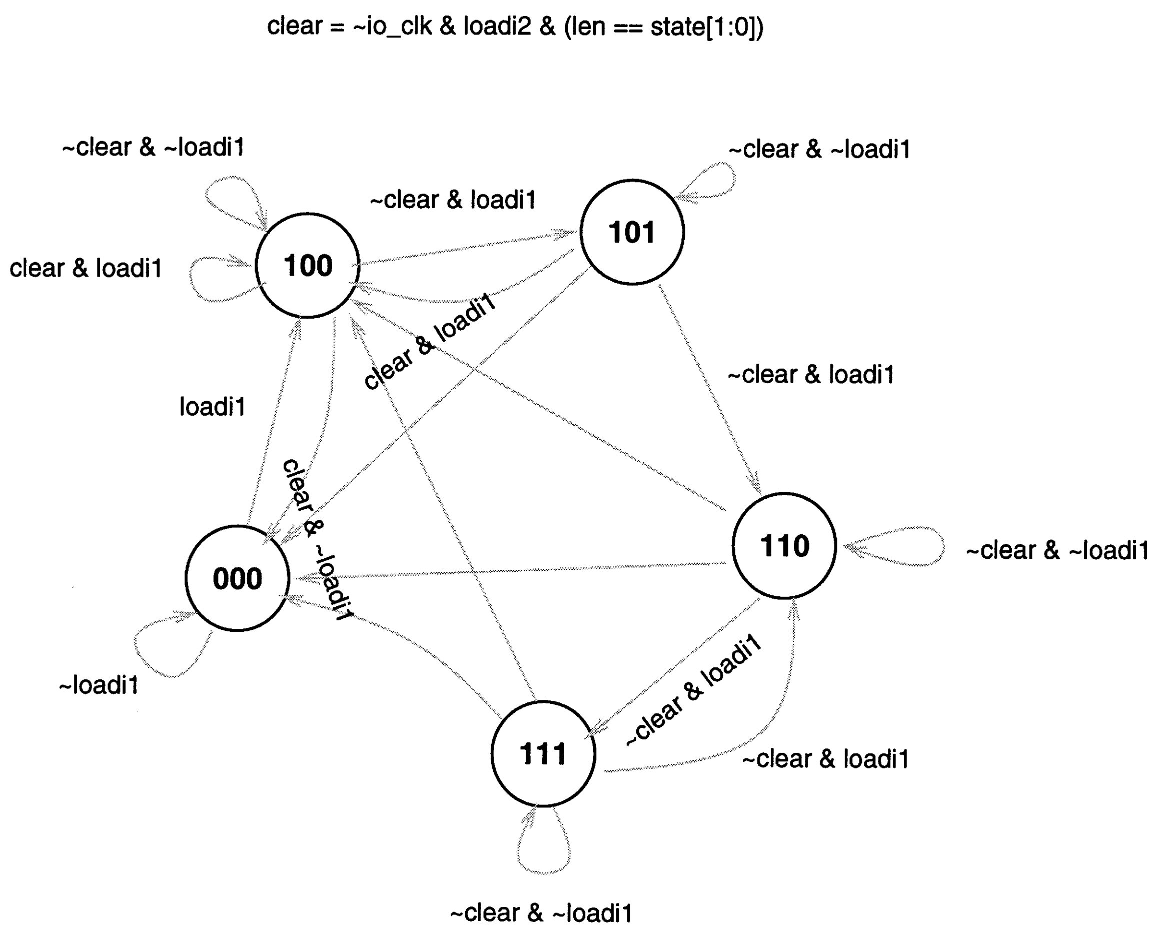


Figure 4-8: State diagram of the I1FSM module. `rst` is omitted for clarity. On `rst` it goes to state 0. Transitions occur on the rising edge of system `clk`.

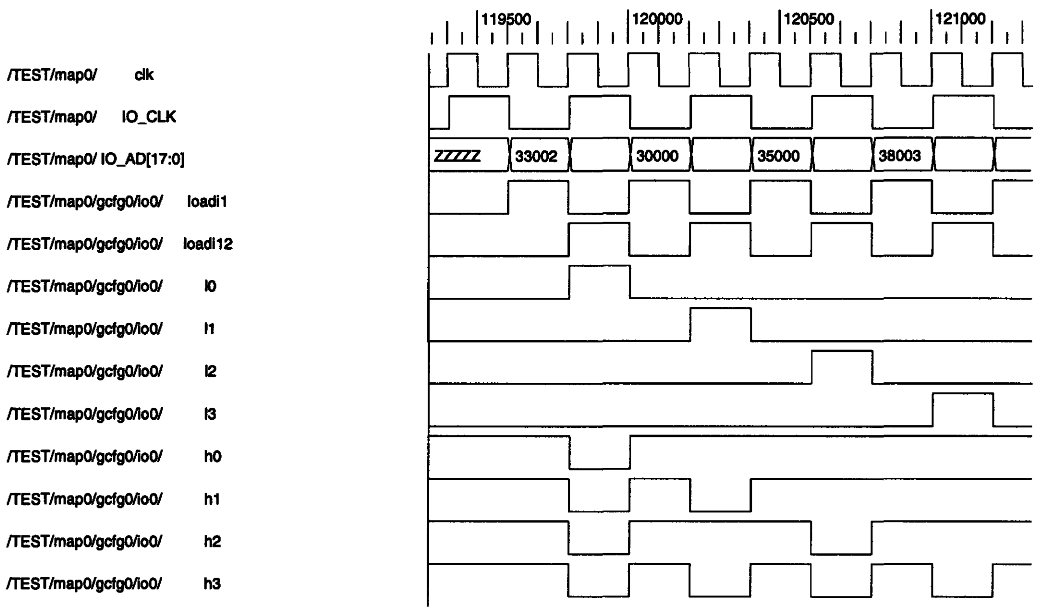


Figure 4-9: Timing diagram of I1FSM select lines

4.5.3 I2FSM

I2FSM is a simple FSM that manages the second stage of the input pipeline. It keeps track of whether the stage is empty and whether it needs to be loaded. The state diagram is shown in figure 4-10. It communicates with I1FSM and with RTNPIPE with a pair of handshakes.

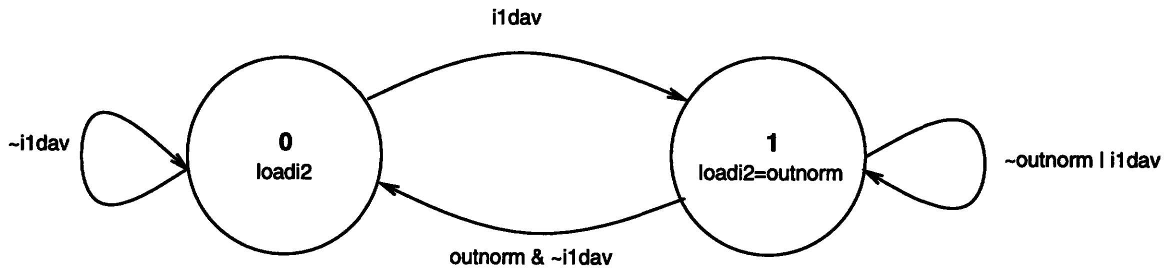


Figure 4-10: State diagram of the I2FSM module. `rst` is omitted for clarity. On `rst` it goes to state 0. Transitions occur on the rising edge of system `clk`.

Chapter 5

Verification

There are three aspects to verification: source-level (RTL) verification, timing verification, and electrical rule verification.

5.1 RTL

To help test the I/O port controller, a Verilog model of the off-chip slave I/O processor was written. The off-chip model knows the communication protocols and responds to requests from the MAP chip as would the real thing. This allows us to test all the pipelines in the controller and to ensure that they are working correctly in concert. For load requests, the data returned is formed by combining the source address with the value of an internal counter. This ensures that all the data returned are unique. The synchronization and pointer bits are also taken from the internal counter so that all combinations of those can be exercised. The length of the result word in packets is determined by two bits in the address. This allows the the variable length word features to be tested. In addition, the off-chip model logs all requests it receives.

The controller code was tested first only with the off-chip model. We applied test input patterns that tested all the different transactions that exist. This helped flush out bugs in both the controller and the off-chip model.

After the isolation testing, we integrated the RTL model of the I/O port controller with the RTL model of the rest of the MAP chip. The MAP chip model was already set up to log all register writes. This allows us to see what is coming back off the I/O port without looking at dumps. Two test programs in assembly were written to exhaustively test the I/O controller. The first test exercises all the non-burst transactions. The second test exercises all the burst transactions.

The non-burst test (see appendix A.1) first issues a series of loads. There are a total of twelve loads, each testing a different combination of source address word length (one to three) and result data word length (one to four). This is an exhaustive testing of the load control and datapaths. Next, there are twelve non-flow-controlled stores, each testing a different combination of target address word length and source data word length. This is an exhaustive testing of the store (non-flow-controlled) control and datapaths. Finally, there are twelve flow-controlled stores, similar to the non-flow-controlled tests. Again, this is an exhaustive test.

The burst test (see appendix A.2) first issues a series of burst loads. There are a total of four loads, each testing a different combination of source address word length, data word length, and result data word length. All four result data word lengths are used. Next, there are twelve non-flow-controlled stores, each testing a different combination of target address word length and source data word length. All four source data word lengths are used. Finally, there are twelve flow-controlled stores, similar to the non-flow-controlled tests. Together, these tests exercise all combinations of address length and count length on burst initiators, and all data lengths for burst loads and stores. This makes a exhaustive test, testing all the control and datapaths related to burst mode transfers.

The results of the tests proved the I/O bus could be saturated from software, showing that end-to-end communication could occur at the full 100MB/s. For one packet and two packet words, a word can be received on the MAP every 5.7 cycles, for bandwidths of 35 MB/s and 70 MB/s respectively. With small words, the software loop becomes the limiting factor in I/O performance. This figure can be improved by software pipelining four requests instead of the current two requests. This is expected to improve the one packet performance to a word every 3 cycles, for a bandwidth of 66 MB/s, and two packet performance to a word every 4 cycles, for a bandwidth of

100 MB/s. For three and four packet words, a word is received every 6 and 8 cycles respectively, which is the theoretical maximum of 100 MB/s. With the larger packets, the I/O bus becomes the limiting factor in I/O performance. There are similar results for stores, with non-flow-controlled stores performing somewhat better, being able to saturate the bus with 2 packet words.

The tests also showed the software and hardware flow control working. For the loads and flow-controlled stores, the issue rate of memory instructions is slowed down enough such that the GCFG is always ready to accept a new request immediately. Thus the hardware flow control at the M–Switch does not get activated. For the non-flow-controlled stores, the issue rate is initially too high for the I/O to handle, thus filling the pipeline. As a result, the GCFG M–Switch buffer gets filled up, thus activating the hardware flow control.

We then modified the MAP chip model to delay C–Switch grants for 16 cycles. This was intended to test the back-pressure paths. Data from the off-chip slave starts to fill up the input pipeline as the first piece of data sits at the head of the pipeline waiting for the C–Switch to be granted. As expected, once the input pipeline filled up, the `IO_MRDY` went low to stop the off-chip slave I/O processor from sending any more data.

As a final test, we synthesized the GCFG and I/O controller and tested that with the rest of the MAP chip using the two tests we wrote. As expected, it behaved identically with the HDL version on a cycle-by-cycle basis.

5.2 Timing

The GCFG/IO were synthesized together as a monolithic block. The lengths of combinational paths were analyzed using the PrimeTime static timing analysis tool. Since this timing was done before the final place and route of the GCFG, PrimeTime cannot properly take into account wire capacitance in doing its timing analysis. By default, PrimeTime assumes zero wire capacitance. To get more realistic results, the capacitances were manually specified on nodes believed to be on the critical path from initial traces. The capacitance was estimated from the fanout of the node. The function used is the same function used during synthesis, which is somewhat rough and arbitrary. Figure 5-1 shows this function. The reason the curve is convex is that the router builds trees to span the connected pins. The marginal cost, in wire length, of adding a new pin decreases since each new pin is probably close to one of the existing pins or to the tree spanning them.

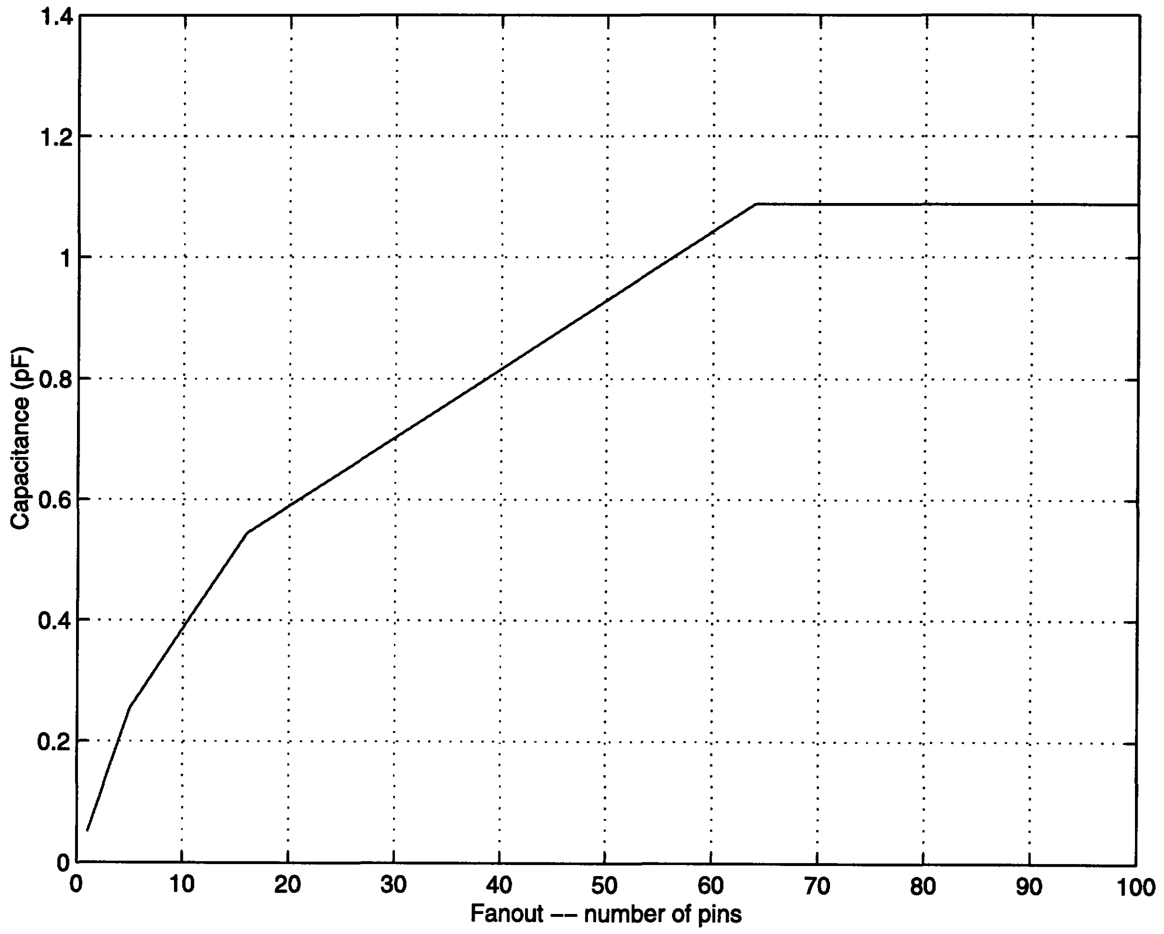


Figure 5-1: Wire capacitance as a function of fanout

According to PrimeTime, the there were some long paths through the I/O which don't make the 100 MHz specification. All these paths have in common the path that starts from clk and goes to `cswack_2`. It was reported to take 8.3 ns. This would be a problem because `cswack_2` feeds many more fairly long combinational paths. Analysis of this path showed that `loado2` was in the critical path and that its logic could easily be moved to the previous cycle. Modifying the logic reduced the time to generate `cswack_2` to 5.343 ns.

The most critical path in the I/O is to the data input of the return pipeline stage 1 state register. This path includes the `load` signal chain in RTNPIPE, of which each stage takes 1.042 ns. Including setup time, the entire path requires 10.44 ns. If it were required to do so, there would be several ways to make up the remaining half nanosecond. The least painful thing to do would be to re-synthesize the GCFG/IO with a timing constraint on that path. This is very likely to work. Another approach would be to accept a little pipeline inefficiency and eliminate one of the stages in the load signal path; the best one would be the `load2` signal, replacing it with `load3`. This reduces the length of the path by 1.042 ns. In addition, one could argue that the .44 ns is in the noise, especially given the guesses that went into the wire capacitance model. Finally, optimizing this path would be useless since there are longer paths in other parts of the chip.

The three important off-chip outputs are the `transmit`, `ready`, and `oe` signals. A timing diagram for these signals is shown in figure 5-2. `Transmit` reaches the off-chip drivers 1.892 ns after the falling edge of `clk`. `Ready` reaches the off-chip drivers 11.195 ns after the rising edge of `clk`. `Oe` reaches the off-chip drivers 1.088 ns after the falling edge of `clk`. These signals are sent, along with the `IO_CLK` signal, to the off-chip module and are latched by the rising edge of `IO_CLK`. `Io_clk` (the signal that goes into the driver that produces IO_CLK) rises about 1.5 ns after the rising edge of clk. Thus, `transmit` has about 3.6 ns of skew tolerance and `oe` has about 4.4 ns. However, the timing for `ready` signal is tight, with a negative skew tolerance. We actually need `IO_CLK` to be delayed about one nanosecond relative to `ready` so that `ready` will be sampled correctly. Unless the off-chip module employs a phase-locked loop to generate its clock, the delay from `IO_CLK` on the MAP and on the off-chip module is guaranteed to be more than a nanosecond because the pad receivers on the off-chip module are going to take about a nanosecond, and then `IO_CLK` needs to buffered up and distributed throughout the chip. The only concern is if `IO_CLK` is delayed by a significant portion of the clock cycle; `IO_AD` only tolerates about 5 ns of skew in either direction as it is only driven for half the I/O cycle. Should the clock skew be too great, a solution would be to PDFF `oe` instead of n-latching it. This delays `oe` by another half system clock cycle and brings it more in line with the timing for `ready`.

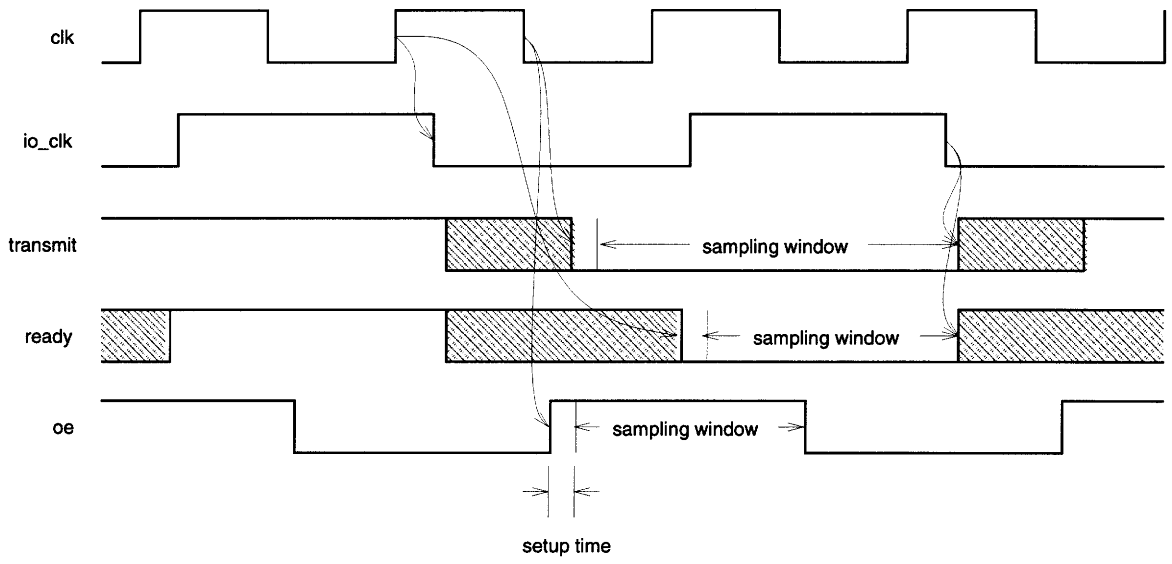


Figure 5-2: I/O output signal timing

Coming on-chip, the timing is tight for the data signals (`IO_AD`). The `IO_IRDY` and `IO_ITX` signals have an extra 10 ns because they start on the rising edge of `IO_CLK` whereas `IO_AD` starts being driven on the falling edge of `IO_CLK`. The critical path is the time it takes to send `IO_CLK` to the off-chip module plus the time it takes to send data back to the MAP. Table 5.1 are rough the estimates of the components of that path.

This path must fit within half a `IO_CLK` cycle, or 10 ns. This means the clock distribution skew needs to be within about 2 ns. If the skew is too great, the data does not arrive in time for the rising edge of `IO_CLK`, which latches the data on the MAP chip. Originally, there was more time because the data was to be latched on a delayed `IO_CLK`, specifically the output of the pad driver. This gives an extra 2.5 ns. This was abandoned because of wiring congestion problems. Still, there are several solutions. The first solution is for the off-chip module to generate the output enable on the *rising* edge of `IO_CLK` with some small delay. The delay ensures that there is no bus contention, as data from the MAP chip necessarily stays past the rising edge

Pad driver input to Pad output	2.50
Inter-package flight time	0.25
Pad receiver input to signal output	1.00
Clock distribution skew	????
Output enable logic	0.50
Pad driver enable to Pad output	2.50
Inter-package flight time	0.25
Pad receiver setup time	1.00
Total	8.00 + ????

Table 5.1: Critical path timing for receiving data. The clock distribution skew is unknown.

of `IO_CLK`. The second is to stretch out the clock cycle. There is some clock speed at which this is guaranteed to work because the timing is designed to work even with zero delays. Since the MAP chip is already going to run at a lower clock rate for other reasons, nothing needs to be done and the timing should work without modification.

5.3 Electrical Rules

For reliable operation, highly capacitive nodes need to be properly driven. The effect of under-driven nodes is poor rise and fall times. This causes high transient currents through the gates with under-driven inputs because both the nmos and pmos chains in the gates are partially on, providing a low-impedance path from power to ground.

Synergy takes this into account when synthesizing modules, so theoretically all nodes should be properly driven. However, there may be problems with the specifications interfaces between the synthesized modules. In addition, verifying the design gives an extra level of security. PrimeTime was used to verify the design both on the module level and on the inter-module level. No particularly slow nodes were found.

Chapter 6

Conclusion

In this thesis, I described the design and implementation of the I/O port controller for the MAP chip. The I/O port controller is responsible for managing the I/O port, which is used for communication between the MAP chip and external devices. The I/O port is intended to be connected to an off-chip module that then connects to other peripheral buses, such as SCSI or SBUS. The I/O port controller was implemented in Verilog HDL and synthesized to a standard cell library.

The I/O port controller was fully tested using both the original Verilog HDL and synthesized forms. Code was written which was able to fully utilize the 100 MB/s bandwidth of the I/O bus using 48-bit and 64-bit words. Timing was verified using a static timing analyzer and found to meet the desired specifications. Therefore the I/O port controller meets all its objectives and specifications.

I learned many things from this project. First, I learned how to carry out the design and implementation of a critical component from beginning to end. Second, I learned to keep things simple. The hardware ought to be as simple as possible yet allow complex things through higher level layers. Third, I learned to optimize only where it matters in order to devote time to more important aspects, such as testing. A part that works at a slow speed is better than one that does not work at all.

Appendix A

Test Programs

A.1 Non-burst mode test

```
#include "test.h"
#define IOPTR 0xC8221FFFFFFE0000

#define ALEN0 0x0000000000000000
#define ALEN1 0x0000800000000000
#define ALEN2 0x0001000000000000
#define ALEN3 0x0001800000000000

#define DLEN0 0x0000000000000000
#define DLEN1 0x0000200000000000
#define DLEN2 0x0000400000000000
#define DLEN3 0x0000600000000000

#define SIZE0 0x0000000000000000
#define SIZE1 0x0000000000008000
#define SIZE2 0x0000000000010000
#define SIZE3 0x0000000000018000

text;

SET_IREG_PTR((IOPTR | ALEN0 | DLEN0 | SIZE0), i2)
SET_IREG_PTR((IOPTR | ALEN0 | DLEN1 | SIZE1), i3)
SET_IREG_PTR((IOPTR | ALEN0 | DLEN2 | SIZE2), i4)
SET_IREG_PTR((IOPTR | ALEN0 | DLEN3 | SIZE3), i5)
SET_IREG_PTR((IOPTR | ALEN1 | DLEN0 | SIZE0), i6)
SET_IREG_PTR((IOPTR | ALEN1 | DLEN1 | SIZE1), i7)
SET_IREG_PTR((IOPTR | ALEN1 | DLEN2 | SIZE2), i8)
SET_IREG_PTR((IOPTR | ALEN1 | DLEN3 | SIZE3), i9)
SET_IREG_PTR((IOPTR | ALEN2 | DLEN0 | SIZE0), i10)
SET_IREG_PTR((IOPTR | ALEN2 | DLEN1 | SIZE1), i11)
SET_IREG_PTR((IOPTR | ALEN2 | DLEN2 | SIZE2), i12)
SET_IREG_PTR((IOPTR | ALEN2 | DLEN3 | SIZE3), i13)

instr memu ld i2, i14;
instr memu ld i3, i15;
instr ialu mov i14, i0
memu ld i4, i14;
instr ialu mov i15, i0
memu ld i5, i15;
instr ialu mov i14, i0
memu ld i6, i14;
instr ialu mov i15, i0
memu ld i7, i15;
instr ialu mov i14, i0
memu ld i8, i14;
instr ialu mov i15, i0
memu ld i9, i15;
instr ialu mov i14, i0
memu ld i10, i14;
instr ialu mov i15, i0
memu ld i11, i15;
instr ialu mov i14, i0
memu ld i12, i14;
instr ialu mov i15, i0
memu ld i13, i15;

instr ialu mov i14, i0;
instr ialu mov i15, i0;

SET_IREG(0x1000010000100000, i14)
SET_IREG(0x0001000100010001, i15)

instr ialu addu i14, i15, i14
memu st i14, i2;
instr ialu addu i14, i15, i14
memu st i14, i3;
instr ialu addu i14, i15, i14
memu st i14, i4;
instr ialu addu i14, i15, i14
memu st i14, i5;
```

```
 instr ialu addu i14, i15, i14
 memu st i14, i6;
 instr ialu addu i14, i15, i14
 memu st i14, i7;
 instr ialu addu i14, i15, i14
 memu st i14, i8;
 instr ialu addu i14, i15, i14
 memu st i14, i9;
 instr ialu addu i14, i15, i14
 memu st i14, i10;
 instr ialu addu i14, i15, i14
 memu st i14, i11;
 instr ialu addu i14, i15, i14
 memu st i14, i12;
 instr ialu addu i14, i15, i14
 memu st i14, i13;

 instr ialu addu i14, i15, i14
 memu stsu i14, i2, cc0;
 instr ialu addu i14, i15, i14
 memu ct cc0 stsu i14, i3, cc0;
 instr ialu addu i14, i15, i14
 memu ct cc0 stsu i14, i4, cc0;
 instr ialu addu i14, i15, i14
 memu ct cc0 stsu i14, i5, cc0;
 instr ialu addu i14, i15, i14
 memu ct cc0 stsu i14, i6, cc0;
 instr ialu addu i14, i15, i14
 memu ct cc0 stsu i14, i7, cc0;
 instr ialu addu i14, i15, i14
 memu ct cc0 stsu i14, i8, cc0;
 instr ialu addu i14, i15, i14
 memu ct cc0 stsu i14, i9, cc0;
 instr ialu addu i14, i15, i14
 memu ct cc0 stsu i14, i10, cc0;
 instr ialu addu i14, i15, i14
 memu ct cc0 stsu i14, i11, cc0;
 instr ialu addu i14, i15, i14
 memu ct cc0 stsu i14, i12, cc0;
 instr ialu addu i14, i15, i14
 memu ct cc0 stsu i14, i13, cc0;

HALT
SPIN
end;
```

A.2 Burst mode test

```
#include "test.h"
#define IOPTR 0xC8221FFFFFFE0000

#define ALEN0 0x0000000000000000
#define ALEN1 0x0000800000000000
#define ALEN2 0x0001000000000000
#define ALEN3 0x0001800000000000

#define DLEN0 0x0000000000000000
#define DLEN1 0x0000200000000000
#define DLEN2 0x0000400000000000
#define DLEN3 0x0000600000000000

#define SIZE0 0x0000000000000000
#define SIZE1 0x0000000000008000
#define SIZE2 0x0000000000010000
#define SIZE3 0x0000000000018000

text;

SET_IREG_PTR((IOPTR | ALEN0 | DLEN0 | SIZE0), i2)
instr ialu imm ##16, i3;
instr memu stscnd ua, 1, i3, i2, cc0; /* initiate burst load */

SET_IREG_PTR((IOPTR | ALEN3 | DLEN0 | SIZE0), i2)
instr ialu br     input;
instr ialu imm    ##0x1000, i9;
instr memu leab   i1, i9, i9;
instr memu lea    i1, #4, i4;


SET_IREG_PTR((IOPTR | ALEN2 | DLEN1 | SIZE1), i2)
instr ialu imm ##16, i3;
instr memu stscnd ua, 1, i3, i2, cc0; /* initiate burst load */

SET_IREG_PTR((IOPTR | ALEN3 | DLEN0 | SIZE0), i2)
instr ialu br     input;
instr;
instr;
instr memu lea    i1, #4, i4;
```

```
SET_IREG_PTR((IOPTR | ALEN1 | DLEN2 | SIZE2), i2)
instr ialu imm ##16, i3;
instr memu stscnd ua, 1, i3, i2, cc0; /* initiate burst load */

SET_IREG_PTR((IOPTR | ALEN3 | DLEN0 | SIZE0), i2)
instr ialu br     input;
instr;
instr;
instr memu lea    i1, #4, i4;


SET_IREG_PTR((IOPTR | ALEN0 | DLEN3 | SIZE3), i2)
instr ialu imm ##16, i3;
instr memu stscnd ua, 1, i3, i2, cc0; /* initiate burst load */

SET_IREG_PTR((IOPTR | ALEN3 | DLEN0 | SIZE0), i2)
instr ialu br     input;
instr;
instr;
instr memu lea    i1, #4, i4;

/**************************************************************/
SET_IREG_PTR((IOPTR | ALEN1 | DLEN0 | SIZE0), i2)
instr ialu imm ##8, i3;
instr memu stscnd ua, 0, i3, i2, cc0; /* initiate burst store */

SET_IREG_PTR((IOPTR | ALEN3 | DLEN0 | SIZE0), i2)
instr ialu br     output_noflow;
instr ialu imm    ##0x1000, i9;
instr memu leab   i1, i9, i9;
instr memu lea    i1, #4, i4;


SET_IREG_PTR((IOPTR | ALEN0 | DLEN1 | SIZE1), i2)
instr ialu imm ##8, i3;
instr memu stscnd ua, 0, i3, i2, cc0; /* initiate burst store */

SET_IREG_PTR((IOPTR | ALEN3 | DLEN1 | SIZE0), i2)
instr ialu br     output_noflow;
instr;
instr;
instr memu lea    i1, #4, i4;
```

```
SET_IREG_PTR((IOPTR | ALEN2 | DLEN2 | SIZE2), i2)
instr ialu imm ##8, i3;
instr memu stscnd ua, 0, i3, i2, cc0; /* initiate burst store */

SET_IREG_PTR((IOPTR | ALEN3 | DLEN2 | SIZE0), i2)
instr ialu br     output_noflow;
instr;
instr;
instr memu lea    i1, #4, i4;


SET_IREG_PTR((IOPTR | ALEN1 | DLEN3 | SIZE3), i2)
instr ialu imm ##8, i3;
instr memu stscnd ua, 0, i3, i2, cc0; /* initiate burst store */

SET_IREG_PTR((IOPTR | ALEN3 | DLEN3 | SIZE0), i2)
instr ialu br     output_noflow;
instr;
instr;
instr memu lea    i1, #4, i4;

/*************************************************************/
SET_IREG_PTR((IOPTR | ALEN2 | DLEN0 | SIZE0), i2)
instr ialu imm ##8, i3;
instr memu stscnd ua, 0, i3, i2, cc0; /* initiate burst store */

SET_IREG_PTR((IOPTR | ALEN3 | DLEN0 | SIZE0), i2)
instr ialu br     output_flow;
instr ialu imm    ##0x200, i9;
instr memu leab   i1, i9, i9;
instr memu lea    i1, #4, i4;


SET_IREG_PTR((IOPTR | ALEN1 | DLEN1 | SIZE0), i2)
instr ialu imm ##8, i3;
instr memu stscnd ua, 0, i3, i2, cc0; /* initiate burst store */

SET_IREG_PTR((IOPTR | ALEN3 | DLEN1 | SIZE0), i2)
instr ialu br     output_flow;
instr;
instr;
instr memu lea    i1, #4, i4;


SET_IREG_PTR((IOPTR | ALEN0 | DLEN2 | SIZE0), i2)
```

```
instr ialu imm ##8, i3;
instr memu stscnd ua, 0, i3, i2, cc0; /* initiate burst store */

SET_IREG_PTR((IOPTR | ALEN3 | DLEN2 | SIZE0), i2)
instr ialu br     output_flow;
instr;
instr;
instr memu lea    i1, #4, i4;


SET_IREG_PTR((IOPTR | ALEN2 | DLEN3 | SIZE0), i2)
instr ialu imm ##8, i3;
instr memu stscnd ua, 0, i3, i2, cc0; /* initiate burst store */

SET_IREG_PTR((IOPTR | ALEN3 | DLEN3 | SIZE0), i2)
instr ialu br     output_flow;
instr;
instr;
instr memu lea    i1, #4, i4;

HALT
SPIN


/***************************************************************/


/* stream i3 (must be even & >= 4) words in,
 * store to memory pointed to by i9 */
/* use pointer i2 to access IO space */
/* return via pointer i4 */
/* trahes i3, i5 and i6 */
/* i9 points past end of memory */

input:

instr ialu ule    i3, #4, cc1
memu ld     i2, i6;

_loop1:
instr ialu cf cc1 br _loop1
memu ld     i2, i5;
instr memu st     i6, #8, i9;
instr ialu sub    i3, #2, i3
memu ld     i2, i6;
```

```
instr ialu ule    i3, #4, cc1
memu st     i5, #8, i9;

instr ialu jmp    i4
memu ld     i2, i5;
instr;
instr memu st     i6, #8, i9;
instr memu st     i5, #8, i9;


/* stream i3 (must be even & >= 4) words out,
 * load from memory pointed to by i9 */
/* use pointer i2 to access IO space */
/* return via pointer i4 */
/* trashes i3, i5 and i6 */
/* i9 points past end of memory */

output_noflow:

instr ialu ule i3, #4, cc1
memu ld i9, #8, i6;

_loop2:
instr ialu cf cc1 br _loop2
memu ld i9, #8, i5;
instr memu st i6, i2;
instr ialu sub i3, #2, i3
memu ld i9, #8, i6;
instr ialu ule i3, #4, cc1
memu st i5, i2;

instr ialu jmp i4
memu ld i9, #8, i5;
instr;
instr memu st i6, i2;
instr memu st i5, i2;




output_flow:

instr ialu ule i3, #4, cc1
memu ld i9, #8, i6;
_loop3:
```

```
instr ialu cf cc1 br _loop3
memu ld i9, #8, i5;
instr memu ct cc0 stsu i6, i2, cc0;
instr ialu sub i3, #2, i3
memu ld i9, #8, i6;
instr ialu ule i3, #4, cc1
memu ct cc0 stsu i5, i2, cc0;

instr ialu jmp i4
memu ld i9, #8, i5;
instr;
instr memu ct cc0 stsu i6, i2, cc0;
instr memu ct cc0 stsu i5, i2, cc0;

end;
```

Bibliography

[1] William J. Dally, Stephen W. Keckler, Nick Carter, Andrew Chang, Marco Fillo, and Whay S. Lee. The map instruction set reference manual v1.53. CVA Memo 59, Massachusetts Institute of Technology, August 1996.

[2] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine Multicomputer. Artifical Intelligence Laboratory Memo 1532, Massachusetts Institute of Technology, 1995.

[3] Marco Fillo and Whay S. Lee. The architecture of the MAP intercluster switch. CVA Memo 86, Massachusetts Institute of Technology, 1996.

[4] Parag Gupta. The design and implementation of the memory unit. CVA Memo 82, Massachusetts Institute of Technology, 1996.