Designing the Port Interface Unit for the Lutonium Asynchronous Microcontroller

Thesis by

Eino-Ville Talvala

In Partial Fulfillment of the Requirements

for the Degree of

Bachelor of Science in Electrical Engineering

California Institute of Technology

Pasadena, California

2003

(Submitted June 5, 2003)

Abstract

The requirements, features, and implementation of the port interface unit for the Lutonium asynchronous microcontroller are covered in detail. The Lutonium is a new microcontroller that is pin- and software compatible with the Intel 8051 family of microcontrollers, and it is being designed by the Caltech Asynchronous VLSI Group. The port interface unit controls five 8-bit I/O ports, exposing them to the processor as directly accessible registers. The port interface unit also sequences external memory bus access phases using the same I/O ports. The final design allows both backward compatibility with the 8051 series and many advanced power-saving features. It is built around a primary state machine responsible for controlling and sequencing basic operations. The transistor networks for the port unit are complete, and extensive simulations have been done to verify the logical correctness of the design. The layout has not yet been completed, but preliminary performance numbers have been extracted from the transistor-level design.

Table of Contents

1. Introduction

The Intel 8051 microcontroller was originally designed in 1980 and has remained highly popular to this day. Dozens of variants of the original 8051 have been released by various semiconductor manufacturers, with added features, speed, or both, tailored for specific tasks. The 8051 and its many derivatives are the world's most popular microcontroller family.

It is for this reason that the Caltech Asynchronous VLSI Group is currently designing the Lutonium microcontroller [1], an asynchronous version of the Intel 8051 microcontroller. The goals of the design are low power consumption and high speed; in short, very high efficiency, on the order of 1000 MIPS/Watt. Funded by DARPA, the project is a showcase for the effectiveness of the asynchronous design approach, applied to a common, widely used microcontroller.

The 8051 is a Harvard architecture microcontroller, in its base form featuring 4 8-bit I/O ports and the ability to access external instruction or data memory through the I/O ports, among many other features. [2]

This thesis details the development of the port interface unit of the Lutonium, which interfaces the Lutonium core to five 8-bit I/O ports. The port interface allows direct access to the ports through memory-mapped registers, and orchestrates external memory bus accesses through the I/O ports. Due to this dual role, the port interface is named the Port Register/Data Memory (PRDM) unit. In order to satisfy both project goals and backwards-compatibility requirements, the PRDM is highly configurable, supporting several power- and time-saving features as well as a default 8051-compatible mode.

The PRDM is designed with a central state machine that contains all branching and sequencing logic, connected to several data path units that route port data bytes between the Lutonium core and the port pins. The state machine is implemented as a large ROM automatically generated by the ROMantic asynchronous ROM generator [10]. All other subunits of the PRDM are designed using the quasi-delay-insensitive asynchronous design style pioneered by the research group. The lowest-level logic units are based on the pre-charge half-buffer cell, a standard cell template, which makes each logic unit relatively straightforward to implement.

2. 8051 Overview

The 8051 microprocessor and its numerous derivatives form the world's most widely used microcontroller family. Every major semiconductor manufacturer has its own versions of the 8051, with various additional peripherals, functions, and options added to the baseline processor. There is a very large installed user base for this microcontroller, and thus compatibility with the baseline specification is of great importance.

The 8051 is an 8-bit microcontroller, with an instruction set optimized for 8-bit control applications. The instructions are between 1 and 3 bytes long, depending on the instruction and its addressing modes. The microcontroller is a Harvard architecture, with separate data and instruction memories. Its internal data memory consists of 127 general purpose registers, and of a varying number of special-function registers (SFRs). The

SFRs include the accumulator, the instruction pointer, the processor status word, as well the registers used to access and configure interrupts, timers, and other peripherals.

The basic 8051 has four 8-bit I/O ports, P0 through P3, with four corresponding SFRs. The ports, except for P0, are quasi-bidirectional; to use a pin for input, a 1 must be written to it. The processor uses weak pull-ups internally, so that an external source can then pull a pin low or high, making the pin function like an input. However, the processor will source current through the pin when configured as an input. Port 0 is open collector when used as an I/O port, requiring external pull-ups if used as an output. However, P0 is truly bidirectional since it will float when used as an input. When used for external memory access, P0 uses strong internal pull-ups, and no external pull-ups are required.

An external memory bus access can either use a 16-bit address read from the microcontroller's 16-bit data pointer register (DPTR, segmented into two 8-bit registers DPH and DPL), or an 8-bit address read from the Lutonium register file.

When used for external memory bus access, Port 0 has the data byte and the least significant byte of the address multiplexed on it. A dedicated Address Latch Enable (ALE) pin is used to signal external circuitry to latch the address LSB before Port 0 switches to either reading or writing the data byte. If a 16-bit address is used, Port 2 is used to output the high byte of the address. In this mode, Port 2 also uses strong internal pull-ups to output the address MSB. Finally, pins 6 and 7 of Port 3 are used to signal a write or a read on the bus, respectively. In the basic 8051, Port 1 is unchanged during an external memory bus access.

After an external memory bus access, the state of Port 2 and its SFR is restored to its pre-access state. Port 0, however, is overwritten with the value FF during the memory access.

The 8051 also includes bit operations, which only affect single bits in a given register. Only some of the registers of the 8051 are bit-accessible, including the port SFRs. Internally, the bit operations are performed by reading the whole byte from a register, modifying the single bit as needed, and then writing the register's value back, all in the same operation cycle.

Any instruction that is considered to be read-modify-write (this includes all bit operations) that is modifying a port register reads the output latch of the port, instead of the input latch. This means, for example, that if Port 0 is set to 80, where pin 7 is pulled high to be used as an input, and that an external source is in fact writing 0 to pin 7, any read-modify-write instruction will still read Port 0's value as 80, not as 00.

In the original 8051, the ports are also used to access external instruction memory. However, for the Lutonium, all of the instruction memory is internal to the unit, and thus external instruction memory access for the 8051 will not be discussed here.

For further details on the 8051 architecture, refer to Philips Semiconductor's set of user manuals for the 8051. [1][2][3]

3. Asynchronous Design Style Overview

Asynchronous circuitry, at the basic level, is digital circuitry that does not use a global clock signal. Because this coordinating signal is absent, the circuitry itself must self-synchronize its internal sequencing and processes. A key feature of this synchronization is the timing assumptions made by the designer of a digital circuit. The timing assumption of a synchronous circuit is simple: The outputs of one stage of computation must be valid at the input of the next stage when the clock cycle completes. This means that during the clock cycle, the outputs of a stage can glitch (assume incorrect values) as the stage's inputs transverse the circuit. As long as the computation is completed before the end of the clock cycle, a synchronous circuit will work correctly. For asynchronous designs, different timing assumptions must be made. While it may seem elegant to strive to make no timing assumptions about the behavior of a circuit, and use only so-called delay-insensitive circuits, it can be proved that one cannot construct circuits of any complexity with such circuits [5].

The circuitry used by the Caltech Asynchronous Research Group is called quasi-delay-insensitive (QDI), referring to the fact that only one minimal timing assumption is made. The assumption is that a signal on a wire that forks to multiple paths reaches the end of each fork at the same time on a transistor-switching timescale. That is, a signal travels down to the end of all the forks of a wire faster than any of the logic gates attached to these forks can respond to the changing signal. This assumption is referred to as the *isochronic fork* assumption. This single assumption allows a rich set of circuits to be implemented, from which any logic circuit can be built. [8] Of course, this timing assumption can be violated, if a signal wire branches and the two segments are of widely varying lengths. However, since most wires are local to a process, it is easy to lay out circuitry that does not violate this assumption. With minimal effort, long-distance wires can also be arranged to preserve isochronity. Note that, in the synchronous world, this same timing assumption is applied to the global clock signal; the condition where the clock signal does not switch everywhere in the circuit at the same time is called clock skew, and is a major problem facing chip designers in synchronous circuits.

To design a QDI logic block, one starts with a top-level description of the block using a language called Communicating Hardware Processes (CHP). In this language, each logic subunit is an individual process, which receives and transmits information over channels. Each channel is a collection of wires, with the sender controlling all but one of the wires. The last wire is an acknowledge wire, controlled by the receiver. All channels are strictly one-way; there is no concept of a true bidirectional bus in CHP. [9]

A message is transmitted on a channel using a four-phase handshake and delay-insensitive codes. A delay-insensitive code is a way of encoding a value on multiple wires where the order of reception of the signals on the wire does not matter; common to all of them is some way of signifying 'no value' or neutrality on the channel. The main delay-insensitive code used in the Lutonium is the one-of-N (1ofn) code. With this code, N wires are required to represent N values. A neutral 1ofn channel maintains all of its outputs at logic low; this represents null or 'no value'. When value N needs to be sent on the channel, the Nth wire on the channel is pulled high; the other wires stay low. This way, the channel does not pass through any intermediate states before stabilizing at its correct output value. Similarly, the channel returns to neutrality simply by dropping that single wire back low. However, the sender must have a way of knowing when the receiver has processed the value sent to it, since the sender cannot assume anything about the speed of propagation or processing of the receiver. For this purpose, the receiver has an acknowledge wire. Upon receiving and processing the inputs, the receiver raises the acknowledge wire high. This signifies to the sender that the data it sent has been received, and the sender will then return the channel to neutral. The receiver, upon seeing the channel return to neutral, will drop the acknowledge wire back low as soon as it is ready to accept more inputs.

This four-phase handshake (sender valid, receiver acknowledge, sender neutral, receiver neutral) combined with delay-insensitive codes allows two CHP processes to transmit data between each other and synchronize their execution. However, a top-level CHP description is usually too large to directly convert to a transistor network, and must first be subdivided into manageable blocks. A CHP description of a logic block can be formally decomposed into smaller blocks connected by channels. This decomposition is continued until each sub-block is small enough to easily convert to a transistor network.

In the Lutonium design, these lowest-level blocks are implemented as pre-charge half-buffers (PCHB). Since each CHP process receives and sends on various channels, the process can be represented as a buffer whose output is some function of its inputs. Therefore, the PCHB model can implement any CHP process, with QDI circuitry. Figure 1 shows the standard parts of a PCHB circuit, which contains several sections. First, there are several completion trees, which determine the state of the inputs and outputs of the circuit. The input validity tree determines if all the inputs to the circuit have become valid, and the output validity tree determines if the outputs of the circuit have become valid. The pre-charge pull-up network pulls the outputs to neutrality at the end of each cycle. The actual computation of the output function is done in the fast pull-down logic, gated by the acknowledge wires from the receivers of the output channels and the acknowledge wires heading back to the senders of the input channels. The acknowledge wire goes high when both the input and output validities are high, and drops back low when both input and output validities return low. (This logic function is called the c-element)

An example of a PCHB circuit is a straightforward 1-bit buffer, which simply sends the value it receives on its input channel forward on its output channel. The input channel, conventionally referred to as L (left) consists of 2 incoming wires forming a 1of2 channel, and an acknowledge wire from the buffer to the sender of the L channel. The output channel R (right) is a mirror image of the L channel, with two outgoing wires forming a 1of2 channel, and an incoming acknowledge wire. The pull-down logic simply implements the identity function; an incoming '1' causes an outgoing '1'.

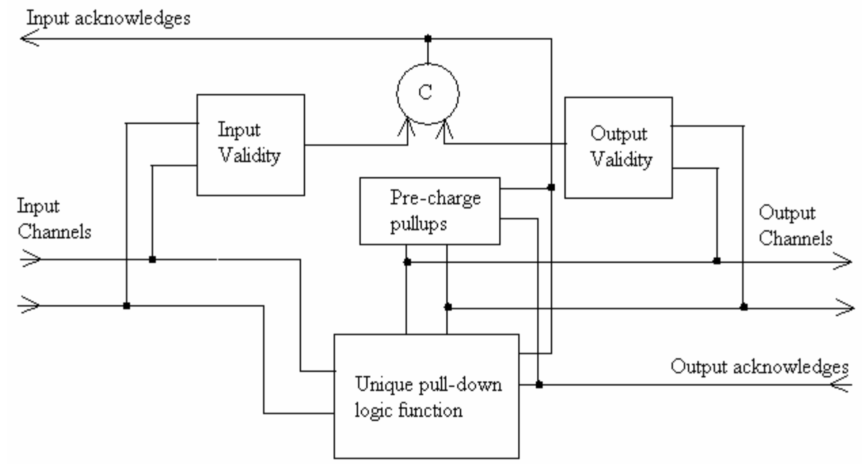


Figure 1: PCHB Cell Block Diagram

The sequence of actions of the PCHB 1-bit buffer goes as follows: At the end of the previous cycle of operations, the buffer's inputs and outputs are both neutral (both wires of the 1of2 channel are low), and both acknowledge wires are low. The input and output validities are both low, signifying the neutrality of the wires. Now, the sender on the L channel sets wire 1 of the L channel high, signifying a '1' being sent down the channel. Both the input validity tree and the pull-down logic receive this value.

In the pull-down network, the input activates transistors that pull the value of wire 1 on the output high. The output of the input validity tree changes to 1, signifying a valid input value. As soon as the pull-down network completes its activity, the output validity tree sees a valid output, and changes to output a 1 as well. Because the buffer has no further need for the input at this point, it can acknowledge the L channel. This happens as the c-element controlling the input acknowledge switches, setting the input acknowledge wire high. This cuts off the drive to the output wires, but a simple state-holding circuit maintains the values on the output wires for now. At this point, the buffer waits for both the L sender and the R receiver to react to its actions. At some point, the L sender will return its output 1of2 channel to neutrality, which returns the input validity to 0. Also, the R receiver will raise its acknowledge wire, signaling its reception of the sent values. This triggers the pre-charge pull-up circuitry, which returns the buffer's outputs to neutrality. This in turn changes the output validity to 0. With both the input and output validities low, the c-element switches to 0, lowering the L channel acknowledge. At some later point, the R receiver will lower its acknowledge wire, which allows the buffer to output values on R again as soon as they are sent on L.

PCHB circuits can easily be modified to contain conditional inputs and outputs (channels that are sent and received on only at certain times, not every cycle), as well as state variables. With those basic templates, any logic circuit can be built from a composition of sets of PCHB half-buffers, which are derived from the decomposition of the CHP description of the circuit.

One of the major advantages of this design style is that the outward functioning of the formally decomposed set of sub-processes is identical to the functioning of the original top-level process. This reduces the number of mistakes that can be made in the design and implementation of the circuit, since each sub-process can be converted to PCHB circuits and tested individually, apart from the rest of the circuit.

Further, adjustments can be made to the circuit in order to improve its performance characteristics. For the Lutonium processor, the goal is to optimize the circuitry for Et^2, where E is a measure of the energy expended by a process and t is a measure of the time taken by the process to complete its calculation. To first order, the Et^2 measure of a circuit is independent of the logic voltage level. In an asynchronous circuit, the logic voltage can be varied a great deal without affecting the logical correctness of the circuit's functioning. However, a lower voltage results in slower-switching transistors, slowing down the processor. But a lower voltage also translates to less power consumption. By modifying the voltage of the circuit, a designer can trade off speed and power requirements with each other; therefore, using a metric such as power consumption or speed when designing a circuit for power efficiency will not give optimal results. Using the Et^2 metric, however, has been shown to be a good method of optimization [7]. If two processors are running at the same rate, the one which is better optimized for Et^2 will consume less power, and if two processors are burning the same amount of power, the one optimized for Et^2 will be running faster.

To optimize a QDI circuit, one can utilize two techniques. The higher-level of these is called slack matching. In short, buffers can often be added into the channels between sub-processes that improve the cycle time of the entire process. This typically is done by equalizing the amount of 'slack' in parallel chains of processes in a circuit. Slack, roughly, is the number of data units that can fit into a process chain at one time. A single PCHB circuit has a slack of ½; two PCHB circuits form a full buffer, with the capability of holding one piece of data inside the chain. To slack match a circuit, one adds buffers to chains of processes until their slack is roughly equal to the slack of other parallel chains of processes in the circuit.

4. Port Interface Requirements, Features, and Top-Level Specification

The techniques described above form the foundation of designing an asynchronous QDI circuit, of which the Lutonium is a complex example. The design style for the Lutonium is top-down; a high-level description is broken down into smaller and smaller pieces, until each piece can be easily implemented as a PCHB circuit. Therefore, a top-level description of the functionality and behavior of the Port Register/Data Memory (PRDM) unit was required before its detailed design could begin.

In writing this description, two main requirements needed to be fulfilled. First, the PRDM must support backward-compatible modes, so that old 8051 code can run on the Lutonium with no or minimal modification. Second, the PRDM, as per Lutonium's design goals, should be low-power, high-efficiency, and clock-independent. It is clear that these two goals are contradictory. The quasi-bidirectional ports of the 8051 are wasteful in power; the memory bus access sequence of the 8051 is keyed off a synchronous external oscillator; the multiplexing of data and address on Port 0 adds to the external circuitry required and thus to overall power consumption. To allow the PRDM to support two conflicting basic goals, it was decided to add a great deal of configurability to the PRDM. This significantly complicates the overall design of the PRDM unit, but also greatly increases the unit's functionality and usefulness.

4.1. PRDM Configuration and SFRs

To begin with, each I/O port was assigned 4 SFRs as opposed to the original single SFRs. These SFRs are labeled P*n* or P*n*c, P*n*d, P*n*o, and P*n*i, where *n* is the port number. The P*n* (or P*n*c) register is the compatibility register, with default functionality identical to the 8051's single port SFR. It is also the only bit-accessible register for each port. The P*n*d register contains the output data byte for the port, while the P*n*o register contains the output latch enables for each port, with 0 being enabled. For example, if the P0d SFR contains 3A, and P0o contains F0, then pins 4, 5, 6, and 7 of Port 0 are configured as inputs, and pins 0,1,2,3 are outputs, with value A output on them. Finally, the P*n*i register contains the input latch data byte. Having all four registers for each port is required in order to support both fully bidirectional ports as well as the backward-compatible quasi-bidirectional ports.

In quasi-bidirectional mode (or compatibility mode) which is the default mode, writes to the compatibility register are redirected to the output enable register, and the data registers hold their default value of 00. External resistors are required in this mode to act as weak pull-ups for the port pins. For example, in compatibility mode, writing 3F to P0c results in 3F being written to P0o. The bits written with a '0' become outputs, outputting the value in the P0d register for that bit. In this case, the value output will always be '0'. For the bits in the P0o register that are written with '1', the matching pins become floating inputs, and the external pull-ups pull the pins high. Therefore, the value 3F is correctly written to the port, duplicating the behavior of the original 8051, including the current sourcing of inputs.

In fully bidirectional mode, writing P*n*c results in a write to P*n*d. In this mode, the programmer must access the P*n*o register directly to specify the input/output status of each port pin. In both modes, reading P*n*c reads the P*n*i register, except in the case of the read-modify-write instruction, in which case the register that will be written is read. So, for example, in fully bidirectional mode, a read-modify-write instruction on P0c reads the P0d register, and then rewrites the value of P0d, while a regular instruction reading the P0c register will redirect the read to the P0i register.

The P*n*c registers are also the only bit-addressable ones, due to the structure of the 8051 bit-addressable memory space. However, the redirection described above allows bit operations to be done either on the output data or the output enable byte.

The PRDM also supports demultiplexing the data and address bytes sent on P0 during an external memory bus access. When demultiplexed, the address byte is written out on P1 instead, the data is still written to P0, and the ALE pin becomes unnecessary. This allows both for faster memory accesses, as well as reduced external hardware, since no external latch is needed to hold the address byte after P0 switches to reading/writing the data byte.

Also added to the PRDM is a fast read mode, where the read cycle is compressed significantly, allowing faster sequential reads off the memory bus, assuming external hardware can keep up with the faster access cycle.

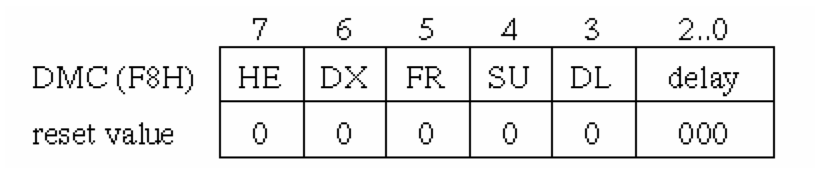
The programmer can also select whether the values of P1 and P2 are restored to their pre-memory bus access state after the read or write concludes. The default state corresponds to the 8051: P1 and P2 are restored after the memory bus access completes. Choosing sustain mode, where P1 and P2 are permanently overwritten, reduces the number of times the port pins change during a memory bus access, saving power, and shortens the access cycle somewhat.

Finally, the PRDM can base its sequencing on two different timing sources: An oscillator signal, or an internal delay line. The oscillator signal can come either from an internal or an external oscillator; the external oscillator is a typical quartz crystal chip used to drive standard synchronous designs, and allows the most backwards compatibility with existing 8051 systems. When using the internal delay line, the programmer can further specify the period of the delay line.

All of the above is controlled through a dedicated SFR, the DMC (Data Memory Control) register; its fields are shown in Figure 2 below.

- Pins 0 through 2 (delayperiod) control the number of oscillator ticks or delay line cycles that the PRDM waits for during a standard delay in its sequencing.

- Pin 3 (DeLay control) controls whether the PRDM uses the internal delay line (DL=1) or an oscillator (DL=0)
- Pin 4 (SUstain) controls whether the values of P1 and P2 are restored (SU=0) after a memory bus accesses, or whether they are left as they are at the end of the access (SU=1)
- Pin 5 (Fast Read) controls whether the fast read mode is enabled. In fast read mode (FR=1), some delays are eliminated in the external memory access sequence.
- Pin 6 (DemultipleX) controls whether the data and address are multiplexed on Port 0 (DX=0) during an external memory access, or whether the address is output on Port 1 (DX=1) and the ALE signal is unused.
- Pin 7 (High Enable) controls whether the PRDM operates in quasi-bidirectional backwards compatibility mode (HE=0), or in fully bidirectional high efficiency mode (HE=1)



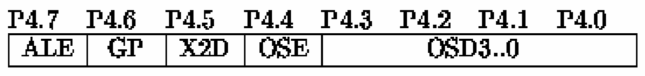
	7	6	5	4	3	2..0
DMC (F8H)	HE	DX	FR	SU	DL	delay
reset value	0	0	0	0	0	000

Figure 2: DMC register fields

The PRDM also supports a fifth port, P4, which is mostly an internal port. It is used to control some of the internal peripherals of the Lutonium, such as the selection of internal or external oscillator to drive both the PRDM's sequencing and the timer peripherals. However, this port also contains the ALE pin, the sole external port of P4 in the default Lutonium package.

- Pins 0 through 4 (OSD3...0) on Port 4 control the internal oscillator period.
- Pin 4 (OSE) switches between the external and internal oscillators
- Pin 5 (X2D) enables or disables the inverter driving the XTAL2 pin of the processor.
- Pin 6 (GP) is a general-purpose pin that could be made available as an IO pin on a larger chip package.
- Pin 7 (ALE) is the pin used to signal external latching circuitry to latch the address off the multiplexed Port 0 during an external memory bus access cycle.

See figure 3 for the fields of Port 4, and figure 4 for the entire SFR memory map of the Lutonium.

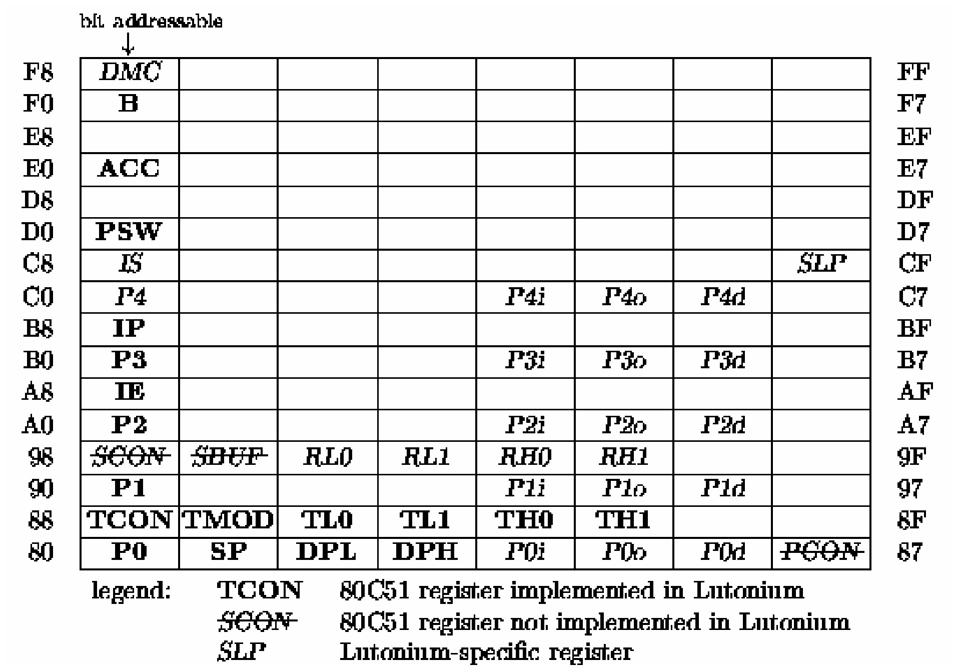


P4.7	P4.6	P4.5	P4.4	P4.3	P4.2	P4.1	P4.0
ALE	GP	X2D	OSE	OSD3..0			

Figure 3: Port 4 fields

4.2. PRDM External Interface

The external memory access sequence for the Lutonium is a typical sequence, possibly modified by the FR and DX flags, both of which speed up the overall access sequence. Turning on demultiplexing removes the entire ALE sequence, and turning on fast read removes a few stages of delay from the sequence. Figure 5 shows the various patterns of the Lutonium's external memory access cycle.



bit addressable

F8	*DMC*								FF
F0	B								F7
E8									EF
E0	ACC								E7
D8									DF
D0	PSW								D7
C8	*IS*							*SLP*	CF
C0	*P4*				*P4i*	*P4o*	*P4d*		C7
B8	IP								BF
B0	P3				*P3i*	*P3o*	*P3d*		B7
A8	IE								AF
A0	P2				*P2i*	*P2o*	*P2d*		A7
98	~~SCON~~	~~SBUF~~	*RL0*	*RL1*	*RH0*	*RH1*			9F
90	P1				*P1i*	*P1o*	*P1d*		97
88	TCON	TMOD	TL0	TL1	TH0	TH1			8F
80	P0	SP	DPL	DPH	*P0i*	*P0o*	*P0d*	~~PCON~~	87

legend:		
	TCON	80C51 register implemented in Lutonium
	~~SCON~~	80C51 register not implemented in Lutonium
	SLP	Lutonium-specific register

Figure 4: Lutonium special function register memory map

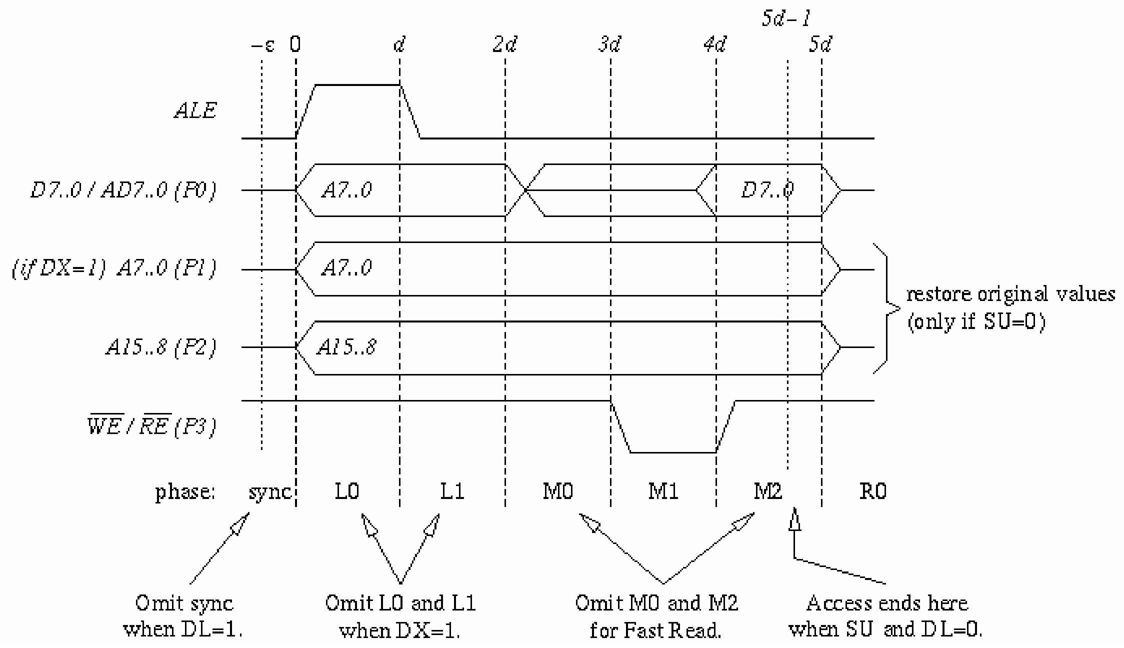


Figure 5: External memory bus access sequence

The Lutonium is being built for a 40-pin package, with the pin out and functionality matching the standard 8051 pin out as much as possible. Figure 6 shows both the Lutonium pin out, and a typical 8051 pin out. The main differences are the replacement of the serial input/output on P3.1 and P3.2 with two programming pins, which can be used to program the Lutonium's internal instruction memory on boot. Also, the PSEN and EA pins, both used to access external instruction memory, are replaced with pins that can be used to finely control the logic supply and well voltages allowing for power versus speed optimization.

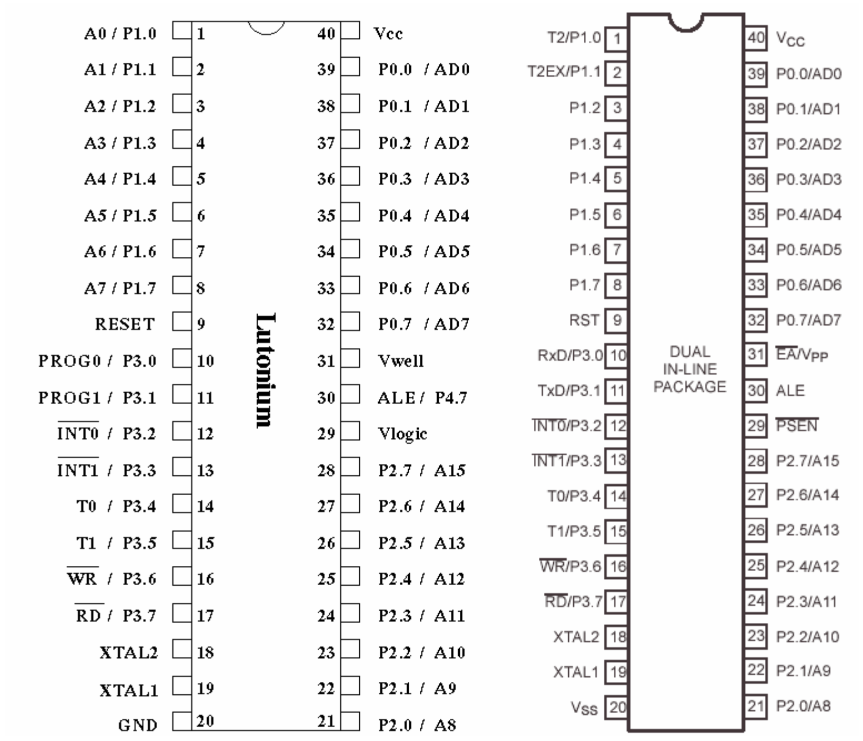


Figure 6: Lutonium and standard 8051 pin outs

The PRDM is not the last stage between the port pins and the core of the Lutonium; each port has its own simple port driver unit that converts the data sent to it by the PRDM into pin voltages. Figure 7 shows the peripheral interface as it relates to the PRDM:

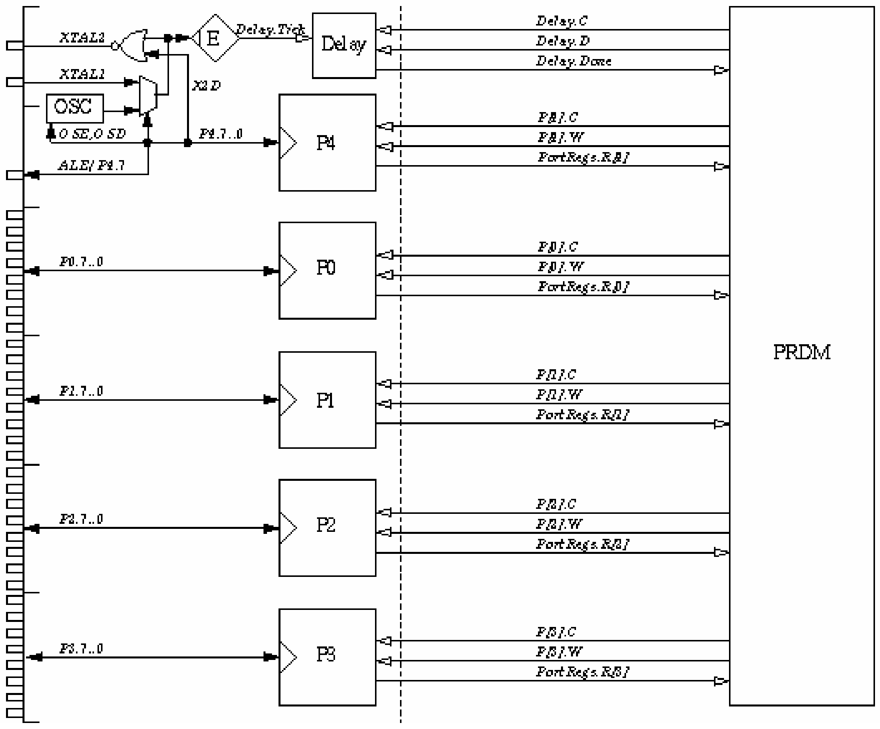


Figure 7: PRDM peripheral interface

Each port driver unit contains a latch storing the output data, the output enable mask, and the synchronization circuitry to sample the state of the pins when needed. The Delay unit is responsible for handling the delays required in the external memory access sequence. It is controlled by PRDM, and can run off either an internal delay line, or an oscillator signal. Using the internal delay line removes any need for an external oscillator chip, especially if the Lutonium timer peripherals aren't in use.

4.3. PRDM Internal Interface

Below in Figure 8 is the internal block diagram of the Lutonium, with parts relevant to the PRDM highlighted. In this diagram, the PRDM is split into its two logical components: the DMem unit, which handles external memory bus accesses, and the PortRegs unit, which handles direct port accesses.

The PRDM receives its control from the decode unit, which is responsible for converting the instruction opcodes to internal control signals. There are a total of 40 unique control signal combinations that the decode unit could send to the PRDM, and the PRDM's internal configuration determines the response of the unit to the decoded instruction.

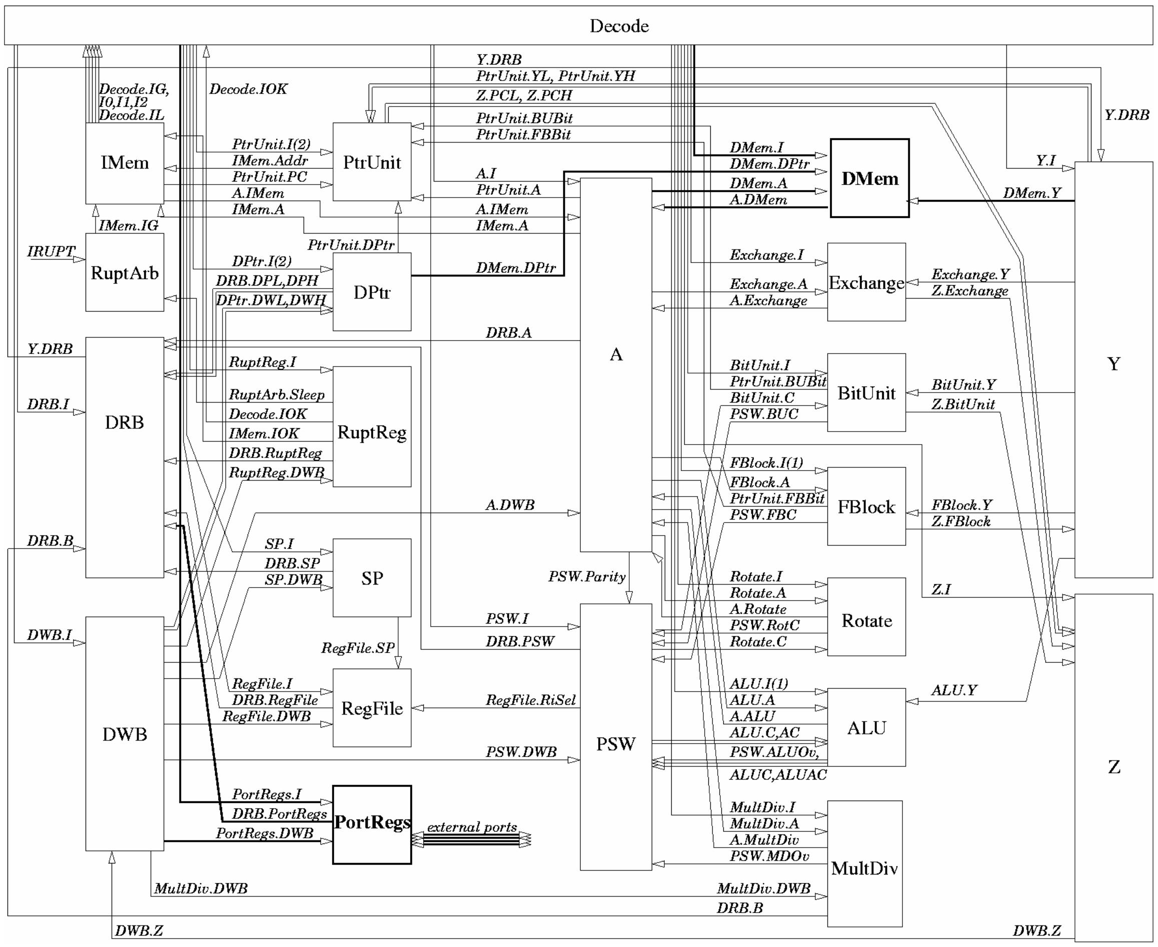


Figure 8: Lutonium block diagram

There are seven instruction channels (abstracted as PortRegs.I and DMem.I in Figure 8) that the decode uses to send the PRDM a command. The primary instruction channel contains the type of command the PRDM needs to execute, such as a port read, port read-modify-write, or a data memory access. The six other channels are conditional; two carry information about the source of a port register read (port number and register, such as P0o), two carry information about the destination of a port register write, and the last two inform the PRDM whether a bus access is a read or a write, and whether to use a 16-bit or an 8-bit address.

For a port register read or write, the PRDM receives and sends its data through the direct read bus (DRB) and the direct write bus (DWB) which route the data to and from other execution units and the Lutonium data registers. For a data memory access, the PRDM has dedicated channels connecting it to the accumulator (A) register, as well as the DPtr register, which is used as the source of a 16-bit address. The PRDM also connects to the Y bus, which is used as the indirect source of an 8-bit address coming from the register file. For more details on the overall Lutonium decomposition see [1] and [11].

5. Design Tradeoffs and Decisions

The PRDM unit's complexity approaches that of a small microprocessor. It has two main tasks: First, mediate access to five I/O ports through 20 SFRs, four per port. Second, perform external memory bus reads and writes through the five I/O ports. The configurability exposed through the DMC register requires the PRDM to have a great deal of control logic implementing the various configuration options.

In starting the design of the PRDM, two main paths of approach presented themselves: First, each subunit of the PRDM could have a great deal of controlling logic built into itself, with only a small combined control unit to sequence the actions. This would result in a faster design, and perhaps a smaller one as well.

Second, all control and decision-making could be placed into one central state machine, making all the other subunits significantly simpler to implement. With this approach, much less complex sequencing circuitry needs to be designed and tested, so the overall design time is less. However, this approach is likely less efficient than the first method, and slower, since all sequencing activity is controlled by a large state machine, with a large cycle time.

In the end, the second approach was chosen as the design approach, for several reasons. First, while the first approach would likely be faster, the speed of the PRDM is not very critical, especially during an external memory bus access, where the access sequence already contains large delays. Most types of accesses to the port registers can be done in one state machine cycle, so the cycle time of the state machine doesn't slow the port register access down. Second, new tools available to the research group made the implementation of a large state machine easy, allowing all the complex decision-making involved in the PRDM to be placed in the state machine. This greatly simplifies the design task for the rest of the circuitry, making it far easier to design and debug. It also speeds up the layout, since more of the decision-making circuitry is placed in the state machine ROM, which can be automatically generated.

6. Final Design

The PRDM went through two design iterations before the final design was completed. The final design has significant improvements over the original, especially in the main state machine.

As shown by the Lutonium block diagram in Figure 8, the PRDM was originally designed as two logically separate units, the Port Register unit representing the port SFRs and the Data Memory unit representing the external memory space. As a consequence, the first PRDM design contained two state machines: One for controlling port register accesses, and one for controlling external memory bus accesses. However, while this split simplified the individual state machine, this design resulted in the duplication of numerous channels, and the design of a large switchbox that routed control between the two state machines. In this design, it took several state machine cycles to read or write a port register, which is expected to be the most common use of the PRDM. The state machines were designed as Moore state machines, with the outputs being associated with the state.

The final design merges these two state machines together, and optimizes the instruction channels between the PRDM and the decode unit. These two optimizations allow the removal of a large, complex switchbox process, and allow most port register accesses to be done in two or less state machine cycles. Figure 9 shows the PRDM state machine diagram. The state machine is a Mealy machine; the outputs of the state machine are associated with the transitions between states, not the states themselves. This approach is more flexible and better matches the message-passing abstraction of the QDI design style.

The PRDM contains 5 unique sub-processes; the block diagram can be seen in Figure 10. The Decode unit seen in the diagram is not actually present in the PRDM; it is a representation of the part of the Lutonium Decode unit which sends out control signals on seven different channels to the PRDM. The Kind channel is the primary control channel, while the other 6 channels are conditional depending on the type of instruction. See Table 1 for the meanings of all the instruction channels.

Most of the control signals are received by the PortControl, the controlling state machine. It sends out commands to the rest of the subunits, and sequences nearly all the PRDM activity. The PortControl also receives the delay done message from the peripheral interface delay unit, allowing it to wait for long periods of time between state machine cycles.

The DataMux unit is the PRDM’s connection to the processor’s data paths. It receives and properly routes data to be written to registers, and the addresses and data to be used for external memory accesses. It sends out values read from registers or from an external memory access.

The DMCRegs unit contains the DMC SFR byte. It has several dedicated channels to the PortControl unit, to which it sends configuration information as needed. The DMCRegs also sends commands to the Delay peripheral unit, setting the delay type and length.

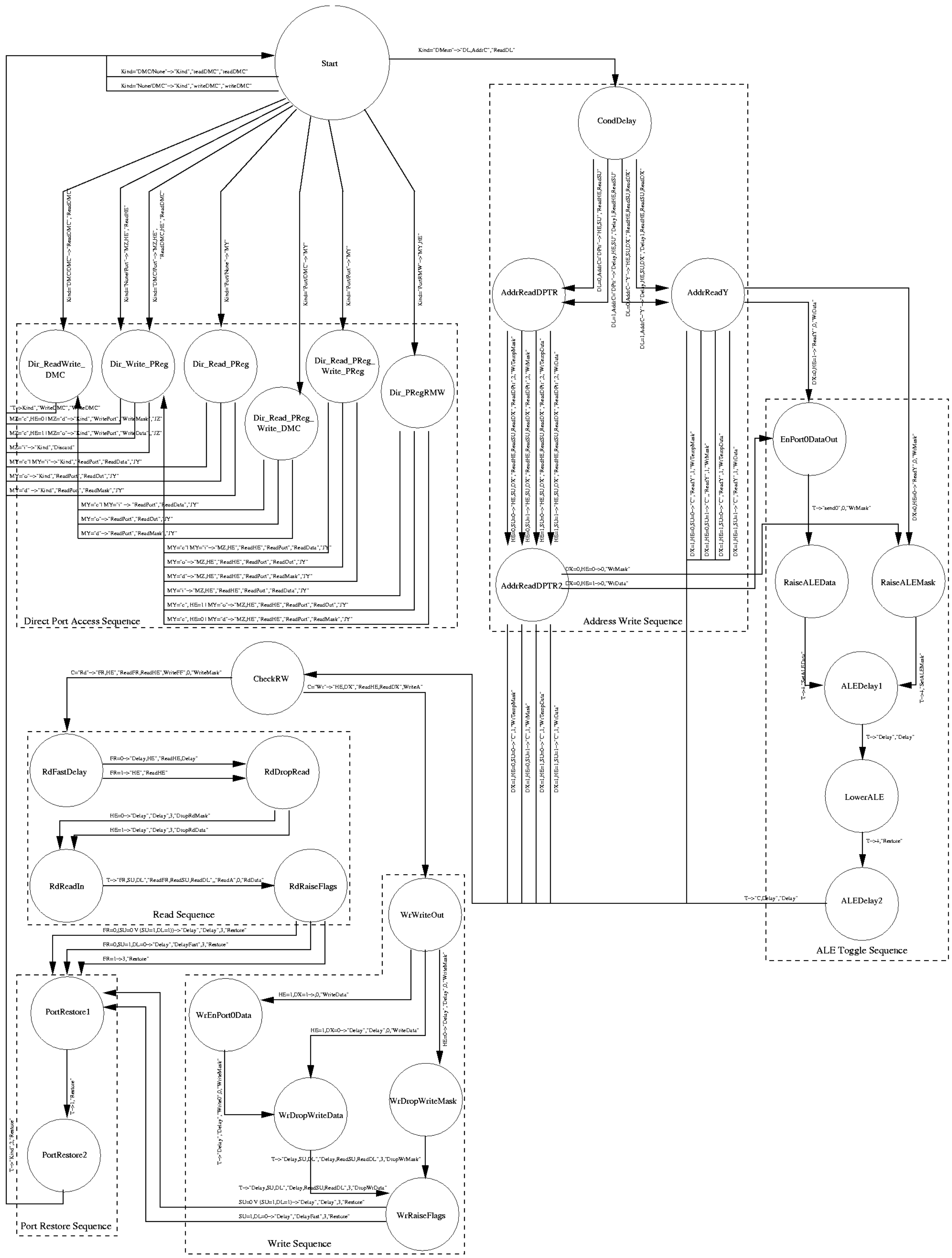


Figure 9: PRDM state machine state diagram

The PortUnitBus routes commands and data to and from the five individual PortUnits. When the PRDM is used for accessing the port SFRs as registers, the PortUnitBus receives the source and destination port numbers directly from the decode unit. During an external memory bus access, the PortControl tells the PortUnitBus which port to route commands and data to.

The five PortUnit processes are all identical. Each interfaces with the matching semi-QDI PortModule process in the peripheral interface. A PortUnit contains two 8-bit registers for storing the output data byte and the output enable byte. The PortUnit is the most complex of the data path subunits, because of the variety of commands it must process.

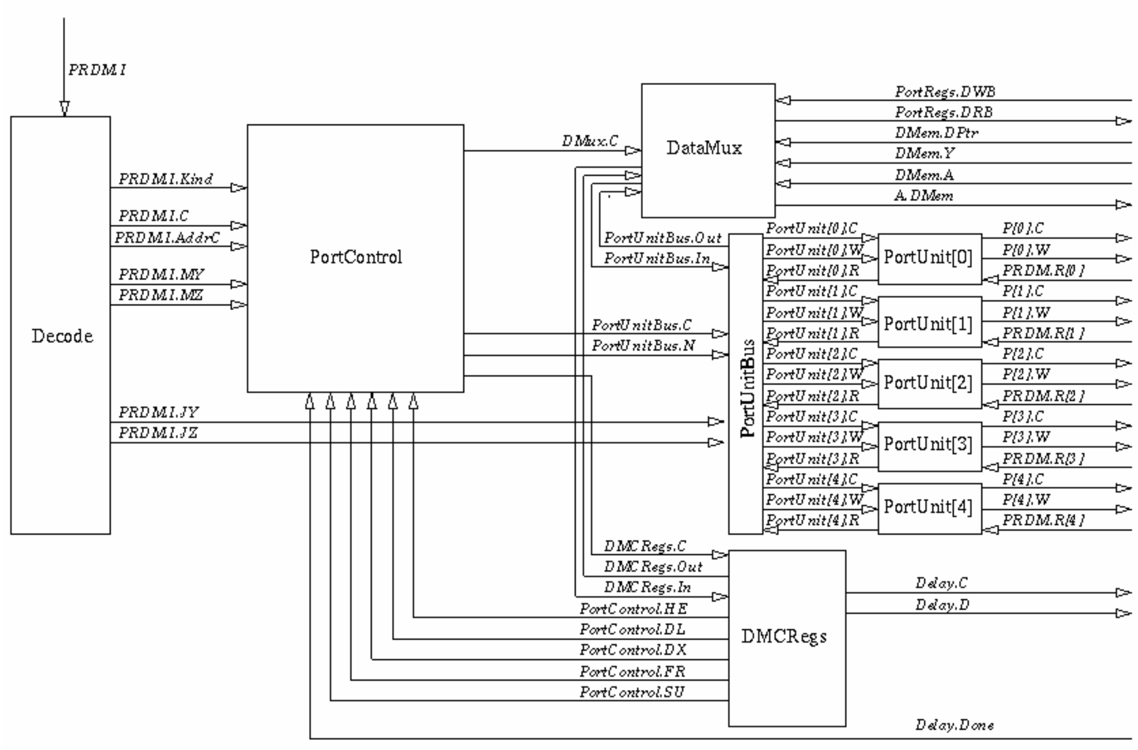


Figure 10: PRDM block diagram

Channel	Contents	When sent	Values
Kind	Primary instruction channel	Always	10 different commands
C	Direction of external memory bus access	For external memory accesses	‘Read’ or ‘Write’
AddrC	Source of external memory bus access address	For external memory accesses	‘DPtr’ (for 16-bit addresses) ‘Y’ (for 8-bit addresses)
MY	Source register for a port register read	For port register reads.	‘c’, ’d’, ’o’, ’i’
MZ	Destination register for a port register write	For port register writes.	‘c’, ’d’, ’o’, ’i’
JY	Source port for a port register read.	For port register reads.	0, 1, 2, 3, 4
JZ	Destination port for a port register write	For port register writes.	0, 1, 2, 3, 4

Table 1: PRDM instruction subchannels

As an example, consider a basic write to Port 0's output data register. First, PortControl receives the command to write a port through the Kind channel. The PortControl needs to then determine which port, and which register of that port, is to be accessed. To determine the register, PortControl requests the value of the HE flag from the DMCRegs unit, and reads in the value of the MZ instruction channel. The HE flag is needed in case the value is 'c', since that port redirects to either 'o' or 'd' depending on the state of the HE flag, as described in section 4.1. In this case, the value of the MZ channel is 'd', so the HE flag is not needed. However, only reading HE when MZ is 'c' would require an extra state, and the energy consumption of the additional state machine cycle far outstrips the savings obtained by sending HE only when needed.

PortControl then sends the appropriate commands to the DataMux and PortUnitBus. In this case, DataMux receives a command to read a data byte from the Lutonium Direct Write Bus (DWB) and to forward it to the PortUnitBus. The PortUnitBus receives the 'write data register' command on its control channel. It also receives a command to read the destination port number from the JZ instruction channel. The PortUnitBus reads the JZ channel, and forwards the command to the appropriate port register unit. It then waits for the data byte from the DataMux process, and forwards it to the PortUnit as well.

The PortUnit receives the instruction and the data byte. It stores the data byte in a local register (where it will be read from in case of a port 0 data register read later), and then sends the data byte along with a 'data write' instruction to the Port 0's driver module, which concludes PRDM's handling of the instruction. PortControl, meanwhile, has reset to its initial state, and is ready to receive a new instruction. Note that due to the pipelining inherent in the design, PortControl can already start processing the next instruction while DataMux, PortUnitBus, and PortUnit0 are finishing the previous instruction.

6.1. PortControl unit

Figure 11 shows the block diagram of the PortControl, the main PRDM state machine. The inputs to the PortControl are all conditional, and most are not used on any given transition of the state machine. The inputs are the various command channels coming from the Lutonium decode process, as well as several of the bits of the DMC register. The outputs of the PortControl are the commands to the other PRDM subunits which compose the PRDM data path. The outputs are conditional as well, since often several subunits are not required to do anything on a given cycle.

The state machine is constructed around a large ROM, which is built using the ROMantic asynchronous ROM generator [10] which can generate transistor networks and layout from a ROM table, and the Rubeg state machine generator, which converts a high-level description of a state machine to a ROM table understandable by Romantic. Both of these tools were developed for the Lutonium by the research group. Both the inputs and outputs of the generated ROM are unconditional; that is, all the inputs and outputs transmit data on each cycle.

A full buffer, the StateBuffer, receives the new state from the ROM and sends it in to the ROM as input on the next state machine cycle. The buffer can also block sending the state until it receives a message from the peripheral Delay unit signifying the completion of a delay period. Through this mechanism, the state machine halts as required for the external memory bus accesses.

While all the ROM's inputs are unconditional, PortControl's inputs are conditional. Therefore, each state machine input has a filter that forwards an actual input for the state machine to the ROM when necessary, and at other times sends a fake value to the ROM.

The input filters are controlled by InputCopy, a small ROM that translates a single command from the main ROM to individual 'Read Input' or 'Generate Fake Value' commands. An important constraint in the design of the main ROM is that it must tell InputCopy which inputs will need to be received on a given state machine cycle during the previous cycle. The ROM must also inform the StateBuffer of an incoming delay on the cycle before as well. This input prediction constraint is one reason why a Mealy state machine is superior to a Moore in terms of the number of transitions required to complete a task. In a Moore machine, all transitions out of a state correspond to the same set of outputs, and therefore must all have the same set of inputs to be read next cycle. If different destination states require different inputs, a Moore machine requires additional states to be added simply for the purpose of preparing the input filters. In a Mealy machine, where each transition can have its own unique outputs, this problem is eliminated. In the PRDM, switching to a Mealy machine structure reduced the final number of state machine states from 63 to 29, and the section dealing specifically with port register access (originally its own state machine) reduced in size from 14 states to 7 states, and the number of state machine cycles required to read or write a single register went from 6 to 2 cycles. The number of cycles required to sequence the most-backward compatible external memory bus read went from 23 cycles to 14 cycles, a significant improvement. See Table 2 for details on the cycle count of various PRDM operations.

The final subunits of the PortControl are the output filters. Since the ROM's outputs are unconditional, even command channels leading to PRDM units that do not need to do anything on a given cycle are sent a command. The output filters remove these spurious commands, and only let the actual commands through to the PRDM data path units.

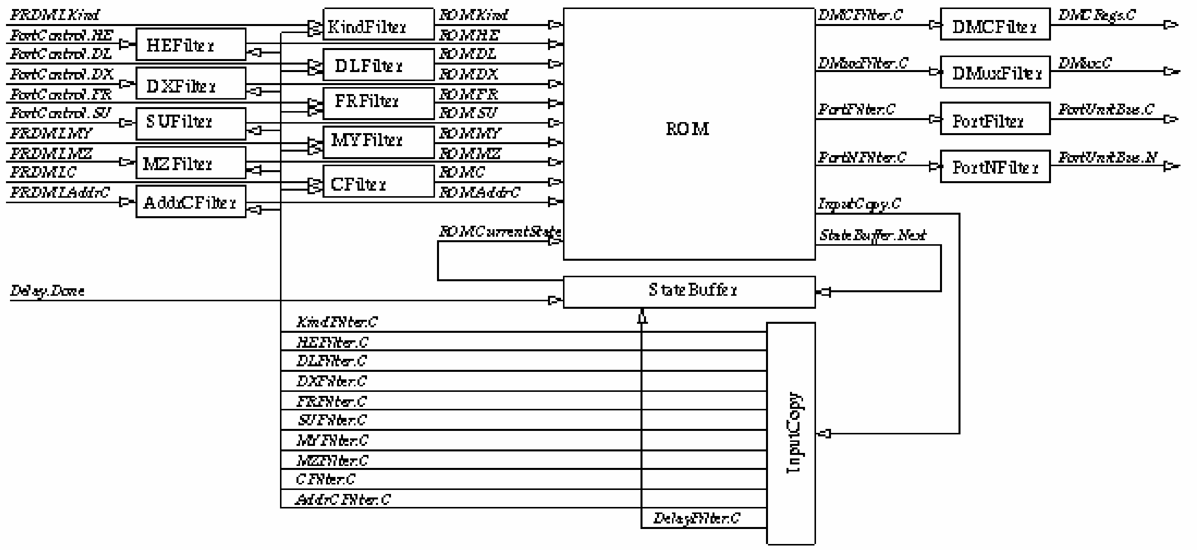


Figure 11: PortControl block diagram

PRDM Operation	State machine transitions
DMC read or write	1
DMC read-modify-write	2
Port register read or write	2
DMC read, port register write	2
Port register read, DMC write	3
Port register read-modify-write	3
Default 16-bit external memory read	15 (subtract one for 8-bit address)
Default 16-bit external memory write	14 (subtract one for 8-bit address)
'Best' 16-bit read (HE, DX, SU, FR)	11 (subtract one for 8-bit address)
'Best' 16-bit write (HE, DX, SU)	11 (subtract one for 8-bit address)

Table 2: State machine transistions for various PRDM operations

6.1. DataMux unit

The DataMux unit contains several split and merge processes that route 8-bit data to and from the rest of the PRDM and the rest of the Lutonium.

The DWBSplit process receives bytes from the Lutonium's direct write bus, and routes them to the DMCRegs or PortUnitBus subunits. The DRBMerge process, analogously, receives bytes from the DMCRegs or the PortUnitBus and sends them to the Lutonium's direct read bus.

The PortUnitBusMerge process receives bytes from DWBSplit and Addr processes in the DataMux, as well as directly from the Lutonium's accumulator, and routes them to the PortUnitBus subunit. In reverse, the PortUnitBusSplit process receives bytes from the PortUnitBus, and sends them to the DRBMerge process, and the accumulator execution unit.

The Addr unit receives addresses to be used for external memory bus accesses. It can either receive a 16-bit address straight from the Lutonium's Data Pointer (DPtr) register, or it can receive an 8-bit address from the main register file indirectly through the Lutonium's Y operand bus. In the case of receiving a 16-bit address, the Addr unit splits it into two separate bytes that are sequentially sent to the PortUnitBusMerge, and from there to the PortUnitBus.

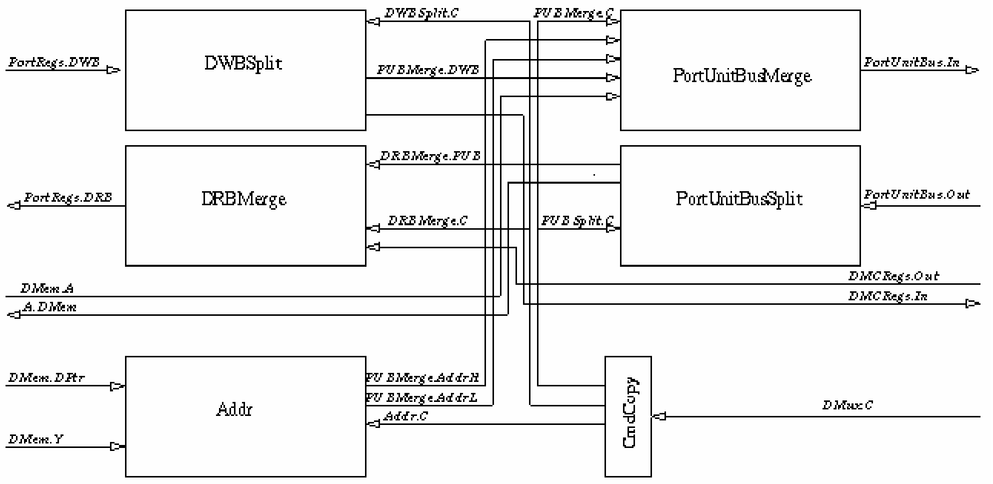


Figure 12: DataMux block diagram

Each process receives its commands through the CmdCopy process, which sends commands only to those subunits that require them, based on the command sent by the PortControl state machine. Figure 12 contains the block diagram of the DataMux unit.

6.2. DMCRegs unit

The DMCRegs unit stores the 8-bit DMC SFR, which is used to configure the PRDM. The PortControl unit accesses several bits of the DMC through dedicated channels, so it was decided to implement the DMCRegs unit as 5 one-bit registers and one 3-bit register (for the delay period). The block diagram for DMCRegs can be seen in Figure 13

The DMCSplit process takes in a byte sent from the DataMux and splits it into 6 values that are sent to the six storage registers. Conversely, the DMCMerge process receives the value of all six registers and composes them together into the complete 1-byte DMC value which is then sent to the DataMux.

HEReg, DXReg, FRReg, and SUReg are all 1-bit registers that can receive a new value from the DMCSplit process, and can send their current value to the DMCMerge process as well as directly to the PortControl state machine.

DLReg, in addition to what the other registers can do, can also sends its value to the peripheral delay unit, to which the value of DLReg acts as a control selecting between the internal delay line and an oscillator signal.

DelayReg contains the 3-bit value specifying the period (in either delay line counts or oscillator ticks) of a standard external access delay. It also contains a small ROM used to decode the 3-bit value into a full byte delay count for both normal and fast read modes. This delay count is then sent to the DelayMerge process.

The DelayMerge process sends a delay count to the peripheral Delay unit in conjunction with the DLReg's control message. Depending on its control message, the DelayMerge can either send the decoded delay count sent to it from the DelayReg, or it can send a 1-tick delay generated by the Delay1Source process. This one tick delay is used to synchronize the beginning of the memory access to coincide with the tick of an oscillator signal.

The DMCCmdBus receives control messages from the PortControl state machine, and forwards them to the appropriate units.

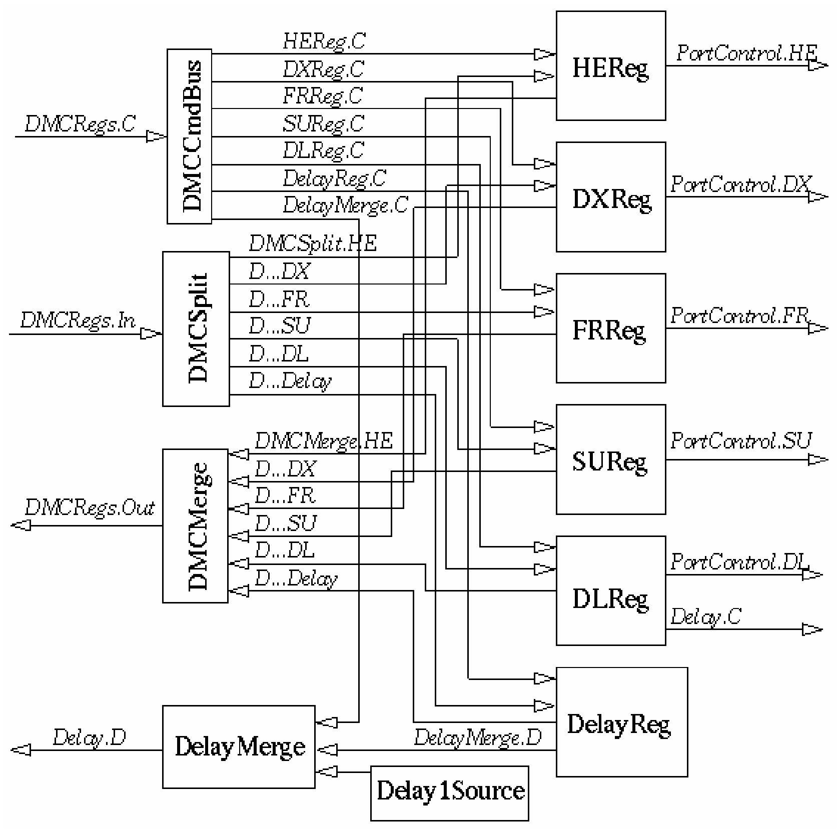


Figure 13: DMCRegs block diagram

6.3. PortUnitBus unit

In the PRDM design, only one port register can be accessed per state machine cycle. While this limitation reduces the speed of the PRDM somewhat, it only affects external memory accesses; for port register accesses, only one port can be read or written at a time in any case. And external memory accesses already contain long wait periods, so a small amount of additional delay is likely not very significant. The primary reason not to implement a full crossbar design, where each port could simultaneously receive data, is the added complexity in the PRDM data paths, and the additional control required to coordinate all the ports at once. See Figure 14 for the unit's block diagram.

The PortUnitBus is the connection between the rest of the PRDM and the five individual port units. It sends and receives byte data from the DataMux, routing it to the proper port unit depending on control from either PortControl or the Lutonium decode unit.

The SelectN process is a merge process which selects one of three inputs to determine the port being accessed for the current cycle. For reads or writes to the port registers, the machine code instruction contains the source or destination port, transmitted from the Lutonium Decode unit to the PortUnitBus on channels JY and JZ, respectively. For external memory bus accesses, the source or destination port is determined by the PortControl state machine. The N channel from the PortControl unit either contains the number of the port to access, or an instruction to read JY or JZ for the port number.

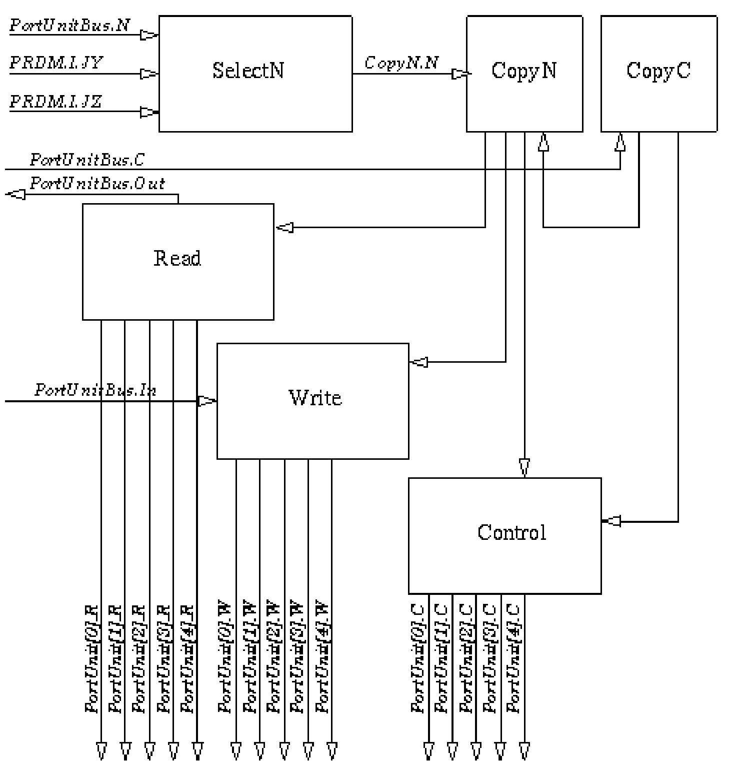


Figure 14: PortUnitBus Block Diagram

The CopyN process sends the number of the port being accessed to the Control process, and possibly the Read or Write processes, depending on the direction of access being performed. Some commands to the port units do not cause data transfer to or from the port unit, in which case only the Control process receives the port number.

The CopyC process receives the instruction to be relayed to the port unit. It sends the instruction to the Control process, and informs the CopyN process whether the instruction is a read or a write, or neither.

The Read process is a byte split process, taking a byte sent from the DataMux and relaying it to one of the five individual port units, based on the port number selected by SelectN. The Write process is the reverse, receiving a data byte from the selected port and forwarding it to the DataMux unit.

The Control unit sends onwards the command the port unit needs to execute, as determined by the PortControl state machine.

6.5. PortUnit

There are five individual port units in the PRDM, each attached through three channels to the actual port drivers of the matching IO port. The port units are identical, except that the mask (output enable) register of Port 4 defaults to all output (00) and all other ports default to all input (FF). Figure 15 has the block diagram of the PortUnit.

The WriteSplit register receives data from the PortUnitBus, and forwards it to the port driver unit. It may also forward the data byte to either the DataReg or the MaskReg unit for storage. When port registers are being accessed directly by the instructions, DataReg and MaskReg are always updated, corresponding to the P*n*d and P*n*o SFRs, respectively. However, during an external memory bus access, several ports' values are temporarily modified (for example, to raise the read or write flag in Port 3), in which case DataReg and MaskReg are left at their old values. When the value of the register needs to be restored to the original value, DataReg and MaskReg can both feed the original data back to the WriteSplit unit, which then forwards it to the port driver. The WriteSplit unit also receives data bytes from the BitFlipper process, which it forwards to the port driver.

The ReadMerge process reads the value of the DataReg, MaskReg, or the value on the port input pins (via the port driver unit) and sends the data onward to the PortUnitBus.

DataReg and MaskReg are both 8-bit registers that receive new values from WriteSplit. Both have three outputs; either to WriteSplit in order to restore the port values to their original state at the end of an external memory access, to ReadMerge, or to the BitFlipper unit.

The BitFlipper unit receives a data byte from either DataReg or MaskReg, and modifies the data byte by raising or lowering a single bit. The BitFlipper is used to activate the read and write flags on P3 of the Lutonium, as well as to raise the ALE pin on P4 of the Lutonium. It reads the MaskReg when the PRDM is running in quasi-bidirectional mode (as the output value is stored in MaskReg in that case), and it reads the DataReg when the PRDM is in full bidirectional mode.

The CCopy unit receives commands from the PortControl via the PortUnitBus unit, and distributes them to the rest of the PortUnit processes. It also generates the proper commands to the port driver unit in the peripheral interface.

SplitSequencer and COutSequencer allow the PortControl to fully restore the value of a port in a single state machine cycle. Restoring a port to its original value requires the write of both the old mask and the old data bytes to the port driver, a simple sequencing action handled by these two units.

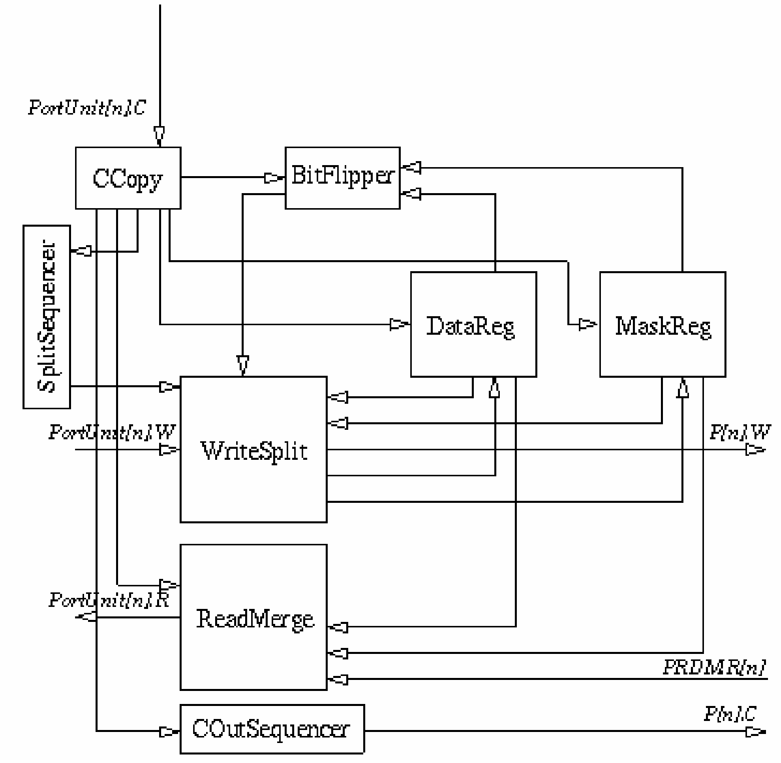


Figure 15: PortUnit block diagram

SplitSequencer sends two commands to the WriteSplit process, one which reads the DataReg and sends it to the port driver, and one which reads the MaskReg and sends it to the port driver. COutSequencer simply sends the port driver a data read and a mask read command in sequence. For all other commands, the two sequencer processes act as simple buffers.

7. Testing

The main test system for the Lutonium is an extensive software simulation suite developed for the project. The system allows sections of the Lutonium to be simulated at various levels of abstraction, starting from top-level descriptions down to actual transistor networks. The simulator runs machine code either compiled from C code or assembly; all of the PRDM test code was written in assembly. Using basic test code and the simulator system, the PRDM was first broken down into its smallest subunits, and then each subunit was converted to a production rule set. This gradual conversion allowed each unit to be tested against a known-working system, speeding up debugging significantly. After the unit was fully converted to a transistor-level design, more rigorous test code was written to fully exercise the design. Three main pieces of test code were utilized in the testing. The first tested every transition in the PRDM state machine diagram, verifying that the outputs of the PRDM were correct. The second tested all the data paths inside the PRDM, writing every possible value through each channel. Finally, the third piece of test code tested every applicable instruction available in the Lutonium with the PRDM, to make sure the instruction decode functioned properly for PRDM instructions. The PRDM design currently passes all the test programs.

Once layout has been completed, automated tools can be used to verify that the circuit as laid out matches the circuit as designed. Therefore, as long as the original design is fully tested, the laid out circuit will be logically correct.

8. Performance Characteristics

While the layout of the PRDM is not yet complete, some estimates of its performance can be extrapolated from the completed transistor net lists and the state machine diagram for the PRDM.

The PortControl state machine contains 29 states. A read or a write of the DMC register takes exactly one state machine transition. A read or a write of a port SFR takes 2 transitions of the state machine. An instruction that both reads and writes an SFR in the PRDM takes between 1 and 3 state transitions to process.

External memory bus accesses take between 9 and 16 transitions to complete, depending on the configuration of the PRDM and the direction of the access. The most-backward compatible read takes 15 transitions for a 16-bit address, and 14 state transitions for an 8-bit address. Correspondingly, the most-backward compatible write takes 14 transitions for a 16-bit address, and 13 for an 8-bit address.

Turning on fully bidirectional ports, and demultiplexed address and data results in smaller transition counts, translating to a smaller energy expenditure. Both reads and writes then take 11 state transitions for a 16-bit address, and 10 for an 8-bit address. Most of the savings in transition counts comes from the removal of the ALE toggle sequence when data and address are separated onto different ports.

Based on transistor-level simulations, a single state machine state transition cycle takes 30 transistor firings (a firing is a transistor changing from on to off or vice versa). While this is slower than the Lutonium core, which has been designed with a cycle time of 22 transitions, this is an acceptable speed, since the PRDM is expected to be a infrequently used execution unit, and an 8051 external memory access is in any case a slow process involving deliberate wait stages.

The entire PRDM contains roughly 40,000 transistors, not including transistors required for stabilizing values on dynamic logic wires (staticizer units). See Table 3 for sub-unit transistor counts.

Simulated energy consumption estimates show that the PRDM consumes roughly 160 pJ for a port register write, 140 pJ for a port register read, and 80 pJ for a DMC SFR read or write.

Subunit	NMOS	PMOS	Total
PortControl	4769	2068	6837
DataMux	1737	1392	3129
DMCRegs	2166	1618	3874
PortUnitBus	1342	1182	2524
PortUnit	2657	2101	4758
Overall	23299	16765	40064

Table 3: PRDM transistor counts

Type of access	Energy usage	Energy/state machine cycle
Port register write	160 pJ	80 pJ / transition
Port register read	140 pJ	70 pJ / transition
DMC register write	80 pJ	80 pJ / transition
DMC register read	80 pJ	80 pJ / transition
Port register read-modify-write (bit op)	250 pJ	83 pJ / transition
Port register read, then write	240 pJ	80 pJ / transition
DMC register read, port register write	175 pJ	88 pJ / transition
Port register read, DMC register write	220 pJ	73 pJ / transition
DMC register read-modify-write	170 pJ	85 pJ / transition
16-bit default memory bus write	1200 pJ	85 pJ / transition
16-bit ‘best’ memory bus write	970 pJ	88 pJ / transition
8-bit default memory bus write	1110 pJ	85 pJ / transition
8-bit ‘best’ memory bus write	880 pJ	88 pJ / transition
16-bit default memory bus read	1295 pJ	86 pJ / transition
16-bit ‘best’ memory bus read	940 pJ	85 pJ / transition
8-bit default memory bus read	1200 pJ	86 pJ / transition
8-bit ‘best’ memory bus read	840 pJ	84 pj / transition

Table 4: PRDM energy consumption

For external memory bus accesses, the default backward-compatible access cycle takes 1200 pJ for a write to the external memory bus. The most energy-efficient mode (8-bit memory, true bidirectional ports, demultiplexed address and data, fast read, sustain on) results in a write cycle consuming 880 pJ. Therefore, a 26% energy savings can be obtained by switching the PRDM to energy-efficient mode, even ignoring the savings from the reduction in external circuitry, lack of an external oscillator, and the removal of the energy-hungry quasi-bidirectional ports. Energy consumption per state machine cycle is roughly constant between all types of accesses, which suggests that the state machine itself is consuming most of the power in the PRDM. See Table 4 for the complete list of energy consumption numbers for the PRDM.

The above numbers do not include any other sections of the Lutonium besides the PRDM (specifically, instruction decode and processor bus energy consumption is not included). Also, the energy consumer by the actual port drivers is also neglected, because it depends on output loading conditions.

The Lutonium's goal is for an average instruction to consume roughly 500 pJ of energy. Clearly, external bus accesses consume significantly more than this, even when the instruction decode and processor buses aren't factored in. This was expected due to the complexity of a memory access cycle and the configurability of the PRDM. The relative infrequency of such memory accesses, however, should minimize the impact of the high energy consumption.

9. Conclusion

The PRDM design is fully functional and tested extensively through software simulators at various levels of abstraction. The final transistor networks of the PRDM are complete, and successfully pass all test programs. The speed and energy consumption of the PRDM are within expected values, and are acceptable given the goals of the Lutonium project. The unit is ready for layout along with the rest of the Lutonium microcontroller, and by all indications it will function correctly when fabricated.

10. References

[1] The Lutonium: A Sub-Nanojoule Asynchronous 8051 Microcontroller. Alain J. Martin, Mika Nyström, Karl Papadantonakis, Paul I. Penzes, Piyush Prakash, Catherine G. Wong, Jonathan Chang, Kevin S. Ko, Benjamin Lee, Elaine Ou, James Pugh, Eino-Ville Talvala, James T. Tong, Ahmet Tura. *9th IEEE International Symposium on Asynchronous Systems & Circuits*, 2003.

[2] 80C51 family programmer's guide and instruction set. Philips Semiconductors, September 1997.

[3] 80C51 family hardware description. Philips Semiconductors, December 1997.

[4] 80C51 family architecture. Philips Semiconductors, March 1995.

[5] The Limitations to Delay-Insensitivity in Asynchronous Circuits. Alain J. Martin. *Sixth MIT Conference on Advanced Research in VLSI*, ed. W.J. Dally, 263-278, MIT Press, 1990.

[6] Pipelined Asynchronous Circuits. Andrew Matthew Lines, *Master's Thesis, California Institute of Technology,* 1995

[7] ET2: A Metric For Time and Energy Efficiency of Computation. Alain J. Martin, Mika Nyström, and Paul Penzes. *Power-Aware Computing*, R.Melhem and R.Graybill ed., Kluwer Academic Publishers, 2001.

[8] Quasi-Delay Insensitive Circuits are Turing-Complete. Rajit Manohar and Alain J. Martin. Invited paper, *Async96 Second International Symposium on Advanced Research in Asynchronous Circuits and Systems,* March 1996.

[9] Synthesis of Asynchronous VLSI Circuits. Alain J. Martin. *Technical Reports, Computer Science Department, California Institute of Technology,* 1991.

[10] ROMantic: Generation and Optimization of Quasi Delay-Insensitive Read-Only Memories. Mika Nyström, Elaine Ou, Alain J. Martin. *Internal Documentation, Asynchronous VLSI Group, Department of Computer Science, California Institute of Technology,* November 2002

[11] 80C51 Decomposition. Karl Papadantonakis. *Internal Documentation, Asynchronous VLSI Group, Department of Computer Science, California Institute of Technology,* April 2003