1 LED Interface

1.1 Purpose of LED Interface

Most Ethernet interfaces have one or more LED's per port to give a user or administrator fast and easy indication of a number of line conditions on each port. In a switch there are two logical places where the LED's could be driven from: from the PHYs, or from the switch chip.

The Strataswitch product line prior to SM-Lite did not support LED status indicators. The assumption was that the PHY itself would support the LED status.

This assumption has been tossed out by more than one vendor, who have requested a specific LED interface. One reason they desire the LED status to come from the switch rather than from the PHYs is that it gives them the ability to change PHYs without worrying about LED compatibility. Another issue is that the gigabit PHYs traditionally haven't had LED interfaces at all.

Unfortunately, each vendor wants an interface very different from all of its competitors. On approach to satisfying their needs is to hardware each interface that is known about at design time, and have a simple register which selects which interface to use. This is inadequate since it precludes satisfying future requirements that are unknown at design time.

Instead, the SML LED interface contains a very small, very simple processor (1500 gates) to do all the appropriate data formatting and interface control. It is vast overkill for the needs of any one customer, but makes it possible to keep everybody happy.

1.2 LED Serial Port

The interface to the LED status indicators is via a serial protocol carried out on two pins: LEDClk, and LEDData. The only mode supported is that which the BCM5228 calls the "low cost" serial interface. If there are ***n*** LED status lights, the interface causes ***n*** clocks, shifting out data out-of-phase with respect to the LEDclk. After all ***n*** bits have been shifted out, the LEDClk and LEDData lines go idle until the next time the LED status is refreshed. Some external shift register is responsible for holding the state of the LED status between scan (refresh) events. Although a given LED may temporarily hold inappropriate state during the scan out, scanning represents less than a 0.1% duty cycle, and is not an issue.

Although the interface presented here is suitable for the "low cost" implementation, there is one feature to facilitate smarter interfaces as well. Just before a new burst of bits, the LEDData signal pulses once. A "low cost" interface will not notice this pulse as it isn't clocked. A smarter interface that desires some type of framing signal can notice the 0→1→0 transition on LEDData without any transitions on LEDClk and can sync on that.

See Figure 1 for a diagram of the scan-out behavior. A simple FPGA or a handful of MSI TTL shift register parts suffice to complete the interface.

See Figure 2 for a diagram of the relationship of LEDClk to LEDData.

Note that the LEDData output changes with the falling edge of LEDClk so that the positive edge of LEDClk can be used to clock LEDData with ample setup and hold margins. LEDClk's period and the frequency of refresh events are tied to the chip's coreclk frequency and are hardwired.

t_{LEDCYC} is nominally 200ns (5 MHz). The high and low times for each phase of the clock is guaranteed to be at least 70 ns. The setup and hold time of LEDData relative to LEDClk's rising edge is a minimum of 50 ns. The drive current for LEDClk and LEDData is 2ma. **If LEDClk must drive many loads, it should be externally buffered.**

Figure 1 -- LEDClk/LEDData Phase Relationship

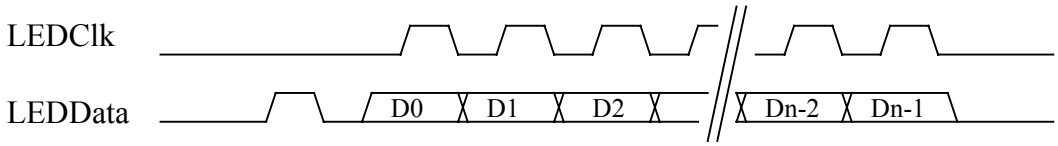
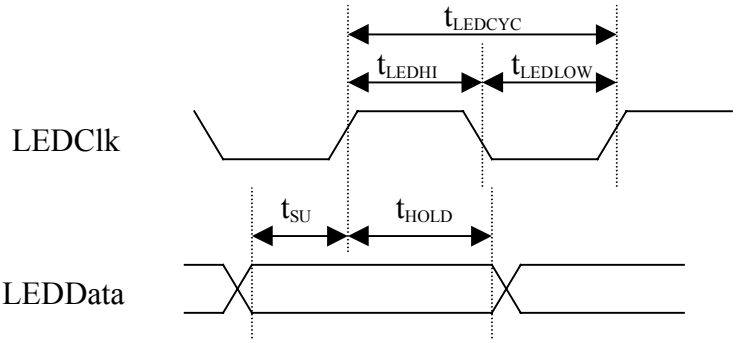
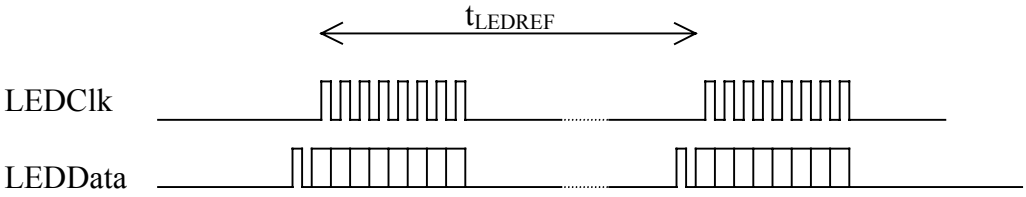


Figure 2 -- LEDCLK/LEDDATA Timing



On a larger timescale, Figure 3 shows how the refresh bursts occur. LEDClk and LEDData are both low during the large gaps between the scanout events. The refresh period is nominally 33 ms (30 Hz), but since it is slaved to the coreclk frequency, this may vary in direct relationship.

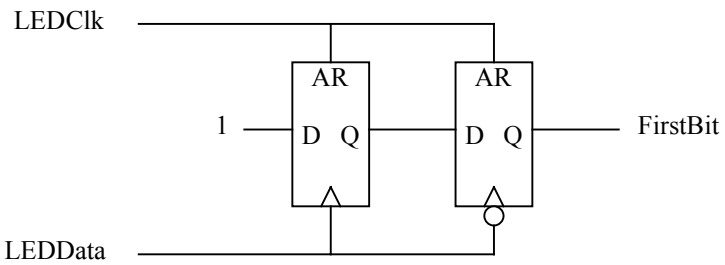
Figure 3 -- LEDClk/LEDData Refresh Interval



On reset, the interface will clock out 300 '0's on the LEDData interface, then it will go idle until the host processor configures the LED controller.

Figure 4 shows an example of how to asynchronously detect the first bit of the LED scan out, if that information is needed. The "FirstBit" signal will be active at the rising edge of LEDClk for the first data bit and inactive at all other rising edges of LEDClk. Of course, this is not the only possible implementation. If a higher-frequency clock is available, a small state machine can look at LEDClk and LEDData and decode this information as well.

Figure 4 -- LEDData Sync Bit Decode



Finally, Table 1 summarizes the timing specifications of the LED serial interface. The cycle time and refresh period parameters will be exactly the same as their nominal rating as long as the chip frequency is 160 MHz, and will vary up and down directly with the core frequency. The hi/low/setup/hold numbers depend on frequency, but on rise and fall times of the signals as well.

Table 1 -- LEDClk/LEDData Timing Specs

parameter	minimum	nominal	maximum
t_{LEDCYC}		200 ns	
t_{LEDHI}	70 ns	100 ns	130 ns
t_{LEDLOW}	70 ns	100 ns	130 ns
t_{SU}	50 ns	90 ns	
t_{HOLD}	50 ns	90 ns	
t_{LEDREF}		30 ms	

1.3 LED Processor

The LED interface is a combination of hardwired control and programmable formatting. Hardwired logic takes care of dividing down the core frequency to produce an approximately 30 Hz LED refresh rate. The period is tied directly to the core frequency, so if the chip is run at a lower frequency, the LED refresh rate will drop in direct proportion.

Each LED refresh period, the following chain of events takes place:

1) Hardware polls each of the ports to gather the most recent status.

2) This status is stored into the first 64 bytes of processor data RAM.

3) The processor is woken up and starts running the user program, starting at address 0x00.

4) The program inspects the status bits collected from all the ports. It reformats the data in any way it chooses (including perhaps inspecting status bits located elsewhere in memory that have been established by a host processor). It constructs the serial scan-out stream bit by bit and stores it into bytes 64-95 of data RAM.

5) The program indicates how many bits need to be shifted out, and halts.

6) Hardware shifts out '*n*' bits (the number specified in step 5) on the LEDClk/LEDData serial interface, obeying all the timing constraints.

7) The unit goes to sleep until the next LED refresh period begins.

The following sections provide more detail on the processor architecture and instruction set.

1.3.1 Host Interface

Before the LED processor can do any work, the host processor must initialize the program RAM for the LED processor and then enable it to run.

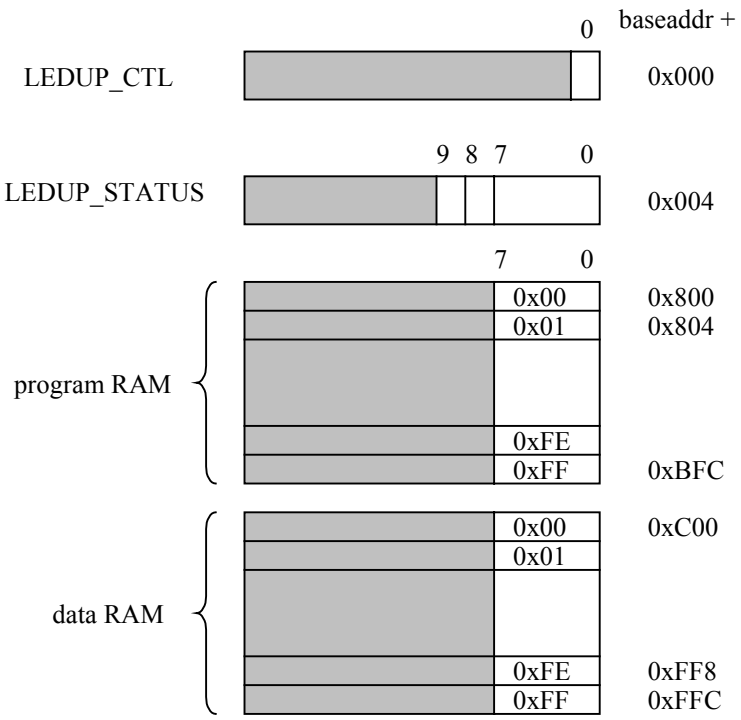
Figure 5 shows the registers of the LED processor visible to the host processor.

LEDUP_CTL contains only a single bit. On reset, this register is set to 0. When this bit is zero, the LED processor is prevented from running. After initializing the program RAM, this bit should be set to 1 to enable the control program to run.

The LEDUP_STATUS register is read-only. The 8 ls bits contain the current program counter of the LED processor; it may be useful for diagnosing if something has gone wrong with the program. Bit 8 is a flag that indicates whether the processor is running (1) or not (0). Bit 9 is a flag that indicates whether the processor is initializing (1) or not (0); this is active during the time when the processor initially shifts out 1000 '0' bits on the LED serial interface.

The LED processor has 256 bytes of program RAM, and 256 bytes of data RAM. The host is free to read or write either of these RAMs at any time. Note that each read or write involves only one byte of data. The LED processor is designed to stall for one cycle while the host makes the access. However, it is recommended that the host processor disable the LED processor while any changes are made, then reenable the LED processor when all changes are finished. This advice is simply to keep complexity to a minimum.

Figure 5 -- LED processor host interface



1.3.2 Processor Architecture

The LED processor is an 8-bit processor in all regards: the registers are 8 bits wide, as are the memory, the PC, and all addresses. The program and data space are separate, and are each 256 bytes deep. Thus, although the LED processor and read and write any byte of the data RAM, the program RAM can be modified only by the host interface.

1.3.2.1 Program Space

The LED processor has 256 bytes of program RAM. The LED processor can only fetch instructions from this RAM; it can't read nor modify the program state in any way. Only the host interface is capable of modifying the program RAM.

There are only two special addresses in the program RAM: 0x00 and 0xFE. At each LED refresh period, the processor wakes up and begins executing from address 0x00. The call/return address stack has limited depth. If the processor makes more `CALL`s than `RET`s, the older call addresses fall off the bottom of the stack. If the processor makes more `RET`s than the depth of the stack, it ends up branching to 0xFE. This should not be used as a "feature" of the processor; rather, by placing a "`JMP 0xFE`" instruction at that address, the programmer can catch "runaway" programs.

1.3.2.2 Data Space

The LED processor has 256 bytes of data RAM. Any byte of data RAM can be

read or written by the LED processor and by the host interface. The data space has some locations that have designated uses. Table 2 lists these addresses. Although architecturally there is space for 32 ports, only 26 ports (ports 0 to 25) are used.

Table 2 – Designated Data RAM locations

Address range	Use
0x00-0x01	status bits for port 0
0x02-0x03	status bits for port 1
…	…
0x3E-0x3F	status bits for port 31
0x40-0x5F	LED scan chain assembly area
0x60-0xFF	undesignated

As the table shows, the status bits for a given port are packed into two bytes. These bytes are refreshed at the LED refresh rate immediately before the LED processor starts running. Table 3 shows what each bit of the two bytes of a given port represent.

Table 3 – Per-port Status Bits

Bit	Meaning	Status=0	Status=1
0	RX	no frames received	frames received
1	TX	no frames transmitted	frames transmitted
2	Collision	no collisions	collisions have occurred
[4:3]	Speed	00=10 Mbps, 01=100 Mbps, 10=1000 Mbps	
5	Duplex	half duplex	full duplex
6	Flow Control	no pause handshake	pause handshake successful
7	Link Up	link down	link up
8	Link Enabled	link disabled	link enabled
9-13	unused		
14	False	always 0	
15	True	always 1	

1.3.2.3 Register Set

The processor has two registers, which are completely symmetric in their use: the "A" and "B" registers. These are what most 8b processors call the accumulator registers. They are frequently the source or destination operand of instructions. They are useful in keeping the program size small in that they can be encoded more compactly then a general data address. The A and B registers can also be used as memory indexing registers, depending on the address mode used by an instruction.

The processor maintains two status bits: the C bit and the Z bit. The C, or "carry," flag is set when an ALU operation results in ALU overflow. The Z flag is set when the result of some ALU computation has a result of 0x00. Each instruction, documented later, specifies whether it affects the C and Z flags.

Finally, there is a four deep, one bit wide "T" stack. This boolean stack can be used to efficiently build up the serial LED data stream. It is explained in more detail later.

1.3.2.4 Address Modes

The processor supports a few addressing modes. Due to a lack of instruction bits in which to encode addressing modes (a problem with all 8b processors), not all instructions allow all addressing modes. In the following table, we use the same nomenclature as the assembler to indicate the address mode.

Table 4 -- Address Modes

Address Mode	Nomenclature examples	The instruction refers to:
Register	`A` `B`	the contents of the `A` or `B` register

Immediate	`3`
	`0x21`
	`label`
	`+(3<<2)+1`
		an immediate 8b constant value

Indirect	`(A)` `(B)`	the data RAM location pointed to by the contents of the `A` or `B`

Absolute	`(3)`
	`(0x21)`
	`(label)`
	`((3<<2)+1)`
		the data RAM location pointed to by the immediate 8b constant value

A few examples may help illustrate these modes better.

Table 5 -- Address Mode Examples

Example	Explanation
`LD A,B`	The contents of the `B` register are copied to the `A` register
`LD A,3`	The immediate value 3 is copied to the `A` register
`LD B,(0x21)`	Data RAM location 0x21 is copied to the `B` register
`LD (0x21),B`	Data RAM location 0x21 is written with the value of the `B` register
`LD A,(phase)`	The symbolic label "phase" represents an 8b constant that selects which data RAM location supplies the new value of the `A` register
`LD (A),077`	The data RAM location indexed by the contents of register `A` is overwritten with the 8b octal constant 077 (63 decimal).
`JMP 0x30`	The program continues fetching instructions from address 0x30.

Note that any expression that begins with an opening parenthesis is interpreted as being an absolute or indirect address mode. Immediate constant expressions that must be parenthesized can start with a unary + or some other idempotent operation to skirt the issue.

1.3.2.5 Subroutines

Unlike real microprocessors, the LED processor doesn't maintain a call/return stack in data memory. Instead, there is a two-deep return address stack that is kept in registers.

Every time a `CALL` is made, the address of the next consecutive instruction is pushed on this return address stack and program execution continues with the immediate address supplied by the instruction. When the new address is pushed on the stack, the oldest entry on the stack disappears. There is no type of exception hardware that detects if more than two `CALL`s are made without a corresponding `RET`.

When a `RET` instruction is later executed, the top return address is popped off the stack and becomes the new PC (program counter). The bottom return address value is loaded with 0xFE. The intent is that this minor feature would allow a programmer to set a trap at 0xFE to catch programming errors where too many `RET` instructions were performed.

1.3.2.6 Instruction Set

The instruction set provides the usual set of arithmetic, logical, bit twiddling, and branching constructs. In addition, a handful of specialized instructions and the "`T`" stack can be used to efficiently access the port status bits and to build the serial LED bit string in memory.

All instructions are either one or two bytes long. All instructions take six coreclk cycles to execute, even if the instruction is two bytes, even if the instruction has to access data memory twice for that instruction. The instruction set is symmetric with respect to the A and B registers. Immediates are symmetric with A and B as long as it makes sense (that is, an immediate can not be a destination). Any instruction that accepts one of "(A)", "(B)", or "(#)" accepts the other modes as well (as long as the encoding results in fewer than three bytes).

Table 8 and Table 9 supply enough information that any programmer familiar with typical assembly language can figure out the function of the typical instructions. Looking at the examples later on will also help clarify their use.

However, the LED-specific operations need further explanation. Here too, examining the example code presented later will help. Almost all of these instructions involve the boolean `T` stack. This stack is one bit wide and four words deep. The instructions below supply a simple way to extract status bits, combine them, and pack them into bytes ready to shift out the LED serial port. Only the `SEND` operation is essential; all the other instructions are merely conveniences.

1.3.2.6.1 `PORT` and `PUSHST` Instructions

As Section 1.3.2.2 explained, the first 64 bytes of the data RAM is subdivided into 32 pairs of bytes. Each pair of byte is automatically loaded with the most recent status gathered from each of the ethernet ports.

PORT takes one argument, which specifies which of the 32 logical ethernet ports is to be inspected. Although there are logically 32 ports, this version of the chip only scans ports 0 to 25; the status bytes for ports 26-31 are undefined. This instruction in-and-of-itself does nothing, but the value supplied is retained and used by the PUSHST instruction.

PUSHST takes one argument; the four least significant bits of the argument specify which of the 16 bits of the two port status bytes is to be extracted. The extracted bit is then pushed on the top of the T stack.

For example, the sequence

```
PORT      3
PUSHST    2
```

causes bit 2 of data RAM location 0x06 to be tested and pushed on the T stack. Although this example uses a constant to specify which port to test, most programs would probably use a variable to set the port and increment this variable as part of a loop.

It is quite possible to access a given bit of a given port via the more common LD and TST instructions. PORT and PUSHST just make it a lot easier. The above sequence could have been coded as:

```
LD      A,3      ; or whatever port specifier you want
ADD     A,A      ; double it - two status bytes per port
TST     (A),2    ; extract bit 2 of data RAM location A
PUSH    CY       ; save the bit on the T stack
```

1.3.2.6.2 PUSH Instruction

This instruction takes one argument. The least significant bit of the argument is extracted and pushed on the T stack. As a special case, the C (carry) flag can be pushed on the T stack with the instruction

```
PUSH    CY
```

1.3.2.6.3 POP Instruction

This instruction removes the top entry of the T stack and copies it into the C (carry) flag. It can be used for futher conditional tests, or it can be useful for simply removing the top of the stack.

For instance, to duplicate the top of stack, the following code sequence can be used:

```
POP
PUSH    CY
PUSH    CY
```

1.3.2.6.4 TAND, TOR, TXOR, INV Instructions

TAND, TOR, and TXOR instructions remove the top two entries from the T stack, perform the specified boolean operation on the two bits, then push the resulting bit back on the stack.

The TINV instruction simply complements the value on the top of the T stack.

All of these instructions are useful for combining status bits to achieve compound status indications. For instance, if an LED is supposed to go active whenever a port has received or transmitted packets, the following code sequence will do the job:

```
PUSHST  RX      ; extract RX bit from current port
PUSHST  TX      ; extract TX bit from current port
TOR             ; replace those two bits with logical OR
```

1.3.2.6.5 PACK Instruction

While the previous instructions were concerned with extracting bits, this operation builds the output bit string that drives the serial LED port. Recall that when the processor finishes, hardware serializes the bit stream starting at bit 0 of location 0x40 in the data RAM and clocks it out on the serial LED interface port.

Every time the hardware begins an LED refresh cycle, just before it wakes up the processor, there is buried state associated with this instruction that is initialized to point to bit 0 of byte 0x40 in the data RAM.

Every time a PACK instruction is issued, the top of the T stack is popped off and stored to the bit location indicated by the buried state; the buried state is then updated to point to the next bit, taking care of byte boundary crossings as required.

Note that the buried bit pointer state is inaccessible. This means that programs that use this instruction must build the bit string in the exact order that it will appear on the serial LED line.

It is possible to entirely ignore this instruction; the hardware will simply shift out whatever bits it finds starting at location 0x40. This instruction could be roughly mimicked by the following few generic instructions:

(at reset)

```
LD      A,0x40
LD      (packbyte),A
SUB     A,A
LD      (packbit),A
```

(then at each place where PACK would have been used)

```
    LD      A,(packbyte) ; get byte pointer
    LD      B,(packbit)  ; get bit pointer
    POP                  ; extract top of T stack into C flag
    BIT     (A),B        ; copy C flag into specified bit
    INC     B            ; point at next bit
    AND     B,7          ; mod 8
    JNZ     UPDPTR
    INC     (packbyte)   ; bump the byte pointer
UPDPTR:
    LD      (packbit),B
```

1.3.2.6.6 SEND Instruction

The SEND instruction takes one argument, which specifies how many bits there are in the LED scan chain. A value of 0x00 means 0 bits, not 256 bits. This is the only

LED-specific instruction that can't be emulated through other means.

When this instruction is performed, the processor halts until the next LED refresh cycle. Hardware then extracts bits starting at bit 0 of data RAM location 0x40 and shifts them out the LED serial port. Since the maximum length LED chain is 255 bits, this requires 32 bytes: RAM 0x40 to 0x5F. However, if the LED scan chain is shorter, the bytes at end of this range can be safely used for any purpose so desired by the programmer.

1.3.2.7 Opcode Maps

The following two tables, Table 6 and Table 7, show all valid opcodes. Empty entries in the tables are illegal opcodes and should not be used. Executing such opcodes results in undefined behavior. All opcodes that contain either "#" or "(#)" as an argument require two bytes; all other instructions are encoded in one byte.

There are four instructions, highlighted in red, which are logically meaningful, but which are not permitted because it would require two immediate values, resulting in a three-byte long instruction.

The mnemonic convention is that if there are two operands for an instruction, the destination is on the left. For example:

```
ADD A,B
```

means to add registers `A` and `B` together, placing the results into register `A`.

Table 6 – Opcodes 0x00 to 0x7F

ms nibble	ls nibble							
	0	1	2	3	4	5	6	7
0	LD A,A	LD A,B	LD A,#		LD A,(A)	LD A,(B)	LD A,(#)	CLC
1	LD B,A	LD B,B	LD B,#		LD B,(A)	LD B,(B)	LD B,(#)	STC
2	PUSH A	PUSH B	PUSH #		PUSH (A)	PUSH (B)	PUSH (#)	PUSH CY
3	PUSHST A	PUSHST B	PUSHST #					CMC
4	LD (A),A	LD (A),B	LD (A),#		LD (A),(A)	LD (A),(B)	LD (A),(#)	
5	LD (B),A	LD (B),B	LD (B),#		LD (B),(A)	LD (B),(B)	LD (B),(#)	RET
6	LD (#),A	LD (#),B	LD (#),#		LD (#),(A)	LD (#),(B)	LD (#),(#)	CALL #
7	JZ #	JC #	JT #		JNZ #	JNC #	JNT #	JMP #
8	INC A	INC B			INC (A)	INC (B)	INC (#)	PACK
9	DEC A	DEC B			DEC (A)	DEC (B)	DEC (#)	POP
A	XOR A,A	XOR A,B	XOR A,#		XOR A,(A)	XOR A,(B)	XOR A,(#)	TXOR
B	OR A,A	OR A,B	OR A,#		OR A,(A)	OR A,(B)	OR A,(#)	TOR
C	AND A,A	AND A,B	AND A,#		AND A,(A)	AND A,(B)	AND A,(#)	TAND
D	CMP A,A	CMP A,B	CMP A,#		CMP A,(A)	CMP A,(B)	CMP A,(#)	TINV
E	SUB A,A	SUB A,B	SUB A,#		SUB A,(A)	SUB A,(B)	SUB A,(#)	
F	ADD A,A	ADD A,B	ADD A,#		ADD A,(A)	ADD A,(B)	ADD A,(#)	

Table 7 – Opcodes 0x80 to 0xFF

ms nibble	ls nibble							
	8	9	A	B	C	D	E	F
0	TST A,A	TST A,B	TST A,#		BIT A,A	BIT A,B	BIT A,#	
1	TST B,A	TST B,B	TST B,#		BIT B,A	BIT B,B	BIT B,#	
2	PORT A	PORT B	PORT #		PORT (A)	PORT (B)	PORT (#)	
3	SEND A	SEND B	SEND #		SEND (A)	SEND (B)	SEND (#)	
4	TST (A),A	TST (A),B	TST (A),#		BIT (A),A	BIT (A),B	BIT (A),#	
5	TST (B),A	TST (B),B	TST (B),#		BIT (B),A	BIT (B),B	BIT (B),#	
6	TST (#),A	TST (#),B	TST (#),#		BIT (#),A	BIT (#),B	BIT (#),#	
7								
8	ROL A	ROL B			ROL (A)	ROL (B)	ROL (#)	
9	ROR A	ROR B			ROR (A)	ROR (B)	ROR (#)	
A	XOR B,A	XOR B,B	XOR B,#		XOR B,(A)	XOR B,(B)	XOR B,(#)	
B	OR B,A	OR B,B	OR B,#		OR B,(A)	OR B,(B)	OR B,(#)	
C	AND B,A	AND B,B	AND B,#		AND B,(A)	AND B,(B)	AND B,(#)	
D	CMP B,A	CMP B,B	CMP B,#		CMP B,(A)	CMP B,(B)	CMP B,(#)	
E	SUB B,A	SUB B,B	SUB B,#		SUB B,(A)	SUB B,(B)	SUB B,(#)	
F	ADD B,A	ADD B,B	ADD B,#		ADD B,(A)	ADD B,(B)	ADD B,(#)	

Table 8 is an alphabetic listing of the instructions and their encodings, and includes which flags are affected, and offers a brief instruction description. To understand the field encodings used in Table 8, please refer to Table 9.

Table 8 -- Instruction Opcode Encoding

encoding	op	dst	src	flags	explanation
typical uP data operations					
0ddd0sss	LD	DDD	SSS		DDD ← SSS
1111dsss	ADD	D	SSS	cy,z	D ← D + SSS
1110dsss	SUB	D	SSS	cy,z	D ← D – SSS
1101dsss	CMP	D	SSS	cy,z	null ← D - SSS (only flags affected)
1100dsss	AND	D	SSS	z	D ← D & SSS
1011dsss	OR	D	SSS	z	D ← D | SSS
1010dsss	XOR	D	SSS	z	D ← D ^ SSS
10000ddd	INC	DDD		cy,z	DDD ← DDD + 1
10010ddd	DEC	DDD		cy,z	DDD ← DDD – 1
10001ddd	ROL	DDD		cy,z	DDD ← { DDD[6:0], DDD[7] }; CY ← DDD[7]
10011ddd	ROR	DDD		cy,z	DDD ← { DDD[0], DDD[7:1] }; CY ← DDD[0]
0ddd11ss	BIT	DDD	SS	z	set bit SS[2:0] of DDD with value of CY
0ddd10ss	TST	DDD	SS	cy	set CY with bit SS[2:0] of DDD
00000111	CLC			cy	CY ← 0
00010111	STC			cy	CY ← 1
00110111	CMC			cy	CY ← !CY
branching operations					
01110ccc	J{CCC}	#			if CCC, PC ← #
01110111	JMP	#			PC ← #
01100111	CALL	#			RET ← PC+2, PC ← #
01010111	RET	#			PC ← RET
LED-specific operations					
00101sss	PORT		SSS		set PORT addressing state with SSS
001100ss	PUSHST		SS		push state bit MEM[R_PORT,SS[3]] bit SS[2:0]
00100sss	PUSH		SSS		push lsb of SSS onto state stack; SSS==7 means push CY bit
10010111	POP			cy	CY ← top of state stack; pop stack
11000111	TAND				ANDs top two bits of T stack boolean stack manipulation
10110111	TOR				ORs top two bits of T stack
10100111	TXOR				XORs top two bits of T stack
11010111	TINV				inverts top bit of T stack
10000111	PACK				pops top of stack to output buffer
00111sss	SEND		SSS		shift out SSS bits on LED port and halt

Table 9 -- Instruction Field Encoding

Field Key	0	1	2	3	4	5	6	7
SSS	A	B	#		(A)	(B)	(#)	
SS	A	B	#					
DDD	A	B			(A)	(B)	(#)	
D	A	B						
CCC*	Z	C	T		NZ	NC	NT	

* JT and JNT test the top of the T stack

1.4 LED Processor Tools

1.4.1 Assembler

A simple assembler, *ledasm*, has been written to help generate the binary which to be loaded into the program RAM. Because the programs are rather small and simple, the assembler does not support advanced features of typical macro assemblers. If macro capabilities are desired, the user should use cpp, m4, or a similar program to get the desired capability. `ledasm` does not support linking.

`ledasm` does support a few niceties. Comments begin with a semicolon. Symbolic labels are supported, up to 31 characters. Constant expressions can be formed using a number of operators, following the precedence rules of the C language.

The input to the assembler is a simple ASCII text file. The assembler produces two output files: a listing file, and a binary hex file. The hex file consists of 16 lines of 16 bytes, where each byte is encoded as a pair of uppercase ASCII hex digits, with a space before hex pair.

1.4.2 Disassembler

The hex files produced by the ledasm, or any hex file in the same format, can be disassembled with leddasm. In and of itself, the program probably doesn't have tremendous utility. However, the disassembler was written in two pieces: a standalone one instruction disassembler that could be used as part of other tools, and a driver program the reads the hex file and calls the one-line disassembler for each instruction.

1.4.3 Simulator

A simple simulation environment can read ledasm hex files and either run through or single step through the instruction stream, allowing the LED programs to be debugged before entering into a system environment. There are commands to inspect memory, load memory from a file, turn tracing on and off, disassemble a range of addresses, single step, continue, reset, and run.

1.5 Example Programs

The following few sections contain five example programs, each one progressively more complicated than the previous one. Although you get a medal if you read all the way through to example 5, reading the first two should give you a good idea of how the instruction set works. Pay close attention to the special instructions for the LED status manipulation: `PORT`, `PUSHST`, `PACK`, `TAND`, `TOR`, `TINV`, and `SEND`.

Warning: although these programs assemble, they have not been tested to work. Use them as suggestive examples, not gospel.

1.5.1 ex1.asm

The following program is about as simple as it gets. The LED status scan chain consists of 26 LEDs, with the status for port 0 scanned out first, with port 25 last. There is only one LED per port. The LED for a given port is driven to 1 (on, presumably) if a port as any receive, transmit, or collision activity during an LED refresh interval.

The program is 23 bytes in length and uses only one extra data location for storage. Note how the "`PORT A`" instruction allows extracting the specified bits relative to a given port. Note how the `PUSHST`, `TOR`, and `PACK` operations work in concert. Note the use of the symbolic labels RX, TX, and COLL make the use of the `PUSHST` instructions more obvious.

```
;-------------------------- start of program -------------------------

begin:
      ld      A,0           ; start with port 0
      ld      (portnum),A

portloop:
      port    A             ; specify which port we're dealing with
      pushst  RX            ; push RX activity status bit on T stack
      pushst  TX            ; push TX activity status bit on T stack
      tor                   ; RX | TX
      pushst  COLL          ; push COLL activity status bit on T stack
      tor                   ; RX | TX | COLL
      pack                  ; pop top of T stack to output buffer

      inc     (portnum)
      ld      A,(portnum)
      cmp     A,26
      jnz     portloop

      send    26            ; send 26 LED pulses and halt


; data storage
portnum equ   0x7F          ; temp to hold which port we're working on


; symbolic labels
; this gives symbolic names to the various bits of the port status fields

RX    equ     0x0           ; received packet
TX    equ     0x1           ; transmitted packet
COLL  equ     0x2           ; collision indicator

;-------------------------- end of program -------------------------
```

1.5.2 ex2.asm

This example is slightly different than Example 1. In this case, each port has three LEDs, one for each of the receive, transmit, and collision activity bits. The data is still scanned out from port 0 to port 25, with RX, TX, then COLL scanned out for port n before the status for port n+1 is scanned out.

The program is 24 bytes long.

```
;-------------------------- start of program -------------------------

begin:
      ld      A,0           ; start with port 0
      ld      (portnum),A

portloop:
      port    A             ; specify which port we're dealing with
      pushst  RX
      pack                  ; send to output buffer
      pushst  TX
      pack                  ; send to output buffer
      pushst  COLL
      pack                  ; send to output buffer

      inc     (portnum)
      ld      A,(portnum)
      cmp     A,26
      jnz     portloop

      send    26*3          ; send 78 LED pulses and halt


; data storage
portnum equ   0x7F          ; temp to hold which port we're working on


; symbolic labels
; this gives symbolic names to the various bits of the port status fields

RX    equ     0x0           ; received packet
TX    equ     0x1           ; transmitted packet
COLL  equ     0x2           ; collision indicator

;-------------------------- end of program -------------------------
```

1.5.3 ex3.asm

This example is a variation on Example 2. The same status bits are presented on three LEDs per port, but the scan order is different. Instead of shifting out all the status for a given port before going on to the next port, we shift out all the receive status bits, then all the transmit activity bits, then all the collision activity bits.

This program is 30 bytes long, and is only somewhat more complicated than Example 2. It also shows the use of subroutines.

```
;-------------------------- start of program -------------------------

begin:
      ld      B, RX
      call    portsub

      ld      B, TX
      call    portsub

      ld      B, COLL
      call    portsub

      send    26*3          ; send 78 LED pulses and halt


portsub:
      ld      A,0           ; start with port 0
      ld      (portnum),A

portloop:
      port    A             ; specify which port we're dealing with
      pushst  B             ; extract given status bit
      pack                  ; send to output buffer

      inc     (portnum)
      ld      A,(portnum)
      cmp     A,26
      jnz     portloop

      ret                   ; return from subroutine


; data storage
portnum equ   0x7F          ; temp to hold which port we're working on


; symbolic labels
; this gives symbolic names to the various bits of the port status fields

RX    equ     0x0           ; received packet
TX    equ     0x1           ; transmitted packet
COLL  equ     0x2           ; collision indicator


;-------------------------- end of program -------------------------
```

1.5.4 ex4.asm

Example four is considerably more complicated than any of the earlier examples. The first two LED status bits are the same as before: receive and transmit activity. However, the third LED is a bicolor LED: it is red, green, or orange depending how the two diodes in the physical LED package are driven. Four bits are required per port to drive the three LEDs. Another factor making this program complicated is that the bicolor LED has a number of states depending on the link status and speed. Finally, the detecting high/low speed for ports 24&25 (the gig ports) requires different handling from detecting the link speed on ports 0-23 (10/100 ports).

Note that the (phase) data in the data RAM lives from invocation to invocation of the program, and is used to set the LED blink duty cycle.

This program is 62 bytes long.

```
;The situation is this:
;
;    26 ports (port 0-25) each have three LEDs.
;          led 0: RX
;          led 1: TX
;          led 2: speed
;             black if disabled
;             blinking amber if  10 Mbps but link down or disabled
;             solid on amber if  10 Mbps and link up
;             blinking green if 100 Mbps but link down or disabled
;             solid on green if 100 Mbps and link up
;                [ ports 24 and 25 are the gig ports and are slightly
;                  different; they are green for 1G, and amber for
;                  anything less, either 10Mbps or 100Mbps ].
;
;    Bicolor LEDs are really two LEDs with two inputs: with one LED
;    of the pair driven, the LED is green; with the other one driven,
;    the LED is red, with both driven, it is amber (yellow).
;
;    The scan out order is RX, TX, inp1, inp2 for port 0, then
;    the same for port 1, and so on through port 25.  Note that
;    ports 24 and 25 need slightly different handling than the
;    other ports.
;
;
;-------------------------- start of program --------------------------

begin:
      ; update refresh phase within 1 Hz
      ld      A, phase
      inc     A             ; next phase
      cmp     A, 30         ; see if 1 sec has gone by
      jc      chkblink
      sub     A, A          ; A = 0

chkblink:
      ld      (phase),A
      sub     B, B          ; B = 0
      cmp     A, 15         ; A already has the phase
```

```
      jnc     noblink
      inc     B             ; B = 1
noblink:
      ld      (blink),B

      sub     A,A           ; start with port 0
      ld      (portnum),A

portloop:
      port    A             ; specify which port we're dealing with
      pushst  RX            ; first put out RX status
      pack                  ; send to output buffer

      pushst  TX            ; next put out RX status
      pack                  ; send to output buffer

      ; here's the tricky part.  to encode bicolor LED speed status,
      ; we do the following:
      ;    inp1 inp2
      ;     0    0    = off   = link disabled
      ;     1    0    = green = full speed
      ;     1    1    = amber = less than full speed
      ;     0    1    = red   = not used

      pushst  LINKEN        ; link enabled

      push    (blink)       ; lsb of (blink) is pushed
      tand                  ; turn off LED if blinking
      pop                   ; cy = LINKEN & blink
      push    cy            ; put it back
      pack                  ; inp1

      push    cy            ; tos = LINKEN & blink
      ld      A,(portnum)
      cmp     A,24
      jnc     dogigs
      pushst  SPEED_C       ; test for 100Mbps speed
      jmp     wrapup

; ports 24 and 25 are the gigabit ports
dogigs:
      pushst  SPEED_M       ; test for 1000Mbps speed

wrapup:
      tinv                  ; make it a test for not full speed
      tand                  ; combine with LINKEN & blink
      pack                  ; inp2

      inc     (portnum)
      ld      A,(portnum)
      cmp     A,26
      jnz     portloop

      send    104           ; send 26*4 LED pulses and halt


; data storage
```

```
blink equ     0x7D          ; 1 if blink on, 0 if blink off

phase equ     0x7E          ; phase within 30 Hz
                            ; should be initialized to 0 on reset

portnum equ   0x7F          ; temp to hold which port we're working on


; symbolic labels
; this gives symbolic names to the various bits of the port status
fields

RX            equ           0x0     ; received packet
TX            equ           0x1     ; transmitted packet
COLL          equ           0x2     ; collision indicator
SPEED_C       equ           0x3     ;  100 Mbps
SPEED_M       equ           0x4     ; 1000 Mbps
DUPLEX        equ           0x5     ; half/full duplex
FLOW          equ           0x6     ; flow control capable
LINKUP        equ           0x7     ; link down/up status
LINKEN        equ           0x8     ; link disabled/enabled status
ZERO          equ           0xE     ; always 0
ONE           equ           0xF     ; always 1

;--------------------------- end of program --------------------------
```

1.5.5 ex5.asm

The fifth, and final, example is the most complicated of all. The host processor places some status bits into data RAM, which control the display of the LED status. Read the comment embedded in the start of the program listing below for the details.

Note the incorporation of the "link aggregation" status bits that are updated by the host processor and are incorporated into the LED stream as appropriate.

This program is 130 bytes long when assembled.

```
;Here is an example program for programming the LEDs on the 5605.
;
;The situation is this:
;
;    26 ports (port 0-25) each have one LED.  The front panel
;    has an 8-position switch so the user can select which
;    status is displayed on the front panel.  The host processor
;    reads the state of the switch periodically and communicates
;    it to this processor in data location 0x7F.
;
;          0x7F=0: link
;          0x7F=1: display collision activity
;          0x7F=2: display TX activity
;          0x7F=3: display RX activity
;          0x7F=4: duplex
;          0x7F=5: speed (1=full speed, 0=lower speed)
;          0x7F=6: flow control
;          0x7F=7: link aggregation
;
;    when link status is chosen, it has four different indications:
;
;          link disabled: LED is off
;          link enabled but down: LED blinks on approx 100 ms, off 900ms
;          link enabled and up: LED is on
;
;    link aggregation indicates which ports are part of a trunk group.
;    this information isn't known to the MACs, and so isn't directly
;    known to this processor.  instead, the host CPU sends a string
;    of 32 (well, 26 meaningful) bits to addresses 0x78 to 0x7B to
;    indicate which ports should have their LED set in this case.  port 0
;    corresponds to bit 0 of 0x78, port 1 is bit 1 of 0x78, etc.
;
;    The scan out order is port 25 to port 0.
;
;    Note that the speed indicator requires special handling because
;    the gig ports (ports 24 and 25) run at higher speed.
;
;
;-------------------------- start of program -------------------------

begin:
      ; update refresh phase within 1 Hz
      ld      A, phase
      inc     A             ; next phase
      cmp     A, 30         ; see if 1 sec has gone by
```

```
      jc      chkblink
      sub     A, A          ; A = 0
chkblink:
      ld      (phase),A
      sub     B, B          ; B = 0
      cmp     A, 3          ; 3 of 30 cycles = 10% duty cycle
      jnc     noblink
      inc     B             ; B = 1
noblink:
      ld      (blink),B

      ld      A,25          ; start with port 25
      ld      (portnum),A

      ; later we need to scan a 32 bit trunkmap.  we store the
      ; address in the form of byte & bit offset explicitly.
      sub     A,A
      ld      (mapbit),A    ; bit 0
      ld      A,trunkmap
      ld      (mapbyte),A   ; starting at byte "trunkmap"

portloop:
      port    A             ; specify which port we're dealing with

      ld      A,(select)
      dec     A
      jc      sel0
      dec     A
      jc      sel1
      dec     A
      jc      sel2
      dec     A
      jc      sel3
      dec     A
      jc      sel4
      dec     A
      jc      sel5
      dec     A
      jc      sel6
      dec     A
      jc      sel7
      ld      B,ZERO
      jmp     simple        ; turn off all LEDs

sel0: ; link status: complex
      pushst  LINKEN
      jnt     wrapup        ; LED is off if disabled
      pushst  LINKUP
      jt      wrapup
      ; port is enabled but the link is down: blink
      push    (blink)       ; push it to status stack
      jmp     wrapup
sel1: ld      B,COLL
      jmp     simple
sel2: ld      B,TX
      jmp     simple
sel3: ld      B,RX
```

```
      jmp     simple
sel4: ld      B,DUPLEX
      jmp     simple
sel5: ; speed is a bit trickier
      ld      B,SPEED_C     ; assume 100 Mbps
      ld      A,(portnum)
      cmp     A,24          ; ports 24&25 are 1000 Mbps full speed
      jc      simple
      ld      B,SPEED_M     ; inc B would work too
      jmp     simple
sel6: ld      B,FLOW
      jmp     simple
sel7: ; link aggregation
      ld      A,(mapbyte)   ; get byte address we're looking for
      ld      B,(mapbit)
      tst     (A),B         ; cy = B'th bit of (A)
      push    CY            ; push it on led status stack
      jmp     wrapup

simple:       ; handle simple case where we test just one bit
      pushst  B             ; test B'th status bit

wrapup: pack                ; send bit to LED buffer

      ; bump trunk map pointers (used only if select==7)
      inc     B
      cmp     B,8
      jc      bump
      inc     (mapbyte)
      sub     B,B
bump: ld      (mapbit),B

      ; bump port pointer -- note we are scanning 25 to 0
      dec     (portnum)
      jnc     portloop

      send    26            ; send 26 LED pulses and halt


; data storage
blink    equ  0x70          ; 1 if blink on, 0 if blink off

phase    equ  0x71          ; phase within 30 Hz
                            ; should be initialized to 0 on reset

portnum  equ  0x72          ; temp to hold which port we're working on

mapbyte  equ  0x73          ; byte pointer to trunkmap
mapbit   equ  0x74          ; bit  pointer to trunkmap
trunkmap equ  0x78          ; a bitmap of four bytes

select   equ  0x7F          ; indicates which status we are displaying


; symbolic labels
; this gives symbolic names to the various bits of the port status
fields
```

```
RX              equ             0x0     ; received packet
TX              equ             0x1     ; transmitted packet
COLL            equ             0x2     ; collision indicator
SPEED_C         equ             0x3     ;  100 Mbps
SPEED_M         equ             0x4     ; 1000 Mbps
DUPLEX          equ             0x5     ; half/full duplex
FLOW            equ             0x6     ; flow control capable
LINKUP          equ             0x7     ; link down/up status
LINKEN          equ             0x8     ; link disabled/enabled status
ZERO            equ             0xE     ; always 0
ONE             equ             0xF     ; always 1

;--------------------------- end of program --------------------------
```