Light52 -- free, open source MCS51 compatible CPU core

Revision 6 - February 3, 2013 Core Datasheet

OVERVIEW

Light52 is a free, small open-source CPU core compatible to the Intel MCS51 architecture.

While the core is within the performance envelope of other free MCS51 cores, the implementation trades area for speed.

This core is smaller than most free and commercial MCS51 cores and its speed is comparable to that of a 6-clocker -- see sections 8 and 9.

The full original MCS51 instruction set is implemented with the possible exception of the BCD opcodes (DA and XCHD) which are optional.

FEATURES

- 100% binary compatible to MCS51 (except possibly for optional BCD instructions).
- Speed comparable to a 6-clocker.
- Configurable through VHDL generics.
- Smaller than most other cores.
- Includes 16-bit timer, UART and I/O ports.
- Additional peripherals and SFRs can be added easily.
- 256 bytes of IRAM -- fixed size.
- Fully synthesizable, static synchronous design with positive edge clocking and no internal tri-states.

Light52 lacks some features usually present in other MCS51 cores and has some important limitations:

SHORTCOMINGS

- No access to off-chip memory.
- Strictly Harvard: XDATA and XCODE spaces can't be merged into a Von Neumann architecture.
- From 2 to 8 clocks per instruction.
- Far slower than most commercial cores: performance/area ratio is worse even though area is much smaller.
- No On-Chip Debugging capability.

1.- Pinout

Table 1: ***Core Signal Pinout***

Signal	Direction	Description
clk	input	Clock, active on rising edge.
reset	input	Active high synchronous reset.
rxd	input	RxD input for on-board UART.
txd	output	TxD output from on-board UART.
external_irq[7..0]	input	High-level-sensitive interrupt inputs
p0_out[7..0]	output	Port P0 8-bit output.
p1_out[7..0]	output	Port P1 8-bit output.
p2_in[7..0]	input	Port P2 8-bit input.
p3_in[7..0]	input	Port P3 8-bit input.

2.- Functional Description

Since the MCS51 architecture is already well documented elsewhere, this datasheet will only deal with those aspects of the core which depart from the original.

In this version of the core, there is no support for shared XCODE/XDATA memory spaces (the core performs simultaneous accesses to XCODE and XDATA and there is no wait state or access arbitration logic yet). The MCU memory model is therefore strictly Harvard.

The peripherals included in the MCU core are generally not compatible to the MS51 peripherals and are somewhat less flexible -- the core trades programmability in run-time for configurability in synthesis time. See section 7 below for a detailed description of available peripherals.

Existing MCS51 programs will generally NOT work unmodified on this core -- code needs to be ported to the available peripherals and their SFRs like it needs to be in any other MCS51 derivative.

Interrupt operation is identical to the original, except of course for the unimplemented peripherals whose interrupt request lines have been left unconnected and are available for user designs.

3.- Special Function Registers

Table 2 lists the SFRs implemented in the current version of the core.

Table 2: ***Light52 Special Function Registers***

Symbol	Description	Direct Address	Bit Address and Symbol								Reset Value
ACC	Accumulator	E0H	E7	E6	E5	E4	E3	E2	E1	E0	00H
B	B register	F0H	F7	F6	F5	F4	F3	F2	F1	F0	00H
DPH	DPTR high	83H									00H
DPL	DPTR low	82H									00H
IE	IRQ Enable	A8H	AF	AE	AD	AC	AB	AA	A9	A8	00H
			EA	-	-	ES	-	-	ET0	EX0	
IP	IRQ Priority	B8H	BF	BE	BD	BC	BB	BA	B9	B8	00H
			-	-	-	PS	-	-	PT0	PX0	
PSW	Program	D0H	D7	D6	D5	D4	D3	D2	D1	D0	00H
	Status Word		CY	AC	F0	RS1	RS0	OV	-	P	
SP	Stack Pointer	81H									07H
P0	Port 0 outp.	80H	87	86	85	84	83	82	81	80	00H
P1	Port 1 outp.	90H	97	96	95	94	93	92	91	90	00H
P2	Port 2 inp.	A0H	A7	A6	A5	A4	A3	A2	A1	A0	
P3	Port 3 inp.	B0H	B7	B6	B5	B4	B3	B2	B1	B0	
T0CON	Timer 0	88H	8F	8E	8D	8C	8B	8A	89	88	00H
	Control		-	-	T0CEN	T0ARL	-	-	-	T0IRQ	
T0L		8CH									00H
T0H		8DH									00H
T0CL		8EH									FFH
T0CH		8FH									FFH
SCON	UART	98H	9F	9E	9D	9C	9B	9A	99	98	00H
	Control		-	-	RxRdy	TxRdy	-	-	RxIrq	TxIrq	
SBUF	Data Buffer	99H									
SBPL	Baud Rate L	9AH	Initialized as per generics UART_BAUD_RATE and UART_CLOCK_RATE								(*1)
SBPH	Baud Rate H	9BH	Initialized as per generics UART_BAUD_RATE and UART_CLOCK_RATE								(*1)
EXTINT	External	C0H	C7	C6	C5	C4	C3	C2	C1	C0	00H
	IRQ Flags		EIRQ7	EIRQ6	EIRQ5	EIRQ4	EIRQ3	EIRQ2	EIRQ1	EIRQ0	

Notes

1 Only if generic UART_HARDWIRED is false, and then write only.

4.- Interrupt Vectors

Interrupt management is identical to the original MCS51. The only difference is that the five available interrupt request inputs are connected to different sources:

Table 3: ***Interrupt Vectors and Sources***

IRQ	Source	Vector	Priority	8051 equivalent
0	External IRQ	0003h	(highest)	IE0
1	Timer 0	000Bh		TF0
2	Unassigned	0013h		IE1
3	Unassigned	001Bh		TF1
4	UART	0023h	(lowest)	RI+TI

Registers IE and IP and the interrupt priority mechanics work exactly as in the original MCS51, and so does instruction RETI.

5.- Object Code Initialization

This core does not have access to external, off-chip memory: the program code is stored internally within the MCU module. And the XCODE memory space cannot be 'merged' with the XDATA space as it can be done in many MCS51 derivatives -- the MCU is *strictly Harvard*. Thus, the XCODE can only be initialized at synthesis time.

The XCODE ROM is initialized at synthesis time with the contents of VHDL generic **OBJ_CODE**, which is expected to be defined in a package named **obj_code_pkg**.

This package must be generated separately for each project and can be considered part of the program application rather than part of the core source -- because it includes the program object code in VHDL format.

The light52 project has adopted the convention that the package **obj_code_pkg** must be defined in a vhdl file placed within the MCS51 program directories; that is, separate from the rest of the VHDL source files -- for this purpose, the **obj_code_pkg** package can be considered as just another object code format.

This way, the object code for different projects using this core (or for the different code samples within this project) can be neatly separated from the core sources.

The project includes a Python script (directory **/tools/build_rom**) which can be used to produce a suitable **obj_code_pkg** package file from an Intel-HEX object file. The code samples in directory **/test** contain usage examples for this script (**makefile**s and/or BAT files **build.bat**).

While the method chosen for object code initialization is clean and vendor-independent, it has a major drawback: The object code must be available at synthesis time, and every time the code changes the synthesis has to be re-run. This may be a big handicap in certain applications.

Subsequent versions of the core may provide the option to use memory initialization files so that the XCODE memory can be initialized post-synthesis.

Included in the same package as the object code are two constants, XCODE_SIZE and XDATA_SIZE. These constants are set manually in the makefile of each project (only C projects, see the Dhrystone demo for an example) and then are included in the invocation to the Python script. These constants are meant to be assigned to the MCU generics XCODE_ROM_SIZE and XDATA_RAM_SIZE, respectively, which determine the size of the XCODE and XDATA memories.

If you use SDCC, you can make the linker check the object code and XRAM usage against these bounds (again, see the demo makefiles for an example of this).

Thus, you can manually set the ROM and RAM size for your project with the confidence that the final application will work using the smallest amount of FPGA memory.

6.- Configuration Generics

Some of the core features can be configured through VHDL generics:

Table 4: ***Core Configuration Generics***

Generic	Default	Description
CODE_ROM_SIZE	2048	Size of XCODE ROM in bytes. Must be >= 512.
XDATA_RAM_SIZE	0	Size of XDATA RAM in bytes.
OBJ_CODE	(dummy)	Object code to be placed on ROM. See previous section.
USE_BRAM_FOR_XRAM (*1)	false	Use extra BRAM as XDATA RAM.
IMPLEMENT_BCD_INSTRUCTIONS	false	True to implement DA and XCHD, false to execute them as NOPs.
SEQUENTIAL_MULTIPLIER (*1)	false	Use sequential multiplier instead of combinational.
UART_HARDWIRED	true	True to hardwire UART baud rate, false to make it configurable at run time.
CLOCK_RATE	5,00E+007	Clock rate (in Hz) assumed by UART and Timer initialization constants.
UART_BAUD_RATE	19200	Default baud rate for UART. Must be <= CLOCK_RATE / 16.
TIMER0_COUNT_RATE	50000	Count frequency of Timer0 (configures T0 prescaler).

Notes

1 Unimplemented, will cause a synthesis assertion failure if given a non-default value.

Object Code and Memory Generics

These generics are meant to be initialized with the constants defined in package **obj_code_pkg** (see previous section). This will guarantee that the RAM requirements for the application are met without any waste of FPGA resources.

Setting XDATA_RAM_SIZE to 0 will remove the XRAM entirely for those projects that don't need it.

The actual size of the synthesized XDATA RAM will of course be a multiple of the underlying BRAM size for the FPGA architecture, so you are advised to choose the value of the generic accordingly.

The XCODE memory can obviously not be removed (remember this core has no access to external memory) and its minimum size has been arbitrarily set to 512 bytes.

Both XCODE and XDATA are limited to the 'natural' addressing range of 64KB. This core does not support any memory banking scheme.

7.- Peripheral Modules

The MCU core includes a number of peripheral modules. These peripherals have been designed hastily in order to provide a working environment for the CPU -- they do not have their own separate test bench, for example.

The current version of the MCU ships with a simple, hardwired UART, a 16-bit timer and four 8-bit input/output ports. This is enough for a few demos and maybe for some simple applications.

For real projects, though, the MCU is meant to be modified with new peripherals tailored to the application. For this purpose, the SFR bus of the CPU has been made accessible within the MCU entity.

SFR Interface of the CPU Module

Table 5: ***CPU SFR Interface Signals***

Signal	Direction	Description
sfr_addr	output	SFR space address, valid when **sfr_vma** is high.
sfr_vma	output	Asserted high when a SFR read/write cycle is started.
sfr_rd	input	Read data. Must be valid by the end of the same cycle **sfr_vma** is asserted.
sfr_wr	output	Write data. Valid when **sfr_vma** is high AND **sfr_we** is high.
sfr_we	output	Asserted high (with **sfr_vma**) for write cycles.

Some of the SFRs are 'internal' cpu SFRs that are, well, internally decoded by the CPU module: ***ACC***, ***B***, ***DPH***, ***DPL***, ***IE***, ***IP***, ***PSW*** and ***SP*** -- they are grouped together in table 2.

All SFR read and write cycles take a single clock cycle. The SFRs must behave as an *asynchronous* memory when seen from the CPU; the SFR input multiplexor implemented outside the CPU module must be combinational.

Bit-Addressable SFRs

SFR addresses which are multiple of 8 are bit-addressable. This is exactly the same behavior as the original 8051. There's nothing special to bit-addressable SFRs other than their address.

In order to make a custom SFR bit-addressable, all you have to do is decode it at an address multiple of 8, the core will do the rest.

7.1.- UART

The light52 UART is a simplified version of the original MCS51 serial port.

Some of the operational parameters of the UART are hardwired and non-configurable in the current version, not even at synthesis time:

1. Number of stop bits hardwired to 1.
2. Parity hardwired to NONE.
3. Number of bits per character hardwired to 8.

Besides, the 9-bit mode of the original MCS51, whith its applications in inter-MCU communication, is unimplemented yet.

Serial port interrupts work the same as in the original serial port (same vector IRQ4 and same interrupt enable flag IE.ES).

The UART core has some limited capability to recover from errors, described in the VHDL source file **light52_uart** and very similar to that of the original UART:

1. Error conditions such as bad start and stop bits are detected and cause the UART to discard the received byte and wait for the next start bit.
2. Bit sampling mismatches are detected and the bit values are decided by majority.

A follow-up version of this core will include flags for those detected errors, as well as TX and RX overruns.

Since all operational parameters are hardwired except possibly the baud rate, the UART setup is easy: set the baud rate by writing to registers SBPL and SBPH and enable interrupts by setting flag IE.ES -- the UART can be operated in polling mode too if desired.

Register SCON

This register reflects the status of the serial port:

	7	6	5	4	3	2	1	0
SCON	0	0	RxRdy	TxRdy	0	0	RxIrq	TxIrq
reset	0	0	0	0	0	0	0	0
	h	h	r	r	h	h	w1c	w1c

Bits marked 'h' are hardwired and can't be modified.

Bits marked 'r' are read only; they are set and reset by the core.

Bits marked 'w1c' (write 1 to clear) are set by the core when an interrupt has been triggered and must be cleared by the software by writing a 1.

TxRdy	Ready to transmit. High when there's no transmission in progress. It is cleared when data is written to SBUF and is raised at the same time a TX interrupt is triggered.
RxRdy	Received data ready. High when there's data in the RX buffer. Raised at the same time the RX interrupt is triggered, cleared when SBUF is read.
RxIrq	RX interrupt pending service. Raised when the RX interrupt is triggered, cleared when 1 is written to it.
TxIrq	TX interrupt pending service. Raised when the TX interrupt is triggered, cleared when 1 is written to it.

When writing to the status/control registers, only flags **TxIrq** and **RxIrq** are affected, and only when writing a '1' as explained above. All other flags are read-only.

Interrupt flags are triggered at the following moments:

TxIrq	Last clock cycle of the TX stop bit.
RxIrq	At 11/16ths of the RX stop bit, only if the stop bit is valid.

Register SBUF

This is the read/write buffer of the serial port. It gives the software access to the 1-byte-deep receive and transmit buffers. These buffers work like the original MCS51 serial port.

	7	6	5	4	3	2	1	0
SBUF	UART Tx/Rx buffer register							
reset	x	x	x	x	x	x	x	x
	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

Bits marked 'r/w' can be read and written to by the CPU and can be updated by the core.

SBUF	Writing to this register will trigger a serial port transmission unless SCON.TxRdy=0. Reading this register will give the last byte received by the UART, if any.

Writing to this register will trigger a transmission unless there is already a transmission going on (flag SCON.TxRdy=0). In which case the last write access will be ignored. there is no overrun flag to signal this event; the user must prevent it from happening.

When a byte is received, the core raises flag SCON.RxRdy=1. If a new byte is received before the last one has been read (i.e. with flag SCON.RxRdy=1), the receive buffer register will be overwritten with the new data. Again, there is no indication that this has happened; the user must make sure to prevent these overruns.

Reading from this register when flag SCON.RxRdy=1 will clear the flag and return the last received byte.

Reading from this register when flag SCON.RxRdy=0 will return undefined data (usually the last received byte but this may change in later versions).

Note that reading SBUF does NOT clear the RxIrq flag. The flag must be cleared explicitly.

Registers SBPH and SBPL

If generic UART_HARDWIRED is set to false, then the UART implements these two write-only registers.

These registers should be loaded with the baud period measured in clock cycles -- no prescaling involved:

BIT_PERIOD = UART_CLOCK_RATE / UART_BAUD_RATE

The bit period register is the 13-bit wide combination of SBPH and SBPL, with the 3 higher bits of SBPH being ignored.

Note that these registers are write only: reading from their addresses will return an indeterminate value (actually, the value of the SCON register). This saves logic and is hardly an inconvenience for the programmer, which will seldom have to read these registers.

These registers are loaded at reset with their default value, defined by generics UART_BAUD_RATE and UART_CLOCK_RATE, according to the same formula above.

When the generic UART_HARDWIRED is set to true, these registers are hardwired to their default value and writing to them has no effect.

Note that the UART is totally independent of the timer and indeed of any other module, unlike the original MCS51.

7.2.- Timer 0

Basic timer, not directly compatible to any of the original MCS51 timers. This timer is totally independent of the UART.

This is essentially a reloadable 16-bit up-counter that optionally triggers an interrupt every time the count reaches a certain value.

Timer Registers

The timer includes 3 registers:

1. A configurable prescaler register of up to 31 bits.
2. A 16-bit compare register accessible through T0CL and T0CH.
3. A 16-bit counter register accessible through T0L and T0H.

Reading T0L or T0H will give the value of the timer register. If the registers are read while the count is enabled, the software has to deal with a possibly inconsistent (T0L,T0H) pair and should apply the usual tricks -- majority vote, etc.

The prescaler is reset to 0 when T0CEN=0.

When T0CEN=1 it counts up to (TIMER0_PRESCALER - 1), then rolls over to 0 and the timer register is incremented.

TIMER0_PRESCALER is a VHDL generic configurable at synthesis time.

The compare register is write-only, in order to save logic. Reading T0CH or T0CL will give the value of T0CON instead.

Timer Operation

The counter register is reset to 0 when T0CEN=0. When flag T0CEN is set to 1, the counter starts counting up at a rate of one count every TIMER0_PRESCALER clock cycles.

When counter register = reload register, the following will happen:

- If flag T0ARL is 0 the core will clear flag T0CEN and and raise flag Irq, triggering an interrupt. The counter will overflow to 0000h and stop.
- If flag T0ARL is 1 then flag T0CEN will remain high and flag Irq will be raised, triggering an interrupt. The counter will overflow to 0000h and continue counting.

Register TSTAT

This register reflects the status of the timer:

	7	6	5	4	3	2	1	0
TSTAT	0	0	T0CEN	T0ARL	0	0	0	T0IRQ
reset	0	0	0	0	0	0	0	0
	h	h	r/w	r/w	h	h	h	w1c

Bits marked 'h' are hardwired and can't be modified.

Bits marked 'r' are read only; they are set and reset by the core.

Bits marked 'r/w' can be read and written to by the CPU and can be reset by the core.

Bits marked 'w1c' (write 1 to clear) are set by the core when an interrupt has been triggered and must be cleared by the software by writing a 1.

T0CEN	Count ENable. Must be set to 1 by the CPU to start the counter. When T0CEN=0 the prescaler and the counter register are reset to 0. Writing a 1 to T0CEN will start the count up. The counter will increment until it matches the compare register value (if T0ARL=1) or until it overflows (if T0ARL=0), at which moment the counter register will roll back to zero.
T0ARL	Auto ReLoad. Set to 1 to enable compare/autoreload mode.
T0IRQ	Timer interrupt pending service. Raised when the timer interrupt is triggered, cleared by writing 1 to it.

Registers T0L,T0H

	7	6	5	4	3	2	1	0
T0L	Counter register value, bits 7..0							
reset	0	0	0	0	0	0	0	0
	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

	7	6	5	4	3	2	1	0
T0H	Counter register value, bits 15..8							
reset	0	0	0	0	0	0	0	0
	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

Bits marked 'r/w' can be read and written to by the CPU and can be reset by the core.

T0H:T0L This is the current value of the counter register. Will be reset to zero when T0CEN=0.

Registers TCL,TCH

	7	6	5	4	3	2	1	0
T0CL	Compare register value, bits 7..0							
reset	1	1	1	1	1	1	1	1
	w	w	w	w	w	w	w	w

	7	6	5	4	3	2	1	0
T0CH	Compare register value, bits 15..8							
reset	1	1	1	1	1	1	1	1
	w	w	w	w	w	w	w	w

Bits marked 'w' can be written to by the CPU but reading them will yield an undefined value.

T0CH:T0CL This is the current value of the reload register.

7.3.- Input/Output Ports

The MCU includes 4 8-bit I/O ports. In order to save logic, the ports are hardwired to be either input or output, and are not configurable even at synthesis time -- it is simpler and cheaper to just add or modify whatever port setup is needed in each particular application than trying to provide for all possibilities in advance.

The port SFR addresses are the same as the original P0..P3 port addresses. However, since the ports are strictly input or strictly output, the behavior of the ports is different in a very important way:

The '**Read-Modify-Write**' behavior of the MCS51 is **not implemented**:

- All instructions reading an input port read the pin regardless of addressing mode.
- All instructions reading an output port read the register regardless of addressing mode.

In short, writing to an input port will not have any effect. Reading an output port will access the port output registers and NOT the pins, as stated above.

All input ports are registered unconditionally every clock cycle, there's no need to provide registers external to the MCU entity.

7.4.- External Interrupt Inputs

The MCU has 8 external interrupt inputs which, in the current version of the core, are meant mostly for debugging.

The inputs are registered and level sensitive. As long as input external_irq[i] is high, flag EXTINT[i] will be high and the interrupt request line IRQ0 of the CPU will be asserted.

Subsequent versions of the core will use edge triggering and add a mask register.

Register EXTINT

This register contains the external interrupt pending flags:

	7	6	5	4	3	2	1	0
EXTINT	EIRQ7	EIRQ6	EIRQ5	EIRQ4	EIRQ3	EIRQ2	EIRQ1	EIRQ0
reset	0	0	0	0	0	0	0	0
	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c

Bits marked 'w1c' (write 1 to clear) are set by the core when an interrupt has been triggered and must be cleared by the software by writing a 1.

EIRQ<i>	External interrupt <i> pending service. Raised when the core input external_irq[i] is high, cleared by writing 1 to it as long as the input has been cleared too.

8.- Opcode Table

Table 6: ***Instruction Set Summary***

	0	1	2	3	4	5	6	7
0	2 NOP 1	7/8 JBC bit 3	6/7 JB bit 3	6/7 JNB bit 3	3/5 JC offset 2	3/5 JNC offset 2	3/5 JZ offset 2	3/5 JNZ offset 2
1	AJMP addr11 2	ACALL addr11 2	AJMP addr11 2	ACALL addr11 2	AJMP addr11 2	ACALL addr11 2	AJMP addr11 2	ACALL addr11 2
2	6 LJMP addr16 3	8 LCALL addr16 3	7 RET 1	RETI 1	5 ORL dir, A 2	5 ANL dir, A 2	5 XRL dir, A 2	5 ORL C, bit 2
3	3 RR A 1	3 RRC A 1	3 RL A 1	3 RLC A 1	5 ORL dir, #imm 3	5 ANL dir, #imm 3	5 XRL dir, #imm 3	4 JMP @A+DPTR 1
4	3 INC A 1	3 DEC A 1	4 ADD A, #imm 2	4 ADDC A, #imm 2	4 ORL A, #imm 2	4 ANL A, #imm 2	4 XRL A, #imm 2	4 MOV A, #imm 2
5	5 INC dir 2	5 DEC dir 2	5 ADD A, dir 2	5 ADDC A, dir 2	5 ORL A, dir 2	5 ANL A, dir 2	5 XRL A, dir 2	5 MOV dir, #imm 3
6	7 INC @R0 1	7 DEC @R0 1	7 ADD A, @R0 1	7 ADDC A, @R0 1	7 ORL A, @R0 1	7 ANL A, @R0 1	7 XRL A, @R0 1	5 MOV @R0, #imm 2
7	7 INC @R1 1	7 DEC @R1 1	7 ADD A, @R1 1	7 ADDC A, @R1 1	7 ORL A, @R1 1	7 ANL A, @R1 1	7 XRL A, @R1 1	5 MOV @R1, #imm 2
8	5 INC R0 1	5 DEC R0 1	5 ADD A, R0 1	5 ADDC A, R0 1	5 ORL A, R0 1	5 ANL A, R0 1	5 XRL A, R0 1	4 MOV R0, #imm 2
9	5 INC R1 1	5 DEC R1 1	5 ADD A, R1 1	5 ADDC A, R1 1	5 ORL A, R1 1	5 ANL A, R1 1	5 XRL A, R1 1	4 MOV R1, #imm 2
A	5 INC R2 1	5 DEC R2 1	5 ADD A, R2 1	5 ADDC A, R2 1	5 ORL A, R2 1	5 ANL A, R2 1	5 XRL A, R2 1	4 MOV R2, #imm 2
B	5 INC R3 1	5 DEC R3 1	5 ADD A, R3 1	5 ADDC A, R3 1	5 ORL A, R3 1	5 ANL A, R3 1	5 XRL A, R3 1	4 MOV R3, #imm 2
C	5 INC R4 1	5 DEC R4 1	5 ADD A, R4 1	5 ADDC A, R4 1	5 ORL A, R4 1	5 ANL A, R4 1	5 XRL A, R4 1	4 MOV R4, #imm 2
D	5 INC R5 1	5 DEC R5 1	5 ADD A, R5 1	5 ADDC A, R5 1	5 ORL A, R5 1	5 ANL A, R5 1	5 XRL A, R5 1	4 MOV R5, #imm 2
E	5 INC R6 1	5 DEC R6 1	5 ADD A, R6 1	5 ADDC A, R6 1	5 ORL A, R6 1	5 ANL A, R6 1	5 XRL A, R6 1	4 MOV R6, #imm 2
F	5 INC R7 1	5 DEC R7 1	5 ADD A, R7 1	5 ADDC A, R7 1	5 ORL A, R7 1	5 ANL A, R7 1	5 XRL A, R7 1	4 MOV R7, #imm 2

Table 7: ***Instruction Set Summary (Continued)***

	8	9	A	B	C	D	E	F
0	5 2 SJMP offset	5 3 MOV DPTR, #imm	5 2 ORL C, /bit	5 2 ANL C, /bit	5 2 PUSH dir	5 2 POP dir	3 1 MOVX A, @DPTR	3 1 MOVX @DPTR, A
1	2 AJMP addr11	2 ACALL addr11	2 AJMP addr11	2 ACALL addr11	5 2 AJMP addr11	7 2 ACALL addr11	2 AJMP addr11	2 ACALL addr11
2	5 2 ANL C, bit	5 2 MOV bit, C	5 2 MOV C, bit	5 2 CPL bit	5 2 CLR bit	5 2 SETB bit	6 1 MOVX A, @R0	5 1 MOVX @R0, A
3	4 1 MOVC A, @A+PC	4 1 MOVC A, @A+DPT	3 1 INC DPTR	3 1 CPL C	3 1 CLR C	3 1 SETB C	6 1 MOVX A, @R1	5 1 MOVX @R1, A
4	10 1 DIV AB	4 2 SUBB A, #imm	3 1 MUL AB	5/6 3 CJNE A, #imm	3 1 SWAP A	4 1 DA A	3 1 CLR A	3 1 CPL A
5	5 3 MOV dir, dir	5 2 SUBB A, dir	2 1 (NOP)	6/7 3 CJNE A, dir	6 2 XCH A, dir	7/8 3 DJNZ dir	5 2 MOV A, dir	4 2 MOV dir, A
6	7 2 MOV dir, @R0	7 1 SUBB A, @R0	6 2 MOV @R0, dir	8/9 3 CJNE @R0, #imm	7 1 XCH A, @R0	8 1 XCHD A, @R0	7 1 MOV A, @R0	5 1 MOV @R0, A
7	7 2 MOV dir, @R1	7 1 SUBB A, @R1	6 2 MOV @R1, dir	8/9 3 CJNE @R1, #imm	7 1 XCH A, @R1	8 1 XCHD A, @R1	7 1 MOV A, @R1	5 1 MOV @R1, A
8	5 2 MOV dir, R0	5 1 SUBB A, R0	5 2 MOV R0, dir	6/7 3 CJNE R0, #imm	6 1 XCH A, R0	7/8 2 DJNZ R0	5 1 MOV A, R0	5 1 MOV R0, A
9	5 2 MOV dir, R1	5 1 SUBB A, R1	5 2 MOV R1, dir	6/7 3 CJNE R1, #imm	6 1 XCH A, R1	7/8 2 DJNZ R1	5 1 MOV A, R1	5 1 MOV R1, A
A	5 2 MOV dir, R2	5 1 SUBB A, R2	5 2 MOV R2, dir	6/7 3 CJNE R2, #imm	6 1 XCH A, R2	7/8 2 DJNZ R2	5 1 MOV A, R2	5 1 MOV R2, A
B	5 2 MOV dir, R3	5 1 SUBB A, R3	5 2 MOV R3, dir	6/7 3 CJNE R3, #imm	6 1 XCH A, R3	7/8 2 DJNZ R3	5 1 MOV A, R3	5 1 MOV R3, A
C	5 2 MOV dir, R4	5 1 SUBB A, R4	5 2 MOV R4, dir	6/7 3 CJNE R4, #imm	6 1 XCH A, R4	7/8 2 DJNZ R4	5 1 MOV A, R4	5 1 MOV R4, A
D	5 2 MOV dir, R5	5 1 SUBB A, R5	5 2 MOV R5, dir	6/7 3 CJNE R5, #imm	6 1 XCH A, R5	7/8 2 DJNZ R5	5 1 MOV A, R5	5 1 MOV R5, A
E	5 2 MOV dir, R6	5 1 SUBB A, R6	5 2 MOV R6, dir	6/7 3 CJNE R6, #imm	6 1 XCH A, R6	7/8 2 DJNZ R6	5 1 MOV A, R6	5 1 MOV R6, A
F	5 2 MOV dir, R7	5 1 SUBB A, R7	5 2 MOV R7, dir	6/7 3 CJNE R7, #imm	6 1 XCH A, R7	7/8 2 DJNZ R7	5 1 MOV A, R7	5 1 MOV R7, A

In the opcode tables above, the number of bytes of the instruction is in the right top corner, in blue.

The number in red in the left top corner is the cycle count (min/max for conditional jumps).

The opcode table has been automatically generated from the cycle count log file for the opcode tester program. The cycle count log file (file **/sim/cycle_count_log.csv**) records the cycle count for all instructions executed by any test bench; but the opcode tester does not execute all opcodes. Those opcodes for which there is no data in the log file are colored in grey, and their cycle count numbers are blank.

Optional instructions (DA and XCHD) are highlighted in red. When not implemented, these opcodes behave as NOP and their cycle count is 2.

Note that all the untested opcodes are part of a wider opcode family for which at least some members have been tested (e.g. ACALL, AJMP instructions).

NOTE: The table generation script is at **/tools/build_opcode_table/svg_op_table.py**. It will produce SVG files which have to be manually converted to PNG and imported into the ODT datasheet file. It is included in the project repository for completeness, though it is not expected to be of any use.

9.- Performance

9.1.- Synthesis Results

These are the synthesis results for the Dhrystone demo.

Table 8: ***Synthesis Results for 'Dhrystone' Demo***

Target Device	Synthesis Type	F_{max}	Resources			
			CPU	Timer	UART	Total for Dhrystone demo
Cyclone-II -C7	Balanced	62 MHz	997 LEs, 29 M4Ks, 1 MUL9	85 LEs	147 LEs	1339 LEs, 29 M4Ks, 1 MUL9
Spartan-3A -4	Balanced	35 MHz	1162 LUTs, 10BRAM, 1MUL	66 LUTs	99 LUTs	1424 LUTs, 10 BRAMs, 1 MUL18

(*1) The Spartan results include only the MCU and none of the DE-1 on-board glue logic (7-segment decoders, etc.)

The Dhrystone demo includes 12KB of ROM and 2 KB of XRAM, besides the IRAM. On the DE-1 board, it also includes some auxiliary glue logic that is included in the above total count *for the Cyclone-II target only*. That glue logic accounts for ~66LEs.

Results for Cyclone-II are the the actual synthesis results obtained for the Dhrystone demo on Terasic's DE-1 board (using top file **c2sb_demo.vhdl**).

Results for the Spartan-3 include about 80 LUTs used as pass-through.

In both cases, the synthesized core implements the BCD opcodes (DA and XCHD). When not implemented, CPU area decreases by about 31 LEs (or 50 to 80 LUTs in the Spartan chip). Clock rate does not change.

The above results have been obtained with Quartus-2 11.1 sp2 and Xilinx ISE Webpack 14.3. The synthesis has been performed *with a single clock rate constraint* and the results must be considered illustrative only.

The project files for Quartus and ISE are included in the repository.

9.2.- Dhrystone 2.1 Benchmark

The MCU has executed a version of the Dhrystone 2.1 benchmark, adapted for MCUs by ECROS Technology and slightly modified to suit the light52 core. It has been compiled with SDCC with default options.

The benchmark has been executed on a DE-1 development board with a Cyclone-II FPGA clocked at 50 MHz using the top module entity **/vhdl/demos/c2sb/c2sb_demo.vhdl**. The benchmark executes 25000 iterations over the Dhrystone loop and produces the following results:

Dhrystone 2.1 Benchmark Results	
1646	Dhrystones per second @ 50 MHz
0.9368	Dhrystone MIPS (*1)
0.0187	Dhrystone MIPS per MHz

(*1) 1 Dhrystone MIPS = 1757 Dhrystones per second

In order to give some context for this benchmark, the following table compares the results for a few representative cores:

CPU	DMIPS/MHz	Advantage vs. light52	F_{max}
Light52	0.0187		62 MHz
Intel MCS51	0.0094	x0.5	12 MHz
CAST R8051XC-2 AF	0.0883	x4.7	35 MHz

(*1) F_{max} on a Cyclone-II FPGA, the same for both cores.

As can be seen below, this core is about twice as fast as a 12-clocker at the same clock rate and therefore can be characterized as a 6-clocker, even if the clock count per instruction is not linearly scalable from the original.

The single-clocker CAST's R8051XC-2 AF has been selected for comparison because its feature set is not far ahead of light52's -- yet it's bigger and not that much faster once the difference in F_{max} is accounted for.