SLC1657 8-Bit RISC uC Core
Technical Reference Manual

Silicore Corporation

•

Silicore Corporation
6310 Butterworth Lane; Corcoran, MN (USA) 55340
TEL: (763) 478-3567 FAX: (763) 478-3568
URL: www.silicore.net

Electronic Design
Sensors • IP Cores

SLC1657 8-bit RISC Microcontroller for VHDL

History:

18 August 2003: SLC1657 Manual Revision 1.5 released under public license. Corresponds to the SLC1657 Core, REV: 2.0. Author: Silicore Corporation.

Contents

1.0 OVERVIEW 5

1.1 FEATURES OF THE SLC1657 6
1.2 RECOMMENDED SKILL LEVEL 7

2.0 SYSTEM ARCHITECTURE 9

2.1 CORE OVERVIEW 9
2.2 EXTERNAL ARCHITECTURE 11
2.3 INTERNAL ARCHITECTURE 16

3.0 PROGRAMMING REFERENCE 27

3.1 REGISTER SET 29
3.2 RESET OPERATION 38
3.3 I/O PORT OPTIONS 39
3.4 TIMER/COUNTER & WATCHDOG OPERATION 41
3.5 POWER-DOWN OPERATION 46
3.6 COMPATIBILITY WITH THE MICROCHIP PART 48
3.7 INSTRUCTION SET 50

4.0 VHDL SYNTHESIS AND TEST 67

4.1 VHDL SIMULATION AND SYNTHESIS TOOLS 67
4.2 VHDL PORTABILITY 69
4.3 REQUIRED RESOURCES ON THE TARGET DEVICE 69
4.4 SOFT CORE INSTALLATION 72
4.5 CORE INTEGRATION 73
4.6 VHDL REFERENCE BOOKS 77

5.0 HARDWARE (VHDL ENTITY) REFERENCE 79

5.1 ALULOGIC ENTITY 79
5.2 BINADDER ENTITY 81
5.3 BUC08NNP ENTITY 82
5.4 BUC11CPP ENTITY 82
5.5 CLOCKDIV ENTITY 82
5.6 INDEXREG ENTITY 83
5.7 INSTRDEC ENTITY 83
5.8 INTRCONV ENTITY 90
5.9 MUX08X04 ENTITY 90
5.10 MUX08X08 ENTITY 91
5.11 MUX11X04 ENTITY 91
5.12 PORTSREG ENTITY 91
5.13 PRESCALE ENTITY 92
5.14 PROGCNTR ENTITY 93
5.15 REG08CNN ENTITY 96
5.16 REG08CPN ENTITY 96
5.17 REG11CNN ENTITY 97
5.18 REG12CRN ENTITY 97
5.19 RESETGEN ENTITY 97
5.20 STATSREG ENTITY 99
5.21 TCOPTREG ENTITY 100
5.22 TIMRCNTR ENTITY 100
5.23 TIMRSYNC ENTITY 101
5.24 TOPLOGIC ENTITY 104
5.25 WATCHDOG ENTITY 104

6.0 IMPLEMENTATION ON THE XILINX SPARTAN 2 FPGA 107

6.1 EVALUATION KIT FOR XILINX SPARTAN 2 FPGA 108
6.2 THE XSP2EVAL EXERCISE 111
6.3 USING THE EMULATION ROM (DOWNLOAD) CAPABILITY 116
6.4 CREATING AN EMBEDDED PROM 118
6.5 VHDL ENTITY REFERENCE FOR XILINX SPARTAN 2 120

7.0 IMPLEMENTATION ON THE ALTERA FLEX 10KE FPGA 129

7.1 EVALUATION KIT FOR ALTERA FLEX 10KE FPGA 130
7.2 THE AF10EVAL EXERCISE 133
7.3 USING THE EMULATION ROM (DOWNLOAD) CAPABILITY 138
7.4 CREATING AN EMBEDDED PROM 140
7.5 VHDL ENTITY REFERENCE FOR ALTERA FLEX 10KE 142

8.0 IMPLEMENTATION ON THE AGERE ORCA 3L FPGA 150

8.1 EVALUATION KIT FOR AGERE ORCA 3L FPGA 151
8.2 THE AGO3EVAL EXERCISE 154
8.3 USING THE EMULATION ROM (DOWNLOAD) CAPABILITY 160
8.4 CREATING AN EMBEDDED ROM 162
8.5 VHDL ENTITY REFERENCE FOR AGERE ORCA 3L 164

APPENDIX A – THE INTEL HEX FORMAT 172

APPENDIX B – GNU LESSER GENERAL PUBLIC LICENSE 174

APPENDIX C – GNU FREE DOCUMENTATION LICENSE 184

INDEX 192

1.0 Overview

The Silicore® SLC1657 is an eight-bit RISC microcontroller. It is delivered as a VHDL[1] soft core[2] module, and is intended for use in both ASIC and FPGA type devices. It is useful for microprocessor based embedded control applications such as: sensors, medical devices, consumer electronics, automotive systems, telecommunications, military and industrial controls.

The core is especially useful wherever there is limited circuit board space. As shown in Figure 1-1, all applications can be integrated into a single FPGA or ASIC device, thereby creating a very compact design. For example, very small sensor circuits can be created with the core.

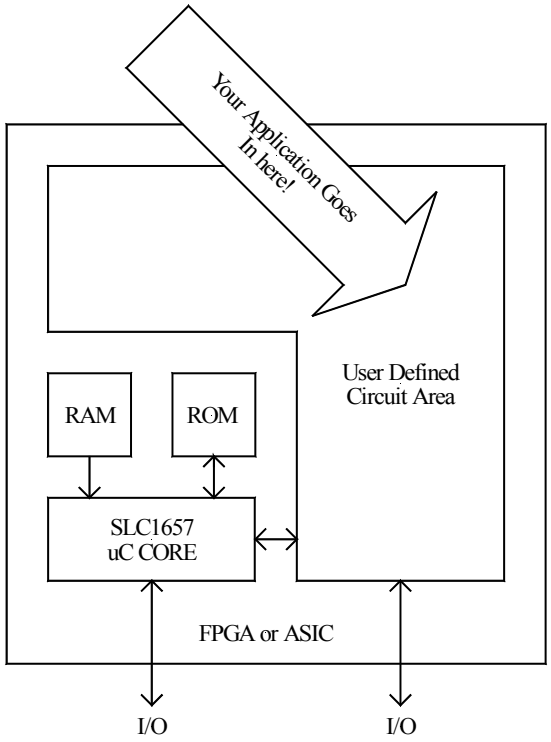


Figure 1-1. Create your own microcontroller with the SLC1657.

[1] VHDL: VHSIC Hardware Description Language.

[2] The term 'soft core' means that the microcontroller is delivered as VHDL source code. This must be synthesized by the user into a usable microcontroller. This is opposed to 'firm cores' or 'hard cores', where the user is prohibited from seeing or adjusting the internal architecture of the product. The SLC1657 is delivered in this way to (a) make it more portable, (b) improve testability (and test creation) and (c) allow the user more flexibility in his/her design.

When implemented on an FPGA device, the SLC1657 offers a completely user-defined microcontroller. This eliminates expensive NRE charges and lengthy lead times which are common for semi-custom integrated circuits. The end user can completely control the entire system integration process.

Furthermore, the core is useful for high volume applications. That's because it is unusually compact, and can be produced inexpensively. It can also be combined with other peripherals on the same device, thereby creating custom, single-chip microcontrollers. This concept also allows the core to be used in devices with a wide variety of options such as package type, temperature range and radiation hardening.

1.1 Features Of The SLC1657

- Eight-bit RISC microcontroller.
- Dual instruction and data buses with Harvard architecture.
- Fast operation...all microcontroller instructions (except branches) require one clock cycle. Branch instructions require two clock cycles.
- Very compact design minimizes gate count.
- 24 input and 48 output I/O lines.
- General purpose, eight-bit timer/counter module.
- Power-down / sleep mode feature for low power applications.
- Instruction ROM: 2,048 x 12 bit. Can be configured as embedded ROM, or as an emulation ROM for software development purposes.
- General purpose registers (RAM): 72 bytes.
- 32 op-code instructions with easy-to-use application software environment.
- Numerous application software tools are available. The SLC1657 is software compatible with the industry standard PIC® series of microcontrollers made by Microchip Technology Inc. There are many software tools available from third-party vendors. These include assemblers, 'C' compilers, simulators and fuzzy logic tools.
- Microcontroller design written in the flexible VHDL hardware description language. The SLC1657 is delivered as a 'soft-core', meaning that all VHDL

source code and test benches are supplied. This allows the user to 'tweak' the design for a particular application. Complete documentation is also provided.

- Straightforward synchronous design simplifies system integration.
- Very simple timing constraint definition.
- The maximum operating speed is a function of the target device technology[3].

1.2 Recommended Skill Level

Figure 1-2 shows the recommended skill (or experience) level required to operate and synthesize the SLC1657. This microcontroller is one of the easiest to use on the market. Users familiar with one or more microprocessor chips should be able to operate and program the core with little or no problem. The user may find it helpful to purchase one or more of the recommended books listed in Chapter 3. Furthermore, the user should be able to find a wide variety of software examples on the internet and other sources. The Parallax® simulator, also described in Chapter 3, is an inexpensive and useful tool for learning the instruction set.

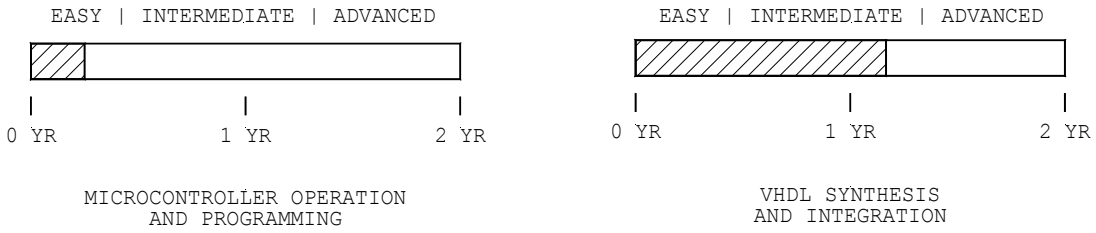


Figure 1-2. Recommended experience level required to operate (program) and synthesize (integrate) the SLC1657.

It is recommended that the user have some experience with VHDL syntax and synthesis before attempting to integrate this core (or almost any other HDL core for that matter) into an FPGA or ASIC. Most VHDL users report a fairly stiff learning curve on their first project, so it's better to have that experience before attempting to integrate the core. Prior experience with one or two medium size VHDL projects should be sufficient. On the other hand, some users may find the SLC1657 an excellent way to learn many of the concepts in the VHDL language. Those users should find the integration experience rewarding. A good way to learn about the core is to use one of the evaluation kits. For ex-

[3] Typical operating speeds on FPGA parts are about 20 MHz (or 20 MIPS). Speeds are much higher for ASICs.

ample, if your target device is a Lucent FPGA, then the core can be synthesized and operated on one of the Silicore evaluation boards.

2.0 System Architecture

The Silicore® SLC1657 was designed with five major objectives. These were to create:

- A compact design...small enough to be used in both FPGA and ASIC devices.
- A fast design...capable of solving 'real world' computing problems.
- A portable design...one which can be used as a synthesizable VHDL core.
- A compatible design...with a variety of software development tools.
- A fast time-to-market design...with plenty of documentation and support.

To achieve these objectives, the SLC1657 was designed as an eight-bit RISC microcontroller. This allows it to meet the criteria for both a compact and a fast design.

Furthermore, a microcontroller type topology is used. The main difference between a microcontroller and a microprocessor is the I/O interface: microcontrollers interface to the outside world with I/O ports, and microprocessors use I/O buses. A microcontroller topology is used because it is easier to integrate as an FPGA or ASIC core.

Furthermore, the SLC1657 has a large base of software tools. The core is instruction compatible with other industry standard microcontrollers. Assemblers, simulators, 'C' compilers and fuzzy logic generators are available for that device. They are low cost, and are available for a number of operating systems from a variety of software suppliers.

Finally, the SLC1657 was designed to facilitate fast time-to-market for the end user. Virtually all design documentation for this product is available from Silicore Corporation, including all VHDL source files and test benches. The product can also be bundled with other services, such as design customization, integration and on-site training.

2.1 Core Overview

The SLC1657 uses a RISC, or *reduced instruction set computer* architecture. One advantage of this architecture is that it uses an *unencoded* instruction stream. This means that most of the control logic is embedded within the instruction itself. This eliminates much of the decode logic required by CISC, or *complex instruction set computer* architectures, which encode their instructions in an intermediate encoding scheme.

Another common feature of the RISC architecture is the use of separate instruction and data buses. This is often called a *Harvard Architecture*, and alleviates the need for a shared main bus. Shared buses can create bottlenecks (in terms of both speed and logic size) because they pass both the instructions and data. Furthermore, they usually require three-state buses, which tend to make them less portable as FPGA cores.

The SLC1657 is intended to solve 'real world' embedded computing problems. Several popular features have been used in the core to support these applications. These include embedded RAM, ROM, I/O ports, a general purpose timer/counter, a watchdog timer and a power-down mode.

During normal use, the SLC1657 uses a 2,048 x 12 bit instruction ROM. In both FPGA and ASIC devices the ROM is created from standard cells, which are usually supplied by the manufacturer of the target device. This allows ROM to be integral to the target device, and eliminates the need for external parts. Also, in SRAM[4] type FPGA devices the ROM can be automatically loaded when the device is configured (during power-up).

An emulation ROM capability can also be used for application software development. This is provided as an optional VHDL entity, and includes a parallel port download interface. PC based download software and parallel port cable are provided with the evaluation kit.

To use the software development environment, the core must be synthesized with an emulation ROM core (which is provided with the kit). Once software development is complete, the emulation core is replaced with the embedded ROM. The emulation ROM can also be used in the target application. This is useful when used as a smart peripheral to another computer.

The SLC1657 uses a segmented addressing architecture for the instruction memory. This architecture uses four instruction memory banks, each having 512 x 12-bit memories. This results in a total instruction capacity of 2,048 12-bit words. Two instruction bank bits in the STATUS register select the current bank to use.

Register RAM also uses a segmented addressing scheme, allowing a total of 72, eight bit general purpose registers. This is in addition to sixteen special and general purpose registers, which are available from all register banks.

The banked architecture is an upgrade from the SLC1655 predecessor. That processor used a single bank of 512 words of memory. The size of the SLC1657 memories can be reduced to make it code compatible with the SLC1655.

I/O on the SLC1657 is handled through a flexible interface with 24 input lines, 48 output lines and three write strobes. These can be configured by the user in several modes. For example they can be used as-is, they can be combined to produce 24 bi-directional three-state lines, or they can connect to intermediate logic such as FIFO buffers.

The core has a general purpose timer/counter. This entity includes an eight-bit counter and a programmable, eight-bit prescaler. The source of the timer/counter input can be either from the internal clock [MCLK / 4] or from an external [TMRCLK] pin. The

[4] SRAM: Static RAM

timer/counter is useful in many real-time applications. For example, it can be used for time interval measurement and pulse counting.

The watchdog timer is popular in embedded control applications. When enabled, the watchdog resets the microcontroller if the RWT instruction is not issued before the end of a time-out period. Furthermore, the time-out period can be increased by routing the watchdog through the timer/counter prescaler.

The SLC1657 has a power-down feature that allows it to reduce power consumption. This is especially useful in low current (e.g. battery powered) applications. A special PWRDN instruction causes the microcontroller to halt operation, thereby reducing current consumption. The actual reduction in power depends upon the clock frequency and quiescent current consumption of the target device[5].

All of these features are controlled by a simple instruction set with a total of 32 op-codes. These include add, subtract, increment, decrement, logical, loop and branch instructions. A branch-to-subroutine and a small (two element) stack is also included in the core.

2.2 External Architecture

Figure 2-1 is a block diagram of the SLC1657 external architecture. This shows the VHDL TOPLOGIC *entity*[6], and illustrates what the microcontroller core looks like to the rest of the FPGA or ASIC device.

The main body of the core is provided in the TOPLOGIC entity. This contains all of the control logic (instruction decoder, registers, etc.) for the device. The user combines TOPLOGIC with other entities to form a complete microcontroller with RAM, ROM and I/O elements.

The RAM and ROM entities are not included[7] in the TOPLOGIC core because the VHDL synthesis standards do not handle these very well. Entity descriptions for these are provided by the FPGA or ASIC vendor as 'standard cells'. Furthermore, this allows the user to choose between two ROM styles:

- Emulation ROM[8]. This allows application code to be downloaded and debugged over a PC (Centronics style) parallel port cable. It is very useful for application software development.

[5] On the SLC1655 ORCA™ FPGA evaluation board, the power reduction is about 90%.

[6] The term 'entity' is VHDL jargon for a *subassembly* or *component*. It is used throughout this manual as a way of describing the various parts of the microcontroller.

[7] Sample entities for specific FPGA target devices are included with the SLC1657 development kits.

[8] PC-compatible download software and cable are provided with the SLC1657 evaluation kit.

- Embedded ROM[9]. The embedded ROM is fixed, and cannot be changed. Once the application software has been designed and tested, it is converted to embedded ROM cells, and the core is re-synthesized.

Separate RAM and ROM elements also makes testing the part much easier, especially in ASIC applications. ASICs tend to have more rigorous testability requirements than reconfigurable FPGA parts. That's because ASICs tend to be screened during the die fabrication process, whereas reconfigurable FPGAs can be 100% pre-tested. The architecture of the SLC1657 allows the core, the RAM and the ROM to be tested separately on the die.

The I/O elements are also provided by the user. Since the core can be used in a large variety of ways, it is better if the user provides these elements. For example, some applications require that the I/O's be used as uni-directional pins (i.e. separate input and output lines), while some require bi-directional, three-state I/O pins.

The external signal descriptions for the core are shown in Table 2-1.

Table 2-1. SLC1657 external signal description.

Signal Name	Input (I) Output (O)	Signal Description
MCLK	I	Microcontroller (master) clock.
PCLK*	I	Program clock / emulation ROM (optional).
PCOUT0-2(7..0)	O	Port control output.
PDAT*	I	Program data / emulation ROM (optional).
PLCH*	I	Program latch / emulation ROM (optional).
PROG*	I	Program enable / emulation ROM (optional).
PTIN0-2(7..0)	I	I/O PORT input.
PTOUT0-2(7..0)	O	I/O PORT output.
PTSTB0-2	O	Port output strobe.
RESET	I	Reset (external).
SLEEP	O	Power-down / sleep mode.
TMRCLK	I	External timer/counter clock source.

Note: (*) active low signal.

[9] Software to create a ROM database is included with the SLC1657 development kit.

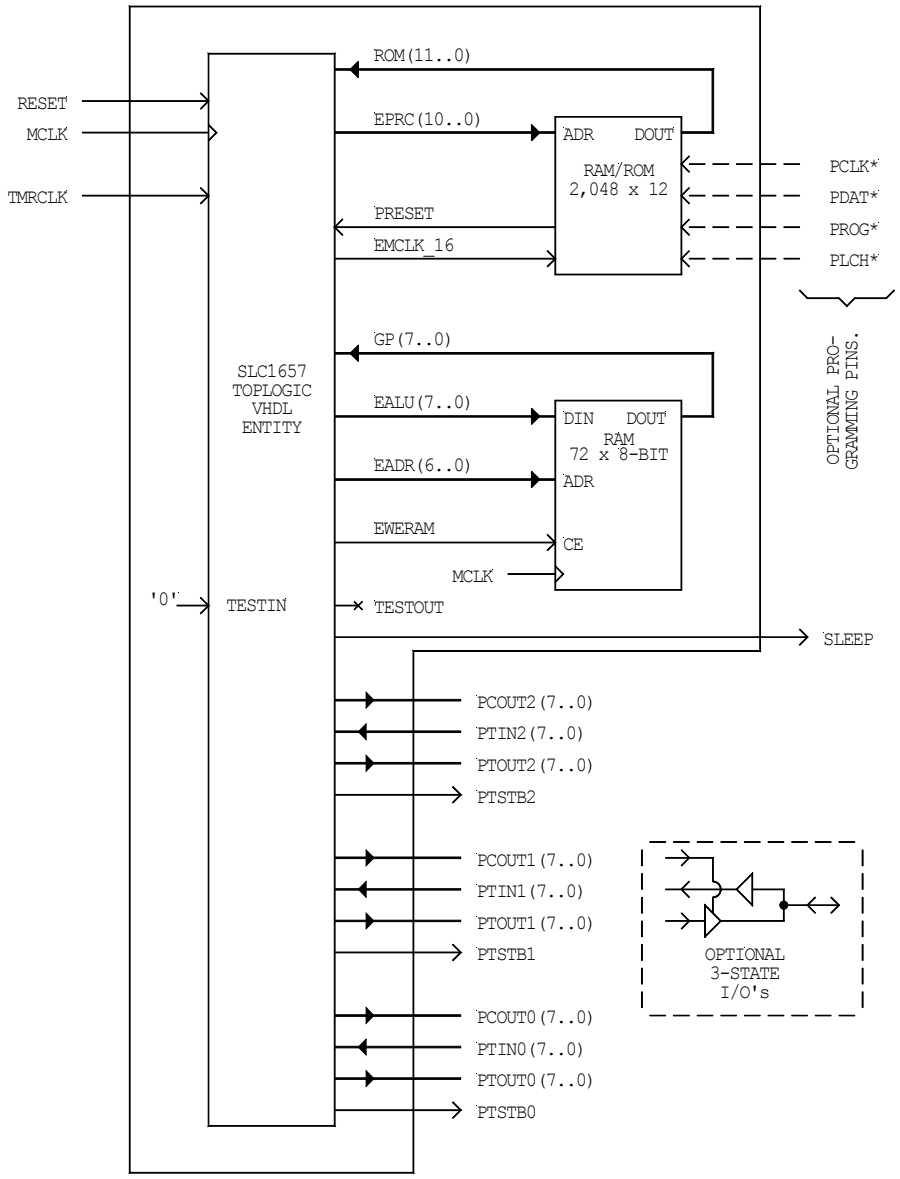


Figure 2-1. SLC1657 external architecture block diagram.

2.2.1 MCLK Signal

The [MCLK] signal synchronizes the internal activity of the core. Its frequency is dependent upon the target device (FPGA, ASIC etc.), and must be evaluated by the user during VHDL synthesis. In most cases the clock may be operated down to 0 Hz, and up to the maximum frequency limit of the target device technology. The duty cycle of the clock is not particularly important, as only the positive going edge of [MCLK] is used. A standard 60/40 duty cycle is adequate for this application.

2.2.2 PCLK* Signal

The [PCLK*] signal is an optional programming clock. It synchronizes the [PDAT*] signal when an emulation ROM entity is used. [PCLK*] is an active low signal.

2.2.3 PCOUT0-2(7..0) Signals

PCOUT0(7..0), PCOUT1(7..0) and PCOUT2(7..0) are eight-bit port control output buses. They can be used as general purpose output ports. They are accessed by writing to the port control registers PC0, PC1 and PC2 (using the MOVP instruction).

When the SLC1657 is configured to operate as part of a three-state bi-directional I/O port, then these buses are generally used to control the three-state operation of the port.

2.2.4 PDAT* Signal

The [PDAT*] signal is an optional programming data pin that is used with an emulation ROM capability. [PDAT*] is an active low signal.

2.2.5 PLCH* Signal

The [PLCH*] signal is an optional programming latch pin. It is used to latch data into the emulation ROM. [PLCH*] is an active low signal.

2.2.6 PROG* Signal

The [PROG*] signal is an optional programming enable pin. When asserted, [PROG*] places the core into the emulation ROM mode. It has the same effect as the external [RESET] signal. [PROG*] is an active low signal.

2.2.7 PTIN0-2(7..0) Signals

PTIN0(7..0), PTIN1(7..0) and PTIN2(7..0) are general purpose input port buses. Input port data is accessed by reading the PORT0, PORT1 and PORT2 registers located at addresses 0x05, 0x06 and 0x07 respectively.

2.2.8 PTOUT0-2 Signals(7..0)

PTOUT0(7..0), PTOUT1(7..0) and PTOUT2(7..0) are general purpose output port buses. Output port data is accessed by writing to the PORT0, PORT1 and PORT2 registers located at addresses 0x05, 0x06 and 0x07 respectively.

2.2.9 PTSTB0-2 Signals

PTSTB0, PTSTB1 and PTSTB2 are output port strobes. They can be used to inform external entities that new data is available at the PTOUT0(7..0), PTOUT1(7..0) and PTOUT2(7..0) buses (respectively). Each strobe becomes active for one [MCLK] edge after writing to the PORT0, PORT1 or PORT2 output ports.

2.2.10 RESET Signal

The [RESET] signal resets all internal circuits. It must be asserted for at least two [MCLK] cycles.

2.2.11 SLEEP Signal

The [SLEEP] signal, when active, indicates that the core has been placed into power-down mode. External entities can use [SLEEP] to turn themselves off, thereby lowering power consumption. This function is especially useful in battery powered applications.

2.2.12 TMRCLK Signal

The [TMRCLK] signal is the external input to the timer/counter. This signal can be operated in synchronous or asynchronous modes (in relation to the [MCLK] pin).

When operated asynchronous mode, the period of the [TMRCLK] signal must exceed the period of [MCLK]. This means that the maximum frequency of the input must be less than 1/2 that of [MCLK]. Stated another way, the [TMRCLK] input must be high for at least one positive [MCLK] edge, and low for another.

In synchronous mode, the [TMRCLK] signal is sampled at every rising edge of [MCLK]. In this case the user must constrain the external design so that [TMRCLK] meets the setup and hold times of the synchronizer in the TIMRCNTR entity. Refer to the TIMRCNTR entity for more details.

2.3 Internal Architecture

The SLC1657 is a register based microcontroller with the internal register set shown in Table 2-2. There are four types of registers: implicit, special purpose, shared general purpose and banked general purpose.

2.3.1 Implicit Registers

The implicit registers include the accumulator (ACCUM), port control (PC0-2), timer/counter option (TCO) control ports and stack (STACK1-2) registers. They are called ‘implicit’ registers because they are implicitly addressed by an instruction. For example, the MOVT instruction moves the accumulator to the timer/counter option register (TCO). The accumulator is a read/write register. The PC0-2 and TCO registers are write-only types.

The stack registers are part of the ‘PROGCNTR’ entity, and are used to store and retrieve the return address during branch-to-subroutine (BSR) and return (RET) instructions. There are only two stack levels, so the user must monitor stack usage accordingly. At first this may seem like an unusually small stack, but they are sufficient.

The SLC1657 follows the industry convention whereby two stack levels are supported. Some application software tools will also support additional stack levels. For example, the ‘CC5X ‘C’ compiler from B Knudsen Data will support additional stack levels if they are implemented in hardware. However, this capability is left to the user to implement.

Table 2-2. Register set (abbreviated).

Register	Address	Bank INDEX(6..5)	# Regs	R/W Access
ACCUM	Implicit	-	1	R/W
PC0	Implicit	-	1	W
PC1	Implicit	-	1	W
PC2	Implicit	-	1	W
TCO	Implicit	-	1	W
STACK1	Implicit	-	1	R/W
STACK2	Implicit	-	1	R/W
INDIRECT	0x00 (*)	All (0-3)	1	R/W
TIMRCNTR	0x01 (*)	All (0-3)	1	R/W
PROGCNTR	0x02 (*)	All (0-3)	1	R/W
STATUS	0x03 (*)	All (0-3)	1	R/W
INDEX	0x04 (*)	All (0-3)	1	R/W
PORT0	0x05 (*)	All (0-3)	1	R/W
PORT1	0x06 (*)	All (0-3)	1	R/W
PORT2	0x07 (*)	All (0-3)	1	R/W
SHARED, GEN PURPOSE	0x08 - 0x0F (*)	All (0-3)	8	R/W
BANKED, GEN PURPOSE	0x10 – 0x1F	0	16	R/W
BANKED, GEN PURPOSE	0x30 – 0x3F	1	16	R/W
BANKED, GEN PURPOSE	0x50 – 0x5F	2	16	R/W
BANKED, GEN PURPOSE	0x70 – 0x7F	3	16	R/W

(*) Indicates shared by accessing the lower 16 bytes of each bank.

2.3.2 Special Purpose Registers

The special purpose registers (0x00 - 0x07) are located in the data address space, and access several dedicated functions. For example, reading the register at address 0x01 returns the current value of the TIMRCNTR register. All of the special purpose registers are read/write types.

2.3.3 General Purpose Registers

The general purpose registers can be used as RAM. They are all read/write types, and are bit-addressable. There are two types of general purpose registers: shared and banked.

There are eight shared general purpose registers. These are accessed between addresses 0x08 and 0x0F, regardless of the state of the register bank selection bits in the INDEX register.

There are four groups of banked general purpose registers, each containing sixteen bytes. These are always accessed through register addresses 0x10 – 0x1F. However, the group (or bank) of registers is selected by bits RB1 and RB0 in the STATUS register. For ex-

ample, if RB1:RB0 are set at '10', then accesses between 0x10 – 0x1F will all ways read or write to the BANKED GENERAL PURPOSE REGISTERS in bank 2.

2.3.4 Interrupts

The SLC1657 does not support interrupts. Polling techniques should be used.

2.3.5 Internal Operation

The internal operation of the SLC1657 is shown in Figure 2-2. This is an abbreviated diagram, but it does show the general relationship between the parts of the microcontroller.

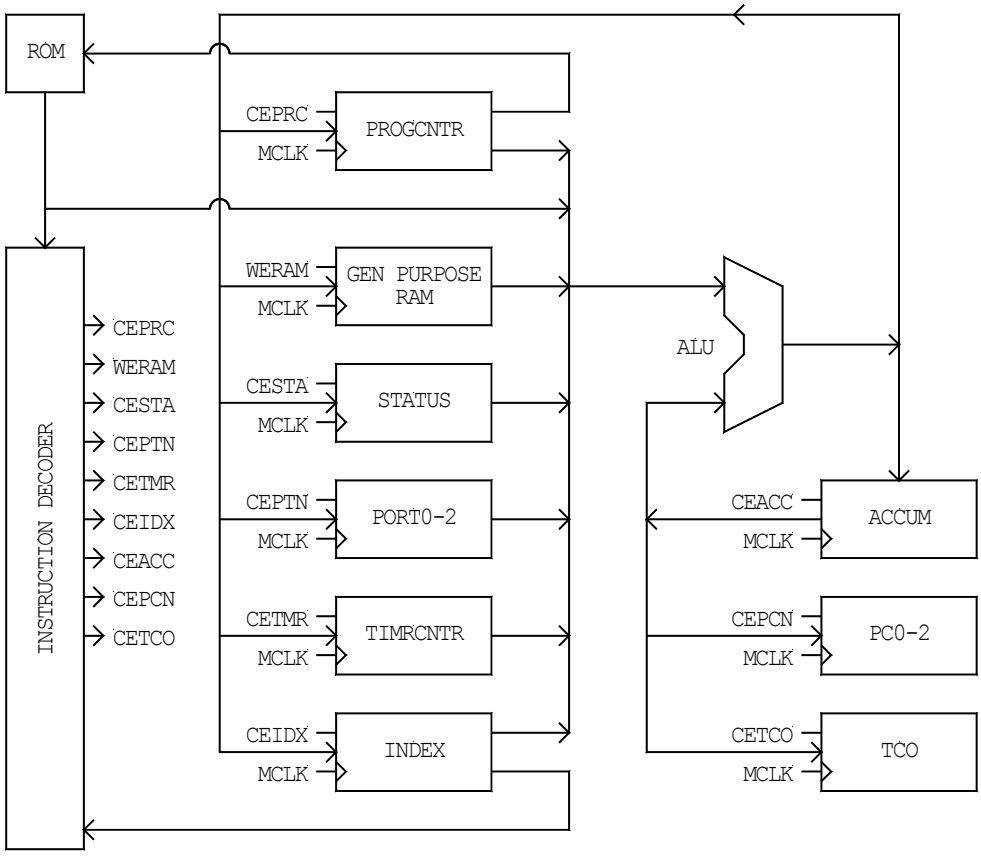


Figure 2-2. Internal architecture (abbreviated).

A program counter provides an address for the instruction ROM. The generation of this address causes an instruction to be fetched, and read by the instruction decoder. The instruction decoder then performs the operation.

Every instruction (except branches) is completed in a single clock cycle. The internal operation of the core is quite simple. After the rising edge of every clock, the output from a single register is routed through the ALU, which operates on the data. The output from the ALU is then latched into one of the registers. The source register, ALU function and destination register is dictated by the current instruction being executed. The instruction decoder controls this activity.

During every cycle the following operations are performed:

1) The program counter generates a new address. The address generated by the program counter actually reflects the instruction following the current instruction. This is called a *pre-fetch* address, and compensates for slow ROMs. During two-cycle branch instructions (BSR and RET) the current address must be *flushed* from the instruction queue, thereby requiring a second clock cycle. This activity is also called *pipelining*.

2) A 12-bit instruction op-code is fetched from ROM.

3) The instruction decoder reads the instruction, determines the source register, and routes its contents to the arithmetic logic unit (ALU). At this time the instruction decoder also informs the ALU what type of operation needs to be performed.

4) The ALU operates on the data. In some cases the accumulator, or data contained within the instruction itself, is used. For example, during the ADD R,D instruction the contents of a register is added to the accumulator.

5) The result of the operation is stored in the accumulator or the register, depending on the addressing mode of the instruction. The instruction decoder determines where to place the data.

6) The next cycle begins at step (1).

2.3.6 Instruction Pipeline Operation

As mentioned above, the instruction fetch mechanism works as a pipeline. Figure 2-3 shows how the pipeline works on a sample set of instructions. During the first clock cycle (after reset), the instruction pipeline is flushed, and the first instruction (MOVI) is fetched. During the second clock cycle, the first instruction is executed, and the second instruction (MOVA) is fetched. This continues until a branch instruction (BRA) is reached.

During the branch-to-subroutine (BSR) operation (cycle 4), the instruction following 'BSR' will be fetched, but it will not be executed. During cycle 5 the extra instruction is flushed from the pipeline, and instruction 10 (the first instruction of the subroutine) will be fetched. During cycle 6 the first instruction of the subroutine is executed.

As you can see, all branches require a second clock cycle to 'flush' the instruction pipeline.

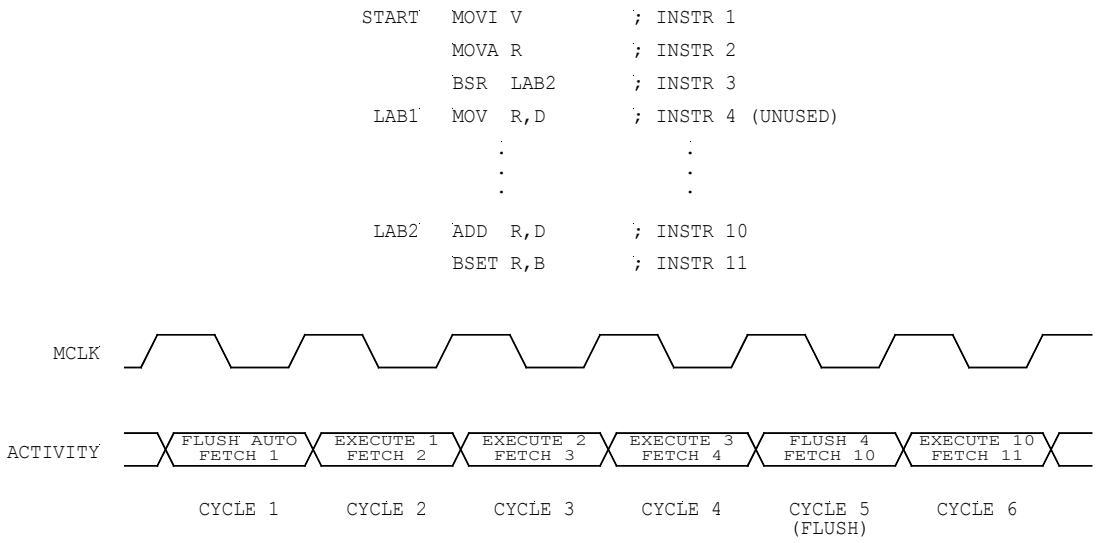


Figure 2-3. Instruction pipeline operation.

2.3.7 Program Memory

Figure 2-4 shows how the program memory (instruction ROM) is organized. The program memory is 2,048 words long, and 12-bits wide. It is split into four banks, each having 512 words of memory. The SLC1657 core uses the banking technique because the architecture has a 12-bit instruction word, which is too short to support program memory addressing longer than nine bits.

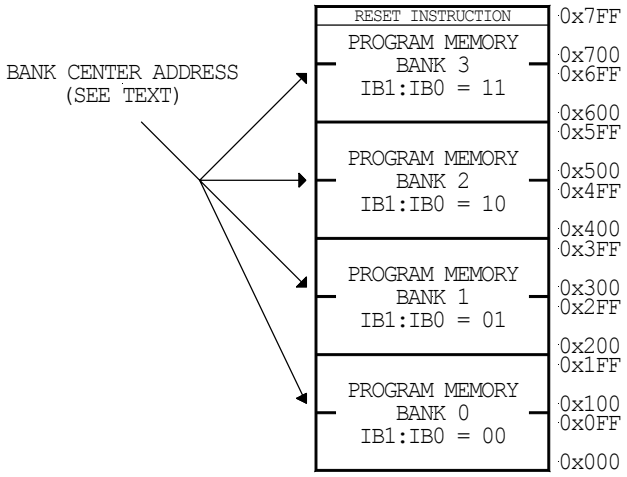


Figure 2-4. Program memory organization.

The SLC1657 core is based upon a smaller device called the SLC1655. The earlier version only supported 512 words of program memory, and was upgraded to support four times the size of the previous memory. To create a larger memory area, the concept of 'banking' is used[10].

Program memory banking in the SLC1657 is accomplished with two instruction bank select bits IB0 and IB1. These are located in the STATUS register, and determine the upper two bits of program memory. These two bits are configured under software control, and are used during the branch (BRA) instruction, the branch-to-subroutine (BSR) instruction and during an update of the PROGCNTR register. For more information, please refer to the section describing the program counter (below).

2.3.8 Reset Instruction

After all resets the instruction pipeline is flushed, and the program counter is forced to address 0x7FF. This causes the instruction at the top of memory (the 'RESET instruction') to be fetched. Generally, this address is programmed with a branch (BRA) instruction, with the branch address being the starting point of the program. However, if a NOP instruction is placed at this address, then the program counter will roll over to address zero.

[10] The banking capability is very similar to the 'mode bit' described in Tracy Kidder's Pulitzer prize winning book: The Soul of a New Machine. There, Kidder describes the development of the MV8000 Eagle computer at Data General Corporation in the late 1970's. The mode bit played a pivotal role in the development of that computer system.

The address of the reset instruction can be changed by modifying the hardware. This is advantageous if less than 2,048 words of program memory are used. For example, if only 512 words are needed, then the reset instruction can be moved to address 0x1FF. For more information please refer to the PROGCNTR entity description located elsewhere in this manual.

2.3.9 Program Counter Operation

The program counter generates the address of the instruction that is fetched from memory.

After a hardware reset, the program counter is forced to 0x7FF. This is the address of the reset instruction.

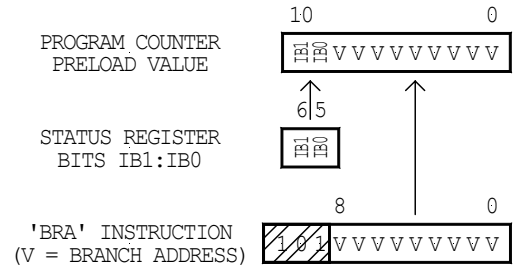
During normal (non-branching) operations, the program counter increments at the end of every cycle.

During branch (BRA) instructions, the program counter is preloaded with a new, 11-bit address. This is shown in Figure 2-5(a). However, the 'BRA' instruction itself only supplies nine of the eleven address bits. The two additional bits are copied from instruction bank select bits 'IB0' and 'IB1', which are located in the STATUS register. These two bits are concatenated with the nine bits in the instruction word to form a complete, 11-bit address. The value is then loaded into the program counter which causes the program to jump to a new location. The instruction pipe is flushed during all branching instructions.

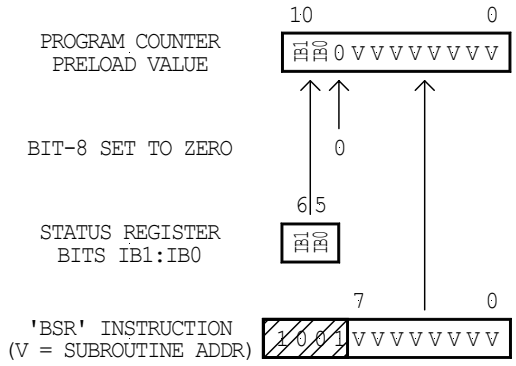
During branch-to-subroutine (BSR) instructions, the current (11-bit) address is pushed onto the stack. A new address is then loaded into the program counter, thereby causing program execution to branch to the subroutine address. This is shown in Figure 2-5(b). The 'BSR' instruction itself only supplies eight of the eleven address bits. Two additional bits are copied from instruction bank select bits 'IB0' and 'IB1', and a third bit is forced to zero. These three bits are concatenated with the eight bits in the instruction word to form a complete, 11-bit address. The value is then loaded into the program counter which causes the program to jump to a new location.

Since bit 8 is forced to zero during the 'BSR' instruction, it also follows that all subroutines must reside in the lower half of a program memory bank. The lower memory is bounded by the 'RAM CENTER ADDRESS' shown in Figure 2-4.

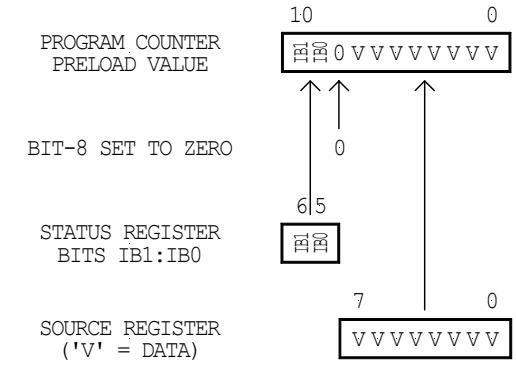
When operating near the end of an instruction bank, it is important to remember that the program counter will roll over from one bank to the next. For example, after fetching a non-branching instruction at address 0x7FF the program counter will roll over to 0x200.



(A) PROGRAM COUNTER PRELOAD ON 'BRA' INSTRUCTION.



(B) PROGRAM COUNTER PRELOAD ON 'BSR' INSTRUCTION.



(C) PROGRAM COUNTER PRELOAD WHEN WRITING TO THE PROGCNTR REGISTER.

Figure 2-5. Program counter operation.

The stack operates like a FILO (first-in, last-out) memory, so that during return-from-subroutine (RET) instructions the oldest stack value is preloaded into the program counter, and the instruction pipeline is flushed. This is shown in Figure 2-6. Subroutines can be called from anywhere in memory because an 11-bit return address is always stored on the stack.

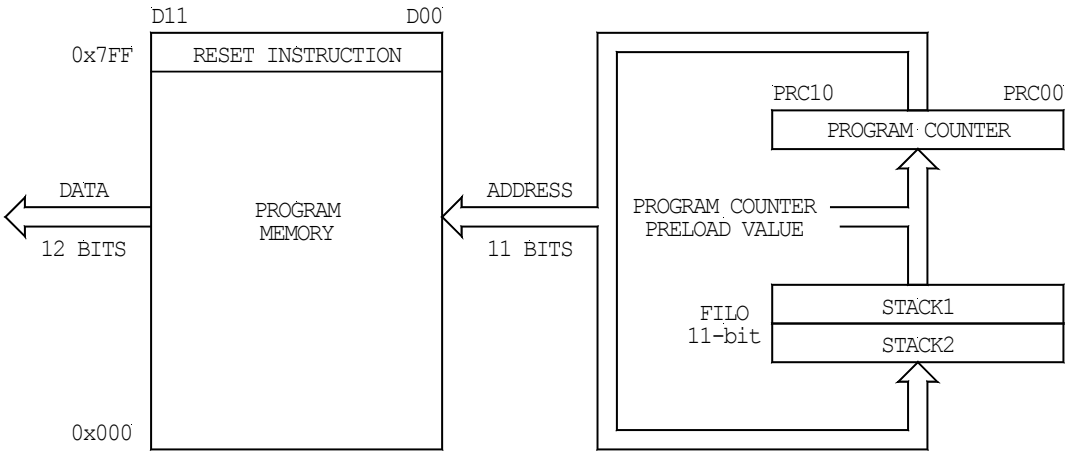


Figure 2-6. Stack operation.

The program counter can also be preloaded by most instructions. This is very useful for relative branch (lookup) tables. Since the program counter can only be preloaded with an 8-bit value from the instruction word, the operation works much like the 'BSR' instruction (i.e. the bits are concatenated in a similar way). This also means that relative branch tables must reside in the lower half of memory. This operation is shown in Figure 2-5(c).

2.3.10 Register Memory

The SLC1657 register memory is broken up into four banks. The register bank is selected by modifying the two register bank selection bits RB0 and RB1 (5 and 6) in the INDEX register.

The four register banks are shown in Figure 2-7. The lower sixteen registers in each bank all map back to BANK 0. The upper sixteen general purpose registers are accessed only from the selected bank

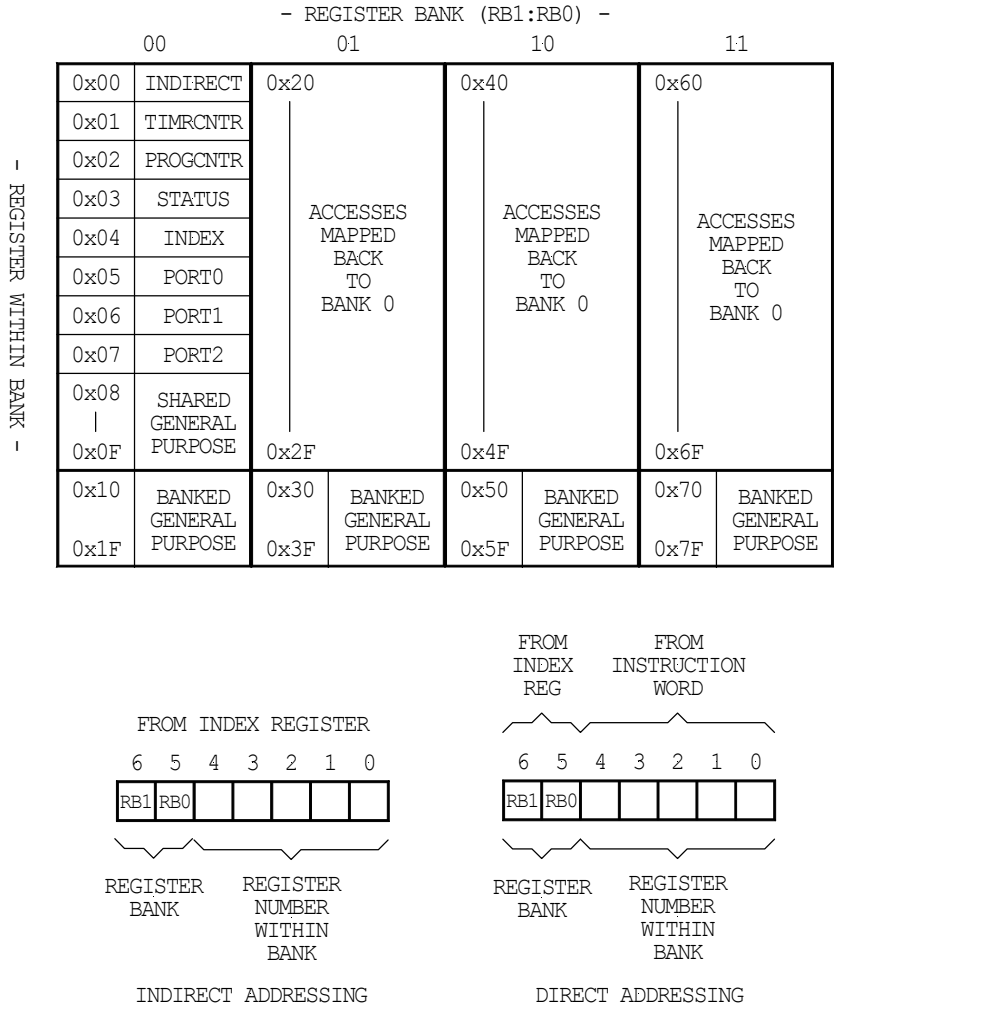


Figure 2-7. Register banking.

When using indirect addressing, register addresses are set up normally in the INDEX register. Subsequent accesses through the INDIRECT register will result in a read or write to the correct register bank.

When using direct addressing, the register number contained in the instruction is concatenated with bits RB1:RB0 in the INDEX register. For example, if a 'MOV 0x10,A' instruction is executed, then the actual register moved to the accumulator will depend upon the state of the RB1:RB0 bits in the INDEX register. If both bits are zero, then the value at register 0x10 will be moved to the accumulator. If RB1:RB0 is '01', then the value at address 0x30 will be moved to the accumulator.

2.3.11 Timer/Counter

A timer/counter functional module can be accessed through the timer/counter (TIMRCNTR) and timer/counter option (TCO) registers. The timer/counter is a general purpose ‘up-counter’ which can be configured to operate off an external or an internal clock. Refer to the timer/counter register descriptions for more details.

3.0 Programming Reference

The SLC1657 has a simple (yet remarkably powerful) instruction set with a total of 32 op-codes. These include add, subtract, increment, decrement, logical, loop and branch instructions.

The SLC1657 has a large base of software tools. The core is instruction compatible with the PIC16C57, a microcontroller made by Microchip Technology Inc. of Chandler, AZ (USA). Assemblers, simulators, 'C' compilers and fuzzy logic generators are available for that device. They are low cost, and are available for a number of operating systems from a variety of suppliers. A partial list of tool and book suppliers are:

- **Assemblers / simulators**:

  microEngineering Labs, Inc.
  Box 7532
  Colorado Springs, CO 80933
  TEL: 719.520.5323
  URL: www.melabs.com

  Microchip Technology, Inc.
  2355 West Chandler Blvd.
  Chandler, AZ USA 85224
  TEL: 602.786.7200
  URL: www.microchip.com

  Parallax, Inc.
  3805 Atherton Road, #102
  Rocklin, CA USA 95765
  TEL: 916.624.8333
  URL: www.parallaxinc.com

- **'C' compilers**:

  B. Knudsen Data (BKD)
  Trondheim, Norway
  URL: www.bknd.com

  Custom Computer Services, Inc.
  Box 2452
  Brookfield, WI 53008
  TEL: 262.797.0455
  URL: www.ccsinfo.com

  Hi-Tech Software LLC
  URL: www.htsoft.com

- **Fuzzy logic Compilers:**

  Inform Software Corporation
  2001 Midwest Road
  Oak Brook, IL USA 60523
  TEL: 630.268.7550
  URL: www.fuzzytech.com

- **Introductory reference books[11]:**

  Easy PIC'n
  PIC'n Up The Pace
  David Benson
  SQUARE 1; P.O. Box 501; Kelseyville, CA USA 95451
  e-mail: sqone@pacific.net
  URL: www.sq-1.com

  Design with PIC Microcontrollers
  John B. Peatman
  Prentice Hall, 1997

  Programming and Customizing the PIC Microcontroller
  Michael Predko
  McGraw-Hill Book Company, 1997

[11] This is a partial list. These books can be ordered through your local bookstore, or on the internet from the Amazon bookstore at <www.amazon.com>. The Amazon website also has book reviews from other users.

3.1 Register Set

A detailed description of the SLC1657 register set is shown in Table 3-1.

3.1.1 Accumulator (ACCUM)

The accumulator (ACCUM) is an implicit register, and is used as a temporary storage location for operands. The contents of the accumulator is undefined at power-up, and is unaffected by reset. Implicit registers are those that are implicitly defined in an instruction word.

3.1.2 Port Control Registers (PC0-2)

The port control registers (PC0, PC1 and PC2) are implicit registers, and determine the state of the associated PCOUT0-2 output pins. If a bit is set to one, then the respective pin is asserted. If a bit is set to zero, the pin is negated. Implicit registers are those that are implicitly defined in an instruction word.

Table 3-1. SLC1657 register set.

REGISTER	ADDRESS		R/W	BIT NUMBER 7	6	5	4	3	2	1	0	RESET VALUE
ACCUM	IMPLICIT		R/W	D7	D6	D5	D4	D3	D2	D1	D0	UUUU UUUU
PC0	IMPLICIT		W	PC0(7)	PC0(6)	PC0(5)	PC0(4)	PC0(3)	PC0(2)	PC0(1)	PC0(0)	1111 1111
PC1	IMPLICIT		W	PC1(7)	PC1(6)	PC1(5)	PC1(4)	PC1(3)	PC1(2)	PC1(1)	PC1(0)	1111 1111
PC2	IMPLICIT		W	PC2(7)	PC2(6)	PC2(5)	PC2(4)	PC2(3)	PC2(2)	PC2(1)	PC2(0)	1111 1111
TCO	IMPLICIT		W		WDT	TCS	TSE	ASGN	PS2	PS1	PS0	-U11 1111
INDIRECT	0x00 (*)		R/W	D7	D6	D5	D4	D3	D2	D1	D0	--------
TIMRCNTR	0x01 (*)		R/W	D7	D6	D5	D4	D3	D2	D1	D0	UUUU UUUU
PROGCNTR	0x02 (*)		R/W	D7	D6	D5	D4	D3	D2	D1	D0	1111 1111
STATUS	0x03 (*)		R/W		IB1	IB0	TO	PD	Z	NC	C	-00T TUUU
INDEX	0x04 (*)		R/W		RB1(D6)	RB0(D5)	D4	D3	D2	D1	D0	1UUU UUUU
PORT0	0x05 (*)		R/W	PT0(7)	PT0(6)	PT0(5)	PT0(4)	PT0(3)	PT0(2)	PT0(1)	PT0(0)	UUUU UUUU
PORT1	0x06 (*)		R/W	PT1(7)	PT1(6)	PT1(5)	PT1(4)	PT1(3)	PT1(2)	PT1(1)	PT1(0)	UUUU UUUU
PORT2	0x07 (*)		R/W	PT2(7)	PT2(6)	PT2(5)	PT2(4)	PT2(3)	PT2(2)	PT2(1)	PT2(0)	UUUU UUUU
SHARED GENERAL PURPOSE	0x08 (*) | 0x0F (*)		R/W | R/W	D7 | D7	D6 | D6	D5 | D5	D4 | D4	D3 | D3	D2 | D2	D1 | D1	D0 | D0	UUUU UUUU | UUUU UUUU
BANKED GENERAL PURPOSE	0x10 | 0x1F	BANK 0	R/W | R/W	D7 | D7	D6 | D6	D5 | D5	D4 | D4	D3 | D3	D2 | D2	D1 | D1	D0 | D0	UUUU UUUU | UUUU UUUU
	0x30 | 0x3F	BANK 1	R/W | R/W	D7 | D7	D6 | D6	D5 | D5	D4 | D4	D3 | D3	D2 | D2	D1 | D1	D0 | D0	UUUU UUUU | UUUU UUUU
	0x50 | 0x5F	BANK 2	R/W | R/W	D7 | D7	D6 | D6	D5 | D5	D4 | D4	D3 | D3	D2 | D2	D1 | D1	D0 | D0	UUUU UUUU | UUUU UUUU
	0x70 | 0x7F	BANK 3	R/W | R/W	D7 | D7	D6 | D6	D5 | D5	D4 | D4	D3 | D3	D2 | D2	D1 | D1	D0 | D0	UUUU UUUU | UUUU UUUU

NOTES: 'U': UNCHANGED AFTER ANY RESET; 'T': CHANGED DEPENDING UPON TYPE OF RESET (REFER TO REGISTER DESCRIPTION FOR DETAILS); 'R/W': READ/WRITE REGISTER; 'W': WRITE ONLY REGISTER; '-': UNDEFINED; HATCHED AREAS INDICATE UNUSED BIT.

BANK NUMBER IS SELECTED BY BANK SELECT BITS BS1:BS0 IN STATUS REGISTER.

(*) ASTERISK INDICATES ADDRESS IS A FUNCTION OF BANK SELECT BITS BS1:BS0. FOR EXAMPLE, WHEN BS1:BS0 IS '00', THEN THE STATUS REGISTER ONLY APPEARS AT ADDRESS 0x03. HOWEVER, WHEN BS1:BS0 IS '10', THEN THE STATUS REGISTER APPEARS AT ADDRESS 0x03 AND 0x43. SEE TEXT FOR DETAILS.

The port control registers are *write-only*, and can only be accessed with the MOVP instruction. The op-code for the MOVP instruction maps registers PC0, PC1 and PC2 into implicit address spaces 5, 6 and 7 respectively.

After reset, the contents of PC0-2 are set to one. This is an important feature when the I/O ports are used in the bi-directional three-state mode. In this mode the I/O ports are placed in their high impedance states after reset since it is not known if the ports are connected to the inputs or outputs of external logic.

When the port control outputs are used in bi-directional three-state mode, each bit is generally assigned to the corresponding pin in the PTOUT0-2 buses. If the PC0-2 register bit is set to one, then the respective pin is placed in high impedance (three-state) mode. If the bit is set to zero, then the pin is enabled as an output.

For more information about the PCO-2 registers, please refer to the *I/O Port Options* section below.

3.1.3 Timer/counter Option Register (TCO)

The timer/counter option register (TCO) is an implicit register, and selects the timer/counter, prescaler and watchdog enable options. Table 3-2 shows how to program the register. The timer/counter option register is 'write-only', and can only be accessed with the MOVT instruction. All of the bits of the timer/counter option register, except for the WDT bit, are set to '1' after the assertion of any reset. The WDT bit is unaffected by reset.

Bit D7 is unused and reserved for future use. It should be set to zero for forward compatibility.

Bit D6 is the watchdog timer enable bit (WDT), and causes the watchdog to be enabled or disabled. When set to one, the watchdog is enabled. When set to zero, the watchdog is disabled. Reset does not affect the bit.

Bit D5 is the timer/counter select bit (TCS), and determines the signal source for the timer/counter. When set to a one, the source of the timer/counter is the external [TMRCNT] pin. When set to a zero, the source of the timer/counter is the microcontroller clock divided by four (i.e. [MCLK] / 4).

Table 3-2. Timer/counter option register (TCO).

Bit No.	Mnemonic	Description
D7	-	Unused/reserved (set to zero).
D6	WDT	Watchdog timer enable: 1: Watchdog enabled 0: Watchdog disabled
D5	TCS	Timer/counter clock source: 1: TMRCNT 0: Internal clock [MCLK / 4]
D4	TSE	TMRCNT edge select: 1: Positive edge 0: Negative edge
D3	ASGN	Prescaler assignment: 1: Assign to watchdog timer 0: Assign to timer/counter
D2, D1, D0	PS2, PS1, PS0	Prescaler divider rate.

Bit D4 is the TMRCNT edge select (TSE), and determines which edge increments the timer/counter. It is only used when the clock source is the external TMRCNT signal (i.e. TCS = '1'). When set to a one, the positive edge is used. When set to zero, the negative edge is used.

Bit D3 is the prescaler assignment (ASGN), and determines whether the prescaler is assigned to the timer/counter or watchdog circuits. The prescaler cannot be assigned to both. When set to a one, the prescaler is assigned to the watchdog timer. When set to a zero, the prescaler is assigned to the timer/counter.

It is recommended that the watchdog timer be cleared before changing the prescaler assignment bit (by executing a 'RWT' instruction). This will prevent an unwanted watchdog time-out.

Bits D2, D1 and D0 are Prescaler select bits (PS2, PS1 and PS0), and determine the division ratio of the prescaler. They should be set as shown in Table 3-3. The prescaler is essentially a divide-by-N counter, where 'N' is the value selected by PS2, PS1 and PS0.

Also note that there are two prescaler division ratios listed: one for the TIMRCNTR and one for the watchdog. This is because the prescaler operates as a binary up counter. When attached to the TIMRCNTR, the prescaler generates an output clock which is used as a clock source to the TIMRCNTR counter. When attached to the watchdog, the prescaler provides a pulse 'level' which triggers the watchdog. [Stated another way, the TIMRCNTR relies on the edges that the prescaler supplies, whereas the watchdog relies on the output level from the prescaler].

Table 3-3. Prescaler select bits.

PS2	PS1	PS0	TIMRCNTR Prescaler Division Ratio	WATCHDOG Prescaler Division Ratio
0	0	0	2	1
0	0	1	4	2
0	1	0	8	4
0	1	1	16	8
1	0	0	32	16
1	0	1	64	32
1	1	0	128	64
1	1	1	256	128

For example, if the positive edge of the [TMRCNT] signal is used (TCS = '1' and TSE = '1), the prescaler is assigned to the timer/counter (ASGN = '0'), a divide-by-8 prescaler is needed and the watchdog timer is disabled, then the TCO register should be set to 0x32.

For more information on the TCO register please refer to the *Timer/counter Operation* section below.

3.1.4 Indirect Register (INDIRECT)

The indirect register (INDIRECT) causes reads or writes to the current address loaded in the INDEX register. It is used for software 'pointers'. The INDIRECT register actually isn't a register at all...it simply causes accesses to other registers.

For example, if the INDEX register contains 0x10, then reading the INDIRECT register at address 0x00 will return the value at location 0x10. In assembly code this would look something like:

```
MOVI    0x14        ; Load accumulator with 0x14
MOVA    0x10        ; Store at address 0x10

MOVI    0x10        ; Load accumulator with 0x10
MOVA    INDEX       ; Store in the INDEX register

MOV     INDIRECT,A  ; Accumulator now contains 0x14
```

If the INDEX register contains 0x00, then reading the INDIRECT register will return a value of 0x00.

For more information about this register (including operation with respect to register banks), please refer to the description of the INDEX register elsewhere in this manual.

3.1.5 Timer/counter Register (TIMRCNTR)

The timer/counter register (TIMRCNTR) loads or returns the eight-bit timer/counter. The TIMRCNTR register is unchanged after any reset.

Reading the TIMRCNTR register returns the value of the timer/counter at the rising [MCLK] edge at the beginning of the cycle. Writing to the TIMRCNTR register loads it at the rising clock edge at the end of the cycle.

For more information on the TIMRCNTR register please refer to the *Timer/counter Operation* section below.

3.1.6 Program Counter Register (PROGCNTR)

The program counter register (PROGCNTR) is used to read and write to the *lower eight bits* of the program counter. The program counter is actually eleven bits wide...so the most significant bits are not accessible from this register.

The PROGCNTR register is set to 0xFF after any reset, and increments one count after every instruction (or two counts after a branch instruction).

Reading the program counter will return the lower eight bits of the address following the instruction. That's because the instantaneous value of the program counter reflects the prefetch address (i.e. the address following the 'read' instruction). For example, the following instruction word located at address 0x56 will load the accumulator with 0x57:

```
0x056          MOV   PROGCNTR,A        ; Move PROGCNTR to accumulator
```

Writing to the program counter preloads it with a new address, and causes a branch. This activity always takes two clock cycles, as the instruction stream must be flushed.

Preloading the program counter is very useful for relative branch (i.e. lookup) tables. When the program counter is preloaded, a new 8-bit address is stored in the program counter, and the instruction pipeline is flushed. If an offset value is added to the program counter, and then stored back into the program counter, then a relative branch will occur.

During program counter preloads, the two most significant bits are set to the value of IB1:IB0 in the STATUS register. Bit 8 is always forced to zero. This means that relative branch tables must reside in the lower half of memory.

For example, consider a lookup table for a sinewave generator. In this example, we have a lookup table with thirty-two sinewave entries. A subroutine is created (in the lower half of memory) which returns a unique sinewave value that depends upon a count between zero and thirty-one. The count value is passed to the subroutine in the accumulator. In the subroutine, the accumulator is added to the program counter, which causes a relative branch to a RET instruction. The RET instruction allows an immediate value to be loaded into the accumulator before returning. Therefore, the count value in the accumulator is converted to a sinewave value, and is then returned in the accumulator thusly:

```
            MOV     COUNT,ACCUM     ; Get the sinewave count (0 ≤ COUNT ≤ 31)
            BSR     GETSINE         ; Go get the lookup value
            .                       ; Return here with lookup value in accumulator
            .
            .
GETSINE     ADD     PC,1            ; Add accumulator to the program counter
            RET     0x00            ; Return with lookup value in accumulator
            RET     0x31            ;       .
            RET     0x61            ;       .
            RET     0x8D            ;       .
            .                       ;       .
            .                       ;       .
            .                       ;       .
```

For more information, refer to the description of the PROGCNTR entity elsewhere in this manual. For more information about the relationship between the program counter and the bank selection bits, please see the section of this manual describing bank selection.

3.1.7 Status Register (STATUS)

The status register (STATUS) is used to monitor the status bits, set the memory bank and to monitor the power up status. All of the bits (except TO and PD) are accessible by reads or writes. The TO and PD bits are read-only. Table 3-4 summarizes the bits in the STATUS register.

Bit D7 is unused and is reserved for future use. It should be set to zero for forward compatibility.

Bits D5 and D6 are the instruction bank select bits IB0 and IB1 respectively. These two bits operate together to set the two most significant bits of program memory space during branch (BRA), branch-to-subroutine (BSR) and program counter preload operations.

Bit D4 is the timeout bit (TO), and indicates whether or not a watchdog reset has occurred. The bit is always set after a power-up reset or an external reset after a PWRDN instruction. It is cleared after a watchdog reset. TO is a read-only bit.

Bit D3 is the power-down bit (PD), and indicates whether or not a reset has occurred after a PWRDN instruction. It is set after a power-up reset or a non-PWRDN watchdog reset. It is always cleared after a PWRDN instruction. PD is a read-only bit.

The TO and PD bits can be used to determine the source of a reset. It is recommended that they be used together as shown in Table 3-5.

The 'TO' and 'PD' bits are both set after a emulation ROM programming reset [PRESET]. This mimics a power-up reset after downloading new code.

For more information on using the TO and PD bits please refer to the *Timer/counter Operation* section below.

Table 3-4. Status register (STATUS).

Bit No.	Mnemonic	Description
D7	-	Unused/reserved (set to zero).
D6	IB1	Instruction bank select bit 1
D5	IB0	Instruction bank select bit 0
D4	TO	Timeout (read only): 1: After power-up reset, RWT or PWRDN instruction 0: After a watchdog timeout
D3	PD	Power-down (read only): 1: After power-up reset or RWT instruction 0: After a PWRDN instruction
D2	Z	Zero bit (read/write): 1: Result of the operation is zero. 0: Result of the operation is non-zero.
D1	NC	Nibble-carry bit (read/write): 1: ADD - carry from bit D3 did occur SUB - borrow to bit D3 did not occur 0: ADD - carry from bit D3 did not occur SUB - borrow to bit D3 did occur
D0	C	Carry bit (read/write): 1: ADD - carry from bit D7 did occur SUB - borrow to bit D7 did not occur ROL/R - '1' shifted from D7/D0 respectively 0: ADD - carry from bit D3 did not occur SUB - borrow to bit D3 did occur ROL/R - '0' shifted from D7/D0 respectively

Table 3-5. TO and PD bits after reset.

TO	PD	Reset Type
0	0	Watchdog reset (from PWRDN)
0	1	Watchdog reset (non-PWRDN)
1	0	External reset (from PWRDN)
1	1	Power-up or other reset or PRESET

Bits D2, D1 and D0 are the he zero (Z), nibble-carry (NC) and carry (C) bits respectively. They indicate the result of some arithmetic and logical operations. Refer to the individual instruction descriptions for more information.

Normally, these bits are set by the arithmetic logic unit (ALU). However, they can also be changed by writing to the STATUS register. In this case, the result presented by the ALU has precedence over the write data itself. For example, a CLR 0x03 instruction will result in the 'Z' bit being set. For this reason, writing to the STATUS register 'Z', 'NC' and 'C' bits should be carefully evaluated. Instructions that do not set the condition code bits (such as BCLR and BSET) are recommended for this case.

For more information, refer to the descriptions of the STATSREG entity, RESETGEN entity, ALULOGIC entity, TCO register and instruction descriptions located elsewhere in this manual. For more information about bits IB1:IB0, please see the sections on bank selection located elsewhere in this manual.

3.1.8 Index Register (INDEX)

The index register (INDEX) is used to perform indirect addressing, and to indicate which register bank is specified.

During indirect addressing it is used in conjunction with the INDIRECT register. To access a value indirectly, the INDEX register is loaded with a seven bit address. Subsequent accesses to the INDIRECT register (at address 0x00) will then access the location pointed to by INDEX. For example, if the INDEX register contains 0x10, then reading the INDIRECT register at address 0x00 will return the value at location 0x10. This function is generally used for software *pointers*.

During normal (non indirect) addressing modes, bits D5 and D6 of the INDEX register are used as register bank selection bits RB0 and RB1. For example, if RB1:RB0 = 0:0, an access to address 0x10 will return the banked general purpose register at address 0x10. However, if RB1:RB0 = 0:1, then an access to address 0x10 will return the banked general purpose register at address 0x30.

The INDEX register is unchanged after reset. The unimplemented bit of the INDEX register always read as '1'.

For more information, refer to the description of the INDIRECT register elsewhere in this manual.

3.1.9 Port Registers (PORT0-2)

The port registers PORT0, PORT1 and PORT2 are used to access the I/O ports. Writing to the port register sets the output value on signals [PTOUT0-2(7..0)]. Reading the port returns the value on [PTIN0-2(7..0)]. Each bit in these registers accesses the corresponding I/O bit. If the bit is set to a one, then the respective output pin is set to one. If the bit is set to zero, then the pin is set to zero. The converse is also true when reading an input port.

For more information about the PORT0-2 registers please refer to the *I/O Port Options* section below.

3.1.10 General Purpose Registers (GENERAL PURPOSE)

The general purpose registers are used for data storage, and are eight bits wide. They are not affected by reset. There are two types of general purpose registers: shared and banked.

The SHARED GENERAL PURPOSE registers are accessible from all four memory banks, regardless of the state of bits RB1 and RB0 in the INDEX register. For example, reading the register at address 0x28 will actually cause a read from address 0x08.

The BANKED GENERAL PURPOSE registers are accessible only from the current register bank, as selected by the register bank select bits RB1 and RB0 in the INDEX register.

3.2 Reset Operation

There are three ways to reset the SLC1657 core. These are (a) with the external reset pin [RESET], (b) with the emulation ROM reset pin [PROG* / PRESET] and (c) by the watchdog reset. All resets generate the following activity:

- All bits in the port control registers PC0-2 are set to '1'.

- The state of STATUS register bits ‘TO’ and ‘PD’ are selected to reflect the source of the reset.

- If the [PRESET] input is the source of the reset, then the ‘TO’ and ‘PD’ are both set (thereby indicating a ‘power-up’ reset). The download circuit is also enabled.

- The watchdog timer is reset. All bits in the TCO register (with the exception of the watchdog enable bit WDT) are set to ‘1’.

- The program counter is preset to 0x7FF. Note that addresses other than 0x7FF are supported by modifications to the BUC11CPP (VHDL hardware) entity. This entity is used by the program counter (PROGCNTR) entity, and contains logic that generates the reset address. Changing this address is especially useful if less than four program memory banks are used in the final design. For example, if only 512 program words are needed, the BUC11CPP entity can be modified so that the reset address is at 0x1FF. This allows ¾ of the memory to be eliminated in the final design.

- The instruction bank select bits IB1:IB0 in the STATUS register are cleared.

- The instruction pipeline is flushed. This is done by clearing the instruction stream, thereby causing a ‘NOP’ instruction to be fetched before the reset instruction.

For more information about reset, please refer to RESETGEN entity and the register descriptions.

3.3 I/O Port Options

There are several ways that the I/O port pins can be implemented in the design. Figure 3-1 shows three typical I/O configurations. These include the bi-directional, single-ended three-state and single-ended I/O configurations.

The core also generates output port strobes PTSTB0-2. These are useful when interfacing to external FIFOs and other devices. Each strobe corresponds to each output port. These strobes are active for one [MCLK] clock cycle.

Reset does not affect the input or output port signals (except on power-up, where the output ports initialize to the power-up state of D-type flip-flops in the target device). If needed, the state of the output ports during reset should be selected with external logic.

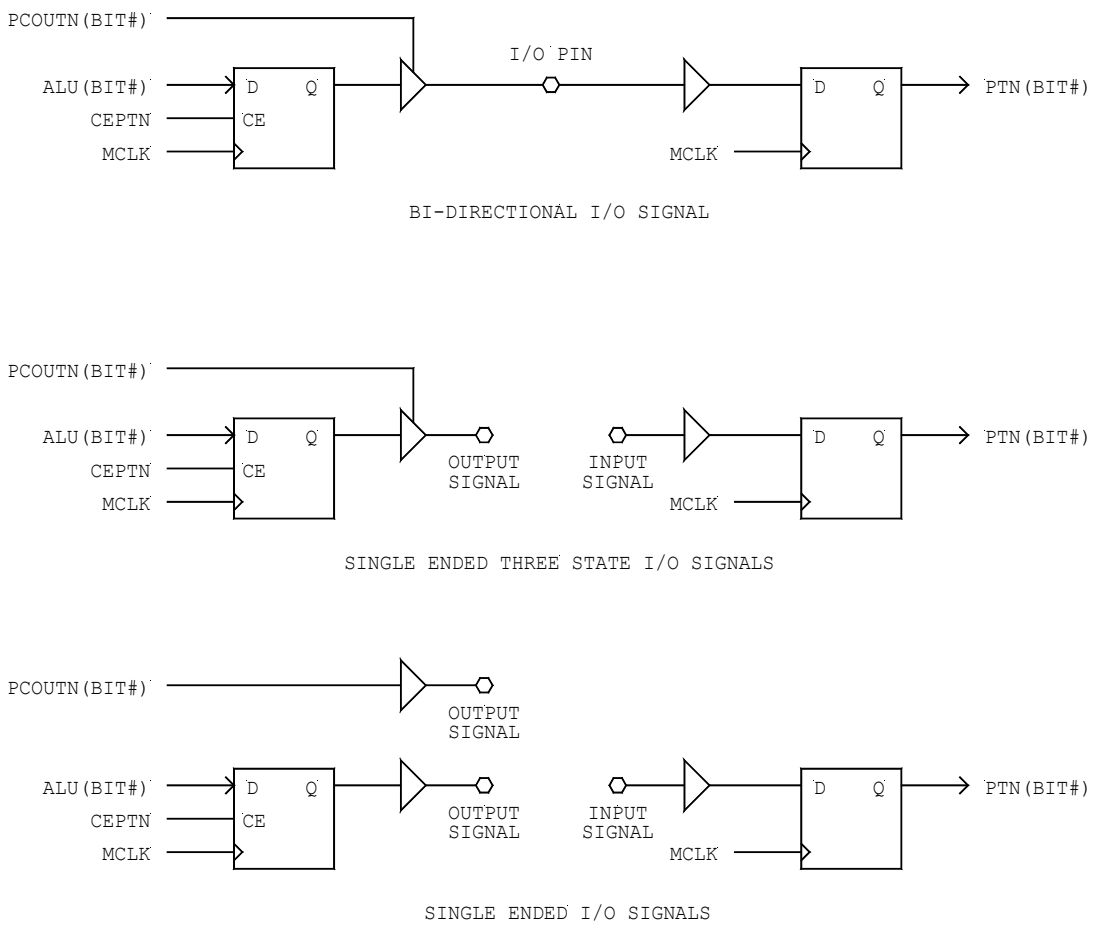


Figure 3-1. I/O port configuration options. The flip-flops shown in the diagram are part of the PORTSREG entity. The I/O buffers are added by the user in the final implementation.

For example, when operating a port in the bi-directional three-state mode (using external logic), then the ports will reset in the three-state condition.

The user must exercise some caution when doing back-to-back writes and reads to the same port register. During writes, the port bits become active just after the positive [MCLK] edge at the *end* of the cycle. During reads, the port bits are latched at the [MCLK] edge immediately *before* the instruction is executed. Therefore, in the bi-directional three configuration the data written to a port isn't valid at the very next instruction.

For example, consider the timing diagram of Figure 3-2. Here we assume that the port is operated in the bi-directional three-state configuration, and that all of the PORT0 bits are in output mode (i.e. PC0 = 0x00). Sometime before the beginning of a code sequence the PORT0 output is 0x57. Writing 0x38 to PORT0 causes the new output data to become

active just after the rising edge of [MCLK] at the end of the cycle. However, since the port input data is sampled on the very same edge, the new data isn't available yet. If PORT0 is read immediately after writing to it, the old value of 0x57 is still obtained. Reading the port a second time causes the new value to be read.

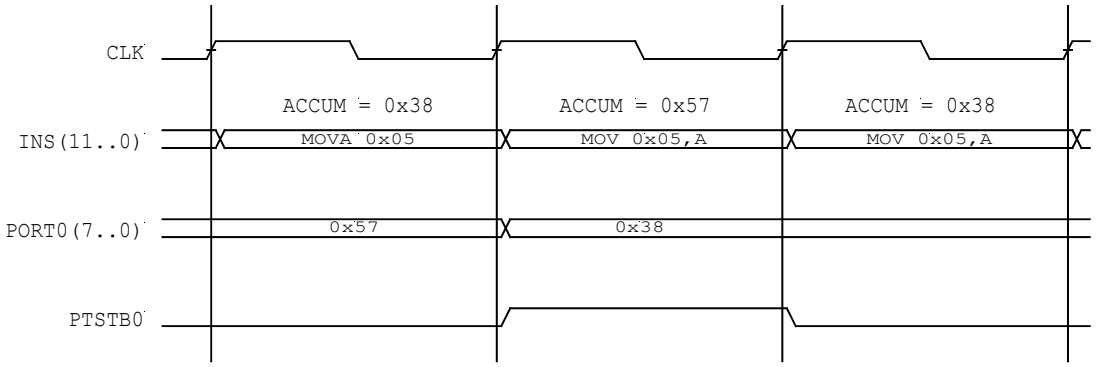


Figure 3-2. Back-to-back PORT0 write and read operations in the bi-directional three-state configuration.

This doesn't imply that each port must be read twice...it just means that input and output data are latched at the very same instant.

For more information please refer to the descriptions of the PORTSREG entity, PC0-2 register, PORT0-2 register and MOVP instruction descriptions elsewhere in this manual.

3.4 Timer/counter & Watchdog Operation

The 8-bit timer/counter is used for general purpose time interval and pulse counting functions. Furthermore, the timer/counter prescaler can be re-assigned to work with the watchdog timer.

3.4.1 Timer Operation

When operated as a timer, the timer/counter is used for general purpose time interval measurements. In this mode, the TIMRCNTR register increments whenever an edge from the clock source or the prescaler occurs. The time base can be derived from the internal clock source [MCLK / 4], or from the external [TMRCLK] pin.

In this mode the timer is usually operated as an elapsed time indicator. At the beginning of the time interval the TIMRCNTR register is cleared, and is then periodically checked to see if the time interval has elapsed.

For example, let's assume that we need to determine when a 1.0 millisecond time period has elapsed. Furthermore, let's assume that we're using the internal [MCLK] source as a time base, that [MCLK] is operating at 5.00 MHz and that the watchdog timer is disabled.

The first step is to initialize the timer/counter option register (TCO). In this case we'll set the WDT bit to '0' (i.e. watchdog disabled), the TCS bit to '0' (i.e. internal clock selected), the TSE bit to '1' (i.e. positive edge select) and the ASGN bit to '0' (i.e. prescaler assigned to the timer/counter). The prescaler divider rate is selected so that an adequate granularity[12] of the clock is obtained. To find the prescaler divider rate we first look at the time base frequency, which is:

$$\text{Time base frequency} = \text{MCLK} / 4 = 5.00 \text{ MHz} / 4 = 1.25 \text{ MHz}$$

This means that the TIMRCNTR register (without the prescaler) will increment at the rate of 1.25 MHz. Since the target rate is about 0.001 MHz (1 / 1.0 ms = 0.001 MHz). The prescaler divisor rate is then:

$$1.25 \text{ MHz} / 0.001 \text{ MHz} = 1{,}020$$

This means that the clock rate has to be stepped down by a factor of 1,020 to get the TIMRCNTR register to tick over at 1.0 millisecond intervals.

To make the software a little simpler, the prescaler value is selected so that the 6th bit (D05) of the TIMRCNTR register is asserted at the end of the 1.0 millisecond interval. This gives a prescale value of:

$$1{,}020 / 2^6 = 1{,}020 / 64 = 15.9 \cong 16$$

Therefore, the prescaler value will be 16, or PS2,1,0 = 0,1,1. This gives a TCO configuration value of B00010011 = 0x13.

Also note that the numbers don't work out evenly to 1.0 millisecond. That would require a time base with an even multiple of 1.0 millisecond. The actual tick rate of the TIMRCNTR register is:

$$\text{Tick rate} = (1 / 1.25 \text{ MHz}) \times (64 \times 16) = 0.8 \text{ millisecond}$$

Here's a sample program that initializes the timer/counter in timer mode (with the parameters just calculated) and performs a section of code every millisecond:

[12] By 'granularity', it is meant that TIMRCNTR register will 'tick-over' at reasonable times.

```
START   MOVI    0x13        ; Initialize TCO register
        MOVT
        CLR     TC          ; Clear the timer/counter register

CHECK   BTSC    TC,5        ; Check if timer/counter bit D05 is set
        BRA     ONE_MS      ; Branch if a millisecond has passed
        .
        .                   ; Other code
        .
        BRA     CHECK       ; Loop back

ONE_MS  CLR     TC          ; Clear the timer/counter register
        .
        .                   ; Activity to be performed every millisecond
        .
        BRA     CHECK       ; Loop back
```

3.4.2 Counter Operation

When operated as a counter, the timer/counter is used for pulse counting. In this mode, the TIMRCNTR register increments whenever an edge from the clock source or the prescaler occurs. Although the count can be derived from the internal clock source [MCLK / 4], it is generally obtained from the external [TMRCLK] pin in this mode.

For example, let's assume that the timer/counter is used to measure the number of incoming pulses from a shaft encoder. [A shaft encoder is simply an optical or magnetic pick-up on a rotating shaft]. Furthermore, let's assume that the program drops into a routine every time the shaft turns past the encoder element.

In this example, we'll monitor the negative edge of the shaft encoder signal, and that the watchdog is enabled and assigned to prescaler. This means the program will look something like this:

```
START   MOVT    0x68        ; Initialize TCO register
        CLR     TC          ; Clear the timer/counter register

CKSHFT  MOV     TC,ACCUM    ; Check if the timer/counter has incremented
        BTSS    STATUS,Z    ; from zero.
        BRA     SNSE_ENC    ; Branch if the encoder has been sensed
        .
        .                   ; Other code
        .
        BRA     CKSHFT      ; Loop back

SNSE_ENC CLR    TC          ; Clear the timer/counter register
        .
        .                   ; Activity to be performed every millisecond
        .
```

```
        BRA         CKSHFT      ; Loop back
```

3.4.3 Watchdog Operation

The watchdog timer is used in two different ways (separately or in combination). As a failure recovery mechanism, the watchdog resets the microcontroller if something has gone wrong (either hardware or software). As a wake-up mechanism, the watchdog resets the microcontroller after a suitable power-down interval. This reduces power consumption.

The watchdog is formed from a 15-bit ripple counter. The counter is driven by the microcontroller clock divided by sixteen [MCLK / 16]. For example, if the microcontroller clock operates at 5.00 MHz, the watchdog timeout period will be:

$$\text{Timeout period} = [1 / (5.00\ \text{MHz} / 16)] \times 2^{(15-1)} \cong 52\ \text{milliseconds}$$

If a longer time-out period is needed, then the output from the watchdog can be routed through the 8-bit prescale counter. For example, the longest watchdog timeout on a 5.00 MHz microcontroller is:

$$\text{Maximum timeout period} = \{[1 / (5.00\ \text{MHz} / 16)] \times 2^{(15)}\} \times 128 \cong 13.4\ \text{seconds}$$

The watchdog ripple counter is cleared in response to a reset or the RWT (reset watchdog timer) instruction. This also clears the prescale counter (if it is attached to the watchdog).

When programming the SLC1657 it is recommended that the watchdog timer be enabled or disabled immediately after a power-up reset. The power-up reset condition can be determined by reading the TO and PD bits in the STATUS register. This discriminates the power-up reset condition from, say, a wakeup reset after PWRDN.

It should be also noted that the TCO register is a write-only register, and that the bit set and bit clear instructions will not work on it. For this reason all of the bits must be set simultaneously.

For example, if the watchdog is to be enabled, then the following code will enable it (once) only after the power-up reset:

```
START:   BTSS    STATUS, 3   ; Test the PD bit
         BRA     CONT        ; Branch if PD = '0'
         BTSS    STATUS,4    ; Test the TO bit
         BRA     CONT        ; Branch if TO = '0'

WEBL:    MOVI    0x7F        ; Set the watchdog enable bit
         MOVT                ; Store it in the TCO register

CONT:  ...                   ; ...and continue
```

If the watchdog is to be disabled, then the following code will disable it (once) only after the power-up reset:

```
START:   BTSS    STATUS,3    ; Test the PD bit
         BRA     CONT        ; Branch if PD = '0'
         BTSS    STATUS,4    ; Test TO bit
         BRA     CONT        ; Branch if TO = '0'

DABL:    MOVI    0x3F        ; Disable the watchdog timer
         MOVT                ; Store it in the TCO register

CONT:  ...                   ; ...and continue
```

In some applications it is desirable that the watchdog be permanently enabled or disabled. This eliminates any possibility that the watchdog can be inadvertently enabled or disabled. In these cases the VHDL source file in the TCOPTREG entity[13] should be changed so that the WDT bit is permanently set or reset.

For more information, refer to the descriptions of the TIMRCNTR entity, the MOVT instruction, the TIMRCNTR register and the STATUS register located elsewhere in this manual.

3.4.4 Changing the Prescale Register

The prescale counter can be changed under software control. Since this function is shared by the timer/counter and the watchdog timer, it is possible to generate an unintended watchdog reset when changing the value of the prescaler. To avoid this problem, it is recommended that the prescaler be changed using the guidelines described in this section.

[13] The VHDL source file for the TCOPTREG entity contains instructions for permanently enabling or disabling the watchdog timer.

When changing the prescaler *from the timer/counter to the watchdog timer*, it is recommended that the following code sequence be used:

```
RWT                          ; Reset the watchdog timer and prescaler
CLR     TIMRCNTR             ; Clear the timer/counter register
MOVI    B'00XX1111'          ; Set the prescale register  to highest level
MOVT
RWT                          ; Reset the watchdog timer and prescaler
MOVI    B'00XX1CCC'          ; Set prescaler to new division rate ('CCC')
MOVT
```

When changing the prescaler *from the watchdog timer to the timer/counter*, it is recommended that the following code sequence be used:

```
RWT                          ; Reset the watchdog timer and prescaler
MOVI    B'XXXX0CCC'          ; Select TIMRCNTR and new prescale value
MOVT
```

3.5 Power-down Operation

The SLC1657 core has a special power-down feature that allows it to reduce power consumption. This is especially useful in low current or battery powered applications. A special PWRDN instruction causes the microcontroller to halt operation, thereby reducing current consumption. The actual reduction in power consumption depends upon the clock frequency and quiescent current consumption of the target FPGA or ASIC device.

After a PWRDN instruction the core is powered down until a reset occurs. The watchdog timer and prescaler (if assigned to the watchdog) are cleared. During power-down the microcontroller clock [MCLK] continues to operate[14], but no instructions are fetched.

The 'PD' and 'TO' STATUS register bits are also affected by the PWRDN instruction. This allows the reset handler routine to determine if the reset is caused by a reset in response to a PWRDN instruction.

The [SLEEP] signal is asserted in response to the PWRDN instruction. This allows external devices to be powered down at the same time. The [SLEEP] signal can also be used to suspend [MCLK] external to the core. This will further reduce power consump-

[14] It is possible to alter the SLC1657 design so that even [MCLK] is suspended during the PWRDN condition. In the standard SLC1657, [MCLK] is used by the watchdog timer (and emulation ROM input pins) to determine when to 'wake-up' from the PWRDN condition. In very low power applications it may be useful to shut off [MCLK] during this interval as well. This will reduce even more power. In this case a separate clock pin must be supplied for the watchdog timer. Contact the factory for details.

tion. However, care should be taken when suspending [MCLK], as this will cause the watchdog timer to stop operating. In this case the external circuit must provide some timing mechanism to restart both the [MCLK] signal and reset the microcontroller.

Let's look at an example that uses both the RWT and PWRDN instructions. This is an example where the microcontroller performs a setup routine once after power-up. After the initial setup, the microcontroller goes to sleep, and periodically wakes up to perform some task. We'll assume a 5.00 MHz microcontroller where the watchdog timeout is about 50 milliseconds. This means that the microcontroller will wake up about every 50 milliseconds, perform the task, and then put itself back to sleep.

The program resets to the 'RESET' label. The first four statements determine if the reset was caused by a watchdog reset from the power-down condition. If it is, then both the 'TO' and 'PD' bits are zero, it jumps to the 'WAKEUP' label, and performs the wakeup routine. Otherwise it assumes that a power-up, external or external reset has occurred. During the power-up sequence a different set of code (including watchdog initialization) is performed:

```
RESET:      BTSC      STATUS, 3     ; Test the PD bit
            BRA       PWRUP         ; Branch if PD ≠ '0'
            BTSS      STATUS,4      ; Test the TO bit
            BRA       WAKEUP        ; Branch if TO = '0'

PWRUP:      MOVI      0x7F          ; Power-up program sequence
            MOVT                    ; Enable the watchdog timer
            .
            .                       ; Additional initialization instructions
            .

WAKEUP:     .                       ; Wakeup program sequence
            .
            .                       ; Application code
            .
            RWT                     ; Perform an RWT instruction if the
            .                       ; wakeup sequence takes longer than
            .                       ; the watchdog timeout period.
            .
            .                       ; Application code
            .
ALLDONE     PWRDN                   ; Execute the power-down command
```

3.6 Compatibility with the Microchip Part

The SLC1657 maintains a high degree of compatibility with the Microchip PIC16C57 part. While all instructions are compatible, there are some differences between the two architectures. These include:

1) The watchdog timer in Microchip part is enabled via a special register in the ROM area. However, this creates quite a nuisance in portable cores, and potential hardware non-portability. In the SLC1657, the watchdog enable bit (WDT) resides in the TCO register.

   When programming the microcontroller, enable or disable the watchdog timer immediately after a power-up reset. The power-up reset condition can be sensed by reading the TO and PD bits in the STATUS register. This discriminates the power-up reset condition from, say, a wakeup reset after PWRDN.

2) The counter/timer external input is latched (clocked) at the beginning of every MCLK cycle on the SLC1657. The same input on the Microchip part is sampled twice during every clock cycle. This deviation is not expected to cause any major problems, and is required to achieve the goal of one instruction per clock cycle. The Microchip part requires four clock cycles for each instruction.

3) Input data on I/O ports PORT0-2 is latched at the beginning of every MCLK cycle on the SLC1657. Input data is not latched on the Microchip part. This deviation is not expected to cause any major problems, and is made to insure proper set-up and hold timing in FPGA and ASIC devices, as well as to simplify the timing specification.

4) The width of PORT0 is eight bits, and not four.

5) The instruction mnemonics will vary between the Silicore, Parallax® and Microchip instruction sets. Table 3-6 shows the mnemonics used by these three companies. It should be noted that while the mnemonics differ, they all result in the same binary instruction op-code when assembled.

Table 3-6. Instruction mnemonic conversion.

Silicore Mnemonic	Parallax Mnemonic	Microchip Mnemonic
ADD	ADD	ADDWF
AND	AND	ANDWF
ANDI	AND	ANDLW
BCLR	CLC/CLR/CLZ/CLRB	BCF
BRA	JMP	GOTO
BSET	SETB/STC/STZ	BSF
BSR	CALL	CALL
BTSC	SNB/SNC/SNZ	BTFSC
BTSS	SB/SC/SKIP/SZ	BTFSS
CLR	CLR	CLRF, CLRW
DEC	DEC/MOV	DECF
DECSZ	DECSZ/MOVSZ	DECFSZ
INC	INC/MOV	INCF
INCSZ	INCSZ/MOVSZ	INCFSZ
MOV	MOV/TEST	MOVF
MOVA	MOV	MOVWF
MOVI	MOV	MOVLW
MOVP	MOV	TRIS
MOVT	MOV	OPTION
NOP	NOP	NOP
NOT	MOV/NOT	COMF
OR	OR	IORWF
ORI	OR/TEST	IORLW
PWRDN	SLEEP	SLEEP
RET	RET	RETLW
ROL	MOV/RL	RLF
ROR	MOV/RR	RRF
RWT	CLR	CLRWDT
SUB	MOV/SUB	SUBWF
SWPN	MOV/SWAP	SWAPF
XOR	XOR	XORWF
XORI	NOT/XOR	XORLW

3.7 Instruction Set

Table 3-7 is a summary of the SLC1657 instruction set. This is followed by a detailed description of each instruction.

Table 3-7. Instruction set summary.

Mnemonic	Oper-and	Description	No. Cycles	STATUS Affected	Op-code
ADD	R,D	ADD register and ACCUM	1	Z,C,NC	0001 11DR RRRR
AND	R,D	AND register with ACCUM	1	Z	0001 01DR RRRR
ANDI	V	AND immediate with ACCUM	1	Z	1110 VVVV VVVV
BCLR	R,B	Clear register bit	1	-	0100 BBBR RRRR
BRA	V	Branch	2	-	101V VVVV VVVV
BSET	R,B	Set register bit	1	-	0101 BBBR RRRR
BSR	V	Branch to subroutine	2	-	1001 VVVV VVVV
BTSC	R,B	Test bit and skip if clear	1(2)	-	0110 BBBR RRRR
BTSS	R,B	Test bit and skip if set	1(2)	-	0111 BBBR RRRR
CLR	R,D	Clear register or ACCUM	1	Z	0000 01DR RRRR
DEC	R,D	Decrement register	1	Z	0000 11DR RRRR
DECSZ	R,D	Dec. register, skip if zero	1(2)	-	0010 11DR RRRR
INC	R,D	Increment register	1	Z	0010 10DR RRRR
INCSZ	R,D	Inc. register, skip if zero	1(2)	-	0011 11DR RRRR
MOV	R,D	Move register	1	Z	0010 00DR RRRR
MOVA	R	Move ACCUM to register	1	-	0000 001R RRRR
MOVI	V	Move immediate to ACCUM	1	-	1100 VVVV VVVV
MOVP	V	Move ACCUM to PC0-2	1	-	0000 0000 0VVV
MOVT	-	Move ACCUM to TCO	1	-	0000 0000 0010
NOP	-	No operation	1	-	0000 0000 0000
NOT	R,D	NOT register	1	Z	0010 01DR RRRR
OR	R,D	OR register with ACCUM	1	Z	0001 00DR RRRR
ORI	V	OR immediate with ACCUM	1	Z	1101 VVVV VVVV
PWRDN	-	Power-down	1	TO, PD	0000 0000 0011
RET	V	Return from subroutine	2	-	1000 VVVV VVVV
ROL	R,D	Rotate register left	1	C	0011 01DR RRRR
ROR	R,D	Rotate register right	1	C	0011 00DR RRRR
RWT	-	Reset watchdog timer	1	TO, PD	0000 0000 0100
SUB	R,D	Subtract ACCUM from register	1	Z,C,NC	0000 10DR RRRR
SWPN	R,D	Swap nibbles in register	1	-	0011 10DR RRRR
XOR	R,D	XOR register with ACCUM	1	Z	0001 10DR RRRR
XORI	V	XOR immediate with ACCUM	1	Z	1111 VVVV VVVV

Key: 'D': destination of the result (0 → ACCUM, 1 → register); 'R': register number (0x00 - 0x1F); 'V': immediate (data, bit number or address); '1(2)': one or two cycles, depending upon result.

Add Register and Accumulator ADD

Description: The contents of the indicated register is added to the accumulator. The result is placed into the register or the accumulator. Two's compliment arithmetic is used.

Mnemonic: ADD R,D

where 'R' is the register number (0x00 - 0x1F) and 'D' is the destination (0 → accumulator; 1 → register).

Number Cycles: 1

Operation: (ACCUM + R) → D; (PC + 1) → PC

STATUS affected: Z, C, NC

Binary op-code: 0001 11DR RRRR

AND Register With Accumulator AND

Description: The contents of the indicated register is logically 'AND'ed with the accumulator. The result is placed into the register or the accumulator.

Mnemonic: AND R,D

where 'R' is the register number (0x00 - 0x1F), and 'D' is the destination (0 → accumulator; 1 → register).

Number Cycles: 1

Operation: (ACCUM and R) → D; (PC + 1) → PC

STATUS affected: Z

Binary op-code: 0001 01DR RRRR

AND Immediate with Accumulator ANDI

Description: An immediate value is logically 'AND'ed with the accumulator. The result is placed into the accumulator.

Mnemonic: ANDI V

where 'V' is an eight-bit immediate value.

Number Cycles: 1

Operation: (ACCUM and V) → ACCUM; (PC + 1) → PC

STATUS affected: Z

Binary op-code: 1110 VVVV VVVV

Clear Register Bit BCLR

Description: The indicated bit in the indicated register is cleared. The result is placed back into the register.

Mnemonic: BCLR R,B

where 'R' is the register number (0x00 - 0x1F), and 'B' is the bit number (0x0 - 0x7).

Number Cycles: 1

Operation: 0 → register bit 'B'; (PC + 1) → PC

STATUS affected: None

Binary op-code: 0100 BBBR RRRR

Branch BRA

Description: The program counter is loaded with the indicated address. This causes program execution to branch to a new location.

Mnemonic: BRA V

where 'V' is a nine-bit address (0x000 - 0x1FF).

Number Cycles: 2

Operation: V → PC

STATUS affected: None

Binary op-code: 101V VVVV VVVV

Set Register Bit BSET

Description: The indicated bit in the indicated register is set. The result is placed back into the register.

Mnemonic: BSET R,B

where 'R' is the register number (0x00 - 0x1F), and 'B' is the bit number (0x0 - 0x7).

Number Cycles: 1

Operation: 1 → reigster bit 'B'; (PC + 1) → PC

STATUS affected: None

Binary op-code: 0101 BBBR RRRR

Branch to Subroutine BSR

Description: The program counter is incremented and pushed onto the stack. The program counter is then loaded with the indicated address.

Mnemonic: BSR V

where 'V' is an eight-bit address (0x00 - 0xFF). Address bit nine is forced to zero. Note that subroutines called by this instruction must reside in the lower half of instruction space.

Number Cycles: 2

Operation: (PC + 1) → STACK; V → PC

STATUS affected: None

Binary op-code: 1001 VVVV VVVV

Test Bit and Skip If Clear BTSC

Description: Test the indicated bit in the indicated register. If the bit is a zero, then skip the next instruction. If the bit is a one, then execute the next instruction. This instruction takes one or two CPU cycles, depending on the state of the bit.

Mnemonic: BTSC R,B

where 'R' is the register number (0x00 - 0x1F), and 'B' is the bit number (0x0 - 0x7).

Number Cycles: Bit set: 1; bit cleared: 2

Operation: if register bit 'B' = 0 then (PC + 2) → PC; else (PC + 1) → PC

STATUS affected: None

Binary op-code: 0110 BBBR RRRR

Test Bit and Skip If SET BTSS

Description: Test the indicated bit in the indicated register. If the bit is a one, then skip the next instruction. If the bit is a zero, then execute the next instruction. This instruction takes one or two CPU cycles, depending on the state of the bit.

Mnemonic: BTSS R,B

where 'R' is the register number (0x00 - 0x1F), and 'B' is the bit number (0x0 - 0x7).

Number Cycles: Bit set: 2; bit cleared: 1

Operation: if register bit 'B' = 1 then (PC + 2) → PC; else (PC + 1) → PC

STATUS affected: None

Binary op-code: 0111 BBBR RRRR

Clear Register or Accumulator CLR

Description: Clear the accumulator or the indicated register.

Mnemonic: CLR R,D

where 'R' is the register number (0x00 - 0x1F), and 'D' is the destination (0 → accumulator; 1 → register). Note: when the destination is the accumulator, then set 'R' equal to 0x00.

Number Cycles: 1

Operation: 0x00 → D; (PC + 1) → PC

STATUS affected: 1 → Z

Binary op-code: 0000 01DR RRRR

Decrement Register DEC

Description: The contents of the indicated register is decremented. The result can be placed into the register or the accumulator.

Mnemonic: DEC R,D

where 'R' is the register number (0x00 - 0x1F), and 'D' is the destination (0 → accumulator; 1 → register).

Number Cycles: 1

Operation: (R - 1) → D; (PC + 1) → PC

STATUS affected: Z

Binary op-code: 0000 11DR RRRR

Decrement Register, Skip if Zero DECSZ

Description: The contents of the indicated register is decremented. If the result of the decrement is zero, then the next instruction is skipped. The result can be placed into the register or the accumulator.

Mnemonic: DECSZ R,D

where 'R' is the register number (0x00 - 0x1F), and 'D' is the destination (0 → accumulator; 1 → register).

Number Cycles: D /= 0x00: 1; D = 0x00: 2

Operation: (R - 1) → D; if (D = 0x00) then (PC+2) → PC else(PC+1) → PC

STATUS affected: None

Binary op-code: 0010 11DR RRRR

Increment Register INC

Description: The contents of the indicated register is incremented. The result can be placed into the register or the accumulator.

Mnemonic: INC R,D

where 'R' is the register number (0x00 - 0x1F), and 'D' is the destination (0 → accumulator; 1 → register).

Number Cycles: 1

Operation: (R + 1) → D; (PC + 1) → PC

STATUS affected: Z

Binary op-code: 0010 10DR RRRR

Increment Register, Skip if Zero INCSZ

Description: The contents of the indicated register is incremented. If the result of the increment is zero, then the next instruction is skipped. The result can be placed into the register or the accumulator.

Mnemonic: INCSZ R,D

where 'R' is the register number (0x00 - 0x1F), and 'D' is the destination (0 → accumulator; 1 → register).

Number Cycles: D /= 0x00: 1; D = 0x00: 2

Operation: (R + 1) → D; if (D = 0x00) then (PC+2) → PC else(PC+1) → PC

STATUS affected: None

Binary op-code: 0011 11DR RRRR

Move Register MOV

Description: The contents of the indicated register is moved to the destination register. The destination can be the register or the accumulator. Moving a register back into itself can be used to set the 'Z' bit.

Mnemonic: MOV R,D

where 'R' is the register number (0x00 - 0x1F), and 'D' is the destination (0 → accumulator; 1 → register).

Number Cycles: 1

Operation: R → D; (PC + 1) → PC

STATUS affected: Z

Binary op-code: 0010 00DR RRRR

Move Accumulator to Register MOVA

Description: The contents of the accumulator is moved to the indicated register.

Mnemonic: MOVA R

where 'R' is the register number (0x00 - 0x1F).

Number Cycles: 1

Operation: (PC + 1) → PC, ACCUM → R

STATUS affected: None

Binary op-code: 0000 001R RRRR

Move Immediate to Accumulator MOVI

Description: The contents of the accumulator is loaded with immediate data.

Mnemonic: MOVI V

where 'V' is the value to be loaded into the accumulator.

Number Cycles: 1

Operation: V → ACCUM

STATUS affected: None

Binary op-code: 1100 VVVV VVVV

Move Accumulator to PC0-2 MOVP

Description: The contents of the accumulator is moved to the indicated port control register (PC0-2).

Mnemonic: MOVP V

where 'V' is the port control register number. For PC0, 'V' = 5; for PC1, 'V' = 6; for PC2, 'V' = 7.

Number Cycles: 1

Operation: ACCUM → PC0-2, (PC + 1) → PC

STATUS affected: None

Binary op-code: 0000 0000 0VVV

Move Accumulator to TCO MOVT

Description: The contents of the accumulator is moved to the timer/counter option (TCO) register.

Mnemonic: MOVT

Number Cycles: 1

Operation: ACCUM $\rightarrow$ TCO, (PC + 1) $\rightarrow$ PC

STATUS affected: None

Binary op-code: 0000 0000 0010

No Operation NOP

Description: A single CPU cycle is performed without affecting any of the internal registers or STATUS bits.

Mnemonic: NOP

Number Cycles: 1

Operation: (PC + 1) $\rightarrow$ PC

STATUS affected: None

Binary op-code: 0000 0000 0000

NOT Register NOT

Description: The contents of the register are inverted. The result can be placed into the register or the accumulator.

Mnemonic: NOT R,D

where 'R' is the register number (0x00 - 0x1F), and 'D' is the destination (0 → accumulator; 1 → register).

Number Cycles: 1

Operation: /R → D; (PC + 1) → PC

STATUS affected: Z

Binary op-code: 0010 01DR RRRR

OR Register With Accumulator OR

Description: The contents of the indicated register is logically 'OR'ed with the accumulator. The result can be placed into the accumulator or the register.

Mnemonic: OR R,D

where 'R' is the register number (0x00 - 0x1F), and 'D' is the destination (0 → accumulator; 1 → register).

Number Cycles: 1

Operation: (ACCUM or R) → D; (PC + 1) → PC

STATUS affected: Z

Binary op-code: 0001 00DR RRRR

OR Immediate with Accumulator ORI

Description: The contents of the accumulator is logically 'OR'ed with an immediate value. The result is placed into the accumulator.

Mnemonic: ORI V

where 'V' is an eight-bit value.

Number Cycles: 1

Operation: (ACCUM or V) → ACCUM; (PC + 1) → PC

STATUS affected: Z

Binary op-code: 1101 VVVV VVVV

Power-down PWRDN

Description: The core is powered down until a reset occurs. The cycle modifies the 'PD' and 'TO' bits in the STATUS register. The watchdog timer and its prescaler (if used for the watchdog) are cleared, and the SLEEP signal is asserted. During power-down the master clock [MCLK] continues to operate, but no instructions are fetched.

Mnemonic: PWRDN

Number Cycles: 1

Operation: Clear watchdog / prescaler, update TO & PD, (PC + 1) → PC

STATUS affected: TO, PD (refer to the *Reset Operation* elsewhere in this manual)

Binary op-code: 0000 0000 0011

Return From Subroutine RET

Description: The accumulator is loaded with an immediate value. The program counter is poped from the stack.

Mnemonic: RET V

where 'V' is an eight-bit value.

Number Cycles: 2

Operation: V → ACCUM; STACK → PC

STATUS affected: None

Binary op-code: 1000 VVVV VVVV

Rotate Register Left ROL

Description: The contents of the indicated register is rotated one bit to the left (through the carry bit). The result can be placed into the accumulator or the register.

Mnemonic: ROL R,D

where 'R' is the register number (0x00 - 0x1F), and 'D' is the destination (0 → accumulator; 1 → register).

Number Cycles: 1

Operation: See description.

STATUS affected: C

Binary op-code: 0011 01DR RRRR

Rotate Register Right ROR

Description: The contents of the indicated register is rotated one bit to the right (through the carry bit). The result can be placed into the accumulator or the register.

Mnemonic: ROR R,D

where 'R' is the register number (0x00 - 0x1F), and 'D' is the destination (0 → accumulator; 1 → register).

Number Cycles: 1

Operation: See description.

STATUS affected: C

Binary op-code: 0011 00DR RRRR

Reset Watchdog Timer RWT

Description: Resets the watchdog timer. This instruction also resets the prescaler if it is assigned to the watchdog timer. STATUS register bits T0 and PD are set.

Mnemonic: RWT

Number Cycles: 1

Operation: Clear watchdog, clear prescaler (if assigned to watchdog),
(PC + 1) → PC

STATUS affected: 1 → TO, 1 → PD

Binary op-code: 0000 0000 0100

Subtract Accumulator From Register SUB

Description: The contents of the accumulator is subtracted from the indicated register. The result can be placed into the accumulator or the register. Two's compliment arithmetic is used.

Mnemonic: SUB R,D

where 'R' is the register number (0x00 - 0x1F), and 'D' is the destination (0 → accumulator; 1 → register).

Number Cycles: 1

Operation: (ACCUM - R) → D; (PC + 1) → PC

STATUS affected: Z, C, NC

Binary op-code: 0000 10DR RRRR

Swap Nibbles in Register SWPN

Description: Swaps the higher and lower nibbles in the indicated register. The result can be placed into the register or the accumulator.

Mnemonic: SWPN R,D

Number Cycles: 1

Operation: See description.

STATUS affected: None

Binary op-code: 0011 10DR RRRR

XOR Register with Accumulator XOR

Description: The contents of the indicated register is logically 'XOR'ed with the accumulator. The result can be placed into the register or the accumulator.

Mnemonic: XOR R,D

where 'R' is the register number (0x00 - 0x1F), and 'D' is the destination (0 → accumulator; 1 → register).

Number Cycles: 1

Operation: (ACCUM xor R) → D; (PC + 1) → PC

STATUS affected: Z

Binary op-code: 0001 10DR RRRR

XOR Immediate with Accumulator XORI

Description: The contents of the accumulator is logically 'XOR'ed with an immediate value. The result is placed into the accumulator.

Mnemonic: XORI V

where 'V' is an eight-bit value.

Number Cycles: 1

Operation: (ACCUM xor V) → ACCUM; (PC + 1) → PC

STATUS affected: Z

Binary op-code: 1111 VVVV VVVV

4.0 VHDL Synthesis and Test

The SLC1657 was created and delivered in the VHDL hardware description language. VHDL source code must be synthesized by the user before operation on a particular target device (such as an FPGA or ASIC). A variety of simulation, synthesis and CASE[15] tools can be used with the core.

Most of the components used by the core are provided with the source code. However, there are a few exceptions. RAM, ROM and I/O drivers must be synthesized with entities provided by the FPGA or ASIC vendor. This is because portable, synthesizable RAM and ROM elements are not supported by the VHDL standards. Examples of complete design solutions (with RAM, ROM and I/O) are provided elsewhere in this manual.

The SLC1657 is provided as a 'soft core'. This means that all VHDL source code and test benches are provided with the design. This enables the user to see inside of the design, thereby allowing a better understanding of it. This is useful from both a design standpoint and from a test standpoint. From a design standpoint the user can tweak the source code to better fit the application. From a test standpoint, it allows the user to create custom test benches that incorporate both the core and other entities on the same IC.

Furthermore, the soft core approach allows the SLC1657 to be synthesized and tested with a variety of software tools. This reduces the cost of special VHDL development software. Users should verify that their software tools conform to the IEEE standards listed in the next section of this manual.

Soft cores are fundamentally different than 'firm cores' or 'hard cores'. These approaches have the advantage of maintainability and security, but limit the creative ability of the end user. Furthermore, they do not permit portable, reliable test methodology, especially in ASIC target devices.

4.1 VHDL Simulation and Synthesis Tools

It is assumed by Silicore Corporation that all simulation and synthesis tools conform to the following standards[16]:

- IEEE Standard VHDL Language Reference Manual, IEEE STD 1076-1993.
- IEEE Standard VHDL Synthesis Packages, IEEE STD 1073.3-1997.

[15] CASE: Computer Aided Software Environment

[16] Copies of the standards can be obtained from: IEEE Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ USA 08855 (800) 678-4333 or from: www.ieee.org

- IEEE Standard Multivalue Logic System for VHDL Model Interoperability, IEEE STD 1164-1993.

In most cases the VHDL source code should be fully portable as long as the simulation and synthesis tools conform to these standards[17]. However, if incompatibilities between the source code and the user's tools are found, please contact Silicore Corporation so that the problem can be resolved.

It is strongly recommended that the user have a set of VHDL simulation tools before integrating the SLC1657. These help in two ways: (a) they build confidence that the core synthesizes correctly and (b) they help resolve any integration problems. The simulation tools do not need to be fancy...a simple non-graphical simulator is adequate.

All original VHDL source files have been edited with the MS-DOS editor. Font style: COURIER (monotype), tab spacing: 4. Almost any editor can be used, but the user may find that the style and formatting of the source code is more readable using this (or a compatible) editor.

[17] The original SLC1657 was developed with PeakVHDL simulation and synthesis tools available from: Protel International. For more information, please refer to: www.peakvhdl.com.

4.2 VHDL Portability

Portability of the VHDL source code is a very high priority in the SLC1657 design. It is assumed that the core will be used in a variety of target devices and tools.

Several proven techniques have been used in the source code to enhance its portability. These apply to the synthesizable code, and not to the test benches. These include:

- No *variable* types are used. Variables tend to synthesize unusual logic in some VHDL compilers, and have not been used in the *synthesizable* entities. For example, all counters are designed with logic functions, and not with incremental variables.

- No internal three-state buses are used. Some FPGA architectures do not support three-state buses well, and have been eliminated from the core (except for the I/O port interfaces, which are user defined). However, some VHDL synthesis tools will automatically create three-state buses on large multiplexors. This is perfectly acceptable if the target device supports them.

- Synchronous resets, synchronous presets and asynchronous resets are used. No asynchronous presets are used in the design. Most FPGA and ASIC flip-flops will handle synchronous resets and presets very well. The asynchronous resets are less portable, but are still supported by most devices. Asynchronous presets are least portable, and have been eliminated from the design.

- Asynchronous (unintended latches) have been eliminated from the design. These are usually the result of incompletely specified *if-then-elsif* VHDL statements.

- Each source file contains one entity/architecture pair. Some simulator and synthesis tools cannot handle more than one entity/architecture pair per file.

4.3 Required Resources on the Target Device

The logic resources required by the SLC1657 are fairly common, and are available in most FPGA and ASIC target devices. However, before synthesis the user should confirm that the following elements are available on the target device:

- A single, global, low skew clock interconnect (for [MCLK]). Most of the logic in the core is synchronous, and a global clock coordinates all of the internal activity.

- Logic elements such as NAND gates, NOR gates, inverters and D-type flip-flops. Only elements defined by the IEEE STD 1164-1993 standard are used in the core.

- D-type flip-flops with asynchronous reset. Although most reset/preset circuits are synchronous, the WATCHDOG entity does require an asynchronous reset.

- D-type flip-flops with known power-up conditions. The SLC1657 has two internal bits ('TO' and 'PD') that must be set to a pre-defined state after a power-up reset. These bits are defined in the RESETGEN entity. It is assumed that all flip-flops power-up in the negated (i.e. cleared) condition. Refer to the RESETGEN entity description for more details.

- 72 x 8-bit synchronous RAM. This is used for the register RAM. The core assumes that the synchronous RAMs are FASM (FPGA and ASIC Subset Model) compatible. For more information, see the FASM synchronous RAM model (below).

- 2,048 X 12 asynchronous ROM block. The core assumes that the asynchronous ROMs are FASM (FPGA and ASIC Subset Model) compatible. For more information, please refer to the FASM asynchronous ROM model (below). In some cases, other types of ROMs may be used. For example, RAM can be substituted for the ROM if field upgrades of the application software are anticipated. Also, the core can be modified to use less than the full 2,048 words or instruction memory.

4.3.1 FASM Synchronous RAM

The FASM[18] synchronous RAM model is used whenever single, read and write clock cycles are used. This memory conforms to the connection and timing diagram shown in Figure 4-1. The SLC1657 core assumes that the register RAM operates in this way.

During write cycles, FASM Synchronous RAM stores input data at the indicated address whenever: (a) the write enable (WE) input is asserted, and (b) there is a rising clock edge.

During read cycles, FASM Synchronous RAM works like an asynchronous ROM. Data is fetched from the address indicated by the ADR() inputs, and is presented at the data output (DOUT). The clock input is ignored. However, during write cycles, the output data is updated immediately during a write cycle.

[18] FASM: FPGA and ASIC Subset Model. The FASM model describes a set of available resources that are common to most FPGA and ASIC target devices. This simplifies the task of creating portable IP cores.

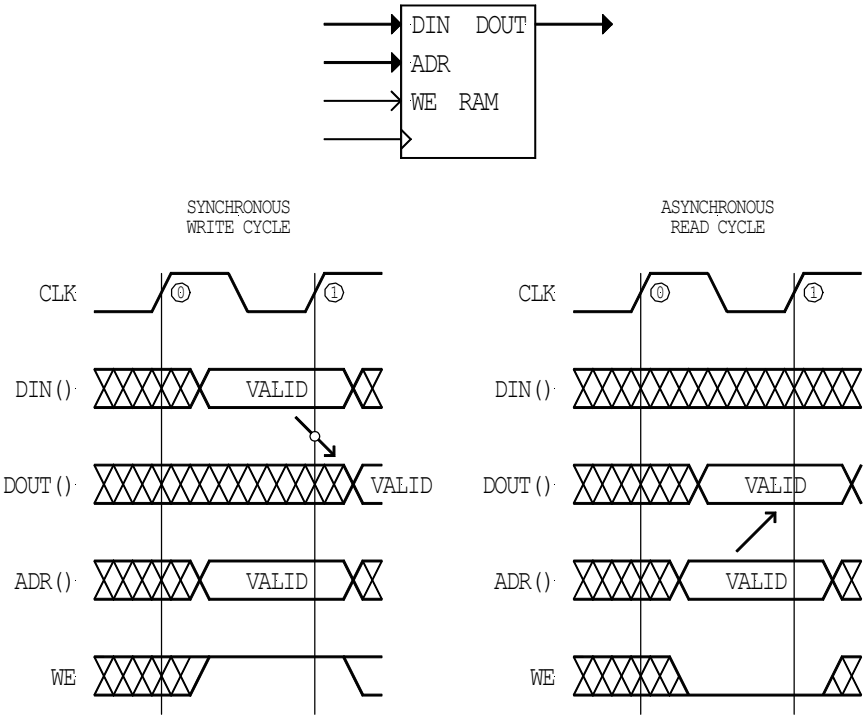


Figure 4-1. FASM synchronous RAM connection and timing diagram.

4.3.2 FASM Asynchronous ROM

The SLC1657 core assumes that the instruction ROM operates like the FASM asynchronous ROM. This memory conforms to the connection and timing diagram shown in Figure 4-2.

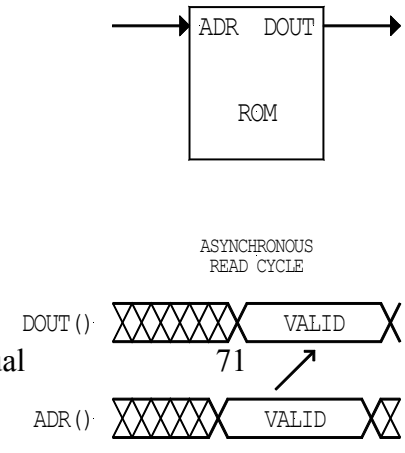


Figure 4-2. FASM asynchronous ROM connection and timing diagram.

Although the SLC1657 core assumes that the instruction ROM works in this manner, other types of memory can be used. For example, in the Xilinx Spartan-II FPGA, relatively large quantities of BLOCK RAM can be used. Unfortunately, these do not conform to the FASM ROM connection or timing. However, a simple interface circuit can be created so that the core can be used with them. For more information, please refer to the Xilinx sample circuits given in Chapter 6.

4.4 Soft Core Installation

The SLC1657 core is distributed as a set of VHDL source files. There is no special software required to install the core. It is recommended that you create a unique directory under the name 'SLC1657', and copy all of the directories (and files) on the distribution disk to the new directory.

Inside the SLC1657 directory there will be several sub-directories. Locate and open the sub-directory labeled 'C:\VHDL_Source\Rev1.0' (or substitute the latest revision number that you want). Note that all revisions are provided with the distribution. This is because the SLC1657 is distributed as source code. Silicore follows the conventional wisdom that, when source code is provided, all revisions of the source are provided to the user. This allows the end user to precisely track and review changes in the design, and to facilitate both forward and backward compatibility of the product. If you are familiar with the Linux operating system, then you will recognize this strategy. Most Linux software is distributed as source code, and all revisions are made available to the end user.

Inside the directory you will find a number of sub-directories named: 'TOPLOGIC', 'BINADDER' and so forth. These names correspond directly to the names of the VHDL entity/architecture pairs. The directory may also contain other files required to simulate or synthesize the particular entity. For example, each sub-directory contains a file called TSTBENCH.VHD. This is the test bench for that particular entity/architecture pair. Also, don't move the TSTBENCH.VHD files between folders. Each entity/architecture has the same filename, and moving these files could cause problems.

- CAUTION -

Each entity/architecture pair directory in the distribution contains a file named 'TSTBENCH.VHD'. This is a test bench file. All of the test benches have the same name, and should be moved between sub-directories with caution to prevent overwriting other test benches.

The source file names for each entity/architecture pair are all eight characters long, and are coupled with a '.VHD' extension. The eight character filename has the same name as the entity. For example, the ALULOGIC entity is stored under filename 'ALULOGIC.VHD', and contains both the entity and architecture descriptions. Also, the test bench for this file is named 'TSTBENCH.VHD'.

- IMPORTANT -
Some test benches require conversions between integers and standard logic vectors. If your test bench contains the statement "work.SLV2INTPAK.all", then it requires a file called 'INTRCONV'. This file is provided as part of the SLC1657 distribution.

4.5 Core Integration

Figure 4-3 shows how to integrate the SLC1657 core into the final application. This is a very general overview, and may need to be adjusted by the user. Specific integration examples for FPGA parts are given, starting in Chapter 6.

Also, this description does not specifically show the application code development. If the user is integrating the embedded ROM version of the product, then it is assumed that the ROM source files will be integrated as part of the process.

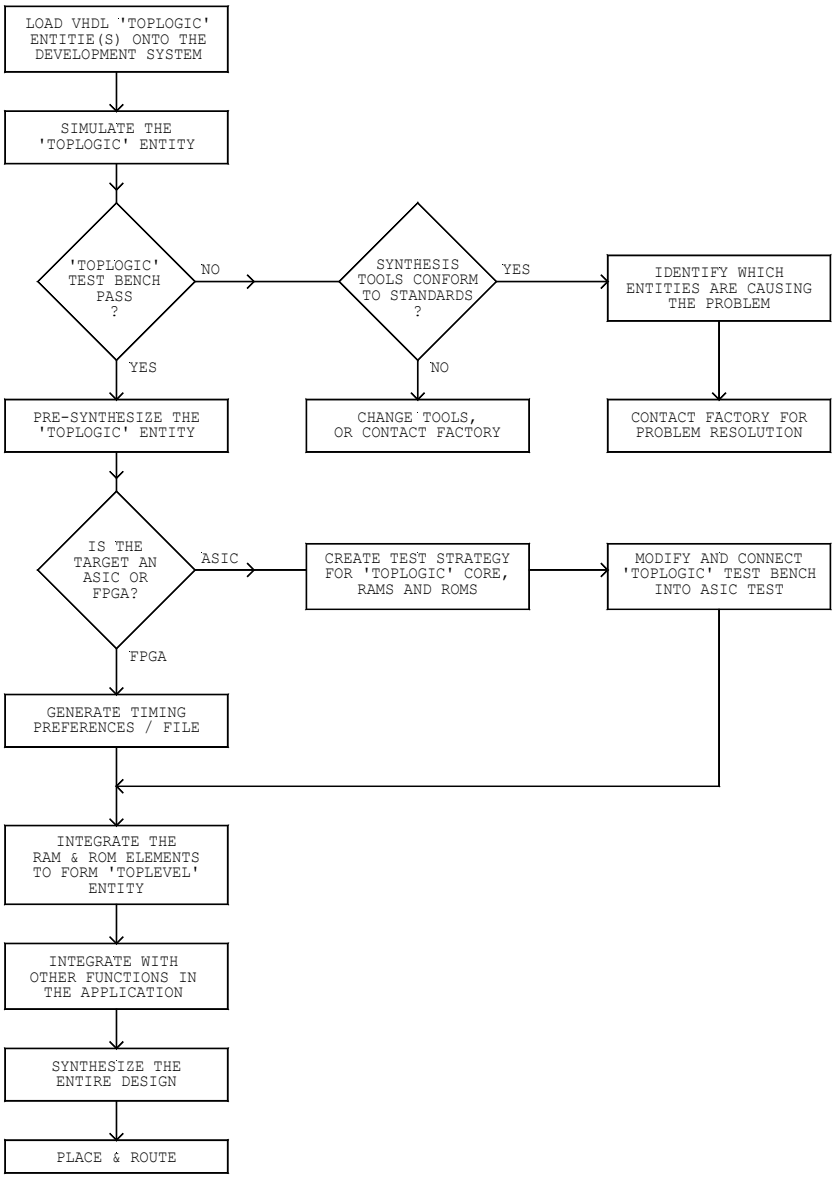


Figure 4-3. General integration of the SLC1657.

Follow these steps to integrate and synthesize the SLC1657:

1) Load the VHDL source files (including entities and test benches) onto the development computer as described in the installation instructions. Enter the TOPLOGIC entity into your simulation tool.

2) Simulate the TOPLOGIC VHDL entity. A complete test bench for the TOPLOGIC entity is provided in the TOPLOGIC file folder. The purpose of this

step is to (a) insure that all of the entities have been correctly installed and (b) verify that the development tools correctly interpret the VHDL source code.

The test benches provided with the core provide only static logic analysis. They do not provide any timing verification. If timing verification is needed, then the user must modify the test benches accordingly.

If the TOPLOGIC test bench simulation does not pass, then determine the cause of the problem. In some cases the simulation tool must be configured to compile using the applicable IEEE standards. Simulation and synthesis tools may have soft switches to enable IEEE compatibility. Refer to the simulator documentation (provided by the tool supplier) for more details.

If the development tools are configured properly, and the TOPLOGIC entity still does not simulate correctly, then the simulation tool may be interpreting the source code improperly. Identify which entities are causing the problem and contact the factory.

Some simulators provide rather cryptic debugging messages, and it may not be obvious where the problem resides. In these cases it may be more expedient to individually simulate each of the TOPLOGIC sub-entities (RESETGEN, PROGCNTR, etc.). Test benches are provided for each entity in the core for this purpose.

Every effort has been made by Silicore Corporation to insure that the SLC1657 VHDL source code conforms to the IEEE standards. However, experience has shown that some simulation and synthesis tools written to these standards may vary somewhat in their implementation. [Stated another way, there is some ambiguity in the way the standards are implemented]. SILICORE will make every effort to help the user resolve these types of problems, should they occur.

3) Pre-synthesize the TOPLOGIC entity. By *pre-synthesize* it is meant that the entity should be synthesized without the intention of placing or routing the design onto the IC. The purpose of this step is to verify that the development tools synthesizes the core correctly. This should be done early in the integration process, as this allows more time to resolve any problems.

4) If the target device is an *ASIC*, then the user must create a test strategy. ASIC integrations vary dramatically from FPGA integrations in the way their test strategies are implemented. For example, SRAM based FPGA devices can be 100% functionally tested at the factory, and require little or no test effort at the die level.

ASICs, on the other hand, require test strategies that allow wafer probing at the die level. Generally, this means that a parametric (timing) test must be created so that each internal function can be tested individually.

This process is complicated by the wide variety of ASIC tools and test strategies. Silicore Corporation makes no attempt to solve this problem, except to provide a starting point for the end user. A good ASIC test strategy is to break the core into four elements:

(a) TOPLOGIC entity
(b) RAM
(c) ROM
(d) I/O elements

Each of these can be independently tested if multiplexors are placed into the path between the RAM, ROM, I/O and the TOPLOGIC core. When the multiplexors are placed into a test mode they can be used to access these components, thereby allowing wafer test.

Furthermore, this method allows a generic test of the TOPLOGIC core. The TOPLOGIC test bench can be adjusted to the wafer die program to test all of the functions of the microcontroller. This causes TOPLOGIC to be tested independent of the ROM application code.

5) If the target device is an *FPGA*, then the user must create timing preferences for the device. This is usually a very simple process on the SLC1657. Generally, there are only two timing preferences which must be specified:

(a) [MCLK] to [MCLK] set-up times.
(b) Input / output to [MCLK] setup times.

The SLC1657 is completely synchronous, so this means the timing specification only has to show the relationship between signals and the clock setup times.

For example, in the LUCENT FPGA evaluation board, the timing preference simply dictates what the clock-to-clock setup times are. This is placed into a ‘preference file’ as follows:

Frequency net "MCLK" 5 MHz;

This simply specifies that the place-and-route tool must adjust all logic and timing paths to operate at a 5 MHz clock speed. In this case, the input / output setup times or omitted because they are irrelevant to the implementation.

It may also be useful to specify the device pinout at this stage of the integration. Refer to the FPGA place and route tool documentation for more details.

6) Integrate the RAM, ROM and I/O elements. The design of these high-level entities will vary from application-to-application.

   The RAMROM and I/O drivers must be synthesized with components provided by the FPGA or ASIC vendor. This is because portable, synthesizable RAM and ROM elements are not supported by the VHDL standards. This also allows the user greater flexibility in designing these elements into the application.

   Refer to the FPGA or ASIC vendor documentation before creating the RAM, ROM and I/O elements.

7) Integrate the application-specific entities into the design. Generally, these will be written and tested as their own core(s). The integration phase generally means connecting the core to the application-specific core(s), and then testing the entire design.

   In some cases, it is necessary to simulate the entire design. Generally, this means creating a VHDL ROM element that simulates the application code. In most cases the ROM entity can be created with the FPGA or ASIC design vendor tools. These same tools will usually create RAM and ROM test benches.

   In some cases, the user may wish to use the emulation ROM entity. This allows the application code to be downloaded to the core, and is very useful for testing the end application on an FPGA. In these cases, it may not make any sense to simulate the entire design until the application code has been completed.

8) Synthesize the entire design. Once the SLC1657 and application-specific cores have been integrated, the entire design must be synthesized. Refer to the synthesis tool documentation for more details.

9) Place and route. When synthesis has been completed, the entire design must be placed and routed onto the FPGA or ASIC. Refer to the place & route tool documentation for more details.

4.6 VHDL Reference Books

Some of the following VHDL reference books may be useful to the user:

- Ashenden, Peter J. The Designer's Guide to VHDL. Morgan Kaufmann Publishers, Inc. 1996. ISBN 1-55860-270-4. Excellent general purpose reference guide to VHDL. Weak on synthesis, stronger on test benches. Good general purpose guide, very complete.

- IEEE Standard VHDL Language Reference Manual. IEEE Std 1076-1993. IEEE, New York NY USA 1993. This is a standard, and not a tutorial by any means. Useful for defining portable VHDL code.

- IEEE Standard VHDL Synthesis Packages. IEEE Std 1076.3-1997. IEEE, New York NY USA 1997. This is a standard, and not a tutorial by any means. Useful for defining portable VHDL code.

- IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164). IEEE Std 1164-1993. IEEE, New York NY USA 1993. This is a standard, and not a tutorial by any means. Useful for defining portable VHDL code.

- Pellerin, David and Douglas Taylor. VHDL Made Easy. Prentice Hall PTR 1997. ISBN 0-13-650763-8. Good introduction to VHDL synthesis and test benches, and closely follows the IEEE standards.

- Skahill, Kevin. VHDL For Programmable Logic. Addison-Wesley 1996. ISBN 0-201-89573-0. Excellent reference for VHDL synthesis. Very good treatment of practical VHDL code for the synthesis of logic elements. Weak on test benches and execution of the IEEE standards.

5.0 Hardware (VHDL Entity) Reference

The SLC1657 core is organized as a series of VHDL entities. These are tied together into an entity called TOPLOGIC. All of the entities, including TOPLOGIC, are described in this chapter. However, the SLC1657 distribution includes other entities that are not described in this chapter. For example, the Xilinx Spartan 2 evaluation board example (presented in Chapter 6) contains other entities that are specific to that implementation. Those additional entities are described in that chapter.

The SLC1657 core was upgraded from its predecessor, the SLC1655. This upgrade increased both the program (instruction) memory, and the register memory. This involved changes to the following entities (along with their associated test files):

- INDEXREG.VHD
- INSTRDEC.VHD
- PROGCNTR.VHD
- STATSREG.VHD
- TOPLOGIC.VHD

5.1 ALULOGIC Entity

Other entities that use this module: TOPLOGIC
Other entities used by this module: BINADDER

The ALULOGIC entity generates all logical and arithmetic operations. During each instruction, data is presented by the register and/or the accumulator at the 'R' and 'A' inputs respectively. At the same time, the INSTRDEC entity specifies the type of ALU function on the [ALF(3..0)] input. The ALULOGIC entity then operates on the input data, and places the result on the [ALU(7..0)] output bus. Table 5-1 shows the ALU function encoding, and the resulting operation.

The ALULOGIC entity also generates the Z, C and NC condition code bits. These are presented on status bus [STA(6..0)], and are latched (or checked) by the STATSREG and INSTRDEC entities. The encoding of [STA(6..0)] is shown in Table 5-2. The status bus also includes a bit test signal, which is used by the INSTRDEC entity during BTSC and BTSS instructions.

During rotate instructions (ROL and ROR), a carry-in bit [CIN] is presented to the ALULOGIC entity by the STATSREG entity. This bit is needed because both instructions rotate through the carry bit. No other operations (ADD, SUB, etc.) use the [CIN] bit.

During bitwise operations (BCLR, etc.), the bit number of the operand is presented to the ALULOGIC using a portion of the instruction bus [INS(7..5)].

Table 5-1. Encoding of ALU function [ALF(3..0)].

ALF(3..0)	General Description	Mnemonic Description	STATUS Bits Affected
0000	PASs through No z	PASN: D → ALUout	-
0001	CLeaR	0x00 → ALUout	Z
0010	SUBtract	(D - A) → ALUout	Z, C, NC
0011	DECrement	(D - 1) → ALUout	Z
0100	OR, logical	OR(A, D) → ALUout	Z
0101	AND, logical	AND(A, D) → ALUout	Z
0110	XOR, logical	XOR(A, D) → ALUout	Z
0111	ADD	(A + D) → ALUout	Z, C, NC
1000	PASs through w/Z	PASZ: D → ALUout	Z
1001	NOT, logical	NOT(D) → ALUout	Z
1010	INCrement	(D + 1) → ALUout	Z
1011	Bit CLeaR	BCLR(bit D) → ALUout	-
1100	ROtate Right	ROR(D) → ALUout	C
1101	ROtate Left	ROL(D) → ALUout	C
1110	SWaP Nibbles	SWPN(D) → ALUout	-
1111	Bit SET	BSET(bit D) → ALUout	-

Table 5-2. Encoding of the STA(6..0) bus.

STA(6..0)	Function
STA(0)	Carry bit (C)
STA(1)	Carry bit (C) clock enable
STA(2)	Nibble-carry bit (NC)
STA(3)	Nibble-carry bit (NC) clock enable
STA(4)	Zero bit (Z)
STA(5)	Zero bit (Z) clock enable
STA(6)	BIT TEST result (not used by STATUS)

5.2 BINADDER Entity

Other entities that use this module: ALULOGIC
Other entities used by this module: NONE

The BINADDER entity is used by the arithmetic logic unit (ALU) to perform add, subtract, increment and decrement functions. All logic in this entity is combinational, meaning that no clocks are used. Figure 5-1 shows a block diagram of the entity.

Data is presented on the [AIN(7..0)] and [BIN(7..0)] inputs of the binary adder functional entity. The entity then performs 2's complement addition. During subtraction, the input and output data buses are inverted to produce the correct results.

Signal [ADDINC] is asserted only during add and increment operations. During these instructions the value at the 'A' input is added to the 'B' input. During subtract and decrement instructions, the 'A' input and 'O', 'C' and 'NC' outputs are inverted.

Signal [ADDSUB] is asserted only during add and subtract operations. During these operations, the 'A' input is added to the 'B' input. During increment and decrement operations, a value of 0x01 is jammed into the 'B' input.

Condition code bits 'C' and 'NC' during add, subtract, increment and decrement functions are also generated by the BINADDER entity. The 'Z' bit, however, is generated near the output of the ALULOGIC entity.

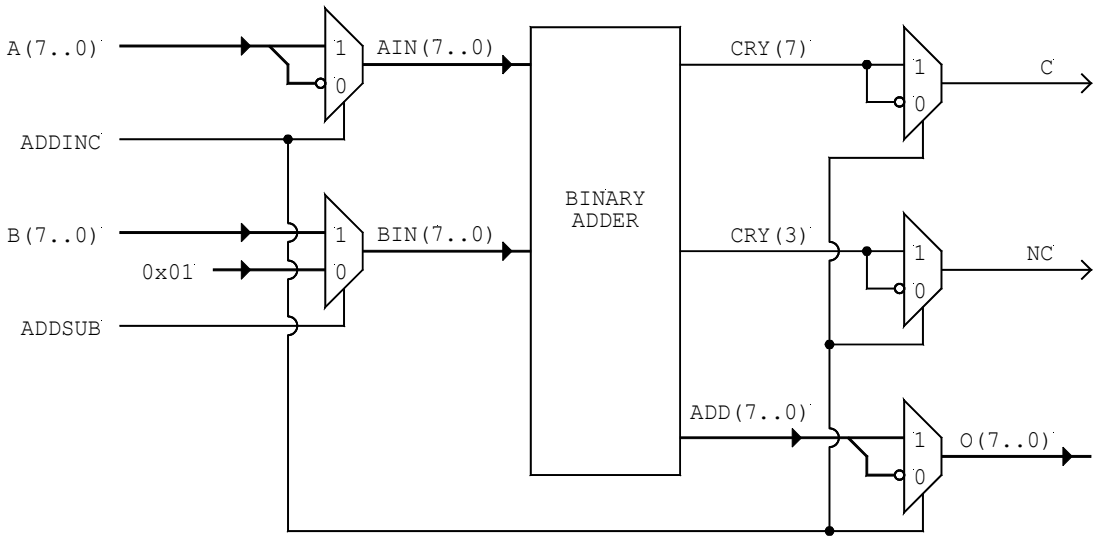


Figure 5-1. Block diagram of the BINADDER entity.

5.3 BUC08NNP Entity

Other entities that use this module: TIMRCNTR
Other entities used by this module: NONE

The BUC08NNP entity is an eight-bit binary ‘up’ counter with preload capabilities. It is used as the counter in the TIMRCNTR entity.

No ‘variable’ types are used in the VHDL counter design. Variables tend to produce unusual code in some VHDL compilers, and have been avoided here. The counter is designed with logic functions, and not with incremental variables.

5.4 BUC11CPP Entity

Other entities that use this module: PROGCNTR
Other entities used by this module: NONE

The BUC11CPP entity is an eleven-bit binary ‘up’ counter with clock enable, synchronous preset and preload capabilities. It is used to generate the program counter. After every assertion of [MRESET], the BUC11CPP is preset to binary 0x7FF to generate the initial instruction address.

No ‘variable’ types are used in the VHDL counter design. Variables tend to produce unusual code in some VHDL compilers, and have been avoided here. The counter is designed with logic functions, and not with incremental variables.

5.5 CLOCKDIV Entity

Other entities that use this module: TOPLOGIC
Other entities used by this module: NONE

The CLOCKDIV entity generates the [MCLK_4] and [MCLK_16] clocks from the microcontroller clock [MCLK]. It is a four bit up counter with synchronous reset. The counter is a ‘free-running’ type, and the synchronous reset is used only for test purposes.

No ‘variable’ types are used in the VHDL counter design. Variables tend to produce unusual code in some VHDL compilers, and have been avoided here. The counter is designed with logic functions, and not with incremental variables.

5.6 INDEXREG Entity

Other entities that use this module: TOPLOGIC
Other entities used by this module: NONE

The INDEXREG entity contains a seven-bit register with clock enable. The register is located on the lower seven bits, and the upper five bit returns logic '1'.

The INDEXREG entity was changed during the upgrade from the SLC1655 to the SLC1657 architecture. The register was increased from five to seven bits to support a larger register memory.

5.7 INSTRDEC Entity

Other entities that use this module: TOPLOGIC
Other entities used by this module: NONE

The INSTRDEC entity provides instruction decoding, routes the address bus, routes the data source, selects the ALU function, and provides master control for the core. Each of these functions is ultimately controlled by the instruction op-code.

The SLC1657 uses a RISC, or *reduced instruction set computer* architecture. One major feature of the RISC architecture is that it uses an *unencoded* instruction stream. That means that control logic is embedded within the instruction itself. This is opposed to CISC, or *complex instruction set computer* architectures, which encode their instructions in an intermediate encoding scheme.

The term 'decode', when applied to the INSTRDEC entity, means that it decodes the specific register address, and does not imply that it decodes an intermediate op-code. The unencoded instruction stream of the RISC processor gives them the advantage of speed and simplicity. Each op-code has direct control logic information that does not need very much decoding. This shrinks the amount of required logic, and speeds up the computer.

Another feature of RISC processors is the use of separate instruction and data buses. This is often called a Harvard architecture, and is useful because it alleviates the need for a shared bus. Shared buses create bottlenecks (in terms of both speed and logic size) because they are used to pass both instructions and data. Furthermore, they usually require the use of three-state buses, which tend to make them less portable across various FPGA devices.

The disadvantage of RISC processors is that they require relatively wide op-codes, and fast instruction memory. For example, the core has a twelve-bit wide memory. This is

opposed to most CISC microcontrollers, which usually have eight-bit wide memories. CISC processors can encode up to 256 instructions in each op-code, whereas the SLC1657 RISC processor has only thirty-two separate instructions.

The instruction speed problem is diminished in the core because of it's instruction pre-fetch capability. This provides one full clock cycle to create a new address, and fetch the op-code from memory.

Table 5-3 shows the list of SLC1657 instructions sorted by binary op-code and addressing type. There are five general instruction types:

- IMPLICIT
- STANDARD
- BITWISE
- BRANCH
- IMMEDIATE

5.7.1 Implicit Instructions

Implicit instructions are those where the function of the op-code is implicitly defined. For example, the RWT instruction implicitly defines a fixed function, and has no explicit address.

During implicit instructions, the INSTRDEC entity decodes all twelve bits of the op-code, and asserts a signal associated with the instruction. For example, during a PWRDN instruction the PWRDN signal is asserted.

Signals CEPC0, CEPC1, CEPC2, CETCO, PWRDN and RWT are generated by implicit instructions. There are no instruction related signals generated by the NOP instruction.

During a PWRDN instruction the core suspends instruction fetches, and halts most of the internal logic. This reduces current consumption. However, the MCLK, MCLK_4 and MCLK_16 clocks continue to operate, as these are used by the watchdog timer. The actual reduction in current consumption after the PWRDN instruction depends upon the target device technology.

Table 5-3. Binary op-codes.

Binary Op-code	Addressing Mode	Mnemonic	ALF	Cycles
0000 0000 0000	IMPLICIT	NOP	PASN	1
0000 0000 0010	IMPLICIT	MOVT	PASN	1
0000 0000 0011	IMPLICIT	PWRDN	PASN	1
0000 0000 0100	IMPLICIT	RWT	PASN	1
0000 0000 0PPP	IMPLICIT	MOVP	PASN	1
0000 00DR RRRR	STANDARD	MOVA[19]	PASN	1
0000 01DR RRRR	STANDARD	CLR/CLRA	CLR	1
0000 10DR RRRR	STANDARD	SUB	SUB	1
0000 11DR RRRR	STANDARD	DEC	DEC	1
0001 00DR RRRR	STANDARD	OR	OR	1
0001 01DR RRRR	STANDARD	AND	AND	1
0001 10DR RRRR	STANDARD	XOR	XOR	1
0001 11DR RRRR	STANDARD	ADD	ADD	1
0010 00DR RRRR	STANDARD	MOV	PASZ	1
0010 01DR RRRR	STANDARD	NOT	NOT	1
0010 10DR RRRR	STANDARD	INC	INC	1
0010 11DR RRRR	STANDARD	DECSZ	DEC	1(2)
0011 00DR RRRR	STANDARD	ROR	ROR	1
0011 01DR RRRR	STANDARD	ROL	ROL	1
0011 10DR RRRR	STANDARD	SWPN	SWPN	1
0011 11DR RRRR	STANDARD	INCSZ	INC	1(2)
0100 BBBR RRRR	BITWISE	BCLR	BCLR	1
0101 BBBR RRRR	BITWISE	BSET	BSET	1
0110 BBBR RRRR	BITWISE	BTSC	PASN	1(2)
0111 BBBR RRRR	BITWISE	BTSS	PASN	1(2)
1000 VVVV VVVV	BRANCH	RET	PASN	2
1001 VVVV VVVV	BRANCH	BSR	PASN	2
101V VVVV VVVV	BRANCH	BRA	PASN	2
1100 VVVV VVVV	IMMEDIATE	MOVI	PASN	1
1101 VVVV VVVV	IMMEDIATE	ORI	OR	1
1110 VVVV VVVV	IMMEDIATE	ANDI	AND	1
1111 VVVV VVVV	IMMEDIATE	XORI	XOR	1

Notes: 'B' specifies a bit number; 'D' specifies a destination (0 → accum; 1 → register); 'P' specifies a port number; 'R' specifies a register number; 'V' is an immediate operand; '1(2)' indicates a conditional branch instruction with one or two cycles.

Once asserted, the PWRDN signal remains asserted until [MRESET] is asserted. [MRESET] is asserted either by a watchdog timer reset [WRESET], an external reset [RESET], or a programming reset [PRESET]. Also, the PWRDN signal is eventually connected to the external [SLEEP] signal.

The PWRDN instruction also causes the watchdog timer and prescaler to reset. This is exactly the same situation as if an RWT instruction were executed. Since the [PWRDN] signal remains asserted until [MRESET], the [RWT] signal is generated using the circuit

[19] Operand 'D' is always '1' for this instruction.

shown in Figure 5-2. That circuit causes [RWT] to be pulsed once when the PWRDN instruction is generated.

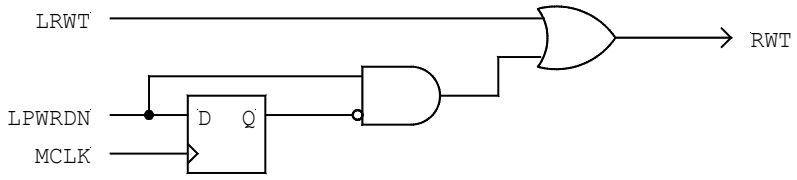


Figure 5-2. Circuit used to generate [RWT] during a PWRDN instruction.

5.7.2 Standard Instructions

Standard instructions are those where the source and the destination register are both defined within the op-code itself. The register number is defined by the lower six bits of the op-code. The destination of the instruction can be the accumulator or a register. For example, during an ADD instruction the source of the data is taken from a register, and is ADD'ed to the accumulator. The destination of the instruction can be either a register (where the source and destination register must be the same) or the accumulator.

The destination is determined by instruction bit INS(5). When this bit is asserted the destination is a register, and when negated the destination is the accumulator. The INSTRDEC entity asserts signal CEACC (Clock Enable ACCumulator) whenever the destination is the accumulator. When the destination is a register, then the appropriate clock or write enable signal is asserted.

During two cycle instructions (e.g. INCSZ) the register destination data is actually latched twice during the cycle. This does not cause a problem, as both data sets have identical numbers.

5.7.3 Bitwise Instructions

Bitwise instructions operate only on registers, and not on accumulator data. Three bits within the instruction defines the bit number to be operated upon. For example, during a BCLR operation, data is taken from a register, the bit specified in the op-code is cleared, and the result is placed back into the source register. In some cases, the ALULOGIC entity will decode the bit number.

5.7.4 Branch Instructions

Branch instructions cause program execution to jump to another location. Eight or nine bit addresses are embedded within the op-code (depending upon the instruction). For example, the BRA instruction causes program the program to begin executing at a new location as specified by the lower nine bits of the op-code. The BSR instruction, however, uses the lower eight bits of op-code. Therefore, subroutines must reside in the lower half of instruction memory.

A portion of each branch instruction is interpreted and handled by the PROGCNTR entity.

5.7.5 Immediate Instructions

Immediate instructions incorporate eight bits of data within the op-code itself. This data immediately 'acts' with the accumulator in the ALU. The result is placed back into the accumulator. For example, the ORI instruction OR'es eight bits of data (located in the op-code) with the accumulator, and places the result back into the accumulator.

5.7.6 ADR Router [ADR(6..0)]

The address router generates the address bus [ADR(6..0)]. It is used by clock enable decode, the register multiplexor and by the general purpose register memories.

The address [ADR(6..0)] is routed using the logic shown in Figure 5-3. This value is concatenated from the register bank select bits RB1:RB0, which are carried by IDX(6..5).

The address embedded in the lower five bits of the op-code [INS(4..0)] is monitored, and in most cases is routed directly to [ADR(4..0)]. However, if register zero is selected [INS(4..0) = 00000], then the value in the index register [IDX(4..0)] is routed to [ADR(4..0)].

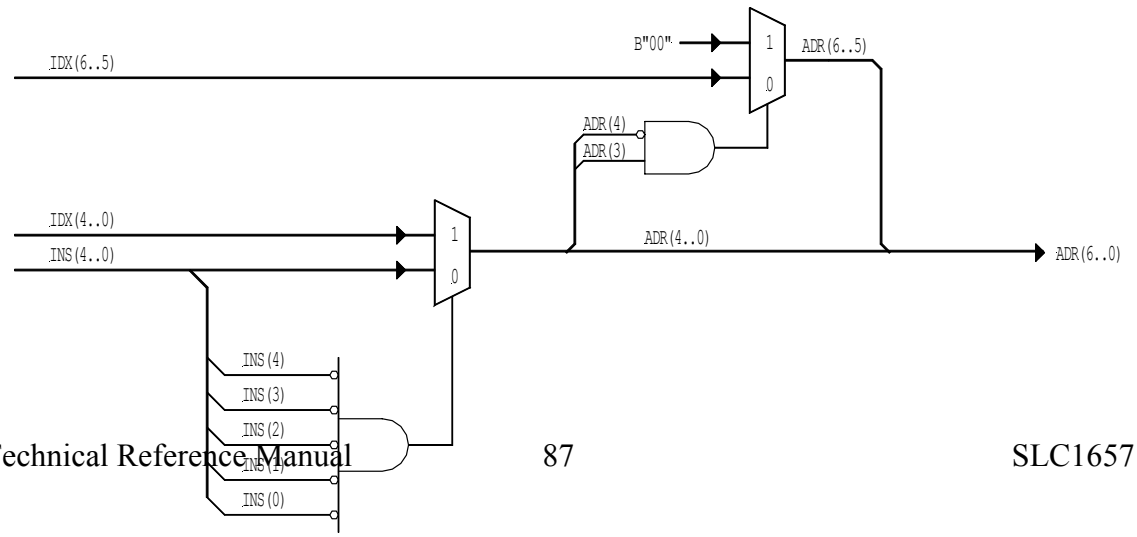


Figure 5-3. INSTRDEC logic for generation of [ADR(6..0)].

If the SHARED GENERAL PURPOSE registers are selected, then the two most significant bits [ADR(6..5)] are set to zero. This maps all accesses to this region down to the lower bank of register RAM. This allows a contiguous RAM block to serve as the register RAM, and simplifies the integration of the core.

The INSTRDEC entity also uses [ADR(6..0)] to (a) generate data selection logic [SEL(1..0)], and (b) to decode clock enable signals CEIDX, CEPRC, CEPT0, CEPT1, CEPT2, CESTA, CETMR and WERAM.

5.7.7 SEL Router [SEL(1..0)]

The SEL router bus [SEL(1..0)] is used by the data source multiplexor to determine if data should come from the special purpose registers, the general purpose registers, the instruction stream or the accumulator.

During implicit instructions, the data source multiplexor is not used. During standard (except for MOVA) and bitwise instructions, the data source comes from the special or general purpose registers. During branch or immediate instructions, the data source comes from the instruction stream. During the MOVA instruction the data source comes from the accumulator.

5.7.8 ALF Function Generator [ALF(3..0)]

The INSTRDEC entity encodes the arithmetic logic unit (ALU) function type onto the [ALF(3..0)] bus. This four-bit bus causes the ALU to perform specific functions. The encoding of [ALF(3..0)] is described with the ALULOGIC entity. The logic which is used to generate [ALF(3..0)] is shown in Figure 5-4.

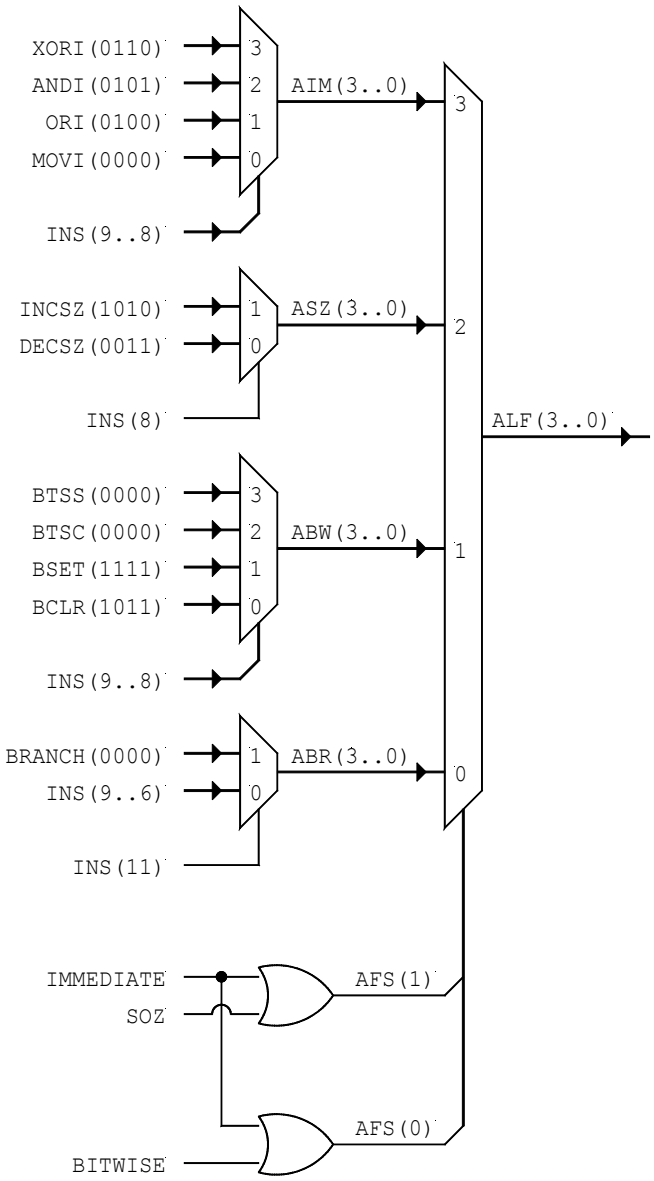


Figure 5-4. ALF function generator.

5.7.9 [ENDCYC] Generator

Assertion of the [ENDCYC] signal causes a new instruction to be latched into the instruction register.

The [ENDCYC] signal is asserted during all implicit, single cycle standard, single cycle bitwise and immediate instructions. It is delayed one cycle during two cycle standard, two cycle bitwise and branch instructions.

The cycle is also extended whenever the program counter is the destination. For example, the ADD 0x002 instruction causes the program counter to be added to the accumulator. The result is then placed back into the program counter. This creates a double cycle because an address must be flushed from the program counter.

During PWRDN instructions the [ENDCYC] signal remains negated. There are no instruction fetches during this time, and [ENDCYC] remains asserted until [MRESET] is generated.

5.7.10 Reset Operation

The [MRESET] signal is not used by the INSTRDEC entity. Whenever [MRESET] is asserted, the REG12CRN entity (instruction register) is reset, thereby generating a NOP instruction. Therefore, the first instruction performed after every reset is technically a NOP instruction.

5.8 INTRCONV Entity

Other entities that use this module: TSTBENCH (many)
Other entities used by this module: NONE

The INTRCONV entity is used only for test benches. It converts integer to std_logic_vector types (and vice-versa).

Some test benches require conversions between integers and standard logic vectors. If your test bench contains the statement “work.SLV2INTPAK.all”, then it requires the ‘INTRCONV’ file.

5.9 MUX08X04 Entity

Other entities that use this module: TOPLOGIC
Other entities used by this module: NONE

The MUX08X04[20] entity multiplexes four, eight-bit buses.

5.10 MUX08X08 Entity

Other entities that use this module: TOPLOGIC
Other entities used by this module: NONE

The MUX08X08 entity multiplexes eight, eight-bit buses.

5.11 MUX11X04 Entity

Other entities that use this module: PROGCNTR
Other entities used by this module: NONE

The MUX11X04 entity multiplexes four, eleven-bit buses.

5.12 PORTSREG Entity

Other entities that use this module: TOPLOGIC
Other entities used by this module: NONE

The PORTSREG entity handles all of the I/O port activity used in conjunction with the PORT0, PORT1 and PORT2 registers. It does not interact with the port control registers PC0-2.

The PORTSREG entity is really just a set of input and output latches, together with an output strobe. Each entity is configured for 8-bit wide I/O ports.

When writing to a port, the PORTSREG entity latches and holds the data . Data becomes active at the output port at the rising edge of [MCLK], at the end of the instruction cycle that generated the access..

[20] MUXWWXSS specify a class of multiplexors where 'WW' is the width of input and output buses and 'SS' specifies the number of selectors.

The entity also provides output strobes [PTSTB0-2]. Each of these strobes corresponds to port registers PORT0-2. When data is written to a port, the port strobe is asserted for one [MCLK] cycle. This is useful when the ports are used in conjunction with external FIFO buffers.

When reading from a port, the PORTSREG entity latches and holds the incoming data. Data is latched at the rising edge of [MCLK], at the beginning of the instruction cycle that generated the access.

For more information please refer to the descriptions of the PORT0-2 register and the I/O options elsewhere in this manual.

5.13 PRESCALE Entity

Other entities that use this module: TIMRCNTR
Other entities used by this module: TIMRSYNC

The PRESCALE entity handles the counter/timer/watchdog prescaler logic. A block diagram of the PRESCALE entity is shown in Figure 5-5.

The entity determines the source of the [WRESET] and [TMRSYN] signals depending upon the state of the TCO register (via TCS, TSE, ASGN and PS(2..0)). The [WRESET] signal can be driven directly from the watchdog timer, or (for longer watchdog time-outs) through the prescale counter. Similarly, the source of the timer increment signal [TMRSYN] can be routed from the [TMRCLK] or [MCLK_4] signals directly through the prescale counter.

For more information refer to the descriptions of the TIMRCNTR, TIMRSYNC and WATCHDOG entities.

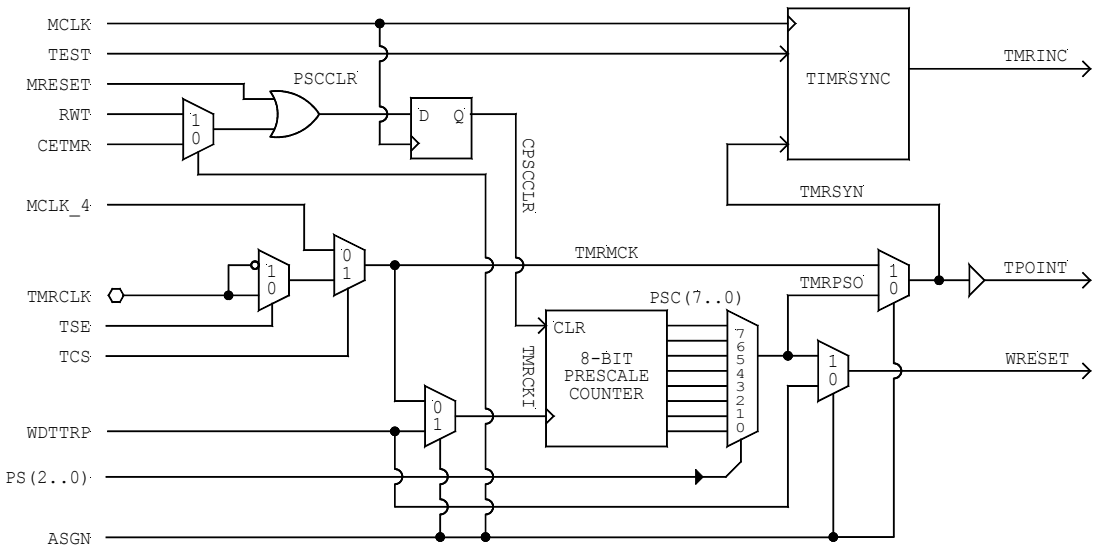


Figure 5-5. Block diagram of the PRESCALE entity.

5.14 PROGCNTR Entity

Other entities that use this module: TOPLOGIC
Other entities used by this module: BUC11CPP, MUX11X04, REG11CNN

The PROGCNTR entity handles all of the program counter and stack control. A block diagram of the entity is shown in Figure 5-6, and a timing diagram in Figure 5-7.

The PROGCNTR entity was changed during the upgrade from the SLC1655 to the SLC1657 architecture. The addressable program memory range was increased from nine to eleven bits.

Features of the PROGCNTR entity include:

- A program counter.
- Two level stack and control logic (used during BSR and RET instructions).

After reset, the program counter presets to the top of memory at address 0x7FF. This is the initial instruction address. A BRA instruction at 0x7FF causes the program to branch to the new location. A NOP instruction at 0x7FF causes program execution to begin at address 0x000.

The reset instruction address can be changed by modifying the hardware. This is done by modifying the BUC11CPP counter to preset to some address other than 0x7FF. For ex-

ample, if only 512 words of program memory are used, then the reset address could be changed to 0x1FF. For more information about changing the preset address, please refer to the BUC11CPP entity and related description.

During most standard, single clock instructions (NOP, ADD etc.), the program counter increments one count at every rising edge of [MCLK]. Some instructions, however, can also modify the program counter. For example, ADD 0x02 adds the accumulator to the lower eight bits of the program counter. During this activity the INSTRDEC entity asserts the [CEPRC] signal, which indicates that the program counter should be preloaded from a concatenation of the [ALU(7..0)] and the [STR(6..5)] bus (i.e. bits IB0 and IB1). Bit eight is set to zero at the same time. For more information about this activity, please refer to the internal architecture description in Chapter 2.

During single cycle standard and bitwise instructions (INCSZ, BTSS, etc.), the program counter increments normally. If a skip condition occurs, the INSTRDEC entity negates [ENDCYC], thereby preventing a new instruction from being fetched. However, the program counter will increment twice during these instructions, thereby flushing the instruction stream.

All branch instructions are double cycles. Op-codes for these instructions (BRA, BSR and RET) are decoded by the PROGCNTR entity. During branch instructions the program counter will increment during the first clock, and then preload with the new address during the second. This flushes the instruction stream.

The BSR instruction causes a new program counter to be loaded during the second half of the cycle. At the same time, the current (return) address is saved in one of two stack locations. The stack is implemented with eleven bit register entities REG11CNN. A stack pointer (SP1 and SP2) determines which of the two stack registers to use. The stack pointer always increments at the end of the first clock cycle.

The RET instruction causes a new program counter to be loaded from one of the two stack registers, and is routed to the D input pins of the BUC11CPP counter using multiplexor MUX11X04. The current state of the stack pointer (SP1 and SP2) determines which stack register to preload from.

During PWRDN cycles, the program counter is stopped, thereby reducing power consumption.

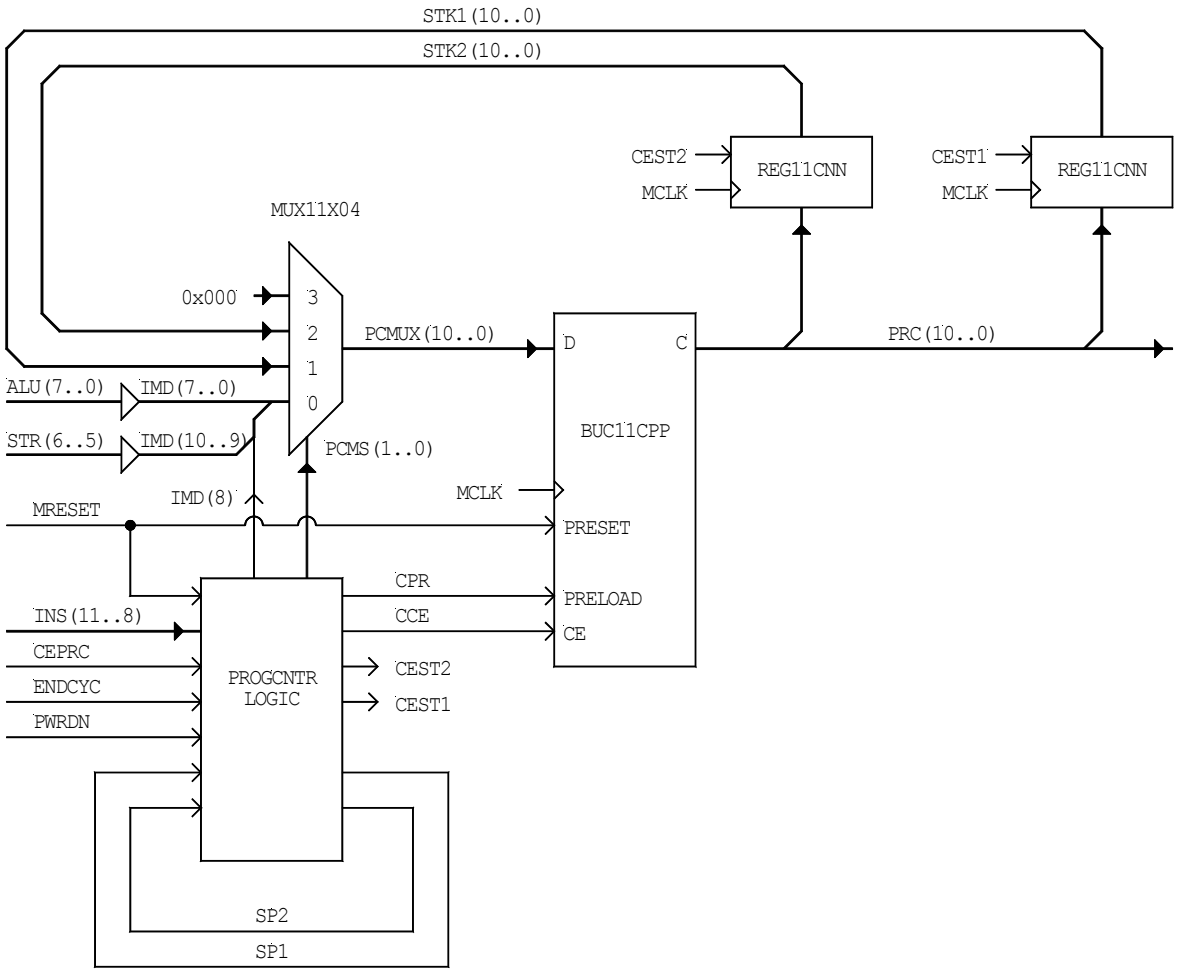


Figure 5-6. PROGCNTR entity block diagram.

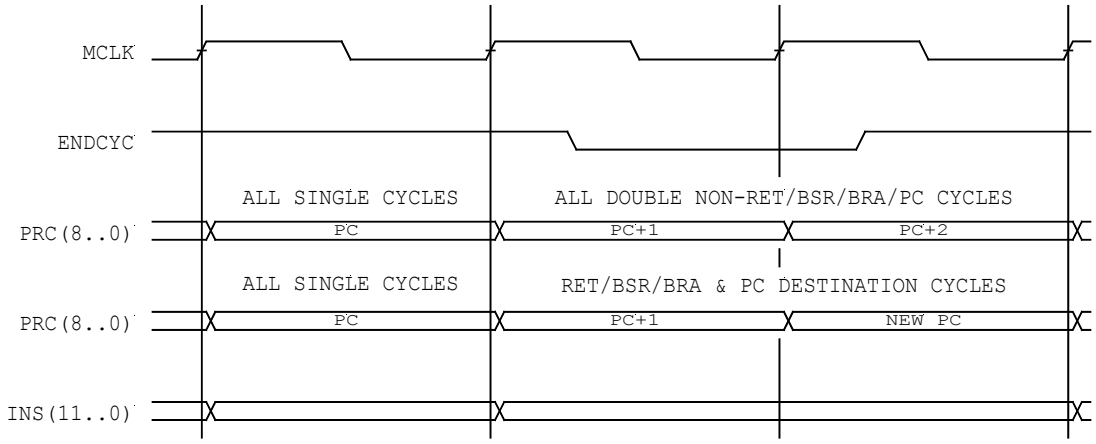


Figure 5-7. PROGCNTR timing diagram.

5.15 REG08CNN Entity

Other entities that use this module: TOPLOGIC
Other entities used by this module: NONE

The REG08CNN[21] entity is an eight-bit register with clock enable.

5.16 REG08CPN Entity

Other entities that use this module: TOPLOGIC
Other entities used by this module: NONE

The REG08CPN entity is an eight-bit register with clock enable and synchronous preset.

[21] REGWWXYZ entities specify a class of positive edge triggered registers where 'WW' is the width of the register in bits (04, 08 etc.), 'X' specifies the presence of a clock enable (C: clock enable; N = no clock enable), 'Y' specifies set, or reset logic (P: synchronous preset; R: synchronous reset; B: synchronous preset and reset; S: asynchronous set; C: asynchronous clear; A: asynchronous set and clear; and N: no set or preset) and 'Z' specifies output enable logic (O: output enable, N: no output enable).

5.17 REG11CNN Entity

Other entities that use this module: PROGCNTR
Other entities used by this module: NONE

The REG11CNN entity is an eleven-bit register with clock enable.

5.18 REG12CRN Entity

Other entities that use this module: TOPLOGIC
Other entities used by this module: NONE

The REG12CRN entity is a twelve-bit register with clock enable and synchronous reset.

5.19 RESETGEN Entity

Other entities that use this module: TOPLOGIC
Other entities used by this module: NONE

The RESETGEN entity provides two functions: (a) it resets the core and (b) it generates the 'TO' (timeout) and 'PD' (power-down) bits. A block diagram of the entity is shown in Figure 5-8.

There are three sources that can generate a microcontroller reset [MRESET]. They include the external reset [RESET], the watchdog reset [WRESET] and the ROM emulation (programming) reset [PRESET]. Each of these signals must be asserted for at least one [MCLK] cycle.

The RESETGEN entity provides an automatic power-up reset capability. This assumes that the FPGA or ASIC target architecture guarentees that all flip-flops power-up in their negated (logic '0') state. Most architectures have this capability. However, the user may wish to generate an external power-up reset instead.

The 'TO' bit is asserted whenever there is a power-up reset or an external reset ([RESET] or [PRESET]) after a PWRDN instruction.

The 'PD' bit is asserted whenever there is a power-up reset or a non-PWRDN watchdog reset [WRESET].

The ‘TO’ and ‘PD’ bits are monitored by software in the STATUS register. They can be used to determine the cause of a reset (external, watchdog, etc.).

The ‘TO’ and ‘PD’ bits are both set after a emulation ROM programming reset [PRESET]. This mimics a power-up reset after downloading new code.

The ‘TO’ and ‘PD’ bits can also be set with the reset watchdog instruction (RWT). This capability is useful in some applications.

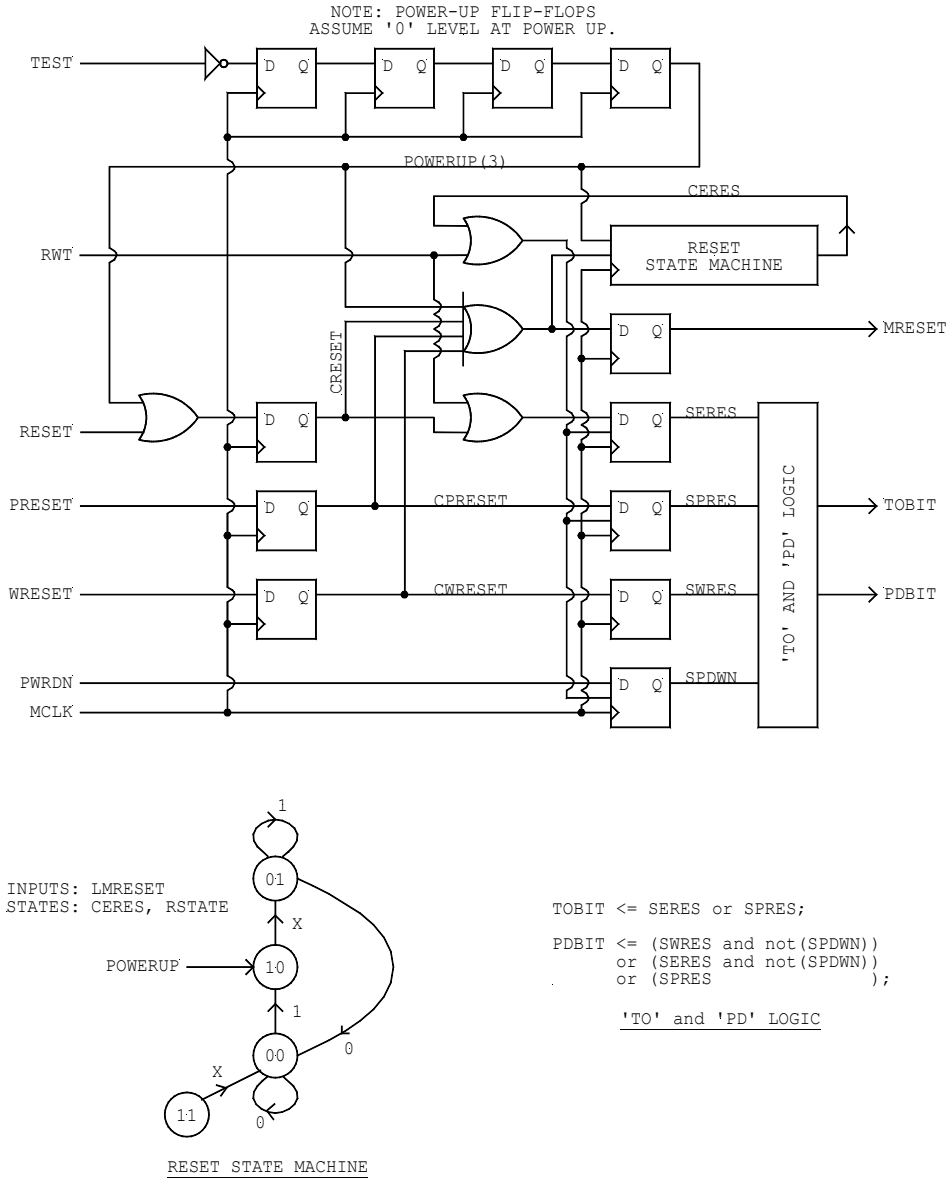


Figure 5-8. Block diagram of the RESETGEN entity.

The power-up condition of these bits can potentially cause some portability problems. Both must be powered up in the asserted (i.e. set) condition. To guarantee this operation, the internal logic in the core assumes that the target device powers up all flip-flops in the negated (i.e. zero) condition.

The circuit should be portable and reliable if *all of the flip-flops* are negated in response to a power-up reset. If power-up state of your particular FPGA or ASIC architecture is indeterminate, then (a) an external power-up function must be added to your circuit, and the VHDL code altered appropriately or (b) the 'TO' and 'PD' bits should be ignored by the application software.

For more information refer to the descriptions of the STATUS register and the STATSREG entity.

5.20 STATSREG Entity

Other entities that use this module: TOPLOGIC
Other entities used by this module: NONE

The STATSREG entity handles the logic for the STATUS register. This register includes the condition code bits ('Z', 'C' and 'NC'), the instruction bank select bits ('IB0', 'IB1') as well as the 'TO' and 'PD'bits.

The STATSREG entity was changed during the upgrade from the SLC1655 to the SLC1657 architecture. The IB0 and IB1 bits were added to increase the addressable program memory range from nine to eleven bits.

Figure 5-9 shows an example of how the three condition code bits ('Z', 'C' and 'NC') are handled. During every instruction, the ALULOGIC entity presents the result of the condition codes on [STA(6..0)]. For more information about the encoding of [STA(6..0)] please refer to the functional description of the ALULOGIC entity.

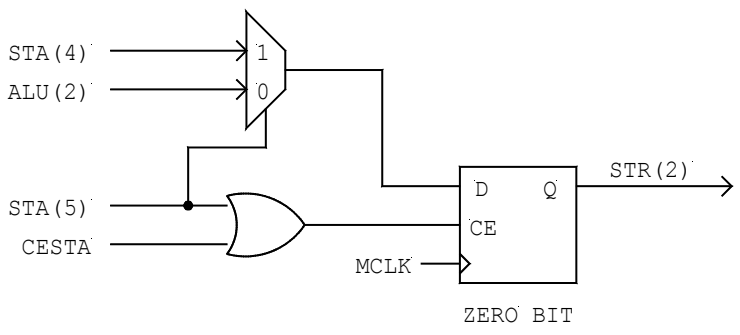


Figure 5-9. STATSREG logic for condition code 'Z' bit logic.
This is similar to how it handles the 'C' and 'NC' bits.

Each condition code bit has both a result condition and a clock enable signal. For example, during a NOT instruction the 'Z' bit is set. If the result of the NOT instruction is zero, the ALU asserts the 'Z' bit [STA(4)]. If the result is not zero, the ALU negates the 'Z' bit [STA(4)]. In either case, the ALU asserts the 'Z' bit clock enable signal [STA(5)], which indicates to the STATSREG entity that it should latch the bit. Since the NOT instruction does not alter the 'C' and 'NC' bits, [STA(1)] and [STA(3)] remain negated.

If the instruction requires writing to the STATUS register, then the result presented by the ALULOGIC entity has precedence over the write data itself. For example, a CLR 0x03 instruction will result in the 'Z' bit being set. For this reason, the consequences of writing to the STATUS register should be carefully evaluated. Instructions that do not set the condition code bits (such as BCLR and BSET) are recommended for this application.

The STATUS register 'TO' and 'PD' bits are generated by the RESETGEN entity. The bits are read-only, and are not affected by a write to the STATUS register.
For more information refer to the description of the RESETGEN entity.

5.21 TCOPTREG Entity

Other entities that use this module: TOPLOGIC
Other entities used by this module: NONE

The TCOPTREG entity contains the timer/counter option register. The lower six bits are composed of a six-bit register with clock enable and synchronous preset. Bit six is a one-bit register with clock enable.

5.22 TIMRCNTR Entity

Other entities that use this module: TOPLOGIC
Other entities used by this module: BUC08NNP, PRESCALE, WATCHDOG

The TIMRCNTR entity includes all of the components for the timer/counter. Figure 5-10 shows the various parts. For more information, refer to the descriptions of the BUC08NNP, PRESCALE and WATCHDOG entities.

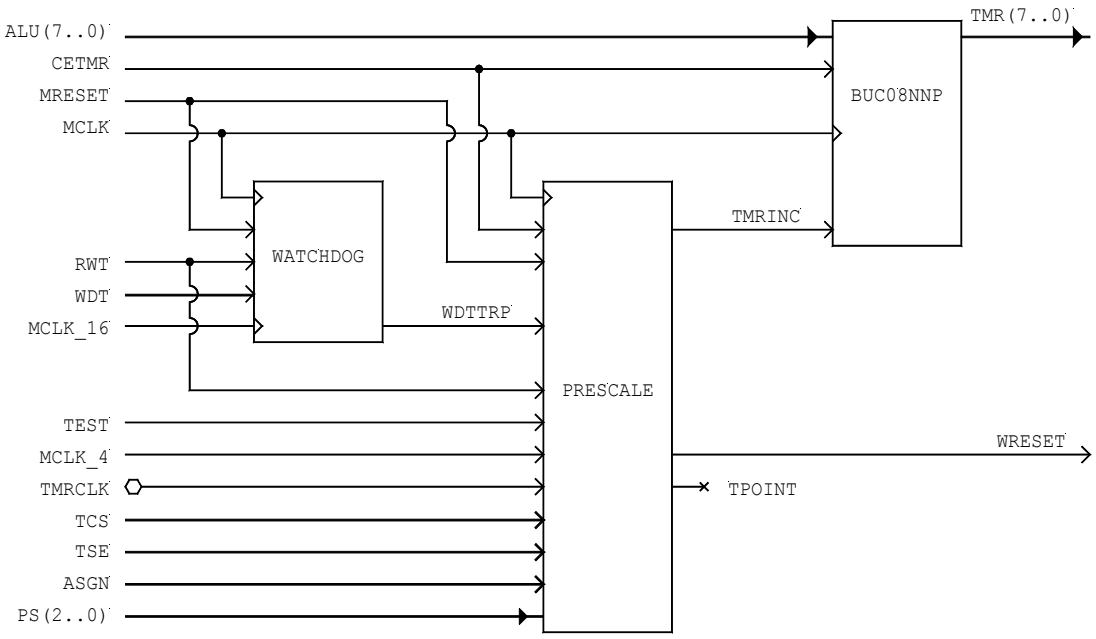


Figure 5-10. TIMRCNTR entity block diagram.

The WATCHDOG entity includes a 15-bit ripple counter for creating the watchdog timer. The watchdog timer is enabled by signal WDT, which is set in the TCO register. The timeout delay of the watchdog is determined by the clock frequency [MCLK / 16] and the use of the prescaler. When the watchdog trips, signal [WDTTRP] is asserted. If the prescaler is not attached to the watchdog (as indicated by the [ASGN] signal), then output [WRESET] is asserted, and the core is reset. If the prescaler is attached to the watchdog, then the watchdog can be programmed to reset the microcontroller after multiple assertions of [WDTTRP].

If the PRESCALE entity is attached to the timer (as indicated by the [ASGN] signal), then it can be used to prescale the source of the timer/counter input. In this mode, the source can be either the [MCLK_4] or the [TMRCLK] pins. Every event from the output of the PRESCALE entity increments the BUC08NNP counter by one bit.

The BUC08NNP entity can be preloaded by any instruction. It is not affected by reset.

5.23 TIMRSYNC Entity

Other entities that use this module: PRESCALE
Other entities used by this module: NONE

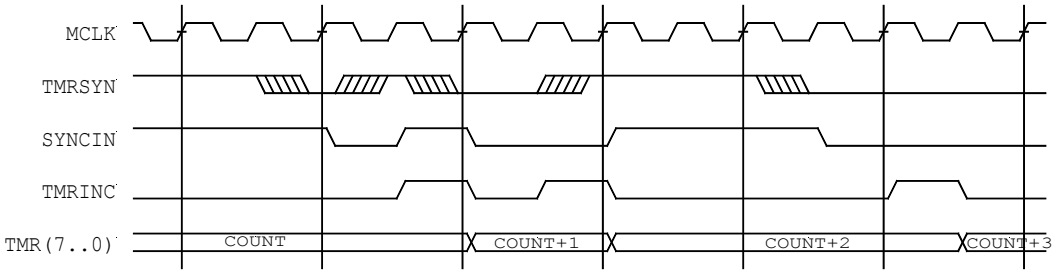
The TIMRSYNC entity is used by the PRESCALE entity. Its purpose is to generate a single increment pulse [TMRINC] after the falling edge of every timer/counter source signal.

Figure 5-11 shows the timing, block and state diagrams for the TIMRSYNC entity. The output of the timer/counter module [TMRSYN] is sampled at the rising edge of [MCLK] with a flip-flop. The flip-flop is used to prevent metastable and race conditions from entering the TIMRSYNC entity, as the [TIMRSYN] input can be asynchronous to [MCLK].

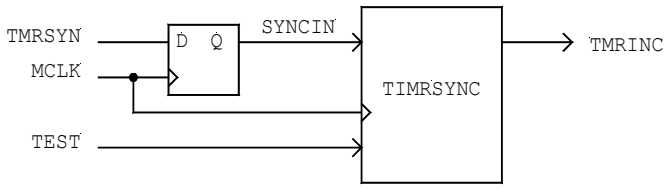
Once the [TMRSYN] input is synchronized, it is passed to the TIMRSYNC state machine using signal [SYNCIN]. The state machine monitors this input, and generates a single output pulse [TMRINC] after every falling edge on [SYNCIN]. This increments the timer counter at the next rising edge of [MCLK].

The timing diagram also shows why there is a minimum high and low time specified for the external timer/counter input. For example, when the TIMRSYNC entity is driven by [TMRCLK], then the input signal must be high for at least one positive [MCLK] edge, and low for another. If [TMRCLK] is synchronized with [MCLK] external to the core, then the signal must meet the setup times for the synchronizer flip-flop. If [TMRCLK] is asynchronous to [MCLK], then the high and low times must exceed the period of the [MCLK] frequency.

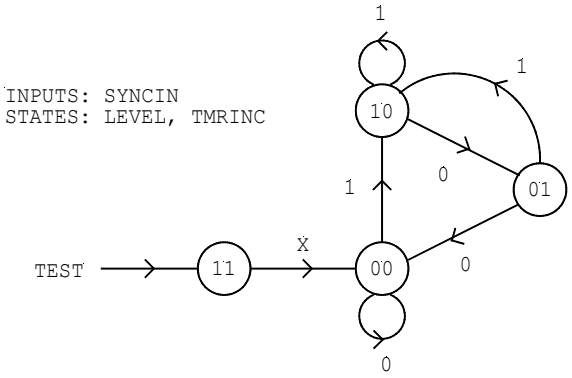
The TIMRSYNC state machine is unaffected by reset. The [TEST] input on the TIMRSYNC entity is used to reset the state machine to an initial condition for test bench purposes, and should be negated during normal operation. This is because the TIMRSYNC state machine is a ‘self-starting’ type (i.e. any initial state will be tolerated).



TIMING DIAGRAM



BLOCK DIAGRAM



STATE DIAGRAM

Figure 5-11. Timing, block and state diagrams for the TIMRSYNC entity.

5.24 TOPLOGIC Entity

Other entities that use this module: NONE
Other entities used by this module: See Figures 5-12 and 5-13.

The TOPLOGIC entity is the highest hierarchical module, and ties together all of the other components in the microcontroller. Figure 5-12 shows a block diagram of the entity, and Figure 5-13 shows the hierarchical relationship of this entity to the other entities that it uses.

The test bench for the TOPLOGIC entity uses a series of test vector files. These are read by the test bench, and include:

- `VECTADDR.TXT`
- `VECTIBNK.TXT`
- `VECTINIT.TXT`
- `VECTINST.TXT`
- `VECTPORT.TXT`
- `VECTPROG.TXT`
- `VECTRBNK.TXT`
- `VECTTIMR.TXT`

5.25 WATCHDOG Entity

Other entities that use this module: TIMRCNTR
Other entities used by this module: NONE

The WATCHDOG entity has a 15-bit ripple counter which counts up after an [MRESET] or [RWT] reset. When the counter reaches its terminal count, the output [WDTRP] is asserted if bit [WDT] is asserted. The counter increments at every rising edge of [MCLK_16].

The ripple counter has an asynchronous reset. This reset is tripped whenever the [MRESET] or [RWT] signals are asserted.

For more information refer to the descriptions of the PRESCALE, TIMRCNTR and TIMRSYNC entities.

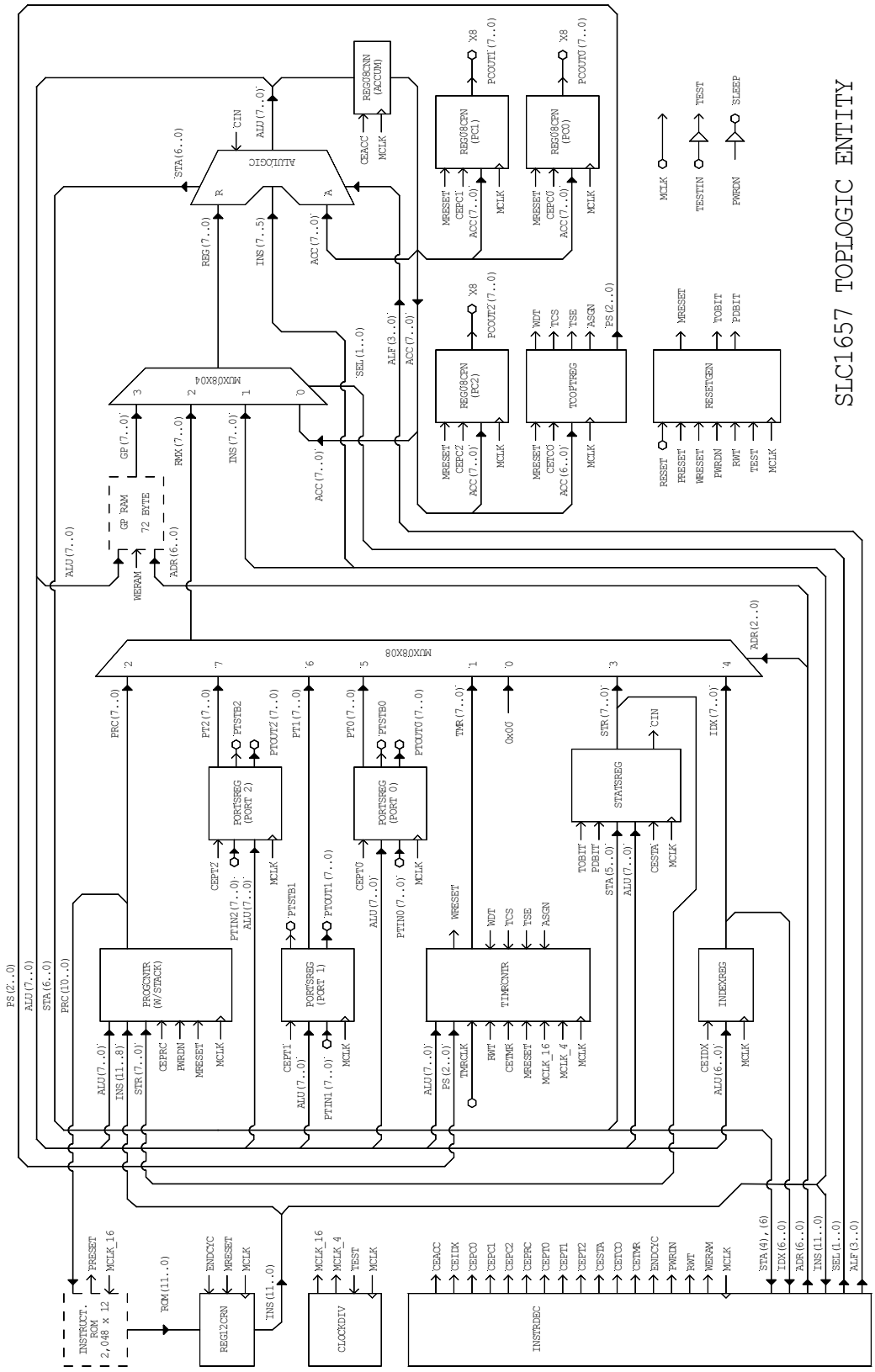


Figure 5-12 TOPLOGIC entity block diagram.

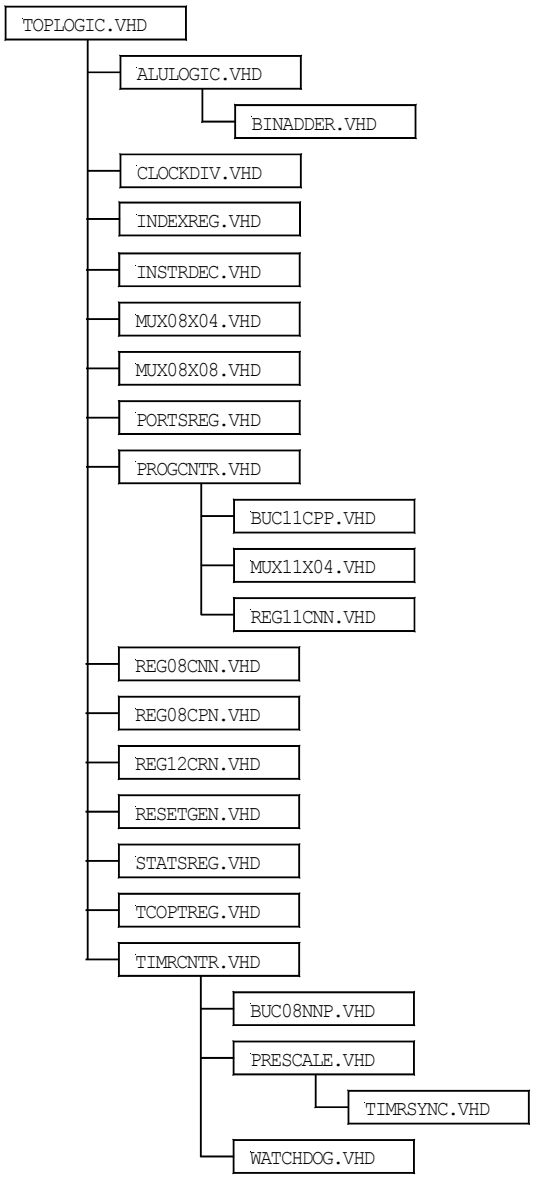


Figure 5-13. TOPLOGIC entity hierarchy.

6.0 Implementation on the Xilinx Spartan 2 FPGA

This chapter describes the steps needed to integrate the SLC1657 onto a Xilinx Spartan 2 FPGA. An exercise is presented whereby a four function calculator is implemented on an evaluation board.

The purpose of this chapter is to:

- Learn about the SLC1657 Evaluation Kit for the Xilinx Spartan 2 FPGA.
- Learn the steps needed to integrate a simple system-on-chip.
- Demonstrate how to simulate the TOPLOGIC entity.
- Demonstrate how to synthesize an IP core.
- Create the register RAM using Xilinx Spartan 2 distributed RAM.
- Create the instruction ROM using Xilinx Spartan 2 block RAM.
- Create a parallel port interface for download and test of application code.
- Integrate the TOPLOGIC core with RAM, ROM and parallel port interface.
- Download and run a 'C' application program for a 10-key calculator.
- Create a fixed PROM.

The following hardware and software tools are used in the exercises:

- PeakVHDL simulation and synthesis tools from Protel International.
- Xilinx Alliance Series Place & Route Software.
- SLC1657 evaluation kit for Xilinx Spartan 2 FPGA.
- CC5X 'C' compiler from B Knudsen Data.
- DOWNLOAD software for testing application code.
- MAKEXCOE software for integration of a ROMable application software.
- PROM programmer[22].

[22] The Needhams EMP-30 PROM programmer was used for the exercise (www.needhams.com).

6.1 Evaluation Kit for Xilinx Spartan 2 FPGA

The evaluation kit for Xilinx Spartan 2 FPGA allows the user to evaluate and test the SLC1657 microcontroller. The kit includes:

- Evaluation board with Xilinx Spartan 2 XC2S50-5 FPGA (see Figure 6-1).
- PROMs for demonstration and calculator functions.
- 16 x 1 LCD display.
- 20-key keypad.
- 5-MHz crystal oscillator.
- 1 KHz RC oscillator.
- 9V battery pack.
- Demonstration program.
- Calculator program.
- PC parallel port download cable and software.
- Technical reference manual.

The evaluation board comes with two embedded software programs. These are 'XDMO', a generic demonstration PROM and 'XCLC', a calculator program. Each resides on a PROM, which contains both the hardware for the SLC1657 microcontroller and the software application programs.

6.1.1 XDMO Software

The XDMO embedded ROM program demonstrates how the SLC1657 can be completely integrated into an FPGA. This includes RAM, ROM and I/O elements. The XDMO embedded ROM demonstration displays the features of the core, and also has a 'stopwatch' function. Follow these simple instructions to operate XDMO:

1) Remove the evaluation board from the anti-static bag[23].

2) Verify that the 8-pin ROM labeled 'XDMO' is located in DIP socket U5 (to the right of the LCD display). There is a 'spare' PROM socket located at U2 (at the top of the board). This socket is not active, and only serves as a holder for the unused PROM. You might need to switch the PROMs around.

3) Connect the +9 VDC battery pack to the evaluation board using connector J1.

[23] The board should be handled at an approved anti-static workstation.

4) Verify that the core boots up, and that display on the evaluation board reads 'SILICORE SLC1657'. This indicates that the microcontroller inside of the FPGA has reset and is running the application code.

5) Push switch 'S17' (the switch marked '0').

6) The features of the core scroll by on the display.

7) Push switch 'S18' (the switch marked '.').

8) Verify that a counter display "00:00 0/10th" appears. Pushing switch S18 ('.') always starts the 'stopwatch' application. Pushing switch S19 ('+/-') starts the stopwatch, and pushing 'S20' ('=') stops it. The stopwatch can be cleared by pushing 'S18' ('.') again. The following table summarizes the switches used by XDMO:

Table 6-1. XDMO Key Functions

Switch	Label	Action
S17	'0'	Marquee of features
S18	'.'	Initiate/clear stopwatch
S19	'+/-'	Start stopwatch
S20	'='	Stop stopwatch

6.1.2 XCLC Software

The XCLC calculator software places the evaluation board into its calculator mode. Install the XCLC PROM into U5 and operate the evaluation board as a four function calculator.

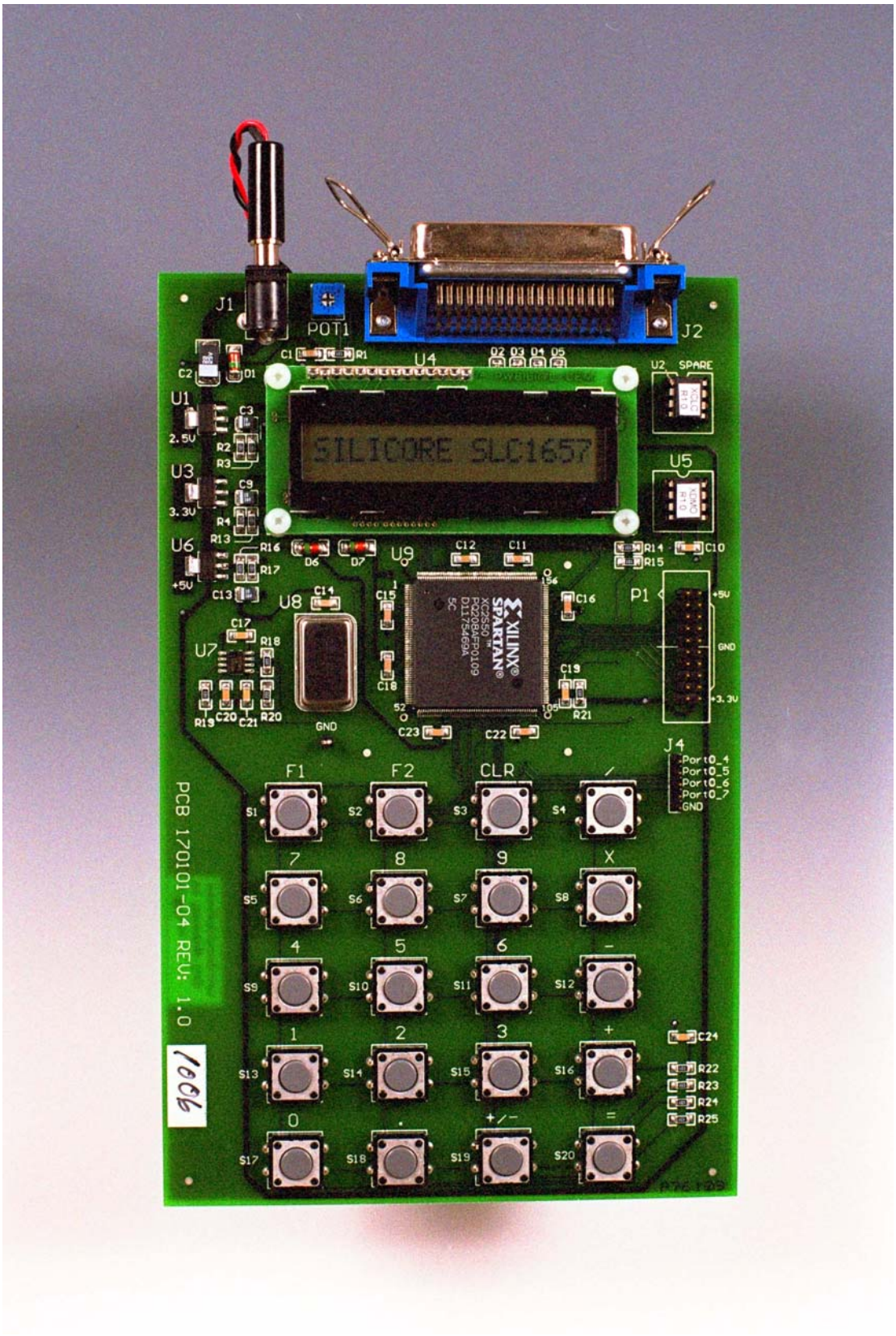


Figure 6-1. Evaluation board for Xilinx Spartan 2 FPGA.

6.2 The XSP2EVAL Exercise

An exercise is given below to better understand the operation of the SLC1657. This creates a system-on-chip called 'XSP2EVAL', which stands for Xilinx SPartan 2 EVALuation system. It's a system-on-chip (SoC) that we'll use to design and run a four function calculator.

The XSP2EVAL system uses several VHDL entities. These are described in detail in section 6.5 (below). The user is encouraged to study the descriptions there, along with the VHDL source code. These entities include:

- XSP2EVAL: Xilinx Spartan 2 Evaluation
- TOPLOGIC: TOP LOGIC design for the SLC1657.
- REGISRAM: REGISter RAM.
- INSTRROM: INSTRuction ROM.
- SEMRMINT: Serial Emulation ROM Interface.
- IBUF, OBUF, IOBUF: I/O pin drivers for Xilinx Spartan 2.

6.2.1 STEP 1 – Simulate the TOPLOGIC Entity

The first step to creating the SLC1657 is to simulate the TOPLOGIC entity. This familiarizes the user with the simulation tools, the SLC1657 IP core and the general operation of all components. This step is identical for all target devices such as Agere, Altera and Xilinx.

Using the Protel PeakVHDL simulation tool, perform the following operations:

1) Create a new directory called 'TLTEST'. [One has been created for you in the EXAMPLES folder if you wish to use it].

2) Open PeakVHDL and create a new project (following the manufacturers directions). Name the project TLTEST, and put it into the 'TLTEST' folder.

3) Add all of the modules in the TOPLOGIC entity into the project. Be sure to preserve the entity hierarchy. The hierarchy is described with the TOPLOGIC entity in Chapter 5. Each entity can be found in its own unique folder in the 'VHDL_source' directory.

   When simulating with the PeakVHDL product, be sure that the highest level module in the hierarchy is the TOPLOGIC test bench (TSTBENCH.VHD from the TOPLOGIC folder).

Also, the TOPLOGIC test bench simulation will need the corresponding test vector files. These are the files with the '*.txt' extension in the TOPLOGIC folder, and should be copied into the TLTEST directory.

When finished, the project window should look something like that shown in Figure 6-2.

4) Simulate the design using the manufacturers directions. At this point the TOP-LOGIC entity should simulate with no errors.

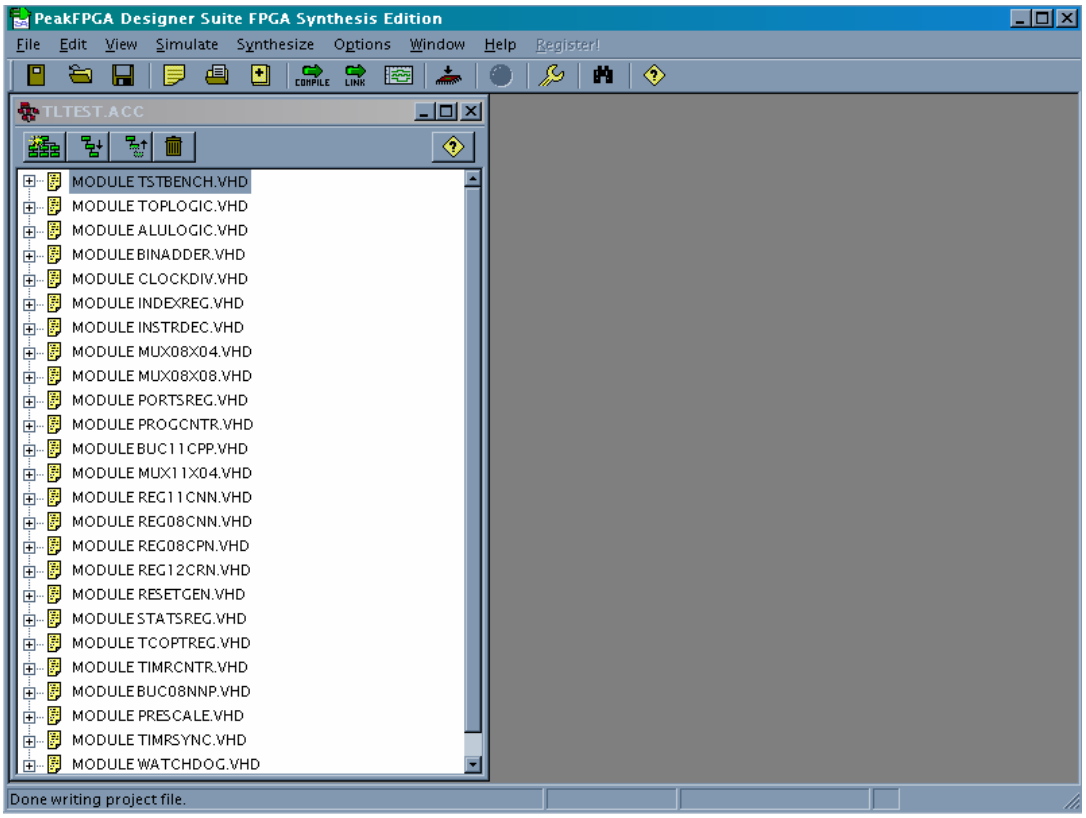


Figure 6-2. PeakVHDL project window.

6.2.2 STEP 2 – Create REGISRAM (Register RAM)

The register RAM is a 128 x 8-bit synchronous memory. It must conform to the FASM guidelines described elsewhere in this manual. There are many ways to build memories, but the simplest is to use the automatic memory generation software that is supplied with most FPGA place & route tools.

Xilinx supplies such a tool with their Alliance Series software. It's called CORE Generator, and is capable of creating the exact memory that's needed. In this example the register RAM will be formed from distributed RAM, meaning that the logic look up tables (LUTs) will be re-configured as RAM. This is opposed to block memory, which is formed from dedicated memory cells on the FPGA. Using the Xilinx CORE Generator tool, create the REGISRAM entity:

1) Create a directory called 'REGISRAM'. [One has been created for you in the Xilinx examples folder].

2) Open the Xilinx CORE Generator tool, and set it up to create a synchronous RAM in the REGISRAM folder. Set up the options thusly:

   Device type: Spartan 2
   File format: VHDL
   Tool type: Other (Protel)
   Netlist bus format: B(I)
   Memory type: distributed
   Component name: regisram
   Depth: 128
   Data width: 8
   Memory type: single port RAM
   MUX construction: LUT based
   Input options: non-registered
   Layout: create RPM

3) Generate REGISRAM.

4) Verify that the REGISRAM folder that you created has a file named 'regisram.edn' in it. This is the EDIF file for the RAM (that we'll use later).

6.2.3 STEP 3 – Create INSTRROM (Instruction ROM)

In the Xilinx Spartan 2, the instruction ROM is actually formed from synchronous block RAMs. This will be configured by the Xilinx Core Generator tool (used above) to form a 2,048 x 12-bit instruction memory. However, the term 'instruction ROM' will be used here, as it is fundamentally a read-only memory.

The XSP2EVAL core implements a parallel port interface called SEMRMINT. This interface allows software to be downloaded to the instruction ROM, thereby making it possible to send application code to the microprocessor. This is very useful for software development purposes.

For now, we'll rely on the download capability to get new application code into the microcontroller. However, later on the INSTRROM entity will be initialized with our application code. Using the Xilinx CORE Generator tool, create the INSTRROM entity:

1) Create a directory called 'INSTRROM'. [One has been created for you in the Xilinx examples folder].

2) Open the Xilinx CORE Generator tool, and set it up to create a synchronous RAM in the INSTRROM folder. Set up the options thusly:

   Device type: Spartan 2
   File format: VHDL
   Tool type: Other (Protel)
   Netlist bus format: B(I)
   Memory type: single port block memory
   Component name: instrrom
   Depth: 2048
   Data width: 12
   Port configuration: read and write
   Global init value: 0 (leave the initialization file box unchecked).

3) Generate INSTRROM.

4) Verify that the INSTRROM folder that you created has a file named 'instrrom.edn' in it. This is the EDIF file for the instruction ROM (that we'll use later).

6.2.4 STEP 4 – Synthesis

The highest level entity/architecture pair in this system is the VHDL source file named 'XSP2EVAL'. This file ties all of the parts of the system together as described in the block and hierarchy diagrams for the XSP2EVAL entity below.

Using the Protel PeakVHDL synthesis tool, perform the following operations:

1) Create a new directory called 'XSP2EVAL'.

2) Open PeakVHDL and create a new project (following the manufacturers directions). Name the project XSP2EVAL, and put it into the 'XSP2EVAL' folder. [This is already done for you in the Xilinx examples folder if you wish to use that.]

3) Add all of the modules in the XSP2EVAL entity into the project. Be sure to preserve the entity hierarchy. The hierarchy is described with the XSP2EVAL entity later in this chapter. The entities relating to TOPLOGIC (e.g. ALULOGIC.VHD)

can be found in its own unique folder in the 'VHDL_source' directory. The entities relating to Xilinx Spartan 2 implementation (e.g. SEMRMINT.VHD) can be found in 'Xilinx' directory.

4) Move the following files into the XSP2EVAL directory: regisram.edn and instrrom.edn. These were created earlier, and are contained in the REGISRAM and INSTRROM directories (respectively).

5) Select 'Spartan 2 Series (EDIF)' in the PeakVHDL synthesis options. Also un-check 'Top Level Module (insert I/O buffers)'. [Note: the I/O buffers are contained in the XSP2EVAL entity, and must not be added by the PeakVHDL synthesis tool].

6) Synthesize the XSP2EVAL system with PeakVHDL.

7) Look in the synthesis log file, and verify that no errors were generated by PeakVHDL.

8) Verify that file 'XSP2EVAL.EDN' is present in the directory. This is the EDIF file created by PeakVHDL.

6.2.5 STEP 5 – Place & Route the Design

The EDIF file created in STEP 4 contains most of the microprocessor logic. The next step is to place and route the design on the Xilinx Spartan 2 FPGA chip. In this example, we'll use the Xilinx Alliance Series software to place and route the design.

Using the Xilinx Alliance Series software tool, perform the following operations:

1) Boot the Alliance design manager.

2) Create a new project. Select 'XSP2EVAL.EDN' as the input file. This was the file that was created in STEP 4, and is the input file for the place and route software.

3) Under 'Part Selector', select the following options:

Family:	SPARTAN2
Device:	XC2S50
Package:	PQ208
Speed Grade:	-5

4) Under 'Constraints File', select 'Custom', and then browse for a file called 'XSP2EVAL.UCF'. This is the user constraints file that contains pin locations, timing specifications and so forth.

5) Place and route the design. Look in the Place & Route report (generated by the Xilinx Design Manager), and verify that there were no errors generated. This report also has statistics for the number of gates used, and so forth.

6.2.6 STEP 6 – Create the PROM

The final step in implementing the design is to create a PROM (Programmable Read Only Memory). The PROM contains all of the logic necessary to implement the SLC1657 microcontroller. Follow the directions for the Xilinx Alliance Series software tool to create the PROM, and program it with your PROM programming system.

The PROM file that we created is formatted as an Intel Hex device, and has a filename of 'XSP2EVAL.mcs' under the examples directory.

The SLC1657 Evaluation Board for Xilinx Spartan 2 uses a Xilinx 1701LPC PROM. Program the PROM and insert it into socket U5.

- IMPORTANT –
The Xilinx 1701LPC PROM can be configured for active low or active high reset. The default on most PROM programmers is active high. However, the evaluation board requires that the PROM be configured for an active low reset. If you fail to do this, then the board will not boot up.

6.3 Using the Emulation ROM (Download) Capability

The steps listed in section 6.2 are used to create a complete SLC1657 system on the Xilinx Spartan 2 evaluation board. That system was programmed onto a PROM, and contains the hardware for the microcontroller. The circuit contains an emulation ROM capability. This allows software instructions to be downloaded into the board over a parallel port cable.

In this example, a sample software program is downloaded over the parallel port cable. To demonstrate its use, a calculator demonstration program called 'CALCDEMO.C' is used. This turns the evaluation board into a four function calculator.

Before downloading, inspect the program called 'CALCDEMO.C'. As you will see, it contains standard 'C' source code. This program is compiled using the 'CC5X' compiler available from B. Knudsen Data (Trondheim, Norway). The compiler produces a file

called ‘CALCDEMO.HEX’, which is the Intel Hex formatted file. Both the ‘C’ source file and the compiled file are provided in the EXAMPLES directory.

Software is downloaded with a program called ‘DOWNLOAD.EXE’. This is an executable file for use under the DOS operating system. DOWNLOAD.EXE reads the Intel Hex formatted file and sends it out the parallel port cable.

Follow these simple instructions to operate the emulation ROM.

1) Remove the evaluation board from the anti-static bag[24].

2) Verify that an 8-pin PROM is loaded into the socket located at ‘U5’. All of the PROMs supplied with the SLC1657 demo board include the emulation ROM capability. Also, there is a ‘spare’ PROM socket located at U2. This socket is not active, and only serves as a holder for an unused PROM.

3) Connect the parallel port download cable to the printed circuit board at connector J2. Connect the other end of the cable to the parallel port connector on a PC computer. This cable is a standard Centronics compatible parallel port cable.

4) Connect the +9 VDC battery pack to the evaluation board.

5) If you are using the PROM created above, then the display will show eight ‘blanks’ on the left hand side of the display. At this point the microcontroller has booted up, but its emulation ROM is empty.

6) On the PC computer, get into DOS mode (if running Windows 95/98). Locate the directory with the program called DOWNLOAD.EXE. Type the following at the DOS command prompt (using the correct path):

   download lpt1 c:\slc1657\xilinx\examples\calcdemo\calcdemo.hex

   This causes the object file called ‘calcdemo.hex’ to be downloaded over the parallel port cable. Once the download is complete, the core will automatically reset and run the program.

   In the command line syntax, ‘lpt1’ refers to the parallel port number. If ‘lpt2’ is used (or some other port), substitute the port number.

   If you have the ‘CC5X’ compiler, then you can edit ‘calcdemo.c’ and compile it. The compiler creates the Intel Hex formatted file called ‘calcdemo.hex’, which can be immediately downloaded to the evaluation board.

[24] The board should be handled at an approved anti-static workstation.

- IMPORTANT –
DOWNLOAD.EXE is intended to be operated from a DOS environment, including the variants under Windows 95 and 98. However, it will not work with Windows NT. Microsoft has implemented security walls on Windows NT to prevent access to the parallel port.

7) Verify that the core boots up, and that display on the evaluation board reads '0'. This indicates that the microcontroller inside of the FPGA has reset and is running the application code.

8) Try the calculator.

6.4 Creating an Embedded PROM

This section describes how to create an embedded ROM. The embedded PROM contains information for both hardware and software.

The PROM created in the example of section 6.1 (above) causes the SLC1657 to boot up without any instruction memory. Under that scenario, software is downloaded and tested over the parallel port cable. However, once the user is satisfied with the code, then it can be embedded into the PROM. This section describes how to create the same ROM, but instead with embedded software attached.

For this example, we'll use the same 'CALCDEMO.HEX' file to create the embedded ROM. However, in this case the 'CALCDEMO.HEX' file will be converted to a Xilinx '.COE' file. The Xilinx '.COE' file is used to initialize the instruction ROM (INSTRROM).

To create the Xilinx '.COE' file, perform the following operations:

1) Move the file 'calcdemo.hex' into the directory called 'MAKEXCOE'.

2) Convert the file by typing: MAKEXCOE CALCDEMO.HEX.

3) The conversion utility will create a file called CALCDEMO.COE. This file will not be used to initialize the ROM.

When creating the embedded ROM, follow all of the same steps as shown in section 6.1 However, substitute the following directions for those given in STEP 3 (creating INSTRROM). The modified instructions are:

Using the Xilinx CORE Generator tool, create the INSTRROM entity:

1) Create a directory called INSTRROM_CALCDEMO. [This step has already been performed for you in the EXAMPLES directory.

2) Open the Xilinx CORE Generator tool, and set it up to create a synchronous RAM in the INSTRROM folder. Set up the options thusly:

   Device type: Spartan 2
   File format: VHDL
   Tool type: Other (Protel)
   Netlist bus format: B(I)
   Memory type: single port block memory
   Component name: instrrom
   Depth: 2048
   Data width: 12
   Port configuration: read and write
   Load init value: check box
   Load file: enter the path for the calcdemo.coe file created above

3) Generate INSTRROM.

4) Verify that the INSTRROM_CALCDEMO folder that you created has a file named 'instrrom.edn' in it. This is the EDIF file for the instruction ROM (that we'll use later).

5) Repeat the rest of the steps for creating the 'XSP2EVAL' above. For your convienience, these steps have already been done for you in the Examples directory under 'XSP2EVAL_CALCDEMO'.

6.5 VHDL Entity Reference for XILINX SPARTAN 2

The VHDL entities used in the Xilinx Spartan 2 Evaluation project are given below. These are specific to this implementation. However, the TOPLOGIC entities (given in Chapter 5) are also used in the example.

6.5.1 LPFILTER Entity

Other entities used by this module: NONE

The LPFILTER entity is a digital low-pass filter. Each of the EMROMINT programming inputs is conditioned by LPFILTER. This prevents noise from the PC-compatible download cable from entering the core. Figure 6-3 shows how the filter works.

The filter input is synchronized to the filter clock [MCLK_16] by a D type flip-flop. This prevents metastable and race conditions from occurring within the filter itself. Once the input is synchronized, it enters the LPFILTER state machine. The state machine is designed so that the input signal must be in its asserted or negated state for at least two [MCLK_16] cycles. This causes short (high frequency) pulses to be rejected, and long (low frequency) signals to be accepted.

Figure 6-3 also shows the filter response. Very low frequencies are passed without attenuation. As the speed of the input signal increases to MCLK_16 / 3, the filter begins to reject the input signal. Signals faster than MCLK_16 are rejected[25].

For example, when the SLC1657 clock [MCLK] operates at 5.00 MHz, the filter passes all frequencies up to about 0.104 MHz. As the input signal increases beyond that point, the low-pass filter begins rejecting the input. Signals faster than 0.313 MHz are totally rejected.

[25] If the input signal frequency exceeds MCLK_16 x 2, then the output of the filter will start to pass some signal. However, the noise found on the parallel cable does not exhibit this behavior and is not a problem.

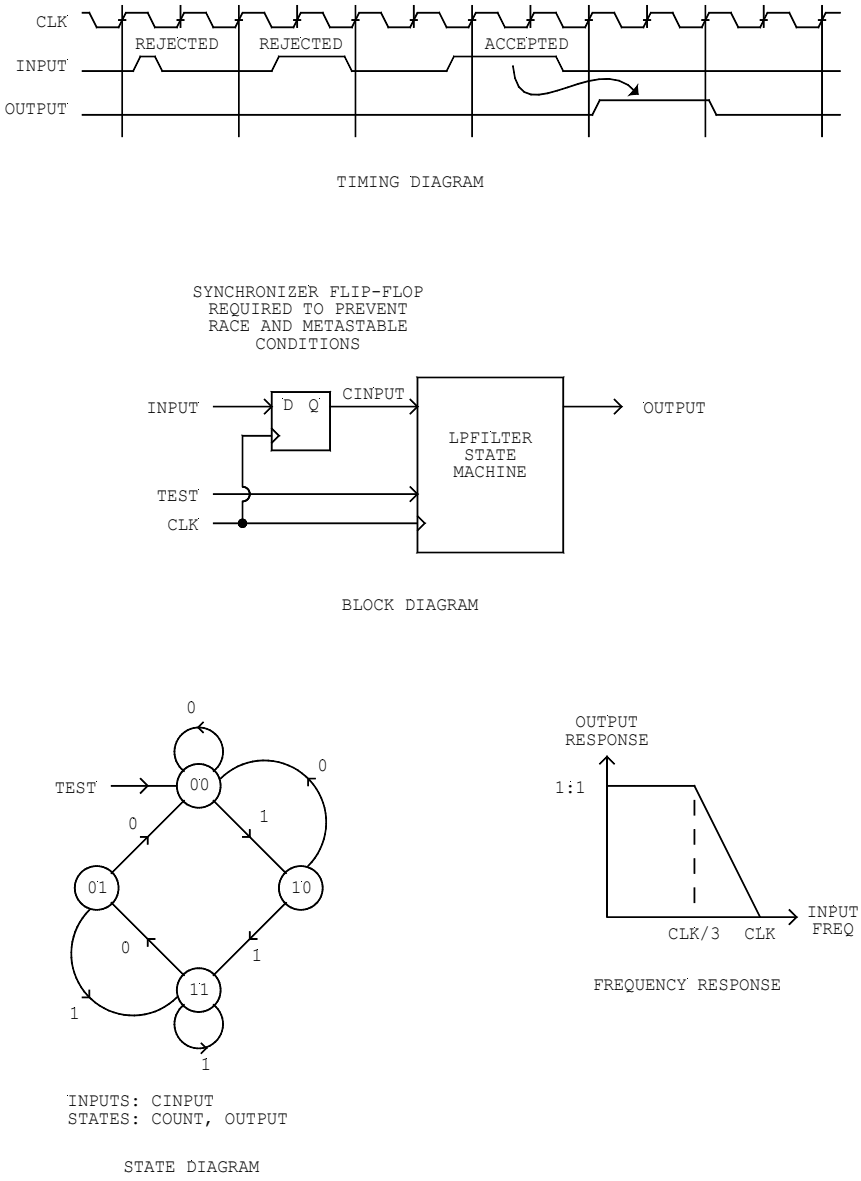


Figure 6-3. LPFILTER entity operation.

6.5.2 MUX11X02 Entity

Other entities used by this module: NONE

The MUX11X02[26] entity multiplexes two, 11-bit buses.

6.5.3 SEMRMINT Entity

Other entities used by this module: LPFILTER, MUX11X02

The SEMRMINT (Serial Emulation RoM INTerface) entity provides an external interface for 2,048 x 12 ROM emulation. It allows programming through four external pins, and is intended for FPGA devices.

The entity also provides signal conditioning for the Xilinx block memory. This memory uses a clocking scheme that is not directly compatible with the SLC1657 ROM interface. The SEMRMINT entity provides a compatible, synchronous interface between the two.

Figure 6-4 shows a block diagram of the SEMRMINT entity. During normal operation the external [PROG*] input is negated. This negates the internal [PRESET] signal, and allows the core to run normally. Addresses from the program counter are routed to the RAM address lines through MUX11X02. The RAM then generates instructions which appear at its [ADR(10..0)] output.

Instructions can be downloaded to the core by connecting a programming cable to the programming enable [PROG*], programming clock [PCLK*], programming data [PDAT*], and programming latch [PLCH*] pins. From a PC-compatible computer this can be done via a Centronics parallel port cable in conjunction with the download software.

Figure 6-5 shows the instruction download timing. The download begins when the [PROG*] signal is asserted. This has the effect of (a) resetting the microcontroller and (b) changing the source of the address bus from the programming counter to the SEMRMINT download circuit.

Once [PROG*] is asserted, the download data is presented to the [PDAT*] input. This is then clocked into the SEMRMINT shift register using the [PCLK*] pin. Address and data information is then clocked into the core using the protocol shown in Figure 6-5.

[26] MUXWWXSS specify a class of multiplexors where 'WW' is the width of input and output buses and 'SS' specifies the number of selectors.

All of the inputs are conditioned by a low pass filter (LPFILTER entity). This prevents spurious noise (which is common on PC parallel port cables) from corrupting incoming data.

When a complete address and data pair is loaded into the shift register, it is latched into the programming RAM using the [PLCH*] signal. A state machine conditions the write pulse, thereby making it compatible with the Xilinx block memory. At this time the sequence can be repeated until all or part of the 2,048 x 12 RAM has been loaded. Once loaded, the [PROG*] input is negated, and the core starts up normally (using the new program).

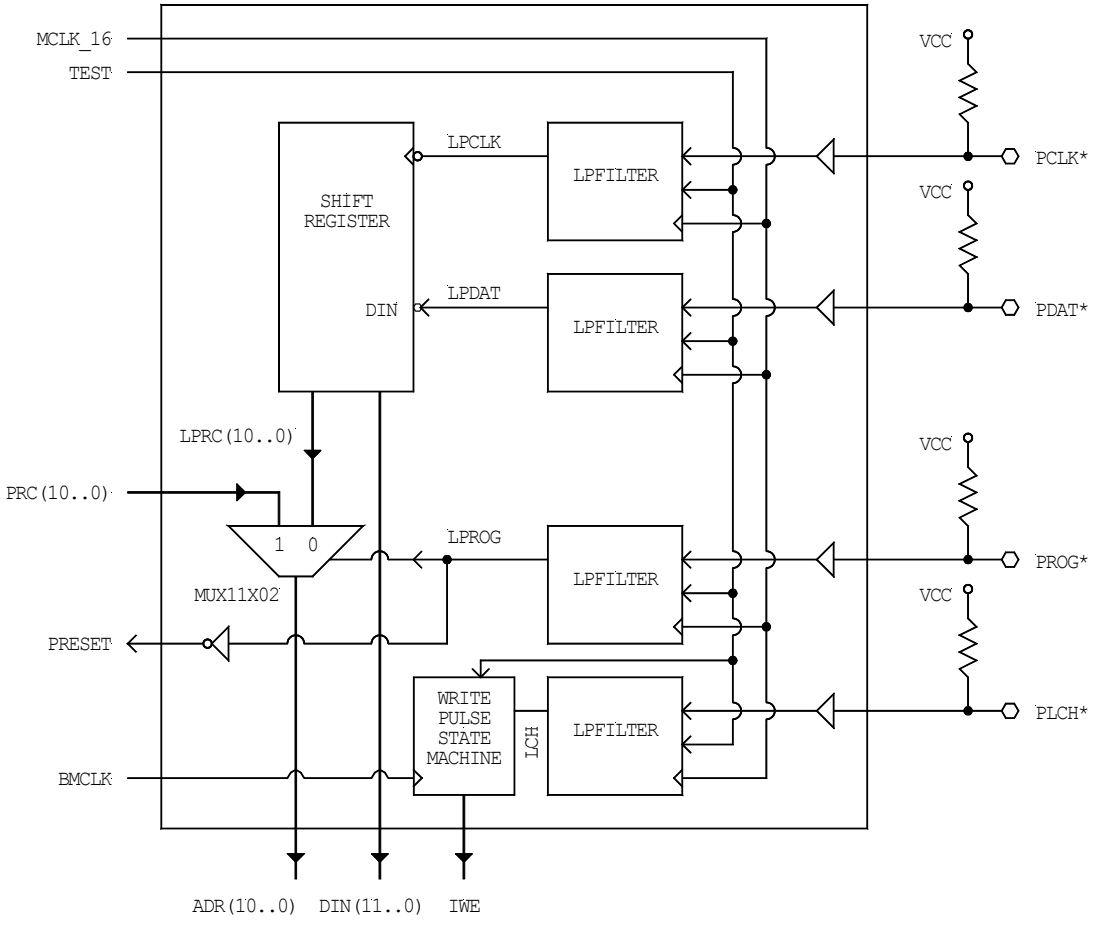


Figure 6-4. SEMRMINT block diagram.

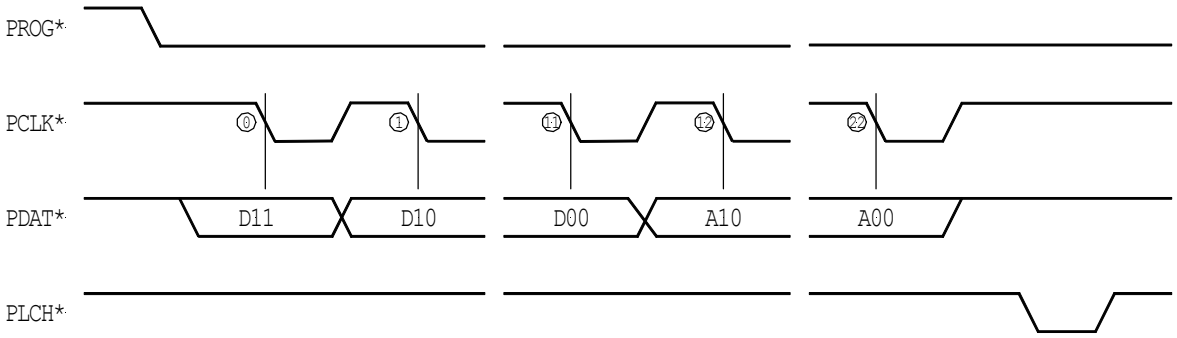


Figure 6-5. SEMRMINT instruction download.

The entity also converts the SLC1657 instruction ROM cycles into a cycle that is compatible with the Xilinx block memory. This conversion is shown in the timing diagram of Figure 6-6.

The left side of the figure shows the instruction fetch cycles. The TOPLOGIC core generates an instruction address after every rising edge of [MCLK]. This clock is inverted to create [N_MCLK] so that the Xilinx block RAM latches the address near to the *falling* edge of [MCLK]. Once latched, the RAM accesses the instruction, and sends it to its data output port. The data must then make its way back to the TOPLOGIC core by the next rising edge of [MCLK]. This makes the Xilinx block memory compatible to the FASM asynchronous ROM cycle used by the SLC1657.

The right side of the figure shows a typical instruction download cycle. After the host computer downloads an instruction address/data pair, it asserts the [PLCH*] signal. This causes the write pulse state machine (located in the SEMRMINT entity) to generate a single, synchronous write pulse. Although the state machine is synchronous with [MCLK], the internal timing insures that the write pulse will be valid during the rising edge on the Xilinx block memory. Furthermore, the state machine allows only one write pulse (lasting for one clock cycle) to be generated, regardless of the duration of the external [PLCH*] signal. The state diagram for the write pulse is shown in Figure 6-7.

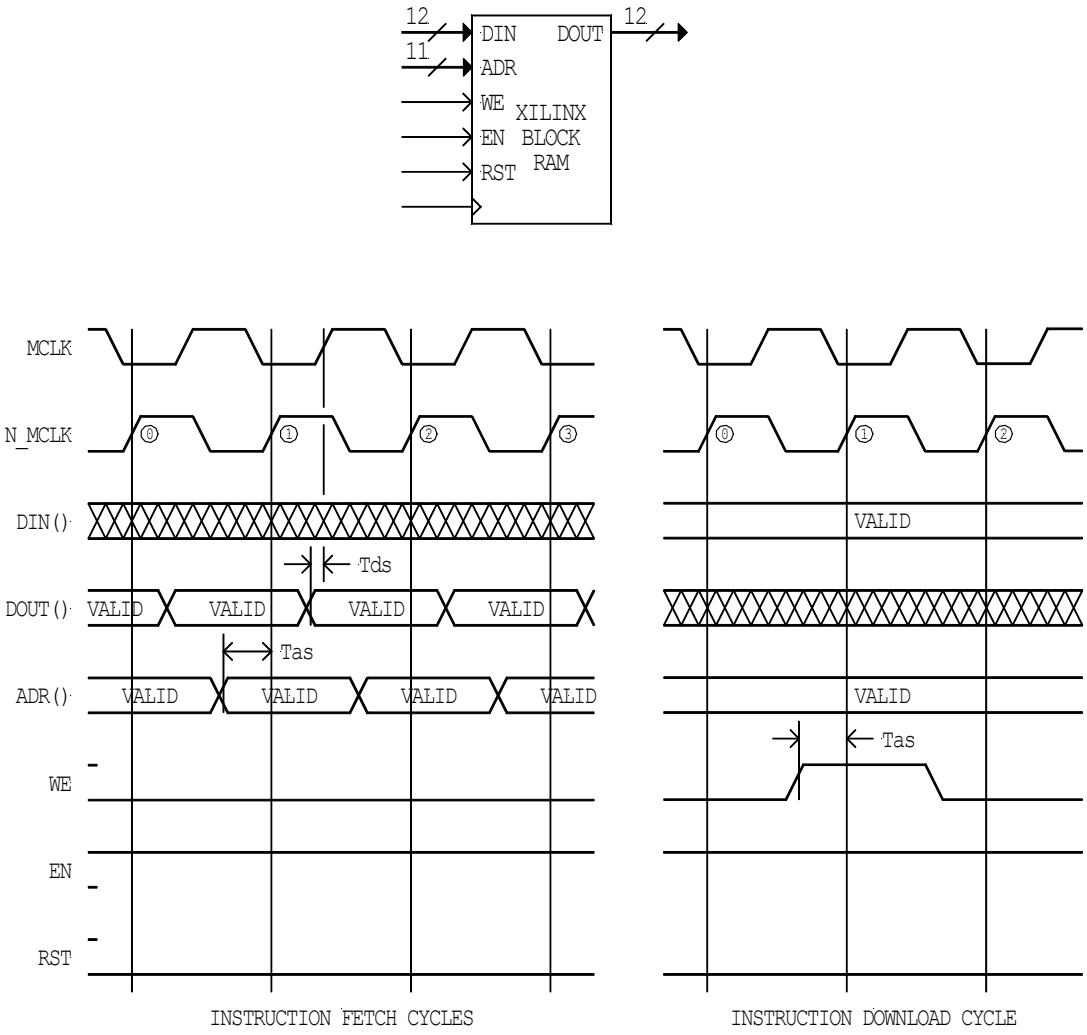


Figure 6-6. Xilinx block RAM cycles.

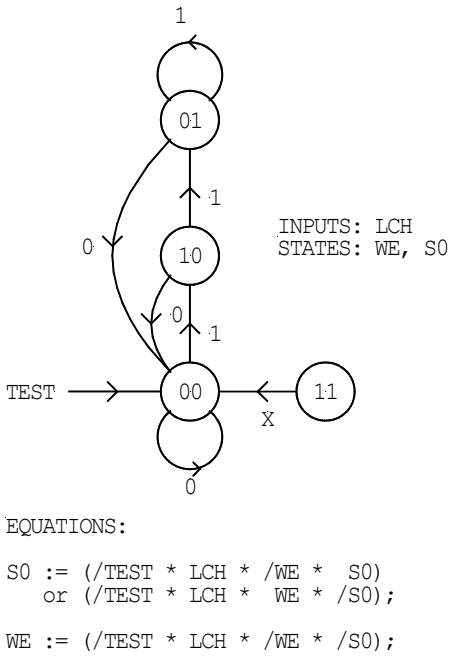


Figure 6-7. State diagram for the write pulse state machine.

6.5.4 XSP2EVAL Entity

Other entities used by this module: SEMRMINT, TOPLOGIC

The XSP2EVAL entity is the highest level entity used in the Xilinx Spartan 2 evaluation project. A block diagram of the entity is shown in Figure 6-8. The heirarchy diagram is shown in Figure 6-9.

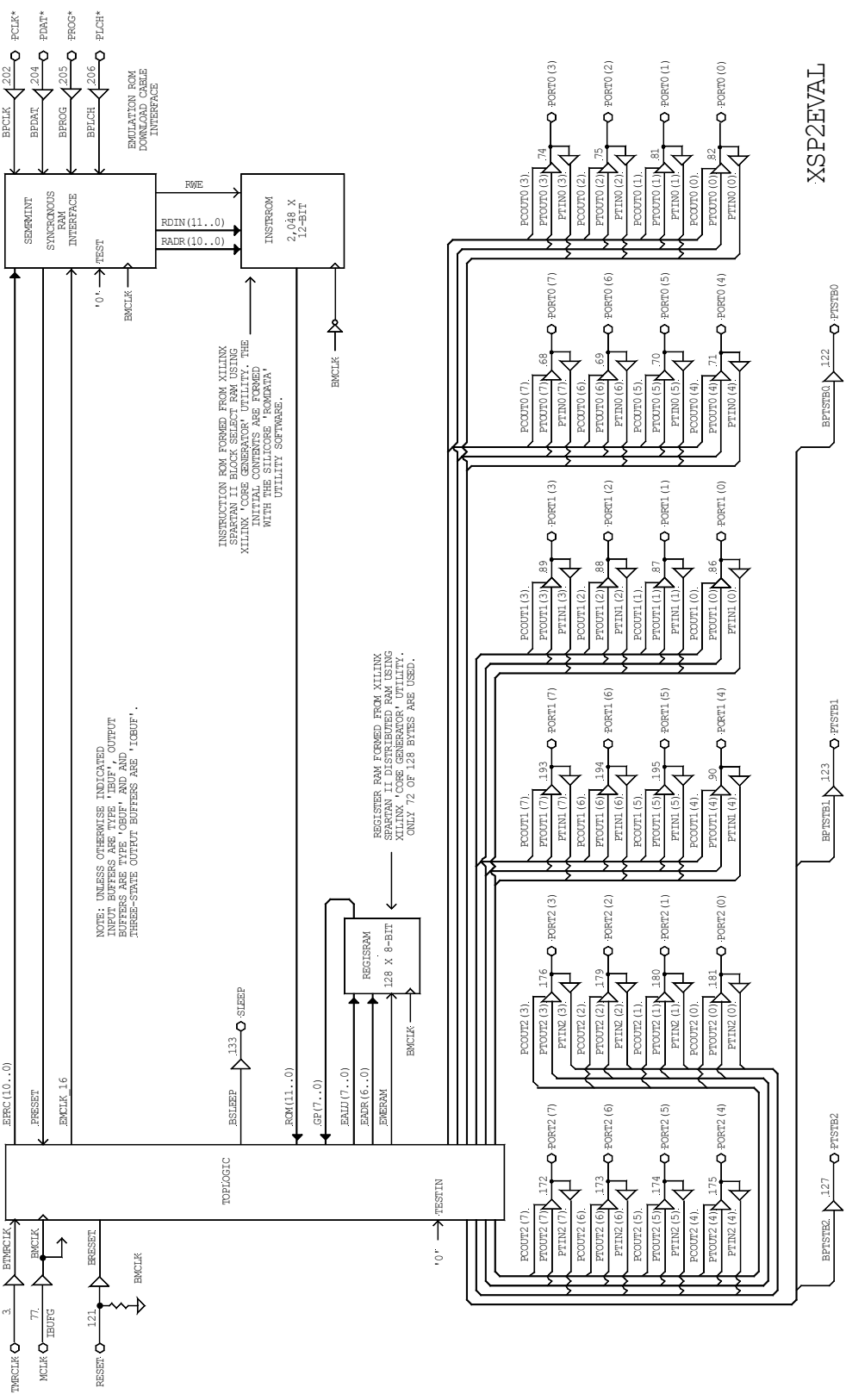


Figure 6-8. Block diagram of the XSP2EVAL entity.

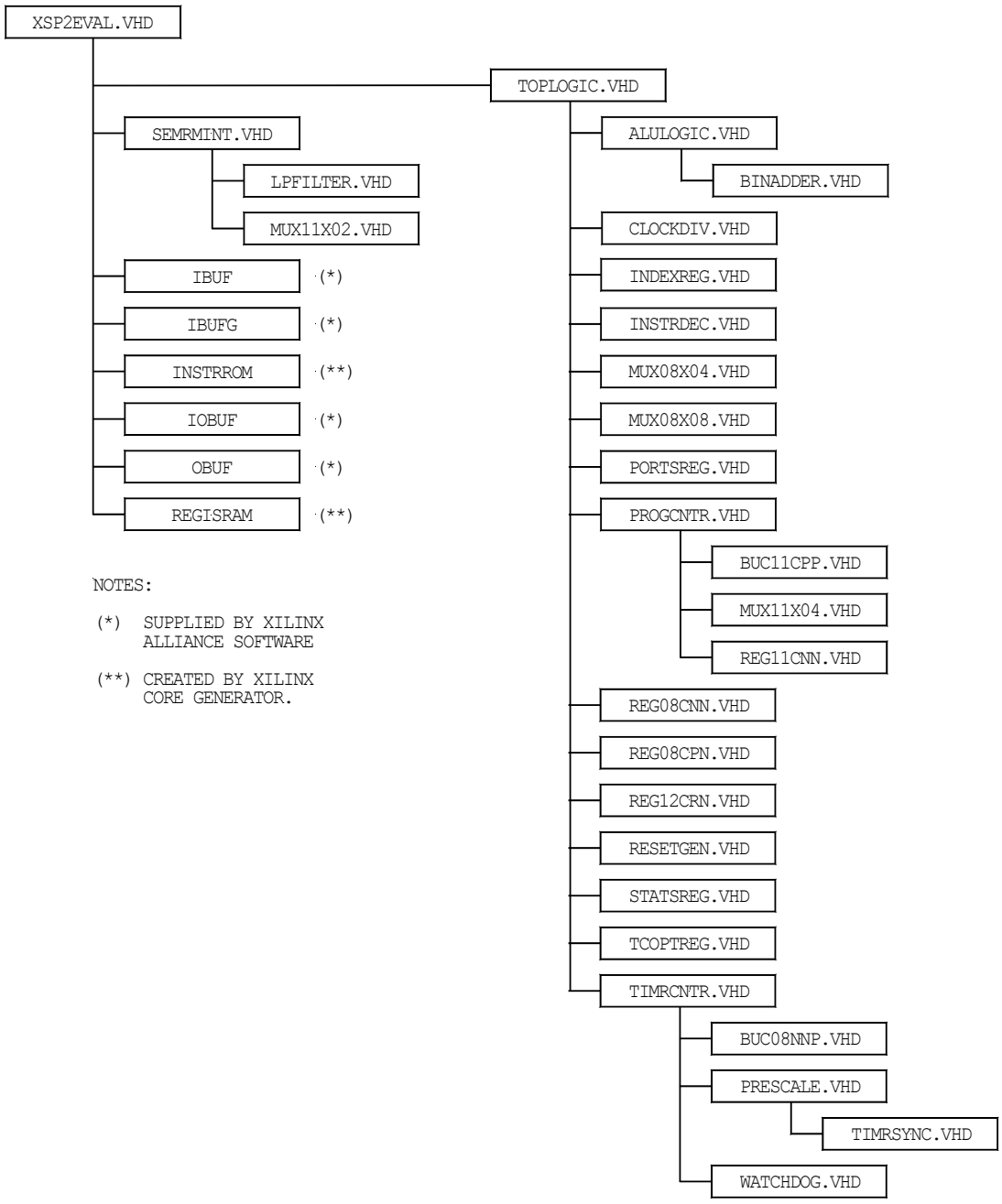


Figure 6-9. Hierarchy diagram for the XSP2EVAL entity.

7.0 Implementation on the Altera FLEX 10KE FPGA

This chapter describes the steps needed to integrate the SLC1657 onto a Altera FLEX 10KE FPGA. An exercise is presented whereby a four function calculator is implemented on an evaluation board.

The purpose of this chapter is to:

- Learn about the SLC1657 Evaluation Kit for the Altera FLEX 10KE FPGA.
- Learn the steps needed to integrate a simple system-on-chip.
- Demonstrate how to simulate the TOPLOGIC entity.
- Demonstrate how to synthesize an IP core.
- Create the register RAM and instruction ROM.
- Create a parallel port interface for download and test of application code.
- Integrate the TOPLOGIC core with RAM, ROM and parallel port interface.
- Download and run a 'C' application program for a 10-key calculator.
- Create a fixed PROM.

The following hardware and software tools are used in the exercises:

- PeakVHDL simulation and synthesis tools from Protel International.
- Altera MAX+PLUS II Place & Route Software.
- SLC1657 evaluation kit for Altera FLEX 10KE FPGA.
- CC5X 'C' compiler from B Knudsen Data.
- DOWNLOAD software for testing application code.
- MAKEXCOE software for integration of a ROMable application software.
- PROM programmer[27].

[27] The Data I/O Plus 48 PROM programmer was used for this exercise.

7.1 Evaluation Kit for Altera FLEX 10KE FPGA

The evaluation kit for Altera FLEX 10KE FPGA allows the user to evaluate and test the SLC1657 microcontroller. The kit includes:

- Evaluation board with Altera FLEX EPF10K50E FPGA (see Figure 7-1).
- PROMs for demonstration and calculator functions.
- 16 x 1 LCD display.
- 20-key keypad.
- 5-MHz crystal oscillator.
- 1 KHz RC oscillator.
- 9V battery pack.
- Demonstration program.
- Calculator program.
- PC parallel port download cable and software.
- Technical reference manual.

The evaluation board comes with two embedded software programs. These are 'ADMO', a generic demonstration PROM and 'ACLC', a calculator program. Each resides on a PROM, which contains both the hardware for the SLC1657 microcontroller and the software application programs.

7.1.1 ADMO Software

The ADMO embedded ROM program demonstrates how the SLC1657 can be completely integrated into an FPGA. This includes RAM, ROM and I/O elements. The ADMO embedded ROM demonstration displays the features of the core, and also has a 'stopwatch' function. Follow these simple instructions to operate ADMO:

1) Remove the evaluation board from the anti-static bag[28].

2) Verify that the 8-pin ROM labeled 'ADMO' is located in DIP socket U5 (to the right of the LCD display). There is a 'spare' PROM socket located at U2 (at the top of the board). This socket is not active, and only serves as a holder for the unused PROM. You might need to switch the PROMs around.

3) Connect the +9 VDC battery pack to the evaluation board using connector J1.

[28] The board should be handled at an approved anti-static workstation.

4) Verify that the core boots up, and that display on the evaluation board reads 'SILICORE SLC1657'. This indicates that the microcontroller inside of the FPGA has reset and is running the application code.

5) Push switch 'S17' (the switch marked '0').

6) The features of the core scroll by on the display.

7) Push switch 'S18' (the switch marked '.').

8) Verify that a counter display "00:00 0/10th" appears. Pushing switch S18 ('.') always starts the 'stopwatch' application. Pushing switch S19 ('+/-') starts the stopwatch, and pushing 'S20' ('=') stops it. The stopwatch can be cleared by pushing 'S18' ('.') again. The following table summarizes the switches used by ADMO:

Table 7-1. ADMO Key Functions

Switch	Label	Action
S17	'0'	Marquee of features
S18	'.'	Initiate/clear stopwatch
S19	'+/-'	Start stopwatch
S20	'='	Stop stopwatch

7.1.2 ACLC Software

The ACLC calculator software places the evaluation board into its calculator mode. Install the ACLC PROM into U5 and operate the evaluation board as a four function calculator.

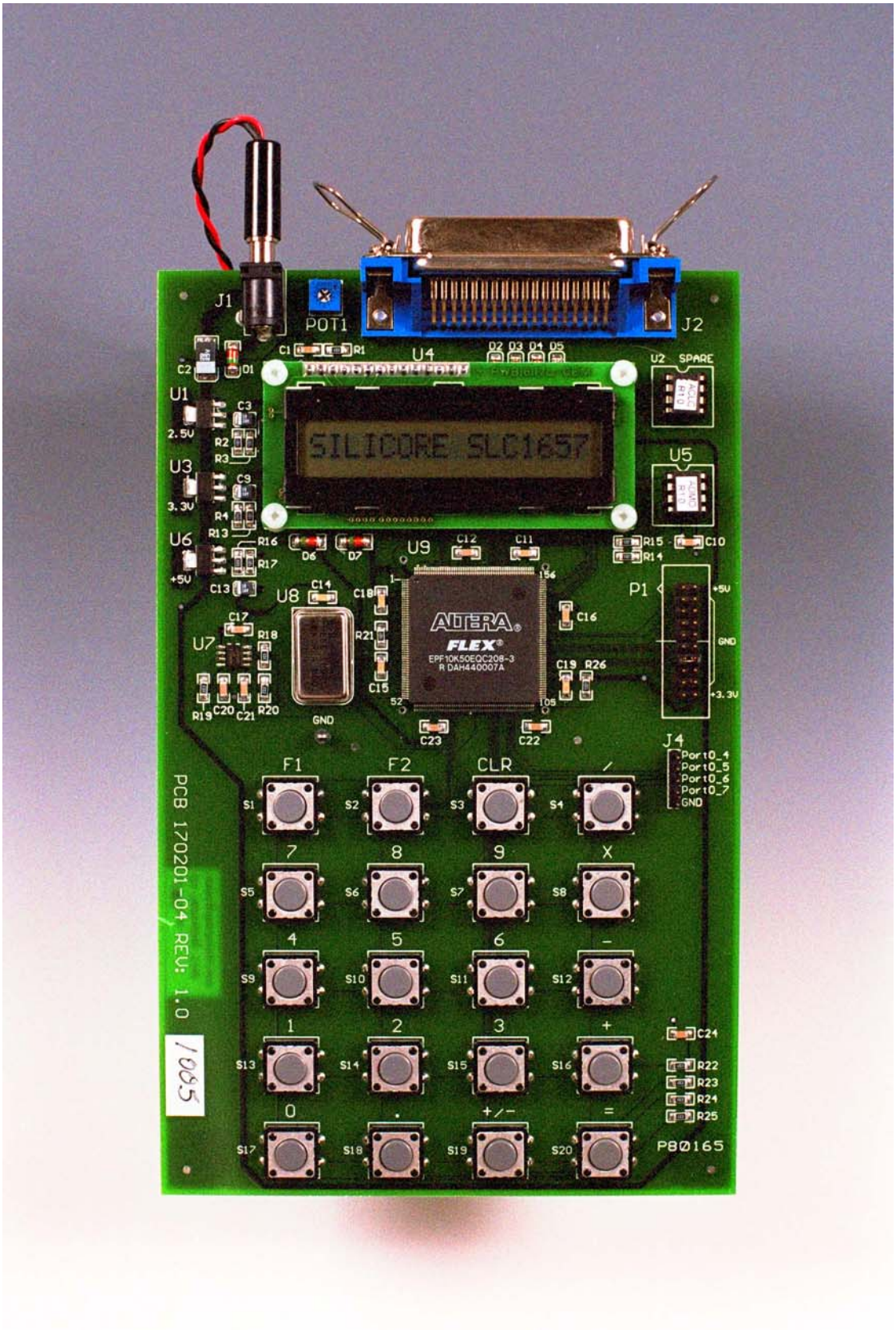


Figure 7-1. Evaluation board for Altera FLEX 10KE FPGA.

7.2 The AF10EVAL Exercise

An exercise is given below to better understand the operation of the SLC1657. This creates a system-on-chip called 'AF10EVAL', which stands for Altera FLEX 10KE EVALuation system. It's a system-on-chip (SoC) that we'll use to design and run a four function calculator.

The AF10EVAL system uses several VHDL entities. These are described in detail in section 7.5 (below). The user is encouraged to study the descriptions there, along with the VHDL source code. These entities include:

- AF10EVAL: Altera FLEX 10KE Evaluation (top level VHDL entity)
- TOPLOGIC: TOP LOGIC design for the SLC1657.
- REGISRAM: REGISter RAM.
- INSTRROM: INSTRuction ROM.
- AEMRMINT: Altera Emulation ROM Interface.

7.2.1 STEP 1 – Simulate the TOPLOGIC Entity

The first step to creating the SLC1657 is to simulate the TOPLOGIC entity. This familiarizes the user with the simulation tools, the SLC1657 IP core and the general operation of all components. This step is identical for all target devices such as Agere, Altera and Xilinx.

Using the Protel PeakVHDL simulation tool, perform the following operations:

1) Create a new directory called 'TLTEST'. [One has been created for you in the EXAMPLES folder if you wish to use it].

2) Open PeakVHDL and create a new project (following the manufacturers directions). Name the project TLTEST, and put it into the 'TLTEST' folder.

3) Add all of the modules in the TOPLOGIC entity into the project. Be sure to preserve the entity hierarchy. The hierarchy is described with the TOPLOGIC entity in Chapter 5. Each entity can be found in its own unique folder in the 'VHDL_source' directory.

   When simulating with the PeakVHDL product, be sure that the highest level module in the hierarchy is the TOPLOGIC test bench (TSTBENCH.VHD from the TOPLOGIC folder).

Also, the TOPLOGIC test bench simulation will need the corresponding test vector files. These are the files with the '*.txt' extension in the TOPLOGIC folder, and should be copied into the TLTEST directory.

When finished, the project window should look something like that shown in Figure 7-2.

4) Simulate the design using the manufacturers directions. At this point the TOP-LOGIC entity should simulate with no errors.

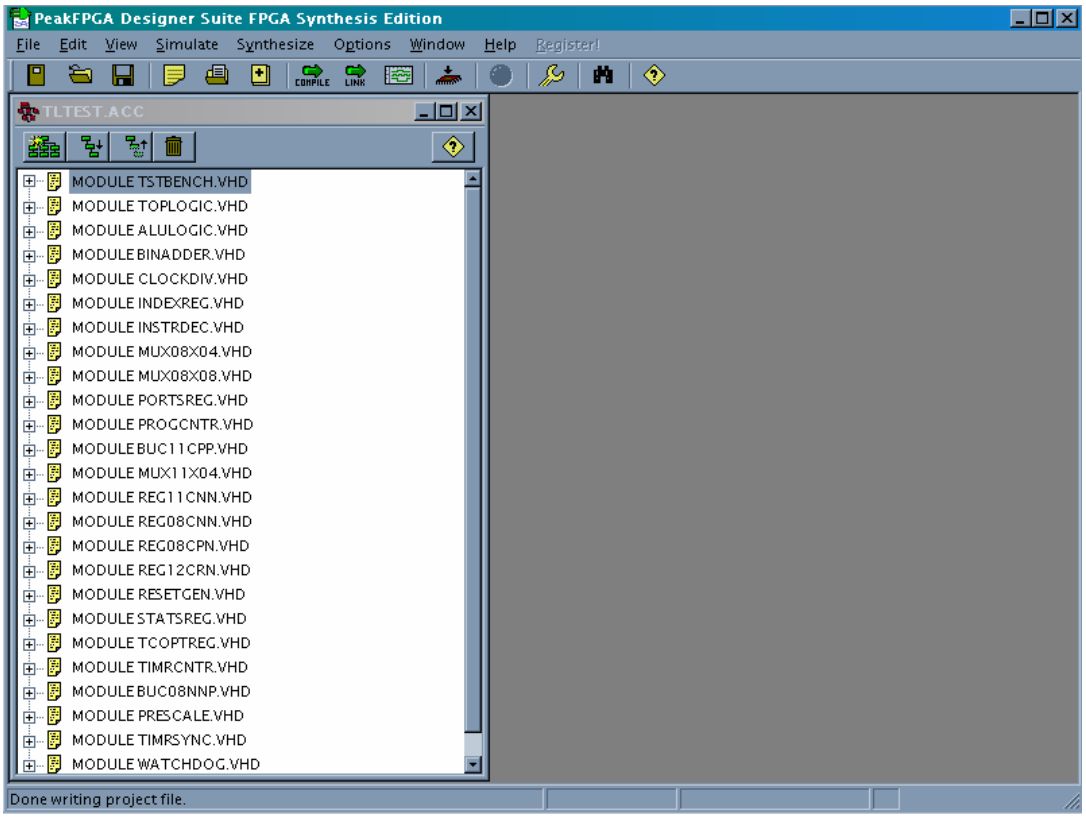


Figure 7-2. PeakVHDL project window.

7.2.2 STEP 2 – Create REGISRAM (Register RAM)

The register RAM is a 128 x 8-bit synchronous memory. It must conform to the FASM SYNCHRONOUS RAM guidelines described elsewhere in this manual. There are many ways to build memories, but the simplest is to use the automatic memory generation software that is supplied with most FPGA place & route tools.

Altera supplies such a tool with their MAX+PLUS II software. It's called the MegaWizard Plug-in Manager, and is capable of creating the exact memory that's needed.

In this example the register RAM will be formed from 'EAB's, or Embedded Array Blocks. Using the Altera MegaWizard Plug-in Manager tool, create the REGISRAM entity:

1) Create a directory called 'REGISRAM'. [One has been created for you in the ALTERA EXAMPLES folder].

2) Open the Altera MegaWizard Plug-in Manager tool, and set it up to create a synchronous RAM in the REGISRAM folder. Set up the options thusly:

   Output file format: VHDL
   Project file: create a project file named 'regisram' in the REGISRAM directory.
   Megafunction type: STORAGE: LPM_RAM_DP
   Data bus width: 8-bit
   Address bus width: 7-bit
   Clocking method: single clock
   Registered port(s): write input (only)
   Memory initialization: none (unchecked)
   Implementation: with EAB's (i.e. box unchecked)

3) Generate REGISRAM.

4) Verify that the REGISRAM folder that you created has a file named 'REGISRAM.vhd' in it. This is a VHDL description for the RAM (that we'll use later).

   If you inspect the REGISRAM.VHD file that was created you will note that there are separate read and write address buses. This is what Altera calls a 'dual port' memory. However, these two address buses will be combined by the AF10EVAL entity. This will create a FASM compatible REGISRAM.

7.2.3 STEP 3 – Create INSTRROM (Instruction ROM)

In the Altera FLEX 10KE, the instruction ROM is formed from asynchronous Embedded Array Blocks (EABs). It must conform to the FASM ASYNCHRONOUS ROM guidelines described elsewhere in this manual. This will be configured by the Altera MegaWizard Plug-in Manager tool (used above) to form a 2,048 x 12-bit instruction memory.

Although the term 'instruction ROM' is used here, this memory is actually a read/write memory. That's because the example will create a downloadable memory interface. This allows application software to be downloaded into the instruction ROM. This is ac-

complished with a parallel port interface called AEMRMINT, and is very useful for software development purposes.

For now, we'll rely on the download capability to get new application code into the microcontroller. However, later we'll initialize the INSTRROM entity with our application code. Using the Altera MegaWizard Plug-in Manager tool, create the INSTRROM entity:

1) Create a directory called 'INSTRROM'. [One has been created for you in the Altera examples folder if you wish to use that].

2) Open the Altera MegaWizard Plug-in Manager tool, and set it up to create an asynchronous RAM in the INSTRROM folder. Set up the options thusly:

   File format: VHDL
   Megafunction type: LPM_RAM_DQ
   Data bus width: 12-bit
   Address bus width: 11-bit
   Registered port: non checked (this is an asynchronous FASM memory)
   Memory initialization: none (unchecked)
   Implementation: with EAB's

3) Generate INSTRROM.

4) Verify that the REGISRAM folder that you created has a file named 'INSTROM.VHD' in it. This is a VHDL description for the ROM (that we'll use later).

7.2.4 STEP 4 – Synthesis

The highest level entity/architecture pair in this system is the VHDL source file named 'AF10EVAL'. This file ties all of the parts of the system together as described in the block and hierarchy diagrams for the AF10EVAL entity below.

Using the Protel PeakVHDL synthesis tool, perform the following operations:

1) Create a new directory called 'AF10EVAL'.

2) Open PeakVHDL and create a new project (following the manufacturers directions). Name the project AF10EVAL, and put it into the 'AF10EVAL' folder. [This is already done for you in the Altera examples folder if you wish to use that.]

3) Add all of the modules in the AF10EVAL entity into the project. Be sure to preserve the entity hierarchy. The hierarchy is described with the AF10EVAL entity

later in this chapter. The entities relating to TOPLOGIC (e.g. ALULOGIC.VHD) can be found in its own unique folder in the 'VHDL_source' directory. The entities relating to Altera FLEX 10KE implementation (e.g. AEMRMINT.VHD) can be found in 'Altera' directory.

4) Move the following files into the AF10EVAL directory: 'REGISRAM.vhd' and 'INSTRROM.vhd'. These were created earlier, and are contained in the REGIS-RAM and INSTRROM directories (respectively). The Altera Max+Plus II place & route software will refer back to these files when routing the design. It will expect them to be in the 'AF10EVAL' directory.

5) Select 'FLEX 10KE Series (EDIF)' in the PeakVHDL synthesis options.

6) Synthesize the AF10EVAL system with PeakVHDL.

7) Look in the synthesis log file, and verify that no errors were generated by PeakVHDL.

8) Verify that file 'AF10EVAL.EDN' is present in the directory. This is the EDIF file created by PeakVHDL.

7.2.5 STEP 5 – Place & Route the Design

The EDIF file created in STEP 4 contains all of the microcontroller logic. The next step is to place and route the design on the Altera FLEX 10KE FPGA chip. In this example, we'll use the Altera Max+Plus II software to place and route the design.

Using the Altera Max+Plus II software tool, perform the following operations:

1) Start the Max+Plus II design manager.

2) Create a new project. Under FILE|PROJECT|NAME select 'AF10EVAL.edn' as the input file. This was the EDIF file that was created in STEP 4, and is the input file for the Max+Plus II place and route software. [This has already been done for you in the Altera EXAMPLES folder].

3) Under ASSIGN|DEVICE, verify (or enter) the part number of the FLEX 10KE FPGA.

4) Under ASSIGN|PIN LOCATION assign the pin locations on the FPGA chip. These are identical to those shown in Figure 7-8, and on the schematic diagram for the evaluation board.

5) Under ASSIGN|TIMING REQUIREMENTS generate a timing constraint for signal [MCLK] of 5.000 MHz.

6) Under ASSIGN|GLOBAL PROJECT DEVICE OPTIONS select the following:

   - EPC1PC8 PROM device
   - Check 'ENABLE INIT_DONE'
   - Check 'MULTIVOLT I/O'
   - Check 'USE LOW VOLTAGE CONFIGURATION DEVICE'

7) Place and route the design by selecting MAXPLUSII|COMPILER. After compiling, look in the 'AF10EVAL.rpt' file. This file reports the specific details of the place and route process, pin locations and so forth. Verify that there were no errors generated during the run.

7.2.6 STEP 6 – Create the PROM

The final step in implementing the design is to create a PROM (Programmable Read Only Memory). The PROM contains all of the logic necessary to implement the SLC1657 microcontroller. The file used to create the PROM is called the 'AF10EVAL.pof'. Use this file to create a PROM.

- IMPORTANT –
The Altera EPC1 PROM can be configured for +5 VDC or +3.3 VDC operation. The PROM is configured for the correct voltage (+3.3 VDC) by the MAX+PLUS II Software. The voltage was selected above by configuring the software for a 'LOW VOLTAGE CONFIGURATION DEVICE'. Failure to select this option may result in unreliable operation.

7.3 Using the Emulation ROM (Download) Capability

The steps listed in section 7.2 are used to create a complete SLC1657 system on the Altera FLEX 10KE evaluation board. That system was programmed onto a PROM, and contains the hardware for the microcontroller. The circuit contains an emulation ROM capability. This allows software instructions to be downloaded into the board over a parallel port cable.

In this example, a sample software program is downloaded over the parallel port cable. To demonstrate its use, a calculator demonstration program called 'CALCDEMO.C' is used. This turns the evaluation board into a four function calculator.

Before downloading, inspect the program called 'CALCDEMO.C'. As you will see, it contains standard 'C' source code. This program is compiled using the 'CC5X' compiler available from B. Knudsen Data (Trondheim, Norway). The compiler produces a file called 'CALCDEMO.HEX', which is the Intel Hex formatted file. Both the 'C' source file and the compiled file are provided in the EXAMPLES directory.

Software is downloaded with a program called 'DOWNLOAD.EXE'. This is an executable file for use under the DOS operating system. DOWNLOAD.EXE reads the Intel Hex formatted file and sends it out the parallel port cable.

Follow these simple instructions to operate the emulation ROM.

1) Remove the evaluation board from the anti-static bag[29].

2) Verify that an 8-pin PROM is loaded into the socket located at 'U5'. All of the PROMs supplied with the SLC1657 demo board include the emulation ROM capability. Also, there is a 'spare' PROM socket located at U2. This socket is not active, and only serves as a holder for an unused PROM.

3) Connect the parallel port download cable to the printed circuit board at connector J2. Connect the other end of the cable to the parallel port connector on a PC computer. This cable is a standard Centronics compatible parallel port cable.

4) Connect the +9 VDC battery pack to the evaluation board.

5) If you are using the PROM created above, then the display will show eight 'blanks' on the left hand side of the display. At this point the microcontroller has booted up, but its emulation ROM is empty.

6) On the PC computer, get into DOS mode (if running Windows 95/98). Locate the directory with the program called DOWNLOAD.EXE. Type the following at the DOS command prompt (using the correct path):

   download lpt1 c:\slc1657\Altera\examples\calcdemo\calcdemo.hex

   This causes the object file called 'calcdemo.hex' to be downloaded over the parallel port cable. Once the download is complete, the core will automatically reset and run the program.

   In the command line syntax, 'lpt1' refers to the parallel port number. If 'lpt2' is used (or some other port), substitute the port number.

[29] The board should be handled at an approved anti-static workstation.

If you have the 'CC5X' compiler, then you can edit 'calcdemo.c' and compile it. The compiler creates the Intel Hex formatted file called 'calcdemo.hex', which can be immediately downloaded to the evaluation board.

- IMPORTANT –
DOWNLOAD.EXE is intended to be operated from a DOS environment, including the variants under Windows 95 and 98. However, it will not work with Windows NT. Microsoft has implemented security walls on Windows NT to prevent access to the parallel port.

7) Verify that the core boots up, and that display on the evaluation board reads '0'. This indicates that the microcontroller inside of the FPGA has reset and is running the application code.

8) Try the calculator.

7.4 Creating an Embedded PROM

This section describes how to create an embedded ROM. The embedded PROM contains information for both hardware and software.

The PROM created in the example of section 7.2 (above) causes the SLC1657 to boot up without any instruction memory. Under that scenario, software is downloaded and tested over the parallel port cable. However, once the user is satisfied with the code, then it can be embedded into the PROM. This section describes how to create the same ROM, but instead with embedded software attached.

For this example, we'll use the same 'CALCDEMO.HEX' file to create the embedded ROM. However, in this case the 'CALCDEMO.HEX' file will be converted to a Altera '.MIF' file. The Altera '.MIF' file is used to initialize the instruction ROM (INSTRROM).

To create the Altera '.MIF' file, perform the following operations:

1) Move the file 'calcdemo.hex' into the directory called 'MAKEAMIF'.

2) Convert the file by typing: MAKEAMIF CALCDEMO.HEX.

3) The conversion utility will create a file called CALCDEMO.MIF. This file will be used to initialize the ROM.

When creating the embedded ROM, follow all of the same steps as shown in section 7.2 However, substitute the following directions for those given in STEP 3 (creating INSTRROM). The modified instructions are:

Using the Altera MegaWizard Plug-in Manager, create the INSTRROM entity:

1) Create a directory called INSTRROM_CALCDEMO. [This step has already been performed for you in the EXAMPLES directory.

2) Open the Altera MegaWizard Plug-in Manager, and set it up to create an asynchronous RAM in the INSTRROM folder. Configure everything the same as in section 7.2, except specify the memory initialization file of 'CALCDEMO.mif'.

3) Generate INSTRROM.

4) Move the 'INSTROM.vhd' file into the INSTRROM_CALCDEMO folder.

5) Repeat the rest of the steps for creating the 'AF10EVAL' above. For your convenience, these steps have already been done for you in the Examples directory under 'AF10EVAL_CALCDEMO'.

7.5 VHDL Entity Reference for ALTERA FLEX 10KE

The VHDL entities used in the Altera FLEX 10KE Evaluation project are given below. These are specific to this implementation. However, the TOPLOGIC entities (given in Chapter 5) are also used in the example.

7.5.1 LPFILTER Entity

Other entities used by this module: NONE

The LPFILTER entity is a digital low-pass filter. Each of the EMROMINT programming inputs is conditioned by LPFILTER. This prevents noise from the PC-compatible download cable from entering the core. Figure 7-3 shows how the filter works.

The filter input is synchronized to the filter clock [MCLK_16] by a D type flip-flop. This prevents metastable and race conditions from occurring within the filter itself. Once the input is synchronized, it enters the LPFILTER state machine. The state machine is designed so that the input signal must be in its asserted or negated state for at least two [MCLK_16] cycles. This causes short (high frequency) pulses to be rejected, and long (low frequency) signals to be accepted.

Figure 7-3 also shows the filter response. Very low frequencies are passed without attenuation. As the speed of the input signal increases to MCLK_16 / 3, the filter begins to reject the input signal. Signals faster than MCLK_16 are rejected[30].

For example, when the SLC1657 clock [MCLK] operates at 5.00 MHz, the filter passes all frequencies up to about 0.104 MHz. As the input signal increases beyond that point, the low-pass filter begins rejecting the input. Signals faster than 0.313 MHz are totally rejected.

[30] If the input signal frequency exceeds MCLK_16 x 2, then the output of the filter will start to pass some signal. However, the noise found on the parallel cable does not exhibit this behavior and is not a problem.

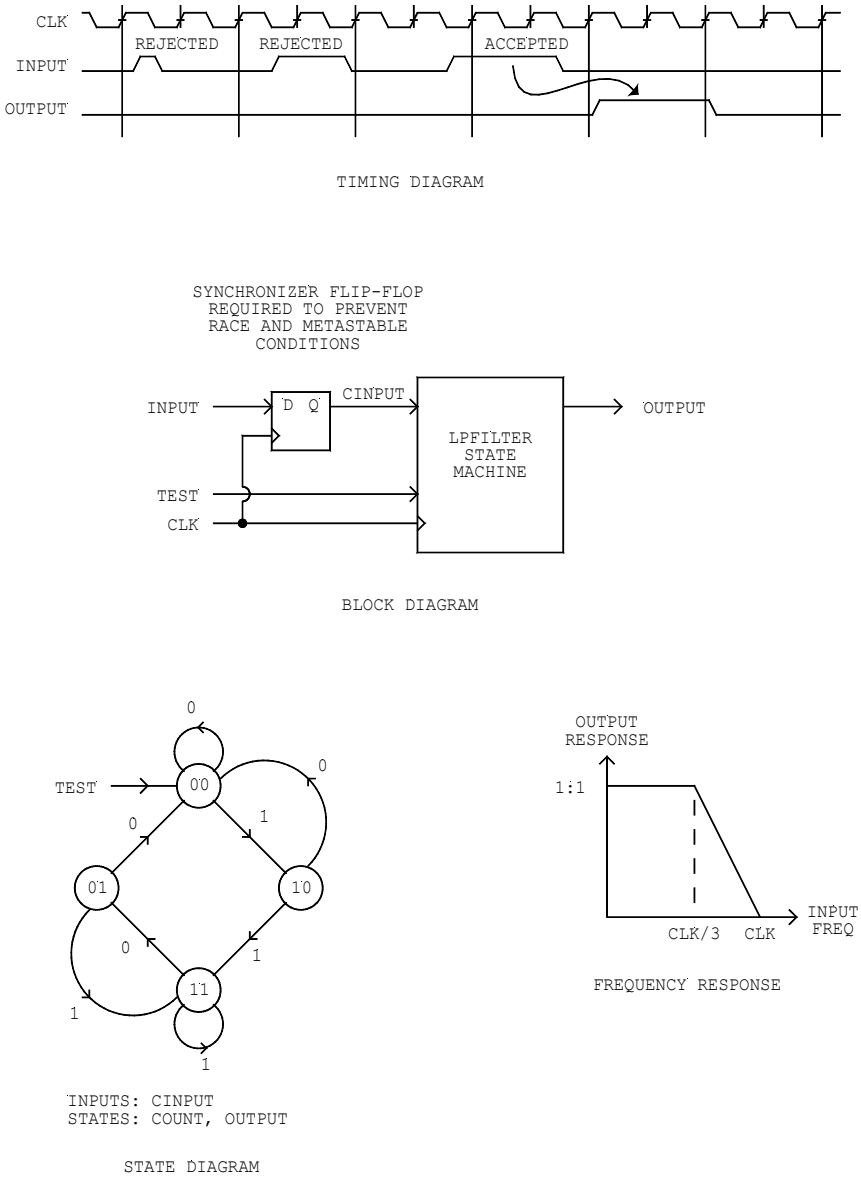


Figure 7-3. LPFILTER entity operation.

7.5.2 MUX11X02 Entity

Other entities used by this module: NONE

The MUX11X02[31] entity multiplexes two, 11-bit buses.

7.5.3 AEMRMINT Entity

Other entities used by this module: LPFILTER, MUX11X02

The AEMRMINT (Altera EMulation RoM INTerface) entity provides an external interface for 2,048 x 12 ROM emulation. It allows programming through four external pins. The entity also provides signal conditioning for the Altera block memory. The Altera block memory is assumed to be configured so that it's compatible with the FASM asynchronous ROM described elsewhere in this manual.

Figure 7-4 shows a block diagram of the AEMRMINT entity. During normal operation the external [PROG*] input is negated. This negates the internal [PRESET] signal, and allows the core to run normally. Addresses from the program counter are routed to the RAM address lines through MUX11X02. The RAM then generates instructions which appear at its [ADR(10..0)] output.

Instructions can be downloaded to the core by connecting a programming cable to the programming enable [PROG*], programming clock [PCLK*], programming data [PDAT*], and programming latch [PLCH*] pins. From a PC-compatible computer this can be done via a Centronics parallel port cable in conjunction with the download software.

Figure 7-5 shows the instruction download timing. The download begins when the [PROG*] signal is asserted. This has the effect of (a) resetting the microcontroller and (b) changing the source of the address bus from the programming counter to the AEMRMINT download circuit.

Once [PROG*] is asserted, the download data is presented to the [PDAT*] input. This is then clocked into the AEMRMINT shift register using the [PCLK*] pin. Address and data information is then clocked into the core using the protocol shown in Figure 7-5.

All of the inputs are conditioned by a low pass filter (LPFILTER entity). This prevents spurious noise (which is common on PC parallel port cables) from corrupting incoming data.

[31] MUXWWXSS specify a class of multiplexors where 'WW' is the width of input and output buses and 'SS' specifies the number of selectors.

When a complete address and data pair is loaded into the shift register, it is latched into the programming RAM using the [PLCH*] signal. A state machine conditions the write pulse and makes it compatible with the FASM asynchronous ROM block memory. The sequence can be repeated until all or part of the 2,048 x 12 RAM has been loaded. Once loaded, the [PROG*] input is negated, and the core starts up normally (using the new program).

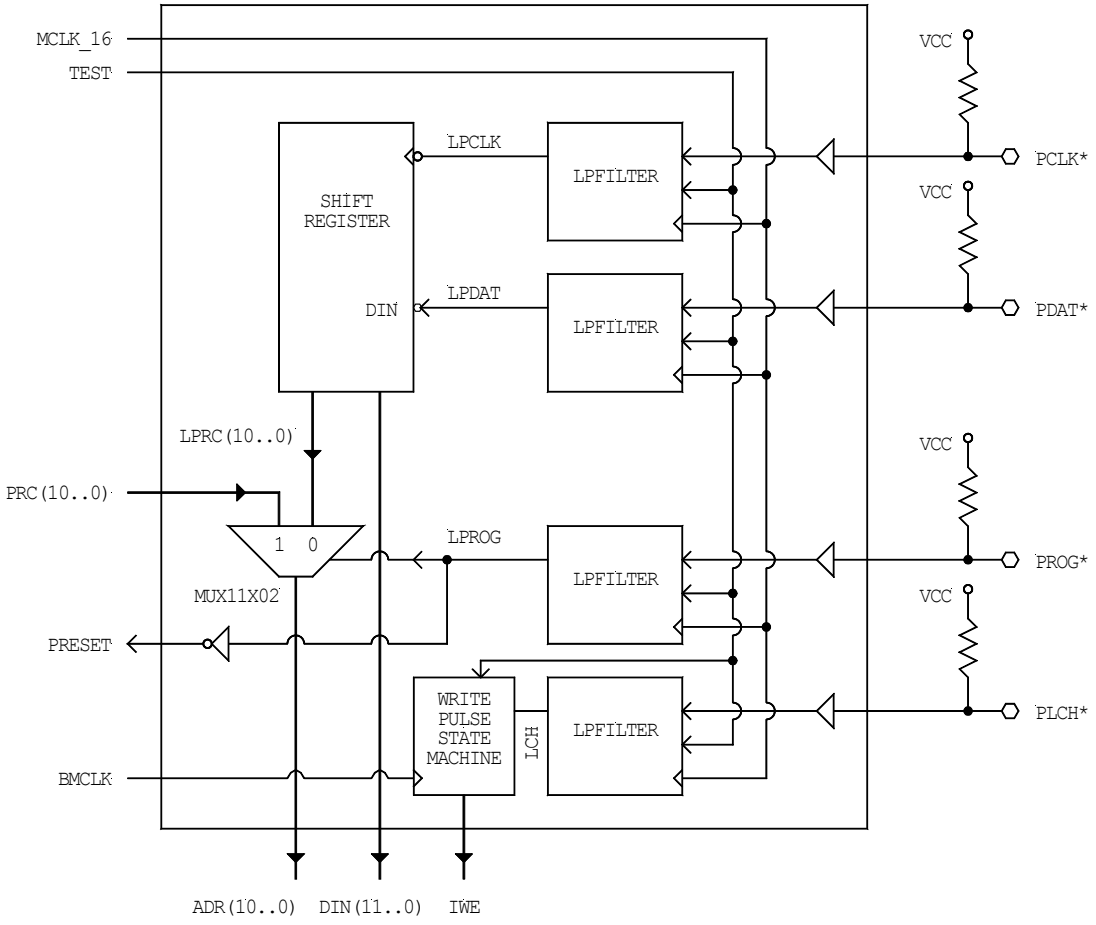


Figure 7-4. AEMRMINT block diagram.

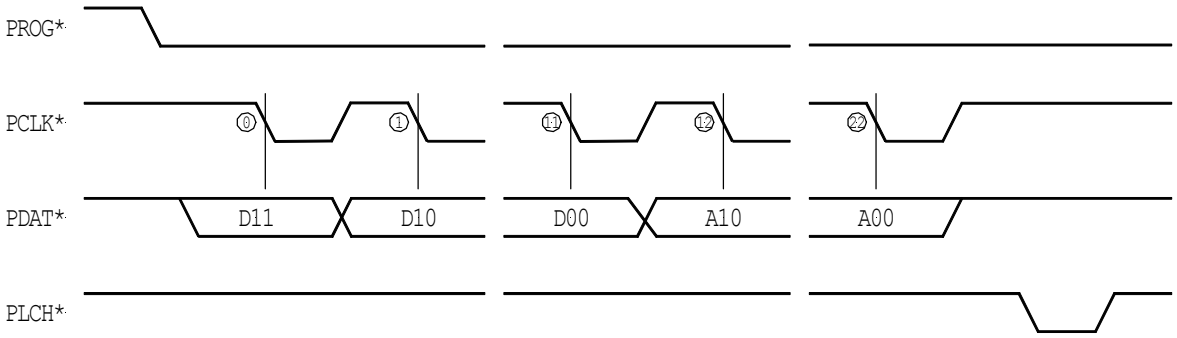


Figure 7-5. AEMRMINT instruction download.

Figure 7-6 shows the memory model that is used by the Altera Block memory. This is a normal FASM asynchronous ROM, except that a data in (DIN) and write enable (WE) ports are added (hence the term 'modified FASM ROM'). These extra functions allow data to be downloaded through the AEMRMINT entity. Furthermore, Altera allows this memory to be initialized. This both allows the CPU to boot with initialized data, and allow downloading of data through the parallel port interface.

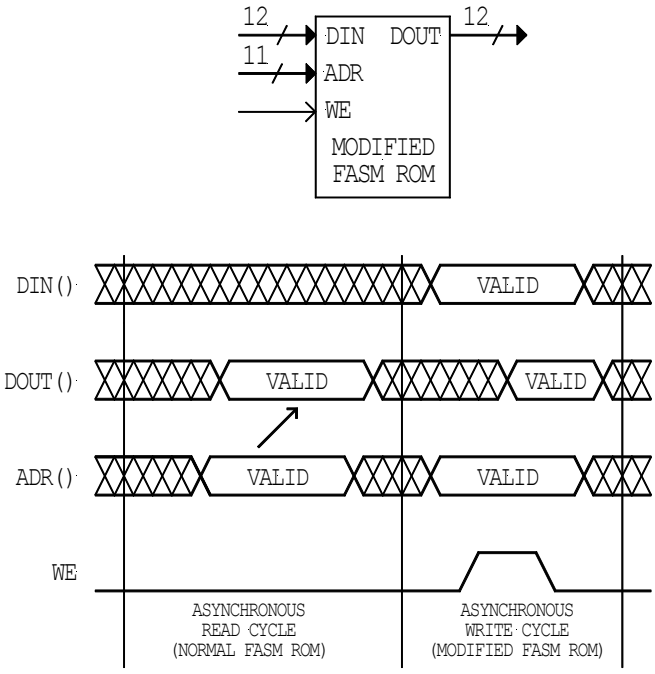


Figure 7-6. Modified FASM ROM.

Figure 7-7 shows the write pulse state machine used by the AEMRMINT entity. This state machine allows a single write-enable (WE) pulse to be generated, regardless of the length of the [PLCH*] signal.

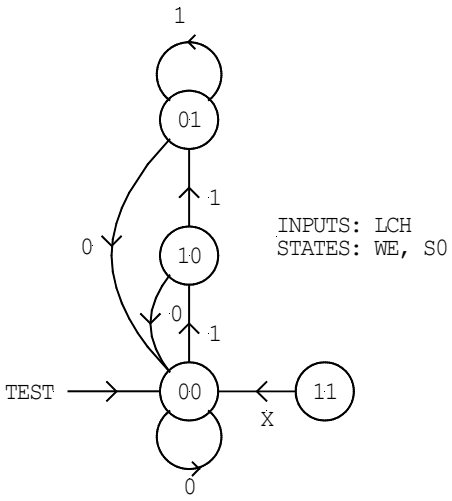


Figure 7-7. State diagram for the write pulse state machine.

7.5.4 AF10EVAL Entity

Other entities used by this module: AEMRMINT, TOPLOGIC

The AF10EVAL entity is the highest level entity used in the Altera FLEX 10KE evaluation project. A block diagram of the entity is shown in Figure 7-8. The heirarchy diagram is shown in Figure 7-9.

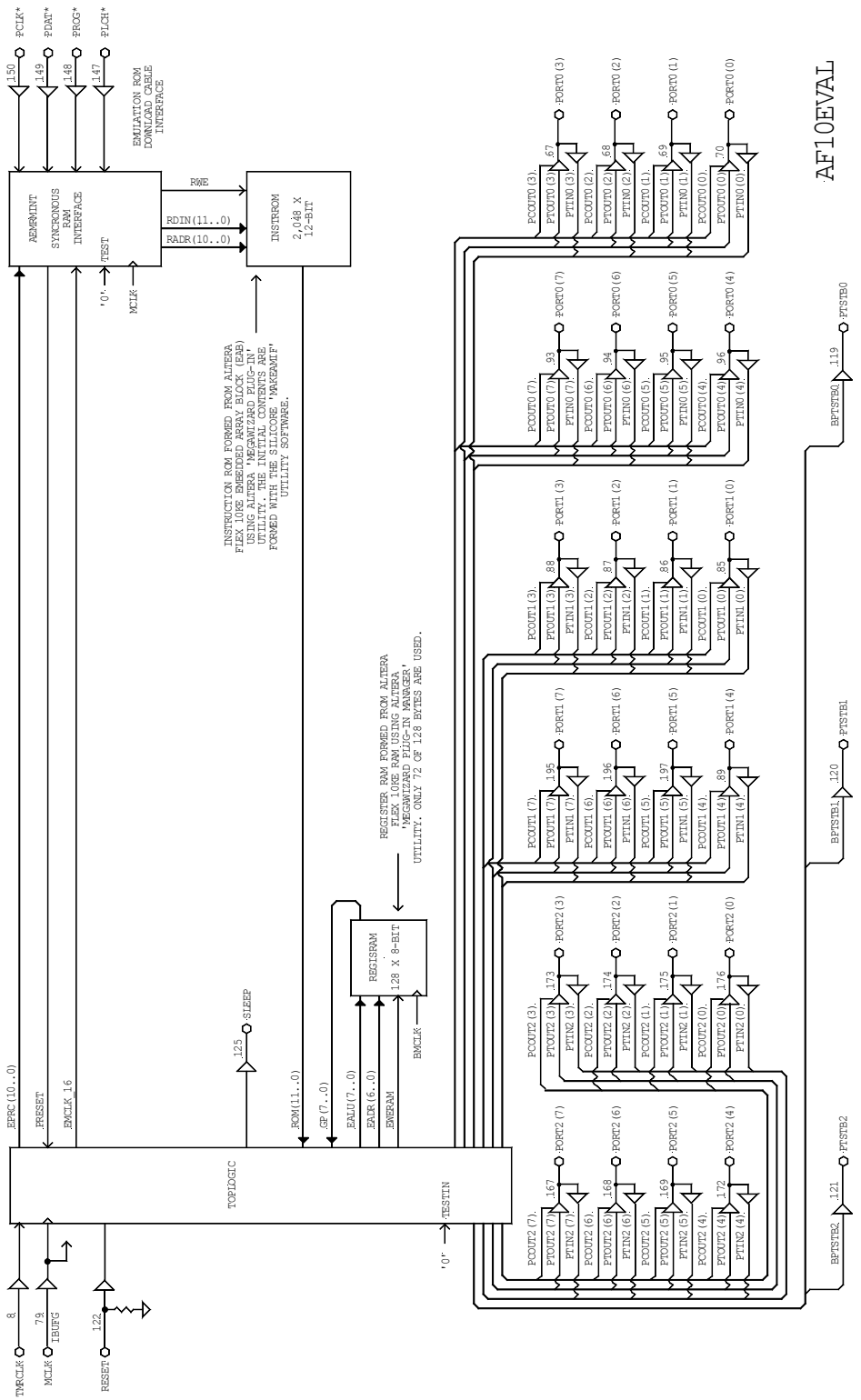


Figure 7-8. Block diagram of the AF10EVAL entity.

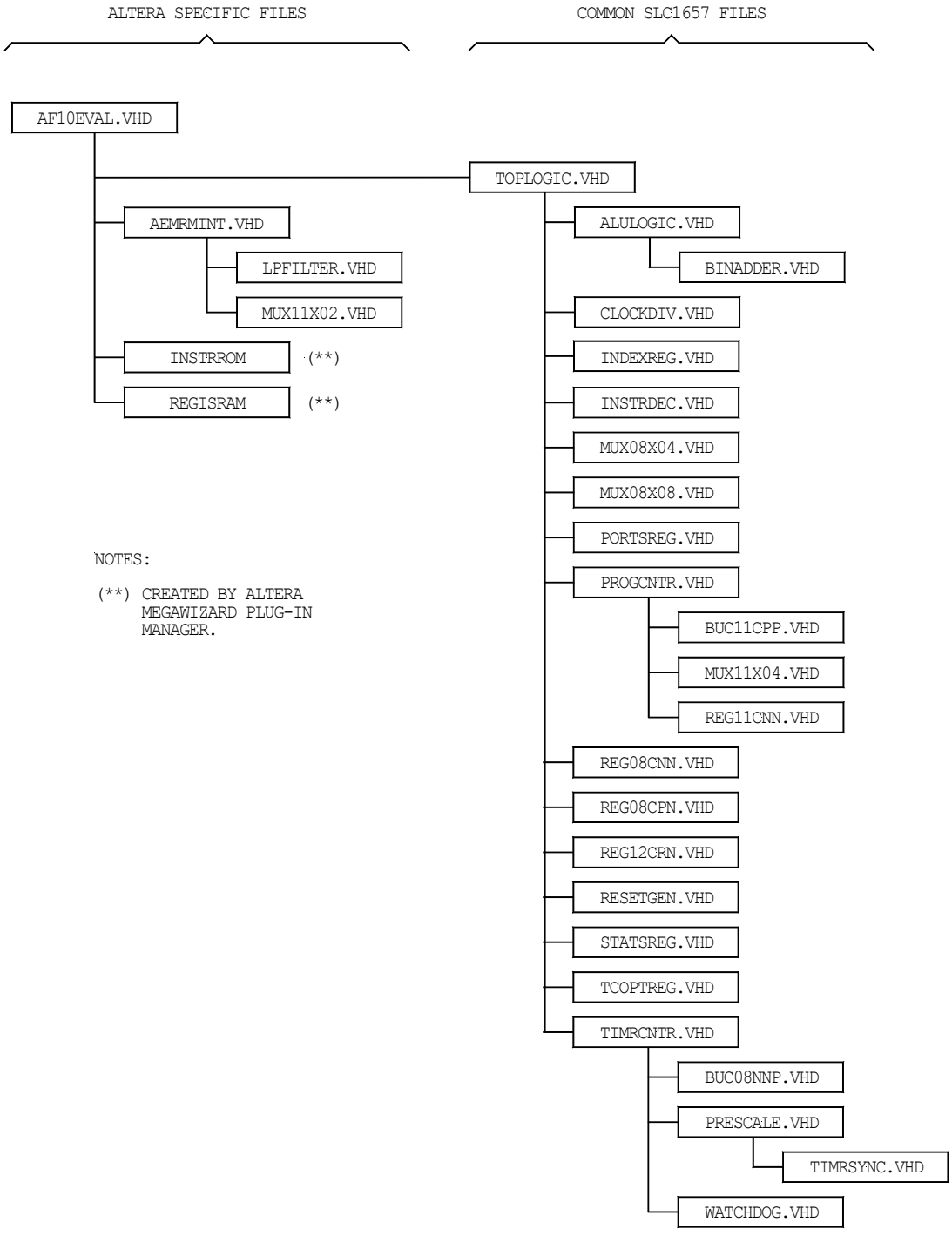


Figure 7-9. Hierarchy diagram for the AF10EVAL entity.

8.0 Implementation on the Agere ORCA 3L FPGA

This chapter describes the steps needed to integrate the SLC1657 onto a Agere[32] ORCA 3L FPGA. An exercise is presented whereby a four function calculator is implemented on an evaluation board.

The purpose of this chapter is to:

- Learn about the SLC1657 Evaluation Kit for the Agere ORCA 3L FPGA.
- Learn the steps needed to integrate a simple system-on-chip.
- Demonstrate how to simulate the TOPLOGIC entity.
- Demonstrate how to synthesize an IP core.
- Create the register RAM and instruction ROM.
- Create a parallel port interface for download and test of application code.
- Integrate the TOPLOGIC core with RAM, ROM and parallel port interface.
- Download and run a 'C' application program for a 10-key calculator.
- Create a fixed PROM.

The following hardware and software tools are used in the exercises:

- PeakVHDL simulation and synthesis tools from Protel International.
- Agere Foundary 2000 Place & Route Software.
- SLC1657 evaluation kit for Agere ORCA 3L FPGA.
- CC5X 'C' compiler from B Knudsen Data.
- DOWNLOAD software for testing application code.
- MAKEOMEM software for integration of a ROMable application software.
- PROM programmer[33].

[32] Agere Systems was formerly known as Lucent Technologies' Microelectronics Group.

[33] The Needhams EMP-30 PROM programmer was used for the exercise (www.needhams.com).

8.1 Evaluation Kit for Agere ORCA 3L FPGA

The evaluation kit for Agere ORCA 3L FPGA allows the user to evaluate and test the SLC1657 microcontroller. The kit includes:

- Evaluation board with Agere ORCA OR3L165B FPGA (see Figure 8-1).
- PROMs for demonstration and calculator functions.
- 16 x 1 LCD display.
- 20-key keypad.
- 5-MHz crystal oscillator.
- 1 KHz RC oscillator.
- 9V battery pack.
- Demonstration program.
- Calculator program.
- PC parallel port download cable and software.
- Technical reference manual.

The evaluation board comes with two embedded software programs. These are 'ODMO', a generic demonstration PROM and 'OCLC', a calculator program. Each resides on a PROM, which contains both the hardware for the SLC1657 microcontroller and the software application programs.

8.1.1 ODMO Software

The ODMO embedded ROM program demonstrates how the SLC1657 can be completely integrated into an FPGA. This includes RAM, ROM and I/O elements. The ODMO embedded ROM demonstration displays the features of the core, and also has a 'stopwatch' function. Follow these simple instructions to operate ODMO:

1) Remove the evaluation board from the anti-static bag[34].

2) Verify that the 44-pin ROM labeled 'ODMO' is located in PLCC socket U5 (to the right of the LCD display). There is a 'spare' PROM socket located at U2 (at the top of the board). This socket is not active, and only serves as a holder for the unused PROM. You might need to switch the PROMs around.

   When removing 44-pin PLCC ROMs, be sure to use the extraction tool supplied with the kit.

3) Connect the +9 VDC battery pack to the evaluation board using connector J1.

[34] The board should be handled at an approved anti-static workstation.

4) Verify that the core boots up, and that display on the evaluation board reads 'SILICORE SLC1657'. This indicates that the microcontroller inside of the FPGA has reset and is running the application code.

5) Push switch 'S17' (the switch marked '0').

6) The features of the core scroll by on the display.

7) Push switch 'S18' (the switch marked '.').

8) Verify that a counter display "00:00 0/10th" appears. Pushing switch S18 ('.') always starts the 'stopwatch' application. Pushing switch S19 ('+/-') starts the stopwatch, and pushing 'S20' ('=') stops it. The stopwatch can be cleared by pushing 'S18' ('.') again. The following table summarizes the switches used by ODMO:

Table 8-1. ODMO Key Functions

Switch	Label	Action
S17	'0'	Marquee of features
S18	'.'	Initiate/clear stopwatch
S19	'+/-'	Start stopwatch
S20	'='	Stop stopwatch

8.1.2 OCLC Software

The OCLC calculator software places the evaluation board into its calculator mode. Install the OCLC PROM into U5 and operate the evaluation board as a four function calculator.

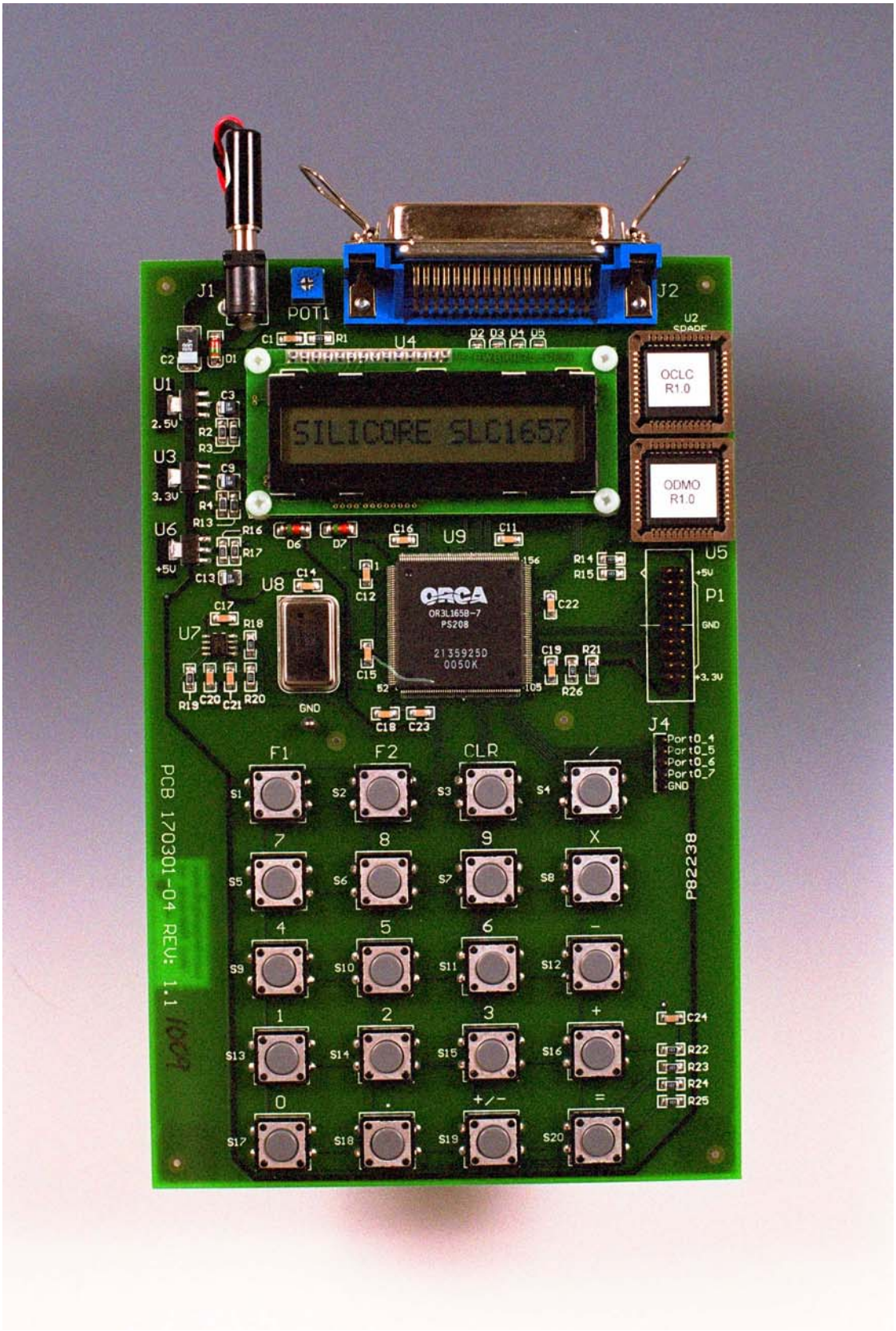


Figure 8-1. Evaluation board for Agere ORCA 3L FPGA.

8.2 The AGO3EVAL Exercise

An exercise is given below to better understand the operation of the SLC1657. This creates a system-on-chip called 'AGO3EVAL', which stands for Agere ORCA 3L EVALuation system. It's a system-on-chip (SoC) that we'll use to design and run a four function calculator.

The AGO3EVAL system uses several VHDL entities. These are described in detail in section 8.5 (below). The user is encouraged to study the descriptions there, along with the VHDL source code. These entities include:

- AGO3EVAL: Agere ORCA 3L Evaluation (top level VHDL entity)
- TOPLOGIC: TOP LOGIC design for the SLC1657.
- REGISRAM: REGISter RAM.
- ROMSEG00-07: INSTRuction ROMs.
- OEMRMINT: ORCA Emulation ROM Interface.

8.2.1 STEP 1 – Simulate the TOPLOGIC Entity

The first step to creating the SLC1657 is to simulate the TOPLOGIC entity. This familiarizes the user with the simulation tools, the SLC1657 IP core and the general operation of all components. This step is the same for all target devices such as Agere, Altera and Xilinx.

Using the Protel PeakVHDL simulation tool, perform the following operations:

1) Create a new directory called 'TLTEST'. [One has been created for you in the EXAMPLES folder if you wish to use it].

2) Open PeakVHDL and create a new project (following the manufacturers directions). Name the project TLTEST, and put it into the 'TLTEST' folder.

3) Add all of the modules in the TOPLOGIC entity into the project. Be sure to preserve the entity hierarchy. The hierarchy is described with the TOPLOGIC entity in Chapter 5. Each entity can be found in its own unique folder in the 'VHDL_source' directory.

   When simulating with the PeakVHDL product, be sure that the highest level module in the hierarchy is the TOPLOGIC test bench (TSTBENCH.VHD from the TOPLOGIC folder).

Also, the TOPLOGIC test bench simulation will need the corresponding test vector files. These are the files with the '*.txt' extension in the TOPLOGIC folder, and should be copied into the TLTEST directory.

When finished, the project window should look something like that shown in Figure 8-2.

4) Simulate the design using the manufacturers directions. At this point the TOPLOGIC entity should simulate with no errors.

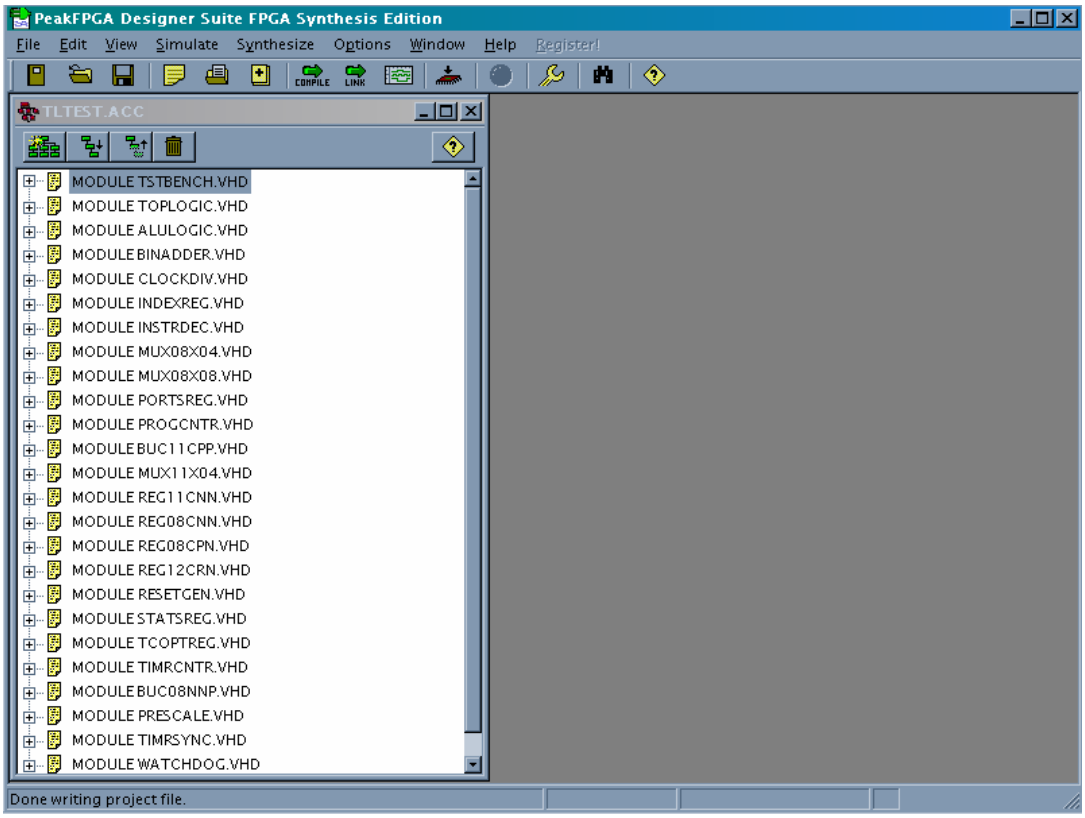


Figure 8-2. PeakVHDL project window.

8.2.2 STEP 2 – Create REGISRAM (Register RAM)

The register RAM is a 128 x 8-bit synchronous memory. It must conform to the FASM SYNCHRONOUS RAM guidelines described elsewhere in this manual. There are many ways to build memories, but the simplest is to use the automatic memory generation software that is supplied with most FPGA place & route tools.

Agere supplies such a tool with their ORCA Foundry 2000 software. It's called SCUBA[35], and is capable of creating the exact memory that's needed. Perform the following instructions to create the REGISRAM entity:

1) Create a directory called 'REGISRAM'. [One has already been created for you in the AGERE | EXAMPLES folder].

2) Open the Agere SCUBA tool, and set it up to create a synchronous RAM in the REGISRAM folder. Set up the options thusly:

   Architecture: OR3c/t00
   Module type: synchronous single port RAM
   Netlist formats: Check EDIF and VHDL
   Module name: REGISRAM
   Output directory: Enter the path to the REGISRAM output directory.
   Netlist destination: Synopsis (Compiler)
   Bus expression style: BusA<0>
   Bus ordering: BIG ENDIAN
   Insert I/O buffers into the netlist: leave unchecked
   RAM size: 128 x 8-bit
   Clock polarity: INVERTED
   Memory file: leave blank (this will be filled in later to create embedded ROMs).
   Other: use SCUBA defaults

3) Generate REGISRAM.

4) Verify that the REGISRAM folder now contains several files, including 'REGISRAM.VHD' and 'REGISRAM.EDN'. These are VHDL descriptions and EDIF files respectively.

8.2.3 STEP 3 – Create ROMSEG00-0F (Instruction ROM)

In the Agere ORCA 3L FPGA, the instruction ROM is formed from sixteen 128 x 12-bit synchronous RAM elements. These eight elements are combined to form a single 2,048 x 12-bit instruction memory.

Multiple ROM elements are used because the ORCA 3L device supports distributed memories, but not block memories. In distributed memory, combinational logic look-up table (LUT) memory is reconfigured as static RAM. However, large distributed memories can be difficult for the Agere router to handle. By experimentation it has been found that if the memory is split up into multiple chunks, then it will be much easier to handle

[35] SCUBA: Synthesis Compiler for User programmaBle Arrays

by the router. Through experimentation, it has been found that the 2,048 word x 12-bit memory (on the OR3L165 device) is best handled with sixteen memory segments.

These eight memory segments are called ROMSEG00, ROMSEG01 and so forth. They are connected together with multiplexors in the AGO3EVAL entity described below.

Although the term 'instruction ROM' is used here, this memory is actually a read/write memory. That's because a downloadable memory interface is used in the example. This allows application software to be downloaded into the instruction ROM. This is accomplished with a parallel port interface called OEMRMINT, and is very useful for software development purposes.

For now, we'll rely on the download capability to get new application code into the microcontroller. However, later we'll initialize the ROM with application code.

Using SCUBA, create the sixteen ROMSEG00-0F entities:

1) Create a directory called ROMSEG. [This directory has been created for you in the AGERE | EXAMPLES folder if you wish to use them].

2) Start SCUBA, and set it up to create a synchronous, single port RAM. Repeat this step for each ROM segment. For ROMSEG00, set up the options thusly:

   Architecture: OR3c/t00
   Module type: synchronous single port RAM
   Netlist formats: Check EDIF and VHDL
   Module name: ROMSEG00
   Output directory: Enter the path to the ROMSEG output directory.
   Netlist destination: Synopsis (Compiler)
   Bus expression style: BusA<0>
   Bus ordering: BIG ENDIAN
   Insert I/O buffers into the netlist: leave unchecked
   RAM size: 128 x 12-bit
   Clock polarity: INVERTED
   Memory file: leave blank (this will be filled in later to create embedded ROMs).
   Other: use SCUBA defaults

   Repeat for each of the sixteen ROM segments. Be sure to name each ROM segment by its own name (e.g. ROMSEG00, ROMSEG01, etc). When finished, the ROMSEG directory should contain the sixteen segments.

8.2.4 STEP 4 – Synthesis

The highest level entity/architecture pair in this system is the VHDL source file named 'AGO3EVAL'. This file ties all of the parts of the system together as described in the block and hierarchy diagrams for the AGO3EVAL entity below.

Using the Protel PeakVHDL synthesis tool, perform the following operations:

1) Create a new directory called 'AGO3EVAL'. [This directory has been created for you in the AGERE | EXAMPLES folder if you wish to use it].

2) Open PeakVHDL and create a new project (following the manufacturers directions). Name the project AGO3EVAL, and put it into the 'AGO3EVAL' folder. [This is already done for you in the AGERE | EXAMPLES folder if you wish to use that.]

3) Add all of the modules in the AGO3EVAL entity into the project. Be sure to preserve the entity hierarchy. The hierarchy is described with the AGO3EVAL entity later in this chapter. The entities relating to TOPLOGIC (e.g. ALULOGIC.VHD) can be found in its own unique folder in the 'VHDL_source' directory. The entities relating to Agere ORCA 3L implementation (e.g. OEMRMINT.VHD) can be found in 'AGERE' directory.

4) Select 'ORCA 3L Series (EDIF)' in the PeakVHDL synthesis options. Also note that 'Agere' devices may be listed under the name 'Lucent'.

5) Synthesize the AGO3EVAL system with PeakVHDL.

6) Look in the synthesis log file, and verify that there were no errors generated by PeakVHDL.

7) Verify that file 'AGO3EVAL.EDN' is present in the directory. This is the EDIF file created by PeakVHDL.

8.2.5 STEP 5 – Place & Route the Design

The EDIF file created in STEP 4 contains all of the microcontroller logic. The next step is to place and route the design on the Agere ORCA 3L FPGA chip. In this example, we'll use the ORCA Foundry 2000 software to place and route the design.

Using the ORCA Foundry 2000 software tool, perform the following operations:

1) Start the ORCA Foundry 2000 Control Center.

2) Create a new project. Use the Control Center wizard to create the project, or under FILE | PROJECT | NAME add the following EDIF files for input:

AGO3EVAL.EDN
REGISRAM.EDN
ROMSEG00.EDN
ROMSEG01.EDN
ROMSEG02.EDN
ROMSEG03.EDN
ROMSEG04.EDN
ROMSEG05.EDN
ROMSEG06.EDN
ROMSEG07.EDN
ROMSEG08.EDN
ROMSEG09.EDN
ROMSEG0A.EDN
ROMSEG0B.EDN
ROMSEG0C.EDN
ROMSEG0D.EDN
ROMSEG0E.EDN
ROMSEG0F.EDN

These are the EDIF files that were created in previous steps, and are the input files to the ORCA Foundry 2000 place and route software. [This has already been done for you in the AGERE | EXAMPLES folder].

3) Under ASSIGN|DEVICE, verify (or enter) the part number of the ORCA 3L FPGA.

4) Assign the pin numbers and maximum frequency of the device. Under ASSIGN | PIN LOCATION assign the pin locations on the FPGA chip. These are identical to those shown in Figure 8-8, and on the schematic diagram for the evaluation board. The easiest way to add these is by adding the preference file called 'TIMING.PRF' to the project. This file is in the AGERE | EXAMPLES foler, and pre-assigns the pinout to match the ORCA 3L evaluation board. Furthermore, this file has timing specifications to specify an MCLK frequency of 5 MHz.

5) Place and route the design by selecting all of the buttons on the ORCA FOUNDRY 2000 STATUS screen. After compiling, look in the 'AGO3EVAL.rpt' file. This file reports the specific details of the place and route process, pin locations and so forth. Verify that there were no errors generated during the run.

8.2.6 STEP 6 – Create the PROM

The final step in implementing the design is to create a PROM (Programmable Read Only Memory). The PROM contains all of the logic necessary to implement the

SLC1657 microcontroller. In the ORCA Foundry 2000 Control Center, create a PROM file by selecting TOOLS | PROM GENERATOR. This generates the file that you will use to create the PROM.

- IMPORTANT –

The ORCA 3L evaluation board uses a Xilinx 1702LPC PROM for configuration. It can be configured for active low or active high reset. The default on most PROM programmers is active high. However, the evaluation board requires that the PROM be configured for an active low reset. If you fail to do this, then the board will not boot up.

8.3 Using the Emulation ROM (Download) Capability

The steps listed in section 8.2 are used to create a complete SLC1657 system on the Agere ORCA 3L evaluation board. That system was programmed onto a PROM, and contains the hardware for the microcontroller. The circuit contains an emulation ROM capability. This allows software instructions to be downloaded into the board over a parallel port cable.

In this example, a sample software program is downloaded over the parallel port cable. To demonstrate its use, a calculator demonstration program called 'CALCDEMO.C' is used. This turns the evaluation board into a four function calculator.

Before downloading, inspect the program called 'CALCDEMO.C'. As you will see, it contains standard 'C' source code. This program is compiled using the 'CC5X' compiler available from B. Knudsen Data (Trondheim, Norway). The compiler produces a file called 'CALCDEMO.HEX', which is the Intel Hex formatted file. Both the 'C' source file and the compiled file are provided in the EXAMPLES directory.

Software is downloaded with a program called 'DOWNLOAD.EXE'. This is an executable file for use under the DOS operating system. DOWNLOAD.EXE reads the Intel Hex formatted file and sends it out the parallel port cable.

Follow these simple instructions to operate the emulation ROM.

1) Remove the evaluation board from the anti-static bag[36].

2) Verify that an 8-pin PROM is loaded into the socket located at 'U5'. All of the PROMs supplied with the SLC1657 demo board include the emulation ROM

[36] The board should be handled at an approved anti-static workstation.

capability. Also, there is a 'spare' PROM socket located at U2. This socket is not active, and only serves as a holder for an unused PROM.

3) Connect the parallel port download cable to the printed circuit board at connector J2. Connect the other end of the cable to the parallel port connector on a PC computer. This cable is a standard Centronics compatible parallel port cable.

4) Connect the +9 VDC battery pack to the evaluation board.

5) If you are using the PROM created above, then the display will show eight 'blanks' on the left hand side of the display. At this point the microcontroller has booted up, but its emulation ROM is empty.

6) On the PC computer, get into DOS mode (if running Windows 95/98). Locate the directory with the program called DOWNLOAD.EXE. Type the following at the DOS command prompt (using the correct path):

    download lpt1 c:\slc1657\Agere\examples\calcdemo\calcdemo.hex

    This causes the object file called 'calcdemo.hex' to be downloaded over the parallel port cable. Once the download is complete, the core will automatically reset and run the program.

    In the command line syntax, 'lpt1' refers to the parallel port number. If 'lpt2' is used (or some other port), substitute the port number.

    If you have the 'CC5X' compiler, then you can edit 'calcdemo.c' and compile it. The compiler creates the Intel Hex formatted file called 'calcdemo.hex', which can be immediately downloaded to the evaluation board.

- IMPORTANT –

DOWNLOAD.EXE is intended to be operated from a DOS environment, including the variants under Windows 95 and 98. However, it will not work with Windows NT. Microsoft has implemented security walls on Windows NT to prevent access to the parallel port.

7) Verify that the core boots up, and that display on the evaluation board reads '0'. This indicates that the microcontroller inside of the FPGA has reset and is running the application code.

8) Try the calculator.

8.4 Creating an Embedded ROM

This section describes how to create an embedded ROM. The embedded ROM contains information for both hardware and software.

The ROM created in the example of section 8.2 (above) causes the SLC1657 to boot up without any valid instructions in memory. Under that scenario, software is downloaded and tested over the parallel port cable. However, once the user is satisfied with the code, then it can be embedded into the ROM. This section describes how to create the same ROM, but instead with embedded software attached.

For this example, we'll use the same 'CALCDEMO.HEX' file to create the embedded ROM. However, in this case the 'CALCDEMO.HEX' file will be converted to Agere '.MEM' files. The Agere '.MEM' files are used to initialize the instruction ROM (ROMSEG00-0F).

This example is intended to run on the AGERE ORCA 3L FPGA. That device uses distributed memory. Unfortunately, it is difficult to route larger (2,048 x 12-bit) memories on this device. In order to solve this problem, the ROM is split into sixteen memory segments, with each segment having a 128 x 12-bit memory size.

To create the Agere '.MEM' file, perform the following operations:

1) Move the file 'calcdemo.hex' into the directory called 'MAKEOMEM' (MAKE Orca MEM file).

2) Convert the file by typing the following at the command line: "MAKEOMEM CALCDEMO.HEX 16". This operation must be done in DOS mode.

3) The conversion utility will create sixteen files called ROMSEG00.MEM, ROMSEG01.MEM, etc. These will be used to initialize the ROM files.

When creating the embedded ROM, follow all of the same steps as shown in section 8.2 However, substitute the following directions for those given in STEP 3 (creating ROMSEG00/01). The modified instructions are:

Using the Agere SCUBA utility, create the ROMSEG00-0F entities:

1) Create a directory called ROMSEG_CALCDEMO. [This step has already been performed for you in the EXAMPLES directory.

2) Open the Agere SCUBA tool, and set it up to create synchronous RAM in the ROMSEG_CALCDEMO folder. Configure everything the same as in section 8.2, except specify the memory initialization file for each ROM segment.

3) Generate ROMSEG00-0F.

4) Move the ROM files into the ROMSEG_CALCDEMO folder.

5) Repeat the rest of the steps for creating the 'AGO3EVAL' above. For your convenience, these steps have already been done for you in the Examples directory under 'ROMSEG_CALCDEMO'.

- WARNING –

The evaluation board uses a 44-pin PLCC package for the PROM. When removing the PROM use a PLCC extraction tool to remove the part. For your convenience, one is supplied with the evaluation kit. Using a screwdriver or other instrument may bend the leads on the PROM, thereby destroying the part.

8.5 VHDL Entity Reference for AGERE ORCA 3L

The VHDL entities used in the AGERE ORCA 3L Evaluation project are given below. These are specific to this implementation. However, the TOPLOGIC entities (given in Chapter 5) are also used in the example.

8.5.1 LPFILTER Entity

Other entities used by this module: NONE

The LPFILTER entity is a digital low-pass filter. Each of the EMROMINT programming inputs is conditioned by LPFILTER. This prevents noise from the PC-compatible download cable from entering the core. Figure 8-3 shows how the filter works.

The filter input is synchronized to the filter clock [MCLK_16] by a D type flip-flop. This prevents metastable and race conditions from occurring within the filter itself. Once the input is synchronized, it enters the LPFILTER state machine. The state machine is designed so that the input signal must be in its asserted or negated state for at least two [MCLK_16] cycles. This causes short (high frequency) pulses to be rejected, and long (low frequency) signals to be accepted.

Figure 8-3 also shows the filter response. Very low frequencies are passed without attenuation. As the speed of the input signal increases to MCLK_16 / 3, the filter begins to reject the input signal. Signals faster than MCLK_16 are rejected[37].

For example, when the SLC1657 clock [MCLK] operates at 5.00 MHz, the filter passes all frequencies up to about 0.104 MHz. As the input signal increases beyond that point, the low-pass filter begins rejecting the input. Signals faster than 0.313 MHz are totally rejected.

[37] If the input signal frequency exceeds MCLK_16 x 2, then the output of the filter will start to pass some signal. However, the noise found on the parallel cable does not exhibit this behavior and is not a problem.

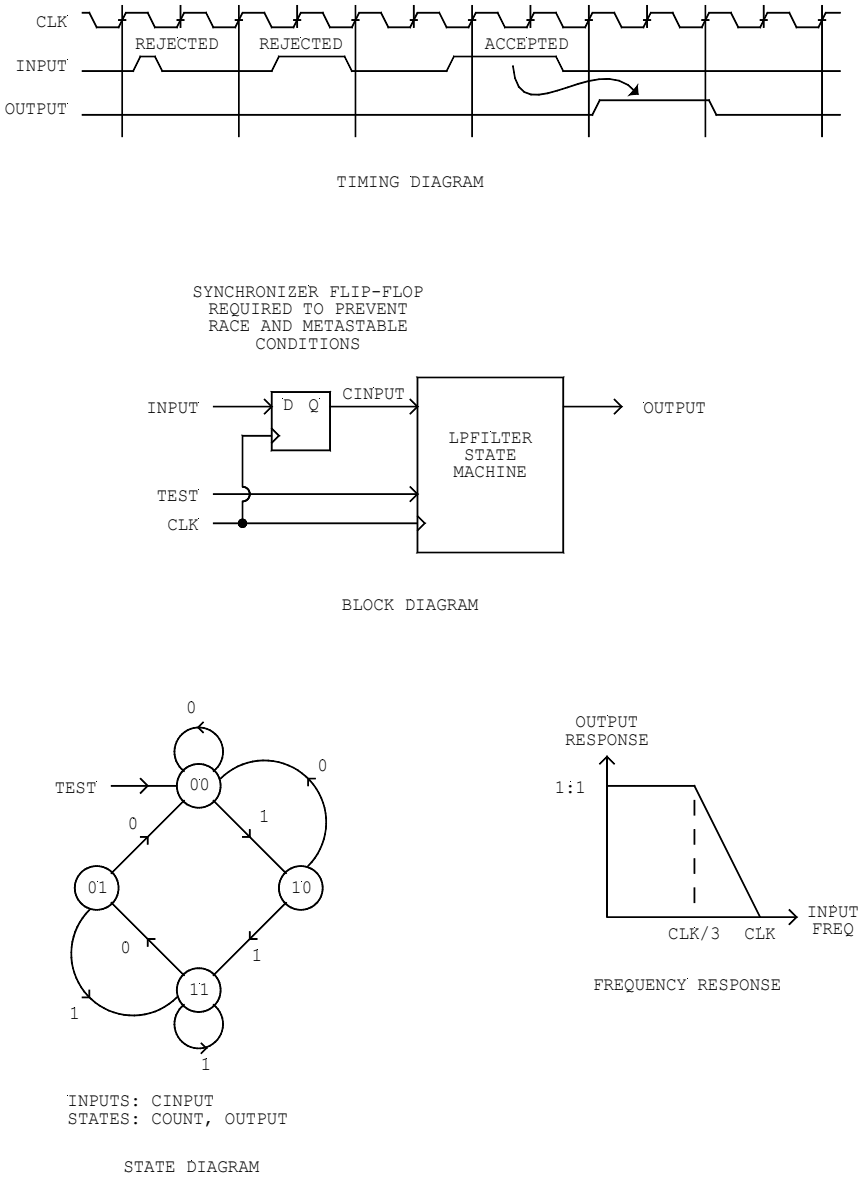


Figure 8-3. LPFILTER entity operation.

8.5.2 MUX11X02 Entity

Other entities used by this module: NONE

The MUX11X02[38] entity multiplexes two, 11-bit buses.

8.5.3 OEMRMINT Entity

Other entities used by this module: LPFILTER, MUX11X02

The OEMRMINT (ORCA EMulation RoM INTerface) entity provides an external interface for 2,048 x 12 ROM emulation. It allows programming through four external pins. The entity also provides signal conditioning for internal memory. The internal memory is assumed to be configured so that it's compatible with the FASM asynchronous ROM described elsewhere in this manual.

Figure 8-4 shows a block diagram of the OEMRMINT entity. During normal operation the external [PROG*] input is negated. This negates the internal [PRESET] signal, and allows the core to run normally. Addresses from the program counter are routed to the RAM address lines through MUX11X02. The RAM then generates instructions which appear at its [ADR(10..0)] output.

Instructions can be downloaded to the core by connecting a programming cable to the programming enable [PROG*], programming clock [PCLK*], programming data [PDAT*], and programming latch [PLCH*] pins. From a PC-compatible computer this can be done via a Centronics parallel port cable in conjunction with the download software.

Figure 8-5 shows the instruction download timing. The download begins when the [PROG*] signal is asserted. This has the effect of (a) resetting the microcontroller and (b) changing the source of the address bus from the programming counter to the OEMRMINT download circuit.

Once [PROG*] is asserted, the download data is presented to the [PDAT*] input. This is then clocked into the OEMRMINT shift register using the [PCLK*] pin. Address and data information is then clocked into the core using the protocol shown in Figure 8-5.

All of the inputs are conditioned by a low pass filter (LPFILTER entity). This prevents spurious noise (which is common on PC parallel port cables) from corrupting incoming data.

[38] MUXWWXSS specify a class of multiplexors where 'WW' is the width of input and output buses and 'SS' specifies the number of selectors.

When a complete address and data pair is loaded into the shift register, it is latched into the programming RAM using the [PLCH*] signal. A state machine conditions the write pulse and makes it compatible with the FASM asynchronous ROM block memory. The sequence can be repeated until all or part of the 2,048 x 12 RAM has been loaded. Once loaded, the [PROG*] input is negated, and the core starts up normally (using the new program).

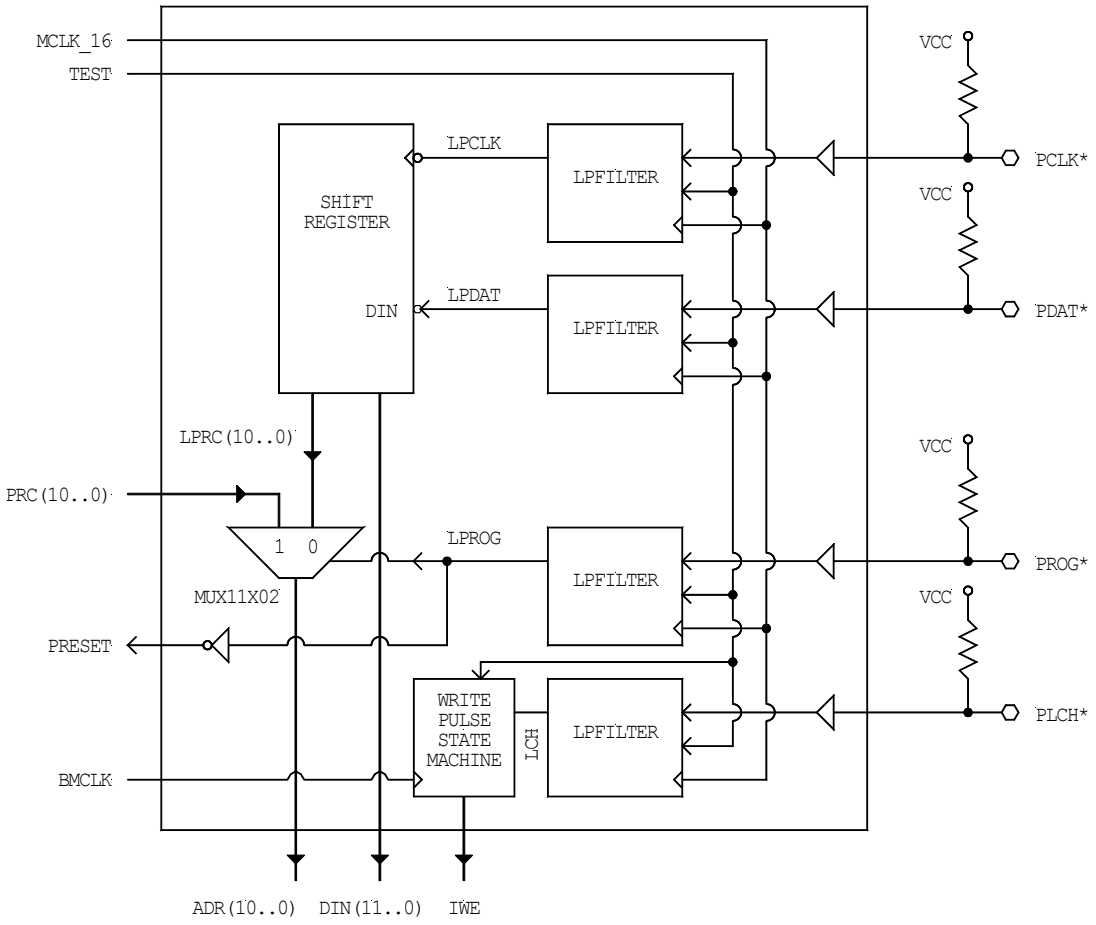


Figure 8-4. OEMRMINT block diagram.

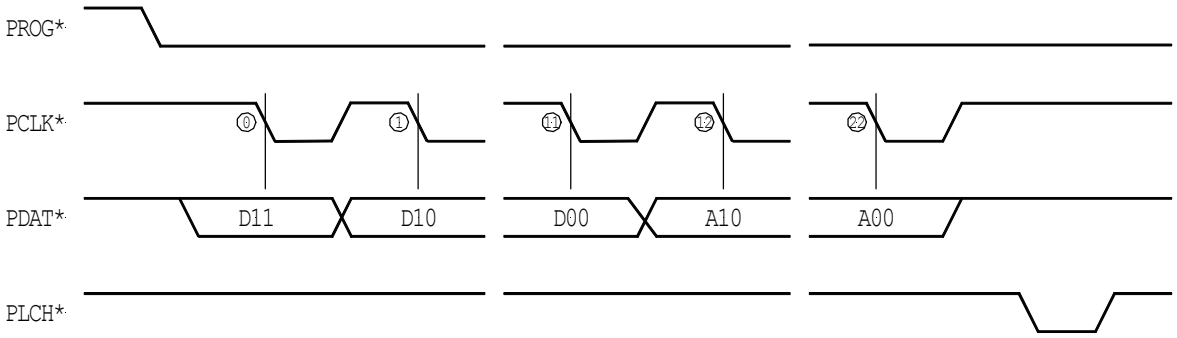


Figure 8-5. OEMRMINT instruction download.

Figure 8-6 shows the memory model that is used by the Agere memory. This is a normal FASM asynchronous ROM, except that a data in (DIN), write enable (WE) and clock (CLK) ports are added (hence the term 'modified FASM ROM'). These extra functions allows data to be downloaded through the OEMRMINT entity. Furthermore, Agere allows this memory to be initialized. This both allows the CPU to boot with initialized data, and allow downloading of data through the parallel port interface.

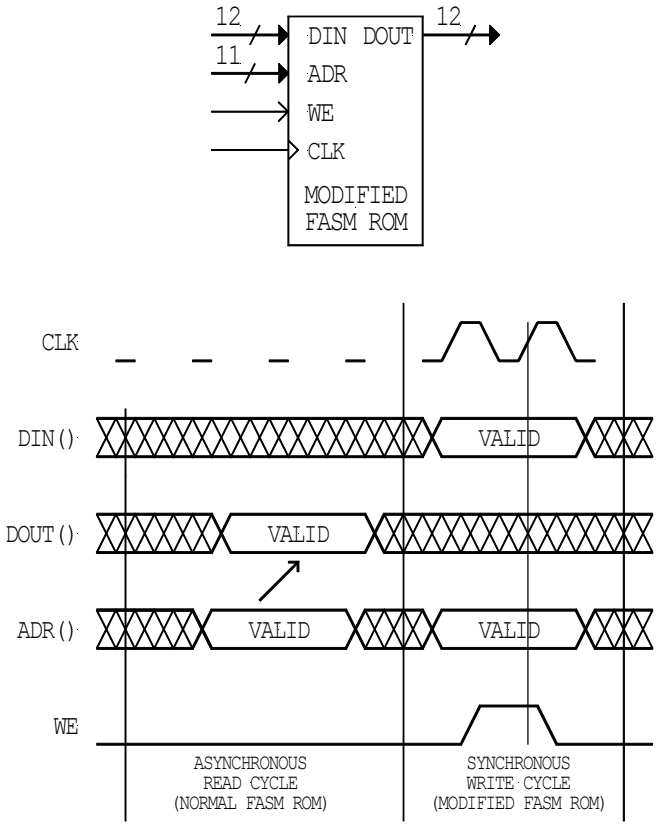


Figure 8-6. Modified FASM ROM.

Furthermore, this ROM is asynchronous during READ cycles, and synchronous during WRITE cycles. Figure 8-7 shows the write pulse state machine used by the OEMRMINT entity. This state machine allows a single write-enable (WE) pulse to be generated, regardless of the length of the [PLCH*] signal. This logic insures that a single, valid write pulse is issued when instructions are downloaded.

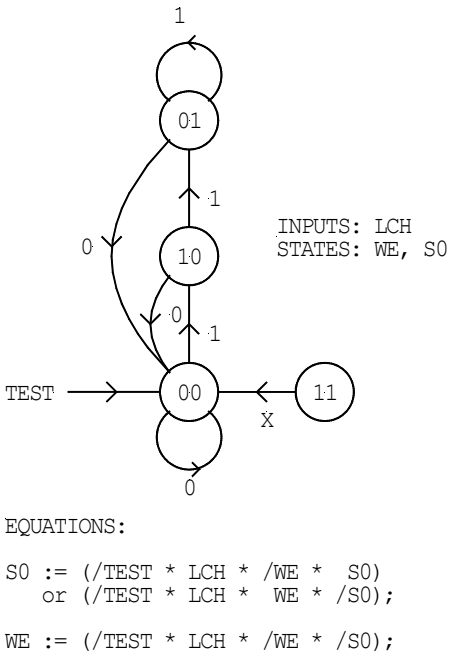


Figure 8-7. State diagram for the write pulse state machine.

8.5.4 AGO3EVAL Entity

Other entities used by this module: OEMRMINT, TOPLOGIC

The AGO3EVAL entity is the highest level entity used in the Agere ORCA 3L evaluation project. A block diagram of the entity is shown in Figure 8-8. The heirarchy diagram is shown in Figure 8-9.

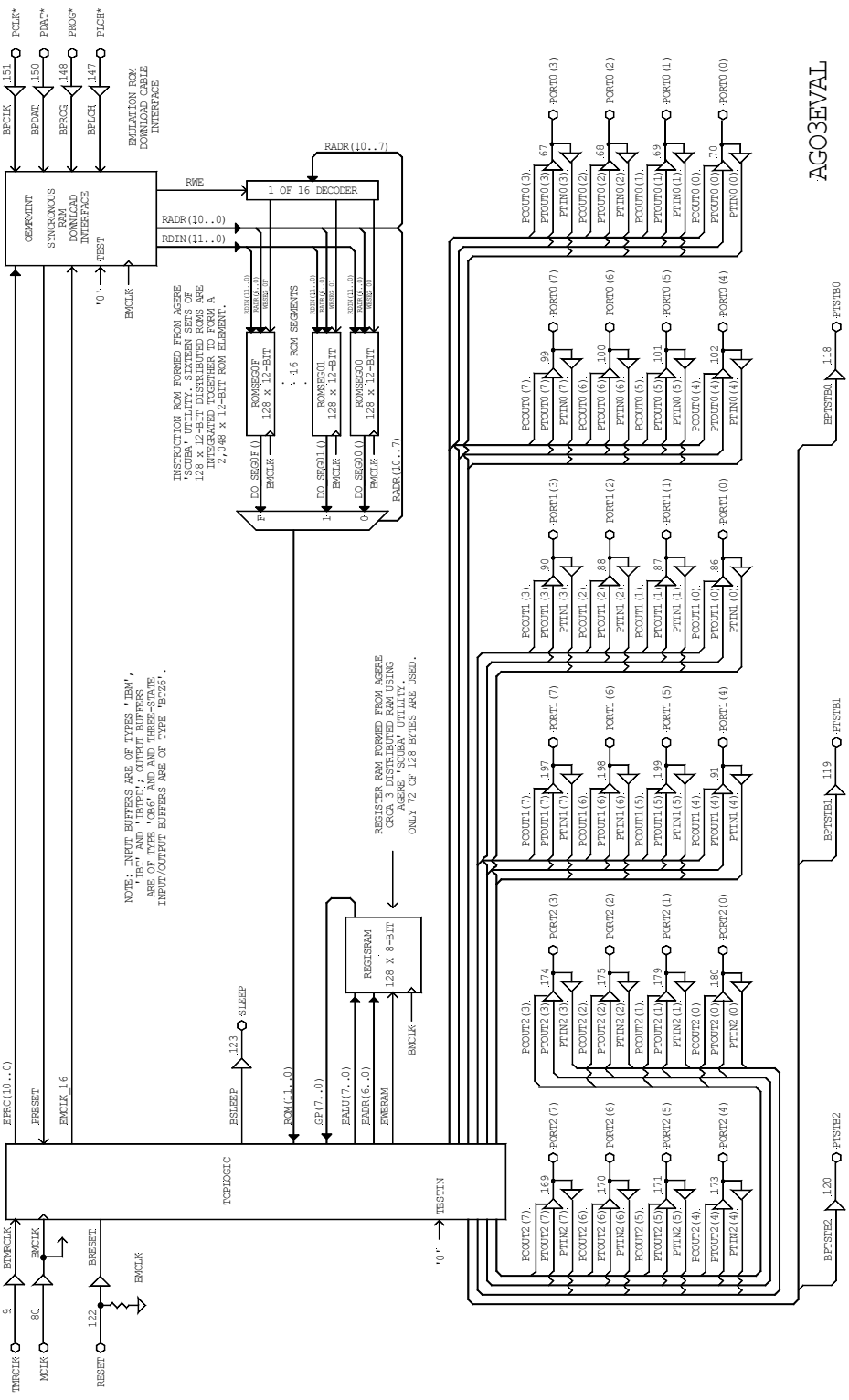


Figure 8-8. Block diagram of the AGO3EVAL entity.

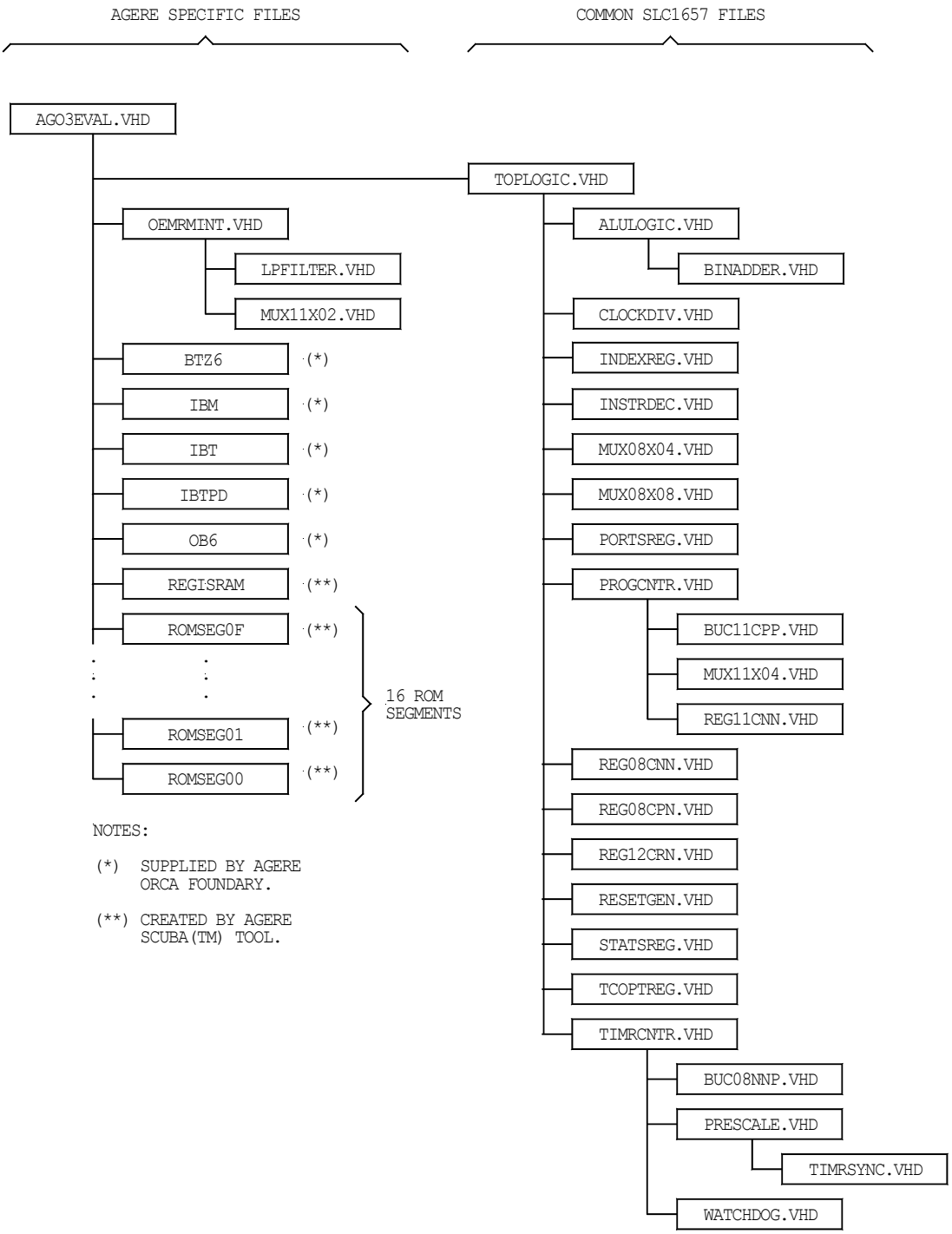


Figure 8-9. Hierarchy diagram for the AGO3EVAL entity.

Appendix A – The Intel HEX Format

Most assemblers and compilers produce data as Intel HEX formatted files. These can usually be identified by a '.hex', '.obj' or '.mcs' file extension, and contain ASCII text. Each line (or record) in the file has the attributes shown in Table A-1.

For example, consider the following record in Intel HEX format:

:0300300002337A1E

This record has the following attributes:

Record length:	0x03 (three bytes of data).
Address:	0x0030 (first data byte goes at address 0x0030).
Record type:	0x00 (normal data).
Data:	0x02, 0x33, 0x7A
Checksum:	0x03 + 0x00 + 0x30 + 0x00 + 0x02 + 0x33 + 0x7A = 0xE2
	The 2's complement of 0xE2 is: 0x100 – 0xE2 = 0x1E.

Also note that the last record of the file is special, and always looks the same. The last record will always be: ":00000001FF".

Table A-1. Attributes for Each Line In an Intel Hex Formatted File	
Character Number In Record	Description
1	Colon ':' record marker (ASCII 0x3A).
2-3	Record length. This field contains the number of data bytes in the record, and is represented by a two-digit hexidecimal number. This is the total number of data bytes, not including the checksum byte nor the first nine characters of the record.
4-7	Starting address. This field contains the address where the data should be loaded. This is a four digit hexidecimal value (i.e. 0x0000 - 0xFFFF).
8-9	Record type. This field indicates the type of record. The possible values are: 0x00 - Data record 0x01 - End of file record. 0x02 - Extended address. 0x03 - Start segment address record. 0x04 - Extended linear address record. 0x05 - Start linear address record.
10-N	Data. Data bytes represented by two digit hexidecimal numbers.
Last Two	Checksum. The last two characters of the line are a checksum for the record. The checksum value is calculated by taking the two's complement of the sum of all the preceding bytes excluding the colon at the beginning of the line, and the checksum itself. Only the least significant byte of the sum is used in the calculation. The two's complement can be found by subtracting the sum from 0x100. For example, the two's complement of 0x03 is: 0x100 – 0x03 = 0xFD.

Appendix B – GNU LESSER GENERAL PUBLIC LICENSE

GNU LESSER GENERAL PUBLIC LICENSE - Version 2.1, February 1999

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

PREAMBLE

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software packages--typically libraries--of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

GNU LESSER GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) The modified work must itself be a software library.

b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free

Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

HOW TO APPLY THESE TERMS TO YOUR NEW LIBRARIES

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> <one line to give the library's name and a brief idea of what it does.>
> Copyright (C) <year> <name of author>
>
> This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.
>
> This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.
>
> You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
>
> Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, ifnecessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the library `Frob' (a library for tweaking knobs) written by James Random Hacker.

<signature of Ty Coon>, 1 April 1990
Ty Coon, President of Vice

That's all there is to it!

Appendix C – GNU Free Documentation License

GNU Free Documentation License - Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or non-commercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque

copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added

by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the
Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this
License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document
> under the terms of the GNU Free Documentation License, Version 1.2
> or any later version published by the Free Software Foundation;
> with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
> A copy of the license is included in the section entitled "GNU
> Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the
> Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

INDEX

ACCUM register, 16, 19, 29
ACLC software, 131
ADD instruction, 51
ADMO software, 130
ADR router [ADR(6..0)], 87
AEMRMINT entity, 144, 147
AF10EVAL exercise, 133
Agere Systems
  AGO3EVAL exercise, 154
  evaluation kit, 151
  ORCA 3L implementation, 150
AGO3EVAL entity, 169
AGO3EVAL exercise, 154
ALF function generator [ALF(3..0)], 89
Altera
  AF10EVAL exercise, 133
  evaluation kit, 130
  FLEX 10KE implementation, 129
ALULOGIC entity, 19, 37, 79
AND instruction, 51
ANDI instruction, 52
application specific entities, 77
applications, 5
architecture, system, 9
ASGN bit, 32
ASIC implementation, 76
ASIC testability, 12
assemblers / simulators, 27
bank
  general purpose registers, 37
  instruction, 10, 20
  register, 10, 17, 24, 33
banked general purpose registers, 38
battery powered applications, 11, 15, 46
BCLR instruction, 52
bi-directional three-state I/O mode, 30
BINADDER entity, 81
bitwise instructions, 86
BRA instruction, 20, 22, 53, 87
BSET instruction, 53
BSR instruction, 16, 22, 54, 94
BTSC instruction, 54
BTSS instruction, 55
BUC08NNP entity, 82
BUC11CPP entity, 82
C (carry) bit, 37, 99
C compilers, 28
changes from SLC1655, 79
clock
  duty cycle, 14
  frequency, 11, 46
  skew, 69
CLOCKDIV entity, 82
CLR instruction, 55
compilers, 28
core integration, 73
core overview, 9
counter/timer operation, 43
DEC instruction, 56
DECSZ instruction, 56
distribution disk, 72
download software, 10
embedded ROM. *See* ROM, embedded
emulation ROM. *See* ROM, emulation
end-of-cycle generator, 90
entity. *See* entity name
external architecture, 11, 13
FASM
  asynchronous ROM, 71, 124
  synchronous RAM, 70
features of the SLC1657, 6
FIFO buffers, external I/O, 10
flush instruction pipeline, 20
FPGA implementation, 76
fuzzy logic, 28
general purpose registers, 17, 38
hardware reference, 79
Harvard architecture, 6, 9
HEX file format, 172
I/O ports
  back-to-back cycles, 40
  driver synthesis, 67, 77
  general description, 10
  options, 39
  reset operation, 30
  signals, 39
  user specified elements, 12
IB0-1 bits, 35
IEEE standards, 68, 70, 75, 78
immediate instructions, 87
implicit instructions, 84
implicit registers, 16
INC instruction, 57
INCSZ instruction, 57
INDEX register, 25, 33, 37
INDEXREG entity, 83
INDIRECT register, 25, 33
installation, 72
INSTRDEC entity, 83
instruction. *See* instruction mnemonic
instruction fetch, 19
instruction mnemonic conversion, 49
instruction pipeline, 19, 20
instruction ROM. *See* ROM
instruction set, 27, 50
integration, 74

Intel HEX file format, 172
internal architecture, 16
internal operation, 18
interrupts, 18
INTRCONV entity, 90
LCLC software, 152
LDMO software, 151
License, source code (LGPL), 174
License, user manual (FDL), 184
LPFILTER entity, 120, 142, 164
Lucent Technologies. *See* Agere Systems
Manual license (FDL), 184
MCLK signal, 14
Microchip Technology Inc., 9
microcontroller topology, 9
MOV instruction, 58
MOVA instruction, 58, 88
MOVI instruction, 59
MOVP instruction, 30, 59
MOVT instruction, 31, 60
MUX08X04 entity, 91
MUX08X08 entity, 91
MUX11X02 entity, 122, 144, 166
MUX11X04 entity, 91
NC (nibble-carry) bit, 37, 99
NOP instruction, 60
NOT instruction, 61
OEMRMINT entity, 166
op-codes, instruction, 11, 85
OR instruction, 61
ORI instruction, 62
overview, SLC1657, 5
PC0-2 registers, 16, 29, 91
PCLK* signal, 14
PCOUT0-2(7..0) signals, 14
PD (power-down) bit, 36, 44, 46, 62, 97, 99
PDAT* signal, 14
PIC16C57 compatibility, 48
PIC16C57, compatibility with, 9
place-and-route, 77
PLCH* signal, 14
PORT0-2 registers, 38, 91
PORTSREG entity, 40, 91
power-down, 10, 11, 46
power-up state of flip-flops, 70
PRESCALE entity, 92
prescaler, 32
  changing, 45
  general operation, 11
  select bits PS0-2, 32
  TCO register, 31
PROG* signal, 15
PROGCNTR entity, 19, 93
PROGCNTR register
  general description, 34
  preloading, 24
  reading, 34
program counter, 22
program memory, 20
PS0-2 (port select) bits, 32
PTIN0-2(7..0) signals, 15
PTOUT0-2 signals, 31
PTOUT02(7..0) signals, 15
PTSTB0-2 signals, 10, 15, 39, 92
PWRDN instruction, 94
  current consumption, 46
  discrimination after reset, 44
  general description, 11, 62
  INSTRDEC entity operation, and, 86, 90
  program counter operation during, 94
  reset operation during, 97
RAM
  FASM compatible, 70
  general description, 10
  required resources on target device, 70
  synthesis, 67, 77
reference books
  PIC16C57, 28
  VHDL, 78
REG08CNN entity, 96, 97
REG08CPN entity, 96
REG12CRN entity, 97
register bank. *See* bank
register set, 17, 29
reset
  general operation, 21, 38
  I/O ports, 39
  INSTRDEC entity, 90
  instruction, 21
  power-up state of flip-flops, 70
RESET signal, 15
RESETGEN entity, 97
resources required on target device, 69
RET instruction, 16, 24, 63, 94
RISC microcontroller, 5, 9, 83
ROL instruction, 63
ROM
  embedded, 12
  emulation, 10, 12, 14, 15, 36, 38, 111, 116, 122, 133, 138, 144, 154, 160, 166
  FASM compatible, 71
  general operation, 10, 19
  speed, 19, 84
  synthesis, 67, 77
ROR instruction, 64
RWT instruction, 64
SCUBA, 156
SEL router [SEL(1..0)], 88
SEMRMINT entity, 122, 126
shared general purpose registers, 38
shared registers, 17
signal. *See* signal name
simulation, 75
skill level, recommended, 7
SLC1655 upgrade, 10, 21, 79, 83, 93, 99
SLEEP signal, 15, 46, 62, 85
SLV2INTPAK, 73, 90
soft core, 5, 67
software tools, 9, 27
Source code license (LGPL), 174
special purpose registers, 17
stack

general operation, 24
registers, 16
STACK1-2 registers, 16
standard instructions, 86
STATSREG entity, 99
STATUS register, 35
SUB instruction, 65
SWPN instruction, 65
synthesis. *See* VHDL, synthesis
target device resources required, 69
TCO register, 16, 31
TCOPTREG entity, 100
TCS (T/C clock source) bit, 31
test benches. *See* VHDL, test benches
timer operation, 41
timer/counter
general operation, 10, 26, 41
TCO register, 31
timing specification, 76
TIMRCNTR entity, 101
TIMRCNTR register, 34
TIMRSYNC entity, 102
TMRCLK signal, 16
TMRCNT edge select, 32
TMRCNT signal, 31
TO (timeout) bit, 36, 44, 46, 62, 97, 99
TOPLOGIC entity, 11, 104
TSE (timer select edge) bit, 32
VHDL
entity/architecture pair, 69
hardware description language, 67
portability, 69
pre-synthesis, 75
RAM entity, 70
reference books, 78
simulation tools, 67
source files, 74
standards, 68
synthesis, 67, 77
synthesis tools, 67
test benches, 67, 73
three-state bus usage, 69
variable type usage, 69
watchdog
general operation, 11, 41, 44
timer, 10
WATCHDOG entity, 104
WDT (watchdog enable) bit, 31
work.SLV2INTPAK.all, 73, 90
XCLC software, 109
XDMO software, 108
Xilinx
block memory, 124
evaluation kit, 108
Spartan 2 implementation, 107
XSP2EVAL exercise, 111
XOR instruction, 66
XORI instruction, 66
XSP2EVAL exercise, 111
Z (zero) bit, 37, 99