

Core8051: Building a Core8051 System on Actel Flash Devices (and the APA300 Starter Kit)

Introduction

This application note describes the steps and design issues involved in implementing a Core8051 design on Actel Flash FPGA devices. Specifically, there is a design example that targets the ProASIC^{PLUS}® APA300-PQ208 Evaluation Board. Users who are familiar with Core8051 and APA device documentation may want to review the "Introduction" section and skip to the "Core8051 on ProASIC^{PLUS} and APA300 Starter Kit" section on page 9.

The basic architecture of the example design for this application note is illustrated in Figure 1. Core8051 is a self-contained microcontroller. For this design, the SRAM inside the APA300 device is used for CODE, XDATA, and Data memory (descriptions of these memories are in the "Core8051 Implementation Considerations" section on page 3 of this document).

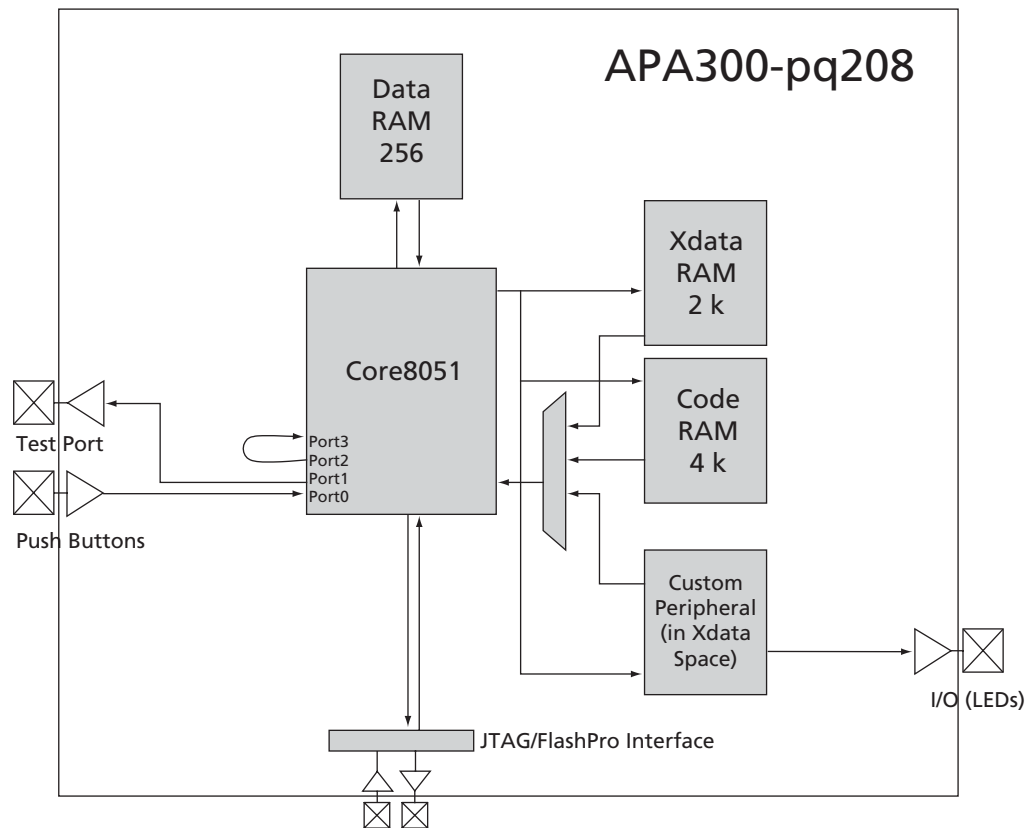


Figure 1 • Example of Simple Core8051 Design (example project)

Required Software and Hardware for Downloading and Running the Example Project:

- ProASIC^{PLUS} Starter Kit
 - Actel Part Number APA-EVAL-KIT
 - Note: this design will also work on the discontinued APA-EVAL-BRD.
- Actel Libero[®] 0EDA Software (Actel order code "LIB-PL-PC-N-1YR")
- Actel FlashPro Lite (Actel order code "FlashPro Lite")

Recommended Additional Products for Modifying or Debugging with the Example Project:

- FS2 tool kit for Core8051
 - FS2 Part Number ISA-ACTEL51
 - Included with Core8051 IP from Actel
- Core8051 IP License (Actel order code "Core8051-SN/AN/AR")
- Keil uVision2/C51 C Compiler/Debugger (optional)
 - Keil Part Number PK51 or DK51
 - Evaluation License is also available (Included with Core8051 IP from Actel)
- SDCC C Compiler (optional – alternative to Keil)
 - (<http://sdcc.sourceforge.net>)
 - FS2 Debugger Upgrades (optional) www.fs2.com

Core8051 and the APA300 FGPA

Core8051 Overview

The Core8051 macro is a high-performance, 8-bit microcontroller IP core. It is a fully functional 8-bit embedded controller that executes all ASM51 instructions and has the same instruction set as the 80C31 and other 8051 family microcontrollers. Unlike competitive MPU products, Core8051 is a true MCU and provides software and hardware interrupts, a serial port, and two timers.

The Core8051 architecture eliminates redundant bus states and implements a two-stage pipeline with parallel execution of fetch and execution phases. Since a cycle is aligned with memory fetch when possible, most of the one-byte instructions are performed in a single cycle. Core8051 uses one clock per cycle. This leads to an average performance increase of eight times (in terms of MIPS) with respect to the Intel device running at the same clock frequency due to the drastically lower number of cycles required per instruction.

This core has been implemented in all of the most popular Actel FPGA devices (see [Core8051 Datasheet](#) for details). The Actel Flash devices are a particularly good fit with their flash-based programmable logic cells and abundant RAM.

ProASIC^{PLUS} Overview

The ProASIC^{PLUS} family of devices, second generation Actel Flash FPGAs, offers enhanced performance over the Actel ProASIC family. It combines the advantages of ASICs with the benefits of programmable devices through nonvolatile Flash technology. This enables engineers to create high-density systems using existing ASIC or FPGA design flows and tools. In addition, the ProASIC^{PLUS} family offers a unique clock conditioning circuit based on two on-board phase-locked loops (PLLs). The family offers up to one million system gates, supported with up to 198 kbits of two-port SRAM and up to 712 user I/Os, all providing PCI compatible performance.

ProASIC^{PLUS} Evaluation Board

The ProASIC^{PLUS} evaluation board has on-board voltage regulation, enabling you to set the I/O voltages (V_{DDP}) to either 2.5 V or 3.3 V. You can generate the system clock using the on-board oscillator and ProASIC^{PLUS} PLLs. Eight LEDs and four switches provide simple inputs and outputs to the system. Prototyping headers connect to all the ProASIC^{PLUS} device I/Os, enabling you to easily add components to the evaluation board. Finally, the board is equipped with programming headers to support ISP programming using FlashPro, FlashPro Lite, or Silicon Sculptor II.

Actel supplies these evaluation boards with an APA075-PQ208 or APA300-PQ208 soldered on, or as a socketed board without an APA chip. An APA300 is required for the example design.

Core8051 Implementation Considerations

There are several configuration options for Core8051 available to the user. These options in configuring Core8051 are defined in this section. This section gives general background and does not specifically describe the example project. The example project details are documented starting with the "Core8051 on ProASIC^{PLUS} and APA300 Starter Kit" section on page 9.

Core8051 Memory Map Considerations

There are four separate memory regions used in Core8051:

- DATA – 256 bytes x 8 bits wide, used for dynamic storage of program data (registers, stack, variables)
- CODE – 64 kbytes x 8 bits wide, used for program storage and interrupt vectors
- XDATA – 64 kbytes x 8 bits wide, used for storage of large data sets, custom-designed peripherals, and extended stack space if necessary
- SFR – 128 bytes x 8 bits wide, a combination of internal (to the core) and external memory for special function registers

CODE, DATA, and XDATA memory spaces are not a part of Core8051 and therefore must be implemented by the user either internal or external to the FPGA. The SFR memory space consists of two parts: internal to the core and external. The internal SFR memory space is implemented in the core. The user must implement the optional, external SFR memory space outside the core.

The Core8051 memories can be implemented either as synchronous or asynchronous. The recommended mode when using RAM that is inside of ProASIC^{PLUS} devices is to make them synchronous on write and asynchronous on read. This is the default configuration that should be considered first. Use the same clock source as the Core8051 for the RAM WCLOCK. The timing for some designs can be improved by changing to synchronous reads as well as writes. When using off-chip RAM, the timing between the Write Enable, Output Enable, and Data should be carefully considered and reviewed after layout.

CODE and XDATA Memory Implementation

The CODE memory space is 64 kbytes x 8 bits wide, and is used for program storage and interrupt vectors. After reset, Core8051 always starts at address 0x0000 in CODE space.

The XDATA memory space is also 64 kbytes x 8 bits wide and is used for storage of large data sets, custom-designed peripherals, and extended stack space if necessary. XDATA and CODE can be constructed to consume less than the entire 64 k memory map, allowing for more efficient memory usage. However, the user must ensure that sufficient memory space is allocated for the program. The Keil compiler/debugger always fills the CODE space starting at 0x0000 (or 0x0800 in the evaluation version) and goes upward. The upper limit of CODE space is dependent on the size of the application. Similarly, the size of XDATA RAM needed is defined by the application code.

The Core8051, DATA, XDATA, and CODE each have their own write-enable (WR) and read (RD) enable signals. It is feasible and common practice to combine the CODE and XDATA memory spaces in the same physical memory. In some applications, XDATA and CODE are even overlaid in the same physical memory region. In these cases, the user must prevent the CODE memory space from being overwritten by XDATA variables. The signals used to implement all of the memory interfaces are defined in [Table 1 on page 4](#).

Table 1 • Core8051 Memory Bus Signal Summary

Memory Space	Data Bus	Address Bus	Rd/Wr Control	Other Controls	Description
XDATA	memdatao[7:0] – output memdatai[7:0] –input	memaddr[15:0]	memwr – write enable memrd – read enable	memacki – acknowledge (for multi-cycle operation)	Memory bus for large data sets in RAM, Flash programming, or custom 8051 peripherals
CODE	memdatao[7:0] – output memdatai[7:0] –input	memaddr[15:0]	dbgmempswr – write enable mempsr – read enable	mempacki – acknowledge (for multi-cycle operation)	Memory bus for program code memory in RAM or Flash
DATA	ramdatao[7:0] – output ramdatai[7:0] –input	ramaddr[7:0]	ramwe – write enable ramoe – read enable		Memory bus for internal Core8051 registers and stack space in RAM
XSFR	sfrdatao[7:0] – output sfrdatai[7:0] – input	Sfraddr[6:0]	sfrwe – write enable sfrre – read enable		(optional) Bus for mapping custom peripherals into Core8051 SFR space

Core8051 treats the CODE memory space as read only. The CODE memory can only be written by the OCI debug circuitry. If the OCI is not used, the application designer will have to develop an alternative method of loading the CODE memory. The user can intentionally map XDATA over CODE memory space so that the 8051 can update all or part of its own code, but this is an advanced application that is beyond the scope of this document. Code/data banking, the use of additional 64 k blocks for CODE or XDATA memory space to accommodate applications with large code or data requirements, can be easily implemented with Core8051. This is due to the flexibility of Actel FPGAs in creating chipselect signals that differentiate between 64 k banks. However, code/data banking is not covered in this document.

DATA Memory Implementation

Internal Data (DATA) may be created on-chip or off-chip. However, it requires single cycle access. This may be a determining factor in your final system clock speed. DATA must always be kept in a separate memory space as it can be accessed in the same cycle as CODE or XDATA. It is common to create DATA memory as 128 or 256 bytes. The lower 128 bytes are required in all Core8051 designs and respond to direct or indirect addressing. The upper 128 bytes provide additional space for variables and the Core8051 stack; however, this space can only be addressed indirectly. Direct addressing of the upper 128 bytes will result in an SFR access, not a DATA access. An example of indirect addressing is an instruction where the target address is stored in a register. Direct addressing is when the address is explicitly expressed in the instruction.

Harvard vs. Von Neumann Architecture and the 8051

A Harvard architecture microprocessor is one with separate memory spaces and busses for code and data. A Von Neumann Architecture puts both code and data into the same memory space. Core8051 is a Harvard architecture from the compiler/software point of view. However, in the actual hardware implementation it is possible to overlay the two memory spaces. This is particularly handy in small applications that do not require a lot of memory. This requires that data segments be located out of the way of the code. It also limits the total amount of space available. The only golden areas are in the bottom few hundred bytes that must be reserved for code (reset and interrupt vectors). Beyond that, the designer can place code/xdata segments anywhere desired, provided the segments do not unintentionally overlap.

SFR Memory Implementation

The SFR internal memory region in Core8051 gives the user access to various functions of the core such as data pointers, timers, interrupts, etc. These registers are defined in the RTL code for Core8051 and no additional logic needs to be added by the user for implementation. See the [Core8051 Datasheet](#) for more information.

External SFR (XSFR) memory is an optional feature that provides a means for connecting custom peripherals or for implementing custom features with Core8051. An alternative method for integrating custom features is to construct them so that they respond to external data (XDATA) memory addresses. The choice between creating custom memory mapped features in XDATA vs. XSFR is up to the designer. More information about designing custom peripherals for Core8051 is available in the "[Memory Regions and Memory Map](#)" section on page 9.

Core8051 Clock Configuration

To ensure proper operation of Core8051, the input clocks must be placed on low skew global buffers such as RCLK or HCLK in Actel antifuse devices, or on clock spines in Actel ProASIC^{PLUS} devices. One method of ensuring proper placement of the input clocks is to use the set_global command in the ProASIC^{PLUS} GCF constraints file used by Actel Designer software. See the "[Layout Constraints \(Example\)](#)" section on page 14 for more information on routing constraints for Designer.

Core8051 has three clock inputs: CLK, CLKPER, and CLKCPU. The most efficient way to implement the Core8051 clocking is to drive all three of these inputs with the same global signal as shown in [Figure 2](#). Combining all clocks will save FPGA resources and ensure a high-speed design with easy-to-analyze timing. However, unifying the three clocks has the drawback of disabling the Core8051's low-power modes, IDLE and STOP. The IDLE and STOP modes depend on having three independent clocks inputs.

If IDLE and/or STOP modes are required, then CLKPER and CLKCPU must be implemented as shown in [Figure 3](#) on page 6. This will make cross-clock-domain skew analysis more difficult and consume more of the FPGA's global clock resources (all three clock inputs to Core8051 must be on low-skew global buffers or clock spines).

Peripherals in the FPGA that are related to Core8051, but not part of it, should use the same clock as the CLKPER input. If unified clocking is used, then that one clock should be used for all user logic. If the gated clocking is used, then the gated clock CLKPER should be used for user logic.

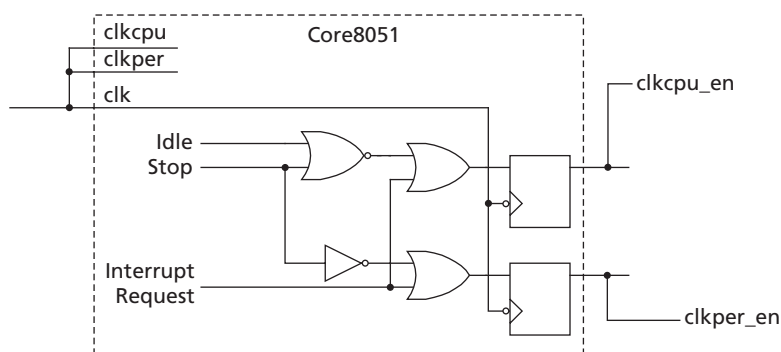


Figure 2 • Core8051 with Unified Clocks (no IDLE/STOP mode)

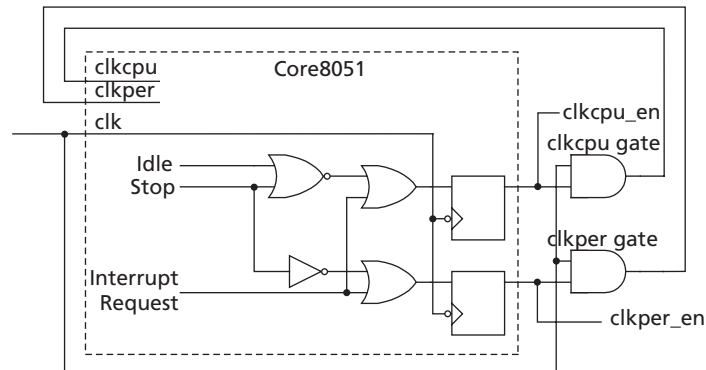


Figure 3 • Core8051 with Gated Clocks (enables IDLE/STOP modes)

Core8051 OCI Debugger Interface Implementation

Introduction

The On-Chip Instrumentation (OCI™) extension to the Actel Core8051 enhances the CPU core functionality, providing run-time control, memory and register visibility, complex breakpoint capability, and a trace history feature, all without using any resources from the core CPU.

OCI Features

The OCI module inside Core8051 has many powerful capabilities including:

- Control via four-pin IEEE-1149.1 (JTAG) port, compatible with daisy-chained multi-core systems
- Start/stop run control through debugreq/debugack signals to core
- Unlimited number of software breakpoints available via 0xA5 opcode
- Single-step by assembly instruction
- Access to all 8051 registers and memory spaces (CODE, XDATA, SFR, and DATA)
- Scalable number of hardware breakpoints (zero to four) consisting of one address/data value and one of the following modes:
 - Code memory execution
 - Code memory read or write
 - External data memory read or write
 - SFR read or write
- Ability to read and write to internal data memory
- Capability to combine two hardware breakpoints to form an address range (lower and upper bound) and masked data value
- Hardware breakpoints may be configured to break emulation, start or stop the trace, or assert a trigger out signal
- Optional break bus can be used to synchronize operation of multiple Core8051 devices connected in a multi-core configuration
- Optional AuxOut signal available to control on-chip test modes or other system-specific functions
- Optional trace history of the most recent branch points allows software reconstruction of execution flow. Memory is configurable in powers of two from two to 256 frames. Branches record both branch-from and branch-to addresses. Trace start/stop actions from the trigger also allocate a trace frame.
- Support for code memory bank switching systems. Additional bits denoting the bank number are supplied by user logic and participate in breakpoint decisions and trace.
- Presence of the OCI does not impact processor performance significantly (see the [Core8051 Datasheet](#) for detailed performance estimates).

Configuring the OCI

Actel customers who purchase either the complete Core8051 source code or Core8051 netlist have access to the OCI module. FPGA resources can be saved by omitting the OCI from the design; however, this will prevent in-system debugging. Omitting the OCI will also force the user to implement another means of programming the CODE memory (which is normally done through the debugger and OCI). Netlist customers are provided netlists with and without the OCI module included.

Note: The number of triggers and trace memory size is encoded into the netlist name as described in Figure 4.

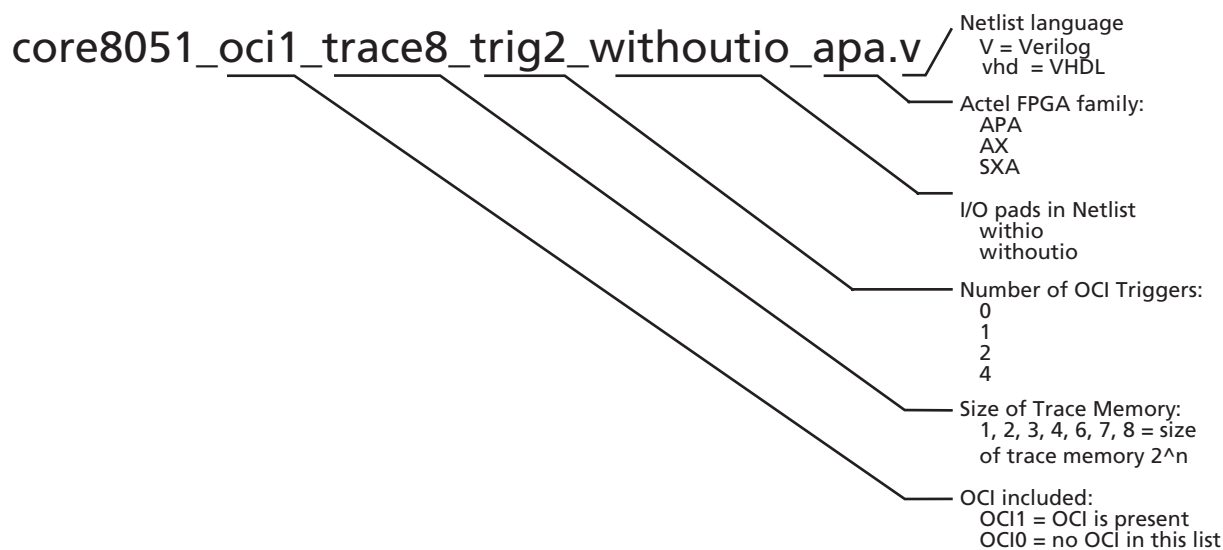


Figure 4 • Core8051 Netlist Naming Conventions

The example project for this application note was constructed using *core8051_oci1_trace0_trig0_withoutio_apa.v*. This implements the design using an OCI without hardware breakpoints (using software breakpoints only) and no trace buffer.

Customers working from the source RTL (rather than netlists) must set the generic parameters for Core8051 appropriately for their needs. These map directly to the netlist options and are defined in great detail in the [Core8051 Datasheet](#).

The generic parameters used by RTL customers to configure Core8051 are defined in the following code segment:

```
component CORE8051
  generic (
    -- set this to 1 to instantiate OCI logic
    USE_OCI          : integer := 0;
    -- set this to 1 to use ProASIC+ UJTAG macro for OCI logic
    USE_UJTAG        : integer := 0;
    -- TRACE_DEPTH
    -- no trace: Set value to 0
    -- 256 depth: Set value to 8
    TRACE_DEPTH: integer := 0;
    -- TRIG_NUM
    -- no triggers: set value to 0
    -- 1 trigger: set value to 1
    -- 2 triggers: set value to 2
    -- 4 triggers: set value to 4
    TRIG_NUM        : integer := 0;
    -- set this to 1 to make nrsto an output from here
```

```
NRSTOUT          : integer := 0;
-- set this to 1 to enable flip-flop optimizations (default is 0)
EN_FF_OPTS      : integer := 0
);
...

```

JTAG Interface

The JTAG connection for the debug interface to Core8051 OCI is dependent on the device family. For ProASIC^{PLUS} devices, the designer should connect the Core8051 JTAG ports (TCK, TDI, TDO, TMS, TRSTB) to the top-level ports in the design. These will pass through synthesis as ports/pins to the design. However, when imported into Actel Designer Software, these ports will be deleted automatically, as the software will detect the internal connection to User JTAG (UJTAG). UJTAG is a special JTAG connection internal to the FPGA that makes use of the same external JTAG pins as the FlashPro Lite interface.

In Antifuse FPGAs, UJTAG does not exist. The JTAG ports of Core8051 must be treated as user I/O pins and placed at pin locations appropriate to the board design.

If the OCI will not be used for downloading or debugging code, then the JTAG pins should be driven with constant values (within the RTL code):

```
TCLK             <= 1
TMS              <= 0
TDI              <= 0
TRSTB           <= 1
TDO              => unconnected

```

OCI Limitations and Gaining Access to FS2 Value-Added OCI Features

When using the OCI in-system debugger for Core8051, there are two types of breakpoints: software and hardware.

Software breakpoints are generated by exchanging the opcode at a particular line in memory with a breakpoint symbol **0xA5**. When Core8051 reaches one of these symbols, it halts operation and waits for the debugger to take control. The debugger tool (either Keil uV2 or FS2) monitors core operation for occurrences of these software breakpoints, and when one is encountered, the normal opcode is exchanged for the breakpoint symbol. The debugger relinquishes control to the Core8051, which will resume operation with the normal opcode that replaced the breakpoint symbol. The number of software breakpoints that can be placed in an application is limited only by the amount of CODE space allocated to the application.

Note: Software breakpoints cannot be used when the CODE memory space is placed in FLASH memory. This is because the FLASH memory cannot be easily rewritten to swap opcodes.

Hardware breakpoints are generated by special hardware in the OCI that monitors the address bus and halts Core8051 operation when a particular CODE address is fetched. In the Core8051 OCI block, hardware breakpoints are implemented using "triggers" (a set of registers that can cause a breakpoint or tracepoint by matching either an address or data bus pattern). Hardware breakpoints have the advantage of working well no matter what type of memory is used to store CODE. The disadvantage is that debugger support becomes more complicated, and only a limited number of hardware breakpoints are available (a maximum of four).

The number of breakpoints and trace memory size available is based upon the type of FS2 license. With the standard FS2 license, only a single breakpoint with no trace functionality is available. For users to gain access to multiple hardware breakpoints with trace memory, an enhanced license from FS2 is required. Contact FS2 directly at www.fs2.com to purchase an upgrade. The maximum for all users is four hardware breakpoints and 256 lines of trace memory.

Core8051 on ProASIC^{PLUS} and APA300 Starter Kit

The example design (shown in Figure 5 on page 11) uses a top-level netlist to instantiate Core8051, the required memories, and connects basic I/O for clock, data, and JTAG (OCI debugger) functionality. Also in the top level netlist is an example of a custom peripheral that responds to XDATA memory access and a multiplexer (MUX) for combining XDATA RAM, CODE RAM, and the custom peripheral into a single data input port on Core8051.

Memory Regions and Memory Map

In addition to CODE and XDATA RAM, this example has an 8-bit peripheral that has been added to Core8051's memory space. This peripheral is accessed via a readable and writable memory address in the Core8051's XDATA memory space. The example peripheral is a simple bank of eight flip-flops. While simplistic in function, it illustrates the mechanism for connecting a peripheral module to Core8051. The address decoding and data path MUXing used to implement this will be the same for more complex peripherals that may be included in a customer design. For the example design, the memory regions are defined in Table 2.

Table 2 • Memory Regions Defined in Example Design

CODE RAM	RAM for holding the application code. This memory is written to by the PC-based debugger tools through the JTAG interface and Core8051. Opcodes are fetched from this memory during normal Core8051 operation. CODE space is 64 k (16-bit address) in size, however, in this design only 4 k is implemented (12-bit address) from 0x0000 to 0x0FFF.
XDATA RAM	General-purpose data RAM for holding large data structures or variables as needed by the application code. XDATA space is 64 k in size; however, in this design only 2 k (11-bit addresses) is implemented from 0x0000 to 0x7ff.
XDATA Peripheral	At XDATA address 0xFF00 a single 8-bit register has been implemented. This address was chosen because it is out of the range of the XDATA RAM. This register is writable and readable. The development board LEDs are driven by this register.
SFR registers	SFR memory region is 128 bytes in size and is contained within Core8051. There are many unused addresses in this region. It is possible to implement customer peripherals so that they respond to these SFR addresses, however, this has not been done in the example.
Data RAM	A minimum of 128-bytes and a maximum of 256-bytes of Data RAM are required by the Core8051. This memory space holds the Core8051 Registers R0 through R7 as well as the local variable and stack space for most applications.

Choosing XDATA Addresses for Peripherals

It is preferable to place XDATA memory mapped peripherals into high address ranges. Most C compilers will, by default, place blocks of application data at or near address 0x0000. Placing registers or blocks of registers that have a specific hardware function in the higher address ranges (such as starting at 0xff00) will minimize the possibility of conflict. The mapping report should always be checked after compilation to make sure that data segments in XDATA memory do not conflict with hardware peripherals and that they map to the XDATA RAM in hardware.

Choosing XSFR Addresses for Peripherals

The ways in which SFR memory is used by a standard Core8051 system is limited to the registers listed below. All other SFR memory addresses may be decoded and used to implement custom peripherals. If an address is chosen that is already implemented inside Core8051, then the external register will be ignored. See Table 3 on page 10.

Table 3 • Predefined SFR Registers

Register	Address	Description
p0	80h	Port 0
Sp	81h	Stack Pointer
dpl	82h	Data Pointer Low 0
dph	83h	Data Pointer High 0
dpl1	84h	Dual Data Pointer Low
dph1	85h	Dual Data Pointer High
pcon	87h	Power Control
tcon	88h	Timer/Counter Control
tmod	89h	Timer Mode Control
tl0	8Ah	Timer 0, low byte
tl1	8Bh	Timer 1, high byte
th0	8Ch	Timer 0, low byte
th1	8Dh	Timer 1, high byte
ckcon	8Eh	Clock Control
p1	90h	Port 1
dps	92h	Data Pointer Select Register
scon	98h	Serial Port 0, Control Register
sbuf	99h	Serial Port 0, Data Buffer
p2	A0h	Port 2
ien0	A8h	Interrupt Enable Register
ip0	A9h	Interrupt Enable Register
p3	B0h	Port 3
ien1	B8h	Interrupt Enable Register
ip1	B9h	Interrupt Enable Register
psw	D0h	Program Status Word

Building the XDATA/CODE DataPath

The XDATA and CODE memory spaces share address and data ports on Core8051. However, they each have their own control signals. CODE uses the signal `dbgmempswr` as a write enable, and `memp srd` as a read enable. XDATA uses the signal `memwr` as a write enable and `memrd` as a read enable.

Outgoing data and address is routed to both memories and any additional peripherals that exist in the design. The returning data must be multiplexed to ensure that the correct memory is driving the Core8051 data input ports. To avoid decoding the address twice (once for the write logic and once for the multiplexer) the example design uses a read select signal that is driven from the peripheral to the multiplexer. This way the address is only decoded inside the peripheral (Figure 5 on page 11).

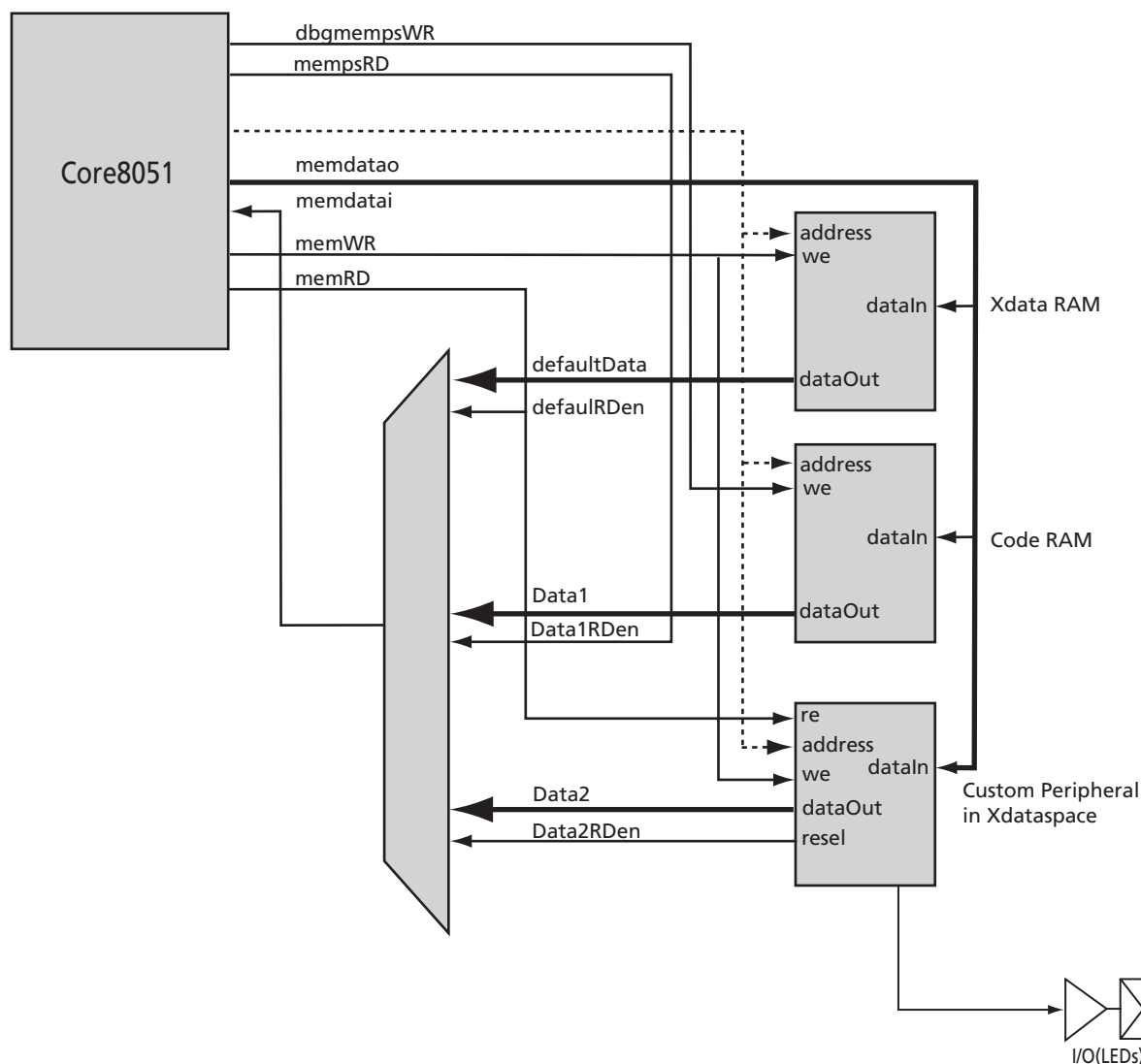


Figure 5 • Details of CODE/XDATA Bus

Description of Custom Memory Mapped Peripheral Implementation

The example memory mapped peripheral is attached to the Core8051 XDATA/CODE bus. It decodes the address bus and *memwr* and *memrd* for control, and also has an 8-bit *datao* input. The outputs from this module are the 8-bit read *datai*, *rdsel*, and *memacki* (*memacki* is the wait state control signal). This module is designed to add one wait state to both read and write operations. This wait state does not affect normal XDATA RAM or CODE RAM accesses, which are still single cycle operations. By asserting *memacki* at the appropriate time in either a read or write cycle an extra (or more if desired) clock cycle is added to the bus operation. During this time Core8051 is effectively halted, so care must be taken to guarantee that these bus wait states are used correctly and do not remain in an indefinite halt state. The *rdsel* signal is used to communicate to the data path multiplexer on a read operation from this peripheral register. When *rdsel* is asserted, the *datai* from this module should be propagated to Core8051.

The waveforms in [Figure 6 on page 12](#) illustrate the logical operation and timing of the module in the example design.

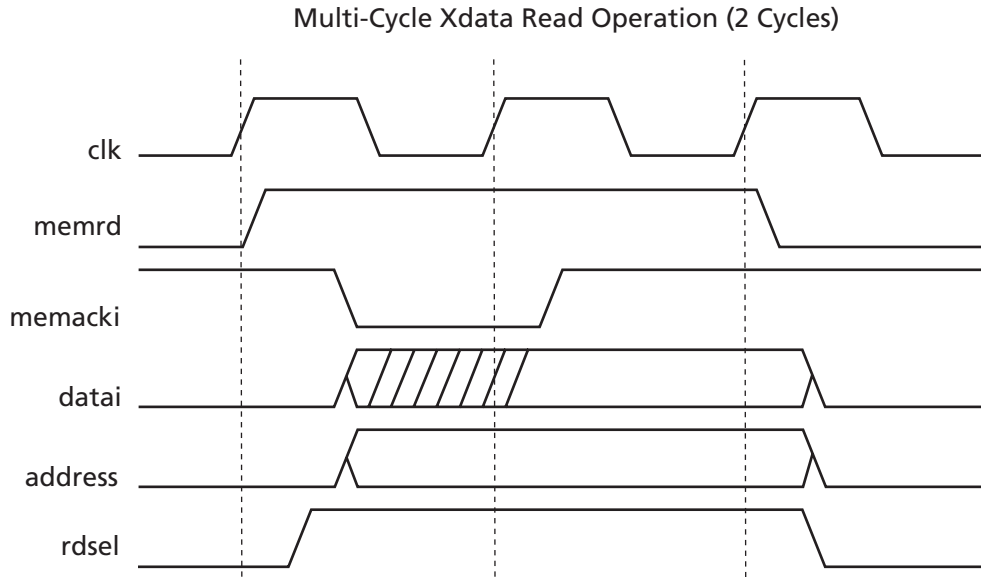


Figure 6 • Multi-Cycle XDATA Read Operation (2 Cycles)

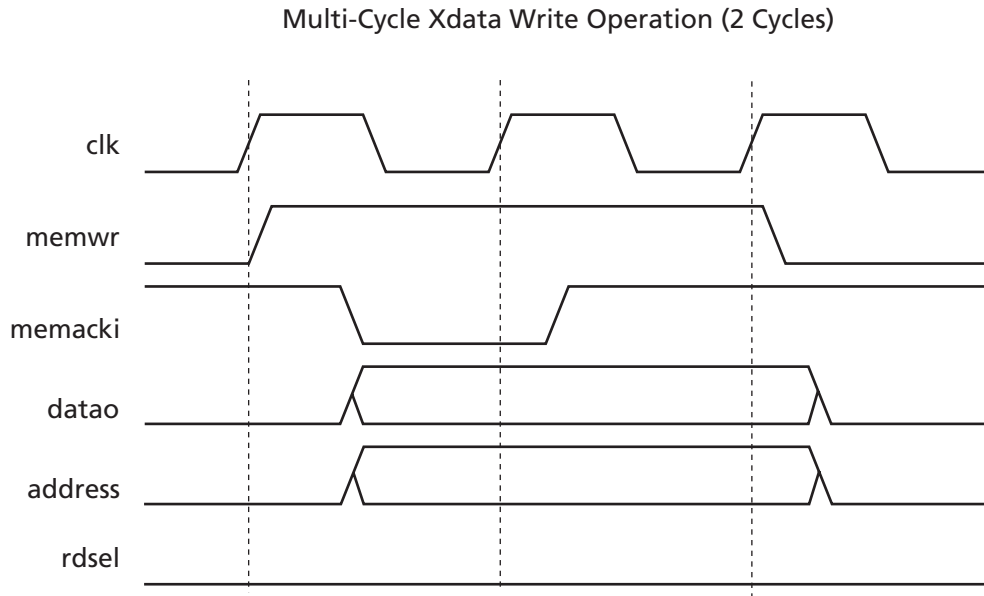


Figure 7 • Multi-Cycle XDATA Write Operation (2 Cycles)

Description of Data Path MUX

The data path MUX is a relatively simple combinational function. Data from the CODE and XDATA memories as well as the XDATA peripheral are qualified by enable signals. These enable signals are prioritized in case of conflict. In the example design, CODE reads have the highest priority followed by the XDATA peripheral register. The XDATA RAM is the lowest priority, default data source.

```

module DataMux ( DefaultData, DefaultDataAcki, Data1, Data1En,
Data1Acki, Data2, Data2En, Data2Acki, DataI, acki );
input [7:0] DefaultData;
input      DefaultDataAcki;
input [7:0] Data1;
input      Data1En;
input      Data1Acki;
input [7:0] Data2;
input      Data2En;
input      Data2Acki;

output [7:0] DataI;
output acki;

assign acki =
    (Data1En)? Data1Acki :
    (Data2En)? Data2Acki :
    DefaultDataAcki;

assign DataI =
    (Data1En)? Data1 :
    (Data2En)? Data2 :
    DefaultData;

endmodule // DataMux

```

The *memacki* signals are also multiplexed in the same way as the data. This way each of the modules or memories on this bus can independently control the number of cycles needed to perform read and write operations.

Synthesis Constraints (Example)

Proper synthesis constraints must be in place to ensure correct compilation of the design. These constraints are synthesis tool specific. The example was synthesized using Synplicity® Version 7.5.1 with the following constraints (from *eval_bd.sdc*):

```

# Constrain target clock frequency.
define_clock -name {clk8051} -freq 40.000 -clockgroup default_clkgroup

# syn_maxfan is used to prevent Synplicity from inserting buffers or
# otherwise duplicating driver. These nets will be distributed on
# global clock resources (specified in Designer constraints).
# Allowing them to be replicated will cause problems both in Designer
# and in Synplicity.

define_attribute {n:clk8051} syn_maxfan {2000}
define_attribute {n:CORE8051_inst.UDRCK} syn_maxfan {2000}

```

Layout Constraints (Example)

Actel Designer requires layout constraints to ensure that the proper nets are routed to the correct I/Os for the evaluation board. To guarantee that timing is met, key clock and reset nets should be constrained as globals. The following excerpt is from *eval_bd.gcf*, the Designer constraints file for this project.

```
// ----- setup global signals -----  
  
// -----  
// source clock for Core8051 (and OCI,Trace Mem)  
// -----  
set_global clk8051;  
  
// -----  
// use spine for UDRCK (clock for debugger/JTAG clock domain)  
// -----  
use_global CORE8051_inst/UDRCK;  
  
// -----  
// Core8051 globally buffered reset (sync to neg edge of clk8051)  
// -----  
  
set_global nrsto;  
  
// -----  
// reset mostly used in OCI  
// -----  
set_global reset_c;  
  
  
// ----- I/O constraints (pinouts)  
// Board clock input  
set_io "24" "boardclk";  
// Board reset input  
set_io "128" "reset";  
// board LEDs  
set_io "87" "led1";  
set_io "90" "led2";  
set_io "91" "led3";  
set_io "92" "led4";  
set_io "93" "led5";  
set_io "94" "led6";  
set_io "95" "led7";  
set_io "96" "led8";  
  
// board switches  
set_io "55" "pb1";  
set_io "63" "pb2";  
set_io "69" "pb3";  
set_io "79" "pb4";
```

Simulation

Simulation of microprocessor systems has challenges that are not present in most FPGA designs. The primary complication is that the application code must be loaded into RAM before beginning operation. In a real system this is done by a debugger or by permanently storing the code in an off-chip ROM or FLASH. In this design, the code is stored in on-chip RAM. To load code into the simulation model for an on-chip RAM, follow these steps:

1. Identify the instance names of the RAM256X9 primitives that are to be used as code memory. Be sure to note the order of the primitives. In the example design, the RAM cells for code memory are named *codeRam_0/M0* through *codeRAM_0/M15*.
2. Export a netlist from designer for simulation purposes.
3. Create a simple C program and compile a HEX output file for it. It is important that this be kept extremely simple. Running large amounts of code in simulation will be very slow. RTL simulation of a Core8051 system is best restricted to individual specific features. Full system verification is best accomplished by testing the actual hardware.
4. The HEX output file from the C compiler will need to be translated into 9-bit binary data (one parity bit, followed by the 8-bit data). An Internet search for *intel hex format* will find websites with information that can assist in decoding the hex file format.
5. The binary representation of the compiled code must then be broken into separate 256-line files. These will be loaded into individual RAM256X9 memory primitives. For the example, only a single RAM256X9 needs to be initialized because the program is less than 256 bytes in length. *CodeRAM_0/M0* is the only RAM that will be initialized. The file that will be loaded into it is *codeM0.dat*. Use the .dat extension on these files.
6. Edit the netlist. For each RAM256X9 primitive initialized with code, a *defparam* (in verilog) or *generic map* (in VHDL) parameter must be set that specifies the binary. For the example design, adding the following line to the top level testbench will be sufficient for loading the first 256 bytes of CODE memory (which is more than enough memory to hold the test program).

```
defparam tb_vlog.APA300_design.codeRam_0.M0.MEMORYFILE = "codeM0.dat";
```

7. After these steps the simulation will automatically load the .dat file into RAM and run as any standard simulation.

For more information on initializing ProASIC^{PLUS} RAM Models, see the Actel application note [Preloading of ProASIC^{PLUS} RAM Models in Simulation Using Libero IDE](#).

ProASIC^{PLUS} Board Settings

The ProASIC^{PLUS} Starter Kit Board and Evaluation Board have several jumpers and connectors that must be set properly for the example design to work. In addition to setting the jumpers as defined in [Table 4](#), 9 V power must be supplied and a FlashPro Lite connected to P1.

Table 4 • ProASIC^{PLUS} Evaluation Board Jumpers

Jumper Instance Name	Usage for Example Design	Description
JP1	Jumper Pins 1&2 (leave pin 3 unconnected)	Selects Voltage for VDDP
JP2	Install Jumper	Use GND as AGND
JP3	Install Jumper	Use 2.5 V for AVDD
JP4	Install Jumper	Connect 40 Mhz Oscillator to FPGA clock pin
JP5 through JP8	Install Jumper	Connects pushbutton to FPGA inputs.
JP9 through JP16	Install Jumper	Connects LEDs to FPGA outputs.
JP17	Install Jumper	Connects reset pushbutton to FPGA input.

File and Directory Descriptions

The data file for this application note (APA300_Core8051.zip) contains the source files for the project described above. Netlists or RTL for the Core8051 IP module are not included. The Designer project file has also not been included. To gain access to Core8051 IP, contact your Actel sales representative. Following is a list of the files that are included with the application note data file:

APA300_Core8051/actgen_RAM/

- contains verilog models for RAMs and ACTgen project

APA300_Core8051/code/

- contains Keil uV2 project and C code

APA300_Core8051/Core8051_netlist/

- <empty – contact Actel Sales Representative>

APA300_Core8051/designer/

- contains Design constraints file and STPL file for device programming
<Designer project file has been excluded>

APA300_Core8051/RTL/

- contains Top level verilog netlist and verilog source for xdata peripheral and data MUX

APA300_Core8051/synth/

- contains Synplicity project and constraints files

APA300_Core8051/sim/

- modelsim simulation project (using tb_use_core8051.v as testbench)

APA300_Core8051/sim/code/

- accelerated version of Keil uV2 project and C code (to make LEDs count at high speed for simulation purposes the wait loop has been removed)

Example Design FPGA Resource Utilization

Table 5 • Core8051 APA300 Demonstration Design Utilization and Performance

Family	Cells or Tiles			Utilization		Performance
	Sequential	Combinatorial	Mem	Device	Total	
ProASIC ^{PLUS}	744	5128	25	APA300-PQ208	62%	21 MHz

Related Documents

Application Notes

Preloading of ProASIC^{PLUS} RAM Models in Simulation Using Libero IDE

http://www.actel.com/documents/APA_RAMModels_Libero_AN.pdf

Datasheets

Core8051 Datasheet

http://www.actel.com/ipdocs/Core8051_DS.pdf

Actel and the Actel logo are registered trademarks of Actel Corporation.
All other trademarks are the property of their owners.



www.actel.com

Actel Corporation

2061 Stierlin Court
Mountain View, CA
94043-4655 USA

Phone 650.318.4200
Fax 650.318.4600

Actel Europe Ltd.

Dunlop House, Riverside Way
Camberley, Surrey GU15 3YL
United Kingdom

Phone +44 (0) 1276 401 450
Fax +44 (0) 1276 401 490

Actel Japan

www.jp.actel.com

EXOS Ebisu Bldg. 4F
1-24-14 Ebisu Shibuya-ku
Tokyo 150 Japan

Phone +81.03.3445.7671
Fax +81.03.3445.7668

Actel Hong Kong

www.actel.com.cn

39th Floor, One Pacific Place
88 Queensway, Admiralty
Hong Kong

Phone 852.227.35712
Fax 852.227.35999