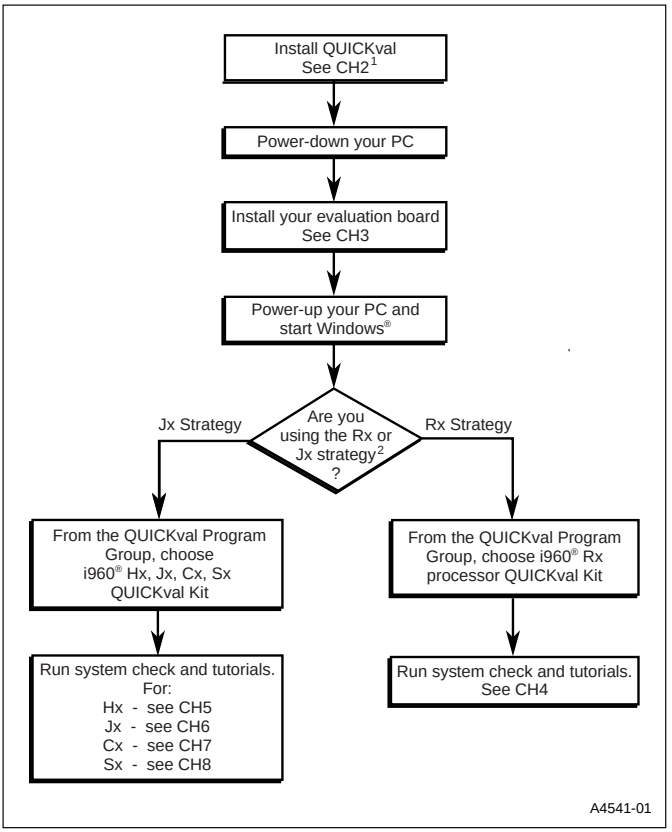
80960 QUICK*val*
Quick Reference Card

Order Number: 649788-003

Where Do I Start?

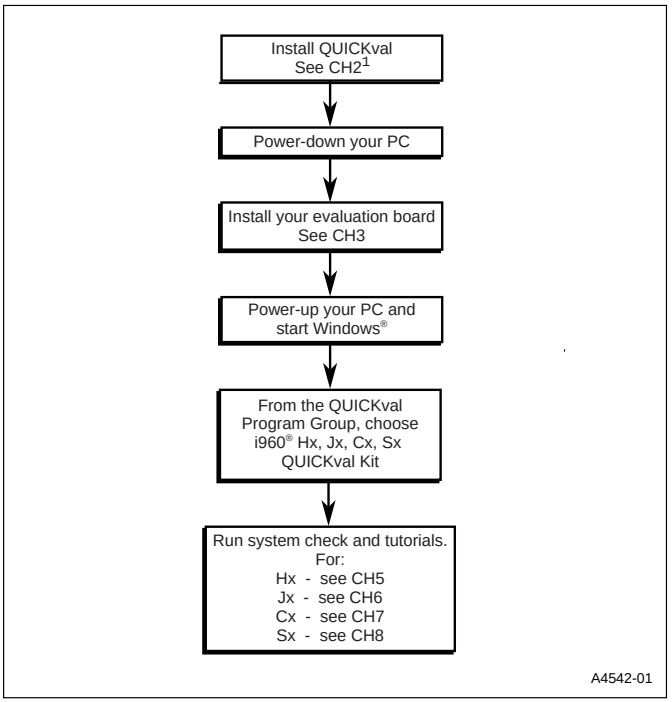
Figure 1. If You Have an i960® Rx Processor QUICKval Kit



1 All chapter references refer to "*Getting Started with the 80960 QUICKval Kit*."

2 See "For i960 Rx Processor Developers: Which Strategy Do You Use?" on Page 4 for more information.

Figure 2. If You Have an i960 Hx, Jx, Cx, Sx Processor QUICKval Kit



[1] All chapter references refer to "*Getting Started with the 80960 QUICKval Kit*."

For i960® Rx Processor Developers: Which Strategy Should You Use?

When writing software for the i960 RP/RD processors, you have two choices for the architecture setting that you use when generating code. You can specify the Rx architecture ("Rx Strategy"), or you can specify the i960 JF architecture ("Jx Strategy"). Use these questions to help you decide which of the two development paths you should follow:

1. How important is forward-compatibility with future I960 processors? Use the Rx Strategy if you wish to minimize the effort involved in moving to future i960 Rx processors.
2. Will you be writing your code from scratch? When writing new applications, follow the Rx Strategy when possible. Tests have shown that there is seldom a significant performance or code size penalty.
3. How important is backward-compatibility with other I960 core processors (e.g., Kx, Cx, Jx)? If you have legacy code that you wish to use with the I960 Rx processors, you may wish to use the Jx Strategy. This gives the most flexibility in terms of available instructions and addressing modes.
4. How much low-level processor access do you need? If you need access to low-level processor resources beyond that provided in the updated assembler pseudo-instructions, you must use the Jx Strategy.

Writing Assembly with the Rx Strategy

- Use -ARx (e.g., -ARP) switch to get CTOOLS Rx Strategy enhancements.
- If migrating code written for other I960 core processors, use the xlate960 utility as a starting point.

xlate960: generates i960 Rx Strategy compatible code to replace instructions and addressing modes that appear in the i960 JF processor only.

- If you need to use some i960 JF processor specific features not supported in the Rx Strategy, use the new assembler pseudo-instructions whenever possible. The assembler pseudo instructions provide an architecture-independent method of performing some of the more common low-level processing operations.

Benefit: The pseudo-instructions should not require modification when the source code is re-assembled for future i960 Rx processors.

Writing Assembly without the Rx Strategy

- Use -AJF switch to write code that is designed for the I960 JF-based Rx (e.g., RP, RD) processors only. You can still simplify future migration efforts by staying within the boundaries of the Rx Strategy whenever possible.
- For low-level processor functionality, you may wish to use the new assembler pseudo-instructions. This eases future migration without excluding use of Jx-Specific constructs.

Please refer to the *i960 Processor Assembler User's Guide* for more information.

QUICKreference Contact Guide

Important Phone Numbers:

Technical Support Group

1-800-628-8686

Intel Literature Center

1-800-548-4725

FaxBACK

1-800-628-2283 or 916-356-3105

Document #-_____

Press ‘2’ for Development Tools

Press ‘3’ for Catalog

Intel Bulletin Board

1-916-356-3600

Important E-Mail Addresses:

80960 Technical Support Group

960tools@intel.com

Important Internet Sites:

Embedded Design Products

http://developer.intel.com/design/product.htm

Technical Support Page

http://developer.intel.com/design/i960/swsup/

i960 Processor Software Tools Patches Page

http://developer.intel.com/design/i960/patches/

Electronic Benchmark Facility

http://developer.intel.com/design/i960/testcntr/

QUICKval Example Programs

Description	Source Files Needed
Hello World: Uses simple printf statement to verify system integrity.	*hello.c* : source file *system.c* : system file
Memory Test: Used for system verification of external memory. The programs perform byte, short, or word writes to external memory, and then they check the addresses written for correctness.	*memtst8.c* : 8 bit memory test *memtst16.c* : 16 bit memory test *memtst32.c* : 32 bit memory test *system.c* : system file
Data Cache: Uses the Minimum Edit Distance Algorithm to demonstrate the effectiveness of the on-chip data cache. This example also shows how to enable and disable the data cache; furthermore, it demonstrates how to configure an area of memory for caching.	*dcache.c* : source file *system.c* : system file
Instruction Cache: Uses simple loop to demonstrate how to enable and disable the instruction cache. It also highlights the performance advantage obtained when using the on-chip instruction cache.	*loop.c* : source file *system.c* : system file
External Interrupts: Shows how to configure the Cyclone board timers to trigger hardware interrupts. This is also an example of using interrupt handlers and placing the handlers in the interrupt table.	*cyint.c* : source file *asm_fns.s* : interrupt handler-SX *int_proc.s* : interrupt handler-all processors but SX *t85c36.c* : eval board timer file *system.c* : system file
Internal Interrupts: Simple timer example that is used to show how to overlay the memory mapped registers with a structure to program the on-chip timers. It also includes routines to instruct you on how to set up interrupt routines using the timers.	*timrcntr.c* : source file *timers.c* : on-chip timer file *system.c* : system file
Halt Mode: This program shows how to make the processor enter halt mode, a power saving state that reduces energy consumption and heat dissipation as it waits to continue code execution. The example uses the on-chip timers to trigger interrupts and “wake” the processor.	*halt.c* : source file *incremen.s* : interrupt handler *system.c* : system file
Fault Handling: Shows the steps taken in setting up the fault handling procedures in the fault and system procedure tables. The faults shown are: arithmetic, constraint, operation, protection, parallel, & type.	*fault.c* : source file *flt_proc.c* : fault procedures *asm_flt.s* : assembly functions to help generate faults *system.c* : system file

QUICKval Example Programs (continued)

Processors Supported	**Compile Line** **<arch> = RD, RP, HD, JF, CF, or SA**
ALL	gcc960 -Fcoff -A<arch> -c hello.c system.c
ALL	gcc960 -Fcoff -A<arch> -c memtst*.c system.c *NOTE*: * refers to 8, 16, or 32.
Rx, Hx, Jx, & Cx	gcc960 -Fcoff -A<arch> -c dcache.c system.c
Rx, Hx, Jx, & Cx	gcc960 -Fcoff -A<arch> -c loop.c system.c
Hx, Jx, Sx, Cx	Sx: gcc960 -Fcoff -ASA -c cyint.c asm_fns.s t85c36.c system.c Other: gcc960 -Fcoff -A<arch> -c cyint.c int_proc.s t85c36.c system.c
Hx & Jx	gcc960 -Fcoff -A<arch> -c timrcntr.c timers.c system.c
Jx	gcc960 -Fcoff -A<arch> -c halt.c incremen.s system.c
Hx, Jx, & Cx	gcc960 -Fcoff -A<arch> -c fault.c flt_proc.c asm_flt.s system.c

QUICKval Example Programs (continued)

Register Cache: Demonstrates the use of the on-chip register cache in reducing the interrupt latency for high priority interrupts.	*reg_int.c* : source file *low_int.s* : interrupt handler for low priority *high_int.s*: interrupt handler for high priority *system.c* : system file
CheckSum: Uses typical checksum routine to show how to add benchmarking routines into source code. It is then used to show you the performance advantage of optimizing compilers and two-pass compilations.	*chksum.c* : source file *system.c* : system file
DMA (i960 Cx): Provides an example of programming the DMA controller of the 80960 CX microprocessor. This example is setup for block mode chaining transfer.	*dma.c*: source file *int_rout.c* : DMA interrupt handling routine *sdma.s* : will configure DMA channel 0 and provide chained linked buffers. *system.c* : system file
DMA (i960 Rx) Tutorial: Demonstrates how to set-up the DMA controller, the Primary Address Translation Unit (ATU), the Secondary ATU, and the PCI-To-PCI Bridge Unit.	*rpdma.c*: source file
Messaging Unit Tutorial: Demonstrates the messaging unit of the i960 Rx processor	*hostcode.c*: source file *rp_code.c*: source file
Cave Tutorial: Uses a tic tac toe game to show how to reduce target memory requirements. The text sections of compressed and uncompressed tic tac toe executables are compared. Additionally, this example demonstrates how to specify functions for compression.	*ttt.c*: source file
Profiling Lab: Teaches you how to use some of CTOOLS advanced profiling features.	*chksum.c:* Source file
Self-Contained Profile Tutorial: Teaches you how to create a self-contained profile that captures the program structure and associates it with the program counters from a raw profile. When the source program changes, the global decision making step interpolates or stretches the counters in the self-contained profile to fit the changed program.	*quick.c:* Source file

QUICKval Example Programs (continued)

Hx & Jx	gcc960 -Fcoff -A<arch> -c reg_int.c low_int.s high_int.s system.c
ALL	gcc960 -Fcoff -A<arch> -O* -c chksum.c system.c *NOTE*: * refers to [0-4] depending on the level of static optimization.
Cx ONLY	gcc960 -Fcoff -ACF -c dma.c int_rout.c sdma.s system.c
Rx ONLY	NA
Rx ONLY	NA
ALL	NA
ALL	NA
ALL	NA

QUICKval Example Programs (continued)

Incremental Profiling Tutorial: Teaches you how to profile a program in pieces and then re-combine them later, a useful methodology when the target execution environment is memory limited.	*fault.c flt_proc.c* *asm_flt.s system.c:* Source files
Local Optimizations: Shows how to use the C compiler with high levels of static optimization for improved runtime performance.	*chksum.c system.c:* Source files
Global Optimizations: Shows how to use program-wide optimizations of the C compiler for increased performance.	*chksum.c system.c:* Source files
C++ Local Optimizations: Shows how to use the C++ compiler with high levels of static optimization for improved runtime performance.	*optimize.cpp:* Source file
C++ Global Optimizations: Shows how to use program-wide optimizations of the C++ compiler for increased performance.	*optimize.cpp:* Source file
C++ Virtual Function Optimizations: In many situations, a call to a virtual function can be replaced by a direct call to a member function, and, if possible, it may be inlined at the call site. This improves the runtime performance of the code.	*optimize.cpp:* Source file
C++ Cave Tutorial: Uses a C++ program to show how to reduce target memory requirements. The text sections of compressed and uncompressed C++ executables are compared. Additionally, this example demonstrates how to specify functions for compression.	*cavecpp.cpp:* Source file
xlate960 Assembly Language Converter Tutorial: Shows you how you can use xlate960 to convert assembly language code written for one i960 processor family member to that of another.	*xlt.s:* Source file
i960 Processor Assembler Pseudo-Instruction Support Tutorial: A tutorial that shows you how to use the new pseudo-ops that have been added to the assembler.	*pseudop.c:* Source file
Linker Directive Language: A hyperlinked manual that describes the linker command options.	

QUICKval Example Programs (continued)

Hx, Jx, & Cx	NA
ALL	NA
ALL	NA
ALL	NA
ALL	NA
ALL	NA
ALL	NA
Rx, Jx	NA
Rx, Hx, Jx, & Cx	NA
ALL	NA

QUICKval Example Programs (continued)

Linker Consumption Tutorial: Shows the ability of the linker, gld960, to consume b.out-format, COFF, or ELF object files and libraries in any combination.	*cyint.c int_proc.s t85c36.c system.c:* Source files
Debugging with gdb960 Tutorial: Uses the Go Fish card game to teach a few useful debugger commands.	*fish.c* : source file *system.c* : system file
ELF/DWARF Debugging Format Tutorial: Demonstrates that at the highest level of module-local optimization, it is possible to set a breakpoint on an in-line function.	*swap.c:* Source file
C++ DWARF-2 Debugging Format Tutorial: Demonstrates that at the highest level of module-local optimization, it is possible to debug a C++ application.	*cppdwarf.cpp:* Source file
Retargeting MON960: The Retargeting MON960 chapter is hyperlinked for your convenience.	
Writing Flash Tutorial: Demonstrates how to update the version of MON960 on your evaluation board.	
i960 Rx Processor Initialization Code: Shows the Memory Controller, System Init, and Hardware Init Code.	
80960 Family Benchmark: Used to compare your processor's performance with other i960 family members.	*chksum.c system.c:* Source files
Remote Evaluation Facility: Guides you through the use of this new benchmarking facility on the World-Wide Web.	

QUICKval Example Programs (continued)

Hx, Jx, & Cx	NA
ALL	gcc960 -Fcoff -A<arch> -c fish.c system.c
ALL	NA
ALL	NA
ALL	NA
ALL	NA
RP	NA
Hx, Jx, Cx, Sx	NA
Hx, Jx, Cx, Sx	NA

i960 Cx, Jx, Hx Processor Features Summary

Feature	80960Cx	80960Jx	80960Hx
Core	Superscalar (maximum 3 inst/clk)	Scalar,clock doubled, clock tripled	Superscalar (max 3 inst/clk), clock doubled, clock tripled
External Bus	32-bit demulti-plexed address and data	32-bit multiplexed address/data	32-bit demulti-plexed address and data, parity on data
Instruction Cache	CA: 1 Kbyte, 2-way CF: 4 Kbyte, 2-way	JA/JF/JD: 4 Kbytes, 2-way JT: 16 Kbytes, 2-way	16 Kbytes, 4-way
Data Cache	CA: None CF: 1Kbyte, direct map, write-through	JA/JF/JD: 2 Kbytes, direct map, write-through JT: 4 Kbytes direct map, write-through	8 Kbytes, 4-way, write-through
Data RAM	1 Kbyte, mapped from 000H to 3FFH	1 Kbyte, mapped from 000H to 3FFH	2 Kbytes, mapped from 000H to 7FFH
Register Cache	5 frames, programmable to 15 frames (more than 5 used Data RAM)	8 frames	5 frames, programmable to 15 frames (more than 5 uses Data RAM)
Memory-mapped Registers	No	Yes	Yes
Direct Memory Access (DMA) Controller	Yes	No	No
Interrupt Controller	Yes	Yes	Yes
Guarded Memory Unit	No	No	Yes
Timers	None	Two	Two
Power Supply	5V	5V, 3.3V, or 3.3V with 5V tolerant	3.3V, 5V tolerant
JTAG	No	Yes	Yes

i960 Rx Processor Features Summary

Frequencies Supported	• 33 MHz, 3.3 Volt Version (80960RP 33/3.3) • 66 MHz, 3.3 Volt Version (80960RD 66/3.3) - Clock Doubled 80960JF Core
Compatibility	• Complies with PCI Local Bus Specification Revision 2.1
High Performance 80960Jx Core	• Sustained One Instruction/Clock Execution • 4 Kbyte Two-Way Set-Associative Instruction Cache • 2 Kbyte Direct-Mapped Data Cache • Sixteen 32-Bit Global Registers • Sixteen 32-Bit Local Registers • Programmable Bus Widths: 8-, 16-, 32-Bit • 1 Kbyte Internal Data RAM • Local Register Cache (Eight Available Stack Frames) • Two 32-Bit On-Chip Timer Units
PCI-to-PCI Bridge Unit	• Primary and Secondary PCI Interfaces • Two 64-Byte Posting Buffers • Delayed and Posted Transaction Support • Forwards Memory, I/O, Configuration Commands from PCI Bus to PCI Bus
Two Address Translation Units	• Connects Local Bus to PCI Buses • Inbound/Outbound Address Translation Support • Direct Outbound Addressing Support
Messaging Unit	• Four Message Registers • Two Doorbell Registers • Four Circular Queues • 1004 Index Registers
Memory Controller	• 256 Mbytes of 32- or 36-Bit DRAM • Interleaved or Non-Interleaved DRAM • Fast Page-Mode DRAM Support • Extended Data Out and Burst • Extended Data Out DRAM Support • Two Independent Banks for SRAM / ROM / Flash (16 Mbytes/Bank; 8- or 32-Bit)

i960 Rx Processor Features Summary (continued)

DMA Controller	• Three Independent Channels • PCI Memory Controller Interface • 32-Bit Local Bus Addressing • 64-Bit PCI Bus Addressing • Independent Interface to Primary and Secondary PCI Buses • 132Mbyte/sec Burst Transfers to PCI and Local Buses • Direct Addressing to and from PCI Buses • Unaligned Transfers Supported in Hardware • Two Channels Dedicated to Primary PCI Bus • One Channel Dedicated to Secondary PCI Bus
I/O APIC Bus Interface Unit	• Multiprocessor Interrupt Management for Intel Architecture CPUs (Pentium® and Pentium Pro Processors) • Dynamic Interrupt Distribution • Multiple I/O Subsystem Support
I2C Bus Interface Unit	• Serial Bus • Master/Slave Capabilities • System Management Functions
Secondary PCI Arbitration Unit	• Supports Six Secondary PCI Devices • Multi-priority Arbitration Algorithm • External Arbitration Support Mode
Private PCI Device Support	• SuperBGA* Package • 352Ball-Grid Array (HL-PBGA)