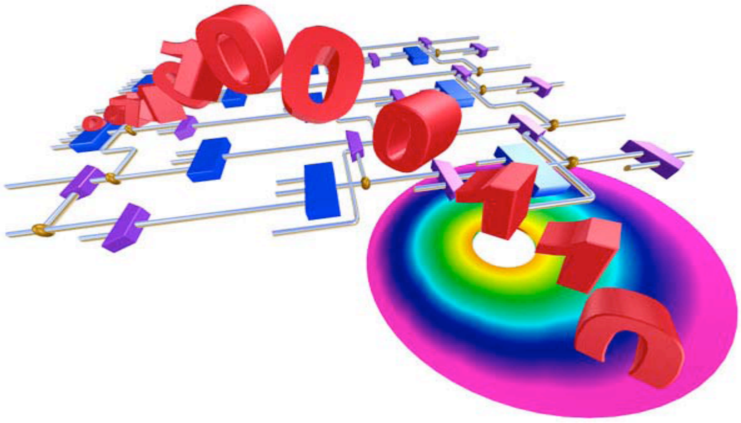
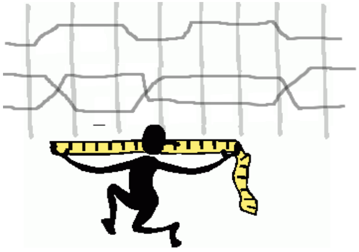
Embedded Systems
Interrupt and Booting



Peter Thorwartl

The Designer’s Challenge Interrupt

Reducing interrupt inclusion design time



Accounting for system interrupt latency

Objectives

After completing this module, you will be able to:

- Describe the interrupt structure of the MicroBlaze™ and PowerPC® processors
- Write an interrupt handler for the targeted processor
- Register the interrupt handler/Interrupt Service Routine (ISR)
- Use an interrupt controller to accommodate multiple interrupts
- Apply proper programming techniques to reduce interrupt latency

Exceptions

- Exception handling is a programming language construct or computer hardware mechanism designed to handle the occurrence of some condition that changes the normal flow of execution
  - Related to the current program flow
  - Typically the result of unexpected error conditions (such as a bus error)
  - Result of illegal operations (guarded memory access)
  - Some exceptions can be programmed to occur (FIT, PIT)
  - A software routine could not execute properly (divide by 0)
  - Handler able to resume execution at the original location

Hardware Interrupts

- An interrupt is an asynchronous signal from hardware indicating the need for attention, or a synchronous event in software indicating the need for a change in execution
  - Embedded processor peripheral (FIT, PIT, for example)
  - External bus peripheral (Uart, EMAC, for example)
  - External interrupts enter via hardware pin(s)
  - Multiple hardware interrupts can be OR’d or utilize an external interrupt controller

Exceptions versus Interrupts

- Exceptions are unexpected events typically signaled from within the processor core
  - •Exception handlers are used to service exceptions
  - •Illegal instructions, OPCODEs, divide by 0, or unaligned data, for example
- Interrupts are a change of control flow typically signaled from outside the processor core
  - •Interrupt Service Routines (ISRs) are used to handle *ints*
  - •Critical and non-critical inputs driven by interrupt sources
- According to IBM: an exception is an event that may or may not be processed; interrupts occur as a result of an exception

Interrupt Handlers

- Service both interrupts and exceptions
  - Current program execution is suspended after the current instruction
  - Context information is saved so that execution can return to the current program
  - Execution is transferred to an interrupt handler to service the interrupt
  - Interrupt handler must be registered
  - Interrupt handler calls an ISR
  - For simple situations, the handler and ISR can be combined operations
  - Each ISR is unique to the task at hand
    - Uart interrupt to process a character
    - Divide-by-zero exception to change program flow
  - When finished
  - Normal — returns to point in program where interrupt occurred
  - Exception — branches to error recovery

Interrupt Types

- Edge-triggered
  - Parameter: SENSITIVITY
  - Rising edge, attribute: EDGE_RISING
  - Falling edge, attribute: EDGE_FALLING
  - Example

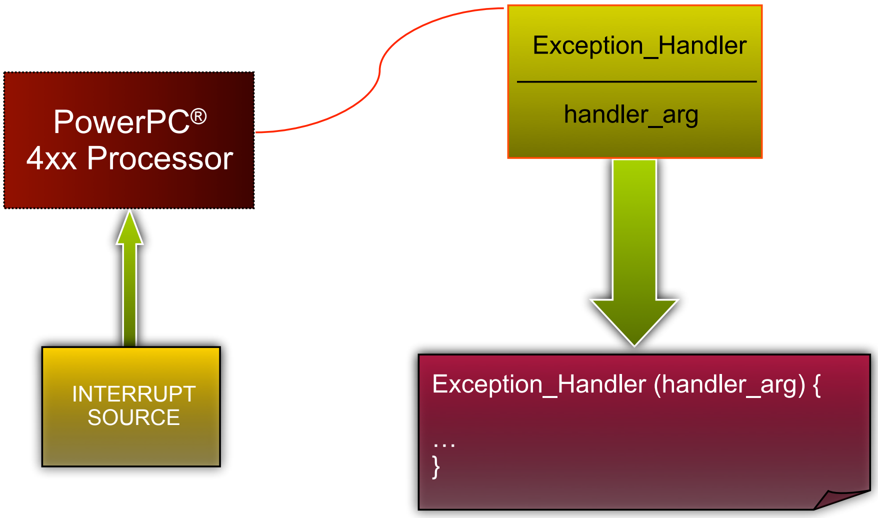
```
PORT interrupt = int_signal, DIR = O, SENSITIVITY =
    EDGE_FALLING, SIGIS = INTERRUPT
```

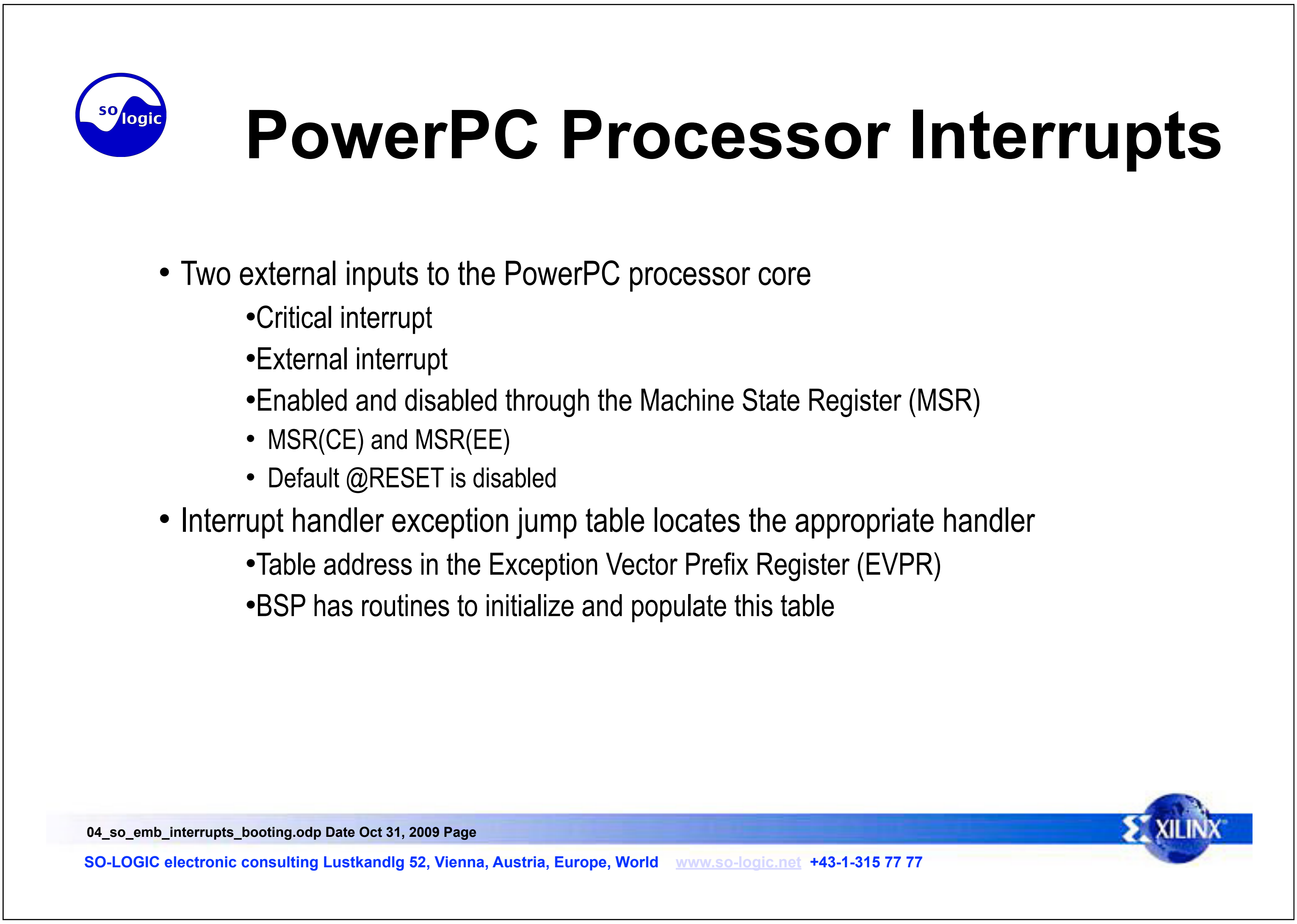
- Level-triggered
  - Parameter: SENSITIVITY
  - High, attribute: LEVEL_HIGH
  - Low, attribute: LEVEL_LOW
  - Example

```
PORT interrupt = int_signal, DIR = O, SENSITIVITY =
    LEVEL_HIGH, SIGIS = INTERRUPT
```

PowerPC Processor

Two interrupt pins, external and critical



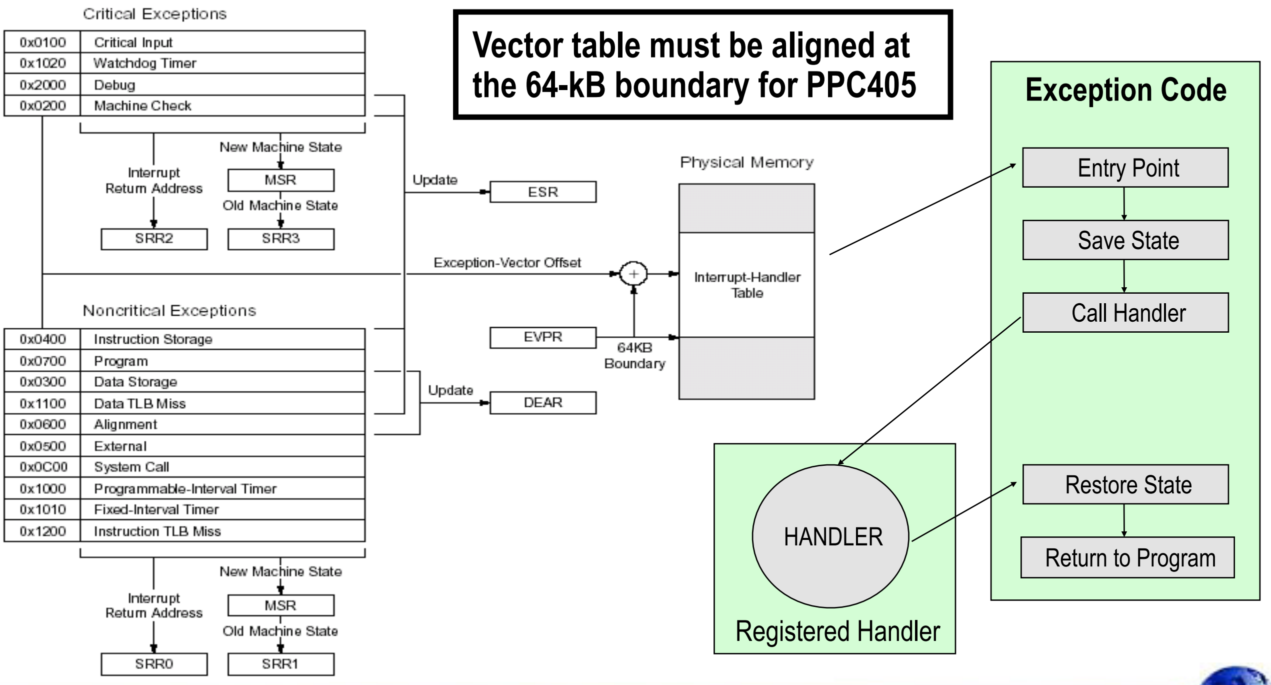


PowerPC Processor Interrupts

- Two external inputs to the PowerPC processor core
  - •Critical interrupt
  - •External interrupt
  - •Enabled and disabled through the Machine State Register (MSR)
    - MSR(CE) and MSR(EE)
    - Default @RESET is disabled
- Interrupt handler exception jump table locates the appropriate handler
  - •Table address in the Exception Vector Prefix Register (EVPR)
  - •BSP has routines to initialize and populate this table

The PPC440 is more customizable. The base of the Exception Vector table is specified with the EVPR. The offset from this base for each exception type is specified with the IVOR 0-15 registers.

PowerPC Processor Exception Mechanism



Vector table must be aligned at the 64-kB boundary for PPC405

Critical Exceptions

0x0100	Critical Input
0x1020	Watchdog Timer
0x2000	Debug
0x0200	Machine Check

New Machine State

Interrupt Return Address

MSR

Old Machine State

SRR2

SRR3

Update

ESR

Exception-Vector Offset

Physical Memory

Interrupt-Handler Table

EVPR

64KB Boundary

Update

DEAR

Noncritical Exceptions

0x0400	Instruction Storage
0x0700	Program
0x0300	Data Storage
0x1100	Data TLB Miss
0x0600	Alignment
0x0500	External
0x0C00	System Call
0x1000	Programmable-Interval Timer
0x1010	Fixed-Interval Timer
0x1200	Instruction TLB Miss

New Machine State

Interrupt Return Address

MSR

Old Machine State

SRR0

SRR1

Exception Code

Entry Point

Save State

Call Handler

Restore State

Return to Program

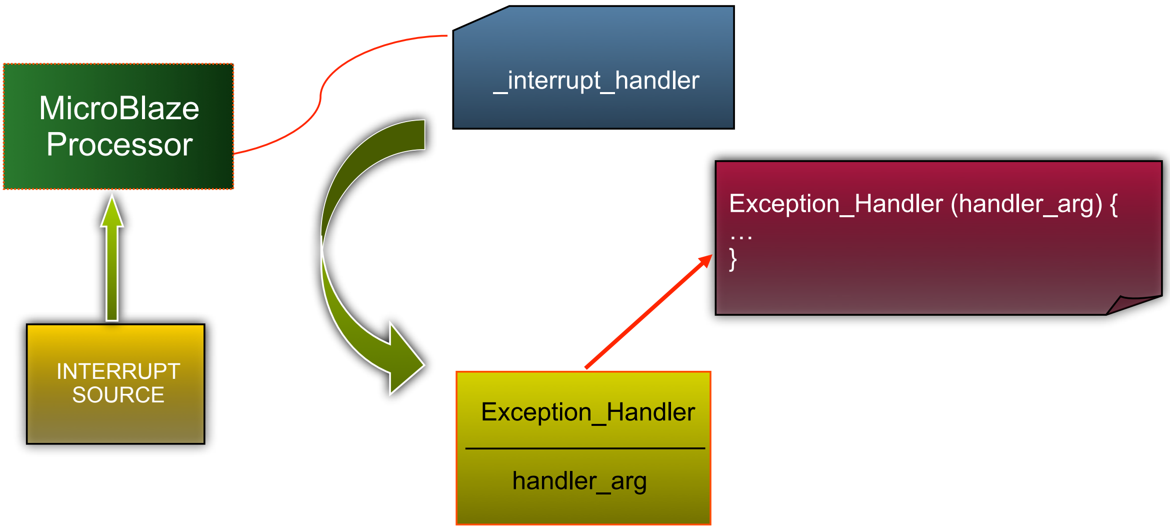
HANDLER

Registered Handler

PowerPC Library Support

- Exception library function calls support
  - •void XExc_Init(void);
    - Initializes the vector table and default handlers
  - •void XExc_RegisterHandler (Xuint8 ExceptionId, XExceptionHandler Handler, void *DataPtr);
    - Allows the registration of a handler
  - •void XExc_RemoveHandler (Xuint8 ExceptionId);
    - Replaces the current handler with a default handler
  - •void XExc_mEnableExceptions (EnableMask);
    - Enables critical and noncritical interrupts
  - •void XExc_mDisableExceptions (DisableMask);
    - Disables critical and noncritical interrupts

MicroBlaze Processor



On interrupts, the MicroBlaze processor jumps to address location 0x10. This is part of the C run-time library and contains a jump to the default interrupt handler (_interrupt_handler).

This function is part of the MicroBlaze processor BSP and is provided by Xilinx. It accesses an interrupt vector table to determine the name of the interrupt handler for the interrupt source.

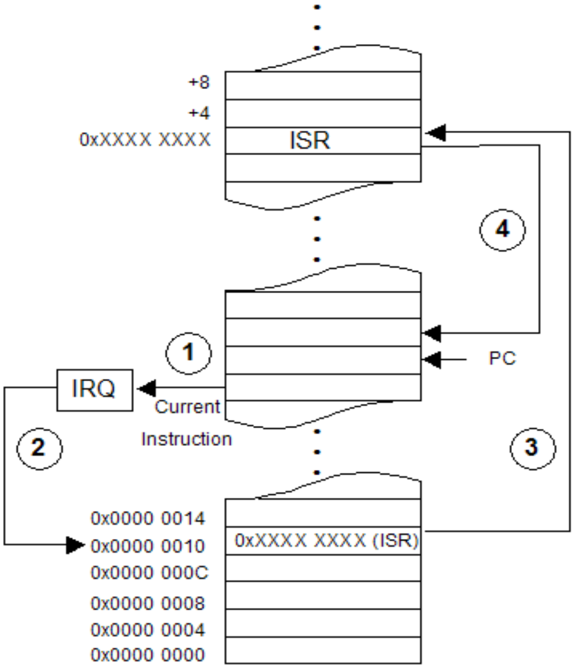
The interrupt vector table is a single entry table. The entry is a combination of the ISR and an argument that should be used with the ISR. This entry can be programmed in the user code.

Functions are provided in the MicroBlaze processor BSP for changing the handler of the interrupt source at run time.

MicroBlaze Processor Interrupts

- Support for five vectored events
  - Reset – cold boot
  - User vector exception – create your own exception
  - **Interrupt – external interrupt input; this is the one of interest for this module**
  - Break – hardware break inputs and software break instructions
  - Hardware exception – illegal instruction, data bus error, unaligned access
- All of the above vectored events operate somewhat similarly
  - See next slide

MicroBlaze Processor Interrupts



1. Interrupt Req. Occurs
   a. Save next instruction location (PC) in GP Reg 14
   b. Save register information
2. Jump to IRQ address
3. Execute _interrupt_handler code
4. Return to main program
   a. Restore register information
   b. Resume operation with PC value prior to IRQ

The MicroBlaze processor supports one external interrupt source (connecting to the interrupt input port). The processor only reacts to interrupts if the Interrupt Enable (IE) bit in the Machine Status Register (MSR) is set to 1. On an interrupt, the instruction in the execution stage completes, while the instruction in the decode stage is replaced by a branch to the interrupt vector (address 0x10). The interrupt return address (the PC associated with the instruction in the decode stage at the time of the interrupt) is automatically loaded into general-purpose register R14. In addition, the processor also disables future interrupts by clearing the IE bit in the MSR.

Interrupts are ignored by the processor if the Break In Progress (BIP) bit in the MSR register is set to 1.

MicroBlaze Processor Interrupts

- MicroBlaze processor functions
  - •void microblaze_enable_interrupts(void)
    - Enables interrupts on the MicroBlaze processor
    - When the MicroBlaze processor starts up, interrupts are disabled. Interrupts must be explicitly turned on with this function
  - •void microblaze_disable_interrupts(void)
    - Disables interrupts on the MicroBlaze processor. This function can be called when entering a critical section of code where a context switch is undesirable
  - •void microblaze_register_handler ((XExceptionHandler)timer_int_handler, (void *) 0);
    - Required for external port interrupt without interrupt controller

MicroBlaze Processor Exceptions

- MicroBlaze processor functions
  - void microblaze_enable_exceptions(void)
    - Enables interrupts on the MicroBlaze processor
    - When the MicroBlaze processor starts up, interrupts are disabled. Interrupts must be explicitly turned on with this function
  - void microblaze_disable_ exceptions(void)
    - Disables interrupts on the MicroBlaze processor. This function can be called when entering a critical section of code where a context switch is undesirable
  - void microblaze_register_exception_handler ((XExceptionHandler)timer_int_handler, (void *) 0);
    - Required for external port interrupt without interrupt controller

Interrupt Inclusion

- Requirements for including an interrupt into your application
  - •Write a *void* software function that services the interrupt
  - •Use the provided device IP routines to facilitate writing the ISR
    - Clear interrupt
    - Get interrupt status
    - Enable/disable interrupt
  - •Register the interrupt handler by using an appropriate function
    - Single external interrupt registers with the processor function
    - Multiple external interrupts register with the interrupt controller

Initializing Interrupt Example: PIT Timer

- When the source of the interrupt or exception is internal to the PowerPC processor, you should explicitly perform the following tasks (using the PIT timer as an example) in the *main()* routine
    - Initialize an exception in the vector space of the handler
        - XExc_Init();
    - Register the interrupt handler
        - XExc_RegisterHandler(XEXC_ID_PIT_INT, (XExceptionHandler) pit_timer_int_handler, (void *) 0);
    - Initialize and enable the device
        - XTime_PITSetInterval(0xffffff00);
        - XTime_PITEnableAutoReload();
    - Arm the device
        - XTime_PITEnableInterrupt() ;
    - Enable PowerPC processor noncritical interrupts
        - XExc_mEnableExceptions(XEXC_NON_CRITICAL);

Initializing Interrupt Example

MicroBlaze Processor + Timer

```
main() {
Xuint32 k;
//  Establish timer operation, set # of cycles TMR0 counts in LOAD reg
XTmrCtr_mSetLoadReg(XPAR_PLB_TIMER_1_BASEADDR, 0, TMR0);
// Reset Timer0 and clear interrupts, variables defined in xtmrctr_l.h
XTmrCtr_mSetControlStatusReg(XPAR_PLB_TIMER_1_BASEADDR,
0,XTC_CSR_INT_OCCURED_MASK | XTC_CSR_LOAD_MASK | XTC_CSR_EXT_GENERATE_MASK);
// register interrupt handler
Microblaze_register_handler((XInterruptHandler) timer_interrupt_handler, (void
*)0);
// Start timer using masks bits defined in xtmctr_l.h library header file
XTmrCtr_mSetControlStatusReg(XPAR_PLB_TIMER_1_BASEADDR, 0,
XTC_CSR_ENABLE_ALL_MASK | XTC_CSR_ENABLE_INT_MASK | XTC_CSR_AUTO_RELOAD_MASK |
XTC_CSR_DOWN_COUNT_MASK);
// Enable microblaze interrupts to turned on interrupt using function
microblaze_enable_interrupts();
// Do other tasks
…
// Disable interrupt when done
microblaze_disable_interrupts():}
```

Interrupt Handler

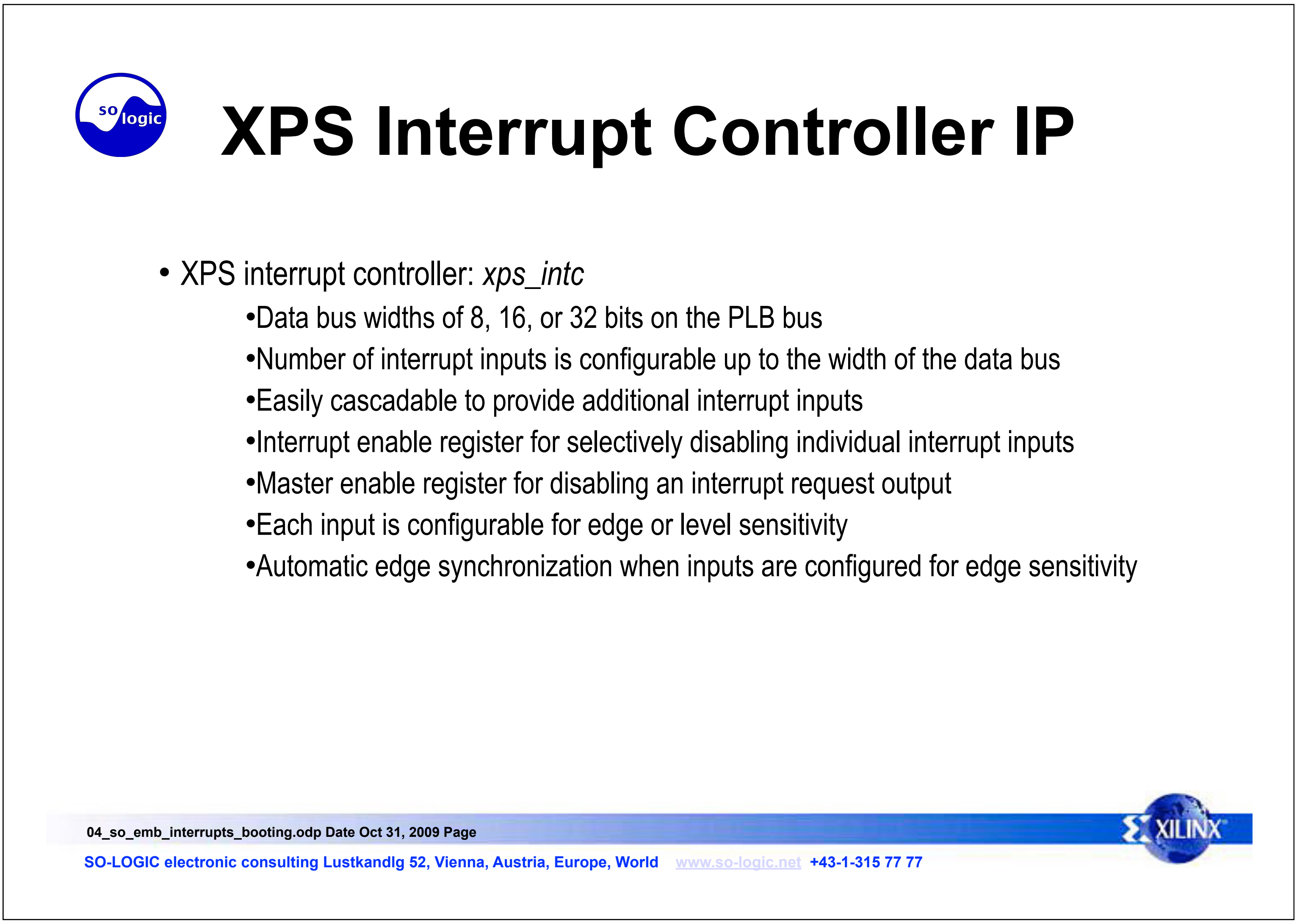
Software Driver: tmrctr for xps_timer

```
void timer_int_handler(void * baseaddr_p) {
Xuint32 tcsr_0;

  /* Get timer status register */
  tcsr_0 = XTmrCtr_mGetControlStatusReg(baseaddr_p,0);
  if (tcsr_0 & XTC_CSR_INT_OCCURED_MASK) {
      /* Do what needs to be done if the source
         of interrupt is timer */
      …
   }
   /* Clear the timer interrupt */
   XTmrCtr_mSetControlStatusReg(XPAR_PLB_TIMER_1_BASEADDR, 0, csr);
}
```

Multiple External Interrupts

- Direct method – OR all interrupts by using the Utility Reduced Logic core
  - Minimal hardware
  - Interrupt service routine must check all possible interrupting sources
  - Higher latency response time
  - Not a compatible method with Xilinx IP software drivers
- Interrupt controller – instantiate the XPS interrupt controller
  - Support up to 32 interrupts
  - Single interrupt to processor
  - Compatible with Xilinx IP software drivers
  - Easy to use



XPS Interrupt Controller IP

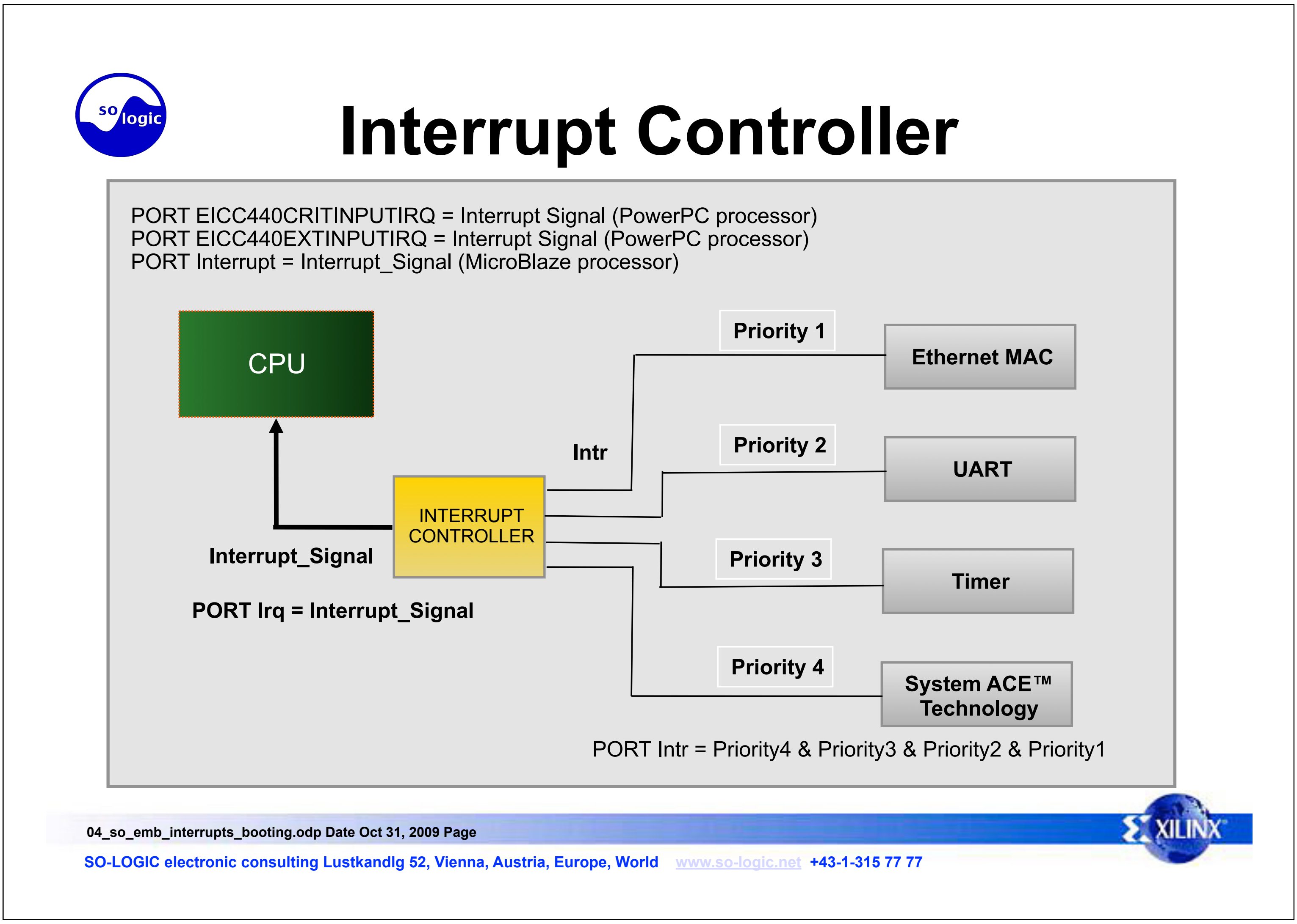
- XPS interrupt controller: *xps_intc*
  - •Data bus widths of 8, 16, or 32 bits on the PLB bus
  - •Number of interrupt inputs is configurable up to the width of the data bus
  - •Easily cascadable to provide additional interrupt inputs
  - •Interrupt enable register for selectively disabling individual interrupt inputs
  - •Master enable register for disabling an interrupt request output
  - •Each input is configurable for edge or level sensitivity
  - •Automatic edge synchronization when inputs are configured for edge sensitivity

Each input is configurable for edge or level sensitivity. Edge sensitivity can be configured for rising or falling; level sensitivity can be active high or low; and automatic edge synchronization when inputs are configured for edge sensitivity.

The output interrupt request pin is configurable for edge or level generation—edge generation configurable for rising or falling and level generation configurable for active high or low.

The interrupt controller is intended for use in a hard vector interrupt system. It does not directly provide auto-vectoring capability. However, it does provide a vector number that can be used in a software-based vectoring scheme.

The PLB interface provides a slave interface on the PLB for transferring data between the PLB interrupt controller and the processor. The PLB interrupt controller registers are memory mapped into the PLB address space and data transfers occur using PLB byte enables. The register addresses are fixed on four-byte boundaries and the registers and the data transfers to and from them are always as wide as the data bus.



Interrupt Controller

PORT EICC440CRITINPUTIRQ = Interrupt Signal (PowerPC processor)
PORT EICC440EXTINPUTIRQ = Interrupt Signal (PowerPC processor)
PORT Interrupt = Interrupt_Signal (MicroBlaze processor)

CPU

INTERRUPT CONTROLLER

Interrupt_Signal

PORT Irq = Interrupt_Signal

Intr

Priority 1

Ethernet MAC

Priority 2

UART

Priority 3

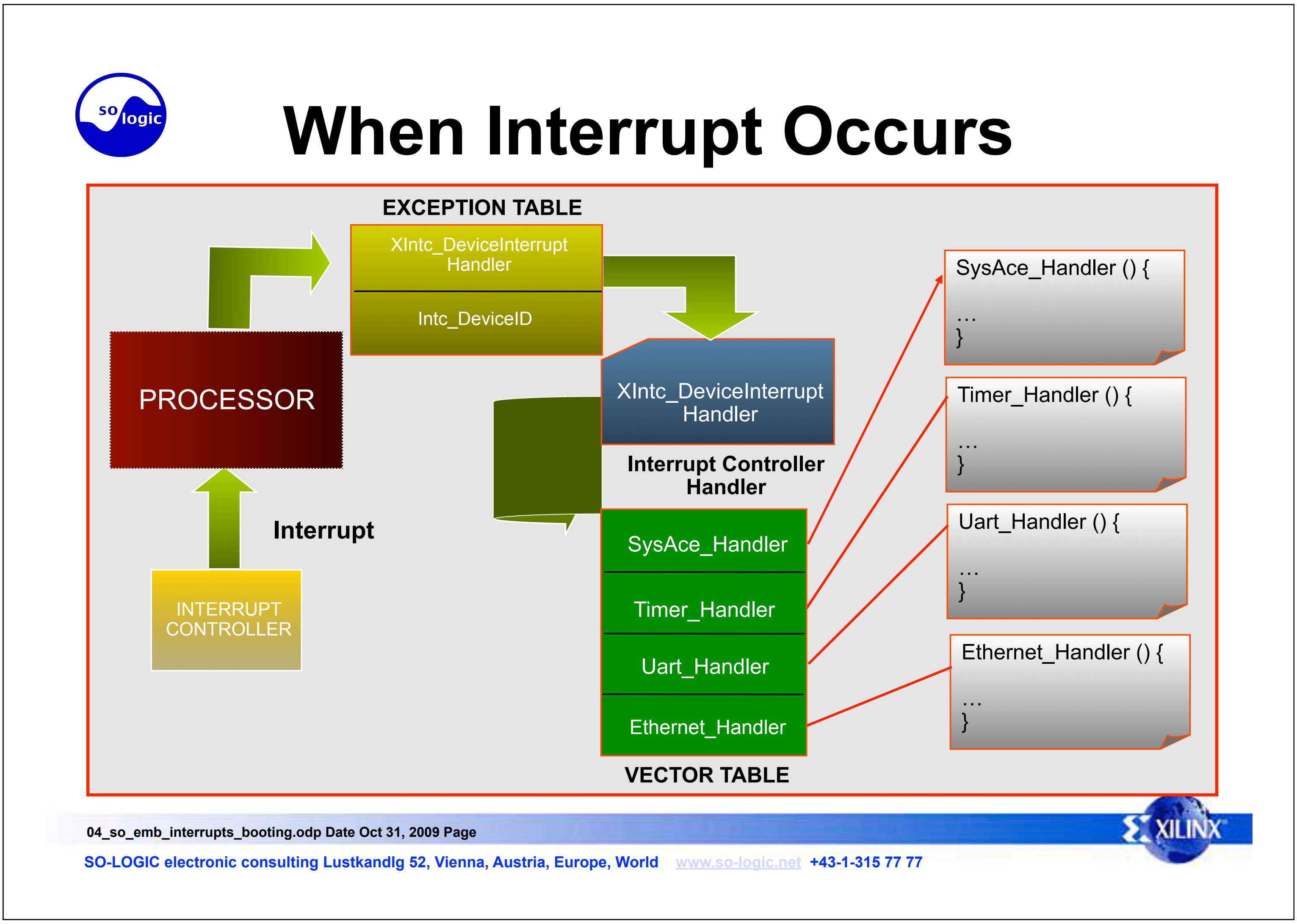
Timer

Priority 4

System ACE™ Technology

PORT Intr = Priority4 & Priority3 & Priority2 & Priority1

When the project is created by Base System Builder, it always includes an interrupt controller even if there is only one interrupting source.



When Interrupt Occurs

EXCEPTION TABLE

XIntc_DeviceInterrupt Handler

Intc_DeviceID

PROCESSOR

XIntc_DeviceInterrupt Handler

Interrupt Controller Handler

Interrupt

INTERRUPT CONTROLLER

SysAce_Handler

Timer_Handler

Uart_Handler

Ethernet_Handler

VECTOR TABLE

SysAce_Handler () {
…
}

Timer_Handler () {
…
}

Uart_Handler () {
…
}

Ethernet_Handler () {
…
}

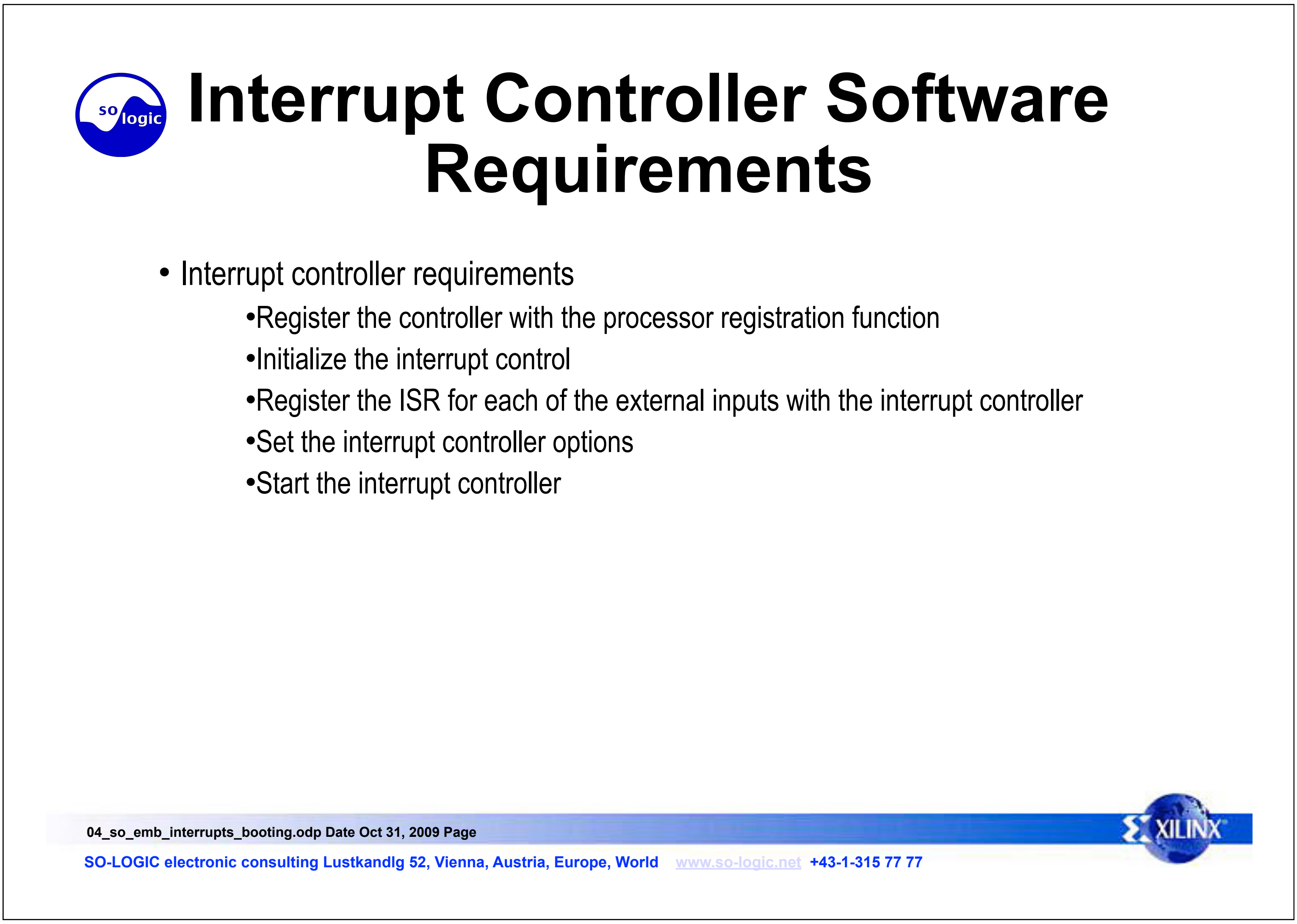
On interrupts, the MicroBlaze processor jumps to the handler (XIntc_DeviceInterruptHandler) of the interrupt controller peripheral by using the interrupt vector table. LibGen automatically registers the handler of the interrupt controller peripheral in the interrupt vector table. The interrupt controller handler services each interrupt signal that is active, starting from the highest priority signal. Each of the peripheral interrupt signals needs to be associated with an interrupt.

Handler routine (also called the Interrupt Service Routine): The interrupt controller handler uses a vector table to store those routines corresponding to each of the interrupt signals. If an interrupt is active, the interrupt controller handler calls the routine corresponding to it. An argument can be associated with such routines and is passed when calling the routine. LibGen automatically generates the vector table used by the interrupt controller handler.

The association of an ISR for a peripheral interrupt signal can be done either in the Microprocessor Software Specifications (MSS) file or registered at run time by using the function provided by the interrupt controller driver (XIntc_Connect, XIntc_RegisterHandler). These functions work on the vector table generated by LibGen. For more information on the exact prototype of these functions, refer to the Device Drivers documentation.

If the ISRs are specified in the MSS file, LibGen automatically registers these routines with the vector table of the interrupt controller driver listed in the order of priority. The base address of the peripherals is registered as the arguments to be passed to the ISR in the vector table.

For the PowerPC processor, apart from the registering of the handler with the exception table, the rest of the processing is similar to the MicroBlaze processor.



Interrupt Controller Software Requirements

- Interrupt controller requirements
  - Register the controller with the processor registration function
  - Initialize the interrupt control
  - Register the ISR for each of the external inputs with the interrupt controller
  - Set the interrupt controller options
  - Start the interrupt controller

These are the steps you perform to enable interrupts.

Interrupt Handler Registration

- The interrupt handler is registered through interrupt controller driver functions (IP name: *xps_intc* or *dcr_intc*)
  - **XIntc_RegisterHandler** or **XIntc_Connect** and **XIntc_Initialize** functions

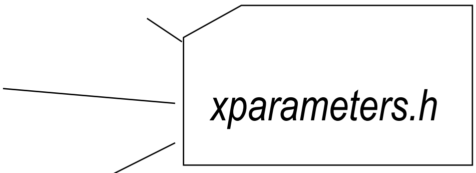
Level 0

```
XIntc_RegisterHandler (XPAR_PLB_INTC_0_BASEADDR,
   XPAR_PLB_INTC_0_MYUART_INTERRUPT_INTR,
   (XInterruptHandler) uart_int_handler,
   (void *)XPAR_MYUART_BASEADDR);
```

OR

```
XIntc_Initialize (&Intc, XPAR_PLB_INTC_0_DEVICE_ID);
XIntc_Connect (&Intc, XPAR_PLB_INTC_0_MYUART_INTERRUPT_INTR,
   (XInterruptHandler) uart_int_handler,
   (void *) XPAR_MYUART_BASEADDR);
```

Level 1



xparameters.h

There are two methods for registering an interrupt handler. One method is to use the XIntc_RegisterHandler function, which requires the base address of the interrupt controller, the interrupt request, the name of the interrupt handler, and the base address of the interrupt requesting device. The other method is to use two functions: XIntc_Initialize and XIntc_Connect.

Interrupt Considerations

- Interrupts are considered asynchronous events
  - •Know the nature of your interrupt
  - Edge or level
  - How the the interrupt is cleared
  - What happens if another event occurs while the interrupt is asserted?
  - •How frequent can the interrupt event occur?
- Can the system tolerate missing an interrupt?

ISR Considerations

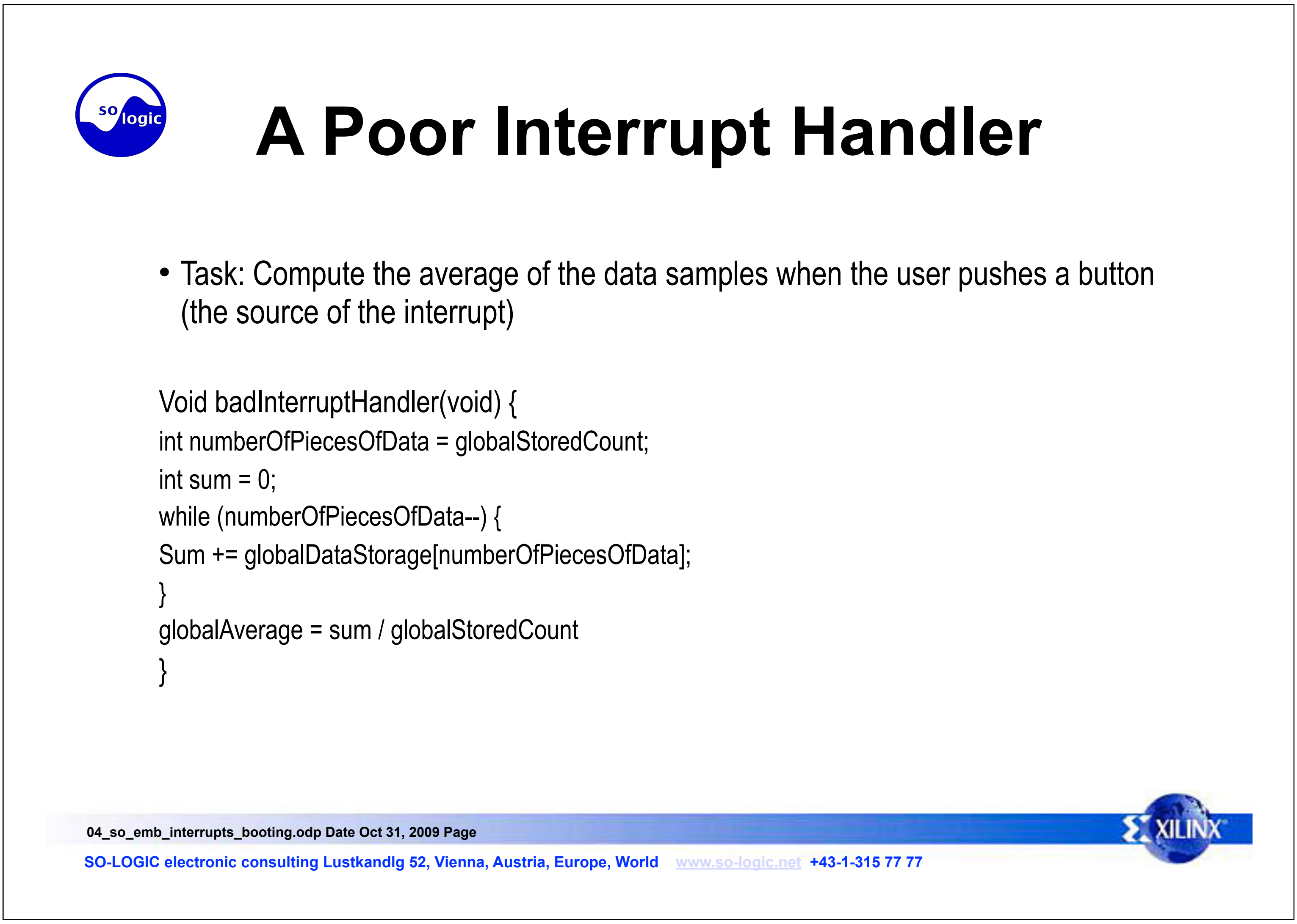
- Timing
  - •What is the latency from the hardware to the ISR?
    - Operating system can aggravate this
    - Are the interrupts prioritized?
  - •How long can the ISR be active before affecting other things in the system?
- Can the ISR be interrupted?
  - •If so, code must be written to be reentrant
  - •A practice to be avoided
- Code portability
  - •Compatible with MicroBlaze and PowerPC processors
  - •Are operating system hooks needed?

ISR Tips and Tricks

- Keep the code short and simple; ISRs can be difficult to debug
- Do not allow other interrupts while in the ISR
    - This is a system design consideration and not a recommended practice
    - Use interrupt priority when using an interrupt controller
- Time is of the essence!
    - Spend as little time as possible in the ISR
    - Do not perform tasks that can be done in the background
    - Use flags to signal background functions
- Make use of the callback argument passed in registration
- Make use of provided interrupt support functions when using IP drivers
- Do not forget to enable interrupts when leaving the handler/ISR

Guidelines for Writing a Good Interrupt Handler

- Keep the interrupt handler code brief (in time)
  - Avoid loops (especially open-ended while statements)
- Keep the interrupt handler simple
  - Interrupt handlers can be very difficult to debug
- Disable interrupts as they occur
  - Re-enable the interrupt as you exit the handler
- Budget your time
  - Interrupts are never implemented for fun—they are required to meet a specified response time
  - Predict how often an interrupt is going to occur and how much time your interrupt handler takes
  - Spending your time in an interrupt handler increases the risk that you may miss another interrupt



A Poor Interrupt Handler

- Task: Compute the average of the data samples when the user pushes a button (the source of the interrupt)

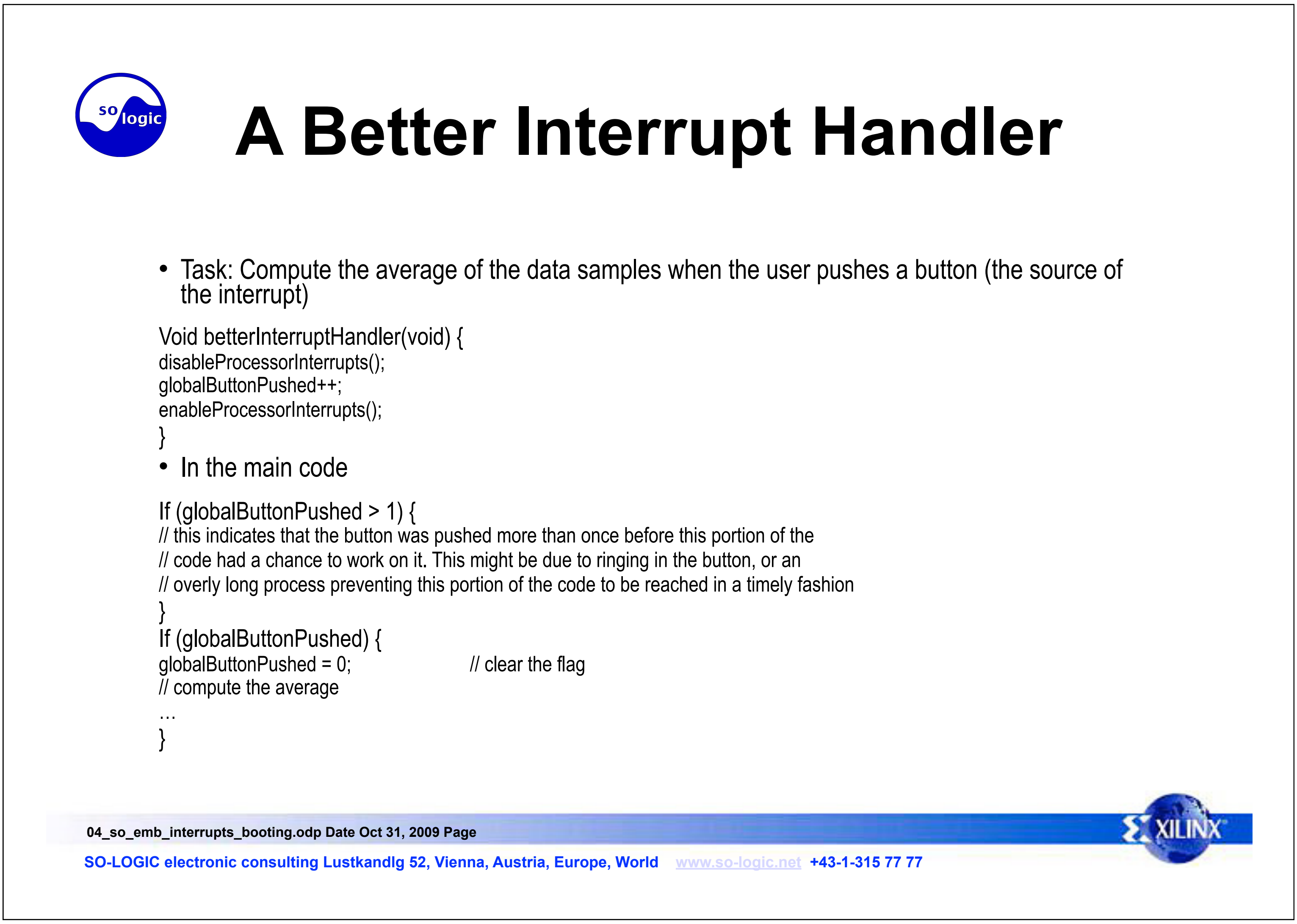
```
Void badInterruptHandler(void) {
int numberOfPiecesOfData = globalStoredCount;
int sum = 0;
while (numberOfPiecesOfData--) {
Sum += globalDataStorage[numberOfPiecesOfData];
}
globalAverage = sum / globalStoredCount
}
```

The amount of time that this interrupt routine takes to run is non-deterministic. If there is only a few pieces of data, then this might run quickly, which is good. If there is a large amount of data, then this routine takes much longer, which is bad. If there is *no* data, then this routine will take the maximum amount of time (counting down from 2^32 – 1 to 0 or about 4 billion), which is terrible.

There is no protection if globalStoredCount is 0 (this will throw a divide by zero error).

There is excessive use of global variables. The side effects could be horrendous. Use of global variables in this case violates reentrancy rules.

If the processor allows interrupts during the handling of an existing interrupt, conflicts with global variables will occur. Review reentrancy rules.



A Better Interrupt Handler

- Task: Compute the average of the data samples when the user pushes a button (the source of the interrupt)

```
Void betterInterruptHandler(void) {
disableProcessorInterrupts();
globalButtonPushed++;
enableProcessorInterrupts();
}
```

- In the main code

```
If (globalButtonPushed > 1) {
// this indicates that the button was pushed more than once before this portion of the
// code had a chance to work on it. This might be due to ringing in the button, or an
// overly long process preventing this portion of the code to be reached in a timely fashion
}
If (globalButtonPushed) {
globalButtonPushed = 0;                    // clear the flag
// compute the average
…
}
```

This interrupt handler is much safer than the previous example because:

- Processor interrupts are disabled on entry and re-enabled on exit, which prevents the interrupt handler from being interrupted.
- A global flag is set that can be monitored by the main loop.

Summary

- An interrupt controller is not required when the number of interrupt sources is less than or equal to the number of interrupt pins on the CPU
- An interrupt controller supports up to 32 interrupt sources and provides a means for assigning priority
- Interrupt handlers are required to perform the desired task when the interrupt occurs. They must be registered through explicit execution of a register handler function

The Designer's Challenge Download and Boot

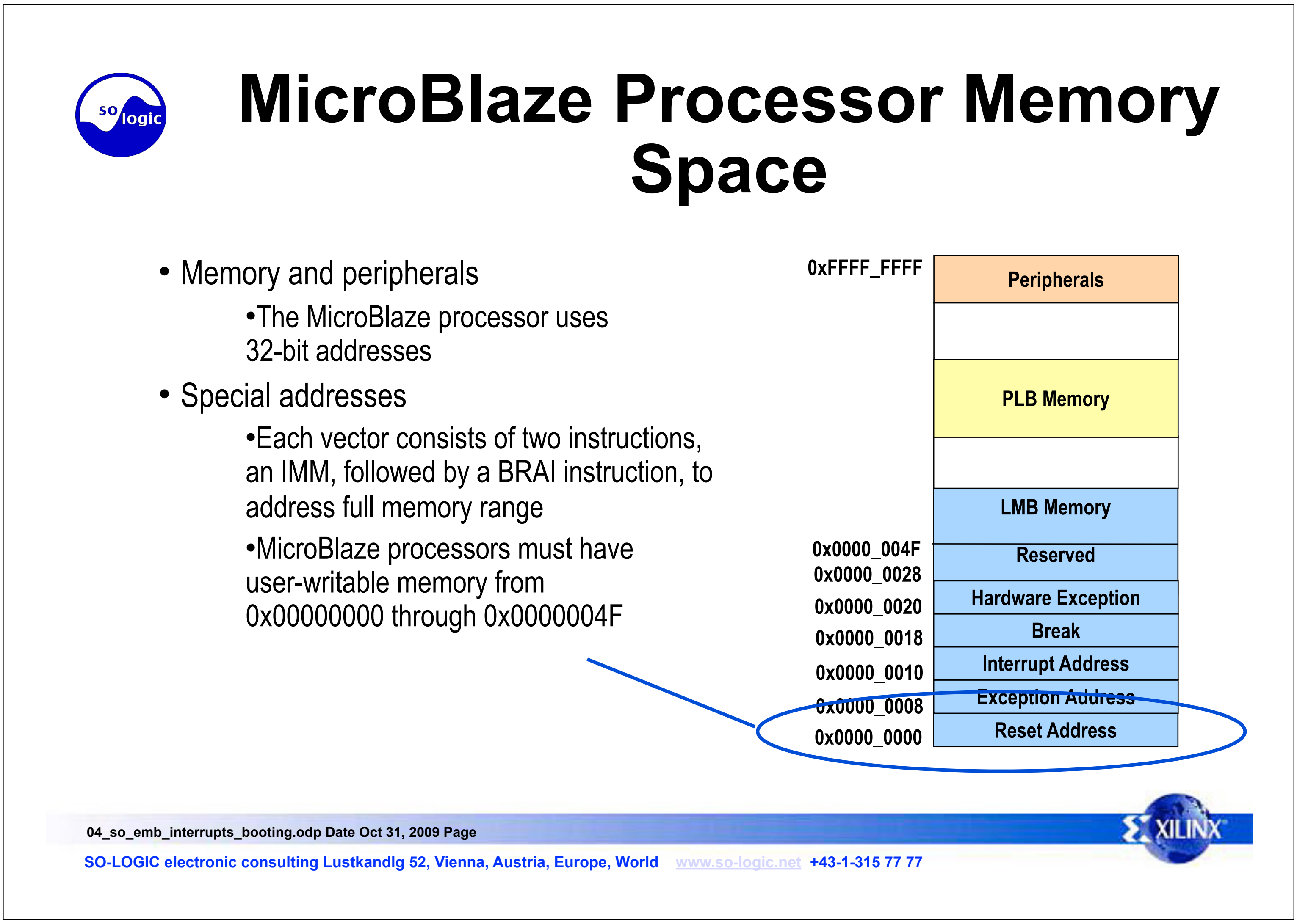
What is the best boot option?

Objectives

- Describe bootload options
- Identify the bootload sequence
- Describe the program load requirements between off-FPGA devices and block RAM
- Use XMD for loading memory and simple program control
- Understand the use of *bootloops* for debugger control
- Write a program to bootload from off-FPGA flash or another peripheral

Microprocessor Boot

- Microprocessors do not start from *main{}*
- The boot process ends with a call to *main{}*
- The boot program is automatically included during the compile, link, locate, and generate ELF file process, **Build All User Applications**
  - Boot code provided
  - Guided by linker script defaults
- Boot process is a processor dependent operation
  - Microblaze™ processor
  - Power PC® processor



MicroBlaze Processor Memory Space

- Memory and peripherals
  - •The MicroBlaze processor uses 32-bit addresses
- Special addresses
  - •Each vector consists of two instructions, an IMM, followed by a BRAI instruction, to address full memory range
  - •MicroBlaze processors must have user-writable memory from 0x00000000 through 0x0000004F

Address	Region
0xFFFF_FFFF	Peripherals
	PLB Memory
	LMB Memory
0x0000_004F	Reserved
0x0000_0028	
0x0000_0020	Hardware Exception
0x0000_0018	Break
0x0000_0010	Interrupt Address
0x0000_0008	Exception Address
0x0000_0000	Reset Address

The actual address map is defined in the Microprocessor Hardware Specification (MHS) file, which contains an address map that specifies the addresses of LMB memory, PLB memory, and external memory and peripherals. The address range grows from 0. The lowest range is the LMB memory. This is followed by the PLB memory, the external memory, and the peripherals.

Some addresses in this address space have predefined meaning. The processor jumps to address:

•0x0 on reset

•0x8 on exception

•0x10 on interrupt

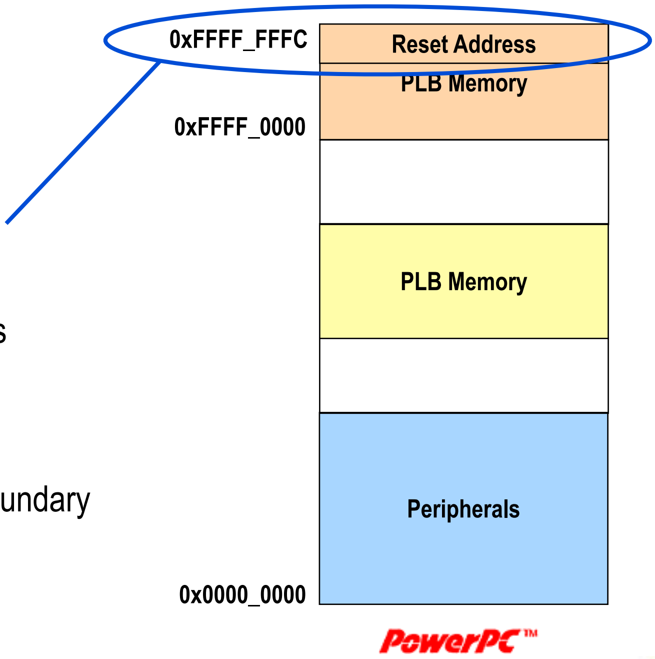
•0x18 on nonmaskable maskable hardware and software break

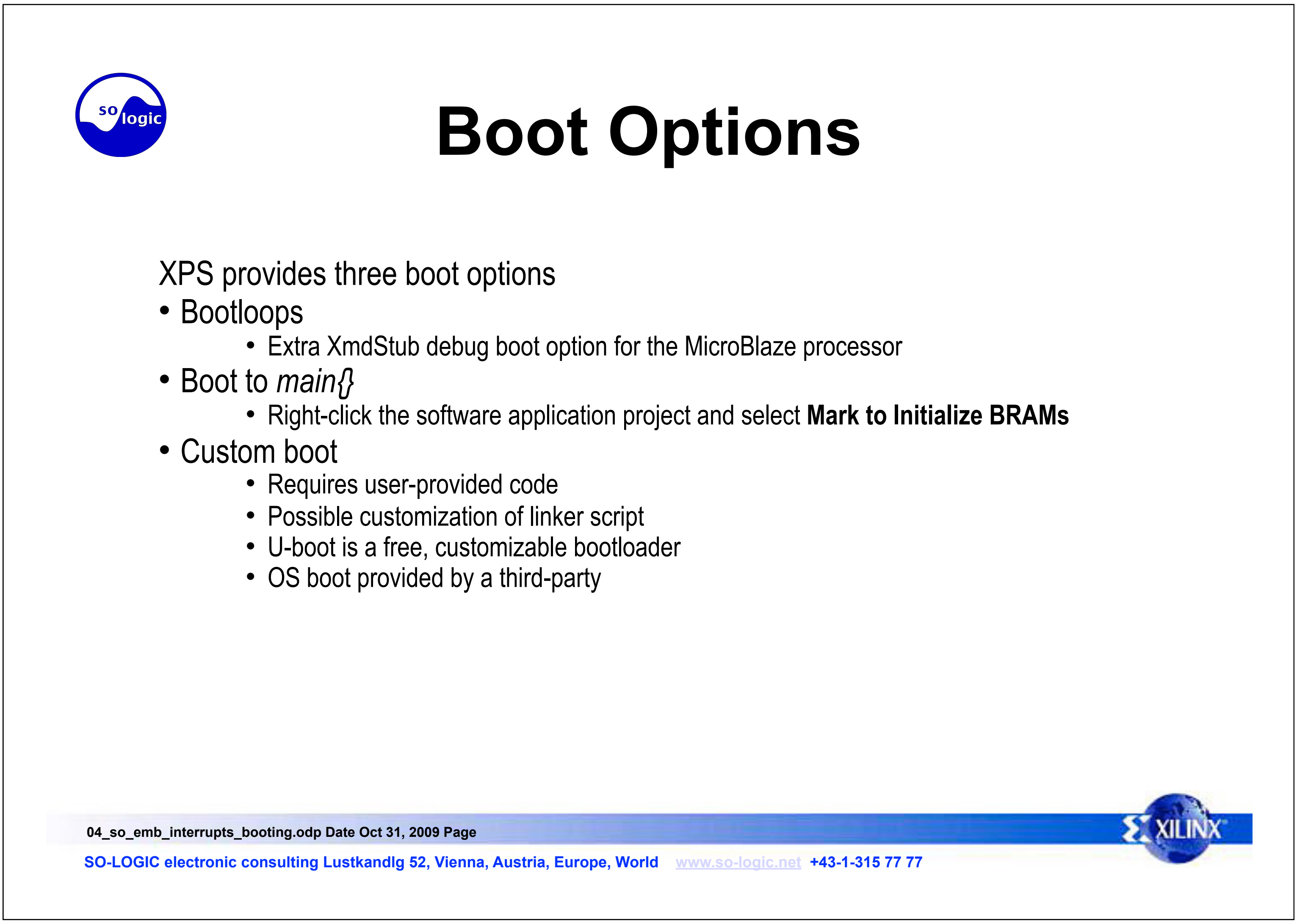
•0x20 on hardware exceptions

Each vector allocates two addresses to allow full address range branching (requires an IMM followed by a BRAI instruction). Register R14 is used to save return addresses on interrupt, R16 on break, and R17 on hardware exception. The address range 0x28 to 0x4F is reserved for future software support by Xilinx.

PowerPC Processor Memory Space

- Memory and peripherals
  - •The PowerPC® 440 processor uses 32-bit addresses
- Special addresses
  - •Every PowerPC processor system should have the boot section starting at 0xFFFFFFFC
  - •The default program space occupies a contiguous address space from 0xFFFF0000 to 0xFFFFFFFF
  - •If interrupt handlers are present, the vector table must start at the 64K boundary





Boot Options

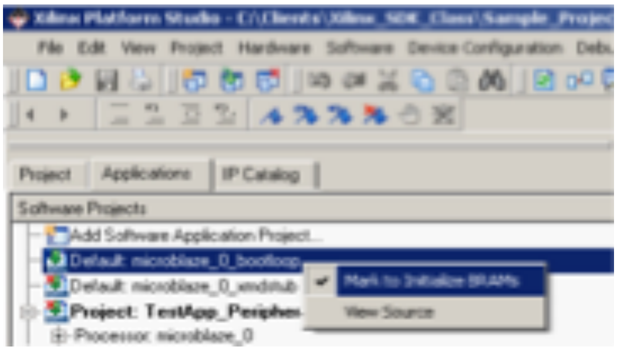
XPS provides three boot options

- Bootloops
  - Extra XmdStub debug boot option for the MicroBlaze processor
- Boot to *main{}*
  - Right-click the software application project and select **Mark to Initialize BRAMs**
- Custom boot
  - Requires user-provided code
  - Possible customization of linker script
  - U-boot is a free, customizable bootloader
  - OS boot provided by a third-party

Initializing the operating system is provided by the OS provider, such as MontaVista, Wind River Systems, or DENX. If you write your own operating system, or customize one from an open source, then you write the initialization code.

Bootloop

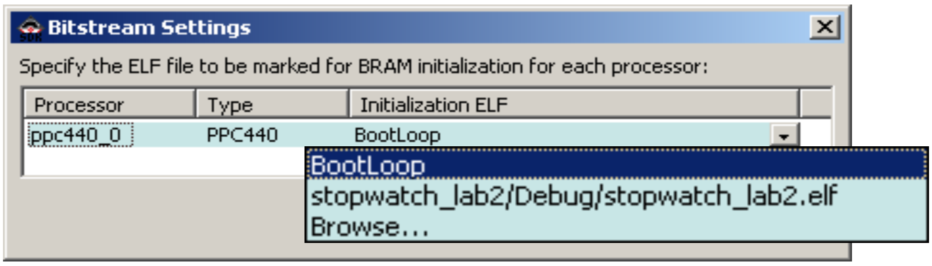
- Used for debugging only!
- Keeps the processor busy after reset until the debugger can take over
- Basically a “branch to *”
  - •Infinite loop
  - •Debugger halts processor and reloads program memory
- Engaged by right-clicking and selecting **Mark to Initialize BRAMs**
  - •Selects bootloop program to compile
  - •Inserted into boot block RAM by data2mem, the **Update Bitstream** command


Xilinx Platform Studio - C:\Clients\Xilinx_SDK_Class\Sample_Projec
File Edit View Project Hardware Software Device Configuration Debu
Project Applications IP Catalog
Software Projects
Add Software Application Project...
Default: microblaze_0_bootloop
Default: microblaze_0_xmdstub
Project: TestApp_Peripher
Processor: microblaze_0
Mark to Initialize BRAMs
View Source

Note to Facilitator: The asterisk is an assembly language notation that indicates the current program counter location. Effectively, this is a loopback to the current location.

SDK Bootloop Options

- The bootloop program will provide a place holder for the processor to do a “branch to *” when the FPGA comes out of configuration



Specify the ELF file to be marked for BRAM initialization for each processor:

Processor	Type	Initialization ELF
ppc440_0	PPC440	BootLoop

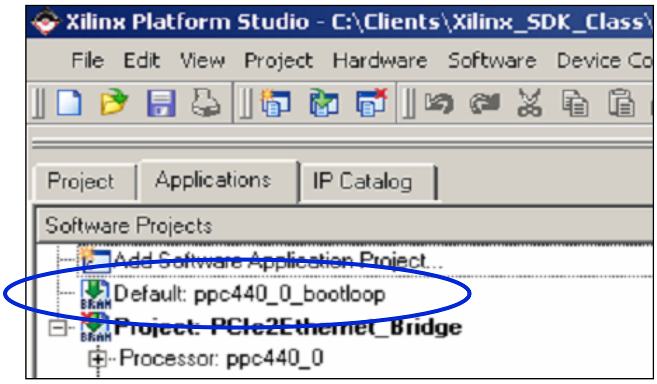
MicroBlaze Processor Bootloop – MDM

- Infinite loop bootloop is selected when the hardware MicroBlaze Debug Module (MDM) is used
  - •The hardware MDM can halt or control the MicroBlaze processor
  - •Recommended debug flow
  - •The MicroBlaze processor bootloop program stored at address 0x00000000

_boot: bri 0

PowerPC Processor Bootloop

- The PowerPC processor always uses an infinite loop bootloop as required by the PPC hardware debug module
- The bootloop code is simple
    - _boot: b _boot
- The code resides at 0xfffffffc, the PowerPC processor reset vector


Xilinx Platform Studio - C:\Clients\Xilinx_SDK_Class\

File Edit View Project Hardware Software Device Co
Project
Applications
IP Catalog
Software Projects
Add Software Application Project...

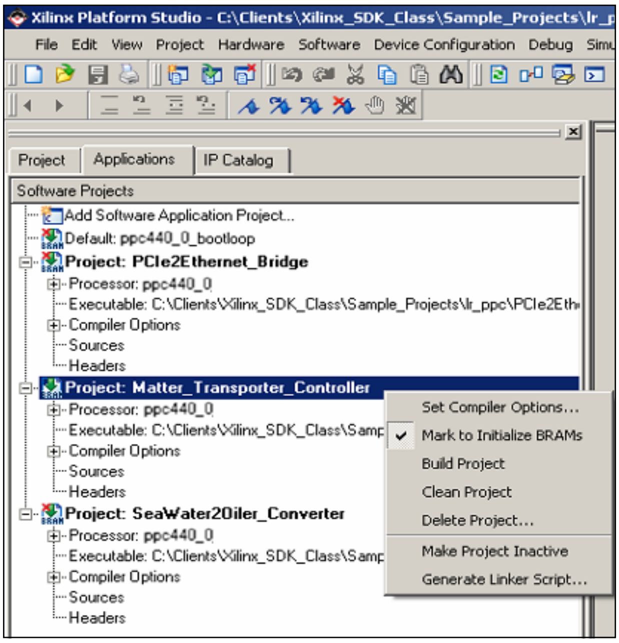
Default: ppc440_0_bootloop

Project: PCIe2Ethernet_Bridge

Processor: ppc440_0

Boot to Main

- This option is the normal run mode that performs the typical boot sequence and starts executing the user application at *main{}*
- Right-click the project and select **Mark to Initialize BRAMs**
- Only one project can be active at a time; notice the red Xs on the unselected software projects

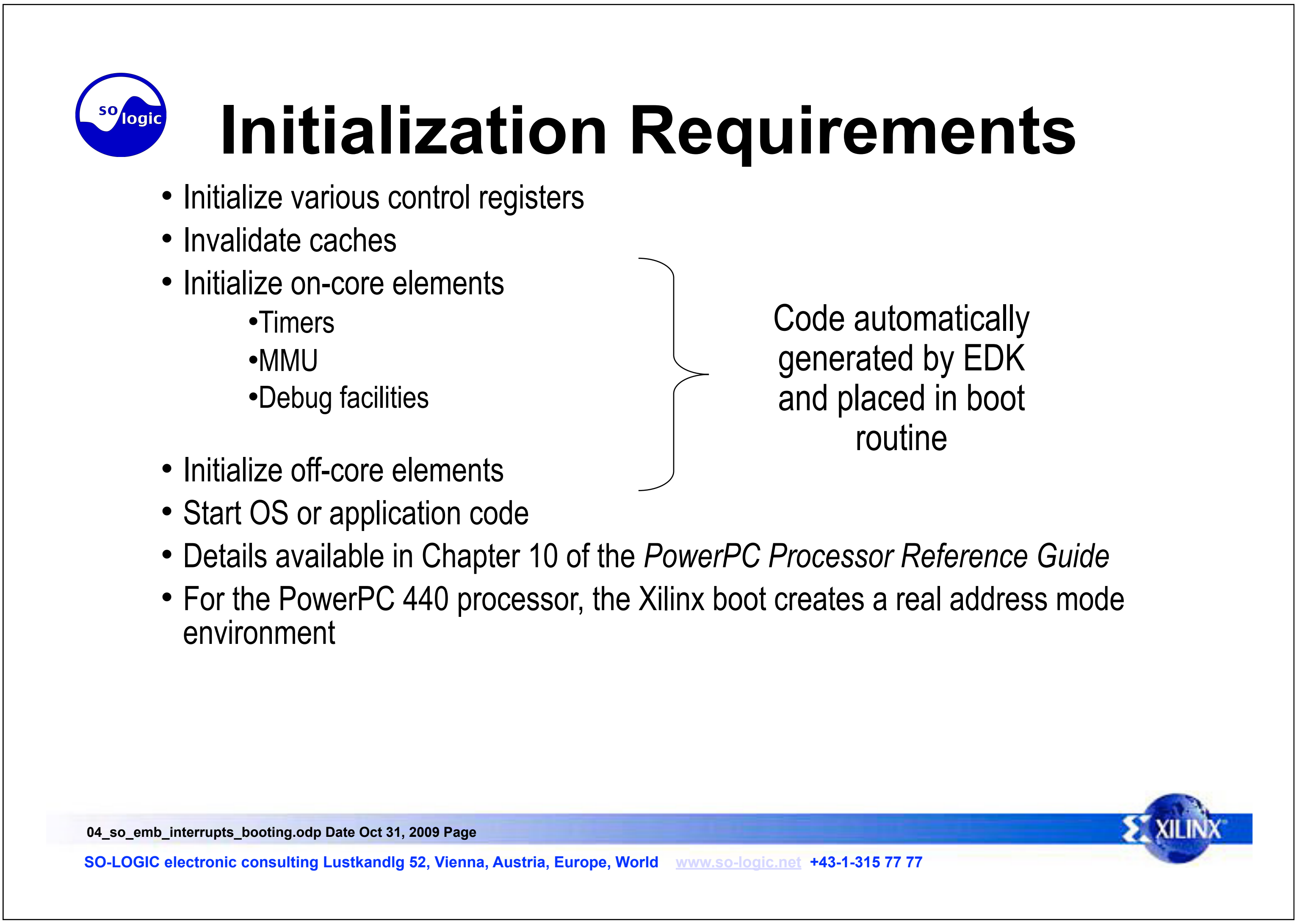


Xilinx Platform Studio - C:\Clients\Xilinx_SDK_Class\Sample_Projects\lr_p

Software Projects
- Add Software Application Project...
- Default: ppc440_0_bootloop
- **Project: PCIe2Ethernet_Bridge**
  - Processor: ppc440_0
  - Executable: C:\Clients\Xilinx_SDK_Class\Sample_Projects\lr_ppc\PCIe2Eth
  - Compiler Options
  - Sources
  - Headers
- **Project: Matter_Transporter_Controller**
  - Processor: ppc440_0
  - Executable: C:\Clients\Xilinx_SDK_Class\Samp
  - Compiler Options
  - Sources
  - Headers
- **Project: SeaWater2Oiler_Converter**
  - Processor: ppc440_0
  - Executable: C:\Clients\Xilinx_SDK_Class\Samp
  - Compiler Options
  - Sources
  - Headers

Custom Boot Options

- Used when it is not desirable to start a user application at *main{}*
  - Operating systems
  - Non-block RAM boot, such as boot from flash
- There will be two application projects, files, linker scripts, and ELF files
  - Target application, including operating system executing in off-FPGA RAM
  - Bootload application that will execute from the processor reset vector
- Requires a custom boot file to replace *crt0.s*
- May require a custom linker script

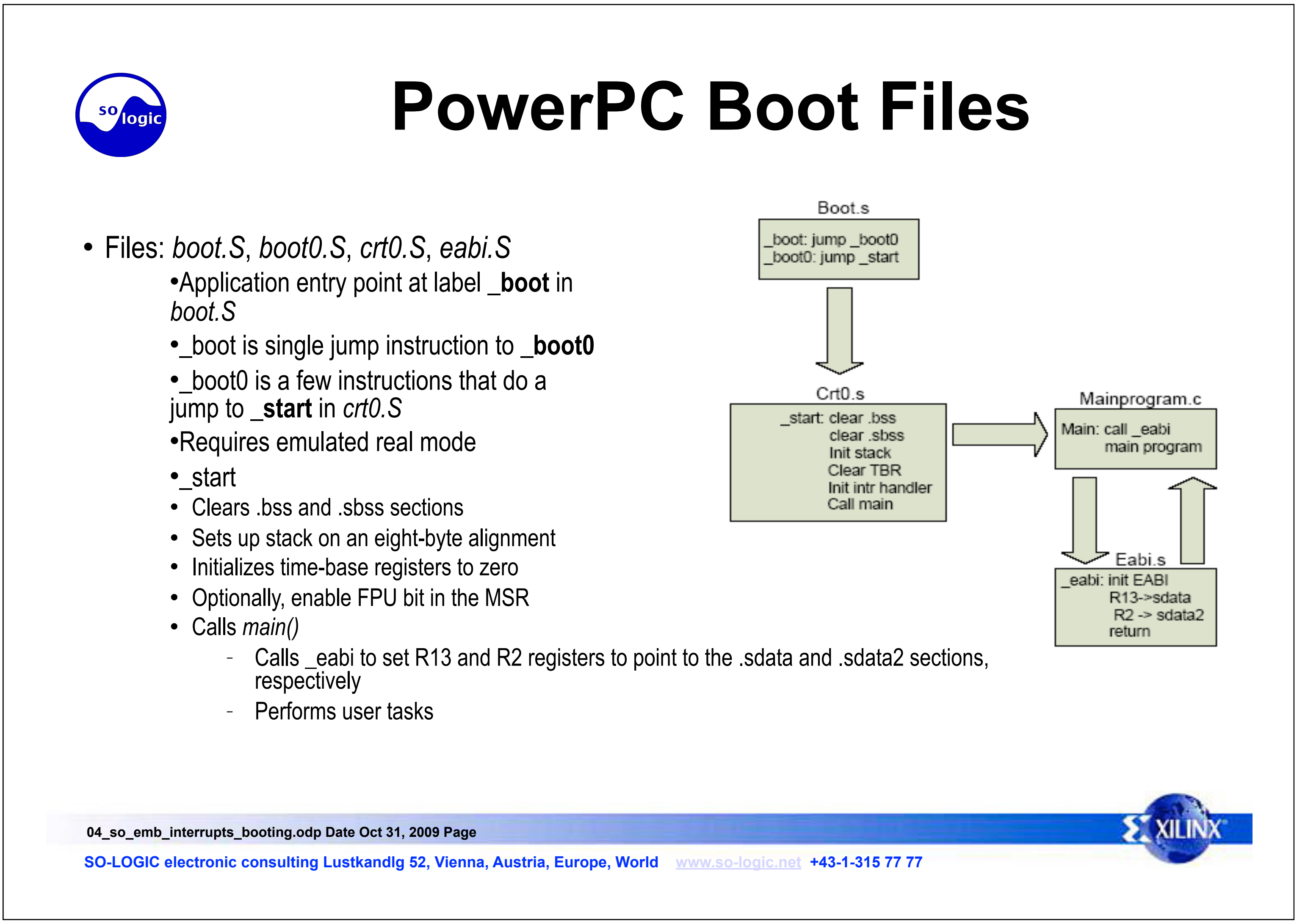


Initialization Requirements

- Initialize various control registers
- Invalidate caches
- Initialize on-core elements
  - •Timers
  - •MMU
  - •Debug facilities
- Initialize off-core elements
- Start OS or application code
- Details available in Chapter 10 of the *PowerPC Processor Reference Guide*
- For the PowerPC 440 processor, the Xilinx boot creates a real address mode environment

Code automatically generated by EDK and placed in boot routine

The PowerPC 440 processor does not support real memory mode. Thus, the Xilinx boot includes MMU initialization to create a real mode mapping off memory at bootload.



PowerPC Boot Files

- Files: *boot.S*, *boot0.S*, *crt0.S*, *eabi.S*
  - •Application entry point at label _**boot** in *boot.S*
  - •_boot is single jump instruction to _**boot0**
  - •_boot0 is a few instructions that do a jump to _**start** in *crt0.S*
  - •Requires emulated real mode
  - •_start
  - Clears .bss and .sbss sections
  - Sets up stack on an eight-byte alignment
  - Initializes time-base registers to zero
  - Optionally, enable FPU bit in the MSR
  - Calls *main()*
    - Calls _eabi to set R13 and R2 registers to point to the .sdata and .sdata2 sections, respectively
    - Performs user tasks

Boot.s

_boot: jump _boot0
_boot0: jump _start

Crt0.s

_start: clear .bss
clear .sbss
Init stack
Clear TBR
Init intr handler
Call main

Mainprogram.c

Main: call _eabi
main program

Eabi.s

_eabi: init EABI
R13->sdata
R2 -> sdata2
return

The *boot.S*, *crt0.S*, and *eabi.S* files contain a minimal set of code for initializing the processor and starting an application.

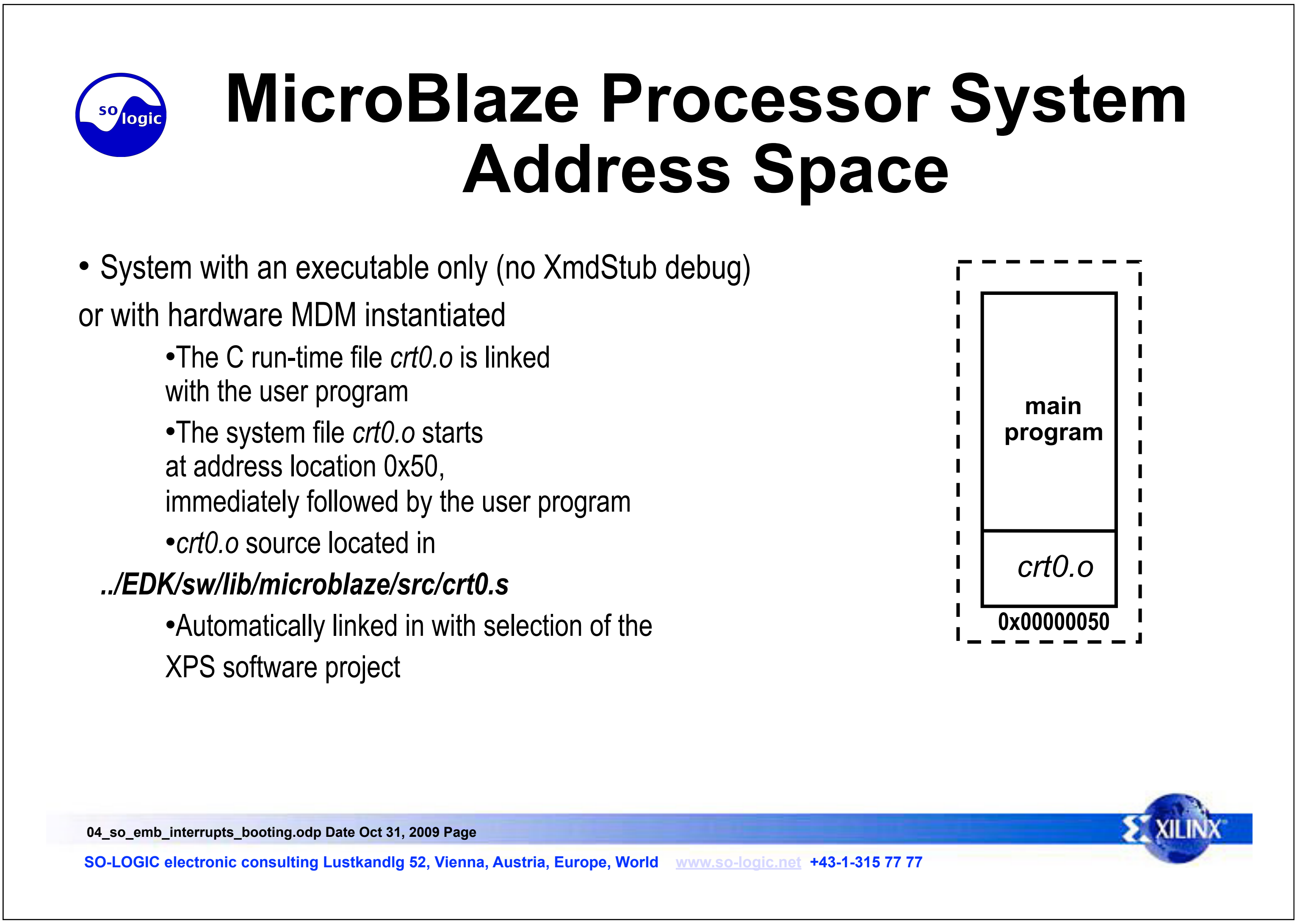
boot.S: Code in the *boot.S* file consists of the boot and boot0 sections. The boot section contains only one instruction, which is labeled with _boot. During the link process, this instruction is mapped to the reset vector, and the _boot label marks the application's entry point. The boot instruction is a jump to the _boot0 label. The _boot0 label must reside within a 23-bit address space of the _boot label. It is defined in the boot0 section. The code in the boot0 section calculates the 32-bit address of the _start label and jumps to it.

crt0.S: Code in the *crt0.S* file starts executing at the _start label. It initializes the .sbss and .bss sections to zero, as required by the ANSI C specification, sets up the stack, initializes some processor registers, and calls the *main()* function. The program remains in an endless loop on return from *main()*.

eabi.S: When an application is compiled and linked with the -msdata=eabi option, GCC inserts a call to the __eabi label at the beginning of the *main()* function. This is the place where register R13 must be set to point to the .sdata and .sbss data sections and register R2 must be set to point to the .sdata2 read-only data section. Code in the *eabi.S* file sets these two registers to the correct values. The _SDA_BASE_ and _SDA2_BASE_ labels are generated by the linker.

MicroBlaze Processor Boot File

- File: *crt0.S*
  - Application entry point at label _**start**
  - Called from reset vector at 0x00000000
  - _start
    - Set up required resets, interrupts, and exception vectors
    - Set up stack pointer, small data anchors, and other registers
    - Clear the BSS memory regions to zero
    - Invoke language initialization functions
    - Initialize interrupt handler and the hardware subsystem
    - Set up arguments for the main procedure
    - Call *main()*



MicroBlaze Processor System Address Space

- System with an executable only (no XmdStub debug) or with hardware MDM instantiated
  - •The C run-time file *crt0.o* is linked with the user program
  - •The system file *crt0.o* starts at address location 0x50, immediately followed by the user program
  - •*crt0.o* source located in ***../EDK/sw/lib/microblaze/src/crt0.s***
  - •Automatically linked in with selection of the XPS software project

Initialization files, such as *crt0.o*, are searched by the compiler only for mb-gcc. For powerpc-eabi-gcc, the C run-time library is a part of the library and is picked up by default from the *libxil.a* library.

This initialization file is used for programs which are to be executed in standalone mode without the use of any bootloader or debugging stubs, such as *xmdstub*. This CRT populates the reset, interrupt, exception, and hardware exception vectors and invokes the second stage startup routine _crtinit. On returning from _crtinit, it ends the program by infinitely looping in the exit label.

Programming Loading Requirements

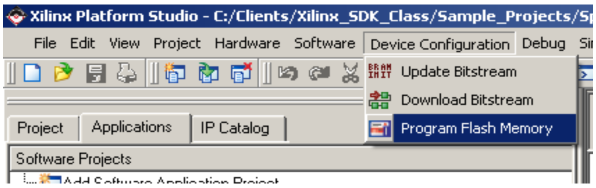
Software	Requirements
Standalone user application that resides in block RAM	Right-click the project and select **Mark to Initialize BRAMs** and then run the **Update Bitstream** command
Operating systems	Have their own proprietary requirements; typically handled by their installed control of LibGen to build the BSP, compile, and link
Large applications that cannot fit into block RAM and execute from external RAM (DDR2)	Bootloader to copy the program from another source

Alternative Boot Options

- Flash memory technology
  - Parallel flash – requires an External Memory Controller (EMC)
  - Compact Flash (CF) – requires a System ACE™ technology chip
  - SPI flash – requires the XPS_SPI_Interface controller
- Bus hardware boot options – secondary, follow-on boot
  - Serial using Uartlite
  - Ethernet
- Boot program is typically located in block RAM
  - Copies the application from the boot device to main system memory
  - When finished, turns control over to the new memory image

Parallel Flash Memory Boot

- Xilinx Platform Studio provides a Program Flash Memory dialog box for programming flash devices within XPS
- Supports flash devices that support Intel command sets
- Supports most all physical flash arrangements
- Supports an executable file as an input format and provides the option of converting it into SREC format

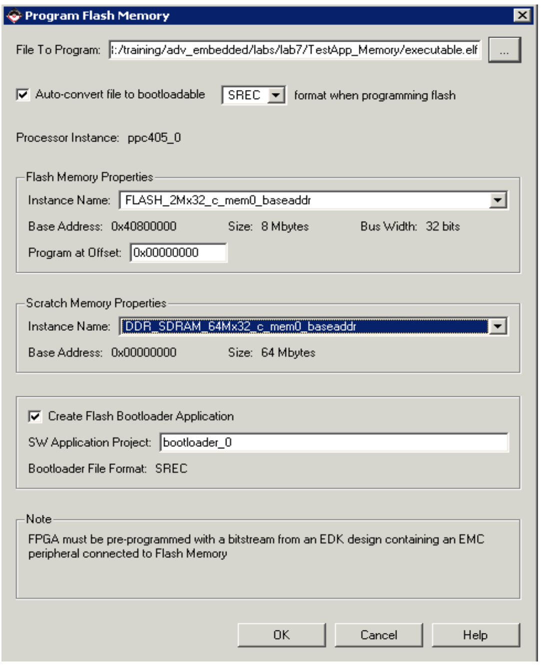

Xilinx Platform Studio - C:/Clients/Xilinx_SDK_Class/Sample_Projects/Sp
File Edit View Project Hardware Software Device Configuration Debug Si
Update Bitstream
Download Bitstream
Program Flash Memory
Project Applications IP Catalog
Software Projects
Add Software Application Project

- Supports flash bootloader creation
- Flash devices interface through the EMC peripheral
- Requires XMD to execute a Tcl file to perform programming and verification functions

SREC refers to the Motorola S-Record file format, and is a standard for representing memory data and regions as ASCII text. The SREC format has several advantages over binary formats. The ASCII encoding allows the files to be edited with a text editor. Also, each record contains a checksum to identify data that has been corrupted during transmission.

Flash Writer Utility

- Object file to be programmed
- Object in ELF or SREC format
- Hardware instance of flash memory
  - Base address, size, and data width automatically determined from the MHS file
- Storage offset from beginning of flash memory – multiple images!
- Programming scratchpad RAM is required
- Separate block RAM bootload application can automatically be created



Program Flash Memory

File To Program: i:/training/adv_embedded/labs/lab7/TestApp_Memory/executable.elf

☑ Auto-convert file to bootloadable SREC format when programming flash

Processor Instance: ppc405_0

Flash Memory Properties

Instance Name: FLASH_2Mx32_c_mem0_baseaddr

Base Address: 0x40800000 Size: 8 Mbytes Bus Width: 32 bits

Program at Offset: 0x00000000

Scratch Memory Properties

Instance Name: DDR_SDRAM_64Mx32_c_mem0_baseaddr

Base Address: 0x00000000 Size: 64 Mbytes

☑ Create Flash Bootloader Application

SW Application Project: bootloader_0

Bootloader File Format: SREC

Note

FPGA must be pre-programmed with a bitstream from an EDK design containing an EMC peripheral connected to Flash Memory

In the Xilinx Platform Studio Program Flash Memory dialog box, you can:

•Select the image to store on flash (ELF or SREC)

•Set the file format conversion if the source file is in ELF format

•Select the flash programming mode

•Select a processor instance in the design

•Select the flash memory and offset

•Select a section of memory in the design as scratch memory; must be at least 32 kB

•Select the bootloader creation flag if desired. You can have multiple bootloader applications in the Application tab, depending on how many different programs have been flashed—each starting at different addresses in flash. Make sure that only one bootloader application is marked to initialize block RAM when using to bootload the system.

Supported flash configurations:

•Single 16-bit device forming a 16-bit data bus

•Paired 8-bit devices forming a 16-bit data bus

•Single 32-bit device forming a 32-bit data bus

•Paired 16-bit devices forming a 32-bit data bus

•Four 8-bit devices forming a 32-bit data bus

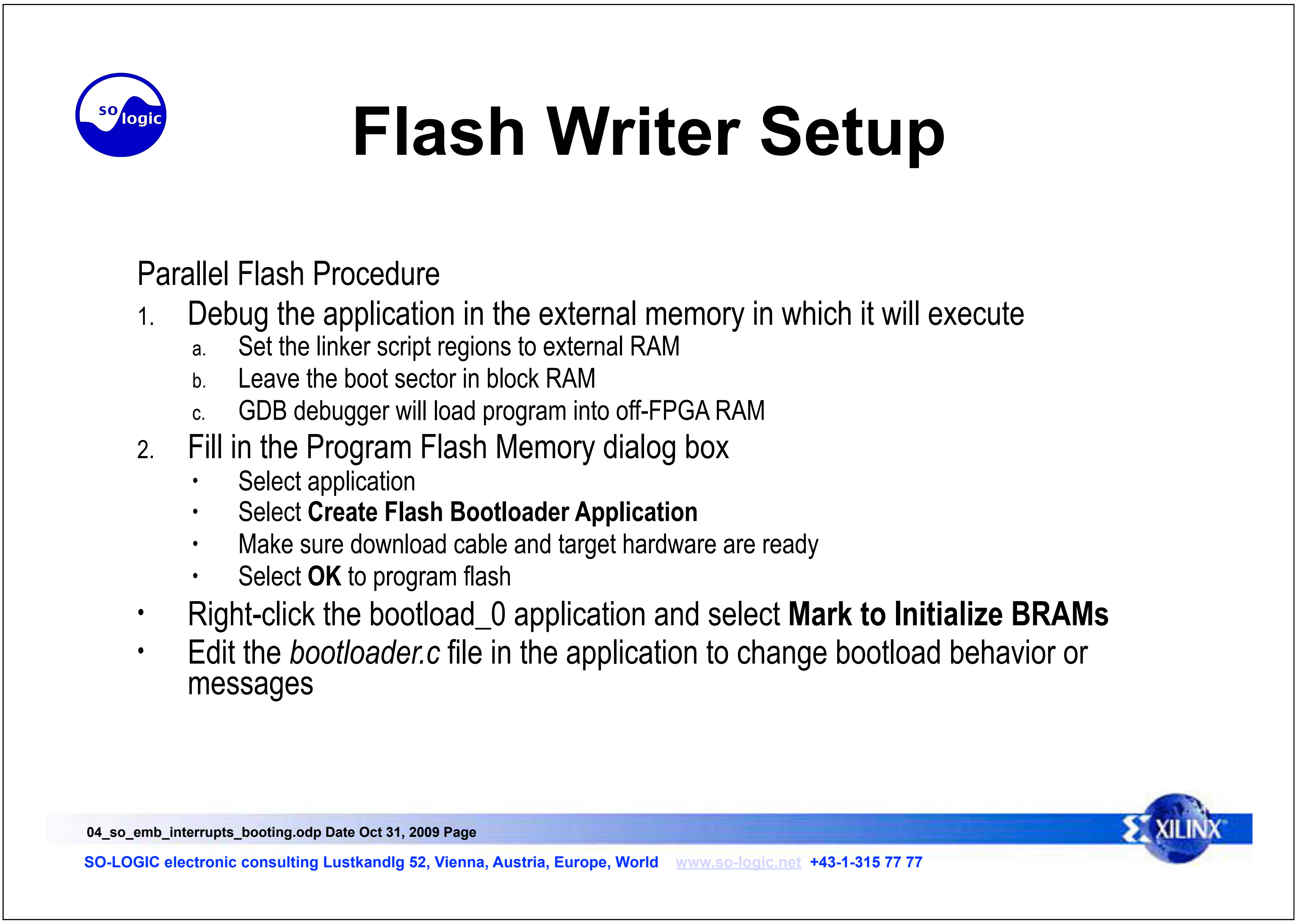
Support for command sets for the following devices:

•Intel/Sharp extended command set

•AMD/Fujitsu standard command set

•Intel standard command set

•AMD/Fujitsu extended command set



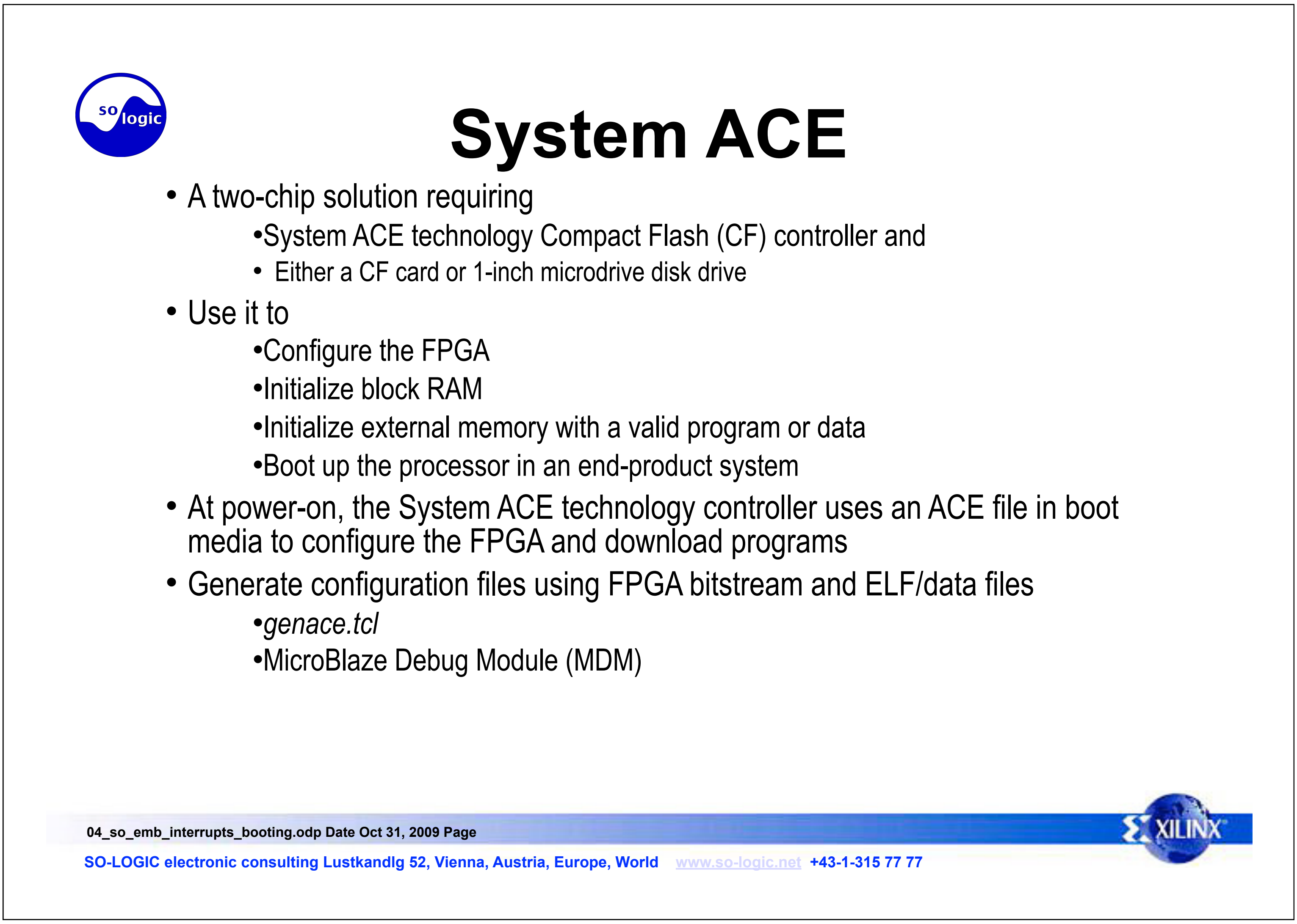
Flash Writer Setup

Parallel Flash Procedure

1. Debug the application in the external memory in which it will execute
   a. Set the linker script regions to external RAM
   b. Leave the boot sector in block RAM
   c. GDB debugger will load program into off-FPGA RAM
2. Fill in the Program Flash Memory dialog box
   - Select application
   - Select **Create Flash Bootloader Application**
   - Make sure download cable and target hardware are ready
   - Select **OK** to program flash

- Right-click the bootload_0 application and select **Mark to Initialize BRAMs**
- Edit the *bootloader.c* file in the application to change bootload behavior or messages

The *bootloader.c* file outputs to the standard out UART peripheral as it is executing.

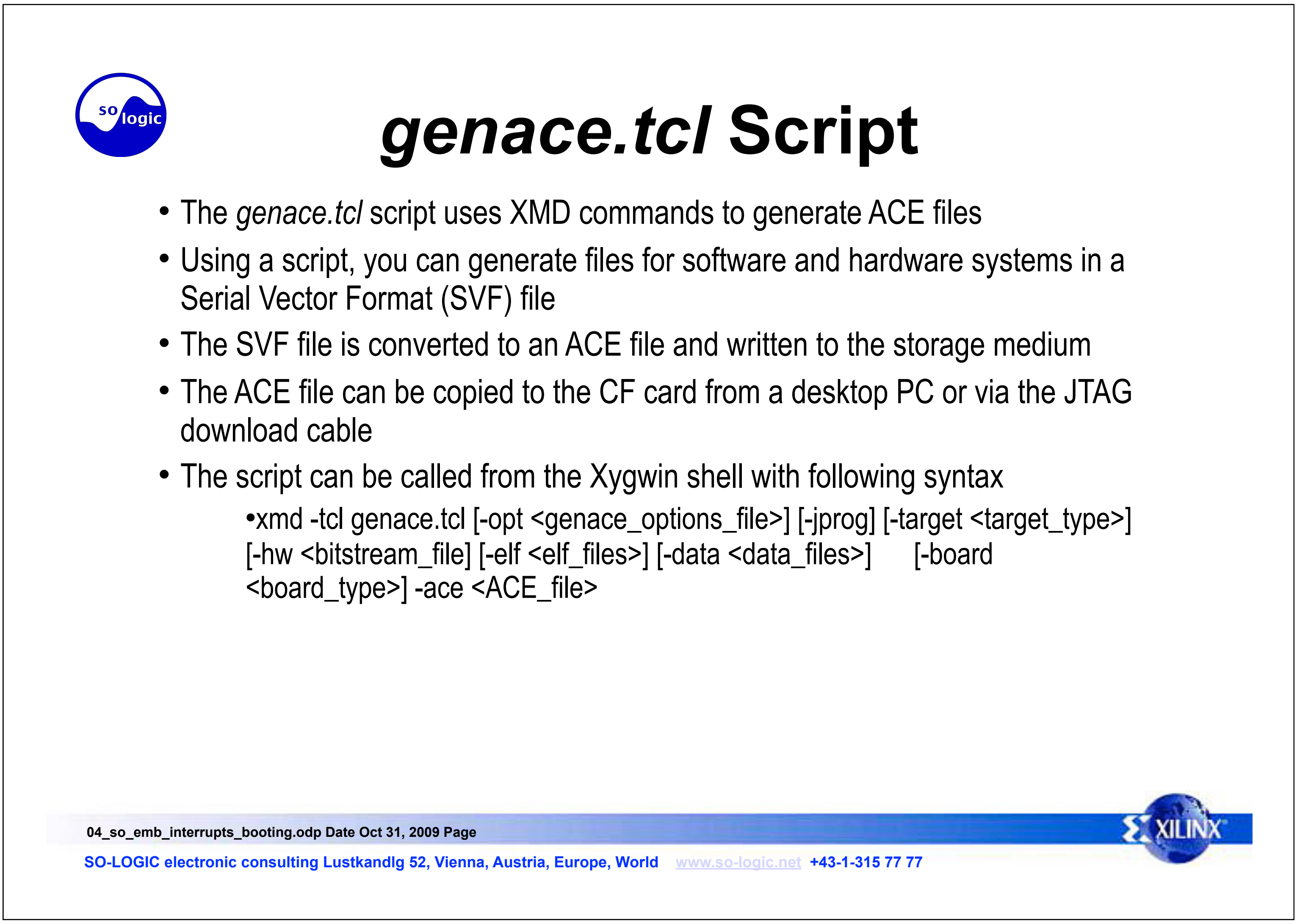


System ACE

- A two-chip solution requiring
  - System ACE technology Compact Flash (CF) controller and
  - Either a CF card or 1-inch microdrive disk drive
- Use it to
  - Configure the FPGA
  - Initialize block RAM
  - Initialize external memory with a valid program or data
  - Boot up the processor in an end-product system
- At power-on, the System ACE technology controller uses an ACE file in boot media to configure the FPGA and download programs
- Generate configuration files using FPGA bitstream and ELF/data files
  - *genace.tcl*
  - MicroBlaze Debug Module (MDM)

On power on, the System ACE technology controller looks for an ACE file in the *Xilinx.sys* directory on the boot media, configures the FPGA, and downloads any programs into memory.

Configuration files must be generated using FPGA bitstream and ELF/data files. EDK provides a Tcl script (*genace.tcl*), which uses XMD commands for this purpose. MicroBlaze-processor systems must use the MDM.



genace.tcl Script

- The *genace.tcl* script uses XMD commands to generate ACE files
- Using a script, you can generate files for software and hardware systems in a Serial Vector Format (SVF) file
- The SVF file is converted to an ACE file and written to the storage medium
- The ACE file can be copied to the CF card from a desktop PC or via the JTAG download cable
- The script can be called from the Xygwin shell with following syntax

```
•xmd -tcl genace.tcl [-opt <genace_options_file>] [-jprog] [-target <target_type>]
[-hw <bitstream_file] [-elf <elf_files>] [-data <data_files>]      [-board
<board_type>] -ace <ACE_file>
```

GenACE options include:

-opt: GenACE options are read from the options file.

-jprog: Clear the existing FPGA configuration. This option should not be specified if you are performing run-time configuration.

-target: Target to use in the system for downloading the ELF/data file. Target types are: ppc_hw and mdm

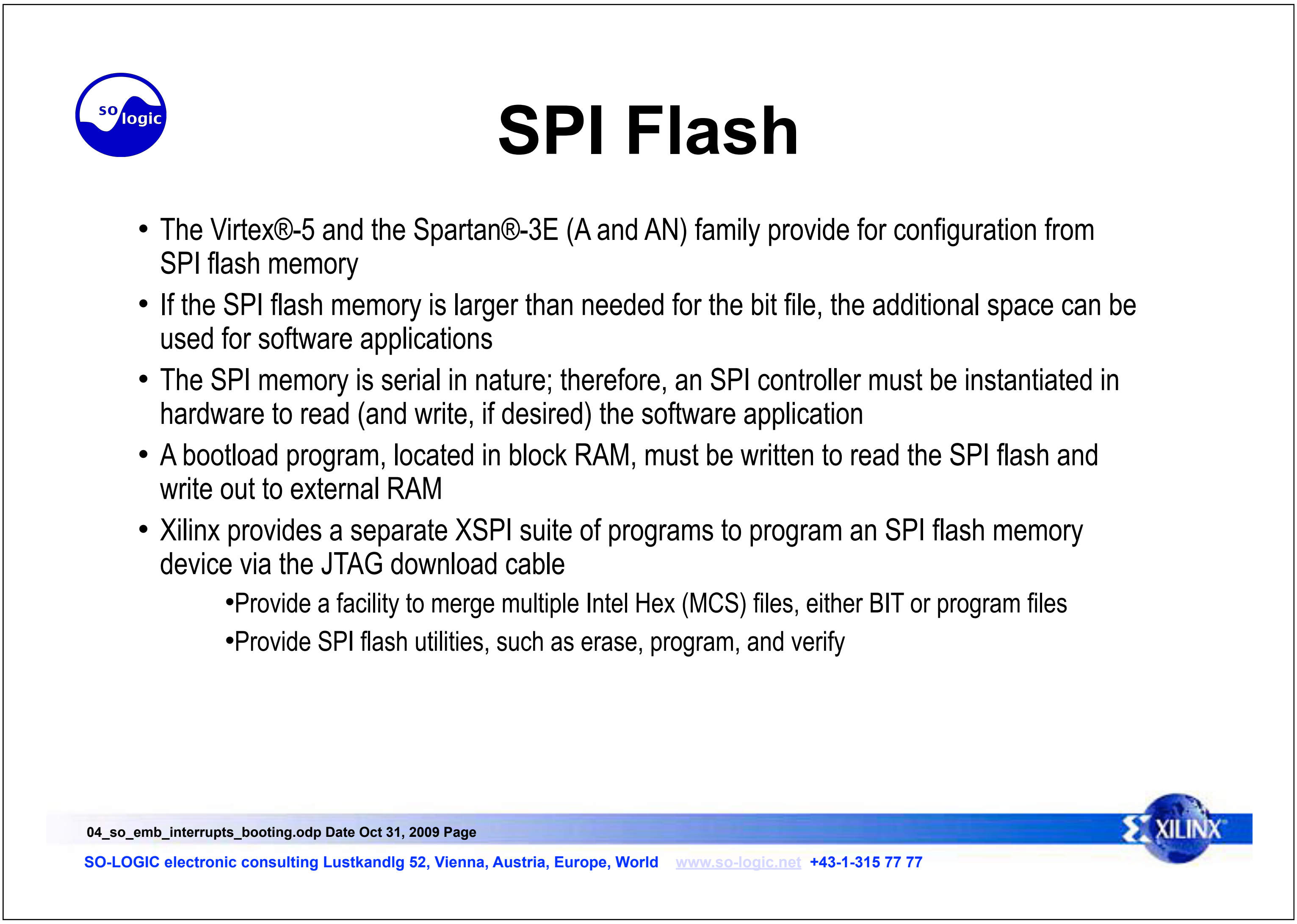
-hw: The bitstream file for the system. If an SVF file is specified, the SVF file is used.

-elf: List of data/binary file and its load address. The load address can be in decimal or hex format (0x prefix needed). If an SVF file is specified, it is used.

-data: This identifies the JTAG chain on the board (devices, IR length, or debug device, for example). The options are given with respect to the System ACE technology interface controller.

-board: The script contains options for some pre-defined boards. Board type options are: ml300 - ml300 board with the Virtex2P7 device; memec - Memec board with the Virtex2P4 device and P160; mbdemo - Xilinx MicroBlaze processor demo board Virtex 21000 device; and auto - Auto Detect Scan Chain and form options for any generic board. The board should be connected for this option. The GenACE options are written out as a *genace.opt* file. The user can use this file to generate an ACE file for the given system, user. The user specifies the -configdevice and -debugdevice option in the Options file.

-ace: The output ACE file. The file prefix should not match any of the input files (bitstream, ELF, data files) prefix.



SPI Flash

- The Virtex®-5 and the Spartan®-3E (A and AN) family provide for configuration from SPI flash memory
- If the SPI flash memory is larger than needed for the bit file, the additional space can be used for software applications
- The SPI memory is serial in nature; therefore, an SPI controller must be instantiated in hardware to read (and write, if desired) the software application
- A bootload program, located in block RAM, must be written to read the SPI flash and write out to external RAM
- Xilinx provides a separate XSPI suite of programs to program an SPI flash memory device via the JTAG download cable
  - •Provide a facility to merge multiple Intel Hex (MCS) files, either BIT or program files
  - •Provide SPI flash utilities, such as erase, program, and verify

The iMPACT software utility included with the ISE software allows the designer to perform program, erase, and verify operations to the SPI flash memory after it has been soldered to the PCB via a USB platform cable and a PC. The iMPACT software supports only select SPI flash memory families from Numonyx and Atmel. See the iMPACT software for the list of supported devices.

Spartan-3 AN FPGA flash:

•Resident internal SPI flash for configuration and software applications; a single-chip solution

•Supported in hardware with XPS_SPI_Interface

•Supported in BSP with the XILISP library (read/write SPI flash; initialize, erase, and other utilities)

•Bootload program support

•Application programs data into the flash memory either directly to the main flash memory array or through one of the SRAM page buffers

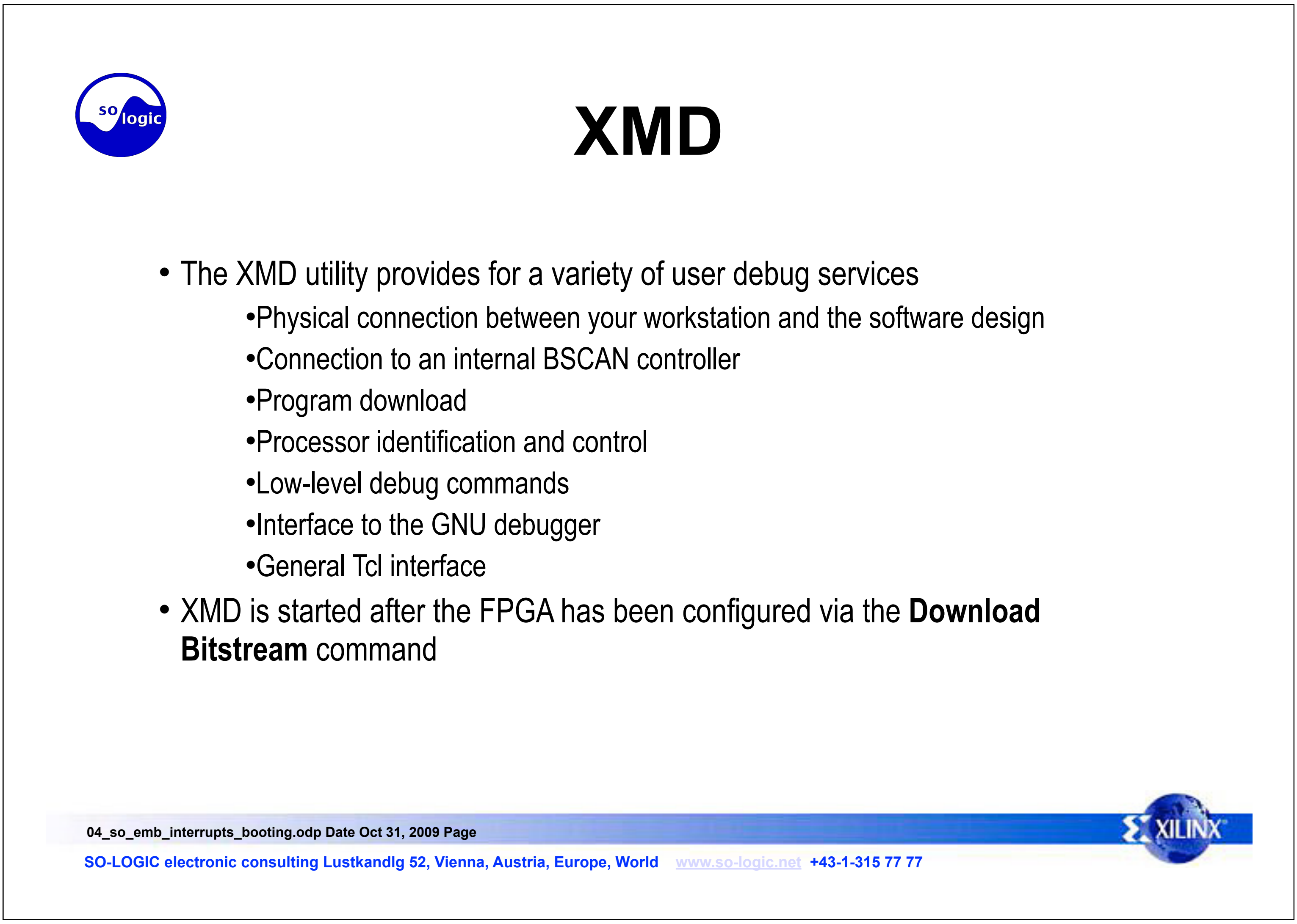
For more information about SPI Flash memory programming, see:

•Application Note XAPP1034: *Reference System: Accessing Spartan-3AN In-System Flash Using XPS*

•Application Note XAPP1053, *Flash Memory Bootloading Using SPI with Spartan-3A DSP 1800A Starter Platform*

•Application Note XAPP800, *Configuring Xilinx FPGAs with SPI Flash Memories Using CoolRunner-II CPLDs*

•*Spartan-3AN In-System Flash User Guide* (www.xilinx.com/support/documentation/user_guides/ug333.pdf)



XMD

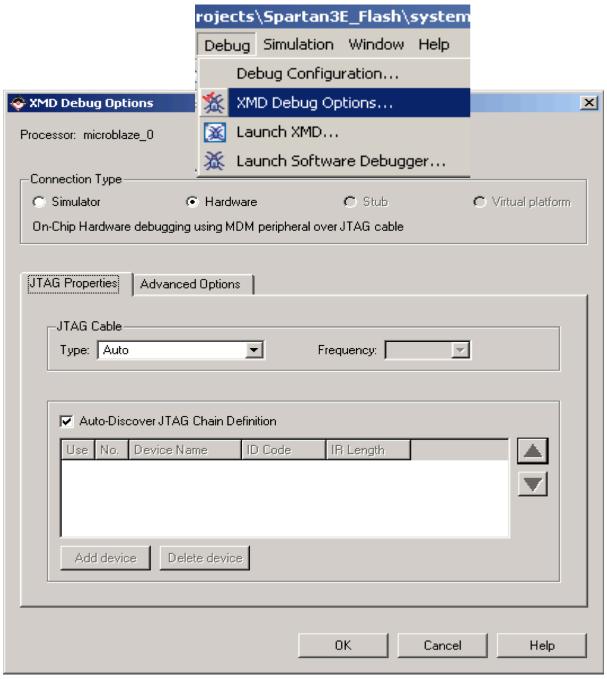
- The XMD utility provides for a variety of user debug services
  - Physical connection between your workstation and the software design
  - Connection to an internal BSCAN controller
  - Program download
  - Processor identification and control
  - Low-level debug commands
  - Interface to the GNU debugger
  - General Tcl interface
- XMD is started after the FPGA has been configured via the **Download Bitstream** command

For more information, see the GDB documentation for downloading, running, and debugging.

Note: GDB requires XMD to be running in order to provide a connection to the hardware and software.

Setting XMD Options

- XMD accommodates various functions
  - Software simulator for the MicroBlaze processor*
  - Connection to hardware via a JTAG cable
  - XMDstub, a software intrusive debugger
- Download cable support
  - Parallel and USB
  - Auto is a good selection
- JTAG chain
  - Support for custom devices
  - Auto is also a good selection



rojects\Spartan3E_Flash\system
Debug Simulation Window Help
Debug Configuration...
XMD Debug Options...
Launch XMD...
Launch Software Debugger...
XMD Debug Options
JTAG Properties
Advanced Options
JTAG Cable
Auto-Discover JTAG Chain Definition
Use No. Device Name ID Code IR Length
Add device
Delete device
OK
Cancel
Help

Processor: microblaze_0

Connection Type

○ Simulator ● Hardware ○ Stub ○ Virtual platform

On-Chip Hardware debugging using MDM peripheral over JTAG cable

Type: Auto

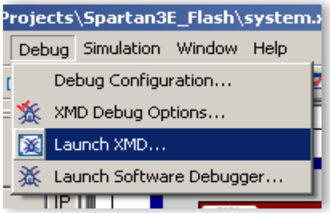
Frequency:

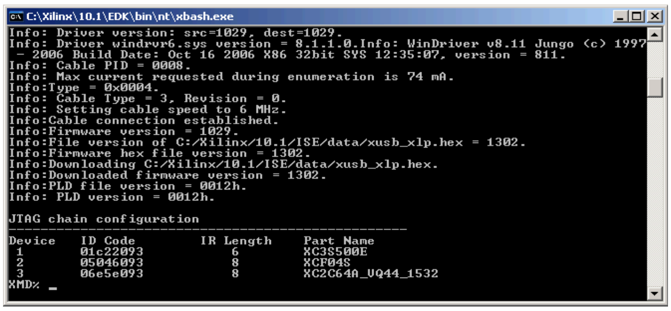
☑ Auto-Discover JTAG Chain Definition

*The PowerPC processor simulator is available through the SDK.

Launching XMD

- Once launched, XMD looks for the download cable and a hardware target on the other side of the cable
- The console screen to the right, shows detection of a USB cable and discovery of three Xilinx parts
- The XMD% command prompt awaits the user to enter commands
- Multiple XMD sessions can be opened simultaneously





```
Info: Driver version: src=1029, dest=1029.
Info: Driver windrvr6.sys version = 8.1.1.0.Info: WinDriver v8.11 Jungo (c) 1997
 - 2006 Build Date: Oct 16 2006 X86 32bit SYS 12:35:07, version = 811.
Info: Cable PID = 0008.
Info: Max current requested during enumeration is 74 mA.
Info:Type = 0x0004.
Info: Cable Type = 3, Revision = 0.
Info: Setting cable speed to 6 MHz.
Info:Cable connection established.
Info:Firmware version = 1029.
Info:File version of C:/Xilinx/10.1/ISE/data/xusb_xlp.hex = 1302.
Info:Firmware hex file version = 1302.
Info:Downloading C:/Xilinx/10.1/ISE/data/xusb_xlp.hex.
Info:Downloaded firmware version = 1302.
Info:PLD file version = 0012h.
Info: PLD version = 0012h.

JTAG chain configuration
--------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       01c22093           6        XC3S500E
 2       05046093           8        XCF04S
 3       06e5e093           8        XC2C64A_VQ44_1532
XMD% _
```

Popular XMD commands for boot and program control include:

•connect – connect to the MicroBlaze or PowerPC processor debug module

•dow – download the ELF executable file

•elf_verify – verify the ELF file with memory image

•run – begin program execution from reset

•con – continue program execution from the current program counter

•stop – stop the target processor

•exit – close the XMD window

XMD will search for a processor when started. The connect command will execute automatically and the software application in block RAM will begin to execute.

Remember to first download the bit file via the Download Bitstream command to configure the FPGA before starting XMD.

Summary

- Smaller target software application program projects that reside entirely in FPGA block RAM boot and execute normally when the project is marked to initialize block RAM
- Bootloops are used to maintain processor behavior until the debugger can take control
- Processor reset vectors must have a boot instruction in place. This is typically handled by the tools and resides in block RAM
- C programs require that the processor environment is set up before beginning to execute *main{}*. This is accomplished automatically by the tools with the inclusion of the *crt0.o* module

Apply Your Knowledge

Q1) What does a linker script do?

Q2) How do you generate a linker script?

Q3) When do you need to write your own linker script?

Apply Your Knowledge

Q4) Describe the steps involved when registering a single interrupt source

Q5) Describe the steps involved when registering multiple interrupt sources with an interrupt controller

Apply Your Knowledge

Q6) What is the bootloop program and why is it used?

Q7) What is the bootload process by which a software application executes from off-chip DDR2 RAM, but the application itself resides in off-chip parallel flash?