Real Time Embedded Systems

"System On Programmable Chip"

NIOSII Architecture & Interrupt services
Some Avalon Peripherals

René Beuchat

Laboratoire d'Architecture des Processeurs

rene.beuchat@epfl.ch

NIOS II

- General Features
- Embedded system NIOSII/Avalon Architecture
- NIOS II Core Architecture
- NIOS II Memory Architecture
- Programming Model Registers
- Exceptions Model
- Instructions
- Custom Instructions

NIOS II

- Some Avalon Peripherals:
  - PIO
  - Timer
  - Performance Counter

NIOS II - General Features

The Nios II processor is a general-purpose RISC processor core, providing:

- Full 32-bit instruction set, data path, and address space
- 32 general-purpose registers
- 32 external interrupt sources
- Single-instruction 32 × 32 multiply and divide producing a 32-bit result
- Dedicated instructions for computing 64-bit and 128-bit products of multiplication
- Floating-point instructions for single-precision floating-point operations
- Single-instruction barrel shifter

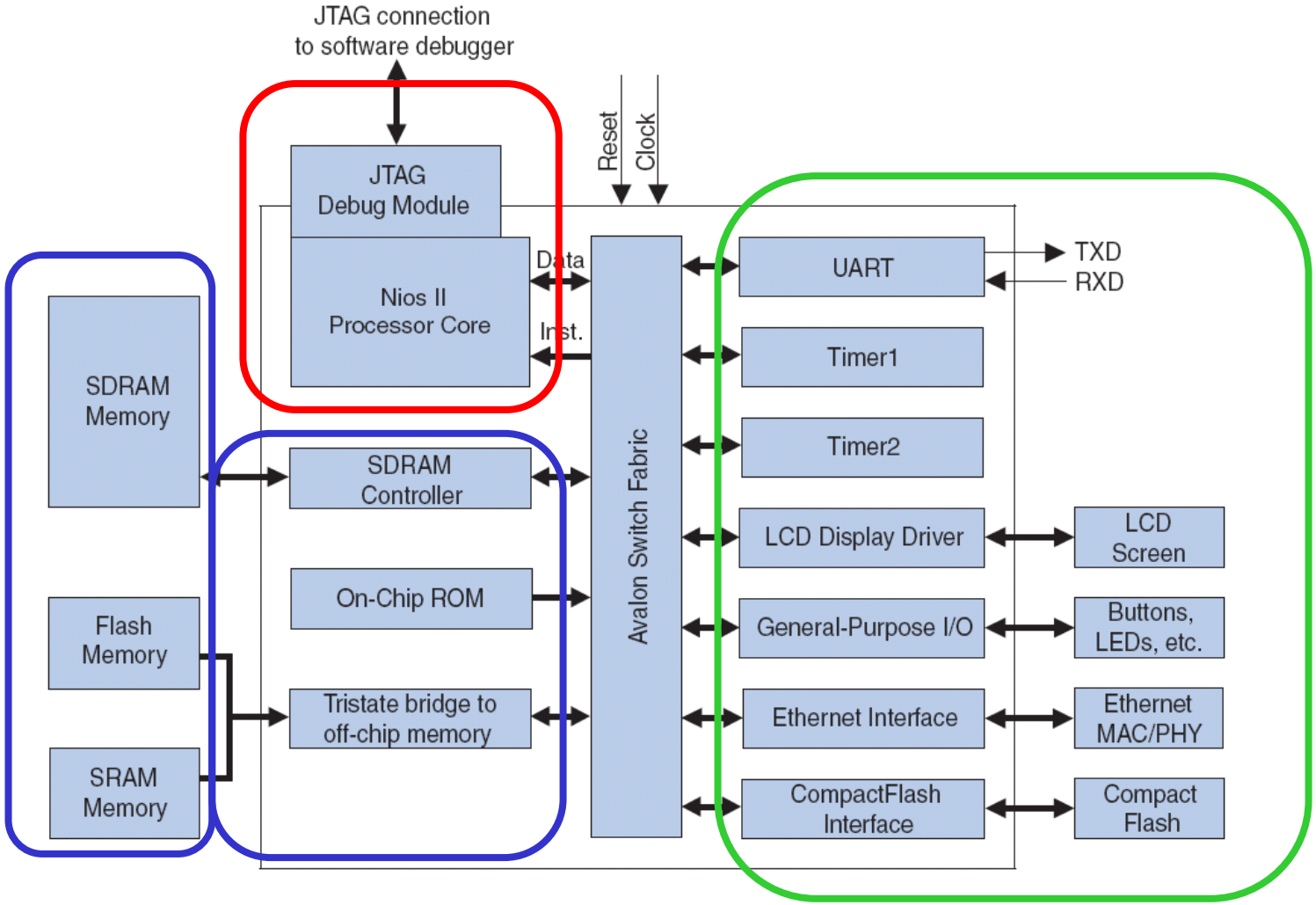
NIOS II - General Features

- Access to a variety of on-chip peripherals, and interfaces to off-chip memories and peripherals
- Hardware-assisted debug module enabling processor start, stop, step and trace under integrated development environment (IDE) control
- Software development environment based on the GNU C/C++ tool chain and Eclipse IDE
- Integration with Altera's SignalTap® II logic analyzer, enabling real-time analysis of instructions and data long with other signals in the FPGA design
- Instruction set architecture (ISA) compatible across all Nios II processor systems

NIOS II - General Features

- A **Nios II processor system** is equivalent to a microcontroller or “computer on a chip” that includes a CPU and a combination of peripherals and memory on a single chip.
- The term “**Nios II processor system**” refers to a Nios II processor core, a set of on-chip peripherals, on chip memory, and interfaces to off-chip memory, all implemented on a single Altera® chip.
- Like a microcontroller family, all Nios II processor systems use a consistent instruction set and programming model.

NIOS II –
Embedded system NIOSII/Avalon Architecture



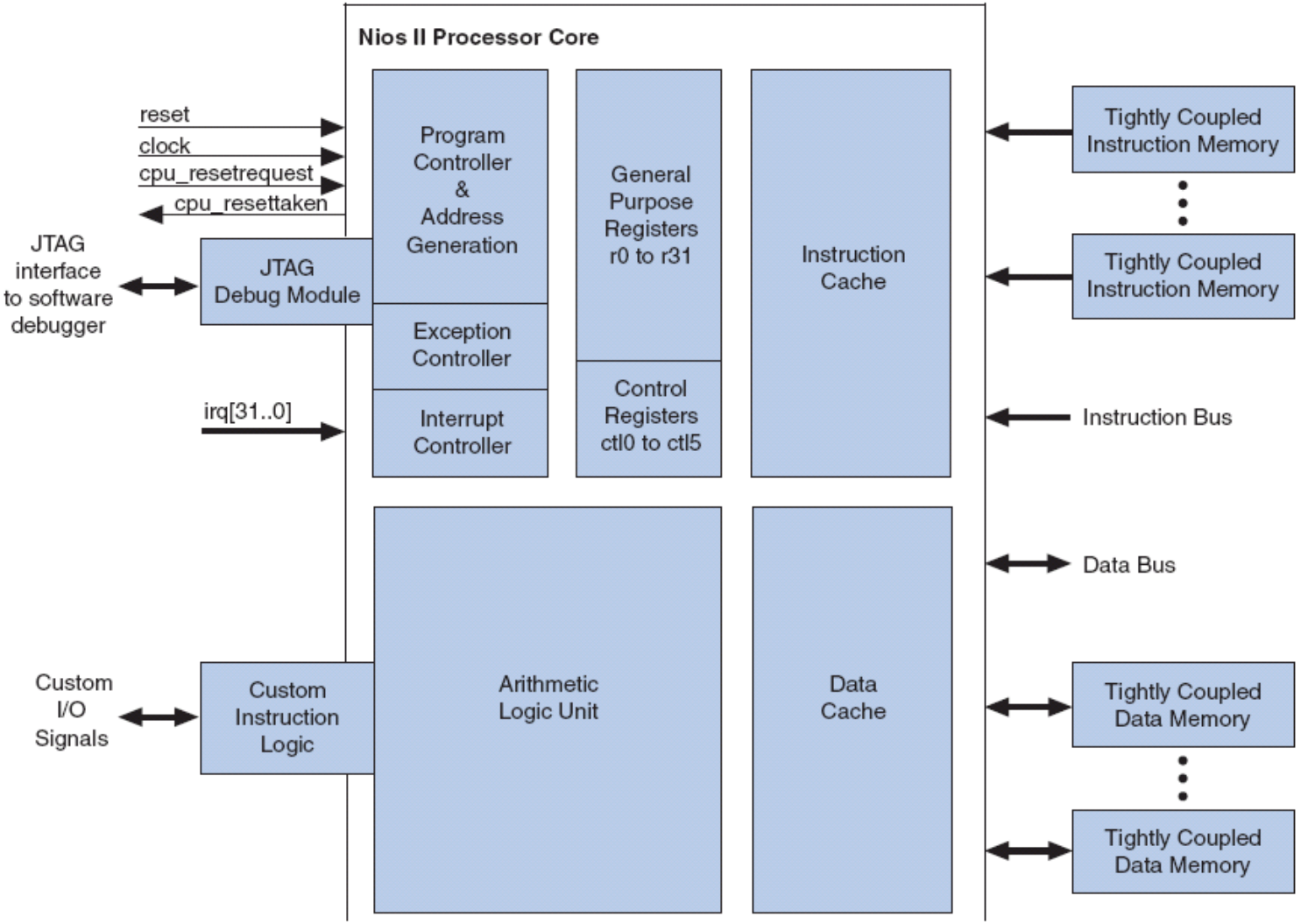
Implementation variables generally fit one of three trade-off patterns:

- more-or-less of a feature;
- inclusion-or-exclusion of a feature;
- Hardware implementation or software emulation of a feature.

An example of each trade-off follows:

- ***More or less of a feature*** —For example, to fine-tune performance, you can increase or decrease the amount of instruction cache memory. A larger cache increases execution speed of large programs, while a smaller cache conserves on-chip memory resources.
- ***Inclusion or exclusion of a feature*** —For example, to reduce cost, you can choose to omit the JTAG debug module. This decision conserves onchip logic and memory resources, but it eliminates the ability to use a software debugger to debug applications.
- ***Hardware implementation or software emulation*** —For example, in control applications that rarely perform complex arithmetic, you can choose for the division instruction to be emulated in software. Removing the divide hardware conserves on-chip resources but increases the execution time of division operations.

NIOS II – Core Architecture



NIOS II – Core Architecture

The Nios II architecture defines the following user-visible functional units:

- Register file, r0..r31, ctl0..ctl5
- Arithmetic logic unit
- Interface to custom instruction logic
- **NO FLAGS (N, Z, V, C) available**
- Exception controller
- Interrupt controller
- Instruction bus
- Data bus
- Instruction and data cache memories
- Tightly coupled memory interfaces for instructions and data
- JTAG debug module

NIOS II –
Programming Model Registers

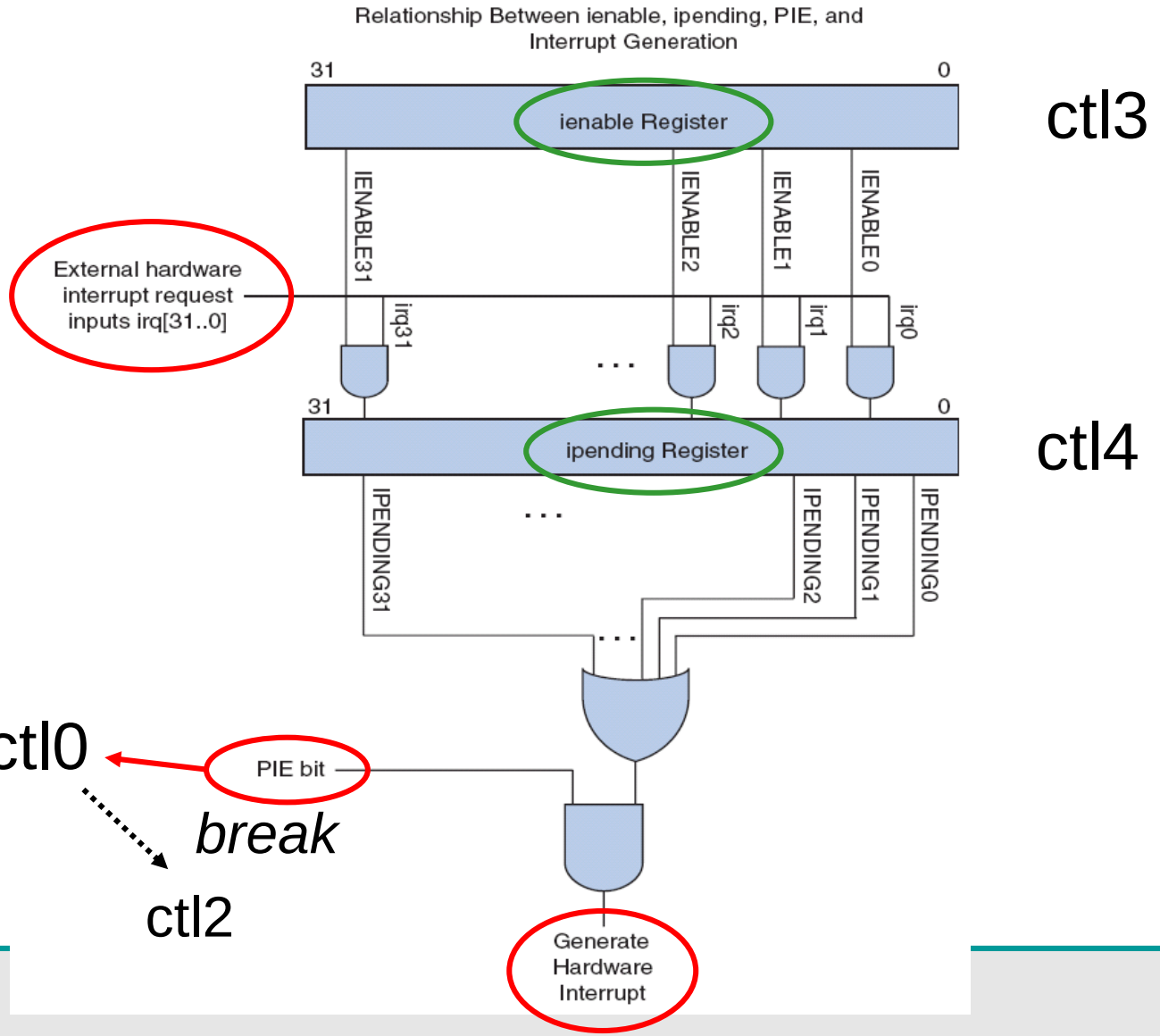
Register	Name	Function	Register	Name	Function
r0	zero	0x00000000	r16		
r1	at	Assembler Temporary	r17		
r2		Return Value	r18		
r3		Return Value	r19		
r4		Register Arguments	r20		
r5		Register Arguments	r21		
r6		Register Arguments	r22		
r7		Register Arguments	r23		
r8		Caller-Saved Register	r24	et	Exception Temporary
r9		Caller-Saved Register	r25	bt	Breakpoint Temporary *(1)*
r10		Caller-Saved Register	r26	gp	Global Pointer
r11		Caller-Saved Register	r27	sp	Stack Pointer
r12		Caller-Saved Register	r28	fp	Frame Pointer
r13		Caller-Saved Register	r29	ea	Exception Return Address
r14		Caller-Saved Register	r30	ba	Breakpoint Return Address *(1)*
r15		Caller-Saved Register	r31	ra	Return Address

Notes to Table 3–1:
(1) This register is used exclusively by the JTAG debug module.

Control registers are accessed differently than the general-purpose registers. The special instructions ***rdctl*** and ***wrctl*** provide the only means to read and write to the control registers.

Register	Name	31…1	0
ctl0	status	Reserved	PIE
ctl1	estatus	Reserved	EPIE
ctl2	bstatus	Reserved	BPIE
ctl3	ienable	Interrupt-enable bits	
ctl4	ipending	Pending-interrupt bits	
ctl5	cpuid	Unique processor identifier	

NIOS II – Interruptions (hardware) at processor level



ctl3

ctl4

ctl0

Inter.

break

ctl1

ctl2

status (ctl0)

- The value in the ***status register*** controls the state of the Nios II processor. All status bits are cleared after processor reset.
- ***PIE***: ***Processor Interrupt Enable*** bit

Bit	Description
PIE bit	PIE is the processor interrupt-enable bit. When PIE is 0, external interrupts are ignored. When PIE is 1, external interrupts can be taken, depending on the value of the `ienable` register.

estatus (ctl1)

- The ***estatus register*** holds a saved copy of the status register during exception processing. One bit is defined: **EPIE**. This is the saved values of PIE.
- The exception handler can examine **estatus** to determine the **preexception** status of the processor. When returning from an interrupt, the ***eret*** instruction causes the processor to copy ***estatus*** back to status, restoring the pre-exception value of status.

bstatus (ctl2)

- The ***bstatus*** register holds a saved copy of the status register during ***debug break processing***. One bit is defined: **BPIE**. This is the saved value of PIE.
- When a break occurs, the value of the **status** register is copied into **bstatus**. Using **bstatus**, the status register can be restored to the value it had prior to the break.

ienable (ctl3)

- The ***ienable register*** controls the handling of external hardware interrupts.
- Each bit of the ***ienable*** register corresponds to one of the interrupt inputs, **irq0** through **irq31**.
- A bit value of **1** means that the corresponding interrupt is **enabled**; 0 → interrupt is disabled.

ipending (ctl4)

- The value of the ***ipending register*** indicates which interrupts are pending.
- A value of 1 in bit *n* means that the corresponding **irq*n*** input is asserted, and that the corresponding interrupt is enabled in the ***ienable*** register.
- The effect of writing a value to the ***ipending register*** is undefined.

cpuid (ctl5)

- The ***cpuid register*** holds a static value that uniquely identifies the processor in a multi-processor system.
- The ***cpuid*** value is determined at system generation time.
- Writing to the ***cpuid*** register has no effect.

NIOS II – Exceptions Model

Exception Controller

- The Nios II architecture provides a simple, **non-vectored exception**
- controller to handle all exception types. All exceptions, including hardware interrupts, cause the processor to transfer execution to a ***single exception address***. The exception handler at this address determines the cause of the exception and dispatches an appropriate exception routine.
- The exception address is specified at system generation time.

Exception Types

- Nios II exceptions fall into the following categories:
  - Hardware interrupt
  - Software trap
  - Unimplemented instruction
  - Other

Integral Interrupt Controller

- The Nios II architecture supports 32 external hardware interrupts.
- The processor core has 32 level-sensitive interrupt request (IRQ) inputs, irq0 through irq31, providing a unique input for each interrupt source.
- IRQ priority is determined by software. The architecture supports nested interrupts.

NIOS II – Interruptions (hardware)

- The software can enable and disable any interrupt source individually through the ***ienable control register***, which contains an interrupt-enable bit for each of the IRQ inputs.
- Software can enable and disable interrupts globally using the PIE bit of the status control register.
- A hardware interrupt is generated if and only if **all three** of these conditions are true:
  - The ***PIE*** bit of the status register (ctl0) is 1
  - An interrupt-request input, irq<*n*>, is asserted
  - The corresponding bit *n* of the ***ienable*** register (ctl3) is 1
- The interrupt handler has to read the **ipendig** (ctl4) register to determine the interrupting source

Interrupt Vector Custom Instruction

→Obsolete,

→now VIC (Vector Interrupt Controller)

- The Nios II processor core offers an interrupt vector custom instruction which accelerates interrupt vector dispatch.
- Include this custom instruction to reduce program's interrupt latency.

NIOS II – Interruptions (hardware)

- The interrupt vector custom instruction is based on a priority encoder with one input for each interrupt connected to the Nios II processor.
- The cost of the interrupt vector custom instruction depends on the number of interrupts connected to the Nios II processor.
- The worse case is a system with 32 interrupts. In this case, the interrupt vector custom instruction consumes about **50 logic elements** (LEs).

NIOS II – Interruptions (hardware), ISR

- A software exception routine determines which of the pending interrupts has the highest priority, and then transfers control to the appropriate ***Interrupt Service Routine (ISR)*.**
- The ISR must stop the interrupt from being visible (either by clearing it at the source or masking it using ienable) before returning and/or before re-enabling PIE.
- The ISR must also save ***estatus*** (ctl1) and ***ea*** (*exception return address*, r29) before re-enabling PIE.

NIOS II – Exceptions

- Interrupts can be re-enabled by writing 1 to the ***PIE*** bit, thereby allowing the current ISR to be interrupted.
- Typically, the exception routine adjusts ***ienable*** so that IRQs of equal or lower priority are disabled before reenabling interrupts.

Software Trap

- When a program issues the trap instruction, it generates a software trap exception. A program typically issues a software trap when the program requires servicing by the operating system.
- The exception handler for the operating system determines the reason for the trap and responds appropriately.

Unimplemented Instruction

- When the processor issues a valid instruction that is not implemented in hardware, an unimplemented instruction exception is generated.
- The exception handler determines which instruction generated the exception.
- If the instruction is not implemented in hardware, control is passed to an exception routine that emulates the operation in software.

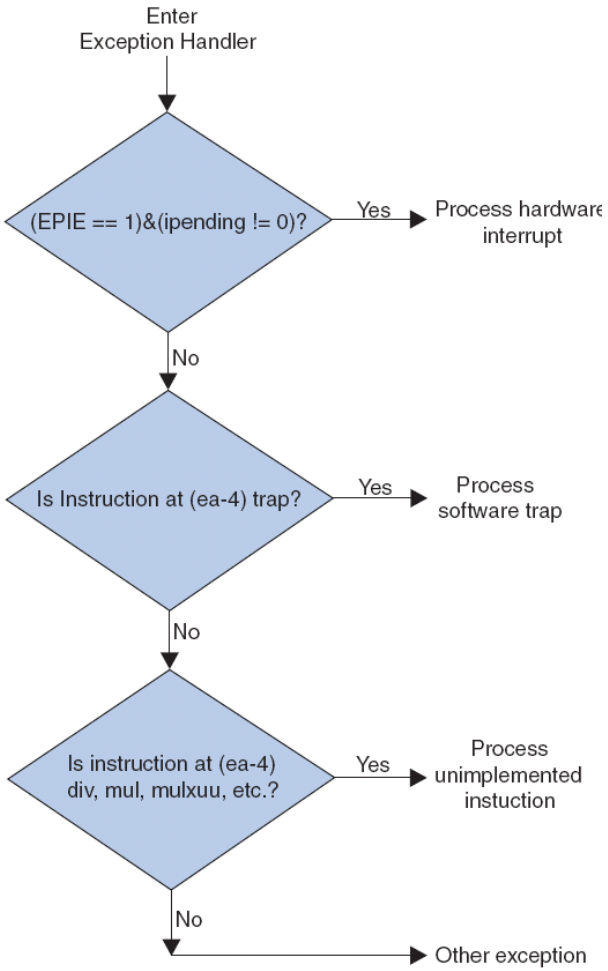
Other Exceptions

- The previous sections describe all of the exception types defined by the Nios II architecture at the time of publishing. However, some processor implementations might generate exceptions that do not fall into the above categories.
- For example, a future implementation might provide a memory management unit (MMU) that generates access violation exceptions. Therefore, a robust exception handler should provide a safe response (such as issuing a warning) in the event that it cannot exactly identify the cause of an exception.

Determining the Cause of Exceptions

- The exception handler must determine the cause of each exception and then transfer control to an appropriate exception routine.
- Remember: There is only one address for interrupts handler for all exceptions for the NIOS II processor → often the case for RISC processors.

NIOS II – Exceptions handling



NIOS II – Exceptions handling

- If the ***EPIE*** bit of the ***estatus*** register (ctl1) is 1 and the value of the ***ipending*** register (ctl4) is non-zero, the exception was caused by an **external hardware** interrupt.
- Otherwise, the exception might be caused by a ***software trap*** or an ***unimplemented instruction***. To distinguish between software traps and unimplemented instructions, **read the instruction at address ea–4** (the Nios II data master must have access to the code memory to read this address). If the instruction is ***trap***, the exception is a software trap. If the instruction at address ea-4 is one of the instructions that can be implemented in software, the exception was caused by an unimplemented instruction.
- If none of the above conditions apply, the exception type is unrecognized, and the exception handler should report the condition.

Nested Exceptions

- Exception routines must take special precautions before:
  - Issuing a trap instruction
  - Issuing an unimplemented instruction
  - Re-enabling hardware interrupts
- Before allowing any of these actions, the exception routine must save ***estatus*** (ctl1) and ***ea*** (r29), so that they can be restored properly before returning.

Returning from an Exception

- The ***eret*** instruction is used to resume execution from the pre-exception address. Except for the ***et*** register (r24), the exception routine must restore any registers modified during exception processing before returning.
- When executing the ***eret*** instruction, the processor:
- 1. Copies the contents of ***estatus*** (ctl1) to ***status*** (ctl0)
- 2. Transfers program execution to the address in the ***ea*** register (r29)

NIOS II – Exceptions handling

Return Address

- The return address requires some consideration when returning from exception processing routines. After an exception occurs, ea contains the address of the instruction *after* the point where the exception was generated.
- When returning from software trap and unimplemented instruction exceptions, execution must resume from the instruction following the software trap or unimplemented instruction. Therefore, ea contains the correct return address.
- On the other hand, hardware interrupt exceptions must resume execution from the interrupted instruction itself. In this case, the exception handler **must subtract 4** from ea to point to the interrupted instruction.

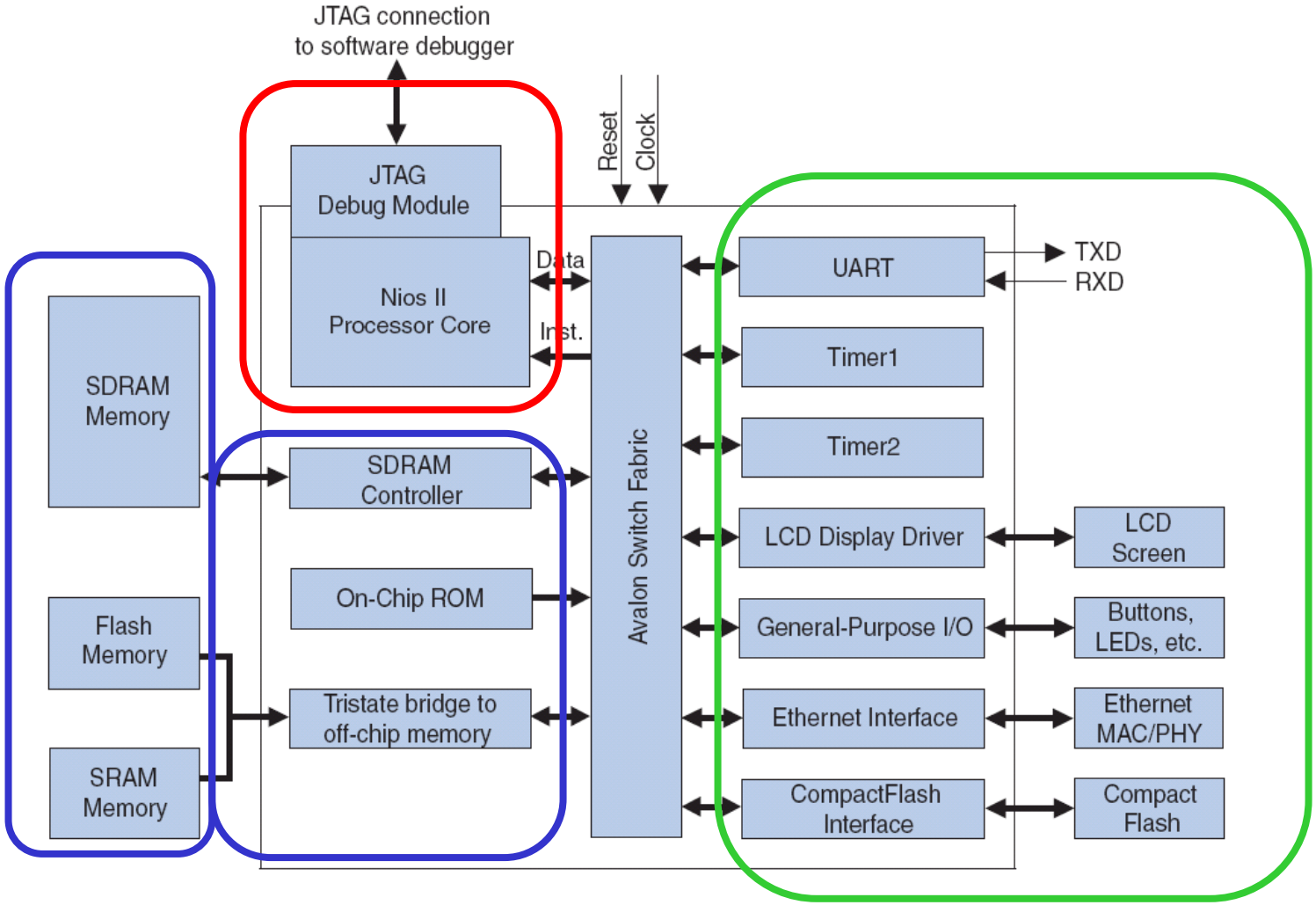
NIOSII – ISR *Interrupt Service Routine* performances

- Performance related to ISR (*Interrupt Service Routine)* processing. The following three key metrics determine ISR performance:
- **Interrupt latency** —the time from when an interrupt is first generated to when the processor runs the first instruction at the exception address.
- **Interrupt response time** —the time from when an interrupt is first generated to when the processor runs the first instruction in the ISR.
- **Interrupt recovery time** —the time taken from the last instruction in the ISR to return to normal processing.

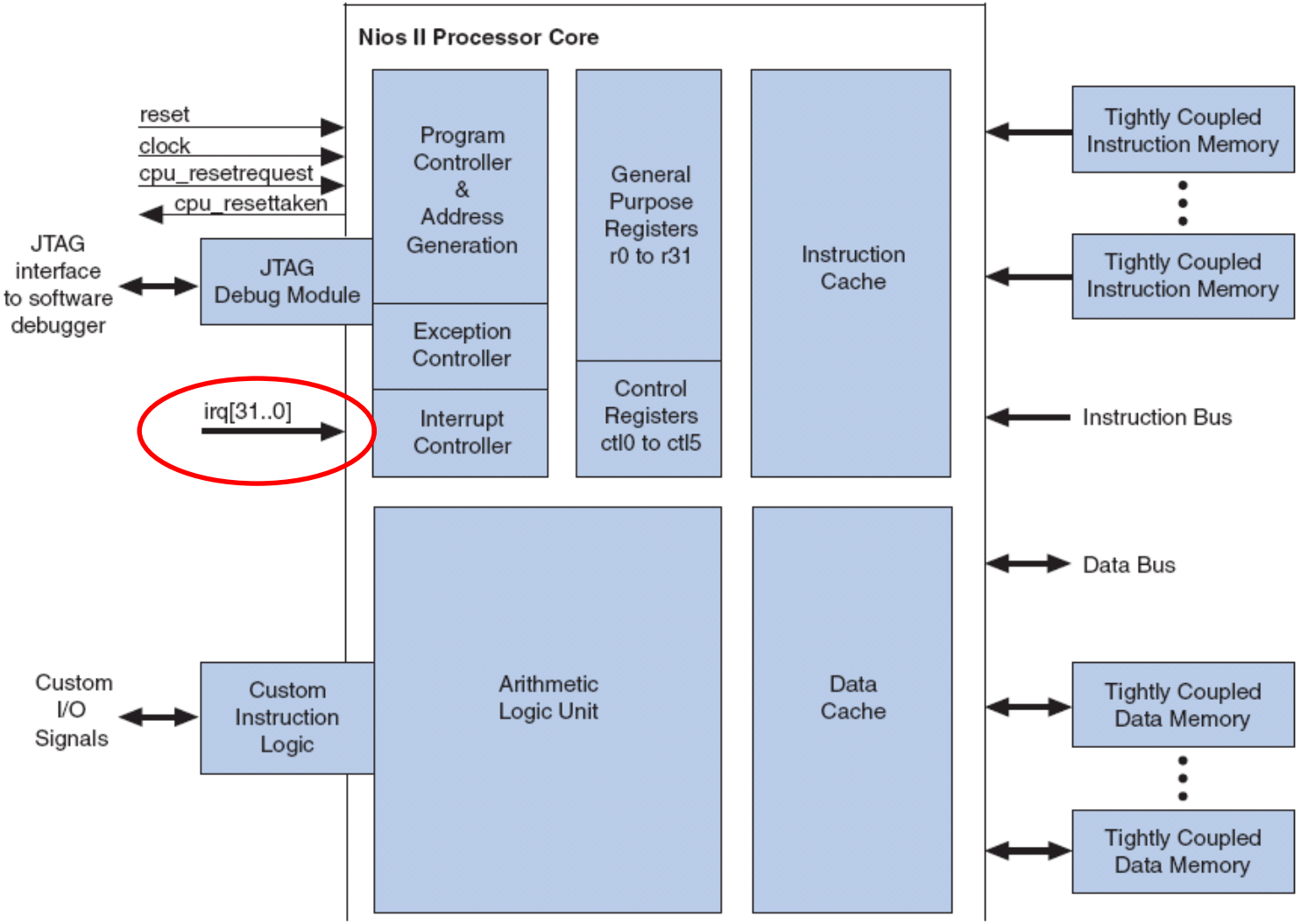
NIOSII - Performance for ISRs

- Because the Nios II processor is highly configurable, there is no single typical number for each metric.
- This section provides data points for each of the Nios II cores under the following assumptions:
- All code and data are stored in on-chip memory.
- The ISR code does not reside in the instruction cache.
- The software under test is based on the Altera-provided HAL exception handler system.
- The code is compiled using compiler optimization level "–O3", or higher optimization.

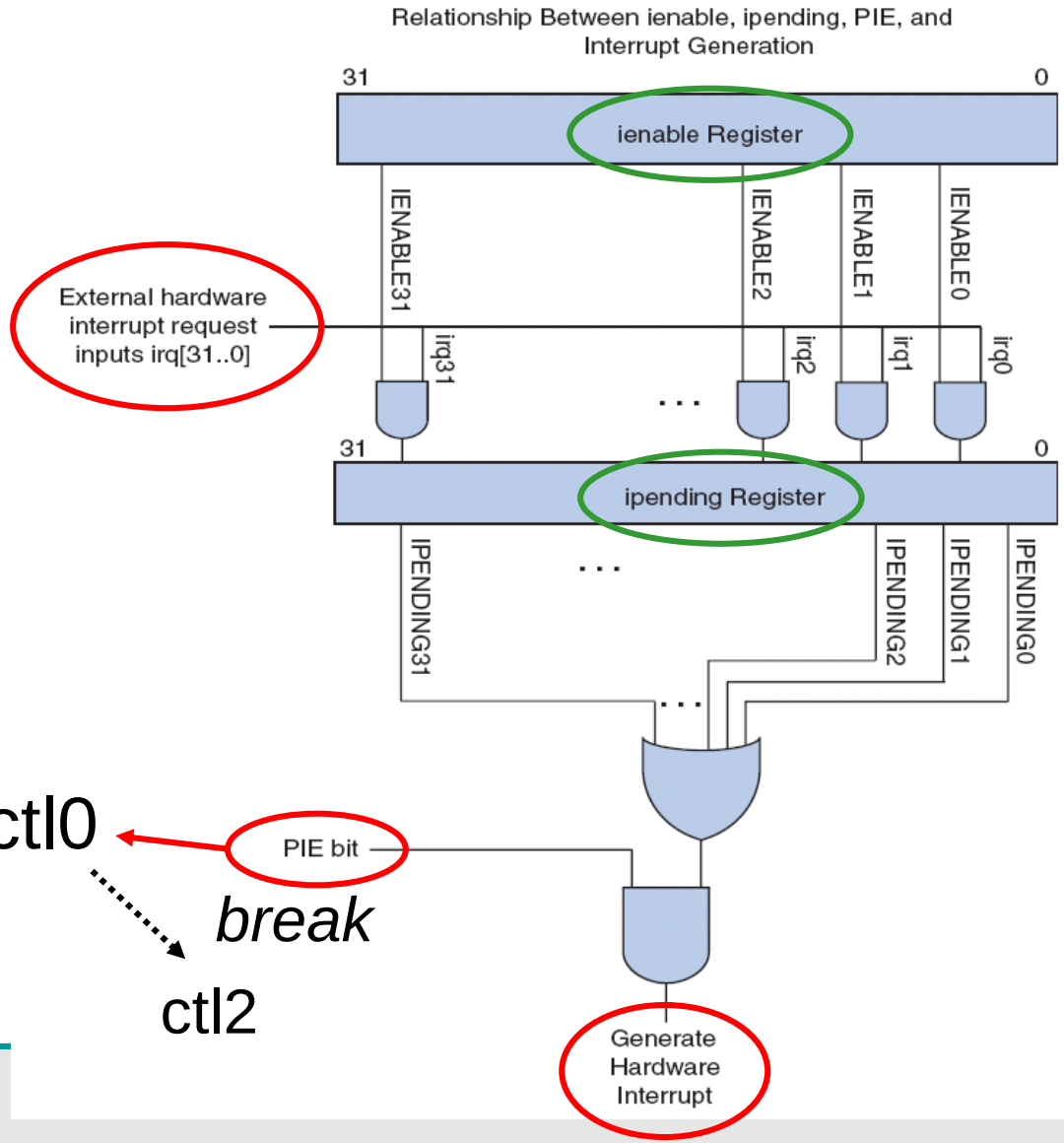
NIOS II –
Embedded system NIOSII/Avalon Architecture



NIOS II – Core Architecture



NIOS II – Interruptions (hardware) at processor level



ctl3

ctl4

ctl0

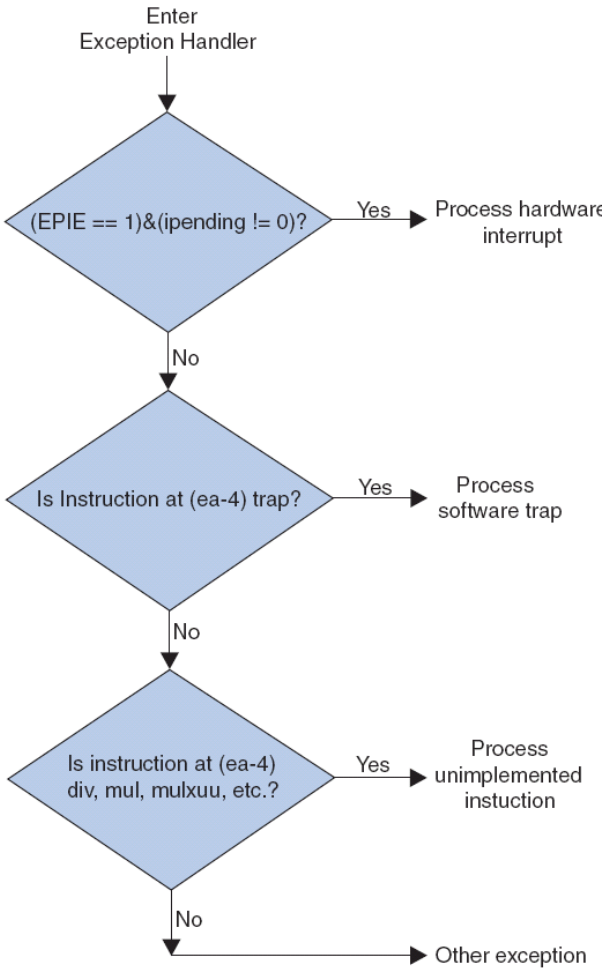
Inter.

break

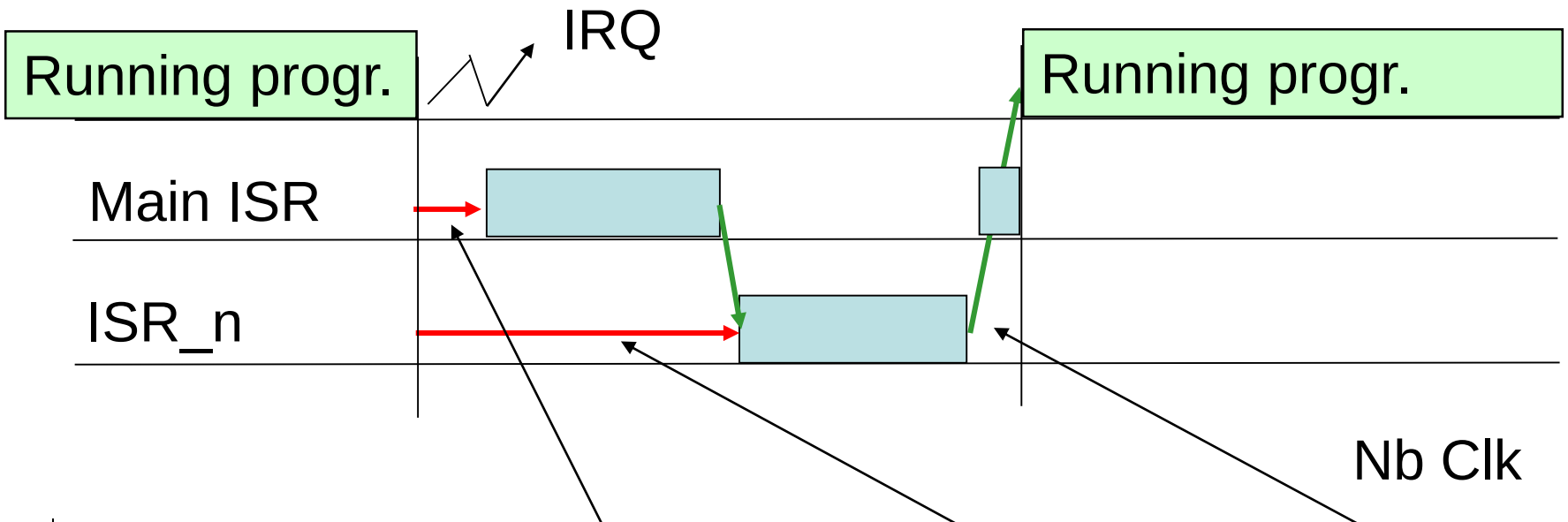
ctl1

ctl2

NIOS II – Exceptions handling



NIOSII - Performance for ISRs



Core	Latency	Response Time	Recovery Time
Nios II/f	10	105	62
Nios II/s	10	128	130
Nios II/e	15	485	222

NIOSII - HAL API for ISRs (Legacy call)

- alt_irq_register()
- alt_irq_disable()
- alt_irq_enable()
- alt_irq_disable_all()
- alt_irq_enable_all()
- alt_irq_interruptible()
- alt_irq_non_interruptible()
- alt_irq_enabled()

NIOSII - ISRs

- An ISR has to be provided for every interrupt source enabled
- The prototype for the ISR function is:

```
void isr (void* context, alt_u32 id);
```

- It will be saved in an array of ISR with the associated ***context***, at ***id*** index
- ***context*** is a pointer to something useful for the associated ISR (pointer on a struct)
- ***id*** is the irq number

NIOSII - HAL API for ISRs

- ISRs run in a restricted environment. A large number of the HAL API calls are **not** available from ISRs. For example, accesses to the HAL file system are not permitted.
- As a general rule, when writing your own ISR, never include function calls that can block waiting for an interrupt.
- In particular, **do not call printf()** from within an ISR unless you are certain that *stdout* is mapped to a non-interrupt-based device driver. Otherwise, *printf()* can deadlock the system, waiting for an interrupt that never occurs because interrupts are disabled.

NIOSII - ISRs

- ***isr*** is a pointer to the function that is called in response to IRQ number ***id***. The two input arguments provided to this function are the context ***pointer*** and ***id***. Registering a null pointer for isr results in the interrupt being disabled.
- The HAL registers the ISR by the storing the function pointer, ***isr***, in a lookup table. The return code from alt_irq_register() is zero if the function succeeded, and nonzero if it failed.
- If the HAL registers the ISR successfully, the associated Nios II interrupt (as defined by id) is locally **enabled** on return from alt_irq_register().
- Hardware-specific initialization might also be required.
- When a specific IRQ occurs, the HAL looks up the IRQ in the lookup table and dispatches the registered ISR.

NIOSII – ISRs (alt_irq_table.h)

From the IRQ_n, the main ISR dispatcher has to found the particular ISR to run and to provide a specific context pointer.

This information is initialize in an array of structure:

```
struct ALT_IRQ_HANDLER {
        void (*handler)(void*, alt_u32);
        void *context;
} alt_irq[ALT_NIRQ];
```

Handler is a pointer to the function to call by the main ISR handler

The corresponding ISR will receive ***context*** and ***id*** value

id	*handler	*context
0	^ISR_0	^context_0
1	^ISR_1	^context_1
2	^ISR_2	^context_2
…	^ISR_...	^context_...
ALT_NIRQ-1	^ISR_n	^context_n

NIOSII – Registering an ISR

Before the software can use an ISR, it must be registered it by calling

```
int alt_irq_register (
        alt_u32 id,
        void* context,
        void (*isr)(void*, alt_u32));
```

The prototype has the following parameters:

- ***id*** is the hardware interrupt number for the device, as defined in **system.h**. Interrupt priority corresponds inversely to the IRQ number. Therefore, **IRQ0** represents the **highest** priority interrupt and **IRQ31** is the **lowest**.
- ***context*** is a pointer used to pass context-specific information to the ISR, and can point to any ISR-specific information. The context value is opaque to the HAL; it is provided entirely for the benefit of the user-defined ISR.

NIOSII - Registering an ISR (source code)

```
int alt_irq_register (alt_u32 id,
                      void* context,
                      void (*handler)(void*, alt_u32)){
  int rc = -EINVAL;
  alt_irq_context status;

  if (id < ALT_NIRQ)  {
    /* interrupts are disabled while the handler tables are updated to ensure that an
      interrupt doesn't occur while the tables are in an inconsistent state.
     */
    status = alt_irq_disable_all ();

    alt_irq[id].handler = handler;
    alt_irq[id].context = context;

    rc = (handler) ? alt_irq_enable (id): alt_irq_disable (id);
    alt_irq_enable_all(status);
  }
  return rc;
}
```

NIOSII – Enable/disable interrupts

- The HAL provides functions to allow a program to disable interrupts for certain sections of code, and re-enable them later.
- ***alt_irq_disable()*** and ***alt_irq_enable()*** allow to disable and enable individual interrupts.
- ***alt_irq_disable_all()*** disables all interrupts, and returns a context value. To re-enable interrupts, call ***alt_irq_enable_all()*** and pass in the context parameter. In this way, interrupts are returned to their state prior to the call to alt_irq_disable_all().
- ***alt_irq_enabled()*** returns nonzero if interrupts are enabled, allowing a program to check on the status of interrupts.

- **Disable interrupts for as short a time as possible. Maximum interrupt latency increases with the amount of time interrupts are disabled.**

NIOSII – Enable/disable interrupts

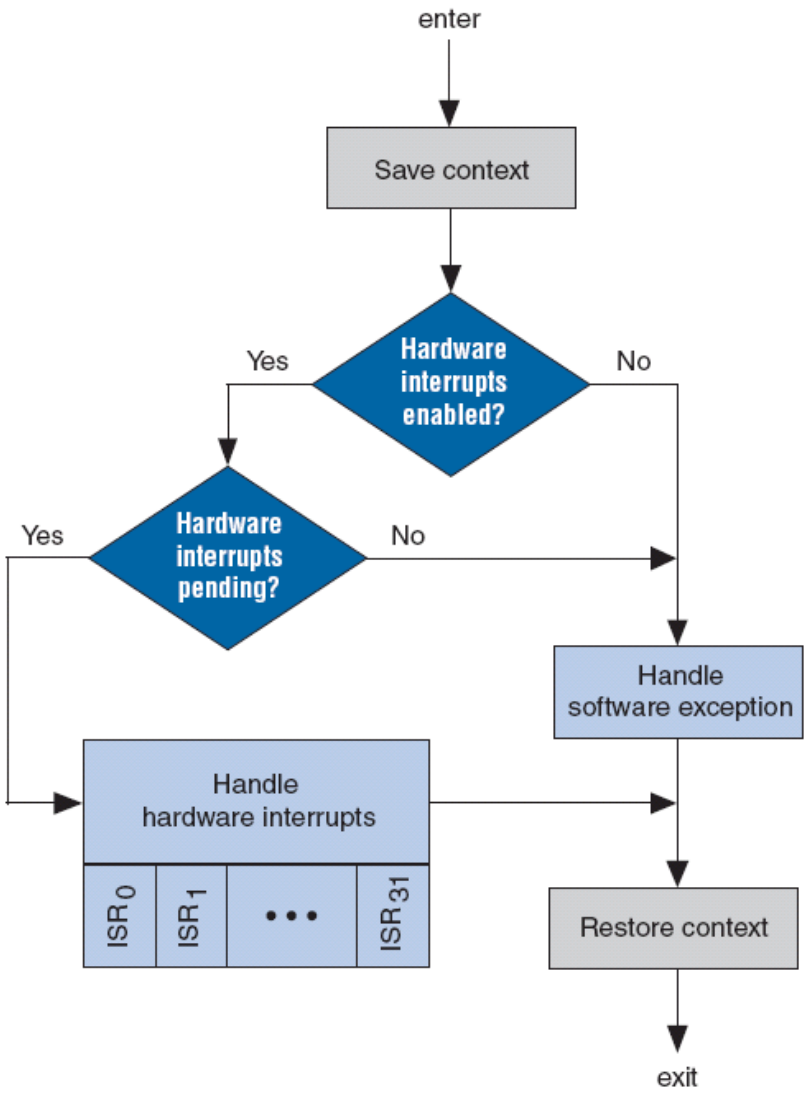
- int **alt_irq_disable** (alt_u32 id);
- int **alt_irq_enable** (alt_u32 id);
  - ***id***: individual IRQ.
  - The return value is zero.

- int **alt_irq_enabled** (void)
  - Returns zero if interrupts are disabled, and non-zero otherwise.

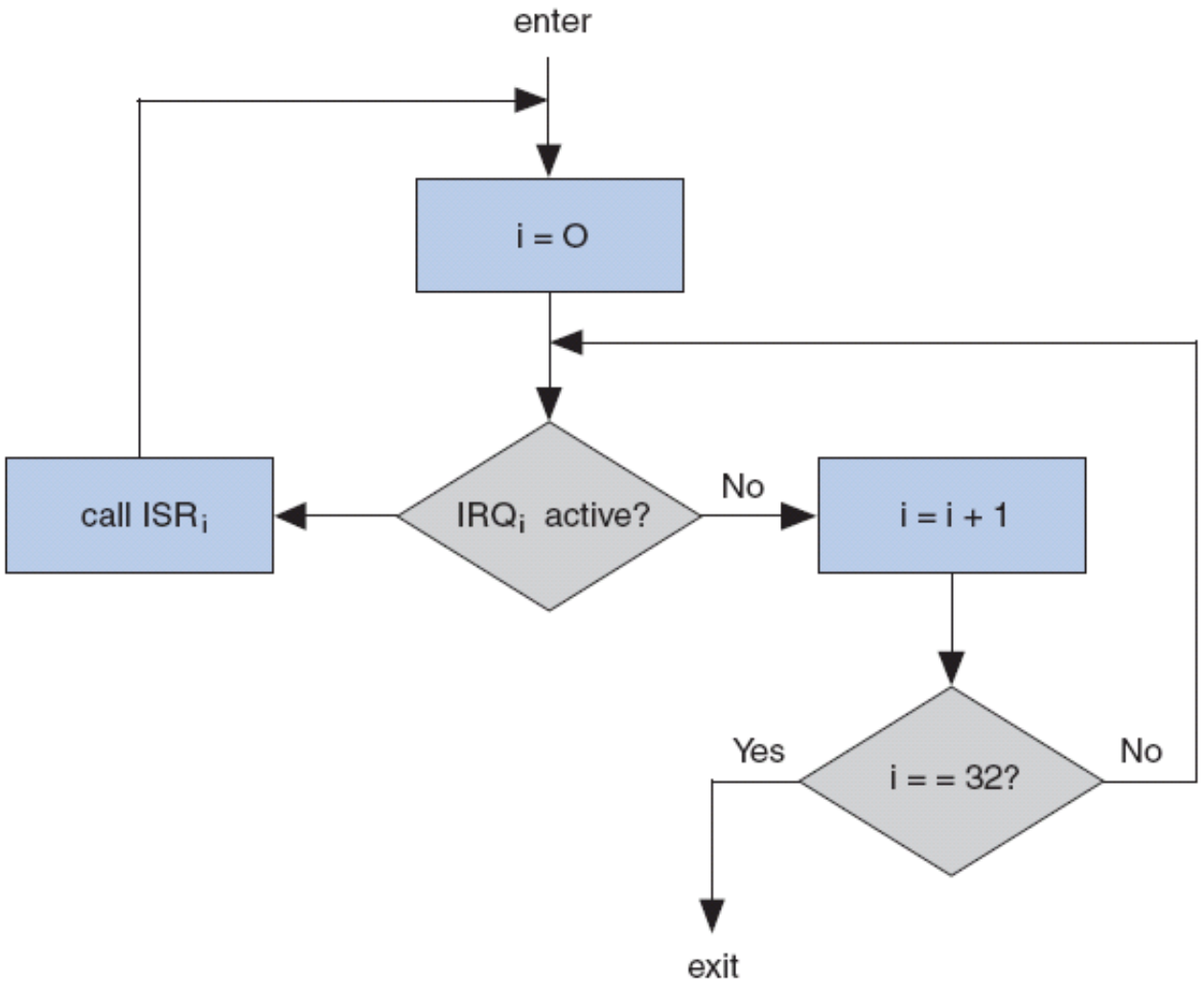
NIOSII – Enable/disable interrupts

- alt_irq_context **alt_irq_disable_all** (void);
- void **alt_irq_enable_all** (alt_irq_context context);
  - The alt_irq_enable_all() function enables all interrupts that were previously disabled by alt_irq_disable_all().
  - The input argument, *context*, is the value returned by a previous call to alt_irq_disable_all(). Using context allows nested calls to alt_irq_disable_all() and alt_irq_enable_all().
  - As a result, alt_irq_enable_all() does not necessarily enable all interrupts such as interrupts explicitly disabled by alt_irq_disable().

NIOSII – ISR work



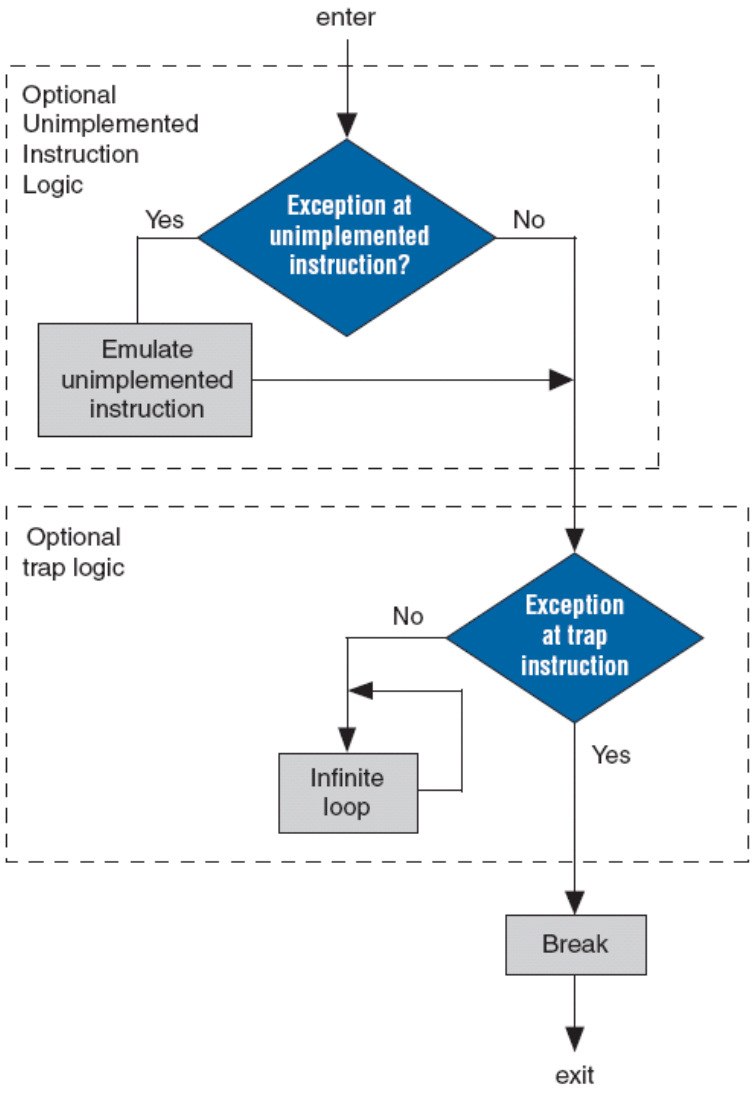
NIOSII – Hardware Interrupt Handler



NIOSII – Main Hardware Interrupt Handler

```
active = alt_irq_pending ();
do  {
    i = 0;
    mask = 1;    /* Test each bit in turn looking for an active interrupt. Once
     one is found, the interrupt handler asigned by a call to alt_irq_register() is
     called to clear the interrupt condition. */
    do {
      if (active & mask) {
        alt_irq[i].handler(alt_irq[i].context, i);
        break;
      }
      mask <<= 1;
      i++;
    } while (1);
    active = alt_irq_pending ();
  } while (active);
```

NIOSII – Software Exception Handler



- An exception routine must never execute an unimplemented instruction. The HAL exception handling system does not support nested software exceptions.

NIOSII – Software Exception Handler

Source files:

Version Quartus II 12.0:

C:\altera\10.1\ip\altera\nios2_ip\altera_nios2\HAL\src

C:\altera\12.0\ip\altera\nios2_ip\altera_nios2\HAL\src

NIOSII – Software Exception Handler

Source files (depend on the system version and Interrupt Controller used (IIC: Internal, EIC: External)):

- alt_exception_entry.S
- alt_exception_muldiv.S
- alt_exception_trap.S
- alt_irq_entry.S
- alt_irq_handler.c
- alt_software_exception.S
- alt_irq_vars.c
- alt_irq_register.c
- alt_iic.c
- alt_instruction_exception_entry.c

NIOSII – Software Exception Handler

Header files:

- alt_irq.h
- alt_irq_entry.h

Assembly files:

- alt_irq_entry.S
- alt_exception_trap.S
- alt_exception_entry.S
- alt_exception_muldiv.S

- Are written in NIOSII assembly language, as they have to manipulate **ctl** registers and save explicitly registers on/from the stack
- Provide **eret** instruction
- Call **alt_irq_handler.c** written in C

NIOSII – Performance

- At laboratory, design a system allowing to measure the latency from Timer IRQ to ISR entry

To access io interface, use the macro ***IORD(), IOWR()*** provided in
C:\altera\10.1\ip\nios2_ip\altera_nios2\HAL\inc**\io.h**

```
#define __IO_CALC_ADDRESS_NATIVE(BASE, REGNUM) \
  ((void *)(((alt_u8*)BASE) + ((REGNUM) * (SYSTEM_BUS_WIDTH/8))))
```

```
#define IORD(BASE, REGNUM) \
  __builtin_ldwio (__IO_CALC_ADDRESS_NATIVE ((BASE), (REGNUM)))
```

```
#define IOWR(BASE, REGNUM, DATA) \
  __builtin_stwio (__IO_CALC_ADDRESS_NATIVE ((BASE), (REGNUM)), DATA))
```

NIOS II - Instructions

- Load and store instructions

Instruction	Description
`ldw` `stw`	The `ldw` and `stw` instructions load and store 32-bit data words from/to memory. The effective address is the sum of a register's contents and a signed immediate value contained in the instruction. Memory transfers can be cached or buffered to improve program performance. This caching and buffering might cause memory cycles to occur out of order, and caching might suppress some cycles entirely. Data transfers for I/O peripherals should use `ldwio` and `stwio`.
`ldwio` `stwio`	`ldwio` and `stwio` instructions load and store 32-bit data words from/to peripherals without caching and buffering. Access cycles for `ldwio` and `stwio` instructions are guaranteed to occur in instruction order and are never suppressed.

NIOS II - Instructions

- Load and store instructions, byte, half-word
- Zero (unsigned) or sign extend data → 32 bits

Instruction	Description

`ldb`
`ldbu`
`stb`
`ldh`
`ldhu`
`sth`
	`ldb`, `ldbu`, `ldh` and `ldhu` load a byte or half-word from memory to a register. `ldb` and `ldh` sign-extend the value to 32 bits, and `ldbu` and `ldhu` zero-extend the value to 32 bits. `stb` and `sth` store byte and half-word values, respectively. Memory accesses can be cached or buffered to improve performance. To transfer data to I/O peripherals, use the "io" versions of the instructions, described below.

`ldbio`
`ldbuio`
`stbio`
`ldhio`
`ldhuio`
`sthio`
	These operations load/store byte and half-word data from/to peripherals without caching or buffering.

- ALU instructions

Instruction	Description

`and`
`or`
`xor`
`nor`
	These are the standard 32-bit logical operations. These operations take two register values and combine them bit-wise to form a result for a third register.

`andi`
`ori`
`xori`
	These operations are immediate versions of the `and`, `or`, and `xor` instructions. The 16-bit immediate value is zero-extended to 32 bits, and then combined with a register value to form the result.

`andhi`
`orhi`
`xorhi`
	In these versions of `and`, `or`, and `xor`, the 16-bit immediate value is shifted logically left by 16 bits to form a 32-bit operand. Zeroes are shifted in from the right.

`add`
`sub`
`mul`
`div`
`divu`
	These are the standard 32-bit arithmetic operations. These operations take two registers as input and store the result in a third register.

`addi`
`subi`
`muli`
	These instructions are immediate versions of the `add`, `sub`, and `mul` instructions. The instruction word includes a 16-bit signed value.

`mulxss` `mulxuu`	These instructions provide access to the upper 32 bits of a 32x32 multiplication operation. Choose the appropriate instruction depending on whether the operands should be treated as signed or unsigned values. It is not necessary to precede these instructions with a `mul`.
`mulxsu`	This instruction is used in computing a 128-bit result of a 64x64 signed multiplication.

NIOS II - Instructions

- MOVE instructions register → register
- Immediate value → register

Instruction	Description

`mov`
`movhi`
`movi`
`movui`
`movia`
	`mov` copies the value of one register to another register. `movi` moves a 16-bit signed immediate value to a register, and sign-extends the value to 32 bits. `movui` and `movhi` move an immediate 16-bit value into the lower or upper 16-bits of a register, inserting zeros in the remaining bit positions. Use `movia` to load a register with an address.

Comparison instructions

- Warning: there is NO flags register
- The result of comparison is 0 or 1 and is write to the destination register

Instruction	Description
cmpeq	==
cmpne	!=
cmpge	signed >=
cmpgeu	unsigned >=
cmpgt	signed >
cmpgtu	unsigned >
cmple	unsigned <=
cmpleu	unsigned <=
cmplt	signed <

NIOS II - Instructions

- All of these compare two registers or a register and an immediate value, and write either 1 (if true) or 0 to the result register. These instructions perform all the equality and relational operators of the C programming language.

Instruction	Description
`cmpltu`	unsigned <

`cmpeqi`
`cmpnei`
`cmpgei`
`cmpgeui`
`cmpgti`
`cmpgtui`
`cmplei`
`cmpleui`
`cmplti`
`cmpltui`
	These instructions are immediate versions of the comparison operations. They compare the value of a register and a 16-bit immediate value. Signed operations sign-extend the immediate value to 32-bits. Unsigned operations fill the upper bits with zero.

- Shift and rotate

Instruction	Description

`rol`
`ror`
`roli`
	The `rol` and `roli` instructions provide left bit-rotation. `roli` uses an immediate value to specify the number of bits to rotate. The `ror` instructions provides right bit-rotation. There is no immediate version of `ror`, because `roli` can be used to implement the equivalent operation.

`sll`
`slli`
`sra`
`srl`
`srai`
`srli`
	These shift instructions implement the << and >> operators of the C programming language. The `sll`, `slli`, `srl`, `srli` instructions provide left and right logical bit-shifting operations, inserting zeros. The `sra` and `srai` instructions provide arithmetic right bit-shifting, duplicating the sign bit in the most significant bit. `slli`, `srli` and `srai` use an immediate value to specify the number of bits to shift.

- Program Control instruction

Instruction	Description
`call`	This instruction calls a subroutine using an immediate value as the subroutine's absolute address, and stores the return address in register `ra`.
`callr`	This instruction calls a subroutine at the absolute address contained in a register, and stores the return address in register `ra`. This instruction serves the roll of dereferencing a C function pointer.
`ret`	The ret instruction is used to return from subroutines called by `call` or `callr`. `ret` loads and executes the instruction specified by the address in register `ra`.
`jmp`	The `jmp` instruction jumps to an absolute address contained in a register. `jmp` is used to implement switch statements of the C programming language.
`br`	Branch relative to the current instruction. A signed immediate value gives the offset of the next instruction to execute.

- Conditional Program Control instruction

Instruction	Description
bge bgeu bgt bgtu ble bleu blt bltu beq bne	These instructions provide relative branches that compare two register values and branch if the expression is true. See "Comparison Instructions" on page 3–17 for a description of the relational operations implemented.

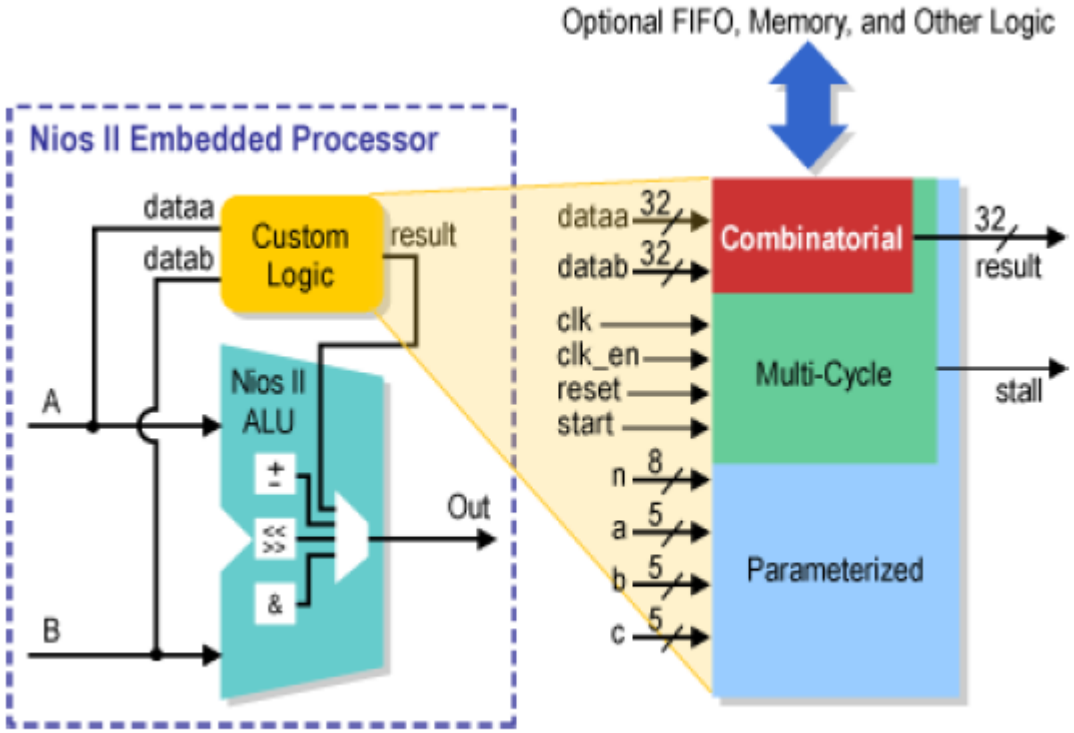
- Others Control instructions

Instruction	Description
`trap` `eret`	The `trap` and `eret` instructions generate and return from exceptions. These instructions are similar to the `call`/`ret` pair, but are used for exceptions. `trap` saves the `status` register in the `estatus` register, saves the return address in the `ea` register, and then transfers execution to the exception handler. `eret` returns from exception processing by restoring `status` from `estatus`, and executing the instruction specified by the address in `ea`.
`break` `bret`	The `break` and `bret` instructions generate and return from breaks. `break` and `bret` are used exclusively by software debugging tools. Programmers never use these instructions in application code.
`rdctl` `wrctl`	These instructions read and write control registers, such as the `status` register. The value is read from or stored to a general-purpose register.

`flushd`
`flushi`
`initd`
`initi`
	These instructions are used to manage the data and instruction cache memories.

`flushp`	This instruction flushes all pre-fetched instructions from the pipeline. This is necessary before jumping to recently-modified instruction memory.
`sync`	This instruction ensures that all previously-issued operations have completed before allowing execution of subsequent load and store operations.

NIOS II - Custom Instructions



NIOS II – Memory – I/O access

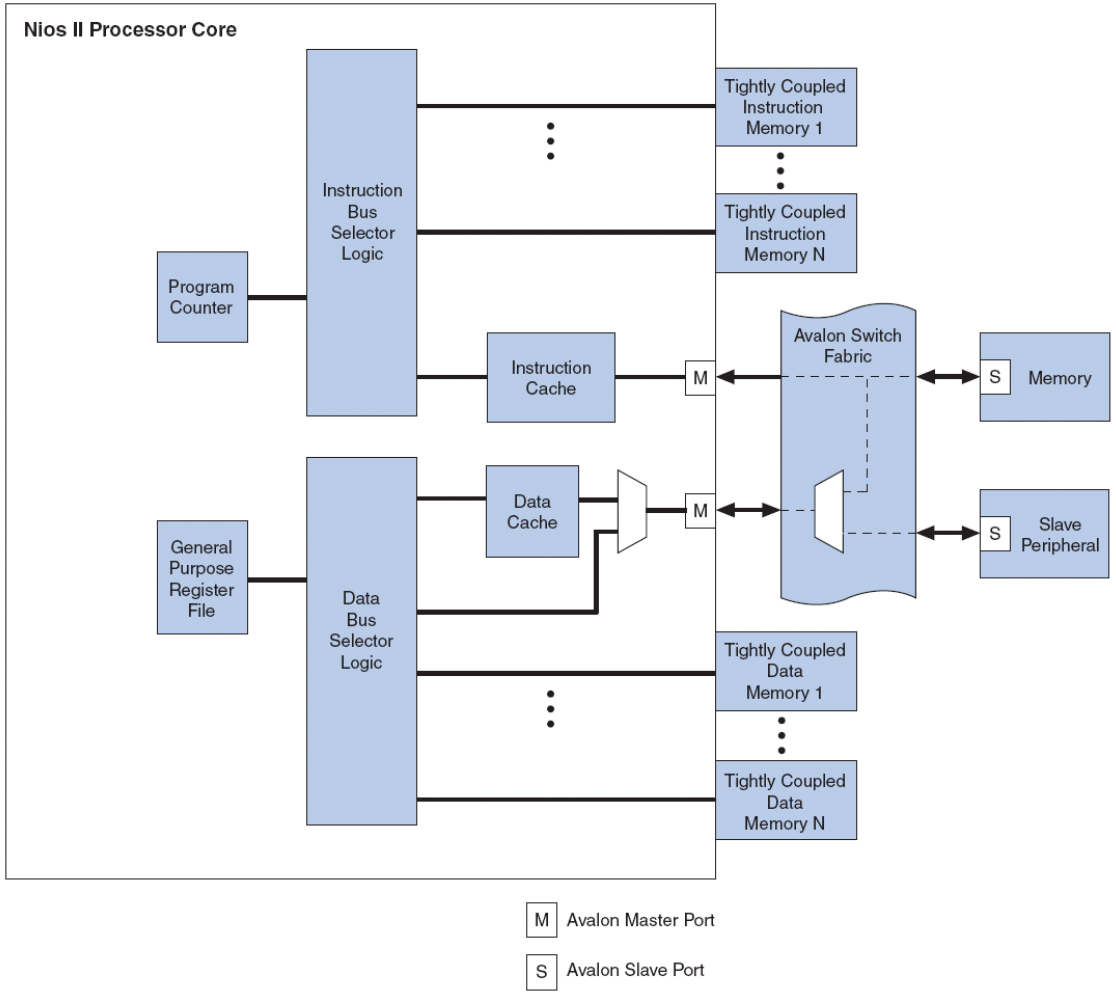
A Nios II core uses one or more of the following to provide memory and I/O access:

- **Instruction master port** - An Avalon master port that connects to instruction memory via Avalon switch fabric
- **Instruction cache** - Fast cache memory internal to the Nios II core
- **Data master port** - An Avalon master port that connects to data memory and peripherals via Avalon switch fabric
- **Data cache** - Fast cache memory internal to the Nios II core
- **Tightly coupled instruction or data memory port** - Interface to fast memory outside the Nios II core

NIOS II – Memory – I/O access

NIOSII - Data Path

- The instruction master port always retrieves 32 bits of data. The instruction master port relies on dynamic bus-sizing logic contained in the Avalon switch fabric.
- By virtue of dynamic bus sizing, every instruction fetch returns a full instruction word, regardless of the width of the target memory.
- Consequently, programs do not need to be aware of the widths of memory in the Nios II processor system.



Cache Bypass Method

- The Nios II architecture provides load and store I/O instructions such as ***ldio*** and ***stio*** that bypass the data cache and force an Avalon data transfer to a specified address.
- Additional cache bypass methods might be provided, depending on the processor core implementation.
- Some Nios II processor cores support a mechanism called ***bit-31 cache bypass*** to bypass the cache depending on the value of the most-significant bit of the address.

NIOS II – Tightly Coupled Memory

- **Tightly coupled memory** provides guaranteed low-latency memory access for performance-critical applications. Compared to cache memory, tightly coupled memory provides the following benefits:
  - Performance similar to cache memory
  - Software can guarantee that performance-critical code or data is located in tightly coupled memory
  - No real-time caching overhead, such as loading, invalidating, or flushing memory
- Physically, a tightly coupled memory port is a separate master port on the Nios II processor core, similar to the instruction or data master port. A Nios II core can have zero, one, or multiple tightly coupled memories.
- The Nios II architecture supports tightly coupled memories for both instruction and data access. Each tightly coupled memory port connects directly to exactly one memory with guaranteed low, fixed latency. The memory is external to the Nios II core and is usually located on chip.

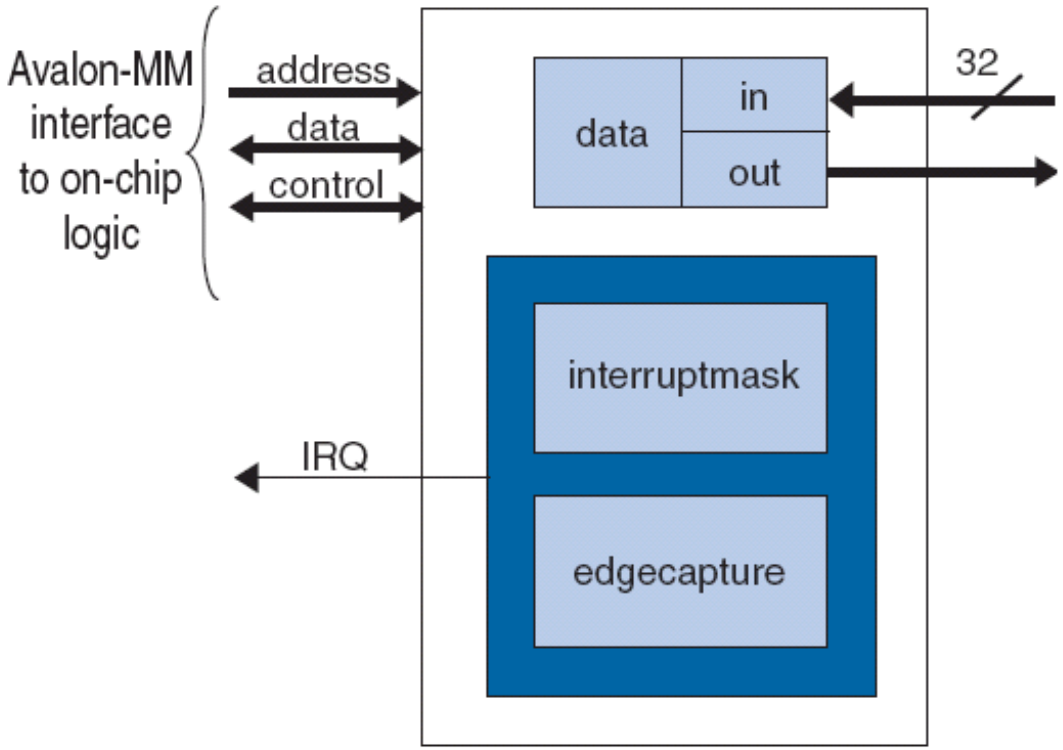
NIOS II – JTAG interface

- The Nios II architecture supports a JTAG debug module that provides onchip emulation features to control the processor remotely from a host PC.
- PC-based software debugging tools communicate with the JTAG debug module and provide facilities, such as:
  - **Downloading programs to memory**
  - **Starting and stopping execution**
  - **Setting breakpoints and watchpoints**
  - **Analyzing registers and memory**
  - **Collecting real-time execution trace data**
- The debug module connects to the JTAG circuitry in an Altera® FPGA.
- External debugging probes can then access the processor via the standard JTAG interface on the FPGA. On the processor side, the debug module connects to signals inside the processor core.

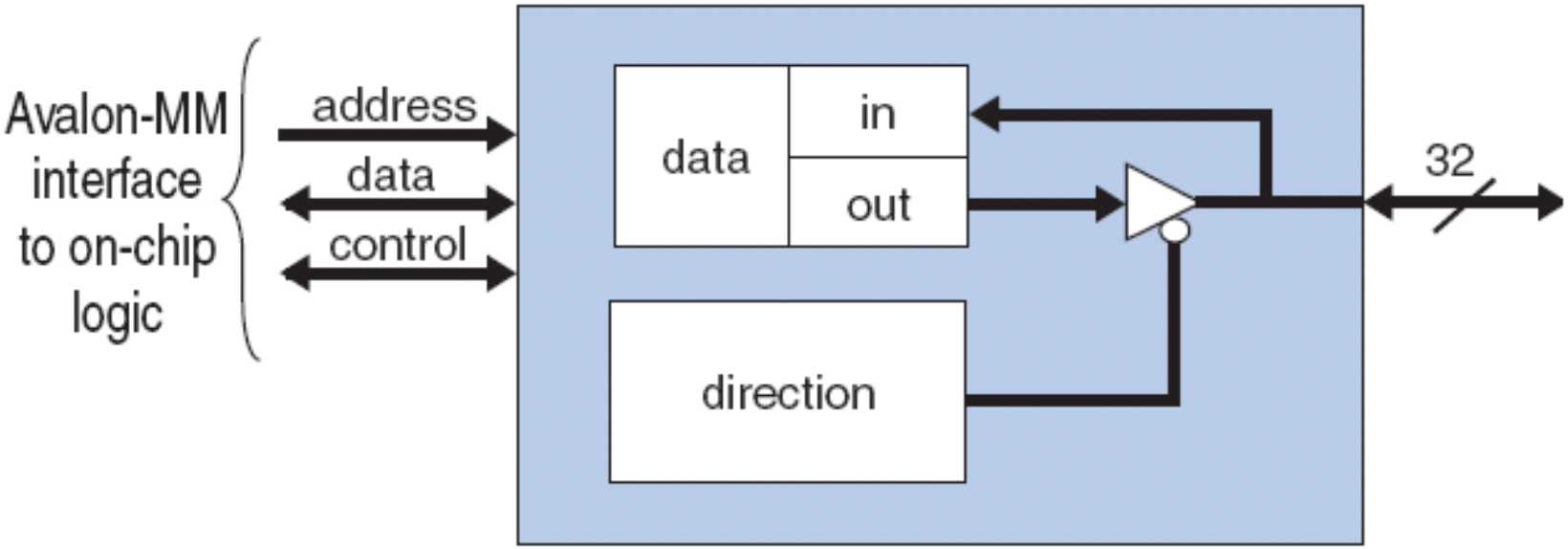
- Modes of configuration :
  - Bidirectional
  - Input
  - Output
  - Input and Output
- Interrupt Request capability

Setting	Description
Bidirectional (tristate) ports	In this mode, each PIO bit shares one device pin for driving and capturing data. The direction of each pin is individually selectable. To tristate an FPGA I/O pin, set the direction to input.
Input ports only	In this mode the PIO ports can capture input only.
Output ports only	In this mode the PIO ports can drive output only.
Both input and output ports	In this mode, the input and output ports buses are separate, unidirectional buses of *n* bits wide.

NIOS II - Some Avalon Peripherals
PIO



- In/Out Mode
- Interrupt request



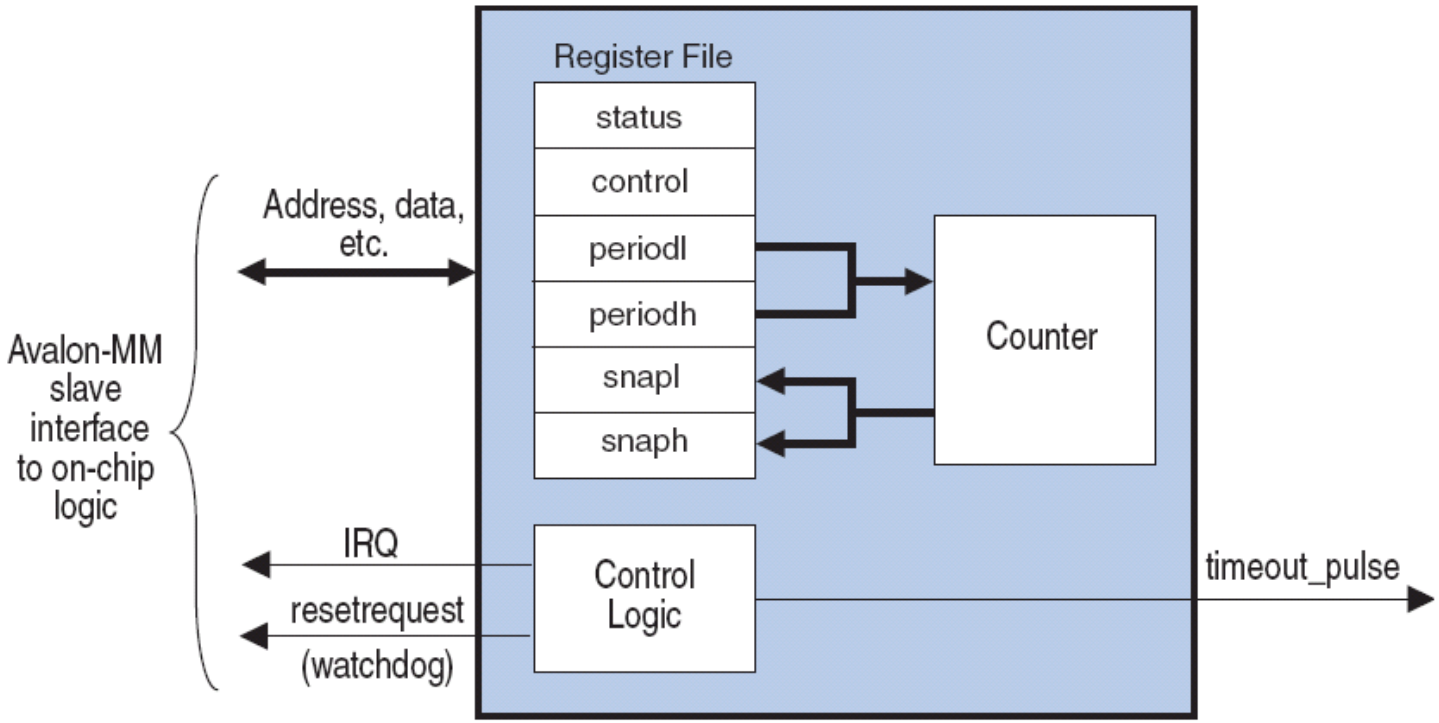
- Bidirectional mode

- 4 registers to control the PIO
- Some features available as SOPC instantiation :
  Edge/level to interrupt request

Offset	Register Name		R/W	(n-1)	...	2	1	0
0	`data`	read access	R	Data value currently on PIO inputs				
		write access	W	New value to drive on PIO outputs				
1	`direction` *(1)*		R/W	Individual direction control for each I/O port. A value of 0 sets the direction to input; 1 sets the direction to output.				
2	`interruptmask` *(1)*		R/W	IRQ enable/disable for each input port. Setting a bit to 1 enables interrupts for the corresponding port.				
3	`edgecapture` *(1)*, *(2)*		R/W	Edge detection for each input port.				

(1) This register may not exist, depending on the hardware configuration. If a register is not present, reading the register returns an undefined value, and writing the register has no effect.
(2) Writing any value to `edgecapture` clears all bits to 0.

NIOS II - Some Avalon Peripherals
Timer



- Timeout_pulse
- IRQ
- Watchdog → Reset request

- 6 registers for status - control – configuration
- Generate a TimeOut when the decrementing Timer counter reach 0, programmed by ***Periodh-PeriodL***

Offset	Name	R/W	Description of Bits						
			15	...	4	3	2	1	0
0	status	RW	*(1)*					RUN	TO
1	control	RW	*(1)*			STOP	START	CONT	ITO
2	periodl	RW	Timeout Period – 1 (bits 15..0)						
3	periodh	RW	Timeout Period – 1 (bits 31..16)						
4	snapl	RW	Counter Snapshot (bits 15..0)						
5	snaph	RW	Counter Snapshot (31..16)						

- 6 registers for status - control – configuration
- Write to one of *Snapshot register* generate the transfer of the current counter's value to ***SnapshotH-SnapshotL*** registers
- Mode can by single or continue (CONT = 1)
- An interrupt can by generated at TimeOut if ITO=1

- Status register

Bit	Name	Read/ Write/ Clear	Description
0	TO	RC	The TO (timeout) bit is set to 1 when the internal counter reaches zero. Once set by a timeout event, the TO bit stays set until explicitly cleared by a master peripheral. Write zero to the `status` register to clear the TO bit.
1	RUN	R	The RUN bit reads as 1 when the internal counter is running; otherwise this bit reads as 0. The RUN bit is not changed by a write operation to the `status` register.

- Control register

Bit	Name	Read/ Write/ Clear	Description
0	ITO	RW	If the ITO bit is 1, the timer core generates an IRQ when the `status` register's TO bit is 1. When the ITO bit is 0, the timer does not generate IRQs.
1	CONT	RW	The CONT (continuous) bit determines how the internal counter behaves when it reaches zero. If the CONT bit is 1, the counter runs continuously until it is stopped by the STOP bit. If CONT is 0, the counter stops after it reaches zero. When the counter reaches zero, it reloads with the 32-bit value stored in the `periodl` and `periodh` registers, regardless of the CONT bit.
2	START (1)	W	Writing a 1 to the START bit starts the internal counter running (counting down). The START bit is an event bit that enables the counter when a write operation is performed. If the timer is stopped, writing a 1 to the START bit causes the timer to restart counting from the number currently held in its counter. If the timer is already running, writing a 1 to START has no effect. Writing 0 to the START bit has no effect.
3	STOP *(1)*	W	Writing a 1 to the STOP bit stops the internal counter. The STOP bit is an event bit that causes the counter to stop when a write operation is performed. If the timer is already stopped, writing a 1 to STOP has no effect. Writing a 0 to the stop bit has no effect. Writing 0 to the STOP bit has no effect. If the timer hardware is configured with the **Start/Stop control bits** option turned off, writing the STOP bit has no effect.

Interrupt Behavior

- The timer core generates an IRQ whenever the internal counter reaches zero and the ITO bit of the control register is set to 1.

Acknowledge the IRQ in one of two ways:

- Clear the TO bit of the status register
- Disable interrupts by clearing the ITO bit of the control register
- Failure to acknowledge the IRQ produces an undefined result.

- Profiling is often necessary to validate the timing of task, process, interrupt latency/response, software performance, etc…
- Some methods exists to do this task :
  - Software only as ***GNU profiler***, gprof
  - Mostly Hardware module as ***performance counter core***
  - Hardware/software ***Interval timer peripheral***

- The main benefit of using the **performance counter** core is the accuracy of the profiling results. The performance counter core is unobtrusive, requiring only a single instruction to start and stop profiling, and no RAM. It is appropriate for high-precision measurements of narrowly targeted sections of code.
- **GNU profiler, gprof** - gprof provides broad low-precision timing information about the entire software system. It uses a substantial amount of RAM, and degrades the real-time performance. For many embedded applications, gprof distorts real-time behavior too much to be useful. Change cache memory capability.
- **Interval timer peripheral** -The interval timer is less intrusive than gprof. It can provide good results for narrowly targeted sections of code. However, the granularity of the results is milliseconds, which is too coarse for many embedded applications.

- The core contains two counters for every section:
  - Time: A 64-bit clock cycle counter.
  - Events: A 32-bit event counter.
- **Section Counters**
  - Each 64-bit time counter records the aggregate number of clock cycles spent in a section of code.
  - The 32-bit event counter records the number of times the section executes.
  - The performance counter core can have up to seven section counters.
- **Global Counter**
  - The global counter controls all section counters. The section counters are enabled only when the global counter is running.

NIOS II - Some Avalon Peripherals
Performance Counter

Offset	Register Name	Bit Description		
		Read	Write	
		31 … 0	31 …	0
0	$T[0]_{lo}$	global clock cycle counter [31: 0]	*(1)*	0 = STOP 1 = RESET
1	$T[0]_{hi}$	global clock cycle counter [63:32]	*(1)*	0 = START
2	Ev[0]	global event counter	*(1)*	*(1)*
3	–	*(1)*	*(1)*	*(1)*
4	$T[1]_{lo}$	section 1 clock cycle counter [31: 0]	*(1)*	0 = STOP
5	$T[1]_{hi}$	section 1 clock cycle counter [63:32]	*(1)*	0 = START
6	Ev[1]	section 1 event counter	*(1)*	*(1)*
7	–	*(1)*	*(1)*	*(1)*
8	$T[2]_{lo}$	section 2 clock cycle counter [31: 0]	*(1)*	0 = STOP
9	$T[2]_{hi}$	section 2 clock cycle counter [63:32]	*(1)*	0 = START
10	Ev[2]	section 2 event counter	*(1)*	*(1)*
11	–	*(1)*	*(1)*	*(1)*
. . .	. . .	. . .	. . .	. . .
$4n + 0$	$T[n]_{lo}$	section *n* clock cycle counter [31: 0]	*(1)*	0 = STOP
$4n + 1$	$T[n]_{hi}$	section *n* clock cycle counter [63:32]	*(1)*	0 = START
$4n + 2$	Ev[n]	section *n* event counter	*(1)*	*(1)*
$4n + 3$	–	*(1)*	*(1)*	*(1)*

(1) Reserved. Read values are undefined. When writing, set reserved bits to zero.

Performance counter functions and Macro

Name	Summary
`PERF_RESET()`	Stops and disables all counters, resetting them to 0.
`PERF_START_MEASURING()`	Starts the global counter and enables section counters.
`PERF_STOP_MEASURING()`	Stops the global counter and disables section counters.
`PERF_BEGIN()`	Starts timing a code section.
`PERF_END()`	Stops timing a code section.
`perf_print_formatted_report()`	Sends a formatted summary of the profiling results to stdout.
`perf_get_total_time()`	Returns the aggregate global profiling time in clock cycles.
`perf_get_section_time()`	Returns the aggregate time for one section in clock cycles.
`perf_get_num_starts()`	Returns the number of counter events.
`alt_get_cpu_freq()`	Returns the CPU frequency in Hz.

- Need the full library (not small C library in NIOS IDE) as floating printing is used.

```
perf_print_formatted_report(
    (void *)PERFORMANCE_COUNTER_BASE,  // Peripheral's HW base address
    alt_get_cpu_freq(),                // defined in "system.h"
    3,                                 // How many sections to print
    "1st checksum_test",               // Display-names of sections
    "pc_overhead",
    "ts_overhead");
```

The preceding example creates a table similar to this:

```
--Performance Counter Report--
Total Time: 2.07711 seconds (103855534 clock-cycles)
+-----------------+--------+-----------+---------------+-----------+
| Section         |      % | Time (sec)| Time (clocks) |Occurrences|
+-----------------+--------+-----------+---------------+-----------+
|1st checksum_test|     50 |   1.03800 |      51899750 |         1 |
+-----------------+--------+-----------+---------------+-----------+
| pc_overhead     |1.73e-05|   0.00000 |            18 |         1 |
+-----------------+--------+-----------+---------------+-----------+
| ts_overhead     |4.24e-05|   0.00000 |            44 |         1 |
+-----------------+--------+-----------+---------------+-----------+
```

- If an interrupt occurs during the measured function execution, this time to execute interrupt routine is counted.

Références

- **Cyclone2, NIOSII**, Altera, www.altera.com
- http://www.altera.com/literature/lit-nio2.jsp

- http://www.altera.com/literature/tt/tt_nios2_system_architect.pdf
- http://www.altera.com/literature/tt/tt_qsys_intro.pdf
- http://www.altera.com/literature/hb/nios2/n2sw_nii52006.pdf
- http://www.altera.com/literature/ug/ug_embedded_ip.pdf
- http://www.altera.com/literature/an/AN595.pdf

- n2sw_nii5v2.pdf
- n2cpu_nii5v1.pdf
- n2cpu_nii5v3.pdf

- From this description and code, it is very easy to add others features as:
- Output Compare function
- Interrupt at specific time
- Reload counter
- Etc…