UNIVERSITY OF LEICESTER

Department of Engineering

EG3205 Programming Microelectronic and Multi-Core Systems

Part II. Multiprocessor and Multicore Systems

Prof. Tanya Vladimirova
MEng MSc PhD CEng FIET SMIEEE MACM FHEA
Email: tv29@le.ac.uk

Outline

- Org Matters
- Lab 4: Shared-Clock TTC Scheduler: implementation
- NIOS II Soft Processor Core: features
- Accessing NIOS II Peripherals
- Mutual exclusion core
- Appendix

Org Matters

- This is the last lecture, no lecture next week
- Assignment
  - Submission Date: Tuesday the 18th December
  - Demonstration: during the Thursday 13th December lab the latest
  - Submit report and two Quartus II projects as one zip file to BB.
- Return the two DE0 and CAN-SPI boards in their original packaging to Mr Dominic Kent in the Department Stores by the 20th December.

Shared Clock Scheduling over CAN:

Lab 4 Software Implementation

Lab 4 Implementation Scenario

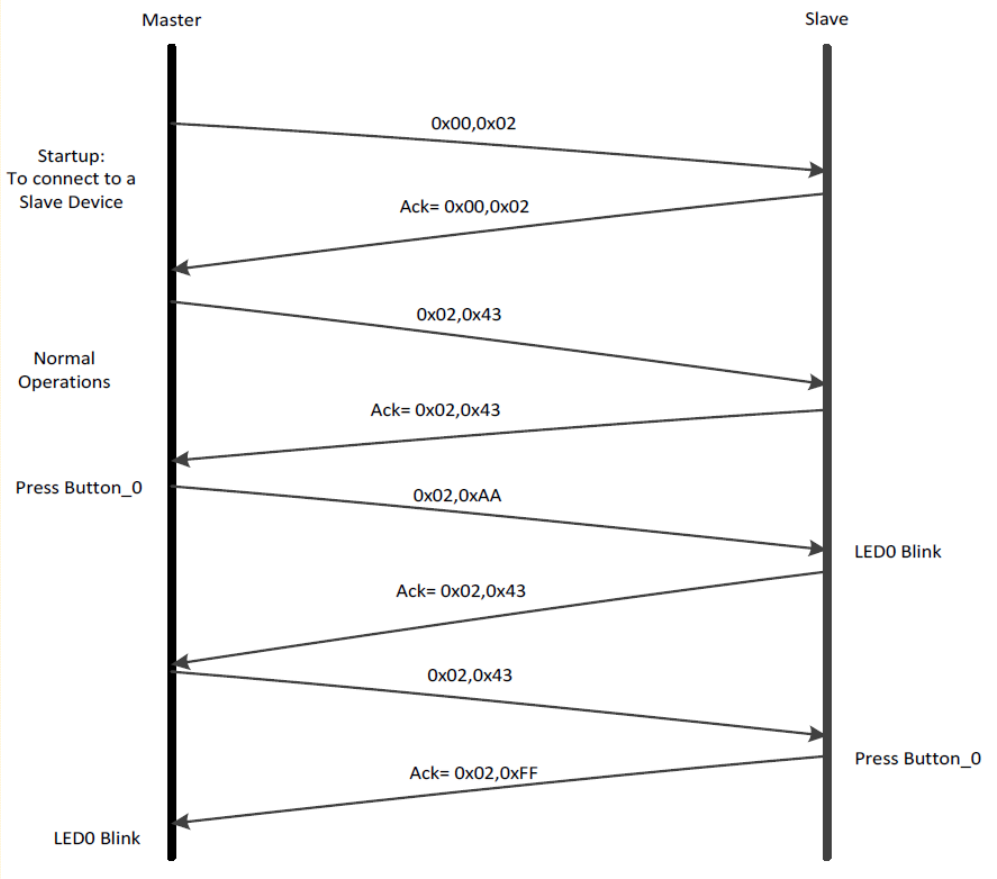
You were expected to demonstrate a successful system operation on the hardware setup showing the following outcomes:

- A heartbeat LED blinking continuously on each board as follows:
  - LED2 - on Master Board,
  - LED3 - on Slave Board.
- When Button 0 on the master board is pressed, LED 0 on the slave board blinks.

Communication between Master and Slave nodes

- Startup
- Normal Operations
- Master Button0 Press
- Slave Button0 Press

Communication and data sequence



Communication between Master and Slave nodes

- For communication among the nodes, you can use and modify the provided CAN-SPI driver as necessary.
  - Relevant details are available in the MCP2515 user guide, the CAN-SPI manual and the embedded IP user guide, which are provided as reference material on Blackboard.
- The following Table shows an example of the software components that will be required to upgrade the TTC scheduler to a shared-clock scheduler.
- You can choose to devise your own tasks and functions too.

Master Node		
	Function Name	**Description**
Shared Clock Scheduler	Void SCH_Init_T0(void)	This function is used to initialize timer 0 and CAN-SPI.
	Void SCH_Start(void)	This function will try to establish connection with the slave on startup.
	Void SCH_Update(void * context)	This function will be executed in the master on timer interrupt. It will send a tick message, process ACK message and check whether a task is available to run or not.
	Void SCC_A_Master_Send_Tick_Message(const tByte SLAVE_INDEX)	This function will send a tick message to the slave over the CAN bus via the CAN-SPI module.
	Void SCC_A_Master_Start_Slave(const tByte SLAVE_ID)	This function will send a start message to the slave over CAN. It will also receive and process an ACK message from slave.
	Void SCC_A_Master_Process_Ack(const tByte SLAVE_INDEX)	This function will receive and process an ACK message from slave

		the slave on startup.
Shared Clock Scheduler	Void SCH_Update(void * context)	This function will be executed in the master on timer interrupt. It will send a tick message, process ACK message and check whether a task is available to run or not.
	Void SCC_A_Master_Send_Tick_Message(const tByte SLAVE_INDEX)	This function will send a tick message to the slave over the CAN bus via the CAN-SPI module.
	Void SCC_A_Master_Start_Slave(const tByte SLAVE_ID)	This function will send a start message to the slave over CAN. It will also receive and process an ACK message from slave.
	Void SCC_A_Master_Process_Ack(const tByte SLAVE_INDEX)	This function will receive and process an ACK message from slave during normal operations.
Tasks	Void HEARTBEAT_Update(void)	This task updates the HeartBeat LED which produces a blinking effect.
	Void LED_ONOFF(void)	This task will turn ON/OFF an LED.
	Void PushButton_Update(void)	This task will receive an input from a push button and change the tick message.
Slave Node		

Slave Node		
	Function Name	**Description**
Shared Clock Scheduler	Void SCH_Init_T0(void)	This function is used to initialize CAN-SPI and its interrupt.
	Void SCH_Start(void)	Wait for the master tick. Once it has arrived, process it and compare it with your own ID. Once ID is matched, enable the slave.
	Void SCH_Update(void * context)	This function will be executed in the slave, on receiving a CAN interrupt. It will process the tick message, send an ACK message and check whether a task is available to run or not.
	tByte SCC_A_Slave_Process_Tick_Message(void)	This function will process the tick message received from the master over the CAN bus via the CAN-SPI module.
	Void SCC_A_Send_Ack_Message(void)	This function will send an ACK message to the master.
	Void HEARTBEAT_Update(void)	This task updates the HeartBeat LED, which produces a blinking effect.

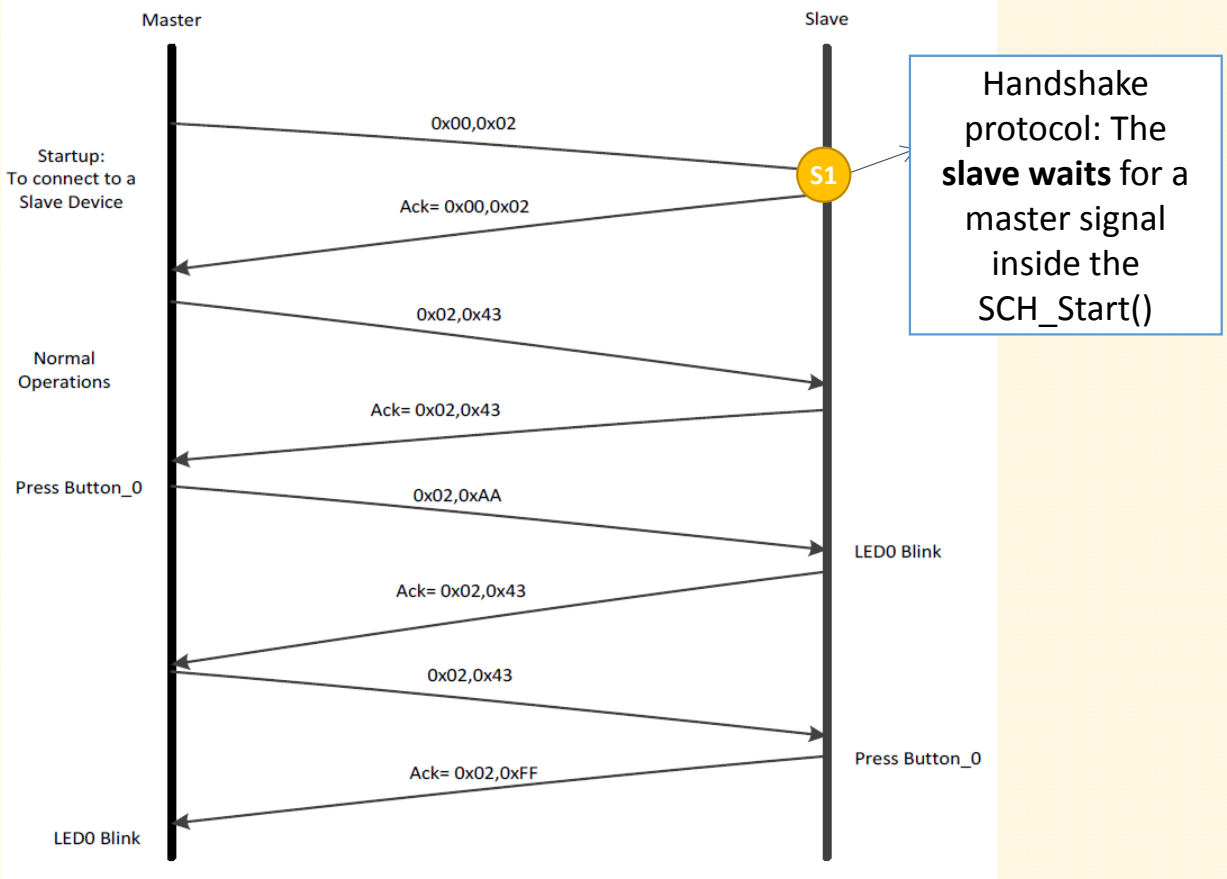
Shared Clock Scheduler		Once it has arrived, process it and compare it with your own ID. Once ID is matched, enable the slave.
	Void SCH_Update(void * context)	This function will be executed in the slave, on receiving a CAN interrupt. It will process the tick message, send an ACK message and check whether a task is available to run or not.
	tByte SCC_A_Slave_Process_Tick_ Message(void)	This function will process the tick message received from the master over the CAN bus via the CAN-SPI module.
	Void SCC_A_Send_Ack_Message(v oid)	This function will send an ACK message to the master.
Tasks	Void HEARTBEAT_Update(void)	This task updates the HeartBeat LED, which produces a blinking effect.
	Void LED_ONOFF(void)	This task will turn ON/OFF an LED.
	Void PushButton_Update(void)	This task will receive an input from a push button and change the tick message.

Communication between Master and Slave nodes Startup Phase

At startup:

- The master will send a message (0x00, 0x02) to the slave to check their presence.
- On receipt of this message, the slave node will send a reply message (0x00, 0x02) to the master node.
  - The second byte of this message is the slave ID.
- After a successful exchange of startup messages, the slave is now connected with the master, and can start onward communications.

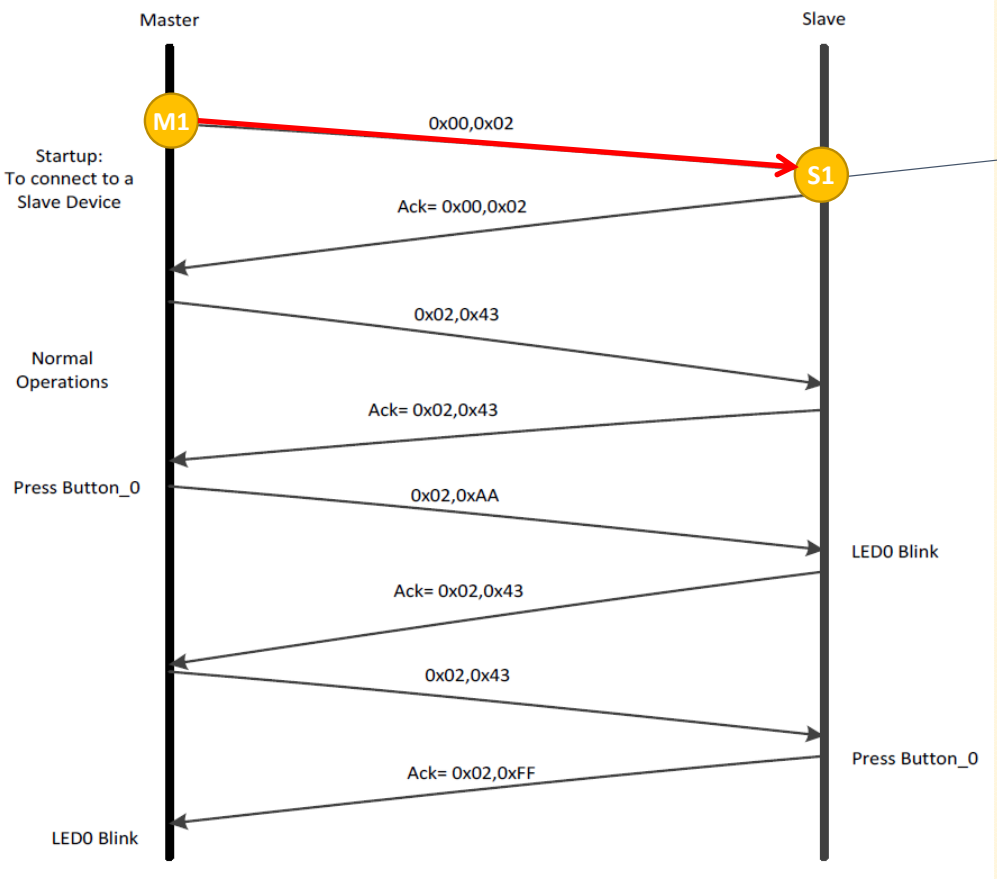
Communication and data sequence



Handshake protocol:

Software Implementation

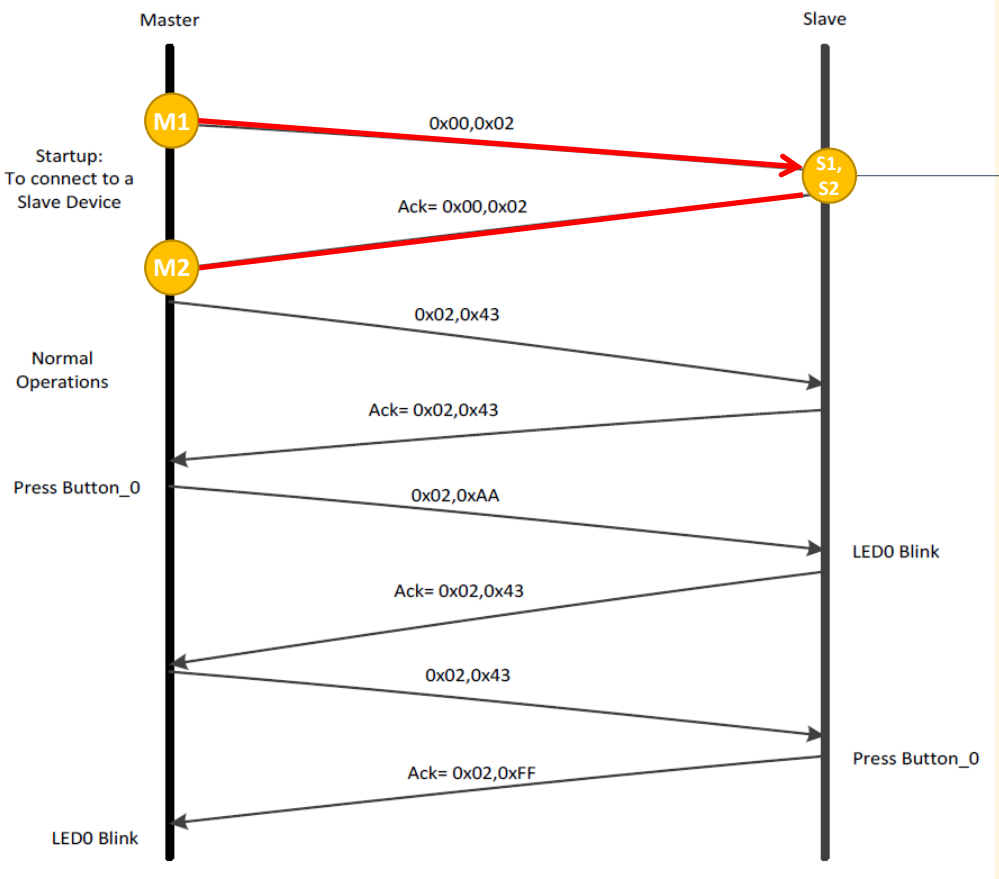
Communication and data sequence



Handshake protocol: The **slave waits** for a master signal inside the SCH_Start()

The Slave needs to be at S1 **before** the Master enters M1

Communication and data sequence



Handshake protocol:
The slave has **accepted** the signal from the master, **sends** a special "handshake" Ack message back to the master and finally **enables CAN SPI interrupts**

Normal operation:

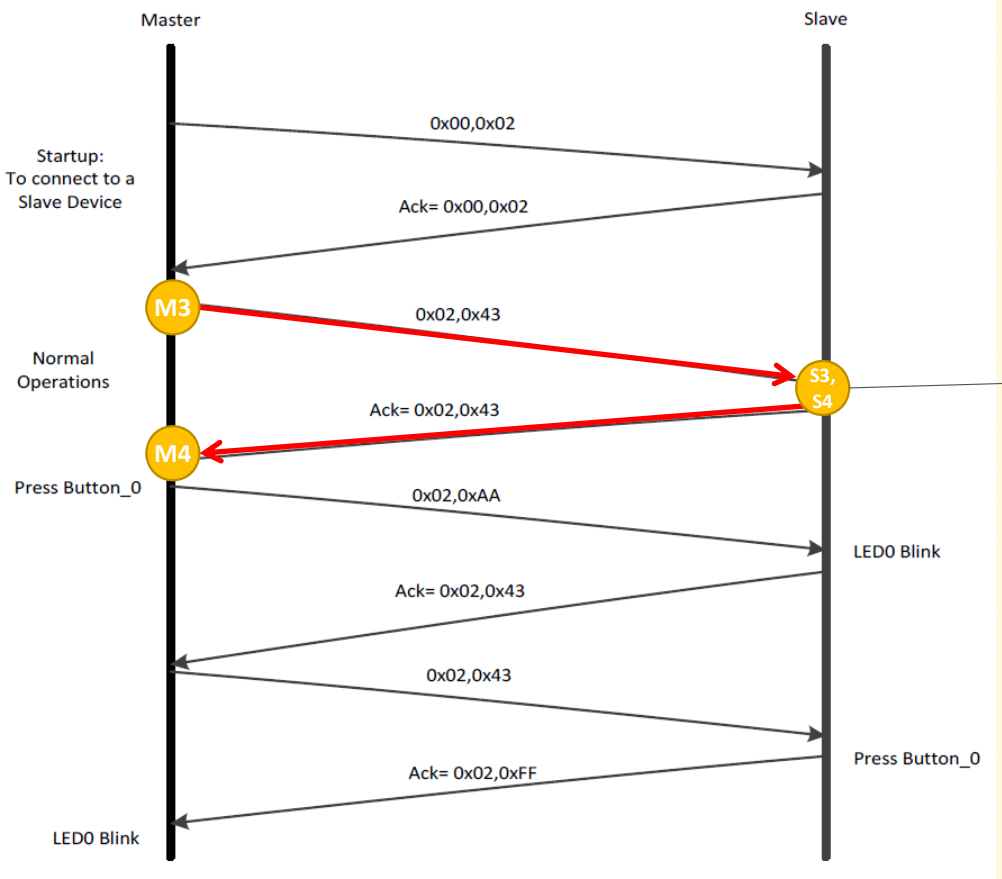
Software Implementation

Communication between Master and Slave nodes Normal Operations:

Normal Operations:

- During normal operations, the master will send a message (0x02, 0x43) to the slave.
  - The first byte of this message will be the slave ID while the second byte will be 0x43.
- On receipt of this message, the slave node will acknowledge the master with an ACK message (0x02,0x43).
  - The first byte of the ACK message will be the slave ID while the following byte will be 0x43.

Communication and data sequence



Normal Operation: No buttons are pressed

Master sends 0x02, 0x43 and the slave Acknowledges the same

Master node: `PushButton_Update()`

```
void PushButton_Update(void)
   {
   // Change the LED from OFF to ON (or vice versa)
//    IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE,
//          IORD_ALTERA_AVALON_PIO_DATA(LED_BASE) ^ LED_pin);

   static unsigned int Duration;
   IOWR_ALTERA_AVALON_PIO_DATA(TEST_1_BASE,
                               IORD_ALTERA_AVALON_PIO_DATA(TEST_1_BASE) ^ TEST1_pin);
   // Read "reset count" switch input (pb0)
   pb0_input = IORD_ALTERA_AVALON_PIO_DATA(KEY_0_BASE);

   if (pb0_input == SW_PRESSED)
      {
      Duration += 1;

      if (Duration > SW_THRES)
         {
         Duration = SW_THRES;

         Sw_pressed_G = 1;  // Switch is pressed...
         Tick_message_data_G[Slave_index_G] = 0xAA;
         return;

         }

      // Switch pressed, but not yet for long enough
      Sw_pressed_G = 0;
      Tick_message_data_G[Slave_index_G] = 'C';
      return;
      }

   // Switch not pressed - reset the count
   Duration = 0;
   Sw_pressed_G = 0;  // Switch not pressed...
   Tick_message_data_G[Slave_index_G] = 'C';

   }
```

Normal operation: Button_0 is not pressed

Store “C” (i.e. “0x43” in HEX) at the second byte of the DATA field of the Ack message

Button0 is pressed on the master node:

Software Implementation

Communication between Master and Slave nodes Button0 Press

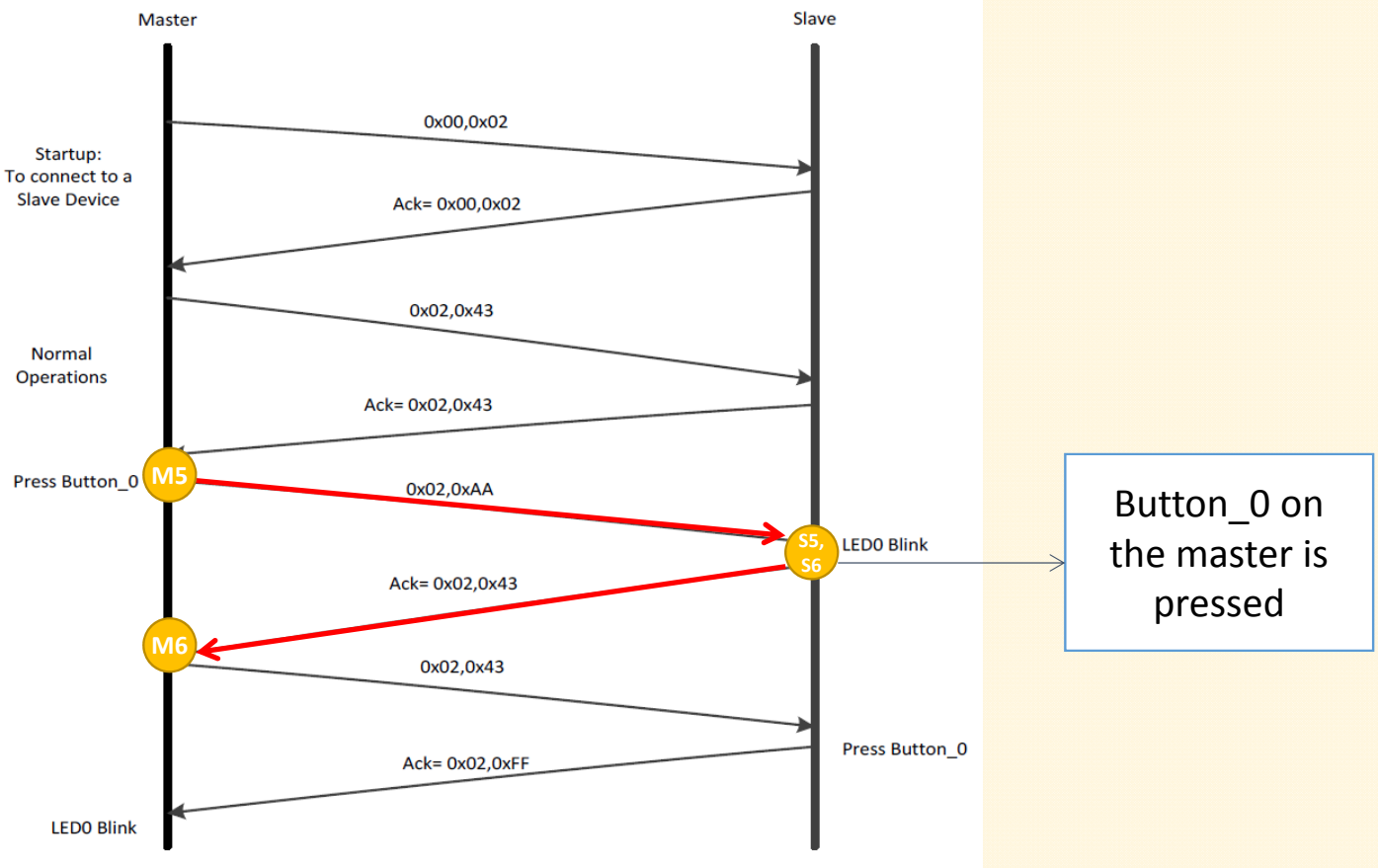
Master Button0 Press:

- On pressing the master board Button0, the master will send a message containing 0x02, 0xAA.
  - The first byte of this message will be the slave ID.
  - In receipt of this message, the slave node will send the same ACK message (0x02,0x43).
  - This will turn ON/OFF LED0 on the slave node board.

Slave Button0 Press:

- On pressing the slave board Button0, the slave will send a modified ACK message (0x02, 0xAA), which will turn ON/OFF LED0 of the master node board.

Communication and data sequence



Master node: `PushButton_Update()`

```
void PushButton_Update(void)
   {
   // Change the LED from OFF to ON (or vice versa)
//   IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE,
//         IORD_ALTERA_AVALON_PIO_DATA(LED_BASE) ^ LED_pin);

   static unsigned int Duration;
   IOWR_ALTERA_AVALON_PIO_DATA(TEST_1_BASE,
                               IORD_ALTERA_AVALON_PIO_DATA(TEST_1_BASE) ^ TEST1_pin);
   // Read "reset count" switch input (pb0)
   pb0_input = IORD_ALTERA_AVALON_PIO_DATA(KEY_0_BASE);

   if (pb0_input == SW_PRESSED)
      {
      Duration += 1;

      if (Duration > SW_THRES)
         {
         Duration = SW_THRES;

         Sw_pressed_G = 1;  // Switch is pressed...
         Tick_message_data_G[Slave_index_G] = 0xAA;
         return;

         }

      // Switch pressed, but not yet for long enough
      Sw_pressed_G = 0;
      Tick_message_data_G[Slave_index_G] = 'C';
      return;
      }

   // Switch not pressed - reset the count
   Duration = 0;
   Sw_pressed_G = 0;  // Switch not pressed...
   Tick_message_data_G[Slave_index_G] = 'C';

   }
```

PushButton0 is pressed: The master has diagnosed that PushButton0 was pressed and loads the “Slave_index_G” byte of the CAN tick data message with **“0xAA”**.

The tick message contacting “0xAA” will be sent on the next tick!

Slave node: LED_ONOFF_Update()

```
void LED_ONOFF_Update(void)
    {

     if (Tick_message_data_G == 0xAA) //Sw_pressed_G == 1)
          {
          // Change the LED from OFF to ON (or vice versa)
          IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE,
                                      IORD_ALTERA_AVALON_PIO_DATA(LED_BASE) ^ LED0_pin);
          }
     else
     {
          // Change the LED from OFF to ON (or vice versa)
          IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE, 0);
     }

    }
```

PushButton0 on master is pressed: The slave checks if the first byte of the DATA field of the tick message is “0xAA”

PushButton0 on master is pressed: The slave toggles the state of LED0

Slave node:
SCC_A_SLAVE_Send_Ack_Message_To_Master()

```
void SCC_A_SLAVE_Send_Ack_Message_To_Master(void)
{
    // Prepare Ack message for transmission to Master

    // First byte of message must be slave ID
    MCP2515_Write_Register(TXBnDm(0,0), SLAVE_ID);

    // Now the data
    MCP2515_Write_Register(TXBnDm(0,1) , Ack_message_data_G);

    /* Send RTS_TXB0_INSTRUCTION Instruction */
    MCP2515_RTS_TXB_Instruction_CMD(RTS_INSTRUCTION_TXB0);

}
```

Store the "Slave_ID" at the first data byte of the Ack message

The slave **sends** an Ack with a data field of 0x02, 0x43

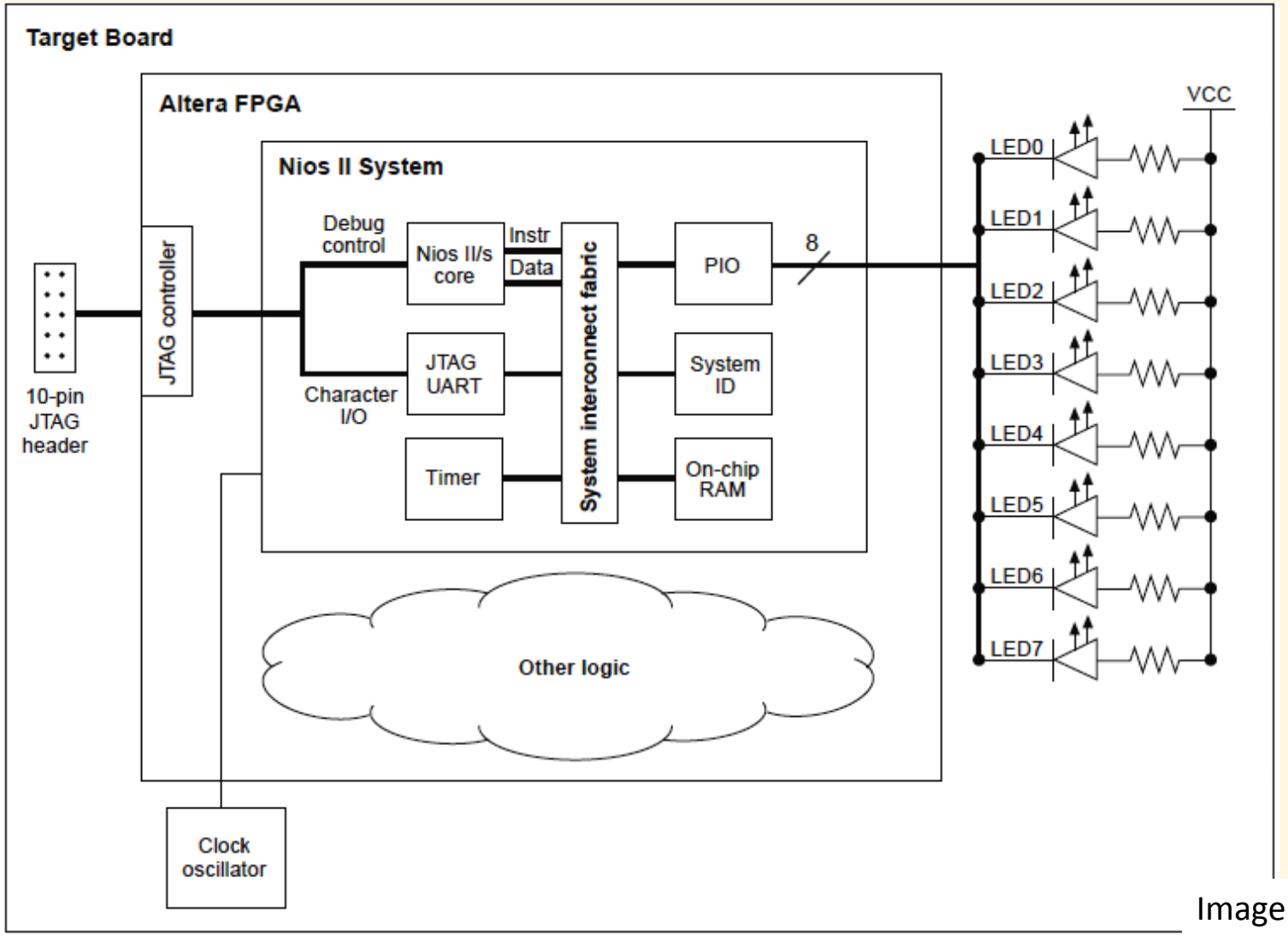
NIOS II Soft Processor Core from Intel/Altera

Features

Our focus

- Integrating a soft processor core in an FPGA is a non-trivial process
- Rather than considering this problem in an abstract way, we will focus on a popular family of IP cores: NIOS II (from Altera).
- Other microprocessor IP cores are supported in a similar way (the tools will vary but the options are similar)
  - For example, you may like to explore the PicoBlaze and MicroBlaze soft processor core options from Xilinx.

Example: NIOS II soft processor core from Altera



Nios II: Overview

- The Nios II processor system, is a configurable “soft” microcontroller, which consists of
  - a Nios II processor core,
  - a set of on-chip peripherals,
  - on-chip memory,
  - interfaces to off-chip memory (and other components),

all implemented on a single Altera FPGA.

- Nios II processor systems all use a consistent instruction set and programming model.
- Altera Nios II documentation:

  http://wl.altera.com/literature/lit-nio2.jsp

Features

NIOS II is a general-purpose RISC processor core with the following features:

- Full 32-bit instruction set, data path, and address space
- 32 general-purpose registers
- Optional shadow register sets

- 32 interrupt sources + external interrupt controller

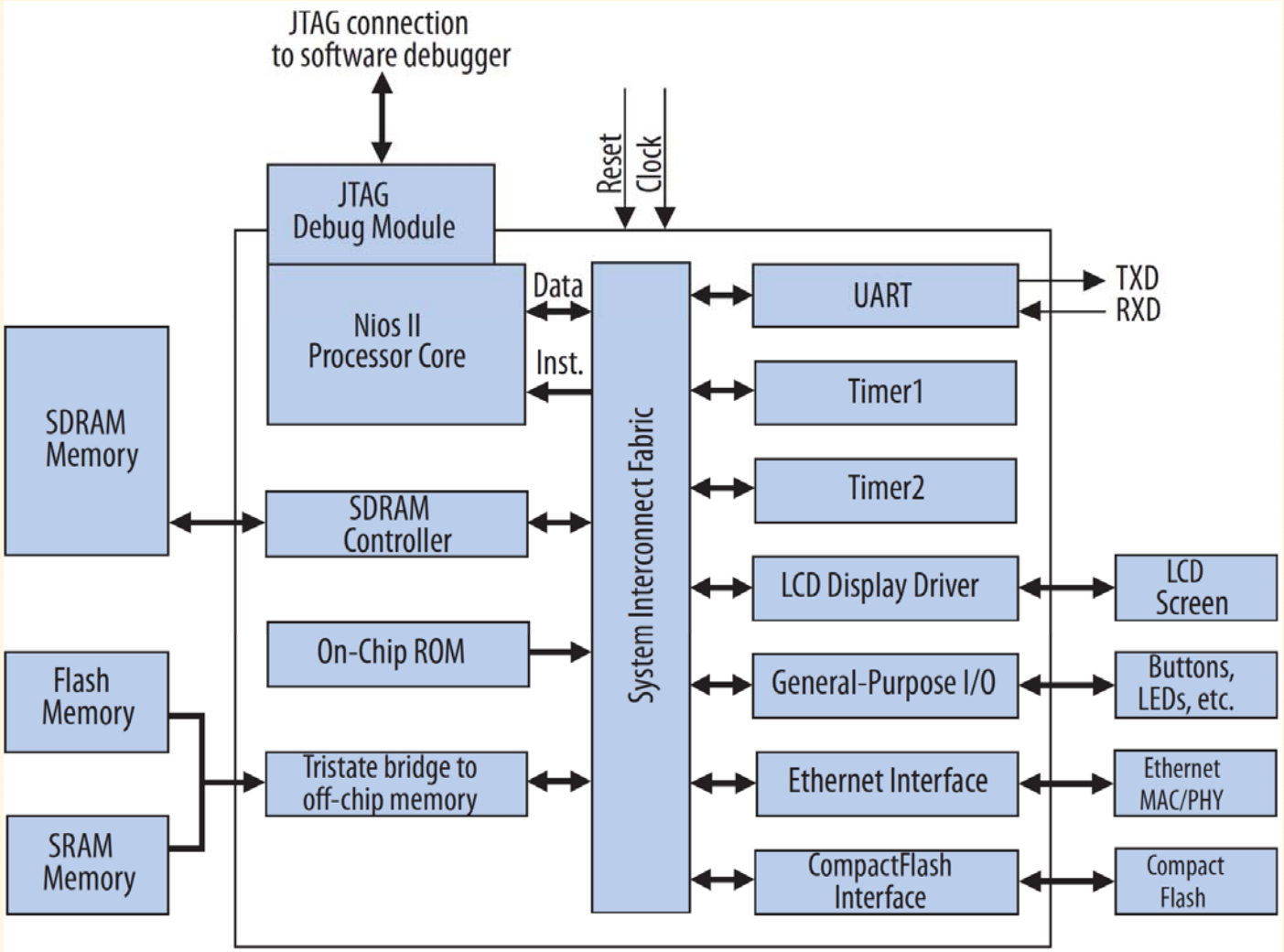
- The ability to add custom instructions

NIOS II: Memory and Peripheral Access

- The Nios II architecture provides memory-mapped I/O access.
  - Both data memory and peripherals are mapped into the address space.
- The Nios II architecture does not specify anything about the existence of memory and peripherals; the quantity, type, and connection of memory and peripherals are system-dependent.
  - Typically, Nios II processor systems contain a mix of fast on-chip memory and slower off-chip memory.
  - Peripherals typically reside on-chip, although interfaces to off-chip peripherals also exist.

Nios II Processor Reference Handbook
(May 2011)

Example of a Nios II Processor System



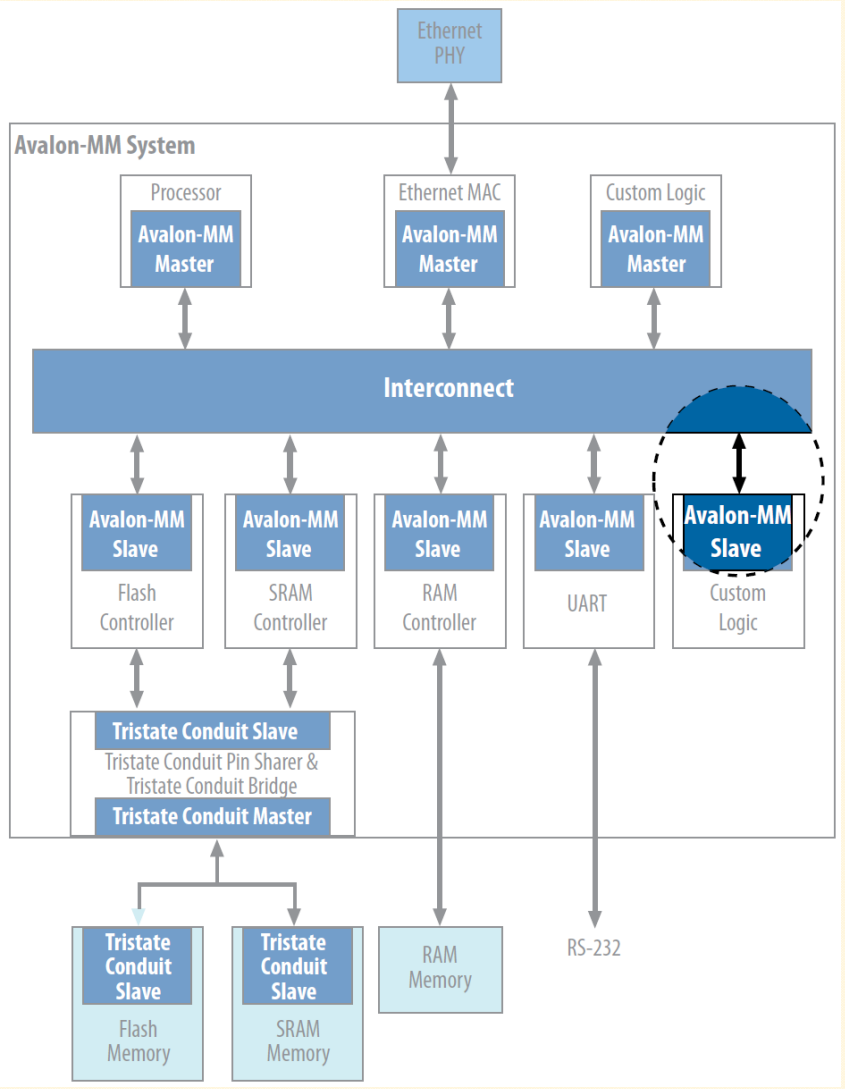
Nios II Classic Processor Reference Handbook, April 2015

Avalon-MM Slave Transfers

The Nios® II processor accesses the control and status registers of on-chip components using an Avalon-MM interface.

The figure shows a typical system, highlighting the Avalon-MM slave interface connection to the interconnect fabric.

Avalon-MM components typically include only the signals required for the component logic.



Avalon Interface Specifications, Altera, March 2015

Accessing NIOS II System Peripherals

Software Constructs

How to access the NIOS II system peripherals

- The PIO core with Avalon interface provides a memory-mapped interface between an Avalon Memory-Mapped (Avalon-MM) slave port and general-purpose I/O ports. The I/O ports connect either to on-chip user logic, or to I/O pins that connect to devices external to the FPGA.
- The PIO core support provides macros for controlling digital output and input
  - The parallel input/output (PIO) core provides a memory-mapped interface between an Avalon® Memory-Mapped (Avalon-MM) slave port and general-purpose I/O ports
- The PIO macros are defined in `altera_avalon_pio_regs.h`
- The base addresses are found in `system.h`
- The macro `IORD_ALTERA_AVALON_PIO_DATA` reads the input port, and the macro `IOWR_ALTERA_AVALON_PIO_DATA` writes the output port

```
#define IORD_ALTERA_AVALON_PIO_DATA(base)             IORD(base, 0)
#define IOWR_ALTERA_AVALON_PIO_DATA(base, data)       IOWR(base, 0, data)
```

Source: Course material by Dr I. Kyriakopolous

How to access the NIOS II system peripherals: Example

- Example: Display “8” in the DE0 HEX0 7-seg display
- Because we want to write to the PIO, we use the `IOWR_ALTERA_AVALON_PIO_DATA` macro. This takes a base address “base” and a data argument “data”
- The base address of the PIO that we want to write to is found in `system.h` and is stored in the `PIO_1_BASE` macro

```
/*
 * pio_1 configuration
 *
 */

#define ALT_MODULE_CLASS_pio_1 altera_avalon_pio
#define PIO_1_BASE 0xa040
#define PIO_1_BIT_CLEARING_EDGE_REGISTER 0
#define PIO_1_BIT_MODIFYING_OUTPUT_REGISTER 0
#define PIO_1_CAPTURE 0
#define PIO_1_DATA_WIDTH 8
...
...
```

Source: Course material by Dr I. Kyriakopolous

How to access the NIOS II system peripherals: Example

- For displaying an “8”, we need to write the value “0x80”
- Therefore, we use the `IOWR_ALTERA_AVALON_PIO_DATA` macro like that:

```
IOWR_ALTERA_AVALON_PIO_DATA(PIO_1_BASE,0x80)
```

Source: Course material by Dr I. Kyriakopolous

How to access the NIOS II system peripherals: Example

- For displaying an “8”, we need to write the value “0x80”
- Therefore, we use the `IOWR_ALTERA_AVALON_PIO_DATA` macro like that:
- `IOWR_ALTERA_AVALON_PIO_DATA (PIO_1_BASE,0x80)`

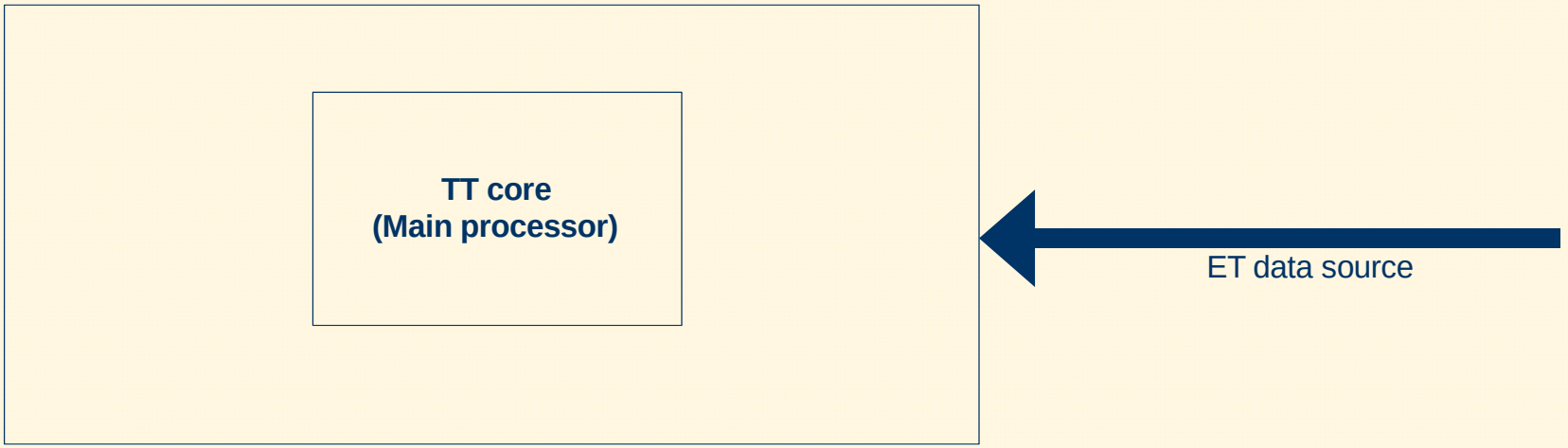
```
/*
 * PIO test
 *
 */

#include <stdio.h>
#include <unistd.h>
#include "system.h"
#include "altera_avalon_pio_regs.h"


int main()
{
  int in,  out;
  while (1)
  {
          in = IORD_ALTERA_AVALON_PIO_DATA(PIO_0_BASE);
          out = in;
          IOWR_ALTERA_AVALON_PIO_DATA(PIO_0_BASE, out);
  }
}
```

Source: Course material by Dr I. Kyriakopolous

Assignment 2:

Mutual Exclusion

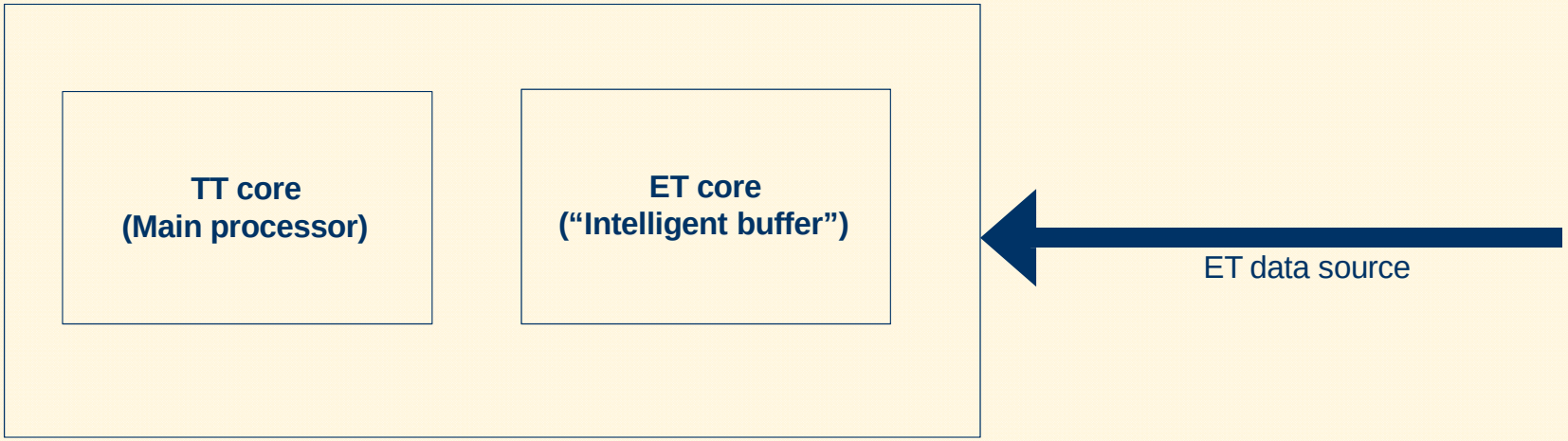
Multi-core TT solutions in an ET world …



Data arrives in bursts Long gaps between bursts
Bursts are of random duration
Gaps are of random duration

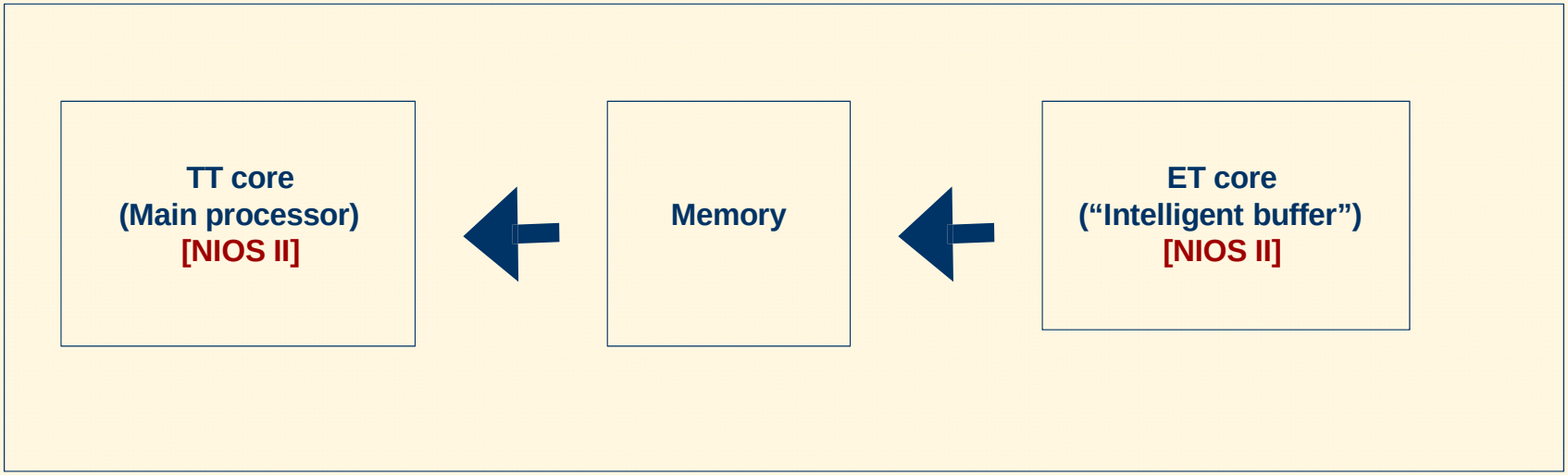
Multi-core TT solutions in an ET world …

“Buffered Input” pattern

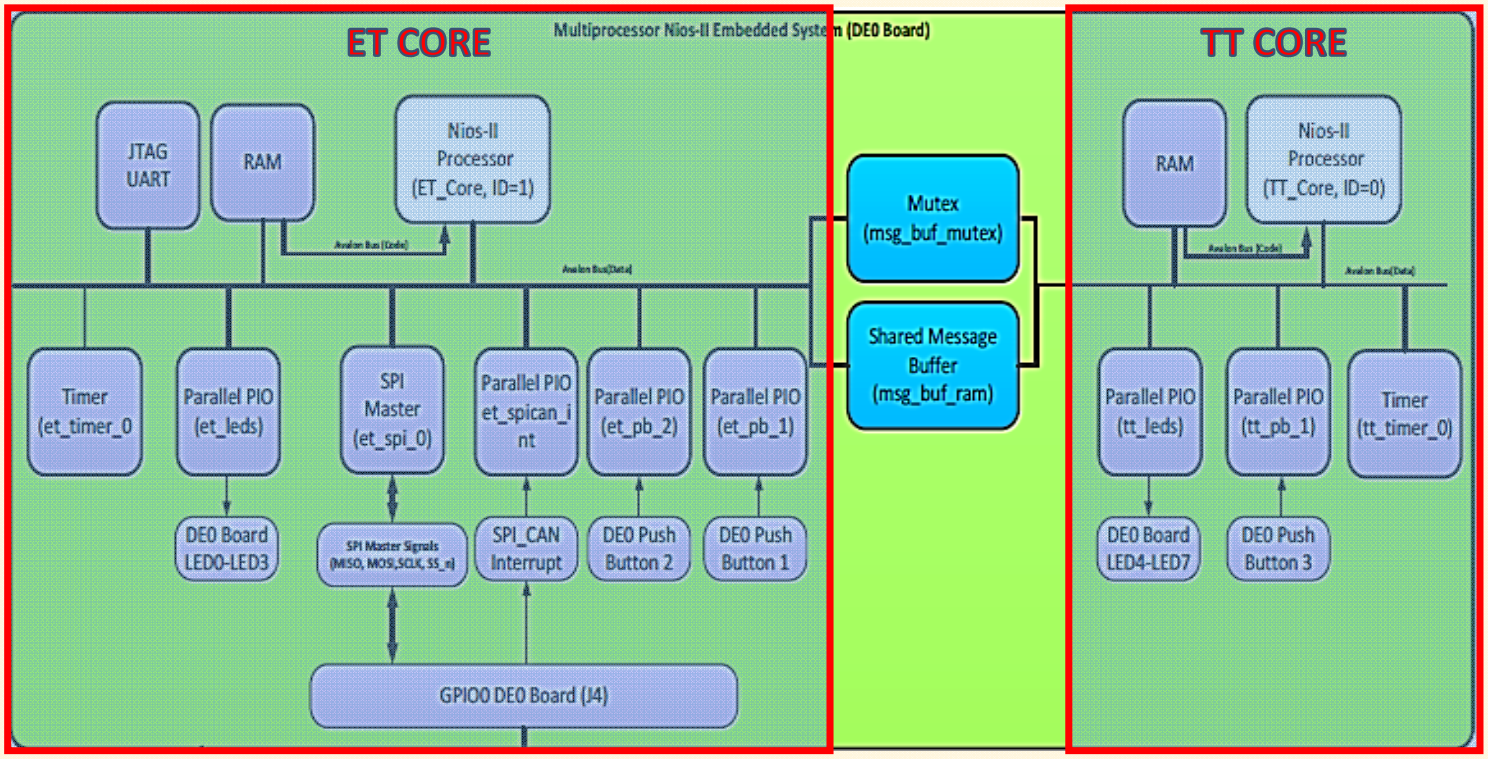


“**One interrupt per processor**”
Processor may employ multiple ET cores
(plus multiple TT cores)

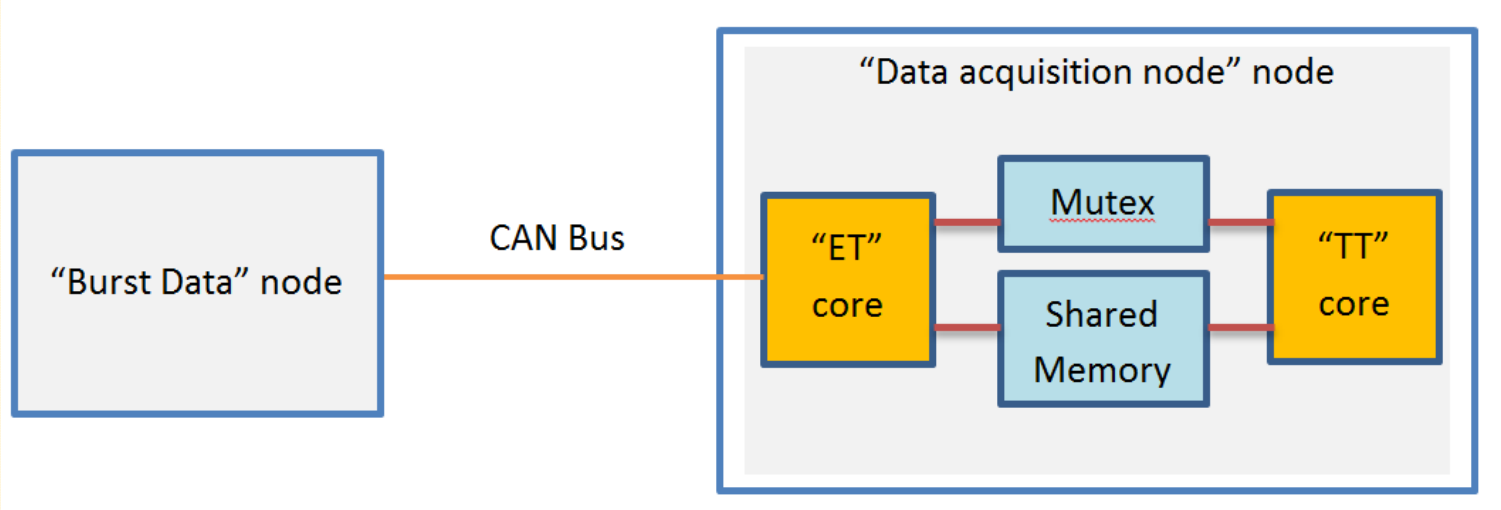
Multi-core TT solutions in an ET world …



Dual-core NIOS II System



Dual-Processor Hardware Test Bed



Mutual Exclusion: Hardware Mutex

- We want to use a **mutex** to ensure mutually exclusive ownership of a shared resource (in this case, of the **shared memory**)
  - The mutex acts like a token that is used to guard access to this resource
  - The mutex does not need to have any interaction with the shared memory however, it needs to be controlled/accessed by both cores
- In Qsys, we create a hardware mutex by adding an **Altera Avalon Mutex** component in the Qsys system
- The **Altera Avalon Mutex** component has 3 input lines:
  - The **clock input** is connected to the **system clock**
  - The **Avalon-MM slave line** is connected to the **data master lines** of the two NIOS II cores (Avalon-MM master lines)
  - The **reset line** is connected to **jtag reset lines** found on both cores and on the clock reset signal

Shared Memory

- The shared memory could be another **On-Chip Memory (RAM or ROM) component**
- The memory type can be RAM, the data width 32 bytes and the memory size 1024 bytes
- The **On-Chip Memory (RAM or ROM)** component has 3 input lines:
  - Clock input: Connected to the **clock**
  - Avalon-MM slave input: Connected to the **data master lines** of the two NIOS II cores (Avalon-MM master lines)
  - Reset input: Connected to **jtag reset lines** found on both cores and on the clock reset signal

Appendix

Lecture 5

Slave node: `SCH_Start()`

```
// Now wait (indefinitely) for appropriate signal from the master
do {
    IOWR_ALTERA_AVALON_PIO_DATA(TEST_1_BASE,
                                IORD_ALTERA_AVALON_PIO_DATA(TEST_1_BASE) ^ TEST1_pin);
    // Wait for CAN message to be received
    do {
        CAN_interrupt_flag = MCP2515_Read_Register(CANINTF);
    } while ((CAN_interrupt_flag & 0x02) == 0);

    // Get the first two data bytes
    Tick_00 = MCP2515_Read_Register(RXBnDm(1,0)); // Get data byte 0, Buffer 1
    Tick_ID = MCP2515_Read_Register(RXBnDm(1,1)); // Get data byte 1, Buffer 1

    // We simply clear *ALL* flags here...
    MCP2515_Write_Register(CANINTF, 0x00);

    if ((Tick_00 == 0x00) && (Tick_ID == SLAVE_ID))
    {
        // Message is correct
        Start_slave = 1;

        // Prepare Ack message for transmission to Master
        MCP2515_Write_Register(TXBnDm(0,0) , 0x00);      // Set data byte 0 (always 0x00)
        MCP2515_Write_Register(TXBnDm(0,1) , SLAVE_ID);  // Slave ID

        /* Send RTS_TXB0_INSTRUCTION Instruction */
        MCP2515_RTS_TXB_Instruction_CMD(RTS_INSTRUCTION_TXB0 );

    }
    else
    {
        // Not yet received correct message - wait
        Start_slave = 0;
    }
} while (!Start_slave);
alt_irq_cpu_enable_interrupts(); //can move to down
```

Handshake protocol: The **slave waits** for a master signal

Master node: `SCH_Start()`

```
// After the initial (long) delay, all (operational) slaves will have timed out.
// All operational slaves will now be in the 'READY TO START' state
// Send them a 'slave id' message to get them started
Slave_index = 0;
do {

  // Find the slave ID for this slave
  Slave_ID = (tByte) Current_Slave_IDs_G[Slave_index];

  Slave_replied_correctly = SCC_A_MASTER_Start_Slave(Slave_ID);

  if (Slave_replied_correctly)
     {
     Num_active_slaves++;
     Slave_index++;
     }
  else
     {
     // Slave did not reply correctly
     // - try to switch to backup device (if available)
     if (Current_Slave_IDs_G[Slave_index] != BACKUP_SLAVE_IDs[Slave_index])
        {
        // There is a backup available: switch to backup and try again
        Current_Slave_IDs_G[Slave_index] = BACKUP_SLAVE_IDs[Slave_index];
        }
     else
        {
        // No backup available (or backup failed too) - have to continue
        //Slave_index++;
        }
     }
  } while (Slave_index < NUMBER_OF_SLAVES);
```

Handshake protocol: The master sends a "handshake" Tick message to the slave with ID equal to "Slave_ID"

Slave node: SCH_Start()

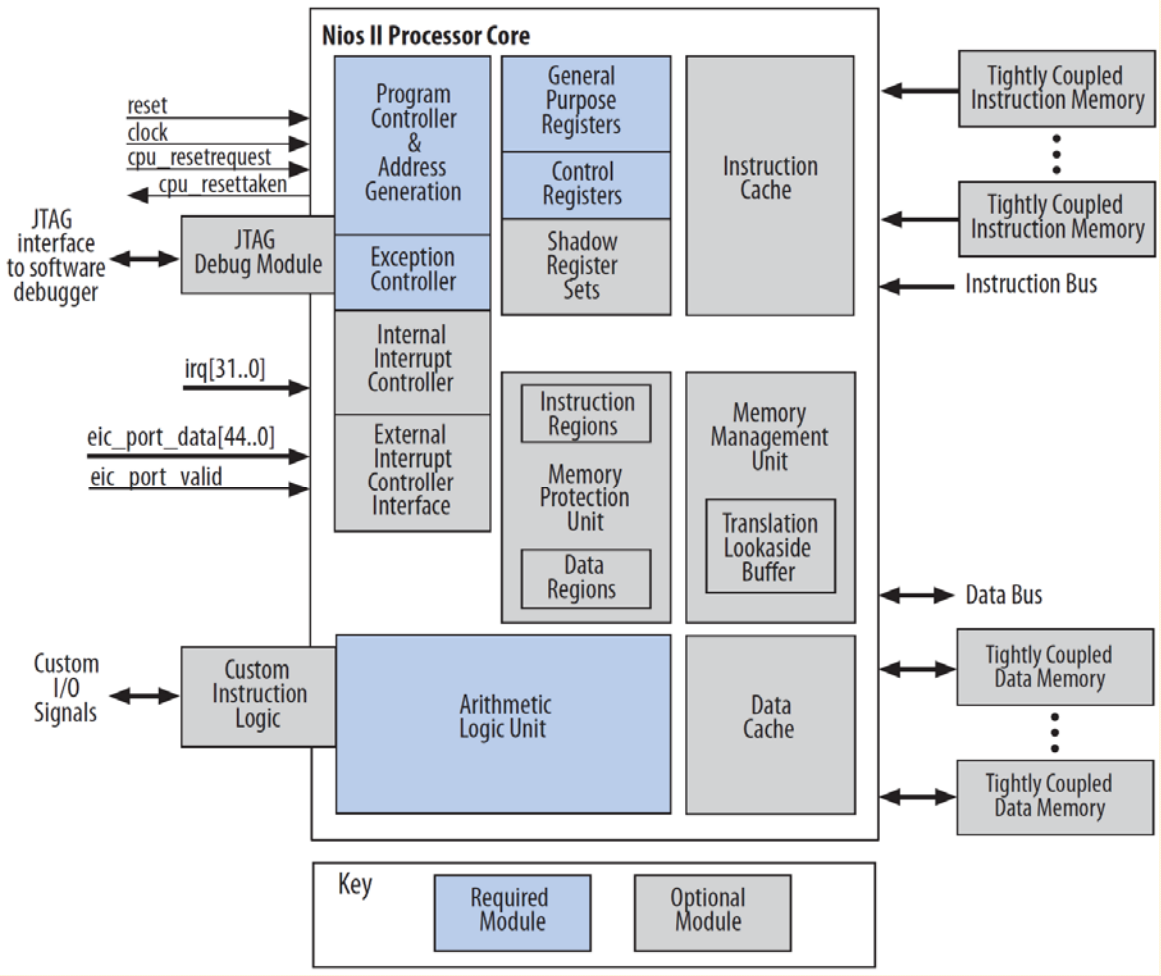
```
// Now wait (indefinitely) for appropriate signal from the master
do {
    IOWR_ALTERA_AVALON_PIO_DATA(TEST_1_BASE,
                                IORD_ALTERA_AVALON_PIO_DATA(TEST_1_BASE) ^ TEST1_pin);
    // Wait for CAN message to be received
    do {
        CAN_interrupt_flag = MCP2515_Read_Register(CANINTF);
    } while ((CAN_interrupt_flag & 0x02) == 0);

    // Get the first two data bytes
    Tick_00 = MCP2515_Read_Register(RXBnDm(1,0)); // Get data byte 0, Buffer 1
    Tick_ID = MCP2515_Read_Register(RXBnDm(1,1)); // Get data byte 1, Buffer 1

    // We simply clear *ALL* flags here...
    MCP2515_Write_Register(CANINTF, 0x00);

    if ((Tick_00 == 0x00) && (Tick_ID == SLAVE_ID))
    {
        // Message is correct
        Start_slave = 1;

        // Prepare Ack message for transmission to Master
        MCP2515_Write_Register(TXBnDm(0,0) , 0x00);      // Set data byte 0 (always 0x00)
        MCP2515_Write_Register(TXBnDm(0,1) , SLAVE_ID);  // Slave ID

        /* Send RTS_TXB0_INSTRUCTION Instruction */
        MCP2515_RTS_TXB_Instruction_CMD(RTS_INSTRUCTION_TXB0 );

    }
    else
    {
        // Not yet received correct message - wait
        Start_slave = 0;
    }
} while (!Start_slave);
alt_irq_cpu_enable_interrupts(); //can move to down
```

Handshake protocol: The slave has **accepted** the signal from the master

Handshake protocol: The slave **sends** a special “handshake” Ack message back to the master

Enable CAN SPI interrupts

NIOS II: block diagram



The functional units of the Nios II architecture form the foundation for the Nios II instruction set.

However, this does not indicate that any unit is implemented in hardware.

The Nios II architecture describes an instruction set, not a particular hardware implementation.

A functional unit can be implemented in hardware, emulated in software, or omitted entirely.

Nios II Classic Processor Reference Handbook, April 2015

Flexibility of a soft processor: Overview

- 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.
- To reduce cost, you can choose to omit the JTAG debug module.
  - This decision conserves on-chip logic and memory resources, but it eliminates the ability to use a software debugger to debug applications.
- In control applications that rarely perform complex arithmetic, you can choose for the division instruction to be emulated in software.
  - Removing the division hardware conserves on-chip resources but increases the execution time of division operations.

Nios II Processor Reference Handbook
(May 2011)

NIOS II (Memory and Peripheral Access)

- The Nios II architecture provides memory-mapped I/O access.
  - Both data memory and peripherals are mapped into the address space of the data master port.
- The Nios II architecture uses little-endian byte ordering.
  - Words and halfwords are stored in memory with the more-significant bytes at higher addresses (i.e. Little Endian*)
- The Nios II architecture does not specify anything about the existence of memory and peripherals; the quantity, type, and connection of memory and peripherals are system-dependent.
  - Typically, Nios II processor systems contain a mix of fast on-chip memory and slower off-chip memory.
  - Peripherals typically reside on-chip, although interfaces to off-chip peripherals also exist.

* the least significant portion of a value is presented first and stored at the lowest address in memory

Nios II Processor Reference Handbook (May 2011)

Exclusion

- Exclusion is a high-level concept in multi-threaded applications of guaranteeing that only N different threads are allowed to do something.
  - If N is 1, then the exclusion is referred to as MUTual-Exclusion (MUTEX, also commonly written as “mutex”), guaranteeing that two different threads of execution cannot both be executing a piece of critical code at the same time.
- A critical section of code can require exclusion due to many reasons, such as treating that section of code as
  - an atomic transaction, managing shared resources, maintaining data integrity, managing available processor loading, minimizing cache collisions, or
  - guaranteeing required operation ordering, to name just a few.

Bryon Moyer (Eds.) Real World Multicore Embedded Systems, 2013, Ch. 12

Exclusion

- There are many synchronization primitives commonly used to achieve exclusion.
- In addition to being used to implement exclusion, many of these synchronization primitives can be used for additional synchronization needs, such as:
  - a thread blocking until another thread's work is complete before it continues,
  - coordinating support threads or children threads, and so on.

Mutual exclusion

- Two semaphore options to consider:
  - Basic (binary) semaphores
  - Counting semaphores
- The other main option is a “mutex”
  - A form of semaphore which is linked to a specific task

Bryon Moyer (Eds.) Real World Multicore Embedded Systems, 2013, Ch. 12