Advanced Computer Organisation

The term 'Advanced Processing System' has become widely used when describing a large number of different applications, with a large number of platforms now available. These range from small microcontrollers and DSPs, to large FPGAs and defining what exactly comprises an advanced processing system is growing more and more challenging.

The aim of this course is to introduce the concept of advanced processing systems and to provide some examples of where they would be deployed in terms of real-life applications. The scope will then be narrowed to focus on the generic advanced processing system architecture and organisation will be explored. Individual components of the architecture will be detailed, starting with the processor, and a high-level overview of some of the fundamental operations, such as interrupts and execution cycles, will be discussed. Some basic bus operations and properties, such as bus arbitration, memory transfers and bandwidth, will also be covered.

General Organisation of Advanced Processing System

An Advanced Processing system is a specialised computing system that is optimised to carry out some dedicated functions. The processor can be regarded as the central element of the hardware system. The software system (a software 'stack') is run on the processor, comprising applications (usually based on an Operating System (OS)), and with a lower layer of software functionality for interfacing with the hardware system. Communication between system elements takes place via interconnections. These may be in the style of direct, point-to-point links, or buses serving multiple components. In the latter case, a protocol is required to manage access to the bus. In general, the architecture of an advanced processing system follows the architecture given in Figure 1

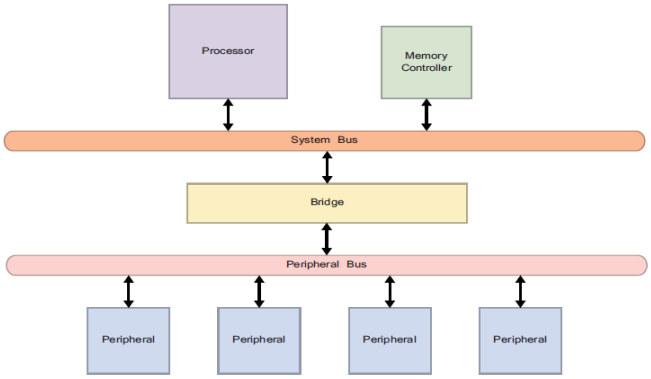


Figure.*1: General Organisation of Advanced* Processing *System*

The individual components in Figure.1 are defined as follows:

•**Processor** — This is the 'brains' of the system. It is programmed to perform the tasks specific to the application of the embedded system.

•**Memory Controller** — Memory controllers manage the reading and writing of data to and from main memory in an embedded system. Memory controllers reside in on-chip soft cores, providing an interface between the system memory and all other parts of the system.

•**Peripherals** — These are the components around the central processing unit. Peripherals can be implemented as individual integrated circuits, be contained on-chip with the processor, or may reside in an area of programmable logic such as an FPGA.

•**System Bus** — In an embedded system with multiple buses, the system bus connects the processor, memory controller and other high-speed devices together. As such, it is the bus with the greatest bandwidth in the system.

•**Peripheral Bus** — The second bus creates two separate sections of the system, allowing two arbiters to control and manage communications across the two buses. This allows devices on the peripheral bus to communicate with each other, even when a high-priority processor-memory transaction is taking place on the system bus.

Processors

A processor is the main controlling unit within an advanced processing system. It controls and orchestrates the system, supports software and co-ordinates exchanges with peripheral components. There are a variety of processors which can be used in an advanced processing system, as outlined below:

•**Microprocessor** — A microprocessor is a single chip integrated circuit which contains the complete central processing unit, and nothing else. In order to make a microprocessor functional, external memory in the form of RAM and ROM, and other peripherals must be added. The CPU in a PC is a good example of a microprocessor.

•**Microcontroller** — A microcontroller contains an entire computing system on a single chip. Unlike a microprocessor, a microcontroller comprises of a CPU with a fixed amount of RAM/ROM and peripherals all within a single Integrated Circuit (IC).

Digital Signal Processor — A digital signal processor is a processor which has been designed specifically for the task of digital signal processing, with an instruction set that is targeted accordingly. DSPs are designed to perform fast arithmetic operations, and are capable of performing a multiply-accumulate operation in a single clock cycle. This makes a DSP very efficient, both in terms of performance and power consumption, when used for specific audio/video tasks, but very poor when used for other tasks due to the restricted instruction set.

•**Embedded Processor** — An embedded processor is one which is physically embedded within the programmable fabric of an FPGA device. Embedded processors come in two varieties — *hard* and *soft* processors. Hard processors are those which are built from dedicated silicon outwith the general-purpose logic of the FPGA device, whereas a soft processor is built using the general-purpose logic of the FPGA. A soft processor must be synthesised to fit into the FPGA fabric. In the case of all embedded processors, both hard and soft, the local memory, bus interconnects, memory controllers and internal peripherals must be realised in the FPGA general-purpose logic.

Co-processors

A coprocessor is a processing core that supplements the functionality of the primary processor, and is optimised for a single, specific task. By off-loading computations from the main processor to one or more co-processing units, the overall system performance can be accelerated.

Whereas the main processor may be used for a variety of different tasks, co-processors are generally used to perform dedicated tasks. Examples of tasks which may be performed on dedicated co-processors are:

•High-speed arithmetic

•Image and video processing

•Digital signal processing

•Data encryption

With reference to FPGA based processing systems, the programmable logic provides a perfect platform in which to create co-processing cores due to the ability to perform parallel execution. This means that complex tasks that would require a large number of sequential CPU clock cycles to compute can be executed far quicker in a processing based co-processor. These are known as *soft coprocessor* cores. Other forms of acceleration can be implemented using dedicated hard processing cores, that do not reside in the FPGA fabric. Collectively, this task off-loading processing is known as *hardware acceleration.*

Processor Cache

A cache is a small memory located between the CPU and main memory. It has a lower access time than main memory and the cache is not accessible via the system bus. Cache is used to store data that is frequently accessed by the processor from main memory. Operations which make use of cached data are therefore much faster than those where the data is in main memory only.

As processors typically read data at a much faster rate than the system's main memory, the processor speed is constrained to that of the memory. By including cache memory in a system — which contains the most frequently accessed data that can be read at a higher rate than main memory — the processor is no longer constrained to the speed of the main memory. This leads to an increased efficiency in terms of data access. The processor speed is still limited, however, in the event of a cache miss — whereby the processor fails in its attempt to read or write data in the cache. This results in the data being read/written to main memory, and increased access latency.
A number of different levels of cache can exist within a system. These are outlined in Figure.2, and expanded upon below.

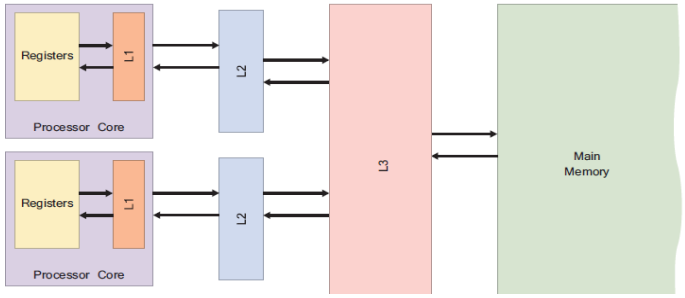


Figure.2

It is useful to first introduce the two types of memory which are used — Dynamic RAM (DRAM) and Static RAM (SRAM) — before going on to discuss the different levels of cache.

Dynamic RAM (DRAM)

DRAM is the most common form of memory that is used in computing systems. A DRAM chip consists of a number of *memory cells* which can each hold 1-bit of data, stored in a capacitor. Every memory cell also features a transistor, which acts as a switch to allow the control circuitry to read or write the state of the capacitor. As both the capacitor and transistor are extremely small, millions of individual memory cells can fit in a single DRAM chip.

Due to the fact that capacitors leak electric charge, the state of the bit of information held by each memory cell will eventually fade unless the charge of the capacitor is periodically refreshed by the

memory controller. The memory controller does this by reading the state of each memory cell and then writing the state back again. This is where dynamic RAM gets its name.

Static RAM (SRAM)

SRAM uses a different technology to store information than DRAM. Whereas each bit of memory in DRAM is stored in a capacitor, SRAM uses latches to store the data. Each memory cell requires 4 or 6 latches to store a single bit of data, and therefore requires much more space on a chip than DRAM, making it more expensive.The upside to SRAM is that it does not need to be refreshed, thus making it much faster than DRAM. Due to the expense of SRAM, it is usually only used in high-speed, low-capacity memory chips.

Level 1 (L1) Cache

L1 cache is the smallest form of cache memory, the size of which is typically 8 to 128 KB. It is implemented in the form of SRAM which is built into the fabric of the processor core and, as such, operates at the same clock rate. L1 cache typically consists of two sections: *data* and *instruction* caches. L1 cache is used to store a local copy of frequently accessed data and instructions, enabling instant access by the processor.

Level 2 (L2) Cache

L2 cache is usually external to individual processing cores, but is located extremely close by. It is larger than L1 cache, typically in the region of 256 to 1024 KB, but has slower access speeds. L2 cache is in the form of DRAM and is unified in a single section (unlike L1 which is split into two sections). Larger quantities of data are constantly read in by L2 cache from main memory before being fed to L1.

Level 3 (L3) Cache

Level 3 (L3) cache is shared among all processor cores. It is also implemented in DRAM, and is the largest form of cache memory, typically in sizes of 2 MB and upwards.

Execution Cycles

In order for a program that is stored in memory to be executed by the processor it must go through an *instruction execution cycle*. This is the process by which a system retrieves an instruction from memory, determines the required actions for that instruction, and executes those actions. Each instruction execution can be divided into three unique parts, as shown in Figure.3. For obvious reasons, this process is sometimes referred to as the *fetch-and-execute* cycle or the *fetch-decode-execute cycle*.

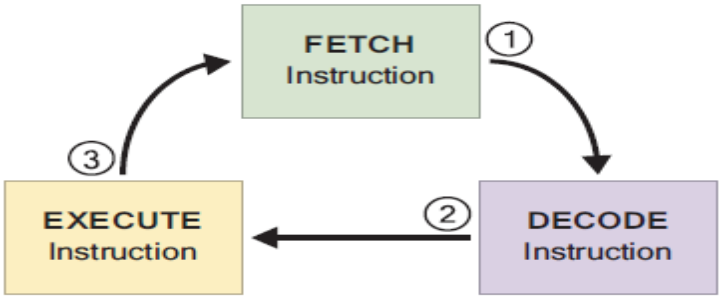


Figure.3: Steps of the instruction execution cycle

Before taking a closer look at each of the stages of the instruction execution cycle in detail, it is useful to understand some of the terms that will be used:

•**Machine Code** — When writing a software application, it is often in a high-level language (such as C/C++) which is easily understood by the developer, and human-readable. Code in this form is meaningless to a processing system, and must be converted, or compiled, into a form that the processor can understand. The final, low-level, output that is readable by the processor is known as machine code – a stream of binary data relating to the program that the processor is able to interpret and process.

•**Opcodes** — An opcode is an operation code which uniquely defines a function to be performed — a machine code representation of a processor instruction. A processor can execute many different operations, so each instruction is assigned a unique numeric code.

•**CPU Instruction Set** — The instruction set for a given processor is the basic set of commands which a processor can understand. It contains a specification of each of the opcodes and native commands that can be performed by that processor.

Fetch Instruction

The fetch instruction is the first stage of the instruction execution cycle. The flow of the operation is outlined in Figure.4.

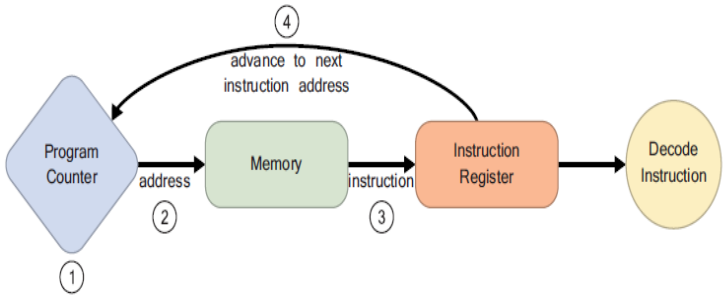


Figure.*4:* Fetch *instruction flow*

With reference to Figure.4, the fetch instruction flow is as follows:

1.The program counter register contains a value that corresponds to an address in memory containing the next instruction to be processed.

2.This value is transferred to the memory address register where the control unit checks the value and fetches the corresponding instruction from memory.

3.That instruction is then stored in the memory buffer register before being transferred to the instruction register.

4.The program counter register is advanced by the control unit so that the value matches the memory location of the next instruction to be executed.

Decode Instruction

Once the instruction has been fetched from memory, the next step is to get the instruction into a form that the processor can understand. This is the decode process. At this point, the instruction that is stored in the instruction register is checked by the control unit. This detects the opcode and addressing mode which have been used, and consequently what actions must be performed to correctly execute the instruction.

There are three main addressing modes:

•**Immediate Addressing** — This requires no lookup of data as all of the required data is located within the operands of the instruction. This makes immediate addressing the fastest, but least flexible mode.

•**Direct Addressing** — Operands of instruction contain the memory address of the required data. Required data must be fetched from this address.

•**Indirect Addressing** — Operands of instruction contain a memory address. Within this memory address is a pointer to the address where the required data is stored. This makes indirect addressing the most flexible, but slowest due to the two data lookups.

Execute Instruction

Depending on the opcode that was determined by the decode process, a number of different actions can be performed at this stage. In general, there are four main groups of actions:

•Transfer of data between the processor and memory.

•Transfer of data between processor and I/O devices.

•Processing data, such as using the Arithmetic Logic Unit (ALU).

•Control operations that change the sequence of subsequent operations. These can be conditional based on the values in a certain flag register.

Once the instruction has been executed, the next instruction to be processed is fetched.

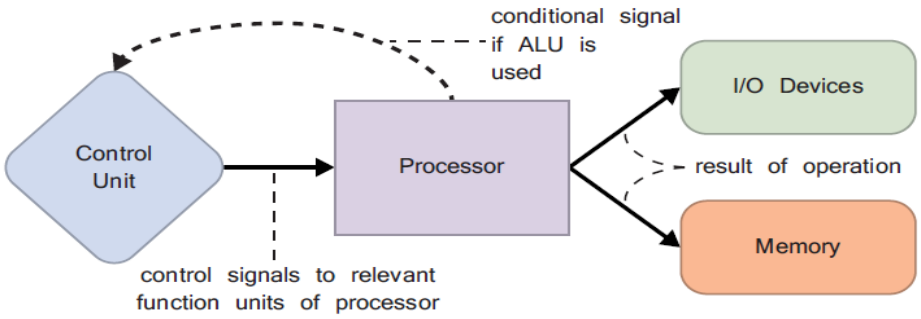


Figure.*5: Execute instruction flow*

The flow of an execute operation cycle is shown in Figure.5.

With reference to Figure.5, the flow is as follows:

1.The control unit passes decoded information from the decode process (as a sequence of control signals) to the required functional units of the processor. This could be, for example, to read values from registers, or writing to a register.

2.The processor performs the required operations, and the output is either stored in memory or sent to an output device.

3.If the ALU is used, a conditional signal is relayed back to the control unit. As an example, the signal could update the address of the program counter so that the next instruction is fetched.

Interrupts

An interrupt is a signal which is generated to indicate to the processor that its attention is required. Interrupts can be generated by hardware processing units and peripherals, and also within the software itself. In hardware, an interrupt signal is an asynchronous signal that is generated by a processing unit to indicate the need for attention from the processor. In software, an interrupt is also an asynchronous event which indicated to the processor that a change in execution is needed. The process of polling the generated interrupts, however, is synchronous.

When the processor receives an interrupt it will halt the task that is currently being processed, and jump to the one which has requested attention. This is in contrast with the process of polling, which is the synchronous sampling of a device's status by the software program. Instead of having the processor constantly poll the I/O ports of a device to see if its attention is needed, the device will interrupt the processor.

Hardware interrupts can be further categorised into the following types:

•**Maskable Interrupts (IRQs)** — The trigger event of a masked interrupt is not always important. It is the task of the programmer to decide whether the event should cause the program to jump to the requested execution or not. Examples of devices which may use maskable interrupts include timers, comparators and ADCs.

•**Non-Maskable Interrupts (NMIs)** — These are interrupts which should never be ignored, and are therefore deemed much more important than maskable interrupts. Events which require NMIs include power-on, external reset (from a physical button) and serious device faults.

•**Inter-Processor Interrupts (IPIs)** — In multiple processor systems, one processor may need to interrupt the operation of another processor. In this situation, an IPI will be generated.

Buses

A bus provides the interconnection by which processors interface with other processors and peripherals. Processors, memory controllers and peripherals connect to the bus via a standard bus interface. The particular bus interface will depend on the particular bus architecture that is in use, but at a basic level a bus consists of address, control (read/write requests and acknowledgements) and data signals [2]. This is detailed in Figure.6, where the three main bus signals are highlighted:

•The **address line** transfers the memory addresses, or target port identification numbers, to I/O devices.

•The **control line** governs the control and timing signals of a system, synchronising operations and transmitting control signals such as interrupt requests and acknowledgements.

•The **data line** is responsible for the transfer of data.

Data transfer occurs in a *bus cycle.*

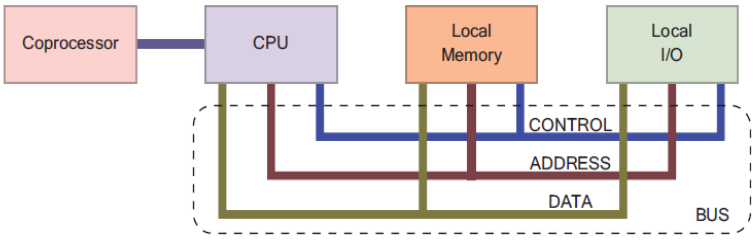


Figure.6

Any coprocessors that are present in an embedded system are directly connected, or closely coupled, to the main processor via the processor bus. In a multi-processor system, individual processor subsystems are connected by the *system bus*.
Buses also contain a bus arbiter which controls access to the bus. Not all connected modules can request access to the bus, and they are categorised accordingly as *bus masters* and *bus slaves* [2].

System and Peripheral Buses

In larger system designs, it may be appropriate to use multiple buses to provide sufficient bandwidth for communication between all processing and peripheral cores. In such systems, the two main types of bus are the *system bus* and the *peripheral bus*.

In system design, the system bus is the main method of providing communication between the various peripheral and processing cores. In smaller system designs, the system bus may be the only bus that is present within the design. In larger, multiple-bus designs, the system bus will be the bus with the largest bandwidth that connects the memory controller and the processor, as well as any high-speed devices. All remaining peripherals, which do not require such high-speed access to the processor and memory controller, will be connected via the peripheral bus.
In order to split the embedded system design into separate domains, a second bus — known as the peripheral bus — can be added. This can be for a number of reasons, such as making the distinction between low-speed and high-speed devices or to provide dedicated bandwidth for the communication between a group of peripheral cores. This allows a set of peripherals to communicate between themselves in parallel with any communication on the system bus, such as a processor-memory access.

Bus Bridge

In order to allow the communication between devices on separate buses, such as the processor (on the system bus) requesting data from a peripheral core (on the peripheral bus), a *bridge* between the two buses is required. A bridge is connected to both buses and communicates requests between them. On one bus the bridge is connected as a bus master, while having a slave connection on the other.

Bus Masters and Slaves

The modules that connect to the buses in an embedded system can be split into two distinct categories:

•**Bus Masters** have the ability to request access to the bus and, as such, are responsible for initiating data transfers by driving the address and control signals. Bus cycles can be initiated, and other bus modules informed of the bus cycle type.

•**Bus Slaves** do not possess the ability to initiate bus cycles, and instead only monitor bus activity. Signals on the address and control lines are decoded, and when addressed, can either place, or accept, data on/from the bus.

Bus Arbitration

As a bus is shared amongst all devices in an embedded system, there is a requirement to determine which bus master device is permitted to use the bus at any one time. The method for determining this access is called *bus arbitration*. If more than one master device requests access to the bus at the same time, it is the job of the bus arbiter to decide which device should be granted access first. In simple terms, the requesting master device with the highest priority (lowest assigned number) will be granted access to the bus first, provided a bus cycle is not already in progress. Lower priority masters (higher assigned numbers) will be placed in a waiting queue, and processed upon completion of the higher priority requests.

Memory Access

The way in which memory controllers are accessed in an embedded system can have a huge effect on overall performance. Even if a very efficient type of memory and memory controller is used, the system performance could suffer from poor memory access control. It is important that the system is structured, and accessed, in a way which will maximise the memory bandwidth whilst keeping the required resources to a minimum.

Programmable Input/Output (I/O)

One way of controlling the movement of data between the memory controller and other peripherals is to route all data transfers via the processor. This method of memory transfer is known as programmable I/O, and allows the system to process memory transfers with a minimum amount of resources. This approach requires that the peripheral and the processor are situated on the same bus, and the processor acts as a central point of communication between all peripherals and the memory. If the number of memory transfer requests between peripherals and memory is high, the processor could spend a lot of time performing memory transfers and less time performing other computations [2].

Programmable I/O is an effective way of managing memory transactions, whilst using a minimum amount of resources, provided that the system implements most of the functionality in the programmable logic. If, however, the processor is required to perform a large number of other computations, other methods may be preferable [2].

Direct Memory Access (DMA)

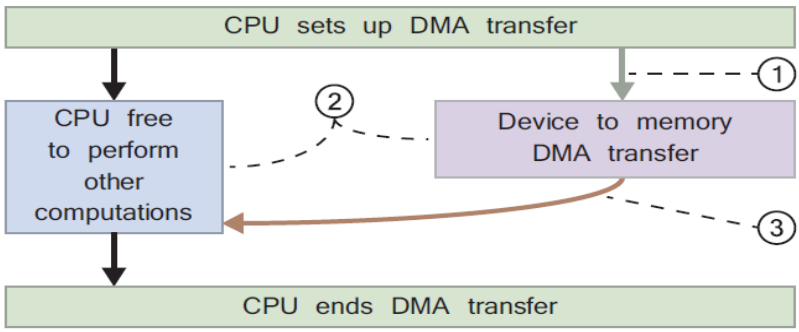
One way of reducing the burden on the processor is to use Direct Memory Access (DMA) to perform memory transfers. Using this approach, the processor issues a memory transfer request to the DMA controller, which will then perform the memory transaction. This allows the processor to perform other tasks while the DMA controller performs the transfer. In this situation, the DMA controller acts as both a bus master and a bus slave. As a master, the DMA controller communicates with the memory controller whilst also arbitrating for the bus. While acting as a slave, the DMA controller sets up memory transfers by responding to requests from bus masters (the processor, in most cases). In order to initiate the transaction, the DMA controller must be provided with the following information [2]:

•**Source address** — the address where the data will be read from.

•**Destination address** — the location where the data should be written to.

•**Transfer length** — the number of bytes that should be transferred.

An overview of a DMA memory transfer is provided in Figure.7.



With reference to Figure.7, the DMA memory transfer process is:

1.The processor sets up the device wishing to use the DMA to transfer data to memory by issuing a DMA command and disables all DMA interrupts.

2.The DMA controller transfers data from the peripheral device to memory leaving the CPU free to perform other computations.

3.Following completion of the data transfer, an interrupt is sent to the CPU to inform it that it can close the DMA transfer.

Bus Bandwidth

Bus bandwidth is the total amount of data that can be transferred on a bus in a given unit of time. The value for bus bandwidth depends on two parameters:

•**Bus data width** — this is the number of physical lines over which the bus transfers data simultaneously. A bus with 32 individual data lines can transmit 32 bits of data simultaneously.

•**Bus frequency** — this is the speed at which the bus operates. This refers to the number of data bits which can be transmitted/received per second. This is defined in Hertz (Hz).

The relationship between these parameters and bus bandwidth is given in Equation (1).

Bus Bandwidth (Mbits/s) = Bus Width (bits) X Bus Frequency (MHz) ---- (1)

As an example, a bus with a data width of 32 bits operating at 10 MHz has a bus bandwidth and, therefore, a maximum throughput of:

Bus Bandwidth (Mbits/s) = 32 bits X 10 MHz =320 Mbits/s (40 Mbytes/s)

Obviously, the more peripherals that are connected to a bus, the higher the required throughput on the bus will be. Therefore, it is important that the bus bandwidth is sufficiently high in order to prevent the system from saturating the bus.

Summary

In this course, the concept of an advanced processing system has been introduced and a generic advanced system architecture has been explored. The role of processors within systems has been discussed, and some of the processing features, such as processor cache and execution cycles have been covered. The function of coprocessors was also introduced as well as the use of software/hardware interrupts.
Communication between all components within a system is reliant on the inclusion of a bus system, and their functionality has been discussed in this chapter. The requirement for multiple bus systems was introduced, and the distinction between bus master and slave devices was reviewed. Bus arbitration and memory access techniques were summarised, and the importance of bus bandwidth was discussed.

References

[1]D. Liu, "Introduction" in *Embedded DSP Processor Design: Application Specific Instruction Set Processors*, Morgan Kaufmann, 2008, pp 1 - 46.

[2]R. Sass and A. G. Schmidt, "Managing Bandwidth" in *Embedded Systems Design with Platform FPGAs: Principles and Practices*, 1st. Ed, Morgan Kaufmann, 2010, pp 295 - 346.

[3]R. Sass and A. G. Schmidt, "System Design" in *Embedded Systems Design with Platform FPGAs: Principles and Practices*, 1st. Ed, Morgan Kaufmann, 2010, pp 115 - 196.