Class Notes: Embedded Computer Systems

Muhammad Obaidullah
M.A.Sc. Candidate, Ryerson University, mobaidullah@ryerson.ca

I. Lecture 1: 7th September 2016

A. Assesment & Evaluation

Lecture, Projects and other Material are available at the course website: http://www.ee.ryerson.ca/~courses/ee8205/

Lab Project:	20%
Project:	40%
Final Exam:	40 %

B. Definitions & Classifications

Embedded System: Any System which performs certain useful function(s) and has some form of information processing machine **in it** (Embedded) is called an embedded system. Embedded Systems are not subset of general purpose computers. **Real-time System:** Any System which responds to externally generated stimulus within a finite and specific time is known as real-time system. If that system makes use of a information processing machine (eg. CPU), then it is also an embedded system. Most real-time systems are also embedded systems.

C. Real-time Systems

1) **Soft real-time:** Systems where deadlines are important but the system won't suffer in case of missing deadlines occasionally. For example: weather data acquision system (no major issue if data for some minutes is not recorded).
2) **Firm real-time:** Systems where deadlines are important. However, there is no benifit from late delivey of service. For example: UDP live video transmission (some video frames can be skipped).
3) **Hard real-time:** Systems where deadlines are extremely important and response should occur within a specific deadline. For example: flight control system (system has to repond immediately to compensate tilt).
4) **Real real-time:** Systems where deadlines are extremely important and missing one deadline will cause system faliure. For example, missile guidance systems.

D. Multi-Tasking & Concurrency

Task: A thing to do within the application. Any function or collection of functions which are performed in one context is called a task. A task is a basic unit of programming that an operating system controls. Depending on how the operating system defines a task in its design, this unit of programming may be an entire program or each successive invocation of a program.

Process: A process, in the simplest terms, is an executing program. One or more threads run in the context of the process. A process has its own memory space to store variables. If a process needs to share some variables with other processes, it needs to do it manually. See figure 1.

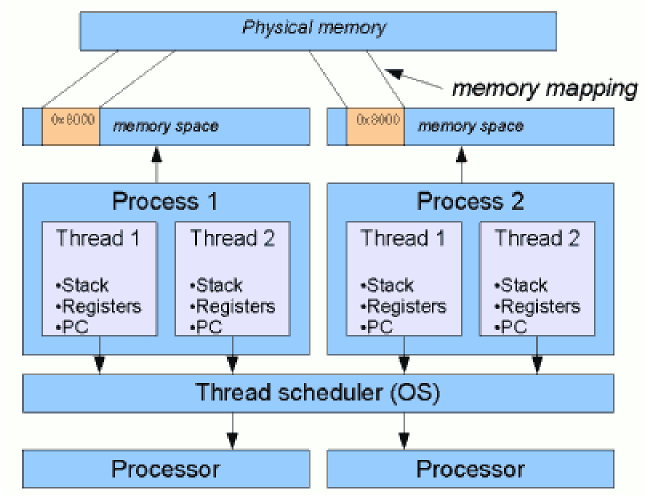


Figure 1: Differences between a process and thread.

Thread: A thread is the basic unit to which the operating system allocates processor time. A thread can execute any part of the process code, including parts currently being executed by another thread. Threads can share variables easily as they are using the same memory space.

Context: Context can be seen as a bucket to pass information around. It is typically used to pass things not necessarily tied directly to a method call, but could still be pertinent. A layperson way of describing it might be s̈tuff you may care about.̈ For example, if you were writing a service to update a value in a db, you'd probably pass in the record id, and the new value.

Multi-Tasking: It is an ability of a information processing machine to perform multiple functions simultaneously.

Concurrency: To occur or exist simultaneously or side by side. However, real-world embedded systems give an appearance of having multiple tasks running concurrently while in reality they are switching fast between them. This is to allow multi-tasking on a single CPU core.

Semaphore: It is a variable or abstract data type that is used for controlling access, by multiple processes, to a common resource in a concurrent system such as a multiprogramming operating system.

Mutex: Short for **MUT**ual **EX**clusion object. A mutex is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously. A mutex is like a lock. A thread can lock it, and then any subsequent attempt to lock it, by the same thread or any other, will cause the attempting thread to block until the mutex is unlocked. See figure 2.

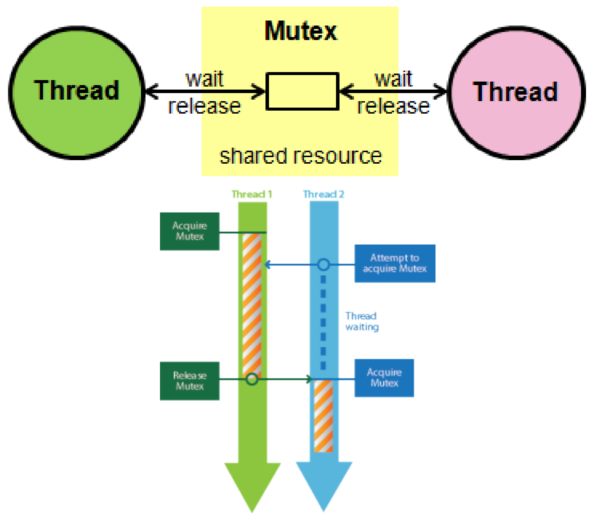


Figure 2: Using a shared object to lock access to critical part of program.

Semaphore vs Mutex: A mutex can only be released by the thread which has ownership, i.e. the thread which previously called the Wait function, (or which took ownership when creating it). A semaphore can be released by any thread.

II. Lecture 2: 14th September 2016

A. *Characteristics of Real-time System*

The largest and most complex embedded system is International Space Station (ISS) and has 20 million lines of Ada code. Real world applications are parallel. Thus, embedded systems are required to perform many tasks at the same time. If parallelism is natively not supported by embedded systems, concurrent desgin can give the appearance of parallelism.
Functional Requirements: Requirements of the system which defines how the embedded systems should work and task it should be able to perform. Speed, deadlines, user-interface, and input/output formats are some examples of functional requirements.
Non-functional Requirements: Requirements which are not related to how the embedded system should operate or work. Manufacturing cost, power, and time-to-market are some examples of non-functional requirements.

B. *Performance Paradox*

Custom logic design, advanced micro-processor power control features, and software design can help reduce power consumption. However there is trade-off between power consumption, chip area, and speed of the embedded system.

C. *Homogenity of Embedded Systems*

1) **Heterogenous System:** Systems which contain different components which perform specific tasks are called heterogenous systems. Eg. CPU + GPU System
2) **Homogenous Systems:** Systems which contain similar components which perform general tasks are called homogenous systems. Eg. Quad-core CPU

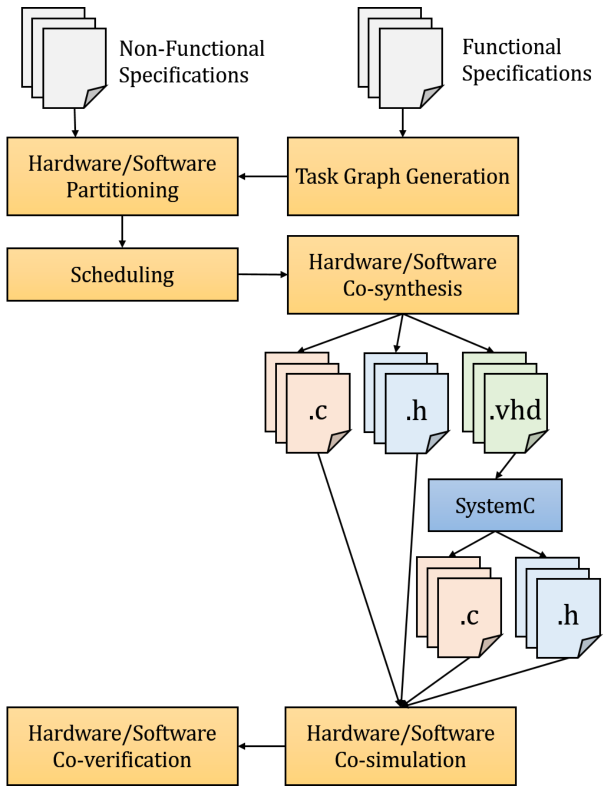
D. *Design Methodologies*

1) **Top-down Design:** Start from overall system design and reach lower hardware and software level.
2) **Bottom-Up Design:** Start by developing software and hardware of the system first and then make the overall system design and connect components together.

Hardware-Software Co-design

E. *Co-design*

Separate hardware and software design has been explored and examined very throughly. But if we combine the design of hardware and software together, there are several advantages of doing this. Time-to-market shortens and several perfomance non-functional requirements can be incorporated into design phase and tasks can be decided to be either performed on hardware or software. The entire workflow for codesign is given in figure 5.


Non-Functional Specifications
Functional Specifications
Hardware/Software Partitioning
Task Graph Generation
Scheduling
Hardware/Software Co-synthesis
.c
.h
.vhd
SystemC
.c
.h
Hardware/Software Co-verification
Hardware/Software Co-simulation
Figure 3: Hardware-Software Co-design workflow.

Why Hardware-Software Co-design?
Implementing entire application in software makes meeting performance requirements difficult. Therefore, intensive portions (computationally complex) of the application are implemented in specific hardware. This specific hardware maybe implemented on an FPGA or an ASIC. When such specific hardware is embedded into a system, it is called an accelerator.

F. *Hardware-Software Partitioning*

Hardware-software partitioning is the task to decide which portion of the application is to be implemented where. It is done in the early stages of system design. Therefore, top-down design is prefered in hardware-software codesign. For example, in image encoding application, loading and

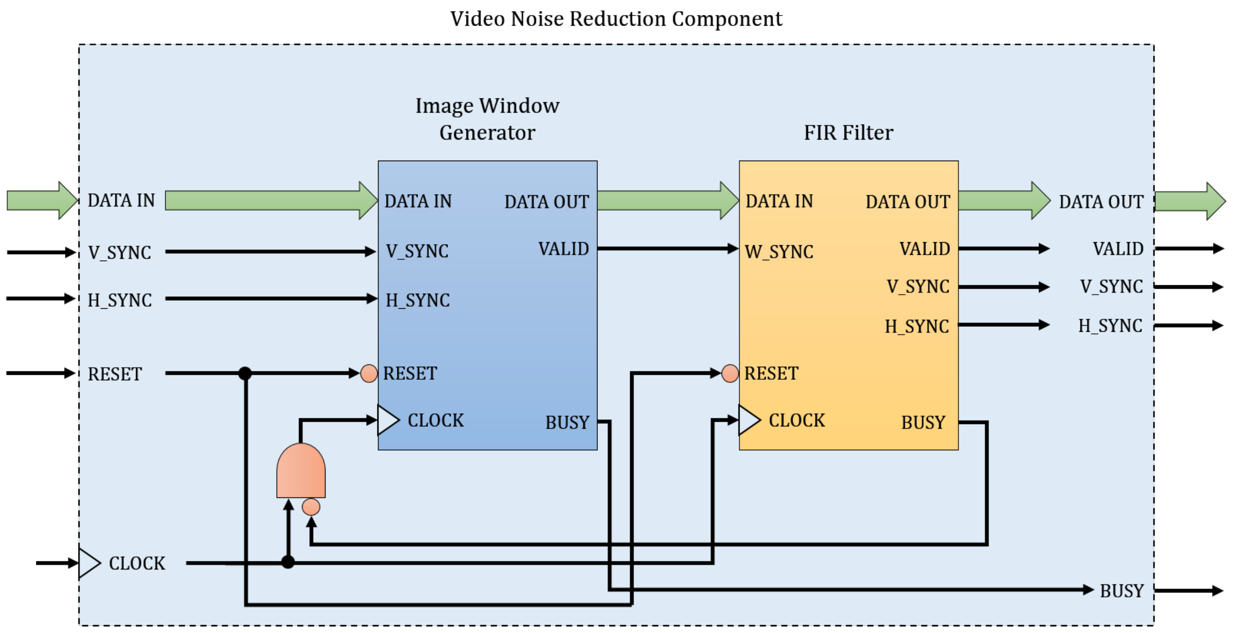


Figure 4: A video noise reduction accelerator implemented in VHDL.

managing the data into memory should be done by software while preforming FFT(Fast-Fourier Transform) should be done by hardware accelerator (FPGA or ASIC).

Accelerator: In computing, hardware acceleration is the use of computer hardware to perform some functions more efficiently than is possible in software running on a more general-purpose CPU. It is pereferable to assign computationally intensive tasks (eg. calculating a derivative) to accelerators while logically intensive tasks (eg. switch statements) to CPU. An example video noise reduction accelerator is shown in figure 4.

HW-SW Co-specification: It is description of system at abstract level which describes hardware specification and software specification as well. Often SystemC is used to describe the system. System description is converted into a task graph representation where nodes are tasks and arrows represent data transfer from task to task and the weight of arrow indicates volume of data transfer.

HW-SW Co-Synthesis: Synthesis (making) hardware and software together. When a designer writes VHDL code and C++ code for the same application to enable working of the entire system.

HW-SW Co-Simulation: Simulation of hardware and software together. When a designer treis to see how the VHDL code and C++ code for the same application are working using a simulator environment or language (SystemC).

HW-SW Co-Verification: Verifying that the hardware and software for the application is working and coperating with each other. This is also usually done using SystemC.

G. Scheduling

Scheduling is to determine when tasks start execution and on what device.

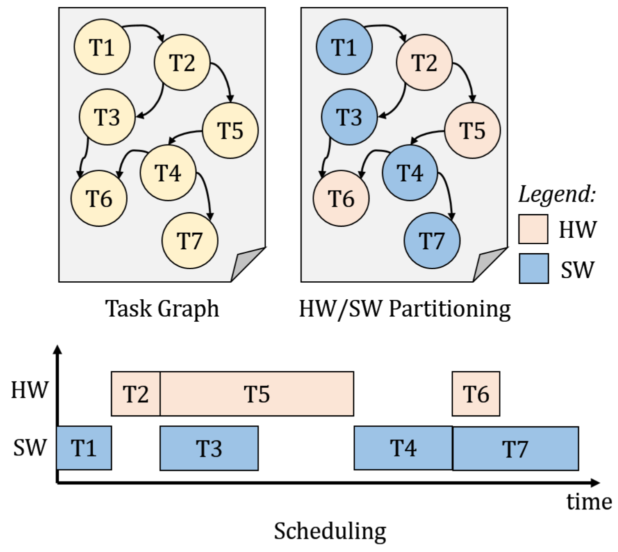


Figure 5: Scheduling of an application task graph.

H. Phases of Co-synthesis

Cosynthesis is automatic or semi-automatic design of hardware and software modules to meet the specifications. Figure 6 explains the stages.

- **Partitioning** Dividing the functionality of an embedded systems into units of computation. A large task is divided into smaller steps and an algorithm is developed. Doing these smaller steps or computations lead to the large task being done.
- **Scheduling** After smaller units of task are identified, it is now time to decide in what order will these tasks be done.
- **Allocation** Assigning processing elements (PEs) to the tasks is known as allocation. In other words, deciding which task will be done which PE.
- **Mapping** Choosing a type of PE to do group of tasks

allocated to it in previous stage. A PE can be a CPU, GPU, FPGA, ASIC or any custom hardware which is capable of doing any useful computation.

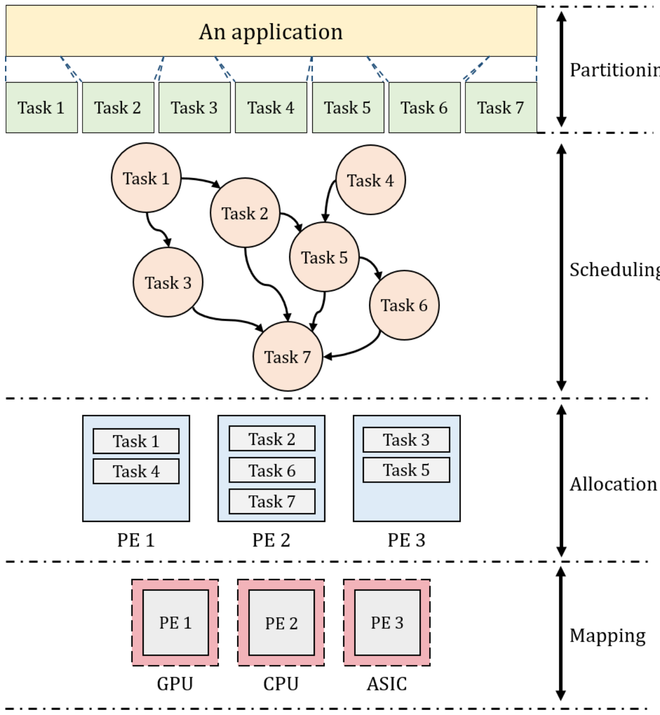


Figure 6: Phases of Co-synthesis.

III. Lecture 3: 21st September 2016

A. *What is SystemC?*

It is a C++ library which includes several C++ classes and macros for event-driven simulation. It was developed to model, verify, and simualte hardware/software systems together.

B. *C++ vs C*

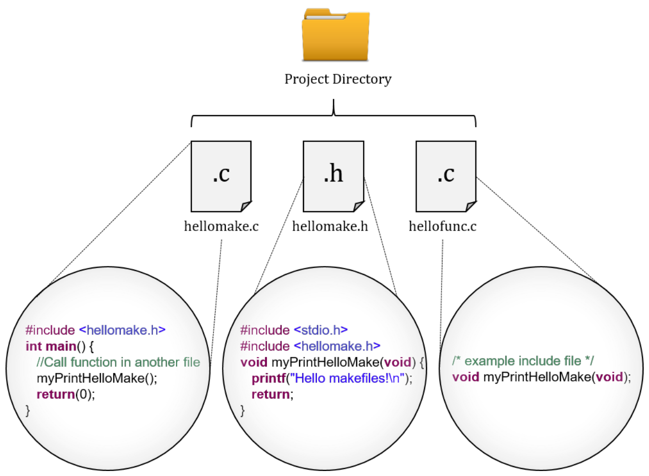
C++ extends standard C language to include classes, objects and other features which allow programmers to better organize their code and offers modularity in form of object oriented programming. When C programs grow large, it gets difficult to organize and structure the code. Also, it gets difficult to read and debug the code as program size grows large. The closest C comes to object oriented programming is by use of structs.

C. *Makefile*

Compiling source code files can be tedious, specially when programmers wants to include several source files and has to type the compiling command everytime for compiling.

Let's start off with the following three files, hellomake.c, hellofunc.c, and hellomake.h, which would represent a typical main program, some functional code in a separate file, and an include file, respectively. Normally, you would compile this collection of code by executing the following command:

```
gcc -o hellomake hellomake.c hellofunc.c -I.
```



```
#include <hellomake.h>
int main() {
  //Call function in another file
  myPrintHelloMake();
  return(0);
}
```

```
#include <stdio.h>
#include <hellomake.h>
void myPrintHelloMake(void) {
  printf("Hello makefiles!\n");
  return;
}
```

```
/* example include file */
void myPrintHelloMake(void);
```

Figure 7: Example C/C++ project structure.

This compiles the two .c files and names the executable hellomake. The -I. is included so that gcc will look in the current directory (.) for the include file hellomake.h. Without a makefile, the typical approach to the test/modify/debug cycle is to use the up arrow in a terminal to go back to your last compile command so you don't have to type it each time, especially once you've added a few more .c files to the mix.

Unfortunately, this approach to compilation has two downfalls. First, if you lose the compile command or switch computers you have to retype it from scratch, which is inefficient at best. Second, if you are only making changes to one .c file, recompiling all of them every time is also time-consuming and inefficient. So, it's time to see what we can do with a makefile. The simplest makefile you could create would look something like:

```
hellomake: hellomake.c hellofunc.c
    gcc -o hellomake hellomake.c hellofunc.c -I
```

If you put this rule into a file called Makefile or makefile and then type make on the command line it will execute the compile command as you have written it in the makefile. Note that make with no arguments executes the first rule in the file. Furthermore, by putting the list of files on which the command depends on the first line after the :, make knows that the rule hellomake needs to be executed if any of those files change. Immediately, you have solved problem 1 and can avoid using the up arrow repeatedly, looking for your last compile command. However, the system is still not being efficient in terms of compiling only the latest changes. A complete sample make file is given below:

```
IDIR =../include
CC=gcc
CFLAGS=-I$(IDIR)

ODIR=obj
LDIR =../lib

LIBS=-lm

_DEPS = hellomake.h
DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))

_OBJ = hellomake.o hellofunc.o
OBJ = $(patsubst %,$(ODIR)/%,$(_OBJ))


$(ODIR)/%.o: %.c $(DEPS)
```

```
	$(CC) -c -o $@ $< $(CFLAGS)

hellomake: $(OBJ)
	gcc -o $@ $^ $(CFLAGS) $(LIBS)

.PHONY: clean

clean:
	rm -f $(ODIR)/*.o *~ core $(INCDIR)/*~
```

D. Setting-up SystemC

1) Download the SystemC from http://accellera.org/downloads/standards/systemc
2) Set the SYSTEMC environment variable to the directory of the SystemC installation. eg. If your installation directory is /usr/local/packages/systemc-2.2.0, run the following command in terminal.

```
% export SYSTEMC=/usr/local/packages/systemc
    -2.2.0
```

3) Once the environment variable is set, you can access the SystemC installation by referring to the SYSTEMC variable. The standard GNU C++ compiler g++ is then used to compile SystemC code and link it against the SystemC libraries. Note that if your are compiling on a 64-bit LRC machine, you will have to supply the -m32 flag (to match the LRC SystemC installation, which is 32-bit):

```
% g++ -m32 -I$SYSTEMC/include  L$SYSTEMC/lib-
    linux   lsystemc   lm <source>
```

4) It is recommended that you create a Makefile for compiling your SystemC sources first to object (.o) files and then linking everything together into a final simulation executable.
5) Include the header in your source code:

```
#include <systemc.h>
```

E. T-Flip Flop in VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity tff_sync_reset is
    port (
        data   :in  std_logic; -- Data input
        clk    :in  std_logic; -- Clock input
        reset  :in  std_logic; -- Reset input
        q      :out std_logic  -- Q output

    );
end entity;

architecture rtl of tff_sync_reset is
    signal t :std_logic;
begin
    process (clk) begin
        if (rising_edge(clk)) then
            if (reset = '0') then
                t <= '0';
            else
                t <= not t;
            end if;
        end if;
    end process;
    q <= t;
end architecture;
```

F. T-Flip Flop in SystemC

```
#include "systemc.h"

SC_MODULE (tff_sync_reset) {
  sc_in<bool> data;
  sc_in<bool> clk;
  sc_in<bool> reset;
  sc_out <bool> q;
  bool q_l ;
  void tff () {
    if (~reset.read()) {
      q_l = 0;
    } else if (data.read()) {
      q_l = !q_l;
    }
    q.write(q_l);
  }
  SC_CTOR(tff_sync_reset) {
    SC_METHOD (tff);
      sensitive << clk.pos();
  }
};
```

IV. Lecture 4: 28th September 2016

A. ROM-based & RAM-based FPGAs

RAM-based FPGAs can be reprogrammed easily while RAM-based FPGAs can only be programmed once. We use RAM-based FPGAs to design and verify the design and ROM-based FPGAs to release the final design. Altera and Xilinx FPGAs are RAM-based.

B. Types of CPU Architectures

1) **Von Neumann**: CPUs which have a single interface to the memory. This interface is shared to access instructions and data. They typically require more time to get the complete data (ie. insturction and data). needed to execute next intruction.

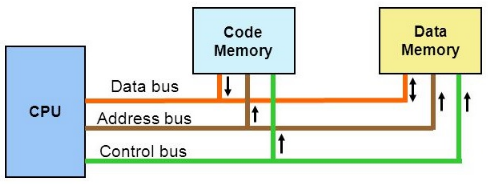


Figure 8: Von Neumann CPU Architecture.

2) **Harvard**: CPUs which have a two interfaces to the memory (dual-port memory). One port to access instructions and other to access data. These CPUs are typically faster than Von Neumann architecture because of data access parallelism.

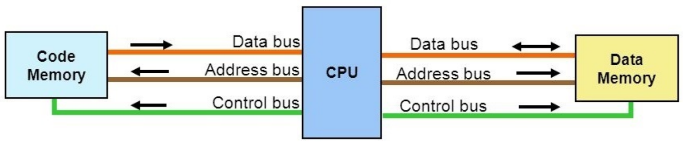


Figure 9: Harvard CPU Architecture.

C. Buses

For many decades people have speculated that Moore's Law might not be true in the next few years, but every time technological advancement has surprised us all. The number of cores on a single die are increasing as a result of advancement in nanometer technology. 14nm chips were already developed by Samsung in December 2012 and research is going on for breaking the 10nm barrier. On-chip bus architecture is one of the crucial components for platform-based SoC design.

Cramming more and more transistors onto single die has lead the IC designers to integrate not one but many modules and cores onto single chip and inter connecting them. Like a computer needs to talk to a peripheral device for input, output, or signal processing, similarly cores and modules inside the chip need to talk to each other for passing information.

Conventionally, modules were directly connected using buses but as number of modules increased, the complexity of the chip grew. To deal with complexity, abstraction and regularity of design is required. IC designers began to think of a network to topology to connect these cores and modules. Common network topologies to interconnect cores include bus, ring, two-dimensional mesh, and crossbar. Popular bus architectures include AMBA bus -ARM's bus architecture, CoreConnect IBMs bus architecture (for PowerPC), Wishbone Common architecture for open source IP design.

To sum it all, a bus is a shared communication link. It is single set of wires used to connect multiple subsystems.

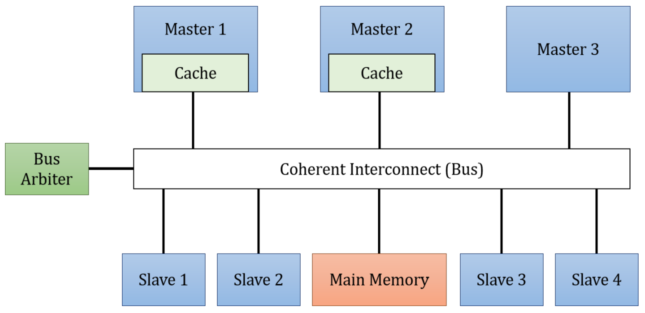


Figure 10: A Typical bus in an embedded system.

1) **Bus Advantages**
   - **Versatility:** New devices can be added easily. Peripherals can be moved between computer systems that use the same bus standard.
   - **Low Cost:** A single set of wires is shared in multiple ways. Less chip area (wires occupy very small chip area), power consumption, and complexity.
2) **Bus Disadvantages**
   - **Communication Bottleneck:** Bandwidth of that bus can limit the maximum I/O throughput.
   - **Limited Bus Speed:** Frequency of operation depends on bus length (if synchronous bus). The number of devices on the bus (and, hence, bus loading) limit the bus speed.
3) **Bus Masters & Slaves** Master is one who starts the bus transaction by issuing the command (and address). Slave is the one who responds to the address by sending data to the master asking for data or by receiving data from the master if the master wants to send data.

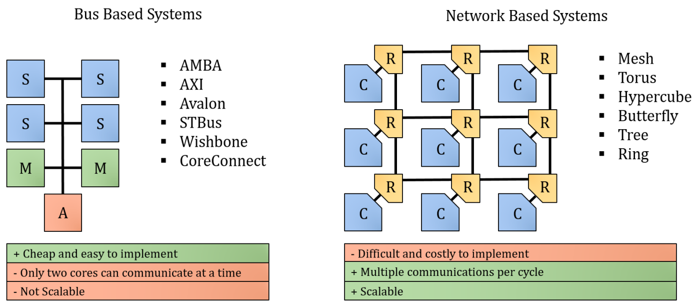


Figure 11: Bus-based system vs network based systems.

4) **Synchronous vs Asynchronous** In a synchronous bus, operations are synchronized to a global bus clock while in an asynchronous bus, a control signal edges (triggers) signal bus events. On-chip buses are generally synchronous.

5) **Avalon Bus** It is a multiplexer based bus from Altera. This means that data path is multiplexed instead of circuit switched. The Avalon bus module (an Avalon bus) is a unit of active logic that takes the place of passive, metal bus lines on a physical PCB. Since the bus is active, it is less prone to fan-out, fan-in problems. Also, adding peripherals to bus is easier since it means adding addional logic into the bus. Hence, it is used majorly in FPGA systems where it should be easier to add or remove bus nodes.

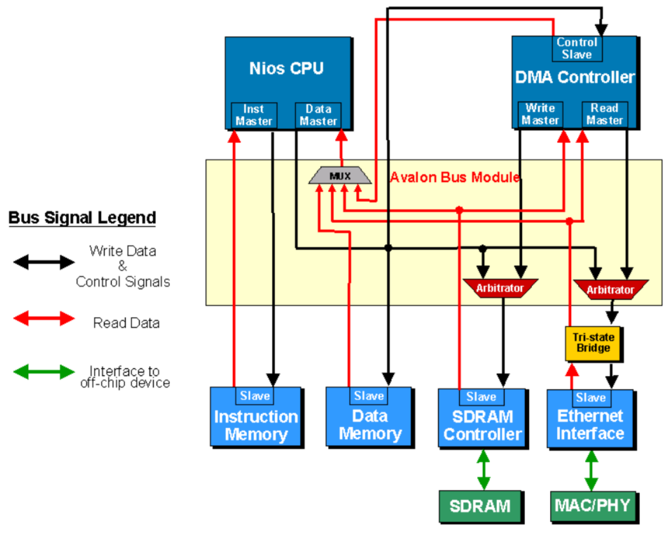


Figure 12: Avalon Bus Architecture.

6) **AMBA Bus** The ARM Advanced Microcontroller Bus Architecture (AMBA) is an open-standard, on-chip interconnect specification for the connection and management of functional blocks in system-on-a-chip (SoC) designs. It facilitates right-first-time development of multi-processor designs with large numbers of controllers and peripherals.
   - **CHI - Coherent Hub Interface** - The highest performance, used in networks and servers.
   - **ACE- AXI Coherency Extensions** - Used in ARM

big.LITTLE systems for smartphones, tablets, etc.

- **AXI - Advanced eXtensible Interface** - The most widespread AMBA interface. Connect 100s of Masters and Slaves in complex SoCs
- **AHB - Advanced High-Performance Bus** - The main system bus in microcontroller usage
- **APB - Advanced Peripheral Bus** - Minimal gate count for peripherals
- **ATB - Advanced Trace Bus** - For moving trace data around the chip, see ARM CoreSight

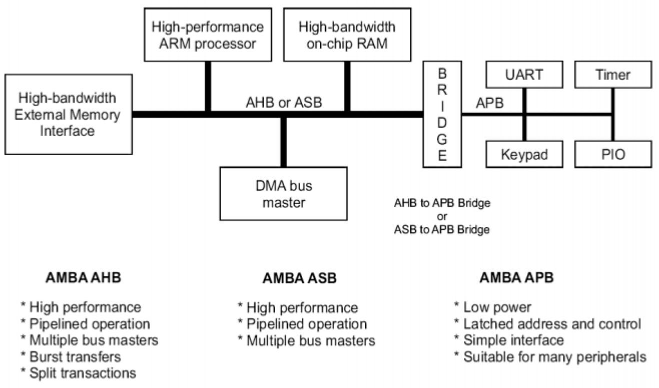


Figure 13: Advanced MCU Bus Architecture (AMBA).

7) **PCI-E Bus** Peripheral Component Interconnect Express (PCIe or PCI-E) is a serial expansion bus standard for connecting a computer to one or more peripheral devices. Every device that's connected to a motherboard with a PCIe link has its own dedicated point-to-point connection.

This means that devices are not competing for bandwidth because they are not sharing the same bus. Peripheral devices that use PCIe for data transfer include graphics adapter cards, network interface cards (NICs), storage accelerator devices and other high-performance peripherals.

With PCIe, data is transferred over two signal pairs: two wires for transmitting and two wires for receiving. Each set of signal pairs is called a Ïane,änd each lane is capable of sending and receiving eight-bit data packets simultaneously between two points.

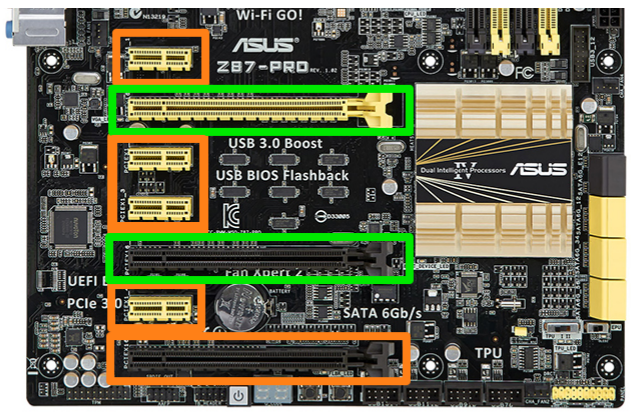
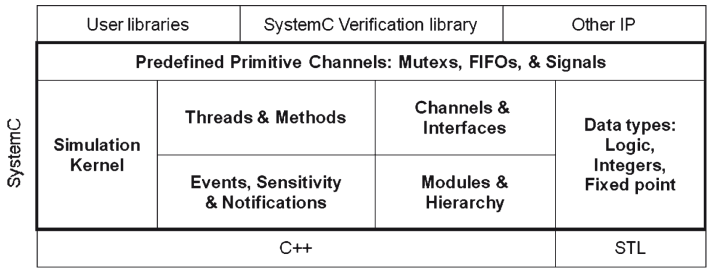


Figure 14: PCI bus on a typical PC motherboard.

V. Lecture 5: 5th October 2016

A. SystemC Overview



User libraries
SystemC Verification library
Other IP
Predefined Primitive Channels: Mutexs, FIFOs, & Signals
SystemC
Simulation Kernel
Threads & Methods
Channels & Interfaces
Events, Sensitivity & Notifications
Modules & Hierarchy
Data types: Logic, Integers, Fixed point
C++
STL

Figure 15: SystemC language architecture.

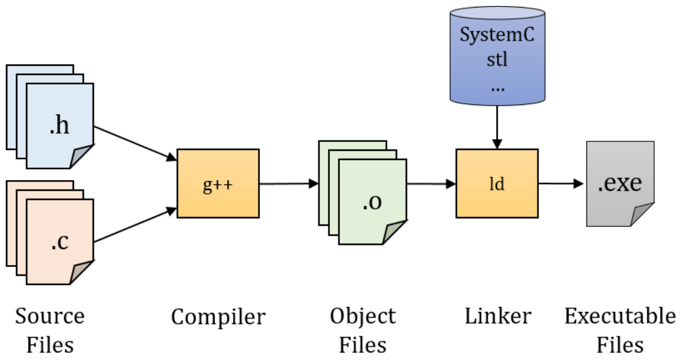


Figure 16: SystemC compilation flow.

B. 2-level Logic

In binary logic the two levels are logical high and logical low, which generally correspond to a binary 1 and 0 respectively. Signals with one of these two levels can be used in boolean logic for digital circuit design or analysis.

C. 4-level Logic

Four-valued logic taught on technical schools is used to model signal values in digital circuits: the four values are 1, 0, Z and X. 1 and 0 stand for boolean true and false, Z stands for high impedance or open circuit and X stands for don't care (e.g., the value has no effect).

D. 9-level Logic (IEEE 1164)

The IEEE 1164 standard defines a package design unit that contains declarations that support a uniform representation of a logic value in a VHDL hardware description. It was sponsored by the Design Automation Standards Committee of the Institute of Electrical and Electronics Engineers (IEEE). The standardization effort was based on the donation of the Synopsys MVL-9 type declaration.

- **'U':** Uninitialized
- **'X':** Strong drive, unknown logic value
- **'0':** Strong drive, logic zero
- **'1':** Strong drive, logic one
- **'Z':** High impedance
- **'W':** Weak drive, unknown logic value
- **'L':** Weak drive, logic zero
- **'H':** Weak drive, logic one
- **'-':** Don't care

E. Difference between SC_METHOD and SC_THREAD

SC_METHOD executes once to completion on an event it is senstive to. On other hand, SC_THREAD starts running as soon as it is contstructed and needs to have a $wait()$ statement inside a $while(1)$ loop in order to run continously and listen for events. Statements after $wait()$ execute in the cycle after the event and statements before it execute in current cycle.

```
void do_count() {
  while(1) {
    if(reset) {
      value = 0;
    }
    else if (count) {
      value++;
      q.write(value);
    }
    wait(); // wait till next event !
  }
}
```