CPRE/COMS 583x Adaptive Computing Systems	Fall Semester 1998
Lectures 1,2,3: August, 25,27, 1998 and September, 03, 1998	
Instructor: Arun K. Somani & Akhilesh Tyagi	Scribe: Sashisekaran Thiagarajan

1 Adaptive Computing Systems - An Introduction

Adaptive Computing Systems are those systems that reconfigure their hardware/software configuration to improve the performance of the applications that are currently being run on them. A common example of adaptive computing is use of personal computers(PCs) fitted with microprocessors in businesses, industries and homes today. The microprocessor in the PC is a good representative for the *General Purpose Computing* model. Other examples of adaptive computing are Digital Signal Processors, a Universal Remote Control unit, Field Programmable Gate Arrays(FPGAs) and Enhanced Programmable Logic Devices(EPLDs).

1.1 General Purpose Computing

General purpose computing defers the binding of functionality to the device during manufacturing until the device is employed to do the given task. This helps in efficiently mass-producing a device with fixed hardware and customizing it later to one's needs. The device can be so designed that the *reconfiguration* of this device can occur at different levels before its employment.

- The customization can occur at the *market* level, depending on the market to which the device is targeted to be sold.
- The *system* level, running different pieces of code at different times.
- The *application* level, where the same set of central resources are used under different instructions rather than building various functional units.
- At the *algorithm* level, where the different algorithms can be used depending on the application.

Some of the main components that make up the general purpose processor are *building blocks* such as registers, ALU, cache memory and clock, the *interconnects* which provide datapaths and help the blocks communicate with each other, and the instruction set of the computer which help in controlling the operation of the blocks. Of these components, the instruction set is the main unit in the general purpose processor which provides the adaptability in computing. Using the various instructions, we can make the processor implement different functions at different times and also allow sharing and reuse of the various hardware blocks. A device such as the general purpose purpose processor which depends solely on the instruction set for adaptive computing and heavily reuses a single piece of active circuitry is referred to a *programmable* device. The general purpose processor is often referred to as a *stored program* processor to reflect the stored program model that the processor uses

for computation. Some of the benefits of the stored program processor over a fixed function processor are: (a) hardware permanency of the processor, (b) the capability to develop applications after fabrication of the device which improves the time to market and also distribution of the large fixed design cost due to economies of scale.

1.2 Configurable Devices - FPGAs

An FPGA, like a general purpose processor and shown in Figure.1, is also composed of processing units, configurable interconnect and instruction memory. However, while the hardware functionality of the general purpose processor is fixed, the hardware functionality of the FPGA can be varied. Such a device where the active circuitry can be *configured* to perform different functions at different times is referred to as a *configurable* or *field-programmable* device. An FPGA is a field-programmable device with a general structure of very high logic capacity and offers a higher ratio of flip-flops to logic resources compared to EPLDs.

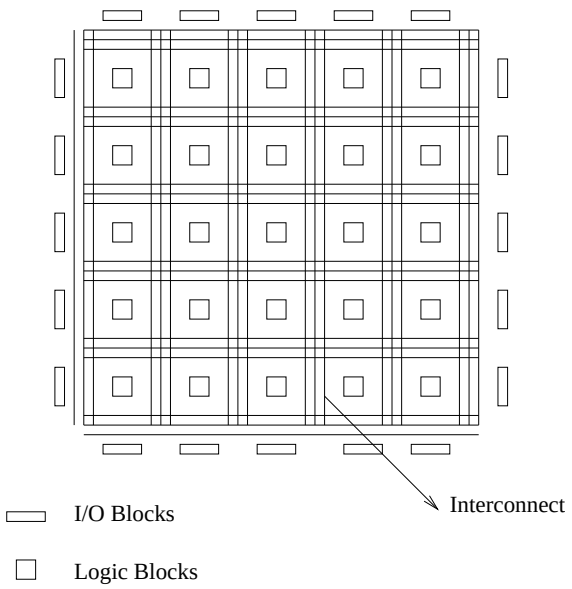

Figure 1: Structure of an FPGA

In general, FPGA will be quite efficient in computing tasks which are *regular*. A regular task is one that is characterized by spatial and/or temporal repetition while an irregular task is more data dependent. The basic processing unit for the FPGA is the *Look up Table(LUT)*. A LUT, as shown in Figure.2 is a small one bit wide memory array, where the address lines for the memory are inputs of the logic block and the one bit output from the memory is the LUT output. A LUT with k inputs would then correspond to a $2^k \times 1$ bit memory , and can realize any logic function of its k inputs by programming the logic function's truth table directly

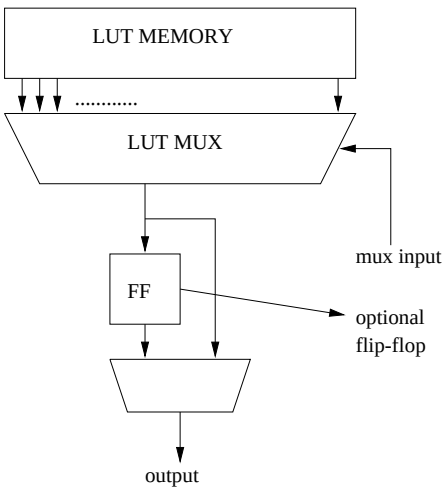

Figure 2: Logic Diagram of a LUT

into the memory. Sometimes the LUT also contains a flip-flop so that both combinational and sequential logic can be implemented. As an example, suppose we need to be able to implement any function of two variables, we use a 2-input LUT and set its contents(four bits) corresponding to the column (0 through 15) as shown in Table.1 that represents the function we want. Each of the columns (0 through 15) represent a distinct function output for the input variables A and B. For eg. column 8 represents output corresponding to the AND function. After setting the contents of the LUT, inputs A and B to the LUT will give us the desired function output.

Table 1: The table of values in columns 0 to 15 that need to be stored in a 2-input LUT for implementing the hardware to represent any function of two variables A and B

A	B	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

1.3 Functional Density

In this subsection, we will briefly discuss through an example of how hardware design can be optimized in different ways with respect to area and time. Suppose that the following function needs to be implemented in hardware:

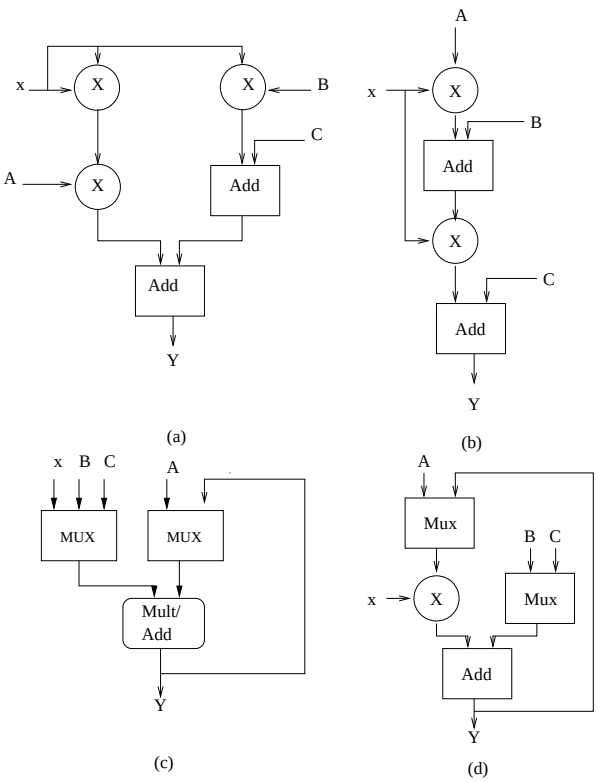

Figure 3: Different Hardware Implementation of the Function $Y = Ax^2 + Bx + C$. (a) requires 3 multiply units and 2 add units, and requires 2 Mult and 1 Add time units, (b) requires 2 multiply and 2 add units and requires 2 Mult and 2 Add time units, (c) requires a compound multiply/add units and a 5-to-1 mux, and requires 2 Mult and 2 Add time units, and (d) requires a multiply unit, an add unit and 2 2-to-1 mux, and requires 2 Mult and 2 Add time units.

$$Y = Ax^2 + Bx + C \tag{1}$$

Obviously, that there exists more than one way to implement this function. Figure.3 shows four such implementations, the number of hardware units they require and their execution time. Depending upon the hardware limitations and execution time requirements of the application where the function Y needs to be implemented, we should be able to make our choice of the design and optimize the performance of the application.

In general, given a set of solutions for a problem, we would like to assess their cost and compare them. However, the task of assigning a single metric to compare the solutions is a hard task. This is because the balance among the computational elements, the interconnect, the memory and the instructions can vary a lot among the solutions. In addition, each of these solutions can be manufactured using different process technologies. Our attempt now will be to develop a common metric to compare computing solutions across a spectrum of computing devices ranging from processors to full-custom designs with ASICs and FPGAs in

between. Since each solution requires different hardware and takes different times, both area and time have to be considered when doing the comparison. We will now look at the peak computation that each of the solutions can deliver to us on a cycle and normalize it to the implementation area and the cycle time. In other words we determine the computation that the device can produce per unit area-time. The unit of time can be taken as seconds (rather than in clock cycles). To take care of the problem of designs manufactured in different process technologies, we evaluate area in terms of the minimum feature size that each technology will allow. This minimum feature size, denoted as λ, is taken as the basic unit for length and width of the blocks in our hardware solutions. The minimum area unit then can be defined as λ^2. Using these units, we now define the term **Functional Density** as the number of functions that can be done per unit area per unit time i.e.,

$$Functional\ Density = \frac{no.\ of\ gate\ evaluations\ needed\ (N_{ge})}{t_{cycle} \times Area} \tag{2}$$

In the above formula, we define N_{ge}, where a "gate evaluation" is referred to as the basic unit of computation. In a processor, this basic unit of computation, referred to as a *bit-op*, is the amount of computation that one bit of the ALU performs in one cycle. With respect to an FPGA, we define one bit-op as equivalent to two 4-LUT operations. This is because add/sub operations can be implemented using two 4-LUTs (one to compute the sum output and the other for the carry). There are some potential drawbacks to this analogy. In the case of logical operations, where one 4-LUT can often perform more than one ALU bit operation, the computing capability of the FPGA has been underestimated. Also, many ALU operations such as shift, move, etc. become interconnect operations in the FPGA and can be easily implemented in the configurable interconnect on the FPGA. Nevertheless, we go ahead and give the computational density as:

$$Processor\ Computational\ Density\ = \frac{N_{ALU} \times w}{t_{cycle} \times A} \tag{3}$$

$$FPGA\ Computational\ Density\ = \frac{N_{4LUTs}}{2 \times t_{cycle} \times A} \tag{4}$$

Where A is the die area, w is the width of the processor ALU, and N_{ALU} is the number of concurrently executing ALUs in the processor. For the processor, t_{cycle} is simply the clock cycle time between instructions. For the FPGA, this is the minimum cycle time for a logic block, flip-flop and minimal interconnect.
We can also calculate the time required to complete the task , T_{task}, as

$$T_{task} = \frac{Number\ of\ Gate\ evaluations\ required\ (N_{ge})}{capacity\ in\ number\ of\ gate\ evaluations\ per\ second\ (D_{cap})} \tag{5}$$

However, the actual yielded capacity, $D_{yieldcap}$ may be less than the intended production capacity which means that the actual time taken, T_{actual}, will be less than T_{task}. In RISC processors, $D_{yieldcap} << D_{cap}$ usually. In reconfigurable computing, we are interested in improving $D_{yieldcap}$ so that T_{actual} decreases and gets closer to T_{task}.

General purpose processors perform badly with the functional density metric defined above. This is due to a variety of reasons. Though the processor has a lot of functional units, their density is badly eroded by the fact that a limited number of tasks are running at any particular time and most of the processor except for the ALU(and some other parts such as the instruction counter) remain idle for most of the time. Another density erosion factor is the cache memory overhead. Most of the time only a small part of the memory is being accessed per cycle compared to the huge area occupied by it. Hence it has been shown that FPGA implementations have an order of magnitude advantage over general purpose processors in terms of peak computational density. This advantage also has been shown to hold over a wide range of process technologies. Hence to calculate the expected functional density(not statistical expected value) for the general purpose processor, we can use the following representation:

$$Expected\ Functional\ Density = \frac{Gate\ Evaluations}{Datapath\ Bit} \times \frac{Datapath\ Bits}{instruction} \times \frac{instructions}{Issue\ Slot} \times \frac{Issue\ Slots}{ClockCycle} \times \frac{1}{area \times t_{cycle}}$$

In the above equation, the datapath refers to the width of ALU of the processor. The issue slots refer to the number of execution units. For example, a superscalar processor could have many instructions being issued out to multiple execution units capable of operating simultaneously. The "area" term in the formula is the silicon die area of the chip and t_{cycle} refers to the clock cycle time.

1.4 References

1. Andre DeHon, "Why Configurable Computing? The Computational Density Advantage of Configurable Architectures", 1998.
2. Andre DeHon, "Comparing Computing Machines", January 1997.