Cpr E 583 Adaptive Computing Systems	Fall Semester 1998
Lecture 6: Thursday September 10	
Instructor: Arun Somani, Akhilesh Tyagi	Scribe: Deepali Deshpande

Multicontext FPGAs

1 Introduction

A configuration of the FPGA is a bit stream that controls the function of the device. Traditional FPGAs store single configuration on-chip. That is, every LUT has a single set of bits describing its function as shown in Figure 1(a). Between the changes in configuration, LUT performs exactly the same function all the time. Since in the design, for example pipelined architecture, it is seldom necessary to evaluate every gate or bit computation simultaneously, most of the FPGA resources or LUTs sit idle. The application can reuse the LUTs that are sitting idle, if they can be reconfigured. The reconfiguration involves bringing new configuration in the look-up table memory from the off-chip memory. This takes long time (in milliseconds). However, it is noticed that the configuration memory of the LUT is much smaller than the active resources required to perform the logic and switching functions on the device. These observations suggest that, it is possible to provide multiple configurations or multiple contexts for a single LUT without causing substantial increase in the area. Once multiple configurations are present on the chip, the application can reuse LUTs by selecting a proper context for it which results in reduced hardware cost.

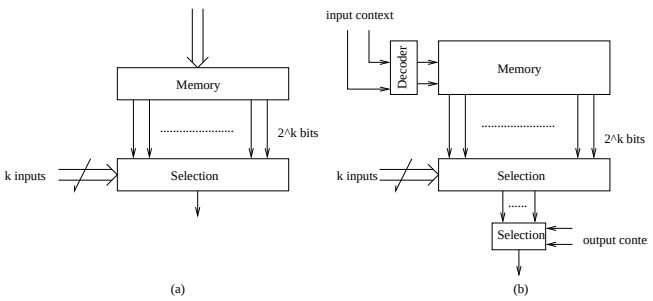

Figure 1: LUT: (a) Traditional and (b) Multiple Context

A multiple context LUT is shown in Figure 1(b). It uses large configuration memory which stores multiple contexts. Input context bits select the row in the memory. The memory context also called input context is set depending on the application or the specific operation to be performed. The first level of selection is done based on one set of input bits or one operand of the operation. The result of the operation is selected by the second operand. This is called output context.

In the last class, we used output context in designing Adder and Multiplier units where the second operand selected the part of the memory row, but the adder and multiplier once configured did the same function all the time. It means we had only one input context. In this class, our goal

is to study latency, throughput and area tradeoffs between single and multicontext FPGAs. In the following sections we analyze three implementations for ASCII to binary conversion, 1)using single context LUTs, 2)Using full pipelining with single context LUTs, and 3)Using temporal pipelining with multicontext LUTs, to find area, latency and throughput tradeoffs.

2 Example: ASCII Hex to Binary Conversion

The basic computation involved in ASCII Hex digit into binary is as follows.

```
if (c >= 0x30 && c <= 0x39)
    res = c - 0x30;
else if (c >= 0x40 && c <=0x46)
    res = c - 0x40 + 10;
else if (c >= 0x60 && c <= 0x66)
    res = c -0x60 + 10;
else
    res = 0;
```

The mapping of this application on the FPGA using 4-LUTs is shown in Figure 2. Each blocks in the figure represents a 4-LUT. The mapping equations used for this implementation are as follows.

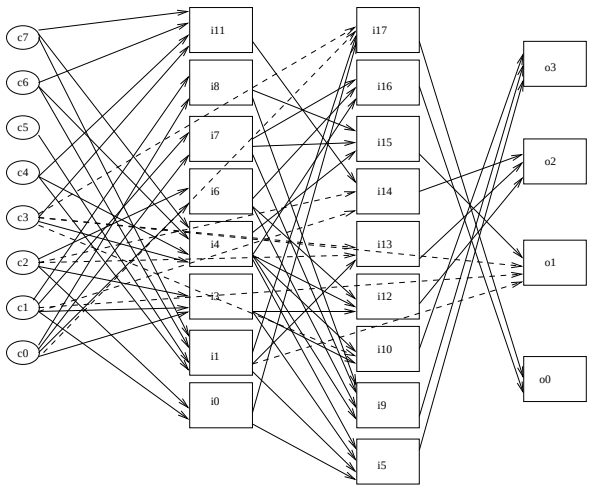

Figure 2: 4-LUT Mapping of ASCII Hex→Binary

```
INORDER = C[7] C[6] C[5] C[4] C[3] C[2] C[1] C[0];
OUTORDER = O[3] O[2] O[1] O[0];
# stage 1-8 LUTs [C[3:0] pass through]
i0 = !C[1] * !C[2] ;
```

```
i1 = C[4] * C[5] * !C[6] * !C[7];
i3 = C[0] * C[1] * !C[2];
i4 = !C[3] * !C[4] * C[6] * !C[7];
i6 = !C[0] * C[2];
i7 = !C[0] * C[1];
i8 = C[0] * !C[1];
i11 = !C[7] * C[6] * !C[4] * !C[3];
# satge 2 - 9 LUTs [i1, C[3], C[1] pass through]
i5 = i0*i1 + i3*i4;
i9 = i6 * i4 + i7*i4 + i8*i4;
i10 = C[3] + i3*i4;
i12 = i3*i4 + i6*i4;
i13 = i1 * !C[3] * C[2];
i14 = C[2] * !C[1] * i11;
i15 = i8*i4 + i7*i4;
i16 = i7*i4 + i6*i4;
i17 = i1 * !C[3] * C[0] + C[0] * i0 * i1;
#stage 3 - 4 LUTs
O[3] = (i10+19) * (i5+i9);
O[2] = i12 + i13 + i14;
O[1] = i1 + !C[3] * C[1] + i15;
O[0] = i16 + i17;
```

The single context implementation shown in Figure 2 uses total 21 4-LUTs. The area of each LUT is $880K\lambda^2$. Hence total area is,

$$A_s = 21 * 880K\lambda^2 = 18.48M\lambda^2$$

With each LUT delay of t_g, the latency of the application is $3t_g$ as there are 3 levels of LUTs from the input to the output. Hence the throughput in this case is given by,

$$T_s = \frac{1}{3t_g}$$

We can improve on the throughput by using traditional or full pipelining. This pipelined implementation produces result on every cycle after initial latency of 3 cycles. Note that for our example, as seen in Figure 2, inputs C<3:0> are required in the second stage and, inputs C<3>, C<1> and i1 from the second stage are required in the third or the output stage. With pipelining inputs and output of each LUT change on every clock cycle. Hence the signal values that are required after 2 or more cycles are required to be latched. This is called retiming of the values. Since flip-flops are only available on LUT outputs, in addition to active 4-LUTs shown in Figure 2, we add 4 LUTs in the second stage and 3 LUTs in the third stage to achieve retiming as shown in Figure 3.

The full pipeline implementation uses 28 4-LUTs. Hence total area is,

$$A_p = 28 * 880K\lambda^2 = 24.64M\lambda^2$$

With one LUT delay of t_g, the throughput of full pipeline implementation is given by ,

$$T_p = \frac{1}{t_g}$$

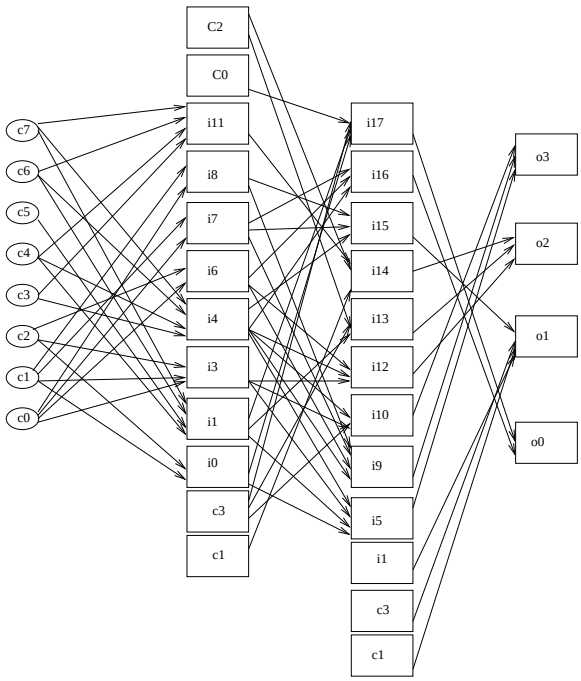

Figure 3: ASCII Hex→Binary Circuit Retimed for Full Pipelining

The pipelined implementation increased the are by 33% but reduced the latency by the same amount. Suppose we do not care for achieving high throughput but only about the latency then we can execute the pipeline, stage by stage or temporally on a multicontext device with 3 contexts for each LUT as shown in Figure 4. The figure shows that multiplexors are required to select the LUT inputs depending on the context. It is possible to eliminate multiplexors in a rerouting network. To achieve the target latency of 3 LUTs, we need to have enough active LUTs to implement the largest stage – the middle one. Consequently we need 3 extra LUTs in the largest stage, making for a total of 12 LUTs. Note that the four retiming LUTs added to the first stage also bring its total LUT usage up to 12 LUTs. If C<3:0> were inputs that did not change during these three cycles, we would only need 10 LUTs.

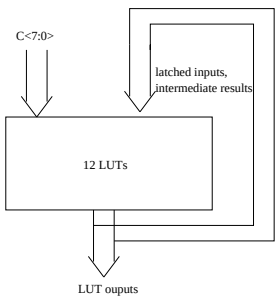

Figure 4: Temporal Pipelining for ASCII Hex→Binary

Due to more context memory, the multicontext LUT is slightly larger than the FPGA LUT (single context). Normally 4-16 contexts are sufficient for different applications. We used 3 contexts for this application. Typically $50 - 80K\lambda^2$ area is required by one context. Our application needs additional 2 contexts, thus, an additional $160K\lambda^2$ area. Hence total area of the multicontext implementation is given by,

$$A_m = 12 * 1040K\lambda^2 = 12.48M\lambda^2$$

If we assume that we can pipeline the configuration read, the multicontext device can achieve comparable delay per LUT evaluation to the single context device. Then the total latency is still $3t_g$. If we do not pipeline the configuration read, the configuration read may add slightly more delay. If the delay per multicontext LUT is t'_g $(< 2t_g)$ then the latency is $3t'_g$. The throughput in this case is given by,

$$T_m = \frac{1}{3t'_g}$$

3 Factors Affecting Throughput

The temporal pipeline implementation studied in the previous section may not provide for perfect packing. Some of the pipeline stages may require more LUTs than other stages. This causes some of the LUTs to sit idle when the pipeline satges requiring less number of LUTs are executed. Again as mentioned before the retiming is extremely important for such implementation. Normally in pipelined application throughput is determined by the slowest pipeline stage. For example, the stage $X2$ shown in Figure 5 is the slowest in the pipeline and hence governs the maximum throughput. Since $X2$ and $X3$ are functioning simultaneously in Figure 5(a), the total area of the second stage is that required by $X2$ and $X3$ units. Note that in this case we can save on area by combining $X1$ and $X3$ in a single pipeline stage and time multiplexing their execution as shown in Figure 5(b). It was possible to combine $X1$ and $X3$ because the latency of the combined operation is less than that of $X2$. Thus, we saved on hardware as well as filled up the available time slot without affecting the final throughput.

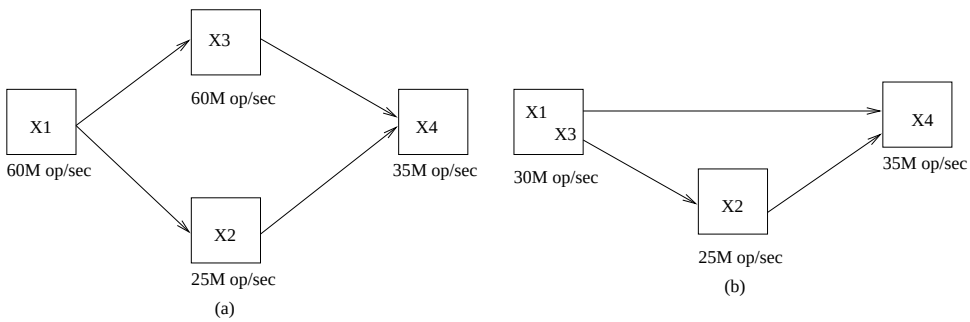

Figure 5: Pipelining Reorganization: (a)Original Pipeline, (b)After Reorganization

The throughput is determined by various factors as listed below.

1. Limited throughput requirement set by the hardware. There are three different hardware factors that affect the throughput,
   (a) Relative processing speeds of different pipeline stages. The throughput adapts with respect to the slowest stage.
   (b) Fixed functional requirements. The functional requirement of the application is fixed. There is no point in doing anything more.
   (c) IO latency and Bandwidth– off-chip speed.
2. Latency limited designs: For such designs input itself is arriving at a slow rate, and therefore, there is no point in trying to increase the throughput more than the input rate.
3. Cyclic dependencies- It is an application dependent factor determining the throughput.

In this class we studied the area-time tradeoffs between different implementations and various throughput related implementation issues. In the next class, we will study how area and time scale with the improvement in the VLSI technology.

Reference

Andre DeHon, "Multicontext Field-Programmable Gate Arrays,"
http://www.cs.berkeley.edu/ amd/CS294S97/reading/day25.html