EEC 180B: Digital Systems II

Spring Quarter 2015

Department of Electrical & Computer Engineering

University of California, Davis

Lab 6: Design and Optimization of a Configurable High Speed Integer Multiplier

---

Objective

Integer multiplication is an important operation in many applications such as computing Fast Fourier Transform, Digital Filtering, and Neural Networks etc. The straightforward shift-add algorithm is too slow, as it would take 64 cycles to perform a 64x64 multiplication. In a modern 64-bit processor, for example the operation must complete in one clock cycle, which is around 300 picoseconds. In battery-operated devices power consumption is also an important concern. However, 64-bit multiplication is not necessary all the time, especially in many embedded systems, so power consumption and performance can be improved by operating at smaller bitwidths such as 32, or 48 bits or treating numbers as unsigned. With Field programmable gate arrays we can create a multiplier that can be *customized* for a given application. So in this laboratory exercise we will explore the design space of architectures for efficient integer multiplication.

Though we will be using multiplication as an example, the methods and techniques learnt in this exercise such as pipelining, minimizing wiring complexity, fanin, and fanout, using tree structures to reduce complexity can be can be applied to a broad class of hardware implementation of VLSI systems for FPGAs and ASICS.

Background

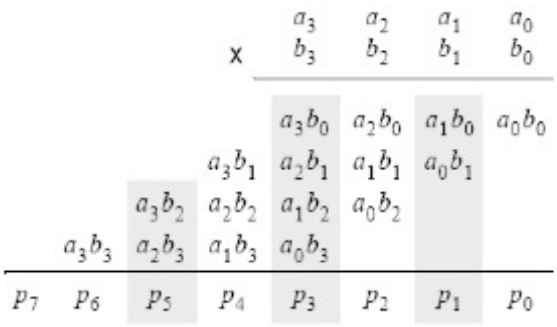

Figure 1 - Multiplication Algorithm

Figure 1 shows the basic algorithm to multiply two unsigned 4-bit numbers and Figure 2 shows the direct hardware implementation of the algorithm. The latency of the multiplier (the time from when the inputs are applied to the time when the outputs are valid) is the path with the longest delay from the input to the output, which for a **n x m** multiplier is n+m-2 full-adders. If n=m=64, the delay will be of the order of 64 full adders plus the interconnect delay which would be much greater than the 300 picoseconds that is our target. So, we need a better solution.

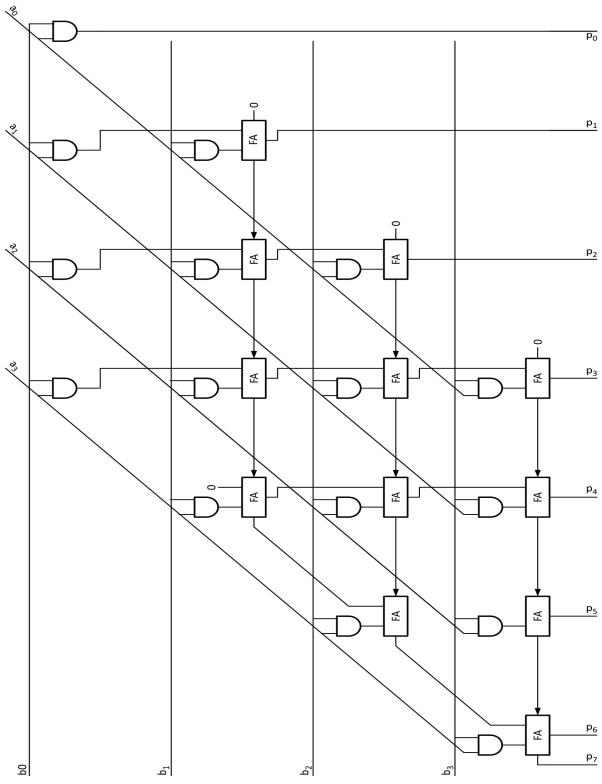

Figure 2 - Hardware Implementation of the 4-bit Multiplier

Improving the Performance of Integer Multiplication

In order to improve the performance of the multiplier dramatically we need to create a better architecture than just mimicking the paper and pencil calculation shown in Figure 1. If you examine the multiplication algorithm in Figure 1 closely, you will see that there are 2 major steps –

a) Partial Product Generation

b) Summation of Partial Products

We can improve the performance of the integer multiplier by reducing the number of partial products and by reducing the complexity of the summation of partial products from O(n) to O(log(n))

Architectural Enhancement 1: Recoding

Recoding the operands can reduce the number of partial products. For example instead of treating each bit of the m-bit multiplier individually, we can group k adjacent bits into a new digit of Radix 2^k which means there are only m/k digits in the multiplier, so there will be m/k partial products instead of m. Specifically, by using a special type of recoding called Booth Recoding we can also handle signed numbers in the two's complement representation.

The key architectural level optimization question to a digital designer is what is the best value of k to reduce the overall complexity (including the wiring) and power consumption while maximizing power consumption.

Read Section 12.2 of your textbook.

Architectural Enhancement 2: Compression Trees

Next, instead of adding the partial products one at a time which gives rise to the linear complexity of the partial product summation, we could reduce the complexity to a logarithmic function of the number of input bits by creating a tree of adders to sum the partial product. There are two ways of doing it. One is to use Wallace Trees also known as 3:2 counters that reduce 3 partial products of the same weight to 2 or one could use 4:2 compressors which combine four partial products at a time.

The key design question for a architect is what is the best building block for creating the partial product summation tree. **Read Section 12.3 of your textbook.**

Architectural Enhancement 3: Carry Look Ahead Adders

A final stage carry propagate adder is required in any implementation of a high-speed multiplier. A simple ripple carry adder would not be appropriate because it would be too slow. So, a carry lookahead adder is needed. The key idea behind a carry lookahead adder is that whether carry is generated or not during the addition of two numbers can be pre-computed from the inputs. In the limit this can be done with a 2 level AND/OR circuit but the fan-in of the gates will be prohibitive. A large adder can be built efficiently by breaking the adder into a set of blocks of reasonable size (say 8 or 16), with carry look-ahead used inside the blocks and ripple-carry between the blocks. Such an adder is called a block carry lookahead adder.

The key design question here is what should be the optimal block size – should it be 16 or should be 8? How about a mixture of both? **Read Section 12.1 of your textbook.**

Architectural Enhancement 4: Pipelining

So far we have discussed architectures that perform one calculation at a time, which means, you present the inputs A and B, wait for the output to be produced and then apply the next set of inputs. Performance of the adder can be improved significantly if we **overlap** the computations i.e. if we start processing the next set of inputs before the current multiplication is complete. This is called pipelining and is one of the most powerful techniques available to a digital designer to improve the throughput of a design. Pipelining requires intermediate registers to keep the **state** of the current computation and the next computation separate. For example, in Figure 2, one could add D flip-flops (registers) along the line labeled b_2, which would allow the design two operate as a 2-stage pipeline. Pipelining improves throughput but increases the area (because additional registers are needed) and also increases dynamic power consumption because the flip-flops have to be clocked continuously. So, the number of stages in a pipeline is a key design question that has to be considered carefully.

Detailed Specification of the Project

The choice of radix for recoding, the fan-in of the compressor tree, the block size of the carry lookahead adder depends on the implementation target – whether it is an ASIC or FPGA and if it is an FPGA what is the its capacity in terms of look up tables, registers, interconnection resources, and memory. It

depends on the gates available in the target library. Methods developed in Chapter 4 and Chapter 5 to estimate and optimize delay using fan-out and logical effort can be used to arrive at the best architecture for a given implementation target.

You will explore the design space of integer multiplication on the Altera CycloneII FPGA.

The goal is to come up with the optimal radix for the Booth recoding, optical block size for carry lookahead adder, partial product compression tree, and pipelining that maximizes throughput and minimizes the area and power consumption for a multiplier that can operate on 32, 48, and 64 bit inputs.

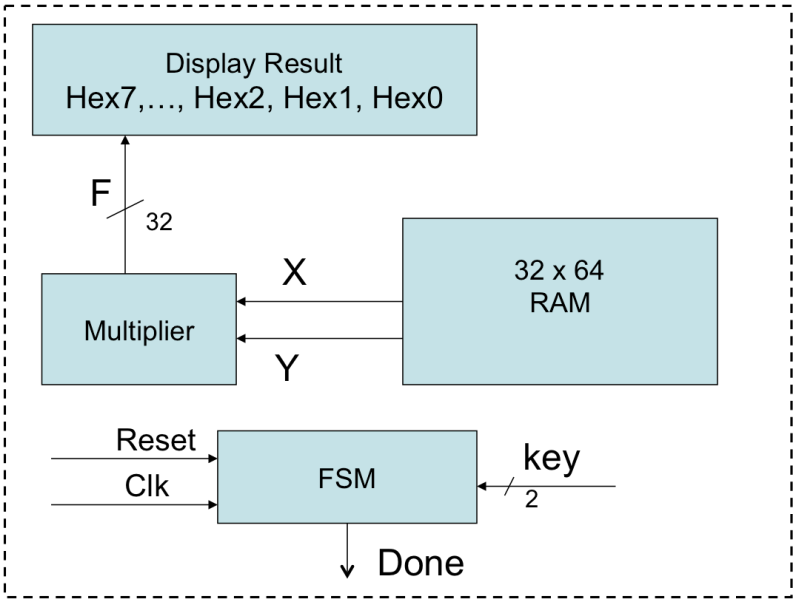

Figure 3 - System Level Architecture of the Design

1. The top-level architecture of your design should be as shown in Figure 3. The RAM has 32 words 64-bit words that serve as inputs
2. Your design should be **programmable**, which means, it receives an additional input (denoted by key) that specifies the width of the input operands and hence the output result.

KEY[1:0]	**FUNCTION**
00	Unsigned 32 bit
01	Signed 32 bit
10	Signed 48 bit
11	Signed 64 bit

The operands will be the two successive words starting from address 0. So, for the first multiplication, the inputs are RAM[0] and RAM[1], while for second operation the inputs are RAM[2] and RAM[3], and so on. So, we have 16 sets of input preloaded into the memory. When operating on bitwidths less than 64, you should just take them from the least significant bit position and ignore the rest. For example, when doing 32 bit arithmetic, you will only consider the least significant 32 bits.

3. Output should be displayed as hexadecimal number on Hex7, Hex6, … Hex2, Hex1 and Hex0, 32 bits at a time starting from the least significant 32 bits. Each result should be displayed for at least 10 seconds.
4. Assume there is a reset switch, which can also serve as the start signal, when de-asserted.
5. The computation can take more than one cycle. So, a Done signal is necessary to announce the completion of the operation and validity of the results.
6. You CANNOT use pre-defined * operator or any IP blocks from Altera. Your top-level design has to be structural design along the lines of Figure 12.12 and should use Booth Recoding, Carry Lookahead Adders, and Wallace Trees. **Otherwise you will not receive any credit for this laboratory exercise.**
7. A fraction of the final grade for the project will depend on how optimized your design is, both in terms of number of resources (Logic elements) and the clock frequency.

PRELAB

1. Read Chapter 12 of the book, especially the 16-bit carry lookahead adder in Figure 12.6, 6x4 2's complement multiplier using radix-4 Booth recoding in Figure 12.9, and Wallace Tree based 4x4 multiplier in Figure 12.12
2. Design a Radix 8 and Radix-16 Booth recoder by writing out a tables like Table 12.1 and Table 21.2 and verify the operation on 8x8 multiply
3. Figure 12.15 shows a 4-2-compressor tree that takes four bits of input and produces two bits of output using two adders. Draw a table like that of Table 12.4 and 12.5 for a 16x16 radix-8 Booth recoded multiplier using these four-input cells. You may assume that pre-added partial products are available.

What should you do?

1. Start with Radix-8 booth encoding, 3:2 compressor tree, and 8-bit Carry

lookahead adder based design.

2. Write a testbench to test the multiplier thoroughly. The testbench should be along the lines of the testbench provided to you in Lab 5. In fact, you can use it with appropriate modifications.
3. Verify and optimize 32 bit signed and unsigned multliplier first and download and demonstrate to the TA. Tabulate the number of Logic Elements used, Number of Flip-flops used, Critical Path of your design, and Power consumption of the design.
4. Verify and optimize the 48 and 64 bit versions and download and demonstrate to the TA. Tabulate the number of Logic Elements used, Number of Flip-flops used, Critical Path of your design, and Power consumption of the design
5. Reimplement the designs with Radix-16 Booth Encoding, 4:2 compressor tree, and 16-bit carry lookahead adder.
6. **Tabulate the number of Logic Elements used, Number of Flip-flops used, Critical Path of your design, Throughput in terms of multiplications per second, and Power consumption for each design. Include it in your final laboratory report.**

Execution of the Project

You will work in a group of 2 for this project. *Total Points is 300*		
Milestone	**Points**	**Deadline**
32 bit signed and unsigned	100 (50 for signed and 50 for unsigned)	Before May 22, 2015
48 and 64 bit signed	100 (50 for 48 bit and 50 for 64 bit)	Before May 30, 2015
Higher Radix designs 64 bit signed	100 (50 for Radix 16 Recoding, 50 for 4:2 compressor tree with 16-bit CLA)	Before June 5, 2015

Extra Credit: (50 to 100 points)

1. Pipeline the design to improve the clock frequency of the circuit.
2. Baugh-Wooley Algorithm for Signed Multiplication.
3. Improving Power Consumption of the Multiplier
4. Extra credit work has to be completed before June 5, 2015.