Project Summary for ECE 429 Introduction to VLSI Design

updated April 20, 2001[1]

Version 1.4.0

The goal of this project is to design a chip which implements a 8-bit ALU which executes AND, OR, NOT, ADD, SUBTRACT, and DIVIDE. All ALU functions execute on two's complement numbers and we will refer to this design as POWER-ALU01. We will be employing a complex algorithm for division called SRT division which is a form of non-restoring division developed by three researchers independently around 1960. This algorithm was the main culprit for the Pentium FDIV bug that was well-publicized by the media around 1995. Some interesting information about the Pentium bug is found at the following URL:

http://www.cs.earlham.edu/~dusko/cs63/fdiv.html.

This is a significant project which will require substantial design effort. A schedule later in this document specifies the milestones for your project. Because of the project complexity, you are encouraged to work in groups of two. However, groups of three or larger will not be allowed. If you decide to work in a group, you must indicate this in your initial proposal that you hand in on April 13. Although this project will involve the use of a standard-cell design, you are welcome to design any block you wish at the custom-level.

1 About SRT Division

One of the first division algorithms is restoring division. Restoring division works by taking two components, the dividend, X, and the divisor, D, and computing the quotient, Q, and its remainder, R. The main equation for division is recursive as follows:

$$X = Q * D + R \ni R < D \tag{1}$$

To obtain a fractional quotient, division can be performed by a sequence of subtractions and shifts. Each cycle or step, called i, the remainder is compared against the divisor. If the remainder is larger than the quotient bit it set to 1, otherwise it is set to 0. The following equation for each step is used:

$$r_i = 2 * r_{i-1} - q_i * D \tag{2}$$

The most difficult step in the division procedure is the comparison between the divisor and the remainder to determine the quotient bit. If this is done by subtracting D from r_i, then one has to be careful if the result is negative. If so, then we must restore the remainder to the previous value. This method is called

[1]Note that this document is updated periodically to provide you with the most reliable information to complete the project. Although previous versions of this document are basically correct, frequent revisions are made in order to help you save time and **not** to hinder your performance. Therefore, make sure you check back periodically for updates. I will add revision numbers to help you identify which version of this document you have.

restoring division. Non-restoring division is an alternative for sequential division by having specific logic for not correcting the quotient. This is achieved by allowing a correction factor within the algorithm. Consequently, the quotient bit is determined by the following rule which allows the quotient to be restored automatically:

$$q_i = \begin{cases} 1 & \text{if } 2 \cdot r_{i-1} \geq 0 \\ \overline{1} & \text{if } 2 \cdot r_{i-1} < 0 \end{cases} \tag{3}$$

The SRT division algorithm is one of most well known and often-used division algorithms today. It was named after Sweeney, Robertson, and Tocher, each of whom developed the idea independently from each other. The idea behind SRT is quite simple. Its main goal is to speed up non-restoring division. This is easily accomplished by allowing a 0 to be a quotient digit. This would change the rule for selecting the quotient digit to the following:

$$q_i = \begin{cases} 1 & \text{if } 2 \cdot r_{i-1} \geq D \\ 0 & \text{if } -D \leq 2 \cdot r_{i-1} < D \\ \overline{1} & \text{if } 2 \cdot r_{i-1} < -D \end{cases} \tag{4}$$

If we manipulate the data for selecting the quotient so that the new selection rule requires a comparison of 1/2, then the logic for comparison is greatly simplified:

$$q_i = \begin{cases} 1 & \text{if } 2 \cdot r_{i-1} \geq 1/2 \\ 0 & \text{if } -1/2 \leq 2 \cdot r_{i-1} < 1/2 \\ \overline{1} & \text{if } 2 \cdot r_{i-1} < -1/2 \end{cases} \tag{5}$$

Therefore, SRT division looks rather easy. However, there is a little bit of a snag. In order to have the ability to have a 0 as a quotient, makes the quotient more complicated. That is, there must be some way to handle the output for a quotient with -1 or +1. Fortunately, we are going to use a method that was introduced in the early 1980's. This method is called on-the-fly conversion which uses two registers to help in the conversion.

A great web page has been created that can help you with some examples with this algorithm. It is available at the following website:

http://www.ecs.umass.edu/ece/koren/arith/simulator/SRT2/

This might be a very useful site when you are creating your testbenches and verifying the correctness of your algorithm.

2 Design Specification

In this project, you will create three functional units which will be controlled by a finite state machine. Although Verilog is recommended for this project, VHDL can be used equally as well.

2.1 POWERALU01 Specification

The purpose of this unit will be two compute several functions which will be the following where Acc stands for the accumulator and A is the input operand. All operations should be on 8 bits that handles two's complement numbers although the division algorithm can be implemented for only unsigned numbers. Table 1 shows the instructions for the POWERALU01 and in Figure 1 the block diagram is shown.

Instruction	Action
Add A	Accumulator = Accumulator + A
Sub A	Accumulator = Accumulator - A
And A	Accumulator = Accumulator && A
Or A	Accumulator = Accumulator || A
Load A	Accumulator = A
Div A	Accumulator = Accumulator / A
Not A	Accumulator = !Accumulator

Table 1: Instructions.

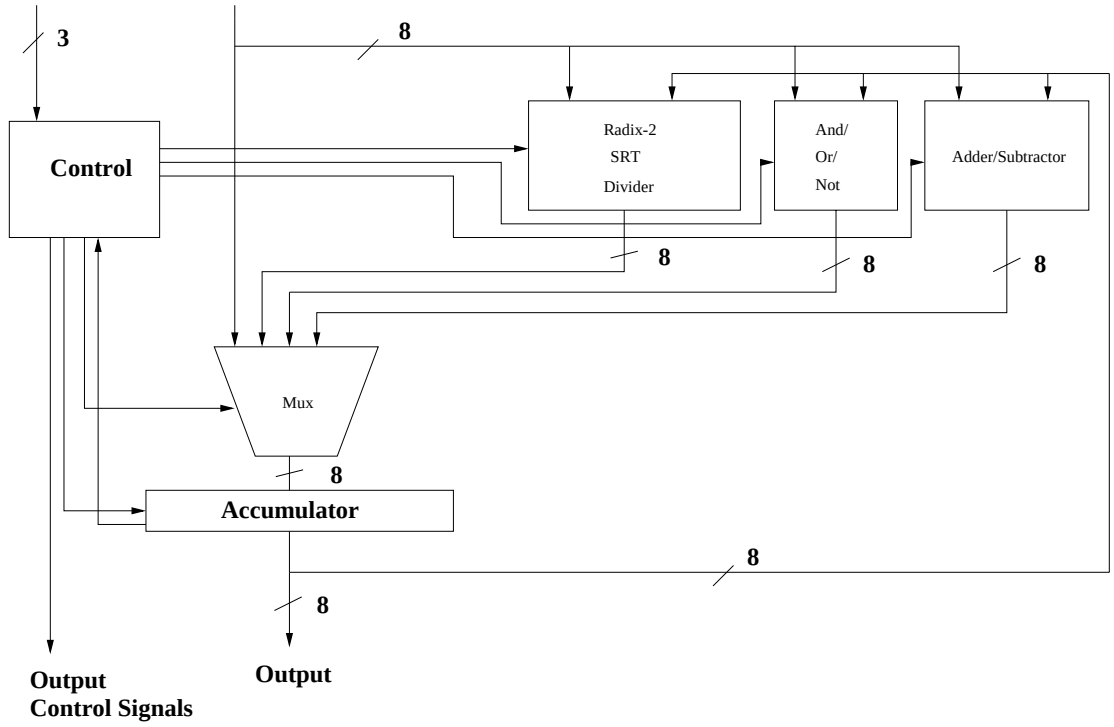

Figure 1: POWERALU01 Block Diagram.

The hardware uses a basic accumulator or register to store each result. The signals operate as follows: When an input operand has been read and the appropriate control signal has been passed to the control, the unit will perform a computation and output the result. A detailed description of the signals is as follows:

Signal	# of Pins	Type	Description
Scan In	1	Input	Takes test input for scan mode
Scan Mode	1	Input	Puts chip into scan mode
Reset	1	Input	A master reset that causes the chip to reset itself
Clock	1	Input	Clock rate
Input[7:0]	8	Input	Input operand
op_type[2:0]	3	Input	Encoded signal indicating operation to be performed
Vdd	1	Input	Power
GND	1	Input	Ground
Idle	1	Output	Signal indicating chip can receive data
Working	1	Output	A signal indicating the chip is working
Error	1	Output	A signal indicating an error
Overflow	1	Output	An overflow occurred
Output[7:0]	8	Output	Result
Scan Out	1	Output	Outputs scan output for scan mode

Table 2: Input/Output Signals.

3 SRT Divide Implementation

The radix-2 or binary SRT division algorithm is quite easy to implement. The main elements that are needed are an adder to add or subtract the partial remainder, multiplexors, registers, and some additional combinatorial devices. Figure 2 shows the basic block diagram for the design. Note that some wires involve either sign-extending or shifting the result which is indicated with the curly braces. Curly braces are used in Verilog for concatenation. Concatenation is very common in hardware and involves taking bits from two or more inputs and combining them to create a larger or smaller subset of the original input. For example, the input {A, B, C} where A, B, and C represent the four-bit quantities 0x4, 0x9, and 0xC, respectively, will form a new 12-bit value 0x49C. Curly braces can also be used to extend values. Suppose, that an input vector is going to sign extend a 13-bit input to a 16-bit quantity. Instead of having multiple values of the most-significant bit inside the curly brace, you can use specify the repeat value inside the curly braces. For example, suppose the 13-bit quantity and the 16-bit quantity are A and B, respectively. Then, instead of having the following Verilog statement

```
assign B = {A[12], A[12], A[12], A}
```

you can use the following shortcut:

```
assign B = {{3{A[12]}}, A}
```

Although, this shortcut uses a lot of curly braces, its extremely useful.

The quotient digit selection is fairly simple using the SRT division algorithm. Using the SRT algorithm, allows the quotient digit selection to be a comparison which is only one bit. However, in order to be safe, this SRT division algorithm will

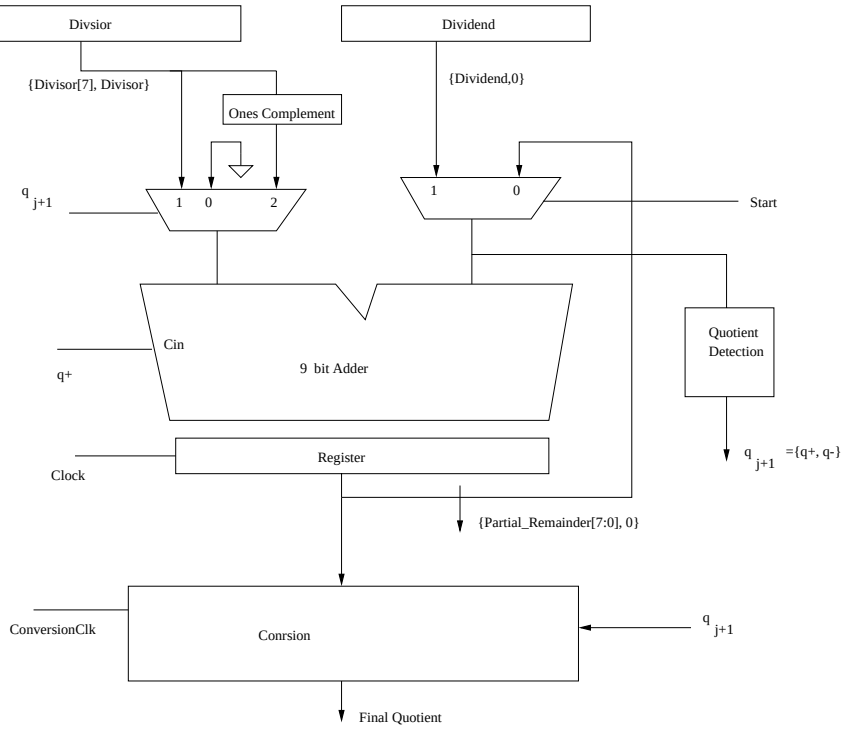

Figure 2: SRT Radix-2 Division.

compare three bits as indicated in Table 3. The three bits, *sign*, *int*, and $f0$ are shown in Figure 3 and are obtained

from the dividend in order to select the proper quotient.

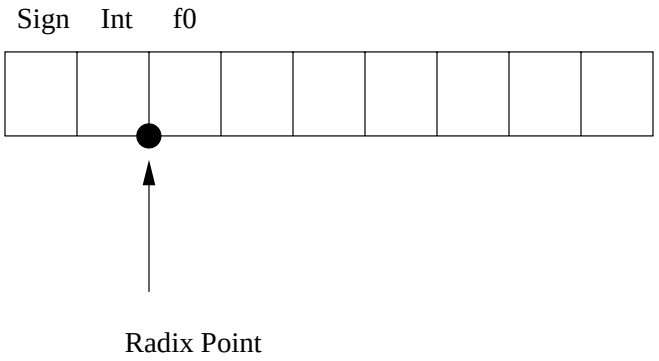

Figure 3: Subdivided Dividend bits.

3.1 On-the-Fly-Conversion

The use of the redundant quotient representation complicates the use of SRT division. The use of a redundant representation has been used extensively in many computers since their original introduction in the early part of the 20th century. For example, Booth encoding is typically used to encode a signal into a redundant notation to allow multiplication of signed quantities with fewer iterations to be possible.

sign	int	f0	Result	Quotient	q_+	q_-
0	0	0	$< 1/2$	0	0	0
0	0	1	$\geq 1/2$	1	1	0
0	1	0	$\geq 1/2$	1	1	0
0	1	1	$\geq 1/2$	1	1	0
1	0	0	$< -1/2$	-1	0	1
1	0	1	$< -1/2$	-1	0	1
1	1	0	$< -1/2$	-1	0	1
1	1	1	$\geq -1/2$	0	0	0

Table 3: Quotient Digit Selection.

In most fixed radix systems that most digital devices employ, the digit set set is restricted to $0, \ldots, r-1$. However, if we allow the following digit set:

$$x_i \in \overline{(r-1)}, \overline{(r-2)}, \ldots, \overline{1}, 0, 1, \ldots, (r-2), (r-1) \tag{6}$$

where $\overline{i}$ equals $-i$ and not $(r-1)-i$ as before. The resulting number system is called the *signed-digit* (SD) number system.

The use of the SD number system allows a numbering system to have some amount of redundancy. This means that a redundant number system has several representations of the same number. For example, both 0111 and $100\overline{1}$ in binary or radix-2 are the same number of 7.

One of the benefits of using the SD number system is that is allows carry-free addition or subtraction. This numbering system is used in the SRT divider. Although the SD numbering system is useful, it unfortunately is somewhat cumbersome to convert from SD notation back to normal binary representation. To convert from SD notation to conventional binary representation involves subtracting the digits with negative weights from the digits with positive weights with a carry-propagate adder. For example, $3\overline{2}1\overline{6}\overline{4}$ is

$$\begin{array}{r} 30060 \\ -02104 \\ \hline 27956 \end{array}$$

Division is an on-line algorithm. In other words, the most significant bit of the result is calculated first. Fortunately, there is an easy algorithm to convert back to conventional representation from SD notation for on-line algorithms. It is called on-the-fly conversion. The basic idea behind on-the-fly conversion is to produce the conversion as the digits of the quotient are produced by performing concatenations instead of any carries/borrows within a carry-propagate adder. This technique is very similar to the carry-select logic in the carry-select adder.

Another drawback to non-restoring division is that if the partial remainder is negative, a correction is needed. This correction involves subtracting one unit in

the last place or *ulp* from the quotient. This can make calculating the quotient more time-consuming especially for SRT division. Fortunately, we can use on-the-fly-conversion to allow both the $quotient$ and $quotient - ulp$ to be converted without having to use an additional carry propagate adder. This involves have two registers calling Q and QM, respectively. In other words, Q and QM are the following:

$$\begin{aligned} Q[k+1] &= Q[k] + q_{k+1} \cdot r^{-(k+1)} \\ QM[k] &= Q[k] - r^{-k} \end{aligned} \tag{7}$$

where r is the radix and k is the iteration for the serial division algorithm.

On-the-fly conversion requires two registers to contain $Q[k]$ and $QM[k]$. These registers are shifted one digit left with insertion into the least-significant digit, depending on the value of q_{k+1} In other words, depending on the what the following quotient digit, the register either chooses Q or QM and concatenates the current converted quotient digit into the least-significant digit. Figure 4 shows the basic structure where a multiplexor is used to select either Q or QM and some combinatorial logic is used to select Q_{in} and QM_{in}. In order to handle shifting after every cycle, the output of the multiplexors are shifted by one (multiplied by 2) and either Q_{in} or QM_{in} are added to the least significant bit for each load. The registers are updated according to the values in Table 4. The values of C_{shiftQ} and $C_{shiftQM}$ are

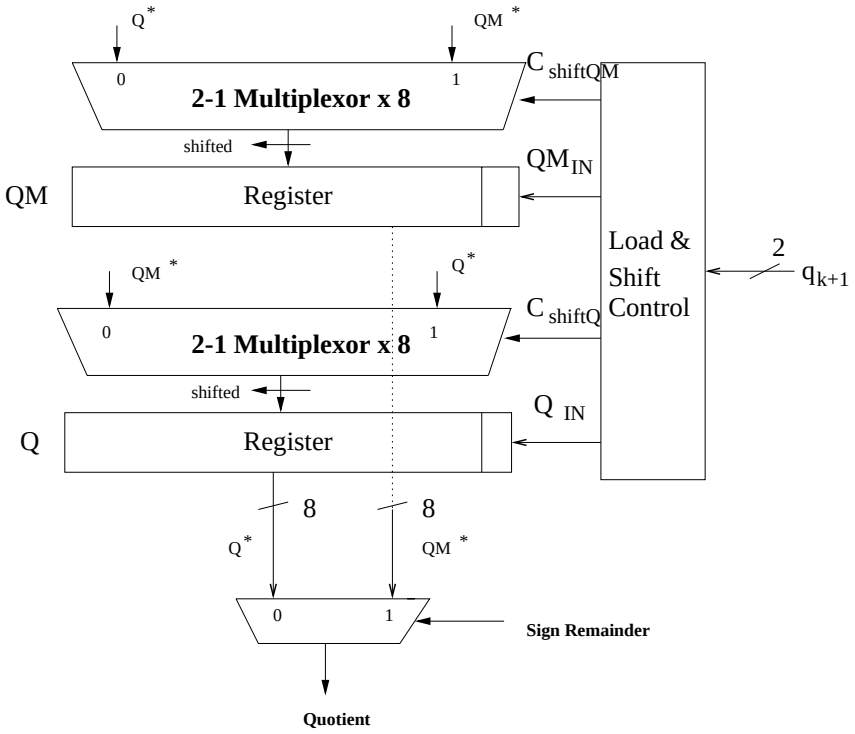

Figure 4: On-the-Fly-Conversion Hardware.

used to control the multiplexors.

For example, suppose that I would like to convert $1101\bar{1}00$ to a conventional representation using on-the-fly conversion. Table 5 shows how on-the-fly-conversion

q_{k+1}	Q_{in}	C_{shiftQ}	Q[k+1]	QM_{in}	$C_{shiftQM}$	QM[k+1]
1	1	1	(Q[k],1)	0	0	(Q[k],0)
0	0	1	(Q[k],0)	1	1	(QM[k],1)
$\overline{1}$	1	0	(QM[k],1)	0	1	(QM[k],0)

Table 4: Radix-2 on-the-fly-conversion.

works updated according to Table 4. At step $k = 0$, the values for both registers are reset which can be accomplished by using a flip-flop that has reset capabilities. In addition, remember that the algorithm is an on-line algorithm. This means that on-the-fly conversion works from the most-significant bit to the least-significant bit. The last value in the register is the final converted value for $Q[k]$ and $QM[k]$ which

0		0	0
1	1	0.1	0.0
2	1	0.11	0.10
3	0	0.110	0.101
4	1	0.1101	0.1100
5	$\overline{1}$	0.11001	0.11000
6	0	0.110010	0.110001
7	0	0.1100100	0.1100011

Table 5: Example Conversion.

is 0.78125 and 0.7734375, respectively. It should be obvious that both of these elements are one *ulp* from each other (i.e. an *ulp* in this case is 2^{-7} or 0.0078125) and 0.78125 is the conventional representation of $1101\overline{1}00$.

4 Design Considerations and Constraints

The following requirements must be met for your design:

- Your chip must fit into a MOSIS "tiny chip" form factor, this means that the overall chip including pads will be 2.2 mm square. We will provide you with a predesigned 40-pin pad frame and pad cells to connect your chip to the outside world; your chip core must fit within this frame. Pad frame magic cells can be downloaded at the following website

  http://www.ece.iit.edu/~jstine/ece429-530/pads/

- You can use any design method you wish to create your chip. However, our experience in last year's course indicates that you will probably be most successful if you use the Synopsys design tools to synthesize standard cell logic blocks from Verilog. Logic blocks should be simulated and synthesized separately first to debug the design and then combined into a single block for final layout.

- Your chip should be designed for easy debugging and testing. Specifically, all sequential logic except for synthesized FSMs should include scan logic that provides a scan path from a "Scan In" pin on the chip through all sequential logic elements, to a "Scan Out" pin. A separate "Scan Mode" input pin should be used to put sequential logic elements in test mode. Since scan logic cannot be easily designed into synthesized FSMs, you can exclude these from the scan chain. However, you should connect the "current state" flip-flop outputs of the FSMs on to output pins through a buffer if you have enough extra pins to do so. This will make it easy to debug the operation of the state machine. We will go over more of this in class

- Your chip should be designed for operation at 10 MHz To receive full credit for the project, your design must be fully simulated in Verilog and IRSIM at the given clock frequency to show correct operation. In addition, your chip should be fully prepared for fabrication by MOSIS following guidelines which will be distributed shortly.

5 Milestones

You will successfully complete this project only if you start immediately. To track your progress, you should hand in reports at the following milestones:

Initial proposal (Due April 12, 2001) Your initial proposal should just be a declaration of who you will be working with.

Architecture Design (Due April 20, 2001) Your architecture design should include a specification of the different subsystems in your design and state diagrams for any FSMs in your architecture. You should hand in a written narrative that describes how these parts will work together. You should create a rough chip floorplan at this point and hand it in.

Component Designs (Due April 27, 2001) At this point you should have designed, simulated, and synthesized each component in your architecture design. You should hand in Verilog descriptions, synthesized block schematics state diagrams, and simulation results for the components at this time. You should also create a more accurate floorplan at this time and hand it in.

Assembled Chip Core (Due April 29, 2001) You should now have all of the subsystems connected together and have simulation well underway. You should hand in a flea plot of your design so far and any simulation results that have been completed.

Final Report (Due May 6, 2001) You should have a complete design, including chip pads and all connections. Your entire chip should be throughly simulated with IRSIM to check for correctness. You should hand in all design work created in previous steps (corrected if necessary), the IRSIM input vectors you

used to test your circuit, and a report describing what you did, what problems you encountered, what you learned. If you work in a team, indicate which parts of the project each team member worked on. There is a checklist posted on the announcements section of the class homepage that is useful in terms for fulfilling all the guidelines.

For more information about the final report, see the final report guidelines published on my web page.

6 Chip Fabrication and Testing

If you complete your chip, you may be able to have your chip fabricated by the MOS Implementation Service. To do this, you must enroll in a follow-on "chip testing" course to be offered during the Fall 2002 semester in which you will test your chip and write a test report that will be submitted to MOSIS describing the operation of your fabricated chip. If you are interested in this option, please let me know and indicate it on the cover page of your final report.