SAN JOSÉ STATE UNIVERSITY
Charles W. Davidson College of Engineering
DEPARTMENT OF ELECTRICAL ENGINEERING

**EE 271** **Final Design Project** **Fall 2016**

# Implementing a 64-bit Signed Binary Multiplier & Divider Circuit

The purpose of this project is to implement, optimize and analyze a 64-bit signed binary multiplier and divider circuit as shown in Figure 1 and Figure 2. The ALU unit can be implemented based on any desired addition/subtraction algorithm and the whole circuit must be modeled at RTL level. Analysis must be performed for four important metrics: number of clock cycles & clock period, total time delay of an operation (multiplying/dividing), circuit area, and power consumption. All your work, including the details of your implementation, test, measurement, analysis, design options, EDA tool setting and configurations must be clearly described as stated in the project report template.

## I. Design Architecture

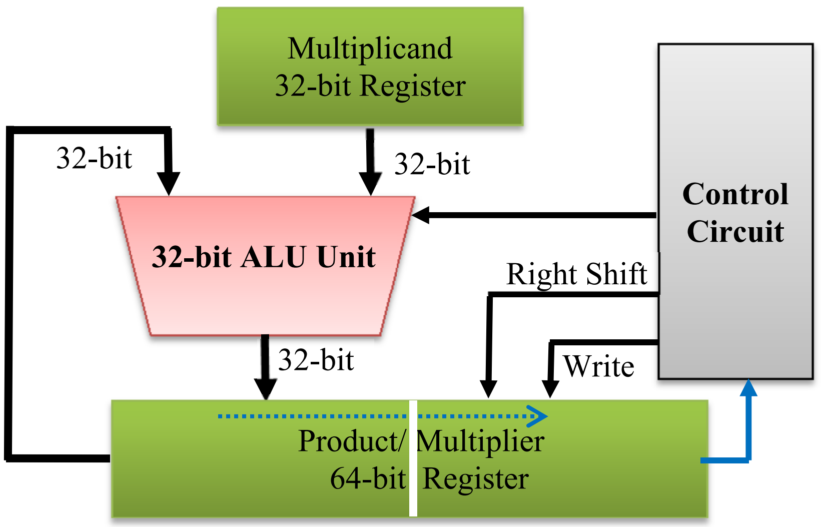

**Figure 1**: Architecture of Multiplier

Multiplication Process:

1. Load multiplicand into Multiplicand Register
2. Load 0 in the upper half of the Product/Multiplier Register and load multiplier in the lower half of the Product/Multiplier register. The least significant bit (LSB) of the Product/Multiplier register is the "OPE" flag
3. Repeat for *n* times (*n* is 32 in this case)
   - If OPE flag is 1, add multiplicand to upper half of the Product/Multiplier Register and store the result back into Product/Multiplier Register. If OPE flag is 0, do nothing
   - Right shift the Product/Multiplier register by 1 bit

***Note***: *At the end of the process, the product (result) is in the Product/Multiplier Register and the multiplier data is deleted.*

For negative operands, the circuit:

- Converts negative numbers to positive
- Operations are performed with the magnitudes
- The results are converted to negative if the signs of the operands are different

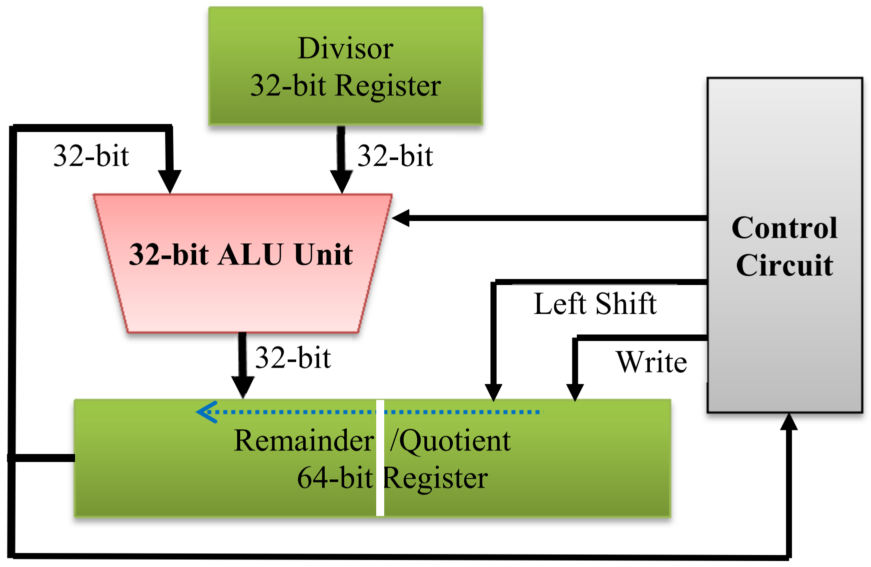

**Figure 2**: Architecture of Divider (non-restoring scheme)

<u>Division Process:</u>

1. Load divisor into Divisor Register
2. Load Dividend into Remainder/Quotient Register
3. Set OPE flag to 1
4. Repeat for *n*+1 times (*n* is 32 in this case)
   a. Subtract (if OPE = 1) or Add (if OPE = 0) Divisor Register from/to upper half of Remainder/Quotient Register and store the result back into upper half of Remainder/Quotient Register
   b. If the result in upper half of Remainder/Quotient Register is:
   <u>Positive or 0</u>:
   - Left shift the Remainder/Quotient Register* by 1 bit and fill up the LSB (vacant bit) with 1
   - Set OPE flag to 1

   <u>Negative</u>:
   - Left shift the Remainder/Quotient Register* by 1 bit and fill up the LSB (vacant bit) with 0
   - Set OPE flag to 0

*: For the last step (step *n*+1), left ship ONLY the lower-half of the Remainder/Quotient Register

***Note**: At the end of the process, the quotient (result) is in the lower-half of the Remainder/Quotient Register, the remainder (result) is in the upper-half of the Remainder/Quotient Register, and the dividend data is deleted.*

<u>For negative operands</u>, the circuit:

- Converts negative numbers to positive
- Operations are performed with the magnitudes
- The results are converted to negative if the signs of the operands are different

## II. Design Specifications

The circuit should be implemented in the way such that the multiplier and divider circuits will share the hardware resources as much as possible (same ALU unit used for both Multiplier and Divider circuit, Multiplicand Register and Divisor Register are the same register, Product/Multiplier Register and Remainder/Quotient Register are the same register, etc.). The **exact** interface ports of the circuit must be as below:

- `opera1 [31:0]`: For multiplication, this is a 32-bit multiplier. For division, this is a 32-bit divisor
- `opera2 [63:0]`: For multiplication, the lower 32 bits represent 32-bit multiplicand. For division, this is a 64-bit dividend
- `result [63:0]`: For multiplication, this is 64-bit product. For division, the lower 32 bits represent 32-bit quotient and the upper 32 bits represent 32-bit remainder
- `muordi`: 0 is for multiplication and 1 is for division
- `clock`
- `reset`
- `start`: You can implement the start signal as either positive-edge active or high-level active.
  Positive-edge active: The circuit is initially idle and will start at the positive-edge of the `start` signal. The circuit restarts whenever the `start` signal once again changes from 0 to 1
  High-level active: The circuit is initially idle and will start whenever the `start` signal is high (logic 1). For each clock, the circuit checks the `start` signal and restarts whenever the `start` signal is high (logic 1). So in this case, the `start` signal should not active for more than 1 clock period
- `valid`: Active-high output signal from the circuit to inform the exterior that the valid result (64-bit product or 64-bit quotient and remainder) is available. The `valid` signal becomes inactive (`0`) at the positive-edge of the `start` signal and become active (`1`) whenever the calculation was completed.

All your work, including the details of your implementation, tests, measurement, and analysis should be included in your report. Note that for this project, the analysis is very important and should be documented clearly and neatly. The report must include exact information as requested. Do NOT skip the information and do NOT report extra information. Grading will be based on the quality and completeness of your project and report.

## III. Libraries and EDA Tools

- Use Toshiba library at **`/apps/toshiba/sjsu/synopsys/tc240c/`** for your synthesis and analysis. This is a technology library with 250 nm technology operating at 2.5V. For best and worst delays (hold and setup time violation checks), use **`tc240c.db_BCCOM25`** and **`tc240c.db_WCCOM25`**, respectively. Note that in using Toshiba library, the `tc240c.workview.sdb` is the symbol library that you should use.
- RTL-level simulations can be performed on any simulator but gate-level (netlist) simulations must be performed with either **Synopsys VCS** or **Cadence NCVERILOG**. Synthesis must be performed with **Synopsys Design Vision** (or **Design Compiler**)
- All simulations must be performed at both RTL and gate (netlist) levels

Other technology libraries available on the system that you can try include the 0.18 µm and 0.25 µm technologies from TSMC and 0.35 µm and 0.5 µm technologies from AMI as listed below:

- `/apps/cadence/OSU_LIB/osu_stdcells/lib/tsmc018`
- `/apps/cadence/OSU_LIB/osu_stdcells/lib/tsmc025`
- `/apps/cadence/OSU_LIB/osu_stdcells/lib/ami035`
- `/apps/cadence/OSU_LIB/osu_stdcells/lib/ami05`

Or default LSI libraries available from Synopsys (`lsi_7k.db, lsi_9k.db, lsi_10k.db`) at `/apps/synopsys/SYNTH/libraries/syn`

## IV. Design Implementation and Testing

The implementation must be done at RTL level in order for you to control the architecture and structure of your designs. You are **NOT** allowed to use Verilog arithmetic operators +, -, *, and / for hardware implementation (but of course you can use these operators for programming purposes and in your testbenches.) You must make sure that the final optimized circuits that you implemented have structures as you intended to implement.

Besides testing each module during the project development, you are required to develop final test cases to comprehensively test the operations of the designs and display (by using both `$display` system function and waveforms) the input operands and output results together with timing data to represent the circuit delay performance. Final test cases must be able to perform pre-synthesis (RTL) functional verification and **post-synthesis (netlist) functional and timing** verifications. Select a set of test data such that you can re-verify the static timing (during the synthesis) with dynamic timing (during post-synthesis simulation). Select the data point that you want to output in order for you to exam the dynamic timing delays as function of the input operands.

*In order to run VCS with* `netlist`*, you need to point to the Verilog source of the target library. As an example of synthesizing with Toshiba library, the VCS command for simulating your netlist named* `mudi64_netlist` *should be as below:*

```
vcs +v2k –debug_all –gui –y /apps/toshiba/sjsu/verilog/tc240c
+libext+.tsbvlibp mudi64_TB.v mudi64_netlist.v
```

Moreover, you can use Cadence simulator (`ncverilog`) as below:

```
   ncverilog -y /apps/toshiba/sjsu/verilog/tc240c +libext+.tsbvlibp
   +access+r mudi64_TB.v mudi64_netlist.v
```

where `.tsbvlibp` is the suffix of Verilog source files of `tc240c` library and `/apps/toshiba/sjsu/verilog/tc240c` is the library directory

Since tc240c library has nanosecond time scale and 10 picosecond time-precision, the timescale directive below should be included in the Verilog testbenches

```
   `timescale 1 ns / 10 ps
```

Note that some Verilog files in this technology library are encrypted (protected) by Cadence EDA tool and VCS may not be able to decrypt them. If that is the case, you will then get error messages. Using `ncverilog` for netlist simulation should be fine since the library was provided to us by Cadence. If anyway you get error message about file protection, try to find out in your netlist which encrypted Verilog files that are used and then try to replace them with non-encrypted files. Below is the list of encrypted files.

- `tsbLD2SFprim.tsbvlibp`
- `tsbSCK2prim.tsbvlibp`
- `tsbMUXXprim.tsbvlibp`
- `tsbSCK1prim.tsbvlibp`
- `tsbCTLprim.tsbvlibp`
- `tsbFD2BASICprim.tsbvlibp`
- `tsbFD4BASICprim.tsbvlibp`
- `tsbCHKprim.tsbvlibp`
- `tsbRCK1prim.tsbvlibp`
- `tsbCLD3SFprim.tsbvlibp`
- `tsbCLD4SFprim.tsbvlibp`
- `tsbMAJprim.tsbvlibp`
- `tsbLDP1prim.tsbvlibp`
- `tsbFD1BASICprim.tsbvlibp`
- `tsbCFD3BASICprim.tsbvlibp`

## V. Project Report and Report Submission

EE271 final project is an individual project. Each student will work on the project independently and no information should be shared among students. Each student is required to submit a full report in **PDF format** on class CANVAS (a report template is provided on the class canvas). Students are responsible for providing completed information such that grader can re-simulate and re-synthesize the design for grading purposes. **Be aware that the submitted report will be plagiarism checked with turn-it-in and students will NOT see the similarity results but only the graders.**

The project report must be in the right format and must include completed information about the project as shown in the report template. Download the report template on the class canvas and edit your report directly from the template. Do not change the format, fonts, and page setup of the report template but just fill-in and writing the requested information. Do NOT skip the information and do NOT report extra information. Grading will be based on the quality and completeness of your project and report.

The report file name MUST be in the format as below:

**SJSUID_271F16_Lastname.pdf**