Dynamic Partial Reconfigurable Calculator Module

List of Authors (Jeanne Beau, Zachary Martin)
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: beau@oakland.edu, zmartin@oakland.edu

***Abstract*—This project levies a DPR calculator design with a serial monitor based user interface to achieve an arithmetic logic unit with extensive scalability at a consistent size.**

I. Introduction

The Dynamic Partial Reconfigurable Calculator Module (DPRCM) is a DPR based calculator which is dynamically modified to fit a given operation assigned by the user. This system integrates a UART serial port terminal to allow the user to pass in mathematical operations and observe their results. The purpose of this project is to illustrate the potential savings in space offered by a DPR design by utilizing the significant scalability of operators while maintaining the small size offered by the FPGA. Other ALU units, on the other hand, require an exponential increase in size with each supported operation as well as increased measures for thermal regulation. Here, we'll explore this approach of a reconfigurable fixed-size ALU with four operators and conclude the experiment with main takeaways and how they relate to some topics covered in class. We will levy the lectures given on dynamic reconfigurable hardware design to complete this project.

II. Methodology

The design plan for this system will integrate distinct software and hardware tasks.

A. Software

The software logic design is shown in the figure below. At the startup of the application, a main menu will be displayed via UART, which will include a general opening message, a list of supported operators, and a prompt for the user to enter a valid operator. The system will check if the input is valid, then prompt for the first number. The user will then be requested to enter a decimal number as the first operand. This process will be repeated for the second operand. The operator, first operand, and second operand will then be parsed and transferred to the hardware accordingly. The system will determine how the ALU will be reconfigured to meet the requirements of the selected operation. It will then wait for the output (either via polling or predetermined delay), read the data, and display the result back on the terminal. Once completed, the system will return to the main menu and await the next operation.

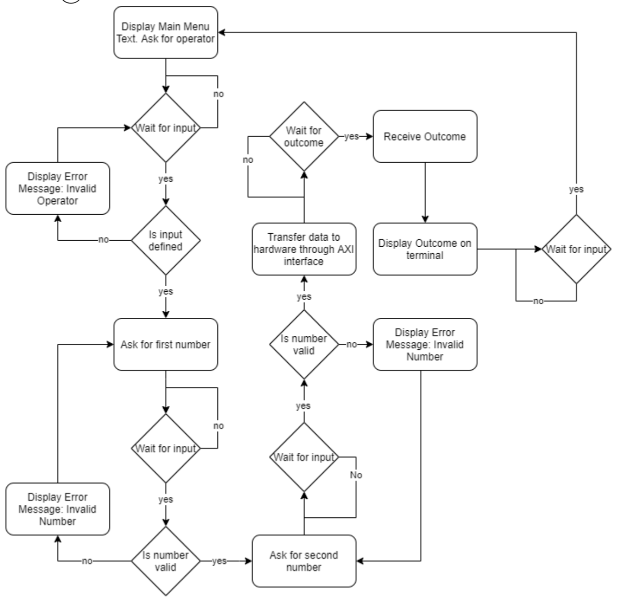

Display Main Menu Text. Ask for operator
no
Wait for input
yes
Display Error Message: Invalid Operator
no
Is input defined
yes
Ask for first number
no
Wait for input
yes
Display Error Message: Invalid Number
no
Is number valid
yes
Ask for second number
Wait for input
No
yes
Is number valid
no
Display Error Message: Invalid Number
yes
Transfer data to hardware through AXI interface
Wait for outcome
no
yes
Receive Outcome
Display Outcome on terminal
Wait for input
no
yes

Figure 2A: Software Control Logic of UART interface

The following listing illustrates the pseudo code of the systems state machine:

```
infinite loop
switch app state

case operator select:
Display_request_for_operator();

stream_terminal_input();

if input is valid operator then
    store operator in variable
else
    app state = operator state
end if

case input one state:
Display_request_for_input_1();

stream_terminal_input();

if input is valid number then
    if input is > 32 bit then
        app state = input one state
        display error input too large
    else if input is > 16 bit
        set bus size variable to 32 bit
    else
        set bus size variable to 16 bit
    end if

    store input one in variable
else
    app state = input one state
    display error invalid number

case input two state:
Display_request_for_input_2();

stream_terminal_input();

if input is valid number then
    if input is > 32 bit then
        app state = input two state
```

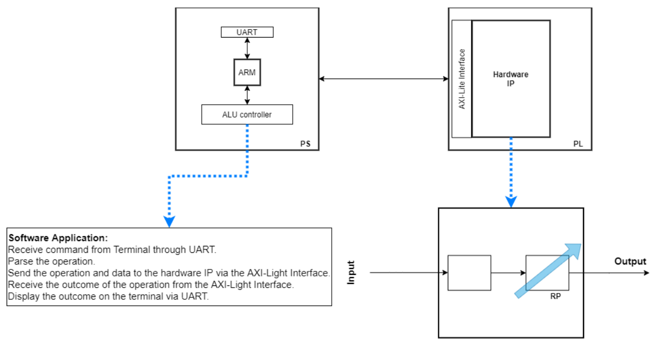
```
            display error input too large
        else if input is > 16 bit
            set bus size variable to 32 bit
        else
            set bus size variable to 16 bit
        end if
        store input two in variable
    else
        app state = input two state
        display error invalid number

    case reconfigure hardware:
    if operator is addition
        if bus size is 32 bit
            upload partial bitstream of 32
              bit bus addition configuration
        else
            upload partial bitstream of 16
              bit bus addition configuration
        end if
    else if operator is subtraction
        if bus size is 32 bit
            upload partial bitstream of 32
              bit bus subtraction configuration
        else
            upload partial bitstream of 16
              bit bus subtraction configuration
        end if
    else if operator is multiplication
        if bus size is 32 bit
            upload partial bitstream of 32
              bit bus multiplication configuration
        else
            upload partial bitstream of 16
              bit bus multiplication configuration
        end if
    else if operator is division
        if bus size is 32 bit
            upload partial bitstream of 32
              bit bus division configuration
        else
            upload partial bitstream of 16
              bit bus division configuration
        end if
    end if

    case write data:
    write input one
    wait 15ms
    write input two
    wait 15 ms

    case read data:
    if bus size is 32 bit
        high 32 bit of 64 bit term_output_64 is
              first AXI read
        low  32 bit of 64 bit term_output_64 is
              second AXI read

        display 64 bit term_output_64 on terminal
    else
        the high 16 bits of the 32 bit term_output_32
              is the first AXI read
        the low 16 bits of the 32 bit term_output_32
              is the second AXI read

        display 32 bit term_output_32 on terminal
    end if

    app state = operator select
```

Listing 2A: Software Control Logic in Pseudocode

This state machine starts with a main menu which displays the name of the application as well as some additional information. The next state waits for the user to enter a valid operator to perform from a list of addition '+', subtraction '-', multiplication '*' and division '/'. If an invalid operator is entered, an error is displayed and the state restarts. Next, the system scans for the user to enter the first operand. If the input is too large an error is returned and the state restarts. Otherwise, if the input is valid then the number is stored into a variable called input_1 and the system moves to the next state. This state will scan for the second operand, similar to the previous state. If successful, the state machine transitions to the processing state. The PS loads the appropriate partial bitstream onto the RP and then passes the inputs through the AXI-Lite interface. Then the software reads two 32 bit values from the hardware, reconfigures the data into an easily readable format, and outputs the final result through the uart terminal. Some additional processing is done to format the result into a more readable configuration for division such that the quotient 'q' and the remainder 'r' are clearly labeled. Once the output is transmitted, the system transitions back to the main menu.

B. Hardware

A high-level overview of the system's hardware design is illustrated in the figure below.



Software Application:
Receive command from Terminal through UART.
Parse the operation.
Send the operation and data to the hardware IP via the AXI-Light Interface.
Receive the outcome of the operation from the AXI-Light Interface.
Display the outcome on the terminal via UART.

Figure 2B: High-Level Overview of Design Architecture

This hardware uses an AXI-Lite peripheral to interface with the software and receive two 32-bit numbers and a given mathematical operation, which will be reconfigured according to the operator. The processed output is stored in registers to be read via the AXI-Lite interface. The reconfigurable partition will contain an ALU unit which will be partially reconfigured as one of four operations (add, subtract, multiply, divide), emphasized in the figure below.

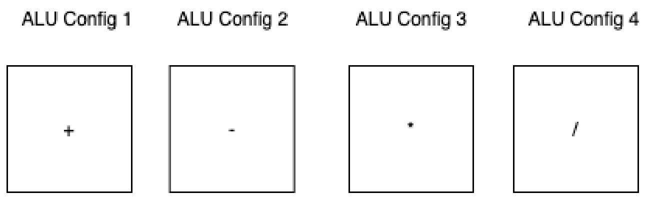


Figure 2C: Reconfigurable Operation Modules

The datapath of the system using the AXI-Lite interface is depicted in the following figure. This architecture will only need 4 slave registers (00 to 11). Registers 00 and 01 will load two 32 bit inputs into the reconfigurable unit, and the 64 bit result will be multiplexed into registers 10 and 11.

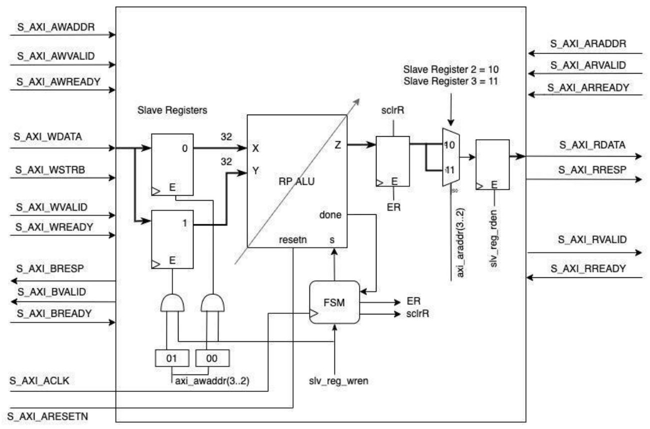


Figure 2D: PL Datapath with AXI-Lite Interface

To keep the reconfigurable hardware simple, the system will always write two 32 bit values for the inputs (one for input one and again for input two) and will read two 32 bit values of the 64 bit result (the most significant 32 bits of the result being stored in the first read, and the lowest 32 bits of the result in the second).

This architecture will also include a finite state machine (next figure) to wait for the inputs as well as the completed operation (done signal) from the ALU. This will be driven by the slv_reg_wren signal to clear the previous result flip flop. It will then wait for this signal a second time, and then the done signal to make the data available to be read from the S_AXI_RDATA bus.

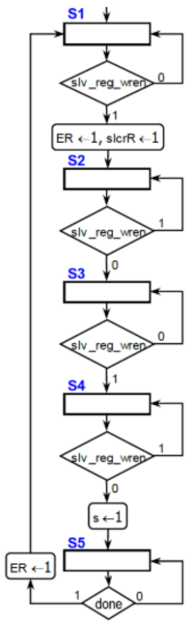


Figure 2E: FSM of ALU AXI-Lite Interface

III. Experimental setup

The experiment will begin with creating the necessary hardware components and simulating the circuit and the IP to be created. Then software will then test the functionality of the AXI-Lite interface to validate successful communication with the hardware. After which, the static and reconfigurable portions will be defined and the filesystem will be set up to ultimately generate the full and partial bitstreams. Lastly, the software portion is written to implement the logic described in Listing 2A.

A. Hardware Generation

The ALU component is created as a file with a generic single character select to generate one four operations. This component is simulated with a testbench in Vivado and each operation performed is tested with various operand inputs. Once the behavior of the circuit is confirmed to be correct, the peripheral is then built and simulated to verify that the circuit functioned through the AXI-Lite bus. The IP is then packaged and a block design is created with the Zynq PS and the custom peripheral. After the VHDL wrapper is generated and the bitstream is exported, a test software program is written to confirm the functionality of the hardware through the AXI-Lite interface. This process allows us to verify that the hardware portion works as expected. The final portion of the hardware setup is to define the static and dynamic portion of the system, as well as recreate a file system similar to the one demonstrated in Laboratory 5. Each of the 4 reconfigurable modules (RMs) source directories are populated with the modified reconfigurable partitions exposed accordingly. Additionally, the folders for implementation and synthesis are created. Once the hierarchy for the file system is established, the TCL script will be able to synthesize each of the RMs.. After synthesis, the floorplan is drawn and an appropriate allocation is determined. The floorplan will eventually be configured appropriately and the modules will each be implemented on to the design. The full and partial bitstreams will be generated (as well as the swapped-byte bin files) and will be made available on the SD to be read in by the Zybo board for DPR.

B. Software Implementation

The first portion of the software implementation is the memory allocation for the partial bitstreams. The code for this memory allocation, and the functions involved with loading the partial bitstreams, was taken from the example code provided by the Dynamic Partial Reconfiguration – PS+PL tutorial [3]. Here, the partial bitstreams are loaded from SD memory into RAM to be called by the PS to reconfigure the reconfigurable partition (RP) as explained in the "Using SD Cards" tutorial [2]. This requires that the board support packages include xilffs with the option of use_strfunc enabled for SD card usage. Since the partial bitstreams are stored in RAM, sufficient stack and heap sizes need to be set using the linker generation tool. For this system, a 10MB stack size is assigned. With the memory allocation completed, the software enters the state machine.

The software will go through a series of sanity checks to validate and apply appropriate modifications. The state machine was run with various inputs to ensure that each error check, such as inappropriate characters and values greater than 32 bit numbers, are appropriately triggered and displayed. In addition, several operations are tested to validate the output of the system. These numbers ranged from small 16 bit values to the maximum allowed values for 32 bit operations in both positive and negative numbers. The results will be verified using Microsoft's programmer calculator application.

IV. Results

A. Simulation

Two testbenches were used for behavioral hardware simulation. The figure below shows the testbench result of the standalone ALU component doing a division operation on Xin and Yin, shown in decimal. The result shows the 64 bit Zout with the most significant 32 bits representing the quotient and the least significant 32 bits being the remainder, displayed in hex.

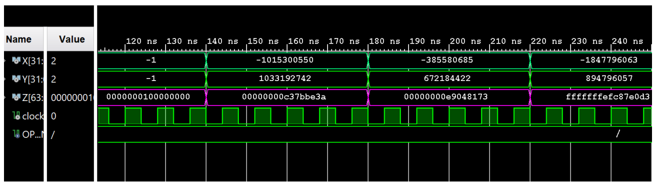


Figure 4A: ALU Component Simulation (Division)

The next figure is the simulated result of the custom AXI-Lite peripheral containing the ALU with the addition operation. We can see that the write data is -1 and -1, and -2 in 64 bits is then available on the read data bus. These two simulations allowed us to confirm the functionality performed as expected.

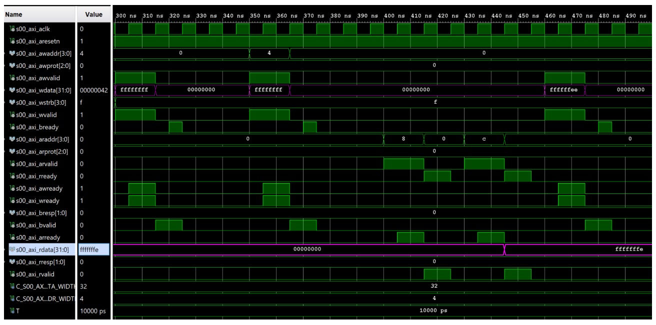


Figure 4B: ALU AXI-Lite Peripheral Simulation (Addition)

B. Implementation and Floorplanning

After synthesizing all four of our reconfigurable modules in the Vivado TCL Shell, the checkpoint for the divider module was read in first, as it was thought to be the module that would be largest in size (this was true). The floorplan was generated and the process continued for each module until we reached the multiplication configuration. When attempting to place the design, it was found that the necessary cells, DSP48, needed for this operation, were missing. Thus, the floorplan step had to be redrawn. The second time, we started with the multiplication RM and regenerated the floorplan constraints, making sure to include the DSP 48 cells. However, during this iteration, we found that the size of the floor plan drawn was too small for the divisor operator. Thus, we needed to create the floorplan constraints a third time. This is of course a big step with creating partial reconfigurations as we observed how each RM may need varied specialized cells or space. This may point to a limitation of a single reconfigurable partition. In this case, the amount of space required for addition, subtraction, and multiplication was significantly less than division. In addition, the multiplication block needed a specific collection of cells in order to perform its function. Our floor plan as a result encompassed a very large partition that would only be needed for one configuration, as well as block off specialized cells that would also only be needed for one configuration. This poses a question as to how design decisions are made in a dynamic partially reconfigurable system where circuits have significantly varied hardware specifications. In this case, we may not actually benefit the same way compared to if the modules needed similar hardware resources. The floorplan was eventually confirmed to be appropriate when each of the configurations were implemented successfully.

C. Software Terminal Output

The following listing was taken from the UART serial terminal output through several operation tests:

```
ECE5736 PARTIAL RECONFIGURABLE CALCULATOR MODULE
          JEANNE BEAU, ZACHARY MARTIN
          06/22/21
ENTER OPERATOR ( '+' , '-' , '*' , '/' )
>+
ENTER FIRST OPERAND
>1234
ENTER SECOND OPERAND
>5678
          1234+5678 = 6912

ENTER OPERATOR ( '+' , '-' , '*' , '/' )
>+
ENTER FIRST OPERAND
>-600000
ENTER SECOND OPERAND
>-567891234
          -600000+-567891234 = -568491234

ENTER OPERATOR ( '+' , '-' , '*' , '/' )
>-
```

```
ENTER FIRST OPERAND
>12345
ENTER SECOND OPERAND
>678910
                12345-678910 = -666565

ENTER OPERATOR ( '+' , '-' , '*' , '/' )
>-
ENTER FIRST OPERAND
>-12345
ENTER SECOND OPERAND
>-6789
                -12345--6789 = -5556

ENTER OPERATOR ( '+' , '-' , '*' , '/' )
>*
ENTER FIRST OPERAND
>-1234
ENTER SECOND OPERAND
>-56789
                -1234*-56789 = 70077626

ENTER OPERATOR ( '+' , '-' , '*' , '/' )
>*
ENTER FIRST OPERAND
>1234
ENTER SECOND OPERAND
>5678
                1234*5678 = 7006652

ENTER OPERATOR ( '+' , '-' , '*' , '/' )
>/
ENTER FIRST OPERAND
>3
ENTER SECOND OPERAND
>2
                3/2 = q:1     r:1

ENTER OPERATOR ( '+' , '-' , '*' , '/' )
>/
ENTER FIRST OPERAND
>4
ENTER SECOND OPERAND
>2
                4/2 = q:2     r:0
```

Listing 4A: UART Terminal Output

During the final validation of the system, the divider was observed to have inconsistent results. The first solution to this problem was to increase the delay between write and read operations through the AXI-Lite interface to the hardware. This had mixed results, allowing for more consistent and accurate outcomes for very small operands but still inconsistent outcomes for larger operands. Additionally, the scale to this delay was not obvious but exponential. Further exploration revealed that the division operator would function correctly every second call. With this observation, the data write operation to the hardware was modified to call twice only for division. The result was a consistent and accurate output for various operand combinations. Some speculation has been made as to the cause of this issue. First being a large propagation delay in the generic divider which is why the increase in delay time had some beneficial effect. Second is that the divider may require additional cycles in the ALU state machine to allow for the data to complete the process. Future work could improve this with a more synchronous and sequential approach, rather than the current combinational design implemented by the library. In addition the component would drive the done signal such that the output register would only store the completed value. This would ensure that we would not need to predict how many clock cycles the procedure would take and assert when the data is ready to be read.

V. Conclusions

In summary, we were able to create a dynamic partially reconfigurable system by way of an on-the-fly adjustable arithmetic logic unit prompted by a software interface via UART user input. The hardware for the operator component as well as the AXI4-Lite was generated and simulated, then tested via software. The peripheral was defined into dynamic and static portions and the bitstreams were generated to be used for DPR. We were able to further expand the software to accept user input via UART and create our intended calculator application defined in Section II. The main design challenges were generating an appropriate floorplan constraint that would fit the needs of each module, as each configuration had different requirements for cell types and size. This led to a discussion about design choices and tradeoffs with what types of designs should include dynamically partially reconfigurable hardware within a single module. The final challenge was discovering our software application resulted in inconsistent results with the division operator. Through simulation, we were able to rule out the possibility that the component itself was behaviorally malfunctioning and we instead experimented with the software. The resulting work around for this obstacle ultimately pointed to the possibility that the number of clock cycles needed for the division was unpredictable, and was usually more than we allowed before reading the result. This is related to the topic discussed in class regarding timing constraints and the relationship between clock cycles and operations per cycle. Further improvements could be made to the hardware to include a more sophisticated synchronous division design, where the done signal would be asserted by the divider and number of clock cycles would not have to be derived in any way. As a result of this experiment, we were introduced to some key insights into dynamic partial reconfigurable systems as well as a unique experience with development on a programmable system on chip.

VI. References

[1] Llamocca, D. (n.d.). Tutorial: Embedded System Design for ZynqTM SoC, "Custom Peripheral for the AXI4-Lite Interface". Rochester; ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT, OAKLAND UNIVERSITY.

[2] Llamocca, D. (n.d.). Tutorial: Embedded System Design for ZynqTM SoC, "Using SD Card" Rochester; ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT, OAKLAND UNIVERSITY.

[3] Llamocca, D. (n.d.). Tutorial: Embedded System Design for ZynqTM SoC , "Dynamic Partial Reconfiguration – PS+PL". Rochester; ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT, OAKLAND UNIVERSITY.