Reconfigurable Instruction Set

Thomas Filarski, Kristi Stefa
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: tfilarski@oakland.edu, kstefa@oakland.edu

***Abstract*— A simple processor that has a reconfigurable instruction and data set, where the outputs are read into software.**

1. Introduction

This project will be designed to run a simple CPU that will take a set of 32-bit data and use this data to compute different tasks. This data will be split into three different sections, the instructions, which will be 3-bits, the data, which will be 16-bits, and a single bit that will instruct the hardware to load the data onto the output FIFO. The remaining 12-bits will be unused for this project. The data will be stored in a LUT which can be reconfigured to hold different 32-bit data which may contain different data and/or instructions.

2. Allocation of Tasks

A. Software routine

This will enable the CPU to start its computations. After the CPU has completed running the set of instructions, the software will then read the results of the instruction set. Once the first instruction set is complete a new set of instructions will be loaded onto the FPGA and then the previous output data will be run. The software will read the results again. The input structure would be similar to Assembly, where the instructions incorporate control bits and 16 or 8 bits of data. The instructions can be written into the test code or a stack can be created in a text file for a long list of procedures.

Pseudo Code

```
program start
  initialize data
  for n to m
        write data x□
         .
         .
        write data x□

        read data y□
        .
        .
        read data y□

  reconfigure hardware


  for n to m
        write data y□
        .
        .
        write data y□

        read data z□
        .
        .
        read data z□
```

B. Hardware Routine

This portion will consist of the AXI interface, input and output FIFOs, and the reconfigurable module. The AXI interface will communicate with the software portion of the design through a set of FIFOs that will control the incoming and outgoing signals. This will be done with an FSM that will keep track of the full and empty signals from the input and output FIFOs respectively.

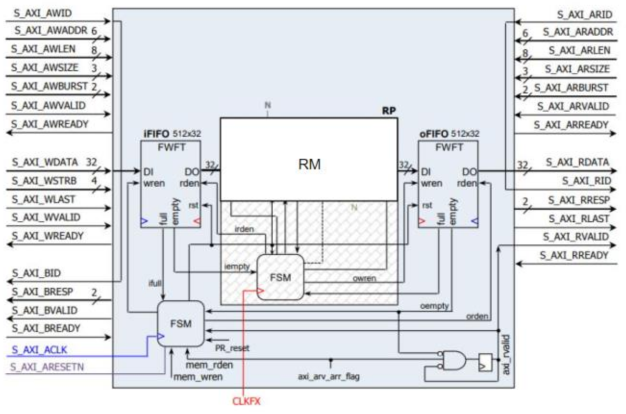
The reconfigurable module will consist of a simple CPU and an FSM. The FSM, shown in Figure 2, that is contained within the RP has 3 states, the first checks to see if the input is empty, if it is then it will move to the second state. Here it will check to see if the input is empty and if the output is full for the input FIFO and output FIFO respectively. If these are both not true then it will move to the third state while setting the signal 'E' high, which is sent to the decoder within the CPU, and allowing the CPU to read data by setting 'rden' to 1. In the third phase it will check if the software wants the data to be loaded onto the output FIFO, if it does not then it will go back to the second phase, if it does then it will enable the data to be written to the output FIFO then move to the second state.

The reconfigurable portion will change the instructions within the CPU. These instructions are listed below in two separate sets and show which signals are set by each instruction. The full architecture will be available in both configurations but the operations will change as seen in the instruction table below.

Instruction	Short Hand	Reg1	Reg2	RegOut	sel
Set 1					
Load1	LD1	1	0	0	0
Load2	LD2	0	1	0	0
Addition	ADD	0	0	1	0
Subtraction	SUB	0	0	1	0
Left-Shift	LSH	0	0	1	0
Right-Shift	RSH	0	0	1	0
Increment	INC	0	0	1	0
Decrement	DEC	0	0	1	0
Set 2					
Load1	LD1	1	0	0	0
Load2	LD2	0	1	0	0
Add Accumulate	AAC	0	0	1	1
Sub Accumulate	SAC	0	0	1	1
Zero's	ZER	0	0	1	0
One's	ONE	0	0	1	0
Passthrough	PST	0	0	1	0
Not	NOT	0	0	1	0

Table 1: The sets of instructions

3. Hardware Design Architecture



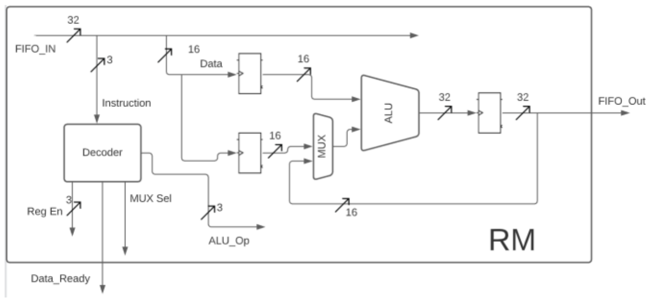


Figure 1(a): Top level AXI design (b) Internal RM design

The reconfigurable portion would consist of the ALU and its inherent operations as well as the instructions inside the decoder and the FSM shown in Figure 1. This will allow two different modes of operations based on what kind of operations are required. The modular component will be the architecture of the processor, with variations in its design structure, register mapping, signal routing, and encoded ALU. This allows for different modes of operation and a flexible design depending on the operations that are required.

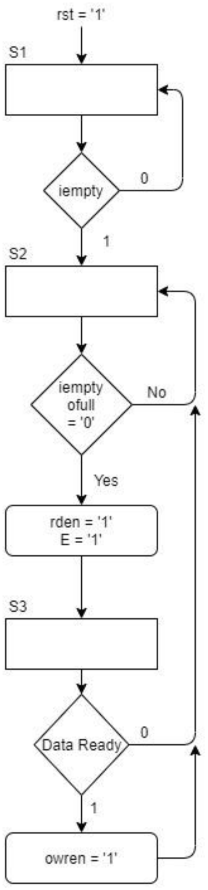


Figure 2: FSM in the RP

The AXI peripheral will provide the means for interfacing with the hardware through the software routine. AXI-Full + FIFOs will be the configuration used and it will surround the RM and a custom FSM. The AXI components will mirror previous projects and edits will come internal to the peripheral and RM. The RM contains some interface logic with the FSM to handle states, resets after DRP, and data_ready when the operations are ready. The FIFOs will hold the instructions and data similar to a stack and then the process will be loaded when the hardware is prepared. The output will be stored on the oFIFO and read for comparison of results.

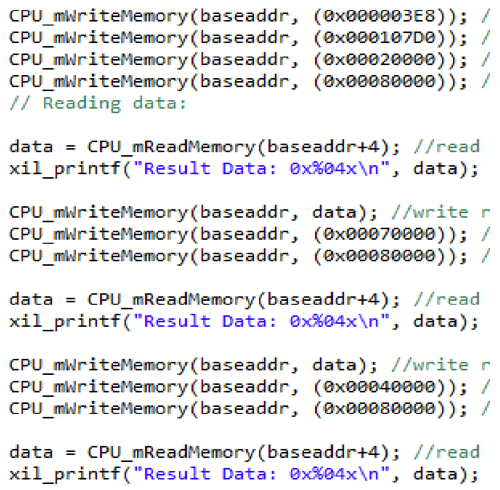
4. Experimental Setup

The experiment and results were focused around using a set list of instructions and data inputs to get a list of expected results. These one cycle operations would include loads, adds/subtracts, and bit operations to create intermediate results that will be stored on the FIFOs. These FIFOs are read by the C code and displayed on the console for verification. A function was created to test each configuration for the set functions and processes as well as save the values between each stage so that the flow is continuous. This allows one set of inputs to be transformed by both sets of instructions and modules before an output is reached. The individual modules are simulated in Vivado for their waveforms and the C code is tested through the serial monitor.

The test scenario for the first processor was to take in the inputs 0x03E8 and 0x07D0 and perform the following operations :

LOAD 1
LOAD 2
ADD 1&2
DECREMENT
SHIFT LEFT

These operations can be seen in the C code below and the final read of the data value.

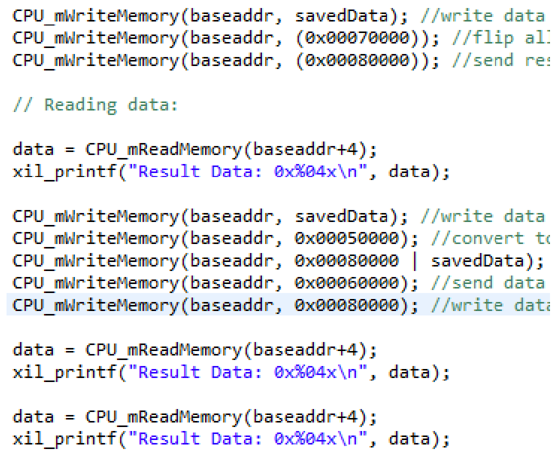


```
CPU_mWriteMemory(baseaddr, (0x000003E8)); /
CPU_mWriteMemory(baseaddr, (0x000107D0)); /
CPU_mWriteMemory(baseaddr, (0x00020000)); /
CPU_mWriteMemory(baseaddr, (0x00080000)); /
// Reading data:

data = CPU_mReadMemory(baseaddr+4); //read
xil_printf("Result Data: 0x%04x\n", data);

CPU_mWriteMemory(baseaddr, data); //write r
CPU_mWriteMemory(baseaddr, (0x00070000)); /
CPU_mWriteMemory(baseaddr, (0x00080000)); /

data = CPU_mReadMemory(baseaddr+4); //read
xil_printf("Result Data: 0x%04x\n", data);

CPU_mWriteMemory(baseaddr, data); //write r
CPU_mWriteMemory(baseaddr, (0x00040000)); /
CPU_mWriteMemory(baseaddr, (0x00080000)); /

data = CPU_mReadMemory(baseaddr+4); //read
xil_printf("Result Data: 0x%04x\n", data);
```

Figure 2: Config 1 Operations

The test scenario for configuration two used a similar code flow and the following operations:

LOAD 1
NOT 1
ONES 1
PASSTHROUGH

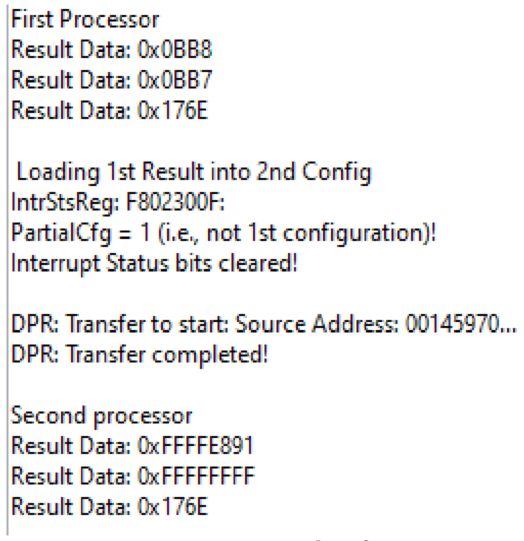


```
CPU_mWriteMemory(baseaddr, savedData); //write data
CPU_mWriteMemory(baseaddr, (0x00070000)); //flip al
CPU_mWriteMemory(baseaddr, (0x00080000)); //send re

// Reading data:

data = CPU_mReadMemory(baseaddr+4);
xil_printf("Result Data: 0x%04x\n", data);

CPU_mWriteMemory(baseaddr, savedData); //write data
CPU_mWriteMemory(baseaddr, 0x00050000); //convert to
CPU_mWriteMemory(baseaddr, 0x00080000 | savedData);
CPU_mWriteMemory(baseaddr, 0x00060000); //send data
CPU_mWriteMemory(baseaddr, 0x00080000); //write data

data = CPU_mReadMemory(baseaddr+4);
xil_printf("Result Data: 0x%04x\n", data);

data = CPU_mReadMemory(baseaddr+4);
xil_printf("Result Data: 0x%04x\n", data);
```

Figure 3: Config 2 Operations

5. Results

The results were verified through the serial monitor at every step and provided the correct results for the operations used. The operations were detailed in the Setup section and the results can be seen below



```
First Processor
Result Data: 0x0BB8
Result Data: 0x0BB7
Result Data: 0x176E

 Loading 1st Result into 2nd Config
IntrStsReg: F802300F:
PartialCfg = 1 (i.e., not 1st configuration)!
Interrupt Status bits cleared!

DPR: Transfer to start: Source Address: 00145970...
DPR: Transfer completed!

Second processor
Result Data: 0xFFFFE891
Result Data: 0xFFFFFFFF
Result Data: 0x176E
```

Figure 4: Data results after operations

The operations were verified with the expected values and the serial print out above. The only issues came with handling the transfer from one config to another though the PS and the implementation of accumulator commands. Implementing the accumulation operations would require some additional logic in the FSM and decoder to accommodate a 2 cycle operation. Other than these instructions, the full set was verified

Conclusions

The original goal of creating a reconfigurable simple processor was achieved and it has proven its utility. The ability to switch between instruction sets and architectures can be expanded to more than 2 modules and allow for a wide variety of uses. Future work would include the implementation of the accumulation operations and more complex operations to be done by the CPU. Other work would be the inclusion of a second RP so that two multi-purpose CPUs could interface with each other and have expanded reconfiguration.