**M**ilitary **A**cademy **R**educed instruction set **C**omputer **2 (MARC2)**

# History.

The MARC processor originally started as a VHDL-based processor for use in the military academy’s computer architecture course. It is a single-stage, four-cycle (i.e., fetch, decode, execute, writeback) processor. It was designed with limited functionality and no performance enhancements in order to reinforce the concepts of computer architecture taught in class. The project continued for several years with the only design verification being a VHDL simulator.

Beginning in the spring of 2003, work was begun to modify the instruction set. This was done mostly to avoid problems with too many old solutions in the barracks. After the instruction set was modified, it was decided to implement the design in a FPGA. There were several reasons for doing this. The first is to demonstrate to the cadets that the design works in real life, not just in simulations. The second reason was to eventually incorporate design implementation into the course subject matter. The goal was to reach a state where the final design verification for each cadet would be testing in hardware. This would improve cadet self-confidence, modernize our instruction base, and reinforce the cadets’ sense of accomplishment.

As work continued, COL Ressler joined the project and created an assembler for the architecture. This assembler, the Military Academy RISC Assembler (MARASM) provided not only basic assembly language tools, but also an emulator, and the ability to generate test files for VHDL simulators, and implementation files for a synthesis tool (e.g., Xilinx ISE, Quartus). He also expanded the project’s concept to include a linkage to the Academy’s compiler course. With this expanded vision, the compiler course would eventually target the MARC2. This linkage would allow the cadets to save effort by working with an architecture they are already familiar with and also allow them to see their compiled code execute on a physical machine when completed.

In 2012, work was done to clean-up the VHDL code and return the MARC2 code to a functional state that could be implemented in a DE2 FPGA demonstration board. The code was debugged and updated to compile in Quartus and was tested on the DE2 board using the MARASM assembler to generate test programs (e.g., marasm\_test\_no\_haz.lst). The top-level design file includes signals and registers whose sole purpose is to allow observable I/O. These signals propagate down to the design sub-blocks creating circuits that exist only to facilitate output.

In the paragraphs that follow, I will attempt to document the state of the architecture to date. There is still work to be done to improve the project. The Input/Output interface could be improved and expanded to allow more complex devices. Interrupts and direct stack access (i.e., pop/push ops) could be added. The status register could be expanded to include carry and overflow. Of course, the trade-off to keep in mind when adding items to the architecture is the impact on the cadets. Adding extra features can make the project too difficult for the cadets if care is not taken. For example, the implementation of the MARC2 in an FPGA is designed to be done without additional work by the cadets (except for debugging). Their existing files are simply dropped into an implementation shell and then compiled and downloaded. There is insufficient time in EE375 to teach both the design and implementation of the architecture. The implementation issues are beyond the scope of the course.

# Design

## Top-level Design



Table 1. MARC2 I/O Interface – DE2 Assignments.

|  |  |
| --- | --- |
| reset | Push Button 0 |
| run | Switch 17 |
| clk | Internal Oscillator @ |
| RESET\_LED | LEDG0 |
| SEC\_LED | LEDR17 |
| reg\_select[2] | Switch 2 |
| reg\_select[1] | Switch 1 |
| reg\_select[0] | Switch 0 |

The MARC2 currently uses the DE2 board’s LCD display to provide output. Using reg\_select[2..0], the user can observe the contents of the system’s registers[0..7].

**MARC 2 Architecture Summary**

**MARC2 CPU**

Control unit

Datapath

AR

PC

SP

IR

CTLWD

PSR

CLK

RST

Mem\_rd

Mem\_cs

Mem\_wr

Run

Data Bus

Address Bus

St\_op

Ld\_op

R\_we

Ctl\_wd

Const\_out

Z

N

* eight 16-bit registers, R0 through R7 are located in the datapath. R0 is hard wired to always output all 0’s
* function unit (ALU) performs the operations listed in the truth table below, also in the datapath:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **S3** | **S2** | **S1** | **S0** | RTL | Operation |
| 0 | 0 | 0 | 0 |  | None (nop); Assert high impedance |
| 1 | 0 | 0 | 1 | F 🡨 A – B | A minus B |
| 1 | 0 | 0 | 0 | F 🡨 A or B | A OR B |
| 0 | 1 | 1 | 1 | F  | Pass B |
| 0 | 1 | 1 | 0 | F  | A plus B |
| 0 | 1 | 0 | 1 | F  | A AND B |
| 0 | 1 | 0 | 0 | F  | Complement A |
| 0 | 0 | 1 | 1 | F  sr A | Logical shift right of A |
| 0 | 0 | 1 | 0 | F  sl A | Logical shift left of A |
| 0 | 0 | 0 | 1 | F  | Pass A |

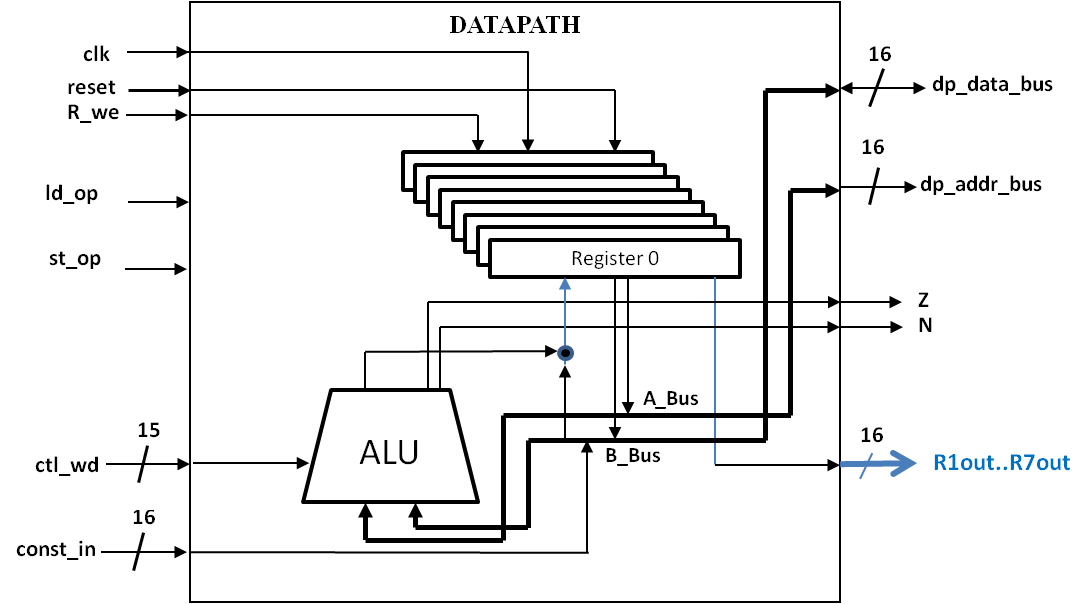
* Operations on or loading immediate/constant values is possible via the **const\_in** signal which is routed into the “B” input of the function unit. The datapath supplies two flags (**Z** and **N**) to indicate the status of the current function unit operation. The Z-flag indicates if the result of a function unit operations is all 0’s (**Z**=1) while the N-flag indicates if the sign (MSb) of the function unit output is negative (**N**=1).
* The various control and selection inputs to the datapath are in the control word (the **ctl\_wd** input) with the following form:

MD MB SelD FS SelA SelB

14 13 12 10 9 6 5 3 2 0

**SelA** and **SelB** select which register places data on the A and B buses out of the register file. **SelD** selects the destination register. **FS** determines the operation of the function unit (Function Unit Select) as listed in the truth table above. **MD** selects whether the output of the function unit or a data word from memory goes to the destination register. **MB** selects either the B bus or the **const\_in** input is one of the inputs to the function unit (the other input is the data on the A bus).

* Other inputs to the datapath include the clock (**clk**), the register file write enable (**R\_we**), and the memory load/store operation flags (**ld\_op** and **st\_op**). The registers in the register file can only be written to when **R\_we** = 1. The **ld\_op** and **st\_op** inputs are used to control the 3-state memory address (**dp\_addr\_bus**) and data (**dp\_data\_bus**) buses connected to the datapath. The datapath should put the data (e.g., an address stored in a register) from the A bus on to the **dp\_addr\_bus** when either **ld\_op** or **st\_op = 1** and the data from the B Bus on to the **dp\_data\_bus** when **st\_op** = 1. When **ld\_op** and **st\_op** = 0, the datapath should put its **dp\_addr\_bus** and **dp\_data\_bus** outputs into the **high Z** state (i.e., high impedance) so other device besides the datapath to put data onto them. A schematic showing the inputs and outputs of the datapath, as described in is given below.



* Upon system reset, the **PC** register is initialized to 0x0080 and the **SP** register is initialized to 0x4FE. The contents of all the datapath registers are initialized to zero.

The MARC2 processor has a load/store architecture and a simple instruction set. There are a total of 19 different instructions in three formats as shown below:

**Format 0**: F RD OP RS1 RS2 N/A **Format 1**: F RD OP imm/disp

(F = 00) (F = 01) 15 13 10 6 3 0 15 13 10 7 0

**OP** **OP**

1010 **nop:** null

1001 **subx:** RD 🡨 RS1 – RS2 000 **hlt**: state ← reset

1000 **orx:**  RD 🡨 RS1 or RS2 001 **ret**: SP ← SP+1\*, PC ← M[SP]

0111 **jmp**: PC ← RS2 011 **addi**: RD ← RD + se imm

0110 **addx**: RD ← RS1 + RS2 100 **ba**: PC ← PC + se disp

0101 **andx**: RD ← RS1 and RS2 101 **bn**: N: PC ← PC + se disp

0100 **notx**: RD ← RS1 110 **bz**: Z: PC ← PC + se disp

0011 **srlx**: RD ← sr RS1 111 **sethi**: RD ← imm || 00000000

0010 **sllx**: RD ← sl RS1

0001 **ld**: RD ← M[RS1]

0000 **st**: M[RS1] ← RS2

**Format 2**: F AF

(F = 1x) **call**: PC ← 0 || AF, M[SP] ← PC, SP ← SP-1

1. 0

**Legend**: \* for these instructions, the SP (memory address) must be incremented before the memory read operation is done

M[reg] – memory word at address stored in reg (either RS1 or SP), || - concatenation, se – sign extended

Note that all the instructions above are 16 bits long and thus are stored in 2 bytes of memory. The instructions are not byte addressable. Because the MARC2 processor will be able to address 216 bytes (64 KB) of memory, it must have a 16-bit Program Counter register (**PC**) and Address register (**AR**). Despite this ability, the processor only has 1 KB of ROM (address 0x0000-0x03FF) and 2 KB of RAM (address 0x400-0xBFF). This can be modified somewhat, but is limited by the FPGA device used in implementation. The first 128 memory locations in low memory (addresses 000016 – 007F16) are reserved for the system. To de-conflict system and user memory space, all user programs will start at address 008016 and all code must be loaded into ROM. All memory transfers will occur on the system address and data buses. This means that micro-operations involving memory listed above like **PC ←** **M[SP]** actually require two steps :

**CU\_addr\_bus ← SP**

**PC ←**  **CU\_data\_bus**

The MARC2 has a memory stack that starts at high memory and builds downward. The stack is used to store subroutine return instruction addresses. The **SP** in the MARC2 points to the next available memory location in the stack. This means that for the **call** instruction, the stack pointer is decremented after the return address (the **PC** value) is pushed onto the stack. However, for the **ret** instruction, the **SP** must be incremented before the return address can be popped off of the stack (and loaded into the **PC**). To accomplish this, the **SP** manipulation for the **ret** instruction must be done in the decode state so that the **PC** can be loaded during the execute state. Target addresses for **call** instructions are calculated as depicted RTL statement of the table above.

The block diagram for the control unit is given below.

**const\_out**

**Mem\_wr**

##### Mem\_rd

**Z**

**N**

### Control

### Unit

**clk**

**ld**\_**op**

**st\_op**

**CU\_data\_bus**

**16**

**CU\_addr\_bus**

**16**

**Mem\_cs**

**R\_we**

**16**

**ctl\_wd**

14

**run**

**rst**

The memory for the system is simulated in the testbench. In the FPGA, they are created using the design tools in the Quartus package. (NOTE: The **Mem\_rd**, **Mem\_wr** and **Mem\_cs** are all active low.) The **CU\_data\_bus** and **CU\_addr\_bus** represent the control unit’s connections to the system’s memory buses.

The **R\_we**, **ld\_op**, **st\_op**, and **ctl\_wd** control signals go to the datapath. The **ld\_op** and **st\_op** signals must be asserted during the execution of load and store operations respectively so that the datapath will take its connections to the memory buses out of the high Z state and put the appropriate information on them.

Inputs into the control unit include the **Z** and **N** signals from the datapath which are used for conditional branch instructions. The **clk** signal comes from the system clock, which for simulation purposes will be generated in the testbench. The **rst** signal (active low) is an external input that, in reality, would come from the system power or reset switch, but for simulation purposes, will be generated in the testbench. The **rst** is an asynchronous reset which forces the processor into the reset state (where the **PC** points to address 008016 (where the first instruction in the program is) and the stack pointer (**SP**) points to high memory (04FE16). The **run** signal is another external input that is used to signal the execution of a program. If **run** is ‘1’, then the processor remains in the reset state. If **run** is ‘0’, the system goes to the fetch state and begins the **F / D / E / WB** computer operation cycle.

# Implementation

There are currently multiple versions of the MARC2 architecture of which some have been implemented in an FPGA. Each of these versions has the same basic processor architecture. The only differences are in the top-level design file that implements the processor.

The EE375 version contains the un-implemented code for use in the course EE375. It contains all the source code plus project problem statements, shell code files (for students), and shell test benches (for students). This code is intended for use during the course to guide the students through the design and synthesis of the MARC2. They start by creating and testing the datapath. Next they create and test the control unit. Lastly, the combine the control unit and datapath to create their processor. They test their processor by using the MARASM software to write an assembly language test program. The assembler generates a text file which is used by the VHDL simulator to test the design. For more details on the MARASM, reference the user’s manual.

The Spartan version of the processor takes the design one step further and implements the EE375 version in a FPGA. The code uses Xilinx ISE 6.2i software to target a Xilinx Spartan XC2S200E, PQ208 FPGA contained on the Digilab IIE system board made by Digilent. This board has two Digilab DIO1 boards (Input/Output boards) connected to it on connectors A and C. The I/O boards provide user inputs in the form of switches and pushbuttons. It also provides a bank of 7-segment displays and some LEDs for output. A break down of the board’s operation is as follows:

Function Signal Location

Display refresh clock – displayclk => IIE on board oscillator

System clock- clk => BTN1 DIO1 (connector A)

System reset- reset => BTN2 DIO1 (connector A)

Run- run => SW1 DIO1 (connector A)

Datapath register select- Dregsel => SW6-SW8 DIO1 (connector A)

Output databus or register- DataOrReg => SW2 DIO1 (connector A)

System Address bus output- ASeg0 => 7-segment displays DIO1 (connector C)

Register or databus output - DSeg0 => 7-segment displays DIO1 (connector A)

The actual pin assignments can be found in the design user constraints file (ucf). The design is implemented with a top level file that instantiates the processor, a 2 KB Single Port Block RAM (address 0x400-0xBFF), and a 1 KB Single Port Block RAM in ROM mode(address 0-0x3FF). The Block RAM modules were created using the Coregen tool in the ISE package.

The idea of this version is that the cadet can ‘drop’ their processor design into the top-level file and then implement it in the hardware board. The program they write to test the design is written in assembly language and assembled by the MARASM software. This software generates the coefficients file (.coe) necessary to initialize the Block ROM. To load a student’s program into the processor, all that is needed is to regenerate the Block ROM with the new .coe file. Once implemented and downloaded to the FPGA, the user can step through the execution of their program in hardware. The value of the address bus is constantly displayed on connector C. The data bus value (thus the instruction fetch from memory and the values stored to memory) or the register contents can be viewed on the displays on Connector A via multiplexing. The cadet can control execution by using the pushbutton as the system clock. This isn’t very elegant, but it accomplishes the goal of seeing the project work in hardware.

The Input/Output (I/O) version of the processor is focused on creating a processor that is suitable for targeting in the academy’s compiler course. The basic MARC2 design does not incorporate any user I/O. Therefore the programs that can be executed on it are somewhat limited or at least uninteresting. To add some more power to the design, addressable I/O was added to the system.

To accomplish this task, the same prototype board and I/O boards were used for the implementation. Some of them physical connections were changed as shown below:

Function Signal Location

Display refresh clock – displayclk => IIE on board oscillator

System clock- clk => BTN1 DIO1 (connector A)

System reset- reset => BTN2 DIO1 (connector A)

Run- run => BTN3 DIO1 (connector A)

Input data indicator- Load\_LED => LED1 DIO1 (connector A)

LSByte input- in\_data\_low => SW1-SW8 (connector A)

MSByte input- in\_data\_high => SW1-SW8 (connector C)

User Output- DSeg0 => 7-segment displays (connector A)

The user input is accomplished via the 16 slide switches on the IO boards. The program obtains the input by executing a load instruction at address 0x4000. When the instruction executes and the address bus achieves the value 0x4000, the Load\_LED becomes active, signaling the user to set the switches. The user output consists of 4 seven segment displays on connector A. These are currently configured to output values as hex numbers. This could easily be changed to allow the programmer to control individual segments on the displays, thus allowing a limited graphical display capability. To write the displays, the program must execute a store instruction to address 0x2000.

The architecture consumes less than 25 percent of the Spartan’s resources. Therefore, there is a lot of room for expansion and modification to the design. The only resource that cannot be greatly modified is the Block RAM. Eight-five percent (12 out of 14 blocks) are used in the current versions. The RAM/ROM can be altered but not expanded too much. Though system speed was never a design criteria, the estimated maximum clock frequency for the design is 30 MHz. This is assuming that the push button system clock was replaced with an oscillator.

The current version, MARC2\_DE2 contains the code that synthesizes and is capable of being implemented on the DE2 FPGA demonstration board. This processor has been tested using two different programs, each of which execute every opcode at least once. The system has not been fully tested to ensure it is fully operational. It is synthesized for the Altera Cyclone IVE, EP4CE115F29C7 FPGA.