# WIMP AVR
# Technical User Guide

## Contents (Click to Jump to Section)

# Design Methodology

The WIMPAVR processor instruction, as shown in Figure 1, is a subset of the overall AVR instruction set [12]. The choice of instructions was kept similar to the WIMP51 instruction set [1] and has a complement of data transfer, logical, arithmetic and branch instructions sufficient for a mini-processor operation. The architecture of the microprocessor was designed based on the block diagram in Figure 2.

| Instruction | Machine Code | Description |
|---|---|---|
| LDI | 1110 kkkk dddd kkkk | Load immediate binary data kkkk kkkk to lower register bank 1dddd (16 - 31) |
| MOV | 0010 11rd dddd rrrr | Moves data from register rrrrr to register ddddd |
| ADC | 0001 11rd dddd rrrr | Adds data in register ddddd to register rrrrr and stores it in register ddddd |
| AND | 0010 00rd dddd rrrr | AND register rrrrr and register ddddd, stores result in register ddddd |
| OR | 0010 10rd dddd rrrr | OR register rrrrr and register ddddd, stores result in register ddddd |
| EOR | 0010 01rd dddd rrrr | XOR register rrrrr and register ddddd, stores result in register ddddd |
| SWAP | 1001 010d dddd 0010 | SWAP lower and upper nibble in register ddddd, stores result in register ddddd |
| RJMP | 1100 kkkk kkkk kkkk | Jump program counter PC = PC + kkkk kkkk kkkk + 1 |
| BRBS (Z) | 1111 00kk kkkk k001 | Jump program counter PC = PC + kk kkkk k + 1 if the Z flag is set |
| SEC | 1001 0100 0000 1000 | Set carry C = 1 |
| CLC | 1001 0100 1000 1000 | Clear carry C = 0 |

Figure 1: WIMPAVR instruction set. The instruction set is similar to the WIMP51 instruction set. But, each instruction is a 16-bit instruction. The machine codes are from the actual AVR microprocessor instruction set.

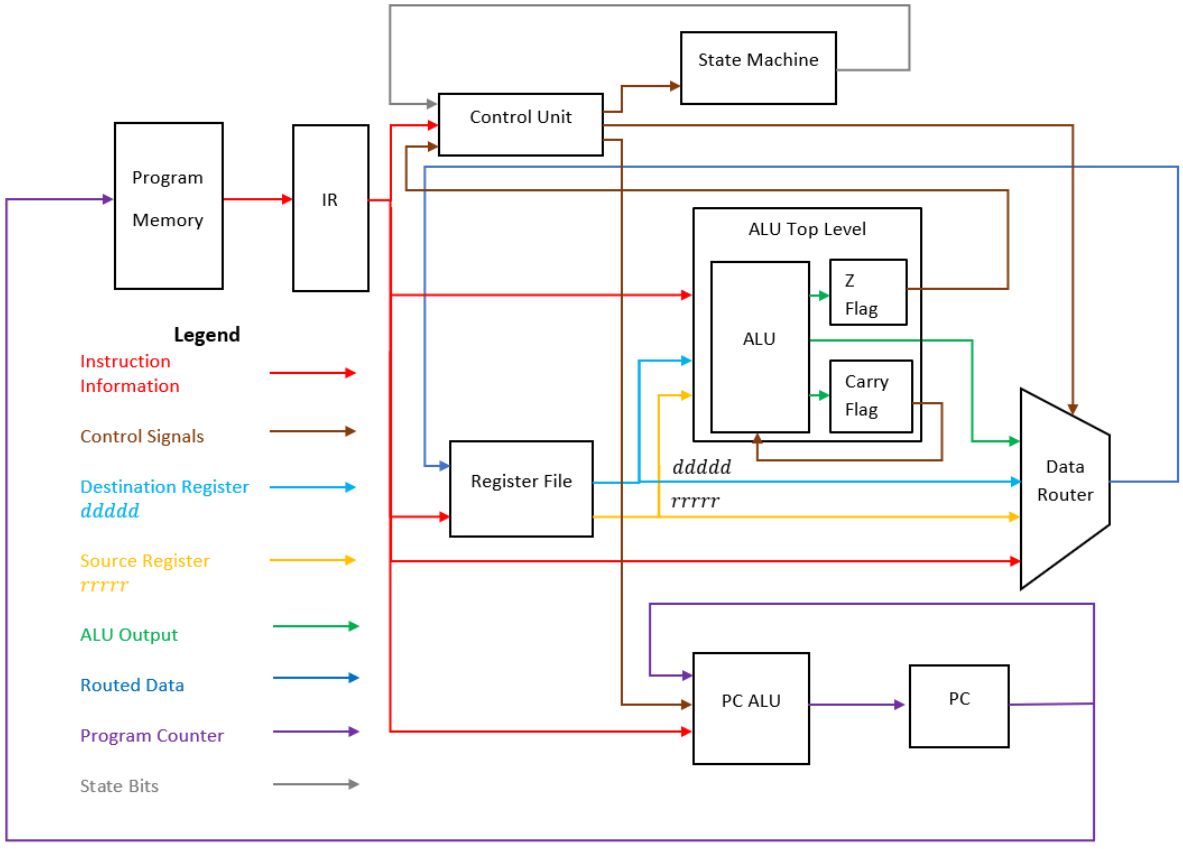

Figure 2: WIMPAVR block diagram. Note, the enable signals are grouped as control signals.

The Program Memory (PM), as seen in Figure 1, is technically not part of the processor and represents the code memory, which is implemented using the available FPGA onboard SRAM chip. Different data paths, as seen in Figure 1, are color coded to ease understanding of data flow. The processors contains two ALUs, one to manipulate data and another to control the Program Counter (PC). The Data Router (DR) allows the routing of 8-bit data from the source to the destination depending on the executed instruction. The State Machine (SM) and the Control Unit (CU) work together to generate the different enable signals required to control the processor's operation. Each unit, in the block diagram, is composed of subsystems of digital logic circuits that process inputs depending on the executed instruction.

The layout of the top-level components was kept close to the layout seen in the block diagram, Figure 1, which would in-turn ease system visualization. Since schematic capture uses Block Diagram Files (BDF), students can easily access the next lower-level logic for each block by double-clicking top-level entity blocks. This feature allows then to easily understand how each top-level entity complex block is made up for sub-block all the way down to gate-level logic. An example of the multi-layered system block can be seen in Figure 3 for the DR block.

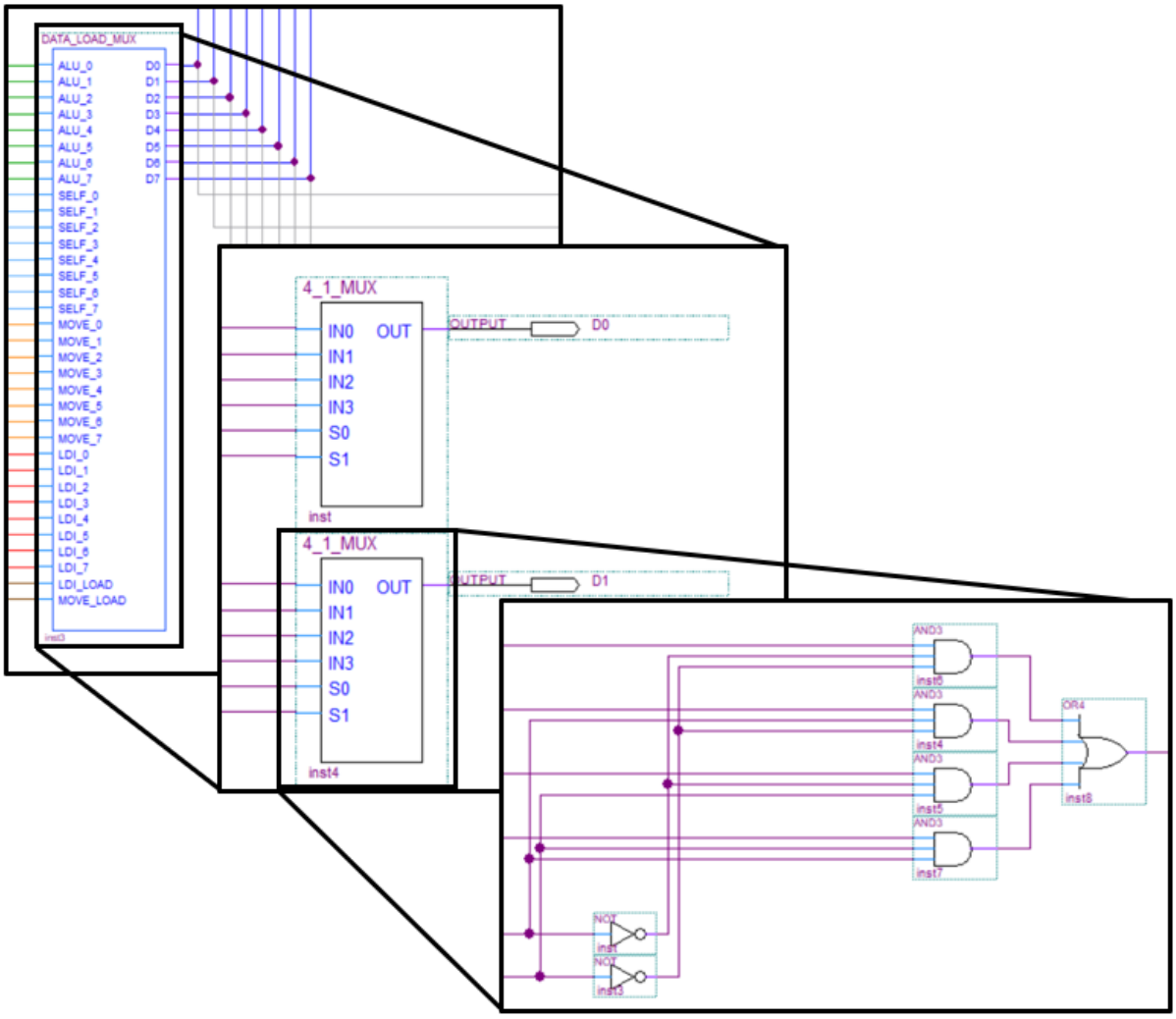


Figure 3. Multi-layered DR block

The top level DR block is composed of 16 4:1 multiplexers to multiplex four different 16-bit values. Each of these 4:1 multiplexers is a subsystem which can be opened to observe the digital logic circuit used to create the 4:1 multiplexer.

This facility, also, allows students to visualize that a simple block can be used in multiple locations on a higher level. For example, one key sub-system, in the design of the WIMPAVR, is the self-loop block. The self-loop block allows a 1-bit memory element to either hold the current memory content or allow new information to enter the memory element, and is based on a 2:1 multiplexer. This 1-bit memory element is used to implement multiple components such 8-bit internal data registers, 16-bit Instruction Register (IR) and Program Counter (PC) and the different flags seen in Figure 2.

# Processor Design

The AVR processor is a two stage pipeline [5] which means that the processor has two basic states: build and fetch/execute. When the processor is turned on, it is in the build state, which allows for the pipeline to be created. As long as the IR does not contain a branch instruction, the processor stays in the fetch/execute state. The execution of each instruction takes exactly one clock cycle. During the fetch/execute state, the processor must make a decision whether to stay in the fetch/execute state or rebuild the pipeline. If the IR contains a branch instruction and if the branch condition, if any, is not satisfied, the processor stays in the fetch/execute state. If the branch condition is satisfied, the processor moves back to the build state to rebuild the pipeline, in which it fetches an instruction from the branch location and incurs a branch penalty, which in turn is for one clock cycle.

## *State Machine*

The AVR processor has two basic states. Therefore, only one state bit is needed for the state machine. In order to facilitate the processor creation and simplify its understanding, two additional states were incorporated. Of these four states, shown below, only three states are part of the process operation.

000 – Idle
001 – Build from Power on
011 – Fetch/Execute
101 – Build from Branch

Three state bits, to represent 4 states, were chosen to ease the design of control signals.

- ***Idle State (000)***: When the processor is first turned on, the state machine will start in the idle state. In this state, processor is off and will not read or operate any instructions. The state machine will continue to be in the idle state until the processor is turned on. The processor is turned on by turning on the master switch located on the FPGA board. When the master switch is turned on, the state machine will remain in the idle state and only move to the build state at the next clock edge. The master switch allows the user to reset the processor and return to the idle state at any point of its operation.
- ***Build State (001)***: The state machine can only transition to the build state after the master switch SW17 has been turned on in the idle state for one clock cycle. In the build state, the AVR processor fetches the instruction, from the program memory, at the PC address, into the IR. Initially (transition from idle state), the PC will be at zero. No instructions are executed during this step. The PC increments by one at the end of the build state. After this step, the state machine will move to the fetch/execute state, unless the master switch is turned off, then the state machine will return to the idle state. The state machine will never be in the build state for consecutive clock cycles.
- ***Fetch/Execute (011)***: The state machine will transition to the Fetch/Execute state after the Build state as long as the master switch remains on. In the Fetch/Execute state, the instruction, loaded in the IR, is executed. The CU generates the necessary control signals to allow the data path components to execute the current instruction. In addition, the processor will fetch the next instruction at the updated PC address from the previous state. Depending on the type of the instruction, the next state of the processor will change. Consider the timing diagram shown in Figure 4, which shows the operation as long as the branch instructions, as seen in Figure 1, do not show up.

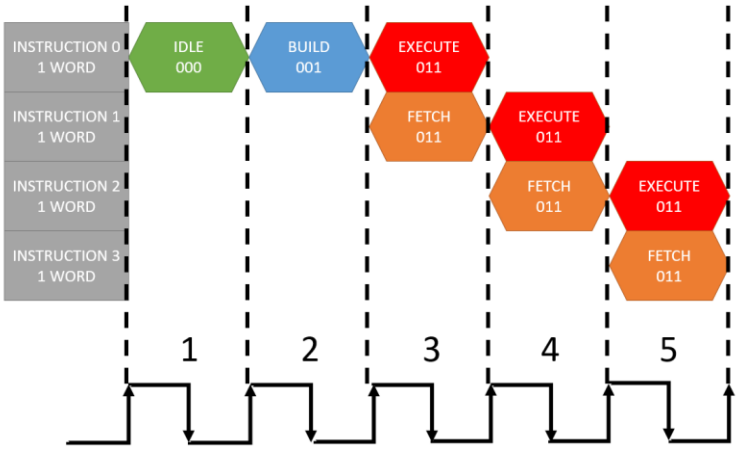


Figure 4: Timing diagram for a sub-program, which does not include branch instruction

As seen in Figure 4, at the end of the third clock cycle, instruction 0 has been executed (all necessary registers are updated), and the instruction 1 has shown up in IR. The processor stays in the Fetch/Execute state as long as the current and subsequent instructions are not branch instructions. Now consider a typical timing diagram shown in Figure 5.

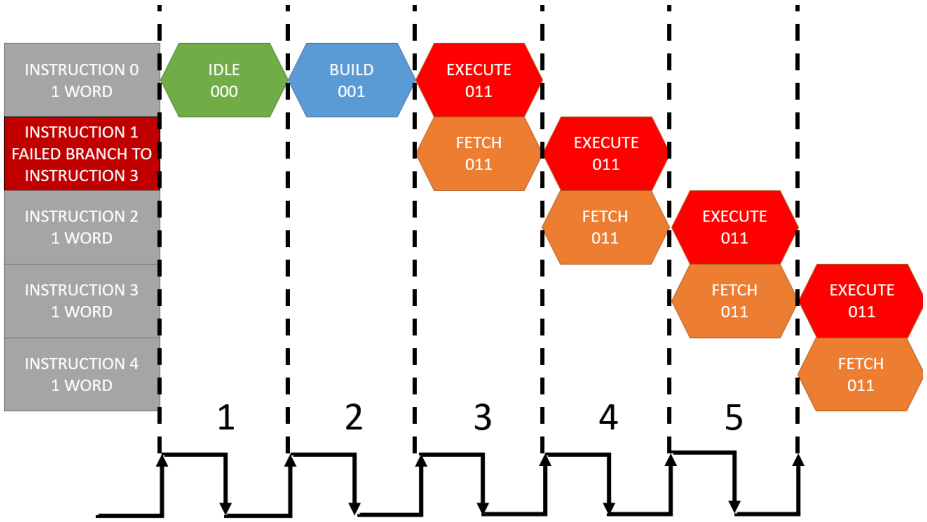


Figure 5: Timing diagram for a sub-program, which includes a branch instruction. But the branch condition, if any, is unsatisfied.

At the end of the third clock cycle, as before, instruction 0 has been executed and instruction 1, which is now a conditional branch instruction, has entered the IR. The processor stays in the Fetch/Execute state in the fourth clock cycle. Since the condition is not satisfied, which is the case here, at the end of the fourth clock cycle, appropriate updates are made and instruction 2 has entered IR. If the master switch is off and a reset button, on the FPGA board, is pressed, the processor will return to the idle state. In general, the processor will return to Fetch/Execute, if it not already in this state, and will stay in the Fetch/Execute state for consecutive clock cycles as long as no branch instructions are successfully executed and the master switch is on. This facility allows the processor to achieve maximum throughput by executing 1 instruction per clock cycle.

- ***Branch State (101)***: Consider the timing diagram shown in Figure 6.

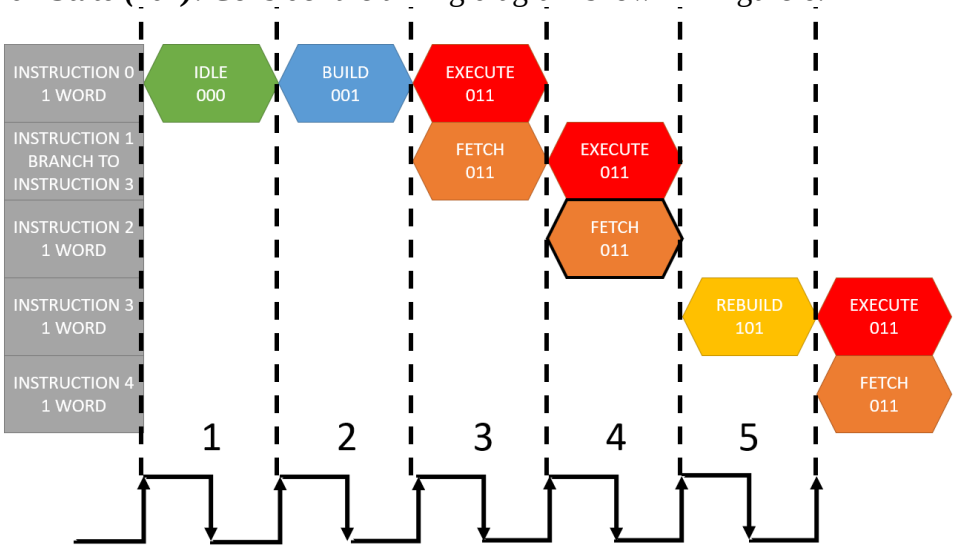


Figure 6: Timing diagram for a sub-program, which includes a branch instruction. But the branch condition, if any, is satisfied.

As before, at the end of the third clock cycle, instruction 0 has been executed and instruction 1, which is a branch instruction, has entered IR. The processor stays in the Fetch/Execute during the fourth clock cycle, during which the processor detects that the branch instruction condition is satisfied. The PC during this clock cycle is now pointing at instruction 2, which is the incorrect instruction. The conditional branch instruction points to instruction 3 as the next instruction. The processor will now enter the Branch state (101, also called Rebuild state as seen in Figure 6) during the fifth clock cycle. PC is updated to point to instruction 3, which enters the IR at the end of the fifth clock cycle. The Branch State operates exactly the same as the Build state. The processor will always return to the Fetch/Execute state after the Rebuild state. It is impossible to remain in the branch state for consecutive clock cycles. Figure 6, also, demonstrates the concept of branch penalty encountered in pipelined systems, and for WIMPAVR processor's 2 stage pipeline the branch penalty is one clock cycle.

The above discussed conceptual timing diagrams are used to explain the expected processor timing analysis for different types of instructions in the WIMPAVR instruction set.

## Control Unit

The CU looks at the instructions in the IR and decides what type of instruction is being executed. There are five main types of instructions based on how the register file responds to the instruction: ALU, Move, Load, Branch, and Status instructions.

### *Reset*

An external push button reset switch is included on the AVR processor. The reset is an active low input and can only be activated if the master switch is off. The reset causes all registers to reset to 0 the state machine to return to the idle state.

### *Program Counter and Instruction Register*

The PC is a 16-bit register that stores the address of the next instruction to be executed. The IR is a 16-bit register that stores the current instruction to be executed during Fetch/Execute. The update of the PC and IR was discussed above in the timing diagrams shown in Figures 4-6.

### *PC ALU and the branch instructions*

The value of the next PC address is controlled by the PC ALU. The PC ALU is enabled only when the AVR processor is not in the idle state. Figure 7 shows the block diagram for the PC ALU.

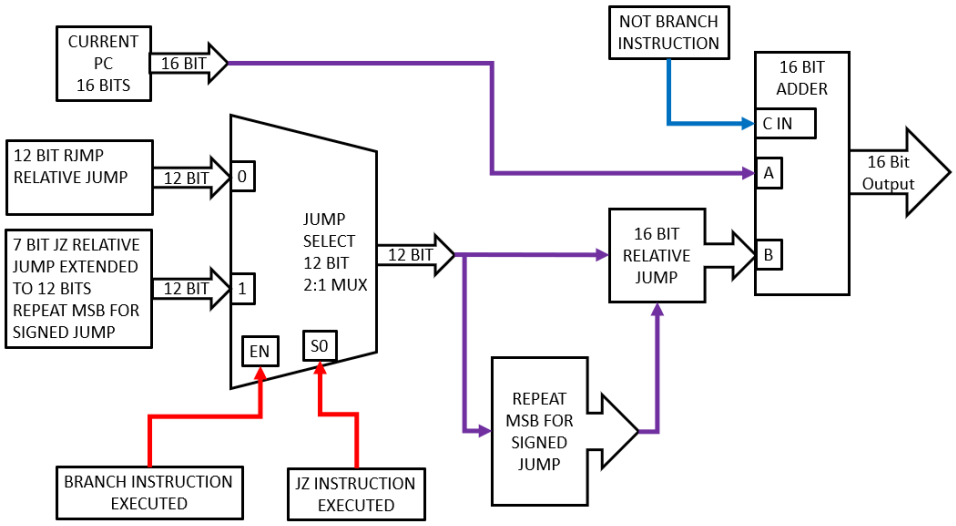


Figure 7: PC ALU block diagram.

The PC ALU is used to manipulate the PC for non-branch instructions, which require the PC to be incremented each clock cycle and for two unique branch instructions: RJMP, which allows for a 12-bit relative and unconditional jump, and BRBS(Z), which allows for a 7-bit relative and conditional (if Z flag is set) jump. The PC ALU is composed of a 16-bit ripple adder and a Jump Selection Block (JSB) as seen in Figure 7. The 16-bit ripple adder is used to add the current PC address with the data from the JSB to calculate the next PC address. The JSB is a 12-bit 2:1 multiplexer with an enable to route the necessary jump data. The enable (EN) of the JSB is controlled by the branch signal, which is generated by the CU, to determine if a branch instruction is decoded. If a branch instruction is not decoded, the output of the JSB is zeros. The 16-bit adder increments the PC, using carry in (C_IN) if the branch instruction is not decoded.

## *Data Router (DR) Multiplexer*

Figure 8 shows the DR block diagram. The microprocessor accesses and moves data from three different locations: the ALU data, immediate data from the instruction, and data in the registers.

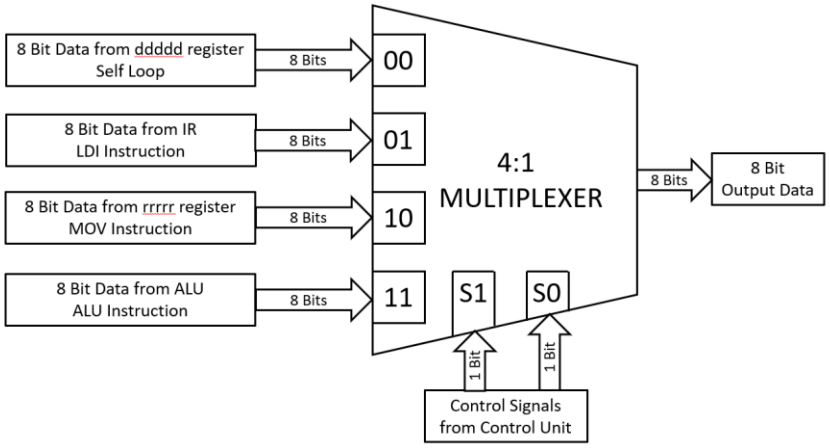


Figure 8: The DR block, which is an 8-bit 4:1 MUX

The destination is always one of the “ddddd” registers defined in the opcode mentioned in Figure 1. One notable difference, as seen in the opcodes mentioned in Figure 1, is that LDI allows data to be only written into 16 registers, where as other instructions can write data into 32 registers. The self-loop path, seen in Figure 8, is the default path used when the current instruction is neither LDI, MOV nor an ALU instruction.

## *ALU Instructions*

Figure 9 shows the implemented ALU top-level block diagram. The ALU operates on two operands: rrrrr and ddddd, which are 32 8-bit registers, and forms the internal RAM. After an ALU instruction, the destination register defined by the five bit sequence ddddd is updated with the output of the ALU. ALU instructions can be further divided into logic & swap instructions, and arithmetic instructions. An 8-bit 2:1 MUX, as seen in Figure 9, routes the ALU result, depending on an ALU instruction, to the DR block, as seen in Figure 2. An internal MUX, in the logic block, routes the correct logical instruction result to the main 2:1 ALU MUX as seen in Figure 9. The Zero (Z) flag status bit is required for the correct operation of the BRBS(Z) branch instruction. Figure 10 shows the block diagram of the Z flag update sub-system. The Z flag is updated only after an add (ADC) instruction is executed and can work with any of 32 registers involved in the add instruction.

### *Status instructions*

The Carry (C) flag, which is the other status bit can be updated using the set and clear carry flag instructions, as seen in Figure 1, or after the add instruction. Figure 11 shows the carry flag update sub-system block diagram.

## *Move and Load Instructions*

Move instructions do not use the ALU, and move data from a source register defined by “rrrrr” to a destination register “ddddd”. The WIMPAVR has only one type of MOV instruction as seen in Figure 1. Load instructions store immediate data, located in the instruction itself, into a destination register “ddddd” without using the ALU or a source register. The WIMP AVR has only one type of load instruction that can only load data to registers 16-31.

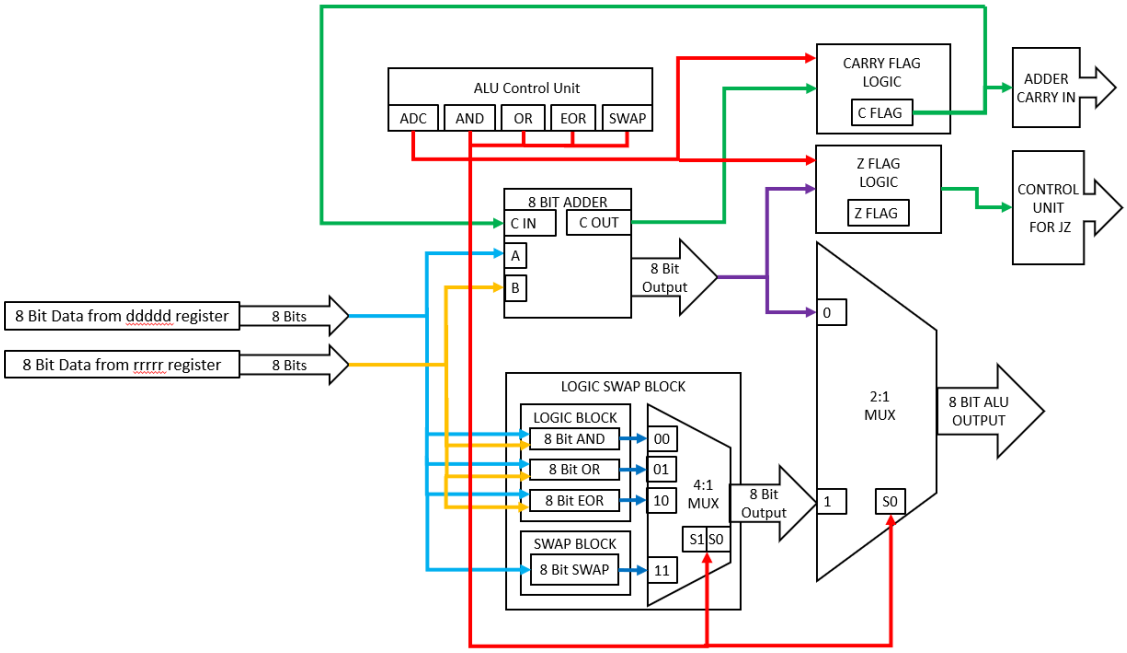


Figure 9: ALU block diagram.

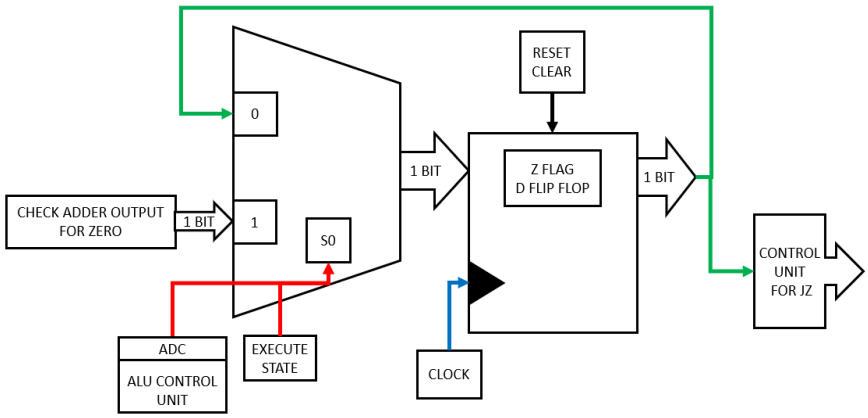


Figure 10: The Z Flag sub-system.

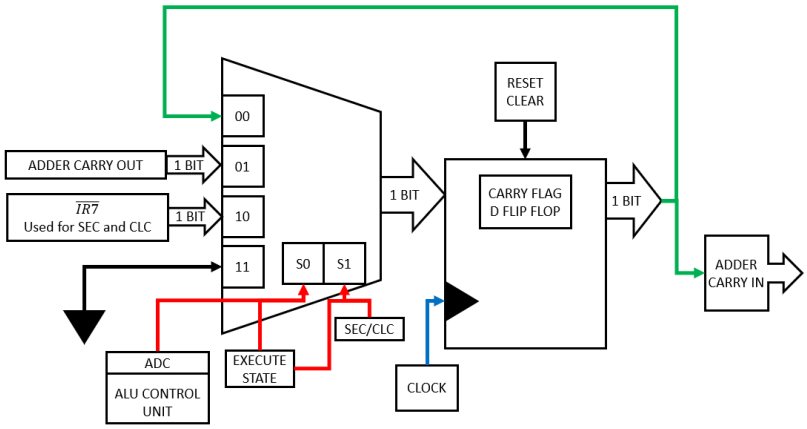


Figure 11: The C flag update sub-system.

These instructions operate on a designed 32×8 register file. All data is stored in the register file, and ALU operations can only be performed on data stored in this register file. Figure 12 shows the block diagram of the created register file.

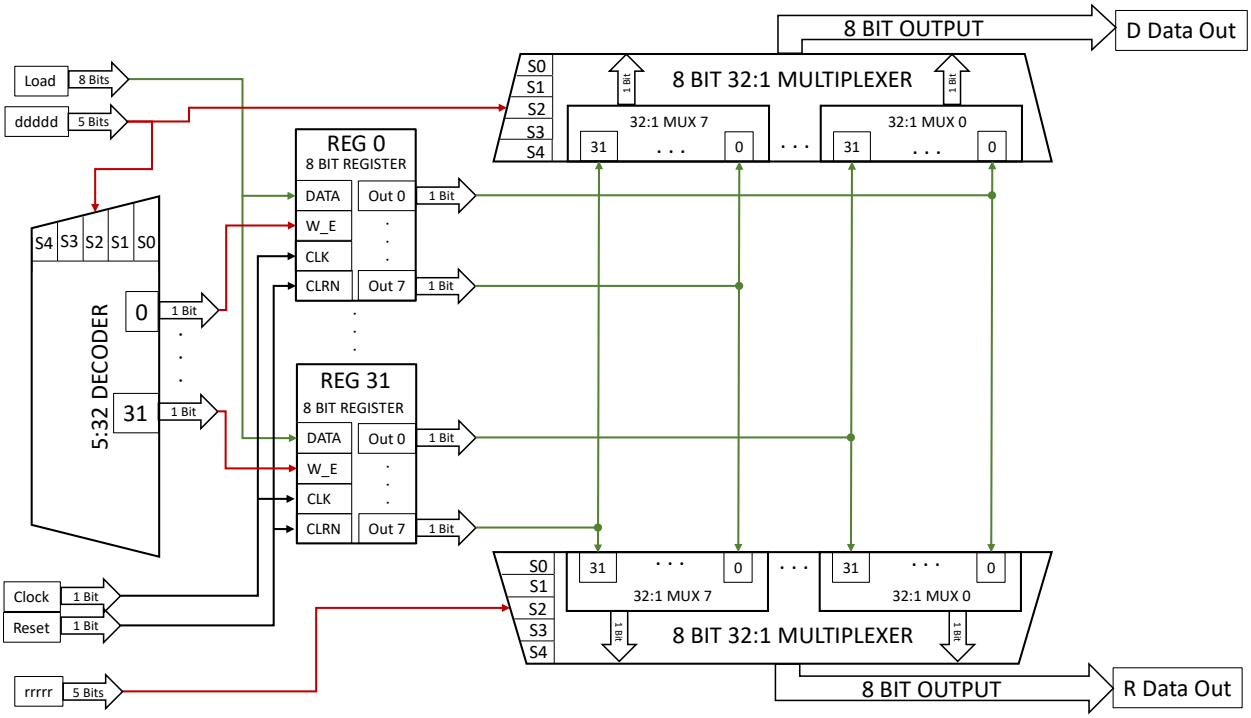


Figure 12: The 32×8 register file block diagram.

The destination address “ddddd” allows the program to write to one of the 32 registers. The two 8-bit 32:1 MUX, shown in Figure 12, allows the processor to access data from two registers at a time, using two 5-bit codes: one to identify the source register rrrrr and another to identify the destination register ddddd. A designer can modify the block by adding more MUX blocks to access multiple register as required. This facility allows a designer to modify the processor relatively easily. Non-ALU and data transfer instructions cause the registers to stay in a hold state by using the self-loop feature incorporated for all registers.

# Additional Design Features

## *Processor Operation Features*

In order to make WIMPAVR an effective teaching tool, additional design components, not shown in Figure 2, were created that provides the user access to the internal signals of the processor. Figure 13 shows the register access and the display features created for WIMPAVR.

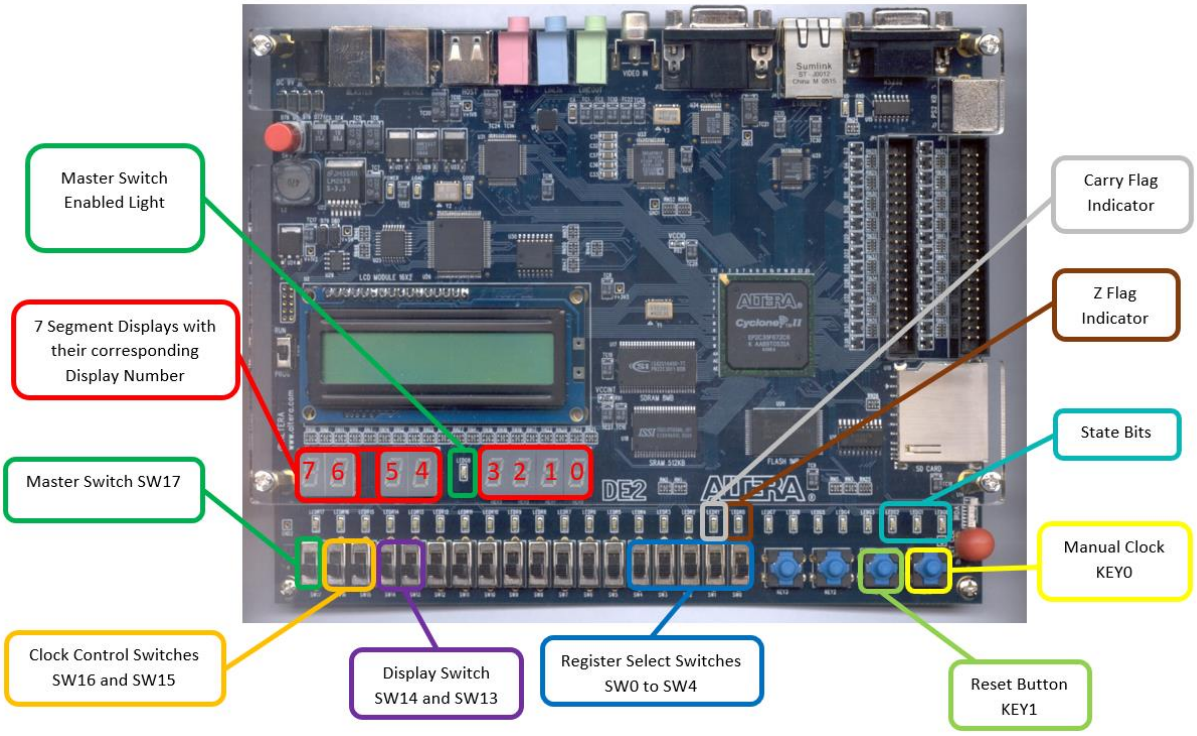


Figure 13: Display features available during WIMPAVR operation

- Seven-Segment Display (SSD) controls: The Altera (now Intel) FPGA board is limited to 8 SSDs as seen in Figure 13. A SSD can display one hexadecimal number which represents four binary bits. The SSDs are used to display key register values within the WIMPAVR which include the PC, IR, Next Instruction, Registered Data, and Output Result. Since there are a limited number of SSDs available on the FPGA board and the user needs access to multiple registers with varied sizes (8 and 16 bits), switches on the FPGA board were used to multiplex register values displayed on the SSDs. Following is a list of multiplexed displays:
    - Switch (SW) 14 down (off): SSDs 3-0 displays the current 16-bit instruction in the IR
    - SW14 up (on): SSDs 3-0 displays the next 16-bit instruction in the program memory accessed using the PC. This feature allows the user to observe pipelined instructions, which is an important AVR microprocessor core facility.
    - SW13 down: SSDs 7-4 displays the 16-bit PC
    - SW13 up: SSDs 5-4 displays the DR block output
    - SW13 up: SSDs 7-6 displays the values of one of the 32 8-bit registers, which are multiplexed using switches SW4-SW0

- Clock control system. This system is used to control the clock speed of the microprocessor. One of the FPGA board's in-built clock sources, 50 MHz clock speed, was slowed down significantly to allow humans to observe the operation of the microprocessor. Figure 14 shows the block diagram of the clock control sub-system.

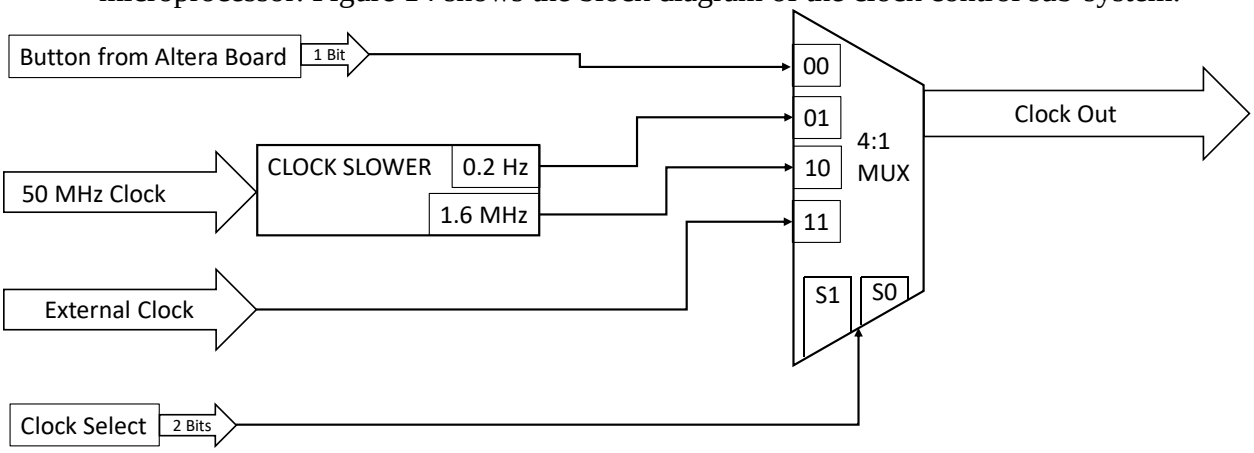


Figure 14: Clock control sub-system. Note the user is allowed to manually control the processor operation using a push-button switch.

The speed of the processor can be slowed to 0.2 Hz so that users can observe the operation of the microprocessor with minimal human intervention. An option to run the processor at 1.6 MHz is also provided to show that the processor can work at higher clock speeds. The microprocessor can also be set to manual mode, using push-button KEY0, on FPGA board, to act as a clock edge. This facility allows the user to take as much time as necessary to observe the processor signals for each state. The clock of the WIMPAVR can be controlled using the clock control switch SW16 and SW15 as seen in Figure 13.

- Master switch and Reset: SW17 is used to either run the processor (on position) or pause the processor (off position). In the pause condition, the processor can be reset by using push-button KEY1 as seen in Figure 13. Note: the reset does not erase the program memory.
- LED data: LEDs, as seen in Figure 13, are used to display the Z and C flags, the processor state and master switch enable indicator.

## ***Processor Programming Features:***

The WIMPAVR processor executes a user-entered program stored in the program memory, which is implemented on the FPGA on-board SRAM chip as shown in Figure 15. The user is required to write the machine coded program directly into the program memory manually. Even though this process is cumbersome and time-consuming, the experience can be invaluable. A separate program was created to allow users to write machine coded instructions into the SRAM. In order to assist users, and minimize errors in entering the machine code, unique display features, as seen in Figure 15, were incorporated.

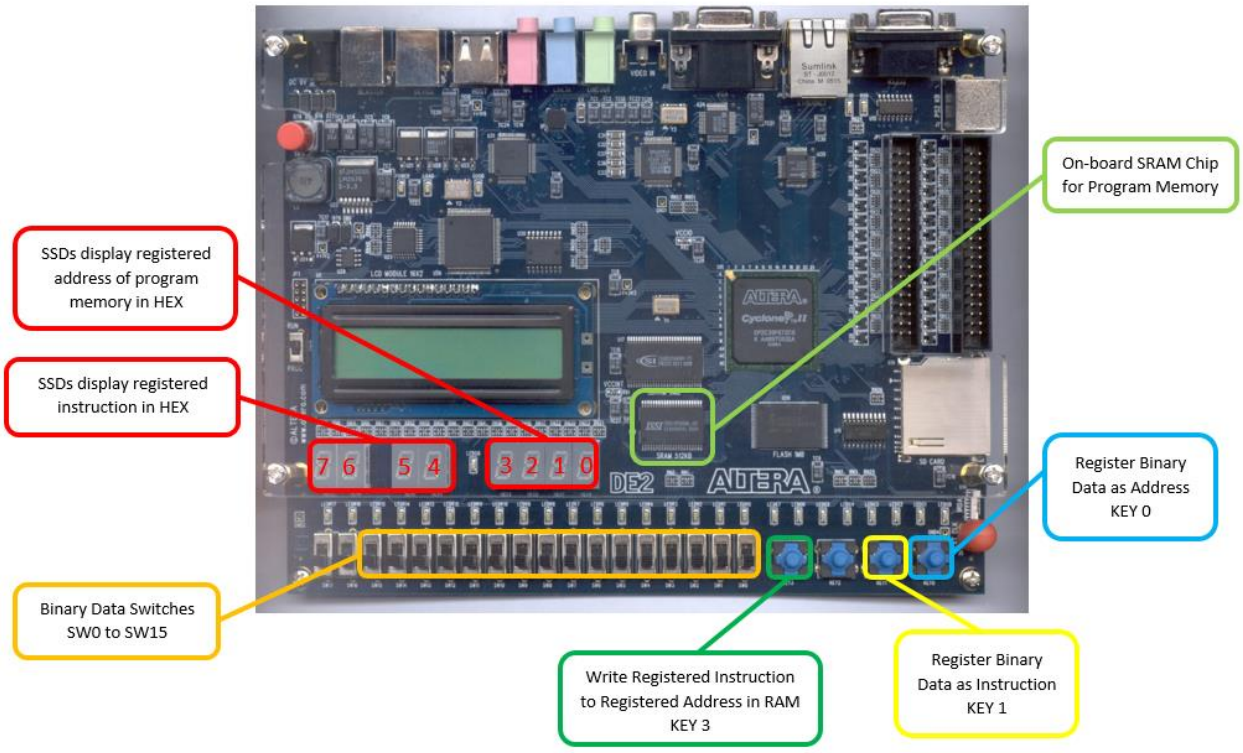


Figure 15: Display features available during the machine code programming of the onboard SRAM chip

Since each WIMPAVR instruction, and the PC, is 16 bits, and there are a limited number of switches available on the FPGA board, following is the process to enter the machine code at a particular program memory location:

1. Setup the program memory address using the switches SW15 – SW0.
2. Press the pushbutton KEY0 to register the memory address. This registered address is connected to the address bus of the onboard SRAM chip. The registered address is displayed on SSDs 3-0. This visual display helps minimize errors.
3. Setup the machine code, as seen in Figure 1, of the instruction, using switches SW15 – SW0, to be entered at the address set in steps 1 and 2.
4. Press the pushbutton KEY1 to register the machine code. This registered machine code is connected to the data bus of the onboard SRAM chip. The status of the switches is displayed on SSDs 7-4. This visual display helps minimize errors.
5. Press the pushbutton KEY3 to write the machine code into the onboard SRAM chip.
6. Repeat steps 1 through 5 till all the machine codes have been entered.

# YouTube Videos

Videos demonstrating the workings of the processor were created to aid in the dissemination and understanding of the processor working and FPGA board features. YouTube videos can be found at:

- WIMPAVR Introduction: https://www.youtube.com/watch?v=uDD9K1mkM-k
- WIMPAVR Instruction Set Review: https://www.youtube.com/watch?v=iUUuAPZLTBo
- WIMPAVR Program Memory Machine Code Upload Process: https://www.youtube.com/watch?v=eobcjZ4kyz0
- WIMPAVR Code Execution Example 1: Simple Program: https://www.youtube.com/watch?v=Jd8KeFuT1hg
- WIMPAVR Code Execution Example 2: Conditional Branching: https://www.youtube.com/watch?v=MFMkeym_ncU
- WIMPAVR Code Execution Example 3: 5*3 Using Repetitive Adding: https://www.youtube.com/watch?v=FwDAmfwAy4w