MEMORANDUM

To: ENGINEERING NOTEBOOK

From: MICHAEL A. MORRIS,

SUBJECT: SYSTEM DESIGN DESCRIPTION (SDD) FOR A SPI-BASED MINIMAL CPU FOR CPLDS

Date: 8/19/2012

CC:

PURPOSE

The memorandum provides the System Design Description (SDD) for a synthesizable Central Processing Unit (CPU) targeted to medium density Complex Programmable Devices (CLPDs). The Minimal CPU – Serial (MiniCPU-S) is intended to interface to memory and I/O using a Serial Peripheral Interconnect (SPI) compatible interface. The MiniCPU-S is designed as a full function microprocessor with a seven (7) function Arithmetic and Logic Unit (ALU), a general purpose stack/workspace, and support for subroutines and branches.

GENERAL CHARACTERISTICS

The primary two objectives of the design are to be synthesizable for medium density CPLDs and to support operation with a serial memory and I/O components. To meet the first objective, the MiniCPU-S implements a minimal architecture and instruction set encoding concepts pioneered by Inmos in its 16-bit and 32-bit microcomputers.

The MiniCPU-S provides an SPI-compatible external memory and I/O device interface. Support is provided for industry standard SEEPROMs and serial MRAM/FRAM which use industry standard command sets. The MiniCPU-S also expects that all external devices will also use an SPI-compatible interface.

PROGRAMMER ACCESSIBLE REGISTERS

The MiniCPU-S can be considered to be an example of a zero (0) address machine. In other words, the instruction set op codes do not allow the programmer direct access to any of the MiniCPU-S' registers. All registers are implicitly addressed by the instruction instead of being explicitly addressed by fields in the instruction. This architectural deci-

sion results in very compact instruction encoding, but only provides indirect access to the processor's registers.

From a programming perspective, the MiniCPU-S instruction set provides indirect access to the following registers:

(1) ALU register stack {A, B, C},
(2) Operand (Op) register,
(3) Workspace (W) pointer,
(4) Instruction (I) pointer.

ALU REGISTER STACK – {A, B, C}

The ALU is the core of the MiniCPU-S, and it consists of a 3 level push down register stack. The TOS register is designated A, the Next-On-Stack (NOS) register is designated B, and the bottom register of the stack is designated C. The register stack, {A, B, C}, provides a compact and efficient mechanism for performing arithmetic and logical computations using algorithms implemented using Reverse Polish Notation.

All ALU functions automatically adjust the ALU register stack as operands are used and automatically push the ALU results into the TOS register. That is, double operand ALU operations implicitly access A and B registers, remove both operands from the ALU stack, and automatically push the result onto the ALU stack. Similarly, single operand ALU functions implicitly access the A register, remove the A register from the ALU stack, and automatically push the result onto the ALU stack.

In infix operand notation, the A register provides the right operand, and the B register provides the left operand of any double operand ALU function. Therefore, ALU functions like addition, which requires two operands, is defined as {A <= B + A, B <= C, C}. This operational concept applies to all of the other two operand ALU functions.

OPERAND REGISTER – OP

The operand register Op is loaded from the lower four bits of the instruction. For direct instructions, the operand register provides a direct operand for ALU operations and a relative offset for load/store and branch instructions. For indirect instructions, the operand register provides the instruction op code.

For all direct instructions, the operand register provides a constant equal in width to the register width of all of the MiniCPU-S' registers, i.e. 16 bits. Since only four bits are loaded into the operand register with each direct instruction, two instructions are used to load the operand register 4 bits at a time. The upper 12 bits of the operand register using no more than three prefix instructions. There are two prefix instructions, PFX (Positive prefix) and NFX (Negative preFiX).

These two instructions allow the operand register to be loaded prior to the execution of a direct or indirect instruction. The operand register is cleared after the execution of any direct or indirect instruction except for the two prefix instructions. Thus, when any other instructions are executed, the operand register may contain an indirect instruction op code, a signed 16-bit address offset (with the base register provided by W, I, or A), or an ALU constant operand.

WORKSPACE POINTER REGISTER – W

Essentially, the workspace pointer corresponds to the stack pointer of a standard processor. Rather than using that term, the term workspace is used in order emphasize that the local variables as well as subroutine return addresses are stored in the workspace.

Local variables are stored at positive offsets from the workspace pointer. Each subroutine is expected to allocate space on the workspace by adjusting the value in the workspace pointer. Prior to exiting, the subroutine is expected to restore the workspace pointer to the value on entry into the subroutine.

On entry, the subroutine return address is stored at workspace offset 0. Negative adjustments allow the subroutines workspace to be increased. Positive adjustments reclaim temporarily allocated workspace.

The compact instruction format of the MiniCPU-S easily allows the first 16 locations relative to the workspace pointer to be accessed using a single byte instruction.

INSTRUCTION POINTER – I

The instruction pointer points to the current instruction. The address of an instruction is byte aligned. The instruction pointer is advanced, i.e. incremented, as each instruction is executed.

When a subroutine call, conditional branch, or unconditional jump is performed, a relative address operation is performed which is based on the address of the following instruction. In other words, the offset is taken from the address of the next instruction regardless of the type of branch operation being performed.

When making a subroutine call, the workspace pointer is pre-decremented and the instruction pointer value which points to the call instruction is written to the workspace. On return, the reloaded instruction pointer is pre-incremented before the next instruction is fetched from instruction memory.

AUXILIARY REGISTERS

In addition to the registers defined above, there is a single bit register which may be indirectly manipulated. A register for tracking inter-bit carries, CY, is also used for multi-

precision arithmetic. The serial nature of the MiniCPU-S' ALU requires a register for holding the inter-bit carries. Two instructions, used for rotating TOS left or right, can be used to initialize the CY register, but otherwise it is not directly accessible.

INSTRUCTION SET OF THE MINICPU-S

The objective of the instruction set of the MiniCPU-S is to provide maximum performance with as small an instruction set as possible. In addition, a compact instruction encoding scheme is used to extract maximum performance while minimizing the width of individual instructions.

BASIC INSTRUCTION FORMAT

The instructions for the MiniCPU-S are encoded in a single byte. The upper nibble of the byte consists of the op code for all of the direct instructions, and the lower nibble is the least significant four bits of the operand register: {IR[3:0], O[3:0]}. In the event that an indirect instruction is to be performed, the upper nibble is the EXEcute (EXE) op code, and the lower nibble is the lower four bits of the indirect instruction's op code: {0xF, Ind[3:0]}.

INSTRUCTION SET DEFINITION

The following table defines the currently defined MiniCPU-S instructions:

Table 1: MiniCPU-S Instruction Set.

Direct Instructions (IR == 0xIy)			Indirect Instructions (IR == 0xFy)		
Code	Mnemonic	Operation	Code	Mnemonic	Operation
0x0-	CALL	Call subroutine	0xF0	CLC	Clear Carry
0x1-	LDK	Load constant	0xF1	SEC	Set Carry
0x2-	LDL	Load local	0xF2	TAW	Transfer A to W
0x3-	LDNL	Load non-local	0xF3	TWA	Transfer W to A
0x4-	STL	Store local	0xF4	DUP	Duplicate A
0x5-	STNL	Store non-local	0xF5	XAB	Exchange A and B
0x6-	NFX	Negative prefix	0xF6	POP	Pop A
0x7-	PFX	Positive prefix	0xF7	RAS	Roll ALU register stack
0x8-	IN	Input word	0xF8	ROR	Rotate A right and set carry
0x9-	INB	Input byte	0xF9	ROL	Rotate A left and set carry
0xA-	OUT	Output word	0xFA	ADC	Add A to B plus carry
0xB-	OUTB	Output byte	0xFB	SBC	Subtract A from B minus carry
0xC-	BEQ	Branch if A is equal to 0	0xFC	AND	Logical AND of A into B
0xD-	BLT	Branch if A is less than 0	0xFD	ORL	Logical OR of A into B
0xE-	JMP	Unconditional jump	0xFE	XOR	Logical XOR of A into B
0xF-	EXE	Execute operand register	0xFF	HLT	Halt processor
			0x60F0	RTS	Return from subroutine
			0x60F1	RTI	Return from interrupt

With this encoding, 33 instructions are presently defined: 15 direct instructions, and 18 indirect instructions. The instructions in Table 1 are all defined as single byte instructions, With the exception of the double byte RTS and RTI instructions.

Furthermore, to define constants, memory pointers, or relative offsets greater than 15, between one and four PFX/NFX instructions may be required. Therefore, the actual instruction encoding encountered in an application will depend on the constants and relative displacements required by the application.

(**Note:** *in the future, the MiniCPU-S instruction set can be extended by defining additional indirect instructions which require one or more prefix instructions to initialize the upper bits of the operand register with the upper bits of the indirect instruction op code. As the instruction set of the MiniCPU-S is presently defined, no prefix instructions are required to specify any of the indirect instructions.*)

(**Note:** *the encoding of the HLT indirect instruction has been chosen to be the default value of erased EEPROMs. The encoding should be adjusted to make the EXE and HLT instruction encodings match the erased, or unprogrammed, state of whatever memory technology is used to implement the MiniCP-S instruction store.*)

CALL (0x0-) – CALL SUBROUTINE

The CALL instruction performs a subroutine call. The instruction pointer to the CALL instruction is pushed onto the workspace. Prior to performing the workspace write, the workspace pointer is decremented, i.e. pre-decremented. Following the write to the workspace of the return address, the instruction pointer is loaded with sum of the address of the instruction following the CALL instruction and operand register, and the operand register is cleared:

*(--W) <= I; I <= I + 1 + Op; Op <= 0;

(**Note:** *when the subroutine is entered, it must adjust the workspace pointer W in order to allocate any required local variables; on entry, W points to the return address. When exiting, the subroutine must adjust W in order to ensure that the local variables are discarded and W points to the return address.*)

LDK (0x1-) – LOAD CONSTANT

The LDK instruction loads the value of the operand register Op into ALU register A, increments the instruction pointer, and clears the operand register:

{A, B, C} <= {Op, A, B}; I <= I + 1; Op <= 0;

LDL (0x2-) – Load Local

The LDL instruction loads the value in memory pointed to by the sum of the workspace pointer and the operand register, increments the instruction pointer, and clears the operand register:

{A, B, C} <= {*(W + Op), A, B}; I <= I + 1; Op <= 0;

LDNL (0x3-) – Load Non-Local

The LDNL instruction loads the value in memory pointed to by the sum of ALU register A and the operand register Op, increments the instruction pointer, and clears the operand register:

{A, B, C} <= {*(A + Op), B, C}; I <= I + 1; Op <= 0;

As shown in the instruction definition given above, the pointer in register A is replaced with the value read from memory.

STL (0x4-) – Store Local

The STL instruction stores the value in the ALU top-of-stack register into the memory location pointed to by the sum of the workspace pointer and the operand register, pops the ALU register stack, increments the instruction pointer, and clears the operand register.

*(W + Op) <= A; {A, B, C} <= {B, C, C}; I <= I + 1; Op <= 0;

STNL (0x5-) – Store Non-Local

The STNL instruction stores the value in ALU register B into the memory location pointed to by the sum of the ALU register A and the operand register, pops both registers from the ALU register stack, increments the instruction pointer, and clears the operand register.

*(A + Op) <= B; {A, B, C} <= {C, C, C}; I <= I + 1; Op <= 0;

As shown in the instruction definition given above, both the pointer in register A and the value in register B are discarded and replaced by the value in register C.

NFX (0x6-) – Negative Prefix

The NFX instruction inserts four data bits read from memory into the operand register, complements the operand register, shifts the operand register four bits to the left, and increments the instruction pointer.

Op <= (~{Op[(N – 1):4], IR[3:0]} << 4); I <= I + 1;

Note that in the instruction definition given above, the lower 4 bits of the instruction byte are inserted into the lower four bits of the operand register. This operation affects the entire operand register regardless of its width. In the MiniCPU-S architecture, the ALU registers, the workspace pointer, and the instruction pointer all have the same width. This characteristics of the registers facilitates the implementation of the local and non-local indexed addressing modes previously defined. The target register width for MiniCPU-S registers is 16 bits. Also note that unlike other instructions (except the PFX instruction defined below), the NFX instruction does not clear the Op register when the instruction completes.

PFX (0x7-) – Positive Prefix

The PFX instruction inserts four data bits read from memory into the operand register, shifts the operand register four bits to the left, and increments the instruction pointer.

Op <= ({Op[(N – 1):4], IR[3:0]} << 4); I <= I + 1;

Note that unlike the NFX instruction, the PFX does not complement or otherwise operate on the operand register except to left shift the register 4 bits. Also note that unlike other instructions (except the NFX instruction defined above), the PFX instruction does not clear the Op register when the instruction completes.

IN (0x8-) – Input Word

The IN instruction reads a word into ALU register A from an SPI-compatible I/O device selected by the value in the operand register (with a device address provided by ALU register A), increments the instruction pointer, and clears the operand register:

{A, B, C} <= {SPI_Rd16[Op, A], B, C}; I <= I + 1; Op <= 0;

Like the LDL and LDNL instructions, IN replaces the device address in ALU register A with the data read from the device.

The notation SPI_Rd16[Op, A] is used to represent a standard 16-bit SPI Read operation. The first parameter, Op, provides the selector for the SPI I/O chip select, and the second parameter, A, provides the address of the data within the address space of the SPI I/O device. A standard SPI read cycle will be used where the SPI read command is followed by the address provided by register A and 16 0s: {0x03, A, 0x0000}. All total, the IN in-

struction transmits 40 bits during an SPI_Rd16 operation, and 16 data bits from the device are read into ALU register A during the last 16 cycles of the operation.

INB (0x9-) – Input Byte

The IN instruction reads a byte into ALU register A from an SPI-compatible I/O device selected by the value in the operand register (with a device address provided by ALU register A), increments the instruction pointer, and clears the operand register:

{A, B, C} <= {SPI_Rd08[Op, A], B, C}; I <= I + 1; Op <= 0;

Like the LDL and LDNL instructions, INB replaces the device address in ALU register A with the data read from the device, The 8-bit value read from the device is loaded into the least significant byte and the upper byte is loaded with 0s.

The notation SPI_Rd08[Op, A] is used to represent a standard 8-bit SPI Read operation. The first parameter, Op, provides the selector for the SPI I/O chip select, and the second parameter, A, provides the address of the data within the address space of the SPI I/O device. A standard SPI read cycle will be used where the SPI read command is followed by the address provided by register A and 8 0s: {0x03, A, 0x00}. All total, the INB instruction transmits 32 bits during an SPI_Rd08 operation, and 8 data bits from the device are read into ALU register A during the last 8 cycles of the operation.

OUT (0xA-) – Output Word

The OUT instruction writes a word from ALU register B to an SPI-compatible I/O device selected by the value in the operand register (with a device address provided by ALU register A), increments the instruction pointer, and clears the operand register:

SPI_Wr16[Op, A, B]; {A, B, C} <= {C, C, C}; I <= I + 1; Op <= 0;

Like the STL and STNL instructions, both parameters are removed from the ALU stack and replaced with ALU register C.

The notation SPI_Wr16[Op, A, B] is used to represent a standard 16-bit SPI Write operation. The first parameter, Op, provides the selector for the SPI I/O chip select, the second parameter, A, provides the address of the data within the address space of the SPI I/O device, and the third parameter, B, provides the 16 bits of output data. Unlike SPI memory devices, the SPI I/O devices are not expected to require an SPI output cycle to enable writes to the device. Thus, the SPI I/O device is expected to accept a standard SPI write command, followed by the address in ALU register A and the data in ALU register B: {0x02, A, B}. All total, the OUT instruction transmits 40 bits of data to the external SPI device.

OUTB (0xB-) – OUTPUT BYTE

The OUTB instruction writes a byte from ALU register B[7:0] to an SPI-compatible I/O device selected by the value in the operand register (with a device address provided by ALU register A), increments the instruction pointer, and clears the operand register:

SPI_Wr08[Op, A, B]; {A, B, C} <= {C, C, C}; I <= I + 1; Op <= 0;

Like the STL and STNL instructions, both parameters are removed from the ALU stack and replaced with ALU register C.

The notation SPI_Wr08[Op, A, B] is used to represent a standard 8-bit SPI Write operation. The first parameter, Op, provides the selector for the SPI I/O chip select, the second parameter, A, provides the address of the data within the address space of the SPI I/O device, and the third parameter, B, provides the 8-bits of output data. Unlike SPI memory devices, the SPI I/O devices are not expected to require an SPI output cycle to enable writes to the device. Thus, the SPI I/O device is expected to accept a standard SPI write command, followed by the address in ALU register A and the data in ALU register B: {0x02, A, B[7:0]}. All total, the OUTB instruction transmits 32 bits of data to the external SPI device.

BEQ (0xC-) – BRANCH IF EQUAL

The BEQ instruction performs a conditional branch if ALU register is equal to 0, otherwise the next instruction is executed, and clears the operand register:

I <= ((A == 0) ? I + 1 + Op : I + 1); Op <= 0;

As can be seen from the instruction definition given above, the branching operation is effectively the same calculation except that the operand register value is masked by the test condition, i.e. the ALU's internal zero flag:

I <= I + 1 + Z & Op;

BLT (0xD-) – BRANCH IF LESS THAN

The BLT instruction performs a conditional branch if ALU register is less than 0, otherwise the next instruction is executed, and clears the operand register:

I <= ((A < 0) ? I + 1 + Op : I + 1); Op <= 0;

As can be seen from the instruction definition given above, the branching operation is effectively the same calculation except that the operand register value is masked by the test condition, i.e. the ALU's internal negative flag:

I <= I + 1 + N & Op;

JMP (0xE-) – UNCONDITIONAL JUMP

The JMP instruction performs an unconditional jump to the location addressed by the sum of the instruction pointer and the operand register, and clears the operand register:

$$I <= I + 1 + Op; Op <= 0;$$

The definition of this instruction indicates that the operand register holds a relative offset to the destination. An alternative definition would be to assign the instruction pointer the value of the operand register:

$$I <= Op; Op <= 0;$$

This alternative definition is easy to implement and use, but locks the program into a specific address range. The first definition allows easy relocation of a program in the address space.

EXE (0xF-) – EXECUTE INDIRECT INSTRUCTION

The EXE instruction executes the indirect instruction whose op code is in the operand register, increments the instruction pointer, and clears the operand register:

$$Execute(Op); I <= I + 1; Op <= 0;$$

The indirect instructions are implemented in the same manner as direct instructions except that their op codes are taken from the operand register instead of the upper four bits of the instruction word. The operation of each of the currently defined indirect instructions is provided in the following subsections.

CLC (0xF1) – Clear Carry

The CLC instruction clears the Carry register, increments the instruction pointer, and clears the operand register.

$$\{A, B, C\} <= \{A, B, C\}; Cy <= 0;$$

SEC (0xF1) – Set Carry

The SEC instruction sets the Carry register, increments the instruction pointer, and clears the operand register.

$$\{A, B, C\} <= \{A, B, C\}; Cy <= 1;$$

TAW (0xF2) – Transfer A to W

The TAW instruction transfers ALU register A to the workspace pointer W, and pops the ALU register stack:

W <= A; {A, B, C} <= {B, C, C};

TWA (0xF3) – Transfer W to A

The TWA instruction pushes the workspace pointer W onto the ALU register stack:

{A, B, C} <= {W, A, B};

DUP (0xF4) – Duplicate A

The DUP instruction duplicates ALU register A in the ALU register stack:

{A, B, C} <= {A, A, B};

XAB (0xF5) – Exchange A and B

The XAB exchanges ALU register stack registers A and B:

{A, B, C} <= {B, A, C};

POP (0xF6) – Pop ALU Register Stack

The POP instruction pops ALU register A from the ALU register stack:

{A, B, C} <= {B, C, C};

RAS (0xF7) – Roll ALU Register Stack

The RAS instruction rolls the ALU register stack:

{A, B, C} <= {B, C, A};

This RAS was defined instead of an instruction exchanging ALU registers A and C, XAC:

{A, B, C} <= {C, A, B};

In either case, either the combination of XAB and RAS, or XAB and XAC can be used to manipulate the order of operands on the ALU register stack.

Using the ALU stack for chained calculations significantly improves overall performance. Having the capability of manipulating the operand order on the ALU register stack reduces the number of times that the ALU register stack must be unloaded and re-

loaded from memory in order to perform calculations. Reducing the number of stores and loads will significantly improve performance particularly when using SPI memories.

Each non-contiguous SPI memory read requires either 33 or 41 SPI clock cycles. Each SPI memory write cycle requires 9 cycles to issue the write enable command, and an additional 33 to 41 clock cycles to write the data to the SPI memory device. Since the SPI clock is generally slower than the processor's clock by a factor of at least two, non-contiguous SPI memory reads and writes may require between 66 and 100 clock cycles to complete. Thus, maintaining computations in the ALU register stack is very important from a performance perspective.

ROR (0xF8) – Rotate A Right and Set Carry

The ROR instruction rotates ALU register A right by an amount defined by a mask in ALU register B, the operands are discarded, and the result is pushed into ALU register A:

{A, B, C} <= {ROR(A, B), C, C}; Cy <= ROR(A, B);

The rotation mask is defined in ALU register B. ALU register B is shifted right after each shift cycle. The LSB of ALU register B determines if ALU register A is rotated or not. If the LSB is a 1, ALU register is rotated, otherwise not rotation occurs.

A <= (B[0] == 1) ? {A[0], A[15:1]} : A;

Thus, the total number of 1 bits is ALU register B determine the number of rotations that are applied to the value in ALU register A.

In addition to the rotation, for each bit in ALU register B that is set, the value of the ALU register A bit about to be rotated is captured in the CY register.

CY <= (B[0] == 1) ? A[0] : CY;

For example, if ALU register B contains the value 0xFFFF, then ALU register A is rotated right 16 positions, which means that a complete rotation has been applied to ALU register A and its value is the same as it was at the start of the instruction.

{A, B, C} <= {0x96A5, 0xFFFF, C}; CY <= -;
{A, B, C} <= {0x96A5, C, C}; CY <= 1 (A[15]);

If ALU register B is loaded with a mask of 0x00FF or any mask containing eight 1s, then ALU register A is rotated 8 bits such that the lower and upper halves of the register are exchanged.

{A, B, C} <= {0x695A, 0x00FF, C}; CY <= -;
{A, B, C} <= {0x5A69, C, C}; CY <= 0 (A[7]);

ROL (0xF9) – Rotate A Left and Set Carry

The ROL instruction rotates ALU register A left by an amount defined by a mask in ALU register B, the operands are discarded, and the result is pushed into ALU register A:

{A, B, C} <= {ROL(A, B), C, C}; Cy <= ROL(A, B);

The rotation mask is defined in ALU register B. ALU register B is shifted right after each shift cycle. The LSB of ALU register B determines if ALU register A is rotated or not. If the LSB is a 1, ALU register is rotated, otherwise not rotation occurs.

A <= (B[0] == 1) ? { A[14:0], A[15]} : A;

Thus, the total number of 1 bits is ALU register B determine the number of rotations that are applied to the value in ALU register A.

In addition to the rotation, for each bit in ALU register B that is set, the value of the ALU register A bit about to be rotated is captured in the CY register.

CY <= (B[0] == 1) ? A[15] : CY;

For example, if ALU register B contains the value 0xFFFF, then ALU register A is rotated right 16 positions, which means that a complete rotation has been applied to ALU register A and its value is the same as it was at the start of the instruction.

{A, B, C} <= {0x695A, 0xFFFF, C}; CY <= -;
{A, B, C} <= {0x96A5, C, C}; CY <= 0 (A[0]);

If ALU register B is loaded with a mask of 0x00FF or any mask containing eight 1s, then ALU register A is rotated 8 bits such that the lower and upper halves of the register are exchanged.

{A, B, C} <= {0xA596, 0x00FF, C}; CY <= -;
{A, B, C} <= {0x96A5, C, C}; CY <= 1 (A[8]);

ADC (0xFA) – 2's Complement Add of A to B Plus Carry

The ADC instructions adds ALU register A to ALU register B plus the initial value of the carry register, discards the operands, and pushes the result onto the ALU register stack:

{A, B, C} <= {{B + A + CY}, C, C}; CY <=C_{16};

The definition of this instruction is designed to allow the implementation of multi-precision arithmetic, but it requires that the carry register be initialized to 0 before the first ADC instruction is executed. If the carry is not initialized, the result may be incorrect.

SBC (0xFB) – 1's Complement Subtract of A From B Plus Carry

The SBC instruction adds the 1's complement of ALU register A to ALU register B plus the initial value of the carry register, discards the operands, and pushes the result onto the ALU register stack:

$$\{A, B, C\} <= \{\{B + \sim A + CY\}, C, C\}; CY <= C_{16};$$

The definition of this instruction is designed to allow the implementation of multi-precision arithmetic, but it requires that the carry register be initialized to 1 before the first SBC instruction is executed. If the carry is not initialized, the 1's complement sum of B + ~A + CY will result in an incorrect 2's complement result.

AND (0xFC) – Bitwise Logical AND of A into B

The AND instruction performs a bitwise logical AND of ALU registers A and B, discards the operands, and pushes the result onto the ALU register stack:

$$\{A, B, C\} <= \{\{B \& A\}, C, C\};$$

ORL (0xFD) – Bitwise Logical OR of A into B

The ORL instruction performs a bitwise logical OR of ALU registers A and B, discards the operands, and pushes the result onto the ALU register stack:

$$\{A, B, C\} <= \{\{B \mid A\}, C, C\};$$

AND (0xFE) – Bitwise Logical XOR of A into B

The XOR instruction performs a bitwise logical XOR of ALU registers A and B, discards the operands, and pushes the result onto the ALU register stack:

$$\{A, B, C\} <= \{\{B \wedge A\}, C, C\};$$

HLT (0xFF) – Processor Halt

The HLT instruction stops the processor. All processor activity stops, and can only be restarted by the assertion and release of the system reset signal or by the assertion of enabled interrupt.

RTS (0x60F0) – Return From Subroutine

The RTS instruction performs a return from subroutine by reading the workspace location pointed to by the workspace pointer and loading that value into the instruction pointer, and incrementing the workspace pointer:

I <= *(W); W <= W + 1;

The RTS instruction code consists of an `NFX 0` (0x60) instruction followed by an `EXE 0` (0xF0) instruction. This instruction sequence sets the operand register to a value of 0xFFF0. This allows the instruction decoder to distinguish between the RTS and CLC instructions, which both use `EXE 0` (0xF0) to initiate the execution of these two indirect instructions.

RTI (0x60F1) – Return From Interrupt

The RTI instruction is intended to perform a return from interrupt. The definition of interrupt handling is not yet complete, and when that is completed, this instruction will be fully defined.

The RTS instruction code consists of an `NFX 0` (0x60) instruction followed by an `EXE 1` (0xF1) instruction. This instruction sequence sets the operand register to a value of 0xFFF1. This allows the instruction decoder to distinguish between the RTI and SEC instructions, which both use `EXE 1` (0xF1) to initiate the execution of these two indirect instructions.

OPERATION OF SPI MEMORY DEVICE INTERFACE

OPERATION OF SPI I/O DEVICE INTERFACE