MEMORANDUM

To: ENGINEERING NOTEBOOK

From: MICHAEL A. MORRIS,

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

Date: 9/4/2013

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, support for subroutines and branches, and support for I/O and interrupts.

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.

Like the Inmos microcomputers, the MiniCPU-S has two classes of instructions: (1) "direct" instructions, and (2) "indirect" instructions. The basic instruction format consists of two four (4) bit components. The upper four bits of every opcode represents a "direct" instruction, and the lower four bits is a direct operand value loaded into an "operand" register. The four "direct" instructions encoded in the upper four bits of each opcode provide fifteen (15) direct instructions and one instruction for executing "indirect" instructions. the "opcode" of the "indirect" instructions is the value in the "operand" register.

(**Note:** *this mechanism allows the MiniCPU-S instruction set to be very compactly encode its instruction set. Furthermore, it allows the instruction set to be extended as required by an application while maintaining the compact encoding required for efficiency.*)

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 decision 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) ALU Carry (Cy) register,
(3) Operand (Op) register,
(4) Workspace (W) pointer,
(5) 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 the A and the 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 (standard algebraic 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 is used for three purposes: (1) immediate operands, (2) relative offsets for branches and subroutine calls, and (3) holding the opcode of an "indirect" instruction. The operand register Op is loaded from the lower four bits of the instruction.

Since only four bits can be loaded into the operand register with each opcode, two "direct" instructions are used to load the operand register 4 bits at a time: PFX (Positive prefix) and NFX (Negative preFiX). These instructions logically shift the operand register left 4 bits, and insert the least significant four bits of the opcode into the least significant bits of the operand register. These two instructions allow the operand register to be loaded prior to the execution of a "direct" or an "indirect" instruction. The 4-bit values in three prefix instructions plus the least significant four bits in a "direct" instruction fill in the 16-bit "operand" register.

The operand register is cleared after the execution of any "direct" or "indirect: instruction except after either of 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 a constant operand to be loaded into the ALU stack.

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, CY, which may be indirectly manipulated. The serial nature of the MiniCPU-S' ALU requires a register for holding the inter-bit carries. Unlike the Inmos microcomputers but like the 6502, CY can also used for multi-precision arithmetic.

Two "indirect" instructions, Clear Carry (CLC) and Set Carry (SEC), directly clear or set the value of the CY register. The CY register is also manipulated by two other instructions, Rotate Right and Set Carry (ROR) and the Rotate Left and Set Carry (ROL). When ROR or ROL are used, the last bit rotated is captured into the CY register.

INSTRUCTION SET OF THE MINICPU-S

The objective of the instruction set of the MiniCPU-S is to provide maximum performance with as minimal 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]}.

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

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-	PFX	Positive prefix	0x20	CLC	Clear Carry
0x1-	NFX	Negative prefix	0x21	SEC	Set Carry
0x2-	EXE	Execute operand register	0x22	TAW	Transfer A to W
0x3-	LDC	Load constant	0x23	TWA	Transfer W to A
0x4-	LDL	Load local	0x24	DUP	Duplicate A
0x5-	LDNL	Load non-local	0x25	XAB	Exchange A and B
0x6-	STL	Store local	0x26	POP	Pop A
0x7-	STNL	Store non-local	0x27	RAS	Roll ALU register stack
0x8-	IN	Input word	0x28	ROR	Rotate A right and set carry
0x9-	INB	Input byte	0x29	ROL	Rotate A left and set carry
0xA-	OUT	Output word	0x2A	ADC	Add A to B plus carry
0xB-	OUTB	Output byte	0x2B	SBC	Subtract A from B minus carry
0xC-	BEQ	Branch if A is equal to 0	0x2C	AND	Logical AND of A into B
0xD-	BLT	Branch if A is less than 0	0x2D	ORL	Logical OR of A into B
0xE-	JMP	Unconditional jump	0x2E	XOR	Logical XOR of A into B
0xF-	CALL	Call subroutine	0x2F	HLT	Halt processor
			0x1020	RTS	Return from subroutine
			0x1021	RTI	Return from interrupt

(**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 except for the single NFX prefix instruction needed to specify the RTS/RTI indirect instructions.*)

PFX (0x0-) – 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.

$$Op <= (\{Op[15:4], IR[3:0]\} << 4); I <= I + 1;$$

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.

NFX (0x1-) – 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:

$$Op <= (\sim\{Op[15:4], IR[3:0]\} << 4); I <= I + 1;$$

These operations on the operand are equivalent to:

$$Op <= \{\sim Op[11:4], \sim IR[3:0], 4'b0000\};$$

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, the operand register is complemented, and then shifted left 4 bits. 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 characteristic of the MiniCPU-S registers facilitates the implementation of the local and non-local indexed addressing modes previously defined. The 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.

LDC (0x3-) – 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 (0x4-) – 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 (0x5-) – 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 (0x6-) – 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 (0x7-) – 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.

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 command and register 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 command and register 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 least significant two bits of the first parameter, Op, provides the selector for the SPI I/O chip select, and the second parameter, A, provides the device command and register address of the data within the SPI I/O device. An SPI I/O read cycle will consist of the 16 bits of ALU register A, followed by 16 dummy bits: {A, 0x0000}. All total, the IN instruction transmits 32 bits during an SPI_Rd16 operation, and 16 data bits from the SPI I/O device are read into ALU register A during the final 16 cycles of the operation.

INB (0x9-) – INPUT BYTE

The INB 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 command and register 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 command and register 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 least significant two bits of the first parameter, Op, provides the selector for the SPI I/O chip select, and the second parameter, A, provides the device command and register address of the data within the SPI I/O device. An SPI I/O read cycle will consist of the 16 bits of ALU register A, followed by 8 dummy bits: {A, 0x00}. All total, the IN □ncur□ction transmits 24 bits during an SPI_Rd08 operation, and 8 data bits from the SPI I/O device are read into ALU register A during the final 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 command and register 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] is used to represent a 16-bit SPI Write operation. The least significant two bits of the first parameter, Op, provides the selector for the SPI I/O chip select, and the second parameter, A, provides the device command and register address of the data within the SPI I/O device. An SPI I/O write cycle will consist of the 16 bits of ALU register A, followed by the 16 bits of ALU Register B: {A, B}. All total, the OUT instruction transmits 32 bits during an SPI_Wr16 operation.

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 command and register

address provided by ALU register A), increments the instruction pointer, and clears the operand register:

SPI_Wr08(Op, A, B[7:0]); {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] is used to represent an 8-bit SPI Write operation. The least significant two bits of the first parameter, Op, provides the selector for the SPI I/O chip select, and the second parameter, A, provides the device command and register address of the data within the SPI I/O device. An SPI I/O write cycle will consist of the 16 bits of ALU register A, followed by 8 bits from ALU register B: {A, B[7:0]}. All total, the OUT instruction transmits 24 bits during an SPI_Wr08 operation.

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.

CALL (0xF-) – 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 pre-decremented by two (2). 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 the reverse the adjustment made to W on entry in order to ensure that the local variables are discarded and W points to the return address.*)

EXE (0x0-) – 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.

INDIRECT INSTRUCTIONS

The operation of each of the currently defined indirect instructions is provided in the following subsections. The first sixteen (16) indirect instructions do not require a prefix instruction, but the last two indirect instructions require a single NFX prefix instruction to precede the EXE direct instruction.

CLC (0x20) – Clear Carry (Op: 0x0000)

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 (0x21) – Set Carry (Op: 0x0001)

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 (0x22) – Transfer A to W (Op: 0x0002)

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 (0x23) – Transfer W to A (Op: 0x0003)

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

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

DUP (0x24) – Duplicate A (Op: 0x0004)

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

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

XAB (0x25) – Exchange A and B (Op: 0x0005)

The XAB exchanges ALU register stack registers A and B:

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

POP (0x26) – Pop ALU Register Stack (Op: 0x0006)

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

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

RAS (0x27) – Roll ALU Register Stack (Op: 0x0007)

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:

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

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 reloaded 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 (0x28) – Rotate A Right and Set Carry (Op: 0x0008)

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 (0x29) – Rotate A Left and Set Carry (Op: 0x0009)

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 (0x2A) – B Plus A Plus Carry (Op: 0x000A)

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 ncurrect.

SBC (0x2B) – B Plus 1's Complement of A Plus Carry (Op: 0x000B)

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 (0x2C) – Bitwise Logical AND of A into B (Op: 0x000C)

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 (0x2D) – Bitwise Logical OR of A into B (Op: 0x000D)

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 (0x2E) – Bitwise Logical XOR of A into B (Op: 0x000E)

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 (0x2F) – Processor Halt (Op: 0x000F)

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 (0x1020) – Return From Subroutine (Op: 0xFFF0)

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` (0x10) instruction followed by an `EXE 0` (0x20) 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` (0x20) to initiate the execution of these two indirect instructions.

RTI (0x1021) – Return From Interrupt (Op: 0xFFF1)

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 RTI 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.

SPI FLASH MEMORY INTERFACE

SPI Flash memory devices are generally characterized as requiring a single byte command, followed by one, two, or three address bytes, and one or more data bytes. In the event or a read operation, the host (MiniCPU-S) transmits to the serial program memory device the read command byte, the address (8, 16, or 24 bits) bytes, and 8 or more dummy bytes of zeros. During the command and address bytes, the SPI Flash memory device generally holds its output in a high impedance state. While the dummy zero bytes are being transmitted from the host device to the SPI Flash memory device, the device is outputting the requested data. (**Note:** *SPI Flash memory devices generally accept commands and addresses MSB first. Therefore, SPI data is generally is also output and input MSB first.*)

Most SPI Flash memory devices allow an unlimited number of bytes to be sequentially read. Most SPI Flash memory devices use a paged architecture for the memory array. This architectural feature is generally not an issue with respect to reads, but is important from a write perspective. When reading an SPI Flash memory device sequentially, the next page is automatically fetched into the internal read buffer when a boundary is crossed so that sequential reads of the memory array can be performed without any limitation other than recirculation back to the very first byte when the very last byte is read.

On writes, however, the write data is written into a buffer whose size matches the page size of the device. Writing more data to the device simply overwrites data in the buffer modulo the page size. When the write cycle terminates on a byte boundary, the data in the write buffer is physically programmed into the selected memory page. The programmer must poll the device, or delay until the write buffer has been programmed into the page.

The industry has adopted a command set for the common operations. Internally, SPI Flash memory devices compatible with the industry standard command set provide a status register that contains a programmable bit that enables the writing of the memory pages. This requires a single command cycle to enable writes to the device, followed by a second SPI cycle that transmits the write command, the desired page number, and the page offset, followed by the data to be written into the selected page. Most SPI Flash memory devices require an SPI write cycle to terminate on a byte boundary. Otherwise the write cycle is aborted, and the contents of the write buffer are discarded.

Finally, for the purposes of the MiniCPU-S, the SPI Flash memory device is expected to be pre-loaded with the desired program image. Support will be provided for the writing program memory one byte at a time, but this is very inefficient and wastes the program

cycles of the SPI Flash memory device since a single byte is written instead of the entire page. The program life of these devices applies to all memory cells in a page rather than to individual cells in the page. Therefore, writing a single byte to a page effectively reduces the program life of the page by the number of bytes in a page.

Most SPI Flash memory devices support a hold feature that effectively disables the device so that an SPI to another device can be performed. The MiniCPU-S memory interface supports this capability for the Flash EPROM device. With sequential accesses, the overhead penalty imposed by the read command and address bytes decreases with the length of the sequential transfer. Using the hold feature of these devices, the sequential access mode can be preserved while the MiniCPU-S performs a random access read/write cycle to an SPI FRAM/MRAM device or IO device.

Given the simple instruction set for the MiniCPU-S, the expectation is that sequential access of program memory will be the norm rather than random accesses of program memory. All program flow branches (conditional branches and unconditional jumps, subroutine calls and returns, and interrupt service routine calls and returns) will require a new SPI memory read cycle to be issued to the memory device. The overhead to access the first instruction will be 82 clock cycles: 2 (SPI cycle delay), 2*32 command and address cycles, and 2*8 output dummy data cycles. If this has to be repeated for each instruction, the overhead would make the performance so poor that further development of the MiniCPU-S would be unwarranted.

Using the hold feature of the SPI Flash memory device means that the 82 cycle penalty would not be incurred again until a program flow branch is encountered. When SPI FRAM/MRAM and IO is required, the program memory device is put into the hold mode, and an SPI read/write cycle to the FRAM/MRAM or IO device is performed. The SPI overhead associated with these devices is generally less than that associated with SPI Flash EPROM devices. The address field of SPI Flash memory devices is usually three bytes, while that of SPI FRAM/MRAM or IO devices is one or two bytes. The expected SPI single cycle (read/write) overhead is expected to be as follows:

(1) SPI Flash – 82 (read) and 100 (write) cycles;
(2) SPI FRAM – 66 (read) and 84 (write) cycles;
(1) IO – 34-50 (read/write) cycles.

(**Note:** *SPI* Flash memory *and FRAM/MRAM devices require a single 8-bit Write Enable command before the write cycle. At the SPI clock rate of ½ of the system clock rate, the write enable command requires 2*8 (16) plus 1 (2) cycles, or 18 cycles.*)

The lack of logic and register resources in the target devices for the MiniCPU-S, i.e. medium density CPLDs such as the Xilinx XC9572, means that extensive address tracking and comparison logic is not an option for a MiniCPU-S implementation. However, it is straight forward to recognize non-sequential program memory reads using a simple instruction decoder. Therefore, although using the hold feature of the SPI Flash memory devices may use additional scarce resources, the expected improvement in overall per-

formance should outweigh the reduction in logic and register resources available to other MiniCPU-S components.

SPI FRAM/MRAM MEMORY INTERFACE

An SPI FRAM/MRAM device is designed to operate in a manner very similar to that used by industry standard SPI Flash memory devices. An SPI FRAM/MRAM device also partitions and protects the memory array in the same manner as industry standard SPI Flash memory devices. However, since the storage cells are ferro-magnetic/magneto-resistive instead of Flash EPROM cells, there is no page buffer. Furthermore, the data is written directly into the storage cell without a programming delay.

SPI FRAM/MRAM devices are designed as drop in replacements for SPI Flash memory devices. Thus, they use the same command set and status register organization as industry standard SPI Flash memory devices. They even provide the same mechanism for verifying the programming cycles are complete. In this manner, although SPI FRAM/MRAM devices do not require a program delay, they can be used with firmware/software that expects to program SPI Flash memory devices without any change to the programming routines.

In the MiniCPU-S, the data memory is expected to use SPI FRAM/MRAM components. These devices support the SPI hold mode. However, given the limited resources of CPLDs, the FRAM/MRAM memory interface will not support the SPI hold mode for data memory accesses. If an SPI FRAM/MRAM memory device replaces an SPI Flash memory devices for program storage, the SPI hold supported for that interface will function as expected.

SPI I/O DEVICE INTERFACE

SPI I/O devices fall into two general classes with respect to SPI. One class functions in a manner similar to that of most SPI memory devices. The other class functions as bidirectional shift registers.

In the first class, an SPI I/O device expects a command followed by a register address. Like SPI memory devices, these SPI I/O devices do not drive the MISO (Master-In, Slave-Out) pin during the time command and register address is being driven by the MiniCPU-S. Therefore, an SPI I/O read cycle for these devices follows the command and address cycle. Most devices in this class support either sequential or repeated access to a single device location. When set to support repeated access to a single location, the device is able to support block transfers to/from internal FIFOs.

In the second class, an SPI I/O device simultaneously receives and transmits. This class of SPI I/O devices generally only support fixed length transfers. For example, an Analog Devices AD7328 8 channel, 13-bit ADC always expects an SPI cycle length of 16 bits.

The device simultaneously receives a command for an internal register or ADC conversion value, and transmits the value of the last command or ADC conversion.

With the first class of SPI devices, the MiniCPU-S Program Control Unit (PCU) must transmit an extra 8 or 16 dummy bits to retrieve the data. The value of the SPI input data pin is undefined until after the command and register address bits have been transmitted by the MiniCPU's SPI. On the other hand, read data from a class 2 device is overlapped with the command/device address data transmitted by the MiniCPU-S.

The definition of the IN/INB and OUT/OUTB instructions have been defined so that these two types of SPI I/O devices can be supported with only a small amount of programming overhead. Two key elements of these instructions support the two operating mode of the SPI I/O interface: (1) the least significant 4 bits of the operand register determine the SPI mode, the device class, and the SPI channel of the device.

In addition to the two SPI channels, i.e. SPI slave select signals, independently supporting access to the program and data memory SPI devices, there are four additional SPI channels dedicated to the I/O. That is, the MiniCPU-S provides a total of 6 SPI slave select signals: (1) nSS_ROM, (2) nSS_RAM, (3) nSS_IO[3:0].

The four I/O SPI channel selects are determined by least significant two bits of the operand register, Op[1:0]. The SPI protocol, or operating mode, is controlled by the setting of bit 2 of the operand register. By default, SPI mode 00 is the default. Setting Op[2] changes the SPI mode from 00 to 11. Finally, the device type is determined by the setting of operand register bit 3, Op[3]. If this bit is clear, then a class 1 SPI I/O cycle is performed. If the bit is set, then a class 2 SPI I/O cycle is performed. The value loaded into the Op register when executing an IN(B)/OUT(B) instruction can be performed as a single byte instruction.

(**Note:** *in the future, further changes to the IN(B)/OUT(B) instructions may be made if additional SPI device types are required. The discussion of the operation of these four instructions is compatible with the definition provided above for the instructions.*)

SPI CHANNEL MAPPING

The MiniCPU-S supports six (6) SPI channels. That is, six slave select outputs are provided from the MiniCPU-S PCU. The following table maps the MiniCPU-S instructions to the six SPI channels.

Code	Mnemonic	ROM	RAM	IO[3:0]	Comments/Notes
0x0-	PFX	X			Constant loaded from instruction space
0x1-	NFX	X			Constant loaded from instruction space
0x2-	EXE	X			Execute Indirect Instructions
0x3-	LDC	X			Constant loaded from instruction space
0x4-	LDL	X	X		Operand loaded from workspace

Code	Mnemonic	ROM	RAM	IO[3:0]	Comments/Notes
0x5-	LDNL	X	X		Operand loaded from workspace
0x6-	STL	X	X		Operand stored to workspace
0x7-	STNL	X	X		Operand stored to workspace
0x8-	IN	X		X	Input data loaded from SPI I/O
0x9-	INB	X		X	Input data loaded from SPI I/O
0xA-	OUT	X		X	Output data stored to SPI I/O
0xB-	OUTB	X		X	Output data stored tp SPI I/O
0xC-	BEQ	X			Target addresses in instruction space
0xD-	BLT	X			Target addresses in instruction space
0xE-	JMP	X			Target address in instruction space
0xF-	CALL	X	X		Return address pushed on workspace
0x20	CLC	X			Next operation from instruction space
0x21	SEC	X			Next operation from instruction space
0x22	TAW	X			Next operation from instruction space
0x23	TWA	X			Next operation from instruction space
0x24	DUP	X			Next operation from instruction space
0x25	XAB	X			Next operation from instruction space
0x26	POP	X			Next operation from instruction space
0x27	RAS	X			Next operation from instruction space
0x28	ROR	X			Next operation from instruction space
0x29	ROL	X			Next operation from instruction space
0x2A	ADC	X			Next operation from instruction space
0x2B	SBC	X			Next operation from instruction space
0x2C	AND	X			Next operation from instruction space
0x2D	ORL	X			Next operation from instruction space
0x2E	XOR	X			Next operation from instruction space
0x2F	HLT	X			Next operation from instruction space
0x1020	RTS	X	X		Return address read from workspace
0x1021	RTI	X	X		Restore CUP state from workspace

As seen from the table, the majority of MiniCPU-S instructions do not perform any operations other than fetching the next location in the instruction memory. This is the main reason for implementing the SPI hold function on the SPI ROM channel. Of the 33 defined instructions, only six (6) require that the ROM's address counter be reloaded when the program flow is changed. Of these six instructions, two of them are conditional branch instructions.

Thus, more than half of the instructions (19) of the MiniCPU-S require only sequential operation of the ROM channel. Eight (8) instructions, LDL/LDNL, STL/STNL, IN/INB, and OUT/OUTB, require the hold function while a read/write operation is conducted using the RAM channel, or one of the four I/O channels. Four of the remaining instructions, CALL, JMP, RTS, and RTI, will require access to the RAM channel. When that is complete, these four instructions require that the ROM's address counter be reloaded so the

instruction can be fetched. The last two instructions, BEQ and BLT, may or may not require a reload of the ROM address counter. If the condition is false, these instructions continue executing in a sequential fashion. Otherwise, the ROM address counter is reloaded, and that address is loaded in to the instruction/operand register in order to continue with the program.

SPI I/O CHANNEL MAPPING

As indicated previously, the IN/INB and OUT/OUTB instructions map to the ROM for the instruction fetch, and to one of four SPI I/O channels for their I/O function. The following table maps the lower four bits of the operand register to the four SPI I/O channels.

Op[3:0]	Type	Mode	IO[3]	IO[2]	IO[1]	IO[0]	Comment
0000	1	3				X	Normal I/O, CPOL – 1, CPHA – 1, Ch – 0
0001	1	3			X		Normal I/O, CPOL – 1, CPHA – 1, Ch – 1
0010	1	3		X			Normal I/O, CPOL – 1, CPHA – 1, Ch – 2
0011	1	3	X				Normal I/O, CPOL – 1, CPHA – 1, Ch – 3
0100	1	1				X	Normal I/O, CPOL – 0, CPHA – 1, Ch – 0
0101	1	1			X		Normal I/O, CPOL – 0, CPHA – 1, Ch – 1
0110	1	1		X			Normal I/O, CPOL – 0, CPHA – 1, Ch – 2
0111	1	1	X				Normal I/O, CPOL – 0, CPHA – 1, Ch – 3
1000	2	3				X	Fast I/O, CPOL – 1, CPHA – 1, Ch – 0
1001	2	3			X		Fast I/O, CPOL – 1, CPHA – 1, Ch – 1
1010	2	3		X			Fast I/O, CPOL – 1, CPHA – 1, Ch – 2
1011	2	3	X				Fast I/O, CPOL – 1, CPHA – 1, Ch – 3
1100	2	1				X	Fast I/O, CPOL – 0, CPHA – 1, Ch – 0
1101	2	1			X		Fast I/O, CPOL – 0, CPHA – 1, Ch – 1
1110	2	1		X			Fast I/O, CPOL – 0, CPHA – 1, Ch – 2
1111	2	1	X				Fast I/O, CPOL – 0, CPHA – 1, Ch – 3

Type 1 devices are classified as normal SPI I/O devices, and type 2 devices are classified as fast SPI I/O devices. Normal SPI I/O devices are characterized by transfer cycles that use a command byte, followed by an address byte (or more), and that header followed by one or more data (writes), or dummy (reads) bytes. Unlike the fast SPI I/O devices, normal SPI I/O devices do not enable their MISO outputs until the after the command and address bytes. Fast SPI I/O devices, like DACs and ADCs, are characterized by transfer cycles where the data transmitted on MOSI is matched by data simultaneously received on MISO. In the case of a fast SPI I/O device like a DAC, the input data is generally discarded, and for a fast SPI I/O device like an ADC, the output data is generally the channel select and control command while the received data is the data from the previous conversion operation. Unlike normal SPI I/O devices, SPI transfer cycles for fast SPI I/O devices are initiated and terminated for each cycle, i.e. there is no equivalent to the continuous read/write cycle of a normal SPI I/O device.

SPI Mode 0 and SPI Mode 3 are described in terms of the SPI Clock's idle phase (CPHA) and clock polarity (CPOL). It is a generally accepted convention to specify the

SPI Mode as {CPOL, CPHA}. Thus, SPI Mode 1, {0, 0}, means that CPHA is 1, and CPOL is 0, and SPI Mode 3, {1, 1}, means that CPHA is 1, and ClkPol is 1.

For the SPI clock, a CPHA of 0 indicates an idle level of 0, and a CPHA of 1 indicates an idle level of 1. The SPI clock polarity determines whether the SPI clock changes state at the start of the first bit cell following the assertion of the SPI slave select (SS) signal. Therefore, a CPOL of 0 indicates that no change is made from the clock idle state at the beginning of an SPI transfer cycle, and a CPOL of 1 indicates that the clock changes state from the idle state of the clock at the beginning of the SPI cycle.

An SPI device shifts its output data, i.e. propagates the SPI data, on the opposite edge of the SPI clock that is specified by the SPI Mode setting. An SPI Mode 3 device samples its input on the rising falling edge of the SPI clock, and shifts its output data on the falling edge. Because the CPOL is a 1 and CPHA is a 1 for Mode 1, there is a high to low transition at the beginning of the fist bit cell because the clock idle level is 1, so that the rising edge of the SPI clock occurs in the middle of each bit cell. In contrast, an SPI Mode 1 device shifts its output on the falling edge of the SPI clock because there is no transition at the beginning of the first bit cell (CPOL = 0) and the clock idle level is a 1 (CPHA = 1). (**Note:** *a transition always occurs in the middle of a bit cell.*)

The following figure illustrates the two SPI modes that the MiniCPU-S uses for SPI transfers.

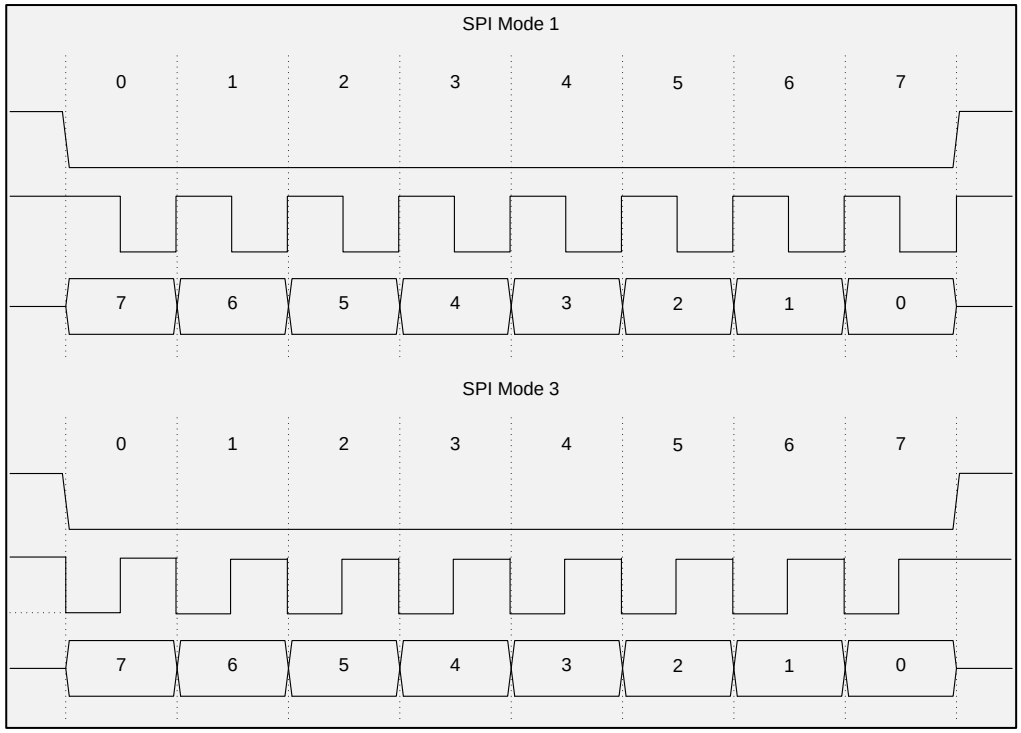

Figure 1: MiniCPU-S Supported SPI Modes.

EXPECTED INSTRUCTION SPI CYCLE TIMES

Assuming a 24-bit address, the following table defines the number SPI cycles, ROM, RAM, and IO cycles, required for each MiniCPU-S instruction:

Table 2: MiniCPU-S Instruction Execution Cycles (SPI cycles).

Op Code	Mnemonic	ROM (Fetch)	RAM	IO	ROM (Next)
0x0-	PFX	8	-	-	-
0x1-	NFX	8	-	-	-
0x2-	EXE {Indirect}	8	-	-	-
0x3Z	LDC 0x000Z	8	-	-	-
0x0Y3Z/0x1Y3Z	LDC 0x00YZ/0xFFyZ	16	-	-	-
0x0W0Y3Z/0x1W0Y3Z	LDC 0x0WYZ/0xFwYZ	24	-	-	-
0x0U0W0Y3Z	LDC 0xUWYZ	32	-	-	-
0x4Z	LDL 0x000Z	8	49	-	-
0x0Y4Z/0x1Y4Z	LDL 0x00YZ/0xFFyZ	16	49	-	-
0x0W0Y4Z/0x1W0Y4Z	LDL 0x0WYZ/0xFwYZ	24	49	-	-
0x0U0W0Y4Z	LDL 0xUWYZ	32	49	-	-
0x5Z	LDNL 0x000Z	8	49	-	-
0x0Y5Z/0x1Y5Z	LDNL 0x00YZ/0xFFyZ	16	49	-	-
0x0W0Y5Z/0x1W0Y5Z	LDNL 0x0WYZ/0xFwYZ	24	49	-	-
0x0U7W0Y5Z	LDNL 0xUWYZ	32	49	-	-
0x6Z	STL 0x000Z	8	58	-	-
0x0Y6Z/0x1Y6Z	STL 0x00YZ/0xFFyZ	16	58	-	-
0x0W0Y6Z/0x1W0Y6Z	STL 0x0WYZ/0xFwYZ	24	58	-	-

Op Code	Mnemonic	ROM (Fetch)	RAM	IO	ROM (Next)
0x0U0W0Y6Z	STL 0xUWYZ	32	58	-	-
0x7Z	STNL 0x000Z	8	58	-	-
0x0Y7Z/0x1Y7Z	STNL 0x00YZ/0xFFyZ	16	58	-	-
0x0W0Y7Z/0x1W0Y7Z	STNL 0x0WYZ/0xFwYZ	24	58	-	-
0x0U0W0Y7Z	STNL 0xUWYZ	32	58	-	-
0x8-	IN {0, Md, Ch}	8	-	33	-
0x9-	INB {0, Md, Ch}	8	-	25	-
0xA-	OUT {0, Md, Ch}	8	-	33	-
0xB-	OUTB {0, Md, Ch}	8	-	25	-
0xCZ	BEQ 0x000Z	8	-	-	32/-
0x0YCZ/0x1YCZ	BEQ 0x00YZ/0xFFyZ	16	-	-	32/-
0x0W0YCZ/0x1W0YCZ	BEQ 0x0WYZ/0xFwYZ	24	-	-	32/-
0x0U0W0YCZ	BEQ 0xUWYZ	32	-	-	32/-
0xDZ	BLT 0x000Z	8	-	-	32/-
0x0YDZ/0x1YDZ	BLT 0x00YZ/0xFFyZ	16	-	-	32/-
0x0W0YDZ/0x1W0YDZ	BLT 0x0WYZ/0xFwYZ	24	-	-	32/-
0x0U0W0YDZ	BLT 0xUWYZ	32	-	-	32/-
0xEZ	JMP 0x000Z	8	-	-	32
0x0YEZ/0x1YEZ	JMP 0x00YZ/0xFFyZ	16	-	-	32
0x0W0YEZ/0x1W0YEZ	JMP 0x0WYZ/0xFwYZ	24	-	-	32
0x0U0W0YEZ	JMP 0xUWYZ	32	-	-	32
0xFZ	CALL 0x000Z	8	58	-	32
0x0YFZ/0x1YFZ	CALL 0x00YZ/0xFFyZ	16	58	-	32
0x0W0YFZ/0x1W0YFZ	CALL 0x0WYZ/0xFwYZ	24	58	-	32
0x0U0W0YFZ	CALL 0xUWYZ	32	58	-	32
0x20	CLC	8	-	-	-
0x21	SEC	8	-	-	-
0x22	TAW	8	-	-	-
0x23	TWA	8	-	-	-
0x24	DUP	8	-	-	-
0x25	XAB	8	-	-	-
0x26	POP	8	-	-	-
0x27	RAS	8	-	-	-
0x28	ROR	8	-	-	-
0x29	ROL	8	-	-	-
0x2A	ADC	8	-	-	-
0x2B	SBC	8	-	-	-
0x2C	AND	8	-	-	-
0x2D	ORL	8	-	-	-
0x2E	XOR	8	-	-	-
0x2F	HLT	8	-	-	0
0x1020	RTS	8	49	-	32
0x1021	RTI	8	113	-	32

The previous table shows all of the expected SPI cycles required for each instruction. The NFX, FPX, and EXE direct instructions are shown diagonally cross-hatched. This indicates that these direct instructions support other instructions, so that their execution time is not pertinent. For example, NFX and PFX are used to assemble/construct constants,

and relative offsets in the operand register in support of many of the direct instructions. Therefore, instead of considering the NFX and PFX instructions as separate, standalone instructions, they should be considered as helpers to other instructions, and so their execution time should be included as part of the execution time of the instruction which it is helping or forming (RTS/RTI).

For example, the CALL instruction uses the contents of the operand register as a relative offset to the instruction pointer in order to determine the address of the subroutine to which to branch. Depending on the relative distance between the current instruction pointer and the subroutine, no PFX/NFX is required, or between one and three NFX/PFX instructions are required to set up the operand register appropriately before the CALL instruction is fetched from instruction memory.

Since there is only a single SPI interface (each of the six SPI channels has a separate slave select), the instruction fetch and execution phases are difficult to overlap. The table shows that 8 SPI cycles are required for a CALL instruction branching to a subroutine located between 0 and 15 locations from the instruction pointer value (after the instruction fetch). If the relative offset is between -256 and +255 then a single NFX/PFX instruction is required to initialize the operand register before the CALL instruction, which will include the least significant four bits of the offset. As the distance increases, additional NFX/PFX instructions precede the CALL instruction opcode. Thus, between 8 and 32 SPI cycles are required to fetch the CALL instruction and set up the operand register.

Following the fetch of the CALL instruction, the SPI ROM Hold input is asserted, and the RAM channel is used write the write enable latch (9) and simultaneously decrement the workspace pointer. Following the write of RAM's write enable latch, the return address is written to the RAM at the location pointed to by the workspace pointer. This requires the writing of the RAM write command (8) the value of the workspace pointer (24), the value of the instruction pointer (16), and finally deasserting the RAM select, the ROM hold, and the ROM select signals (1). Thus, to save the return address in the workspace (RAM) requires 58 SPI cycles.

Since the program flow is not sequential, the ROM's internal address counter must be reinitialized by starting a new read cycle. The ROM select is reasserted, a read command (8) and a new address (24) are transmitted to the ROM in order to fetch the op code at the destination address of the CALL instruction. Thus, to set up the next instruction fetch cycle from ROM, 90 SPI cycles are required. The cost of the 8 SPI cycles required to fetch the next instruction's op code from the ROM is attributed to the CALL instruction.

If the instruction execution and next instruction fetch can be overlapped, then the column labeled **ROM (Next)** shows no cost for that operation. With the exception of the RTS/RTI instructions, all currently defined indirect instructions share this characteristic, and sequential fetches from the ROM can be made.

MINICPU-S EXECUTION UNIT (EU)

TheExecution Unit (EU) of the MiniCPU-S processor provides the control state machine, the instruction register (IR), and the SPI interface. The control state machine is tasked with controlling the operations of the instruction register, the SPI interface, instruction pointer, the workspace pointer, the operand register, and the Serial ALU.

The MiniCPU-S EU, MiniCPU-S PCU and the MiniCPU-S Serial ALU are expected to be implemented in separate CPLDs. This allows the EU and the Serial PCU and the Serial ALU to target a medium density XC9572 CPLD instead of the higher density XC95144 CPLD. XC9572 CPLDs are available in 44-pin and 84-pin PLCC packages. A slightly higher density component, the XC95108 CPLD, is also available in the 84-pin PLCC package. (**Note:** *the MiniCPU-S' Serial PCU and Serial ALU components have each been fitted into their own XC9572-xPC44 CPLD.*)

The implementation of the MiniCPU-S processor targeted to CPLDs is a 16-bit processor. For SPI ROM/RAM. the MiniCPU-S SPI clock is one half (½) of the processor's clock frequency. For SPI I/O, the MiniCPU-S SPI clock is one fourth (¼) of the processor's clock frequency. The SPI memory devices being targeted by the MiniCPU-S are devices which may be able to operate at 40 MHz. The maximum frequency that the MiniCPU-S' Serial ALU module can support is approximately 71.428 MHz (*when implemented in an XC9572-7PC44 CPLD*). Therefore, the MiniCPU-S will not attempt to implement a 1:1 ratio between the SPI clock and the processor clock (*even though an implementation of the serial ALU module in a+3.3V XC9572XL CPLD can support operation at 40 MHz*).

Furthermore, the 16-bit width of the MiniCPU-S PCU and Serial ALU registers coupled with the 8-bit instruction width means that most instructions will require 16 processor clocks to complete. This means that increasing the read speed of the serial ROM/RAM devices using a 1:1 clock ratio between the processor clock and the SPI clock will complete the fetch of the next instruction (8 bits) at least 8 clock cycles prior to the completion of the currently executing instruction. Furthermore, and more importantly, the next instruction must be held in an additional holding register. Given the lack of registers in CPLDs, the additional holding register is not a desirable feature.

The appeal of the additional performance to be gained by reading the memory with the SPI clock equal to the processor clock frequency is tempered by the need for additional registers. Since the objective is to stay within some combination of 72 or 108 macro cell devices, additional registers is not a viable option to consider when CPLDs are the target technology for the MiniCPU-S. Therefore, in a CPLD, the MiniCPU-S will use a basic 2:1 ratio between the SPI clock frequency and system clock frequency for accesses to SPI memory devices and a 4:1 ratio for accesses to SPI I/O devices.

EU STATE MACHINE

The expected execution times table in the preceding section and the SPI clock frequency discussion in this section essentially define the basic execution cycle of the MiniCPU-S in a CPLD. Limiting the number of temporary registers implemented in the CPLD means that the fetch of the next instruction cannot be overlapped with the execution of the current instruction. As the next instruction byte is read from memory, the current instruction could be executed, but that requires a temporary register in which to build the next instruction while the current instruction register's value is used to control the serial ALU module. Ideally, the control word for the serial ALU module would be written to that module, and the new instruction shifted directly into the instruction register. However, the serial ALU module is at 100% utilization in terms of macrocells, so no further growth in either macrocells or registers is allowed, or the implementation target would need to be changed.

EU SPI OPERATIONS

The PCU's control state machine can therefore be defined as consisting of several independent operations:

1) Terminate memory hold;
2) Memory read address transfer;
3) Memory read (8) and assert hold (instruction fetch);
4) Memory read (8 or 16);
5) Memory Write Enable Latch;
6) Memory write address transfer;
7) Memory write (8 or 16);
8) Type 1 SPI I/O read (8 or 16);
9) Type 1 SPI I/O write (8 or 16);
10) Type 2 SPI I/O data exchange (8 or 16).

The first seven of these operations are directed toward the instruction and data memory devices, and the last three operations support the SPI I/O capabilities of the MiniCPU-S. Of the memory device operations, the first four operations support memory reads and the last three operations support memory writes. Each of the operations will be discussed in the following subsections

Terminate Memory Hold

The Terminate Memory Hold SPI operation is used to deassert the HOLD signal on the memory device to which a non-sequential read/write operation is directed. If the MiniCPU-S execution engine determines that a non-sequential read/write operation is required, and the HOLD signal for the corresponding memory device is asserted, then this operation must be performed. The operation terminates the current SPI transfer cycle to the device, and allows a new SPI transfer cycle to be initiated to the device.

SPI memory devices generally require each SPI cycle to be initiated by the assertion of a device-specific slave select signal, and terminated by the deassertion of that signal. Furthermore, an SPI command must be issued for each SPI transfer cycle, followed by any required parameters and any required output data (writes) or any required dummy data (reads). If each 8-bit instruction fetch followed this sequence of operations, then the overhead associated with each instruction fetch would be 24 or 32 (depending of the number of bytes required to transfer the memory address) clock cycles, and then an additional 8 cycles would be required to read the instruction byte. The high overhead associated with the SPI instruction memory read cycles would reduce the performance of the MiniCPU-S.

However, most SPI memory devices provide a mechanism for suspending the SPI transfer cycle, and initiating a transfer cycle using the same SPI I/O pins (SCK, MOSI, and MISO) for a transfer using another device on the same bus. This mechanism uses a device-specific control signal generally known as HOLD. Like the device-specific slave select signal, the master must assert HOLD when it needs to interrupt the current SPI transfer cycle using that device and initiate another SPI transfer cycle on another device.

An SPI device which supports this feature will ignore transitions on SCK, MOSI, and MISO while the slave select and HOLD signals are both asserted. In this manner, any number of SPI transfer cycles can be initiated and terminated, and then the SPI master can continue the interrupted SPI transfer cycle by simply releasing HOLD and continuing the cycle. The only restriction to using this feature is that slave select and HOLD must be dedicated to the SPI device, and must not deassert at any time while SPI transfer cycles are being performed with other SPI devices.

Thus, the MiniCPU-S memory interface will support HOLD for both memory devices. This will allow either device to be used for instruction memory. The PCU state machine will assert HOLD after the read of an instruction from whichever device the instruction was fetched. This will allow the MiniCPU-S execution engine to execute the instruction, and the remove the HOLD and fetch the next instruction without transmitting a read command and read address to the device.

Memory Read Address Transfer

When a new memory cycle is required, the MiniCPU-S execution engine uses this operation to assert the slave select of the appropriate memory device, transmit the SPI read command code (0x03), and the appropriate 16-bit memory address. This operation flows into either the *Memory Read and Assert Hold* operation, or the *Memory Read* operation.

Non-sequential reads of either instructions or data require the MiniCPU-S execution state machine to terminate an active transfer cycle or to initiate a new transfer cycle. Termination of a transfer cycle is required if HOLD and slave select are asserted, and the next access is to the same device and is not sequential with the previous access.

In the event that a non-sequential access to a device with HOLD asserted, the execution engine will first use the *Terminate Memory Hold* operation to terminate the active transfer cycle, and then a new *Memory Read Address Transfer* operation will be performed so that the internal address pointer of the device is properly set. The execution engine will chain these operations together whenever a program branch occurs, or a read operation occurs to a non-sequential location. (**Note:** *the CPLD implementation of the MiniCPU-S is incapable of storing the current address and the previous address because of the resource utilization issues present. Therefore, reading an operand from the same memory device from which the last instruction was read will result in the termination of that memory read cycle, and starting another memory read cycle. When this situation arises, the performance of the MiniCPU-S will be the same as if the memory HOLD function was unimplemented or unused. For maximum performance, one memory device should be used for instructions and the other for data, and it is for this reason that the MiniCPU-S provides support for two memory devices.*)

SPI devices typically use an MSB-first shift cycle. For the MiniCPU-S, this means that during the 8 cycles during which the SPI read command is transferred, the MiniCPU-S must compute the address because binary arithmetic must be performed LSB-first. Since the processor is operating at twice the SPI shift clock rate, the address computation can be completed during the 8 SPI cycles used to write the read command to the selected device.

Similarly, during normal sequential instruction processing, the reading of 8 bits from the instruction memory device allows the instruction pointer to be incremented. In the case of conditional branches, if the condition is not true, then a program branch is not required, and the next sequential location is read. During those 8 SPI read cycles, the instruction pointer can be incremented to match the internal memory pointer of the device. If the condition is true, a program branch is required which also requires the termination of the memory hold and the initiation of another memory cycle, and the new instruction pointer is computed during the transmittal of the SPI read command. Unconditional branches are handled in the same manner as conditional branches when the condition is true.

Subroutine calls and returns are handled in a similar manner, but the pushing and popping of the return address from the workspace requires two memory read operations: one to the workspace (data) memory, and the other to the instruction memory.

Memory Read and Assert HOLD

This operation follows the *Memory Read Address Transfer* operation when the execution engine reads an instruction from memory. The MiniCPU-S execution engine never uses this operation when operands are read from memory.

Memory Read

This operation always follows a *Memory Read Address Transfer* when operands are read from memory. The operation may be used for either reading a byte or a word from the

selected memory device. This operation always terminates the SPI transfer cycle of the selected device. This operation outputs either 8 or 16 dummy bits. While the dummy bits are being output on MOSI, the read data is being shifted in on the selected clock edge from MISO.

Memory Write Enable Latch

This operation must precede any write operation if the memory device is an SPI SEEPROM, MRAM, or FRAM: these devices use the SPI WREN command (0x06) to enable writes to their memory arrays. In SEEPROMs, when the write data transfer cycle completes, the selected page is programmed with the data written to the internal page buffer during the transfer cycle. The high speed write/programming cycle of MRAMs and FRAMs does not require a page buffer, and the write/programming cycle completes within the period of an 8-bit transfer cycle.

These three types of SPI memory devices require the WREN command to be transmitted as a single byte SPI transfer cycle in order to allow the write operation, which is expected follow immediately, to write/program one or two bytes into the memory array of the selected device.

The MiniCPU-S supports these three types of devices for one memory device, and supports either MRAM/FRAM or SRAMs for the other. Two I/O pins on the MiniCPU-S PCU determine the type of SPI memory devices attached. A logic zero indicates that an MRAM/FRAM device is attached to the PCU. A logic one indicates that the device is an SEEPROM or an SRAM. If the MiniCPU engine determines that an SPI SEEPROM or and SPI SRAM device is attached to the selected memory, then the M*emory Write Enable Latch* operation is bypassed and the execution engine proceeds directly to the *Memory Write Address Transfer* operation. Since the SEEPROM requires the WREN command to enable memory array programming, this characteristic of the memory interface means that the MiniCPU-S execution engine will be unable to program SEEPROMs. However, given that SEEPROMs require polling of their internal status register to determine when a programming cycle is complete, this limitation is not particularly onerous, and merely means that an SEEPROM attached to the MiniCPU-S is simply treated as a ROM.

Memory Write Address Transfer

This operation must precede each memory write. It functions in the same manner as the *Memory Read Address Transfer* operation previously described, except that the SPI command code is 0x02 instead of 0x03.

Memory Write

This operation writes 8 or 16 bits using MOSI to the selected device. At the completion of the required transfer, the SPI transfer cycle is terminated.

Type 1 SPI I/O Read

This operation is used by the MiniCPU-S execution engine to read from a normal SPI I/O device. This operation is only used by the execution engine to execute an IN or an INB instruction. (*Refer to the section describing the MiniCPU-S' SPI I/O support for a more detailed description of how this SPI operation will perform its function.*)

Type 1 SPI I/O Write

This operation is used by the MiniCPU-S execution engine to write to a normal SPI I/O device. This operation is only used by the execution engine to execute an OUT or an OUTB instruction. (*Refer to the section describing the MiniCPU-S' SPI I/O support for a more detailed description of how this SPI operation will perform its function.*)

Type 2 SPI I/O Data Exchange

This operation is used by the MiniCPU-S execution engine to write data to and read data from a fast SPI I/O device. This operation is only used by the execution engine to execute an IN/OUT or an INB/OUTB instruction. (*Refer to the section describing the MiniCPU-S' SPI I/O support for a more detailed description of how this SPI operation will perform its function.*)

EU STATE MACHINE STATES

The PCU state machine generally requires only a limited number of top level (macro) states: (1) instruction fetch, and (2) execute. Within these two top level (macro) states there are a number of second level (micro) states. In general, these second level (micro) states correspond roughly to the SPI operations described in the previous subsection. This section will describe the PCU state machine in terms of the macro and micro states, with the micro states being shown in the pseudo code as "subroutines".

MAPPING MINICPU-S INSTRUCTIONS TO EU SPI OPERATIONS

As in a preceding section, the 33 MiniCPU-S instructions are onto the PCU SPI operations described in subsection – *EU SPI Operations*. The purpose of the mapping is to help define the MiniCPU-S' execution engine, i.e. the PCU state machine. Before tabulating the mapping of SPI operations onto each MiniCPU-S instruction, several important concepts need to be more fully defined.

First, memory addresses are signed. There are two memory chip selects, and in general, each is allocated half of the total address space of the MiniCPU: 32 kB each. The lower portion of memory is defined as occupying the region from 0x8000 (-32,768) to 0xFFFF (-1), and this address range is generally considered to be allocated to the workspace, i.e. data memory. The workspace pointer (W) is initialized to 0x0000 on reset. This means that the return address of the first subroutine will be stored in location 0xFFFE, since W is pre-decremented (by 2) when a CALL instruction is executed. It also means that the

initial program will not have access to any workspace storage locations. Therefore, if the boot program requires any storage, it will need to adjust the initial value of W to allocate any workspace storage that the boot program may require. The upper portion of memory is defined as occupying the region from 0x0000 to 0x7FFF, and this address range is generally considered to be allocated as instruction memory. The instruction pointer (I) is initialized to 0x0000 on reset. This defines the starting address of the bootstrap program.

Thus, the workspace grows towards 0x8000 and the instruction space grows towards 0x7FFF. Furthermore, to make the most effective use of the dual slave selects for SPI memory devices, a program's instructions and data need to come different devices so that the HOLD function can be used with one device for improved sequential fetches of instructions while the other uses the full read/write address transfer cycles to provide random access to data.

Second, the assertion of the HOLD FF for each of the two memory devices will indicate which memory device provided the last instruction. Further, only one HOLD FF will be asserted at any time. This means that the ***Terminate Memory Hold*** operation is always executed if a program branch operation (*unconditional jumps, conditional branches with a true condition, subroutine calls and subroutine returns*) is targeted to a device whose HOLD FF is asserted. If a data read/write operation is to be directed to a memory device whose HOLD FF is asserted, then the ***Terminate Memory Hold*** operation will be performed before the address transfer cycle to the target memory device is performed. Furthermore, if a sequential instruction fetch is directed to a device whose HOLD FF is not asserted, then an address transfer cycle must be performed before the instruction may be read from the device. (**Note:** *this last memory interface requirement is a consequence of the program reading data from, or writing data to the memory device from which instructions are being fetched. Since instruction memory and data memory are mapped uniformly onto the memory space of the MiniCPU-S, the programmer is required to segregate instructions and data in order to provide maximum performance using SPI memory interface.*)

Code	Mnemonic	
0x0-	PFX	
0x1-	NFX	
0x2-	EXE	
0x3-	LDC	
0x4-	LDL	
0x5-	LDNL	
0x6-	STL	
0x7-	STNL	
0x8-	IN	
0x9-	INB	
0xA-	OUT	

Code	Mnemonic	
0xB-	OUTB	
0xC-	BEQ	
0xD-	BLT	
0xE-	JMP	
0x0-	CALL	if(Hold) Terminate_Memory_Hold()
0x20	CLC	
0x21	SEC	
0x22	TAW	
0x23	TWA	
0x24	DUP	
0x25	XAB	
0x26	POP	
0x27	RAS	
0x28	ROR	
0x29	ROL	
0x2A	ADC	
0x2B	SBC	
0x2C	AND	
0x2D	ORL	
0x2E	XOR	
0x2F	HLT	
0x1020	RTS	
0x1021	RTI	

INSTRUCTION SET USAGE FREQUENCY

The instruction usage frequency is not yet known. Previous work by Inmos in the area of instruction set encoding for this type of 0 address machine ensured that many of the same instructions that are defined for the MiniCPU-S were encoded as single byte instructions. The MiniCPU-S has followed the example set by Inmos in defining the direct instructions and the indirect instructions.

Given the architecture of the MiniCPU-S and its target implementation technology, load (LDC/LDL/LDNL) and input (IN/INB) instructions must be efficiently implemented. Like accumulator-based, one (1) address architectures such as the Western Design Center 65C02, the Motorola MC6800, the Intel 8080, 8085, 8048, and 8051, the Zilog Z80, and the Microchip PIC16C5x, the MiniCPU-S must be able to efficiently load and unload its stack during operation. The ability of the MinCPU-S ALU register stack to support chained computations using Reverse Polish Notation, means that the number of times that partial results in its ALU stack have to be unloaded to memory is reduced. This feature compares favorably to the loading and unloading required of the accumulator in many accumulator-based processors. However, the limited depth of the MiniCPU-S ALU stack does mean that it is very likely to require unloading at a rate higher than would be required if the stack was deeper, or if a 2 or 3 address register-based architecture was used instead of the stack-based, 0 address architecture selected.

It is expected that the MiniCPU-S will make frequent use of the LDL/LDNL instructions, less frequent use of the STL/STNL instructions. It is generally accepted that that the load/store architectures used in reduced instruction set computers demonstrate a load to store instruction ratio in the range of 3:1 to 5:1. The simple nature of the MiniCPU-S instruction set and addressing modes means that the ratio of LDL/LDNL instructions to STL/STNL instructions should be about or slightly less than 3:1. Industry standard SPI Flash program and/or FRAM/MRAM data memory devices require that a set write enable latch command (0x06) must be issued before a write operation will be accepted and executed. This means that for an SPI memory device, whether a Flash EPROM or an FRAM/MRAM device, each write operation has an additional overhead equivalent to 9 SPI clock periods during which the set write enable latch command (0x06) must be transmitted to the SPI memory device in addition to the normal overhead of 24/32 SPI clock periods which consist of the 8-bit write command and the 16/24-bit address cycles.

Memory Read Address Transfer

This operation asserts slave select, and then sends the SPI read command code, 0x02, followed by the appropriate 16-bit address. At the completion of this operation, the execution engine continues with an 8-bit or a 16-bit read memory operation.

The current plan is to use two 32 kB MRAM/FRAM devices for the memory of the MiniCPU-S. These devices use a two byte address. Most SPI devices use an MSB first format