XAPP213 (v1.1) October 4, 2000

8-Bit Microcontroller for Virtex Devices

Author: Ken Chapman

Summary

The Constant (k) Coded Programmable State Machine (KCPSM) presented in this application note is a fully embedded 8-bit microcontroller macro for the Virtex™ and Spartan®-II devices. The module is remarkably small at just 35 CLBs, less than half of the smallest Spartan XC2S15 device, and virtually free in an XCV2000 device by consuming less than 0.37% of the device CLBs.

This KCPSM provides 49 different instructions, 16 registers, 256 directly and indirectly addressable ports, and a maskable interrupt at 35 million instructions per second (MIPs). This performance exceeds that of traditional discrete microcontroller devices, making the KCPSM a cost-attractive solution for data processing as well as control algorithms.

Fully embedded including the program memory, the KCPSM can be connected to many other functions and peripherals tuned to a specific design. Processing distributed over multiple KCPSM processors within a single device is suitable for applications such as neural networks.

Introduction

Size constraint is a compelling factor for using the KCPSM module. An amazingly small microcontroller can be designed to occupy under 35 Virtex CLBs, less than 10% of the smallest Virtex device (XCV50) and less than 0.6% of an XCV1000 device. Besides the small logic utilization, a single block RAM is used to form a ROM for storage of up to 256 programming instructions. Even with such size constraints, the performance is maintained in the range of 25 MIPs to 35 MIPS, depending on the device speed grade. Figure 1 is a block diagram of a KCPSM module.

The Virtex KCPSM modules require no external support and provide a flexible environment for other logic connections into the KCPSM module.

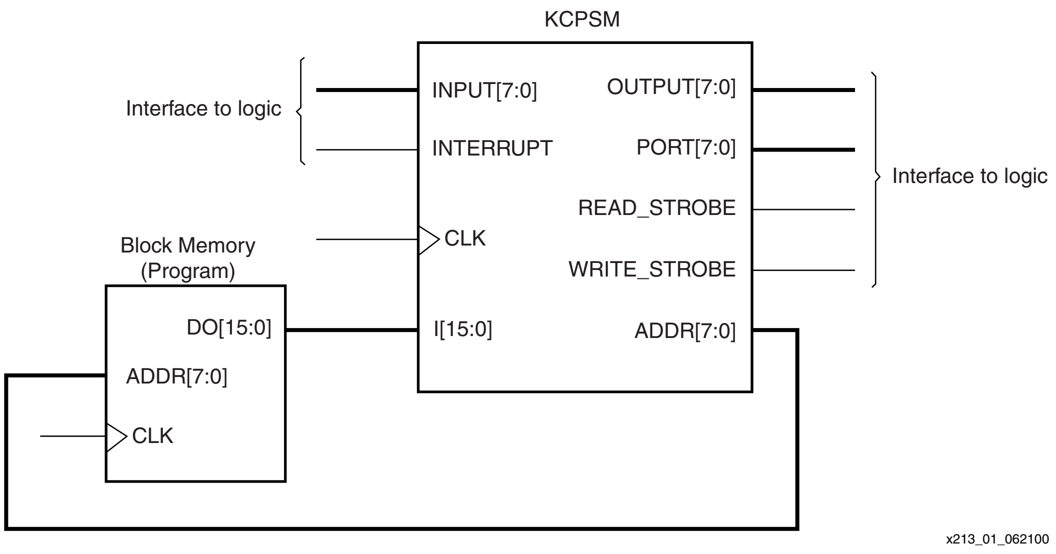


Figure 1: **KCPSM Module Block Diagram**

Understanding KCPSM

The KCPSM module is a soft macro for place and route tools to merge with the design logic. Figure 2 is a plot from the EPIC viewer showing the macro in isolation in the XCV50 device. The 35 CLBs consume less than 10% of an XCV50 device.

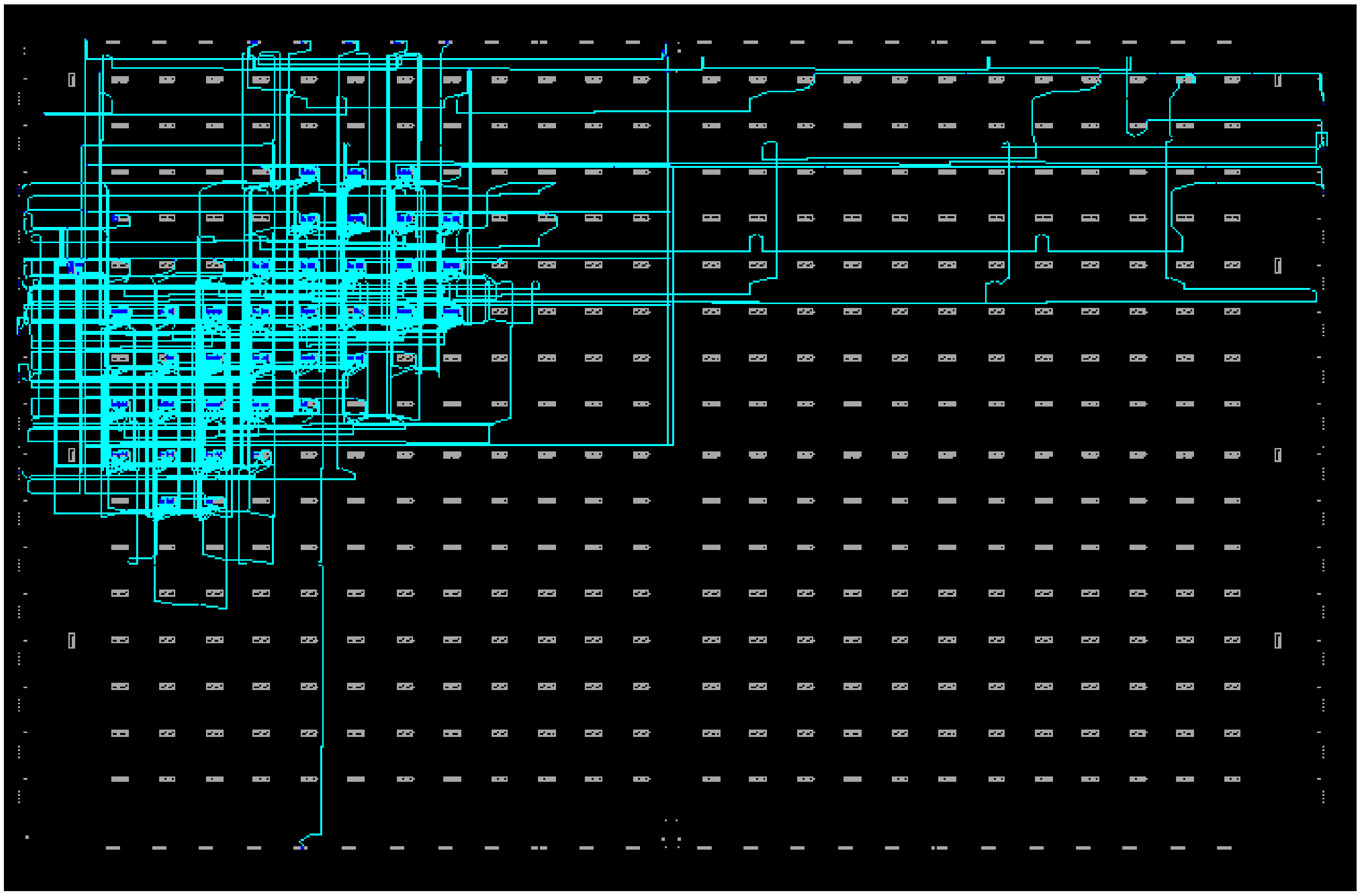


Figure 2: **EPIC View of a KCPSM Macro in an XCV50 Device**

KCPSM Architecture

Figure 3 shows the KCPSM architecture.

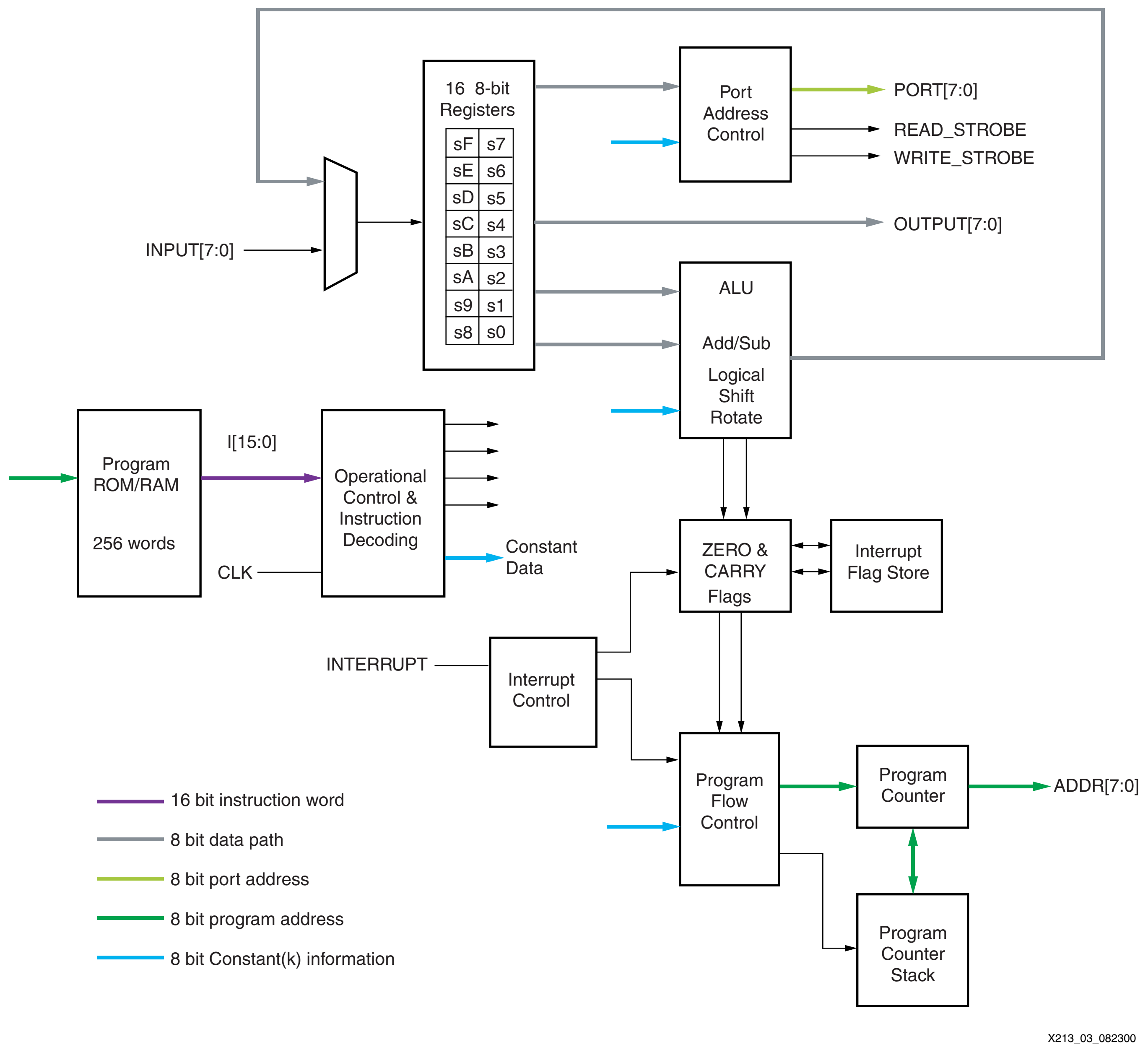


Figure 3: **KCPSM Architecture**

KCPSM Feature Set

General Purpose Registers

The feature set includes 16 general purpose 8-bit registers, s0 to sF. The register operations are completely flexible; no registers are reserved for special tasks or given priority over other registers.

ALU

The Arithmetic Logic Unit (ALU) provides all the simple operations expected in an 8-bit processing unit.

All operations are performed using an operand provided by any register. The result is returned to the same register. For operations requiring a second operand, a second register is specified or a constant 8-bit value is embedded in the instruction. The ability to specify any constant value with no penalty to the program size or to its performance enhances the simple instruction set. To clarify, the ability to "ADD 1" is the equivalent of a dedicated INCREMENT operation. For operations requiring more than 8 bits, addition and subtraction operations include an option to carry. Bit-wise operators (LOAD, AND, OR, XOR) provide the ability to manipulate and test values. There is also a very comprehensive Shift and Rotate group.

Flags Program Flow Control

The ALU operation results affect the ZERO and CARRY flags. Using conditional and non-conditional program flow control instructions, this information determines the execution sequence of the program. `JUMP` commands specify absolute addresses within the program space.

CALL and RETURN commands provide subroutine facilities for commonly used sections of code. A CALL command is made to a specified absolute address, while a program counter stack preserves the return address. The stack provides for a nested CALL with a depth of up to 15 levels, adequate for the program size supported.

Input/Output

The KCPSM has 256 input ports and 256 output ports. An 8-bit address value provided on the PORT bus together with a READ or WRITE strobe signal indicates the accessed port. The port address is either supplied in the program as an absolute value, or specified indirectly as the contents of any of the 16 registers. Indirect addressing is ideal when accessing a block of memory constructed from block or distributed RAM.

During an INPUT operation, the value provided at the input port is transferred into any of the 16 registers. An input operation is indicated by a READ_STROBE output pulse. Although using this signal in the design input interface logic is not vital, it indicates that data has been acquired by the KCPSM.

During an OUTPUT operation, the contents of any of the 16 registers are transferred to the output port. A WRITE_STROBE output pulse indicates an output operation. This strobe signal is used in the design output interface logic, ensuring that only valid data is passed to external systems.

Interrupt

The process provides a single interrupt input signal. Using simple logic, multiple signals can be combined and applied to this one input signal. By default, the effect of the interrupt signal is disabled (masked) and is under program control to be enabled and disabled as required.

An active interrupt forces the KCPSM to initiate a "CALL FF" (i.e., a subroutine call to the last program memory location) for the designer to define a suitable course of action. Automatically, the interrupt process preserves the contents of the current ZERO and CARRY flags and disables any further interrupts. A special RETURNI command is used to ensure that the end of an interrupt service routine restores the status of the flags and controls.

Constant (k) Coded Values

The KCPSM is in many ways a machine based on constants. Constant values are specified for use in the following aspects of a program:

- Constant data value for use in an ALU operation
- Constant port address to access a specific piece of information or control logic external to the KCPSM
- Constant address values for controlling the execution sequence of the program

The KCPSM instruction set coding is designed to allow constants to be specified within any instruction word. Hence, the use of a constant carries no additional overhead to the program size or its execution. This effectively extends the simple instruction set with a whole range of "virtual instructions."

Constant Cycles

All instructions under all conditions execute over two clock cycles. When determining the execution time of a program, particularly when embedded into a real time situation, a constant execution rate is of great value.

Constant Program Length

The program length is 256 instructions, conforming to the 256 x 16 format of a single Virtex block RAM. All address values are specified as 8-bits contained within the instruction coding. The fixed memory size promotes a consistent level of performance from the module.

Using the KCPSM Macro

The KCPSM macro is provided in the form of an EDIF netlist. Figure 4 is a diagram of this macro. The Foundation software tool has a schematic feature to "create" a macro symbol from the netlist under the **Hierarchy** menu. The resulting symbol is editable. This symbol is inserted as a "black box" into any Virtex design. A block RAM is utilized to store the program code and must be connected to behave as a ROM. Figure 5 shows a RAM generated by the COREGen Module using the ***<name>*.coe** file obtained from the KCPSMBLE assembler. The assembler also provides an EDIF file for immediate use as a second "black box" for a block RAM to be both initialized and pre-connected in the ROM configuration. This alternative "black box" ROM is shown in Figure 6.

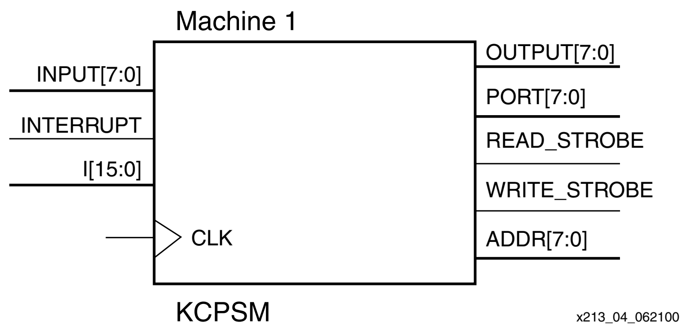


Figure 4: **KCPSM Macro**

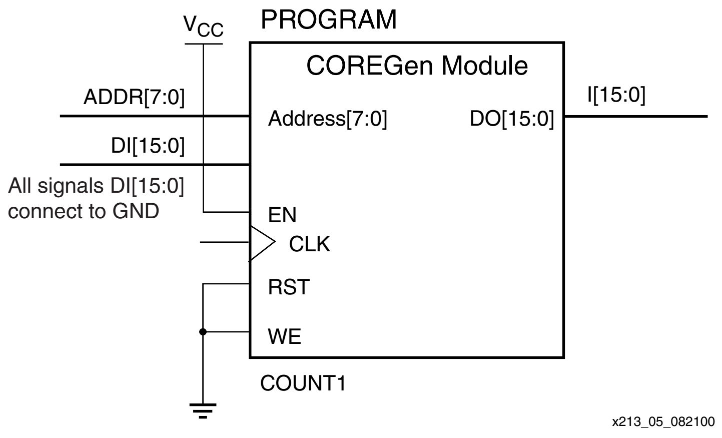


Figure 5: **RAM Generated by the Core Generator Module**

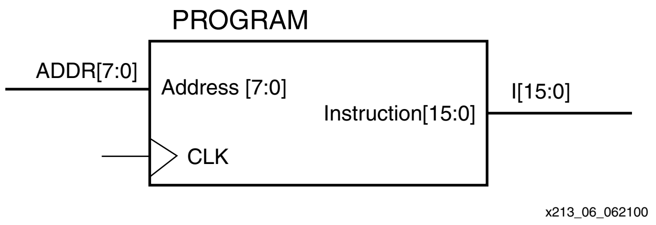


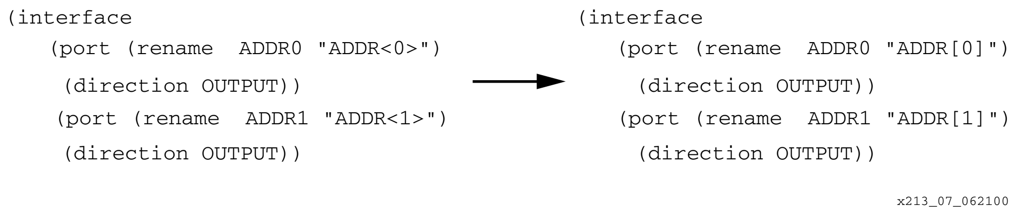
Figure 6: **Black Box ROM**

The KCPSM Macro in VHDL

The KCPSM macro and the block RAM are instantiated to reflect the diagrams in Figures 4, 5, and 6. In VHDL, the macro is instantiated using the following declaration.

```
component kcpsm is
     port map (          addr: out std_logic_vector(7 downto 0);
                            i: in  std_logic_vector(15 downto 0);
                        input: in  std_logic_vector(7 downto 0);
                       output: out std_logic_vector(7 downto 0);
                         port: out std_logic_vector(7 downto 0);
                  read_strobe: out std_logic;
                 write_strobe: out std_logic;
                    interrupt: in  std_logic;
                          clk: in  std_logic);
     end component;
```

The bus delimiter used within the EDIF file to specify bus pins might be incompatible with the HDL tool. Since EDIF is a simple text format, use a test editor to globally replace the "<" and ">" symbols with the appropriate brackets for the HDL tool. Figure 7 shows a small section of the EDIF converted to square brackets.



```
(interface                                 (interface
   (port (rename  ADDR0 "ADDR<0>")            (port (rename  ADDR0 "ADDR[0]")
    (direction OUTPUT))           ──►          (direction OUTPUT))
    (port (rename  ADDR1 "ADDR<1>")           (port (rename  ADDR1 "ADDR[1]")
    (direction OUTPUT))                        (direction OUTPUT))
```

x213_07_062100

Figure 7: **EDIF File Converted to Square Brackets**

Complete KCPSM Instruction Set

This section lists a complete instruction set representing all op-codes in hexadecimal.

1. "X" and "Y" refer to the definition of the storage registers "s" in range 0 to F.
2. "kk" represents a constant value in range 00 to FF.
3. "aa" represents an address in range 00 to FF.
4. "pp" represents a port address in range 00 to FF.

Program Control Group

```
81aa   JUMP aa
91aa   JUMP Z,aa
95aa   JUMP NZ,aa
99aa   JUMP C,aa
9Daa   JUMP NC,aa

83aa   CALL aa
93aa   CALL Z,aa
```

97aa	CALL NZ,aa
9Baa	CALL C,aa
9Faa	CALL NC,aa
8080	RETURN
9080	RETURN Z
9480	RETURN NZ
9880	RETURN C
9C80	RETURN NC

Note: Call and Return supports a stack depth of up to 15.

Shift and Rotate Group

Dx0E	SR0sX
Dx0F	SR1sX
Dx0A	SRXsX
Dx08	SRAsX
Dx0C	RR sX
Dx06	SL0sX
Dx07	SL1sX
Dx04	SLXsX
Dx00	SLAsX
Dx02	RL sX

Logical Group

0xkk	LOAD sX,kk
1xkk	AND sX,kk
2xkk	OR sX,kk
3xkk	XOR sX,kk
Cxy0	LOAD sX,sY
Cxy1	AND sX,sY
Cxy2	OR sX,sY
Cxy3	XOR sX,sY

Arithmetic Group

4xkk	ADD sX,kk
5xkk	ADDCY sX,kk
6xkk	SUB sX,kk
7xkk	SUBCY sX,kk
Cxy4	ADD sX,sY
Cxy5	ADDCY sX,sY
Cxy6	SUB sX,sY
Cxy7	SUBCY sX,sY

Input/Output Group

Axpp	INPUT sX,pp
Bxy0	INPUT sX,(sY)
Expp	OUTPUT sX,pp
Fxy0	OUTPUT sX,(sY)

Interrupt Group

80F0	RETURNI ENABLE
80D0	RETURNI DISABLE
8030	ENABLE INTERRUPT
8010	DISABLEINTERRUPT

Program Control Group

JUMP

Under normal conditions, the program counter (PC) increments to point to the next instruction (Figure 8). The address space is fixed to 256 locations (00 to FF hex), making the program counter 8-bits wide. The top of the memory is FF hex and will increment to 00.

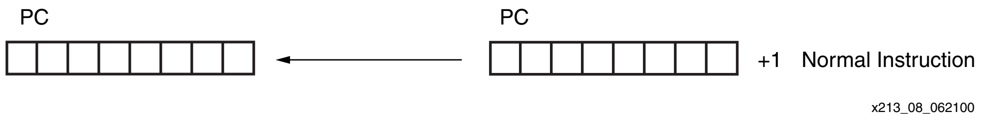


Figure 8: **Program Counter**

The JUMP instruction is used to modify the sequence by specifying a new address. However, the JUMP instruction can be conditional. A conditional JUMP is only performed if a test performed on either the ZERO flag or CARRY flag is valid. The JUMP instruction has no effect on the status of the flags (Figure 9).

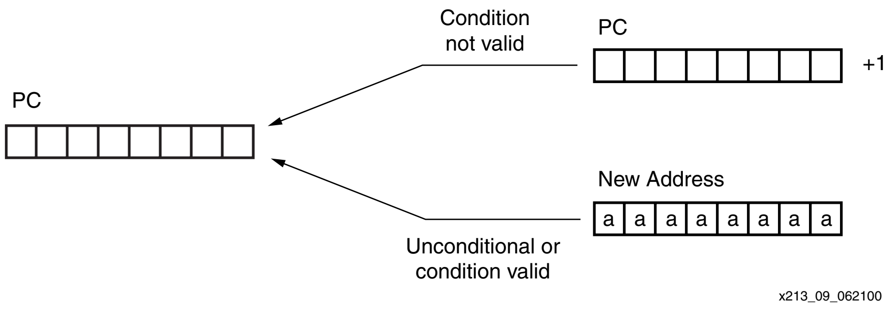


Figure 9: **JUMP Instruction**

Each JUMP instruction must specify the 8-bit address as a two digit hexadecimal value. The assembler supports labels to simplify this process.

```
81aa JUMP    aa      Unconditional JUMP to address "aa"
91aa JUMP Z,  aa      Conditional JUMP to address "aa" performed when ZERO
                      flag is set
95aa JUMP NZ, aa      Conditional JUMP to address "aa" performed when ZERO
                      flag is reset
99aa JUMP C,  aa      Conditional JUMP to address "aa" performed when CARRY
                      flag is set
9Daa JUMP NC, aa      Conditional JUMP to address "aa" performed when CARRY
                      flag is reset
```

CALL

The CALL instruction is similar in operation to the JUMP instruction. It modifies the normal program execution sequence by specifying a new address. The CALL instruction is conditional. In addition to supplying a new address, the CALL instruction also causes the current PC value to be pushed onto the program counter stack. The CALL instruction has no effect on the status of the flags (Figure 10).]

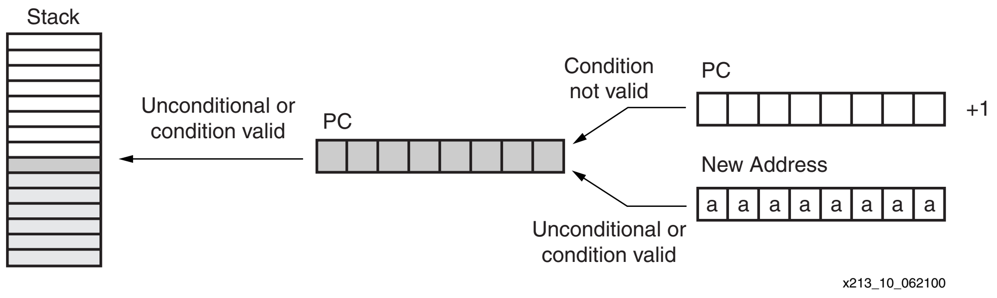


Figure 10: **CALL Instruction**

The program counter stack supports a depth of 15 address values, enabling a nested CALL sequence to the depth of 15 levels to be performed. Since the stack is also used during an interrupt operation, at least one of these levels should be reserved when interrupts are enabled.

The stack is implemented as a separate cyclic buffer. When the stack is full, it overwrites the oldest value. Each CALL instruction must specify the 8-bit address as a 2-digit hexadecimal value. To simplify this process, labels are supported in the assembler.

Hence, it is not necessary to reset the stack pointer when performing either a software or hardware reset. Therefore, there are no instructions to control the stack and no program memory is reserved for the stack.

```
83aa CALL    aa      Unconditional JUMP to address "aa"
93aa CALL Z,  aa     Conditional CALL to address "aa" performed when ZERO
                     flag is set
97aa CALL NZ, aa     Conditional CALL to address "aa" performed when ZERO
                     flag is reset
9Baa CALL C,  aa     Conditional CALL to address "aa" performed when CARRY
                     flag is set
9Faa CALL NC, aa     Conditional CALL to address "aa" performed when CARRY
                     flag is reset
```

RETURN

The RETURN instruction is the complement to the CALL instruction. The RETURN instruction is also conditional. In Figure 11, the new PC value is formed internally by incrementing the last value on the program address stack, ensuring the program executes the instruction following the CALL instruction which resulted in the subroutine. The RETURN instruction has no effect on the status of the flags.

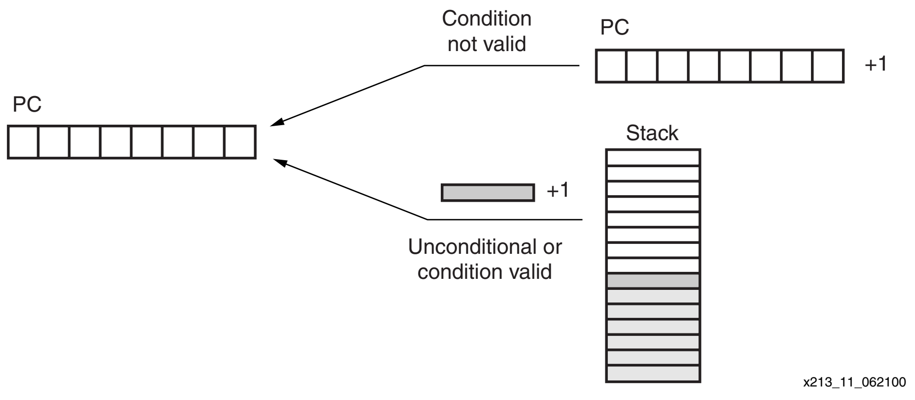


Figure 11: **RETURN Instruction**

The programmer must ensure that a RETURN is only performed in response to a previous CALL instruction, so that the program counter stack contains a valid address. The cyclic implementation of the stack continues to provide values for RETURN instructions that cannot be defined. Each RETURN only specifies the condition for flag tests.

```
8080 RETURN    Unconditional return to address following last CALL.
9080 RETURN Z  Return to address following last CALL provided that ZERO flag
               is set
9480 RETURN NZ Return to address following last CALL provided that ZERO flag
               is reset
9880 RETURN C  Return to address following last CALL provided that CARRY
               flag is set
9C80 RETURN NC Return to address following last CALL provided that CARRY
               flag is reset
```

Interrupt Group

RETURNI

The RETURNI instruction is a special variation of the RETURN instruction (Figure 12). It concludes an interrupt service routine. The RETURNI is unconditional and always loads the program counter (PC) with the last address on the program counter stack. The address does not increment in this case, because the instruction at the address stored needs to be executed. The RETURNI instruction restores the flags to the point of interrupt condition. It also determines the future ability of interrupts using ENABLE and DISABLE as an operand.

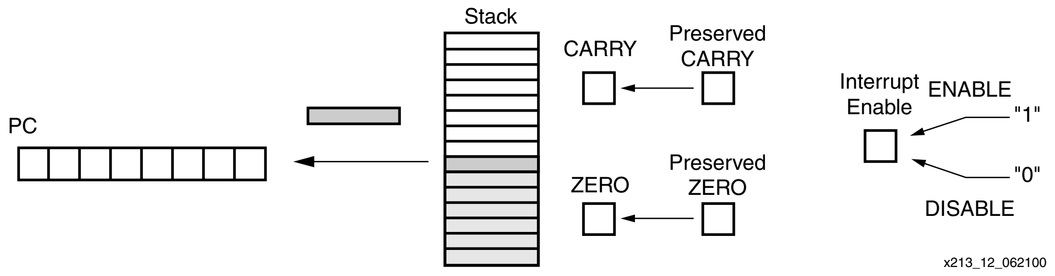


Figure 12: **RETURNI Instruction**

A RETURNI is only performed in response to an interrupt. Each RETURNI must specify if a further interrupt is enabled or disabled.

```
80F0 RETURNIENABLE  Return from interrupt routine and enable a future
                    interrupt
80D0 RETURNIDISABLE Return from interrupt routine and disable a future
                    interrupt
```

ENABLE INTERRUPT and DISABLE INTERRUPT

These instructions are used to set and reset the INTERRUPT ENABLE flag. Before using ENABLE INTERRUPT, a suitable interrupt routine must be associated with the interrupt address vector (FF). Never enable interrupts while performing an interrupt service (Figure 13).

```
8030 ENABLE  INTERRUPT      Enable future interrupt
8010 DISABLE INTERRUPT      Disable future interrupt
```

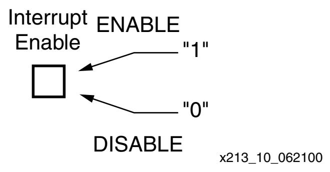


Figure 13: **ENABLE INTERRUPT Instruction**

Logical Group

LOAD

The LOAD instruction specifies the contents of any register. The new value is either a constant or the contents of any other register. The LOAD instruction has no effect on the status of the flags (Figure 14).

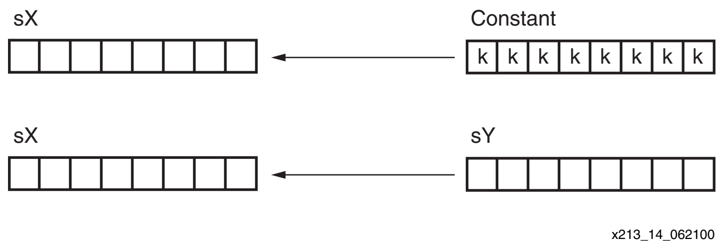


Figure 14: **LOAD Instruction**

Since the LOAD instruction does not affect the flags, it is used to reorder and assign register contents at any stage of the program execution. Because the load instruction is able to assign a constant with no impact to the program size or performance, the load instruction is the most obvious way to assign a value or clear a register.

Some implied "virtual" instructions are listed.

LOAD s0,s0	Loading any register with its own contents achieves nothing and hence is a NO OPERATION consuming two clock cycles. This is used to form a delay in the program.
LOAD sX,00	Loading zero is the equivalent of a CLEAR register command.

Each LOAD instruction specifies the destination register as "s" followed by a single hexadecimal digit (sX). It then specifies the source register value in a similar way (sY), or as an 8-bit constant using two hexadecimal digits (kk). The assembler supports labels to simplify the use of constants.

```
Cxy0 LOAD sX,sY  Load Register X with the contents of Register Y
0xkk LOAD sX,kk  Load Register X with a constant value
```

AND

The AND instruction performs a bit-wise logical AND operation between two operands. For example, 00001111 AND 00110011 produces the result 00000011. The first operand is any register, and it is the register assigned the result of the operation. A second operand is also any register, or an 8-bit constant value (Figure 15). Flags are affected by this operation.

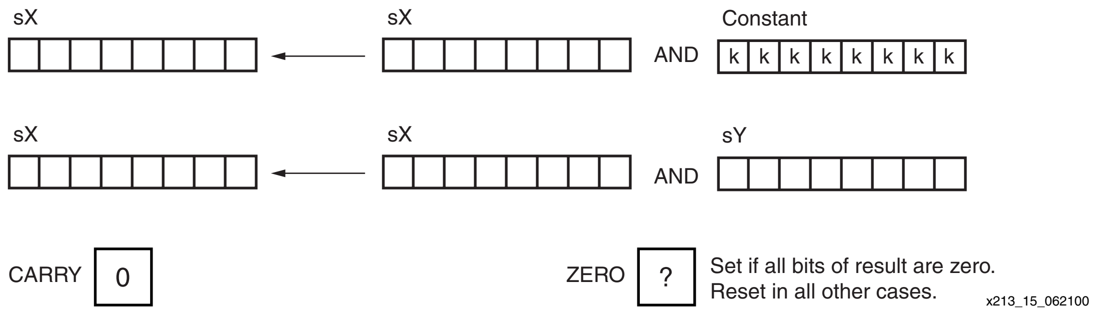


Figure 15: **AND Instruction**

The AND operation is useful in performing tests on the contents of a register. The status of the ZERO flag controls the flow of the program.

INPUT	s5, 56	This example reads an input port (address 56).
AND	s5, 04	It then tests bit 2 of the captured byte stored in register 5. The contents are overwritten.
CALL NZ,	input active	A non-zero result indicates that the tested bit was set and causes a CALL to a service routine.

Each AND instruction must specify the first operand register as "s" followed by a single hexadecimal digit (sX). This register also forms the destination for the result. The second operand specifies a second register value in a similar way (sY), or specifies an 8-bit constant using two hexadecimal digits (kk). The assembler supports labels to simplify the use of constants.

```
Cxy1 AND sX,sY   Bit-wise AND of Register X with the contents of Register Y
1xkk AND sX,kk   Bit-wise AND of Register X with a constant value
```

OR

The OR instruction performs a bit-wise logical OR operation between two operands. For example, 00001111 OR 00110011 produces the result 00111111. The first operand is any register. This register is assigned as the result of this operation. A second operand is also any register, or an 8-bit constant value (Figure 16). Flags are affected by the OR operation.

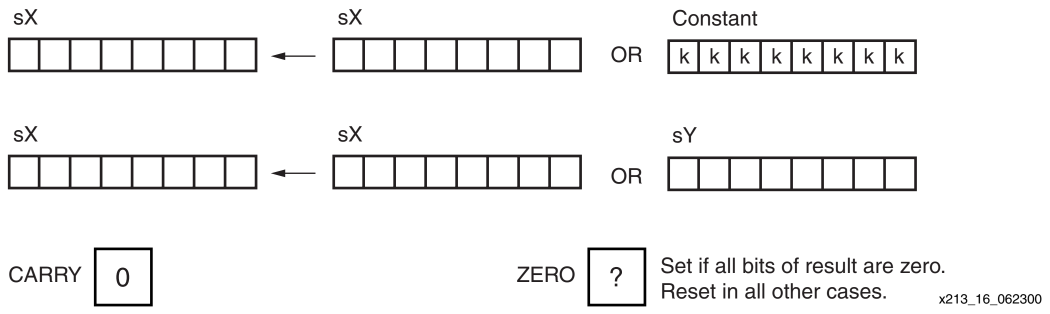


Figure 16: **OR Instruction**

Useful in forming control signals, the OR instruction provides a way to force setting any bit of the specified register. The use of OR sX,00 determines if the contents of a register are zero without changing the contents of the register.

A useful virtual instruction is

OR sX,00	Clear CARRY flag and test register for ZERO.

Each OR instruction must specify the first operand register as "s" followed by a single hexadecimal digit (sX). This register also forms the destination for the result. The second operand must then specify a second register value in a similar way (sY), or specify an 8-bit constant using two hexadecimal digits. The assembler supports labels to simplify the use of constants.

```
Cxy2 OR sX,sY  Bit-wise OR of Register X with the contents of Register Y
2xkk OR sX,kk  Bit-wise OR of Register X with a constant value
```

XOR

The XOR instruction performs a bit-wise logical XOR operation between two operands. For example, 00001111 XOR 00110011 produces the result 00111100. The first operand is any register, and this register is assigned the result of the operation. A second operand is also any register, or an 8-bit constant value. Flags are affected by this operation (Figure 17).

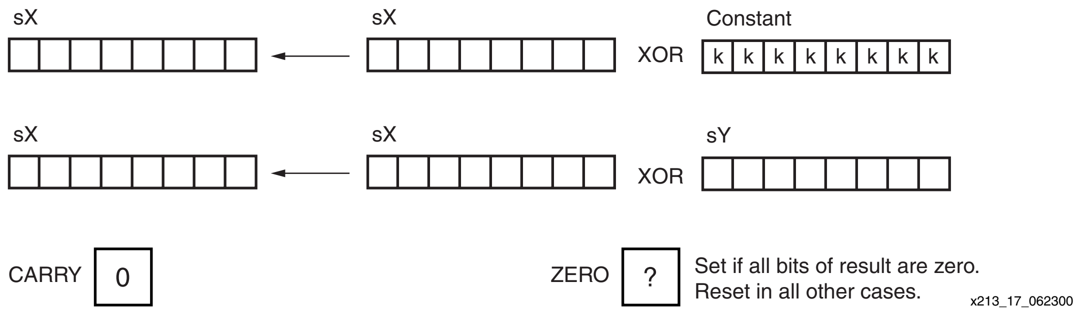


Figure 17: **XOR Instruction**

The XOR operation is useful for inverting bits contained in a register. This operation is useful in forming control signals.

```
       LOAD s5,01
loop:  XOR s5,03
       OUTPUT s5,56
       CALL delay
       JUMP loop
```

In this example, the XOR instruction keeps writing the contents of register "s5" to port 56 at intervals set by a delay subroutine. The XOR operation is used to modify the polarity of bit0 and bit1 during each cycle. The initial load value of register 5 has ensured that bit0 and bit1 are different.

Each XOR instruction must specify the first operand register as "s" followed by a single hexadecimal digit (sX). This register also forms the destination for the result. The second operand must then specify a second register value in a similar way (sY), or specify an 8-bit constant using two hexadecimal digits (kk). The assembler supports labels to simplify the use of constants.

```
Cxy3 XOR sX,sY Bit-wise XOR of Register X with the contents of Register Y
3xkk XOR sX,kk Bit-wise XOR of Register X with a constant value
```

Arithmetic Group

ADD

The ADD instruction performs an 8-bit unsigned addition of two operands. The first operand is any register, and it is this register that is assigned the result of the operation. A second operand is also any register, or an 8-bit constant value (Figure 18). Flags are affected by this operation.

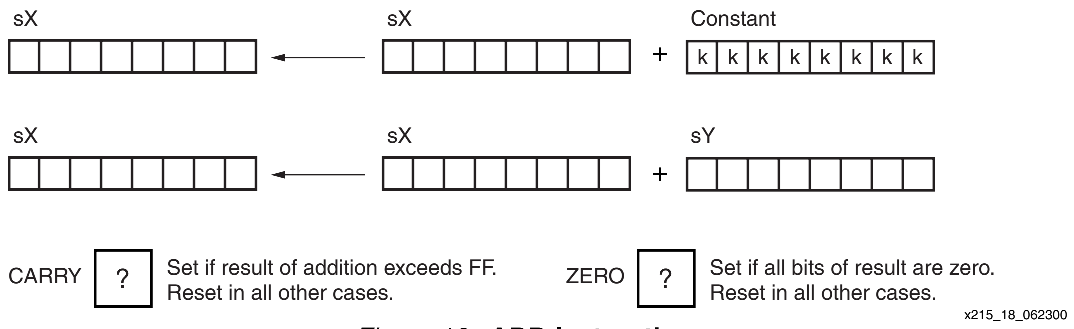

sX
sX
Constant
+
k k k k k k k k
sX
sX
sY
+
CARRY
?
Set if result of addition exceeds FF.
Reset in all other cases.
ZERO
?
Set if all bits of result are zero.
Reset in all other cases.
x215_18_062300

Figure 18: **ADD Instruction**

The ADD operation has many applications of which real addition is just one. Note that this instruction does not use the CARRY as an input, and hence, there is no need to condition the flags before use.

The ability to specify any constant is useful in forming control sequences or counters. An obvious virtual instruction is

ADD sX,01 Equivalent to INCREMENT the register value

Each ADD instruction must specify the first operand register as "s" followed by a single hexadecimal digit (sX). This register forms the destination for the result. The second operand must then specify a second register value in a similar way (sY), or specify an 8-bit constant using two hexadecimal digits (kk). The assembler supports labels to simplify the use of constants.

```
Cxy4 ADD sX,sY Addition of Register X with the contents of Register Y
4xkk ADD sX,kk Addition of Register X with a constant value
```

ADDCY

The ADDCY instruction performs an unsigned addition of two 8-bit operands together with the contents of the CARRY flag. The first operand is any register, and this register is assigned the result of the operation. A second operand is also any register, or an 8-bit constant value (Figure 19). Flags are affected by this operation.

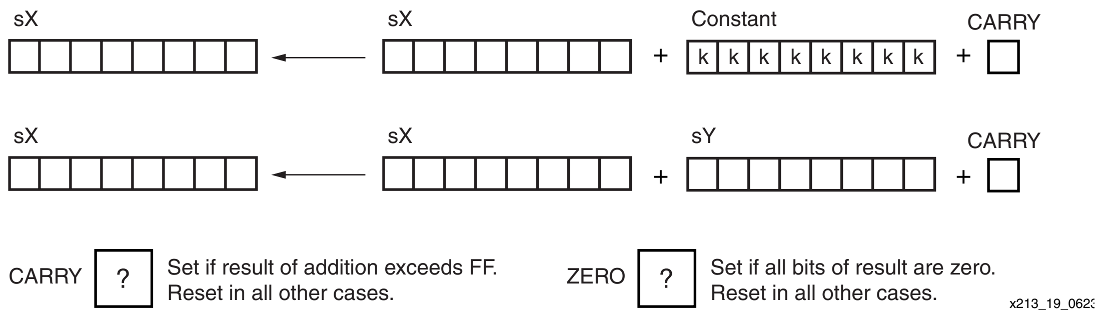

sX
sX
Constant
CARRY
k k k k k k k k
+
+
sX
sX
sY
CARRY
+
+
CARRY
?
Set if result of addition exceeds FF.
Reset in all other cases.
ZERO
?
Set if all bits of result are zero.
Reset in all other cases.
x213_19_0623

Figure 19: **ADDCY Instruction**

The ADDCY operation is used in the formation of adder and counter processes exceeding eight bits.

ADD s7,01	Here, a 16-bit counter is formed by a pair of registers. First, the lower byte (s7) increments.
ADDCY s8,00	Next, the effect of a carry from the lower byte is used to increment the upper byte (s8).

Each ADDCY instruction must specify the first operand register as "s" followed by a single hexadecimal digit (sX). This register also forms the destination for the result. The second operand must then specify a second register value in a similar way (sY), or specify an 8-bit constant using two hexadecimal digits (kk). The assembler supports labels to simplify the use of constants.

```
Cxy5 ADDCY sX,sY Addition of Register X with the contents of Register Y
                 and CARRY
5xkk ADDCY sX,kk Addition of Register X with a constant value
                 and CARRY
```

SUB

The SUB instruction performs an 8-bit unsigned subtraction of two operands. The first operand is any register, and this register is assigned the result of the operation. The second operand is also any register, or an 8-bit constant value (Figure 20). Flags are affected by this operation.

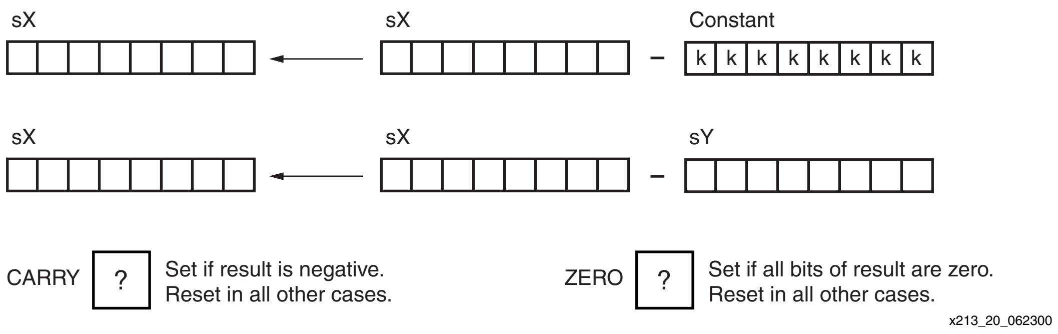

sX
sX
Constant
k k k k k k k k
sX
sX
sY
CARRY ?
Set if result is negative.
Reset in all other cases.
ZERO ?
Set if all bits of result are zero.
Reset in all other cases.
x213_20_062300

Figure 20: **SUB Instruction**

The SUB operation has many applications, of which real subtraction is just one. Note that this instruction does not use the CARRY as an input, and hence there is no need to condition the flags before use. The CARRY flag now indicates when an underflow has occurred and that the result is actually negative. For example, if "s5" contains 27 hex and the instruction SUB s5,35 is performed, then the stored result is F2 hex and the CARRY flag is set.

The ability to specify any constant is useful in forming control sequences or down counters. An obvious virtual instruction is, therefore,

SUB sX,01	Equivalent to DECREMENT the register value

Each SUB instruction must specify the first operand register as "s" followed by a single hexadecimal digit (sX). This register also forms the destination for the result. The second operand must then specify a second register value in a similar way (sY), or specify an 8-bit constant using two hexadecimal digits (kk). The assembler supports labels to simplify the use of constants.

```
Cxy6 SUB sX,sY Subtract contents of Register Y from contents of Register X
6xkk SUB sX,kk Subtract a constant value from contents of Register X
```

SUBCY

The SUBCY instruction performs an 8-bit unsigned subtraction of two operands together with the contents of the CARRY flag. The first operand is any register, and this register is assigned the result of the operation. The second operand is also any register, or an 8-bit constant value (Figure 21). Flags are affected by this operation.

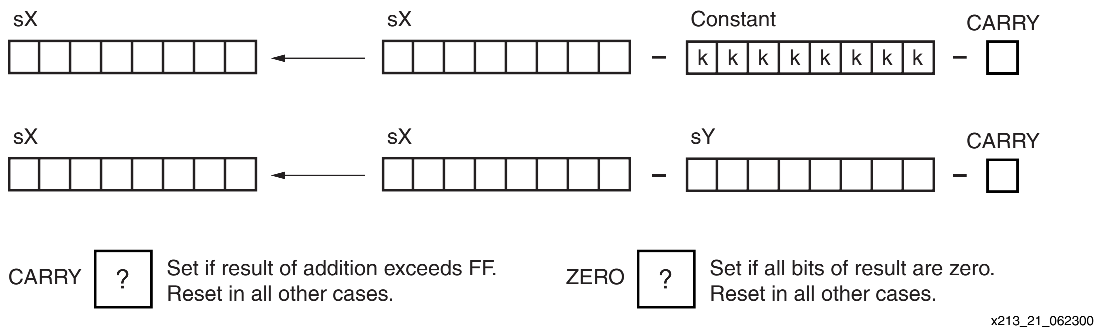

sX
sX
Constant
k k k k k k k k
CARRY
sX
sX
sY
CARRY
CARRY
?
Set if result of addition exceeds FF.
Reset in all other cases.
ZERO
?
Set if all bits of result are zero.
Reset in all other cases.
x213_21_062300

Figure 21: **SUBCY Instruction**

The SUBCY operation is used in the formation of subtract and down-counter processes exceeding 8 bits.

SUB s7,01	Here, a 16 bit down counter is formed by a pair of registers. First the lower byte (s7) is decremented.
SUBCY s8,00	Next, the effect of a "borrow" from the lower byte is used to decrement the upper byte (s8).

Each SUBCY instruction must specify the first operand register as "s" followed by a single hexadecimal digit (sX). This register also forms the destination for the result. The second operand must then specify a second register value in a similar way (sY), or specify an 8-bit constant using two hexadecimal digits (kk). The assembler supports labels to simplify the use of constants.

```
Cxy7 SUBCY sX,sY Subtract contents of Register and CARRY from contents of
                 Register X
7xkk SUBCY sX,kk Subtract a constant value and CARRY from contents of
                 Register X
```

Shift and Rotate Group

SR0, SR1, SRX, SRA, RR

The shift and rotate right group all modify the contents of a single register to the right (Figure 22). All instructions in the group have an effect on the flags.

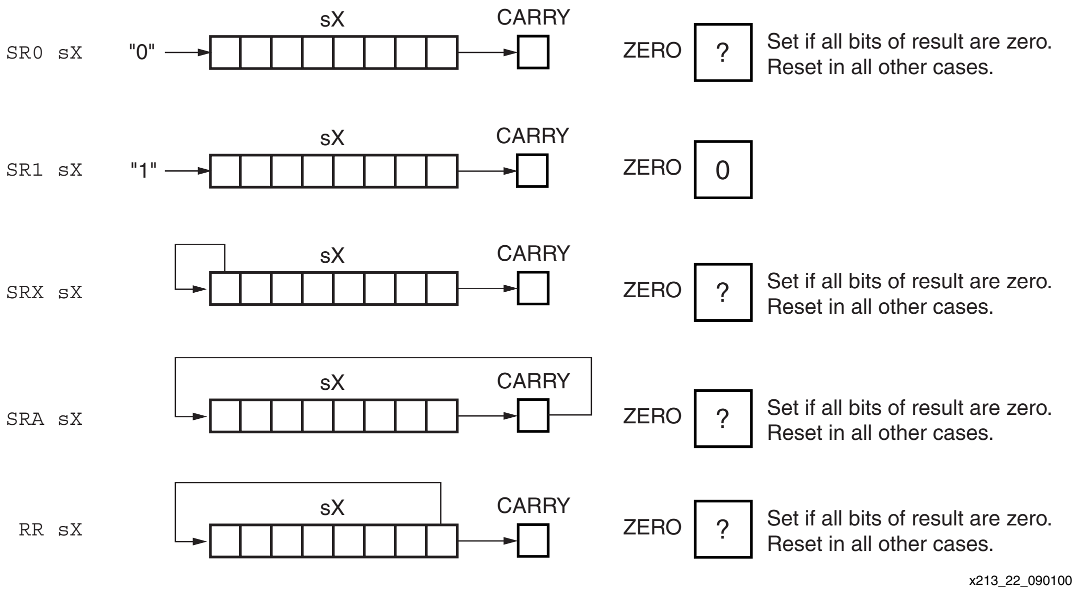

SR0 sX
"0"
sX
CARRY
ZERO
?
Set if all bits of result are zero.
Reset in all other cases.
SR1 sX
"1"
sX
CARRY
ZERO
0
SRX sX
sX
CARRY
ZERO
?
Set if all bits of result are zero.
Reset in all other cases.
SRA sX
sX
CARRY
ZERO
?
Set if all bits of result are zero.
Reset in all other cases.
RR sX
sX
CARRY
ZERO
?
Set if all bits of result are zero.
Reset in all other cases.
x213_22_090100

Figure 22: **Right Shift Register Instructions**

Each instruction must specify the register as "s" followed by a single hexadecimal digit.

```
Dx0E  SR0 sX    Shift register X right by one place injecting "0"
Dx0F  SR1 sX    Shift register X right by one place injecting "1"
Dx0A  SRX sX    Shift register X right by one place performing sign extension
Dx08  SRA sX    Shift register X right by one place injecting CARRY flag
Dx0C  RR  sX    Rotate register X right by one place replacing MSB with LSB
```

SL0, SL1, SLX, SLA, RL

The shift and rotate left group all modify the contents of a single register to the left (Figure 23). All instructions in the group have an effect on the flags.

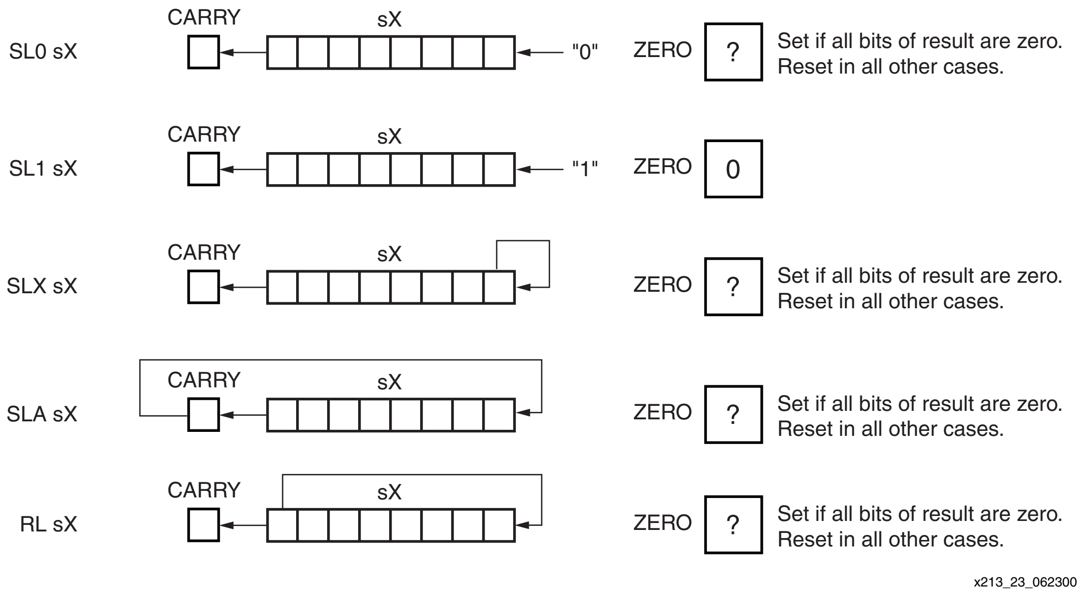

SL0 sX
CARRY
sX
"0"
ZERO
?
Set if all bits of result are zero.
Reset in all other cases.
SL1 sX
CARRY
sX
"1"
ZERO
0
SLX sX
CARRY
sX
ZERO
?
Set if all bits of result are zero.
Reset in all other cases.
SLA sX
CARRY
sX
ZERO
?
Set if all bits of result are zero.
Reset in all other cases.
RL sX
CARRY
sX
ZERO
?
Set if all bits of result are zero.
Reset in all other cases.
x213_23_062300

Figure 23: **Left SHIFT Register Instructions**

Each instruction must specify the register as "s" followed by a single hexadecimal digit.

```
Dx06 SL0 sX      Shift register X left by one place injecting "0"
Dx07 SL1 sX      Shift register X left by one place injecting "1"
Dx04 SLX sX      Shift register X left by one place performing sign extension
Dx00 SLA sX      Shift register X left by one place injecting CARRY flag
Dx02 RL  sX      Rotate register X left by one place replacing LSB with MSB
```

Input and Output Group

INPUT

The INPUT instruction enables data values external to the KCPSM to be transferred into any one of the internal registers (Figure 24). The port address (in the range 00 to FF) is defined by a constant value, or indirectly as the contents of the any other register. The flags are not affected by this operation.

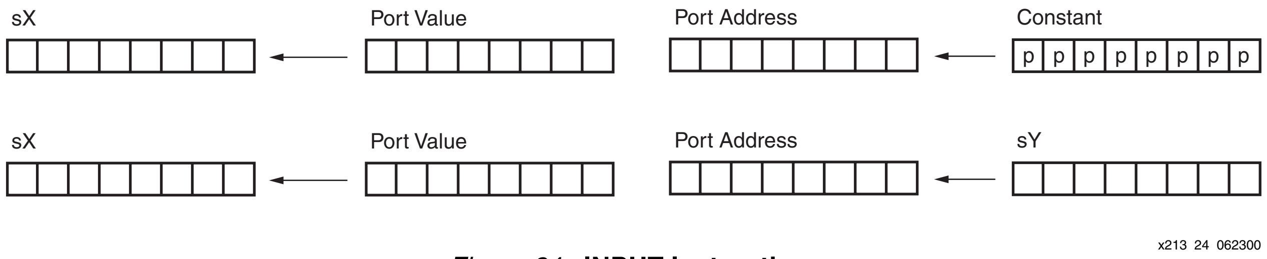


Figure 24: **INPUT Instruction**

The user interface logic is required to decode the port address value and supply the correct data. The signal waveforms are shown in Figure 25. Note that the READ_STROBE provides an indicator that a port has been read, but it is not vital to qualify a valid address.

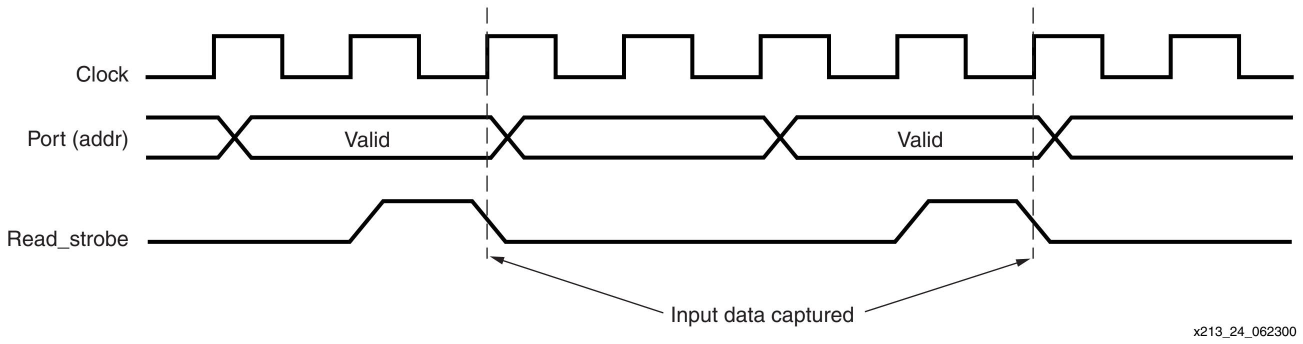


Figure 25: **INPUT Signal Waveform**

Each INPUT instruction must specify the destination register as "s" followed by a single hexadecimal digit (sX). It must then specify the input port address using a register value in a similar way (sY), or specify an 8-bit constant using two hexadecimal digits (pp). The assembler supports labels to simplify the use of port address constants.

```
Bxy0 INPUT sX,sY    Transfer value from port specified by contents of
                    register Y to register X
Axpp INPUT sX,pp    Transfer value from port specified by constant to
                    register X
```

OUTPUT

The OUTPUT instruction enables the contents of any register to transfer to logic external to the KCPSM. The port address (in the range 00 to FF) is defined by a constant value, or indirectly as the contents of the any other register (Figure 26). The flags are not affected by this operation.

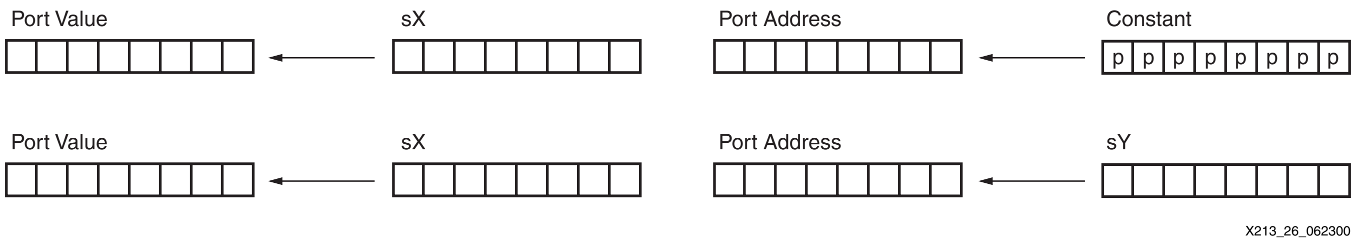


Figure 26: **OUTPUT Instruction**

The user interface logic is required to decode the port address value and enable the correct logic to capture the data value. The WRITE_STROBE is used in this case to ensure the transfer of valid data only. The signal waveforms are shown in Figure 27.

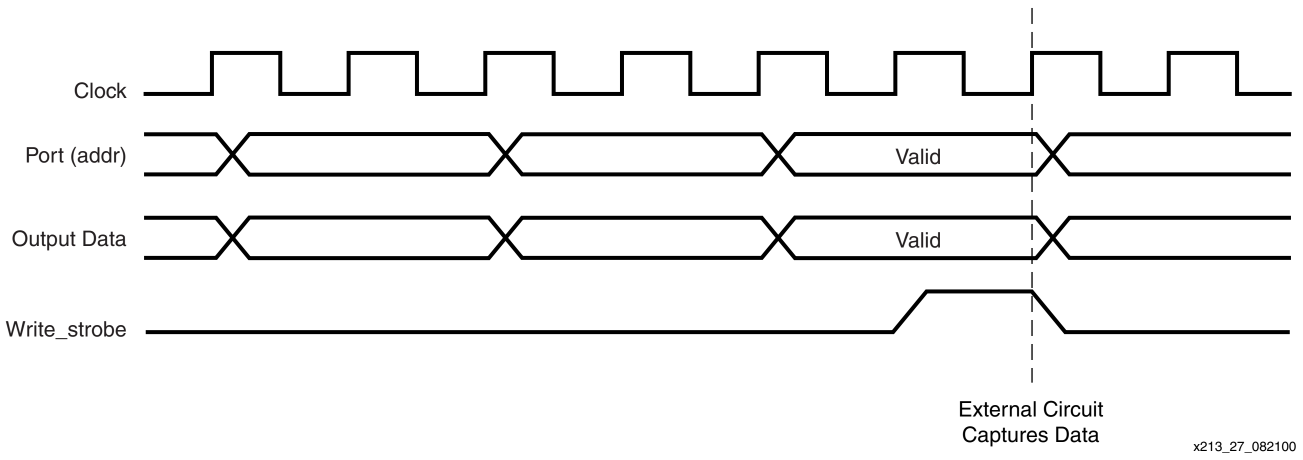


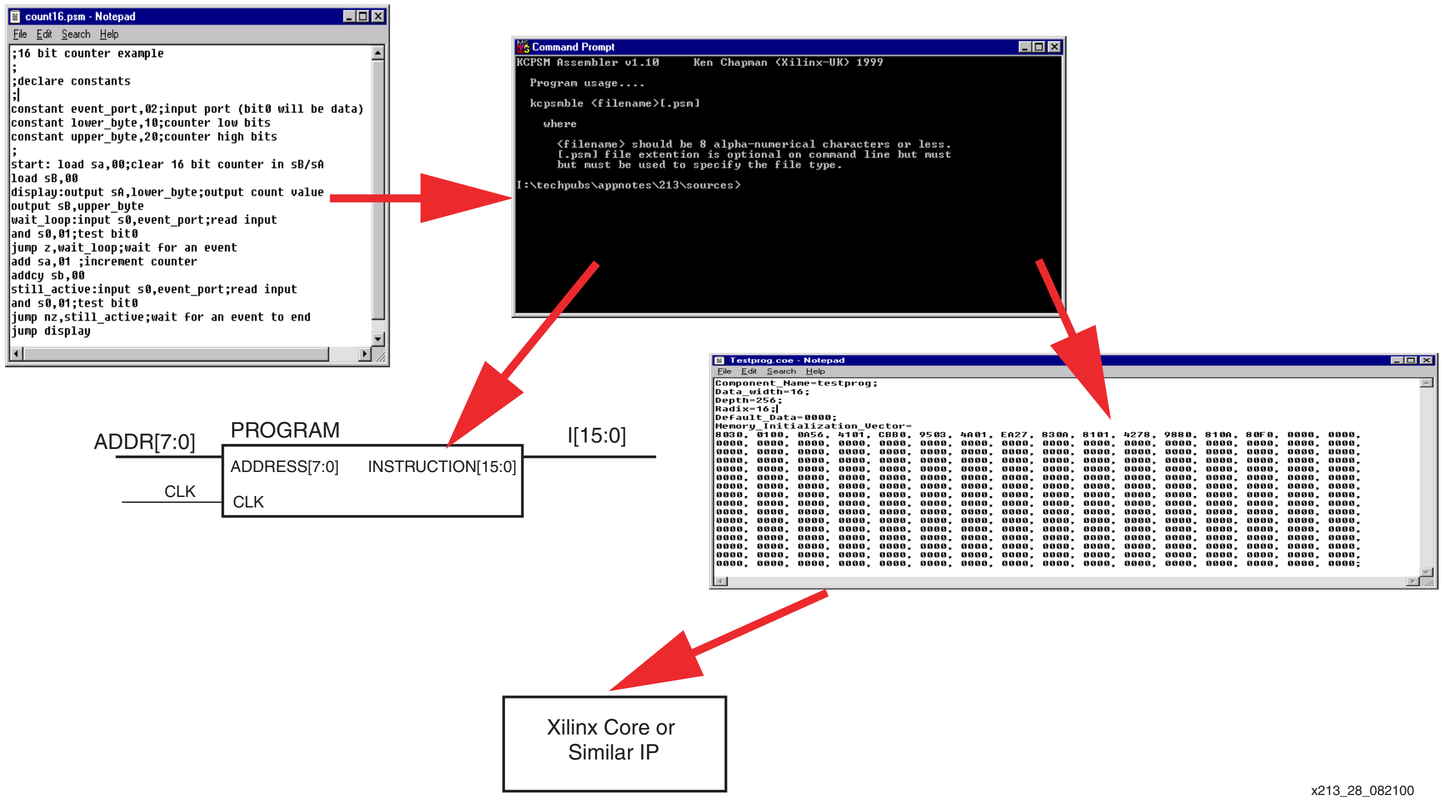
Figure 27: **OUTPUT Signal Waveform**

Each OUTPUT instruction must specify the source register as "s" followed by a single hexadecimal digit (sX). It must then specify the output port address using a register value in a similar way (sY), or specify an 8-bit constant using two hexadecimal digits (pp). The assembler supports labels to simplify the use of port address constants.

```
Fxy0 OUTPUT sX,sY   Transfer value from register X to port specified by
                    contents of register Y
Expp OUTPUT sX,pp   Transfer value from register X to port specified by
                    constant.
```

KCPSMBLE Assembler

To simplify the generation of programs, an assembler called KCPSMBLE is provided. Programs are best written with either the standard Notepad or Word tools. The file is saved with a **`.psm`** file extension (eight character name limit). Figure 28 illustrates the process for using the KCPSMBLE files.



```
;16 bit counter example
;
;declare constants
;
constant event_port,02;input port (bit0 will be data)
constant lower_byte,10;counter low bits
constant upper_byte,20;counter high bits
;
start: load sa,00;clear 16 bit counter in sB/sA
load sB,00
display:output sA,lower_byte;output count value
output sB,upper_byte
wait_loop:input s0,event_port;read input
and s0,01;test bit0
jump z,wait_loop;wait for an event
add sa,01 ;increment counter
addcy sb,00
still_active:input s0,event_port;read input
and s0,01;test bit0
jump nz,still_active;wait for an event to end
jump display
```

```
KCPSM Assembler v1.10     Ken Chapman (Xilinx-UK) 1999

  Program usage....

  kcpsmble <filename>[.psm]

    where

      <filename> should be 8 alpha-numerical characters or less.
      [.psm] file extention is optional on command line but must
      but must be used to specify the file type.

I:\techpubs\appnotes\213\sources>
```

```
Component_Name=testprog;
Data_width=16;
Depth=256;
Radix=16;
Default_Data=0000;
Memory_Initialization_Vector=
8030, 0100, 0A56, 4101, CBB0, 9503, 4A01, EA27, 830A, 8101, 4278, 9880, 810A, 80F0, 0000, 0000,
0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000,
0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000,
0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000,
0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000,
0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000,
0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000,
0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000,
0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000,
0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000,
0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000,
0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000,
0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000,
0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000,
0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000,
0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000;
```

Figure 28: **KCPSMBLE Assembler**

kcpsmble <*prog_name*>[.psm]

Place the KCPSMBLE.EXE file in the same directory as the program file. Open a DOS box and navigate to the directory. Then run the assembler **`kcpsmble <`*`filename`*`>[.psm]`**. The assembler executes very quickly and the display often appears immediately.

Direct EDIF netlist (<*prog_name*>.edn).

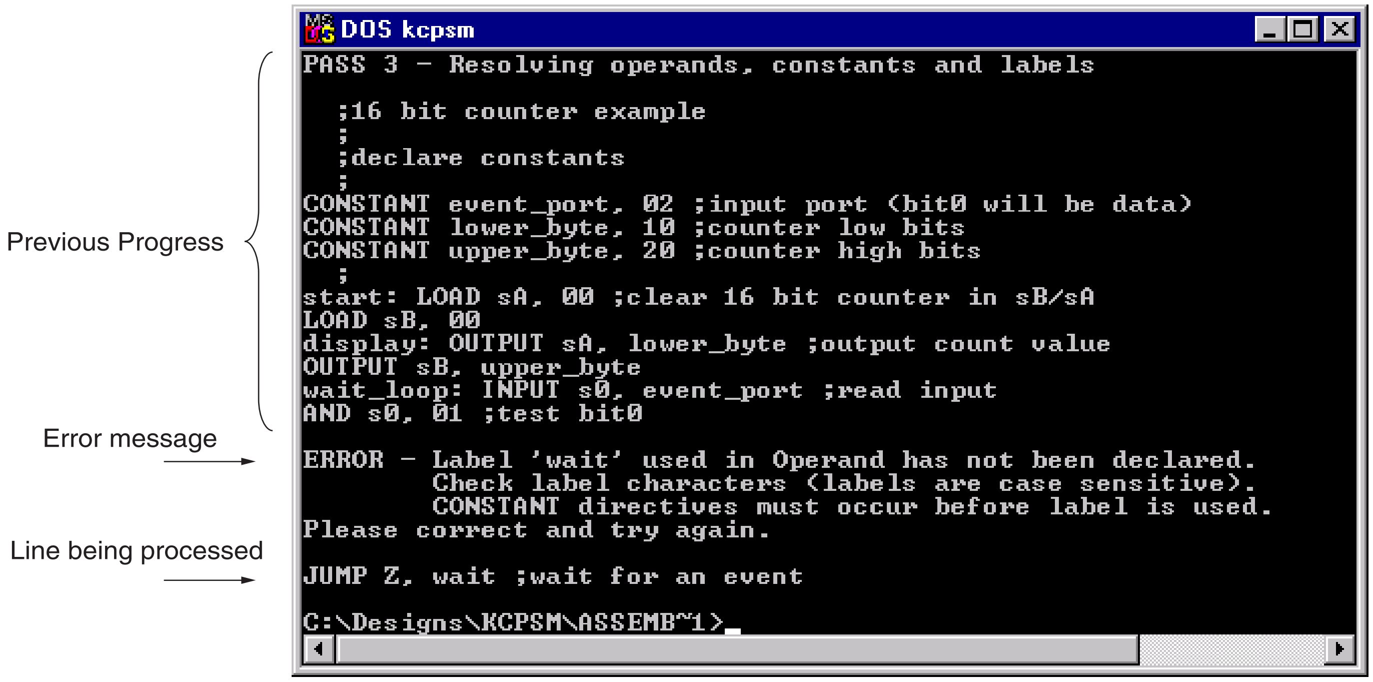
This immediately provides a "black box" in which the block RAM which has been initialized and connected in the ROM configuration (Figure 6).

Coefficient File (<*prog_name*>.coe).

This file is easily read and modified and used in conjunction with the Core Generator to include the program in any block RAM configuration. This is useful for accessing dual port block RAM configurations.

Assembler Errors

The assembler stops as soon as an error is detected. A short message is displayed to help determine the reason for the error. The assembler also displays the line it was analyzing when the problem was detected. Since the execution of the assembler is very fast, the display often appears to be immediate. Fix each reported problem in turn and re-execute the assembler.



```
PASS 3 - Resolving operands, constants and labels

   ;16 bit counter example
   ;
   ;declare constants
   ;
CONSTANT event_port, 02 ;input port (bit0 will be data)
CONSTANT lower_byte, 10 ;counter low bits
CONSTANT upper_byte, 20 ;counter high bits
   ;
start: LOAD sA, 00 ;clear 16 bit counter in sB/sA
LOAD sB, 00
display: OUTPUT sA, lower_byte ;output count value
OUTPUT sB, upper_byte
wait_loop: INPUT s0, event_port ;read input
AND s0, 01 ;test bit0

ERROR - Label 'wait' used in Operand has not been declared.
        Check label characters (labels are case sensitive).
        CONSTANT directives must occur before label is used.
Please correct and try again.

JUMP Z, wait ;wait for an event

C:\Designs\KCPSM\ASSEMB~1>_
```

x213_29_062600

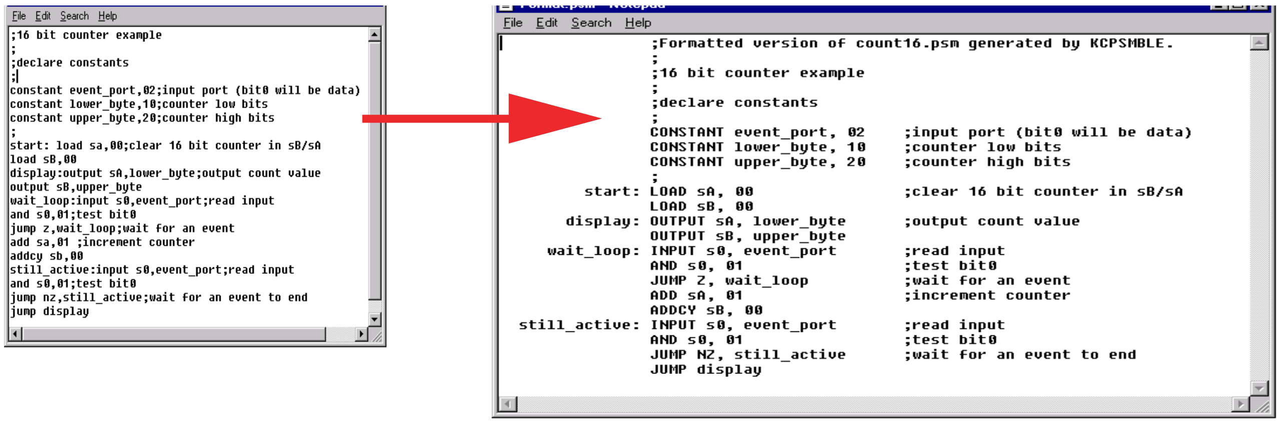
Figure 29: **Assembler Error Display**

FORMAT.PSM File

When a program passes through the assembler, additional files to the `.coe` and `.edn` files are produced to assist the programmer. One of these files is called `FORMAT.PSM`, which is the original program reformatted. The `FORMAT.PSM` file provides:

- Separate labels and comments
- Puts all commands in upper case and correctly spaces operands
- Gives registers an "sX" format
- Converts constants to upper case

Looking at this file shows if the expected interpretation has occurred. This utility is an easy way to write programs, while KCPSMBLE formats the work.



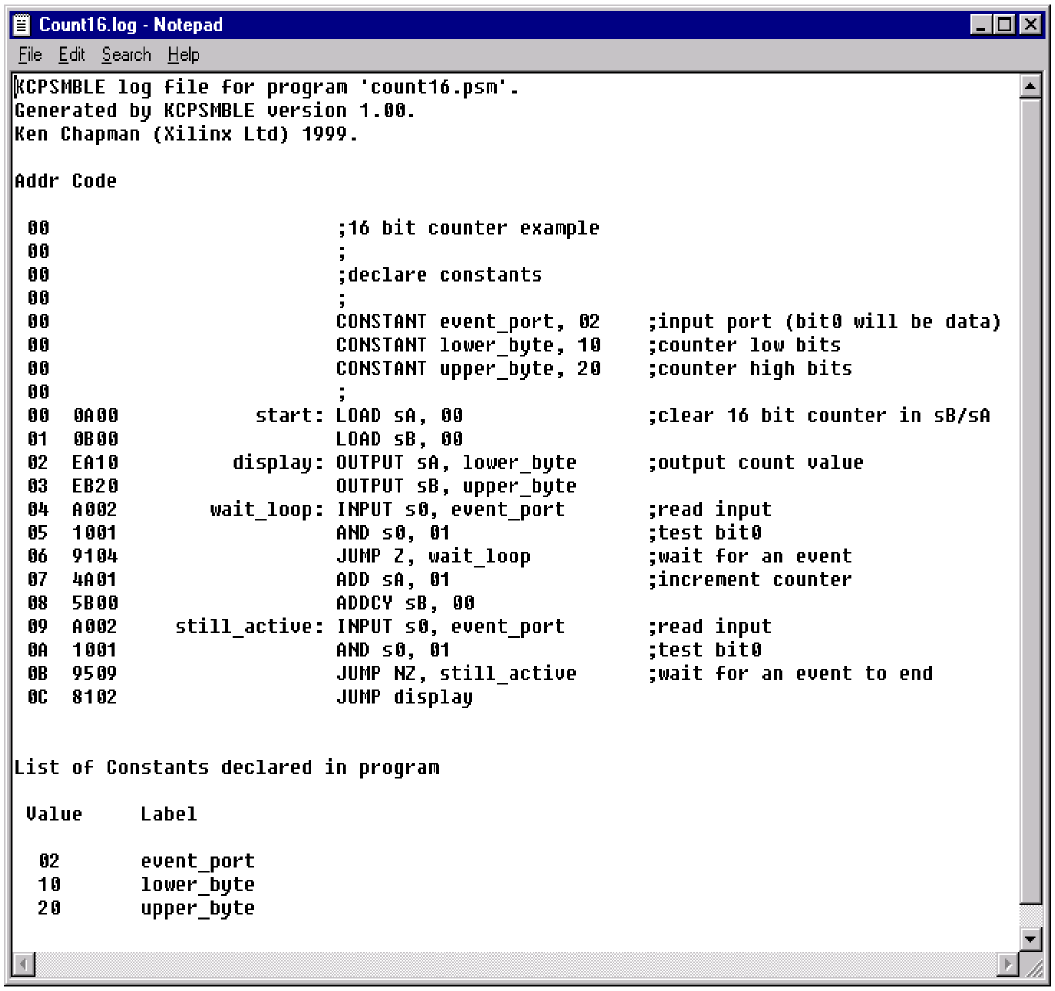
```
;16 bit counter example
;
;declare constants
;
constant event_port,02;input port (bit0 will be data)
constant lower_byte,10;counter low bits
constant upper_byte,20;counter high bits
;
start: load sa,00;clear 16 bit counter in sB/sA
load sB,00
display:output sA,lower_byte;output count value
output sB,upper_byte
wait_loop:input s0,event_port;read input
and s0,01;test bit0
jump z,wait_loop;wait for an event
add sa,01 ;increment counter
addcy sb,00
still_active:input s0,event_port;read input
and s0,01;test bit0
jump nz,still_active;wait for an event to end
jump display
```

```
              ;Formatted version of count16.psm generated by KCPSMBLE.
              ;
              ;16 bit counter example
              ;
              ;declare constants
              ;
              CONSTANT event_port, 02    ;input port (bit0 will be data)
              CONSTANT lower_byte, 10    ;counter low bits
              CONSTANT upper_byte, 20    ;counter high bits
              ;
       start: LOAD sA, 00                ;clear 16 bit counter in sB/sA
              LOAD sB, 00
     display: OUTPUT sA, lower_byte      ;output count value
              OUTPUT sB, upper_byte
   wait_loop: INPUT s0, event_port       ;read input
              AND s0, 01                 ;test bit0
              JUMP Z, wait_loop          ;wait for an event
              ADD sA, 01                 ;increment counter
              ADDCY sB, 00
still_active: INPUT s0, event_port       ;read input
              AND s0, 01                 ;test bit0
              JUMP NZ, still_active      ;wait for an event to end
              JUMP display
```

Figure 30: **FORMAT.PSM File**

<*prog_name*>.log File

The `.log` file provides the assembly process detail. It is possible to observe how each instruction and directive is used. Address and op-code values are associated with each line of the program and displays a list of the constants declared (Figure 31).



```
KCPSMBLE log file for program 'count16.psm'.
Generated by KCPSMBLE version 1.00.
Ken Chapman (Xilinx Ltd) 1999.

Addr Code

 00                         ;16 bit counter example
 00                         ;
 00                         ;declare constants
 00                         ;
 00                         CONSTANT event_port, 02    ;input port (bit0 will be data)
 00                         CONSTANT lower_byte, 10    ;counter low bits
 00                         CONSTANT upper_byte, 20    ;counter high bits
 00                         ;
 00  0A00            start: LOAD sA, 00                ;clear 16 bit counter in sB/sA
 01  0B00                   LOAD sB, 00
 02  EA10          display: OUTPUT sA, lower_byte      ;output count value
 03  EB20                   OUTPUT sB, upper_byte
 04  A002        wait_loop: INPUT s0, event_port       ;read input
 05  1001                   AND s0, 01                 ;test bit0
 06  9104                   JUMP Z, wait_loop          ;wait for an event
 07  4A01                   ADD sA, 01                 ;increment counter
 08  5B00                   ADDCY sB, 00
 09  A002     still_active: INPUT s0, event_port       ;read input
 0A  1001                   AND s0, 01                 ;test bit0
 0B  9509                   JUMP NZ, still_active      ;wait for an event to end
 0C  8102                   JUMP display


List of Constants declared in program

 Value     Label

  02       event_port
  10       lower_byte
  20       upper_byte
```

x213_31_062600

Figure 31: ***<prog_name>*.log File**

The log file helps confirm that all the labels have been correctly applied. In Figure 31, the label "display" is associated with address 02, and hence, the op-code for the JUMP is assigned the operand value of 02.

Foundation Simulation (.hex File)

When using the Foundation simulator, the contents of the block RAM used to hold the program are not automatically initialized.

The KCPSMBLE assembler provides a file called ***<prog_name>*.hex** to initialize the block RAM. The loading sequence is shown in Figure 32. Each time the simulation is reset to time zero, the block RAM is loaded again. The **Edit** option is used to verify that the memory content is loaded. Then the block RAM component in the design hierarchy is highlighted, **Select** is pressed and the hex file is located.

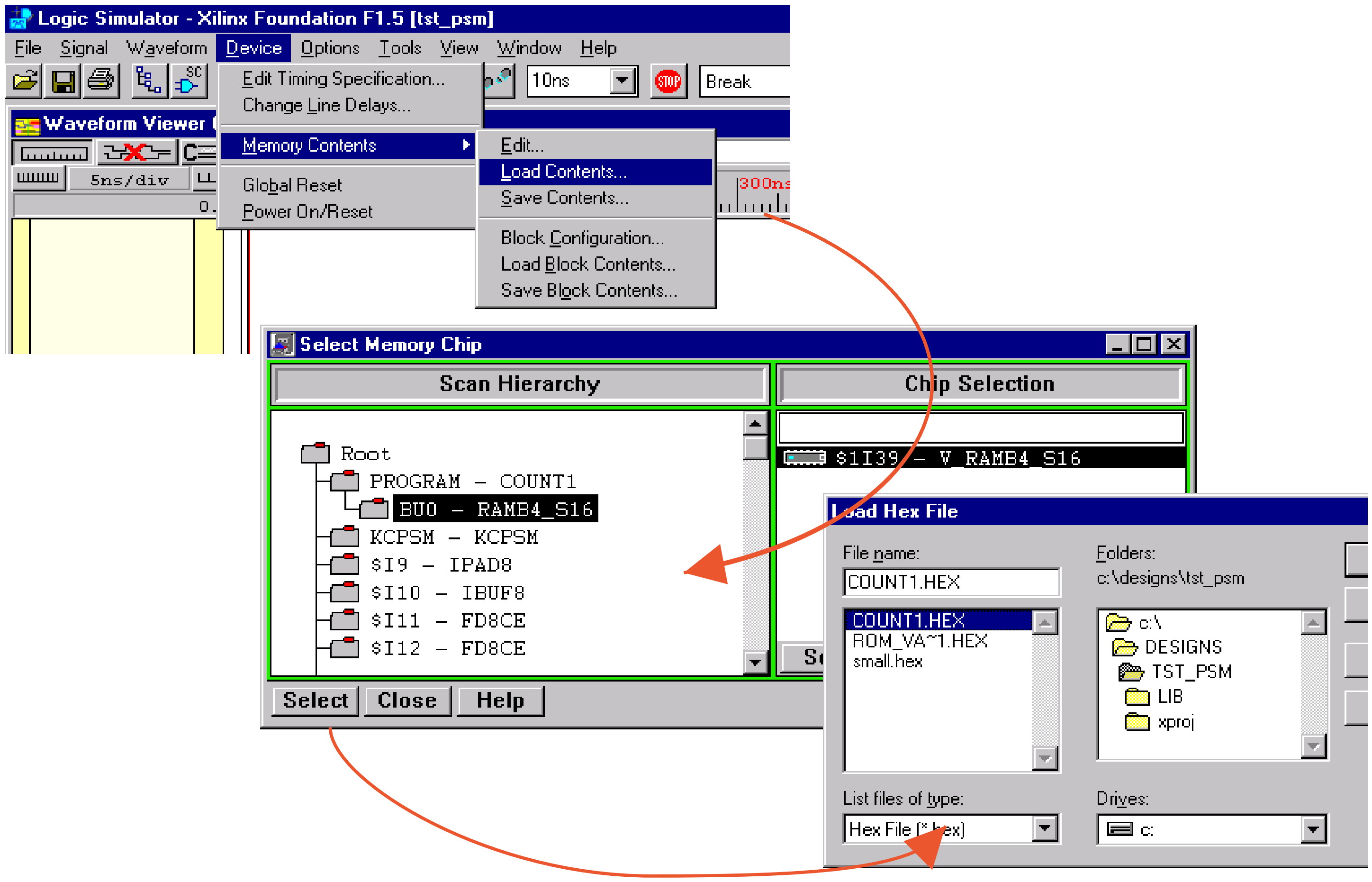


Figure 32: **Foundation Simulation**

Program Syntax

Probably the best way to understand what is and what is not valid syntax is to look at the examples and try the assembler. However, some simple rules are of assistance from the beginning. To assure that the correct program syntax is used, the following suggestions are recommended:

No blank lines. A blank line is ignored by the assembler and removed from any formatted files. A blank comment is the appropriate way to document a blank line.

Comments. Any item on a line following a semi-colon (;) is ignored by the assembler. Concise comments should be used to keep the program manageable for the relatively simple assembler.

Registers. All registers are defined as the letter "s" immediately followed by a single hexadecimal character in the range of 0 to F. The assembler accepts any mixture of upper and lower case characters and automatically converts them to the "sX" format, where "X" is one of the following: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

Constants. A constant is specified in the form of a two-digit hexadecimal value (range 00 to FF). The assembler accepts any mixture of upper and lower case characters and automatically converts them to upper case.

Labels. Labels are comprised of any user-defined text string, and are *case sensitive* for additional flexibility. No spaces are allowed, but the underscore character is supported. Valid characters are 0 to 9, a to z, and A to Z. Labels are used to identify a program line for reference in a JUMP or CALL instruction ended by a colon, as in the following example:

```
wait_loop1: INPUT s0, event_port       ;read input
            AND s0, 01                 ;test bit0
            JUMP Z, wait_loop1         ;wait for an event
```

Program Syntax Instructions

The instruction format is described under **Complete KCPSM Instruction Set**, page 6. The assembler is very forgiving over the use of spaces and <TAB> characters, but instructions and the first operand must be separated by at least one space. Instructions with two operands must ensure that a comma (,) separator is used.

The assembler accepts any mixture of upper and lower case characters for the instruction and automatically converts them to upper case. The following examples all show acceptable instruction specifications, but the formatted output shows how it was expected.

load s5,7E	KCPSMBLE	LOAD s5, 7E
AddCY s8,SE		ADDCY s8, sE
ENABLE interrupt		ENABLE INTERRUPT
Output S2, (S8)		OUTPUT s2, (s8)
jump Nz, 67		JUMP NZ, 67
ADD sF, step_value		ADD sF, step_value
INPUT S9,28		INPUT s9, 28
sl1 se		SL1 sE
RR S8		RR s8

Most other syntax problems are solved by reading the error messages provided by the assembler.

Assembler Directives

The KCSPMBLE assembler currently supports two directives. These commands are used purely by the assembly process and do not correspond to any instructions executed by KCPSM macro.

CONSTANT (k) Directive

This directive provides a way to assign an 8-bit constant value to a label. In this way, the program can declare constants such as port addresses and particular values before they are needed in the program. By predefining constant values (rather than entering them as actual values in the program commands), it is both easier to work with them and easier to understand the meaning of the program. The following example illustrates the constant directive syntax and its uses.

16-bit Counter Example

```
              CONSTANT  event_port, 02      ;input port (bit0 will be data)
              CONSTANT  lower_byte, 10      ;counter low bits
              CONSTANT  upper_byte, 20      ;counter high bits
              CONSTANT  zero, 00            ;counter high bits
                                            ;
       start: LOAD sA, zero                 ;clear 16 bit counter in sB/sA
              LOAD sB, zero
     display: OUTPUT sA, lower_byte         ;output count value
              OUTPUT sB, upper_byte
  wait_loop1: INPUT s0, event_port          ;read input
              AND s0, 01                    ;test bit0
              JUMP Z, wait_loop1            ;wait for an event
              ADD sA, 01                    ;increment counter
              ADDCY sB, 00
still_active: INPUT s0, event_port          ;read input
              AND s0, 01                    ;test bit0
              JUMP NZ, still_active         ;wait for an event to end
              JUMP display
```

In this example, **zero** is used to specify a data constant for clearing some registers. The other constants are used to define port addresses, and subsequent use in I/O operations makes it easy to understand which ports are being accessed. Notice that the **event_port** is used multiple times, and hence, the **CONSTANT** directive enables a single point of change.

ADDRESS Directive

The **ADDRESS** directive provides a way to force the assembly of the following instructions commencing at a new address value. This is useful for separating subroutines into specific locations and is vital for handling interrupts. The following log file shows the same 16-bit counter example modified to use subroutines to clear and increment the registers that are used as a counter. Although not absolutely necessary, the subroutines have been forced to specific memory locations. The simple syntax of the **ADDRESS** directive can also be observed in this example.

```
00  83B0            start: CALL clear_count16
01  EA10          display: OUTPUT sA,lower_byte  ;output count value
02  EB20                   OUTPUT sB,upper_byte
03  A002       wait_loop1: INPUT s0,event_port   ;read input
04  1001                   AND s0,  01           ;test bit0
05  9103                   JUMP Z, wait_loop1    ;wait for an event
06  83A0                   CALL inc_count16
07  A002     still_active: INPUT s0, event_port  ;read input
08  1001                   AND s0, 01            ;test bit0
09  9507                   JUMP NZ, still_active ;wait for an event to end
0A  8101                   JUMP display
0B                                               ;Subroutines
A0                         ADDRESS A0            ;increment counter
A0  4A01                   inc_count16: ADD sA, 01
A1  5B00                   ADDCY sB, 00
A2  8080                   RETURN
B0                         ADDRESS B0            ;clear counter
B0  0A00    clear_count16: LOAD sA, 00
B1  0B00                   LOAD sB, 00
B2  8080                   RETURN
B3                                               ;
```

Design Example

The following design and coding example illustrates the KCPSM in action. No attempt has been made to optimize the code. In fact, the opposite is true, in order to show as many aspects as possible in a small example.

The example shows how 8-bit data values are read, stored, sorted, and finally output in ascending order. The complete circuit diagram is shown in Figure 33. Besides illustrating reading and writing data to ports, this example shows a separate memory implemented in distributed RAM.

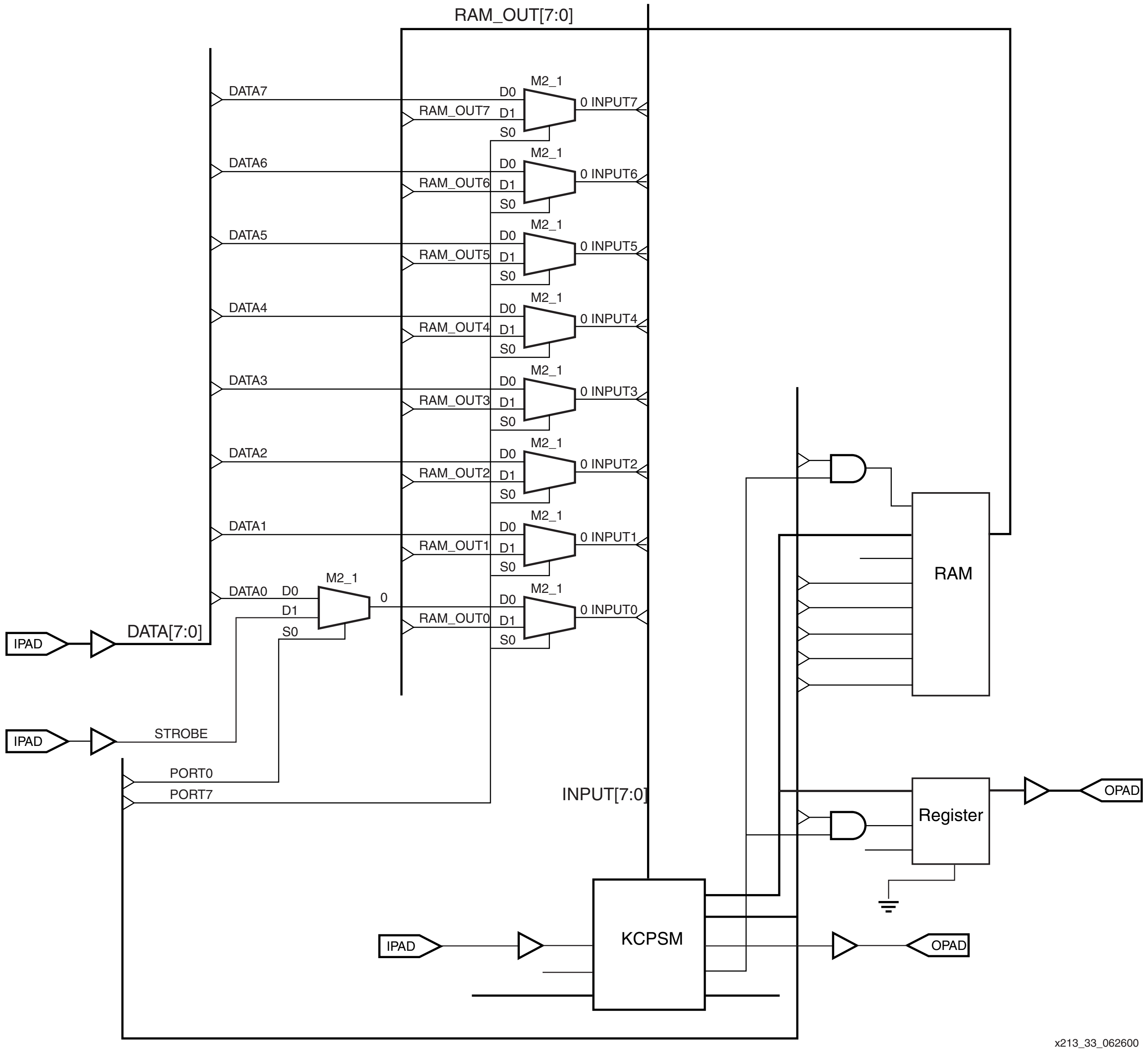


Figure 33: **Design Example: Complete Circuit**

Output Logic

The PORT bus is decoded to ensure correct access to the results register and distributed RAM module. Careful allocation of port addresses reduces the decoding logic. In all cases, the WRITE_STROBE ensures that only valid data is stored.

RAM is written for all addresses 80 hex and above. Since there are only 32 RAM locations, the program does not exceed address 9F hex for correct access.

The Data register is written when PORT5 address bit is asserted. This is address 20 Hex (32 decimal). This minimum decoding is safe as it cannot conflict with the RAM space during normal operations.

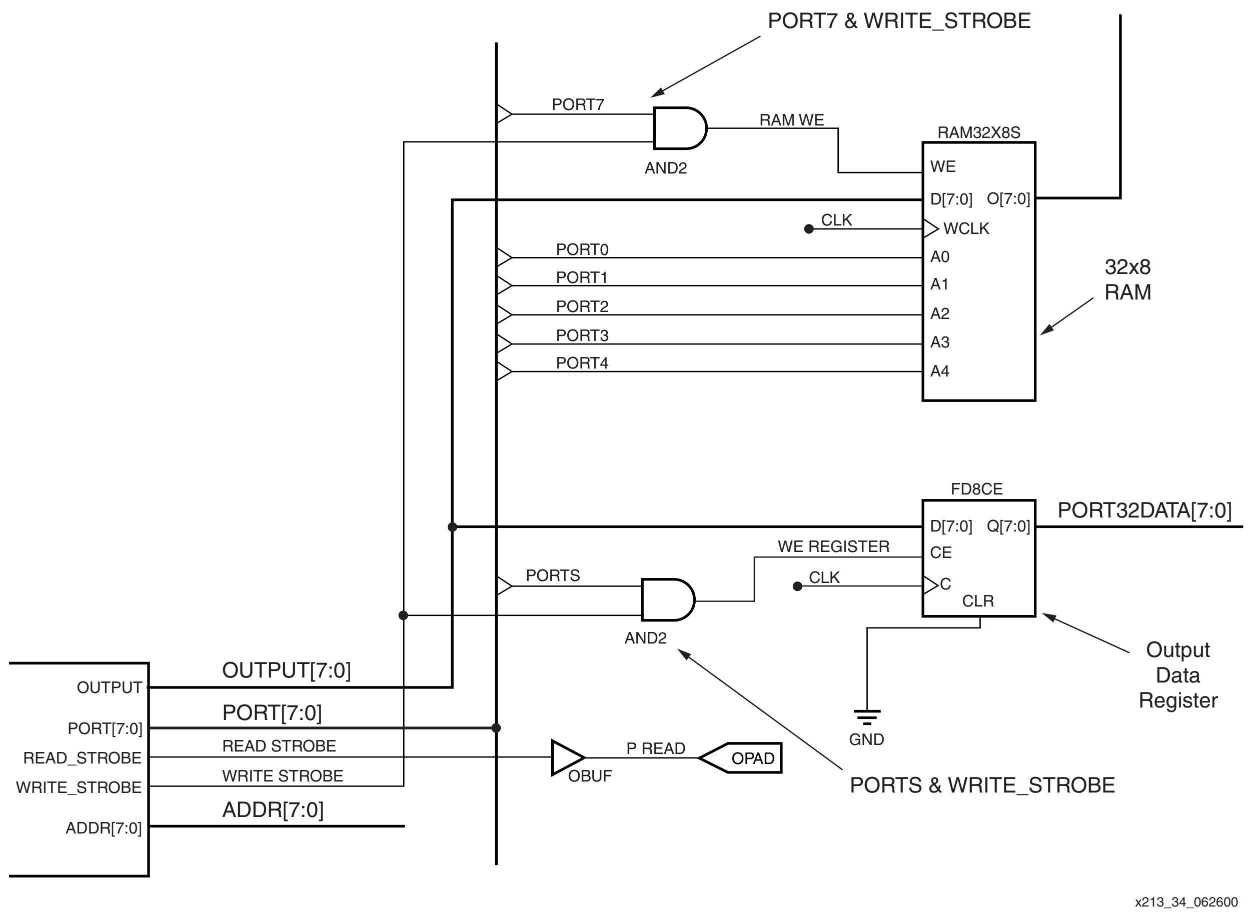


Figure 34: **Design Example: Detail for Output**

Input Logic

The PORT bus is decoded to ensure correct selection of input data to the KCPSM input port. Careful allocation of port addresses reduces the decoding logic. There is no actual requirement to use the READ_STROBE to qualify the port address.

A main 8-bit wide 2:1 multiplexer is controlled by PORT7, such that all read accesses of 80 hex and above are taken from the 32 x 8 RAM. The PORT[4:0] signals are also being used by the RAM (as an address) to provide the correct data.

Addresses below 80 hex select the input data samples. However, the LSB is provided with a further multiplexer under the control of PORT0. Hence, EVEN addresses correctly access the 8-bit input samples; ODD addresses giving access to the input strobe (but require masking the upper bits). The program uses address 00 for data and address 01 for the strobe.

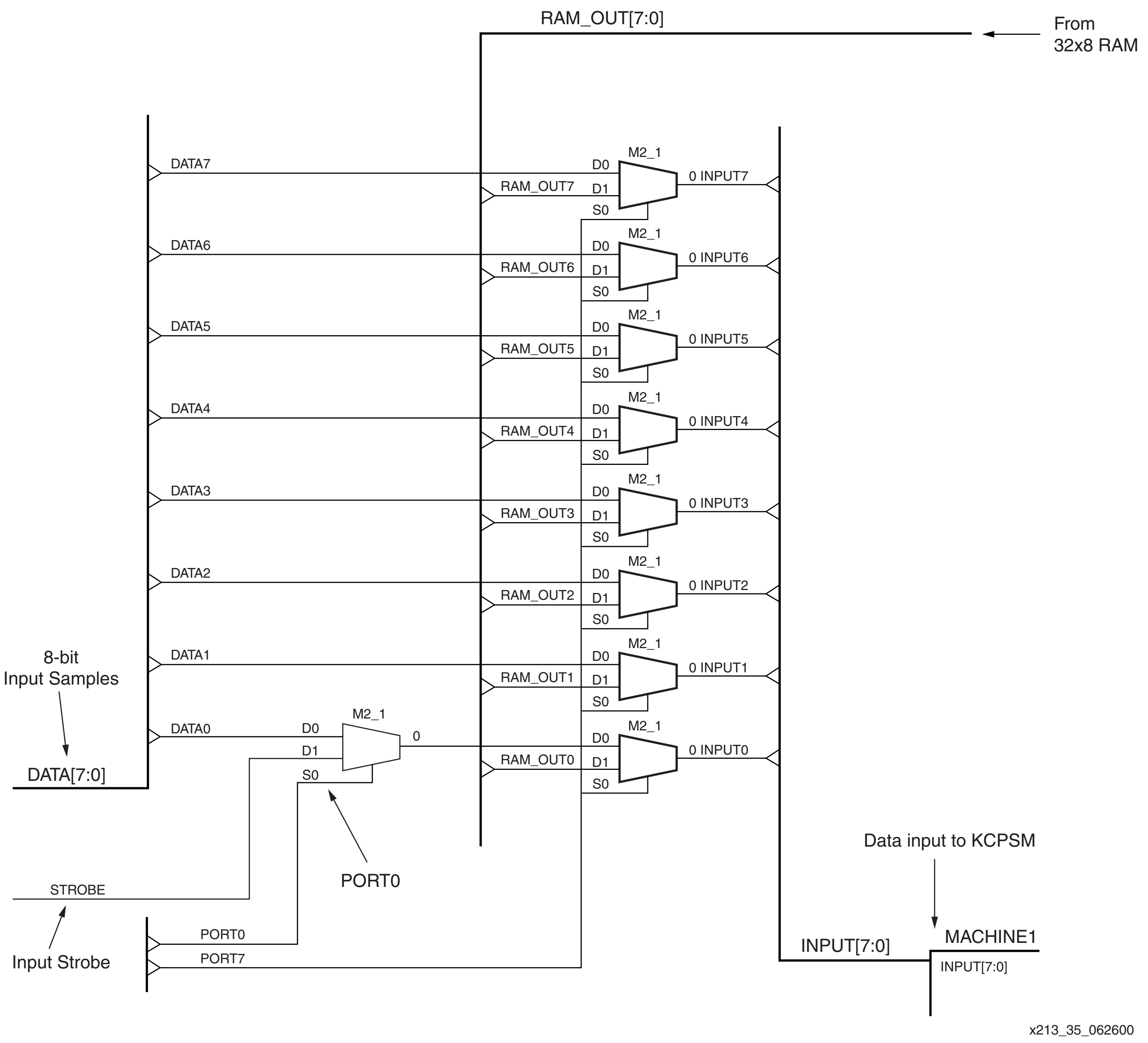


Figure 35: **Design Example: Detail for Output**

Coding Example

The PSM program is provided below (and supplied as a file). It is not an optimum implementation in order to illustrate various techniques and instructions in action. Even so, the program only consumes 41 locations of the 256 available in the program memory. The program is divided over the next few pages to add explanations.

In the first portion, the constants are used to define port addresses. A constant sets the number of samples to collect and sort. Note the number of references to this constant in the program (must be a hex value). The 20-input samples are collected and stored in external RAM. "sA" is the incremented memory pointer, and "s2" is counting (down) as the samples are being collected. Subroutine `collect_data` actually fetches a data sample into register "s1."

```
                     CONSTANT  data_input_port, 00
                     CONSTANT  strobe_input_port, 01;lsb contains strobe
                     CONSTANT  data_output_port, 20
                     CONSTANT  ram_base_addr, 80
                     CONSTANT  data_sets, 14           ;work with 20 values
                                                       ;
              start: LOAD sA,  ram_base_addr          ;point to first memory location
                     LOAD s2,  data_sets
       collect_data: CALL read_data                    ;fetch data into 's1'
                     OUTPUT s1, (sA)                   ;write to external RAM
                     ADD sA, 01                        ;next RAM address
                     SUB s2, 01                        ;check data collected
                     JUMP NZ, collect_data
```

The following section covers a standard bubble sort of the data stored in external memory. At completion, the lowest values are at the lowest addresses and highest values at highest addresses.

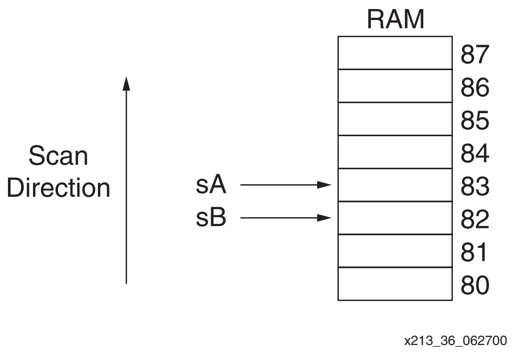


Figure 36: **Code Example, Bubble Sort Arrangement**

The memory is scanned 19 times (the number of samples less one). Registers "sA" and "sB" are used as memory pointers. Each scan of memory starts at the base address (80) and works upwards, taking with it the largest value to the highest address. Subsequent scans access one less memory location each time, as the largest values are already at the top of memory. The result port is defined by a constant.

In the following subroutine, "sC" determines the number of scans to be performed and the number still remaining to be completed. Data is read from memory and compared. Subroutine "sC" swaps data using "sA" and "sB" pointers. Note the conditional **`CALL`**. "sD" determines the number of stages to perform in each scan and the number still remaining in the current scan.

```
sort_data:      LOAD sC, data_sets          ;define number scans
                SUB sC, 01
start_scan:     LOAD sD, sC                 ;checks per scan
                LOAD sA, ram_base_addr      ;set ram pointers
scan_loop:      LOAD sB, sA
                ADD sA, 01
                INPUT s0, (sB)              ;read 2 values
                INPUT s1, (sA)
                SUB s1, s0                  ;compare
                CALL C, swop                ;swap values in s0 > s1
                SUB sD, 01                  ;check scan progress
                JUMP NZ, scan_loop
                SUB sC, 01                  ;check number of scans
                JUMP NZ, start_scan
```

Finally, the sorted values are written to the output register. The subroutines are then defined.

In the following subroutine, the loop is using "sA" to point at memory locations. The result port is defined by a constant. At the end, the whole program is repeated.

```
;write data
               LOAD sA, ram_base_addr      ;point to first memory location
               LOAD s2, data_sets
     next_out: INPUT s1, (sA)              ;read RAM
               OUTPUT s1, data_output_port ;write to port
               ADD sA, 01                  ;next RAM address
               SUB s2, 01                  ;check progress
               JUMP NZ, next_out
               ;
               JUMP start
```

The following subroutine reads the input sample in response to a strobe signal. It starts with a loop waiting for the strobe signal to be High. Next it reads the data into "s1" followed by a loop waiting for the strobe signal to be Low.

```
;subroutine to collect input data into 's1'
     read_data:INPUT s0, strobe_input_port
               AND s0, 01                  ;test strobe
               JUMP Z, read_data
               INPUT s1, data_input_port   ;read data
   end_strobe: INPUT s0, strobe_input_port
               AND s0, 01                  ;test strobe
               JUMP NZ, end_strobe
               RETURN
```

The following subroutine is called when values in memory need to be exchanged. Temporary use is made of "s0" and "s1" registers.

```
               ;subroutine to swop memory contents
          swop:INPUT s0, (sA)
               INPUT s1, (sB)
               OUTPUT s1, (sA)
               OUTPUT s0, (sB)
               RETURN
```

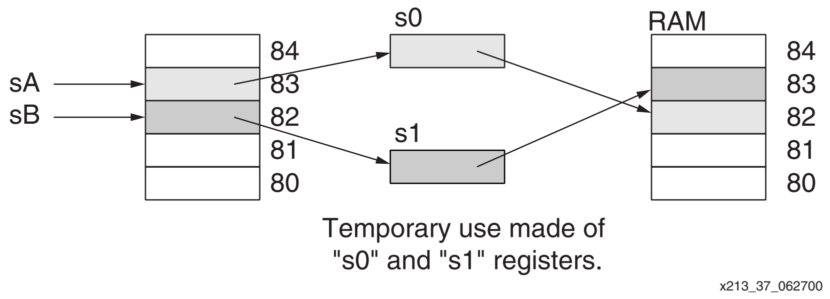


Figure 37: **Code Example, Data Swop Process**

Interrupt Handling

Effective interrupt handling, and how and when an interrupt is used, are not covered in this document. The information supplied, however, is adequate to assess the capability of KCPSM and to create interrupt-based systems.

Default State. By default, the interrupt input is disabled. This means that the entire 256 words of program space are used without any regard to interrupt handling or use of the interrupt instructions.

Enabling Interrupts. For an interrupt to take place, the ENABLE INTERRUPT command must be used. At critical stages of program execution where an interrupt is unacceptable, a DISABLE INTERRUPT is used. Since an active interrupt automatically disables the interrupt input, the interrupt service routine ends with a RETURNI instruction, which also includes the option to ENABLE or DISABLE the interrupt input as it returns to the main program.

During an interrupt, the program counter is pushed onto the stack and the values of the CARRY and ZERO flags are preserved (for restoration by the RETURNI instruction). The interrupt input is automatically disabled. Finally, the program counter is forced to address FF from which the next instruction is executed.

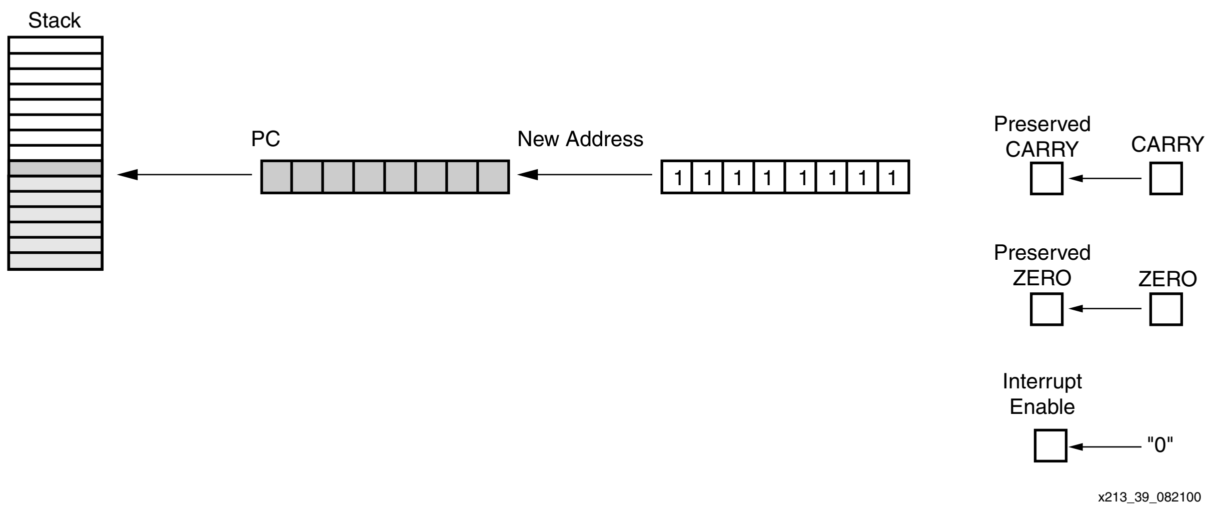


Figure 38: **Effects of an Active Interrupt**

Basics of Interrupt Handling

Since the interrupt forces the program counter to address FF, a jump vector to a suitable interrupt handling routine should be located at this address. Without such a JUMP instruction, the program will "roll over" to address zero. This is a valid way to provide KCPSM with a hardware reset, because the program counter stack is cyclic and the currently preserved addresses are ignored.

In normal cases, an interrupt service routine is provided. The routine can be located at any position in the program and jumped to by the interrupt vector located at the FF address. The service routine performs the required tasks and then ends in RETURNI with ENABLE or DISABLE.

Simple Example

The example in Figure 39 illustrates a very simple interrupt handling routine. The KCPSM is generally involved with generating waveforms to an output by writing the values 55 and AA to the **`waveform_port`** (port address 02). This is done at regular intervals and decrements a register (s0) counter in a loop.

When an interrupt is asserted, the KCPSM breaks off from the waveform generation, increments a separate counter register (sA), and writes the counter value to the **`counter_port`** (port address 04).

Figure 39 shows the external circuits used to capture port data. Note the simplified port decoding through careful selection of port addresses.

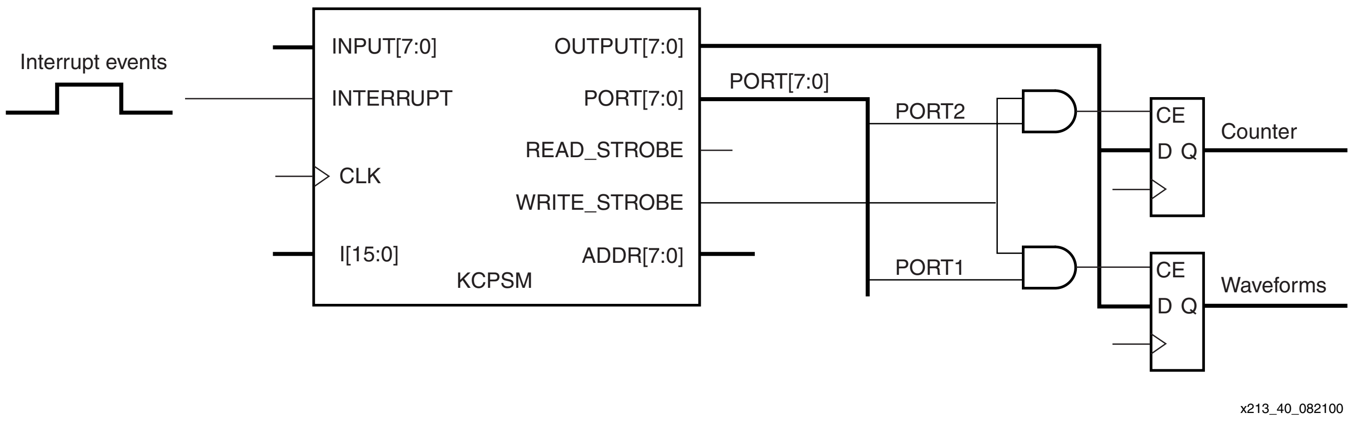


Figure 39: **Circuit for Interrupt Example**

Basic Interrupt Service Routine

The assembler log file below shows that the interrupt service routine has been forced to compile at address B0, and the waveform generation is based in the normal lower addresses. This makes it easier to observe the interrupt in action in the operation waveforms.

In the file, Addr starts the main program delay loop where most time is spent. The interrupt service routine is located at address B0 onwards. The interrupt vector is set at address FF and causes a JUMP to the service routine.

```
Addr Code

  00                                                 ;Interrupt example
  00                                                 ;
  00                       CONSTANT waveform_port,02;bit0 will be data
  00                       CONSTANT counter_port,04
  00                                                 ;
  00   0A00         start: LOAD sA, 00               ;reset interrupt counter
  01   02AA                LOAD s2, AA               ;initial output condition
  02   8030                ENABLE INTERRUPT
  03                                                 ;
  03   E202   drive_wave: OUTPUT s2, waveform_port
  04   0007                LOAD s0, 07               ;delay size
  05   6001          loop: SUB s0, 01                ;delay loop
  06   9505                JUMP NZ, loop
  07   32FF                XOR s2, FF                ;toggle waveform
  08   8103                JUMP drive_wave
  09                                                 ;
  B0                       ADDRESS B0
  B0   4A01 int_routine:  ADD sA, 01                ;increment counter
  B1   EA04                OUTPUT sA, counter_port
  B2   80F0                RETURNI ENABLE
  B3                                                 ;
  FF                       ADDRESS FF                ;set interrupt vector
  FF   81B0                JUMP int_routine
```

Interrupt Operation

Figure 40 is an actual simulation of the KCPSM performing the example program at the time of an interrupt.

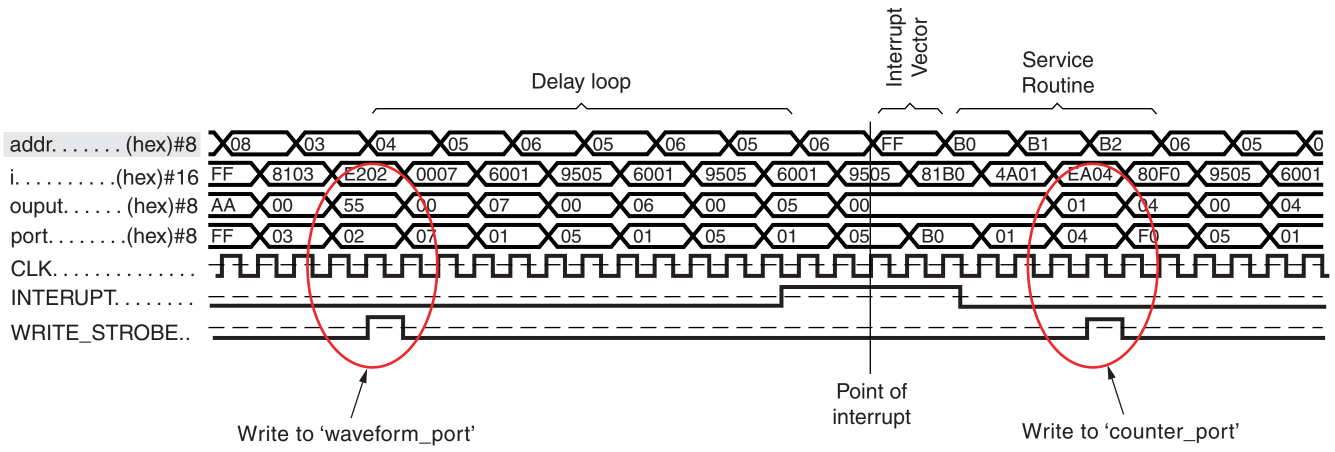

Delay loop
Interrupt Vector
Service Routine
addr. . . . . . . (hex)#8
i. . . . .(hex)#16
ouput. . . . . . (hex)#8
port. . . . .(hex)#8
CLK. . . . .
INTERUPT. . . . .
WRITE_STROBE..
08 03 04 05 06 05 06 05 06 FF B0 B1 B2 06 05 0
FF 8103 E202 0007 6001 9505 6001 9505 6001 9505 81B0 4A01 EA04 80F0 9505 6001
AA 00 55 00 07 00 06 00 05 00 01 04 00 04
FF 03 02 07 01 05 01 05 01 05 B0 01 04 F0 05 01
Write to 'waveform_port'
Point of interrupt
Write to 'counter_port'
x213_41_062700

Figure 40: **Simulation of Waveforms**

By observing the address bus, it is possible to see that the program is busy generating the waveforms and writing the 55 pattern value to the port (02). While in the delay loop, which repeats addresses 05 and 06, it receives an interrupt pulse.

It can be seen that the KCPSM took a few cycles to respond to this particular pulse (see **Timing of Interrupt Pulses**, below) before forcing the address bus to FF. From FF, the obvious **`JUMP`** to the service routine can be seen to follow, and a resulting counter value representing a first interrupt event is written to the port (04).

The operation of an interrupt in KCPSM is also visible. The last active address before the interrupt is 06. The JUMP instruction obtained at this address (op-code 9505) is *not* executed. The preserved flags are those that were set at the end of the instruction at the previous address (**`SUB s0,01`**). The RETURNI has restored the flags and returned the program to address 06, so that the instruction can be executed at last.

Timing of Interrupt Pulses

In Figure 41, a constant two cycles per instruction are maintained at all times. Since this includes an interrupt, the use of a single cycle pulse for an interrupt can be risky. However, this figure shows the exact cycle in which the interrupt is observed and the true reaction rate of the KCPSM.

The interrupt pulses, like all signals, are sampled on the rising edge of the clock. It can be seen that the pulses are active at different times relative to the address cycle, with one being captured and the other ignored.

.

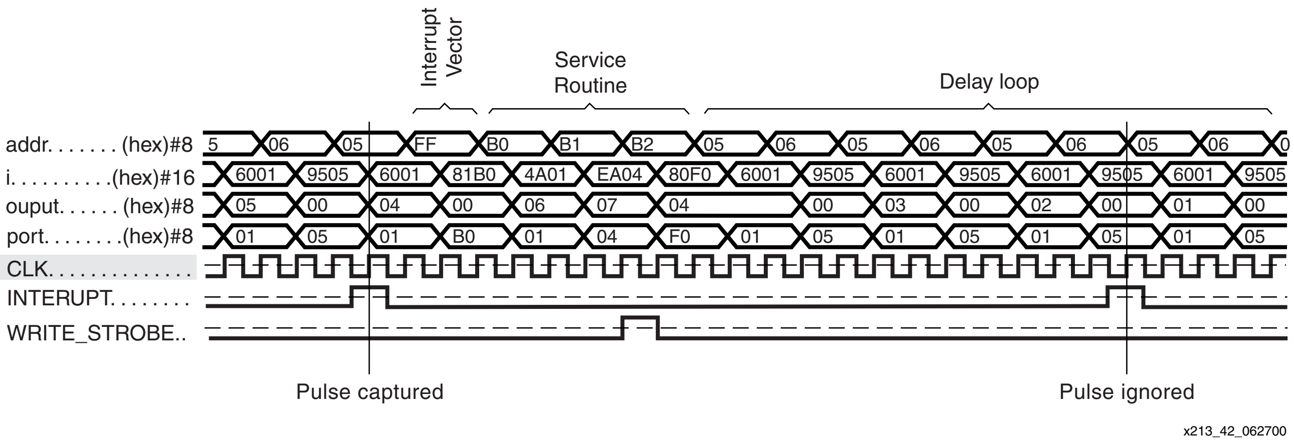


Figure 41: **Interrupt Pulse Timing**

Therefore, it is advisable that an interrupt signal be active for a minimum of two KCPSM rising clock cycle edges. An improvement would be for the interrupt service routine to acknowledge the interrupt to the external logic. There are three ways to achieve this:

1. Service routine writes to a specific port to acknowledge interrupt and reset the driving pulse. Some may consider this method wasteful.
2. Read a specific port to determine the reason for the interrupt and use READ_STROBE to reset pulse generation circuit.
3. Decode the address bus to verify that the address FF has been enforced. This is probably the most elegant solution, as it exploits the embedded use of the processor in the FPGA.

Continuous Interrupt

In the event that the interrupt remains active, or has become active again by the time the interrupt service routine has been completed, the KCPSM is able to respond immediately as soon as control has been passed back to the main program.

The following illustration uses the same example, but this time with an interrupt pulse of longer duration.

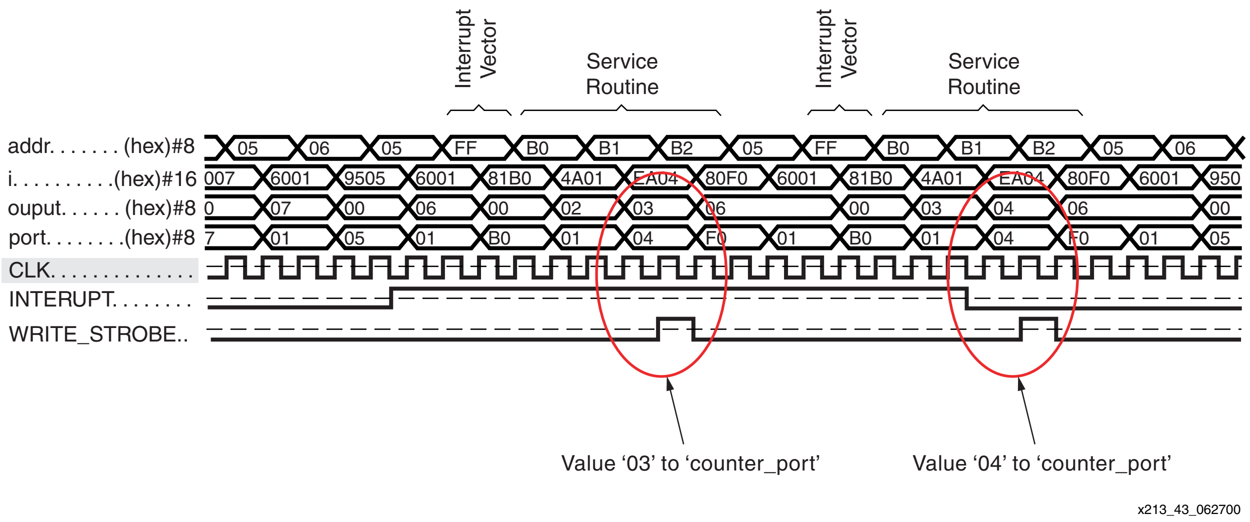

Interrupt Vector
Service Routine
Interrupt Vector
Service Routine
addr. . . . . . . (hex)#8
i. . . . .(hex)#16
ouput. . . . . . (hex)#8
port. . . . .(hex)#8
CLK. . . . .
INTERUPT. . . . .
WRITE_STROBE..
Value '03' to 'counter_port'
Value '04' to 'counter_port'
x213_43_062700

Figure 42: **Continuous Interrupt Timing**

The instruction at address 05 is abandoned by the interrupt process the first time. The RETURNI (with ENABLE) restores the flags and again fetches the instruction at address 05 (op-code 6001). However, the interrupt is still active, so the instruction is abandoned again. Finally, after this service routine, the instruction is fetched and able to execute.

Future Developments

This first version of the 8-bit microcontroller (KCPSM) for Virtex devices is supplied in a basic usable package (xapp213.zip). Much of the future of this small module depends on how it is used and the feedback received. The following list indicates likely developments, but in no preferred order.

SMART-IP Version

The SMART-IP version of the module, in which relative placement is predetermined, may be necessary to consider two layouts for the macro ("left-hand" and "right-hand") associated with the block RAM on each side of the Virtex device.

Testing Version

This version of the macro has received more testing than the SMART-IP version. The test version includes real executions in silicon of some relatively complex mathematical calculations. However, just as it is difficult to actually test software, the testing of software running on hardware is even more difficult. As more test programs are written and executed both in simulation and on hardware, potential problems and misunderstandings are revealed.

Likewise, the more the assembler and macro are used by various people, the more improvements can be implemented. Any issues found should be sent by e-mail to: **support@xilinx.com**.

VHDL Behavioral Model

An HDL-based simulation model is a useful tool in the development of embedded systems where the KCPSM can play an active role in the control of its surroundings. A behavioral model helps to demonstrate the processes being executed and provide reasonable simulation run-time performance.

Examples

Code examples are needed to build up a library of applications and useful subroutines to provide with the module. These examples demonstrate how the KCPSM works with additional logic.

Support Functions

Providing add-on block functions, such as a counter-timer, UART, and multiple interrupt control port, will bring further predefined functionality to the KCPSM designer.

Code Debugger

A very simple code emulation program is needed to allow program code operation to be verified before compiling into the embedded design. Almost certainly using the `<prog_name>.log` file output from KCPSMBLE as an input.

Conclusion

A microprocessor module does not have to be large or expensive when implemented in a Virtex or Spartan-II device. The Virtex architectural features (block memory, distributed memory, dedicated multiplexers, and carry logic) are ideal for the construction of fully embedded microprocessor modules.

The KCPSM macro is a simple 8-bit processor with an instruction set for basic control functions and data manipulation. This is achieved with just 35 CLBs and one block RAM. Even with a silicon utilization over performance objective, over 30 MIPs of processing power shows the very high performance provided by Xilinx devices. This simple 8-bit processor achieves significant data processing algorithms and control. For example, performance measurements of 150,000 16-bit multiplications per second are available for audio and control digital signal processing.

When a processor is completely embedded within an FPGA, no I/O resources are required to communicate with other modules in the same FPGA. Additionally, system design flexibility is included along with savings on PCB requirements, power consumption, and EMI. Whenever a special type of instruction is required, it can be created in hardware (other CLBs) and connected to the KCPSM as a kind of coprocessor. Indeed, there is nothing to prevent a coprocessor from being another KCPSM module. In this way, even the 256-instruction program length is not a limitation.

Finally, the optimal processor results when the instruction set is tuned to the system algorithms. By creating individual processor architectures with flexible software inside the hardware, the KCPSM shows how programmable state machines are very silicon efficient.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
09/25/00	1.0	Initial Xilinx release.
10/04/00	1.1	Minor text edits to make the copy more readable.