XILINX®

PicoBlaze™

KCPSM3

8-bit Micro Controller
for Spartan-3,
Virtex-II and Virtex-IIPRO

For Spartan-II(E) and Virtex(E) please use KCPSM
Virtex-II and Virtex-IIPro are also supported by KCPSM2

Ken Chapman

Xilinx Ltd

October 2003   Rev.7

Contents

XILINX

Limitations

Limited Warranty and Disclaimer. These designs are provided to you "as is". Xilinx and its licensors make and you receive no warranties or conditions, express, implied, statutory or otherwise, and Xilinx specifically disclaims any implied warranties of merchantability, non-infringement, or fitness for a particular purpose. Xilinx does not warrant that the functions contained in these designs will meet your requirements, or that the operation of these designs will be uninterrupted or error free, or that defects in the Designs will be corrected. Furthermore, Xilinx does not warrant or make any representations regarding use or the results of the use of the designs in terms of correctness, accuracy, reliability, or otherwise.

Limitation of Liability. In no event will Xilinx or its licensors be liable for any loss of data, lost profits, cost or procurement of substitute goods or services, or for any special, incidental, consequential, or indirect damages arising from the use or operation of the designs or accompanying documentation, however caused and on any theory of liability. This limitation will apply even if Xilinx has been advised of the possibility of such damage. This limitation shall apply not-withstanding the failure of the essential purpose of any limited remedies herein.

This module is **not** supported by general Xilinx Technical support as an official Xilinx Product.
Please refer any issues initially to the provider of the module.

Any problems or items felt of value in the continued improvement of KCPSM3 would be gratefully received by the author.

Ken Chapman
Senior Staff Engineer - Applications Specialist
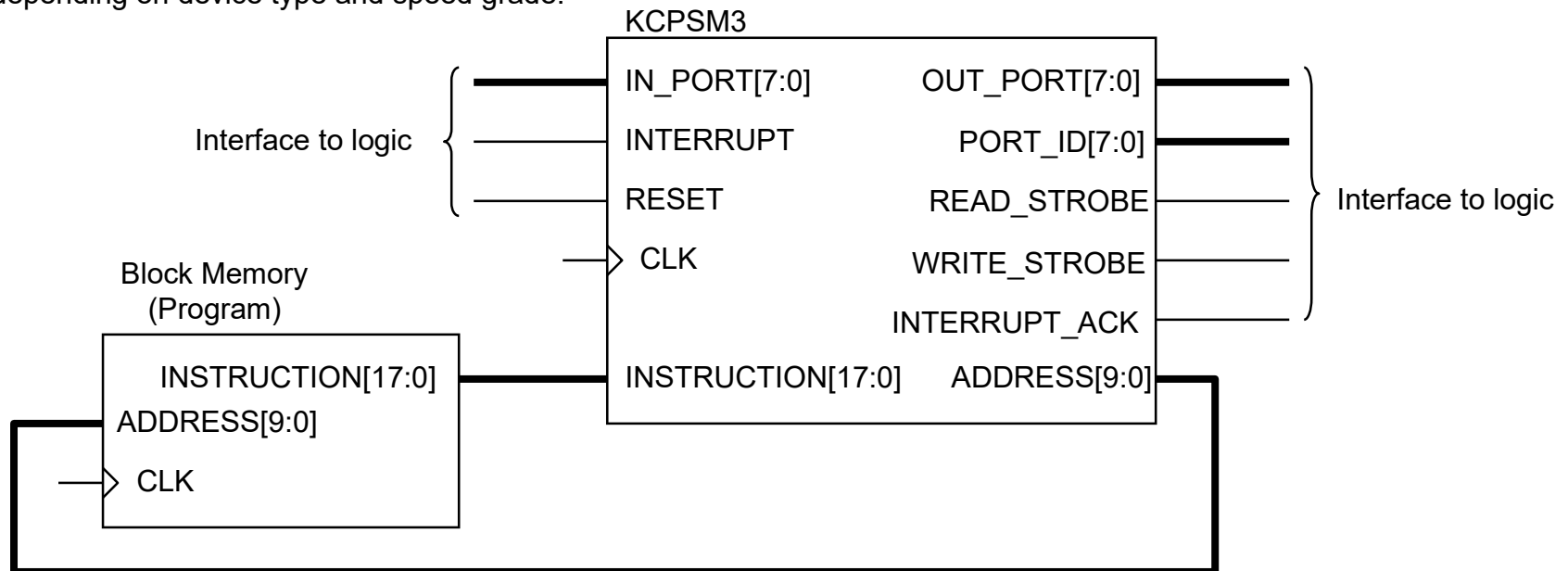email: chapman@xilinx.com

The author would also be pleased to hear from anyone using KCPSM or KCPSM2 with information about your application and how these macros have been useful.

What is KCPSM3 ?

KCPSM3 is a very simple 8-bit microcontroller primarily for the Spartan-3 devices but also suitable for use in Virtex-II and Virtex-IIPRO devices. Although it could be used for processing of data, it is most likely to be employed in applications requiring a complex, but non-time critical state machine. Hence it has the name of '(K)constant Coded Programmable State Machine'.

This revised version of popular KCPSM macro has still been developed with one dominant factor being held above all others - Size! The result is a microcontroller which occupies just **96 Spartan-3 Slices** which is just 5% of the XC3S200 device and less than 0.3% of the XC3S5000 device. Together with this small amount of logic, a single block RAM is used to form a ROM store for a program of up to 1024 instructions. Even with such size constraints, the performance is respectable at approximately **43 to 66 MIPS** depending on device type and speed grade.

KCPSM3

Interface to logic
- IN_PORT[7:0]
- INTERRUPT
- RESET
- CLK

- OUT_PORT[7:0]
- PORT_ID[7:0]
- READ_STROBE
- WRITE_STROBE
- INTERRUPT_ACK

Interface to logic

Block Memory (Program)
- INSTRUCTION[17:0]
- ADDRESS[9:0]
- CLK

- INSTRUCTION[17:0]
- ADDRESS[9:0]

One of the most exciting features of the KCPSM3 is that it is totally embedded into the device and requires no external support. The very fact that ANY logic can be connected to the module inside the Spartan-3 or Virtex-II device means that any additional features can be added to provide ultimate flexibility. It is not so much what is inside the KCPSM3 module that makes it useful, but the environment in which it lives.

XILINX

KCPSM3 is small!

KCPSM3 is supplied as VHDL and as a pre-compiled soft macro which is handled by the place and route tools to merge with the logic of a design. In large devices, the KCPSM3 is virtually free! The potential to place multiple KCPSM3 within a single design is obvious. Whenever a non time critical complex state machine is required, this macro is easy to insert and greatly simplifies design.
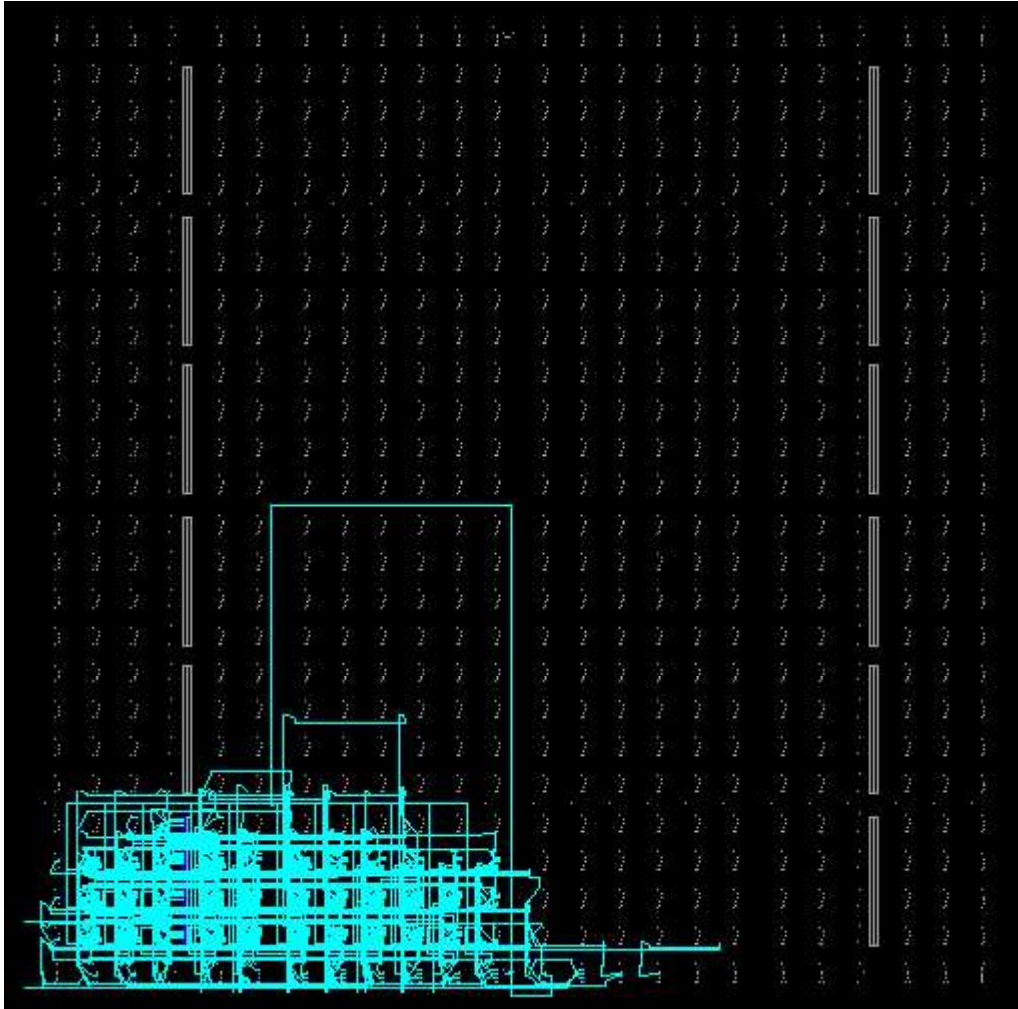
This plot from the FPGA Editor viewer shows the macro in isolation within the XC3S200 Spartan-3 device.

96 Slices

5% of XC3S200 Spartan-3 device

~87MHz in -4 Speed Grade

~43.5 MIPS

KCPSM3 is small!

This plot from the Xilinx Floorplanner shows the same implementation of KCPSM3 in an XC3S200 Spartan-3 device. This makes it easier to appreciate the actual logic resources required by the macro without the interconnect obscuring the detail.
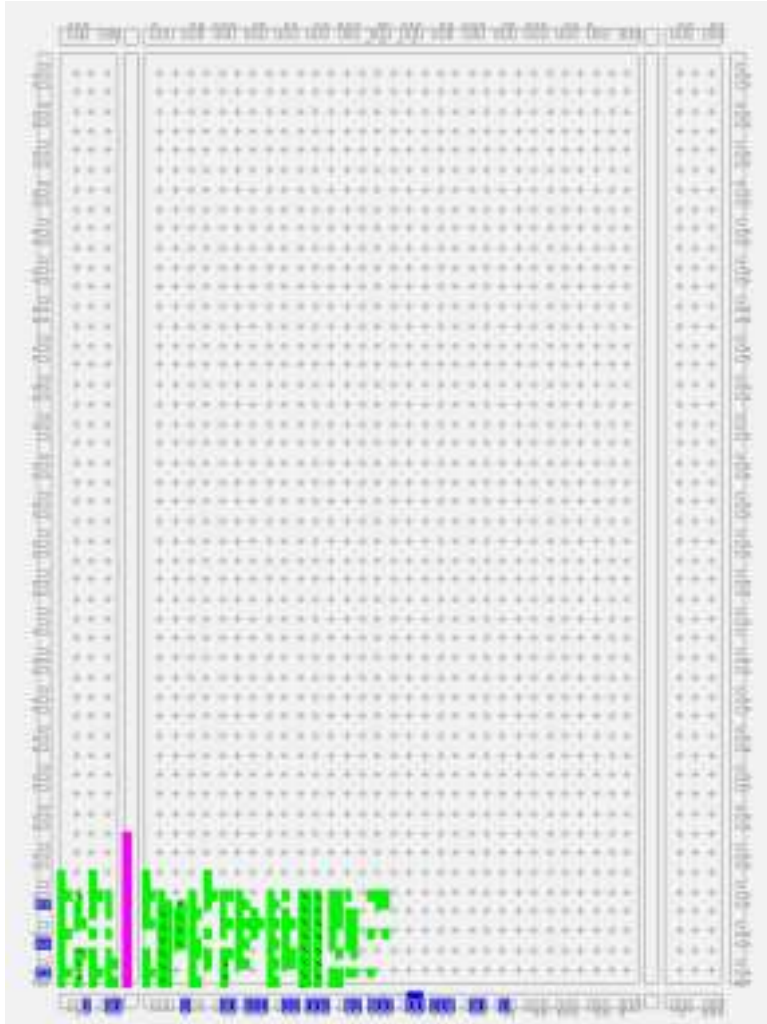
The placement in this Floorplanner view was achieved using a simple area constraint in the project UCF file.

```
INST processor_* LOC=SLICE_X0Y0:SLICE_X19Y4;
```

Such constraints are not required in normal designs and it has only been used in this case because so little of the device is occupied. Experiments have shown that placement constraints have very little effect on performance.

The FPGA Editor view shown to the right was the result when no constraints were used. The size is still 96 slices but this is now a little less obvious! The performance was actually a little higher than when using the area constraint indicating that a 'tidy' design is not always the fastest!

XILINX
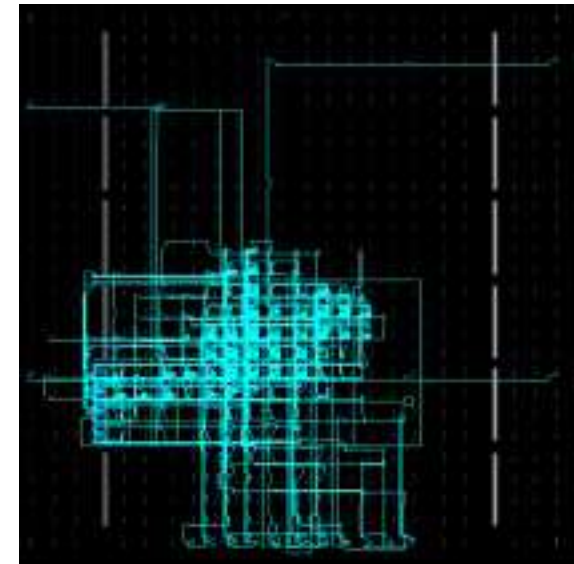
Size and Performance

The following device resource information is taken from the ISE reports for the KCPSM3 macro in an XC3S200 device. The reports reveal the features that are utilised and the efficiency of the macro. The 96 'slices' reported by the MAP process in this case may reduce to the minimum of 89 'slices' when greater packing is used to fit a complete design into a device.

XST Report

```
LUT1       : 2
LUT2       : 6       109 LUTs
LUT3       : 68      (55 slices)
LUT4       : 33

MUXCY      : 39      Carry and MUX logic
MUXF5      : 9       (Free with LUTs)
XORCY      : 35

FD         : 24
FDE        : 2
FDR        : 30      76 Flip_flops
FDRE       : 8       (Free with LUTs)
FDRSE      : 10
FDS        : 2

RAM16X1D   : 8    ── Register bank (8 slices)
RAM32X1S   : 10   ── Call/Return Stack (10 slices)
RAM64X1S   : 8    ── Scratch Pad Memory (16 slices)

            Total = 89 Slices
```

MAP Report

```
Number of occupied Slices : 96 out of 1920    5%
Number of Block RAMs      :  1 out of   12    8%
Total equivalent gate count for design:  74,814
```

12 × KCPSM3 can fit into the XC3S200 device (40% of the logic slices remaining). An equivalent gate count of 897,768 gates in a 200,000 gate device!

TRACE Report

```
Device,speed: xc3s200,-4 (PREVIEW 1.22 2003-03-16)
Minimum period:  11.403ns
(Maximum frequency:  87.696MHz)
```

43.8 MIPS

TRACE Report for Virtex-IIPRO

```
Device,speed: xcvp2,-7 (ADVANCED 1.76 2003-03-16)
Minimum period:  7.505ns
(Maximum frequency:  133.245MHz)   66.6 MIPS
```

XILINX

KCPSM3 Architecture

16 Registers 8-bit

sF	s7
sE	s6
sD	s5
sC	s4
sB	s3
sA	s2
s9	s1
s8	s0

IN_PORT[7:0]

Port Address Control

PORT_ID[7:0]
READ_STROBE
WRITE_STROBE

pp

OUT_PORT[7:0]

ALU
Arithmetic Logical Shift Rotate

kk

PARITY

Scratch Pad Memory 64-Bytes

ss

18 bit instruction word
8 bit data path
8 bit port address
10 bit program address
Constants

ZERO & CARRY flags

Interrupt Shadow Flags

INTERRUPT
INTERRUPT_ACK

Interrupt Control

Program Flow Control

aaa

Program Counter

ADDRESS[9:0]

Program ROM/RAM

1024 words

INSTRUCTION[17:0]

Operational control & Instruction decoding

RESET
CLK

aaa / pp / ss / kk

Program Counter Stack

XILINX

KCPSM3 Feature Set

Features new to KCSPM3

KCPSM3 is a very simple processor architecture and anyone familiar with PSM, KCSPM or KCSPM2 will recognise that this is just the latest in a close family of 8-bit programmable state machines (see 'PicoBlaze Comparison'). The motivation to develop this variant was the release of Spartan-3 devices and the highly constructive feedback from so many users of its predecessors.

Spartan-3 has adopted the 18Kbit Block RAM elements previously seen in the Virtex-II devices. This enables KCPSM3 to support programs up to 1024 locations which overcomes the most commonly encountered limit of KCPSM with Spartan-II(E).

At the risk of making KCPSM3 appear more complex than previous versions, some additional features have been included to address the most popular requests. COMPARE and TEST instructions enable register contents to be interrogated without changing their contents. The TEST instruction also calculates PARITY, useful for many communication applications. A 64-byte internal scratch pad memory allows many more variables to be held internally, more intuitive programs to be written and will typically eliminate requirement for memory attached to the I/O ports. Finally, an interrupt acknowledgement signal is provided.

The additional features make KCPSM3 26% larger than KCPSM and 14% larger than KCPSM2. However, It is expected that the additional features will enable more efficient programs to be written and for designs to require less peripheral logic.

Program Size

KCPSM3 supports a program up to a length of 1024 instructions utilising one block memory. Requirements for larger program space are typically addressed by using multiple KCPSM3 processors each with an associated block memory to distribute the various system tasks. Programs requiring significantly more memory are normally the domain of a full data processor such as MicroBlaze with its C-language programming support.

16 General Purpose Registers.

There are 16 general purpose registers of 8-bits specified as 's0' through to 'sF' which may be renamed in the assembler code. All operations are completely flexible about the use of registers with no registers reserved for special tasks or having any priority over any other register. There is no accumulator as any register can be adopted for this task.

XILINX

KCPSM3 Feature Set

ALU

The Arithmetic Logic Unit (ALU) provides many simple operations expected in an 8-bit processing unit.
All operations are performed using an operand provided from any register (sX). The result is returned to the same register.
For operations requiring a second operand, a second register can be specified (sY) or a constant 8-bit value (kk) can be supplied. The ability to specify any constant value with no additional penalty to program size or performance enhances the simple instruction set i.e. the ability to 'ADD 1' is the same as a dedicated INCREMENT operation.
Addition (ADD) and Subtraction (SUB) have the option to include the carry flag as an input (ADDCY and SUBCY) for the support of arithmetic operations requiring more than 8-bits.
LOAD, AND, OR and XOR bit-wise operators provide ability to manipulate and test values.
Comprehensive SHIFT and ROTATE group.
COMPARE and TEST instructions enable register contents to be tested without altering their contents and determine PARITY.

Flags and Program Flow Control

The results of ALU operations determine the status of the ZERO and CARRY flags. The ZERO flag is set whenever the ALU result has all bits reset (00_{16}). The CARRY flag is set when there is an overflow from an arithmetic operation. It is also used to capture the bit moved out of a register during shift and rotate instructions. During a TEST instruction, the carry flag is used to indicate if the 8-bit temporary result has ODD PARITY.

This status of the flags can be used to determine the execution sequence of the program using conditional and non-conditional program flow control instructions. JUMP commands are used to specify absolute addresses (aaa) within the program space. CALL and RETURN commands provide sub-routine facilities for commonly used sections of code. A CALL is made to an absolute address (aaa) and an internal program counter stack preserves the associated address required by the RETURN instruction. The stack supports up to 31 nested subroutine levels.

Reset

The RESET input forces the processor back into the initial state. The program will execute from address '000' and interrupts will be disabled. The status flags and CALL/RETURN stack will also be reset. Note that register contents are not affected.

XILINX

KCPSM3 Feature Set

Input/Output

KCPSM3 effectively has 256 input ports and 256 output ports. The port being accessed is indicated by an 8-bit address value provided on the 'PORT_ID'. The port address can be specified in the program as an absolute value (pp), or may be indirectly specified as the contents of any of the 16 registers ((sY)).

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 pulse being output on the READ_STROBE. It is not always necessary to use this signal in the input interface logic, but it can be useful to indicate that data has been acquired by the processor. During an 'OUTPUT', the contents of any of the 16 registers are transferred to the output port. An output operation is indicated by a pulse being output on the WRITE_STROBE. This strobe signal will be used by the interface logic to ensure that only valid data is passed to external systems. Typically, WRITE_STROBE will be used as a clock enable or write enable (see 'READ and WRITE STROBES').

Scratch Pad Memory

This is an internal 64 byte general purpose memory. The contents of any of the 16 registers can be written to any of the 64 locations using a STORE instruction. The complementary FETCH instruction allows the contents of any of the 64 memory locations to be written to any of the 16 registers. This allows a much greater number of variables to be held within the boundary of the processor and tends to reserve all of the I/O space for real inputs and output signals.

The 6-bit address to specify a scratch pad memory location can be specified in the program as an absolute value (ss), or may be indirectly specified as the contents of any of the 16 registers (sY). Only the lower 6-bits of the register are used, so care must be taken not to exceed the 00 - $3F_{16}$ range of the available memory.

Interrupt

The processor provides a single INTERRUPT input signal. Simple logic can be used to combine multiple signals if required. Interrupts are disabled (masked) by default, and are then enabled and disabled under program control. An active interrupt forces KCPSM3 to initiate a 'CALL 3FF' (a subroutine call to the last program memory location) from where the user can define a suitable jump vector to an Interrupt Service Routine (ISR). At this time, a pulse is generated on the INTERRUPT_ACK output, the ZERO and CARRY flags are automatically preserved and any further interrupts are disabled. The 'RETURNI' instruction ensures that the end of an ISR restores the status of the flags and specifies if future interrupts will be enabled or disabled.

Constant(k) Coded

The KCPSM3 is in many ways a machine based on Constants…....

Constant Values

Constant values may be specified for use in most 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 KCPSM3.
- Constant address values for controlling the execution sequence of the program.
- Constant address values for accessing internal scratch pad memory.

The KCPSM3 instruction set coding has been 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 will execute over 2 clock cycles.
Such constant execution rate is of great value when determining the execution time of a program particularly when embedded into a real time situation.

Constant Program Length

The program length is 1024 instructions and therefore conforms to the 1024x18 format of a single Spartan-3, Virtex-II or Virtex-IIPRO Block RAM. This means that all address values are specified as 10-bits contained within the instruction coding (the assembler supports line labels to simplify the writing of programs). The fixed memory size promotes a consistent level of performance from the module. See also 'Sharing Program Space'.

Using KCPSM3 (VHDL)

The principle method by which KCPSM3 will be used is in a VHDL design flow. The KCPSM3 macro is provided as source VHDL (kcspm3.vhd) which has been written to provide an optimum and predictable implementation in a Spartan-3 or Virtex-II(PRO) device. The code is suitable for implementation and simulation of the macro. It has been developed and tested using XST for implementation and ModelSim for simulation. The code should not be modified in any way.

VHDL Component declaration of KCPSM3

```
component kcpsm3
    Port (        address : out std_logic_vector(9 downto 0);
              instruction : in std_logic_vector(17 downto 0);
                  port_id : out std_logic_vector(7 downto 0);
             write_strobe : out std_logic;
                 out_port : out std_logic_vector(7 downto 0);
              read_strobe : out std_logic;
                  in_port : in std_logic_vector(7 downto 0);
                interrupt : in std_logic;
            interrupt_ack : out std_logic;
                    reset : in std_logic;
                      clk : in std_logic);
    end component;
```

VHDL Component instantiation of KCPSM3

```
processor: kcpsm3
    port map(        address => address_signal,
                 instruction => instruction_signal,
                     port_id => port_id_signal,
                write_strobe => write_strobe_signal,
                    out_port => out_port_signal,
                 read_strobe => read_strobe_signal,
                     in_port => in_port_signal,
                   interrupt => interrupt_signal,
               interrupt_ack => interrupt_ack_signal,
                       reset => reset_signal,
                         clk => clk_signal);
```

XILINX

Connecting the Program ROM

The principle method by which KCPSM3 program ROM will be used is in a VHDL design flow. The KCPSM3 assembler will generate a VHDL file in which a block RAM and its initial contents are defined (see assembler notes for more detail). This VHDL can be used for implementation and simulation of the processor. It has been developed and tested using XST for implementation and ModelSim for simulation.

VHDL Component declaration of program ROM

```
component prog_rom
    Port (        address : in std_logic_vector(9 downto 0);
              instruction : out std_logic_vector(17 downto 0);
                      clk : in std_logic);
    end component;
```

VHDL Component instantiation of program ROM

```
program: prog_rom
    port map(       address => address_signal,
                instruction => instruction_signal,
                        clk => clk_signal);
```

Note - The name of the program ROM (shown as 'prog_rom' in the above examples) will depend on the name of your program. For example, if your program file was called 'phone.psm', then the assembler will generate a program ROM definition file called 'phone.vhd'.

To aid with development, a VHDL file called 'embedded_kcpsm3.vhd' is also supplied in which the KCPSM3 macro is connected to its associated block RAM program ROM. This entire module can be embedded in the design application, or simply used to cut and paste the component declaration and instantiation information into your own code.

 Note: It is recommended that 'embedded_kcpsm3.vhd' is used for the generation of an ECS schematic symbol.

XILINX

Verilog and System Generator

Although the primary design flow is VHDL, KCPSM3 can be used in any design flow supported by Xilinx. The assembler also generates program memory definition files suitable for Verilog and the Simulink based System Generator design flows.

<filename>.v - The assembler generates a Verilog file in which a block RAM and its initial contents are defined (see assembler notes for more detail). This Verilog can be used for implementation and simulation of the processor. The kcspm3.ngc file will be used to define the processor.

kcpsm3.ngc - The NGC file provided was generated by synthesising the kcpsm3.vhd file with XST (without inserting I/O buffers). This file can be used as a 'black box' in a Spartan-3, Virtex-II or Virtex-IIPRO design, and it will be merged with the rest of your design during the translate phase (ngdbuild). Note that busses are defined in the style 'IN_PORT<7:0>' with individual signals 'in_port_0' through to 'in_port_7'.

<filename>.m - The assembler generates a m-function used to define the contents of a System Generator memory block within the MATLAB Simulink environment. (see System Generator documentation for more information on this design flow).

<filename>.coe - The COE file generated by the assembler is suitable for use with the Xilinx Core Generator. The file defines the initial contents of a block ROM. The files generated by Core Generator can then be used as normal in your chosen design flow and connected to the kcpsm3 'black box' in your design (see assembler notes for more details).

Simulation Models

If the NGC file is used in the design flow, then some form of back annotated description will be required for simulation of your design in order to fill in the 'black box' details. The following command can be used to generate a Verilog simulation model (see the Xilinx online manuals for more details - Synthesis and Simulation Design Guide - section 6).

```
ngd2ver kcspm3.ngd sim_model_kcpsm3.v
```

KCPSM3 Instruction Set

'X' and 'Y' refer to the definition of the storage registers 's' in the range 0 to F.
'kk' represents a constant value in the range 00 to FF.
'aaa' represents an address in the range 000 to 3FF.
'pp' represents a port address in the range 00 to FF.
'ss' represents an internal storage address in the range 00 to 3F.

Program Control Group

```
JUMP aaa
JUMP Z,aaa
JUMP NZ,aaa
JUMP C,aaa
JUMP NC,aaa

CALL aaa
CALL Z,aaa
CALL NZ,aaa
CALL C,aaa
CALL NC,aaa

RETURN
RETURN Z
RETURN NZ
RETURN C
RETURN NC
```

Note that call and return supports
  up to a stack depth of 31.

Arithmetic Group

```
ADD sX,kk
ADDCY sX,kk
SUB sX,kk
SUBCY sX,kk
COMPARE sX,kk

ADD sX,sY
ADDCY sX,sY
SUB sX,sY
SUBCY sX,sY
COMPARE sX,sY
```

Interrupt Group

```
RETURNI ENABLE
RETURNI DISABLE

ENABLE INTERRUPT
DISABLE INTERRUPT
```

Logical Group

```
LOAD sX,kk
AND sX,kk
OR sX,kk
XOR sX,kk
TEST sX,kk

LOAD sX,sY
AND sX,sY
OR sX,sY
XOR sX,sY
TEST sX,sY
```

Storage Group

```
STORE sX,ss
STORE sX,(sY)
FETCH sX,ss
FETCH sX,(sY)
```

Shift and Rotate Group

```
SR0 sX
SR1 sX
SRX sX
SRA sX
RR sX

SL0 sX
SL1 sX
SLX sX
SLA sX
RL sX
```

Input/Output Group

```
INPUT sX,pp
INPUT sX,(sY)
OUTPUT sX,pp
OUTPUT sX,(sY)
```

JUMP

Under normal conditions, the program counter (PC) increments to point to the next instruction. The address space is fixed to 1024 locations (000 to 3FF hex) and therefore the program counter is 10 bits wide. It is worth noting that the top of memory is 3FF hex and will increment to 000.

PC ← PC + 1   Normal Instruction
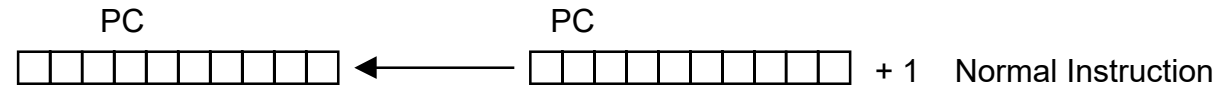
The JUMP instruction may be used to modify this sequence by specifying a new address. However, the JUMP instruction may be conditional. A conditional JUMP will only be performed if a test performed on either the ZERO flag or CARRY flag is valid. The JUMP instruction has no effect on the status of the flags.

Condition not valid → PC + 1

PC

New Address
a a a a a a a a a a

Unconditional or condition valid

```
JUMP aaa
JUMP Z,aaa
JUMP NZ,aaa
JUMP C,aaa
JUMP NC,aaa
```
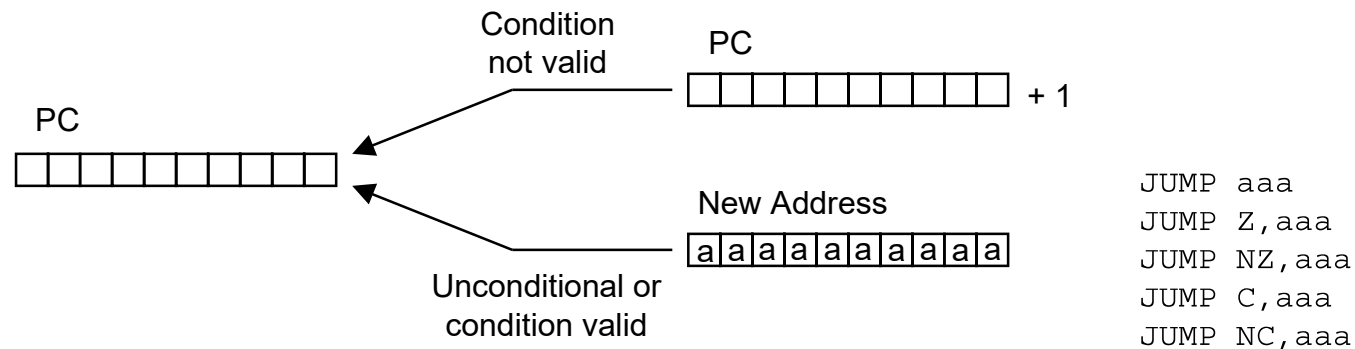
Each JUMP instruction must specify the 10-bit address as a 3 digit hexadecimal value. The assembler supports labels to simplify this process.

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0				a	a	a	a	a	a	a	a	a	a

Bit 12   0 - UNCONDITIONAL
          1 - CONDITIONAL

Bit 11	Bit 10	Condition
0	0	if Zero
0	1	if NOT Zero
1	0	if Carry
1	1	if NOT Carry

XILINX

CALL

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


```
CALL aaa
CALL Z,aaa
CALL NZ,aaa
CALL C,aaa
CALL NC,aaa
```
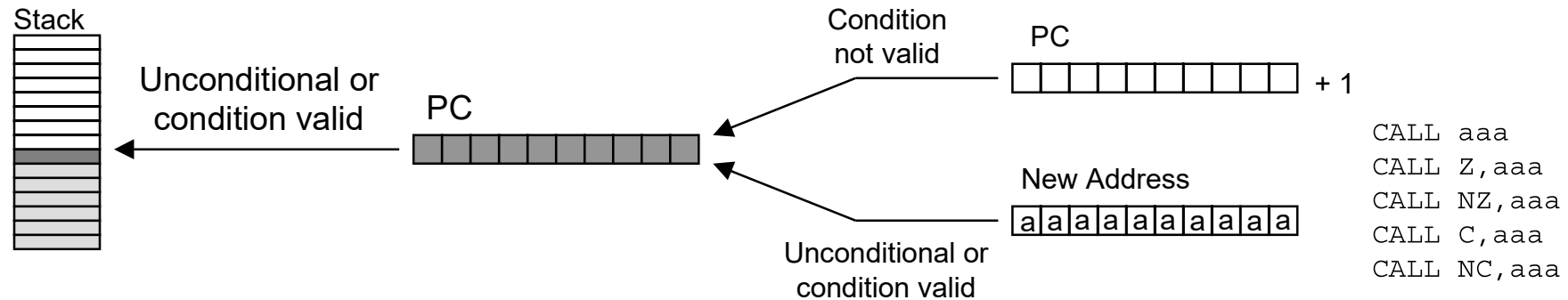
The program counter stack supports a depth of 31 address values. This enables nested 'CALL' sequences to a depth of 31 levels to be performed. However, the stack will also be used during an interrupt operation and hence at least one of these levels should be reserved when interrupts are enabled. The stack is implemented as a separate cyclic buffer. When the stack becomes full, it simply overwrites the oldest value. Hence it is not necessary to reset the stack pointer when performing a software reset. This also explains why there are no instructions to control the stack and why no other memory needs to be reserved or provided for the stack.

Each CALL instruction must specify the 10-bit address as a 3 digit hexadecimal value. The assembler supports labels to simplify this process.

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0				a	a	a	a	a	a	a	a	a	a

Bit 12   0 - UNCONDITIONAL
          1 - CONDITIONAL

Bit 11	Bit 10	Condition
0	0	if Zero
0	1	if NOT Zero
1	0	if Carry
1	1	if NOT Carry
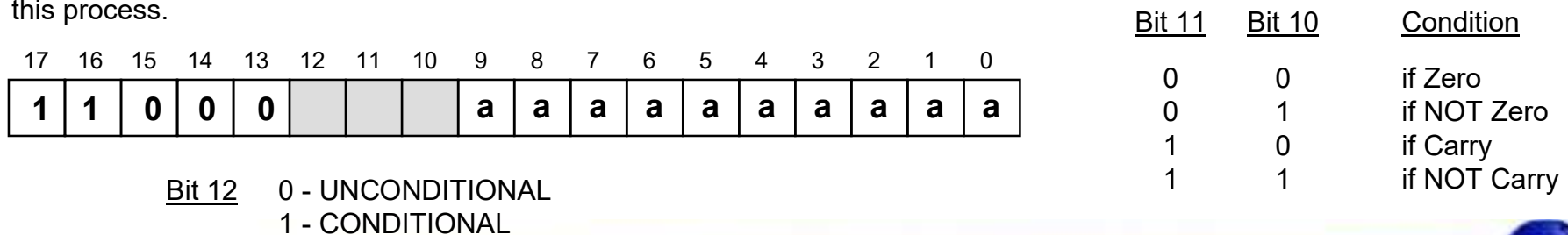
RETURN

The RETURN instruction is the complement to the CALL instruction. The RETURN instruction may also be conditional. In this case the new program counter (PC) value will be formed internally by incrementing the last value on the program address stack. This ensures that the program will execute the instruction following the CALL instruction which resulted in the subroutine. The RETURN instruction has no effect on the status of the flags.

Condition
not valid

PC

PC + 1

Stack

+ 1

RETURN
RETURN Z
RETURN NZ
RETURN C
RETURN NC

Unconditional or
condition valid
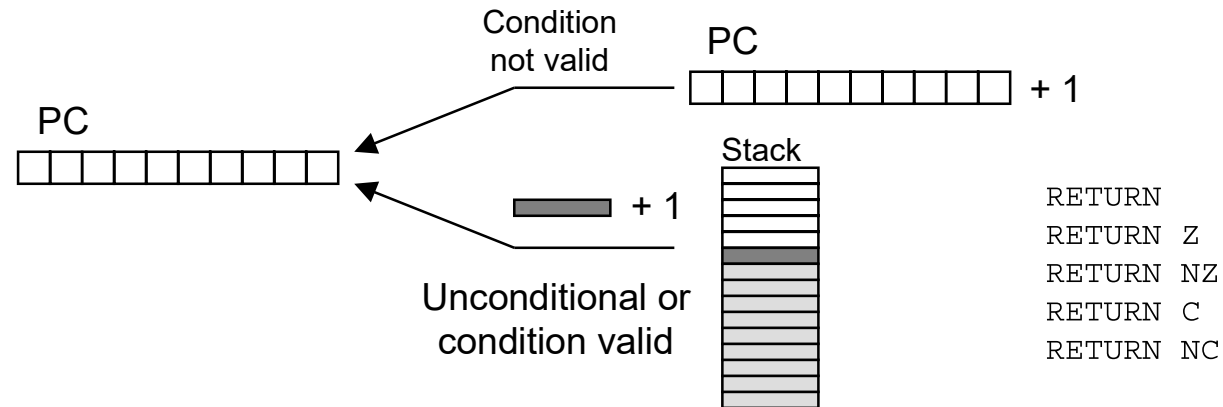
It is the responsibility of the programmer to ensure that a RETURN is only performed in response to a previous CALL instruction such that the program counter stack contains a valid address. The cyclic implementation of the stack will continue to provide values for RETURN instructions which can not be defined.

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1				0	0	0	0	0	0	0	0	0	0

Bit 12   0 - UNCONDITIONAL
          1 - CONDITIONAL

Bit 11	Bit 10	Condition
0	0	if Zero
0	1	if NOT Zero
1	0	if Carry
1	1	if NOT Carry

XILINX

RETURNI

The RETURNI instruction is a special variation of the RETURN instruction which should be used to conclude an interrupt service routine. The RETURNI is unconditional and therefore will always load the program counter (PC) with the last address on the program counter stack (the address is not incremented in this case since the instruction at the address stored will need to be executed). The RETURNI instruction restores the flags to the condition they were in at the point of interrupt. The RETURNI also determines the future ability of interrupts using ENABLE and DISABLE as an operand.
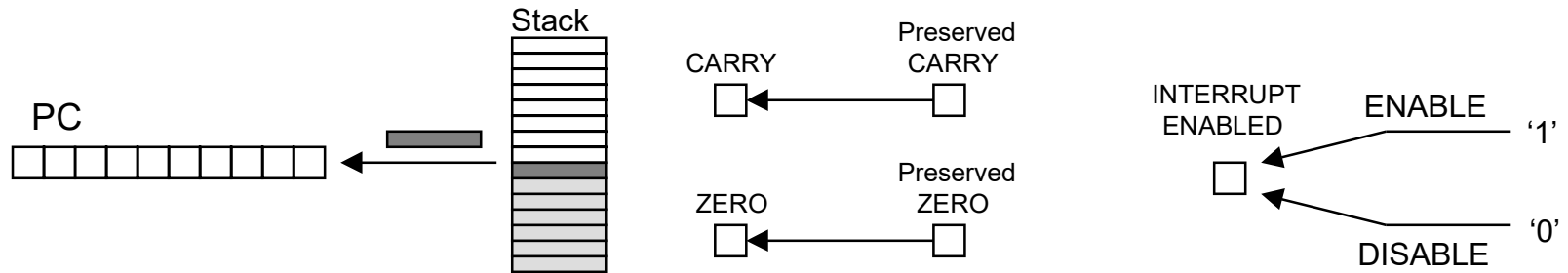
It is the responsibility of the programmer to ensure that a RETURNI is only performed in response to an interrupt. Each RETURNI must specify if further interrupt is to be enabled or disabled.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RETURNI ENABLE	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RETURNI DISABLE	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

ENABLE/DISABLE INTERRUPT

These instructions are used to set and reset the INT_ENABLE flag. Before using ENABLE INTERRUPT a suitable interrupt routine must be associated with the interrupt address vector (located at address 3FF). Interrupts should never be enabled whilst performing an interrupt service routine.

INT_ENABLE □ ◀── ENABLE '1'
              ◀── DISABLE '0'
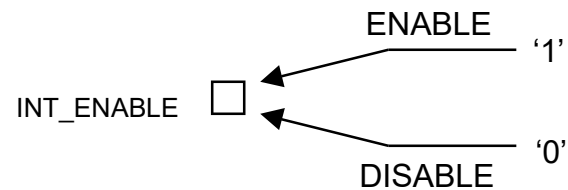
Interrupts are masked when the INT_ENABLE flag is low. This is the default state of the flag following device configuration or a KCPSM3 reset. The INT_ENABLE is also reset during an active interrupt.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ENABLE INTERRUPT	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DISABLE INTERRUPT	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

LOAD

The LOAD instruction provides a method for specifying the contents of any register. The new value can be a constant, or the contents of any other register. The LOAD instruction has no effect on the status of the flags.

sX   Constant

sY
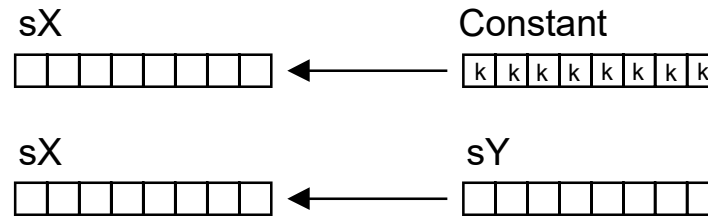
Since the LOAD instruction does not effect the flags it may be used to reorder and assign register contents at any stage of the program execution. The ability to assign a constant with no impact to the program size or performance means that the load instruction is the most obvious way to assign a value or clear a register.

The first operand of a LOAD instruction must specify the register to be loaded as register 's' followed by a hexadecimal digit. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

LOAD sX,kk

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	x	x	x	x	k	k	k	k	k	k	k	k

sX   Constant

LOAD sX,sY

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	x	x	x	x	y	y	y	y	0	0	0	0

sX   sY

XILINX
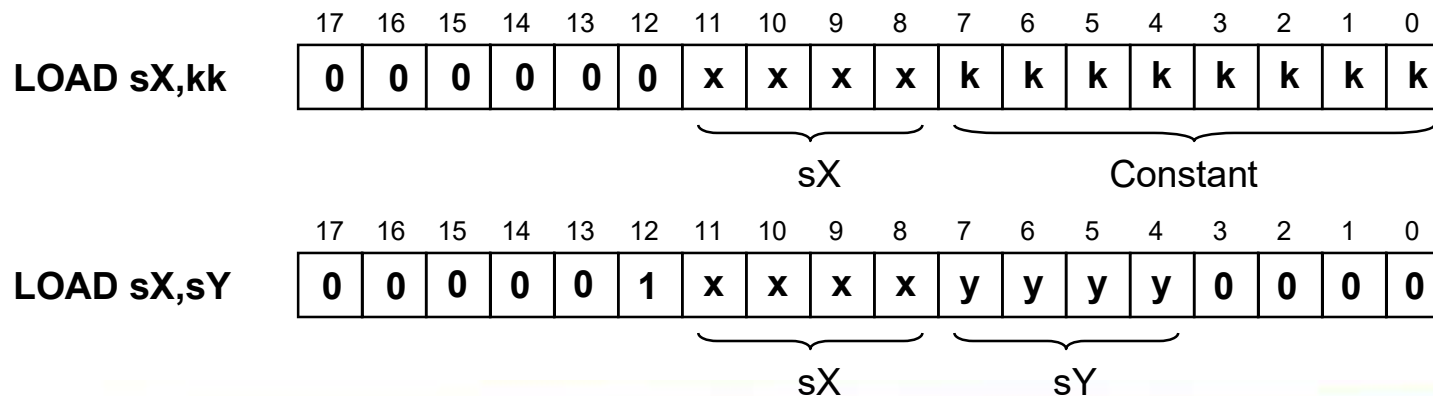
AND

The AND instruction performs a bit-wise logical 'AND' operation between two operands. For example 00001111 AND 00110011 will produce the result 00000011. The first operand is any register, and it is this register which will be assigned the result of the operation. A second operand may also be any register or an 8-bit constant value. Flags will be effected by this operation. The AND operation is useful for resetting bits of a register and performing tests on the contents (see also TEST instruction). The status of the ZERO flag will then control the flow of the program.

sX   sX   Constant

◄───────── AND k k k k k k k k

sX   sX   sY

◄───────── AND

CARRY **0**   ZERO **?**   Set if all bits of result are zero.
Reset in all other cases.
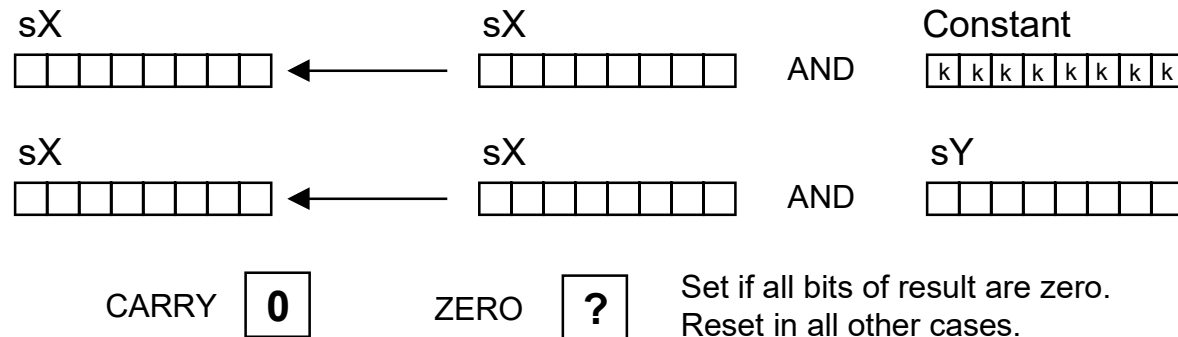
Each AND instruction must specify the first operand register as 's' followed by a hexadecimal digit. This register will also form the destination for the result. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AND sX,kk	0	0	1	0	1	0	x	x	x	x	k	k	k	k	k	k	k	k

sX   Constant

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AND sX,sY	0	0	1	0	1	1	x	x	x	x	y	y	y	y	0	0	0	0

sX   sY

OR

The OR instruction performs a bit-wise logical 'OR' operation between two operands. For example 00001111 OR 00110011 will produce the result 00111111. The first operand is any register, and it is this register which will be assigned the result of the operation. A second operand may also be any register or an 8-bit constant value. Flags will be effected by this operation. OR provides a way to force any bits of the specified register to be set which can be useful in forming control signals.

sX   sX   Constant

$sX \leftarrow sX$ OR $k\ k\ k\ k\ k\ k\ k\ k$

sX   sX   sY

$sX \leftarrow sX$ OR sY

CARRY **0**   ZERO **?**   Set if all bits of result are zero.
Reset in all other cases.

Each OR instruction must specify the first operand register as 's' followed by a hexadecimal digit. This register will also form the destination for the result. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.
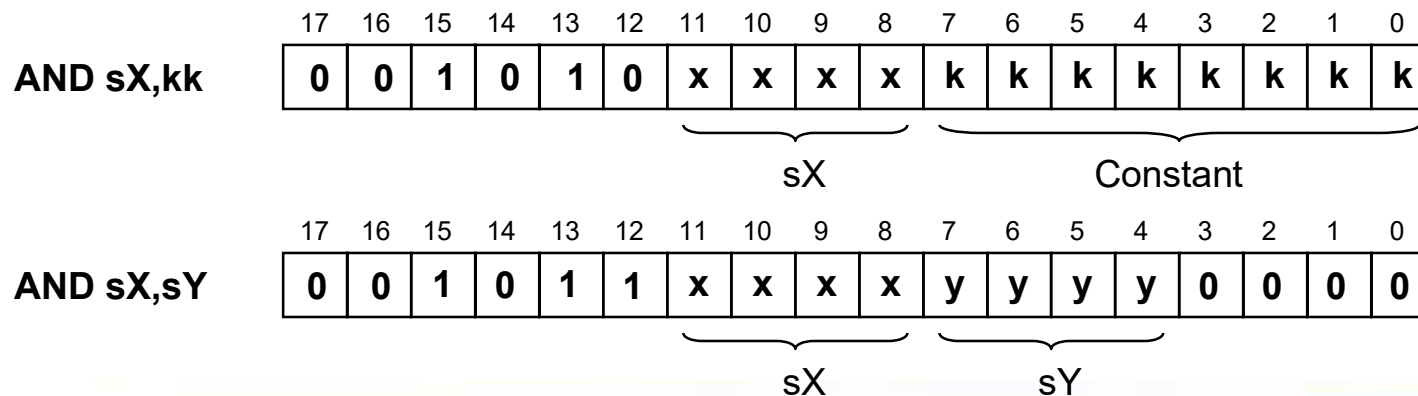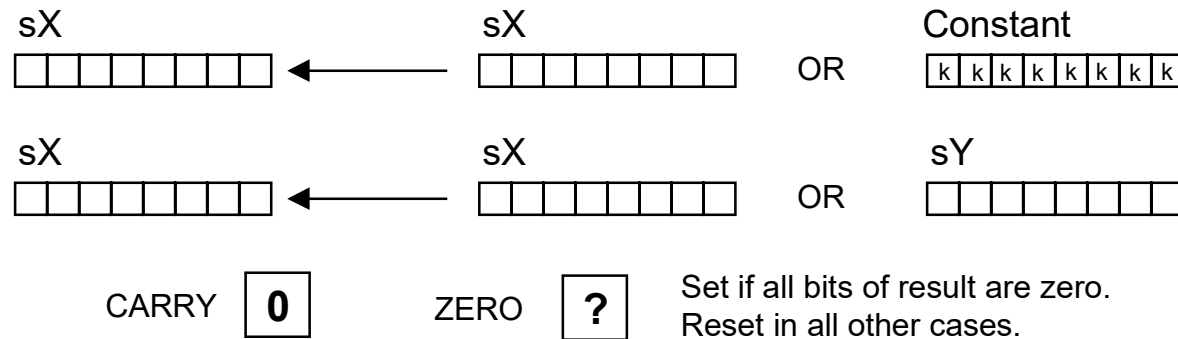
	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OR sX,kk	0	0	1	1	0	0	x	x	x	x	k	k	k	k	k	k	k	k

sX   Constant

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OR sX,sY	0	0	1	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0

sX   sY

XOR

The XOR instruction performs a bit-wise logical 'XOR' operation between two operands. For example 00001111 XOR 00110011 will produce the result 00111100. The first operand is any register, and it is this register which will be assigned the result of the operation. A second operand may also be any register or an 8-bit constant value. Flags will be effected by this operation. The XOR operation is useful for inverting bits contained in a register which is useful in forming control signals.

sX	sX Constant
sX	sX sY

CARRY **0**   ZERO **?**   Set if all bits of result are zero.
Reset in all other cases.

Each XOR instruction must specify the first operand register as 's' followed by a hexadecimal digit. This register will also form the destination for the result. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.
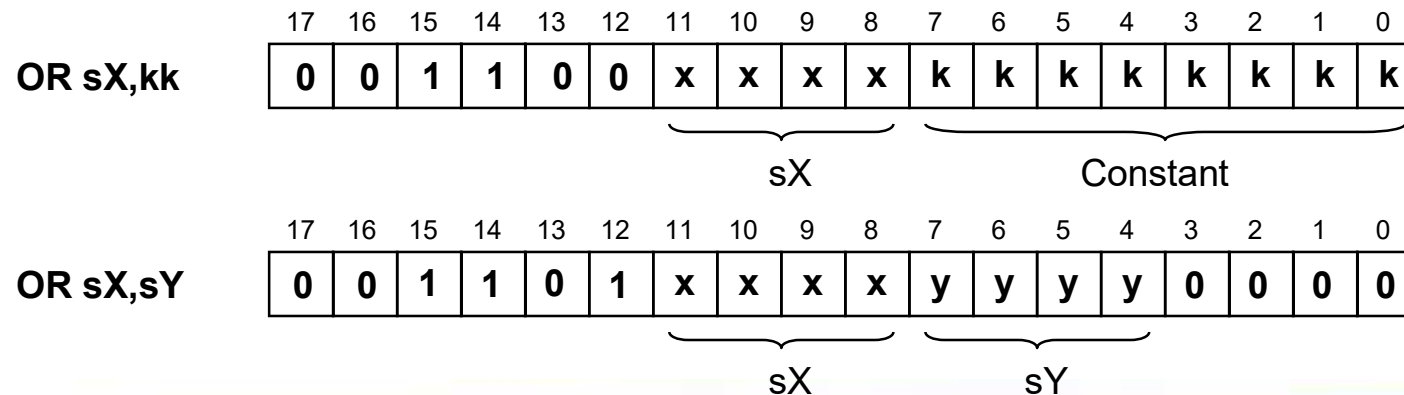
XOR sX,kk

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	x	x	x	x	k	k	k	k	k	k	k	k

sX   Constant

XOR sX,sY

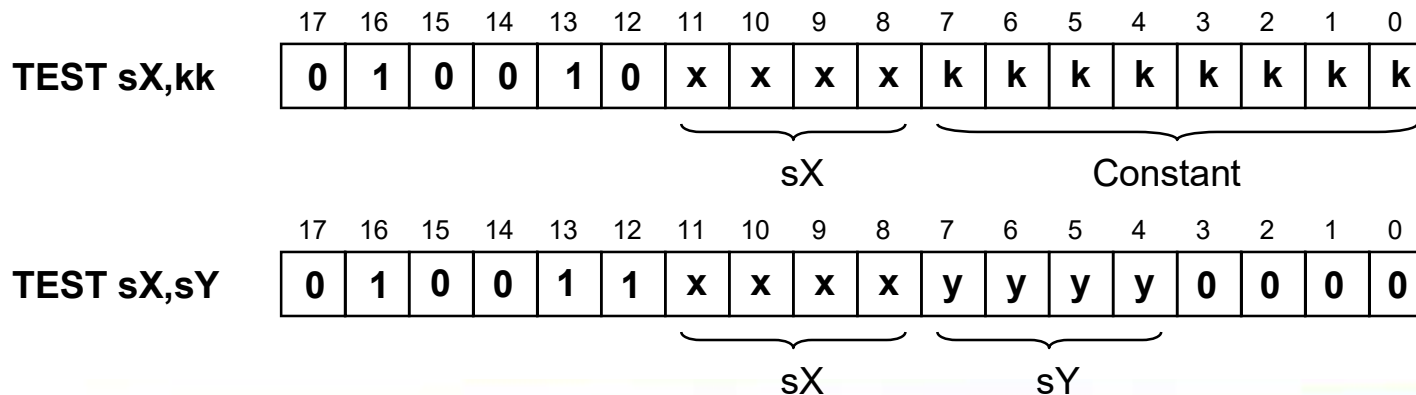| 17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0

sX   sY

TEST

The TEST instruction performs a bit-wise logical 'AND' operation between two operands. Unlike the 'AND' instruction, the result of the operation is discarded and only the flags are affected. The ZERO flag is set if all bits of the temporary result are low. The CARRY flag is used to indicate the **ODD PARITY** of the temporary result. Parity checks typically involve a test of all bits, i.e. if the contents of 's5' = 3D (00111101), the execution of TEST s5,FF will set the CARRY flag indicating ODD parity. Bit testing is typically used to isolate a single bit. For example TEST s5,04 will test bit2 of the 's5' register which would set the CARRY flag if the bit is high (reset if the bit is low) and set the ZERO flag if the bit is low (reset if the bit is high).

Temporary	sX	Constant
	←	AND k k k k k k k k
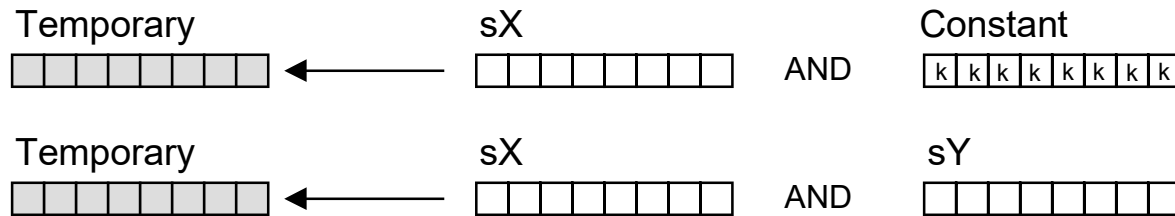
Temporary	sX	sY
	←	AND

CARRY **?** Set if there are an odd number of bits set to '1' in the temporary result.

ZERO **?** Set if all bits of temporary result are zero. Reset in all other cases.

Each TEST instruction must specify the first operand register as 's' followed by a hexadecimal digit. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

TEST sX,kk

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	x	x	x	x	k	k	k	k	k	k	k	k

sX = bits 11–8, Constant = bits 7–0

TEST sX,sY

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	x	x	x	x	y	y	y	y	0	0	0	0

sX = bits 11–8, sY = bits 7–4

XILINX

ADD

The ADD instruction performs an 8-bit addition of two operands. The first operand is any register, and it is this register which will be assigned the result of the operation. A second operand may also be any register or an 8-bit constant value. Flags will be effected by this operation. 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 and counters.
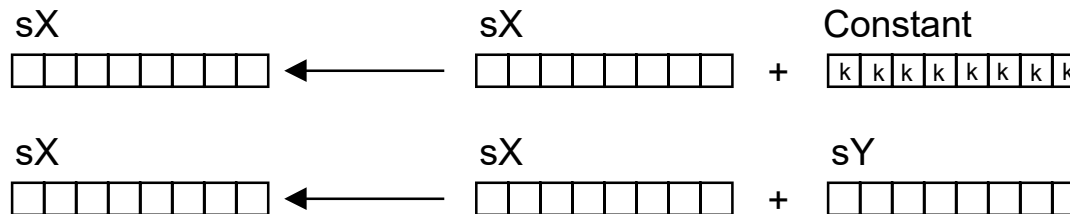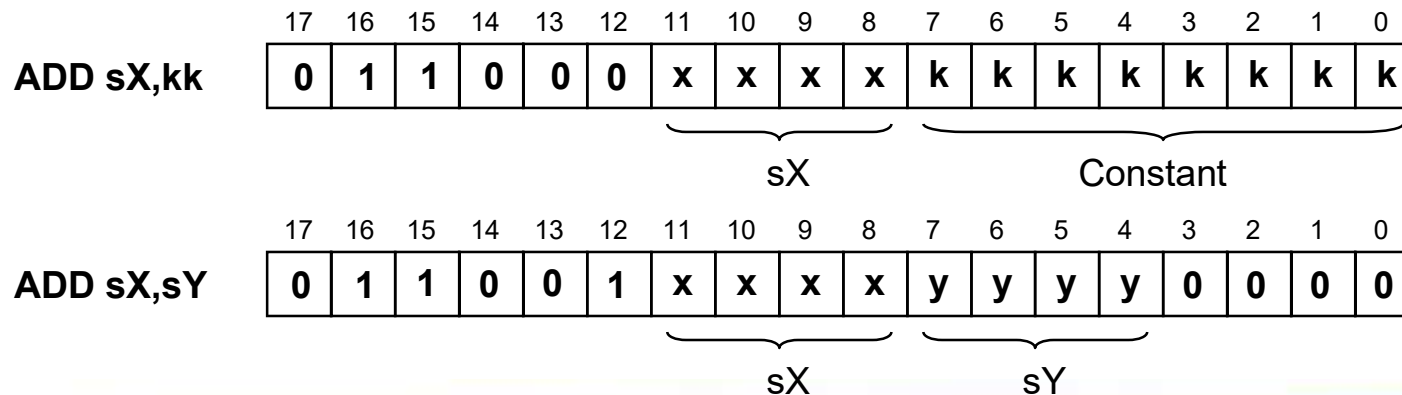
sX ← sX + Constant (k k k k k k k k)

sX ← sX + sY

CARRY [?] Set if result of addition exceeds FF. Reset in all other cases.
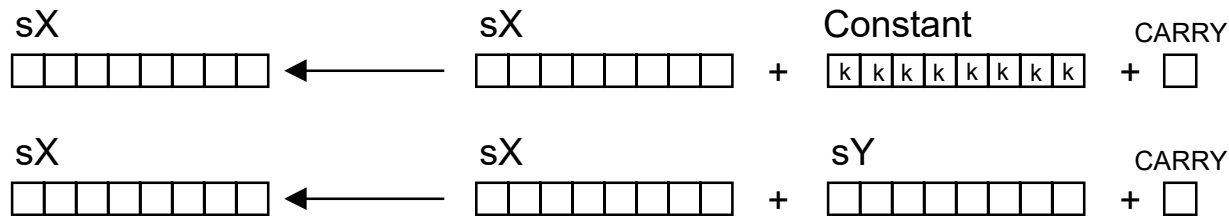
ZERO [?] Set if all bits of result are zero. Reset in all other cases.

Each ADD instruction must specify the first operand register as 's' followed by a hexadecimal digit. This register will also form the destination for the result. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD sX,kk	0	1	1	0	0	0	x	x	x	x	k	k	k	k	k	k	k	k

sX   Constant

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD sX,sY	0	1	1	0	0	1	x	x	x	x	y	y	y	y	0	0	0	0

sX   sY

XILINX

ADDCY

The ADDCY instruction performs an addition of two 8-bit operands together with the contents of the CARRY flag. The first operand is any register, and it is this register which will be assigned the result of the operation. A second operand may also be any register or an 8-bit constant value. Flags will be effected by this operation. The ADDCY operation can be used in the formation of adder and counter processes exceeding 8 bits.
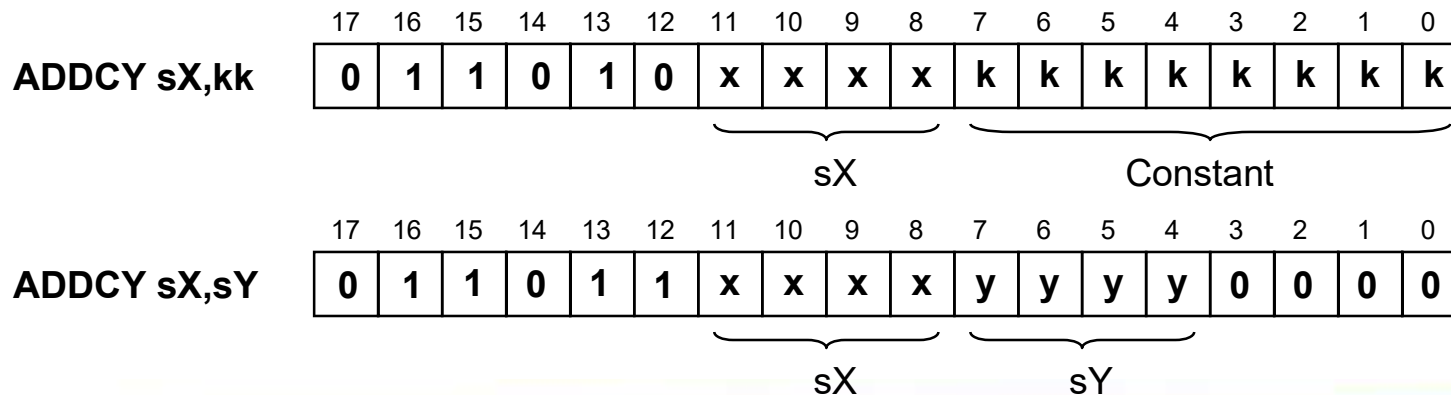
sX	sX	Constant	CARRY
⬜⬜⬜⬜⬜⬜⬜⬜ ←	⬜⬜⬜⬜⬜⬜⬜⬜ +	k k k k k k k k +	⬜

sX	sX	sY	CARRY
⬜⬜⬜⬜⬜⬜⬜⬜ ←	⬜⬜⬜⬜⬜⬜⬜⬜ +	⬜⬜⬜⬜⬜⬜⬜⬜ +	⬜

CARRY ? Set if result of addition exceeds FF.
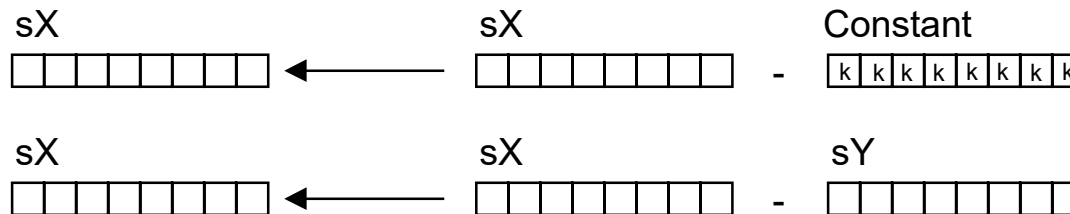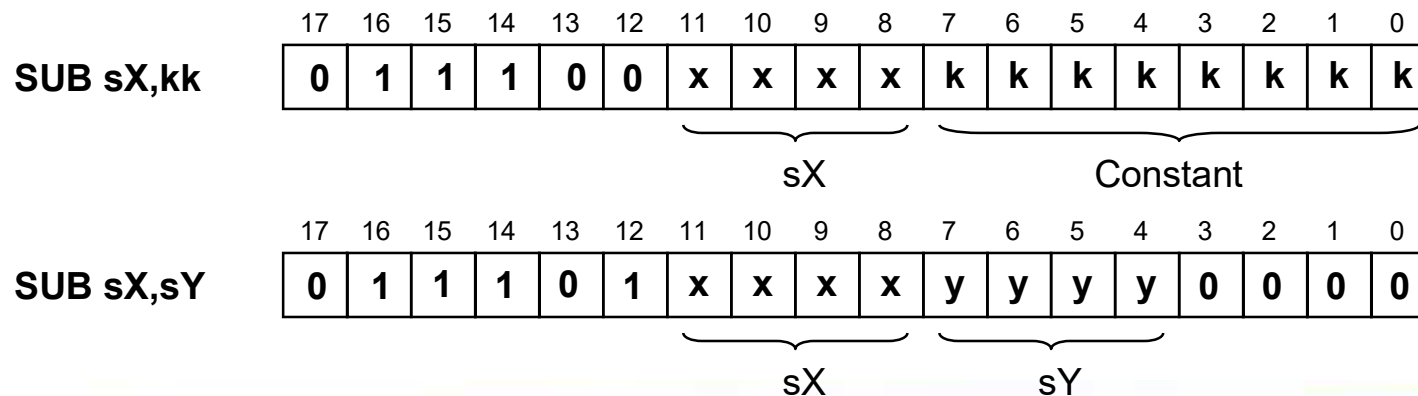Reset in all other cases.

ZERO ? Set if all bits of result are zero.
Reset in all other cases.

Each ADDCY instruction must specify the first operand register as 's' followed by a hexadecimal digit. This register will also form the destination for the result. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

ADDCY sX,kk

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	x	x	x	x	k	k	k	k	k	k	k	k

sX (bits 11–8)   Constant (bits 7–0)

ADDCY sX,sY

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	x	x	x	x	y	y	y	y	0	0	0	0

sX (bits 11–8)   sY (bits 7–4)

XILINX

SUB

The SUB instruction performs an 8-bit subtraction of two operands. The first operand is any register, and it is this register which will be assigned the result of the operation. The second operand may also be any register or an 8-bit constant value. Flags will be effected by this operation. 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 indicates when an underflow has occurred. For example, if 's05' contains 27 hex and the instruction SUB s05,35 is performed, then the stored result will be F2 hex and the CARRY flag will be set.

sX ← sX - Constant (k k k k k k k k)

sX ← sX - sY

CARRY [?] Set if result is negative.
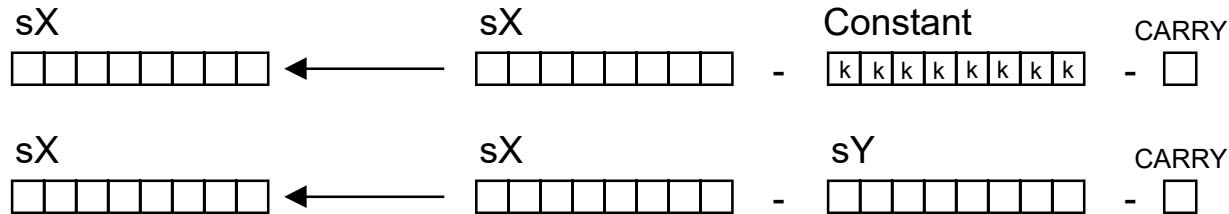Reset in all other cases.

ZERO [?] Set if all bits of result are zero.
Reset in all other cases.

Each SUB instruction must specify the first operand register as 's' followed by a hexadecimal digit. This register will also form the destination for the result. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SUB sX,kk	0	1	1	1	0	0	x	x	x	x	k	k	k	k	k	k	k	k

sX (bits 11-8) Constant (bits 7-0)

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SUB sX,sY	0	1	1	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0

sX (bits 11-8) sY (bits 7-4)

SUBCY

The SUBCY instruction performs an 8-bit subtraction of two operands together with the contents of the CARRY flag. The first operand is any register, and it is this register which will be assigned the result of the operation. The second operand may also be any register or an 8-bit constant value. Flags will be effected by this operation. The SUBCY operation can be used in the formation of subtract and down counter processes exceeding 8 bits.
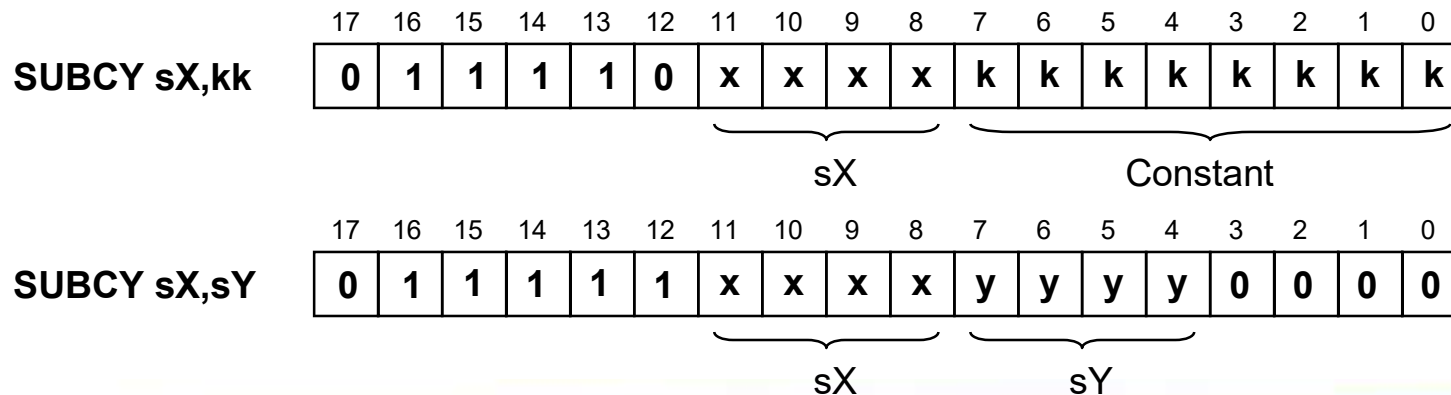
sX	sX	Constant — CARRY
⬅	−	k k k k k k k k − ☐
sX	sX	sY — CARRY
⬅	−	− ☐

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.

Each SUBCY instruction must specify the first operand register as 's' followed by a hexadecimal digit. This register will also form the destination for the result. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SUBCY sX,kk	0	1	1	1	1	0	x	x	x	x	k	k	k	k	k	k	k	k

sX   Constant

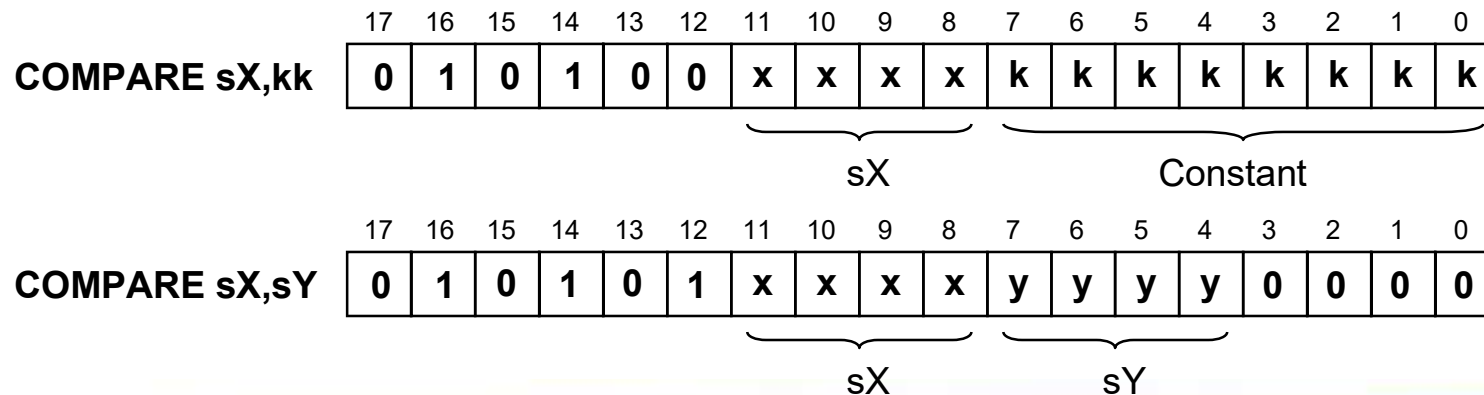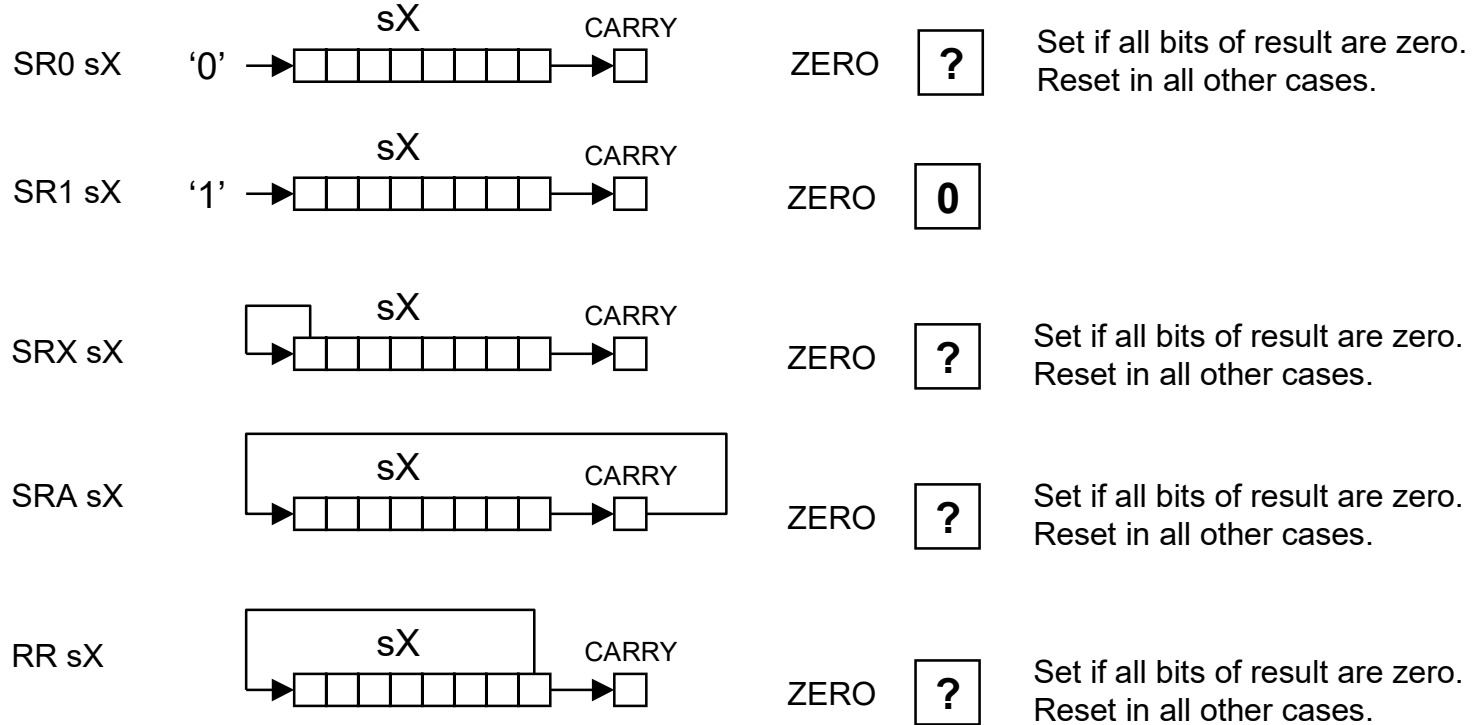|	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SUBCY sX,sY	0	1	1	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0

sX   sY

COMPARE

The COMPARE instruction performs an 8-bit subtraction of two operands Unlike the 'SUB' instruction, the result of the operation is discarded and only the flags are affected. The ZERO flag is set when all the bits of the temporary result are low and indicates that both input operands were identical. The CARRY flag indicates when an underflow has occurred and indicates that the second operand was larger than the first. For example, if 's05' contains 27 hex and the instruction COMPARE s05,35 is performed, then the CARRY flag will be set (35>27) and the ZERO flag will be reset (35≠27).

Temporary   ←   sX   -   Constant (k k k k k k k k)

Temporary   ←   sX   -   sY
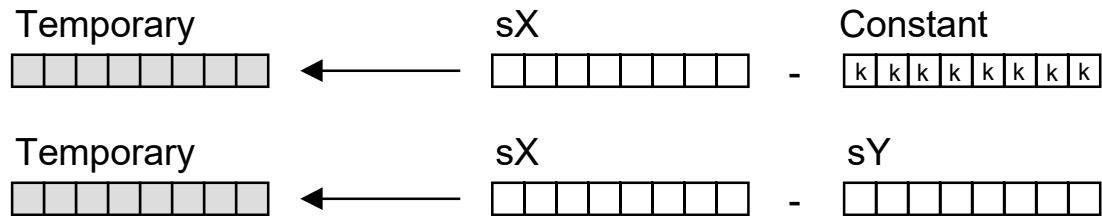
CARRY [?]   Set if 'sY' or 'kk' is **greater than** 'sX'. Reset in all other cases.

ZERO [?]   Set if operands are **equal**. Reset in all other cases.

Each COMPARE instruction must specify the first operand register as 's' followed by a hexadecimal digit. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COMPARE sX,kk	0	1	0	1	0	0	x	x	x	x	k	k	k	k	k	k	k	k

                                  sX   Constant

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COMPARE sX,sY	0	1	0	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0

                                    sX   sY

SR0, SR1, SRX, SRA, RR

The shift and rotate right group all modify the contents of a single register. All instructions in the group have an effect on the flags.
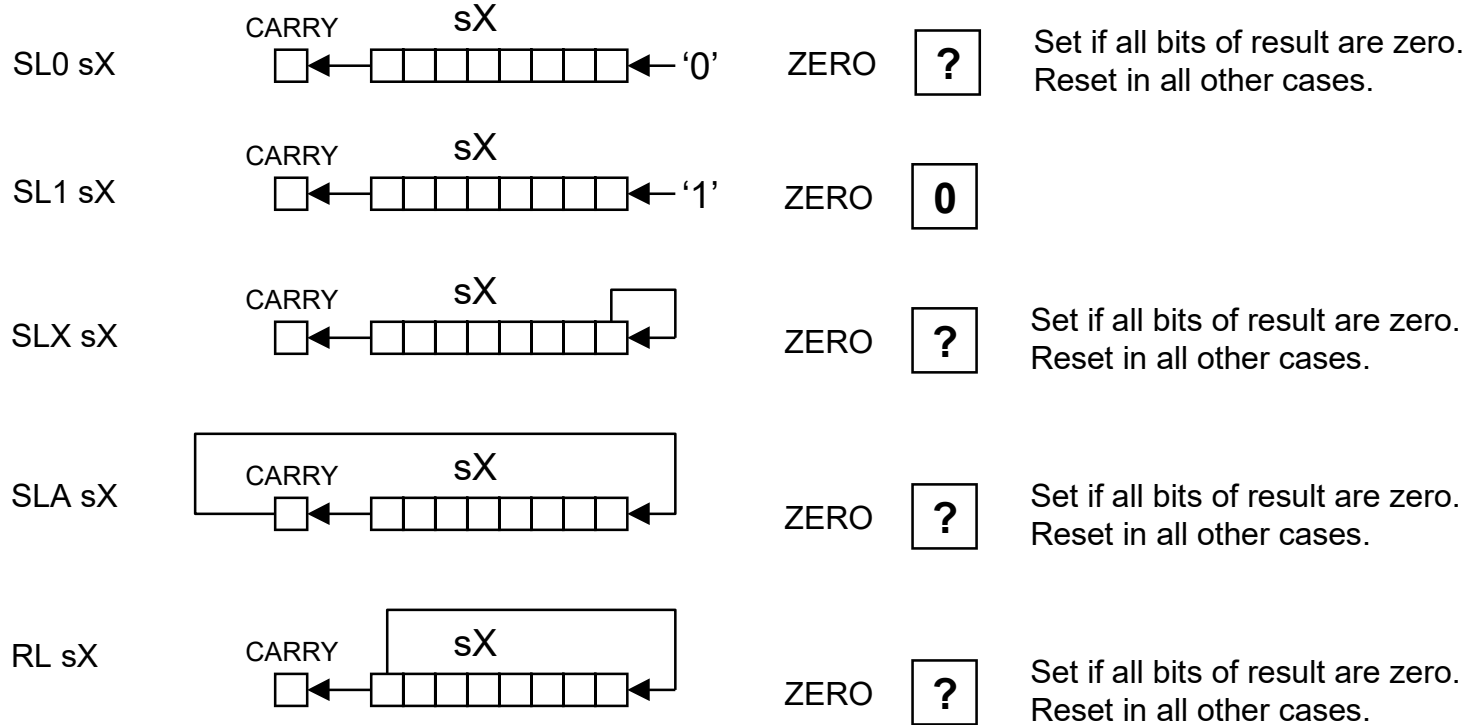
SR0 sX	'0' → [sX] → CARRY	ZERO	**?**	Set if all bits of result are zero. Reset in all other cases.
SR1 sX	'1' → [sX] → CARRY	ZERO	**0**	
SRX sX	[sX] → CARRY	ZERO	**?**	Set if all bits of result are zero. Reset in all other cases.
SRA sX	[sX] → CARRY	ZERO	**?**	Set if all bits of result are zero. Reset in all other cases.
RR sX	[sX] → CARRY	ZERO	**?**	Set if all bits of result are zero. Reset in all other cases.

Each instruction must specify the register as 's' followed by a hexadecimal digit. The assembler supports register naming to simplify the process.

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	x	x	x	x	0	0	0	0	1			

sX (bits 11–8)

Bit 2	Bit 1	Bit 0	Instruction
1	1	0	SR0 sX
1	1	1	SR1 sX
0	1	0	SRX sX
0	0	0	SRA sX
1	0	0	RR sX

XILINX

SL0, SL1, SLX, SLA, RL

The shift and rotate left group all modify the contents of a single register. 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 ← (bit 0) — ZERO [?] Set if all bits of result are zero. Reset in all other cases.

SLA sX — CARRY ← sX (with carry into bit 0) — ZERO [?] Set if all bits of result are zero. Reset in all other cases.

RL sX — CARRY ← sX (rotate) — ZERO [?] Set if all bits of result are zero. Reset in all other cases.

Each instruction must specify the register as 's' followed by a hexadecimal digit. The assembler supports register naming to simplify the process.

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	x	x	x	x	0	0	0	0	0			

sX = bits 11–8

Bit 2	Bit 1	Bit 0	Instruction
1	1	0	SL0 sX
1	1	1	SL1 sX
1	0	0	SLX sX
0	0	0	SLA sX
0	1	0	RL sX

OUTPUT

The OUTPUT instruction enables the contents of any register to be transferred to logic external to KCPSM3. The port address (in the range 00 to FF) can be defined by a constant value or indirectly as the contents of any other register. The Flags are not affected by this operation.

Port Value		sX		PORT_ID Address		Constant
☐☐☐☐☐☐☐☐	←	☐☐☐☐☐☐☐☐		☐☐☐☐☐☐☐☐	←	p p p p p p p p

Port Value		sX		PORT_ID Address		sY
☐☐☐☐☐☐☐☐	←	☐☐☐☐☐☐☐☐		☐☐☐☐☐☐☐☐	←	☐☐☐☐☐☐☐☐
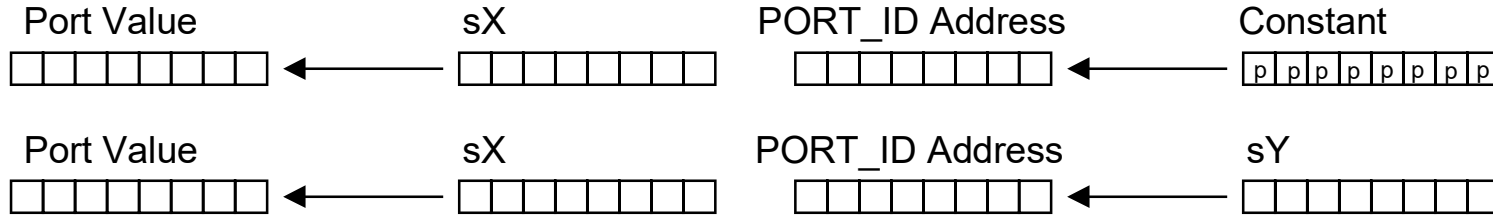
The user interface logic is required to decode the PORT_ID port address value and capture the data provided on the OUT_PORT. The WRITE_STROBE is set during an output operation (see 'READ and WRITE STROBES'), and should be used to clock enable the capture register or write enable a RAM (see 'Design of Output Ports').

Each OUTPUT instruction must specify the source register as 's' followed by a hexadecimal digit. It must then specify the output port address using a register value in a similar way or specify an 8-bit constant port identifier using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.
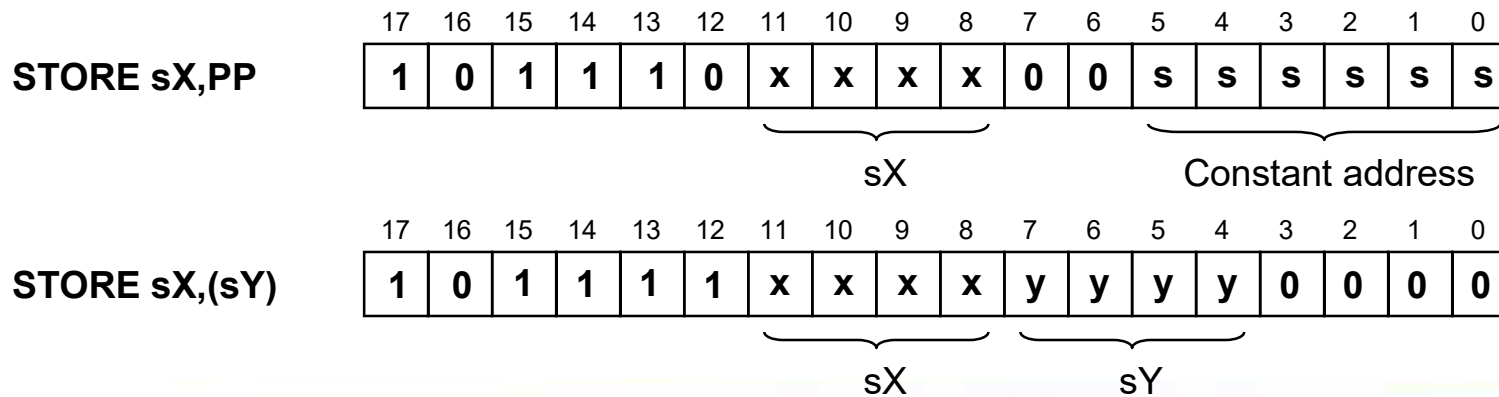
OUTPUT sX,PP

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	x	x	x	x	p	p	p	p	p	p	p	p

sX = bits 11–8, Constant PORT_ID = bits 7–0

OUTPUT sX,(sY)

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0

sX = bits 11–8, sY = bits 7–4
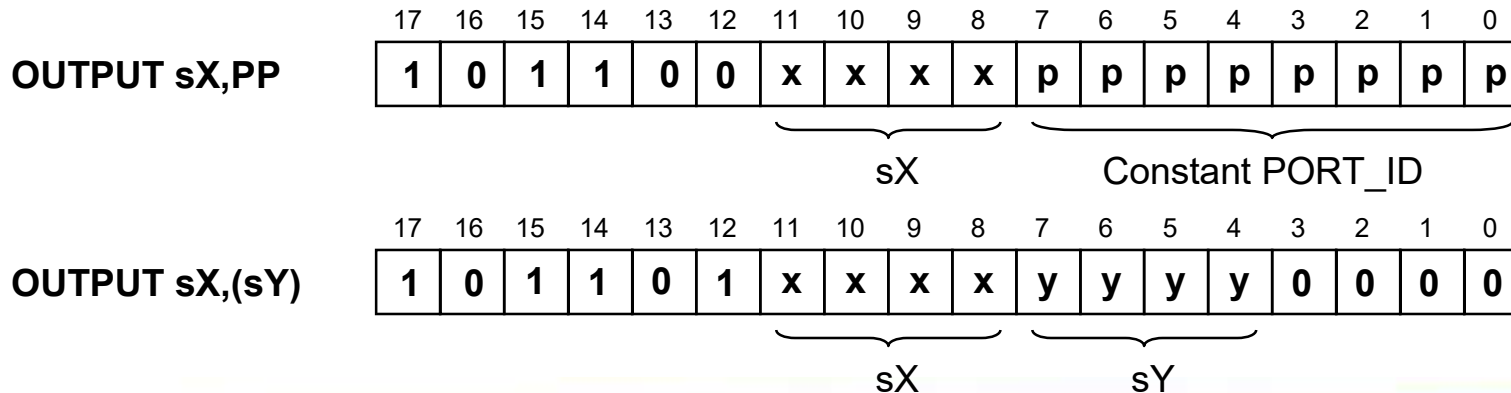
INPUT

The INPUT instruction enables data values external to KCPSM3 to be transferred into any one of the internal registers. The port address (in the range 00 to FF) can be defined by a constant value or indirectly as the contents of any other register. The Flags are not affected by this operation.
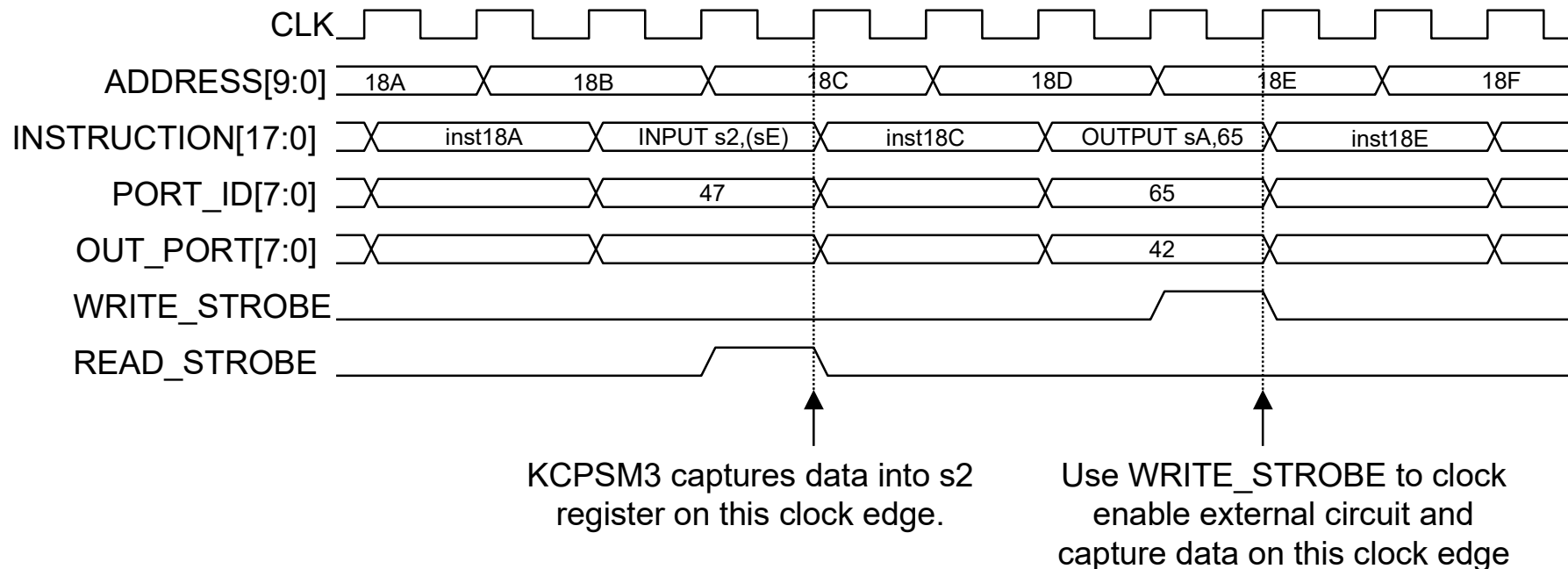
sX	Port Value	PORT_ID Address	Constant
☐☐☐☐☐☐☐☐ ←	☐☐☐☐☐☐☐☐	☐☐☐☐☐☐☐☐ ←	p p p p p p p p

sX	Port Value	PORT_ID Address	sY
☐☐☐☐☐☐☐☐ ←	☐☐☐☐☐☐☐☐	☐☐☐☐☐☐☐☐ ←	☐☐☐☐☐☐☐☐
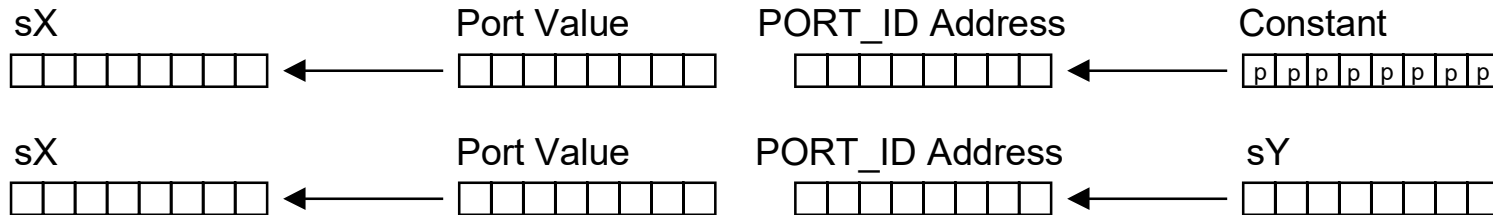
The user interface logic is required to decode the PORT_ID port address value and supply the correct data to the IN_PORT. The READ_STROBE is set during an input operation (see 'READ and WRITE STROBES'), but it is not always necessary for the interface logic to decode this strobe. However, it can be useful for determining when data has been read, such as when reading a FIFO buffer (see 'Design of Input Ports').

Each INPUT instruction must specify the destination register as 's' followed by a hexadecimal digit. It must then specify the input port address using a register value in a similar way or specify an 8-bit constant port identifier using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

INPUT sX,PP

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	x	x	x	x	p	p	p	p	p	p	p	p

sX (bits 11-8), Constant PORT_ID (bits 7-0)

INPUT sX,(sY)

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0

sX (bits 11-8), sY (bits 7-4)
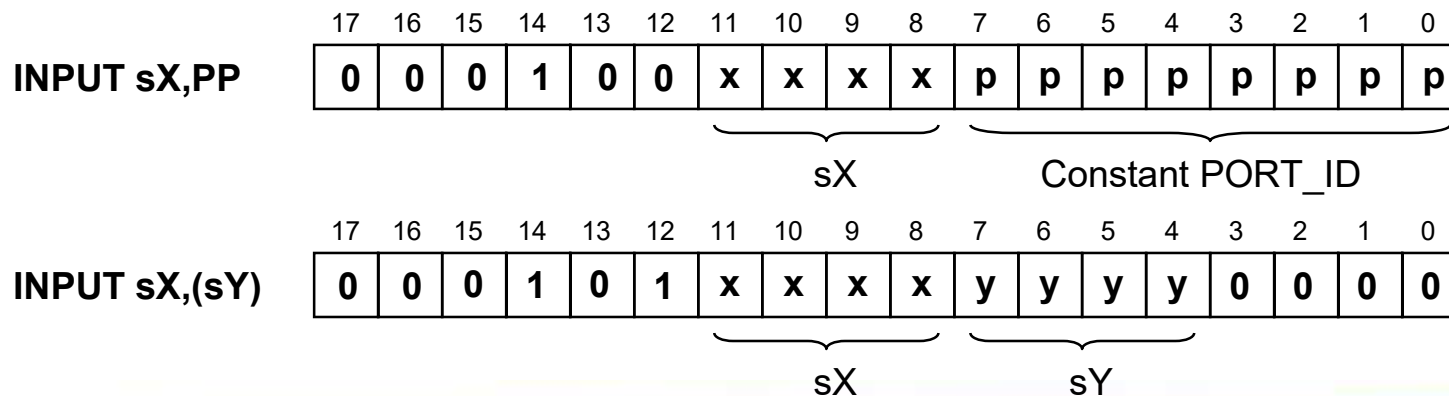
STORE

The STORE instruction enables the contents of any register to be transferred to the 64-byte internal scratch pad memory. The storage address (in the range 00 to 3F) can be defined by a constant value or indirectly as the contents of any other register. The Flags are not affected by this operation.

Scratch pad memory — sX — Constant: 0 0 s s s s s s

Scratch pad memory — sX — sY
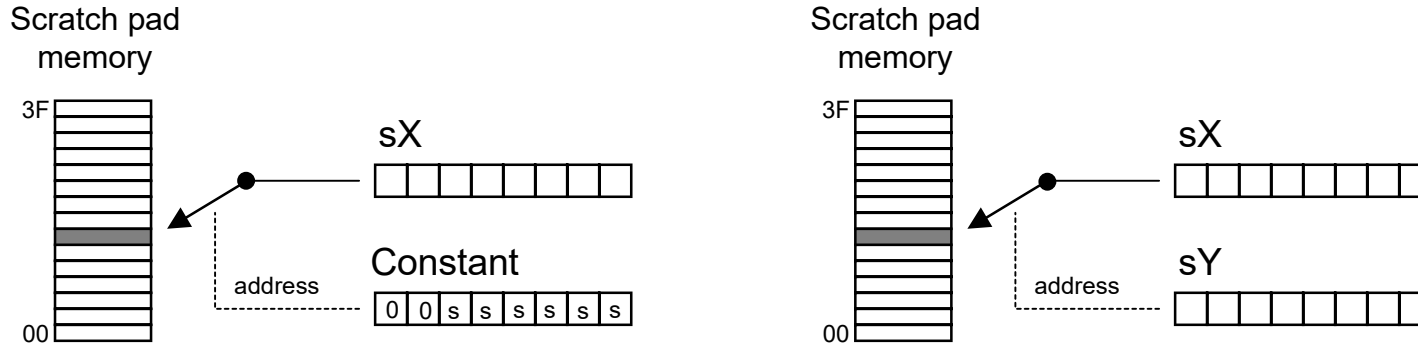
Each STORE instruction must specify the source register as 's' followed by a hexadecimal digit. It must then specify the storage address using a register value in a similar way or specify a 6-bit constant storage address using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process. Although the assembler will reject constants greater than 3F, it is the responsibility of the programmer to ensure that the value of 'sY' is within the address range.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
STORE sX,PP	1	0	1	1	1	0	x	x	x	x	0	0	s	s	s	s	s	s

sX (bits 11-8), Constant address (bits 5-0)

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
STORE sX,(sY)	1	0	1	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0

sX (bits 11-8), sY (bits 7-4)

FETCH

The FETCH instruction enables data held in the 64-byte internal scratch pad memory to be transferred any of the internal registers. The storage address (in the range 00 to 3F) can be defined by a constant value or indirectly as the contents of any other register. The Flags are not affected by this operation.

Scratch pad memory

Scratch pad memory
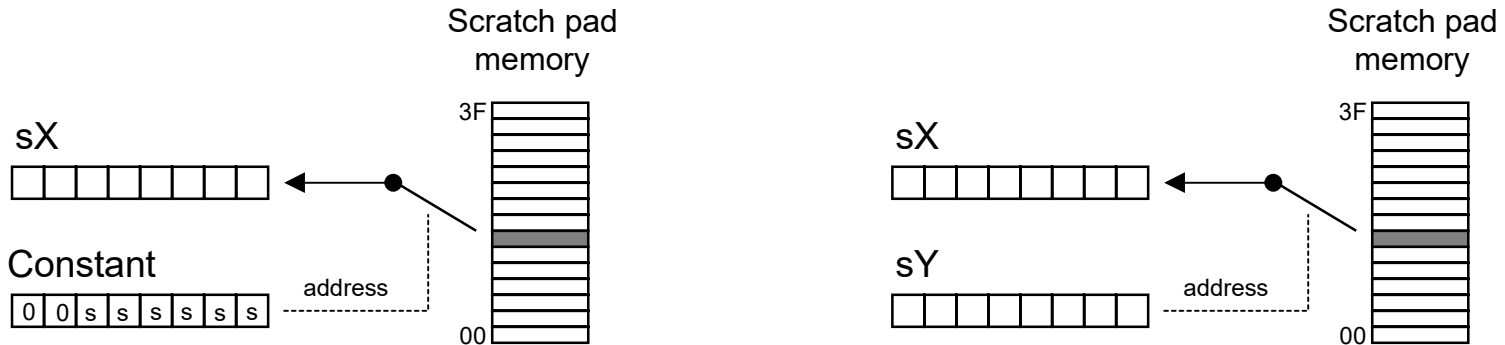
Each FETCH instruction must specify the destination register as 's' followed by a hexadecimal digit. It must then specify the storage address using a register value in a similar way or specify a 6-bit constant storage address using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process. Although the assembler will reject constants greater than 3F, it is the responsibility of the programmer to ensure that the value of 'sY' is within the address range.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FETCH sX,PP	0	0	0	1	1	0	x	x	x	x	0	0	s	s	s	s	s	s

sX   Constant address

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FETCH sX,(sY)	0	0	0	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0

sX   sY
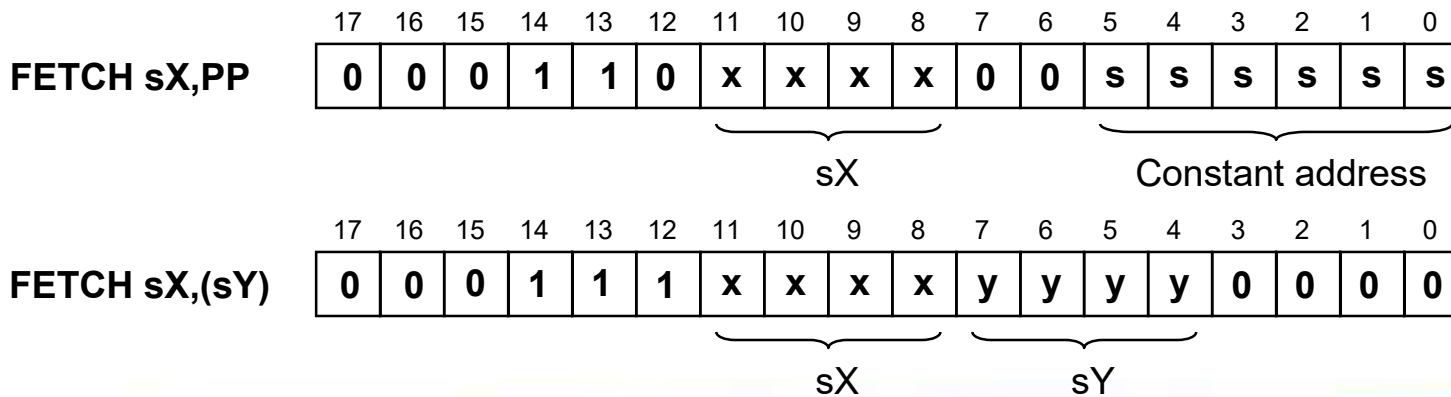
READ and WRITE STROBES

These pulses are used by external circuits to confirm input and output operations. In the waveforms below, it is assumed that the content of register sE is 47, and the content of register sA is 42.

KCPSM3 captures data into s2 register on this clock edge.

Use WRITE_STROBE to clock enable external circuit and capture data on this clock edge

PORT_ID[7:0] is valid for 2 clock cycles providing additional time for external decoding logic and enabling the connection of synchronous RAM. The WRITE_STROBE is provided on the second clock cycle to confirm an active write by KCPSM3. In most cases, the READ_STROBE will not be utilised by the external decoding logic, but again occurs in the second cycle and indicates the actual clock edge on which data is read into the specified register.

Note for timing critical designs, your timing specifications can allow 2 clock cycles for PORT_ID and data paths, and only the strobes need to be constrained to a single clock cycle. Ideally, a pipeline register can be inserted where possible (see 'Design of Input Ports', 'Design of Output Ports' and 'Connecting Memory').

RESET

KCPSM3 contains an internal reset control circuit to ensure the correct start up of KCPSM3 following device configuration or global reset (GSR). This reset can also be activated within your design.

The KCPSM3 reset is sampled synchronous to the clock and used to form a controlled internal reset signal which is distributed locally as required. A small 'filter' circuit (see right) ensures that the release of the internal reset is clean and controlled.
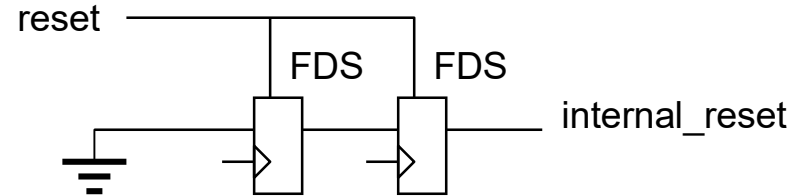
Release of Reset after configuration.
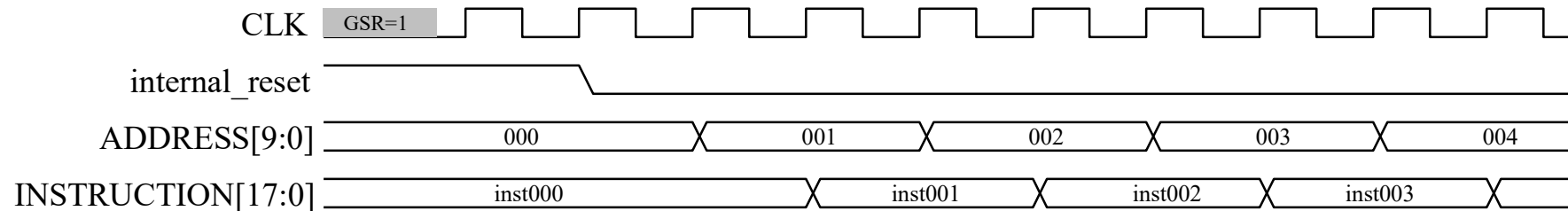
Application of user reset input   The reset input can be tied to logic '0' if not required and the 'filter' will still be used to ensure correct power-up sequence.
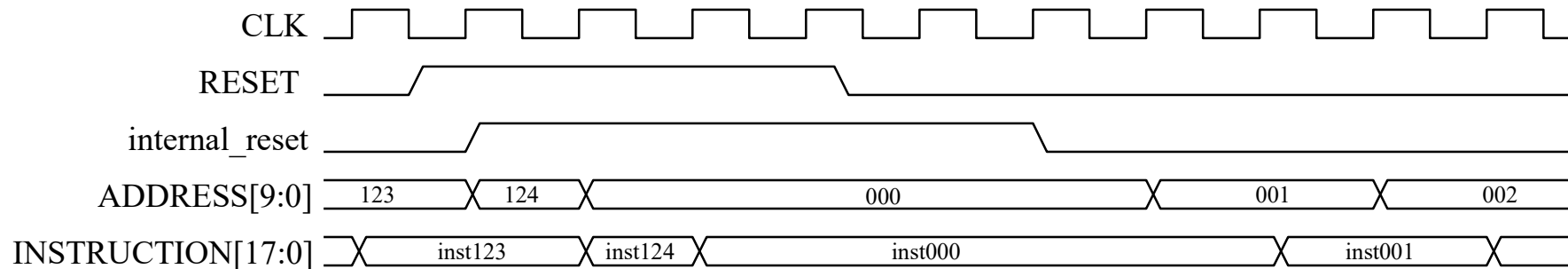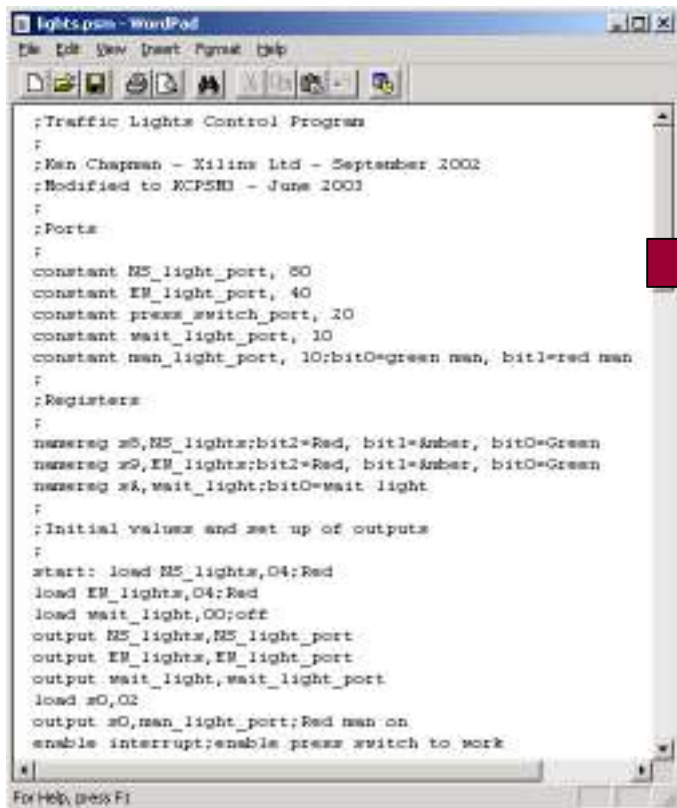
KCPSM3 Assembler

The KCPSM3 Assembler is provided as a simple DOS executable file together with three template files. Copy **all** the files KCPSM3.EXE, ROM_form.vhd, ROM_form.v and ROM_form.coe into your working directory.

Programs are best written with either the standard Notepad or Wordpad tools. The file is saved with a '.psm' file extension (8 character name limit).

Open a DOS box and navigate to the working directory. Then run the assembler 'kcpsm3 <filename>[.psm]' to assemble your program. It all happens very fast!!

<filename>.vhd <filename>.v <filename>.coe <filename>.m

Spartan-3/Virtex-II Block RAM program ROM definition files

Assembler Errors

The assembler will stop as soon as an error is detected. A short message will be displayed to help determine the reason for the error. The assembler will also display the line it was analyzing when it detected the problem. The user should fix each reported problem in turn and re-execute the assembler.

```
KCPSM3                                                          _ □ ×
03A JUMP NZ, inner_short;inner loop complete after 1004 clock cycles
03B SUB s1, 01;outer loop 250x1004 clock cycles
03C JUMP NZ, inner_short;inner loop complete after 1,000,000 clock cycles
03D SUB s2, 01
03E JUMP NZ, outer_short
03F RETURN
3E0 ADDRESS 3E0
3E0 ;
3E0 ;Long delay for sequencing of lights.
3E0 ;approx 30 seconds at 10MHz
3E0 ;
3E0 delay_30sec: LOAD s3, 1E
3E1 inner_long: CALL delay_1second

ERROR - Address is not 3-digits: delay_1second

        Provide a correct absolute address in range 000 to 3FF or
        a matching line label. Note that labels are case sensitive.

Please correct and try again.

KCPSM3 complete.


C:\DESIGN~1.1I\kcpsm3\ASSEMB~1>
```

Previous Progress

Line being processed ⟶

Error message ⟶

Since the execution of the assembler is very fast, it is unlikely that you will be able to 'see' it making progress and the display will appear to be immediate. If you would like to review everything that the assembler has written to the screen, the DOS output can be redirected to a text file using…….   **kcpsm3 <filename>[.psm] > screen_dump.txt**

XILINX

Assembler Files

The KCPSM3 assembler actually reads four input files and generates 15 output files. These are described in more detail on the following pages.

<filename>.psm } Program file

ROM_form.vhd

ROM_form. v → **KCPSM3.EXE**

ROM_form. coe

pass1.dat
pass2.dat
pass3.dat   Assembler intermediate
pass4.dat   processing files
pass5.dat   (may be useful for debugging)

ROM definition files for a variety of design flows
<filename>.vhd
<filename>.v
<filename>.coe
<filename>.m

ROM definition files for other utilities
<filename>.hex
<filename>.dec

<filename>.log
constant.txt   Assembler
labels.txt   report files

<filename>.fmt } Formatted version of user input file

Note - All output files are overwritten each time the assembler is executed.

The 'hex' and 'dec' files provide the program ROM contents in unformatted hexadecimal and decimal which is useful for conversion to other formats not supported directly by the assembler. There is no further description in this manual.

XILINX

ROM_form.vhd File

This file provides the template for the VHDL file generated by the assembler and suitable for synthesis and simulation. This file is provided with the assembler and must be placed in the working directory.

The supplied ROM_form.vhd template file defines a Single Port Block RAM for Spartan-3, Virtex-II or Virtex-IIPRO configured as a ROM. You can adjust this template to define the type of memory you want. The template supplied includes some additional notes on how the template works

ROM_form.vhd

```
entity {name} is
    Port (      address : in std_logic_vector(9 downto 0);
            instruction : out std_logic_vector(17 downto 0);
                    clk : in std_logic);
    end {name};
--
architecture low_level_definition of {name} is
.
.
.
attribute INIT_00 of ram_1024_x_18 : label is  "{INIT_00}";
attribute INIT_01 of ram_1024_x_18 : label is  "{INIT_01}";
attribute INIT_02 of ram_1024_x_18 : label is  "{INIT_02}";
```

The assembler reads the ROM_form.vhd template and simply copies the information into the output file <filename>.vhd. There is no checking of syntax, so any alterations are the responsibility of the user.

The template contains some special text strings contained in {} brackets. These are {`begin template`}, **{name}**, and a whole family of initialisation identifiers such as **{INIT_01}**. The assembler uses {begin template} to identify where the VHDL definition begins. It then intercepts and replaces all other special strings with the appropriate information. {name} is replaced with the name of the input program '.psm' file.

XILINX

ROM_form.v File

This file provides the template for the Verilog file generated by the assembler and suitable for synthesis and simulation. This file is provided with the assembler and must be placed in the working directory.

The supplied ROM_form.v template file defines a Single Port Block RAM for Spartan-3, Virtex-II or Virtex-IIPRO configured as a ROM. You can adjust this template to define the type of memory you want. The template supplied includes some additional notes on how the template works

ROM_form.v

```
module {name} (address, instruction, clk);

input [9:0] address;
input clk;

output [17:0] instruction;
.
.
.
defparam ram_1024_x_18.INIT_00  = 256'h{INIT_00};
defparam ram_1024_x_18.INIT_01  = 256'h{INIT_01};
defparam ram_1024_x_18.INIT_02  = 256'h{INIT_02};
```

The assembler reads the ROM_form.v template and simply copies the information into the output file <filename>.v. There is no checking of syntax, so any alterations are the responsibility of the user.

The template contains some special text strings contained in {} brackets. These are {`begin template`}, **{name}**, and a whole family of initialisation identifiers such as **{INIT_01}**. The assembler uses {begin template} to identify where the Verilog definition begins. It then intercepts and replaces all other special strings with the appropriate information. {name} is replaced with the name of the input program '.psm' file.

ROM_form.coe File

This file provides the template for the coefficient file generated by the assembler and suitable for the Core Generator. This file is provided with the assembler and must be placed in the working directory.

The supplied ROM_form.coe template file defines a Dual Port Block RAM for Spartan-3, Virtex-II or Virtex-IIPRO in which the A-port is read only and the B-port is read/write. You can adjust this template to define the type of memory you want Core Generator to implement.

ROM_form.coe

```
component_name={name};
width_a=18;
depth_a=1024;
.
.
memory_initialization_radix=16;
global_init_value=00000;
memory_initialization_vector=
```

KCPSM3 Assembler

<filename>.coe

```
component_name=simple;
width_a=18;
depth_a=1024;
.
.
memory_initialization_radix=16;
global_init_value=00000;
memory_initialization_vector=
01400, 23412, 09401, 100A0, 0C018, 35401, 34000, 00000, ...
```

The assembler reads the ROM_form.coe template and simply copies the information into the output file <filename>.coe. There is no checking of syntax, so any alterations are the responsibility of the user.

The template may contain the special text string **{name}** which the assembler will intercept and replace with the name of the program file. In this example you can see that {name} has been replaced with 'simple'.

It is vital that the last line of the template contains the key words...
memory_initialization_vector=
These are used by the Core Generator to identify that the data values follow, and the assembler will append the 1024 values required. Indeed, the template could simply contain this one line provided the Core Generator GUI is used to define all other parameters.

XILINX

`<filename>.fmt` File

When a program passes through the assembler additional files to the '.vhd' and '.coe' files are produced to be of assistance to the programmer. One of these is called '<filename>.fmt'. This file is the original program but formatted to look nice. Looking at this file is also an easy way to see that everything has been interpreted the way you had expected.

`<filename>.psm`

```
constant max_count, 18;count to 24 hours
namereg s4,counter_reg;define register for counter
constant count_port, 12
start: load counter_reg,00;initialise counter
loop:output counter_reg,count_port
add counter_reg,01;increment
load s0,counter_reg
sub s0,max_count;test for max value
jump nz,loop;next count
jump start;reset counter
```

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

Write your PSM program quickly and then use KCPSM3 to make a nice formatted version for you to adopt as your own.

KCPSM3 Assembler

`<filename>.fmt`

```
        CONSTANT max_count, 18          ;count to 24 hours
        NAMEREG s4, counter_reg         ;define register for counter
        CONSTANT count_port, 12
 start: LOAD counter_reg, 00            ;initialise counter
  loop: OUTPUT counter_reg, count_port
        ADD counter_reg, 01             ;increment
        LOAD s0, counter_reg
        SUB s0, max_count               ;test for max value
        JUMP NZ, loop                   ;next count
        JUMP start                      ;reset counter
```

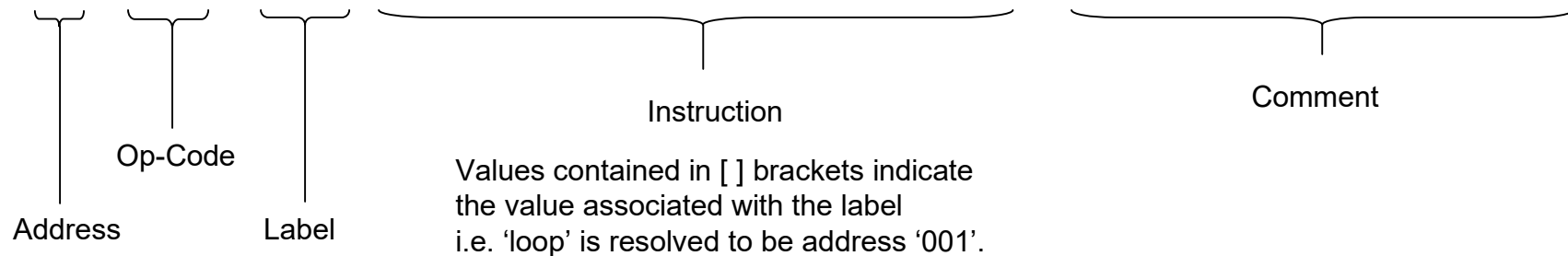**XILINX**

<filename>.log File

The '.log' file provides you with the most detail about the assembly process which has been performed. This is where you can observe how each instruction and directive has been used. Address and op-code values are associated with each line of the program and the actual values of addresses, registers, and constants defined by labels are specified.

<filename>.log

```
KCPSM3 Assembler log file for program 'simple.psm'.
Generated by KCPSM3 version 1.01
Ken Chapman (Xilinx Ltd) 2003.

 Addr Code

 000                   CONSTANT max_count, 18                   ;count to 24 hours
 000                   NAMEREG s4, counter_reg                  ;define register for counter
 000                   CONSTANT count_port, 12
 000  00400  start: LOAD counter_reg[s4], 00                    ;initialise counter
 001  2C412   loop: OUTPUT counter_reg[s4], count_port[12]
 002  18401         ADD counter_reg[s4], 01                     ;increment
 003  01040         LOAD s0, counter_reg[s4]
 004  18018         ADD s0, max_count[18]                       ;test for max value
 005  35401         JUMP NZ, loop[001]                          ;next count
 006  34000         JUMP start[000]                             ;reset counter
```

Address

Op-Code

Label

Instruction

Values contained in [ ] brackets indicate the value associated with the label i.e. 'loop' is resolved to be address '001'.

Comment

constant.txt & labels.txt Files

These two files provide a list of the line labels and their associated addresses, and a list of constants and their values as defined by 'constant' directives in the program file. These can be useful during the development and testing of larger programs.
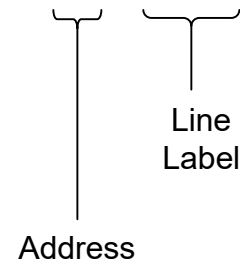
constant.txt

```
Table of constant values and their specified constant labels.

18   max_count
12   count_port
```

Constant
Label

Value
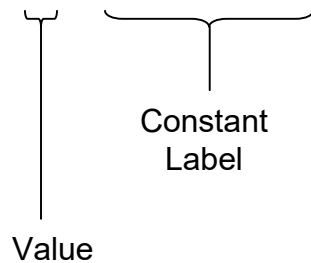
labels.txt

```
Table of addresses and their specified labels.

000   start
001   loop
```

Line
Label

Address

pass.dat Files

These are really internal files to the assembler and represent intermediate stages of the assembly process. These files will typically be ignored, but may just help in identifying how the assembler has interpreted the program file syntax. The files are automatically deleted at the start of the assembly process. If there is an error detected by the assembler, the '.dat' files will only be complete until the point of the last successful processing.

Part of **pass1.dat**

```
        LABEL-
  INSTRUCTION-add
     OPERAND1-counter_reg
     OPERAND2-01
      COMMENT-;increment
```

Part of **pass5.dat**

```
     ADDRESS-002
        LABEL-
    FORMATTED-ADD counter_reg, 01
    LOGFORMAT-ADD counter_reg[s4], 01
INSTRUCTION-ADD
     OPERAND1-counter_reg
    OP1 VALUE-s4
     OPERAND2-01
    OP2 VALUE-01
      COMMENT-;increment
```

The '.dat. Files segment the information from each line into the different fields. Each pass resolves more information.

The example shown here is related to the line……..

```
ADD counter_reg, 01 ;increment
```

It can be seen that pass1 has purely segmented the fields of the line. In the final pass5, you can see that the assembler has resolved all the relevant information.

Program Syntax

Probably the best way to understand what is and is not valid syntax is to look at the examples and try the assembler. However there are some simple rules which are of assistance from the beginning.

No blank lines - A blank line will be ignored by the assembler and removed from any formatted files. If you would like to keep a line use a blank comment (a semicolon).

Comments - Any item on a line following a semi-colon (;) will be ignored by the assembler. Whilst comments are useful, it is helpful if they are kept concise otherwise you will have very long lines and find it difficult to print out programs and log files.

Registers - All registers should be defined as the letter 's' immediately followed by one hexadecimal digit the range 0 to F. The assembler will accept any mixture of upper and lower case characters and automatically convert them to the 'sX' format where 'X' is one of 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. The NAMEREG directive can be used to assign new register names.

Constants - A constant must be specified using hexadecimal. Data values and Port Addresses in range 00 to FF. Memory store values in the range 00 to 3F and program addresses in the range 000 to 3FF. The assembler will accept any mixture of upper and lower case characters and automatically convert them to upper case.

Labels - Labels are any text string which the user defines. Labels are case sensitive for additional flexibility. Labels must not contain any spaces although the under-score character is supported. Valid characters are '0' to '9', 'a' to 'z', and 'A' to 'Z'. Again it is helpful for labels to be reasonably concise if only for the formatting of a program to be reasonable. Labels which could be confused with hexadecimal values or register specifications are rejected by the assembler.

Line Labels - A label is used to identify a program line for reference in a JUMP or CALL instruction and should be followed by a colon (:). The following example shows the use of a label to identify a program line and its use later in a JUMP instruction.

```
loop: OUTPUT counter_reg, count_port
      ADD counter_reg, 01              ;increment
      LOAD s0, counter_reg
      SUB s0, max_count                ;test for max value
      JUMP NZ, loop                    ;next count
```

Program Syntax

Instructions - The instructions should be of the format described in the "KCPSM3 instruction set" page of this document. 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 will accept any mixture of upper and lower case characters for the instruction and automatically convert them to upper case. The following examples all show acceptable instruction specifications, but the formatted output shows how it was expected.

```
load s5,7E                                    LOAD s5, 7E

AddCY s8,SE                                   ADDCY s8, sE

ENABLE interrupt                              ENABLE INTERRUPT

Output S2,   (S8)          Assembler          OUTPUT s2, (s8)

jump  Nz,  2a7                                JUMP NZ, 2A7

ADD sF, step_value                            ADD sF, step_value

INPUT S9,28                                   INPUT s9, 28

sl1 se                                        SL1 sE

store S8,(Sf)                                 STORE s8, (sF)
```

Most other syntax issues will be solved by reading the error messages provided by the assembler.

CONSTANT Directive

The assembler supports three assembler directives. These are commands included in the program which are used purely by the assembly process and do not correspond to instructions executed by KCPSM3.

The CONSTANT directive provides a way to assign an 2-digit hexadecimal value to a label. In this way the program can declare constants such as port and storage addresses and particular data values needed in the program. By defining constant values in this way it is often easier to understand their meaning in the program rather than using absolute values in the program lines. The following example illustrates the directive syntax and its uses.

```
                CONSTANT light_port, 03      ;light sensor port
                CONSTANT light_sensor, 01    ;bit0 is light sensor
                CONSTANT temp_sensor, 40     ;temperature sensor port
                NAMEREG sF, light_count_msb  ;16-bit light pulse counter
                NAMEREG sE, light_count_lsb
                NAMEREG sD, new_temp         ;current temperature
                CONSTANT peak_temp, 2E       ;peak temperature memory
 light_test:    INPUT s1, light_port         ;test for light
                TEST s1, light_sensor
                JUMP Z, temp_test            ;jump if no light
                ADD light_count_lsb, 01      ;increment counter
                ADDCY light_count_msb, 00
  temp_test:    INPUT new_temp, temp_sensor  ;read temperature
                FETCH s2, peak_temp
                COMPARE s2, new_temp         ;compare with peak value
                JUMP NC, light_test          ;new value is smaller
                STORE new_temp, peak_temp    ;write new peak value
                JUMP light_test
```

Note - **A constant is global**. Even if a constant is defined at the end of the program file, it can be used in instructions anywhere in the program.

Constant names must not contain any spaces although the under-score character is supported. Valid characters are '0' to '9', 'a' to 'z', and 'A' to 'Z'.

'light_port' and 'temp_sensor' are used to specify port addresses. This is particularly useful when defining the hardware interface, and allows the program to be developed before the I/O addresses are fully defined. 'light_sensor' is being used to specify a data constant which in this case identifies which bit is to be tested. 'peak_temp' defines a scratch pad memory location which is then used to hold a variable.

NAMEREG Directive

The NAMEREG directive provides a way to assign a new name to any of the 16 registers. In this way the program can refer to 'variables' by name rather than as absolute register specifications. By naming registers in this way it is often easier to understand the meaning in the program without the need for so many comments. It can also help to prevent inadvertent reuse of a register with associated data corruption.

Important - The NAMEREG directive is applied in-line with the code by the assembler. Before the NAMEREG directive, the register will be named in the 'sX' style. Following the directive, <u>only</u> the new name will apply. It is also possible to rename a register again (i.e. NAMEREG counter_reg, hours) and only the new name will apply in the subsequent program lines.

```
                CONSTANT light_port, 03     ;light sensor port
                CONSTANT light_sensor, 01   ;bit0 is light sensor
                CONSTANT temp_sensor, 40    ;temperature sensor port
                NAMEREG sF, light_count_msb ;16-bit light pulse counter
                NAMEREG sE, light_count_lsb
                NAMEREG sD, new_temp        ;current temperature
                CONSTANT peak_temp, 2E      ;peak temperature memory
 light_test:    INPUT s1, light_port        ;test for light
                TEST s1, light_sensor
                JUMP Z, temp_test           ;jump if no light
                ADD light_count_lsb, 01     ;increment counter
                ADDCY light_count_msb, 00
  temp_test:    INPUT new_temp, temp_sensor ;read temperature
                FETCH s2, peak_temp
                COMPARE s2, new_temp        ;compare with peak value
                JUMP NC, light_test         ;new value is smaller
                STORE new_temp, peak_temp   ;write new peak value
                JUMP light_test
```

Register names must not contain any spaces although the under-score character is supported. Valid characters are '0' to '9', 'a' to 'z', and 'A' to 'Z'.

The register 'sD' has been renamed to be '`new_temp`' and is then used in multiple instructions making it clear what the meaning of the register contents actually are.

ADDRESS Directive

The ADDRESS directive provides a way force the assembly of the following instructions commencing at a new address value. This is useful for separating subroutines into specific locations, and vital for handling interrupts. The address must be specified as a 3-digit hexadecimal value in the range '00' to '3FF'.

In the following code segment, the ADDRESS directive defines the address for the interrupt vector.

```
        JUMP NZ, inner_long
        RETURN
        ;Interrupt Service Routine
  ISR: LOAD wait_light, 01                 ;register press of switch
        OUTPUT wait_light, wait_light_port ;turn on light
        RETURNI DISABLE                    ;continue light sequence but no more interrupts
        ADDRESS 3FF                        ;Interrupt vector
        JUMP ISR
        ;end of program
```

The log file clearly shows that the ADDRESS directive has forced the last instruction into the highest memory location in the program RAM. This is the address to which the program counter is forced during an active interrupt.

```
3E3  357E1        JUMP NZ, inner_long[3E1]
3E4  2A000        RETURN
3E5               ;Interrupt Service Routine
3E5  00A01   ISR: LOAD wait_light[sA], 01                         ;register press of switch
3E6  2CA10        OUTPUT wait_light[sA], wait_light_port[10] ;turn on light
3E7  38000        RETURNI DISABLE                             ;continue light sequence but...
3FF               ADDRESS 3FF                                 ;Interrupt vector
3FF  343E5        JUMP ISR[3E5]
3FF               ;end of program
```

KCPSM and KCPSM2 Compatibility

KCPSM and KCPSM2 are very much 'brothers' with many similarities (see 'PicoBlaze Comparison'). However, each has been tuned to the specific device architecture so there are differences.

<u>**Common points**</u>

The KCPSM3 assembler has slightly different rules concerning which labels for lines, constants, and registers are acceptable. Therefore, it may be necessary to adjust some of the user names in your program code. Typically, labels are nicely 'descriptive' and this issue will not be encountered.

The KCPSM3 macro has an INTERRUPT_ACK output signal which the previous versions did not have. It is not vital to use this signal in your design, but should be included in the component port definitions.

The internal scratch pad memory will often mean that external memory connected to I/O ports can be removed. This will simplify the logic design and require the code to reflect the use of STORE and FETCH instructions in place of INPUT and OUTPUT.

<u>**KCPSM to KCPSM3**</u>

KCPSM3 is in every way a superset of of KCPSM so there will be very few issues migrating a KCPSM based design and code. The address range of KCPSM3 supports a program which is four times larger than KCPSM and therefore all programs will be able to fit. Code will need to reflect that absolute address values need to be specified with 3 hexadecimal digits (not 2). The use of line labels will mean that most cases will be handled automatically by the assembler, but special care should be taken with ADDRESS directives. Most critical is that the interrupt vector will need to be located at '3FF' (not FF).

<u>**KCPSM2 to KCPSM3**</u>

KCPSM3 has 16 registers compared with the 32 registers of KCPSM2. The default register names used in KCPSM2 are 's00' to 's1F' and will need to be modified to conform to the default names 's0' to 'sF' available in KCSPM. Although the use of NAMEREG directives will be helpful, some fundamental changes will almost certainly be required to compensate for the lower number of available registers. The internal scratch pad memory provides 64 locations which should more than compensate for the lower number of registers but obviously requires a change to the coding style. The program address range and interrupt vector are identical.

XILINX

PicoBlaze Comparison

This chart shows a comparison of the features offered by the FPGA variants of PicoBlaze. XAPP387 describes the CoolRunner implementation of an 8-bit micro controller which was also based on the original KCPSM processor.

	KCPSM	KCPSM2	KCPSM3
Target Devices	Spartan-II, Spartan-IIE, Virtex, Virtex-E	Virtex-II, Virtex-IIPRO	Spartan-3, Virtex-II, Virtex-IIPRO
Program Size	256 instructions (256×16 Block RAM)	1024 instructions (1024×18 Block RAM)	1024 instructions (1024×18 Block RAM)
Registers	16	32	16
Scratch-Pad Memory	-	-	64 Bytes
Size	76 Slices	84 Slices	96 Slices
CALL/RETURN stack	15 levels	31 levels	31 levels
Features and Comments	Smallest and oldest! Very well used and proven. Relatively small program space.	Register rich. Virtex-II devices only. Can **not** migrate design directly to Spartan-3.	COMPARE and TEST instructions, PARITY test, Scratch-pad memory, INTERRUPT_ACK signal

As with most things, there is a clear trend for PicoBlaze to become larger as more features are added. The author welcomes all feedback regarding this trend to determine the size acceptable for a programmable state machine (PSM).

XILINX

Interrupt Handling

Effective interrupt handling is a skillful task and this document does not attempt to explain how and when an interrupt should be used. The information supplied should be adequate for the capability of KCPSM3 to be assessed and for interrupt based systems to be created.

Default State - By default the interrupt input is disabled. This means that the entire 1024 words of program space can be 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 a program execution where an interrupt would be unacceptable, a DISABLE INTERRUPT can be used. Since an active interrupt will automatically disable the interrupt input, the interrupt service routine will end with a RETURNI instruction which also includes the option to ENABLE or DISABLE the interrupt input as it returns to the main program.

What happens during an interrupt? The program counter is pushed onto the stack and the values of the CARRY and ZERO flags are preserved (to be restored by the RETURNI instruction). The interrupt input is automatically disabled. Finally the program counter is forced to address 3FF (last program memory location) from which the next instruction is executed.

Effects of an active interrupt.
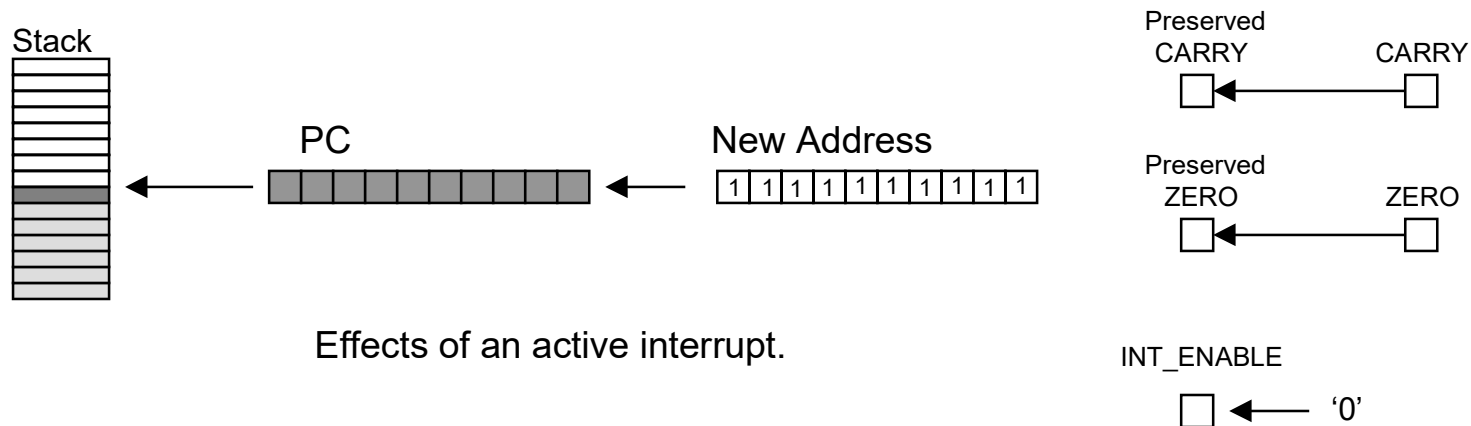
Basics of Interrupt Handling

Since the interrupt will force the program counter to address '3FF' it will generally be necessary to ensure that a jump vector to a suitable interrupt service routine (ISR) is located at this address otherwise the program will 'roll over' to address zero.

In most cases an ISR will be provided. The routine can be located at any position in the program and jumped to by the interrupt vector located at the '3FF" address. The ISR will perform the required tasks and then end in RETURNI with ENABLE or DISABLE.

Simple Example - The following example illustrates a very simple interrupt handling routine…….

The KCPSM3 is generally involved with generating waveforms to an output by writing the values '55' and 'AA' to the 'waveform_port' (port address 02). It does this at regular intervals by decrementing a register (s0) based counter 7 times in a loop.

When an interrupt is asserted, the KCPSM3 breaks off from the waveform generation and simply increments a separate counter register (sA) and writes the counter value to the 'counter_port' (port address 04).
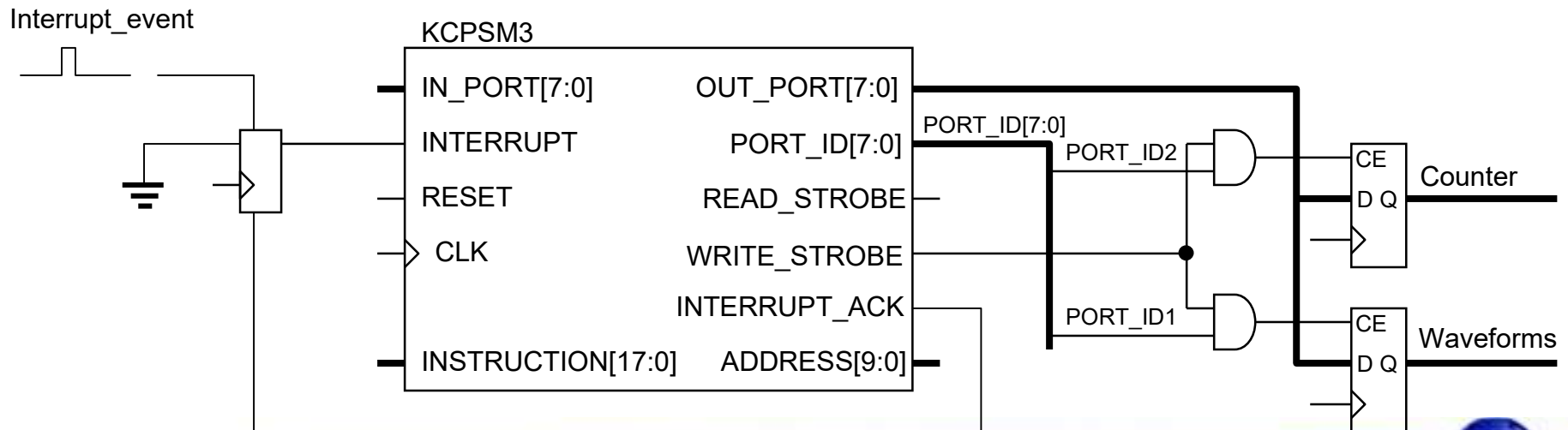
Example Design (VHDL)

The following VHDL shows the addition of the data capture registers and interrupt control to the processor. Note the simplified port decoding logic through careful selection of port addresses. The complete VHDL file is supplied as 'kcpsm3_int_test.vhd'.

```vhdl
IO_registers: process(clk)
  begin

    if clk'event and clk='1' then

      -- waveform register at address 02
      if port_id(1)='1' and write_strobe='1' then
        waveforms <= out_port;
      end if;

      -- Interrupt Counter register at address 04
      if port_id(2)='1' and write_strobe='1' then
        counter <= out_port;
      end if;

    end if;
  end process IO_registers;
```

```vhdl
interrupt_control: process(clk)
begin

  if clk'event and clk='1' then

    if interrupt_ack='1' then
      interrupt <= '0';
     elsif interrupt_event='1' then
      interrupt <= '1';
     else
      interrupt <= interrupt;
    end if;

  end if;
end process interrupt_control;
```

XILINX

Interrupt Service Routine

In the assembler log file for the example, it can be seen that the interrupt service routine has been force to compile at address '2B0', and that the waveform generation is located in the base addresses. This makes it easier to observe the interrupt in action in the operation waveforms. This program is supplied as 'int_test.psm' for you to assemble yourself.

```
000                          ;Interrupt example
000                          ;
000                          CONSTANT waveform_port, 02                    ;bit0 will be data
000                          CONSTANT counter_port, 04
000                          CONSTANT pattern_10101010, AA
000                          NAMEREG sA, interrupt_counter
000                          ;
000  00A00     start:  LOAD interrupt_counter[sA], 00                     ;reset interrupt counter
001  002AA             LOAD s2, pattern_10101010[AA]                       ;initial output condition
002  3C001             ENABLE INTERRUPT
003                          ;
003  2C202  drive_wave:  OUTPUT s2, waveform_port[02]
004  00007             LOAD s0, 07                                          ;delay size
005  1C001      loop:  SUB s0, 01                                          ;delay loop
006  35405             JUMP NZ, loop[005]
007  0E2FF             XOR s2, FF                                           ;toggle waveform
008  34003             JUMP drive_wave[003]
009                          ;
2B0                          ADDRESS 2B0
2B0  18A01  int_routine:  ADD interrupt_counter[sA], 01                    ;increment counter
2B1  2CA04             OUTPUT interrupt_counter[sA], counter_port[04]
2B2  38001             RETURNI ENABLE
2B3                          ;
3FF                          ADDRESS 3FF
3FF  342B0             JUMP int_routine[2B0]
```

Main program delay loop where most time is spent

Interrupt Service Routine (located at address 2B0 onwards)

Interrupt vector set at address 3FF and causing JUMP to service routine

XILINX

Interrupt Operation

The waveforms below taken from an actual ModelSim-XE simulation show the operation of KCPSM3 when executing the example program at the time of an interrupt. The VHDL test bench used to generate these waveforms is supplied as 'testbench.vhd'.

By observing the address bus, it is possible to see that the program is busy generating the waveforms and even shows the 'waveforms' port being written the 'AA' pattern value. Then whilst in the delay loop which repeats addresses '005' and '006' it receives an interrupt pulse.

It can be seen that KCPSM3 took a few clock cycles to respond to this particular pulse (see 'timing of interrupt pulses') before forcing the address bus to '3FF' and issuing an INTERRUPT_ACK pulse. From '3FF', the obvious JUMP to the service routine located at '2B0' can be seen to follow and a new counter value (in this case '03') is written to the 'counter' port.
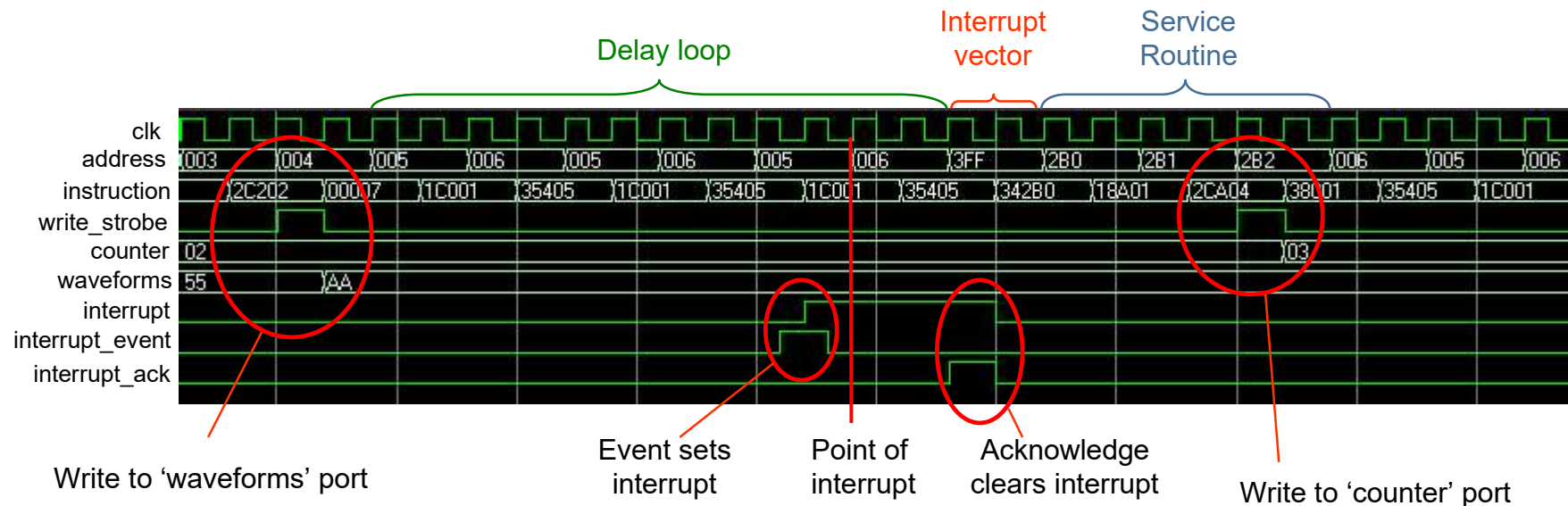
The operation of a KCPSM3 interrupt can also be observed. It can be seen that the last address active before the interrupt is '006'. The JUMP NZ instruction obtained at this address (op-code 35405) is <u>not</u> executed. The flags preserved are those which were set at the end of the instruction at the previous address (SUB s0,01). The RETURNI has restored the flags and returned the program to address '006' in order that the JUMP NZ instruction can at last be executed.

XILINX

Timing of Interrupt Pulses

It is clear from the previous simulation waveforms that the constant two cycles per instruction is maintained at all times. Since this includes an interrupt, the use of single cycle pulse for interrupt can be risky. However, the following waveform can be used to determine the exact cycle on which the interrupt is observed and the true reaction rate of KCPSM3.

Interrupt sampled on clock edge associated
with change of address

CLK		
INTERRUPT		
INTERRUPT_ACK		
ADDRRESS[9:0]	005	006	005	006	3FF	2B0
INSTRUCTION[17:0]	1C001	35405	1C001	35405	342B0	

2 cycles
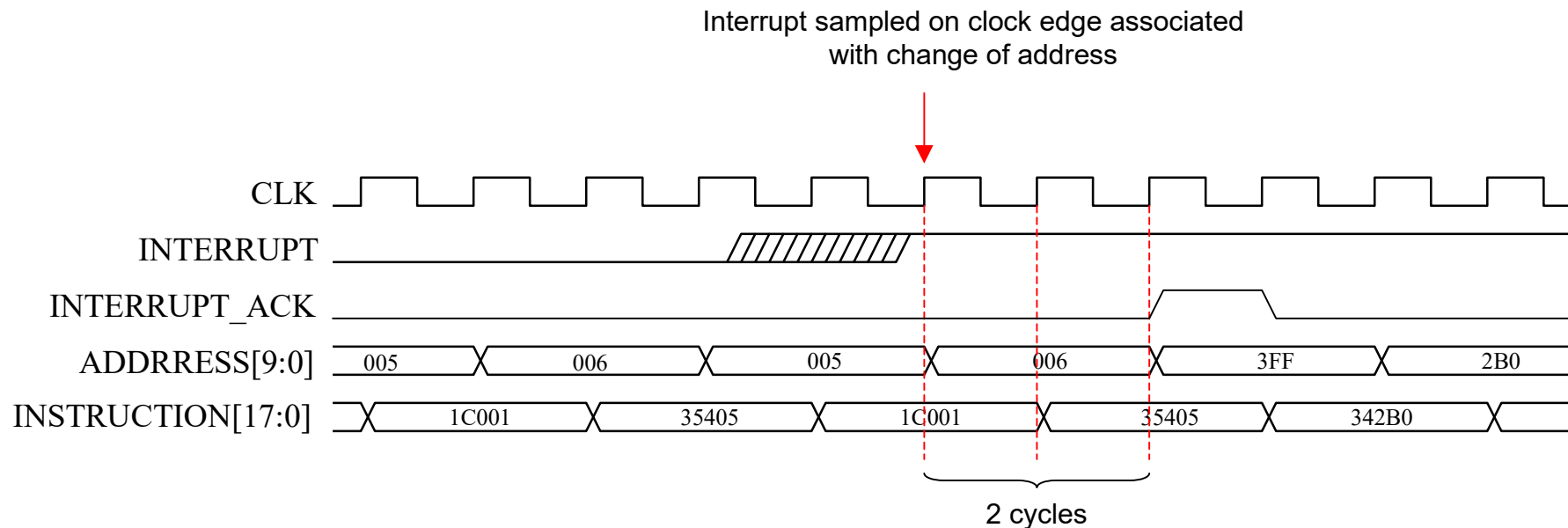
It is therefore advisable that an interrupt signal should be active for a minimum of two KCPSM3 rising clock cycle edges. It is generally advisable to use the INTERRUPT_ACK signal in a similar way to that demonstrated in the example to ensure that an interrupt is not missed.

When using logic to combine multiple sources of interrupt, a typical interrupt service routine will read a specific port to determine the reason for interrupt. In this case, the READ_STROBE and PORT_ID can be decoded and used to clear the external interrupt register.

CALL/RETURN Stack

KCPSM3 contains an automatic embedded stack which is used to store the program counter value during a CALL instruction or interrupt and restore the program counter value during a RETURN or RETURNI instruction. The stack does not need to be initialised or require any control by the user. However, **the stack can only support nested subroutine calls to a depth of 31**.

This simple program can calculate the sum of all integers up to a certain value, i.e. 'sum_of_value' when value=5 is 1+2+3+4+5=15. In this case, the sum of integers up to the value 31 (1F hex) is calculated to be 496 (01F0 hex). This is achieved by using a recursive call of a subroutine and results in the full depth of the call/return stack being utilised. Obviously, this is not a particularly efficient implementation of this algorithm, but it does fully test the stack.
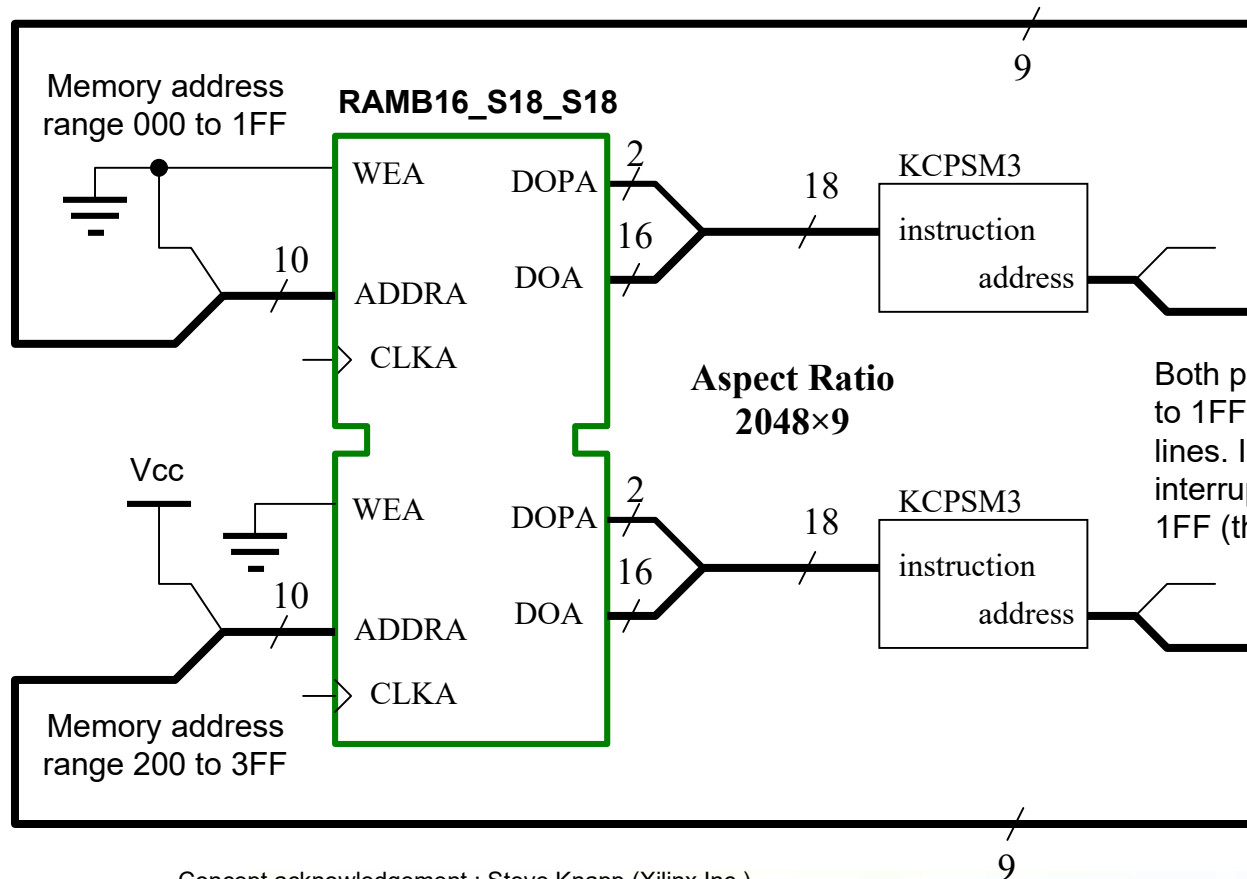
Increasing value to 20 (32 decimal) will result in incorrect operation of KCPSM3. The stack is a cyclic buffer, so the 'bottom' of the stack will be overwritten by the 'top' of the stack during the 32nd nested CALL instruction.

```
               NAMEREG s0, total_low
               NAMEREG s1, total_high
               NAMEREG s8, value
               ;
        start: LOAD value, 1F            ;find sum of all values to 31
               LOAD total_low, 00        ;clear 16-bit total
               LOAD total_high, 00
               CALL sum_to_value         ;calculate sum of all numbers up to value
               OUTPUT total_high, 02     ;Result will be 496 (01F0 hex)
               OUTPUT total_low, 01
               JUMP start
               ;
               ;Subroutine called recursively
               ;
 sum_to_value: ADD total_low, value      ;perform 16-bit addition
               ADDCY total_high, 00
               SUB value, 01             ;reduce value by 1
               RETURN Z                  ;finished if down to zero
               CALL sum_to_value         ;recursive call of subroutine
               RETURN                    ;definitely finished!
```

Sharing Program Space

For ease of design and possibly to meet system performance requirements, it is often desirable to use multiple KCPSM3 macros in the same device. Each KCPSM3 is designed to work with a single Block RAM which provides 1024 locations in the Spartan-3 and Virtex-II devices. For many control and state machine applications, this program size may be found to be excessive and lead to wasted block memory resources.

Since block RAM is dual port, it is quite possible to connect two KCPSM3 macros to the same block memory…..

Memory address range 000 to 1FF

RAMB16_S18_S18

WEA DOPA 2
      DOA 16 18 KCPSM3 instruction address

10 ADDRA

CLKA

Aspect Ratio 2048×9

Vcc

WEA DOPA 2
      DOA 16 18 KCPSM3 instruction address

10 ADDRA

CLKA

Memory address range 200 to 3FF

9

Both processors use an address range 000 to 1FF and only use the lower 9 address lines. Interrupts will still work, but the interrupt vector must be placed at address 1FF (the last effective memory location).

Concept acknowledgement : Steve Knapp (Xilinx Inc.)
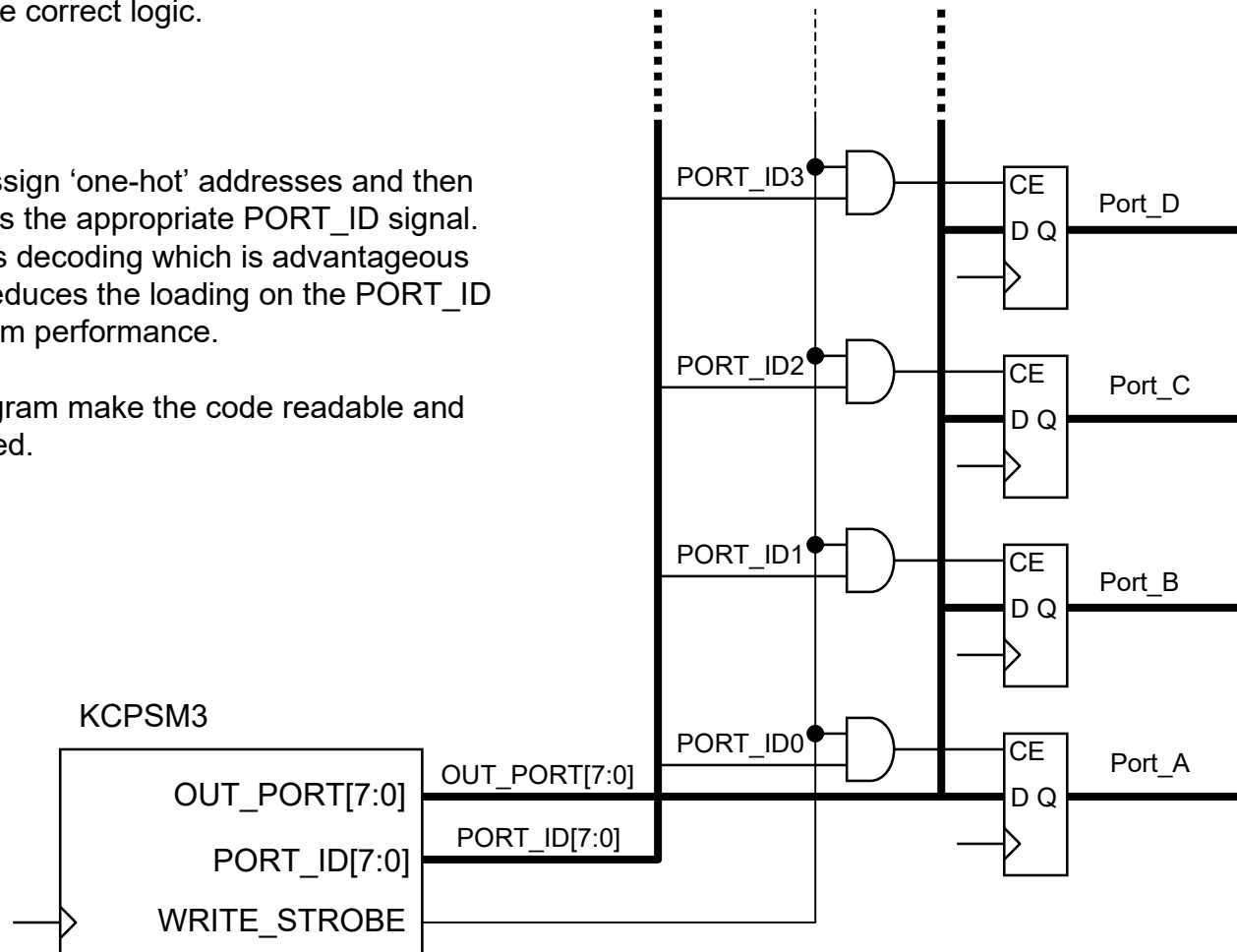
Design of Output Ports

Being thoughtful about your interface circuit design will enable the logic to remain compact and performance to be maintained. The following diagrams show suitable circuits for output ports, input ports and connection of memory. If you are using a synthesis tool, it is advisable to check that your code is not describing a circuit which is more complex than is really required and that the synthesis tool is implementing the correct logic.

Simple Outputs

For 8 or less simple output ports try to assign 'one-hot' addresses and then make sure that your design only decodes the appropriate PORT_ID signal. This greatly reduces the logic for address decoding which is advantageous for lower cost and performance. It also reduces the loading on the PORT_ID bus which is often critical to overall system performance.

Use of CONSTANT directives in the program make the code readable and help ensure that the correct ports are used.

```
CONSTANT Port_A,01
CONSTANT Port_B,02
CONSTANT Port_C,04
CONSTANT Port_D,08
;
OUTPUT s0,Port_A
OUTPUT s1,Port_B
OUTPUT s2,Port_C
OUTPUT s4,Port_D
```


Note that all blocks share a common clock

Design of Output Ports

Fully Decoded Outputs and high performance

When there is a requirement to address blocks of memory and many simple ports, a large number of the 256 output port locations may be used requiring the PORT_ID addresses to be more fully decoded. If performance is critical, then careful design will again be advantageous.

The key observation is that during a write operation the PORT_ID and OUT_PORT are provided for 2 clock cycles with the WRITE_STROBE only active during the second of the two cycles (see read and write strobes). Although time specifications can be used to cover the 2-cycle paths, it is often easier to insert pipeline stages and split the address decoding effort as shown here.

Port Mapping

Dual Port (16 bytes) - 00 to 0F

Single Port (32 bytes) - 20 to 3F

Port_A - 40

KCPSM3
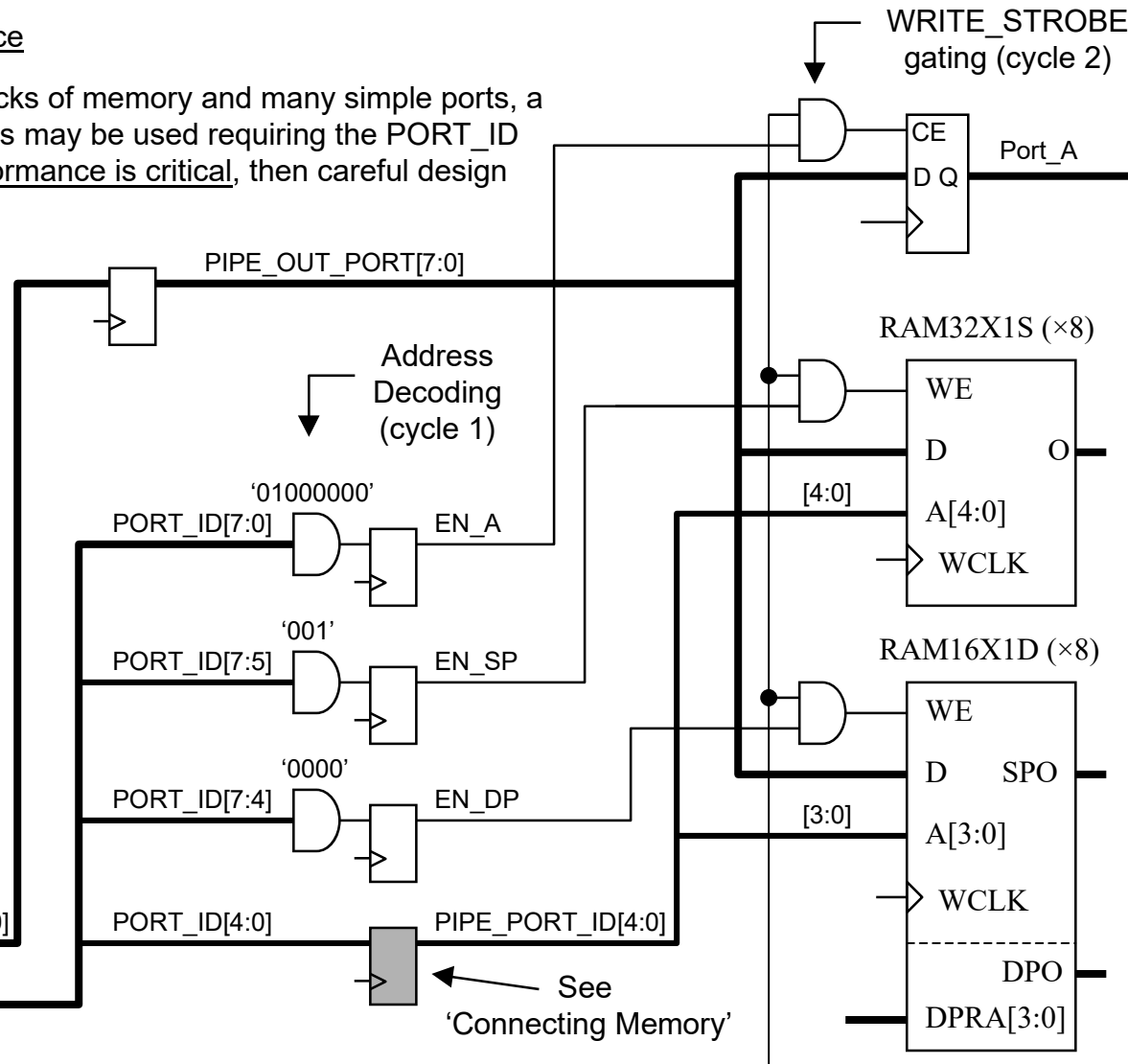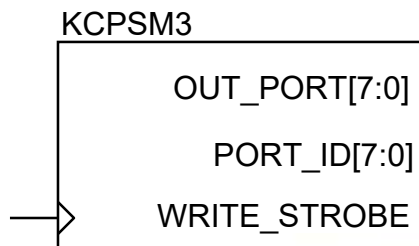
Note that all blocks share a common clock

Design of Input Ports

The connection of input ports leads to the definition of a multiplexer. Obviously the size of this multiplexer is proportional to the number of inputs and having many inputs can lead to issues with performance unless care is taken with the description of this multiplexer structure.

Simple Inputs

For 8 or less simple input ports the single multiplexer is ideally suited to connect the various input signals to the IN_PORT. It is advisable to check the results of synthesis to ensure that the special MUXF5 and MUXF6 are being employed to make the most efficient multiplexer structure.

Note that all blocks share a common clock
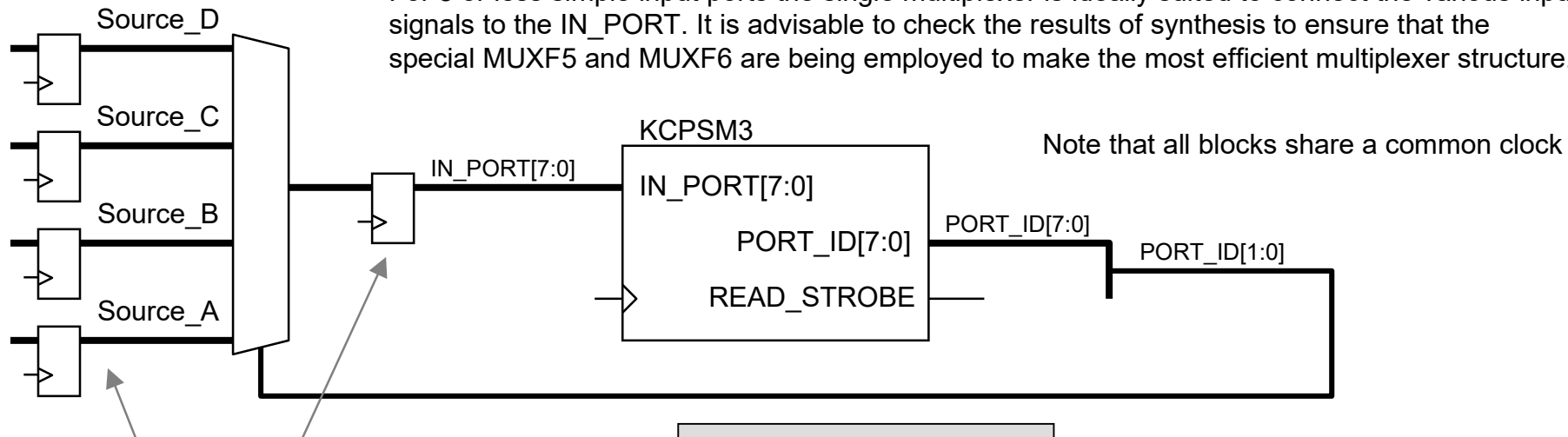
Because the PORT_ID is valid for 2 clock cycles the multiplexer can be registered to maintain performance.

In the majority of cases, the actual clock cycle at which an input is read by the processor isn't critical. Therefore the paths from the sources can typically be registered such as using the I/O registers when coming from actual device pins. This will help simplify time specifications, avoid reports of 'false paths' and lead to reliable designs.

```
CONSTANT Source_A,00
CONSTANT Source_B,01
CONSTANT Source_C,02
CONSTANT Source_D,03
;
INPUT s0,Source_A
INPUT s1,Source_B
INPUT s2,Source_C
INPUT s3,Source_D
```

The multiplexer means that the best addresses to assign for input ports are normal binary encoding.
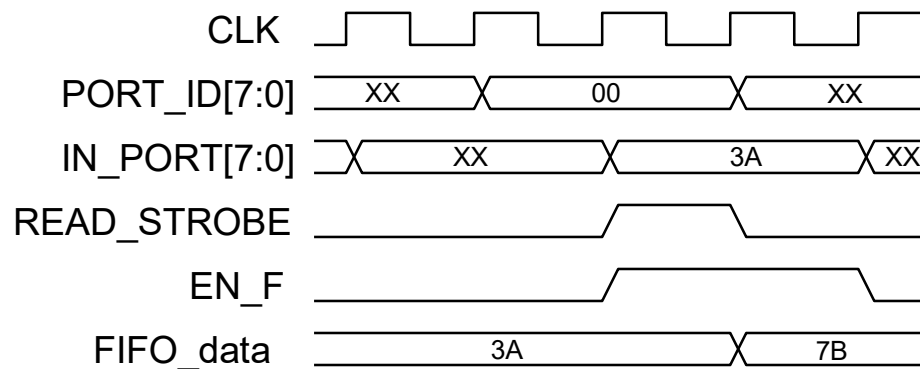
IMPORTANT

Failure to include a register anywhere in the path from PORT_ID to IN_PORT is the most common reason for observing significantly lower clock rates than indicated in the 'Size and Performance' section of this manual. So make sure you have one!

Design of Input Ports

Occasionally it will be important that a circuit providing data to KCPSM3 to know that it has been read. The obvious example is a FIFO buffer which will then prepare the next data to be read.

Note that all blocks share a common clock
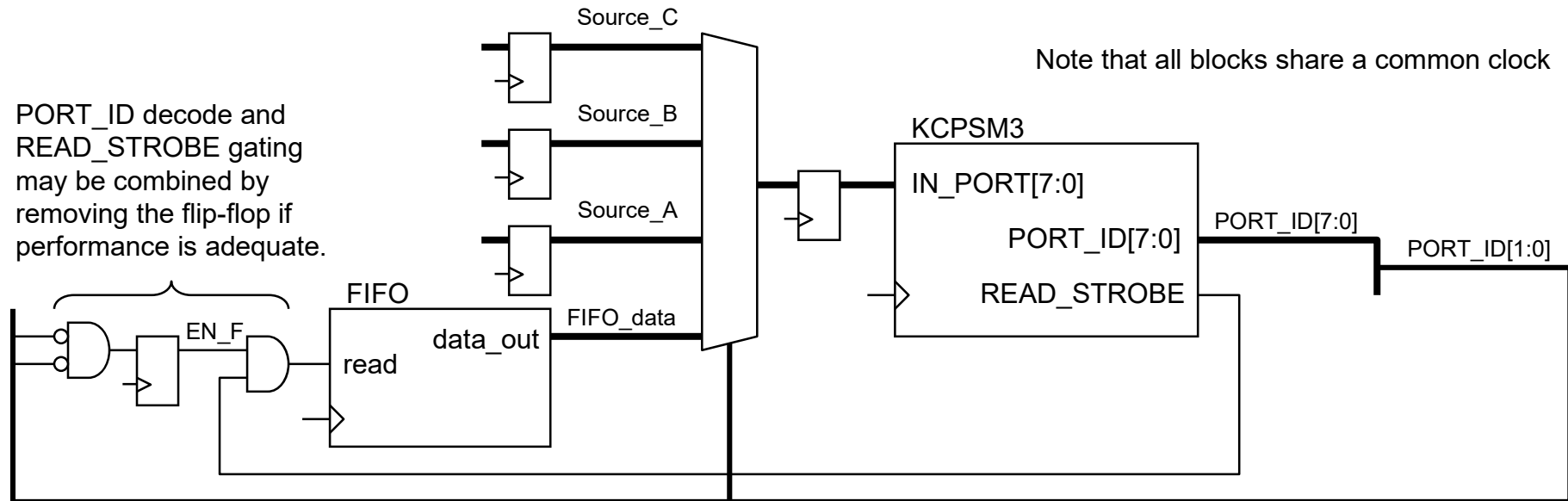
PORT_ID decode and READ_STROBE gating may be combined by removing the flip-flop if performance is adequate.

Source_C

Source_B

Source_A

FIFO_data

FIFO

data_out

read

EN_F

KCPSM3

IN_PORT[7:0]

PORT_ID[7:0]

READ_STROBE

PORT_ID[7:0]

PORT_ID[1:0]

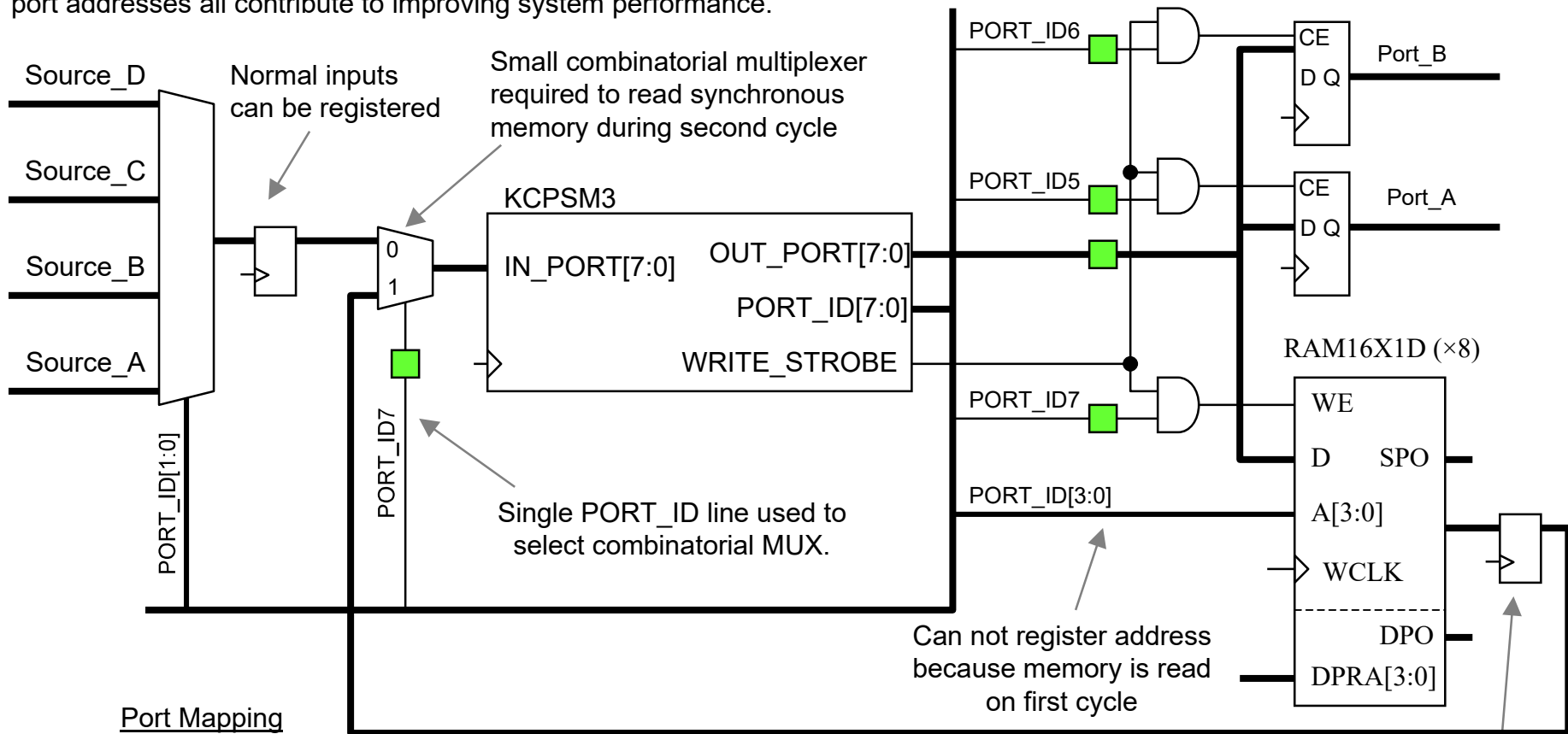| CLK				
PORT_ID[7:0]	XX	00	XX	
IN_PORT[7:0]	XX	3A	XX	
READ_STROBE				
EN_F				
FIFO_data	3A	7B		

The data path from the FIFO is quite separate to the read acknowledgement circuit.

In this example the FIFO is assigned the address '00'. Initially the FIFO is providing data with the value '3A'. The act of reading the port causes the FIFO to provide the next data of value '7B'.

XILINX

Connecting Memory

The connection of memory (Dual port is ideal for communication with other modules) is the most common cause for reduction in system performance. Observing where pipeline registers can be inserted, splitting the input multiplexer and careful allocation of port addresses all contribute to improving system performance.

Source_D

Source_C

Source_B

Source_A

Normal inputs can be registered

Small combinatorial multiplexer required to read synchronous memory during second cycle

KCPSM3

IN_PORT[7:0]   OUT_PORT[7:0]

PORT_ID[7:0]

WRITE_STROBE

PORT_ID[1:0]

PORT_ID7

Single PORT_ID line used to select combinatorial MUX.

PORT_ID6

PORT_ID5

CE
D Q   Port_B

CE
D Q   Port_A

RAM16X1D (×8)

PORT_ID7

PORT_ID[3:0]

WE

D   SPO

A[3:0]

WCLK

DPO

DPRA[3:0]

Can not register address because memory is read on first cycle

Port Mapping

Input Source_A - 00
Input Source_B - 01
Input Source_C - 02
Input Source_D - 03

Read/Write memory (16 bytes) - 80 to 8F
Output Port_A - 20
Output Port_B - 40

Register output of distributed RAM to make 'synchronous read' style. This breaks the 2-cycle path mid-way. Block memory is 'synchronous read' by default.

■ = Additional places to insert flip-flops if really necessary for performance.

XILINX