# EE354L
# Picoblaze Basics
# Some Significant Aspects

Date of first creation: 7/11/2016
Date of last revision: 9/15/2018, 3/2/2020

**Gandhi Puvvada**

Reference: KCPSM6_User_Guide_30Sept14.pdf from Xilinx
https://www.xilinx.com/products/intellectual-property/picoblaze.html

## Items crossed off

For initial acquaintance of the picoblaze and its operation, a few of its advanced features are not that important.

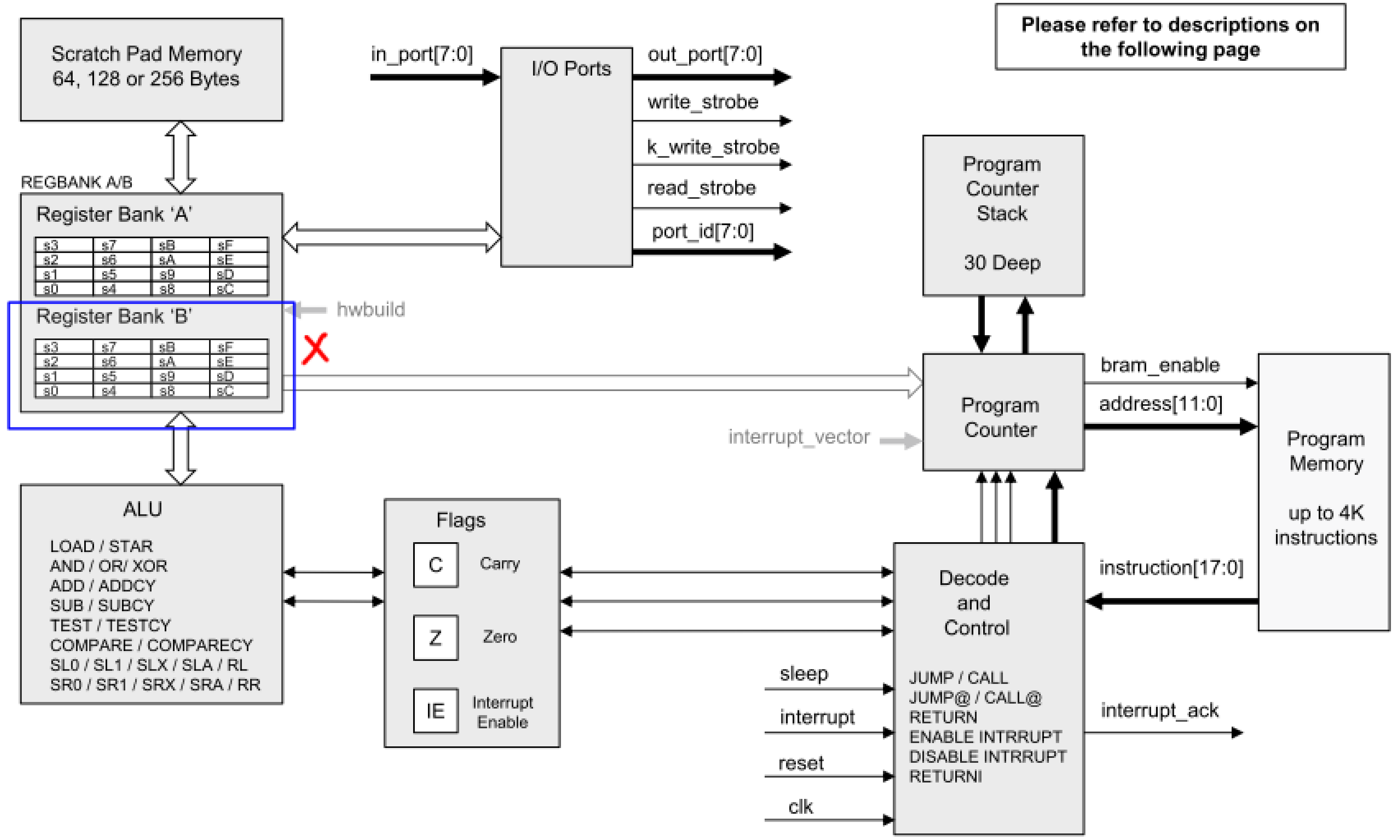
I have crossed them off so that you can focus on the essential aspects first.

Significant items that were crossed off:
1. The second register bank and the instructions associated with it
2. Scratch pad memory operations
3. Shift and rotate operations
4. Sleep, rdl and reset pins

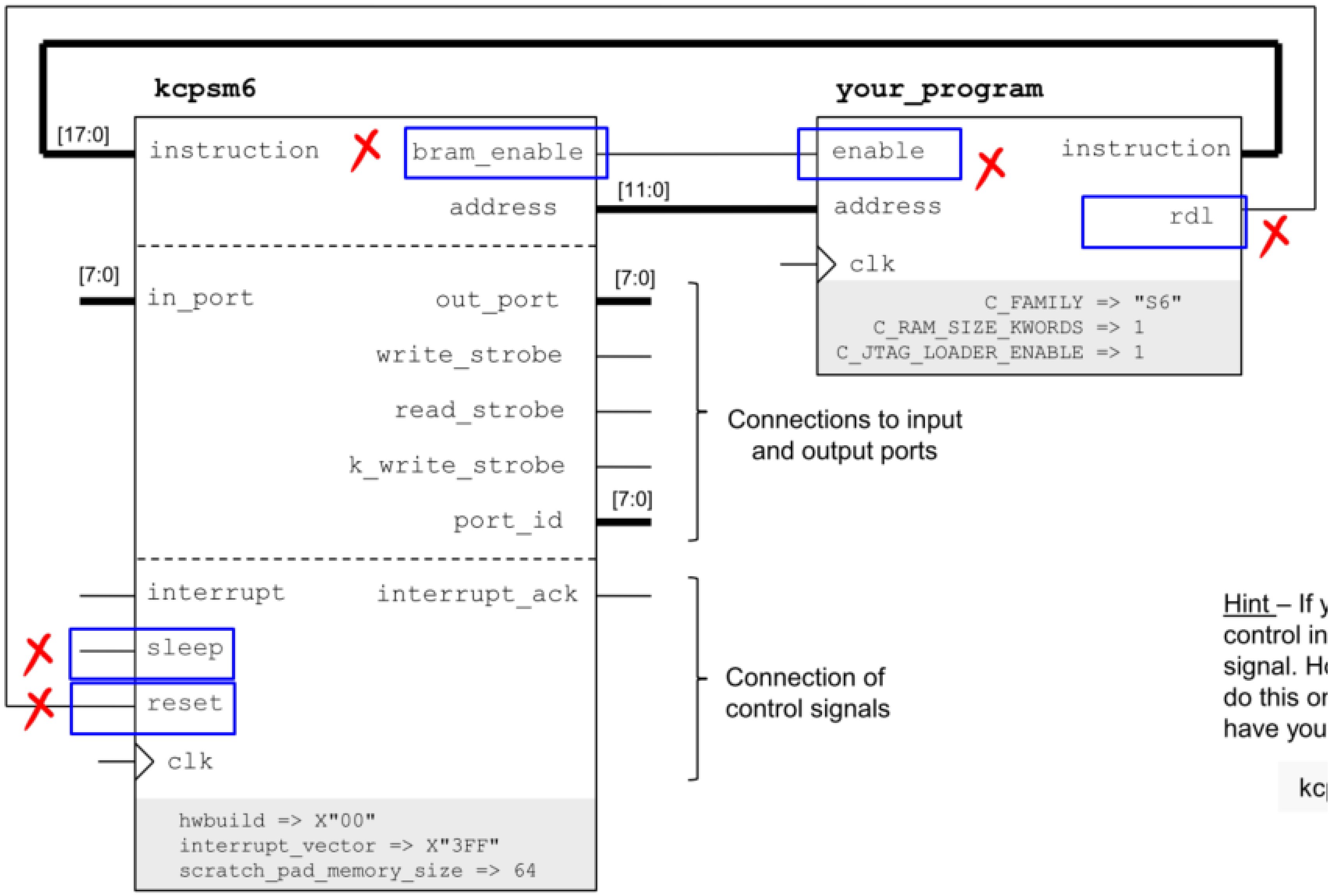
# Contents

# KCPSM6 Architecture and Features

**Please refer to descriptions on the following page**



XILINX

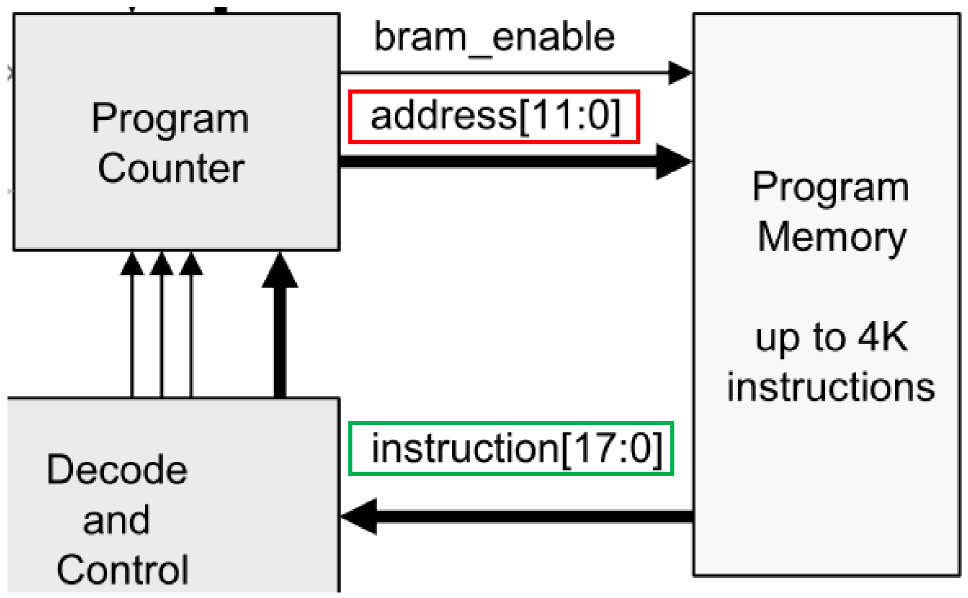


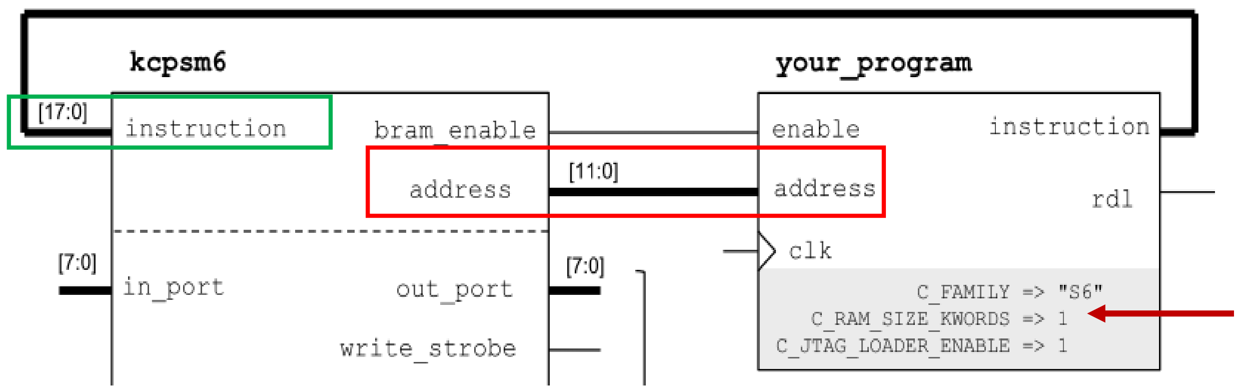
Hint – If y
control in
signal. H
do this o
have you

kc

# Instruction Memory size

The address of the instruction memory is 12 bits.
So you can have up to 4K of instruction memory. 2^12 = 4K.
The choices for the size of instruction memory are 1K, 2K, and 4K.
The instructions are 18-bit long.
Hence the sizes of the instruction memory are 1Kx18, 2Kx18, 4Kx18.





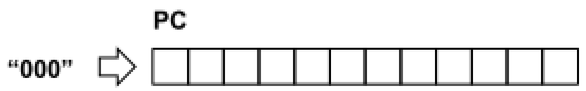
## Starting value of PC on reset

It is basic and important that the PC starts with a known desired value after reset in every processor and the very first instruction to be fetched is located there. This starting address may vary for different processors. Notably, for the x86 processors, it is FFFF0 hex.

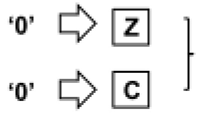
For our Picoblaze and for many processors, it is all zeros (000 hex).

## RESET

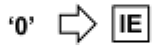
Following device configuration KCPSM6 generates an internal reset to ensure predictable a reliable operation. The 'reset' input can then be driven High at any time during operation to force a restart (e.g. When the 'rdl' signal from JTAG Loader is asserted).



Program counter (PC) is forced to address '000' ready to fetch and execute the instruction located in the first location of the program memory.

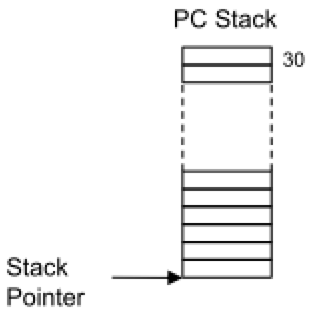


The zero and carry flags are reset.



Interrupts are disabled.

Register bank 'A' is selected and therefore 'A' is the default bank of registers.



The pointer in the program counter stack is reset to ensure that the program is able to execute programs in which up to 30 nested subroutine calls can be made.

Note – If you should inadvertently write a program whose execution results in stack overflow or stack underflow then KCPSM6 will automatically generate an internal reset.

Hint– Following device power up and configuration the contents of all registers and scratch pad memory locations will be zero. Any subsequent reset will perform all the items shown above but registers and scratch pad memory will retain the values. This can be useful in certain application but your code should not rely of values being zero if manual reset is to be used during operation.

Picoblaze ISA (Instruction Set Architecture)

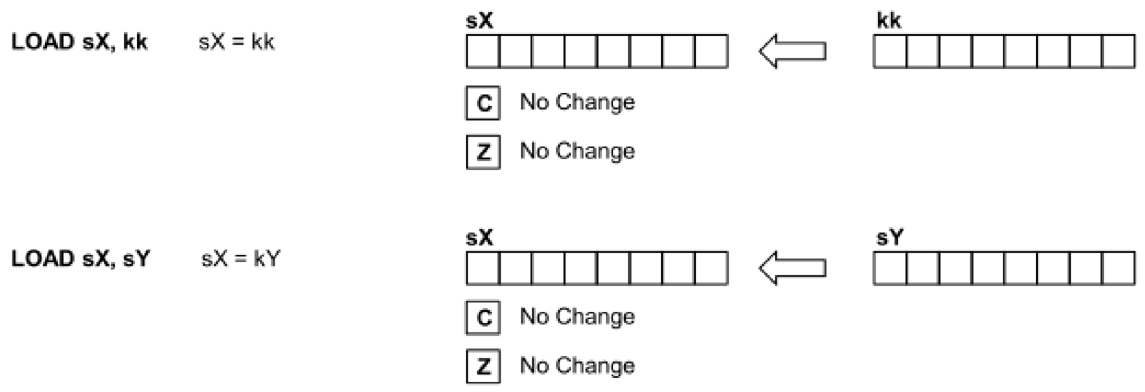
# Introduction to the Assembly Language Instructions

# Instruction format

The instructions are 18-bit long and are written in 5-digit hex (see page 54, copy of which is on the next page). Left most hex digit goes from 0 to 3 only, hence it requires only 2 binary bits. The opcode is the left-most 6 bits (left-most 2 hex digits).
Example: LOAD sX, kk is translated as 01xkk

| Page | Opcode | Instruction |
|---|---|---|
| **Register loading** | | |
| 55 | 00xy0 | LOAD sX, sY |
| 55 | 01xkk | LOAD sX, kk |

```
aaa : 12-bit address 000 to FFF
 kk : 8-bit constant 00 to FF
 pp : 8-bit port ID 00 to FF
  p : 4-bit port ID 0 to F
 ss : 8-bit scratch pad location 00 to FF
  x : Register within bank s0 to sF
  y : Register within bank s0 to sF
```



# KCPSM6 Instruction Set

| | |
|---|---|
| aaa | 12-bit address 000 to FFF |
| kk | 8-bit constant 00 to FF |
| pp | 8-bit port ID 00 to FF |
| p | 4-bit port ID 0 to F |
| ss | 8-bit scratch pad location 00 to FF |
| x | Register within bank s0 to sF |
| y | Register within bank s0 to sF |

## Register loading

| Page | Opcode | Instruction |
|---|---|---|
| 55 | 00xy0 | LOAD sX, sY |
| 55 | 01xkk | LOAD sX, kk |
| 71 | 16xy0 | STAR sX, sY |
| 71 | 17xkk | STAR sX, kk |

X

## Logical

| Page | Opcode | Instruction |
|---|---|---|
| 56 | 02xy0 | AND sX, sY |
| 56 | 03xkk | AND sX, kk |
| 57 | 04xy0 | OR sX, sY |
| 57 | 05xkk | OR sX, kk |
| 58 | 06xy0 | XOR sX, sY |
| 58 | 07xkk | XOR sX, kk |

## Arithmetic

| Page | Opcode | Instruction |
|---|---|---|
| 59 | 10xy0 | ADD sX, sY |
| 59 | 11xkk | ADD sX, kk |
| 60 | 12xy0 | ADDCY sX, sY |
| 60 | 13xkk | ADDCY sX, kk |
| 61 | 18xy0 | SUB sX, sY |
| 61 | 19xkk | SUB sX, kk |
| 62 | 1Axy0 | SUBCY sX, sY |
| 62 | 1Bxkk | SUBCY sX, kk |

## Test and Compare

| Page | Opcode | Instruction |
|---|---|---|
| 63 | 0Cxy0 | TEST sX, sY |
| 63 | 0Dxkk | TEST sX, kk |
| 64 | 0Exy0 | TESTCY sX, sY |
| 64 | 0Fxkk | TESTCY sX, kk |
| 65 | 1Cxy0 | COMPARE sX, sY |
| 65 | 1Dxkk | COMPARE sX, kk |
| 66 | 1Exy0 | COMPARECY sX, sY |
| 66 | 1Fxkk | COMPARECY sX, kk |

Page 54

## Shift and Rotate

| Page | Opcode | Instruction |
|---|---|---|
| 67 | 14x06 | SL0 sX |
| 67 | 14x07 | SL1 sX |
| 67 | 14x04 | SLX sX |
| 67 | 14x00 | SLA sX |
| 67 | 14x02 | RL sX |
| 68 | 14x0E | SR0 sX |
| 68 | 14x0F | SR1 sX |
| 68 | 14x0A | SRX sX |
| 68 | 14x08 | SRA sX |
| 68 | 14x0C | RR sX |

X

## Register Bank Selection

| Page | Opcode | Instruction |
|---|---|---|
| 70 | 37000 | REGBANK A |
| 70 | 37001 | REGBANK B |

X

## Input and Output

| Page | Opcode | Instruction |
|---|---|---|
| 73 | 08xy0 | INPUT sX, (sY) |
| 73 | 09xpp | INPUT sX, pp |
| 74 | 2Cxy0 | OUTPUT sX, (sY) |
| 74 | 2Dxpp | OUTPUT sX, pp |
| 78 | 2Bkkp | OUTPUTK kk, p |

## Scratch Pad Memory

(64, 128 or 256 bytes)

| Page | Opcode | Instruction |
|---|---|---|
| 81 | 2Exy0 | STORE sX, (sY) |
| 81 | 2Fxss | STORE sX, ss |
| 82 | 0Axy0 | FETCH sX, (sY) |
| 82 | 0Bxss | FETCH sX, ss |

X

## Interrupt Handling

| Page | Opcode | Instruction |
|---|---|---|
| 83 | 28000 | DISABLE INTERRUPT |
| 83 | 28001 | ENABLE INTERRUPT |
| 84 | 29000 | RETURNI DISABLE |
| 84 | 29001 | RETURNI ENABLE |

## Jump

| Page | Opcode | Instruction |
|---|---|---|
| 87 | 22aaa | JUMP aaa |
| 88 | 32aaa | JUMP Z, aaa |
| 88 | 36aaa | JUMP NZ, aaa |
| 88 | 3Aaaa | JUMP C, aaa |
| 88 | 3Eaaa | JUMP NC, aaa |
| 89 | 26xy0 | JUMP@ (sX, sY) |

## Subroutines

| Page | Opcode | Instruction |
|---|---|---|
| 92 | 20aaa | CALL aaa |
| 93 | 30aaa | CALL Z, aaa |
| 93 | 34aaa | CALL NZ, aaa |
| 93 | 38aaa | CALL C, aaa |
| 93 | 3Caaa | CALL NC, aaa |
| 94 | 24xy0 | CALL@ (sX, sY) |
| 96 | 25000 | RETURN |
| 97 | 31000 | RETURN Z |
| 97 | 35000 | RETURN NZ |
| 97 | 39000 | RETURN C |
| 97 | 3D000 | RETURN NC |
| 98 | 21xkk | LOAD&RETURN sX, kk |

X

## Version Control

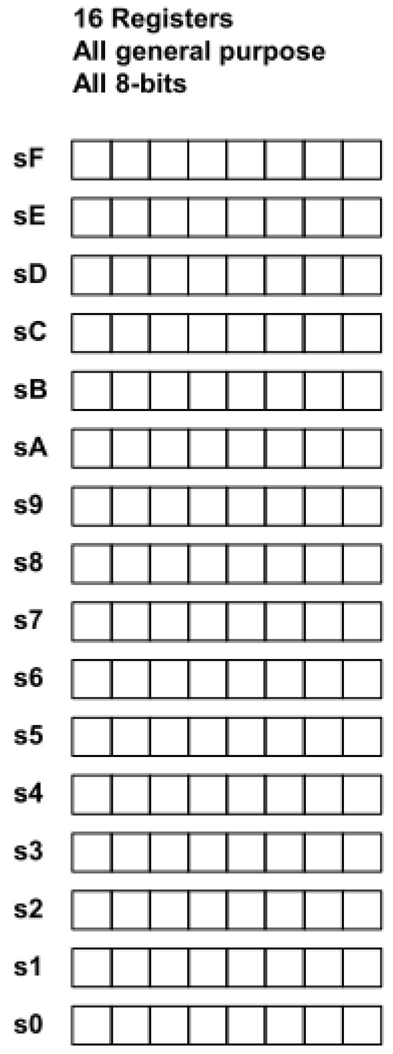
| Page | Opcode | Instruction |
|---|---|---|
| 101 | 14x80 | HWBUILD sX |

X

XILINX

# Instruction format continued

Notice that the register file has 16 registers, each of 8 bits.
The processor is an 8-bit processor, meaning it will add or subtract or move 8 bits at a time. Since there are 16 registers, the registers are named s0 through sF. The register ID is four bits (one hex digit).



| Page | Opcode | Instruction |
|---|---|---|
| **Register loading** | | |
| 55 | 00xy0 | LOAD sX, sY |
| 55 | 01xkk | LOAD sX, kk |

```
LOAD sX, sY
00xy0
LOAD s2, s5
00250
```

$000000\_0010\_0101\_\boxed{0000}$

4 bits wasted!
They are zeros always

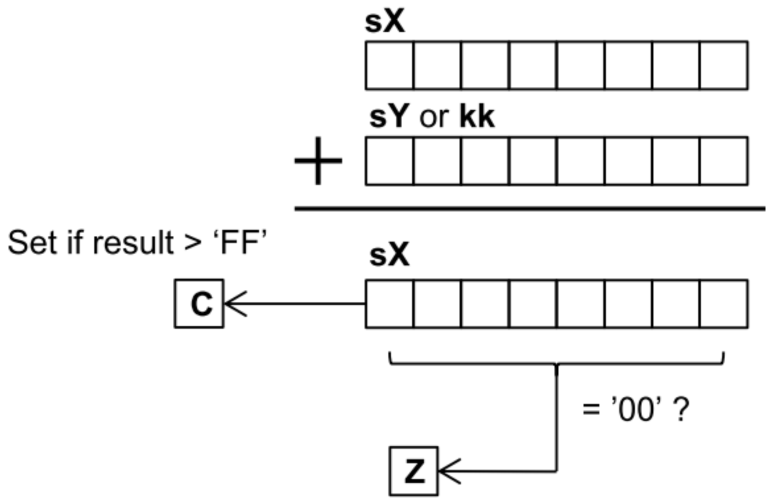
# Arithmetic and Logic instructions

3 operands or 2 operands?
In many processors these instructions are three operand instructions. Example:
add destination, source_1, source_2
However to fit into the 18 bits of the instruction, picoblaze designer has made the source_1 the destination also.

18 = 6 + 4 + 8

**ADD sX, kk** sX = sX + kk

**ADD sX, sY** sX = sX + sY



**Logical**

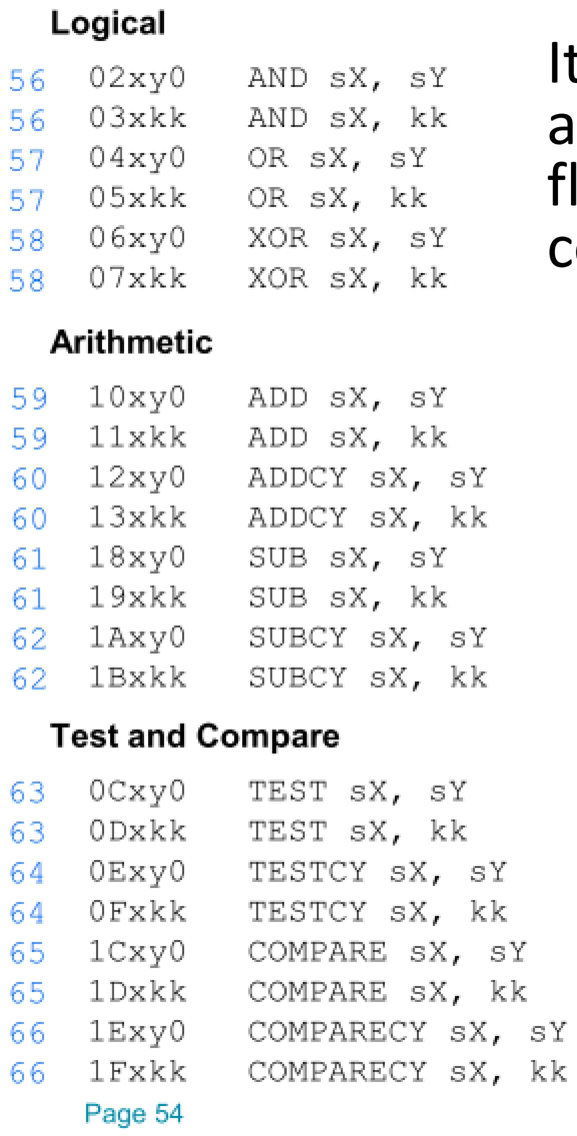
| | | |
|---|---|---|
| 56 | 02xy0 | AND sX, sY |
| 56 | 03xkk | AND sX, kk |
| 57 | 04xy0 | OR sX, sY |
| 57 | 05xkk | OR sX, kk |
| 58 | 06xy0 | XOR sX, sY |
| 58 | 07xkk | XOR sX, kk |

**Arithmetic**

| | | |
|---|---|---|
| 59 | 10xy0 | ADD sX, sY |
| 59 | 11xkk | ADD sX, kk |
| 60 | 12xy0 | ADDCY sX, sY |
| 60 | 13xkk | ADDCY sX, kk |
| 61 | 18xy0 | SUB sX, sY |
| 61 | 19xkk | SUB sX, kk |
| 62 | 1Axy0 | SUBCY sX, sY |
| 62 | 1Bxkk | SUBCY sX, kk |

# The Carry (C) and the Zero (Z) flags

It is important to understand how the Arithmetic, Logical, Test, and Compare instructions change the Carry (C) and the Zero (Z) flags and further how the conditional jumps, conditional calls, and conditional returns utilize the Carry (C) and the Zero (Z) flags.



**Logical**

| | | |
|---|---|---|
| 56 | 02xy0 | AND sX, sY |
| 56 | 03xkk | AND sX, kk |
| 57 | 04xy0 | OR sX, sY |
| 57 | 05xkk | OR sX, kk |
| 58 | 06xy0 | XOR sX, sY |
| 58 | 07xkk | XOR sX, kk |

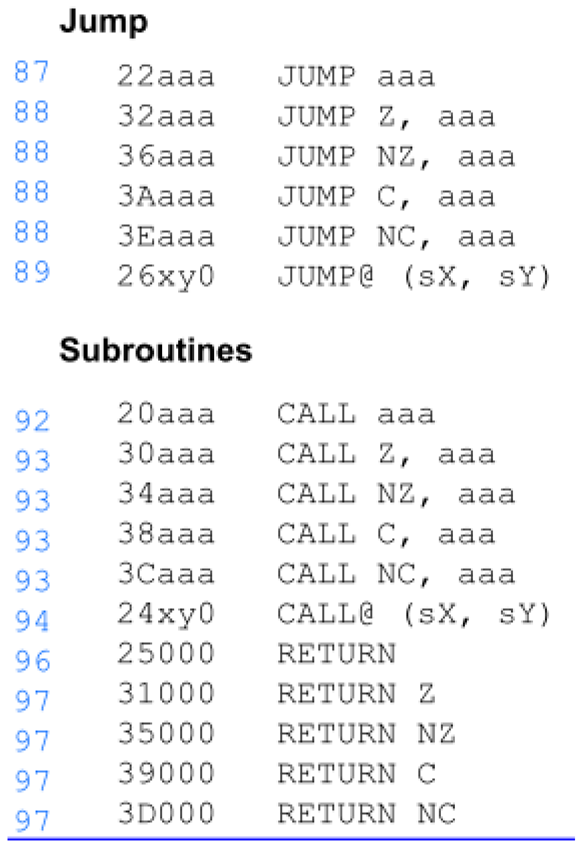
**Arithmetic**

| | | |
|---|---|---|
| 59 | 10xy0 | ADD sX, sY |
| 59 | 11xkk | ADD sX, kk |
| 60 | 12xy0 | ADDCY sX, sY |
| 60 | 13xkk | ADDCY sX, kk |
| 61 | 18xy0 | SUB sX, sY |
| 61 | 19xkk | SUB sX, kk |
| 62 | 1Axy0 | SUBCY sX, sY |
| 62 | 1Bxkk | SUBCY sX, kk |

**Test and Compare**

| | | |
|---|---|---|
| 63 | 0Cxy0 | TEST sX, sY |
| 63 | 0Dxkk | TEST sX, kk |
| 64 | 0Exy0 | TESTCY sX, sY |
| 64 | 0Fxkk | TESTCY sX, kk |
| 65 | 1Cxy0 | COMPARE sX, sY |
| 65 | 1Dxkk | COMPARE sX, kk |
| 66 | 1Exy0 | COMPARECY sX, sY |
| 66 | 1Fxkk | COMPARECY sX, kk |

Page 54



**Jump**

| | | |
|---|---|---|
| 87 | 22aaa | JUMP aaa |
| 88 | 32aaa | JUMP Z, aaa |
| 88 | 36aaa | JUMP NZ, aaa |
| 88 | 3Aaaa | JUMP C, aaa |
| 88 | 3Eaaa | JUMP NC, aaa |
| 89 | 26xy0 | JUMP@ (sX, sY) |

**Subroutines**

| | | |
|---|---|---|
| 92 | 20aaa | CALL aaa |
| 93 | 30aaa | CALL Z, aaa |
| 93 | 34aaa | CALL NZ, aaa |
| 93 | 38aaa | CALL C, aaa |
| 93 | 3Caaa | CALL NC, aaa |
| 94 | 24xy0 | CALL@ (sX, sY) |
| 96 | 25000 | RETURN |
| 97 | 31000 | RETURN Z |
| 97 | 35000 | RETURN NZ |
| 97 | 39000 | RETURN C |
| 97 | 3D000 | RETURN NC |

# Carry, Borrow, and Odd Parity

Even though the name of the flag is “Carry”, its meaning changes depending on the context.

Logical operations (AND, OR, and XOR) will reset the carry to zero.
The ADD operation will set the carry if the out-going carry C8 is true (otherwise will reset it).

The COMPARE operation performs comparison by performing subtraction. The COMPARE and SUBtract operations will set the carry if the subtrahend is bigger (otherwise will reset the carry). So the carry flag can be viewed as representing the out-going borrow in subtraction.

In the case of TEST instructions, AND operation is performed to see if the result is zero or if the result has odd number of 1’s as indicated by the carry flag. So the carry flag can be viewed as an Odd Parity of the result 8 bits in the case of the TEST (and an Odd Parity of the result 8 bits plus one incoming carry bit in the case of the TESTCY).

# Multi-precision arithmetic/logical operations

Processors are categorized as 8-bit or 16-bit or 32-bit or 64-bit processors based on their internal data bus width. A 32-bit processor for example, has a 32-bit ALU and can add the contents of two 32-bit registers in one stroke. However, even an 8-bit processor can perform 32-bit addition by splitting it into 4 8-bit additions and using ADDCY for the last 3 add operations to include the carry from the previous 3 operations.

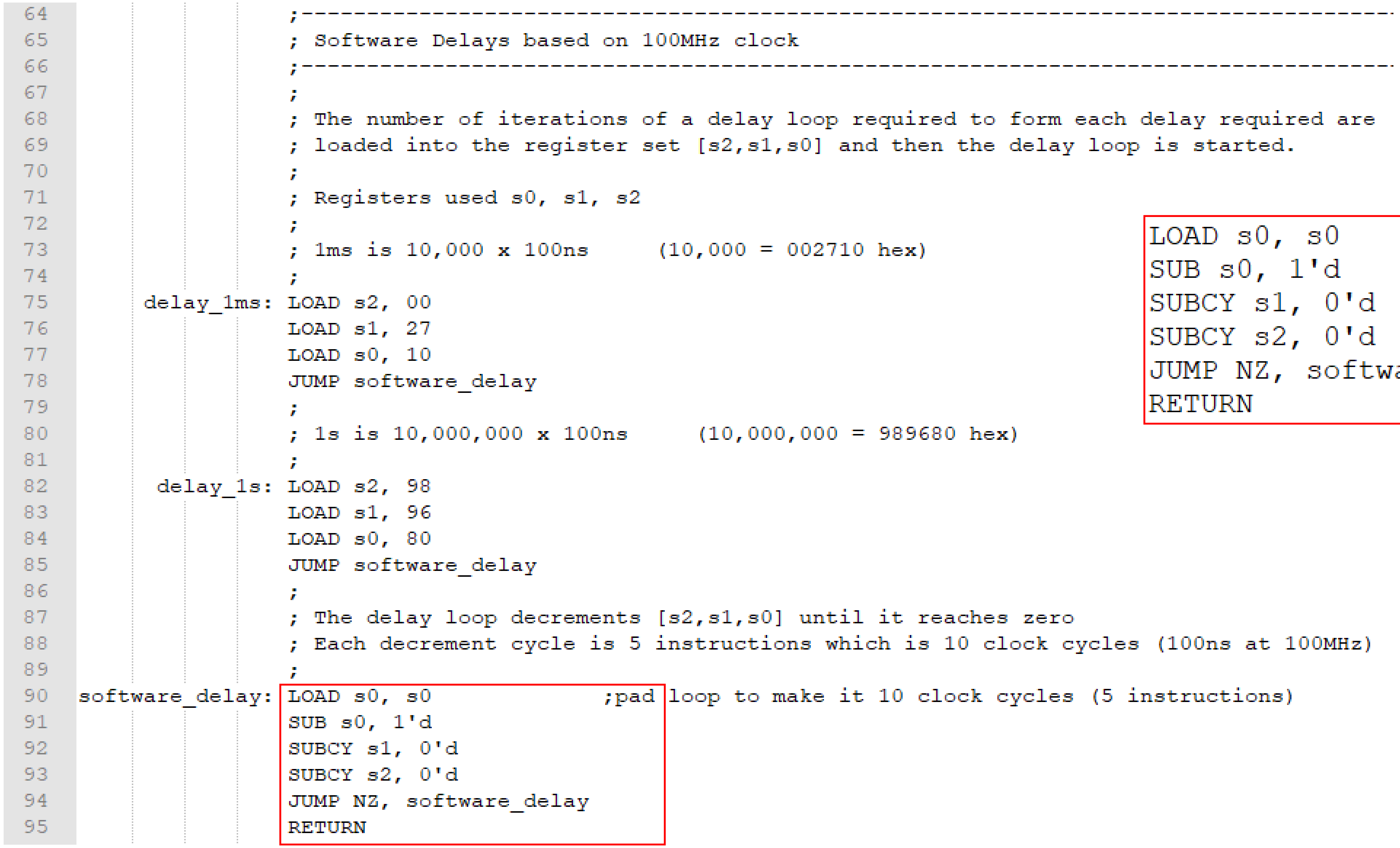
```
ADD   s0, 1'd
ADDCY s1, 0'd
ADDCY s2, 0'd
ADDCY s3, 0'd
```

Shown on the side is a 32-bit incrementation and a 32-bit decrementation code using ADD/ADDCY and SUB/SUBCY instructions. Assume that the four registers, s3, s2, s1, s0 contain the 32-bit value with s3 containing the most significant byte and the s0 containing the least significant byte.

```
SUB   s0, 1'd
SUBCY s1, 0'd
SUBCY s2, 0'd
SUBCY s3, 0'd
```

We may need to create a one-millisecond delay loop routine by counting instructions (thereby clocks). The 32-bit incrementor and the 32-bit decrementer routines come handy.

# Software Delays based on 100MHz clock



```
                   ;------------------------------------------------------------------------------
                   ; Software Delays based on 100MHz clock
                   ;------------------------------------------------------------------------------
                   ;
                   ; The number of iterations of a delay loop required to form each delay required are
                   ; loaded into the register set [s2,s1,s0] and then the delay loop is started.
                   ;
                   ; Registers used s0, s1, s2
                   ;
                   ; 1ms is 10,000 x 100ns     (10,000 = 002710 hex)
                   ;
        delay_1ms: LOAD s2, 00
                   LOAD s1, 27
                   LOAD s0, 10
                   JUMP software_delay
                   ;
                   ; 1s is 10,000,000 x 100ns     (10,000,000 = 989680 hex)
                   ;
         delay_1s: LOAD s2, 98
                   LOAD s1, 96
                   LOAD s0, 80
                   JUMP software_delay
                   ;
                   ; The delay loop decrements [s2,s1,s0] until it reaches zero
                   ; Each decrement cycle is 5 instructions which is 10 clock cycles (100ns at 100MHz)
                   ;
   software_delay: LOAD s0, s0             ;pad loop to make it 10 clock cycles (5 instructions)
                   SUB s0, 1'd
                   SUBCY s1, 0'd
                   SUBCY s2, 0'd
                   JUMP NZ, software_delay
                   RETURN
```

```
LOAD s0, s0                ;pad
SUB s0, 1'd
SUBCY s1, 0'd
SUBCY s2, 0'd
JUMP NZ, software_delay
RETURN
```

## Subroutine calls and returns from subroutines

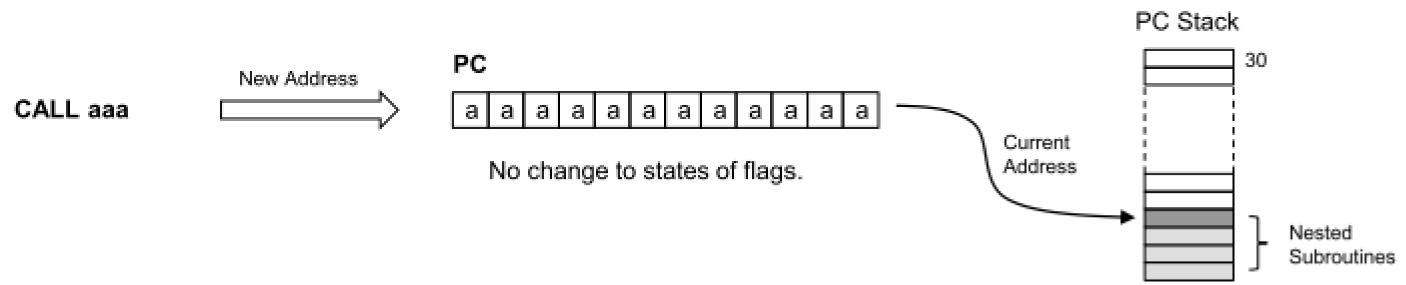
Picoblaze supports subroutine calls and returns from subroutines.
It has a 30-deep PC stack. Nested subroutine calls are supported.

```
40      CALL aaa
41      LOAD sA, 00'd
```

In the above, the CALL instruction is at 40 and the return address (the address of the instruction in calling program to which we need to return) is 41. Some processors increment 40 to 41 as part of the execution of the CALL instruction and save 41 on the stack. Then the RETURN instruction retrieves the return address from the stack and causes return to the main program. However in the picoblaze, the CALL instruction address 40 is saved on the stack and the RETUN instruction retrieves 40, increments it to 41 and then causes return to 41.

## CALL aaa

'CALL aaa' is an unconditional CALL to a subroutine which pushes the current contents of the program counter (PC) onto the stack and loads the PC with the address defined by the value 'aaa'. A subroutine should end with a 'RETURN' instruction which will pop the last pushed address off of the stack, increment it and load it back into the program counter such that the program then executes the instruction following the initial CALL. Please also see the description of 'JUMP aaa' regarding the valid range of 'aaa' values and how the assembler is typically used to resolve their values for you.



No change to states of flags.

Whist the PC Stack is completely dedicated and automatic you are entirely responsible for making sure that for each CALL made to a subroutine you have a *corresponding* RETURN. You must also ensure that execution of your program does not exceed 30 'nested' subroutines but this limit is rarely challenged by typical programs. Remember that an interrupt is a special case equivalent to a call and will use one level. If the stack does overflow then KCPSM6 will automatically reset.

Example  Within some code a CALL is made to a subroutine called 'inc_count' which contains a 12-instruction procedure that increments a 32-bit number stored in 4 bytes of scratch pad memory. The corresponding RETURN at the end of the subroutine allows the program to continue.

```
AND s0, 01
OUTPUT s0, status
CALL inc_count32
LOAD s0, 38
JUMP main_loop
```

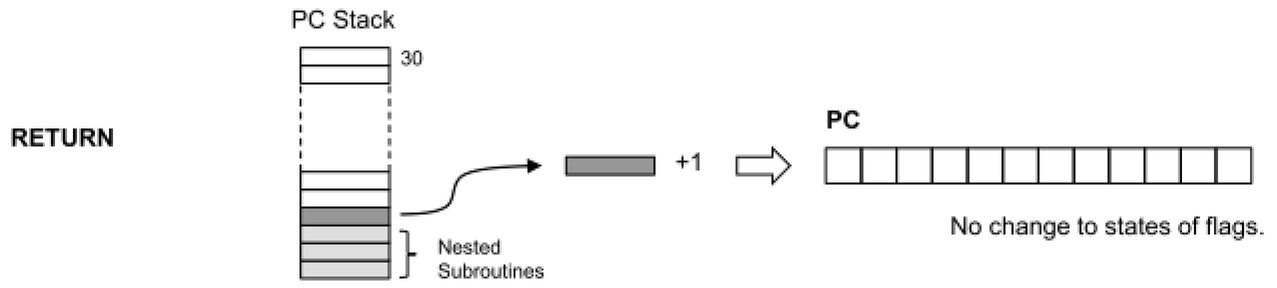
```
inc_count32: FETCH s0, count0
             FETCH s1, count1
             FETCH s2, count2
             FETCH s3, count3
             ADD s0, 1'd
             ADDCY s1, 00
             ADDCY s2, 00
             ADDCY s3, 00
             STORE s0, count0
             STORE s1, count1
             STORE s2, count2
             STORE s3, count3
             RETURN
```

Hint – A subroutine can be located anywhere in a program relative to the CALL instructions that invoke it but it is vital that the subroutine is only executed as the result of a CALL otherwise their will be no address in the PC stack to correspond with the subsequent RETURN instruction.

XILINX.

## RETURN

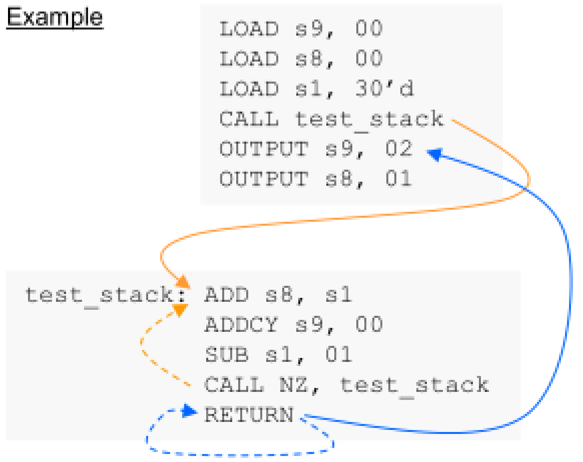
The 'RETURN' instruction is used to unconditionally complete a subroutine. The last address pushed on to the PC Stack by the previous call to the subroutine is popped off the stack, incremented and loaded into the program counter. This automatic process ensures that the return is made to the address following the CALL instruction that initiated the subroutine.



No change to states of flags.

Whist the PC Stack is completely dedicated and automatic you are entirely responsible for making sure that each RETURN is only executed to complete a subroutine that was invoked by the *corresponding* call instruction. If your code should incorrectly execute a RETURN that results in stack underflow then KCPSM6 will automatically reset. Remember that an interrupt is a special case equivalent to a call and requires a corresponding RETURNI instruction.

Example



```
                LOAD s9, 00
                LOAD s8, 00
                LOAD s1, 30'd
                CALL test_stack
                OUTPUT s9, 02
                OUTPUT s8, 01


test_stack: ADD s8, s1
            ADDCY s9, 00
            SUB s1, 01
            CALL NZ, test_stack
            RETURN
```

This example illustrates the general arrangement in which one part of the program calls a subroutine. In most cases line labels are used to make the code easier to write and maintain and the assembler resolves the actual addresses.

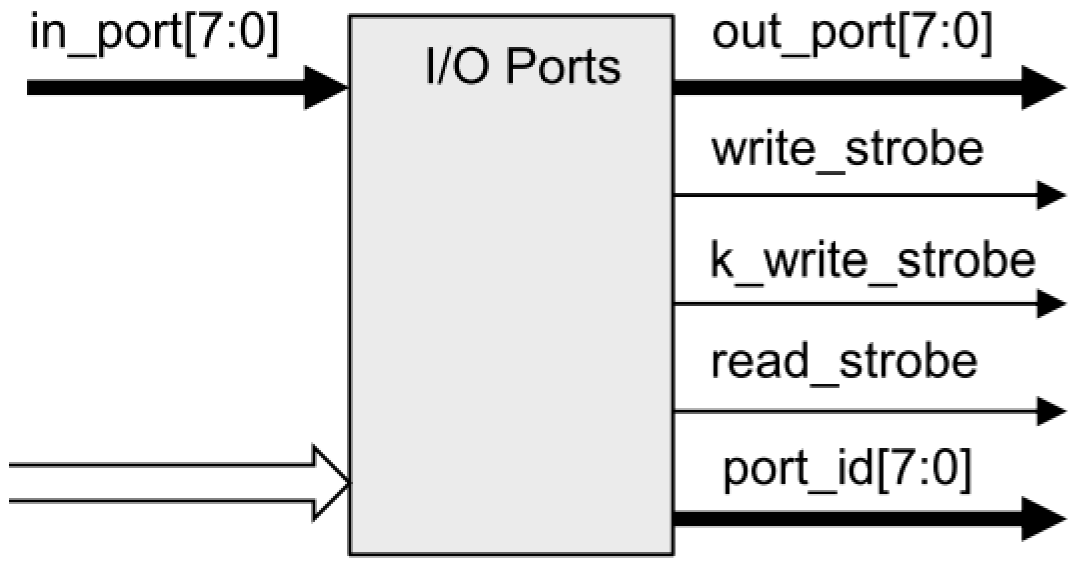
The subroutine labelled 'test_stack' is called from the main program. When this subroutine completes the RETURN forces the program counter to the address corresponding with the instruction immediately following the CALL which in this case is an OUTPUT instruction.

Whilst this example does show the general arrangement it actually describes a rather special case when we look at the code in detail. In the main program [s9,s8] has been cleared and then 's1' has been loaded with 30 decimal. The 'test_stack' subrouine adds the value of 's1' to [s9,s8] and then decrements the value in 's1'. But each time 's1' is not zero it actually calls 'test_stack' again. Hence this subroutine is called 30 times and eventually [s9,s8] will be the sum of all values from 1 to 30 which is 465 (01D1 hex). When 's1' does reach zero, KCPSM6 will execute the RETURN instruction 30 times until it eventually returns to the main program. Hence there is no restriction on how subroutines are arranged providing you do not exceed 30 levels and every CALL has a corresponding RETURN.

## Interface with the Fabric Logic via Input ports and Output ports

# Interface with the Fabric Logic via Input and Output ports

Data from input port
For example from 8 Switches



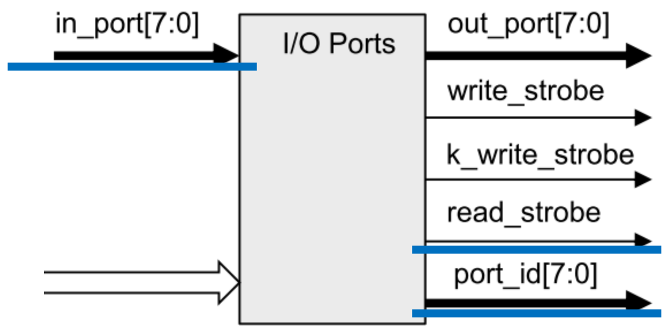
Data to output port
For example to 8 LEDs

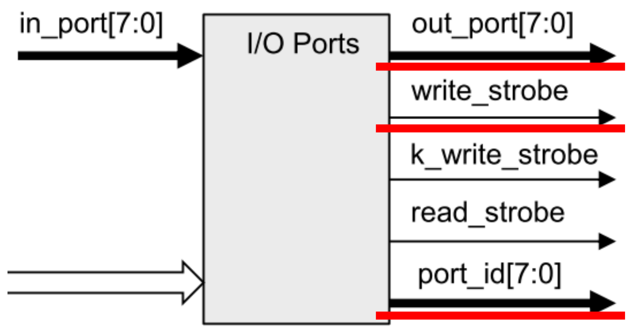
Address of the port
Input port or Output port

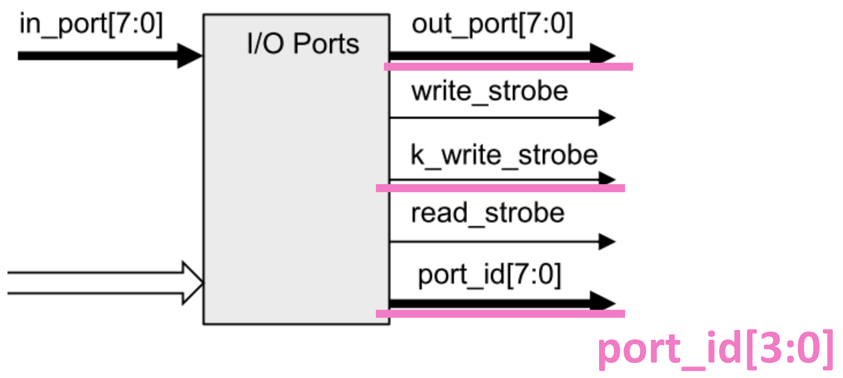
Using the 8-bit port_id[7:0], we can generate 256 port addresses and talk to them (exchange data with them).

So, all together do we have 256 I/O ports or do we have 256 input ports and 256 output ports?

# Interface with the Fabric Logic via Input and Output ports







**256 Input ports (Read only)**

**8-bit port_id qualified by read_strobe produces 256 IDSPs (Input Device Select Pulses)**

**256 output ports (Write only)**

**8-bit port_id qualified by write_strobe produces 256 ODSPs (Output Device Select Pulses)**

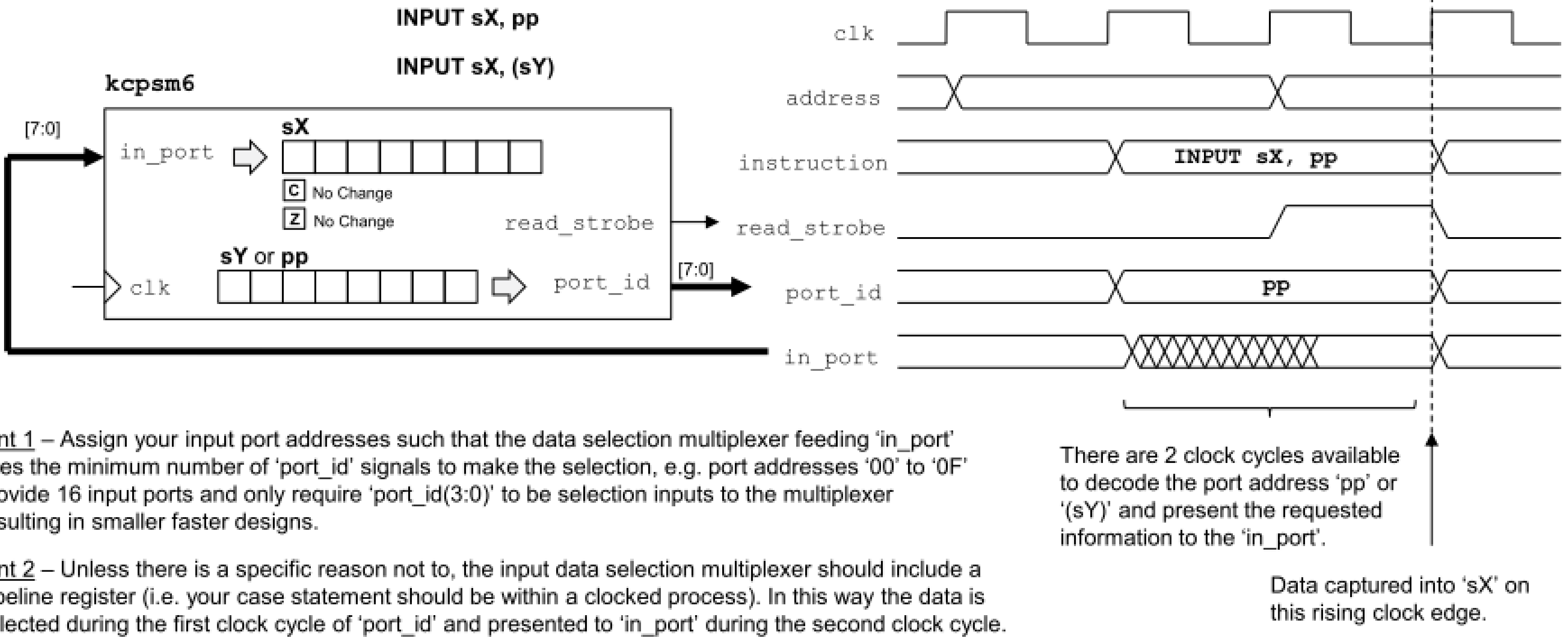
**16 output ports (Write only)**

**4-bit port_id qualified by k_write_strobe produces 16 k_ODSPs (k_Output Device Select Pulses)**

## INPUT sX, pp
## INPUT sX, (sY)

**IMPORTANT**

An 'INPUT' instruction enables KCPSM6 to read information ~~from the~~ from your hardware design into a register 'sX' using a general purpose input port specified by an 8-bit constant value 'pp' or the contents of another register '(sY)'. KCPSM6 presents the port address defined by 'pp' or '(sY)' on 'port_id' and your hardware interface is then responsible for selecting and presenting the appropriate information to the 'in_port' so that it can be captured into the 'sX' register. An active High ('1') synchronous pulse is also generated on the 'read_strobe' pin and may be used by the hardware interface to confirm when a particular port has been read.


INPUT sX, pp
INPUT sX, (sY)
kcpsm6
[7:0]
in_port
sX
C No Change
Z No Change
read_strobe
sY or pp
clk
port_id
[7:0]
clk
address
instruction
INPUT sX, pp
read_strobe
port_id
PP
in_port
There are 2 clock cycles available to decode the port address 'pp' or '(sY)' and present the requested information to the 'in_port'.
Data captured into 'sX' on this rising clock edge.

Hint 1 – Assign your input port addresses such that the data selection multiplexer feeding 'in_port' uses the minimum number of 'port_id' signals to make the selection, e.g. port addresses '00' to '0F' provide 16 input ports and only require 'port_id(3:0)' to be selection inputs to the multiplexer resulting in smaller faster designs.

Hint 2 – Unless there is a specific reason not to, the input data selection multiplexer should include a pipeline register (i.e. your case statement should be within a clocked process). In this way the data is selected during the first clock cycle of 'port_id' and presented to 'in_port' during the second clock cycle. Failure to define a pipeline register anywhere in the 'port_id' to 'in_port' path is the most common reason for PicoBlaze designs failing to meet the required performance (a 'false path' for one clock cycle) .

Hint 3 – 'read_strobe' can be ignored in most cases and never needs to be part of the multiplexer feeding 'in_port'. However, some functions such as a FIFO buffer do need to know when they have been read and it is in those situations that 'read_strobe' together with a decode of the appropriate value of 'port_id' would be used to generate a "port has been read" pulse to confirm when a read has taken place.

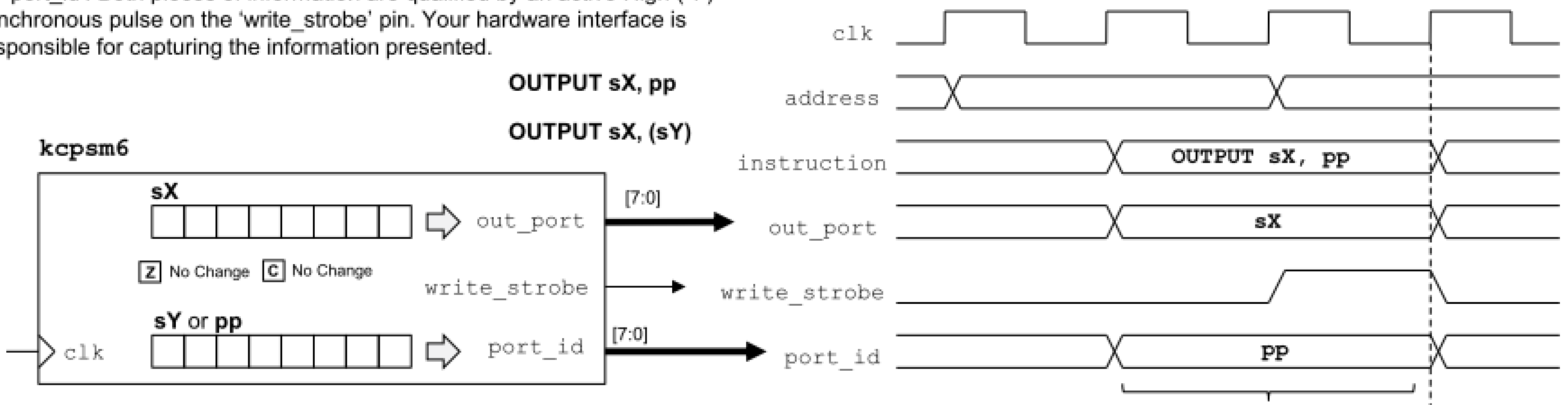
## OUTPUT sX, pp
## OUTPUT sX, (sY)

**IMPORTANT**

An 'OUTPUT' instruction is used to transfer information from a register 'sX' to a general purpose output port specified by an 8-bit constant value 'pp' or the contents of another register '(sY)'. KCPSM6 presents the contents of the register 'sX' on 'out_port' and the port address defined by 'pp' or '(sY)' is presented on 'port_id'. Both pieces of information are qualified by an active High ('1') synchronous pulse on the 'write_strobe' pin. Your hardware interface is responsible for capturing the information presented.

**OUTPUT sX, pp**

**OUTPUT sX, (sY)**



There are 2 clock cycle available to decode the port address 'pp' or '(sY)'

Note that 'out_port' and 'port_id' will vary during the execution of other instructions but 'write_strobe' will only be active during an OUTPUT instruction.

The value presented on 'out_port' should be captured on the rising edge of the clock when 'write_strobe' is High.

Hint – In most cases a fixed port address 'pp' is used so CONSTANT directives provide an ideal way to why track your port assignments and make your code easier to write, understand and maintain.

Examples

```
CONSTANT LED_port, 05
;
LOAD s3, 3A
OUTPUT s3, LED_port
```

If you want to keep your designs small and fast then assign port addresses that facilitate smaller logic functions.

In this example a set of 8 LEDs are mapped to port 05 hex and only 3-bits of 'port_id' together with 'write_strobe' are decoded.

VHDL

```
~~if clk'event and clk = '1' then~~
~~  if write_strobe = '1' then~~
~~    if port_id(2 downto 0) = "101" then~~
~~      led <= out_port;~~
~~    end if;~~
~~  end if;~~
~~end if;~~
```

```
OUTPUT s6, (s2)
OUTPUT s4, 40
OUTPUT sB, 64'd
```

Decimal values can be used to specify port addresses but hex or binary values are normally easier to work with when defining the hardware.

XILINX

## Input Ports IMPORTANT

KCPSM6 can read 8-bit values from up to 256 general purpose input ports using its 'INPUT sX, pp' and 'INPUT sX, (sY)' instructions. A complete description is provided in the reference section later in this document but here we can see this put into practice so that KCPSM6 can read the 8 DIP switches and 5 'direction' push buttons on the ML605 board.

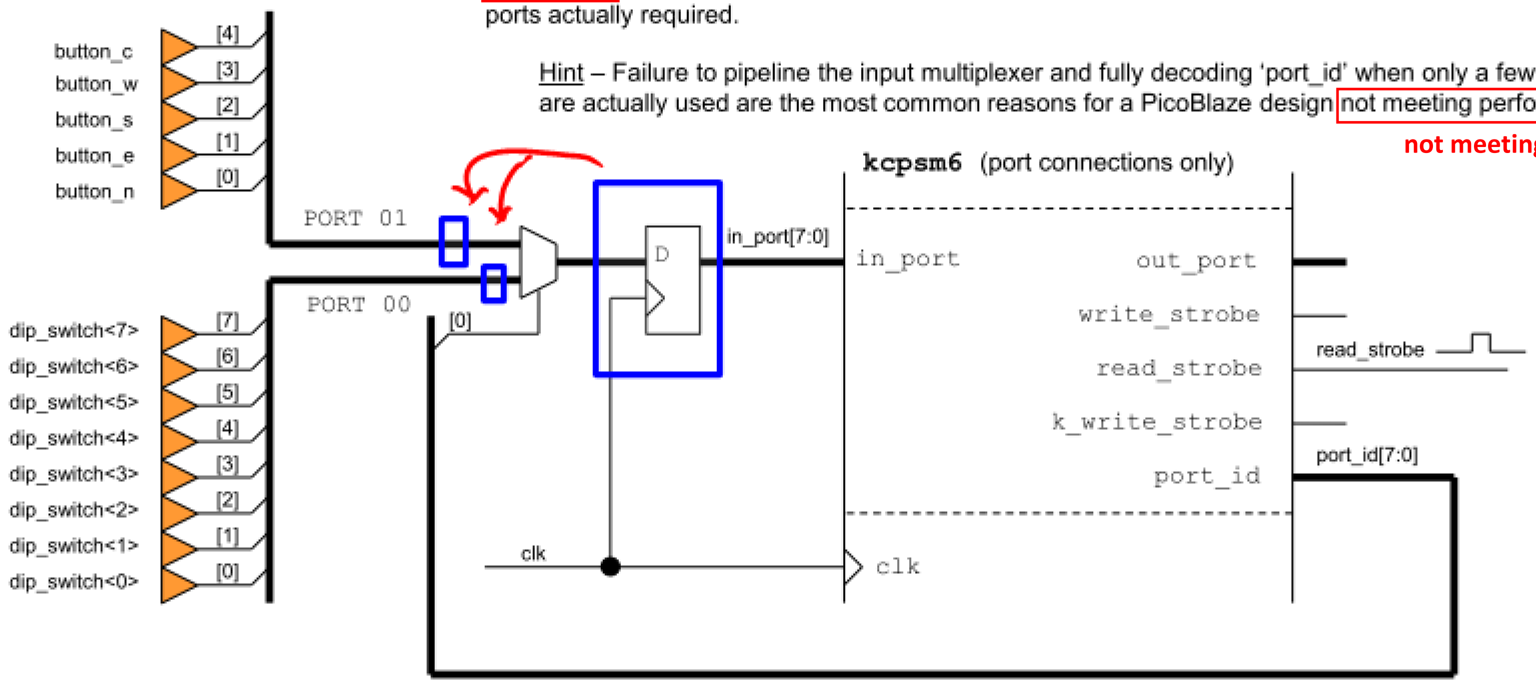
Please see next page for the corresponding VHDL and PSM

When KCPSM6 executes an 'INPUT' instruction it sets 'port_id' to specify which of 256 ports it wants to read from and it is the task of the hardware circuit to present that 8-bit data to the 'in_port'. A simple multiplexer can be used to achieve this. It is best practice to pipeline the output of the multiplexer and to only use the appropriate number of bits of 'port_id' to facilitate the number of input ports actually required.

Hint – Failure to pipeline the input multiplexer and fully decoding 'port_id' when only a few input ports are actually used are the most common reasons for a PicoBlaze design not meeting performance.

**not meeting clock period.**

In a non-FPGA based design such as an ASIC or a custom VLSI, it is common practice to use tristate buffers in place of a mux. You want to enable a tri-state buffer only after confirming that the address is stabilized and is not changing. This is to make sure that multiple tristate buffers are not enabled unintentionally for a short length of time during address transitions. Hence, I have moved the pipeline register to the upstream of the mux. In my design, there is a register for every input port, and it captures data on every clock. The data read by the processor is one clock delayed and in most cases that is not harmful (particularly in our application with manual operation of switches and buttons.



Hint - The 'read_strobe' is also pulsed High when KCPSM6 executes an 'INPUT' instruction but this does not need to be used to qualify the multiplexer selection. This strobe would be used in situations where the circuit being read needs to know when data has been captured. The most obvious example is reading data from a FIFO so that it can discard the oldest information and present the information to be read on its output.

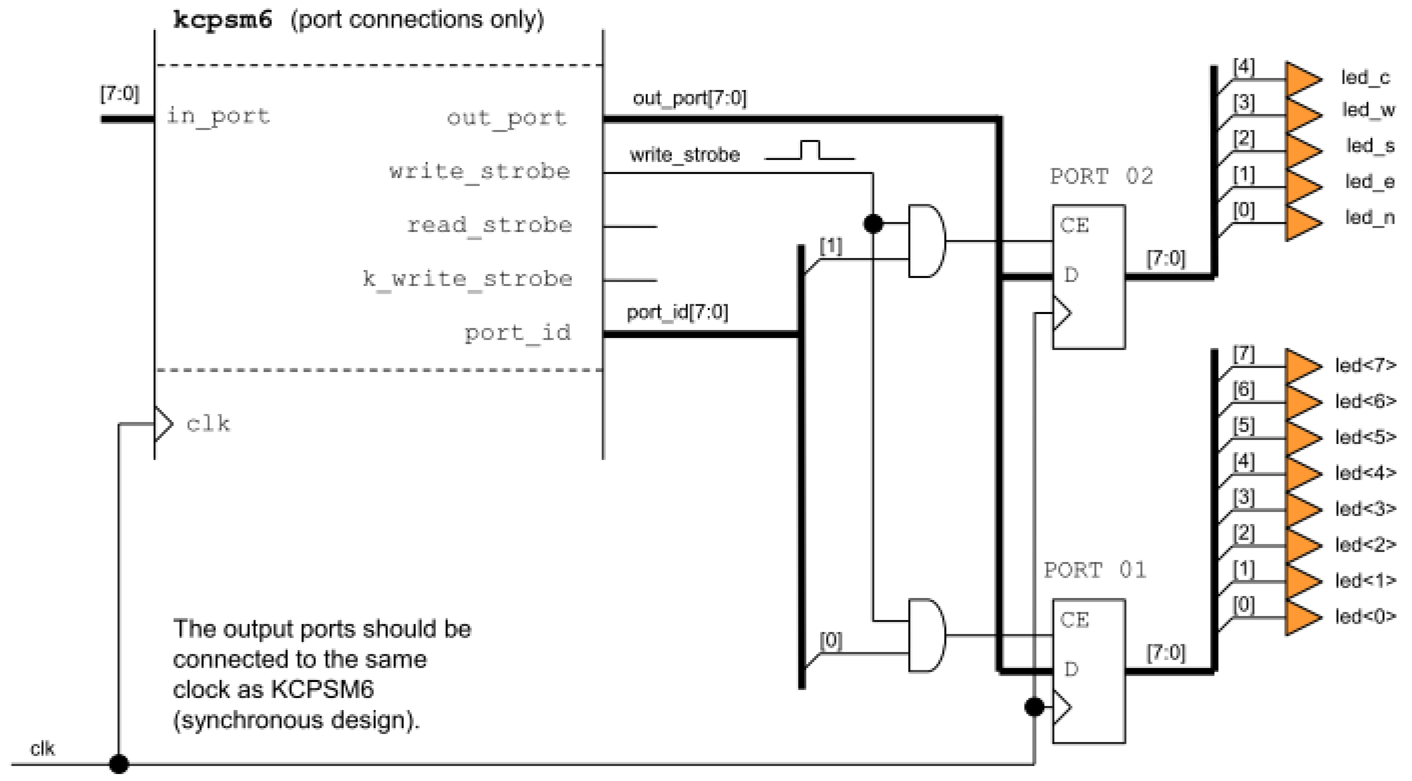
XILINX

## Output Ports IMPORTANT

KCPSM6 can output 8-bit values to up to 256 general purpose output ports using its 'OUTPUT sX, pp' and 'OUTPUT sX, (sY)' instructions. A complete description is provided in the reference section later in this document but here we can see this put into practice so that KCPSM6 can control the 8 general purpose LEDs and 5 'direction' LEDs on the ML605 board.

Please see next page for the corresponding VHDL and PSM

When KCPSM6 executes an 'OUTPUT' instruction it sets 'port_id' to specify which of 256 ports it wants to write the 8-bit data value present on 'out_port'. A single clock cycle enable pulse is generated on 'write_strobe' and **your hardware must use 'write_strobe' to qualify the decodes of 'port_id'** to ensure that only the correct register (or peripheral) captures the 'out_port' value.

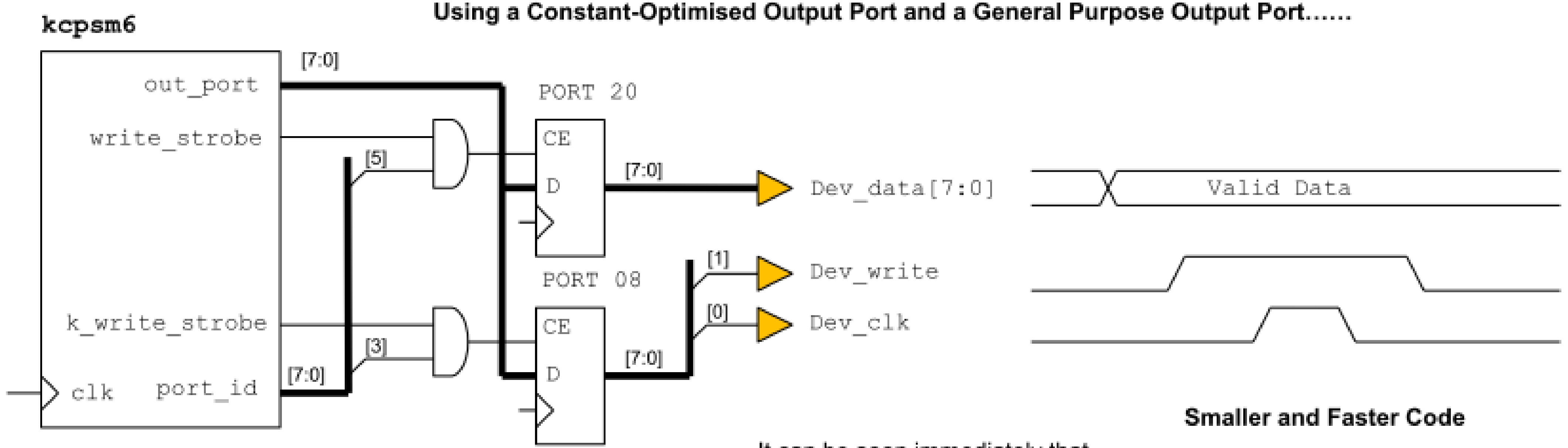


The output ports should be connected to the same clock as KCPSM6 (synchronous design).

# Constant-Optimised Output Ports

Returning to the same example of writing data to an external device we can see that port 08 hex has now been allocated to a constant-optimised output port by using the 'k_write_strobe' whilst port 20 hex is still associated with 'write_strobe' because the data is naturally variable. So there is very little difference in the hardware as long as you remember that only port_id[3:0] are defined during an OUTPUTK instruction. Note also that you could now have two different output ports with the same address; on for variable data and the other for constant values (see page 79).

**Using a Constant-Optimised Output Port and a General Purpose Output Port……**



```
CONSTANT Dev_data_port, 20
CONSTANT Dev_control_port, 08
```

It can be seen immediately that all the LOAD instructions have been eliminated saving code space and reducing the execution time. This also means that register 's0' used previously to define the sequence of values is now free for another purpose.

**Smaller and Faster Code**

```
OUTPUT s1, Dev_data_port
OUTPUTK 00000010'b, Dev_control_port
OUTPUTK 00000011'b, Dev_control_port
OUTPUTK 00000010'b, Dev_control_port
OUTPUTK 00000000'b, Dev_control_port
```

## Interrupts, Interrupt Vector, and ISR

Please see the next 5 pages.
One observation, we wish to make, is that the RETURNI (pronounced Return-I, I for Interrupt) does not increment the address retrieved from the stack. So Picoblaze, on recognizing that there is an interrupt request, would save the return address on the stack. This is slightly deferent from the CALL instruction execution. So the stack should not be called the CALL address stack or RETUN address stack. Perhaps that is why the Picoblaze user guide calls it a PC Stack!

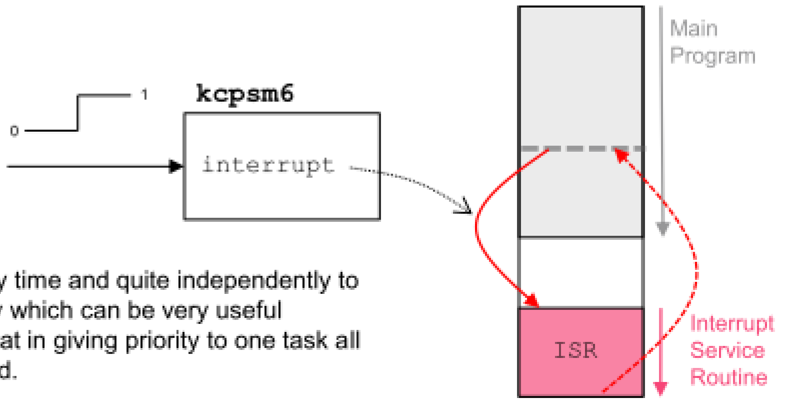
Please go through the discussion about where to locate the ISR and how to specify the interrupt vector parameter while instantiating the picoblaze processor in the top design, and also how to use the assembler directive ADDRESS to specify the location of the ISR routine to the assembler.

# Interrupts

Interrupts can be extremely useful so KCPSM6 provides an 'interrupt' input pin, an 'interrupt_ack' output pin, an optional 'interrrupt_vector' generic and three interrupt related instructions. However, it would be fair to say that interrupts are quite an advanced technique and require understanding, thought and preparation to be used wisely and successfully. This subject is made more interesting because each KCPSM6 is fully embedded into your FPGA design meaning that you have the option to define hardware dedicated to servicing tasks in a way that simply isn't available when using a standard microcontroller device. In fact, many PicoBlaze users have discovered that because each PicoBlaze is so small and efficient, it is often beneficial to use multiple instances within the same design in order that each is dedicated to a particular task and therefore avoiding the requirements for interrupts altogether. So it is well worth considering what an interrupt actually does and when it provides greatest benefit in a KCPSM6 design.

## What does an interrupt do?

To state the obvious, an interrupt is used to interrupt the normal program execution sequence of KCPSM6. This means that when the 'interrupt' input is driven High ('1'), it will force KCPSM6 to abandon the code that it is executing, save its current operational state and divert its attention to executing a special section of program code known as an Interrupt Service Routine (ISR). Once the interrupt has been serviced, KCPSM6 returns to the program at the point from which it was interrupted and restores the operational states so that it can resume execution of the program as if nothing had happened.

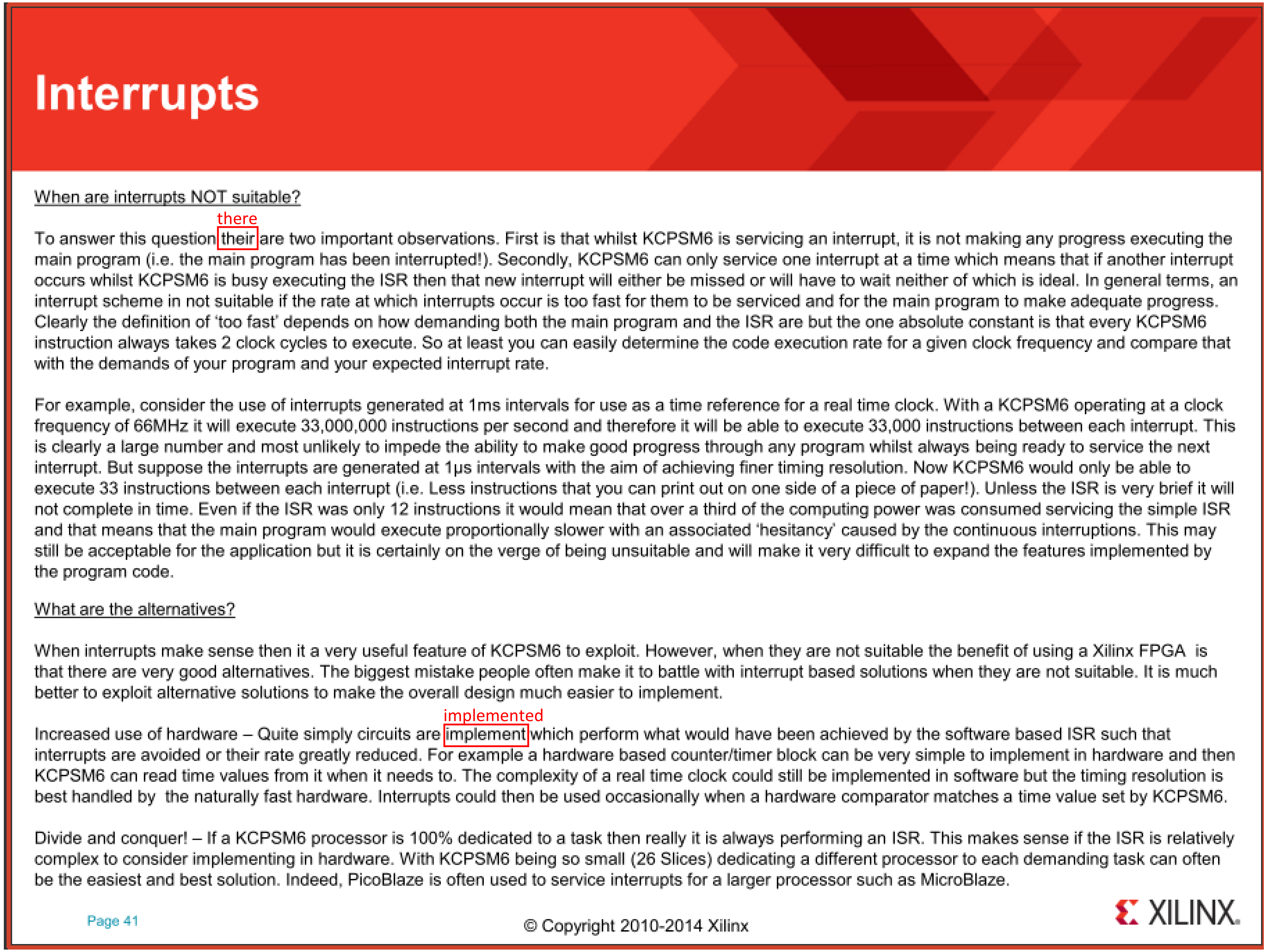


Hence the interrupt mechanism provides a way for KCPSM6 to react to an event at any time and quite independently to the main tasks being performed. In other words an interrupt is given the highest priority which can be very useful particularly when reacting to a critical situation. However, it must also be recognised that in giving priority to one task all other tasks can be interrupted and hence their execution rates can be erratic or delayed.

## When do interrupts make sense?

The key observation is that an interrupt has the highest priority. So clearly the most obvious application for an interrupt is to react quickly to system critical or emergency situations. These may be such rare events that they may never happen in normal operation such as the detection of a fire and the need to activate the water sprinklers. More common situations are less of an emergency but important to system integrity with a good example being the requirement to react to a FIFO buffer becoming full so that data is read from it before it actually overflows and data is lost.

Another application involves a regular or semi-regular steam of interrupts to KCPSM6 which become a fundamental part of the way in which the program normally operates. For example a hardware counter could easily generate an interrupt every milli-second which KCPSM6 uses as the reference for an accurate  real time clock. The main program possibly enabling that clock to be set, displayed and for controlling the times at which appliances must be turned on and off. Alternatively KCPSM6 may use each interrupt as the trigger to perform a sequence of tasks but do virtually nothing else whilst waiting.



## Interrupts

### When are interrupts NOT suitable?

To answer this question $\overset{\text{there}}{\boxed{\text{their}}}$ are two important observations. First is that whilst KCPSM6 is servicing an interrupt, it is not making any progress executing the main program (i.e. the main program has been interrupted!). Secondly, KCPSM6 can only service one interrupt at a time which means that if another interrupt occurs whilst KCPSM6 is busy executing the ISR then that new interrupt will either be missed or will have to wait neither of which is ideal. In general terms, an interrupt scheme in not suitable if the rate at which interrupts occur is too fast for them to be serviced and for the main program to make adequate progress. Clearly the definition of 'too fast' depends on how demanding both the main program and the ISR are but the one absolute constant is that every KCPSM6 instruction always takes 2 clock cycles to execute. So at least you can easily determine the code execution rate for a given clock frequency and compare that with the demands of your program and your expected interrupt rate.

For example, consider the use of interrupts generated at 1ms intervals for use as a time reference for a real time clock. With a KCPSM6 operating at a clock frequency of 66MHz it will execute 33,000,000 instructions per second and therefore it will be able to execute 33,000 instructions between each interrupt. This is clearly a large number and most unlikely to impede the ability to make good progress through any program whilst always being ready to service the next interrupt. But suppose the interrupts are generated at 1µs intervals with the aim of achieving finer timing resolution. Now KCPSM6 would only be able to execute 33 instructions between each interrupt (i.e. Less instructions that you can print out on one side of a piece of paper!). Unless the ISR is very brief it will not complete in time. Even if the ISR was only 12 instructions it would mean that over a third of the computing power was consumed servicing the simple ISR and that means that the main program would execute proportionally slower with an associated 'hesitancy' caused by the continuous interruptions. This may still be acceptable for the application but it is certainly on the verge of being unsuitable and will make it very difficult to expand the features implemented by the program code.

### What are the alternatives?

When interrupts make sense then it a very useful feature of KCPSM6 to exploit. However, when they are not suitable the benefit of using a Xilinx FPGA is that there are very good alternatives. The biggest mistake people often make it to battle with interrupt based solutions when they are not suitable. It is much better to exploit alternative solutions to make the overall design much easier to implement.

Increased use of hardware – Quite simply circuits are $\overset{\text{implemented}}{\boxed{\text{implement}}}$ which perform what would have been achieved by the software based ISR such that interrupts are avoided or their rate greatly reduced. For example a hardware based counter/timer block can be very simple to implement in hardware and then KCPSM6 can read time values from it when it needs to. The complexity of a real time clock could still be implemented in software but the timing resolution is best handled by the naturally fast hardware. Interrupts could then be used occasionally when a hardware comparator matches a time value set by KCPSM6.

Divide and conquer! – If a KCPSM6 processor is 100% dedicated to a task then really it is always performing an ISR. This makes sense if the ISR is relatively complex to consider implementing in hardware. With KCPSM6 being so small (26 Slices) dedicating a different processor to each demanding task can often be the easiest and best solution. Indeed, PicoBlaze is often used to service interrupts for a larger processor such as MicroBlaze.

XILINX.

# ‘interrupt_vector’ and ‘ADDRESS’ Directive

See pages 83-85 for interrupt related instructions

When KCPSM6 responds to an interrupt it executes the equivalent of a CALL instruction as well as the interrupt specific tasks such as preserving the states of the flags. The interrupt vector is the address that KCPSM6 effectively calls and it has the default value of 3FF hex. However, this can be set to any value within the range of the program memory available in your design using the ‘interrupt_vector’ generic in your HDL design description.

```
processor: kcpsm6
  generic map (                  hwbuild => X"00",
                        interrupt_vector => X"3FF",
                 scratch_pad_memory_size => 64)
  port map(                      address => address,
                             instruction => instruction,
Etc...
```

Component declaration (part of) showing the default values of the three generics.

**VHDL**

```
processor: kcpsm6
  generic map (                  hwbuild => X"41",
                        interrupt_vector => X"F80",
                 scratch_pad_memory_size => 256)
  port map(       address => address,
              instruction => instruction,
Etc...
```

Component Instantiation (part of) showing that the interrupt vector has been set to ‘F80’ hex.

PSM file...

```
      ADDRESS F80
      ;
ISR : ADD sF, 1'd
      RETURNI ENABLE
```

**ADDRESS Directive**

Use the ADDRESS directive in your PSM code to force the ISR to be assembled starting at the same address as the interrupt vector.

What is a good address for ‘interrupt_vector’?

3FF is the last location in a 1K program memory and is consistent with KCPSM, KCPSM-II and KCPSM3. So for direct compatibility with legacy PicoBlaze programs this is the best address to start with and hence the reason why it is the default. Of course you could modify the program and vector.

Generally the most convenient address is somewhere *close to the end* of the program memory available but leaving enough space for the ISR. This means that the ISR can begin servicing the interrupt immediately. It is also convenient from a programming perspective because the ADDRESS directive must be used to align the start of the ISR code with interrupt vector and having this as the last section of your PSM program allows your main program the flexibility to expand up to it. As your code becomes stable you can always fine tune your matching ‘interrupt_vector’ and ADDRESS directive for best memory fit.

What are bad values? If you try to put your ISR somewhere in the middle of your program then you will probably find that you are always having to adjust the ADDRESS directive and ‘interrupt_vector’ which is just an inconvenient waste of time as well as error prone. The absolute worst address would be zero! Under no circumstances would you want your ISR to execute on power up or following a reset (RETURNI should only be used following an interrupt).

XILINX.

is for us

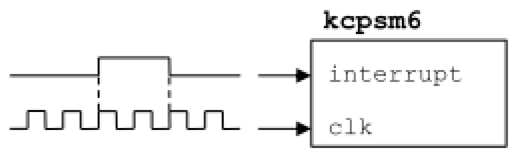
# Hardware arrangements for KCPSM6 Interrupt

See pages 83-85 for interrupt related instructions

The KCPSM6 processor has two pins dedicated to interrupts; an 'interrupt' input and an 'interrupt_ack' output. To initiate an interrupt the 'interrupt' input must be driven High and the fundamental interrupt response time is just 3 or 4 clock cycles. As shown on the next page (Interrupt Waveforms) the interrupt input is sampled once every two clock cycles consistent with the instruction execution rate. For this reason it is vital that the interrupt input is High at the right time to be observed by KCPSM6 and the easiest way to achieve that us to drive the interrupt input High for longer than one clock cycle. There are two fundamental schemes that can be used which can really be describes as being 'open-loop' and 'closed loop'.

'Open-Loop' interrupt pulse

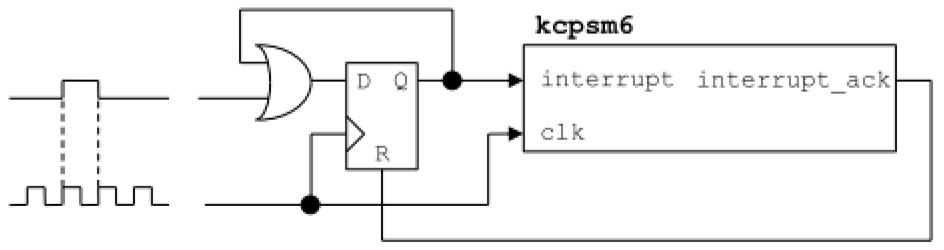
The simplest way of initiating an interrupt is to generate an active High pulse that has a duration of 2 clock cycles. The pulse can be longer but should have returned Low before the ISR completes otherwise KCPSM6 will immediate think there is another interrupt to service (remember that each instruction executes in 2 clock cycles so some ISR's may not take very many clock cycles). Once KCPSM6 observes the High level on its interrupt input it will abandon the next instruction and immediately move to the ISR.



The simplicity of the 'open-loop' method is obvious but it must also be recognised that any open loop system has its limitations. In this case there is the potential for KCPSM6 to miss an interrupt request and therefore fail to service it. This could happen if the KCPSM6 program has deliberately disabled interrupts or is already servicing a previous ISR. KCPSM6 will also ignore the interrupt input whilst held in sleep mode. Therefore this technique should only be used if you can predict that KCPSM6 will always be ready to respond to an interrupt request or if it is acceptable for interrupts to be missed.

'Closed-Loop' interrupt (recommended)

In this scheme your design drives the interrupt signal High to request an interrupt and then keeps driving it High until KCPSM6 generates an 'interrupt_ack' pulse confirming that it has seen it. This ensures that the interrupt will always be observed by KCPSM6 when it is able to. If interrupts have been temporarily disabled deliberately, or whilst servicing a previous interrupt, then the response will be delayed but the event can not be missed. Likewise, if KCPSM6 is held in sleep mode when the interrupt is requested it will remain active until KCPSM6 is allowed to wake up and observe it.



Hint – Some systems can require a more comprehensive closed-loop arrangement in which KCPSM6 would be expected to indicate when the ISR has completed rather than just started (which is what 'interrupt_ack' signifies). This can be achieved using an output port with associated 'OUTPUT' or 'OUTPUTK' instructions at the end of your ISR. Alternatively you could detect when instruction[17:12] = "101001" corresponding with the 'RETURNI' instruction being fetched from the program memory.

XILINX.

# Interrupt Waveforms

See pages 83-85 for interrupt related instructions

An interrupt is performed when the 'interrupt' input is driven High, interrupts have been enabled by the program and KCPSM6 is not in sleep mode or otherwise busy servicing a previous interrupt. When KCPSM6 detects an interrupt it forces the next instruction to be abandoned, preserves the current states of the 'Z' and 'C' flags, notes the current bank selection ('A' or 'B') and then forces the program counter to the interrupt vector (default value is 3FF hex which is the last location of a 1K program memory but can be set to any value using the 'interrupt_vector' generic).

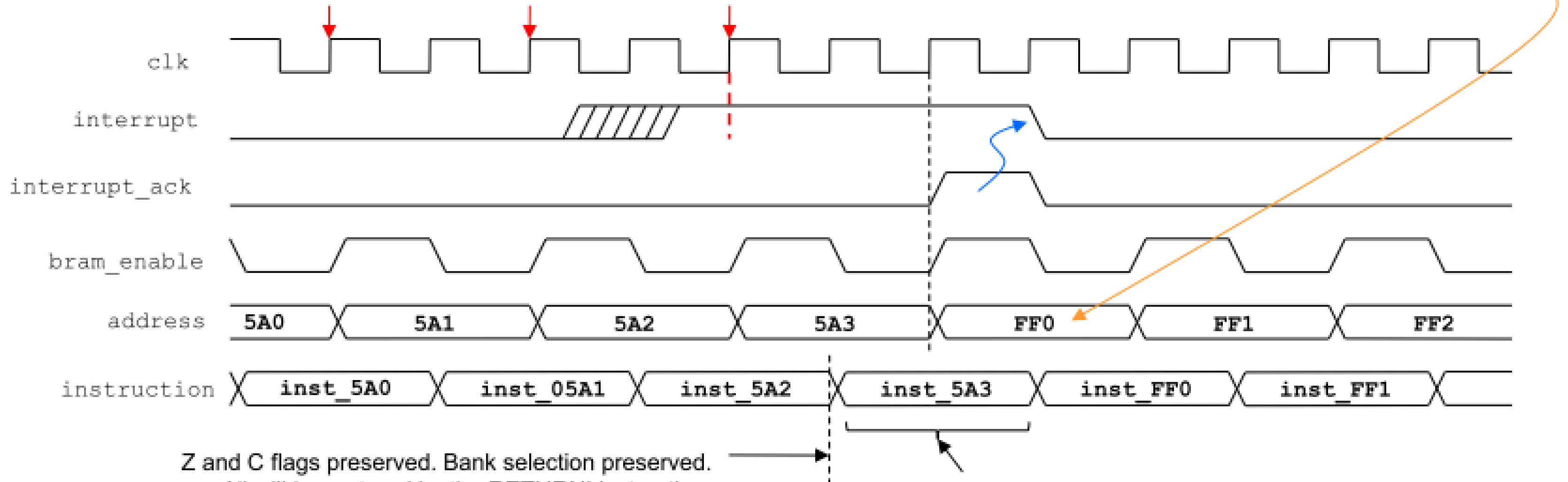
The waveforms shown below illustrate a normal response to an interrupt when interrupts have been enabled within the program and KCPSM6 is ready to respond. In the hardware design the interrupt vector was set to FF0 hex and a 'closed-loop' interrupt scheme used (implemented by the VHDL shown on the right) to ensure that the interrupt pulse can not be missed.

VHDL

```
interrupt_control: process(clk)
  begin
    if clk'event and clk = '1' then
      if interrupt_ack = '1' then
        interrupt <= '0';
       else
        if kcpsm6_interrupt = '1' then
          interrupt <= '1';
         else
          interrupt <= interrupt;
        end if;
      end if;
    end if;
end process interrupt_control;
```

```
interrupt_vector => X"FF0",
```

The 'interrupt' input is sampled on the rising clock edges that the address



Z and C flags preserved. Bank selection preserved.
All will be restored by the RETURNI instruction.

The last instruction read from program memory before the interrupt takes place is abandoned. This will be the first instruction executed following a RETURNI after the interrupt has been serviced.

XILINX.