

03/23/09

**VHDL / FPGA Design
Lecture Notes**

Agenda

- Parameterization of VHDL components
- Hardware replication using “generate” statements
- Soft-Core Microprocessors for hardware / software system-on-an-FPGA

Parameterization

- Ashenden ... Ch 12. Rushton 10.10
- Enhance the reusability and flexibility of our VHDL code blocks by making certain qualities about them *parameters* that can be easily changed.
- keyword : generic
- Parameter is set at instantiation, not in the part itself (though a default value can be set)

Instantiating parameterized components

```
entity regn is
generic (n: positive:=2);
port (
  clk : in std_logic;
  d : in std_logic_vector(n-1 downto 0);
  enable : in std_logic;
  q : out std_logic_vector(n-1 downto 0)
);
end regn;
```

```
architecture behavior of regn is
begin
  my_reg : process(clk)
  begin
    if rising_edge(clk) then
      if enable = '1' then
        q <= d;
      end if;
    end if;
  end process my_reg;
end behavior;
```

MYREG16 : regn generic map (n => 16)

port map (clk,d,enable,q);

MYREG2 : regn port map (clk,d,enable,dataout)

default value for generic is used unless otherwise specified

Another example – a counter

```
entity clkdivider is
    generic (divideby : natural := 2);
    Port ( clk : in std_logic;
           resetn : in std_logic;
           clkout : out std_logic);
end clkdivider;
```

Creates a 1 clock wide pulse to be used as an enable for a unit that needs to be enabled once every “divideby” clocks. If divideby isn’t specified, it will become a divide-by-two entity

Architecture (first write without generic)

```
architecture Behavioral of clkdivider is
signal cnt : natural range 0 to 3;
begin
process(clk,resetn)
begin
    if resetn='0' then
        cnt<=0;
    elsif rising_edge(clk) then
        if (cnt = 3) then
            cnt <= 0;
        else
            cnt <= cnt+1;
        end if;
    end if;
end process;
clkout <= '1' when cnt=3 else '0';
end Behavioral;
```

Architecture (with generic)

```
architecture Behavioral of clkdivider is
signal cnt : natural range 0 to divideby-1;
begin

process(clk,resetn)
begin
    if resetn='0' then
        cnt<=0;
    elsif rising_edgeE(clk) then
        if (cnt = divideby-1) then
            cnt <= 0;
        else
            cnt <= cnt+1;
        end if;
    end if;
end process;
clkout <= '1' when cnt=divideby-1 else '0';
end Behavioral;
```

Convenient to Use

Now, whenever we want a clock divider to make those 1-clk wide pulses (to be used as enables) at certain rates, we just do this :

```
make2ms : clkdivider
  generic map (divideby => 100000)
  port map (clk=> clk50,resetn => rstn,clkout =>en_2ms);
```

```
Make1ms : clkdivider
  generic map (divideby => 50000)
  port map (clk=> clk50,resetn => rstn,clkout =>en_1ms);
```

Complex Heirarchy

Review :

3 constructs in the VHDL concurrent domain :

CSA : $a \leq a$ and b;

Process

Component Instantiation

U5: myblock port map (a,b,c);

All these “are” pieces of hardware, which operate in parallel (I.e. concurrently)

The component instantiation is the main tool used for heirarchy :
Top-down design: break the design up into components (entity / architecture pairs)

Complex Heirarchy

Many designs are highly parallel, and thus can fit a model where the same component is used multiple times.

```
Entity patternrecognizer_3ch is
  port (ch1_in, ch2_in, ch3_in : in std_logic_vector(7 downto 0);
        found1, found2, found3 : out std_logic);
end patternrecognizer_3ch;
```

architecture structure of patternrecognizer is ...

...

```
CHANNEL1 : patrec port map (in => ch1_in, found => found1);
CHANNEL2 : patrec port map (in => ch2_in, found => found2);
CHANNEL3 : patrec port map (in => ch3_in, found => found3);
```

Generate Statements

For large numbers of instantiations, in highly parallel designs, the **generate** statement can construct this for you.

generate construct lives at architecture level – has syntax similar to a **for loop**. (for loop exists only in process, and is by nature sequential code)

```
gen : for I in 0 to 7 generate  
    recognizers: patrec port map (ch_in(I),found(I));  
end generate;
```

These are *concurrent* statements – think of the generate as only saving you typing.

Generate Statements

```
gen : for I in 0 to 7 generate  
    recognizers: patrec port map (ch_in(I),found(I));  
end generate;
```

Order of execution is irrelevant (there is no order to the concurrent statements, thus the range direction of the loop can not make any difference.

Note the requirement to place the interconnection signals as arrays, so that they can be indexed.

Must have a constant range

This **for generate** loop with component instantiation is the most widely used application of the generate (and is the one in most books)

If-Generate Statements

The means to conditionally include certain “pieces of hardware”, is also provided. Consider the case where based on the value of a generic : “use_revised”, we instantiate different variations of our pattern recognizer.

```
gen : if use_revised generate
    rec: revised_patrec port map (ch_in(I),found(I));
end generate;

gen2 : if not use_revised generate
    rec: patrec port map (ch_in(I),found(I));
end generate;
```

the condition for the if – in this case, use_revised, must be of type boolean.

Note : there is no **else** generate, which is why we have the slightly awkward construct above, in order to handle both cases.

Generate flexibility

As the **generate** lives at architecture level, it can be used to generate any of the 3 things that also live at the architecture level.

Use generate to generate CSAs, processes, or component instantiations

```
entity adder_ripple is
  generic( n : natural);
  port (a,b : in std_logic_vector(n-1 downto 0);
        cin : in std_logic;
        cout : out std_logic;
        sum : out std_logic_vector(n-1 downto 0));
end adder_ripple
.....
signal c : std_logic_vector(n downto 0);
c(0) <= cin;
gen : for I in 0 to n-1 generate
  sum(I) <= a(I) xor b(I) xor c(I);
  c(I+1) <= (a(I) and b(I)) or
    (a(I) and c(I)) or
    (b(I) and c(I));
end generate;
cout <= c(n);
```

Generate flexibility

```
entity optional_register is
  generic( n : natural; store : boolean);
  port (d : in std_logic_vector(n-1 downto 0);
        clk : in std_logic;
        q : out std_logic_vector(n-1 downto 0));
end optional_register;
-- a parameterizable n-bit register, which if the store parameter
-- is set to zero, becomes just a straight signal assignment
architecture behavioral of optional_register is
begin
gen: if store generate
  reg: process(clk)
  begin
    if rising_edge(clk) then q <= d;
    end if;
  end process;
end generate;

notgen : if not store generate
  q <= d;
end generate;
```

Processors in FPGA/ASICs

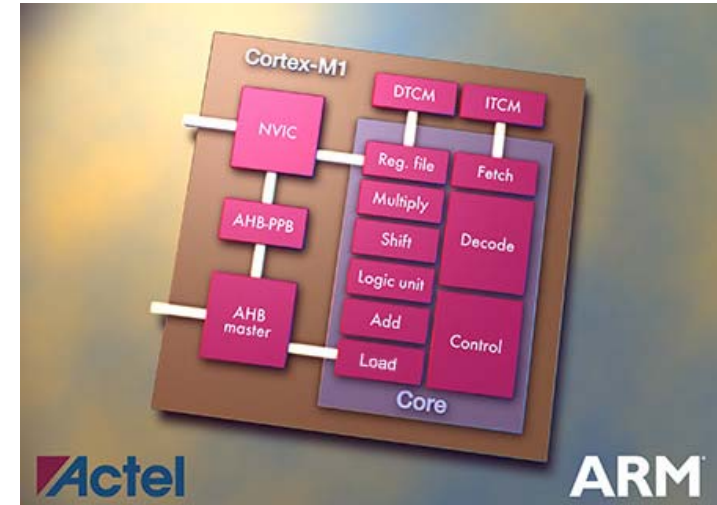
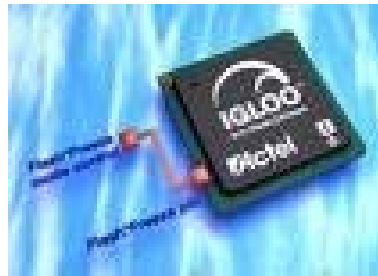
- **Soft Core**
 - A piece of intellectual property (“just” information) which encapsulates the design of a processor.
 - Synthesized, placed, routed just like any design that you’d make
 - Altera NIOS II, Xilinx Microblaze, Xilinx Picoblaze, ARM, endless free and for cost choices.
- **Hard Core**
 - FPGA fabric and traditional microprocessor (i.e., not implemented in FPGA fabric) on the same chip.
 - PowerPC (Xilinx Virtex II Pro, etc.), ARM (in Altera in the past)
 - Generally Higher Performance

Watch the market develop here – Soft Core processor’s are exploding

Uses for soft-core FPGA uC/uP

- “Free” processor for algorithmically complex tasks
 - If the system has an FPGA for other reasons and has space available, the processor comes without cost
- Tighter system integration
 - The more tasks in the system that the FPGA can perform, the fewer chips required
 - Heritage code can be used on the new highly integrated system if a soft core clone of old processor is used
- Custom Processor
 - Processor can be customized (by you) or can be offered by the vendor as highly customizable (in the case of soft cores)

Examples



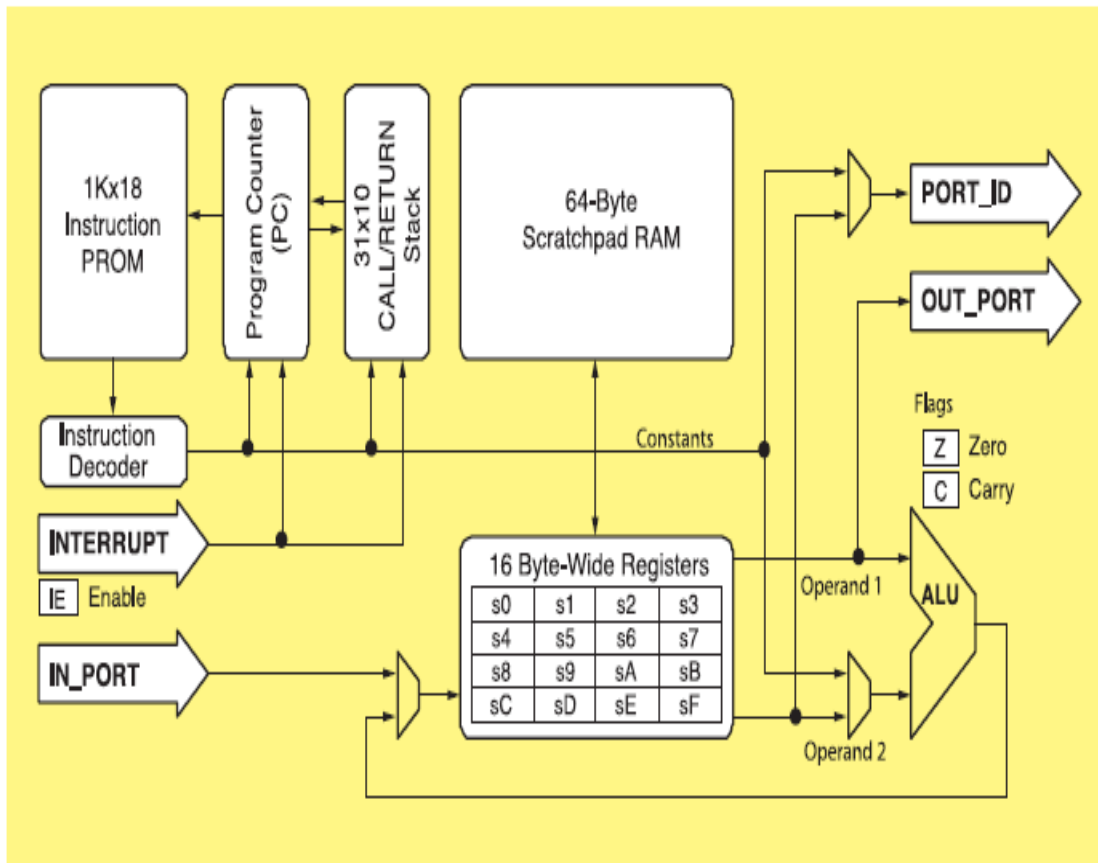
- Xilinx Virtex4 FX:
 - Large SRAM based FPGA with one or more IBM Power-PC Cores on die, along with the rest of the FPGA fabric
- Actel Igloo
 - Flash based FPGA, optimized for very low power operation. Soft Core ARM (Cortex M1) processors are available for use.

PicoBlaze

www.xilinx.com/picoblaze, ug129.pdf

- Soft Core Processor
 - Not behavioral VHDL – manually pre-compiled
 - Built by instantiation of Xilinx raw primitives – still easily simulated with Modelsim
- Size is most important emphasis
 - Only 96 Slices – consider that your XC3S500E “cheap” FPGA has 4656 slices!
- Primary Uses
 - Take the place of a complex state machine
 - Some tasks seem more naturally suited to sequential software
 - Changing software will be faster than changing VHDL
 - Could handle some data processing
- Simple external interface
- Easy to learn instruction set

PicoBlaze Block Diagram*

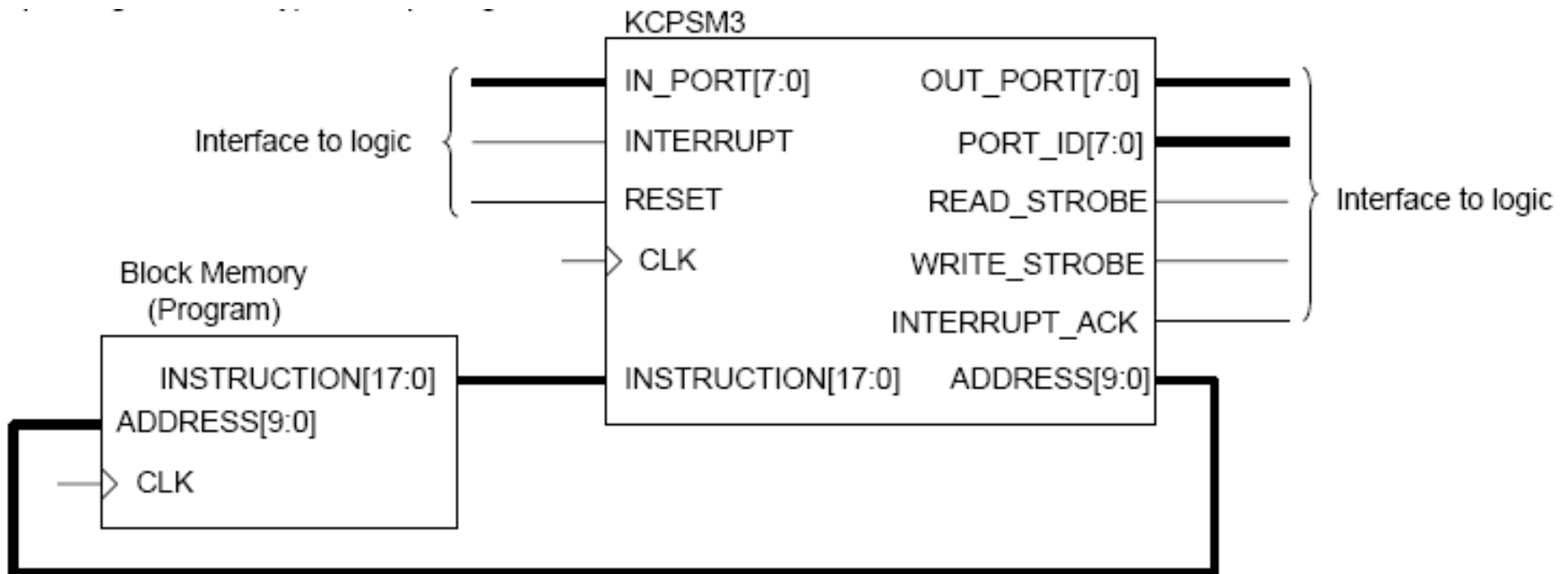


69 MIPS achieved with NO effort on XC3SE

PicoBlaze Performance and Features C

Feature	PicoBlaze for Spartan-3, Virtex-II/Pro and Virtex-4
Program Space	1024
Instruction Size	18-bit
Internal Program	Yes
8-Bit Registers	16
Stack Depth	31
Assembler	KCPSM3
Size	96 Spartan-3 slices
Performance	44 MIPS (Spartan-3) 76 MIPS (Virtex-II) 100 MIPS (Virtex-II Pro) 100 MIPS (Virtex-4 LX, SX) 102 MIPS (Virtex-4 FX)

Inputs/Outputs



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 (kcpsm3.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 kcpasm3
  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: kcpasm3
  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);
```

KCPSM3?

- K-coded programmable state machine
- “State machine based on constants” in several ways
 - Constant data values for ALU
 - Constant port addresses
 - Constant instruction addresses
- Written by Ken Chapman 😊

Some Features

- All instructions take 2 clock cycles
- Programs maximum (without ugly games, described in documentation) 1024 instructions in Spartan III
 - Clearly designed so program memory could fit in 1 BRAM (1kx18)
 - Dictated by size of block ram, and size of address field in opcode structure

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

SRO 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)

Logical, Arithmetic

- Operands are
 - sX and
 - Constant, or...
 - Another register sY
- Result goes in sX

Logical

- LOAD sx, kk
 - Puts constant kk in the Sx register
- AND sx, kk
 - Result in sx
- OR sX, kk
- XOR sX, kk
- All instructions with another S register as well
 - LOAD sx, sy AND sx, sy OR sx, sy XOR sx, sy
 - Result goes in Sx register

Arithmetic

- **ADD**
 - Adds two 8-bit numbers
 - Affects Carry, Zero Flags
- **ADDCY**
 - Same as *ADD*, but uses carry flag as *Cin*
- **SUB, SUBCY**
 - $sX = sX - sY$ (or constant)
 - *C* flag indicates when underflow has occurred

Shifts and Rotates

Rightmost bit goes to C flag

- SR0 sx , SR1 sx
 - Shift right, bringing in '0' or '1'
- SRX sX
 - Shift right, sign extending
- SRA sx
 - Rotate right through carry
- RR sx
 - Rotate right without carry (carry still gets rightmost bit as its shifted)

SL

- SL0 sX
 - SL1 sX
 - SLX sX
 - SLA sX
 - RL sX
-
- Same as SR. For SLX, **LSB gets replicated**

Program Control : JUMP

- JUMP aa
 - Unconditionally jump to address aa (hex)
- JUMP Z, aa
 - Jump if the Z flag is set to address aa
- JUMP NZ, aa
- JUMP C, aa
 - Jump if the C flag is set to address aa
- JUMP NC, aa

Program Control : Call/Return

- CALL (all the same varieties as jump)
 - Jump to address aa and put return address on the stack
 - Use for subroutines
 - Spartan III Picoblaze has a 31-element stack, so you can have subroutine entry up to 31 levels
- RETURN (same conditional varieties)
 - Pops the last pushed PC address from the stack and goes there.

Basics of Interrupt Handling

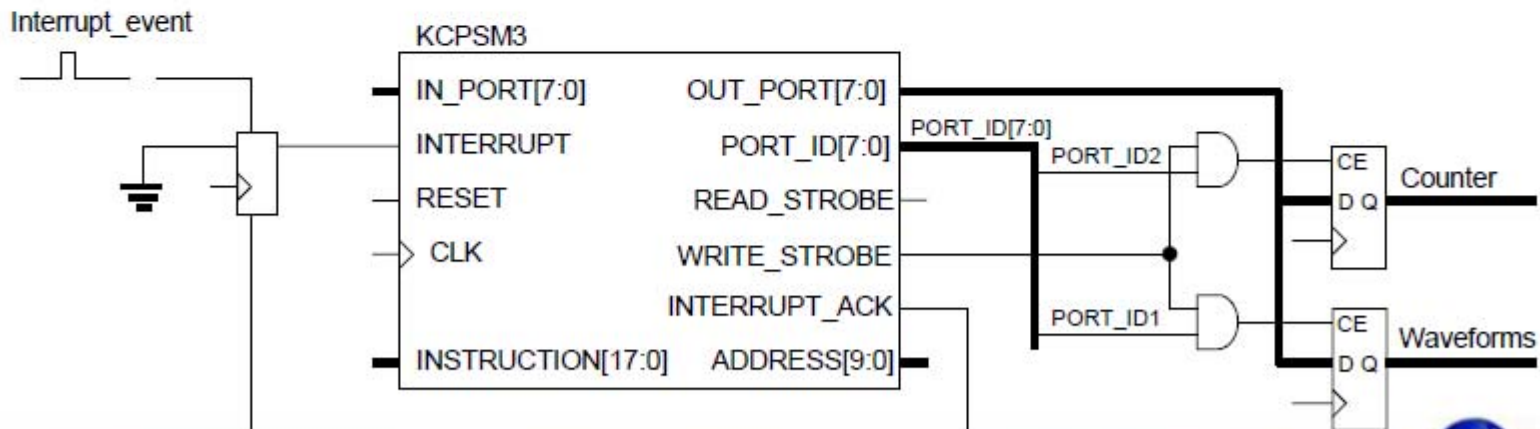
Since the interrupt will force the program counter to address '3FF' it will generally be necessary to ensure that a jump vector to a suitable interrupt service routine (ISR) is located at this address otherwise the program will 'roll over' to address zero.

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'.

```
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;
```

```
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;
```

Interrupt Service Routine

In the assembler log file for the example, it can be seen that the interrupt service routine has been forced 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          ;
009 2B0          ADDRESS 2B0
009 18A01      int_routine: ADD interrupt_counter[sA], 01          ;increment counter
010 2CA04          OUTPUT interrupt_counter[sA], counter_port[04]
011 38001          RETURNI ENABLE
012 2B3          ;
013 3FF          ADDRESS 3FF
014 342B0          JUMP int_routine[2B0]
```

Main program delay loop where most time is spent

Interrupt Service Routine (located at address 2B0 onwards)

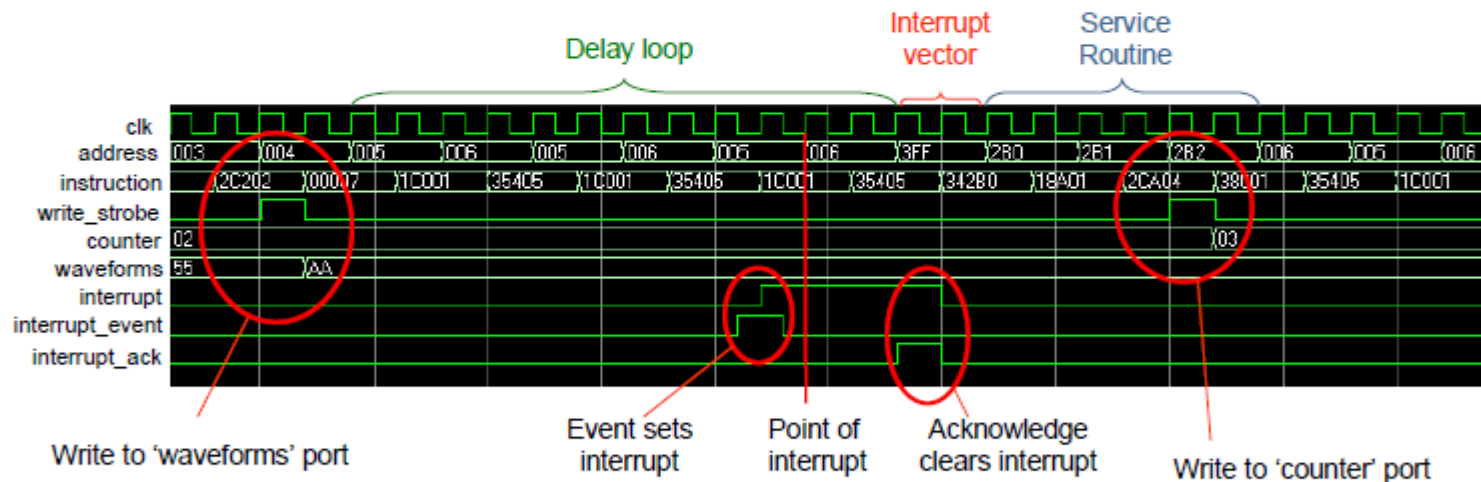
Interrupt vector set at address 3FF and causing JUMP to service routine

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 not 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.

RETURNI

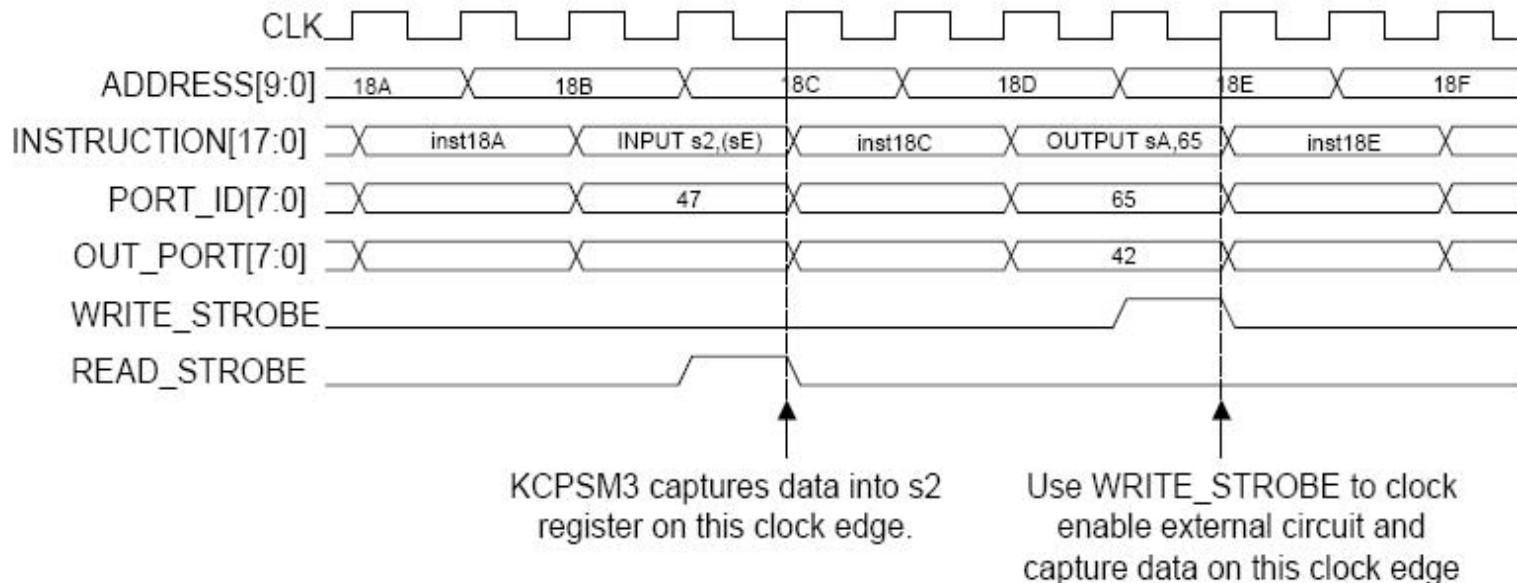
- At the end of the interrupt service routine, code must have a RETURNI instruction so that flags are restored.
 - RETURNI Enable
 - Enables interrupts
 - RETURNI Disable
 - Leaves them off
- Note that interrupts should never be allowed to happen while servicing an interrupt (i.e. there is only a “1-deep stack” for saving the flags)
- Note also that registers are not preserved, so you are responsible for your own run-time model. E.g. if an ISR destroys register contents, the rest of the program must not use those registers.

INPUT / OUTPUT

- INPUT sX , PP (where PP is 8bit address)
 - Takes contents of external port location PP and puts it into sX
 - Details on interfacing to this bus on next slide
- INPUT sX , (sY)
 - Takes contents of external port location specified by the value of sY and puts it in sX
- OUTPUT is the same

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.



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').

FETCH / STORE

- Same syntax as for input and output. Difference is that data is stored and retrieved from the 64 byte internal scratchpad memory. (A good place for variables)
- `FETCH sX, PP` (PP = 6 bit adr)

Thats It?

- Picoblaze has very few instructions
- Leads to perhaps more complex code
- Easier to learn what can and can't be done

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.

Asynchronous ROM

```
entity program is
port ( adr : in std_logic_vector(7 downto 0);
      data : out std_logic_vector(15 downto 0));
end program;
```

```
architecture behavior of program is
begin
  with adr select
    data <= x"1ab0" when x"00"
           x"3c09" when x"01",
           ...etc;
end behavior;
```

One of the things you need to do for every embedded processor situation is figure out where to put the program, and how to get it there. This is an example of the simplest such structure

ROM using initialized Block RAM

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
library unisim;
use unisim.all;
entity blockprogram is
port (clk : in std_logic;
      a  : in std_logic_vector(9 downto 0);
      do : out std_logic_vector(17 downto 0));
end blockprogram;
```

← Xilinx components for instantiation

This is in actuality what the Picoblaze was designed for. The whole Structure of the processor is such that the program memory is to fit In one Spartan III block RAM (1kx18)

Attributes

- Used to retrieve information about types, objects, or other items within a model
- For annotating a model with additional information (not the behavior, and/or structure)

Predefined Attributes

- Used to retrieve information about types, objects, or other items within a model
- Scalar Type Attributes (complete list in Ash. figure 20-1)
 - T'left, T'right, T'low...etc.
- Array type and object Attributes
 - A'left, A'right, A'low, A'high, A'range, A'length ...etc.
- Signals
 - 'delayed, 'event, 'stable...etc

User defined Attributes

- For annotating a model with additional information
 - physical design information
 - pins, cell locations, layout constraints
 - synthesis information
 - encoding for enumerated types
 - resource hints
- Attributes are processed by synthesis / place+route tool
 - some are rather standard, but these depend on synthesis tool
 - usually same functionality can be had via constraints editor, or menu driven options.
 - some users prefer to have all functionality defined in VHDL
 - may make design synthesizer dependent and not as portable.

FSM Attributes Example

```
.  
. .  
--replace the enumerated type with std_logic_vectors  
--      type state_type is (START, STATE1, STATE2, STATE3, STATE4);  
--  
--      -- stores current and next expected state  
--      signal Next_State, Current_State: state_type;  
  
signal Next_State, Current_State : std_logic_vector(2 downto 0);  
constant START   : std_logic_vector(Current_State'range) := "000";  
constant STATE1  : std_logic_vector(Current_State'range) := "001";  
constant STATE2  : std_logic_vector(Current_State'range) := "010";  
constant STATE3  : std_logic_vector(Current_State'range) := "011";  
constant STATE4  : std_logic_vector(Current_State'range) := "100";  
constant ERR1    : std_logic_vector(Current_State'range) := "101";  
constant ERR2    : std_logic_vector(Current_State'range) := "110";  
constant ERR3    : std_logic_vector(Current_State'range) := "111";  
  
attribute fsm_extract : string;  
attribute fsm_extract of Current_State : signal is "no";  
  
--stores dipsw status, used to detect changes  
signal dipsw_latch1 : std_logic_vector(3 downto 0);  
signal dipsw_latch2 : std_logic_vector(3 downto 0);  
  
. . .
```

For More Info...

- <http://toolbox.xilinx.com/docsan/xilinx92/books/docs/xst/xst.pdf>

User defined Attributes :Syntax

- in declarative section of architecture :

```
attribute pin_number : positive;  
attribute encoding : bit_vector;  
attribute INIT_00 : string;
```

- in architecture body

```
attribute pin_number of enable:signal is 14;
```

entity name

entity class

entity is not to be confused with VHDL entity.. entity class can be:
procedure, constant, component, group, signal, variable,...etc
(Ashenden 20.2)

ROM using initialized Block RAM

```
component RAMB4_S16
-- synopsys translate_off
  generic (
    INIT_00 : bit_vector := X"03A50943002501D90C1300390C250A0506860405050500060CF0000700050C00";
    INIT_01 : bit_vector := X"0000000000000000000000000000000000000000000000000000000000000000";
    INIT_02 : bit_vector := X"1000000000000000000000000000000000000000000000000000000000000000";
    INIT_03 : bit_vector := X"1000000000000000000000000000000000000000000000000000000000000000";
    INIT_04 : bit_vector := X"0000000000000000000000000000000000000000000000000000000000000000";
    INIT_05 : bit_vector := X"0000000000000000000000000000000000000000000000000000000000000000";
    INIT_06 : bit_vector := X"0000000000000000000000000000000000000000000000000000000000000000";
    INIT_07 : bit_vector := X"0000000000000000000000000000000000000000000000000000000000000000";
    INIT_08 : bit_vector := X"0000000000000000000000000000000000000000000000000000000000000000";
    INIT_09 : bit_vector := X"0000000000000000000000000000000000000000000000000000000000000000";
    INIT_0A : bit_vector := X"0000000000000000000000000000000000000000000000000000000000000000";
    INIT_0B : bit_vector := X"0000000000000000000000000000000000000000000000000000000000000000";
    INIT_0C : bit_vector := X"0000000000000000000000000000000000000000000000000000000000000000";
    INIT_0D : bit_vector := X"0000000000000000000000000000000000000000000000000000000000000000";
    INIT_0E : bit_vector := X"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
    INIT_0F : bit_vector := X"0000000000000000000000000000000000000000000000000000000000000000" )
-- synopsys translate_on
  port (DI      : in STD_LOGIC_VECTOR (15 downto 0);
        EN      : in STD_logic;
        WE      : in STD_logic;
        RST     : in STD_logic;
        CLK     : in STD_logic;
        ADDR    : in STD_LOGIC_VECTOR (7 downto 0);
        DO      : out STD_LOGIC_VECTOR (15 downto 0));
end component;
```

generics provide SIMULATION MODEL with initialized values

Initialized Block RAM for program ROM

```
attribute INIT_00: string;attribute INIT_01: string;attribute INIT_02: string;attribute INIT_03: string;
attribute INIT_04: string;attribute INIT_05: string;attribute INIT_06: string;attribute INIT_07: string;
attribute INIT_08: string;attribute INIT_09: string;attribute INIT_0A: string;attribute INIT_0B: string;
attribute INIT_0C: string;attribute INIT_0D: string;attribute INIT_0E: string;attribute INIT_0F: string;

attribute INIT_00 of RAMB_EXAMPLE : label is "ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff";
attribute INIT_01 of RAMB_EXAMPLE : label is "1000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_02 of RAMB_EXAMPLE : label is "1000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_03 of RAMB_EXAMPLE : label is "1000000000000000000000000000000000000000000000000000000000000000";

... etc.

begin

ramb_example : RAMB4_S16 port map (DI => "0000000000000000",
    EN => '1',
    WE => '0',
    RST => '0',
    CLK => clk,
    ADDR => a,
    DO => bigdo);

do <= bigdo(11 downto 0);
end syn;
```

Attributes are passed to BITFILE generation utility for inclusion in the configuration stream

Automation

- To make this useful, there needs to be some automated translation between program (the output of your assembler) and “INIT codes”
- For the picoblaze, the output of the assembler produces the whole program_rom entity directly

Programming Flow

- Design top level VHDL which includes a picoblaze (of course)
- Program Assembly language in “myprog.psm”, just a text file
- Run KCPSM.EXE on that file
- KCPSM uses some template files to produce a file named “myprog.vhd”
 - Contains generics for simulation
 - Init code attributes for synthesis
 - Include this file in your project, and instantiate the program rom in your design as if you wrote it yourself.