Interrupts on Picoblaze

EE354L Lecture and Lab

Vikram & Chethan
Gandhi Puvvada
March 2017

Revised substantially and rewritten
Gandhi Puvvada
March 2019

Section A: General Introduction to Interrupts and Interrupts in Picoblaze

Section B: Introduction to test_nexys3_Verilog.v

Section C: Relation between the test_nexys3_verilog.v and this Interrupts Lab

Section D: Cause of Interrupt -- Through polling in ISR
Cause of Interrupt -- Through prioritization logic in the fabric logic

Section E: ISR part of Walking LEDs using an indirect jump instr. JUMP@ (sX,sY)

Section F: Priority among interrupts – Fixed priority here. Starving?

Section G: Lab Part 1 and Part 2 – what is given and what is to be done

Section A

General Introduction to
Interrupts
and
Interrupts in Picoblaze

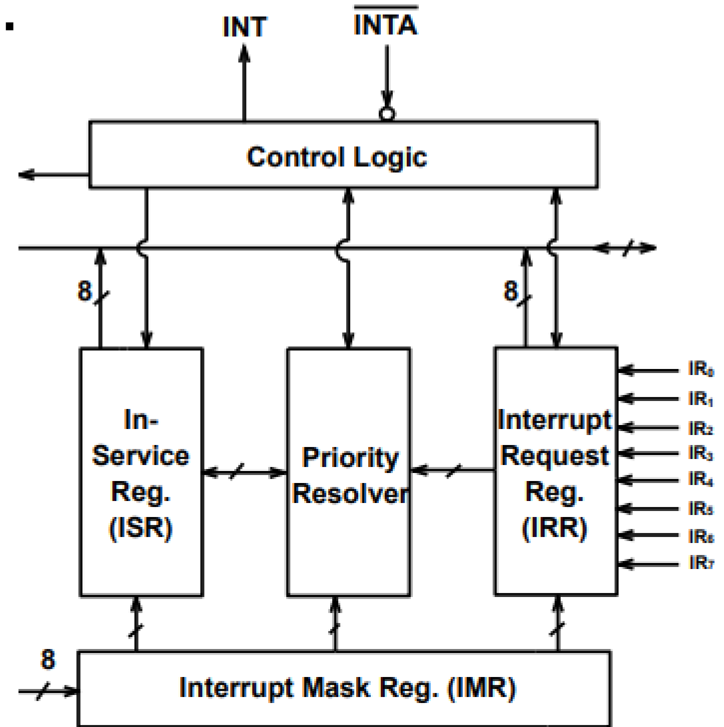
General Introduction

- An interrupt is a signal to the processor from hardware (*or software*) indicating an event that needs immediate attention.
  Note: Software interrupts are not covered in this course and Picoblaze does not support software interrupts anyways.
- The processor responds by suspending its current activities, saving its state, and executing a function, called an interrupt service routine(ISR), to deal with the event.
- This interruption is temporary. After the ISR finishes, the processor resumes the main program that was interrupted (and temporarily suspended).

General Introduction continued .

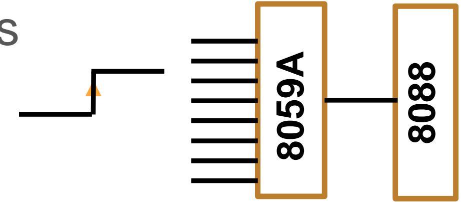
- You will come across this topic in EE454L, EE459L, EE457, EE557, EE560, etc.

- In the Intel x86 family, there is a chip, 8259A, a "Programmable Interrupt Controller" (in short PIC), which receives multiple interrupts, prioritizes and conveys a type number to the processor so that the processor gets to serve the interrupt quickly without incurring the latency associated with polling and finding the cause of interrupt.



General Introduction continued …

- We will not cover here the advanced features 8259A such as
  - -- interrupt-masking
  - -- edge-sensitive vs. level-sensitive interrupts
  - -- nested interrupt services among the prioritized interrupts
  - -- rotating priority among equal-priority requests



- But we wish to cover here a fabric logic mechanism to emulate “vectored interrupts” in most processor-based systems (particularly employed in Real-Time control systems, for example in autonomous vehicles (self-driving cars)).

GM’s self-driving car

Picoblaze Interrupt

Interrupts can be useful so KCPSM6 provides

a. ‘interrupt’ input pin,

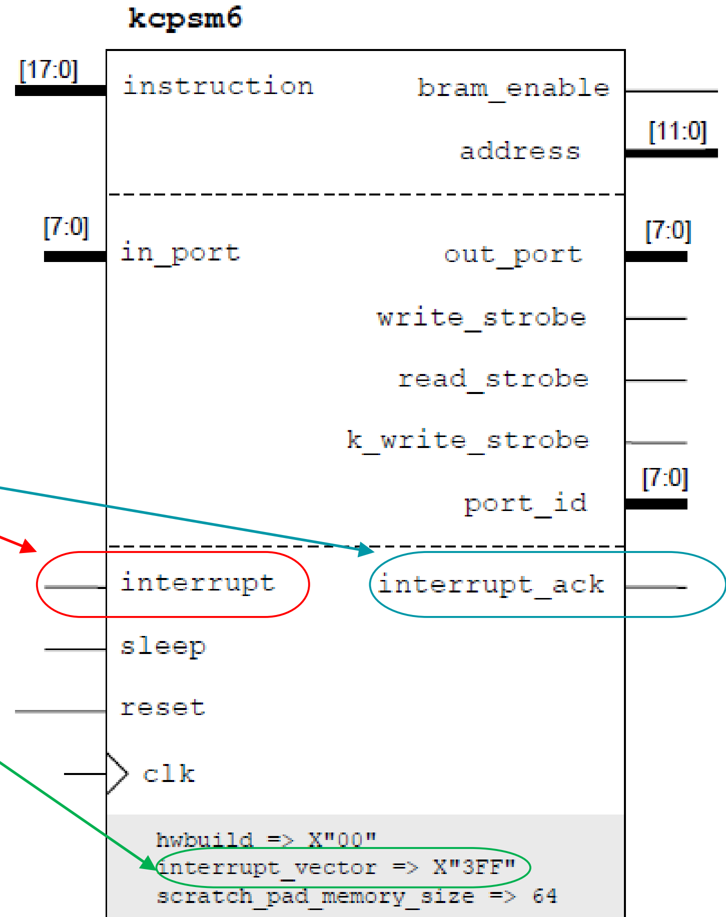
b. ‘interrupt_ack’ output pin,

c. an ‘interrrupt_vector’ parameter

d. four interrupt related instructions

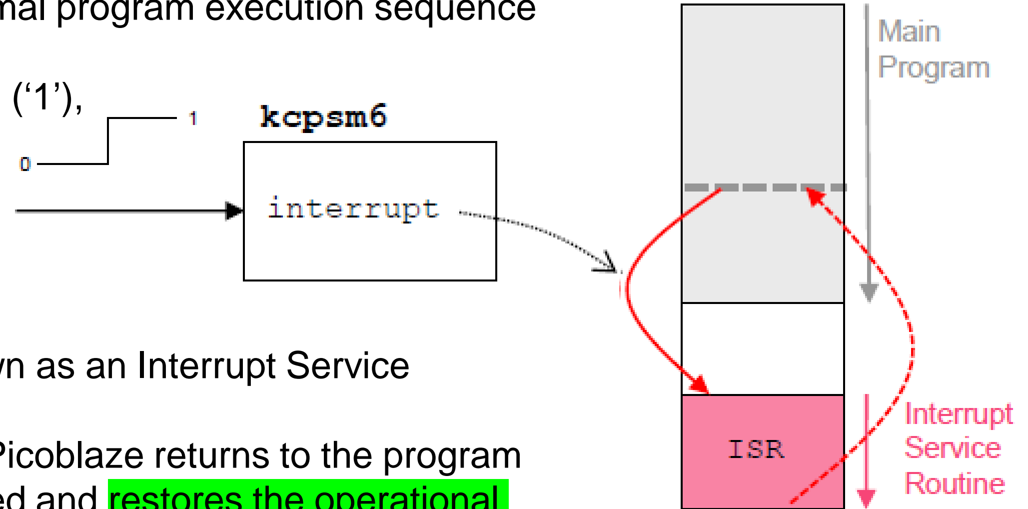
Interrupt Handling

```
28000   DISABLE INTERRUPT
28001   ENABLE INTERRUPT
29000   RETURNI DISABLE
29001   RETURNI ENABLE
```



Interrupt Mechanism in PicoBlaze

- An interrupt is used to interrupt the normal program execution sequence of Picoblaze
- When the 'interrupt' input is driven High ('1'), it will force Picoblaze to suspend the code that it is executing, and serve the interrupting device.
- Picoblaze saves its current operational state and jumps to execute a special section of program code known as an Interrupt Service Routine (ISR).
- Once the interrupt has been serviced, Picoblaze returns to the program at the point from where it was interrupted and restores the operational state so that it can resume execution of the program as if nothing had happened.
- Interrupts provide a way for Picoblaze to react to an event at any time and quite independently to the main task being performed.



Interrupt Vector

```
kcpsm6 #(
   .interrupt_vector     (12'h3FF),
   .scratch_pad_memory_size(64),
   .hwbuild            (8'h00))
```

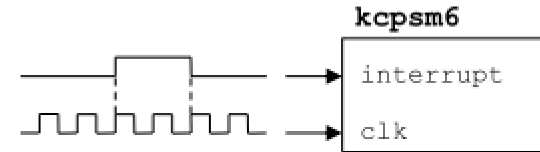
```
kcpsm6 #(
   .interrupt_vector     (12'h3C0),
   .scratch_pad_memory_size(64),
   .hwbuild            (8'h41))
```

- The interrupt vector is the address that Picoblaze effectively calls
- Its default value is 3FF hex.
- This can be set to any value within the range of the program memory available in your design (obviously not at 000 hex) (usually towards the end of the 1K (000-3FF) or 2K (000-7FF) or 4K (000-FFF).

In Picoblaze, every instruction takes exactly 2 clocks, no more, no less!

Extract from page 40 of KCPSM6_User_Guide_30Sept14

'Open-Loop' interrupt pulse

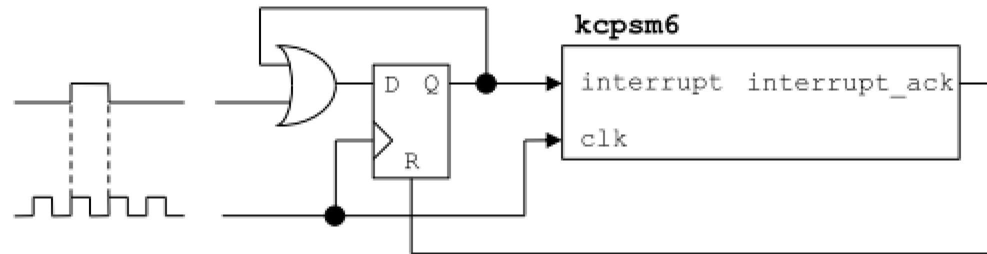


The simplest way of initiating an interrupt is to generate an active High pulse that has a duration of 2 clock cycles. The pulse can be longer but should have returned Low before the ISR completes otherwise KCPSM6 will immediate think there is another interrupt to service (remember that each instruction 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.

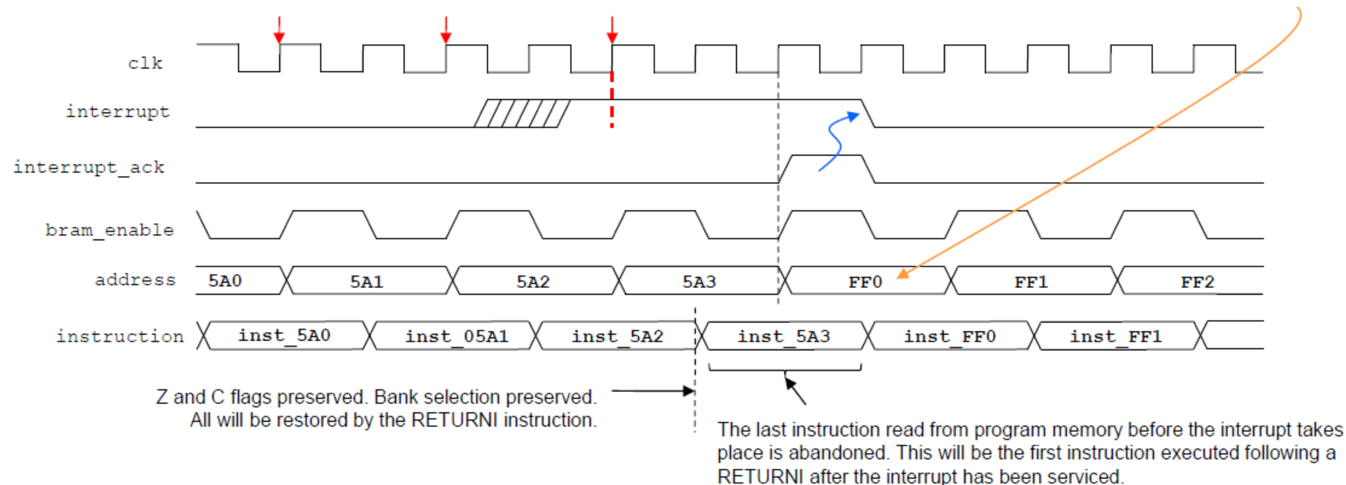


Interrupt Waveforms

- An interrupt is performed when the 'interrupt' input is driven High (provided, of course, if interrupts have been enabled by the main program using ENABLE INTERRUPT after RESET and they continue to be enabled)

- When Picoblaze detects an interrupt at the *beginning* of an instruction, it forces the next instruction in the program, that was prefetched, to be abandoned, preserves its address (the return address) and the current state of the 'Z' and the 'C' flags (on the PC stack), and then forces the program counter to change to the interrupt vector (FF0 in the example waveform below)

.interrupt_vector (12'hFF0),



Page 83

KCPSM6_User_Guide_30Sept14

Reproduced on the next page

INTERRUPT ENABLE

DISABLE INTERRUPT

Main points:

Interrupts are disabled when you switch on power (on Reset).

The boot up code decides when to turn-on interrupts.

It uses ENABLE INTERRUPT instruction which sets the IE Flag.

Interrupts are disabled before the start of "Critical Code" section, and are reenabled at the end. Critical code is a piece of code, when the intervention of an interrupt can have a detrimental affect. An example is a code on the side, producing a precise 6-clocks wide pulse.

ENABLE INTERRUPT
DISABLE INTERRUPT

Page 83

KCPSM6_User_Guide_30Sept14

See also pages 40-44

These instructions are used to control when interrupts are allowed to happen. Following device configuration or the application of a reset to the KCPSM6 macro the program starts executing from address zero and interrupts are disabled. Quite simply, this means that a High level on the 'interrupt' input will be ignored. The 'ENABLE INTERRUPT' instruction is used to enable interrupts by setting the interrupt enable flag (IE = 1). Hence this instruction needs to be included at a suitable point in your code to activate the 'interrupt' input such that KCPSM6 will react to an interrupt request. 'ENABLE INTERRUPT' has no other effects.

INTERRUPT ENABLE IE ← '1'

Z No Change C No Change

Important – You should *never* execute an 'ENABLE INTERRUPT' within your ISR (i.e. anywhere between the interrupt vector and the RETURNI instruction). Only one interrupt can be serviced at a time and if you re-enable interrupts before the end of the ISR then there is every risk that another interrupt may occur.

The 'DISABLE INTERRUPT' instruction is used to disable interrupts interrupts by clearing the interrupt enable flag (IE = 0). This would typically be used to temporarily prevent an interrupt from interfering with the execution of a critical section of code. 'DISABLE INTERRUPT' has no other effects.

DISABLE INTERRUPT IE ← '0'

Z No Change C No Change

Hint – It is considered good coding practice if these instructions are only executed when they actually modify the state of the interrupt enable flag. Whilst it does not cause a problem to execute the instruction in a way that confirms the state (e.g. using 'ENABLE INTERRUPT' when IE is already '1') such a coding style makes it less clear at what points you in your code interrupts are enabled and disabled and this can lead to confusion when debugging in the long term.

Examples

```
           TEST s6, 02
           JUMP NZ, no_pulse
           DISABLE INTERRUPT
           OUTPUTK 01, trigger_port
           LOAD s0, s0
           LOAD s0, s0
           OUTPUTK 00, trigger_port
           ENABLE INTERRUPT
no_pulse:  LOAD s3,  JUMP Z,
```

This section of code is taken from a program at a point when interrupts are enabled and therefore subject to interruption at any time that the interrupt input is driven High.

The state of Bit1 of register 's6' is tested, and if it is High, a pulse is generated on Bit0 of 'trigger_port'. A pair of 'LOAD s0, s0' instructions are used to stretch the pulse to be exactly 6 clock cycles in duration (3 instructions).

If an interrupt were to occur whilst generating the pulse then its duration could be considerably increased would have been unacceptable in this example. So to ensure that the pulse would always be 6 clock cycles long, interrupts are temporarily disabled only when the time critical code is executed.

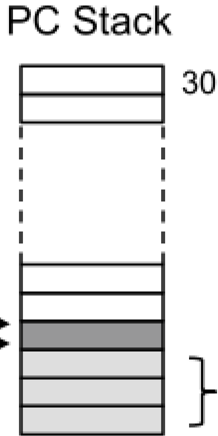
Another example would be to temporarily disable interrupts whist the main program reads information from scratch pad memory that was put there by a previous ISR. This would ensure that the information read is a complete set and not a mixture of the information resulting from 2 separate interrupts.

© Copyright 2010-2014 Xilinx

Page 84

KCPSM6_User_Guide_30Sept14

Reproduced on the next page



Main points:

PC Stack (also commonly known as a Program Stack or a Return Address Stack) is an important part of

(i) the call to and corresponding return from a subroutine

(ii) the program control transfer to and corresponding return from an ISR

ISR = Interrupt Service Routine

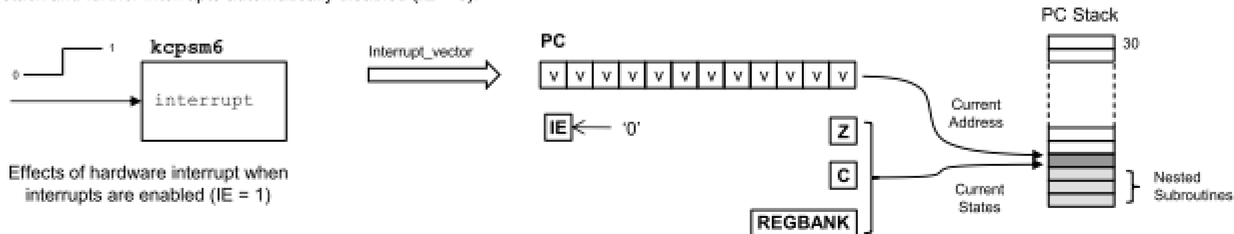
Stack is a LIFO (Last-in First-out) data structure.

In the case of a transfer of program control to an ISR, besides the return address, the flag bits (Z, C, RegBank) are pushed on to and popped from the stack

RETURNI ENABLE
RETURNI DISABLE

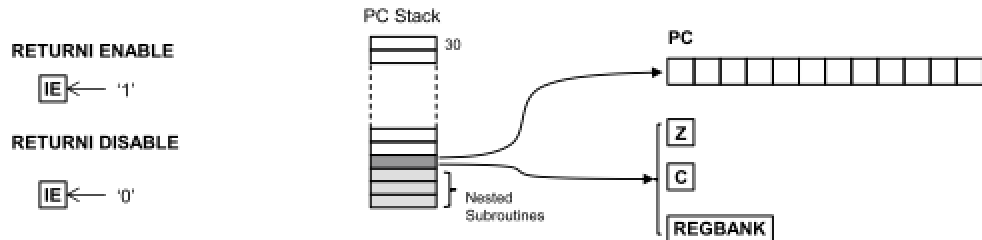
See also pages 40-44

When an interrupt occurs the program counter is loaded with the interrupt vector and the current address (corresponding with the location of the instruction that is abandoned) is pushed onto the stack. In addition, the states of the carry flag (C), the zero flag (Z) and the register bank selection are also pushed onto the stack and further interrupts automatically disabled (IE = 0).



Effects of hardware interrupt when interrupts are enabled (IE = 1)

The 'RETURNI' instruction is similar to the unconditional 'RETURN' instruction *but it must only be used to terminate an interrupt service routine* (ISR). When the 'RETURNI' is executed, the last address held on the PC Stack is popped off and loaded directly into the program counter so that the program resumes execution starting with the instruction that was abandoned when the interrupt occurred. In addition, the RETURNI restores the values of the carry flag (C), the zero flag (Z) and the register bank selection so that they are exactly the same as when the interrupt occurred. Either the 'ENABLE' or 'DISABLED operand must be used to specify if interrupts are to be enabled or disabled on return from the ISR.



Continued on next page....

© Copyright 2010-2014 Xilinx

XILINX

Page 85

KCPSM6_User_Guide_30Sept14

Reproduced on the next page

Main points:
Please differentiate between RETURN (to return from a subroutine) and RETURNI (to return from an ISR).
RETURNI pops the flags also besides the return address. Stack should be balanced.

In execution, each CALL should match with one RETURN (no more, no less).
Similarly, each interrupt should match with one RETURNI at the end of the service (no more, no less).

Examples of RETURNI ENABLE and RETURNI DISABLE.

RETURNI ENABLE
RETURNI DISABLE

See also pages 40-44

Important 1 – Always terminate an ISR with a 'RETURNI' and always terminate a normal subroutine with 'RETURN'. The execution of the inappropriate instruction will result in incorrect operation. Obviously that would be bad enough, but combined with the whole concept of interrupts occurring at any point in the execution of the main code the symptoms of the incorrect operation failure can be subtle and make it extremely difficult to identify the cause.

Important 2 – Just as each 'RETURN' must be executed to correspond with the 'CALL' that invoked a normal subroutine, a 'RETURNI' must only be executed to correspond with the interrupt that invoked the ISR. Your ISR can exploit KCPSM6's ability to implement nested subroutines just as they can be used in any part of your program but it is vital that each level is invoked and completed in order. The maximum number of levels is 30 and it should be remembered that an interrupt requires one of these levels. If an interrupt does result in a stack overflow then KCPSM6 will automatically generate an internal reset. Likewise if RETURNI is used in a way that results in a stack underflow then KCPSM6 will also reset itself automatically.

Examples

```
ISR: ADD sE, 1'd
     ADDCY sF, 0'd
     RETURNI ENABLE
```

This simple ISR increments the 16-bit value contained in the register pair [sF, sE]. This may relate to a scheme in which interrupts occur at regular intervals to provide the basis for a real time clock or timer (i.e. the value held in [SF,sE] is then used by the main program when required). The 'RETURNI ENABLE' instruction terminates the ISR and enables interrupts ready for the next time.

```
ISR: INPUT sF, int_data0
     STORE sF, 2A
     INPUT sF, int_data1
     STORE sF, 2B
     LOAD sE, 2A
     RETURNI DISABLE
```

This ISR reads two bytes of information from input ports and stores them in scratch pad memory. It is reasonable to assume that this information relates in some way to the reason for the interrupt and therefore probably represents some important information that had to be captured at that particular time. It can also be imagined that the main program needs to process this special information in some way with the value '2A' loaded into register 'sE' signifying that information has been captured and stored starting at location 2A hex. It can be imagined that the main program must be given time to process the captured information so the 'RETURNI DISABLE' instruction terminates the ISR but prevents a further interrupts overwriting the important information before it has been used. The main program would use an 'ENABLE INTERRUPT' once it had.

```
ISR: LOAD sA 00
     CALL motor_drive
     RETURNI ENABLE
```

Providing all the normal rules of nested subroutines are followed then an ISR can also make use of subroutines.

```
motor_drive: OUTPUT sA, PWM_value
             OUTPUTK 01, update_strobe
             OUTPUTK 00, update_strobe
             RETURN
```

Hint – Be very careful to make sure that no code executed as part of your ISR procedure contains an 'ENABLE INTERRUPT' instruction.

© Copyright 2010-2014 Xilinx

XILINX

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.

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
```

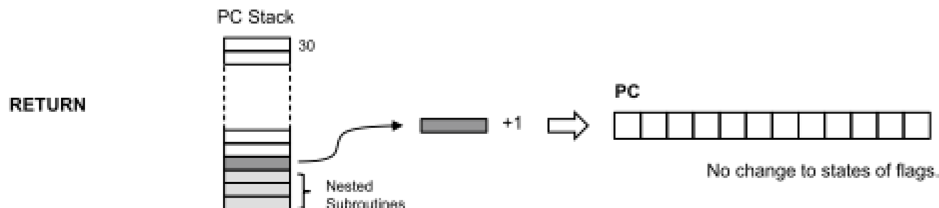
Hint – A subroutine can be located anywhere in a program relative to the CALL instructions that invoke it but it is vital that the subroutine is only executed as the result of a CALL otherwise their will be no address in the PC stack to correspond with the subsequent RETURN instruction.

© Copyright 2010-2014 Xilinx

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.



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.

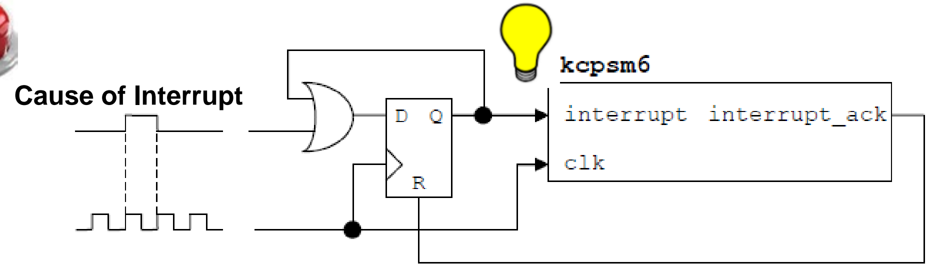
Whilst this example does show the general arrangement it actually describes a rather special case when we look at the code in detail. In the main program [s9,s8] has been cleared and then 's1' has been loaded with 30 decimal. The 'test_stack' subrouine adds the value of 's1' to [s9,s8] and then decrements 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.

© Copyright 2010-2014 Xilinx

XILINX

Interrupt Mechanism in PicoBlaze

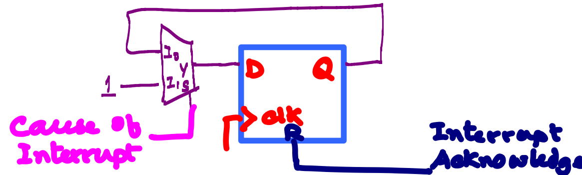
- Closed-Loop interrupt design
    a. Design drives the interrupt signal High to request an interrupt
    b. Then keeps driving it High until Picoblaze generates an 'interrupt_ack' pulse confirming that it has seen it.



- This ensures that the interrupt will always be observed by Picoblaze when it is able to.

- If interrupts have been temporarily disabled deliberately (for example, while executing a critical section of code), or whilst servicing a previous interrupt, then the response will be delayed but the event can not be missed.

Note: The OR gate above is the logic simplification of the recirculating mux on the side.



Section B

Introduction to
test_nexys3_verilog.v

test_nexys3_verilog.v

The test_nexys3_verilog.v does the following three things.

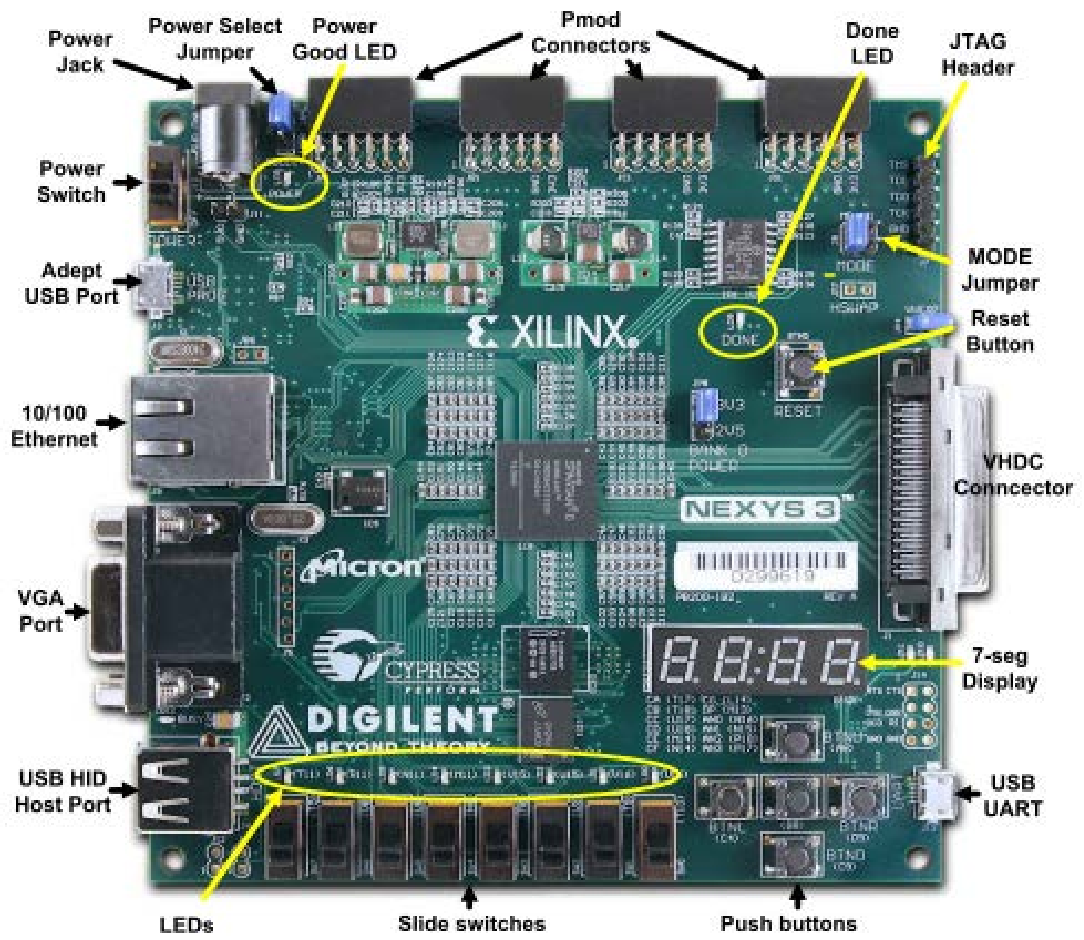
1. Displays walking LED pattern using a slow_bits pattern

```
always @ (slow_bits) // divclk[26:24]
  begin
    case (slow_bits)
      3'b000: walking_leds = 8'b00000001 ;
```

2. When one of the 4 push buttons is pressed, the walking LED display is suspended and a pair of LEDs glows steadily corresponding to the PB pressed.

3. The 8 switches are read. Their true and compliment values are displayed on the 4 SSDs.

Please run through the test_nexys3_verilog.v code once.



Power Jack
Power Select Jumper
Power Good LED
Pmod Connectors
Done LED
JTAG Header
Power Switch
MODE Jumper
Adept USB Port
Reset Button
10/100 Ethernet
VHDC Conncector
VGA Port
7-seg Display
USB HID Host Port
USB UART
LEDs
Slide switches
Push buttons
XILINX
NEXYS 3
Micron
CYPRESS
DIGILENT

Clock Division
Dp Dot Point flashing

```
wire [2:0]    slow_bits;         // to control walking led pattern
wire [1:0]    sev_seg_clk;           // to control SSD scanning
reg  [27:0]   divclk;

always @(posedge board_clk, posedge reset)
  begin
      if (reset)
      divclk <= 0;
      else
      divclk <= divclk + 1'b1;
  end
//-------------
assign Dp = divclk[25]; // The dot point on each SSD flashes
                  // divclk[25] (~1.5Hz) = (100MHz / 2**26)
              // count the number of flashes for a minute, you should get about 90

//assign sys_clk = divclk[18];  // a slow clock for use by students' core design
                                 // divclk[18] (~191Hz) = (1000MHz / 2 **19)
assign sev_seg_clk  = divclk[16:15]; // 7 segment display scanning is completed
                            // every divclk[16] (~763Hz) = (100MHz / 2 **17)

assign slow_bits  =  divclk[26:24];
```

Switches to SSDs

```
// SSD (Seven Segment Display)

reg  [3:0] SSD;
wire [3:0] SSD3, SSD2, SSD1, SSD0;

assign An0  = ~(~(sev_seg_clk[1]) && ~(sev_seg_clk[0]));  // when sev_seg_clk = 00
assign An1  = ~(~(sev_seg_clk[1]) &&  (sev_seg_clk[0]));  // when sev_seg_clk = 01
assign An2  = ~( (sev_seg_clk[1]) && ~(sev_seg_clk[0]));  // when sev_seg_clk = 10
assign An3  = ~( (sev_seg_clk[1]) &&  (sev_seg_clk[0]));  // when sev_seg_clk = 11

assign SSD0 = {  Sw3,   Sw2,   Sw1,   Sw0};
assign SSD1 = {  Sw7,   Sw6,   Sw5,   Sw4};
assign SSD2 = { ~Sw3,  ~Sw2,  ~Sw1,  ~Sw0};
assign SSD3 = { ~Sw7,  ~Sw6,  ~Sw5,  ~Sw4};

always @ (sev_seg_clk, SSD0, SSD1, SSD2, SSD3)
begin
   case (sev_seg_clk)
         2'b00: SSD = SSD0;
         2'b01: SSD = SSD1;
         2'b10: SSD = SSD2;
         2'b11: SSD = SSD3;
   endcase
end
```

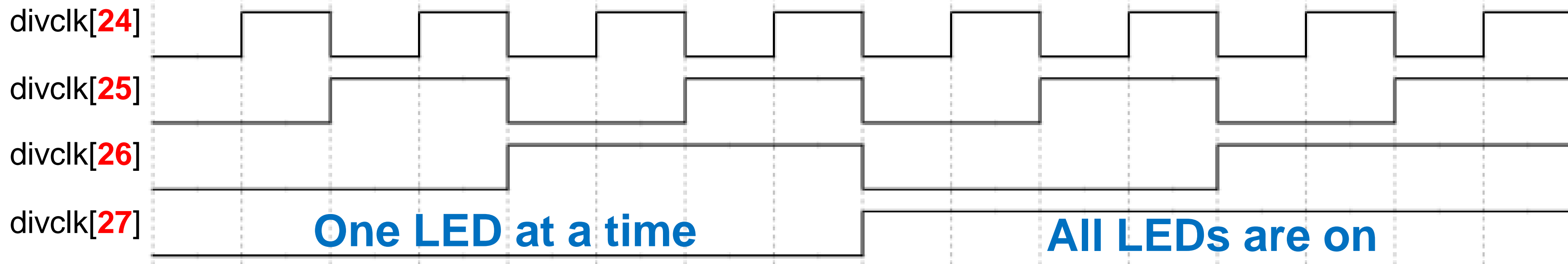
Hex to SS conversion

```
assign   {Ca, Cb, Cc, Cd, Ce, Cf, Cg} = cathodes;
//
// Following is Hex-to-SSD conversion.
always @ (SSD)
    begin
      case (SSD)
        4'b0000: cathodes = 7'b0000001 ; // 0
        4'b0001: cathodes = 7'b1001111 ; // 1
        4'b0010: cathodes = 7'b0010010 ; // 2
        4'b0011: cathodes = 7'b0000110 ; // 3
        4'b0100: cathodes = 7'b1001100 ; // 4
        4'b0101: cathodes = 7'b0100100 ; // 5
        4'b0110: cathodes = 7'b0100000 ; // 6
        4'b0111: cathodes = 7'b0001111 ; // 7
        4'b1000: cathodes = 7'b0000000 ; // 8
        4'b1001: cathodes = 7'b0000100 ; // 9
        4'b1010: cathodes = 7'b0001000 ; // A
        4'b1011: cathodes = 7'b1100000 ; // b
        4'b1100: cathodes = 7'b0110001 ; // C
        4'b1101: cathodes = 7'b1000010 ; // d
        4'b1110: cathodes = 7'b0110000 ; // E
        4'b1111: cathodes = 7'b0111000 ; // F
        default: cathodes = 7'bXXXXXXX ; // default
      endcase
    end
```

Walking LEDs

```
assign button_pressed = BtnL | BtnU | BtnD | BtnR ;

assign {Ld7, Ld6, Ld5, Ld4, Ld3, Ld2, Ld1, Ld0} = leds;
assign leds = button_pressed ? {BtnL, BtnL, BtnU, BtnU, BtnD, BtnD, BtnR, BtnR} :
                              ( divclk[27] ? 8'b11111111 : walking_leds );
// Notice that when divclk[27] is zero, the slow_bits (i.e. divclk[26:24])
// go through a complete sequence of 000-111.

always @ (slow_bits)
  begin
    case (slow_bits)
      3'b000: walking_leds = 8'b00000001 ;
      3'b001: walking_leds = 8'b00000010 ;
      3'b010: walking_leds = 8'b00000100 ;
      3'b011: walking_leds = 8'b00001000 ;
      3'b100: walking_leds = 8'b00010000 ;
      3'b101: walking_leds = 8'b00100000 ;
      3'b110: walking_leds = 8'b01000000 ;
      3'b111: walking_leds = 8'b10000000 ;
      default:walking_leds = 8'bXXXXXXXX ;
    endcase
  end
```

divclk[27], divclk[26:24], walking LEDs




```
assign leds = button_pressed ? {BtnL, BtnL, BtnU, BtnU, BtnD, BtnD, BtnR, BtnR} :
                    ( divclk[27] ? 8'b11111111 : walking_leds );
```

Section C

Relation between the
test_nexys3_verilog.v
and this
Interrupts Lab

Push Buttons, slow_bits, Walking LEDS

test_nexys3_verilog.v

```verilog
151 //-------------
152 // Buttons and LEDs
153
154   wire         button_pressed;
155
156   assign button_pressed = BtnL | BtnU | BtnD | BtnR ;
157
158   wire  [7:0]   leds;
159   reg   [7:0]   walking_leds;
160
161   assign {Ld7, Ld6, Ld5, Ld4, Ld3, Ld2, Ld1, Ld0} = leds;
162   assign leds = button_pressed ? {BtnL, BtnL, BtnU, BtnU, BtnD, BtnD, BtnR, BtnR} :
163                            ( divclk[27] ? 8'b11111111 : walking_leds );
164   // Notice that when divclk[27] is zero, the slow_bits (i.e. divclk[26:24])
165   // go through a complete sequence of 000-111.
166
167   always @ (slow_bits)
168     begin
169       case (slow_bits)
170         3'b000: walking_leds = 8'b00000001 ;
171         3'b001: walking_leds = 8'b00000010 ;
172         3'b010: walking_leds = 8'b00000100 ;
173         3'b011: walking_leds = 8'b00001000 ;
174         3'b100: walking_leds = 8'b00010000 ;
175         3'b101: walking_leds = 8'b00100000 ;
176         3'b110: walking_leds = 8'b01000000 ;
177         3'b111: walking_leds = 8'b10000000 ;
178         default:walking_leds = 8'bXXXXXXXX ;
179       endcase
180     end
```

divclk[27] provides flashing effect

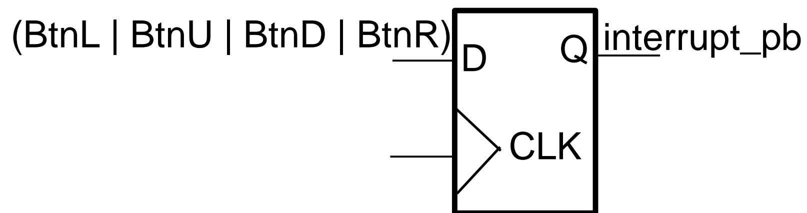
slow_bits divclk[26:24] provided the walking leds effect

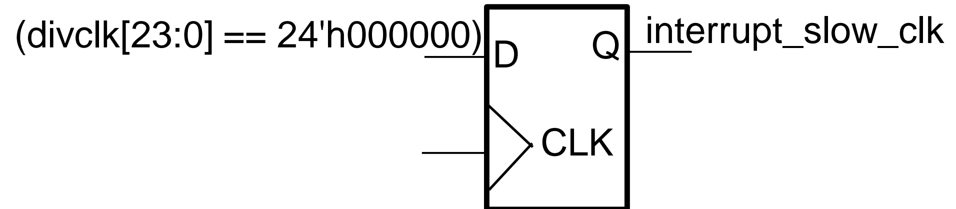
Replaced by the Picoblaze

We cause (divclk[23:0] == 24'h000000) condition to interrupt the Picoblaze.

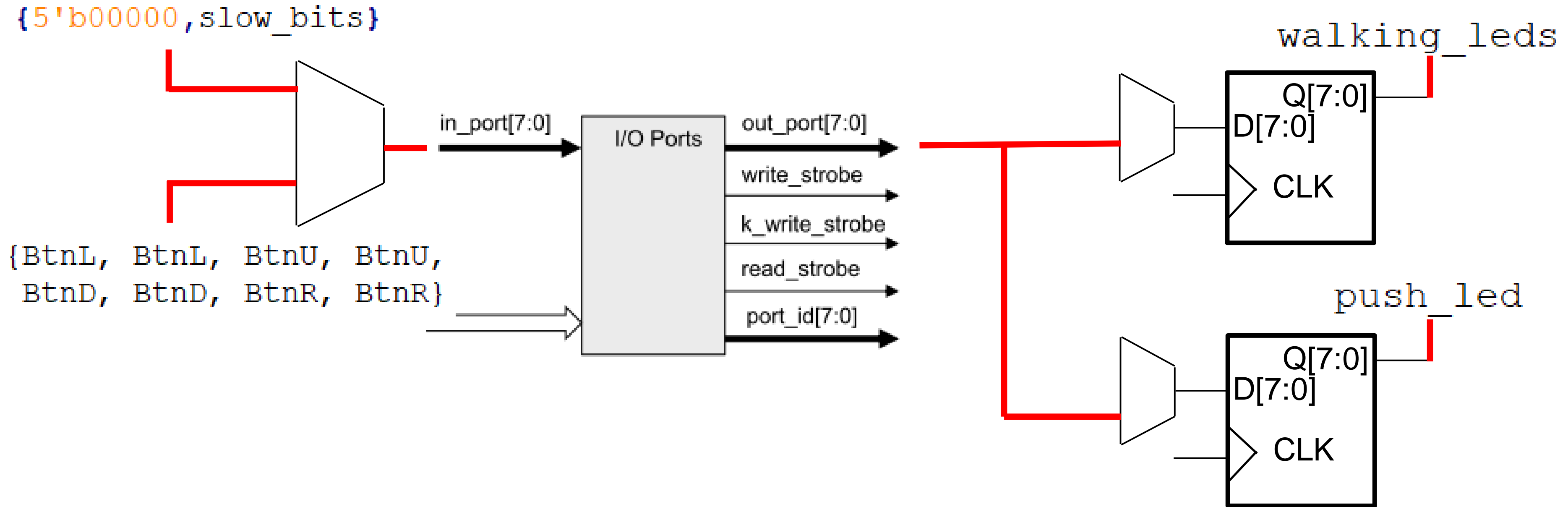
This interruption occurs at the beginning of each of the eight combinations of divclk[26:24] and each time, Picoblaze is interrupted, it reads the divclk[26:24] in an input port called slow_clk, and accordingly sends a walking LED pattern to the running_led_output (an output port).

On being interrupted by the interrupt_pb, the picoblaze reads the four push buttons (replicated as eight push buttons) in an input port called pb_input and sends out the same through an output port called pb_led_output.





Two input ports and two output ports for the above



Hex to 7-seg conversion remains in the fabric logic

test_nexys3_verilog.v

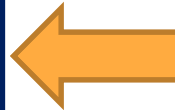
```
// Following is Hex-to-SSD conversion.
always @ (SSD)
    begin
      case (SSD)
        4'b0000: cathodes = 7'b0000001 ; // 0
        4'b0001: cathodes = 7'b1001111 ; // 1
        4'b0010: cathodes = 7'b0010010 ; // 2
        4'b0011: cathodes = 7'b0000110 ; // 3
        4'b0100: cathodes = 7'b1001100 ; // 4
        4'b0101: cathodes = 7'b0100100 ; // 5
        4'b0110: cathodes = 7'b0100000 ; // 6
        4'b0111: cathodes = 7'b0001111 ; // 7
        4'b1000: cathodes = 7'b0000000 ; // 8
        4'b1001: cathodes = 7'b0000100 ; // 9
        4'b1010: cathodes = 7'b0001000 ; // A
        4'b1011: cathodes = 7'b1100000 ; // b
        4'b1100: cathodes = 7'b0110001 ; // C
        4'b1101: cathodes = 7'b1000010 ; // d
        4'b1110: cathodes = 7'b0110000 ; // E
        4'b1111: cathodes = 7'b0111000 ; // F
        default: cathodes = 7'bXXXXXXX ; // default
      endcase
    end
```

Displaying Switch data on the 4 SSDs

test_nexys3_verilog.v

```
assign sev_seg_clk  = divclk[16:15];
```

```
181  //
182  // SSD (Seven Segment Display)
183
184  reg  [3:0] SSD;
185  wire [3:0] SSD3, SSD2, SSD1, SSD0;
186
187  assign An0  = ~(~(sev_seg_clk[1]) && ~(sev_seg_clk[0]));
188  assign An1  = ~(~(sev_seg_clk[1]) &&  (sev_seg_clk[0]));
189  assign An2  = ~( (sev_seg_clk[1]) && ~(sev_seg_clk[0]));
190  assign An3  = ~( (sev_seg_clk[1]) &&  (sev_seg_clk[0]));
191
192  assign SSD0 = {  Sw3,   Sw2,   Sw1,   Sw0};
193  assign SSD1 = {  Sw7,   Sw6,   Sw5,   Sw4};
194  assign SSD2 = { ~Sw3,  ~Sw2,  ~Sw1,  ~Sw0};
195  assign SSD3 = { ~Sw7,  ~Sw6,  ~Sw5,  ~Sw4};
196
197  always @ (sev_seg_clk, SSD0, SSD1, SSD2, SSD3)
198  begin
199     case (sev_seg_clk)
200             2'b00: SSD = SSD0;
201             2'b01: SSD = SSD1;
202             2'b10: SSD = SSD2;
203             2'b11: SSD = SSD3;
204     endcase
205  end
```



Picoblaze reads these switches through an input port and returns them through an output port

One input port and one output port for the switches

switch_to_SSD

{Sw7, Sw6, Sw5, Sw4,
Sw3, Sw2, Sw1, Sw0}

in_port[7:0]

I/O Ports

out_port[7:0]

write_strobe

k_write_strobe

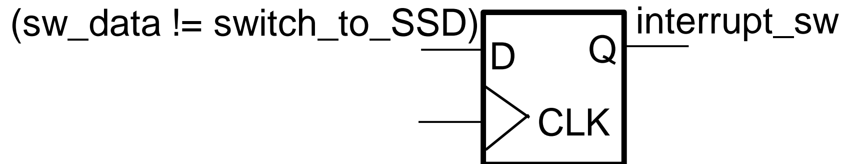
read_strobe

port_id[7:0]

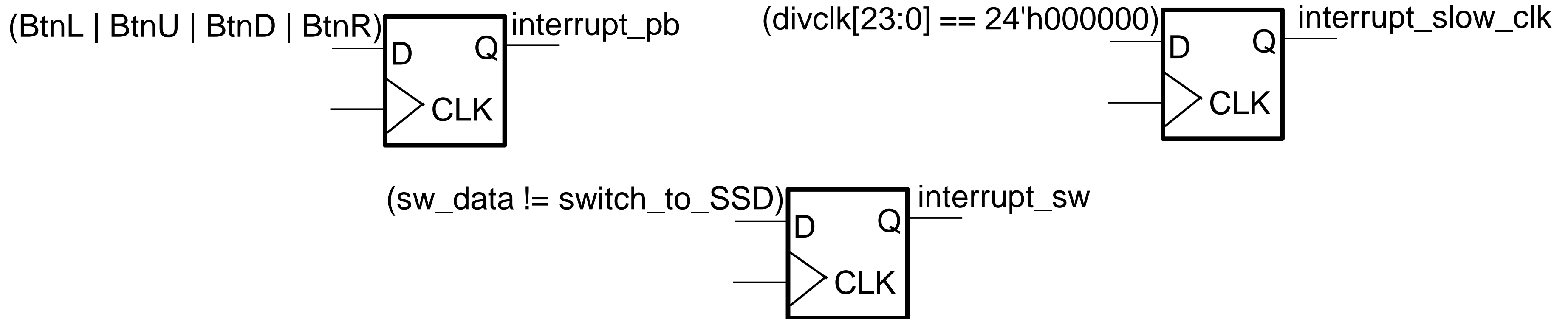
Q[7:0]

D[7:0]

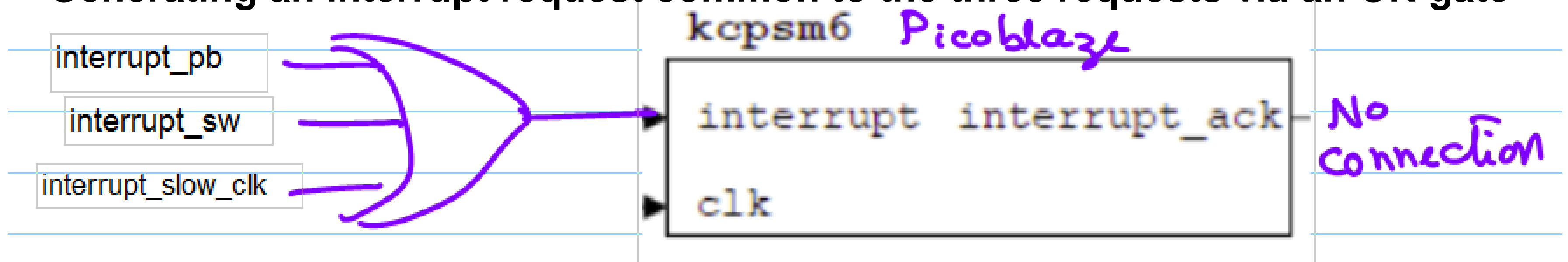
CLK



Three interrupt requests are generated and held in interrupt-request registers (FFs). An OR gate combines the requests into one and conveys it to the interrupt input pin of the Picoblaze.



Generating an interrupt request common to the three requests via an OR gate



Section D

How does Picoblaze determine cause of the interrupt and how does it render the needed service?

1. Through polling in ISR, 2. Through prioritization logic in fabric logic

Goals of this Lab

- To design and demonstrate Interrupt servicing by polling using Picoblaze and test it on NEXYS 3 board. This is Part 1 of the lab with "_Polling" in the name of the source files.

- To design and demonstrate Interrupt servicing by "priority encoding in fabric logic" such that Picoblaze does not have to identify the cause of the Interrupt requestor by polling. The Interrupt Latency (the delay in starting the needed service) is reduced. This is Part 2 of the lab with "_no_Polling" in the name of the source files.

- To understand interrupts, write Interrupt Service Routine (ISR) in Assembly language, and design fabric logic to complete an interrupt-based service system.

Section D Part 1

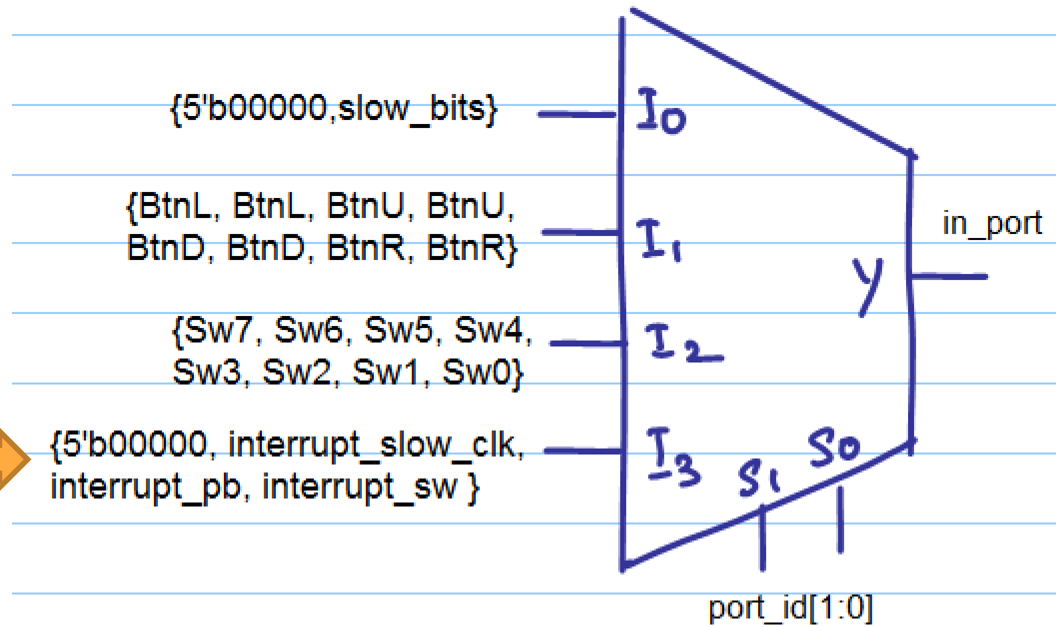
Here Picoblaze determines cause of the interrupt through polling in ISR.

Depending on the cause it jumps to a section of the service routine to render the service.

Polling Interrupt requestors (Part 1 of the lab)

- When the Interrupt is driven High, the Picoblaze does not know who caused the interrupt. In the fabric logic in the TOP design, we can arrange, several single-bit registers (FFs, one for each cause). Picoblaze can read all of these together (concatenated together to form one 8-bit input port) at the beginning of the ISR.
- In the ISR in the .psm file, you poll to see which one of the requestors caused the interrupt (if multiple interrupt-requesting devices are present) (in our lab we have three requests).
- After determining the interrupt cause through polling, you jump to that particular section of ISR, provide the needed service, and finally return to the main program (using usually RETURNI ENABLE).

Four Input ports



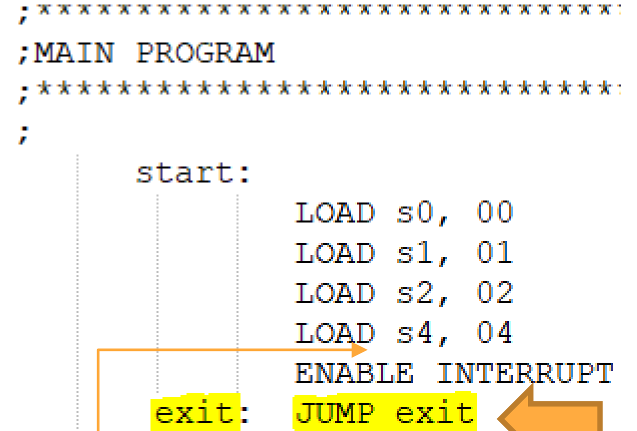

```
always @ (*)

begin

    case (port_id[1:0])
        2'b00 : in_port <= {5'b00000,slow_bits};
        2'b01 : in_port <= {BtnL, BtnL, BtnU, BtnU, BtnD, BtnD, BtnR, BtnR};
        2'b10 : in_port <= {Sw7, Sw6, Sw5, Sw4, Sw3, Sw2, Sw1, Sw0};
        2'b11 : in_port <= {5'b00000, interrupt_slow_clk, interrupt_pb, interrupt_sw };
    endcase
end
```

Main Program running on Picoblaze

Here, the main program is nothing but waiting for interrupts to occur in an infinite loop.

This may be a common practice in many processor-based control systems, where the processor basically waits for an event to occur to serve.

In our main program, we initialized s0, s1, s2, and s4 registers with suitable constants so that we can use them in ISR, to produce the three acknowledge pulses, via bits of the `interrupt_acknowledged` register.



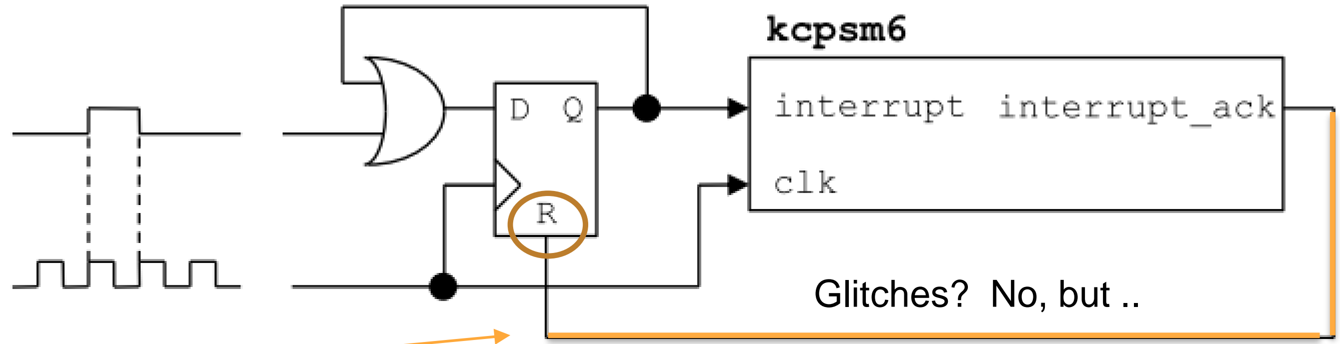

```
OUTPUT  s0, interrupt_acknowledged        ;
```

Find out why you need this line in the Main program then?

It is not right to use an asynchronous clear or set on FFs except for initialization during power-on reset

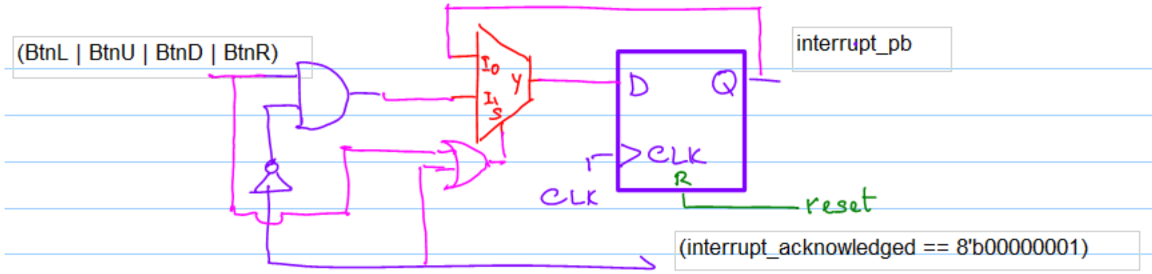
Page 43

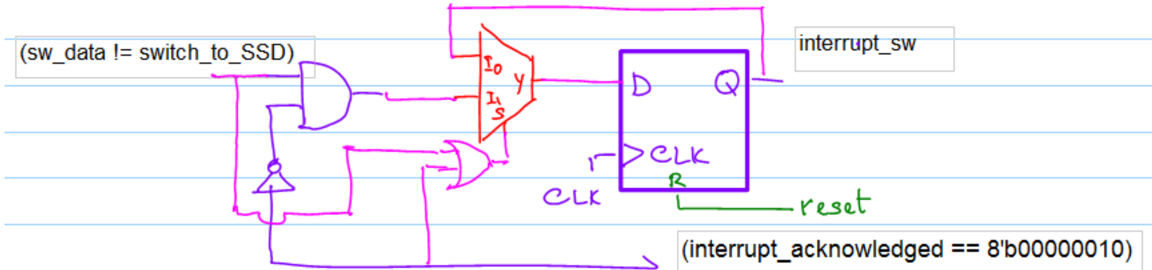
KCPSM6_User_Guide_30Sept14

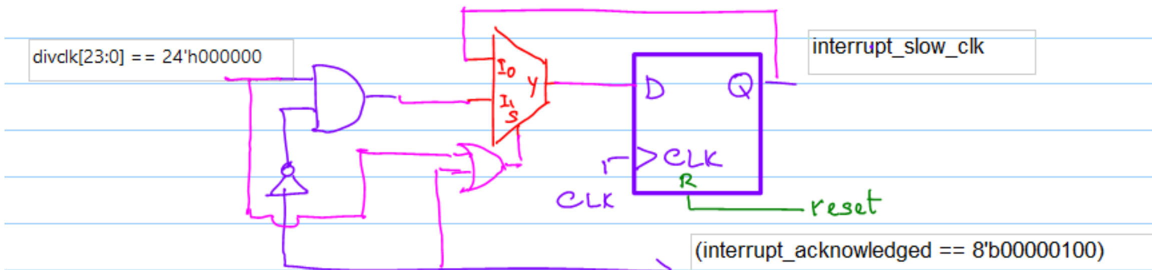


If the highlighted line is a control signal produced by a combinational logic such as OFL, it is very likely to have glitches, and then the above design would be a major blunder. Even though the interrupt_ack pulse is a clean (glitch-free) pulse here, the above design is still discouraged as it is a violation of DFT (Design for Testability taught in EE680). So we will use synchronous clearing of the three interrupt request FFs in our design.

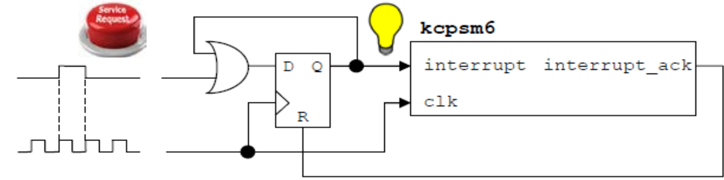
Generating Interrupt Requests in Fabric Logic



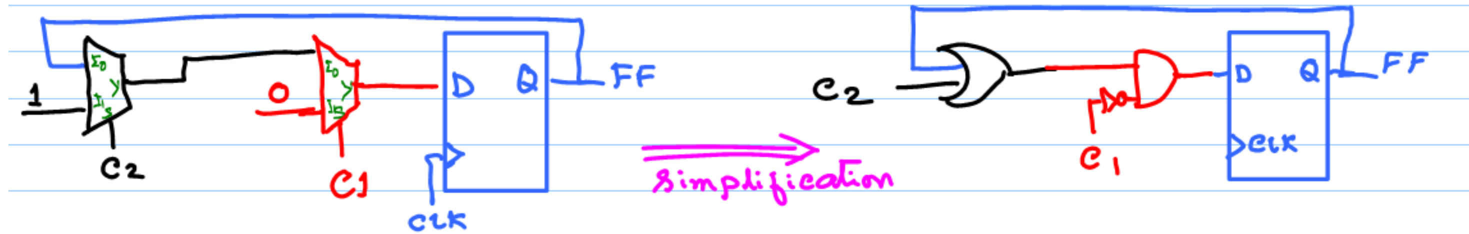




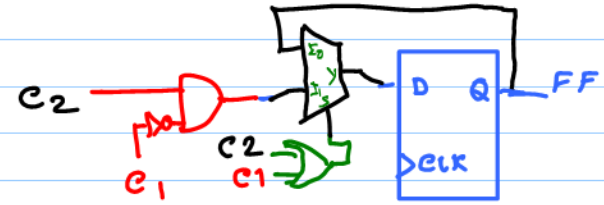
The logic in the above page can be drawn more elegantly as derived below.




```
always @(posedge clk)// we should avoid all asynchronous actions during normal operation
  if (C1) FF <= 0;  // should clear synchronously // Interrupt Acknowledged
  else if (C2) FF <= 1; // should set // Short-lived Interrupt Request was made
  else FF <= FF; // should remain stay put // this statement can be avoided as it is implicit in Verilog coding
```



Logic on the last page was drawn "not-so-thoughtfully" :(
It appears that I have started with a recirculating mux on the FF and built the logic around that!

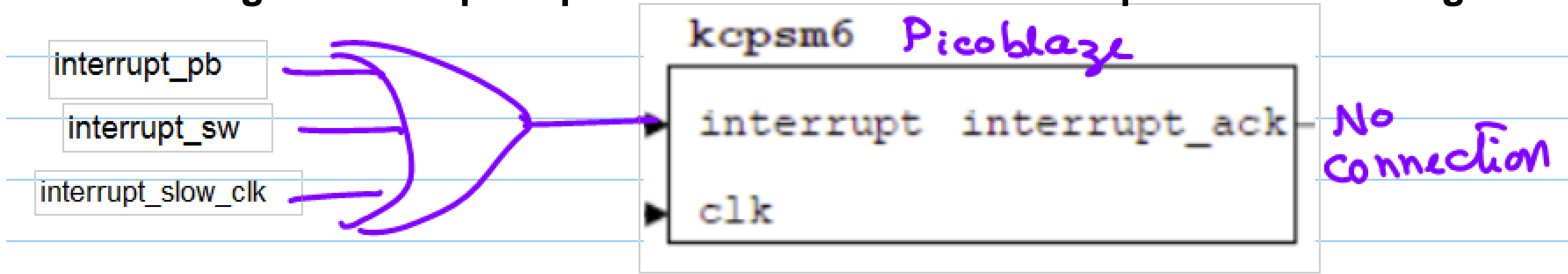


```verilog
//=========
if(interrupt_acknowledged == 8'b00000001)   // Indicates Push Button Interrupt was acknowledged by PicoBlaze
    begin
        interrupt_pb <= 1'b0; // Synchronous reset
    end
else if(BtnL | BtnU | BtnD | BtnR) //Push Button Interrupt Condition
    begin
        interrupt_pb <= 1'b1;
    end
//=========
if(interrupt_acknowledged == 8'b00000010)// Indicates Switch Interrupt was acknowledged by PicoBlaze
    begin
        interrupt_sw <= 1'b0; // Synchronous reset
    end
else if(sw_data != switch_to_SSD) //Set interrupt_sw if the current Switch Data is different from the output
    begin
        interrupt_sw <= 1'b1;
    end
//=========
if(interrupt_acknowledged == 8'b00000100)// Indicates Wlaking LED Interrupt was acknowledged by PicoBlaze
    begin
        interrupt_slow_clk <= 1'b0; // Synchronous reset
    end
else if(divclk[23:0] == 24'h000000) //Walking LED Interrupt Condition Check
begin
    interrupt_slow_clk <= 1'b1;
end
//=========
```

In the fabric logic

Extract from interrupt_polling_picoblaze.v

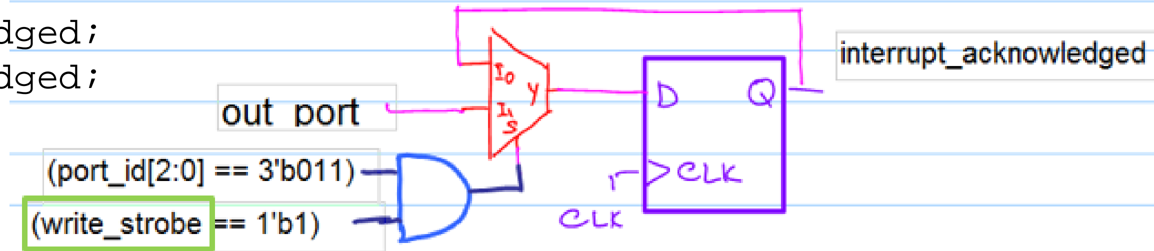
Generating an interrupt request common to the three requests via an OR gate




```
//Set interrupt signal to 1 if any one ore more of the Interrupt conditions are satisfied
assign  interrupt = ((interrupt_sw == 1'b1) || (interrupt_pb == 1'b1) || (interrupt_slow_clk == 1'b1));
```

Generating individual interrupt acknowledge pulses

```
; Acknowledge the Push Button Interrupt
OUTPUT   s1, interrupt_acknowledged;
OUTPUT   s0, interrupt_acknowledged;
```



Four Output ports

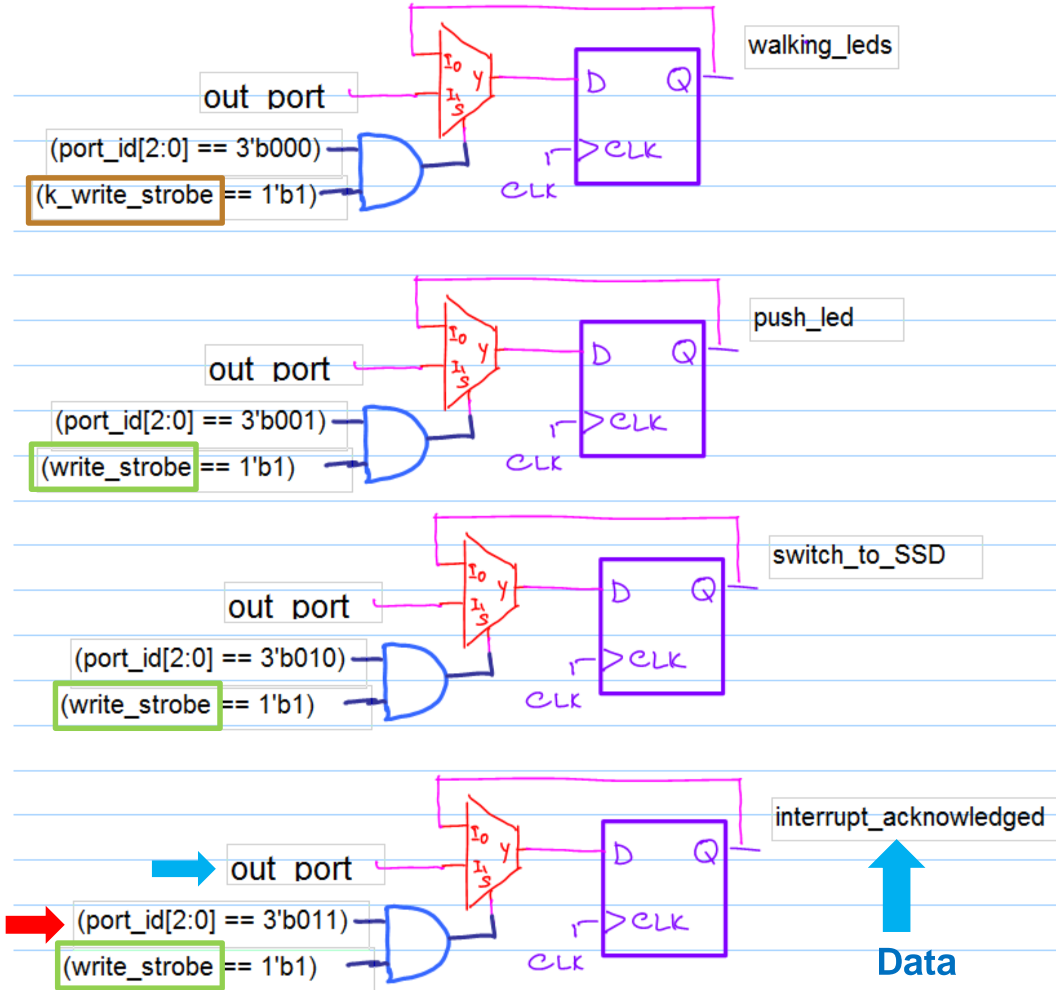
```
OUTPUTK 10000000'b, running_led_output ;
;LED Pattern * - - - - - - -
```

```
INPUT    sB, pb_input    ;
OUTPUT   sB, pb_led_output;
```

```
INPUT   sC, Switch_Data_Input;
OUTPUT  sC, Switch_Data_SSD_Output;
```

```
OUTPUT   s1, interrupt_acknowledged;
OUTPUT   s0, interrupt_acknowledged;
```

Data Address



OUTPUT and OUTPUTK instructions

OUTPUT sX, pp
OUTPUT sX, (sY)
Page 74
KCPSM6_User_Guide_30Sept14

Constant-Optimised Output Ports
OUTPUTK kk, p
Page 75-78
KCPSM6_User_Guide_30Sept14

All Picoblaze instructions are 18-bits long (no more, no less).

OUTPUT sX, pp
6-bit Opcode 4 8

OUTPUTK kk, p
6-bit Opcode 8 4

To facilitate specifying an 8-bit constant directly as data (to be sent out through the output port), the port _ID field in the OUTPUTK was reduced from 8 bits (pp) to 4 bits (p).

Section D Part 2

Here fabric logic prioritizes the prevailing interrupt requests and on behalf of the highest prevailing request, it fills a jump address in a register called "interrupt_cause_jump_address".

The ISR simply reads this and jumps to it.

Interrupt without Polling (Part 2 of the lab)

- When the Interrupt is driven High, the Picoblaze does not know who caused the interrupt. Polling mechanism can take substantial time if there are many interrupting devices. The latency caused may be undesirable or unacceptable.

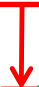
- The top design in the fabric logic can deposit the exact jump address corresponding to the highest priority request currently prevailing in a register called say "interrupt_cause_jump_address".

- In the first few lines of the **ISR** (in the **.psm** file), you can read this "interrupt_cause_jump_address" through an input port (into say **sY** register) and make a jump to this address indirectly using the Jump to an Indirect Address instruction
  JUMP@ (sX, sY)

- We use the ADDRESS directive to place sections of the ISR routines in such a way that the Picoblaze directly jumps to the start of the required section (without polling to see who caused the interrupt).

Lines from prom_interrupt_no_polling_picoblaze.psm

```
;*********************************************
;Interrupt Vector
;*********************************************

        ADDRESS 3A0                                  ;
ISR:    LOAD sF, 03                                  ;
        INPUT sA, interrupt_cause_jump_address       ;
        JUMP@ (sF, sA)                               ;


        ADDRESS 3B0                                  ;
ISR_PB: OUTPUT  s1, interrupt_acknowledged           ;
        OUTPUT  s0, interrupt_acknowledged           ;
        INPUT  sB, pb_input                          ;
        OUTPUT sB, pb_led_output                     ;
        RETURNI ENABLE                               ;
```



ADDRESS 3A0

ADDRESS 3B0

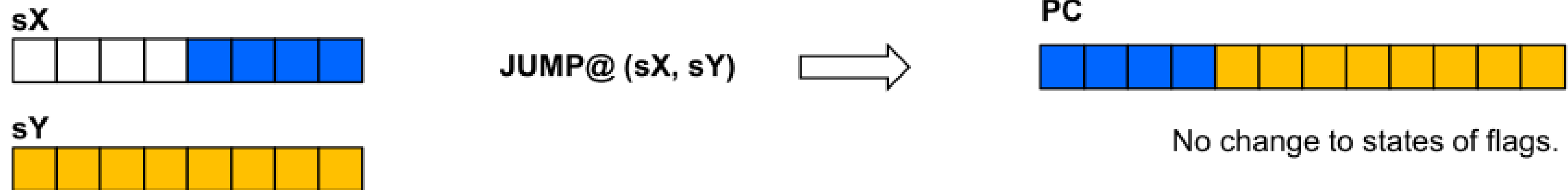
ADDRESS 3C0

ADDRESS 3D0

JUMP@ (sX, sY)

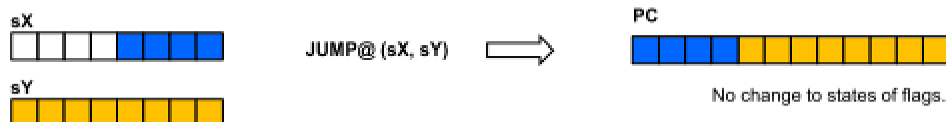
Indirect Jump

The 'JUMP@ (sX, sY)' is an unconditional JUMP which forces the program counter (PC) to the address defined by the contents of the 'sX' and 'sY' registers.



JUMP@ (sX, sY)

The 'JUMP@ (sX, sY)' is an unconditional JUMP which forces the program counter (PC) to the address defined by the contents of the 'sX' and 'sY' registers.



The 12-bit address is formed of the lower 4-bits of the 'sX' register and all 8-bits of the 'sY' register. The upper 4-bits of 'sX' are ignored and the contents of both registers are unaffected by the operation. There is no restriction on which registers can be used but it would be common coding practice to assign an adjacent pair such as 'sB' and 'sA'.

Since the destination address is defined by the contents of the registers this is powerful instruction but also has the potential to be dangerous! You are entirely responsible for writing a program in which the computed address presented by the pair of registers corresponds with a valid location within your physical program space. The KCPSM6 assembler can do nothing to prevent you computing an inappropriate address but it does provide a mechanism to enable you to determine the addresses associated with line labels as shown in the following example.

Example This example assumes that a user selects an option from a menu by providing a numerical ASCII character in the range "1" to "4" (this range could easily be extended). The program reads this character, converts it to a value in the range 0 to 3 and then jumps to the appropriate routine of 'choice'.

```
INPUT s0, selection_port
COMPARE s0, "1"
JUMP Z, choice1
COMPARE s0, "2"
JUMP Z, choice2
COMPARE s0, "3"
JUMP Z, choice3
COMPARE s0, "4"
JUMP Z, choice4
```

```
       LOAD sB, menu'upper
       LOAD sA, menu'lower
       INPUT s0, selection_port
       SUB s0, "1"
       ADD sA, s0
       ADDCY sB, 00
       JUMP@ (sB, sA)
menu:  JUMP choice1
       JUMP choice2
       JUMP choice3
       JUMP choice4
```

Without the 'JUMP@' instruction the menu would be implemented by a sequential series of compare and jumps (as shown on the right) which does not scale very well but is suitable when there is a small number of choices. Using the 'JUMP@' can help when there are lots of choices and also means that the execution time is the same regardless of the selection being made.

The KCPSM6 assembler provides 'upper and 'lower attributes that can be used with labels to define the 8-bit constants to be loaded into the registers. These abstracts of the LOG file show how the upper and lower parts of the address are resolved into 'kk' values.

```
7B4  01B07    LOAD sB, 07[menu'upper]
7B5  01ABB    LOAD sA, BB[menu'lower]
```

```
7BB  22862  menu: JUMP 862[choice1]
```

Hint - The 'upper and 'lower attributes can also be Used to derive 'kk' values for use in other instructions Such as 'ADD sX, kk' or 'COMPARE sX, kk'.

© Copyright 2010-2014 Xilinx

XILINX.

Lines from the TOP design in the fabric logic (interrupt_no_polling_picoblaze.v)

Later assignments override earlier assignments!
What if we change the order of the top 3 “if” statements?
And what if we swap the positions of the top three “ifs” with the bottom three “ifs”?

```
if(BtnL | BtnU | BtnD | BtnR) //Push Button Interrupt Condition
begin
    interrupt_pb <= 1'b1;
    interrupt_cause_jump_address <= 8'hB0;  //ISR for Push Button Interrupt starts at 0x3C0
end

if(sw_data != switch_to_SSD) //Set interrupt_sw if the current Switch Data is different from
begin
    interrupt_sw <= 1'b1;
    interrupt_cause_jump_address <= 8'hC0;  //ISR for Switch Data Interrupt starts at 0x3B0
end

if(divclk[23:0] == 24'h000000) //Walking LED Interrupt Condition Check
begin
    interrupt_slow_clk <= 1'b1;
    interrupt_cause_jump_address <= 8'hD0;  //Walking LED Interrupt starts at 0x3D0
end

// clear the interrupt_pb or interrupt_sw or interrupt_slow_clk based on the interrupt_acknc

if(interrupt_acknowledged == 8'b00000001)   // Indicates Push Button Interrupt was acknowled
begin
    interrupt_pb <= 1'b0;
end
else if(interrupt_acknowledged == 8'b00000010) // Indicates Switch Interrupt was acknowledge
begin
    interrupt_sw <= 1'b0;
end
else if(interrupt_acknowledged == 8'b00000100) // Indicates Walking LED Interrupt was acknow
begin
    interrupt_slow_clk <= 1'b0;
end
```

Note 8’hB0

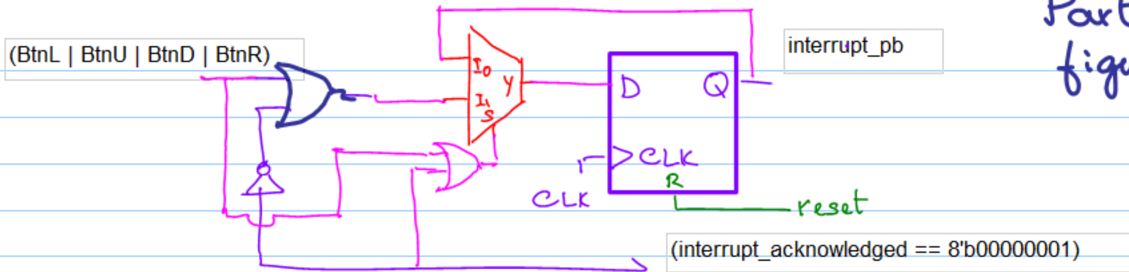
Note 8’hC0

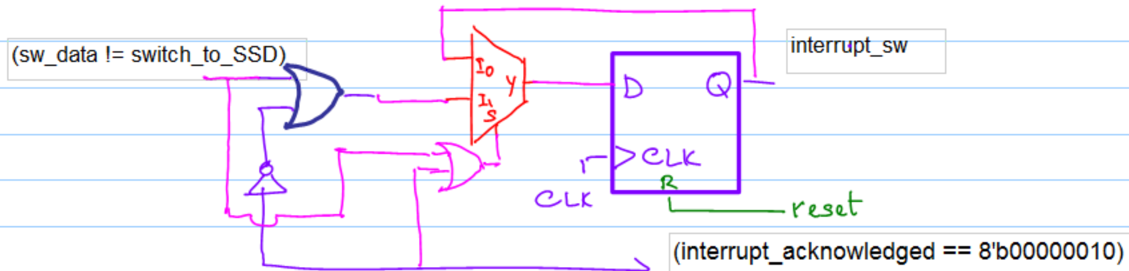
Note 8’hD0

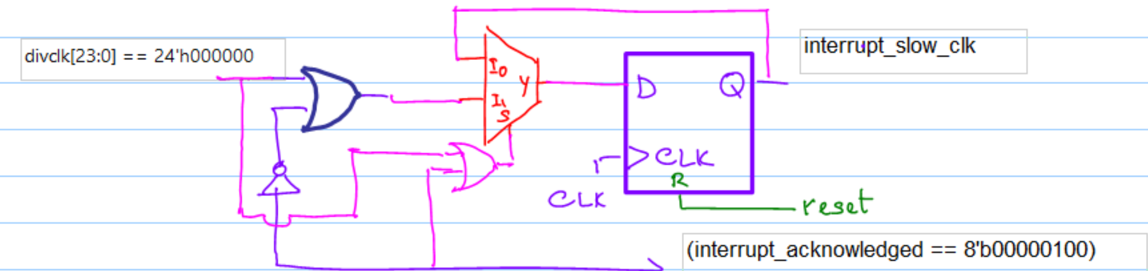
Generating Interrupt Requests in Fabric Logic

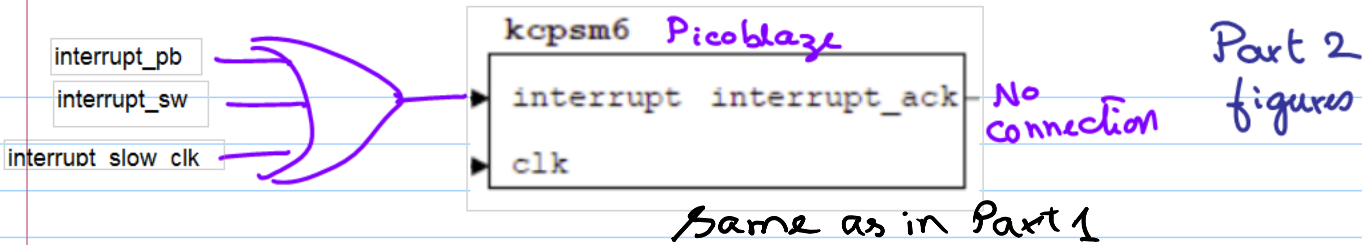
Part 2 figures

Same as in Part 1!

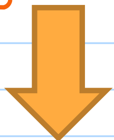


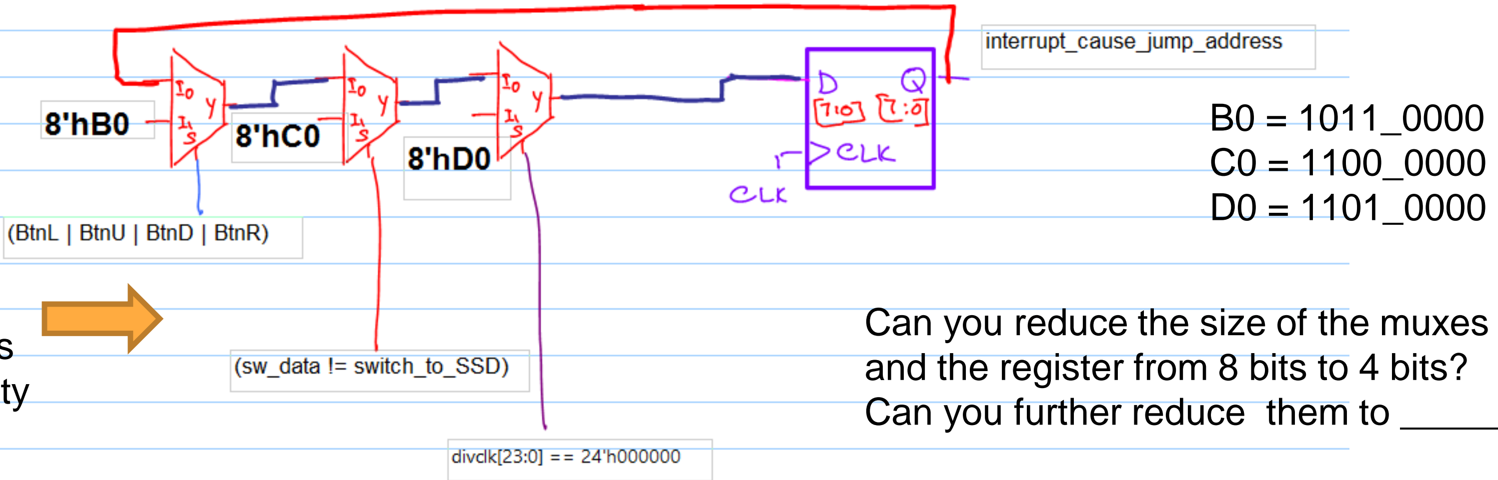






only in Part 2





B0 = 1011_0000
C0 = 1100_0000
D0 = 1101_0000

Order of muxes
Indicates priority

Can you reduce the size of the muxes and the register from 8 bits to 4 bits?
Can you further reduce them to _______ ?

Section E

ISR part of Walking LEDs pattern generation
A different method of coding using
an indirect jump instruction

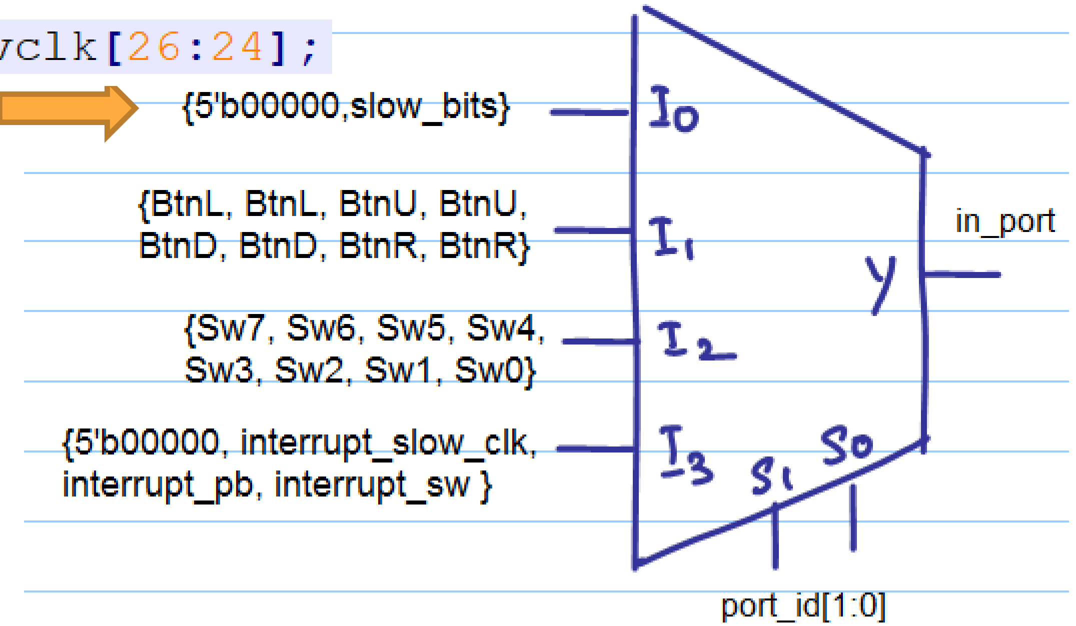
JUMP@ (sX, sY)

Indirect Jump

Page 89
KCPSM6_User_Guide_30Sept14

```
CONSTANT slow_clk,   00    ; port00 used for loading info of slow_clk
  INPUT sD, slow_clk
assign slow_bits   =  divclk[26:24];
```

Four Input ports




```
always @ (*)

begin

    case (port_id[1:0])
        2'b00 : in_port <= {5'b00000,slow_bits};
        2'b01 : in_port <= {BtnL, BtnL, BtnU, BtnU, BtnD, BtnD, BtnR, BtnR};
        2'b10 : in_port <= {Sw7, Sw6, Sw5, Sw4, Sw3, Sw2, Sw1, Sw0};
        2'b11 : in_port <= {5'b00000, interrupt_slow_clk, interrupt_pb, interrupt_sw };
    endcase
end
```

Extract from prom_interrupt_polling_picoblaze.psm

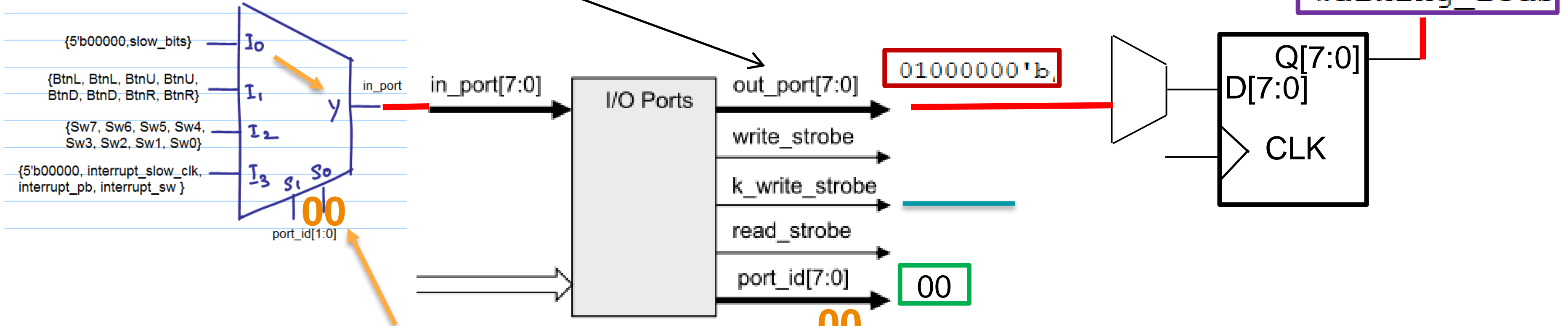
```
                INPUT sD, slow_clk

compare_seven:  COMPARE sD, 07
                JUMP NZ, compare_six
                OUTPUTK 10000000'b, running_led_output      ; LED Pattern * - - - - - - -
                RETURNI ENABLE

compare_six:    COMPARE sD, 06
                JUMP NZ, compare_five
                OUTPUTK 01000000'b, running_led_output      ; LED Pattern - * - - - - - -
                RETURNI ENABLE

compare_five:   COMPARE sD, 05
                JUMP NZ, compare_four
                OUTPUTK 00100000'b, running_led_output      ; LED Pattern - - * - - - - -
                RETURNI ENABLE

compare_four:   COMPARE sD, 04
                JUMP NZ, compare_three
                OUTPUTK 00010000'b, running_led_output      ; LED Pattern - - - * - - - -
                RETURNI ENABLE

compare_three:  COMPARE sD, 03
                JUMP NZ, compare_two
                OUTPUTK 00001000'b, running_led_output      ; LED Pattern - - - - * - - -
                RETURNI ENABLE

compare_two:    COMPARE sD, 02
                JUMP NZ, compare_one
                OUTPUTK 00000100'b, running_led_output      ; LED Pattern - - - - - * - -
                RETURNI ENABLE

compare_one:    COMPARE sD, 01
                JUMP NZ, compare_zero
                OUTPUTK 00000010'b, running_led_output      ; LED Pattern - - - - - - * -
                RETURNI ENABLE

compare_zero:   ;COMPARE sD, 00 ; well this is the last choice
                ;JUMP NZ, exit  ; it is wrong to go back to the main program improperly, y
                OUTPUTK 00000001'b, running_led_output      ; LED Pattern - - - - - - - *
                RETURNI ENABLE
```

Overhead control lines
These can be avoided
by using the
indirect jump instruction.

```
compare_six:    COMPARE sD, 06
                JUMP NZ, compare_five
                OUTPUTK 01000000'b, running_led_output
                RETURNI ENABLE
```

Since the remaining are just two instructions
(and no more no less) , we can do an indirect jump
based on SD, if slow bits read into SD are thoughtfully arranged.

```
// 'k_write_strobe' is used to qualify all writes to general output ports.
if (k_write_strobe == 1'b1)
begin
  // Write to output_port at port address 001 binary
  if (port_id[2:0] == 3'b000)
  begin
      walking_leds <= out_port;
  end
```




```
INPUT sD, slow_clk

CONSTANT slow_clk,                00        ; port00 used for loading info of slow_clk

CONSTANT running_led_output,      00        ; port00 used for outputting running led info to LEDs

          OUTPUTK 01000000'b, running_led_output     ; LED Pattern - * - - - - - -
```

OUTPUTK means use the k_write_strobe

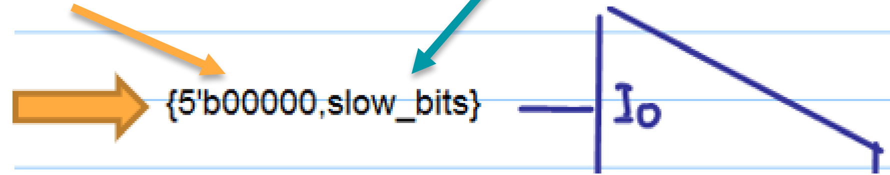
```
CONSTANT slow_clk,   00    ; port00 used for loading info of slow_clk
  INPUT sD, slow_clk
```

```
assign slow_bits  =  divclk[26:24];
```

Port address 00

8 combinations
==➔ 8 LED patterns representing the walking LED pattern

Port data {5’b00000,slow_bits}



Does any of the following choices for the data read into the sD register of the Picoblaze facilitate an indirect jump using jump @(sX,sY) to go to the exact pair of instructions among the 8 pairs of instructions?

Consider
{4’b0000,slow_bits,1’b0}
or
{3’b000,slow_bits,2’b00}

Is it better to have {4’b1111,slow_bits,1’b0}

Then the 8 combinations are
F0, F2, F4, F6, F8, FA, FC, FE

```
LOAD    sE, 03
INPUT   sD, slow_clk
JUMP@  (sE, sD)
```

```
            ADDRESS 3F0
compare_zero:  OUTPUTK 00000001'b, running_led_output  ; LED Pattern - - - - - - - *
               RETURNI ENABLE

compare_one:   OUTPUTK 00000010'b, running_led_output  ; LED Pattern - - - - - - * -
               RETURNI ENABLE

compare_two:   OUTPUTK 00000100'b, running_led_output  ; LED Pattern - - - - - * - -
               RETURNI ENABLE

compare_three: OUTPUTK 00001000'b, running_led_output  ; LED Pattern - - - - * - - -
               RETURNI ENABLE

compare_four:  OUTPUTK 00010000'b, running_led_output  ; LED Pattern - - - * - - - -
               RETURNI ENABLE

compare_five:  OUTPUTK 00100000'b, running_led_output  ; LED Pattern - - * - - - - -
               RETURNI ENABLE

compare_six:   OUTPUTK 01000000'b, running_led_output  ; LED Pattern - * - - - - - -
               RETURNI ENABLE

compare_seven: OUTPUTK 10000000'b, running_led_output  ; LED Pattern * - - - - - - -
               RETURNI ENABLE
```

Section F

Priority among interrupts

No nested priority here

So, only fixed priority?

No rotating priority?

Can a higher priority requestor mask or starve a lower priority requestor?

5x2 = 10 bit files were provided for you to tryout

Like in the GCD lab and other labs, the TA's .bit files will have the DOT points in SSDs flashing.

```
assign Dp = divclk[25];
```

But in students-completed designs, the dot points should be turned off.

Frankly, you will not be able to tell the difference between the `_polling` and `_no_polling` when you tryout, as latency reduction in the case of `_no_polling` is too small for human detection. One needs to use a *chipscope* (an on-chip logic analyzer) to discern.

LX16

test_nexys3_Verilog_LX16.v

Normal Priority (PBLP)
interrupt_polling_picoblaze_PBLP_TAs_LX16.bit
interrupt_no_polling_picoblaze_PBLP_TAs_LX16.bit

PB Highest Priority (PBHP)
interrupt_polling_picoblaze_PBHP_TAs_LX16.bit
interrupt_no_polling_picoblaze_PBHP_TAs_LX16.bit

LX45

test_nexys3_Verilog_LX45.v

Normal Priority (PBLP)
interrupt_polling_picoblaze _PBLP_TAs_LX45.bit
interrupt_no_polling_picoblaze _PBLP_TAs_LX45.bit

PB Highest Priority (PBHP)
interrupt_polling_picoblaze_PBHP_TAs_LX45.bit
interrupt_no_polling_picoblaze_PBHP_TAs_LX45.bit

Normal Priority (PBLP) and PB Highest Priority (PBHP)

In this lab, we have three interrupts. What happens if all of them or two of them go active together (simultaneously)?

```
if(BtnL | BtnU | BtnD | BtnR)          interrupt_pb <= 1'b1;
if(sw_data != switch_to_SSD)           interrupt_sw <= 1'b1;
if(divclk[23:0] == 24'h000000)         interrupt_slow_clk <= 1'b1;
```

Notice that only the `interrupt_pb` can last long (as long as one or more push buttons are pressed). `(divclk[23:0] == 24'h000000)` is active for one clock at a time. `(sw_data != switch_to_SSD)` is active for a very short time (time it takes to recognize the interrupt and to serve him) which may of the order of 20 to 30 clocks. So, for a human to notice, a higher priority request should be continuously on and be starving the lower priority requestor.

By design, we override and halt the display of walking LEDs pattern, when one or more push buttons are pressed. So `interrupt_pb` wins over `interrupt_slow_clk` anyways! So, the competition is between the `interrupt_pb` and the `interrupt_sw`.

Normal Priority (PBLP) and PB Highest Priority (PBHP) continued ..

So, for this lab, if we make the `interrupt_pb` the lowest priority interrupt **(PBLP)**, both `interrupt_pb` and `interrupt_sw` seem to work together with neither seemingly overriding the other. We just call this the "**Normal Priority (PBLP)**".

And, if the `interrupt_pb` is given higher priority over the `interrupt_sw` **(PBHP)**, then only the `interrupt_pb` works and `interrupt_sw` is ignored, when both are active. We call this "**PB Highest Priority (PBHP)**".

Note that in the normal priority (PBLP) case, whenever there is a change in switches, the `interrupt_sw` is honored and the SSDs are updated momentarily and the `interrupt_pb` service resumes. So for human eye, both services seem to go on simultaneously.

You can experiment the two cases. Hold one or more push buttons pressed with one hand, flip some switches with the other hand, and see if the SSDs respond.

What controls the relative priority?

In the Part 1 of the lab (**interrupt_polling part**), it is the order of polling at the beginning of the ISR, which controls the order of priority. So, in this part, it is the software method of controlling the priority.

In the Part 2 of the lab (**interrupt_no_polling part**), it is the prioritization logic in the fabric logic, which decides whether to write 8’hB0 or 8’hC0 or 8’hD0 in the interrupt_cause_jump_address register.

Extract from prom_interrupt_polling_picoblaze.psm

Does this create Normal priority (PBLP) or PBHP? ______________

```
ISR:    ADDRESS 3C0
; Initial polling to find the cause. The order of polling creates an implicit priority
    INPUT sA, interrupt_cause          ; Load the Interrupt cause into sA register
    AND sA, 00000010'b                 ; Check to see if Push Button Interrupt is enabled
    JUMP NZ, ISR_PB                    ; If non zero, Jump to Push Button ISR
    INPUT sA, interrupt_cause          ; Load the Interrupt cause into sA register
    AND sA, 00000001'b                 ; Check to see if SW change Interrupt is enabled
    JUMP NZ, ISR_SW                    ; If non zero, Jump to Switch Button ISR
    JUMP ISR_WALKING                   ; Else Jump to Walking LED Service Routine
```

Extract from interrupt_no_polling_picoblaze.v

Does this create Normal priority (PBLP) or PBHP? ____________

```
if(BtnL | BtnU | BtnD | BtnR)
begin
    interrupt_pb <= 1'b1;
    interrupt_cause_jump_address <= 8'hB0;
    //ISR for Push Button Interrupt starts at 0x3C0
end

if(sw_data != switch_to_SSD)
begin
    interrupt_sw <= 1'b1;
    interrupt_cause_jump_address <= 8'hC0;
    //ISR for Switch Data Interrupt starts at 0x3B0
end

if(divclk[23:0] == 24'h000000)
begin
    interrupt_slow_clk <= 1'b1;
    interrupt_cause_jump_address <= 8'hD0;
    //ISR for Walking LED Interrupt starts at 0x3D0
end
```

Section G

Lab Part 1 and Part 2

What is given ?
and
What do you need to complete?

Lab Part 1 and Part 2

This is a two-part lab.
Two .zip files (one for each part) are given.

Each .zip file contains a .psm in the "assembly" subdirectory and .v and .ucf files in the design_files subdirectory like in the case of the other Picoblaze labs.
As you know, .psm file is the assembly language program and a .v file defines the fabric logic.
Please add ROM_form.v, and the the assembler .exe file (kcpsm6.exe) to the assembly subdirectory for each part from your other projects. Similarly, please add the Picoblaze design Verilog file kcpsm6.v to the design files subdirectory. When you assemble a .psm file, say prom_interrupt_polling_picoblaze.psm, it produces prom_interrupt_polling_picoblaze.v, which you need to transfer to the design_files subdirectory and create Xilinx project there.

It is important to go through the video lecture posted along with these slides before attempting to work on the lab.

The hope is that you would understand fully the 4 files (one .v file and one .psm file for each part) by the end of this lab!

Lab Part 1

prom_interrupt_polling_picoblaze.psm ← Given in completed form
interrupt_polling_picoblaze.v ← Given in completed form

1.1 Change .psm file to change the walking LEDs part of the code and reduce 8 pairs of overhead lines (compare and perform conditional jump lines) by using an indirect jump instruction as per **Section E** of this handout. You need to change the following part of the .v (fabric logic file) accordingly.
1.2 Assemble the .psm file.
1.3 Synthesize the .v file (which instantiates the Picoblaze and the PROM). Generate the .bit file. Download the .bit file to Nexys-3 and demonstrate to your TA.

```
always @ (*)

begin

    case (port_id[1:0])
        2'b00 : in_port <= {5'b00000,slow_bits};
        2'b01 : in_port <= {BtnL, BtnL, BtnU, BtnU, BtnD, BtnD, BtnR, BtnR};
        2'b10 : in_port <= {Sw7, Sw6, Sw5, Sw4, Sw3, Sw2, Sw1, Sw0};
        2'b11 : in_port <= {5'b00000, interrupt_slow_clk, interrupt_pb, interrupt_sw };
    endcase
end
```

```
assign Dp = divclk[25]; // The dot point on each SSD flashes for the TA's design.
                        // divclk[25] (~1.5Hz) = (100MHz / 2**26)
                    // count the number of flashes for a minute, you should get about 90
```

We have already commented this line out in your exercise file.

Lab Part 2

prom_interrupt_no_polling_picoblaze.psm ← Nearly empty file is given. You create this. Refer to **Section D Part 2**
interrupt_no_polling_picoblaze.v ← Given in completed form

1.1 copy prom_interrupt_polling_picoblaze.psm (the original .psm file of lab Part 1) to prom_interrupt_no_polling_picoblaze.psm
1.2 Revise it to read interrupt_cause_jump_address in place of interrupt_cause. So in the .psm, you will change the following existing line

```
CONSTANT interrupt_cause,              03        ; port03 used for loading info of interrupt cause
```

to

```
CONSTANT interrupt_cause_jump_address, 03        ; port03 used for loading info of interrupt cause
```

And you will change the first part of the ISR from the left-side excerpt to the right-side sequence.

```
        ADDRESS 3C0
ISR:    ; Initial polling to find the cause.
        INPUT sA, interrupt_cause
        AND sA, 00000010'b
        JUMP NZ, ISR_PB
        INPUT sA, interrupt_cause
        AND sA, 00000001'b
        JUMP NZ, ISR_SW
        JUMP ISR_WALKING
```

```
        ADDRESS 3A0
ISR:    LOAD sF, 03
        INPUT sA, interrupt_cause_jump_address
        JUMP@ (sF, sA)
```

You will also use `ADDRESS 3B0`, `ADDRESS 3C0`, and `ADDRESS 3D0` for positioning the three sections of ISR.
1.3 Assemble the .psm file. Synthesize the .v file. Generate .bit file. Demonstrate to your TA.

Demonstrate to your TA, your understanding of relative priorities among the requestors (a) in Part 1 and (b) in Part 2 and how to change their relative priorities in each part, if needed.

Refer to Section F of this handout and also check your Part 1 and Part 2 codes. Complete the three blanks below with `interrupt_pb`, **`interrupt_sw`**, and **`interrupt_slow_clk`**, in appropriate order to indicate their relative priorities.

HP = Highest Priority, **MP** = Medium Priority, and **LP** = Lowest Priority

Part 1 code as completed by you has priorities in the following order.

________________ (HP), ________________ (MP), ________________ (LP).

This is an implementation of __________________ (Normal Priority PBLP/PBHP).

Part 2 code as completed by you has priorities in the following order.

________________ (HP), ________________ (MP), ________________ (LP).

This is an implementation of __________________ (Normal Priority PBLP/PBHP).

Congratulations on going through this lengthy lab handout and completing the lab!