*Lab 2: Hello Avalon-MM

How to design an IP communicating with Avalon-MM interface?

Written by Geng You Chen
(gengyouchen@gmail.com)

*Go much deeper than Lab 1

* Similarly to Lab 1, now in Lab 2 you need to design a SOPC system running three C/C++ functions at the same time:
  * Function 1: According to the value read from 7-Segment Displays every three seconds, repeatly print some information on Character LCD.
  * Function 2: Set new value to 7-Segment Displays every 100 milliseconds. The new value is increasing one per 100 milliseconds.
  * Function 3: Control Red LEDs according to Toggle Switches anytime.
* Similarly to Lab 1, you will use µC/OS-II for multi-task programming. Because µC/OS-II is RTOS (Real Time Operating System), not general purpose operating system such as Windows and Linux, it guarantees higher priority tasks "always preempt" CPU from lower priority tasks. Since Function 3 never hands over CPU to other functions, you must set Function 3 to the lowest priority (the biggest priority number).
* In Lab 2, you need to design a controller (IP) for 7-Segment Displays. Because your IP will be attached to Altera Avalon Bus, your IP need to communicate with Avalon-MM interface.

*What is Memory Mapping?

* In your C/C++ program, when you dynamically allocate some data in the memory (e.g. SDRAM), it will return a pointer which stores the memory address of your data. Next time, you can easily access your data through this memory address.
* Similarly, in Altera SOPC Builder, when you attach a hardware to the bus, Nios II EDS will copy its base address to a header file named “system.h”. Therefore, your C/C++ program can easily access interface registers of the hardware through this base address plus some offsets you need.
* You can’t tell the difference between a address pointed to a real memory and a address pointed to a hardware. Because the I/O of the hardware are memory-mapped to the I/O of the real memory by the Altera Avalon Bus, Nios II CPU doesn’t need any special instruction to control any hardware, just treats entire bus as a real memory and does memory accessing.
* Another viewpoint is that there’s no difference between a real memory and a normal hardware because a memory controller is also a hardware. When you think you are accessing a certain memory address, in reality, you are accessing a base address of a memory controller plus offsets.

*With the Memory Mapping, why we still need IORD/IOWR?

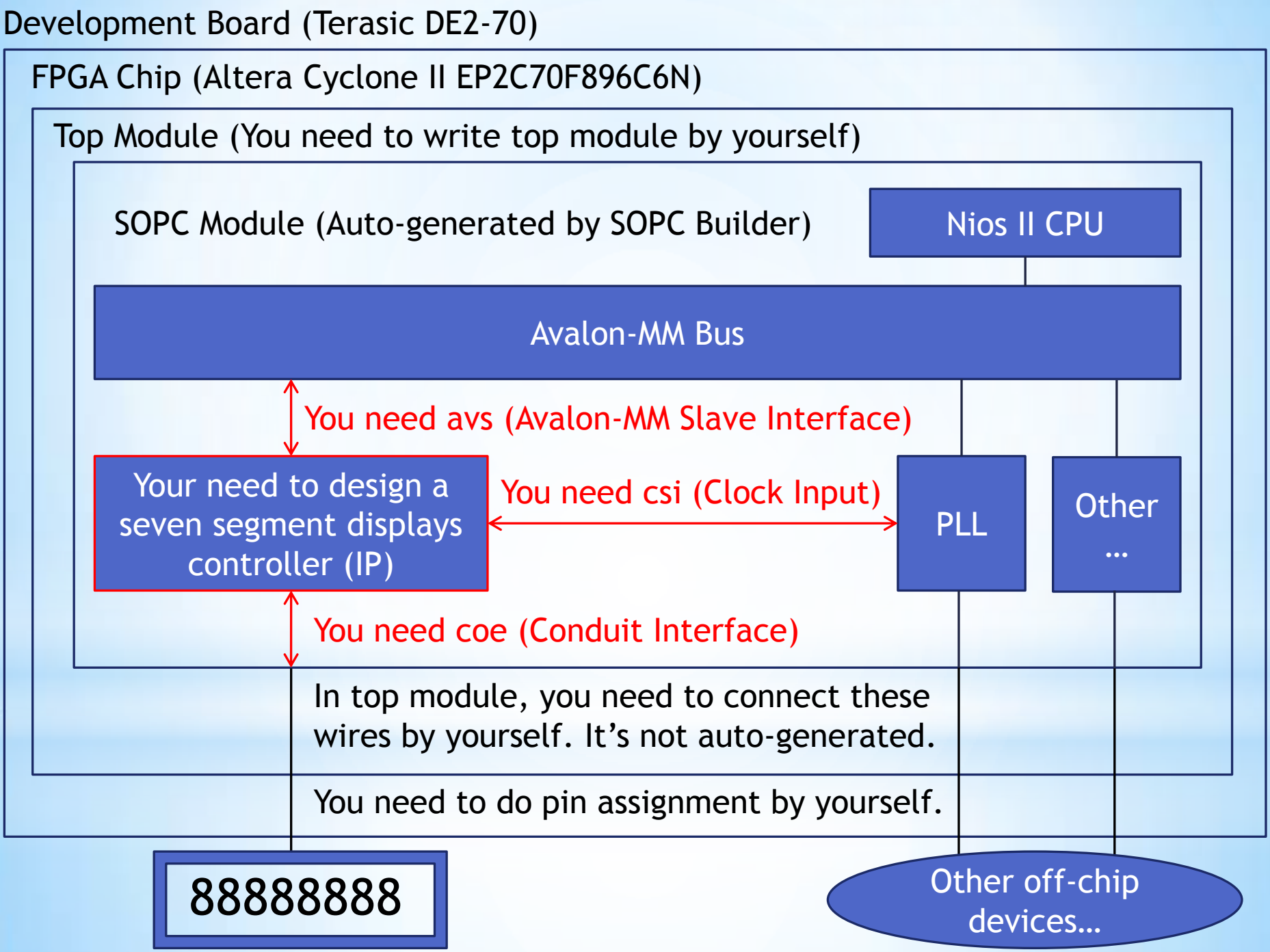
* Compared with the clock of CPU, accessing memories needs a very long time, especially off-chip memories (e.g. SDRAM). To improve the speed, inside CPU, there's a cache which stores frequently used data. For example, assume the cache can store only 1 data, then:
  1. Now CPU wants to read X, so it tries to find X in the cache. However, the cache is empty. Therefore, it moves X from the memory to the cache and reads it.
  2. Now CPU wants to read X. It can find X in the cache and directly read it.
  3. Now CPU wants to write X. It can find X in the cache and directly write it.
  4. Now CPU wants to read X. It can find X in the cache and directly read it.
  5. Now CPU wants to read Y, so it tries to find Y in the cache. However, there's no Y in the cache and the cache is full. Therefore, it moves X from the cache back to the memory. Then, it moves Y from the memory to the cache and reads it.
* For a real memory, the cache mechanism is good because it decreases the need of accessing memory. However, some memory accessing is not for real memory, but memory-mapped to the hardware I/O. These memory accessing cannot be held in the cache, otherwise, the result of I/O may be incorrect or delayed. Therefore, we need to use IORD and IOWR to bypass the cache mechanism .

*Avalon-MM Interface

* For C/C++ programmers, they can see that all interface registers of your IP are directly attached to the Avalon-MM Bus. They can access each interface registers according to bus addresses.
* However, it's not true for IP designer. You cannot define interface registers as I/O pins of IP module. To talk with the Avalon-MM Bus, your I/O pins need to follow the Avalon-MM Interface:
  * Define an input pin named write_n (only one bit) and an input pin named writedata (can be multiple bits as you wish). The Avalon-MM Bus will use write_n to tell your IP that now the C/C++ program calls a IOWR function and the value of writedata is the data in IOWR(base, offset, data).
  * Define an input pin named read_n (only one bit) and an output pin named readdata (can be multiple bits as you wish). The Avalon-MM Bus will use read_n to tell your IP that now the C/C++ program calls a IORD function. After a clock cycle, the Avalon-MM Bus will read the value of readdata and use this value as the return value of IORD function.
  * Define a input pin named address (can be multiple bits as you wish). When the C/C++ program calls IORD or IOWR, the value of address is the offset in IORD(base, offset) or IOWR(base, offset, data).
* Your IP needs to read or write your interface registers according to I/O pins (write_n, writedata, read_n, readdata, address) of the Avalon-MM Interface.

*All interfaces in Avalon Bus

* Clock Interfaces
  * Clock Input (csi): You need clk and reset_n for hardware synchronization.
  * Clock Output (cso): PLL uses cso to output the special-frequency clocks.
* Avalon-MM Interfaces
  * Avalon-MM Master (avm): CPU uses avm to be a master of all IPs on bus.
  * Avalon-MM Slave (avs): Your IP uses avs to be a slave of CPU on the bus.
* Interrupt Interfaces
  * Interrupt Sender (ins): Timer uses ins to interrupt CPU from current task.
  * Interrupt Receiver (inr): CPU uses inr to accept interrupt from IPs on bus.
* Avalon-MM Tristate Interfaces
  * Avalon-MM Tristate Master (atm): Tristate bridge uses atm to talk to its IPs.
  * Avalon-MM Tristate Slave (ats): Flash controller attached to bridge uses ats.
* Avalon-ST Interfaces
  * Avalon-ST source (aso): Some IPs use aso to directly send data bypass CPU.
  * Avalon-ST sink (asi): Some IPs use asi to directly receive data bypass CPU.
* Conduit Interfaces (coe): You need export to connect outside SOPC module.


Development Board (Terasic DE2-70)
FPGA Chip (Altera Cyclone II EP2C70F896C6N)
Top Module (You need to write top module by yourself)
SOPC Module (Auto-generated by SOPC Builder)
Nios II CPU
Avalon-MM Bus
You need avs (Avalon-MM Slave Interface)
Your need to design a seven segment displays controller (IP)
You need csi (Clock Input)
PLL
Other ...
You need coe (Conduit Interface)
In top module, you need to connect these wires by yourself. It's not auto-generated.
You need to do pin assignment by yourself.
88888888
Other off-chip devices...

To design a seven segment controller, I need to define the I/O pins of my module. Here I name 3 interfaces: s0 (Clock Input), s1(Avalon-MM Slave) and s2(Conduit).

```
module seg7 (

    // Define "s0" as "Clock Input"
    input csi_s0_clk,
    input csi_s0_reset_n,

    // Define "s1" as "Avalon Memory Mapped Slave"
    input avs_s1_write_n,
    input avs_s1_read_n,
    input [2:0] avs_s1_address,
    input [3:0] avs_s1_writedata,
    output reg [3:0] avs_s1_readdata,

    // Define "s2" as "Conduit"
    output [6:0] coe_s2_export_oHEX0_D,
    output [6:0] coe_s2_export_oHEX1_D,
    output [6:0] coe_s2_export_oHEX2_D,
    output [6:0] coe_s2_export_oHEX3_D,
    output [6:0] coe_s2_export_oHEX4_D,
    output [6:0] coe_s2_export_oHEX5_D,
    output [6:0] coe_s2_export_oHEX6_D,
    output [6:0] coe_s2_export_oHEX7_D,
    output coe_s2_export_oHEX7_DP,
    output coe_s2_export_oHEX0_DP,
    output coe_s2_export_oHEX1_DP,
    output coe_s2_export_oHEX2_DP,
    output coe_s2_export_oHEX3_DP,
    output coe_s2_export_oHEX4_DP,
    output coe_s2_export_oHEX5_DP,
    output coe_s2_export_oHEX6_DP
);
```

Inside my IP module, I create 8 interface registers for 8 seven-segment displays. Each seven-segment displays can shows: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, b, C, d, F. Therefore, I need 4 bits to represent these 16 conditions.

```
// Create interface registers
reg [3:0] num0;
reg [3:0] num1;
reg [3:0] num2;
reg [3:0] num3;
reg [3:0] num4;
reg [3:0] num5;
reg [3:0] num6;
reg [3:0] num7;
```

I read or write 8 interface registers according to I/O pins of Avalon-MM Slave Interface.
(The coding style is bad for real IC design, but I think it's good for educational purpose)

```
// Read or write interface registers according to Avalon-MM interface
always @ (posedge csi_s0_clk or negedge csi_s0_reset_n) begin
   if (csi_s0_reset_n == 1'b0) begin
      num0 <= 4'b0000;
      num1 <= 4'b0000;
      num2 <= 4'b0000;
      num3 <= 4'b0000;
      num4 <= 4'b0000;
      num5 <= 4'b0000;
      num6 <= 4'b0000;
      num7 <= 4'b0000;
   end else if (avs_s1_read_n == 1'b0) begin
      case (avs_s1_address)
         3'b000: avs_s1_readdata <= num0;
         3'b001: avs_s1_readdata <= num1;
         3'b010: avs_s1_readdata <= num2;
         3'b011: avs_s1_readdata <= num3;
         3'b100: avs_s1_readdata <= num4;
         3'b101: avs_s1_readdata <= num5;
         3'b110: avs_s1_readdata <= num6;
         3'b111: avs_s1_readdata <= num7;
      endcase
   end else if (avs_s1_write_n == 1'b0) begin
      case (avs_s1_address)
         3'b000: num0 <= avs_s1_writedata;
         3'b001: num1 <= avs_s1_writedata;
         3'b010: num2 <= avs_s1_writedata;
         3'b011: num3 <= avs_s1_writedata;
         3'b100: num4 <= avs_s1_writedata;
         3'b101: num5 <= avs_s1_writedata;
         3'b110: num6 <= avs_s1_writedata;
         3'b111: num7 <= avs_s1_writedata;
      endcase
   end
end
```

I use my binary_to_seven_segment_converter sub-module to decode the binary representation used by interface registers to the seven-segment representation. Also, I don't want decimal points so I turn off all of them (active-low signals). These output signals will be exported to the top module (outside SOPC module).

```
  // Convert binary codes to seven segment codes
  binary_to_seven_segment_converter u0(.seven_segment(coe_s2_export_oHEX0_D), .binary(num0));
  binary_to_seven_segment_converter u1(.seven_segment(coe_s2_export_oHEX1_D), .binary(num1));
  binary_to_seven_segment_converter u2(.seven_segment(coe_s2_export_oHEX2_D), .binary(num2));
  binary_to_seven_segment_converter u3(.seven_segment(coe_s2_export_oHEX3_D), .binary(num3));
  binary_to_seven_segment_converter u4(.seven_segment(coe_s2_export_oHEX4_D), .binary(num4));
  binary_to_seven_segment_converter u5(.seven_segment(coe_s2_export_oHEX5_D), .binary(num5));
  binary_to_seven_segment_converter u6(.seven_segment(coe_s2_export_oHEX6_D), .binary(num6));
  binary_to_seven_segment_converter u7(.seven_segment(coe_s2_export_oHEX7_D), .binary(num7));

  // Turn off all decimal point
  assign coe_s2_export_oHEX0_DP = 1'b1;
  assign coe_s2_export_oHEX1_DP = 1'b1;
  assign coe_s2_export_oHEX2_DP = 1'b1;
  assign coe_s2_export_oHEX3_DP = 1'b1;
  assign coe_s2_export_oHEX4_DP = 1'b1;
  assign coe_s2_export_oHEX5_DP = 1'b1;
  assign coe_s2_export_oHEX6_DP = 1'b1;
  assign coe_s2_export_oHEX7_DP = 1'b1;
endmodule
```

The binary_to_seven_segment_converter module used by my IP is defined as follows:

```
module binary_to_seven_segment_converter (
    output reg [6:0] seven_segment,
    input [3:0] binary
);
    always @ (binary) begin
        case (binary)
            4'b0000: seven_segment <= 7'b1000000;
            4'b0001: seven_segment <= 7'b1111001;
            4'b0010: seven_segment <= 7'b0100100;
            4'b0011: seven_segment <= 7'b0110000;
            4'b0100: seven_segment <= 7'b0011001;
            4'b0101: seven_segment <= 7'b0010010;
            4'b0110: seven_segment <= 7'b0000010;
            4'b0111: seven_segment <= 7'b1111000;
            4'b1000: seven_segment <= 7'b0000000;
            4'b1001: seven_segment <= 7'b0010000;
            4'b1010: seven_segment <= 7'b0001000;
            4'b1011: seven_segment <= 7'b0000011;
            4'b1100: seven_segment <= 7'b1000110;
            4'b1101: seven_segment <= 7'b0100001;
            4'b1110: seven_segment <= 7'b0000110;
            default: seven_segment <= 7'b0001110;
        endcase
    end
endmodule
```

To let my IP be a component in SOPC Builder, I click Project -> New component.

Altera SOPC Builder - Lab2_SOPC.sopc (C:\EECS2002\Lab2\Lab2_SOPC.sopc)

File Edit Module System View Tools Nios II Help

System Contents | System Generation

Component Library

Project
- New component...

Library
- Avalon Verification Suite
- Bridges and Adapters
- Interface Protocols
- Legacy Components
- Memories and Memory Controllers
- Merlin Components
- Peripherals
- PLL
- Processor Additions
- Processors
- SLS
- Video and Image Processing

New... | Edit... | Add...

Target

Device Family: Cyclone II

Clock Settings

Name	Source	MHz
clock_from_board	External	50.0
clock_to_CPU	PLL.c0	100.0
clock_to_SDRAM	PLL.c1	100.0

Add | Remove

Use	Connections	Module	Description	Clock	Base
☑		PLL	Avalon ALTPLL	[inclk_interfa...	
		pll_slave	Avalon Memory Mapped Slave	clock_from...	0x09411040
☑		on_chip_memory	On-Chip Memory (RAM or ROM)	[clk1]	
		s1	Avalon Memory Mapped Slave	clock_to_C...	0x09408000
☑		SSRAM_controller	Cypress CY7C1380C SSRAM	[clk]	
		s1	Avalon Memory Mapped Tristate Slave	clock_to_C...	0x09200000
☑		SDRAM_controller	SDRAM Controller	[clk]	
		s1	Avalon Memory Mapped Slave	clock_to_S...	0x04000000
☑		Flash_controller	Flash Memory Interface (CFI)		
		s1	Avalon Memory Mapped Tristate Slave	clock_to_C...	0x08800000

Remove | Edit... | Address Map... | Filters... | Filter: Default

Warning: **CPU**: Custom Instruction components can be edited through the Component Editor.

Warning: **CPU**: Disabling the assign CPUID control register value manually will no longer auto-assigns unique control register value. This option will always be turned on with de

Info: **Flash_controller**: Flash memory capacity: 8.0 MBytes (8388608 bytes).

Info: **toggle_switches_controller**: PIO inputs are not hardwired in test bench. Undefined values will be read from PIO inputs during simulation.

Exit | Help | Prev | Next | Generate

Component Editor - new_component_hw.tcl

File Templates

Introduction | HDL Files | Signals | Interfaces | HDL Parameters | Library Info

Introduction

The component editor is a tool for creating and modifying *IP components*. You can start with an existing Verilog or VHDL design block, and the component editor helps you create an IP component for that block. Alternatively, you can use the component editor to manually define the interfaces to custom logic.

The output of the component editor is a file with name ending **_hw.tcl**. For an input HDL file named ***filename*.v** or ***filename*.vhd**, the component editor saves a component file named ***filename*_hw.tcl** in the directory where the source HDL file is located.

To make your component appear in the component library, configure the IP Search Path to include the path to the ***_hw.tcl** file. To use your component in other projects or to share your component with other designers, a good practice is to move all HDL source files and the ***_hw.tcl** file into a separate directory and store it in a central location. You can also include supporting files, such as software drivers and simulation files, in the same directory.

You can provide software driver files for your component hardware. If you will use your component in Nios II processor systems, you will likely need to create a software driver to integrate into the Nios II board support package (BSP).

This introduction is not important,
so I directly click the next button.

Info: Your component is not based on HDL. An instance of this component will export reversed-direction signals to co

Warning: The component has no signals.

Help | Prev | Next | Finish...

Component Editor - seg7_hw.tcl*

File Templates

Introduction | HDL Files | Signals | Interfaces | HDL Parameters | Library Info

About HDL Files

HDL Files:

File Name	Synth	Sim	Top
seg7.v	✓	✓	◉

Add... | Remove | Reanalyze HDL Files

Create HDL Template...

Top Level Module: seg7

I add the Verilog file of my IP hardware here.

Warning: **s1**: **readdata[4]** width must be in {8, 16, 32, 64, 128, 256, 512, 1024} for dynamic addressing

Warning: **s1**: **writedata[4]** width must be in {8, 16, 32, 64, 128, 256, 512, 1024} for dynamic addressing

Help | Prev | Next | Finish...

Component Editor - seg7_hw.tcl*

File Templates

Introduction | HDL Files | Signals | Interfaces | HDL Parameters | Library Info

About Signals

Name	Interface	Signal Type	Width	Direction
csi_s0_clk	s0	clk	1	input
csi_s0_reset_n	s0	reset_n	1	input
avs_s1_write_n	s1	write_n	1	input
avs_s1_read_n	s1	read_n	1	input
avs_s1_address	s1	address	3	input
avs_s1_writedata	s1	writedata	4	input
avs_s1_readdata	s1	readdata	4	output
coe_s2_export_oHEX0_D	s2	export	7	output
coe_s2_export_oHEX1_D	s2	export	7	output
coe_s2_export_oHEX2_D	s2	export	7	output
coe_s2_export_oHEX3_D	s2	export	7	output
coe_s2_export_oHEX4_D	s2	export	7	output
coe_s2_export_oHEX5_D	s2	export	7	output
coe_s2_export_oHEX6_D	s2	export	7	output
coe_s2_export_oHEX7_D	s2	export	7	output
coe_s2_export_oHEX7_DP	s2	export	1	output
coe_s2_export_oHEX0_DP	s2	export	1	output
coe_s2_export_oHEX1_DP	s2	export	1	output
coe_s2_export_oHEX2_DP	s2	export	1	output
coe_s2_export_oHEX3_DP	s2	export	1	output
coe_s2_export_oHEX4_DP	s2	export	1	output
coe_s2_export_oHEX5_DP	s2	export	1	output
coe_s2_export_oHEX6_DP	s2	export	1	output

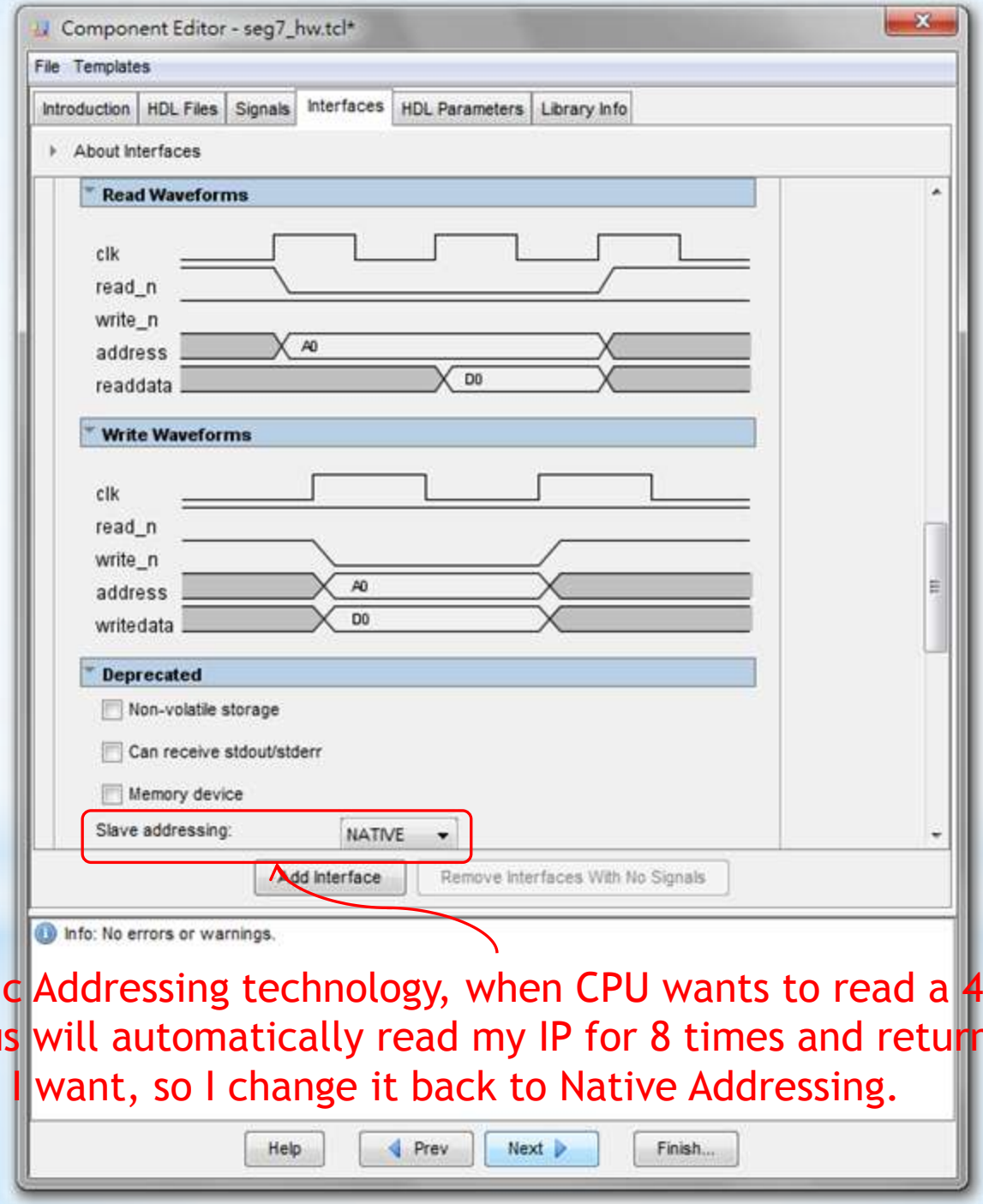
Add Signal | Remove Signal

Warning: **s1**: **readdata[4]** width must be in {8, 16, 32, 64, 128, 256, 512, 1024} for dynamic addressing

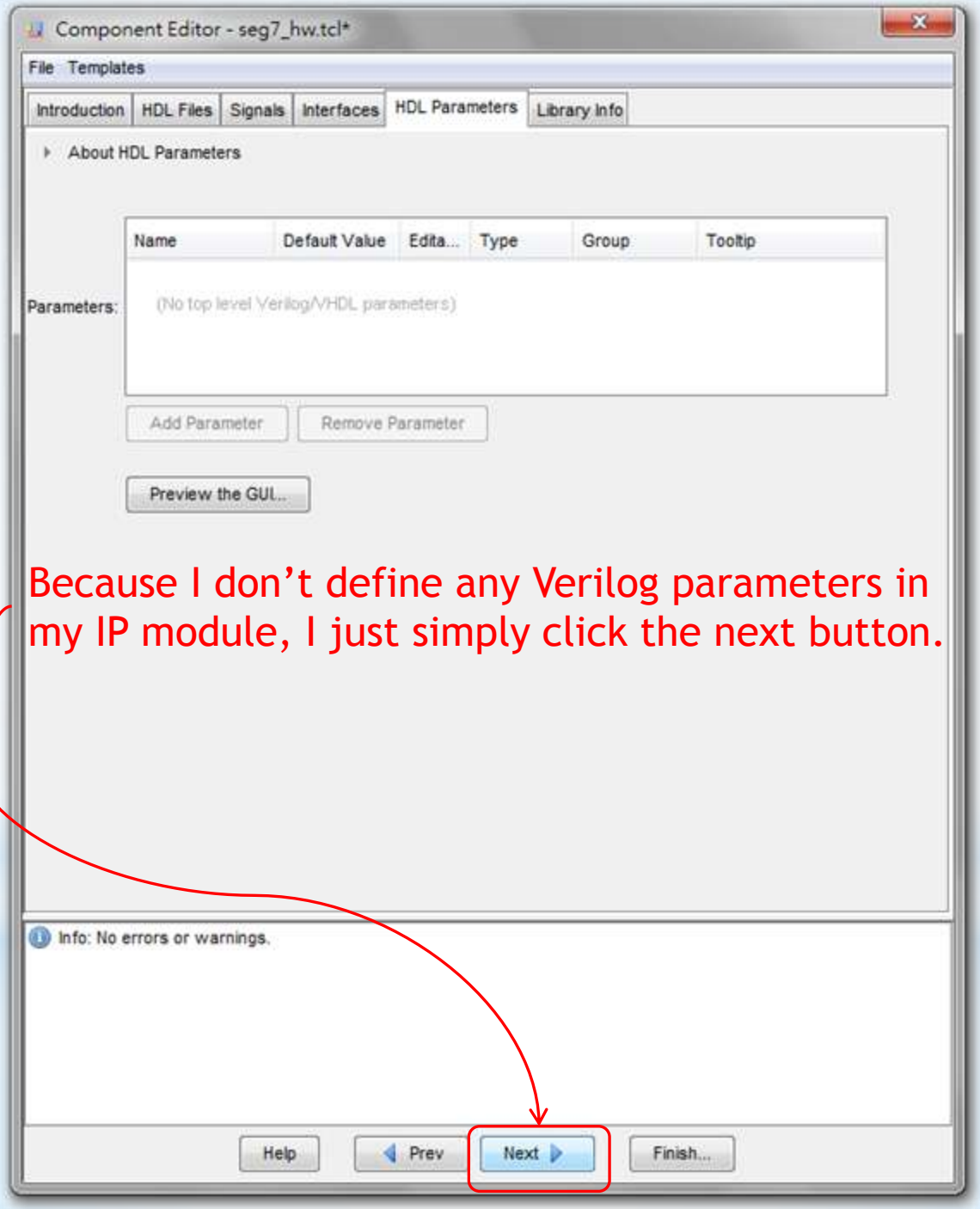
Warning: **s1**: **writedata[4]** width must be in {8, 16, 32, 64, 128, 256, 512, 1024} for dynamic addressing

The signals here are correct, so I directly click the next button.

Help | Prev | Next | Finish...



For new Dynamic Addressing technology, when CPU wants to read a 4-bit output of my IP, Avalon Bus will automatically read my IP for 8 times and return a 32-bits value. This is not what I want, so I change it back to Native Addressing.



Because I don't define any Verilog parameters in my IP module, I just simply click the next button.

Component Editor - seg7_hw.tcl*

File Templates

Introduction | HDL Files | Signals | Interfaces | HDL Parameters | Library Info

About Library Info

Parameters

Name:	seg7
Display Name:	Seven Segment Displays Controller
Version:	1.0
Group:	EECS2002
Description:	This IP can controll 7-segment displays on DE2-70 board.
Created by:	Geng You Chen
Icon:	
Documentation:	Title / URL

For readability, I fill some information about my IP hardware.

Info: No errors or warnings.

Help | Prev | Next | Finish...

Now I can see my IP in component library. I simply click my IP to add it to my SOPC module.

Altera SOPC Builder - Lab2_SOPC.sopc (C:\EECS2002\Lab2\Lab2_SOPC.sopc)

File Edit Module System View Tools Nios II Help

System Contents | System Generation

Component Library

- Project
  - New component...
- Library
  - Avalon Verification Suite
  - Bridges and Adapters
  - EECS2002
    - Seven Segment Displays Controller
  - Interface Protocols
  - Legacy Components
  - Memories and Memory Controllers
  - Merlin Components
  - Peripherals
  - PLL
  - Processor Additions
  - Processors
  - SLS

New... Edit... Add...

Target

Device Family: Cyclone II

Clock Settings

Name	Source	MHz
clock_from_board	External	50.0
clock_to_CPU	PLL.c0	100.0
clock_to_SDRAM	PLL.c1	100.0

Add Remove

Use	Connections	Module	Description	Clock	Base
✓		PLL	Avalon ALTPLL	[inclk_interfa...	
		pll_slave	Avalon Memory Mapped Slave	clock_from...	0x
✓		on_chip_memory	On-Chip Memory (RAM or ROM)	[clk1]	
		s1	Avalon Memory Mapped Slave	clock_to_C...	0x
✓		SSRAM_controller	Cypress CY7C1380C SSRAM	[clk]	
		s1	Avalon Memory Mapped Tristate Slave	clock_to_C...	0x
✓		SDRAM_controller	SDRAM Controller	[clk]	
		s1	Avalon Memory Mapped Slave	clock_to_S...	0x
✓		Flash_controller	Flash Memory Interface (CFI)		
		s1	Avalon Memory Mapped Tristate Slave	clock_to_C...	0x

Remove Edit... Address Map... Filters... Filter: Default

Warning: CPU: Custom Instruction components can be edited through the Component Editor.

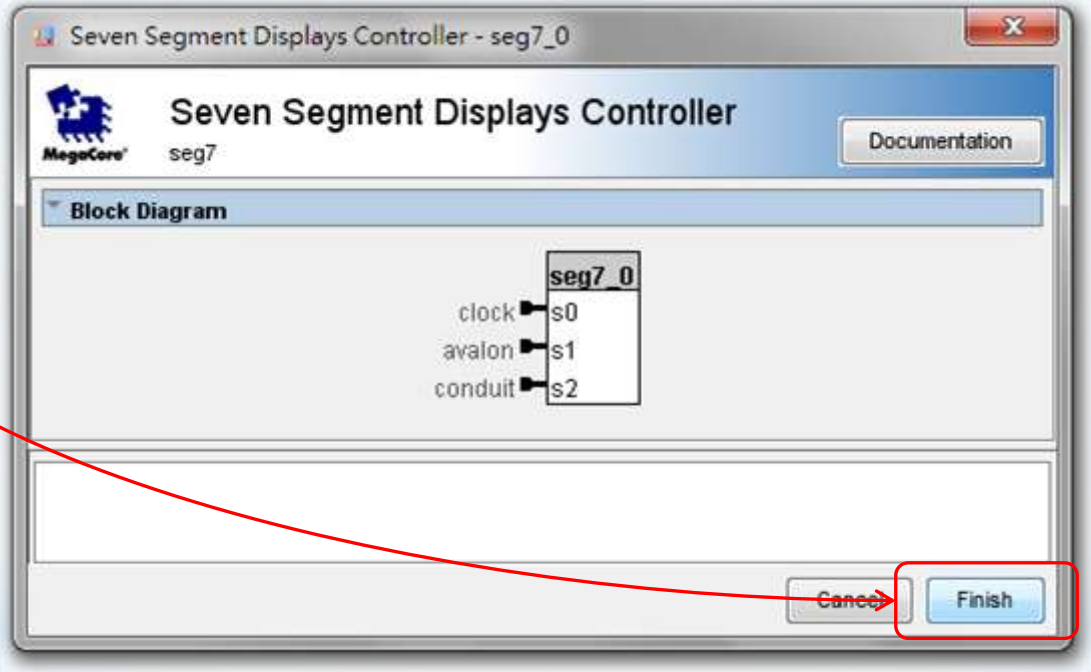
Warning: CPU: Disabling the assign CPUID control register value manually will no longer auto-assigns unique control register value. This option will always be turned on with de

Info: Flash_controller: Flash memory capacity: 8.0 MBytes (8388608 bytes).

Info: toggle_switches_controller: PIO inputs are not hardwired in test bench. Undefined values will be read from PIO inputs during simulation.

Exit Help Prev Next Generate

The default value works, so I directly click the finish button.



I rename my new controller to seven_segment_displays_controller and use clock_to_CPU as its clock source.

Altera SOPC Builder - Lab2_SOPC.sopc* (C:\EECS2002\Lab2\Lab2_SOPC.sopc)

File Edit Module System View Tools Nios II Help

System Contents | System Generation

Component Library

Project
- New component...

Library
- Avalon Verification Suite
- Bridges and Adapters
- EECS2002
  - Seven Segment Displays Controller
- Interface Protocols
- Legacy Components
- Memories and Memory Controllers
- Merlin Components
- Peripherals
- PLL
- Processor Additions
- Processors
- SLS

New... Edit... Add...

Target

Device Family: Cyclone II

Clock Settings

Name	Source	MHz
clock_from_board	External	50.0
clock_to_CPU	PLL.c0	100.0
clock_to_SDRAM	PLL.c1	100.0

Add Remove

Use	Connections	Module	Description	Clock
☑		green_LEDs_controller	PIO (Parallel I/O)	[clk]
		s1	Avalon Memory Mapped Slave	clock_to_CPU
☑		system_clock_timer	Interval Timer	[clk]
		s1	Avalon Memory Mapped Slave	clock_to_CPU
☑		timestamp_timer	Interval Timer	[clk]
		s1	Avalon Memory Mapped Slave	clock_to_CPU
☑		sysid	System ID Peripheral	[clk]
		control_slave	Avalon Memory Mapped Slave	clock_to_CPU
☑		seven_segment_displays_controller	Seven Segment Displays Controller	
		s1	Avalon Memory Mapped Slave	clock_to_CPU

Remove Edit... Address Map... Filters... Filter: Default

Warning: CPU: Custom instruction components can be edited through the Component Editor.

Warning: CPU: Disabling the assign CPUID control register value manually will no longer auto-assigns unique control register value. This option will always be turned on with default value se

Info: Flash_controller: Flash memory capacity: 8.0 MBytes (8388608 bytes).

Info: toggle_switches_controller: PIO inputs are not hardwired in test bench. Undefined values will be read from PIO inputs during simulation.

Exit Help Prev Next Generate

I need to connect the exported signals (Conduit Interface) of SOPC module to I/O pins of top module. These wires are not auto-generated by SOPC Builder.

```
    // For toggle_switches_controller:
    .in_port_to_the_toggle_switches_controller(iSW),

    // For push_button_switches_controller:
    .in_port_to_the_push_button_switches_controller(iKEY),

    // For red_LEDs_controller:
    .out_port_from_the_red_LEDs_controller(oLEDR),

    // For green_LEDs_controller:
    .out_port_from_the_green_LEDs_controller(oLEDG),

    // the_seven_segment_displays_controller
    .coe_s2_export_oHEX0_DP_from_the_seven_segment_displays_controller(oHEX0_DP),
    .coe_s2_export_oHEX0_D_from_the_seven_segment_displays_controller(oHEX0_D),
    .coe_s2_export_oHEX1_DP_from_the_seven_segment_displays_controller(oHEX1_DP),
    .coe_s2_export_oHEX1_D_from_the_seven_segment_displays_controller(oHEX1_D),
    .coe_s2_export_oHEX2_DP_from_the_seven_segment_displays_controller(oHEX2_DP),
    .coe_s2_export_oHEX2_D_from_the_seven_segment_displays_controller(oHEX2_D),
    .coe_s2_export_oHEX3_DP_from_the_seven_segment_displays_controller(oHEX3_DP),
    .coe_s2_export_oHEX3_D_from_the_seven_segment_displays_controller(oHEX3_D),
    .coe_s2_export_oHEX4_DP_from_the_seven_segment_displays_controller(oHEX4_DP),
    .coe_s2_export_oHEX4_D_from_the_seven_segment_displays_controller(oHEX4_D),
    .coe_s2_export_oHEX5_DP_from_the_seven_segment_displays_controller(oHEX5_DP),
    .coe_s2_export_oHEX5_D_from_the_seven_segment_displays_controller(oHEX5_D),
    .coe_s2_export_oHEX6_DP_from_the_seven_segment_displays_controller(oHEX6_DP),
    .coe_s2_export_oHEX6_D_from_the_seven_segment_displays_controller(oHEX6_D),
    .coe_s2_export_oHEX7_DP_from_the_seven_segment_displays_controller(oHEX7_DP),
    .coe_s2_export_oHEX7_D_from_the_seven_segment_displays_controller(oHEX7_D),

    // System reset:
    .reset_n(cpu_reset_n)
);

endmodule
```

To let other C/C++ programmers understand the meaning of each interface registers, I provide a Register Map in my IP driver (just a simple C/C++ header).

```
#ifndef __SEG7_H__
#define __SEG7_H__

#include <io.h>

// ==================================================================
// Register Map
// ==================================================================

#define SEG7_NUM0 0
#define SEG7_NUM1 1
#define SEG7_NUM2 2
#define SEG7_NUM3 3
#define SEG7_NUM4 4
#define SEG7_NUM5 5
#define SEG7_NUM6 6
#define SEG7_NUM7 7
```

I also do Hardware Abstraction in my IP driver which provides 2 high-level functions.

```
// ==========================================================================
// Hardware Abstraction
// ==========================================================================

int get_seg7(void *base)
{
    int value = 0;
    value += IORD(base, SEG7_NUM0) * 1;
    value += IORD(base, SEG7_NUM1) * 10;
    value += IORD(base, SEG7_NUM2) * 100;
    value += IORD(base, SEG7_NUM3) * 1000;
    value += IORD(base, SEG7_NUM4) * 10000;
    value += IORD(base, SEG7_NUM5) * 100000;
    value += IORD(base, SEG7_NUM6) * 1000000;
    value += IORD(base, SEG7_NUM7) * 10000000;
    return value;
}

void set_seg7(void *base, int value)
{
    IOWR(base, SEG7_NUM0, value / 1 % 10);
    IOWR(base, SEG7_NUM1, value / 10 % 10);
    IOWR(base, SEG7_NUM2, value / 100 % 10);
    IOWR(base, SEG7_NUM3, value / 1000 % 10);
    IOWR(base, SEG7_NUM4, value / 10000 % 10);
    IOWR(base, SEG7_NUM5, value / 100000 % 10);
    IOWR(base, SEG7_NUM6, value / 1000000 % 10);
    IOWR(base, SEG7_NUM7, value / 10000000 % 10);
}

#endif
```

I change the default hello_ucosii.c file into the following 3 tasks C/C++ program. Noticed that for µC/OS-II, the biggest priority number has the lowest priority.

```
#include <io.h>
#include <stdio.h>
#include "includes.h"
#include "system.h"
#include "seg7.h"
OS_STK s1[2048], s2[2048], s3[2048];
void f1(void *p) {
    while (1) {
        printf("EECS2002 Lab2\nName: 9760111\n");
        OSTimeDlyHMSM(0, 0, 3, 0);
        printf("call: get_seg7()\nreturn: %d\n", get_seg7((void*) SEVEN_SEGMENT_DISPLAYS_CONTROLLER_BASE));
        OSTimeDlyHMSM(0, 0, 3, 0);
    }
}
void f2(void *p) {
    int i = 0;
    while (1) {
        set_seg7((void*) SEVEN_SEGMENT_DISPLAYS_CONTROLLER_BASE, ++i);
        OSTimeDlyHMSM(0, 0, 0, 100);
    }
}
void f3(void *p) {
    while (1) {
        IOWR(RED_LEDS_CONTROLLER_BASE, 0, IORD(TOGGLE_SWITCHES_CONTROLLER_BASE, 0));
    }
}
int main() {
    OSTaskCreateExt(f1, 0, (void*) &s1[2047], 1, 1, s1, 2048, 0, 0);
    OSTaskCreateExt(f2, 0, (void*) &s2[2047], 2, 2, s2, 2048, 0, 0);
    OSTaskCreateExt(f3, 0, (void*) &s3[2047], 3, 3, s3, 2048, 0, 0);
    OSStart();
    return 0;
}
```

*Review Questions

1. µC/OS-II can guarantee that the most important task in our system can response to our user within a given time. Explain it. (Hint: Preemptive) This special characteristic is useful in some real-world applications. Give at least one example of its applications. (Hint: Emergency)
2. Nios II CPU can't tell the difference between a real memory and our IP hardware. Explain it. (Hint: Memory Mapping) This characteristic may cause incorrect I/O results of our IP. Explain it. (Hint: Cache Memory) How to prevent these incorrect I/O results? (Hint: Bypass Cache)
3. In the Avalon-MM Interface, when write_n goes low, writedata becomes the data in IOWR(base, offset, data) at the same clock cycle. However, when read_n goes low, the return value of IORD is defined as readdata at the next cycle, not the current cycle. Explain it. (Hint: Bi-direction)
4. A SOPC system has a Ethernet Controller downloading a DVD movie file from webs, and a SD card controller writing this file to a 8GB SD card. Which Avalon interface may have the better performance, Avalon-ST or Avalon-MM Tristate or Avalon-MM? Why? (Hint: Data Path)