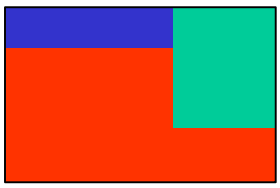
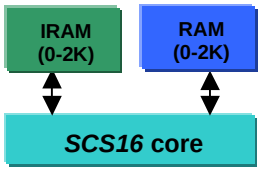
Zorba Technologies



SCS16 Core – Overview

Table Of Contents

1. Introduction

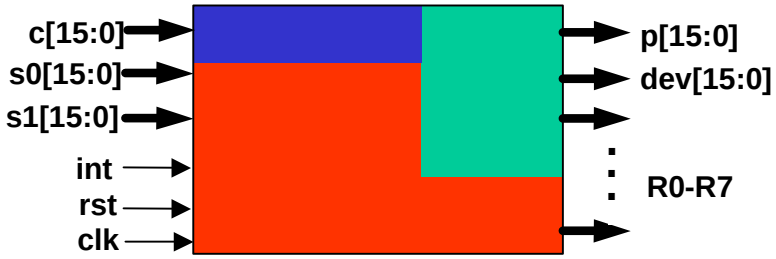


The **SCS16** soft core is a single cycle RISC nano-controller. Its target is not to replace the main controller on the Chip, but to replace FSMs and un-structured Random Logic. Since it is as responsive as FSMs to external triggers, yet, as programmable (and flexible) as a micro-controller, it can save a lot of area on the device. It does that by implementing in sequence functions traditionally implemented by a few FSMs in parallel. In addition, it shortens the design cycle and reduces cost by using software instead of hardware. This allows for more flexibility in the design, makes it possible for the Back-end phase to commence while still doing Front-end design, and greatly reduces the costs of ECO's (only one mask layer needs to be changed).
The **SCS16**'s programming language is very easy to use and has a **C-like** syntax. However, since the **SCS16** is HW driven, **each code line represents 1 clock cycle**, so that there are no surprises after compilation.
The **SCS16** is delivered with its own Development Environment, including a GUI.

Features

- Very low gate count – 20K, including Instruction and Data Memory
- Very low cost of ownership
- Harvard Architecture – 26 bit Instruction path and 16 bit Data path
- Up to 2K (11bit) of addressable memory (separate for Instruction and Data)
- Able to react to a maximum of 16 external events within 1 clock cycle
- Powerful Burst I/O Mechanism
- Parallelism – up to 5 operations per command in one clock cycle
- Single Cycle Execution for every command
- ASIC/FPGA Integration
- Vendor and Technology Independent
- Fully Synchronous and Static design
- 100MHz at 0.18 process
- Highly Flexible Design - Easily extendable instruction set (in order to support customer specific needs)
- 16 bit ALU

2. Interfaces



Inputs

c[15:0]: These 16 fast sense and react lines can be used to externally trigger processes inside the core. Among other things, they are used for data qualification and flow control. These inputs can be either active high or active low.

Related Commands:

```
wait [!]Cx                    - Flow Control
L_loop,wait_store [!]Cx…      - Burst Input data qualifier
L_loop,wait_load  [!]Cx…      - Burst Output
if ([!]Cx)…                   - Polling
```

S0, S1 [15:0]: 2 external busses for data input. These busses can also be used in burst mode. The Number of external data sources could easily be extended. Please consult us for more details.

Related Commands:

```
Rx=Sx          - Register Store
```

Interrupt: The interrupt signal is active high. There is no internal masking mechanism for the Interrupt Input. However, Interrupts could easily be masked in software by enabling/disabling this signal with an **SCS16** output and an external "AND". For more details please refer to Internal Capabilities section**.**

Related Commands:

```
rti         - Return from Interrupt to PC+1 (terminates the
              ISR execution)
```

rst: This signal resets all of the SCS16 registers as well as the PC. It is active high. The RST signal's negation should be synchronous and hence demands an active clock signal.

clk: The maximum frequency is Technology Dependent – for instance using a 0.18 process, the core can run at >100MHz (worst case). Stopping the clock preserve the state of the core. No minimum frequency specified.

Outputs

Regs_Out: There are 8x16-bit General Purpose Registers in the **SCS16**. These registers could be reflected to the outside world. For more details please refer to **Registers.**

p[15:0]: These 16 one cycle pulse signals could be used to qualify data output or for hand shaking. When activated by software, these signals are asserted for 1 clock cycle and are then negated. The pulse signals are active high. When an executed code line contains a pulse command, the pulse is activated in the cycle following the execution of this code line. When a pulse is asserted continuously for a few cycles, the result is a level signal. Pulses can be activated in parallel with the execution of other commands.

Related Commands:

```
Pulse -> Px       - Pulse Assertion
```

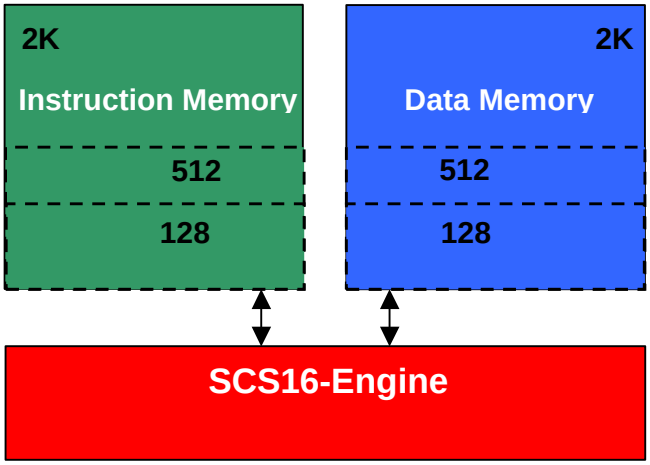
dev[15:0]: These 4x4 level signals could be used to control external devices. When activated by software, these signals are asserted and stay that way until negated by software. The commands that manipulate the Dev bus control each nibble at a time – Dev0, Dev1, Dev2, or Dev3.

Related Commands:

```
device_x -> 4'bxx1x
device_x -> 3
```

3. Memory

Memory accesses are word (16 bit) wide. However, a separate byte (LSB or MSB) could be loaded to (or stored from) the according byte in the register. All addressing modes are supported.
All **SCS16** memory interfaces are internal and the user should not concern him self with their behavior.
The sizes of the Instruction and Data Memories are easily defined by the user in the verilog simulations module.
Technology wise – The instantiated memory cells should be Synchronous.
The following scheme shows how the **SCS16** could be instantiated with different sizes of instruction and data memories. In this example we picked the 128, 512 and 2K entries options, of course, there are other possibilities.



Related Commands:

```
RAM[20 + R6]= Rx        - Memory Store
Rx = RAM[address]       - Memory Load
RAM[address]= Rx_l      - Memory Store lsbyte
Rx_h = RAM[address]     - Memory Load msbyte
RAM[R6++] = value       - Dynamic Memory Pointer
RAM[value]= Rx          - Store Immediate
RAM[R6]= value          - Store Immediate
```

4. Internal Capabilities

Interrupt There are three kinds of scenarios when responding to an **Interrupt** signal assertion:

Case 1 - While running in main level code (not a subroutine or an interrupt routine), assertion of this signal will cause a PC jump to the dedicated ISR (Interrupt Service Routine) address.

Case 2 - While running in subroutine level code (not an interrupt routine), assertion of this signal will be sampled and initiate an Interrupt pending state. The SCS16 will continue executing the subroutine. After the subroutine is completed the PC returns to the main level. After execution of the next code line, the PC jumps to the dedicated ISR (Interrupt Service Routine) address (as in the previous case)

Case 3 - While executing an ISR a new interrupt can be made pending, assuming the Interrupt signal was negated before asserted a second time (this is done in order to verify the same Interrupt request does not trigger more than one Interrupt routine execution). After the Interrupt routine has been executed, and the PC has returned to the main level, the new interrupt will be handled (as in the first case).

Registers:

1. GPRs - There are 8x16 bit Rd/Wr General Purpose Registers, **R0-R7,** which could be reflected to the outside world. Data in these registers could be manipulated either in the word (16 bit), byte, or bit levels.
Each of these registers could be pushed to (or popped from) a shadow register, in order to change the working environment's context (this is especially useful when handling an ISR or performing a subroutine). All the registers, or a few of them, could be pushed/popped in 1 clock cycle.
All the registers could be used in the following operations:

- **ALU Operations** –The registers could be used as the source or the destination for all arithmetic, logic and bit shifting operations.
- **Memory Accesses** – The registers could be used as source or destination for any memory access.
- **Bit Operations** – The registers could be used as source or destination for all bit manipulation operations.
- **Linear Search** – The registers could be used as the searched object or the intermediate register (the register that holds a value from the memory to be compared with the searched object).
- **Burst Commands** – The registers could be used as data buffers between the memory and the data bus (which, in burst-load operations, is the register's own reflection to the outside world).
- **Shadowing Commands** – In one clock cycle all the registers, or a few of them, could be Pushed into (or Popped from) the shadow registers. This helps create a

new context, which is very useful for executing ISR's and subroutines.

- **Flow Control** – The registers could be used as loop counters, or could be compared (in whole or only one bit) or checked for logic conditions of branch operations.

Memory Addressing - Only Registers R6 and R7 could be used as memory addressing pointers.

FIFO Managing - Only Registers R0 and R1 could be loaded with the FIFO status fields and used to update the FIFO's status. Please refer to **FIFO Handling** for more details.

Related Registers Commands:

```
if (Rx>Ry) {...}        - Flow Control
if (Cx) R6              - Memory Pointer
RAM[address]=Rx         - Memory Store
Rx=RAM[address]         - Memory Load
Rx=Sx                   - Register Store
RAM[R6++]=value         - Dynamic Memory Pointer
Rx=Value                - Load Immediate
RAM[value]=Rx           - Store Immediate
RAM[R6]=value           - Store Immediate
Rx=Ry                   - Move
R4=R3+R2                - ALU Operation
```

2. Shadow registers - There are 8x16 bit Shadow registers, **R0_b-R7_b,** which could be used to create a new context by using the Push/POP commands. Please refer to **Shadowing** for more details.

Related Commands:

```
push Rx Ry        - Shadowing
pop Rx            - Reconstructing the original context
```

Subroutines: A subroutine could be called from the main flow, from an ISR, or from another Subroutine. Only two levels of nesting are allowed for subroutines (either from the main flow or from an ISR).

Related Commands:

```
gosub L_sub_address     - Goto subroutine
return                  - Return to main course of program
```

Shadowing: One level of stacking is allowed for all the registers (or a few of them). When using the PUSH/POP all (or a few of) the registers are stored into (or loaded from) the shadow registers in one cycle. This helps create a new context, which is very useful for executing ISR's and subroutines.

Related Commands:

```
push Rx Ry……      - Shadowing
pop Rx            - Reconstructing the original context
```

ALU/ALUI operations: The ALU is 16-bit wide. The **SCS16** supports all normal ALU and ALU Immediate operations. This includes Arithmetic Operations, Logic Operations and Bit Shifting Operations. The operations supported are between one register to another and between a register to an immediate value, be it binary, decimal or hexadecimal.

Related Commands:

```
R4=R3+R2          - Arithmetic Operation
R4=R3+5           - Immediate Arithmetic Operation
R7=R5<<R1         - Shift Operation
R7=29<<R1         - Immediate Shift Operation
R0=R4&R6          - Logic Operation
```

Bit/Range Manipulation: The **SCS16** supports Insertion and Extraction of Various bit ranges to/from registers in accordance.

When extracting a range of bits from a register (or an immediate value) to another register, the specified bits will be moved to the object register's least significant bits.

When inserting a register (or an immediate value) to a range of bits in another register, the source value's least significant bits will be moved to the object register's specified bits.

Related Commands:

```
R4[12:10]=R2            - Bit Insertion
R5[12:10]=100           - Immediate Bit Insertion
R7=R3[3:0]              - Bit Extraction
```

CRC The SCS16 has dedicated Hardware that supports 'on the fly' CRC checking and generation. This instruction is optional. Please consult Zorba Technologies for more details.

Linear Search: The **SCS16** supports searching for a word (or part of it) in a specified memory range. This is implemented by a loop, in which each iteration compares the value of a register with the value loaded from memory to another register. One comparison is achieved in each clock cycle. The comparison's 'true' result could be defined as equal, greater than or less than. The compared address' pointer could advance in either 1 or a specified step size in each iteration.

Related Commands:

```
L_label,breakif R2>R1=RAM[R7+=R5]
L_label,breakif R2[4:0]==R1=RAM[R7++@
L_label,breakif R2>R1=RAM[R7++@
L_label,breakif R2==R4=RAM[R7++]
```

Loop: The **SCS16** supports loop statements. The loop itself does not have to start right after the loop declaration. Instead, the loop declaration states the label of the loop's starting point as well as the loop's ending point. The number of iterations could be either a register value, or an immediate one.

Related Commands:

```
loop L_start L_end R5
loop L_setp  L_step 30
```

FIFO Handling: The **SCS16** supports management of soft FIFOs in its memory. For each FIFO 3 words are stored in memory. The first one is the FIFO Manager, which includes the following fields:
Current size, maximum size and Full and Empty bits.
The second word is the Read Address – Offset from the Base Address.
The third word is the Write Address – Offset from the Base Address.
When Writing or Reading to/from the FIFO the user should take the following actions:

- Load the FIFO Manager to R0
- Load the Read/Write address to R1
- Check in R0 the Empty bit (for a Read access) or the Full bit (for a Write access)
- If FIFO status allows, continue with Reading from (or Writing to) the address in R1
- Update both registers in one cycle by using an **SCS16** FIFO handling instruction
- Write back R0 and R1 to the memory

Related Commands:

```
(R0,R1)=fifo_wr(R0,R1)  - Update statuses after a Write access
(R0,R1)=fifo_rd(R0,R1)  - Update statuses after a Read access
```

Burst I/O: The **SCS16** supports two directions of bursts.
The first is receiving data from an external device, via an Sx bus, sampling the data into a register and storing the register's contents in the memory.
The second direction is loading data from the memory into a register, and sending the register's value by connecting its outputs to an external device.
All these phases construct a pipeline, in which a data word is moved from each station to the next one in each clock cycle.
While receiving data, each iteration, usually, depends on assertion of a qualifying signal (implemented by Cx) from the sending device.
While sending data, each cycle is usually accompanied by a data qualifying pulse (implemented by Px) asserted by the **SCS16.**
While working in a 100MHz frequency (with 16 bits of data transferred each clock cycle), the **SCS16** can sustain a data rate of 1.6 GBits/Sec.

Related Commands:

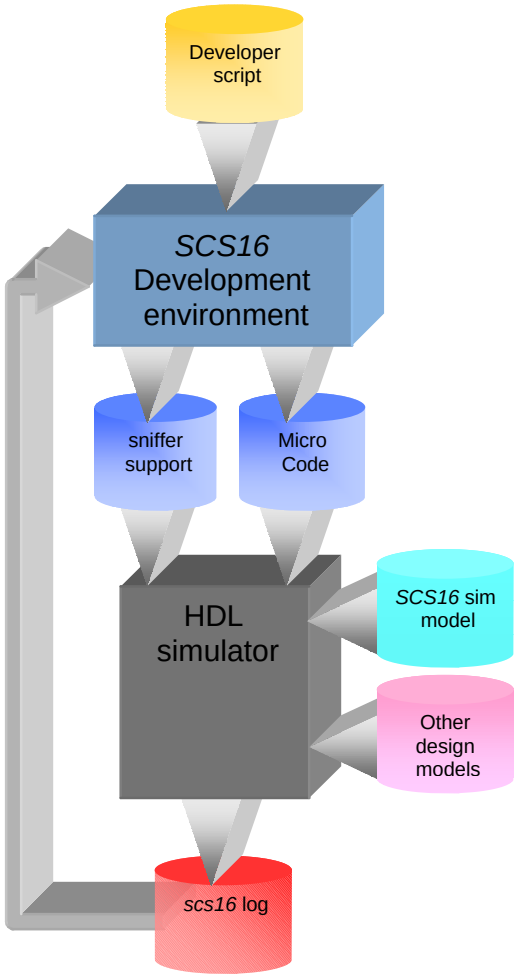
```
L_loop, wait_store !c1 RAM[R7++]=R4=s0 breakif c7
L_loop, wait_store  c4 RAM[R6++]=R5=s0 breakif !c4 pulse->p1
L_loop, wait_store  c7 RAM[R6++]=R2=s0 breakif !c7 pulse->p2
L_loop, wait_load   c2 R0=RAM[R7++]    pulse->p1   breakif !c0
```

5. The Programming Language

The programming language is C-based. It is very easy to learn and use. Since the **SCS16** is hardware driven, each code line represents 1 clock cycle, so there are no surprises after compilation. The commands are highly parallel. This means that in many cases the user can include a few instructions in one code line and these instructions will all be executed in parallel in the same clock cycle.
The programming language includes a few special tokens that represent the **SCS16**'s unique abilities, for instance, pulse, wait_store, wait_load, etc.
In case a costumer finds that a sequence of operations repeats itself quite often in his design, it is possible to add a new instruction to the **SCS16**' s instruction set. This way the sequence of operations could be implemented in one clock cycle, and executed each time the new instruction is activated.

For a detailed discussion regarding the programming language please refer to the **SCS16 user manual.**

6. The Development Environment



The **SCS16** Development Environment can operate in either of two modes:

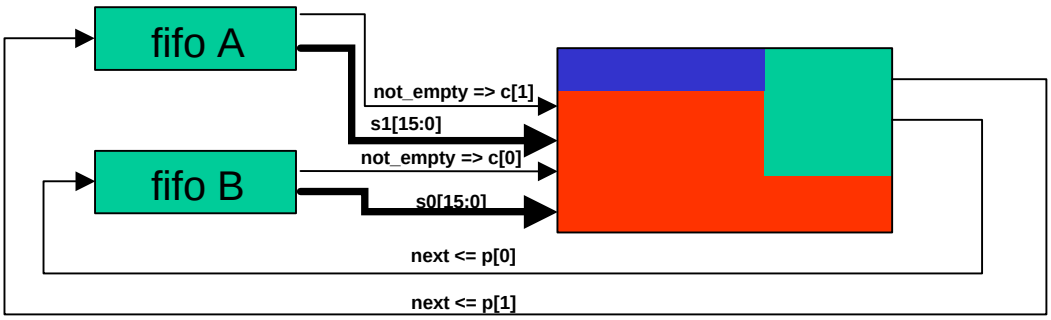
?? **Code Development and Debug –** In this mode the source file is compiled and debugged independently according to compilation rules. The **SCS16** Development environment supports line coloring and indentation, and also working in a multi-window environment. After compilation has successfully ended the compiler outputs two main files. The first one is the binary Opcode, to be loaded into the HDL Simulator's module of Instruction Memory by the 'readmemb' command. The second one is to be loaded into the HDL simulator as well in order to provide sniffing support, so that, while the HDL Simulation is running, a PC Log file is created.

?? **Simulation Review –** After running an HDL Simulation, the resulting PC Log file could be loaded back into the **SCS16** Development Environment to be reviewed and compared with the developer's source code**.** While in this mode, highlighting a code line in the source file highlights the according PC cycle in the PC Log file**,** and vice versa**.** The PC Log file also shows changes made to each **SCS16** register or interface in each PC cycle.

For a detailed discussion regarding the development environment please refer to the **SCS16 user manual.**

7. Connecting the SCS16 with other modules

Getting Data from FIFO's



In case we want to get data from external FIFOs, the **SCS16** should be connected to the FIFOs as shown in this scheme.
For a burst read, the appropriate code line should be:

```
L_label, wait_store c[1] ram[R7++]=R1=s1 breakif !c[1] pulse-> p1
```

This line should be supported by a loop declaration. What happens here is a burst. In each iteration **all** the following actions are taken:

- The data bus (S1) is sampled by register R1.
- The previous value of register R1 is stored in the memory at address R7.
- R7 is incremented
- A pulse signal is asserted in order to request that the FIFO's next word is sent

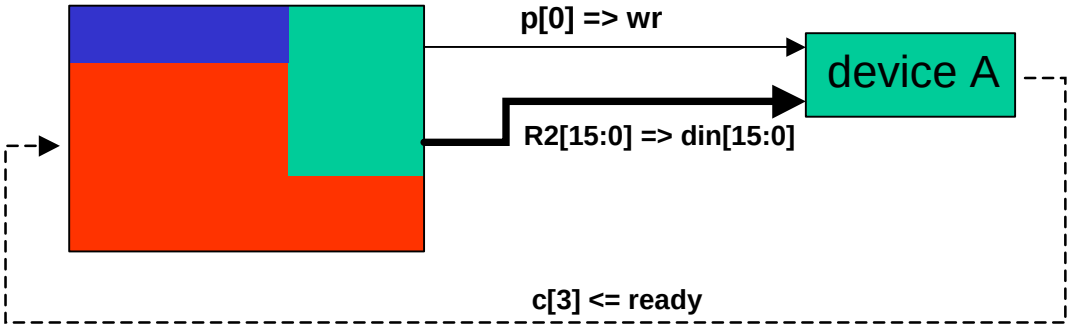
All this continues as long as the FIFO is not empty (C1 is asserted).

For only 1 iteration, on the other hand:

```
if (c[1]) {
      R1 = s1 pulse->p1
      }
```

If the FIFO isn't empty (C1 asserted), the data bus is sampled by Register R1 and a pulse signal is asserted.

Outputting/Writing Data to an External Device



In case we want to push data to an external device, the **SCS16** should be connected to the device as described in this scheme.
For a burst write, the appropriate code should be:

```
loop L_wr L_wr 10
L_wr, R2=ram[R7++] pulse-> p0
```

The process described here is, actually, a burst. The loop declaration defines the start line and the end line of the loop. Since in this case they are both the same, only one line, labeled L_wr, is executed 10 times. In each iteration **all** the following actions are taken:

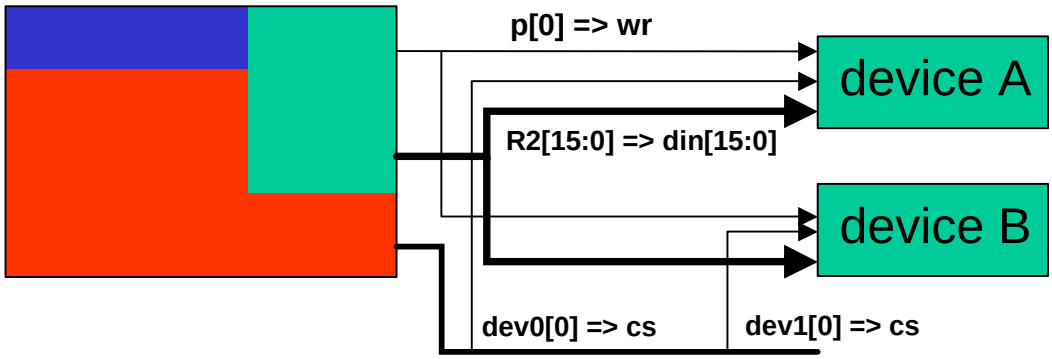
- The value stored at address R7 in the memory is loaded into register R2
- R7 is incremented
- A pulse signal is asserted – implementing a Wr_En.

For only 1 iteration, on the other hand:

```
if (c[3]) {
      R2 = ram[R7] pulse->p0
      }
```

If the device is ready (C3 asserted), the value stored at address R7 in the memory is loaded into register R2 and a Wr_En pulse is asserted.

Outputting/Writing Data to an Multiple External Devices

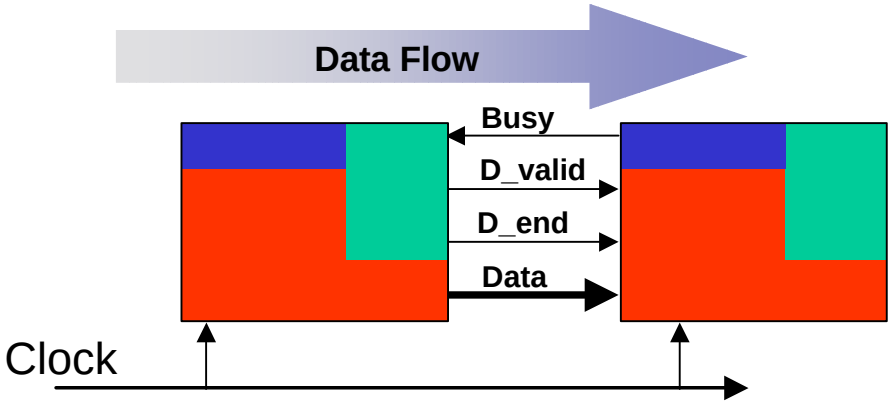


In case we're transmitting data to a few external devices in parallel, the connections shown in the scheme are quite similar to the previous example. There's one shared data bus, and one shared Wr_En signal. The CS (Chip Select) signals of devices A and B are implemented by the lsb's of Dev0 and Dev1, in accordance.
For a burst write, the appropriate code should be:

```
loop L_wr L_wr 10
device0=> 4'b0001
L_wr, R2=ram[R7++] pulse-> p0
```

The code here is also almost the same as in the previous example. The only difference is that the loop itself does not start right after the loop declaration. Before that we have an initialization line that asserts CS to device A.
Since the loop declaration always states the actual loop's start and end points, the loop itself (as shown in this example) does not have to come right after the loop declaration. The user can insert a few initialization commands before the loop starts.

Connecting two SCS16 Cores in Serial

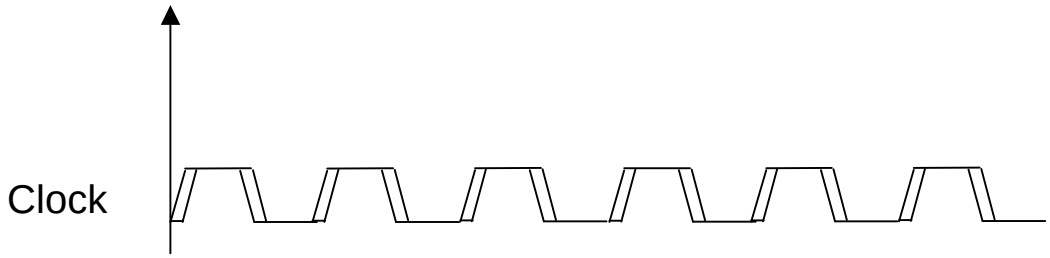


This structure could be useful for implementing an assembly line, where each **SCS16** accepts a packet, applies a specific function to it, and then sends it to the next link in the chain.
One can tell the interface between two **SCS16** cores is glue less. In this case the left **SCS16** is the master of the interface. It asserts D_Valid, to qualify the data it pushes, and D_End to declare there's currently no data available for sending. In this example there is no problem to plant wait states on the fly, by negating D_Valid, and then asserting it again. When the all the data has been transferred, D_End is asserted, and the right **SCS16** asserts Busy, to let the other **SCS16** know it is now working on the received data.

8.Timing Diagrams:

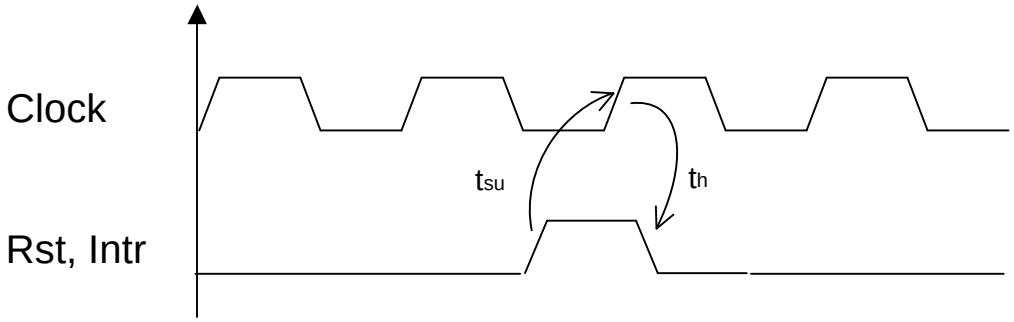
Note: All t_{su}, t_h and t_{pd} are Technology dependent. This section points out the stage in the clock cycle to which these timing demands refer.

Clk



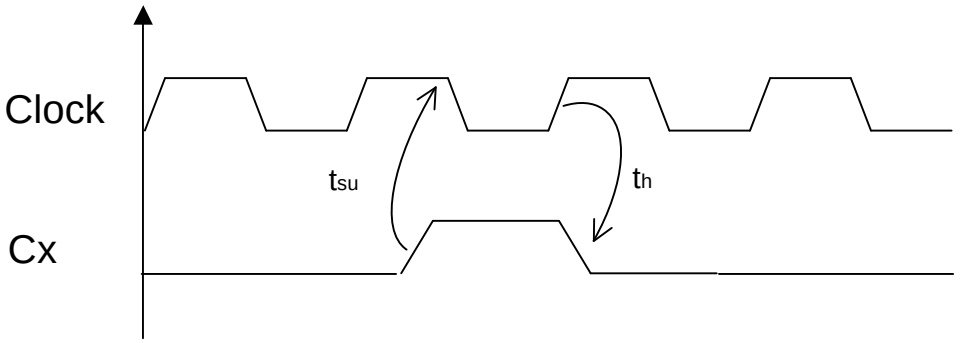
The clock's duty cycle must be between 48% and 52% of the whole clock cycle.

Rst, Intrrupt



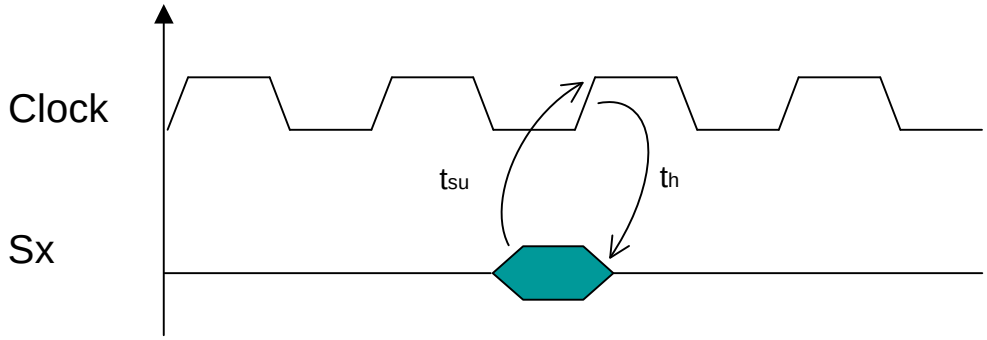
Both the Rst and Intr. Signals are sampled at the clock's positive edge.

Cx



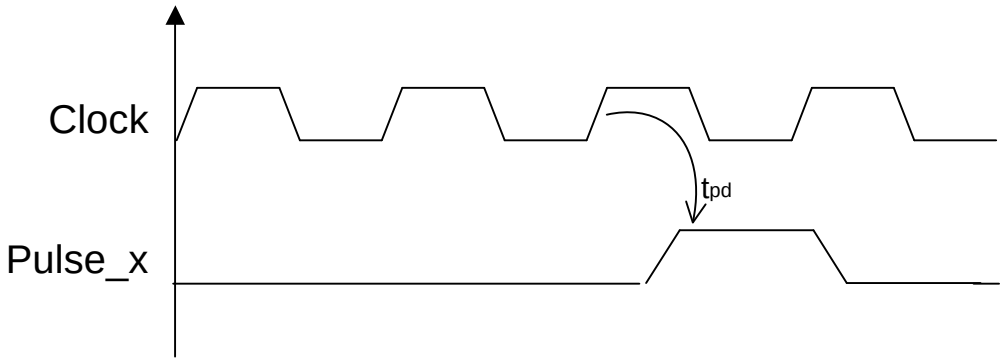
All Cx signals are sampled at the clock's negative edge and must stay stable until after the following positive edge.

Sx



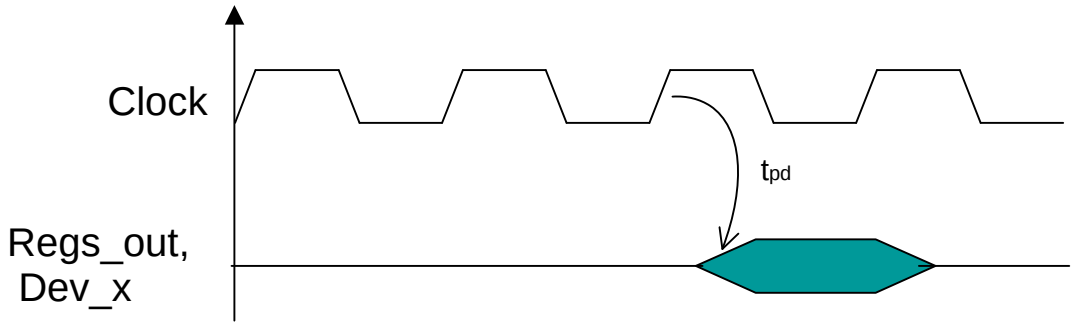
Both the Sx. busses are sampled at the clock's positive edge.

Pulse



The Pulse signals are valid after the clock's positive edge (one cycle after the execution of the code line from which the Pulse signals were activated).

Regs_out, Dev_x



Both the Registers and the Device bus are valid after the clock's positive edge.

For more information, please contact maxn@zorba-tec.com

Israel
Zorba Technologies Limited
Office Address: 14 Haraby Mibachrah street, Tel Aviv 66849
Mail Address: P.O.B 8126 Tel-Aviv 61081
TelFax: +972 3 6829315