DESIGN A MASTER DEVICE WITH AVALON MASTER INTERFACE

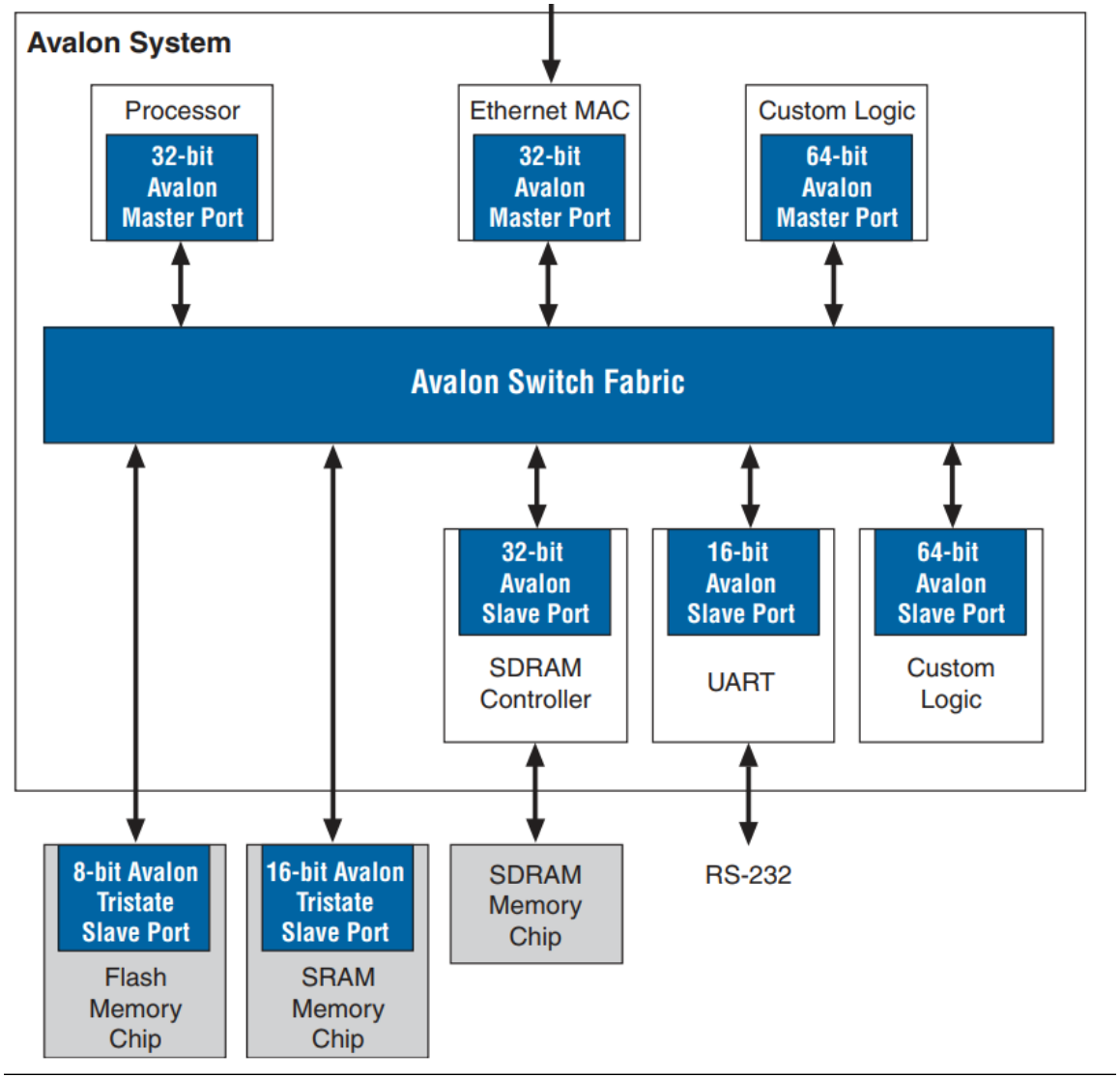
Kiet Tran

11/2023

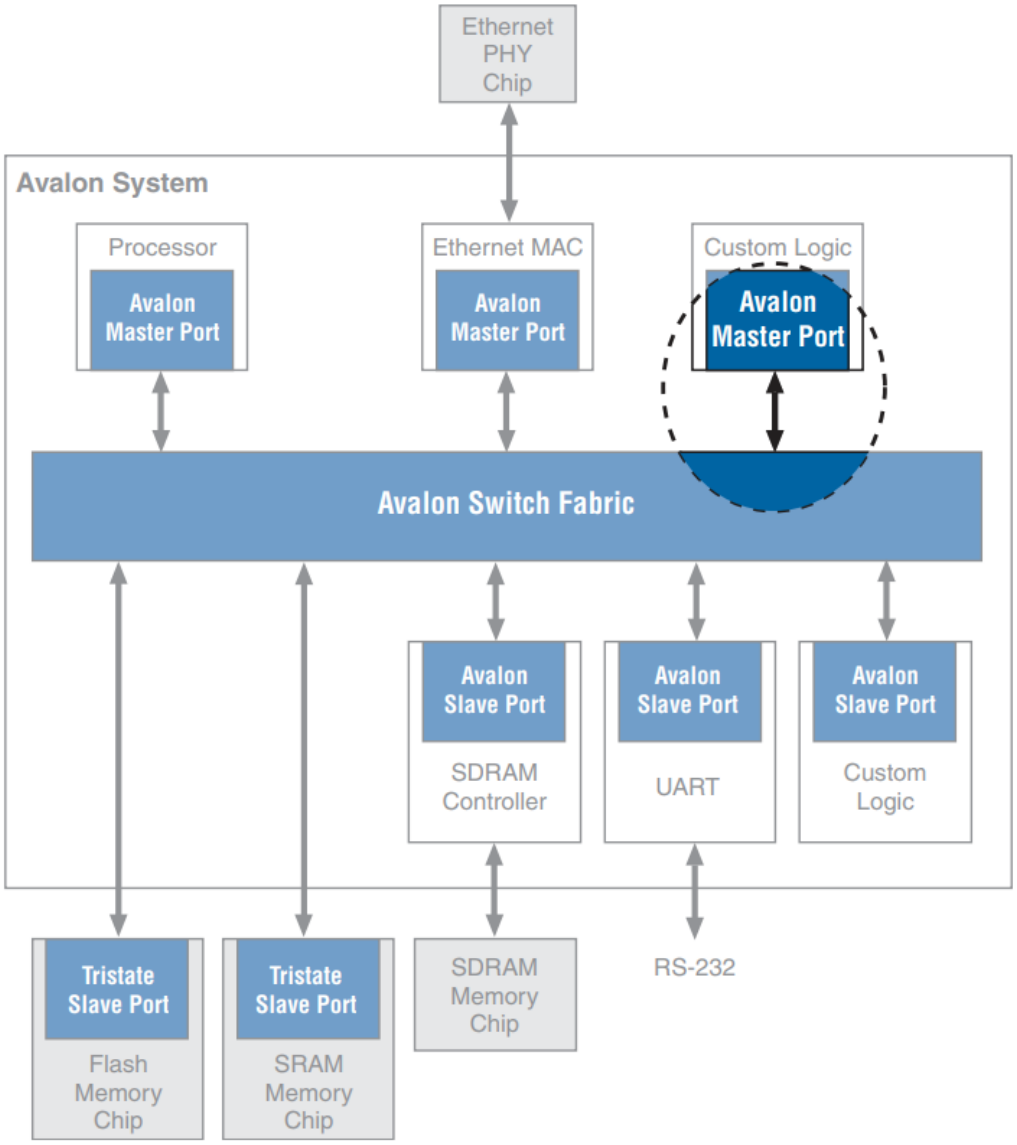
AGENDA

- Overview Avalone Master ports
- Basic transform from Master
- Master Design
- FSM
- Simulation
- Assignment

MASTER DEVICE



MASTER PORTS



MASTER PORTS

Table 2: Avalon Master Port Signals				
Signal Type	**Width**	**Direction**	**Required**	**Description**
Fundamental Signals				
`clk`	1	In	Yes	Synchronization clock for the Avalon slave interface. All signals are synchronous to `clk`.
`waitrequest`	1	In	Yes	Forces the master port to wait until the Avalon switch fabric is ready to proceed with the transfer.
`address`	1-32	Out	Yes	Address lines from the master port to the Avalon switch fabric. The `address` signal represents a byte address. However, the master port must assert `address` on word boundaries only.
`read`	1	Out	No	Read request signal from master port. Not required if master port

MASTER PORTS

Table 2: Avalon Master Port Signals

Signal Type	Width	Direction	Required	Description
				never performs read transfers. If used, `readdata` or `data` must also be used.
`readdata`	8,16,32,64, 128 (1)	In	No	Data lines from the Avalon switch fabric for read transfers. Not required if the master port never performs read transfers. If used, `read` must also be used, and `data` cannot be used.
`write`	1	Out	No	Write request signal from master port. Not required if the master port never performs write transfers. If used, `writedata` or `data` must also be used.
`writedata`	8,16,32,64, 128 (1)	Out	No	Data lines to the Avalon switch fabric for write transfers. Not required if the master port never performs write transfers. If used, `write` must also be used, and `data` cannot be used.
`byteenable`	0,2,4,8, 16	Out	No	Byte-enable signals to enable specific byte lane(s) during write transfers to memories of width greater than 8 bits. The master port must assert all `byteenable` lines during read transfers.
Pipeline Signals				
`readdatavalid`	1	In	No	Used for pipelined read transfers with latency.

MASTER PORTS

Table 2: Avalon Master Port Signals

Signal Type	Width	Direction	Required	Description
				Indicates that valid data from the Avalon switch fabric is present on the `readdata` lines. Required if the master is pipelined.
`flush`	1	Out	No	Used for pipelined read transfers. The master port asserts `flush` to clear any pending transfers in the pipeline.
Burst Signals				
`burstcount`	2-32	Out	No	Used for burst transfers. Indicates the number of transfers in a burst.
Flow Control Signals				
`endofpacket`	1	In	No	Used for transfers with flow control. Indicates an end-of-packet condition from the Avalon switch fabric. Implementation is peripheral specific.
Tristate Signals				
`data`	8,16,32,64, 128			Bidirectional read and write data for tristate master ports. If used, `readdata` and `writedata` cannot be used.
Other Signals				
`irq`	1, 32	In	No	Indicates when one or more slave ports have requested an interrupt. If `irq` is a 32-bit vector, each line corresponds directly to the `irq` signal on a slave port, with no inherent assumption of priority. If `irq` is one bit wide, it is the logical OR of all slave `irq` signals, and the interrupt priority is encoded on `irqnumber`.

BASIC TRANSFORMS

4.1.1. waitrequest

The `waitrequest` signal is a master port input that indicates that the Avalon switch fabric is not ready to proceed with a transfer. There is one golden rule that applies to all master transfers: Obey the `waitrequest` signal.

At the start of all transfers, a master port asserts the appropriate signals to initiate the transfer, and then waits until the Avalon switch fabric deasserts `waitrequest`.

The Avalon switch fabric deasserts `waitrequest` when not performing a transfer with a master port.

BASIC TRANSFORMS

4.1.2. address

Master addresses represent byte addresses, regardless of the data-width of the master port. A master port can assert only addresses aligned to word boundaries, based on the master port's data width. For example, a 32-bit master port can assert only addresses aligned to 4-byte boundaries, such as 0x00, 0x04, 0x08, 0x0C, etc. In this case, the Avalon switch fabric ignores the lower two bits of `address`. To write to a specific byte within a data word, the master port must use the `byteenable` signal.

BASIC TRANSFORMS

4.1.3. readdata & writedata

The `readdata` and `writedata` signals carry the data associated with a transfer. A master port can use one, none, or both of these signals. These signals must be 8, 16, 32, 64, or 128 bits wide. If a master port uses both `readdata` and `writedata`, the widths must be equal for both signals.

4.1.4. read & write

The `read` and `write` signals are 1-bit outputs from the master port to indicate when it is about to start a new read or write transfer.

The timing diagrams of transfers below demonstrate each transfer as an isolated event, but under realistic circumstances transfers can occur in succession. For example, after the master port terminates a read transfer, it can continue asserting `read` to assert another read transfer on the next cycle.

BASIC TRANSFORMS

4.1.5. byteenable

The `byteenable` signal is a vector signal with one line for every byte lane in `writedata`. During write transfers, a master port greater than 8 bits wide can assert the `byteenable` signal to specify which byte lane(s) to write.

A master port must assert all `byteenable` lines during read transfers.

Table 4 shows some example cases of `byteenable` during write transfers for a 32-bit master port.

Table 4: Byte-Enable Example for a 32-Bit Slave Port

Byteenable [3..0]	Write Action
1111	Write full 32-bits
0011	Writes lower 2 bytes
1100	Writes upper 2 bytes
0001	Write byte 0 only
0100	Write byte 2 only

BASIC TRANSFORMS

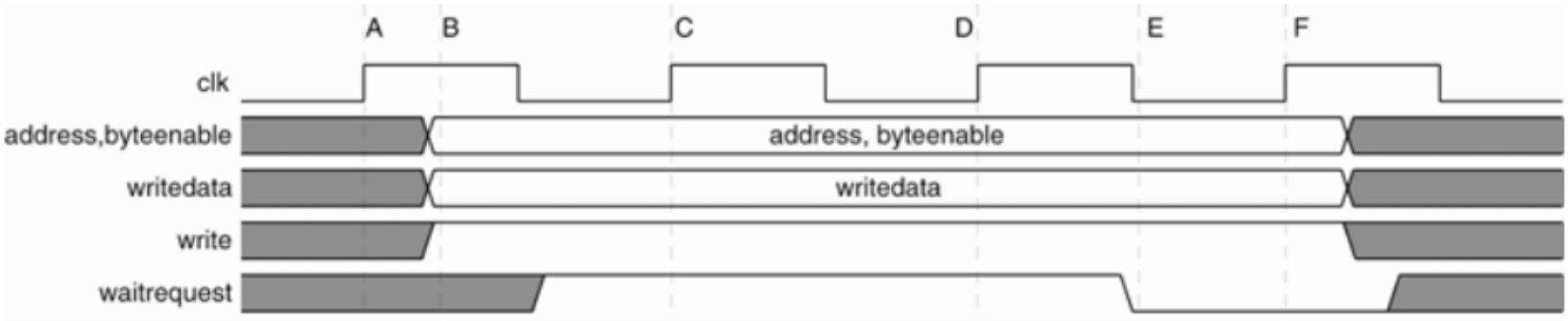
Figure 15: Master Read Transfer with Wait-States

Notes to Figure 15:

(A) First cycle starts on the rising edge of `clk`.
(B) Master asserts valid `address`, `byteenable` and `read`.
(C) Avalon switch fabric asserts `waitrequest` before the next rising edge of `clk`.
(D) Master port accepts `waitrequest` at the rising edge of `clk`. This cycle becomes a wait-state.
(E-F) As long as `waitrequest` is asserted, master port holds all outputs constant.
(G) Valid `readdata` returns from Avalon switch fabric.
(H) Avalon switch fabric deasserts `waitrequest`.
(I) Master port captures `readdata` on the next rising edge of `clk` and deasserts all outputs. The read transfer ends here, and the next cycle could be the start of another transfer.

BASIC TRANSFORMS

Figure 17: Master Write Transfer with Wait-States



Notes to Figure 17:

(A) First cycle starts on the rising edge of `clk`.

(B) Master asserts valid `address`, `writedata` and `write`.

(C) `waitrequest` is asserted at the rising edge of `clk`, so this cycle becomes the first wait-state. Master holds all outputs constant.

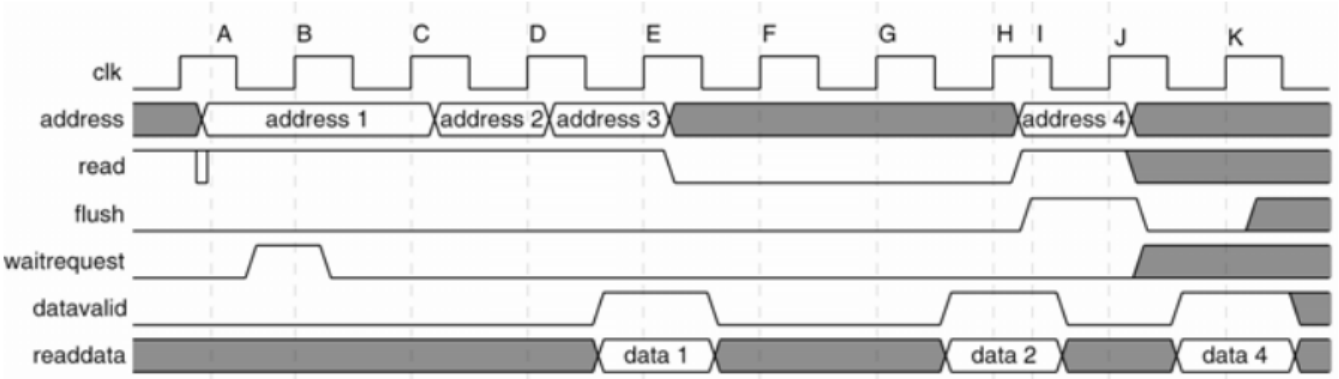
(D) `waitrequest` is asserted at the rising edge of `clk` again, so this becomes the second wait-state. Master holds all outputs constant.

(E) Avalon switch fabric deasserts `waitrequest`.

(F) `waitrequest` is not asserted at the rising edge of `clk`, so master deasserts all outputs, and the write transfer terminates. Another read or write transfer may follow on the next cycle.

BASIC TRANSFORMS

Figure 20: Master Pipelined Read Transfer



Notes to Figure 20:

(A) Master initiates a read transfer by presenting `address` and `read` for the address phase of the new transfer.

(B) Avalon switch fabric is asserting `waitrequest`, so the master port waits and asserts `address` and `read` for another cycle.

(C) The Avalon switch fabric deasserts `waitrequest`, and captures `address` at the next rising edge of `clk`. `readdatavalid` is not asserted, so master does not capture `readdata`.

(D) The Avalon switch fabric captures a new address at the rising edge of `clk`. `readdatavalid` is not asserted, so master does not capture `readdata`.

(E) The Avalon switch fabric captures a new address at the rising edge of `clk` (making a total of three pending transfers). `readdatavalid` is asserted, so the master captures valid `readdata` (data 1).

(F) `readdatavalid` is not asserted, so master does not capture `readdata`.

(G) `readdatavalid` is not asserted, so master does not capture `readdata`.

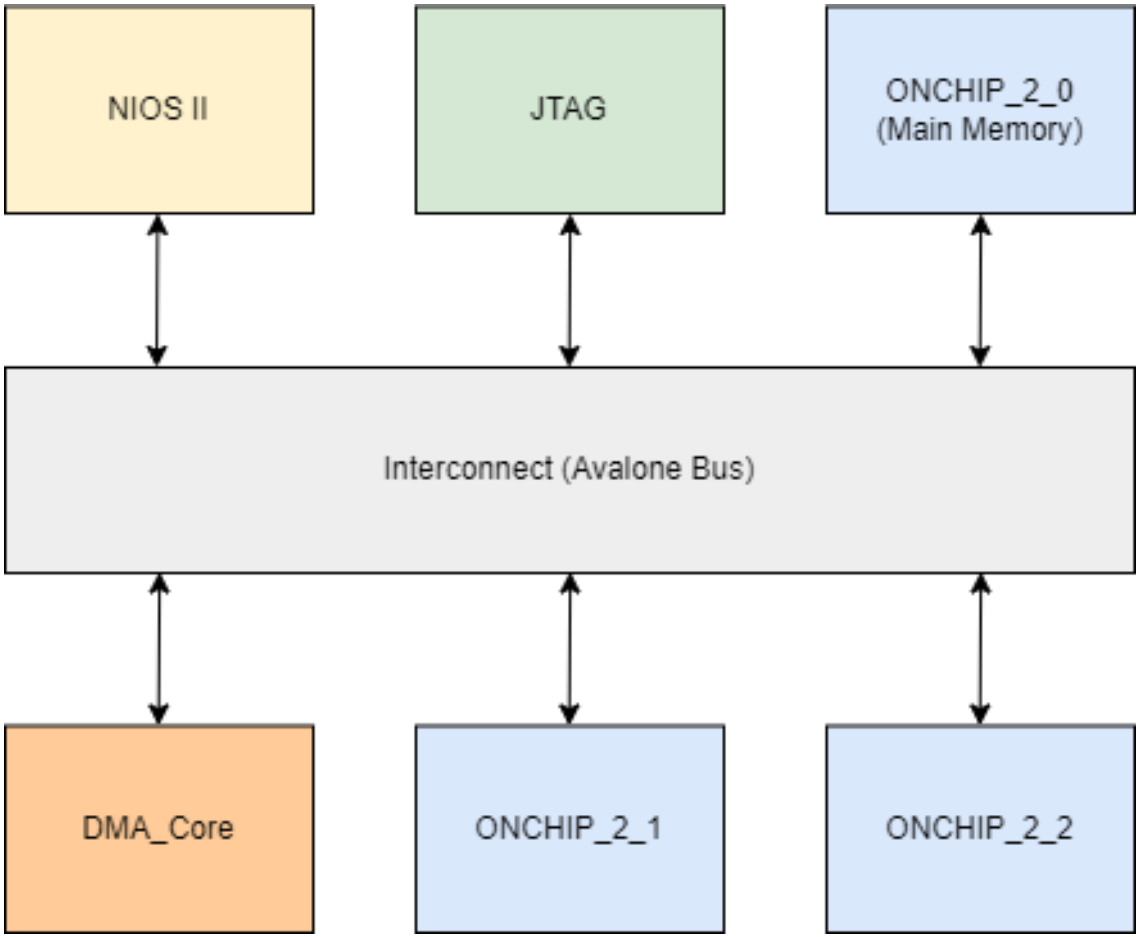
(H) `readdatavalid` is asserted, so master captures valid `readdata` (data 2).

(I) Master presents `address` and `read` for a new read transfer.

(J) `readdatavalid` is not asserted, so master does not capture `readdata`. Master asserts `flush`, causing the Avalon switch fabric to flushes the pending transfer (address 3). Avalon switch fabric captures the new address.

(K) `readdatavalid` is asserted, so master captures valid `readdata` (data 4). At this point, no transfers are pending.

SYSTEM


NIOS II
JTAG
ONCHIP_2_0
(Main Memory)
Interconnect (Avalone Bus)
DMA_Core
ONCHIP_2_1
ONCHIP_2_2

DMA – DIRECT MEMORY ACCESS

1.What is DMA?

1. DMA stands for Direct Memory Access.
2. It's a feature that enhances the data transfer capabilities of a computer system.

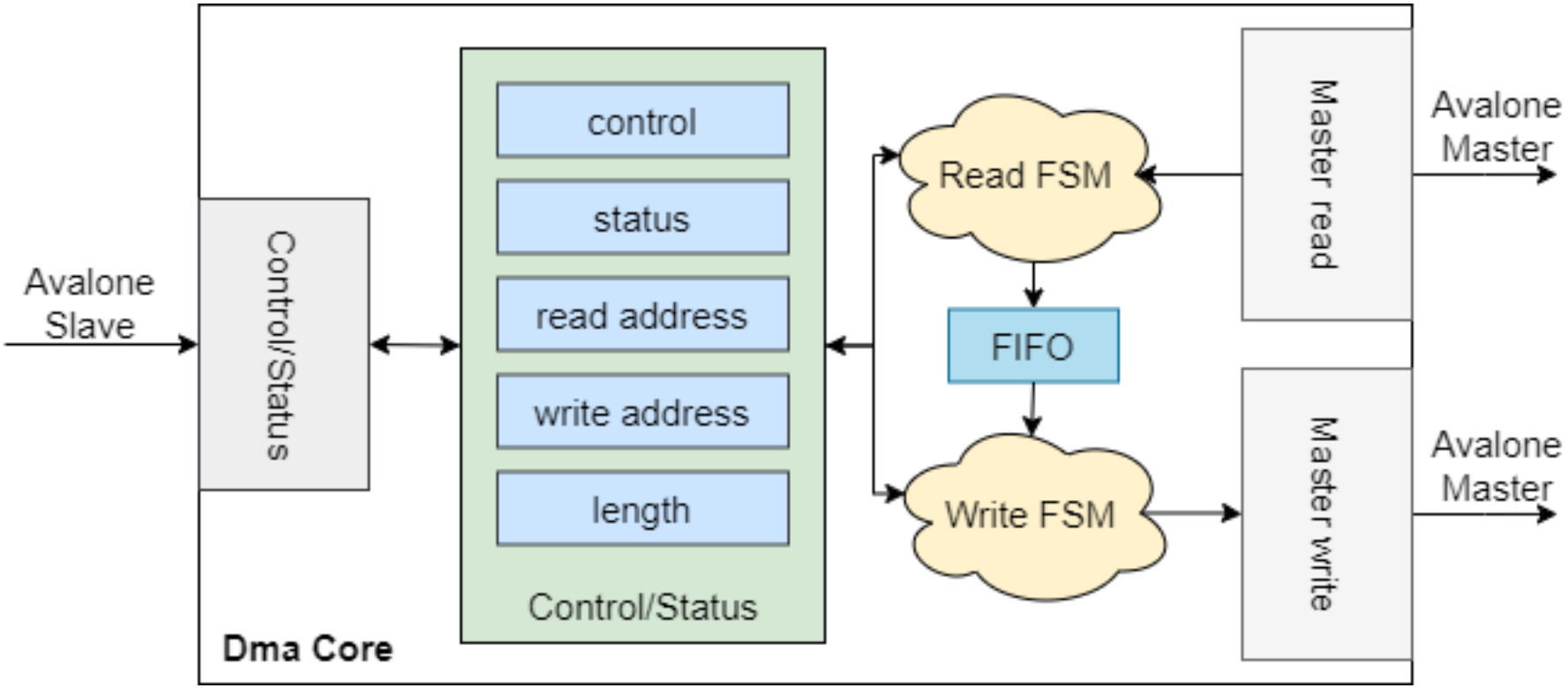
2.Why Use DMA?

1. DMA allows data to be transferred between memory and peripheral devices without CPU involvement.
2. This offloads the CPU, freeing it for other tasks.

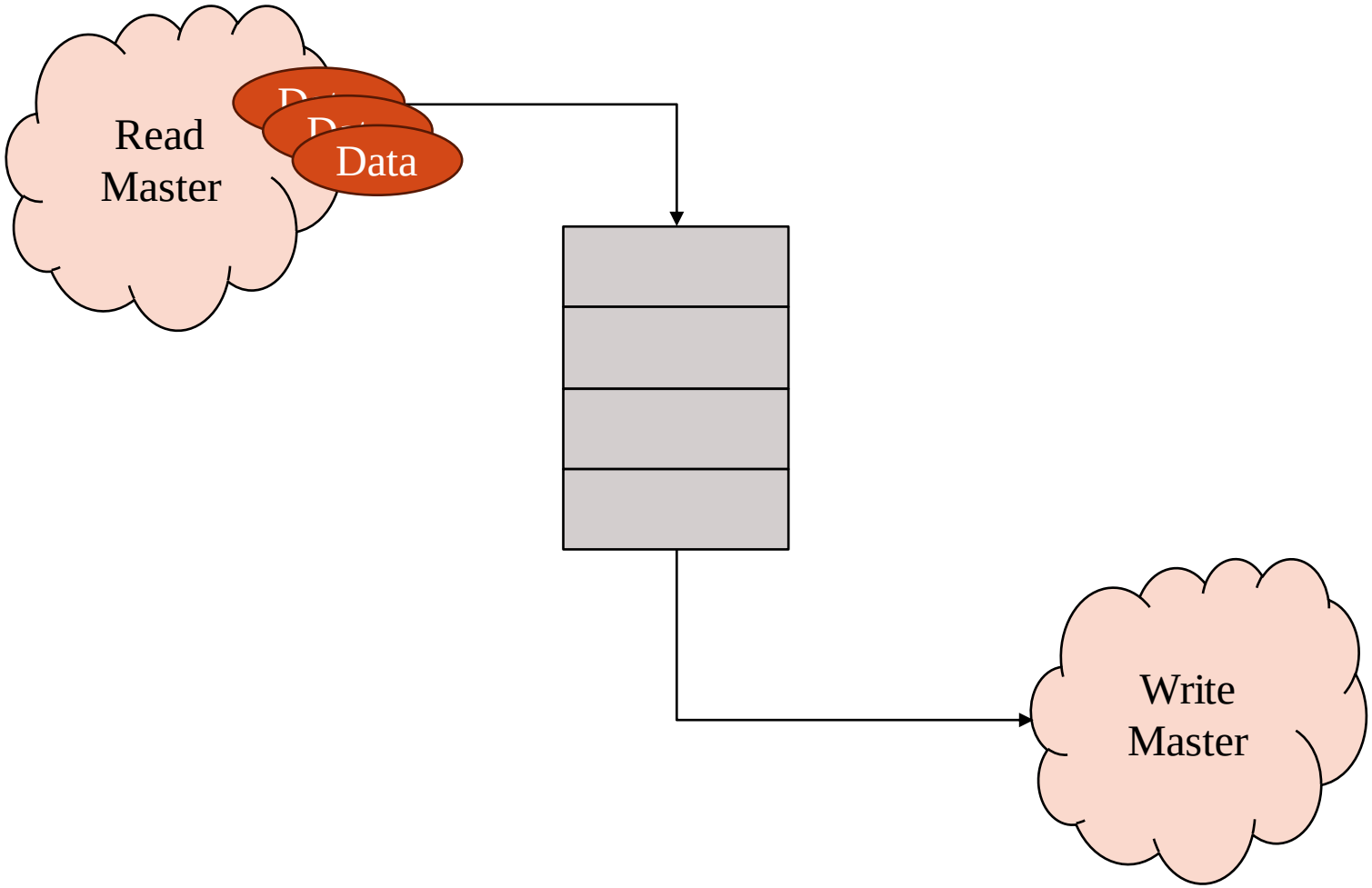
3.Key Features of DMA

1. **Faster Data Transfer**: DMA can transfer data more quickly and efficiently than the CPU.
2. **Reduced CPU Overhead:** DMA reduces the burden on the CPU, improving overall system performance.

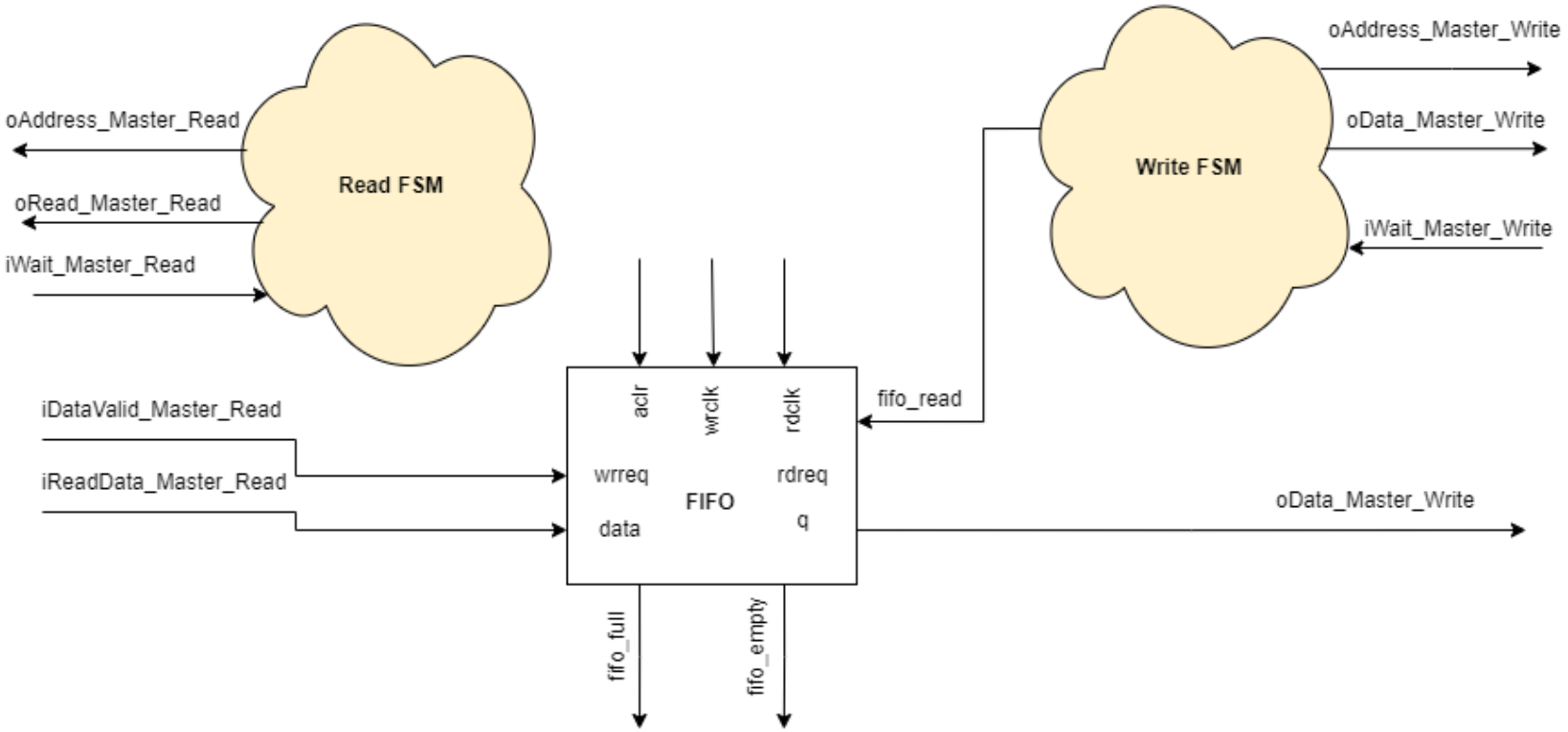
DMA CORE


Avalone
Slave
Control/Status
control
status
read address
write address
length
Control/Status
Dma Core
Read FSM
FIFO
Write FSM
Master read
Master write
Avalone
Master
Avalone
Master

CONCEPT



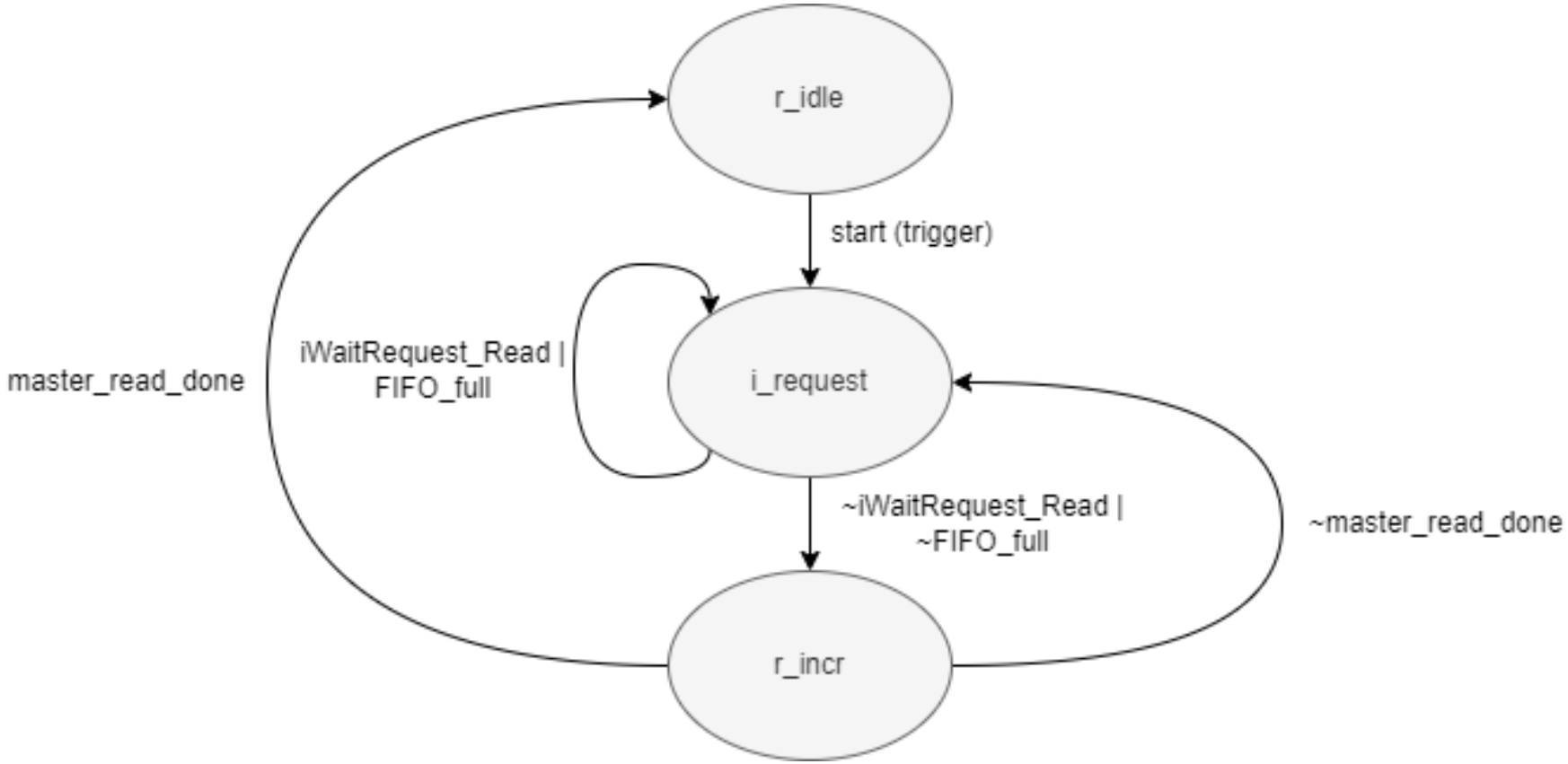
FIFO CONNECTION



CONTROL/STATUS

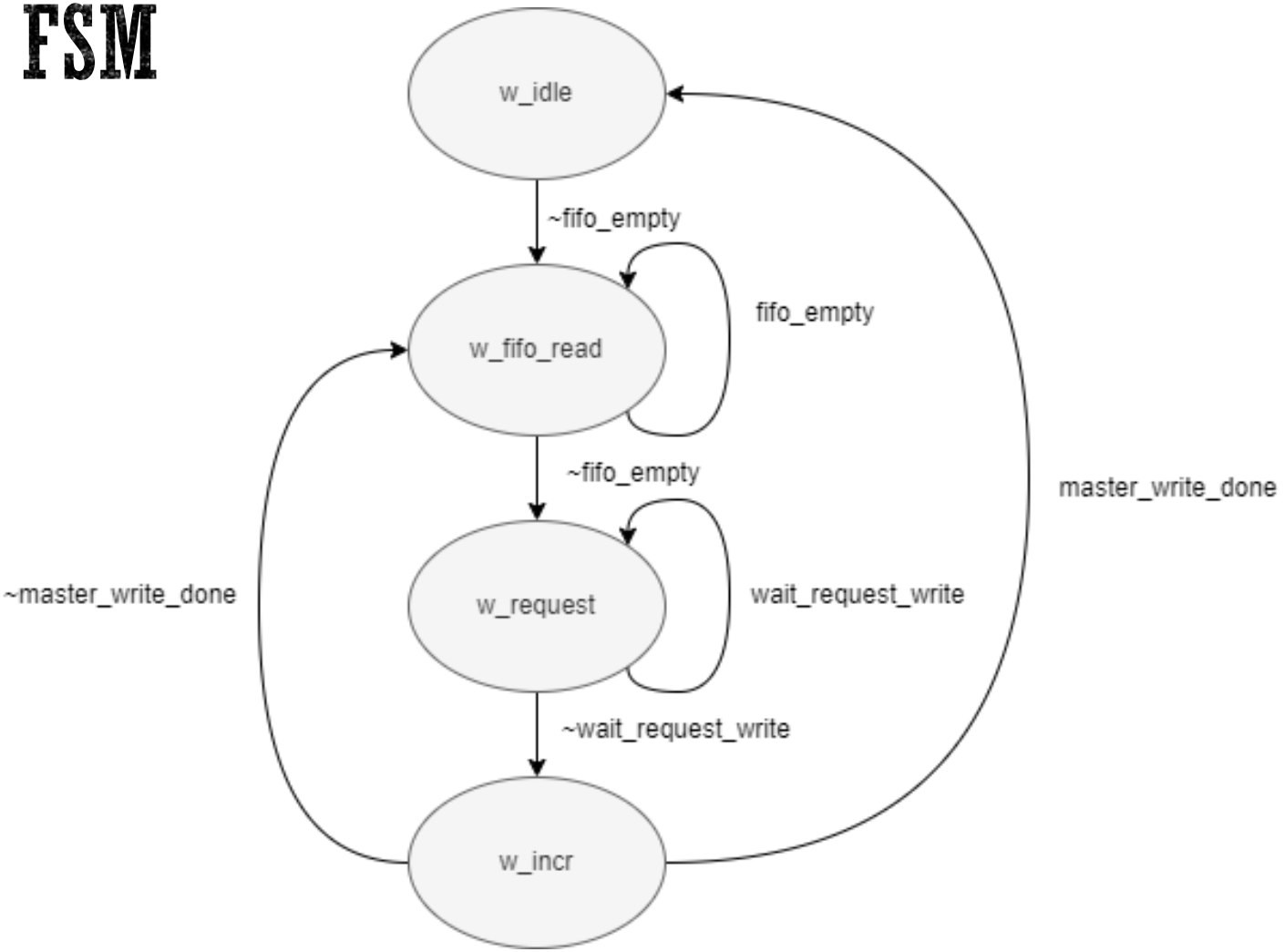
Offset	Name	Access	Description
0	Control	R/W	Control register Bit 0 - Start . Start DMA operation. This bit is auto clear when DMA done. Bit 1 - FIFO_Clear. Clear FIFO data
1	Read address	R/W	Start address for reading
2	Write address	R/W	Start address for write
3	Length	R/W	Length of data (byte)
4	Status	R	Status register Bit 0 : Start. Start = 1, DMA is running, else DMA is idle Bit 1: FIFO clear. This bit is mirror of FIFO_Clear bit in control register. Bit 2: FIFO empty. Bit 3: FIFO full. Bit 4: DMA done. This bis is auto cleared when the Start bit in Control register is set to 1.

READ FSM



- The start_trigger is a rising edge pulse of start bit.
- In state "r_incr", address is increase by 4.

WRITE FSM

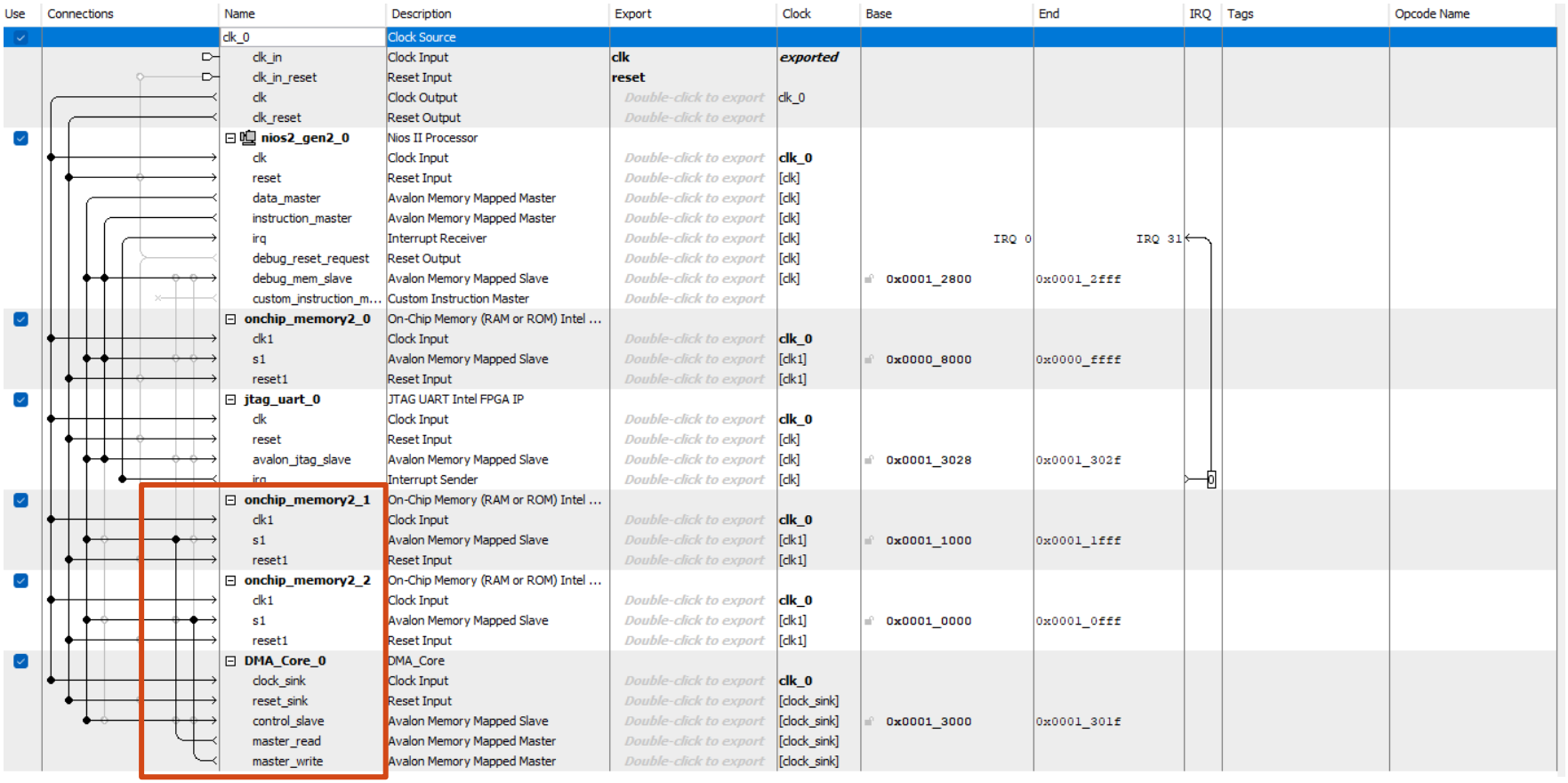


- In state "w_incr", address is increase by 4.

SOFTWARE SEQUENCE

- Set start read address
- Set start write address
- Set length
- Set enable start bit in control register
- SW polling the "done" bit in status register to check DMA is done of not
  - If DMA done -> break

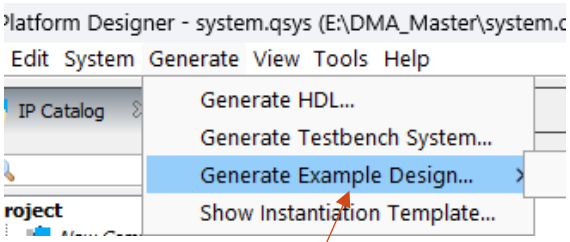
SIMULATION



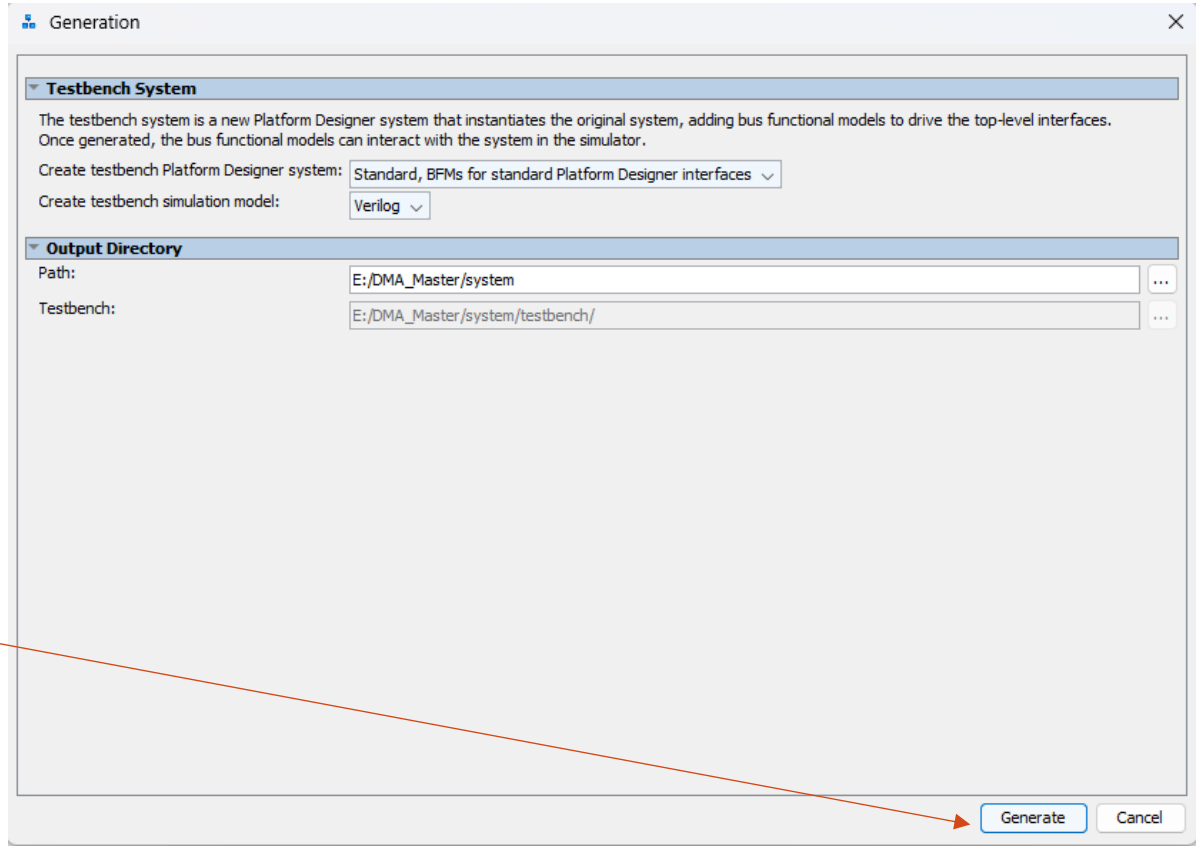
Master read -> read from onchip_memory_2_1

Master write -> write to onchip_memory_2_2

SIMULATION



Generate testbench



SIMULATION

```c
#include "sys/alt_stdio.h"
#include "system.h"

void init_mem1(){
    volatile int* mem_ptr = (int *) ONCHIP_MEMORY2_1_BASE;
    int i = 0;
    for (i = 0; i < 8; i++){
        *(mem_ptr + i) = i;
    }
}

int main()
{
  alt_putstr("Hello from Nios II!\n");
  init_mem1();
  volatile int* dma_ptr = (int*)(DMA_CORE_0_BASE);
  volatile int* mem2_ptr = (int*)(ONCHIP_MEMORY2_2_BASE);
  unsigned int status = 0;
  unsigned int i = 0;
 /*init DMA*/
  *(dma_ptr + 1) = ONCHIP_MEMORY2_1_BASE;
  *(dma_ptr + 2) = ONCHIP_MEMORY2_2_BASE;
  *(dma_ptr + 3) = 32;
  *(dma_ptr + 0) = 1;

  /* Event loop never exits. */
  while (1){
      status = *(dma_ptr + 4);
      if ((status & 0x10) == 0x10){
          alt_putstr("Dma Done\n");
          for (i = 0; i < 32/8; i++){
              printf ("mem2[%d] = %d\n", i, *(mem2_ptr + i));
          }
          break;
      }
  }

  return 0;
}
```

Init data in onchip_2_1 with value from 0->7

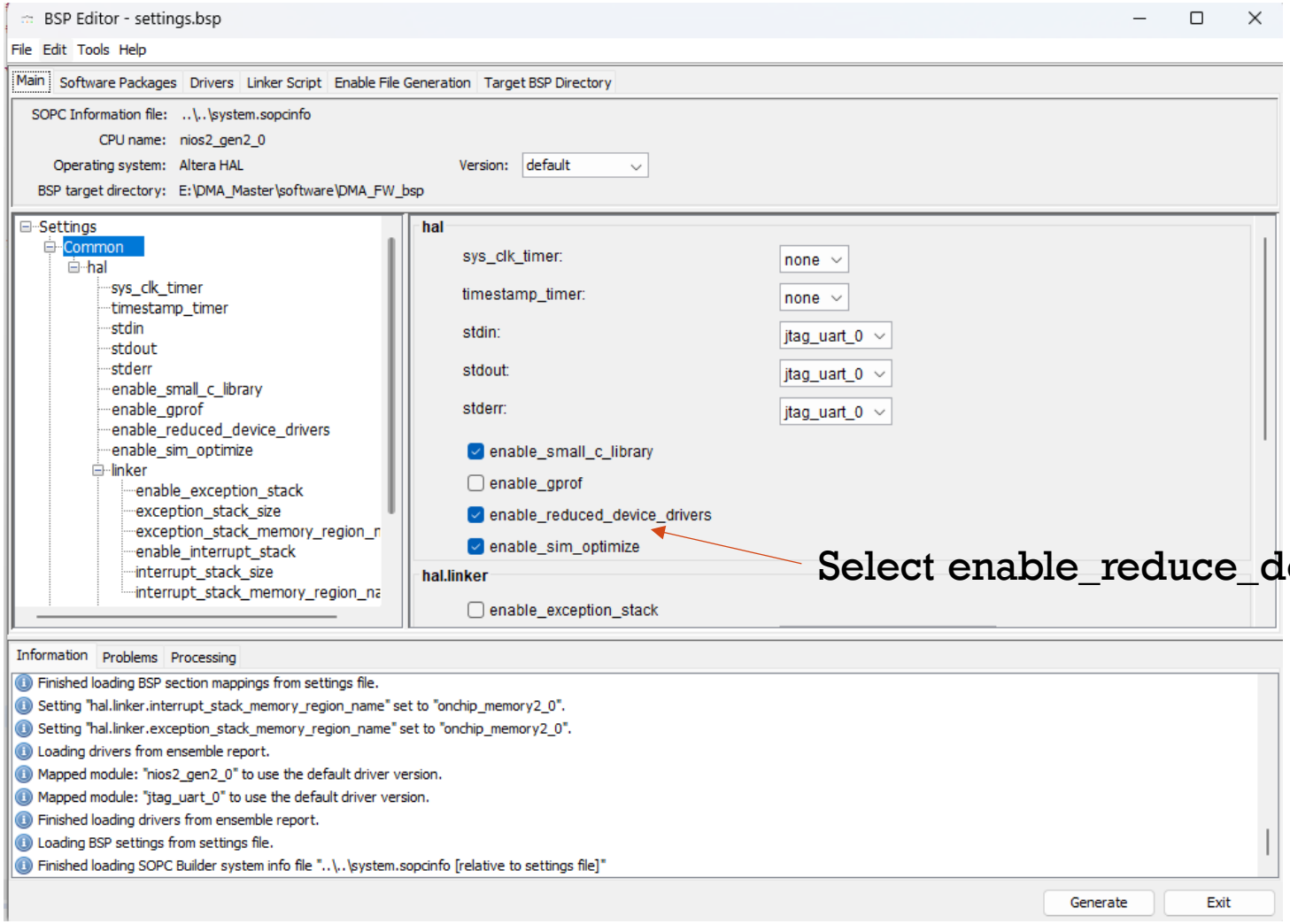
Write:
Start address
Write address
Length
Enable start bit

Polling DMA_DONE bit

Print data from onchip_2_2

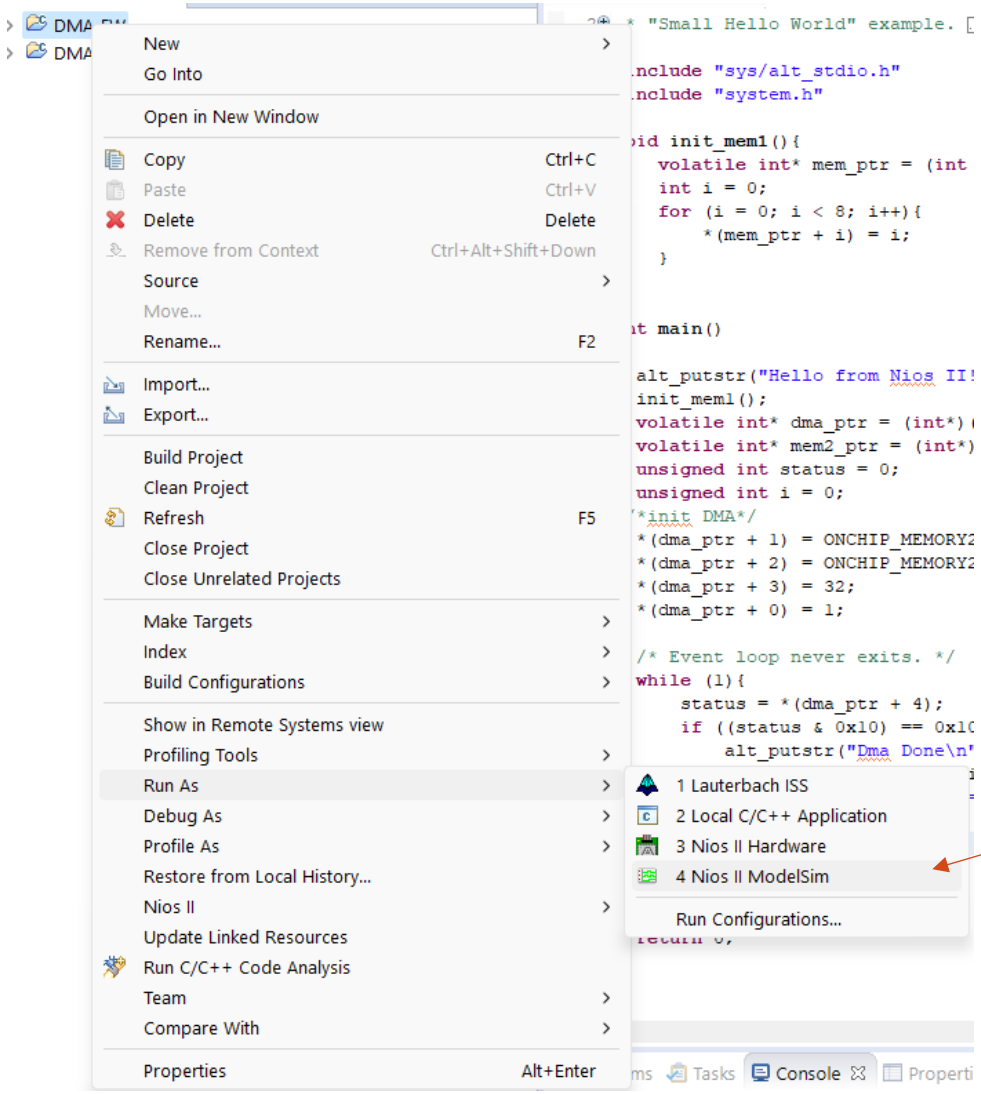
Create SW project on Eclipse with code like this.

SIMULATION



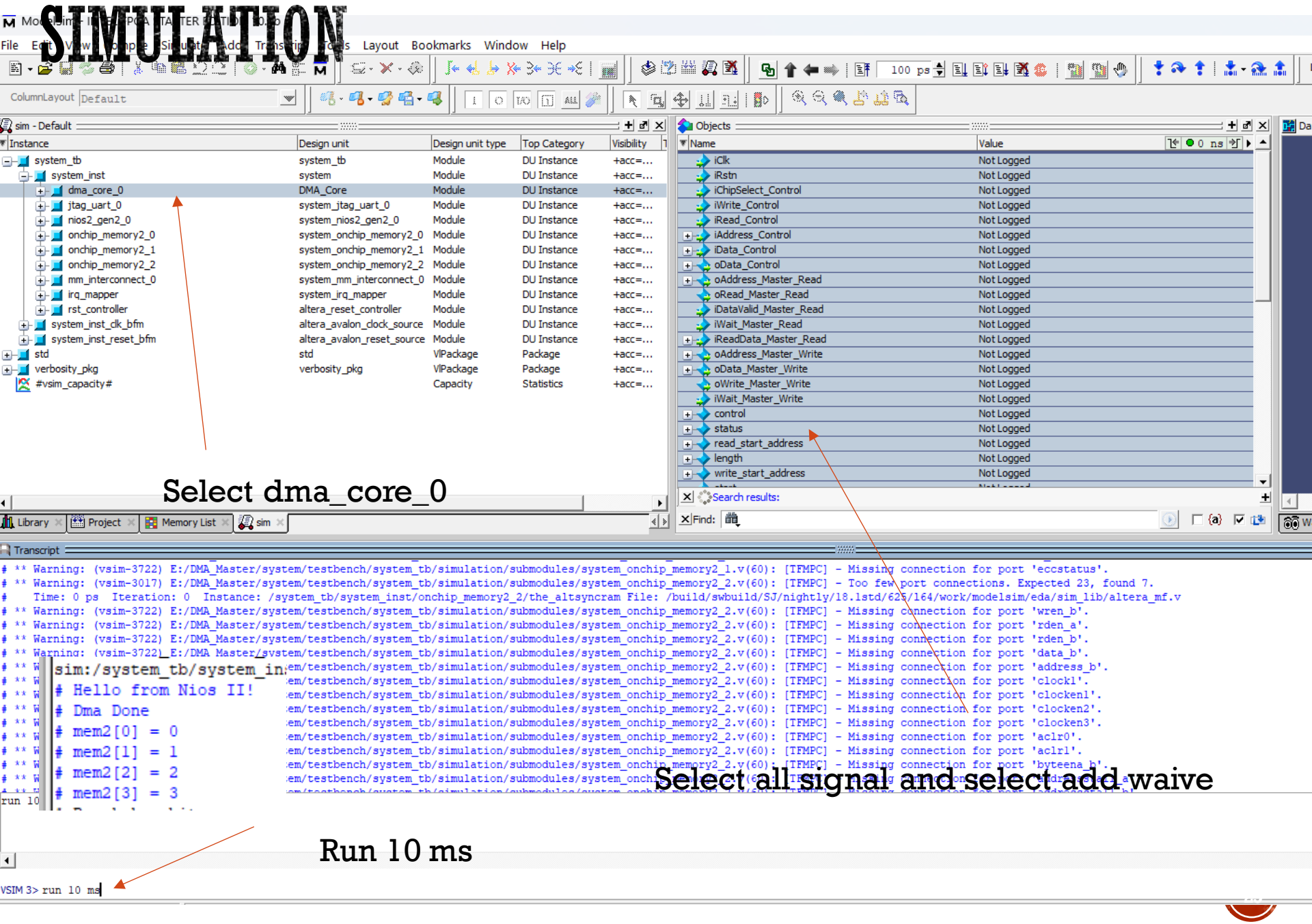
Select enable_reduce_device_drivers

SIMULATION



Run simulation

SIMULATION

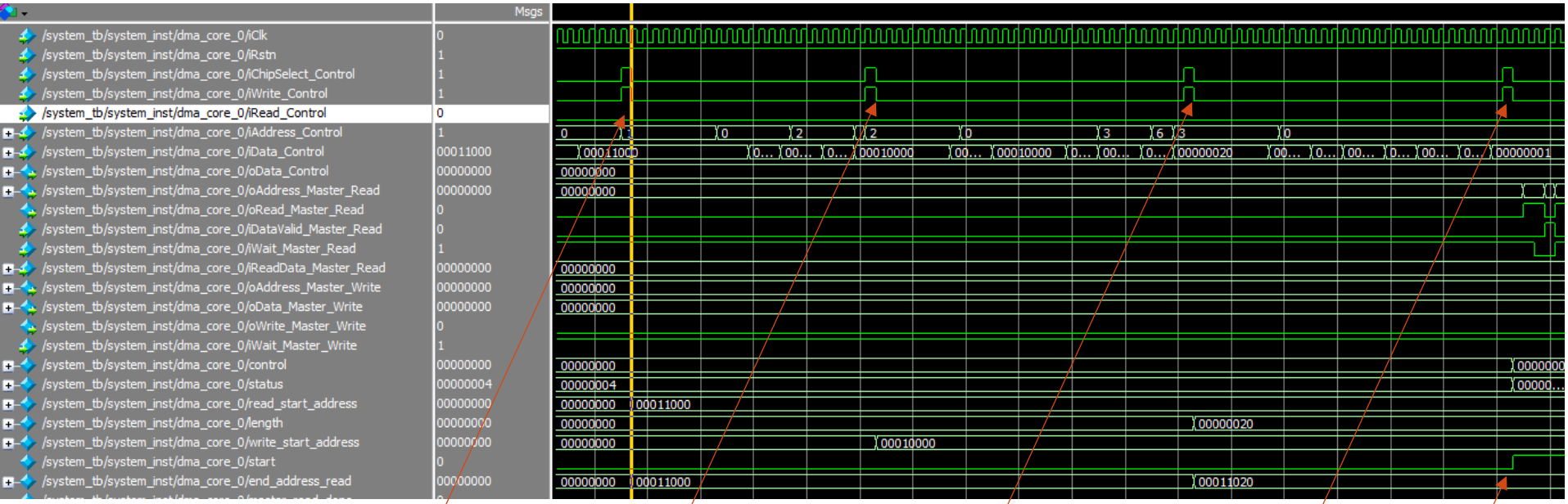


Select dma_core_0

Select all signal and select add waive

Run 10 ms

SIMULATION



Set read address

Set write address

Set length

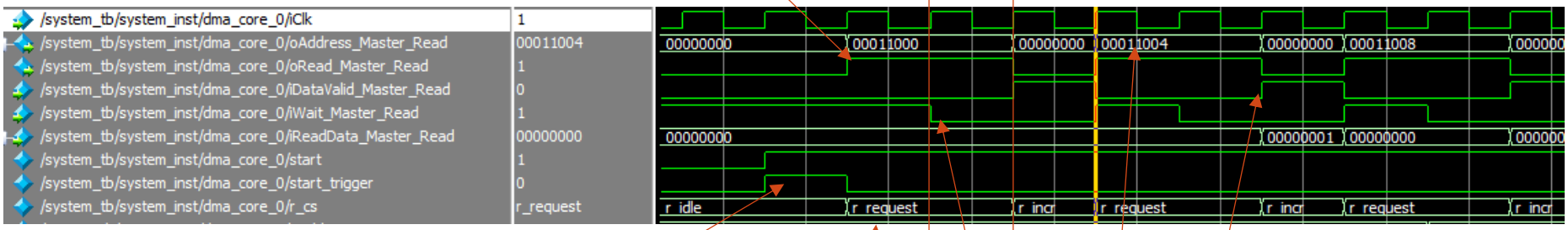
Enable start bit

```
sim:/system_tb/system_in
# Hello from Nios II!
# Dma Done
# mem2[0] = 0
# mem2[1] = 1
# mem2[2] = 2
# mem2[3] = 3
```

Start = 1 after enabled by SW

SIMULATION

This start, FSM assert address + read signal



Start trigger is a rising edge of start

Read start change from r_idle to r_request.

Wait request is zero, start change to r_incr and increase address by 4

Read data valid indicate data arrived is true

Wait request = 1 → hold start request more a cycle

```
sim:/system_tb/system_in
# Hello from Nios II!
# Dma Done
# mem2[0] = 0
# mem2[1] = 1
# mem2[2] = 2
# mem2[3] = 3
```

Result from simulation console

Master read

RESOURCE

Link project: DMA_Master.7z

ASSIGNMENT

- Design a module named as "Sort_Array":
  - Has DMA read to read 32 input element from Memory.
  - Sort 32 element from smallest to largest.
  - Store 32 elements into a FIFO and read-out by software by order.

Submit:

- RTL code + Testbench
- Report: diagram + waveform + explanation

SUBMIT PREVIOUS ASSIGNMENT

- Link: https://forms.office.com/r/QSM71nbwNb
- End time: 08/11/2023 – 9h00 PM