

Memory I/O Engine subdesign Design, VHDL, & Simulation Notes

This document provides a top down conceptual view, with details, of the VHDL-based Memory Input Output Interface Engine, a subdesign for the SFC04 project. As with the some of the other sub-designs (Prescaler, Push4Buttons, USBioEngine, and LED&DUTdrv) it has been developed as a stand alone module for gaining the ability to perform more detailed behavioral simulations to verify functionality before incorporation into the main design module.

Due to the complexity of this particular engine, a partial design simulation was executed on just the address controller portion, when it was completed enough for testing and VHDL rework. Therefore, the description of the MEMioEngine will include the address controller as a separate entity.

The following is a short list of the main topics covered by this document:

Page #	Topic:
2	General Functional Description: How it Works
4	Physical Memory Address Generator Scheme (diagram)
5	Memory Mapping Scheme, Byte Lane Control & Banks (diagram)
6	MEMioEng Interface Signals
8	MEMioEng Interface Operational Specifications (how to use this subdesign)
11	MEMioEng Address Controller Test Bench Waveform diagram (simulation driver)
12	MEMioEng Address Controller Simulation Results
15	MEMioEng whole design Test Bench Waveform diagram (simulation driver)
16 & 19	Simulation of dual tristate buses difficulty Note
16	MEMioEng whole design Simulation Results
22→30	MEMioEng VHDL listing

During the discussion of the functions of various I/O lines, references will be made to pieces of the MEMioEngine as presented in the embedded (at the end of this document) VHDL code. There is an octal-state sequence engine that is used to handle memory read and write operations. The waiting state is state = 0. Moving from this state is the only conditional element in the engine. What this means is that once a read or write starts, the sequential state machine will cycle through its appropriate states unconditionally.

The engine states for a Write to Memory operation are:
0.....waiting...0→1→2→3→0.....waiting...

The engine states for a Read from Memory operation are:
0.....waiting...0→4→5→6→0.....waiting...

There is also a special “read” feature built into the engine’s sequencing that permits “continuous” back-to-back reads to be executed, with automatic address incrementing occurring.

The address controller is designed with “hooks” for the main engine to preload any address needed for reads or writes.

Memory I/O Engine subdesign Design, VHDL, & Simulation Notes

General Functional Description How It Works

Because “slow” operations are actually desirable for this particular demonstration board’s needs, the engine design performs single byte reads and writes (in no particular hurry), from and to the memory, for the main engine’s information transfer needs. To change the access technique to a 32-bit access method, the byte lane control signals need to be decoupled from the address counter, and all actively driven together, instead of in an encoded one-hot method, as it is now.

The main engine interface provides it with five different requests for service, all being delivered with a simple high pulse (100nS duration) synchronous to the system clock. The read request for service, via RdMemRq, may be extended longer for implementing continuous read operations. An additional handshake line, therefore, has been added for orderly read sequencing purposes. The requests for service are:

Service Request Signal	Service response executed by MEMioEng
RdMemRq	Using current Bank, Address, & Byte: Read one or more Byte(s)
NewMemWr	Using MemOutHold & current Bank, Address, & Byte: Write Byte
newLoAddWr	Using MemOutHold preload Address Counter’s Low Byte Address
newHiAddWr	Using MemOutHold preload Address Counter’s High Byte Address
RstAddressCntr	Synchronously Reset the preloadable Address Counter

It is very important that the main engine ensures that only one service request is issued at a time, and that no new ones are presented until the corresponding busy signals indicate that the MEMioEngine is waiting to start a new service operation. There is no additional VHDL code added in the MEMioEngine to “lock-out” simultaneous differing requests for service.

For each of the Service Request Signals tabulated above, there is an associated “busy” signal. The assertion “high” indicates that the service request has been acknowledged and that the MEMioEngine is now performing the requested service. When that same busy signal negates “low,” this is an indication to the main engine that the operation has been completed. The exception to the “busy” signal state definition is what occurs during a single or continuous read operations from memory. For reads, the “busy” signal is asserted when the data has already been fetched from memory and it has been registered and presented to the main engine for retrieval purposes on the MemInPipe port. The “additional handshake line” (mentioned above) for read operations is a simple signal from the main engine that it “got the read data,” (GotMemRd) which subsequently results in the negation of the read operation’s “busy” signal. If the RdMemRq signal is changed from a simple 100nS high pulse for a single memory read, to an extended assertion high, then the MEMioEng will automatically read the next address location, until the main engine “tells” it to stop reading by the negation of that service request signal.

Memory I/O Engine subdesign Design, VHDL, & Simulation Notes

The service request “busy” signals are:

“BUSY” Signal	Causal Request	What’s Happening when asserted:
MemRdBsy	RdMemRq	Read data has been registered into MemInPipe
MemWrBsy	NewMemWr	Writing process is running
LoAddWrBsy	NewLoAddWr	Preloading address counter’s Low Byte
HiAddWrBsy	NewHiAddWr	Preloading address counter’s High Byte
LoAddWrBsy & HiAddWrBsy	RstAddressCntr	Resetting the entire address counter to zeroes

There are two byte-wide data ports between the MEMioEngine and the main engine; one in each direction. When a write of data to the memory, or to either the upper or lower byte of the address counter is to be executed, the main engine holds the data it is presenting to the MEMioEngine on its output port called MemOutHold. It then pulses high for 100nS the appropriate request for service signal: NewMemWr, NewLoAddWr, or NewHiAddWr, respectively. The corresponding “busy” signal will assert: MemWrBsy, LoAddWrBsy, or HiAddWrBsy, respectively. When the “busy” signal negates (low) the operation is complete, and the main engine is free to perform another task. The information provided by the main engine on MemOutHold should be held steady while the “busy” signal is asserted.

When a read of a single byte of data from the memory is to be executed, the main engine simply pulses high (100nS) the RdMemRq signal. The MemRdBsy signal will be asserted on the same clock edge that actually registers the read data into the MemInPipe registers. This byte of read data is available for retrieval by the main engine when the “busy” signal is asserted. After the main engine has moved the memory read data from the MemInPipe to its destination, the main engine asserts a high pulse (100nS) on GotMemRd which negates the read memory “busy” signal, freeing the MEMioEngine to accept another services request.

If multiple, back-to-back, continuous reading of memory is desired, then the main engine asserts (and keeps it high) the RdMemRq service request signal. The main engine is stating “keep reading and posting data into MemInPipe until I tell you to quit.” When “new” data has been read and posted into MemInPipe, the “busy” signal asserts and the main engine can retrieve the data. When it has moved the data to its destination, then the main engine pulses high (100nS) the GotMemRd handshake line. This clears the “busy” signal and enables the state machine to automatically fetch the next byte of information. This process continues until the main engine no longer wants anymore read data. During the last byte read operation, while the “busy” signal is asserted but prior to the GotMemRd pulse generation, the main engine should negate the request for service by negating (low) the RdMemRq signal.

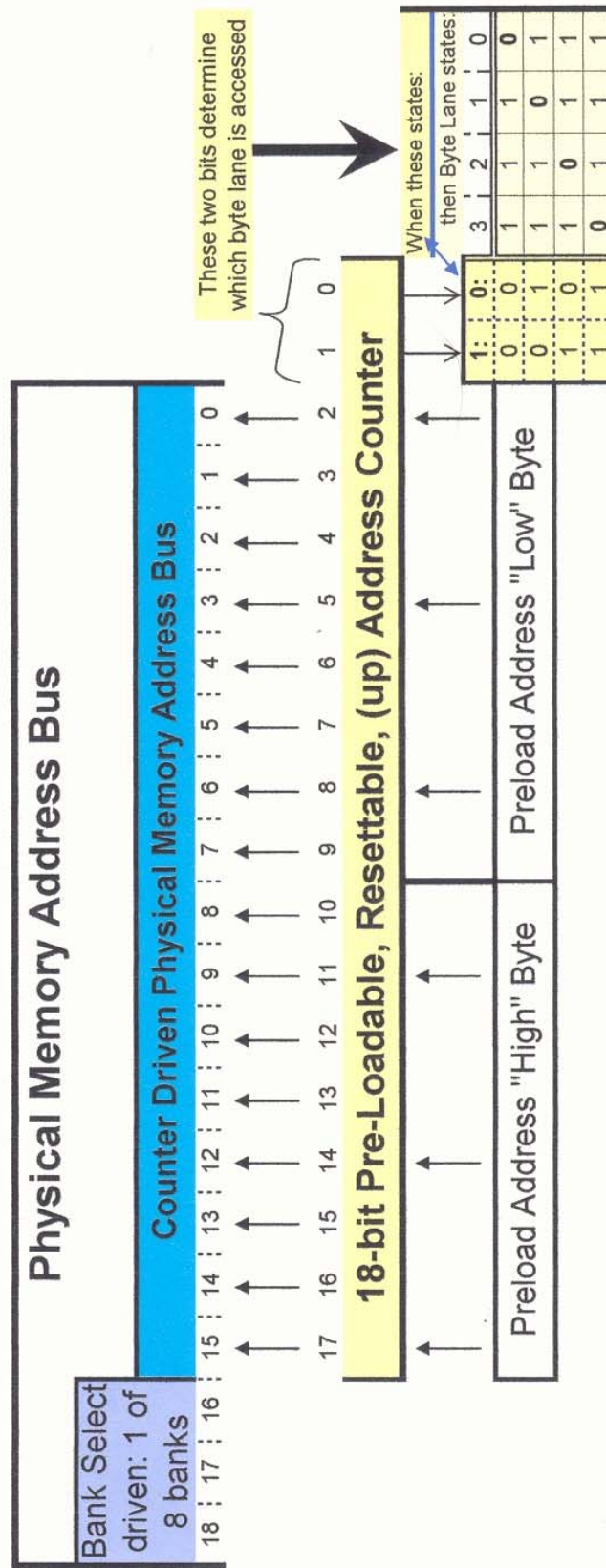
Page 4 shows the Physical Memory Address Generator Scheme that has been implemented in this VHDL code. Later, a detailed description of its operation will show how the automatic address counting works. Page 5 presents the memory mapping scheme, byte lane control, and the “bank” selection model for this memory and its interface circuits.

Memory I/O Engine subdesign Design, VHDL, & Simulation Notes

Printed:
7/24/2005

SFC04 Project: **Physical Memory Address Generator Scheme**

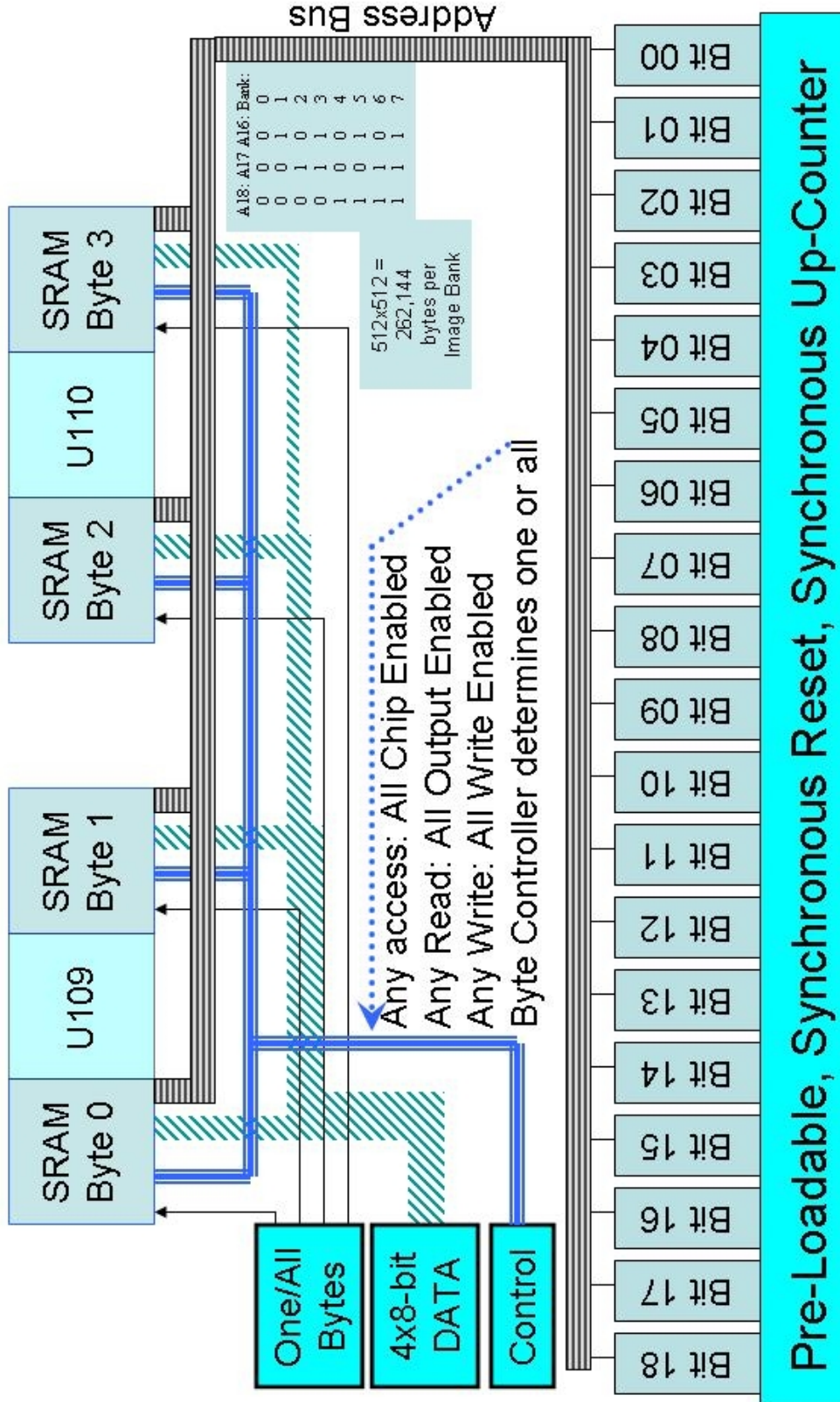
Developed
by GUSTECH



NOTE: To maintain orderly operations, the least significant two bits of the address counter (that drive the byte lane control signals) are always reset to "0" on any Reset or address preload operation.

Memory Mapping Scheme Byte Lane Control & Banks

Access can be 32-bit wide (image processing), or 8-bit wide (Host read & write interfacing)



Memory I/O Engine subdesign Design, VHDL, & Simulation Notes

As eluded to on page 5's diagram, the MEMioEngine can be modified, if desired, to provide faster, 32-bit wide read and write operations for image processing tasks. This can be done with just a few changes to the MEMioEngine's VHDL, mostly in the byte lane enable circuit area, and the port widths used between this subdesign and that of the main engine's.

MEMioEngine Interface Signals:

This subdesign has two primary ports; one is with the physical memory chips external to this Xilinx part, and the other is with the main engine, contained (future past tense) within this Xilinx part.

to/from SRAM Devices...

There is a bidirectional 32-bit data interface between the MEMioEngine and the physical SRAM devices. The VHDL code calls this Sram_DQ.

There are 10 control lines broken into two sets of five each, one set per physical SRAM device. Per device there is a chip enable signal, a write enable signal, a read output enable signal and two byte lane enable signals. All ten control lines are asserted low signals. The VHDL code calls these signals: Sram_Bxy_E, Sram_Bxy_W, Sram_Bxy_G, and Sram_B(3:0), respectively, where "xy" indicates byte lanes "01" for U109 and "23" for U110.

These generic memory devices, STMicroelectronics' M68AW512M, have three different methods for read data, and three different methods for writing data. The read method implemented in this VHDL code is called "Output Enable Controlled" Read mode, where the shortest asserted signal is Sram_Bxy_G. Both "gate" Output Enable signals are asserted simultaneously during read operations during Memstates 5 & 6. The longer chip enable signals are both asserted low during read operations during Memstates 4, 5, & 6. Since only one byte lane is enabled at a time by the address controller, only one byte of the four will be actively driven from the memory devices.

The write method implemented in this VHDL code is called "Write Enable Controlled" Write mode. As implied by the name, it is the Sram_Bxy_W signal that actually writes data into memory, one byte at a time. The registration edge is the rising edge, occurring on the transition from the asserted state to the negated state. For the write sequence, both "write" enable signals are asserted simultaneously during Memstate , with the memory registration occurring during the move to Memstate 3. The longer chip enable signals are both asserted low during write operations during Memstates 1, 2, & 3. Since only one byte lane is enabled at a time by the address controller, only one byte of the four will be actively latched into the memory devices.

Note: the address bus is always active. The memory devices are held in standby when the Memstate engine is "waiting," for lower power consumption.

Memory I/O Engine subdesign Design, VHDL, & Simulation Notes

to/from “main engine”...

There is an 8-bit unidirectional registered data output port called MemInPipe (referenced to the main engine; “IN” to it) that provides read data from the memory to the main engine. It is valid while the read operation’s “busy” signal “MemRdBsy” is asserted.

There is an 8-bit unidirectional static data input port called DataOutHold (referenced to the main engine; “OUT” from it) that provides “write” data to the memory or address counter preloads. This may be registered at the main engine’s side of the design. I must remain stable during all three “busy” indications where this data is the source for the writes.

There are five service request signals coming from the main engine that are acted upon by the MEMioEngine. These were covered in detail on page 2.

There are five different “busy” indications, one unique indication per service request type. These were mostly covered in detail on page 3. What was not discussed there is the special busy state of two signals that indicate that a reset of the address counter “is occurring.” Unlike the separate low and high byte preloads of the address counter, where only one busy signal is asserted (Low or High), when the service request is “Reset the whole address counter” invoked by the 100nS high pulse on RstAddressCntr, both LoAddWrBsy and HiAddWrBsy assert simultaneously for the duration of the operation.

An extra handshake line is added for orderly control of read operations to indicate that the read data registered for the main engine in MemInPipe, has been retrieved, and that the MEMioEngine is free to do something else (more reads or more waiting). The interaction of these signals were also discussed in detail on page 3.

As presented at the bottom of page 4 and the top of page 5, there are eight separate banks of memory, each $\frac{1}{4}$ MByte in size. The 3 BankSelect signals from the main engine determine which bank is used by manipulating the 3 most significant bits of the memory devices’ address lines, **statically**. Therefore, the main engine should never change these lines while any service requests or “busy” signals are asserted active.

from System resources...

There are only two remaining signals not discussed so far. The main system clock, called sysclk, is a 10MHz free running clock for this subdesign’s process statements. The push button #1 caused, low assertion, global asynchronous reset is used to initialize all of the signals included in the synchronous process statements.

Memory I/O Engine subdesign Design, VHDL, & Simulation Notes

MEMioEngine Interface Operational Specifications How to use the “protocol”!

A simple, step-by-step description of each service type will be presented, focusing on the movement of signals in and out of this subdesign as it pertains to the interests of the main engine, since it will be communicating with the MEMioEngine in a “command-response” protocol. Details of all states for each service type are presented in the presentation of the Simulation Results for the whole design, much later in this document.

Write to the “low” Byte of the Address Counter.

Please refer back to the diagram on page 4 to see where the low byte preloads the address counter.

1. Main Engine places the new “low” byte for the address counter on the **DataOutHold** port, and keeps it steady (registered?).
2. Main Engine pulses high for 100nS the **NewLoAddWr** service request signal.
3. Main Engine “notices” the assertion state of the **LoAddWrBsy** “busy” signal.
4. Main Engine “knows” the low byte address preload is complete when **LoAddWrBsy** returns to its negated “low” state.

Write to the “high” Byte of the Address Counter.

Please refer back to the diagram on page 4 to see where the high byte preloads the address counter.

1. Main Engine places the new “high” byte for the address counter on the **DataOutHold** port, and keeps it steady (registered?).
2. Main Engine pulses high for 100nS the **NewHiAddWr** service request signal.
3. Main Engine “notices” the assertion state of the **HiAddWrBsy** “busy” signal.
4. Main Engine “knows” the high byte address preload is complete when **HiAddWrBsy** returns to its negated “low” state.

Reset the Whole Address Counter.

A reset of the whole address counter also resets the byte lane selection to zero.

1. Main Engine pulses high for 100nS the **RstAddressCntr** service request signal.
2. Main Engine “notices” the assertion state of both the **LoAddWrBsy** and **HiAddWrBsy** “busy” signals.
3. Main Engine “knows” the address counter’s reset is complete when both **LoAddWrBsy** and **HiAddWrBsy** return to their negated “low” states.

Memory I/O Engine subdesign

Design, VHDL, & Simulation Notes

Write a byte to Memory.

Every “write” access to memory also automatically increments the address counter, cycling through all four byte lanes 0→1→2→3 for each physical memory address location. If the memory “write” is not to be placed into the next byte location, then preload the address counter with the correct address using the protocol presented on page 8. If the new location with a new address is not in byte lane 0, then perform the necessary number of “dummy” read operations to increment the byte lane count until the correct byte lane is ready for the required “new” write to memory.

1. Main Engine places the write to memory “byte” on the **DataOutHold** port, and keeps it steady (registered?).
2. Main Engine pulses high for 100nS the **NewMemWr** service request signal.
3. Main Engine “notices” the assertion state of the **MemWrBsy** “busy” signal.
4. Main Engine “knows” the memory write operation is complete when **MemWrBsy** returns to its negated “low” state.

Read just one byte from Memory.

Every read access from memory also automatically increments the address counter, cycling through all four byte lanes 0→1→2→3 for each physical memory address location. If the memory “read” is not from the next byte location, then preload the address counter with the correct address using the protocol presented on page 8. If the new location with a new address is not in byte lane 0, then perform the necessary number of “dummy” read operations to increment the byte lane count until the correct byte lane is ready for the required “new” read from memory.

1. Main Engine pulses high for 100nS the **RdMemRq** service request signal.
2. Main Engine “**WAITS FOR**” the assertion state of the **MemRdBsy** “busy” signal.
3. When **MemRdBsy** is asserted, the main engine “reads” the memory data from its **MemInPipe** port and moves the data appropriately to its next destination.
4. Main Engine pulses high for 100nS the **GotMemRd** handshake signal.
5. Main Engine “knows” the memory read operation is complete when **MemRdBsy** returns to its negated “low” state.

Memory I/O Engine subdesign Design, VHDL, & Simulation Notes

Read continuous from Memory.

1. Main Engine asserts high the **RdMemRq** service request signal.
2. Main Engine “**WAITS FOR**” the assertion state of the **MemRdBsy** “busy” signal.
3. When MemRdBsy is asserted, the main engine “reads” the memory data from its **MemInPipe** port and moves the data appropriately to its next destination.
4. Main Engine pulses high for 100nS the **GotMemRd** handshake signal.
5. Main Engine “knows” the memory read operation is complete when **MemRdBsy** returns to its negated “low” state.
6. Return to step 2 until the “last” byte is being read.
7. If “last” byte then:
 - a. between steps 3 and 4 insert:
 - b. Main Engine negates low the **RdMemRq** service request signal.

100nS Pulse Generator Example:

Note, the color coding used below is representative of Xilinx tools for VHDL source files.

Within a “sysclk” process statement include:

```
if <conditions for pulse assertion are satisfied> then
    REQUESTSignal <= '1'; -- assert signal pulse high
elsif REQUESTSignal = '1' then
    REQUESTSignal <= '0'; -- negate signal low
end if;
```

Also, within the asynchronous reset portion of the process statement include:

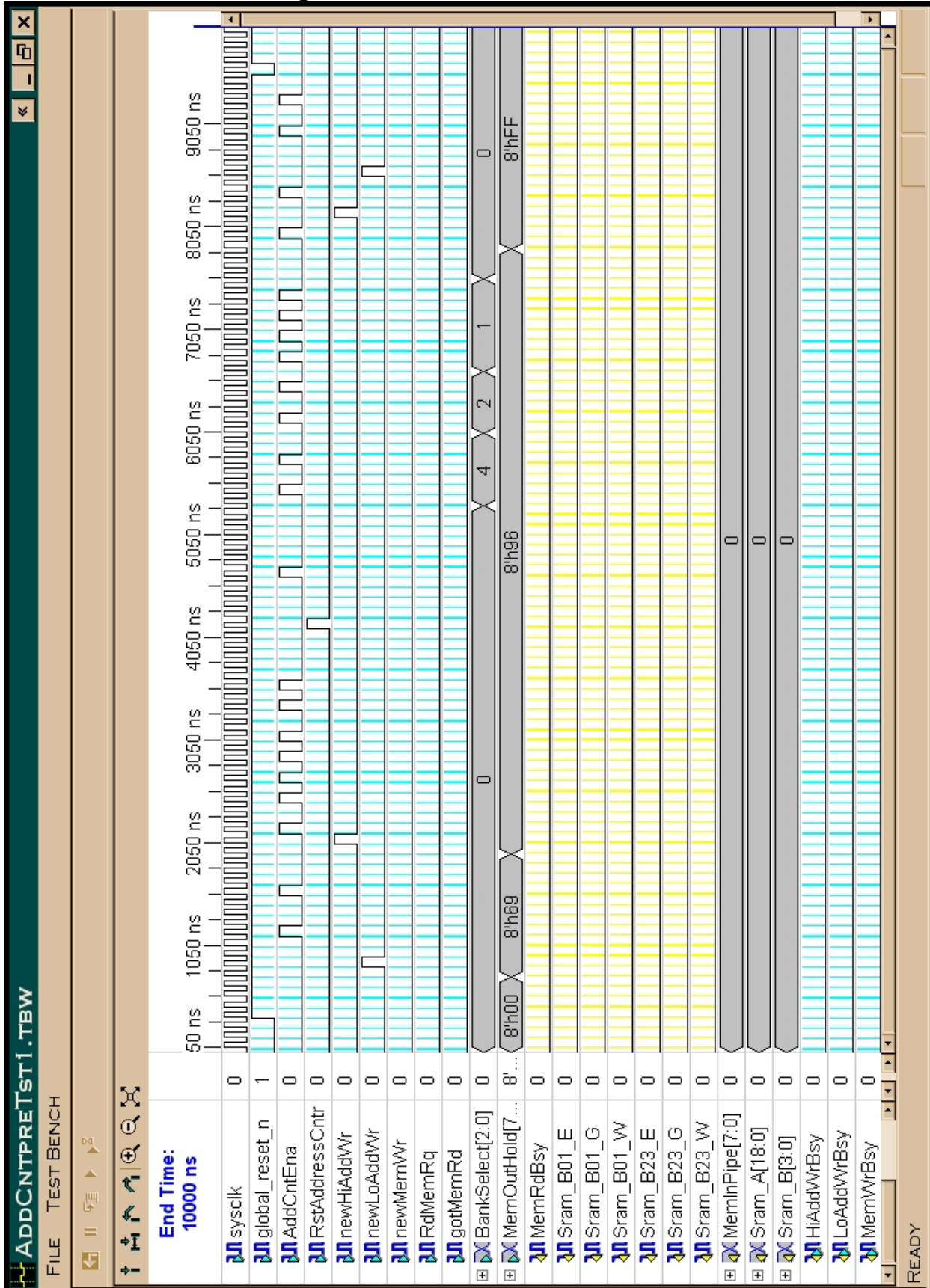
```
REQUESTSignal <= '0'; -- initialize signal low
```

Develop the conditions for pulse assertion carefully so that unintended pulse repetitions do NOT occur.

Address Controller subset of design:

The next page presents the TestBench waveform that was used to independently test a subset of the MEMioEngine directly involved with handling the address counter functions. The emulated “AddCntEna” signal is fully qualified in the complete subdesign VHDL code.

Memory I/O Engine subdesign Design, VHDL, & Simulation Notes



Memory I/O Engine subdesign Design, VHDL, & Simulation Notes

MEMioEngine Address Controller Simulation Results

At $\approx 0.8\mu\text{S}$ the main engine emulates the service request to preload the low address byte, with “x69” on the MemOutHold port, and with “000” on the BankSelect lines. The AddCntEna pulse occurring at $\approx 1.15\mu\text{S}$ preloads the address counter to a new value of “x001A4” which drives the address bus to a new value of “x00069.” The byte lane select remains at the original reset value of 0. The “busy” signal for loading the low byte of address information is asserted until the AddCntEna pulse is detected.

At $\approx 1.5\mu\text{S}$ AddCntEna is pulsed again. Its high state during the next rising edge of sysclk causes the address counter to increment to “x001A5” which causes a new byte lane select of 1, while keeping the physical memory’s address bus at “x00069.”

At $\approx 2.05\mu\text{S}$ the main engine emulates the service request to preload the high address byte, with “x96” on the MemOutHold port. The AddCntEna pulse occurring at $\approx 2.15\mu\text{S}$ preloads the address counter to a new value of “x259A4” which drives the address bus to a new value of “x09669.” The byte lane select has been reset to a value of 0. The “busy” signal for loading the high byte of address information is asserted until the AddCntEna pulse is detected.

At $\approx 2.45\mu\text{S}$ AddCntEna is pulsed again. Its high state during the next rising edge of sysclk causes the address counter to increment to “x259A5” which causes a new byte lane select of 1, while keeping the physical memory’s address bus at “x09669.”

At $\approx 2.65\mu\text{S}$ AddCntEna is pulsed again. Its high state during the next rising edge of sysclk causes the address counter to increment to “x259A6” which causes a new byte lane select of 2, while keeping the physical memory’s address bus at “x09669.”

At $\approx 2.85\mu\text{S}$ AddCntEna is pulsed again. Its high state during the next rising edge of sysclk causes the address counter to increment to “x259A7” which causes a new byte lane select of 3, while keeping the physical memory’s address bus at “x09669.”

At $\approx 3.05\mu\text{S}$ AddCntEna is pulsed again. Its high state during the next rising edge of sysclk causes the address counter to increment to “x259A8” which causes a new byte lane select of 0, and changing the physical memory’s address bus to the next location of “x0966A.”

At $\approx 3.35\mu\text{S}$ AddCntEna is pulsed again. Its high state during the next rising edge of sysclk causes the address counter to increment to “x259A9” which causes a new byte lane select of 1, while keeping the physical memory’s address bus at “x0966A.”

At $\approx 3.55\mu\text{S}$ AddCntEna is pulsed again. Its high state during the next rising edge of sysclk causes the address counter to increment to “x259AA” which causes a new byte lane select of 2, while keeping the physical memory’s address bus at “x0966A.”

Memory I/O Engine subdesign Design, VHDL, & Simulation Notes

At $\approx 4.15\mu\text{s}$ the RstAddressCntr service request is pulsed high for 100nS. Its high state during the next rising edge of sysclk causes the address counter to completely reset. This has the subsequent effect of driving address “x00000” out to the physical memory address bus, and also selecting byte lane 0 (it was last on byte lane 2). The two “busy” signals for resetting the entire address counter are asserted until the AddCntEna pulse is detected.

At $\approx 5.35\mu\text{s}$ the BankSelect lines from the main engine are changed from “000” to “100” which is a selection of bank 4. This statically changes the physical memory address bus value to “x40000.” Remember that BankSelect changes should only occur during “non-busy” times, and that it has no effect on the address counter state. The existing byte lane value will not change due to a change in the BankSelect lines.

At $\approx 5.45\mu\text{s}$ AddCntEna is pulsed again. Its high state during the next rising edge of sysclk causes the address counter to increment to “x00001” which causes a new byte lane select of 1, while keeping the physical memory’s address bus at “x40000.”

At $\approx 5.75\mu\text{s}$ AddCntEna is pulsed again. Its high state during the next rising edge of sysclk causes the address counter to increment to “x00002” which causes a new byte lane select of 2, while keeping the physical memory’s address bus at “x40000.”

At $\approx 6.05\mu\text{s}$ the BankSelect lines from the main engine are changed from “100” to “010” which is a selection of bank 2. This statically changes the physical memory address bus value to “x20000.” The byte lane select remains at 2.

At $\approx 6.25\mu\text{s}$ AddCntEna is pulsed again. Its high state during the next rising edge of sysclk causes the address counter to increment to “x00003” which causes a new byte lane select of 3, while keeping the physical memory’s address bus at “x20000.”

At $\approx 6.45\mu\text{s}$ AddCntEna is pulsed again. Its high state during the next rising edge of sysclk causes the address counter to increment to “x00004” which causes a new byte lane select of 0, and changes the physical memory’s address bus to “x20001.”

At $\approx 6.65\mu\text{s}$ the BankSelect lines from the main engine are changed from “010” to “001” which is a selection of bank 1. This statically changes the physical memory address bus value to “x10001.” The byte lane select remains at 0.

At $\approx 6.75\mu\text{s}$ AddCntEna is pulsed again. Its high state during the next rising edge of sysclk causes the address counter to increment to “x00005” which causes a new byte lane select of 1, while keeping the physical memory’s address bus at “x10001.”

At $\approx 6.95\mu\text{s}$ AddCntEna is pulsed again. Its high state during the next rising edge of sysclk causes the address counter to increment to “x00006” which causes a new byte lane select of 2, while keeping the physical memory’s address bus at “x10001.”

Memory I/O Engine subdesign Design, VHDL, & Simulation Notes

At $\approx 7.15\mu\text{S}$ AddCntEna is pulsed again. Its high state during the next rising edge of sysclk causes the address counter to increment to “x00007” which causes a new byte lane select of 3, while keeping the physical memory’s address bus at “x10001.”

At $\approx 7.35\mu\text{S}$ AddCntEna is pulsed again. Its high state during the next rising edge of sysclk causes the address counter to increment to “x00008” which causes a new byte lane select of 0, changing the physical memory’s address bus to “x10002.”

At $\approx 7.55\mu\text{S}$ the BankSelect lines from the main engine are changed from “001” to “000” which is a selection of bank 0. This statically changes the physical memory address bus value to “x00002.” The byte lane select remains at 0.

At $\approx 7.85\mu\text{S}$ the MemOutHold information is changed by the main engine to “xFF,” in anticipation of changing the high byte address information.

curve ball... ☹

At $\approx 7.95\mu\text{S}$ AddCntEna is pulsed again (instead of doing a high byte address counter preload). Its high state during the next rising edge of sysclk causes the address counter to increment to “x00009” which causes a new byte lane select of 1, while keeping the physical memory’s address bus to “x00002.” (the extra AddCntEna was added to show the movement of the byte lane select from its current location of 1 back to 0 on the next event:)

At $\approx 8.15\mu\text{S}$ the main engine emulates the service request to preload the high address byte, with “xFF” on the MemOutHold port, and with “000” still on the BankSelect lines. The AddCntEna pulse occurring at $\approx 8.35\mu\text{S}$ preloads the address counter to a new value of “x3FC08” which drives the address bus to a new value of “x0FF02.” The byte lane select is reset to a value of 0. The “busy” signal for loading the high byte of address information is asserted until the AddCntEna pulse is detected.

At $\approx 8.55\mu\text{S}$ the main engine emulates the service request to preload the low address byte, with “xFF” still on the MemOutHold port, and with “000” still on the BankSelect lines. The AddCntEna pulse occurring at $\approx 8.95\mu\text{S}$ preloads the address counter to a new value of “x3FFFC” which drives the address bus to a new value of “x0FFFF.” The byte lane select is reset to a value of 0. The “busy” signal for loading the high byte of address information is asserted until the AddCntEna pulse is detected.

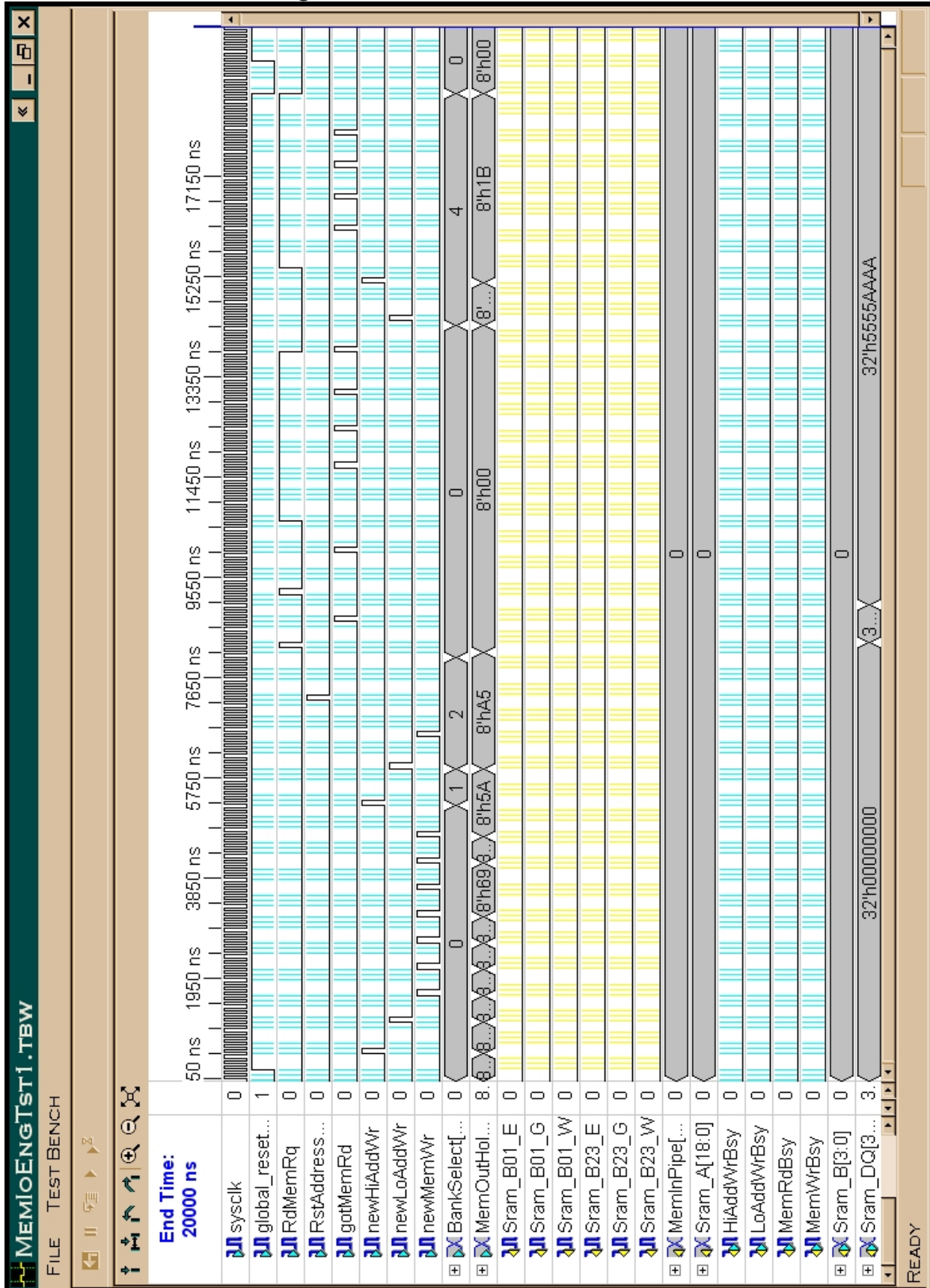
At $\approx 9.25\mu\text{S}$ AddCntEna is pulsed again. Its high state during the next rising edge of sysclk causes the address counter to increment to “x3FFFD” which causes a new byte lane select of 1, while keeping the physical memory’s address bus to “x0FFFF.”

At $\approx 9.55\mu\text{S}$ the global_reset_n signal is asserted low. The Address counter is zeroed, which causes the byte lane to go to 0, and the physical memory address bus = “x0000.”

The next page presents the Test Bench waveform used to simulate the entire finished MEMioEngine VHDL code.

Memory I/O Engine subdesign

Design, VHDL, & Simulation Notes



Memory I/O Engine subdesign Design, VHDL, & Simulation Notes

Dual-VHDL drive of Tristate Bus Difficulty Note:

A problem with simulating dual VHDL packages driving a common tristate bus was encountered again. In the USBioEngine subdesign, the tristate shared bus was split into two pieces during simulations only. That was not done again for the MEMioEngine design verification process. The impact of this problem occurs during read operations. The first read “failure” actually verifies that proper timing of all of the signals is occurring. What fails is the actual data values being moved from memory to the MemInPipe for the main engine to retrieve the data. This first occurrence will be highlighted when it is discussed in the detailed simulation results below. [See details on page 19, in blue font.](#)

MEMioEngine Simulation Results

During the first $\approx 5\mu\text{s}$ the MEMioEngine is reset, the address counter’s high and low bytes are preloaded, and 7 different writes to memory are executed to sequential address locations.

From the beginning ($T=0\text{nS}$) the `global_reset_n` asserted low zeroes the address counter, and therefore the physical memory’s address bus and the byte lane. `BankSelect` is set to bank 0.

Address Preloading:

At $\approx 0.55\mu\text{s}$ the `MemOutHold` port is driven by the main engine to “xAA” and a service request is issued to preload the high byte of the address counter by the assertion of a 100nS pulse on `NewHiAddWr`. On the next rising edge of `sysclk`, its associated “busy” signal `HiAddWrBsy` is asserted. During the next clock cycle the “internal signal” (emulated in the previous simulation analysis; VHDL code driven here) `AddCntEna` is asserted. On the next rising edge of `sysclk`, `HiAddWrBsy` is negated, `AddCntEna` is negated, and the SRAM’s address bus = “x0AA00.”

At $\approx 1.15\mu\text{s}$ the `MemOutHold` port is driven by the main engine to “x55” and a service request is issued to preload the low byte of the address counter by the assertion of a 100nS pulse on `NewLoAddWr`. On the next rising edge of `sysclk`, its associated “busy” signal `LoAddWrBsy` is asserted. During the next clock cycle the “internal signal” `AddCntEna` is asserted. On the next rising edge of `sysclk`, `LoAddWrBsy` is negated, `AddCntEna` is negated, and the SRAM’s address bus = “x0AA55.”

Start of Seven Memory Writes:

At $\approx 1.65\mu\text{s}$ a write service request is issued by the main controller by the 100nS pulse assertion (high) of `NewMemWr`. On the next rising edge of `sysclk`, the `SRAM_DQ` lines change from all tristated to a drive value of “ZZZZZZFF” since the byte lane select = 0, and the “busy” signal `MemWrBsy` is asserted high. The SRAM’s address bus still = “x0AA55.” The next rising edge of `sysclk` increments the Memstate Engine to state =1 and asserts the memory chip enable signals (low) `SRAM_B01_E` and `SRAM_B23_E`. The next rising edge of `sysclk` increments the Memstate Engine to state =2 and asserts the memory write enable signals (low) `SRAM_B01_W` and `SRAM_B23_W`. The next rising edge of `sysclk` increments the Memstate Engine to state

Memory I/O Engine subdesign

Design, VHDL, & Simulation Notes

=3 and negates the memory write enable signals (high) **SRAM_B01_W and SRAM_B23_W**, which actually performs the write operation into memory. The “internal signal” **AddCntEna** is also asserted high in order to cause an address increment to occur on the next **sysclk**. The next rising edge of **sysclk** increments the Memstate Engine back to state =0 and negates the memory chip enable signals (high) **SRAM_B01_E and SRAM_B23_E**, and releases the **SRAM_DQ** drivers back to their tristate condition, and negates the “busy” signal **MemWrBsy** so that the main engine knows that the write operation is now complete.

At $\approx 2.15\mu\text{s}$ the **MemOutHold** port is driven by the main engine to “x00” and a write service request is issued by the main controller by the 100nS pulse assertion (high) of **NewMemWr**. On the next rising edge of **sysclk**, the **SRAM_DQ** lines change from all tristated to a drive value of “ZZZZ00ZZ” since the byte lane select = 1, and the “busy” signal **MemWrBsy** is asserted high. The SRAM’s address bus still = “x0AA55.” The next rising edge of **sysclk** increments the Memstate Engine to state =1 and asserts the memory chip enable signals (low) **SRAM_B01_E and SRAM_B23_E**. The next rising edge of **sysclk** increments the Memstate Engine to state =2 and asserts the memory write enable signals (low) **SRAM_B01_W and SRAM_B23_W**. The next rising edge of **sysclk** increments the Memstate Engine to state =3 and negates the memory write enable signals (high) **SRAM_B01_W and SRAM_B23_W**, which actually performs the write operation into memory. The “internal signal” **AddCntEna** is also asserted high in order to cause an address increment to occur on the next **sysclk**. The next rising edge of **sysclk** increments the Memstate Engine back to state =0 and negates the memory chip enable signals (high) **SRAM_B01_E and SRAM_B23_E**, and releases the **SRAM_DQ** drivers back to their tristate condition, and negates the “busy” signal **MemWrBsy** so that the main engine knows that the write operation is now complete.

Exactly the same sequence of events happens again starting at $\approx 2.65\mu\text{s}$ with the write data at the **MemOutHold** port = x”96.” The **SRAM_DQ** lines = “ZZ96ZZZZ” and the byte lane select = 2 for this write operation. The SRAM’s address bus still = “x0AA55.”

Exactly the same sequence of events happens again starting at $\approx 3.15\mu\text{s}$ with the write data at the **MemOutHold** port = x”69.” The **SRAM_DQ** lines = “69ZZZZZZ” and the byte lane select = 3 for this write operation. The SRAM’s address bus still = “x0AA55.” However, at the transition from Memstate = 3 back to Memstate = 0, the address counter’s automatic increment changes the SRAM’s address bus to “x0AA56” and the byte lane select again = 0.

Exactly the same sequence of events happens again starting at $\approx 3.65\mu\text{s}$ with the write data at the **MemOutHold** port still = x”69.” The **SRAM_DQ** lines = “ZZZZZZ69” and the byte lane select = 0 for this write operation. The SRAM’s address bus still = “x0AA56.”

Exactly the same sequence of events happens again starting at $\approx 4.15\mu\text{s}$ with the write data at the **MemOutHold** port = x”A5.” The **SRAM_DQ** lines = “ZZZZA5ZZ” and the byte lane select = 1 for this write operation. The SRAM’s address bus still = “x0AA56.”

Exactly the same sequence of events happens again starting at $\approx 4.65\mu\text{s}$ with the write data at the **MemOutHold** port = x”5A.” The **SRAM_DQ** lines = “ZZ5AZZZZ” and the byte lane select = 2 for this write operation. The SRAM’s address bus still = “x0AA56.”

Memory I/O Engine subdesign Design, VHDL, & Simulation Notes

The write repetition rate is running at ½ MByte per second, one every 500nS.

This concludes the back-to-back write sequence testing.

Another Address Preloading:

At $\approx 5.25\mu\text{s}$, while the MemOutHold port is driven by the main engine to “x5A” a service request is issued to preload the high byte of the address counter by the assertion of a 100nS pulse on **NewHiAddWr**, and the BankSelect is changed to = 1. On the next rising edge of sysclk, its associated “busy” signal **HiAddWrBsy** is asserted. During the next clock cycle the “internal signal” AddCntEna is asserted. On the next rising edge of sysclk, HiAddWrBsy is negated, AddCntEna is negated, and the SRAM’s address bus = “x15A56.”

At $\approx 5.95\mu\text{s}$ the MemOutHold port is driven by the main engine to “xA5” and a service request is issued to preload the low byte of the address counter by the assertion of a 100nS pulse on **NewLoAddWr**, and the BankSelect is changed to = 2. On the next rising edge of sysclk, its associated “busy” signal **LoAddWrBsy** is asserted. During the next clock cycle the “internal signal” AddCntEna is asserted. On the next rising edge of sysclk, LoAddWrBsy is negated, AddCntEna is negated, and the SRAM’s address bus = “x25AA5.”

Last Write Operation:

At $\approx 6.55\mu\text{s}$ a write service request is issued by the main controller by the 100nS pulse assertion (high) of **NewMemWr**. On the next rising edge of sysclk, the SRAM_DQ lines change from all tristated to a drive value of “ZZZZZA5” since the byte lane select = 0, and the “busy” signal **MemWrBsy** is asserted high. The SRAM’s address bus still = “x25AA5.” The next rising edge of sysclk increments the Memstate Engine to state =1 and asserts the memory chip enable signals (low) **SRAM_B01_E and SRAM_B23_E**. The next rising edge of sysclk increments the Memstate Engine to state =2 and asserts the memory write enable signals (low) **SRAM_B01_W and SRAM_B23_W**. The next rising edge of sysclk increments the Memstate Engine to state =3 and negates the memory write enable signals (high) **SRAM_B01_W and SRAM_B23_W**, which actually performs the write operation into memory. The “internal signal” AddCntEna is also asserted high in order to cause an address increment to occur on the next sysclk. The next rising edge of sysclk increments the Memstate Engine back to state =0 and negates the memory chip enable signals (high) **SRAM_B01_E and SRAM_B23_E**, and releases the SRAM_DQ drivers back to their tristate condition, and negates the “busy” signal MemWrBsy so that the main engine knows that the write operation is now complete, and it bumps the address counter up 1 count which changes the byte lane select to = 1.

Memory I/O Engine subdesign Design, VHDL, & Simulation Notes

Reset Address Counter:

At $\approx 7.22\mu\text{s}$ the main engine asserts (high) for 100nS the service request line **RstAddressCntr**. On the next rising edge of sysclk the Reset the Address Counter “busy” state is indicated back to the main engine by the simultaneous assertion of the **LoAddWrBsy** *and* **HiAddWrBsy** signals. On the next rising edge of sysclk the internal signal AddCntEna is asserted. On the last rising edge of sysclk during this operation both LoAddWrBsy and HiAddWrBsy signals are negated (low) (“telling” the main engine that the reset of the address counter is now finished), and the internal AddCntEna is negated, and the physical memory address bus now = “x20000.”

First Memory Read Operation:

At $\approx 8.25\mu\text{s}$ a memory read service request is issued by the main engine by the assertion of the **RdMemRq** signal. As it turns out, this particular assertion is a 100nS pulse, which means that there is the current intention of only reading a single byte from memory instead of a continuous read of many bytes.. The next rising edge of sysclk causes the simultaneous assertion (low) of **SRAM_B01_E** and **SRAM_B23_E** which enables both memory chips, and the transition of the MemState from the waiting state = 0 to the first read operation state = 4.

The next rising edge of sysclk causes the simultaneous assertion (low) of **SRAM_B01_G** and **SRAM_B23_G** which enables the “read” output enabled bus drivers on both memory chips, and the transition of the MemState from state = 4 to the second read operation state = 5.

Special simulation (failure?) note: At this time in the TestBench Waveform there was a change of the SRAM_DQ data bus to attempt to drive the memory bus lines to a value of “xAAA555.” However, the TestBench VHDL file used as a simulation driver failed to override the tristate drive coming from the MEMioEng’s (UUT) VHDL file. Looking through help files and the Xilinx web site resources failed to identify any “options” or “settings” that needed to be adjusted to permit logic states of “0” and ‘1’ to always override undriven states of “Z”. ViewLogic VHDL, and other ASIC tools, and the Altera tools all permit this feature for handling tristate buses in simulation. I am sure Xilinx does too; I just can’t seem to figure out how.

The next rising edge of sysclk causes the actual registration of the memory data bus information into the MemInPipe registers, as (erroneously) depicted as being the registration of “ZZZZZZZZ.” The timing is correct, the “data” is not being driven correctly (see blue note above). The sysclk change also causes the transition of the MemState from state = 5 to the last read operation state = 6. The same sysclk edge also causes the assertion of the **MemRdBsy** signal (high) as an indication to the main engine that data is now available to be retrieved from its MemInPipe port. Finally, sysclk also asserts the internal signal AddCntEna in preparation for an automatic address counter incrementation.

On the last rising edge of sysclk during this read operation, SRAM_B01_G, SRAM_B23_G, SRAM_B01E, and SRAM_B23_E are all negated low; and, AddCntEna is negated low; and, the address counter is incremented such that the physical memory address bus still shows “x00000” while the byte lane selected is now = 1.

Memory I/O Engine subdesign Design, VHDL, & Simulation Notes

wait...; we're not quite finished yet...

At $\approx 8.45\mu\text{S}$ the MemRdBsy signal was asserted, and it still is after the read operation has been completed by the MEMioEng. At $\approx 8.72\mu\text{S}$ the main engine sends a 100nS (high) pulse on the **GotMemRd** handshake line, indicating that the information that was just posted in the MemInPipe port has been retrieved. While asserted high, the next rising edge of sysclk at $\approx 8.75\mu\text{S}$ causes the negation of the MemRdBsy signal. This tells the MEMioEng that it is free to continue with the next service request.

Second Memory Read Operation:

At $\approx 9.25\mu\text{S}$ a second memory read service request is issued, this time with the TestBench Waveform attempting to drive the SRAM_DQ lines with "x5555AAAA." Again, the MemInPipe is registered with (the same) "xZZ" data. From now on, MemInPipe registration events are not verified in the simulation results since Z's are only being registered in this dual-VHDL simulation environment. When the entire read operation is completed, the physical memory address bus still = "x00000" but, the byte lane selected is now = 2. The GotMemRd pulse is delivered by the main engine at $\approx 10.05\mu\text{S}$, negating the MemRdBsy signal, completing the read operation.

Four Continuous Read Operations:

At $\approx 10.65\mu\text{S}$ the first of four "continuous" read operations is started. The sequencing is completely identical to a single read operation with the exception of the "state" versus "pulse" characteristic of the service request signal RdMemRq. It is maintained in a high state by the main engine, using the GotMemRd handshake line to clear one read operation before automatically performing the next one at the next byte location from memory. The four, very slow, reads span $\approx 3.5\mu\text{S}$, with the RdMemRq signal being negated by the main engine on the same sysclk edge that the last GotMemRd pulse is asserted. The final physical memory address bus value = "x00001" while the byte lane selected ends with = 2.

The Last Address Preloading:

(this time, low byte is preloaded first, followed by high byte)

At $\approx 14.45\mu\text{S}$, while the MemOutHold port is driven by the main engine to "x92" a service request is issued to preload the high byte of the address counter by the assertion of a 100nS pulse on **NewLoAddWr**, and the BankSelect is changed to = 4. On the next rising edge of sysclk, its associated "busy" signal **LoAddWrBsy** is asserted. During the next clock cycle the "internal signal" AddCntEna is asserted. On the next rising edge of sysclk, LoAddWrBsy is negated, AddCntEna is negated, and the SRAM's address bus = "x40092."

At $\approx 15.15\mu\text{S}$ the MemOutHold port is driven by the main engine to "x1B" and a service request is issued to preload the high byte of the address counter by the assertion of a 100nS pulse on **NewHiAddWr**, and the BankSelect remains unchanged to = 4. On the next rising edge of

Memory I/O Engine subdesign

Design, VHDL, & Simulation Notes

sysclk, its associated “busy” signal **HiAddWrBsy** is asserted. During the next clock cycle the “internal signal” **AddCntEna** is asserted. On the next rising edge of sysclk, **HiAddWrBsy** is negated, **AddCntEna** is negated, and the SRAM’s address bus = “x41B92.” Of course, the byte lane selected = 0.

4-½ more Continuous Read Operations:

Starting on the very next sysclk event following the completion of the high address byte preload event (above), at $\approx 15.45\mu\text{S}$ the first of five “continuous” read operations is started. The sequencing is completely identical to the quad continuous read operations described above, again with the “state” versus “pulse” characteristic of the service request signal **RdMemRq**. It is maintained in a high state by the main engine, using the **GotMemRd** handshake line to clear one read operation before automatically performing the next one at the next byte location from memory. The first four reads are completed with the assertion of the fourth **GotMemRd** pulse occurring at $\approx 17.95\mu\text{S}$. The fifth read operation automatically starts (because **RdMemRq** is still asserted high) at $\approx 18.05\mu\text{S}$ with the transition of the **MemState** $\rightarrow 4$, and the assertion of the SRAM enable signals. The read operation is completed by the **MEMioEngine**, and it remains in the waiting state = 0 for the next **GotMemRd** pulse to clear the still asserted busy signal **MemRdBsy**.

final surprise...

At $\approx 18.75\mu\text{S}$ the system wide **global_reset_n** pulse is asserted low by the push of button #1. The main engine signal is negated by this event, as well as the zeroing of both the **BankSelect** and **MemOutHold** values. Inside the **MEMioEngine** the following changes occur as a result of this reset pulse:

1. Address counter changes from “x06E4D” to “x00000.” This causes:
2. The Physical Memory Address Bus changes from “x41B93” to “x00000”; and,
3. The byte lane select changes from 1 to 0.
4. The “tristated” value registered into **MemInPipe** changes back to “x00.”
5. The memory read busy signal **MemRdBsy** is negated low.

End Of Simulation Results Analysis.
It seems to work just fine.

The next page starts the **MEMioEng** VHDL listing.

Memory I/O Engine subdesign Design, VHDL, & Simulation Notes

```
-----  
-- Company: GUSTECH  
-- Engineer: Tom Gustin  
--  
-- Create Date: 13:38:50 07/24/05  
-- Design Name: MEMioEngine  
-- Module Name: MEMioEng - Behavioral  
-- Project Name: SFC04  
-- Target Device:  
-- Tool versions:  
-- Description: This is the Memory Input Output Engine
```

that interfaces between 16Mbits of static RAM and the "main engine". The hardware interface consists of 32bit wide bidirectional Data and 19 bits of address and 10 access control lines.

There is only one access width used for this SRAM bank. When doing image processing, reads & writes could be 32 bits wide to speed up the process. However, since there is a desire to actually have image processing occur at a "slow" rate to provide sufficient time to "compromise" sensors, this module performs only byte data moves. All the hardware hooks exist to change this VHDL in order to perform 32 bit wide data reads and writes from and to this SRAM bank, should that be a future desire.

Original image sizes are currently fixed at a size of 512 by 512 (bytes). This means that there are actually 8 sets of 512 x 512 banks. The top three memory address lines, A18, A17, & A16, are encoded as bank select signals, for this version of the VHDL.

```
--  
-- Dependencies:  
--  
-- Revision:  
-- Revision 0.01 - File Created  
-- Additional Comments:  
--
```

```
-----  
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
use IEEE.STD_LOGIC_ARITH.ALL;  
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity MEMioEng is  
  Port ( sysclk : in std_logic;  
        -- system 10MHz clock  
        global_reset_n : in std_logic;  
        -- system global reset on low state  
  
        -- SRAM Chip(s) Interface lines:  
        Sram_DQ : inout std_logic_vector(31 downto 0);  
        -- SRAM's bidirectional byte-lane controlled data bus  
        Sram_A : out std_logic_vector(18 downto 0);  
        -- SRAM's 19 bit address; total of 2097152 bytes (2MB)  
        -- Logically divided into eight address banks of
```

Memory I/O Engine subdesign

Design, VHDL, & Simulation Notes

```
-- 1/4MByte each; bank selected (7->0) is determined
-- by natural binary encoding of top three address
-- lines (A18, A17, A16) as driven by the BankSelect inputs
Sram_B : inout std_logic_vector(3 downto 0);
-- SRAM's Byte Lane control signals. Reads and writes
-- of each byte will be controlled by these four lines
-- They really behave as the actual least significant
-- address discriminators. The least significant two bits
-- of the internal 18-bit address counter are decoded for
-- a one-hot asserted low enabling of one byte lane at a time.
Sram_B01_G : out std_logic;
-- SRAM's "gate" Output Enable for Reads for byte lanes
-- 0 & 1 on the SRAM chip U109. This signal is asserted
-- low to actively drive the 16 data outputs. NOTE:
-- This signal is driven simultaneously with SRAM_B23_G.
-- Negating SRAM_B01_G high tristates the SRAM's data bus.
Sram_B23_G : out std_logic;
-- SRAM's "gate" Output Enable for Reads for byte lanes
-- 2 & 3 on the SRAM chip U110. This signal is asserted
-- low to actively drive the 16 data outputs. NOTE:
-- This signal is driven simultaneously with SRAM_B01_G.
-- Negating SRAM_B23_G high tristates the SRAM's data bus.
Sram_B01_E : out std_logic;
-- SRAM's Chip "Enable" is asserted low for both reads and
-- writes, from and to U109, for byte lanes 0 & 1. It is
-- negated high at all other times, placing U109 in
-- standby mode for lower power consumption. SRAM_B01_E
-- is driven simultaneously with SRAM_B23_E.
Sram_B23_E : out std_logic;
-- SRAM's Chip "Enable" is asserted low for both reads and
-- writes, from and to U110, for byte lanes 2 & 3. It is
-- negated high at all other times, placing U110 in
-- standby mode for lower power consumption. SRAM_B23_E
-- is driven simultaneously with SRAM_B01_E.
Sram_B01_W : out std_logic;
-- SRAM's "write" enable is asserted low, and its negation
-- rising edge actually clocks the write data into the
-- selected address location with selected byte location
-- granularity. SRAM_B01_W write enables for U109's byte
-- lanes 0 & 1, and it is driven simultaneously with the
-- SRAM_B23_W signal.
Sram_B23_W : out std_logic;
-- SRAM's "write" enable is asserted low, and its negation
-- rising edge actually clocks the write data into the
-- selected address location with selected byte location
-- granularity. SRAM_B23_W write enables for U120's byte
-- lanes 2 & 3, and it is driven simultaneously with the
-- SRAM_B01_W signal.

-- "Main Engine" Interface signals & data buses
-- Address Bank selection:
BankSelect : in std_logic_vector(2 downto 0);
-- These three bits statically drive the top three memory
-- address lines A18->A16, respectively. They should NOT
-- be changed during the time when ANY of the memory read
-- or write operations are running. (no requests, no busies)

-- writes of data & address preloads from main engine
MemOutHold : in std_logic_vector(7 downto 0);
-- 8 bit "write" data held "steady state" and presented to
-- this port by the main engine. Note, the data could be
-- bound for a byte location in memory, or it could also be
-- a preload of the low or high bytes of the memory address.
-- When the busy signal for the associated "new" service
-- request is negated, indicating the end of the write
-- operation, then MemOutHold information can be changed.
newMemWr : in std_logic;
-- A 100nS high pulse on this input from the main engine is
-- a request to write the information on the MemOutHold input
```

Memory I/O Engine subdesign

Design, VHDL, & Simulation Notes

```
-- port to the current memory address selected, using only the
-- one byte lane selected via SRAM_B(3:0)
    MemWrBsy : inout std_logic;
-- MemWrBsy is a state busy signal that is asserted on the
-- synchronous detection of the high state of the newMemWr
-- pulse from the main engine, indicating an acknowledgment
-- of the request for service for a memory write sequence.
-- This signal is negated at the end of the write operation
-- by this MEMioEngine, informing the main engine that it is
-- ready for another write operation and that the MemOutHold
-- data lines can be changed safely.
    newLoAddWr : in std_logic;
-- A 100nS high pulse on this input from the main engine is
-- a request to write the information on the MemOutHold input
-- port to the the Low Byte Address Preload locations of the
-- pre-loadable address counter. MemOutHold(7->0) will be
-- synchronously preloaded into the current states of bits
-- 9->2 of the counter, which in turn actually drive the
-- memory chip address lines of 7->0, respectively.
    LoAddWrBsy : inout std_logic;
-- LoAddWrBsy is a state busy signal that is asserted on the
-- synchronous detection of the high state of the newLoAddWr
-- pulse from the main engine, indicating an acknowledgment
-- of the request for service for a Low Address preload write.
-- This signal is negated at the end of the write operation
-- by this MEMioEngine, informing the main engine that it is
-- ready for another write operation and that the MemOutHold
-- data lines can be changed safely. This busy signal is
-- also asserted (with the HiAddWrBsy signal) as a busy signal
-- for the RstAddressCntr service request (see below)
    newHiAddWr : in std_logic;
-- A 100nS high pulse on this input from the main engine is
-- a request to write the information on the MemOutHold input
-- port to the the High Byte Address Preload locations of the
-- pre-loadable address counter. MemOutHold(7->0) will be
-- synchronously preloaded into the current states of bits
-- 17->10 of the counter, which in turn actually drive the
-- memory chip address lines of 15->8, respectively.
    HiAddWrBsy : inout std_logic;
-- HiAddWrBsy is a state busy signal that is asserted on the
-- synchronous detection of the high state of the newHiAddWr
-- pulse from the main engine, indicating an acknowledgment
-- of the request for service for a High Address preload write.
-- This signal is negated at the end of the write operation
-- by this MEMioEngine, informing the main engine that it is
-- ready for another write operation and that the MemOutHold
-- data lines can be changed safely. This busy signal is
-- also asserted (with the LoAddWrBsy signal) as a busy signal
-- for the RstAddressCntr service request (see below)
    RstAddressCntr : in std_logic;
-- A 100nS high pulse on this input from the main engine
-- will synchronously create a reset on the 18 bit address
-- counter. The result is that the memory chips address
-- lines 15 -> 0 will all be zeroes, and the byte sequencer
-- will be reset to byte 0 also. The top 3 address lines to
-- the memory chips, A18->A16 are statically controlled by
-- the BankSelect lines from the main engine, the currently
-- active bank. When this pulse is detected, both of the busy
-- signals LoAddWrBsy and HiAddWrBsy will simultaneously be
-- asserted, and both with simultaneously be negated when the
-- address counter's reset is completed.
-- NOTE: This standalone "reset address counter" request signal
-- (although it could) should NOT be asserted when any other
-- busy signals are asserted. Indeterminant behavior will
-- result if attempted.

    MemInPipe : out std_logic_vector(7 downto 0);
-- This 8-bit registered output port drives the last "read"
-- data requested from memory towards the main engine. It
```

Memory I/O Engine subdesign Design, VHDL, & Simulation Notes

```
-- registered just prior to an automatic address incrementation.
RdMemRq : in std_logic;
-- This is an automatic Read Memory Request signal that enables
-- continuous reads, back-to-back, as long as it is asserted
-- high. When detected high, the MEMioEngine will read the data
-- from the "current" bank, address and byte lane location, then
-- register it into the MemInPipe feeding the data to the main
-- engine. When the registration occurs, the MemRdBsy signal is
-- asserted, letting the main engine know that there is new memory
-- read data at its MemInPipe input bus available for moving.
-- If only a single byte is to be read, then this RdMemRq signal
-- should be negated by the main engine prior to the acknowledgment
-- of retrieving the read data from the MemInPipe (by the assertion
-- of the handshake line GotMemRd for 100nS). If continuous reads,
-- or bursts of reads (4 bytes of image data, for instance) are to
-- be fetched, then the duration of this signal's assertion enables
-- more than one byte reads.
    gotMemRd : in std_logic;
-- This asserted high 100nS pulse is an acknowledgment from the main
-- engine that it has fetched the previously posted read data from
-- its MemInPipe port. This event will negate the MemRdBsy signal
-- indicating to the MEMioEng that the last read is completed, and
-- that it should read again if RdMemRq is still asserted high.
MemRdBsy : inout std_logic;
-- This Memory Read Busy "state" signal is asserted when new read
-- data is registered into the MemInPipe feeding the main engine,
-- and it is negated (low) when the handshake line GotMemRd presents
-- a 100nS high pulse assertion, indicating that the main engine
-- retrieved the read data from the MemInPipe and that the MEMioEngine
-- is finished reading (if RdMemRq is negated low) or that it should
-- read the next memory data (if RdMemRq is asserted high for "keep
-- reading until I tell you to stop.")

end MEMioEng;
```

architecture Behavioral of MEMioEng is

```
-- ***** Internal Signal Declarations *****

signal MemAddCntr : std_logic_vector(17 downto 0);
-- 18-bit resettable, preloadable up counter:
-- Bits 0 & 1 are decoded for the byte lane selections.
-- These two bits are reset to "00" during a reset and
-- either/both preload events (see next).
-- Bits 0->9 drive the memory address bus 0->7, and are
-- preloadable as a byte lane called "Low" byte.
-- Bits 10->17 drive the memory address bus 8->15, and are
-- preloadable as a byte lane called "HIGH" byte.
-- Reference separate document for "Physical Memory
-- Address Generator Scheme" for more details.
signal AddCntEna : std_logic;
-- Address counter enabled (when '1') For: synchronous reset,
-- or low byte address preload, or high byte address preload,
-- or for an automatic +1 count incrementation.
constant AddressZeroes : std_logic_vector(17 downto 0)
    := "000000000000000000";
-- Reset 18-bit address counter to all zeroes "constant"

-- Memory read and write state machine State Definitions:
signal MemState : std_logic_vector(2 downto 0);
-- 3-bit sequential state machine; the memory engine; the
-- core of activities to and from the memory chips, from
-- and to the main engine. Its main function is the orderly
-- processing of memory access control lines during memory
-- writes and (single or continuous) reads.

constant memstate0 : std_logic_vector(2 downto 0) := "000"; -- state 0
-- state = 0 is the waiting state between read or write operations
-- There is a conditional launch to state = 1 if a memory write
```

Memory I/O Engine subdesign Design, VHDL, & Simulation Notes

```
-- service request is detected from the main engine
-- There is a conditional launch to state = 4 if a memory read
-- service request is detected from the main engine
constant memstate1 : std_logic_vector(2 downto 0) := "001"; -- state 1
-- unconditional transitioning states: 1->2->3->0
-- Memory Write operation: enable SRAMs & assert low write enables
-- occur on leaving state = 1 going to state = 2
constant memstate2 : std_logic_vector(2 downto 0) := "010"; -- state 2
-- Memory Write operation: negate high write enables
-- occur on leaving state = 2 going to state = 3
constant memstate3 : std_logic_vector(2 downto 0) := "011"; -- state 3
-- End of Write Operation; disable SRAMs
-- occurs on leaving state = 3 going back to state = 0
constant memstate4 : std_logic_vector(2 downto 0) := "100"; -- state 4
-- unconditional transitioning states: 4->5->6->0
-- Memory Read operation: enable SRAMs & assert low "G" Output enables
-- occur on leaving state = 4 going to state = 5
constant memstate5 : std_logic_vector(2 downto 0) := "101"; -- state 5
-- Memory Read operation: Register read data into MemInPipe
-- occurs on leaving state = 5 going to state = 6
constant memstate6 : std_logic_vector(2 downto 0) := "110"; -- state 6
-- End of Read Operation; disable SRAMs
-- occurs on leaving state = 6 going back to state = 0
constant memstate7 : std_logic_vector(2 downto 0) := "111"; -- state 7
-- State = 7 is currently reserved. Go back to state = 0

begin

-- ***** Start of concurrent statements *****

-- Address bus drivers
Sram_A(18 downto 16) <= BankSelect(2 downto 0);
Sram_A(15 downto 0) <= MemAddCntr(17 downto 2);

-- Byte lane select drivers
Sram_B(0) <= '0' when -- enable byte lane 0
  MemAddCntr(1 downto 0) = "00"
  else '1'; -- disable byte lane 0

Sram_B(1) <= '0' when -- enable byte lane 1
  MemAddCntr(1 downto 0) = "01"
  else '1'; -- disable byte lane 1

Sram_B(2) <= '0' when -- enable byte lane 2
  MemAddCntr(1 downto 0) = "10"
  else '1'; -- disable byte lane 2

Sram_B(3) <= '0' when -- enable byte lane 3
  MemAddCntr(1 downto 0) = "11"
  else '1'; -- disable byte lane 3

-- Memory write Data Bus drivers

-- The current "design" setup time is 200nS & hold time is 100nS
-- super safe margins

-- Couple the MemOutHold port to byte lane 3
-- otherwise tristate those drivers
Sram_DQ(31 downto 24) <= MemOutHold when
  (MemWrBsy = '1' and Sram_B(3) = '0')
  else "ZZZZZZZ";

-- Couple the MemOutHold port to byte lane 2
-- otherwise tristate those drivers
Sram_DQ(23 downto 16) <= MemOutHold when
  (MemWrBsy = '1' and Sram_B(2) = '0')
  else "ZZZZZZZ";
```

Memory I/O Engine subdesign

Design, VHDL, & Simulation Notes

```

--          Couple the MemOutHold port to byte lane 1
-- otherwise tristate those drivers
Sram_DQ(15 downto 8) <= MemOutHold when
  (MemWrBsy = '1' and Sram_B(1) = '0')
  else "ZZZZZZZ";

--          Couple the MemOutHold port to byte lane 0
-- otherwise tristate those drivers
Sram_DQ(7 downto 0) <= MemOutHold when
  (MemWrBsy = '1' and Sram_B(0) = '0')
  else "ZZZZZZZ";

-- ***** Start of Process statements *****

MemIO: process (sysclk, global_reset_n)
begin
  if global_reset_n='0' then -- asynchronous reset
    MemAddCntr <= (others => '0');
    MemInPipe <= (others => '0');
    LoAddWrBsy <= '0'; -- clear Low Address Write busy signal
    HiAddWrBsy <= '0'; -- clear High Address Write busy signal
    MemWrBsy <= '0'; -- clear Memory Write busy signal
    MemRdBsy <= '0'; -- clear Memory Read busy signal
    Memstate <= Memstate0; -- initialize Memory State Machine

    AddCntEna <= '0'; -- initialize Address Count Enable
    Sram_B01_W <= '1'; -- negate write pulse for Bytes 0 & 1
    Sram_B23_W <= '1'; -- negate write pulse for Bytes 2 & 3
    Sram_B01_G <= '1'; -- negate output enable for bytes 0 & 1
    Sram_B23_G <= '1'; -- negate output enable for bytes 2 & 3
    Sram_B01_E <= '1'; -- negate SRAM Chip enable for bytes 0 & 1
    Sram_B23_E <= '1'; -- negate SRAM Chip enable for bytes 2 & 3

    -- to the "waiting" state

  elsif sysclk'event and sysclk='1' then -- rising edge of sysclk:

    -- Case sequential engine statements for MemStates...
    -- Moving from state 0, the waiting state, is the only
    -- conditional state in this engine. All other state
    -- changes occur unconditionally. A memory write operation
    -- consists of states: 0->1->2->3->0. A memory read operation
    -- currently uses states: 0->4->5->6->0.
    case memstate is
      when memstate0 => -- test for write service request first.
        -- If there is no write service request, then test for
        -- a read service request. If neither, continue waiting.
        if MemWrBsy = '1' then
          memstate <= memstate1; -- go do write operation
        elsif (RdMemRq = '1' and MemRdBsy = '0') then
          memstate <= memstate4; -- go do read operation
        else
          memstate <= memstate0; -- keep waiting
        end if;

      when memstate1 => -- state = 1: Write: Enable for write
        memstate <= memstate2;

      when memstate2 => -- state = 2: Write: negate prior assertions
        memstate <= memstate3;

      when memstate3 => -- state = 3: Write: finish closing signals
        memstate <= memstate0;

      when memstate4 => -- state = 4: Read Enable for read
        memstate <= memstate5;

      when memstate5 => -- state = 5: Read: Register data into MemInPipe
        memstate <= memstate6;
    end case;
  end if;
end process;

```

Memory I/O Engine subdesign

Design, VHDL, & Simulation Notes

```
when memstate6 => -- state = 6: Read: finish closing signals
    memstate <= memstate0;

when others => -- state = 7 is currently reserved (unused)
    memstate <= memstate0;

end case;

-- end of case section

-- Address Counter Enable (count up one count):
if (AddCntEna = '0' and -- if it is not currently asserted &
    (LoAddWrBsy = '1' or -- an address low byte preload or reset
    HiAddWrBsy = '1' or -- or high byte preload is active, or
    memstate = memstate2 or -- when finishing a write or read of
    memstate = memstate5)) then -- memory then increment the address
    AddCntEna <= '1';
elsif AddCntEna = '1' then -- negate the pulse for 100nS duration
    AddCntEna <= '0';
end if;

-- Address Counter synchronous tasks subsection:
if AddCntEna='1' then -- (see above statement for this signal)
    -- When the address counter is enabled, it is because:
    -- 1) A low byte preload and byte lane reset is requested, or
    -- 2) A high byte preload & byte lane reset is requested, or
    -- 3) A reset of the whole counter & byte lane is requested, or
    -- 4) An incrementation of the address counter is needed.
    if (LoAddWrBsy = '1' and HiAddWrBsy = '0') then
        MemAddCntr(9 downto 2) <= MemOutHold;
        -- Preload address counter with new Low Byte
        MemAddCntr(1 downto 0) <= "00";
        -- and reset the byte lane enable bits, and
        LoAddWrBsy <= '0';
        -- negate the busy signal so that the main engine knows
        -- that the low address byte preload has been done.
    elsif (LoAddWrBsy = '0' and HiAddWrBsy = '1') then
        MemAddCntr(17 downto 10) <= MemOutHold;
        -- Preload address counter with new High Byte
        MemAddCntr(1 downto 0) <= "00";
        -- and reset the byte lane enable bits, and
        HiAddWrBsy <= '0';
        -- negate the busy signal so that the main engine knows
        -- that the High address byte preload has been done.
    elsif (LoAddWrBsy = '1' and HiAddWrBsy = '1') then
        MemAddCntr(17 downto 0) <= AddressZeroes;
        LoAddWrBsy <= '0';
        HiAddWrBsy <= '0';
        -- negate both busy signals so that the main engine
        -- knows that the address counter reset has been done.
    else
        MemAddCntr <= MemAddCntr + 1;
        -- if not a reset or preload event then it can
        -- only be enabled for incrementation purposes.
        -- Bump the counter up one.
        -- The actual assertion of address count enable occurs:
        -- during MemState3 for writes and
        -- during Memstate6 for reads. In both cases, the
        -- actual address change occurs early in state0.
    end if; -- end of synchronous enabled counter tasks
end if; -- end of address counter section

-- Memory I/O and address counter related "busy" signal generators
-- These requests come from the main engine after the data (if any)
-- is presented on MemOutHold (not needed for address counter resets)
-- The asserted busy signals will then be acted upon by the address
-- counter controller (except for memory writes), including their
-- final state negations.

if newMemWr = '1' then -- a Memory write cycle is requested
```

Memory I/O Engine subdesign

Design, VHDL, & Simulation Notes

```
MemWrBsy <= '1'; -- assert Memory Write "busy"
elsif (MemWrBsy = '1' and memstate = memstate3) then
    MemWrBsy <= '0'; -- Write is complete, no longer "busy"
end if;

if newLoAddWr = '1' then -- a low address preload is requested
    LoAddWrBsy <= '1'; -- assert Low Address Write "busy"
end if;

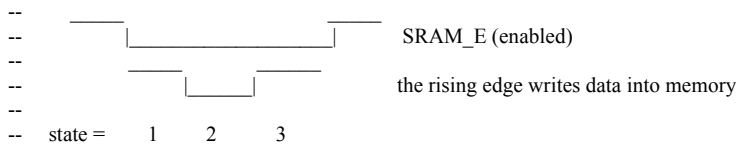
if newHiAddWr = '1' then -- a high address preload is requested
    HiAddWrBsy <= '1'; -- assert High Address Write "busy"
end if;

if RstAddressCntr = '1' then -- a low address preload is requested
    LoAddWrBsy <= '1'; -- assert Low Address Write "busy"
    HiAddWrBsy <= '1'; -- assert High Address Write "busy"
    -- both "busies" indicate a address counter reset is being performed
end if;

-- Memory Read Busy signal handler
if memstate = memstate5 then
    MemRdBsy <= '1'; -- a Memory read cycle has started because
    -- there is no write service request and there is
    -- a (single or continuous) read service request
    -- and the last read (if any) has been retrieved
    -- by the main engine. When the memory read data is
    -- actually registered into the MemInPipe registers,
    -- then assert Memory Read "busy" signal to main engine
elsif (MemRdBsy = '1' and GotMemRd = '1') then
    MemRdBsy <= '0'; -- clear Memory Read Busy signal to main engine
    -- when it indicates it has retrieved the last
    -- read data from the MemInPipe
end if;
```

-- Memory Write control line generator

```
--
--
--
--
--
--
-- state = 1 2 3
```



```
if memstate = memstate1 then
    Sram_B01_W <= '0'; -- assert "low" Write pulse during memory
    Sram_B23_W <= '0'; -- state = 2.
elsif memstate = memstate2 then
    Sram_B01_W <= '1'; -- negate "high" write pulses and end of
    Sram_B23_W <= '1'; -- of state = 2 (writes into memory)
end if;

-- remember that only one byte lane is enabled by the address
-- counter; therefore only one byte is written even though
-- the write pulses for four bytes are driven.
```

```
-- Memory "ENABLE" control lines generator
-- They are enabled during the times when memory reads or writes
-- are being performed, such that they are asserted low during all
-- MEMioEngine states not equal to zero (the waiting state)
if (memstate = memstate0 and
    (MemWrBsy = '1' or
    (RdMemRq = '1' and MemRdBsy = '0'))) then
    SRAM_B01_E <= '0';
    SRAM_B23_E <= '0';
elsif (memstate = memstate3 or
    memstate = memstate6) then
    SRAM_B01_E <= '1';
    SRAM_B23_E <= '1';
```

Memory I/O Engine subdesign Design, VHDL, & Simulation Notes

end if;

```
-- Memory Read Control line generator
--
-- _____ SRAM_E (enabled)
-- _____ SRAM_G (read output enable)
-- _____ Registration edge into MemInPipe
-- state 0 4 5 6 0

if memstate = memstate4 then
    Sram_B01_G <= '0'; -- assert "low" read output enable during
    Sram_B23_G <= '0'; -- memory state = 5 & 6.
elsif memstate = memstate6 then
    Sram_B01_G <= '1'; -- negate "high" at end of      of state = 6
    Sram_B23_G <= '1'; -- memory data was registered into MemInPipe
                        -- on the transition from state = 5 to 6.
end if;

-- remember that only one byte lane is enabled by the address
-- counter; therefore only one byte is "read" even though
-- the read output enables for all four bytes are driven.
```

```
-- Read Memory byte lane multiplexer & MemInPipe registration logic
-- reference "timing" diagram for above Sram_G assignments. The
-- registration of the one byte of data from memory occurs on the
-- transition from memstate=5 to memstate=6.
```

```
if memstate = memstate5 then
    if Sram_B(0) = '0' then -- byte lane 0 memory read
        MemInPipe <= Sram_DQ(7 downto 0);
    elsif Sram_B(1) = '0' then -- byte lane 1 memory read
        MemInPipe <= Sram_DQ(15 downto 8);
    elsif Sram_B(2) = '0' then -- byte lane 2 memory read
        MemInPipe <= Sram_DQ(23 downto 16);
    elsif Sram_B(3) = '0' then -- byte lane 3 memory read
        MemInPipe <= Sram_DQ(31 downto 24);
    end if;
end if;
```

```
end if; -- end of main IF
end process; -- end of MemIO process
```

```
end Behavioral;
```