M3 MBus Implementation

<mbus-team@umich.edu>

Pat Pannuto <ppannuto@umich.edu>
Yoonmyung Lee <sori@umich.edu>
Ye-Sheng Kuo <samkuo@umich.edu>
ZhiYoong Foo <zhiyoong@umich.edu>
Ben Kempke <bpkempke@umich.edu>
David Blaauw <blaauw@umich.edu>
Prabal Dutta <prabal@umich.edu>

Revision 0.4 — July 21, 2015

This document details the implementation of MBus for the M3 project. Other devices compatible with MBus should not require any information from this document. This document covers the hardware design of the M3 MBus implementation.

Contents

1 RTL Design

The M3 MBus implementation defines two major components: a *Bus Controller* and a *Layer Controller*. The bus controller understands the MBus protocol and presents a simple word-wide interface to higher layers. The generic layer controller provides a register file and a memory interface, sufficient for most devices.

In addition, the M3 MBus implementation requires some support blocks: a *Sleep Controller*, a *Wire Controller*, and an *Interrupt Controller*. The are hand-optimized (during layout), always-on components designed for minimal power draw. This design is depicted in Figure 1:

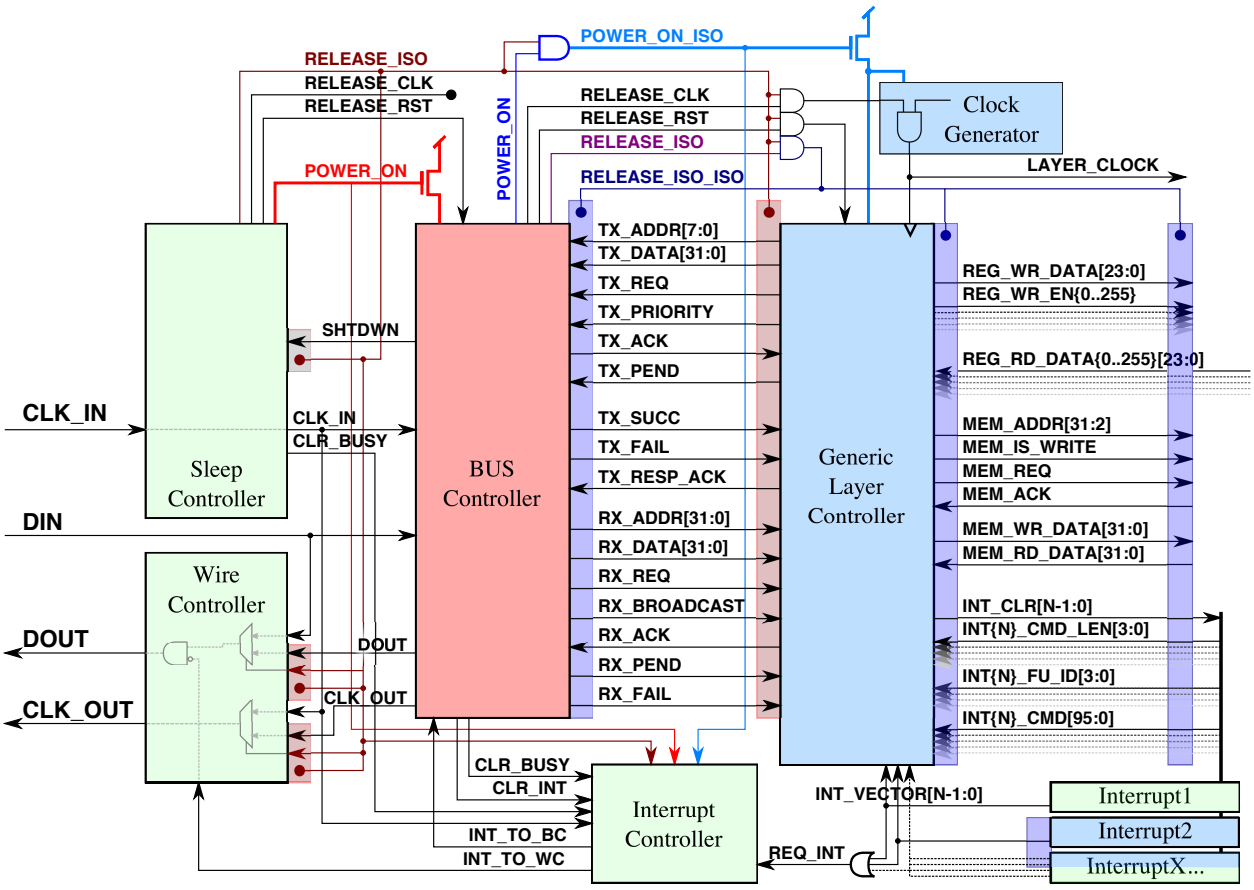

Figure 1: *Implementation block diagram*— The green blocks are always-on components. The two power-gated power domains are shown in red (Bus Controller) and blue (Layer Controller). Module I/O isolation is indicated by the large rectangles controlled by `ISO(LATE)` signals. Circuits elements presented are for conceptual understanding not final design of glitch-free logic. The global reset signal is omitted for clarity.

The four standard power control signals are:

Signal Name	Function	Power-Up	Power-Down
`POWER_ON`	Controls Power-Gating	1st	2nd
`RELEASE_CLK`	Supply Clock to Internal Logic	2nd	2nd
`RELEASE_ISO`	Electrically Isolate Module I/O	3rd	1st
`RELEASE_RST`	(De)Assert Reset	4th	2nd

The `RELEASE_CLK` signal may be omitted if the power-gated layer does not have an internal clock.

1.1 Module Block Details

This section provides details on the inner workings of each of the supplied modules.

1.1.1 Sleep Controller

The job of the sleep controller is to wake the bus controller when a message on the bus begins. The sleep controller is also responsible for powering down the bus controller when requested, using edges from MBus to do so. For layers capable of waking on events other than MBus transactions, the sleep controller may be extended or need to coordinate with other sleep controllers. This document assumes only a basic sleep controller.

Signals

Power Signals The sleep controller uses both edges of the `CLK_IN` signal to generate the power signals. The rising edge that resolves arbitration is used to release the power gating. The falling edge (Priority Drive) is unused and reserved for implementations that require a `RELEASE_CLK` signal. The next rising edge (Priority Latch) drives `RELEASE_RST` and the falling edge (Begin Transmission) drives `RELEASE_ISO`. The next rising edge latches Bit 0 of the transmission.

MBus Signals The sleep controller samples and passes through the MBus `CLK_IN` signal. The sleep controller uses this line to clock its internal state machine. The sleep controller and the bus controller can be safely considered synchronous modules with respect to one another as they both rely on the MBus`CLK_IN` signal to provide a clock.

Module Signals The sleep controller has only one non-reset input signal: `SHTDWN`. This signal is asserted by the bus controller to request that it be put to sleep. This signal is sampled synchronously with the MBus clock. The `SHTDWN` signal may only be asserted during the falling edge Drive Control Bit 1.

The sleep controller is also responsible for generating a `CLR_BUSY` output signal when it powers off the bus controller. This signal is normally generated by the bus controller at the end of MBus transactions and is used to indicated to the interrupt controller that the bus is no longer busy.

1.1.2 Wire Controller

The wire controller is responsible for physically driving the MBus wires. This separation is necessary to ensure that the data and clock lines are forwarded even when the bus controller is power gated. The wire controller is very simple module, conceptually it is little more than a mux that either forwards the values coming in from the bus or values coming from the bus controller. The wire controller also must allow for the interrupt controller to induce a glitch on the bus for wakeup as discussed in the Power Design section of the *MBus Specification*.

Changing the wire controller muxes requires extreme care to ensure that glitches are not introduced. In practice, this means only changing the clock mux signal when `CLK_IN` is already high. The data mux is more complicated. During Idle, the data line is high, any glitches while pulling it low (so long as it ultimately is held low) will be significantly shorter than t_{long} and therefore ignored. During transmission, the state of the data line is not always known when control transitions are required. MBus ignores the data line while the clock is low. Any potentially glitch-inducing changes to the data mux must occur on the falling edge of clock to avoid uncertainty.

The wire controller also facilitates external interrupts as a wakeup source via a *glitch inducer*. The goal of the glitch inducer circuit is to request a bus arbitration cycle and *lose* it. Nominally, this results in no winner of arbitration and the bus will quickly reset. However, the glitch inducer logic must also correctly handle the case where another node was performing genuine arbitration at the same time and a real message takes place.

1.1.3 Interrupt Controller

The interrupt controller is responsible for directing interrupt events to the appropriate modules at the appropriate times. The destination of an interrupt event depends on the current power state of the bus controller and layer controller.

If the layer controller is powered on, the interrupt controller blocks the interrupt signal as the layer controller will handle the interrupt and generate any appropriate messages. If the layer controller is powered off, however, the bus controller must wake the layer controller. To do this, the bus controller must harvest edges from MBus. To generate the power signals, the bus controller must observe a transaction while the `INT_TO_BC` signal is asserted. To generate these edges, the interrupt controller must induce a glitch.

A Subtle Detail: Extreme care must be taken when this glitch is induced, in particular it is important to ensure that MBus is idle so that a real glitch is not created. The simple approach would be to ask the bus controller to simply export a "bus busy" signal, asserted while a transmission is active. Such a signal is not sufficient, however. If a bus controller is not powered on, it is incapable of asserting busy. If two nodes have interrupts near each other in time, the first node will induce a glitch unobserved by the second node. If an interrupt occurs on the second node between end of t_{long} and resultant falling edge of `CLK_IN` and the rising edge for arbitration, the second node will unintentionally and worse unknowingly win arbitration. As the node does not know that it has won arbitration, it will never end the transmission and the MBus will be locked until the master node's watchdog counter expires.

To avert this issue, the interrupt controller keeps an internal sense of "bus busy". The interrupt controller latches the bus as busy whenever `CLK_IN` goes low. The internal busy signal is then cleared by an explicit signal from the bus controller at the end of MBus transactions. As a special case, the sleep controller must clear the busy status when it powers the bus controller down as the bus controller will be powered off at the end of the transaction and incapable of generating the signal. This clear busy signal is internally treated as the last step in the power-down sequence.

1.1.4 Bus Controller

The bus controller is responsible for handling all possible bus events, including generating acknowledgments if it is the target of a transaction. The bus controller is only obligated to hold one word of a transaction, devices wishing to receive messages longer than one word in length must have a local FIFO to store partial messages in their layer controller.

Signals

Power Signals The bus controller provides the same power control signals to the layer controller that the sleep controller provided to the bus controller. For basic layers, the bus controller is responsible for waking and sleeping the layer controller.

The bus controller should only wake the layer controller if there is a message of *non-zero* length being transmitted to the layer's address. In practice this means the bus controller should not begin waking the layer controller until it has received the *third* data bit. Layer controllers should **not** be awoken for broadcast messages.

Once the bus controller elects to begin waking the layer controller, it **must** wake the layer controller completely. Even if another data bit is never sent (Interrupt) after the bus controller elects to wake the layer controller, the bus controller still has the Begin Interrupt, Control Bit 0, Control Bit 1, and Begin Idle edges to use to wake the layer controller.

The bus controller will indicate an erroneous wake-up by asserting `RX_FAIL` signal. The layer controller must acknowledge (`RX_ACK`) the erroneous transmission before the bus controller is capable of receiving another message. This enables the layer controller to take action to put itself back to sleep after a spurious wakeup.

MBus Signals While the bus controller is not directly connected to the external pads for all MBus signals, it does use all MBus inputs:

INPUT	CLK_IN	Bus Clock In
OUTPUT	CLK_OUT	Bus Clock Out
INPUT	DIN	Bus Data In
OUTPUT	DOUT	Bus Data Out

Module Signals

INPUT	RESET	Global system reset
OUTPUT	SHTDWN	Request power gating
INPUT	TX_ADDR[7:0]	Address to transmit
INPUT	TX_DATA[31:0]	Data to transmit
INPUT	TX_REQ	Request to transmit
INPUT	TX_PRIORITY	Is high priority message?
OUTPUT	TX_ACK	Acknowledge request to transmit
INPUT	TX_PEND	More data pending
OUTPUT	TX_SUCC	Transmit successful
OUTPUT	TX_FAIL	Transmit failed
INPUT	TX_RESP_ACK	Acknowledge TX successful / fail
OUTPUT	RX_ADDR[31:0]	Destination address of RX'd packet
OUTPUT	RX_DATA[31:0]	Data received
OUTPUT	RX_REQ	Data is ready
OUTPUT	RX_BROADCAST	RX'd message was a broadcast message
INPUT	RX_ACK	RX_DATA has been saved
OUTPUT	RX_PEND	More data is coming
OUTPUT	RX_FAIL	Abort current RX

The bus controller provides a single-word interface to MBus to higher level modules. To ensure that the bus controller can send a timely ACK for a received message, modules are obligated to be able to (eventually) receive a minimum of one word. That is, if the RX_REQ line is raised, the module *must* eventually receive that word by signaling RX_ACK. If another message is received while RX_REQ is still high, the bus controller will Interrupt the bus indicating "01" (~EoM, TX/RX Error). This serves to (i) save bus bandwidth by canceling a message that will not be received, and (ii) indicate to the transmitting layer that there is still a pending message (or message part) in the receiving layer.

If the TX_PEND signal is asserted but a TX_REQ for a new word is not sent by the time the first word has been sent, the bus controller will Interrupt the bus indicating "01" (~EoM, TX/RX Error). The bus controller will also assert TX_FAIL, at which point the transaction must be aborted.

The RX_PEND signal requires special attention. When the RX_REQ signal rises, if RX_PEND is also high, it indicates that there is more data to follow. If a module asserts RX_ACK in response it is obligating itself to receive at least one more word after the current word[1]. If a module cannot receive another word beyond the word it is currently latching, it should ignore RX_REQ. When the next word is received by the bus controller, it will detect that RX_REQ is still high and abort the entire transaction, Interrupting to indicate RX Error. In practice this means a transmitting node may believe it has sent one more word than was actually received. As the entire transaction is NAK'd however, the only implications are for the software flow control estimation, which can compensate.

For simple nodes that only support single word transactions, this RX_PEND subtlety is important. Such a node should wire its final acknowledgment output signal in a manner such as

```
TX_ACK_out = (RX_PEND_in) ?  1'b0 :  internal_ack_signal;
```

to ensure it does not attempt receipt of multi-word transactions. The TX_PEND signal can simply be tied low.

[1] This is logically a continuation of the original contract. In an idle state, a module has received zero words thus far and is obligated to be able to receive one more. If a module ACKs a word while RX_PEND is high, it is accepting the current word *and* committing to receive the next pending word.

Parameters The bus controller requires relatively little configuration. The only available parameter is the address(es) that this bus instance should respond to:

`ADDRESS`	Address(es) to receive and acknowledge
`ADDRESS_MASK`	Which bits of `ADDRESS` are significant

1.1.5 Generic Layer Controller

The layer controller is responsible for communicating with the bus controller and facilitating multi-word transactions. It federates access to the individual components on a MBus member node. The generic layer controller is designed such that unused modules (e.g. memory access and control) can be synthesized out if they are unused.

Power Signals The generic layer controller can only be woken by the bus controller. Layers capable of generating an interrupt while M3 is in sleep mode (e.g. an alarm) must wire into the glitch inducer via the interrupt controller to wake the layer and the whole M3 system. The generic layer controller does not output any power signals.

There is no explicit shutdown signal from the layer controller to the bus controller. Rather, the layer controller issues a broadcast message announcing to the bus its intention to shut down. The bus controller will recognize the special broadcast message and shut down the layer controller once it is sent. The actual shutdown does not begin until the bus controller successfully transmits the `EoM` bit at the end of the transaction.

Module Signals (Bus Side) All layer controllers have a common set of signals to interface with the bus controller, as shown in Figure 1. Details of these signals are presented in the bus controller documentation: 1.1.4 Module Signals.

Module Signals (Interface Side) The generic layer controller expects to communicate with two types of hardware: a register file and a basic memory. One or both of these may be instantiated.

Register File: The register file is for small, simple configuration, status, or action bits. It presents an 8 bit address space with 24 bit wide data. Writes are pulse-triggered, with a unique write control line for each of the registers. Data to read must always be valid, with no indication that data is being read.

Registers may be smaller than 24 bits. Unused bits for writing should be left disconnected. Unused bits for reading **must** be tied **low**.

OUTPUT	`REG_WR_EN{0..255}`	Register write enable lines
OUTPUT	`REG_WR_DATA[23:0]`	Register data
INPUT	`REG_RD_DATA{0..255}`[23:0]	Wires from registers

Memory: The memory interface is for larger amounts of data (images, audio, packets, etc). Memory is a 32 bit address space with 32 bit data. The memory space does not alias the register file address space.

The memory interface is a simple two-wire handshake that indicates a request when signals are valid and an acknowledgment when the data has been latched. Memory must be fast enough that it can support streaming reads / writes at the line speed of MBus (see Relative Clock Frequencies for detail).

OUTPUT	`MEM_ADDR[31:2]`	Memory address
OUTPUT	`MEM_IS_WRITE`	Read/Write selection
OUTPUT	`MEM_REQ`	Request is ready
INPUT	`MEM_ACK`	Response is ready
OUTPUT	`MEM_WR_DATA[31:0]`	Data to write
INPUT	`MEM_RD_DATA[31:0]`	Data that was read

Module Signals (Interrupt Interface) To generate messages internally, the local node interrupts the layer controller. Interrupt priority is fixed and ranked by the interrupt index. Interrupts are non-interruptible and are serviced completely before handling the next or new interrupt. If an interrupt and a message from the bus arrive at the same time, the interrupt takes priority.

Conceptually, a local interrupt "fakes" the receipt of a message from the bus, thus the data format is the exact same as bus commands. To generate a memory *write* command for example, the interrupt payload is a memory *read* command, because the response to a memory read request is to generate a memory write. The `CMD_LEN` specifies the length in words of the payload that is valid. A length of zero indicates that no command should be executed (the `FU_ID` and `CMD` fields are ignored) and can be cleared immediately – this is useful for generating wakeup requests with no immediate command to execute.

INPUT	`INT_VECTOR[N-1:0]`	Interrupt requests
OUTPUT	`INT_CLR[N-1:0]`	Clear interrupts
INPUT	`INT{N}_CMD_LEN[1:0]`	Word length of payload
INPUT	`INT{N}_FU_ID[3:0]`	Command the LC "receives"
INPUT	`INT{N}_CMD[95:0]`	Command payload

1.2 Module Interdependencies

In addition to the module signals, the following additional requirements must be considered for the blocks to operate correctly.

1.2.1 Clock Domains

The bus controller and layer controller are on separate clock domains. Care must be taken with the `REQ` and `ACK` two-wire handshake to ensure that signals are all double-latched. Double latches should not be required for the other signals are they are only sampled after the `REQ` or `ACK` line is stable.

The generic layer controller does not perform double-latching with any of the signals attached to the register file or memory. These blocks are expected to run on the same clock as the layer controller. Designs that violate this assumption must make modifications to ensure signal stability.

1.2.2 Relative Clock Frequencies

Nominally, the layer controller would only require activity once every 32 MBus clock cycles. In practice, however, there is a double-latched, bidirectional handshake between the bus controller and layer controller. At a minimum then, the layer controller must be at least 1/8 the speed of the bus.

This timing constraint further extends to the memory if longer bus transactions are going to be supported. The generic layer controller is a write-through device, it does not buffer memory requests. Consider the memory latency in cycles M to be defined as the maximum possible number of cycles between the assertion of `MEM_REQ` before the `MEM_ACK` response is asserted, then the minimum clock speed of the layer controller and memory is $\frac{6+M}{32}$ of the bus speed.

There is no upper bound on the layer speed relative to the bus. Layer controller clock speeds are generally configurable.

2 Document Revision History

- Revision 0.4 (r12314) – July 21, 2015
  - Remove dedicated CPU layer controller
  - Enhance generic layer controller interrupt interface
  - Move messages and register defintions to MPQ in MBus Specification
- Revision 0.3 (r7649) – Apr 18, 2013
  - Add functional unit addresses
  - Change addressing to reflect new prefix-style addressing
  - Update power-gating details
  - Add Wire Controller
  - Add Interrupt Controller
  - Add Channel 3 Information
  - Update layer controller information to match actual design
  - Use the word *Reserved* in the programmer model
- Revision 0.2 (r3194) – Mar 4, 2013
  - Add power-gating information
  - Add `RX_FAIL` signal
  - Solidify Interrupt TODO
  - Add `PRIORITY` signal
- Revision 0.1 (r2847) – Jan 22, 2013
  - Initial revision