12. Addressing

This clause describes how addresses are transformed between a slave's memory map and a master's address space. Clause 13 also describes how to determine which bits of the memory map are visible in the master's address space.

Indirect interfaces and the associated indirectly accessible memory map do not directly add to the address map. Rather, each indirect interface requires a separate address map calculation. Address map calculation for indirect interfaces is the same as a slave bus interface. The indirect address field and indirect data field are analogous to the logical address and data ports of a slave bus interface.

NOTE—In both Clause 12 and Clause 13, *equation variables* are denoted with letters offset in parenthesis, e.g., `(b)`, `(d)`, etc; whereas, the equations themselves are numbered (and offset in parenthesis), e.g., `(1)`, `(31)`, etc. Any subsequent references are shown as superscripted (and enclosed in parenthesis), e.g., $\texttt{address\_offset}^{(a)}$ or $\texttt{item\_width}^{(4)}$.

12.1 Calculating the bit address of a bit in a memory map

A *memory map* consists of a set of address blocks, subspace maps, and banks containing further address blocks, subspace maps, and banks (to any number of levels). To calculate the address of a bit within an address block or subspace map relative to the containing memory map, its bit address needs to be calculated relative to its parent. If that parent is a bank, how that bank modifies the address needs to be calculated first, and then continue working up the bank structure until the memory map is reached. To do so, the following formulas apply.

— For a bit in an address block directly in a memory map:

$$address\_offset = offset\ in\ address\ unit\ bits \tag{a}$$

The *address_offset* describes the offset in address-unit-bits. In IP-XACT, the following items' offset are described in address-unit-bits: addressBlock base-address, register and register-file addressOffset, bank baseAddress, and subspaceMap baseAddress.

$$bit\_offset = offset\ in\ bits \tag{b}$$

The *bit_offset* describes the offset in bits. In IP-XACT, the bitOffset for fields is described in bits.

$$addressBlock\_bit\_address = ((address\_offset^{(a)} + addressBlock.baseAddress) \times memoryMap.addressUnitBits) + bit\_offset^{(b)} \tag{1}$$

— For a bit in a subspace map:

$$subspaceMap\_bit\_address = ((address\_offset^{(a)} + subspaceMap.baseAddress) \times addressSpace.addressUnitBits) + bit\_offset^{(b)} \tag{2}$$

However, the following formulas need to be used on any containing banks.

a) For an item (bank, subspace map, or address block) within a serial bank:

$$container\_addressUnitBits := memoryMap.addressUnitBits \mid addressSpace.addressUnitBits \tag{3}$$

$$item\_width := addressBlock\_width^{(14)} \mid subspaceMap\_width^{(15)} \mid bank\_width(^{(16)} \tag{4}$$

$$item\_range := addressBlock\_range^{(8)} \mid subspaceMap\_range^{(9)} \mid bank\_range^{(11)} \tag{5}$$

$$item\_rows = ceiling((item\_range^{(5)} \times container\_addressUnitBits^{(3)}) / item\_width^{(4)}) \tag{6}$$

The effective range of an item is its range rounded up to the nearest complete row:

$$item\_effective\_range = item\_rows^{(6)} \times item\_width^{(4)} \qquad (7)$$

The range of an item is calculated depending on its type:

1) For an address block the range is the value of the range subelement;

$$addressBlock\_range = addressBlock.range \qquad (8)$$

2) For a subspace map which references a segment, the range is the value of the segment's range elements; for other subspace maps, the range is the value of the address space's range subelement, then the range is normalized by multiplying it with the address space's address unit bits and then divided by the subspace map's memory map address unit bits;

$$subspaceMap\_range = (addressSpace\_range^{(10)} \times addressSpace.addressUnitBits) / memoryMap.addressUnitBits \qquad (9)$$

$$addressSpace\_range := addressSpace.range \mid addressSpace.segment.range \qquad (10)$$

3) For a bank the range is dependent on its alignment;

$$bank\_range := serial\_bank\_range^{(12)} \mid parallel\_bank\_range^{(13)} \qquad (11)$$

4) For a serial bank, the range is the sum of the effective ranges of the subitems;

$$serial\_bank\_range = \sum_{i=0}^{n-1} item\_effective\_range_{[i]}{}^{(7)} \qquad (12)$$

5) For a parallel bank, the range is the (largest `item_rows` of all the subitems) x (`bank_width`/`addressUnitBits`);

$$parallel\_bank\_range = max(item\_rows^{(6)}[0], ..., item\_rows^{(6)}[n-1]) \times parallel\_bank\_width(^{(18)} / container\_addressUnitBits^{(3)} \qquad (13)$$

(i.e., the effective range of an item is its range rounded up to the nearest complete row)

The width of an item is calculated depending on its type:

6) For an address block, the width is defined as the value of the **width** subelement;

$$addressBlock\_width = addressBlock.width \qquad (14)$$

7) For a subspace map, the width is the width of the address space of the referenced bus interface;

$$subspaceMap\_width = addressSpace.width \qquad (15)$$

8) For a serial bank, the width is the width of the widest subitem;

$$bank\_width := serial\_bank\_width^{(17)} \mid parallel\_bank\_width^{(18)} \qquad (16)$$

$$serial\_bank\_width = max(item\_width^{(4)}[0], ..., item\_width^{(4)}[n-1]) \qquad (17)$$

9) For a parallel bank, the width is the sum of the widths of the subitems.

$$parallel\_bank\_width = \sum_{i=0}^{n-1} item\_width_{[i]}{}^{(4)} \qquad (18)$$

b) For an offset within item n in a serial bank:

$$serial\_bank\_bit\_address = bit\_offset^{(b)} + \left(\sum_{i=0}^{n-1} item\_effective\_range_{[i]}{}^{(7)} \times container\_addressUnitBits^{(3)} \right) \qquad (19)$$

c) For a bit within item n in a parallel bank containing m items:

$$bit\_offset\_in\_row = bit\_offset^{(b)} \bmod item\_width^{(4)}[n] + \sum_{i=0}^{n-1} item\_width_{[i]}{}^{(4)} \qquad (20)$$

$$row\_bit\_offset = \sum_{i=0}^{n-1} item\_width_{[i]}{}^{(4)} \times (bit\_offset^{(b)} / item\_width^{(4)}[n]) \qquad (21)$$

$$parallel\_bank\_bit\_address = row\_bit\_offset^{(21)} + bit\_offset\_in\_row^{(20)} \qquad (22)$$

Once the bit address within a top-level bank has been calculated, the bit address within the memory map can be derived from the following formula:

$$bank\_bit\_address := parallel\_bank\_bit\_address^{(22)} \mid serial\_bank\_bit\_address^{(19)} \qquad (23)$$

$$memory\_map\_bit\_address = address\_space\_bit\_address = block\_bit\_address^{(25)} + (item.baseAddress \times container\_addressUnitBits^{(3)}) \qquad (24)$$

$$block\_bit\_address := subspaceMap\_bit\_address^{(2)} \mid addressBlock\_bit\_address^{(1)} \mid bank\_bit\_address^{(23)} \qquad (25)$$

12.2 Calculating the bus address at the slave bus interface

The bus address of a bit at the slave bus interface can be derived from the following formulas:

$$slave\_bus\_address = memory\_map\_bit\_address^{(24)} / slave.bitsInLau \qquad (26)$$

On a bus, the bit offset gives the offset within the LAU of the bit using the following formula:

$$slave\_bit\_offset = memory\_map\_bit\_address^{(24)} \bmod slave.bitsInLau \qquad (27)$$

12.3 Calculating the address at the indirect interface

The address of a bit at an indirect interface can be derived from the following formula:

$$indirect\_interface\_bus\_address = memory\_map\_bit\_address^{(24)} / indirect\_interface.bitsInLau \qquad (28)$$

The bit offset gives the offset within the LAU of the bit using the following formula:

$$indirect\_interface\_bit\_offset = memory\_map\_bit\_address^{(24)} \bmod indirect\_interface.bitsInLau \quad (29)$$

12.4 Address modifications of a channel

The address at the mirrored slave interface can be derived from the following formula:

$$mirrored\_slave\_bus\_address = (slave\_bus\_address^{(26)} \times slave.bitsInLau) / mirroredSlave.bitsInLau \quad (30)$$

This is then modified by the remap address:

$$mirrored\_slave\_row\_address = mirrored\_slave\_bus\_address^{(30)} + (mirroredSlave.baseAddress.remapAddress) \quad (31)$$

where *remapAddress* is the remap address for the current state of the channel.

$$mirrored\_master\_bus\_address = (mirrored\_slave\_row\_address^{(31)} \times mirroredSlave.bitsInLau) / mirroredMaster.bitsInLau \quad (32)$$

12.5 Addressing in the master

The bus address at the master bus interface can be derived from the following formula:

$$master\_bus\_address = (mirrored\_master\_bus\_address^{(32)} \times mirroredMaster.bitsInLau) / master.bitsInLau \quad (33)$$

This gives a bit address of

$$master\_bit\_address = master\_bus\_address^{(33)} \times master.bitsInLau \quad (34)$$

The bit address may then be converted to an addressing unit address and offset using the formulas:

$$addressSpace\_bus\_address = master\_bit\_address^{(34)} / addressSpace.addressUnitBits \quad (35)$$

$$addressSpace\_bit\_offset = master\_bit\_address^{(34)} \bmod addressSpace.addressUnitBits \quad (36)$$

12.6 Address translation in a bridge

The address at the master interface for a bridge can be derived from the following formulas:

a) The bus address at the master bus interface is:

$$bridge\_master\_bus\_address = slave\_bus\_address^{(26)} \quad (37)$$

This gives a bit address of

$$bridge\_master\_bit\_address = bridge\_master\_bus\_address^{(37)} \times bridge\_master.bitsInLau + bridge\_master.addressSpaceRef.baseAddress \times addressSpace.addressUnitBits \quad (38)$$

The master bit address (also equal to the address space bit address) may be converted to an addressing unit address and offset of the *addressSpace* using the formulas:

$$bridge\_address\_space\_address = bridge\_master\_bit\_address^{(38)} / addressSpace.addressUnitBits \quad (39)$$

bridge_address_space_offset = bridge_master_bit_address$^{(38)}$ *mod addressSpace.addressUnitBits* *(40)*

b) The bit address may also be converted to the address of the bridged slave interface by using the following formulas:

1) For a transparent bridge:

bridge_slave_address = bridge_address_space_address$^{(39)}$ x *addressSpace.addressUnitBits / bridge_slave.bitsInLau* *(41)*

2) For an opaque bridge:

bridge_slave_address = (((bridge_address_space_address$^{(39)}$ *– segment.addressOffset)* x *addressSpace.addressUnitBits) / bridge_slave.bitsInLau) + slave_bus_address*$^{(26)}$ *(42)*

c) When an indirect interface is bridging to a master, the bit address may also be converted to the address of the indirect interface using the following formulas:

1) For a transparent bridge:

bridge_indirect_interface_address = (bridge_address_space_address$^{(39)}$ x *addressSpace.addressUnitBits) / bridge_indirect_interface.bitsInLau* *(43)*

2) For an opaque bridge:

bridge_indirect_interface_address = (((bridge_address_space_address$^{(39)}$ *- segment.addressOffset)* x *addressSpace.addressUnitBits) / bridge_indirect_interface.bitsInLau) + slave_bus_address*$^{(26)}$ *(44)*

13. Data visibility

The addressing descriptions in Clause 12 presume each bus interface only maps a single logical address port (a port with an **isAddress** qualifier) and a single logical data port (a port with an **isData** data qualifier). See also: 5.6 and 5.9.

If a bus interface maps more than one address or data port, then each combination of address and data ports implies a separate addressing and data visibility calculation. To calculate the address map for a particular type of transaction, the data and address ports that transaction uses need to be known first.

The most common case for multiple data ports in a single bus interface is where there are separate read and write data ports; however, their relevant properties of the read and write data ports are typically identical—giving identical read and write address maps.

13.1 Mapped address bits mask

The mapped address bits need to be taken into account when calculating the data visibility to the interface by deriving a mask from the set of address bits mapped in the interface. This mask is '`bitwise and`ed' with the `bus_address`.

interface_mapped_address_bits = a mask derived from the set of address bits mapped in the interface (c)

interface_bus_address := slave_bus_address$^{(26)}$ | *mirrored_slave_bus_address*$^{(30)}$ | *mirrored_master_bus_address*$^{(32)}$ | *master_bus_address*$^{(33)}$ | *bridge_master_bus_address*$^{(37)}$ *(45)*

interface_visible_bus_address = interface_bus_address$^{(45)}$ *& interface_mapped_address_bits*$^{(c)}$ *(46)*

13.2 Address modifications of an interconnection

The bus address is carried between adjacent bus interfaces (slave and mirrored slave, master and mirrored master, or master and slave) on the bus's **isAddress** logical port. If this port is a wire port, the address is always carried as parallel bits with the least significant bit of the address on logical bit `0` of the port. The interconnection can modify the address in two ways:

a) If some address bits are not connected, addresses with those bits set are not accessible from the master.
    1) Examine the logical vectors in the port maps to determine which address bits are connected.
    2) Transactional ports always carry all address bits across the interconnection.

b) If the value of **bitsInLau** differs on the two sides of the interconnection, the interpretation of the address as a bit address can vary by the ratio of the interfaces' **bitsInLau**. This, however, does not alter the actual bus address.

13.3 Bit steering in a channel

How addresses are modified within a channel depends on the value of **bitSteering** in the mirrored slave interface. It also depends on the relative width of the mirrored master and mirrored slave data ports, where this width is defined to be the total number of bits of the logical data port that are mapped in the bus interface. If `bitSteering` is `on`, or the slave is wider than or the same width as the master, the addresses are simply modified to take into account any change in **bitsInLau** between the mirrored slave and the mirrored master, as shown in the following formula:

mirrored_master_steering_on_visible_bus_address = mirrored_master_bus_address$^{(32)}$ *& mirrored_master_mapped_address_bits*$^{(c)}$ *(47)*

If `bitSteering` is `off` and the mirrored slave is narrower than the mirrored master, the address is adapted so all locations in the slave's memory map are visible:

mirroredMaster_width = relative width of the dataport of the mirrored master interface *(d)*

mirroredSlave_width = relative width of the dataport of the mirrored slave interface *(e)*

mirrored_slave_bit_address = mirrored_slave_row_address$^{(31)}$ x *mirroredSlave.bitsInLau* *(48)*

mirrored_master_bit_address = mirrored_slave_bit_address$^{(48)}$ *mod mirroredSlave_width*$^{(e)}$ + *((mirrored_slave_bit_address*$^{(48)}$ */ mirroredSlave_width*$^{(e)}$*)* x *mirroredMaster_width*$^{(d)}$*)* *(49)*

mirrored_master_steering_off_visible_bus_address = mirrored_master_bit_address$^{(49)}$ */ mirroredMaster.bitsInLau & mirrored_master_mapped_address_bits*$^{(c)}$ *(50)*

Finally, **bitSteering** has a different meaning in a mirrored slave interface than in a master or slave interface. In a master or slave interface, it means the component shall modify which bit lanes are used for data when accessing narrow devices. In a mirrored slave interface, it means the addresses from a mirrored master interface are not modified for transfers to a narrower mirrored slave data port.

13.4 Visibility of bits

A bit in the slave's memory map is visible in the master's address space if:

— it is in an address range visible to the master;

— the master and slave agree on which bit lane the bit should appear and this bit lane is connected between the master and the slave.

13.4.1 Visible address ranges

Two conditions need to be fulfilled for an address in the slave to be visible to the master.

a) The address at the mirrored slave shall be within the range supported by the mirrored slave interface:

$$mirrored\_slave\_visible\_bus\_address < mirroredSlave.baseAddress.range \qquad (51)$$

b) The address in the address space shall be within the range supported by the master address space for that bus interface:

$$0 <= master\_bit\_address^{(34)} < addressSpace.range \times addressSpace.addressUnitBits \qquad (52)$$

13.4.2 Bit lanes in memory maps

The local bit lane of a bit in an address block is:

$$address\_block\_bit\_lane = addressBlock.bit\_offset^{(b)} \text{ mod } addressBlock.width \qquad (53)$$

Similarly, in a subspace map the bit lane is:

$$subspace\_map\_bit\_lane = subspaceMap.bit\_offset^{(b)} \text{ mod } addressSpace.width \qquad (54)$$

where the *addressSpace.width* is the width of the address space of the referenced master bus interface.

If the address block or subspace map is at the top-level of the memory map or only within serial banks, the bit lane in the memory map is the local bit lane.

$$local\_bit\_lane := address\_block\_bit\_lane^{(53)} \mid subspace\_map\_bit\_lane^{(54)} \qquad (55)$$

If it is item *n* in a parallel bank, then:

$$bank\_bit\_lane = \sum_{i=0}^{n-1} item\_width_{[i]}^{(4)} + local\_bit\_lane^{(55)} \qquad (56)$$

If it is in multiple hierarchical parallel banks, this formula is applied at each higher level with the lower-level *bank_bit_lane* replacing *local_bit_lane*.

The bit lane in the memory map is the top-level *bank_bit_lane*.

13.4.3 Bit lanes in address spaces

The bit lane in an address space can be derived from the following formula:

$$address\_space\_bit\_address = master\_bit\_address^{(34)} \qquad (57)$$

$$address\_space\_bit\_lane = address\_space\_bit\_address^{(57)} \bmod addressSpace.width \qquad (58)$$

13.4.4 Bit lanes in bus interfaces

In a bus interface, the logical bit numbers of the data port carry the corresponding bit lanes. For example, if a slave bus interface has a data port with a logical vector of `[15:8]`, this port can access bit lanes `15` to `8` of the memory map and logical bit lanes `15` to `8` in the connected mirrored slave or master interface.

13.4.5 Bit lanes in channels

All bus interfaces on a channel shall use the same logical numbering of data port bits. This means data bits cannot be moved between bit lanes in a channel by giving the mirrored bus interfaces different logical to physical mappings on their data ports.

13.4.6 Bit steering in masters and slaves

Bit steering only takes effect when the master and the slave have data ports of different widths. If they do and bit steering is enabled (i.e., `bitSteering` is `on` in the master or slave interface) for the bus interface with the wider data port, then this data port shall move its copy of output data to the correct bit lanes for the narrower port and read its input data from the correct bit lanes for the narrower port.

If bit steering is disabled in the wider port, the master can only access data at a particular address when the bit lane for that address in the address space is connected (through the bus interfaces and a channel) to the bit lane for the corresponding address in the memory map.

The following also apply.

— The **bitSteering** value has a different meaning in mirrored slaves. See 12.2.

— Some buses with bit steering may only support certain data port widths. Describing which widths are supported is outside the scope of IP-XACT.

— Bit steering allows software or hardware away from the bus interface to work without knowing the width of devices on the far side of the bus. To provide this functionality, a bus supporting bit steering normally gives the same address bits to all devices, irrespective of their widths, and does not adapt addresses to the width of the slave bus interfaces (i.e., `bitSteering` is `on` in the mirrored slave bus interfaces). Thus, a non-bitsteering master on such a bus only has access to some of the memory rows of narrower slaves.